```java
import java.util.Iterator;

/**
 * {@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 Charan nanduri and Evan Frisbie
 *
 */
public class Map4<K, V> extends MapSecondary<K, V> {

    /*
     * Private members
    ------------------------------------------------------------
     */

    /**
```

```
48      * Default size of hash table.
49      */
50     private static final int DEFAULT_HASH_TABLE_SIZE = 101;
51
52     /**
53      * Buckets for hashing.
54      */
55     private Map<K, V>[] hashTable;
56
57     /**
58      * Total size of abstract {@code this}.
59      */
60     private int size;
61
62     /**
63      * Computes {@code a} mod {@code b} as % should have been
   defined to work. -
64      * DONE
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         //possible way, although it is throwing an error.
80         /*
81          * if (a < 0) { a = -a; } if (b < 0) { b = -b; }
82          *
83          * int result = a % b;
84          */
85         // this way ensures mod instead of remainder, even with neg
   values.
86         return (((a % b) + b) % b);
87     }
88
89     /**
```

```
 90      * Creator of initial representation. - DONE
 91      *
 92      * @param hashTableSize
 93      *            the size of the hash table
 94      * @requires hashTableSize > 0
 95      * @ensures <pre>
 96      * |$this.hashTable| = hashTableSize  and
 97      * for all i: integer
 98      *     where (0 <= i  and  i < |$this.hashTable|)
 99      *   ($this.hashTable[i, i+1) = <{}>)  and
100      * $this.size = 0
101      * </pre>
102      */
103     @SuppressWarnings("unchecked")
104     private void createNewRep(int hashTableSize) {
105         /*
106          * With "new Map<K, V>[...]" in place of "new Map[...]" it
    does not
107          * compile; as shown, it results in a warning about an
    unchecked
108          * conversion, though it cannot fail.
109          */
110         this.hashTable = new MapSecondary[hashTableSize];
111         this.size = 0;
112         for (int i = 0; i < hashTableSize; i++) {
113             this.hashTable[i] = new Map1L<K, V>();
114         }
115
116     }
117
118     /*
119      * Constructors
    ------------------------------------------------------------
120      */
121
122     /**
123      * No-argument constructor. - DONE
124      */
125     public Map4() {
126
127         this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
128
129     }
130
131     /**
```

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

```
    >";
173         /*
174          * This cast cannot fail since the assert above would have
    stopped
175          * execution in that case: source must be of dynamic type
    Map4<?,?>, and
176          * the ?,? must be K,V or the call would not have compiled.
177          */
178         Map4<K, V> localSource = (Map4<K, V>) source;
179         this.hashTable = localSource.hashTable;
180         this.size = localSource.size;
181         localSource.createNewRep(DEFAULT_HASH_TABLE_SIZE);
182     }
183
184     /*
185      * Kernel methods
    ------------------------------------------------------------
186      */
187
188     // DONE
189     @Override
190     public final void add(K key, V value) {
191         assert key != null : "Violation of: key is not null";
192         assert value != null : "Violation of: value is not null";
193         assert !this.hasKey(key) : "Violation of: key is not in
    DOMAIN(this)";
194
195         // Hash the key and get a location from mod
196         int location = mod(key.hashCode(), this.hashTable.length);
197         this.hashTable[location].add(key, value);
198         this.size++;
199     }
200
201     // DONE
202     @Override
203     public final Pair<K, V> remove(K key) {
204         assert key != null : "Violation of: key is not null";
205         assert this.hasKey(key) : "Violation of: key is in
    DOMAIN(this)";
206
207         Pair<K, V> result = null;
208
209         int location = mod(key.hashCode(), this.hashTable.length);
210         Map<K, V> hashList = this.hashTable[location];
211
```

```
212         if (hashList == null) {
213             result = null;
214         } else {
215             result = hashList.remove(key);
216             this.size -= 1;
217         }
218         return result;
219         //return this.hashTable[mod(key.hashCode(),
    this.hashTable.length)]
220         //         .remove(key); could also work
221     }
222 
223     // DONE
224     @Override
225     public final Pair<K, V> removeAny() {
226         assert this.size() > 0 : "Violation of: this /= empty_set";
227 
228         int i = 0;
229         Map<K, V> hashList = this.hashTable[i];
230         while (hashList.size() == 0) {
231             i++;
232             hashList = this.hashTable[i];
233         }
234         this.size--;
235         return hashList.removeAny();
236     }
237 
238     // DONE
239     @Override
240     public final V value(K key) {
241         assert key != null : "Violation of: key is not null";
242         assert this.hasKey(key) : "Violation of: key is in
    DOMAIN(this)";
243 
244         int length = this.hashTable.length;
245         int location = mod(key.hashCode(), length);
246 
247         V result = this.hashTable[location].value(key);
248 
249         return result;
250     }
251 
252     // DONE
253     @Override
254     public final boolean hasKey(K key) {
```

```
255         assert key != null : "Violation of: key is not null";
256
257         int length = this.hashTable.length;
258         int hashed = key.hashCode();
259         int location = mod(hashed, length);
260
261         boolean result = false;
262
263         result = this.hashTable[location].hasKey(key);
264
265         return result;
266     }
267
268     // DONE
269     @Override
270     public final int size() {
271
272         int result = 0;
273
274         for (int i = 0; i < this.hashTable.length; i++) {
275             if (this.hashTable[i].size() > 0) {
276                 result += this.hashTable[i].size();
277             }
278         }
279
280         return result;
281     }
282
283     @Override
284     public final Iterator<Pair<K, V>> iterator() {
285         return new Map4Iterator();
286     }
287
288     /**
289      * Implementation of {@code Iterator} interface for {@code
    Map4}.
290      */
291     private final class Map4Iterator implements Iterator<Pair<K,
    V>> {
292
293         /**
294          * Number of elements seen already (i.e., |~this.seen|).
295          */
296         private int numberSeen;
297
```

```
298         /**
299          * Bucket from which current bucket iterator comes.
300          */
301         private int currentBucket;
302 
303         /**
304          * Bucket iterator from which next element will come.
305          */
306         private Iterator<Pair<K, V>> bucketIterator;
307 
308         /**
309          * No-argument constructor.
310          */
311         Map4Iterator() {
312             this.numberSeen = 0;
313             this.currentBucket = 0;
314             this.bucketIterator =
    Map4.this.hashTable[0].iterator();
315         }
316 
317         @Override
318         public boolean hasNext() {
319             return this.numberSeen < Map4.this.size;
320         }
321 
322         @Override
323         public Pair<K, V> next() {
324             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
325             if (!this.hasNext()) {
326                 /*
327                  * Exception is supposed to be thrown in this case,
    but with
328                  * assertion-checking enabled it cannot happen
    because of assert
329                  * above.
330                  */
331                 throw new NoSuchElementException();
332             }
333             this.numberSeen++;
334             while (!this.bucketIterator.hasNext()) {
335                 this.currentBucket++;
336                 this.bucketIterator =
    Map4.this.hashTable[this.currentBucket]
337                         .iterator();
```

```
338             }
339             return this.bucketIterator.next();
340         }
341 
342         @Override
343         public void remove() {
344             throw new UnsupportedOperationException(
345                     "remove operation not supported");
346         }
347 
348     }
349 
350 }
351 
```