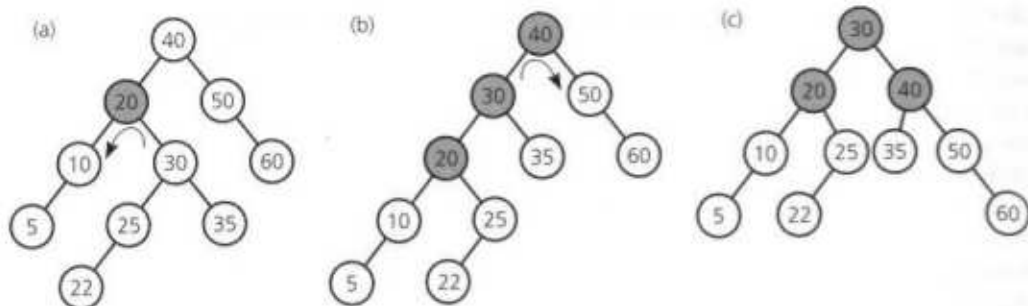

FIGURE 12-42

(a) Before; (b) during; and (c) after a double rotation

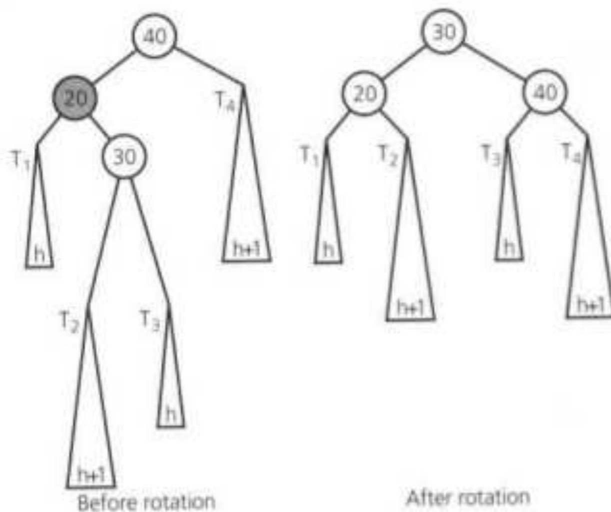

FIGURE 12-43

Before and after a double rotation that decreases the tree's height

12.2 Hashing

The binary search tree and its balanced variants, such as 2-3, 2-3-4, red-black, and AVL trees, provide excellent implementations of the ADT table. They allow you to perform all of the table operations quite efficiently. If, for example, a table contains 10,000 items, the operations `tableRetrieve`, `tableInsert`, and `tableDelete` each require approximately $\log_2 10{,}000 = 13$ steps. As impressive as this efficiency may be, situations do occur for which the search-tree implementations are not adequate.

As you know, time can be vital. For example, when a person calls the 911 emergency system, the system detects the caller's telephone number and

searches a database for the caller's address. Similarly, an air traffic control system searches a database of flight information, given a flight number. Clearly these searches must be rapid.

A radically different strategy is necessary to locate (and insert or delete) an item virtually instantaneously. Imagine an array `table` of *N* items—with each array slot capable of holding a single table item—and a seemingly magical box called an "address calculator." Whenever you have a new item that you want to insert into the table, the address calculator will tell you where you should place it in the array. Figure 12-44 illustrates this scenario.

Table operations without searches

You can thus easily perform an insertion into the table as follows:

```
tableInsert(in newItem:TableItemType)

   i = the array index that the address calculator
          gives you for newItem's search key
   table[i] = newItem
```

An insertion is O(1); that is, it requires constant time.

You also use the address calculator for the `tableRetrieve` and `tableDelete` operations. If you want to retrieve an item that has a particular search key, you simply ask the address calculator to tell you where it would insert such an item. Because you would have inserted the item earlier by using the `tableInsert` algorithm just given, if the desired item is present in the table, it will be in the array location that the address calculator specifies.

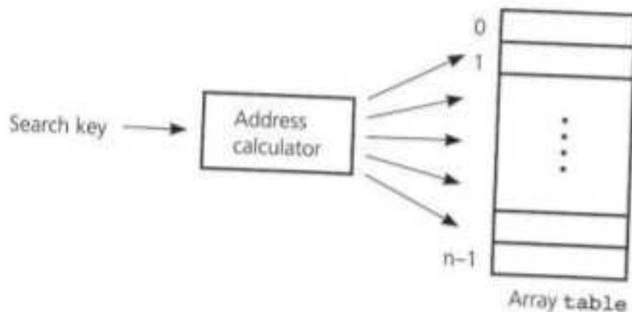

FIGURE 12-44
Address calculator

Thus, the retrieval operation appears in pseudocode as follows:

```
tableRetrieve(in searchKey:KeyType,
              out tableItem:TableItemType)
              throw TableException
```

```
i = the array index that the address calculator
      gives you for an item whose search key
      equals searchKey

if (table[i].getKey() != searchKey)
   Throw a TableException
else
   tableItem = table[i]
```

Similarly, the pseudocode for the deletion operation is

```
tableDelete(in searchKey:KeyType)
            throw TableException

   i = the array index that the address calculator
         gives you for an item whose search key
         equals searchKey

   if (table[i].getKey() != searchKey)
      Throw a TableException
   else
      Delete the item from table[i]
```

It thus appears that you can perform the operations `tableRetrieve`, `tableInsert`, and `tableDelete` virtually instantaneously. You never have to search for an item; instead, you simply let the address calculator determine where the item should be. The amount of time required to carry out the operations is O(1) and depends only on how quickly the address calculator can perform this computation.

A hash function tells you where to place an item in an array called a hash table

If you are to implement such a scheme, you must, of course, be able to construct an address calculator that can, with very little work, tell you where a given item should be. Address calculators are actually not as mysterious as they seem; in fact, many exist that can approximate the idealized behavior just described. Such an address calculator is usually referred to as a **hash function**. The scheme just described is an idealized description of a technique known as **hashing**, and the array `table` is called the **hash table**.

To understand how a hash function works, consider the 911 emergency system mentioned earlier. If, for each person, the system had a record whose search key was the person's telephone number, it could store these records in a search tree. Although searching a tree would be fast, faster access to a particular record would be possible by storing the records in an array `table`, as follows. You store the record for a person whose telephone number is *t* into `table[t]`. Retrieval of the record, then, is almost instantaneous given its search key *t*. For example, you can store the record for the telephone number 123-4567 in `table[1234567]`. If you can spare 10 million memory locations for `table`, this approach is fine. You need not use memory so extravagantly, however, because 911 systems are regional. If you consider only one telephone

exchange, for example, you can store the record for the number 123-4567 in `table[4567]` and get by with an array `table` of 10,000 locations.

The transformation of 1234567 into an array index 4567 is a simple example of a hash function. A hash function h must take an arbitrary integer x and map it into an integer that you can use as an array index. In our example, such indexes would be in the range 0 through 9999. That is, h is a function such that for any integer x,

$h(x) = i$, where i is an integer in the range 0 through 9999

A hash function maps an integer into an array index

Because the database contains records for every telephone number in a particular exchange, the array `table` is completely full. In this sense, our example is not typical of hashing applications and serves only to illustrate the idea of a hash function. What if many fewer records were in the array? Consider, for example, an air traffic control system that stores a record for each current flight according to its four-digit flight number. You could store a record for Flight 4567 in `table[4567]`, but you still would need an array of 10,000 locations, even if only 50 flights were current.

A different hash function would save memory. If you allow space for a maximum of 101 flights, for example, so that the array `table` has indexes 0 through 100, the necessary hash function h should map any four-digit flight number into an integer in the range 0 through 100.

If you have such a hash function h—and you will see several suggestions for hash functions later—the table operations are easy to write. For example, in the `tableRetrieve` algorithm, the step

```
i = the array index that the address calculator
      gives you for an item whose search key
      equals searchKey
```

is implemented simply as

```
i = h(searchKey)
```

In the previous example, `searchKey` would be the flight number.

The table operations appear to be virtually instantaneous. But is hashing really as good as it sounds? If it really were this good, there would have been little reason for developing all those other table implementations. Hashing would beat them hands down!

Why is hashing not quite as simple as it seems? You might first notice that since the hashing scheme stores the items in an array, it would appear to suffer from the familiar problems associated with a fixed-size implementation. Obviously, the hash table must be large enough to contain all of the items that you want to store. This requirement is not the crux of the implementation's difficulty, however, for—as you will see later—there are ways to allow the hash table to grow dynamically. The implementation has a major pitfall, even given the assumption that the number of items to be stored will never exceed the

A perfect hash function maps each search key into a unique location of the hash table

A perfect hash function is possible if you know all the search keys

Ideally, you want the hash function to map each x into a unique integer i. The hash function in the ideal situation is called a **perfect hash function**. In fact, it is possible to construct perfect hash functions if you know all of the possible search keys that *actually* occur in the table. You have this knowledge for the 911 example, since everyone is in the database, but not for the air traffic control example. Usually, you will not know the values of the search keys in advance.

Collisions occur when the hash function maps more than one item into the same array location

In practice, a hash function can map two or more search keys x and y into the *same* integer. That is, the hash function tells you to store two or more items in the same array location `table[i]`. This occurrence is called a **collision**. Thus, even if fewer than 101 items were present in the hash table `table[0..100]`, h could very well tell you to place more than one item into the same array location. For example, if two items have search keys 4567 and 7597, and if

$$h(4567) = h(7597) = 22$$

h will tell you to place the two items into the same array location, `table[22]`. That is, the search keys 4567 and 7597 have collided.

Even if the number of items that can be in the array at any one time is small, the only way to avoid collisions completely is for the hash table to be large enough that each possible search-key value can have its own location. If, for example, Social Security numbers were the search keys, you would need an array location for each integer in the range 000000000 through 999999999. This situation would certainly require a good deal of storage! Because reserving vast amounts of storage is usually not practical, collision-resolution schemes are necessary to make hashing feasible. Such resolution schemes usually require that the hash function place items evenly throughout the hash table.

To summarize, a typical hash function must

Requirements for a hash function

- Be easy and fast to compute
- Place items evenly throughout the hash table

Note that the size of the hash table affects the ability of the hash function to distribute the items evenly throughout the table. The requirements of a hash function will be discussed in more detail later in this chapter.

Consider now several hash functions and **collision-resolution schemes**.

Hash Functions

It is sufficient for hash functions to operate on integers

It is sufficient to consider hash functions that have an arbitrary integer as an argument. Why? If a search key is not an integer, you can simply map the search key into an integer, which you then hash. At the end of this section, you will see one way to convert a string into an integer.

There are many ways to convert an arbitrary integer into an integer within a certain range, such as 0 through 100. Thus, there are many ways to construct a hash function. Many of these functions, however, will not be suitable. Here are several simple hash functions that operate on positive integers.

Selecting digits. If your search key is the nine-digit employee ID number 001364825, you could select the fourth digit and the last digit, to obtain 35 as the index to the hash table. That is,

$h(001364825) = 35$ *(select the fourth and last digits)*

Therefore, you would store the item whose search key is 001364825 in `table[35]`.

You do need to be careful about which digits you choose in a particular situation. For example, the first three digits of a Social Security number are based on the geographic region in which the number was assigned. If you select only these digits, you will map all people from the same state into the same location of the hash table.

Digit-selection hash functions are simple and fast, but generally they do not evenly distribute the items in the hash table. A hash function really should utilize the entire search key.

Digit selection does not distribute items evenly in the hash table

Folding. One way to improve on the previous approach of selecting digits is to add the digits. For example, you can add all of the digits in 001364825 to obtain

$0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29$ *(add the digits)*

Therefore, you would store the item whose search key is 001364825 in `table[29]`. Notice that if you add all of the digits from a nine-digit search key,

$0 \leq h(\text{search key}) \leq 81$

That is, you would use only `table[0]` through `table[81]` of the hash table. To change this situation or to increase the size of the hash table, you can group the digits in the search key and add the groups. For example, you could form three groups of three digits from the search key 001364825 and add them as follows:

$001 + 364 + 825 = 1{,}190$

For this hash function,

$0 \leq h(\text{search key}) \leq 3 * 999 = 2{,}997$

Clearly, if 2,997 is larger than the size of the hash table that you want, you can alter the groups that you choose. Perhaps not as obvious is that you can apply more than one hash function to a search key. For example, you could select some of the digits from the search key before adding them, or you could either select digits from the previous result 2,997 or apply folding to it once again by adding 29 and 97.

Applying more than one hash function to a single search key

Modulo arithmetic. Modulo arithmetic provides a simple and effective hash function that we will use in the rest of this chapter. For example, consider the function[6]

$h(x) = x \bmod tableSize$

6. Remember that this book uses "mod" as an abbreviation for the mathematical operation modulo. In C++, the modulo operator is %.

where the hash table `table` has *tableSize* elements. In particular, if *tableSize* is 101, $h(x) = x \bmod 101$ maps any integer x into the range 0 through 100. For example, h maps 001364825 into 12.

For $h(x) = x \bmod tableSize$, many x's map into `table[0]`, many x's map into `table[1]`, and so on. That is, collisions occur. However, you can distribute the table items evenly over all of `table`—thus reducing collisions—by choosing a prime number as *tableSize*. For instance, 101 in the previous example is prime. The choice of table size will be discussed in more detail later in this chapter. For now, realize that 101 is used here as a simple example of a prime table size. For the typical table, it is much too small.

The table size should be prime

Converting a character string to an integer. If your search key is a character string—such as a name—you could convert it into an integer before applying the hash function $h(x)$. To do so, you could first assign each character in the string an integer value. For example, for the word "NOTE" you could assign the ASCII values 78, 79, 84, and 69, to the letters N, O, T, and E, respectively. Or, if you assign the values 1 through 26 to the letters A through Z, you could assign 14 to N, 15 to O, 20 to T, and 5 to E.

If you now simply add these numbers, you will get an integer, but it will not be unique to the character string. For example, the string "TONE" will give you the same result. Instead, write the numeric value for each character in binary and concatenate the results. If you assign the values 1 through 26 to the letters A through Z, you obtain the following for the string "NOTE":

N is 14, or 01110 in binary

O is 15, or 01111 in binary

T is 20, or 10100 in binary

E is 5, or 00101 in binary

Concatenating the binary values gives you the binary integer

01110011111010000101

which is 474,757 in decimal. You can apply the hash function $x \bmod tableSize$ for $x = 474{,}757$.

Now consider a more efficient way to compute 474,757. Rather than converting the previous binary number to decimal, you can evaluate the expression

$$14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0$$

This computation is possible because we have represented each character as a 5-bit binary number, and 2^5 is 32.

By factoring this expression, you can minimize the number of arithmetic operations. This technique is called Horner's rule and results in

Horner's rule minimizes the number of computations

$$((14 * 32 + 15) * 32 + 20) * 32 + 5$$

Although both of these expressions have the same value, the result in either case could very well be larger than a typical computer can represent; that is, an overflow can occur.

Because we plan to use the hash function

$h(x) = x \bmod tableSize$

you can prevent an overflow by applying the modulo operator after computing each parenthesized expression in Horner's rule. The implementation of this algorithm is left as an exercise.

Resolving Collisions

Consider the problems caused by a collision. Suppose that you want to insert an item whose search key is 4567 into the hash table `table`, as was described previously. The hash function $h(x) = x \bmod 101$ tells you to place the new item in `table[22]`, because 4567 mod 101 is 22. Suppose, however, that `table[22]` already contains an item, as Figure 12-45 illustrates. If earlier you had placed 7597 into `table[22]` because 7597 mod 101 equals 22, where do you place the new item? You certainly do not want to disallow the insertion on the grounds that the table is full: You could have a collision even when inserting into a table that contains only one item!

Two approaches to collision resolution

Two general approaches to collision resolution are common. One approach allocates another location *within* the hash table to the new item. A second approach changes the structure of the hash table so that each location

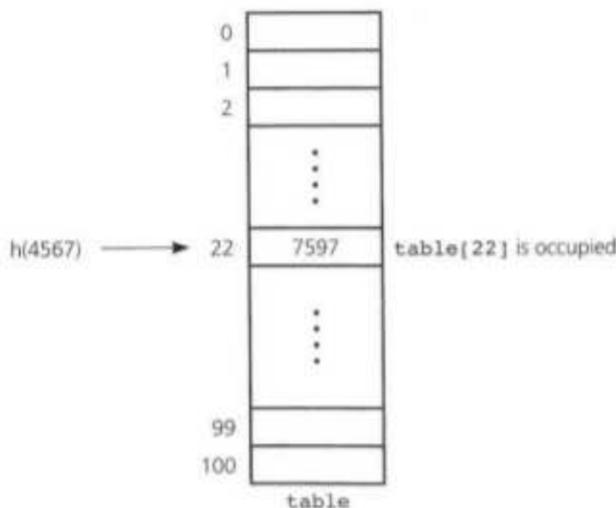

FIGURE 12-45

A collision

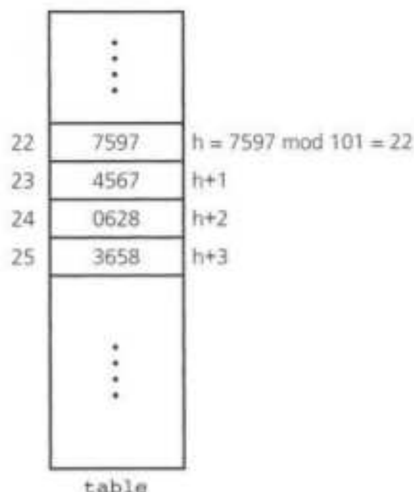

FIGURE 12-46
Linear probing with $h(x) = x \bmod 101$

`table[i]` can accommodate more than one item. The collision-resolution schemes described next exemplify these two approaches.

Approach 1: Open addressing. During an attempt to insert a new item into a table, if the hash function indicates a location in the hash table that is already occupied, you probe for some other empty, or open, location in which to place the item. The sequence of locations that you examine is called the **probe sequence**.

Such schemes are said to use **open addressing**. The concern, of course, is that you must be able to find a table item efficiently after you have inserted it. That is, the `tableDelete` and `tableRetrieve` operations must be able to reproduce the probe sequence that `tableInsert` used and must do so efficiently.

The difference among the various open-addressing schemes is the technique used to probe for an empty location. We briefly describe three such techniques.

Begin at the hash location and search the table sequentially

Linear probing. In this simple scheme to resolve a collision, you search the hash table sequentially, starting from the original hash location. More specifically, if `table[h(searchKey)]` is occupied, you check the table locations `table[h(searchKey)+1]`, `table[h(searchKey)+2]`, and so on until you find an available location. Figure 12-46 illustrates the placement of four items that all hash into the same location `table[22]` of the hash table, assuming a

hash function $h(x) = x \bmod 101$. Typically, you *wrap around* from the last table location to the first table location if necessary.

In the absence of deletions, the implementation of `tableRetrieve` under this scheme is straightforward. You need only follow the same probe sequence that `tableInsert` used until you either find the item you are searching for, reach an empty location, which indicates that the item is not present, or visit every table location.

Deletions, however, add a slight complication. The `tableDelete` operation itself is no problem. You merely find the desired item, as in `tableRetrieve`, and delete it, making the location empty. But what happens to `tableRetrieve` after deletions? The new empty locations that `tableDelete` created along a probe sequence could cause `tableRetrieve` to stop prematurely, incorrectly indicating a failure. You can resolve this problem by allowing a table location to be in one of three states: occupied (currently in use), empty (has not been used), or deleted (was once occupied but is now available). You then modify the `tableRetrieve` operation to continue probing when it encounters a location in the deleted state. Similarly, you modify `tableInsert` to insert into either empty or deleted locations.

Three states: occupied, empty, deleted

One of the problems with the linear-probing scheme is that table items tend to **cluster** together in the hash table. That is, the table contains groups of consecutively occupied locations. This phenomenon is called *primary clustering*. Clusters can get close to one another and, in fact, merge into a larger cluster. Large clusters tend to get even larger. ("The rich get richer.") Thus, one part of the table might be quite dense, even though another part has relatively few items. Primary clustering causes long probe searches and therefore decreases the overall efficiency of hashing.

Clustering can be a problem

Quadratic probing. You can virtually eliminate primary clusters simply by adjusting the linear probing scheme just described. Instead of probing consecutive table locations from the original hash location `table[h(searchKey)]`, you check locations `table[h(searchKey)+1`^{2}`]`, `table[h(searchKey)+2`^{2}`]`, `table[h(searchKey)+3`^{2}`]`, and so on until you find an available location. Figure 12-47 illustrates this open-addressing scheme—which is called **quadratic probing**—for the same items that appear in Figure 12-46.

Unfortunately, when two items hash into the same location, quadratic probing uses the same probe sequence for each item. This phenomenon—called *secondary clustering*—delays the resolution of the collision. Although the analysis of quadratic probing remains incomplete, it appears that secondary clustering is not a problem.

Double hashing. Double hashing, which is yet another open-addressing scheme, drastically reduces clustering. The probe sequences that both linear probing and quadratic probing use are *key independent*. For example, linear probing inspects the table locations sequentially no matter what the hash key is. In contrast, double hashing defines *key-dependent* probe sequences. In this scheme the probe sequence still searches the table in a linear order, starting at

	table	
	⋮	
22	7597	$h = 7597 \bmod 101 = 22$
23	4567	$h+1^2$
24		
25		
26	0628	$h+2^2$
	⋮	
31	3658	$h+3^2$
	⋮	

FIGURE 12-47

Quadratic probing with $h(x) = x \bmod 101$

A hash address and a step size determine the probe sequence

the location $h_1(key)$, but a second hash function h_2 determines the size of the steps taken.

Although you choose h_1 as usual, you must follow these guidelines for h_2:

Guidelines for the step-size function h_2

$$h_2(key) \neq 0$$
$$h_2 \neq h_1$$

Clearly, you need a nonzero step size $h_2(key)$ to define the probe sequence. In addition, h_2 must differ from h_1 to avoid clustering.

For example, let h_1 and h_2 be the primary and secondary hash functions defined as

Primary and secondary hash functions

$$h_1(key) = key \bmod 11$$
$$h_2(key) = 7 - (key \bmod 7)$$

where a hash table of only 11 items is assumed, so that you can readily see the effect of these functions on the hash table. If $key = 58$, h_1 hashes *key* to table location 3 (58 mod 11), and h_2 indicates that the probe sequence should take steps of size 5 (7 – 58 mod 7). In other words, the probe sequence will be 3, 8, 2 (wraps around), 7, 1 (wraps around), 6, 0, 5, 10, 4, 9. On the other hand, if $key = 14$, h_1 hashes *key* to table location 3 (14 mod 11), and h_2 indicates that the probe sequence should take steps of size 7 (7 – 14 mod 7), and so the probe sequence would be 3, 10, 6, 2, 9, 5, 1, 8, 4, 0.

Each of these probe sequences visits *all* the table locations. This phenomenon always occurs if the size of the table and the size of the probe step are relatively prime, that is, if their greatest common divisor is 1. Because the size of a hash table is commonly a prime number, it will be relatively prime to all step sizes.

Figure 12-48 illustrates the insertion of 58, 14, and 91 into an initially empty hash table. Because $h_1(58)$ is 3, you place 58 into `table[3]`. You then find that $h_1(14)$ is also 3, so to avoid a collision, you step by $h_2(14) = 7$ and place 14 into `table[3 + 7]`, or `table[10]`. Finally, $h_1(91)$ is 3 and $h_2(91)$ is 7. Because `table[3]` is occupied, you probe `table[10]` and find that it, too, is occupied. You finally store 91 in `table[(10 + 7) % 11]`, or `table[6]`.

Using more than one hash function is called rehashing. While more than two hash functions can be desirable, such schemes are difficult to implement.

Increasing the size of the hash table. With any of the open-addressing schemes, as the hash table fills, the probability of a collision increases. At some point, a larger hash table becomes desirable. If you use a dynamically allocated array for the hash table, you can increase its size whenever the table becomes too full.

You cannot simply double the size of the array, as we did in earlier chapters, because the size of the hash table must remain prime. Secondly, you do not copy the items from the original hash table to the new hash table. If your hash function is x mod *tableSize*, it changes as *tableSize* changes. Thus, you need to apply your new hash function to every item in the old hash table before placing it into the new hash table.

Approach 2: Restructuring the hash table. Another way to resolve collisions is to change the structure of the array `table`—the hash table—so that it can accommodate more than one item in the same location. We describe two such ways to alter the hash table.

Each hash-table location can accommodate more than one item

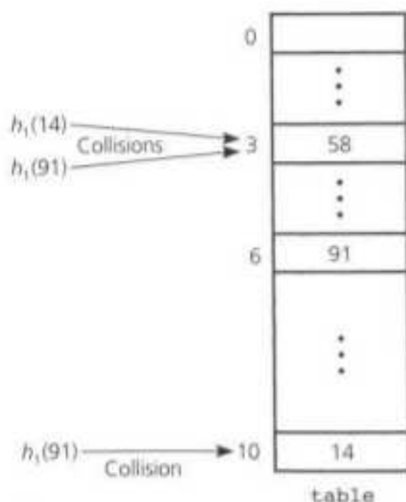

FIGURE 12-48

Double hashing during the insertion of 58, 14, and 91

```
typedef KeyedItem TableItemType;
/** ADT table.
 * Hash table implementation.
 * Assumption: A table contains at most one item with a
 *             given search key at any time. */
class HashTable
{
public:
// constructors and destructor:
   HashTable();
   HashTable(const HashTable& table);
   virtual ~HashTable();

// table operations:
   virtual bool tableIsEmpty() const;
   virtual int tableGetLength() const;
   virtual void tableInsert(const TableItemType& newItem)
                           throw(TableException);
   virtual void tableDelete(KeyType searchKey)
                           throw(TableException);
   virtual void tableRetrieve(KeyType searchKey,
                           TableItemType& tableItem) const
                           throw(TableException);

protected:
   int hashIndex(KeyType searchKey) const;  // Hash function

private:
   static const int HASH_TABLE_SIZE = 101;  // Size of hash
                                            // table
   typedef ChainNode * HashTableType[HASH_TABLE_SIZE];

   HashTableType table;            // Hash table */
   int           size;             // size of ADT table
}; // end HashTable
// End of header file.

/** @file KeyedItem.h
 * Provides basis for classes that need a search key value. */
typedef desired-type-of-search-key KeyType;

class KeyedItem
{
public:
   KeyedItem() {}
   KeyedItem(const KeyType& keyValue)
             : searchKey(keyValue) {}
```

```
   KeyType getKey() const // returns search key
   { return searchKey;
   }  // end getKey
private:
   KeyType searchKey;
}; // end KeyedItem

/** @file ChainNode.h.
 * Provides the chain node definition for the hash table. */
#include "KeyedItem.h"

class ChainNode
{
private:
   ChainNode(const KeyedItem & nodeItem,
             ChainNode *nextNode = NULL)
            : item(nodeItem), next(nextNode) {}
   KeyedItem item;
   ChainNode *next;

   friend class HashTable;
}; // end ChainNode
```

The class `KeyedItem` can be used as the base class for the items that are stored in the table. The `KeyedItem` class was first presented in Chapter 10 and provides a data field for the search key. The search key is used by the `hashIndex` method in the class `HashTable` to generate a hash index value.

When you insert a new item into the table, you place it at the beginning of the linked list that the hash function indicates. The following pseudocode describes the insertion algorithm:

```
tableInsert(in newItem:TableItemType)
            throw TableException

   searchKey = the search key of newItem
   i = hashIndex(searchKey)
   p = pointer to a new node
   Throw TableException according to whether the
     previous memory allocation is successful
   p->item = newItem
   p->next = table[i]
   table[i] = p
```

When you want to retrieve an item, you search the linked list that the hash function indicates. The following pseudocode describes the retrieval algorithm:

```
tableRetrieve(in searchKey:KeyType,
              out tableItem:TableItemType)
              throw TableException

   i = hashIndex(searchKey)
   p = table[i]

   while ( (p != NULL) &&
           (p->item.getKey() != searchKey) )
      p = p->next

   if (p == NULL)
      Throw a TableException
   else
      tableItem = p->item
```

The deletion algorithm is very similar to the retrieval algorithm and is left as an exercise. (See Exercise 14.)

Separate chaining is thus a successful approach to resolving collisions. With separate chaining, the size of the ADT table is dynamic and can exceed the size of the hash table, because each linked list can be as long as necessary. As you will see in the next section, the length of these linked lists affects the efficiency of retrievals and deletions.

Separate chaining successfully resolves collisions

The Efficiency of Hashing

An analysis of the average-case efficiency of hashing involves the **load factor** α, which is the ratio of the current number of items in the table to the maximum size of the array `table`. That is,

$$\alpha = \frac{\textit{Current number of table items}}{\textit{tableSize}}$$

The load factor measures how full a hash table is

α is a measure of how full the hash table `table` is. As `table` fills, α increases and the chance of collision increases, so search times increase. Thus, hashing efficiency decreases as α increases.

Unlike the efficiency of earlier table implementations, the efficiency of hashing does not depend solely on the number N of items in the table. While it is true that for a fixed *tableSize*, efficiency decreases as N increases, for a given N you can choose *tableSize* to increase efficiency. Thus, when determining *tableSize*, you should estimate the largest possible N and select *tableSize* so that α is small. As you will see shortly, α should not exceed $2/3$.

Unsuccessful searches generally require more time than successful searches

Hashing efficiency for a particular search also depends on whether the search is successful. An unsuccessful search requires more time in general than

a successful search. The following analyses[8] enable a comparison of collision-resolution techniques.

Linear probing. For linear probing, the approximate average number of comparisons that a search requires is

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right] \quad \text{for a successful search, and}$$

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]^2 \quad \text{for an unsuccessful search}$$

As collisions increase, the probe sequences increase in length, causing increased search times. For example, for a table that is two-thirds full ($\alpha = 2/3$), an average unsuccessful search might require at most five comparisons, or probes, while an average successful search might require at most two comparisons. To maintain efficiency, it is important to prevent the hash table from filling up.

Do not let the hash table get too full

Quadratic probing and double hashing. The efficiency of both quadratic probing and double hashing is given by

$$\frac{-\log_e(1-\alpha)}{\alpha} \quad \text{for a successful search, and}$$

$$\frac{1}{1-\alpha} \quad \text{for an unsuccessful search}$$

On average, both techniques require fewer comparisons than linear probing. For example, for a table that is two-thirds full, an average unsuccessful search might require at most three comparisons, or probes, while an average successful search might require at most two comparisons. As a result, you can use a smaller hash table for both quadratic probing and double hashing than you can for linear probing. However, because they are open-addressing schemes, all three approaches suffer when you are unable to predict the number of insertions and deletions that will occur. If your hash table is too small, it will fill up, and search efficiency will decrease.

Open-addressing schemes require a good estimate of the number of insertions and deletions

Separate chaining. Because the `tableInsert` operation places the new item at the beginning of a linked list within the hash table, it is O(1). The `tableRetrieve` and `tableDelete` operations, however, are not as fast. They each require a search of the linked list of items, so ideally you would like for these linked lists to be short.

Insertion is instantaneous

For separate chaining, *tableSize* is the number of linked lists, not the maximum number of table items. Thus, it is entirely possible, and even likely, that the current number of table items *N* exceeds *tableSize*. That is, the load

8. D. E. Knuth, *Searching and Sorting*, vol. 3 of *The Art of Computer Programming* (Menlo Park, CA: Addison-Wesley, 1973).

factor α, or *N/tableSize*, can exceed 1. Because *tableSize* is the number of linked lists, *N/tableSize*—that is, α—is the average length of each linked list.

Some searches of the hash table are unsuccessful because the relevant linked list is empty. Such searches are virtually instantaneous. For an unsuccessful search of a nonempty linked list, however, `tableRetrieve` and `tableDelete` must examine the entire list, or α items in the average case. On the other hand, a successful search must examine a nonempty linked list. In the average case, the search will locate the item in the middle of the list. That is, after determining that the linked list is not empty, the search will examine $\alpha/2$ items.

Thus, the efficiency of the retrieval and deletion operations under the separate-chaining approach is

Average-case efficiency of retrievals and deletions

$1 + \frac{\alpha}{2}$ for a successful search, and

α for an unsuccessful search

Even if the linked lists typically are short, you should still estimate the worst case. If you seriously underestimate *tableSize* or if most of the table items happen to hash into the same location, the number of items in a linked list could be quite large. In fact, in the worst case, all *N* items in the table could be in the same linked list!

As you can see, the time that a retrieval or deletion operation requires can range from almost nothing—if the linked list to be searched either is empty or has only a couple of items in it—to the time required to search a linked list that contains all the items in the table, if all the items hashed into the same location.

Comparing techniques. Figure 12-50 plots the relative efficiency of the collision-resolution schemes just discussed. When the hash table `table` is about half full—that is, when α is 0.5—the techniques are nearly equal in efficiency. As the table fills and α approaches 1, separate chaining is the most efficient. Does this mean that we should discard all other search algorithms in favor of hashing with separate chaining?

No. The analyses here are average-case analyses. Although an implementation of the ADT table that uses hashing might often be faster than one that uses a search tree, in the worst case it can be much slower. If you can afford both an occasional slow search and a large *tableSize*—that is, a small α—then hashing can be an attractive table implementation. However, if you are performing a life-and-death search for your city's poison control center, a search-tree implementation would at least provide you with a guaranteed bound on its worst-case behavior.

In the worst case, a hashing implementation of a table can be much slower than other implementations

Furthermore, while separate chaining is the most time-efficient collision-resolution scheme, you do have the storage overhead of the pointers in the linked list. If the data records in the table are small, the pointers add a

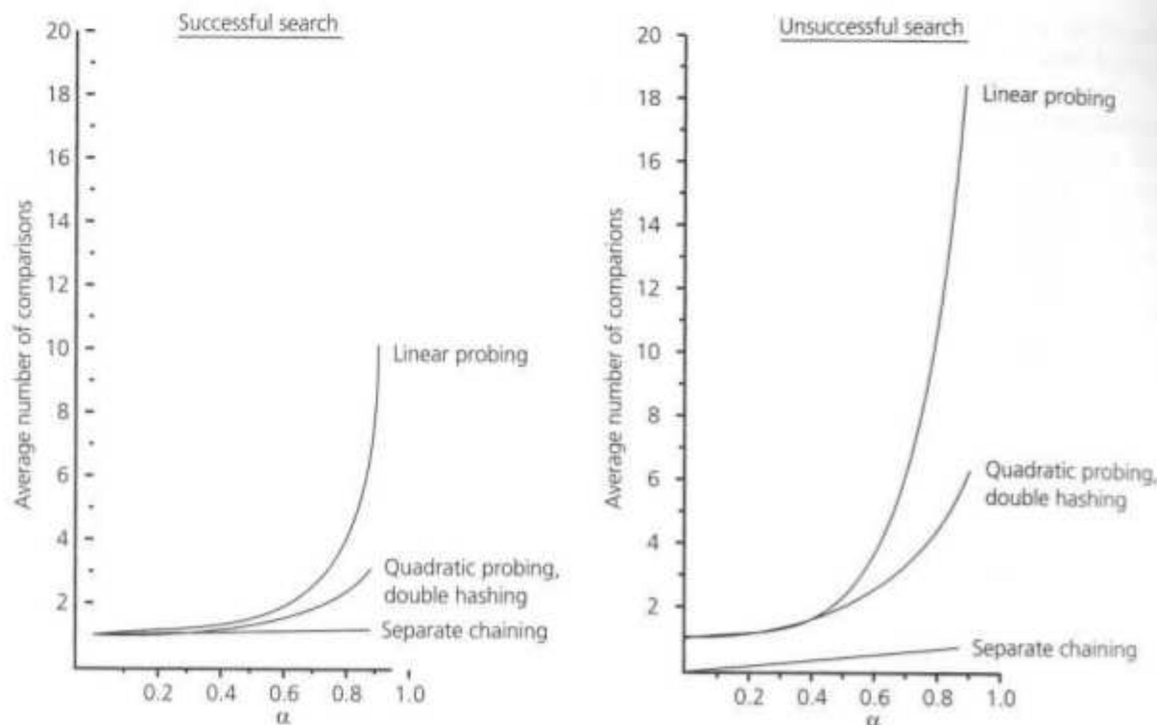

FIGURE 12-50

The relative efficiency of four collision-resolution methods

significant overhead in storage, and you may want to consider a simpler collision-resolution scheme. On the other hand, if the records are large, the addition of a pointer is insignificant, so separate chaining is a good choice.

What Constitutes a Good Hash Function?

Before we conclude this introduction to hashing, consider in more detail the issue of choosing a hash function to perform the address calculations for a given application. A great deal has been written on this subject, most of which is beyond the mathematical level of this book. However, this section will present a brief summary of the major concerns.

- **A hash function should be easy and fast to compute.** If a hashing scheme is to perform table operations almost instantaneously and in constant time, you certainly must be able to calculate the hash function rapidly. Most of the common hash functions require only a single division (like the modulo function), a single multiplication, or some kind of "bit-level" operation on the internal representation of the search key. In all these cases, the requirement that the hash function be easy and fast to compute is satisfied.

- **A hash function should scatter the data evenly throughout the hash table.** Unless you use a perfect hash function—which is usually impractical to construct—you typically cannot avoid collisions entirely. For example, to achieve the best performance from a separate-chaining scheme, each entry `table[i]` should contain approximately the same number of items in its chain; that is, each chain should contain approximately $N/tableSize$ items (and thus no chain should contain significantly more than $N/tableSize$ items). To accomplish this goal, your hash function should scatter the search keys evenly throughout the hash table.

You cannot avoid collisions entirely

There are two issues to consider with regard to how evenly a hash function scatters the search keys.

- **How well does the hash function scatter random data?** If every search-key value is equally likely, will the hash function scatter the search keys evenly? For example, consider the following scheme for hashing nine-digit ID numbers:

  `table[0..39]` is the hash table, and
  the hash function is $h(x)$ = (first two digits of x) mod 40

  The question is, given the assumption that all employee ID numbers are equally likely, does a given ID number x have equal probability of hashing into any one of the 40 array locations? For this hash function, the answer is no. Only ID numbers that start with 19, 59, and 99 map into `table[19]`, while only ID numbers that start with 20 and 60 map into `table[20]`. In general, three different ID *prefixes*—that is, the first two digits of an ID number—map into each array location 0 through 19, while only two different prefixes map into each array location 20 through 39. Because all ID numbers are equally likely—and thus all prefixes 00 through 99 are equally likely—a given ID number is 50 percent more likely to hash into one of the locations 0 through 19 than it is to hash into one of the locations 20 through 39. As a result, each array location 0 through 19 would contain, on average, 50 percent more items than each location 20 through 39.

  Thus, the hash function

  $h(x)$ = (first two digits of x) mod 40

  does not scatter random data evenly throughout the array `table[0..39]`. On the other hand, it can be shown that the hash function

  A function that does not scatter random data evenly

  $h(x) = x \bmod 101$

  does, in fact, scatter random data evenly throughout the array `table[0..100]`.

  A function that does scatter random data evenly

- **How well does the hash function scatter nonrandom data?** Even if a hash function scatters random data evenly, it may have trouble with nonrandom data. In general, no matter what hash function you select, it is always possible that the data will have some unlucky pattern that will result in uneven

scattering. Although there is no way to guarantee that a hash function will scatter all data evenly, you can greatly increase the likelihood of this behavior.

As an example, consider the following scheme:

`table[0..99]` is the hash table, and
the hash function is $h(x)$ = first two digits of x

If every ID number is equally likely, h will scatter the search keys evenly throughout the array. But what if every ID number is not equally likely? For instance, a company might assign employee IDs according to department, as follows:

10xxxxx Sales
20xxxxx Customer Relations
...
90xxxxx Data Processing

Under this assignment, only 9 out of the 100 array locations would contain any items at all. Further, those locations corresponding to the largest departments (Sales, for example, which corresponds to `table[10]`) would contain more items than those locations corresponding to the smallest departments. This scheme certainly does not scatter the data evenly. Much research has been done into the types of hash functions that you should use to guard against various types of patterns in the data. The results of this research are really in the province of more advanced courses, but two general principles can be noted here:

General requirements of a hash function

1. The calculation of the hash function should *involve the entire search key*. Thus, for example, computing a modulo of the entire ID number is much safer than using only its first two digits.

2. If a hash function uses modulo arithmetic, ***the base should be prime;*** that is, if h is of the form

   $h(x) = x \bmod tableSize$

   then *tableSize* should be a prime number. This selection of *tableSize* is a safeguard against many subtle kinds of patterns in the data (for example, search keys whose digits are likely to be multiples of one another). Although each application can have its own particular kind of patterns and thus should be analyzed on an individual basis, choosing *tableSize* to be prime is an easy way to safeguard against some common types of patterns in the data.

Table Traversal: An Inefficient Operation Under Hashing

For many applications, hashing provides the most efficient implementation of the ADT table. One important table operation—traversal in sorted order—performs poorly when hashing implements the table. As was mentioned previously, a good

hash function scatters items as randomly as possible throughout the array, so that no ordering relationship exists between the search keys that hash into `table[i]` and those that hash into `table[i+1]`. As a consequence, if you must traverse the table in sorted order, you first would have to sort the items. If sorting were required frequently, hashing would be a far less attractive implementation than a search tree.

Items hashed into `table[i]` and `table[i+1]` have no ordering relationship

Traversing a table in sorted order is really just one example of a whole class of operations that hashing does not support well. Many similar operations that you often wish to perform on a table require that the items be ordered. For example, consider an operation that must find the table item with the smallest or largest value in its search key. If you use a search-tree implementation, these items are in the leftmost and rightmost nodes of the tree, respectively. If you use a hashing implementation, however, you do not know where these items are—you would have to search the entire table. A similar type of operation is a **range query**, which requires that you retrieve all items whose search keys fall into a given range of values. For example, you might want to retrieve all items whose search keys are in the range 129 to 755. This task is relatively easy to perform by using a search tree (see Exercise 3), but if you use hashing, there is no efficient way to answer the range query.

In general, if an application requires any of these ordered operations, you should probably use a search tree. Although the `tableRetrieve`, `tableInsert`, and `tableDelete` operations are somewhat more efficient when you use hashing to implement the table instead of a balanced search tree, the balanced search tree supports these operations so efficiently itself that, in most contexts, the difference in speed for these operations is negligible (whereas the advantage of the search tree over hashing for the ordered operations is significant).

Hashing versus balanced search trees

In the context of external storage, however, the story is different. For data that is stored externally, the difference in speed between hashing's implementation of `tableRetrieve` and a search tree's implementation may well be significant, as you will see in Chapter 14. In an external setting, it is not uncommon to see a hashing implementation of the `tableRetrieve` operation and a search-tree implementation of the ordered operations used simultaneously.

Implementing a *HashMap* Class Using the STL

The standard C++ library does not contain a hash table class. However, there are several implementations of the STL that provide `hash_map` and `hash_set` classes. These classes will most likely be included in the next revision of the C++ STL. In the meantime, if programmers would like to provide a hash function to use with a table, they can either download one of the available implementations or write a hash table class themselves.

The following `HashMap` class is a template class derived from existing STL containers. It is implemented with separate chaining using a vector of `maps`. The vector holds the dynamic hash buckets, where each bucket is a map that holds elements with the same hash value. The hash function must be supplied as a template parameter, along with the key type, value, and the optional comparison function object for the `map` class.

The following files contain an ADT for a *HashMap*.

```
/** @file HashMap.h
 * The HashMap is derived from the STL vector and map. */

#include <vector>
#include <map>

using namespace std;
template <typename Key, typename T, typename Hash>
class HashMap : private vector<map<Key, T> >
{
public:

   /** Constructor
    * @pre The HashMap is empty.
    * @post The HashMap is initialized to hold maxBuckets. The
    *       Hash template parameter is assigned to the hash
    *       variable. */
   HashMap(const int maxBuckets);

   /** Overloads the subscript operator for the HashMap class
    * @pre The HashMap contains a hash for type T.
    * @post The value of a hashed key is returned. */
   T& operator[](Key& key);

   /** Hashes the key to find the vector index. Finds the map
    *  element using the key as the index.
    * @pre The HashMap contains a hash for type T.
    * @post An iterator to the (key, item) pair is returned.
    *       If the item is not in the map, the iterator points
    *       to the end of the map. */
   map<Key, T>::const_iterator findItem(const Key& key);

   /** Hashes the key to find the vector index. Inserts the
    *  (key, item) pair into the map at that index.
    * @pre The HashMap contains a hash for the type T.
    * @post The (key, item) pair is inserted at the hashed
    *       index. */
   void insert(const Key& key, const T& item);

   /** Removes the item with the Key k in the hash table.
    * @pre The HashMap contains a hash for the type T.
    * @post The (key, item) pair at the hashed index is
    *       removed. The number of items removed is returned
    *       (either 0 or 1). */
   int erase(const Key& k);
```

```
   /** The hash function object. */
   Hash hash;
}; // end HashMap

#include "HashMap.cpp"
// End of header file

/** @file HashMap.cpp */

template <typename Key, typename T, typename Hash>
HashMap<Key, T, Hash>::HashMap(int maxBuckets)
{
   hash = Hash();
   resize(maxBuckets+1);
}  // end constructor

template <typename Key, typename T, typename Hash>
T& HashMap<Key, T, Hash>::operator[](Key& key)
{
   return at(hash(key))[key];
}  // end operator[]

template <typename Key, typename T, typename Hash>
map<Key, T>::const_iterator HashMap<Key, T, Hash>::
             findItem(const Key& key)
{
   map<Key, T>::const_iterator it;
   int index = hash(key);
   it = at(index).find(key);

   return it;
}  // end findItem

template <typename Key, typename T, typename Hash>
void HashMap<Key, T, Hash>::insert (const Key& key,
                                    const T& item)
{
   int index = hash(key);
   at(index).insert(make_pair(key, item));
}  // end insert

template <typename Key, typename T, typename Hash>
int HashMap<Key, T, Hash>::erase(const Key& key)
{
    int index = hash(key);
    return at(index).erase(key);
```