Project 5: Hash Tables and Its Applications

Due: Tues, Nov 21

Objectives: Understand and get familiar with the hash table data structure, along with its application in managing user accounts.

Task: Implement a hash table ADT and other supporting user interfaces; develop a simple password server program.

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. The two files `hashtable.h` and `hashtable.hpp` will be provided to you, which contain some helpful functions that you will need to use in developing the hash table class template.
- 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 & k)**: check if key k is in the hash table.
- **bool match(const std::pair<K, V> &kv)** : 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 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 semicolon character (:) (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 &k)**: 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, `passserver.h` and `passserver.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:

  1. **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).
  2. **~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.

3. **bool load(const char *filename)**: load a password file into the HashTable object. Each line contains a pair of username and encrypted password.
4. **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.
5. **bool addUser(std::pair<string, string> && kv):** move version of addUser.
6. **bool removeUser(const string & k)**: delete an existing user with username k.
7. **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).
8. **bool find(const string & user):** check if a user exists (if user is in the hash table).
9. **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.
10. **size_t size():** return the size of the HashTable (the number of username/password pairs in the table).
11. **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.x program):

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

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 `proj5.cpp`.

- A partial implementation of proj5.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 `proj5.x` 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

Extra-credit (10 points)

You may submit an alternative version to your program named sproj5.cpp, in which the program hides the user's entries whenever the user keys in a password or new password.

- Do not use the getpass() function, which is obsolete.

Provided Partial Code

The following partial code has been provided to you.

1. hashtable.h: partial implementation
2. hashtable.hpp: partial implementation
3. proj5.cpp: driver program, partial implementation.
4. proj5.x : sample executable for linprog.cs.fsu.edu
5. sproj5.x: sample executable with hidden password entries for linprog.cs.fsu.edu
6. test1: sample test case (which contains the commands that a user will type). You can redirect it to proj5.x as "proj5.x < test1". Results will save in the file "outtest1"
7. scrypt.cpp: sample program to use crypt() to encrypt password.
8. scrypt.x: executable code of scrypt.cpp.

Deliverables

1. Your implementation must be entirely contained in the following files, which MUST be named in the same way.

    1. hashtable.h
    2. hashtable.hpp
    3. passserver.h
    4. passserver.cpp
    5. proj5.cpp
    6. sproj5.cpp (optional, for extra-credit)
    7. makefile

2. Submit all the files in a tar file via the blackboard system. If you have implemented the extra-points version, please indicate so when you submit your homework.

3. 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(s) must be named proj5.x and sproj5.x (for extra-credit option).

4. The interaction and output (including error messages) of your client's executable(s) must behave in the same manner as the distributed proj5.x and sproj5.x (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(s) and the output file(s) created by the distributed executable(s).

You can also run the "proj5.x" file directly from linprog from my account, with this command:

```
~myers/dsprog/proj5.x
```

Points will be deducted for not complying with these requirements.