CHAPTER 27

Hashing

Objectives

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

27.1 Introduction

<key point>

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

<end key point>

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

<margin note>why hashing?

27.2 What Is Hashing?

<key point>

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

<end key point>

Before introducing hashing, let us review map, which is a data structure that is implemented using hashing. Recall that a *map* (introduced in Section 22.6) is a container object that stores entries. Each entry contains two parts: a *key* and a *value*. The key, also called a *search key*, is used to search for the corresponding value. For example, a dictionary can be stored in a map, in which the words are the keys and the definitions of the words are the values.

<margin note>map

<margin note>key

<margin note>value

<box>

NOTE

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

<end box>

<key term>dictionary

<key term>hash table

<*key term*>associative array

The STL defines the `map` and `multimap` classes for modeling maps. You will learn the concept of hashing and use it to implement a map in this chapter.

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

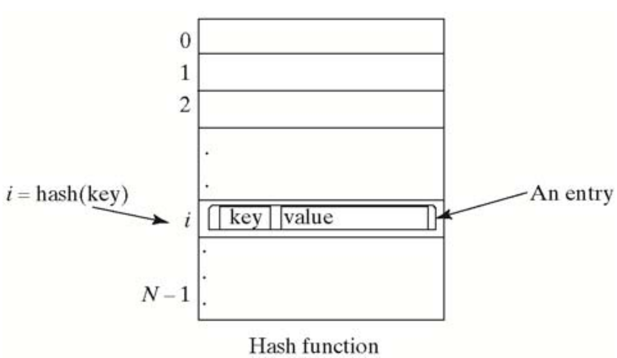
<*margin note*>hash table

<*key term*>hash function

<*margin note*>hashing

Figure 27.1

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



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. Although there are ways to deal with collisions, which are discussed later in this chapter, it is better to avoid collisions in the first place. Thus, you should design a fast and easy-to-compute hash function that minimizes collisions.

<key term>perfect hash function

<margin note>collision

<check point>

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

<end check point>

27.3 Hash Functions and Hash Codes

<key point>

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

<end key point>

C++ 11 defines a member function **`hash_code()`** in the **`type_info`** class, which returns an integer hash code for an element of a primitive type or an object type depending on the type and value of the element. The contract for the **`hash_code()`** function is as follows:

<key term>hash code

<margin note>hash_code()

1. The function returns the same hash code if the two elements are equal.
2. During the execution of a program, invoking the **`hash_code`** funciton multiple times returns the same integer, provided that the object's data are not changed.
3. Two unequal objects may have the same hash code, but the function in C++11 is to avoid too many such cases.

The following sections discuss the implementation of the hash code for numbers and strings.

27.3.1 Hash Codes for Primitive Types

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

<margin note>short, int, char

For a search key of the type **`float`**, use **`floatToIntBits(key)`** as the hash code, where **`floatToIntBits(float f)`** returns an **`int`** value whose bit representation is the same as the bit representation for the floating number `f`. Thus, two different search keys of the `float` type will have different hash codes. See Programming Exercise 27.6 for the implementation of `floatToIntBits`.

<*margin note*>float

For a search key of the type `long long` (`long long` is 64 bits in C++11), simply casting it to `int` would not be a good choice, because all keys that differ in only the first 32 bits will have the same hash code. To take the first 32 bits into consideration, divide the 64 bits into two halves and perform the exclusive-or operation to combine the two halves. This process is called *folding*. The hash code for a `long` key is

<*margin note*>long

<*margin note*>folding

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

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

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

<*margin note*>double

<*margin note*>folding

```
long bits = doubleToLongLongBits(key);
int hashCode = static_cast<int>(bits ^ (bits >> 32));
```

See Programming Exercise 27.7 for the implementation of `doubleToLongLongBits`.

27.3.2 Hash Codes for Strings

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

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

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

where s_i is s[i]. This expression is a polynomial for some positive *b*, so this is called a *polynomial hash code*. Using Horner's rule for polynomial evaluation (see Section 6.14), the hash code can be calculated efficiently as follows:

<***key term***>polynomial hash code

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

This computation can cause an overflow for long strings, but arithmetic overflow is ignored in C++. You should choose an appropriate value *b* to minimize collisions. Experiments show that good choices for *b* are 31, 33, 37, 39, and 41.

27.3.3 Compressing Hash Codes

The hash code for a key can be a large integer that is out of the range for the hash-table index, so you need to scale it down to fit in the index's range. Assume the index for a hash table is between **0** and **N-1**. The most common way to scale an integer to 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. We will choose **N** to be a value of the power of **2**. There is a good reason for this choice. When **N** is a value of the power of **2**,

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

is the same as

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

The ampersand, **&**, is a bitwise AND operator (see Appendix G, Bitwise Operations). The AND of two corresponding bits yields a **1** if both bits are **1**. For example, assume **N = 4** and **hashCode = 11**, **11 % 4 = 3**, which is the same as **01011 & 00011 = 11**. The **&** operator can be performed much faster than the **%** operator.

To ensure that the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function. The supplemental function is defined as:

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

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

The complete hash function is defined as:

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

This is the same as

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

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

<check point>

27.2 What is a hash code? What is the hash code for `short`, `int`, and `char`?

27.3 How is the hash code for a `float` value computed?

27.4 How is the hash code for a `long` value of 64 bits computed?

27.5 How is the hash code for a `double` value computed?

27.6 How is the hash code for a `string` object computed?

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

27.8 If `N` is a value of the power of `2`, is `N / 2` same as `N >> 1`?

27.9 If `N` is a value of the power of `2`, is `m % N` same as `m & (N – 1)` for any integer `m`?

<end check point>

27.4 Handling Collisions Using Open Addressing

<key point>

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.

<end key point>

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

<key term>open addressing

27.4.1 Linear Probing

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

<margin note>add entry

<key term>linear probing

<box>

NOTE

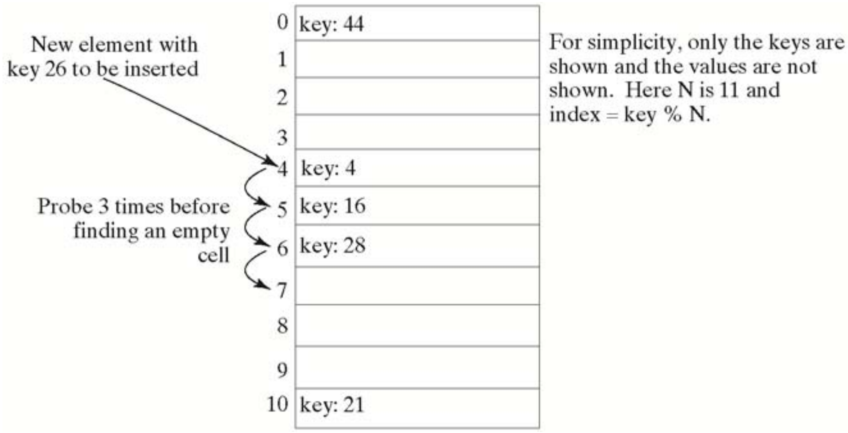
When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

<margin note>circular hash table

<end box>

Figure 27.2

Linear probing finds the next available location sequentially.



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.

<margin note>search entry

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

<margin note>remove entry

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.

<key term>cluster

<box>

PEDAGOGICAL NOTE

For an interactive GUI demo to see how linear probing works, go to www.cs.armstrong.edu/liang/animation/HashingLinearProbingAnimation.html, as shown in Figure 27.3.

<animation icon>linear probing animation on Companion Website

<end box>

27.4.2 Quadratic Probing

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

<*key term*>quadratic probing

Figure 27.3

The animation tool shows how linear probing works.

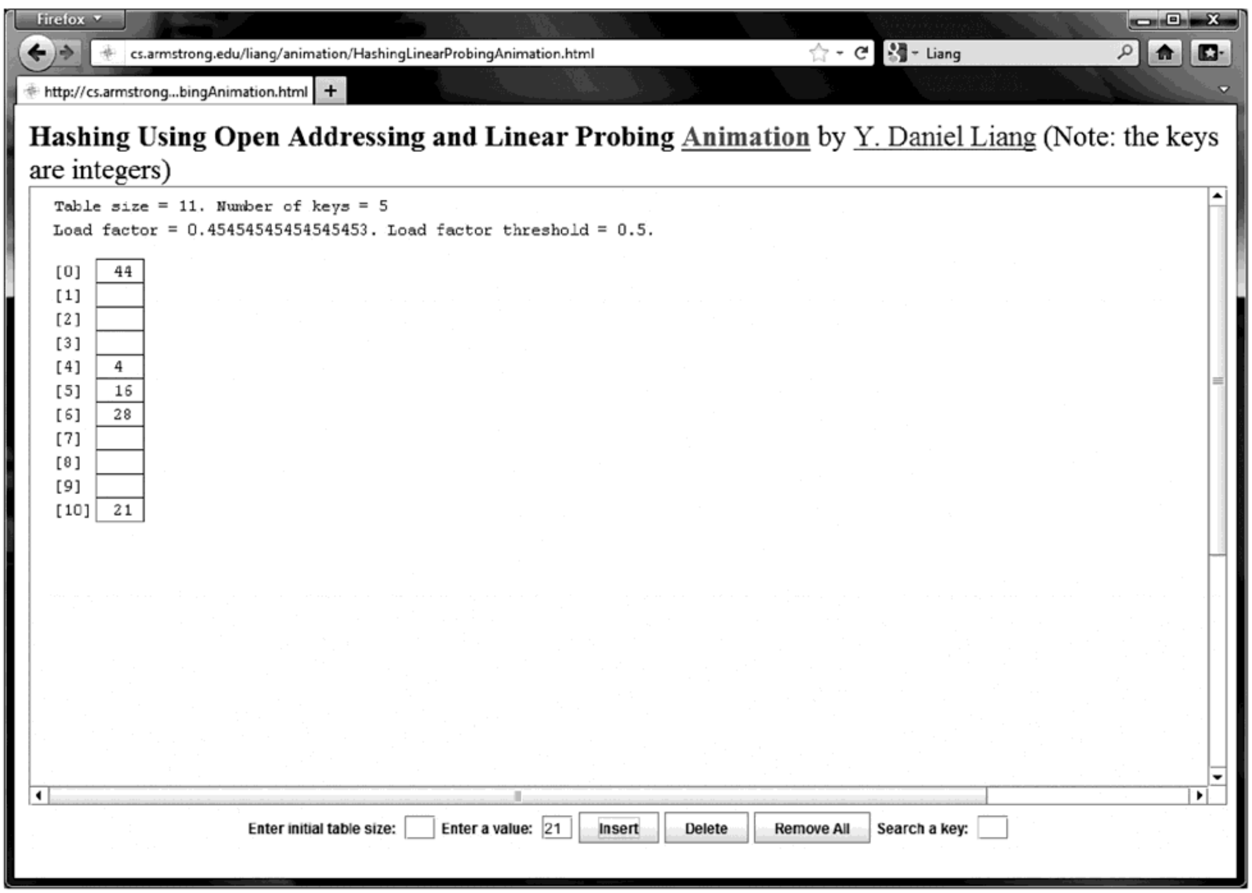
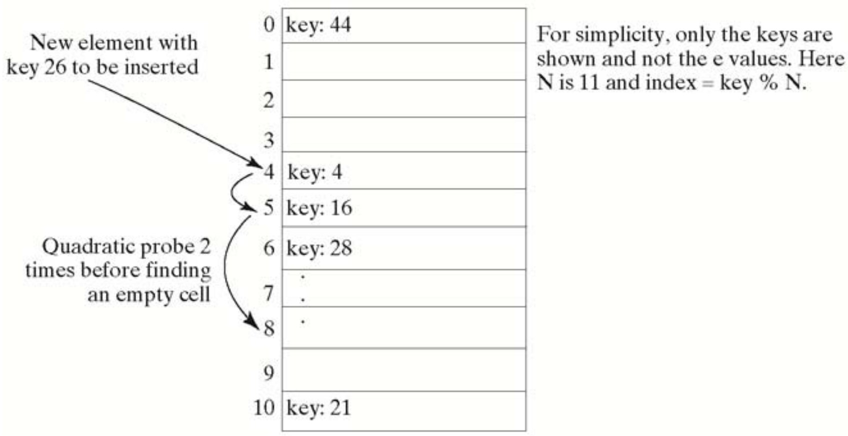


Figure 27.4

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



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

<key term>secondary clustering

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

<box>

PEDAGOGICAL NOTE

For an interactive GUI demo to see how quadratic probing works, go to www.cs.armstrong.edu/liang/animation/HashingQuadraticProbingAnimation.html, as shown in Figure 27.5.

<animation icon>quadratic probing animation on Companion Website

<end box>

27.4.3 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 *h'(key)* on the keys to determine the increments to avoid the clustering problem. Specifically, double hashing looks at the cells at indices (*k* + *j* * *h'(key)) % N*, for $j \geq 0$, that is, *k % N*, (*k* + *h'(key)) % N*, (*k* + *2* * *h'(key)) % N*, (*k* + *3* * *h'(key)) % N*, ..., and so on.

<*key term*>double hashing

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

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

For a search key of `12`, we have

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

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

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

Figure 27.5

The animation tool shows how quadratic probing works.

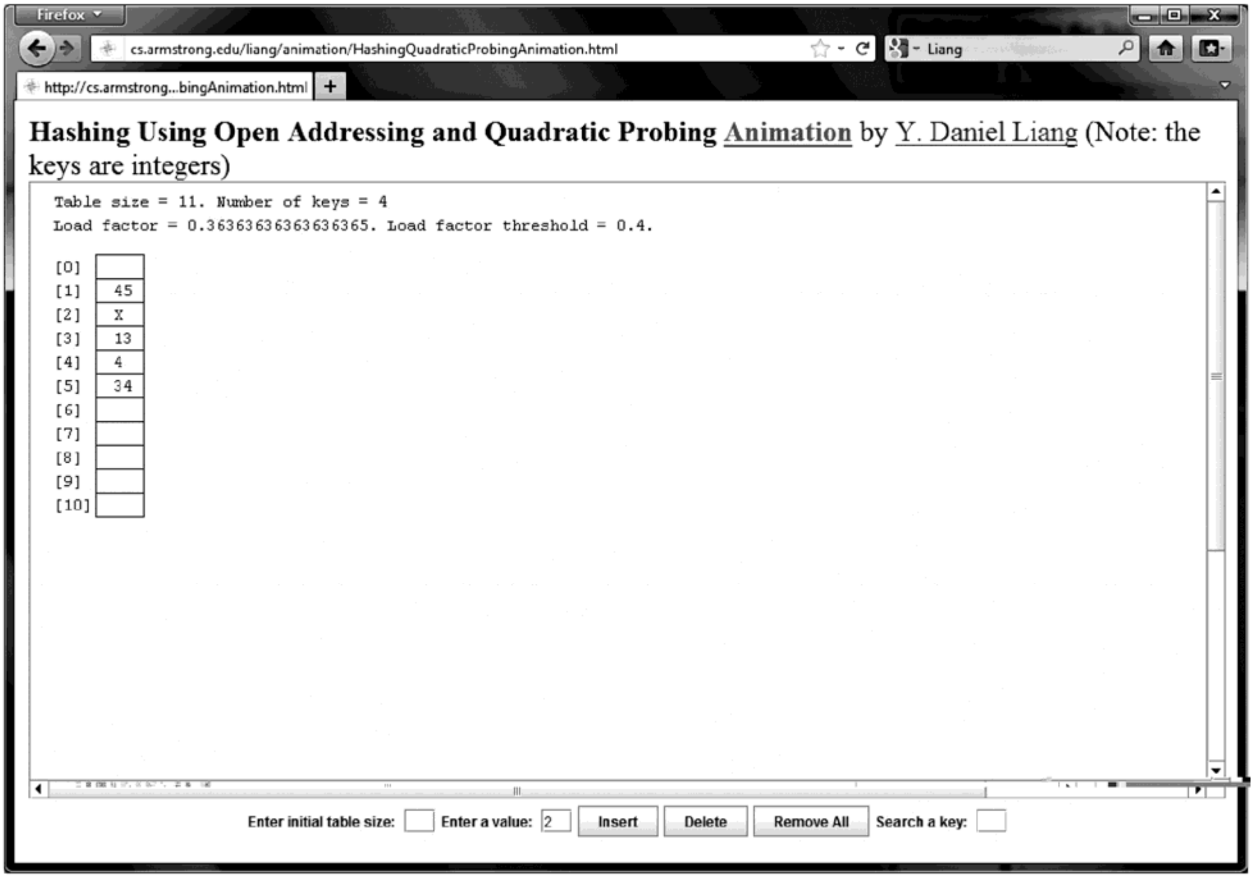
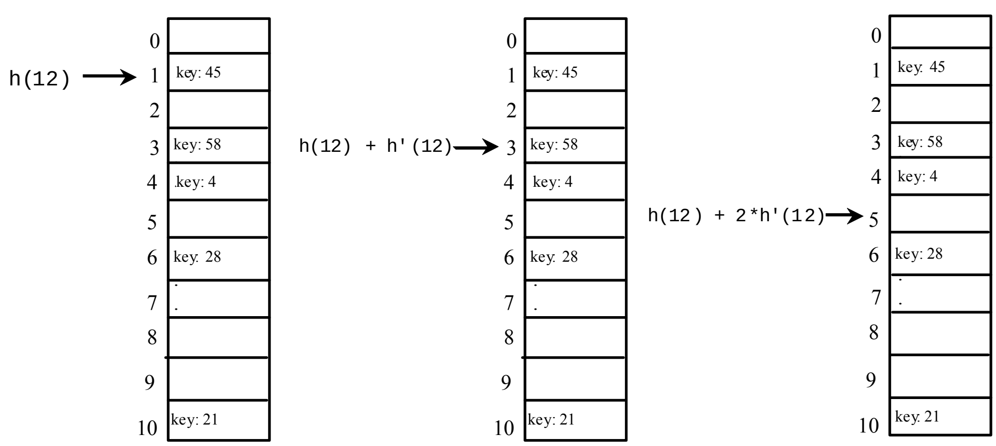


Figure 27.6

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



<check point>

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

27.11 Describe the clustering problem for linear probing.

27.12 What is secondary clustering?

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

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

27.15 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;
```

<end check point>

27.5 Handling Collisions Using Separate Chaining

<key point>

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

<end key point>

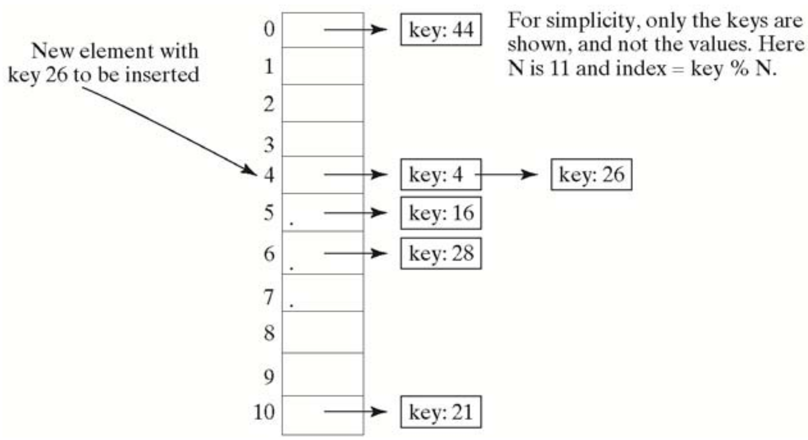
*<key term>*separate chaining

*<margin note>*implementing bucket

You can implement a bucket using an array, `ArrayList`, or `LinkedList`. We will use `LinkedList` for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head, as shown in Figure 27.7.

Figure 27.7

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



<check point>

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

<end check point>

27.6 Load Factor and Rehashing

<key point>

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

<end key point>

*<key term>*rehashing

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

*<key term>*load factor

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 a 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.

Keeping the load factor under a certain threshold is important for the performance of hashing. Whenever the load factor exceeds the threshold, you need to increase the hash-table size and *rehash* all the entries in the map into the new larger 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.

<margin note>threshold

<margin note>rehash

<box>

PEDAGOGICAL NOTE

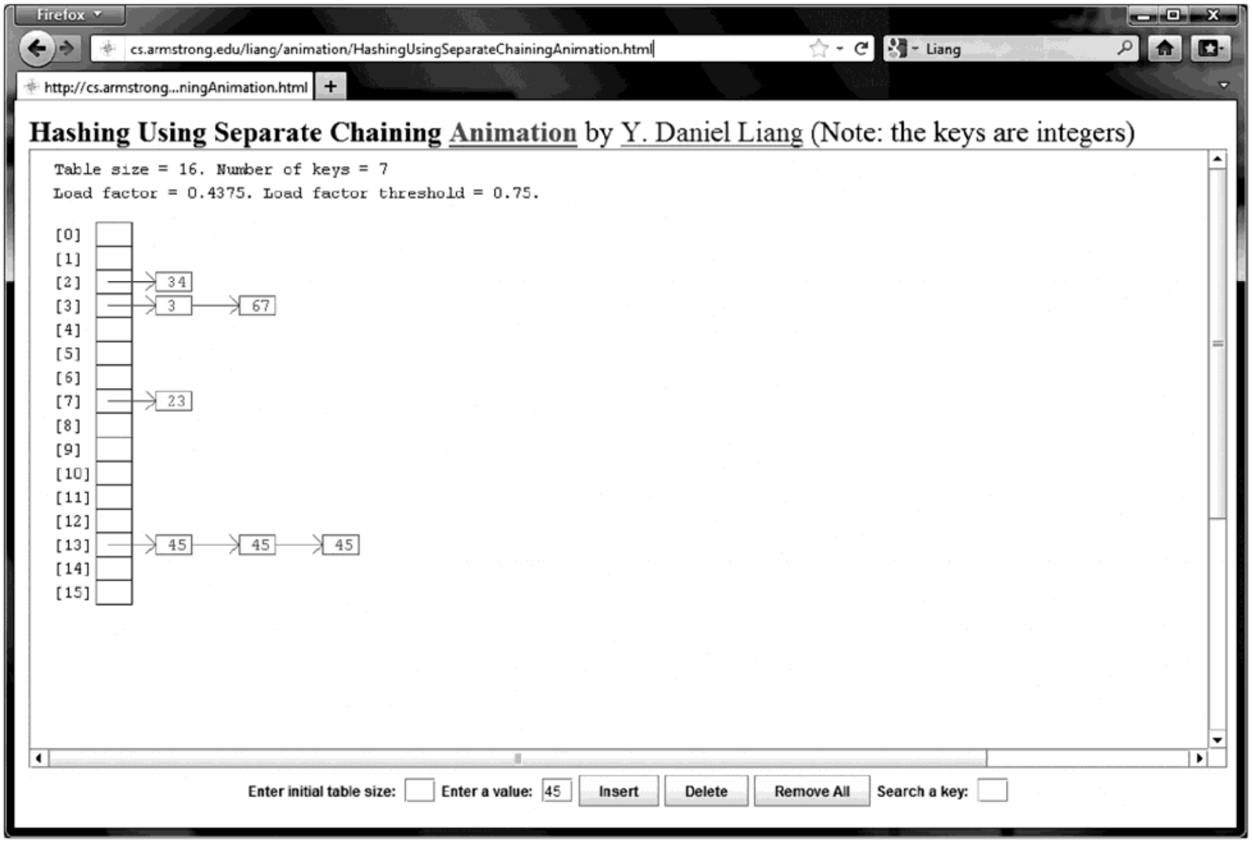
For an interactive GUI demo to see how separate chaining works, go to www.cs.armstrong.edu/liang/animation/HashingUsingSeparateChainingAnimation.html, as shown in Figure 27.8.

<animation icon>separate chaining animation on Companion Website

<end box>

Figure 27.8

The animation tool shows how separate chaining works.



<check point>

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

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

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

<end check point>

27.7 Implementing a Map Using Hashing

<key point>

A map can be implemented using hashing.

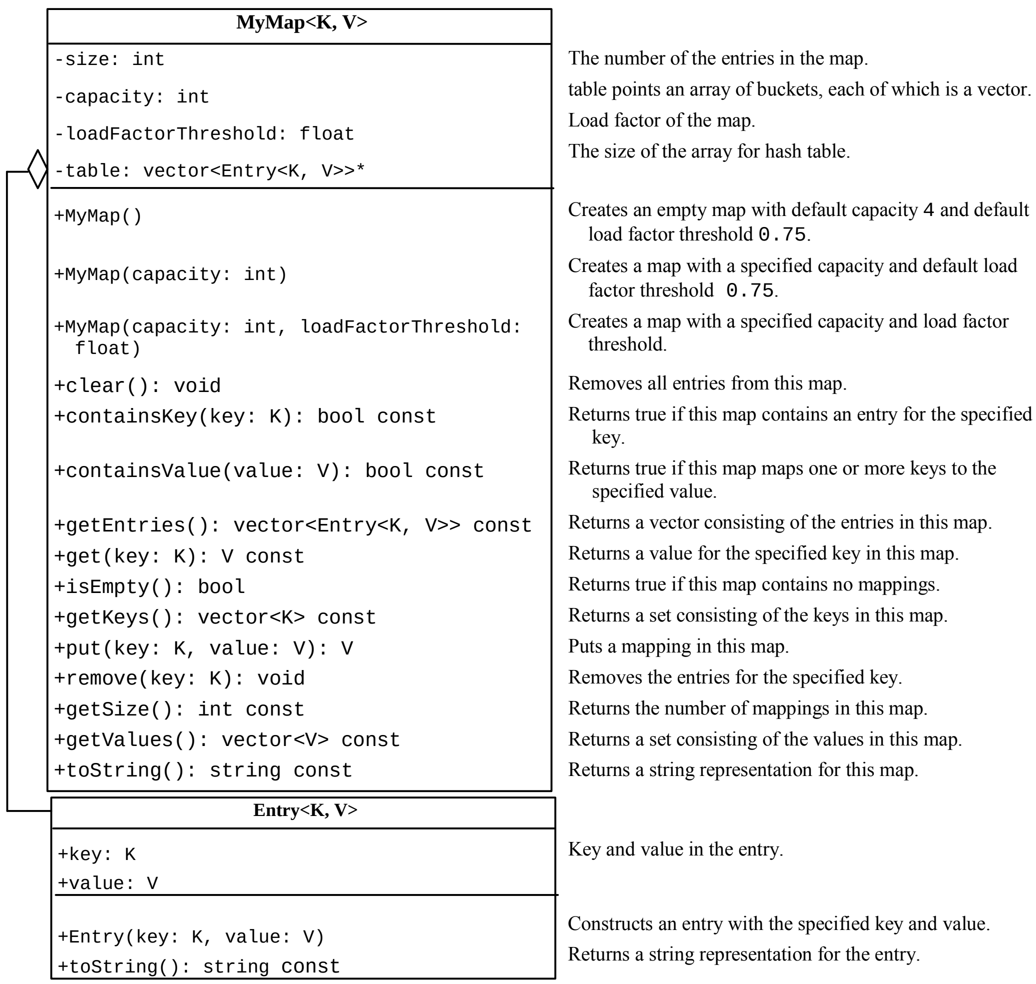
<end key point>

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

We design our custom `map` class and name it `MyMap`, as shown in Figure 27.9.

Figure 27.9

`MyMap` implements map using hashing.



How do you implement `MyMap`? 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 `MyMap` using a binary tree, the search time will be $O(\log n)$ if the tree is well balanced. Nevertheless, you can implement `MyMap` using hashing to obtain an $O(1)$ time search algorithm. Listing 27.1 implements `MyMap` using separate chaining.

Listing 27.1 MyMap.h

```
#ifndef MYMAP_H
```

```
#define MYMAP_H

#include <vector>
#include <string>
#include <sstream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int DEFAULT_INITIAL_CAPACITY = 4;
float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
unsigned MAXIMUM_CAPACITY = 1 << 30;

template<typename K, typename V>
class Entry // Represent an entry with key and value
{
public:
  Entry(K key, V value)
  {
    this->key = key;
    this->value = value;
  }

  string toString() const
  {
    stringstream ss;
    ss << "[" << key << ", " << value << "]";
    return ss.str();
  }

  K key;
  V value;
};

template<typename K, typename V>
class MyMap
{
public:
  MyMap();
  MyMap(int initialCapacity);
  MyMap(int initialCapacity, float loadFactorThreshold);

  V put(K key, V value);
  V get(K key) const;
  int getSize() const;
  bool isEmpty() const;
  vector<Entry<K,V>> getEntries() const;
  vector<K> getKeys() const;
  vector<V> getValues() const;
  string toString() const;
  bool containsKey(K key) const;
  bool containsValue(V value) const;
  void remove(K key);
  void clear();

private:
  int size;
  float loadFactorThreshold;
  int capacity;
```

```cpp

  // Hash table is an array with each cell as a vector
  vector<Entry<K, V>>* table;

  int hash(int hashCode) const;
  unsigned hashCode(K key) const;
  int supplementalHash(int h) const;
  int trimToPowerOf2(int initialCapacity);
  void rehash();
  void removeEntries();
};

template<typename K, typename V>
MyMap<K, V>::MyMap()
{
  capacity = DEFAULT_INITIAL_CAPACITY;
  table = new vector<Entry<K, V>>[capacity];
  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR;
  size = 0;
}

template<typename K, typename V>
MyMap<K, V>::MyMap(int initialCapacity)
{
  capacity = initialCapacity;
  table = new vector<Entry<K, V>>[capacity];
  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR;
  size = 0;
}

template<typename K, typename V>
MyMap<K, V>::MyMap(int initialCapacity, float loadFactorThreshold)
{
  if (initialCapacity > MAXIMUM_CAPACITY)
    capacity = MAXIMUM_CAPACITY;
  else
    capacity = trimToPowerOf2(initialCapacity);

  this->loadFactorThreshold = loadFactorThreshold;
  table = new vector<Entry<K, V>>[capacity];
  size = 0;
}

template<typename K, typename V>
V MyMap<K, V>::put(K key, V value)
{
  if (get(key) != NULL)
  { // The key is already in the map
    int bucketIndex = hash(hashCode(key));
    for (Entry<K, V>& entry: table[bucketIndex])
    {
      if (entry.key == key)
      {
        V oldValue = entry.value;
        // Replace old value with new value
        entry.value = value;
        // Return the old value for the key
        return oldValue;
      }
    }
```

```
  }

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

    rehash();
  }

  int bucketIndex = hash(hashCode(key));

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

  size++; // Increase size

  return value;
}

template<typename K, typename V>
V MyMap<K, V>::get(K key) const
{
  int bucketIndex = hash(hashCode(key));

  for (Entry<K, V>& entry: table[bucketIndex])
    if (entry.key == key)
      return entry.value;

  return NULL;
}

template<typename K, typename V>
bool MyMap<K, V>::isEmpty() const
{
  return size == 0;
}

template<typename K, typename V>
vector<Entry<K, V>> MyMap<K, V>::getEntries() const
{
  vector<Entry<K, V>> v;

  for (int i = 0; i < capacity; i++)
  {
    for (Entry<K, V>& entry: table[i])
      v.push_back(entry);
  }

  return v;
}

template<typename K, typename V>
bool MyMap<K, V>::containsKey(K key) const
{
  return get(key) != NULL;
}

template<typename K, typename V>
```

```
bool MyMap<K, V>::containsValue(V value) const
{
  for (int i = 0; i < capacity; i++)
  {
    for (Entry<K, V> entry: table[i])
      if (entry.value == value)
        return true;
  }

  return false;
}

template<typename K, typename V>
void MyMap<K, V>::remove(K key)
{
  int bucketIndex = hash(hashCode(key));

  // Remove the first entry that matches the key from a bucket
  if (table[bucketIndex].size() > 0)
  {
    for (auto p = table[bucketIndex].begin();
        p != table[bucketIndex].end(); p++)
      if (p->key == key)
      {
        table[bucketIndex].erase(p);
        size--; // Decrease size
        break; // Remove just one entry that matches the key
      }
  }
}

template<typename K, typename V>
void MyMap<K, V>::clear()
{
  size = 0;
  removeEntries();
}

template<typename K, typename V>
void MyMap<K, V>::removeEntries()
{
  for (int i = 0; i < capacity; i++)
  {
    table[i].clear();
  }
}

template<typename K, typename V>
vector<K> MyMap<K, V>::getKeys() const
{
  // Left as exercise
}

template<typename K, typename V>
vector<V> MyMap<K, V>::getValues() const
{
  // Left as exercise
}

template<typename K, typename V>
```

```
string MyMap<K, V>::toString() const
{
  stringstream ss;
  ss << "[";

  for (int i = 0; i < capacity; i++)
  {
    for (Entry<K, V>& entry: table[i])
      ss << entry.toString();
  }

  ss << "]";
  return ss.str();
}

template<typename K, typename V>
unsigned MyMap<K, V>::hashCode(K key) const
{
  return typeid(key).hash_code();
}

template<typename K, typename V>
int MyMap<K, V>::hash(int hashCode) const
{
  return supplementalHash(hashCode) & (capacity - 1);
}

template<typename K, typename V>
int MyMap<K, V>::supplementalHash(int h) const
{
  h ^= (h >> 20) ^ (h >> 12);
  return h ^ (h >> 7) ^ (h >> 4);
}

template<typename K, typename V>
int MyMap<K, V>::trimToPowerOf2(int initialCapacity)
{
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
  }

  return capacity;
}

template<typename K, typename V>
void MyMap<K, V>::rehash()
{
  vector<Entry<K, V>> set = getEntries(); // Get entries
  capacity <<= 1; // Double capacity
  delete[] table; // Delete old hash table
  table = new vector<Entry<K, V>>[capacity]; // Create a new hash table
  size = 0; // Reset size to 0

  for (Entry<K, V>& entry: set)
    put(entry.key, entry.value); // Store to new table
}

template<typename K, typename V>
```

```
300  int MyMap<K, V>::getSize() const
301  {
302    return size;
303  }
304
305  #endif
```

<margin note (line 10)>default initial capacity

<margin note (line 11)>default load factor

<margin note (line 12)>maximum capacity

<margin note (line 16)>class Entry

<margin note (line 19)>Entry constructor

<margin note (line 25)>toString() for entry

<margin note (line 32)>key in the entry

<margin note (line 33)>value in the entry

<margin note (line 37)>class MyMap

<margin note (line 40)>MyMap constructors

<margin note (line 44)>MyMap public functions

<margin note (line 58)>MyMap private data

<margin note (line 58)>MyMap private data

<margin note (line 63)>MyMap private functions

<margin note (line 74)>no-arg constructor

<margin note (line 77)>array for hash table

<margin note (line 83)>constructor

<margin note (line 92)>constructor

<margin note (line 105)>put

<margin note (line 116)>replace old value

<margin note (line 129)>rehash

<margin note (line 135)>add a new entry

<margin note (line 143)>get

<margin note (line 148)>a match

<margin note (line 106)>isEmpty

<margin note (line 161)>getEntries

<margin note (line 161)>add an entry to v

<margin note (line 175)>containsKey

<margin note (line 181)>containsValue

<margin note (line 186)>a match

<margin note (line 194)>remove

<margin note (line 203)>a match

<margin note (line 205)>remove an entry

<margin note (line 213)>clear

<margin note (line 216)>remove all entries

<margin note (line 220)>remove entries

<margin note (line 229)>getKeys

<margin note (line 235)>getValues

<margin note (line 241)>toString

<margin note (line 257)>hashCode

<margin note (line 263)>hash

<margin note (line 269)>supplementalHash

<margin note (line 276)>trimToPowerOf2

<margin note (line 287)>rehash

<end listing 27.1>

The `MyMap` class implements the `MyMap` interface using separate chaining. The parameters that determine the hash-table capacity (line 60) and load factors (line 59) are defined in the class. The default initial capacity is `4` (line 10) and the maximum capacity is 2^{30} (line 12). The current hash-table capacity is designed as a value of the power of `2`. The default load-factor threshold is `0.75f` (line 11). You can specify a custom load-factor threshold when constructing a map. The custom load-factor threshold is stored in `loadFactorThreshold` (line 59). The data field `size` denotes the number of entries in the map (line 58). The hash table is an array. Each cell in the array is a vector (line 63).

<*margin note*>hash-table parameters

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 73–80), a map with the specified capacity and a default load-factor threshold (lines 82–89), and a map with the specified capacity and load-factor threshold (lines 91–102).

<*margin note*>three constructors

The **`put(key, value)`** function adds a new entry into the map (lines 104-140). The function first tests if the key is already in the map (line 107), if so, it locates the entry and replaces the old value with the new value in the entry for the key (line 116) and the old value is returned (line 118). If the key is new in the map, the new entry is created in the map (line 135). Before inserting the new entry, the function checks whether the size exceeds the load-factor threshold (line 124). If so, the program invokes **`rehash()`** (line 129) to increase the capacity and store entries into the new larger hash table.

<*margin note*>put

The **`rehash()`** function first copies all entries into a vector (line 289), doubles the capacity (line 290), deletes the current hash table (line 291), creates a new hash table (line 292), and resets the size to **`0`** (line 293). The function then copies the entries into the new hash table (lines 295–296). The **`rehash`** function takes $O(capacity)$ time. If no rehash is performed, the **`put`** function takes $O(1)$ time to add a new entry.

<*margin note*>rehash

The **`get(key)`** function returns the value of the first entry with the specified key (lines 143–152). This function takes $O(1)$ time. This function invokes **`hashCode(key)`**, which returns the hash code for the key (line 259).

<*margin note*>get

<*margin note*>hashCode

The **`hash()`** function invokes the **`supplementalHash`** to ensure that the hashing is evenly distributed to produce an index for the hash table (lines 263–266). This function takes $O(1)$ time.

<*margin note*>hash

The **`remove(key)`** function removes the entry with the specified key in the map (lines 193–210). This function takes $O(1)$ time.

<*margin note*>remove

The `getSize()` function simply returns the size of the map (lines 299–303). This function takes $O(1)$ time.

<margin note>getSize

The `getKeys()` function returns all keys in the map as a set. The function finds the keys from each bucket and adds them to a set (lines 229–232). This function takes $O(capacity)$ time. For the implementation of this function, see Programming Exercise 27.10.

<margin note>getKeys

The `getValues()` function returns all values in the map. The function examines each entry from all buckets and adds it to a set (lines 234–238). This function takes $O(capacity)$ time. For the implementation of this function, see Programming Exercise 27.11.

<margin note>values

The `clear` function removes all entries from the map (lines 212–217). It invokes `removeEntries()`, which deletes all entries in the buckets (lines 222–225). The `removeEntries()` function (lines 219-226) takes $O(capacity)$ time to clear all entries in the table.

<margin note>clear

<margin note>removeEntries

The `containsKey(key)` function checks whether the specified key is in the map by invoking the `get` function (lines 174–178). Since the `get` function takes $O(1)$ time, the `containsKey(key)` function takes $O(1)$ time.

<margin note>containsKey

The `containsValue(value)` function checks whether the value is in the map (lines 181–191). This function takes $O(capacity + size)$ time. It is actually $O(capacity)$, since $capacity > size$.

<margin note>containsValue

The `getEntries()` function returns a vector that contains all entries in the map (lines 160–172). This function takes $O(capacity)$ time.

<margin note>getEntries

The `isEmpty()` function simply returns true if the map is empty (lines 154–158). This function takes $O(1)$ time.

<margin note>isEmpty

Table 27.1 summarizes the time complexities of the functions in **MyMap**.

Table 27.1

Time Complexities for Functions in ***MyMap***

Functions	*Time*
clear()	$O(capacity)$
containsKey(key: Key)	$O(1)$
containsValue(value: V)	$O(capacity)$
getEntries()	$O(capacity)$
get(key: K)	$O(1)$
isEmpty()	$O(1)$
getKeys()	$O(capacity)$
put(key: K, value: V)	$O(1)$
remove(key: K)	$O(1)$
getSize()	$O(1)$
getValues()	$O(capacity)$
rehash()	$O(capacity)$

Since rehashing does not happen very often, the time complexity for the **put** function is $O(1)$. Note that the complexities of the **clear**, **getEntries**, **getKeys**, **getValues**, and **rehash** functions depend on **capacity**, so to avoid poor performance for these functions you should choose an initial capacity carefully.

Listing 27.2 gives a test program that uses **MyMap**.

Listing 27.2 TestMyMap.cpp

```
#include <iostream>
#include <string>
#include "MyMap.h"
using namespace std;

int main()
{
  // Create a map
```

```
 9    MyMap<string, int> map;
10    map.put("Smith", 30);
11    map.put("Anderson", 31);
12    map.put("Lewis", 29);
13    map.put("Cook", 29);
14    map.put("Smith", 65);
15
16    cout << "Entries in map: " << map.toString() << endl;
17    cout << "The age for " << "Lewis is " <<
18      map.get("Lewis") << endl;
19    cout << "Is Smith in the map? " <<
20      (map.containsKey("Smith") ? "true" : "false") << endl;
21    cout << "Is age 33 in the map? " <<
22      (map.containsValue(33) ? "true" : "false") << endl;
23
24    map.remove("Smith");
25    cout << "Entries in map: " << map.toString() << endl;
26
27    map.clear();
28    cout << "Entries in map: " << map.toString() << endl;
29
30    return 0;
31  }
```

<output>

```
Entries in map: [[Smith, 65][Anderson, 31][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: []
```

<end output>

<margin note (line 8)>create a map

<margin note (line 9)>put entries

<margin note (line 15)>display entries

<margin note (line 17)>get value

<margin note (line 19)>is key in map?

<margin note (line 21)>is value in map?

<margin note (line 23)>remove entry

<margin note (line 26)>clear map

<end listing 27.2>

The program creates a map using **MyMap** (line 9) and adds five entries into the map (lines 10–14). Line 10 adds key **Smith** with value **30** and line 14 adds **Smith** with value **65**. The latter value replaces the former value. The map actually has only four entries. The program displays the entries in the map (line 16), gets a value for a key (line 18), checks whether the map contains the key (line 20) and a value (line 22), removes an entry with the key **Smith** (line 24), and redisplays the entries in the map (line 25). Finally, the program clears the map (line 27) and displays an empty map (line 28).

<check point>

27.20 What is **1 << 30** in line 13 in Listing 27.1? What are the integers resulted from **1 << 1**, **1 << 2**, and **1 << 3**?

27.21 What are the integers resulted from **32 >> 1**, **32 >> 2**, **32 >> 3**, and **32 >> 4**?

27.22 In Listing 27.2, will the program work if **vector** is replaced by **list**?

27.23 Describe how the **put(key, value)** function is implemented in the **MyMap** class.

27.24 Show the printout of the following code.

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

<end check point>

27.8 Implementing Set Using Hashing

<key point>

A hash set can be implemented using a hash map.

<end key point>

A *set* (introduced in Chapter 22) is a data structure that stores distinct values. The C++ STL defines the **set** and **multiset** for modeling sets. You can implement them using the same approach as for implementing **MyMap**. The only difference is that key/value pairs are stored in the map, while elements are stored in the set.

We design our custom set class to mirror **set** and name the it **MySet**, as shown in Figure 27.10.

<margin note>MySet

Figure 27.10

MySet *implements the set class.*

MySet<K>	
-size: int	The number of the keys in the set.
-capacity: int	table points an array of buckets, each of which is a vector.
-loadFactorThreshold: float	Load factor of the map.
-table: vector<K>*	The size of the array for hash table.
+MySet()	Creates an empty set with default capacity 4 and default load factor threshold 0.75.
+MySet(capacity: int)	Creates a set with a specified capacity and default load factor threshold 0.75.
+MySet(capacity: int, loadFactorThreshold: float)	Creates a set with a specified capacity and load factor threshold.
+clear(): void	Removes all keys from this set.
+contains(key: K): bool const	Returns true if this set contains the specified key.
+isEmpty(): bool	Returns true if this set contains no set keys.
+add(key: K): bool	Adds a key in this set. Return true if successful.
+remove(key: K): bool	Removes the specified key. Return true if successful.
+getSize(): int const	Returns the number of keys in this set.
+getKeys(): vector<K> const	Returns the keys in the set to a vector.
+toString(): string const	Returns a string representation for this set.

Listing 27.4 shows the **MySet** interface and Listing 27.5 implements **MySet** using separate chaining.

Listing 27.3 MySet.h

```
#ifndef MYSet_H
#define MYSet_H

#include <vector>
#include <string>
#include <sstream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int DEFAULT_INITIAL_CAPACITY = 4;
float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
unsigned MAXIMUM_CAPACITY = 1 << 30;

template<typename K>
```

```
class MySet
{
public:
  MySet();
  MySet(int initialCapacity);
  MySet(int initialCapacity, float loadFactorThreshold);

  int getSize() const;
  bool isEmpty() const;
  bool contains(K key) const;
  bool add(K key);
  bool remove(K key);
  void clear();
  vector<K> getKeys() const;
  string toString() const;

private:
  int size;
  float loadFactorThreshold;
  int capacity;

  // Hash table is an array with each cell as a vector
  vector<K>* table;

  int hash(int hashCode) const;
  unsigned hashCode(K key) const;
  int supplementalHash(int h) const;
  int trimToPowerOf2(int initialCapacity);
  void rehash();
  void removeKeys();
};

template<typename K>
MySet<K>::MySet()
{
  capacity = DEFAULT_INITIAL_CAPACITY;
  table = new vector<K>[capacity];
  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR;
  size = 0;
}

template<typename K>
MySet<K>::MySet(int initialCapacity)
{
  capacity = initialCapacity;
  table = new vector<K>[capacity];
  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR;
  size = 0;
}

template<typename K>
MySet<K>::MySet(int initialCapacity, float loadFactorThreshold)
{
  if (initialCapacity > MAXIMUM_CAPACITY)
    capacity = MAXIMUM_CAPACITY;
  else
    capacity = trimToPowerOf2(initialCapacity);

  this->loadFactorThreshold = loadFactorThreshold;
```

```
  table = new vector<K>[capacity];
  size = 0;
}

template<typename K>
bool MySet<K>::add(K key)
{
  if (contains(key)) return false; // key is already in the set

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

    rehash();
  }

  int bucketIndex = hash(hashCode(key));

  // Add a new entry (key, value) to hashTable[index]
  table[bucketIndex].push_back(key);

  size++; // Increase size

  return true;
}

template<typename K>
bool MySet<K>::isEmpty() const
{
  return size == 0;
}

template<typename K>
bool MySet<K>::contains(K key) const
{
  int bucketIndex = hash(hashCode(key));

  for (K& e: table[bucketIndex])
    if (e == key)
      return true;

  return false;
}

template<typename K>
bool MySet<K>::remove(K key)
{
  int bucketIndex = hash(hashCode(key));

  // Remove the first entry that matches the key from a bucket
  if (table[bucketIndex].size() > 0)
  {
    for (auto p = table[bucketIndex].begin();
        p != table[bucketIndex].end(); p++)
      if (*p == key)
      {
        table[bucketIndex].erase(p);
        size--; // Decrease size
```

```
        return true; // Remove just one entry that matches the key
      }
  }

  return false;
}

template<typename K>
void MySet<K>::clear()
{
  size = 0;
  removeKeys();
}

template<typename K>
vector<K> MySet<K>::getKeys() const
{
  vector<K> v;

  for (int i = 0; i < capacity; i++)
  {
    for (K& e: table[i])
      v.push_back(e);
  }

  return v;
}

template<typename K>
void MySet<K>::removeKeys()
{
  for (int i = 0; i < capacity; i++)
  {
    table[i].clear();
  }
}

template<typename K>
string MySet<K>::toString() const
{
  stringstream ss;
  ss << "[";

  for (int i = 0; i < capacity; i++)
  {
    for (K& e: table[i])
      ss << e << " ";
  }

  ss << "]";
  return ss.str();
}

template<typename K>
unsigned MySet<K>::hashCode(K key) const
{
  return typeid(key).hash_code();
}

template<typename K>
```

```
195  int MySet<K>::hash(int hashCode) const
196  {
197    return supplementalHash(hashCode) & (capacity - 1);
198  }
199
200  template<typename K>
201  int MySet<K>::supplementalHash(int h) const
202  {
203    h ^= (h >> 20) ^ (h >> 12);
204    return h ^ (h >> 7) ^ (h >> 4);
205  }
206
207  template<typename K>
208  int MySet<K>::trimToPowerOf2(int initialCapacity)
209  {
210    int capacity = 1;
211    while (capacity < initialCapacity) {
212      capacity <<= 1;
213    }
214
215    return capacity;
216  }
217
218  template<typename K>
219  void MySet<K>::rehash()
220  {
221    vector<K> set = getKeys(); // Get entries
222    capacity <<= 1; // Double capacity
223    delete[] table; // Delete old hash table
224    table = new vector<K>[capacity]; // Create a new hash table
225    size = 0; // Reset size to 0
226
227    for (K& e: set)
228      add(e); // Store to new table
229  }
230
231  template<typename K>
232  int MySet<K>::getSize() const
233  {
234    return size;
235  }
236
237  #endif
```

<margin note (line 11)>default initial capacity

<margin note (line 12)>default max load factor

<margin note (line 13)>maximum capacity

<margin note (line 16)>class MySet

<margin note (line 33)>size

<margin note (line 35)>current capacity

<margin note (line 34)>load-factor threshold

<margin note (line 38)>hash table

<margin note (line 49)>no-arg constructor

<margin note (line 58)>constructor

<margin note (line 67)>constructor

<margin note (line 80)>add

<margin note (line 104)>isEmpty

<margin note (line 110)>contains

<margin note (line 122)>remove

<margin note (line 143)>clear

<margin note (line 173)>toString

<margin note (line 195)>hash

<margin note (line 201)>supplementalHash

<margin note (line 208)>trimToPowerOf2

<margin note (line 219)>rehash

<margin note (line 232)>getSize

<end listing 27.3>

Implementing **`MySet`** is very similar to implementing **`MyMap`** except that the keys are stored in the hash table for **`MySet`**, but the entries (key/value pairs) are stored in the hash table for **`MyMap`**.

<margin note>MySet vs. MyMap

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 48–55), a set with the specified capacity and a default load factor (lines 57–64), and a set with the specified capacity and load factor (lines 66–77).

<margin note>three constructors

The **`add(key)`** function adds a new key into the set. The function first checks if the key is already in the set (line 82). If so, the function returns false. The function then checks whether the size exceeds the load-factor threshold (line 85). If so, the program invokes **`rehash()`** (line 90) to increase the capacity and store keys into the new larger hash table.

<margin note>add

The **`rehash()`** function first copies all keys in a list (line 221), doubles the capacity (line 222), obtains a new threshold (line 223), creates a new hash table (line 224), and resets the size to **0** (line 225). The function then copies the keys into the new larger hash table (lines 227–228). The **rehash** function takes *O*(*capacity*) time. If no rehash is performed, the **add** function takes *O*(1) time to add a new key.

<margin note>rehash

The **clear** function removes all keys from the set (lines 142–147). It invokes **removeKeys()**, which clears all table cells (line 168). Each table cell is a vector that stores the keys with the same hash code. The **removeKeys()** function takes *O*(*capacity*) time.

<margin note>clear

The **contains(key)** function checks whether the specified key is in the set by examining whether the designated bucket contains the key (lines 109–119). This function takes *O*(1) time.

<margin note>contains

The **remove(key)** function removes the specified key in the set (lines 121–140). This function takes *O*(1) time.

<margin note>remove

The **getSize()** function simply returns the number of keys in the set (lines 231–235). This function takes *O*(1) time.

<margin note>size

The **hashCode**, **hash**, **supplementalHash**, and **trimToPowerOf2** are the same as in the **MyMap** class.

Table 27.2 summarizes the time complexity of the functions in **MySet**.

Table 27.2

Time Complexities for Functions in **MySet**

Functions	*Time*
clear()	*O*(*capacity*)
contains(k: K)	*O*(1)
add(k: K)	*O*(1)
remove(k: K)	*O*(1)
isEmpty()	*O*(1)

`getSize()`	*O*(1)
`getKeys()`	*O*(*size*)
`rehash()`	*O*(*capacity*)

Listing 27.4 gives a test program that uses **MySet**.

Listing 27.4 TestMySet.cpp

```
#include <iostream>
#include <string>
#include "MySet.h"
using namespace std;

int main()
{
  // Create a MySet
  MySet<string> set;
  set.add("Smith");
  set.add("Anderson");
  set.add("Lewis");
  set.add("Cook");
  set.add("Smith");

  cout << "Keys in set: " << set.toString() << endl;
  cout << "Number of keys in set: " << set.getSize() << endl;
  cout << "Is Smith in set? " << set.contains("Smith") << endl;

  set.remove("Smith");
  cout << "Names in set are ";
  for (string s: set.getKeys())
    cout << s << " ";

  set.clear();
  cout << "\nKeys in set: " << set.toString() << endl;

  return 0;
}
```

<output>

```
Keys in set: [Cook Anderson Smith Lewis ]

Number of keys in set: 4

Is Smith in set? true

Names in set are Cook Anderson Lewis

Keys in set: []
```

<end output>

<margin note (line 4)>create a set

<margin note (line 5)>add keys

<margin note (line 11)>display keys

<margin note (line 12)>set size

<margin note (line 15)>remove key

<margin note (line 17)>for-each loop

<margin note (line 20)>clear set

<end listing 27.6>

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

<check point>

27.25 Describe how the **`add(k)`** function is implemented in the **`MySet`** class.

27.26 Describe how the **`remove(k)`** function is implemented in the **`MySet`** class.

27.27 Describe how the **`contains(k)`** function is implemented in the **`MySet`** class.

<end check point>

Key Terms

associative array 998

cluster 1002

dictionary 998

double hashing 1003

hash code 999

hash function 998

hash map 1016

hash set 1016

hash table 998

linear probing 1001

load factor 1005

open addressing 1001

perfect hash function 998

polynomial hash code 1000

quadratic probing 1002

rehashing 1005

secondary clustering 1003

separate chaining 1005

Chapter Summary

1. A *map* is a data structure that stores entries. Each entry contains two parts: a *key* and a *value*. The key is also called a *search key*, which is used to search for the corresponding value. You can implement a map to obtain *O*(1) time complexity on searching, retrieval, insertion, and deletion using the hashing technique.
2. A *set* is a data structure that stores elements. You can use the hashing technique to implement a set to achieve *O*(1) time complexity on searching, insertion, and deletion for a set.
3. *Hashing* is a technique that retrieves the value using the index obtained from a key without performing a search. A typical *hash function* first converts a search key to an integer value called a *hash code*, then compresses the hash code into an index to the *hash table*.
4. A *collision* occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: *open addressing* and *separate chaining*.
5. Open addressing is the process of finding an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.
6. The *separate chaining* scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a

container that holds multiple entries.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

****27.1** (*Implement `MyMap` using open addressing with linear probing*) Create a new concrete class that implements `MyMap` using open addressing with linear probing. For simplicity, use `f(key) = key % size` as the hash function, where size is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.2** (*Implement `MyMap` using open addressing with quadratic probing*) Create a new concrete class that implements `MyMap` using open addressing with quadratic probing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

****27.3** (*Implement `MyMap` using open addressing with double hashing*) Create a new concrete class that implements `MyMap` using open addressing with double hashing. For simplicity, use `f(key) = key % size` as the hash function, where `size` is the hash-table size. Initially, the hash-table size is `4`. The table size is doubled whenever the load factor exceeds the threshold (`0.5`).

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

****27.5** (*Implement `MySet` using `MyMap`*) Implement `MySet` using `MyMap`. Note that you can create entries with (`key, key`), rather than (`key, value`).

****27.6** (*Implement `floatToIntBits`*) Write the following function that returns a 32-bit float value as an int. Note that the float value and int have the same binary representation.

```
int floatToIntBits(float value)
```

****27.7** (*Implement `doubleToLongLongBits`*) Write the following function that returns a 64-bit double value as a long long. Note that the long long value and double have the same binary representation.

```
int doubleToLongLongBits(double value)
```

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

```
int hashCodeForString(string& s)
```

27.9** (*Implement* ***getKeys) Implement the ***getKeys*** function defined in Listing 24.1.

27.10** (*Implement* ***getValues) Implement the ***getValues*** function defined in Listing 24.1.

27.11** (*Implement* ***MyMap *with iterator*) Modify Listing 24.1 to define an iterator class and add the functions **begin()** and **end()** that return an iterator for traversing the entries in the map.

27.12** (*Implement* ***MyMultiMap) Modify Listing 24.1 to implement **MyMultiMap** to store key/value entries with duplicate keys allowed.

27.13** (*Implement* ***MySet *with iterator*) Modify Listing 24.3 to define an iterator class and add the functions **begin()** and **end()** that return an iterator for traversing the keys in the map.

27.14** (*Implement* ***MyMultiSet) Modify Listing 24.3 to implement **MyMultiSet** to store keys with duplicate keys allowed.

27.15** (*Compare* ***MySet *and* ***vector***) **MySet** is defined in Listing 24.3. Write a program that generates **1000000** random integers between **0** and **999999**, shuffles them, and stores them in a **vector** and in a **MySet**. Generate a list of **1000000** random integers between **0** and **1999999**. For each number in the list, test if it is in the array list and in the hash set. Run your program to display the total test time for the array list and for the hash set.