```
import java.util.Iterator;

/** A hash table with linear probing and the MAD hash function */

public class HashTableMap<K,V> implements Map<K,V> {

  /** Nested class for an entry in a hash table. */
  public static class HashEntry<K,V> implements Entry<K,V> {
    protected K key;
    protected V value;
    public HashEntry(K k, V v) { key = k; value = v; }
    public V getValue() { return value; }
    public K getKey() { return key; }
    public V setValue(V val) {
      V oldValue = value;
      value = val;
      return oldValue;
      }
    public boolean equals(Object o) {
      HashEntry<K,V> ent;
      try { ent = (HashEntry<K,V>) o; }
      catch (ClassCastException ex) { return false; }
      return (ent.getKey() == key) && (ent.getValue() == value);
    }

  /** Entry visualization. */
    public String toString() {
      return "(" + key + "," + value + ")";
    }
  }

  protected Entry<K,V> AVAILABLE = new HashEntry<K,V>(null, null);
  protected int n = 0;               // number of entries in the dictionary
  protected int prime, capacity; // prime factor and capacity of bucket array
  protected Entry<K,V>[] bucket;// bucket array
  protected long scale, shift;   // the shift and scaling factors

  /** Creates a hash table with prime factor 109345121 and capacity 1000. */
  public HashTableMap() { this(109345121,1000); }
  /** Creates a hash table with prime factor 109345121 and given capacity. */
  public HashTableMap(int cap) { this(109345121, cap); }

  /** Creates a hash table with the given prime factor and capacity. */
  public HashTableMap(int p, int cap) {
    prime = p;
    capacity = cap;
    bucket = (Entry<K,V>[]) new Entry[capacity]; // safe cast
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(prime-1) + 1;
    shift = rand.nextInt(prime);
  }
```

```
/** Determines whether a key is valid. */
protected void checkKey(K k) {
  if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashValue(K key) {
  return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
}

/** Returns the number of entries in the hash table. */
public int size() { return n; }

/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }

/** Returns an iterable object containing all of the keys. */
public Iterable<K> keys() {
  PositionList<K> keys = new NodePositionList<K>();
  for (int i=0; i<capacity; i++)
    if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
        keys.addLast(bucket[i].getKey());
  return keys;
}

/** Helper search method - returns index of found key or -(a + 1),
 * where a is the index of the first empty or available slot found. */
protected int findEntry(K key) throws InvalidKeyException {
  int avail = -1;
  checkKey(key);
  int i = hashValue(key);
  int j = i;
  do {
    Entry<K,V> e = bucket[i];
    if ( e == null) {
        if (avail < 0)
          avail = i;       // key is not in table
        break;
    }
    if (key.equals(e.getKey())) // we have found our key
        return i;          // key found
    if (e == AVAILABLE) {  // bucket is deactivated
        if (avail < 0)
          avail = i;       // remember that this slot is available
    }
    i = (i + 1) % capacity;       // keep looking
  } while (i != j);
  return -(avail + 1);  // first empty or available slot
}
```

```
/** Returns the value associated with a key. */
public V get (K key) throws InvalidKeyException {
  int i = findEntry(key);  // helper method for finding a key
  if (i < 0) return null;  // there is no value for this key, so reutrn null
  return bucket[i].getValue();     // return the found value in this case
}

/** Put a key-value pair in the map, replacing previous one if it exists. */
public V put (K key, V value) throws InvalidKeyException {
  int i = findEntry(key); //find the appropriate spot for this entry
  if (i >= 0)    //  this key has a previous value
    return ((HashEntry<K,V>) bucket[i]).setValue(value); // set new value
  if (n >= capacity/2) {
    rehash(); // rehash to keep the load factor <= 0.5
    i = findEntry(key); //find again the appropriate spot for this entry
  }
  bucket[-i-1] = new HashEntry<K,V>(key, value); // convert to proper index
  n++;
  return null;  // there was no previous value
}

/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  capacity = 2*capacity;
  Entry<K,V>[] old = bucket;
  bucket = (Entry<K,V>[]) new Entry[capacity]; // new bucket is twice as big
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(prime-1) + 1;              // new hash scaling factor
  shift = rand.nextInt(prime);                    // new hash shifting factor
  for (int i=0; i<old.length; i++) {
    Entry<K,V> e = old[i];
    if ((e != null) && (e != AVAILABLE)) { // a valid entry
          int j = - 1 - findEntry(e.getKey());
          bucket[j] = e;
    }
  }
}

/** Removes the key-value pair with a specified key. */
public V remove (K key) throws InvalidKeyException {
  int i = findEntry(key);        // find this key first
  if (i < 0) return null;          // nothing to remove
  V toReturn = bucket[i].getValue();
  bucket[i] = AVAILABLE;            // mark this slot as deactivated
  n--;
  return toReturn;
}
```

```
  /** Returns an iterable object containing all of the entries. */
  public Iterable<Entry<K,V>> entries() {
    PositionList<Entry<K,V>> entries = new NodePositionList<Entry<K,V>>();
    for (int i=0; i<capacity; i++)
      if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
          entries.addLast(bucket[i]);
    return entries;
  }

  /** Returns an iterable object containing all of the values. */
  public Iterable<V> values() {
    PositionList<V> values = new NodePositionList<V>();
    for (int i=0; i<capacity; i++)
      if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
          values.addLast(bucket[i].getValue());
    return values;
  }
}
```