9.2 An ADT for Maps

Maps are collections of keyed records. Thus, in order to provide an abstract data type for maps we first need one for keyed records.

ADT: Keyed Record

A *keyed record* is an ordered pair of objects named *key* and *value*.

Operations

1. *Initialize*: Create a keyed record having a given key and given value.
2. *Key*: Return the key object in this record.
3. *Value*: Return the value object in this record.
4. *Update*: Replace the value object in this record with a given value object.

Listing 9.2 translates this ADT into a Java interface named `Entry`. Its UML diagram is shown in Figure 9.1. The *Initialize* operation will be implemented as a two-argument constructor, so it does not appear in the interface. The *Key* and *Value* operations are defined as `getKey()` and `getValue()` methods, and the *Update* operation is defined as the `setValue()` method.

LISTING 9.2: An `Entry` Interface

```
1  public interface Entry {
2    public Object getKey();
3    // RETURN: key;
4    // POST: key is the first object in this ordered pair;
5
6    public Object getValue();
7    // RETURN: value;
8    // POST: value is the second object in this ordered pair;
9
10   public void setValue(Object value);
11   // POST: value is the second object in this ordered pair;
12 }
```

This defines two "getter" methods and one "setter" method. This gives read–write access to the value component, but restricts the key component to read-only access. That is consistent with the constraints of maps, which require their key fields to be immutable.

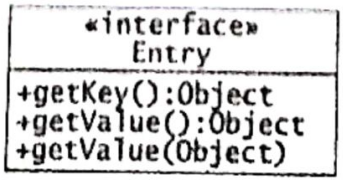

FIGURE 9.1 An `Entry` interface.

ADT: Map

A *map* is a collection of keyed records within which the keys are unique.

Operations

1. *Initialize*: Create an empty map.
2. *Search*: For a given key, search the table for a record that has that key. If found, return its value; otherwise, return `null`.
3. *Insert/Update*: For a given record, search the table for a record that has that key. If found, replace the value of the table's record with the value of the given record and return the replaced value; otherwise, insert the given record into the table.
4. *Delete*: For a given key, search the table for a record that has that key. If found, delete it from the table and return its value; otherwise, return `null`.
5. *Count*: Return the number of records in the table.

This ADT specifies only a minimal set of operations. As we shall see later, Java adds more functionality to its `java.util.Map` interface.

The Map ADT is translated directly into a Java interface in Listing 9.3. It's UML diagram is shown in Figure 9.2. The *Initialize* operation will be implemented as a no-argument constructor, so it does not appear in the interface. The *Search* operation is defined as the `get()` method, the *Insert/Update* operation is defined as the `put()` method, the *Delete* operation is defined as the `remove()` method, and the *Count* operation is defined as the `size()` method. As we shall see, these method signatures are the same as those defined in the `java.util.Map` interface. Moreover, the method signatures in the `Entry` interface in Listing 9.2 are the same as those defined in the `java.util.Map.Entry` interface, which is a sub-interface (or *nested interface*) of the `java.util.Map` interface.

Notice how the value returned by `get()`, `put()`, and `remove()` transmits the outcome of the operation. The `null` value signals failure with the `get()` and `remove()` methods. But the `put()` method serves double duty: It is used both to insert new records and to modify existing records. So it returns `null` to signal success at inserting the new record, and non-`null` to signal success at modifying an existing record.

When we think of data structures and abstract data types, we usually imagine them to be implemented in computer memory. But the ADT may apply to much larger structures. For example, the Internet can be regarded as one enormous map. Its keys are the URLs and their values are the web pages that they address. For example, the URL

```
http://www.paris.org/Musees/Louvre/Treasures/gifs/Mona_Lisa.jpg
```

brings up an image of the Mona Lisa from the Louvre in Paris.

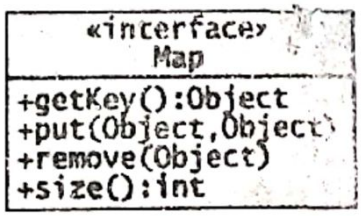

FIGURE 9.2 A Map interface.

LISTING 9.3: A Map Interface

```
1  public interface Map {
2
3    public Object get(Object key);
4    // RETURN: value;
5    // POST: if value!=null, then (key,value) is in this map;
6    //       if value==null, then no record in this map has the given key;
7
8    public Object put(Object key, Object value);
9    // RETURN: oldValue;
10   // POST: if oldValue==null, then (key,value) is in this map;
11   //       if oldValue!=null, then (key,oldValue) was in this map;
12
13   public Object remove(Object key);
14   // RETURN: oldValue;
15   // POST: if oldValue==null, no record in this map has the given key;
16   //       if oldValue!=null, then (key,oldValue) was in this map;
17
18   public int size();
19   // RETURN: n;
20   // POST: this map contains n records;
21 }
```

9.3 Hash Tables

Without further organization of the data structure, access to a keyed table is sequential. If the table is sorted by its key values and stored in an array, then binary searching could be implemented, improving the access time from $\Theta(n)$ to $\Theta(\lg n)$. With *hashing*, however, we can do better without sorting.

A *hash function* for a keyed table is a function that returns the location (the array index) of the unique record in the able that has the given key value. A *hash table* is a keyed table that has a hash function.

The Java `Object` class defines a `hashCode()` method that returns an `int` for any object. Thus, in Java, every object has its own hash code. The code is computed from the data stored in the object.

For example, the Java hash code for the `String` object `"IT"` is 2347. This is $31 \cdot 73 + 84$. Similarly, the hash code for `"SE"` is 2642, which is $31 \cdot 83 + 69$. So we can see how the Java hash function works, at least for a two-character `String` object `iso`: It returns the integer value

```
31*iso.charAt(0) + iso.charAt(1)
```

Table 9.3 shows the Java `hashCode()` values for the seven ISO strings used in Table 9.2.

In general, the Java `hashCode()` value for an object can be any `int`, positive or negative. To use it as an array index, it must be converted into an integer h in the range $0 \le h < m$, where m is the length of the array. A standard technique for doing that is to use the integer

```
int h = (key.hashCode() & 0x7FFFFFFF) % entries.length;
```

key	hashCode(key)
AT	2099
FR	2252
DE	2177
GR	2283
IT	2347
PT	2564
SE	2642

TABLE 9.3 The hashCode() method.

The literal 0x7FFFFFFF is hexadecimal for the bit string 01111111111111111111111111111111 (a "0" followed by 31 "1"s). The operator & is the "bitwise and" operator. When applied to an integer k using the mask 0x7FFFFFFF, it simply changes k's leading bit to 0, which is the same as assigning to k its absolute value $|k|$. The operation x % m gives the remainder from the division of x by m, producing a nonnegative integer less than m. For example, if k is –1836, then the expression

```
(k & 0x7FFFFFFF) % m
```

will evaluate to 7 if m is 41.

A hash function provides nearly constant time access to elements in a hash table. This is seen from the get() and put() methods in the HashTable class shown in Listing 9.4. The array element entries[i] is accessed on lines 6 and 10, where i is the index returned by hash(key). That element, entries[hash(key)], is the value of the key/value pair stored in the hash table.

LISTING 9.4: A Naive Hash Table Class

```
1  public class HashTable implements Map {
2    private Entry[] entries = new Entry[11];
3    private int size;
4
5    public Object get(Object key) {
6      return entries[hash(key)].value;
7    }
8
9    public Object put(Object key, Object value) {
10     entries[hash(key)] = new Entry(key,value);
11     ++size;
12     return null;
13   }
14
15   public Object remove(Object key) {
16     int h = hash(key);
```

```
17      Object value = entries[h].value;
18      entries[h] = null;
19      --size;
20      return value;
21    }
22
23    public int size() {
24      return size;
25    }
26
27    private class Entry {
28      Object key, value;
29      Entry(Object k, Object v) { key = k; value = v; }
30    }
31
32    private int hash(Object key) {
33      return (key.hashCode() & 0x7FFFFFFF) % entries.length;
34    }
35  }
```

The code in Listing 9.4 is consistent with the specifications in the `java.util.Map` interface, which is discussed later in this chapter. That interface requires the `get()`, `put()`, `remove()`, and `size()` methods implemented here. This `HashTable` class defines a `private` inner class named `Entry` (at lines 27–30) for its key/value pairs. Its two-argument constructor, defined at line 29, is used by the `HashTable` constructor at line 10. The data structure for this class is an array of `Entry` objects, defined at line 2.

This `HashTable` class is naive because it has no way to handle exceptional situations that are likely to occur. In the example, inserting those seven `Country` records in a table of length $m = 11$ worked fine because every key hashed to a different location. That will not always happen.

Suppose that the next record to be inserted is the `Country` record for the United Kingdom. Its ISO key is `"GB"`, which hashes to index 1. The `put()` method for this naive `HashTable` class would insert the record at `entries[1]`, overwriting the record for Portugal. This is called a *collision*. Clearly, it should be managed differently.

0	
1	"PT",("Portugal","Portuguese",35672,9918040)
2	"SE",("Sweden","Swedish",173732,8911296)
3	
4	"IT",("Italy","Italian",116300,56735130)
5	
6	"GR",("Greece","Greek",50900,10707135)
7	
8	"FR",("France","French",211200,58978172)
9	"AT",("Austria","German",32378,8139299)
10	"DE",("Germany","German",137800,82087361)

FIGURE 9.3 A hash table of size 7.

9.4 Linear Probing

The simplest way to resolve collisions in a hash table is to put the colliding record in the next available cell in the array. This algorithm is called *linear probing*, because in each "probe" we increment the array index by 1. It is also called *open addressing*, because an element is not always placed in the slot indexed by its hash value; it can end up anywhere in the table.

In the previous example, the call

```
put("GB", new Country("United Kingdom", "English", 94500, 59113439));
```

should resolve the collision by inserting this new record in `entries[3]`, since `"GB"` hashes to index 1 and `entries[1]` and `entries[2]` are occupied. Then the call `get("GB")` would find the record, first by hashing to index 1, and then probing sequentially to `entries[1]`, `entries[2]`, and finally `entries[3]` before finding the record there. This probing is shown in Figure 9.4.

Suppose that the next insertion is

```
put("NL", new Country("Netherlands", "Dutch", 16033, 15807641));
```

The key `"NL"` hashes to index 8, so the linear probing algorithm would search `entries[8]`, `entries[9]`, and `entries[10]`, before finding an empty cell at `entries[0]`. The algorithm "wraps around" the end of the array, like a circular list, as shown in Figure 9.5.

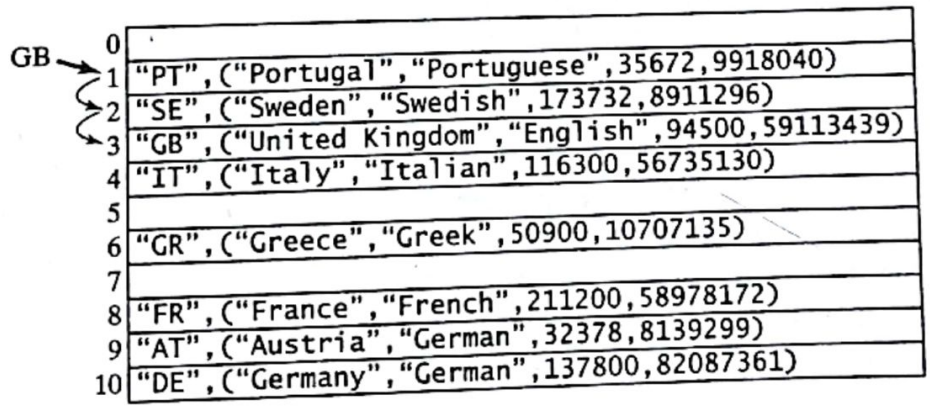

FIGURE 9.4 Linear probing.

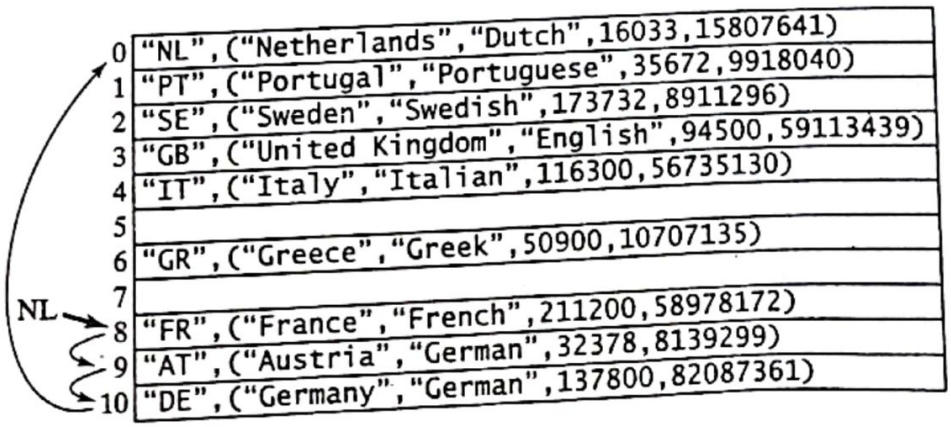

FIGURE 9.5 Wrapping around.

The revised code for the put() method would look like this:

```
public Object put(Object key, Object value) {
  int h = hash(key);
  for (int i = 0; i < entries.length; i++) {
    int j = (h+i)%entries.length;
    Entry entry=entries[j];
    if (entry == null) {
      entries[j] = new Entry(key,value);
      ++size;
      ++used;
      return null;  // insertion success
    }
  }
  throw new IllegalStateException();  // failure: table overflow
  return null;
}
```

The expression (h+i)%entries.length handles the wraparound part of the algorithm. In the "NL" example, h = 8 and i increments in the loop with values 0, 1, 2, and 3, so (h+i) increments through 8, 9, 10, and 11. As long as (h+i) < 11, the value of the expression (h+i)%entries.length will be simply (h+i). When (h+i) = 11, the value of the expression (h+i)%entries.length will be 0 because 11 divided by 11 has remainder 0. In general, the expression (h+i)%entries.length will equal (h+i) or (h+i) − 11, according to whether (h+i) < entries.length or (h+i) ≥ entries.length.

The revision for the get() method is similar.

Revising the remove() method (at line 15) is a bit more complicated. Suppose we remove the "SE" record at entries[2] and then call get("GB"). If remove("SE") sets entries[2] to null, then we will never find "GB" at entries[3], because the search stops when null is encountered. We could just search the entire table, but that would make the algorithm run in linear time, instead of the near-constant time promised by the hash table. A better solution is to replace the deleted Entry object with a special dummy variable that will not stop the linear probing search.

We define a special Entry object named NIL like this:

```
private final Entry NIL = new Entry(null, null);  // dummy
```

Then the revised remove() method looks like this:

```
public Object remove(Object key) {
  int h = hash(key);
  for (int i = 0; i < entries.length; i++) {
    int j = (h+i)%entries.length;
    if (entries[j] == null) break;
    if (entries[j].key.equals(key)) {
      Object value = entries[j].value;
      entries[j] = NIL;
      --size;
      return value;
    }
  }
  return null;  // failure: key not found
}
```

Now we can modify the put() method again, interpreting the "next available cell" as being the next entries[j] that is either null or NIL:

```
if (entries[j] == null || entries[j] == NIL) {
  entries[j] = new Entry(key,value);
  ++size;
  return null;
}
```

9.5 Rehashing

The next improvement to make to our HashTable class is to handle the overflow problem. The solution is to rebuild the table using a larger array. This is called *rehashing*.

Here is a rehash() method that creates an array that is more than twice the size of the existing array and moves all the entries from the old array to the new one:

```
private void rehash() {
  Entry[] oldEntries = entries;
  entries = new Entry[2*oldEntries. length+1];
  for (int k = 0; k < oldEntries.length; k++) {
    Entry entry = oldEntries[k];
    if (entry == null || entry == NIL) continue;
    int h = hash(entry.key);
    for (int i = 0; i < entries.length; i++) {
      int j = nextProbe(h,i);
      if (entries[j]==null) {
        entries[j] = entry;
        break;
      }
    }
  }
  used = size;
}
```

First it assigns a new reference, oldEntries[], to the existing array so that the class field name entries[] can be used to reference the new array. The length of the new array is set to $2m+1$, where m is the length of the old array. The "+1" is used to make the length an odd number, thereby reducing its number of divisors. The hash table's performance tends to be better if its array length has fewer divisors.

Note that reallocating the array entries[] automatically redefines the hash() method because it changes the value of entries.length. Also note the rehash() method does not create any new Entry objects. It simply provides new references (in the new array) to the existing objects. The size does not change.

In our solution to the remove() problem described in the previous section, you may have noticed that its creates another problem. As more insertions are deletions are performed, more and more references to the NIL object will be stored in the array. No matter how many remove() operations occur, the probing sequences will never get any shorter. This increases the frequency of collisions, thus making the linear probing sequences longer and degrading the general performance of the data structure.

The rehash() method removes all the NIL references; *i.e.*, it does not copy any of them to the new array. So calling rehash() will generally boost performance. Of course, it is more efficient to initialize the hash table with plenty of space to begin with, rather than to start small and then have to rehash many times.

In any case, experience has shown that calling rehash() before the table becomes full is a good strategy. This is done by setting a threshold size, which triggers a call to rehash(). But instead of storing the threshold value, we specify a maximum ratio $r = n/m$, where n = size and m = entries.length. This ratio is called the *load factor*. Its upper limit is typically set at around 75% or 80%. For example, if we set it at 75% and begin with an array length of 11, then rehash() will be called when the size exceeds 8.25 (*i.e.*, after the 9th record has been inserted). That will rebuild the hash table to a length of 23.

LISTING 9.5: A Correct Hash Table Class

```
1  public class HashTable implements Map {
2    private Entry[] entries;
3    private int size, used;
4    private float loadFactor;
5    private final Entry NIL = new Entry(null, null);  // dummy
6
7    public HashTable(int capacity, float loadFactor) {
8      entries = new Entry[capacity];
9      this.loadFactor = loadFactor;
10   }
11
12   public HashTable(int capacity) {
13     this(capacity, 0.75F);
14   }
15
16   public HashTable() {
17     this(101);
18   }
19
20   public Object get(Object key) {
21     int h = hash(key);
22     for (int i = 0; i < entries.length; i++) {
23       int j = nextProbe(h,i);
24       Entry entry=entries[j];
25       if (entry == null) break;
26       if (entry == NIL) continue;
27       if (entry.key.equals(key)) return entry.value;  // success
28     }
29     return null;  // failure: key not found
30   }
31
32   public Object put(Object key, Object value) {
33     if (used > loadFactor*entries.length) rehash();
34     int h = hash(key);
35     for (int i = 0; i < entries.length; i++) {
```

```
      int j = nextProbe(h,i);
      Entry entry = entries[j];
      if (entry == null) {
        entries[j] = new Entry(key, value);
        ++size;
        ++used;
        return null;  // insertion success
      }
      if (entry == NIL) continue;
      if (entry.key.equals(key)) {
        Object oldValue = entry.value;
        entries[j].value = value;
        return oldValue;  // update success
      }
    }
    return null;  // failure: table overflow
  }

  public Object remove(Object key) {
    int h = hash(key);
    for (int i = 0; i < entries.length; i++) {
      int j = nextProbe(h,i);
      Entry entry = entries[j];
      if (entry == null) break;
      if (entry == NIL) continue;
      if (entry.key.equals(key)) {
        Object oldValue = entry.value;
        entries[j] = NIL;
        --size;
        return oldValue;  // success
      }
    }
    return null;  // failure: key not found
  }

  public int size() {
    return size;
  }

  private class Entry {
    Object key, value;
    Entry(Object k, Object v) { key = k; value = v; }
  }

  private int hash(Object key) {
    if (key == null) throw new IllegalArgumentException();
    return (key.hashCode() & 0x7FFFFFFF) % entries.length;
  }

  private int nextProbe(int h, int i) {
```

```
86      return (h + i)%entries.length;          // Linear Probing
87    }
88
89    private void rehash() {
90      Entry[] oldEntries = entries;
91      entries = new Entry[2*oldEntries.length+1];
92      for (int k = 0; k < oldEntries.length; k++) {
93        Entry entry = oldEntries[k];
94        if (entry == null || entry == NIL) continue;
95        int h = hash(entry.key);
96        for (int i = 0; i < entries.length; i++) {
97          int j = nextProbe(h,i);
98          if (entries[j] == null) {
99            entries[j] = entry;
100           break;
101         }
102       }
103     }
104     used = size;
105   }
106 }
```

The complete corrected `HashTable` class is shown in Listing 9.5. It includes the `loadFactor` field and the constant `NIL`, as well as three constructors and the `private rehash()` method. The constructors allow the client to set the initial array capacity and the load factor, which otherwise are given the default values of 0.75 and 101, respectively.

The `private nextProbe()` method generates the indexes for the probing sequence. It is used by the `get()`, `put()`, `remove()`, and `rehash()` methods as they sequentially search for the next available space or the for a given key. The run time of each of these four methods is proportional to the number of calls that they make to `nextProbe()`.

Note that the `put()` method now includes code that will update a stored record. When used with a key that is already in the table, it replaces the existing value with a value passed to `put()` and returns the old value. That is why `put()` returns `null` in the other two cases (*i.e.*, inserting a new record or table overflow): to distinguish those outcomes from an update.

9.6 Other Collision Resolution Algorithms

Linear probing is a simple and fairly effective method for resolving collisions. However, if the hash function fails to distribute the records uniformly throughout the table, then linear probing can lead to long chains of records bunched together. This is called *primary clustering*.

For example, suppose we insert these nine countries in an empty table of length $m = 101$:

```
put("FI", new Country("Finland", "Finnish", 130100, 5158372));
put("IQ", new Country("Iraq", "Arabic", 168754, 22427150));
put("IR", new Country("Iran", "Farsi", 636000, 65179752));
put("SK", new Country("Slovakia", "Slovak", 18859, 5396193));
put("CA", new Country("Canada", "English", 3851800, 31006347));
put("LY", new Country("Libya", "Arabic", 679400, 4992838));
```

```
put("IT", new Country("Italy", "Italian", 116300, 56735130));
put("PE", new Country("Peru", "Spanish", 496200, 26624582));
put("IS", new Country("Iceland", "Islenska", 40000, 272512));
```

Their hash() values are shown in Table 9.4. With linear probing, we get 26 collisions wh inserting these nine records in that order:

"FI" → 21
"IQ" → 21 → 22
"IR" → 22 → 23
"SK" → 22 → 23 → 24
"CA" → 21 → 22 → 23 → 24 → 25
"LY" → 21 → 22 → 23 → 24 → 25 → 26
"IT" → 24 → 25 → 26 → 27
"PE" → 24 → 25 → 26 → 27 → 28
"IS" → 23 → 24 → 25 → 26 → 27 → 28 → 29

This shows how clustering can degrade performance. The single cluster has no gaps wit it. So new records like "PE" and "IS" that hash near the beginning of the cluster have to w through it one step at a time, causing more collisions and wasting time.

One alternative to linear probing is called *quadratic probing*. That algorithm resolves co sions by incrementing in increasingly greater steps, instead of by 1 each time. To implemen replace lines 23, 33, 51, and 86 in Listing 9.5 on page 276 with

```
int j = (h + i*i)%entries.length;
```

Instead of adding `i` to `h` after each collision, we add `i*i`. (Quadratic means square the variab The increment sequence is 1, 4, 9, 16, 25, · · · . These are the (absolute) increments from the tial hash value `h`. The successive (relative) increments are 1, 3, 5, 7, 9, · · · .[1]

key	hash(key)
"FI"	21
"IQ"	21
"IR"	22
"SK"	22
"CA"	21
"LY"	21
"IT"	24
"PE"	24
"IS"	23

TABLE 9.4 Hashing Keys.

[1] 3 = 4 − 1; 5 = 9 − 4; 7 = 16 − 9; 9 = 25 − 16; etc.

With quadratic probing, the same nine-record input sequence results in only 13 collisions:

"FI" → 21
"IQ" → 21 → 22
"IR" → 22 → 23
"SK" → 22 → 23 → 26
"CA" → 21 → 22 → 25
"LY" → 21 → 22 → 25 → 30
"IT" → 24
"PE" → 24 → 25 → 28
"IS" → 23 → 24 → 27

The probe sequence for "LY" is shown in Figure 9.6.

Quadratic probing produced half as many collisions as linear probing. The improvement results because quadratic probing leaves unused gaps, thereby causing less clustering than linear probing.

But quadratic probing still has problems. For example, suppose we are hashing into a table of length $m = 11$. Suppose also that a key hashes to index $j = 3$ and encounters repeated collisions. The probe sequence will be 3, 4, 7, 1, 8, 6, 6, 8, 1, 7, 4, 3, 4, 7, 1, 8, 6, 6, 8, 1, 7, 4, 3, ... This is a sparse periodic sequence. It reaches only 6 of the 11 cells. If those six cells are occupied, then `put()` will fail, even if the other five cells are unoccupied.

The solution is to set the threshold load factor at 50%. It can be shown that the sparse periodic sequences that can result from quadratic probing will always reach more than half of all the cells in the table. In this example, $6 > 11/2$. So if no more than half of all cells can be occupied, then those sparse periodic probe sequences will still find empty cells.

Even with limiting the load factor to 50%, double hashing can still suffer from *secondary clustering*. This the same problem that linear probing has: Two different keys that hash to the same value will have the same probe sequence. The solution to that problem is to use a second, independent hash function to determine the probe sequence. That is called *double hashing*.

Like linear probing, double hashing uses a constant increment in its probe sequence. But that increment, determined by the second hash function, is usually greater than 1. So, like quadratic probing, double hashing avoids primary clustering by spreading out its probe sequence.

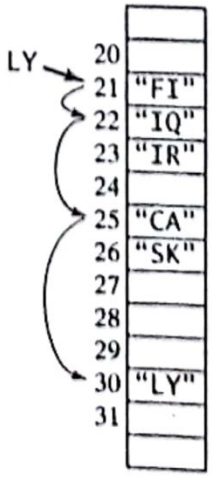

FIGURE 9.6 Quadratic probing.

For double hashing, we need a second hash function, say `hash2()`. We can use the same ode as in lines 85–87 of Listing 9.5, except with this version of line 86:

```
return 1 + (key.hashCode() & 0x7FFFFFFF) % (entries.length - 1);
```

other words, we use $h = c \% m$ for `hash()` and $d = 1 + c \% (m - 1)$ for `hash2()`. This guaran- es that the increment d will be in the range $1 \le d < m$.

The other changes to Listing 9.5 needed for double hashing are to replace lines 21, 34, and 5 with

```
int h=hash(key), d=hash2(key);
```

place the call to `nextProbe(h, i)` with `nextProbe(h, d, i)` at lines 23, 36, and 57, make milar changes in the `rehash()` method, and add the `hash2()` method, as described above.

With that implementation, the same nine-record input sequence of Table 9.4 results in only ve collisions:

```
"FI" → 21
"IQ" → 21 → 89
"IR" → 22
"SK" → 22 → 97
"CA" → 21 → 85
"LY" → 21 → 91
"IT" → 24
"PE" → 24 → 99
"IS" → 23
```

We have seen that both quadratic probing and double hashing solve the primary clustering roblem seen with linear probing. Double hashing also solves the problem of secondary cluster- g, which occurs with both linear and quadratic probing. Quadratic probing also requires the reshold load factor to be set at 50% to avoid sparse periodic probe sequences.

.7 Separate Chaining

he hashing algorithms described in the previous section are called *open addressing*, because they ek open positions in the array to resolve collisions. The alternative, called *closed addressing*, oids collisions altogether by allowing more than one record to be stored at a hash location. This quires a more complex data structure.

Instead of an array of records, we use an array of buckets, where a *bucket* is a collection of cords. The simplest data structure to use for a bucket is a linked list. This is called *separate haining*.

Listing 9.6 shows an implementation of our `HashTable` class using separate chaining. The nly difference in the code for this structure is that the inner `Entry` class has a `next` field (at line 2) to implement the singly linked lists.

Although the data structure is more complicated for hash tables with closed addressing, ie code is actually a bit shorter. We need no dummy `NIL` entry, nor any collision resolution lgorithms.

The get() method hashes to the target chain (at line 20) and then searches that linked list

LISTING 9.6: **Closed Addressing by Separate Chaining**

```
public class HashTable {
  private Entry[] entries;
  private int size;
  private float loadFactor;

  public HashTable(int capacity, float loadFactor) {
    entries = new Entry[capacity];
    this.loadFactor = loadFactor;
  }

  public HashTable(int capacity) {
    this(capacity, 0.75F);
  }

  public HashTable() {
    this(101);
  }

  public Object get(Object key) {
    int h = hash(key);
    for (Entry e = entries[h]; e != null; e = e.next) {
      if (e.key.equals(key)) return e.value;  // success
    }
    return null;  // failure: key not found
  }

  public Object put(Object key, Object value) {
    int h = hash(key);
    for (Entry e = entries[h]; e != null; e = e.next) {
      if (e.key.equals(key)) {
        Object oldValue = e.value;
        e.value = value;
        return oldValue;  // successful update
      }
    }
    entries[h] = new Entry(key,value,entries[h]);
    ++size;
    if (size > loadFactor*entries.length) rehash();
    return null;  // successful insertion
  }

  public Object remove(Object key) {
    int h = hash(key);
    for (Entry e = entries[h], prev=null; e!=null; prev=e, e=e.next
      if (e.key.equals(key)) {
        Object oldValuev=ve.value;
        if (prev == null) entries[h] = e.next;
        else prev.next = e.next;
```

```
        --size;
        return oldValue;  // success
      }
    }
    return null;  // failure: key not found
  }

  public int size() {
    return size;
  }

  private class Entry {
    Object key, value;
    Entry next;
    Entry(Object k, Object v, Entry n) { key=k; value=v; next=n; }
    public String toString() {
      return key + "=" + (Country)value;
    }
  }

  private int hash(Object key) {
    if (key == null) throw new IllegalArgumentException();
    return (key.hashCode() & 0x7FFFFFFF) % entries.length;
  }

  private void rehash() {
    Entry[] oldEntries = entries;
    entries = new Entry[2*oldEntries.length+1];
    for (int k = 0; k < oldEntries.length; k++) {
      for (Entry old = oldEntries[k]; old != null; ) {
        Entry e = old;
        old = old.next;
        int h = hash(e.key);
        e.next = entries[h];
        entries[h] = e;
      }
    }
  }
}
```

Note that with closed addressing, the load factor may exceed the length of the backing array ce there is no limit to the length of the chains. Nevertheless, the hash table's performance will grade if we allow long chains. So we still include rehashing when the table `size` exceeds its eshold.

Figure 9.7 shows how the hash table looks after loading the 15 European Union countries, ng a capacity of 11 and a load factor of 2.0. This shows only the keys, not the complete records.

Closed addressing has the obvious advantage of preventing collisions: If each bucket can hold itrarily many keys, then it won't overflow. The disadvantage is that some chains can become y long. That slows the search because each chain is searched sequentially. A very unbalanced ay of bucket chains can destroy the constant time access that hashing is meant to provide.

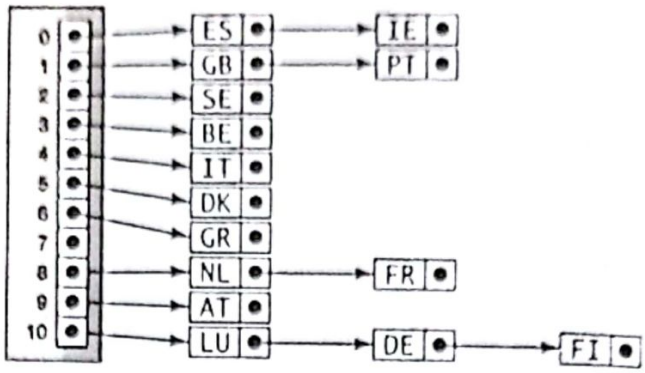

FIGURE 9.7 Closed addressing with chaining.

The risk of nonconstant time sequential searching also exists with open addressing. If most of the keys hash to the same value (a very bad hash function, indeed), then the same kind of long chains can result. Consequently, closed addressing tends to be the choice in many applications.

9.8 The `java.util.HashMap` Class

The standard Java class libraries include a hash table class named `HashMap`, defined in the `java.util` package. It is very similar to the `HashTable` class defined in Listing 9.6 on page 282. It uses closed addressing with a default initial capacity of 101 and a default maximum load factor of 0.75. It implements the `java.util.Map` interface, which is shown in Listing 9.7.

Notice that this interface includes a `public` subinterface named `Entry` to represent key value pairs. As a subinterface, it can be used as a type name in the form `Map.Entry` (like a `static` member) in classes that implement the `Map` interface.

Listing 9.8 shows a program that tests the `java.util.HashMap` class.

The program first instantiates the `HashMap` class as an object named `map`. It puts eight records into it, prints its `keySet` and its `size`, and uses its `get()` method to retrieve the record for the key `"ES"`. Then it repeats the `get()` call at line 15 and assigns its output value to the `Country` object named `es`. Note that it has to use the expression `(Country)` to cast the object as a `Country` object because the `get()` method returns an object of general `Object` type.

Next, the program at line 16 changes the `population` field of the `es` object to 40,000,000. This assignment is done independently of the hash table. Nevertheless, the next call to `get()` shows that the change was made to the corresponding entry in the table. That is because the hash table stores only references to its `Entry` value objects. Any outside reference to them can change the table's values. This is consistent with the notion that a hash table is a generalized array. Objects stored in an array can be externally modified the same way without using the array.

The call to `remove()` deletes the `"ES"` entry from the table and returns its `Country` object. The next call to `get()` at line 19 returns `null`, confirming that it has been deleted from the table. The last two outputs also verify that the record has been removed.

The `java.util` package also includes a `HashTable` class, but that was superseded by the `HashMap` class in Java 1.2 in 1999. It is more consistent with the general Java Collections Framework.

LISTING 9.7: The java.util.Map Interface

```
public interface Map {
  public void       clear();
  public boolean    containsKey(Object key);
  public boolean    containsValue(Object value);
  public Set        entrySet();
  public boolean    equals(Object object);
  public Object     get(Object key);
  public int        hashCode();
  public boolean    isEmpty();
  public Set        keySet();
  public Object     put(Object key, Object value);
  public void       putAll(Map map);
  public Object     remove(Object key);
  public int        size();
  public Collection values();
  public interface Entry {
    public boolean equals(Object object);
    public Object  getKey();
    public Object  getValue();
    public int     hashCode();
    public Object  setValue(Object value);
  }
}
```

LISTING 9.8: Testing the java.util.HashMap Class

```
public class TestMap {
  public static void main(String[] args) {
    java.util.Map map = new java.util.HashMap();
    map.put("AT", new Country("Austria", "German", 32378, 8139299));
    map.put("BE", v Country("Belgium", "Dutch", 11800, 10182034));
    map.put("DK", new Country("Denmark", "Danish", 16639, 5356845));
    map.put("FR", new Country("France", "French", 211200, 58978172));
    map.put("GR", new Country("Greece", "Greek", 50900, 10707135));
    map.put("IE", new Country("Ireland", "English", 27100, 3632944));
    map.put("IT", new Country("Italy", "Italian", 116300, 56735130));
    map.put("ES", new Country("Spain", "Spanish", 194880, 39167744));
    System.out.println("map.keySet(): " + map.keySet());
    System.out.println("map.size(): " + map.size());
    System.out.println("map.get(\"ES\"): " + map.get("ES"));
    Country es = (Country)map.get("ES");
    es.population = 40000000;
    System.out.println("map.get(\"ES\"): " + map.get("ES"));
    System.out.println("map.remove(\"ES\"): " + map.remove("ES"));
    System.out.println("map.get(\"ES\"): " + map.get("ES"));
    System.out.println("map.keySet(): " + map.keySet());
    System.out.println("map.size(): " + map.size());
  }
}
```