

```

1 import java.util.Iterator;
2
3 import components.map.Map;
4 import components.map.MapSecondary;
5 import components.queue.Queue;
6 import components.queue.Queue1L;
7
8 /**
9  * {@code Map} represented as a {@code Queue} of pairs with implementations of
10 * primary methods.
11 *
12 * @param <K>
13 *     type of {@code Map} domain (key) entries
14 * @param <V>
15 *     type of {@code Map} range (associated value) entries
16 * @convention <pre>
17 * for all key1, key2: K, value1, value2: V, str1, str2: string of (key, value)
18 *     where (str1 * <(key1, value1)> is prefix of $this.pairsQueue and
19 *           str2 * <(key2, value2)> is prefix of $this.pairsQueue and
20 *           str1 /= str2)
21 *     (key1 /= key2)
22 * </pre>
23 * @correspondence this = entries($this.pairsQueue)
24 */
25 public class Map2<K, V> extends MapSecondary<K, V> {
26
27     //private static final V = null;
28     /*
29     * Private members -----
30     */
31
32     /**
33     * Pairs included in {@code this}.
34     */
35     private Queue<Pair<K, V>> pairsQueue;
36
37     /**
38     * Finds pair with first component {@code key} and, if such exists, moves it
39     * to the front of {@code q}.
40     *
41     * @param <K>
42     *     type of {@code Pair} key
43     * @param <V>
44     *     type of {@code Pair} value
45     * @param q
46     *     the {@code Queue} to be searched
47     * @param key
48     *     the key to be searched for
49     * @updates q
50     * @ensures <pre>
51     *     perms(q, #q) and
52     *     if there exists value: V (<(key, value)> is substring of q)
53     *     then there exists value: V (<(key, value)> is prefix of q)
54     * </pre>
55     */
56     private static <K, V> void moveToFront(Queue<Pair<K, V>> q, K key) {
57         assert q != null : "Violation of: q is not null";

```

```

58     assert key != null : "Violation of: key is not null";
59
60     Queue<Pair<K, V>> temp = new Queue1L<>();
61     Queue<Pair<K, V>> temp2 = new Queue1L<>();
62     while (q.length() > 0) {
63         Pair<K, V> pair = q.dequeue();
64         if (pair.key().equals(key)) {
65             temp.enqueue(pair);
66         } else {
67             temp2.enqueue(pair);
68         }
69     }
70     while (temp2.length() > 0) {
71         temp.enqueue(temp2.dequeue());
72     }
73     q.transferFrom(temp);
74
75 }
76
77 /**
78  * Creator of initial representation.
79  */
80 private void createNewRep() {
81     this.pairsQueue = new Queue1L<Pair<K, V>>();
82 }
83
84 /*
85  * Constructors -----
86  */
87
88 /**
89  * No-argument constructor.
90  */
91 public Map2() {
92     this.createNewRep();
93 }
94
95 /*
96  * Standard methods -----
97  */
98
99 @SuppressWarnings("unchecked")
100 @Override
101 public final Map<K, V> newInstance() {
102     try {
103         return this.getClass().getConstructor().newInstance();
104     } catch (ReflectiveOperationException e) {
105         throw new AssertionError(
106             "Cannot construct object of type " + this.getClass());
107     }
108 }
109
110 @Override
111 public final void clear() {
112     this.createNewRep();
113 }
114

```

```

115     @Override
116     public final void transferFrom(Map<K, V> source) {
117         assert source != null : "Violation of: source is not null";
118         assert source != this : "Violation of: source is not this";
119         assert source instanceof Map2<?, ?>
120             : "" + "Violation of: source is of dynamic type Map2<?,?>";
121         /*
122          * This cast cannot fail since the assert above would have stopped
123          * execution in that case: source must be of dynamic type Map2<?,?>, and
124          * the ?,? must be K,V or the call would not have compiled.
125          */
126         Map2<K, V> localSource = (Map2<K, V>) source;
127         this.pairsQueue = localSource.pairsQueue;
128         localSource.createNewRep();
129     }
130
131     /*
132     * Kernel methods -----
133     */
134
135     @Override
136     public final void add(K key, V value) {
137         assert key != null : "Violation of: key is not null";
138         assert value != null : "Violation of: value is not null";
139         assert !this.hasKey(key) : "Violation of: key is not in DOMAIN(this)";
140
141         Pair<K, V> p = new SimplePair<>(key, value);
142         this.pairsQueue.enqueue(p);
143     }
144
145     @Override
146     public final Pair<K, V> remove(K key) {
147         assert key != null : "Violation of: key is not null";
148         assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";
149
150         moveToFront(this.pairsQueue, key);
151         Pair<K, V> p = this.pairsQueue.dequeue();
152         return p;
153     }
154
155     @Override
156     public final Pair<K, V> removeAny() {
157         assert this.size() > 0 : "Violation of: |this| > 0";
158
159         Pair<K, V> p = this.pairsQueue.dequeue();
160         return p;
161     }
162
163     @Override
164     public final V value(K key) {
165         assert key != null : "Violation of: key is not null";
166         assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";
167
168         moveToFront(this.pairsQueue, key);
169         V val = this.pairsQueue.front().value();
170     }

```

```
172         return val;
173     }
174
175     @Override
176     public final boolean hasKey(K key) {
177         assert key != null : "Violation of: key is not null";
178
179         boolean hasKey = false;
180         moveToFront(this.pairsQueue, key);
181         if (this.pairsQueue.front().value().equals(key)) {
182             hasKey = true;
183         }
184         return hasKey;
185     }
186
187     @Override
188     public final int size() {
189
190         int size = this.pairsQueue.length();
191         return size;
192     }
193
194     @Override
195     public final Iterator<Pair<K, V>> iterator() {
196         return this.pairsQueue.iterator();
197     }
198
199 }
200
```