CHAPTER 48

HASHING

Objectives

- To know what hashing is for (§48.3).
- To obtain the hash code for an object and design the hash function to map a key to an index (§48.4).
- To handle collisions using open addressing (§48.5).
- To know the differences among linear probing, quadratic probing, and double hashing (§48.5).
- To handle collisions using separate chaining (§48.6).
- To understand the load factor and the need for rehashing (§48.7).
- To implement `MyHashMap` using hashing (§48.8).

48.1 Introduction

why hashing?

The preceding chapters introduced 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.

48.2 Map

map
key
value

Recall that a *map* is a data structure that stores entries. Each entry contains two parts: *key* and *value*. The key is also called a *search key*, which is used to search for the corresponding value. For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.

dictionary
hash table
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. You will learn the concept of hashing and use it to implement a map in this chapter. In the chapter exercises, you will implement `LinkedHashMap` and `TreeMap`.

48.3 What is Hashing?

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, can we 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 48.1, 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.

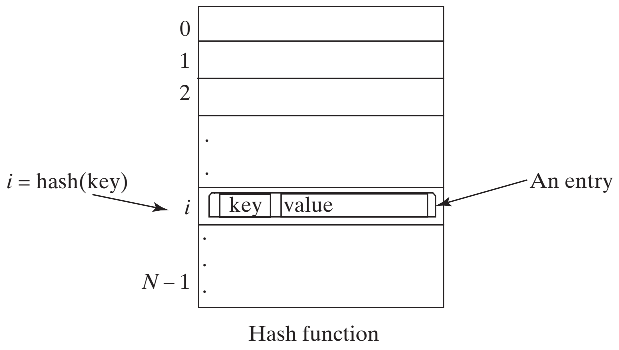


FIGURE 48.1 A hash function maps a key to an index in the hash table.

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 that a *collision* has occurred. We will discuss how to deal with collisions later. Although there are ways to deal with them, it is better to avoid collisions in the first place. So, you should design a fast and easy-to-compute hash function that minimizes collisions.

collision

48.4 Hash Functions and Hash Codes

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.

hash code

Java's root class `Object` has the `hashCode` method that returns an integer hash code. By default, the method returns the memory address for the object. The general contract for the `hashCode` is as follows:

- You should override the `hashCode` method whenever the `equals` method is overridden to ensure that two equal objects return the same hash code.
- 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.
- Two unequal objects may have the same hash code, but you should implement the `hashCode` method to avoid too many such cases.

48.4.1 Hash Codes for Primitive Types

For a search key of the type `byte`, `short`, `int`, and `char`, simply cast it to `int`. So, 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 that `floatToIntBits(float f)` returns an `int` value whose bit representation is the same as the bit representation for the floating number `f`. So, 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 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*. So, the hashing code is

`long`

folding

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

Note that `>>` is the right-shift operator that shifts the bits 32 position 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 a search key of the type `double`, first convert it to a `long` value using `doubleToLongBits`, then perform a folding as follows:

`double`

folding

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

48.4.2 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 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*. By Horner's rule, it can be evaluated efficiently as follows:

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

This computation can cause an overflow for long strings. Arithmetic overflow is ignored in Java. You should choose an appropriate value b to minimize collision. Experiments show that the 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`.

48.4.3 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. You need to scale it down to fit in the range of the index. Assume the index for a hash table is between `0` and `N-1`. The most common way to scale an integer to between `0` and `N-1` is to use

```
h(hashCode) = hashCode % N
```

To ensure that the indices are spread evenly, choose `N` to be a prime number greater than `2`.

Ideally you should choose a prime number for `N`. However, it is time consuming to find a large prime number. In the Java API implementation for `java.util.HashMap`, `N` is conveniently set to a value of power `2`. To ensure the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function. The supplemental function is defined as follows:

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

`^` and `>>>` are bitwise exclusive-or and right-shift operations. See Supplement Part III.D, "Bitwise Operations," on the Companion Website.

The primary hash function is defined as follows:

```
h(hashCode) = supplementalHash(hashCode) % N
```

Note that the function can also be written as

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

since `N` is a power of `2`.

48.5 Handling Collisions Using Open Addressing

open addressing
separate chaining
linear probing
quadratic probing
double hashing

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

48.5.1 Linear Probing

add entry

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 48.2.

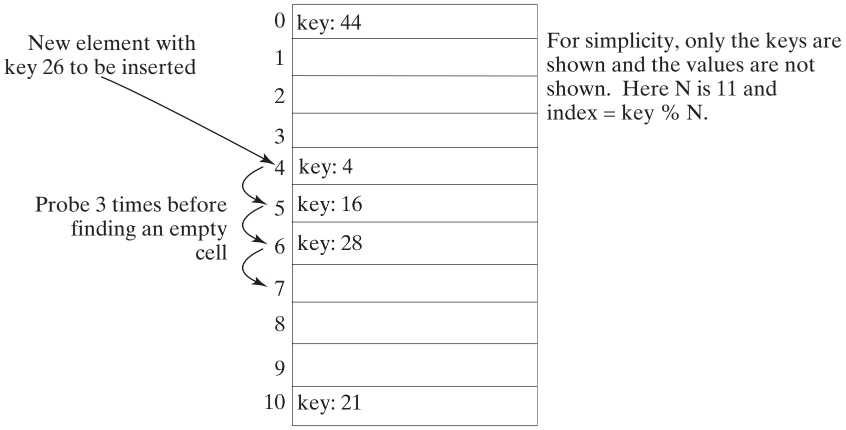


FIGURE 48.2 Linear probe finds the next available location sequentially.

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

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 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, available, or empty. Note that an empty 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.

Pedagogical Note

Follow the link www.cs.armstrong.edu/liang/animation/HashingLinearProbingAnimation.html to see how to hashing with linear probing works, as shown in Figure 48.3.

linear probing animation

48.5.2 Quadratic Probing

Quadratic probing can avoid the clustering problem 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 $j \geq 0$, i.e., `k`, `(k + 1) % n`, `(k + 4) % n`, `(k + 9) % n`, . . . , and so on, as shown in Figure 48.4.

secondary clustering

Quadratic probing works in the same way as linear probing except for the change of search sequence. Quadratic probing avoids the clustering problem in linear probing, but it has its own clustering problem, called *secondary clustering*; i.e., the entries that collide with an occupied entry use the same probe sequence.

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.

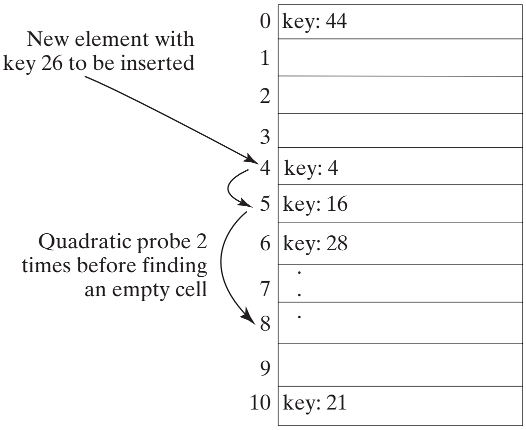
Hashing Using Open Address and Linear Probing Animation by Y. Daniel Liang (Note: the keys are integers)

```
Table size = 11. Number of keys = 5
Load factor = 0.45454545454545453. Load factor threshold = 0.5.

[0]  44
[1]
[2]
[3]
[4]  4
[5]  16
[6]  28
[7]
[8]
[9]
[10] 21
```

Enter initial table size: 11 Enter a value: 21

FIGURE 48.3 The animation tool shows how linear probing works.



New element with key 26 to be inserted

Quadratic probe 2 times before finding an empty cell

For simplicity, only the keys are shown and not the e values. Here N is 11 and index = key % N.

FIGURE 48.4 Quadratic probe increases the next index in the sequence by j^2 for $j = 1, 2, 3, \ldots$.

48.5.3 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 j^2 for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

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

For a search key of `12`, we have

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

The probe sequence for key `12` starts at index `1` with an increment `2`, as shown in Figure 48.5.

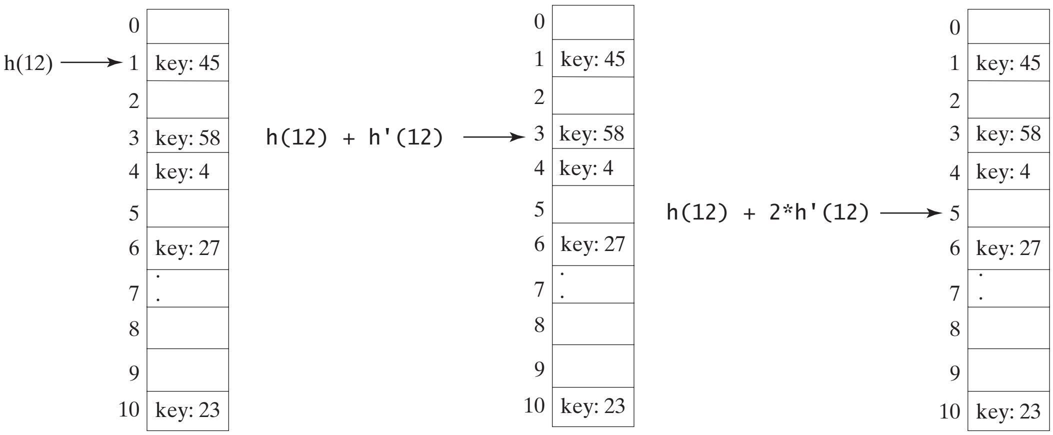
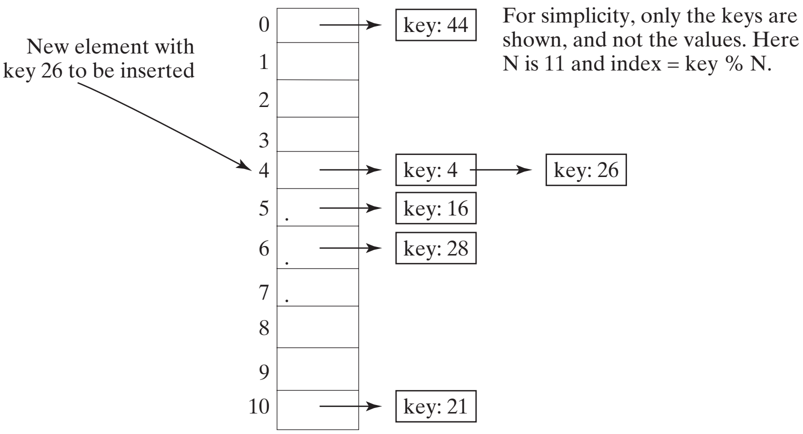


FIGURE 48.5 The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.

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 the probe sequence that reaches the entire table. Note that the second function should never have a zero value, since zero is not an increment.



New element with key 26 to be inserted

For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

FIGURE 48.6 Separate chaining chains the entries with the same hash index in a bucket.

48.6 Handling Collisions Using Separate Chaining

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.

bucket

implementing bucket

You may 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 48.6.

48.7 Load Factor and Rehashing

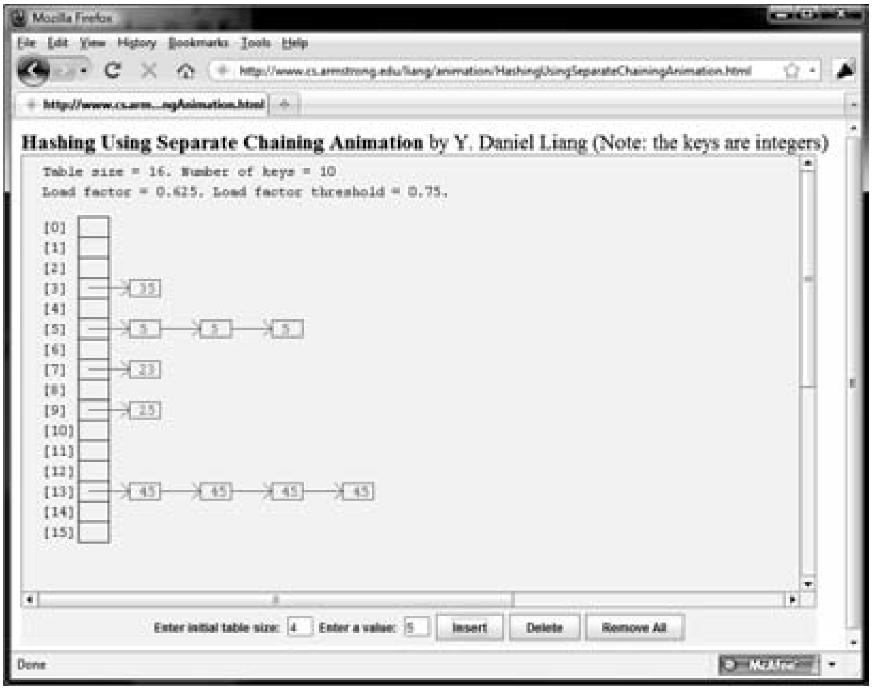
load factor

Load factor λ measures how full the hash table is. It is the ratio of the size of the map to the size of the hash table, i.e., $\lambda = \frac{n}{N}$, where n denotes the number of elements and N the number of locations in the hash table.

Note that λ is zero if the map 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 collision increases. Studies show that you should maintain the load factor under `0.5` for the open addressing scheme and under `0.9` for the separate chaining scheme.

threshold

Keeping the load factor under a certain threshold is important for the performance of hashing. In the implementation of `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



Hashing Using Separate Chaining Animation by Y. Daniel Liang (Note: the keys are integers)

```
Table size = 16. Number of keys = 10
Load factor = 0.625. Load factor threshold = 0.75.
```

Enter initial table size: 4 Enter a value: 5

FIGURE 48.7 The animation tool shows how separate chaining works.

size and *rehash* all the entries in the map to the new hash table. Notice that 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.

rehash

Pedagogical Note

Follow the link www.cs.armstrong.edu/liang/animation/HashingUsingSeparateChainingAnimation.html to see how to hashing with linear probing works, as shown in Figure 48.7.

separate chaining animation

48.8 Implementing a Map Using Hashing

Now you know 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`** with some minor variations. In the **`java.util.Map`** interface, the keys are distinct. However, a map may allow duplicate keys. Our map interface allows duplicate keys. We name the interface **`MyMap`** and a concrete class **`MyHashMap`**, as shown in Figure 48.8.

duplicate keys

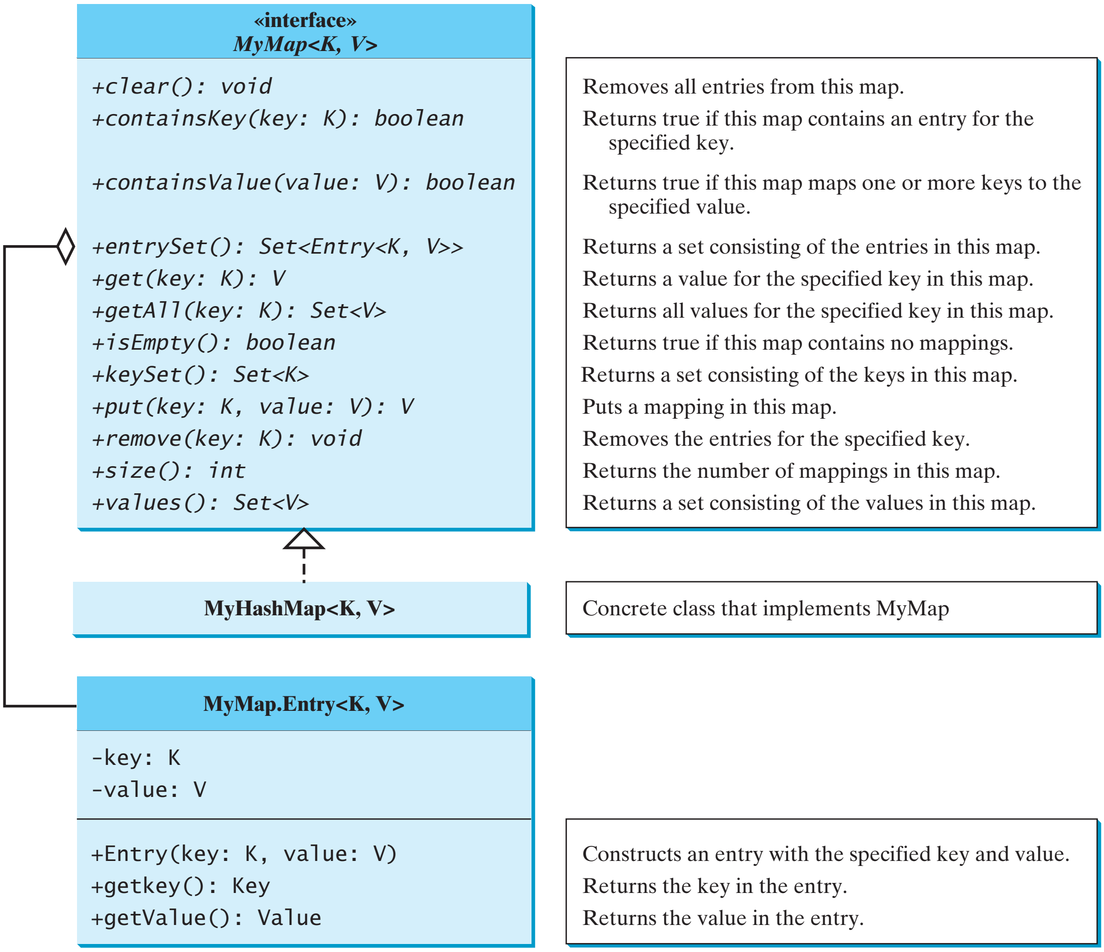


FIGURE 48.8 **`MyHashMap`** implements the **`MyMap`** interface.

The `get(key)` method gets one of the values that match the key. The `getAll(key)` method retrieves all values that match the key.

How do you implement `MyHashMap`? If you use an `ArrayList` and store a new entry at the end of the list, the search time will be $O(n)$. If you implement `MyHashMap` using an AVL tree, the search time will be $O(\log n)$. Nevertheless, you can implement `MyHashMap` using hashing to obtain an $O(1)$ time search algorithm. Listing 48.1 shows the `MyMap` interface and Listing 48.2 implements `MyHashMap` using separate chaining.

LISTING 48.1 MyMap.java

interface **MyMap**
clear
containsKey
containsValue
entrySet
get
getAll
isEmpty
keySet
put
remove
size
values
Entry inner class

```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);

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

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

  /** Return all values for the specified key in this map */
  public java.util.Set<V> getAll(K key);

  /** Return true if this map contains no 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 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;
    }

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

LISTING 48.2 MyHashMap.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;

  // 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];
  }
```

class **MyHashMap**

default initial capacity

maximum capacity

current capacity

default load factor

load-factor threshold

size

hash table

no-arg constructor

constructor

constructor

```java

  /** Remove all of the entries from this map */
```

clear

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

  /** Return true if the specified key is in the map */
```

containsKey

```java
  public boolean containsKey(K key) {
    if (get(key) != null)
      return true;
     else
      return false;
  }

  /** Return true if this map contains the specified value */
```

containsValue

```java
  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;
  }

  /** Return a set of entries in the map */
```

entrySet

```java
  public java.util.Set<MyMap.Entry<K,V>> entrySet() {
    java.util.Set<MyMap.Entry<K, V>> set =
      new java.util.HashSet<MyMap.Entry<K, V>>();

    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;
  }

  /** Return the first value that matches the specified key */
```

get

```java
  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;
  }

  /** Return all values for the specified key in this map */
```

getAll

```
  public java.util.Set<V> getAll(K key) {
    java.util.Set<V> set = new java.util.HashSet<V>();
    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))
          set.add(entry.getValue());
    }

    return set;
  }

```

isEmpty

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

```

keySet

```
  /** 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<K>();

    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

```
  /** Add an entry (key, value) into the map */
  public V put(K key, V value) {
    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 it is not created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<Entry<K, V>>();
    }

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

    size++; // Increase size

    return value;
  }

```

remove

```
  /** Remove the entries for the specified key */
  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
        }
    }
  }

  /** Return the number of mappings in this map */
```

size

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

  /** 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<V>();

    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;
    }

    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`

`toString`

```
226      }
227    }
228
229    /** Rehash the map */
230    private void rehash() {
231      java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
232      capacity <<= 1; // Double capacity
233      table = new LinkedList[capacity]; // Create a new hash table
234      size = 0; // Clear size
235
236      for (Entry<K, V> entry: set) {
237        put(entry.getKey(), entry.getValue()); // Store to new table
238      }
229    }
240
241    /** Return a string representation for this map */
242    public String toString() {
243      StringBuilder builder = new StringBuilder("[");
244
245      for (int i = 0; i < capacity; i++) {
246        if (table[i] != null && table[i].size() > 0)
247          for (Entry<K, V> entry: table[i])
248            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 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). You can construct a map with the specified capacity and a default load-factor threshold (lines 32–34). You can construct 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()` that deletes all entries in the buckets (lines 221–227). 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.

`containsKey`

The `containsKey(key)` method checks whether the specified key is in the map by by invoking the `get` method (lines 55–60). Since `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.

getAll The `getAll(key)` method returns the value of all entries with the specified key (lines 106–117). This method takes $O(1)$ time.

isEmpty The `isEmpty()` method simply returns true if the map is empty (lines 120–122). 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 add them to a set (lines 125–137). This method takes $O(capacity)$ time.

put The `put(key, value)` method adds a new entry into the map. The method first checks whether the size exceeds the load-factor threshold (line 141). If so, invoke `rehash()` (line 145) to increase the capacity and store entries into the new 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 clears the size (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 all entries 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 add it to a set (lines 185–197). This method takes $O(capacity)$ time.

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

Table 48.1 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 that the complexities of the `clear`, `entrySe`t, `keySet`, `values`, and `rehash`

TABLE 48.1 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)	$O(1)$
getAll(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()	$O(capacity)$

methods depend on `capacity`, so to avoid poor performance for these methods you should choose an initial capacity carefully.

Listing 48.3 gives a test program that uses `MyHashMap`.

LISTING 48.3 TestMyHashMap.java

```
 1 public class TestMyHashMap {
 2   public static void main(String[] args) {
 3     // Create a map
 4     MyMap<String, Integer> map = new MyHashMap<String, Integer>();
 5     map.put("Smith", 30);
 6     map.put("Anderson", 31);
 7     map.put("Lewis", 29);
 8     map.put("Cook", 29);
 9
10     System.out.println("Entries in map: " + map);
11
12     System.out.println("The age for " + "Lewis is " +
13       map.get("Lewis").intValue());
14
15     System.out.println("Is Smith in the map? " +
16       map.containsKey("Smith"));
17     System.out.println("Is age 33 in the map? " +
18         map.containsValue(33));
19
20     map.remove("Smith");
21     System.out.println("Entries in map: " + map);
22
23     map.clear();
24     System.out.println("Entries in map: " + map);
25   }
26 }
```

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, 30][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), adds entries to the map (lines 5–8), displays the entries (line 10), gets a value for a key (line 13), checks whether the map contains the key (line 16) and a value (line 18), removes an entry with the key "Smith" (line 20), and redisplays the entries in the map (line 22).

48.9 Set

set

A *set* 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 red-black trees.

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

MySet
MyHashSet

We design our custom `Set` interface to mirror `java.util.Set` with some minor variations. The `java.util.Set` interface extends `java.util.Collection`. Our set interface is the root interface. We name the interface `MySet` and a concrete class `MyHashSet`, as shown in Figure 48.9.

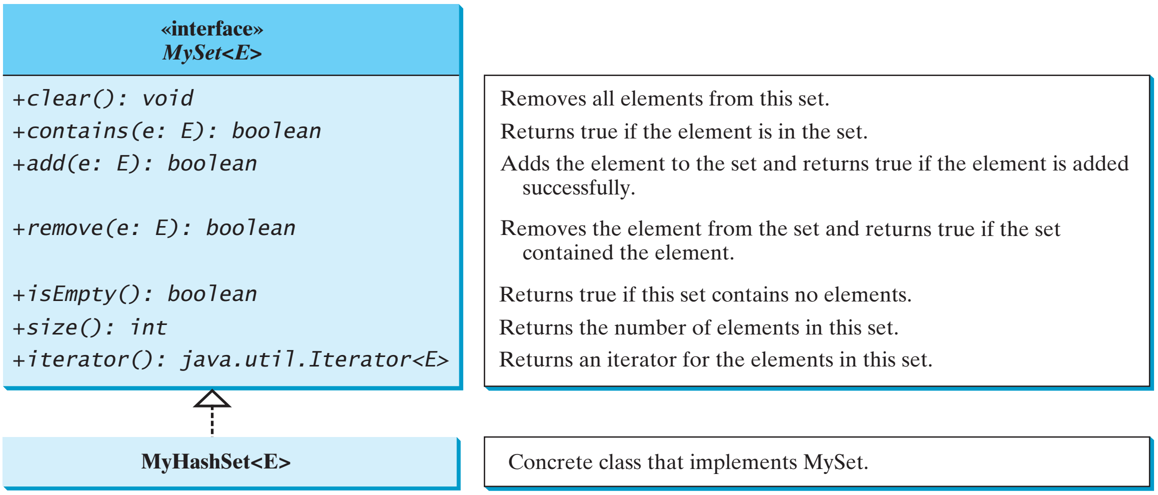


FIGURE 48.9 `MyHashSet` implements the `MySet` interface.

Listing 48.4 shows the `MySet` interface and Listing 48.5 implements `MyHashSet` using separate chaining.

LISTING 48.4 MySet.java

clear
contains
add
remove
isEmpty
size
iterator

```java
public interface MySet<E> {
  /** Remove all elements from this set */
  public void clear();

  /** Return true if the element is in the set */
  public boolean contains(E e);

  /** Add an element to the set */
  public boolean add(E e);

  /** Remove the element from the set */
  public boolean remove(E e);

  /** Return true if the set contains no elements */
  public boolean isEmpty();

  /** Return the number of elements in the set */
  public int size();

  /** Return an iterator for the elements in this set */
  public java.util.Iterator iterator();
}
```

LISTING 48.5 MyHashSet.java

class **MyHashSet**

```java
import java.util.LinkedList;

public class MyHashSet<E> implements MySet<E> {
```

```
  // Define the default hash-table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 16;

  // 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 entries in the set
  private int size = 0;

  // Hash table is an array with each cell that is 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) {
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

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

  /** Remove all elements from this set */
  public void clear() {
    size = 0;
    removeElements();
  }

  /** Return true if the element is in the set */
  public boolean contains(E e) {
    int bucketIndex = hash(e.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      for (E element: bucket)
        if (element.equals(e))
          return true;
    }

```

default initial capacity

maximum capacity

current capacity

default max load factor

load-factor threshold

size

hash table

no-arg constructor

constructor

constructor

`clear`

`contains`

```
    return false;
  }

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

add

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

    if (size > 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 it is not created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<E>();
    }

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

    size++; // Increase size

    return true;
   }

  /** 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 it is not created
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      for (E element: bucket)
        if (e.equals(element)) {
          bucket.remove(element);
          break;
        }
    }

    size--; // Decrease size

    return true;
  }

  /** Return true if the set contains no elements */
```

isEmpty

```
  public boolean isEmpty() {
    return size == 0;
  }

  /** Return the number of elements in the set */
```

size

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

iterator

```

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

```

inner class

```
  /** 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
    MyHashSet<E> set;

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

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

      return false;
    }

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

   /** Remove the current element and refresh the list */
    public void remove() {
      // Delete the current element from the hash set
      set.remove(list.get(current));
      list.remove(current); // Remove the current element from the list
    }
  }

```

hash

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

```

supplementalHash

```
  /** Ensure the hashing is evenly distributed */
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }

```

trimToPowerOf2

```
  /** Return a power of 2 for initialCapacity */
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1;
    }

    return capacity;
  }
```

```java

  /** 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

```java
  private void rehash() {
    java.util.ArrayList<E> list = setToList(); // Copy to a list
    capacity <<= 1; // Double capacity
    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

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

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

    return list;
  }

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

toString

```java
  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();
  }
}
```

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. The elements are all distinct in **MyHashSet**, but two entries may have the same keys in **MyHashMap**.

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). You can construct a set with the specified capacity and a default load factor (lines 32–34). You can construct a set with the specified capacity and load factor (lines 38–46).

clear

The **clear** method removes all entries from the map (lines 49–52). It invokes **removeElements()** that deletes all elements in the buckets (lines 187–193). This 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 (lines 55–65). This method takes *O*(1) time.

add

The **add(element)** method adds a new element into the set. The method first checks whether the size exceeds the load-factor threshold (line 72). If so, invoke **rehash()** (line 76) to increase the capacity and store entries into the new hash table.

rehash

The **rehash()** method first copies all elements in a list (line 197), doubles the capacity (line 198), obtains a new threshold (line 198), and creates a new hash table (line 199), and clears the size (line 200). The method then copies the entries into the new hash table (lines 202–203). 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 95–114). This method takes *O*(1) time.

size

The **size()** method simply returns the size of the set (lines 122–124). 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 141). The variable **current** points to the element in the list. Initially, **current** is **0** (line 135), 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 153). Invoking **remove()** removes the current element in the iterator from the set.

hash

The **hash()** method invokes the **supplementalHash** to ensure that the hashing is evenly distributed to produce an index for the hash table (lines 166–174). This method takes *O*(1) time.

Table 48.2 summarizes the time complexity of the methods in **MyHashSet**.

Listing 48.6 gives a test program that uses **MyHashSet**.

LISTING 48.6 TestMyHashSet.java

```java
public class TestMyHashSet {
  public static void main(String[] args) {
    // Create a MyHashSet
    MySet<String> set = new MyHashSet<String>();
    set.add("Smith");
    set.add("Anderson");
```

create a set
add elements

display elements
set size
remove element
display elements
clear set

```
 7      set.add("Lewis");
 8      set.add("Cook");
 9
10      System.out.println("Elements in set: " + set);
11      System.out.println("Number of elements in set: " + set.size());
12      System.out.println("Is Smith in set? " + set.contains("Smith"));
13
14      set.remove("Smith");
15      System.out.println("Elements in set: " + set);
16
17      set.clear();
18      System.out.println("Elements in set: " + set);
19    }
20  }
```

```
Elements in set: [Smith, Lewis, Anderson, Cook]
Number of elements in set: 4
Is Smith in set? true
Elements in set: [Lewis, Anderson, Cook]
Elements in set: []
```

TABLE 48.2 Time Complexities for Methods in `MyHashMap`

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*)

The program creates a set using `MyHashSet` (line 4), adds elements to the set (lines 5–8), displays the elements (line 10), gets the size (line 11), checks whether the set contains the element (line 12), removes an element (line 14), and clears the set (line 17).

KEY TERMS

associative array	48–2
clustering	48–5
dictionary	48–2
double hashing	48–4
hash code	48–3
hash function	48–2
hash map	48–2
hash set	48–18
hash table	48–2
linear probing	48–4
load factor	48–8
open addressing	48–4
perfect hash function	48–2
polynomial hash code	48–4
rehashing	48–8
secondary clustering	48–5
separate chaining	48–4

CHAPTER SUMMARY

1. A *map* is a data structure that stores entries. Each entry contains two parts: *key* and *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 search, 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 search, insertion, and deletion for a set.
3. *Hashing* is a technique that retrieves the value using the index obtained from 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 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.

REVIEW QUESTIONS

Sections 48.1–48.5

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

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

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

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

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

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

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

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

48.9 Describe the clustering problem for linear probing.

48.10 What is the secondary clustering?

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

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

48.13 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;
```

48.14 Suppose the size of the table is 10. What is the probe sequence for a key 12 using the following double hashing functions?

```
h(k) = k % 10;
h'(k) = 7 - k % 7;
```

Sections 48.6–48.8

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

48.16 In Listing 48.5, the `remove` method in the iterator removes the current element from the set. It also removes the current element from the internal list (line 165):

```
// Remove the current element from the list
list.remove(current);
```

Why is it necessary?

PROGRAMMING EXERCISES

48.1** (*Implementing `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 `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

48.2** (*Implementing `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 `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

48.3** (*Implementing `MyMap` using open addressing with double hashing*) Create a new concrete class that implements `MyMap` using open addressing with double 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 `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

48.4** (*Modifying `MyHashMap` with distinct keys*) Modify `MyHashMap` so that all entries in it have different keys.

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

48.6** (*Animating linear probing*) Write a Java applet that animates linear probing as shown in Figure 48.3. You can change the initial size of the hash-table in the applet. Assume the load-factor threshold is `0.75`.

48.7** (*Animating separate chaining*) Write a Java applet that animates `MyHashSet` as shown in Figure 48.7. You can change the initial size of the hash table. Assume the load-factor threshold is `0.75`.