Hashing Methods

We are now ready to study several hashing methods. After we ... several different methods, we create a simple hashing algorithm that incorporates several of them. The hashing techniques that we study are shown in Figure 2-8.

Direct Method

In **direct hashing,** the key is the address without any algorithmic manipulation. The data structure must therefore contain an element

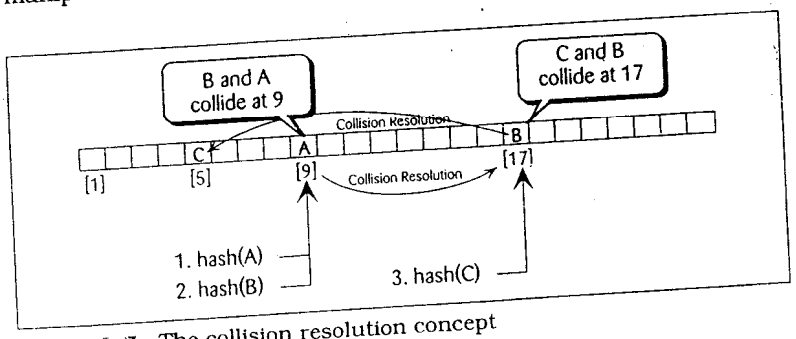

Figure 2-7 The collision resolution concept

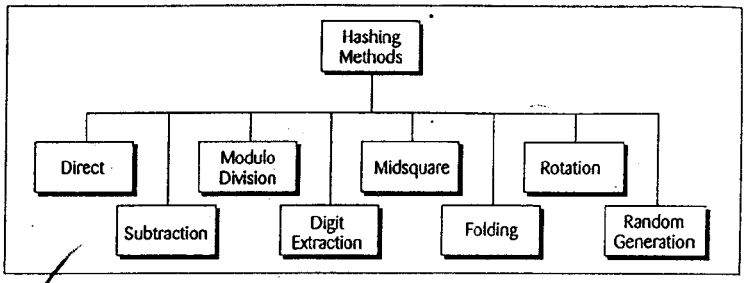

Figure 2-8 Basic hashing techniques

for every possible key. While the situations where you can use direct hashing are limited, when it can be used it is very powerful because it guarantees that there are no synonyms. Let's look at two applications.

First, consider the problem in which we need to total monthly sales by the days of the month. For each sale, we have the date and the amount of the sale. In this case, we create an array of 31 accumulators. As we read the sales records for the month, we use the day of the month as the key for the array and add the sale amount to the corresponding accumulator. The accumulation code is shown below.

```
dailySales[sale.day] = dailySales[sale.day] + sale.amount
```

Now let's consider a more complex example. Imagine that a small organization has fewer than 100 employees. Each employee is assigned an employee number between 1 and 100. In this case, if we create an array of 100 employee records, the employee number can be directly used as the address of any individual record. This concept is shown in Figure 2-9.

As you study Figure 2-9, note that not every element in the array contains an employee's record. While in our daily sales example, every element was used, more often than not there are some empty elements in hashed lists. In fact, as we will see later, all hashing techniques other than direct hashing require that some of the elements be empty to reduce the number of collisions.

As you may have noticed, although this is the ideal method, its application is very limited. For example, we cannot have the social security number as the key using this method because social security numbers are 9 digits. In other words, if we use the social security number as the key, we need an array as big as 999,999,999 records but we would use less than 100 one of them. Let's turn our attention, therefore, to hashing techniques that map a large population of possible keys into a small address space.

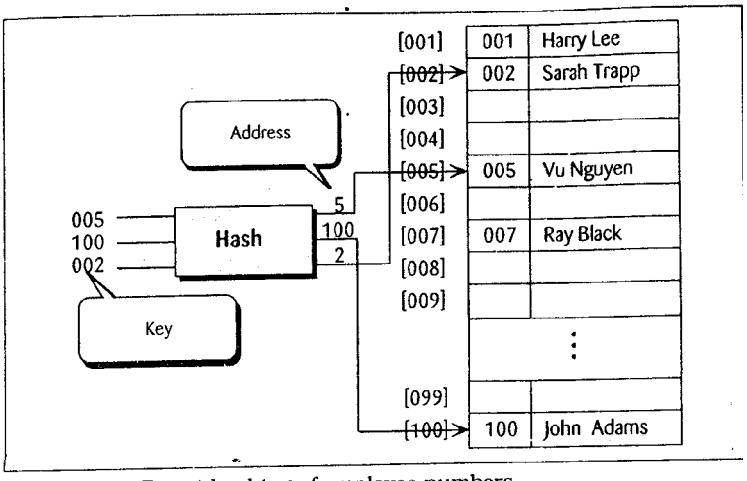

Figure 2-9 Direct hashing of employee numbers

Subtraction Method

Sometimes we have keys that are consecutive but do not start from one. For example, a company may have only 100 employees, but the employee numbers start from 1000 and go to 1100. In this case, we use a very simple hashing function that subtracts 1000 from the key to determine the address. The beauty of this example is that it is simple and that it also guarantees that there will be no collisions. Its limitations are the same as direct hashing: it can only be used for small lists in which the keys map to a densely filled list.

Modulo-Division Method

Also known as **division-remainder**, the **modulo-division** method divides the key by the array size and uses the remainder plus one for the address. This gives us the simple hashing algorithm shown below where list size is the number of elements in the array.

```
address = key MODULUS listSize + 1
```

While this algorithm works with any list size, a list size that is a prime number produces fewer collisions than other list sizes. We should therefore try, whenever possible, to make the array size a prime number.

As our little company begins to grow, we realize that soon we will have more than 100 employees. Planning for the future, we create a new employee numbering system that will handle 1,000,000 employees. We also decide that we want to provide data space for up to 300 employ-

ees. The first prime number greater than 300 is 307. We therefore choose 307 as our list (array) size. Our new employee list and some of its hashed addresses is shown in Figure 2-10.

To demonstrate, let's hash Bryan Devaux's employee number, 121267.

```
121267 / 307 = 395 with remainder of 2
Therefore: hash(121267) = 3
```

Digit-Extraction Method

Using **digit extraction,** selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three-digit address (000–999), we could select the first, third, and fourth digits (from the left) and use them as the address. Using the keys from Figure 2-10, we would hash them to the addresses shown below.

```
379452 ⇒ 394
121267 ⇒ 112
378845 ⇒ 388
160252 ⇒ 102
045128 ⇒ 051
```

Midsquare Method

In **midsquare hashing,** the key is squared and the address selected from the middle of the squared number. The most obvious limitation

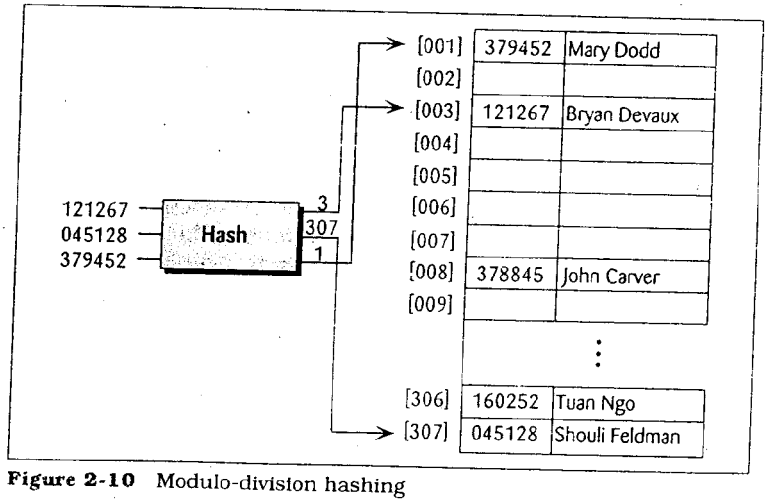

Figure 2-10 Modulo-division hashing

of this method is the size of the key. Given a key of six digits, the product will be 12 digits, which is beyond the maximum integer size of many computers. Because most personal computers can handle a nine-digit integer, lets demonstrate the concept with keys of four digits. Given a key of 9452, the midsquare address calculation is shown below using a four-digit address (0000–9999).

```
9452 * 9452 = 89340304: address is 3403
```

As a variation on the midsquare method, we can select a portion of the key, such as the middle three digits, and then use them rather than the whole key. This allows the method to be used when the key is too large to square. For example, for the keys in Figure 2-10, we can select the first three digits and then use the midsquare method as shown below. (We select the third, fourth, and fifth digits as the address.)

```
379452: 379 * 379 = 143641 ⇒ 364
121267: 121 * 121 = 014641 ⇒ 464
378845: 378 * 378 = 142884 ⇒ 288
160252: 160 * 160 = 025600 ⇒ 560
045128: 045 * 045 = 002025 ⇒ 202
```

Note that in the midsquare method, the same digits must be selected from the product. For that reason, we consider the product to have sufficient leading zeros to make it the full six digits.

Folding Methods

There are two **folding methods** that are used, fold shift and fold boundary.

In **fold shift,** the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part. For example, imagine we want to map social security numbers into three-digit addresses. We divide the nine-digit social security number into three, three-digit numbers, which are then added. If the resulting sum is greater than 999, then we discard the leading digit. This method is seen in Figure 2-11(a).

In **fold boundary,** the left and right numbers are folded on a fixed boundary between them and the center number. This results in the two outside values being reversed as seen in Figure 2-11(b). It is interesting to note that the two folding methods give different hashed addresses.

Rotation Method

Rotation hashing is generally not used by itself but rather is incorporated in combination with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee numbers and part numbers. Often a simple hashing algorithm tends

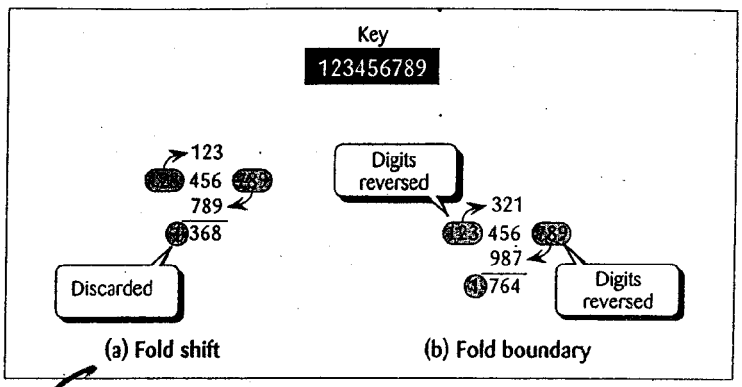

Figure 2-11 Hash fold examples

to create synonyms when hashing keys are identical except for the last character. Rotating the last character to the front of the key minimizes this effect. For example, consider the example of a six-digit employee number that might be used in a large company shown in Figure 2-12.

Examine the rotated key carefully. Because all keys now end in 60010, they would obviously not work well with modulo division. On the other hand, if we used a simple fold shift hash on the original key and a two-digit address, the addresses would be sequential starting with 62. Using a shift hash on the rotated key results in the following series of addresses: 26, 36, 46, 56, 66, which has the desired effect of spreading the data more evenly across the address space. Rotation is often used in combination with folding and pseudorandom hashing.

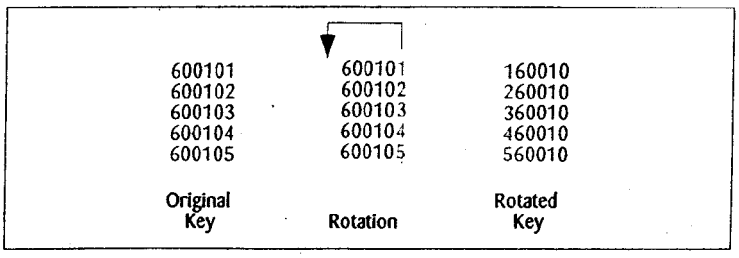

Figure 2-12 Rotation hashing

Pseudorandom Method

In the **pseudorandom method,** the key is used as the seed in a pseudo random number generator and the resulting random number ther scaled into the possible address range using modulo division. A com mon random number generator is shown below.

```
y = ax + c
```

To use the pseudorandom number generator as a hashing metho we set x to the key, multiply it by the coefficient, a, and then add constant, c. The result is then divided by the list size with the remainde plus 1 (see Modulo-Division Method on page 46) being the hashed ac dress. For maximum efficiency, the factors a and c should be prim numbers. Let's demonstrate the concept with an example from Figu 2-10 on page 47. To keep the calculation reasonable, we use 17 and for the factors a and c, respectively. Also, the list size in the examp is the prime number 307.

```
y = ((17 * 121267) + 7) modulo 307 + 1
y = (2061539 + 7) modulo 307 + 1
y = 2061546 modulo 307 + 1
y = 41 + 1
y = 42
```

We will see this pseudorandom number generator again when discuss collision resolution.

Hashing Algorithm

Before we conclude our discussion of hashing methods, we need describe a complete hashing algorithm. While the hashing metho may work well when we hash a key to an address in an array, hashi to large files is generally more complex. It often requires extensive ana sis of the population of keys to be hashed to determine the numb of synonyms and the length of the collision strings produced by t algorithm. While the study of such analysis is beyond the scope of th text, we present the algorithm described below as a simple example a hashing algorithm for a large file.

Assume that we have an alphanumeric key consisting of up to bytes that we need to hash this key into a 32 bit address. The fi step is to convert the alphanumeric key into a number. This is do by adding the American Standard Code for Information Interchar (ASCII) value for each character to an accumulator that will be t address. As each character is added, we rotate the bits in the addr to maximize the distribution of the values. After the characters in t key have been completely hashed, we take the absolute value of t address and then map it into the address range for the file. This lo is seen in Algorithm 2-6.