Multiple-Key Indexing

For this project you provide indexing capabilities for a simple database file. The database will consist of a sequence of logical records, of varying sizes (similar to the initial data file for the hash table project). Each record will contain multiple fields (data values), some of which will be unique to that record and some of which may be duplicated in other records. The database file will never contain two copies of the same record.

Specifically, each record will be of the following form:

Name:	`Fred Flintstone`	newline-terminated string
Address1:	`101 Rock Road`	newline-terminated string
Address2:	`Stoneville TN 12345`	3 tab-separated strings
Phone:	`555.303.1119`	newline-terminated string

You will build two indices for the database file. The first will be a hash table, using the Name field as its key. Names are guaranteed to be unique (i.e., no two different records will contain the same name string). Hash table details are given below. The second index will be an inverted table (section 9.3.3 of Kruse/Ryba and the course notes), using the zip code as a key value. It is certainly possible that two different records will contain the same zip code. Details of the inverted table are also given below.

Each index will store key values and the addresses of the corresponding records in the database file. The address of a record is defined to be the byte offset to the first byte of the file (i.e., the first character of the name field in this case). Note that byte counts start at 0.

Given the byte offset to a record, it is easy to advance the read cursor (extraction point from the file stream) to that offset and then or write the record. You will probably want to make use of the input `stream` functions `seekg()` and `tellg()`, and the corresponding output stream functions `seekp()` and `tellp()`.

Since this project requires performing interleaved reads and writes on the same file, you should use an `fstream` to access the file, rather than separate `ifstream` and `ofstream` objects. There's not much difference between the two, but it is somewhat dangerous to access a file concurrently via two or more different streams.

Note: the complete list of database records is NOT to be stored in memory. Individual records may be read from disk to memory as needed.

On startup, the program will read the database file and build the specified indices. Next, the program will read a command file and carry out the commands, writing any output to a log file.

Data Structures:

The primary data structures of this project are a hash table and an inverted table. Your implementation is under the following specific requirements:

- You must encapsulate the two indices as C++ classes. The use of templates is preferred but not required (i.e., no penalty).
- The data that is stored in each index slot must be encapsulated in a C++ class.
- Given the nature of this project, no two distinct database records will contain duplicate names. Your implementation should detect the insertion of a duplicate name and reject the insertion.
- For testing, both of your indices should have the ability to display themselves to a specified output stream.

Without exception, data members of classes should be private.

Hash Table Index:

The hash table in this project does NOT store the data records. Rather, it stores a hash record containing a key value (name string) and the address of the corresponding record.

When a lookup is performed, the hash table should be passed either a hash record (possibly containing a dummy address) or a key value. A successful lookup should result in the hash table returning the corresponding hash record it found, or the corresponding record address; a failed lookup should result in the hash table returning a logically invalid hash record (negative values are useful here) or a logically invalid record address.

The hash table should use the same hash function as before, and linear probing. Since this uses linear probing, there is no reason to worry about failed insertions unless the table is completely full. The condition for terminating a probe should be if the number of slots examined reaches the size of the hash table. As with the hash table project, insertion should use the first tombstone along the probe sequence, if there is one.

The number of slots in the hash table should be two times the number of records in the database file.

Inverted Table Index:

The inverted table contains an array of table records. Each table record contains a key value (zip code string) and the address of a database record. The array of records should be sorted by zip code so that binary search can be used. The inverted table should be initialized to contain as many slots as the hash table. Typically, some of these will not be used.

Insertions to the inverted table may require shifting entries. However, if the inverted table is full then insertions will simply fail. Records may also be deleted. In that case, the corresponding entry in the inverted table should be deleted as well, requiring a shift of part of the array.

Program Invocation:

Your program **must** take the names of the input and output files from the command line — failure to do this will irritate the person for whom you will demo your project. The program will be invoked as:

```
index  <database file name>  <command file name>  <log file name>
```

If either of the specified input files does not exist, the program should print an appropriate error message and either exit or prompt the use for a correction.

Database File Description:

The database file will contain a header specifying the number of records (this value is used to determine the size of the hash table and the inverted table). The remainder of the database file consists of a sequence of records (as described earlier), delimited by lines of 20 hyphens:

The database file is guaranteed to conform to the specified syntax.

```
Number of records in database file:   10
--------------------
Willis, Connie
314 Perimeter Place
Delta UT    88888
555.258.8002
--------------------
de Camp, L Sprague
Rama Road
Hobbiton    M E   00001
555.999.0001
--------------------
Pournelle, Jerry
900 Vine Blvd
Burbank     CA    90001
555.404.7891
--------------------
. . .

Robinson, Kelly
7000 Granite St
Brooklyn    NY    09041
555.701.0012
------------------------------
Howard, Robert E
8to1 Stochastic Ave
Colombo     Sri Lanka   00509
555.333.5546
------------------------------
```

Driver File Description:

Lines beginning with a semicolon (`';'`) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the input file.

Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of "tokens" which will be separated by single tab characters. A newline character will immediately follow the final "token" on each line.

`name` `<name>`

This causes the hash table to be searched for the location of the record, if any, that contains the given string in its name field. After the search, a message should be printed to the log file indicating either that the search failed, or printing the address of the record followed by the contents of the record (all fields).

`zip` `<zipcode>`

This has the same effect as the previous command except that the inverted table is searched for the address of a matching record. In this case, there may be multiple matches to the given zip code. If that happens, you should report each match (formatted as for the **`name`** command.

`insert` `<name>` `<street>` `<city>` `<state>` `<zip>` `<phone>`

If a record containing the given name is already in the database file, this has no effect. If the indices are full, this has no effect. Otherwise, this causes the updating of both indices and the insertion of a new record at the end of the database file (just move to the end and start writing). Log an informative message in all cases.

`delete` `<name>`

This causes the hash table to be searched for the location of the matching record, if any. If a match is found, then the corresponding record should be marked with a tombstone. The inverted table must also be updated to reflect the deletion of this record. The record should NOT be physically deleted from the file since that would require too much work.

A message confirming the deletion should be logged, including the address of the deleted record. If no match is found, a failure message should be logged.

`change` `<name>` `<phone>`

Assuming there is a record containing the specified name, its phone number field is replaced with the given string. A success message, including the record address, or a failure message, should be logged.

`hashtable`

Causes a formatted display of the hash table to be written to the log file. The display should list each cell of the table, indicating that the cell is empty, or is a tombstone, or showing the key value and address stored there.

`invertedtable`

Causes a formatted display of the inverted table to be written to the log file. The display should list each cell of the table, indicating that the cell is empty, or is a tombstone, or showing the key value and address stored there.

`exit`

This causes the indexing application to terminate. The command file is guaranteed to end with an exit command.

You may assume that the command file will conform to the given syntax, so syntactic error checking is not required. If an error occurs during the parsing of the command file, there's an error in your code. However, your program should still attempt to recover, by "flushing" the current command and proceeding to the next input line.

A sample command file will be posted shortly. In the meantime, you should be able to easily construct one for the given database file.

Log File Description:

Since this assignment will be graded by TAs, rather than the Curator, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct. The first two lines should contain your name, course (CS 2604), and project title.

After the indices are built from the database file, you should dump the contents of each index (formatted nicely) to the log file.

The remainder of the log file output should come directly from your processing of the command file. You are required to echo each command that you process to the log file so that it's easy to determine which command each section of your output corresponds to.

Submitting Your Program:

You will submit a zipped file containing your project to the Curator System (read the *Student Guide*), and it will be archived until you demo it for one of the GTAs. Instructions for submitting are contained in the *Student Guide*. You will find a list of the required contents for the zipped file on the course website. Follow the instructions there carefully; it is very common for students to suffer a loss of points (often major) because they failed to include the specified items.

Be very careful to include all the necessary source code files. It is amazingly common for students to omit required header or cpp files. In such a case, it is obviously impossible to perform a test of the submitted program unless the student is allowed to supply the missing files. When that happens, to be fair to other students, we must assess the late penalty that would apply at the time of the demo.

You will be allowed up to five submissions for this assignment, in case you need to correct mistakes. Test your program thoroughly before submitting it. If you discover an error you may fix it and make another submission. Your last submission will be graded, so fixing an error after the due date will result in a late penalty.

The submission client can be found at: http://spasm.cs.vt.edu:8080/curator/

Programming Standards:

The GTAs will be carefully evaluating your source code on this assignment for programming style, so you should observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Evaluation:

You will schedule a demo with your assigned GTA. At the demo, you will download your submitted project, perform a build, and run your program on the supplied test data. The GTA will evaluate the correctness of your results. In addition, the GTA will evaluate your project for good internal documentation and software engineering practice.

Here is an estimate of how much weight will be given to each of the factors that the GTA will consider may be added here later.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided with the earlier project specifications in the header comment for your main source code file.