```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 David P. and Zach B.
 *
 */
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;

    /**
```

```java
     * Computes {@code a} mod {@code b} as % should have been defined to work.
     *
     * @param a
     *            the number being reduced
     * @param b
     *            the modulus
     * @return the result of a mod b, which satisfies 0 <= {@code mod} < b
     * @requires b > 0
     * @ensures <pre>
     * 0 <= mod  and  mod < b  and
     * there exists k: integer (a = k * b + mod)
     * </pre>
     */
    private static int mod(int a, int b) {
        assert b > 0 : "Violation of: b > 0";
        // calculate modulo operation and adjust the result to ensure it is non neg
        int mod = a % b;
        if (mod < 0) {
            mod += b;
        }
        return mod;
    }

    /**
     * Creator of initial representation.
     *
     * @param hashTableSize
     *            the size of the hash table
     * @requires hashTableSize > 0
     * @ensures <pre>
     * |$this.hashTable| = hashTableSize  and
     * for all i: integer
     *     where (0 <= i  and  i < |$this.hashTable|)
     *   ($this.hashTable[i, i+1) = <{}>)  and
     * $this.size = 0
     * </pre>
     */
    @SuppressWarnings("unchecked")
    private void createNewRep(int hashTableSize) {
        /*
         * With "new Map<K, V>[...]" in place of "new Map[...]" it does not
         * compile; as shown, it results in a warning about an unchecked
         * conversion, though it cannot fail.
         */

        // make new hash table with specific size
        this.hashTable = new Map[hashTableSize];
        // create new map for each entry of hash-table and set size to 0.
        for (int index = 0; index < hashTableSize; index++) {
            this.hashTable[index] = new Map2<K, V>();
        }
        this.size = 0;
    }

    /*
     * Constructors ------------------------------------------------------------
     */
```

```java

    /**
     * No-argument constructor.
     */
    public Map4() {
        this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
    }

    /**
     * Constructor resulting in a hash table of size {@code hashTableSize}.
     *
     * @param hashTableSize
     *            size of hash table
     * @requires hashTableSize > 0
     * @ensures this = {}
     */
    public Map4(int hashTableSize) {
        this.createNewRep(hashTableSize);
    }

    /*
     * Standard methods ------------------------------------------------------
     */

    @SuppressWarnings("unchecked")
    @Override
    public final Map<K, V> newInstance() {
        try {
            return this.getClass().getConstructor().newInstance();
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep(DEFAULT_HASH_TABLE_SIZE);
    }

    @Override
    public final void transferFrom(Map<K, V> source) {
        assert source != null : "Violation of: source is not null";
        assert source != this : "Violation of: source is not this";
        assert source instanceof Map4<?, ?> : ""
                + "Violation of: source is of dynamic type Map4<?,?>";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type Map4<?,?>, and
         * the ?,? must be K,V or the call would not have compiled.
         */
        Map4<K, V> localSource = (Map4<K, V>) source;
        this.hashTable = localSource.hashTable;
        this.size = localSource.size;
        localSource.createNewRep(DEFAULT_HASH_TABLE_SIZE);
    }

```

```java
    /*
     * Kernel methods -----------------------------------------------------
     */

    @Override
    public final void add(K key, V value) {
        assert key != null : "Violation of: key is not null";
        assert value != null : "Violation of: value is not null";
        assert !this.hasKey(key) : "Violation of: key is not in DOMAIN(this)";

        //compute index using mod, then retrieve map and add key pair to the map.
        int index = mod(key.hashCode(), this.hashTable.length);
        Map<K, V> map = this.hashTable[index];
        map.add(key, value);
        this.size += 1;
    }

    @Override
    public final Pair<K, V> remove(K key) {
        assert key != null : "Violation of: key is not null";
        assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";

        //calculate index, retrieve map1, remove key from map1, and return removed pair.
        int index = mod(key.hashCode(), this.hashTable.length);
        Map<K, V> map1 = this.hashTable[index];
        Pair<K, V> removedPair = map1.remove(key);
        this.size -= 1;
        return removedPair;
    }

    @Override
    public final Pair<K, V> removeAny() {
        assert this.size() > 0 : "Violation of: this /= empty_set";

        // set removed pair to null
        Pair<K, V> removedPair = null;
        // set index to 0
        int index = 0;
        // set boolean to false to solve infinite loop
        boolean found = false;
        // use this while loop to indicate when an element has been removed.
        while (index < this.hashTable.length && !found) {
            Map<K, V> map1 = this.hashTable[index];
            if (map1.size() > 0) {
                // remove any element from the first non empty map1
                removedPair = map1.removeAny();
                this.size--;
                found = true;
            }
            index++;
        }
        // if none are find, return null.
        return removedPair;

    }

    @Override
```

```
234     public final V value(K key) {
235         assert key != null : "Violation of: key is not null";
236         assert this.hasKey(key) : "Violation of: key is in DOMAIN(this)";
237 
238         // set index = mod key and the length, and if the hashtable has index
239         // return the value in the key.
240         int index = mod(key.hashCode(), this.hashTable.length);
241         Map<K, V> map1 = this.hashTable[index];
242         return map1.value(key);
243 
244     }
245 
246     @Override
247     public final boolean hasKey(K key) {
248         assert key != null : "Violation of: key is not null";
249 
250         // calculate index by using mod of the key's hash code.
251         int index = mod(key.hashCode(), this.hashTable.length);
252         Map<K, V> map1 = this.hashTable[index];
253         // check if map1 contains key, returning result.
254         return map1.hasKey(key);
255     }
256 
257     @Override
258     public final int size() {
259         // just return the size
260         return this.size();
261     }
262 
263     @Override
264     public final Iterator<Pair<K, V>> iterator() {
265         return new Map4Iterator();
266     }
267 
268     /**
269      * Implementation of {@code Iterator} interface for {@code Map4}.
270      */
271     private final class Map4Iterator implements Iterator<Pair<K, V>> {
272 
273         /**
274          * Number of elements seen already (i.e., |~this.seen|).
275          */
276         private int numberSeen;
277 
278         /**
279          * Bucket from which current bucket iterator comes.
280          */
281         private int currentBucket;
282 
283         /**
284          * Bucket iterator from which next element will come.
285          */
286         private Iterator<Pair<K, V>> bucketIterator;
287 
288         /**
289          * No-argument constructor.
290          */
```

```
291         Map4Iterator() {
292             this.numberSeen = 0;
293             this.currentBucket = 0;
294             this.bucketIterator = Map4.this.hashTable[0].iterator();
295         }
296 
297         @Override
298         public boolean hasNext() {
299             return this.numberSeen < Map4.this.size;
300         }
301 
302         @Override
303         public Pair<K, V> next() {
304             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
305             if (!this.hasNext()) {
306                 /*
307                  * Exception is supposed to be thrown in this case, but with
308                  * assertion-checking enabled it cannot happen because of assert
309                  * above.
310                  */
311                 throw new NoSuchElementException();
312             }
313             this.numberSeen++;
314             while (!this.bucketIterator.hasNext()) {
315                 this.currentBucket++;
316                 this.bucketIterator = Map4.this.hashTable[this.currentBucket]
317                         .iterator();
318             }
319             return this.bucketIterator.next();
320         }
321 
322         @Override
323         public void remove() {
324             throw new UnsupportedOperationException(
325                     "remove operation not supported");
326         }
327 
328     }
329 
330 }
331 
```