

CHAPTER

11

Hashing

Objectives

- To know what hashing is for (§11.3).
- To obtain the hash code for an object and design the hash function to map a key to an index (§11.4).
- To handle collisions using open addressing (§11.5).
- To know the differences among linear probing, quadratic probing, and double hashing (§11.5).
- To handle collisions using separate chaining (§11.6).
- To understand the load factor and the need for rehashing (§11.7).
- To implement MyHashMap using hashing (§11.8).

11.1 Introduction

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.

11.2 Map

<Side Remark: map>

<Side Remark: key>

<Side Remark: value>

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.

NOTE:

<Side Remark: dictionary>

<Side Remark: hash table>

<Side Remark: associative array>

A map is also called a *dictionary*, a *hash table*, or an *associative array*. The new trend is to use the term *map*.

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` is implemented using `LinkedList`, and `java.util.TreeMap` is implemented using red-black trees. You will learn the concept of hashing and use it to implement a map in this chapter. In the chapter exercise, you will implement `LinkedHashMap` and `TreeMap`.

11.3 Hashing

<Side Remark: hash table>

<Side Remark: hash function>

<Side Remark: 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 11.1, 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 key without performing a search.

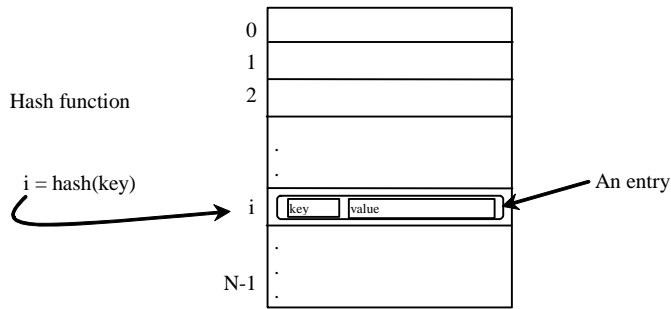


Figure 11.1

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

<Side Remark: perfect hash function>

<Side Remark: collision>

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 collisions, the best way is to avoid collisions in the first place. So, you should design a fast and easy-to-computer hash function that minimizes collisions.

11.4 Hash Functions and Hash Codes

<Side Remark: hash code>

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

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 `equal` 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.

11.4.1 Hash Codes for Primitive Types

<Side Remark: byte, short, int, char>

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.

<Side Remark: float>

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.

<Side Remark: long>

<Side Remark: folding>

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 operator to combine the two halves. This process is called folding. So, the hashing code is

```
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 bit-wise exclusive or operator. It operates on two corresponding bits of the binary operands. For example, 1010110 ^ 0110111 yields 1100001.

<Side Remark: double>

<Side Remark: folding>

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

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

11.4.2 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 same letters such as tod and dot.

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

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

<Side Remark: polynomial hash code>

where s_i is s.charAt(i). This expression is a polynomial in a for some positive b. So, this is called a polynomial hash code. By Horner's rule, it can be evaluated efficiently as follows:

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

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 is 31, 33, 37, 39, and 41. In the String class, the hashCode is overridden using the polynomial hash code with b being 31.

11.4.3 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 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);  
}
```

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 .

11.5 Handling Collisions Using Open Addressing

<Side Remark: open addressing>

<Side Remark: separate chaining>

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*.

<Side Remark: linear probing>

<Side Remark: quadratic probing>

<Side Remark: double hashing>

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*.

11.5.1 Linear Probing

<Side Remark: 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]`, 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 11.2.

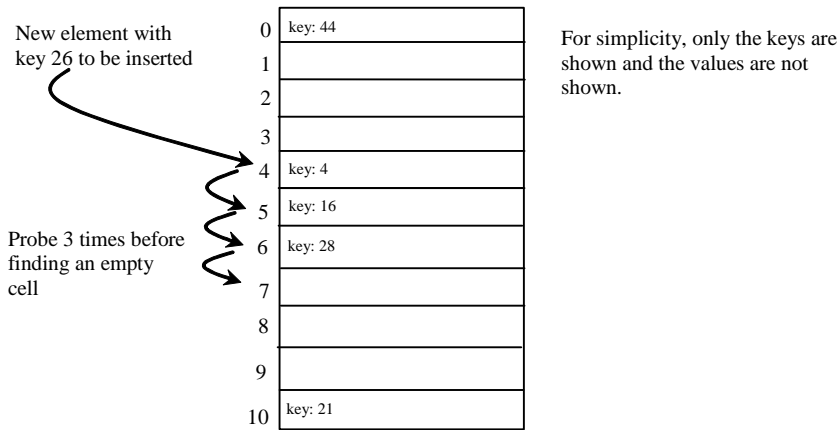


Figure 11.2

Linear probe finds the next available location sequentially.

NOTE:

<Side Remark: circular hash table>

When probing reaches the end of the table, it continues at the beginning of the table. Thus, the hash table is treated as if it were circular.

<Side Remark: 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 $\text{hashTable}[k \% n]$ contains the entry. If not, check whether $\text{hashTable}[k+1 \% n]$ contains the entry, and so on, until it is found, or an empty cell is reached.

<Side Remark: 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.

<Side Remark: 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.

11.5.2 Quadratic Probing

<Side Remark: secondary clustering>

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 11.3.

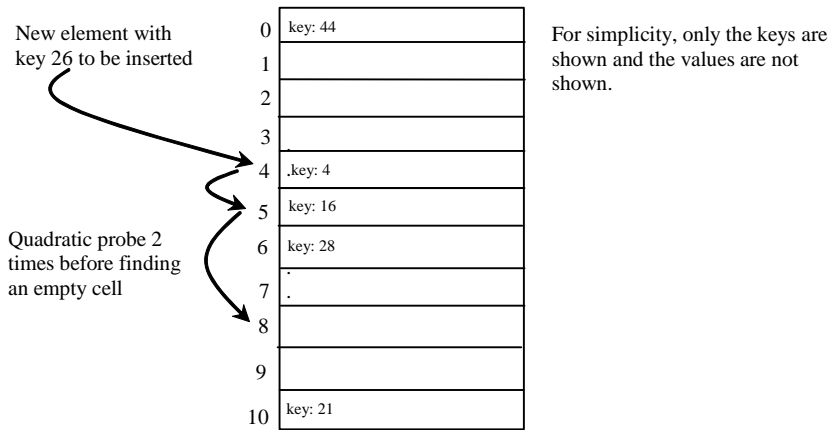


Figure 11.3

Quadratic probe increases the next index in the sequence by j^2 for $j = 1, 2, 3, \dots$

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 to be found for insertion as long as the table is not full. However, there is no such guarantee for quadratic probing.

11.5.3 Double Hashing

<Side Remark: 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 11.4.

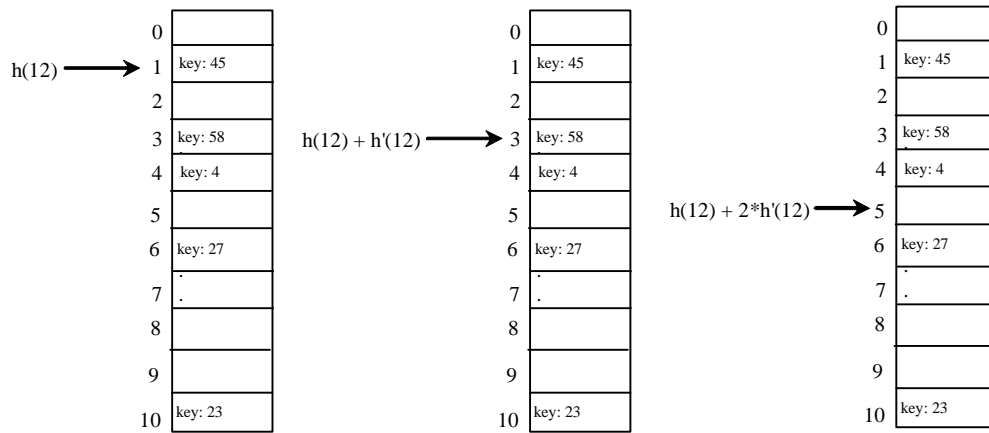


Figure 11.4

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.

11.6 Handling Collisions Using Separate Chaining

<Side Remark: 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.

<Side Remark: implementing bucket>

You may implement a bucket using an array, ArrayList, or LinkedList. However, LinkedList is most appropriate, because it requires least memory. For this reason, a bucket is usually implemented using LinkedList. 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 11.5.

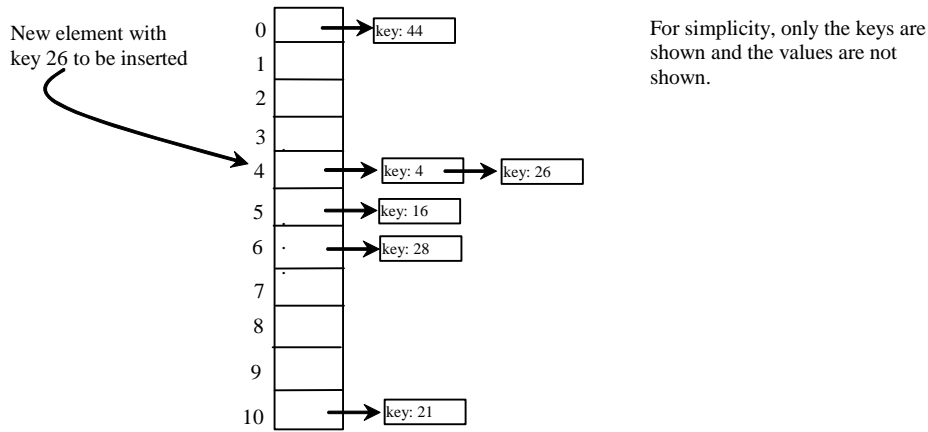


Figure 11.5

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

11.7 Load Factor and Rehashing

<Side Remark: load factor>

Load factor λ is a measure of 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 occupied cells and N denotes the number of locations in the hash table.

number of locations in the hash table.

Note that λ is zero if the map is empty. For open addressing schemes, λ is 1 if the hash table is full. For the open addressing schemes, λ is between 0 and 1. 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 schemes and maintain the load factor under 0.9 for the separate chaining scheme.

<Side Remark: threshold>

<Side Remark: rehash>

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 size and *rehashed* 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. As you see rehashing is costly, to reduce the likelihood of rehashing, you should at least double the hash table size. Even with periodic rehashing, hashing is an efficient implementation for map. See Exercise 11.a for comparing various implementations of map.

11.8 Implementing 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 the performance using the load factor, and how to increase the table size

and rehash to maintain the performance. This section demonstrates how to implement map using separate chaining.

<Side Remark: duplicate keys>

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 11.6.

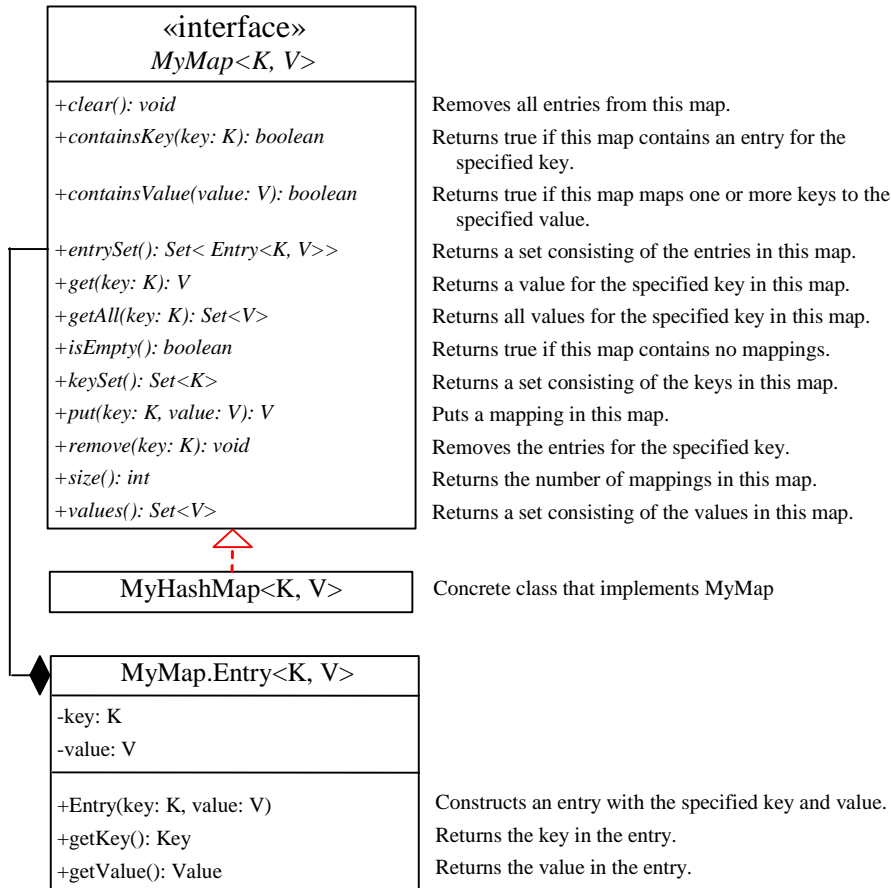


Figure 11.6

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 implement MyHashMap using an ArrayList and store a new entry at the end of the list, the search time would be $O(n)$. If you implement MyHashMap using an AVL tree, the search time would be $O(\log n)$. Nevertheless, you can implement it using hashing to obtain an $O(1)$ time search algorithm. Listing 11.1 shows the MyMap interface and Listing 11.2 implements MyHashMap using separate chaining.

Listing 11.1 MyMap.java

PD: Please add line numbers in the following code

<Side Remark line 1: interface MyMap>
<Side Remark line 3: clear>
<Side Remark line 6: containsKey>
<Side Remark line 9: containsValue>
<Side Remark line 12: entrySet>
<Side Remark line 15: get>
<Side Remark line 18: getAll>
<Side Remark line 21: isEmpty>
<Side Remark line 24: keySet>
<Side Remark line 27: put>
<Side Remark line 30: remove>
<Side Remark line 33: size>
<Side Remark line 36: values>
<Side Remark line 39: Entry inner class>

```
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 the entries 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 11.2 MyHashMap.java

*****PD: Please add line numbers in the following code*****

```

<Side Remark line 3: class MyHashMap>
<Side Remark line 5: default initial capacity>
<Side Remark line 8: maximum capacity>
<Side Remark line 11: current capacity>
<Side Remark line 14: default load factor>
<Side Remark line 17: load factor>
<Side Remark line 20: threshold>
<Side Remark line 23: size>
<Side Remark line 26: hash table>
<Side Remark line 29: no-arg constructor>
<Side Remark line 35: constructor>
<Side Remark line 41: constructor>
<Side Remark line 53: clear>
<Side Remark line 59: containsKey>
<Side Remark line 72: containsValue>
<Side Remark line 86: entrySet>
<Side Remark line 102: get>
<Side Remark line 115: getAll>
<Side Remark line 129: isEmpty>
<Side Remark line 134: keySet>
<Side Remark line 149: put>
<Side Remark line 177: remove>
<Side Remark line 192: size>
<Side Remark line 197: values>
<Side Remark line 212: hash>
<Side Remark line 217: supplementalHash>
<Side Remark line 223: trimToPowerOf2>
<Side Remark line 233: removeEntries>

```

<Side Remark line 242: [rehash](#)>
<Side Remark line 251: [toString](#)>

```
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 = 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_LOAD_FACTOR = 0.75f;

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

    // It is loadFactor * capacity, updated when capacity increases
    private int threshold = (int)(loadFactor * capacity);

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

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

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

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

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

        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
        table = new LinkedList[capacity];
    }

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

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

        return false;
    }
}
```

```

    /** Return true if this map contains the specified 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;
    }

    /** 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<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 */
    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 */
    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;
    }

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

    /** 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;
}

/** Add an entry (key, value) into the map */
public V put(K key, V value) {
    if (size > threshold) {
        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();
    }

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

    size++; // Increase size

    return value;
}

/** Remove the entries for the specified key */
public void remove(K key) {
    int bucketIndex = hash(key.hashCode());

    // Create a linked list for the bucket if it is not created
    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
}

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

/** Return a set consisting of the values in this map */
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 */
private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
}

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

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

```

```

private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
        capacity <<= 1;
    }

    return capacity;
}

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

/** Rehash the map */
private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Double capacity
    threshold = (int)(capacity * loadFactor); // Update threshold
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Clear size

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

/** Return a string representation for this map */
public String toString() {
    StringBuilder builder = new StringBuilder();

    for (int i = 0; i < capacity; i++) {
        if (table[i] != null)
            builder.append(table[i].toString());
    }

    return builder.toString();
}
}

```

<Side Remark: 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 16 (line 6) 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 is 0.75f (line 14). You can specify a custom load when constructing a map. The custom load factor is stored in `loadFactor` (line 17). Once the capacity and load factor are defined, the `threshold` is fixed (line 20). The data field `size` denotes the number of entries in the map (line 23). The hash table is an array. Each cell in the array is a linked list (line 26).

<Side Remark: three constructors>

Three constructors are provided to construct a map. You can construct a default map with the default capacity and load factor using the no-arg constructor (lines 29-31). You can construct a map with the specified capacity and a default load factor (lines

35-37). You can construct a map with the specified capacity and load factor (lines 41-50).

<Side Remark: clear>

The clear method removes all entries from the map (lines 53-56). It invokes removeEntries() that deletes all entries in the buckets (lines 233-239). This method takes $O(\text{capacity})$ time.

<Side Remark: containsKey>

The containsKey(key) method checks whether the specified key is in the map by examining whether the designated bucket contains the key (lines 63-65). This method takes $O(1)$ time.

<Side Remark: containsValue>

The containsValue(value) method checks whether the value is in the map (lines 72-83). This method takes $O(\text{capacity} + \text{size})$ time. It is actually $O(\text{capacity})$, since $\text{capacity} > \text{size}$.

<Side Remark: entrySet>

The entrySet() method returns a set that contains all entries in the map (lines 86-99). This method takes $O(\text{capacity})$ time.

<Side Remark: get>

The get(key) method returns the value of the first entry with the specified key (lines 106-109). This method takes $O(1)$ time.

<Side Remark: getAll>

The getAll(key) method returns the value of all entries with the specified key (lines 120-122). This method takes $O(1)$ time.

<Side Remark: isEmpty>

The isEmpty() method simply returns true if the map is empty (lines 129-131). This method takes $O(1)$ time.

<Side Remark: 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 137-142). This method takes $O(\text{capacity})$ time.

<Side Remark: 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 150). If so, invoke rehash() (line 154) to increase the capacity and store entries into the new hash table.

<Side Remark: rehash>

The rehash() method first copies all entries in a set (line 239), doubles the capacity (line 240), obtains a new threshold (line 241), creates a new hash table (line 242), and clears the size (line 243). The method then copies the entries into the new hash table (lines 245-247). The rehash method takes $O(\text{capacity})$ time. If no rehash is performed, the put method takes $O(1)$ time to add a new entry.

<Side Remark: remove>

The remove(key) method removes all entries with the specified key in the map (lines 173-185). This method takes $O(1)$ time.

<Side Remark: size>

The size() method simply returns the size of the map (lines 188-190). This method takes $O(1)$ time.

<Side Remark: 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 193-205). This method takes $O(\text{capacity})$ time.

<Side Remark: hash>

The hash() method invokes the supplementalHash to ensure that the hashing is evenly distributed to produce an index for the hash table (lines 208-216). This method takes $O(1)$ time.

Table 11.1 summarizes the time complexity of the methods in MyHashMap.

Table 11.1

Time Complexities for Methods in MyHashMap

Methods	Time
<code>clear()</code>	$O(\text{capacity})$
<code>containsKey(key: Key)</code>	$O(1)$
<code>containsValue(value: V)</code>	$O(\text{capacity})$
<code>entrySet()</code>	$O(\text{capacity})$
<code>get(key: K)</code>	$O(1)$
<code>getAll(key: K)</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>keySet()</code>	$O(\text{capacity})$
<code>put(key: K, value: V)</code>	$O(1)$
<code>remove(key: K)</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>values()</code>	$O(\text{capacity})$
<code>rehash()</code>	$O(\text{capacity})$

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

11.9 Set

<Side Remark: `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` is implemented using `LinkedList`, and `java.util.TreeSet` is implemented using red-black trees.

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

<Side Remark: `MySet`>

<Side Remark: `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 11.7.

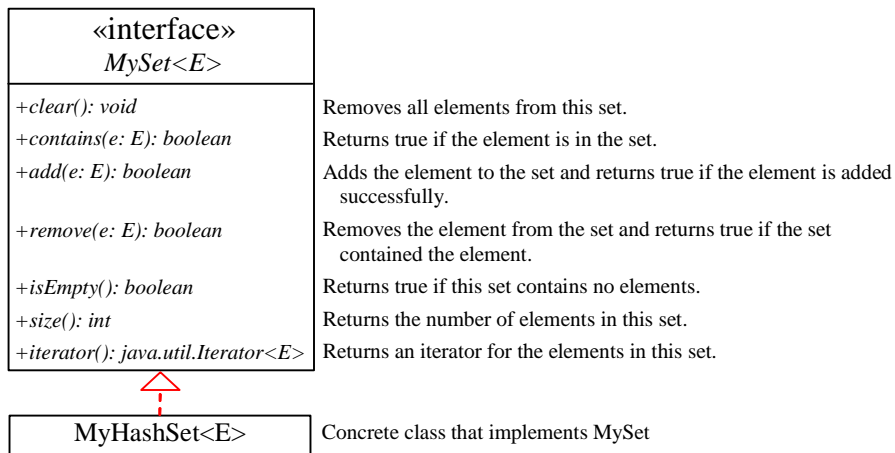


Figure 11.7

MyHashSet implements the *MySet* interface.

Listing 11.3 shows the *MySet* interface and Listing 11.4 implements *MyHashSet* using separate chaining.

Listing 11.3 MySet.java

*****PD: Please add line numbers in the following code*****

```

<Side Remark line 3: clear>
<Side Remark line 6: contains>
<Side Remark line 9: add>
<Side Remark line 12: remove>
<Side Remark line 15: isEmpty>
<Side Remark line 18: size>
<Side Remark line 21: iterator>

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 elements 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 11.4 MyHashSet.java

PD: Please add line numbers in the following code

<Side Remark line 3: class MyHashSet>
 <Side Remark line 5: default initial capacity>
 <Side Remark line 8: maximum capacity>
 <Side Remark line 11: current capacity>
 <Side Remark line 14: default load factor>
 <Side Remark line 17: load factor>
 <Side Remark line 20: threshold>
 <Side Remark line 23: size>
 <Side Remark line 26: hash table>
 <Side Remark line 29: no-arg constructor>
 <Side Remark line 35: constructor>
 <Side Remark line 41: constructor>
 <Side Remark line 53: clear>
 <Side Remark line 59: contains>
 <Side Remark line 72: add>
 <Side Remark line 80: rehash>
 <Side Remark line 99: remove>
 <Side Remark line 121: isEmpty>
 <Side Remark line 126: size>
 <Side Remark line 131: iterator>
 <Side Remark line 136: inner class>
 <Side Remark line 170: hash>
 <Side Remark line 175: supplementalHash>
 <Side Remark line 181: trimToPowerOf2>
 <Side Remark line 191: rehash>
 <Side Remark line 213: setToList>
 <Side Remark line 228: toString>

```
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_LOAD_FACTOR = 0.75f;

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

    // It is loadFactor * capacity, updated when capacity increases
    private int threshold = (int)(loadFactor * capacity);

    // 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 LOAD FACTOR);
}

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

/** Construct a set with the specified initial capacity
 * and load factor */
public MyHashSet(int initialCapacity, float loadFactor) {
    if (initialCapacity > MAXIMUM CAPACITY)
        this.capacity = MAXIMUM CAPACITY;
    else
        this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactor = loadFactor;
    threshold = (int)(capacity * loadFactor);
    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;
    }

    return false;
}

/** Add an element to the set */
public boolean add(E e) {
    if (contains(e))
        return false;

    if (size > threshold) {
        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();
    }

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

    size++; // Increase size

    return true;
}

/** Remove the element from the set */
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 */
public boolean isEmpty() {
    return size == 0;
}

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

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

/** Inner class for iterator */
private class MyHashSetIterator implements java.util.Iterator {

```

```

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

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

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

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

    table = new LinkedList[capacity]; // Create a new hash table

    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 */
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 */
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
    builder.append(list.get(list.size() - 1) + "]);

    return builder.toString();
}
}

```

<Side Remark: [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 and 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.

<Side Remark: 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 29-31). You can construct a set with the specified capacity and a default load factor (lines 35-37). You can construct a set with the specified capacity and load factor (lines 41-50).

<Side Remark: clear>

The clear method removes all entries from the map (lines 53-56). It invokes removeElements() that deletes all elements in the buckets (lines 191-197). This method takes $O(\text{capacity})$ time.

<Side Remark: contains>

The contains(element) method checks whether the specified element is in the set by examining whether the designated bucket contains the element (lines 63-65). This method takes $O(1)$ time.

<Side Remark: 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 76). If so, invoke rehash() (line 80) to increase the capacity and store entries into the new hash table.

<Side Remark: rehash>

The rehash() method first copies all elements in a list (line 201), doubles the capacity (line 202), obtains a new threshold (line 203), creates a new hash table (line 204), and clears the size (line 205). The method then copies the entries into the new hash table (lines 207-209). The rehash method takes $O(\text{capacity})$ time. If no rehash is performed, the add method takes $O(1)$ time to add a new element.

<Side Remark: remove>

The remove(element) method removes the specified element in the set (lines 99-118). This method takes $O(1)$ time.

<Side Remark: size>

The `size()` method simply returns the size of the set (lines 126-128). This method takes $O(1)$ time.

<Side Remark: 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 145). The variable `current` points to the element in the list. Initially, `current` is `0` (line 139), 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 158). Invoking `remove()` removes the current element in the iterator from the set.

<Side Remark: hash>

The `hash()` method invokes the `supplementalHash` to ensure that the hashing is evenly distributed to produce an index for the hash table (lines 208-216). This method takes $O(1)$ time.

Table 11.2 summarizes the time complexity of the methods in `MyHashSet`.

Table 11.2

Time Complexities for Methods in `MyHashMap`

Mehtods	Time
<code>clear()</code>	$O(\text{capacity})$
<code>contains(e: E)</code>	$O(1)$
<code>add(e: E)</code>	$O(1)$
<code>remove(e: E)</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>iterator()</code>	$O(\text{capacity})$
<code>rehash()</code>	$O(\text{capacity})$

Key Terms

*****PD: Please place terms in two columns same as in intro6e.**

- associative array
- clustering
- dictionary
- double probing

hash code
hash function
hash map
hash set
hash table
linear probing
load factor
open addressing
perfect hash function
polynomial hash code
rehashing
secondary clustering
separate chaining

Chapter Summary

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.

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.

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*, and then compresses the hash code into an index to the hash table.

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*.

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 11.1-11.3

11.1

What is a hash function? What is a perfection hash function? What is a collision?

11.2

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

11.3

How is the hash code for a Float object computed?

11.4

How is the hash code for a Long object computed?

11.5

How is the hash code for a Double object computed?

11.6

How is the hash code for a String object computed?

11.7

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

11.8

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

11.9

Describe the clustering problem for linear probing.

11.10

What is the secondary clustering?

11.11

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

11.12

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

11.13

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:

$$\begin{aligned}h(k) &= k \% 11; \\h'(k) &= 7 - k \% 7;\end{aligned}$$

11.14

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

$$\begin{aligned}h(k) &= k \% 10; \\h'(k) &= 7 - k \% 7;\end{aligned}$$

11.15

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

11.16

In Listing 11.4, 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):

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

Is this necessary?

Programming Exercises

11.1**

(Implementing MyHashMap using open addressing with linear probing)
Modify MyHashMap using open addressing with linear probing.

11.2**

(Implementing MyHashMap using open addressing with quadratic probing)
Modify MyHashMap using open addressing with quadratic probing.

11.3**

(Implementing MyHashMap using open addressing with double hashing)
Modify MyHashMap using open addressing with double hashing.

11.4**

(Modifying MyHashMap with distinct keys) Modify MyHashMap so that all entries in MyHashMa have different keys.

11.5**

(Implementing MyHashSet using java.util.HashMap) Implement MyHashSet using java.util.HashMap. Note that you can create entries with (key, key), rather than (key, value).

11.6**

(Animating linear probing) Write a Java applet that animates linear probing as shown in Figure 11.8. You can change the size of the hash table in the applet.

Figure 11.8

The applet illustrate how linear probing works.

11.7**

(Animating separate chaining) Write a Java applet that animates MyHashSet as shown in Figure 11.9.

Figure 11.9

The applet illustrate how separate chaining works.