Simple Hash Table

For this project you will implement a simple hash table using open hashing and a specific version of the quadratic probe strategy. In particular, you will use the quadratic function (k^2 + k)/2 to compute the probe steps.

The hash table will be used here to store structured records, and it should be implemented as a formal Java generic for reusability (by you on a future project).

The hash table must support all the usual functionality, as discussed in class. For this project we will focus primarily on building the initial hash table, searching by key value for records in the table, inserting new records to the table, and deleting old records. In order to determine if the hash table organization is correct, you will instrument the insert and search functions to display the probe sequence (indices visited in performing a search).

You should design your implementation so that changes to the probe strategy are painless.

Because this assignment will be auto-graded using a test harness we will provide, your implementation must conform to the public interface below, and include at least all of the package, public and private members that are shown. As usual, you may include any additional members and/or methods you find useful or necessary.

The interface `Hashable` and the enumerated type `slotState` are discussed at the end of this specification.

```java
// The test harness will belong to the following package; the hash table
// implementation will belong to it as well.  In addition, the hash table
// implementation will specify package access for all data members in order
// that the test harness may have access to them.
//
package Minor.P3.DS;

public class HashTable<T extends Hashable> {

   T[]          Table;         // stores data objects
   slotState[] Status;        // stores corresponding status values
   int          Size;          // dimension of Table
   int          Usage;         // # of data objects stored in Table
   FileWriter   Log;           // target of logged output
   boolean      loggingOn;     // write output iff this is true

   // Construct hash table with specified size.
   // Pre:
   //      Sz is a positive integer of the form 2^k.
   // Post:
   //      this.Table is an array of dimension Sz; all entries are null
   //      this.Status is an array of dimension Sz; all entries are EMPTY
   //      this.Usage == 0
   //      this.Log == null
   //      this.loggingOn == false
   //
   public HashTable(int Sz) { . . . }
```

```
// Attempt insertion of Elem.
// Pre:
//      Elem is a proper object of type T
// Post:
//      If Elem already occurs in Table (according to equals()):
//         this.Table is unchanged
//         this.Usage is unchanged
//      Otherwise:
//         Elem is added to Table
//         this.Usage is incremented
//      If loggingOn == true:
//         indices accessed during search are written to Log and
//         success/failure message is written to Log
// Return: reference to inserted object or null if insertion fails
//
public T Insert(T Elem) throws IOException { . . . }

// Search Table for match to Elem (according to equals()).
// Pre:
//      Elem is a proper object of type T
// Post:
//      No member of the hash table object is changed.
//      If loggingOn == true:
//         indices accessed during search are written to Log and
//         success/failure message is written to Log
// Return reference to matching data object, or null if no match
//      is found.
public T Find(T Elem) throws IOException { . . . }

// Delete data object that matches Elem.
// Pre:
//      Elem is a proper object of type t
// Post:
//      If Elem does not occur in Table (according to equals()):
//         this.Table is unchanged
//         this.Usage is unchanged
//      Otherwise:
//         matching reference in Table is null
//         this.Usage is decremented
//      If loggingOn == true:
//         indices accessed during search are written to Log and
//         success/failure message is written to Log
// Return reference to deleted object, or null if not found.
public T Delete(T Elem) throws IOException { . . . }

// Reset hash table to (almost) same state as when first constructed.
// Post:
//      this.Table is an array of dimension Sz; all entries are null
//      this.Status is an array of dimension Sz; all entries are
//          EMPTY
//      this.Opt is unchanged
//      this.Usage == 0
//      this.Log  is unchanged
//      this.loggingOn is unchanged
//
public void Clear() { . . . }
```

```
   // Reset hash table's log output target.
   // Pre:
   //      Log is an open on a file, or is null
   // Post:
   //      If Log is null, no changes are made.
   //      Otherwise: this.Log == Log
   // Return true iff this.Log has been changed and is not null.
   //
   public boolean setLog(FileWriter Log) { . . . }

   // Turn internal logging on.
   // Post:
   //      If this.Log is not null, loggingOn == true
   //      Otherwise, loggingOn == false
   // Returns final value of loggingOn.
   //
   public boolean logOn() { . . . }

   // Turn internal logging off.
   // Post:
   //      loggingOn == false
   // Returns final value of loggingOn.
   //
   public boolean logOff() { . . . }
}
```

The test driver for your hash table will begin by hashing a set of records into the table, and then executing a sequence of searches, deletions and insertions on the table.

Data Structures:

Your hash table implementation is under the following specific requirements:

- You must encapsulate the hash table as a formal Java generic. The hash table should not supply a hash function; the hash function will be supplied by the data objects that are stored in the table.
- On insertion, tombstones should be "recycled". That is, if the insertion search passed any tombstones before finding an empty slot, the new value should be inserted in the first slot that contained a tombstone.
- The underlying physical structure must be a simple, dynamically allocated array. There is no requirement the hash table provide a resizing mechanism.

Most of the rest of the specification describes the environment in which your submission will be tested. Strictly speaking, you can ignore those parts, but they may give you ideas about your own testing.

Command File Description:

The test harness will read commands from a file. You do not have to implement any handling for this file, but we will describe it anyway, since samples will be supplied and you may wish to adopt them for your own testing. Lines beginning with a semicolon (`';'`) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the command file.

Each 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.

`insert``<tab><name><tab><street address><tab><city>`

This will cause a record containing the given data to be inserted to the hash table, if possible. For each insert command you must log the indices of the slots that are accessed during the search, followed by "inserted". If the insertion fails, log "duplicate" or simply "not inserted", as appropriate. Note: if quadratic probing is used, it is possible that an insertion will fail even if there is an empty slot in the table.

`delete``<tab><name>`

This will cause the record containing the given name to be deleted from the hash table, if a matching record occurs there. For each delete command you must log the indices of the slots that are accessed during the search, followed by "deleted". If the insertion fails, log "not found".

`find``<tab><name>`

This causes the hash table to be searched for a record containing the given name. You must log the indices of the slots that are accessed during the search, followed by "found", or if the insertion fails, by "not found".

`show``<space>[`**`full`**` | `**`empty`**` | `**`tombstones`**`]`

This causes the hash table to log the indices of all the slots that are in the specified state. If no slots are in that state, log "none".

`clear`

This causes the hash table to restore itself to its initial, empty state, switch from its current probe strategy to the other one, and log "table reset".

`dump`

This causes the hash table to display itself to the log. This is only for your testing purposes; the command files actually used by the Curator will NOT specify this command, but it may be included in sample data files.

`exit`

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

Legend: in the commands above:

`<name>`	a string of arbitrary length
`<street address>`	a string of arbitrary length
`<city>`	a string of arbitrary length

You may assume that the command file will conform to the given syntax, so syntactic error checking is not required. Sample command files will be available on the website.

Log File Description:

The test harness will write its output to a file named `HashOut.txt`. Your implementation will not write output to this file, aside from logging table indices that are accessed during insertion, deletion and search operations. In those cases, you must match the formatting shown in the sample logs on the website.

Testing:

We will be testing your implementation with our own test driver. We may (or may not) release information about that driver before the assignment is due. In any case, it is your responsibility to design and carry out a sensible test of your implementation before submitting it. For that purpose, you may share test code (**but absolutely no code for the hash table itself!!**) via the class Forum.

Evaluation:

You should document your implementation in accordance with the *Programming Standards* page on the course website. It is possible that your implementation will be evaluated for documentation and design, as well as for correctness of results. If so, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

What to turn in and how:

This assignment will be auto-graded using a test harness on the Curator system. The testing will be done under Windows (which should not matter at all) using Java version 1.7.0.

Submit a single `.java` file containing your `HashTable` generic to the Curator System. Submit nothing else. Your solution should not write anything to standard output.

Your submitted source file will be placed in the appropriate subdirectory with the packaged test code, and will then be compiled with the test harness.

Instructions, and the appropriate link, for submitting to the Curator are given in the *Student Guide* at the Curator website:

`http://www.cs.vt.edu/curator/`.

You will be allowed to submit your solution multiple times; the highest score will be counted.

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 source code file.

Other issues:

There must be some way to product hash values for the data objects that are inserted to the hash table. In this assignment, that will be accomplished by requiring that those data objects extend the following interface:

```
public interface Hashable {

   public int Hash();
}
```

Note that the method `Hash()` does not know the table size and therefore does not necessarily return a valid table index. It is the responsibility of the client code, in this case that's the `HashTable` implementation, to compute a valid table index.

The enumerated type `slotState` is used to support the use of sensible labels for state information within the hash table implementation. They are declared as follows:

```
package MinorP3.DS;

public enum slotState {EMPTY, FULL, TOMBSTONE};
```