Chapter 15
Maps and Hashing

A map was a fine thing to study when you were disposed to think of something else . . .

— George Eliot, *Middlemarch,* 1874

One of the most useful data structures introduced in Chapter 5 is the **`Map`** class, which provides an association between keys and values. The primary goal of this chapter is to show you how maps can be implemented efficiently using a clever representation called a ***hash table,*** which makes it possible to find the value of a key in constant time. Before doing so, however, it makes sense to start with a less efficient implementation that is not nearly so clever just to make sure that you understand what is involved in implementing the operations required for a map. The following section defines an array-based implementation for the **`Map`** class. The rest of the chapter then looks at various strategies for improving on that simple design.

15.1 Implementing maps using arrays

Figure 15-1 shows a slightly simplified implementation of the **`map.h`** interface, which leaves out four features of the library version of the interface: removing existing map entries, deep copying, selection using square brackets, and the ability to iterate over the keys in a map. Even in its restricted form, the interface is quite useful, and it makes sense to investigate possible implementations of the fundamental operations before adding more sophisticated features to the interface.

The simplest strategy for representing a map is to store each key/value pair in an array. That array, moreover, needs to be dynamic so that it can expand if the number of entries in the map grows beyond the initial allocation. Each key/value pair is stored in a structure with the following definition:

```
struct KeyValuePair {
   KeyType key;
   ValueType value;
};
```

The identifiers **`KeyType`** and **`ValueType`** are the template parameters used to define the **`Map`** class.

By this time, the code necessary to implement a collection class based on a dynamic array should be almost second nature. You have, after all, seen almost exactly the same code in the implementation of the character stack in Chapter 12, the array-based implementation of the editor buffer in Chapter 13, and three different implementations of the linear structures in Chapter 14. In each case, the private section of the class contains a dynamic array, a variable to hold the capacity, a variable to hold the actual number of elements, and a method to expand the capacity of the array when it runs out of space. The **`mappriv.h`** file that defines this structure appears in Figure 15-2.

FIGURE 15-1 Simplified interface for the map abstraction

```
/*
 * File: map.h
 * -----------
 * This interface exports a simplified version of the Map class.
 */

#ifndef _map_h
#define _map_h

/*
 * Class: Map<KeyType,ValueType>
 * -----------------------------
 * The Map class maintains an association between keys and values of
 * the specified types.
 */

template <typename KeyType,typename ValueType>
class Map {

public:

/*
 * Constructor: Map
 * Usage: Map<KeyType,ValueType> map;
 * ----------------------------------
 * Initializes a new empty map that associates keys and values.
 */

   Map();

/*
 * Destructor: ~Map
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage associated with this map.
 */

   ~Map();

/*
 * Method: size
 * Usage: int nEntries = map.size();
 * ---------------------------------
 * Returns the number of entries in this map.
 */

   int size();

/*
 * Method: isEmpty
 * Usage: if (map.isEmpty()) . . .
 * -------------------------------
 * Returns true if this map contains no entries.
 */

   bool isEmpty();
```

☞

FIGURE 15-1 Simplified interface for the map abstraction (continued)

```
/*
 * Method: clear
 * Usage: map.clear();
 * -------------------
 * Removes all entries from this map.
 */

   void clear();

/*
 * Method: put
 * Usage: map.put(key, value);
 * ---------------------------
 * Associates key with value in this map.  Any previous value associated
 * with key is replaced by the new value.
 */

   void put(KeyType key, ValueType value);

/*
 * Method: get
 * Usage: ValueType value = map.get(key);
 * --------------------------------------
 * Returns the value associated with key in this map.  If key is not
 * found, the get method signals an error.
 */

   ValueType get(KeyType key);

/*
 * Method: containsKey
 * Usage: if (map.containsKey(key)) . . .
 * --------------------------------------
 * Returns true if there is an entry for key in this map.
 */

   bool containsKey(KeyType key);

#include "mappriv.h"

};

#include "mapimpl.cpp"

#endif
```

FIGURE 15-2 Contents of the `mappriv.h` file for the array-based map

```
/*
 * File: mappriv.h
 * ----------------
 * This file defines the private section of the Map class using an
 * array-based representation.
 */

private:

/*
 * Type: KeyValuePair
 * -------------------
 * This structure combines a key and a value into a single unit.
 */

   struct KeyValuePair {
      KeyType key;
      ValueType value;
   };

/* Instance variables */

   KeyValuePair *array;     /* Dynamic array of key-value pairs  */
   int capacity;            /* The capacity of the dynamic array */
   int count;               /* The current number of entries     */

/* Private method prototypes */

   void expandCapacity();
   int findKey(KeyType key);
```

One possible implementation for the array-based version of the **`Map`** class appears in Figure 15-3. This implementation uses the linear-search algorithm to find an existing key in the map, as expressed in the private method **`findKey`**, which is called by each of the methods **`get`**, **`put`**, and **`containsKey`**:

```
template <typename KeyType,typename ValueType>
int Map<KeyType,ValueType>::findKey(KeyType key) {
   for (int i = 0; i < count; i++) {
      if (array[i].key == key) return i;
   }
   return -1;
}
```

This method returns the index at which a particular key appears in the list of keys already included in the array; if the key does not appear, **`findKey`** returns −1. The use of the linear-search algorithm means that the **`get`**, **`put`**, and **`containsKey`** methods are all $O(N)$.

FIGURE 15-3 Contents of the `mapimpl.cpp` file for the array-based map

```
/*
 * File: mapimpl.cpp
 * -----------------
 * This file contains the implementation of the map.h interface.
 * Because of the way C++ compiles templates, this code must be
 * available to the compiler when it reads the header file.
 */

#ifdef _map_h

#include "error.h"

/* Constants */

const int INITIAL_CAPACITY = 10;  /* Initial capacity for the dynamic array */

/*
 * Implementation notes
 * --------------------
 * The implementation of the methods in this class are short enough that
 * they should be entirely straightforward, particularly given their
 * similarity to the other classes that use dynamic arrays as their
 * internal representation.
 */

template <typename KeyType,typename ValueType>
Map<KeyType,ValueType>::Map() {
   capacity = INITIAL_CAPACITY;
   count = 0;
   array = new KeyValuePair[capacity];
}

template <typename KeyType,typename ValueType>
Map<KeyType,ValueType>::~Map() {
   delete[] array;
}

template <typename KeyType,typename ValueType>
int Map<KeyType,ValueType>::size() {
   return count;
}

template <typename KeyType,typename ValueType>
bool Map<KeyType,ValueType>::isEmpty() {
   return count == 0;
}

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::clear() {
   count = 0;
}
```

☞

FIGURE 15-3 Contents of the `mapimpl.cpp` file for the array-based map (continued)

```
template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::put(KeyType key, ValueType value) {
   int index = findKey(key);
   if (index == -1) {
     if (count == capacity) expandCapacity();
     index = count++;
     array[index].key = key;
   }
   array[index].value = value;
}

template <typename KeyType,typename ValueType>
ValueType Map<KeyType,ValueType>::get(KeyType key) {
   int index = findKey(key);
   if (index == -1) error("get: No value for key");
   return array[index].value;
}

template <typename KeyType,typename ValueType>
bool Map<KeyType,ValueType>::containsKey(KeyType key) {
   return findKey(key) != -1;
}

/*
 * Private method: expandCapacity
 * ------------------------------
 * Doubles the capacity of the array containing the key/value pairs.
 */

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::expandCapacity() {
   KeyValuePair *oldArray = array;
   capacity *= 2;
   array = new KeyValuePair[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}

/*
 * Private method: findKey
 * -----------------------
 * Returns the index at which the key appears, or -1 if it is not found.
 */

template <typename KeyType,typename ValueType>
int Map<KeyType,ValueType>::findKey(KeyType key) {
   for (int i = 0; i < count; i++) {
      if (array[i].key == key) return i;
   }
   return -1;
}

#endif
```

It is possible to improve the performance of the **get** and **containsKey** methods by keeping the keys in sorted order and applying the binary-search algorithm, which was introduced in section 7.5. Binary search reduces the search time to $O(\log N)$, which represents a dramatic improvement over the $O(N)$ time that linear search requires. Unfortunately, there is no obvious way to apply that same optimization to the **put** method. Although it is certainly possible to check whether the key already exists in the map—and even to determine exactly where a new key needs to be added—in $O(\log N)$ time, inserting the new key/value pair at that position requires shifting every subsequent entry forward. That operation requires $O(N)$ time.

15.2 Lookup tables

The map abstraction comes up so frequently in programming that it is worth putting some effort into improving its performance. The implementation plan described in the preceding section—storing the key/value pairs in sorted order in a dynamic array—offers $O(\log N)$ performance for the **get** operation and $O(N)$ performance for the **put** operation. It is possible to do much better.

When you are trying to optimize the performance of a data structure, it is often helpful to identify performance enhancements that apply to some special case and then look for ways to apply those algorithmic improvements more generally. This section introduces a specific problem for which it is easy to find constant-time implementations of the **get** and **put** operations. It then goes on to explore how a similar technique might help in a more general context.

In 1963, the United States Postal Service introduced a set of two-letter codes for the individual states, districts, and territories of the United States. The codes for the 50 states appear in Figure 15-4. Although you might also want to translate in the opposite direction as well, this section considers only the problem of translating two-letter codes into state names. The data structure that you choose must therefore be able to represent a map from two-letter abbreviations to state names.

FIGURE 15-4 USPS abbreviations for the 50 states

AK	Alaska	HI	Hawaii	ME	Maine	NJ	New Jersey	SD	South Dakota
AL	Alabama	IA	Iowa	MI	Michigan	NM	New Mexico	TN	Tennessee
AR	Arkansas	ID	Idaho	MN	Minnesota	NV	Nevada	TX	Texas
AZ	Arizona	IL	Illinois	MO	Missouri	NY	New York	UT	Utah
CA	California	IN	Indiana	MS	Mississippi	OH	Ohio	VA	Virginia
CO	Colorado	KS	Kansas	MT	Montana	OK	Oklahoma	VT	Vermont
CT	Connecticut	KY	Kentucky	NC	North Carolina	OR	Oregon	WA	Washington
DE	Delaware	LA	Louisiana	ND	North Dakota	PA	Pennsylvania	WI	Wisconsin
FL	Florida	MA	Massachusetts	NE	Nebraska	RI	Rhode Island	WV	West Virginia
GA	Georgia	MD	Maryland	NH	New Hampshire	SC	South Carolina	WY	Wyoming

You could, of course, encode the translation table in a **`Map<string,string>`**. If you look at this problem strictly from the client's point of view, the details of the implementation aren't particularly important. In this chapter, however, the goal is to identify new implementation strategies for maps that operate more efficiently. In this example, the important question to ask is whether the fact that the keys are two-letter strings makes it possible to implement this association more efficiently than the array-based strategy from the preceding section.

As it turns out, the two-character restriction on the keys makes it easy to reduce the complexity of the lookup operation to constant time. All you need to do is store the state names in a two-dimensional grid in which the letters in the state abbreviation are used to compute the row and column indices. To select a particular element from the grid, all you have to do is break the state abbreviation down into the two characters it contains, subtract the ASCII value of `'A'` from each character to get an index between 0 and 25, and then use these two indices to select a row and column. Thus, given a grid that has already been initialized to contain the state abbreviations as shown in Figure 15-5, you can use the following function to convert an abbreviation to the corresponding state name:

```
string getStateName(string key, Grid<string> & grid) {
   char row = key[0] - 'A';
   char col = key[1] - 'A';
   if (!grid.inBounds(row, col) || grid[row][col] == "") {
      error("No state name for " + abbr);
   }
   return grid[row][col];
}
```

This function contains nothing that looks like the traditional process of searching an array. What happens instead is that the function performs simple arithmetic on the character codes and then looks up the answer in a grid. There are no loops in the implementation or any code that depends at all on the number of keys in the map. Looking up an abbreviation in the table must therefore run in $O(1)$ time.

The grid used in the **`getStateName`** function is an example of a ***lookup table,*** which is a programming structure that makes it possible to obtain a desired value simply by computing the appropriate index in a table, which is typically an array or a grid. The reason that lookup tables are so efficient is that the key tells you immediately where to look for the answer. In the current application, however, the organization of the table depends on the fact that the keys always consist of two uppercase letters. If the keys could be arbitrary strings—as they are in the library version of the **`Map`** class—the lookup-table strategy no longer applies, at least in its current form. The critical question is whether it is possible to generalize this strategy so that it applies to the more general case.

FIGURE 15-5 First nine columns of the state lookup table

	A	B	C	D	E	F	G	H	I	. . .
A										0
B										1
C	California									2
D					Delaware					3
E										4
F										5
G	Georgia									6
H									Hawaii	7
I	Iowa			Idaho						8
J										9
K										10
L	Louisiana									11
M	Massachusetts			Maryland	Maine				Michigan	12
N			North Carolina	North Dakota	Nebraska			New Hampshire		13
O								Ohio		14
P	Pennsylvania									15
Q										16
R									Rhode Island	17
S			South Carolina	South Dakota						18
T										19
U										20
V	Virginia									21
W	Washington								Wisconsin	22
X										23
Y										24
Z										25
	0	1	2	3	4	5	6	7	8	

If you think about how this question applies to real-life applications, you may discover that you in fact use something akin to the lookup-table strategy when you search for words in a dictionary. If you were to apply the array-based map strategy to the dictionary-lookup problem, you would start at the first entry, go on to the second, and then the third, until you found the word. No one, of course, would apply this algorithm in a real dictionary of any significant size. But it is also unlikely that you would apply the $O(\log N)$ binary search algorithm, which consists of opening the dictionary exactly at the middle, deciding whether the word you're searching for appears in the first or second half, and then repeatedly applying this algorithm to smaller and smaller parts of the dictionary. In all likelihood, you would take advantage of the fact that most dictionaries have thumb tabs along the side that indicate where the entries for each letter appear. You look for words starting with *A* in the *A* section, words starting with *B* in the *B* section, and so on.

These thumb tabs represent a lookup-table that gets you to the right section, thereby reducing the number of words through which you need to search.

At least for those maps that use strings as their key type, it is possible to apply the same strategy to the map abstraction. In this type of map, each key begins with some character value, although that character is not necessarily a letter. If you want to simulate the strategy of using thumb tabs for every possible first character, you can divide the map into 256 independent lists of key/value pairs—one for each starting character. Whenever the client calls `put` or `get` with some key, the code can choose the appropriate list on the basis of the first character. If the characters used to form keys are uniformly distributed, this strategy would reduce the average search time by a factor of 256.

Unfortunately, keys in a map—like words in a dictionary—are not uniformly distributed. In the dictionary case, for example, many more words begin with *C* than with *X*. If you use a map in an application, it is likely that most of the 256 possible first characters never appear at all. As a result, some of the lists will remain empty, while others become quite long. The increase in efficiency you get by applying the first-character strategy therefore depends on how common the first character in the key happens to be.

On the other hand, there is no reason that you have to use only the first character of the key as you try to optimize the performance of the map. The first-character strategy is simply the closest analogue to what you do with a physical dictionary. What you need is a strategy in which the value of the key tells you where to find the location of the value, as it does in a lookup table. That idea is most elegantly implemented using a technique called *hashing,* which is described in the following section.

15.3 Hashing

The best way to improve the efficiency of the map implementation is to come up with a way of using the key to determine, at least fairly closely, where to look for the corresponding value. Choosing any obvious property of the key, such as its first character or even its first two characters, runs into the problem that keys are not equally distributed with respect to that property.

Given that you are using a computer, however, there is no reason that the property you use to locate the key has to be something easy for a *human* to figure out. To maintain the efficiency of the implementation, the only thing that matters is whether the property is easy for a *computer* to figure out. Since computers are better at computation than humans are, allowing for algorithmic computation opens a much wider range of possibilities.

The computational strategy called ***hashing*** operates as follows:

1. Select a function *f* that transforms a key into an integer value. That value is called the ***hash code*** of that key. The function that computes the hash code is called, naturally enough, a ***hash function.*** An implementation of the map abstraction that uses this strategy is conventionally called a ***hash table.***
2. Use the hash code for a key as the starting point as you search for a matching key in the table. You might, for example, use the hash code as an index into an array of lists, each of which holds all the key/value pairs that correspond to that hash code. To find an exact match, all you need to do is search through the list of key/value pairs in that list. As long as the hash function always returns the same value for any particular key, you know that the value, if it exists, must be in that list. Suppose, for example, that you call **`put`** on a key whose hash code is 17. According to the basic hashing algorithm, the **`put`** method must store that key and its value on list #17. If you later call **`get`** with that same key, the hash code you get will be the same, which means that the desired key/value pair must be on list #17, if it exists in the hash table at all.

The `hashmap.h` interface

Before diving into the details of the hash table implementation, it is important to point out that the Standard Template Library does not actually use hash tables to represent maps. In C++, the standard libraries adopt a different strategy that uses as its underlying representation a structure called a *binary search tree,* which you will learn about in Chapter 16. Using binary search trees to implement maps is somewhat less efficient than using hash tables but has the advantage of making it possible to iterate through the keys in order.

Many modern languages allow programmers to choose a representation for maps to match the needs of each application. The Java libraries, for example, export two different classes that implement the map idea. The **`HashMap`** class uses a hash table as its underlying representation; the **`TreeMap`** class uses a binary search tree. Clients use the **`HashMap`** class when they require the best possible performance and the **`TreeMap`** class whenever the order of keys is important.

The Stanford libraries adopt a similar strategy. To maintain consistency with the STL version of maps, the **`map.h`** interface uses a tree-based implementation. For applications that require better performance, the Stanford libraries also export a separate **`HashMap`** class through the **`hashmap.h`** interface. The methods exported by the two classes are exactly the same. The only difference visible to the client is the order of iteration, which is unpredictable in the **`HashMap`** case.

The decision of the C++ designers not to include hash maps in the Standard Template Library makes a certain amount of sense. The performance advantage of hashing is not that large in practice; moreover, having a single map abstraction reduces the conceptual complexity of the libraries. As a student of computer science, however, it is essential for you to learn about hashing as you study algorithms and data structures. For one thing, it is extremely important in practice, and there are many languages for which the library designers made the opposite decision by including hashing as the only option. For another, hashing surely ranks as one of the most elegant algorithmic discoveries from the early history of the computing field, even though no one is sure who actually came up with the idea.

Designing the data structure

The first step toward implementing a hash table is to design the data structure. As the preceding section suggests, each hash code serves as an index into an array of linked lists. Each list is traditionally called a ***bucket.*** Whenever you call **`put`** or **`get`**, you select the appropriate bucket by applying the hash function to the key. That function gives you an integer, which is likely to be larger than the number of buckets you have in your hash table. You can, however, convert an arbitrarily large nonnegative hash code into a bucket number by dividing the hash code by the number of buckets and taking the remainder. Thus, if the number of buckets is stored in the variable **`nBuckets`** and the function **`hashCode`** computes the hash code for a given key, you can use the following line to compute the bucket number:

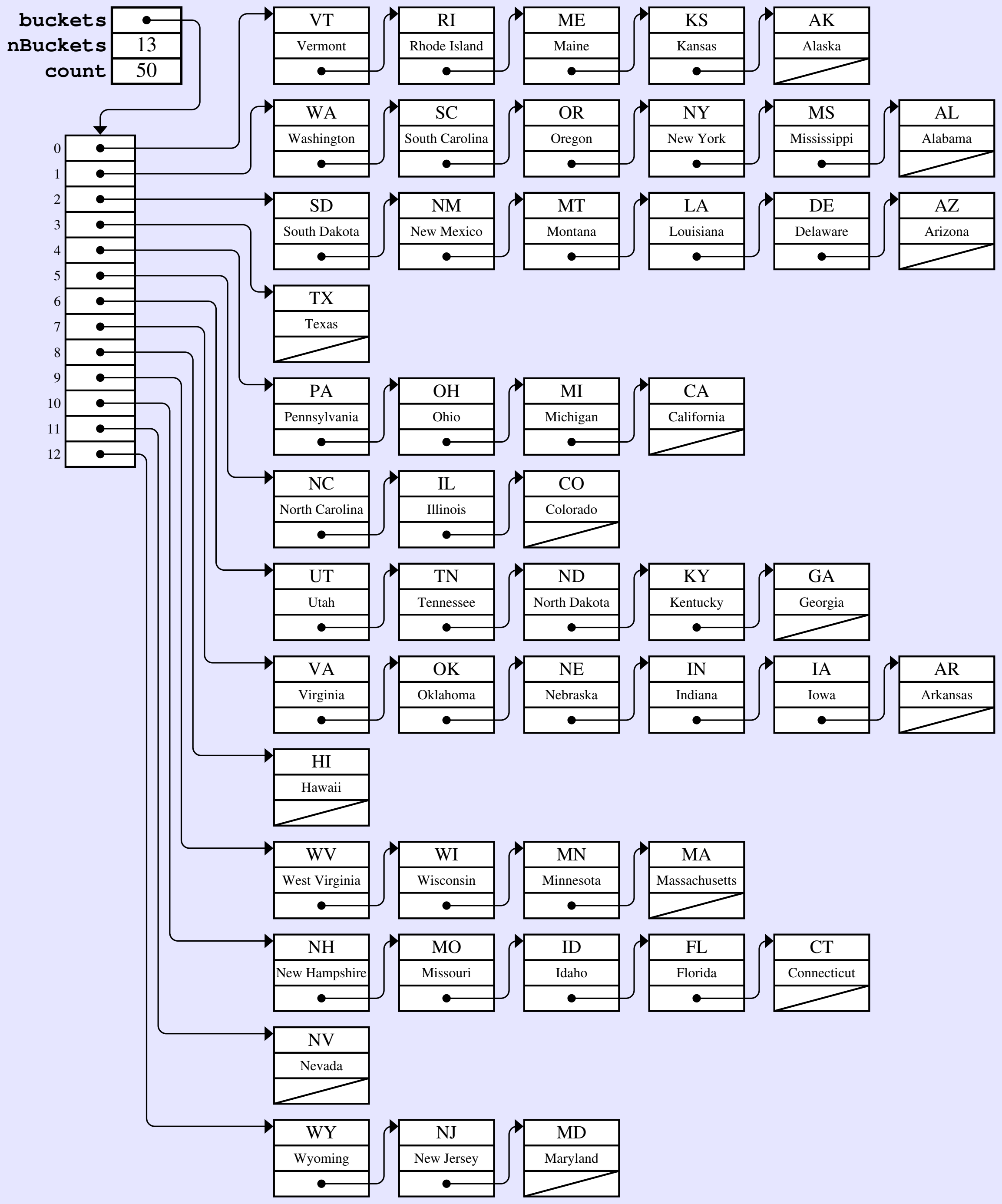
```
int bucket = hashCode(key) % nBuckets;
```

A bucket number represents an index into an array, each of whose elements is a pointer to the first cell in a list of key/value pairs. Colloquially, computer scientists say that a key ***hashes to a bucket*** if the hash function applied to the key returns that bucket number. Thus, the common property that links all the keys in a single linked list is that they all hash to the same bucket. Having two or more different keys hash to the same bucket is called ***collision.***

To help you visualize the representation of a hash table, Figure 15-6 shows how the abbreviations for the 50 states fit into a table with 13 buckets. The abbreviations **`AK`**, **`KS`**, **`ME`**, **`RI`**, and **`VT`** all hash to bucket #0; **`AL`**, **`MS`**, **`NY`**, **`OR`**, **`SC`**, and **`WA`** all hash to bucket #1; and so on. By distributing the keys among the buckets, the **`get`** and **`put`** functions have a much shorter list to search.

The private section of the **`HashMap`** class must contain the instance variables and type definitions necessary to represent the data structure depicted in Figure 15-6. The private section of the object contains three instance variables: a dynamic array containing linked lists of the individual entries (**`buckets`**), the size of that array (**`nBuckets`**), and the number of entries in the hash table (**`count`**).

FIGURE 15-6 Hash table containing the state abbreviations



Each of the elements in the **buckets** array is a linked list of the key/value pairs that hash to that bucket. The cells in the chain are similar to those you have seen in the earlier linked list examples except for the fact that each cell contains both a key and a value. The definition for the **Cell** structure for this list therefore looks like this:

```
struct Cell {
   KeyType key;
   ValueType value;
   Cell *link;
};
```

The definition of this structure and the declarations of the instance variables appear in the **hashmappriv.h** file shown in Figure 15-7.

If you look at the declaration of the instance variable **buckets** in Figure 15-7, the syntax may initially seem confusing. Up to now, the dynamic arrays you've created for the other collection classes have been declared as pointers to the base type. Here, the declaration reads

```
Cell **buckets;
```

As in the earlier examples, **buckets** is a dynamic array and is therefore represented in C++ as a pointer to the initial element in the array. Each element, moreover, is a pointer to the first cell in the linked-list chain of key/value pairs. The **buckets** variable is therefore a pointer to a pointer to a cell, which accounts for the double star.

In addition to the definition of the **Cell** structure and the instance variables, the **hashmappriv.h** file defines a private method called **findCell** that searches for a key in a linked-list chain. This method is used in the **get**, **put**, and **containsKey** methods to avoid duplicating the common code. The code for **findCell** fits the standard pattern for looping through the cells of a linked list. The only unusual thing about **findCell** is that the implementation appears in the **hashmappriv.h** file. Up to now, this book has imposed a rigid separation of prototypes and implementations between the **.h** and the **.cpp** files. For template classes, that separation is less important because the compiler needs access to the complete implementation in any case. The typical client won't look at either the **hashmappriv.h** or the **hashmapimpl.cpp** file, and it therefore doesn't enhance the principle of information hiding to enforce that separation.

Particularly given the fact that the code to process the linked list is already provided in the **findCell** method, the implementation of the **HashMap** class is straightforward. The code appears in the **hashmapimpl.cpp** file in Figure 15-8.

FIGURE 15-7 Contents of the `hashmappriv.h` file

```
/*
 * File: hashmappriv.h
 * -------------------
 * This file contains the private section of the hashmap.h interface.
 */

/*
 * Notes on the representation
 * ---------------------------
 * The HashMap class is represented using a hash table that keeps the
 * key/value pairs in an array of buckets, where each bucket is a
 * linked list of elements that share the same hash code.  If two or
 * more keys have the same hash code (which is called a "collision"),
 * each of those keys will be on the same list.
 */

private:

/* Type definition for cells in the bucket chain */

   struct Cell {
      KeyType key;
      ValueType value;
      Cell *link;
   };

/* Instance variables */

   Cell **buckets;     /* Dynamic array of pointers to cells */
   int nBuckets;       /* The number of buckets in the array */
   int count;          /* The number of entries in the map   */

/* Private methods */

/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * ----------------------------------------
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell containing
 * the matching key.  If no match is found, the function returns NULL.
 * Given that this method is already embedded in a file marked private,
 * it makes sense to implement it here rather than doing so in the
 * hashmapimpl.cpp file.
 */

   Cell *findCell(int bucket, ValueType key) {
      Cell *cp = buckets[bucket];
      while (cp != NULL && key != cp->key) {
         cp = cp->link;
      }
      return cp;
   }
```

FIGURE 15-8 Implementation of the `HashMap` class

```
/*
 * File: hashmapimpl.cpp
 * ---------------------
 * This file contains the private section of the hashmap.cpp interface.
 * Because of the way C++ compiles templates, this code must be
 * available to the compiler when it reads the header file.
 */

#ifdef _hashmap_h

#include "error.h"

/* Constant definitions */

const int INITIAL_BUCKET_COUNT = 101;

/*
 * Implementation notes: HashMap constructor and destructor
 * --------------------------------------------------------
 * The constructor allocates the array of buckets and initializes
 * each bucket to the empty list.  The destructor must free the memory,
 * but can do so by calling clear.
 */

template <typename KeyType,typename ValueType>
HashMap<KeyType,ValueType>::HashMap() {
   nBuckets = INITIAL_BUCKET_COUNT;
   buckets = new Cell*[nBuckets];
   for (int i = 0; i < nBuckets; i++) {
      buckets[i] = NULL;
   }
   count = 0;
}

template <typename KeyType,typename ValueType>
HashMap<KeyType,ValueType>::~HashMap() {
   clear();
}

/*
 * Implementation notes: size, isEmpty
 * -----------------------------------
 * These methods are simple because the structure stores the entry count.
 */

template <typename KeyType,typename ValueType>
int HashMap<KeyType,ValueType>::size() {
   return count;
}

template <typename KeyType,typename ValueType>
bool HashMap<KeyType,ValueType>::isEmpty() {
   return count == 0;
}
```

☞

FIGURE 15-8 Implementation of the `HashMap` class (continued)

```
/*
 * Implementation notes: clear
 * ---------------------------
 * This method frees all the cells to avoid a memory leak.
 */

template <typename KeyType,typename ValueType>
void HashMap<KeyType,ValueType>::clear() {
   for (int i = 0; i < nBuckets; i++) {
      Cell *cp = buckets[i];
      while (cp != NULL) {
         Cell *oldCell = cp;
         cp = cp->link;
         delete oldCell;
      }
   }
   count = 0;
}

/*
 * Implementation notes: put, get, containsKey
 * -------------------------------------------
 * These methods rely on the findCell method from the hashmappriv.h file.
 */

template <typename KeyType,typename ValueType>
void HashMap<KeyType,ValueType>::put(KeyType key, ValueType value) {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   if (cp == NULL) {
      cp = new Cell;
      cp->key = key;
      cp->link = buckets[bucket];
      buckets[bucket] = cp;
      count++;
   }
   cp->value = value;
}

template <typename KeyType,typename ValueType>
ValueType HashMap<KeyType,ValueType>::get(KeyType key) {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   if (cp == NULL) error("get: no value for key");
   return cp->value;
}

template <typename KeyType,typename ValueType>
bool HashMap<KeyType,ValueType>::containsKey(KeyType key) {
   int bucket = hashCode(key) % nBuckets;
   return findCell(bucket, key) != NULL;
}

#endif
```

Defining a hash function for strings

Although Figure 15-8 defines the implementations of the methods in the **`HashMap`** class, it does not include the **`hashCode`** function that sits at the heart of the hashing algorithm. The **`hashCode`** function—or, more correctly, the collection of overloaded **`hashCode`** functions for each possible key type—is not defined as a method in the **`HashMap`** class but instead appears as a free function. For that reason, the prototypes and implementations for those functions will be split between the **`hashmap.h`** and **`hashmap.cpp`** files, just as they have been in the libraries you have been writing since Chapter 2.

Although they are typically short, hash functions tend to be subtle and difficult to understand on a first reading. The hash function for strings used in the **`HashMap`** class, which was developed by Daniel J. Bernstein, Professor of Mathematics at the University of Illinois at Chicago, looks like this:

```
const int HASH_SEED = 5381;
const int HASH_MULTIPLIER = 33;
const int HASH_MASK = unsigned(-1) >> 1;

int hashCode(string str) {
   unsigned hash = HASH_SEED;
   int n = str.length();
   for (int i = 0; i < n; i++) {
      hash = HASH_MULTIPLIER * hash + str[i];
   }
   return int(hash & HASH_MASK);
}
```

The hash function used for strings in any particular library will presumably not look exactly like this one, but most implementations will have much the same structure. In this implementation, the code iterates through each character in the key, updating a value stored in the local variable **`hash`**, which is declared as an **`unsigned`** integer and initialized to the seemingly random constant 5381. On each loop cycle, the **`hashCode`** function multiplies the previous value of **`hash`** by a constant called **`HASH_MULTIPLIER`** and then adds the ASCII value of the current character. At the end of the loop, the result is not simply the value of **`hash`** but instead computed by means of the rather odd-looking expression

```
int(hash & HASH_MASK)
```

Given the amount of confusing code present in such a short function, you should feel perfectly justified in deciding that the intricacies of the **`hashCode`** function are not worth understanding in detail. The point of all the complexity is to ensure that the results of the **`hashCode`** function are as unpredictable as possible given a particular set of keys. The details as to how the function does so, while interesting in their own right as a theoretical question, are not of immediate concern to clients of the **HashMap** class. Choosing the right **hashCode** function, however, can have a significant effect on the efficiency of the implementation

To see how the design of the **hashCode** function can affect efficiency, consider what might happen if you use the following, much simpler implementation:

```
int hashCode(string str) {
   int hash = 0;
   int n = str.length();
   for (int i = 0; i < n; i++) {
      hash += str[i];
   }
   return hash;
}
```

This code is far more understandable. All it does is add up the ASCII codes for the characters in the string, which will be a nonnegative integer unless the string is hugely long. Beyond the fact that long strings might cause integer overflow and result in negative results (which justifies the inclusion of the bug symbol), writing **hashCode** in this way is likely to cause collisions in the table if the keys happen to fall into certain patterns. The strategy of adding the ASCII values means that any keys whose letters are permutations of each other would collide. Thus, **cat** and **act** would hash to the same bucket. So would the keys **a3**, **b2**, and **c1**. If you were using this hash table in the context of a compiler, variable names that fit such patterns would all end up hashing to the same bucket.

At the cost of making the code for the **hashCode** function more obscure, you can reduce the likelihood that similar keys will collide. Figuring out how to design such a function, however, requires a reasonably advanced knowledge of computer science theory. Most implementations of the **hashCode** function use techniques that are similar to those for generating pseudorandom numbers, as discussed in Chapter 2. In both domains, it is important that the results are hard to predict. In the hash table, the consequence of this unpredictability is that keys chosen by a programmer are unlikely to exhibit any higher level of collision than one would expect by random chance.

Even though careful choice of a hash function can reduce the number of collisions and thereby improve performance, it is important to recognize that the *correctness* of the algorithm is not affected by the collision rate. The only requirement is that the hash function has to return a nonnegative integer. If it does, the map implementation will still work even if the hash function always returns 0. In that case, every key would end up in the chain attached to bucket #0. Programs that use such a hash function would run slowly because every key is linked into the same chain, but they would nonetheless continue to give the correct results.

Determining the number of buckets

Although the design of the hash function is important, it is clear that the likelihood of collision also depends on the number of buckets. If the number is small, collisions occur more frequently. In particular, if there are more entries in the hash table than buckets, collisions are inevitable. Collisions affect the performance of the hash table strategy because they force **`put`** and **`get`** to search through longer chains. As the hash table fills up, the number of collisions rises, which in turn reduces the performance of the hash table.

It is important to remember that the goal of using a hash table is to optimize the **`put`** and **`get`** methods so that they run in constant time, at least in the average case. To achieve this goal, it is important that the linked-list chains emerging from each bucket remain short. To do so, you need to keep the number of buckets relatively large in comparison to the number of entries. Assuming that the hash function does a good job of distributing the keys evenly among the buckets, the average length of each bucket chain is given by the formula

$$\lambda = \frac{N_{\text{entries}}}{N_{\text{buckets}}}$$

For example, if the total number of entries in the table is three times the number of buckets, the average chain will contain three entries, which in turn means that three string comparisons will be required, on average, to find a key. The value λ is called the ***load factor*** of the hash table

For good performance, you want to make sure that the value of λ remains small. Although the mathematical details are beyond the scope of this text, maintaining a load factor of 0.7 or less means that the average cost of looking up a key in a **`HashMap`** is $O(1)$. Smaller load factors imply that there will be lots of empty buckets in the hash table array, which wastes a certain amount of space. Hash tables represent a good example of a time-space tradeoff, a concept introduced in Chapter 13. By increasing the amount of space used by the hash table, you can improve performance, but there is little advantage in reducing the load factor below the 0.7 threshold.

Unless the **`HashMap`** class is engineered for a particular application in which the number of keys is known in advance, there is no way to choose a value for **`nBuckets`** that works well for all clients. If a client keeps entering more and more entries into a map, the performance will eventually decline. If you want to maintain good performance, the best approach is to allow the implementation to increase the number of buckets dynamically. For example, you can design the implementation so that it allocates a larger hash table if the load factor in the table ever reaches a certain threshold. Unfortunately, if you increase the number of buckets, the bucket numbers all change, which means that the code to expand the table must reenter every key from the old table into the new one. This process is called ***rehashing.*** Although rehashing can be time-consuming, it is performed infrequently and therefore has minimal impact on the overall running time of the application. You will have a chance to implement the rehashing strategy in exercise 4.

Hash functions for other types

Although strings are by far the most common key type in practice, the use of templates in the **`HashMap`** definition makes it possible to use other key types as well. The code to implement the **`HashMap`** class imposes two requirements on the key type:

1. The key type must support the **`==`** comparison operator so that the code can tell whether two keys are identical.
2. The code for the **`HashMap`** class must have access to an overloaded version of the **`hashCode`** function that produces a nonnegative integer for every value of the key type. For the common types like **`string`** and the primitive types, those functions are exported by the **`hashmap.h`** interface itself. For types that are specific to an application, the client must supply this function.

In many cases, these functions can be extremely simple. For example, the **`hashCode`** function for integers is simply

```
int hashCode(int key) {
   return key & HASH_MASK;
}
```

The **`HASH_MASK`** constant is the same as the one defined in the section on the hash function for strings and consists of a word whose internal representation contains a **`1`** in every bit position except the sign bit, which is **`0`**. The **`&`** operator, which is covered in more detail in Chapter 18, has the effect of removing the sign bit from **`key`**, which ensures that the value of **`hashCode`** cannot be negative.

Writing good hash functions for compound types requires a certain amount of mathematical sophistication to ensure that the hash codes are distributed uniformly across the space. There is, however, a simple expedient that you can use to produce a reasonable hash function for any type that exports a **`toString`** method. All you have to do is convert the value to a string and then use the string version of **`hashCode`** to deliver the result. Using this approach, you could write a **`hashCode`** function for the **`Rational`** class like this:

```
int hashCode(Rational r) {
   return hashCode(r.toString());
}
```

Computing this function requires more execution time than performing arithmetic operations on the numerator and denominator, but the function is much easier to code.

Summary

The focus of this chapter has been a variety of strategies for implementing the basic operations provided by the **`Map`** class, even though this chapter stops short of introducing the library implementation of **`Map`**, which requires the material in Chapter 16.

Important points in this chapter include:

- It is possible to implement the basic map operations by storing key/value pairs in a dynamic array. Keeping the array in sorted order makes it possible for **`get`** to run in $O(\log N)$ time, even though **`put`** remains $O(N)$.
- Specific applications may make it possible to implement map operations using a lookup table in which both **`get`** and **`put`** run in $O(1)$ time.
- Maps can be implemented very efficiently using a strategy called *hashing,* in which keys are converted to an integer that determines where the implementation should look for the result.
- A common implementation of the hashing algorithm is to allocate a dynamic array of *buckets,* each of which contains a linked list of the keys that hash to that bucket. As long as the ratio of the number of entries to the number of buckets does not exceed about 0.7, the **`get`** and **`put`** methods operate in $O(1)$ time on average. Maintaining this performance as the number of entries grows requires periodic *rehashing* to increase the number of buckets.
- The detailed design of a hash function is subtle and requires mathematical analysis to achieve optimum performance. Even so, any hash function that delivers nonnegative integer values will produce correct results.

Review questions

1. For the array-based implementation of maps, what algorithmic strategy does the chapter suggest for reducing the cost of the **`get`** method to $O(\log N)$ time?

2. If you implement the strategy suggested in the preceding question, why does the **`put`** method still require $O(N)$ time?

3. What is a *lookup table?* In what cases is this strategy appropriate?

4. What disadvantages would you expect from using the ASCII value of the first character in a key as its hash code?

5. What is meant by the term *bucket* in the implementation of a hash table?

6. What is a *collision?*

7. Explain the operation of the **findCell** function in the **hashmappriv.h** file shown in Figure 15-7.

8. The **hashCode** function for strings presented in the text has a structure similar to that of a random-number generator. If you took that similarity too literally, however, you might be tempted to write the following **hash** function:

   ```
   int hashCode(string str) {
      return randomInteger(0, HASH_MASK);
   }
   ```

   Why would this approach fail?

9. Would the **HashMap** class still operate correctly if you supplied the following **hashCode** function:

   ```
   int hashCode(string str) {
      return 42;
   }
   ```

10. What time-space tradeoff arises in the implementation of a hash table?

11. What is meant by the term *load factor?*

12. What is the approximate threshold for the load factor that ensures that the average performance of the **HashMap** class will remain $O(1)$?

13. What is meant by the term *rehashing?*

14. What two operations must every key type implement?

15. Suppose that you wanted to use the **Point** class from Chapter 6 as a key type in a **HashMap**. What simple strategy does the chapter offer for implementing the necessary **hashCode** function?

Exercises

1. Although it presumably made mathematics more difficult, the Romans wrote numbers using letters to stand for various multiples of 5 and 10. The characters used to encode Roman numerals have the following values:

I	→	1
V	→	5
X	→	10
L	→	50
C	→	100
D	→	500
M	→	1000

Design a lookup table that makes it possible to determine the value of each letter in a single array selection. Use this table to implement a function

```
int romanToDecimal(string str);
```

that translates a string containing a Roman numeral into its numeric form. To compute the value of a Roman numeral, you simply add the values corresponding to each letter, subject to one exception: If the value of a letter is less than the letter that follows it, that value should be subtracted from the total instead of added. For example, the Roman numeral string

MCMLXIX

corresponds to

$$1000 - 100 + 1000 + 50 + 10 - 1 + 10$$

or 1969. The **C** and the **I** are subtracted rather than added because each of those letters is followed by a letter with a larger value.

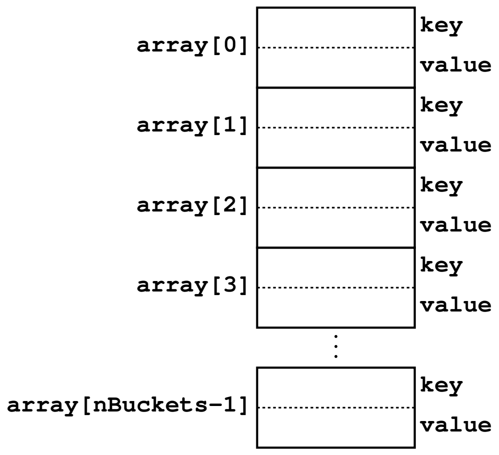
2. Modify the code in Figure 15-3 so that **put** always keeps the keys in sorted order in the array. Change the implementation of the private **findKey** method so that it uses binary search to find the key in $O(\log N)$ time.

3. The implementations of the **Map** and **HashMap** abstractions in this chapter do not include the **remove** method, which deletes a key from the table. Extend the implementation of the array-based and hash-table maps so that they implement the **remove** method.

4. Extend the implementation of the hash table from Figures 15-7 and 15-8 so that the bucket array expands dynamically. Your implementation should keep track of the load factor for the hash table and perform a rehashing operation if the load factor exceeds the limit indicated by a constant defined as follows:

```
const double REHASH_THRESHOLD = 0.7;
```

5. Although the bucket-chaining approach used in the text is extremely effective in practice, other strategies exist for resolving collisions in hash tables. In the early days of computing—when memories were small enough that the cost of introducing extra pointers was taken seriously—hash tables often used a more memory-efficient strategy called ***open addressing,*** in which the key/value pairs are stored directly in the array, like this:



For example, if a key hashes to bucket #2, the open-addressing strategy tries to put that key and its value directly into the entry at **`array[2]`**.

The problem with this approach is that **`array[3]`** may already be assigned to another key that hashes to the same bucket. The simplest approach to dealing with collisions of this sort is to store each new key in the first free cell at or after its expected hash position. Thus, if a key hashes to bucket #2, the **`put`** and **`get`** functions first try to find or insert that key in **`array[2]`**. If that entry is filled with a different key, however, these functions move on to try **`array[3]`**, continuing the process until they find an empty entry or an entry with a matching key. As in the ring-buffer implementation of queues in Chapter 13, if the index advances past the end of the array, it should wrap around back to the beginning. This strategy for resolving collisions is called ***linear probing.***

Reimplement the **`HashMap`** class using open addressing with linear probing. Make sure your function generates an error if the client tries to enter a new key into a table that is already full.

6. As noted at the beginning of this chapter, the implementations of the map abstraction have been simplified by eliminating deep copying and selection using square brackets. Rewrite both the array-based version of **`Map`** and the **`HashMap`** class so that they support these operations.