DATA STRUCTURE WITH C

(PGDIT02)

(PG - DIPLOMA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

DATA STRUCTURE WITH C

(DMSIT04)

(MSC - IT)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

Lesson 1

Introduction and overview

1.0 Objectives

This lesson introduces some basic data structures that are useful in designing efficient algorithms for large classes of problems.

After completion of this lesson the student will be able to know about

- Data organization
- Different data structures
- Algorithm
- Complexity of an algorithm

Structure of the Lesson:

1.1 Introduction

In computer science, a data structure is an arrangement of data in a computer's memory. A well-designed data structure allows set of operations on its values. These operations vary functionally from one data structure to another data structure. The representation of a particular data structure in the memory of a computer is called a storage structure.

We will see different kinds of data structures later in this book. In the next section, elementary data organization, overview of data structures will be discussed.

1.2 Elementary Data Organization

Data: Data means value or set of values. A *data item* refers to a single unit of values. Data items that are divided into subitems are called *grouped data items*. The data items, which cannot be further divided into subitems, are called *elementary data items*.

For *example*, an employee's name may be divided into 3 subitems,
First name, middle name and last name but the social security number would normally be treated as a single item. Thus employee name is a grouped data item and social security number is an elementary data item.

Entity: An *entity* is one that has certain attributes and which may be assigned values. The values may be either numeric or nonnumeric. For example, an employee in an organization is an entity.

The possible attributes and the corresponding values for an entity in the present example are:

Attributes:	EmpName	DOB	Sex	Security Number
Values:	Ravi Anand	30/12/80	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 and is called the *domain* of attribute. Domain is the set of all possible values that could be assigned to the particular attribute. For example, in the EMPLOYEE entity, the attribute Sex has domain as {M, F}. It can be noted that, an entity after assigning its values gives composite data.

Information: Information can be defined as meaningful data or processed data.

Collections of data are frequently organized into a hierarchy of fields, records and files.

Field is a single elementary unit of information representing an attribute of an entity.

Record is the collection of field values of a given entity.

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 is called a *primary key*, and the value k1, K2, … in such a field are called *keys* or *key values*.

Example:

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

There are 5 data items Employee number, Employee Name, Address, Telephone Number and Dues Owed. Employee Name and Address may be group items. Here the Employee number field is a primary key. Note that the Address and Telephone Number fields may not serve as primary keys, 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 or variable–length of 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, records may contain different lengths. For example, student records have variable lengths, since different students take different number of courses. Usually variable-length records have a minimum and a maximum length.

Data are also organized into more complex types of data structures. The study of such data structures, includes the following 3 steps:

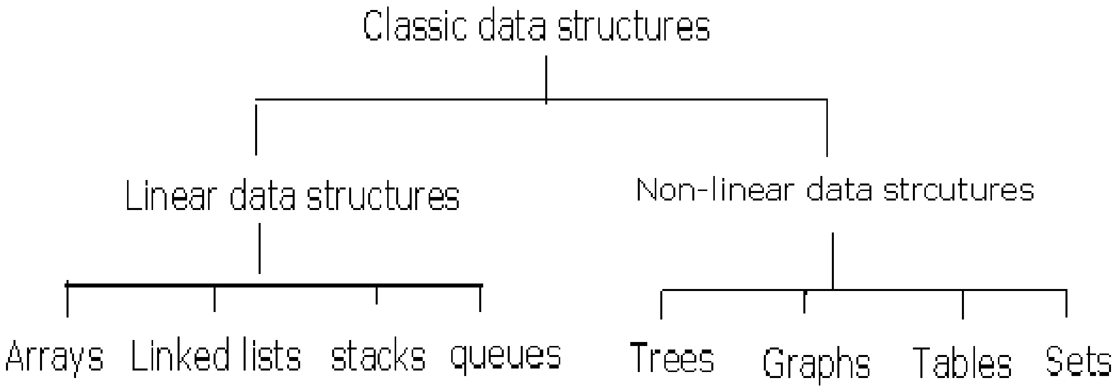
1. Logical or mathematical description of the structure
2. Implementation of the structure on a Simputer.

   [The Simputer is a small inexpensive handheld computer. The word “Simputer” is an acronym for “simple, inexpensive and multilingual people’s 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

1.3 Overview Of Data Structures

In computer science, several data structures are known depending on area of applications. Of them, few data structures are there which are frequently used in all application areas. These data structures are known as fundamental data structures or classical data structures. Fig. 1.1 gives a classification of all classic data structures.

Fig. 1.1



Classic data structures are classified into 2 main classes:
Linear data structures and non- linear data structures

Linear data structures – Elements are stored in a memory location in sequential order.
Examples: Arrays, Stacks, Queues, Linked lists

Non-linear data structures – The data structure that represents hierarchical relationship.
Examples: Tree, Graph, and Table, Set

1.4 What is a data structure?

Data Structure: The logical or mathematical model of a particular organization of data is called *data structure*.

Or

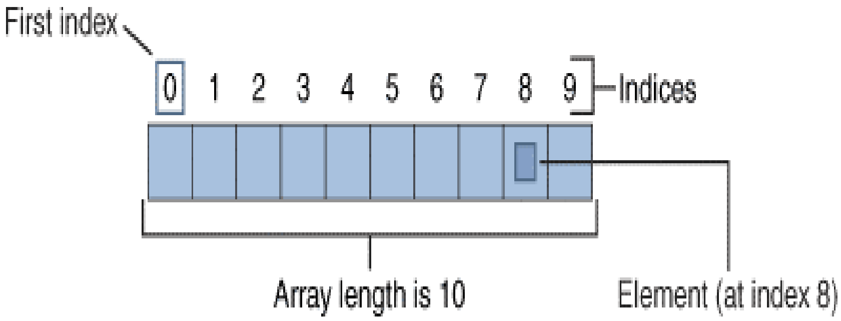
The way in which sets of data are organized in a particular system is called *data structure*.

- **Arrays:** The simplest type of data structure is a linear (or one-dimensional) array.

An array is a finite, ordered and collection of homogeneous data elements. Array is finite because it contains only limited number of elements, and ordered, as all the elements are stored one by one.

Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is given by an *index*, which is also called a subscript. The index usually uses a consecutive range of integers. The length of an array is established when the array is created. After creation, an array is a fixed-length structure.

Fig. 1.2



An *array element* is one of the values within an array and is accessed by its position within the array.

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)…..A(N)

Or by the bracket notation

A[1], A[2], A[3],…..,A[N]

The number K in A[K] is called a subscript and A[K] is called a *subscripted variable*.

The parenthesis 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.

Linear arrays are called *one-dimensional* arrays because each element in an array is referenced by one subscript.

A *two-dimensional* array is a collection of similar data elements where each element is referenced by subscripts (such arrays are called *matrices* in mathematics, and *tables* in business applications).

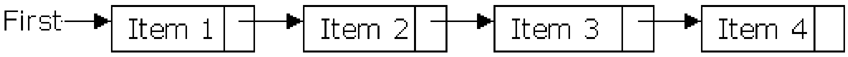
- **Linked lists:** Mathematically, a list is a finite sequence of items. We develop a data structure that allows implementing lists in which items can be added or deleted. Consider the list:

Item 1, Item 2, Item 3, Item 4

Implementation of this linked structure is as shown in Fig.1.3.

Each element in the structure consists of 2 memory locations. The first contains the item itself; the second contains a pointer to the next element.

Fig. 1.3



Trees: A tree is collection of nodes, but each node may *refer* multiple neighbors. Trees can be used to model *hierarchical organization* of data. The data structure that reflects this relationship is called a *rooted tree graph* or simply a *tree*. Trees will be defined and discussed in detail in chapter7.

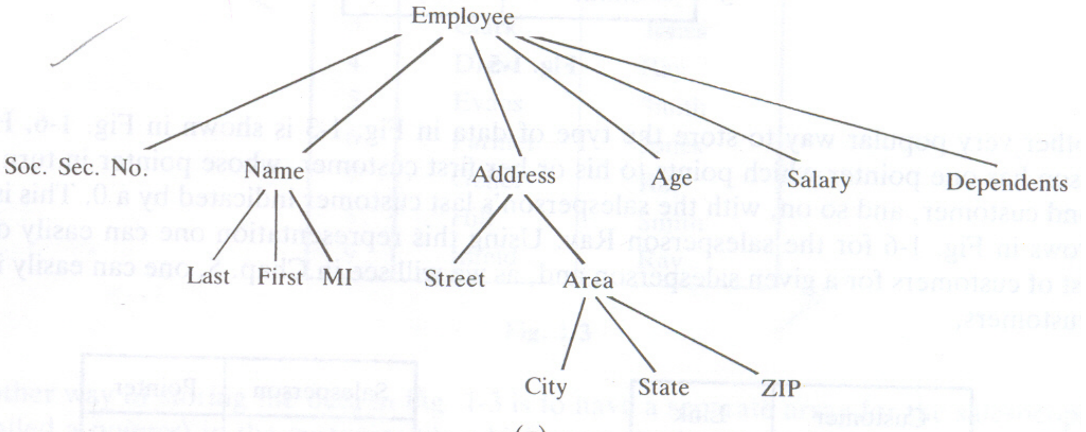
Example: Record Structure

A record may consist of both the group items and the elementary items as its attributes. It can 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 name, First name and Middle name. Address may also 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. 1.4

Fig. 1.4



(*a*)

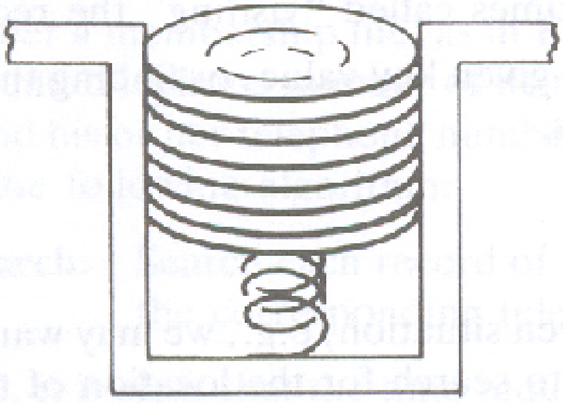
There are data structures other than arrays, linked lists and trees briefly described below.

Stack: A stack is a homogeneous collection of items of any one type, arranged linearly with access at one end only, called the *top*. This means that data can be added or removed from only the top. Formally this type of stack is called a Last In First Out (LIFO) stack. Data is added to the stack using the *Push* operation, and removed using the *Pop* operation.

This structure is similar in its operation to a stack of dishes on a spring system, as pictured in Fig. 1.5. 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

Fig.1.5

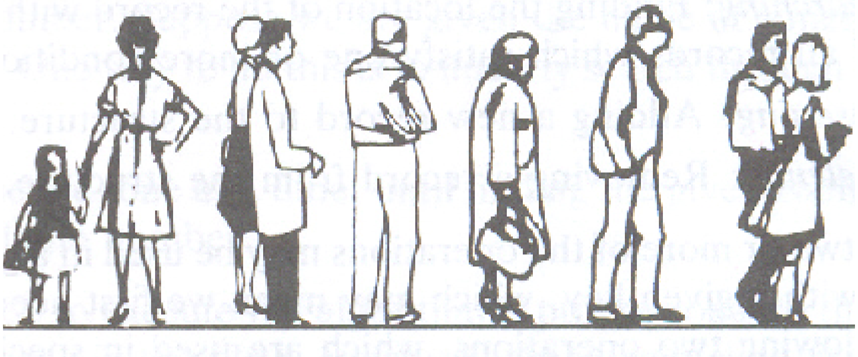


(*a*) Stack of dishes.

Queue: is an ordered homogeneous group of items in which the items are added at one end (the back) and are removed from the other end (the front). This is known as a First In First Out (FIFO) data structure. Items can only be removed from the queue in the order that they are inserted into it. Queues in computer science are very similar to queues in real life.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. 1.6 the first person in line is the first person to board the bus

Fig. 1.6



(*b*) Queue waiting for a bus.

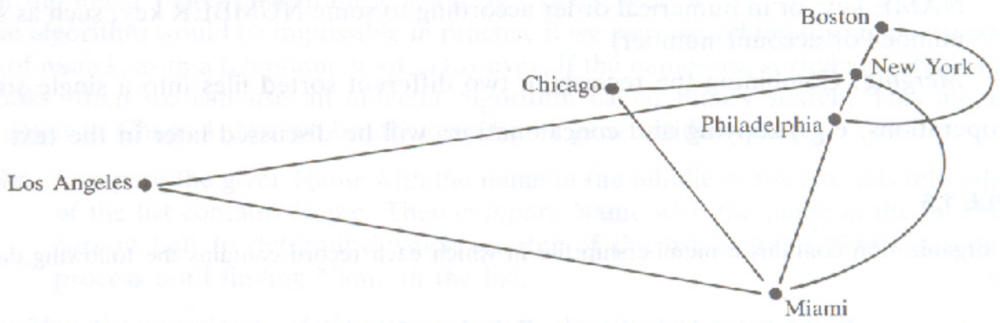
Graph: Data sometimes contain a relationship between pairs of elements, which is not necessarily hierarchical in nature.

A *graph* G= (V, E) consists of a finite non-empty set of *vertices* V and set of *edges* E.

If the edges are ordered pairs (v, w) of vertices, then the graph is said to be *directed*, v is called the *tail* and w is the *head* of the edge (v, w). If the edges are unordered pairs of distinct vertices then the graph is said to be *undirected*.

For example, suppose an airline fly only between the cities connected by lines in Fig. 1.7

Fig. 1.7



(*c*) Airline flights.

1.5 Data Structure Operations

The data appearing in our data structures are processed by means of certain operations. Some of the operations are:

Traversing*:* Accessing each record exactly once so that certain items in the record may be processed.

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.

Inserting: Adding a new record to the structure

Deleting*:* Removing a record from the structure

Some times two or more operations may be used in a given situation.

Ex: To delete the record with the given key, first of all we need to search for the location of the record.

The following two operations are used in special situation.

1) **Sorting:** Arranging the records in some logical order.

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.

1.6 What is an Algorithm?

An *algorithm* defines some process, how an operation works. It is specified in terms of simple steps. Each step must be clearly implementable in a mechanical way. An algorithm must satisfy the following properties.

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. **Definiteness:** Each instruction must be clear and unambiguous.
4. **Finiteness**: If we trace out the instructions of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
5. Effectiveness: Every instruction must be very basic so

   that it can be carried out, in principle, by a person using only pencil and paper.

Algorithms that are definite and effective are also called computational process.

Example: Operating system of a digital computer

- **Complexity:** The complexity of an algorithm is the function, which gives the running time and/or space in terms of the input size. We analyze algorithms to determine their cost in terms of time and storage. (The notion of complexity will be treated in Chapter 2)

Time complexity Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity The space complexity of an algorithm is the amount of memory it needs to run to completion.

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 trade off.

- **Searching Algorithms**

Consider a membership file, 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:** Linear search is a search algorithm, also known as sequential search, which is suitable for searching a set of data for a particular value. It searches each record of the file, one at a time, until a match is found.

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 number in the file is to be picked, 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 takes more time, if we were searching through a list consisting of thousands of names, as in a telephone book. However, if the names are stored alphabetically, as in telephone books, then we can search in an efficient manner. Implementing this idea is called binary search. This algorithm is discussed in detail in Chap. 4, but we briefly describe its general idea below.

- **Binary search:** The most common application of binary search is to find a specific value in a sorted list. The search begins by examining the value in the center of the list, because the values are sorted, it then knows whether the value occurs before or after the center value, and searches through the correct half in the same way.

Compare the given number with the number in the middle of the list, this list tells which of the list contains number. Then compare number with the number in the first half part, or in the second half part depending upon the number. Continue the process until finding number in the list.

The complexity of the binary search algorithm is given by

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

Thus, for example, it will not require more than 15 comparisons to find a given name in a list containing 25000 names.

The binary search algorithm is a very efficient algorithm. It has some major drawbacks. Specifically, the algorithm assumes that there is a direct access to the middle number in the list or a sublist. This means that the list must be stored in some type of array. 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. According, a linearly sorted list may not be the best data structure for a bank.

1.7 Characteristics Of Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out access	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search

Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Hash Table	Very fast access if key is known	Slow deletes Access slow if key is not
	Quick inserts	known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items

1.8 Summary

This chapter covers important data structures and their use in application development. It focuses on an introduction to data structures, defining what data structures are, how the efficiency of data structures are analyzed, and why this analysis is important.

We began by introducing the idea of a data structure as a way of organizing data in a computer's memory.

We examined a class of data structures called linear data structures that are organized in a linear fashion. We have discussed the linear data structures arrays, linked lists, stacks and queues.

In the stack, items can only be added and removed from the top of the stack. Items can only be removed from the queue in the order that they are inserted into it.

We looked at another class of data structures call nonlinear data structures that are organized quite differently from the computer's memory. These data structures included trees and graphs.

1.9 Model Questions

1. Define a data Structure?
2. What are the *different* types of data structures?
3. Explain the different operations of data structures?
4. What is an Algorithm?
5. Define i) Time Complexity ii) Space Complexity.

1.10 References

- Theory and problems of data structures

  Schaum's outline series

- Data structures

  Horowitz Sahani

- K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science,
J.K.C College, GUNTUR

Lesson 2

PRELMINARIES

2.0 Objectives

After completion of this chapter the student will be able to know the following concepts.

- Mathematical notations and functions
- Algorithmic notation
- Control structures
- Complexity of algorithms

Structure of the Lesson:

2.1 Introduction

This chapter describes some mathematical notations and the format that will be used to present algorithms.

Implementing the algorithms is easy, if they are organized into hierarchies of modules as shown in fig. 2.1. In this organization, each program contains first a main module, which gives a general description of the algorithm; this main module refers to certain submodules, which contain more detailed information than the main module. Each of the submodules may refer to more detailed submodules and so on. The organization of a program into such a hierarchy of modules normally requires certain basic flow patterns and logical structures, which are usually associated with the notion of structured programming. These flow patterns and logical structures will be reviewed in this chapter

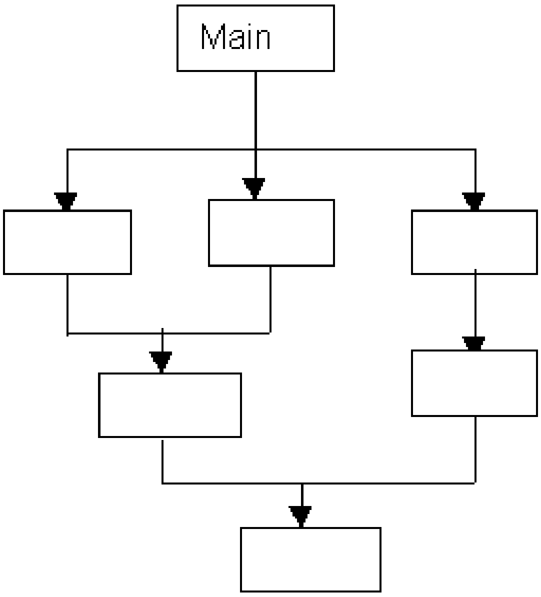


Fig. 2.1 A hierarchy of modules

This chapter begins with a brief outline and discussion of various mathematical functions, which occur in the study of algorithms and in computer science in general, and the chapter ends with a discussion of the different kinds of variables that can appear in our algorithms and programs.

The notion of the complexity an algorithm is also covered in this chapter. This important measurement of algorithms gives us a tool to compare different algorithmic solutions to a particular problem such as searching or sorting. The concept of an algorithm and its complexity is fundamental not only to data structures but also to almost all areas of computer science.

2.2 Mathematical Notation and Functions

This section gives various mathematical functions, which appear very often in the analysis of algorithms and in computer science in general, together with their notation

2.2.1 Floor and ceiling Functions

Let x be any real number. Then x lies between two integers called the floor and the ceiling of x. Specifically,

$\lfloor x \rfloor$ Called the floor of x, denotes the greatest integer that does not exceed x.

$\lceil x \rceil$ Called the ceiling of x, denotes the least integer that is not less than x.

Example

$$\lfloor 3.14 \rfloor = 3,\ \lfloor \sqrt{5} \rfloor = 2, \lfloor -8.5 \rfloor = 9, \lfloor 7 \rfloor = 7$$

$$\lceil 3.14 \rceil = 4,\ \left| \sqrt{5} \right| = 3, \lceil -8.5 \rceil = -8, \lceil 7 \rceil = 7$$

2.2.2 Remainder Function

Modular Arithmetic

Let k be any integer and let M be a positive integer. Then

K (mod M)

(read k modulo M) will denote the integer remainder when k is divided by M. More exactly, k (mod M) is the unique integer r such that

[illegible] $1-5 \equiv 8,\ 2+10 \equiv 0 \equiv 12$

K = Mq + r where 0$\leq$ r< M

The term **“mod”** is also used for the mathematical congruence relation, which is denoted and defined as follows:

if and only if M divides b-a

M is called the modulus, and $a \equiv b$ (mod M) is read “ a is congruent to b modulo M”.

The following aspects of the congruence relation are frequently useful

and

Arithmetic modulo M refers to the arithmetic operations of addition, multiplication and subtraction where the arithmetic value is replaced by its equivalent value in the set

{0, 1,2, … M-1}

or in the set

{1, 2, 3… M}

For example, in arithmetic modulo 12, sometimes called “clock” arithmetic,

(The use of 0 or M depends on the application.)

2.2.3 Integer and Absolute value Functions

Let be any real number. The *integer value* of written INT ( ) converts x into an integer by deleting (truncating) the fractional part of the number. Thus

INT (3.14) = 3, INT = 2, INT (-8.5) = -8, INT (7)=7

Observe that INT $(x)=\lfloor x\rfloor$or INT $(x)=\lceil x\rceil$according to whether x is positive or negative

The absolute value of the real number written

ABS or $|x|$is defined as the greater of x or -

Hence ABS(0) = 0, and, for ABS(x) = or ABS( )= - depending on whether is positive or negative.

Example:

2.2.4 Summation Symbol

Here we introduce the summation symbol (the Greek letter sigma). Consider a sequence a1, a2, a3, ... Then the sums

and $a_m + a_{m+1} + \dots + a_n$

will be denoted, respectively, by

$$\sum_{j=1}^{n} a_j \text{ and } \sum_{j=m}^{n} a_j$$

The letter j in the above expression is called a dummy index or dummy variable.

Example:

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

$$\sum_{j=2}^{5} j^2 = 2^2 + 3^2 + 4^2 + 5^2 = 4 + 9 + 16 + 25 = 54$$

$$\sum_{j=1}^{n} j = 1 + 2 + \dots + n$$

The last sum in example has the value n (n + 1)/ 2. That is,

1 + 2 + 3 + … + n = n (n + 1)/ 2

2.2.5 Factorial Function

The product of the positive integers from 1 to n, is denoted by n! (read “n” factorial”). That is,
n! = 1.2.3… (n-2)(n-1) n

It is also convenient to define 0!= 1.

Example:

(a) 2!= 1.2 = 2; 3!= 1.2.3 =6; 4!= 1.2.3.4 = 24

(b) For n> 1, we have n!= n. (n –1)! Hence

5! = 5.4! = 5. 24 = 120!; 6! = 6.5! = 6.120 = 720

- **Permutations**

A permutation of a set of n elements is an arrangement of the elements in a given order. For example, the permutations of the set consisting of the elements a, b, c are as follows:

abc, acb, bac, cab, cba

Example:

4! = 24 permutations of a set with 4 elements

5! = 120 permutations of a set with 5 elements

2.2.6 Exponents and Logarithms

Recall the following definitions for integer exponents (where m is a positive integer):

a^m = a. a. …..a (m times), $a^0 = 1$, $a^{-m} = 1/ a^m$

Exponents are extended to include all rational numbers by defining, for any rational number m/n,

$$\sqrt[n]{a^m} = \left(\sqrt[n]{a}\right)^m$$

$a^{m/n} =$

Example:

$2^4 = 16$, $2^{-4} = 1/ 2^4 = 1/16$, $125^{2/3} = 5^2 = 25$

In fact, exponents are extended to include all real numbers by defining, for any real number x,

$a^x = \lim_{r \to x} a^r$ Where r is rational number

Accordingly the exponential function $f(x) = a^x$ is defined for all real numbers.

Logarithms are related to exponents as follows. Let b be a positive number. The logarithm of any positive number x to the base b, written

$$\log_b^x$$

It represents the exponent to which b must be raised to obtain x. That is

$y = \log_b^x$ and $b^y = x$ are equivalent statements.

Accordingly,

$\log_2 8 = 3$ since $2^3 = 8$; $\log_{10} 100 = 2$ since $10^2 = 100$

$\log_2$ 64= 6 since $2^6 = 64$; $\log_{10}$ 0.001 = -3 since 10^{-3}= 0.001

For any base b $\log_b$ 1=0 since $b^{0\ =1}$

$\log_b$ = 1 since b^1 =b

The logarithm of a negative number and the logarithm of 0 are not defined.

We can view the exponential and logarithmic functions as inverse of each other.

$f(x) = b^x$ and $g(x) = \log_b x$

We can view the logarithms with different base values. Those are,

Logarithms to the base 10 (called common logarithms), Logarithms to the base e (called natural logarithms) and Logarithms to the base 2 (called binary logarithms)

2.3 Algorithmic Notation

An algorithm is a finite step-by-step list of well-defined instructions for solving a particular problem. This section describes the format that is used to present algorithms throughout the text. This algorithmic notation is described by examples.

Example:

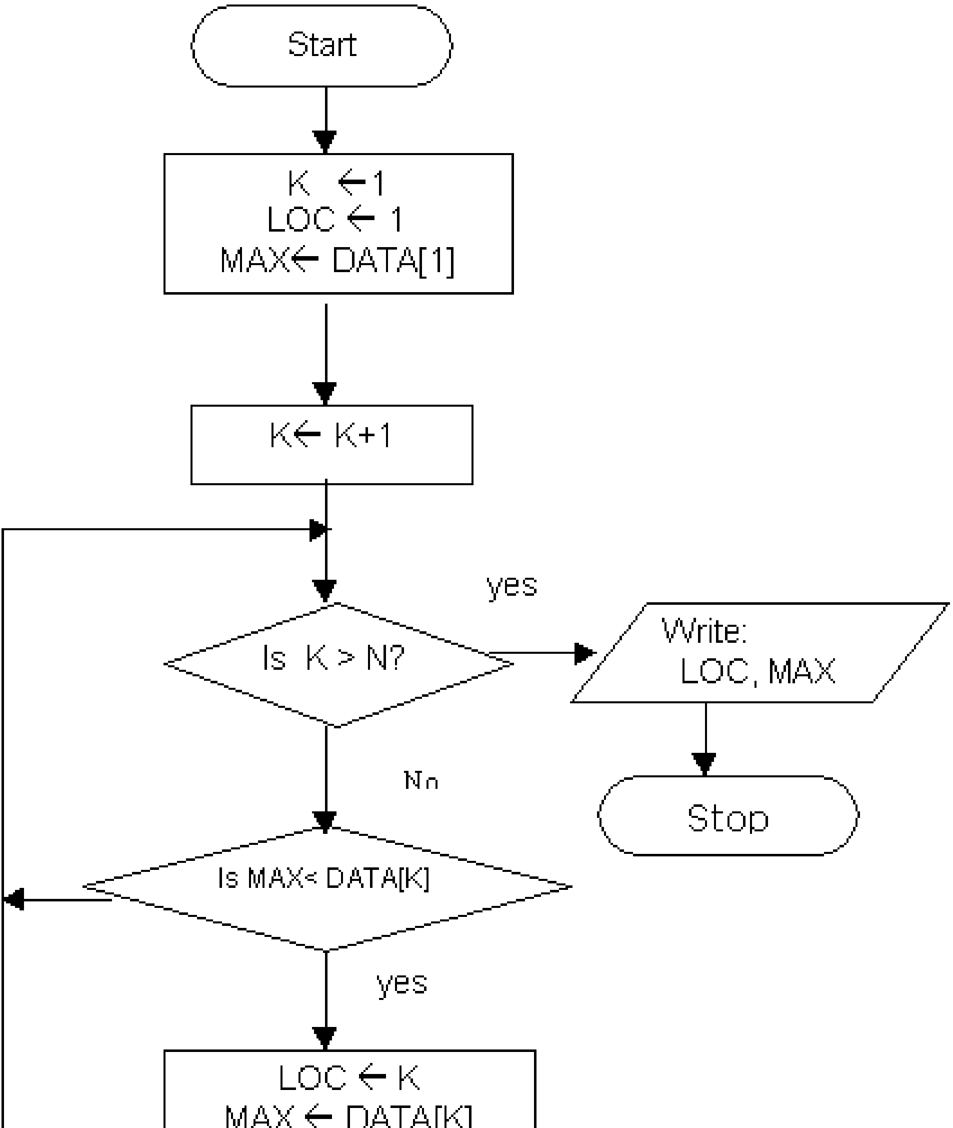
An array DATA of numerical values is in memory. We want to find the location LOC and the value MAX of the largest element of DATA. Given no other information about DATA, one way to solve the problem is as follows:

- Initially begin with LOC=1 and MAX= DATA[1].

  Then compare MAX with each successive element DATA[K] of DATA.

  If DATA[K] exceeds MAX, then update LOC and MAX so that LOC =K and MAX=DATA[K].

  The final values appearing in LOC and MAX give the location and value of the largest element of DATA.



't is as shown in fig. 2.2.

array DATA with N numerical values.

f the largest element of DATA. The

```
1.  [Initialize.] Set K:= 1, LOC :=1 and MAX := DATA[1].
2.  [Increment counter.] Set K: = K+1.
3.  [Test counter.] If K>N, then:
      Write: LOC, MAX and Exit
4.  [Compare and update.] If MAX< DATA [K], then:
      Set LOC := K and MAX:= DATA[K].
5.  [Repeat loop.] Go to Step 2.
```

The above algorithm consists of two parts. The first part is a paragraph, which tells the purpose of the algorithm identifies the variables that occur in the algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.

Steps, Control, Exit

The steps of the algorithm are executed one after the other, beginning with step 1, unless indicated otherwise. Control may be transferred to step n of the algorithm by the statement “ Go to Step n.” For example, Step 5 transfers control back to Step 2 in Algorithm 2.1.

If several statements appear in the same step, then they are executed from left to right.

Ex: Set K:=1, LOC :=1 and MAX :=DATA[1].

The algorithm is completed when the statement *exit* is encountered. This statement is similar to the STOP statement used in flowcharts.

Comments

Each step may contain a comment in brackets, which indicates the main purpose of the step. The comment will usually appear at the beginning or the end of the step.

Variable Names

Variable names will use capital letters, as in MAX and DATA. Single-letter names of variables used as counters or subscripts will also be capitalized in the algorithms (K and N, for example).

Assignment Statement

Assignment statements will use the dots-equal notation: = that is used in Pascal. For example,

Max: = DATA [1]

Assigns the value in DATA [1] to MAX. Some texts use the backward arrow ← or the equal sign = for this operation.

Input and Output

To take the data may be input use read statement with the following form:

Read: Variables names

Similarly, messages placed in quotation marks, and data in variables may be output use a write or print statement with the following form:

Write: Messages and/or variable names

Procedures

The term “procedure” will be used for an independent algorithmic module, which solves a particular problem. The word “algorithm” will be reserved for the solution of general problems. The term “procedure” will also be used to describe a certain type of subalgorithm.

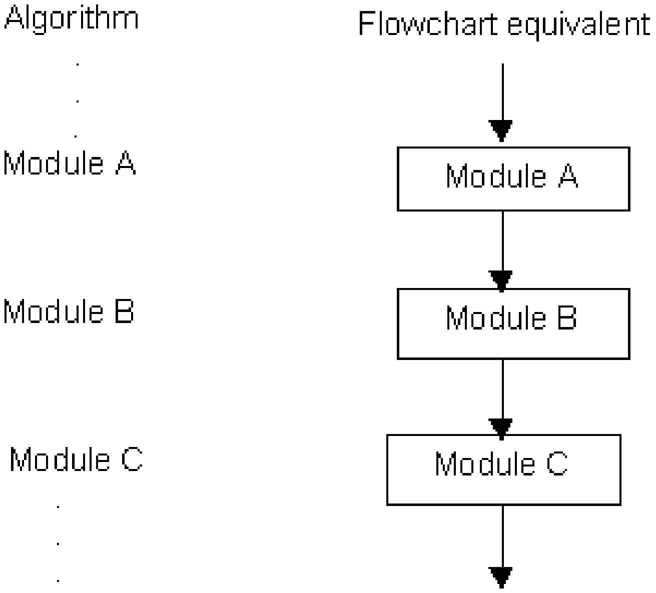
2.4 Control Structures

Algorithms and their equivalent computer programs are more easily understood if they mainly use self-contained modules and three types of logic, or flow of control, called

1) Sequence logic, or sequential flow
2) Selection logic, or conditional flow
3) Iteration logic, or repetitive flow

2.4.1 Sequence Logic

The sequence may be presented explicitly, by means of numbered steps, or implicitly, by the order in which the modules are written as shown in fig. 2.3



ence Logic

2.4.2 Selection Logic

Selection logic contains a number of conditions, which lead to a selection of one out of several alternative modules. The structures which implement this logic are called conditional structures or If structures. These conditional structures fall into three types.

(1) Single alternative: This structure has the form

```
If condition, then:
        [Module A]
[End of If structure.]
```

The logic of this structure is as shown in fig. 2.4. If the condition is true then it executes the module A, which may consist of one or more statements. Otherwise Module A is skipped and control transfers to the next step of the algorithm.

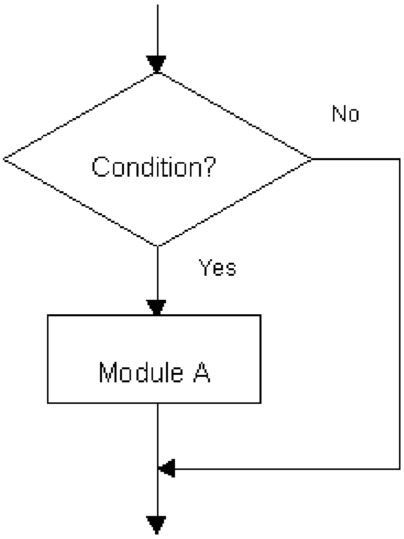


Fig. 2.4 Single Alternative

(2) Double alternative: This structure has the form

```
If condition, then:
        [Module A]
Else:
        [Module B]
[End of If structure.]
```

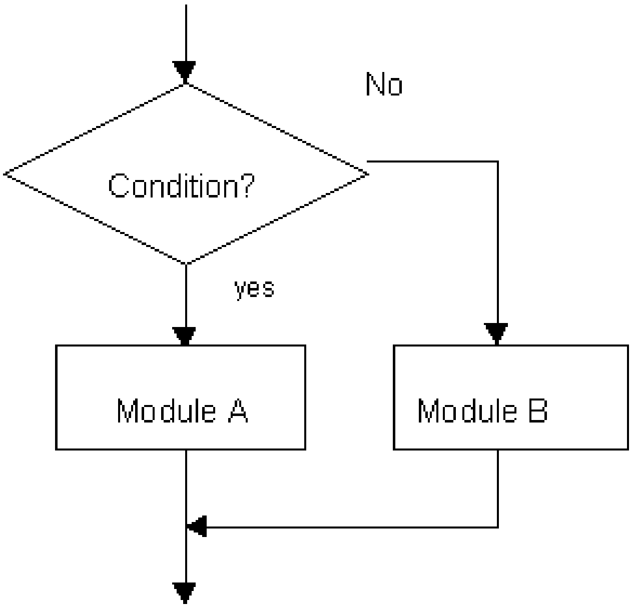


Fig 2.5 Double alternative

The logic of this structure is as shown in fig.2.5. As indicated by the flowchart, if the condition holds, then Module A is executed; otherwise B is executed.

(3) Multiple alternative: This structures has the form

```
If condition (1), then:
        [Module $A_1$]
Else If condition (2), then:
        [Module $A_2$]
Else If condition (M), then:
        [Module $A_M$]
Else:
        [Module B]
[End of If structure.]
```

The logic of this structure allows only one of the modules to be executed. Specifically, either the module that follows the first condition, which holds, is executed, or the module, which follows the final Else statement, is executed.

Example: The solution of the quadratic equation

$ax^2+bx+c=0$ where a ? 0, given by the quadratic formula

$$X = -b \pm \sqrt{\frac{b^2 - 4ac}{2a}}$$

The quantity $D=b^2 - 4ac$ is called the discriminate of the equation.

If D is negative, then there are no real solutions.

If D=0, then there is only one real solution, x= -b/2a.

If D is positive, the formula gives the two distinct real solutions. The following algorithm finds the solutions of a quadratic equation.

Algorithm 2.2: (Quadratic Equation.) This algorithm inputs the coefficients a, b, c of a quadratic equation and outputs the real solutions.

```
1. Read a, b, c
2. Set D:=B2-4AC
3. If D > 0, then:
    a) Set  x1:=(-b + d) /2a and x2:=(-b – d) /2a
    b) Write: x1, x2
Else if d=0, then:
    a) Set x:=-b/2a.
    b) Write: ‘unique solution’, x
```

```
Else:
        Write: 'no real solutions'
    [End Of If Structure.]
4. Exit
```

2.4.3 Iteration Logic

The third kind of logic refers to either of two types of structures involving loops. Each type begins with a Repeat statement and is followed by a module, called the *body of the loop*.

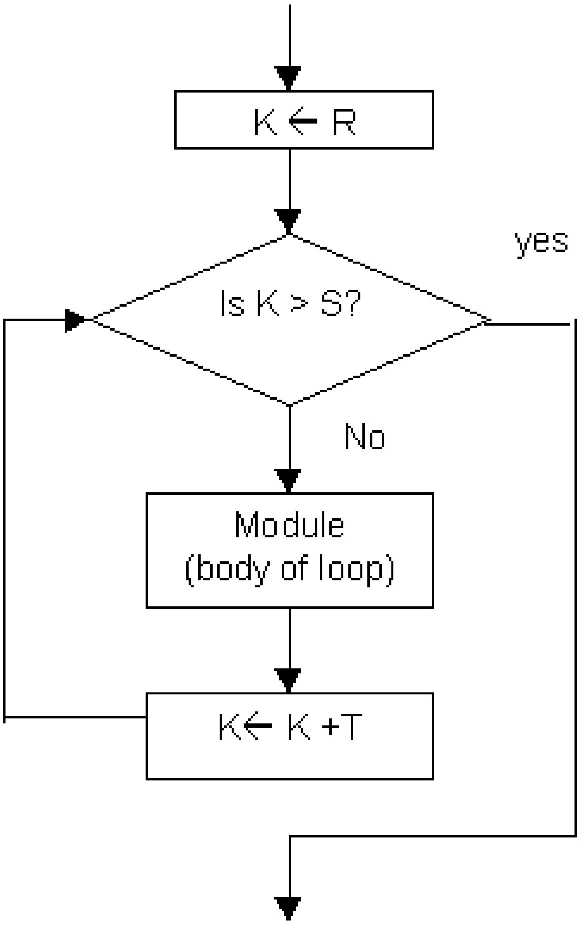
The **repeat-for-loop uses** an index variable k, to control the loop. The loop will usually have the form:

```
Repeat for k = R to S by T:
        [Module]
[End of loop.]
```

The logic of this structure is as shown in fig. 2.6

Fig. 2.6 Repeat-For structure

Fig. 2.6 Repeat-For structure



Here R is called the *initial value*, S the *end value* or *test value*, and T is the *increment*. Observe that the body of the loop is executed first with K=R, then with K=R+T, then K=R+2T, and so on. The cycling ends when K>S.

The **repeat - while** loop uses a condition to control the loop. The loop will usually have the form

```
Repeat while condition:
        [Module]
[End of loop.]
```

The logic of this structure is as shown in fig. 2.7. Here the cycling continues until the condition is false.

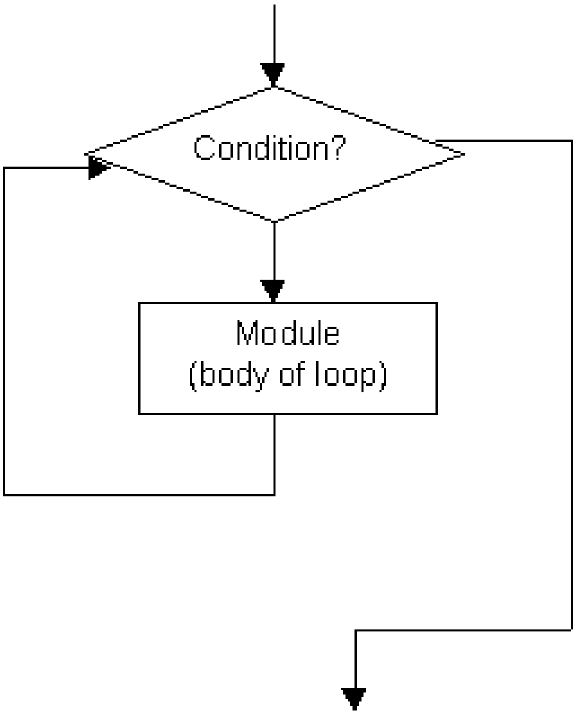


Fig. 2.7 Repeat-While structure

Example:

Algorithm 2.1 is rewritten using a repeat–while

Algorithm 2.3: (Largest Element in Array) Given a nonempty array DATA with N numerical values, this algorithm finds the location LOC and the value MAX of the largest element of DATA.

```
1.  [Initialize.] Set K:=1,LOC:=1 and MAX:=DATA[1]
2.  Repeat Steps 3 and 4 while K<=N:
3.      If MAX<DATA[K], then:
            Set LOC:=K and MAX:=DATA[K]
        [End of If structure.]
```

```
4. Set k:=k+1;
      [End of Step 2 loop.]
5. Write: LOC, MAX
6. Exit
```

2.5 Complexity of Algorithms

The analysis of algorithms is a major task in computer science. Inorder to compare algorithm, we must have some criteria to measure the efficiency of our algorithms. This section discusses this important topic.

The **complexity** of an algorithm M is the function f(n) which gives the running and /or storage space requirement of the algorithm in terms of the size n of the input data. The storage space required by an algorithm is a multiple of the data size n. Accordingly, the term “complexity” refer to the running time of the algorithm.

- **Measures of Times**

  - The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.
  - The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.
  - The *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

The **space** is measured by counting the maximum of memory needed by the algorithm.

The following example illustrates that the function f(n), which gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data.

Example: Linear Search

Suppose a linear array Data contains n elements, and a specific ITEM of information is given. We want to find the location LOC of ITEM in the array DATA. If the ITEM is not found in DATA to send some message, such as LOC=0, to indicate that ITEM does not appear in DATA.

The linear search algorithm solves this problem by comparing ITEM, one by one, with each element in DATA. That is, compare ITEM with DATA [1], then DATA [2], and so on, until we find LOC such that ITEM = DATA [LOC]. The Algorithm is as follows:

Algorithm 2.4: (Linear Search) A linear array DATA with n elements, and suppose a specific ITEM of information is given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC=0.

```
 1. [Initialize] Set K:=1 and LOC:=0
2. Repeat Step 3 and 4 while LOC=0 and K<=N
3.        If ITEM = DATA[K], then:Set LOC :=K
4.                 Set K:=K+1. [Increments counter.]
```

```
          [End of Step 2 loop.]
5.        If LOC=0, then:
                    Write: ITEM is not in the DATA
          Else:
                    Write: LOC id the location of ITEM
          [End of IF structure.]
6. Exit
```

The complexity of the search algorithm is given by the number C of comparisons between ITEM and DATA[K]. We seek C(n) for the worst case and the average case.

Worst Case

The worst case occurs when ITEM is the last element in the array DATA or is not there at all. In either situation, we have

$$C(n)=n$$

Accordingly, C(n)=n is the worst-case complexity of the linear search algorithm.

Average Case

Assume that ITEM does appear in DATA, and that it is equally to occur at any position in the array. Accordingly, the number of comparisons can be any of the numbers 1, 2, 3...n and each number occurs with probability p=1/n.

Then

$$C(n) = 1.1/n + 2.1/n +...+ n.1/n$$

$$= (1 + 2 +...+ n)\ 1/n$$

$$= n(n+1)/2.1/n = n+1/2$$

The average number of comparisons needed to find the location of ITEM is approximately equal to half the number of elements in the DATA list.

2.6 Subalgorithms

A *subalgorithm* is a complete and independently defined algorithmic module, which is used (or invoked or called) by some main algorithm or by some other subalgorithm.

A subalgorithm receives values called arguments, from an originating (calling) algorithm. It performs computations and then sends back the result to the calling algorithm. The subalgorithm is defined independently so that any different algorithms may call it or called at different times in the same algorithm. The relationship between an algorithm and a subalgorithm is similar to the relationship between a main program and a subprogram in a programming language.

The main difference between the format of a subalgorithm and that of an algorithm is that the subalgorithm will usually have a heading of the form

NAME (PAR_1, PAR_2... PAR_k)

Here Name refers to the name of the subalgorithm. The parameters PAR_1, PAR_2 ... PAR_k refer are used to transmit data between the subalgorithm and the calling algorithm.

Another difference is that the subalgorithm will have a return statement rather than exit statement. That means control is transferred back to the calling program when the execution of the subprogram is completed.

Subalgoritms fall into two basic categories:

- Function subalgorithms
- Procedure subalgorithms

One major difference between the subalgorithms is that the function subalgorithm returns only single value to the calling algorithm, whereas the procedure subalgorithm may send back more than one value.

Example:

The following function subalgorithm MEAN finds the average AVE of three numbers A, B and C.

Function 2.5: MEAN (A, B, C)

```
1. Set AVE:= ( A + B + C) / 3
2. Return (AVE)
```

MEAN is the name of the subalgorithm and A, B and C are the parameters. The Return statement includes the variable AVE in parenthesis, whose value is returned to the calling program.

An algorithm invokes the subalgorithm MEAN in the same way as a function; subprogram is invoked by a calling program. For example, suppose an algorithm contains the statement

Set TEST:=MEAN (T_1, T_2, T_3)

Where T_1, T_2 and T_3 are test scores. The argument values T_1, T_2 and T_3 are transferred to the parameters A, B, C in the subalgorithm, the subalgorithm MEAN is executed, and then the value of AVE is returned to the program and replaces MEAN (T_1, T_2, T_3) in the statement. Hence the average of T_1, T_2 and T_3 is assigned to TEST.

Example:

The following procedure SWITCH interchanges the values of AAA and BBB.

Procedure 2.6: SWITCH (AAA, BBB)

```
1. Set TEMP:=AAA, AAA:=BBB and BBB:=TEMP
2. Return
```

The procedure is invoked by means of a call statement. For example, the call statement,

Call SWITCH (BEG, AUX)

When the procedure SWITCH is invoked the argument of BEG and AUX are transferred to the parameters AAA and BBB respectively. The procedure is executed, which interchanges the values of AAA and BBB, and then the new values of AAA and BBB are transferred back to BEG and AUX, respectively.

2.7 Variables, Data types

Each variable in any of our algorithms or programs has a data type, which determines the code that is used for sorting its value. Four such data types follow:

1. Character: Here data are coded using some character code such as EBCDIC or ASCII. The 8-bit EBCDIC code of some characters is as shown in Fig. 2.7. A single character is normally stored in a byte.
2. Integer (or fixed point): Here positive integers are coded using binary representation, and negative integers by some binary variation such as 2's complement.
3. Real (or floating point): Here numerical data are coded using the exponential form of the data.
4. Logical: Here the variable can have only the value true or false; Hence it may be coded using only one bit, 1 for true and 0 for false.

Built-in data type

With every programming language, there is a set of data types called built-in data types. For example in **c** the data types that are available as built-in are listed below:

int, float, char, double, enum etc.

Example:

Suppose a 32-bit memory location X contains the following sequence of bits:

01101100 11000111 11010110 01101100

There is no way to know the content of the cell unless the data type of x is known.

1. Suppose X is declared to be of character type and EBCDIC is used. Then the four characters %GO% are stored in X.
2. Suppose X is declared to be of some other type, such as integer or real. Then an integer or real number is stored in X.

Local and Global variables

Normally, each program module contains its own list of variables, called *local variables*, which can be accessed only by the given program module. Subprogram modules may contain parameters, variables that transfer data between a subprogram and its calling program.

2.8 SUMMARY

In this chapter we have seen an algorithmic notation and some mathematical functions. We have discussed different types of control structures. Those are sequence logic, selection logic and iteration logic. In sequence logic steps are executed serially. In selection logic depending on the condition it follows the next statements. In iteration logic until the condition fails steps are executed repeatedly. It also describes how to analyze the complexity of algorithms.

It also covers subalgorithms, which is called by some main algorithm or by some other subalgorithm

2.9 Model Questions

1. What are the different types of control structures?
2. Write about the complexity of an algorithm.
3. Describe the difference between local variables, parameters and global variables.
4. Write a short notes on subalgorithms.

2.10 References

- Theory and problems of data structures
  Schaum's outline series

- Data structures
  Horowitz Sahani

- K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science,
J.K.C College, GUNTUR.

Lesson 3

STRING PROCESSING

3.0 Objectives

- Learn about Strings and their Memory Representations
- Learn different operations on Strings
- Learn about Word Processing
- Learn different Pattern Matching Techniques

Structure of the Lesson:

3.1. Introduction to Strings

3.2. Memory Representation of Strings

3.3. Character Data Type

3.4. String Operations

3.5. Word Processing

3.6. Pattern Matching

3.6.1. First Pattern Matching Algorithm

3.6.2. Second Pattern Matching Algorithm

3.7. Summary

3.8. Technical Terms

3.9. Model Questions

3.10. References.

3.1 Introduction to Strings

The set of characters used by each programming language to communicate with the computer is called the character set. Generally this set includes three subsets of characters namely, Alphabets (both lower and upper case English alphabets), Digits (0 to 9), and Special characters (eg:- +,-,$,= etc).

String: It is a finite sequence of zero or more characters. The number of characters in it is called the *length of the string*. A string with zero characters is called an *empty string* or a *null string*. In programming languages a string is delimited by quotation marks.

Examples:

1) ‘DATA STRUCTURES’
2) ‘ ‘
3) ‘❐❐’ (❐ — is symbol for blank space.)

3.2 Memory Representation of Strings

Strings are stored in a computer memory using three types of structures. They are

1) Fixed-length structures
2) Variable-length structures with fixed maximums.
3) Linked structures.

Fixed-length structures: This type of storage is also called Record-Oriented storage because; each line of print is viewed as a record, where all records have the same length. Generally each record consists of 80 characters. The FORTRAN and COBOL programming languages employ this kind of storage. The main advantage of this type of storage is that it is easy to access and update data from any given record. But this method has the following disadvantages:

1) Wastage of time in reading records with mostly blank space characters.
2) Difficult to represent records that require more space than available.
3) Corrections to a record, consisting more or fewer characters than the original text require entire record to be changed.

Example for fixed-length storage:

Input Data:-

```
C  PROGRAM PRINTING TWO INTEGERS IN INCREASING ORDER
        READ *, J, K
        IF(J .LE. K) THEN
                PRINT *, J, K
        ELSE
                PRINT *, K, J
        ENDIF
        STOP
END
```

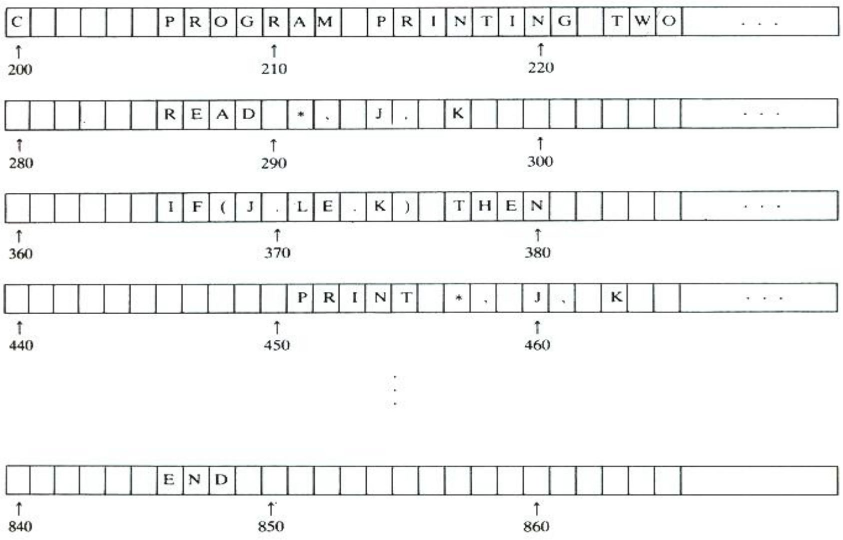


Fig. 3.1 Records stored sequentially in the computer.

In the above example it is difficult to insert a new record, as it requires all the succeeding records to be moved to new locations. Using a linear array called POINT, which gives the address of each successive record, can solve this problem. With this records need not be stored consecutively.

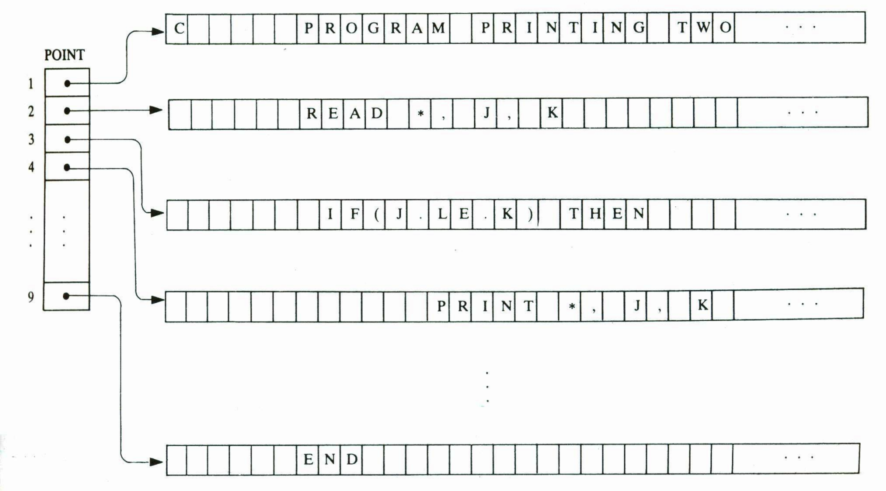
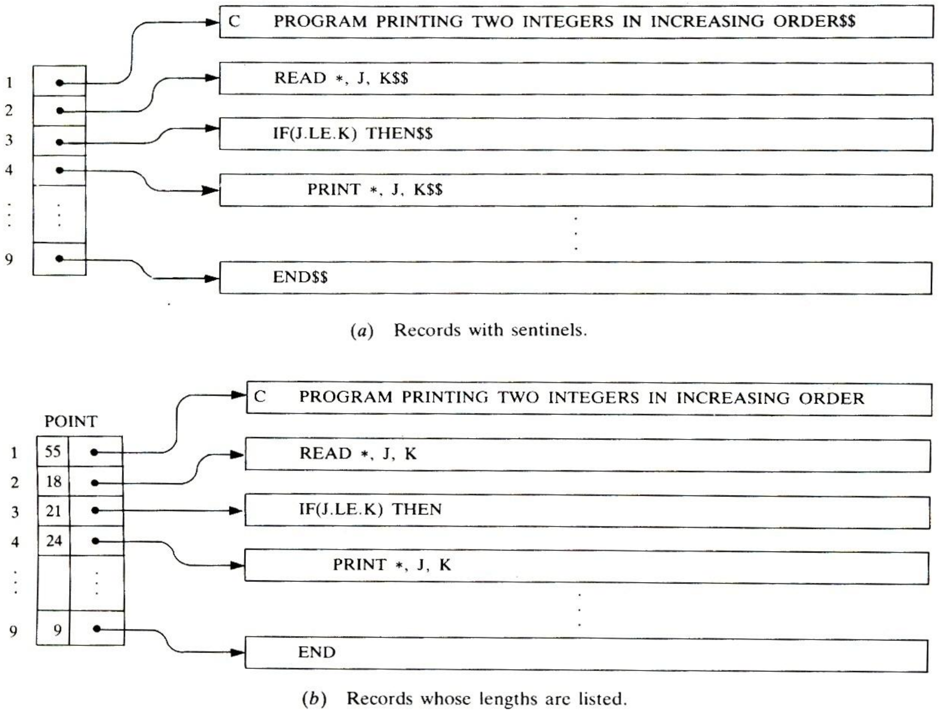


Fig.3.2 Reccrds stored using pointers.

Variable-length storage with fixed maximum: It is desirable to know the length of a string for certain string operations. So including the length of each record or a delimiter symbol along with it is useful. In this representation a symbol like $$ is placed at the end of each record or length of each record is stored in the POINT array along with the address. This allows us to quickly retrieve string when it occupies only the beginning part of the memory location. Sometimes string may be stored one after another by using separation mark such as two dollar signs or by using the pointer array giving the location of the strings. This method is advantageous when the strings require very little modifications and is mostly used in permanent storage mediums



(*a*) Records with sentinels.

(*b*) Records whose lengths are listed.

Fig. 3.3

Linked Storage: Extensive word processing applications require modifying strings in several ways such as inserting and deleting characters, words, phrases and paragraphs etc. For these string operations the linked storage is more suitable than fixed-length storage of strings. In linked representation strings are stored using linked lists. A one-way list is an ordered sequence of memory cells, called nodes. Each node contains an *item* and an address of the next node called link. The *item* here is either a single character or a fixed number of characters. The link gives the address of the node containing the next character or group of characters. The following picture is a schematic diagram of a one-way list.

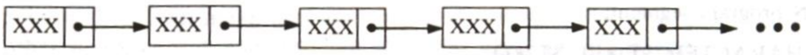
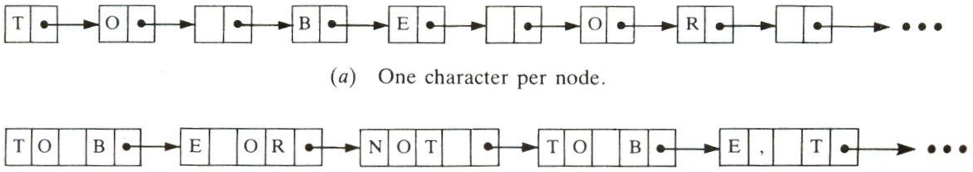


Fig.3.4 Linked list.

Example:



(*a*) One character per node.

(*b*) Four characters per node.

Fig. 3.5

3.3 Character Data Type

Most programming languages define a character data type. Usually each language has its own rules to form and manipulate character variables. Generally the character variables fall into three categories. They are:

1) Static — Its length is defined before program is executed.
2) Semi-static — Its length varies within a range during program execution.
3) Dynamic — Its length can change during program execution.

Examples: The FORTRAN programming language uses static character variables. The BASIC programming language defines semi-static character variables whose name ends with a dollar sign. These variables may change their length but cannot exceed a fixed bound. BASIC uses double quotation marks to denote string constants. In Pascal, a character variable can represent only a single character, and hence a string is represented by a linear array of characters. (an array is a linear sequence of items stored consecutively.)

3.4 String Operations

The most common set of operations on strings include finding the length, concatenating two strings, finding a substring of a string. The length of a string is the total number of characters in the string. The blank space character (?) also contributes to the length of a string. Joining two strings end to end gives the concatenation of the strings. It is denoted by the symbol //. Let S1 and S2 are two strings, then their concatenation is denoted by S1//S2 and it is the string consisting of the characters of S1 followed by the characters of S2.

Examples:

1) S1= ‘DATA’ and S2= ‘STRUCTURES’ then S3=S1//S2=’DATASTRUCTRES’
2) S1=’DATA’, S2=’ ?’, and S3=’STRUCTURES’ then S4=S1//S2//S3=’DATA ?STRUCTURES’

Substring: In the above example S1 and S3 are called substrings of S4. A string Y is called a substring of a string S if there exist strings X and Z such that S = X//Y//Z. If X is an empty string, then Y is called an *initial substring* of S. If Z is an empty string then, Y is called a *terminal substring* of S. From example 2 the string S1=’DATA’ is *initial substring* of S4, and the string S3=’STRUCTURES’ is *terminal substring* of S4. To access a substring from a given string the following information is required:

1) The name of the string or the string itself.
2) Position of the first character of the substring in the given string.
3) The length of the substring or the position of the last character of the substring.

The substring operation statement is written as:

SUBSTRING (string, initial, length)

Example: SUBSTRING(‘DATA STRUCUTRES’, 6,4) = ‘TRUC’

SUBSTRING(‘DATA STRUCTURES’, 5,6) = ‘❐STRUC’

Pattern Matching: Finding the position where a string pattern P first appears in a given string text T is known as pattern matching. If the pattern P does not appear in the text T, then this operation returns 0. This operation takes two arguments, which can be either string variables or constants. It is written as:

INDEX (text, pattern)

Example: Let T = 'DATA STRUCTURES' then

INDEX (T, 'AT') = 2

INDEX (T, ' STR') = 5

Length: It is the number of characters in a string. This operation requires one argument i.e., the string itself or its name. This operation is written as:

LENGTH (string).

Example: Let S1='DATA STRUCTURES' then LENGTH (S1) = 15.

Trim: The leading and trailing blank characters in a string can be truncated. This operation is called trim. Omitting the leading blank characters is called left trim and that of trailing blank characters is called right trim. This operation requires just the string itself or its name. It is written as:

TRIM (string).

Example: TRIM (' DATA ') = 'DATA'

3.5 Word Processing

Using a computer to create, edit and print documents is called word processing. An application that enables us to perform these operations is called a word processor program. Word processors usually allow the following operations:

Insert, delete, replace, cut, copy, paste, word wrap, and print text.

Insertion: To insert a string S into a text T at a position K, we write the operation as INSERT (text, position, string). The INSERT function can be implemented using SUBSTRING () and concatenation operations as:

INSERT (T, K, S) = SUBSTRING (T, 1, K-1) // S // SUBSTRING (T, K, LENGTH(T)-K+1).

Example: INSERT ('DATA STRUCTURES', 1, 'COMPLEX ') = 'COMPLEX DATA STRUCTURES'

Deletion: To remove characters from a text T, a position should be identified starting from which a number of characters are deleted. This operation is denoted by DELETE (text, position, length). The delete function can be implemented using the string operations as given below:

DELETE (T, K, L) =

SUBSTRING (T, 1, K-1) // SUBSTRING (T, K+L, LENGTH (T)-K-L+1)

A pattern P can be deleted from text T by calling the delete function as:

DELETE (T, INDEX (T,P), LENGTH(P)) This function call removes the first occurrence of the pattern P in the text T. If the pattern P does not occur in T the text T remains unchanged.

Examples: DELETE ('DATA STRUCTURES', 2,3) = 'D STRUCTURES'

DELETE (‘DATA STRUCTURES’, 5,1) = ‘DATASTRUCTURES’

The following algorithm can be used to remove every occurrence of a pattern P from the text T.

Algorithm:

```
Step 1: [Find index of P.] Set k:=index(T,P).
Step 2: Repeat while K $\neq$ 0:
                a) [Delete P from T.]
Set T := DELETE(T, INDEX (T,P), LENGTH (P)).
                b) [Update index.] Set K := INDEX (T,P).
[End of loop.]
Step 3: Write T.
Step 4: Exit.
```

Replacement: To replace a pattern P1 by a pattern P2 we write:

REPLACE(text, P1, P2).

Examples: REPLACE (‘COLLEGE’,’L’, ‘T’) = ‘COTTEGE’

REPLACE (‘POCKET’,’P’,’R’) = ‘ROCKET’

REPLACE (‘DATA’, ‘AD’,’BA’) = ‘DATA’ in this example the pattern ‘AD’ does not occur in the given text ‘DATA’, so the text remains unchanged.

Algorithm to replace every occurrence of a pattern P by a pattern Q in a text T is given below:

Algorithm:

```
Step 1: [Find index of P.] Set K := INDEX ( T, P).
Step 2: Repeat while K $\neq$ 0:
          [Replace P by Q.] Set T:=REPLACE ( T, P, Q).
          [Update index.] Set K :=INDEX (T, P)
     [End of loop.]
Step 3: Write: T.
Step 4: Exit.
```

Observation: This algorithm may not terminate in some cases where P is a sub-string of Q

3.6 Pattern Matching

Searching for a string pattern P in a text T is called pattern matching. Generally the length of P is less than or equal to the length of T. In pattern matching algorithms characters are denoted by lower case and exponents are used to denote repetition. For example a^2b^3 means aabbb.

First Pattern Matching Algorithm:

This algorithm proceeds by comparing P with each of the substrings of T starting from left to right, until a match is found. Let W_k=SUBSTRING(T,K,LENGTH(P)). First P is compared, character-by -

character, with the first substring W_1. If they match, the index of P in T is 1. Next P is compared with W_2 and so on until all W_ks are exhausted.

```
First Pattern matching algorithm:
Step1. Set K:=1 and MAX:= S-R+1.
Setp2. Repeat steps 3 to 5 while K ≤ MAX
Step3. Repeat for L = 1 to R
            IF(P[L] ≠ T[K+L-1] then: Goto step 5.
Step4. Set INDEX = K, and Exit.
Setp5. set K := K+1
Step6. set INDEX = 0;
Step7. Exit.
```

The complexity of this pattern-matching algorithm is measured in number of comparisons between characters of P and W_k. In average case the complexity is $O(n^2)$ where n is the sum of length of P and length of T.

Second Pattern Matching Algorithm:

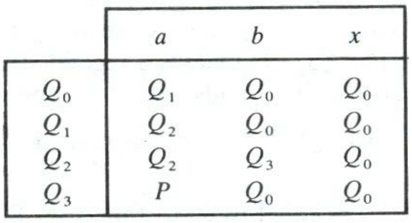
The Second Pattern Matching algorithm uses a table, which is derived from a particular pattern P but is independent of the text T. The preparation of such a table is explained with the following example. Let P = aaba. Suppose T = $T_1T_2T_3$…, where T_1 denotes the I^{th} character of T; and suppose the first two characters of T match those of P; i.e., suppose T = aa… then T has one of the following three forms: (i) T=aab… (ii) T = aaa… (iii) T = aax… , where x is any character different from a or b. Suppose we read T_3 and find that T_3=b. Then we next read T_4 to see if T_4= a, which will give a match of P with W_1. On the other hand, suppose T_3 = a. Then we know that $P \neq W_1$; but we also know that W_2= aa…..,i.e., that the first two characters of the substring W_2 match those of P. Hence we next read T_4 to see if T_4=b. Last, suppose T_3 =x. Then we know that $P \neq W_1$ but we also know that $P \neq W_2$ and $P \neq W_3$, since x does not appear in P. Hence we next read T_4 to see if T_4=a, i.e., to see if the first character of W_4 matches the first character of P.

There are two important points to the above procedure. First, when we read T_3, we need only compare T_3 with those characters, which appear in P. If none of these match, then we are in the last case, of a character *x* which does not appear in P. Second, after reading and checking T_3, we next read T_4; we do not have to go back again in the text T

In the above picture (a) contains the table that is used in our second pattern matching algorithm for the pattern P = *aaba.* (In both the table and the accompanying graph, the pattern P and its substrings Q will be represented by italic capital letters.) The table is obtained as follows. First of all, we let Q, denote the initial substrings of *P* of length *I*; hence

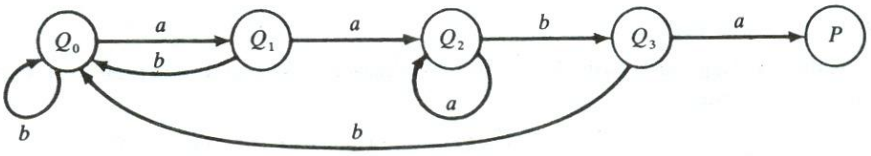
$$Q_0 = \wedge, \qquad Q_1 = a, \qquad Q_2 = a^2, \qquad Q_3 = a^2b, \qquad Q_4 = a^2ba = P$$

(Here $Q_0 = \wedge$ is the empty string.). These initial substrings of P, excluding P itself, label the rows of the table. The columns of the table are labeled *a, b* and *x,* where *x* represents any character that doesn't appear in the pattern *P*. Let *f* be the function determined by the table; i.e., let $f(Q_i, t)$ denote the entry in the table in row Q_i and column *t* (where *t* is any character). This entry $f(Q_i, t)$ is defined to be the largest *Q* that appears as a terminal substring in the string Q_it, the concatenation of Q_i



	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

(a) Pattern matching table.



(b) Pattern matching graph.

Fig. 3.6

and *t*. For example,

a^2 is the largest Q that is a terminal substring of $Q_2a = a^3$, so $f(Q_2, a) = Q_2$

$\wedge$ is the largest Q that is a terminal substring of $Q_2b = ab$, so $f(Q_1, b) = Q_0$

a is the largest Q that is a terminal substring of $Q_0a = a$, so $f(Q_0, a) = Q_1$

$\wedge$ is the largest Q that is a terminal substring of $Q_3x = a^2bx$, so $f(Q_3, x) = Q_0$ ....

and so on. Although $Q_1 = a$ is a terminal substring of $Q_2a = a^3$, we have $f(Q_2, a) = Q_2$, because Q_2, is also a terminal substring of $Q_2a = a^3$ and Q_2 is larger than Q_1. We note that $f(Q_i, x) = Q_0$ for any Q, since x does not appear in the pattern P. Accordingly the column corresponding to x is usually omitted from the table.

Our table can also be pictured by the labeled directed graph in the above picture. The graph is obtained as follows. First, there is a node in the graph corresponding to each initial substring Q_i of P. The Q's are called the *states* of the system, and Q_0 is called the *initial* state. Second, there is an arrow (a directed edge) in the graph corresponding to each entry in the table. Specifically, if

$$f(Q_i, t) = Q_j$$

then there is an arrow labeled by the character *t* from Q_i to Q_j. For example, $f(Q_2, b) = Q_3$, so there is an arrow labeled *b* from Q_2 to Q_3. For notational convenience, we have omitted all arrows labeled *x*, which must lead to the initial state Q_0.

We are now ready to give the second pattern matching algorithm for the pattern P=aaba. Let T = T_1, T_2, T_3, …, T_N denote the *n*-character –string text which is searched for the pattern P. Beginning with the intial state Q_0 and using the text T, we will obtain a sequence of states S_1, S_2 ,S_3 ,…. , as follows. We let $S_1 = Q_0$ and we read the first character T_1. From either the table or the graph in Fig 3.8, the pair (S_1,T_1) yields a second state S_2; that is , $F(S_1,T_1) = S_2$. We read the next character T_2.The pair (S_2, T_2) yields a state S_3, and so on. There are two possibilities

(1). Some state S_k= P, is the desired pattern. In this case, P does appear in T and its index is K- LENGTH(P).

(2) No state S_1, S_2, S_3 ,….,S_{N+1} is equal to P. In this case, P does not appear in T.

We illustrate the algorithm with two different texts using the pattern P = aaba.

Second Pattern Matching Algorithm: The pattern matching table $F(Q_1, T)$ of a pattern P is in memory, and the input is an N-character string T = T_1, $T_{2,}$ T_3 , …, T_N . This algorithm finds the INDEX of P in T.

```
1.  [Initialize.] Set K:= 1 and $S_1$ =$Q_0$.
2.  Repeat Steps 3 to 5 while $S_k \neq P$ and $K \leq N$.
3.          Read $T_k$.
4.          Set $S_{k+1}$ := F($S_k$, $T_k$).[Finds next state.]
5.          Set K:=K+1.[Updates counter.]
      [End of Step 2 loop.]
6.  [Successful?.]
    If $S_k$ =  P , then:
      INDEX = K-LENGTH (P).
    Else:
      INDEX =0.
    [End of If structure.]
7.  Exit.
```

3.7 Summary

String is a finite sequence of zero or more characters. The number of characters in it is called the *length of the string*. A string with zero characters is called an *empty string* or a *null string*. In programming languages a string is delimited by quotation marks. Strings are stored in a computer memory using three types of structures. They are, Fixed-length structures, Variable-length structures with fixed maximums, Linked structures.

Extensive word processing applications require modifying strings in several ways such as inserting and deleting characters, words, phrases and paragraphs etc. For these string operations the linked storage is more suitable than fixed-length storage of strings. In linked representation strings are stored using linked lists. Each node in the list contains an item and a link. The *item* is either a single character or a fixed number of characters. The link gives the address of the node containing the next character or group of characters.

The most common set of operations on strings include finding the length, concatenating two strings, finding a substring of a string. The length of a string is the total number of characters in the string. The blank space character (?) also contributes to the length of a string. Joining two strings end to end gives the concatenation of the strings.

Finding the position where a string pattern P first appears in a given string text T is known as pattern matching. If the pattern P does not appear in the text T, then this operation returns 0. This operation takes two arguments, which can be either string variables or constants. There are two popular algorithms for pattern matching. They are known as First Pattern Matching Algorithm and

Second Pattern Matching Algorithm. The first algorithm proceeds by comparing successive substrings of a given text with the desired pattern, and stops when the required pattern is found. The Second Pattern Matching algorithm uses a table, which is derived from a particular pattern P but is independent of the text T

3.8 Technical Terms

Character Set: The set of characters used by each programming language to communicate with the computer.

String: A finite sequence of zero or more characters.

Concatenation: Joining two strings end to end.

Word Processing: Using a computer to create, edit and print documents.

Pattern: A sequence of character. May be a word, phrase or a sentence.

Text: A set of strings.

Substring: A portion of a string.

3.9 Model Questions

1. What is Character Set?
2. Define String. Describe String Operations.
3. What is meant by Word Processing?
4. Explain different representations for Strings in memory.
5. Explain the second Pattern Matching Algorithm.
6. Compare first Pattern Matching Algorithm with Second Pattern Matching Algorithm.

3.10 References

1. Theory And Problems Of Data Structures

By SEYMOUR LIPSCHUTZ

2. Data Structures and Algorithms, Addison-Wesley

By Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft

- Y. VENKATESWARA RAO, M.C.A.,
Lecturer, Dept. Of Computer Science,
JKC College, GUNTUR-6

Lesson 4

Arrays Records and Pointers

4.0 Objectives

- Learn about Arrays and their Memory Representations
- Learn different operations on Arrays
- Learn about Records and their Memory Representations
- Learn different operations on Records
- Understand Multidimensional Arrays
- Learn about Sparse Matrices

Structure of the Lesson:

4.1 Introduction to Arrays

An array is a linear data structure in which a linear relationship is established between the elements by storing them sequentially. The elements of the array are referenced respectively by an index set consisting of n consecutive numbers. The number of elements in the array is referred to as its length or size. Let A be an array, then its elements are denoted as A1, A2, A3, … An. In general the length or the number of data elements of the array can be obtained by the formula Length = UB – LB +1. Where UB(upper bound) is the largest index and LB(lower bound) is the smallest index of the array. Programming languages use different notations for array elements. For example FORTRAN language denotes array elements as A(1), A(2) etc, where as Pascal and C family of languages use square brackets to contain index as A[1], A[2] etc. C -family of languages (C, C++, and Java) use index values starting from 0 (zero) to (n-1) where n is the size of array.

Example:

Let DATA be a 5-element linear array of integers such that

DATA [1] = 10

DATA [2] = 20

DATA [3] = 30

DATA [4] = 40

DATA [5] = 50

Such an array is generally pictured as

1	10
2	20
3	30
4	40
5	50

4.2 Representation of Linear Arrays in Memory

An array is stored in sequential locations in a computer's memory. The computer remembers the address of first element of an array as the base address and calculates the address of other elements of the array using the formula:

LOC (LA[K]) = Base (LA) + w (K – lower bound)

Where LA is the array name, w is the number of words per memory cell for the array LA. Given the index K it is possible to locate and access the content of LA[K] without scanning any other element of LA.

Example: Let AUTO be an array containing the number of automobiles sold each year from 1932 through 1984. For this array the index set can be started at 1932 rather than at 1. AUTO[K] = number of automobiles sold in the year K. Then LB = 1932 and UB = 1984, and the number of elements in array AUTO is: (UB-LB+1) = 55. Suppose the array AUTO appears in memory from location 200 as shown in the following picture, then LOC(AUTO[1932]) = 200, LOC (AUTO [1933])

= 204, LOC(AUTO[1934]) = 208 … as each element of the array occupies four memory words (w = 4).

LOC(AUTO[1965]) = Base(AUTO) + w (1965 – LB)

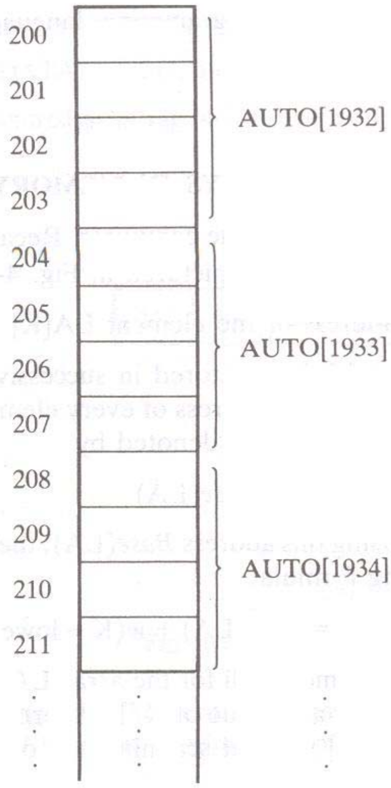
= 200 + 4 (1965 – 1932)

= 200 + 4 (33)

= 200 + 132 = 332.

4.3. Traversing a Linear Arrays

The elements of an array are processed, by traversing the array. While the array is being



can be applied on each element of the array. Calling such a be initialized. The following algorithm traverses a line array

```
K := LB.
hile K <= UB.
pply PROCESS( ) to LA[K].
```

```
Step 4:         [Increase counter.] Set K := K+1.
        [End of step 2 loop.]
Step 9 Exit.
```

4.4. Insertion and Deletion in Linear Array

Inserting an element into an array means adding another element to the collection. Deleting an element from an array means removing the element from the array. An element can be easily inserted at the 'end' of a linear array, provided the array is large enough to accommodate the new element. To insert an element in the middle, on an average half of the elements of the array must be relocated to accommodate the new element. Similarly deleting an element from the end of an array is a simple operation, where as deleting an element in the middle needs, on an average, half of the array elements to be relocated. The following algorithms allow us to insert and delete elements from an array.

Algorithm: (Inserting into a linear array) INSERT(LA, N, K, ITEM)

```
Step 1: [Initialize counter.] Set J:= N.
Step 2: Repeat Steps 3 and 4 while J $\geq$ K.
Step 3: [Move J^th element downward.] Set LA[J+1]:=LA[J].
Step 4: [Decrease counter.] Set J := J–1.
        [End of step 2 loop.]
Step 5: [Insert element.] Set LA[K]:= ITEM.
Step 6: [Reset N.] Set N := N+1.
Step 7: Exit.
```

The above algorithm inserts the element ITEM into the array LA at position K. The array LA has N elements. This algorithm works by first shifting the elements from index K to N one position down. Then placing the ITEM at position K in the array LA.

Algorithm: (Deleting from a Linear Array.) DELETE (LA, N, K, ITEM)

```
Step 1: Set ITEM := LA[K].
Step 2: Repeat for J = K to N–1:
        [Move J+1 element upward.] Set LA[J]:= LA[J+1].
        [End of loop.]
Step 3: [Reset the number N of elements in LA.] Set N := N–1.
Step 4: Exit.
```

The above algorithm deletes the Kth element from the array LA. It works by first taking the Kth element in ITEM and then moves up the K+1th element to Nth element.

4.5. Sorting and Searching Linear Array elements

Sorting means rearranging the elements of a linear list in ascending or descending order. There are a number of techniques to perform sorting. In this section we will learn one simple sorting method called Bubble Sort. Let A be a list of N numbers, referred as A[1], A[2], A[3], … A[N].

The Bubble sort algorithm to arrange elements of A in ascending order works as follows:

Step 1: Compare A[1] with A[2] and arrange them in desired order, so that A[1]<a[2]. Then compare A[2] with A[3] and arrange them in desired order, so that A[2]<A[3]. Continue until you compare A[N-1] with A[N] and arrange them so that A[N-1]<A[N]. This step involves N-1 comparisons and at the end of the step A[N] contains the largest element in the list.

Step 2: Repeat step1 with one less comparison; i.e. stop after comparing A[N-2] with A[N-1] as A[N] is already in sorted order.

Step 3: Repeat step1 with two fewer comparisons. I.e. stop after rearranging the pair (A[N-3], A[N-2]).

…………………………………………………….

…………………………………………………….

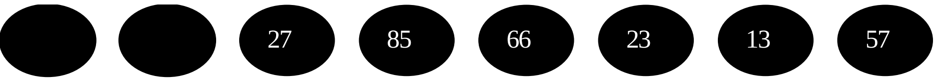
…………………………………………………….

Step N-1: Compare A[1] with A[2] and arrange them so that A[1]<A[2].

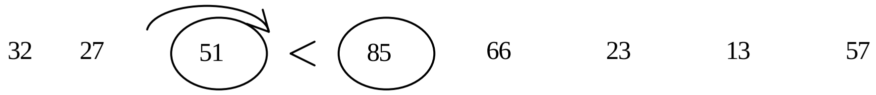
Example:
Consider the list of numbers 32, 51, 27, 85, 66, 23, 13, 57. The following list of steps show the first pass of the bubble sort algorithm applied on this list to arrange the elements in ascending order.

Pass 1:

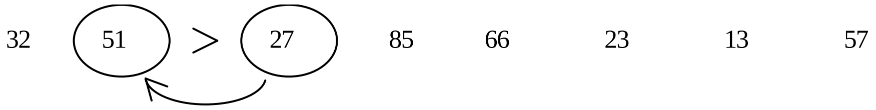
Step1: 32 < 51, so no need to interchange them.



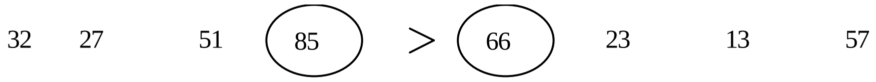
Step2: 51 > 27, interchange their positions.



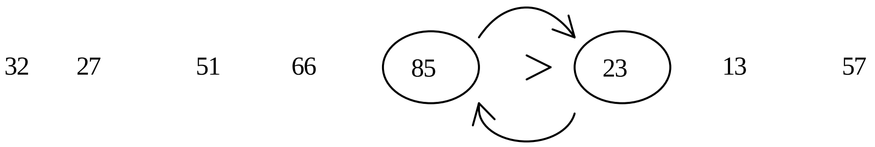
Step3: 51 < 85, no need to interchange.



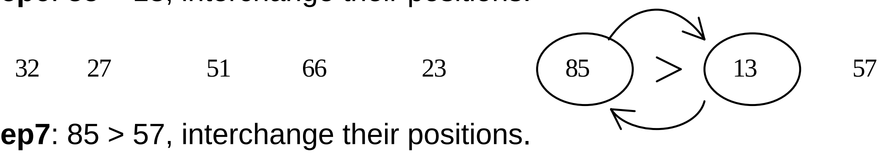
Step4: 85 > 66, interchange their positions.



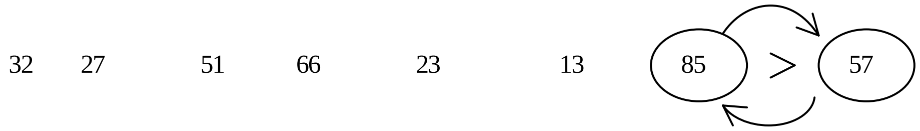
Step5: 85 > 23, interchange their positions.



Step6: 85 > 13, interchange their positions.



Step7: 85 > 57, interchange their positions.



After Step7 the list of numbers is: 32, 27, 51, 23, 13, 57, 85

Now it can be observed that 85 is the last element. I.e. after first pass the largest element 85 is pushed to the last position. The next pass requires only 5 comparisons. Pass3 requires only 4 comparisons and so on. Proceeding like this, the list requires seven passes to arrange the numbers in ascending order.

Algorithm for Bubble Sort

```
Step1: repeat steps 2 and 3 for K=1 to N-1.
Step2: Set PTR:=1
Step3: Repeat while PTR =N-K:
            a) If A[PTR]>A[PTR+1], then:
Interchange A[PTR] and A[PTR+1].
            b) Set PTR := PTR+1.
[End of step3 loop.]
[End of step1 loop.]
Step4: Exit.
```

Complexity of the Bubble sort is calculated by counting the number of comparisons (key operations). There are N-1 comparisons in first pass, N-2 in the second pass and so on. Hence the complexity

$$f(n) = (n-1) + (n-2)+ \ldots+2+1$$

$$= n(n-1)/2$$

$$= O(n^2).$$

Linear Search on arrays: The operation of finding location of an item in a list is called searching. The search is said to be successful if the item exits in the list otherwise it is unsuccessful. When there is no other information about the list of items, then a linear or sequential search is performed, by comparing each item in the list. This method traverses the list sequentially to locate item in the list, hence it is called linear or sequential search. The following algorithm performs linear search on a list A with N elements.

Algorithm:

```
Step1: [Insert ITEM at the end of A] Set A[N+1] := ITEM.
Step2: [Initialize counter.] Set LOC := 1.
Step3: [Search for ITEM.]
        Repeat while A[LOC] ? ITEM:
        Set LOC:=LOC+1.
        [End of loop.]
Step4: [Successful?] If LOC = (N+1), then: Set LOC:=0.
Step5: Exit.
```

Note: In algorithms, the sentences written in square brackets ([]) are comments, given for explaining the steps. They can be ignored however.

Binary Search: An efficient search method, called binary search, can be performed on a list if its elements are in sorted order. This method is also called dictionary search. The working of this method is as follows:

Locate the middle element of the list, and see if it is the desired *item*. If it is not the desired *item*, then the *item* may occur in the first half of the list if it is less than the middle element, otherwise it may occur in the second half of the list. Accordingly the procedure is repeated for the probable half of the list and so on until the *item* is found or it is confirmed that the *item* does not exist in the list. Complexity of Binary Search algorithm is $O(\log_2 n)$ where n is the number of items in the list. The binary search algorithm requires two conditions. They are: 1) the list must be sorted 2) The middle element must be accessible directly in any sublist.

Algorithm for Binary search

```
Step1: [Initialize variables.]
        Set BEG:=LB, END:=UB and MID = INT((BEG+END)/2).
Step2: Repeat steps 3 and 4 while BEG = END and A[MID] ?ITEM.
Step3:  If ITEM<A[MID], then:    Set END := MID-1.
        Else:   Set BEG := MID+1.
        [End of if structure.]
Step4:  Set MID := INT((BEG+END)/2).
        [End of step2 loop.]
Step5: If A[MID] = ITEM, then:  Set LOC := MID.
        Else: Set LOC:= NULL.
        [End of if structure.]
Step6: Exit.
```

4.6. Multi-Dimensional Arrays

Arrays with more than one dimension are called multi-dimensional arrays. In these arrays the elements are referenced by two or more subscripts.

A two-dimensional m x n array A is a collection of w data elements such that each element is specified by a pair of integers called subscripts, with the property that $1 \le J \le m$ and $1 \le K \le n$. where w is equal to the product of m and n.

The element of A with first subscript J and second subscript K will be denoted by A[J,K] are $A_{j,k}$

Representation of Two-dimensional Arrays

Let A be a two-dimensional M x N array. A is represented in memory by a block of (M.N) memory locations. Two-dimensional array may be represented in one of the two orders: 1) row-major 2) column-major. The computer calculates the address of the elements by using the following formulae.

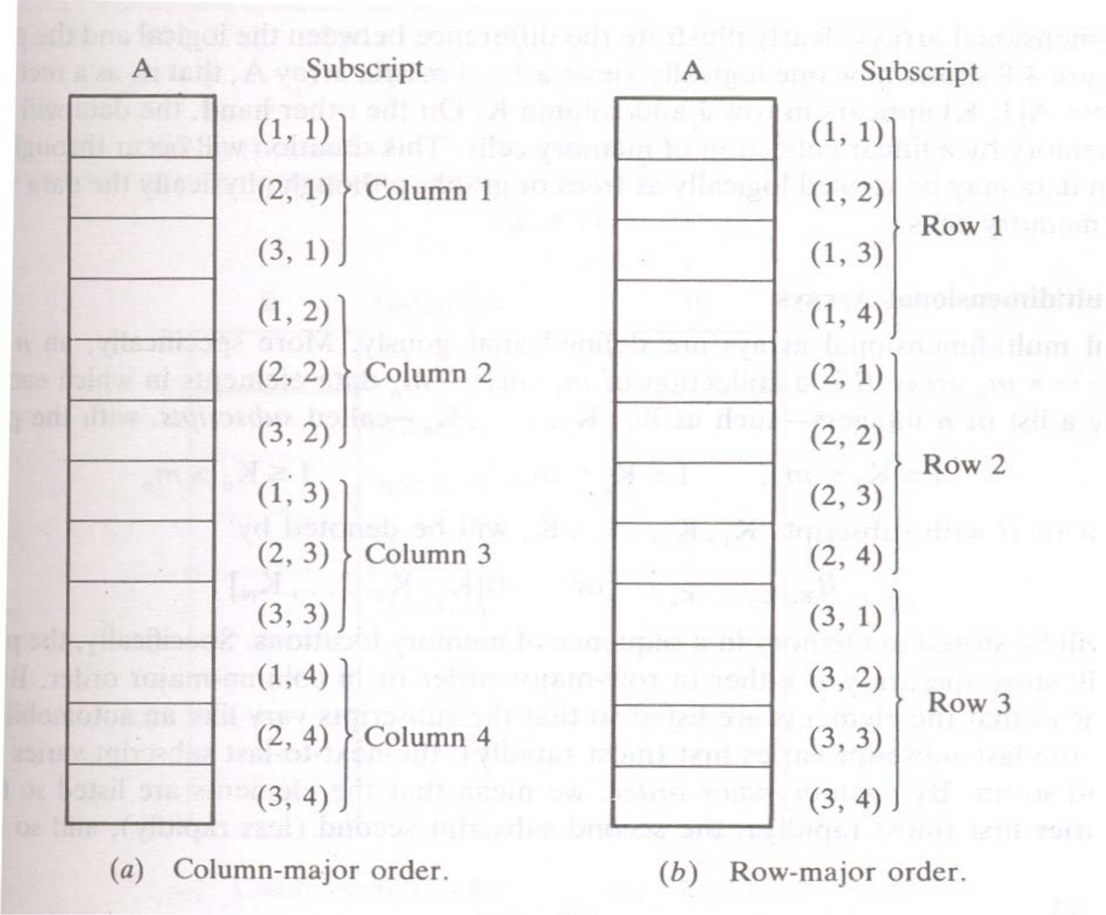
(column-major) LOC(A[J,K]) = Base(A) + w[M(K-1)+ (J-1)]

(row-major) LOC(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]

Base(A) is the address of the first element in two-dimensional array A.

w denotes the number of words per-memory location for the array A

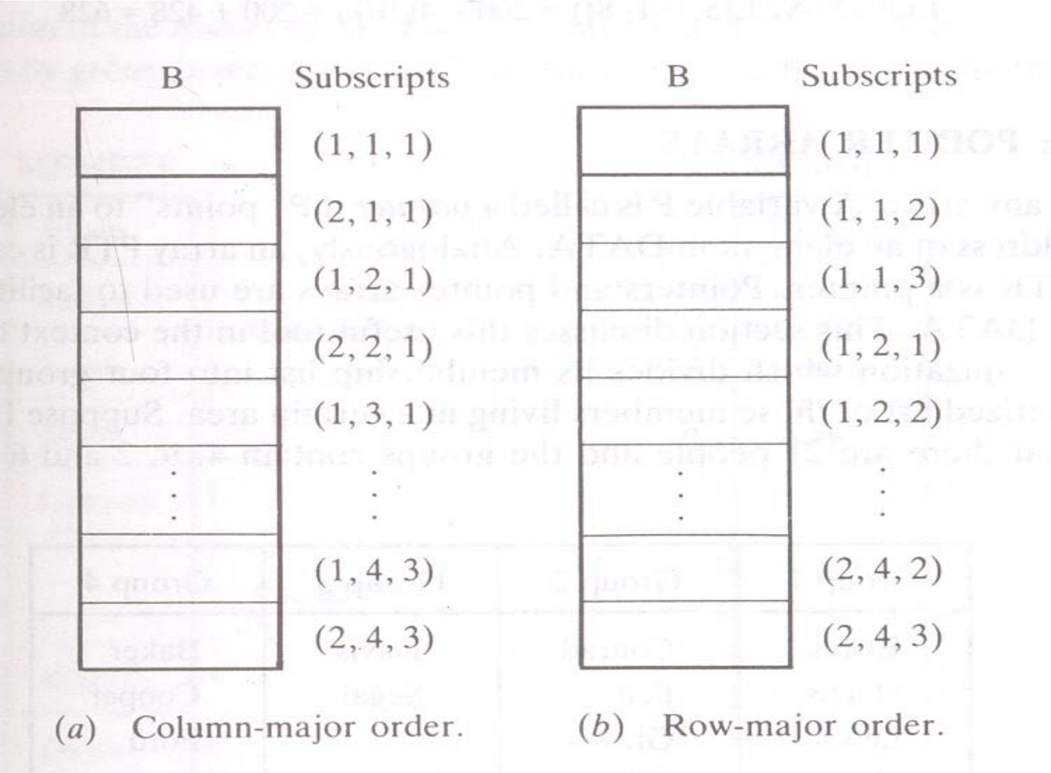
General Multidimensional Arrays: An n-dimensional m_1 x m_2 x m_3 x ….x m_n array B is a collection of $m_1.m_2.m_3….m_n$ data elements in which each element is specified by a list of n integers (such as k1,k2,k3,…kn) called subscripts with the property that $1 \le k_1 \le m_1$, $1 \le k_2 \le m_2$, … $1 \le k_n \le m_n$. The element of B with subscripts K_1, K_2, … Kn is denoted by $B_{k1,k2,k3,..kn}$ or $B[k_1,k_2,…k_n]$. The array will be stored in memory in a sequence of memory locations. Either row-major or a column-major notation is followed to represent the array. Memory representation of a 3-dimensional array is shown in the followingpicture



(*a*) Column-major order. (*b*) Row-major order.

4.7. Pointers

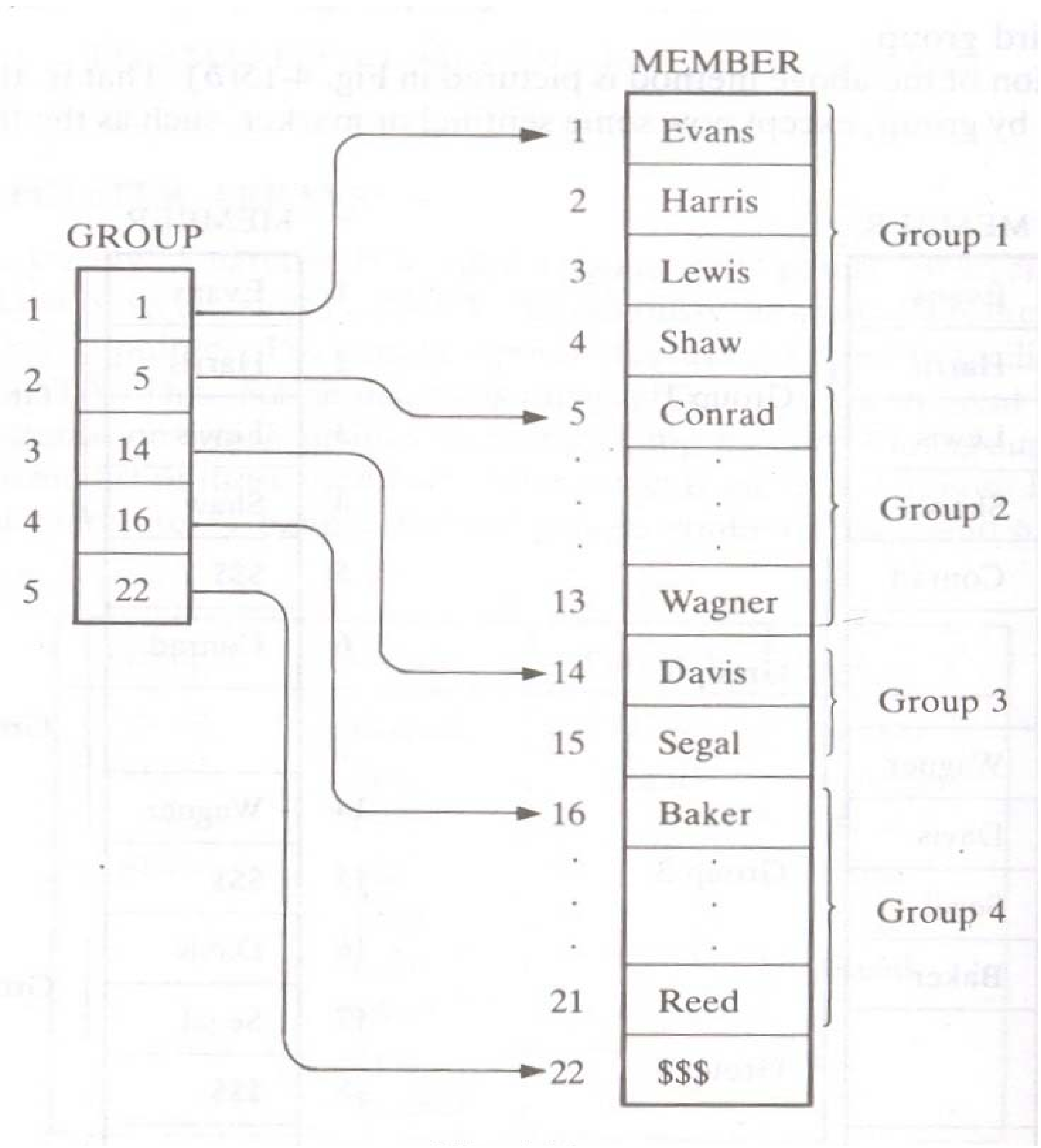
A pointer is a variable that points to another variable in memory. A pointer may point to an element in an array. Let DATA be an array in memory. If a pointer variable P contains address of an element of DATA then P is called pointer to that element. An array of such pointers is called pointer-array. Pointers and Pointer-arrays can be used to efficient processing of data. Consider an example of representing the membership names of an organization, divided into four groups. Four different lists need to be stored in memory. The lengths of these lists vary in size. Choosing a two-dimensional array for storing these names would lead to wastage of memory space. A better way is to store these lists of membership names in a one-dimensional array and remember the starting address of each list in an array of pointers. In this method we require a special sentinel symbol (like '$$$') to mark the end of lists or data. The following picture shows the memory representation for this example.



(*a*) Column-major order. (*b*) Row-major order.

4.8. Records

A record is a collection of related data items. Each item in a record is called a field or attribute. A file is a collection of similar records. A record may contain non-homogeneous data, i.e., the data items in a record may have different data types. The items in a record are indexed by attribute names not by subscripts. In this way a record differs from an array. Qualifying an item name with appropriate group item names in the structure, accesses that particular item of a record. This qualification is indicated by using decimal points(periods), to separate group items from subitems. The following example shows a record structure and different levels in it.



Eg:- Record for each newborn baby in a hospital:

1. Newborn
    2. Name
    2. Sex
    2. Birthday
        3. Month
        3. Day
        3. Year
    2. Father
        3. Name
        3. Age

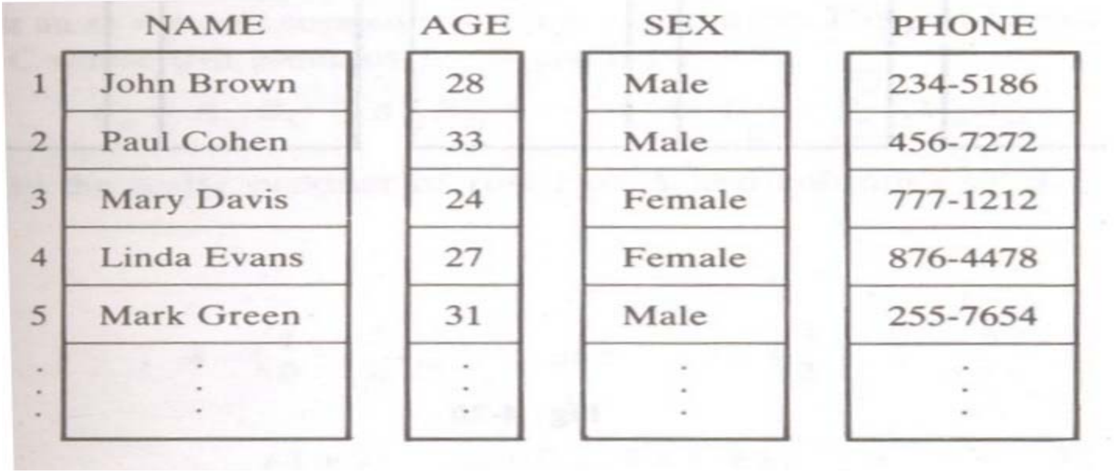
2. Mother
    3. Name
    3. Age

In this record structure the qualified name Newborn.Father.age refers to the age item of the 'Father' attribute in the Newborn record.

Representation of Records in Memory

Records may contain non-homogeneous data. So they cannot be stored in an array. Some programming languages provide hierarchical structures to represent records. However a file of records may be stored in memory as a collection of parallel arrays, where elements in different arrays with the same subscript belong to the same record. For example, consider a membership record with the following items: Name, Age, Sex, Phone. A file of these records may be represented in memory as in the following picture.

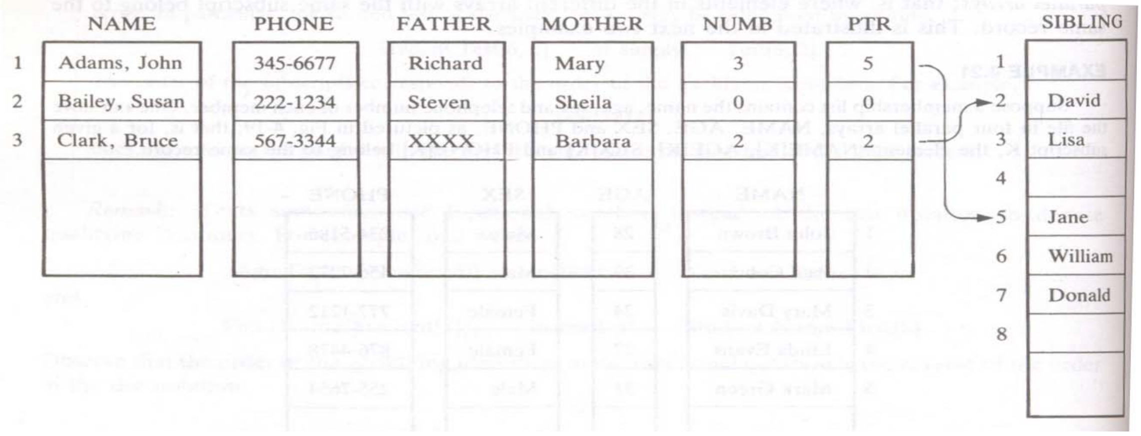
Records may be of variable length. Such records may be difficult to represent in memory using just parallel arrays. Additional arrays are needed to store the varying information. All records share the additional array(s). Each record contains a pointer into the extra array along with the length of the information pertaining to that record. For example consider a school maintaining the following records for its students. Each student record contains the following fields: Name, Telephone number, Father, Mother and siblings. Here the siblings information is of variable length. A student may have zero siblings or has more siblings. These varying length records can be represented in memory as shown in the following picture.



	NAME	AGE	SEX	PHONE
1	John Brown	28	Male	234-5186
2	Paul Cohen	33	Male	456-7272
3	Mary Davis	24	Female	777-1212
4	Linda Evans	27	Female	876-4478
5	Mark Green	31	Male	255-7654
⋮	⋮	⋮	⋮	⋮

4.9. Matrices and Sparse Matrices

In mathematics matrix refers to an arrangement of numbers into rows and columns. In computer science a two-dimensional array can be called a matrix. Each row in a matrix is called a Vector. A matrix with same number of rows and columns is called a n-square matrix. Algebraic operations on matrices include addition, subtraction and multiplication of two matrices. For addition and subtraction the two matrices should have the same dimension. For multiplication the number of columns of the first matrix must be equal to the number of rows in the second matrix. Suppose A be an (m x p) matrix and B is a (p x n) matrix. The product of A and B, (in that order) written as AB, is the (m x n) matrix C whose ijth element is given by $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{ip}B_{pj}$. An algorithm for



	NAME	PHONE	FATHER	MOTHER	NUMB	PTR
1	Adams, John	345-6677	Richard	Mary	3	5
2	Bailey, Susan	222-1234	Steven	Sheila	0	0
3	Clark, Bruce	567-3344	XXXX	Barbara	2	2

	SIBLING
1	
2	David
3	Lisa
4	
5	Jane
6	William
7	Donald
8	

multiplying matrices is given below.

Algorithm: MATMUL(A, B, C, M, P, N)

This algorithm computes C = AB. Where A is M x P matrix, B is P x N matrix and C is M x N matrix.

```
Step1. Repeat steps 2 to 4 for I = 1 to M:
Step2.     Repeat steps 3 and 4 for J = 1 to N:
Step3          set C[I,J] := 0
Step4.         Repeat for K = 1 to P:
Step5.             C[I,J] := C[I,J] + A[I,K]*B[K,J].
               [End of step 4 loop.]
           [End of step2 loop.]
       [End of step1 loop.]
Step6. Exit.
```

Sparse Matrices

Matrices with a relatively high proportion of zero entries are called sparse matrices. Two examples of sparse matrices are shown in the following picture

In a sparse matrix if the non-zero elements occur along the main diagonal, then it is called a diagonal sparse matrix. If the non-zero elements of a sparse matrix are below the main diagonal then it is called a (lower) triangular matrix. A sparse matrix can be represented as a general two-dimensional array in memory. But it is unnecessary to store zero elements in a matrix. So an alternative method, in which only non-zero elements of the matrix are considered, is adapted to store sparse matrices in memory. In this method a sparse matrix is stored as a three column matrix. In each row, the third column contains a non-zero element of the sparse matrix. The first and second columns contain the row and column index values of the element in third column. The first row in this representation usually contains the total number of rows, total number of columns and the total number of non-zero elements in the sparse matrix. The following table shows a sample sparse matrix representation in memory. This table represents a 5 x 5 sparse matrix, which has a total of 10 non-zero elements in it.

Row	Col	Element
5	5	10
1	3	40
…	…	…

4.10. Summary

An array is a linear data structure in which a linear relationship is established between the

array are referenced respectively by an

r of elements in the array is referred to

denoted as A1, A2, A3, … An. In general

be obtained by the formula

nemory. The computer remembers the

s and calculates the address of other

ase (LA) + w (K – lower bound)

memory cell for the array LA. Given the

[K] without scanning any other element

$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix}$$

(a) Triangular matrix.

$$\begin{pmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & -7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(b) Tridiagonal matrix.

Inserting an element into an array means adding another element to the collection. Deleting an element from an array means removing the element from the array. An element can be easily inserted at the ‘end’ of a linear array, provided the array is large enough to accommodate the new element. To insert an element in the middle, on an average half of the elements of the array must be relocated to accommodate the new element. Similarly deleting an element from the end of an array is a simple operation, where as deleting an element in the middle needs, on an average, half of the array elements to be relocated.

Sorting means rearranging the elements of a linear list in ascending or descending order. There are a number of techniques to perform sorting. In this section we will learn one simple sorting method called Bubble Sort. Let A be a list of N numbers, referred as A[1], A[2], A[3], … A[N].

Binary Search means, locate the middle element of the list, and see if it is the desired item. If it is not the desired item, then the item may occur in the first half of the list if it is less than the middle element, otherwise it may occur in the second half of the list. Accordingly the procedure is repeated for the probable half of the list and so on until the item is found or it is confirmed that the item does not exist in the list.

Let A be a two-dimensional M x N array. A is represented in memory by a block of (M.N) memory locations. Two-dimensional array may be represented in one of the two orders: 1) row-major 2) column-major. The computer calculates the address of the elements by using the following formulae.

(column-major) LOC(A[J,K]) = Base(A) + w[M(K-1)+ (J-1)]

(row-major) LOC(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]

Base(A) is the address of the first element in two-dimensional array A.

w denotes the number of words per-memory location for the array A.

A record is a collection of related data items. Each item in a record is called a field or attribute. A file is a collection of similar records. A record may contain non-homogeneous data. i.e the data items in a record may have different data types. The items in a record are indexed by attribute names not by subscripts. In this way a record differs from an array. Qualifying an item name with appropriate group item names in the structure, accesses that particular item of a record.

Records may contain non-homogeneous data. So they cannot be stored in an array. Some programming languages provide hierarchical structures to represent records. However a file of records may be stored in memory as a collection of parallel arrays, where elements in different arrays with the same subscript belong to the same record.

4.11. Technical Terms

Array: An array is a linear data structure in which a linear relationship is established between the elements by storing them sequentially.

Sparse Matrix: Matrices with a relatively high proportion of zero entries are called sparse matrices.

Record: A record is a collection of related data items. Each item in a record is called a field or attribute.

File: A file is a collection of similar records.

Multi-dimensional Array: Arrays with more than one dimension are called multi-dimensional arrays. In these arrays the elements are referenced by two or more subscripts.

Pointer: A pointer is a variable that points to another variable in memory. A pointer may point to an element in an array.

Searching: The operation of finding location of an item in a list is called searching.

Sorting: Sorting means rearranging the elements of a linear list in ascending or descending order

4.12. Model Questions

1. Define Array. Discuss its representation in memory.
2. Write an algorithm to sort an array of numbers to ascending order using bubble sort.
3. Define a file and what does it consist of? Discuss the representation of records in memory.
4. Explain how pointers can be used in programming.
5. What is a sparse matrix? Explain the memory representation methods for multidimensional matrices.

4.13. References

1. Theory And Problems Of Data Structures
By SEYMOzxUR LIPSCHUTZ

2. Data Structures and Algorithms, Addison-Wesley
By Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft

-Y. VENKATESWARA RAO, M.C.A.,
Lecturer,Dept. Of Computer Science,
JKC College,GUNTUR-6.

Lesson 5

LINKED LISTS

5.0 Objectives

- Learn about Linked Lists and their Memory Representations
- Learn different operations on Linked Lists
- Understand Garbage Collection
- Learn Two-Way and Header Lists

Structure of the Lesson:

5.1 Introduction to Linked Lists

List means a linear collection of data items. Data processing frequently involves storing and processing data organized into lists. One way to store such data is arrays. The relation between the elements of an array is reflected by their physical placement in memory, but not by any information in the elements. It is very easy to compute the address of an array element. The disadvantage of arrays is that, it is expensive to insert and delete an element in an array. Resizing an array is also expensive. So arrays are considered to be static.

Another way of storing a list in memory is in the form of a linked list. Each element in a linked list contains the address of the next element in the list. So successive elements in the list need not occupy adjacent space in memory. Accordingly insertion and deletion are less expensive in linked lists.

Definition: A linked list or one-way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. Each node contains *information field* and *link field*. The schematic diagram of a linked list is shown in the following picture. The starting node address is in START. The link field of the last node contains null pointer.

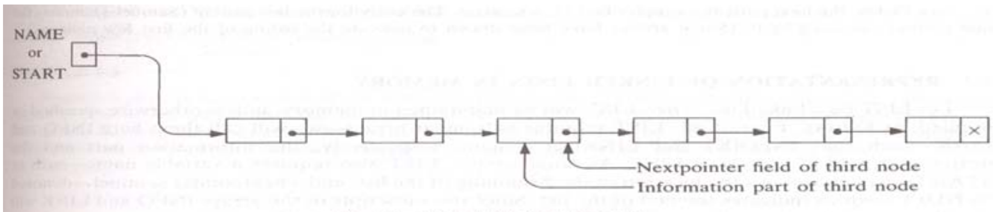
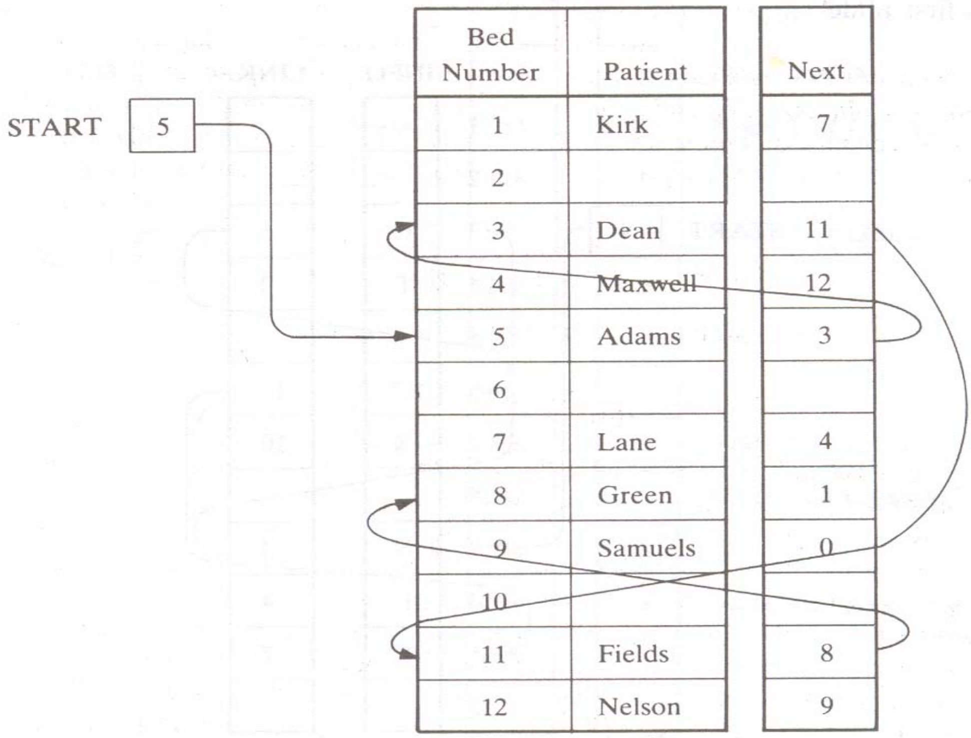

NAME
or
START
Nextpointer field of third node
Information part of third node

Fig. 5-2 Linked list with 6 nodes.



Bed Number	Patient	Next
1	Kirk	7
2		
3	Dean	11
4	Maxwell	12
5	Adams	3
6		
7	Lane	4
8	Green	1
9	Samuels	0
10		
11	Fields	8
12	Nelson	9

Fig. 5-3

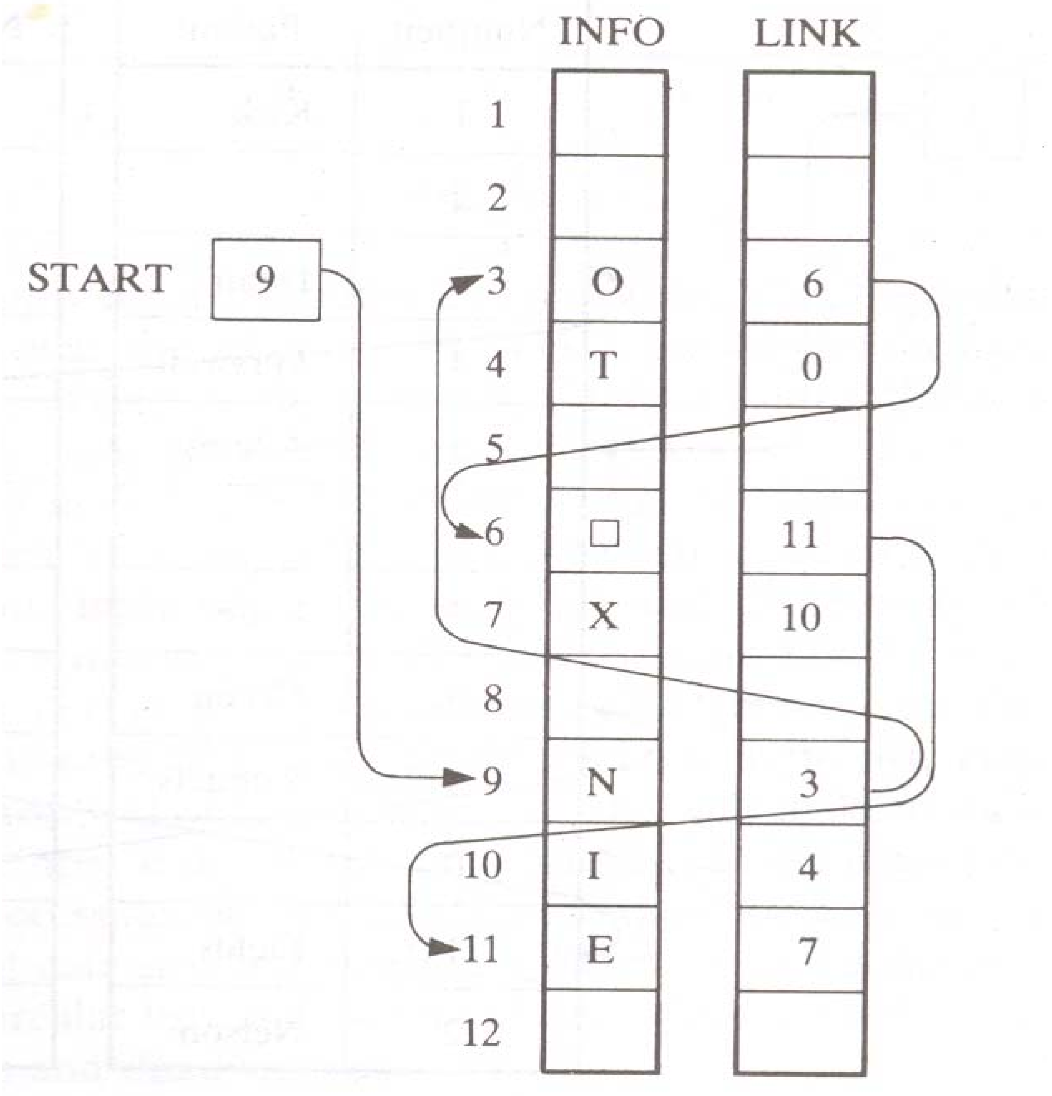


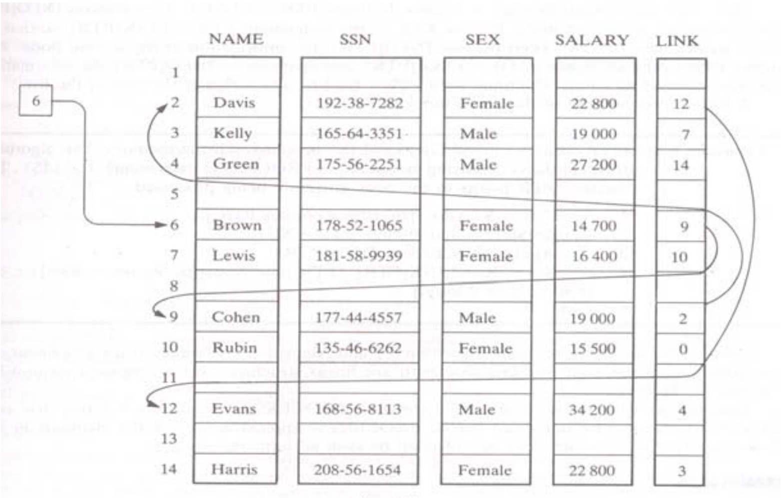
Fig. 5-4

An example of linked list is shown in the following picture.

5.2. Representation of Linked Lists in Memory

Linked list can be represented in memory using linear arrays. Let LIST be a linked list. It requires two arrays INFO and LINK to represent LIST in memory. The INFO array contains information and LINK contains next node address. Thus INFO[K] and LINK[K] represent Kth element in the list. A variable START contains the beginning location of the list. An example of linked list representation using linear arrays is shown in the following figure.

It is possible to store two similar lists using same pair of linear arrays, if the information in the two lists if of the same type. In the following example two lists of test scores are represented using the same pair of arrays TEST and LINK. The representation is shown in fig 5.4. The variable ALG contains the starting address of a list of scores in Algebra and another variable GEOM contains the starting address of list of scores in Geometry.



	NAME	SSN	SEX	SALARY	LINK
1					
2	Davis	192-38-7282	Female	22 800	12
3	Kelly	165-64-3351	Male	19 000	7
4	Green	175-56-2251	Male	27 200	14
5					
6	Brown	178-52-1065	Female	14 700	9
7	Lewis	181-58-9939	Female	16 400	10
8					
9	Cohen	177-44-4557	Male	19 000	2
10	Rubin	135-46-6262	Female	15 500	0
11					
12	Evans	168-56-8113	Male	34 200	4
13					
14	Harris	208-56-1654	Female	22 800	3

Fig. 5-7

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

Name, SSN, Sex, Monthly Salary

This information requires four parallel arrays, say NAME, SSN, SEX, SALARY. The following figure shows a linked list representation of this data with an additional array LINK.

5.3. Traversing and Searching a Linked List

Traversing is an important operation on linked lists. It supports other operations on linked lists. Let LIST be a linked list in memory, stored using linear arrays INFO and LINK. The starting address of the list is in START. To traverse this list you require a pointer variable PTR, which points to the node that is currently being processed. The following algorithm can be used to traverse a linked list.

```
Step1: Set PTR := START
Step2: Repeat Steps 3 and 4 while PTR $\neq$ NULL.
Step3:     Apply PROCESS to INFO[PTR].
Step4:     Set PTR := LINK[PTR].
          [End of Step2 loop.]
Step5: Exit.
```

In the above algorithm, PTR is initialized with START, then PROCESS is applied to

INFO[PTR]. The pointer variable PTR is updated so that it points to the next node in the list and again PROCESS is applied to that node. This continues until the last node is processed. Other operations like printing each node, counting the number of nodes use the above given traversal algorithm for the list. For example, the algorithms for PRINT and COUNT are given here.

Procedure PRINT(INFO, LINK, START)

```
Step1: Set PTR := START.
Step2: Repeat Steps 3 and 4 while PTR $\neq$ NULL
Step3:     Write: INFO[PTR]
Step4:     Set PTR :=  LINK[PTR]
Step5: [End of step2 loop.]
Step6. Return.
```

Procedure COUNT(INFO, LINK, START, NUM)

```
Step1: Set NUM := 0
Step2: Set PTR := START
Step3: Repeat Steps 4 and 5 while PTR $\neq$ NULL
Step4:     Set NUM := NUM + 1.
Step5:     Set PTR := LINK[PTR].
          [End of step 2 loop.]
Step6: Return.
```

Searching a linked list

Let LIST be a linked list stored in memory using INFO and LINK arrays. Given an ITEM, a search can be performed on the list for the presence of ITEM in the list. If the list is a sorted one, then the search concludes in less time. The following algorithm searches a linked list for an ITEM.

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

```
1. Set PTR := START
2. Repeat Step 3 while PTR     NULL
3. If ITEM = INFO[PTR], then:
           Set LOC := PTR, and Exit.
       Else: Set PTR := LINK[PTR].
     [End of step 2 loop.]
4. Set LOC := NULL.
```

If the list is in sorted order, the search algorithm can be refined such that it concludes in much less time. The following algorithm is for search operation on a list sorted in ascending order:

Algorithm SRCHSL(INFO, LINK, START, ITEM, LOC)

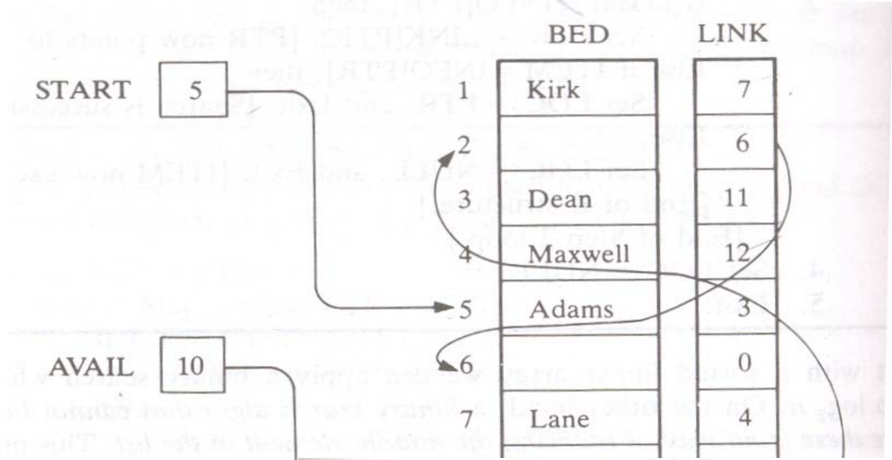
```
 1.   Set PTR := START.
 2.   Repeat Step 3 while PTR     NULL.
 3.      If ITEM < INFO[PTR], then:
            Set PTR := LINK[PTR]
         Else if ITEM = INFO[PTR], then:
            Set LOC := PTR, and then Exit.
         Else:  Set  LOC := NULL, and Exit.
      [End of step 2 loop.]
 4.   Set LOC := NULL.
5. Exit.
```

5.4. Memory Allocation

Linked list is a dynamic structure. It allows insertion and deletion of nodes. These operations require a mechanism to provide unused memory space for the new nodes and to collect memory of deleted nodes for later use. For this mechanism a special list, called *free list*, can be maintained in memory, which consists of unused memory cells. The starting address of this list may be placed in AVAIL variable. The following picture shows a linked list and a free list coexisting in memory.

Garbage Collection

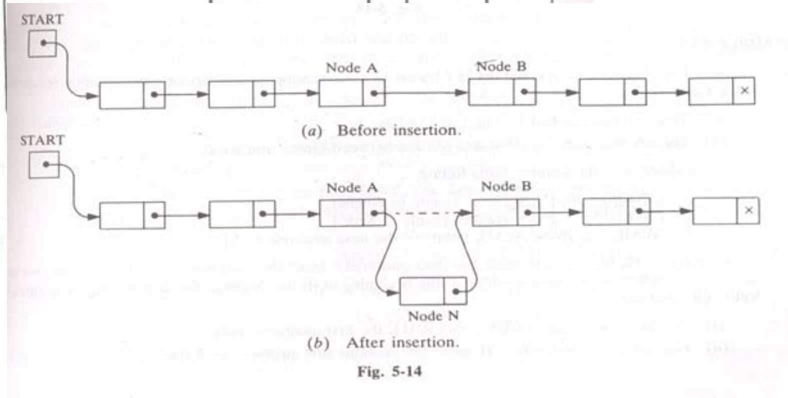
When a node is deleted from a linked list, its memory becomes reusable. This memory space may be immediately inserted into *free list* so that it is available to be used. This method may be time-consuming for an operating system. An alternative for this is that the operating system of a computer may periodically collect all the deleted space onto the *free list*. Any technique, which does this collection, is called *garbage collection*. The garbage collection usually takes place in two steps. First, the computer tags the memory cells in use, and then collects all untagged cells into *free list*. The garbage collection may be initiated when very few locations are available in *free list*.



nodes being inserted into a data structure.
handle the overflow situation carefully so
to the situation where you try to delete a

t

. To insert a node between A and B, the
ged such that, A points to the new node N
g diagram



(a) Before insertion.

(b) After insertion.

Fig. 5-14

The memory space for the new node will come from the AVAIL list. The first available free node in the AVAIL list is taken for new node N. This node is removed from AVAIL list as it joins the data list. The required steps are:

1. The nextpointer field of node A now points to the new node N,
2. AVAIL now points to the second node in the free pool, to which node N previously pointed.
3. The nextpointer field of node N now points to node B, to which node A previously pointed.

The above steps are explained with the following schematic diagram.

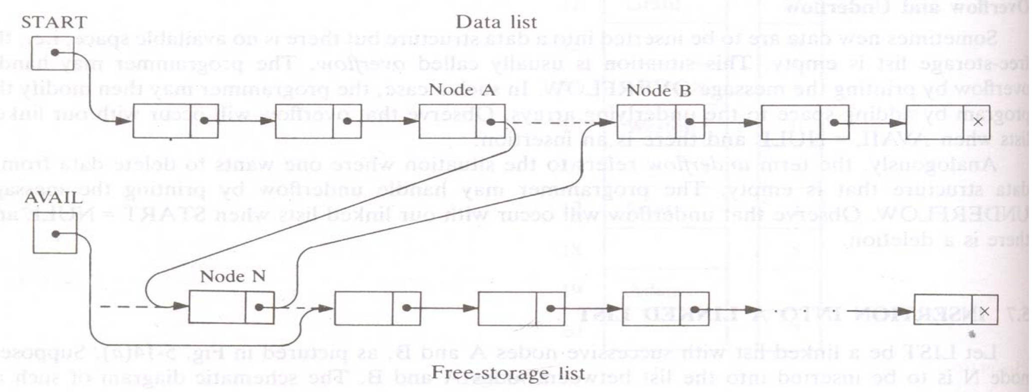


Fig. 5-15

Insertion Algorithm

A new node can be inserted at any position in a linked list. Broadly three distinct places in the list can be identified. They are:

1. At the beginning of a list.
2. After a node with a given location.
3. Into a sorted list (at proper place).

Insertion at the Beginning of a List

It is very easy to insert a new node at the beginning of a list. The algorithm for this is given below. Algorithm to insert a node at the beginning of a linked list.

```
INSFIRST( INFO, LINK, START, AVAIL, ITEM)
  1. [OVERFLOW?] if AVAIL = NULL, then, Write: OVERFLOW, and Exit
  2. [Remove first node in the AVAIL list.]
     Set NEW := AVAIL and AVAIL := LINK[AVAIL]
  3. Set INFO[NEW] := ITEM.
  4. Set LINK[NEW] := START
  5. Set START := NEW.
  6. Exit.
```

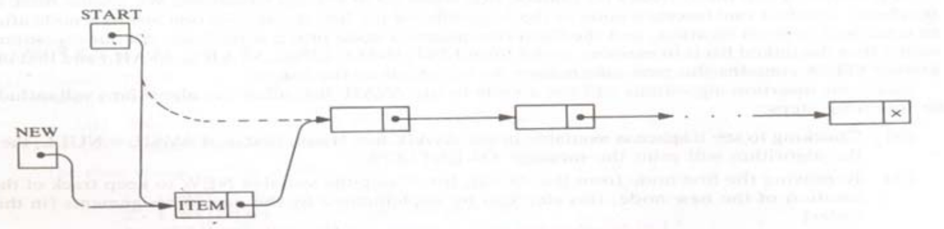


Fig. 5-18 Insertion at the beginning of a list.

Inserting after a Given Node

The following algorithm inserts ITEM so that ITEM follows a node A.

```
INSLOC( INFO, LINK, START, AVAIL, LOC, ITEM)
 1.  [OVERFLOW?] if AVAIL = NULL, then, Write: OVERFLOW, and Exit
 2.  [Remove first node in the AVAIL list.]
    Set NEW := AVAIL and AVAIL := LINK[AVAIL]
 3.  Set INFO[NEW] := ITEM.
 4.  If LOC = NULL, then:
        Set LINK[NEW] := START and START:= NEW
     Else:
        Set LINK[NEW] := LINK[LOC] and LINK[LOC] := NEW
 5.  Exit.
```

Inserting into a Sorted List

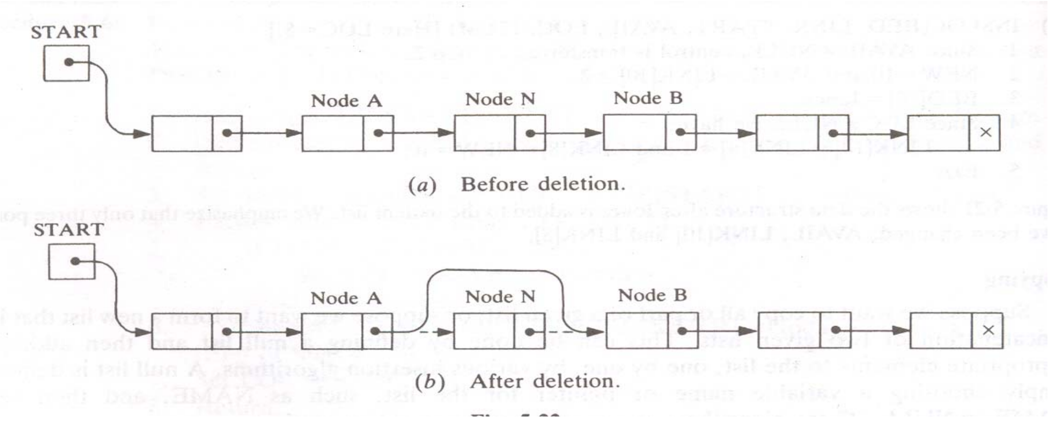
To insert an ITEM into a sorted list, first its suitable location should be found, i.e., two nodes A and B in the list have to be identified such that INFO(A) < ITEM ≤ INFO(B). The following FIND algorithm traverses the linked list and finds a suitable location for ITEM. Once the suitable location is found the INSLOC algorithm can be devised to insert ITEM the desired location. The FIND procedure finds the suitable location for ITEM in the list and assigns it to LOC.

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

```
 1.  [List empty?] If START = NULL, then Set LOC := NULL, and Return
 2.  If ITEM < INFO[START], then Set LOC := NULL and Return
 3.  Set SAVE := START and PTR := LINK[START].
 4.  Repeat steps 5 and 6 while PTR ≠ NULL
 5.     If ITEM < INFO[PTR], then Set LOC := SAVE and Return
 6.     Set SAVE := PTR and PTR := LINK[PTR].
      [End of step 4 loop.]
 7.  Set LOC := SAVE
 8.  Return.
```

Deletion from a Linked List

Nodes can be dynamically deleted from a linked list. To delete a node you need to adjust the next pointer field of its preceding node, so that it points to the succeeding node. This is shown in the following schematic diagram. A deleted node is joined at the beginning of AVAIL list for reuse



(*a*) Before deletion.

(*b*) After deletion.

Nodes can be deleted from any position in a linked list. To popular cases are:

1. To delete the node following a given node.
2. To delete a node with a given ITEM of information.

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. The following algorithm deletes node N with location LOC in the linked list and joins it at the beginning of AVAIL list. In this algorithm LOC contains the location of the node to be deleted from the list and LOCP contains the location of its previous node.

```
DEL( INFO, LINK, START, AVAIL, LOC, LOCP)
1.  If LOCP = NULL, then:
       Set START := LINK[START]
          Else:   Set LINK[LOCP] := LINK[LOC]
2.  Set LINK[LOC] := AVAIL and AVAIL = LOC.
3.  Exit.
```

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Given an ITEM of information, a node N containing ITEM is searched in the LIST. If ITEM does not appear in the LIST then LOC is set to NULL. If ITEM is the information in the first node of LIST then LOC is set to START and LOCP is set to NULL. The FIND procedure, given earlier, can be used with a small modification, to search for ITEM in the LIST. After finding the location LOC of ITEM the DEL procedure can be used to delete the desired node.

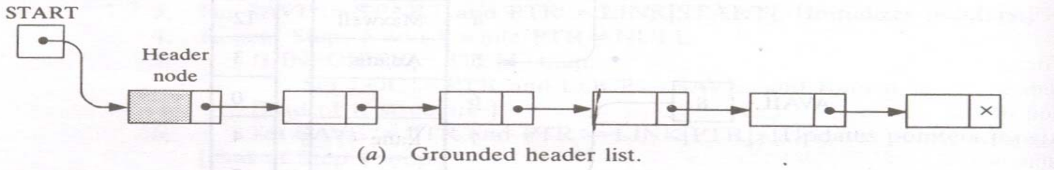
Algorithm to find the location of a node with ITEM as information

```
FINDLOC( INFO, LINK, START, ITEM, LOC, LOCP)
  1.  If START = NULL, then:
          Set LOC := NULL and LOCP := NULL, and Return
  2.  If INFO[START= ITEM, then:
          Set LOC:=START and LOCP := NULL, and Return.
  3.  Set SAVE := START and PTR:=LINK[START].
  4.  Repeat Steps 5 and 6 while PTR ≠ NULL
  5.      If INFO[PTR] = ITEM, then:
              Set LOC := PTR and LOCP := SAVE, and Return.
  6.  Set SAVE := PTR and PTR := LINK[PTR]
      [END OF STEP 4 LOOP.]
  7.  Set LOC := NULL
  8.  Exit.
```

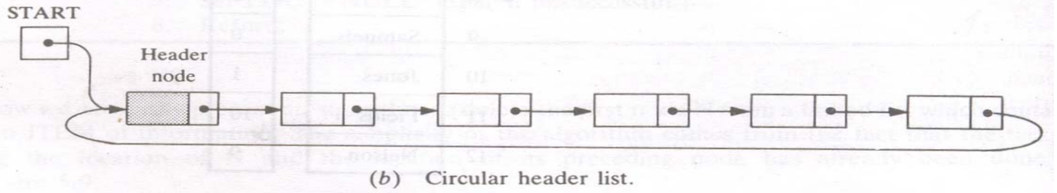
6.6. Header Linked Lists

A header linked list is a linked list which always contains a special node, called header node, at the beginning of the list. Two types of popular header lists are: Grounded header list and Circular header list.

A Grounded Header List is a header list, in which the last node points to NULL. In a Circular header list the last node points back to the header node. The schematic diagrams of these lists are given in the following picture



(*a*) Grounded header list.

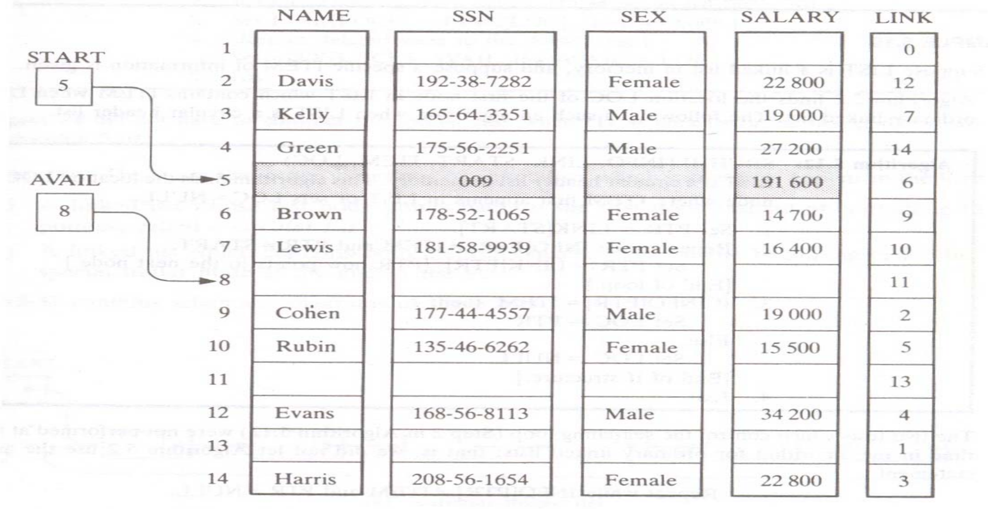


(*b*) Circular header list.

Circular header lists are frequently used because of the ease of operating these lists. In Circular header lists all pointers contain valid addresses and every node has a predecessor. Handling first nodes require no special attention in header lists. An example of a header list represented in memory is given in the following picture.

5.7. Two-Way Header Lists (circular)

A two-way list is a collection of nodes where each node contains two links, a forward link and a backward link. The forward link of a node points to its next node, where as the backward link points to its preceding node in the list. The advantage of a two-way list over a one-way list is, the ability to traverse the list in both the directions. However this comes at a price of an additional pointer variable in each node. The advantage of a circular header list can be combined with that of a two-way list and a circular header two-way list can be formed. In this list the two end nodes point back to the header node. This list requires only one list pointer variable START to store the header node address.

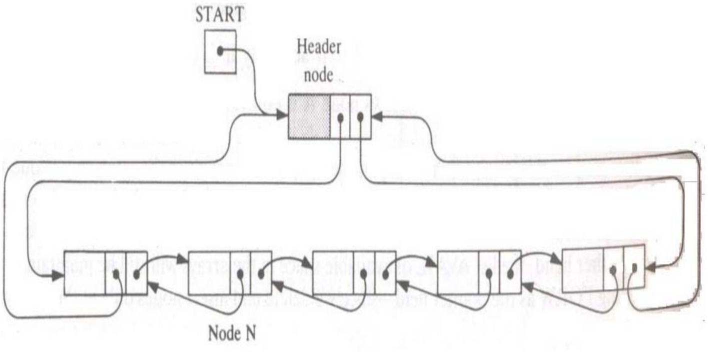

START
5
AVAIL
8

	NAME	SSN	SEX	SALARY	LINK
1					0
2	Davis	192-38-7282	Female	22 800	12
3	Kelly	165-64-3351	Male	19 000	7
4	Green	175-56-2251	Male	27 200	14
5		009		191 600	6
6	Brown	178-52-1065	Female	14 700	9
7	Lewis	181-58-9939	Female	16 400	10
8					11
9	Cohen	177-44-4557	Male	19 000	2
10	Rubin	135-46-6262	Female	15 500	5
11					13
12	Evans	168-56-8113	Male	34 200	4
13					1
14	Harris	208-56-1654	Female	22 800	3

Operations on Two-way Lists

Traversing: Traversing a two-way list is similar to that of a one-way list. You can start from the header node and traverse the list using either the *next* or *prev* pointer. The traversal can also be started from the first node.

Searching: To search for an item in a two way circular header list by traversing the list. Traversing is possible using either of the pointers/links. You can traverse from header node to the header node again using, either of the links. If the list is in sorted order and the current position is somewhere in



Two-way circular header list.

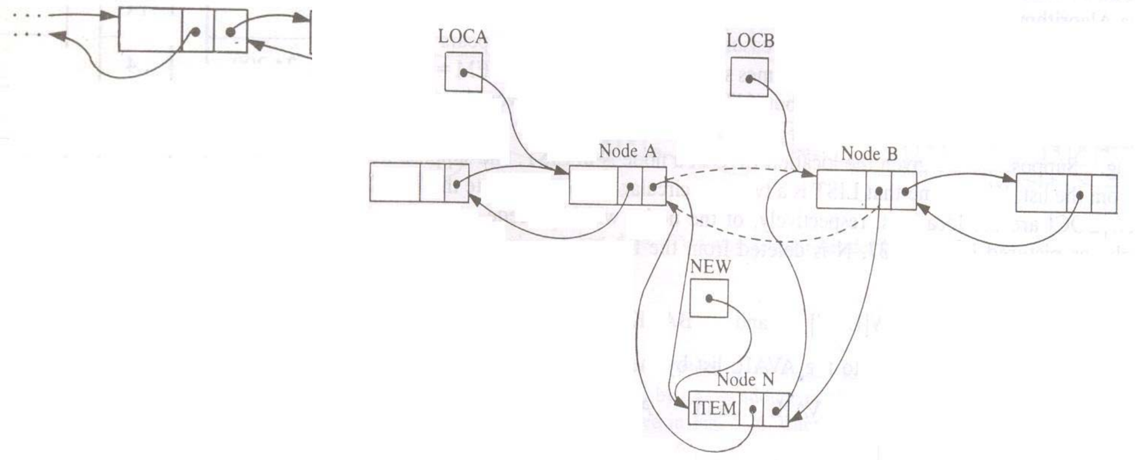
the middle of the list, then you can choose the direction of the search by comparing the search item with the current node in the list.

Deleting: A node N at location LOC in a two-way header list can be deleted by using the following list of steps:

```
FORW[BACK[LOC]] := FORW[LOC]
BACK[FORW[LOC]] := BACK[LOC]
FORW[LOC] := AVAIL
AVAIL = LOC
```

Inserting: Given the locations LOCA and LOCB of two adjacent nodes A and B in a LIST, you can insert a given ITEM of information between nodes A and B, with the following steps:

```
NEW:=AVAIL
AVAIL := FORW[AVAIL]
INFO[NEW] := ITEM
FORW[LOCA] := NEW,        FORW[NEW] := LOCB
BACK[LOCB] := NEW,        BACK[NEW] := LOCA
```



Inserting node N.

5.8. Sur

List means a linear collection of data items. Data processing frequently involves storing and processing data organized into lists. Each element in a linked list contains the address of the next element in the list. Accordingly insertion and deletion are less expensive in linked lists.

A linked list or one-way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. Each node contains *information field* and *link field*. Linked list can be represented in memory using linear arrays. Let LIST be a linked list. It requires two arrays INFO and LINK to represent LIST in memory. The INFO array contains information and LINK contains next node address. A variable START contains the beginning location of the list.

Traversal, insertion and deletion are frequently performed operations on a linked list. A two-way list is similar to a one-way list, but its nodes contain two links. These two links are to succeeding and preceding nodes. A two-way list allows traversal in forward as well as backward directions. Insertion and deletion operations on two-way lists are very similar to those on a one-way list, except that you need to adjust two links instead of one. A two-way list can be circular also. In a circular two-way list the first node is adjacent to last node. An extra header node can be attached to a two-way list. Then the list becomes two-way header list. In a two-way header list, if the last node contains the header node reference, then it is called a two-way circular header list. The header node may contain information pertaining to the entire list

5.9. Technical Terms

LIST: List means a linear collection of data items.

ONE-WAY LIST: A linked list or one-way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

NODE: A node contains information field and link field.

TWO-WAY LIST: A list whose nodes contain the forward and backward links.
CIRCULAR LIST: A list in which the first node is adjacent to the last node.

5.10. Model Questions

1. Explain the operations on a linked-list.
2. Explain the operations on two-way linked list.
3. Describe a two-way circular header list. How do you insert a node into this list?
4. Write the differences between a one-way list and a two-way list.
5. Explain how a linked list be represented using linear arrays.
6. What is meant by garbage collection? Why is it needed?

5.11. References

1. Theory And Problems Of Data Structures
By SEYMOUR LIPSCHUTZ

2. Data Structures and Algorithms, Addison-Wesley

By Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft

- Y. VENKATESWARA RAO, M.C.A.,
Lecturer,Dept. Of Computer Science,
JKC College, GUNTUR

Lesson 6

Stacks, Queues And Recursion

6.0 Objectives

- Learn about Stacks and their Memory Representations
- Learn different operations on Stacks
- Learn about Queues and their Memory Representations
- Learn different operations on Queues and kinds of Queues
- Understand Stacks in converting recursive procedures to non-recursive.
- Learn about Stack Applications.

Structure of the Lesson:

6.1 Introduction to Stacks

A stack is a list of elements in which an element may be inserted or deleted only at one end, called **top** of the stack. Inserting an element into stack is called **push** operation, and removing the top element of a stack is called **pop** operation. The following diagrams show examples of stacks.

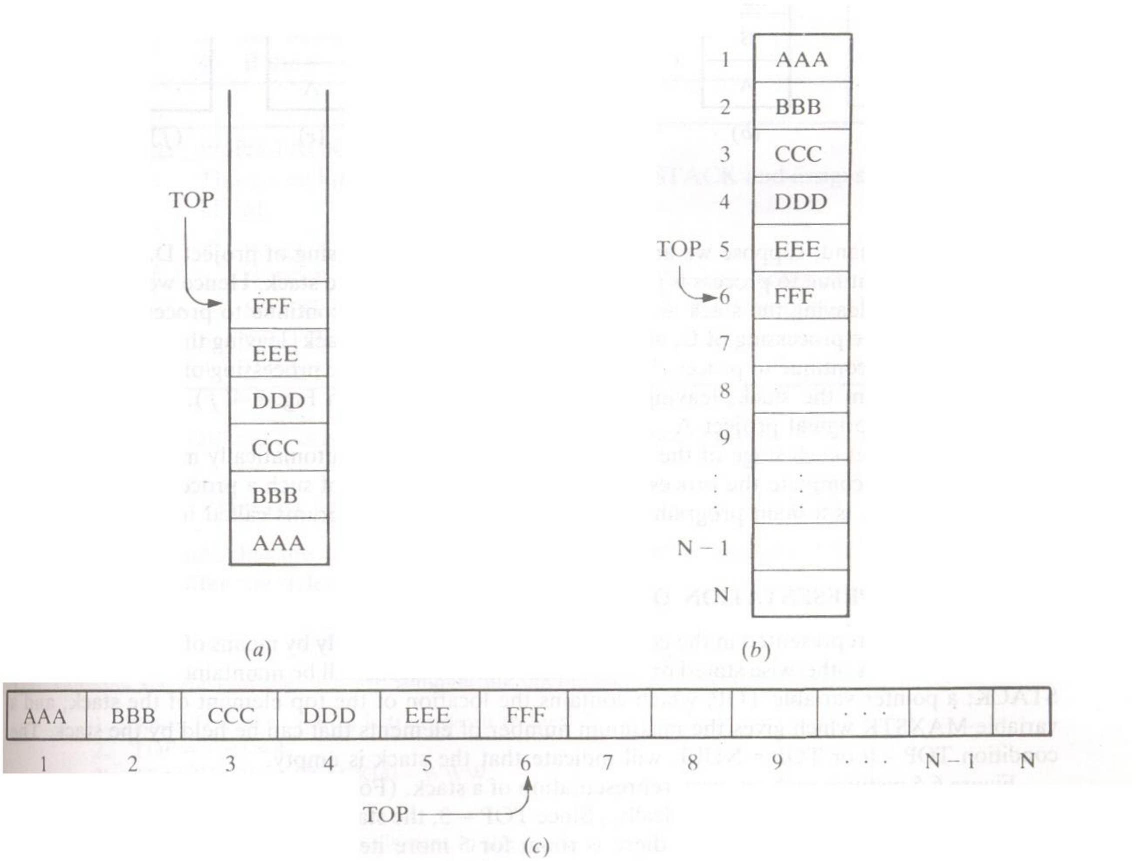


Fig. 6.1 Diagrams of stacks.

6.2. Representation of Stacks in Memory and Stack Operations

Stacks may be represented in the computer using linear arrays. A linear array STACK may be used along with a pointer variable TOP, which contains the location of top element of the stack in the linear array STACK. The maximum size of the linear array may be stored in a variable called MAXSTK. Initially TOP is set to NULL. The push and pop operations operate on the STACK using the TOP index. The following algorithms may be used to push and pop operations on a stack.

```
PUSH(STACK, TOP, MAXSTK, ITEM)
[ This algorithm places ITEM at the top of the stack, provided it is not full. ]
Step 1. If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
Step 2. Set TOP := TOP +1.
Step 3. Set STACK[TOP] := ITEM.
Step 4. Return
```

POP(STACK, TOP, ITEM)

[This algorithm removes the top item on the stack and places it in ITEM, when the stack is not empty.]

```
Step 1. If TOP = 0, then: Print: UNDERFLOW, and Return.
Step 2. Set ITEM:= STACK[TOP].
Step 3. Set TOP: = TOP -1.
Step 4. Return.
```

6.3. Stack Applications

Stack data structure has various applications in computer science. The following is a list of few such applications.

1. Evaluating Postfix expressions.
2. Transforming an Infix expression to a postfix expression.
3. Implementing recursive algorithms as non-recursive algorithms.
4. In sorting algorithms like Quick Sort.
5. In solving Towers of Hanoi problem.

6.3.1 Evaluating Postfix expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm uses a STACK to hold operands, evaluates P

```
Step1. Add a right parenthesis at the end of P.
Step2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the
       sentinel ‘)’ is encountered.
Step3. If and operand is encountered, put it on STACK.
Step4. If an operator ⊗ is encountered, then:
       a)   Remove the two top elements of STACK, where A is the top element and B is
            the next-to-top element.
       b)   Evaluate B ⊗ A.
       c)   Place the result of (b)  back on STACK.
         [End of If structure.]
       [End of step 2 loop.]
Step5. Set VALUE equal to the top element on STACK.
Step6. Exit.
```

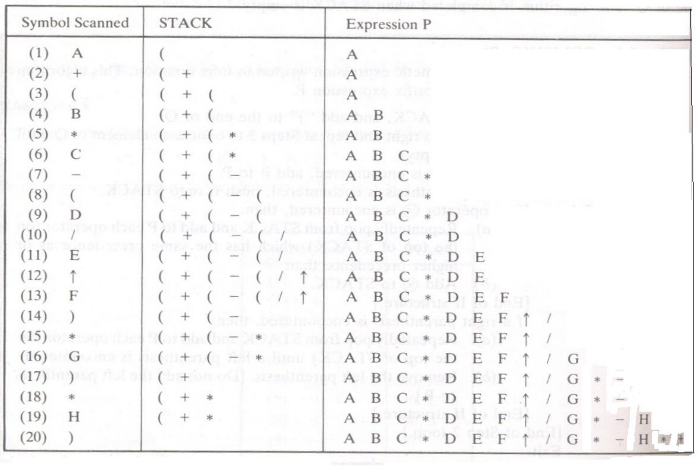
6.3.2 Transforming Infix Expression into Postfix Expression

Let Q be an arithmetic expression written in infix notation. Q may contain operator, operands and parentheses. We assume that the operators in Q consist only of exponentiations (^), multiplications (*), divisions (/), additions (+) and subtractions (-) in their usual precedence order. The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression will be constructed from left to right using the operands from Q and operators which are removed from STACK

POLISH (Q, P)

```
1. Push '(' onto STACK, and  add ')' to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to ^ for each element of Q until the stack is
   empty :
3.     If an operand is encountered, add it to P.
4.     If a left parenthesis is encountered, push it onto STACK.
5.     If an operator ⊗ is encountered, then:
         a) Repeatedly pop from STACK and add to P each operator (on the top of STACK)
            which has the same as or higher precedence than ⊗.
         b) Add ⊗ to STACK.
         [End of If structure.]
6. If a right parenthesis is encountered, then:
         a) Repeatedly pop from STACK and add to P each operator ( on the top of STACK )
            until a left parenthesis is encountered.
         b) Remove the left parenthesis.
    [ End of If structure. ]
  [ End of step 2 loop. ]
7. Exit.
```

For the arithmetic expression Q : A + (B * C – (D / E ^ F) * G) * H the postfix expression P is A B C * D E F ^ / G * - H * +. The trace of the algorithm for this example is given in the following figure.



Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) −	(+ (−	A B C *
(8) (	(+ (− (	A B C *
(9) D	(+ (− (	A B C * D
(10) /	(+ (− (/	A B C * D
(11) E	(+ (− (/	A B C * D E
(12) ↑	(+ (− (/ ↑	A B C * D E
(13) F	(+ (− (/ ↑	A B C * D E F
(14))	(+ (−	A B C * D E F ↑ /
(15) *	(+ (− *	A B C * D E F ↑ /
(16) G	(+ (− *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * −
(18) *	(+ *	A B C * D E F ↑ / G * −
(19) H	(+ *	A B C * D E F ↑ / G * − H
(20))		A B C * D E F ↑ / G * − H * +

Fig. 6.2

6.3.3 Quick Sort

Quick soft is an algorithm of divide-and-conquer type. In this algorithm, the problem of sorting a set is reduced to the problem of sorting two smaller sets. Suppose A is the following list of 12 numbers

44 33 11 55 77 90 40 60 99 22 88 66

The reduction step of quick-sort algorithm finds the final position of one of the numbers. Let us take the first number 44. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

22 33 11 55 77 90 40 60 99 **44** 88 66

Beginning with 22, scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list:

22 33 11 **44** 77 90 40 60 99 55 88 66

Proceeding like that, 44 is finally positioned in the list such that all numbers less than 44 are to its left and all numbers in the list and greater than 44 are to its right. These numbers now form two sublists, the first sublist (numbers less than 44) and the second sublist (numbers greater than 44).

22 33 11 40 **44** 90 77 60 99 55 88 66

[← First sublist →] [← Second sublist →]

The reduction step is repeated with each sublist. The following algorithm sorts a list of numbers using quick-sort procedure

QUICK(A, N, BEG, END, LOC)

```
1. Set LEFT := BEG, RIGHT := END and LOC:= BEG.
2. a. Repeat while A[LOC] $\le$ A[RIGHT] and LOC $\ne$ RIGHT:
        RIGHT := RIGHT – 1.
   b. If LOC = RIGHT, then: Return.
   c. If A[LOC] > A[RIGHT], then:
        i)   TEMP := A[LOC] , A[LOC] := A[RIGHT ], A[RIGHT ] := TEMP.
        ii)  Set LOC := RIGHT.
        iii) Go to step 3.
     [End of if structure.]
```

```
3.  a. Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC :  LEFT := LEFT +1.
    b. If LOC = LEFT, then: Return.
    c. If A[LEFT] > A[LOC], then
        i. TEMP := A[LOC],   A[LOC] := A[LEFT], A[LEFT]:= TEMP.
        ii. Set LOC := LEFT.
        iii. Go to step 2.
    [End of If structure.]
```

Algorithm to soft a list of numbers using quick-sort.

```
1.  TOP := NULL.
2.  If N > 1, then: top := top +1, LOWER[1] := 1, UPPER[1] := N.
3.  Repeat steps 4 to 7 while TOP ≠ NULL.
4.  Set BEG := LOWER[TOP], END := UPPER[TOP], TOP = TOP –1.
5.  Call QUICK(A, N, BEG,END, LOC).
6.  If BEG < LOC-1, then:
        TOP := TOP + 1, LOWER[TOP] := BEG, UPPER[TOP] := LOC –1.
    [End of If structure.]
7.  If LOC +1 < END, then:
        TOP := TOP +1 , LOWER[TOP] := LOC +1, UPPER[TOP] := END.
    [End of If structure.]
8.  Exit.
```

6.3.4 Towers of Hanoi

Towers of Hanoi, is a popular problem in computer science. This problem can be solved using stack data structure. The problem can be described as, suppose there are three pegs, labeled A, B and C. Suppose on peg A there are placed a finite number of disks with decreasing size as shown in the following picture. The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary. The rules of the game are as follows:

1. Only one disk may be moved at a time. Specifically, only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk

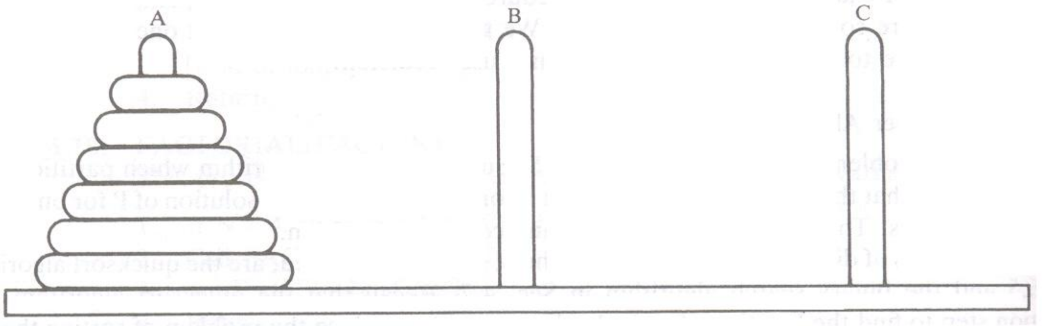
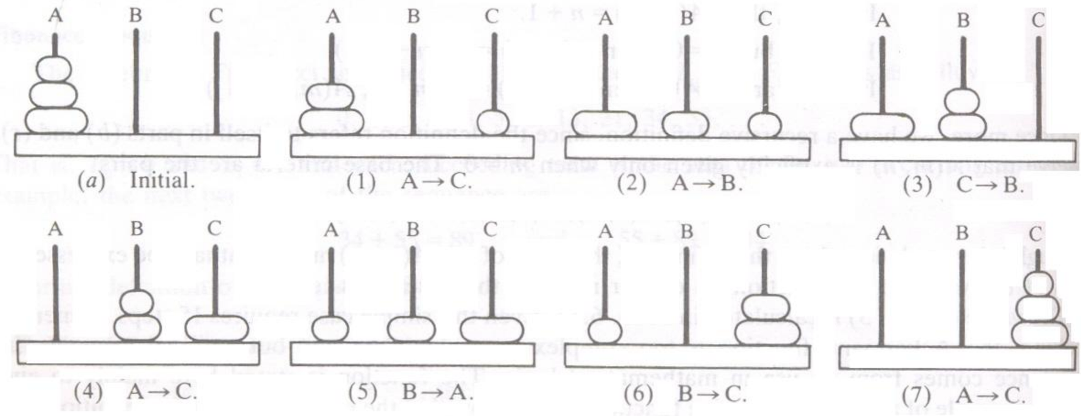


Fig. 6.3 Initial setup of Towers of Hanoi with $n = 6$.

The solution for Towers of Hanoi problem with n = 3 can be stated in seven steps as below:

1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C



(*a*) Initial. (1) A→C. (2) A→B. (3) C→B.

(4) A→C. (5) B→A. (6) B→C. (7) A→C.

Fig. 6.4

A recursive solution as stated below, can be devised for the problem for n>1 disks

Step1. Move the top n-1 disks from peg A to peg B.

Step2. Move the top disk from peg A to peg C.

Step3. Move the top n-1 disks from peg B to peg C

The recursive solution can be implemented as an algorithm. A non-recursive solution is also possible using a STACK. Both the algorithms are given below.

```
TOWER (N, BEG, AUX, END)
    1. If n = 1, then:
        a. Write: BEG → END.
        b. Return.
    2. Call TOWER (N-1, BEG, END, AUX). [Move N-1 disks from BEG to AUX.]
    3. Write: BEG → END.
    4. Call TOWER (N-1, AUX, BEG, END). [Move N-1 disks from BEG to END.]
    5. Return.
```

TOWER (N, BEG, AUX, END)

This is a non-recursive solution to the Towers of Hanoi Problem for N disks. Stacks STN, STBEG, STAUX, STEND and STADD will correspond to the variables N, BEG, AUX, END and ADD.

```
1.  Set TOP := NULL.
2.  If N = 1, then:
        a.  Write BEG → END.
        b.  Go to step 5.
3.  a.  i. Set TOP := TOP +1.
        ii. Set STN[TOP] := N,  STBEG[TOP] := BEG,  STAUX[TOP] := AUX,
                STEND[TOP := END, STADD[TOP] := 3.
    b. Set N := N-1, BEG := BEG, AUX := END, END := AUX.
    c.  Go to step 1.
4.  Write BEG → END.
5.  a. i. Set TOP := TOP +1
       ii. Set STN[TOP] := N, STBEG[TOP] := BEG, STAUX[TOP] := AUX,
                STEND[TOP] := END, STADD[TOP] := 5.
    b. Set N := N-1, BEG := AUX, AUX := BEG, END := END.
    c. Go to step 1.
6.  a. If TOP := NULL, then: Return.
    b. i. Set N := STN[TOP], BEG := STBEG[TOP], AUX := STAUX[TOP],
              END := STEND[TOP], ADD := STADD[TOP].
       ii. Set TOP := TOP –1.
    c. Go to step ADD.
```

6.3.5 Translation of a Recursive Procedure to Non-recursive Procedure

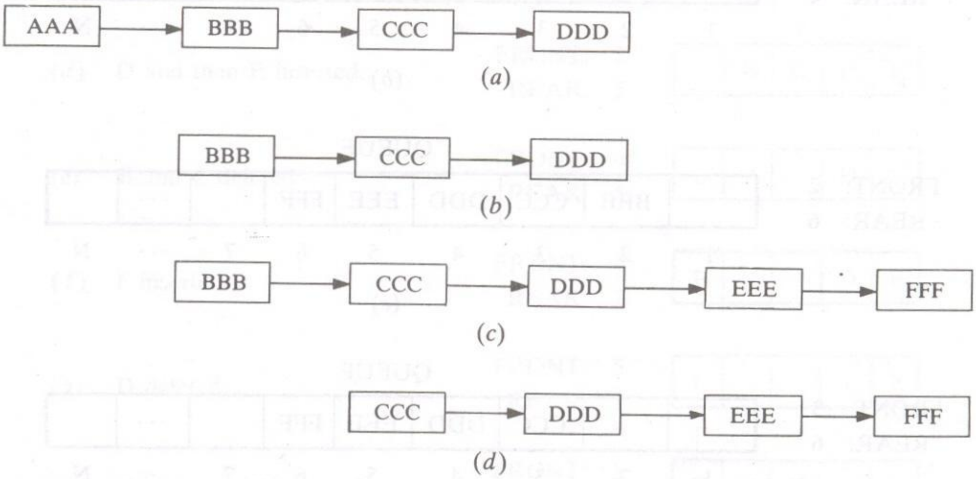
Suppose P is a recursive procedure, and the recursive call for P comes from P only. The translation of such a procedure P into non-recursive procedure is accomplished as:

1. Define a stack STPAR for each parameter PAR.
2. Define a stack STVAR for each local variable VAR.
3. Define a local variable ADD and a stack STADD to hold return addresses.

Each time the recursive procedure is called the current values of the parameters and local variables are pushed onto the corresponding stacks for future processing.

6.4. Queue Data Structure and Operations

A **queue** is a linear list of elements in which deletions can take place only at one end called the **front** and the insertions can take place only at the other end, called the **rear**. A queue is also called a FIFO list. FIFO stands for First-In First-Out. Queues may be represented in computer memory in various ways. Usual choices for representing queues are linear lists and linear arrays. The following picture shows the representation of a queue using a linear list.



Each representation of queue requires the two pointer variables FRONT and the REAR. The following picture shows the representation of a QUEUE using a linear array. The index variables FRONT and REAR are used to point to the front element and the last element of the queue. A deletion operation in QUEUE can be implemented by the assignment FRONT := FRONT +1. An insertion operation in a QUEUE is implemented by the assignment REAR := REAR +1.

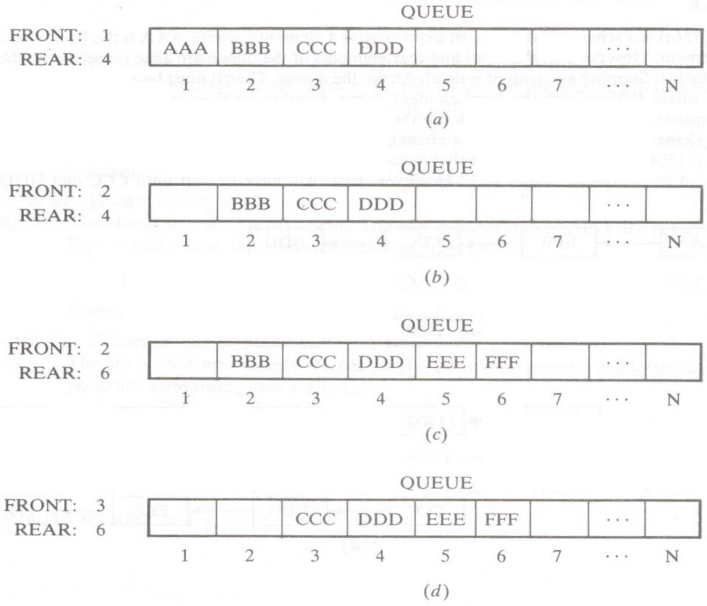


Fig.6.6 Array representation of a queue.

For a queue of size N, after N insertions the queue becomes full and the REAR index becomes N. Now if all the N elements are removed from the queue, the FRONT index also becomes N. At this stage both the pointers FRONT and REAR can be reset to 1, to allow fresh insertion of elements into the queue. This way of operating a queue is called circular queue. A list of insertions and deletions on a circular queue are shown in the following picture.

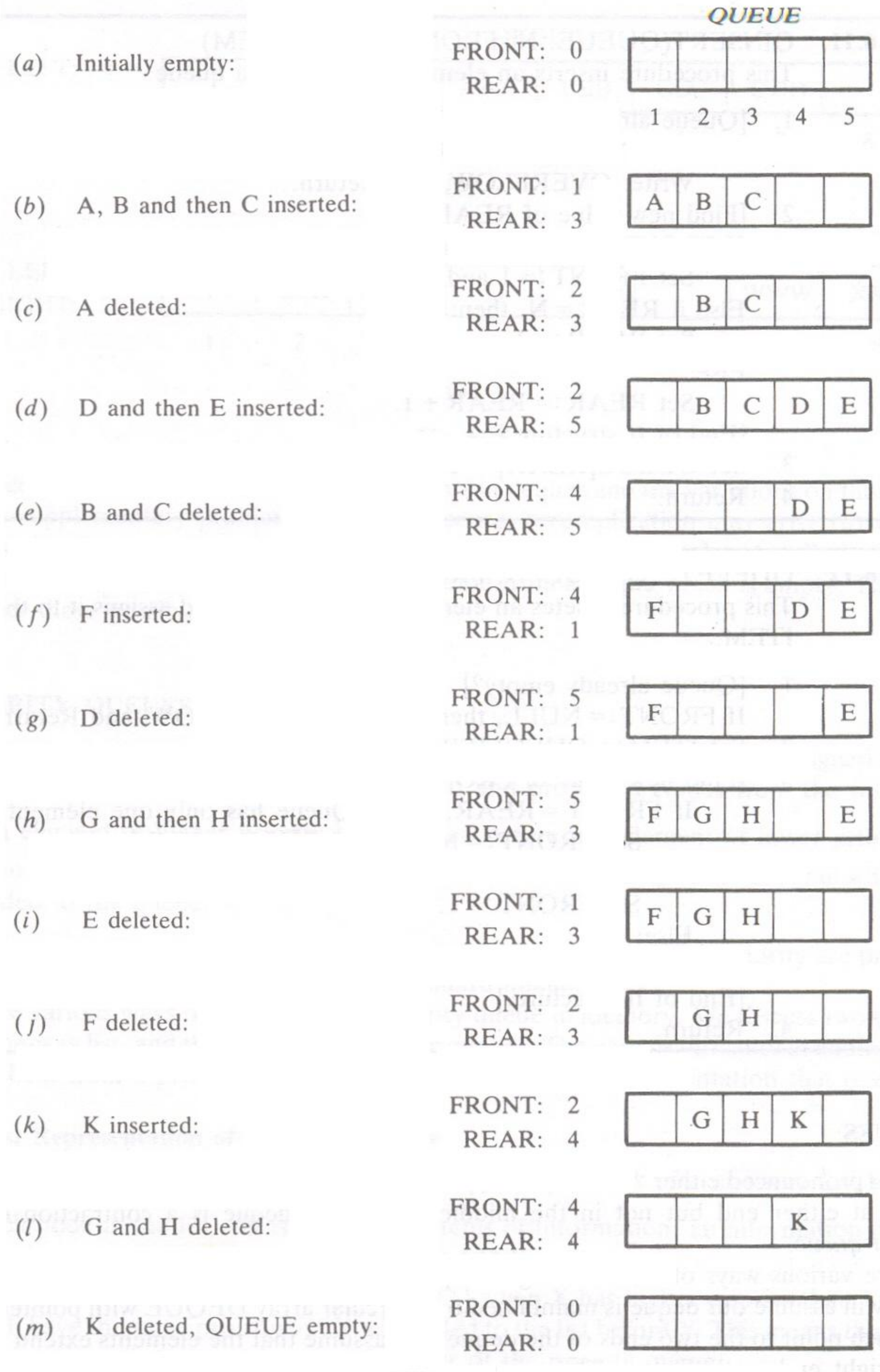

QUEUE
(a) Initially empty: FRONT: 0 REAR: 0
1 2 3 4 5
(b) A, B and then C inserted: FRONT: 1 REAR: 3 A B C
(c) A deleted: FRONT: 2 REAR: 3 B C
(d) D and then E inserted: FRONT: 2 REAR: 5 B C D E
(e) B and C deleted: FRONT: 4 REAR: 5 D E
(f) F inserted: FRONT: 4 REAR: 1 F D E
(g) D deleted: FRONT: 5 REAR: 1 F E
(h) G and then H inserted: FRONT: 5 REAR: 3 F G H E
(i) E deleted: FRONT: 1 REAR: 3 F G H
(j) F deleted: FRONT: 2 REAR: 3 G H
(k) K inserted: FRONT: 2 REAR: 4 G H K
(l) G and H deleted: FRONT: 4 REAR: 4 K
(m) K deleted, QUEUE empty: FRONT: 0 REAR: 0

Fig. 6.7

The following algorithms can be used to insert an element into a circular queue and to remove an element from a circular queue

Procedure

QINSERT(QUEUE, N, FRONT, REAR, ITEM)

This procedure inserts an element ITEM into a queue.

```
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
           Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
          Set FRONT := 1 and REAR := 1
   Else if REAR = N, then:
           Set REAR := 1
   Else:
          Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Procedure

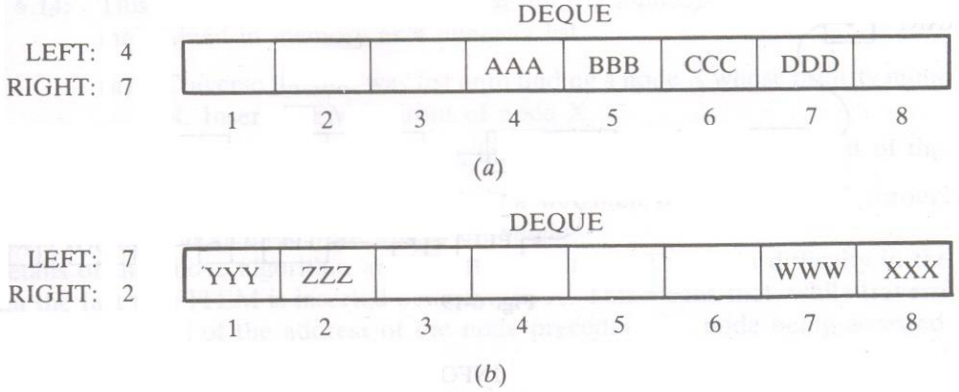
QDELETE(QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue and assigns it to the variable ITEM.

```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
           If FRONT = REAR, then: [Queue has only one element to start.]
             Set FRONT := NULL and REAR := NULL.
        Else if FRONT = N, then:
              Set FRONT := 1.
        Else:
               Set FRONT := FRONT + 1.
   [End of If structure]
4. Return
```

6.5. DEQUES

A deque is a linear list in which elements can be added or removed at either end but not in the middle. It is also called a double-ended queue. A deque can be represented in a computer's memory in various ways. One possible way is to represent using a circular array with two pointers LEFT and RIGHT. These pointers point to the two ends of the deque. In this representation the condition LEFT = NULL will be used to indicate that a deque is empty. The following picture shows a deque with a few elements in it and the LEFT and RIGHT index variables



(*a*)

(*b*)

Fig. 6.8

There are two variations of a deque. They are:

1. Input-restricted deque.
2. Output-restricted deque.

An input-restricted deque allows insertions at only one end of the list, but allows deletions at both ends of the list. An output-restricted deque is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

6.6. Priority Queues

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which element s are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Tow elements with the same priority are processed according to the order in which they were added to the queue.

An example for a priority queue is a timesharing system. In a timesharing system programs of higher priority are processed first, and programs with the same priority from a standard queue. A priority queue can be implemented using a one-way list as follows:

1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.
2. A node X precedes a node Y in the list a) when X has higher priority than Y or b) when both have the same priority but X was added to the list before Y.

The following algorithms can be used to insert an element into a priority queue and to remove an element from a priority queue.

Algorithm

This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list

1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm

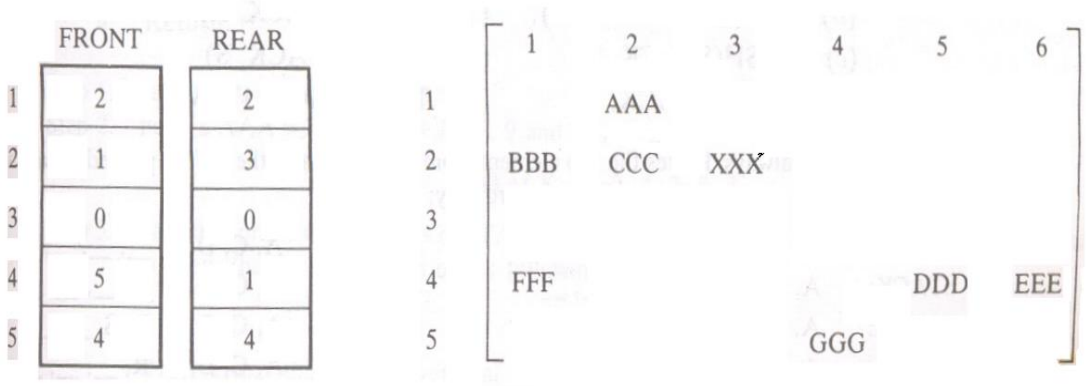
This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(*a*) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

(*b*) If no such node is found, insert ITEM as the last element of the list.

Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority. Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. If each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. The following figure shows the representation for the priority queue.



	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Fig. 6.9

The following algorithms can be used for insertion and deletion operations on priority queues

Algorithm

The algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm

This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

6.7. Summary

A stack is a list of elements in which an element may be inserted or deleted only at one end, called **top** of the stack. Inserting an element into stack is called **push** operation, and removing the top element of a stack is called **pop** operation. Stacks may be represented in the computer using linear arrays. A linear array STACK may be used along with a pointer variable TOP, which contains the location of top element of the stack in the linear array STACK. Stack data structure has various applications in computer science.

Stack data structure has various applications in computer science.

Evaluating Postfix expressions.

Transforming an Infix expression to a postfix expression.

Implementing recursive algorithms as non-recursive algorithms.

In solving Towers of Hanoi problem.

A **queue** is a linear list of elements in which deletions can take place only at one end called the **front** and the insertions can take place only at the other end, called the **rear**. A queue is also called a FIFO list. FIFO stands for First-In First-Out. Queues may be represented in computer memory in various ways. Usual choices for representing queues are linear lists and linear arrays. The following picture shows the representation of a queue using a linear list. A deque is a linear list in which elements can be added or removed at either end but not in the middle. It is also called a double-ended queue. A deque can be represented in a computer's memory in various ways. One possible way is to represent using a circular array with two pointers LEFT and RIGHT. These pointers point to the two ends of the deque. In this representation the condition LEFT = NULL will be used to indicate that a deque is empty. A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which element s are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Tow elements with the same priority are processed according to the order in which they were added to the queue.

An example for a priority queue is a timesharing system. In a timesharing system programs of higher priority are processed first, and programs with the same priority from a standard queue. A priority queue can be implemented using a one-way list as follows:

1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.
2. A node X precedes a node Y in the list a) when X has higher priority than Y or b) when both have the same priority but X was added to the list before Y.

6.8 Technical Terms

Stack: Is a linear list in which elements can be inserted or removed only at one end known as top of the stack.

Top: It is a pointer that points to the top-most element in the stack.

Push: Inserting an element onto a stack.

Pop: Removing the top element of the stack.

Queue: It is a linear list in which elements are inserted at the rear end, and removed from the front end.

Insert: Inserting an element at the rear of a Queue.

Delete: Removing the front element of a Queue.

Circular Queue: A Queue whose front edge is considered as adjacent to the rear edge.

Priority Queue: A Queue whose elements are assigned different priorities and removed according to their priorities.

Deque: A Double Ended Queue. It allows insertions / deletions at both ends.

Postfix: A notation to write arithmetic expressions. Popularly used in computer science due to ease of evaluation. In this notation the operator is written after its operands.

Infix: A notation to write arithmetic expressions. In this notation the operator is written between its operands. Many programming languages support infix expressions.

Prefix: It is a notation to write arithmetic expressions. In this notation the operator precedes its operands.

Recursion: A method of calling a procedure from with in itself.

6.9. Model Questions

1. Define Stack data structure. Write the algorithms for push and pop operations.
2. Write a non-recursive algorithm for solving Towers of Hanoi problem using stack.
3. Write a recursive algorithm for Towers of Hanoi problem.
4. Explain the Quick sort procedure. Write its algorithm.
5. Define Queue, Deque, Priority Queue.
6. Differentiate between a Queue and a Priority Queue.
7. Write an algorithm for evaluating a postfix expression.
8. Explain how to convert a recursive procedure to non-recursive procedure.
9. Discuss the alternatives in representing Stack and Queue data structures in computer memory.

6.10. References

1. Theory And Problems Of Data Structures

By SEYMOUR LIPSCHUTZ

2. Data Structures and Algorithms, Addison-Wesley

By Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft

-Y. VENKATESWARA RAO, M.C.A.,
Lecturer, Dept. Of Computer Science,
JKC College, GUNTUR-6.

Lesson 7

Trees -1

7.0 Objectives:

The data structures that we have seen so far (stacks, queues, and linked lists) were linear data structures. In this lesson we consider a data structure named as tree that is useful in many applications. It starts with the general tree and then concentrates on binary trees, tree traversal techniques.

- To know about hierarchical structures
- To know about trees
- To learn representing trees in memory
- To know about different types of binary trees
- Describes tree traversing techniques

Structure of the Lesson:

7.1 Introduction

A tree is a non-linear data structure in which elements are represented as nodes and are linked together in hierarchical fashion. *Example*: Organization charts, the structure of mathematical formulas, database information systems. Trees are used to represent a collection of items or elements along with their relationship with one another.

7.2 Hierarchical Structures

Trees are very useful conceptual tool for representing relationships among certain classes of items. The relationship of the elements in a tree is one to many. For example, a family tree fig. 7.1 as shown below:

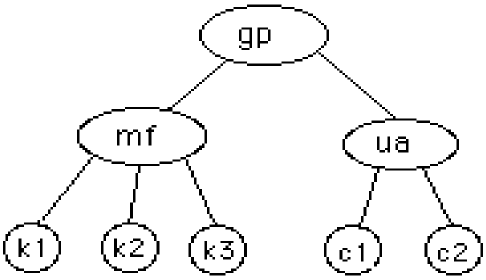


Fig. 7.1 Hierarchical Structure

7.3 What Is A Tree?

A tree is a collection of nodes with no cycles. A tree consists of a distinguished node R, called the root, and zero or more nonempty (sub) trees T_1, T_2 ... T_k each of whose roots are connected by a directed edge from R. The tree has the ability to grow and expand, and is therefore a dynamic, flexible, and open-ended system.

In computer science, a compiler uses trees for representing algebraic expressions. The following tree fig. 7.2 specifies an unambiguous order for evaluating an expression.

$$E = (a-b)/((c*d)+e)$$

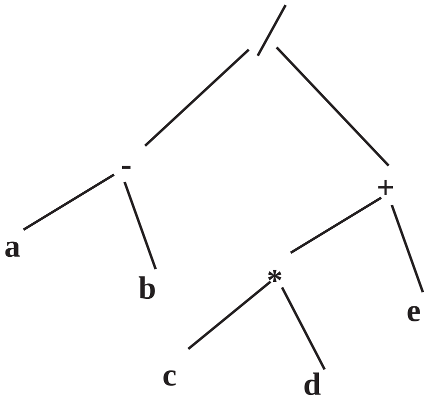


Fig. 7.2

7.4 General Characteristics Of Trees

1. Each element of a tree is called a **node** or a **vertex**.
2. There is a unique first node called the **root** that has no predecessors, but may have many successors.
3. A node with no children is called a **leaf**.
4. An **internal node** is any node that is not a leaf (so this includes the root). fig. 7.3 shows a sample tree.

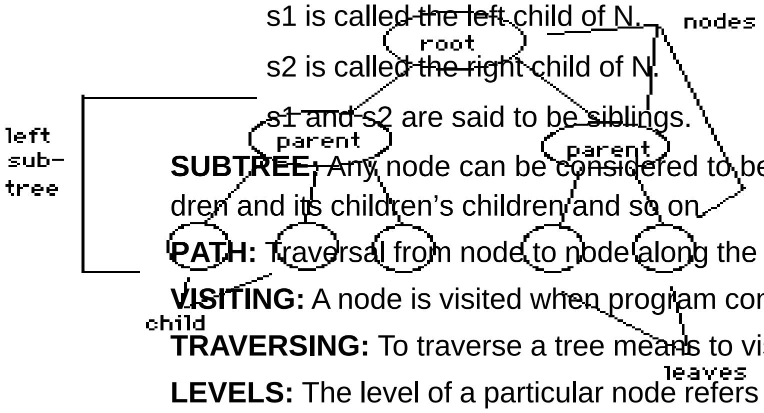


Fig. 7.3 a sample tree

7.5 Tree Terminology

Suppose N is a node in T with left successor s1 and right successor s2. Then N is called the parent (or father) of s1 and s2.

s1 is called the left child of N.

s2 is called the right child of N.

s1 and s2 are said to be siblings.

SUBTREE: Any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.

PATH: Traversal from node to node along the edges results in a sequence called path.

VISITING: A node is visited when program control arrives at the node, usually for processing.

TRAVERSING: To traverse a tree means to visit all the nodes in some specified order.

LEVELS: The level of a particular node refers to how many generations the node is from the root. Root is assumed to be level 0.

KEYS: Key value is used to search for the item or perform other operations on it.

DEPTH: The depth or height of a tree T is the maximum number of nodes in a branch of T.

Level number: Each node in a binary tree T is assigned a level number. The root R of the tree T has level 0,and the level of any other node in the tree is one more than the level of its father.

Nodes with the same level number belong to the same generation.

The **level** of a node is the number of *branches* traversed in reaching it from the root.

The **height** or **depth** of a tree is the maximum level of any leaf in the tree. fig. 7.4 shows the height of a tree.

.

Fig. 7.4

Note: Height of a tree is 1 more than the largest level number of T. [Different textbooks have different definitions of height.]

Let v is the father of w, and w is a son of v. If there is a path from v to w, then v is an **ancestor** of w and w is a **descendant** of v.

Children of the same parent are said to be **siblings.**

The nodes immediately below (to the left and right of) a given node are called its **children**.

The node immediately above a given node is called its **parent.**

dresrdered Tree An ordered tree is a tree in which the sons of each vertex are ordered. When drawing an ordered tree, assume that the sons of each vertex are ordered from left to right. A binary tree is an ordered tree.

Example:

Figure 7.5 pictures a general tree T with 11 nodes,

A, B, C, D, E, F, G, H, J, K, L

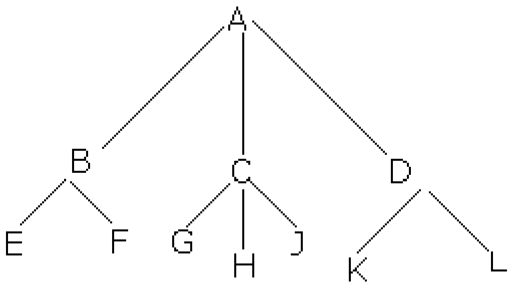


Fig. 7.5

Root of a tree T is the node at the top of the diagram, and the children of a node are ordered from left to right.

A is the root of T, and A has three children, the first child B, the second child C and the third child D observe that:

(a) The node C has three children.

(b) Each of the nodes B and D has two children.

(c) The nodes E, F, G, H, J, k, and L have no children

7.6 Representation Of General Trees

Suppose T is a general tree. T will be maintained in memory by means of a linked representation, which uses three parallel arrays INFO, CHILD (or DOWN) and SIBL (or HORZ), and a pointer variable ROOT as follows. First of all, each node N of T will correspond to an allocation k such that:

(1) INFO [k] contains the data at node N.

(2) CHILD [k] contains the location of the first child of N. The condition CHILD[k] =NULL indicates that N has no children.

(3) SIBL[k] contains the location of the next sibling of N. The condition SIBL[k]=NULL indicates that N is the last child of its parent.

Root contain the location of the root R of T.

The above representation may be extended to represent a forest F consisting of trees T_1, T_2, . . . , T_m by assuming the roots of the trees are siblings. In this case, ROOT will contain the location of the root R_1 of the first tree T_1 or when F is empty, ROOT will be equal to NULL.

Example:

Consider the general tree T in fig. 7. 5. Suppose the data of the nodes of T are stored in an array INFO as in Fig. 7.6(a). The structural relationships of T are as follows:

(a) The root A of T is stored in INFO[2], set ROOT:=2.

(b) The first child of A is the node B, which is stored in INFO[3], set CHILD[2]:=3. Since A has no sibling, set SIBL[2]:=NULL.

(c) The first child of B is the node E, which is stored in INFO[15], set CHILD[3]:=15. Since node

C is the next sibling of B and C is stored in INFO[4], set SIBL[3]:=4.

	INFO
1	
2	A
3	B
4	
5	L
6	H
7	J
8	K
9	C
10	D
11	G
12	F
13	E
14	

Fig. 7.6(a)

	CHILD	SIBL
1	4	
2	3	0
3	13	9
4	14	
5	0	0
6	0	7
7	0	0
8	0	5
9	11	10
10	8	0
11	0	6
12	0	0
13	0	12
14	0	

Fig. 7.6(b)

Fig. 7.6(b) gives the final values in CHILD and SIBL. The AVAIL list of empty nodes is maintained by the first array, CHILD, where AVAIL =1

7.7 Binary Trees

The binary tree is a fundamental data structure. It is useful for storing sorted data and for retrieving stored data. A *binary tree* is a special type of tree where each node has exactly 0, 1, or 2 children.

7.7.1 What is a Binary Tree?

A binary tree T is defined as a finite set of elements, called nodes, such that:

a) T is empty (called the null tree or empty tree), or

b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary tree T1 and T2.

If T does contain a root R, then T1 is left sub tree and T2 is right sub tree of R. If T1 is non-empty, then its root is called the left successor of R. Similarly, if T2 is nonempty, then its root is called the right successor of R.

- Every node in a binary tree can have at most two children.

  The two children of each node are called the left child and right child corresponding to their positions.

- A node can have only a left child or only a right child or it can have no children at all.

The following fig. 7.7 shows a sample binary tree.

Fig. 7.7 sample binary tree

Skewed Binary Tree

There are 2 types of binary trees

1. Left skewed binary tree 2. Right skewed binary tree

The following fig. 7.8 shows right skewed binary tree

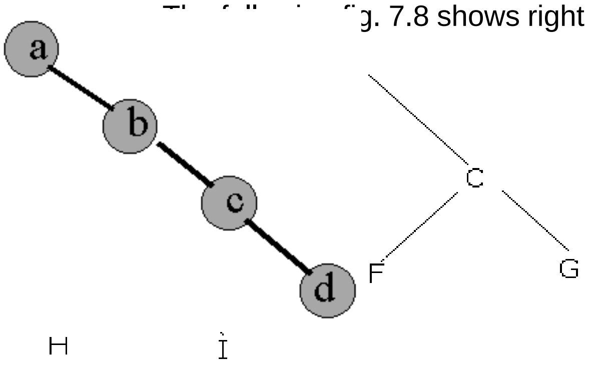


Fig. 7.8

7.7.2 Complete Binary Tree

Consider any binary tree T. Each node T can have at most two children. According to this, the level r of T can have at most 2^r nodes.

Definition: The Tree T is said to be *complete* if all its levels, except the last have the maximum number of possible nodes.

A complete binary tree has the following two properties:

- All internal nodes have two children.
- All levels except the last have the maximum number of possible nodes. It is filled from left to right.

The complete binary tree is as shown in fig 7.9(a).

Fig. 7.9(a) **Fig. 7.9(b)**

The tree in fig. 7.9(b) is not a complete binary tree, because the lowest level is not filled from left to right.

The advantage of a complete tree is that pointers are not required to implement its structure. The position of a node is implicit in the structure.

In a complete tree there is a connection between the height *h* and the number of nodes *n*. For k=2 (a complete binary tree) we have:

Noting that the total number of nodes is the sum of the number of nodes at each level

$$1 + 2 + 2^2 + ... + 2^{(h-1)} < n <= 1 + 2 + 2^2 + ... + 2^h$$

$$2^h - 1 < n <= 2^{(h+1)} – 1$$

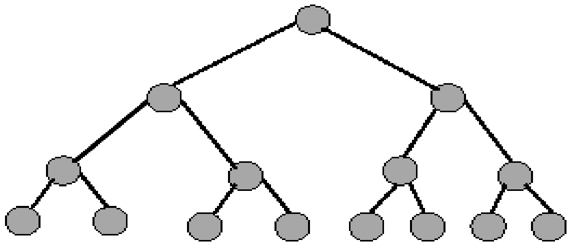
$$2^h < n+1 <= 2^{(h+1)}$$

Take log of both sides:

$$h < \log_2 (n+1) <= h + 1$$
$$h = \text{ceiling} (\log_2(n+1) - 1) .$$

Full Binary Tree:

A full binary tree of a given height h has $2^h – 1$ nodes. A full binary tree is as shown in fig. 7.10.

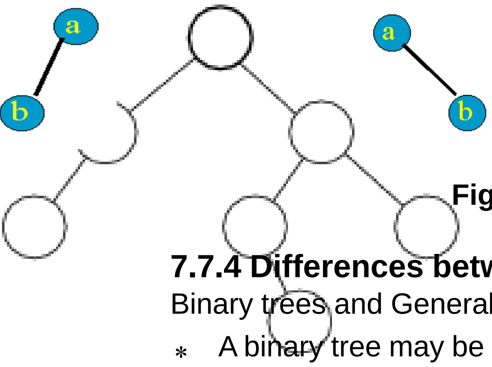


Height 4 full binary tree.

7.7.3 Extended Binary Trees: 2-Trees

A Binary tree T is said to be a 2- tree or an extended binary tree if each node n has either 0 or 2 children. The nodes with 2 children are called *internal nodes*, and the nodes with 0 children are called *external nodes*. The nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.

The term Extended binary tree comes from the following operation. Consider any binary tree T, and then T may be converted into a 2- tree by replacing each empty subtree by a new node. The following fig. 7.11 shows converting Binary tree into extended binary tree.



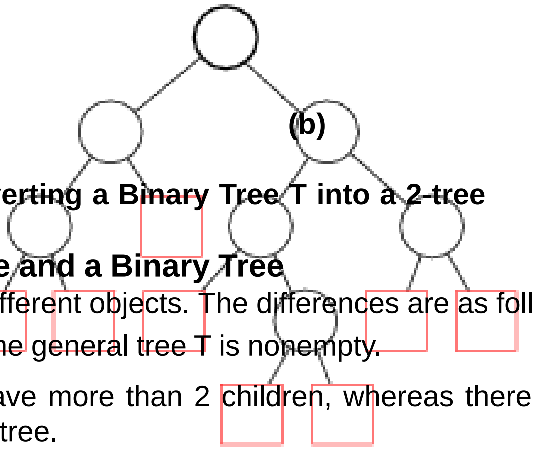


Fig. 7.11 Converting a Binary Tree T into a 2-tree

7.7.4 Differences between a Tree and a Binary Tree

Binary trees and General trees are different objects. The differences are as follows:

- A binary tree may be empty, but the general tree T is nonempty.

No node in a binary tree may have more than 2 children, whereas there is no limit on the number of children of a node in a tree.

The subtrees of a binary tree are ordered; those of a tree are not ordered.

If we consider the following two trees:

Different when viewed as binary trees.
Same when viewed as trees

b is left child of a b is right child of a

7.8 Representing Binary Trees

To represent a binary tree T in memory, mainly we have a 2 ways.

1. Linked representation.
2. Sequential representation.

7.8.1 Linked Representation Of Binary Trees

Consider a binary tree T. To represent a binary tree T in memory by using linked representation it uses three parallel arrays, INFO, LEFT, RIGHT and a pointer variable ROOT as follows. Each node N of T will correspond to a location K such that

INFO [k] contains the data at the node N.

LEFT [k] contains the location of the left child of node N.

RIGHT [k] contains the location of the right child of node N.

ROOT – it will contain the location of the root node. If any subtree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then ROOT will contain the null value.

- Each tree node is represented as an object whose data type is Tree Node.

Linked representation assumes the structure of a node as shown in fig. 7.12.

Fig. 7.12

Here Lc and Rc are two link fields to store the address of left child and right child of a node. DATA is the information content of the node.

Each node has three fields left, info, right fields. Info field to hold data.

Left field to point the left child. Right field to point right child.

If there is no left and right child, then left or right field points to NULL, which is indicated by dot.

The root node is pointed to by the pointer binary tree, which identifies the tree, without which it is not possible to access the tree at all.

Advantage:

It allows dynamic memory allocation. Hence, the size of the tree can be changed as and when needed. The only limitation is the availability of the required memory.

Disadvantage:

Linked representation requires extra memory to maintain the pointers.

Example:

Consider the binary tree T in fig. 7.7. A schematic diagram of the linked representation of T appears in fig. 7.13. Observe that each node is pictured with its three fields, and that the empty subtrees are pictured by using X for the null entries. Fig. 7.14 shows how this linked representation may appear in memory. The choice of 20 elements for the arrays is arbitrary. Observe that the AVAIL list is maintained as a one-way list using the array LEFT

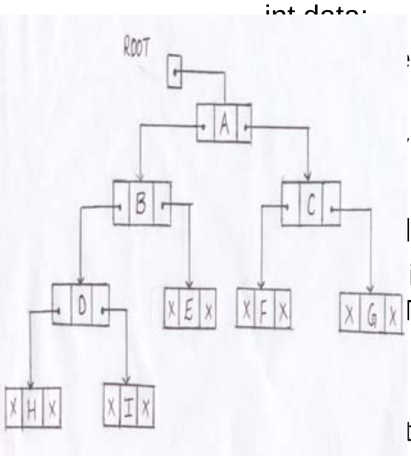
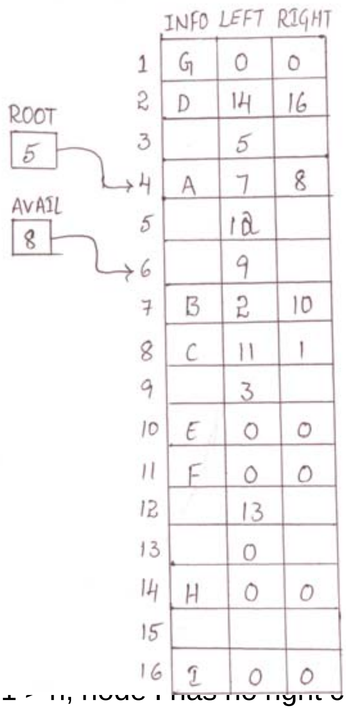


Fig 7.13


ROOT
5
AVAIL
8

	INFO	LEFT	RIGHT
1	G	0	0
2	D	14	16
3		5	
4	A	7	8
5		12	
6		9	
7	B	2	10
8	C	11	1
9		3	
10	E	0	0
11	F	0	0
12		13	
13		0	
14	H	0	0
15			
16	I	0	0

Fig. 7.14

General format of a binary tree node:

*

```
struct node Type
  {
    int data;
        e T
        ype
```

l F y Trees

ig a single array of size 2^k-1. Position 1 in the array Th sition I is located at position 2I and the right son at

t c

But nod rent.

Left chil

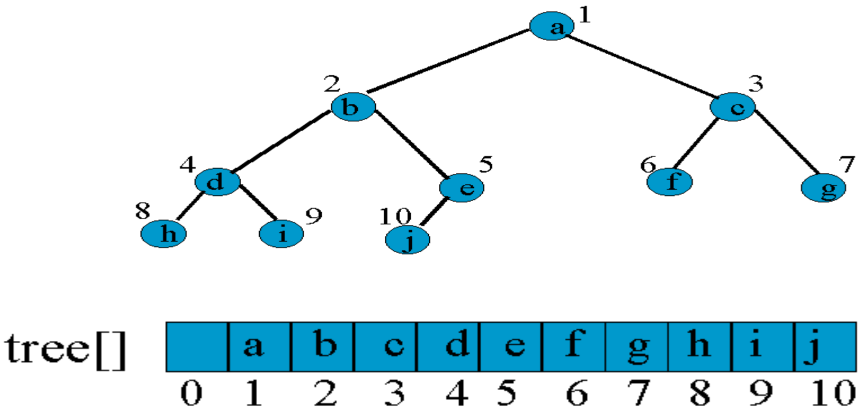
But if 2i

Right ch

But if 2i+ 1 > n, node i has no right child

Array representation

Number the nodes using the numbering scheme for a full binary tree
Store the node numbered i in tree[i].



tree[]		a	b	c	d	e	f	g	h	i	j
	0	1	2	3	4	5	6	7	8	9	10

Fig. 7.15

Node ‘a’ is stored in position 1 of the array binary tree.

* The left child of node ‘a’ is stored in position 2.
  The right child of node ‘a’ is stored in position 3.
  The left child of ‘b’ is stored in position 4.
  The right child of ‘b’ is stored in position 5.

Which one of the representation should be preferred?

- The implicit or array representation is simpler and it clearly saves storage space for trees, which are almost full, since it eliminates the need for left and right fields. However implicit representation can be used only when we know the maximum number of nodes in the tree. Since we can declare an array only if the size of it is known in advance.

- Linked representation requires the left, right fields for each node. However there is no restriction on the total no. of nodes on the tree but limited only by the amount of available memory.

7.9 Traversing Binary Trees

Traversing a binary tree involves visiting the root and traversing it’s left and right subtrees. The only difference among the methods is the order in which these three operations are performed.

Note: Visiting each node in a specified order is called *traversing*.

7.9.1 Trees and Recursion

Because of the recursive nature of trees, it is much easier to express tree algorithms using recursion rather than iteration. This is a major benefit of trees, because we can express some very complex operations efficiently with recursion. Consider an algorithm that traverses a tree printing the data element of the node.

According to the definition, a tree is either empty or has the structure:

Root

T (left)   T (right)

If the tree is empty, the traversal algorithm takes no action, i.e., the base case. If the tree is not empty, then the traversal algorithm must perform three tasks: it must print the data in the root node, and then traverse the left subtree and right subtree.

There are basically three standard ways of traversing a binary tree:

Preorder: 1) Process the root R.
2) Traverse the left subtree of R in preorder.
3) Traverse the right subtree of R in preorder.

Inorder: 1) Traverse the left subtree of R in inorder.
2) Process the root.
3) Traverse the right subtree of R in inorder.

Postorder: 1) Traverse the left subtree of R in postorder.
2) Traverse the right subtree of R in postorder.
3) Process the root.

In these 3 traversing methods the left subtree of R is always traversed before the right subtree. Sometimes these are called node-left-right (NLR) traversal, the left-node-right (LNR) traversal, and the left-right-node (LRN) traversal.

7.9.2 Traversal Algorithms

A *traversal algorithm* is a procedure for systematically visiting every vertex of an ordered binary tree. Many algorithms that use binary trees proceed in two phases.

1. Builds a binary tree, and
2. Traverse the tree.

Binary Tree Traversal in C:

We may implement the traversal of binary trees in C by recursive routines. The parameter to each routine is a pointer to the root node of a binary tree.

Preorder traversal:

```
Void pretraverse(NODEPTR tree)
  {
        if (tree != NULL)
        {
                printf(“%d \n”, tree->info);
                pretraverse(tree->left);
                pretraverse(tree->right);
        }
 }
```

Example:

Fig. 7.16

To do a traversal, we just make recursive calls to the algorithm. For example, the preorder traversal of the above tree is as follows:

Call 1) Processes F first, and then proceeds down to the left subtree of F.

Call 2) Next, we process this subtree in preorder. So we process the root first, which is C.
Call 3) Then, we proceed down to its left subtree and process that in preorder, so we process A. There is no left subtree or right subtree of A so call 3 is completed.
We return to call 2 where we have completed the left subtree traversal.

In preorder the right subtree traversal processes E (new call 3). Since this has no left or right subtree, new call 3 is completed.

Call 2 is also completed, so we return up to call 1 where we have finished the left subtree traversal.

Now, we have to traverse the right subtree in preorder (new call 2).
We process the root of the right subtree G in preorder. G has no left subtree, but it does have a right subtree, so we process that subtree in preorder (new call 3).

We process H and have completed call 3 because H has no subtrees. We have also completed call 2, and call 1. The preorder traversal is FCAEGH.

Inorder Traversal:

Inorder traversal will cause all the nodes to be visited in ascending order. Steps involved in Inorder traversal (recursion) are:

```
Void intraverse (NODEPTR tree)
  {
        if (tree!=NULL)
        {
                intraverse(tree->left);
                printf (“%d \n”, tree->info);
                intraverse(tree->right); }   }
```

Example:

In the same manner, the inorder traversal can be evaluated by using the algorithm for the given above tree. The inorder traversal is ACEFGH.

- **Postorder Traversal:**

```
Void posttraverse(NODEPTR tree)
  {
        if (tree!=NULL)
        {
                posttraverse(tree->left);
                posttraverse(tree->Right);
                printf("%d \n", tree->info);
        }
 }
```

The postorder for the given above tree is AECHGF.

7.9.3 Non-Recursive Implementation Of Traversals

Traversal Algorithms Using Stacks

This section discusses the implementation of the three standard traversals of T by using stack. Assume a binary tree T is maintained in memory by some linked representation.

TREE (INFO, LEFT, RIGHT, ROOT)

*

Preorder Traversal:

The Preorder algorithm uses a variable PTR (pointer), which will contain the location of the node N currently being scanned.

L (N) denotes the left child of node N

R (N) denotes the right child of node of N

An array Stack–hold the addresses of nodes

Algorithm: PREORD (INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. Algorithm does a preorder traversal of T.

```
1.   [Initially push NULL onto STACK, and initialize PTR.]
        Set TOP: =1,STACK [1]: =NULL and PTR: =ROOT.
2.   Repeat steps 3 to 5 while PTR! =NULL:
3.   Apply PROCESS to INFO [PTR]
4.   [Right child?]
      If RIGHT [PTR]! =NULL, then: [Push on Stack]
                 Set TOP: =TOP + 1
     STACK [TOP]: =RIGHT [PTR]
     [End of If structure.]
```

```
5.   [Left Child?]
       If LEFT [PTR]! =NULL, then:
     Set PTR: = LEFT [PTR]
       Else  [pop from Stack]
                 Set PTR: =STACK [Top]
                 TOP: =TOP – 1
        [End of If structure.]
      [End of step 2 loop.]
6.   Exit.
```

Inorder Traversal:

The inorder traversal algorithm also uses a variable pointer PTR, which will contain the location of the node N currently being scanned, and an array stack.

Algorithm: INORD (INFO, LEFT, RIGHT, ROOT)

```
1. [Push NULL onto STACK and initialize PTR.]
           Set top: =1
           STACK [1]: =NULL
           PTR:  =ROOT
2. Repeat while PTR! =NULL
     a) Set TOP: =TOP+ 1 and STACK [TOP]: =PTR   [Saves node]
     b) Set PTR: =LEFT [PTR].    [Updates PTR]
   [End of loop.]
3. Set PTR: =STACK [TOP] and TOP: =TOP –1.
             [Pops node from stack.]
4. Repeat steps 5 to 7 while PTR! =NULL.
5.  Apply PROCESS to INFO [PTR
6. If RIGHT[PTR]!=NULL then
           a)  set PTR:=RIGHT[PTR]
           b)  Go to step 3.
   [End of If Structure.]
7. Set PTR: =STACK [Top] and TOP: =Top-1
     [pops node]
     [End of step 4 loop.]
8. Exit.
```

Post Order Traversal:

The post order traversal algorithm is more complicated than the preceding two algorithms, because here we may have to save a node N in two different situations. We may distinguish between the two cases by pushing either N or its negative, -N onto STACK. A variable PTR is used which contains the location of the node N that is currently being scanned.

Algorithm: POSTORD (INFO, LEFT, RIGHT, and ROOT)

An array stack is used to temporarily hold the addresses of nodes.

```
1.  [Push NULL onto STACK and initialize PTR.]
    Set TOP:=1, STACK [1] :=NUL L and PTR: =ROOT.
2.  [Push left-most path onto STACK.]
    Repeat steps 3 to 5 while PTR! =NULL.
3.  Set TOP:=TOP + 1 and STACK[TOP]:=PTR.
     [Pushes PTR on STACK.]
4.  If RIGHT[PTR] !=NULL then [Push on Stack.]
    Set TOP:=TOP + 1 and STACK[TOP]:=-RIGHT[PTR].
    [End of If Structure.]
5.  Set PTR:=LEFT[PTR].[Updates pointer PTR.]
6.  Set PTR:=STACK[TOP] and TOP:=TOP-1;
    [Pops node from STACK.]
7.  Repeat while PTR>0
        a)  Apply PROCESS to INFO[PTR].
        b)  Set PTR:=STACK[TOP] and TOP := TOP –1.
            [Pops node from STACK.]
  [End of loop.]
8.  If PTR < 0, then
        a)  Set PTR:=-PTR.
        b)  Go to Step 2.
  [End of If Structure.]
9. Exit.
```

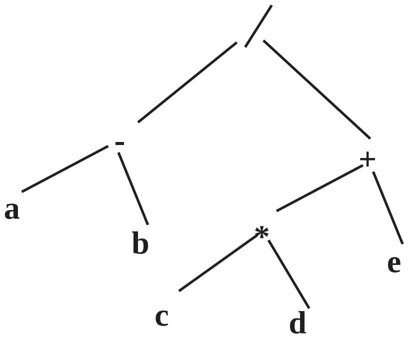
7.10 Binary Expression Tree

A special kind of binary tree in which:

- Each leaf node contains a single operand.
- Each inner vertex contains a single binary operator.
- The left and right subtrees of an operator node represent sub-expressions that must be evaluated before applying the operator at the root of the subtree.

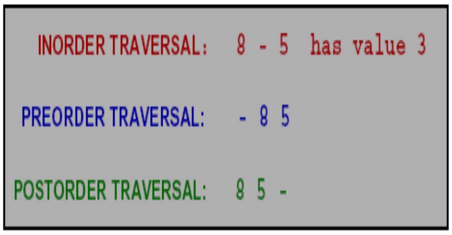
One of the interesting features of traversals is how they work on algebraic expressions. This is one of the reasons why they are so useful in compilers. If you take any tree representation of an algebraic expression where operators are the internal nodes and the operands are the leaves:

$$E = (a-b)/((c*d)+e)$$



A preorder traversal of fig.7.17 will give the prefix representation: /-ab+*cde
An inorder traversal gives infix: a-b/c*d+e and
The postorder traversal gives postfix: ab-cd*e+/.

Examples:

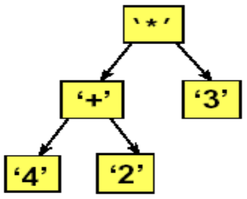


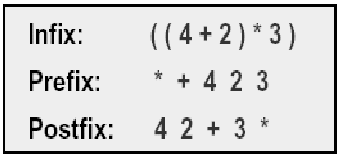
INORDER TRAVERSAL: 8 - 5 has value 3

PREORDER TRAVERSAL: - 8 5

POSTORDER TRAVERSAL: 8 5 -

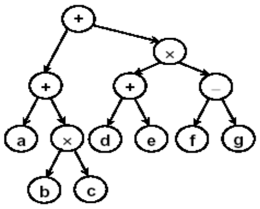
Fig 7.18





Infix: ((4+2)*3)
Prefix: * + 4 2 3
Postfix: 4 2 + 3 *

Fig 7.19



Preorder traversal: ++axbcx+de-
Inorder traversal: a+bxc+d+exf-g
Postorder: abcx+de+fg-x+

Fig 7.20

7.11 Summary

The term “tree” is used in Computer Science to denote a particular type of abstract data structure. Trees contain data in structures called nodes, which are in turn linked to other nodes in the tree.

- Hierarchical structures place elements in nodes that originate from a root.
- Nodes in a tree are subdivides into levels in which the top most level holds the root node.
  - Any node in a tree may have multiple successors at the next level. Hence a tree is a non-linear structure.
- Tree terminology with which you should be familiar:
  parent, child, descendants, leaf node, interior node, subtree.
- Binary trees
  A **binary tree** is a special type of tree where each node has exactly 0,1, or 2 children.
  data are located on relatively short paths from the root.
  a complete binary tree has the highest possible density.
- Traversing through a tree
  The most commonly used tree traversals are
  1) inorder (LNR)
  2) postorder(LRN)
  3) preorder(NLR)

7.12 Model Questions

1. Define tree?
2. What is binary tree?
3. What are the different traversals of the tree. Write the recursive procedures?
4. How the level of a node in a binary tree is defined.
5. Draw the tree representations to prefix expression
   a) * + ab + ac b) a + bc + de

SOME BINARY TREE OPERATIONS

- Determine the height.
- Determine the number of nodes.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.

7.13 REFERNCES

Theory And Problems Of Data Structures
Schaum’s Outline Series
Data Structures
Horowitz Sahani

K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science,
J.K.C College, GUNTUR.

Lesson 8

Trees -2

8.0 Objectives

After completion of this chapter the student able

- To learn usage of memory by using Threads.
- To describe searching an element from the Binary tree.
- To know sorting by using Heap data structure.
- Huffman's coding

Structure of the Lesson:

8.1 Introduction

In the previous lesson we have discussed binary trees. Now we will see more about binary trees. It helps to know about the header nodes and threads. It also covers binary search tree, by using this we can easily search an element from the binary tree. The heap data structure is useful to sort elements from the binary tree. Heap sort proceeds in two phases. First we must arrange the entries in the list so that they satisfy the requirements for a heap. Second, repeatedly remove the top of the heap and promote another entry to take its place

8.2 Header nodes

A binary tree T is maintained in memory by a linked representation. We can add a special node called a header node to the beginning of T. When this header node is used, the tree pointer variable, HEAD will point to the header node, and the left pointer of the header node will point to the root of T.

Suppose a binary tree T is empty. Then T will contain a header node, but the left pointer of the header node will contain the null value. Thus the condition

LEFT [HEAD] = NULL

will indicate an empty tree.

If a node has an empty subtree, then the pointer field for the subtree will contain the address of the header node instead of the null value. Accordingly no pointer will ever contain an invalid address. Thus the condition

LEFT [HEAD] = HEAD

will indicate an empty subtree

8.3 Threads

Traversing a binary tree is common operation. If we look at the linked representation of any binary tree, we notice that there are more null links than the actual pointers. There will be n+1 links and 2n total links. Some of the entries in the pointer fields LEFT and RIGHT will contain null elements.

To use the space more efficiently we can replace null entries by special pointers. The special pointers point to nodes higher in the tree. These special pointers are called *threads.* If the binary tree has such pointers it is called Threaded binary tree.

The threads in a threaded tree are indicated by dotted lines. In computer memory, an extra 1-bit TAG field may be used to distinguish threads from ordinary pointers or threads may be denoted by negative integers when ordinary pointers are denoted by positive integers.

There are many ways to thread a binary tree T, but each threading will correspond to a particular traversal of T. We can choose one-way threading or Two-way threading

8.3.1 Representation of Threaded Binary Tree

Before going to represent a threaded binary tree, we have to first decide to which node a thread should point. There are three ways to thread a binary tree that corresponding to inorder traversal is called *inorder threading* and that corresponding to preorder traversal is called preorder *threading*. Similarly, the post order threading is also possible.

- **Inorder Threading**

In the one-way threading of T, a thread will appear in the right field of a node and will point to the next node in the inorder traversal of T.

In the two-way threading of T, a thread will appear in the LEFT field of a node and will point to the preceding node in the inorder traversal of T. Further more the left pointer of the first node and the right pointer of the last node will contain the null value when T does not have a header node, but will point to the header node when T does have a header node as shown in fig 8.2

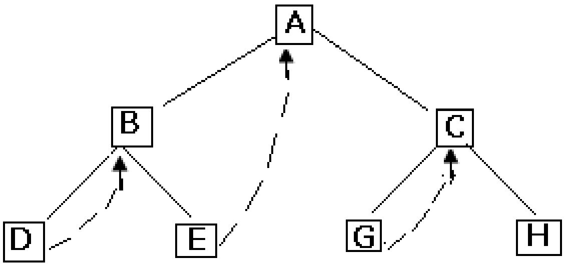


Fig. 8.1 Inorder Threading

Fig. 8.1 shows the binary tree with threads replacing NULL pointers in nodes with empty right subtrees. The threads are drawn with dotted lines to differentiate them from tree pointers. Note that the right most node in each tree still has a NULL right pointer, since it has no inorder successor. Such trees are called right in-threaded binary trees.

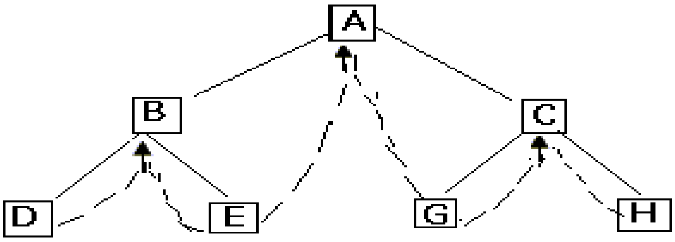


Fig. 8.2 Two-Way Inorder Threading

8.3.2 Advantages of Threaded Binary Tree

- The traversal operation is faster than that of its unthreaded version, because with threaded binary tree non-recursive implementation is possible, which can run faster and does not require the stack management.
- The second advantage is, we can efficiently determine the predecessor and successor nodes starting from any node. In case of unthreaded binary tree, this stack is more time consuming and difficult. For this case, a stack is required to provide upward pointing information in the tree whereas in a threaded binary tree, without using a stack mechanism the same can be carried out with the threads.

- Any node can be accessible from any other node. Threads are usually more to upward where as links are downward. Thus in threaded tree, one can move in either direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.
- Insertions into and deletions from threaded tree are although time consuming operations but these are very easy to implement.

Note: A pointer called thread is different from a tree pointer that is used to link a node to its left or right subtree.

8.4 Binary search trees

This section discusses one of the most fundamental problems in Data Structure and Algorithm Design, a binary search tree.

Suppose T is a binary tree. Then T is called a binary search tree if each node N of T has the following property: The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N. In addition, we need a DS that can process insertion and deletion.

8.4.1 Binary Search Tree

i) Is a binary tree that is “sorted” according to some particular order.

ii) If a node has value N, all values in its left sub-tree are less than or equal to N, and all values in its right sub-tree are greater than N.

- **Binary** because no node has more than two children.
- **Search** because the nodes are ordered for convenience in searching.

Operations those are generally encountered to manipulate this data structures are:

- Searching for a data.
- Inserting any data into it.
- Deleting any data into it.
- Traversing the tree.

Example:

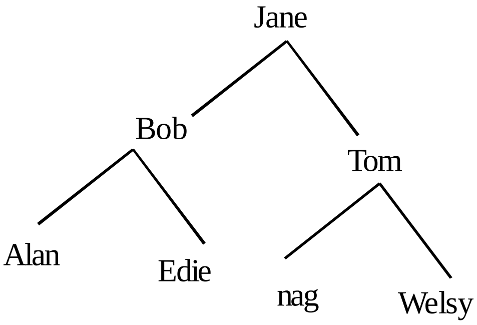


Fig. 8.3

Searching a binary search tree for an item is like doing a binary search. This is one of the real advantages of the tree structure. To find *Wesley* in a binary search tree requires 3 comparisons: Jane, Tom, and Wesley. A linear search would have required looking at every item in the list.

- **Advantage:** Searching for a data in a binary search tree is much faster than in arrays or linked lists.

Suppose an ITEM of information is given .The following algorithm finds the location of ITEM in the binary search tree T, or insets ITEM as a new node in its appropriate place in the tree.

a) Compare ITEM with the root node N of the tree.

i). If ITEM < N, proceed to the left child of N.

ii). If ITEM >N, Proceed to the right child of N.

b) Repeat Step (a) until one of the following occurs:

i) We meet a node N such that ITEM=N.
In this case the search is successful.

ii) We meet an empty subtree, which indicates search
is unsuccessful, and insert ITEM in place of the
empty subtree.

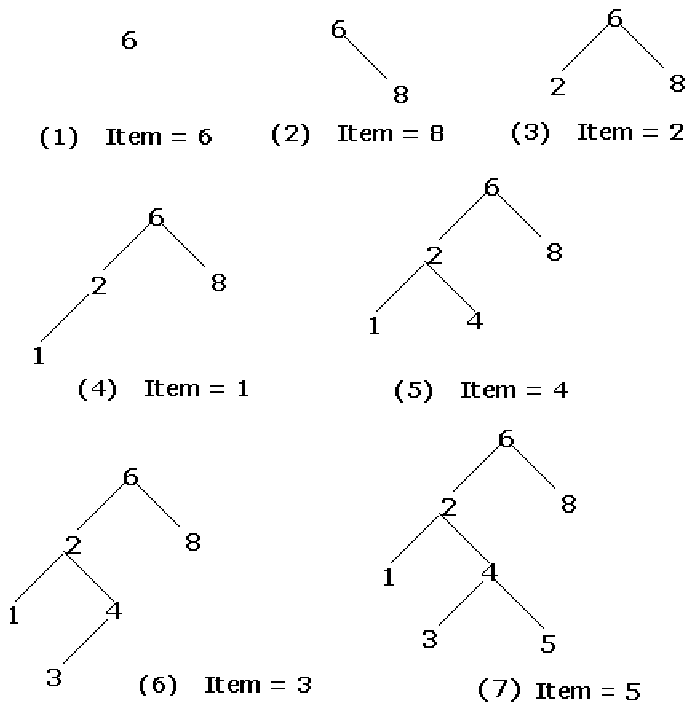
Example:

Suppose the following six numbers are inserted in order into an empty binary search tree.

6 8 2 1 4 3 5

The following fig. 8.4 shows the stages of the tree

8.4.2 Procedure for the Search



(1) Item = 6 (2) Item = 8 (3) Item = 2

(4) Item = 1 (5) Item = 4

(6) Item = 3 (7) Item = 5

Fig. 8.4

It performs search, insert, and deletion operations efficiently. If an array is used, an insertion or deletion requires that approximately half of the elements of the array be moved. Insertion or deletion in a search tree requires that only a few pointers be adjusted.

The following procedure finds the location of a given ITEM and its parent. The procedure traverses down the tree using the pointer PTR and the pointer SAVE for the parent node.

Procedure 8.1

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are 3 special cases:

i) LOC=NULL and PAR = NULL will indicate that the tree is empty.

ii) LOC! =NULL and PAR=NULL will indicate that ITEM is the root of T.

iii) LOC=NULL and PAR! =NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

```
1. If ROOT = NULL, then set LOC:=NULL and PAR:=NULL, and Return.
2. If ITEM = INFO[ROOT], then set LOC:=ROOT and PAR:=NULL, and Return.
3. If ITEM < INFO[ROOT], then
    Set PTR: =LEFT [ROOT] and SAVE: =ROOT.
   Else
       Set PTRL: =RIGHT [ROOT] and SAVE: =ROOT
4. Repeat steps 5 and 6 while PTR !=NULL
5. If ITEM=INFO [PTR], then
        Set LOC: =PTR and PAR: =SAVE, and Return.
6. If ITEM < INFO[PTR] then
        Set SAVE: =PTR and PTR: =LEFT [PTR]
   Else
        Set SAVE: =PTR and PTR: =RIGHT [PTR]
7. Set LOC:=NULL and PAR:=SAVE.
8. EXIT.
```

Observe that in step 6, we move to the left child or the right child according to whether ITEM < INFO [PTR] or ITEM > INFO[PTR].

- **Search And Insertion Algorithm**

Algorithm*:*

INBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

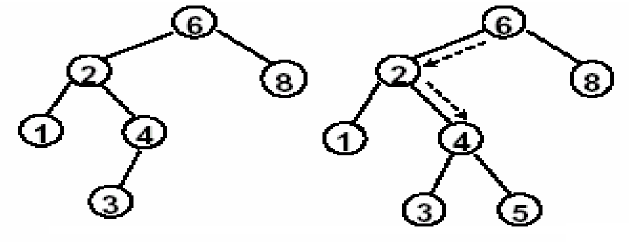
Algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T at location LOC.

```
1. Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
        [Procedure 8.1]

2. If LOC != NULL, then Exit.

3. [Copy ITEM into new node in AVAIL list.]
        a)  If AVAIL = NULL, then:
                        write OVERFLOW, and Exit.
a)  Set NEW:=AVAIL, AVAIL:=LEFT[AVAIL]
                        and INFO[NEW]:=ITEM.
    b)  Set LOC:=NEW, LEFT[NEW]:=NULL and RIGHT[New]:=NULL.
4. [Add ITEM to tree.]
        If PAR=NULL, then
                Set ROOT:=NEW
        Else  if ITEM <INFO[PAR], then
                Set LEFT[PAR]:=NEW
        Else:
                Set RIGHT[PAR]:=New
        [End of If Structure.]
5.  Exit.
```

Observe that in step 4 there are three possibilities (1) The tree is empty (2) ITEM is added as a left child and (3) ITEM is added as a right child.

The following algorithm searches a binary search tree and inserts a new record into the tree if the search is unsuccessful.

Example 8.3 Inserting 5 into a tree



Insertion of 5

8.4.3 Complexity of the Searching Algorithm

Suppose we are searching for an item of information in a binary search tree T. The number of comparisons is bounded by the depth of the tree.

Take n data items, A_1, A_2 . . . A_n, the items are inserted in order into a binary search tree T. The average running time f(n) to search for an item in a binary tree T with n elements is proportional to $\log_2$ n, that is, f(n)=O($\log_2$n).

8.4.4 Application of Binary Search Trees

A binary tree is a useful data structure when two-way decisions must be made at each point in a process.

Consider a collection n data items, A_1, A_2 . . . $A_{n,}$ to find and delete all duplicates in a list of numbers. To do this, compare each number with all those that precede it. It involves a large number of comparisons.

- **Algorithm A**:

Scan the elements from A_1, to A_N (from left to right).

a) For each element A_k, compare A_k with A_1, A_2 . . . A_{k-1}, i.e., compare A_k with those elements, which precede $A_{k.}$

b) If A_k does occur among A_1, A_2 . . . A_{k-1} then delete A_k.

After all elements have been scanned, there will be no duplicates.

Example:

Suppose Algorithm A is applied to the following list of 15 numbers:

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

Step 1: Compare each number with the previous number, find duplicates from the given list, we will get

A2=A5=10

A4=A8=12

A6=A14=11

A7=A11=20

Step 2: Delete duplicates from the list.

After deleting the duplicates the list is as follows:

A1=14
A2=10
A3=17
A4=12
A6=11
A7=20
A9=18
A10=25
A12=8
A13=22
A15=23

- **Time Complexity Of Algorithm A:**

The time complexity of the Algorithm is determined by the number of comparisons to identify the duplicates. Assume that the number of duplicates is less than the number of data items in the comparison.

Number of duplicates D < Number of data items N.

$A_1, A_2 \ldots A_{k,}$ sequence of elements, from which A_k will require approximately k-1 comparisons, since we compare A_k with items $A_1, A_2 \ldots A_{k-1.}$The number of comparisons required by algorithm A is

$0 + 1 + 2 + 3 + \ldots + (n-2) + (n-1) = (n-1)n/2 = O(n^2)$.

We can reduce the number of comparisons by using binary search tree. The procedure is as follows:

- The first number in the list is placed in a node that is the root of a binary tree with empty left and right subtrees.
- Each successive number in the list is compared to the number in the root. If it matches, we have a duplicate.
  - If it is smaller, examine the left subtree.
  - If it is larger, examine the right subtree.
  - If the subtree is empty, the number is not a duplicate and is placed into a new node at that position in the tree.
  - If the sub tree is non-empty, compare the number to the contents of the root of the subtree and the entire process is repeated with the subtree.

Algorithm B:

Build a binary search tree T using the elements $A_1, A_{2,} \ldots A_{k.}$ In building the tree, delete A_k from the list whenever the value of A_k already appears in the tree.

The total number of comparisons required by algorithm B is approximately n $\log_2$ n, that is, f(n) = O(n $\log_2$ n).

Example: Consider the following list of 15 numbers:

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

Apply algorithm B to this list of 15 numbers. The tree is as shown in fig. 8.6.

The number of comparisons algorithm B requires

0 +1 +1 + 2 + 2 + 3 + 2 + 3 + 3 + 3 + 3 + 2 + 4 + 4 + 5= 38

Algorithm A requires

0 +1+ 2 + 3 + 2 + 4 + 5 + 4 + 6 + 7 + 6 + 8 + 9 + 5 +10= 72

If *x* is stored at the root, the left subtree contains all keys less than *x* and the right subtree contains all keys greater than *x.*

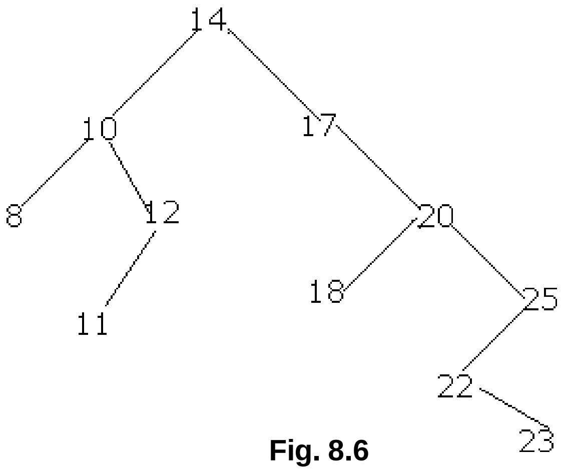


Fig. 8.6

8.4.5 Deleting Nodes in a Binary Search Tree

This section gives an algorithm, which deletes ITEM from the binary tree T. The deletion algorithm first uses Procedure 8.1 to find the location of the node N which contains ITEM and also the location of the parent node P(N). Deleting an ITEM from the tree depends on the number of children of node N. There are 3 cases

If the node to be removed

- is a leaf: remove with no problem.
- has no left(right) child: replace the node with right(left) child.
- has both children: find the smallest element in the right subtree and replace it with this element.

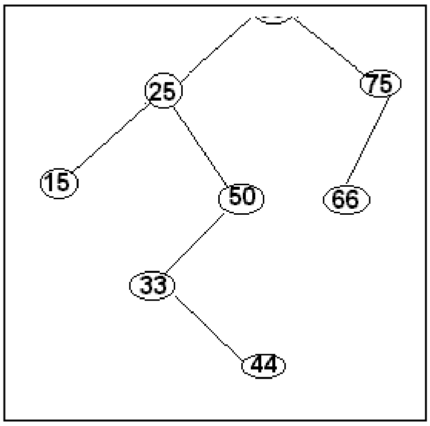
Case 1: N has no children. Then N is deleted from T by replacing the location of N in the parent node P(N) by the null pointer.

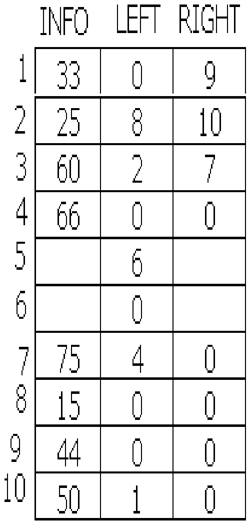
Case 2: N has exactly one child. Then N is deleted from T by replacing the location of N in P(N) by the location of the only child of N.

Case 3: N has 2 children. Let S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T and then replacing node N in T by the node S(N).

In all three cases, the memory space of the deleted node N is returned to the AVAIL list

ROOT	AVAIL
3	5

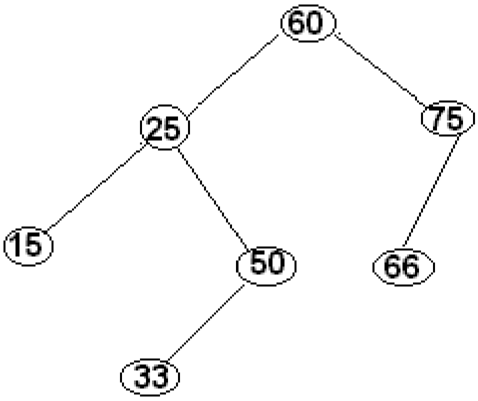


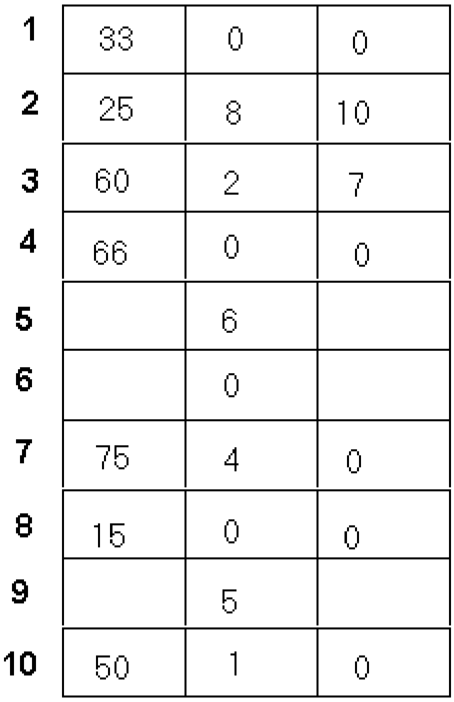


	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

a) Before Deletions **(b) Linked representation**

Fig. 8.7

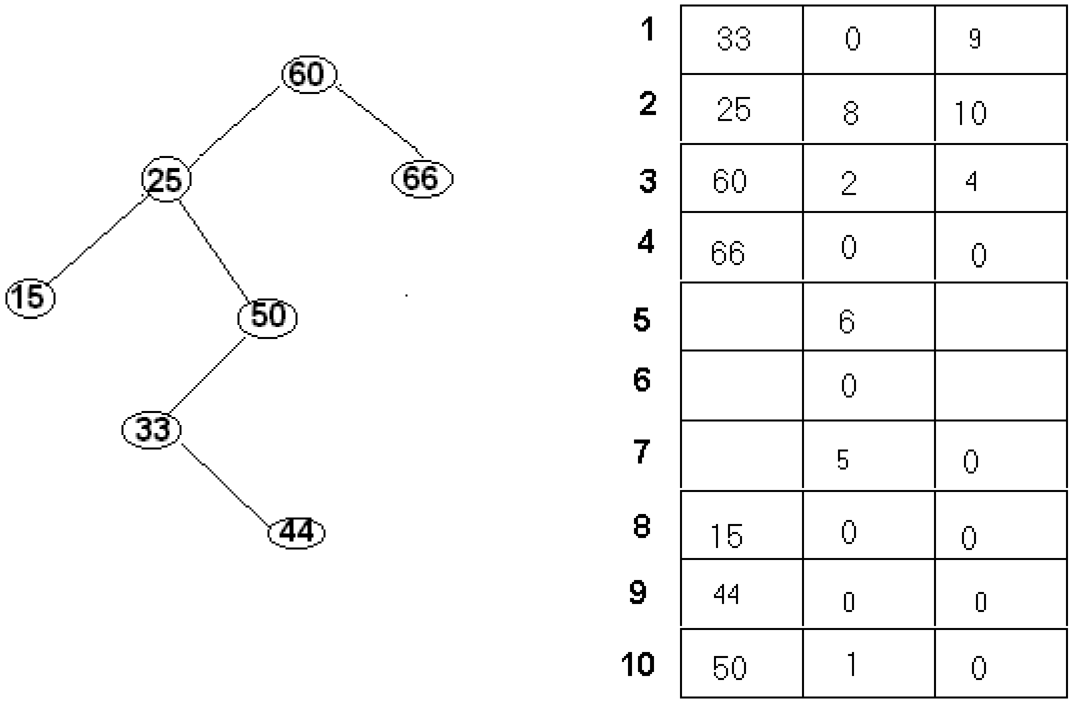




1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

(a)Node 44 is deleted **(b)Linked representation**

Fig. 8.8



1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	0
8	15	0	0
9	44	0	0
10	50	1	0

(a) Node 75 is deleted **(b) Linked representation**

Fig. 8.9

Procedure 8.3

CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, Where N does not have two children. The pointer Par gives the location of the parent of N or else PAR=NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N, or else CHILD=NULL indicates N has no children.

```
1. [Initializes CHILD]
    If LEFT [LOC]! =NULL and RIGHT [LOC]=NULL, then
        Set CHILD: =NULL
    Else if LEFT [LOC]! =NULL, then
        Set CHILD: =LEFT [LOC]
    Else
        Set CHILD: =RIGHT [LOC]
    [End of If Structure.]
```

```
 2. If PAR! =NULL, then
          If LOC=LEFT [PAR], then
                  Set LEFT [PAR]: =CHILD
          Else
                  Set RIGHT [PAR]: =CHILD
       [End of If Structure]
    Else:
          Set ROOT: =CHILD.
    [End of If Structure.]
3. Return.
```

Procedure 8.4

CASE B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR=NULL indicates that N is the root node. The Pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

```
  1. [Find SUC and PARSUC]
          a) Set PTR: =RIGHT [LOC] and SAVE: =LOC.
          b) Repeat while LEFT [PTR]! =NULL:
                  Set SAVE: =PTR and PTR: =LEFT [PTR]
          [End of loop.]
a)   Set SUC: =PTR and PARSUC: =SAVE.

 2. [Delete inorder successor, using Procedure CASE A.]
          Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)

 3. [Replace node N by its inorder successor.]
      a)      If  PAR !=NULL , then
              If LOC=LEFT [PTR], then
                  Set LEFT [PAR]: =SUC.
             Else
                  Set RIGHT [PAR]: =SUC.
            [End of If structure.]
      b)    Set LEFT [SUC]: =LEFT [LOC] and
            RIGHT [SUC]: =RIGHT [LOC].
  3.Return.
```

Example:

- **Removal Of Leaf Nodes**

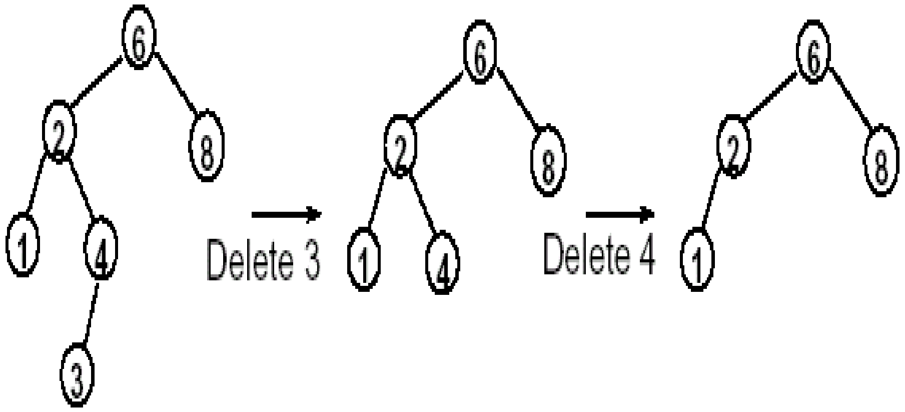


Fig. 8.10

- **Removing Nodes with Single Child**

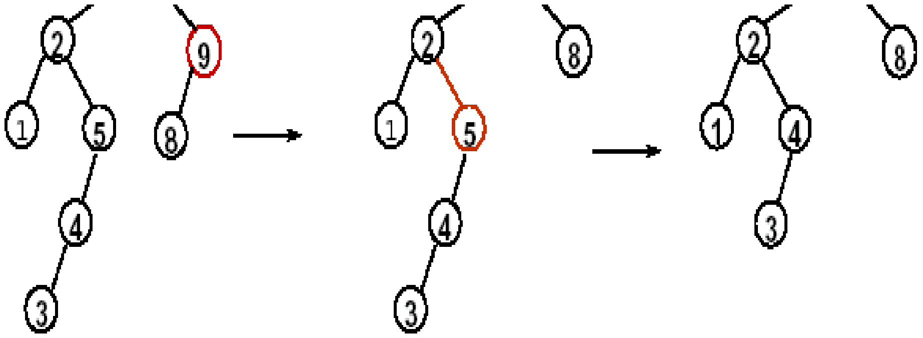


Fig. 8.11

Removing Nodes with Two Children

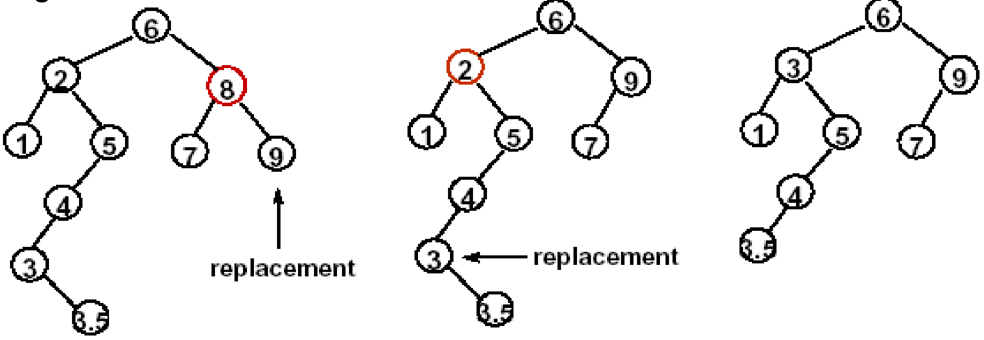


Fig. 8.12

Algorithm 8.5

DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

This algorithm deletes ITEM from the tree.

```
1. [Find the locations of ITEM and its parent, using procedure 8.1 ]
       Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2. [ITEM in tree?]
       If LOC=NULL, then write ITEM not in tree, and Exit.

3. [Delete Node containing ITEM]
       If RIGHT [LOC]! =NULL and LEFT [LOC]! =NULL, then
              Call CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
       Else
              Call CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
       [End of if structure]

4. [Return deleted node to the AVAIL list]
       Set LEFT [LOC]: =AVAIL and AVAIL: =LOC
5.   Exit.
```

8.5 Heap and Heap sort

This section discusses another tree structure, called a heap. The heap is used in sorting algorithm called heap sort.

The binary *heap data structure* is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

We represent heaps in level order, going from left to right. The array corresponding to the heap above is [25, 13, 17, 5, 8, 3].

8.5.1 Heap property

Suppose H is a complete binary tree with n elements. Then H is called a heap, or a Max Heap, if it satisfies the following properties:

i) For each node N in H, the value at N is greater than or equal to the value of each of the children of N.

ii) The value at N is greater than or equal to the value at any of the descendants of N.

Thus, the largest element in a heap is stored at the root.

Min Heap: The value at N is less than or equal to the value at any of the children of N.

An important consideration in using a heap is maintaining the “*heap property”* while keeping the tree complete. The only way to do this is to control carefully the inserts.

Example:

By the definition of a heap, all the tree levels are completely filled except possibly for the lowest level, which is filled from the left up to a point. Clearly a heap of height *h* has the minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of height *h* -1 and 2^h-1 nodes. Hence the minimum number of nodes possible in a heap of height *h* is 2^h. Clearly a heap of height h, has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height *h* and it has 2^{h+1}-1 nodes.

Following fig. 8.13 is not a heap, because it has only the heap property - it is not a complete binary tree. Recall that to be complete; a binary tree has to fill up all of its levels with the possible exception of the last one, which must be filled in from the left side.

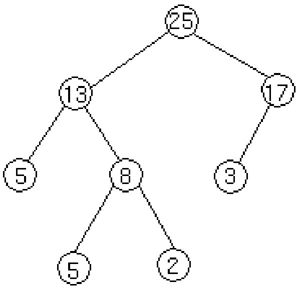


Fig. 8.13

8.5.2 Inserting Element in the Heap

Suppose H is a heap with N elements, and suppose an ITEM of information is given. We insert ITEM into the heap H as follows:

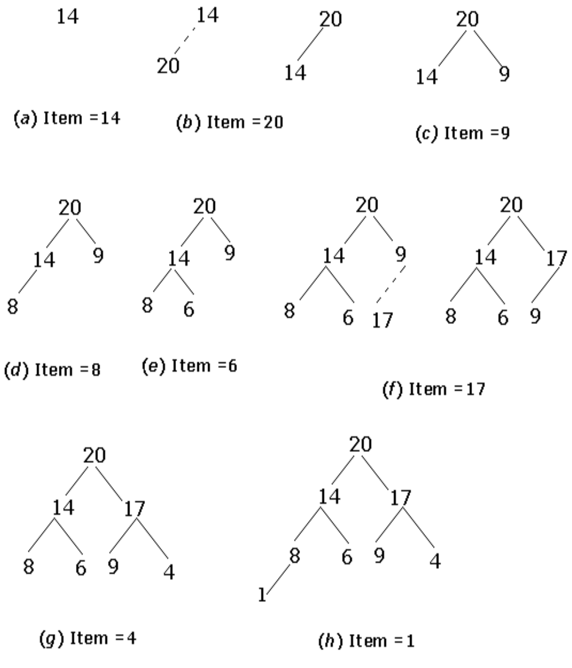
1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.

2) Then let ITEM rise to its "appropriate place" in H so that H is finally a heap.

Example:

Suppose we want to build a heap h from the following list of numbers.

14 20 9 8 6 17 4 1



(*a*) Item =14 (*b*) Item =20 (*c*) Item =9

(*d*) Item =8 (*e*) Item =6 (*f*) Item =17

(*g*) Item =4 (*h*) Item =1

Fig. 8.14

- **Insertion Procedure**

Procedure 8.6: INSHEAP (TREE N, ITEM)

A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure insets ITEM as a new element of H.PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

```
1.  [Add new node to H and initialize PTR.]
    Set N:=N+1 and PTR:=N
2.  [Find location to insert ITEM.]
    Repeat Steps 3 to  while PTR < 1.
3.   Set PAR:= PTR/2]. [Location of parent node.]
4.  If ITEM <= TREE[PAR], then;
    Set TREE[PTR]:=ITEM, and Return.
    [End of If structure.]
5.  Set TREE[PTR]:=TREE[PAR].[Moves node down.]
6.  Set PTR:=PAR.[Updates PTR.]
    [End of Step 2 loop.]
7. [Assign ITEM as the root of H.]
   set TREE[1]:=ITEM.
8.  Return.
```

Example: Inserting a new node

Suppose we have a heap as follows

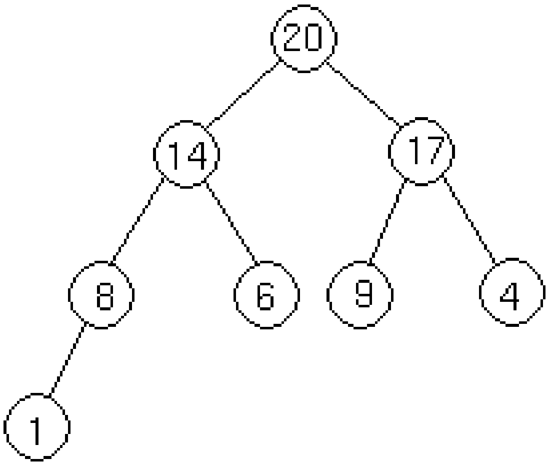


Fig. 8.15

Let's suppose we want to add a node with key 15 to the heap. First, we add the node to the tree at the next place available at the lowest level of the tree.

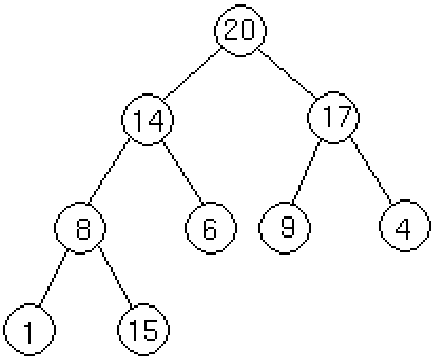


Fig. 8.16

Compare the new node key 15 with its parent, 8 < 15.

Now the heap property fails in this case, so swap the nodes with the key 8 and 15, now the tree is as follows

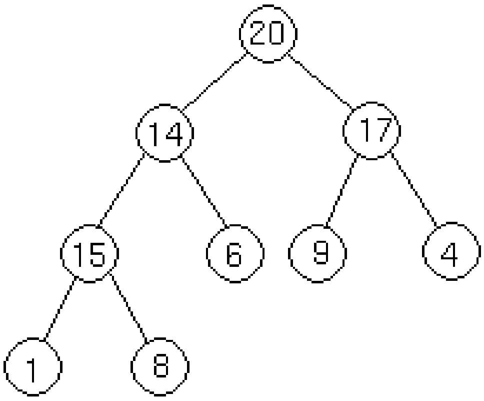
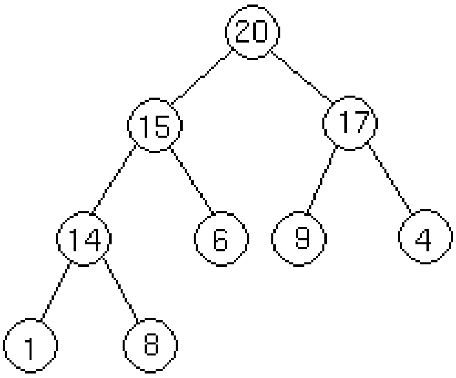


Fig. 8.17

Now we do the same thing again, comparing the new node to its parent. Since 14 < 15, we have to do another swap. Now the tree is as follows:



Now we are done, because 15 <=20.

Four basic procedures on heap are

1. Heapify, which runs in O(lg *n*) time.
2. Build-Heap, which runs in linear time.
3. Heap Sort, which runs in O(*n* lg *n*) time.
4. Extract-Max, which runs in O(lg *n*) time

- **Maintaining the Heap Property**

Heapify is a procedure for manipulating heap data structure. It is given an array A and index i into the array. The subtree rooted at the children of A[*i*] are heap but node A[*i*] itself may possibly violate the heap property i.e., A[*i*] < A[2*i*] or A[*i*] < A[2*i* +1]. The procedure 'Heapify' manipulates the tree rooted at A[*i*] so it becomes a heap.

- **Outline of Procedure Heapify**

Heapify picks the largest child key and compare it to the parent key. If parent key is larger than child key then heapify quits, otherwise it swaps the parent key with the largest child key. So that now the parent key becomes larger than its children.

Note: The swap may destroy the heap property of the sub tree rooted at the largest child node. In this case, Heapify calls itself again using largest child node as the new root

8.5.3 Deleting the root of a Heap

Suppose H is a heap with N elements, and suppose we want to delete the root R of H. This is accomplished as follows:

1) Assign the root R to some variable ITEM.
2) Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
3) (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

Procedure 8.7: DelHEAP (TREE, N, ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE[1] of h to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

```
1. Set ITEM:=TREE [1]. [Removes root of H.]
2. Set LAST:=TREE[N] and N:=n-1.
        [Removes last node of H.]
3. Set PTR:=1, LEFT:=2 and RIGHT:=3.[Initializes pointers.]
4. Repeat Steps 5 to 7 while RIGHT <=N.
5. If LAST >=TREE[LEFT] and LAST >=TREE[RIGHT], then:
        Set TREE[PTR]:=LAST and Return.
[End of If Structure.]
```

```
6. IF TREE[RIGHT]<=TREE[LEFT], then:
                set TREE[PTR]:=TREE[LEFT] and PTR:=LEFT.
        Else:
                Set TREE[PTR]:=TREE[RIGHT] and PTR:=RIGHT.
  [End of If Structure.]

7. Set LEFT:=2*PTR and RIGHT:=LEFT + 1.
        [End of Step 4 loop.]

8. If LEFT=N and if LAST<TREE[LEFT], then set PTR:=LEFT.

9. Set TREE[PTR]:=LAST.

10. Return.
```

Step 4 loop repeats as long as LAST has a right child. Step 8, in which LAST does not have a right child but does have a left child (which has to be the last node in H).

8.5.4 Application to Sorting

Suppose an array A with N elements is given. The heap sort algorithm to sort A consists of the following two phases:

Phase A: Build a heap H out of the elements of A.
Phase B: Repeatedly delete the root element of H.

Since the root of H always contains the largest node in H, Phase B deletes the elements of A in decreasing order. The algorithm uses procedures 8.6 and 8.7.

Algorithm 8.8 HEAPSORT (A, N)

An array A with N element is given. This algorithm sorts the elements of A.

```
1. [Build a heap H, using procedure 8.6]
        Repeat for J=1 to N –1;
                Call INSHEAP(A, j, A[J+1])
        [End of loop.]
2. [Sort A by repeatedly deleting the root of H, using procedure 8.7]
        Repeat while N > 1:
          a) Call DELHEAP(A,N,ITEM);
          b) Set A[N+1]:=ITEM.
        [End of loop.]
3.Exit.
```

The purpose of step 2(b) is to save space. We can use another array B to hold the sorted elements of A and replace Step 2(b) by

```
                        Set B[N+1]:=ITEM
```

8.5.5 Complexity of Heap Sort

Suppose the heap sort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phase separately.

- **Phase A:** Suppose H is a heap. The number of comparisons to find the appropriate place of a new element ITEM in H cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number g(n) of comparisons to insert the n elements of A into H is bounded as follows:

  $$g(n)<=n \log_2 n$$

  The running time of *Phase A* of heap sort is proportional to n $\log_2$ n.

- **Phase B:** Suppose H is as complete tree with m elements, and suppose the left and right subtrees of H are heaps and L is the root of H. Observe that the reheaping uses four comparisons to move the node L one step down the tree H. Since the depth of H does not exceed $\log_2 m$, reheaping uses at most $4\log_2 m$ comparisons to find the appropriate place of L in the tree H. This means that the total number h(n) of comparisons to delete the n elements of A from H, which requires reheaping n times, is bounded as follows:

  $$H (n) <=4n \log_2 n$$

Accordingly, the running time of Phase B of heap sort is also proportional to $n\log_2 n$.

Since each phase requires time proportional to $n\log_2 n$, the running time to sort the n-element array A using heap sort is proportional to $n\log_2 n$.

$$\text{i.e, } f(n)=O(n \log_2 n).$$

This gives a worst-case complexity of the heap sort algorithm.

8.6 Path lengths; Huffman algorithm

An extended binary tree or 2-tree is a binary tree T in which each node has either 0 or 2 children. The nodes with 0 children are called external nodes, and the nodes with 2 children are called internal nodes. The following tree shows a 2-tree where the internal nodes are denoted by circles and the external nodes are denoted by squares. In any 2-tree, the number N_E of external nodes is 1 more than the number N_I of internal nodes; that is,

$$N_E = N_I + 1$$

For example, for the 2-tree in fig. 8.19 N_I=6,, and $N_E=N_I+1=7$.

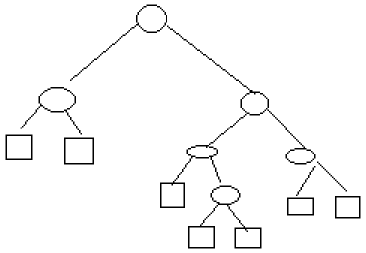


Fig. 8.19

In a 2-tree T algorithms the internal nodes represent tests and the external nodes represent actions. Accordingly, the running time of the algorithm may depend on the lengths of the paths in the tree.

The external path length L_E of a 2-tree to be the sum of all path lengths summed over each path from the root R of T to an external node. The internal path length L_I of T is defined analogously.

L_E= 2+2+3+4+4+3+3=21 and L_I= 0+1+1+2+3+2=9

Observe that L_I+2n = 9+2.6 = 9+12 = 21 = L_E

Where n=6 is the number of internal nodes, the formula

$$L_E=L_I + 2n$$

Is true for any 2-tree with n internal nodes.

Suppose T is a 2-tree with n external nodes, and suppose each of the external nodes is assigned a (nonnegative) weight. The (external) weighted path length p of the tree T is defined to be the sum of the weighted path lengths; i.e,

$$P = W_1L_1 + W_2L_2 + \ldots.. + Wn + L_n$$

Where Wi and Li denote the weight and length of an external nod Ni.

Consider now the collection of all 2-trees with n external nodes. Clearly the complete tree among them will have a minimal external path length L_E.

The following fig. 8.20, fig. 8.21, fig. 8.22 shows three 2-trees, T1, T2 and t3, each having external nodes with weights 2, 3, 5 and 11.the weighted path lengths of the three trees are as follows:

P1=2.2 + 3.2 + 5.2 + 11.2 = 42

P2=2.1 + 3.3 + 5.3 + 11.2=48

P3=2.3 + 3.3 + 5.2 + 11. 1 +36

The quantities P1 and P3 indicate that the complete tree need not give a minimum length p, and the quantities p2 and p3 indicate that similar trees need not give the same lengths

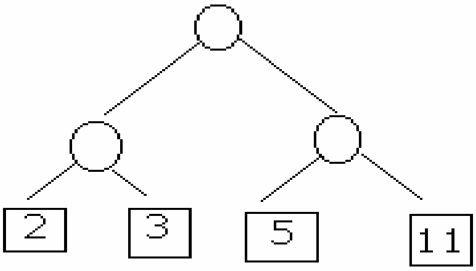


Fig. 8. 20

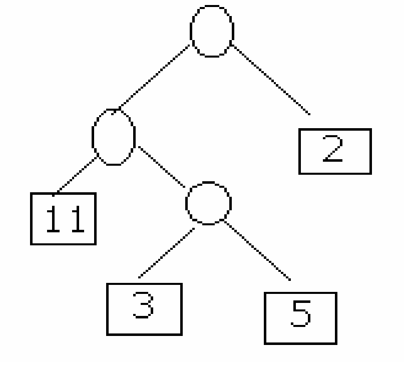


Fig. 8. 21

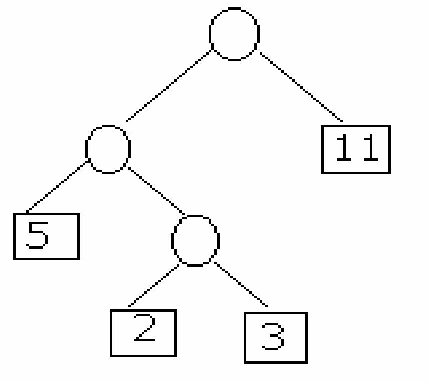


Fig. 8. 22

Among all the 2-trees with n external nodes and with the given n weights, find a tree T with a minimum –weighted path length. Huffman gave an algorithm, which we now state, to find such a tree T.

Observe that the Huffman algorithm is recursively defined in terms of the number of weights and the solution for one weight is simply the teee with one node. On the other hand, we use an equivalent iterated form of the Huffman algorithm constructing the tree from the bottom up rather than from the top down.

HUFFMAN’S ALGORITHM:

Suppose w1 and w2 are two minimum weights among the n given weights w1, w2, wn. Find a tree **T1** that gives a solution for the n-1 weights

W1+ W2, W3, W4…Wn.

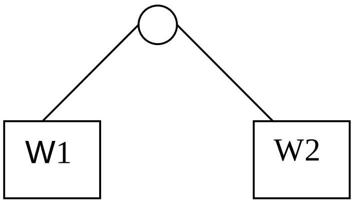


Fig. 8.23

Then, in the tree T1**,** replace the external node

W1 + w2

by the sub tree the new 2-tree T is the desired solution.

- **Application to coding**

Suppose a collection of n data items, A1, A2, A3…..An, are to be coded by means of strings of bits.

One way to do this is to code each item by an r-bit string where

2r-1<n<=2r

For example, a 48-character set is frequently coded in memory by using 6-bit strings. One can no use 5-bit strings, since 25<48<26.

Suppose the data items do not occur with the same probability. Then memory space may be conserved by using variable length strings where items which occur frequently are assigned shorter strings and items which occur infrequently are assigned longer strings. This section discusses a coding using variable-length strings that is based on the Huffman tree T for weighted data items.

Consider the extended binary tree T in fig 8.24 whose external nodes are the items U, V, W, X, Y and Z. Observe that each edge from an internal node to a left child is labeled by the bit 0 and each edge to a right child is labeled by the bit 1.The Huffman code assigns to each external node the sequence of bits from the root to the node.

Thus the tree T in fig. 8.24 determines the following code for the external nodes:

U:00 V:01 W:100 x:1010 y:1011 z:11

This code has the "prefix" properrty; i.e., the code of any items is not an initial substring of the code of any other item. This means there cannot be any ambiguity in decoding any message using a Huffman code

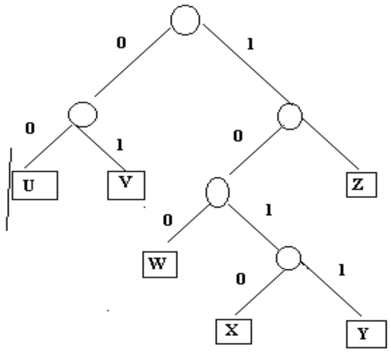


Fig. 8.24

8.7 Summary

- A binary search tree stores data by value instead of position

  The simple rules are = return

  < go left

  > go right

Until finding a NULL subtree, build a binary search tree that dose not allow duplicate values.

- The insertion algorithm can be used to define the path to locate a data value in the tree.
- The removal of an item from a binary search tree is more difficult and involves finding a replacement node among the remaining values.
- Heap sort is a relatively simple algorithm built upon the heap data structure. Implementation requires additional space, but it is possible to do a heap sort in place.
- Heap sort is not a stable sort, so the original ordering of equal elements may not be maintained

8.8 Model Questions

1. What is a threaded binary tree and what is its main advantage over an ordinary binary tree?
2. Show the tree after inserting the following integers in sequence into an initially empty binary search tree.

   18 20 23 8 6 15

3. Write an algorithm for heap sort to sort the n elements.
4. Write about Huffman coding.

8.9 References

Theory and problems of Data Structures,

Schaum's Outline Series

Data Structures and Algorithms, Reading, MA: Addison-Wesley, 1983.

A. Aho, J. Hop croft, J.D. Ullman

Introduction to Algorithms, New York: McGraw-Hill, 1991.

T. Cormen, C. Leiserson, R. Rivest

K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science,
J.K.C College,GUNTUR

Lesson9

Sorting

9.0 Objectives

After completion of this lesson the student will be able to learn how to implement the following sorting and searching algorithms

- Insertion sort
- Selection sort
- Merge sort
- Radix sort
- Hashing

Structure of the Lesson:

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. Sorting a large list into numerical or alphabetical order is a common task in computing. The performance of a sorting algorithm can be measured by the time it takes to sort a list containing 'n' items.

Consider a file, each record in a file F can contain many fields. One field contains the unique value to determine the records in the file. Such a field k is called a *primary key*, and the value k_1, k_2, . . . in such a field are called *keys* or *key values*.

- Sorting the file F refers to sorting F with respect to a particular primary key.
- Searching in F refers to searching for the record with a given key value.

9.2 Sorting

It is the problem of taking 'n' items and *rearranging* them into the total order. There are different sorting algorithms to arrange the items in sorted order, they are:

- *Insertion* - Putting an element in the appropriate place in a sorted list yields a larger sorted list.
- *Exchange* - Rearrange pairs of elements, which are out of order, until no such pairs remain
- *Selection* - Extracts the largest element from the list. Remove it, and repeat.
- *Distribution* - Separate into piles based on the first letter and then sort each pile.
- *Merging* - Two sorted lists can be easily combined to form a sorted list.

Let A be a list of n elements A_1, A_2, . . . 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

$$A_{1 <=} A_{2 <=} A_{3 <=} . . . A_n$$

Example: **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! = 40320 ways that the numbers 11, 22 . . . 88 can appear in Data

9.2.1 Complexity Of Sorting Algorithms

The complexity of a sorting algorithm can be measured by the time it takes to sort a list containing n items. Each sorting algorithm S will be made up of the following operations, Where $A_{1,}$ $A_{2,}$ . . . 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 cases to consider the complexity of algorithm: the worst case and average case.

9.2.1 Lower Bounds

Suppose S is an algorithm which sorts n items a_1, a_2… a_n. Assume we have an decision tree T corresponding to the algorithm S. T is an extended binary search tree where the external nodes correspond to the n! ways that can appear n items in memory. The internal nodes correspond to the different comparisons that may take place during the execution of the algorithm S.

The following fig. 9.1 shows a decision tree for sorting n=3 items. Tree T has n! external nodes. 3! = 6 external nodes.

The values of D and E for the tree follow:

D=3 and E= 1/6 (2 + 3 + 3 + 3 + 3 + 2) = 2.667

The corresponding algorithm S requires at most D= 3 comparisons and, on the average E= 2.667 comparisons to sort the n =3 items

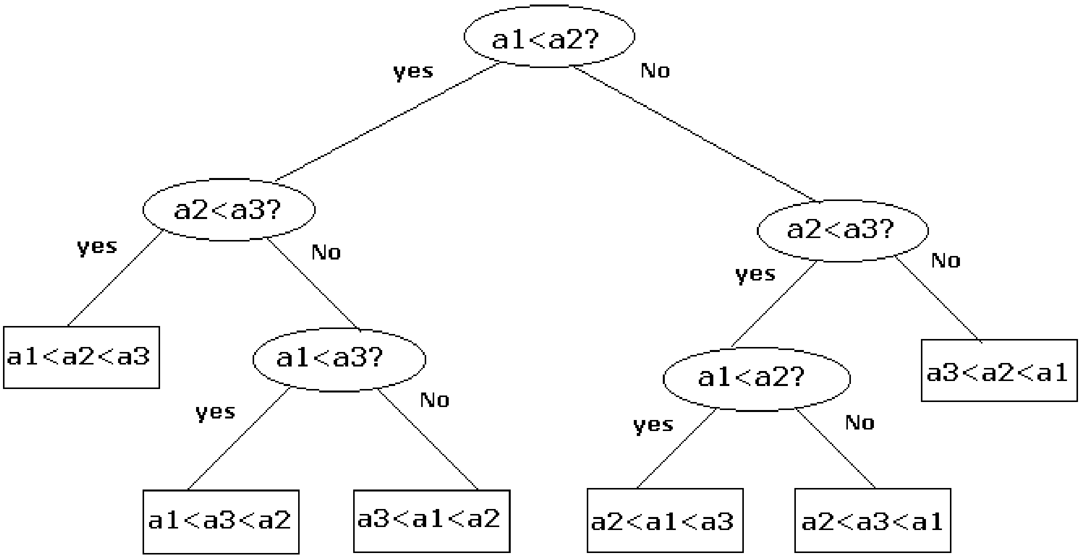


Fig. 9.1

∗ **Sorting Files:** Suppose a file F of records r_1, r_2…r_n is stored in memory. “Sorting F” refers to sorting file F with respect to some field K with corresponding values k_1, k_2…k_n. That is, the records are ordered so that

$$K_1 <= k_2 <= . . . k_n$$

The field k is called the *sort key.* (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

9.3 Insertion Sort

It is called insertion sort, because on the i^{th} pass, we insert the i^{th} element A [i] into its rightful place among A[1]…A[i-1] which is previously sorted. Suppose an array A with n elements A[1], A[2], …,A[N] is in memory. To sort n elements in the array A, the insertion sort algorithm

- First scans array A from A[1] to A[N], then
- Insert 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],
i.e. 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

■ Inserting A[K]

- 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 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 is an element A[J] such that A[J]<= A[K]. Otherwise we must check to see if we are comparing A[K] with A[1].

This condition can be accomplished by introducing a sentinel element

A[0] = $-\infty$ (or a very small number).

Algorithm 9.1: INSERTION (A, N)

This algorithm sorts the array A with N elements.

```
1.  Set A[0]= - $\infty$ [Initializes sentinel elements.]
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.
                              [Insert element in proper place.]
                 [End of Step 2 loop.]
        6. Return.
```

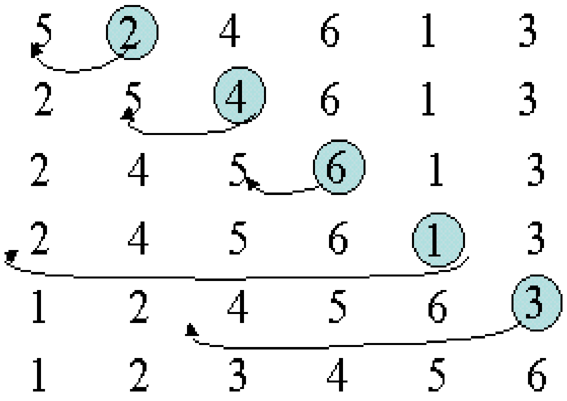
Observe that inner loop is controlled by the variable PTR, and there is an outer loop, which uses K as an index.

Example:

Suppose an array A contains 6 elements as follows:

5 2 4 6 1 3

The circled element indicates the A[K] in each pass of the algorithm, and the arrow indicates the proper place for inserting A[K] as shown in fig. 9.2.



5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Fig. 9.2

In the insertion sort, the first two numbers are compared. If the first number is larger than the second number then the second number is inserted in front of the first number. The process goes on by scanning to the right until a small number is found. It is inserted at the correct location, and the rest of the numbers slide one position to the right.

This process is repeated until the end of the list is reached. The list is then sorted. The basic operation is thus the insertion of a single element into a sequence of sorted elements so that the resulting sequence is sorted.

9.3.1 Complexity of Insertion Sort

The number f(n) of comparisons in the insertion sort algorithm can be easily calculated.

The *worst case* occurs when the array A is reverse order and the inner loop must use the maximum number K-1 of comparisons. Hence

$$F(n) = 1 + 2 + \ldots + (n-1) = n(n-1)/2 = O(n^2)$$

On the *average*, there will be approximately (K-1)/2 comparisons in the inner loop. Accordingly, for the average case, F (n) = 1/2 + 2/2 +… +(n-1)/2 = n(n-1)/4 = $O(n^2)$

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	n(n-1)/2 = $O(n^2)$	n(n-1)/4 = $O(n^2)$

9.4 Selection Sort

The most natural and easiest sorting algorithm is *selection sort*. The idea behind selection sort is the selection of the smallest (or largest) element from a sequence of elements. The selection-sort algorithm for sorting elements 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.

If elements are in an array, swap the first with the smallest element- thus only one array is necessary. If elements are in a linked list, we must keep two lists, one sorted and one unsorted, and always add the new element to the back of the sorted list.

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.

Pass 1. Find the location LOC of the smallest element 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:

Suppose an array A contains 5 elements as follows:

5 7 3 2 8

Fig.9.3 shows how the elements are interchanging in each pass. Observe that LOC gives the location of the smallest among A[K], A[K+1],….,A[N] during pass K.

Pass	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**
K=1, LOC=4	5	7	3	2	8
K=2, LOC=3	2	7	3	5	8
K=3, LOC=4	2	3	7	5	8
K=4, LOC=4	2	3	5	7	8
K=5, LOC=5	2	3	5	7	8
Sorted:	2	3	5	7	8

Fig. 9.3 Selection sort for n=5 items

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.

After comparing MIN with the last element A[N], MIN will contain the smallest among the elements A[K], A[K+1] … A[N] and LOC will contain its location.

Procedure 9.2: MIN (A, K, N, LOC)

An array A is in memory. This procedure finds the location LOC of the smallest element among A[K], A[K+1],…, A[N].

```
1.  Set Min := A[K] and LOC :=K. [ Initializes pointers.]
2.  Repeat for J=K+1, K+2, …, N:
    IF MIN> A[J], then : Set Min :=A[J] and LOC:=A[J] and LOC:=J. [End of loop.]
3.  Return.
```

Algorithm 9.3: (Selection Sort) SELECTION (A, N)

This algorithm sorts the array A with N Elements.

```
1. Repeat Steps 2 and 3 for K=1, 2, ….,N-1:
2. Call MIN(A, K, N, LOC).
3. [Interchange A[K] and A[LOC].]
```

```
            Set TEMP:=A[K] ,A[K]:=A[LOC] and
            A[LOC] :=TEMP.
            [End of Step 1 loop.]
        4.  Exit.
```

9.4.1 Complexity of Selection sort

First note that the number f(n) of comparisons in the selection sort algorithm is independent of the original order of the elements. MIN (A, K, N, LOC) requires n-k comparisons.

That is, there are n-1 comparisons during pass 1 to find the smallest element, there are n-2 comparisons during pass 2 to find the second smallest element, and so on. Accordingly,

$$F(n)=(n-1) + (n-2) + \ldots + 2 +1 = n(n-1)/2 =O(n^2)$$

Selection sort will give the best performance for large elements with minimum sorted items. It requires fewer moves than an insertion sort.

Algorithm	**Worst Case**	**Average Case**
Selection Sort	n(n-1)/2=O(n2)	$n(n-1)/2=O(N^2)$

9.5 Merging

Suppose A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with n = r + s elements is called *merging*.

Merging is the process of combining two or more sorted sublists into a third sorted list.

To do this place the elements of B after the elements of A and then use some sorting algorithm on the entire list. To merge the elements of A and B use the following Algorithm 9.4 in this section.

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 :=A + 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
```

9.5.1 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. Accordingly, the number f(n) of comparisons cannot exceed n:

$$f(n) <= n = O(n)$$

- **Non-regular Matrices**

Suppose A, B and c are three 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

$$UBA= LBA + r - 1 \text{ and } UBB=LBB + s-1$$

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.
```

This procedure is called MERGE, where as Algorithm 9.4 is called Merging.

- **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. We 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.

Example:

Suppose A has 5 elements and suppose B has 100 elements. The merging A and B by Algorithm 9.4 use approximately 100 comparisons. On the other hand, only approximately log100=7 comparisons are needed to find the proper place to insert an element of A into B using a binary search. Hence only approximately 5.7=35 comparisons are need to merge A and B using the binary search and insertion algorithm.

The binary search and insertion algorithms do not take into account the fact that A is sorted. Accordingly, the algorithm may be improved in two ways as follows.

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], 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], the location of all words with the same effect.

9.6 Merge-Sort

Merge-sort on an input sequence s with n elements consists of three steps:

- **Divide:** Partition S into two sequences s1 and s2 of about n/2 elements each.
- **Recur:** Recursively sort s1 and s2.
- **Conquer:** Merge s1 and s2 into a unique sorted sequence.

Example:

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   30,77
-----   -----   -----   -----   -----   -----   -----
```

Pass 2: Merge each pair of elements 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.

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 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. The algorithm requires at most log n passes. Each pass merges a total of n elements, and each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

f(n)<=n log n

The main drawback of merge-sort is that it requires an auxiliary array with n elements

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 is useful to alphabetize 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 start with "A", the second class consists of those class "B", and so on. During the second pass, each class is alphabetized according to the second letter of the name and so on. For example, more than 12 letters, the names are alphabetized, with at most 12 passes.

Algorithm

- Sort by the least significant digit first (counting sort) Numbers with the same digit go to same bin.
- Reorder all the numbers: the numbers in bin 0 precede the numbers in bin 1, which precede the numbers in bin 2, and so on.
- Sort by the next least significant digit.
- Continue this process until the numbers have been sorted on all k digits.

Example:

Suppose 8 cards are punched as follows:

275, 087, 426, 061, 509, 170, 677, 503

Given to a card sorter, the numbers would be sorted in three phases, as shown in fig. 9.4

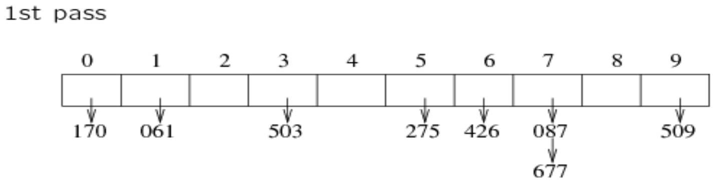


Fig. 9.4

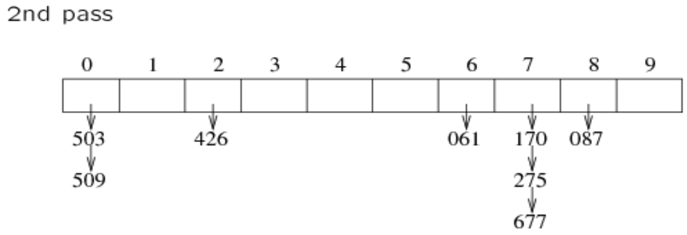


Fig. 9.5

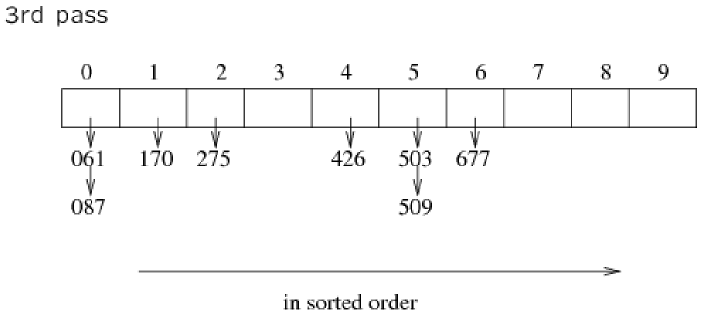


Fig. 9.6

a) In the first pass, the unit digits are sorted into pockets. The cards are collected pocket by pocket, from pocket 9 to pocket 0. The cards are now input to the sorter.

b) In the second pass, the tens digits are sorted in the 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:

061,0 87, 170, 275, 426, 503, 509, 677

Thus the cards are now sorted.

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

$$C \leq 8 * 3 * 10$$

The 8 comes from the eight 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 $A_1, A_2 \dots A_n$ 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 Ai is represented by means of s of the digits:

$$\mathbf{A_i = d_{i1} d_{i2} \dots d_{is}}$$

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) <= 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 it needs d * n memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. We can minimize this drawback by using linked lists rather than array to store the items during a given pass even though it requires 2 * n memory locations.

9.8 Summary

In this chapter we have discussed different sorting and searching algorithms. These are very helpful to arrange the elements in an alphabetical order.

Insertion sort is useful when the problem size is small. In selection sort each time we are selecting smallest element, and placing it into it's correct position. Merging is useful to combine two sorted sets. Merging and merge-sort both are different. Merge-sort using dividing and conquer approach to sort the elements.

9.9 Questions

1. Write an algorithm for Insertion sort.
2. Write an algorithm for selection sort.
3. Sort the following numbers by using merge-sort.
4. Sort the following numbers by using radix sort

9.10 References

Theory and problems of data structures

Schaum's outline series

Data structures

Horowitz Sahani

K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science,
J.K.C College, GUNTUR.

Lesson10

Searching

10.0 Objectives

After completion of this chapter the student able to know about the following concepts:

- What is Hashing
- Defining hash function
- To describe collision resolution strategies
- Rehashing

Structure of the Lesson:

10.1 Introduction

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. Choosing one algorithm depends on the properties of the data and the operations performing on the data.

10.1 Introduction

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. Choosing one algorithm depends on the properties of the data and the operations performing on the data.

10.2 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 or does not belong to S. The searching algorithm that is used depends mainly on the type of data structure 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 related searching, usually to insert ITEM in the table or delete we have to search for the location of the ITEM. The insertion or deletion also requires a certain amount of execution time. The execution time depends on the type of data structure that is used to implement.

Generally there is a trade off between data structures with fast searching algorithms and data structures with fast modification algorithms. This situation is illustrated below,

(1) *Sorted array*: we 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 be used when there is a great deal of searching but only very little data modification.

(2) *Linked list:* Here we can 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, it searches only a certain path P in the tree T, on the average it requires only O(log n) comparisons. Furthermore, the tree T is maintained in memory by a linked list representation, so only certain pointers need be changed after the location of the insertion or deletion is found. The main draw back 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.

- **Searching Files, Searching Pointers**

Suppose a file F of records R_1, R_2, . . . 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. To simplify the searching use an auxiliary sorted array of pointers. Binary search can be used to quickly find the location LOC of the record with the given key.

10.3 Hashing

The implementation of hash tables is frequently called hashing. Hashing is a technique used for performing insertion, deletions and finds the constant average time.

Generally the search for an identified key is carried out via a sequence of comparisons. Hashing differs from this, the address or the location of an identifier x is obtained by computing some arithmetic functions. F(x) gives the address of x.

The memory available to maintain the symbol table is assumed to be sequential. This memory is referred to as the hash table HT. The hash table is partitioned into buckets HT(0),. . . HT(b-1). Each bucket is capable of holding s records. Thus a bucket is said to consist of s slots, each slot being large enough to hold 1 record. Usually s=1 and each bucket can hold exactly 1 record.

Hashing is divided into two parts:

1) Hash functions and
2) Collision resolutions

Hash index/value: A hash value or hash index is used to index the hash table (array). The hash index is an integer (to index an array). A hash function takes a key and returns a hash value/index.

10.3.1 Hash Functions

1. Takes some kind of data (the key) as input and produce an address (possibly modulo the table size) as output.
2. The resulting address is hash value possibly some (large) random integer.

Now we need a hash function/algorithm H that should be very fast and creates a good distribution of hash values so that the items (based on their keys) are distributed evenly through the array.

- **Mod function**

Stands for **modulo**

* When you divide x by y, you get a result and a remainder
* Mod is the remainder

8 mod 5 = 3

9 mod 5 = 4

10 mod 5 = 0

15 mod 5 = 0

* Thus for key-value mod M, multiples of M give the same result, 0
* But multiples of other numbers do not give the same result. Problem arises when we have two keys that hash in the same array entry this is called a *collision*. We can say collision and overflow occur simultaneously.

- **Defining Hash Function**

A hashing function H, transforms an identifiers k into a bucket address in the hash table. The following are the 3 ways of defining a hash function.

1. Mid Square

Key is multiplied by itself. Middle value of the square is used as the index. Then the hash function h is defined by

$$H(k) = l$$

Where l is obtained by deleting digits from both ends of k^2.

Example:

If the key is 12345, square of this key is 152199025.

If we go for 2-digit address, we select 4th and 5th position and we get 19 as the index.

2. Division

The integer key x is divided by table size m and the remainder is taken as the hash value. (The number m is chosen to be a prime number or a number without small divisors). The hash function H is defined by

$$H(x) = x \pmod{m} + 1$$

Example:

Choose a prime number m close to 99, such as m= 97. Then

H(3205) =4,

H(7148) = 67

H(2345) = 17

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) = x (mod m) + 1 to obtain:

H(3205) = 4 + 1 =5,

H(7148) 67 + 1 = 68,

H(2345) 17 + 1= 18

1. Folding

Key is broken into several parts of some length of the required address and then added. Final carry is ignored.

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

Example:

H(3205) maps to 32 + 05 =37

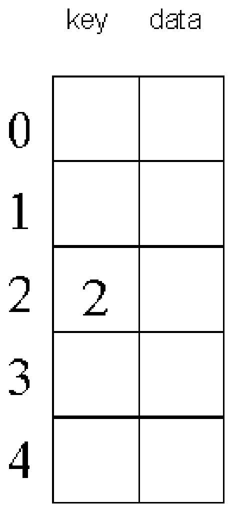
H(7148) = 71 + 48 = 19 ignored 1.

In this method the identifier k is partitioned into several parts, all but the last being of the same length. Then parts are then added together to obtain the hash address for k.

- **Hash Tables: Insert Example**

This gives an idea about inserting a new record into a table. The following fig. 10.1 shows inserting a new record into the empty table. Fig 10.2 shows inserting next record. Inserting next record is as shown in fig. 10.3.

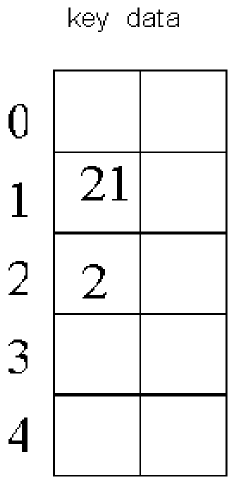
Insert 2



	key	data
0		
1		
2	2	
3		
4		

Fig. 10.1

Insert 21



	key	data
0		
1	21	
2	2	
3		
4		

Fig. 10.2

Insert 34

	key	data
0		
1	21	…
2	2	…
3		
4	34	…

Fig. 10. 3

The following fig. 10.4 shows how the collision occurs

Insert 54

There is a ***collision*** at array entry 4

???

Fig. 10.4

10.4 Collision Resolution

Suppose we want to add a new record R with key k to file F, but the memory location address H(k) is already occupied. This situation is called collision. 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*.

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 on the load factor λ.

- **Dealing with Collisions**

There are two ways to resolve collision:

1. **Hashing with Chaining ("Separate Chaining"):** Every hash table entry contains a pointer to a linked list of keys that hash in the same entry.

2. **Hashing with Open Addressing:** Every hash table entry contains only one key. If a new key hashes to a table entry, which is filled, systematically examine other table entries until you find one empty entry to place the new key.

10.4.1 Separate Chaining

To perform an insert, we traverse down the appropriate list to check whether the element is already in place. If the element turns out to be new, it is inserted either at the front of the list or at the end of list (which ever is easier).

To perform deletion, again the same hash function is used to produce address. At that address we have to traverse the linked list, find the element, delete it and do necessary modifications to links.

Coming to searching, again address is calculated basing on the hash function and search for the element in the list at that address.

Example: The problem is that, keys 34 and 54 hash in the same entry (4). We solve this *collision* by placing all keys that hash in the same hash table entry in a **chain** (linked list) **or bucket** (array) pointed by this entry as shown in fig. 10.5.

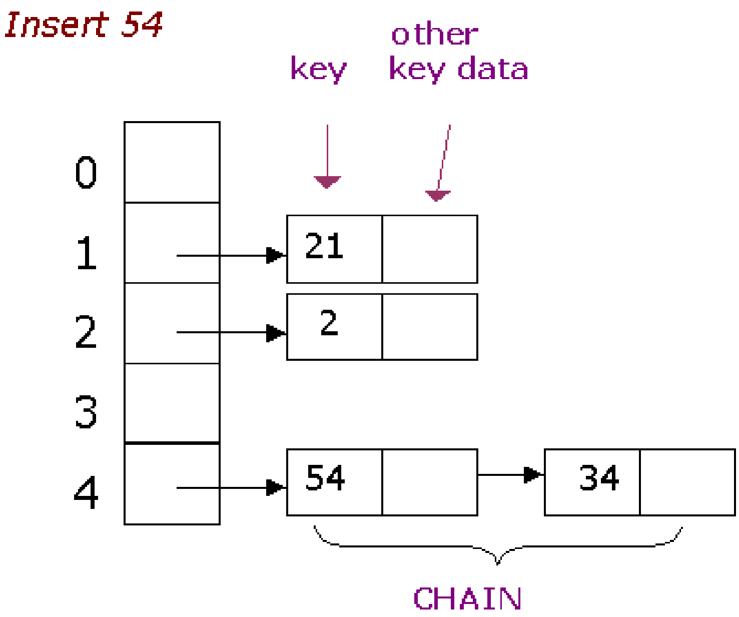


Fig. 10.5

Inserting 101 is as shown in fig. 10.6.

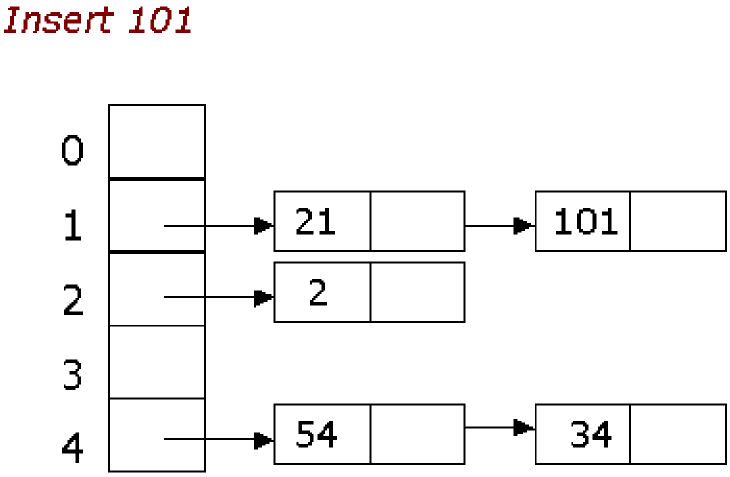


Fig. 10.6

- **Running time to insert/search/delete**
  - **Insert:** It takes O(1) time to compute the hash function and insert at head of linked list.
  - **Search:** It is proportional to max linked list length.
  - **Delete:** Same as search.

Disadvantage

Separate chaining requires pointers. This slows the algorithm down a bit because of the time required to allocate new cells, and also essentially requires the implementation of a record data structure. Open addressing hashing is an alternative to resolving collisions with linked lists.

10.4.2 Hashing with Open Addressing

We have studied hashing with chaining, using a list to store the items that hash to the same location. Another option is to store all the items (references to single items) directly in the table.

Open Addressing

Collisions are resolved by examining other table indexes i_0, i_1, i_2, … until an empty slot is located. The key is first mapped to an array cell using the hash function (e.g. key % array-size). If there is a collision find, then there are different algorithms to find (to probe for) the next array cell. Those are:

1. Linear
2. Quadratic
3. Double Hashing

1. Linear Probing

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.

Choose the next available array cell

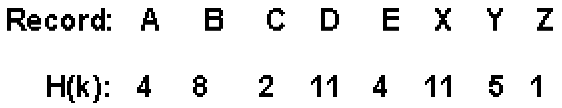
- First try array Index = hash value + 1.
- Then try array Index = hash value + 2.
- Be sure to wrap around the end of the array!
- Array Index = (array Index + 1) % array Size.
- Stop when you have tried all possible array indices.

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.

Note: If the array is full, you need to throw an exception or, better to resize the array.

Example:

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 as shown in fig. 10.7



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

Fig. 10.7

Suppose the 8 records are entered into the table T in the above order. Then the file F will appear in memory as shown in fig. 10.8.

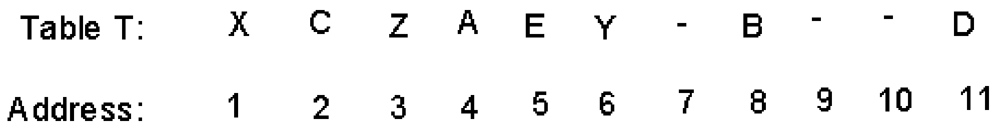


Table T:	X	C	Z	A	E	Y	-	B	-	-	D
Address:	1	2	3	4	5	6	7	8	9	10	11

Fig. 10.8

Y is the only record with hash address H(k)= 5, the record is not assigned to T[5], since T[5] has been filled by E because of 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= 1 + 1 + 1 + 1+ 2 + 2 + 2 + 3/8 = 13/≅1.6

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

U = 7 + 6 + 5 + 4 + 3 + 2 + 1 + 2 + 1 + 1 + 8/11

= 40/11≅3.6

It adds the number of probes to find an empty location for each of the 11 locations.

Disadvantage: clustering, it increases the average search time for a record.

2. Quadratic Probing

Linear probing uses a more complex function to calculate the next cell to try. The quadratic probing uses the following hash function to calculate the next cell.

Suppose a record R with key k has the hash address H(k)= h then 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.

Note: Quadratic probing eliminates primary clustering; elements that hash to the same position will probe the same alternative cell. This is known as secondary clustering.

3. Double Hashing

Apply a second hash function after the first. The second hash function, like the first, is dependent on the key. Secondary hash function must:

- be different than the first
- and obviously, not generate a zero.

Suppose a record R with key k has the hash address

$H(k) = h$ and $H'(k) = h' \neq m$.

Then we linearly search the locations with addresses

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

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

10.5 Rehashing

If the table gets too full, the running time for the operations will start taking too long and inserts might fail for open addressing hashing with quadratic resolution.

When we have to go for rehashing

- If the table half full
- When an insertion fails
- When table reaches a certain load factor

10.6 Summary

Hashing is a process; in this an item is placed into a structure based on a key-to-address transformation. Hashing is a way of performing optimal searches and retrievals. Collision: Once a space in the table is occupied, no other item can be placed there unless the existing item is deleted first.

Linear Probing- If *m* is the number of possible positions in the table, then the linear probe continues until either an empty spot is found, or until *m*-1 locations have been searched. One thing that happens with linear probing however is clustering.

Double Hashing is another method of collision resolution, but unlike the linear collision resolution, double hashing uses a second hashing function, which normally limits multiple collisions.

Open addressing; in this new locations are examined until an empty one is found.

10.7 Technical Terms

Hash table: Tables which can be searched for an item in **O(1)** time using a hash function to form an address from the key.

Hash function: Function which, when applied to the key, produces a integer which can be used as an address in a hash table.

Collision: When a hash function maps two different keys to the same table address, a collision is said to occur.

Linear probing: A simple re-hashing scheme in which the next slot in the table is checked on a collision.

Quadratic probing: A re-hashing scheme in which, a higher (usually 2^{nd}) order function of the hash index is used to calculate the address.

10.8 Questions

1. What is hashing?
2. Define hash table
3. What are the collision resolution techniques?
4. Explain the different hash functions with examples

10.9 References

Theory and problems of data structures

Schaum's outline series

Data structures

Horowitz Sahani

K.SIRISHA, M.Sc (Computer Science),
Lecturer, Dept. Of Computer Science
J.K.C College, GUNTUR