Chapter 7

Sets and Maps

Chapter Objectives

- To understand the Java `Map` and `Set` interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance tradeoffs
- To learn how to implement both hash table forms
- To be introduced to the implementation of `Map`s and `Set`s
- To see how two earlier applications can be implemented more easily using `Map` objects for data storage

In Chapter 2, we introduced the Java Collections Framework, focusing on the `List` interface and the classes that implement it (`ArrayList` and `LinkedList`). The classes that implement the `List` interface are all indexed collections. That is, there is an index or a subscript associated with each member (element) of an object of these classes. Often an element's index reflects the relative order of its insertion in the `List` object. Searching for a particular value in a `List` object is generally an $O(n)$ process. The exception is a binary search of a sorted object, which is an $O(\log n)$ process.

In this chapter, we consider the other part of the `Collection` hierarchy: the `Set` interface and the classes that implement it. `Set` objects are not indexed, and the order of insertion of items is not known. Their main purpose is to enable efficient search and retrieval of information. It is also possible to remove elements from these collections without moving other elements around. By contrast, if an element is removed from the collection in an `ArrayList` object, the elements that follow it are normally shifted over to fill the vacated space.

A second, related interface is the `Map`. `Map` objects provide efficient search and retrieval of entries that consist of pairs of objects. The first object in each pair is the key (a unique value), and the second object is the information associated with that key. You retrieve an object from a `Map` by specifying its key.

We also study the hash table data structure. The hash table is a very important data structure that has been used very effectively in compilers and in building dictionaries. It can be

used as the underlying data structure for a `Map` or `Set` implementation. It stores objects at arbitrary locations and offers an average constant time for insertion, removal, and searching.

We will see two ways to implement a hash table and how to use it as the basis for a class that implements the `Map` or `Set`. We will not show you the complete implementation of an object that implements `Map` or `Set` because we expect that you will use the ones provided by the Java API. However, we will certainly give you a head start on what you need to know to implement these interfaces.

Sets and Maps

7.1 Sets and the `Set` Interface
7.2 Maps and the `Map` Interface
7.3 Hash Tables
7.4 Implementing the Hash Table
7.5 Implementation Considerations for Maps and Sets
7.6 Additional Applications of Maps
Case Study: Implementing a Cell Phone Contact List
Case Study: Completing the Huffman Coding Problem
7.7 Navigable Sets and Maps

7.1 Sets and the `Set` Interface

We introduced the Java Collections Framework in Chapter 2. We covered the part of that framework that focuses on the `List` interface and its implementers. In this section, we explore the `Set` interface and its implementers.

Figure 7.1 shows the part of the Collections Framework that relates to sets. It includes interfaces `Set`, `Sortedset`, and `Navigableset`; abstract class `AbstractSet`; and actual classes `HashSet`, `TreeSet`, and `ConcurrentSkipListSet`. The `HashSet` is a set that is implemented using a hash table (discussed in Section 7.3). The `TreeSet` is implemented using a special kind of binary search tree, called the Red–Black tree (discussed in Chapter 9). The `ConcurrentSkipListSet` is implemented using a skip list (discussed in Chapter 9). In Section 6.5, we showed how to use a `TreeSet` to store an index for a term paper.

The `Set` Abstraction

The Java API documentation for the interface `java.util.Set` describes the `Set` as follows:

> A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one `null` element. As implied by its name, this interface models the mathematical *set* abstraction.

What mathematicians call a *set* can be thought of as a collection of objects. There is the additional requirement that the elements contained in the set are unique. For example, if we have the set of fruits `{"apples", "oranges", and "pineapples"}` and add `"apples"` to it, we still have the same set. Also, we usually want to know whether or not a particular object is a member of the set rather than where in the set it is located. Thus, if `s` is a set, we would be interested in the expression

```
s.contains("apples")
```

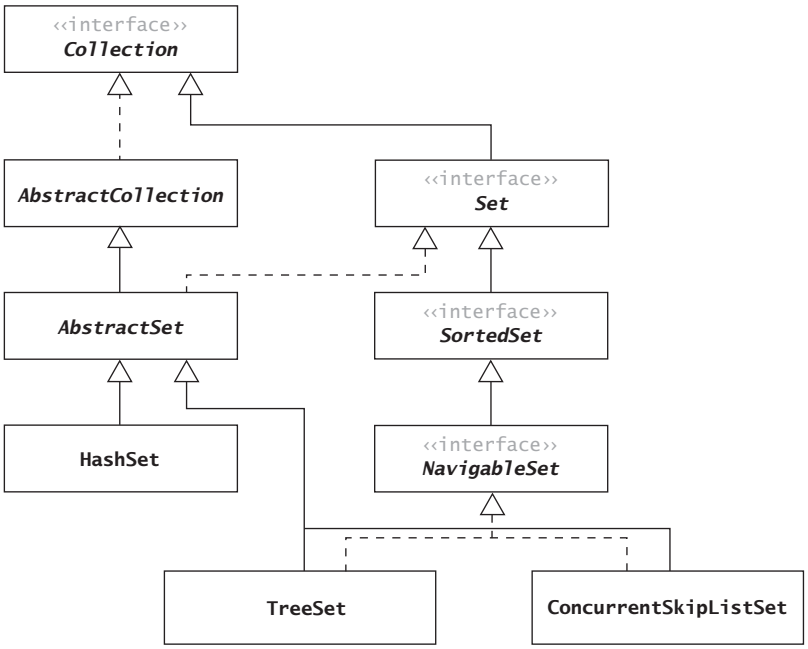


FIGURE 7.1
The `Set` Hierarchy

which returns the value `true` if `"apples"` is in set `s` and `false` if it is not. We would not have a need to use a method such as

```
s.indexOf("apples")
```

which might return the location or position of `"apples"` in set `s`. Nor would we have a need to use the expression

```
s.get(i)
```

where `i` is the position (index) of an object in set `s`.

We assume that you are familiar with sets from a course in discrete mathematics. Just as a review, however, the operations that are performed on a mathematical set are testing for membership (method `contains`), adding elements, and removing elements. Other common operations on a mathematical set are *set union* (A ∪ B), *set intersection* (A ∩ B), and *set difference* (A – B). There is also a *subset operator* (A ⊂ B). These operations are defined as follows:

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.
  Example: {1, 3, 5, 7} ∪ {2, 3, 4, 5} is {1, 2, 3, 4, 5, 7}
- The intersection of sets A, B is the set whose elements belong to both A and B.
  Example: {1, 3, 5, 7} ∩ {2, 3, 4, 5} is {3, 5}
- The difference of sets A, B is the set whose elements belong to A but not to B.
  Examples: {1, 3, 5, 7} – {2, 3, 4, 5} is {1, 7}; {2, 3, 4, 5} – {1, 3, 5, 7} is {2, 4}
- Set A is a subset of set B if every element of set A is also an element of set B.
  Example: {1, 3, 5, 7} ⊂ {1, 2, 3, 4, 5, 7} is true

The Set Interface and Methods

A `Set` has required methods for testing for set membership (`contains`), testing for an empty set (`isEmpty`), determining the set size (`size`), and creating an iterator over the set (`iterator`). It has optional methods for adding an element (`add`) and removing an element (`remove`). It

TABLE 7.1
Some `java.util.Set<E>` Methods (with Mathematical Set Operations in Italics)

Method	Behavior
`boolean add(E obj)`	Adds item `obj` to this set if it is not already present (optional operation) and returns **`true`**. Returns false if `obj` is already in the set
`boolean addAll(Collection<E> coll)`	Adds all of the elements in collection `coll` to this set if they're not already present (optional operation). Returns **`true`** if the set is changed. Implements *set union* if `coll` is a `Set`
`boolean contains(Object obj)`	Returns **`true`** if this set contains an element that is equal to `obj`. Implements a test for *set membership*
`boolean containsAll(Collection<E> coll)`	Returns **`true`** if this set contains all of the elements of collection `coll`. If `coll` is a set, returns **`true`** if this set is a subset of `coll`
`boolean isEmpty()`	Returns **`true`** if this set contains no elements
`Iterator<E> iterator()`	Returns an iterator over the elements in this set
`boolean remove(Object obj)`	Removes the set element equal to `obj` if it is present (optional operation). Returns **`true`** if the object was removed
`boolean removeAll(Collection<E> coll)`	Removes from this set all of its elements that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set difference* operation
`boolean retainAll(Collection<E> coll)`	Retains only the elements in this set that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set intersection* operation
`int size()`	Returns the number of elements in this set (its cardinality)

provides the additional restriction on constructors that all sets they create must contain no duplicate elements. It also puts the additional restriction on the `add` method that a duplicate item cannot be inserted. Table 7.1 shows the commonly used methods of the `Set` interface. The `Set` interface also has methods that support the mathematical set operations. The required method `containsAll` tests the subset relationship. There are optional methods for set union (`addAll`), set intersection (`retainAll`), and set difference (`removeAll`). We show the methods that are used to implement the mathematical set operations in italics in Table 7.1.

Calling a method "optional" means just that an implementer of the `Set` interface is not required to provide it. However, a method that matches the signature must be provided. This method should throw the `UnsupportedOperationException` whenever it is called. This gives the class designer some flexibility. For example, if a class instance is intended to provide efficient search and retrieval of the items stored, the class designer may decide to omit the optional mathematical set operations.

FOR PYTHON PROGRAMMERS

The Python `Set` class is similar to the Java `HashSet` class. Both have operations for creating sets, adding and removing objects, and forming union, intersection, and difference.

EXAMPLE 7.1 Listing 7.1 contains a `main` method that creates three sets: `setA`, `setAcopy`, and `setB`. It loads these sets from two arrays and then forms their union in `setA` and their intersection in `setAcopy`, using the statements

```
setA.addAll(setB);         // Set union
setAcopy.retainAll(setB);  // Set intersection
```

Running this method generates the output lines below. The brackets and commas are inserted by method `toString`.

```
The 2 sets are:
[Jill, Ann, Sally]
[Bill, Jill, Ann, Bob]
Items in set union are: [Bill, Jill, Ann, Sally, Bob]
Items in set intersection are: [Jill, Ann]
```

LISTING 7.1
Illustrating the Use of Sets

```
public static void main(String[] args) {

    // Create the sets.
    String[] listA = {"Ann", "Sally", "Jill", "Sally"};
    String[] listB = {"Bob", "Bill", "Ann", "Jill"};
    Set<String> setA = new HashSet<>();
    Set<String> setAcopy = new HashSet<>();  // Copy of setA
    Set<String> setB = new HashSet<>();

    // Load sets from arrays.
    for (String s : listA) {
        setA.add(s);
        setAcopy.add(s);
    }
    for (String s : listB) {
        setB.add(s);
    }
    System.out.println("The 2 sets are: " + "\n" + setA
                       + "\n" + setB);
    // Display the union and intersection.
    setA.addAll(setB);         // Set union
    setAcopy.retainAll(setB);  // Set intersection
    System.out.println("Items in set union are: " + setA);
    System.out.println("Items in set intersection are: "
                       + setAcopy);
}
```

Comparison of Lists and Sets

Collections implementing the `Set` interface must contain unique elements. Unlike the `List.add` method, the `Set.add` method will return **`false`** if you attempt to insert a duplicate item.

Unlike a `List`, a `Set` does not have a `get` method. Therefore, elements cannot be accessed by index. So if `setA` is a `Set` object, the method call `setA.get(0)` would cause the syntax error `method get(int) not found`.

Although you can't reference a specific element of a `Set`, you can iterate through all its elements using an `Iterator` object. The loop below accesses each element of `Set` object `setA`.

However, the elements will be accessed in arbitrary order. This means that they will not necessarily be accessed in the order in which they were inserted.

```
// Create an iterator to setA.
Iterator<String> setAIter = setA.iterator();
while (setAIter.hasNext()) {
    String nextItem = setAIter.next();
    // Do something with nextItem
    . . .
}
```

We can simplify the task of accessing each element in a `Set` using the Java 5.0 enhanced **`for`** statement.

```
for (String nextItem : setA) {
    // Do something with nextItem
    . . .
}
```

EXERCISES FOR SECTION 7.1

SELF-CHECK

1. Explain the effect of the following method calls.

```
Set<String> s = new HashSet<String>();
s.add("hello");
s.add("bye");
s.addAll(s);
Set<String> t = new TreeSet<String>();
t.add("123");
s.addAll(t);
System.out.println(s.containsAll(t));
System.out.println(t.containsAll(s));
System.out.println(s.contains("ace"));
System.out.println(s.contains("123"));
s.retainAll(t);
System.out.println(s.contains("123"));
t.retainAll(s);
System.out.println(t.contains("123"));
```

2. What is the relationship between the `Set` interface and the `Collection` interface?

3. What are the differences between the `Set` interface and the `List` interface?

4. In Example 7.1, why is `setAcopy` needed? What would happen if you used the statement

```
setAcopy = setA;
```

to define `setAcopy`?

PROGRAMMING

1. Assume you have declared three sets `a`, `b`, and `c` and that sets `a` and `b` store objects. Write statements that use methods from the `Set` interface to perform the following operations:

a. `c = (a ∪ b)`

b. `c = (a ∩ b)`

c. `c = (a – b)`

d.
```
if (a ⊂ b)
      c = a;
  else
      c = b;
```

2. Write a `toString` method for a class that implements the `Set` interface and displays the set elements in the form shown in Example 9.1.

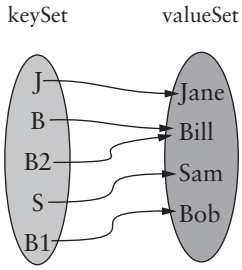
7.2 Maps and the `Map` Interface

The `Map` is related to the `Set`. Mathematically, a `Map` is a set of ordered pairs whose elements are known as the key and the value. The key is required to be unique, as are the elements of a set, but the value is not necessarily unique. For example, the following would be a map:

{(**J**, Jane), (**B**, Bill), (**S**, Sam), (**B1**, Bob), {**B2**, Bill)}

The keys in this example are strings consisting of one or two characters, and each value is a person's name. The keys are unique but not the values (there are two Bills). The key is based on the first letter of the person's name. The keys **B1** and **B2** are the keys for the second and third person whose name begins with the letter B.

FIGURE 7.2
Example of Mapping



You can think of each key as "mapping" to a particular value (hence the name *map*). For example, the key **J** maps to the value Jane. The keys **B** and **B2** map to the value Bill. You can also think of the keys as forming a set (keySet) and the values as forming a set (valueSet). Each element of keySet maps to a particular element of valueSet, as shown in Figure 7.2. In mathematical set terminology, this is a *many-to-one mapping* (i.e., more than one element of keySet may map to a particular element of valueSet). For example, both keys **B** and **B2** map to the value Bill. This is also an *onto mapping* in that all elements of valueSet have a corresponding member in keySet.

A `Map` can be used to enable efficient storage and retrieval of information in a table. The key is a unique identification value associated with each item stored in a table. As you will see, each key value has an easily computed numeric code value.

EXAMPLE 7.2 When information about an item is stored in a table, the information stored may consist of a unique ID (identification code, which may or may not be a number) as well as descriptive data. The unique ID would be the key, and the rest of the information would represent the value associated with that key. Some examples follow.

Type of Item	Key	Value
University student	Student ID number	Student name, address, major, grade-point average
Customer for online store	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

In the above examples, the student ID number may be assigned by the university, or it may be the student's social security number. The e-mail address is a unique address for each customer, but it is not numeric. Similarly, a part ID could consist of a combination of letters and digits.

In comparing maps to indexed collections, you can think of the keys as selecting the elements of a map, just as indexes select elements in a `List` object. The keys for a map, however, can have arbitrary values (not restricted to 0, 1, etc., as for indexes). As you will see later, an implementation of the `Map` interface should have methods of the form

```
V get(Object key)
V put(K key, V value)
```

The `get` method retrieves the value corresponding to a specified key; the `put` method stores a key–value pair in a map.

The `Map` Hierarchy

Figure 7.3 shows part of the `Map` hierarchy in the Java API. Although not strictly part of the `Collection` hierarchy, the `Map` interface defines a structure that relates elements in one set to elements in another set. The first set, called the *keys*, must implement the `Set` interface; that is, the *keys* are unique. The second set is not strictly a `Set` but an arbitrary `Collection` known as the *values*. These are not required to be unique. The `Map` is a more useful structure than the `Set`. In fact, the Java API implements the `Set` using a `Map`.

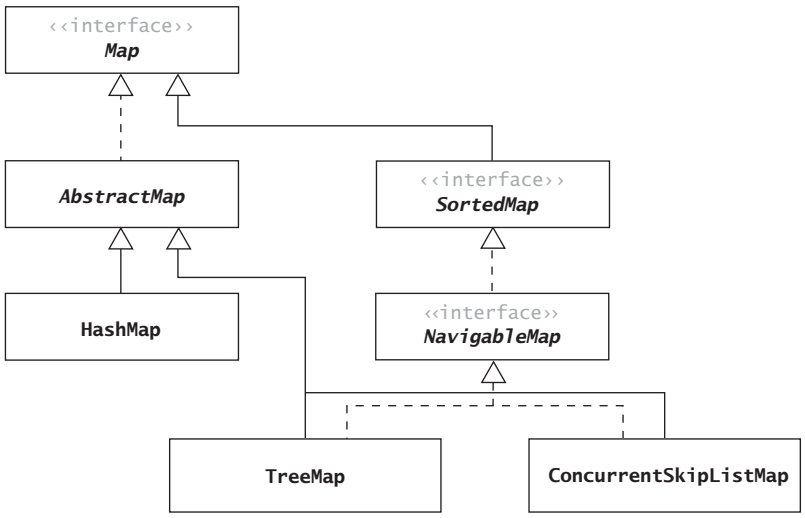


FIGURE 7.3
The `Map` Hierarchy

The `TreeMap` uses a Red–Black binary search tree (discussed in Chapter 9) as its underlying data structure, and the `ConcurrentSkipListMap` uses a skip list (also discussed in Chapter 9) as its underlying data structure. We will focus on the `HashMap` and show how to implement it later in the chapter.

The `Map` Interface

Methods of the `Map` interface (in Java API `java.util`) are shown in Table 7.2. The `put` method either inserts a new mapping or changes the value associated with an existing mapping. The `get` method returns the current value associated with a given key or null if there is none. The `getOrDefault` method returns the provided `default` value instead of null if the key is not present. The `remove` method deletes an existing mapping.

The `getOrDefault` method is a default method, which means that the implementation of this method is defined in the interface. The code for `getOrDefault` is equivalent to the following:

```
default getOrDefault(Object key, V defaultValue) {
    V value = get(key);
    if (value != null) return value;
    return defaultValue;
}
```

TABLE 7.2
Some `java.util.Map<K, V>` Methods

Method	Behavior
`V get(Object key) default`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present
`V getOrDefault(Object key, V default)`	Returns the value associated with the specified key. Returns `default` if the key is not present
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`
`void forEach(BiConsumer<K, V>)`	Performs the action given by the BiConsumer to each entry in the map, binding the key to the first parameter and the value to the second
`int size()`	Returns the number of `key-value` mappings in this map

Both `put` and `remove` return the previous value (or `null`, if there was none) of the mapping that is changed or deleted. There are two type parameters, `K` and `V`, and they represent the data type of the key and value, respectively.

EXAMPLE 7.3 The following statements build a `Map` object that contains the mapping shown in Figure 7.2.

```
Map<String, String> aMap = new HashMap<>();
                           // HashMap implements Map
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

The statement

```
System.out.println("B1 maps to " + aMap.get("B1"));
```

would display `"B1 maps to Bob"`. The statement

```
System.out.println("Bill maps to " + aMap.get("Bill"));
```

would display `"Bill maps to null"` because `"Bill"` is a value, not a key.

EXAMPLE 7.4 In Section 6.5, we used a binary search tree to store an index of words occurring in a term paper. Each data element in the tree was a string consisting of a word followed by a three-digit line number.

Although this is one approach to storing an index, it would be more useful to store each word and all the line numbers for that word as a single index entry. We could do this by storing the index in a `Map` in which each word is a key and its associated value is a list of all the line numbers at which the word occurs. While building the index, each time a word is encountered, its list of line numbers would be retrieved (using the word as a key) and the most recent line number would be appended to this list (a `List<Integer>`). For example, if the word *fire* has

already occurred on lines 4 and 8 and we encounter it again on line 20, the `List<Integer>` associated with *fire* would reference three `Integer` objects wrapping the numbers 4, 8, and 20.

Listing 7.2 shows method `buildIndex` (adapted from `buildIndex` in Listing 6.8). Data field index is a `Map` with key type `String` and value type `List<Integer>`.

```
private Map<String, List<Integer>> index;
```

The statement

```
List<Integer> lines = index.getOrDefault(token, new ArrayList<>());
```

retrieves the value (an `ArrayList<Integer>`) associated with the next `token` or an empty `ArrayList` if this is the first occurrence of `token`. The statements

```
lines.add(lineNum);
index.put(token, lines);            // Store the list.
```

add the new line number to the `ArrayList` `lines` and store it back in the `Map`. In Section 7.5, we show how to display the final index.

LISTING 7.2
Method `buildIndexAllLines`

```
/** Reads each word in a data file and stores it in an index
    along with a list of line numbers where it occurs.
    @post Lowercase form of each word with its line
          number is stored in the index.
    @param scan A Scanner object
 */
public void buildIndex(Scanner scan) {
    int lineNum = 0;     // Line number

    // Keep reading lines until done.
    while (scan.hasNextLine()) {
        lineNum++;

        // Extract each token and store it in index.
        String token;
        while ((token = scan.findInLine(PATTERN))  != null) {
            token = token.toLowerCase();
            // Get the list of line numbers for token
            List<Integer> lines = index.getOrDefault(token, new ArrayList<>());
            lines.add(lineNum);
            index.put(token, lines);    // Store new list of line numbers
        }
        scan.nextline();    // Clear the scan buffer
    }
}
```

EXERCISES FOR SECTION 7.2

SELF-CHECK

1. If you were using a `Map` to store the following lists of items, which data field would you select as the key, and why?
   a. textbook title, author, ISBN (International Standard Book Number), year, publisher
   b. player's name, uniform number, team, position
   c. computer manufacturer, model number, processor, memory, disk size
   d. department, course title, course ID, section number, days, time, room

2. For the `Map index` in Example 7.4, what key–value pairs would be stored for each token in the following data file?

```
this line is first
and line 2 is second
followed by the third line
```

3. Explain the effect of each statement in the following fragment on the index built in Self-Check Exercise 2.

```
lines = index.get("this");
lines = index.get("that");
lines = index.get("line");
lines.add(4);
index.put("is", lines);
```

PROGRAMMING

1. Write statements to create a `Map` object that will store each word occurring in a term paper along with the number of times the word occurs.
2. Write a method `buildWordCounts` (based on `buildIndex`) that builds the `Map` object described in Programming Exercise 1.

7.3 Hash Tables

Before we discuss the details of implementing the required methods of the `Set` and `Map` interfaces, we will describe a data structure, the *hash table,* that can be used as the basis for such an implementation. The goal behind the hash table is to be able to access an entry based on its key value, not its location. In other words, we want to be able to access an element directly through its key value rather than having to determine its location first by searching for the key value in an array. (This is why the `Set` interface has method `contains(obj)` instead of `get(index)`.) Using a hash table enables us to retrieve an item in constant time (expected **O**(1)). We say expected **O**(1) rather than just **O**(1) because there will be some cases where the performance will be much worse than **O**(1) and may even be **O**(*n*), but on the average, we expect that it will be **O**(1). Contrast this with the time required for a linear search of an array, **O**(*n*), and the time to access an element in a binary search tree, **O**(log *n*).

Hash Codes and Index Calculation

The basis of hashing (and hash tables) is to transform the item's key value to an integer value (its *hash code*) that will then be transformed into a table index. Figure 7.4 illustrates this process for a table of size *n*. We discuss how this might be done in the next few examples.

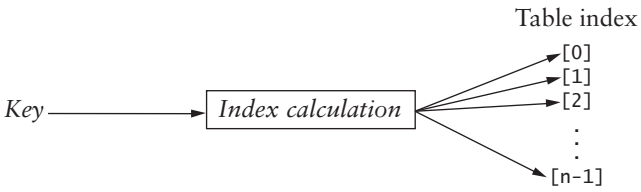


FIGURE 7.4 Index Calculation for a Key