Table 6.1 Multi-key access and its inverted table

(a) Records of a Telephone Exchange

Index	*Name*	*Address*	*Phone*
1	K.R. Narayana	Maker Towers #6	257696
2	A.B. Vajpayee	9 Vivekananda Road	257459
3	L.K. Advani	11 Von Kasturba Marg	257583
4	Mamta Banerjee	342 Patel Avenue	257423
5	Y. Sinha	5 SBI Road	257504
6	D. Kulkarni	369 Faculty Colony	257564
7	T. Krishnamurthy	185 Faculty Colony	257579
8	N. Puranjay	409 Medical Colony	257409
9	Tadi Tabi	Officers Mess #52	257871

(b) Inverted Table

Name	*Address*	*Phone*
2	7	8
6	6	4
1	1	2
3	8	5
4	9	6
8	4	7
7	5	3
9	2	1
5	3	9

6.4 HASH TABLES

There are other types of tables which help us to retrieve information very efficiently. The ideal *hash table* is merely an array of some constant size; the size depends on the application where it will be used. The hash table contains key values with pointers to the corresponding records. The basic idea of a hash table is that we have to place a key value into a location in the hash table; the location will be calculated from the key value itself. This one-to-one correspondence between a key value and an index in the hash table is known as *address calculation indexing* or more commonly *hashing*. In the present section, we will discuss hashing techniques and their related issues.

6.4.1 Hashing Techniques

The main idea behind any hashing technique is to find a one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in Figure 6.6, where K denotes a set of key values, I denotes a range of indices and H denotes the mapping function from K to I.

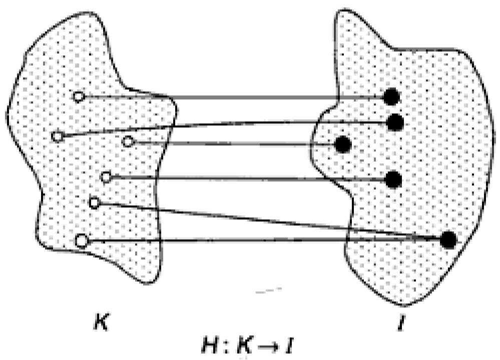


Figure 6.6 Concept of hashing.

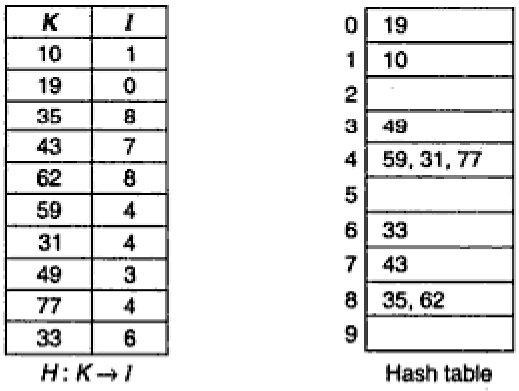
It may be noted that the mapping is subjective, that is all key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the *hash function*. A particular hashing technique uses a particular hash function. The hash function plays a dominant role in hashing techniques. There are two principal criteria in deciding a hash function $H:K \rightarrow I$ as follows:

1. The function H should be very easy and quick to compute.
2. The function H should as far as possible give two different indices for two different key values.

As an example, let us consider a hash table of size 10 whose indices are 0, 1, 2, ..., 8, 9. Suppose a set of key values are: 10, 19, 35, 43, 62, 59, 31, 49, 77, 33. Let us assume the hash function H is as stated below:

- Add the two digits in the key.
- Take the digit at the unit place of the result as the index; ignore the digit at the tenth place, if any.

Using this hash function, the mappings from key values to indices and to hash table are shown in Figure 6.7. In this example, for the given set of key values, the hash function does



K	*I*
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

$H : K \rightarrow I$

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

Figure 6.7 Example of hashing.

not distribute them uniformly over the hash table; some entries are there which are empty, and in some entries more than one key value needs to be stored. Allotment of more than one key value in one location in the hash table is called *collision*. We have found three collisions for 62, 31 and 77 in the above-mentioned example.

It can be noted that $|K| = |I|$, that is, the number of key values is the same as the size of the hash table, but this is not the case always. In general, $|K| > |I|$.

The following are some hash functions which are very common and popularly applied in various applications.

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number N of keys in K. The hash function H is then defined by

$$H(k) = k(\text{MOD } h) \qquad \text{if indices start from 0}$$

or

$$H(k) = k(\text{MOD } h) + 1 \qquad \text{if indices start from 1}$$

where $k \in K$, a key value. The operator MOD defines the modulo arithmetic operation, which is equal to the remainder of dividing k by h. For example, if $k = 31$ and $h = 13$ then

$$H(31) = 31(\text{MOD } 13) = 5$$

or

$$H(31) = 31(\text{MOD } 13) + 1 = 6$$

The number h is usually chosen to be a prime number or a number without small divisors, since this usually minimizes the number of collisions. Generally, h is a prime number and equal to the size of the hash table.

Midsquare method

Another hash function which has been widely used in many applications is the midsquare method. The method is defined as follows:

The hash function H is defined by $H(k) = x$, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. This selection usually depends on the size of the hash table. It needs to be emphasized that the same criteria should be used for selecting the bits or digits for all of the keys.

As an example, suppose the key values are of the integer type, and we require 3-digit addresses. Our selection criteria are to select 3 digits at even positions starting from the right-most digit in the square. Let us see the address calculations, for 3 distinct keys and with the hash function, as defined above:

k	:	1234	2345	3456
k^2	:	1522756	5499025	11943936
$H(k)$	:	525	492	933

Here, we observe that the second, the fourth, and the sixth digits, counting from the right, are chosen for the hash addresses.

The midsquare method has been criticized because of time-consuming computation (multiplication operation), but it usually gives good results so far as the uniform distribution of the keys over the hash table is concerned.

Folding method

Another fair method for a hash function is the folding method. The method can be defined as follows:

Partition the key k into a number of parts $k_1, k_2, ..., k_n$, where each part, except possibly the last, has the same number of bits or digits as the required address width. Then the parts are added together, ignoring the last carry, if any. Alternatively,

$$H(k) = k_1 + k_2 + \cdots + k_n$$

where the last carry, if any, is ignored. If the keys are in binary form, the exclusive-OR operation may be substituted for addition. There are many variations known in this method. One is called the *fold shifting method,* where the even number parts, $k2$, $k4$, ... are each reversed before the addition. Another variation is called the *fold boundary method.* Here, two boundary parts, namely, k_1 and k_n, each are reversed and then added to all other parts. As an example, let us take the size of each part to be 2; the following calculations are performed on the given key values (integers) as shown below:

k:	1522756	5499025	11943936
Chopping:	01 52 27 56	05 49 90 25	11 94 39 36
Pure folding:	01 + 52 + 27 + 56 = 136	05 + 49 + 90 + 25 = 169	11 + 94 + 39 + 36 = 180
Fold shifting:	10 + 52 + 72 + 56 = 190	50 + 49 + 09 + 25 = 133	11 + 94 + 93 + 36 = 234
Fold boundary:	10 + 52 + 27 + 65 = 154	50 + 49 + 90 + 52 = 241	11 + 94 + 39 + 63 = 207

Folding is a hashing function which is also useful in converting multi-word keys into a single word so that another hashing function can be used on that. In fact, the term 'hashing' comes from this technique of 'chopping' a key into pieces.

Digit analysis method

The basic idea of this hashing function is to form hash addresses by extracting and/or shifting the extracted digits or bits of the original key. As an example, given a key value, say 6732541, it can be transformed to the hash address 427 by extracting the digits in even positions and then reversing this combination. For a given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and then rearrangement is based on some analysis. To do this, an analysis is performed to determine which key positions should be used in forming hash addresses. For each criterion, hash addresses are calculated and then a graph is plotted, then that criterion is selected which produces the most uniform distribution, that is with the smallest peaks and valleys.

This method is particularly useful in the case of static files where the key values of all the records are known in advance.

We have assumed the key values as integers in our previous discussions, but it need not be so always. In fact, any key value can be represented by a string of characters and then ASCII values of its constituent characters can be taken to convert it into a numeric value. Thus, assuming that a key value $k = k_1k_2k_3 \dots k_n$, where each k_i is the constituent character in k. The hash function using the division method is stated as below in algorithm *HashDivision*.

Algorithm HashDivision

Input: K, the key value in the form of a string of characters whose hash address is to be calculated.

Output: *INDEX*, a positive integer as the hash address.

Data structure: Hash table in the form of an array. H is the size of the hash table which is used for modulo arithmetic operation.

```
Steps:
1. i = 1                              // i is the pointer to the string K
2. keyVal = 0                         // To store the keyvalue of K
3. While (K[i] ≠ NULL) do
4.     keyVal = keyVal + K[i]         // Add the ASCII value of K
5.     i = i + 1                      // Move to the next character
6. EndWhile
7. INDEX = keyVal MOD H + 1           // Find the remainder modulo
8. Return (INDEX)
9. Stop
```

The algorithms for other hash functions can be designed likewise; these have been left as exercises for the student.

Assignment 6.2

1. Write the algorithms *HashMidSquare*, *HashFoldingPure*, *HashFoldingShift*, *HashFoldingBoundary*, for the *midSquare* method and different variations of the folding method, respectively, for a given key value K in the form of a string of characters.
2. Suppose a file contains 100 records. What should be the size of the hash table and hence h? Create an array of key values of the records as hash table. Generate 100 random numbers and assume them as key values. Apply different hash functions to calculate hash addresses and load the key values into the hash table.
3. Generate 100 random numbers each of 6 digits, say. Assuming these as static key values, obtain a digital analysis on it based on three different criteria. Plot a graph for the hash values obtained with these criteria and then select the best criteria.

4. A hash function will be termed good if it satisfies both the criteria, namely, quick to compute and uniformity of distribution. The first criteria can be controlled by using a suitable technique of computation. For example, suppose, for a key $k = k_1k_2k_3$, the hash function is $H(k) = k_1 + k_2 * 27 + k_3 * 27^2$. Using Horner's rule for evaluating a polynomial, this can be calculated as:

$$H(k) = (k_3 * 27 + k_2) * 27 + k_1$$

Extend the Horner's rule to compute the hash address for the following:

(a) $H(k_1k_2k_3 \cdots k_r) = \sum_{i=1}^{r} k_i \, 27^{i-1}$

Hint: Additions can be replaced by the bit-wise exclusive-OR operation for increased speed.

(b) $H(k_1k_2k_3 \cdots k_r) = \sum_{i=1}^{r} k_i \, 32^{i-1}$

Hint: Additions can be replaced by the bit-wise exclusive-OR for increased speed and multiplication by 32 is not really a multiplication, it can be done by shifting the bit by 5.

(c) A hybrid hash function $H^*(k)$ of $H(k)$ can be obtained by applying the division method over $H(k)$ as stated below:

$$H^*(k) = H(k) \text{ MOD } h$$

h being the size of the hash table.

Obtain the algorithm for such a hybrid hash function.

6.4.2 Collision Resolution Techniques

Whatever the hash function used in hashing, the complete removal of collisions is almost impossible. This can be emphasized with an example called *birth day surprise*. Suppose there is a class of 24 students and they are having the same year of birth. We want to know the probability that two students have the same date of birth. The probability can be calculated as follows:

Open the calendar of the year of their birth. Assume that there are 365 days. Start with any student, and put a tick on his birthday date on the calendar. Now, the probability that the second student has a different birthday is 364/365. Tick this date off. The probability that a third student has a different birthday is now 363/365. Continuing this way, we see that if the first $(n - 1)$ students have different birthdays, then the probability that the nth student has a different birthday is

$$\frac{365 - (n - 1)}{365} \quad \text{or} \quad \frac{365 - n + 1}{365}$$

Since the birthdays of different people are independent, we obtain the probability that n students all have a different birthday is

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365 - n \times 1}{365}$$

This probability can be calculated as less than 0.5 whenever $n \geq 24$.

In other words, suppose there is a hash table of size 365 and we want to store the records of all the 24 students based on birthdays as their key values. It is therefore a fifty-fifty chance that two of the students have the same birthday and hence a collision.

So, collision in hashing cannot be ignored, whatever be the size of the hash table. The next question arises therefore is what to do if their is a collision? There are several techniques to resolve the collisions. Two important methods are listed below:

(a) Closed hashing (also called *linear probing*)
(b) Open hashing (also called *chaining*).

6.4.3 Closed Hashing

The simplest method to resolve a collision is *closed hashing*. Suppose there is a hash table of size h and the key value of interest is mapped to an address location i, with a hash function. The closed hashing then can be stated as follows:

Start with the hash address where the collision has occurred, let it be i. Then follow the following sequence of locations in the hash table and do the sequential search.

$$i,\ i + 1,\ i + 2,\ \ldots,\ h,\ 1,\ 2,\ \ldots,\ i - 1$$

The search will continue until any one of the following cases occurs:

- The key value is found.
- An unoccupied (or empty) location is encountered.
- The searches reaches the location where the search had started.

The first case corresponds to the successful search and the last two cases correspond to unsuccessful search. Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is also alternatively termed *linear probing*; probe means key comparison.

Let us illustrate the method with an example. Assume that there is a hash table of size 10 and the hash function uses the division method with remainder modulo 7, namely, $H(k) = k \bmod (7 + 1)$. Let us consider the build up of the hash table (initially, the table is empty) with the following set of key values:

15 11 25 16 9 8 12 8

The loading of the hash table will take place successively by performing a search for a key and inserting it into the table in an empty room if the key is not in the table and leaving if it is overflow, that is, no free room to accommodate any further key value. This is illustrated in Figure 6.8.

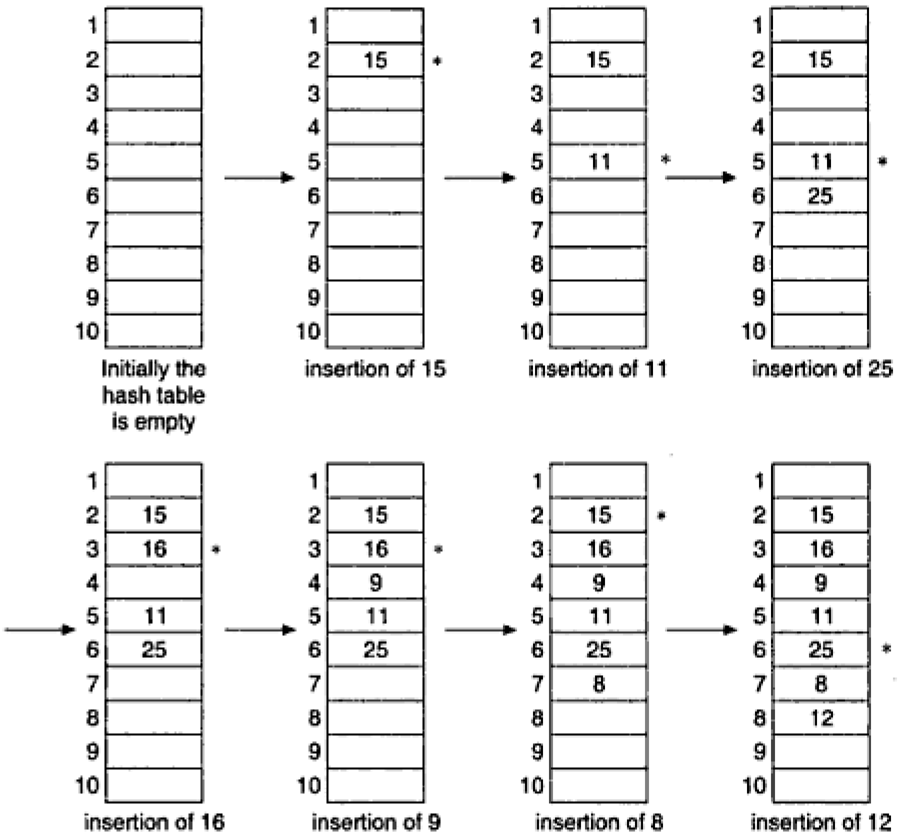


Figure 6.8 Building up a hash table.

Next, let us define the operation for searching a key-value and inserting a key-value. The algorithm *HashLinearProbe* for searching a key value *K* in a hash table of size *HSIZE* is given below:

Algorithm HashLinearProbe

Input: *K* is the key value of search. *INSERT* is a flag for the insertion operation.

Output: Return the location if it is found in the hash table else if *INSERT* is TRUE put *K* in the table if table has not overflown otherwise return NULL.

Data structures: A hash table *H* of size HSIZE in the form of an array.

```
Steps:
 1. flag = FALSE                          // Flag for continuation of looping
 2. index = HashFunction(K)     // Calculate the hash address using a hash function
 3. If (K = H[index]) then                                // If there is a hit
 4.     Return(index)
 5.     Exit                                          // End of the execution
 6. Else
 7.     i = index + 1                                // Set to the next location
```

(*Contd.*)

```
8.        While ($i \neq$ index) and (not flag) do
9.            If (($H[i]$ = NULL) or ($H[i] < 0$)) then              // If the cell is free
10.               If (INSERT) then                                   // True option for insertion
11.                   $H[i] = K$                                     // Put the key value into the hash table
12.                   flag = TRUE
13.               EndIf
14.           Else                                                   // Cell is occupied
15.               If ($H[i] = K$) then                               // Match
16.                   flag = TRUE
17.                   Return($i$)
18.                   Exit                                           // End of the execution
19.               Else                                               // No match
20.                   $i = i$ MOD $h$+1                              // Closed looping
21.               EndIf
22.           EndIf
23.       EndWhile
24.       If (flag = FALSE) and ($i$ = index) then   // No match and reach to the starting point
25.           Print "The table is overflow"
26.       EndIf
27.   EndIf
28.   Stop
```

Note Step 9 in the above algorithm. Here, we assume that whenever a key value is deleted from the hash table its corresponding entries are made negative instead of NULL. Writing an algorithm for deleting a key value is straightforward and is left as an exercise.

Drawback of closed hashing and its remedies

The major drawback of closed hashing is that, as half of the hash table is filled, there is a tendency towards *clustering*; that is key values are clustered in large groups and as a result a sequential search becomes slower and slower. This kind of clustering is typically known as *primary clustering*.

The following are some solutions known to avoid this situation:

(a) Random probing
(b) Double hashing or rehashing
(c) Quadratic probing.

Let us briefly discuss each of the above solutions.

Random probing. This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in the linear probing method. The random sequence generated by the pseudo random number generator contains all the positions between 1 and h, the highest location of the hash table. An example of a pseudo random number generator that produces such a random sequence of locations is given below:

$$i = (i + m) \text{ MOD } h + 1$$

where i is a number in the sequence, and m and h are integers that are relatively prime to each other (that is, their greatest common divisor is 1). For example, suppose $m = 5$ and $h = 11$ and initially $i = 2$, then the above-mentioned pseudo random number generator generates the sequence as:

$$8, 3, 9, 4, 10, 5, 11, 6, 1, 7, 2$$

We stop producing the numbers when the first location is duplicated. Observe that here all the numbers between 1 and 11 are generated but randomly. We can avoid primary clustering if the probe follows the said random sequence.

Double hashing. Random hashing however is not free form clustering. Another type of clustering, called *secondary clustering*, is involved here. In particular, clustering occurs when two keys are hashed into the same location. In such an instance, if the same sequence of locations is generated for two different keys by the random probing method then clustering takes place. An alternative approach to avoid the secondary clustering problem is to use a second hash function in addition to the first one. This second hash function results in the value of m for the pseudo random number generator as employed in the random probing method. This second function should be selected in such a way that the hash addresses generated by the two hash functions are distinct and the second function generates a value m for the key k so that m and h are relatively prime. Let us consider the following example.

Suppose $H_1(k)$ is the initially used hash function and $H_2(k)$ is the second one. These two functions are defined as

$$H_1(k) = (k \text{ MOD } h) + 1$$

$$H_2(k) = (k \text{ MOD } (h - 4)) + 1$$

Let $h = 11$ and $k = 50$ for an instance. Then, $H_1(50) = 7$ and $H_2(50) = 2$. Therefore, $H_1(50) \neq H_2(50)$, that is, H_1 and H_2 are independent and $m = 2$, $h = 11$ are relatively prime. Hence, using $i = [(i + 2) \text{ MOD } 11] + 1$, and initially $i = 7$, we have the random sequence as

$$10, 2, 5, 8, 11, 3, 6, 9, 1, 4, 7$$

Now, let us choose another key value which has the same hash address as that of 50 (that is, 7) with the first hash function H_1. Let it be 28 (since $H_1(28) = 28 \text{ MOD } 11 + 1 = 7$). Then

$$H_2(28) = 28 \text{ MOD } 7 + 1 = 5$$

So using $i = [(i + m) \text{ MOD } 11]$ with $i = 7$ and $m = 5$, we get the sequence:

$$2, 8, 3, 9, 4, 10, 5, 11, 6, 1, 7$$

Thus, for the two key values where the hash address is the same and using rehashing, two different random sequences are generated, thereby alleviating the secondary clustering.

Quadratic probing. Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations $i + 1$, $i + 2$, $i + 3$, etc. are probed; but in quadratic probing, the next locations to be probed are $i + 1^2$, $i + 2^2$, $i + 3^2$, etc. Mathematically, if h is the size of the hash table and $H(k)$ is the hash function then the quadratic probing searches the locations:

$$H(k) + i^2 \text{ MOD } h \quad \text{for } i = 1, 2, 3, \ldots$$

Note that in quadratic probing the increment function is i^2. It also assumes the hash table as close (or circular) as in linear probing.

This method, no doubt, substantially reduces primary clustering, but it does not probe all the locations in the table. Lemma 6.1 gives the information regarding the number of location that it can probe at most.

Lemma 6.1

If h denotes the size of the hash table then the number of distinct positions that will be probed is $(h + 1)/2$.

Proof: Suppose that the hash address for a given key k is x. Then the ith probe will look like

$$x + i^2 \bmod h = x + [1 + 3 + 5 + \cdots + (2i - 1)] \bmod h$$

or,

$$(2i - 1) \le h$$

i.e.

$$i = \frac{h+1}{2} \tag{6.2}$$

Hence proved.

Example: Suppose h =11 and the hash address of the key is x. Then the different locations with a quadratic probe are x, $x + 1$, $x + 4$, $x + 9$, $x + 5$, $x + 3$ with $(11 + 1)/2 = 6$ probes.

Drawback of quadratic probing: For linear probing, it is not advisable to let the hash table get nearly full because in that case we may have to search the entire table and thus performance degrades. For quadratic probing, the situation is even more drastic: there is no guarantee of finding an empty cell once more than half of the table gets full or even before that if the table size is not prime. Lemma 6.2 supports the above situation.

Lemma 6.2

If quadratic probing is used and the table size is prime, then a new key value can always be inserted if the table is at least half full.

Proof (By the method of contradiction): Let the table size h be an (odd) prime number greater than 3. We show that the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Two of these locations are

$$x + i^2 \text{ MOD } h \quad \text{and} \quad x + j^2 \text{ MOD } h$$

where $0 < i, j \le \lfloor h/2 \rfloor$, and x is the hash address of a key. Suppose by contradiction, these locations are the same, but $i = j$. Then

$$x + i^2 \text{ MOD } h = x + j^2 \text{ MOD } h$$

or

$$(i^2 - j^2) \text{ MOD } h = 0$$

or

$$(i - j) \times (i + j) \text{ MOD } h = 0$$

Since h is prime, it follows that either $i - j$ or $i + j$ is divisible by h. Again $i \neq j$, $i, j \leq \lfloor h/2 \rfloor$, so $(i - j)$ MOD $h \neq 0$. The second option is also not possible as $i, j < \lfloor h/2 \rfloor$, their sum can never be $m \times h$, for $m = 1, 2, 3, \ldots$.

Thus, the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Since the element to be inserted can also be placed in the location to which it hashes, if there are no collisions, any element has $\lfloor h/2 \rfloor$ locations into which it can be placed. Hence, proved.

In quadratic probing, it is also very crucial that the table size should be a prime. If the table size is not prime, the number of alternate locations can be severely reduced. As an example, if the table size is 16, (or a power of 2), then the only alternate locations would be at distances 1, 4, 9, etc.

6.4.4 Open Hashing

So far we have discussed the closed hashing methods of collision resolution. The closed hashing method deals with arrays as hash tables and thus we are able to refer quickly to random positions in the tables. But there are two main difficulties with this technique: First, it is very difficult to handle the situation of table overflow in a satisfactory manner. Second, the key values are haphazardly intermixed and, on the average, the majority of the keys are far from their hash locations, thus increasing the number of probes which degrades the overall performance.

To resolve these problems another hashing method called *open hashing* (also called *separate chaining*, or simply *chaining*) is known. The chaining method is discussed in the following paragraphs.

The chaining method uses a hash table as an array of pointers; each pointer points a linked list. That is, here the hash table is an array of list headers. In Figure 6.9, a hash table of size 10 is considered. The index of the hash table varies from 0 to 9 and key values are taken as integers. The hash address for a key is decided by its last digit (means the right most digit).

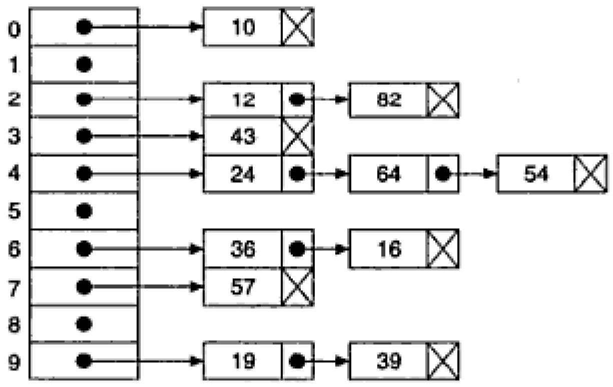


Figure 6.9 An open hashing.

For a given key value, the hash address is calculated. It then searches the linked list pointed by the pointers at that location. If the element is found it returns the pointer to the node containing that key value else inserts the element at the end of that list. The implementation of open hashing is stated in the algorithm *HashChaining* as follows:

Algorithm HashChaining

Input: K is the item of interest. *INSERT* is a flag for the option of insertion.
Output: If K is found in the hash table then return the pointer of the node which contains the key value K else insert K into the linked list when the *INSERT* flag is TRUE.
Data structure: Hash table H having size *HSIZE* storing pointer to the single linked list structure.

Steps:

```
 1. index = HashFunction(K)                      // Calculate the hash address of K
 2. ptr = H[index]                               // ptr is a pointer to any node in the list
 3.  flag = FALSE                                // flag for controlling the search
 4. While (ptr ≠ NULL) and (flag = FALSE) do
 5.     If (ptr→DATA = K) then                   // End of search
 6.         flag = TRUE
 7.         Return(ptr)
 8.         Exit                                 // End of execution
 9.     Else
10.         ptr = ptr.LINK                       // Move to the next node
11.     EndIf
12. EndWhile
13. If (flag = FALSE) then
14.     Print "Key value does not exist"
15.     If (INSERT) then
16.         InsertEnd_SL(H[index])               // Insert it into the table
17.     EndIf
18. EndIf
19. Stop
```

A key value if it exist can be deleted from a hash table for which a procedure *HashKeyDelete*(...) can be written. This is left as an exercise for the reader.

Advantages and disadvantages of chaining

There are several advantages of the chaining method. The most important advantages are stated below:

1. An overflow situation never arises. The hash table maintains lists which can contain any number of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.

3. Insertion and deletion become a quick and an easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in a single linked list.
4. Finally, open hashing is best suitable in applications where the number of key values varies drastically as open hashing uses dynamic storage management policy.

The only disadvantage of the chaining method is that of maintaining linked lists and extra storage space for link fields.

Assignment 6.3

As an alternative to the collision resolution technique, *bucket hashing* can be used. In this method, the hash table is a collection of buckets and each bucket contains a few number of key values decided by the bucket size. Let us assume that all buckets are of the same size. Here, the hash function calculates an address of a bucket and then finally the key value is searched in that bucket. Figure 6.10 illustrates this concept for a hash table with buckets whose size is 3.

(a) Compare bucket hashing with open hashing and closed hashing.

(b) Write algorithms to search a key value, insert a key value and delete a key value in bucket hashing.

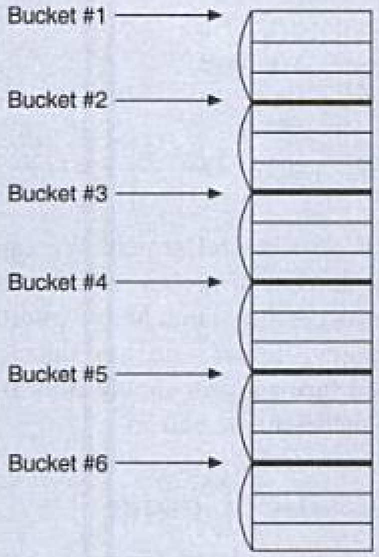


Figure 6.10 Bucket hashing.

6.4.5 Comparison of Collision Resolution Techniques

We will conclude the discussion of hash tables by giving an analytical comparison of various collision resolution techniques discussed. Let us define the *load factor*, λ, of a hash table as

$$\lambda = \frac{\text{Total number of key values}}{\text{Size of the hash table}} \tag{6.3}$$

So $\lambda = 1.0$ means that the number of key values is the same as the total capacity of the hash table. We also define $S(\lambda)$ and $U(\lambda)$ as

$S(\lambda)$ = average number of probes for a successful search.
$U(\lambda)$ = average number of probes for an unsuccessful search.

These two quantities will measure the performance of collision resolution methods.

Analysis of closed hashing

To analyze the performance of closed hashing, let us assume the case of random probing and ignore the problem of clustering for the sake of simplicity.

Let us first consider the case of unsuccessful search. It is evident that the probability that the first probe hits an occupied cell is λ, the load factor. The probability that a probe hit an empty cell is $1 - \lambda$. The probability that the unsuccessful search terminates in exactly two probes is therefore $\lambda(1 - \lambda)$. Arguing similarly this way, the probability that exactly k probes are made in an unsuccessful search is $\lambda^{k-1}(1 - \lambda)$. The average number of probes for an unsuccessful search is therefore

$$U(\lambda) = \sum_{k=1}^{\infty} k\lambda^k (1-\lambda) = (1-\lambda)\sum_{k=1}^{\infty} k\lambda^k \tag{6.4a}$$

Since $\lambda \le 1$ and $\sum_{k=1}^{\infty} k\lambda^k = \frac{1}{(1-\lambda)^2}$, we have

$$U(\lambda) = (1-\lambda)\frac{1}{(1-\lambda)^2} = \frac{1}{1-\lambda} \tag{6.4b}$$

Next, let us consider the case of a successful search. We can think of this problem through insertion of key values. Then the number of probes required will be exactly one more than the number of probes made in the unsuccessful search before inserting the item. Let us consider the case when the table is initially empty. In that state, key values are inserted one at a time. Now as the items are inserted, the load factor grows slowly from 0 to λ. Thus, we can express the average number of probes in a successful search as

$$\begin{aligned} S(\lambda) &= \frac{1}{\lambda}\int_0^{\lambda} U(x)\,dx \\ &= \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}\,dx \\ &= \frac{1}{\lambda}\ln\frac{1}{1-\lambda} \end{aligned} \tag{6.5}$$

A similar calculation can be performed for closed hashing with linear probing. This is left as an assignment for the student.

Assignment 6.4

For closed hashing with linear probing prove that:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \tag{6.6a}$$

and

$$U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right) \tag{6.6b}$$

Analysis of open hashing

Let us recall the case of chaining. In chaining, we move to the linked list before doing any probes. Suppose that a list contains n key values. Assuming that the key values are equally probable in any list, the expected number of key values on any list is n/h, h being the size of the hash table. This is nothing but l, the load factor. Now, if the list contains n items, the number of key comparisons for an unsuccessful search is n. Thus, the average number of probes for an unsuccessful search is

$$U(\lambda) = \lambda \tag{6.7}$$

Now, suppose the search is successful. From the analysis of sequential search over a list of n items, we can write

$$\text{Number of comparisons } = \frac{1}{n}\sum_{i}^{n} i = \frac{n+1}{2} \tag{6.8}$$

Assume that an item is equally probable in any place. Since the average number of key values in any list is λ, the average number of probes in a successful search is

$$S(\lambda) = \frac{\lambda+1}{2} \tag{6.9}$$

We can draw several conclusions from the results thus obtained. Let us draw a graph (Figure 6.11) for these results. From this graph, the following points are evident:

1. Open hashing always requires fewer probes than closed hashing.
2. Chaining is especially advantageous when the load factor is significantly low.
3. With closed hashing and successful search, linear probing is not significantly slower if λ is high. For unsuccessful searches, however, clustering will occur which quickly degenerates into a long sequential search.

We might therefore conclude that if searches are quite likely to be successful and the load factor is moderate, closed hashing is quite satisfactory, but in other circumstances open hashing is promising.

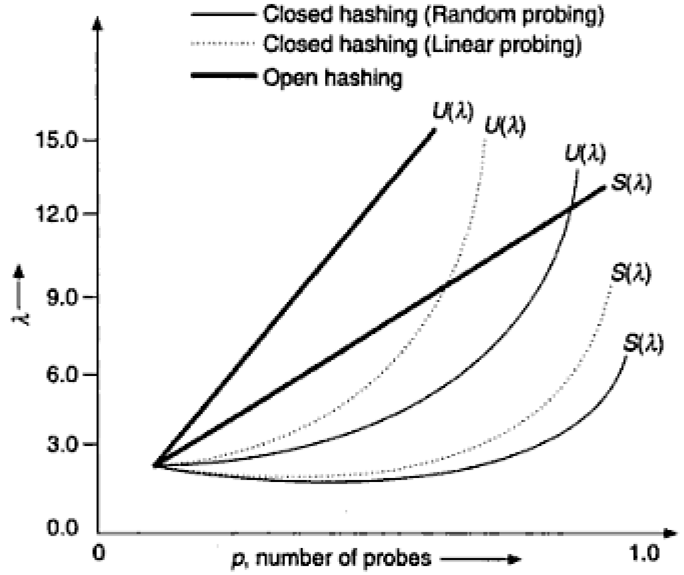


Figure 6.11 Comparison of various collision resolution techniques.

6.5 PROBLEMS TO PONDER

6.1 Suppose there is a hash table of size H. If λ be the load factor and ω be the word size for a key value, then find the total storage space required for the following cases:

(a) Open hashing (assume that one word is required for a link field)

(b) Closed hashing.

6.2 A hash function is defined as $H(k) = r_i$, where $r_1, r_2, \ldots, r_n$ is a sequence of random numbers between 1 and n (each integer appears exactly once).

(a) Prove that if the hash table is not full then this hashing always resolves collision.

(b) Does this technique eliminate clustering?

(c) If l be the load factor of the table, what is the expected time for a

(i) successful search?

(ii) unsuccessful search?

6.3 In quadratic hashing, the probes are carried out in the sequence

$$H(k) + q^2,\ H(k) + (q-1)^2,\ \ldots,\ H(k) + 1,\ H(k),\ H(k) - 1,\ \ldots,\ H(k) - q^2$$

where $q = (h-1)/2$, h being the size of the hash table. Prove that this method resolves collision and avoids clustering.

6.4 In open hashing, we can save time if the nodes in chain are kept in order by key value. How many probes, on an average will be done in the case of

(a) unsuccessful search?

(b) successful search?

6.5 The hash function should be such that it can be calculated very quickly.

(a) Show how if i^2 is known then $(i + 1)^2$ can be obtained from it by addition only.

(b) Show that the following expression generates random numbers between 1 and m, if m and c are prime to each other:

$$y = (y + c) \text{ MOD } m \text{ expression}$$

Assume a suitable starting value for y.

REFERENCES

Bell, J., A hash code eliminating secondary clustering, The quadratic quotient method, *Communication of the ACM*, 13, 1970.

Enbody, R.J. and H.C. Du, Dynamic hashing schemes, *Computing Surveys*, 20, 1988.

Gonnet, G.H. and R. Baeza Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Massachusetts, 1988.

Gotlieb, C.C. and L.R. Gotlieb, *Data Types and Structures*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

Guibas, L.J. and E. Szemerdi, The analysis of double hashing, *Sciences*, 16, 1978.

Knuth, D.E., Sorting and Searching, *The Art of Computer Programming*, 3, Addison-Wesley, Reading, Massachusetts, 1984.

Maurrer, W.D. and T.G. Lewis, Hash table methods, *Computing Surveys*, 7, 1995.

McKenzie, B.J., R. Harries, and T. Bell, Selecting a hashing algorithm, *Software Practice and Experience*, 20, 1990.

Morris, R., Scatter storage techniques, *Communication of the ACM*, 11, 1998.