There is no optimal sort that is $O(n)$ irrespective of the contents or order of the input. Time requirements of most of the classical sorts vary from $O(n2)$ to $O(n \log n)$. In most of the cases, the time required for sorting depends on the original sequence of data. For some sorts where the input data is almost in sorted order, the sorting time is of $O(n)$. When the input data is in reverse sorting order, the sorting time would be $O(n^2)$. For other sorts, the time required for sorting irrespective of the order of the original data is $O(n \log n)$. For sort selection, there is only the file but also other factors contributing to sorting time are to be considered. However there is no such thing as best general sort technique, and the selection of a particular technique is dependent on the specific circumstances.

Sort selection is to be followed by efficiency in programming. Time considerations take upper hand compared to space constraints. The ideal sort is the in-place sort wherein the elements to be sorted are manipulated within the space occupied by the original unsorted input. The additional space required is in the form of constant number of locations regardless of the input size.

In terms of searching, it is proven that the best possible time complexity to search n objects would be $O(\log_2 n)$.

9.10 HASHING

The search time of each algorithm discussed so far depends on the number n of elements in the collection S of data. This section discusses a searching technique, called *hashing* or *hash addressing*, which is essentially independent of the number n.

The terminology which we use in our presentation of hashing will be oriented toward file management. First of all, we assume that there is a file F of n records with a set K of keys which uniquely determine the records in F. Secondly, we assume that F is maintained in memory by a table T of m memory locations and that L is the set of memory addresses of the locations in T. For notational convenience, we assume that the keys in K and the addresses in L are (decimal) integers. (Analogous methods will work with binary integers or with keys which are character strings, such as names, since there are standard ways of representing strings by integers.)

The subject of hashing will be introduced by the following example.

Example 9.12

Suppose a company with 68 employees assigns a 4-digit employee number to each employee which is used as the primary key in the company's employee file. We can, in fact, use the employee number as the address of the record in memory. The search will require no comparisons at all. Unfortunately, this technique will require space for 10 000 memory locations, whereas space for fewer than 30 such locations would actually be used. Clearly, this tradeoff of space for time is not worth the expense.

The general idea of using the key to determine the address of a record is an excellent idea, but it must be modified so that a great deal of space is not wasted. This modification takes the form of a function H from the set K of keys into the set L of memory addresses. Such a function.

$$H: K \to L$$

is called a *hash function* or *hashing function*. Unfortunately, such a function H may not yield distinct values; it is possible that two different keys k_1 and k_2 will yield the same hash address.

This situation is called *collision*, and some method must be used to resolve it. Accordingly, the topic of hashing is divided into two parts: (1) hash functions and (2) collision resolutions. We discuss these two parts separately.

Hash Functions

The two principal criteria used in selecting a hash function $H: K \to L$ are as follows. First of all, the function H should be very easy and quick to compute. Second the function H should, as far as possible, uniformly distribute the hash addresses throughout the set L so that there are a minimum number of collisions. Naturally, there is no guarantee that the second condition can be completely fulfilled without actually knowing beforehand the keys and addresses. However, certain general techniques do help. One technique is to "chop" a key k into pieces and combine the pieces in some way to form the hash address $H(k)$. (The term "hashing" comes from this technique of "chopping" a key into pieces.)

We next illustrate some popular hash functions. We emphasize that each of these hash functions can be easily and quickly evaluated by the computer.

(a) *Division method.* Choose a number m larger than the number n of keys in K. (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.) The hash function H is defined by

$$H(k) = k \pmod m \qquad \text{or} \qquad H(k) = k \pmod m + 1$$

Here $k \pmod m$ denotes the remainder when k is divided by m. The second formula is used when we want the hash addresses to range from 1 to m rather than from 0 to $m - 1$.

(b) *Midsquare method.* The key k is squared. Then the hash function H is defined *by*

$$H(k) = l$$

where l is obtained by deleting digits from both ends of k^2. We emphasize that the same positions of k^2 must be used for all of the keys.

(c) *Folding method.* The key k is partitioned into a number of parts, $k_1, \ldots, k_r$, where each part, except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry. That is,

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

where the leading-digit carries, if any, are ignored. Sometimes, for extra "milling," the even-numbered parts, $k_2, k_4, \ldots$, are each reversed before the addition.

Example 9.13

Consider the company in Example 9.9, each of whose 68 employees is assigned a unique 4-digit employee number. Suppose L consists of 100 two-digit addresses: 00, 01, 02, ..., 99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

(a) *Division method.* Choose a prime number m close to 99, such as $m = 97$. Then

$$H(3205) = 4, \qquad H(7148) = 67, \qquad H(2345) = 17$$

That is, dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, and dividing 2345 by 97 gives a remainder of 17. In the case that the memory

addresses begin with 01 rather than 00, we choose that the function $H(k) = k \pmod m) + 1$ to

division:

$H(3205) = 4 + 1 = 5$, $H(7148) = 67 + 1 = 68$, $H(2345) = 17 + 1 = 18$

middle-square method. The following calculations are performed:

k:	3205	7148	2345
k^2:	10 272 025	51 093 904	5 499 025
$H(k)$:	72	93	99

folding method. Chopping the key k into two parts and adding yields the following hash addresses:

$H(3205) = 32 + 05 = 37$, $H(7148) = 71 + 48 = 19$, $H(2345) = 23 + 45 = 68$

Observe that the leading digit 1 in $H(7148)$ is ignored. Alternatively, one may want to reverse the second part before adding, thus producing the following hash addresses:

$H(3205) = 32 + 50 = 82$, $H(7148) = 71 + 84 + 55$, $H(2345) = 23 + 54 = 77$

Subtraction method. In some cases, keys are consecutive but they do not start from 1. In such cases, we subtract a number from the key to determine the address. For example if a company has 200 employees, then the employee numbers can start from 2001, it need not start with 1. In this case, we use subtraction hashing, a simple hashing method and subtract 2000 from the key to determine the address.

Example 9.14

Figure 9.13 shows the hash table of 10 students whose roll numbers start from 101 and goes up to 110. Assume that you have to find the address of the student Mary.

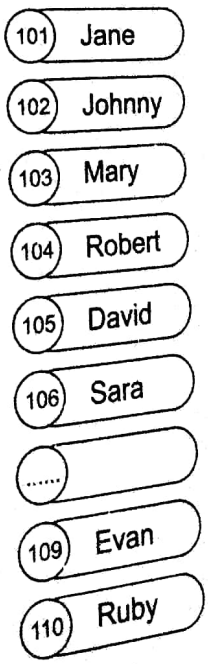

Fig. 9.13 Hash table

Using subtraction method, just subtract 100 from the key 103. Therefore the address is 3.

The following program demonstrates implementation of Hash function in C using subtraction nethod as the basis:

Program 9.8

```
#include <stdio.h>
#include <conio.h>

void main()
{
   int HASH(int);
   int i;
   clrscr();

   printf("Press any key to generate the Hash Table for Employee
   Code (keys) 2000-2020 ");
   getch();
   printf("\n\n*****HASH TABLE*****\n");
   printf("\nKey\tAddress");
   for(i=2000;i<=2020;i++)
     printf("\n%d\t   %d",i,HASH(i));

   getch();
}

int HASH(int k)
{
   return(k-2000);
}
```

Output:

```
Press any key to generate the Hash Table for Employee Code (keys)
2000-2020

*****HASH TABLE*****

Key       Address
2000        0
2001        1
2002        2
2003        3
2004        4
2005        5
2006        6
2007        7
2008        8
```

```
9
10
11
12
13
14
15
16
17
18
19
20
```

Digit extraction method. The selected keys are extracted from the key and made use as its address using a method called digit extraction. In this case, we select specific digits from the key k and use it as an address. For example, suppose we want to hash a 6 digit employee number, say 123457, to a three digit address, we could select the first, third, and fourth digits, from the left, and use them as an address. So our address will be 124.

Example 9.15

Roll number of a student Williams is 160252. Hash the number to a three digit address using digit extraction method.

Let just select the first, third and fourth digits, and use it as the address

$$160252 \rightarrow 102$$

Therefore the address is 102.

Rotation hashing method. This method is generally not used by itself, but is used in combination with other hashing methods. This method is especially useful when keys are assigned serially, as in the case of employee numbers. A simple hashing algorithm is used to create synonyms when hashing keys are identical except for the last character. In Fig. 9.14, the keys are rotated using this method.

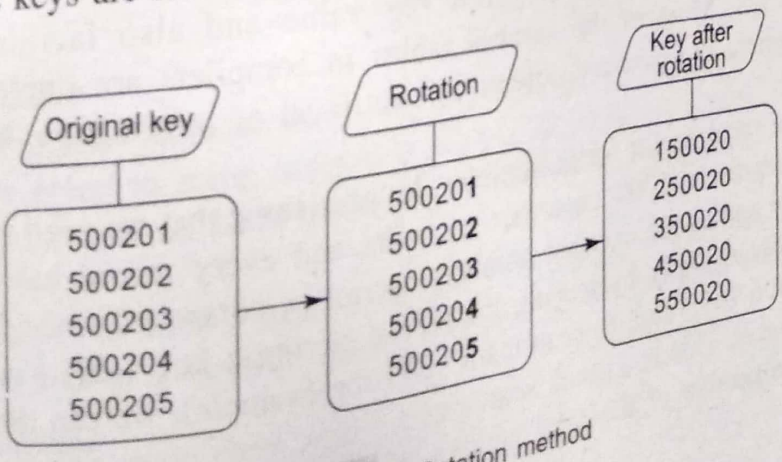

Fig. 9.14 Rotation method

Notice that all keys end in 50020. Rotation is generally used in combination with the folding method.

Example 9.16

Figure 9.15 shows the hash table of five students. Arrange the keys using rotation hashing method

200101	Jane
200102	Johnny
200103	Mary
200104	Robert
200105	David

Fig. 9.15 Hash table

Using the rotation hashing method, the keys can be rearranged as shown below in Fig. 9.16:

120010	Jane
220010	Johnny
320010	Mary
420010	Robert
520010	David

Fig. 9.16 Hash table after rearranging keys rotation hashing method

Hash Table

The data structure which is used for storing records is called a *hash table*. It enables us to search a record rapidly by making use of a given key value and also facilitates easy insertion and deletion of records. Most of the symbol tables in compilers are implemented using hash tables. In systems where the primary objective is retrieval of information, hash tables are used extensively.

For illustrating the requirement for hash tables, let us assume that we need to store 1000 records which are used to represent 1000 customer accounts and every record has an account number which is also the key and as the account numbers cannot be controlled since they range from 1 to 1000. This can be solved easily by storing the record having a key value i in the array data's i^{th} element. Then, given a key value to search for (789 for example), we can directly go to element 789 of the data to find it. This is a quick search as we have been given a record and we also know where to find it. Irrespective of the total number of records which are stored, the search time is

However, in these type of searches, there is a need to assign key values which are within 1000 in the above example) a fixed range of array index. We also need to have considerable ge space to favor all the records.

ow suppose we are dealing with records representing information about an organization's loyees, and that the keys are social security numbers. In cases when there are 1000 employees, e can be 10^9 social security numbers possibly. And there is no possibility to know which of 10^9 social security numbers will be allocated to 1000 employees in advance. In situations these when there is a need for allocating storage making use of social security numbers, e are two major problems. The first one would be lack of storage. Secondly when there is gh storage available, 999,999,000 ($10^9 - 10^3$) storage elements will be wasted. The main ective of a hash table and hash searching is to help directly locate records using the given value.

ample 9.17

ure 9.17 shows a hash table of the names of the students in a class. The keys are the names. e run the hash method on the keys, and store the keys at the index given by the hash method as own below.

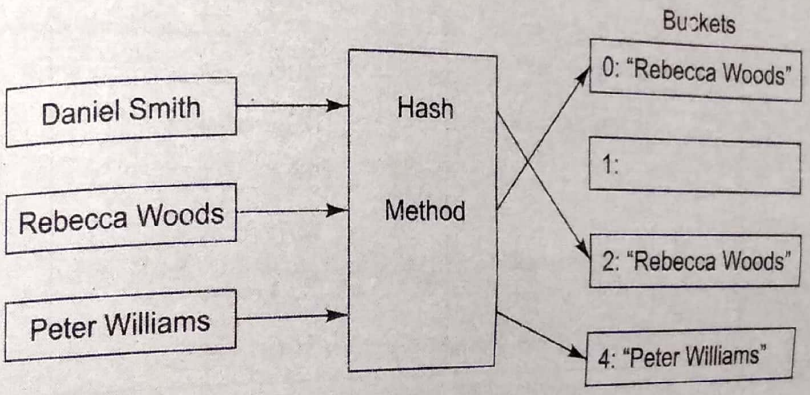

Fig. 9.17 Hash table

ilding a Hash Table

ppose there are eighteen records, each with a three-digit numeric key value. The records and r key values are given in some order that cannot be predicted easily. The requirement is to ld a table in the form of an array containing all the 18 key values, so that searching can be formed efficiently.

One way to do this is to build a ***hash table*** in which the key values can be searched to access relevant records. Such a hash table can be built by implementing the following:

- A hashing function
- Appropriate collision resolution policy

A hashing function is nothing but a method for calculating the table address pertaining to a key. This function is useful only in cases only where the address can be calculated quickly. Further, an adequate collision resolution policy is also implemented along with the hashing function to resolve the situations where duplicate table address values are generated.

Consider the hash table shown in Table 9.2. The hash function used here assigns to any key the remainder after division of the key by the table size. If the table size is m and key value = k, then remainder = k mod m = remainder.

In the following table, we can see that the eighteen key values along with their order. We can also see the hashing function values that have been assigned to each one of them. With division-type hashing functions, the divisor, or table size, is usually a prime number greater than the number of entries to be placed in the table. Here, we have taken this prime number to be 23.

Table 9.2 Hash Addresses

Hashing		*Hashing*	
Key Value	*Address*	*Key Value*	*Address*
018 →	19	468 →	08
392 →	01	814 →	09
179 →	18	720 →	07
359 →	14	260 →	07
663 →	19	802 →	20
262 →	09	364 →	19
639 →	18	976 →	10
321 →	22	774 →	15
097 →	05	566 →	14

Let us now analyze how the hashing address for 019 has been computed. Since the size of the table is 23, hence 019 = 0 × 23 + 19, i.e. 19 is the remainder. Thus, 19 is assigned as the corresponding hashing address. Similarly, the hashing address can be calculated as each key value is entered, and the key value is placed into that element of the table. Figure 9.18(a) shows a snapshot of the table after the first four key values are entered.

When the input is 663 i.e. the fifth key value, 19 is its hashing address, but we can also see that the 19th element of the table has already been occupied by another key value. This event is known as *collision*. The *collision resolution policy* is the method used to determine where to store a value that has undergone collision. *Linear probing* is one of the most straightforward policies for solving all the such conflicts. It proceeds through the table starting form the collision element and places the colliding entry at the first element which is found unoccupied. On reaching the end of the table in search of an empty element, the table circles to the top. In the above situation, 663 is entered at element 20. The final table is shown in Fig. 9.18(b). A total of eight initial collisions occurred in building it.

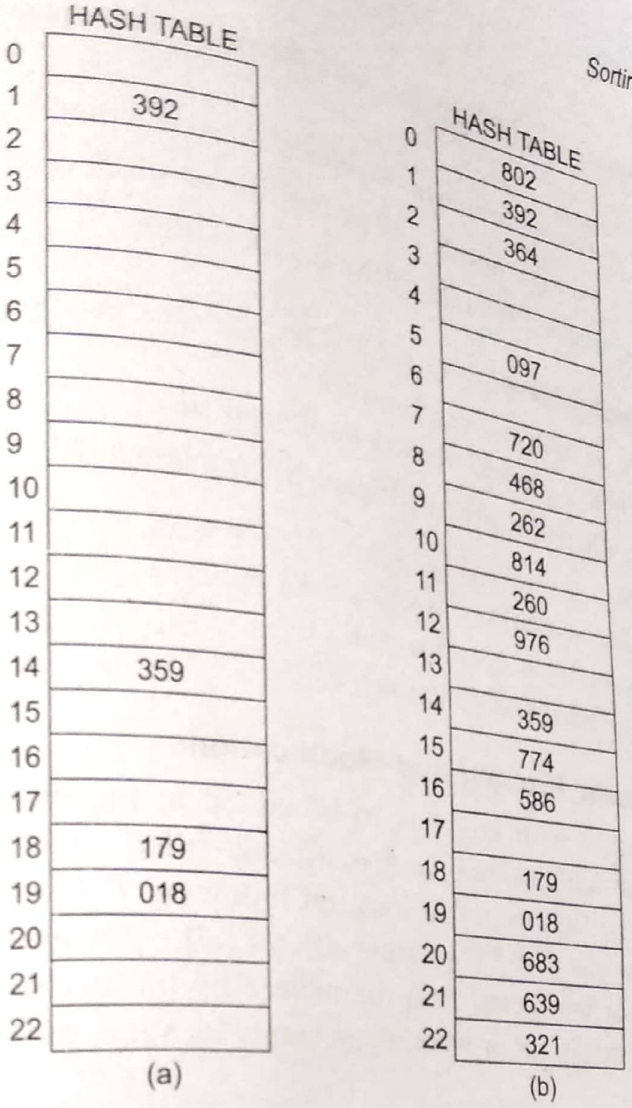

Fig. 9.18 Hash Table (a) after Four Entries and (b) Finally

...ery time a location of the table is visited for trying to attempt to insert a key value, we have ...*one probe*. Thus, the entering of first key value requires one probe; similarly we need two ...es are required for entering the fifth key value.

...ision Resolution

...ose we want to add a new record R with key k to our file F, but suppose the memory location ...ess $H(k)$ is already occupied. This situation is called *collision*. This subsection discusses two ...eral ways of resolving collisions. The particular procedure that one chooses depends on many ...ors. One important factor is the ratio of the number n of keys in K (which is the number of ...ords in F) to the number m of hash addresses in L. This ratio, $\lambda = n/m$, is called the *load factor*. ...First we show that collisions are almost impossible to avoid. Specifically, suppose a student ...s has 24 students and suppose the table has space for 365 records. One random hash function is ...hoose the student's birthday as the hash address. Although the load factor $\lambda = 24/365 \approx 7\%$ is ...small, it can be shown that there is a better than fifty-fifty chance that two of the students ...the same birthday.

...e efficiency of a hash function with a collision resolution procedure is measured by the ...ge number of *probes* (key comparisons) needed to find the location of the record with a given

key k. The efficiency depends mainly on the load factor λ. Specifically, we are interested in the following two quantities:

$S(\lambda)$ = average number of probes for a successful search

$U(\lambda)$ = average number of probes for an unsuccessful search

These quantities will be discussed for our collision procedures.

Collision Resolution Techniques

There are two broad ways of collision resolution:

(i) Open Addressing, where an array-based implemented.

(ii) Separate Chaining, where an array of linked list implemented.

Open Addressing includes:

- Linear probing (linear search)
- Quadratic probing (nonlinear search), and
- Double hashing (uses two hash functions).

(i) Open Addressing: Linear Probing and Modifications

Suppose that a new record R with key k is to be added to the memory table T, but that the memory location with hash address $H(k) = h$ is already filled. One natural way to resolve the collision is to assign R to the first available location following $T[h]$. (We assume that the table T with m locations is circular, so that $T[1]$ comes after $T[m]$.) Accordingly, with such a collision procedure, we will search for the record R in the table T by linearly searching the locations $T[h]$, $T[h+1]$, $T[h+2]$, ... until finding R or meeting an empty location, which indicates an unsuccessful search.

The above collision resolution is called *linear probing*. The average numbers of probes for a successful search and for an unsuccessful search are known to be the following respective quantities:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

(Here $\lambda = n/m$ is the load factor.)

The table below summarizes the characteristics of the various open addressing probing sequences.

Table 9.3 Characteristics of the open addressing probing sequences

probing sequence	*primary clustering*	*capacity limit*	*size restriction*
linear probing	yes	none	none
quadratic probing	no	$\lambda < \frac{1}{2}$	M must be prime
double hashing	no	none	M must be prime

terested in the

ut that the
esolve the
he table T
collision
ons $T[h]$,
successful

bes for a
espective

probing

on
me
me

Example 9.18

Suppose the table T has 11 memory locations, $T[1]$, $T[2]$, ..., $T[11]$, and suppose the file F consists of 8 records, A, B, C, D, E, X, Y and Z, with the following hash addresses:

Record:	A,	B,	C,	D,	E,	X,	Y,	Z
$H(k)$:	4,	8,	2,	11,	4,	11,	5,	1

Suppose the 8 records are entered into the table T in the above order. Then the file F will appear in memory as follows:

Table T:	X,	C,	Z,	A,	E,	Y,	—,	B,	—,	—,	D
Address:	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11

Although Y is the only record with hash address $H(k) = 5$, the record is not assigned to $T[5]$, since $T[5]$ has already been filled by E because of a previous collision at $T[4]$. Similarly, Z does not appear in $T[1]$.

The average number S of probes for a successful search follows:

$$S = \frac{1+1+1+1+2+2+2+3}{8} = \frac{13}{8} \approx 1.6$$

The average number U of probes for an unsuccessful search follows:

$$U = \frac{7+6+5+4+3+2+1+2+1+1+8}{11} = \frac{40}{11} \approx 3.6$$

The first sum adds the number of probes to find each of the 8 records, and the second sum adds the number of probes to find an empty location for each of the 11 locations.

One main disadvantage of linear probing is that records tend to *cluster*, that is, appear next to one another, when the load factor is greater than 50 percent. Such a clustering substantially increases the average search time for a record. Two techniques to minimize clustering are as follows:

1. *Quadratic probing.* Suppose a record R with key k has the hash address $H(k) = h$. Then, instead of searching the locations with addresses h, $h + 1$, $h + 2$, ..., we linearly search the locations with addresses

$$h,\ h+1,\ h+4,\ h+9,\ h+16,\ \ldots,\ h+i^2,\ldots$$

If the number m of locations in the table T is a prime number, then the above sequence will access half of the locations in T.

2. *Double hashing.* Here a second hash function H' is used for resolving a collision, as follows. Suppose a record R with key k has the hash addresses $H(k) = h$ and $H'(k) = h' \neq m$. Then we linearly search the locations with addresses

$$h,\ h+h',\ h+2h',\ h+3h',\ \ldots$$

If m is a prime number, then the above sequence will access all the locations in the table T.

Remark: One major disadvantage in any type of open addressing procedure is in the implementation of deletion. Specifically, suppose a record R is deleted from the location $T[r]$. Afterwards, suppose we meet $T[r]$ while searching for another record R'. This does not necessarily mean that the search is unsuccessful. Thus, when deleting the record R, we must label the location $T[r]$ to indicate that it previously did contain a record. Accordingly, open addressing may seldom be used when a file F is constantly changing.

(ii) Chaining

Chaining involves maintaining two tables in memory. First of all, as before, there is a table T in memory which contains the records in F, except that T now has an additional field LINK which is used so that all records in T with the same hash address h may be linked together to form a linked list. Second, there is a hash address table LIST which contains pointers to the linked lists in T.

Suppose a new record R with key k is added to the file F. We place R in the first available location in the table T and then add R to the linked list with pointer $LIST[H(k)]$. If the linked lists of records are not sorted, then R is simply inserted at the beginning of its linked list. Searching for a record or deleting a record is nothing more than searching for a node or deleting a node from a linked list, as discussed in Chapter 5.

The average number of probes, using chaining, for a successful search and for an unsuccessful search are known to be the following approximate values:

$$S(\lambda) \approx 1 + \frac{1}{2}\lambda \qquad \text{and} \qquad U(\lambda) \approx e^{-\lambda} + \lambda$$

Here the load factor $\lambda = n/m$ may be greater than 1, since the number m of hash addresses in L (not the number of locations in T) may be less than the number n of records in F.

Example 9.19

Consider again the data in Example 9.18, where the 8 records have the following hash addresses:

Record:	A,	B,	C,	D,	E,	X,	Y,	Z
$H(k)$:	4,	8,	2,	11,	4,	11,	5,	1

Using chaining, the records will appear in memory as pictured in Fig. 9.19. Observe that the location of a record R in table T is not related to its hash address. A record is simply put in the first node in the AVAIL list of table T. In fact, table T need not have the same number of elements as the hash address table.

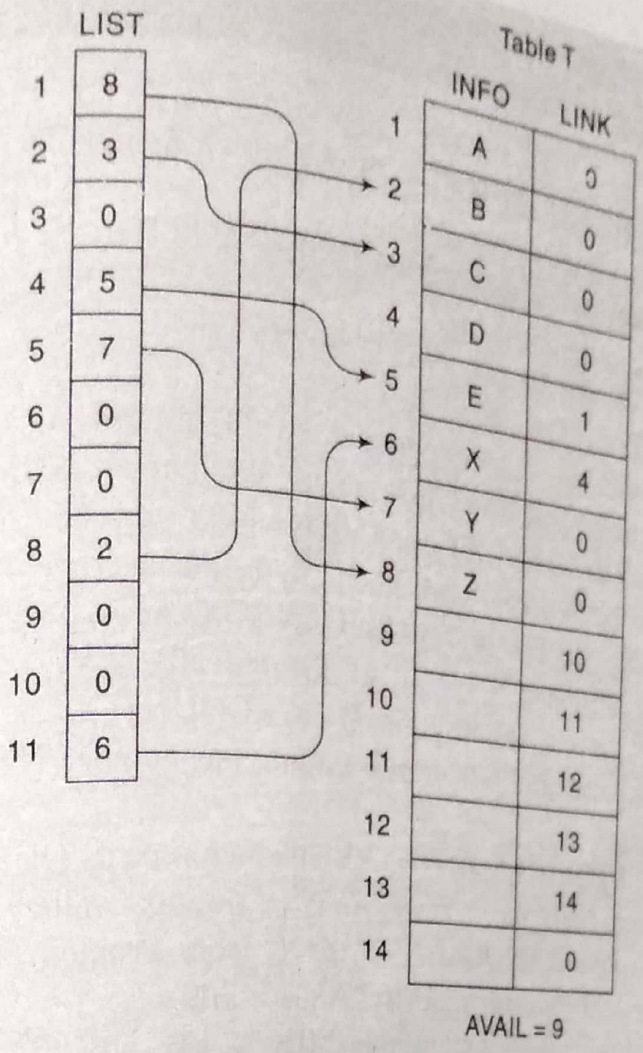

Fig. 9.19

The main disadvantage to chaining is that one needs $3m$ memory cells for the data. Specifically, re are m cells for the information field INFO, there are m cells for the link field LINK, and re are m cells for the pointer array LIST. Suppose each record requires only 1 word for its rmation field. Then it may be more useful to use open addressing with a table with $3m$ ations, which has the load factor $\lambda \leq 1/3$, than to use chaining to resolve collisions.

SOLVED PROBLEMS

Q1 Define a hash table. Explain briefly.

A hash table is a data structure used for the storage of records. It provides a means of rapid searching for a record with a given key value and adapts well to the insertion and deletion of records.

Q2 Explain sort stability briefly.

Sort stability is an attribute of a sort, indicating that data with equal keys maintain their relative input order in the output.

SUPPLEMENTARY PROBLEMS

Sorting

9.1 Write a subprogram RANDOM(DATA, N, K) which assigns N random integers between 1 and K to the array DATA.

9.2 Translate insertion sort into a subprogram INSERTSORT(A, N) which sorts the array A with N elements. Test the program using:
(a) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
(b) D, A, T, A, S, T, R, U, C, T, U, R, E, S

9.3 Translate insertion sort into a subprogram INSERTCOUNT(A, N, NUMB) which sorts the array A with N elements and which also counts the number NUMB of comparisons.

9.4 Write a program TESTINSERT(N, AVE) which repeats 500 times the procedure INSERTCOUNT(A, N, NUMB) and which finds the average AVE of the 500 values of NUMB. (Theoretically, AVE $\approx N^2/4$.) Use RANDOM(A, N, 5*N) from Problem 9.1 as each input. Test the program using N = 100 (so, theoretically, AVE $\approx N^2/4 = 2500$).

9.5 Translate quicksort into a subprogram QUICKCOUNT(A, N, NUMB) which sorts the array A with N elements and which also counts the number NUMB of comparisons. (See Sec. 6.5.)

9.6 Write a program TESTQUICKSORT(N, AVE) which repeats QUICKCOUNT(A, N, NUMB) 500 times and which finds the average AVE of the 500 values of NUMB. (Theoretically, AVE $\approx N \log_2 N$.) Use RANDOM(A, N, 5*N) from Problem 9.1 as each input. Test the program using N = 100 (so, theoretically, AVE ≈ 700).

9.7 Translate Procedure 9.2 into a subprogram MIN(A, LB, UB, LOC) which finds the location LOC of the smallest elements among A[LB], A[LB + 1], ..., A[UB].

9.8 Translate selection sort into a subprogram SELECTSORT(A, N) which sorts the array with N elements. Test the program using:
(a) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
(b) D, A, T, A, S, T, R, U, C, T, U, R, E, S

Searching, Hashing

9.9 Suppose an unsorted linked list is in memory. Write a procedure

SEARCH(INFO, LINK, START, ITEM, LOC)

which (a) finds the location LOC of ITEM in the list or sets LOC := NULL for an unsuccessful search and (b) when the search is successful, interchanges ITEM with the element in front of it. (Such a list is said to be *self-organizing*. It has the property that elements which are frequently accessed tend to move to the beginning of the list.)

9.10 Consider the following 4-digit employee numbers (see Example 9.10):

9614, 5882, 6713, 4409, 1825

Find the 2-digit hash address of each number using (a) the division method, with $m = 97$; (b) the midsquare method; (c) the folding method without reversing; and (d) the folding method with reversing.