Chapter Nine

Sorting and Searching

9.1 INTRODUCTION

Sorting and searching are fundamental operations in computer science. *Sorting* refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data. *Searching* refers to the operation of finding the location of a given item in a collection of items.

There are many sorting and searching algorithms. Some of them, such as heapsort and binary search, have already been discussed throughout the text. The particular algorithm one chooses depends on the properties of the data and the operations one may perform on the data. Accordingly, we will want to know the complexity of each algorithm; that is, we will want to know the running time $f(n)$ of each algorithm as a function of the number n of input items. Sometimes, we will also discuss the space requirements of our algorithms.

Sorting and searching frequently apply to a file of records, so we recall some standard terminology. Each record in a file F can contain many fields, but there may be one particular field whose values uniquely determine the records in the file. Such a field K is called a *primary key*, and the values $k_1, k_2, \ldots$ in such a field are called *keys* or *key values*. Sorting the file F usually refers to sorting F with respect to a particular primary key, and searching in F refers to searching for the record with a given key value.

This chapter will first investigate sorting algorithms and then investigate searching algorithms. Some texts treat searching before sorting.

9.2 SORTING

Let A be a list of n elements $A_1, A_2, \ldots, A_n$ in memory. *Sorting* A refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is, so that

$$A_1 \leq A_2 \leq A_3 \leq \cdots \leq A_n$$

Since A has n elements, there are $n!$ ways that the contents can appear in A. These ways correspond precisely to the $n!$ permutations of 1, 2,..., n. Accordingly, each sorting algorithm must take care of these $n!$ possibilities.

Example 9.1

Suppose an array DATA contains 8 elements as follows:

DATA: 77, 33, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 33, 44, 55, 66, 77, 88

Since DATA consists of 8 elements, there are 8! = 40 320 ways that the numbers 11, 22,..., 88 can appear in DATA.

Complexity of Sorting Algorithms

The complexity of a sorting algorithm measures the running time as a function of the number n of items to be sorted. We note that each sorting algorithm S will be made up of the following operations, where $A_1, A_2, \ldots, A_n$ contain the items to be sorted and B is an auxiliary location:

(a) Comparisons, which test whether $A_i < A_j$ or test whether $A_i < B$
(b) Interchanges, which switch the contents of A_i and A_j or of A_i and B
(c) Assignments, which set $B := A_i$ and then set $A_j := B$ or $A_j := A_i$

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

There are two main cases whose complexity we will consider; the worst case and the average case. In studying the average case, we make the probabilistic assumption that all the $n!$ permutations of the given n items are equally likely. (The reader is referred to Sec. 2.5 for a more detailed discussion of complexity.)

Previously, we have studied the bubble sort (Sec. 4.6), quicksort (Sec. 6.6) and heapsort (Sec. 7.10). The approximate number of comparisons and the order of complexity of these algorithms are summarized in the following table:

Algorithm	Worst Case	Average Case
Bubble Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$
Quicksort	$\frac{n(n+3)}{2} = O(n^2)$	$1.4n \log n = O(n \log n)$
Heapsort	$3n \log n = O(n \log n)$	$3n \log n = O(n \log n)$

Note first that the bubble sort is a very slow way of sorting; its main advantage is the simplicity of the algorithm. Observe that the average-case complexity ($n \log n$) of heapsort is the same as that of quicksort, but its worst-case complexity ($n \log n$) seems quicker than quicksort (n^2). However, empirical evidence seems to indicate that quicksort is superior to heapsort except on rare occasions.

Lower Bounds

The reader may ask whether there is an algorithm which can sort n items in time of order less than $O(n \log n)$. The answer is no. The reason is indicated below.

Suppose S is an algorithm which sorts n items $a_1, a_2, \ldots, a_n$. We assume there is a *decision tree* T corresponding to the algorithm S such that T is an extended binary search tree where the external nodes correspond to the $n!$ ways that n items can appear in memory and where the internal nodes correspond to the different comparisons that may take place during the execution of the algorithm S. Then the number of comparisons in the worst case for the algorithm S is equal to the length of the longest path in the decision tree T or, in other words, the depth D of the tree, T. Moreover, the average number of comparisons for the algorithm S is equal to the average external path length $\bar{E}$ of the tree T.

Figure 9.1 shows a decision tree T for sorting $n = 3$ items. Observe that T has $n! = 3! = 6$ external nodes. The values of D and $\bar{E}$ for the tree follow:

$$D = 3 \quad \text{and} \quad \bar{E} = \frac{1}{6}(2 + 3 + 3 + 3 + 3 + 2) = 2.667$$

Consequently, the corresponding algorithm S requires at most (worst case) $D = 3$ comparisons and, on the average, $\bar{E} = 2.667$ comparisons to sort the $n = 3$ items.

Accordingly, studying the worst-case and average-case complexity of a sorting algorithm S is reduced to studying the values of D and $\bar{E}$ in the corresponding decision tree T. First, however, we recall some facts about extended binary trees (Sec. 7.11). Suppose T is an extended binary tree with N external nodes, depth D and external path length $E(T)$. Any such tree cannot have more than 2^D external nodes, and so

$$2^D \geq N \quad \text{or equivalently} \quad D \geq \log N$$

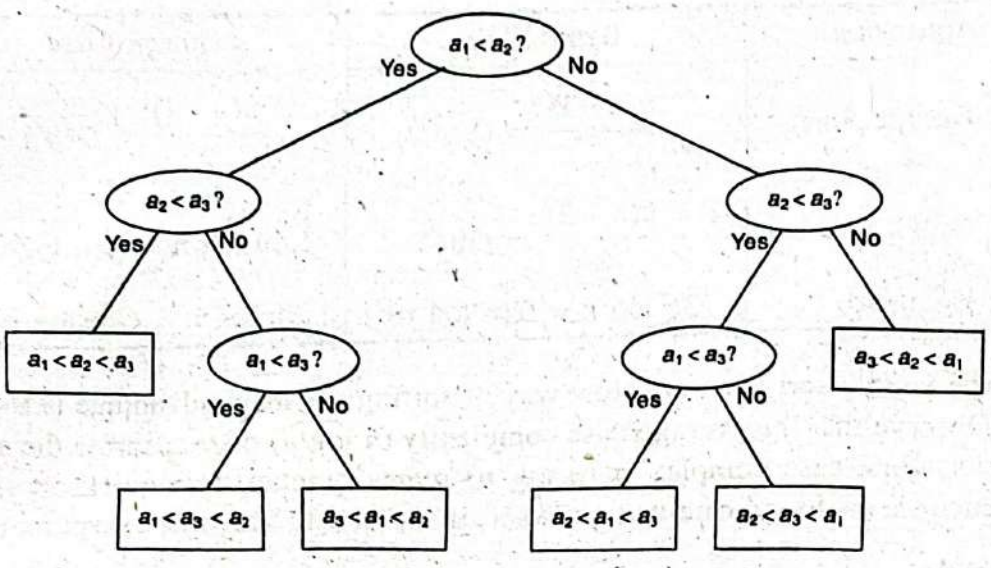

Fig. 9.1 *Decision Tree T for Sorting n = 3 items*

Furthermore, T will have a minimum external path length $E(L)$ among all such trees with N nodes when T is a complete tree. In such a case,

$$E(L) = N \log N + O(N) \geq N \log N$$

The $N \log N$ comes from the fact that there are N paths with length $\log N$ or $\log N + 1$. and the $O(N)$ comes from the fact that there are at most N nodes on the deepest level. Dividing $E(L)$ by the number N of external paths gives the average external path length $\bar{E}$. Thus, for any extended binary tree T with N external nodes,

$$\bar{E} = \frac{E(L)}{N} \geq \frac{N \log N}{N} = \log N.$$

Now suppose T is the decision tree corresponding to a sorting algorithm S which sorts n items. Then T has $n!$ external nodes. Substituting $n!$ for N in the above formulas yields

$$D \geq \log n! \approx n \log n \quad \text{and} \quad \bar{E} \geq \log n! \approx n \log n$$

The condition $\log n! \approx n \log n$ comes from Stirling's formula, that

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \cdots\right)$$

Thus $n \log n$ is a lower bound for both the worst case and the average case. In other words, *$O(n \log n)$ is the best possible for any sorting algorithm which sorts n items.*

Sorting Files; Sorting Pointers

Suppose a file F of records $R_1, R_2, \ldots, R_n$ is stored in memory. "Sorting F" refers to sorting F with respect to some field K with corresponding values $k_1, k_2, \ldots, k_n$. That is, the records are ordered so that

$$k_1 \leq k_2 \leq \cdots \leq k_n$$

The field K is called the *sort key*. (Recall that K is called a *primary key* if its values uniquely determine the records in F.) Sorting the file with respect to another key will order the records in another way.

Example 9.2

Suppose the personnel file of a company contains the following data on each of its employees:

Name Social Security Number Sex Monthly Salary

Sorting the file with respect to the Name key will yield a different order of the records than sorting the file with respect to the Social Security Number key. The company may want to sort the file according to the Salary field even though the field may not uniquely determine the employees. Sorting the file with respect to the Sex key will likely be useless; it simply separates the employees into two subfiles, one with the male employees and one with the female employees.

Sorting a file F by reordering the records in memory may be very expensive when the records are very long. Moreover, the records may be in secondary memory, where it is even more time-consuming to move records into different locations. Accordingly, one may prefer to form an auxiliary array POINT containing pointers to the records in memory and then sort the array POINT with respect to a field KEY rather than sorting the records themselves. That is, we sort POINT so that

$$\text{KEY[POINT[1]]} \le \text{KEY[POINT[2]]} \le \cdots \le \text{KEY[POINT[N]]}$$

Note that choosing a different field KEY will yield a different order of the array POINT.

Example 9.3

Figure 9.2(a) shows a personnel file of a company in memory. Figure 9.2(b) shows three arrays; POINT, PTRNAME and PTRSSN. The array POINT contains the locations of the records in memory, PTRNAME shows the pointers sorted according to the NAME field, that is,

$$\text{NAME[PTRNAME[1]]} < \text{NAME[PTRNAME[2]]} < \cdots < \text{NAME[PTRNAME[9]]}$$

and PTRSSN shows the pointers sorted according to the SSN field, that is,

$$\text{SSN[PTRSSN[1]]} < \text{SSN[PTRSSN[2]]} < \cdots < \text{SSN[PTRSSN[9]]}$$

Given the name (EMP) of an employee, one can easily find the location of NAME in memory using the array PTRNAME and the binary search algorithm. Similarly, given the social security number NUMB of an employee, one can easily find the location of the employee's record in memory by using the array PTRSSN and the binary search algorithm. Observe, also, that it is not even necessary for the records to appear in successive memory locations. Thus inserting and deleting records can easily be done.

28-A

	NAME	SSN	SEX	SALARY
1				
2	Davis	192-38-7282	Female	22 800
3	Kelly	165-64-3351	Male	19 000
4	Green	175-56-2251	Male	27 200
5				
6	Brown	178-52-1065	Female	14 700
7	Lewis	181-58-9939	Female	16 400
8				
9	Cohen	177-44-4557	Male	19 000
10	Rubin	135-46-6262	Female	15 500
11				
12	Evans	168-56-8113	Male	34 200
13				
14	Harris	208-56-1654	Female	22 800

(a)

	POINT	PTRNAME	PTRSSN
1	2	6	10
2	3	9	8
3	4	2	12
4	6	12	4
5	7	4	9
6	9	14	6
7	10	3	7
8	12	7	2
9	14	10	14

(b)

Fig. 9.2

9.3 INSERTION SORT

Suppose an array A with n elements A[1], A[2], ..., A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2], ..., A[K – 1]. That is:

Pass 1. A[1] by itself is trivially sorted.

Pass 2. A[2] is inserted either before or after A[1] so that: A[1], A[2] is sorted.

Pass 3. A[3] is inserted into its proper place in A[1], A[2], that is, before A[1], between A[1] and A[2], or after A[2], so that: A[1], A[2], A[3] is sorted.

Pass 4. A[4] is inserted into its proper place in A[1], A[2], A[3] so that:

A[1], A[2], A[3], A[4] is sorted.

.....

Pass N. A[N] is inserted into its proper place in A[1], A[2], ..., A[N – 1] so that:

A[1], A[2], ..., A[N] is sorted.

This sorting algorithm is frequently used when n is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

There remains only the problem of deciding how to insert A[K] in its proper place in the sorted subarray A[1], A[2], ..., A[K – 1]. This can be accomplished by comparing A[K] with A[K – 1], comparing A[K] with A[K – 2], comparing A[K] with A[K – 3], and so on, until first meeting an

28-B

element A[J] such that A[J] ≤ A[K]. Then each of the elements A[K – 1], A[K – 2], ..., A[J + 1] is moved forward one location, and A[K] is then inserted in the J + 1st position in the array.

The algorithm is simplified if there always is an element A[J] such that A[J] ≤ A[K]; otherwise we must constantly check to see if we are comparing A[K] with A[1]. This condition can be accomplished by introducing a sentinel element A[0] = –∞ (or a very small number).

Example 9.4

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Figure 9.3 illustrates the insertion sort algorithm. The circled element indicates the A[K] in each pass of the algorithm, and the arrow indicates the proper place for inserting A[K].

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1:	–∞	(77)	33	44	11	88	22	66	55
K = 2:	–∞	77	(33)	44	11	88	22	66	55
K = 3:	–∞	33	77	(44)	11	88	22	66	55
K = 4:	–∞	33	44	77	(11)	88	22	66	55
K = 5:	–∞	11	33	44	77	(88)	22	66	55
K = 6:	–∞	11	33	44	77	88	(22)	66	55
K = 7:	–∞	11	22	33	44	77	88	(66)	55
K = 8:	–∞	11	22	33	44	66	77	88	(55)
Sorted:	–∞	11	22	33	44	55	66	77	88

Fig. 9.3 *Insertion Sort for n = 8 Items*

The formal statement of our insertion sort algorithm follows.

Algorithm 9.1: (Insertion Sort) INSERTION(A, N).
This algorithm sorts the array A with N elements.

1. Set A[0] := –∞. [Initializes sentinel element.]
2. Repeat Steps 3 to 5 for K = 2, 3, ..., N:
3. Set TEMP := A[K] and PTR := K – 1.
4. Repeat while TEMP < A[PTR]:
   (a) Set A[PTR + 1] := A[PTR]. [Moves element forward.]
   (b) Set PTR := PTR – 1.
   [End of loop.]
5. Set A[PTR + 1] := TEMP. [Inserts element in proper place.]
   [End of Step 2 loop.]
6. Return.

Observe that there is an inner loop which is essentially controlled by the variable PTR, and there is an outer loop which uses K as an index.

Complexity of Insertion Sort

The number $f(n)$ of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K – 1 of comparisons. Hence

$$f(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Furthermore, one can show that, on the average, there will be approximately (K – 1)/2 comparisons in the inner loop. Accordingly, for the average case,

$$f(n) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4} = O(n^2)$$

Thus the insertion sort algorithm is a very slow algorithm when n is very large.

The above results are summarized in the following table:

Algorithm	*Worst Case*	*Average Case*
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

Remark: Time may be saved by performing a binary search, rather than a linear search, to find the location in which to insert A[K] in the subarray A[1], A[2], ..., A[K – 1]. This requires, on the average, log K comparisons rather than (K – 1)/2 comparisons. However, one still needs to move (K – 1)/2 elements forward. Thus the order of complexity is not changed. Furthermore, insertion sort is usually used only when n is small, and in such a case, the linear search is about as efficient as the binary search.

9.4 SELECTION SORT

Suppose an array A with n elements A[1], A[2], ..., A[N] is in memory. The selection sort algorithm for sorting A works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on. More precisely:

Pass 1. Find the location LOC of the smallest in the list of N elements
A[1], A[2], ..., A[N], and then interchange A[LOC] and A[1]. Then: A[1] is sorted.

Pass 2. Find the location LOC of the smallest in the sublist of N – 1 elements
A[2], A[3], ..., A[N], and then interchange A[LOC] and A[2]. Then:
A[1], A[2] is sorted, since A[1] ≤ A[2].

Pass 3. Find the location LOC of the smallest in the sublist of N – 2 elements A[3], A[4], ..., A[N], and then interchange A[LOC] and A[3]. Then:
A[1], A[2], ..., A[3] is sorted, since A[2] ≤ A[3].

... .

... .

Pass N – 1. Find the location LOC of the smaller of the elements A[N – 1], A[N], and then interchange A[LOC] and A[N – 1]. Then:
A[1], A[2], ..., A[N] is sorted, since A[N – 1] ≤ A[N].

Thus A is sorted after N – 1 passes.

Example 9.5

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Applying the selection sort algorithm to A yields the data in Fig. 9.4. Observe that LOC gives the location of the smallest among A[K], A[K + 1], ..., A[N] during Pass K. The circled elements indicate the elements which are to be interchanged.

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1, LOC = 4	(77)	33	44	(11)	88	22	66	55
K = 2, LOC = 6	11	(33)	44	77	88	(22)	66	55
K = 3, LOC = 6	11	22	(44)	77	88	(33)	66	55
K = 4, LOC = 6	11	22	33	(77)	88	(44)	66	55
K = 5, LOC = 8	11	22	33	44	(88)	77	66	(55)
K = 6, LOC = 7	11	22	33	44	55	(77)	(66)	88
K = 7, LOC = 7	11	22	33	44	55	66	(77)	88
Sorted:	11	22	33	44	55	66	77	88

Fig. 9.4 *Selection Sort for n = 8 Items*

There remains only the problem of finding, during the Kth pass, the location LOC of the smallest among the elements A[K], A[K + 1], ..., A[N]. This may be accomplished by using a variable MIN to hold the current smallest value while scanning the subarray from A[K] to A[N]. Specifically, first set MIN := A[K] and LOC := K, and then traverse the list, comparing MIN with each other element A[J] as follows:

(a) If MIN ≤ A[J], then simply move to the next element.
(b) If MIN > A[J], then update MIN and LOC by setting MIN := A[J] and LOC := J.

and assign the smaller element to C[PTR]. Then we increment PTR by setting PTR := PTR + 1, and we either increment NA by setting NA := NA + 1 or increment NB by setting NB := NB + 1, according to whether the new element in C has come from A or from B. Furthermore, if NA > r, then the remaining elements of B are assigned to C; or if NB > s, then the remaining elements of A are assigned to C.

The formal statement of the algorithm follows.

Algorithm 9.4: MERGING(A, R, B, S, C)

Let A and B be sorted arrays with R and S elements, respectively. This algorithm merges A and B into an array C with N = R + S elements.

1. [Initialize.] Set NA := 1, NB := 1 and PTR := 1.
2. [Compare.] Repeat while NA ≤ R and NB ≤ S:
   If A[NA] < B[NB], then:
   (a) [Assign element from A to C.] Set C[PTR] := A[NA].
   (b) [Update pointers.] Set PTR := PTR + 1 and NA := NA + 1.
   Else:
   (a) [Assign element from B to C.] Set C[PTR] := B[NB].
   (b) [Update pointers.] Set PTR := PTR + 1 and NB := NB + 1.
   [End of If structure.]
   [End of loop.]
3. [Assign remaining elements to C.]
   If NA > R, then:
   Repeat for K = 0, 1, 2, ..., S – NB:
   Set C[PTR + K] := B[NB + K].
   [End of loop.]
   Else:
   Repeat for K = 0, 1, 2, ..., R – NA:
   Set C[PTR + K] := A[NA + K].
   [End of loop.]
   [End of If structure.]
4. Exit.

Complexity of the Merging Algorithm

The input consists of the total number $n = r + s$ of elements in A and B. Each comparison assigns an element to the array C, which eventually has n elements. Accordingly, the number $f(n)$ of comparisons cannot exceed n:

$$f(n) \leq n = O(n)$$

In other words, the merging algorithm can be run in linear time.

Nonregular Matrices

Suppose A, B and C are matrices, but not necessarily regular matrices. Assume A is sorted, with r elements and lower bound LBA; B is sorted, with s elements and lower bound LBB; and C has lower bound LBC. Then UBA = LBA + r − 1 and UBB = LBB + s − 1 are, respectively, the upper bounds of A and B. Merging A and B now may be accomplished by modifying the above algorithm as follows.

Procedure 9.5: MERGE(A, R, LBA, S, LBB, C, LBC)
This procedure merges the sorted arrays A and B into the array C.

1. Set NA := LBA, NB := LBB, PTR := LBC, UBA := LBA + R − 1, UBB := LBB + S − 1.
2. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
3. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
4. Return.

Observe that this procedure is called MERGE, whereas Algorithm 9.4 is called MERGING. The reason for stating this special case is that this procedure will be used in the next section, on merge-sort.

Binary Search and Insertion Algorithm

Suppose the number r of elements in a sorted array A is much smaller than the number s of elements in a sorted array B. One can merge A with B as follows. For each element A[K] of A, use a binary search on B to find the proper location to insert A[K] into B. Each such search requires at most log s comparisons; hence this binary search and insertion algorithm to merge A and B requires at most r log s comparisons. We emphasize that this algorithm is more efficient than the usual merging Algorithm 9.4 only when $r << s$, that is, when r is much less than s.

Example 9.6

Suppose A has 5 elements and suppose B has 100 elements. Then merging A and B by Algorithm 9.4 uses approximately 100 comparisons. On the other hand, only approximately log 100 = 7 comparisons are needed to find the proper place to insert an element of A into B using a binary search. Hence only approximately $5 \cdot 7 = 35$ comparisons are needed to merge A and B using the binary search and insertion algorithm.

The binary search and insertion algorithm does not take into account the fact that A is sorted. Accordingly, the algorithm may be improved in two ways as follows. (Here we assume that A has 5 elements and B has 100 elements.)

(1) *Reducing the target set.* Suppose after the first search we find that A[1] is to be inserted after B[16]. Then we need only use a binary search on B[17], ..., B[100] to find the proper location to insert A[2]. And so on.

(2) *Tabbing.* The expected location for inserting A[1] in B is near B[20] (that is, $B[s/r]$), not near B[50]. Hence we first use a linear search on B[20], B[40], B[60], B[80] and B[100] to find B[K] such that A[1] ≤ B[K], and then we use a binary search on B[K – 20], B[K – 19], ..., B[K]. (This is analogous to using the tabs in a dictionary which indicate the location of all words with the same first letter.)

The details of the revised algorithm are left to the reader.

9.6 MERGE-SORT

Suppose an array A with n elements A[1], A[2], ..., A[N] is in memory. The merge-sort algorithm which sorts A will first be described by means of a specific example.

Example 9.7

Suppose the array A contains 14 elements as follows:

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

33, 66 22, 40 55, 88 11, 60 20, 80 44, 50 30, 70

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

22, 33, 40, 66 11, 55, 60, 88 20, 44, 50, 80 30, 77

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

11, 22, 33, 40, 55, 60, 66, 88 20, 30, 44, 50, 77, 80

Pass 4. Merge the two sorted subarrays to obtain the single sorted array

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

The original array A is now sorted.

The above merge-sort algorithm for sorting an array A has the following important property. After Pass K, the array A will be partitioned into sorted subarrays where each subarray, except possibly the last, will contain exactly $L = 2^K$ elements. Hence the algorithm requires at most $\log n$ passes to sort an n-element array A.

The above informal description of merge-sort will now be translated into a formal algorithm which will be divided into two parts. The first part will be a procedure MERGEPASS, which uses Procedure 9.5 to execute a single pass of the algorithm; and the second part will repeatedly apply MERGEPASS until A is sorted.

The MERGEPASS procedure applies to an n-element array A which consists of a sequence of sorted subarrays. Moreover, each subarray consists of L elements except that the last subarray may have fewer than L elements. Dividing n by 2*L, we obtain the quotient Q, which tells the number of pairs of L-element sorted subarrays; that is,

$$Q = INT(N/(2*L))$$

(We use INT(X) to denote the integer value of X.) Setting S = 2*L*Q, we get the total number S of elements in the Q pairs of subarrays. Hence R = N – S denotes the number of remaining elements. The procedure first merges the initial Q pairs of L-element subarrays. Then the procedure takes care of the case where there is an odd number of subarrays (when R ≤ L) or where the last subarray has fewer than L elements.

The formal statement of MERGEPASS and the merge-sort algorithm follow:

Procedure 9.6: MERGEPASS(A, N, L, B)

The N-element array A is composed of sorted subarrays where each subarray has L elements except possibly the last subarray, which may have fewer than L elements. The procedure merges the pairs of subarrays of A and assigns them to the array B.

1. Set Q := INT(N/(2*L)), S := 2*L*Q and R := N – S.
2. [Use Procedure 9.5 to merge the Q pairs of subarrays.]
   Repeat for J = 1, 2, ..., Q:
   (a) Set LB := 1 + (2*J – 2)*L. [Finds lower bound of first array.]
   (b) Call MERGE(A, L, LB, A, L, LB + L, B, LB).
   [End of loop.]
3. [Only one subarray left?]
   If R ≤ L, then:
   Repeat for J = 1, 2, ..., R:
   Set B(S + J) := A(S + J).
   [End of loop.]
   Else:
   Call MERGE(A, L, S + 1, A, R, L + S + 1, B, S + 1).
   [End of If structure.]
4. Return.

Algorithm 9.7: MERGESORT(A, N)

This algorithm sorts the N-element array A using an auxiliary array B.

1. Set L := 1. [Initializes the number of elements in the subarrays.]
2. Repeat Steps 3 to 6 while L < N:
3. Call MERGEPASS(A, N, L, B).
4. Call MERGEPASS(B, N, 2 * L, A).
5. Set L := 4 * L.
   [End of Step 2 loop.]
6. Exit.

Since we want the sorted array to finally appear in the original array A, we must execute the procedure MERGEPASS an even number of times.

Complexity of the Merge-Sort Algorithm

Let $f(n)$ denote the number of comparisons needed to sort an n-element array A using the merge-sort algorithm. Recall that the algorithm requires at most $\log n$ passes. Moreover, each pass merges a total of n elements, and by the discussion on the complexity of merging, each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

$$f(n) \leq n \log n$$

Observe that this algorithm has the same order as heapsort and the same average order as quicksort. The main drawback of merge-sort is that it requires an auxiliary array with n elements. Each of the other sorting algorithms we have studied requires only a finite number of extra locations, which is independent of n.

The above results are summarized in the following table:

Algorithm	*Worst Case*	*Average Case*	*Extra Memory*
Merge-Sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

9.7 RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

Example 9.8

Suppose 9 cards are punched as follows:

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

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 9.6:

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass

Fig. 9.6

(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, 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 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:

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

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:

$S(\lambda)$ = 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.

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:

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

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.

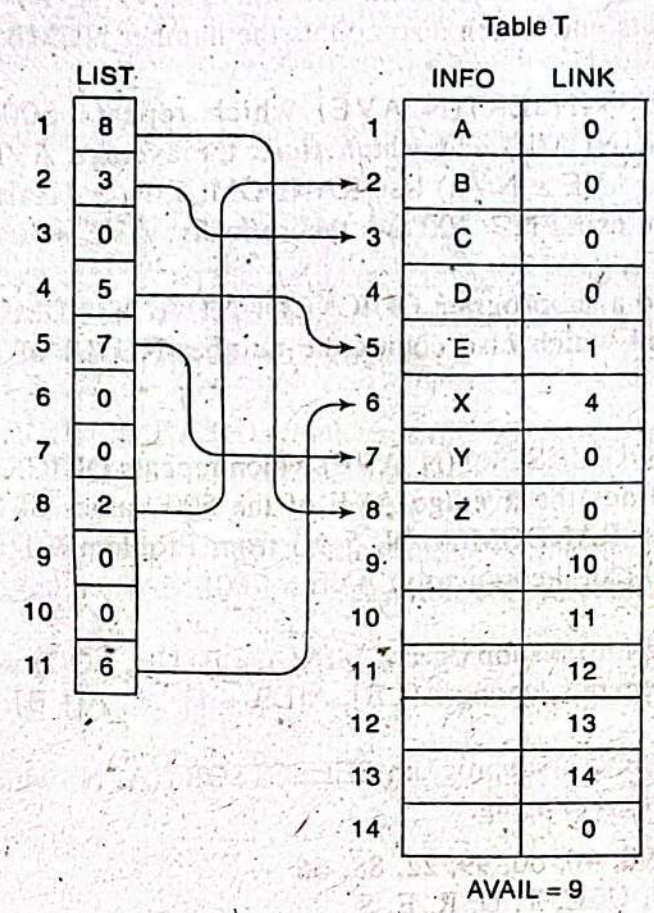

Fig. 9.7

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

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.

9.11 Consider the data in Example 9.11. Suppose the 8 records are entered into the table T in the reverse order Z, Y, X, E, D, C, B, A. (a) Show how the file F appears in memory. (b) Find

the average number S of probes for a successful search and the average number U of probes for an unsuccessful search. (Compare with the corresponding results in Example 9.11.)

9.12 Consider the data in Example 9.12 and Fig. 9.7. Suppose the following additional records are added to the file:

(P, 2), (Q, 7), (R, 4), (S, 9)

(Here the left entry is the record and the right entry is the hash address.) (a) Find the updated tables T and LIST. (b) Find the average number S of probes for a successful search and the average number U of probes for an unsuccesful search.

9.13 Write a subprogram MID(KEY, HASH) which uses the midsquare method to find the 2-digit hash address HASH of a 4-digit employee number key.

9.14 Write a subprogram FOLD(KEY, HASH) which uses the folding method with reversing to find the 2-digit hash address HASH of a 4-digit employee number key.