```
import java.io.*;

/****
 *
 * Class HashTableTest exercises the <a href= HashTable.html> HashTable </a>
 * class using a simple integer-valued hash table entry.  Tests are run in the
 * following phases:
 *                                                                      <pre>
 *
 *     Phase 1: Test the constructor, building tables of sizes 1, 5, 500, and
 *              the default size; confirm the size of each table.
 *
 *     Phase 2: Test the enter method, lookup, and delete methods on table of
 *              size 1.
 *
 *     Phase 3: Test enter method by filling up a table of size 5, including
 *              check of the TableFull exception.
 *
 *     Phase 4: Test the lookup method on the full table of size 5, expecting
 *              O(N) performance on lookups given full table.
 *
 *     Phase 5: Test the delete method on table of size 5, removing each entry.
 *
 *     Phase 6: Repeat phases 3 through 5 to exercise the somewhat tricky
 *              implementation of delete that uses active/inactive flags.
 *
 *     Phase 7: Successively test the enter, lookup, and delete methods on the
 *              the same table of size 5, with 0 through 7 entries.  Expect
 *              O(N) performance on lookups since entries are all marked as
 *              inactive instead of being null.
 *
 *     Phase 8: Rerun phase 7 on a new table of size 5, expecting O(c)
 *              performance on early lookups.
 *
 *     Phase 9: Test enter and lookup on a larger more sparsely populated
 *              table, expecting O(c) performance.
 *                                                                   </pre>
 *
 * @author Gene Fisher
 * @version 8may01
 *
 */

public class HashTableTest {

    /**
     * Allocate a HashTable and test the HashTable methods.
     */
    public static void main(String[] args)
            throws HashTableFull, HashIndexInvalid {

        /*
         * Test the constructor.
         */
        phase1();

        /*
         * Test minimum size table.
         */
        phase2();

        /*
         * Fill up table of size 5.
         */
        phase3(table5);

        /*
         * Lookup in Phase 3 table.
         */
        phase4(table5);

        /*
         * Delete all from Phase 3 table.
         */
        phase5(table5);

        /*
         * Rerun phases 3-5 to test tricky delete.
         */
        phase6(table5);

        /*
         * Successively call enter, lookup, delete with differing numbers of
         * entries.
         */
        phase7(table5);

        /*
         * Rerun Phase 7 on a new table, expecting better performance.
         */
        phase7(new HashTable(5));

        /*
         * Test on bigger sparse table.
         */
        phase9(table500);

    }

    /**
     * Construct tables of sizes 1, 5, 500, and the default size.
     */
    public static void phase1() {

        table1 = new HashTable(1);
        table5 = new HashTable(5);
        table500 = new HashTable(500);
        tableDefault = new HashTable();

        System.out.println(
            "Confirming table sizes of 1, 500, and 1000: " +
            table1.size + ", " + table5.size + ", " + table500.size + ", " +
```

```
            tableDefault.size);
    }

    /**
     * Test enter, lookup, and delete on table of size 1.
     */
    public static void phase2() throws HashTableFull, HashIndexInvalid {
        IntEntry entry = new IntEntry(0);

        table1.enter(entry);
        System.out.println("Size 1 table after entering 0: " + "\n" +
            table1.toString());
        System.out.println("Lookup in size 1 table: " +
            table1.lookup(new Integer(0))[0]);
        System.out.println("Size 1 table after delete: " +
            table1.delete(new Integer(0)));
    }

    /**
     * Test enter method by filling up a table of size 5, including check of
     * the TableFull exception.
     */
    protected static void phase3(HashTable table) {

        int i;
        IntEntry entry;

        /*
         * Dump table to confirm default initialization.  */
        System.out.println("\nTable dump before any activity:\n" +
            table.toString());

        /*
         * Test the enter method, including handling exceptions if they occur.
         */
        for (i = 0; i < 7; i++) {
            try {
                table.enter(new IntEntry(i*5));
                System.out.println("\nTable dump after entering key " +
                    (i*5) + ":\n" + table.toString());
            }
            catch (HashTableFull error) {
                System.out.println(
                    "Hash table full at entry attempt " + i*5);
            }
            catch (HashIndexInvalid error) {
                System.out.println(
                    "Invalid hash index value of: " + error.index);
            }
        }

    }

    /**
     * Test the lookup method on the full table of size 5, expecting O(N)
     * performance on lookups given full table.
     */
    protected static void phase4(HashTable table) {

        int i;
        IntEntry entry;
        HashTableEntry[] entries;

        /*
         * Test the lookup method.
         */
        System.out.println();
        for (i = 0; i < 7; i++) {
            entries = table.lookup(new Integer(i*5));
            if (entries != null)
                entry = (IntEntry) entries[0];
            else
                entry = null;
            System.out.println("Value of entry for key " + i*5 + ": " +
                (entry != null ? entry.toString() : "null"));
        }

    }

    /**
     * Test the delete method on table of size 5, removing each entry.
     */
    protected static void phase5(HashTable table) {

        int i;

        /*
         * Test the delete method.
         */
        System.out.println();
        for (i = 0; i < 7; i++) {
            table.delete(new Integer(i*5));
            System.out.println("\nTable dump after delete of key " +
                (i*5) + "\n" + table.toString());
        }

    }

    /**
     * Repeat phases 3 through 5 to exercise the somewhat tricky implementation
     * of delete that uses active/inactive flags.
     */
    protected static void phase6(HashTable table) {
        phase3(table);
        phase4(table);
        phase5(table);
    }

    /**
     * Successively test the enter, lookup, and delete methods on the the same
     * table of size 5, with 0 through 7 entries.  Expect O(N) performance on
     * lookups since entries are all marked as inactive instead of being null.
```

```
225      */
226     protected static void phase7(HashTable table) {
227
228         int i;
229         IntEntry entry;
230         HashTableEntry[] entries;
231
232         entries = table.lookup(new Integer(0));
233         System.out.println(
234             "\nValue of lookup in empty table should be null: " + entries);
235
236         for (i = 0; i < 7; i++) {
237
238             try {
239                 table.enter(new IntEntry(i*5));
240                 System.out.println("\nTable dump after entering key " +
241                     (i*5) + ":\n" + table.toString());
242             }
243             catch (HashTableFull error) {
244                 System.out.println(
245                     "Hash table full at entry attempt " + i*5);
246             }
247             catch (HashIndexInvalid error) {
248                 System.out.println(
249                     "Invalid hash index value of: " + error.index);
250             }
251
252             entries = table.lookup(new Integer(i*5));
253             if (entries != null)
254                 entry = (IntEntry) entries[0];
255             else
256                 entry = null;
257             System.out.println("Value of entry key for " + i*5 + ": " +
258                 (entry != null ? entry.toString() : "null"));
259         }
260
261     }
262
263     /**
264      * Test enter and lookup on a larger more sparsely populated table,
265      * expecting O(c) performance.
266      */
267     protected static void phase9(HashTable table)
268             throws HashTableFull, HashIndexInvalid {
269
270         int i;
271         IntEntry entry;
272         HashTableEntry[] entries;
273
274         System.out.println("Entering 249");
275         table.enter(entry = new IntEntry(249));
276         dumpSingleLookupResults(table, new Integer(249));
277
278         System.out.println("Entering 749");
279         table.enter(entry = new IntEntry(249 + 500));
280         dumpSingleLookupResults(table, new Integer(249 + 500));
281
282         System.out.println("Entering 749 again");
283         table.enter(entry = new IntEntry(249 + 500));
284         dumpMultiLookupResults(table, new Integer(249 + 500), 2);
285
286         System.out.println("Entering 749 a third time");
287         table.enter(entry = new IntEntry(249 + 500));
288         dumpMultiLookupResults(table, new Integer(249 + 500), 3);
289
290         System.out.println("Entering 250");
291         table.enter(entry = new IntEntry(250));
292         dumpSingleLookupResults(table, new Integer(250));
293
294         System.out.println("\n");
295     }
296
297     /**
298      * Dump the result of what is expected to be a single-value lookup.
299      */
300     protected static void dumpSingleLookupResults(
301             HashTable table, Integer key) {
302
303         int i;
304         IntEntry entry;
305         HashTableEntry[] entries;
306
307         System.out.println("\nLooking in the following table for " +
308             key + ":\n" + table.toStringNonNull());
309
310         entries = table.lookup(key);
311         entry = (IntEntry) entries[0];
312         System.out.println("\nSingle lookup result should be " + key +
313             ",1: " + entry.toString() + "," + entries.length + "\n");
314
315     }
316
317     /**
318      * Dump the result of what is expected to be a n-value lookup.
319      */
320     protected static void dumpMultiLookupResults(
321             HashTable table, Integer key, int n) {
322
323         int i;
324         IntEntry entry;
325         HashTableEntry[] entries;
326
327         System.out.println("\nLooking in the following table for " +
328             key + ":\n" + table.toStringNonNull());
329
330         entries = table.lookup(key);
331         System.out.print("\nMulti lookup result should be " + key +
332             " repeated " + n + " times: ");
333         for (i = 0; i < entries.length; i++) {
334             entry = (IntEntry) entries[i];
335             System.out.print(entry.toString() + " ");
336         }
```

```
337         System.out.println("\n");
338
339     }
340
341
342     /***
343      * Inner class IntEntry is a very simple implementation of HashTableEntry
344      * with a single Integer field.  This field serves as both the key and
345      * value for the entry.
346      */
347     protected static class IntEntry implements HashTableEntry {
348
349         /**
350          * Construct an entry with the given int value.
351          */
352         public IntEntry(int value) {
353             this.value = new Integer(value);
354         }
355
356         /**
357          * Return the key value of this entry, which is just the single Integer
358          * value.
359          */
360         public Object getKey() {
361             return value;
362         }
363
364         /**
365          * Compute the hash function of this entry as (key + 2) % tableSize.
366          * The "+ 2" part is to facilitate testing by having values hash to
367          * some other table location than themselves.
368          */
369         public int hash(Object key, int tableSize) {
370             return (((Integer) key).intValue() + 2) % tableSize;
371         }
372
373         /**
374          * Return the value of this entry as a string.
375          */
376         public String toString() {
377             return value.toString();
378         }
379
380         /** The single Integer value of this hash table entry */
381         protected Integer value;
382
383     }
384
385
386     /** Testing tables */
387     protected static HashTable
388         table1, table5, table500, tableDefault;
389
390 }
```