```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 Qinuo Shi & Yiming Cheng
 *
 */
public class Map4<K, V> extends MapSecondary<K, V> {

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

    /**
     * Default size of hash table.
     */
    private static final int DEFAULT_HASH_TABLE_SIZE = 101;

    /**
     * Buckets for hashing.
     */
    private Map<K, V>[] hashTable;

    /**
     * Total size of abstract {@code this}.
     */
    private int size;

    /**
```

```
63      * Computes {@code a} mod {@code b} as % should have been defined to work.
64      *
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 <= {@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 mod = a % b;
80
81         if (mod < 0) {
82             mod += b;
83         }
84
85         return mod;
86     }
87
88     /**
89      * Creator of initial representation.
90      *
91      * @param hashTableSize
92      *            the size of the hash table
93      * @requires hashTableSize > 0
94      * @ensures <pre>
95      * |$this.hashTable| = hashTableSize  and
96      * for all i: integer
97      *     where (0 <= i  and  i < |$this.hashTable|)
98      *   ($this.hashTable[i, i+1) = <{}>)  and
99      * $this.size = 0
100     * </pre>
101     */
102    @SuppressWarnings("unchecked")
103    private void createNewRep(int hashTableSize) {
104        /*
105         * With "new Map<K, V>[...]" in place of "new Map[...]" it does not
106         * compile; as shown, it results in a warning about an unchecked
107         * conversion, though it cannot fail.
108         */
109        this.size = 0;
110        this.hashTable = new Map[hashTableSize];
111
112        for (int i = 0; i < hashTableSize; i++) {
113            this.hashTable[i] = new Map2<>();
114        }
115
116    }
117
118    /*
119     * Constructors ------------------------------------------------------------
```

```
120     */
121
122    /**
123     * No-argument constructor.
124     */
125    public Map4() {
126
127        /*
128         * Create a representation
129         */
130        this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
131
132    }
133
134    /**
135     * Constructor resulting in a hash table of size {@code hashTableSize}.
136     *
137     * @param hashTableSize
138     *            size of hash table
139     * @requires hashTableSize > 0
140     * @ensures this = {}
141     */
142    public Map4(int hashTableSize) {
143
144        /*
145         * Use given number to create a representation
146         */
147        this.createNewRep(hashTableSize);
148
149    }
150
151    /*
152     * Standard methods ----------------------------------------------------
153     */
154
155    @SuppressWarnings("unchecked")
156    @Override
157    public final Map<K, V> newInstance() {
158        try {
159            return this.getClass().getConstructor().newInstance();
160        } catch (ReflectiveOperationException e) {
161            throw new AssertionError(
162                    "Cannot construct object of type " + this.getClass());
163        }
164    }
165
166    @Override
167    public final void clear() {
168        this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
169    }
170
171    @Override
172    public final void transferFrom(Map<K, V> source) {
173        assert source != null : "Violation of: source is not null";
174        assert source != this : "Violation of: source is not this";
175        assert source instanceof Map4<?, ?> : ""
176                + "Violation of: source is of dynamic type Map4<?,?>";
```

```
177         /*
178          * This cast cannot fail since the assert above would have stopped
179          * execution in that case: source must be of dynamic type Map4<?,?>, and
180          * the ?,? must be K,V or the call would not have compiled.
181          */
182         Map4<K, V> localSource = (Map4<K, V>) source;
183         this.hashTable = localSource.hashTable;
184         this.size = localSource.size;
185         localSource.createNewRep(DEFAULT_HASH_TABLE_SIZE);
186     }
187 
188     /*
189      * Kernel methods ----------------------------------------------------------
190      */
191 
192     @Override
193     public final void add(K key, V value) {
194         assert key != null : "Violation of: key is not null";
195         assert value != null : "Violation of: value is not null";
196         assert !this.hasKey(key) : "Violation of: key is not in DOMAIN(this)";
197 
198         int bucket = mod(key.hashCode(), this.hashTable.length);
199         this.hashTable[bucket].add(key, value);
200         this.size++;
201 
202     }
203 
204     @Override
205     public final Pair<K, V> remove(K key) {
206         assert key != null : "Violation of: key is not null";
207         assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";
208 
209         int bucket = mod(key.hashCode(), this.hashTable.length);
210         Pair<K, V> removePair = this.hashTable[bucket].remove(key);
211         this.size--;
212         return removePair;
213 
214     }
215 
216     @Override
217     public final Pair<K, V> removeAny() {
218         assert this.size() > 0 : "Violation of: this /= empty_set";
219 
220         int intBaskets = 0;
221         int removeInt = 0;
222         Pair<K, V> removePair;
223         /*
224          * Use loop to find one pair from hashtable to remove
225          */
226         while (intBaskets < this.hashTable.length) {
227             if (this.hashTable[intBaskets].size() != 0) {
228                 removeInt = intBaskets;
229                 intBaskets = this.hashTable.length;
230             }
231             intBaskets++;
232         }
233         removePair = this.hashTable[removeInt].removeAny();
```

```
234         this.size--;
235         return removePair;
236 
237     }
238 
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         /*
245          * Get the value
246          */
247         V value = this.hashTable[mod(key.hashCode(), this.hashTable.length)]
248                 .value(key);
249         return value;
250 
251     }
252 
253     @Override
254     public final boolean hasKey(K key) {
255         assert key != null : "Violation of: key is not null";
256 
257         /*
258          * directly return hasKey() to check whether there is "key" in it.
259          */
260         return this.hashTable[mod(key.hashCode(), this.hashTable.length)]
261                 .hasKey(key);
262 
263     }
264 
265     @Override
266     public final int size() {
267 
268         int size = this.size;
269 
270         return size;
271     }
272 
273     @Override
274     public final Iterator<Pair<K, V>> iterator() {
275         return new Map4Iterator();
276     }
277 
278     /**
279      * Implementation of {@code Iterator} interface for {@code Map4}.
280      */
281     private final class Map4Iterator implements Iterator<Pair<K, V>> {
282 
283         /**
284          * Number of elements seen already (i.e., |~this.seen|).
285          */
286         private int numberSeen;
287 
288         /**
289          * Bucket from which current bucket iterator comes.
290          */
```

```
291         private int currentBucket;
292
293         /**
294          * Bucket iterator from which next element will come.
295          */
296         private Iterator<Pair<K, V>> bucketIterator;
297
298         /**
299          * No-argument constructor.
300          */
301         Map4Iterator() {
302             this.numberSeen = 0;
303             this.currentBucket = 0;
304             this.bucketIterator = Map4.this.hashTable[0].iterator();
305         }
306
307         @Override
308         public boolean hasNext() {
309             return this.numberSeen < Map4.this.size;
310         }
311
312         @Override
313         public Pair<K, V> next() {
314             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
315             if (!this.hasNext()) {
316                 /*
317                  * Exception is supposed to be thrown in this case, but with
318                  * assertion-checking enabled it cannot happen because of assert
319                  * above.
320                  */
321                 throw new NoSuchElementException();
322             }
323             this.numberSeen++;
324             while (!this.bucketIterator.hasNext()) {
325                 this.currentBucket++;
326                 this.bucketIterator = Map4.this.hashTable[this.currentBucket]
327                         .iterator();
328             }
329             return this.bucketIterator.next();
330         }
331
332         @Override
333         public void remove() {
334             throw new UnsupportedOperationException(
335                     "remove operation not supported");
336         }
337
338     }
339
340 }
341
```