```java
import java.util.Iterator;

import components.map.Map;
import components.map.MapSecondary;
import components.queue.Queue;
import components.queue.Queue1L;

/**
 * {@code Map} represented as a {@code Queue} of pairs 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>
 * for all key1, key2: K, value1, value2: V, str1, str2: string of (key, value)
 *     where (str1 * <(key1, value1)> is prefix of $this.pairsQueue and
 *            str2 * <(key2, value2)> is prefix of $this.pairsQueue and
 *            str1 /= str2)
 *   (key1 /= key2)
 * </pre>
 * @correspondence this = entries($this.pairsQueue)
 */
public class Map2<K, V> extends MapSecondary<K, V> {

    //private static final V  = null;
    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Pairs included in {@code this}.
     */
    private Queue<Pair<K, V>> pairsQueue;

    /**
     * Finds pair with first component {@code key} and, if such exists, moves it
     * to the front of {@code q}.
     *
     * @param <K>
     *            type of {@code Pair} key
     * @param <V>
     *            type of {@code Pair} value
     * @param q
     *            the {@code Queue} to be searched
     * @param key
     *            the key to be searched for
     * @updates q
     * @ensures <pre>
     * perms(q, #q)  and
     * if there exists value: V (<(key, value)> is substring of q)
     *  then there exists value: V (<(key, value)> is prefix of q)
     * </pre>
     */
    private static <K, V> void moveToFront(Queue<Pair<K, V>> q, K key) {
        assert q != null : "Violation of: q is not null";
        assert key != null : "Violation of: key is not null";

        Queue<Pair<K, V>> temp = new Queue1L<>();
        Queue<Pair<K, V>> temp2 = new Queue1L<>();
        while (q.length() > 0) {
            Pair<K, V> pair = q.dequeue();
            if (pair.key().equals(key)) {
                temp.enqueue(pair);
            } else {
                temp2.enqueue(pair);
            }
        }
        while (temp2.length() > 0) {
            temp.enqueue(temp2.dequeue());
        }
        q.transferFrom(temp);

    }

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {
        this.pairsQueue = new Queue1L<Pair<K, V>>();
    }

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

    /**
     * No-argument constructor.
     */
    public Map2() {
        this.createNewRep();
    }

    /*
     * 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();
    }

```

```java
    @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 Map2<?, ?>
                : "" + "Violation of: source is of dynamic type Map2<?,?>";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type Map2<?,?>, and
         * the ?,? must be K,V or the call would not have compiled.
         */
        Map2<K, V> localSource = (Map2<K, V>) source;
        this.pairsQueue = localSource.pairsQueue;
        localSource.createNewRep();
    }

    /*
     * 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)";

        Pair<K, V> p = new SimplePair<>(key, value);
        this.pairsQueue.enqueue(p);

    }

    @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)";

        moveToFront(this.pairsQueue, key);
        Pair<K, V> p = this.pairsQueue.dequeue();
        return p;

    }

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

        Pair<K, V> p = this.pairsQueue.dequeue();
        return p;
    }

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

        moveToFront(this.pairsQueue, key);
        V val = this.pairsQueue.front().value();
        return val;
    }

    @Override
    public final boolean hasKey(K key) {
        assert key != null : "Violation of: key is not null";

        boolean hasKey = false;
        moveToFront(this.pairsQueue, key);
        if (this.pairsQueue.front().value().equals(key)) {
            hasKey = true;
        }
        return hasKey;
    }

    @Override
    public final int size() {

        int size = this.pairsQueue.length();
        return size;
    }

    @Override
    public final Iterator<Pair<K, V>> iterator() {
        return this.pairsQueue.iterator();
    }

}
```