

```

1 import java.util.Iterator;
2
3 /**
4  * {@code Map} represented as a hash table using {@code Map}s for the buckets,
5  * with implementations of primary methods.
6  *
7  * @param <K>
8  *     type of {@code Map} domain (key) entries
9  * @param <V>
10 *     type of {@code Map} range (associated value) entries
11 *
12 * @convention <pre>
13 * |$this.hashTable| > 0 and
14 * for all i: integer, pf: PARTIAL_FUNCTION, x: K
15 *     where (0 <= i and i < |$this.hashTable| and
16 *           <pf> = $this.hashTable[i, i+1) and
17 *           x is in DOMAIN(pf))
18 *     ([computed result of x.hashCode()] mod |$this.hashTable| = i)) and
19 * for all i: integer
20 *     where (0 <= i and i < |$this.hashTable|)
21 *     ([entry at position i in $this.hashTable is not null]) and
22 * $this.size = sum i: integer, pf: PARTIAL_FUNCTION
23 *     where (0 <= i and i < |$this.hashTable| and
24 *           <pf> = $this.hashTable[i, i+1))
25 *           (|pf|)
26 * </pre>
27 *
28 * @correspondence <pre>
29 * this = union i: integer, pf: PARTIAL_FUNCTION
30 *     where (0 <= i and i < |$this.hashTable| and
31 *           <pf> = $this.hashTable[i, i+1))
32 *           (pf)
33 * </pre>
34 *
35 * @author Gabe Azzarita and Ty Fredrick
36 *
37 */
38 public class Map4<K, V> extends MapSecondary<K, V> {
39
40     /*
41      * Private members -----
42      */
43
44     /**
45      * Default size of hash table.
46      */
47     private static final int DEFAULT_HASH_TABLE_SIZE = 101;
48
49     /**
50      * Buckets for hashing.
51      */
52     private Map<K, V>[] hashTable;
53
54     /**
55      * Total size of abstract {@code this}.
56      */
57     private int size;
58
59     /**
60      * Computes {@code a} mod {@code b} as % should have been defined to work.
61      */

```

```

65     * @param a
66     *         the number being reduced
67     * @param b
68     *         the modulus
69     * @return the result of a mod b, which satisfies  $0 \leq \{\text{@code mod}\} < b$ 
70     * @requires b > 0
71     * @ensures <pre>
72     * 0 <= mod and mod < b and
73     * there exists k: integer (a = k * b + mod)
74     * </pre>
75     */
76     private static int mod(int a, int b) {
77         assert b > 0 : "Violation of: b > 0";
78
79         int temp = a;
80
81         if (temp > 0) {
82             while (temp >= b) {
83                 temp -= b;
84             }
85         } else {
86             while (temp < 0) {
87                 temp += b;
88             }
89         }
90
91         return temp;
92     }
93
94     /**
95     * Creator of initial representation.
96     *
97     * @param hashTableSize
98     *         the size of the hash table
99     * @requires hashTableSize > 0
100    * @ensures <pre>
101    * |$this.hashTable| = hashTableSize and
102    * for all i: integer
103    *   where (0 <= i and i < |$this.hashTable|) acce
104    *   ($this.hashTable[i, i+1) = <{}>) and
105    * $this.size = 0
106    * </pre>
107    */
108    @SuppressWarnings("unchecked")
109    private void createNewRep(int hashTableSize) {
110        /*
111         * With "new Map<K, V>[...]" in place of "new Map[...]" it does not
112         * compile; as shown, it results in a warning about an unchecked
113         * conversion, though it cannot fail.
114         */
115        this.hashTable = new Map[hashTableSize];
116
117        // Initialize maps
118        for (int i = 0; i < hashTableSize; i++) {
119            this.hashTable[i] = new Map1L<K, V>();
120        }
121    }
122 }
123

```

```

124    /*
125     * Constructors -----
126     */
127
128    /**
129     * No-argument constructor.
130     */
131    public Map4() {
132        this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
133    }
134
135    /**
136     * Constructor resulting in a hash table of size {@code hashTableSize}.
137     *
138     * @param hashTableSize
139     *         size of hash table
140     * @requires hashTableSize > 0
141     * @ensures this = {}
142     */
143    public Map4(int hashTableSize) {
144        this.createNewRep(hashTableSize);
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  @Override
189  public final void add(K key, V value) {
190      assert key != null : "Violation of: key is not null";
191      assert value != null : "Violation of: value is not null";
192      assert !this.hasKey(key) : "Violation of: key is not in DOMAIN(this)";
193
194      // bucket found using mod function and is index for array
195      int bucket = mod(key.hashCode(), this.hashTable.length);
196      // increase total size of map
197      this.size++;
198      this.hashTable[bucket].add(key, value);
199
200  }
201
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      // bucket found using mod function and is index for array
208      int bucket = mod(key.hashCode(), this.hashTable.length);
209      // decrease total size of map
210      this.size--;
211      return this.hashTable[bucket].remove(key);
212  }
213
214  @Override
215  public final Pair<K, V> removeAny() {
216      assert this.size() > 0 : "Violation of: this != empty_set";
217
218      // Variables needed for while loop
219      int i = 0;
220      boolean mapFound = false;
221
222      // Loop until we find non-empty map, storing index for later
223      while (i < this.hashTable.length && !mapFound) {
224          if (this.hashTable[i].size() > 0) {
225              mapFound = true;
226          } else {
227              i++;
228          }
229      }
230
231      // decrease total size of map
232      this.size--;
233      return this.hashTable[i].removeAny();
234  }
235
236  @Override
237  public final V value(K key) {
238      assert key != null : "Violation of: key is not null";
239      assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";
240
241      // bucket found using mod function and is index for array
```

```
242         int bucket = mod(key.hashCode(), this.hashTable.length);
243         return this.hashTable[bucket].value(key);
244     }
245
246     @Override
247     public final boolean hasKey(K key) {
248         assert key != null : "Violation of: key is not null";
249
250         // bucket found using mod function and is index for array
251         int bucket = mod(key.hashCode(), this.hashTable.length);
252         return this.hashTable[bucket].hasKey(key);
253     }
254
255
256     @Override
257     public final int size() {
258         return this.size;
259     }
260
261     @Override
262     public final Iterator<Pair<K, V>> iterator() {
263         return new Map4Iterator();
264     }
265
266     /**
267      * Implementation of {@code Iterator} interface for {@code Map4}.
268      */
269     private final class Map4Iterator implements Iterator<Pair<K, V>> {
270
271         /**
272          * Number of elements seen already (i.e., |~this.seen|).
273          */
274         private int numberSeen;
275
276         /**
277          * Bucket from which current bucket iterator comes.
278          */
279         private int currentBucket;
280
281         /**
282          * Bucket iterator from which next element will come.
283          */
284         private Iterator<Pair<K, V>> bucketIterator;
285
286         /**
287          * No-argument constructor.
288          */
289         Map4Iterator() {
290             this.numberSeen = 0;
291             this.currentBucket = 0;
292             this.bucketIterator = Map4.this.hashTable[0].iterator();
293         }
294
295         @Override
296         public boolean hasNext() {
297             return this.numberSeen < Map4.this.size;
298         }
299
300         @Override
```

```
301     public Pair<K, V> next() {
302         assert this.hasNext() : "Violation of: ~this.unseen /= <>";
303         if (!this.hasNext()) {
304             /*
305              * Exception is supposed to be thrown in this case, but with
306              * assertion-checking enabled it cannot happen because of assert
307              * above.
308              */
309             throw new NoSuchElementException();
310         }
311         this.numberSeen++;
312         while (!this.bucketIterator.hasNext()) {
313             this.currentBucket++;
314             this.bucketIterator = Map4.this.hashTable[this.currentBucket]
315                 .iterator();
316         }
317         return this.bucketIterator.next();
318     }
319
320     @Override
321     public void remove() {
322         throw new UnsupportedOperationException(
323             "remove operation not supported");
324     }
325
326 }
327
328 }
329
```