CPSC 221, Summer Term 2013
Programming Assignment 2

Date of Latest Revision to this Document:
- posted June 14, 2013

History of Non-Trivial Changes: *none.*

Part marks *will be awarded*, so even if you don't finish everything make sure that your code compiles on our undergrad Linux machines (these machines tend to have the same software releases, so as long as it works on one, you're OK), and make sure that your code has at least some of the desired functionality.

Objectives
- Gain more experience with C++, including classes.
- Work with hash functions, hashing principles, collision resolution policies, tombstones.
- Learn more about the implications of table size, hash function and collision resolution on the performance of a hash table.
- Work with random numbers.
- Measure performance (average number of probes).

This assignment has FOUR PARTS. Do each part in order so that you have at least something to submit for each section.

Note that you will create a `Hasher` class to handle the hashing related activities. We have created a framework for you in `hashDriver.cpp` to make thing easier. Note that you will be providing your own hash functions, `hashTableSize`, etc. Use the textbook, your class notes, and your own good judgment to determine the appropriate hash functions, modulus, etc.

Due Dates and Times
This assignment is due just *before* midnight at 11:59pm on Thursday, June 20. For handin purposes, the name of this assignment is 'pa2'. Why Thursday, June 20? That's the last day of classes. As always, we will not mark late submissions.

*****NOTE***** Our part-mark policy emphasizes the incremental coding approach. That is, do **not code large chunks of your program before testing**. Code in small increments and test **frequently**, ensuring that your code compiles and runs. This will make debugging **much easier**, and ensure that at almost any stage you have at least something you can turn in for partial marks.

Part 1: Generating the Data Files

Deliverables:

- Write a short, independent program `genData.cpp` that generates *n* lines containing key-value pairs and writes them to a file. (Later, you'll read this file into your driver program.)
- Specify the filename as a runtime argument. (eg: `genData 25 small.txt` will create a new file called "small.txt" with 25 records).
- Save any files that you generate and use for performance testing so you can compare improvements (if any) after making changes to your hash table.

How to generate test files:

To create the **key** (the string), generate 8 numbers between 0 and 25. These numbers will represent the letters to use for each position in the string. For example, generating [7, 1, 25, 4] yields the string "HBZE".

For the data **value** (the integer), just use the incremented values 1, 2, 3, ...*n,* corresponding to current data line number (these will serve the purpose of the placeholder "data" that we want to store). Here is an example of the first part of a generated file, for subsequent input into your driver program:

```
HBZEJKGA    1
RHJMIVTA    2
```

The significance of this randomized file is that it is similar in spirit to the way benchmarks operate in the software industry. For example, electronic commerce benchmark applications may populate a database with random values (subject to certain constraints and statistical distributions). Examples include the official TPC-W benchmarks used by vendors like Oracle, IBM, and Microsoft. Once the databases and other data structures are populated with such data, benchmark queries are run to perform searches and display the results—to determine who has the "best" software, in terms of query performance (e.g., Oracle vs. DB2 vs. SQL Server).

Part 2: Create Two Hash Functions

Design two functions to hash the keys in your generated files from part 1:

```
int goodHash(string key);
int poorHash(string key);
```

For your `poorHash()` function you might want to use the sum of the ASCII character values of the string, modulus `hashTableSize`. Before doing any coding, try this small program to see how it works:

```
#include <iostream>
using namespace std;
int main(){
  cout << 'A' + 0 << endl; // try others, too
}
```

Now look at http://www.cplusplus.com/reference/string/string/at/ and .../length/. It is possible to implement `poorHash()` as described above using only a few of lines of code.

Do some research and think about what you have learned in class to come up with `goodHash()`.

Describe your hash functions in your `README` file and why you think they are good/bad. Note that it doesn't matter if they turn out both to be bad (or good!), just try your best. Part of this assignment is about testing performance to learn about your functions later on, anyway.

Part 3: Search, Insert, and Remove Functions; Collision Resolution Policies, including Tombstones (the `Hasher` class)

Deliverables:

- The `hashDriver.cpp` program, which includes all necessary functionality outlined below. A sample skeleton has been provided to get you started. Do not edit `TableEntry` or remove any functionality. You may, however, add functionality to `Hasher` and edit the main function.
- Implement the following hash searches (*probe sequences*) for keys in the hash table.
  a) Quadratic probe sequence, as shown in class
  b) Double hashing, using an appropriate secondary hash function of your choice.
- A tombstone system.
- An interface that works with the sample main function in `hashDriver.cpp`. **Note that this function is just for demonstration purposes so you can see how client code will use the classes and to ensure that your class meets the necessary specifications. You are responsible for creating your own driver (you can use this as a start) and testing your program thoroughly. See part 4.**

  *****When grading we will use our own main function.*****

Behaviour In Depth:

An instance of your `Hasher` class will use either the `goodHash()` or the `poorHash()` function, probe using either quadratic or double hashing, and have a constant capacity (i.e.: hashTableSize). The constructor should restrict the table to this behaviour. In other words, you will have four choices: *good+quadratic, good+double, poor+quadratic, poor+double*.

Your `search()` function should compute the hash value for the given key using the current hash function, and resolve collisions using one of the above strategies. You should pass it an additional argument (a single character) to indicate the intent of the search. This will determine how tombstones are treated in the search.

> **Recall:** Tombstones are needed so that your search will know when it can safely stop probing. Suppose you have the following *probe sequence* for some new key *k*: subscripts [120, 21, 40, 7, 195, 28], meaning that *k* and its associated value can be inserted into the hash table at location 28 (the proceeding five subscripts in the sequence generated collisions).
>
> Next, suppose key *j* is located in array location 40, and that we want to `remove(`*j*`)`. You can't just empty the entry at subscript 40, since a future search for *k* would stop at subscript 40 thinking that this is the end of the probe sequence—but it isn't.
>
> A tombstone indicates that the element with subscript 40 is currently empty and the table can accept an insertion there, but has not always been empty and so should not cause a search to terminate.
>
> A tombstone is any way of indicating this (e.g. a special value stored in the entry).

For simple lookups/search (`'s'`):

- If the key is found, return *true* and the subscript of the key's location
- ...otherwise return *false*

For insertion (`'i'`):

- If no acceptable slot for the entry exists (the table is effectively full), return *false*
- ...otherwise return *true* as well as the subscript of the first available spot in the probe sequence, which is where the new (**key**, **value**) entry will go. You must keep track of the number of probes required by each successful insert (for Part 4).

The `insert()` function should call the search function first (with the argument `'i'`), to locate an acceptable location for the insertion. If the insertion fails, the function should output "Table full."

Duplicates: Although it is very unlikely that your randomly generated file will contain any duplicate keys, you can not assume that the keys are unique. If you receive a duplicate entry,

consider it as an update to a previously existing entry and replace any information with the new information.

The `remove()` function should also call the search function. If the key is not found, then return false. Otherwise it should set the tombstone and return true.

The `printTable` command will display the subscript, the key and the data value of all non-empty elements in the table. No specific order is required; easiest is to simply loop through the underlying array. (You can assume that the empty string will never be stored, so you may initialize the keys to "" and bypass printing the empty elements. If you choose to implement your `Hasher` class as a template, you will have to handle this differently.)

Part 4: The hashDriver Program, Measurements, Questions

In this step, you will measure the number of probes required in a probe sequence by each of your hash functions. Ensure that you have written a `Hasher` constructor will accept the following arguments:

`hashType probeType loadFactor fileName` where:

`hashType`	is either `g` (for good) or `p` (for poor)
`probeType`	is either `d` (for double) or `q` (for quadratic)
`loadFactor`	is the load factor (eg: `0.25`)
`fileName`	is the name of the input file (eg: `small.txt`) used to create an instance of your `Hasher` class

Note that you will have to find the number of records in *`fileName`* before calculating the needed capacity-- your `Hasher` constructor should probably use a slightly higher prime number as the actual capacity (if you choose to do so, as described in Part 3 above).

Insert each record **(key** and **value)** in *`fileName`* into the table, and then display the actual load factor, capacity, size, and the average number of probes. This information will be most useful when dealing with larger files (e.g., 1000+ rows). Record these statistics in the following table. (Use 0.25, 0.50, 0.75 as the requested *loadFactor* and record the actual load factor in the table.) Typing the results into your README file in plain ASCII format is fine. You can provide more data if you like, but the following 6 trials will be sufficient for full marks for this part of the assignment.

Hash Function	Load Factor	hashTableSize (capacity)	# of Rows Inserted (size)	Average # of probes per insert	Expected # of probes (as per Donald Knuth's estimates)
goodHash	0.25				
goodHash	0.50				
goodHash	0.75				
poorHash	0.25				
poorHash	0.50				
poorHash	0.75				

Donald Knuth's estimates are listed in the course notes (you needn't provide these estimates for double hashing).

Questions:

In your README.txt file, answer the following:
1. How did you choose your hash functions? Why did you believe one hash function would be better than the other? Is your actual load factor much smaller than requested? Why or why not?
2. What is the relationship (if any) between load factor and the quality of the hash function?
3. How did you results compare to the Knuth estimates for quadratic probing? Discuss briefly (one or two lines is fine).

Final Deliverables Summary (to be submitted online, not on paper):

• All source code that you wrote (e.g. **.cpp**, **.hpp** or **.h** files) that has been tested to ensure that it runs on the ugrad machines. *Plan ahead.*
• One or two input data files (in case the marker wants to test using them).
• A **README.txt** file that gives any special instructions or comments to the marker, and including the table of measurements, similar in style to that shown above. If your **README.txt** file is missing or named anything else, you will lose marks!
• Some sample output, if you like—to encourage the marker to look more closely at your program, just in case your program doesn't seem to work (when the marker tests it).
• Don't worry about additional error checking, unless explicitly mentioned above.
• **You must comment your code adequately,** but don't go overboard.
• Provide PRE and POST conditions appropriately.
• And here is **what you should *not* hand in: .o files, executables, directories, backup files**, core dumps, recipes, horoscopes, money, and small rodents (i.e. irrelevant stuff).

Don't forget to comment your code and include pre- and post-conditions for any functions you may write. You will lose marks if you do not.

Other Notes

Partners

You may work in partners (no more than two people, *no exceptions*). If you choose to work in partners, some words of caution: Ensure that *both* partners are equally well versed in the program being written (otherwise the partner not doing his share of the work will be at a disadvantage having had less experience with the course material). The best way to work together is to take turns playing the role of the "driver", that is, the person on the keyboard. The other person should be discussing the project with the driver, ensuring that both understand what is happening. After a pre-designated amount of time (or after an objective has been met) be sure to switch roles.

Capacity note:

The capacity of your hash table (and hence the modulus used by your hash functions), will depend on the number of records in the input file and the load factor. You will likely find that forcing this capacity to be a prime number (equal to, or a little greater than, the requested capacity) decreases the likelihood that insert will fail to find an acceptable location. The following is an efficient C++ function to check if an integer is a prime number or not. If you use it, be sure to include the attribution to the original author (as given). You are welcome to supply your own instead. You are not required to force the hashTableSize to be prime (but you will probably want to).

```
// modified from an algorithm written by Francesco Balena
// downloaded from http://www.devx.com/vb2themax/Tip/19051
/** @pre  x > 0
 *  @post if x was prime, true returned, else false
 */
const bool isPrime(const int x){
  if (x == 1 || x == 2 || x == 3 || x == 5) return true;
  if (x % 2 == 0 || x % 3 == 0) { return false; }
  int incr = 4;    // NOTE: sqrt needs #include <math.h>
  const int maxFact = (int) sqrt( (double) x );
  for (int fact = 5; fact <= maxFact; fact += incr) {
    if (x % fact == 0) { return false; }
    incr = 6 - incr;
  }
  return true;
}
```

You will also need to implement the following instance methods (among others):

```
/** Returns the key stored at given subscript. */
const string getKey(const int subscript);
/** Returns the data value at given subscript. */
const int getValue(const int subscript);
/** Returns the number of probes. */
const int getProbes(void);
/** Returns the hashTableSize. */
const int getCapacity(void);
/** Returns the number of non-empty elements. */
const int getSize(void);
```