CS 320
Project 4
Hash Table Analysis using C++

Due Date: See CS320-01 Blackboard Website.
Implementation Requirement: **VS 2017/19** C++ Solution/Project, ***x86***, *Console Project.*
Required Name of VS 2017/19 C++ Solution/Project: *CS320P04LastName*
You will need create a project and copy over the .cpp, .h, and .txt files from the starter project.
Required Name of Excel Spreadsheet: *CS320P04LastNameResults.xlsx*
Required Name of Word Document Spreadsheet: *CS320P04LastNameAnswers.docx*

TURN IN:

- Include test executions in a *Tests* folder.
- Zip up your **entire** project, *CS320P04LastName*. **and** your *output files* (as presented below), *spreadsheet,* and word document into a zip file named: *CS320P04LastName.zip*.
- Upload zip file, *CS320P04LastName.zip,* to Project Assessment Icon. ***Remember***, the Zip file must contain all of the above and any other documents describing your project.

Project Overview:

In this project you are to write a Visual Studio 2017/19 C++ program that will count the number of probes (collisions) for several Hash Table implementations and display your results in an Excel spreadsheet. You will use linear probing, quadratic probing, and chaining, all without rehashing. You will use these three implementations with unsigned keys and a default hash function (provided by Visual C++ *xstddef.h* header file, which is implicitly included) and capture the number of probes needed for entering a random sequence of unsigned keys and then capture the number of probes needed for successful and unsuccessful lookups (using a contains member function) of another permutation of unsigned keys on each table. You will then use supplied hash functions, which are defined in a namespace to distinguish them from the default hash functions. You will repeat these same tests with string keys using a universe of strings from a given dictionary.

Generating unsigned Keys:
You are given code to generate a random sequence of 60,000 unsigned keys from a universe of {0,1, …, 59,999}. The first 30,000 keys of the permutation are inserted sequentially starting at index 0 into a Hash Table, which is constructed to have enough buckets so that it will have a load factor of 10% after 30000 keys are inserted. (For example, if 10,000 keys are inserted, the table must have 100,000 buckets.) Eight other tables are constructed with enough buckets to have respective load factors of 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90%. The same permutation of 30,000 unsigned keys are inserted into to each of these tables. To help organize the testing, these nine tables are stored in an array in positions 1, 2, … 9 corresponding to the respective 10%, 20%, …, 90% load factors. The position is unused, however a small default value of 101 buckets is used for that unused Hash Table.

The original permutation of 60,000 unsigned universe used to create tables with the above load factors, are randomly permuted again. The first 30,000 keys of the 60,000 of the new permutation are used to count the number of probes needed for successful lookups (contains(key) in the code) and the number of probes needed for unsuccessful lookups. You must keep probe counts for each of the Hash Tables different load factors.

To get similar results to the provide numbers for the sample data given for linear probing, you must use the provided random number generators and must use the same seeds for the random numbers. After you meet the specifications of the minimum output required for this project, you may find it interesting to try different seeds and see what results you get.

Generating string Keys:

Similarly, you will analyze nine Hash Tables with string keys are constructed. The universe of strings is a dictionary that is supplied and is read into an array of strings. As with the {0, 1, …, 59,999} unsigned universe, you will read the first 60,000 words of the provided dictionary. A random permutation of these 60,000 words are generated and the first 30,000 words are inserted into the table. As with the unsigned number case, you must randomly permute the permutation again to generate a sequence for computing the number probes needed for each of the successful calls to *contains(x)* and the number of probes need for each the unsuccessful calls to *contains(x)*.

Counting the number of probes:

A global unsigned variable, probes, is used to communicate between the *insert(key*) member function and the driver code and the *contains(key)* member function and the driver. Read about external variables in C and C++. For each call to *insert(key)* or *contains(key)* you must **reset** the probes variable to 0. However, you need to have accumulators to store the total number of probes.

Writing your code:

To illustrate the organization and requirements of your code, you are given a Hash Table with linear probing, called *HashTableLinear<HashedObj>*. Also, to analyze the number of probes needed, a free function, called *Test_HT_unsigned_linear()* is given; it will take the generated keys and insert them into each of the Hash Tables (of the various load factors) and use the second permutation of keys to count the number of probes needed for the successful and unsuccessful lookups. All probe counts are all written to a file called *HashTableAnalysis.txt*. Only the successful and unsuccessful counts for testing contains(key) are placed in the Excel spreadsheet. These counts are compared against the theoretical counts.

You will need to modify the Quadratic Probing and Chaining implementations so that the right number of probes are counted. You will also need to complete the respective driver functions. So, you will need following tests: unsigned Linear (given), unsigned Quadratic, unsigned Chaining, string Linear, string Quadratic, and string Chaining. You must generate the data using the Visual C++ *xstddef.h*-provided hash functions.

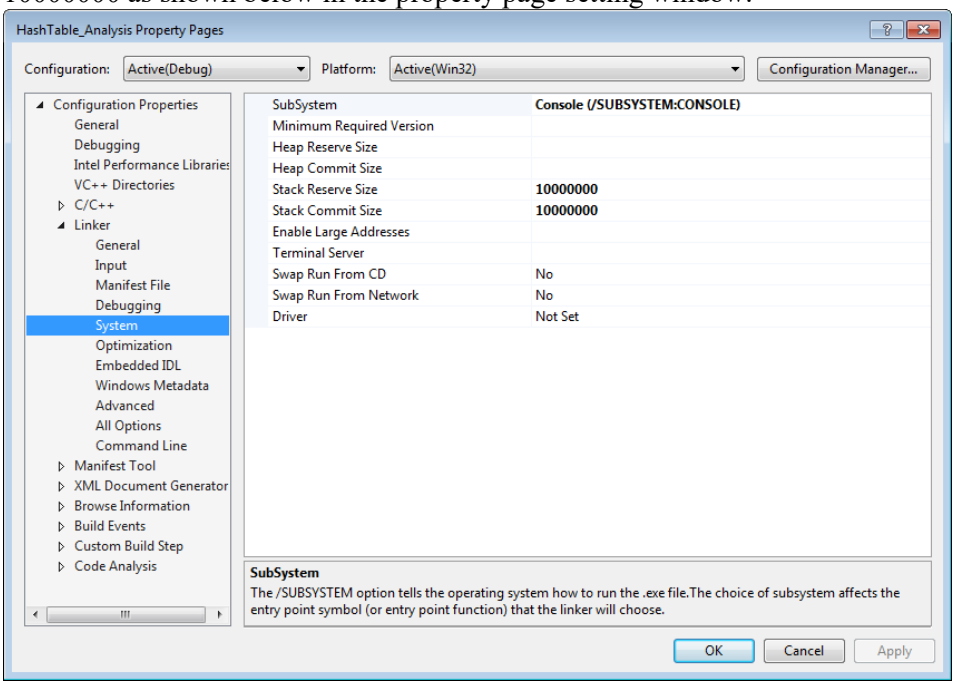
You can leave all your code for the tests for the Visual C++ hash function, but for using the "my" hash functions, you must comment out the default Visual C++ data member declaration and uncomment the hash function found in the namespace *MY_HASH_FUNCTIONS* and defined in the provided *MyHashFunctions.h* header file. Also, I suggest that you rename the output file (e.g. *HashTableAnalysisMyFuns.txt*) so as to not overwrite your previous results. After doing these edits, re-build your solution/project. Run your code. Capture the data from your second set of executions (using *MY_HASH_FUNCTIONS* namespace) and **move the appropriate data into the spreadsheet**. See the spreadsheet template for the format. Make sure you use formulas where appropriate. Make sure you save the spreadsheet in an appropriate format so that the formulas don't disappear.

You are given the following code file:

File Name	Contents	What to do?
LinearProbing.h	HashTableLinear<HashedObj> definition along with modification for counting number of probes.	Nothing, except for changing the namespace of hash functions used when necessary.
QuadraticProbing.h	HashTableQuad<HashedObj> definition as given. Needs modification.	Code given, but you must modify code to count the number of probes needed in each call to appropriate member functions. Also, you will need to change namespace of hash functions when necessary.
SeparateChaining.h	HashTableChaining<HashedObj> definition as given. Needs modification.	Code given, but you must modify code to count the number of probes needed in each call to appropriate member functions. Also, you will need to change namespace of hash functions when necessary. Hint: because the code given uses STL lists, you will need to modify traversing the lists so you can count the probes.
Utilities.h	Declarations of functions needed by all Hash Table implementations	No modifications.
Utilities.cpp	Definitions of the routines declared in Utilities.h	No modifications.
UniformRandom.h	Definitions of Random Generators.	No modifications. You must use this code for random number generation.
TestHashTables.cpp	Driver to generate the keys and then run tests.	You do not have to modify this code.
MyHashFunctions.h	Has the definition of my hash functions in MY_HASH_FUNCTIONS namespace	No modifications.
Test_HT_unsigned_Linear.h Test_HT_unsigned_Linear.cpp	This drives the HashTableLinear tables for unsigned keys.	Code is given. Use as a guide for writing the other Test_HT_unsigned_yyyy.h and .cpp files.
Test_HT_string_Linear.cpp Test_HT_string_Linear.h	This drives the HashTableLinear tables for string keys.	Code is given. Use as a guide for writing the other Test_HT_string_yyyy.h and .cpp files.
Test_HT_string_Chaining.h Test_HT_string_Quadratic.cpp Test_HT_string_Quadratic.h Test_HT_unsigned_Chaining.cpp Test_HT_unsigned_Chaining.h Test_HT_unsigned_Quadratic.cpp Test_HT_unsigned_Quadratic.h	All .cpp need to be written.	All .cpp need to be written.

Constraints:

In your Project property settings go to Linker->System and set the Stack Reserve and Commit sizes to 10000000 as shown below in the property page setting window.



Input / Output Specifications:

INPUT: Use the file *dictionary.txt* for testing hashing of strings.
OUPUT:

- Write progress messages to standard output as given in the Test_HT_unsigned_Linear.cpp code.
- Write all statistics to *HashTableAnalysis.txt* as shown in Test_HT_unsigned_Linear.cpp. Create a new (overwrite existing) file for each run. Append all the results as all the tests run. See sample *HashTableAnalysisFormat.txt* for an example for formatting purposes.

Spreadsheet Format:

You will need to modify the given template spreadsheet. Note that some numbers are provided the theoretical rows. These are not necessarily correct. These have to be replaced with the theoretical functions that you write as a function of the load factor cells. The theoretical formulas are based upon those as shown in your textbook. Note that there are no theoretical rows for quadratic probing. The data that your program generates is used to put into the experimental data rows in the spreadsheet.

Here is the general structure for the spreadsheet in outline form.

1) Default hash functions
    a) unsigned
        i) linear probing
            (1) theoretical successful probes (you need to calculate with formula as a function of load factor)
            (2) experimental successful probes (from running your code)
            (3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
            (4) experimental unsuccessful probes (from running your code)
        ii) quadratic probing
            (1) experimental successful probes (from running your code)
            (2) experimental unsuccessful probes (from running your code)

iii) chaining
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

b) string
i) linear probing
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes (from running your code)
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

ii) quadratic probing *(note: no theoretical formulas are to be used)*
(1) experimental successful probes (from running your code)
(2) experimental unsuccessful probes (from running your code)

iii) chaining
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

2) My hash functions
a) unsigned
i) linear probing
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes (from running your code)
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

ii) quadratic probing *(note: no theoretical formulas are to be used)*
(1) experimental successful probes (from running your code)
(2) experimental unsuccessful probes (from running your code)

iii) chaining
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

b) string
i) linear probing
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes (from running your code)
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

ii) quadratic probing *(note: no theoretical formulas are to be used)*
(1) experimental successful probes (from running your code)
(2) experimental unsuccessful probes (from running your code)

iii) chaining
(1) theoretical successful probes (you need to calculate with formula as a function of load factor)
(2) experimental successful probes
(3) theoretical unsuccessful probes (you need to calculate with formula as a function of load factor)
(4) experimental unsuccessful probes (from running your code)

Shown below is a sampling of the first few group [1 a)] for using the default unsigned hash function.

HashTable<unsigned> default Hash Functions									
Linear Probing									
Load Factors	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Theoretical Avg Successul Probes	1.055556	1.125	1.214286	1.333333	1.5	1.75	2.166667	3	5.5
Experimental Avg Successful Probes	***1.05419***	***1.05654***	***1.04895***	***1.49546***	***1.26193***	***1.78189***	***1.92914***	***2.54341***	***3.79509***
Theoretical Avg Unsuccessful Probes	1.117284	1.28125	1.520408	1.888889	2.5	3.625	6.055556	13	50.5

Experimental Avg UnSuccessful Probes	*1.10329*	*1.13393*	*1.14207*	*2.09446*	*1.77093*	*4.10628*	*4.84442*	*8.58355*	*33.7023*
HashtTable<unsigned> default Hash Functions									
Quadratic Probing									
Load Factors	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Experimental Avg Successful Probes									
Experimental Avg UnSuccessful Probes									
HashtTable<unsigned> default Hash Functions									
Chaining									
Load Factors	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Theoretical Avg Successul Probes	1.05	1.1	1.15	1.2	1.25	1.3	1.35	1.4	1.45
Experimental Avg Successful Probes	***1.05319***	***1.048***	***1.02957***	***1.36589***	***1.14884***	***1.20515***	***1.28007***	***1.6436***	***1.34386***
Theoretical Avg Unsuccessful Probes	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
Experimental Avg UnSuccessful Probes	***1.10106***	***1.10851***	***1.06873***	***1.70673***	***1.28951***	***1.4262***	***1.56328***	***2.22554***	***1.69974***

ANSWER the following:

1. Using default hash functions, what worked best for unsigned keys: Linear, Quadratic, or Chaining? **Support answer**.
2. Using default hash functions, what worked best for string keys: Linear, Quadratic, or Chaining? **Support answer**.
3. Using my hash functions, what worked best for unsigned keys: Linear, Quadratic, or Chaining? **Support answer**.
4. Using my hash functions, what worked best for string keys: Linear, Quadratic, or Chaining? **Support answer**.
5. Did you notice any tests taking longer than the other tests? Which ones? Why?