Simple Hash Table

For this project you will implement a simple hash table using open hashing and a hybrid probe strategy. 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 hash function and probe strategy are painless.

Be careful about infinite loops when probing. Also be careful of the possibility that quadratic probing may fail to find an empty slot. If quadratic probing has not found an empty slot after probing $\lfloor N/2 \rfloor$ locations, where N is the number of slots in the hash table, abandon it and apply linear probing, starting at the home slot where the collision occurred.

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 types `slotState` and `probeOption` 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 MinorP3.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
   probeOption Opt;          // current probing strategy

   // Construct hash table with specified size and probe strategy.
   // Pre:
   //      Sz is a positive integer, Opt is LINEAR or QUADRATIC.
   // 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 == Opt
   //      this.Usage == 0
   //
   public HashTable(int Sz, probeOption Opt) { . . . }
```

```
// 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
//          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.
//       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() { . . . }
```

```
   // Provides a useful display of the buffer pool's contents.
   // Pre:
   //       Log is open on an output file
   // Post:
   //       HashTable object is unchanged
   public void Display(FileWriter Log) throws IOException {

       if ( Usage == 0 ) {
           Log.write("Hash table is empty.\n");
           return;
       }

       for (int pos = 0; pos < Size; pos++) {
           if ( Status[pos] == slotState.FULL ) {
               Log.write(pos + ":  " + Table[pos] + "\n");
           }
           else if ( Status[pos] == slotState.TOMBSTONE ) {
               Log.write(pos + ":  tombstone" + "\n");
           }
           else {
               Log.write(pos + ":  empty" + "\n");
           }
       }
   }
}
```

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.

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 buffer pool 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 1.6.18.

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 produce 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 long 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 your `Hashtable` implementation, to compute a valid table index.

The enumerated types `slotState` and `probeOption` are 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};
```

```
package MinorP3.DS;

public enum probeOption {LINEAR, QUADRATIC};
```