CS 3100 – *Data Structures and Algorithms*

Project #6 - Hash Table Indexing

Learning Objectives

- Implement a data structure to meet given specifications
- Design, implement, and use a closed hash table data structure
- Use a hash table as an index into a separate data store

Overview

Your task for this assignment is to implement a database of student records. Your database will consist of two parts: an unsorted vector of student records, and an index of student IDs implemented as a closed hash table.

The Record Store

Student records will be stored in an unsorted vector of class Record. The definition of class Record will be provided for you in the file `Record.h`.

The HashTable class

Your hash table should be implemented as an array of `MAXHASH` objects of class Slot. A Slot contains an integer key, and an integer index. The definition of class Slot will be provided for you in the file `Slot.h`. In order to more easily test collision resolution, and error handling for full HashTables we will use a very small table for testing. The value of `MAXHASH` should initially be `#defined` to 20.

To implement the hash table, you should create a class called HashTable, implemented in the files HashTable.h and HashTable.cpp. Your class should support the following operations:

- **`bool HashTable::insert(int key, int index, int& collisions)`** – Insert a new key/index pair into the table. Duplicate keys are not allowed. This function should return **`true`** if the key/index pair is successfully inserted into the hash table, and **`false`** if the pair could not be inserted (for example, due to a duplicate key already found in the table, or if the HashTable is full). If the insertion is successful, the number of collisions occuring during the insert operation should be stored in **`collisions`**.

- **`bool HashTable::remove(int key)`** – If there is a record with the given key in the hash table, it is deleted (changing the slot type to `emptyAfterRemoval` and the function returns **`true`**; otherwise the function returns **`false`**.

- **`bool HashTable::find(int key, int& index, int& collisions)`** – If a record with the given key is found in the hash table, the function returns **`true`**, a copy of the index is returned in `index`, and the number of collisions during the find operation is reported in `collisions`. Otherwise the function returns **`false`**.

- **`float HashTable::alpha()`** - returns the current loading factor, α, of the hash table.

- Your HashTable class should also overload `operator<<` such that `cout << myHashTable;` prints all the information in the table in the following format:

```
HashTable contents:
HashTable Slot 9: Key = 112233, Index = 2
HashTable Slot 4: Key = 223344, Index = 0
HashTable Slot 2: Key = 334455, Index = 1
```

Note: *operator<< should not print empty or tombstone slots.*

The hash and probe functions

Because this is a closed hash, you will need both a hash function and a probe function. A hash function will be provided for you in the file `hashfunction.h`. Your hash must use **pseudo-random probing** to resolve collisions.

The Database class

Your database consists of two parts, the record store and the hash table. In other words, the private data members of your Database class should include the following:

```
class Database {
private:
    HashTable indexTable;
    vector<Record> recordStore;
}
```

Your Database class should support the following operations:

- **`bool Database::insert(const Record& newRecord, int& collisions)`** – Insert a new student record into the database. This function should:
    - Check to make sure the UID is not already in the HashTable.
    - If not, insert the Record into the recordStore, and
    - Insert the UID and the slot the Record occupies in the recordStore into the HashTable.

  This function should return **`true`** if the record is successfully inserted into the database, and **`false`** if the record could not be inserted (for example, due to a duplicate key already found in the HashTable, or if the HashTable is full). If the insertion is successful, the number of collisions occuring during the HashTable::insert() operation should be returned in **`collisions`**.

- **`bool Database::remove(int key)`** – If there is a record with the given key in the Database, it is deleted and the function returns **`true`**; otherwise the function returns **`false`**. Deleting from the database removes both the hash table (index) entry, and the record store (vector) entry. **You must delete records from the recordStore in a way that does not waste memory, and does not break the index**. One way to do this is to copy the last record in the vector into the position holding the record to be deleted, and then use pop_back() to remove the last element of the vector.

- **`bool Database::find(int uid, Record& foundRecord, int& collisions)`** – If a record with the given uid is found in the Database, the function returns **`true`**, the number of collisions during the search is returned in `collisions`, and a copy of the record is stored in **`foundRecord`**. Otherwise the function returns **`false`**.

- **`float Database::alpha()`** - returns the current loading factor, α, of the Database's hash table (This function can simply call **`HashTable::alpha()`**.

- Your Database should also overload **`operator<<`** such that **`cout << myDatabase;`** prints all the information in the database in the following format:

```
Database contents:
HashTable Slot: 9, recordStore slot 2 -- Gates, Bill (U00112233): Senior
HashTable Slot: 4, recordStore slot 0 -- Cook, Tim (U00223344): Sopohomore
HashTable Slot: 2, recordStore slot 1 -- Zuckerberg, Mark (U00334455): Freshman
```

Note: *operator<< should not print empty or tombstone slots.*

Main program

You should use your student database in a main program that allows a user to insert, search, and delete from the database. Searching the database by UID should be done using the hash table, and should report the number of collisions encountered during the search.

Example program operation

```
Would you like to (I)nsert or (D)elete a record, (S)earch for a record,
  (P)rint the database, or (Q)uit?
Enter action:  I
Inserting a new record.
Last name:  Doe
First name:  Jane
UID: 1234
Year:  Junior
Record inserted (0 collisions during insert).

Would you like to (I)nsert or (D)elete a record, (S)earch for a record,
  (P)rint the database, or (Q)uit?
Enter action:  S
Enter UID to search for: 1234
Searching... record found (3 collisions during search)
----------------------------
Last name:  Doe
First name:  Jane
UID: 1234
Year:  Junior
```

```
----------------------------

Would you like to (I)nsert or (D)elete a record, (S)earch for a record,
  (P)rint the database, or (Q)uit?
Enter action:  S
Enter UID to search for: 2345
Searching... record not found

Would you like to (I)nsert or (D)elete a record, (S)earch for a record,
  (P)rint the database, or (Q)uit?
Enter action:  P

Database contents:
HashTable Slot: 9, recordStore slot 2 -- Doe, Jane (U00001234): Junior
HashTable Slot: 4, recordStore slot 0 -- Cook, Tim (U00223344): Sopohomore
HashTable Slot: 2, recordStore slot 1 -- Zuckerberg, Mark (U00334455): Freshman

Would you like to (I)nsert or (D)elete a record, (S)earch for a record,
  (P)rint the database, or (Q)uit?
Enter action:  Q
Exiting.
```

Turn in and Grading

Please zip your entire project directory into a single file called Project4.zip.

This project is worth 50 points, distributed as follows:

Task	Points
`HashTable::insert` stores key/index pairs in the hash table using appropriate hashing and collision resolution, and correctly rejects duplicate keys and reports correct collision counts. Insert correctly re-uses space from previously-deleted records.	5
`Database::insert` stores records in the recordStore and hashes the UID and recordStore slot in the HashTable. Insert correctly rejects duplicate keys and reports correct collision counts. Insert correctly re-uses space from previously-deleted records. Insert correctly returns false when the index (hash table) is full.	5
`HashTable::find` correctly finds records in the hash table using appropriate hashing and collision resolution, and returns true and places the index associated with the search key into `index` if the key is found, or returns **`false`** when the requested key is not present in the hash table.	5
`Database::find` uses HashTable::find to find UIDs in the hash table. Returns true and places the appropriate record in foundRecord if the UID is found, or returns **`false`** when the requested UID is not present in the Database.	5
`HashTable::remove` correctly finds and deletes records from the table, without interfering with subsequent search or insert operations.	5
`Database::remove` correctly finds and deletes records from the database, without interfering with subsequent search or insert operations. Also, removing records does not waste memory in the	5

record store.	
`Database::alpha` correctly calculates and returns the load factor of the database's hashTable.	5
`operator<<` is correctly overloaded for class HashTable and class Database as described above.	5
Code is well organized, well documented, and properly formatted. Variable names are clear, and readable. Classes are declared and implemented in seperate (.cpp and .h) files.	5
Appropriate use of public and private class member data. No global variables or unnecessary member variables. Efficient and well-designed code. No memory leaks when creating and deleting hashTables or databases.	5