# 11

# Hash Tables

IN THIS CHAPTER

- Introduction to Hashing
- Open Addressing
- Separate Chaining
- Hash Functions
- Hashing Efficiency
- Hashing and External Storage

A *hash table* is a data structure that offers very fast insertion and searching. When you first hear about them, hash tables sound almost too good to be true. No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) in big O notation. In practice this is just a few machine instructions.

For a human user of a hash table, this is essentially instantaneous. It's so fast that computer programs typically use hash tables when they need to look up tens of thousands of items in less than a second (as in spelling checkers). Hash tables are significantly faster than trees, which, as we learned in the preceding chapters, operate in relatively fast O(logN) time. Not only are they fast, hash tables are relatively easy to program.

Hash tables do have several disadvantages. They're based on arrays, and arrays are difficult to expand after they've been created. For some kinds of hash tables, performance may degrade catastrophically when a table becomes too full, so the programmer needs to have a fairly accurate idea of how many data items will need to be stored (or be prepared to periodically transfer data to a larger hash table, a time-consuming process).

Also, there's no convenient way to visit the items in a hash table in any kind of order (such as from smallest to largest). If you need this capability, you'll need to look elsewhere.

However, if you don't need to visit items in order, and you can predict in advance the size of your database, hash tables are unparalleled in speed and convenience.

# Introduction to Hashing

In this section we'll introduce hash tables and hashing. One important concept is how a range of key values is transformed into a range of array index values. In a hash table this is accomplished with a hash function. However, for certain kinds of keys, no hash function is necessary; the key values can be used directly as array indices. We'll look at this simpler situation first and then go on to show how hash functions can be used when keys aren't distributed in such an orderly fashion.

## Employee Numbers as Keys

Suppose you're writing a program to access employee records for a small company with, say, 1,000 employees. Each employee record requires 1,000 bytes of storage. Thus, you can store the entire database in only 1 megabyte, which will easily fit in your computer's memory.

The company's personnel director has specified that she wants the fastest possible access to any individual record. Also, every employee has been given a number from 1 (for the founder) to 1,000 (for the most recently hired worker). These employee numbers can be used as keys to access the records; in fact access by other keys is deemed unnecessary. Employees are seldom laid off, but even when they are, their records remain in the database for reference (concerning retirement benefits and so on). What sort of data structure should you use in this situation?

### Index Numbers As Keys

One possibility is a simple array. Each employee record occupies one cell of the array, and the index number of the cell is the employee number for that record. This type of array is shown in Figure 11.1.

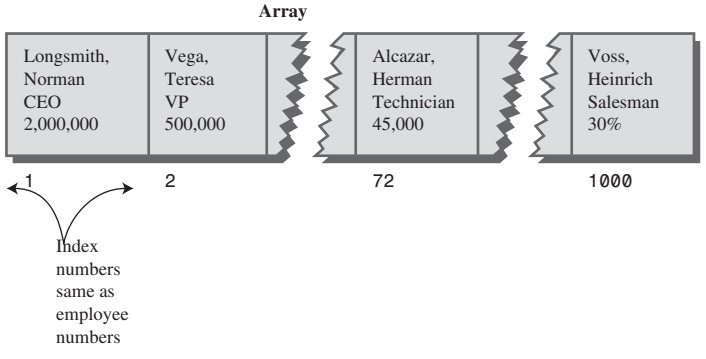


***FIGURE 11.1*** Employee numbers as array indices.

As you know, accessing a specified array element is very fast if you know its index number. The clerk looking up Herman Alcazar knows that he is employee number 72, so he enters that number, and the program goes instantly to index number 72 in the array. A single program statement is all that's necessary:

```
empRecord rec = databaseArray[72];
```

Adding a new item is also very quick: You insert it just past the last occupied element. The next new record—for Jim Chan, the newly hired employee number 1,001—would go in cell 1,001. Again, a single statement inserts the new record:

```
databaseArray[totalEmployees++] = newRecord;
```

Presumably, the array is made somewhat larger than the current number of employees, to allow room for expansion, but not much expansion is anticipated.

#### Not Always So Orderly

The speed and simplicity of data access using this array-based database make it very attractive. However, our example works only because the keys are unusually well organized. They run sequentially from 1 to a known maximum, and this maximum is a reasonable size for an array. There are no deletions, so memory-wasting gaps don't develop in the sequence. New items can be added sequentially at the end of the array, and the array doesn't need to be very much larger than the current number of items.

## A Dictionary

In many situations the keys are not so well behaved as in the employee database just described. The classic example is a dictionary. If you want to put every word of an English-language dictionary, from *a* to *zyzzyva* (yes, it's a word), into your computer's memory so they can be accessed quickly, a hash table is a good choice.

A similar widely used application for hash tables is in computer-language compilers, which maintain a *symbol table* in a hash table. The symbol table holds all the variable and function names made up by the programmer, along with the address where they can be found in memory. The program needs to access these names very quickly, so a hash table is the preferred data structure.

Let's say we want to store a 50,000-word English-language dictionary in main memory. You would like every word to occupy its own cell in a 50,000-cell array, so you can access the word using an index number. This will make access very fast. But what's the relationship of these index numbers to the words? Given the word *morphosis*, for example, how do we find its index number?

### Converting Words to Numbers

What we need is a system for turning a word into an appropriate index number. To begin, we know that computers use various schemes for representing individual characters as numbers. One such scheme is the ASCII code, in which *a* is 97, *b* is 98, and so on, up to 122 for *z*.

However, the ASCII code runs from 0 to 255, to accommodate capitals, punctuation, and so on. There are really only 26 letters in English words, so let's devise our own code, a simpler one that can potentially save memory space. Let's say *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z*. We'll also say a blank is 0, so we have 27 characters. (Uppercase letters aren't used in this dictionary.)

How do we combine the digits from individual letters into a number that represents an entire word? There are all sorts of approaches. We'll look at two representative ones, and their advantages and disadvantages.

### Adding the Digits

A simple approach to converting a word to a number might be to simply add the code numbers for each character. Say we want to convert the word *cats* to a number. First, we convert the characters to digits using our homemade code:

c = 3

a = 1

t = 20

s = 19

Then we add them:

3 + 1 + 20 + 19 = 43

Thus, in our dictionary the word *cats* would be stored in the array cell with index 43. All the other English words would likewise be assigned an array index calculated by this process.

How well would this work? For the sake of argument, let's restrict ourselves to 10-letter words. Then (remembering that a blank is 0), the first word in the dictionary, *a*, would be coded by

0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1

The last potential word in the dictionary would be *zzzzzzzzzz* (10 Zs). Our code obtained by adding its letters would be

26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260

Thus, the total range of word codes is from 1 to 260. Unfortunately, there are 50,000 words in the dictionary, so there aren't enough index numbers to go around. Each array element will need to hold about 192 words (50,000 divided by 260).

Clearly, this presents problems if we're thinking in terms of our one word-per-array element scheme. Maybe we could put a subarray or linked list of words at each array element. Unfortunately, such an approach would seriously degrade the access speed. Accessing the array element would be quick, but searching through the 192 words to find the one we wanted would be slow.

So our first attempt at converting words to numbers leaves something to be desired. Too many words have the same index. (For example, *was*, *tin*, *give*, *tend*, *moan*, *tick*, *bails*, *dredge*, and hundreds of other words add to 43, as *cats* does.) We conclude that this approach doesn't discriminate enough, so the resulting array has too few elements. We need to spread out the range of possible indices.

### Multiplying by Powers

Let's try a different way to map words to numbers. If our array was too small before, let's make sure it's big enough. What would happen if we created an array in which every word, in fact every potential word, from *a* to *zzzzzzzzzz*, was guaranteed to occupy its own unique array element?

To do this, we need to be sure that every character in a word contributes in a unique way to the final number.

We'll begin by thinking about an analogous situation with numbers instead of words. Recall that in an ordinary multi-digit number, each digit-position represents a value 10 times as big as the position to its right. Thus 7,546 really means

$7*1000 + 5*100 + 4*10 + 6*1$

Or, writing the multipliers as powers of 10:

$7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$

(An input routine in a computer program performs a similar series of multiplications and additions to convert a sequence of digits, entered at the keyboard, into a number stored in memory.)

In this system we break a number into its digits, multiply them by appropriate powers of 10 (because there are 10 possible digits), and add the products.

In a similar way we can decompose a word into its letters, convert the letters to their numerical equivalents, multiply them by appropriate powers of 27 (because there are 27 possible characters, including the blank), and add the results. This gives a unique number for every word.

Say we want to convert the word *cats* to a number. We convert the digits to numbers as shown earlier. Then we multiply each number by the appropriate power of 27 and add the results:

$3*27^3 + 1*27^2 + 20*27^1 + 19*27^0$

Calculating the powers gives

3*19,683 + 1*729 + 20*27 + 19*1

and multiplying the letter codes times the powers yields

59,049 + 729 + 540 + 19

which sums to 60,337.

This process does indeed generate a unique number for every potential word. We just calculated a 4-letter word. What happens with larger words? Unfortunately, the range of numbers becomes rather large. The largest 10-letter word, *zzzzzzzzzz*, translates into

$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$

Just by itself, $27^9$ is more than 7,000,000,000,000, so you can see that the sum will be huge. An array stored in memory can't possibly have this many elements.

The problem is that this scheme assigns an array element to every potential word, whether it's an actual English word or not. Thus, there are cells for *aaaaaaaaaa*, *aaaaaaaaab*, *aaaaaaaaac*, and so on, up to *zzzzzzzzzz*. Only a small fraction of these cells are necessary for real words, so most array cells are empty. This situation is shown in Figure 11.2.

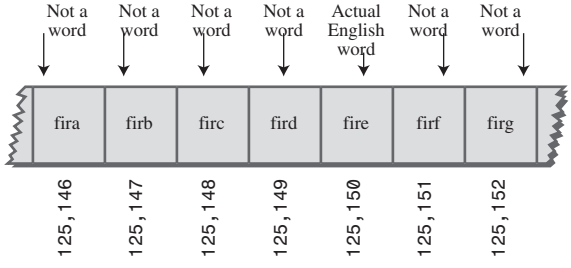


***FIGURE 11.2*** Index for every potential word.

Our first scheme—adding the numbers—generated too few indices. This latest scheme—adding the numbers times powers of 27—generates too many.

## Hashing

What we need is a way to compress the huge range of numbers we obtain from the numbers-multiplied-by-powers system into a range that matches a reasonably sized array.

How big an array are we talking about for our English dictionary? If we have only 50,000 words, you might assume our array should have approximately this many elements. However, it turns out we're going to need an array with about twice this many cells. (It will become clear later why this is so.) So we need an array with 100,000 elements.

Thus, we look for a way to squeeze a range of 0 to more than 7,000,000,000,000 into the range 0 to 100,000. A simple approach is to use the modulo operator (%), which finds the remainder when one number is divided by another.

To see how this approach works, let's look at a smaller and more comprehensible range. Suppose we squeeze numbers in the range 0 to 199 (we'll represent them by the variable `largeNumber`) into the range 0 to 9 (the variable `smallNumber`). There are 10 numbers in the range of small numbers, so we'll say that a variable `smallRange` has the value 10. It doesn't really matter what the large range is (unless it overflows the program's variable size). The Java expression for the conversion is

```
smallNumber = largeNumber % smallRange;
```

The remainders when any number is divided by 10 are always in the range 0 to 9; for example, 13%10 gives 3, and 157%10 is 7. This is shown in Figure 11.3. We've squeezed the range 0–199 into the range 0–9, a 20-to-1 compression ratio.

A similar expression can be used to compress the really huge numbers that uniquely represent every English word into index numbers that fit in our dictionary array:

```
arrayIndex = hugeNumber % arraySize;
```

This is an example of a *hash function*. It *hashes* (converts) a number in a large range into a number in a smaller range. This smaller range corresponds to the index numbers in an array. An array into which data is inserted using a hash function is called a *hash table*. (We'll talk more about the design of hash functions later in the chapter.)

To review: We convert a word into a huge number by multiplying each character in the word by an appropriate power of 27.

hugeNumber = $ch0*27^9 + ch1*27^8 + ch2*27^7 + ch3*27^6 + ch4*27^5 + ch5*27^4 + ch6*27^3 + ch7*27^2 + ch8*27^1 + ch9*27^0$

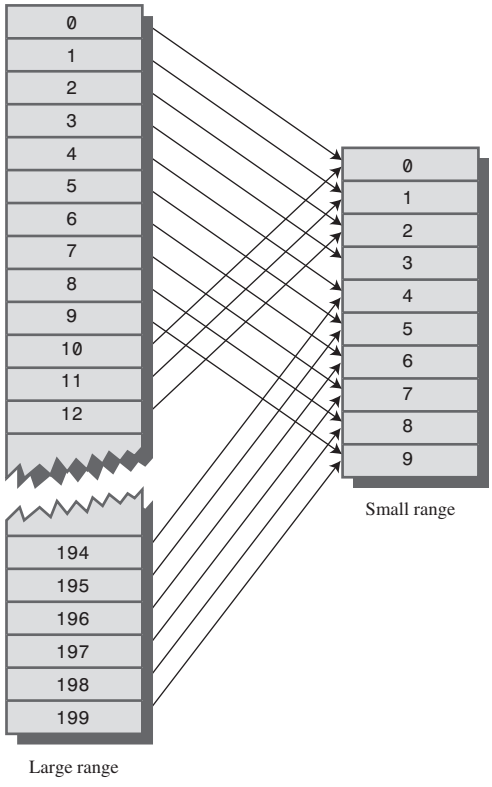


***FIGURE 11.3*** Range conversion.

Then, using the modulo operator (%), we squeeze the resulting huge range of numbers into a range about twice as big as the number of items we want to store. This is an example of a hash function:

```
arraySize = numberWords * 2;
arrayIndex = hugeNumber % arraySize;
```

In the huge range, each number represents a potential data item (an arrangement of letters), but few of these numbers represent actual data items (English words). A hash function transforms these large numbers into the index numbers of a much smaller array. In this array we expect that, on the average, there will be one word for every two cells. Some cells will have no words; and others, more than one.

A practical implementation of this scheme runs into trouble because `hugeNumber` will probably overflow its variable size, even for type `long`. We'll see how to deal with this problem later.

## Collisions

We pay a price for squeezing a large range into a small one. There's no longer a guarantee that two words won't hash to the same array index.

This is similar to what happened when we added the letter codes, but the situation is nowhere near as bad. When we added the letters, there were only 260 possible results (for words up to 10 letters). Now we're spreading this out into 50,000 possible results.

Even so, it's impossible to avoid hashing several different words into the same array location, at least occasionally. We had hoped that we could have one data item per index number, but this turns out not to be possible. The best we can do is hope that not too many words will hash to the same index.

Perhaps you want to insert the word *melioration* into the array. You hash the word to obtain its index number but find that the cell at that number is already occupied by the word *demystify*, which happens to hash to the exact same number (for a certain size array). This situation, shown in Figure 11.4, is called a *collision*.

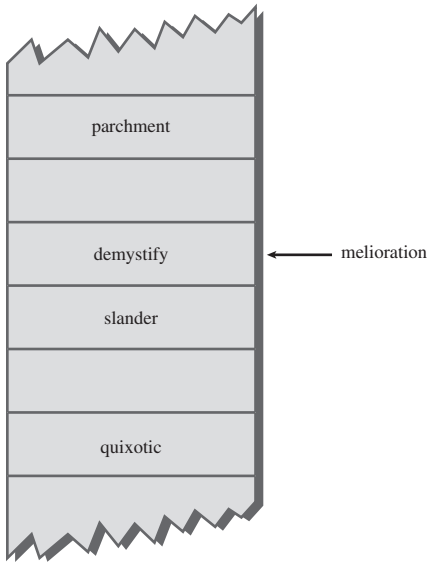


***FIGURE 11.4*** A collision.

It may appear that the possibility of collisions renders the hashing scheme impractical, but in fact we can work around the problem in a variety of ways.

Remember that we've specified an array with twice as many cells as data items. Thus, perhaps half the cells are empty. One approach, when a collision occurs, is to search the array in some systematic way for an empty cell and insert the new item there, instead of at the index specified by the hash function. This approach is called *open addressing*. If *cats* hashes to 5,421, but this location is already occupied by *parsnip*, then we might try to insert *cats* in 5,422, for example.

A second approach (mentioned earlier) is to create an array that consists of linked lists of words instead of the words themselves. Then, when a collision occurs, the new item is simply inserted in the list at that index. This is called *separate chaining*.

In the balance of this chapter we'll discuss open addressing and separate chaining, and then return to the question of hash functions.

So far we've focused on hashing strings. This is realistic, because many hash tables are used for storing strings. However, many other hash tables hold numbers, as in our employee-number example. In the discussion that follows, and in the Workshop applets, we use numbers—rather than strings—as keys. This makes things easier to understand and simplifies the programming examples. Keep in mind, however, that in many situations these numbers would be derived from strings.

## Open Addressing

In open addressing, when a data item can't be placed at the index calculated by the hash function, another location in the array is sought. We'll explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are *linear probing*, *quadratic probing*, and *double hashing*.

### Linear Probing

In linear probing we search sequentially for vacant cells. If 5,421 is occupied when we try to insert a data item there, we go to 5,422, then 5,423, and so on, incrementing the index until we find an empty cell. This is called linear probing because it steps sequentially along the line of cells.

#### The Hash Workshop Applet

The Hash Workshop applet demonstrates linear probing. When you start this applet, you'll see a screen similar to Figure 11.5.

In this applet the range of keys runs from 0 to 999. The initial size of the array is 60. The hash function has to squeeze the range of keys down to match the array size. It does this with the modulo operator (%), as we've seen before:

```
arrayIndex = key % arraySize;
```

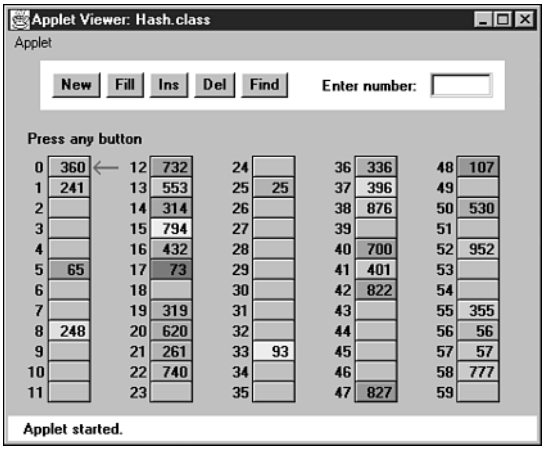


***FIGURE 11.5*** The Hash Workshop applet at startup.

For the initial array size of 60, this is

```
arrayIndex = key % 60;
```

This hash function is simple enough that you can solve it mentally. For a given key, keep subtracting 60 until you get a number less than 60. For example, to hash 143, subtract 60, giving 83, and then 60 again, giving 23. This is the index number where the algorithm will place 143. Thus, you can easily check that the algorithm has hashed a key to the correct address. (An array size of 10 is even easier to figure out, as a key's last digit is the index it will hash to.)

As with other applets, operations are carried out by repeatedly pressing the same button. For example, to find a data item with a specified number, click the Find button repeatedly. Remember, finish a sequence with one button before using another button. For example, don't switch from clicking Fill to some other button until the `Press any key` message is displayed.

All the operations require you to type a numerical value at the beginning of the sequence. The Find button requires you to type a key value, for example, while New requires the size of the new table.

**The New Button** You can create a new hash table of a size you specify by using the New button. The maximum size is 60; this limitation results from the number of cells that can be viewed in the applet window. The initial size is also 60. We use this number because it makes it easy to check whether the hash values are correct, but as we'll see later, in a general-purpose hash table, the array size should be a prime number, so 59 would be a better choice.

**The Fill Button** Initially, the hash table contains 30 items, so it's half full. However, you can also fill it with a specified number of data items using the Fill button. Keep clicking Fill, and when prompted, type the number of items to fill. Hash tables work best when they are not more than half or at the most two-thirds full (40 items in a 60-cell table).

You'll see that the filled cells aren't evenly distributed in the cells. Sometimes there's a sequence of several empty cells and sometimes a sequence of filled cells.

Let's call a sequence of filled cells in a hash table a *filled sequence*. As you add more and more items, the filled sequences become longer. This is called *clustering*, and is shown in Figure 11.6.

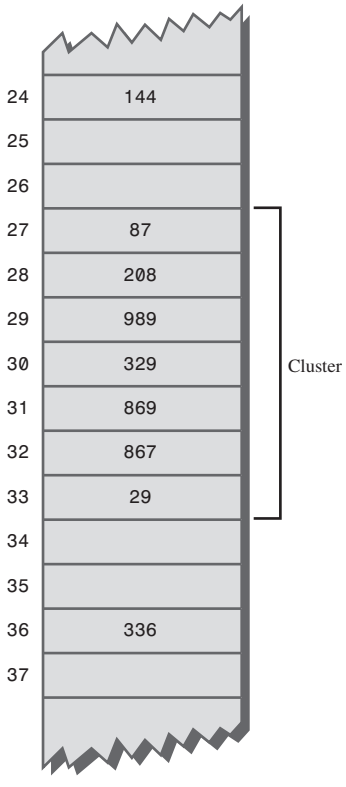


***FIGURE 11.6*** An example of clustering.

When you use the applet, note that it may take a long time to fill a hash table if you try to fill it too full (for example, if you try to put 59 items in a 60-cell table). You may think the program has stopped, but be patient. It's extremely inefficient at filling an almost-full array.

Also, note that if the hash table becomes completely full, the algorithms all stop working; in this applet they assume that the table has at least one empty cell.

**The Find Button** The Find button starts by applying the hash function to the key value you type into the number box. This results in an array index. The cell at this index may be the key you're looking for; this is the optimum situation, and success will be reported immediately.

However, it's also possible that this cell is already occupied by a data item with some other key. This is a collision; you'll see the red arrow pointing to an occupied cell. Following a collision, the search algorithm will look at the next cell in sequence. The process of finding an appropriate cell following a collision is called a *probe*.

Following a collision, the Find algorithm simply steps along the array looking at each cell in sequence. If it encounters an empty cell before finding the key it's looking for, it knows the search has failed. There's no use looking further because the insertion algorithm would have inserted the item at this cell (if not earlier). Figure 11.7 shows successful and unsuccessful linear probes.

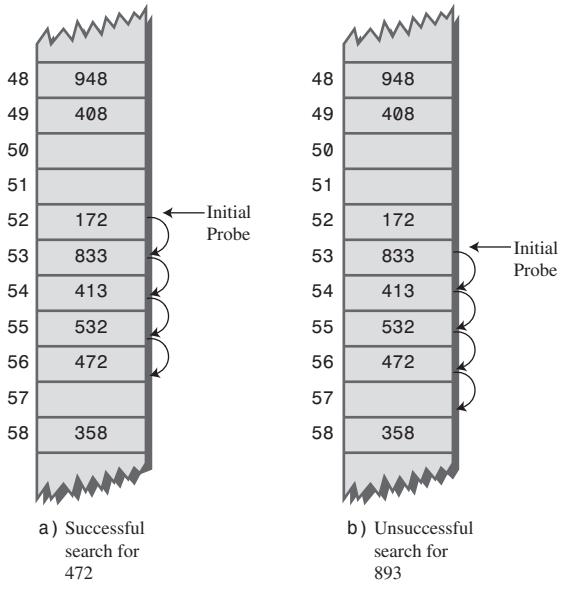


***FIGURE 11.7*** Linear probes.

**The Ins Button** The Ins button inserts a data item, with a key value that you type into the number box, into the hash table. It uses the same algorithm as the Find button to locate the appropriate cell. If the original cell is occupied, it will probe linearly for a vacant cell. When it finds one, it inserts the item.

Try inserting some new data items. Type in a three-digit number and watch what happens. Most items will go into the first cell they try, but some will suffer collisions and need to step along to find an empty cell. The number of steps they take is the *probe length*. Most probe lengths are only a few cells long. Sometimes, however, you may see probe lengths of four or five cells, or even longer as the array becomes excessively full.

Notice which keys hash to the same index. If the array size is 60, the keys 7, 67, 127, 187, 247, and so on up to 967 all hash to index 7. Try inserting this sequence or a similar one. Such sequences will demonstrate the linear probe.

**The Del Button** The Del button deletes an item whose key is typed by the user. Deletion isn't accomplished by simply removing a data item from a cell, leaving it empty. Why not? Remember that during insertion the probe process steps along a series of cells, looking for a vacant one. If a cell is made empty in the middle of this sequence of full cells, the Find routine will give up when it sees the empty cell, even if the desired cell can eventually be reached.

For this reason a deleted item is replaced by an item with a special key value that identifies it as deleted. In this applet we assume all legitimate key values are positive, so the deleted value is chosen as –1. Deleted items are marked with the special key `*Del*`.

The Insert button will insert a new item at the first available empty cell or in a `*Del*` item. The Find button will treat a `*Del*` item as an existing item for the purposes of searching for another item further along.

If there are many deletions, the hash table fills up with these ersatz `*Del*` data items, which makes it less efficient. For this reason many hash table implementations don't allow deletion. If it is implemented, it should be used sparingly.

### Duplicates Allowed?

Can you allow data items with duplicate keys to be used in hash tables? The fill routine in the Hash Workshop applet doesn't allow duplicates, but you can insert them with the Insert button if you like. Then you'll see that only the first one can be accessed. The only way to access a second item with the same key is to delete the first one. This isn't too convenient.

You could rewrite the Find algorithm to look for all items with the same key instead of just the first one. However, it would then need to search through all the cells of every linear sequence it encountered. This wastes time for all table accesses, even when no duplicates are involved. In the majority of cases you probably want to forbid duplicates.

### Clustering

Try inserting more items into the hash table in the Hash Workshop applet. As it gets more full, the clusters grow larger. Clustering can result in very long probe lengths. This means that accessing cells at the end of the sequence is very slow.

The more full the array is, the worse clustering becomes. It's not usually a problem when the array is half full, and still not too bad when it's two-thirds full. Beyond this, however, performance degrades seriously as the clusters grow larger and larger. For this reason it's critical when designing a hash table to ensure that it never becomes more than half, or at the most two-thirds, full. (We'll discuss the mathematical relationship between how full the hash table is and probe lengths at the end of this chapter.)

## Java Code for a Linear Probe Hash Table

It's not hard to create methods to handle search, insertion, and deletion with linear probe hash tables. We'll show the Java code for these methods and then a complete `hash.java` program that puts them in context.

### The `find()` Method

The `find()` method first calls `hashFunc()` to hash the search key to obtain the index number `hashVal`. The `hashFunc()` method applies the % operator to the search key and the array size, as we've seen before.

Next, in a `while` condition, `find()` checks whether the item at this index is empty (`null`). If not, it checks whether the item contains the search key. If the item does contain the key, `find()` returns the item. If it doesn't, `find()` increments `hashVal` and goes back to the top of the `while` loop to check whether the next cell is occupied. Here's the code for `find()`:

```
public DataItem find(int key)    // find item with key
// (assumes table not full)
   {
   int hashVal = hashFunc(key);  // hash the key

   while(hashArray[hashVal] != null)  // until empty cell,
      {                               // found the key?
      if(hashArray[hashVal].getKey() == key)
         return hashArray[hashVal];   // yes, return item
      ++hashVal;                 // go to next cell
      hashVal %= arraySize;      // wrap around if necessary
      }
   return null;                  // can't find item
   }
```

As hashVal steps through the array, it eventually reaches the end. When this happens, we want it to wrap around to the beginning. We could check for this with an if statement, setting hashVal to 0 whenever it equaled the array size. However, we can accomplish the same thing by applying the % operator to hashVal and the array size.

Cautious programmers might not want to assume the table is not full, as is done here. The table should not be allowed to become full, but if it did, this method would loop forever. For simplicity we don't check for this situation.

### The insert() Method

The insert() method, shown here, uses about the same algorithm as find() to locate where a data item should go. However, it's looking for an empty cell or a deleted item (key –1), rather than a specific item. When such an empty cell has been located, insert() places the new item into it.

```
public void insert(DataItem item) // insert a DataItem
// (assumes table not full)
   {
   int key = item.getKey();      // extract key
   int hashVal = hashFunc(key);  // hash the key
                                 // until empty cell or -1,
   while(hashArray[hashVal] != null &&
                              hashArray[hashVal].iData != -1)
      {
      ++hashVal;                 // go to next cell
      hashVal %= arraySize;      // wrap around if necessary
      }
   hashArray[hashVal] = item;    // insert item
   }  // end insert()
```

### The delete() Method

The following delete() method finds an existing item using code similar to find(). When the item is found, delete() writes over it with the special data item nonItem, which is predefined with a key of –1.

```
public DataItem delete(int key)  // delete a DataItem
   {
   int hashVal = hashFunc(key);  // hash the key

   while(hashArray[hashVal] != null)  // until empty cell,
      {                               // found the key?
      if(hashArray[hashVal].getKey() == key)
```

```
         {
         DataItem temp = hashArray[hashVal]; // save item
         hashArray[hashVal] = nonItem;       // delete item
         return temp;                        // return item
         }
      ++hashVal;                 // go to next cell
      hashVal %= arraySize;      // wrap around if necessary
      }
   return null;                  // can't find item
   }  // end delete()
```

### The `hash.java` Program

Listing 11.1 shows the complete `hash.java` program. In this program a `DataItem` object contains just one field, an integer that is its key. As in other data structures we've discussed, these objects could contain more data or a reference to an object of another class (such as `employee` or `partNumber`).

The major field in class `HashTable` is an array called `hashArray`. Other fields are the size of the array and the special `nonItem` object used for deletions.

***LISTING 11.1*** The `hash.java` Program

```
// hash.java
// demonstrates hash table with linear probing
// to run this program: C:>java HashTableApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                              // (could have more data)
   private int iData;             // data item (key)
//-----------------------------------------------------------
   public DataItem(int ii)        // constructor
      { iData = ii; }
//-----------------------------------------------------------
   public int getKey()
      { return iData; }
//-----------------------------------------------------------
   }  // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;    // array holds hash table
   private int arraySize;
```

***LISTING 11.1*** Continued

```
   private DataItem nonItem;        // for deleted items
// -------------------------------------------------------------
   public HashTable(int size)       // constructor
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);   // deleted item key is -1
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
      for(int j=0; j<arraySize; j++)
         {
         if(hashArray[j] != null)
            System.out.print(hashArray[j].getKey() + " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public int hashFunc(int key)
      {
      return key % arraySize;       // hash function
      }
// -------------------------------------------------------------
   public void insert(DataItem item) // insert a DataItem
   // (assumes table not full)
      {
      int key = item.getKey();      // extract key
      int hashVal = hashFunc(key);  // hash the key
                                    // until empty cell or -1,
      while(hashArray[hashVal] != null &&
                     hashArray[hashVal].getKey() != -1)
         {
         ++hashVal;                 // go to next cell
         hashVal %= arraySize;      // wraparound if necessary
         }
      hashArray[hashVal] = item;    // insert item
      }  // end insert()
```

***LISTING 11.1*** Continued

```
// --------------------------------------------------------------
   public DataItem delete(int key)  // delete a DataItem
      {
      int hashVal = hashFunc(key);  // hash the key

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // found the key?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem;       // delete item
            return temp;                        // return item
            }
         ++hashVal;                 // go to next cell
         hashVal %= arraySize;      // wraparound if necessary
         }
      return null;                  // can't find item
      }  // end delete()
// --------------------------------------------------------------
   public DataItem find(int key)    // find item with key
      {
      int hashVal = hashFunc(key);  // hash the key

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // found the key?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // yes, return item
         ++hashVal;                 // go to next cell
         hashVal %= arraySize;      // wraparound if necessary
         }
      return null;                  // can't find item
      }
// --------------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashTableApp
   {
   public static void main(String[] args) throws IOException
      {
      DataItem aDataItem;
      int aKey, size, n, keysPerCell;
```

***LISTING 11.1*** Continued

```
                                // get sizes
      System.out.print("Enter size of hash table: ");
      size = getInt();
      System.out.print("Enter initial number of items: ");
      n = getInt();
      keysPerCell = 10;
                                // make table
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)        // insert data
         {
         aKey = (int)(java.lang.Math.random() *
                                       keysPerCell * size);
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aDataItem);
         }

      while(true)                   // interact with user
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
               break;
            case 'i':
            System.out.print("Enter key value to insert: ");
               aKey = getInt();
               aDataItem = new DataItem(aKey);
               theHashTable.insert(aDataItem);
               break;
            case 'd':
               System.out.print("Enter key value to delete: ");
               aKey = getInt();
               theHashTable.delete(aKey);
               break;
            case 'f':
               System.out.print("Enter key value to find: ");
               aKey = getInt();
```

***LISTING 11.1*** Continued

```
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  {
                  System.out.println("Found " + aKey);
                  }
               else
                  System.out.println("Could not find " + aKey);
               break;
            default:
               System.out.print("Invalid entry\n");
            }  // end switch
         }  // end while
      }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // end class HashTableApp
////////////////////////////////////////////////////////////////
```

The `main()` routine in the `HashTableApp` class contains a user interface that allows the user to show the contents of the hash table (enter `s`), insert an item (`i`), delete an item (`d`), or find an item (`f`).

Initially, the program asks the user to input the size of the hash table and the number of items in it. You can make it almost any size, from a few items to 10,000. (Building larger tables than this may take a little time.) Don't use the `s` (for show) option on tables of more than a few hundred items; they scroll off the screen and displaying them takes a long time.

A variable in `main()`, `keysPerCell`, specifies the ratio of the range of keys to the size of the array. In the listing, it's set to 10. This means that if you specify a table size of 20, the keys will range from 0 to 200.

If you want to see what's going on, it's best to create tables with fewer than about 20 items so that all the items can be displayed on one line. Here's some sample interaction with `hash.java`:

```
Enter size of hash table: 12
Enter initial number of items: 8

Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** ** 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: f
Enter key value to find: 66
Found 66

Enter first letter of show, insert, delete, or find: i
Enter key value to insert: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** 100 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: d
Enter key value to delete: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** -1 113 5 66 ** 117 ** 47
```

Key values run from 0 to 119 (12 times 10, minus 1). The ** symbol indicates that a cell is empty. The item with key 100 is inserted at location 4 (the first item is numbered 0) because 100%12 is 4. Notice how 100 changes to –1 when this item is deleted.

### Expanding the Array

One option when a hash table becomes too full is to expand its array. In Java, arrays have a fixed size and can't be expanded. Your program must create a new, larger array, and then insert the contents of the old small array into the new large one.

Remember that the hash function calculates the location of a given data item based on the array size, so items won't be located in the same place in the large array as they were in the small array. You can't therefore simply copy the items from one array to the other. You'll need to go through the old array in sequence, cell by cell, inserting each item you find into the new array with the `insert()` method. This is called *rehashing*. It's a time-consuming process, but necessary if the array is to be expanded.

The expanded array is usually made twice the size of the original array. Actually, because the array size should be a prime number, the new array will need to be a bit more than twice as big. Calculating the new array size is part of the rehashing process.

Here are some routines to help find the new array size (or the original array size, if you don't trust the user to pick a prime number, which is usually the case). You start off with the specified size and then look for the next prime larger than that. The `getPrime()` method gets the next prime larger than its argument. It calls `isPrime()` to check each of the numbers above the specified size.

```
private int getPrime(int min)   // returns 1st prime > min
      {
      for(int j = min+1; true; j++)    // for all j > min
         if( isPrime(j) )              // is j prime?
            return j;                  // yes, return it
      }
// ------------------------------------------------------------
   private boolean isPrime(int n)      // is n prime?
      {
      for(int j=2; (j*j <= n); j++)    // for all j
         if( n % j == 0)               // divides evenly by j?
            return false;              // yes, so not prime
      return true;                     // no, so prime
      }
```

These routines are not the ultimate in sophistication. For example, in `getPrime()` you could check 2 and then odd numbers from then on, instead of every number. However, such refinements don't matter much because you usually find a prime after checking only a few numbers.

Java offers a class `Vector` that is an array-like data structure that can be expanded. However, it's not much help because of the need to rehash all data items when the table changes size.

## Quadratic Probing

We've seen that clusters can occur in the linear probe approach to open addressing. Once a cluster forms, it tends to grow larger. Items that hash to any value in the range of the cluster will step along and insert themselves at the end of the cluster, thus making it even bigger. The bigger the cluster gets, the faster it grows.

It's like the crowd that gathers when someone faints at the shopping mall. The first arrivals come because they saw the victim fall; later arrivals gather because they wondered what everyone else was looking at. The larger the crowd grows, the more people are attracted to it.

The ratio of the number of items in a table to the table's size is called the *load factor*. A table with 10,000 cells and 6,667 items has a load factor of 2/3.

```
loadFactor = nItems / arraySize;
```

Clusters can form even when the load factor isn't high. Parts of the hash table may consist of big clusters, while others are sparsely inhabited. Clusters reduce performance.

Quadratic probing is an attempt to keep clusters from forming. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

### The Step Is the Square of the Step Number

In a linear probe, if the primary hash index is x, subsequent probes go to x+1, x+2, x+3, and so on. In quadratic probing, probes go to x+1, x+4, x+9, x+16, x+25, and so on. The distance from the initial probe is the square of the step number: $x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$, $x+5^2$, and so on.

Figure 11.8 shows some quadratic probes.

It's as if a quadratic probe became increasingly desperate as its search lengthened. At first it calmly picks the adjacent cell. If that's occupied, it thinks it may be in a small cluster, so it tries something 4 cells away. If that's occupied, it becomes a little concerned, thinking it may be in a larger cluster, and tries 9 cells away. If that's occupied, it feels the first tinges of panic and jumps 16 cells away. Pretty soon, it's flying hysterically all over the place, as you can see if you try searching with the HashDouble Workshop applet when the table is almost full.

### The HashDouble Applet with Quadratic Probes

The HashDouble Workshop applet allows two different kinds of collision handling: quadratic probes and double hashing. (We'll look at double hashing in the next section.) This applet generates a display much like that of the Hash Workshop applet, except that it includes radio buttons to select quadratic probing or double hashing.

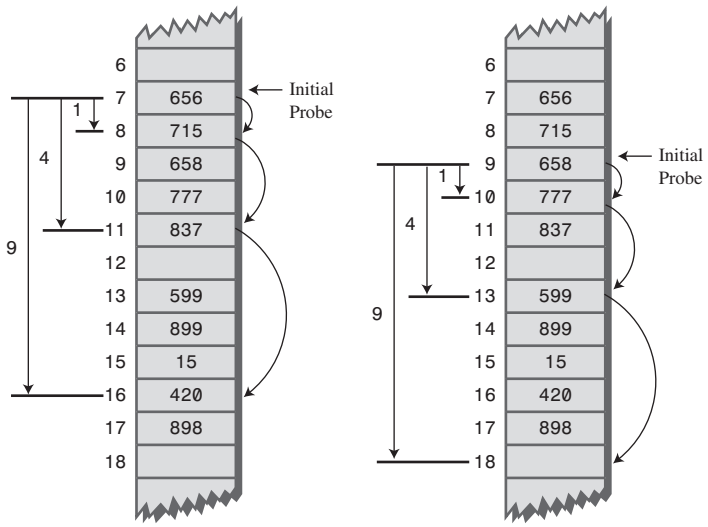


***FIGURE 11.8*** Quadratic probes.

To see how quadratic probes look, start up this applet and create a new hash table of 59 items using the New button. When you're asked to select double or quadratic probe, click the Quad button. After the new table is created, fill it four/fifths full using the Fill button (47 items in a 59-cell array). This is too full, but it will generate longer probes so you can study the probe algorithm.

Incidentally, if you try to fill the hash table too full, you may see the message `Can't complete fill`. This occurs when the probe sequences get very long. Every additional step in the probe sequence makes a bigger step size. If the sequence is too long, the step size will eventually exceed the capacity of its integer variable, so the applet shuts down the fill process before this happens.

When the table is filled, select an existing key value and use the Find key to see whether the algorithm can find it. Often the key value is located at the initial cell, or the one adjacent to it. If you're patient, however, you'll find a key that requires three or four steps, and you'll see the step size lengthen for each step. You can also use Find to search for a non-existent key; this search continues until an empty cell is encountered.

**TIP**

Important: Always make the array size a prime number. Use 59 instead of 60, for example. (Other primes less than 60 are 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, and 2.) If the array size is not prime, an endless sequence of steps may occur during a probe. If this happens during a Fill operation, the applet will be paralyzed.

### The Problem with Quadratic Probes

Quadratic probes eliminate the clustering problem we saw with the linear probe, which is called *primary clustering*. However, quadratic probes suffer from a different and more subtle clustering problem. This occurs because all the keys that hash to a particular cell follow the same sequence in trying to find a vacant space.

Let's say 184, 302, 420, and 544 all hash to 7 and are inserted in this order. Then 302 will require a one-step probe, 420 will require a four-step probe, and 544 will require a nine-step probe. Each additional item with a key that hashes to 7 will require a longer probe. This phenomenon is called *secondary clustering*.

Secondary clustering is not a serious problem, but quadratic probing is not often used because there's a slightly better solution.

## Double Hashing

To eliminate secondary clustering as well as primary clustering, we can use another approach: *double hashing*. Secondary clustering occurs because the algorithm that generates the sequence of steps in the quadratic probe always generates the same steps: 1, 4, 9, 16, and so on.

What we need is a way to generate probe sequences that depend on the key instead of being the same for every key. Then numbers with different keys that hash to the same index will use different probe sequences.

The solution is to hash the key a second time, using a different hash function, and use the result as the step size. For a given key the step size remains constant throughout a probe, but it's different for different keys.

Experience has shown that this secondary hash function must have certain characteristics:

- It must not be the same as the primary hash function.
- It must never output a 0 (otherwise, there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop).

Experts have discovered that functions of the following form work well:

```
stepSize = constant - (key % constant);
```

where `constant` is prime and smaller than the array size. For example,

```
stepSize = 5 - (key % 5);
```

This is the secondary hash function used in the HashDouble Workshop applet. Different keys may hash to the same index, but they will (most likely) generate different step sizes. With this hash function the step sizes are all in the range 1 to 5. This is shown in Figure 11.9.

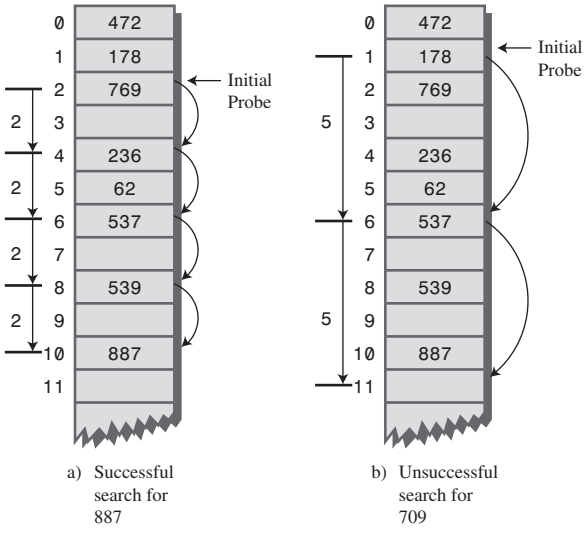


***FIGURE 11.9*** Double hashing.

### The HashDouble Applet with Double Hashing

You can use the HashDouble Workshop applet to see how double hashing works. It starts up automatically in Double-hashing mode, but if it's in Quadratic mode, you can switch to Double by creating a new table with the New button and clicking the Double button when prompted. To best see probes at work, you'll need to fill the table rather full, say to about nine/tenths capacity or more. Even with such high load factors, most data items will be found immediately by the first hash function; only a few will require extended probe sequences.

Try finding existing keys. When one needs a probe sequence, you'll see how all the steps are the same size for a given key, but that the step size is different—between 1 and 5—for different keys.

### Java Code for Double Hashing

Listing 11.2 shows the complete listing for `hashDouble.java`, which uses double hashing. It's similar to the `hash.java` program (Listing 11.1), but uses two hash functions: one for finding the initial index and the second for generating the step size. As before, the user can show the table contents, insert an item, delete an item, and find an item.

***LISTING 11.2*** The `hashDouble.java` Program

```
// hashDouble.java
// demonstrates hash table with double hashing
// to run this program: C:>java HashDoubleApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                                // (could have more items)
   private int iData;               // data item (key)
//-------------------------------------------------------------
   public DataItem(int ii)          // constructor
      { iData = ii; }
//-------------------------------------------------------------
   public int getKey()
      { return iData; }
//-------------------------------------------------------------
   }  // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;     // array is the hash table
   private int arraySize;
   private DataItem nonItem;         // for deleted items
// -------------------------------------------------------------
   HashTable(int size)               // constructor
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
      for(int j=0; j<arraySize; j++)
```

***LISTING 11.2*** Continued

```
         {
         if(hashArray[j] != null)
            System.out.print(hashArray[j].getKey()+ " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// --------------------------------------------------------------
   public int hashFunc1(int key)
      {
      return key % arraySize;
      }
// --------------------------------------------------------------
   public int hashFunc2(int key)
      {
      // non-zero, less than array size, different from hF1
      // array size must be relatively prime to 5, 4, 3, and 2
      return 5 - key % 5;
      }
// --------------------------------------------------------------
                                    // insert a DataItem
   public void insert(int key, DataItem item)
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);  // hash the key
      int stepSize = hashFunc2(key); // get step size
                                    // until empty cell or -1
      while(hashArray[hashVal] != null &&
                     hashArray[hashVal].getKey() != -1)
         {
         hashVal += stepSize;        // add the step
         hashVal %= arraySize;       // for wraparound
         }
      hashArray[hashVal] = item;     // insert item
      }  // end insert()
// --------------------------------------------------------------
   public DataItem delete(int key)   // delete a DataItem
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size
```

*LISTING 11.2* Continued

```
      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem;       // delete item
            return temp;                        // return item
            }
         hashVal += stepSize;            // add the step
         hashVal %= arraySize;           // for wraparound
         }
      return null;                   // can't find item
      }  // end delete()
// ---------------------------------------------------------
   public DataItem find(int key)     // find item with key
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // yes, return item
         hashVal += stepSize;            // add the step
         hashVal %= arraySize;           // for wraparound
         }
      return null;                   // can't find item
      }
// ---------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashDoubleApp
   {
   public static void main(String[] args) throws IOException
      {
      int aKey;
      DataItem aDataItem;
      int size, n;
                                 // get sizes
      System.out.print("Enter size of hash table: ");
```

***LISTING 11.2*** Continued

```
size = getInt();
System.out.print("Enter initial number of items: ");
n = getInt();
                            // make table
HashTable theHashTable = new HashTable(size);

for(int j=0; j<n; j++)      // insert data
   {
   aKey = (int)(java.lang.Math.random() * 2 * size);
   aDataItem = new DataItem(aKey);
   theHashTable.insert(aKey, aDataItem);
   }

while(true)                 // interact with user
   {
   System.out.print("Enter first letter of ");
   System.out.print("show, insert, delete, or find: ");
   char choice = getChar();
   switch(choice)
      {
      case 's':
         theHashTable.displayTable();
         break;
      case 'i':
         System.out.print("Enter key value to insert: ");
         aKey = getInt();
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aKey, aDataItem);
         break;
      case 'd':
         System.out.print("Enter key value to delete: ");
         aKey = getInt();
         theHashTable.delete(aKey);
         break;
      case 'f':
         System.out.print("Enter key value to find: ");
         aKey = getInt();
         aDataItem = theHashTable.find(aKey);
         if(aDataItem != null)
            System.out.println("Found " + aKey);
         else
```

*LISTING 11.2* Continued

```
                System.out.println("Could not find " + aKey);
             break;
          default:
             System.out.print("Invalid entry\n");
          }  // end switch
       }  // end while
    }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // end class HashDoubleApp
////////////////////////////////////////////////////////////////
```

Output and operation of this program are similar to those of `hash.java`. Table 11.1 shows what happens when 21 items are inserted into a 23-cell hash table using double hashing. The step sizes run from 1 to 5.

*TABLE 11.1* Filling a 23-Cell Table Using Double Hashing

| Item Number | Key | Hash Value | Step Size | Cells in Probe Sequence |
|---|---|---|---|---|
| 1 | 1 | 1 | 4 | |
| 2 | 38 | 15 | 2 | |
| 3 | 37 | 14 | 3 | |
| 4 | 16 | 16 | 4 | |

*TABLE 11.1* Continued

| Item Number | Key | Hash Value | Step Size | Cells in Probe Sequence |
|---|---|---|---|---|
| 5 | 20 | 20 | 5 | |
| 6 | 3 | 3 | 2 | |
| 7 | 11 | 11 | 4 | |
| 8 | 24 | 1 | 1 | 2 |
| 9 | 5 | 5 | 5 | |
| 10 | 16 | 16 | 4 | 20 1 5 9 |
| 11 | 10 | 10 | 5 | |
| 12 | 31 | 8 | 4 | |
| 13 | 18 | 18 | 2 | |
| 14 | 12 | 12 | 3 | |
| 15 | 30 | 7 | 5 | |
| 16 | 1 | 1 | 4 | 5 9 13 |
| 17 | 19 | 19 | 1 | |
| 18 | 36 | 13 | 4 | 17 |
| 19 | 41 | 18 | 4 | 22 |
| 20 | 15 | 15 | 5 | 20 2 7 12 17 22 4 |
| 21 | 25 | 2 | 5 | 7 12 17 22 4 9 14 19 1 6 |

The first 15 keys mostly hash to a vacant cell (the 10th one is an anomaly). After that, as the array gets more full, the probe sequences become quite long. Here's the resulting array of keys:

```
** 1 24 3 15 5 25 30 31 16 10 11 12 1 37 38 16 36 18 19 20 ** 41
```

### Table Size a Prime Number

Double hashing requires that the size of the hash table is a prime number. To see why, imagine a situation in which the table size is not a prime number. For example, suppose the array size is 15 (indices from 0 to 14), and that a particular key hashes to an initial index of 0 and a step size of 5. The probe sequence will be 0, 5, 10, 0, 5, 10, and so on, repeating endlessly. Only these three cells are ever examined, so the algorithm will never find the empty cells that might be waiting at 1, 2, 3, and so on. The algorithm will crash and burn.

If the array size were 13, which is prime, the probe sequence will eventually visit every cell. It's 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, and so on and on. If there is even one empty cell, the probe will find it. Using a prime number as the array size makes it impossible for any number to divide it evenly, so the probe sequence will eventually check every cell.

A similar effect occurs using the quadratic probe. In that case, however, the step size gets larger with each step and will eventually overflow the variable holding it, thus preventing an endless loop.

In general, double hashing is the probe sequence of choice when open addressing is used.

## Separate Chaining

In open addressing, collisions are resolved by looking for an open cell in the hash table. A different approach is to install a linked list at each index in the hash table. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list; there's no need to search for empty cells in the primary array. Figure 11.10 shows how separate chaining looks.

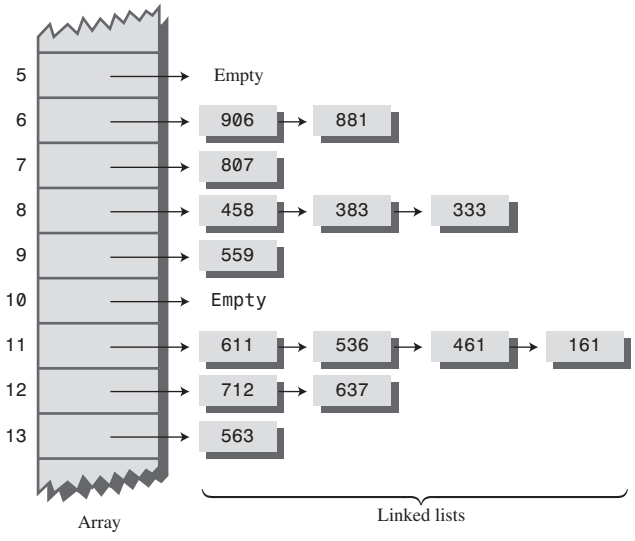


***FIGURE 11.10*** Example of separate chaining.

Separate chaining is conceptually somewhat simpler than the various probe schemes used in open addressing. However, the code is longer because it must include the mechanism for the linked lists, usually in the form of an additional class.

### The HashChain Workshop Applet

To see how separate chaining works, start the HashChain Workshop applet. It displays an array of linked lists, as shown in Figure 11.11.

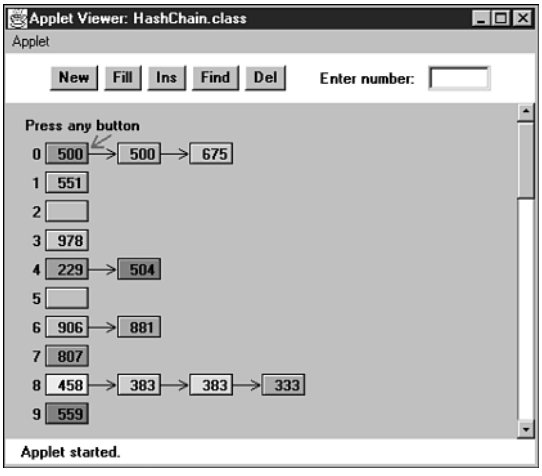


*FIGURE 11.11* Separate chaining in the HashChain Workshop applet.

Each element of the array occupies one line of the display, and the linked lists extend from left to right. Initially, there are 25 cells in the array (25 lists). This is more than fits on the screen; you can move the display up and down with the scrollbar to see the entire array. The display shows up to six items per list. You can create a hash table with up to 100 lists and use load factors up to 2.0. Higher load factors may cause the linked lists to exceed six items and run off the right edge of the screen, making it impossible to see all the items. (This may happen very occasionally even at the 2.0 load factor.)

Experiment with the HashChain applet by inserting some new items with the Ins button. You'll see how the red arrow goes immediately to the correct list and inserts the item at the beginning of the list. The lists in the HashChain applet are not sorted, so insertion does not require searching through the list. (The example program will demonstrate sorted lists.)

Try to find specified items using the Find button. During a Find operation, if there are several items on the list, the red arrow must step through the items looking for the correct one. For a successful search, half the items in the list must be examined on the average, as we discussed in Chapter 5, "Linked Lists." For an unsuccessful search, all the items must be examined.

### Load Factors

The load factor (the ratio of the number of items in a hash table to its size) is typically different in separate chaining than in open addressing. In separate chaining it's normal to put N or more items into an N cell array; thus, the load factor can be 1 or greater. There's no problem with this; some locations will simply contain two or more items in their lists.

Of course, if there are many items on the lists, access time is reduced because access to a specified item requires searching through an average of half the items on the list. Finding the initial cell takes fast O(1) time, but searching through a list takes time proportional to M, the average number of items on the list. This is O(M) time. Thus, we don't want the lists to become too full.

A load factor of 1, as shown in the initial Workshop applet, is common. With this load factor, roughly one-third of the cells will be empty, one-third will hold one item, and one-third will hold two or more items.

In open addressing, performance degrades badly as the load factor increases above one-half or two-thirds. In separate chaining the load factor can rise above 1 without hurting performance very much. This makes separate chaining a more robust mechanism, especially when it's hard to predict in advance how much data will be placed in the hash table.

#### Duplicates

Duplicates are allowed and may be generated in the Fill process. All items with the same key will be inserted in the same list, so if you need to discover all of them, you must search the entire list in both successful and unsuccessful searches. This lowers performance. The Find operation in the applet finds only the first of several duplicates.

#### Deletion

In separate chaining, deletion poses no special problems as it does in open addressing. The algorithm hashes to the proper list and then deletes the item from the list. Because probes aren't used, it doesn't matter if the list at a particular cell becomes empty. We've included a Del button in the Workshop applet to show how deletion works.

#### Table Size

With separate chaining, making the table size a prime number is not as important as it is with quadratic probes and double hashing. There are no probes in separate chaining, so we don't need to worry that a probe will go into an endless sequence because the step size divides evenly into the array size.

On the other hand, certain kinds of key distributions can cause data to cluster when the array size is not a prime number. We'll have more to say about this problem when we discuss hash functions.

#### Buckets

Another approach similar to separate chaining is to use an array at each location in the hash table, instead of a linked list. Such arrays are sometimes called *buckets*. This approach is not as efficient as the linked list approach, however, because of the

problem of choosing the size of the buckets. If they're too small, they may overflow, and if they're too large, they waste memory. Linked lists, which allocate memory dynamically, don't have this problem.

## Java Code for Separate Chaining

The `hashChain.java` program includes a `SortedList` class and an associated `Link` class. Sorted lists don't speed up a successful search, but they do cut the time of an unsuccessful search in half. (As soon as an item larger than the search key is reached, which on average is half the items in a list, the search can be declared a failure.)

Deletion times are also cut in half; however, insertion times are lengthened because the new item can't just be inserted at the beginning of the list; its proper place in the ordered list must be located before it's inserted. If the lists are short, the increase in insertion times may not be important.

If many unsuccessful searches are anticipated, it may be worthwhile to use the slightly more complicated sorted list, rather than an unsorted list. However, an unsorted list is preferred if insertion speed is more important.

The `hashChain.java` program, shown in Listing 11.3, begins by constructing a hash table with a table size and number of items entered by the user. The user can then insert, find, and delete items, and display the list. For the entire hash table to be viewed on the screen, the size of the table must be no greater than 16 or so.

***LISTING 11.3*** The `hashChain.java` Program

```
// hashChain.java
// demonstrates hash table with separate chaining
// to run this program: C:>java HashChainApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Link
   {                                 // (could be other items)
   private int iData;                // data item
   public Link next;                 // next link in list
// -------------------------------------------------------------
   public Link(int it)               // constructor
      { iData= it; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void displayLink()         // display this link
      { System.out.print(iData + " "); }
```

***LISTING 11.3*** Continued

```
   }  // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;               // ref to first list item
// -------------------------------------------------------------
   public void SortedList()          // constructor
      { first = null; }
// -------------------------------------------------------------
   public void insert(Link theLink)  // insert link, in order
      {
      int key = theLink.getKey();
      Link previous = null;          // start at first
      Link current = first;
                                     // until end of list,
      while( current != null && key > current.getKey() )
         {                           // or current > key,
         previous = current;
         current = current.next;     // go to next item
         }
      if(previous==null)             // if beginning of list,
         first = theLink;            //    first --> new link
      else                           // not at beginning,
         previous.next = theLink;    //    prev --> new link
      theLink.next = current;        // new link --> current
      }  // end insert()
// -------------------------------------------------------------
   public void delete(int key)       // delete link
      {                              // (assumes non-empty list)
      Link previous = null;          // start at first
      Link current = first;
                                     // until end of list,
      while( current != null && key != current.getKey() )
         {                           // or key == current,
         previous = current;
         current = current.next;     // go to next link
         }
                                     // disconnect link
      if(previous==null)             //   if beginning of list
         first = first.next;         //      delete first link
      else                           //   not at beginning
```

***LISTING 11.3*** Continued

```
         previous.next = current.next; //    delete current link
      }  // end delete()
// -------------------------------------------------------------
   public Link find(int key)         // find link
      {
      Link current = first;          // start at first
                                     // until end of list,
      while(current != null &&  current.getKey() <= key)
         {                           // or key too small,
         if(current.getKey() == key)    // is this the link?
            return current;          // found it, return link
         current = current.next;     // go to next item
         }
      return null;                   // didn't find it
      }  // end find()
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
   }  // end class SortedList
////////////////////////////////////////////////////////////////
class HashTable
   {
   private SortedList[] hashArray;   // array of lists
   private int arraySize;
// -------------------------------------------------------------
   public HashTable(int size)        // constructor
      {
      arraySize = size;
      hashArray = new SortedList[arraySize];  // create array
      for(int j=0; j<arraySize; j++)          // fill array
         hashArray[j] = new SortedList();     // with lists
      }
```

***LISTING 11.3*** Continued

```
// --------------------------------------------------------------
   public void displayTable()
      {
      for(int j=0; j<arraySize; j++) // for each cell,
         {
         System.out.print(j + ". "); // display cell number
         hashArray[j].displayList(); // display list
         }
      }
// --------------------------------------------------------------
   public int hashFunc(int key)      // hash function
      {
      return key % arraySize;
      }
// --------------------------------------------------------------
   public void insert(Link theLink)  // insert a link
      {
      int key = theLink.getKey();
      int hashVal = hashFunc(key);   // hash the key
      hashArray[hashVal].insert(theLink); // insert at hashVal
      }  // end insert()
// --------------------------------------------------------------
   public void delete(int key)       // delete a link
      {
      int hashVal = hashFunc(key);   // hash the key
      hashArray[hashVal].delete(key); // delete link
      }  // end delete()
// --------------------------------------------------------------
   public Link find(int key)         // find link
      {
      int hashVal = hashFunc(key);   // hash the key
      Link theLink = hashArray[hashVal].find(key);  // get link
      return theLink;                // return link
      }
// --------------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashChainApp
   {
   public static void main(String[] args) throws IOException
      {
```

***LISTING 11.3*** Continued

```
      int aKey;
      Link aDataItem;
      int size, n, keysPerCell = 100;
                                    // get sizes
      System.out.print("Enter size of hash table: ");
      size = getInt();
      System.out.print("Enter initial number of items: ");
      n = getInt();
                                    // make table
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)        // insert data
         {
         aKey = (int)(java.lang.Math.random() *
                                         keysPerCell * size);
         aDataItem = new Link(aKey);
         theHashTable.insert(aDataItem);
         }
      while(true)                   // interact with user
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
               break;
            case 'i':
               System.out.print("Enter key value to insert: ");
               aKey = getInt();
               aDataItem = new Link(aKey);
               theHashTable.insert(aDataItem);
               break;
            case 'd':
               System.out.print("Enter key value to delete: ");
               aKey = getInt();
               theHashTable.delete(aKey);
               break;
            case 'f':
               System.out.print("Enter key value to find: ");
```

*LISTING 11.3* Continued

```
               aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  System.out.println("Found " + aKey);
               else
                  System.out.println("Could not find " + aKey);
               break;
            default:
               System.out.print("Invalid entry\n");
            }  // end switch
         }  // end while
      }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // end class HashChainApp
////////////////////////////////////////////////////////////////
```

Here's the output when the user creates a table with 20 lists, inserts 20 items into it, and displays it with the `s` option:

```
Enter size of hash table: 20
Enter initial number of items: 20
Enter first letter of show, insert, delete, or find: s
```

```
0. List (first-->last): 240 1160
1. List (first-->last):
2. List (first-->last):
3. List (first-->last): 143
4. List (first-->last): 1004
5. List (first-->last): 1485 1585
6. List (first-->last):
7. List (first-->last): 87 1407
8. List (first-->last):
9. List (first-->last): 309
10. List (first-->last): 490
11. List (first-->last):
12. List (first-->last): 872
13. List (first-->last): 1073
14. List (first-->last): 594 954
15. List (first-->last): 335
16. List (first-->last): 1216
17. List (first-->last): 1057 1357
18. List (first-->last): 938 1818
19. List (first-->last):
```

If you insert more items into this table, you'll see the lists grow longer but maintain their sorted order. You can delete items as well.

We'll return to the question of when to use separate chaining when we discuss hash table efficiency later in this chapter.

## Hash Functions

In this section we'll explore the issue of what makes a good hash function and see whether we can improve the approach to hashing strings mentioned at the beginning of this chapter.

### Quick Computation

A good hash function is simple, so it can be computed quickly. The major advantage of hash tables is their speed. If the hash function is slow, this speed will be degraded. A hash function with many multiplications and divisions is not a good idea. (The bit-manipulation facilities of Java or C++, such as shifting bits right to divide a number by a multiple of 2, can sometimes be used to good advantage.)

The purpose of a hash function is to take a range of key values and transform them into index values in such a way that the key values are distributed randomly across all the indices of the hash table. Keys may be completely random or not so random.

## Random Keys

A so-called *perfect* hash function maps every key into a different table location. This is only possible for keys that are unusually well behaved and whose range is small enough to be used directly as array indices (as in the employee-number example at the beginning of this chapter).

In most cases neither of these situations exists, and the hash function will need to compress a larger range of keys into a smaller range of index numbers.

The distribution of key values in a particular database determines what the hash function needs to do. In this chapter we've assumed that the data was randomly distributed over its entire range. In this situation the hash function

```
index = key % arraySize;
```

is satisfactory. It involves only one mathematical operation, and if the keys are truly random, the resulting indices will be random too, and therefore well distributed.

## Non-Random Keys

However, data is often distributed non-randomly. Imagine a database that uses car-part numbers as keys. Perhaps these numbers are of the form

033-400-03-94-05-0-535

This is interpreted as follows:

- Digits 0–2: Supplier number (1 to 999, currently up to 70)
- Digits 3–5: Category code (100, 150, 200, 250, up to 850)
- Digits 6–7: Month of introduction (1 to 12)
- Digits 8–9: Year of introduction (00 to 99)
- Digits 10–11: Serial number (1 to 99, but never exceeds 100)
- Digit 12: Toxic risk flag (0 or 1)
- Digits 13–15: Checksum (sum of other fields, modulo 100)

The key used for the part number shown would be 0,334,000,394,050,535. However, such keys are not randomly distributed. The majority of numbers from 0 to 9,999,999,999,999,999 can't actually occur (for example, supplier numbers higher than 70, category codes that aren't multiples of 50, and months from 13 to 99). Also, the checksum is not independent of the other numbers. Some work should be done to these part numbers to help ensure that they form a range of more truly random numbers.

### Don't Use Non-Data

The key fields should be squeezed down until every bit counts. For example, the category codes should be changed to run from 0 to 15. Also, the checksum should be removed because it doesn't add any additional information; it's deliberately redundant. Various bit-twiddling techniques are appropriate for compressing the various fields in the key.

### Use All the Data

Every part of the key (except non-data, as just described) should contribute to the hash function. Don't just use the first four digits or some such expurgation. The more data that contributes to the key, the more likely it is that the keys will hash evenly into the entire range of indices.

Sometimes the range of keys is so large it overflows type `int` or type `long` variables. We'll see how to handle overflow when we talk about hashing strings in a moment.

To summarize: The trick is to find a hash function that's simple and fast, yet excludes the non-data parts of the key and uses all the data.

### Use a Prime Number for the Modulo Base

Often the hash function involves using the modulo operator (%) with the table size. We've already seen that it's important for the table size to be a prime number when using a quadratic probe or double hashing. However, if the keys themselves may not be randomly distributed, it's important for the table size to be a prime number no matter what hashing system is used.

This is true because, if many keys share a divisor with the array size, they may tend to hash to the same location, causing clustering. Using a prime table size eliminates this possibility. For example, if the table size is a multiple of 50 in our car-part example, the category codes will all hash to index numbers that are multiples of 50. However, with a prime number such as 53, you are guaranteed that no keys will divide evenly into the table size.

The moral is to examine your keys carefully and tailor your hash algorithm to remove any irregularity in the distribution of the keys.

## Hashing Strings

We saw at the beginning of this chapter how to convert short strings to key numbers by multiplying digit codes by powers of a constant. In particular, we saw that the four-letter word *cats* could turn into a number by calculating

$$key = 3*27^3 + 1*27^2 + 20*27^1 + 19*27^0$$

This approach has the desirable attribute of involving all the characters in the input string. The calculated key value can then be hashed into an array index in the usual way:

```
index = (key) % arraySize;
```

Here's a Java method that finds the key value of a string:

```
public static int hashFunc1(String key)
   {
   int hashVal = 0;
   int pow27 = 1;                        // 1, 27, 27*27, etc

   for(int j=key.length()-1; j>=0; j--) // right to left
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal += pow27 * letter;        // times power of 27
      pow27 *= 27;                      // next power of 27
      }
   return hashVal % arraySize;
   }  // end hashFunc1()
```

The loop starts at the rightmost letter in the word. If there are N letters, this is N-1. The numerical equivalent of the letter, according to the code we devised at the beginning of this chapter (a=1 and so on), is placed in `letter`. This is then multiplied by a power of 27, which is 1 for the letter at N-1, 27 for the letter at N-2, and so on.

The `hashFunc1()` method is not as efficient as it might be. Aside from the character conversion, there are two multiplications and an addition inside the loop. We can eliminate a multiplication by taking advantage of a mathematical identity called Horner's method. (Horner was an English mathematician, 1773–1827.) This states that an expression like

$$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$$

can be written as

```
(((var4*n + var3)*n + var2)*n + var1)*n + var0
```

To evaluate this equation, we can start inside the innermost parentheses and work outward. If we translate this to a Java method, we have the following code:

```
public static int hashFunc2(String key)
      {
      int hashVal = key.charAt(0) - 96;
```

```
    for(int j=1; j<key.length(); j++)    // left to right
       {
       int letter = key.charAt(j) - 96;  // get char code
       hashVal = hashVal * 27 + letter;  // multiply and add
       }
    return hashVal % arraySize;          // mod
    }  // end hashFunc2()
```

Here we start with the leftmost letter of the word (which is somewhat more natural than starting on the right), and we have only one multiplication and one addition each time through the loop (aside from extracting the character from the string).

The `hashFunc2()` method unfortunately can't handle strings longer than about seven letters. Longer strings cause the value of `hashVal` to exceed the size of type `int`. (If we used type `long`, the same problem would still arise for somewhat longer strings.)

Can we modify this basic approach so we don't overflow any variables? Notice that the key we eventually end up with is always less than the array size because we apply the modulo operator. It's not the final index that's too big; it's the intermediate key values.

It turns out that with Horner's formulation we can apply the modulo operator (%) at each step in the calculation. This gives the same result as applying the modulo operator once at the end but avoids overflow. (It does add an operation inside the loop.) The `hashFunc3()` method shows how this looks:

```
public static int hashFunc3(String key)
   {
   int hashVal = 0;
   for(int j=0; j<key.length(); j++)    // left to right
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal = (hashVal * 27 + letter) % arraySize; // mod
      }
   return hashVal;                      // no mod
   }  // end hashFunc3()
```

This approach or something like it is normally taken to hash a string. Various bit-manipulation tricks can be played as well, such as using a base of 32 (or a larger power of 2) instead of 27, so that multiplication can be effected using the shift operator (>>), which is faster than the modulo operator (%).

You can use an approach similar to this to convert any kind of string to a number suitable for hashing. The strings can be words, names, or any other concatenation of characters.

### Folding

Another reasonable hash function involves breaking the key into groups of digits and adding the groups. This ensures that all the digits influence the hash value. The number of digits in a group should correspond to the size of the array. That is, for an array of 1,000 items, use groups of three digits each.

For example, suppose you want to hash nine-digit Social Security numbers for linear probing. If the array size is 1,000, you would divide the nine-digit number into three groups of three digits. If a particular SSN was 123-45-6789, you would calculate a key value of 123+456+789 = 1368. You can use the % operator to trim such sums so the highest index is 999. In this case, 1368%1000 = 368. If the array size is 100, you would need to break the nine-digit key into four two-digit numbers and one one-digit number: 12+34+56+78+9 = 189, and 189%100 = 89.

It's easier to imagine how this works when the array size is a multiple of 10. However, for best results it should be a prime number, as we've seen for other hash functions. We'll leave an implementation of this scheme as an exercise.

## Hashing Efficiency

We've noted that insertion and searching in hash tables can approach O(1) time. If no collision occurs, only a call to the hash function and a single array reference are necessary to insert a new item or find an existing item. This is the minimum access time.

If collisions occur, access times become dependent on the resulting probe lengths. Each cell accessed during a probe adds another time increment to the search for a vacant cell (for insertion) or for an existing cell. During an access, a cell must be checked to see whether it's empty, and—in the case of searching or deletion—whether it contains the desired item.

Thus, an individual search or insertion time is proportional to the length of the probe. This is in addition to a constant time for the hash function.

The average probe length (and therefore the average access time) is dependent on the load factor (the ratio of items in the table to the size of the table). As the load factor increases, probe lengths grow longer.

We'll look at the relationship between probe lengths and load factors for the various kinds of hash tables we've studied.

### Open Addressing

The loss of efficiency with high load factors is more serious for the various open addressing schemes than for separate chaining.

In open addressing, unsuccessful searches generally take longer than successful searches. During a probe sequence, the algorithm can stop as soon as it finds the desired item, which is, on the average, halfway through the probe sequence. On the other hand, it must go all the way to the end of the sequence before it's sure it can't find an item.

### Linear Probing

The following equations show the relationship between probe length (P) and load factor (L) for linear probing. For a successful search it's

$$P = ( 1 + 1 / (1 - L)^2 ) / 2$$

and for an unsuccessful search it's

$$P = ( 1 + 1 / (1 - L) ) / 2$$

These formulas are from Knuth (see Appendix B, "Further Reading"), and their derivation is quite complicated. Figure 11.12 shows the result of graphing these equations.

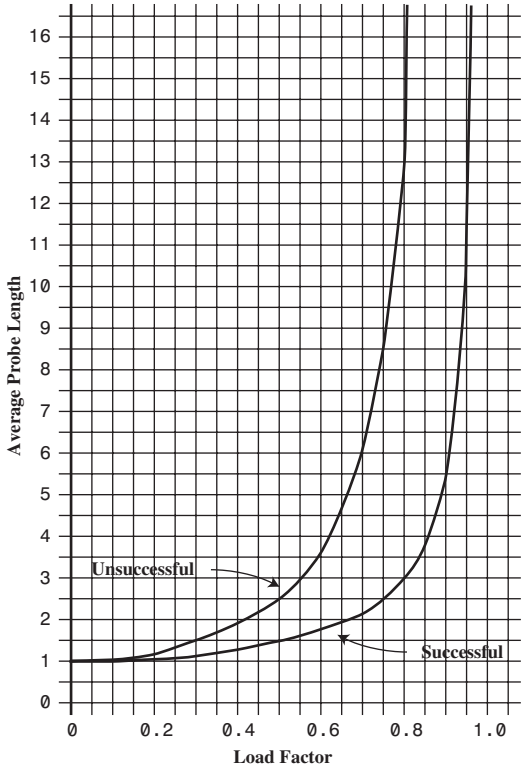


***FIGURE 11.12*** Linear probe performance.

At a load factor of 1/2, a successful search takes 1.5 comparisons and an unsuccessful search takes 2.5. At a load factor of 2/3, the numbers are 2.0 and 5.0. At higher load factors the numbers become very large.

The moral, as you can see, is that the load factor must be kept under 2/3 and preferably under 1/2. On the other hand, the lower the load factor, the more memory is needed for a given amount of data. The optimum load factor in a particular situation depends on the trade-off between memory efficiency, which decreases with lower load factors, and speed, which increases.

#### Quadratic Probing and Double Hashing

Quadratic probing and double hashing share their performance equations. These equations indicate a modest superiority over linear probing. For a successful search, the formula (again from Knuth) is

$$-\log_2(1-\text{loadFactor}) / \text{loadFactor}$$

For an unsuccessful search it is

$$1 / (1-\text{loadFactor})$$

Figure 11.13 shows graphs of these formulas. At a load factor of 0.5, successful and unsuccessful searches both require an average of two probes. At a 2/3 load factor, the numbers are 2.37 and 3.0, and at 0.8 they're 2.90 and 5.0. Thus, somewhat higher load factors can be tolerated for quadratic probing and double hashing than for linear probing.

## Separate Chaining

The efficiency analysis for separate chaining is different, and generally easier, than for open addressing.

We want to know how long it takes to search for or insert an item into a separate-chaining hash table. We'll assume that the most time-consuming part of these operations is comparing the search key of the item with the keys of other items in the list. We'll also assume that the time required to hash to the appropriate list and to determine when the end of a list has been reached is equivalent to one key comparison. Thus, all operations require `1+nComps` time, where `nComps` is the number of key comparisons.

Let's say that the hash table consists of `arraySize` elements, each of which holds a list, and that `N` data items have been inserted in the table. Then, on the average, each list will hold `N` divided by `arraySize` items:

```
AverageListLength = N / arraySize
```

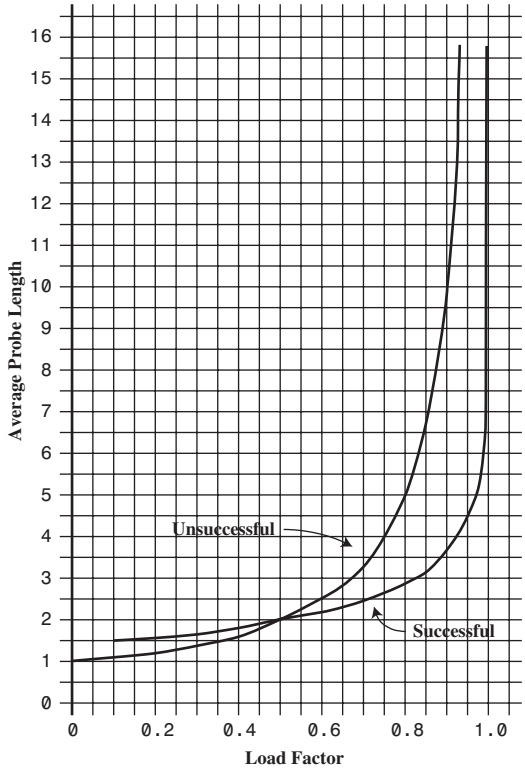


***FIGURE 11.13*** Quadratic-probe and double-hashing performance.

This is the same as the definition of the load factor:

```
loadFactor = N / arraySize
```

so the average list length equals the load factor.

### Searching

In a successful search, the algorithm hashes to the appropriate list and then searches along the list for the item. On the average, half the items must be examined before the correct one is located. Thus, the search time is

```
1 + loadFactor / 2
```

This is true whether the lists are ordered or not. In an unsuccessful search, if the lists are unordered, all the items must be searched, so the time is

```
1 + loadFactor
```

These formulas are graphed in Figure 11.14.

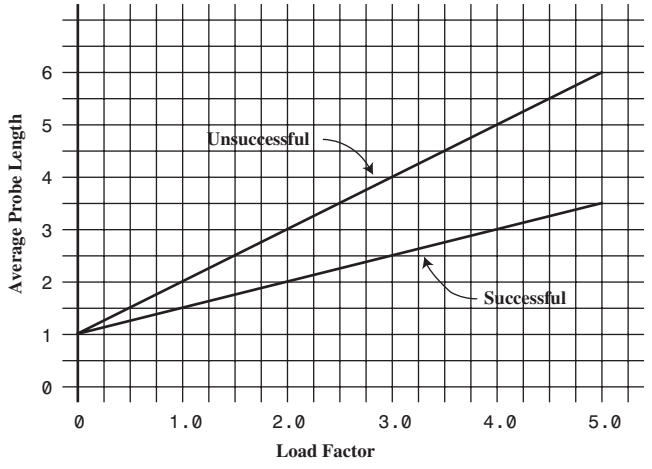


***FIGURE 11.14*** Separate-chaining performance.

For an ordered list, only half the items must be examined in an unsuccessful search, so the time is the same as for a successful search.

In separate chaining it's typical to use a load factor of about 1.0 (the number of data items equals the array size). Smaller load factors don't improve performance significantly, but the time for all operations increases linearly with load factor, so going beyond 2 or so is generally a bad idea.

#### Insertion

If the lists are not ordered, insertion is always immediate, in the sense that no comparisons are necessary. The hash function must still be computed, so let's call the insertion time 1.

If the lists are ordered, then, as with an unsuccessful search, an average of half the items in each list must be examined, so the insertion time is

```
1 + loadFactor / 2
```

## Open Addressing Versus Separate Chaining

If open addressing is to be used, double hashing seems to be the preferred system by a small margin over quadratic probing. The exception is the situation in which plenty of memory is available and the data won't expand after the table is created; in this case linear probing is somewhat simpler to implement and, if load factors below 0.5 are used, causes little performance penalty.

If the number of items that will be inserted in a hash table isn't known when the table is created, separate chaining is preferable to open addressing. Increasing the load factor causes major performance penalties in open addressing, but performance degrades only linearly in separate chaining.

When in doubt, use separate chaining. Its drawback is the need for a linked list class, but the payoff is that adding more data than you anticipated won't cause performance to slow to a crawl.

## Hashing and External Storage

At the end of Chapter 10, "2-3-4 Trees and External Storage," we discussed using B-trees as data structures for external (disk-based) storage. Let's look briefly at the use of hash tables for external storage.

Recall from Chapter 10 that a disk file is divided into blocks containing many records, and that the time to access a block is much larger than any internal processing on data in main memory. For these reasons the overriding consideration in devising an external storage strategy is minimizing the number of block accesses.

On the other hand, external storage is not expensive per byte, so it may be acceptable to use large amounts of it, more than is strictly required to hold the data, if by so doing we can speed up access time. This is possible using hash tables.

### Table of File Pointers

The central feature in external hashing is a hash table containing block numbers, which refer to blocks in external storage. The hash table is sometimes called an *index* (in the sense of a book's index). It can be stored in main memory or, if it is too large, stored externally on disk, with only part of it being read into main memory at a time. Even if it fits entirely in main memory, a copy will probably be maintained on the disk and read into memory when the file is opened.

### Non-Full Blocks

Let's reuse the example from Chapter 10 in which the block size is 8,192 bytes, and a record is 512 bytes. Thus, a block can hold 16 records. Every entry in the hash table points to one of these blocks. Let's say there are 100 blocks in a particular file.

The index (hash table) in main memory holds pointers to the file blocks, which start at 0 at the beginning of the file and run up to 99.

In external hashing it's important that blocks don't become full. Thus, we might store an average of 8 records per block. Some blocks would have more records, and some fewer. There would be about 800 records in the file. This arrangement is shown in Figure 11.15.

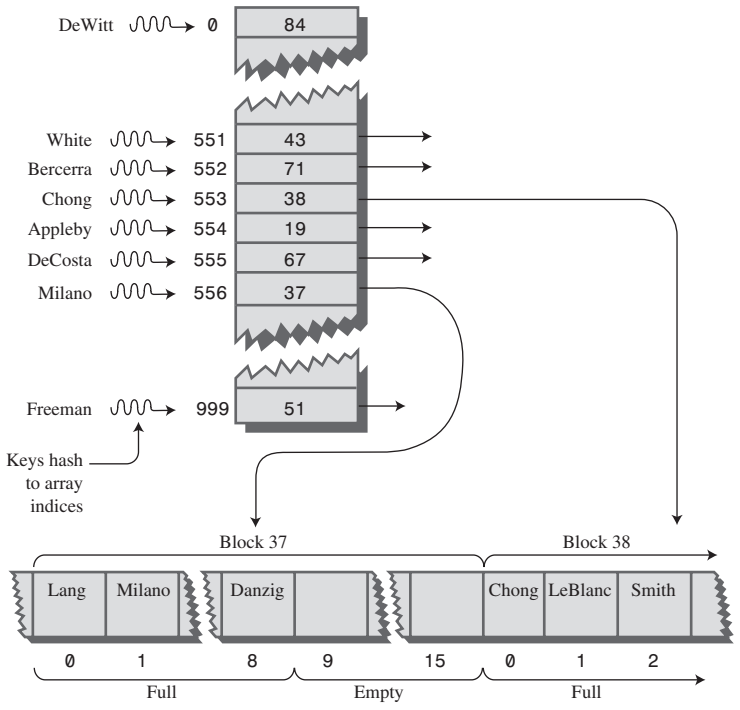


*FIGURE 11.15* External hashing.

All records with keys that hash to the same value are located in the same block. To find a record with a particular key, the search algorithm hashes the key, uses the hash value as an index to the hash table, gets the block number at that index, and reads the block.

This process is efficient because only one block access is necessary to locate a given item. The downside is that considerable disk space is wasted because the blocks are, by design, not full.

To implement this scheme, we must choose the hash function and the size of the hash table with some care so that a limited number of keys hash to the same value. In our example, we want only eight records per key, on the average.

## Full Blocks

Even with a good hash function, a block will occasionally become full. This situation can be handled using variations of the collision-resolution schemes discussed for internal hash tables: open addressing and separate chaining.

In open addressing, if, during insertion, one block is found to be full, the algorithm inserts the new record in a neighboring block. In linear probing this is the next block, but it could also be selected using a quadratic probe or double hashing. In separate chaining, special overflow blocks are made available; when a primary block is found to be full, the new record is inserted in the overflow block.

Full blocks are undesirable because an additional disk access is necessary for the second block; this doubles the access time. However, this is acceptable if it happens rarely.

We've discussed only the simplest hash table implementation for external storage. There are many more complex approaches that are beyond the scope of this book.

## Summary

- A hash table is based on an array.
- The range of key values is usually greater than the size of the array.
- A key value is hashed to an array index by a hash function.
- An English-language dictionary is a typical example of a database that can be efficiently handled with a hash table.
- The hashing of a key to an already-filled array cell is called a collision.
- Collisions can be handled in two major ways: open addressing and separate chaining.
- In open addressing, data items that hash to a full array cell are placed in another cell in the array.
- In separate chaining, each array element consists of a linked list. All data items hashing to a given array index are inserted in that list.
- We discussed three kinds of open addressing: linear probing, quadratic probing, and double hashing.
- In linear probing the step size is always 1, so if x is the array index calculated by the hash function, the probe goes to x, x+1, x+2, x+3, and so on.
- The number of such steps required to find a specified item is called the probe length.
- In linear probing, contiguous sequences of filled cells appear. They are called primary clusters, and they reduce performance.
- In quadratic probing the offset from x is the square of the step number, so the probe goes to x, x+1, x+4, x+9, x+16, and so on.

- Quadratic probing eliminates primary clustering but suffers from the less severe secondary clustering.
- Secondary clustering occurs because all the keys that hash to the same value follow the same sequence of steps during a probe.
- All keys that hash to the same value follow the same probe sequence because the step size does not depend on the key, but only on the hash value.
- In double hashing the step size depends on the key and is obtained from a secondary hash function.
- If the secondary hash function returns a value s in double hashing, the probe goes to x, x+s, x+2s, x+3s, x+4s, and so on, where s depends on the key but remains constant during the probe.
- The load factor is the ratio of data items in a hash table to the array size.
- The maximum load factor in open addressing should be around 0.5. For double hashing at this load factor, searches will have an average probe length of 2.
- Search times go to infinity as load factors approach 1.0 in open addressing.
- It's crucial that an open-addressing hash table does not become too full.
- A load factor of 1.0 is appropriate for separate chaining.
- At this load factor a successful search has an average probe length of 1.5, and an unsuccessful search, 2.0.
- Probe lengths in separate chaining increase linearly with load factor.
- A string can be hashed by multiplying each character by a different power of a constant, adding the products, and using the modulo operator (%) to reduce the result to the size of the hash table.
- To avoid overflow, we can apply the modulo operator at each step in the process, if the polynomial is expressed using Horner's method.
- Hash table sizes should generally be prime numbers. This is especially important in quadratic probing and separate chaining.
- Hash tables can be used for external storage. One way to do this is to have the elements in the hash table contain disk-file block numbers.

## Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Using big O notation, say how long it takes (ideally) to find an item in a hash table.
2. A _____ transforms a range of key values into a range of index values.
3. Open addressing refers to
   a. keeping many of the cells in the array unoccupied.
   b. keeping an open mind about which address to use.
   c. probing at cell x+1, x+2, and so on until an empty cell is found.
   d. looking for another location in the array when the one you want is occupied.
4. Using the next available position after an unsuccessful probe is called _____.
5. What are the first five step sizes in quadratic probing?
6. Secondary clustering occurs because
   a. many keys hash to the same location.
   b. the sequence of step lengths is always the same.
   c. too many items with the same key are inserted.
   d. the hash function is not perfect.
7. Separate chaining involves the use of a _____ at each location.
8. A reasonable load factor in separate chaining is _____.
9. True or False: A possible hash function for strings involves multiplying each character by an ever-increasing power.
10. The best technique when the amount of data is not well known is
    a. linear probing.
    b. quadratic probing.
    c. double hashing.
    d. separate chaining.
11. If digit folding is used in a hash function, the number of digits in each group should reflect _____.
12. True or False: In linear probing an unsuccessful search takes longer than a successful search.

13. In separate chaining the time to insert a new item
    a. increases linearly with the load factor.
    b. is proportional to the number of items in the table.
    c. is proportional to the number of lists.
    d. is proportional to the percentage of full cells in the array.

14. True or False: In external hashing, it's important that the records don't become full.

15. In external hashing, all records with keys that hash to the same value are located in _____.

## Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

1. In linear probing, the time for an unsuccessful search is related to the cluster size. Using the Hash workshop applet, find the average cluster size for 30 items filled into 60 cells, with a load factor of 0.5. Consider an isolated cell (that is, with empty cells on both sides) to be a cluster of size 1. To find the average, you could count the number of cells in each cluster and divide by the number of clusters, but there's an easier way. What is it? Repeat this experiment for a half-dozen 30-item fills and average the cluster sizes. Repeat the entire process for load factors of 0.6, 0.7, 0.8, and 0.9. Do your results agree with the chart in Figure 11.12?

2. With the HashDouble Workshop applet, make a small quadratic hash table, with a size that is *not* a prime number, say 24. Fill it very full, say 16 items. Now search for non-existent key values. Try different keys until you find one that causes the quadratic probe to go into an unending sequence. This happens because the quadratic step size, modulo a non-prime array size, forms a repeating series. The moral: Make your array size a prime number.

3. With the HashChain applet, create an array with 25 cells, and then fill it with 50 items, with a load factor of 2.0. Inspect the linked lists that are displayed. Add the lengths of all these linked lists and divide by the number of lists to find the average list length. On the average, you'll need to search this length in an unsuccessful search. (Actually, there's a quicker way to find this average length. What is it?)

## Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's Web site.)

**11.1** Modify the `hash.java` program (Listing 11.1) to use quadratic probing.

**11.2** Implement a linear probe hash table that stores strings. You'll need a hash function that converts a string to an index number; see the section "Hashing Strings" in this chapter. Assume the strings will be lowercase words, so 26 characters will suffice.

**11.3** Write a hash function to implement a digit-folding approach in the hash function (as described in the "Hash Functions" section of this chapter). Your program should work for any array size and any key length. Use linear probing. Accessing a group of digits in a number may be easier than you think. Does it matter if the array size is not a multiple of 10?

**11.4** Write a `rehash()` method for the `hash.java` program. It should be called by `insert()` to move the entire hash table to an array about twice as large whenever the load factor exceeds 0.5. The new array size should be a prime number. Refer to the section "Expanding the Array" in this chapter. Don't forget you'll need to handle items that have been "deleted," that is, written over with –1.

**11.5** Instead of using a linked list to resolve collisions, as in separate chaining, use a binary search tree. That is, create a hash table that is an array of trees. You can use the `hashChain.java` program (Listing 11.3) as a starting point and the `Tree` class from the `tree.java` program (Listing 8.1) in Chapter 8. To display a small tree-based hash table, you could use an inorder traversal of each tree.

The advantage of a tree over a linked list is that it can be searched in O(logN) instead of O(N) time. This time savings can be a significant advantage if very high load factors are encountered. Checking 15 items takes a maximum of 15 comparisons in a list but only 4 in a tree.

Duplicates can present problems in both trees and hash tables, so add some code that prevents a duplicate key from being inserted in the hash table. (Beware: The `find()` method in `Tree` assumes a non-empty tree.) To shorten the listing for this program, you can forget about deletion, which for trees requires a lot of code.