CHAPTER

27

Hashing

Objectives

- To understand what hashing is and for what hashing is used (§27.2).
- To obtain the hash code for an object and design the hash function to map a key to an index (§27.3).
- To handle collisions using open addressing (§27.4).
- To know the differences among linear probing, quadratic probing, and double hashing (§27.4).
- To handle collisions using separate chaining (§27.5).
- To understand the load factor and the need for rehashing (§27.6).
- To implement `MyHashMap` using hashing (§27.7).
- To implement `MyHashSet` using hashing (§27.8).

27.1 Introduction

Key Point

Hashing is superefficient. It takes O(1) time to search, insert, and delete an element using hashing.

why hashing?

The preceding chapter introduced binary search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called *hashing*. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time.

27.2 What Is Hashing?

Key Point

Hashing uses a hashing function to map a key to an index.

map
key
value

Before introducing hashing, let us review map, which is a data structure that is implemented using hashing. Recall that a *map* (introduced in Section 21.5) is a container object that stores entries. Each entry contains two parts: a *key* and a *value*. The key, also called a *search key*, is used to search for the corresponding value. For example, a dictionary can be stored in a map, in which the words are the keys and the definitions of the words are the values.

dictionary

associative array

> **Note**
> A map is also called a *dictionary*, a *hash table*, or an *associative array*.

The Java Collections Framework defines the `java.util.Map` interface for modeling maps. Three concrete implementations are `java.util.HashMap`, `java.util.LinkedHashMap`, and `java.util.TreeMap`. `java.util.HashMap` is implemented using hashing, `java.util.LinkedHashMap` using `LinkedList`, and `java.util.TreeMap` using red–black trees. (Bonus Chapter 41 will introduce red–black trees.) You will learn the concept of hashing and use it to implement a hash map in this chapter.

hash table
hash function
hashing

If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So does that mean we can store the values in an array and use the key as the index to find the value? The answer is yes—if you can map a key to an index. The array that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*. As shown in Figure 27.1, a hash function obtains an index from a key and uses the index to retrieve the value for the key. *Hashing* is a technique that retrieves the value using the index obtained from the key without performing a search.

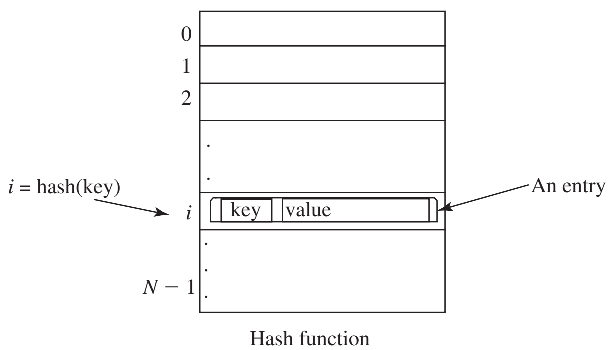

FIGURE 27.1 A hash function maps a key to an index in the hash table.

perfect hash function

How do you design a hash function that produces an index from a key? Ideally, we would like to design a function that maps each search key to a different index in the hash table. Such a function is called a *perfect hash function*. However, it is difficult to find a perfect hash function. When two or more keys are mapped to the same hash value, we say a *collision* has occurred. Although there are ways to deal with collisions, which will be are discussed later in this chapter, it is better to avoid collisions in the first place. Thus, you should design a fast and easy-to-compute hash function that minimizes collisions.

collision

Check Point

27.2.1 What is a hash function? What is a perfect hash function? What is a collision?

27.3 Hash Functions and Hash Codes

Key Point

A typical hash function first converts a search key to an integer value called a hash code, then compresses the hash code into an index to the hash table.

A hash code is a number generated from an object. This code allows an object to be stored/retrieved quickly in a hash table. Java's root class `Object` has the `hashCode()` method, which returns an integer *hash code*. By default, the method returns the memory address for the object. The general contract for the `hashCode` method is as follows:

hash code
`hashCode()`

1. You should override the `hashCode` method whenever the `equals` method is overridden to ensure two equal objects return the same hash code.
2. During the execution of a program, invoking the `hashCode` method multiple times returns the same integer, provided that the object's data are not changed.
3. Two unequal objects may have the same hash code, but you should implement the `hashCode` method to avoid too many such cases.

27.3.1 Hash Codes for Primitive Types

For search keys of the type `byte`, `short`, `int`, and `char`, simply cast them to `int`. Therefore, two different search keys of any one of these types will have different hash codes.

`byte, short, int, char`

For a search key of the type `float`, use `Float.floatToIntBits(key)` as the hash code. Note `floatToIntBits(float f)` returns an `int` value whose bit representation is the same as the bit representation for the floating number `f`. Thus, two different search keys of the `float` type will have different hash codes.

`float`

For a search key of the type `long`, simply casting it to `int` would not be a good choice, because all keys that differ in only the first 32 bits will have the same hash code. To take the first 32 bits into consideration, divide the 64 bits into two halves and perform the exclusive-or operation to combine the two halves. This process is called *folding*. The hash code for a `long` key is

`long`
folding

```
int hashCode = (int)(key ^ (key >> 32));
```

Note `>>` is the right-shift operator that shifts the bits 32 positions to the right. For example, `1010110 >> 2` yields `0010101`. The `^` is the bitwise exclusive-or operator. It operates on two corresponding bits of the binary operands. For example, `1010110 ^ 0110111` yields `1100001`. For more on bitwise operations, see Appendix G, Bitwise Operations.

For a search key of the type `double`, first convert it to a `long` value using the `Double.doubleToLongBits` method, then perform a folding as follows:

`double`
folding

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

27.3.2 Hash Codes for Strings

Search keys are often strings, so it is important to design a good hash function for strings. An intuitive approach is to sum the Unicode of all characters as the hash code for the string. This approach may work if two search keys in an application don't contain the same letters, but it will produce a lot of collisions if the search keys contain the same letters, such as **tod** and **dot**.

A better approach is to generate a hash code that takes the position of characters into consideration. Specifically, let the hash code be

$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \cdots + s_{n-1}$$

polynomial hash code

where s_i is **s.charAt(i)**. This expression is a polynomial for some positive b, so this is called a *polynomial hash code*. Using Horner's rule for polynomial evaluation (see Section 6.7), the hash code can be calculated efficiently as follows:

$$(\cdots((s_0 * b + s_1) * b + s_2) * b + \cdots + s_{n-2}) * b + s_{n-1}$$

This computation can cause an overflow for long strings, but arithmetic overflow is ignored in Java. You should choose an appropriate value b to minimize collisions. Experiments show that good choices for b are 31, 33, 37, 39, and 41. In the **String** class, the **hashCode** is overridden using the polynomial hash code with b being **31**.

27.3.3 Compressing Hash Codes

The hash code for a key can be a large integer that is out of the range for the hash-table index, so you need to scale it down to fit in the index's range. Assume the index for a hash table is between **0** and **N-1**. The most common way to scale an integer to a number between **0** and **N–1** is to use

```
index = hashCode % N;
```

Ideally, you should choose a prime number for **N** to ensure the indices are spread evenly. However, it is time consuming to find a large prime number. In the Java API implementation for **java.util.HashMap**, **N** is set to an integer power of **2**. There is a good reason for this choice. When **N** is an **int** value power of **2**, you can use the **&** operator to compress a hash code to an index on the hash table as follows:

```
index = hashCode & (N – 1);
```

index will be between **0** and **N – 1**. The ampersand, **&**, is a bitwise AND operator (see Appendix G, Bitwise Operations). The AND of two corresponding bits yields a **1** if both bits are **1**. For example, assume **N = 4** and **hashCode = 11**. Thus, **11 & (4 – 1) = 1011 & 0011 = 0011**.

To ensure the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function in the implementation of **java.util.HashMap**. This function is defined as

```
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}
```

^ and **>>>** are bitwise exclusive-or and unsigned right-shift operations (see Appendix G). The bitwise operations are much faster than the multiplication, division, and remainder operations. You should replace these operations with the bitwise operations whenever possible.

The complete hash function is defined as

```
h(hashCode) = supplementalHash(hashCode) & (N – 1)
```

The supplemental hash function helps avoid collisions for two numbers with the same lower bits. For example, both **11100101 & 00000111** and **11001101 & 00000111** yield **00000111**. But **supplementalHash(11100101) & 00000111** and **supplementalHash(11001101) & 00000111** will be different. Using a supplemental function reduces this type of collision.

> **Note**
> In Java, an `int` is a 32-bit signed integer. The `hashCode()` method returns an `int` and it may be negative. If a hash code is negative, `hashCode % N` would be negative. But `hashCode & (N – 1)` will be non-negative for an `int` value `N` because `anyInt & aNonNegativeInt` is always non-negative.

27.3.1 What is a hash code? What is the hash code for `Byte`, `Short`, `Integer`, and `Character`?

27.3.2 How is the hash code for a `Float` object computed?

27.3.3 How is the hash code for a `Long` object computed?

27.3.4 How is the hash code for a `Double` object computed?

27.3.5 How is the hash code for a `String` object computed?

27.3.6 How is a hash code compressed to an integer representing the index in a hash table?

27.3.7 If `N` is an integer power of the power of `2`, is `N / 2` same as `N >> 1`?

27.3.8 If `N` is an integer power of the power of `2`, is `m % N` same as `m & (N – 1)` for a positive integer `m`?

27.3.9 What is `Integer.valueOf("-98").hashCode()` and what is `"ABCDEFGHIJK."hashCode()`?

27.4 Handling Collisions Using Open Addressing

A collision occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: open addressing and separate chaining.

Open addressing is the process of finding an open location in the hash table in the event of a collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.

open addressing

27.4.1 Linear Probing

add entry

linear probing

When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially. For example, if a collision occurs at `hashTable[k % N]`, check whether `hashTable[(k+1) % N]` is available. If not, check `hashTable[(k+2) % N]` and so on, until an available cell is found, as shown in Figure 27.2.

> **Note**
> When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

circular hash table

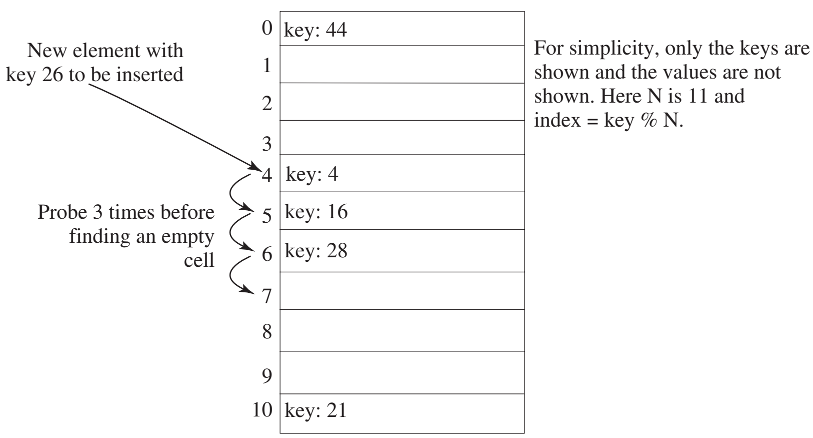

FIGURE 27.2 Linear probing finds the next available location sequentially.

search entry

To search for an entry in the hash table, obtain the index, say `k`, from the hash function for the key. Check whether `hashTable[k % N]` contains the entry. If not, check whether `hashTable[(k+1) % N]` contains the entry, and so on, until it is found, or an empty cell is reached.

remove entry

To remove an entry from the hash table, search the entry that matches the key. If the entry is found, place a special marker to denote that the entry is available. Each cell in the hash table has three possible states: occupied, marked, or empty. Note a marked cell is also available for insertion.

cluster

Linear probing tends to cause groups of consecutive cells in the hash table to be occupied. Each group is called a *cluster*. Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even larger clusters, further slowing down the search time. This is a big disadvantage of linear probing.

linear probing animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how linear probing works, go to http://liveexample.pearsoncmg.com/dsanimation/LinearProbingeBook.html, as shown in Figure 27.3.

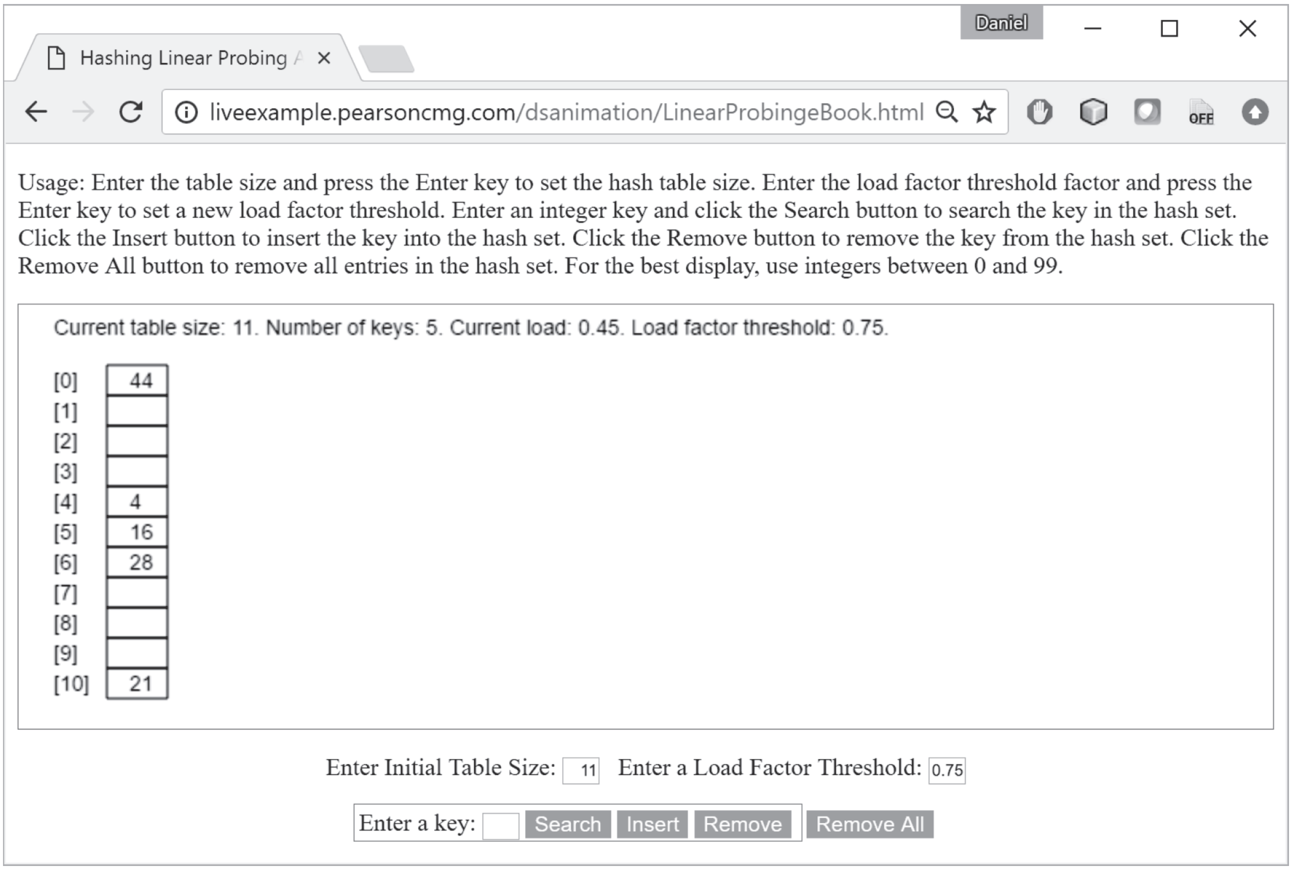

FIGURE 27.3 The animation tool shows how linear probing works.

27.4.2 Quadratic Probing

quadratic probing

Quadratic probing can avoid the clustering problem that can occur in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + 1) \% N$, $(k + 4) \% n$, $(k + 9) \% N$, and so on, as shown in Figure 27.4.

Quadratic probing works in the same way as linear probing except for a change in the search sequence. Quadratic probing avoids linear probing's clustering problem, but it has its own

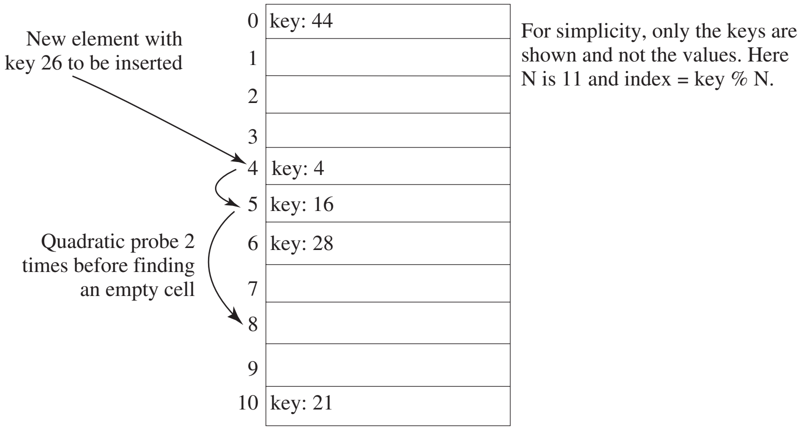

FIGURE 27.4 Quadratic probing increases the next index in the sequence by j^2 for $j = 1, 2, 3, \ldots$.

clustering problem, called *secondary clustering*; that is, the entries that collide with an occupied entry use the same probe sequence.

secondary clustering

Linear probing guarantees that an available cell can be found for insertion as long as the table is not full. However, there is no such guarantee for quadratic probing.

Pedagogical Note

For an interactive GUI demo to see how quadratic probing works, go to http://liveexample.pearsoncmg.com/dsanimation/QuadraticProbingeBook.html, as shown in Figure 27.5.

quadratic probing animation on Companion Website

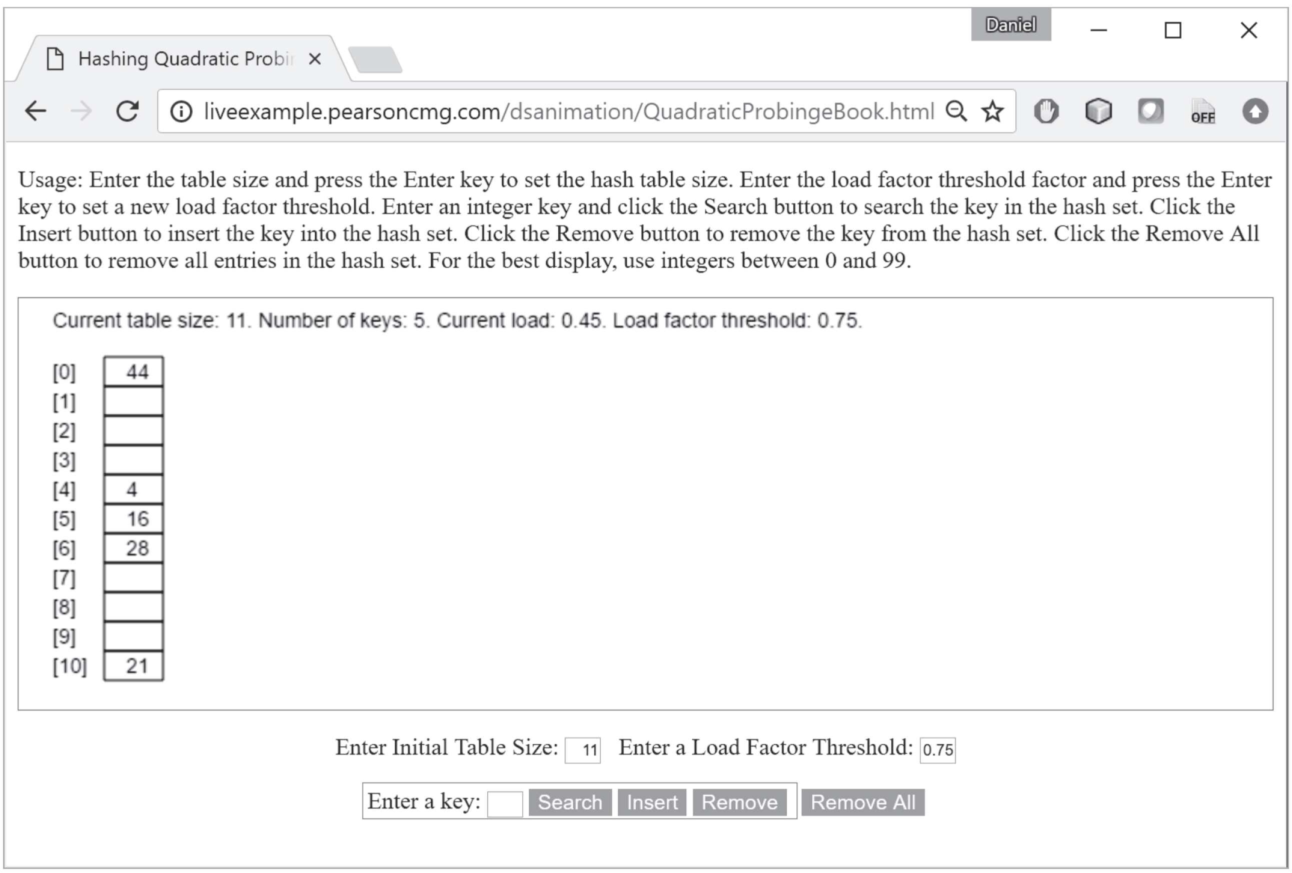

FIGURE 27.5 The animation tool shows how quadratic probing works.

27.4.3 Double Hashing

double hashing

Another open addressing scheme that avoids the clustering problem is known as *double hashing*. Starting from the initial index k, both linear probing and quadratic probing add an increment to k to define a search sequence. The increment is **1** for linear probing and $\mathbf{j^2}$ for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash function $h'(key)$ on the keys to determine the increments to avoid the clustering problem. Specifically, double hashing looks at the cells at indices $(k + j * h'(key))$ % N, for $j \geq 0$, that is, k % N, $(k + h'(key))$ % N, $(k + 2 * h'(key))$ % N, $(k + 3 * h'(key))$ % N, and so on.

For example, let the primary hash function `h` and secondary hash function `h'` on a hash table of size **11** be defined as follows:

```
h(key) = key % 11;
h'(key) = 7 – key % 7;
```

For a search key of **12**, we have

```
h(12) = 12 % 11 = 1;
h'(12) = 7 – 12 % 7 = 2;
```

Suppose the elements with the keys **45**, **58**, **4**, **28**, and **21** are already placed in the hash table as shown in Figure 27.6. We now insert the element with key **12**. The probe sequence for key **12** starts at index **1**. Since the cell at index **1** is already occupied, search the next cell at index **3** `(1 +1 * 2)`. Since the cell at index **3** is already occupied, search the next cell at index **5** `(1 + 2 * 2)`. Since the cell at index **5** is empty, the element for key **12** is now inserted at this cell.

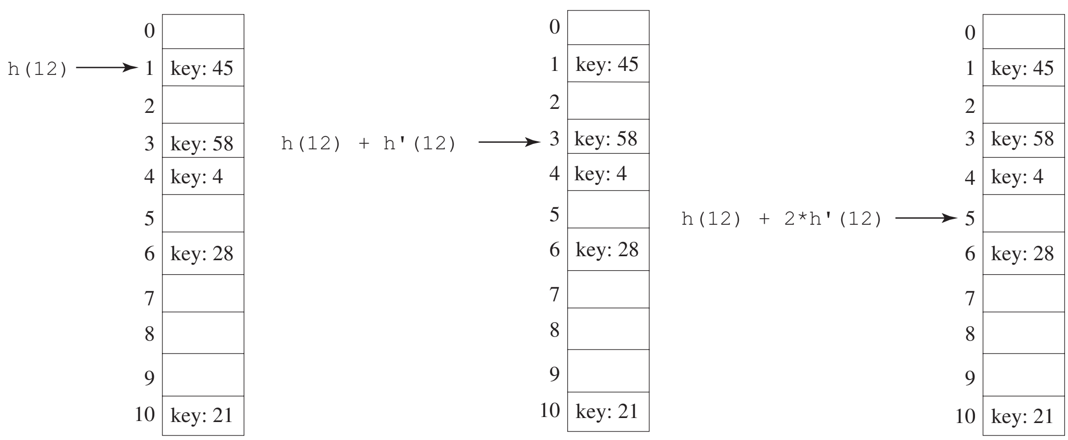

FIGURE 27.6 The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.

The indices of the probe sequence are as follows: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10. This sequence reaches the entire table. You should design your functions to produce a probe sequence that reaches the entire table. Note the second function should never have a zero value, since zero is not an increment.

double hashing animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how double hashing works, go to liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html, as shown in Figure 27.7.

FIGURE 27.7 The animation tool shows how double hashing works.

Check Point

27.4.1 What is open addressing? What is linear probing? What is quadratic probing? What is double hashing?

27.4.2 Describe the clustering problem for linear probing.

27.4.3 What is secondary clustering?

27.4.4 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using linear probing.

27.4.5 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using quadratic probing.

27.4.6 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using double hashing with the following functions:

```
h(k) = k %  11;
h'(k) = 7 – k % 7;
```

27.5 Handling Collisions Using Separate Chaining

The separate chaining scheme places all entries with the same hash index in the same location, rather than finding new locations. Each location in the separate chaining scheme uses a bucket to hold multiple entries.

Key Point

separate chaining
implementing bucket

The preceding section introduced handling collisions using open addressing. The open addressing scheme finds a new location when a collision occurs. This section introduces handling collisions using separate chaining. The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

You can implement a bucket using an array, `ArrayList`, or `LinkedList`. We will use `LinkedList` for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head, as shown in Figure 27.8.

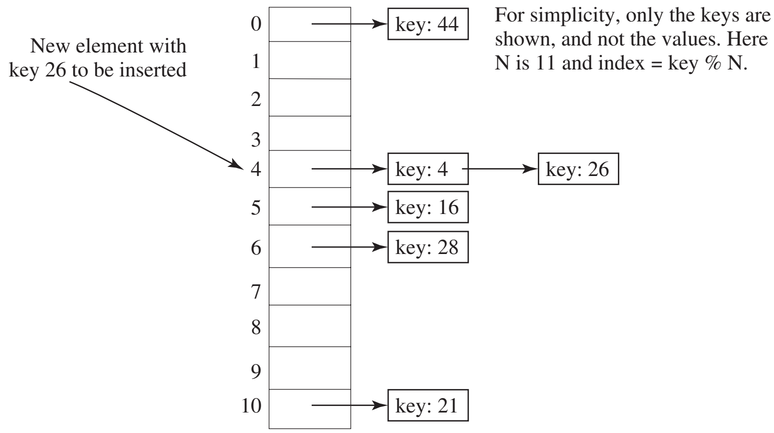

FIGURE 27.8 Separate chaining scheme chains the entries with the same hash index in a bucket.

separate chaining animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how separate chaining works, go to liveexample.pearsoncmg.com/dsanimation/SeparateChainingeBook.html, as shown in Figure 27.9.

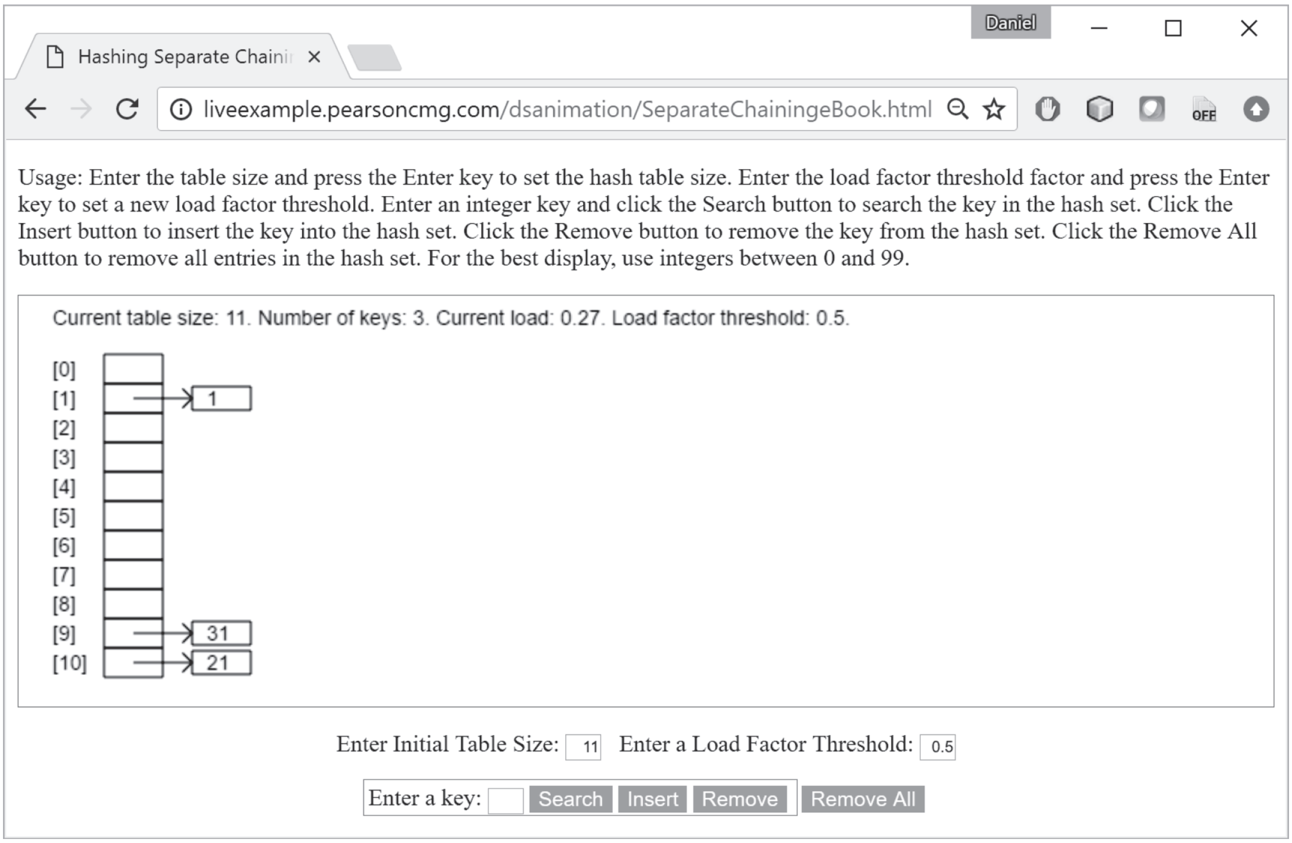

FIGURE 27.9 The animation tool shows how separate chaining works.

27.5.1 Show the hash table of size 11 after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using separate chaining.

Check Point

27.6 Load Factor and Rehashing

Key Point

The load factor measures how full a hash table is. If the load factor is exceeded, increase the hash-table size and reload the entries into a new larger hash table. This is called rehashing.

rehashing

Load factor

Load factor λ (*lambda*) measures how full a hash table is. It is the ratio of the number of elements to the size of the hash table, that is, $\lambda = \frac{n}{N}$, where n denotes the number of elements and N the size of the hash table.

Note λ is zero if the hash table is empty. For the open addressing scheme, λ is between `0` and `1`; λ is `1` if the hash table is full. For the separate chaining scheme, λ can be any value. As λ increases, the probability of a collision also increases. Studies show you should maintain the load factor under `0.5` for the open addressing scheme and under `0.9` for the separate chaining scheme.

threshold

rehash

Keeping the load factor under a certain threshold is important for the performance of hashing. In the implementation of the `java.util.HashMap` class in the Java API, the threshold `0.75` is used. Whenever the load factor exceeds the threshold, you need to increase the hash-table size and *rehash* all the entries in the map into a new larger hash table. Notice you need to change the hash functions, since the hash-table size has been changed. To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even with periodic rehashing, hashing is an efficient implementation for map.

Check Point

27.6.1 What is load factor? Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using linear probing.

27.6.2 Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using quadratic probing.

27.6.3 Assume the hash table has the initial size 4 and its load factor is 0.5; show the hash table after inserting entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40, using separate chaining.

27.7 Implementing a Map Using Hashing

Key Point

A map can be implemented using hashing.

Now you understand the concept of hashing. You know how to design a good hash function to map a key to an index in a hash table, how to measure performance using the load factor, and how to increase the table size and rehash to maintain the performance. This section demonstrates how to implement a map using separate chaining.

We design our custom `Map` interface to mirror `java.util.Map` and name the interface `MyMap` and a concrete class `MyHashMap`, as shown in Figure 27.10.

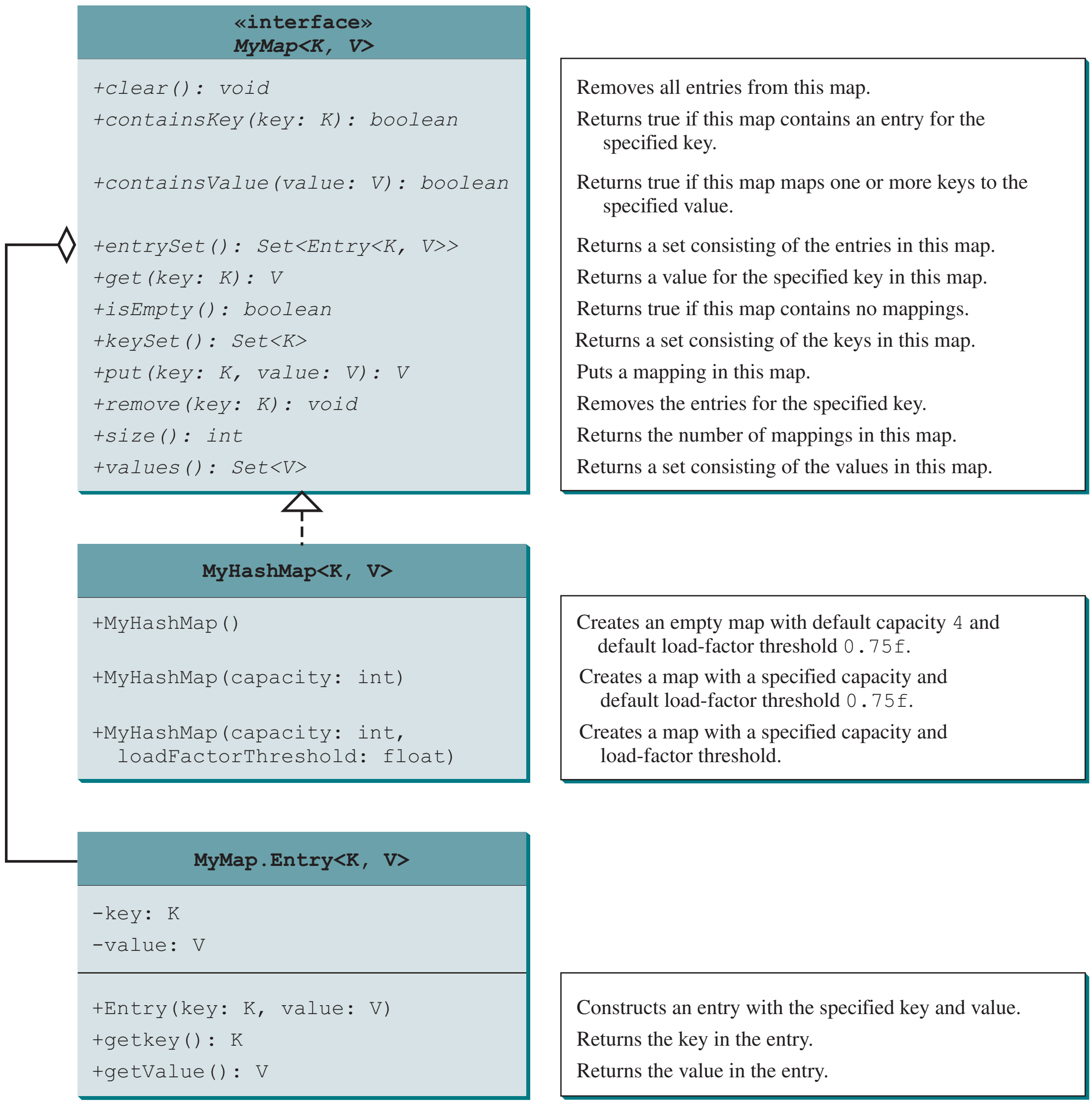

FIGURE 27.10 MyHashMap implements the MyMap interface.

How do you implement MyHashMap? We will use an array for the hash table and each element in the hash table is a bucket. The bucket is a LinkedList. Listing 27.1 shows the MyMap interface and Listing 27.2 implements MyHashMap using separate chaining.

LISTING 27.1 MyMap.java

interface MyMap
clear
containsKey
containsValue

```java
public interface MyMap<K, V> {
  /** Remove all of the entries from this map */
  public void clear();

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);

```

```java
  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();

  /** Return the value that matches the specified key */
  public V get(K key);

  /** Return true if this map doesn't contain any entries */
  public boolean isEmpty();

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

  /** Remove an entry for the specified key */
  public void remove(K key);

  /** Return the number of mappings in this map */
  public int size();

  /** Return a set consisting of the values in this map */
  public java.util.Set<V> values();

  /** Define an inner class for Entry */
  public static class Entry<K, V> {
    K key;
    V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey() {
      return key;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      return "[" + key + ", " + value + "]";
    }
  }
}
```

`entrySet`
`get`
`isEmpty`
`keySet`
`put`
`remove`
`size`
`values`
`Entry` inner class

LISTING 27.2 `MyHashMap.java`

```java
import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
  // Define the default hash-table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash-table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;
```

class `MyHashMap`
default initial capacity
maximum capacity

current capacity
default load factor
load-factor threshold
size
hash table
no-arg constructor
constructor
constructor
clear
containsKey
containsValue

```
  // Current hash-table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor used in the hash table
  private float loadFactorThreshold;

  // The number of entries in the map
  private int size = 0;

  // Hash table is an array with each cell being a linked list
  LinkedList<MyMap.Entry<K,V>>[] table;

  /** Construct a map with the default capacity and load factor */
  public MyHashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity and
   * default load factor */
  public MyHashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity
   * and load factor */
  public MyHashMap(int initialCapacity, float loadFactorThreshold) {
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all of the entries from this map */
  public void clear() {
    size = 0;
    removeEntries();
  }

  @Override /** Return true if the specified key is in the map */
  public boolean containsKey(K key) {
    if (get(key) != null)
      return true;
    else
      return false;
  }

  @Override /** Return true if this map contains the value */
  public boolean containsValue(V value) {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          if (entry.getValue().equals(value))
```

```
            return true;
        }
      }

    return false;
  }

```

entrySet

```
  @Override /** Return a set of entries in the map */
  public java.util.Set<MyMap.Entry<K,V>> entrySet() {
    java.util.Set<MyMap.Entry<K, V>> set =
      new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry);
      }
    }

    return set;
  }

```

get

```
  @Override /** Return the value that matches the specified key */
  public V get(K key) {
    int bucketIndex = hash(key.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key))
          return entry.getValue();
    }

    return null;
  }

```

isEmpty

```
  @Override /** Return true if this map contains no entries */
  public boolean isEmpty() {
    return size == 0;
  }

```

keySet

```
  @Override /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet() {
    java.util.Set<K> set = new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry.getKey());
      }
    }

    return set;
  }

```

put

```
  @Override /** Add an entry (key, value) into the map */
  public V put(K key, V value) {
    if (get(key) != null) { // The key is already in the map
      int bucketIndex = hash(key.hashCode());
```

```
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key)) {
          V oldValue = entry.getValue();
          // Replace old value with new value
          entry.value = value;
          // Return the old value for the key
          return oldValue;
        }
    }

    // Check load factor
    if (size >= capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");

      rehash();
    }

    int bucketIndex = hash(key.hashCode());

    // Create a linked list for the bucket if not already created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<Entry<K, V>>();
    }

    // Add a new entry (key, value) to hashTable[index]
    table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

    size++; // Increase size

    return value;
  }

  @Override /** Remove the entries for the specified key */
```

remove

```
  public void remove(K key) {
    int bucketIndex = hash(key.hashCode());

    // Remove the first entry that matches the key from a bucket
    if (table[bucketIndex] != null) {
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key)) {
          bucket.remove(entry);
          size--; // Decrease size
          break; // Remove just one entry that matches the key
        }
    }
  }

  @Override /** Return the number of entries in this map */
```

size

```
  public int size() {
    return size;
  }

  @Override /** Return a set consisting of the values in this map */
```

values

```
  public java.util.Set<V> values() {
    java.util.Set<V> set = new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
```

```
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry.getValue());
      }
    }

    return set;
  }

  /** Hash function */
```

hash

```
  private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
  }

  /** Ensure the hashing is evenly distributed */
```

supplementalHash

```
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }

  /** Return a power of 2 for initialCapacity */
```

trimToPowerOf2

```
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    }

    return capacity;
  }

  /** Remove all entries from each bucket */
```

removeEntries

```
  private void removeEntries() {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        table[i].clear();
      }
    }
  }

  /** Rehash the map */
```

rehash

```
  private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size to 0

    for (Entry<K, V> entry: set) {
      put(entry.getKey(), entry.getValue()); // Store to new table
    }
  }

  @Override /** Return a string representation for this map */
```

toString

```
  public String toString() {
    StringBuilder builder = new StringBuilder("[");

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null && table[i].size() > 0)
        for (Entry<K, V> entry: table[i])
          builder.append(entry);
```

```
249     }
250
251     builder.append("]");
252     return builder.toString();
253   }
254 }
```

hash-table parameters

The `MyHashMap` class implements the `MyMap` interface using separate chaining. The parameters that determine the hash-table size and load factors are defined in the class. The default initial capacity is `4` (line 5) and the maximum capacity is 2^{30} (line 8). The current hash-table capacity is designed as a value of the power of `2` (line 11). The default load-factor threshold is `0.75f` (line 14). You can specify a custom load-factor threshold when constructing a map. The custom load-factor threshold is stored in `loadFactorThreshold` (line 17). The data field `size` denotes the number of entries in the map (line 20). The hash table is an array. Each cell in the array is a linked list (line 23).

Three constructors

Three constructors are provided to construct a map. You can construct a default map with the default capacity and load-factor threshold using the no-arg constructor (lines 26–28), a map with the specified capacity and a default load-factor threshold (lines 32–34), and a map with the specified capacity and load-factor threshold (lines 38–46).

`clear`

The `clear` method removes all entries from the map (lines 49–52). It invokes `removeEntries()`, which deletes all entries in the buckets (lines 221–227). The `removeEntries()` method takes $O(capacity)$ time to clear all entries in the table.

`containsKey`

The `containsKey(key)` method checks whether the specified key is in the map by invoking the `get` method (lines 55–60). Since the `get` method takes $O(1)$ time, the `containsKey(key)` method takes $O(1)$ time.

`containsValue`

The `containsValue(value)` method checks whether the value is in the map (lines 63–74). This method takes $O(capacity + size)$ time. It is actually $O(capacity)$, since $capacity > size$.

`entrySet`

The `entrySet()` method returns a set that contains all entries in the map (lines 77–90). This method takes $O(capacity)$ time.

`get`

The `get(key)` method returns the value of the first entry with the specified key (lines 93–103). This method takes $O(1)$ time.

`isEmpty`

The `isEmpty()` method simply returns true if the map is empty (lines 106–108). This method takes $O(1)$ time.

`keySet`

The `keySet()` method returns all keys in the map as a set. The method finds the keys from each bucket and adds them to a set (lines 111–123). This method takes $O(capacity)$ time.

`put`

The `put(key, value)` method adds a new entry into the map. The method first tests if the key is already in the map (line 127), if so, it locates the entry and replaces the old value with the new value in the entry for the key (line 134) and the old value is returned (line 136). If the key is new in the map, the new entry is created in the map (line 156). Before inserting the new entry, the method checks whether the size exceeds the load-factor threshold (line 141). If so, the program invokes `rehash()` (line 145) to increase the capacity and store entries into a new larger hash table.

`rehash`

The `rehash()` method first copies all entries in a set (line 231), doubles the capacity (line 232), creates a new hash table (line 233), and resets the size to `0` (line 234). The method then copies the entries into the new hash table (lines 236–238). The `rehash` method takes $O(capacity)$ time. If no rehash is performed, the `put` method takes $O(1)$ time to add a new entry.

`remove`

The `remove(key)` method removes the entry with the specified key in the map (lines 164–177). This method takes $O(1)$ time.

`size`

The `size()` method simply returns the size of the map (lines 180–182). This method takes $O(1)$ time.

`values`

The `values()` method returns all values in the map. The method examines each entry from all buckets and adds it to a set (lines 185–197). This method takes $O(capacity)$ time.

`hash`

The `hash()` method invokes the `supplementalHash` to ensure the hashing is evenly distributed to produce an index for the hash table (lines 200–208). This method takes $O(1)$ time.

Table 27.1 summarizes the time complexities of the methods in **MyHashMap**.

Since rehashing does not happen very often, the time complexity for the **put** method is $O(1)$. Note the complexities of the **clear**, **entrySet**, **keySet**, **values**, and **rehash** methods depend on **capacity**, so to avoid poor performance for these methods, you should choose an initial capacity carefully.

TABLE 27.1 Time Complexities for Methods in **MyHashMap**

Methods	*Time*
`clear()`	$O(capacity)$
`containsKey(key: Key)`	$O(1)$
`containsValue(value: V)`	$O(capacity)$
`entrySet()`	$O(capacity)$
`get(key: K)`	$O(1)$
`isEmpty()`	$O(1)$
`keySet()`	$O(capacity)$
`put(key: K, value: V)`	$O(1)$
`remove(key: K)`	$O(1)$
`size()`	$O(1)$
`values()`	$O(capacity)$
`rehash()`	$O(capacity)$

Listing 27.3 gives a test program that uses **MyHashMap**.

LISTING 27.3 TestMyHashMap.java

```java
public class TestMyHashMap {
  public static void main(String[] args) {
    // Create a map
    MyMap<String, Integer> map = new MyHashMap<>();
    map.put("Smith", 30);
    map.put("Anderson", 31);
    map.put("Lewis", 29);
    map.put("Cook", 29);
    map.put("Smith", 65);

    System.out.println("Entries in map: " + map);

    System.out.println("The age for Lewis is " +
      map.get("Lewis"));

    System.out.println("Is Smith in the map? " +
      map.containsKey("Smith"));
    System.out.println("Is age 33 in the map? " +
      map.containsValue(33));

    map.remove("Smith");
    System.out.println("Entries in map: " + map);

    map.clear();
    System.out.println("Entries in map: " + map);
  }
}
```

create a map
put entries
display entries
get value
is key in map?
is value in map?
remove entry

```
Entries in map: [[Anderson, 31][Smith, 65][Lewis, 29][Cook, 29]]
The age for Lewis is 29
Is Smith in the map? true
Is age 33 in the map? false
Entries in map: [[Anderson, 31][Lewis, 29][Cook, 29]]
Entries in map: []
```

The program creates a map using `MyHashMap` (line 4) and adds five entries into the map (lines 5–9). Line 5 adds key `Smith` with value `30` and line 9 adds `Smith` with value `65`. The latter value replaces the former value. The map actually has only four entries. The program displays the entries in the map (line 11), gets a value for a key (line 14), checks whether the map contains the key (line 17) and a value (line 19), removes an entry with the key `Smith` (line 21), and redisplays the entries in the map (line 22). Finally, the program clears the map (line 24) and displays an empty map (line 25).

Check Point

27.7.1 What is `1 << 30` in line 8 in Listing 27.2? What are the integers resulted from `1 << 1`, `1 << 2`, and `1 << 3`?

27.7.2 What are the integers resulted from `32 >> 1`, `32 >> 2`, `32 >> 3`, and `32 >> 4`?

27.7.3 In Listing 27.2, will the program work if `LinkedList` is replaced by `ArrayList`? In Listing 27.2, how do you replace the code in lines 56–59 using one line of code?

27.7.4 Describe how the `put(key, value)` method is implemented in the `MyHashMap` class.

27.7.5 In Listing 27.2, the `supplementalHash` method is declared static. Can the `hash` method be declared static?

27.7.6 Show the output of the following code:

```
MyMap<String, String> map = new MyHashMap<>();
map.put("Texas", "Dallas");
map.put("Oklahoma", "Norman");
map.put("Texas", "Austin");
map.put("Oklahoma", "Tulsa");

System.out.println(map.get("Texas"));
System.out.println(map.size());
```

27.7.7 If `x` is a negative `int` value, will `x & (N – 1)` be negative?

hash set
hash map
set

27.8 Implementing Set Using Hashing

Key Point

A hash set can be implemented using a hash map.

A *set* (introduced in Chapter 21) is a data structure that stores distinct values. The Java Collections Framework defines the `java.util.Set` interface for modeling sets. Three concrete implementations are `java.util.HashSet`, `java.util.LinkedHashSet`, and `java.util.TreeSet`. `java.util.HashSet` is implemented using hashing, `java.util.LinkedHashSet` using `LinkedList`, and `java.util.TreeSet` using binary search trees.

You can implement `MyHashSet` using the same approach as for implementing `MyHashMap`. The only difference is that key/value pairs are stored in the map, while elements are stored in the set.

MyHashSet

Since all the methods in `HashSet` are inherited from `Collection`, we design our custom `HashSet` by implementing the `Collection` interface, as shown in Figure 27.11.

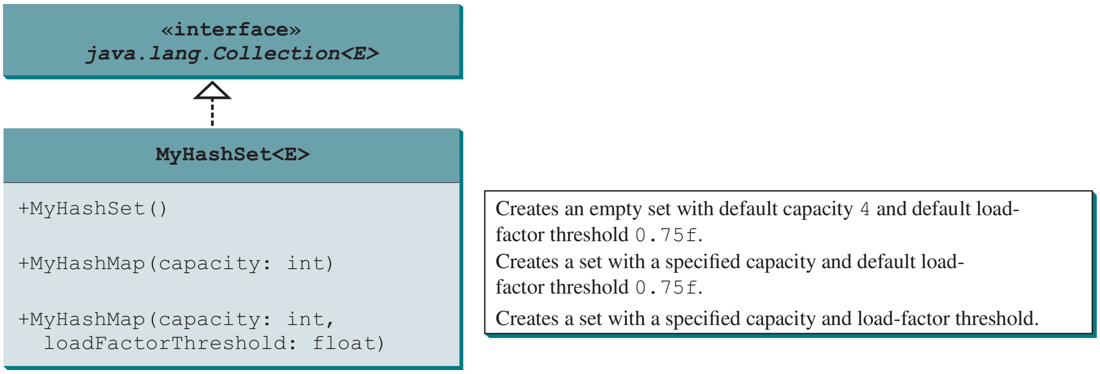

FIGURE 27.11 `MyHashSet` implements the `Collection` interface.

Listing 27.4 implements `MyHashSet` using separate chaining.

LISTING 27.4 MyHashSet.java

```
import java.util.*;

public class MyHashSet<E> implements Collection<E> {
  // Define the default hash-table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash-table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash-table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load-factor threshold used in the hash table
  private float loadFactorThreshold;

  // The number of elements in the set
  private int size = 0;

  // Hash table is an array with each cell being a linked list
  private LinkedList<E>[] table;

  /** Construct a set with the default capacity and load factor */
  public MyHashSet() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity and
   * default load factor */
  public MyHashSet(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity
   * and load factor */
  public MyHashSet(int initialCapacity, float loadFactorThreshold) {
```

class `MyHashSet`
default initial capacity
maximum capacity
current capacity
default max load factor
load-factor threshold
size
hash table
no-arg constructor
constructor
constructor

```
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all elements from this set */
```

clear

```
  public void clear() {
    size = 0;
    removeElements();
  }

  @Override /** Return true if the element is in the set */
```

contains

```
  public boolean contains(E e) {
    int bucketIndex = hash(e.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      return bucket.contains(e);
    }

    return false;
  }

  @Override /** Add an element to the set */
```

add

```
  public boolean add(E e) {
    if (contains(e))  // Duplicate element not stored
      return false;

    if (size + 1 > capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");

      rehash();
    }

    int bucketIndex = hash(e.hashCode());

    // Create a linked list for the bucket if not already created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<E>();
    }

    // Add e to hashTable[index]
    table[bucketIndex].add(e);

    size++; // Increase size

    return true;
  }

  @Override /** Remove the element from the set */
```

remove

```
  public boolean remove(E e) {
    if (!contains(e))
      return false;

    int bucketIndex = hash(e.hashCode());

```

```
    // Create a linked list for the bucket if not already created
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      bucket.removed(e);
    }

    size--; // Decrease size

    return true;
  }

  @Override /** Return true if the set contain no elements */
  public boolean isEmpty() {
    return size == 0;
  }

  @Override /** Return the number of elements in the set */
  public int size() {
    return size;
  }

  @Override /** Return an iterator for the elements in this set */
  public java.util.Iterator<E> iterator() {
    return new MyHashSetIterator(this);
  }

  /** Inner class for iterator */
  private class MyHashSetIterator implements java.util.Iterator<E> {
    // Store the elements in a list
    private java.util.ArrayList<E> list;
    private int current = 0; // Point to the current element in list
    private MyHashSet<E> set;

    /** Create a list from the set */
    public MyHashSetIterator(MyHashSet<E> set) {
      this.set = set;
      list = setToList();
    }

    @Override /** Next element for traversing? */
    public boolean hasNext() {
      return current < list.size();
    }

    @Override /** Get current element and move cursor to the next */
    public E next() {
      return list.get(current++);
    }

    /** Remove the current element returned by the last next() */
    public void remove() {
      // Left as an exercise
      // You need to remove the element from the set
      // You also need to remove it from the list
    }
  }

  /** Hash function */
  private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
```

isEmpty

size

iterator

inner class

hash

```
  }

  /** Ensure the hashing is evenly distributed */
```

supplementalHash

```
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }

  /** Return a power of 2 for initialCapacity */
```

trimToPowerOf2

```
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    }

    return capacity;
  }

  /** Remove all e from each bucket */
  private void removeElements() {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        table[i].clear();
      }
    }
  }

  /** Rehash the set */
```

rehash

```
  private void rehash() {
    java.util.ArrayList<E> list = setToList(); // Copy to a list
    capacity <<= 1; // Same as capacity *= 2. <= is more efficient
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0;

    for (E element: list) {
      add(element); // Add from the old table to the new table
    }
  }

  /** Copy elements in the hash set to an array list */
```

setToList

```
  private java.util.ArrayList<E> setToList() {
    java.util.ArrayList<E> list = new java.util.ArrayList<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        for (E e: table[i]) {
          list.add(e);
        }
      }
    }

    return list;
  }

  @Override /** Return a string representation for this set */
```

toString

```
  public String toString() {
    java.util.ArrayList<E> list = setToList();
    StringBuilder builder = new StringBuilder("[");

    // Add the elements except the last one to the string builder
```

```
    for (int i = 0; i < list.size() - 1; i++) {
      builder.append(list.get(i) + ", ");
    }

    // Add the last element in the list to the string builder
    if (list.size() == 0)
      builder.append("]");
    else
      builder.append(list.get(list.size() - 1) + "]");

    return builder.toString();
  }

  @Override
  public boolean addAll(Collection<? extends E> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean containsAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean removeAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean retainAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public <T> T[] toArray(T[] arg0) {
    // Left as an exercise
    return null;
  }
}
```

override `addAll`

override `containsAll`

override `removeAll`

override `retainAll`

override `toArray()`

`MyHashSet` vs. `MyHashMap`

The **`MyHashSet`** class implements the **`MySet`** interface using separate chaining. Implementing **`MyHashSet`** is very similar to implementing **`MyHashMap`** except for the following differences:

1. The elements are stored in the hash table for **`MyHashSet`**, but the entries (key/value pairs) are stored in the hash table for **`MyHashMap`**.

2. **`MyHashSet`** implements **`Collection`**. Since **`Collection`** implements **`Iterable`**, the elements in **`MyHashSet`** are iterable.

three constructors

Three constructors are provided to construct a set. You can construct a default set with the default capacity and load factor using the no-arg constructor (lines 26–28), a set with the specified capacity and a default load factor (lines 32–34), and a set with the specified capacity and load factor (lines 38–46).

clear

The **clear** method removes all elements from the set (lines 49–52). It invokes **removeElements()**, which clears all table cells (line 181). Each table cell is a linked list that stores the elements with the same hash table index. The **removeElements()** method takes $O(capacity)$ time.

contains

The **contains(element)** method checks whether the specified element is in the set by examining whether the designated bucket contains the element (line 59). This method takes $O(1)$ time because the bucket size is considered very small.

add

The **add(element)** method adds a new element into the set. The method first checks if the element is already in the set (line 67). If so, the method returns false. The method then checks whether the size exceeds the load-factor threshold (line 70). If so, the program invokes **rehash()** (line 74) to increase the capacity and store elements into a new larger hash table.

rehash

The **rehash()** method first copies all elements to a list (line 188), doubles the capacity (line 189), creates a new hash table (line 190), and resets the size to **0** (line 191). The method then copies the elements into the new larger hash table (lines 193–195). The **rehash** method takes $O(capacity)$ time. If no rehash is performed, the **add** method takes $O(1)$ time to add a new element.

remove

The **remove(element)** method removes the specified element in the set (lines 93–108). This method takes $O(1)$ time.

size

The **size()** method simply returns the number of elements in the set (lines 116–118). This method takes $O(1)$ time.

iterator

The **iterator()** method returns an instance of **java.util.Iterator**. The **MyHashSetIterator** class implements **java.util.Iterator** to create a forward iterator. When a **MyHashSetIterator** is constructed, it copies all the elements in the set to a list (line 135). The variable **current** points to the element in the list. Initially, **current** is **0** (line 129), which points to the first element in the list. **MyHashSetIterator** implements the methods **hasNext()**, **next()**, and **remove()** in **java.util.Iterator**. Invoking **hasNext()** returns true if **current < list.size()**. Invoking **next()** returns the current element and moves **current** to point to the next element (line 145). Invoking **remove()** removes the element called by the last **next()**.

hash

The **hash()** method invokes the **supplementalHash** to ensure the hashing is evenly distributed to produce an index for the hash table (lines 157–159). This method takes $O(1)$ time.

The methods **containsAll**, **addAll**, **removeAll**, **retainAll**, **toArray()**, and **toArray(T[])** defined in the **Collection** interface are overridden in **MyHashSet**. Their implementations are left as exercises in Programming Exercise 27.11.

Table 27.2 summarizes the time complexity of the methods in **MyHashSet**.

Table 27.2 Time Complexities for Methods in **MyHashSet**

Methods	*Time*
clear()	$O(capacity)$
contains(e: E)	$O(1)$
add(e: E)	$O(1)$
remove(e: E)	$O(1)$
isEmpty()	$O(1)$
size()	$O(1)$
iterator()	$O(capacity)$
rehash()	$O(capacity)$

Listing 27.5 gives a test program that uses **MyHashSet**.

LISTING 27.5 TestMyHashSet.java

```
1  public class TestMyHashSet {
2    public static void main(String[] args) {
3      // Create a MyHashSet
4      java.util.Collection<String> set = new MyHashSet<>();
5      set.add("Smith");
6      set.add("Anderson");
7      set.add("Lewis");
8      set.add("Cook");
9      set.add("Smith");
10
11     System.out.println("Elements in set: " + set);
12     System.out.println("Number of elements in set: " + set.size());
13     System.out.println("Is Smith in set? " + set.contains("Smith"));
14
15     set.remove("Smith");
16     System.out.print("Names in set in uppercase are ");
17     for (String s: set)
18       System.out.print(s.toUpperCase() + " ");
19
20     set.clear();
21     System.out.println("\nElements in set: " + set);
22   }
23 }
```

create a set
add elements
display elements
set size
remove element
foreach loop
clear set

```
Elements in set: [Cook, Anderson, Smith, Lewis]
Number of elements in set: 4
Is Smith in set? true
Names in set in uppercase are COOK ANDERSON LEWIS
Elements in set: []
```

The program creates a set using **MyHashSet** (line 4) and adds five elements to the set (lines 5–9). Line 5 adds **Smith** and line 9 adds **Smith** again. Since only nonduplicate elements are stored in the set, **Smith** appears in the set only once. The set actually has four elements. The program displays the elements (line 11), gets its size (line 12), checks whether the set contains a specified element (line 13), and removes an element (line 15). Since the elements in a set are iterable, a foreach loop is used to traverse all elements in the set (lines 17–18). Finally, the program clears the set (line 20) and displays an empty set (line 21).

Check Point

27.8.1 Why can you use a foreach loop to traverse the elements in a set?

27.8.2 Describe how the **add(e)** method is implemented in the **MyHashSet** class.

27.8.3 Can lines 100–103 in Listing 27.4 be removed?

27.8.4 Implement the **remove()** method in lines 150–152?

KEY TERMS

associative array 1038
cluster 1042
dictionary 1038
double hashing 1044
hash code 1039
hash function 1038
hash map 1056
hash set 1056
hash table 1038
linear probing 1041

load factor 1047
open addressing 1041
perfect hash function 1038
polynomial hash code 1040
quadratic probing 1042
rehashing 1047
separate chaining 1045

Chapter Summary

1. A *map* is a data structure that stores entries. Each entry contains two parts: a *key* and a *value*. The key is also called a *search key*, which is used to search for the corresponding value. You can implement a map to obtain $O(1)$ time complexity on searching, retrieval, insertion, and deletion using the hashing technique.

2. A *set* is a data structure that stores elements. You can use the hashing technique to implement a set to achieve $O(1)$ time complexity on searching, insertion, and deletion for a set.

3. *Hashing* is a technique that retrieves the value using the index obtained from a key without performing a search. A typical *hash function* first converts a search key to an integer value called a *hash code*, then compresses the hash code into an index to the *hash table*.

4. A *collision* occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: *open addressing* and *separate chaining*.

5. Open addressing is the process of finding an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.

6. The *separate chaining* scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

****27.1** (*Implement `MyMap` using open addressing with linear probing*) Create a new concrete class that implements `MyMap` using open addressing with linear probing. For simplicity, use `f(key) = key % size` as the hash function, where size is the hash-table size. Initially, the hash-table size is `6`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.2** (*Implement `MyMap` using open addressing with quadratic probing*) Create a new concrete class that implements `MyMap` using open addressing with quadratic probing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `6`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.3** (*Implement `MyMap` using open addressing with double hashing*) Create a new concrete class that implements `MyMap` using open addressing with double hashing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `6`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.4** (*Modify `MyHashMap` with duplicate keys*) Modify `MyHashMap` to allow duplicate keys for entries. You need to modify the implementation for the `put(key, value)` method. Also add a new method named `getAll(key)` that returns a set of values that match the key in the map.

****27.5** (*Implement `MyHashSet` using `MyHashMap`*) Implement `MyHashSet` using `MyHashMap`. Note you can create entries with (`key, key`), rather than (`key, value`).

****27.6** (*Animate linear probing*) Write a program that animates linear probing, as shown in Figure 27.3. You can change the initial size of the hash-table in the program. Assume that the load-factor threshold is `0.8`.

****27.7** (*Animate separate chaining*) Write a program that animates `MyHashMap`, as shown in Figure 27.8. You can change the initial size of the table. Assume that the load factor threshold is `0.8`.

****27.8** (*Animate quadratic probing*) Write a program that animates quadratic probing, as shown in Figure 27.5. You can change the initial size of the hash-table in the program. Assume the load-factor threshold to be `0.8`.

****27.9** (*Implement hashCode for string*) Write a method that returns a hash code for string using the approach described in Section 27.3.2 with `b` value `31`. The function header is as follows:

```
public static int hashCodeForString(String s)
```

****27.10** (*Compare `MyHashSet` and `MyArrayList`*) `MyArrayList` is defined in Listing 24.2. Write a program that generates `1000000` random double values between `0` and `999999` and stores them in a `MyArrayList` and in a `MyHashSet`. Generate a list of `1000000` random double values between `0` and `1999999`. For each number in the list, test if it is in the array list and in the hash set. Run your program to display the total test time for the array list and for the hash set.

****27.11** (*Implement set operations in `MyHashSet`*) The implementations of the methods `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` are omitted in the `MyHashSet` class. Implement these methods. Also add a new constructor `MyHashSet(E[] list)` in the `MyHashSet` class. Test your new `MyHashSet` class using the code at https://liveexample.pearsoncmg.com/test/Exercise27_11.txt.

****27.12** (`setToList`) Write the following method that returns an `ArrayList` from a set:

```
public static <E> ArrayList<E> setToList(Set<E> s)
```

***27.13** (*The `Date` class*) Design a class named `Date` that meets the following requirements:

- Three data fields `year`, `month`, and `day` for representing a date
- A constructor that constructs a date with the specified year, month, and day
- Override the `equals` method
- Override the `hashCode` method. (For reference, see the implementation of the `Date` class in the Java API.)

***27.14** (*The `Point` class*) Design a class named `Point` that meets the following requirements:

- Two data fields `x` and `y` for representing a point with getter methods
- A no-arg constructor that constructs a point for (`0`, `0`)

- A constructor that constructs a point with the specified **x** and **y** values
- Override the **equals** method. Point **p1** is said to be equal to point **p2** if `p1.x == p2.x` and `p1.y == p2.y`.
- Override the **hashCode** method. (For reference, see the implementation of the **Point2D** class in the Java API.)

***27.15** (*Modify Listing 27.4 MyHashSet.java*) The book uses **LinkedList** for buckets. Replace **LinkedList** with **AVLTree**. Assume **E** is **Comparable**. Redefine **MyHashSet** as follows:

```
public class MyHashSet<E extends Comparable<E>> implements
    Collection<E> {
...
}
```

Test your program using the main method in Listing 27.5.