(a) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(b) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

(c) In the third and final pass, the hundreds digits are sorted into pockets.

When the cards are collected after the third pass, the numbers are in the following order:

128, 143, 321, 348, 361, 366, 423, 538, 543

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$$C \leq 9*3*10$$

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix $d = 10$ digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., $d = 10$ for decimal digits, $d = 26$ for letters and $d = 2$ for bits), and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1}d_{i2} \ldots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence the number $C(n)$ of comparisons for the algorithm is bounded as follows:

$$C(n) \leq d*s*n$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$, so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. In other words, radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need $d*n$ memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require $2*n$ memory locations.

9.8 SEARCHING AND DATA MODIFICATION

Suppose S is a collection of data maintained in memory by a table using some type of data structure. Searching is the operation which finds the location LOC in memory of some given ITEM of information or sends some message that ITEM does not belong to S. The search is said to be

successful or unsuccessful according to whether ITEM does or does not belong to S. The searching algorithm that is used depends mainly on the type of data structure that is used to maintain S in memory.

Data modification refers to the operations of inserting, deleting and updating. Here data modification will mainly refer to inserting and deleting. These operations are closely related to searching, since usually one must search for the location of the ITEM to be deleted or one must search for the proper place to insert ITEM in the table. The insertion or deletion also requires a certain amount of execution time, which also depends mainly on the type of data structure that is used.

Generally speaking, there is a tradeoff between data structures with fast searching algorithms and data structures with fast modification algorithms. This situation is illustrated below, where we summarize the searching and data modification of three of the data structures previously studied in the text.

(1) *Sorted array.* Here one can use a binary search to find the location LOC of a given ITEM in time $O(\log n)$. On the other hand, inserting and deleting are very slow, since, on the average, $n/2 = O(n)$ elements must be moved for a given insertion or deletion. Thus a sorted array would likely be used when there is a great deal of searching but only very little data modification.

(2) *Linked list.* Here one can only perform a linear search to find the location LOC of a given ITEM, and the search may be very, very slow, possibly requiring time $O(n)$. On the other hand, inserting and deleting requires only a few pointers to be changed. Thus a linked list would be used when there is a great deal of data modification, as in word (string) processing.

(3) *Binary search tree.* This data structure combines the advantages of the sorted array and the linked list. That is, searching is reduced to searching only a certain path P in the tree T, which, on the average, requires only $O(\log n)$ comparisons. Furthermore, the tree T is maintained in memory by a linked representation, so only certain pointers need be changed after the location of the insertion or deletion is found. The main drawback of the binary search tree is that the tree may be very unbalanced, so that the length of a path P may be $O(n)$ rather than $O(\log n)$. This will reduce the searching to approximately a linear search.

Remark: The above worst-case scenario of a binary search tree may be eliminated by using a height-balanced binary search tree that is rebalanced after each insertion or deletion. The algorithms for such rebalancing are rather complicated and lie beyond the scope of this text.

Searching Files, Searching Pointers

Suppose a file F of records $R_1, R_2, \ldots, R_N$ is stored in memory. Searching F usually refers to finding the location LOC in memory of the record with a given key value relative to a primary key field K. One way to simplify the searching is to use an auxiliary sorted array of pointers, as discussed in Sec. 9.2. Then a binary search can be used to quickly find the location LOC of the record with the given key. In the case where there is a great deal of inserting and deleting of records in the file, one might want to use an auxiliary binary search tree rather than an auxiliary sorted array. In any case, the searching of the file F is reduced to the searching of a collection S of items, as discussed above.

9.9 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 9.9

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 9.10

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:

$$3205, \quad 7148, \quad 2345$$

(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 obtain:

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

(b) *Midsquare 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

Observe that the fourth and fifth digits, counting from the right, are chosen for the hash address.

(c) *Folding method.* Chopping the key k into two parts and adding yields the following hash addresses:

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

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:

$$H(3205) = 32 + 50 = 82, \quad H(7148) = 71 + 84 = 155, \quad H(2345) = 23 + 54 = 77$$

Collision Resolution

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

The efficiency of a hash function with a collision resolution procedure is measured by the average 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:

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

These quantities will be discussed for our collision procedures.

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.)

Example 9.11

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.

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 9.12

Consider again the data in Example 9.11, 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. 9.7. 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.

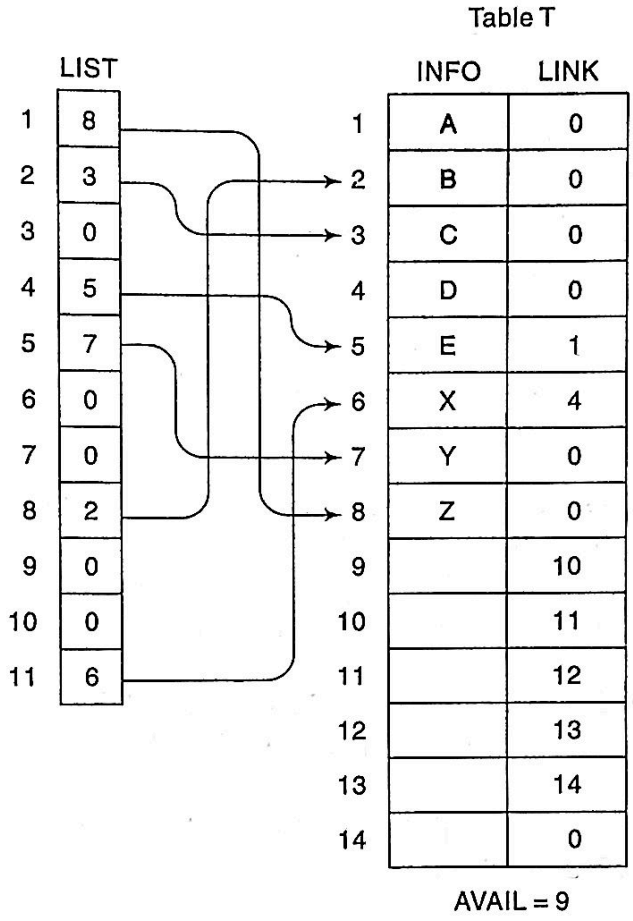

Fig. 9.7

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

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