Chapter 1

Introduction and Overview

1.1 INTRODUCTION

This chapter introduces the subject of data structures and presents an overview of the content of the text. Basic terminology and concepts will be defined and relevant examples provided. An overview of data organization and certain data structures will be covered along with a discussion of the different operations which are applied to these data structures. Last, we will introduce the notion of an algorithm and its complexity, and we will discuss the time-space tradeoff that may occur in choosing a particular algorithm and data structure for a given problem.

1.2 BASIC TERMINOLOGY; ELEMENTARY DATA ORGANIZATION

Data are simply values or sets of values. A *data item* refers to a single unit of values. Data items that are divided into subitems are called *group items*; those that are not are called *elementary items*. For example, an employee's name may be divided into three subitems—first name, middle initial and last name—but the social security number would normally be treated as a single item.

Collections of data are frequently organized into a hierarchy of *fields*, *records* and *files*. In order to make these terms more precise, we introduce some additional terminology.

An *entity* is something that has certain *attributes* or properties which may be assigned values. The values themselves may be either numeric or nonnumeric. For example, the following are possible attributes and their corresponding values for an entity, an employee of a given organization:

Attributes:	Name	Age	Sex	Social Security Number
Values:	JOHN BROWN	34	M	134-24-5533

Entities with similar attributes (e.g., all the employees in an organization) form an *entity set*. Each attribute of an entity set has a *range* of values, the set of all possible values that could be assigned to the particular attribute.

The term "information" is sometimes used for data with given attributes, or, in other words, meaningful or processed data.

The way that data are organized into the hierarchy of fields, records and files reflects the relationship between attributes, entities and entity sets. That is, a *field* is a single elementary unit of information representing an attribute of an entity, a *record* is the collection of field values of a given entity and a *file* is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items, but the value in a certain field may uniquely determine the record 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*.

EXAMPLE 1.1

(*a*) Suppose an automobile dealership maintains an inventory file where each record contains the following data:

Serial Number, Type, Year, Price, Accessories

The Serial Number field can serve as a primary key for the file, since each automobile has a unique serial number.

(*b*) Suppose an organization maintains a membership file where each record contains the following data:

Name, Address, Telephone Number, Dues Owed

Although there are four data items, Name and Address may be group items. Here the Name field is a

primary key. Note that the Address and Telephone Number fields may not serve as primary keys, since some members may belong to the same family and have the same address and telephone number.

Records may also be classified according to length. A file can have fixed-length records or variable-length records. In *fixed-length records*, all the records contain the same data items with the same amount of space assigned to each data item. In *variable-length records*, file records may contain different lengths. For example, student records usually have variable lengths, since different students take different numbers of courses. Usually, variable-length records have a minimum and a maximum length.

The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures. The study of such data structures, which forms the subject matter of this text, includes the following three steps:

(1) Logical or mathematical description of the structure

(2) Implementation of the structure on a computer

(3) Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure

The next section introduces us to some of these data structures.

Remark: The second and third of the steps in the study of data structures depend on whether the data are stored (*a*) in the main (primary) memory of the computer or (*b*) in a secondary (external) storage unit. This text will mainly cover the first case. This means that, given the address of a memory location, the time required to access the content of the memory cell does not depend on the particular cell or upon the previous cell accessed. The second case, called *file management* or *data base management*, is a subject unto itself and lies beyond the scope of this text.

1.3 DATA STRUCTURES

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a *data structure*. The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary. This section will introduce us to some of the data structures which will be discussed in detail later in the text.

Arrays

The simplest type of data structure is a *linear* (or *one-dimensional*) *array*. By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually $1, 2, 3, \ldots, n$. If we choose the name A for the array, then the elements of A are denoted by subscript notation

$$a_1, a_2, a_3, \ldots, a_n$$

or by the parenthesis notation

$$A(1), A(2), A(3), \ldots, A(N)$$

or by the bracket notation

$$A[1], A[2], A[3], \ldots, A[N]$$

Regardless of the notation, the number K in A[K] is called a *subscript* and A[K] is called a *subscripted variable*.

Remark: The parentheses notation and the bracket notation are frequently used when the array name consists of more than one letter or when the array name appears in an algorithm. When using this

notation we will use ordinary uppercase letters for the name and subscripts as indicated above by the A and N. Otherwise, we may use the usual subscript notation of italics for the name and subscripts and lowercase letters for the subscripts as indicated above by the a and n. The former notation follows the practice of computer-oriented texts whereas the latter notation follows the practice of mathematics in print.

EXAMPLE 1.2

A linear array STUDENT consisting of the names of six students is pictured in Fig. 1-1. Here STUDENT[1] denotes John Brown, STUDENT[2] denotes Sandra Gold, and so on.

STUDENT

1	John Brown
2	Sandra Gold
3	Tom Jones
4	June Kelly
5	Mary Reed
6	Alan Smith

Fig. 1-1

Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript. A *two-dimensional array* is a collection of similar data elements where each element is referenced by two subscripts. (Such arrays are called *matrices* in mathematics, and *tables* in business applications.) Multidimensional arrays are defined analogously. Arrays will be covered in detail in Chap. 4.

EXAMPLE 1.3

A chain of 28 stores, each store having 4 departments, may list its weekly sales (to the nearest dollar) as in Fig. 1-2. Such data can be stored in the computer using a two-dimensional array in which the first subscript denotes the store and the second subscript the department. If SALES is the name given to the array, then

SALES[1, 1] = 2872, SALES[1, 2] = 805, SALES[1, 3] = 3211, . . . , SALES[28, 4] = 982

The size of this array is denoted by 28×4 (read 28 by 4), since it contains 28 *rows* (the horizontal lines of numbers) and 4 *columns* (the vertical lines of numbers).

Store \ Dept.	1	2	3	4
1	2872	805	3211	1560
2	2196	1223	2525	1744
3	3257	1017	3686	1951
. . .	. . .	. . .	. . .	. . .
28	2618	931	2333	982

Fig. 1-2

Linked Lists

Linked lists will be introduced by means of an example. Suppose a brokerage firm maintains a file where each record contains a customer's name and his or her salesperson, and suppose the file contains the data appearing in Fig. 1-3. Clearly the file could be stored in the computer by such a table, i.e., by two columns of nine names. However, this may not be the most useful way to store the data, as the following discussion shows.

	Customer	Salesperson
1	Adams	Smith
2	Brown	Ray
3	Clark	Jones
4	Drew	Ray
5	Evans	Smith
6	Farmer	Jones
7	Geller	Ray
8	Hill	Smith
9	Infeld	Ray

Fig. 1-3

Another way of storing the data in Fig. 1-3 is to have a separate array for the salespeople and an entry (called a *pointer*) in the customer file which gives the location of each customer's salesperson. This is done in Fig. 1-4, where some of the pointers are pictured by an arrow from the location of the pointer to the location of the corresponding salesperson. Practically speaking, an integer used as a pointer requires less space than a name; hence this representation saves space, especially if there are hundreds of customers for each salesperson.

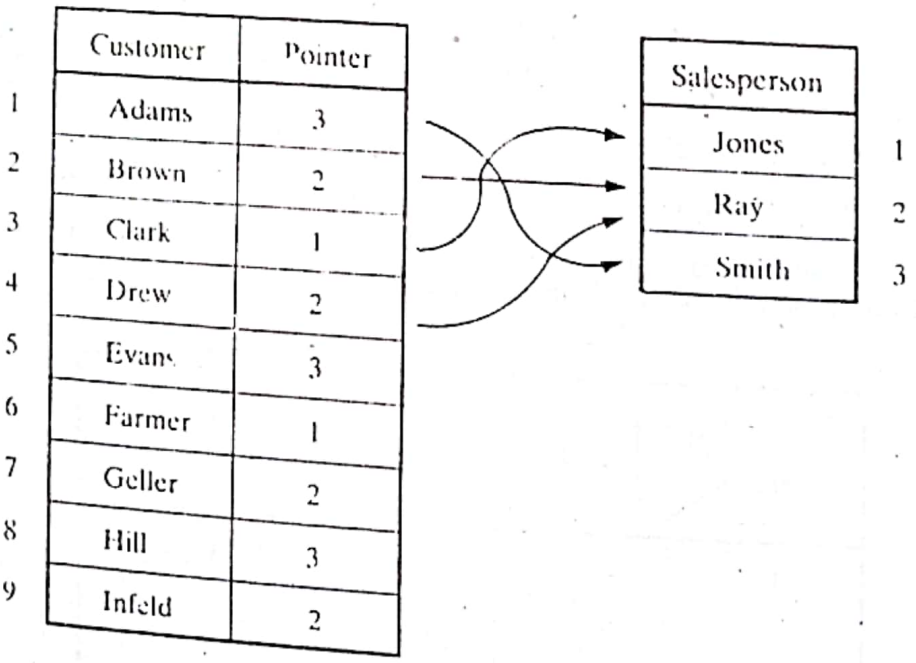


Fig. 1-4

Suppose the firm wants the list of customers for a given salesperson. Using the data representation in Fig. 1-4, the firm would have to search through the entire customer file. One way to simplify such a

search is to have the arrows in Fig. 1-4 point the other way; each salesperson would now have a set of pointers giving the positions of his or her customers, as in Fig. 1-5. The main disadvantage of this representation is that each salesperson may have many pointers and the set of pointers will change as customers are added and deleted.

	Salesperson	Pointer
1	Jones	3, 6
2	Ray	2, 4, 7, 9
3	Smith	1, 5, 8

Fig. 1-5

Another very popular way to store the type of data in Fig. 1-3 is shown in Fig. 1-6. Here each salesperson has one pointer which points to his or her first customer, whose pointer in turn points to the second customer, and so on, with the salesperson's last customer indicated by a 0. This is pictured with arrows in Fig. 1-6 for the salesperson Ray. Using this representation one can easily obtain the entire list of customers for a given salesperson and, as we will see in Chap. 5, one can easily insert and delete customers.

	Customer	Link
1	Adams	5
2	Brown	4
3	Clark	6
4	Drew	7
5	Evans	8
6	Farmer	0
7	Geller	9
8	Hill	0
9	Infeld	0

Salesperson	Pointer	
Jones	3	1
Ray	2	2
Smith	1	3

Fig. 1-6

The representation of the data in Fig. 1-6 is an example of linked lists. Although the terms "pointer" and "link" are usually used synonymously, we will try to use the term "pointer" when an element in one list points to an element in a different list, and to reserve the term "link" for the case when an element in a list points to an element in that same list.

Trees

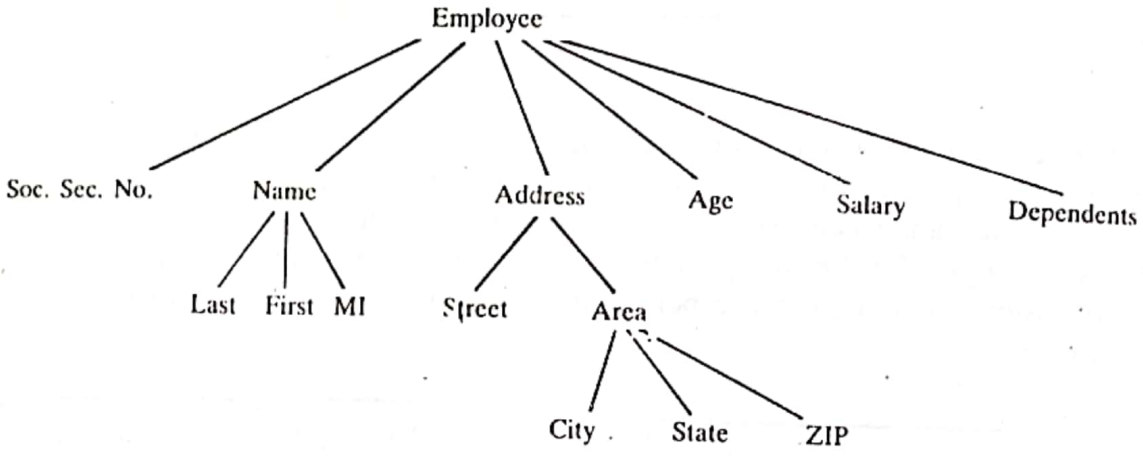
Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a *rooted tree graph* or, simply, a *tree*. Trees will be defined and discussed in detail in Chap. 7. Here we indicate some of their basic properties by means of two examples.

EXAMPLE 1.4 Record Structure

Although a file may be maintained by means of one or more arrays, a record, where one indicates both the group items and the elementary items, can best be described by means of a tree structure. For example, an employee personnel record may contain the following data items:

Social Security Number, Name, Address, Age, Salary, Dependents

However, Name may be a group item with the subitems Last, First and MI (middle initial). Also, Address may be a group item with the subitems Street address and Area address, where Area itself may be a group item having subitems City, State and ZIP code number. This hierarchical structure is pictured in Fig. 1-7(*a*). Another way of picturing such a tree structure is in terms of levels, as in Fig. 1-7(*b*).



(*a*)

```
01 Employee
     02 Social Security Number
     02 Name
          03 Last
          03 First
          03 Middle Initial
     02 Address
          03 Street
          03 Area
               04 City
               04 State
               04 ZIP
     02 Age
     02 Salary
     02 Dependents
```

(*b*)

Fig. 1-7

EXAMPLE 1.5 Algebraic Expressions

Consider the algebraic expression

$$(2x + y)(a - 7b)^3$$

Using a vertical arrow (↑) for exponentiation and an asterisk (*) for multiplication, we can represent the expression by the tree in Fig. 1-8. Observe that the order in which the operations will be performed is reflected in the diagram: the exponentiation must take place after the subtraction, and the multiplication at the top of the tree must be executed last.

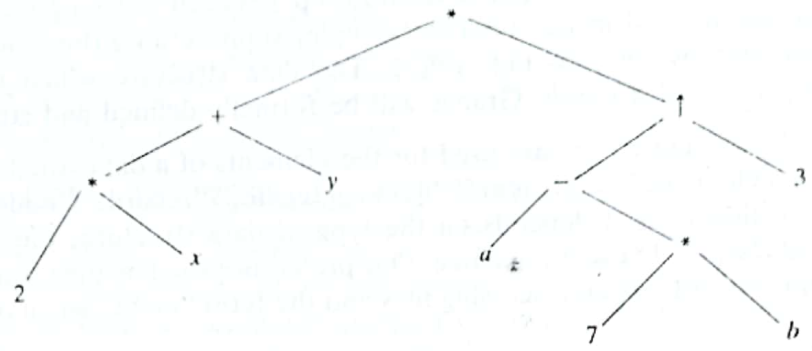
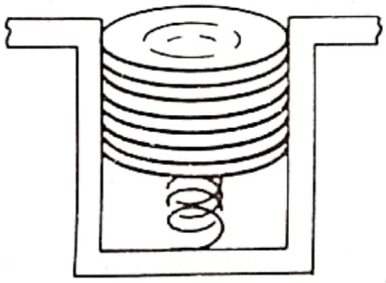


Fig. 1-8

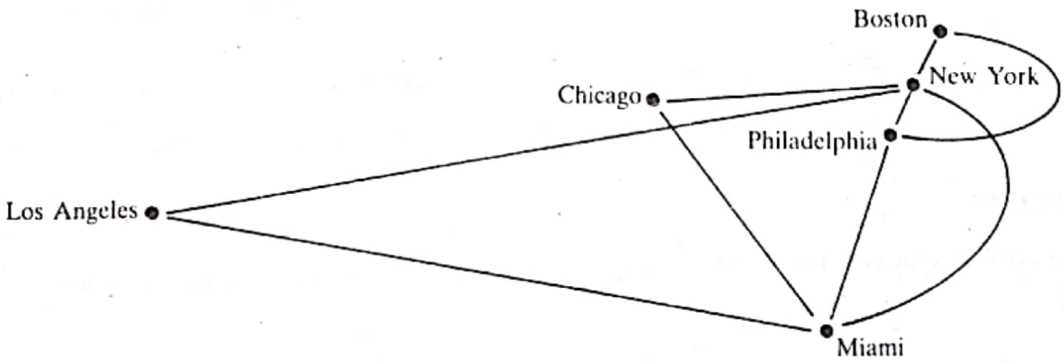
There are data structures other than arrays, linked lists and trees which we shall study. Some of these structures are briefly described below.

(*a*) *Stack*. A stack, also called a last-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the *top*. This structure is similar in its operation to a stack of dishes on a spring system, as pictured in Fig. 1-9(*a*). Note that new dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the stack.



(*a*) Stack of dishes.

(*b*) Queue waiting for a bus.



(*c*) Airline flights.

Fig. 1-9

(*b*) *Queue*. A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list. This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. 1-9(*b*): the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection—the first car in line is the first car through.

(c) *Graph*. Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. 1-9(c). The data structure which reflects this type of relationship is called a *graph*. Graphs will be formally defined and studied in Chap. 8.

Remark: Many different names are used for the elements of a data structure. Some commonly used names are "data element," "data item," "item aggregate," "record," "node" and "data object." The particular name that is used depends on the type of data structure, the context in which the structure is used and the people using the name. Our preference shall be the term "data element," but we will use the term "record" when discussing files and the term "node" when discussing linked lists, trees and graphs.

1.4 DATA STRUCTURE OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

The following four operations play a major role in this text:

(1) *Traversing*: Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)

(2) *Searching*: Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.

(3) *Inserting*: Adding a new record to the structure.

(4) *Deleting*: Removing a record from the structure.

Sometimes two or more of the operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

The following two operations, which are used in special situations, will also be considered:

(1) *Sorting*: Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

(2) *Merging*: Combining the records in two different sorted files into a single sorted file

Other operations, e.g., copying and concatenation, will be discussed later in the text.

EXAMPLE 1.6

An organization contains a membership file in which each record contains the following data for a given member:

Name, Address, Telephone Number, Age, Sex

(a) Suppose the organization wants to announce a meeting through a mailing. Then one would traverse the file to obtain Name and Address for each member.

(b) Suppose one wants to find the names of all members living in a certain area. Again one would traverse the file to obtain the data.

(c) Suppose one wants to obtain Address for a given Name. Then one would search the file for the record containing Name.

(d) Suppose a new person joins the organization. Then one would insert his or her record into the file.

(e) Suppose a member dies. Then one would delete his or her record from the file.

(*f*) Suppose a member has moved and has a new address and telephone number. Given the name of the member, one would first need to search for the record in the file. Then one would perform the "update"—i.e., change items in the record with the new data.

(*g*) Suppose one wants to find the number of members 65 or older. Again one would traverse the file, counting such members.

1.5 ALGORITHMS: COMPLEXITY, TIME-SPACE TRADEOFF

An algorithm is a well-defined list of steps for solving a particular problem. One major purpose of this text is to develop efficient algorithms for the processing of our data. The time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size. (The notion of complexity will be treated in Chap. 2.)

Each of our algorithms will involve a particular data structure. Accordingly, we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and the frequency with which various data operations are applied. Sometimes the choice of data structure involves a time-space tradeoff: by increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data, or vice versa. We illustrate these ideas with two examples.

Searching Algorithms

Consider a membership file, as in Example 1.6, in which each record contains, among other data, the name and telephone number of its member. Suppose we are given the name of a member and we want to find his or her telephone number. One way to do this is to linearly search through the file, i.e., to apply the following algorithm:

Linear Search: Search each record of the file, one at a time, until finding the given Name and hence the corresponding telephone number.

First of all, it is clear that the time required to execute the algorithm is proportional to the number of comparisons. Also, assuming that each name in the file is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with n records is equal to $n/2$; that is, the complexity of the linear search algorithm is given by $C(n) = n/2$.

The above algorithm would be impossible in practice if we were searching through a list consisting of thousands of names, as in a telephone book. However, if the names are sorted alphabetically, as in telephone books, then we can use an efficient algorithm called binary search. This algorithm is discussed in detail in Chap. 4, but we briefly describe its general idea below.

Binary Search: Compare the given Name with the name in the middle of the list; this tells which half of the list contains Name. Then compare Name with the name in the middle of the correct half to determine which quarter of the list contains Name. Continue the process until finding Name in the list.

One can show that the complexity of the binary search algorithm is given by

$$C(n) = \log_2 n$$

Thus, for example, one will not require more than 15 comparisons to find a given Name in a list containing 25 000 names.

Although the binary search algorithm is a very efficient algorithm, it has some major drawbacks. Specifically, the algorithm assumes that one has direct access to the middle name in the list or a sublist. This means that the list must be stored in some type of array. Unfortunately, inserting an element in an array requires elements to be moved down the list, and deleting an element from an array requires elements to be moved up the list.

The telephone company solves the above problem by printing a new directory every year while keeping a separate temporary file for new telephone customers. That is, the telephone company updates its files every year. On the other hand, a bank may want to insert a new customer in its file almost instantaneously. Accordingly, a linearly sorted list may not be the best data structure for a bank.

An Example of Time-Space Tradeoff

Suppose a file of records contains names, social security numbers and much additional information among its fields. Sorting the file alphabetically and using a binary search is a very efficient way to find the record for a given name. On the other hand, suppose we are given only the social security number of the person. Then we would have to do a linear search for the record, which is extremely time-consuming for a very large number of records. How can we solve such a problem? One way is to have another file which is sorted numerically according to social security number. This, however, would double the space required for storing the data. Another way, pictured in Fig. 1-10, is to have the main file sorted numerically by social security number and to have an auxiliary array with only two columns, the first column containing an alphabetized list of the names and the second column containing pointers which give the locations of the corresponding records in the main file. This is one way of solving the problem that is used frequently, since the additional space, containing only two columns, is minimal for the amount of extra information it provides.

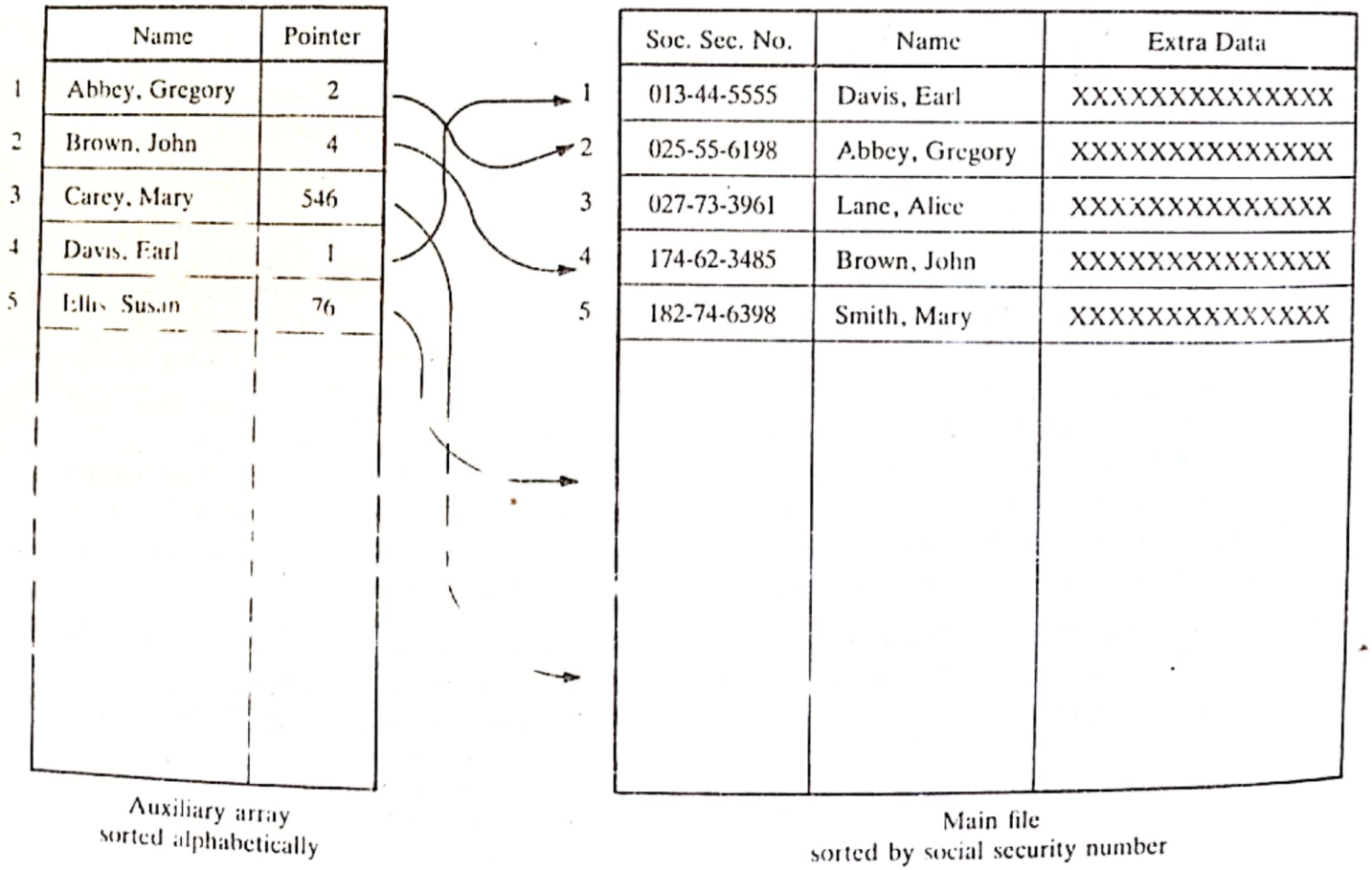


Fig. 1-10

Remark: Suppose a file is sorted numerically by social security number. As new records are inserted into the file, data must be constantly moved to new locations in order to maintain the sorted order. One simple way to minimize the movement of data is to have the social security number serve as the address of each record. Not only would there be no movement of data when records are inserted,

but there would be instant access to any record. However, this method of storing data would require one billion (10^9) memory locations for only hundreds or possibly thousands of records. Clearly, this tradeoff of space for time is not worth the expense. An alternative method is to define a function H from the set K of key values—social security numbers—into the set L of addresses of memory cells. Such a function H is called a *hashing function*. Hashing functions and their properties will be covered in Chap. 9.

Solved Problems

BASIC TERMINOLOGY

1.1 A professor keeps a class list containing the following data for each student:

Name, Major, Student Number, Test Scores, Final Grade

(*a*) State the entities, attributes and entity set of the list.

(*b*) Describe the field values, records and file.

(*c*) Which attributes can serve as primary keys for the list?

(*a*) Each student is an entity, and the collection of students is the entity set. The properties, name, major, and so on, of the students are the attributes.

(*b*) The field values are the values assigned to the attributes, i.e., the actual names, test scores, and so on. The field values for each student constitute a record, and the collection of all the student records is the file.

(c) Either Name or Student Number can serve as a primary key, since each uniquely determines the student's record. Normally the professor uses Name as the primary key, but the registrar may use Student Number.

1.2 A hospital maintains a patient file in which each record contains the following data:

Name, Admission Date, Social Security Number, Room, Bed Number, Doctor

(*a*) Which items can serve as primary keys?

(*b*) Which pair of items can serve as a primary key?

(*c*) Which items can be group items?

(*a*) Name and Social Security Number can serve as primary keys. (We assume that no two patients have the same name.)

(*b*) Room and Bed Number in combination also uniquely determine a given patient.

(*c*) Name, Admission Date and Doctor may be group items.

1.3 Which of the following data items may lead to variable-length records when included as items in the record: (*a*) age, (*b*) sex, (*c*) name of spouse, (*d*) names of children, (*e*) education, (*f*) previous employers?

Since (*d*) and (*f*) may contain a few or many items, they may lead to variable-length records. Also, (*e*) may contain many items, unless it asks only for the highest level obtained.

1.4 Data base systems will be only briefly covered in this text. Why?

"Data base systems" refers to data stored in the secondary memory of the computer. The implementation and analysis of data structures in the secondary memory are very different from those in the main memory of the computer. This text is primarily concerned with data structures in main memory, not secondary memory.

DATA STRUCTURES AND OPERATIONS

1.5 Give a brief description of (*a*) traversing, (*b*) sorting and (*c*) searching.

(*a*) Accessing and processing each record exactly once

(*b*) Arranging the data in some given order

(*c*) Finding the location of the record with a given key or keys

1.6 Give a brief description of (*a*) inserting and (*b*) deleting.

(*a*) Adding a new record to the data structure, usually keeping a particular ordering

(*b*) Removing a particular record from the data structure

1.7 Consider the linear array NAME in Fig. 1-11, which is sorted alphabetically.

(*a*) Find NAME[2], NAME[4] and NAME[7].

(*b*) Suppose Davis is to be inserted into the array. How many names must be moved to new locations?

(*c*) Suppose Gupta is to be deleted from the array. How many names must be moved to new locations?

(*a*) Here NAME[K] is the *k*th name in the list. Hence,

NAME[2] = Clark, NAME[4] = Gupta, NAME[7] = Pace

(*b*) Since Davis will be assigned to NAME[3], the names Evans through Smith must be moved. Hence six names are moved.

(*c*) The names Jones through Smith must be moved up the array. Hence four names must be moved.

NAME

1	Adams
2	Clark
3	Evans
4	Gupta
5	Jones
6	Lane
7	Pace
8	Smith

Fig. 1-11

1.8 Consider the linear array NAME in Fig. 1-12. The values of FIRST and LINK[K] in the figure determine a linear ordering of the names as follows. FIRST gives the location of the first name in the list, and LINK[K] gives the location of the name following NAME[K], with 0 denoting the end of the list. Find the linear ordering of the names.

The ordering is obtained as follows:

FIRST = 5, so the first name in the list is NAME[5], which is Brooks.
LINK[5] = 2, so the next name is NAME[2], which is Clark.
LINK[2] = 8, so the next name is NAME[8], which is Fisher.
LINK[8] = 4, so the next name is NAME[4], which is Hansen.
LINK[4] = 10, so the next name is NAME[10], which is Leary.
LINK[10] = 6, so the next name is NAME[6], which is Pitt.
LINK[6] = 1, so the next name is NAME[1], which is Rogers.
LINK[1] = 7, so the next name is NAME[7], which is Walker.
LINK[7] = 0, which indicates the end of the list.

Thus the linear ordering of the names is Brooks, Clark, Fisher, Hansen, Leary, Pitt, Rogers, Walker. Note that this is the alphabetical ordering of the names.

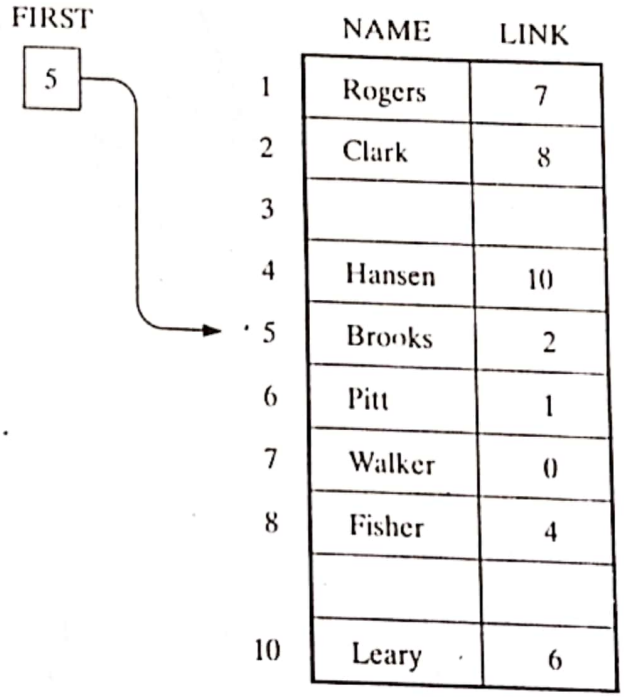
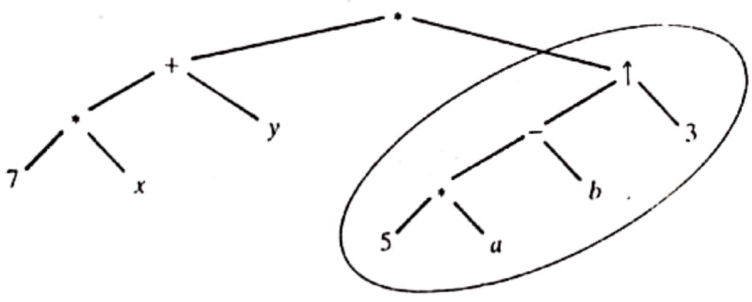


Fig. 1-12



1.9 Consider the algebraic expression $(7x + y)(5a - b)^3$. (*a*) Draw the corresponding tree diagram as in Example 1.5. (*b*) Find the scope of the exponential operation. (The scope of a node v in a tree is the subtree consisting of v and the nodes following v.)

(*a*) Use a vertical arrow (↑) for exponentiation and an asterisk (*) for multiplication to obtain the tree in Fig. 1-13.

(*b*) The scope of the exponentiation operation ↑ is the subtree circled in the diagram. It corresponds to the expression $(5a - b)^3$.

1.10 The following is a tree structure given by means of level numbers as discussed in Example 1.4:

01 Employee 02 Name 02 Number 02 Hours 03 Regular 03 Overtime 02 Rate

Draw the corresponding tree diagram.

The tree diagram appears in Fig. 1-14. Here each node v is the successor of the node which precedes v and has a lower level number than v.

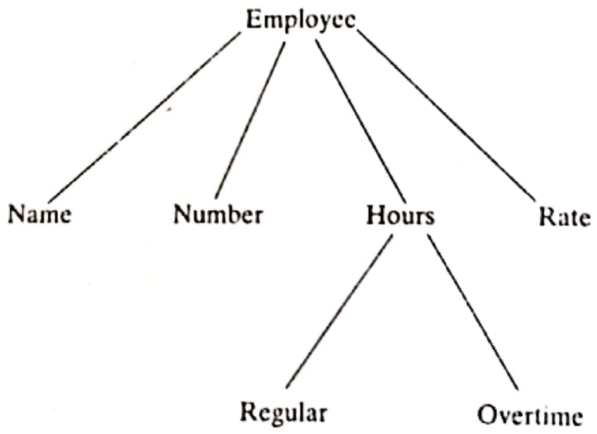


Fig. 1-14

1.11 Discuss whether a stack or a queue is the appropriate structure for determining the order in which elements are processed in each of the following situations.

(*a*) Batch computer programs are submitted to the computer center.

(*b*) Program A calls subprogram B, which calls subprogram C, and so on.

(*c*) Employees have a contract which calls for a seniority system for hiring and firing.

(*a*) Queue. Excluding priority cases, programs are executed on a first come, first served basis.

(*b*) Stack. The last subprogram is executed first, and its results are transferred to the next-to-last program, which is then executed, and so on, until the original calling program is executed.

(*c*) Stack. In a seniority system, the last to be hired is the first to be discharged.

1.12 The daily flights of an airline company appear in Fig. 1-15. CITY lists the cities, and ORIG[K] and DEST[K] denote the cities of origin and destination, respectively, of the flight NUMBER[K]. Draw the corresponding directed graph of the data. (The graph is directed because the flight numbers represent flights from one city to another but not returning.)

The nodes of the graph are the five cities. Draw an arrow from city A to city B if there is a flight from A to B, and label the arrow with the flight number. The directed graph appears in Fig. 1-16.

	CITY
1	Atlanta
2	Boston
3	Chicago
4	Miami
5	Philadelphia

(*a*)

	NUMBER	ORIG	DEST
1	701	2	3
2	702	3	2
3	705	5	3
4	708	3	4
5	711	2	5
6	712	5	2
7	713	5	1
8	715	1	4
9	717	5	4
10	718	4	5

(*b*)

Fig. 1-15

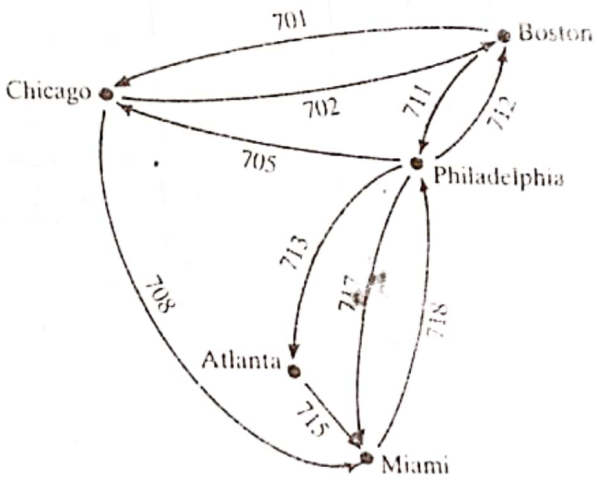


Fig. 1-16

COMPLEXITY; SPACE-TIME TRADEOFFS

1.13 Briefly describe the notions of (*a*) the complexity of an algorithm and (*b*) the space-time tradeoff of algorithms.

(*a*) The complexity of an algorithm is a function $f(n)$ which measures the time and/or space used by an algorithm in terms of the input size n.

(*b*) The space-time tradeoff refers to a choice between algorithmic solutions of a data processing problem that allows one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa.

1.14 Suppose a data set S contains n elements.

(*a*) Compare the running time T_1 of the linear search algorithm with the running time T_2 of the binary search algorithm when (i) $n = 1000$ and (ii) $n = 10\,000$.

(*b*) Discuss searching for a given item in S when S is stored as a linked list.

(*a*) Recall (Sec. 1.5) that the expected running of the linear search algorithm is $f(n) = n/2$ and that the binary search algorithm is $f(n) = \log_2 n$. Accordingly, (i) for $n = 1000$, $T_1 = 500$ but $T_2 = \log_2 1000 \approx 10$; and (ii) for $n = 10\,000$, $T_1 = 5000$ but $T_2 = \log_2 10\,000 \approx 14$.

(*b*) The binary search algorithm assumes that one can directly access the middle element in the set S. But one cannot directly access the middle element in a linked list. Hence one may have to use a linear search algorithm when S is stored as a linked list.

1.15 Consider the data in Fig. 1-15, which gives the different flights of an airline. Discuss different ways of storing the data so as to decrease the time in executing the following:

(*a*) Find the origin and destination of a flight, given the flight number.

(*b*) Given city A and city B, find whether there is a flight from A to B, and if there is, find its flight number.

(*a*) Store the data of Fig. 1-15(*b*) in arrays ORIG and DEST where the subscript is the flight number, as pictured in Fig. 1-17(*a*).

(*b*) Store the data of Fig. 1-15(*b*) in a two-dimensional array FLIGHT where FLIGHT[J, K] contains the flight number of the flight from CITY[J] to CITY[K], or contains 0 when there is no such flight, as pictured in Fig. 1-17(*b*).

	ORIG	DEST
701	2	3
702	3	2
703	0	0
704	0	0
705	5	3
706	0	0
⋮	⋮	⋮
715	1	4
716	0	0
717	5	4
718	4	5

(*a*)

FLIGHT	1	2	3	4	5
1	0	0	0	715	0
2	0	0	701	0	711
3	0	702	0	708	0
4	0	0	0	0	718
5	713	712	705	717	0

(*b*)

Fig. 1-17

16 Suppose an airline serves n cities with s flights. Discuss drawbacks to the data representations used in Fig. 1-17(*a*) and Fig. 1-17(*b*).

(*a*) Suppose the flight numbers are spaced very far apart; i.e., suppose the ratio of the number s of flights to the number of memory locations is very small, e.g., approximately 0.05. Then the extra storage space may not be worth the expense.

(*b*) Suppose the ratio of the number s of flights to the number n of memory locations in the array FLIGHT is very small, i.e., that the array FLIGHT is one that contains a large number of zeros (such an array is called a sparse matrix). Then the extra storage space may not be worth the expense.