```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Random;

import components.map.Map;
import components.map.Map1L;
import components.map.MapSecondary;

/**
 * {@code Map} represented as a hash table using {@code Map}s for the buckets,
 * with implementations of primary methods.
 *
 * @param <K>
 *            type of {@code Map} domain (key) entries
 * @param <V>
 *            type of {@code Map} range (associated value) entries
 * @convention <pre>
 * |$this.hashTable| > 0  and
 * for all i: integer, pf: PARTIAL_FUNCTION, x: K
 *     where (0 <= i  and  i < |$this.hashTable|  and
 *            <pf> = $this.hashTable[i, i+1)  and
 *            x is in DOMAIN(pf))
 *   ([computed result of x.hashCode()] mod |$this.hashTable| = i))  and
 * for all i: integer
 *     where (0 <= i  and  i < |$this.hashTable|)
 *   ([entry at position i in $this.hashTable is not null])  and
 * $this.size = sum i: integer, pf: PARTIAL_FUNCTION
 *     where (0 <= i  and  i < |$this.hashTable|  and
 *            <pf> = $this.hashTable[i, i+1))
 *   (|pf|)
 * </pre>
 * @correspondence <pre>
 * this = union i: integer, pf: PARTIAL_FUNCTION
 *            where (0 <= i  and  i < |$this.hashTable|  and
 *                   <pf> = $this.hashTable[i, i+1))
 *          (pf)
 * </pre>
 *
 * @author Zhuoyang Li + Xinci Ma
 *
 */
public class Map4<K, V> extends MapSecondary<K, V> {

```

```
44      /*
45       * Private members
    ------------------------------------------------------------
46       */
47
48      /**
49       * Default size of hash table.
50       */
51      private static final int DEFAULT_HASH_TABLE_SIZE = 101;
52
53      /**
54       * Buckets for hashing.
55       */
56      private Map<K, V>[] hashTable;
57
58      /**
59       * Total size of abstract {@code this}.
60       */
61      private int size;
62
63      /**
64       * Computes {@code a} mod {@code b} as % should have been
    defined to work.
65       *
66       * @param a
67       *            the number being reduced
68       * @param b
69       *            the modulus
70       * @return the result of a mod b, which satisfies 0 <=
    {@code mod} < b
71       * @requires b > 0
72       * @ensures <pre>
73       * 0 <= mod  and  mod < b  and
74       * there exists k: integer (a = k * b + mod)
75       * </pre>
76       */
77      private static int mod(int a, int b) {
78          assert b > 0 : "Violation of: b > 0";
79          int result = a % b;
80          if (result < 0) {
81              result += b;
82          }
83
84          return result;
85      }
86
87      /**
```

```
 88      * Creator of initial representation.
 89      *
 90      * @param hashTableSize
 91      *            the size of the hash table
 92      * @requires hashTableSize > 0
 93      * @ensures <pre>
 94      * |$this.hashTable| = hashTableSize  and
 95      * for all i: integer
 96      *     where (0 <= i  and  i < |$this.hashTable|)
 97      *   ($this.hashTable[i, i+1) = <{}>)  and
 98      * $this.size = 0
 99      * </pre>
100      */
101     @SuppressWarnings("unchecked")
102     private void createNewRep(int hashTableSize) {
103         /*
104          * With "new Map<K, V>[...]" in place of "new
    Map[...]" it does not
105          * compile; as shown, it results in a warning about
    an unchecked
106          * conversion, though it cannot fail.
107          */
108         this.hashTable = new Map[hashTableSize];
109         for (int i = 0; i < hashTableSize; i++) {
110             this.hashTable[i] = new Map1L<K, V>();
111         }
112         this.size = 0;
113
114     }
115
116     /*
117      * Constructors
    -----------------------------------------------------------
118      */
119
120     /**
121      * No-argument constructor.
122      */
123     public Map4() {
124
125         this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
126
127     }
128
129     /**
130      * Constructor resulting in a hash table of size {@code
    hashTableSize}.
```

```
131      *
132      * @param hashTableSize
133      *            size of hash table
134      * @requires hashTableSize > 0
135      * @ensures this = {}
136      */
137     public Map4(int hashTableSize) {
138
139         this.createNewRep(hashTableSize);
140
141     }
142
143     /*
144      * Standard methods
    ------------------------------------------------------------
145      */
146
147     @SuppressWarnings("unchecked")
148     @Override
149     public final Map<K, V> newInstance() {
150         try {
151             return
    this.getClass().getConstructor().newInstance();
152         } catch (ReflectiveOperationException e) {
153             throw new AssertionError(
154                     "Cannot construct object of type " +
    this.getClass());
155         }
156     }
157
158     @Override
159     public final void clear() {
160         this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
161     }
162
163     @Override
164     public final void transferFrom(Map<K, V> source) {
165         assert source != null : "Violation of: source is not
    null";
166         assert source != this : "Violation of: source is not
    this";
167         assert source instanceof Map4<?, ?> : ""
168                 + "Violation of: source is of dynamic type
    Map4<?,?>";
169         /*
170          * This cast cannot fail since the assert above would
    have stopped
```

```
171          * execution in that case: source must be of dynamic
    type Map4<?,?>, and
172          * the ?,? must be K,V or the call would not have
    compiled.
173          */
174         Map4<K, V> localSource = (Map4<K, V>) source;
175         this.hashTable = localSource.hashTable;
176         this.size = localSource.size;
177         localSource.createNewRep(DEFAULT_HASH_TABLE_SIZE);
178     }
179 
180     /*
181      * Kernel methods
    ----------------------------------------------------------
182      */
183 
184     @Override
185     public final void add(K key, V value) {
186         assert key != null : "Violation of: key is not null";
187         assert value != null : "Violation of: value is not
    null";
188         assert !this.hasKey(key) : "Violation of: key is not
    in DOMAIN(this)";
189 
190         //use the hash function to find the index of the
    bucket
191         int index = mod(key.hashCode(),
    this.hashTable.length);
192         this.hashTable[index].add(key, value);
193         this.size++;
194     }
195 
196     @Override
197     public final Pair<K, V> remove(K key) {
198         assert key != null : "Violation of: key is not null";
199         assert this.hasKey(key) : "Violation of: key is in
    DOMAIN(this)";
200 
201         int index = mod(key.hashCode(),
    this.hashTable.length);
202         this.size--;
203         return this.hashTable[index].remove(key);
204     }
205 
206     @Override
207     public final Pair<K, V> removeAny() {
208         assert this.size() > 0 : "Violation of: this /=
```

```
    empty_set";
209
210         //use a random number generator to choose a bucket
211         Random rand = new Random();
212         int index = rand.nextInt(this.hashTable.length);
213         while (this.hashTable[index].size() == 0) {
214             index = rand.nextInt(this.hashTable.length);
215         }
216         this.size--;
217         return this.hashTable[index].removeAny();
218
219     }
220
221     @Override
222     public final V value(K key) {
223         assert key != null : "Violation of: key is not null";
224         assert this.hasKey(key) : "Violation of: key is in
    DOMAIN(this)";
225
226         //use the hash function to find the index of the
    bucket
227         int index = mod(key.hashCode(),
    this.hashTable.length);
228         return this.hashTable[index].value(key);
229
230     }
231
232     @Override
233     public final boolean hasKey(K key) {
234         assert key != null : "Violation of: key is not null";
235
236         int index = mod(key.hashCode(),
    this.hashTable.length);
237         return this.hashTable[index].hasKey(key);
238     }
239
240     @Override
241     public final int size() {
242         return this.size;
243     }
244
245     @Override
246     public final Iterator<Pair<K, V>> iterator() {
247         return new Map4Iterator();
248     }
249
250     /**
```

```
251     * Implementation of {@code Iterator} interface for
    {@code Map4}.
252     */
253    private final class Map4Iterator implements
    Iterator<Pair<K, V>> {
254
255        /**
256         * Number of elements seen already (i.e., |
    ~this.seen|).
257         */
258        private int numberSeen;
259
260        /**
261         * Bucket from which current bucket iterator comes.
262         */
263        private int currentBucket;
264
265        /**
266         * Bucket iterator from which next element will come.
267         */
268        private Iterator<Pair<K, V>> bucketIterator;
269
270        /**
271         * No-argument constructor.
272         */
273        Map4Iterator() {
274            this.numberSeen = 0;
275            this.currentBucket = 0;
276            this.bucketIterator =
    Map4.this.hashTable[0].iterator();
277        }
278
279        @Override
280        public boolean hasNext() {
281            return this.numberSeen < Map4.this.size;
282        }
283
284        @Override
285        public Pair<K, V> next() {
286            assert this.hasNext() : "Violation of:
    ~this.unseen /= <>";
287            if (!this.hasNext()) {
288                /*
289                 * Exception is supposed to be thrown in this
    case, but with
290                 * assertion-checking enabled it cannot
    happen because of assert
```

```
291                  * above.
292                  */
293                 throw new NoSuchElementException();
294             }
295             this.numberSeen++;
296             while (!this.bucketIterator.hasNext()) {
297                 this.currentBucket++;
298                 this.bucketIterator =
    Map4.this.hashTable[this.currentBucket]
299                         .iterator();
300             }
301             return this.bucketIterator.next();
302         }
303 
304         @Override
305         public void remove() {
306             throw new UnsupportedOperationException(
307                     "remove operation not supported");
308         }
309 
310     }
311 
312 }
313 
```