School of Electrical and Computer Engineering
CptS223: Advanced Data Structures C/C++
Spring 2018
Homework #10 (Programming Assignment)
Due: Wednesday 04/25/2018 @ 11:59pm

Instructions: All source code must be in C++. You are required to use the Unix environment. You submission should accompany a report that explains your work. Your code will be tested on EECS servers.

For this programming assignment, you have the option of either working as a TEAM if you prefer, or as an individual. A TEAM should contain at most 2 students including yourself. Grading will NOT differentiate between the two - i.e., if you are working as a team of two people, then the same grade will be applied to both of you.

If working as a team, both of you should submit the same package. However, you will need to clearly mention team members' names in your report. If working as a team, also remember to include both your last names in the zip file of your submission.

In the rest of this document, the term "YOU" is used either to refer to you (if you are working individually) or your team (if you work as a team).

The final material for submission should be entirely written by you. If you decide to consult with others or refer materials online, you MUST give due credits to these sources (people, books, webpages, etc.) by listing them in the report that accompanies your submission. Note that no points will be deducted for referencing these sources. However, your discussion/consultation should be limited to the initial design level. Sharing or even showing your source code/assignment solution verbiage to anyone else in the class, or direct reproduction of source code/verbiage from online resources, will all be considered plagiarism, and will therefore be awarded ZERO points and subject to the WSU Academic Dishonesty policy. Reproducing from the Weiss textbook is an exception to this rule, and such reproduction is encouraged wherever possible. Please note that most of the source codes for your textbook are available from the following website:

https://users.cs.fiu.edu/~weiss/dsaa_c++4/code/

Grading will be based on correctness, coding style, implementation efficiency, exception handling, source code documentation, and the written report.

All assignments files (code and report) should be zipped/tarred into one archive folder (named after your last name/s), and submitted through WSU Blackboard by the submission deadline.

Note that submissions through any other means for example by emailing your package to the instructor/TA/TAs will be discarded and will NOT be graded. So please follow the above instructions carefully.

Late submission policy: A late penalty of 10% will be assessed for late submissions within the next 24 hours (i.e., until midnight of the next day after the submission deadline). After this one-time late submission, no submissions will be accepted.

1. Introduction

For this programming assignment, you will be comparing the performance of different hash functions and different collision resolution implementations. More specifically, you will be testing hash table implementations to store and retrieve a set of strings. The input is a set of 'n' strings, and for all practical purposes, you can assume that the length of each string can be bound by a small constant. You need to build and maintain a searchable hash table for this input. Obviously there are many ways you can design the hash table, depending on what hash function you pick, and what collision resolution strategy you choose.

2. Aim 1: Comparing Different Collision Resolution Algorithms

The first goal in this assignment is to compare different collision resolution strategies while keeping the hash function fixed. For this aim, you will be implementing and experimentally comparing three collision resolution mechanisms: chaining, linear probing, and quadratic probing. For doing this, you do not have to write the code from scratch. The source code for chaining and quadratic probing are already downloadable from the Weiss textbook's webpage:

https://users.cs.fiu.edu/~weiss/dsaa_c++4/code/

You need to make sure that the source codes that you will be using for this assignment are STL compliant. If they are not, you will need to make appropriate modifications. You can also modify the code for your assignment as needed. You might want to think of adding your own lines of code for additional testing/experimentation.

Notice that you may need to implement the code to do linear probing. This should be a very simple modification of the quadratic probing code (more specifically, to the function that finds the next position of probe). Call

the resulting source files for linear probing as LinearProbing.h and LinearProbing.cpp.

You will also notice that the hash function for converting a string key into a hash table index used by all these three implementations are the same (for example, defined as "*int hash(const string & key, int tableSize)*" inside the .cpp files. For this aim, we will keep this hash function as-is, and just experiment by varying the collision resolution implementations, including chaining, linear probing and quadratic probing.

You will also notice that the quadratic probing code initializes the hash *tablesize* to a default size of for example 101, and then does a rehash anytime the number of elements in the hash table reaches half of the hash *tablesize*. And rehashing doubles the *tablesize*. Leave these things as they are and keep the same setup for linear probing as well.

You will also notice that the chaining implementation initializes the hash *tablesize* to a default size of for example 101. However it does NOT do rehashing. Again, leave this as is.

Experimental plan:

Input files: There are two input files provided for your testing and performance measurements. These input files include:

- (***"Data file"***) The first file is *OHenry.txt*. This is a file that contains exactly one word per line. The "word's" definition includes all alphanumeric and special characters except the whitespace character. (This text file is actually a set of O Henry's short stories that was downloaded and transformed it into this single word per line format so that it is easy to read.) There are a total of 10,377 words in this file and represents the input strings to be inserted into the hash table. The number of input strings is denoted by '*n*' (i.e., *n=10377* in this assignment).

- (***"Query file"***) The second file is *queries.txt*. This file contains a list of 1,500 words which should be used as "queries" to search for in the hash table. The number of queries is denoted by *'m'* (i.e., *m=1500* in this assignment).

For the sake of simplicity of your code, we will assume that the names of these two input files will not change and that these two input files are always made available in the current working directory of execution. Of course you need to ensure this when you run and test the code.

Write a test driver function, call it *TestAim1.cpp*, whose *main()* function performs the following sequence of operations in order:

1. Open the input data file (*OHenry.txt*) and load the contents into a vector of strings. Let us call this vector object "*DataArray*".
2. Open the input query file (*queries.txt*) and load the contents into another vector of strings. Let us call this vector object "*QueryArray*".
3. Instantiate all of the above three hash table (HT) implementations. Let us denote the corresponding objects as "*ChainingHT*", "*LinearProbingHT*" and "*QuadraticProbingHT*".
4. Analysis of Chaining
    a. Call a function "*InsertIntoChainingHT(DataArray)*". This function should insert each word in *DataArray*, one by one, into *ChainingHT*. Of course, recall that no duplicates are allowed in the hash table.
    b. Initialize a separate timer called *InsertionTimerChainingHT* and record the time taken to do all the insertions into this variable. Basically, this timer should contain the sum of the time taken to do all *'n'* insertions. From this you can calculate the average time per insertion. Make sure you include only the time to do the insert, which means only the time the code spends inside the *insert()* function call. The way to do this would be, set a timer t_1 at entrance of the *insert()* function and set a

timer t_2 just before the exit of the *insert()* function (it does not matter whether insertion was successful or not), and then add [*t2-t1*] to the global timer variable *InsertionTimerChainingHT*.

c. Also, initialize and keep track of a variable called *CollisionsChainingHT* to count the total number of collisions across all insertions. Note that this is same as counting the total number of times you are inside the while loop within function *findPos()* across all insertions.

d. Next, we will do searches and time the searches. To do this, call a function "*SearchChainingHT(QueryArray)*" which does a find for every query in the list of queries. For analyzing this, just keep track a global timer variable called *SearchTimerChainingHT* which should contain the total time to do searches for all the *'m'* queries. From this, you can calculate the average time per search.

5. Analysis of Linear Probing and Quadratic Probing: Apply the same procedure as described above to analyze the linear probing and quadratic probing implementations.

At the end of this experiment, you should generate the following table as part of your report and include your obtained performance numbers.

Collision Strategy	**Insert**			**Search**	
	Total time	**Average time**	**Total number of collisions**	**Total time**	**Average time**
Chaining					
Linear Probing					
Quadratic Probing					

Please make sure that you specify the unit for the run-time values int the appropriate columns in the above table (i.e., hours or minutes or seconds or milliseconds or microseconds, as appropriate).

3. Aim 2: Comparing Different Hash Functions

The main goal of this experiment is to compare different hash functions while keeping the collision strategy fixed. You will compare three different string hashing functions (given in the textbook) keeping the collision strategy fixed at Quadratic Probing. The three hash functions to use are all given in the textbook, Figure 5.2, Figure 5.3 and Figure 5.4. Note that Figure 5.4 is the same as the default hash function used in the source code for Aim 1. For the purpose of analysis, let us refer to these three hash functions as follows:

1. Figure 5.2: "Simple hash function"
2. Figure 5.3: "Prefix hash function"
3. Figure 5.4: "Full-length hash function"

For comparing these three hash functions, first add the other two hash functions into the source code for Quadratic Probing (obviously using different names to refer to those functions). Then, repeat the same procedure that you used to analyze Quadratic Probing method in Aim 1, with the only exception that now you should keep track of separate timer variables and collision variables for the three different hash functions. As a result of this experiment, you should generate the following table:

Hash function	Insert			Search	
	Total time	Average time	Total number of collisions	Total time	Average time
Simple					
Prefix					
Full-length					

Please note that you do not have to repeat the experiment for the row corresponding to the Full-length. The results should already be available from the corresponding entry in Aim 1.

5. Report

In a separate document (e.g., Word or PDF), not more than 2 pages, compile the following sections:

Section A: Problem statement. In 1-2 sentences state the goal(s) of this exercise.

Section B: Experimental setup.

- Specify the machine architecture (CPU, clock speed, RAM) where all the testing was conducted.
- How many experiments were performed and averaged, to determine each point in your plot?

Section C: Experimental Results. In this section, include the following:

- The two tables as described above.
- Are the observations made in the above plots as per your theoretical expectations? Justify.

5. Grading Rubric

This assignment is mainly about empirical testing and analysis. There is some design component but not much. Thus, for grading this programming assignment, we will primarily look at how well you have designed experiments, what the plots look like, and have you offered satisfactory/convincing rationale to explain your observations. Therefore, the points during grading will be distributed as follows (the whole assignment is worth a total of 100 points):

CODING (45 pts):

- (10 pts): Are all versions of the hash tables and functions implemented correctly?
- (10 pts): Is the code implemented in an efficient way? i.e., are there parts in the code that appear redundant, or implemented in ways that can be easily improved? Does the code conform to good coding practices of Objected Oriented programming?
- (15 pts): Does the code compile and run successfully on a couple of test cases?
- (10 pts): Is the code documented well and generally easy to read (with helpful comments and pointers)?

REPORT (55 pts):

- (15 pts): Is the experimental plan technically sound? Is the experimental setup specified clearly?
- (10 pts): Are results shown as plots in a well annotated manner and are general trends in the results visible?
- (30 pts): Are the justifications/reasons provided to explain the observations analytically sound? Is there a reasonable attempt to explain anomalies (i.e., results that go against analytical expectations), if any?

Obviously to come up with the above evaluation, the TAs are going to both read and run your code.