Programming Assignment 8: Hash Tables

1. Introduction

This assignment provides students with an opportunity to work with A Hash Table using Open Addressing and implementing the buckets of the hash table as sorted linked lists. Since this is the last assignment and everyone is a bit extra busy at the end of the semester, the assignment has been kept a bit simpler than the instructor's initial impulse, in an effort to not make the last few days of the semester unduly difficult, and to give some time for students to reflect on and integrate the wide range of concepts covered in the class prior to the final exam.

The basic idea of the assignment is to use a hash table to organize a small phone book. Just for variety, the index key will be the phone number rather than the user's last or first names. One of the instructor's impulses that was suppressed was to have students construct two idices, using two hash tables; one indexed by phone number and one indexed by user names. Having done one, the other should be quite easy. Students should feel free to try that on their own if they feel the assignment as presented short-changes them on the possible educational experience or somewhat insults their intelligence. No such short-changing or insult was intended.

The Make file provided implements 3 tests: test1, test2, and test3. For each of these a reference output file giving the “correct” output for each test is given and the Make file steps compare your implementation's output with the reference output. As with previous assignments, we have tired, and believe we have succeeded, in giving you all the output statements you require to be able to match the reference output precisely. However, as before, if we have overlooked an output statement let us know and we will correct the oversight. Minor differences of formatting are not all that significant, but a perfect match is an aid to efficient grading and should be too difficult to achieve.

2. Getting Started

The starter code provided defines ATDs, classes, for the hash table, the bucket, the linked list, and the phone book entry. These are defined in the header file and C++ source file pairs bearing the obvious names. In addition, the code for the main program is defined in “main.cpp” and some reasonably familiar utility routines are defined in “Utility.h” and “Utility.cpp”. No changes to “main.cpp” should be required and only four one line changes are required to “Utility.cpp”. The rest of the implementation you need to do is in three files: “Hash_Table.cpp”, “Bucket.cpp” and “Linked_List.cpp”.

The following is an example of how the program can be run from the command line:

```
$ ./hash_table infile_db.txt infile_cmd.txt 10
```

This builds the phone book from the information specified in “infile_db.txt” and then processes the commands specified in “infile_cmd.txt”. As with previous assignments, the main program does little except obtain the command line arguments and call the utility routines that first build the phone book, **process_phonebook_db()** and then execute commands using the phone book data structure, **process_phonebook_cmd()**. Only a single line of code should need to be added to the routine building the phone book hash table, and three one line changes to the code of the routine reading and executing the commands that use the hash table once it is built.

The three test in the Make file all use the same set of data and the same set of commands, only differing in the last argument: the number of entries in the Hash Table. In the command illustrated above, 10 buckets are defined. The other tests define 20 and 30 buckets, respectively. This is to illustrate how hash tables, and their hash functions, can be parameterized to use different table sizes quite easily. Dynamically resizing the hash table as the number of entries, and thus the load factor, grows is also quite simple. Use of a larger phone number data set and implementation of a feature to dynamically check the load factor as entries were added, resizing and rehashing at a given load factor threshold, were also features at first attractive to the instructor, but not included due to this being the last assignment. Students feeling educationally short-changed or intellectually affronted should feel free to add these features in their implementation.

2.1. Hash Table

This class is the bulk of the assignment. You should be able to implement all methods, following the API specified in the “Hash_Table.h” header file. Please implement the methods in the same order specified in the header file to aid grading. You may wish to consider the reference output files in some detail to make the output of your implementation a perfect match. The instructors have tried to include all the relevant output lines. Let them know if some are missing, although students should be able to construct their own to match the example output at this point in the semester's learning curve.

Methods for this class should include:

Hash_Table(int size)
This is the constructor of an empty hash table with **size** buckets.

~Hash_Table( )
This is the destructor for the Hash Table. Remember that one important basic principle of object design is that any call to **new** in the constructor generally requires a corresponding call to **delete** in the destructor.

insert(string firstname, string lastname, string number)
This is the routine that inserts a new element in the hash table and thus in the phone book. Remember that the assignment is to use the phone number as the key for hash table operations.

search(string key, ostream &os)
This routine searches for an element in the hash table associated with the specified key. For convenience in the assignment, rather than generally good design, we have it take an output stream port as an argument so that this routine can print the search result, be it success or failure.

get_load_factor( )
This is routine calculates and returns the hash table load factor.

print [overloaded output operator]
This represents the overloaded output operator which prints the contents of the hash table. Note that this routine prints the contents of each bucket in the order in which they are on the bucket list, and prints the buckets from the beginning of the table to the end. Students may find it useful to consult the reference output as a help in getting their output format to correspond to the reference output exactly. An exact match would be appreciated by the graders.

uint32_t Hash_Table::SuperFastHash(const char *data, int len)
This is the hash function provided, which we obtained as open source code available under an LGPL license. It seems to work well, so we thought it worth a try. The URL for the source code WWW site is in the starter source code.

2.2. Bucket

this class implements the bucket. The hash Table, at its essential core, is simply an array of buckets.

Methods for this class should include:

Bucket( )
This is the constructor of an empty bucket.

~Bucket( )
This is the destructor of the bucket. Remember that all calls to **new** need corresponding calls to **delete**, even if the calls to **new** are not issued in the constructor.

get_size( )
This routine returns the size of the bucket, the number of elements in it.

insert(PB_entry* in_entry)
This method inserts a Phone Book Entry instance into the Linked List implementing the bucket.

PB_entry* search(string key)
This method searches the bucket for an entry with the specified key. Remember that the key we have chosen for this assignment is the phone number string. Obviously, this method returns a pointer to the relevant phone book entry.

print [overloaded output operator]
This represents the overloaded output operator. This operator is responsible for outputting the contents of the bucket on the output stream argument, and returning the output stream, as with all such overloaded operator routines.

2.3. Linked List

This class implements the Linked List that is used for each bucket. Since the implementation of the linked list should be painfully familiar and trivially easy for students at this point, we have provided the basics of the implementation. Student will have to complete the implementation of only the **sorted_insert( )** and **retrieve( )** methods.

Methods for this class should include:

Linked_List( )
This is the constructor of an empty list.

~Linked_list( )
This is the destructor of a linked list.

sorted_insert(PB_entry* entry)
This method inserts the Phone Book Entry instance pointed to by the parameter into the sorted liked list. Students must provide an implementation of this routine. Note that students may, if they find it easier, to initially implement a simpler unsorted insert and later refine their implementation.

retrieve(string key)
This method looks in the linked list instance to see if a node with the specified key is present. Students must provide an implementation of this routine. Note that students may, if they find it easier, to initially implement a simpler unsorted retrieve and later refine their implementation in coordination with how they implement **sorted_insert( )**.

is_empty( )
This method indicates if the linked list is empty or not.

get_size( )
This method returns the number of elements in the list.

print(ostream& os)
This method prints the contents of the linked list on the output stream supplied as an argument.

2.4. Phone Book Entry

This class implements a single Phone Book Entry. This is the essential data record of the implementation which is, in turn, encapsulated by the linked list node, and this stored in the buckets of the hash table. This class is simple enough that we provide its complete implementation in the starter code. Students should ensure, however, that they thoroughly understand the implementation provided.

Methods for this class should include:

PB_entry(std::string first, std::string last, std::string number)
This is the constructor of a Phone book entry, taking the first name, last name and phone number of a user as arguments. No explicit destructor is specified because it is not needed. Students should be able to explain by this class, in contrast to others, can reply on the automatically generator default destructor.

string get_key( )

This method returns the key of the record. Remember that in this assignment we are using the string representing the phone number.

print [overloaded output operator]

This is the overloaded print operator outputting the data contained by the phone book entry instance in a standard format. Note that this uses private **format_number( )** method to format the phone number. A minor point, but a good example of how private methods can be used to help implement public methods/

string format_number( )

This is a private method providing a formatted version of the entry's phone number.

3. Input & Output

Your implementation is expected to process the input files provided, and to produce output precisely matching the reference output for the three tests using different hash table sizes. Output differing on in the number of spaces on a line or the number of blank lines should not be detected as a difference. This is what the “-b -B” arguments to the calls to the “diff” program in the makefile indicate to the command. Other differences should be easy to eliminate from an otherwise correct implementation with some reasonably careful inspection of the reference output and examination of the student's implementation of the relevant output routines.

3.1. Phone Book Creation and Operation Commands

There are no commands related to creating the Phone Book. The **process_phonebook_db( )** routine defined in “Utility.cpp” assumes that the input file provides one Phone Book Entry per row, with each row containing three fields: first-name, last-name, and phone-number. Students should only need to supply a single source line to complete the implementation of this routine.

The commands related to using the hash table constructed by processing the commands in the first file, include: **print, search,** and **load_factor**. These commands are processed by the **process_phonebook_cmd( )** routine and require the student to supply a single source line to complete the implementation of the code for each command in the routine.

Command Discussions

The semantics of the commands should, mostly, be evident from the names of the commands.

print

Print the hash table. Note that this is done bucket by bucket, in the order in which the buckets appear in the hash table.

search

Search the hash table for an entry with the specified phone number.

load_factor

Print the hash table load factor.

3.1. Sample Input

There are only two input files: “infile_db.txt” and “infile_cmd.txt”. The first containes the data for the phone book entries, one per line. The second contains the set of pheon book commands. Both are simple enough that students should have no problem understanding their meaning and how the **hash_table** command should process them.

3.2. Sample Output

The Make file is set up to run three tests, each of which use the wame two input files, but which differ in the size of the hash table created and used. Reference output is provided for each test and students should be able to get the output of an otherwise correctly completed implementation to match the reference output precise. Or, at least precisely enough to avoid any objection from the calls to the **diff** command provided in the Make file.

4. Grading Criteria

40%	24 pts	Hash Table
20%	12 pts	Chaining
10%	6 pts	Sorted Chain
10%	6 pts	Load Factor
10%	6 pts	Output
10%	6 pts	Documentation and coding style
100%	60 pts	Total