Project #05: Hashing Illinois Specialized License Plates

Complete By: Saturday, November 16th @ 11:59pm
Assignment: Illinois license plate hashing
Policy: Individual work only, late work *is* accepted (see "Policy" section for more details)
Submission: via Gradescope --- limited to 12 submissions

Background

Like many states in the USA, the state of Illinois allows car owners to order specialized license plates, e.g. "ILUVUIC". In this assignment you're going to using hashing to efficiently process fines against cars with specialized license plates.

Rules for specialized license plates

There are a variety of formats for specialized Illinois license plates; your program needs to accept both *Personalized* and *Vanity* plates. Here are the formats:

```
//
// Personalized:
//   letters and numbers, with a space between the letters
//   and numbers.  Format: 1-5 letters plus 1..99 *OR*
//   6 letters plus 1..9
//
//   Examples: A 1, B 99, ZZZZZ 1, ABCDEF 3
//
// Vanity:
//   Format: 1-3 numbers *OR* 1-7 letters
//
//   Examples: 007, 1, 42, X, AAA, ZZZEFGH
//
```

Your hash function should return a valid index into the hash table if the license plate fits one of the formats above; if the license plate does not follow one of these formats, your hash function should return -1.

File input

The input to the program will come from a text file that contains N > 0 lines; N will be a multiple of 2. The input comes in the form of 2-line pairs: a ticket **fine** followed by a license **plate**. Here's an example input file:

```
20
A 1
35
B 99
100
ZZZZZ 1
80
A 1
250
ABCDEF 3
75
007
99
A 1234
```

The first two lines form the pair **(20, "A 1")**, which means the owner of license plate "A 1" received a ticket fine of $20. The fine is always an integer value, and the license plate is a string that may contain 1 or more blanks. Notice that later in the file, the owner of "A 1" received another fine for $80. Finally, notice that the last pair **(99, "A 1234")** is invalid because the license plate does not follow the formatting rules outlined earlier; this input pair should be ignored.

Since we are inputting strings that may contain blanks, we cannot use the **>>** operator. Instead, use the **getline** function to input both the fine and the license plate. Here's a skeleton input loop:

```
ifstream  infile(infilename);
.
.  // make sure file was opened successfully
.

string fine;
string plate;

getline(infile, fine);

while (!infile.eof())
{
   getline(infile, plate);

   .
   .  // process (fine, plate) pair:
   .

   getline(infile, fine);
}
```

File output

The program's file output contains M >= 0 lines, in alphabetical order by license plate, with one line for each valid license plate in the input. A line contains the license plate surrounded by "", followed by a space, followed by $, followed the sum of all fines received by that license plate. For example, given the input file shown earlier, the program should produce the following file output:

```
"007" $75
"A 1" $100
"ABCDEF 3" $250
"B 99" $35
"ZZZZZ 1" $100
```

There are 5 lines of output because the input file contains 5 valid license plates. Notice the lines are in alphabetical order by license plate, and the amount for plate "A 1" is $100 because that's the sum of the individual fines $20 and $80. The file output will be graded for correctness by Gradescope, so follow this format exactly.

To write to a file, you first create an **ofstream** object. Then use the **<<** operator, much like console output. Example:

```
ofstream outfile(outfilename);

outfile << "\"" << plate << "\"" << " $" << amount << endl;
```

Note that if the file does not exist, it will be created. If the file already exists, the contents will be deleted before the new contents are written.

Program behavior

The program starts by inputting 2 values from the console keyboard: the **hashtable size**, and the **base filename**. For example:

```
Enter hashtable size> 100000
Enter base filename> tickets1
```

Assume the hashtable size is a positive integer at least 10x larger than the # of valid license plates in the given input file. The input filename is the base filename with ".txt" appended; the output file name is the base filename with "-output.txt" appended. Example: given the base filename of "tickets1", this implies the input file is "tickets1.txt" and the output file is "tickets1-output.txt". While not required, it's helpful if the program outputs this information to the console, e.g.

```
Reading 'tickets1.txt'...
Sorting...
Writing 'tickets1-output.txt'...
```

The actual console output does not matter, it will not be graded for correctness. However, when the underlying hashtable goes out of scope, the destructor will automatically output some stats to the console. These stats *will* help us evaluate the quality of your implementation. For example, suppose the input to your program is the data shown on page 2 --- which contains 6 valid (fee, plate) pairs. If your hashing approach generates no collisions, the best possible stats for this input would be:

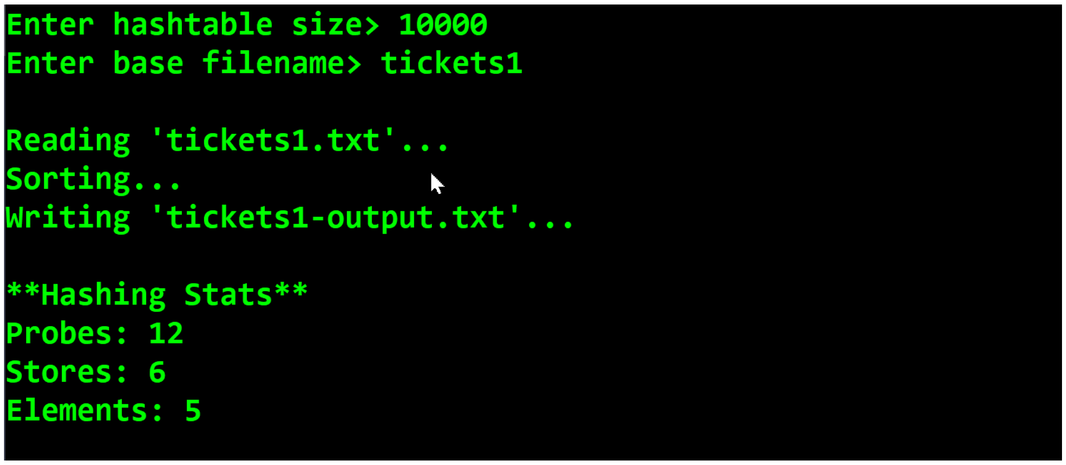
```
**Hashing Stats**
Probes: 6
Stores: 6
Elements: 5
```

The # of probes is the best indicator of how well your hash function is working. If the # of probes is 6, this means you probed the hashtable 6 times --- once per valid license plate, and encountered 0 collisions. Since there are 6 valid input pairs, each one requires a store to the hashtable --- hence 6 stores. Finally, there are only 5 elements in the hashtable because one of the input pairs (80, "A 1") is the same as an earlier license plate, which updates an existing element of the hashtable. What's a bad set of stats for this input data? What if your program outputs:

```
**Hashing Stats**
Probes: 32
Stores: 6
Elements: 5
```

This implies that you probed the hash table 32 times --- 32 probes for 6 different license plates means your hash function is generating lots of collisions, and thus you have to do lots of probes when you search, and lots of probes for a free space when you insert.

In general, if your hash function is working well, the # of collisions should be small, which means the # of probes should be roughly 2x the # of valid license plates in the input. Some of this depends on how you implement your search and insert functions. In my case, search requires 1 probe and insert requires 1 probe, so the best my program does is 2 probes per valid license plate. Here's the console output from my solution based on the input data from page 2, which is stored in the input file "tickets1.txt":



```
Enter hashtable size> 10000
Enter base filename> tickets1

Reading 'tickets1.txt'...
Sorting...
Writing 'tickets1-output.txt'...

**Hashing Stats**
Probes: 12
Stores: 6
Elements: 5
```

Again, the format of the console output does not matter for grading purposes; the stats will be automatically shown. The graded output will be the contents of "tickets1-output.txt", which is shown at the top of page 3.

Getting Started

A Codio project "**cs251-project05-hashing**" is provided with skeleton files: "main.cpp", "ILplates.h", "ILplates.cpp", and "hashtable.h". A good chunk of the main() program is being provided, in particular the processing of the input file and calls to the hashing functions. Your job will be to implement the hashing functions, along with the sorting and output of the results.

Requirements

Part #1 of your assignment is to implement a **Hash** function for specialized IL license plates, and then write **Search** and **Insert** functions based on linear probing to store and accumulate ticket fees for each license plate. In this approach, the key is the license plate, and the value is the total amount of fines against this license plate. The design for your hashing functions are given in the header file "ILplates.h"; your implementations will go in "ILplates.cpp". You cannot change the .h file --- for grading purposes, you must work within the constraints of the design as given:

```
class ILplates
{
private:
  hashtable<string, int>& HT;  // reference to underlying hashtable:

public:
  //
  // constructor
  //
  ILplates(hashtable<string, int>& ht)
    : HT(ht)
  { }

  int  Hash(string plate);
  int  Search(string plate);
  void Insert(string plate, int newValue);
};
```

See "ILplates.cpp" for specifics on each function. Part #2 of your assignment is to write a program in "main.cpp" to use your hashing functions to input license plate data, sum the fines per plate, sort the results by license plate, and output to a file. You must write the sort function yourself, you cannot use a built-in sort function. You can use whatever sort algorithm you want, even bubble or insertion sort. The only requirement is that you write the sort function yourself; sort the license plate strings using the natural C++ ordering as defined by the < operator.

If you look carefully at the "ILplates.h" header file shown above, you'll notice that the underlying hashtable is not an array --- but in fact an object of type **hashtable<string, int>**. The string is the license plate (the key), and the int is the total amount of fines for this license plate (the value). This object replaces an array of structures, and supports collisions via probing. Normally your Search and Insert functions would be accessing the hashtable array using HT[i].Key and HT[i].Value. Now you'll access the hashtable using **HT.Get(…)** and **HT.Set(…)**.

Why are we doing this? Two reasons. First, it's a good lesson in abstraction. The **hashtable<TKey, TValue>** class provides an abstraction of a hashtable that supports probing. Second, it allows us to collect stats about how well your Hash function and probing algorithm is working with regards to collisions. For this to work, your main() program needs to be structured as follows:

```
int main()
{
  int    N;        // = 10000;
  string basename; // = "tickets1";

  cout << "Enter hashtable size> ";
  cin >> N;

  hashtable<string, int>  ht(N);  // underlying hashtable
  ILplates  hashplates(ht);       // your hashing functions

  cout << "Enter base filename> ";
  cin >> basename;
  cout << endl;
```

Your main() program will now open the input file, and start hashing using the **hashplates** object; the **ht** object should not be used until it comes to do the sorting. When you need to sort, call ht.Keys() and ht.Values() to retrieve the keys and values from the hashtable --- the license plates and corresponding amounts of fines. Sort by key, keeping the values associated with the proper key. After you have sorted, output the keys and values to the output file.

For completeness, here's the contents of the "hashtable.h" header file, which defines the hashtable<TKey, TValue> abstraction. Implementation details are deleted to save space:

```
/*hashtable.h*/

//
// Implements a hashtable, providing Get and Set functions
// for accessing the underlying array.
//
// It is assumed that collisions are handled using probing.
// To support this, each location contains an "Empty" flag
// which is true if the array location is empty, and false
// if the location is in use.  Each locadtion also contains
// both the key and value, for resolving collisions.
//

#pragma once

#include <iostream>
#include <vector>
```

```
using namespace std;

template<typename TKey, typename TValue>
class hashtable
{
private:
  struct KeyValuePair
  {
    bool   Empty;
    TKey   Key;
    TValue Value;

    KeyValuePair()
    {
      Empty = true;
      Key   = {};   // default initializer
      Value = {};   // default initializer
    }
  };

  int             N;
  KeyValuePair*  Hashtable;  // array of (key, value) pairs

  int _Probes;
  int _Stores;


public:
  hashtable(int size)
  {
    N = size;
    Hashtable = new KeyValuePair[N];

    _Probes = 0;
    _Stores = 0;
  }

  virtual ~hashtable()
  {
    ...
  }

  //
  // size
  //
  // Returns the size N of the hash table.
  //
  int Size()
  {
    return N;
  }

  //
```

```
    // Get
    //
    // Gets the data from the given index of the hash table: the Empty
    // status (T/F), the key, and the value.  If Empty is true, the key
    // and value will be the default values in C++ for their types.  If
    // the index is outside the bounds of the hashtable, the behavior
    // of this function is undefined.
    //
    void Get(int index, bool& empty, TKey& key, TValue& value)
    { ... }

    //
    // Sets
    //
    // Sets the location of the hash table to contain the given key and
    // value; the "Empty" status is set to false since this location now
    // contains data.  Any existing data at this location is overwritten.
    // If the index is outside the bounds of the hashtable, the behavior
    // of this function is undefined.
    //
    void Set(int index, TKey key, TValue value)
    { ... }

    //
    // Keys
    //
    // Returns the keys in the hashtable, in order from 0..N-1.  Only
    // returns those keys where Empty is false.
    //
    vector<TKey> Keys()
    { ... }

    //
    // Values
    //
    // Returns the va.ues in the hashtable, in order from 0..N-1.  Only
    // returns those values where Empty is false.
    //
    vector<TValue> Values()
    { ... }

    //
    // Stats
    //
    // Returns some statistics about how the hashtable was used: # of
    // probes, # of stores, and # of elements currently in the table.
    //
    void Stats(int& probes, int& stores, int& elements)
    { ... }

  };
```

Program submission

When you submit your program for grading, we will test 2 things: your hashing functions in "ILplates.cpp", and your overall hashing program in "main.cpp". No other .cpp files will be compiled, and your implementation in "ILplates.cpp" must adhere to the design as declared in "ILplates.h". We will provide two Gradescope submissions: **Project 05-01 hashing** and **Project 05-02 program**. Submit "ILplates.cpp" to the first one, and both .cpp files to the second one.

Your work will receive a tentative grade for correctness, and then will undergo manual review for commenting, readability, approach, and adherence to requirements. Note that you'll be limited to a total of 12 submissions in each case. Commenting/readability/approach will count for 10% of project score. Adherence to requirements may count anywhere from 10% to 100%.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .