of the hashing algorithm. That is, each hashing method can be used with each of the collision resolution methods. In this section we discuss the collision resolution methods shown in Figure 2-13.

Before we discuss the collision resolution methods, however, we need to cover two more concepts. Because of the nature of hashing algorithms, it is necessary to have some empty elements in a list at all times. In fact, we define a full list as a list in which all elements except one contain data. As a rule of thumb, a hashed list should not be allowed to become more than 75% full. This leads us to our first concept, load factor. The **load factor** of a hashed list is the number of elements in the list divided by the number of physical elements allocated for the list expressed as a percentage. Traditionally, load factor is assigned the symbol alpha (α). The formula in which k represents the number of filled elements in the list and n represents the total number of elements allocated to the list is:

$$\alpha = \frac{k}{n} \times 100$$

Because there can never be more than n elements in a list, α will always be a percentage.

As data are added to a list and collisions are resolved, some hashing algorithms tend to cause data to group within the list. This tendency of data to build up unevenly across a hashed list is known as **clustering.** Clustering is a concern because it is usually created by collisions. If the list contains a high degree of clustering, then the number of probes to locate an element grows and the processing efficiency of the list is reduced.

Two distinct types of clusters have been identified by computer scientists. The first, **primary clustering** occurs when data become clustered around a home address. Primary clustering is easy to identify. Consider for example, the population clusters found in any state. If you

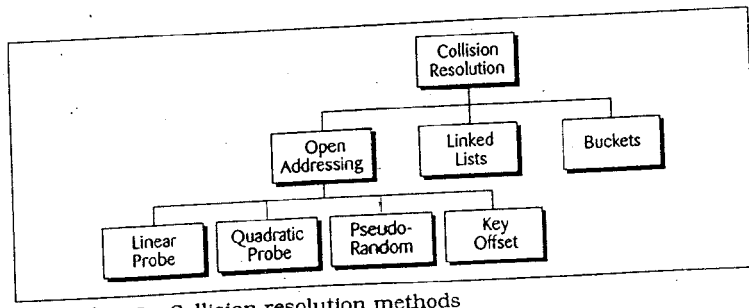

Figure 2-13 Collision resolution methods

Linear Probe

Our first collision resolution method is also the simplest. In a **linear probe,** when data cannot be stored in the home address, we resolve the collision by adding one to the current address. For example, let's add two more elements to the modulo-division method example in Figure 2-10 on page 47. The results are shown in Figure 2-14. When we insert key 070918, we find an empty element and insert it with no collision. When we try to insert key 166702, however, we have a collision at location 002. We try to resolve the collision by adding one to the address and inserting the new data at location 003. However, this address is also filled. We therefore add another one to the address and this time find an empty location, 004, where we can place the new data.

As an alternative to a simple linear probe, we can alternately add one, subtract two, add three, subtract four, and so forth until an empty element is located. In either method, the code for the linear probe must ensure that the next collision resolution address lies within the boundaries of the list. Thus, if a key hashes to the last location in the list, adding one must produce the address of the first element of the list. On the contrary, if the key hashes to the first element of the list, subtracting one must produce the address of the last element in the list.

Linear probes have two advantages. First they are quite simple to implement. Second, data tend to remain near their home address. This can be important in implementations where being near the home address is important, such as when we hash to a disk address. On the other hand, it tends to produce primary clustering. Additionally, they tend to make the search algorithm more complex especially after data have been deleted.

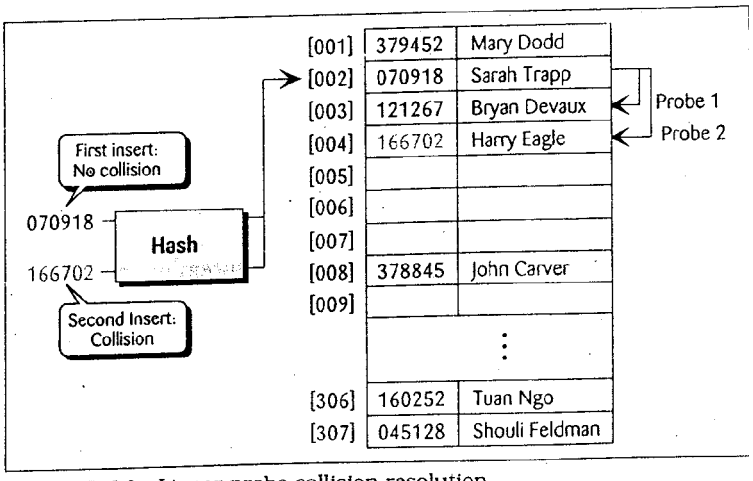

Figure 2-14 Linear probe collision resolution

Quadratic Probe

Primary clustering, although not necessarily secondary clustering, can be eliminated by adding a value other than one to the current address. One easily implemented method is to use the **quadratic probe.** In the quadratic probe, the increment is the collision probe number squared. Thus, for the first probe, we add 1^2; for the second collision probe, we add 2^2; for the third collision probe we add 3^2; and so forth until we either find an empty element or we exhaust the possible elements. To ensure that we don't run off the end of the address list, we use the modulo of the quadratic sum for the new address. This sequence is seen in Table 2-2, which for simplicity assumes a collision at location 1 and a list size of 100.

A potential disadvantage of the quadratic probe is the time required to square the probe number. We can eliminate the multiply however by using an increment factor that increases by two each probe. Adding the increment factor to the previous increment gives us the next increment, which as you can see by the last column in Table 2-2 is the equivalent of the probe squared.

The quadratic probe has one limitation: It is not possible to generate a new address for every element in the list. For example, in our example in Table 2-2, only 42 of the probes will generate unique addresses. The other 58 locations in the list will not be probed. To see two examples of duplicate addresses in Table 2-2, extend it to 16 probes. The solution to the problem is to use a list size that is a prime number. When the list size is a prime number, at least half of the list is reachable, which is a reasonable number.

Pseudorandom Collision Resolution

The last two open addressing methods are collectively known as **double hashing.** In each method, rather than using an arithmetic probe function, the address is rehashed. As will be apparent during their discussion, both methods prevent primary clustering.

The first method uses a **pseudorandom** number to resolve the

Probe Number	Collision Location	Probe² and Increment	New Address	Increment Factor	Next Increment
1	1	$1^2 = 1$	02	1	1
2	2	$2^2 = 4$	06	3	4
3	6	$3^2 = 9$	15	5	9
4	15	$4^2 = 16$	31	7	16
5	31	$5^2 = 25$	56	9	25
6	56	$6^2 = 36$	92	11	36
7	92	$7^2 = 49$	41	13	49

Table 2-2 Quadratic collision resolution increments

new address = (old address + attempt^2) % list size

collision. We saw the pseudorandom number generator as a hashing method in Pseudorandom Method on page 50. We now use it as a collision resolution method. In this case, rather than using the key as a factor in the random number calculation, we use the address. Consider the collision we created in Figure 2-14 on page 54. We now resolve the collision using the following pseudorandom number generator where a is 3 and c is -1.

```
y = (ax + c) modulo listSize + 1
= (3 * 2 + (-1)) Modulo 307 + 1
= 6
```

In this example, the collision is resolved by placing the new data in element 006 (Figure 2-15). We have to keep the coefficients small to fit our example. A better set of factors would use a large prime number for a, such as 1,663.

While pseudorandom numbers are a relatively simple solution, they do have one significant limitation. Once a collision occurs, there is only one collision resolution path through the list that is followed by all keys. (This deficiency also occurs in the linear and quadratic probes.) Because this can create significant secondary clustering, we should look for a method that produces different collision paths for different keys.

Key Offset

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the pseudorandom number generator

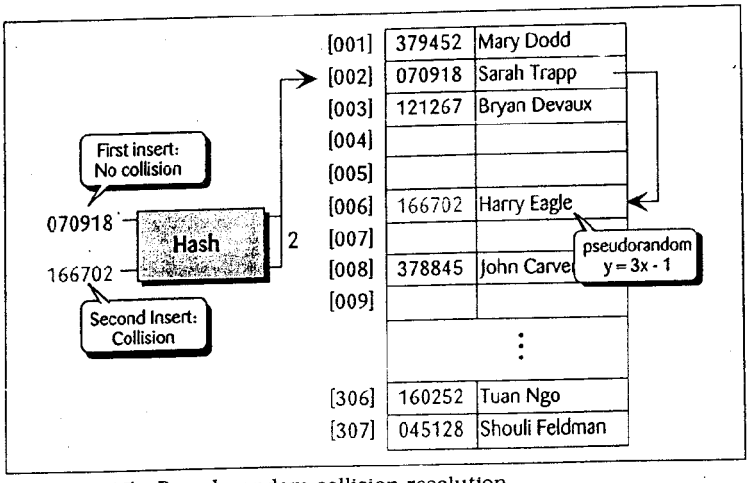

Figure 2-15 Pseudorandom collision resolution

produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address *and* the key. One of the simplest versions simply adds the quotient of the key divided by the list size to the address to determine the next collision-resolution address as shown in the formula below.

```
offSet   = [key / listSize]
address  = ((offSet + old address) modulo listSize) + 1
```

For example, when the key is 166702 and the list size is 307, using the modulo-division hashing method we generate an address of 2. As shown in Figure 2-15, this synonym of 070918 produces a collision at address 2. Using key offset to calculate the next address, we get 239 as shown below.

```
offSet   = [166702 / 307] = 543
address  = ((543 + 002) modulo 307) + 1 = 239
```

If 239 were also a collision, we would repeat the process to locate the next address as shown below.

```
offSet   = [166702 / 307] = 543
address  = ((543 + 239) modulo 307) + 1 = 169
```

To really see the effect of key offset, we need to calculate several different keys, all hashing to the same home address. In Table 2-3 we calculate the next two collision probe addresses for three keys that collide at address 002.

Note that each key resolves its collision at a different address for both the first and second probes.

Linked List Resolution

A major disadvantage to open addressing is that each collision resolution increases the probability of future collisions. This disadvantage is eliminated in the second approach to collision resolution, linked lists. A **linked list** is an ordered collection of data in which each element

Key	Home Address	Key Offset	Probe 1	Probe 2
166702	2	543	239	169
572556	2	1,865	026	050
067234	2	219	222	135

Table 2-3 Key-offset examples

contains the location of the next element. For example, in Figure 2-16, array element 002, Sarah Trapp, contains a pointer to the next element, Harry Eagle, which in turn contains a pointer to the third element, Chris Walljasper. We will study the maintenance of linked lists in the next chapter.

Linked list resolution uses a separate area to store collisions and chains all synonyms together in a linked list. It uses two storage areas, the prime area and the **overflow area.** Each element in the prime area contains an additional field, a link head-pointer to a linked list of overflow data in the overflow area. When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area. While the overflow area can be any data structure, it is typically implemented as a linked list in dynamic memory. Figure 2-16 shows the linked list from Figure 2-15 with the three synonyms for address 002.

While the linked list data can be stored in any order, a LIFO sequence or a key sequence is the most common. The LIFO sequence is the fastest for inserts because the linked list does not have to be scanned to insert the data. The element being inserted into overflow is simply placed at the beginning of the linked list and linked to the node in the prime area. Key sequenced lists, with the key in the prime area being the smallest, provide for faster search retrieval. Which one is used depends on the application.

Bucket Hashing

Another approach to handling the problem of collision is to hash to **buckets,** nodes that accommodate multiple data occurrences. Because

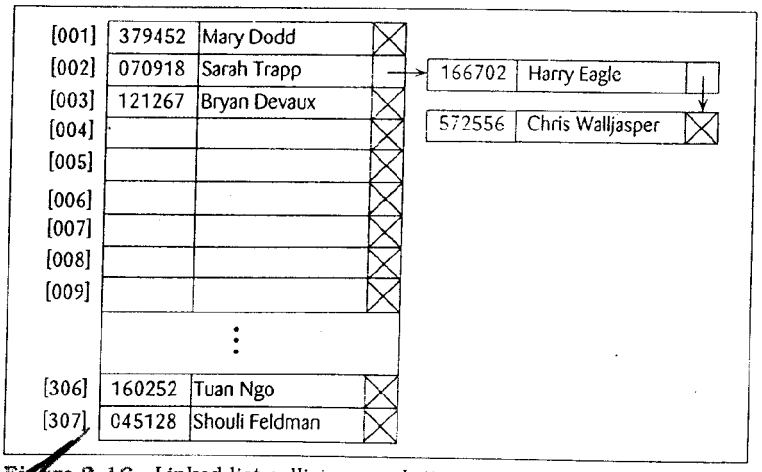

Figure 2-16 Linked list collision resolution

a bucket can hold multiple pieces of data, collision are postponed until the bucket is full. Assume for example, that in our Figure 2-16 list, each address is big enough to hold data about three employees. Under this assumption, there would not be a collision until we tried to add a fourth employee to an address. There are two problems with this concept. First, it uses significantly more space because many of the buckets will be empty or partially empty at any given time. Second, it does not completely resolve the collision problem. At some point, a collision will occur and need to be resolved. When it does, a typical approach is to use a linear probe, assuming that the next element will have some empty space. Figure 2-17 demonstrates the bucket approach.

Study the second bucket in Figure 2-17. Note that it contains the data for three entries, all of which hashed to address 2. We will not get a collision until the fourth key, 572556 in our example, is inserted into the list. When a collision finally occurs, that is when the bucket if full, any of the collision resolution methods may be used. For example, in Figure 2-17, when we inserted 572556 a collision occurred because bucket 2 was full. We then used a linear probe to insert it into location 3. Also note that for efficiency, we have placed the keys within a bucket in ascending key sequence.

Combination Approaches

There are several approaches to resolving collisions. As we saw with the hashing methods, a complex implementation will often use multiple. For example, one large database implementation hashes to a bucket. If the bucket is full, it uses a set number of linear probes,

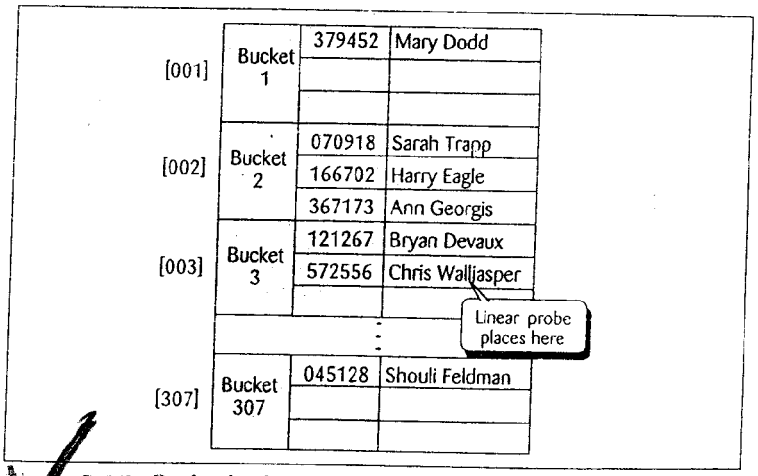

Figure 2-17 Bucket hashing