CMSC 341 - Project 4: Mass Vaccination Site Application - Spring 2021

Due: Tuesday, May 11, before 9:00 pm

- Please note: There is ***no late submission*** for this project. You must turn it in on-time to receive credit.

- Please ***do not rename the files***. Linux file names are case sensitive.
- Please ***do not develop code*** in the shared directory.

Addenda

- 04/23/21 - Clarification: a sample algorithm is added to the testing section.
- 04/23/21 - Clarification: using STL libraries is not allowed in the HashTable class. We can use STL libraries in mytest.cpp for the testing purposes.

Objectives

- To implement a hash table.
- To implement linear probing to manage hash collisions.
- To implement quadratic probing to manage hash collisions.

Introduction

We are planning to prepare a mass vaccination site. A part of the plan is developing a software application to manage the appointments. The operators of the site need to search for people who have booked an appointment for receiving vaccine. People can book a time slot, and after receiving the vaccine the names will be removed from the system. Then, the main operations in the system are insertion, search, and deletion. Since the efficiency of operations is extremely important, we have decided to use a hash table to manage the information. Your task is to develop this part of the system. The keys are names, we know that this can cause collisions and clustering in a hash table, since there are many common names. To increase efficiency, we use two different collision handling policies. The application can change the table size, and the collision handling policy based on some specific criteria. When a change is required, we need to rehash the entire table.

Hash Table

A hash table is a data structure which implements the map ADT. It stores the pairs of key-value. The key determines the index number of an array cell (bucket) in which the value will be stored. A hash function calculates the index number. If there is more than one value for a key in the data set, the hash table uses a collision handling scheme to find another cell for storing the new value.

Collision handling

Linear Probing

When we try to find an index in the hash table for a key and we find out that the determined bucket is taken, we need to find another bucket to store the key. One way of finding an empty bucket is using linear probing. In this project we use the following equation to find the next bucket, in which i = 0,1,2,3 ....

index = (Hash(key) + i) % TableSize

Quadratic Probing

When we try to find an index in the hash table for a key and we find out that the determined bucket is taken, we need to find another bucket to store the key. In a linear probing we store the information in the next available bucket. This can cause clustering, i.e. having many data points in a row. Clustering reduces the serach efficiency. Using quadratic probing instead of storing data in the next available bucket, we calculate a jump step to find the next bucket for storage. In this project we use the following equation to find the next bucket, in which i = 0,1,2,3 ....

index = ((Hash(key) % TableSize) + (i x i)) % TableSize

Assignment

Your assignment is to complete the class HashTable and the appropriate test functions.

The application starts with a hash table which is using linear probing. After certain criteria appearing it will switch to the table with quadratic probing and transfers all data from the current table to the new one. It continues to use the table with quadratic probing until certain criteria appear, then the application switches back to the table with linear probing and transfers all data to that table. In fact, at any time there is only one table used for all operations and the application can switch to another table once certain criteria are met.

After an insertion, if the load factor becomes greater than 0.5, we need to rehash to a new hash table and switch probing methods (linear becomes quadratic, or quadratic becomes linear). The capacity of the new table would be the smallest prime number greater than 4 times the current number of data points. The current number of data points is total number of occupied buckets minus total number of deleted buckets.

After a deletion, if the number of deleted buckets is more than 25 percent of the total number of occupied buckets, we need to rehash to a new table and switch probing methods (linear becomes quadratic, or quadratic becomes linear). The capacity of the new table would be the smallest prime number greater than 4 times the current number of data points. The current number of data points is total number of occupied buckets minus total number of deleted buckets.

During a rehashing process the deleted buckets will be removed from the system permanently. They will not be transferred to the new table.

For this project, you are provided with the following files:

- hash.h – header file for the HashTable class.
- hash.cpp – the template file for the HashTable class. Complete your HashTable implementation in this file.
- person.h – header file for the Person class.
- person.cpp – implementation of the Person class. You should not need to modify this class.
- driver.cpp – A sample driver program.
- driver.txt – A sample output produced by driver.cpp.

Specifications

HashTable Class

The HashTable class uses the Person class. It has a member variable to store a pointer to a hash function. It also has a member variable to store a pointer to an array of Person objects. This array is the hash table and the name of the Person object is used as the key for hashing purposes.

HashTable::HashTable(unsigned size, hash_fn hash)	The alternative constructor, size is an unsigned integer to specify the length of hash table, and hash is a function pointer to a hash function. The type of hash is defined in hash.h. The table size must be a prime number between MINPRIME and MAXPRIME. If the user passes a size less than MINPRIME, the capacity must be set to MINPRIME. If the user passes a size larger than MAXPRIME, the capacity must be set to MAXPRIME. If the user passes a non-prime number the capacity must be set to the smallest prime number greater than user's value.
HashTable::~HashTable()	Destructor, deallocates the memory.
bool HashTable::insert(Person person)	This function inserts an object into the hash table. The insertion index is determined by applying the hash function m_hash that is set in the HashTable constructor call and then reducing the output of the hash function modulo the table size. Hash collisions should be resolved using current probing policy which is set in m_currentProb. At the creation of HashTable object the policy is LINEAR. This policy can change during the execution based on meeting certain

	criteria. After every insertion we need to check for those criteria, and if it is required we need to rehash the entire table using the alternative probing policy. All member variables need to be reinitialized to their proper values. If the "person" object is inserted the function returns true, otherwise it returns false. A Person object can only be inserted once. The hash table does not contain duplicate objects. Please note, every Person object is a unique object carrying the person's name and the vaccine serial number. The person's name is the key which is used for hashing.
bool HashTable::remove(Person person)	This function removes a data point from the hash table. In a hash table we do not empty the bucket, we only tag it as deleted. To tag a removed bucket we assign the DELETED object to the bucket. The DELETED object is defined in hash.h. To find the bucket of the object we should use the current probing policy which is set in m_currentProb. The probing policy can change during the execution based on meeting certain criteria. After every deletion we need to check for those criteria, and if it is required we need to rehash the entire table using the alternative probing policy. All member variables need to be reinitialized to their proper values. If the "person" object is found and is deleted, the function returns true, otherwise it returns false.
unsigned HashTable::tableSize() const	This function returns the size of the hash table.
unsigned HashTable::numEntries() const	This function returns the number of elements in the hash table.
void HashTable::setProbingPolicy(probing prob)	At any time the operations are managed using a collision handling policy. We can change the collision handling policy. In this application we may use either linear probing or quadratic probing. A change in probing must enforce rehashing the data into a new table. Probing types are defined as enum type.
float HashTable::lambda() const	This function returns the load factor of the hash table. The load factor is the ratio of occupied buckets to the table size.
float HashTable::deletedRatio() const	This function returns the ratio of the deleted buckets to the total number of occupied buckets .
void HashTable::dump()	This function dumps the contents of the hash table in index order. It prints the contents of the hash table in array-index order.
int HashTable::findNextPrime(int current)	This function returns the smallest prime number greater than the passed argument "current". If "current" is less than or equal to MINPRIME, the function returns the next prime number after MINPRIME. If "current" is greater than or equal to MAXPRIME, the function returns MAXPRIME. In a hash table we'd like to use a table with prime size. Then, everytime we need to determine the size for a new table we use this function. **Note:** The implementation of this function is provided.
bool HashTable::isPrime(int number)	This function returns true if the passed argument "number" is a prime number, otherwise it returns false. **Note:** The implementation of this function is provided.

Additional Requirements

Requirement: Private helper functions may be added to the HashTable class; however, they must be declared in the private section of the class declaration.

Requirement: HashTable must implement a hash table in which each bucket is a Person object. The hashing key is the person's name which is accessible using Person::key().

Requirement: The hash table size and hash function are specified in the constructor call. The table must be dynamically allocated in the constructor. Please refer to the description of HashTable constructor for the details about the table size.

Requirement: Hash collisions must be resolved by either linear probing or quadratic probing depending on the current probing policy. The probing policies are defined as an enum type in hash.h.
The equation for quadratic probing is ***index = ((Hash(key) % TableSize) + (i x i)) % TableSize***.

Requirement: Hash indices must be computed by applying the hash function to the key value and reducing the hash output modulo the table size.

Requirement: The allocated memory to the hash table must be dynamically managed at execution time when there is rehashing.

Requirement: When rehashing, the deleted buckets will be removed from the system. No deleted bucket will be transferred to the new table.

Requirement: Here are the rules for lazy deletion,

- Treat deleted element as occupied when inserting.
- Treat deleted element as occupied when searching.

Requirement: No STL containers or additional libraries may be used in the HashTable class. STL containers can be used in mytest.cpp for testing purposes.

Testing

You must write your own, extensive test driver called mytest.cpp; this file contains your Tester class, all test functions, all test cases, hash function, and the main function. You can copy the hash function, and the random data generator code from the sample driver.cpp to your mytest.cpp. Your test driver program must compile with your HashTable class and it must run to completion.

Your driver file must be called ***mytest.cpp***; no other name is accepted for the driver program.

Following is a non-exhaustive list of tests to perform on your implementation. Please check the testing guidelines, it helps you to write effecive test cases.

Testing HashTable Class

- Test the insertion operation in the hash table. The following presents a sample algorithm to test the normal insertion operation:
  - There is some non-colliding data points in the hash table.
  - Insert multiple non-colliding keys.
  - Check whether they are inserted in the correct bucket (correct index).
  - Check whether the data size changes correctly.
- Test the remove operation in the hash table.
- Test linear probing is working correctly after insertions and removals.
- Test quadratic probing is working correctly after insertions and removals.
- Test the rehashing is triggered after insertions and removals.
- Test changing the probing method triggers a rehash.

Random Numbers for Testing

For testing purposes, we need data. Data can be written as fixed values or can be generated randomly. Writing fixed data values might be a tedious work since we need a large amount of data points. The approach for creating data will be your decision.

In the file driver.cpp there is a piece of code which generates pseudorandom numbers to be used as vaccine serial numbers. Since this code is using a fixed seed value, on the same machine it always generates the same sequence of numbers. That is why the numbers are called pseudorandom numbers, they are not real random numbers. Please note, the numbers are machine dependent, therefore, the numbers you see in the sample file driver.txt might be different from the numbers your machine generates.

Memory leaks and errors

- Run your test program in valgrind; check that there are no memory leaks or errors.
  Note: If valgrind finds memory errors, compile your code with the -g option to enable debugging support and then re-run valgrind with the -s and --track-origins=yes options. valgrind will show you the lines numbers where the errors are detected and can usually tell you which line is causing the error.
- Never ignore warnings. They are a major source of errors in a program.

What to Submit

- Please ***do not create sub-directories*** in the shared directory.
- Please ***clean up the shared directory*** after testing. Only the required files for project will remain in the shared directory. Development files such as precompiled gch, object files, your executable files, or vgcore dump files are large;these files consume disk quota in submission folders.

You must submit the following files to the proj4 directory.

- hash.h
- hash.cpp
- person.h
- person.cpp
- mytest.cpp

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
    cp hash.h hash.cpp person.h person.cpp mytest.cpp ~/cs341proj/proj4/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.

CMSC 341 - Spring 2021 CSEE | UMBC