CONTENTS

Chapters		Pages
	SECTION-A	
1.	Programming Methodologies	1–11
2.	Concepts of Data Types and Data Structures	12–23
	SECTION-B	
3.	Concepts of Pointers	24–31
4.	Arrays, Stacks, Queues and Linked List	32–112
	SECTION-C	
5.	Trees	113–147
	SECTION-D	
6.	Searching and Sorting	148–195

SYLLABUS

DATA STRUCTURE THROUGH 'C'

SECTION A

1. Problem solving concepts, top down and bottom up design,structured programming.
2. Concept of data type and data structure, differences between data type and data structures, view of data structures at logical level, implementation level and application level, Built-in-data structures and user defined data structures.

SECTION B

3. Concept of dynamic variables, difference between static and dynamic variables, *concepts of pointer variables.*
4. Study of the following user defined data structures using static and variables.
   - — Built-in data structures like arrays, records.
   - — User defined data structures like stacks, queues, linked lists, circular linked lists, doubly linked list.

SECTION C

5. Non-linear data structures: trees, terminology of trees, concepts and applications of binary trees, *tree traversal techniques and algorithms.*

SECTION D

6. Sorting and searching algorithms and their efficiency considerations.

7. Considerations for choice of proper data structure.

NOTES

SECTION A

CHAPTER 1 PROGRAMMING METHODOLOGIES

★ LEARNING OBJECTIVES ★

1.1 Introduction
1.2 Characteristics of a Good Program
1.3 Techniques of Problem Solving
1.4 Structured Programming
1.5 Modular Programming
1.6 Top-down Programming
1.7 Bottom-up Programming
1.8 Summary
1.9 Test Yourself

1.1 INTRODUCTION

A **program** is a sequence of instructions written in a programming language. There are various programming languages, each having its own advantages for program development. Generally every program takes an input, manipulates it and provides an output as shown below :

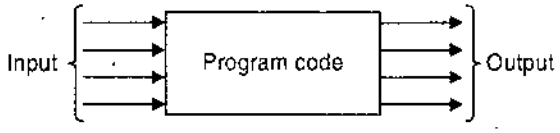


Fig. 1. *A conceptual view of a program.*

John Von Neumann proposed that if a program was stored in memory, program instructions could be easily changed just by loading a new program. Also as the program executed, it could easily change the instructions in memory. This is called the **stored program concept.**

NOTES

For better designing of a program, a systematic planning must be done. Planning makes a program more **efficient** and more **effective.** A programmer should use planning tools before coding a program. By doing so, all the instructions are properly interrelated in the program code and the logical errors are minimized.

There are various planning tools for mapping the program logic, such as **flowcharts, pseudocode, decision tables** and **hierarchy charts** etc. A program that does the desired work and achieves the goal is called an effective program whereas the program that does the work at a faster rate is called an efficient program.

The software designing includes mainly two things—*program structure* and *program representation.* The program structure means how a program should be. The program structure is finalised using top-down approach or any other popular approach. The program structure is obtained by joining the subprograms. Each subprogram represents a logical subtask.

The program representation means its presentation style so that it is easily readable and presentable. A user friendly program (which is easy to understand) can be easily debugged and modified, if need arises. So the programming style should be easily understood by everyone to minimize the wastage of time, efforts and cost.

Change is a way of life, so is the case with software. The modification should be easily possible with minimum efforts to suit the current needs of the organization. This modification process is known as **program maintenance.**

Flowcharting technique is quite useful in describing program structure and explaining it. The other useful techniques for actually designing the programs are :

(*i*) Modular programming

(*ii*) Top-down design (Stepwise refinement)

(*iii*) Structured programming.

1.2 CHARACTERISTICS OF A GOOD PROGRAM

The different aspects of evaluating a program are : efficiency, flexibility, reliability, portability and robustness etc. These characteristics are given below :

(*i*) **Efficiency.** It is of three types : programmer effort, execution time and memory space utilization. The high level languages are used for programmer efficiency. But, a program written in machine language or assembly language is quite compact and takes less machine time, and memory space. So depending on the requirement, a compromise between programmer's effort and execution time can be made.

NOTES

(*ii*) **Flexibility.** A program that can serve many purposes is called a flexible program. For example, CAD (Computer Aided Design) software are used for different purposes such as : Engineering drafting, printed circuit board layout and design, architectural design. CAD can also be used in graphs and reports presentation.

(*iii*) **Reliability.** It is the ability of a program to work its intended function accurately even if there are temporary or permanent changes in the computer system. Programs having such ability are known as reliable.

(*iv*) **Portability.** It is desirable that a program written on a certain type of computer should run on different type of computer system. A program is called *portable* if it can be transferred from one system to another with ease. This feature helps a lot in research work for easy movement of programs. High level language programs are more portable than the programs in assembly language.

(*v*) **Robustness.** A program is called *robust* if it provides meaningful results for all inputs (correct or incorrect). If correct data is supplied at run time, it will provide the correct result. In case the entered data is incorrect, the robust program gives an appropriate message with no run time errors.

(*vi*) **User friendly.** A program that can be easily understood even by a novice is called user friendly. This characteristic makes the program easy to modify if the need arises. Appropriate messages for input data and with the display of result make the program easily understandable.

(*vii*) **Self-documenting code.** The source code which uses suitable names for the identifiers is called self-documenting code. A cryptic (difficult to understand) name for an identifier makes the program complex and difficult to debug later on (even the programmer may forget the purpose of the identifier). So a good program must have self-documenting code.

1.3 TECHNIQUES OF PROBLEM SOLVING

Computer problem-solving can be summed up in one word—*it is demanding !* It is a combination of many small parts put together in a complex way, and therefore difficult to understand. It requires much thought, careful planning, logical accuracy, continuous efforts, and attention to detail. Simultaneously it can be a challenging, exciting, and satisfying experience with a lot of room for personal creativity and expression. If computer problem-solving is approached in this spirit then the chances of success are very bright.

For solving a problem on a computer a set of explicit and unambiguous instructions is written in a programming language. This set of instructions is called a *program.* An algorithm (step by step procedure to solve a problem in unambiguous finite number of steps) written in a programming language is a

program. So, an algorithm corresponds to a solution to a problem which is *independent* of any programming language.

Problem solving is a creative process which largely defies systematization and mechanization. Everyone acquires some problem-solving skills during his/her student life which he/she may or may not be aware of.

Some steps for problem solving improve the performance of the problem solver. No universal methods are available for it. Different people use different strategies. In simple words we can say logically that computer problem solving is about understanding.

1.3.1 Understanding of the Problem

When lot of efforts are made in understanding the problem we are dealing with, chances of success are also bright. We cannot hope to make useful progress in solving a problem until it is clear, what it is we are trying to solve. The preliminary investigation may be thought of as the *problem definition phase.* The problem definition defines what the problem is without any reference to the possible solutions. It is a simple statement, may be one to two pages and should sound like a problem. The problem definition should be in user language and it should be described from the user's point of view. It usually should not be defined in technical computer terms. As the analyst assigns the programs to different programmers module-wise, the programmers understand the problem given to them. The programmers define the problem of each program on a document and proceed for the next step. In simple words, a lot of care should be taken in working out precisely what must be done.

The problem solver should obtain information on the following three aspects of the problem after the analyses :

1. Input specification
2. Output specification
3. Special processing, if any.

1. *Input Specifications*

The input specifications should give the following information :

(*i*) Specific data values to be used as input in the program.

(*ii*) Input data format *i.e.*, order, spacing, accuracy and units.

(*iii*) The valid range of input data.

(*iv*) Restrictions, if any, on use of these data values and what to do if an input data is not accepted by the computer, should it be ignored or modified.

(*v*) The indication of end of input data (if specified by a special symbol).

2. *Output Specifications*

The output is obtained on executing a program. The output specifications must clearly define the values required and their formats etc. The output specifications must include the following information :

(*i*) The output data values required.

(*ii*) Output data format *i.e.*, precision (number of significant digits), accuracy, units, the position on the output sheet and suitable headings for making the output readable.

(*iii*) Amount of output required because the program has to be coded according to the number of output data values required.

3. *Special Processing, if any*

It means processing of input data under some conditions. If conditions are violated, certainly results are going to be incorrect. The processing under special condition(s) and the recovery action should be handled carefully. If the special processing conditions are ignored and left in the problem definition phase, it may be a costly affair later on.

So, in the problem definition phase, detailed information about input, output and special processing is gathered. These conditions are taken into consideration while solving the problem. The method of solution is not specified in this phase.

1.3.2 Step by Step Solution for the Problem

There are many ways to solve most of the problems and also many solutions to most of the problems. This situation makes the job of problem-solving a difficult task. When we have many ways to solve a problem it is usually difficult to recognize quickly which paths are likely to be fruitless and which paths may be productive.

A block often occurs after the problem definition phase, because people become concerned with details of the implementation *before* they have completely understood or worked out an implementation-independent solution. The problem solver should not be too concerned about detail. That can be taken into account when the complexity of the problem as a whole has been brought under control. The old computer proverb states, **"the sooner you start coding your program the longer it is going to take".**

An approach that often allows us to make a start on a problem is to take a specific example of the general problem we wish to solve and try to work out the mechanism that will allow us to solve this particular problem (*e.g.*, if you want to find the top scorer in an examination, choose a particular set of marks and work out the mechanism for finding the highest marks in this set).

This approach of focusing on a particular problem can often give us a platform we need for making a start on the solution to the general problem. It is not always possible that the solution to a specific problem or a specific class of

NOTES

problems is also a solution to the general problem. We should specify our problem very carefully and try to establish whether or not the proposed algorithm (step by step procedure in a finite number of steps to solve a problem) can meet those requirements. If there are any similarities between the current problem and other problems that we have solved or we have seen solved, we should be aware of it. In trying to get a better solution to a problem, sometimes too much study of the existing solution or a similar problem forces us down the same reasoning path (which may not be the best) and to the same dead end. Therefore, a better and wiser way to get a better solution to a problem is, try to solve the problem *independently.*

Any problem we want to solve should be viewed from a variety of angles. When all aspects of the problem have been seen, one should start solving it. Sometimes, in some cases it is assumed that we have already solved the problem and then try to work backwards to the starting conditions. The most crucial thing of all in developing problem-solving skills is practice.

Probably the most widely known and most often used principle for problem-solving is the *divide-and-conquer* strategy. The given problem is divided into two or more subproblems which can hopefully be solved more efficiently by the same technique. If it is possible to continue in this way we will finally reach the stage where the subproblems are small enough to be solved without further splitting.

This way of breaking down the solution to a problem has been widely used with searching, selection and sorting algorithms.

1.4 STRUCTURED PROGRAMMING

The main objectives of structured programming are :

- Readability
- Clarity of programs
- Easy modification
- Reduced testing problems.

The **goto** statement should be avoided so far as possible. The three basic building blocks for writing structured programs are given below :

1. Sequence Structure
2. Loop or Iteration
3. Binary Decision Structure

1. **Sequence Structure :**

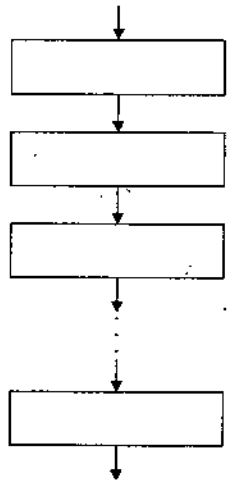


Fig. 2. *Sequence structure.*

It consists of a single statement or a sequence of statements with a single entry and single exit as show above.

2. **Loop or Iteration :**

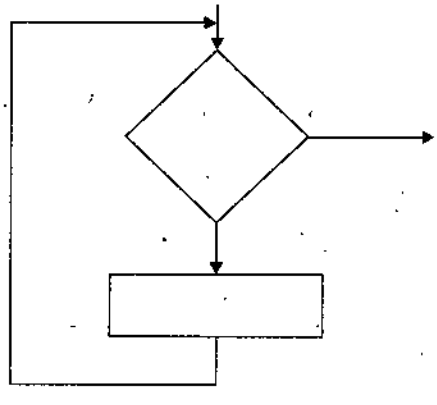


Fig. 3. *Loop or iteration.*

It consists of a condition (simple or compound) and a sequence structure which is executed condition based as shown above.

3. **Binary Decision Structure :**

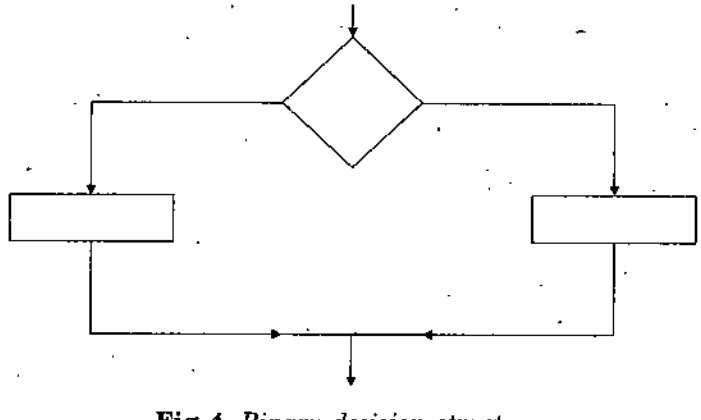


Fig 4. *Binary decision structure.*

It consists of a condition (simple or compound) and two branches out of which one is to be followed depending on the condition being true or false as shown above.

NOTES

1.5 MODULAR PROGRAMMING

Breaking down of a problem into smaller independent pieces (modules) helps us to focus on a particular module of the problem more easily without worrying about the entire problem. No processing outside the module should affect the processing inside the module. It should have only one entry point and one exit point. We can easily modify a module without affecting the other modules. Using this approach the writing, debugging and testing of programs becomes easier than a monolithic program. A *modular* program is readable and easily modifiable. Once we have checked that all the modules are working properly, these are linked together by writing the main module. The main module activates the various modules in a predetermined order. For example, Figure 5 illustrates this concept :

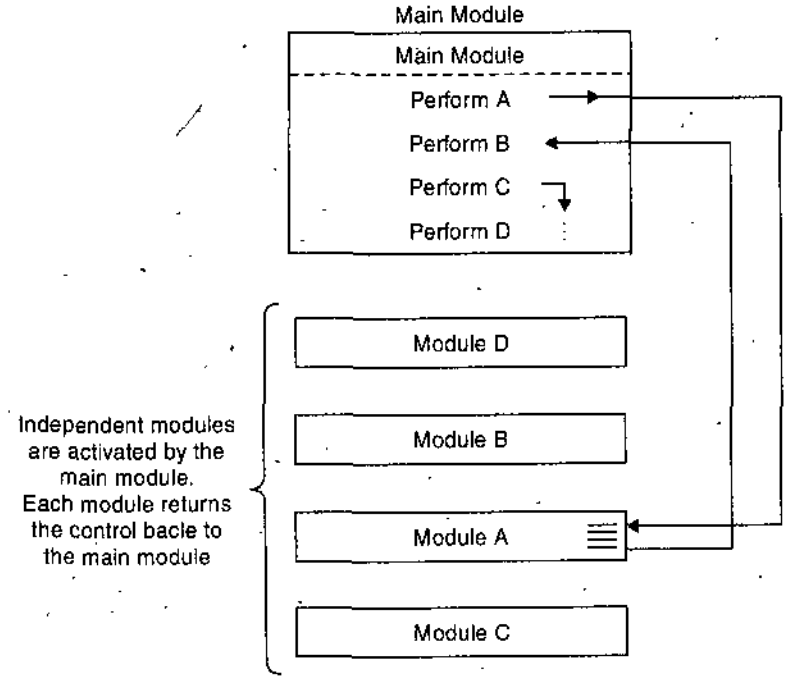


Fig. 5.

It must be noted that each module can be further broken into other submodules.

NOTES

1.5.1 Characteristics of Modular Approach

(*i*) The problem to be solved is broken down into major components, each of which is again broken down if required. So the process involves working from the most general, down to the most specific.

(*ii*) There is one entry and one exit point for each module.

(*iii*) In general each module should not be more than half a page long. If not so, it should be split into two or more submodules.

(*iv*) Two-way decision statement are based on IF..THEN, IF..THEN..ELSE, and nested IF structures.

(*v*) The loops are based on the consistent use of WHILE..DO and REPEAT..UNTIL loop structures.

1.5.2 Advantages of Modular Approach

(*i*) Some modules can be used in many different problems.

(*ii*) Modules being small units can be easily tested and debugged.

(*iii*) Program maintenance is easy as the malfunctioning module can be quickly identified and corrected.

(*iv*) The large project can be easily finished by dividing the modules to different programmers.

(*v*) The complex modules can be handled by experienced programmers and the simple modules by junior ones.

(*vi*) Each module can be tested independently.

(*vii*) The unfinished work of a programmer (due to some unavoidable circumstances) can be easily taken over by someone else.

(*viii*) A large problem can be easily monitored and controlled.

(*ix*) This approach is more reliable.

(*x*) Modules are quite helpful in clarification of the interfaces between major parts of the problem.

1.6 TOP-DOWN PROGRAMMING

Program development includes designing, coding, testing and verification of a program in any computer language. For writing a good program, the top down design approach can be used. It is also called **systematic programming** or **hierarchical program design** or **stepwise refinement**. A complex problem is broken into smaller subproblems, further each subproblem is broken into a number of smaller subproblems and so on till the subproblems at the lowest level are easy to solve. Similarly a large program is broken into a number of

NOTES

subprograms and in turn each subprogram is further decomposed into subprograms and so on. Suppose we want to solve a problem S, which can be decomposed into subproblems S1, S2 and S3 and so on. Let the program for S, S1, S2, S3 be denoted by P, P1, P2, P3 respectively. Further suppose that S2 is solved by decomposing it into subproblems S21 and S22 and program P21 and P22 are written for these. This operation of coding a subprogram in terms of lower level subprograms is known as the **process of stepwise refinement.** Figure 6 shows the hierarchical decomposition of P into its subprograms and sub-subprograms.

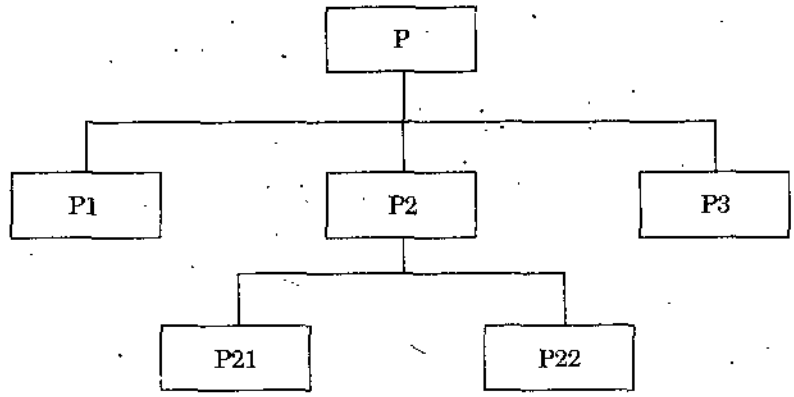


Fig. 6.

The advantages of the top-down design approach are :

1. A large problem is divided into a number of smaller problems using this approach. The decomposition is continued till the subproblems at the lowest level become easy to solve. So the overall problem solving becomes easy.
2. If we use the top-down approach for a problem then top-down programming method can be used for coding modules at various stages. So, the top level modules can be coded without coding the lower level modules earlier. This approach, is better than the bottom-up approach where programming starts first at the lowest level modules.
3. It helps in top-down testing and debugging of programs.
4. The programs become user friendly (that is easy to read and understand) and easy to maintain and modify.
5. Different programmers can write the modules for different levels.

1.7 BOTTOM-UP PROGRAMMING

The bottom-up programming approach is the reverse of the top-down programming. The process starts with identification of a set of modules which are either available or to be constructed. An attempt is made to combine the lower level modules to form modules of a high level. This process of combining modules is ontinued until the program is realised. The main drawback of the bottom-up programming approach is the assumption that the lowest level modules can be completely specified beforehand, which in reality is seldom possible. Thus, in the bottom-up approach, quite often it is found that the final program obtained by combining the predetermined lowest level modules does not meet all the requirements of the desired program.

Here no attempt is made to compare the advantages and disadvantages of the top-down and bottom-up programming. However, program development through top-down approach is widely accepted to be better than the bottom-up approach.

1.8 SUMMARY

- A program is a sequence of instructions written in a programming language.
- John vohn Neumann proposed that the programs be stored in memory. This is called the stored program concept.
- Modular programming is breaking down of a problem into smaller independent pieces (modules).
- The main objectives of structured programming are readability, clarity of programs, easy modification and reduced testing problems.
- Top-down programming is also known as the *process of stepwise refinement.*
- Bottom-up programming approach is the reverse of the top-down programming.

1.9 TEST YOURSELF

Answer the following questions :

1. What are the characteristics of a good program?
2. Discuss the techniques of problem solving.
3. Write a short note on the following :

   (*a*) Structured programming concepts

   (*b*) Modular programming
4. Explain the concept of top-down and bottom-up programming.

❑❑❑

NOTES

CHAPTER 2 CONCEPTS OF DATA TYPES AND DATA STRUCTURES

★ LEARNING OBJECTIVES ★

2.1 Introduction
2.2 Concept of Data
2.3 Concept of Data Type
2.4 Concept of a Data Structure
2.5 Concept of Primitive Data Type
2.6 Logical versus Physical Representation
2.7 Primitive and Simple Data Structures
2.8 Types of Data Structures
2.9 Operations on the Data Structures
2.10 Summary
2.11 Test Yourself

2.1 INTRODUCTION

The fundamental nature of programming and data processing requires efficient algorithms for access of the data in main memory and storage devices. The effectiveness is directly linked to the structure of the data being processed. The data structure describes the way the data is organized and stored in memory for the convenience of processing. This Chapter gives an introduction to Data Types and Data Structures.

Generally learners are asked to write programs which solve simple problems and use small amount of data. Therefore, they need not concern themselves about how the data are stored in computer's main memory and how slowly or quickly the operations of retrieval and updations are performed. However, when a complex and time-consuming problem is to be solved or when a large amount of data is to be used then it is very important that the data be organized in main memory so as to give faster access to data and the program be written accordingly. Otherwise, main memory space as well as computer time required for various operations may be wasted.

NOTES

2.2 CONCEPT OF DATA

A collection of facts, of observations, of occurences etc., is called data. Elements of this collection are called **data items.** It is something raw that is processed by the computer program to give useful information. Data can be represented numerically, alphabetically, using special symbols such as '+', '–', '%', '>', '<', etc.

2.3 CONCEPT OF DATA TYPE

Data is a collection of facts, observations etc. This collection of data items can be divided into groups such that the members of each group share a common set of properties. Such groups are called data types.

For example, integers. These can be added, subtracted, multiplied and compared.

Consider another example of the set of sets—elements of this set are sets which can undergo the operations of union, intersection and difference.

Consider a collection of 3 × 3 matrices—These can be added, multiplied and inverted (non-singular ones).

Consider a collection of strings—Strings can be compared, concatenated and broken into parts.

The first example shows that integers have a number of properties in common viz. all integers can be added, all integers can be subtracted, all integers can be compared etc. Integers are commonly written as –3, –2, –1, 0, 1, 2, 3, ... In this case, values of data type integer can be from the set of integers.

Similarly, all 3 × 3 matrices can be added, multiplied etc., and these can be commonly written in the following form :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

In this case, the data type 3 × 3 matrix can assume values from the set of all 3 × 3 matrices.

Thus, different data items belonging to different groups have different representations and have different sets of operations that can be performed on them. This gives rise to the concept of a data type. Commonly known data types are integers, real numbers and characters.

2.3.1 Definition of Data Type

A data type is a name given to the set of all data items possessing a given ensemble of properties. When we say that the number 7 is an integer, 7 shares a number of properties with other integers. In case of integers, we need not

NOTES

specify what properties are shared and how the various operations can be performed on integers because it is automatically understood. However, in the case of other data types, the properties and the operations must be specified by means of a set of axioms. Thus, we can define many data types by means of an appropriate set of axioms.

2.4 CONCEPT OF A DATA STRUCTURE

New data types can be defined in terms of previously defined one or more data types. Suppose data-types A, B and C have already been defined and a new data type D is defined in terms of A, B and C. Then values of data type D, can be decomposed into values of previously defined data types A, B and C. We call A, B and C as component data types. For example, consider an employee with attributes : Name, Age and Salary. Name is of String type, Age is of integer type and salary is of real number type. By combining three different types, we build a new data type and call it EMPLOYEE which has its component types as string, integer and real. The string data type itself is defined in terms of another data type character.

Such data types that are composed of previously defined data types are called **Data Structure.** 'EMPLOYEE' is an example of a value or instance of a data structure called record. The Name itself is an instance of data structure called string. The organization of data items in a data structure is characterized by accessing mechanisms that are used to store and retrieve individual data items.

2.5 CONCEPT OF PRIMITIVE DATA TYPE

In the above example 'EMPLOYEE' is an instance of a data structure. As in this example, it is possible that components of a structured data type may themselves be structured. Components of the components of structured data type may again be themselves structured and so on. Therefore, values of data structure may be decomposed into values, of component data types, values of component data types may further be decomposed into values of component data types and so on. Finally, we reach a stage, where we can not further decompose a value of component data types, *i.e.,* it is indivisible or atomic. The data types that are atomic are commonly called **primitive, unstructured** data types. (Truly speaking what one person may regard as indivisible, the other person may regard it as divisible. However, here we regard those data types as indivisible, whose further decomposition is not meaningful).

Examples of primitive data types are integers, real numbers, characters and boolean. On most computers, these data types are available as their built-in features. These primitive data types are discussed below :

2.5.1 Integer

Integer is the simplest data type. Mathematically, it is an element of the set of integers {....., – n, – (n – 1), ... – 2, – 1, 0, 1, 2, ...}. In case of computers, integer data type can assume values only from a subset of the set of integers which is determined by the word length of the computer. For example, if the word length is of 16 bits and two's complement method is used for storing negative integer, the data type integer can assume values in the range of –32768 to 32767.

Operations that can be performed on pairs of integers are standard arithmetic operations such as addition, subtraction, multiplication and integer division. Integer division gives the quotient after ignoring the remainder. Negation can be performed on single integers.

2.5.2 Real

Real is another simple data type which is also used very commonly. A variable of data type real can assume values from a subset of real numbers. This subset of real numbers is again determined by representation and number of bits used to store the number.

Operations that can be performed on them are addition, multiplication, division and subtraction.

The result of arithmetic operations performed on real numbers may not be accurate. Accuracy again depends on real number representation in computer memory.

2.5.3 Boolean

Third simple data type is Boolean or logical data type. A variable of this type can have only one of the two values denoted by "True" or "False".

The operations that can be performed on these are 'AND', 'OR' and 'NOT'. These operations are defined as given in Table 1 :

Table 1. Operations on Boolean Data Type

X	*Y*	*X AND Y*	*X OR Y*	*NOT X*
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

The 'True' or 'False' values may also arise as a result of comparisons such as 12 > 16 gives 'False' value and 12 < 19 gives 'True' value.

2.5.4 Character

Fourth primitive data type is character. Values that can be assumed by a variable of the type character, are from the set of characters defined for the computer system. Different computers may have different character sets.

NOTES

For example, a character set might be {0, 1, ..., 9, A, B, ..., Z, a, b, ..., z, –, +, *, /} which includes digits, upper and lower case alphabets and special characters. The character set defined by American National Standard Institute is the ASCII character set which is the most commonly available on computers. The set of characters is ordered and therefore operation of comparison of characters can be performed.

2.5.5 Data Type : Pointer

The data type pointer is an unstructured data type. The pointer type variable contains the address of the location of another variable.

Operations that can be performed on pointers are comparison of pointer variable for equality/inequality assigning value of one pointer variable to another pointer variable and assigning NULL pointer *i.e.,* no valid address. In C language, pointer variables are declared using the symbol *.

2.6 LOGICAL VERSUS PHYSICAL REPRESENTATION

The primitive data types discussed in this chapter and the structured data types mentioned earlier are at logical level. A programmer can use these data types in his/her programs and can perform operations defined on them without requiring any knowledge of how these data types are represented in memory and how the operations are implemented. For any given data structure, there may be several different physical representations in memory. Following are the examples of different physical representations of integers and characters.

2.6.1 Integers : Physical Representation

One method of storing an integer in main memory is **sign and magnitude form.** In this method, one bit is used to represent sign and rest of the bits of the computer word are used to represent magnitude of the integer.

Second method of representing an integer in memory is using one's complement method.

Third method is two's complement method. Each method has its advantages as well as disadvantages. The range of values assumed by integers and the algorithms for addition, subtraction, division and multiplication all depend on the physical representation of integers. The user need not concern himself/herself about the physical representation. His/Her concern is at logical level only.

2.6.2 Characters : Physical Representation

Characters are represented in main memory using some encoding scheme. Most commonly used encoding schemes are ASCII and EBCDIC. There are other

encoding schemes also such as BCD code. EBCDIC (Extended Binary Coded Decimal Interchange Code) was developed by IBM for IBM computers where as ASCII (American Standard Code for Information Interchange) was developed by the American National Standard Institute. Both codes have different physical representation. EBCDIC always uses 8 bits per character and ASCII uses either 7 bits or 8 bits-per character. When it uses 8 bits per character then 8th bit is either for parity check or for extending the character set, because with 7 bits only 128 character can be encoded. Order of ASCII characters is 0-9 digits followed by letters A-Z and a-z respectively whereas in EBCDIC code the order is a-z, A-Z followed by 0-9. The order of characters is called a **collating sequence.** The operation of comparison of character strings depends on the coding scheme used.

2.6.3 Conclusion

The physical representation of a data type determines :

(*i*) The values, a data type can assume and

(*ii*) algorithms for various operations that can be performed on the data type.

Therefore, while choosing the most appropriate representation for a data type one must consider the following :

(*i*) The range of values the data type must have.

(*ii*) The operations that have to be performed on the data type.

(*iii*) The word length of main memory.

(*iv*) The other relevant characteristics of computer.

Various physical representations and algorithms for operations on logical data type will be discussed in this book.

2.7 PRIMITIVE AND SIMPLE DATA STRUCTURES

The important aspect to be considered is the structuring of data at their most primitive level within a computer *i.e.,* the data structures that typically are directly operated upon by the machine level instructions. Primitive data constitute the numbers and characters, which are built into a programming language. The examples of Primitive Data Structures are ***Integer, Boolean and Characters.*** The other data structures can be constructed from one or more primitives. The simple data structures built from primitives are ***Strings, Arrays, and Records,*** supported by many programming languages.

Table 2. Types of Data Structures

Data Structures	*Types*
Primitive data structures	Integer
	Boolean
	Character
Simple data structures	String
	Array
	Record
Compound Data structures	
Linear	Stack
	Queue
	Linked List
Non-linear	
Binary	Binary Trees
	Binary Search Tree
N-ary	Graph
	General Tree
	M-way Search Tree
	B-Tree
File Organizations	Sequential
	Relative
	Indexed–Sequential

2.8 TYPES OF DATA STRUCTURES

A data structure is a logical method of representing data in memory using the simple and complex data types provided by the language.

The data structures can be classified into following two types :

2.8.1 Simple Data Structures

These data structures are generally built from fundamental data types *i.e.,* int, float, char etc. Following data structures can be termed as simple data structures :

(*i*) Array (*ii*) Structure.

2.8.2 Compound Data Structures

These data structures are formed by using simple data structures and are more complex. Its two types are :

(*i*) Linear data structures (*ii*) Non-Linear data structures.

(*i*) **Linear data structures.** These are single level data structures, having their elements in a sequence. Examples of linear data structures are :

(*a*) Stack (*b*) Queue (*c*) Linked list.

(*ii*) **Non-linear data structures.** These are multilevel data structures. Examples of non-linear data structures are :

(*a*) Tree (*b*) Graph.

Figure 1 shows all the data structures :

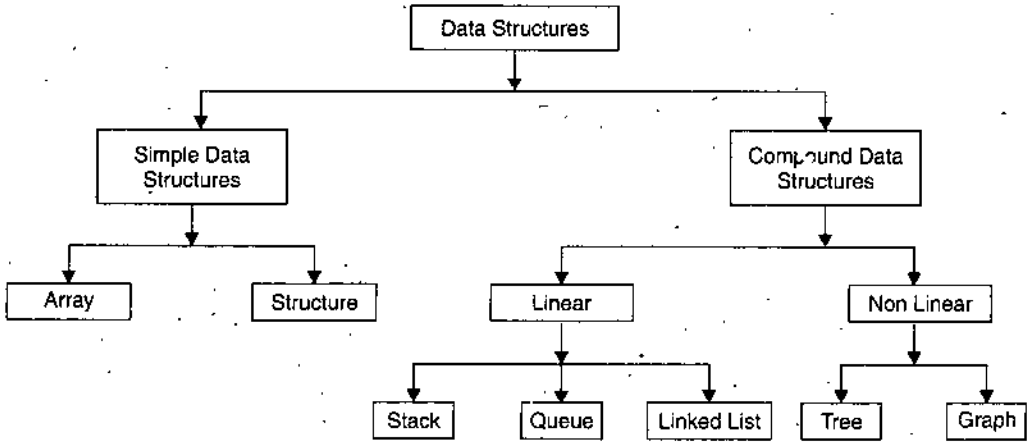


Fig. 1. *Different types of data structures*

2.8.3 Array

It is a collection of homogeneous (similar type) data elements. An array is also called linear data structure. It's elements are stored in computer memory in a linear fashion. Figure 2 shows this :

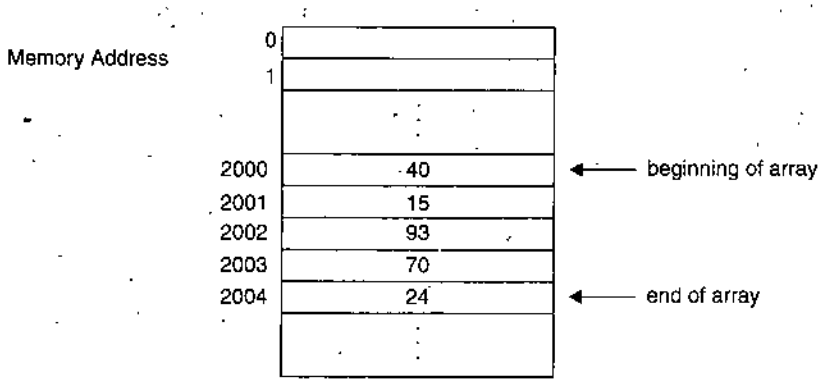


Fig. 2. *A Sequential representation of an array having 5 elements*

2.8.4 Structure

It is a collection of logically related fields in which the fields may be of same or different types. The fields that construct the structure are called members of the

structure. For example, a student record or structure may contain the following fields :

Roll Number, Student Name, Class, Address, Marks.

NCTES

2.8.5 Stack

It is defined as a list (a linear data structure) in which all the insertions and deletions are performed at one end called the TOP of stack. The insertion operation is known as **PUSH** *and the deletion operation as* **POP.** *The information is processed in LIFO (Last In First Out) way.* For example, Pile of books.

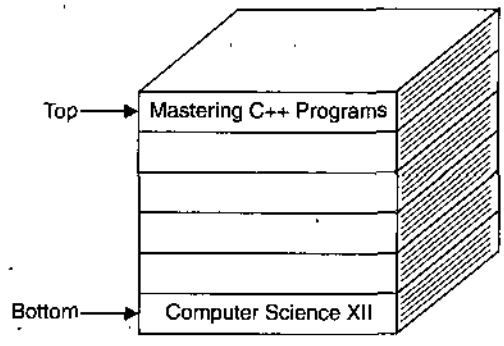


Fig. 3. *Books kept in the form of a stack*

2.8.6 Queue

It is defined as a list (a linear data structure) in which deletion and addition (insertion) operations are performed at **FRONT** *and* **REAR** *respectively. The information is processed in FIFO (First In First Out) or FCFS (First Come First Served) way.* For example, Persons entering airway reservation counter.

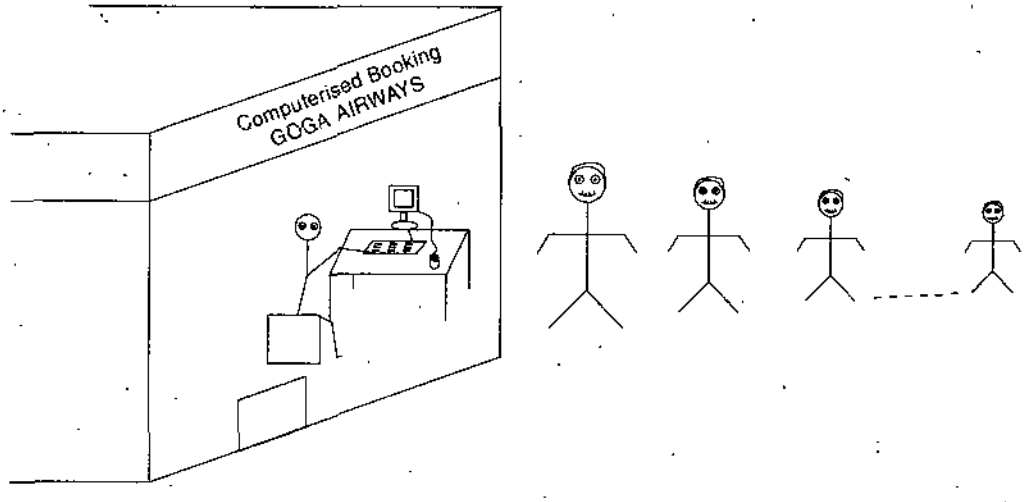


Fig. 4. *Queue of persons at a reservation counter*

2.8.7 Linked List

It is defined as a linear collection of data elements called nodes, where each node consists of two parts i.e., information and pointer to next node. The last node

contains **NULL** *pointer. A list pointer variable* **FIRST** *or* **START** *contains the address of the first node in the list.* A linked list having no node is called NULL list or empty list. Figure 5 illustrates a linked list having 4 nodes :

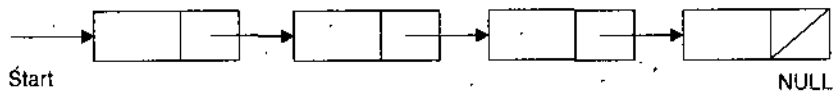


Fig. 5. *A linked list*

It is a dynamic data structure which can grow or shrink as per our requirement.

2.8.8 Tree

It is defined as a non linear collection of nodes (having no loops) having a specially designated node called the root and the remaining nodes can be partitioned into m (m ≥ 0) disjoint subsets. In computer science the conventional way of representing a tree is upside down *i.e.,* the root on the top and the remaining nodes downward.

A special class of tree in which each node except root can't have more than two nodes known as **left** and **right** subtrees of the original tree is called **binary tree.**

Figure 6 illustrates a **binary tree :**

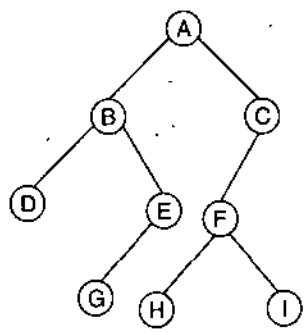


Fig. 6. *Binary Tree*

2.8.9 Graph

It is defined as a set of **nodes** (or **vertices**) and a set of **arcs** (or edges) where each arc in it is specified by a pair of nodes. Figure 7 shows a **graph :**

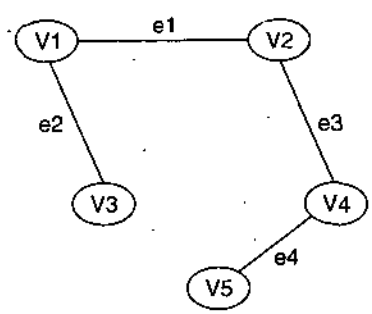


Fig. 7. *Graph representation*

NOTES

In case the arcs are ordered pairs, the graph is said to be a **directed graph** (or **diagraph**). Figure 8 illustrates a diagraph :

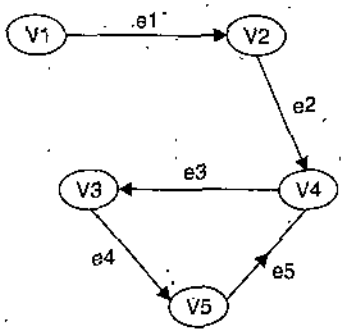


Fig. 8. *Diagraph representation*

2.9 OPERATIONS ON THE DATA STRUCTURES

The operations performed on the data structures include the following :

Traversal is a technique in which each element is processed individually and separately.

Search is an activity in which a particular record or item is to be found.

Insertion is a process in which a new element is added into the structure.

Deletion is a process in which a given item is removed from the structure.

Sorting is a process in which all elements are arranged in a specific order so that each item can be retrieved easily.

Merging is a process in which two structures are combined into a single structure.

2.10 SUMMARY

- A *data structure* is a logical method of representing data in memory using the simple and complex data types provided by the language.
- Data structures can be classified into two types—*simple data structures and compound data structures.*
- Simple data structures are generally built from fundamental data types.
- Compound data structures are formed using simple data structures and are more complex.
- The two types of compound data structure are *linear* and *non-linear.*
- *Linear data structures* are single level data structures, having their elements in a sequence. For example, *stack, queue, linked list.*

- *Non-linear data structures* are multilevel data structures. For example, *tree, graph.*
- The operations performed on the data structures are—*traversal, searching, insertion, deletion, sorting* and *merging* etc.
- A linear data structure may be implemented using either a sequential storage allocation or linked storage allocation.

NOTES

2.11 TEST YOURSELF

Answer the following questions :

1. Describe, in brief, the various data structures.
2. Define the following:
   (*a*) Data
   (*b*) Data Type
3. Discuss the concept of primitive data type.
4. Describe the various operations that, in general, can be performed on different data structures.

NOTES

SECTION B

CHAPTER 3 CONCEPTS OF POINTERS

★ LEARNING OBJECTIVES ★

3.1 Introduction
3.2 Declaring and Initializing a Pointer
3.3 Accessing a Variable Using Pointer
3.4 Static Variable
3.5 Summary
3.6 Test Yourself

3.1 INTRODUCTION

Pointers are very useful and important feature of C language. A beginner may find it a little confusing to start with. But once the concept of pointers is clear the user can write complex code with great ease, using this powerful tool, making C an excellent language.

A pointer is a variable which holds a memory address which is the location of some other variable in memory. As a pointer is a variable, its value is also stored in another memory location. Any variable declared in a C program has two components :

(*i*) Address of the variable

(*ii*) Value stored in the variable.

For example,

```
int x = 587;
```

The above declaration tells the C compiler for :

(*a*) Reservation of space in memory for storing the value.

(*b*) Associating the name **x** with this memory location.

(*c*) Storing the value 587 at this location.

It can be represented with the following figure :

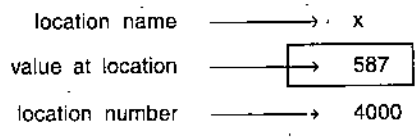


Fig. 1. *Representation of a variable.*

Here, the address 4000 is assumed one, it may be some other address also. Remember that *the address of a variable is the address of the first byte occupied by that variable in memory.* Also the values are stored in binary form inside the memory.

Let the address of **x** be assigned to a variable **ptr** having address 4036. Since the value of **ptr** is the address of the variable **x**, the value of x can be accessed using the value of **ptr** or in other words we can say that the variable **ptr** 'points to' the variable x so it is called a 'pointer'. The above concept can be represented as given below :

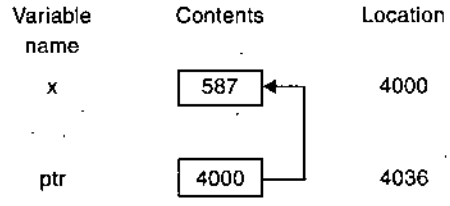


Fig. 2. *Illustration of a pointer as a variable.*

Pointers are frequently used in C language, as they offer a number of benefits to the users. They include :

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function parameters.
3. Pointers permit references to functions and thereby allowing passing of functions as parameters to other functions.
4. For saving the storage space by using the pointer arrays for character strings.
5. Pointers allow C to support dynamic memory management (*i.e.*, allocation/ deallocation of memory at run time).
6. Dynamic data structures such as *structures, linked lists, stacks, queues* and *trees* can be easily manipulated using pointers.
7. For reducing the size and complexity of programs.
8. For fast execution of programs.

NOTES

3.2 DECLARING AND INITIALIZING A POINTER

For storing the address of a variable, we must declare the appropriate pointer variable for it. The syntax for a pointer declaration is given below :

```
type *ptr_name;
```

Here, type specifies the type of the variable that is to be pointed to by the pointer **ptr_name.**

* represents the variable **ptr_name** as a pointer variable and it needs a memory location too.

For example,

```
int *ptr;    /* declaration of an integer pointer */
int x = 547;
ptr = &x;    /* ptr stores the address of x */
```

The actual address of a variable in memory is not known to us. So the **&** (address operator) is needed for returning the address of the variable following it *i.e.,* a variable name is followed after **&**. Similarly, the following statements

```
float *fptr, fvalue;
char *cptr, ch;
fvalue = 40.5;
ch = 'A';
fptr = &fvalue;
cptr = &ch;
```

show the pointer initialization, by first declaring the pointer variables and then making the pointer variables to point to their respective data type variables. *A pointer variable contains garbage until it is initialized.* We should not use a pointer before initializing it.

Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.

The data type of the pointer must be same as the data type of the variable to which it points.

In C, the assignment of an absolute address is not allowed to a pointer variable. *For example,*

```
int *iptr;
iptr = 258;  /* invalid assignment */
```

We can initialize a pointer variable while declaring it, as given below :

```
int num = 85;
int *iptr = &num;    /* initialization while declaration */
```

Note that variable **num** is first declared and then its address stored in pointer variable **iptr.**

NOTES

The following program prints the different types of variables and their addresses. As the memory addresses are unsigned integers so we can use %u or %lu format for printing the address values in integer form or %x format for printing the address values in hexadecimal form.

```
/* illustration of address of (&) operator for getting address */

#include<stdio.h>
main()
{
  char ch;
  int x;
  float y;
  x=376;
  y=12.5;
  ch='J';   /* ASCII value of 'J' gets stored in ch */
  clrscr();
  printf("The addresses are shown in decimal form\n\n");
  printf("You may get some other addresses on your system\n\n");
  printf("\nValue of ch = %c",ch);
  printf("\nAddress of ch is %u", &ch);
  printf("\n\nValue of x = %d",x);
  printf("\nAddress of x is %u",&x);
  printf("\n\nValue of y = %.2f",y);
  printf("\nAddress of y is %u",&y);
  getch(); /* freeze the monitor */
}
```

PROGRAM 1

The output of Program 1 will be :

```
The addresses are shown in decimal form
You may get some other addresses on your system
Value of ch = J
Address of ch is 65489
Value of x = 376
Address of x is 65490
Value of y = 12.50
Address of y is 65492
```

NOTES

3.3 ACCESSING A VARIABLE USING POINTER

In C, the value of a variable (once its address has been assigned to a pointer variable) can be accessed using the unary operator * (asterisk) known as the indirection operator.

The operator * is followed by an address and it can be kept in mind as 'value at address'. *For example,*

```
int value, num, *iptr;
value = 2007;
iptr = &value;
num = *iptr;
```

after the execution of the above statements num and value both have 2007.

In C, the pointers and addresses are utilized by means of symbolic names. A statement like *376 will not work at all. The following program prints the value of variables using the indirection operator ' * ' alongwith the addresses.

```
/* illustration of indirection operator (*) for printing values */

#include<stdio.h>
main()
{
  char ch, *cptr;
  int x, *iptr;
  float y, *fptr;
  x=376;
  y=12.5;
  ch='J';     /* ASCII value of 'J' gets stored in ch */
  cptr=&ch;
  iptr=&x;
  fptr=&y;
  clrscr();
  printf("The addresses are shown in Hexadecimal form\n\n");
  printf("You may get some other addresses on your system\n\n");
  printf("\nValue of ch = %c",*cptr);
  printf("\nAddress of ch is %x",cptr);
  printf("\n\nValue of x = %d",*iptr);
  printf("\nAddress of x is %x",iptr);
  printf("\n\nValue of y = %.2f",*fptr);
  printf("\nAddress of y is %x",fptr);
  getch(); /* freeze the monitor */
}
```

PROGRAM 2

The output of Program 2 will be :

```
The addresses are shown in Hexadecimal form
You may get some other addresses on your system
Value of ch = J
Address of ch is ffcb
Value of x = 376
Address of ch is ffcc
Value of y = 12.50
Address of y is ffce
```

The operation of writing the value or manipulating it by using * as a prefix with a pointer variable or pointer expression is called **dereferencing** pointers. In C, a pointer stores the address of another variable which in turn can store address of another variable and so on. Therefore we can have a pointer that stores another pointer's address. *For example,*

```
int val=336, *ptr, **ptr_to_ptr;
ptr=&val;
ptr_to_ptr=&ptr;
```

Here *ptr denotes an integer pointer

**ptr_to_ptr denotes a pointer to an integer pointer.

The following program illustrates different ways to print the value of addresses and data pointed to by simple variable, pointer and pointer to pointer :

```
/* illustrate concept of pointers */
   #include<stdio.h>
   main()
   {
     int val=336,*ptr,**ptr_to_ptr;
     ptr=&val;     /* store address of val in ptr */
     ptr_to_ptr=&ptr; /* store address of ptr in ptr_to_ptr */
     clrscr();
     printf("The addresses are shown in decimal form\n");
     printf("\nYou may get some other addresses on your system\n");
     printf("\nAddress of val is %u",&val);
     printf("\nAddress of val is %u",ptr);
     printf("\nAddress of val is %u",*ptr_to_ptr);
     printf("\nAddress of ptr is %u",&ptr);
     printf("\nAddress of ptr is %u",ptr_to_ptr);
     printf("\nAddress of ptr_to_ptr is %u", &ptr_to_ptr);
     printf("\nValue of ptr is %u",ptr);
     printf("\nValue of ptr_to_ptr is %u",ptr_to_ptr);
     printf("\nValue of val = %d",val);
```

NOTES

NOTES

```
    printf("\nValue of val = %d",*(&val));
    printf("\nValue of val = %d",*ptr);
    printf("\nValue of val = %d",**ptr_to_ptr);
    getch(); /* freeze the monitor */
}
```

PROGRAM 3

The output of Program 3 will be :

```
The addresses are shown in decimal form
You may get some other addresses on your system
Address of val is 65490
Address of val is 65490
Address of val is 65490
Address of ptr is 65492
Address of ptr is 65492
Address of ptr_to_ptr is 65494
Value of ptr is 65490
Value of ptr_to_ptr is 65492
Value of val = 336
Value of val = 336
Value of val = 336
Value of val = 336
```

Figure 3 makes the output of the above program more clear :

Variable name	Contents	Location
val	336	65490
ptr	65490	65492
ptr_to_ptr	65492	65494

Fig. 3. *Illustration of pointer to a pointer.*

3.4 STATIC VARIABLE

In *computer programming,* a **static variable** is a *variable* that has been *allocated statically*—whose lifetime extends across the entire run of the program. This is in contrast to the more ephemeral *automatic variables,* whose storage is allocated and deallocated on the *call stack*; and in contrast to object whose storage is *dynamically allocated.*

In many programming languages, such as *Pascal*, all *local variables* are *automatic* and all *global variables* are allocated statically. In these languages, the term "static variable" is generally not used since "local" and "global" suffice to cover all the possibilities.

NOTES

3.5 SUMMARY

- A *pointer* is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The vlaue of a variable cannot be assigned to a pointer variable.
- Before initialization a pointer variable contains garbage. Therefore, we must not use a pointer variable before it is assigned, the address of a variable.
- The definition of a pointer variable allocates memory only for the pointer variable, not for the variable to which it points.
- The *indirection operator* * is a unary operator as it operates only on a pointer variable.
- Pointers can be used to make a function return more than one value simultaneously.

3.6 TEST YOURSELF

Answer the following questions :

1. In what way can the assignment of an initial value be included in the declaration of a pointer variable ?
2. Differentiate between & and * operators.
3. What do you understand by a pointer to a pointer ? Can this be extended to any level ? Verify.
4. What is a static variable ?

NOTES

CHAPTER 4 ARRAYS, STACKS, QUEUES AND LINKED LIST

★ LEARNING OBJECTIVES ★

4.1 Introduction
4.2 Arrays
4.3 One-Dimensional Array
4.4 Two-Dimensional Arrays
4.5 Records
4.6 Defining a Structure
4.7 Stack
4.8 Stack as an Array (Array Implementation of Stack)
4.9 Operations on Stack
4.10 Stack as a Linked List (Linked Implementation of Stack)
4.11 Recursion
4.12 Queue
4.13 Operations on Queue
4.14 Queue as an Array
4.15 *Linked Implementation of a Queue*
4.16 Implementation of a Queue as a Circular Linked List
4.17 Dequeue (Double Ended Queue)
4.18 Priority Queue
4.19 Linked List
4.20 Advantages of Linked List Over Arrays
4.21 Types of Linked Lists
4.22 Operations on Singly Linked Lists
4.23 Circular Linked Linear List
4.24 Applications of Linear Linked Lists
4.25 Doubly Linked List or Two Chains
4.26 Operations on a Doubly Linked List
4.27 Summary
4.28 Test Yourself

NOTES

4.1 INTRODUCTION

An array is the most commonly used data structure. Almost all programming languages have arrays as their built-in data types. We may have one dimensional arrays or multi-dimensional arrays.

An *array* is a finite, ordered set of homogeneous elements. By ordered set, we mean that each element of the set has a unique position and can be accessed by referring to its position within the set. By homogeneous, we mean that all the elements of the set are of same type. All the elements are either real or integer or character or any other type.

Stack is an important subclass of list in which insertion or deletion of an element are allowed only at one end insertion and deletion operations are known as PUSH and POP respectively. A queue is a subclass of lists in which insertion and deletion take place at specific ends *i.e.,* REAR and FRONT respectively. The term 'list' means a linear collection of elements. In a linked representation of a simple list, the address of the next element must also be stored explicity with each element.

4.2 ARRAYS

An array is a collection of the homogeneous (same type) elements that are referred by a common name. It is also called a subscripted variable as the elements of an array are used by the name of an array and an index or subscript. Arrays are of two types :

(i) one-dimensional arrays (ii) multi-dimensional arrays (2 or more).

4.2.1 One-dimensional Arrays

The syntax of declaring a single-dimensional array in C is as follows :

```
type variable_name[SIZE];
```

An array must be explicitly defined so that the compiler can allocate memory for it. In the above declaration, type defines the base type of the array *i.e.* type of each element. SIZE defines the number of elements the array can store. For example,

```
int arr[10] ;
```

Here **arr** is the name of the array, **SIZE** is 10 and it is of **int** type. The array subscript always starts from zero. So, arr [4] would refer to the fifth element in the array **arr** where 4 is the array index or subscript.

NOTES

The entire array can be shown as in figure 1.

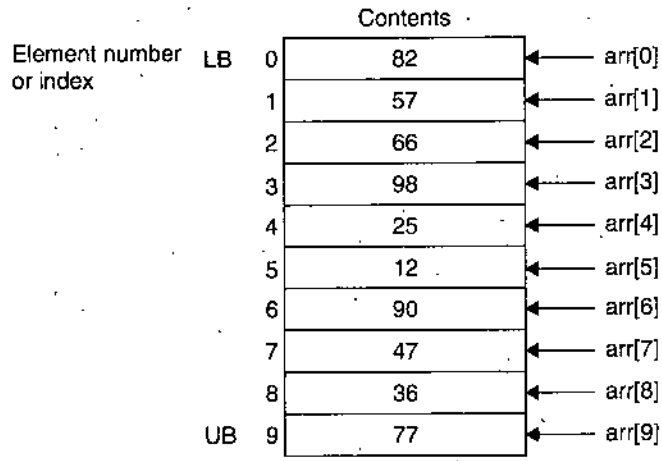


Fig. 1. *Schematic representation of an array* ***arr*** *[10]*

Here **LB** denotes lower bound of the index or subscript and **UB** the upper bound. The elements of one dimensional array are stored in the memory locations by sequential allocation technique.

NOTE

In the memory ***arr*** *actually refers to the starting position or the address of the area which gets allocated for storage of elements of the array. So,* ***arr*** *stores the address of* ***arr[0],*** *the starting element of the array.*

A vector is a mathematical term used for the collection of numbers which are analogous *i.e.* one-dimensional (linear) array. So in C, a vector can represent only integers and floating point numbers.

4.2.2 Address Calculation

The array elements are stored in contiguous memory locations by sequential allocation technique. The address of ith element of the array can be obtained if we know :

1. The starting address *i.e.* the address of the first element called Base address denoted by **B.**
2. The size of the element in the array denoted by **S.**

Consider the array **arr** in which LB<=UB, *i.e.* **arr [LB : UB]**

Where **LB** denotes the lower bound of the index and **UB** the upper bound of index. The address of ith element is given by :

$$\text{address of arr[i]} = B+(i-LB)*S$$

where LB <=i<=UB

NOTE

In C array index always start from 0 for one-dimensional arrays.

4.2.3 Two-dimensional Arrays (Multi-dimensional Array)

A two-dimensional array is a grid having rows and columns in which each element is specified by two subscripts. It is the simplest of multi-dimensional arrays. The first subscript identifier is the row number and the second subscript identifier is the column number. For example,

An array **a [m] [n]** is an **m** by **n** table having **m** rows and **n** columns containing m × n elements. The size of the array (total number of elements) is obtained by calculating m × n.

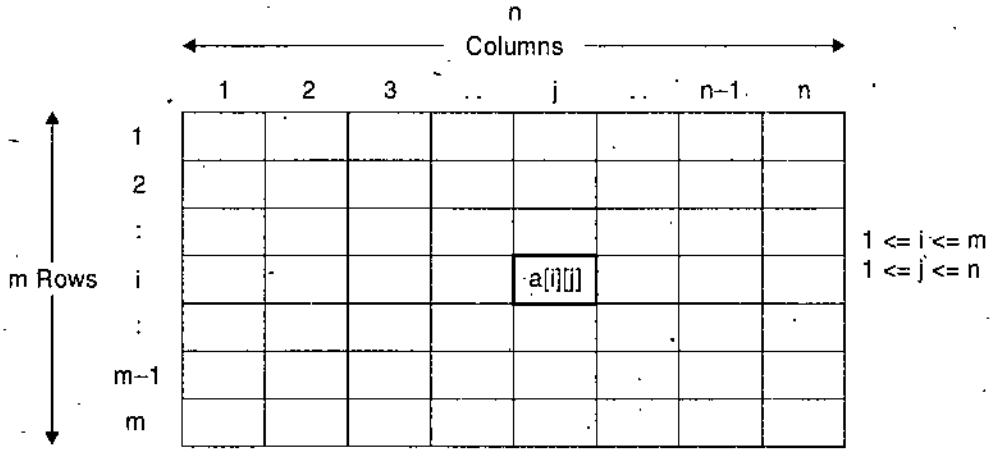


Fig. 2

Here **a[i][j]** denotes the element in the ith row and jth column. Size of the array is **m × n.**

The syntax of declaring a two-dimensional array in C is as follows :

```
type variable_name [number of rows][number of columns];
```

For example, `int a[5][5];`

Here '**a**' is the name of the array of type **int** of size 5 by 5.

The array elements are **a[0][0], a[0][1], ..., a[4][4].**

So, the two dimensional array is defined using two subscripts in the form of a matrix.

4.2.4 Sequential Allocation for Two-dimensional Array

Suppose we have a two dimensional array a[1 : 3, 1 : 4] of type **int.** An integer requires two bytes of storage.

Since the main memory of a computer is linear, two-dimensional array cannot be stored in its natural grid form. The array elements are stored linearly using one of the following methods :

(*i*) *Row Major Storage*

(*ii*) *Column Major Storage.*

NOTES

The Row Major Storage is shown in figure 3. Using this method a two dimensional array is stored with all the elements of first-row in sequence followed by the elements of second row and so on.

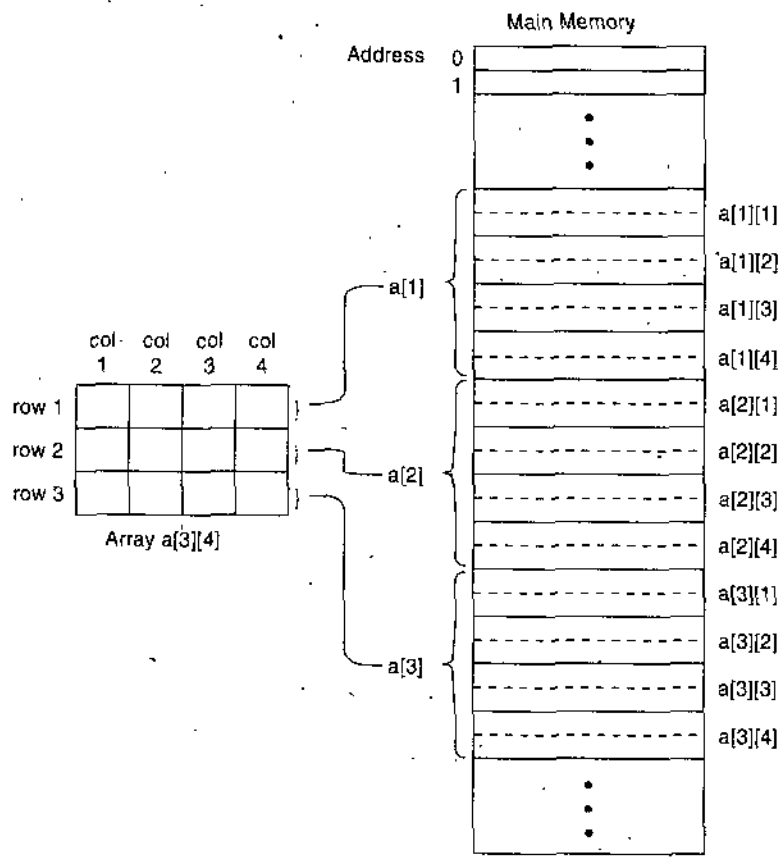


Fig. 3. *Row major storage of array a[1 : 3][1 : 4]*

4.2.5 Address Calculation of Elements of Array a[LB1 : UB1, LB2 : UB2]

Let i, j denote the row and column index where LB1<=i<=UB1 and LB2<=j<=UB2

Address of an element = B+(number of elements before it) * S

Where B is base address and S denotes size of each element.

Also the number of rows = M = (UB1 – LB1 + 1)

and the number of columns = N = (UB2 – LB2 + 1)

Using Row Major order the address of a [i] [j] is given by,

Address of a[i][j] = B + (Number of elements before a [i] [j]) * S

= B + (Number of elements in (i – LB1) rows
+ number of elements in ith row before column j) * S

= B + [(i – LB1) * N + (j – LB2)] * S

The Column Major Storage of the array a[1 : 3, 1 : 4] is shown in figure 4.

NOTES

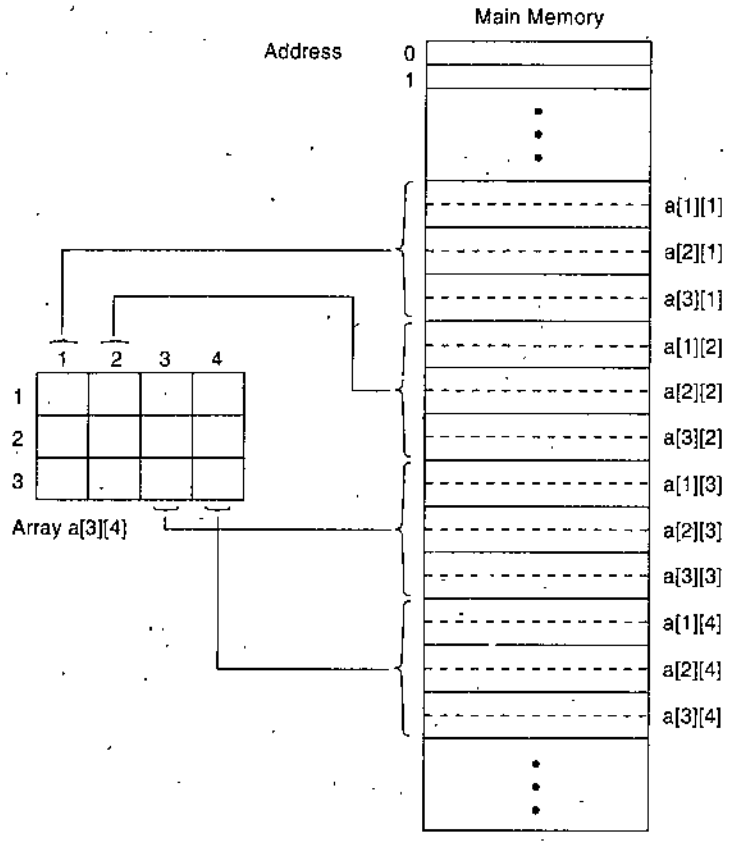


Fig. 4. *Column Major Storage of array a[1 : 3][1 : 4]*

The Column Major Storage stores all the elements of first-column in sequence followed by the elements of second column and so on.

Using Column Major order the address of a[i][j] is given by,

Address of a[i][j] = B + (Number of elements before a[i][j]) * S

= B + (Number of elements in jth column before row i
+ number of elements in (j – LB2) columns) * S

= B + [(i – LB1) + (j – LB2) * M] * S

NOTE *The array index for row and column for two dimensional arrays always start from 0 in C.*

Example 1. *An array X[7][20] is stored in the memory with each element requiring 2 bytes of storage. If the base address of array is 2000, calculate the location of X[3][5] when the array X is stored in column major order.*

NOTE *X[7][20] means valid row indices are 0 to 6 and valid column indices are 0 to 19.*

Solution. Here, Base address B = 2000

Size of an element S = 2 bytes

NOTES

Number of rows $M = 6 - 0 + 1 = 7$

$\because M = UB1 - LB1 + 1 \ (LB1 = 0)$

Number of columns $N = 19 - 0 + 1 = 20$

$\because N = UB2 - LB2 + 1 \ (LB2 = 0)$

The array is stored in column major order.

$$\text{Address of } X[i][j] = B + [(i - LB1) + (j - LB2) * M] * S$$

$$\therefore \text{Address of } X[3][5] = 2000 + [(3 - 0) + (5 - 0) * 7] * 2$$

$$= 2000 + [3 + 35] * 2 = 2000 + 76 = 2076.$$

4.3 ONE-DIMENSIONAL ARRAY

Traversal

It means visiting each element (from start to end) one after the other. For example, traversal in the array shown in figure 5 :

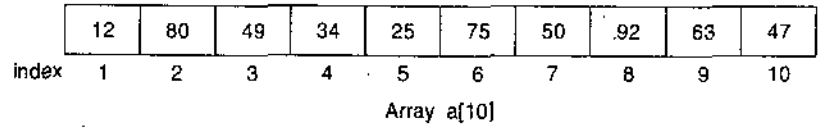


Fig. 5

would be processing of a[1], a[2], a[3], ..., a[10].

4.3.1 Algorithm for Traversal

Let A be an array of size N. We have to traverse through the array (*i.e.*, visit each element) and perform some desired operation on each element. Let the desired operation be denoted by OPERATE. I denotes the array index. Assuming lower bound starts with 1.

1. Repeat for I = 1, 2, ..., N

   OPERATE on A[I]

2. End

NOTE *In C the array index starts from 0.*

The following function in 'C' illustrates the concept of traversal in a one dimensional array :

```
/* function definition add() */

float add(float a[],int n)
```

NOTES

```
{
  int i; /* local variable */
  float total=0.0;
  for(i=0;i<n;i++)
    total += a[i];
  return(total);
}
```

4.3.2 Insertion of an Element in an Array

Let A be an array of size N, having M elements (M < N). DATA is the element to be inserted at position POS (POS ≤ M + 1). For insertion elements from positions M, M – 1, M – 2, ..., POS are shifted downward by one position and the element DATA is inserted at position POS. After insertion there are M + 1 elements in the array. Figure 6 illustrates this concept :

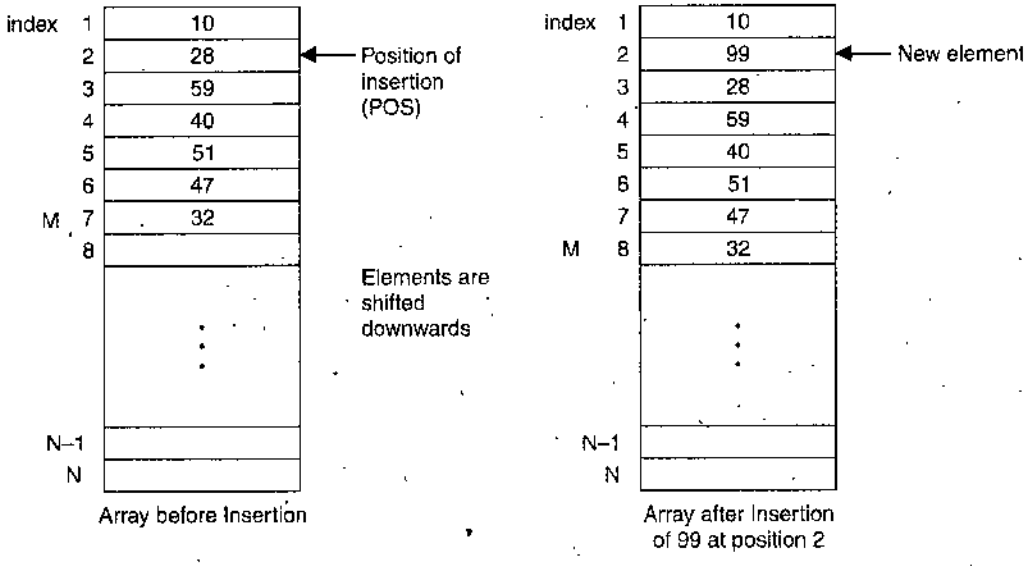


Fig. 6. *Insertion in an array at a specific position*

NOTE

Insertion is not possible if the array is already full but replacement of an existing element is possible.

4.3.3 Algorithm Insertion of an Element in an Array at a Specific Position

Let A be an array as explained above. I denotes the array index. Assuming array index starts at 1.

1. Repeat for I = M, M – 1, M – 2, ..., POS

   A[I + 1] = A[I]

2. A[POS] = DATA

3. M = M + 1

4. End

NOTES

The following function in 'C' illustrates the above concept :

```
/*  function definition insert() */

void insert(int a[], int n,int data, int position)
{
  int i; /* local variable */
  /* back shifting of elements */
  for(i=n-1;i>=position-1;i--)
    a[i+1]=a[i];
  /* insertion of element */
  a[position-1]=data;
}
```

4.3.4 Insertion of an Element in a Sorted Array given in Ascending Order

Let A be an array of size N, having M elements in ascending order (M < N). DATA is the element to be inserted. It is required that the array remains sorted after insertion. As the element is to be placed at its proper position, insert the

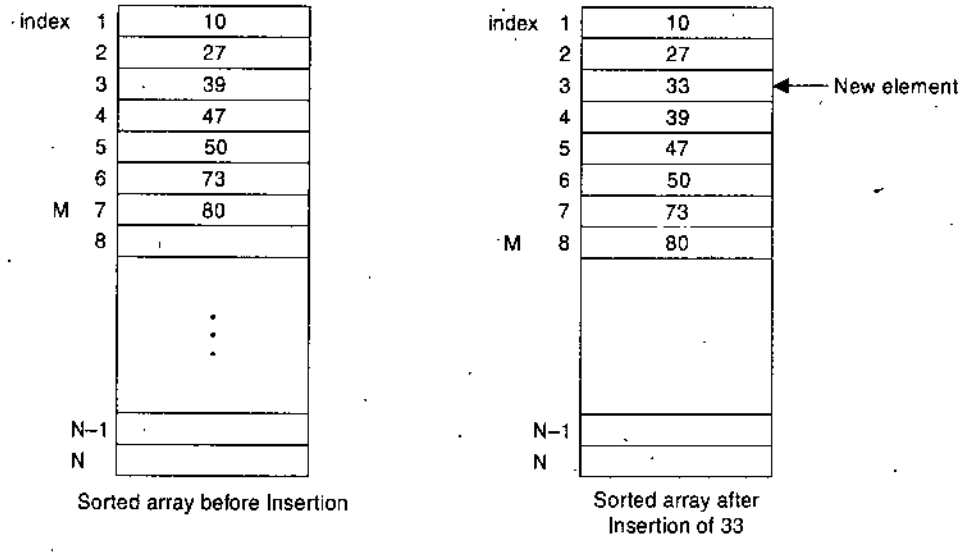


Fig. 7. *Insertion in a sorted array*

DATA in the end if it is greater than or equal to last element otherwise the position of insertion is found first by using linear search method and we stop at the location where an element greater than DATA is present. Start from beginning

and after getting the position of insertion (say POS), the elements are shifted downwards from last position (*i.e.*, M) to POS and then DATA is inserted. After insertion there are M + 1 elements in the array. Figure 7 illustrates this concept.

NOTES

4.3.5 Algorithm Insertion of an Element in a Sorted Array

Let A be an array having M elements in ascending order of size N (M < N). DATA is the element to be inserted. Array remains sorted after insertion. I and POS denote array indices. Assuming array index begins with 1.

1. If (DATA ≥ A[M]) Then
   {
   A[M+1]=DATA
   goto step 6
   }
2. POS = 1
3. Repeat while (A[POS] ≤ DATA)
   POS = POS + 1
4. Repeat for I = M, M – 1, M – 2, ..., POS
   A[I + 1] = A[I]
5. A[POS] = DATA
6. M = M + 1
7. End

The following function in 'C' illustrates the above concept with array index beginning at 0 :

```
/* function definition insert() */

void insert(int a[],int n,int data)
{
  int i,position; /* local variables */
  if(data>=a[n-1]) /* when data is >= last element */
    a[n]=data;
  else
    {
      position=0; /* initialise position */
      while(a[position] <= data)
        position++;
```

NOTES

```
        /* back shifting of elements */
        for(i=n-1;i>=position-1;i--)
          a[i+1]=a[i];
        /* insertion of element */
        a[position]=data;
      }
}
```

4.3.6 Deletion of an Element from an Array

Let A be an array having N elements. Deletion of an element means its removal from the array. Deletion may not be possible if the element does not exist. Deletion can be done in any one of following ways :

(*i*) *Deletion of an element from an array from a specific position*

(*ii*) *Deletion of an element from an unsorted array*

(*iii*) *Deletion of an element from a sorted array (say ascending order).*

(*i*) Deletion of an Element from an Array from a Specific Position

In this case the elements are shifted from the next position to the last position, one position upwards taking into consideration the position of deletion. For example, figure 8 shows the deletion of element from position 5 in array A having 8 elements.

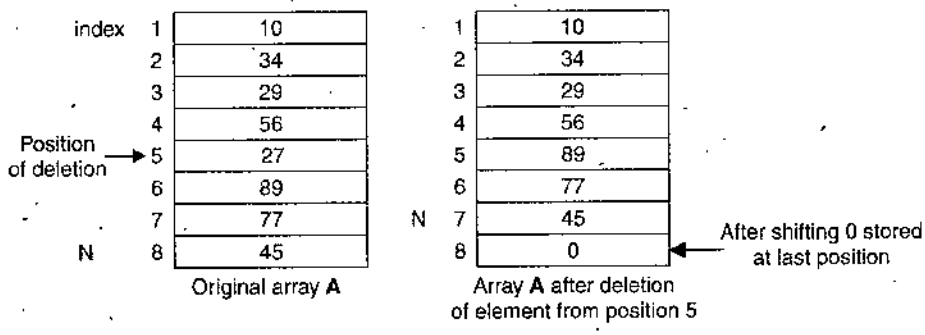


Fig. 8. *Deletion of an element from an array from a specific position*

4.3.7 Algorithm Deletion of an Element from an Array from a Specific Position

Let A be an array having N elements. POS is the position of deletion (POS ≤ N). Let I denote the array index. Assume that the array index begins at 1 and the value 0 is stored in the last position after deletion of element. N – 1 elements are left after deletion.

1. Repeat for I = POS + 1, POS + 2, ..., N

   A[I–1] = A[I]

2. A[N] = 0

3. N = N – 1

4. End.

The following function in 'C' illustrates the above concept with array index beginning at 0 :

NOTES

```
/* function definition del() */

void del(int a[],int n,int pos)
  {
    int i; /* local variable */
    /* deletion of element */
    for(i=pos;i<n;i++)
      a[i-1]=a[i];
    a[n-1]=0; /* enter 0 for last element */
  }
```

(*ii*) Deletion of an Element from an Unsorted Array

In this case the element to be deleted is searched first using **linear search** and then deleted (if present). Only first occurrence of the element will be deleted. For example, figure 9 shows the deletion of element 66 from an array A having 8 elements.

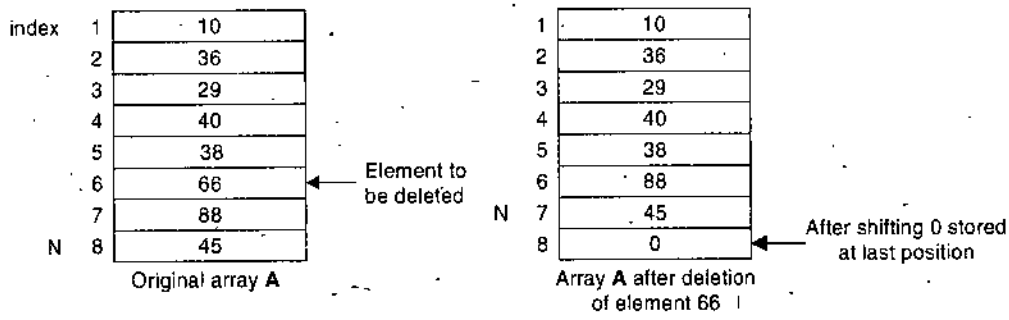


Fig. 9. *Deletion of an element from an unsorted array*

4.3.8 Algorithm Deletion of an Element from an Unsorted Array

Let A be an array having N elements. DATA is the element to be deleted. I, POS denote the array indices. If DATA is available then its first occurrence is deleted. Assuming array index begins at 1. N–1 elements are left after deletion (if possible).

1. Repeat for I = 1, 2, ..., N

```
  {
    If (A[I]=DATA) Then
    {
```

NOTES

```
        Repeat for POS=I+1,I+2,...,N
        {
            A[POS-1] = A[POS]
        }
        A[N] = 0
        N = N-1
        goto step 3
    }
}
```

2. Write (DATA, " Can't be deleted")

3. End.

The following function in 'C' illustrates the above concept with array index beginning at 0 :

```
/* function definition del() */

int del(int a[],int data,int n)
  {
    int i,pos; /* local variables */
    /* deletion of data only first occurence */
    for(i=0;i<n;i++)
      {
        if(a[i]== data)
          {
            for(pos=i+1;pos<n;pos++)
                a[pos-1]=a[pos];
            a[n-1]=0; /* enter 0 for last element */
            return(i);
          }
      }
    return(-1);
  }
```

(*iii*) Deletion of an Element from a Sorted Array (say ascending order)

First of all it is checked whether the element to be deleted is smaller than first element or larger than the last element, if so the deletion is not possible.

The element in a sorted array can be searched by any one of the two searching techniques *i.e.*, linear search or binary search. If the element is found, it is removed and the remaining elements after it are shifted upwards by one position. For example, figure 10 shows the deletion of element 50 from array A.

NOTES

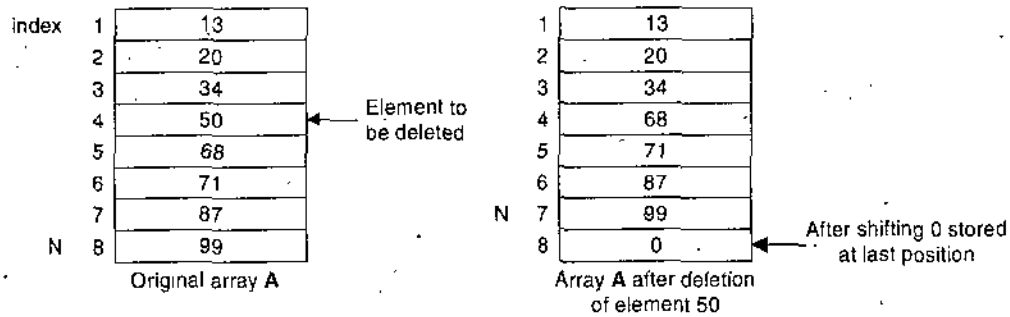


Fig. 10. *Deletion of an element from a sorted array*

4.3.9 Algorithm Deletion of an Element from a Sorted Array (given in ascending order)

Let A be an array having N elements in ascending order. DATA is the element to be deleted. I, POS denote array index. If DATA is available then its first occurrence is deleted (using linear search method). Assuming array index begins at 1. N–1 elements are left after deletion.

1. If ((DATA < A[1]) OR (DATA > A[N]) Then

```
{
   write(DATA," Can't be deleted")
   goto step 3
}
```

2. Repeat for I = 1, 2, 3, ..., N

```
{
 If (A[I] > DATA) Then
   {
    write(DATA," Can't be deleted")
    goto step 3
   }
else
   {
    If (A[I] = DATA) Then
      {
       Repeat for POS=I+1,I+2,...,N
         {
          A[POS-1]=A[POS]
         }
       A[N]=0
       N=N-1
       goto step 3
      }
   }
}
```

3. End.

NOTES

The following function in 'C' illustrates the above concept with array index beginning at 0 :

```
/*function definition del() */

int del(int a[],int n, int data)
  {
    int i, position; /* local variables */
    /* deletion of only first occurence of data (if possible) */
    if( data<a[0] || data>a[n-1]) /* when data out of range */
        return(-1);
    position=0;
    while(a[position]<data)
      position++;
    if(a[position] == data)
    {
      /* shifting of elements */
      for(i=position+1;i<n:i++)
        a[i-1]=a[i];
      a[n-1]=0; /* enter 0 for last element */
      return(position);
    }
    return(-1); /* when data not found */
}
```

4.4 TWO-DIMENSIONAL ARRAYS

Traversal

It means visiting each element one after the other.

4.4.1 Algorithm for Traversal

Let A be an array of size M × N. We have to traverse through the array and perform some desired operation on each element. Let the desired operation be denoted by OPERATE. I, J denote the array indices. Assuming lower bound start at 1 for both row and column.

1. Repeat for I = 1, 2, ..., M

```
    {
      Repeat for J=1,2,...N
       OPERATE on A[I,J]
    }
```

2. End

NOTE *In C the array indices start from 0.*

The following program illustrates the concept of traversal in a two dimensional array **a** of order 5 × 5 by displaying the elements we will enter and then the elements divisible by 10. When a two dimensional array is read or displayed, actually we are traversing through it.

NOTES

```
/* Traversal of a two dimensional array
   display elements of a two dimensional array divisible by 10 */

#include<stdio.h>

void main()
{
  void display10(int a[5][5]); /* function prototype */
  int a[5][5],i,j;
  clrscr();
  printf("Enter the array of order 5 * 5\n");
  /* row wise reading */
  for(i=0;i<5;i++)
    {
      for(j=0;j<5;j++)
        scanf("%d",&a[i][j]);
    }
   /*  echo the data */
   printf("\nGiven array is:\n\n");
  for(i=0;i<5;i++)
    {
      for(j=0;j<5;j++)
       printf("%d",a[i][j]);
      printf("\n");
    }
  display10(a); /* function call */
  getch(); /* freeze the screen until some key is pressed */
}

/* function definition display10() */

void display10(int a[5][5])
  {
     int i,j,count = 0; /* local variables */
     printf("\nElements divisible by 10 are :\n\n");
     for(i=0;i<5;i++)
       {
         for(j=0;j<5;j++)
           {
```

NOTES

```
            if( a[i][j] % 10 ==0 )
            {
              printf(" %d ",a[i][j]);
              count++;
            }
          }
        }
    if(count==0)
      printf("Not present in the array\n\n");
}
```

PROGRAM 1

The output of program 1 will be :

```
Enter the array of order 5 * 5
35  10  30  50  26
67  19  28  40  32
80  18  56  90  20
12  34  68  75  70
29  16  10  88  22
Given array is :
35    10    30    50    26
67    19    28    40    32
80    18    56    90    20
12    34    68    75    70
29    16    10    88    22
Elements divisible by 10 are :
10  30  50  40  80  90  20  70  10
Enter the array of order 5 * 5
11  76  28  49  37
93  41  32  54  39
78  13  51  98  26
16  29  45  81  61
87  84  91  47  71
Given array is :
11    76    28    49    37
93    41    32    54    39
78    13    51    98    26
16    29    45    81    61
87    84    91    47    71
Elements divisible by 10 are :
Not present in the array
```

NOTES

In the above program first of all an array a of size 5 × 5 is read and then echoed or displayed. So in writing each element of the array we are traversing through it. The function **display10()** is called, having argument–the *array*. The function displays all the elements which are divisible by 10 and keeps their **count** also, so that if no element in the array is divisible by 10 an appropriate message is displayed.

Let us consider some additional examples to perform the specified operations on the two dimensional arrays :

4.4.2 Algorithm Finding Sum of Elements on Either Diagonals of a N × N Array

Let A be an array of size N × N. I, J denote array indices for row and column respectively. SUM stores the sum of elements on both diagonals. Assuming array indices I, J begin with 1.

1. SUM = 0
2. Repeat for I = 1, 2, ..., N

```
{
 Repeat for J=1,2,...,N
  {
   If ( (I=J) OR ( (I+J) =N+1 ) ) Then
    SUM = SUM + A[I,J]
  }
}
```

3. Write SUM
4. End

The following examples show the two dimensional arrays of order 2 × 2 and 3 × 3 alongwith the sum on either diagonals :

8	9
4	7

Here sum of elements on either diagonals is 28, and the array

1	2	3
4	5	6
7	8	9

has sum of elements on either diagonals as 25 (element 5 is added only once).

The following function in 'C' implements the above concept with array indices beginning at 0 :

```
/* function definition sum_diagonal() */

int sum_diagonal(int a[SIZE][SIZE],int order)
```

NOTES

```
{
  int i,j,sum=0;
  /* sum of elements on both diagonals */
  for(i=0;i<order;i++)
    {
      for(j=0;j<order;j++)
        {
          if( (i==j) || (i+j == order-1) )
            sum+=a[i][j];
        }
    }
  return(sum);
}
```

4.4.3 Algorithm Printing of the Upper Half and Lower Half of a N × N Array

Let A be an array of size N × N. We want to print the upper half of the array. I, J denote the array indices for row and column respectively. Assuming that indices begin with 1.

1. Write ('Upper Half is')
2. Repeat for I = 1, 2, ..., N

```
{
  Repeat for J=I,I+1,....., N
  Write(A[I,J]) properly
}
```

3. Write ('Lower Half is')
4. Repeat for I = 1, 2, ..., N

```
{
  Repeat for J=1,2,....,I
  Write(A[I,J]) properly
}
```

5. End

The following function in 'C' implements this concpet with array indices beginning at 0 :

```
/* function definition print_triangles() */

void print_triangles(int a[SIZE][SIZE],int order)
  {
    int i,j,k; /* local variables */
```

NOTES

```
    /* upper triangle */
    printf("\nUpper triangle of matrix is\n");
    for(i=0;i<order;i++)
      {
        for(k=0;k<i;k++)
          printf("\t");
        for(j=i;j<order;j++)
          printf("%8d",a[i][j]);
        printf("\n");
      }
    /* lower triangle */
    printf("\nLower triangle of matrix is\n");
    for(i=0;i<order;i++)
      {
        for(j=0;j<=i;j++)
          printf("%8d",a[i][j]);
        printf("\n");
      }
}
```

4.4.4 Matrix Multiplication

Let A and B be matrices of orders m × n and q × p respectively. The **product** AB of matrices A and B is defined only when the number of columns in A equals the number of rows in B, *i.e.,* n = q. When n = q, the product is a m × p matrix C with the property

$$C(i, j) = \sum_{k=1}^{n} A(i,k) * B(k,j), \quad 1 \le i \le m, \; 1 \le j \le p$$

In the product AB, A is called the **pre-factor** of AB and B, the **post-factor** of AB.

The procedure of writing elements of AB is shown below diagrammatically :

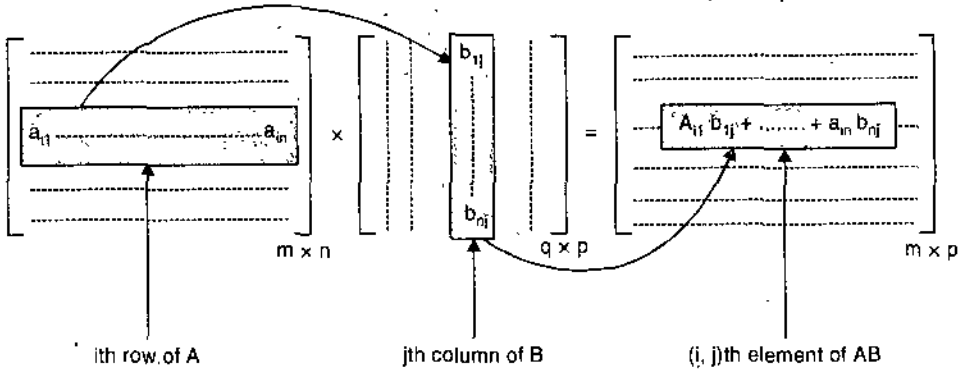


Fig. 11

NOTES

For example, let $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and $B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$

Here the product AB is defined, because the number of columns (=2) of A equals the number of rows (=2) of B. In this case,

$$AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} a_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \end{bmatrix}$$

To take another example, let $A = \begin{bmatrix} 6 & 9 \\ 2 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} 2 & 6 & 0 \\ 7 & 9 & 8 \end{bmatrix}$

The product AB is defined, because number of columns (=2) of A is equal to number of rows (= 2) of B. The order of product AB is 2×3.

The product AB is calculated by following the procedure given below :

For first row of AB

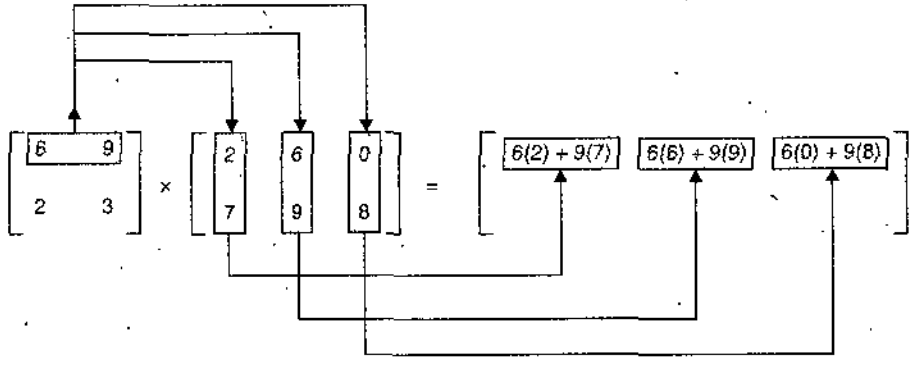


Fig. 12

For second row of AB

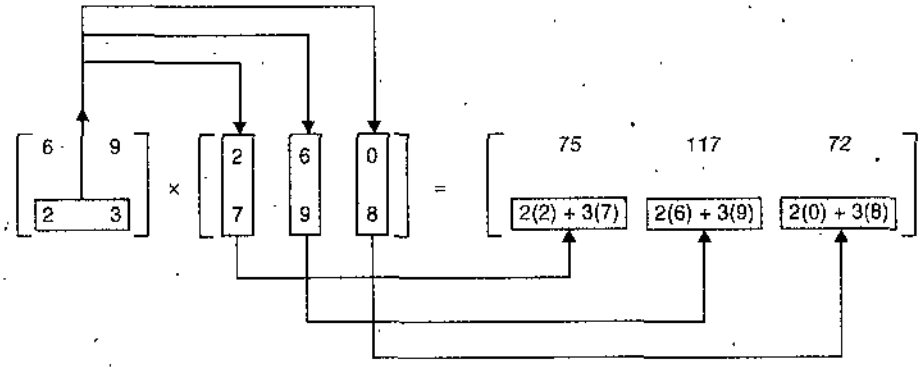


Fig. 13

In short, the product AB is written as

$$\begin{bmatrix} 6 & 9 \\ \longrightarrow & \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 2 & 6 & 0 \\ 7\downarrow & 9 & 8 \end{bmatrix} = \begin{bmatrix} 6(2)+9(7) & 6(6)+9(9) & 6(0)+9(8) \\ 2(2)+3(7) & 2(6)+3(9) & 2(0)+3(8) \end{bmatrix}$$

$$= \begin{bmatrix} 12+63 & 36+81 & 0+72 \\ 4+21 & 12+27 & 0+24 \end{bmatrix} = \begin{bmatrix} 75 & 117 & 72 \\ 25 & 39 & 24 \end{bmatrix}$$

Fr the above matrices, the product BA is not defined, because number of columns *(= 3) of B is not equal to number of rows (=2) of A.*

NOTES

4.4.5 Algorithm : Matrix Multiplication

Given two matrices A and B of orders m×n and q×p respectively. This algorithm multiplies the two matrices (if possible) and stores the result in matrix C (of order m×p). I,J and K denote array indices.

1. If n = q Then
   Begin
   Repeat for I = 1, 2, ...; m
   Begin
   Repeat for J = 1, 2, ...; p
   Begin
   C[I,J] ← 0
   Repeat for K = 1, 2, ... , n
   C[I,J] ← C[I,J] + (A[I,K])*B[K,J])
   End
   End
   End
   Else
   Write ('Matrix multiplication not possible')
2. End.

The following function in 'C' illustrates the above concept with array indices beginning at 0 :

```
/* function definition matmul() */

void matmul(int x[][S],int y[][S],int z[][S],int row1,int colm1,int
            colm2)
{
    int i,j,k; /* local variables */
    if(colm1==row2)
```

NOTES

```
    {
      for(i=0;i<row1;i++)
        {
          for(j=0;j<colm2;j++)
            {
              z[i][j]=0;
              for(k=0;k<colm1;k++)
                z[i][j]+=x[i][k]*y[k][j];
            }
        }
    }
  else
    printf("\n\nMatrix multiplication not possible\n");
}
```

4.5 RECORDS

In commercial data processing, we need to store all information about one object at one place and that it as a single unit. For example, in a payroll application, we need the following information about an employee :

Employee number, Employee name, Department, Basic Pay, Date of increment and Pay scale.

In examination processing, we need to store data about students. This data may be :

Student roll number, Student Name, Class, Section, Marks and Student Address.

In both the above cases, we have a collection of elements which are of different types but still they are to be treated as a single unit. A finite, ordered set of elements of same or different types is called a **record structure.** It is a compound structure made up from constituent type elements. Thus the complete information about an employee or a student may be called a **record.** The constituent types in these cases are of types integers, real number and string. The *string itself is a data structure made from data type character.*

In general, we have a record structure R = {R1, R2, ..., Rn), in which there are elements R1, R2, ..., Rn of different types. These R1, R2, ..., Rn are called *fields* of a record. A *field* which may further be divided into other fields is

called a *group field.* For example, 'Date' field may be divided into 'Year', 'Month', 'Day'. So 'Date' is a group field. A field which cannot be divided into subfields is called an elementary field.

In C, we can create and use the data types other than the fundamental data types. These are known as user-defined data types. Data types using the keyword **struct** are known as structures. As seen earlier arrays have similar data type elements. In C, a structure is a collection of mixed data types referenced by a single name. It is a group of related data items (structure elements) of arbitrary types.

NOTES

4.6 DEFINING A STRUCTURE

The general syntax of declaring a structure is :

```
struct <name>
{
  <type> <member1>;
  <type> <member2>;
  ....
  ....
  <type> <memberN>;
} <struct variables>; /* this semicolon is a must */
```

where <name> – is the name of the structure *i.e.,* name of the new data type. The keyword **struct** and <name> are used to declare structure variable(s).

<struct variables> – name(s) of structure variables.

The individual members of a structure can be ordinary variables, array, or other structures. The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable defined outside of the structure. A storage class however, can't be assigned to an individual member, and individual members can't be initialized within a structure type declaration.

NOTE *We can omit either the <name> or the <struct variables> but not both.*

For example,

```
struct employee
{
  int empl_no;
```

NOTES

```
    char name[30];
    char designation[20];
    char deptt[20];
} emp;
```

The structure variable can also be declared as :

```
struct employee
{
    int empl_no;
    char name[30];
    char designation[20];
    char deptt[20];
};
struct employee emp;
```

Thus, *emp* is a variable of type *employee*. In other words, *emp* is structure type variable whose composition is identified by the tag *employee*.

When we declare a structure, a data type is defined, that is no memory space is reserved.

The C compiler automatically allocates sufficient memory to store all the elements that constitute the structure, when we declare structure variable. All the members of the structure are stored in contiguous memory locations in the order of their declaration.

More than one variables can also be declared at the same time.

4.7 STACK

It is an important subclass of lists in which insertion or deletion of an element are allowed only at one end. The insertion and deletion operations are known as **PUSH** and **POP** respectively. The most accessible element denotes the **top** and least accessible element the **bottom** of the stack. It is known as a **LIFO** (**L**ast **I**n **F**irst **O**ut) list as the elements are removed in the opposite order from that in which they were added to the stack. For example, pile of trays in a cafeteria figure 14, railway shunting system for cars as shown in figure 15 :

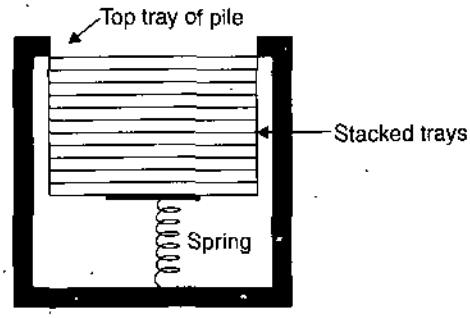


Fig. 14. *A cafeteria-tray holder.*

From a pile of trays in a cafeteria, only one tray is made available to any person from the top by the action of a spring at the tray counter. When a top tray is removed, the load on the spring becomes lighter and next available tray appears at the surface of the counter. When a tray is placed on the top of the pile the entire pile is pushed down and this tray appears above the tray counter.

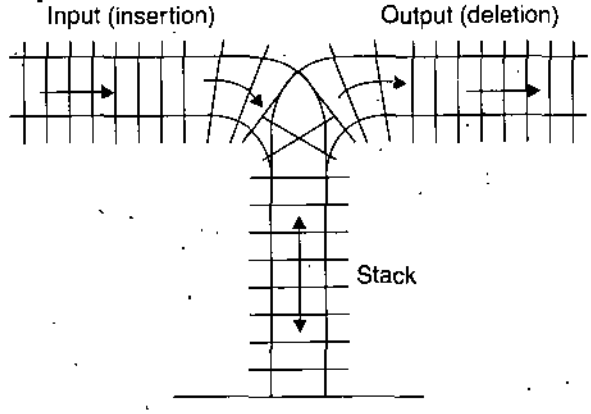


Fig. 15. *A railway shunting system shown in the form of a stack.*

In the railway shunting system, the last car to be placed on the stack will be removed as the first one.

From above discussion we conclude that a **stack** is an ordered collection of elements into which new elements may be inserted and from which existing elements may be deleted at one end, called the **top** of the stack.

4.8 STACK AS AN ARRAY (ARRAY IMPLEMENTATION OF STACK)

We know that *array is a static data structure.* So the space is allocated according to maximum number of elements present at that point of time. Therefore, *creation* of a stack as an array requires the number of elements in advance. Figure 16 shows the representation of a stack as an array :

NOTES

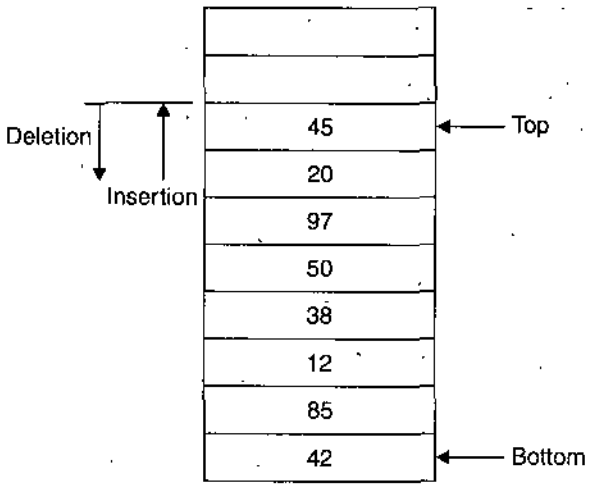


Fig. 16. *Array implementation of a stack having size 10.*

4.9 OPERATIONS ON STACK

When we add an element to a stack, we say that we *PUSH* it on the stack and if we delete an element from a stack, we say that we *POP* it from the stack. Let us see how stack of figure 16 grows or shrinks when we PUSH or POP an element.

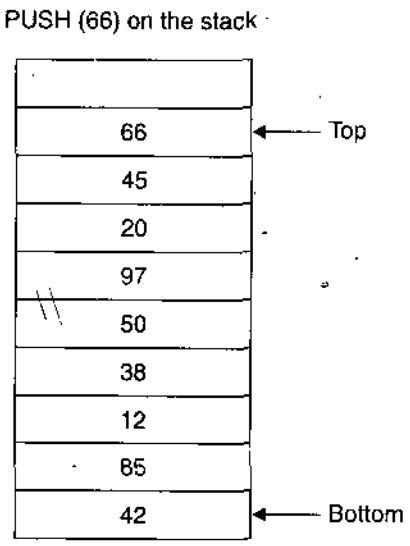


Fig. 17

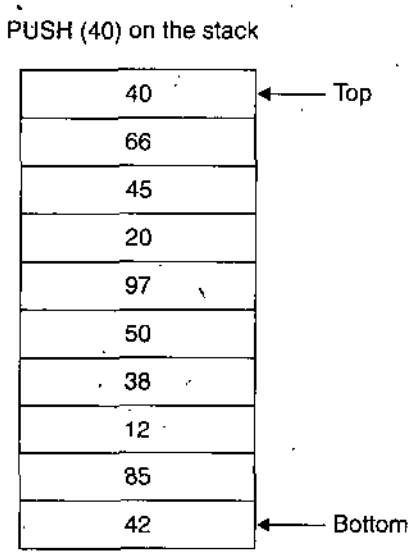


Fig. 18

Now, we cannot PUSH any other element as the stack is already full. If we do so, an **overflow** takes place.

When POP operation is performed the stack looks like,

POP an element from the stack

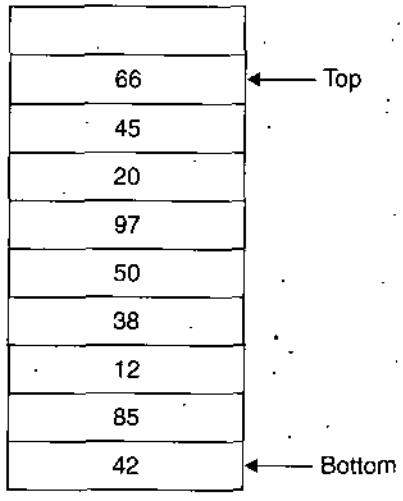


Fig. 19

When a stack is empty, it contains no element, and it is not possible to POP the stack. Therefore, before popping an element, we must check that the stack is not empty. If we do so, an **underflow** takes place.

4.9.1 Algorithm Insertion (PUSH) in a Stack as an Array

Let S be a stack having size N and DATA an element to be inserted. TOP denotes the position of the top element in the stack. Assuming the index in stack begins with 1 and go upto N.

1. If (TOP = N) Then
   ```
   {
     Write('Stack Overflow')
     goto step 4
   }
   ```
2. TOP = TOP + 1
3. S[TOP] = DATA
4. End.

In C the array index always begins with 0 and if the array size is N it varies from 0 to N–1.

4.9.2 Algorithm Deletion (POP) in a Stack as an Array

Let S be a stack having TOP as the position of the top element. DATA stores the value of the deleted element (if possible). Assuming the index in the stack begins with 1.

1. If (TOP = 0) Then
   ```
   {
     write('Stack Underflow on POP')
   ```

NOTES

```
        goto step 4
    }
```

2. DATA = S[TOP]
3. TOP = TOP – 1
4. End.

The following functions in C implements the PUSH and POP operations discussed above with array index beginning with 0 :

```
/* function definition PUSH() */

void PUSH(int S[],int data) /* function to insert element */
  {
    if(top==SIZE-1)
      {
        printf("\nStack Overflow\n");
        exit(1);
      }
    else
      {
        S[++top]=data;
        if(top==SIZE-1)
          printf("\nNo element will be inserted next time\n");
      }
  }

/* function definition POP() to delete element */

void POP(int S[])
  {
    if(top<0)
      {
        printf("\nStack Underflow\n");
        exit(1);
      }
    else
      {
        printf("\nPopped element : %d",S[top--]);
        if(top<0)
          printf("\n\nNo element left in the stack now\n");
      }
  }
```

NOTES

4.10 STACK AS A LINKED LIST (LINKED IMPLEMENTATION OF STACK)

When the number of elements are not specified in a stack, the array implementation may not be useful. As mentioned earlier a linked list is a dynamic data structure and it can store any number of elements (The limitations of the memory are always there).

Let us assume that an available area of storage (memory) for the node structure consists of available nodes as shown in figure 20, where AVAIL is a pointer variable storing the address of the top node in the stack.

Here, NEWPTR stores the address of next available node which was originally pointed to by AVAIL. LINK(AVAIL) shown in figure 20 (*i*) becomes AVAIL after taking topmost node from the availability stack as shown in figure 20 (*ii*). A node cannot be taken if AVAIL is NULL. Availability stack is also known as **free storage pool**.

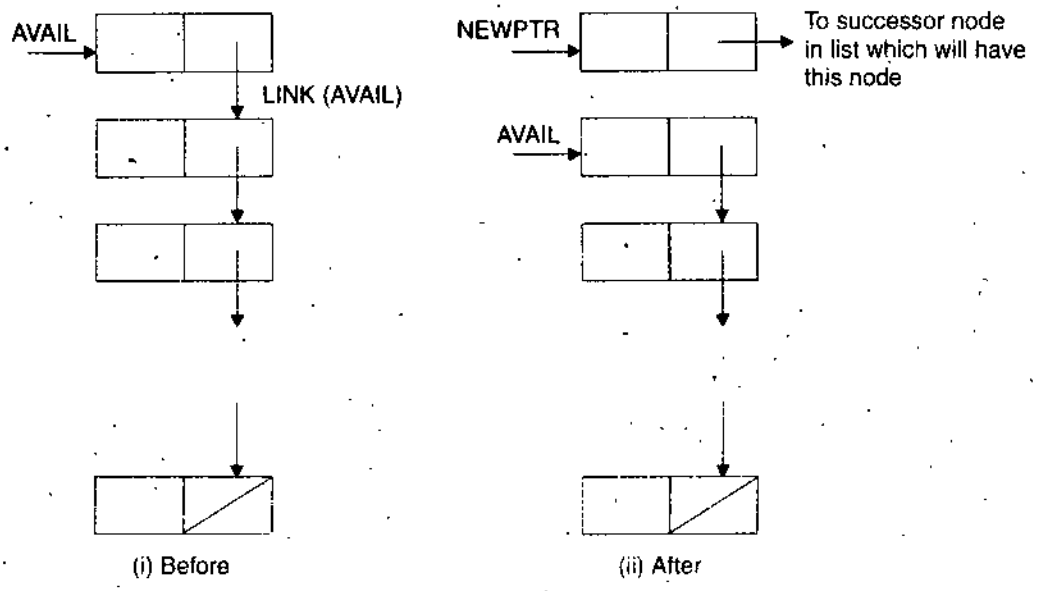


Fig. 20. *Availability stack before and after taking a node from it.*

4.10.1 Algorithm for Getting a Node from Availability Stack

1. If (AVAIL = NULL) Then
   {
   Write('Availability Stack Underflow')
   goto step 4.
   }
2. NEWPTR = AVAIL
3. AVAIL = LINK(AVAIL)
4. End.

NOTES

After getting a node it can be used as per our requirement. *The dynamic allocation of memory in C is done by using the function* ***malloc()*** *as shown :*

The node structure can be declared first as

```
typedef struct nodetype /* declare node type */
  {
    int info;
    struct nodetype *next;
  }node;
```

Now the following statement creates a node dynamically :

```
node *newptr = (node *) malloc(sizeof(node));
if(newptr==NULL)
  printf("\nAvailability Stack Underflow\n");
```

When a node is deleted (when it is no longer required) it is returned to the availability stack for further use. If the address of the deleted node is given by the variable FREEPTR, then the link field of this node is set to the current value of AVAIL and then FREEPTR becomes the new value of pointer AVAIL. Figure 21 illustrates this :

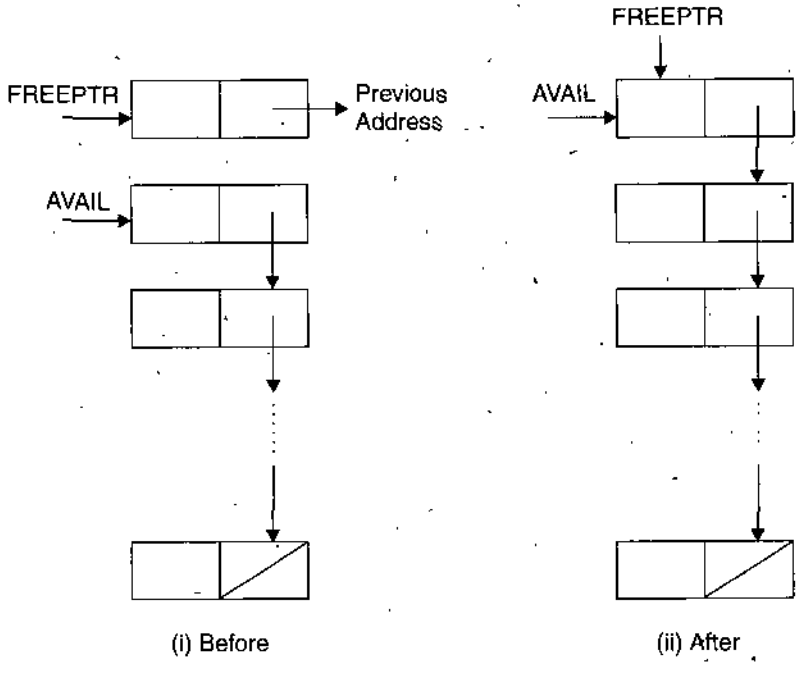


Fig. 21. *Availability stack before and after returning a node to it.*

4.10.2 Algorithm for Returning a Node to Availability Stack

1. LINK(FREEPTR) = AVAIL
2. AVAIL = FREEPTR
3. End.

The deallocation of memory (when we donot require it) in C is done by using the function **free()** *as shown :*

If **freeptr** is the address of a node which we want to delete where the appropriate changes have been performed in a linked list, the following statement deletes the node and then the memory again becomes available for use.

```
free(freeptr); /* make the memory free for use */
```

As the basic concepts of taking a node from availability stack and returning it back when it is no longer required have been explained. So, let us concentrate on actual implementation of stack as a linked list.

We know that stack is a linear data structure in which insertions and deletions are done only from one end called TOP of the stack. Linked implementation of stack is preferred over array implementation when length of the stack is unpredictable. Figure 22 shows the linked implementation of a stack having 5 elements 20, 25, 15, 17 and 42 with the element 20 at the top of stack :

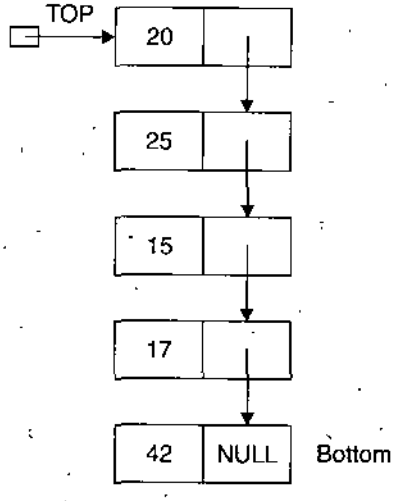


Fig. 22. *Linked representation of a stack.*

Here, TOP is a pointer variable that contains a pointer to the top node of the stack.

After PUSH operation of DATA having value 55 the linked list looks as shown in figure 23 :

NOTES

NOTES

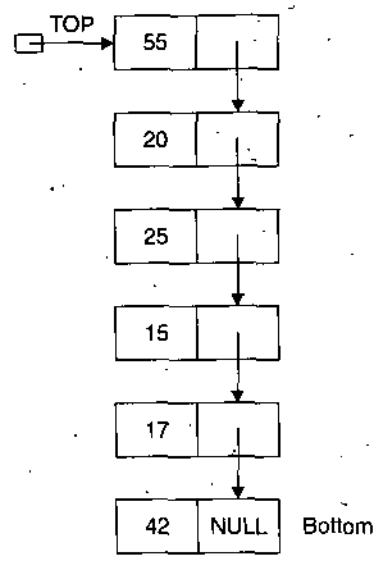


Fig. 23. *Linked representation of a stack after PUSH.*

When POP operation is performed the stack looks like,

POP an element from the stack

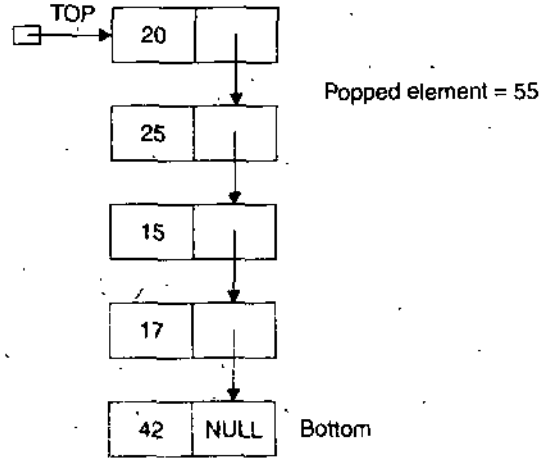


Fig. 24. *Linked representation of a stack after POP.*

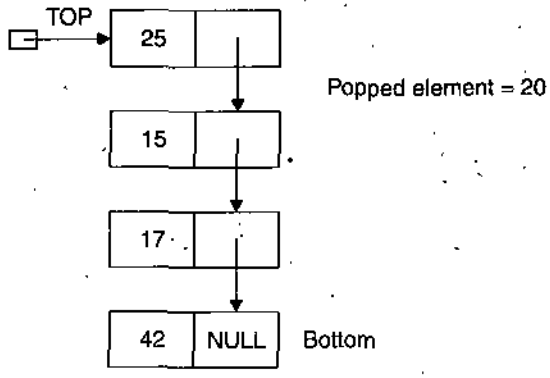


Fig. 25. *Linked representation of a stack after POP.*

When a stack is empty, TOP points to NULL and it is not possible to POP the stack. *If we try to pop an element now, an* ***underflow*** *takes place.*

4.10.3 Algorithm Insertion (PUSH) in a Stack as a Linked List

Let DATA be the element to be inserted in a stack having TOP as the pointer containing the address of the top element. AVAIL is a pointer to the top element of the availability stack. NEWPTR denotes address of new node.

1. If (AVAIL = NULL) Then

```
{
  Write('Availability Stack Underflow')
  goto step 6
}
```

2. NEWPTR = AVAIL
3. AVAIL = LINK(AVAIL)
4. INFO (NEWPTR) = DATA

   LINK (NEWPTR) = TOP
5. TOP = NEWPTR
6. End.

Using this representation we are using the pool of available nodes and we will never have to test whether a particular stack is full.

4.10.4 Algorithm Deletion (POP) in a Stack as a Linked List

Let TOP be the pointer having the address of the top element of the stack. After deletion the node is returned back to the availability stack having its top pointer as AVAIL.

NOTES

NOTES

1. If (TOP = NULL) Then
   {
   Write('Empty Stack, Underflow')
   goto step 7
   }
2. FREEPTR = TOP
3. Write('Info of deleted node is ', INFO(FREEPTR))
4. TOP = LINK(TOP)
5. LINK(FREEPTR) = AVAIL
6. AVAIL = FREEPTR
7. End.

The C implementation for getting a node when required and deleting a node have been given earlier.

Following functions in C implements the PUSH and POP operations on a stack using linked implementation :

```
/* function definition PUSH() */

void PUSH(int data) /* function to push a node */
  {
    /* get memory for a node */
    node *newptr= (node *) malloc(sizeof(node));
    if(newptr)
      {
        newptr->info= data;
        newptr->next= top;
        top=newptr;
      }
    else
      {
        printf("\nCannot create new node\n\n");
        getch();
        exit(1);
      }
  }

/* function definition POP() */

void POP() /* function to pop a node from a linked stack */
  {
    node *freeptr=NULL;
```

```
if(!top)
  {
    printf("\nEmpty Stack, Underflow\n\n");
    exit(1);
  }
else
  {
    freeptr=top;
    printf("\nPopped element : %d\n",freeptr->info);
    top=top->next;
    free(freeptr); /* make the memory free for use */
    if(!top)
      printf("\nNo element left in the stack now\n\n");
  }
}
```

4.11 RECURSION

One of the important applications of stacks is recursion. It is an important facility in many programming languages such as PASCAL, C and C++. Many problems can be described in recursive manner. *Recursion is the name given to the technique of defining a process in terms of itself.* For example,

The factorial function can be recursively defined as

$$\text{FACTORIAL (N)} = \begin{cases} 1, \text{ if } N = 0 \\ N * \text{FACTORIAL (N} - 1), \text{otherwise} \end{cases}$$

Here FACTORIAL (N) is defined in terms of FACTORIAL (N – 1), which in turn is defined in terms of FACTORIAL (N – 2), etc., until finally FACTORIAL (0) is reached, having value as 1.

A stack is used for calculating the factorial of a number.

One more example of recursion is the algorithm for finding the greatest common divisor of two integers, *i.e.,* Euclid's algorithm defined as :

$$\text{GCD(m, n)} = \begin{cases} \text{GCD (n, m), if } n > m \\ m, \text{ if } n = 0 \\ \text{GCD (n, MOD (m, n)), otherwise} \end{cases}$$

Here MOD (m, n) is m modulo n—the remainder on dividing m by n. A stack is used for finding the GCD or HCF (highest common factor) of two integers.

NOTES

4.12 QUEUE

A queue is a subclass of lists in which insertion and deletion take place at specific ends i.e., Rear and Front respectively. It is a FIFO (**First In First Out**) or FCFS (**First Come First Served**) data structure which is often used to **simulate** real world situations. Figure 26 represents a queue and illustrates how an insertion is made to the right of the right most element in the queue *i.e.*, Rear, and how a deletion is made by deleting the leftmost element of the queue *i.e.*, Front.

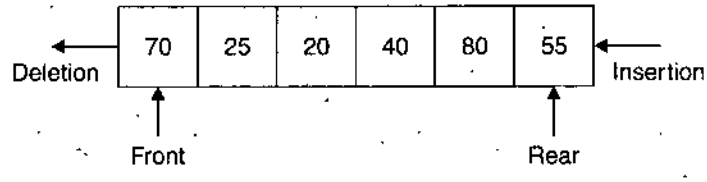


Fig. 26. *Representation of a queue.*

Some examples of a queue are

(*i*) Persons entering a cinema hall.

(*ii*) A *time sharing* computer system where many users share the system simultaneously. Here the user programs that are waiting to be processed form a waiting queue. The queue may not be operated in FIFO basis, but on some complex priority scheme known as a *priority queue.*

(*iii*) The line of vehicles waiting to proceed in some fixed direction at an intersection of roads.

(*iv*) Selection of next file to be printed from a list of files on a printer.

4.13 OPERATIONS ON QUEUE

The basic operations that can be performed on a queue are :

(*i*) Creation of queue
(*ii*) Check for empty queue
(*iii*) Check for full queue
(*iv*) Insert an element in queue
(*v*) Delete an element from queue
(*vi*) Display queue.

A queue can be either implement as an array or a linked list depending on the requirement.

NOTES

4.14 QUEUE AS AN ARRAY

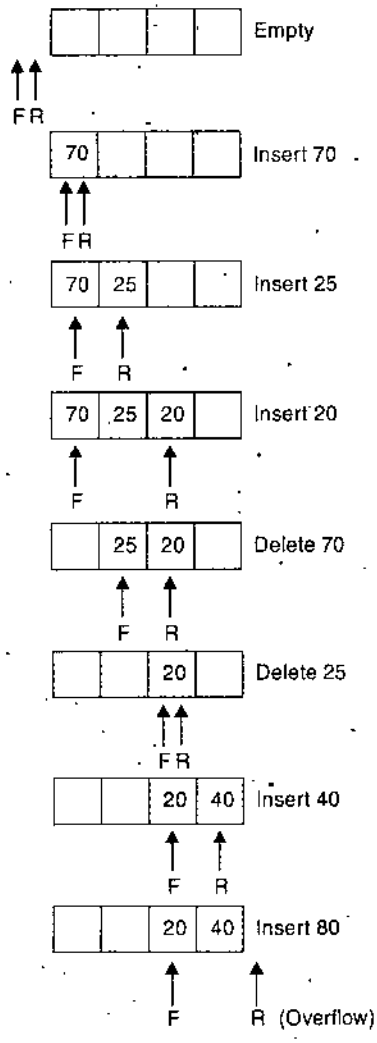


Fig. 27. *Insert and delete operations on a simple queue.*

Consider an example where the size of the queue is four elements. Initially, the queue is empty. It is required to insert elements 70, 25 and 20, delete 70 and 25 and insert 40 and 80. The queue status is shown in figure 27. Note that an overflow occurs on trying to insert element 80, even though the first two locations have no elements. Here **F** and **R** denote Front and Rear position respectively.

4.14.1 Algorithm Insertion in a Queue as an Array

Let Q be queue having size N. DATA is the element to be inserted. F and R denote the front and rear positions in the queue. Assuming the index begins at 1. Initially F and R are 0.

```
1. If(R=N) Then
   {
    Write('Insertion not possible')
    goto step 5
   }
```

NOTES

```
2. R=R+1
3. Q[R]=DATA
4. If(F=0) Then
      F=1
5. End.
```

4.14.2 Algorithm Deletion from a Queue as an Array

Let Q be a queue. F and R denote the front and rear positions in the queue. DATA is a temporary variable which stores the deleted element (if possible). Assuming the index begins at 1.

```
1. If(F=0) Then
   {
     Write('Deletion not possible')
     goto step 4
   }
2. DATA=Q[F]
3. If(F=R) Then
     {
       F=0
       R=0
     }
   Else
       F=F+1
4. End.
```

The above algorithms can waste a lot of memory if the front index F never reaches upto rear index R. Actually, an arbitrary large amount of memory would be needed to store the elements. This method should be used only when the queue is emptied at certain intervals.

NOTE *In C the array index always begins with 0.*

Following functions in C implement the queue as an array with array index beginning at 0 and performs the insert and delete operations :

```
/* function definition insert () */

void insert(int Q[],int data)
  {
    if(rear == SIZE-1)
      {
       printf("\nInsertion not possible\n");
          exit(1);
      }
    else
```

```
        Q[++rear]=data;

    if(rear == 0)   /* if first element was inserted */
        front=rear;
    if(rear == SIZE-1)
        printf("\nNo element will be inserted next time\n\n");
  }

/* function definition del() */

void del(int Q[])
  {
    if(front<0)
      {
        printf("\nDeletion not possible\n");
        exit(1);
      }
    else
      {
        printf("\nDeleted element : %d\n",Q[front]);
        if(front==rear)
          front=rear=-1;
        else
          front++;
      }
  }
```

Using the above method, it is possible to come across a situation when the queue is empty but it is not possible to insert any new element in the queue. So this implementation is not acceptable.

To overcome this drawback, we can implement the queue like a stack where one end of the stack is fixed *i.e.*, in the queue also, we fix the front of the queue so that it always represents the first element of the array. On deletion of an element (if possible) the entire queue is shifted towards the beginning of the array. In this case only the rear index is required, since the front element of the queue is always at the starting position of the array.

This technique can be easily implemented but it is too inefficient as each deletion requires shifting of elements and the logic is not correct. If a large number of elements are present in the queue, the shifting takes a lot of time and in turn makes it a costly affair.

A better solution to this problem is to use the array holding the queue as a **circular array.**

NOTES

4.15 LINKED IMPLEMENTATION OF A QUEUE

Another way to implement queues is as linked lists. It overcomes the drawback of a queue used as an array because many times the locations in an array remain unused or the array size may not be enough to store the desired number of elements if we wish more than the array size at run time. In linked implementation two pointers, **front** and **rear** point to the first and last element of the list as shown in figure 28 :

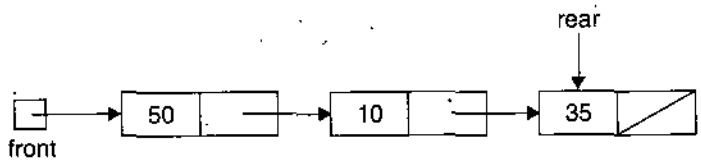


Fig. 28. *Linked representation of a queue having three nodes.*

We can check whether a queue is empty or not by checking its **front** pointer, whether it is NULL or not.

A new element can be inserted at the end of the list after the last node which is pointed to by the pointer **rear.** A care must be taken when an element is inserted into an empty queue as in this case we need to adjust the **front** pointer to the new node.

When an element is to be deleted from a queue, we must check that the queue is not empty. If only one node is present in the queue and we are deleting it, we must set **rear** to NULL, indicating that the queue is now empty.

4.15.1 Algorithm Insertion in a Linked Queue

Let DATA be the element to be inserted in a queue having **FRONT** and **REAR** as the pointers containing the addresses of the front and rear elements. The new element is always inserted only at the rear end, *i.e.*, REAR gets modified when insertion takes place. AVAIL is a pointer to the top element of the availability stack. NEWPTR denotes address of new node. Initially FRONT and REAR are NULL.

```
1. If(AVAIL=NULL) Then
   {
    Write ('Availability stack underflow')
    goto step 6
   }
2. NEWPTR=AVAIL
3. AVAIL=LINK(AVAIL)
4. INFO(NEWPTR)=DATA
   LINK(NEWPTR)=NULL
5. If(REAR=NULL) Then
   {
    FRONT=NEWPTR
```

```
    REAR=NEWPTR
  }
Else
  {
    LINK(REAR)=NEWPTR
    REAR=NEWPTR
  }
6. End.
```

Using this representation we are using the pool of available nodes and we will never have to test whether a particular queue is full.

The insertion in a linked queue is shown in figure 29 :

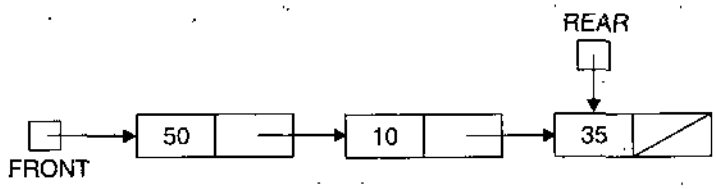


Fig. 29 (*a*) *Linked queue having three nodes.*

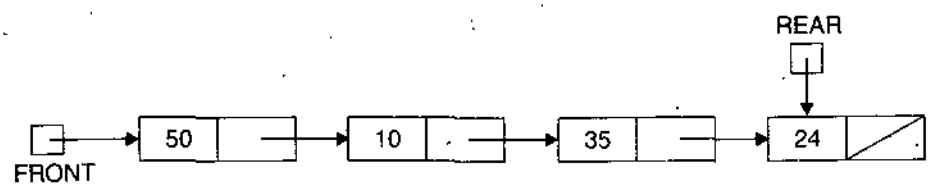


Fig. 29 (*b*) *Linked queue after insertion of a node having data 24.*

4.15.2 Algorithm Deletion from a Linked Queue

Let FRONT be the pointer having the address of the first element of the queue. As the deletion always takes place from the front of the queue, so FRONT gets modified when deletion takes place. The deleted node is returned back to availability stack having its top pointer as AVAIL. TEMP is a temporary pointer.

```
1. If(FRONT=NULL) Then
   {
     Write('Empty Queue')
     goto step 8
   }
2. TEMP=FRONT
3. Write('Deleted element is ', INFO(TEMP))
4. FRONT=LINK(FRONT)
5. If(FRONT=NULL) Then
     REAR=NULL
6. LINK(TEMP)=AVAIL
```

NOTES

```
7. AVAIL=TEMP
8. End.
```

Here steps 6 and 7 are used for making the memory free for further use *i.e.*, the node is returned to availability stack after deletion.

The deletion in a linked queue is shown in figure 30 :

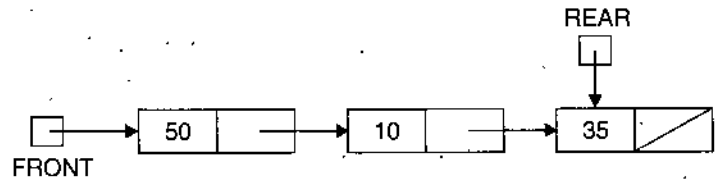


Fig. 30 (*a*) *Linked queue having three nodes.*

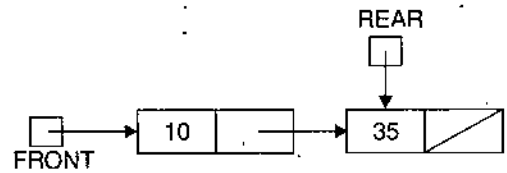


Fig. 30 (*b*) *Linked queue after deletion of a node*

Following functions in C implements the insert and delete operations on a queue using linked implementation :

```
/* function definition insert() */

void insert(int data)
  {
    /* allocate memory for a node */
    node *newptr= (node *) malloc(sizeof(node));
    newptr->info=data;
    newptr->link=NULL;
    if( front == NULL && rear == NULL ) /* function call */
      front=rear=newptr;
    else
    {
      rear->link=newptr;
      rear=newptr;
    }
  }

/* function definition del() */

int del()
  {
    node *temp; /* local variable declared */
    if(!front)
```

```
    {
      printf("\nQueue Underflow\n");
      return(1); /* return error signal */
    }
  else
    {
      temp=front;
      front=front->link;
      if(!front) /* if queue becomes empty on deletion */
        rear=NULL;
      printf("\nDeleted element : %d\n",temp->info);
      return(0);
    }
}
```

4.16 IMPLEMENTATION OF A QUEUE AS A CIRCULAR LINKED LIST

A queue can be implemented as a circular linked list also. In a circular linked list we need only a REAR pointer and the following node is its front. For inserting an element into the rear of a circular queue, the element is inserted into the front of the queue and the circular list pointer is then advanced one element, so that the new element becomes the rear. Figure 31 illustrates a circular list :

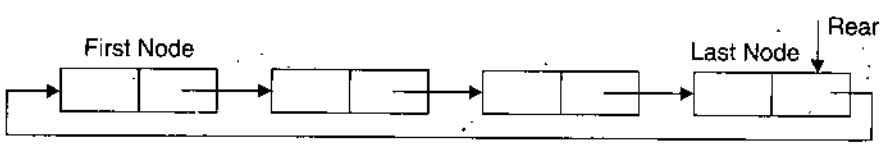


Fig. 31. *First and last nodes of a circular list.*

4.17 DEQUEUE (DOUBLE ENDED QUEUE)

It is a linear list in which insertions and deletions are made to or from either end of the structure. Figure 32 illustrates this :

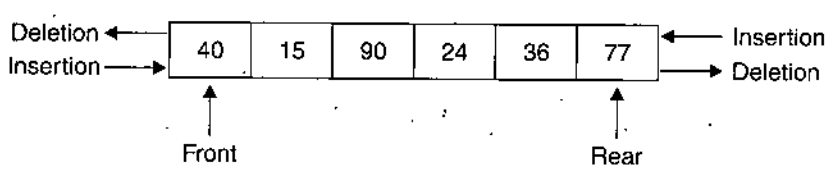


Fig. 32. *A deque.*

NOTES

It is more general than a stack or a queue. It is of two types :

(*i*) **Input-restricted deque** : Insertion allowed at only one end but deletions, allowed at both ends.

Figure 33 illustrates this :

Fig. 33. *Input restricted deque (insertion allowed at only one end).*

(*ii*) **Output-restricted deque :** Deletion allowed at only one end but insertions allowed at both ends. Figure 34 illustrates this :

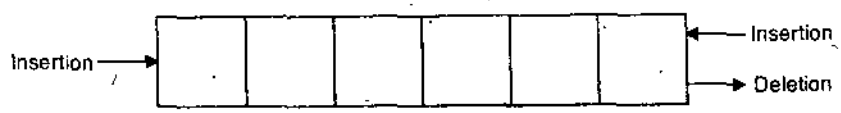


Fig. 34. *Output restricted deque (deletion allowed at only one end).*

4.18 PRIORITY QUEUE

A queue in which we can insert or delete (remove) items from any position depending on some priority is known as a **priority queue.** For example, in a multiuser system, the CPU is needed by many programs and it is utilised by the programs one at a time depending on some priority. The CPU is first used by the program having the highest priority. A priority queue can be splitted into several queues if needed. Figures 35 and 36 illustrate a priority queue :

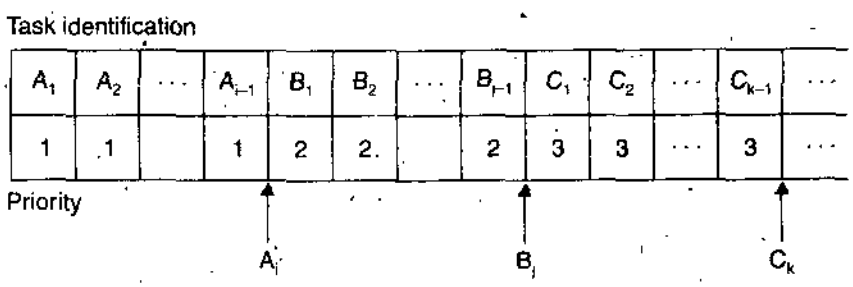


Fig. 35. *A priority queue as a single queue with insertions allowed at any position.*

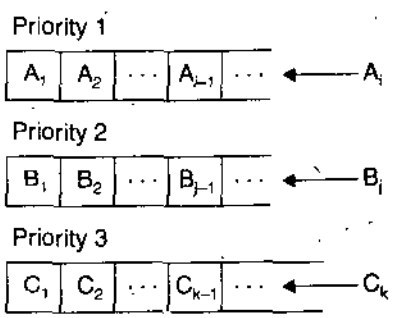


Fig. 36. *A priority queue viewed as a set of queues.*

4.18.1 Priority Queue Using Array

To maintain a priority queue in memory, we use multidimensional array, *i.e.,* use a separate queue for each level of priority (priority number). Each queue will appear as a separate circular array and have its own set of pair of pointers called *Front* and *Rear*. Assume that each queue is of same size. So, we need only a two-dimensional array in which number of rows is equal to number of priorities and the elements are added to the respective queue depending upon its priority number. Figure 37 illustrates a priority queue of size 5.

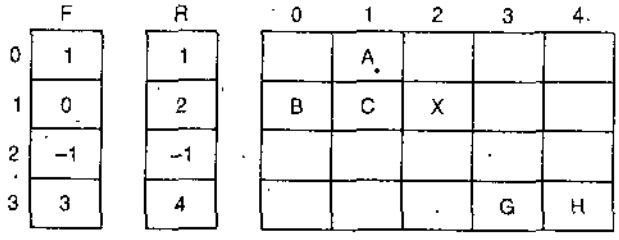


Fig. 37. *Priority queue using arrays.*

In the above priority queue, F[*i*] and R[*i*] contain the front and rear elements of row *i* of queue, *i.e.,* the row maintains the queue of elements with a priority number *i*. Whenever we are performing the insertion operation, we have to read the data item along with priority number. Then it will search for that row for empty cell, if it is found then place the item in that cell and rear and front pointers are modified as in a circular queue. If it is full, then it indicates overflow condition for that row or priority number. To delete an item from the priority queue, first it will check for the element in first row, if it is not empty, simply it deletes the element like normal queue. If it is empty, then it check for next row, if it not empty, delete the element from that row otherwise it checks for next row.If all front and rear items are –1, then it indicates underflow condition if we try to delete the element. So, the deletions are taken first from first row to last row.

4.18.2 Priority Queue Using Singly Linked List

In a singly linked list, the data items are maintained according to the priority. Each node in this queue contains three fields. One is actual data item, the second field is the priority number of the data item and the third field is link to next item in the list. At any point of time the highest priority element is in the front of the list, so that directly we can delete the item from the front of the list. If the two data items have the same priority, then we can consider their entry sequence in FIFO order. Figure 38 illustrates the implementation of priority queue using linked list.

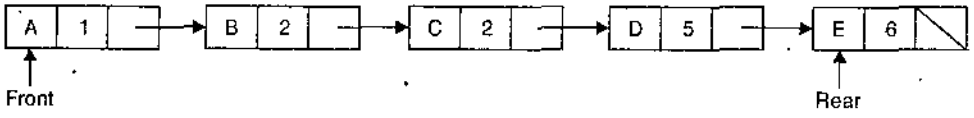


Fig. 38. *Priority queue.*

NOTES

4.19 LINKED LIST

The term 'list' means a linear collection of elements. Array is an example of linear lists which we studied earlier. However, the problems of sequential representation of lists (*e.g.*, of arrays) are fixed size, wastage of time in shifting of elements for insertions and deletions and requirement of homogeneity of elements. Thus, if size of memory required is not known in advance, or if many insertions and deletions are expected or the elements are of different types, linked representation can be used. Here, we will study linked representation of only simple lists in which all the elements are of same type.

In a linked representation of a simple list, the address of the next element must also be stored explicitly with each element. For example, if we have list of names say (Apoorva, Aanchal, Aman, Ankit, Anjali) then assuming that one name is stored in one memory location, figure 39 shows the linked representation :

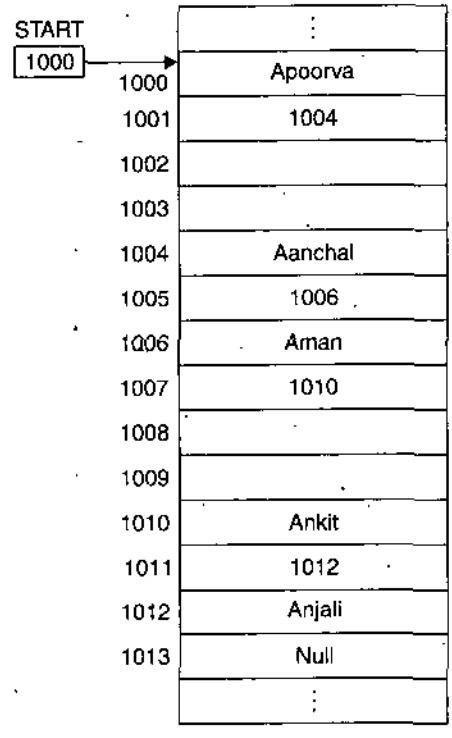


Fig. 39. *Linked implementation of a list of names*

Here, **Null** means that no next element exists in the list. This linked list can be shown as follows :

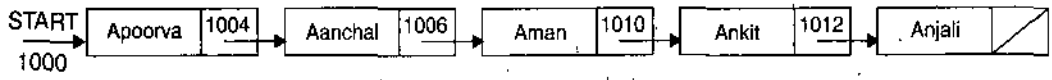


Fig. 40. *Linked list of 5 nodes*

Here, START contains the address of first element. The elements of linked list are called nodes. In general, a node must have some **information** and a **link** or **pointer** for storing the address of next node (if any) as shown in figure 41.

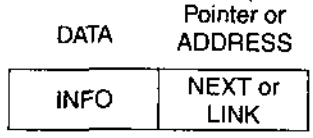


Fig. 41. *A node in a linked list*

We follow the addresses to access the elements in the logical order. It must be noted that the list in memory may not be physically contigeous or sequential. Also the **binary search** method **cannot** be applied on linked lists due to the fact that the location numbers of the elements may not be continuous.

The memory allocation for arrays is always **static**, as the number of elements is generally known in advance.

The memory allocation during program run time is known as **dynamic memory allocation**. Memory can be allocated or used (when required) and released or deallocated (when not required any more) using this technique. Data structures like **linked lists**, **trees** and **graphs** use this technique for their memory allocation.

4.20 ADVANTAGES OF LINKED LIST OVER ARRAYS

The main advantages of linked lists over arrays are :

1. It is not necessary to know the number of elements and allocate memory for them beforehand. Memory can be allocated as and when necessary.
2. Insertions and deletions can be handled efficiently without having to restructure the list.
3. The individual elements *i.e.*, nodes can be scattered anywhere in memory and no contiguous space is required like array elements.

4.21 TYPES OF LINKED LISTS

There are different types of linked lists. These are given below :

(*i*) **Singly linked lists**

(*ii*) **Circular linked lists**

NOTES

NOTES

(*iii*) **Doubly or two-way linked lists**

(*iv*) **Circular doubly linked lists.**

(*i*) **Singly linked lists.** In a singly linked list each node contains data or info and a single link which attaches it to the next node in the list. It is shown in figure 40.

(*ii*) **Circular linked lists.** A circular linked list contains a pointer from the last node to the first node as shown in figure 42. In fact, there is no first or last node, as all the nodes are linked in a circular way. One can always start traversing the list from any node and visit all the nodes, provided, pointer to any node in the list is known.

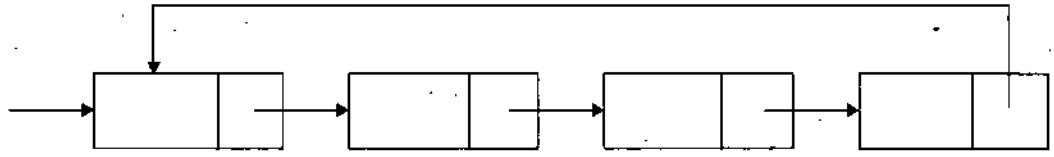


Fig. 42. *Illustration of a circular linked list*

(*iii*) **Doubly linked lists.** In a doubly linked list each node contains data and two links, one to the previous node and one to the next node. Figure 43 illustrates a doubly linked list.

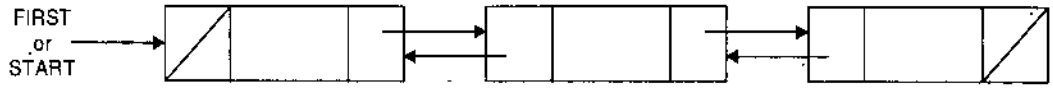


Fig. 43. *Illustration of a doubly linked list*

(*iv*) **Circular doubly linked lists.** For reaching any node from any other node, a circular list is very useful. In a similar way, it is useful to make a doubly linked list also a circular list. Figure 44 illustrates a circular doubly linked list.

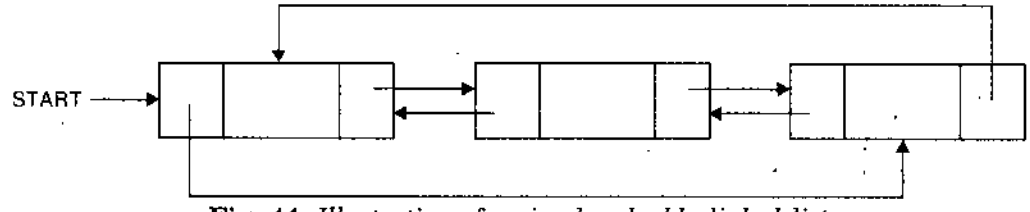


Fig. 44. *Illustration of a circular doubly linked list*

4.22 OPERATIONS ON SINGLY LINKED LISTS

Like other languages having pointer facility, we can treat a linked lists as a abstract data type and perform the following basic operations :

NOTES

(*i*) Creation of a list

(*ii*) Traversal

(*iii*) Count the number of elements (nodes)

(*iv*) Searching

(*v*) Insertion of a node

(*vi*) Deletion of a node

(*vii*) Modifying the contents of node

(*viii*) Reversal of a list

(*ix*) Concatenation of two lists

(*x*) Merging of two lists

(*xi*) Splitting of a list

(*xii*) Dividing a list into odd positioned and even positioned nodes

(*xiii*) Sorting etc.

Before we discuss the algorithms and implement the basic operations, we give below the vairables that are used in the algorithms with their meanings :

FIRST or START	:	This variable is of data type pointer which contains the address of the first node in the linked list. If FIRST or START contains NULL it means an empty linked list.
PTR	:	This is a variable of type pointer and contains the address of a node.
INFO(PTR) or DATA(PTR)	:	This variable contains the value stored in the data portion of the node pointed to by the pointer vairable PTR.
LINK(PTR) or NEXT(PTR)	:	This variable contains the value stored in the link field or next field of the node pointed to by the pointer variable PTR, which is the address of the next node in the linked list.

Successive nodes of the linked list can be accessed through the pointer variable PTR (say). The symbol ← in algorithms denotes the assignment operation.

4.22.1 Algorithm for Traversal in a Linked List

Traversing in a linked list means accessing each node in it successively starting from the beginning and making the desired changes in one or more data fields of the node. Let the desired operation be denoted by CHANGE without actually specifying what changes are to be made. For example CHANGE may specify printing data of all the nodes, count the number of nodes currently available etc. Initially, the PTR must have the value of START or FIRST and then successively, it must have the address of the next node so as to access the next node in the

NOTES

linked list. The algorithm will terminate when the entire linked list has been traversed, *i.e.*, when PTR is NULL. The nodes contain INFO and LINK fields as information and address to next node.

1. PTR ← START
2. Repeat upto step 4 while PTR ≠ NULL and goto step 5
3. Apply CHANGE to INFO(PTR)
4. PTR ← LINK(PTR)
5. End

Figure 45 illustrates the concept discussed above showing the different addresses stored at different times in vairable PTR (initially having START and finally NULL pointer) :

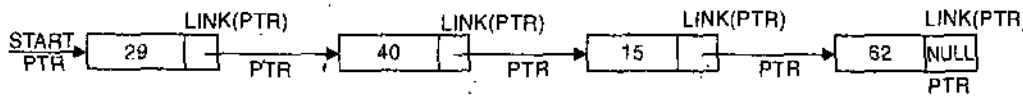


Fig. 45. *Traversing a Linked List*

4.22.2 Algorithm for Counting Number of Elements (Nodes)

Given FIRST, a pointer to the first element of linear list whose node contains INFO and LINK fields as information and address to next node. Suppose we want to count the number of nodes in linked list then the following algorithm will accomplish it :

1. COUNT ← 0
2. PTR ← START
3. Repeat upto step 5 while PTR ≠ NULL and goto step 6
4. COUNT ← COUNT+1
5. PTR ← LINK(PTR)
6. Write COUNT
7. End.

4.22.3 Searching an Element

As mentioned earlier, the **binary search** method **cannot** be applied on linked lists due to the fact that the location numbers of the elements (nodes) may not be continuous. It is one of the limitaions of linked lists as there is no way to find the location of the middle element of the list. Further, we may have the linked list as sorted or unsorted.

4.22.4 Linked List is Unsorted

Suppose we have a list of elements stored in a linked list. We want to find out whether the element VALUE is in the list or not. The pointer variable PTR is assigned the adress START, *i.e.*, the address of the first node in the linked list.

The searching process continuous until VALUE is found or the entire linked list is traversed. LOC gives the address of the desired element (VALUE). LOC is NULL if VALUE is not found in the linked list. The algorithm is given below :

1. PTR ← START
2. Repeat steps 3 and 4 while PTR ≠ NULL and goto step 5
3. If INFO(PTR) = VALUE Then
   goto step 5
4. PTR ← LINK(PTR)
5. LOC ← PTR
6. End.

Figure 46 shows the searching of a node having VALUE 8 in an unsorted linked list :

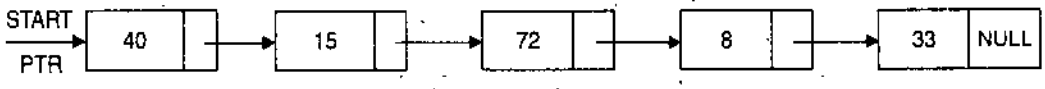


Fig. 46 (*a*) *Linked list having 5 nodes*

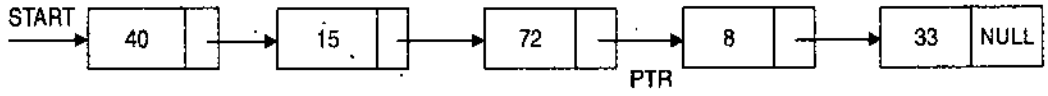


Fig. 46 (*b*) *Appropriate location searched for VALUE 8 in the linked list*

Searching is unsuccessful in case the desired element (VALUE) is not present in the linked list.

The following function in C implements the above concept :

```
/* function definition search()*/

void search(int value)
{
  node *ptr=start; /* local variable */
  while(ptr) /* while ptr != NULL */
  {
    if(ptr->info == value)
      {
        printf("\n\n%d found at location number %lu\n",value,ptr);
        return;
      }
    ptr=ptr->link;
  }
  printf("\n%d not found in the link list", value);
}
```

NOTES

4.22.5 Linked List is Sorted

Suppose we have a list of elements given in ascending ordr in a linked list, we want to find whether the element VALUE is in the list or not. The pointer variable PTR is assigned the address START, *i.e.*, the address of the first node in the linked list. The searching process may terminate when we get the desired VALUE or if any element of the linked list is found greater than VALUE or the entire linked list is traversed. LOC gives the address of the desired element (VALUE). LOC is NULL if VALUE is not found in the linked list. The algorithm is given below :

1. PTR ← START
2. Repeat while (LINK(PTR) ≠ NULL and INFO(PTR)<VALUE)
   PTR ← LINK(PTR)
3. If INFO (PTR) = VALUE Then
   LOC ← PTR
   Else
   LOC ← NULL
4. End

Figure 47 shows the searching of a node having value 58 in a sorted linked list :

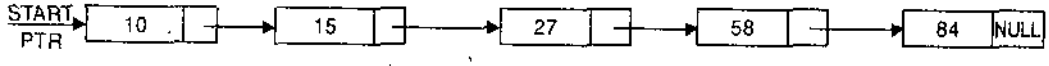


Fig. 47 (*a*) *Linked list having 5 nodes in ascending order of info*

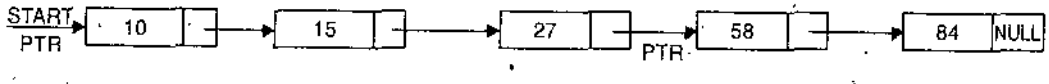


Fig. 47 (*b*) *Appropriate location searched for VALUE 58 in the ordered linked list*

Search is unsuccessful in case the desired element (VALUE) is not present in the linked list.

Following function in C implements the above concept :

```
/* function definition search() */

void search(int value)
  {
    node *ptr=start; /*.local variable */
    while( ptr->link != NULL && ptr->info < value)
      ptr=ptr->link;
    if(ptr->info == value)
    {
      printf("\n\n%d found at location number %lu\n", value,ptr);
    }
```

```
    else
    {
      printf("\n\%d not found in the link list", value);
    }
}
```

NOTES

4.22.6 Insertion into a Linked List

For inserting an element in a linked list, we first of all get a free node, assign the element to be inserted to the INFO field of the node, and finally place the new node at the appropriate position by proper pointer adjustment. The insertion can take place at any of the following positions :

- Insertion in the beginning of the linked list.
- Insertion in the end of the linked list.
- Insertion at the desired position in the linked list.
- Insertion into an ordered linked list (say in ascending order).

4.22.7 Insertion in the Beginning

Given VALUE a new element to be inserted, START a pointer which points to the first elements of the linear list whose node contains INFO and LINK fields as information and address of the next node. This algorithm inserts VALUE before the node being pointed to by START. AVAIL is a pointer that points to the top element of the availability stack. NEWPTR is a temporaly pointer variable.

Figure 48 shows the insertion of a node in the beginning of linked list :

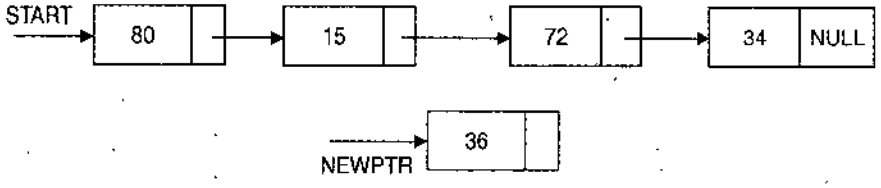


Fig. 48 (*a*) *Before inserting the node*

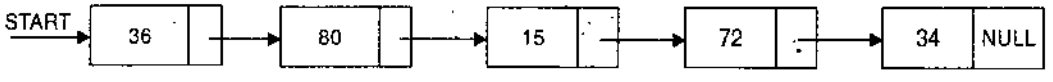


Fig. 48 (*b*) *After inserting the node in the beginning*

The algorithm for insertion in the beginning is given below :

1. [Check for memory availability?]

   If AVAIL = NULL Then

   ```
   {
     Write ('Availability stack underflow')
     goto step 6
   }
   ```

2. [Obtain address of next free node]
   NEWPTR ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialise fields of the new node obtained]
   INFO(NEWPTR) ← VALUE
   LINK(NEWPTR) ← START
5. [Make newnode as the first node of the list]
   START ← NEWPTR
6. End.

The above concept has been implement in Program 1.

4.22.8 Insertion in the End

Given VALUE a new element to be inserted, START a pointer which points to the first element of the linear list whose node contains INFO and LINK fields as information and address of the next node. This algorithm inserts VALUE in the end of the linked list. AVAIL is a pointer that points to the top element of the availability stack. NEWPTR and PTR temporary pointer variables. Figure 49 shows the insertion of a node in the end of linked list :

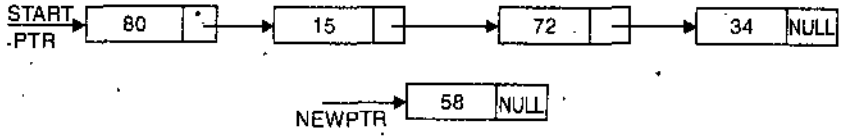


Fig. 49 (*a*) *Before inserting the node*

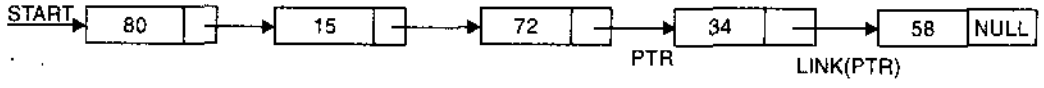


Fig. 49 (*b*) *After inserting the node in the end*

The algorithm for insertion in the end is given below :

1. [Check for memory availability ?]
   If AVAIL = NULL Then
   {
   `Write('Availability stack underflow')`
   `goto step 9.`
   }
2. [Obtain address of next free node]
   NEWPTR ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)

4. [Initialize fields of the new node obtained]
   INFO(NEWPTR) ← VALUE
   LINK(NEWPTR) ← NULL
5. [Check whether the link list is empty ?]
   If START = NULL Then
   {
   ```
   START ← NEWPTR
   goto step 9
   ```
   }
6. [Initialise search for the last node of list]
   PTR ← START
7. [Search for end of the list]
   Repeat while (LINK(PTR) ≠ NULL)
   PTR ← LINK(PTR)
8. [Add the node in the end]
   LINK(PTR) ← NEWPTR
9. End.

The above concept has been implemented in Program 2

4.22.9 Insertion at the Desired Position

This involves adding a new node to the linked list. The node can be added anywhere. Figure 50 shows the insertion of a node at the 4th position in the list :

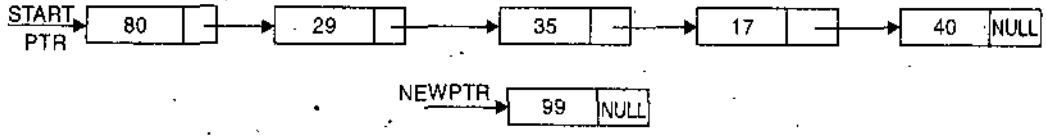


Fig. 50 (*a*) *Before inserting the node*

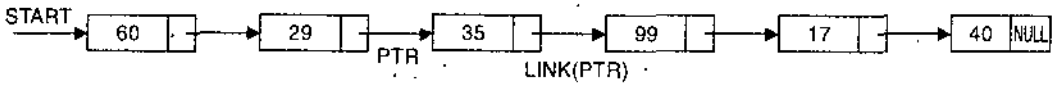


Fig. 50 (*b*) *After inserting the node at 4th position*

For insertion we perform the following steps :

(*i*) Allocate memory for the new node.

(*ii*) Enter data for the info part of the node.

(*iii*) Search for the appropriate position of insertion.

(*iv*) Modify the pointer(s) so that the new node is inserted at the desired position.

NOTES

The algorithm for insertion of VALUE at the desired positon (POSITION) is given below :

1. [Check for memory availability ?]

```
If AVAIL = NULL Then
{
   Write('Availability stack underflow')
   goto step 8
}
```

2. [Obtain address of next free node]

   NEWPTR ← AVAIL

3. [Remove freenode from availability stack]

   AVAIL ← LINK(AVAIL)

4. [Initialise fields of the new node obtained]

   INFO(NEWPTR) ← VALUE

   LINK(NEWPTR) ← NULL

5. [Initialise search for the desired position]

   PTR ← START

   STEPS ← 1

6. [Search for the desired position]

```
Repeat while(STEPS < POSITION – 1)
{
   PTR ← LINK(PTR)
   STEPS ← STEPS + 1
}
```

7. [Add the node in the list]

```
If(POSITION = 1) Then
   {
      LINK(NEWPTR) ← START
         START ← NEWPTR
   }
Else
   {
      LINK(NEWPTR) ← LINK(PTR)
      LINK(PTR) ← NEWPTR
   }
```

8. End

The following function in 'C' inserts a node at the desired position in a linked list :

NOTES

```
/* function definition insert() */

void insert(int position,int data)
 {
  node *newptr=NULL; /* local variable */
  node *ptr=start;
  int steps=1;

  /* create a new node */

  newptr= (node*) malloc(sizeof(node));
  newptr->info=data

  /* search for the desired position */

  while(steps<position-1)
    {
      ptr=ptr->link;
      steps++;
    }
  if(position==1) /* if node is to be inserted at the first place */
    {
      newptr->link=start;
      start=newptr;
    }
  else
    {
      newptr->link=ptr->link;
      ptr->link=newptr
    }
}
```

4.22.10 Insertion in an ordered (ascending order) linked list

This involves adding a new node to the linked list. The node is added in such a way that the ordering is preserved (*i.e.*, linked list remains sorted after insertion). Figure 51 shows the insertion of a node in a sorted linked list :

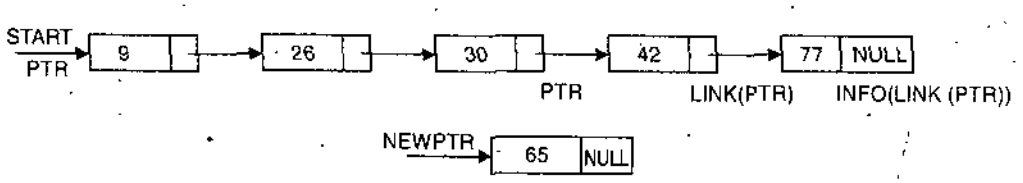


Fig. 51 (*a*) *Before inserting the node*

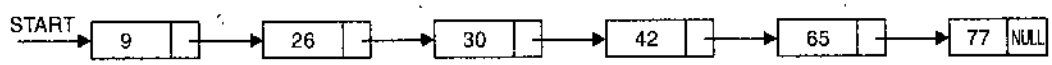


Fig. 51 (*b*) *After inserting the node*

NOTES

4.22.11 Algorithm for Insertion in an Ordered Linked List

Given START the pointer variable having the address of the first node in the linked list having its elements in ascending order. VALUE is the new element to be inserted. AVAIL is address of the topmost node of the availability stack. NEWPTR and PTR are temporary pointer variables. It is required that after insertion of new element the ordering of into fields is preserved.

1. [Check for memory availability ?]

   If AVAIL = NULL Then

   {

   ```
   Write('Availability stack underflow')
   goto step 6
   ```

   }

2. [Obtain address of next free node]

   NEWPTR ← AVAIL

3. [Remove free node from availability stack]

   AVAIL ← LINK(AVAIL)

4. [Initialize info field of the new node obtained]

   INFO(NEWPTR) ← VALUE

5. [Check if list is empty or new node preceeds all other nodes ?]

   If ((START = NULL) OR (INFO(NEWPTR) ≤ INFO(START)) Then

   {

   LINK(NEWPTR) ← START

   START ← NEWPTR

   }

   Else

   {

   [Search for the proper place of insertion, *i.e.,* intialize search]

   PTR ← START

   [Search for predecessor of new node]

   Repeat while((LINK(PTR) ≠ NULL) AND (INFO(LINK(PTR))< INFO(NEWPTR)))

   PTR ← LINK(PTR)

   [Insertion]

   LINK(NEWPTR) ← LINK(PTR)

LINK(PTR) ← NEWPTR

}

6. End

The following function in C implements the above concept :

NOTES

```
/* function definition insert() */

void insert(int data)
{
  node *newptr=NULL; /* local variable */
  node *ptr;

  /* create a new node and initialise it */

  newptr= (node *) malloc(size of(node));
  newptr->info=data;

  /* check if list is empty or new node preceedes all other nodes */

  if( (start==NULL) || (newptr->info <= start->info) )
  {
    newptr->link=start;
    start=newptr;
  }
  else
  {
    /* search for the proper place of insertion */

    ptr=start; /* initialse ptr with start */

    while((ptr->link!=NULL)&&((ptr->link)->info<(newptr->info)))
      ptr=ptr->link;
    /* insertion */
      newptr->link=ptr->link;
      ptr->link=newptr;
  }
}
```

4.22.12 Deletion of a Node

This involves the deletion of a node from the linked list. The deletion can be from any where in the list. *For example,*

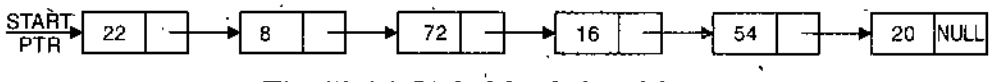


Fig. 52 (*a*) *Linked list before deletion.*

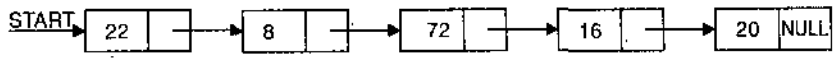


Fig. 52 (*b*) *Linked list after deletion of node from position 5.*

NOTES

The following steps are followed to delete a node from a given position :

(*i*) Store the address of START in a temporary pointer (say PTR).

(*ii*) Search for the desired position by traversing the list.

(*iii*) If the first node is to be deleted then make the pointer LINK (START) as the new START and free *i.e.,* deallocate the memory occupied by the temporary pointer which was having the address of initial START poitner and stop.

(*iv*) Store the pointer of the LINK field of the node to be deleted in the LINK of the previous of the node to be deleted and free the memory occupied by the temporary pointer having the address of the node to be deleted.

The algorithm for deletion from any position in a linked list is given below : Given POS the position of deletion, START a pointer which points to the first element of the linear list whose node contains INFO and LINK fields as information and address of the next node. This algorithm deletes the node from the specified position (if possible). AVAIL is a pointer that points to the top element of the availability stack. PTR and TEMP are temporary pointer variables. After deletion the deleted node is returned to the availablitiy stack. START will be changed only when POS is 1. Also POS lies between 1 and TOTAL (total number of nodes in the linked list).

```
1.  If (START = NULL) Then
    {
      Write('Linked list has no nodes so deletion not possible')
        goto step 8
    }
2.  If (POS ≤ 0 OR POS > TOTAL) Then
    {
      Write('Wrong position number given')
        goto step 8
    }
3.  If POS = 1 Then
    {
      TEMP ← START
      START ← LINK(START)
      goto step 7
    }
```

NOTES

4. STEPS ← 1
   PTR ← START
5. Repeat while (STEPS < POS – 1)
   {
     PTR ← LINK(PTR)
     STEPS = STEPS + 1
   }
6. TEMP ← LINK(PTR)
   LINK(PTR) ← LINK(LINK(PTR))
7. [Return node to availability stack]
     LINK(TEMP) ← AVAIL
       AVAIL ← TEMP
8. End.

The following function in C implements the above concept :

```
/* function definition del_node()*/

void del_node(int position)
{
  node *temp,*ptr=NULL; /* local variable */
  int steps=1;

  /* search for desired position */

  if(position==1) /* if element is to be deleted from start */
    {
     temp = start;
     start=start->link;
     free(temp); /* deallocate memory */
    }
  else
    {
      ptr=start;
      while(steps<position-1)
        {
          ptr=ptr->link;
          steps++;
        }
      temp = ptr->link;
      ptr->link=ptr->link->link;
      free(temp); /* deallocate memory */
    }
}
```

NOTES

4.22.13 Modifying the Contents of a Node

This involves the replacement of the info of an existing node by a new value. It is shown in the following example :

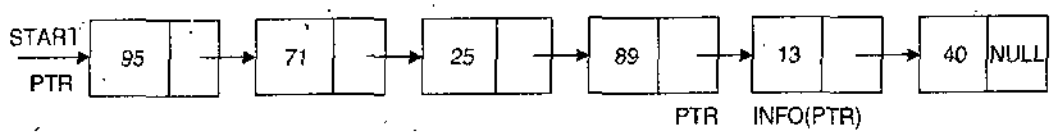


Fig. 53 (*a*) *Linked list before modification*

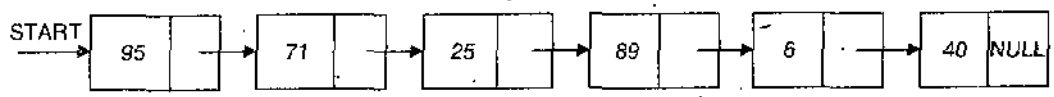


Fig. 53 (*b*) *Linked list after modification of contents of node number 5*

The following steps are followed to delete a node from a given position :

(*i*) Store the address of *START* in a temporary pointer (say PTR).

(*ii*) Search for the desired position by traversing the list.

(*iii*) Replace the old contents by new contents (value), if node found.

The algorithm for modifying the contents of a node is given below :

Given START a pointer variable storing the address of the first element in the linked list. Each node in the linked list (if any) has INFO and LINK as its information and pointer field (having the address of new node) respectively. PTR is a temporary pointer. OLDVALUE is the info to be replaced and NEWVALUE is the info to replace it.

1. If (START = NULL) Then
   {
   `Write('Linked list has no nodes so modification not possible')`
   goto step 5
   }
2. PTR←START
3. Repeat while (INFO(PTR) ≠ OLDVALUE) AND (LINK(PTR)≠NULL))
   PTR←LINK(PTR)
4. If (INFO(PTR) = OLDVALUE)
   INFO(PTR) ← NEWVALUE
   Else
   Write('The node to be modified not found in the linked list')
5. End

The following function in C implements the above concept :

NOTES

```
/* function definition modify() */

void modify(int oldvalue, int newvalue)
{
  node *ptr=start; /* initialise ptr with start */

  /* search for the desired info (first occurence only) */

  while(ptr->info != oldvalue && ptr->link != NULL)
    ptr=ptr->link;

  /* if info found then replace oldvalue by newvalue */

  if(ptr->info == oldvalue)
    ptr->info=newvalue;
  else
    {
     printf("\n\n%d not found in the linked list\n",oldvalue);
     del_list(); /* deallocate the memory */
     getch();
     exit();
    }
}
```

4.22.14 Reversal of a Linked List

Given a linked list pointed to by START whose nodes have INFO and LINK fields respectively. PREVIOUS, CURRENT and NEXTPTR are temporary pointer variables used for swapping of pointers and traversal in the linked list. Let us take a linked list of 4 nodes as shown below :

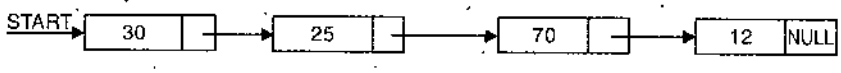


Fig. 54 (*a*) *Linked list having 4 nodes.*

The following steps are followed for reversing it :

1. We begin traversing the linked list by initially taking the address of first node as CURRENT and address of its predecessor node as PREVIOUS which is presently NULL. While traversing the linked list we perform the following changes in the address of nodes (*i.e.*, pointers) until we reach in the end of the linked list.
   (*i*) Store the LINK(CURRENT) in NEXTPTR
   (*ii*) Replace the LINK(CURRENT) by PREVIOUS
   (*iii*) Make the CURENT pointer as PREVIOUS
   (*iv*) Make the NEXTPTR as CURRENT

NOTES

2. At last when CURRENT is NULL, set START to store the address of the last node in the given linked list, *i.e.*, PREVIOUS.

These pointer changes are shown in figure 54(*b*) :

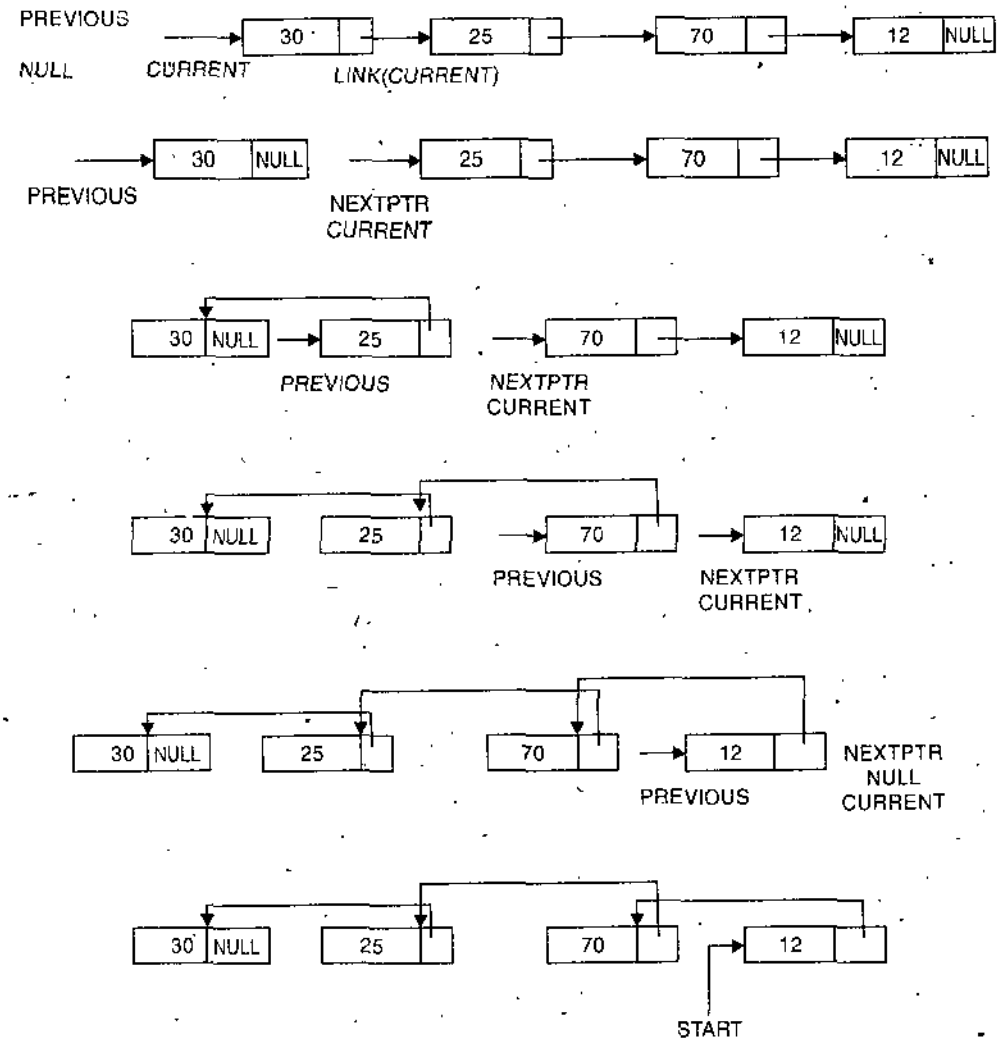


Fig. 54 (*b*) *Illustration of reversal of a linked list.*

The algorithm for reversing a linked list is given below :

Given START the address of the first node in the linked list. PREVIOUS, CURRENT and NEXTPTR and temporary pointer variables for swapping of pointers. The node of the linked list has INFO and LINK as information and address of next node respectively.

1. If (START = NULL OR LINK(START) = NULL) Then

   goto step 5.

2. PREVIOUS ← NULL

   CURRENT ← START

NOTES

3. Repeat while (CURRENT ≠ NULL)

```
{
        NEXTPTR  ← LINK(CURRENT)
   LINK(CURRENT) ← PREVIOUS
        PREVIOUS ← CURRENT
        CURRENT  ← NEXTPTR
}
```

4. START ← PREVIOUS
5. End

4.22.15 Concatenation of Two Singly Linked Lists

Suppose we are given two linked lists having FIRST and SECOND as the addresses of their first nodes as shown below :

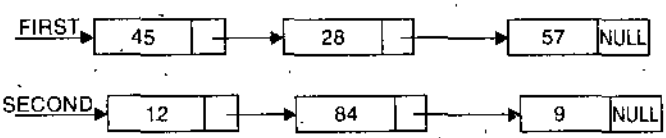


Fig. 55 (*a*) *Linked lists before concatenation*

We want to produce a new singly linked list which is to be pointed to by a pointer variable THIRD. This new list is to be obtained by concatenating the two linked lists pointed to by FIRST and SECOND, as shown below :

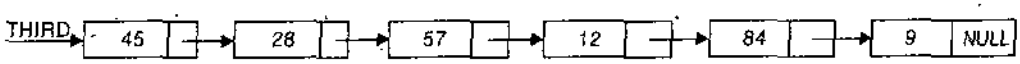


Fig. 55 (*b*) *Concatenated linked list*

The following steps are peformed for concatenation :

1. Copy the pointer FIRST to THIRD.
2. Traverse the first list until the end of the linked list is reached, using a temporary pointer variable PTR.
3. Change the NULL pointer in the last node of the first linked list so as to point to the first node of the second linked list pointed to by pointer variable SECOND.

The algorithm for concatenation of two linked lists is given below :

Given two linked lists being pointed to by pointer variables FIRST and SECOND respectively. We are to concatenate these two linked lists such that the second linked list is joined after the first linked list. PTR is a temporary pointer variable.

1. *THIRD ← FIRST*
2. If (FIRST = NULL) Then

```
{
   THIRD ← SECOND
```

```
            goto step 7
        }
```

3. If (SECOND = NULL) Then
   goto step 7
4. PTR ← FIRST
5. Repeat while (LINK(PTR) ≠ NULL)
   PTR ← LINK(PTR)
6. LINK(PTR) ← SECOND
7. End

NOTES

4.23 CIRCULAR LINKED LINEAR LIST

So far we have discussed linked linear lists in which the last node contains the NULL pointer. A slight change in the linked linear list results in a further improvement in processing of list. This is done by replacing the NULL pointer in the last node of a linked list with the address of its first node. Such a linked list is called a **circular linked linear list** or simply a **circular list.** Figure 56 illustrates the structure of a non-empty circular list having no NULL pointer.

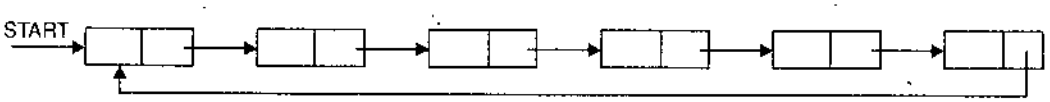


Fig. 56. *Circular linked list having 5 nodes*

In a circular linked list travessal is only in the forward direction.

4.23.1 Advantages of Circular List Over Singly Linked Lists

Main advantages are given below :

1. In a circular list every node of the list is accessible from a given node. That is, from this given node, all nodes can be visited by simply chaining through the list.
2. The deletion of a node does not require the address of the first node of the linked list (as in case of singly linked list, where searching takes place from the first node of the list). The searching of an element can take place from the address of node itself, when its address is given.
3. Operations like concatenation and splitting become more efficient in case of circular lists.

The memory declarations and allocation for representing circular linked lists are the same as for linear linked list.

All the operations that are allowed on singly linked linear lists can be easily implemented for circular linked lists, except the following :

1. The LINK field of the newly inserted last node in the linked list points to the first node.
2. While checking for the end of the circular linked list, we compare the LINK field with the address of the first node.

4.23.2 Disadvantage of Circular Lists

When we are traversing a circular list, we must be careful as there is a possiblity to get into an infinite loop! In processing a circular list, it is important that we are able to detect the end of the list. We can help guarantee the detection of the end by placing a special node which can be easily identified in the circular linked list. This special node is often called the ***header node*** of the circular linked list. The main advantage of this technique is that the linked list can never be empty. As most of the algorithms require the testing of a linked list as to whether it is empty, we observe that this advantage is really important. Figure 57 shows a circular linked list with a header node, where the variable HEAD denotes the address of the heder node.

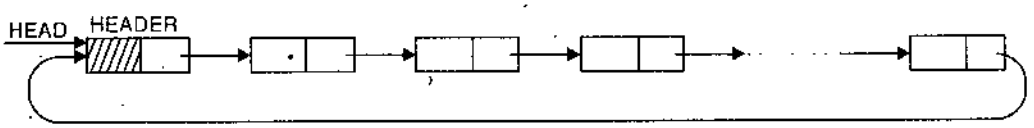


Fig. 57. *A circular linked list with a heder node*

Note that the INFO field in the header node is not used, which is shown by shaded field. Header node may contain length of the list or any other information in its INFO field. An empty circular linked list is represented by having LINK (HEAD) = HEAD, as shown in figure 58 :

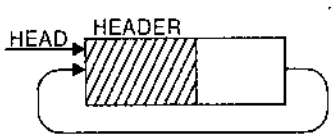


Fig. 58. *An empty circular linked list with a header node*

The algorithm for insertion of a node at the head of a circular linked list with a header node is given below :

Given a circular linked list having starting address denoted by HEAD. Avail denotes the address of the top most node of the availability stack. NEWPTR is a temporary pointer. VALUE denotes the INFO of the node to be inserted.

1. [Check the availability stack ?]

   IF (AVAIL = NULL) Then

   {

```
        Write('No free memory')
        goto step 5
    }
```

NOTES

2. [Obtain address of next free node]
   NEWTPR ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and its link to the list]
   INFO(NEWPTR) ← VALUE
   LINK(NEWPTR) ← LINK(HEAD)
   LINK(HEAD) ← NEWPTR
5. End

Figure 59 illustrates the insertion of a node at the head of a circular linked list :

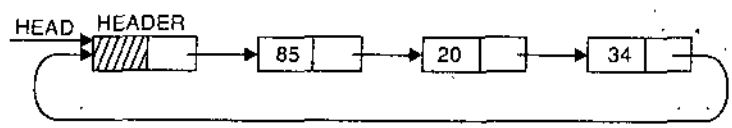


Fig. 59 (*a*) *A circular linked list having 3 nodes*

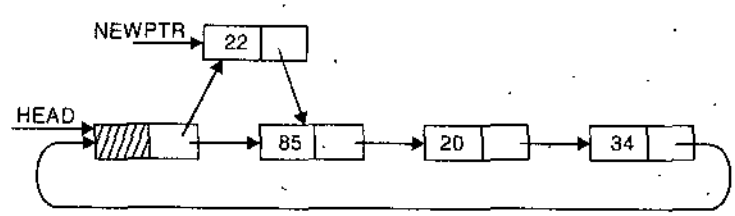


Fig. 59 (*b*) *Circular linked list after insertion of node at the head with info 22*

4.24. APPLICATIONS OF LINEAR LINKED LISTS

Like arrays, linked list is a very useful data structure. It can be used for implementing the following :

(*i*) To model many different abstract data types such as stacks, queues, trees and graphs.

(*ii*) Polynomial representation and manipulation.

(*iii*) To maintain a dictionary of names.

(*iv*) To perform arithmetic operations to some arbitrary precision.

(*v*) To represent sparse matrices.

Let us describe polynomial representation and manipulation.

4.24.1 Polynomial representation and manipulaton using linked lists

NOTES

We can represent a polynomial such as :

$$x^4 + 5x^2 - 7x + 4$$

using a linked list shown in figure 60 :

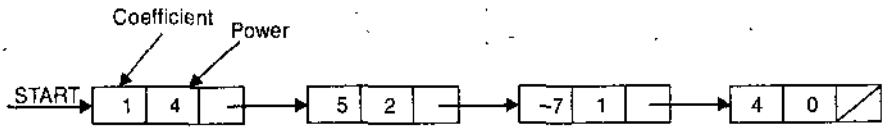


Fig. 60. *Representation of a polynomial using a linked list*

Here, each term of the polynomial is represented by a node. A node is of fixed size having three fields, first representing the coefficient, second representing the power or exponent and the third is a pointer to the next node of the list. So the node structure is represented as shown in figure 61 :

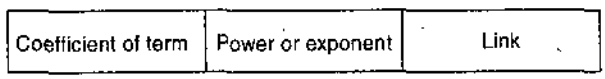


Fig. 61. *Node structure to represented a polynomial term*

Thus the declaration of above shown node type in C language, having integer coefficients are given below :

```
typedef struct nodetype /* create node type */
{
   int coeff;
   int power;
   struct nodetype *link; /* pointer to the next node */
}node;
node  *start;
```

In order to achieve greater efficiency in processing, the polynomial can be stored in decreasing order of powers by term.

4.25 DOUBLY LINKED LIST OR TWO WAY CHAINS

So far, we have been restricted to traversing linked linear lists in one direction. There are situations when it is required and many times indispensable that a list be traversed in both directions, that is, either forward or backward. In such a situation each node must have two link or pointer fields instead of usual one link field. The links are used to denote the predecessor and successor of a node in the doubly linked list. The link or pointer storing the address of the predecessor of a node is called the *left* link, and the link or pointer that storing the address

NOTES

of its successor its *right* link. A linked list having this type of node is called a *doubly linked list* or a *two way chain*. Figure 62 represents a doubly linked list, where START and LAST are pointer variables storing the address of the left-most and right-most nodes in the linked list, repectively.

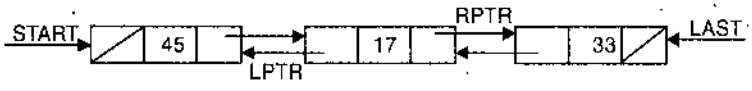


Fig. 62. *A doubly linked linear list having 3 nodes*

The left link of the left-most node and the right link of the right-most node are both NULL, representing the end of the list for each direction, that is, the first node in a doubly linked list has no predecessor and the last node has no successor. The variables LPTR and RPTR are used to denote the left and right links of a node, respectively.

4.25.1 Representation of Doubly Linked Lists

Suppose we wish to have a doubly linked list storing integers, then the following declaration in C language can be used for a node structure :

```
typedef struct doubly_list
   {
      struct doubly_list *lptr;
      int info;
      struct doubly_list *rptr;
   } node;
   node *start,*last;
```

Now the following statement creates a node dynamically :

```
start = (node *) malloc (sizeof (node));
if (start == NULL)
printf("\nAvailability stack underflow\n");
```

4.26 OPERATIONS ON A DOUBLY LINKED LIST

The following operations can be performed on a doubly linked list :

1. Make an empty doubly linked list
2. Traverse the given doubly linked list
3. Insert new nodes in the doubly linked list
4. Delete existing nodes from the doubly linked list

1. Make an empty doubled linked list

After declaring the node structure an empty doubly linked list is created just by the following statement :

```
start = last = NULL;
```

2. Traverse the given doubly linked list

Traversing a doubly linked list is similar to traversing a singly linked list. But in a doubly linked list traversing can be done in both directions. For traversing the doubly linked list in forward direction, we make use of forward links whereas for traversing the doubly linked list in reverse direction there are two ways

(*i*) First we traverse the list in forward direction and then in the backward direction. This type of traversal is generally used when the LAST pointer is not mentioned.

(*ii*) Traverse the doubly linked list from end to beginning, that is, in the backward direction using the pointer LAST to begin the traversal and using backward links.

The algorithm for traversal in a doubly linked list is given below:

Given START and LAST, the pointers to the first and last nodes of the doubly linked list whose node contains LPTR, INFO and RPTR fields as left link, information and right link respectively. MOVE denotes a temporary pointer variable. Let the desired operation while traversing be denoted by CHANGE.

1. MOVE ← START
2. [Traversing in the forward direction]

```
Repeat while MOVE ≠ NULL
   {
      Apply CHANGE to INFO(MOVE)
         MOVE ← RPTR(MOVE)
   }
```

3. MOVE ← LAST
4. [Traversing in the backward direction]

```
Repeat while MOVE ≠ NULL
   {
      Apply CHANGE to INFO(MOVE)
         MOVE ← LPTR(MOVE)
   }
```

5. End

3. Insert new nodes in the doubly linked list

For inserting an element in a doubly linked list, we first of all get a free node, assign the element to be inserted to the INFO field of the node, and finally place the new node at the appropriate position by proper pointer adjustment. The insertion can take place at any of the following positions :

- Insertion in the beginning of the doubly linked list
- Insertion in the end of the doubly linked list

- Insertion at the desired position in the doubly linked list
- Insertion into an ordered doubly linked list (say in ascending order)

NOTES

4.26.1 Insertion in the beginning of the doubly linked list

Given VALUE a new element to be inserted, START and LAST the pointers which point to the first and last element of the doubly linked list whose node contains LPTR, INFO and RPTR fields as left link, information and right link respectively. This algorithm inserts VALUE before the node being pointed to by START. AVAIL is a pointer that points to the top element of the availability stack. NEWPTR is a temporary pointer variable. Initially START and LAST are NULL. Figure 63 shows the insertion of a node in the beginning of the doubly linked list :

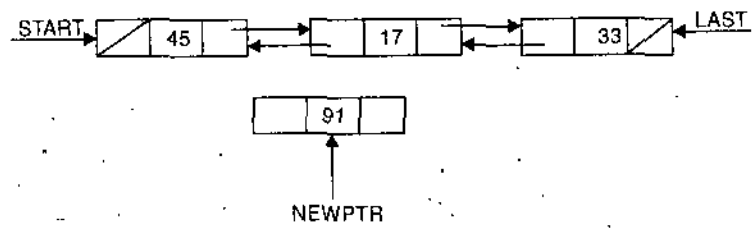


Fig. 63 (*a*) *Before inserting the node*

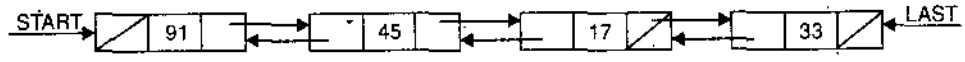


Fig. 63 (*b*) *After inserting the node in the beginning*

The algorithm for insertion in the beginning of the doubly linked list is given below :

1. [Check for memory availability?]

```
IF AVAIL = NULL Then
{
   Write('Availability stack underflow')
   goto step 6
}
```

2. [Obtain address of next free node]

   NEWPTR ← AVAIL

3. [Remove free node from availability stack]

   AVAIL ← LINK(AVAIL)

4. [Initialize fields of the new node obtained]

   INFO(NEWPTR) ← VALUE

   LPTR(NEWPTR) ← NULL

   RPTR(NEWPTR) ← START

5. [Make new node as the first node of the doubly linked list]

```
If(LAST = NULL) Then
{
   START ← NEWPTR
   LAST ← NEWPTR
}
Else
{
   LPTR(START) ← NEWPTR
   START ← NEWPTR
}
```

6. End

4.26.2 Insertion in the end of the doubly linked list

Given VALUE a new element to be inserted, START and LAST the pointers which point to the first and last element of the doubly linked list whose node contain LPTR, INFO and RPTR fields as left link, information and right link respectively. This algorithm inserts VALUE in the end of the doubly linked list AVAIL is a pointer that points to the top element of the availability stack. NEWPTR is a temporary pointer variable. Initially START and LAST are NULL. Figure 64 shows the insertion of a node in the end of the doubly linked list :

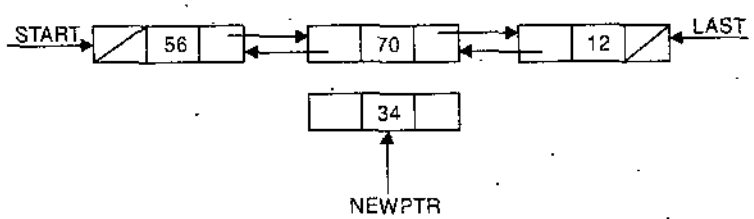


Fig. 64 (*a*) *Before inserting the node*

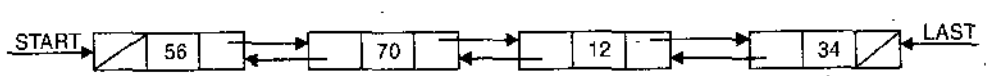


Fig. 64 (*b*) *After inserting the node in the end of the doubly linked list*

The algorithm of insertion in the end of the doubly linked list is given below :

1. [Check for memory availability ?]

```
IF AVAIL = NULL Then
{
   Write('Availability stack underflow')
   goto step 6
}
```

2. [Obtain address of next free node]

   NEWPTR ← AVAIL

NOTES

3. [Remove free node from availability stack]

   AVAIL ← LINK(AVAIL)

4. [Initialize fields of the new node obtained]

   INFO(NEWPTR) ← VALUE

   RPTR(NEWPTR) ← NULL

5. [Make new node as the last node of the doubly linked list]

```
If (LAST = NULL) Then
{
   LPTR(NEWPTR) ← NULL
   START ← NEWPTR
   LAST ← NEWPTR
}
Else
{
   RPTR(LAST) ← NEWPTR
   LPTR(NEWPTR) ← LAST
   LAST ← NEWPTR
}
```

6. End

The following functions in 'C' implement the insertion in the beginning and at end of a doubly linked list :

```
/* function definition insert_beginning() */

void insert_beginning(int data)
{
  node *newptr=NULL; /* local variable */

  /* create a new node and initialise it */

  newptr = (node *) malloc(size of(node));

  newptr->info=data;
  newptr->lptr=NULL;
  newptr->rptr=start;

  /* make new node as the first node of the doubly linked list */

  if(last==NULL) /* if the linked list is empty */
    start=last=newptr;
  else
```

NOTES

```
    {
     start->lptr=newptr;
     start=newptr;
    }
}

/* function definition insert_end() */

void insert_end(int data)
{
  node *newptr=NULL; /* local variable */

  /* create a new node and initialise it */

  newptr = (node *) malloc(size of(node));
  newptr -> info = data;
  newptr -> rptr = NULL;

  /* make new node as the last node of the doubly linked list */

  if(last==NULL) /* if the linked list is empty */
   {
    newptr->lptr=NULL;
   start=last=newptr;
   }
  else
   {
    /* insert the node in the end of linked list /
    last->rptr=newptr;
    newptr->lptr=last;
    last=newptr;
   }
}
```

4.26.3 Insertion at the desired position in a doubly linked list

This involves adding a new node to the doubly linked list. The node can be added anywhere. Figure 65 shows the insertion of a node at the 3rd position in the doubly linked list :

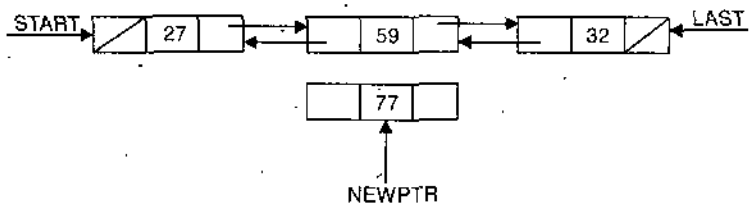


Fig. 65 (*a*) *Before inserting the node*

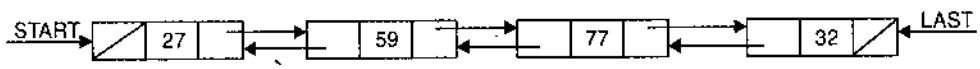


Fig. 65 (*b*) *After inserting the node at 3rd position*

NOTES

For insertion we perform the following steps :

(*i*) Allocate memory for the new node.

(*ii*) Enter data for the info part of the node.

(*iii*) Search for the appropriate position of insertion.

(*iv*) Modify the pointers so that the new node is inserted at the desired position.

The algorithm for insertion of VALUE in a doubly linked list (having COUNT number of nodes) at the desired position is given below :

1. [Check for memory availability ?]

```
IF AVAIL = NULL Then
{
   Write('Availability stack underflow')
   goto step 8
}
```

2. [Obtain address of next free node]

   NEWPTR ← AVAIL

3. [Remove free node from availability stack]

   AVAIL ← LINK(AVAIL)

4. [Initialize info field of the new node obtained]

   INFO(NEWPTR) ← VALUE

5. [Check if insertion is in the beginning]

```
IF (POSITION = 1) Then
{
   LPTR(NEWPTR) ← NULL
   RPTR(NEWPTR) ← START
   If LAST = NULL Then
      {
         START ← NEWPTR
         LAST ← NEWPTR
      }
   Else
      {
         LPTR(START) ← NEWPTR
         START ← NEWPTR
      }
   goto step 8
}
```

NOTES

6. [Check if insertion is in the end]

```
IF (POSITION = COUNT + 1) Then
{
   RPTR(NEWPTR) ← NULL
   RPTR(LAST) ← NEWPTR
   LPTR(NEWPTR) ← LAST
   LAST ← NEWPTR
   goto step 8
}
```

7. [Search for the desired position and insert]

```
MOVE ← START
STEPS ← 1
REPEAT WHILE (STEPS <POSITION)
  {
    MOVE ← RPTR(MOVE)
    STEPS ← STEPS + 1
  }
    LPTR(NEWPTR) ← LPTR(MOVE)
    RPTR(NEWPTR) ← MOVE
    LPTR(MOVE) ← NEWPTR
    RPTR(LPTR(NEWPTR)) ← NEWPTR
```

8. End

4.26.4 Insertion into an ordered doubly linked list (ascending order)

This involves adding a new node to the doubly linked list. The node is added in such a way that the ordering is preserved (that is, linked list remains sorted after insertion). Figure 66 shows the insertion of a node in a sorted doubly linked list :

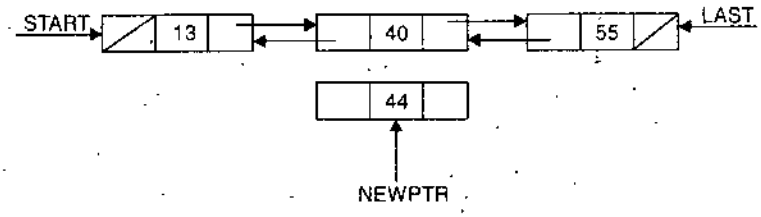


Fig. 66 (*a*) *Before inserting the node*

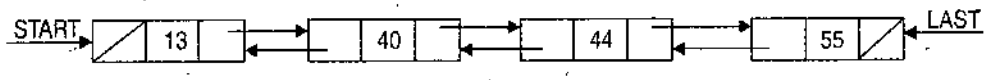


Fig. 66 (*b*) *After inserting the node*

4. Deletion of a node from a doubly linked list

This involves the deletion of a node from the doubly linked list. A number of possibilities are there. If the doubly linked list has a single node, then a deletion results in a empty list with the left-most and right-most pointers being set to NULL. The node being deleted could be the left-most node of the doubly linked list. In this case the pointer variable START must be changed. Similar situation can arise in case we want to delete the right-most node of the doubly linked list. The deletion of a node can take place from the middle of the doubly linked also. For example,

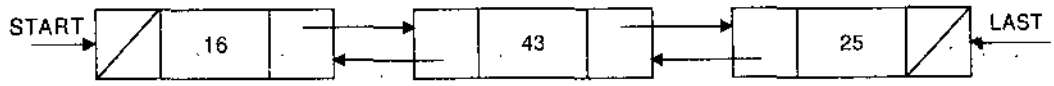


Fig. 67 (*a*) *Doubly linked list before deletion*

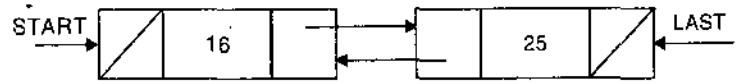


Fig. 67 (*b*) *Doubly linked list after deletion of node from position 2*

The following steps are followed to delete a node from a given position :

(*i*) If the doubly linked list is empty then write underflow and return.

(*ii*) If the doubly linked list has only one node then

Set the left and right pointers of the list to NULL

Else

If the leftmost node in the doubly linked list being deleted then

Delete the node and change the left pointer of the list

Else

Search for the position of the node to be deleted

If the rightmost node in the doubly linked list being deleted then

Delete the node and change the right pointer of the list

Else

Delete the node from the middle of the doubly linked list

(*iii*) Deallocate the memory used by the node and return

4.27 SUMMARY

- An array is a collection of the homogeneous (same type) elements that are referred by a common name.
- An array is also called a subscripted variable as the elements of an array are used by the name of an array and an index or subscript.

NOTES

- Arrays are of two types :
  (*i*) one-dimensional arrays
  (*ii*) multi-dimensional arrays (2 or more).
- Stacks and Queues can be implemented using arrays.
- Traversing means visiting each element (from start to end) one after the other.
- Insertion is not possible if the array is already full but replacement of an existing element is possible.
- Stack is an ordered collection of items into which new items may be inserted and from which items mays be deleted at one end, called top of the stack.
- Insertion in a stack is called PUSH and deletion as POP.
- Stack is a container, which follows LIFO principle.
- Stack can be implemented as an array and as a linked list.
- A queue is an ordered list in which all insertions take place at one end, the *rear*, where as all deletion takes place at other end, the *front*. Therefore, 'Queues' work on the concept of First In First Out **(FIFO)** principle.
- The process of inserting items is called *enqueueing*, and removing item from a queue is called *dequeueing*.
- A Deque is a linear list in which the elements can be added or removed at either end but not in middle.
- A Priority Queue is a collection of elements such that each element has been assigned a priority and based on the order in which elements are deleted and processed.
- Linked list is possible to grow and shrink size at any time.
- A linked list is a chain of structures in which each structure consists of data as well as pointers, which store the address (link) of the next logical structure in the list.

4.28 TEST YOURSELF

Answer the following questions :

1. What is an array ? Discuss its different types.
2. Write an algorithm for insertion of an element in a sorted array.
3. List the disadvantage(s) of implementing stacks as arrays. Describe means for overcoming the problems.
4. Write a C program to read an array A of integers and push all even numbers and odd numbers read into two stacks and then display them by popping the stacks.

NOTES

5. Declare a stack using array that contains int type numbers, and define pop and push function using C syntax.
6. How will you know that a linear queue is full ?
7. Write two applications of queue.
8. What is the drawback of linear queue ?
9. What is a dequeue ?
10. What is a priority queue ?
11. List some of the disadvantages of implementing queues as arrays. How will you overcome the problems ?
12. *Let Q be a non empty queue and S be an empty stack. Write a C program* to reverse the order of items in Q.
13. Write an algorithm and a function in C for insertion of a node in a sorted linked list given in ascending order.
14. What is a linked list ?
15. Describe different types of linked lists.
16. Write a short note on applications of linked lists.
17. Describe the procedures for inserting and deleting nodes from a double linked list with an example.
18. Differentiate between Circular and Doubly–linked lists.

❑❑❑

SECTION C

CHAPTER 5 TREES

★ LEARNING OBJECTIVES ★

5.1 Introduction
5.2 General Trees
5.3 Binary Tree
5.4 Properties of Binary Trees
5.5 Implementation of Binary Trees
5.6 Binary Tree Traversal Methods
5.7 Binary Tree Traversal Algorithms Using Stacks (*i.e.*, Iterative Algorithms)
5.8 Binary Search Tree
5.9 Summary
5.10 Test Yourself

5.1 INTRODUCTION

So far in the text we have discussed linear data structures such as arrays, stacks, queues and linked lists. Each element in these data structures is followed by one next element. There is another type of data structure which is non-linear data structure. In a non-linear data structure, each element may have more than one next element. One such non-linear data structure is a **tree.**

5.2 GENERAL TREES

A general tree T is a finite set of zero, one or more nodes (or elements) (R_1, R_2, ..., R_n) such that

NOTES

(*i*) There is one specially designated node called the root of the tree. Let it be denoted by R_1.

(*ii*) The remaining nodes R_2, R_3, ..., R_n are partitioned into m ($m \geq 0$) disjoint subsets, each of which is itself a tree. These trees may be denoted by T_1, T_2, ..., T_m.

T_1, T_2, ..., T_m are known as **subtrees of tree T.**

5.2.1 Empty Tree

A tree with no nodes is called an **empty tree.**

Representation of Tree

There are many ways to represent a tree structure. One of the simplest way of drawing a tree in computer science is upside down so that the root of the tree is at the top and the branches are in the downward direction as shown in figure 1.

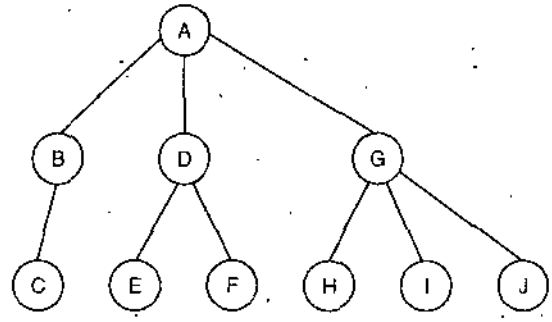


Fig. 1. *A general tree.*

Here the root of the tree is A. There are three subtrees of the root A, with roots B, D and G. The root B has one subtree. The root C has empty subtree. The root D and G have two and three subtrees respectively. The nodes (except root) having no subtrees are called leafs or terminal nodes. Any node (except the root and leafs) is called non-terminal node.

5.2.2 Level of a Node

The level of a node is equal to the length of the path from the root to the node.

The root of the tree has level = 0.

For example in figure 1 the levels are :

Level of A = 0

Level of B, D, G = 1

Level of C, E, F, H, I, J = 2

NOTES

Degree of a Node

The degree of a node is the number of children it has. The degree of a leaf is 0. For example, in figure 1 the degree of node G is 3.

Degree of a Tree

The degree of a tree is the maximum of its node degrees. The degree of tree in figure 1 is 3.

Height/Depth of a Tree

If level of the root is denoted by 0, then

Height/Depth of tree = 1 + maximum {levels of all nodes in the tree}

The height/depth of tree given in figure 1 is 3 (as maximum level is 2).

Parent and Child Relationship

In figure 1, the node A is **parent** of B, D and G.

B, D and G are called **children** of A.

C is the child of B.

E and F are children of D.

H, I and J are children of G.

The children nodes of a given parent node are called **siblings** or **brothers.**

There are many applications of general trees. However, a special class of general trees is binary tree, which is a very useful data structure.

5.3 BINARY TREE

A tree is called a binary tree if it has a finite set of nodes that is either empty or contains a single element called the root of the tree and all the other nodes are partitioned into two disjoint subsets, each of which itself is a binary tree.

The two subsets are called the **left subtree** and the **right subtree** of the original **tree.**

Let us consider the binary tree shown in figure 2.

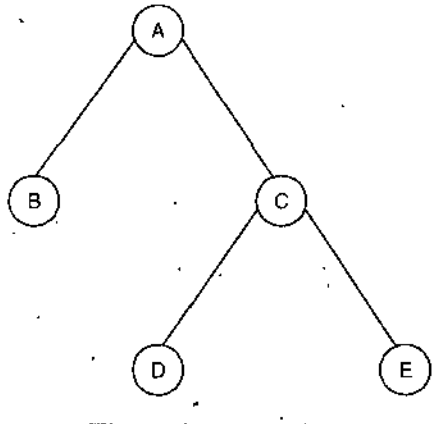


Fig. 2 *A Binary Tree.*

NOTES

Here, A is the root of the tree and the following two subtrees are :

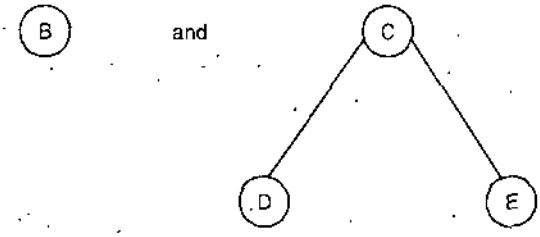


Fig. 3

The left subtree of the original tree is (B) and it has no subtrees.

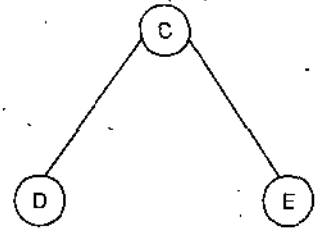
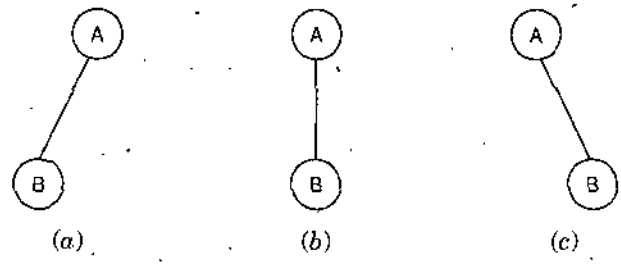


Fig. 4

The right subtree has root as (C) and two subtrees are (D) and (E).

Binary trees are different from General **trees.** A binary tree has left and right subtrees, whereas, in general trees there is no left or right **subtree.**



(*a*) (*b*) (*c*)

Fig. 5

In figure 5 (*a*) is having a left subtree and no right subtree. Figure 5 (*b*) has no meaning and figure 5 (*c*) is having a right subtree.

In figure 5 (*a*) B is called the left child of A and in figure 5 (*c*) B is called the right child of A.

5.4 PROPERTIES OF BINARY TREES

Property 1. *The drawing of every binary tree with n elements, n > 0, has exactly n–1 edges.*

Proof : Every element in a binary tree (except the root) has exactly one parent. There is exactly one edge between each child and its parent. So the number of edges is exactly n–1.

Property 2. *A binary tree of height h, $h \geq 0$, has at least h and at most $2^h - 1$ elements in it.*

Proof : Since there must be at least one element at each level, the number of elements is at least h. As each element can have at most two children, the number of elements at level i is at most 2^{i-1}, $i > 0$. For $h = 0$, the total number of elements is 0, which equals $2^0 - 1$. For $h > 0$, the number of elements cannot exceed,

$$\sum_{i=1}^{h} 2^{i-1} = 2^{1-1} + 2^{2-1} + 2^{3-1} + \ldots\ldots + 2^{h-1}$$

$$= 1 + 2 + 2^2 + \ldots\ldots + 2^{h-1}$$

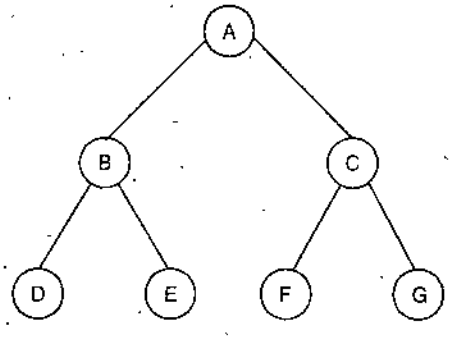
$$= \frac{1 \times (2^h - 1)}{2 - 1} \qquad \left[\because S_n = \frac{a(r^n - 1)}{r - 1}\right]$$

$$= 2^h - 1.$$

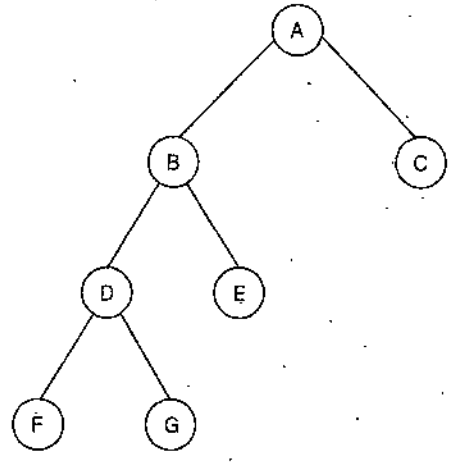
Property 3. *The height of a binary tree that contains n, $n \geq 0$, elements is at most n at least $\lceil \log_2 (n + 1) \rceil$.*

Proof : Since there must be at least one element at each level, the height cannot exceed n. From Property 2, we know that a binary tree of height h can have no more than $2^h - 1$ elements. So $n \leq 2^h - 1$. Hence, $h \geq \log_2 (n + 1)$. Since h is an integer, we have $h \geq \lceil \log_2 (n + 1) \rceil$.

A binary tree of height h that contains exactly $2^h - 1$ elements is called a **full binary tree.** The binary tree of figure 6 (*a*) is a full binary tree of height 3. The binary trees of figures 6 (*b*) and (*c*) are not full binary trees.

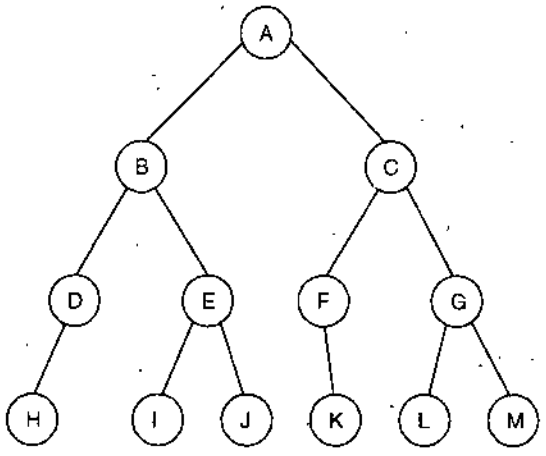


(*a*)



(*b*)

NOTES



(c)

Fig. 6. *Binary Trees.*

Figure 7 shows a full binary tree of height 4.

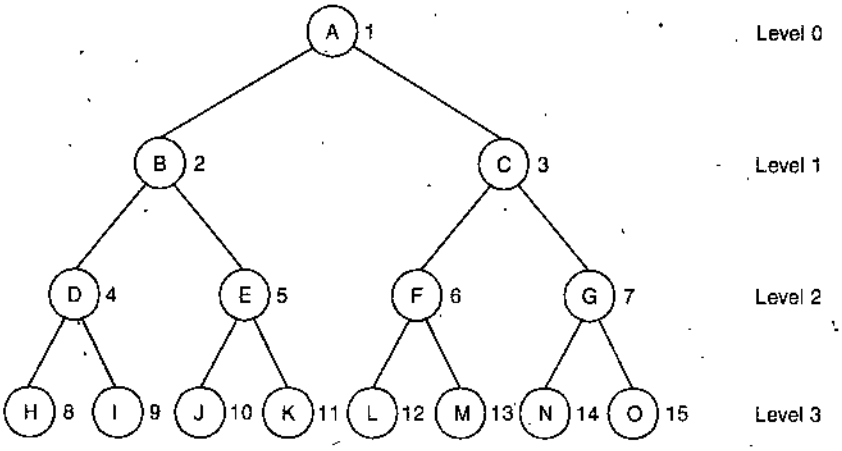


Fig. 7. *Full binary tree of height 4.*

Suppose we assign numbers to the elements of a full binary tree of height h using the numbers 1 through $2^h - 1$. We begin at level 0. Within levels the elements are numbered left to right. The elements of the full binary tree of figure 7 have been numbered in this way. Now suppose we delete the k, $k \geq 0$, elements numbered $2^h - i$, $1 \leq i \leq k$ for any k. The resulting binary tree is called a **complete binary tree.** Figure 8 illustrates some examples.

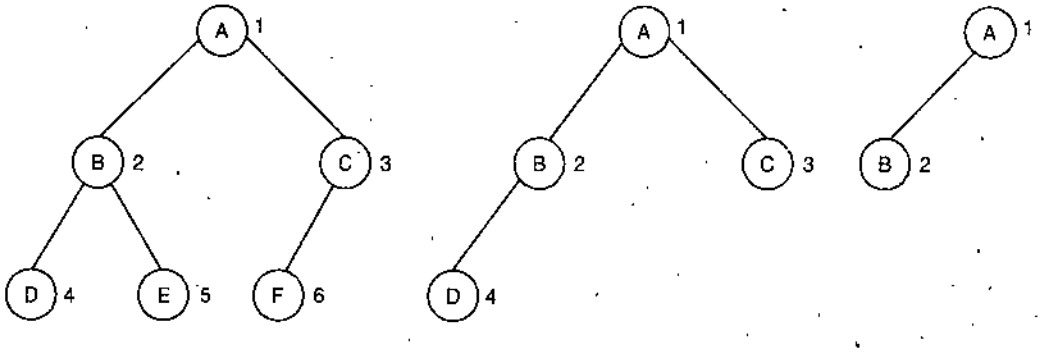


Fig. 8. *Complete binary trees.*

Note that a *full binary tree* is a special case of a *complete binary tree*. Also note that the height of a complete binary tree that contains n elements is $\lceil \log_2 (n + 1) \rceil$.

There exists a nice relationship among the numbers assigned to an element and its children in a complete binary tree, as given by Property 4.

Property 4. *Let i, $1 \leq i \leq n$, be the number assigned to an element of a complete binary tree. The following are true :*

1. *If $i = 1$, then this element is the root of the binary tree. If $i > 1$, then the parent of this element has been assigned the number $\lfloor i/2 \rfloor$*
2. *If $2i > n$, then this element has no left child. Otherwise, its left child has been assigned the number $2i$.*
3. *If $2i + 1 > n$, then this element has no right child. Otherwise, its right child has been assigned the number $2i + 1$.*

Proof : Can be established by induction on i. It is left as an exercise for the readers.

5.5 IMPLEMENTATION OF BINARY TREES

Binary trees can be implemented by using an array or by using pointers.

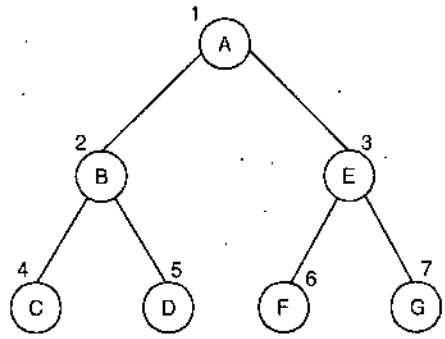
5.5.1 Array Implementation of a Binary Tree

A binary tree can be stored as an array. A two dimensional array with three columns and number of rows equal to number of nodes in the tree can be used to store a binary tree. The first column store contents of data field of each node, second column contains pointer to the left child of the node and the third column contains pointer to the right child of the node. A dash '–' in second or third column represents empty subtree.

Consider the binary tree given in figure 9(a), its array implementation is shown in figure 9 (b).

For convenience, we assign numbers to nodes as they are inserted in the tree.

NOTES



(a)

	DATA	LPTR	RPTR
1	A	2	3
2	B	4	5
3	E	6	7
4	C	—	—
5	D	—	—
6	F	—	—
7	G	—	—

(b)

Fig. 9

5.5.2 Linked Implementation of a Binary Tree

There is a limitation to the array implementation of a tree. The array size is fixed at **compile time.** Therefore, the most suitable implementation is obtained by using pointers which allows the tree to grow or shrink as per requirement during program execution making it a dynamic data structure. It is also called as linked implementation of a tree. In linked representation of the tree, each node has three fields, *i.e.,*

- DATA field or INFO field.
- LPTR field containing a pointer to the left subtree.
- RPTR field containing a pointer to the right subtree.

For example, consider the binary tree shown in figure 10 :

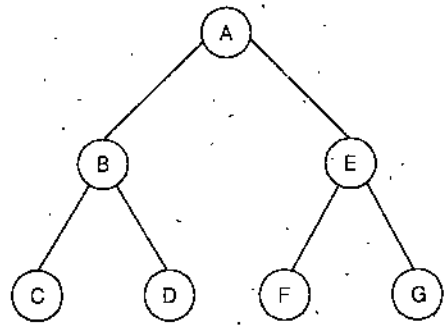


Fig. 10

The binary tree of figure 10 can be shown using linked implementation as illustrated in figure 11.

NOTES

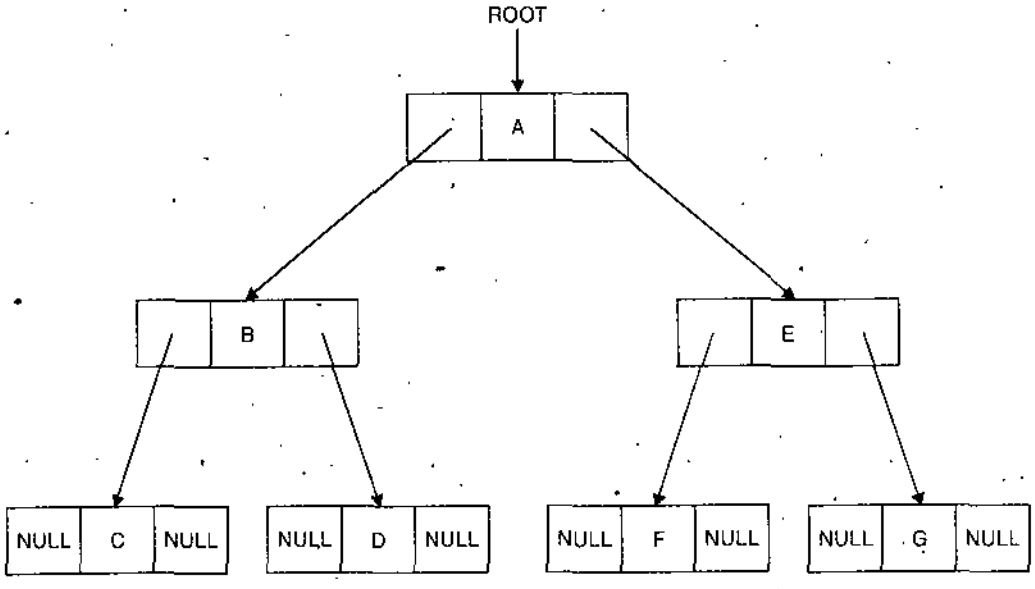


Fig. 11

5.6 BINARY TREE TRAVERSAL METHODS

The traversal of a binary tree means visiting each node in the tree exactly once and performing some operation on it. A full traversal gives a linear order of the information in a tree. As mentioned earlier a binary tree is defined as either an empty tree or consists of a node called **root** and two subtrees, *i.e.,* the **left subtree** and the **right subtree.** Further, left subtree (or right subtree) is again a binary tree which is either empty tree or consists of root and left subtree and right subtree. Thus we see that the definition of tree is recursive. Therefore, traversal of a tree can also be done recursively, using recursive procedure.

There are 6 different ways of traversing a binary tree. These are given below :

(*i*) Visit root, traverse left subtree, traverse right subtree (called **preorder**)

(*ii*) Traverse left subtree, visit root, traverse right subtree (called **inorder**)

(*iii*) Traverse left subtree, traverse right subtree, visit root (called **postorder**)

(*iv*) Visit root, traverse right subtree, traverse left subtree

(*v*) Traverse right subtree, visit root, traverse left subtree

(*vi*) Traverse right subtree, traverse left subtree, visit root.

If convention is adopted then we traverse left before right then only first three traverals remain, *i.e.,* PREORDER, INORDER and POSTORDER. These names have been assigned due to the fact that there is a natural correspondence between these traversals and producing the PREFIX, INFIX and POSTFIX forms of an

NOTES

expression. Here we will consider only first three of these. Suppose we want to print data in each node of a binary tree. It can be done as given below :

5.6.1 Preorder Traversal

To traverse a non-empty binary tree in preorder we perform the following three operations :

(*i*) Visit the root and perform the desired operation

(*ii*) Traverse the left subtree in preorder

(*iii*) Traverse the right subtree in preorder.

For example, consider the binary tree given in figure 12 (*a*) :

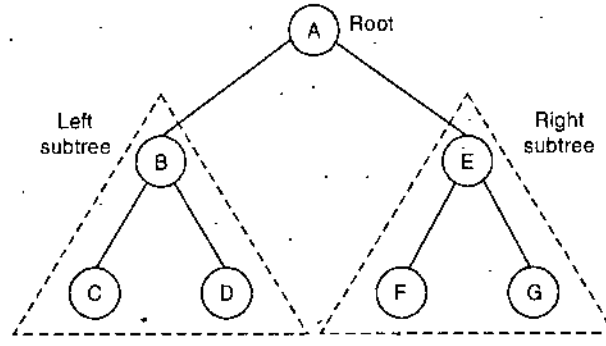


Fig. 12 (*a*)

- We first visit the root, *i.e.,* (A) and print its contents.
- Then we traverse the left subtree of (A) which is shown below :

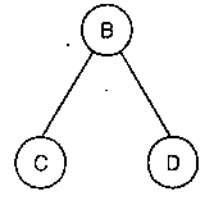


Fig. 12 (*b*)

The root of this tree is (B) so its contents are printed.

- Then we visit left subtree, which is only (C). Its contents are printed. It has no further subtrees.
- Now we visit the right subtree of (B), which is (D) and print its contents. (D) has no further subtrees. Thus, traversal of the left subtree of (A) is over.

- Next, we visit the right subtree of (A) which is shown below :

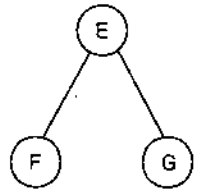


Fig. 12 (c)

The root of this tree is (E) so its contents are printed.

- Then we visit the left subtree of (E) which is only (F). Its contents are printed. It has no further subtrees.
- Now we visit the right subtree of (E), which is (G) and print its contents.

(G) has no further subtrees. Thus, traversal of the right subtree of (A) is over.

Thus, traversal of this complete binary tree gives the following result :

A B C D E F G.

5.6.2 Algorithm for PREORDER Traversal in a Binary Tree

Procedure RPREORDER(T). Given a binary tree whose root node address is given by a pointer variable T and whose node structure is the same as previously described, this algorithm traverses the tree in preorder in a recursive manner.

1. [Process the root node]

   If T ≠ NULL Then

   Write(DATA(T))

   else

   Write('EMPTY TREE')

   Return

2. [Process the left subtree]

   If LPTR(T) ≠ NULL Then

   Call RPREORDER(LPTR(T))

3. [Process the right subtree]

   If RPTR(T) ≠ NULL Then

   Call RPREORDER(RPTR(T))

4. [Finished]

   Return

NOTES

5.6.3 Inorder Traversal

To traverse a non-empty binary tree in inorder we perform the following three operations :

(*i*) Traverse the left subtree in inorder.

(*ii*) Visit the root and perform the desired operation.

(*iii*) Traverse the right subtree in inorder.

For example, consider the binary tree given in figure 12 (*a*). The output of inorder traversal of this tree is given below :

C B D A F E G

5.6.4 Algorithm for INORDER Traversal in a Binary Tree

Procedure RINORDER(T). Given a binary tree whose root node address is given by a pointer variable T and whose node structure is the same as previously described, this algorithm traverses the tree in inorder, again in a recursive manner.

```
1. [Check for empty tree]
       If T = NULL Then
           Write('EMPTY TREE')
           Return
2. [Process the left subtree]
       If LPTR(T) ≠ NULL Then
           Call RINORDER(LPTR(T))
3. [Process the root node]
       Write(DATA(T))
4. [Process the right subtree]
       If RPTR(T) ≠ NULL Then
           Call RINORDER(RPTR(T))
5. [Finished]
       Return
```

5.6.5 Postorder Traversal

To traverse a non-empty binary tree in postorder we perform the following three operations :

(*i*) Traverse the left subtree in postorder

(*ii*) Traverse the right subtree in postorder

(*iii*) [illegible] the root and perform the desired operation.

For example, consider the binary tree given in figure 12 (*a*). The output of postorder traversal of this tree is given below :

C D B F G E A

NOTES

5.6.6 Algorithm for POSTORDER Traversal in a Binary Tree

Procedure RPOSTORDER(T). Given a binary tree whose root node address is given by a pointer variable T and whose node structure is the same as previously described, this procedure traverses the tree in postorder, in a recursive manner.

1. [Check for empty tree]
   If T = NULL Then
   Write('EMPTY TREE')
   Return
2. [Process the left subtree]
   If LPTR(T) ≠ NULL Then
   Call RPOSTORDER(LPTR(T))
3. [Process the right subtree]
   If RPTR(T) ≠ NULL Then
   Call RPOSTORDER(RPTR(T))
4. [Process the root node]
   Write(DATA(T))
5. [Finished]
   Return

Example : *Consider the expression tree implemented as a binary tree in figure 13. Give the output of preorder, inorder and postorder tree traversal of this tree. What do these output represent ?*

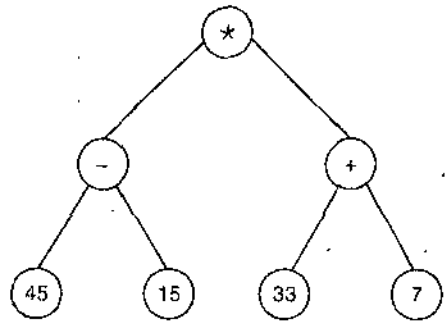


Fig. 13

Solution : Using *preorder traversal* method, we get

*, –, 45, 15, +, 33, 7

Here, commas are written for sake of readability.

It represents the expression in prefix notation.

NOTES

Using *inorder traversal* method, we get,

45, –, 15, *, 33, +, 7

which is an expression in infix notation.

Using *postorder traversal* method, we get,

45, 15, –, 33, 7, +, *

which is an expression in postfix notation.

5.7 BINARY TREE TRAVERSAL ALGORITHMS USING STACKS (*i.e.*, ITERATIVE ALGORITHMS)

We can no fill in the details of the general algorithms given in the previous section for the preorder, inorder, and postorder traversals of a binary tree. These algorithms are written as procedures with one parameter. The only parameter required is a pointer variable which contains the address of the root of the tree. Although recursive algorithms would probably be the simplest to write for the traversals of binary trees, we will formulate algorithms which are both iterative and recursive.

Let us consider the traversal of binary trees by iteration. Since in traversing a tree it is required to descend and subsequently ascend parts of the tree, pointer information which will permit movement up the tree must be temporarily stored. Observe that the structural information that is already present in the tree permits the downward movement from the root of the tree. Because movement up the tree must be made in a reverse manner from that taken in descending a tree, a stack is required to save pointer variables as the tree is traversed.

5.7.1 Preorder Traversal

A general algorithm for a preorder traversal of a binary tree using iteration is now given.

1. If the tree is empty
   then write tree empty and return
   else place the pointer to the root of the tree on the stack
2. Repeat step 3 while the stack is not empty
3. Pop the top pointer off the stack
   Repeat while the pointer value is not null
   Write the data associated with the node
   If right subtree is not empty Then

Stack the pointer to the right subtree

Set pointer value to left subtree

We will now provide a procedure for traversing a tree in preorder.

Procedure PREORDER(T). Given a binary tree whose root node address is given by a pointer variable T and whose structure is the same as previously described, this procedure traverses the tree in preorder, in an iterative manner. S and TOP denote an auxiliary stack and its associated top index, respectively. The pointer variable P denotes the current node in the tree.

NOTES

1. [Initialize]

    If T = NULL Then

        Write('EMPTY TREE')

        Return

    Else

        TOP ← 0

        Call PUSH(S, TOP, T)

2. [Process each stacked branch address]

    Repeat step 3 while (TOP > 0)

3. [Get stored address and branch left]

    P ← POP(S, TOP)

    Repeat while P ≠ NULL

        Write(DATA(P))

        If RPTR(P) ≠ NULL Then

            Call PUSH(S, TOP, RPTR(P)) (store address of nonempty right subtree)

        P ← LPTR(P) (branch left)

4. [Finished]

    Return

Step 1 checks for an empty tree and exits if T = NULL. Otherwise, it stacks the address of the root node. Step 2 controls the processing of the tree. The addresses of yet untraversed subtrees are kept on the stack. In the third step of the algorithm, we visit and process a node. The address of the right branch of such a node, if it exists, is stacked and a chain of left branches is followed until this chain ends. At this point, we reenter step 3 and delete from the stack the address of the root node of the most recently encountered right subtree and process it according to step 3. A trace of the algorithm for the binary tree given in figure 14 appears in Table 1, where the right-most element in the stack is considered to be its top element and the notation "NE", for example, denotes the address of node E. The visit of a node in this case merely involves the output of the label for that node.

NOTES

An equivalent procedure for a recursive preorder traversal of a binary tree is easily formulated.

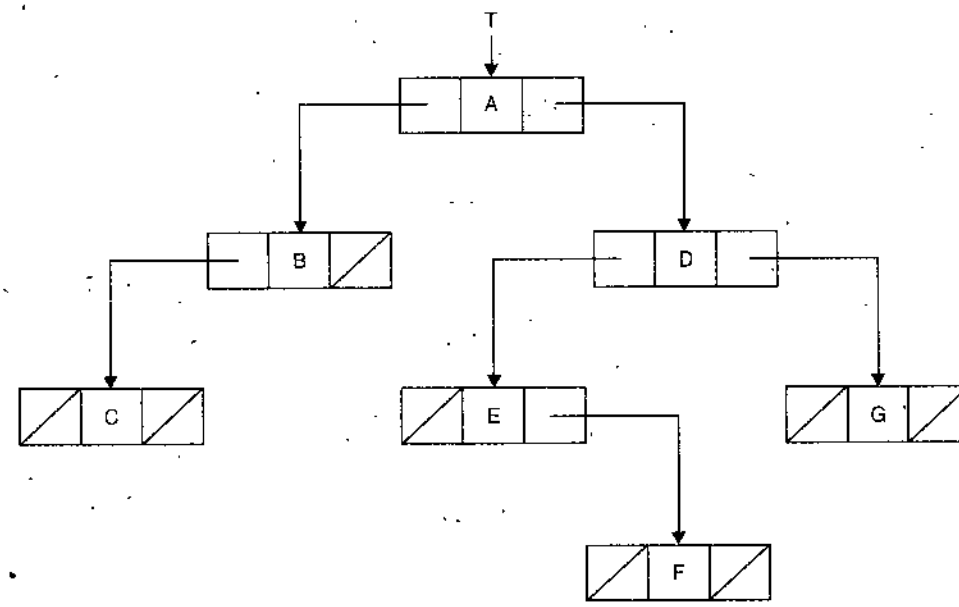


Fig. 14 *Linked representation of a binary tree.*

Table 1. Trace of Procedure PREORDER for figure 14

Stack Contents	*P*	*Visit P*	*Output String*
NA			
	NA	A	A
ND	NB	B	AB
ND	NC	C	ABC
ND	NULL		
	ND	D	ABCD
NG	NE	E	ABCDE
NG NF	NULL		
NG	NF	F	ABCDEF
NG	NULL		
	NG	G	ABCDEFG
	NULL		

5.7.2 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, which will hold the addresses of nodes for future processing. In fact, with this algorithm, a node is processed only when it is popped from STACK.

NOTES

Algorithm : Initially push NULL onto STACK (for a sentinel) and then set PTR = ROOT. Then repeat the following steps until NULL is popped from STACK.

(*a*) Proceed down the left-most path rooted at PTR, pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

(*b*) [Backtracking.] Pop and process the nodes on STACK. If NULL is popped, then Exit. If a node N with a right child R(N) is processed, set PTR = R(N) (by assigning PTR = RIGHT[PTR]) and return to Step (*a*).

We emphasize that a node N is processed only when it is popped from STACK.

Example : *Consider the binary tree T in figure 15. We simulate the above algorithm with T, showing the contents of STACK.*

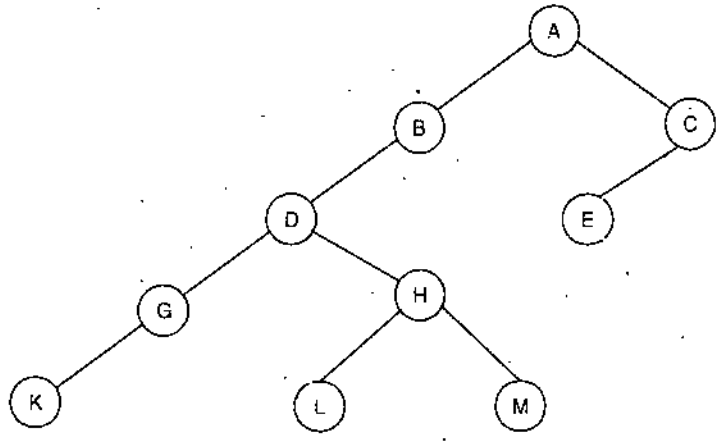


Fig. 15

1. Initially push NULL onto STACK :

   STACK : NULL.

   Then set PTR = A, the root of T.

2. Proceed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G and K onto STACK :

   STACK : NULL, A, B, D, G, K.

   (No other node is pushed onto STACK, since K has no left child.)

3. [Backtracking.] The nodes K, G and D are popped and processed, leaving :

   STACK : NULL, A, B.

   (We stop the processing at D, since D has a right child.) Then set PTR = H, the right child of D.

4. Proceed down the left-most path rooted at PTR = H, pushing the nodes H and L onto STACK :

   STACK : NULL, A, B, H, L.

   (No other node is pushed onto STACK, since L has no left child.)

5. [Backtracking.] The nodes L and H are popped and processed, leaving :

   STACK : NULL, A, B.

   (We stop the processing at H, since H has a right child.) Then set PTR = M, the right child of H.

6. Proceed down the left-most path rooted at PTR = M, pushing node M onto STACK :

   STACK : NULL, A, B, M.

   (No other node is pushed onto STACK, since M has no left child.)

7. [Backtracking.] The nodes M, B and A are popped and processed, leaving :

   STACK : NULL.

   (No other element of STACK is popped, since A does have a right child.) Set PTR = C, the right child of A.

8. Proceed down the left-most path rooted at PTR = C, pushing the nodes C and E onto STACK :

   STACK : NULL, C, E.

9. [Backtracking.] Node E is popped and processed. Since E has no right child, node C is popped and processed. Since C has no right child, the next element, NULL, is popped from STACK.

The algorithm is now finished, since NULL is popped from STACK. As seen from Steps 3, 5, 7 and 9, the nodes are processed in the order K, G, D, L, H, M, B, A, E, C. This is the required inorder traversal of the binary tree T.

A formal presentation of our inorder traversal algorithm is given below :

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

A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the address of nodes.

1. [Push NULL onto STACK and initialize PTR.]

   Set TOP = 1, STACK[1] = NULL and PTR = ROOT.

2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]

   (*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 : [Backtracking.]

5. Apply PROCESS to INFO[PTR].

6. [Right child?]

If RIGHT[PTR] ≠ NULL, Then

(*a*) Set PTR = RIGHT[PTR].

(*b*) Go to Step 2.

[End of If structure.]

7. Set PTR = STACK[TOP] and TOP = TOP – 1. [Pops node.]

[End of Step 4 loop.]

8. Exit.

5.7.3 Postorder Traversal

A general algorithm for an iterative postorder traversal of a binary tree is now presented.

1. If the tree is empty Then

   Write empty tree and return

   Else

   Initialize the stack and initialize pointer value to root of tree

2. Start an infinite loop to repeat upto step 5

3. Repeat while pointer value is not null

   Stack current pointer value

   Set pointer value to left subtree

4. Repeat while top pointer on stack is negative

   Pop pointer off stack

   Write data associated with positive value of this pointer

   If stack is empty Then

   return

5. Set pointer value to the right subtree of the value on top of the stack

   Stack the negative value of the pointer to the right subtree

The following algorithm iteratively traverses a binary tree in postorder :

Procedure POSTORDER(T). The same node structure described previously is assumed, and T is a variable which contains the address of the root of the tree. A stack S is also required again, but this time each node will be stacked twice, once when its left subtree is traversed and once when its right subtree is traversed. On completion of these two traversals, the particular node is processed. Consequently, we need two types of stack entries, the first indicating that a left subtree is being traversed, and the second that a right subtree is being traversed. For convenience we will use negative pointer values to indicate the second type of entry. This, of course, assumes that valid pointer data are always nonzero and positive.

NOTES

```
1. [Initialize]
        If T = NULL Then
            Write('EMPTY TREE')
            Return
        Else
            P ← T
            TOP ← 0
2. [Traverse in postorder]
        Repeat upto step 5 while true
3. [Descend left]
        Repeat while P ≠ NULL
            Call PUSH(S, TOP, P)
            P ← LPTR(P)
4. [Process a node whose left and right subtrees have been traversed]
        Repeat while S[TOP] < 0
            P ← POP(S, TOP)
            Write(DATA(P))
            If TOP = 0 Then (Have all nodes been processed?)
            Return
5. [Branch right and then mark node from which we branched]
        P ← RPTR(S[TOP])
        S[TOP] ← – S[TOP]
```

The first step checks for an empty tree. In step 2 an infinite loop is initiated to ensure that the entire tree is processed. In the third step, a chain of left branches is followed and the address of each node encountered is stacked. Step 4 prints out the data associated with those nodes whose right and left subtrees have been traversed, indicated by a negative pointer value. In step 5, the right subtree of the node on top of the stack is placed in P to be traversed in the next iteration of the loop. The address of this node is negated, indicating that both left and right subtrees have been traversed and that its data may be printed.

Example 1. *For a binary tree T, the preorder and inorder travel sequences are as given below :*

Preorder : A, B, C, D, E, F, G, H, I

Inorder :D, C, B, A, G, F, H, I, E

Construct the binary tree T.

Solution : The binary tree T is constructed as given below :

The first node in the preorder traversal is always the root node of the binary tree. Thus the root node of the binary tree T is node A.

NOTES

After finding the root node of the binary tree, our goal is to find the nodes that form the left subtree and the right subtree of node A.

We know that, inorder traversal is traverse the left subtree, visit the root and finally traverse the right subtree. Thus all the nodes to the left of node A in inorder traversal form its left subtree and all the nodes to the right of A in inorder traversal form its right subtree.

From the above discussion we have :

Preorder	(A)	B, C, D	E, F, G, H, I
	Root	Left subtree L_{TA}	Right subtree R_{TA}
Inorder	D, C, B	(A)	G, F, H, I, E
	Left subtree L_{TA}	Root	Right subtree R_{TA}

Figure 16 shows the partial binary tree formed so far :

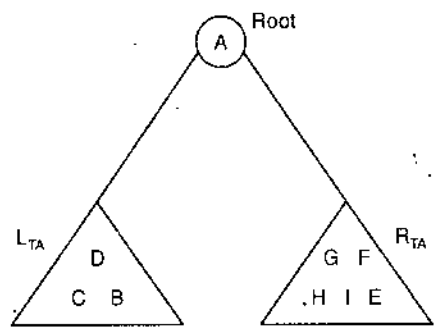


Fig. 16

Similarly, we can form the left subtree L_{TA} and right subtree R_{TA}.

Let us first take the left subtree L_{TA}. The preorder and inorder sequences of the left subtree L_{TA} are given below :

Preorder B, C, D

Inorder D, C, B

From these traversal sequences we see that the root of the subtree is node B and its left subtree consists of nodes D and C. So it has no right subtree.

From the above discussion we have :

Preorder	(B)	C, D	
	Root	Left subtree L_{TB}	Right subtree R_{TB}
Inorder	D, C	(B)	
	Left subtree L_{TB}	Root	Right subtree R_{TB}

NOTES

Figure 17 shows the partial binary tree formed so far :

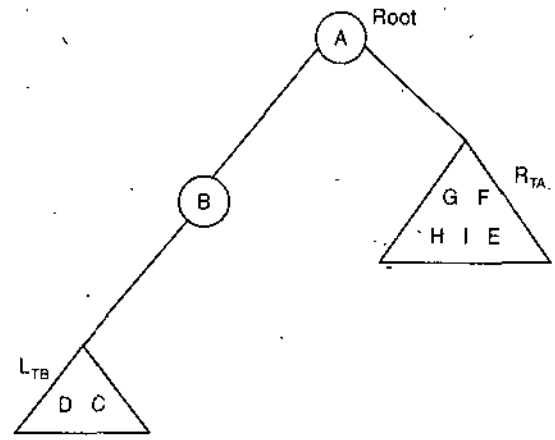


Fig. 17

Now let us take the left subtree L_{TB}. The preorder and inorder sequences of the left subtree L_{TB} are given below :

Preorder C, D

Inorder D, C

From these traversal sequences we see that the root of the subtree is node C and its left subtree consists of node D. So it has no right subtree.

From the above discussion we have :

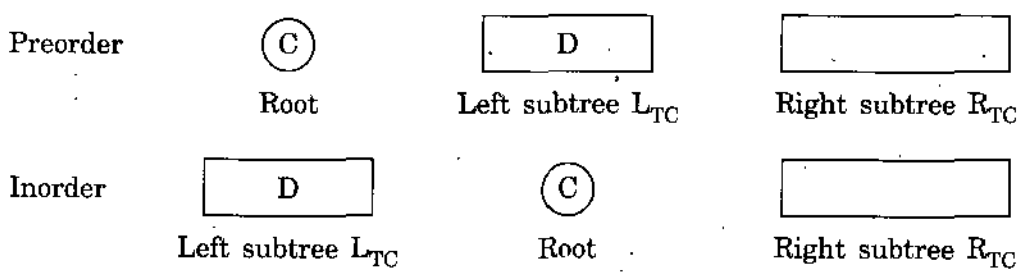


Figure 18 shows the partial binary tree formed so far :

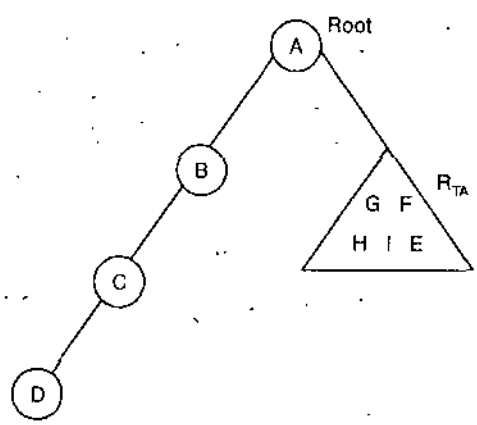


Fig. 18

Thus we have formed the left subtree L_{TA} of node A, which is the root of the binary tree.

Now let us take the right subtree R_{TA}. The preorder and inorder traversal sequences of R_{TA} are given below :

Preorder E, F, G, H, I

Inorder G, F, H, I, E

From these traversal sequences we see that the root of the subtree R_{TA} is node E, its left subtree consists of nodes F, G, H, I and its right subtree is empty.

From the above discussion we have :

Preorder	(E)	F, G, H, I	
	Root	Left subtree L_{TE}	Right subtree R_{TE}
Inorder	G, F, H, I	(E)	
	Left subtree L_{TE}	Root	Right subtree R_{TE}

Figure 19 shows the partial binary tree formed so far :

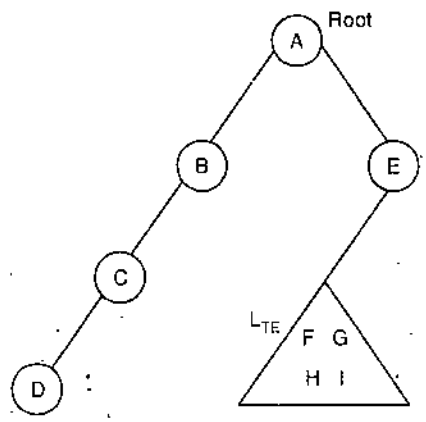


Fig. 19

Now let us take the left subtree L_{TE}. The preorder and inorder traversal sequences of L_{TE} are given below :

Preorder F, G, H, I

Inorder G, F, H, I

From these traversal sequences we see that the root of the subtree L_{TE} is node F, its left subtree consists of node G and right subtree has nodes H, I.

From the above discussion we have,

Preorder	(F)	G	H, I
	Root	Left subtree L_{TF}	Right subtree R_{TF}

NOTES

NOTES

Inorder | G | F | H, I

Left subtree L_{TF} Root Right subtree R_{TF}

Figure 20 shows the partial binary tree formed so far :

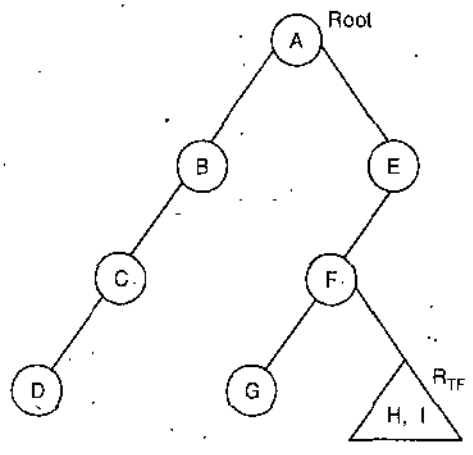


Fig. 20

Now let us take the right subtree R_{TF}. The preorder and inorder travel sequences of R_{TF} are given below :

Preorder H, I

Inorder H, I

From these traversal sequences we see that the root of the subtree R_{TF} is node H, its left subtree is empty and the right subtree has a single node I.

From the above discussion we have

Preorder | H | | I

Root Left subtree L_{TH} Right subtree R_{TH}

Inorder | | H | I

Left subtree L_{TH} Root Right subtree R_{TH}

Figure 21 shows the binary tree after the above step and this is the required binary tree :

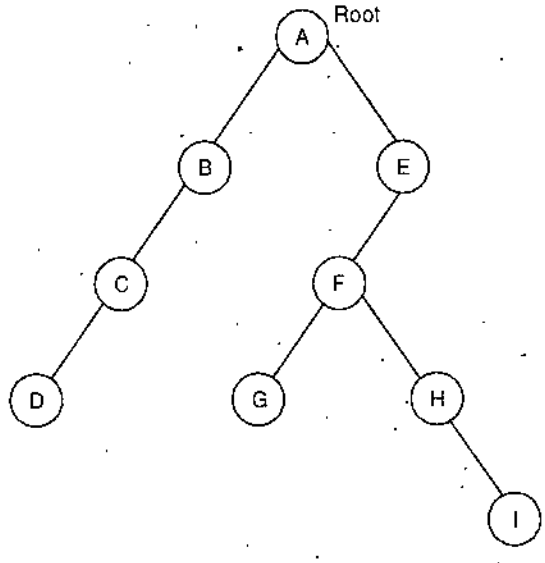


Fig. 21

5.8 BINARY SEARCH TREE

A special kind of binary tree is called a **binary search tree**. In a binary search tree, the data value stored in any node is greater than the data value stored in its left child node and less than the data value stored in its right child node, assuming that there are no duplicate values.

The formal definition of a *binary search tree* is given below :

A **binary search tree** is a binary tree that may be empty. A nonempty binary search tree satisfies the following properties :

1. Every element has a key (or value) and no two elements have the same key; therefore, all keys are distinct.
2. The keys (if any) in the left subtree of the root are smaller than the key in the root.
3. The keys (if any) in the right subtree of the root are larger than the key in the root.
4. The left and right subtrees of the root are also binary search trees.

There is some redundancy in this definition. Properties 2,3 and 4 together imply that the keys must be distinct. Therefore, property 1 can be replaced by the following property :

The root has a key.

NOTES

For example, the tree shown in figure 22 is a binary search tree :

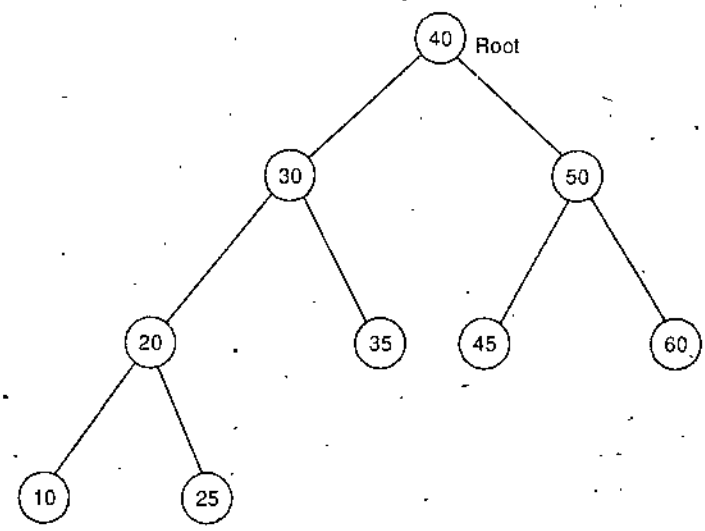


Fig. 22. *A Binary Search Tree*

Binary search tree is a very useful data structure. In system software such as loaders, assemblers and compilers, we generally need to build symbol tables of key words or reserved words. These tables are very often searched for a specific key word. In that case, if symbol table is implemented by a binary search tree then the number of comparisons for searching a specific data value can be reduced. The reason for this is that we can tell in which half (left or right) of the tree, the data value may lie with only one comparison.

For example, if we are searching for a right data value 45, we compare 45 with 40 and we find that 45 > 40 and therefore 45 must lie in the right subtree. Then, we compare 45 with 50. Since 45 < 50, therefore 45 must lie in the left subtree. We go to left side and compare it with the node and find it is equal to 45. Thus, we need 3 comparisons to locate the data value 45.

NOTE *Binary search tree is an application of a binary tree.*

We can remove the requirement that all elements in a binary search tree are distinct. Now we replace smaller in property 2 by $\leq$ and large in property 3 by $\geq$; the resulting tree is called a **binary search tree with duplicates**.

An **indexed binary search tree** is derived from an ordinary binary search tree by adding the field *Left size* to each tree node. This field gives the number of elements in the node's left subtree plus one. Figure 23 shows two indexed binary search trees.

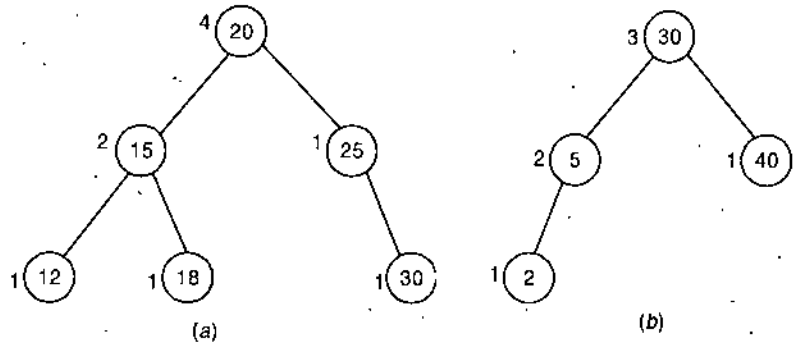


Fig. 23. *Indexed Binary Search Trees*

The number inside a node is the element key, while that outside is the value of *Left size*. Notice that *Leftsize* also gives the rank of an element with respect to the elements in its subtree. For example, in the tree of figure 23 (*a*), the elements (in sorted order) in the subtree with root 20 are 12, 15, 18, 20, 25 and 30. The rank of the root is four (*i.e.*, the element in the root is the fourth element in sorted order). In the subtree with root 25, the element (in sorted order) are 25 and 30, so the rank of 25 is one and its *Left size* value is 1.

5.8.1 Searching and Insertion in a Binary Search Tree

The formal presentation of our search and insertion algorithm will use the following procedure, which finds the locations 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. This procedure will also be used in the next section, on deletion.

Procedure 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 three 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. [Tree empty?]

   If ROOT = NULL, then: Set LOC = NULL and PAR = NULL, and Return.

2. [ITEM at root?]

   If ITEM = INFO[ROOT], then: Set LOC = ROOT and

   PAB = NULL, and Return.

NOTES

3. [Initialize pointers PTR and SAVE.]
   If ITEM < INFO[ROOT], then:
      Set PTR = LEFT[ROOT] and SAVE = ROOT.
   Else:
      Set PTR = RIGHT[ROOT] and SAVE = ROOT.
   [End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. [ITEM found?]
   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].
   [End of If structure.]
   [End of Step 4 loop.]
7. [Search unsuccessful.] 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].

The formal statement of our search and insertion algorithm follows.

Algorithm: INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

A binary search tree T is in memory and an ITEM of information is given. This 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 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.
   (b) Set NEW = AVAIL, AVAIL = LEFT[AVAIL] and INFO[NEW] = ITEM.
   (c) 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.

NOTES

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.

5.8.2 Deleting in a Binary Search Tree

Suppose T is a binary search tree, and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T.

The deletion algorithm first uses searching procedure to find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases :

Case 1. N has no children. Then N is deleted from T by simply 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 simply replacing the location of Nin P(N) by the location of the only child of N.

Case 3. N has two children. Let S(N) denote inorder successor of N. (The reader can verify that S(N) does not have a left child). Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Observe that the third case is much more complicated than the first two cases. In all three cases, the memory space of the deleted node N is returned to the AVAIL list.

Our deletion algorithm will be stated in terms of Procedures given below. The first procedure refers to Cases 1 and 2, where the deleted node N does not have two children; and the second procedure refers to Case 3, where N does have two children. There are many subcases which reflect the fact that N may be a left child, a right child or the root. Also, in Case 2, N may have a left child or a right child.

Second procedure treats the case that the deleted node N has two children. We note that the inorder successor of N can be found by moving to the right child of N and then moving repeatedly to the left until meeting a node with an empty left subtree.

NOTES

Procedure 2 : CASEA(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.
3. Return.

Procedure 3 : CASEB(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.]
   (*c*) Set SUC = PTR and PARSUC = SAVE.

2. [Delete inorder successor, using Procedure 2.]
   Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC.)
3. [Replace node N by its inorder successor.]
   (a) If PAR ≠ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] = SUC.
       Else:
           Set RIGHT[PAR] = SUC.
       [End of If structure.]
   Else:
       Set ROOT = SUC.
   [End of If structure.]
   (b) Set LEFT[SUC] = LEFT[LOC] and RIGHT[SUC] = RIGHT[LOC].
4. Return.

NOTES

We can now formally state our deletion algorithm using above procedures as building blocks.

Algorithm : DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

1. [Find the locations of ITEM and its parent, using searching Procedure.]
   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 CASEA(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.

NOTES

Example : *Make a binary search tree of values 80, 40, 150, 100 and 30.*

Solution : The binary search tree is constructed as follows :

1. First set the ROOT of the binary tree as NULL, *i.e.*, make an empty binary tree.

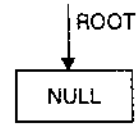


Fig. 24 (*a*)

2. The first value to be stored is 80. We now search the tree and find, it is empty, so a new node is allocated and 80 is stored in this node. Now it becomes the first node and also the root node as shown below :

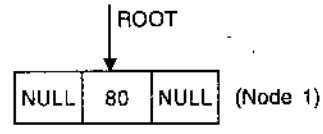


Fig. 24 (*b*)

3. The second value to be inserted is 40, the tree is searched and it is found that 40 must be inserted in left child and the tree becomes as follows :

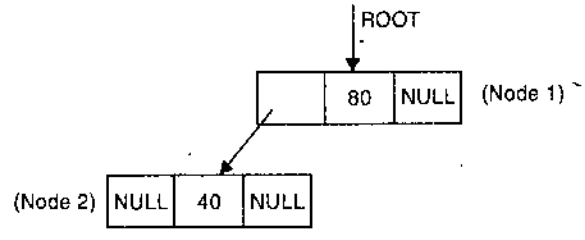


Fig. 24 (*c*)

4. The next value to be inserted is 150. Again the tree is searched and it is known that the number 150 must be inserted as a right child of Node 1. After insertion of 150 the tree becomes :

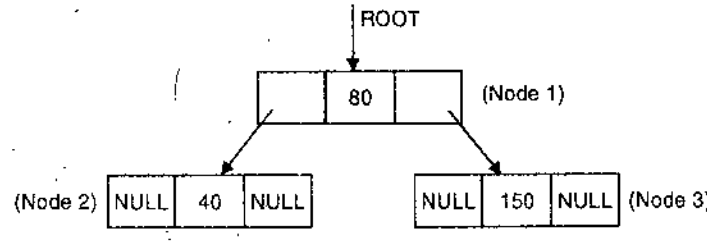


Fig. 24 (*d*)

5. The next value to be inserted is 100. Which will be inserted as shown below after searching its appropriate place :

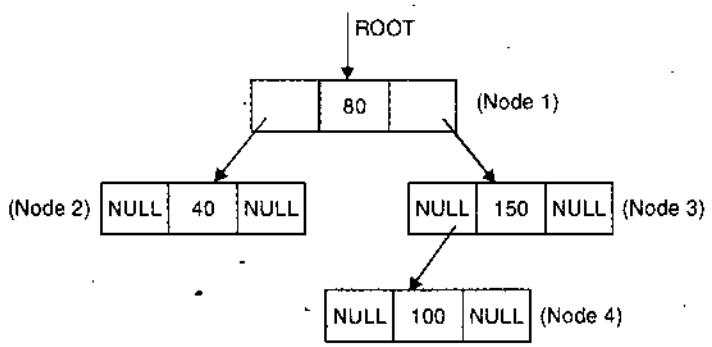


Fig. 24 (*e*)

6. The next value to be inserted is 30, which will be inserted as shown below after searching its appropriate place :

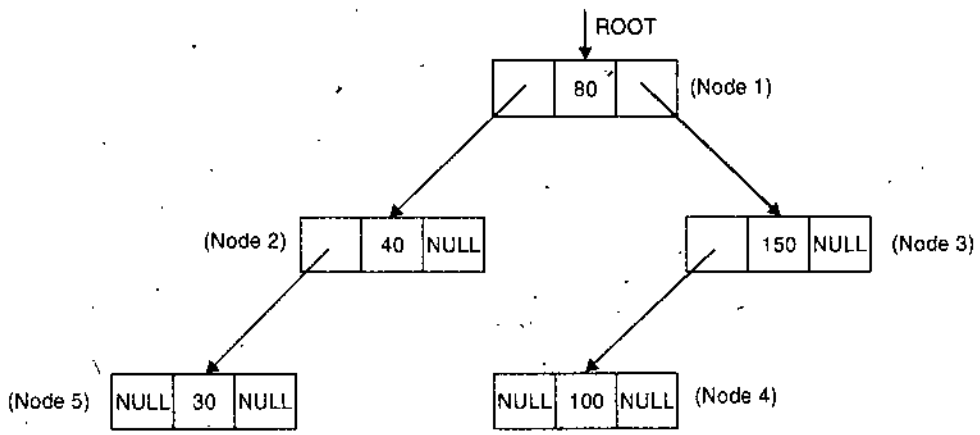


Fig. 24 (*f*)

5.9 SUMMARY

- A tree is often used to represent a *Hierarchy*.
- Tree is a Non-Linear data structure.
- A Tree is a data structure used to represent data containing a hierarchical relation between its elements.
- Tree can be used to represent the Unix file system in which files and subdirectories are stored under directories. Another example is to represent the records in a file in which elementary items are stored under group items.

NOTES

- There are various types of trees such as Unbalanced Binary Tree and Balanced Binary Tree.
- A Binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left and right subtrees.
- Traversal of a tree is to visit each node exactly once, for example searching the particular nodes. Let T be a binary tree, there are different ways to proceed, and the methods differ primarily in the order in which they visit the nodes.
- The tree traversal methods are *preorder, inorder* and *postorder.*
- Many algorithms that use binary trees proceed in two phases. The first phase builds a binary tree and the second traverses the tree.

5.10 TEST YOURSELF

Answer the following questions :

1. For a binary tree T, the in-order and post-order traversal sequences are as follows :

   In-order : D B F E A G C L J H K

   Post-order : D F E B G L J K H C A

   Draw the binary tree T.

2. For a binary tree T, the pre-order and in-order traversal sequences are as follows :

   Pre-order : G B Q A C K F P D E R H

   In-order : Q B K C F A G P E D H R

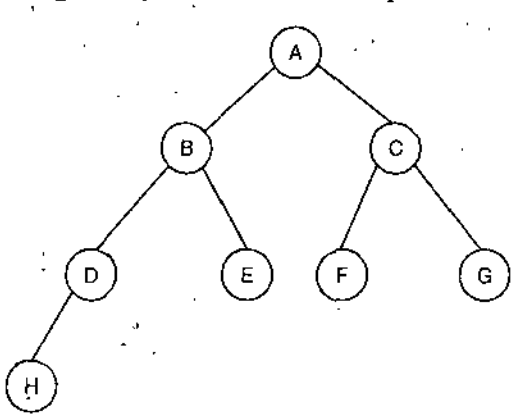
   (*a*) What is the height of the tree ?

   (*b*) What are the internal nodes ?

   (*c*) What is its post-order traversal sequence ?

3. What is the maximum number of nodes at *k*th level of a binary tree ?

4. Given the following binary tree. Answer the questions in the content of it.



NOTES

(*i*) What is the root of this tree ?

(*ii*) What is the parent of D ?

(*iii*) What are the children of C ?

(*iv*) What are the children of A ?

(*v*) What are the siblings of E ?

(*vi*) What are the descendents of B ?

(*vii*) What are ancestors of H ?

(*viii*) What is the level of D ?

(*ix*) List the leaf nodes of the tree.

(*x*) List all integral nodes.

(*xi*) Draw subtrees of A and B.

(*xii*) Draw left subtrees of A and B respectively.

(*xiii*) Draw right subtrees of A and C.

(*xiv*) Give the output of preorder tree traversal of the above tree.

(*xv*) Give the output of in-order tree traversal of the above tree.

(*xvi*) Give the output of postorder tree traversal of the above tree.

5. For a binary tree T, the pre-order and in-order traversal sequences are as follows :

Pre-order : A B D E G H C F

In-order : D B G E H A C F

NOTES

SECTION D

CHAPTER 6 SEARCHING AND SORTING

★ LEARNING OBJECTIVES ★

6.1 Introduction
6.2 Searching
6.3 Sorting
6.4 Summary of Sorting Methods
6.5 Summary
6.6 Test Yourself

6.1 INTRODUCTION

Data stored in an organized manner requires to be accessed for processing. Locating a particular data item in the memory involves searching the data item. Searching is a technique where the memory is scanned for the required data. Computer systems are often used to store large amounts of data from which an individual element or record must be retrieved according to some search specification. Thus the efficient storage of data to facilitate fast searching is an important issue.

Data can be represented in various formats with reference to the data structures they are held in. Accessing data involves time and memory. Unorganized data takes longer time to be accessed compared to ordered data. Data can be ordered in various ways. Sorting is one of the methods of ordering data which is done based on various techniques.

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing with numerical data or alphabetically, with character data.

This chapter deals with different searching techniques to find the required data and an investigation regarding the performance of some searching algorithms and the data structures they use.

This chapter also deals with various sorting techniques and their algorithms.

6.2 SEARCHING

NOTES

Searching in a one-dimensional array can be done using any one of the two methods :

(*i*) *Linear Search* (*ii*) *Binary Search.*

(*i*) Linear Search

In this method, each element of the array is compared with the element to be searched one by one. The searching ends on getting the first occurrence of the element or when the entire array has been traversed. When the element is not found in the array we say that the search is unsuccessful. Linear search can be applied in any one of the two ways :

(*a*) Linear Search in an unsorted array.

(*b*) Linear Search in a sorted array (say array given in ascending order).

(a) Linear Search in an unsorted array

Consider the array given in figure 1 :

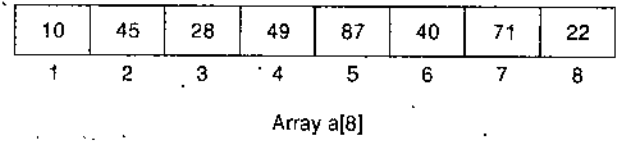


Fig. 1

In case we want to search 71, it is found at location number (position) 7. Search will be unsuccessful for element 50.

6.2.1 Algorithm for Linear Search in an Unsorted Array

Let A be an array of size N. We are to search for the element DATA. I denotes the array index. Assuming lower bound starts with 1.

1. Repeat for I = 1, 2, ..., N upto step 2
2. If (A[I] = DATA) Then

```
{
  Write("Successful search")
  Write(DATA, " found at position ",I)
  goto step 4
}
```

3. Write("Unsuccessful search")
4. End.

On an average, linear search requires N/2 comparisons. In the worst case, N comparisons are required.

NOTES

The following function in 'C' illustrates the above concept :

```
/* function definition linear_search() */

int linear_search(float a[],int n,float data)
  {
    int i; /* local variable */
    i=0;
    while(i<n)
      {
        if(a[i]==data)
          return(i);
        i++;
      }
    return(-1); /* when entire array has been exhausted */
}
```

(b) *Linear Search in a sorted array (given in ascending order)*

Consider the array given in figure 2 :

1	2	5	7	10	45	69	94
1	2	3	4	5	6	7	8

Array a[8]

Fig. 2

In case we want to search 10, it is found at location number (position) 5. Search will be unsuccessful for element 4 and we terminate search on reaching the element 5 as remaining elements are bigger than the element to be searched, *i.e.*, 4.

6.2.2 Algorithm for Linear Search in a Sorted Array (Ascending Order)

Let A be a sorted array of size N (having elements in ascending order). We are to search for the element DATA. I denotes array index. The non-existence of the element in the array can be declared without searching the entire array. Assuming lower bound starts with 1.

1. Repeat for I = 1, 2, ..., N upto step 2
2. If (A[I] = DATA) Then

```
{
 Write("Successful search")
 Write(DATA," found at position ",I)
 goto step 4
}
Else
```

```
  {
    If(A[I]>DATA) Then
    goto step 3
  }
```

3. Write("Unsuccessful search")

4. End

The following function in C illustrates the above concept :

```
/* function definition linear_search() */

int linear_search(float a[],int n,float data)
  {
    int i=0; /* local variable */
    /* searching */
    while(i<n)
      {
        if(a[i]== data) /* when data is found */
          return(i);
        if(a[i]>data) /* when array element bigger than data is found */
          break;
        i++;
      }
    return(-1); /* when element is not found */
}
```

Linear search is quite time consuming, if the element to be searched lies near the last element, as many comparisons may be required. Binary search saves a lot of time and lesser number of comparisons may be needed but the most important condition for applying it is that the array elements must be in sorted order (either ascending or descending order).

(*ii*) Binary Search

Binary search method requires much less number of comparisons than linear search. It can be used only for sorted arrays.

To search an element say DATA the approximate middle entry of the array is located, and its value is checked. If its value is greater than DATA, the value of the middle element of the first half is located and compared with DATA and the procedure is repeated on the first half until the required value is found or the search interval becomes empty. If the value in the middle position is smaller than DATA, the value of the middle element of the second half is compared with DATA and the procedure is repeated on the second half until the required value is found or the search interval becomes empty.

NOTES

NOTES

As the length of the array to be searched is reduced by half at each step and the array is divided into two equal parts, this method is known as binary search.

The efficiency of this method can be imagined by the fact that only twenty steps will be required in searching a value (element) in an array having more than million keys (values).

6.2.3 Algorithm Binary Search on Array given in Ascending Order

Let A be an array of size N having elements in ascending order. Let DATA be the element to be searched. LOW, HIGH and MID denote the lowest, highest and middle positions of a search interval. Search becomes unsuccessful when LOW > HIGH. Assuming the index begins at 1. The algorithm is defined nonrecursively.

1. LOW = 1
   HIGH = N
2. Repeat while (LOW ≤ HIGH) upto step 4
3. MID = Integral part of ((LOW + HIGH)/2)
4. If (A[MID] = DATA) Then

```
{
  Write("Successful search")
  Write(DATA," found at position ",MID)
  goto step 6
}
Else
{
  If(DATA>A[MID]) Then
    LOW=MID+1
  Else
    HIGH=MID-1
}
```

5. Write ("Unsuccessful search")
6. End.

Let us apply the above algorithm to an example. Suppose array **A** contains elements

5, 17, 30, 52, 66, 70, 83, 91 (N = 8)

and we wish to search for 66.

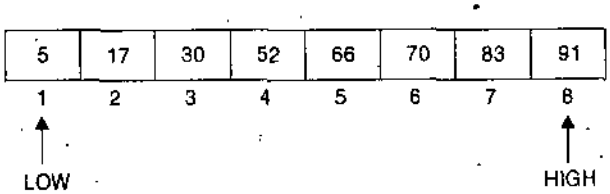


Fig. 3 (a)

LOW = 1 and HIGH = N = 8

Is (LOW ≤ HIGH) ? Yes

MID = Integral part of ((1 + 8)/2) = 4

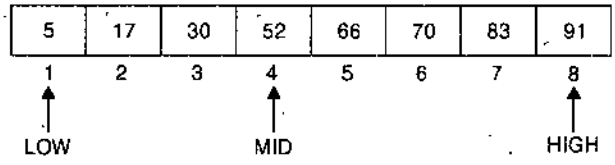


Fig. 3 (*b*)

Is (A[4] = 66) ? No

66 > A[4], repeat the steps with LOW = MID + 1 = 4 + 1 = 5 and HIGH = 8

Is (LOW ≤ HIGH) ? Yes

MID = Integral part of ((5 + 8)/2) = 6

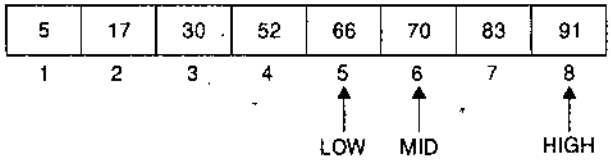


Fig. 3 (*c*)

Is (A[6] = 66) ? No

66 < A[6], repeat the steps with LOW = 5 and HIGH = MID – 1 = 6 – 1 = 5

Is (LOW ≤ HIGH) Yes

MID = Integral part of ((5 + 5)/2) = 5

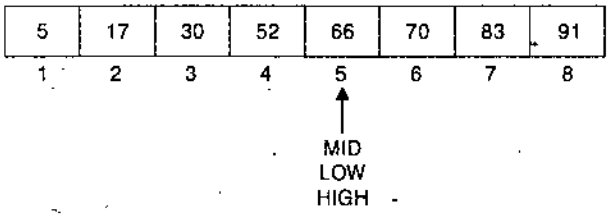


Fig. 3 (*d*)

Is (A[5] = 66) ? Yes

Write(66," found at position ", 5)

The above algorithm when applied on the array A for searching an element not present in the array works as given below :

Let the array elements be 2, 6, 7, 8, 9 and we wish to search for 4.

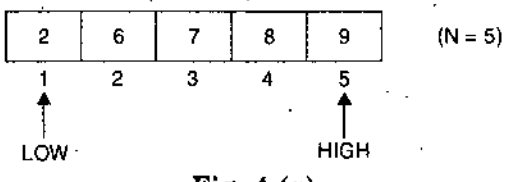


Fig. 4 (*a*)

NOTES

NOTES

LOW = 1 and HIGH = N = 5

Is (LOW ≤ HIGH) ? Yes

MID = Integral part of ((1 + 5)/2) = 3

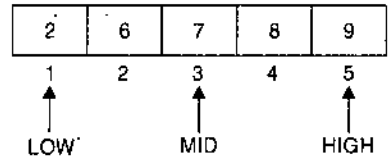


Fig. 4 (*b*)

Is (A[3] = 4) ? No

4 < A[3], repeat the steps with LOW = 1 and HIGH = MID – 1 = 3 – 1 = 2

Is (LOW ≤ HIGH) ? Yes

MID = Integral part of ((1 + 2)/2) = 1

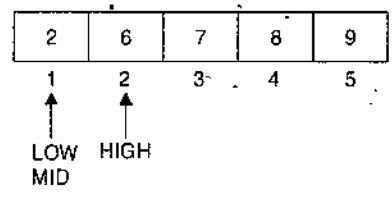


Fig. 4 (*c*)

Is (A[1] = 4) ? No

4 > A[1], repeat the steps with LOW = MID + 1 = 1 + 1 = 2 and HIGH = 2

Is (LOW ≤ HIGH) ? Yes

MID = Integral part of ((2 + 2)/2) = 2

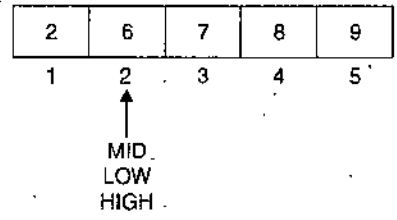


Fig. 4 (*d*)

Is (A[2] = 4) ? No

4 < A[2], repeat the steps with LOW = 2 and HIGH = MID – 1 = 2 – 1 = 1

Is (LOW ≤ HIGH) ? No

Write("Unsuccessful search")

The following function in C implements the above concept :

```
/* function definition binary_search() */

int binary_search(float a[],int n,float data)
  {
    int low,high,mid;
    /* searching */
    low=0;
    high=n-1;
    while(low<=high)
      {
        mid=(low+high)/2;
        if(a[mid]==data) /* when element is found */
          return(mid);
        else
          {
            if(data>a[mid])
              low=mid+1;
            else
              high=mid-1;
          }
      }

    return(-1); /* when element is not found in array */
}
```

NOTES

6.3 SORTING

Sorting means arranging the elements in some specific order i.e., either ascending or descending order. The various sorting techniques available are :

(*i*) Insertion sort
(*ii*) Selection sort
(*iii*) Bubble sort or Exchange sort
(*iv*) Quick sort
(*v*) Merge sort
(*vi*) Radix sort
(*vii*) Shell sort
(*viii*) Heap sort etc.

The sorting techniques are discussed below :

(*i*) Insertion Sort

Let A be an array having N elements A[1], A[2], ..., A[N]. Initially, the first element is assumed to be sorted. In first pass, the second element *i.e.*, A[2] is inserted into its proper place in the sorted part of the array. Similarly, in the next pass, the third element *i.e.*, A[3] is placed. To make space for insertion, some of the sorted elements must be moved down in the array. After each pass

NOTES

the subarray becomes sorted from start to the element we are placing (say CURRENT). The array becomes sorted after applying N–1 passes. This algorithm is frequently used when N is small. For example, consider the array A having 6 elements as shown in figure 5.

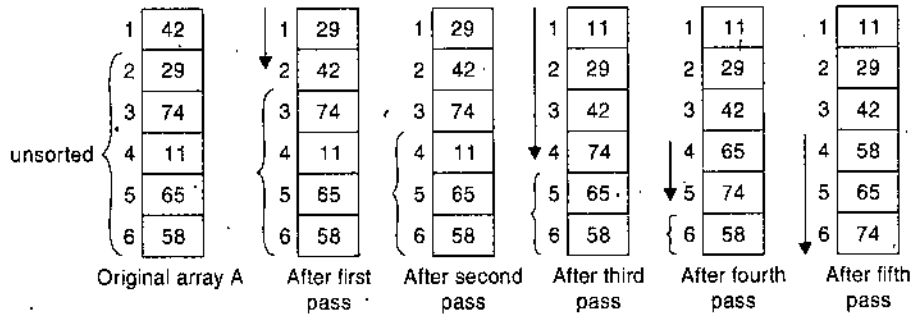


Fig. 5 *Illustration of Insertion Sort in ascending order.*

For any pass we store the element to be placed in temporary variable CURRENT. Start from first position and move downward till either the element found is greater than CURRENT or we reach the position of the element to be placed. In case we have reached the same location (*i.e.*, the position of the element to be placed) then the element lies properly otherwise move the elements downward from one position less than that of the element to be placed to the position we have located an element greater than CURRENT. Now insert the element CURRENT here.

6.3.1 Algorithm Insertion Sort

Let A be an array having N elements. We want to sort the elements in ascending order. CURRENT denotes the value of the element to be placed at proper position during a pass. POS is used for finding the appropriate position of CURRENT among the elements above it (if possible). I, J denote the array indices. Assuming the array index begins at 1.

1. Repeat for I = 2, 3, ..., N upto step 5
2. CURRENT = A[I]
3. POS = 1
4. Repeat while ((POS < I) and (A[POS] ≤ CURRENT))
   POS = POS + 1
5. If (POS ≠ I) Then

```
{
  Repeat for J=I-1,I-2,...,POS
    {
      A[J+1]=A[J]
    }
  A[POS]=CURRENT
}
```

6. End.

The following function in 'C' illustrates this concept with array index beginning at 0 :

NOTES

```
/* function definition insertion_sort() */

void insertion_sort(float a[],int n)
  {
    int i,j, current,pos; /* local variables */

    /* sorting */
    for(i=1;i<n;i++)
      {
        current=a[i]; /* current denotes the element to be arranged */
        pos=0;

        /* pos increased till values in array are <= current */

        while( (pos<i) && (a[pos]<=current) )
          pos++;
        if(pos != i) /* if position of element is not appropriate */
          {
            /* shifting */
            for(j=i-1;j>=pos;j--)
              a[j+1]=a[j];
            /* insertion at appropriate position */
              a[pos]=current;
          }
      }
}
```

(*ii*) Selection Sort

In this method we perform a search in the array, starting from the first element, to find the position of element with the smallest value. The element with the smallest value (if found) is swapped (or interchanged) with the first element in the array. As a result of this interchange, the smallest element is placed in the first position of the array. In the second pass or iteration we find the position of second smallest element starting from the second element onwards.

If such an element exists we interchange this element with the second element in the array. This process is repeated on the remaining array elements until we have placed all the elements in the proper order.

N–1 passes are required in this sorting technique as each pass places one element properly. For example, consider the following array A having 6 elements,

42 29 74 11 65 58

NOTES

In first pass, the position of the smallest element 11 is located and it is interchanged with the first element *i.e.*, 42. Figure 6 shows the array after each pass or iteration.

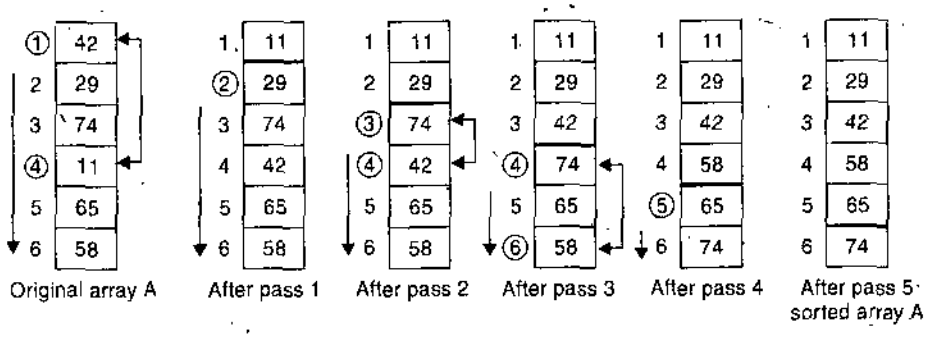


Fig. 6 *Illustration of Selection Sort in ascending order.*

The encircled indices indicate the assumed position of the smallest element and actual position of the smallest element during a pass. The downward arrow indicates the remaining portion of array which is to be searched for position of least element.

6.3.2 Algorithm Selection Sort for Ascending Order

Let A be an array having N elements. We want to sort the array in ascending order. PASS denotes the pass counter and MIN_INDEX the position of the smallest element during a pass. Variable TEMP is used for interchanging (swapping) two elements. I denotes array index. Assume the array index begins at 1.

1. Repeat for PASS = 1, 2, 3; ..., N–1

```
{
  MIN_INDEX=PASS
  Repeat for I=PASS+1,PASS+2,...,N
    {
      If (A[I]<A[MIN_INDEX]) Then
        MIN_INDEX=I
    }
  If(PASS≠MIN_INDEX) Then
    {
      TEMP=A[PASS]
      A[PASS]=A[MIN_INDEX]
      A[MIN_INDEX]=TEMP
    }
}
```

2. End

In general for the ith pass, N+i comparisons are made for searching smallest element. The maximum number of interchanges required is N–1 as there is at most one interchange during a pass. But, the actual number of interchanges

may be less than N–1 as the array elements may be in an order which may not require interchange for each element (if placed properly).

The following function in C implements the above concept :

NOTES

```
/* function definition selection_sort() */

void selection_sort(float a[],int n)
{
   float temp; /* temp is used here for swapping */
   int i,pass,min_index;

   /* min_index denotes position of the least element during a pass */

   /* sorting */
   for(pass=0;pass<n-1;pass++)
     {
       /* assume pass as the position of the least element */

       min_index=pass;

       /*search for the position of the least elements among the remaining
         elements of the array ( if any ) */
       for(i=pass+1;i<n;i++)
         {
           if(a[i]<a[min_index])
             min_index=i;
         }
         /* if assumption is not appropriate */
         if(pass != min_index)
         {
           /* swap the elements */
                    temp = a[pass];
                 a[pass] = a[min_index];
            a[min_index] = temp;
         }
     }
}
```

(iii) Bubble Sort or Exchange Sort

In this method we pass through the array sequentially many times. Each pass places the largest unsorted element in its proper position by comparing each element in the array with its successor element and swapping the two elements if these are not in proper order. The number of exchanges prior to each pass is initialized to 0 and incremented if two elements are swapped. This procedure is

NOTES

repeated from start to one position less than that of the last unsorted element (as the elements are compared pairwise when we reach second last element in the unsorted array the last element is also included).

The first pass places the largest element in the array at the last location. If no exchanges take place during any pass the next pass is not applied and the array becomes sorted resulting into algorithm termination, otherwise, we move one position up after placing an element properly, this element is left out in the next pass. Again exchanges are initialised to 0 and next pass is applied to place the largest element left in the unsorted part of the array. An extra pass is applied after the array becomes sorted for checking that no exchanges takes place in next pass. In the worst case N–1 passes are applied for sorting N elements. Only one pass is needed if the given array is already sorted. As lighter (smaller) elements move up in the array during a pass and heavier (bigger) elements move down and finally each element "**bubbles**" upto its exact location, this is why the method is known as **bubble sort.**

For example, consider the array A having 6 elements as shown in figure 7 :

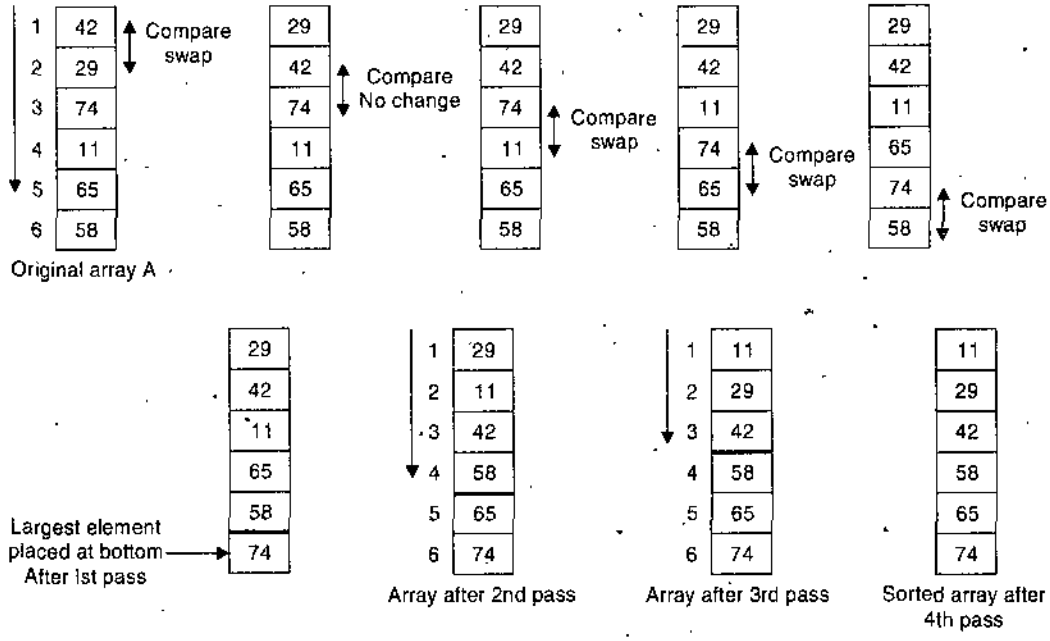


Fig. 7 *Illustration of Bubble sort.*

We may check that for the above data,

Number of exchanges in first pass = 4

Number of exchanges in second pass = 2

Number of exchanges in third pass = 1

Number of exchanges in fourth pass = 0, so array becomes sorted after 4th pass.

6.3.3 Algorithm Bubble Sort for Ascending Order

NOTES

Let A be an array having N elements. We want to sort the array in ascending order. PASS denotes the pass counter and LAST the position of the last unsorted element during a pass. EXCHS denote the number of exchanges during a pass. Variable TEMP is used for swapping of elements. I denotes array index. Assume the array index begins at 1.

1. LAST = N
2. Repeat for PASS = 1, 2, ..., N–1 upto step 5
3. EXCHS = 0
4. Repeat for I = 1, 2, ..., LAST–1

```
{
  If (A[I]>A[I+1]) Then
    {
      TEMP=A[I]
      A[I]=A[I+1]
      A[I+1]=TEMP
      EXCHS=EXCHS+1
    }
}
```

5. If (EXCHS = 0) Then

```
        goto step 6
Else
        LAST=LAST-1
```

6. End

NOTE *It is better to use bubble sort when array elements are partially or fully sorted. Only one pass is required when the given array is already sorted.*

The following function in C implements the above concept with array index beginning at 0 :

```
/* function definition bubble_sort() */

void bubble_sort(float a[],int n)
  {
    float temp; /* temp is used here for swapping */
    int i,last,pass,exchs;

    /*last denotes the position of last unsorted element
      pass denotes pass counter
      exchs denotes the number of exchanges during a pass */
    /* sorting */
    pass=0;
    last=n-1;
```

NOTES

```
        do
          {
            pass++;
            exchs=0
            for(i=0;i<last;i++)
              {
                if(a[i]>a[i+1]
                  {
                    exchs++;
                    temp=a[i];
                    a[i]=a[i+1]; /* swapping */
                    a[i+1]=temp;
                  }
              }
            last--;
          }
        while( (exchs!=0) && (pass!=n-1) );
        printf("\n\nNumber of pass(es)used for sorting = %d\n",pass);
}
```

(iv) Quick Sort

Given an array A of N elements. The **quick sort** method uses the divide-and-conquer approach for sorting the elements. In this method the N elements to be sorted are artitioned into three segments (or groups)—a left segment *left*, a middle segment *middle*, and a right segment *right*. The middle segment contains only one element; no element in *left* has a value larger than the value of the element in *middle;* and no element in *right* has a value that is smaller than that of the middle element. As a result, the elements in *left* and *right* can be sorted independently, and no merge is required after the sorting of *left* and *right*. The element in *middle* is called the **pivot** or **partitioning element.** The sort method is explained more precisely as given below :

Suppose variables LB and UB represent the indices of the first and last elements of the array respectively.

IF(LB < UB) Then

```
{ Select an element from A[LB : UB] for middle. This element is the pivot.
  Partition the remaining elements into the segments left and right so that
        no element in left has a value larger than that of the pivot and
        no element in right has a value smaller than that of the pivot.
        Sort left using quick sort recursively.
        Sort right using quick sort recursively.
}
```

The answer is *left* followed by *middle* followed by *right*.

NOTES

We can improve the performance by appropriate selection of pivot. Let us consider the case when the *pivot* is always the element at position **LB**. To begin with, assign the index or position of the first element of the array to I variable, and index or position of the last element of the array to J variable. Now perform the following :

1. [Swap element ≥ pivot on left side with elements <= pivot on right side]
   (*a*) staring with the element with position I+1, the array is scanned from left to right, comparing each element in it with the element *pivot*, till element greater than or equal to the element pivot is found, taking into cosideration that I ≤ UB.
   (*b*) Starting with the element with position J, the array is scanned from right to left, comparing each element in it with the element **pivot**, till element smaller than or equal to the element **pivot** is found, taking into consideration that J > LB.
2. [Check if swap pair found]
   If (I ≥ J) Then
   goto step 4
3. [Swap the elements]
   Swap the elements A[I] and A[J]
   goto step 1
4. [Place **pivot** at proper position]
   Assign the value of A[J] to A[LB], and store **pivot** in A[J]

As this procedure ends, the first element, **pivot**, of the original array will be lying at its final position that is *middle*. The elements in *left* will be less than this element and the elements in *right* will be greater than this element.

The same procedure can be now seperately applied on *left* and *right* sub-arrays.

For example, consider the array A having 6 elements as shown below :

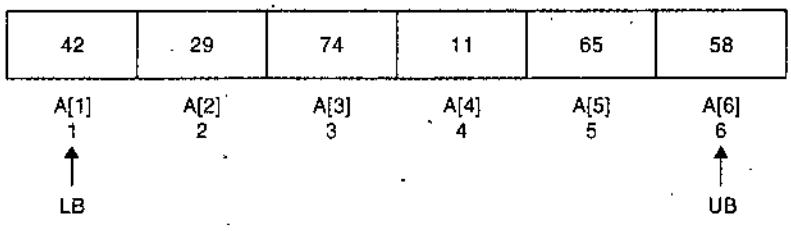


Fig. 8 (*a*)

Here UB > LB, the pivot element is A[LB], that is 42. To begin with set I = 1, J = 6

NOTES

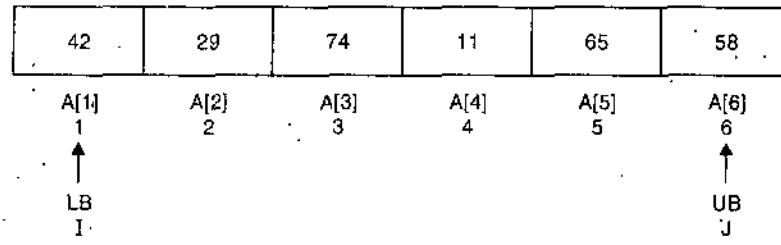


Fig. 8 (*b*)

Start scanning elements from left with position I + 1

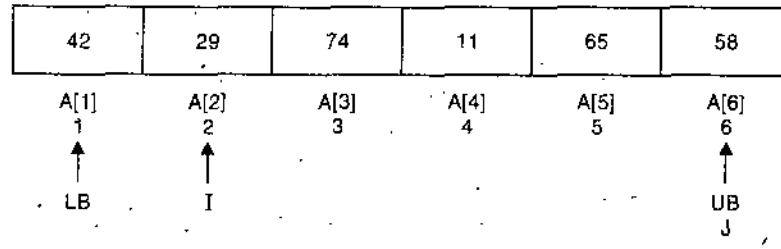


Fig. 8 (*c*)

Since A[2] < 42, we increase the value of I by 1 to get

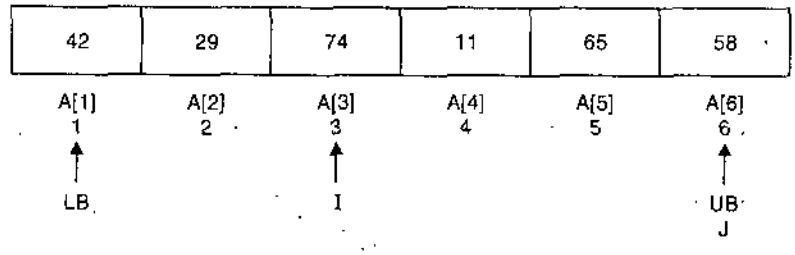


Fig. 8 (*d*)

Now, A[3] **is not** < 42, so start scanning the elements from right with position J = 6. Since A[6] > 42, we decrease the value of variable J by 1 to get,

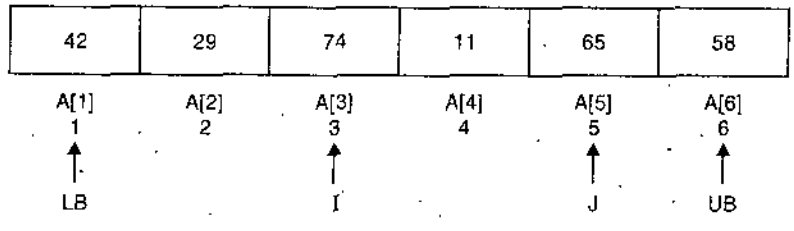


Fig. 8 (*e*)

Since again A[5] > 42, we decrease the value of J to get

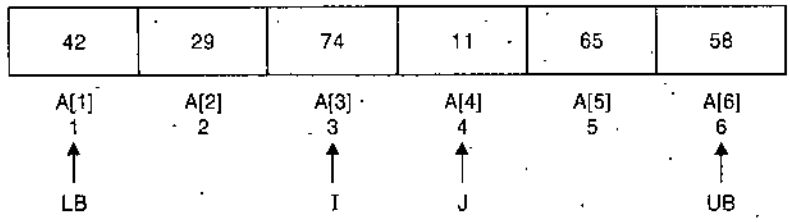


Fig. 8 (*f*)

Now, A[4] **is not** > 42 and at this stage I < J so swap the elements A[I] and A[J] to get

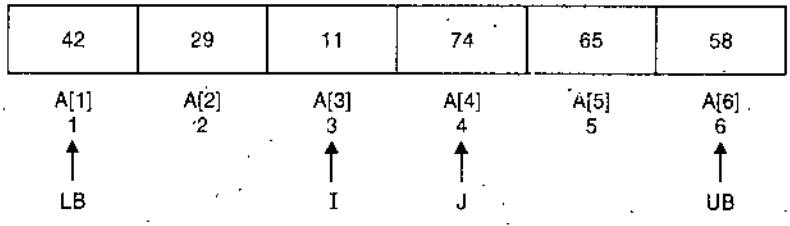


Fig. 8 (*g*)

Now, again start from left with position I+1, *i.e.,* 4

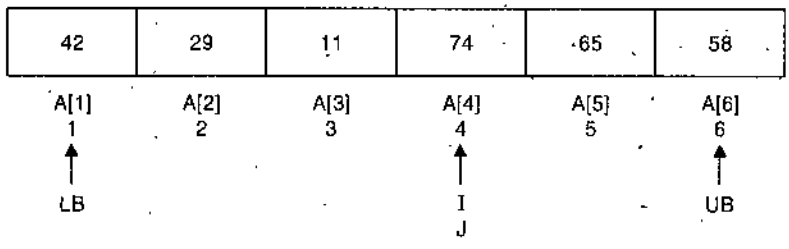


Fig. 8 (*h*)

Since A[4] **is not** < 42, so start scanning from right with position J, *i.e.,* 4. Since A[4] > 42, we decrease the value of J to get

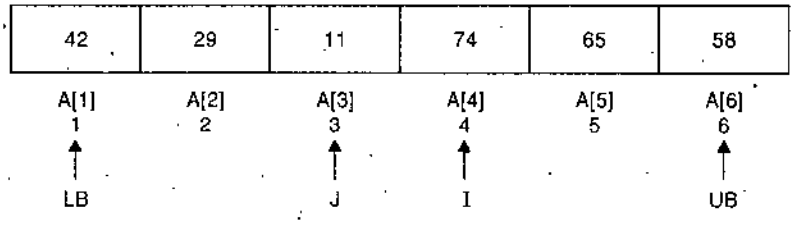


Fig. 8 (*i*)

Since A[3] **is not** > 42 and at this tage I ≥ J, thus indicating that element 42 is to be placed in its final position. Store the element A[J] in A[LB] and then place 42 in A[J] to get the following and terminate the above procedure.

NOTES

NOTES

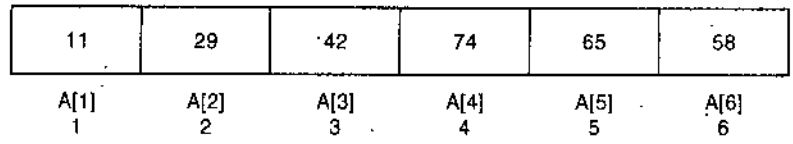


Fig. 8 (*j*)

So the original array A has been divided into three segments as shown

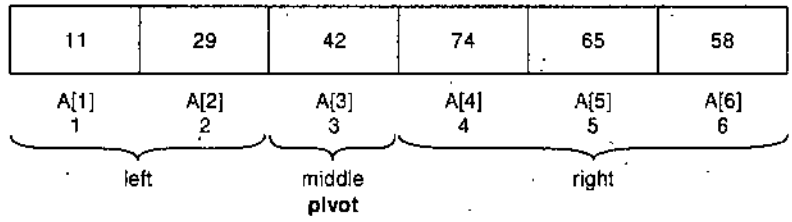


Fig. 8 (*k*)

As we can see, the elements in *left* segment are smaller than 42, and the elements in *right* segment are greater than 42.

6.3.4 Algorithm Quicksort (A, LB, UB)

Given A an array having N elements. This algorithm sorts this array in ascending order using quick sort method. LB and UB denote position of the first and last elements respectively. I and J are array indices. PIVOT contains the element to be placed in its final position within the sorted subtable. TEMP is used for swapping of elements. FLAG is a logical variable which indicates the end of the process that places the PIVOT in its final position. When FLAG becomes false, the given array has been partitioned into three segments.

1. [Initialize]
   FLAG ← TRUE
2. [Perform sorting]
   If (LB ≥ UB) Then
      goto step 8
3. I ← LB
   J ← UB
   PIVOT ← A[LB]
4. Repeat while (FLAG)
   {
   ```
   I ← I + 1
   Repeat while (A[I] < PIVOT and I ≤ UB)
     I ← I + 1
   Repeat while (A[J] > PIVOT and J > LB)
     J ← J - 1
   If(I ≥ J) Then
     Flag ← false
   ```

```
    Else
    {
       [Swap the elements]
          TEMP ← A[I]
          A[I] ← A[J]
          A[J] ← TEMP
    }
  }
```

5. [Place PIVOT at its proper position]

   A[LB] ← A[J]

   A[J] ← PIVOT

6. [Sort left segment]

   CALL Quicksort (A, LB, J–1)

7. [Sort right segment]

   CALL Quicksort (A, J+1, UB)

8. End.

The above algorithm is used initially by the statement **CALL Quicksort (A, 1, N).** The following program implements the above concept :

```
/* quick sort for ascending order */

#include<stdio.h>
#define SIZE 20
void main()
{
   void enter(float [],int); /* function prototype */
   void display(float [],int);
   void quick_sort(float [],int,int);
   float a[SIZE];
   int n;
   clrscr();
   printf("Enter number of elements <= %d\n",SIZE);
   scanf("%d",&n);
   printf("\nEnter %d elements\n\n",n);
   enter(a,n); /* function call */
   /* echo the data */
   printf("\nGiven array is\n\n");
   display(a,n); /* function call */
   /* sorting */
   quick_sort(a,0,n-1); /* function call */
   printf("\n\nSorted array is\n\n");
   display(a,n); /* function call */
   getch(); /* freeze the monitor */
}
```

NOTES

```
/* function definition enter() */

void enter(float a[],int n)
{
   int i; /* local variable */
   for(i=0;i<n;i++)
     scanf("%f",&a[i]);
}

/* function definition display() */

void display(float a[],int n)
{
   int i; /* local variable */
   for(i=0;i<n;i++)
     printf("%8.2f",a[i]);
}
/* recursive function definition quick_sort() */

void quick_sort(float arr[],int lb,int ub)
{
   int i,j; /* local variables */
   float pivot_value,temp;

   if(lb>=ub) /* base case for recursive function */
     return;

   i=lb; /* i is used as left to right cursor */
   j=ub; /* j is used as right to left cursor */

   pivot_value=arr[lb];

   /*swap elements >= pivot_value on left side
     with elements <= pivot_value on right side */

   while(1)
     {
       do
         {
           /* find >= element on left side */
           i++;
         } while(arr[i]<pivot_value && i<=ub);

         while(arr[j]>pivot_value && j>lb)
         {
           /* find <= element on right side */
           j--;
         }
       if(i>=j)  /* when swap pair not found */
           break;
```

```
/* swapping of elements using variable temp */

      temp = arr [i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/* place pivot_value at middle position that is j */

   arr[lb] = arr[j];
    arr[j] = pivot_value;

/* sort left segment */

quick_sort(arr,lb,j-1); /* recursive call to function */

/* sort right segment */

quick_sort(arr,j+1,ub); /* recursive call to function */
}
```

NOTES

PROGRAM 1

The output of program 1 will be :

```
Enter number of elements <= 20
6
Enter 6 elements
42 29 74 11 65 58
Given array is
        42.00          29.00          74.00          11.00          65.00
        58.00
Sorted array is
        11.00          29.00          42.00          58.00          65.00
        74.00
```

In the above program first of all the number of elements are entered and then the elements using the function **enter()** having arguments—the *array* and the *number of elements*. The entered array is echoed using the function **display()** having arguments the *array* and the *number of elements*. Function **quick_sort()** is called with arguments—the *array*, lower-bound *upper bound* of the array indices. Function **quick_sort()** sorts the elements in ascending order by calling itself repeatedly, that is, using recursion. The control is returned to **main()** and the sorted array is shown using function **display()**.

NOTES

(v) Merge Sort

Merge sort algorithm uses the divide-and-conquer method for sorting purpose. Given an array A having N elements, with LB and UB denoting the lower and upper bound of array indices. We want to arrange the elements in ascending order. This algorithm has the following general structure:

If N is one, terminate; otherwise partition the collection of elements into two halves or collections, sort each; combine (merge) the sorted halves or collections into a single sorted collection. It is a recursive method with the base case—the number of elements in the array are not more than one.

We can define the merge sort algorithm recursively as given below :

```
If (LB < UB) Then
    {
       Divide the array A into two halves
       Mergesort the left half
       Mergesort the right half
       Merge the two-sorted halves into one sorted array
    }
```

For example,

Consider the array A having 6 elements, that is N = 6

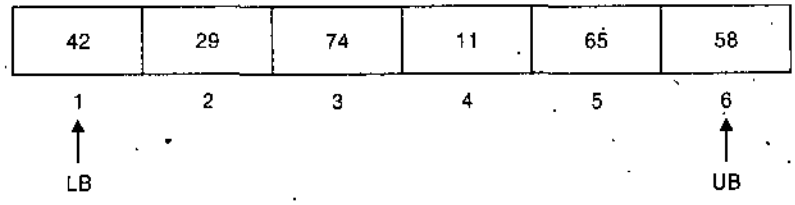


Fig. 9 (*a*) *Original array A.*

As LB < UB, so we first divide the array A into two sub arrays at position MIDDLE, where

MIDDLE = Integral part of ((LB + UB)/2) = Integral part of ((1+6)/2) = 3

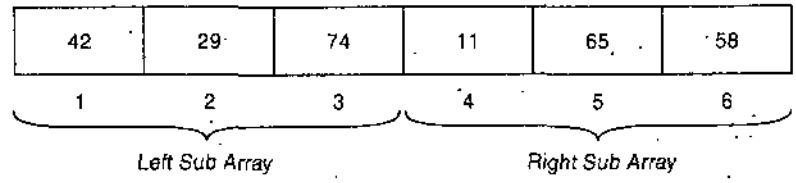


Fig. 9 (*b*) *Original array A divided into two halves.*

and first take the left subarray. It is again divided into two sub arrays at MIDDLE = Integral part of ((1+3/2) = 2 as shown below

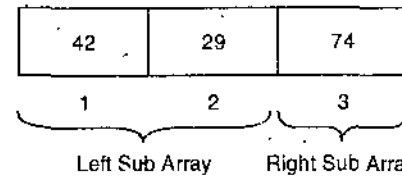


Fig. 9 (*c*) *Left sub-array of original array A divided into two halves.*

Now for left sub array, we again use the same method, dividing it again into sub arrays of one element each at MIDDLE = Integral part of ((1+2)/2)=1

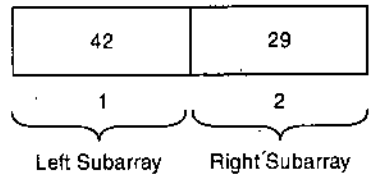


Fig. 9 (*d*) *Left subarray shown in fig 9 (c) further divided.*

Subarrays of size one as mentioned earlier, require no sorting. So the right subarray of the left subarray of original array A does not require further division.

The subarrays shown in Fig. 9 (*d*) merge to result into the sorted array and

Fig. 9 (*e*) *Mergings of subarrays in fig. 9 (d).*

the right sub array in Fig. 9 (c) on merging with the just sorted array gives the following sorted array.

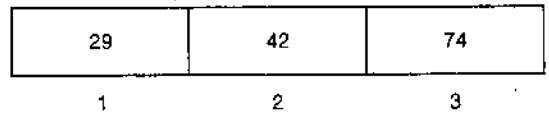


Fig. 9 (*f*) *Merging of subarrays shown in fig. 9 (e) and fig. 9 (c).*

Now the left half of the given array is sorted, we apply the same method on the right sub arrays of the original array. First we divide it into two sub arrays at MIDDLE = Integral part of ((4+6)/2) = 5 as shown below

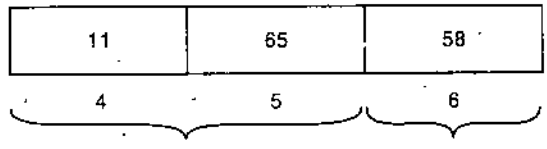


Fig. 9 (*g*) *Right subarray of original array A divided into two halves.*

Now for left sub array, we again use the same method, dividing it again into subarrays of one element each at MIDDLE = Integral part of ((4+5)/2) = 4

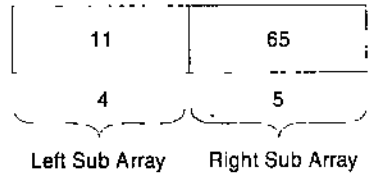


Fig. 9 (*h*) *Left subarray shown in fig 9 (g) further divided.*

NOTES

NOTES

As the right subarray in fig is of size one, so it requires no sorting. The subarrays in Fig. 9 (*h*) on merging result into the sorted array and the right subarray

Fig. 9 (*i*) *Merging of subarrays shown in fig. 9 (h).*

in fig 9 (*g*) on merging with the just sorted array gives the following sorted array

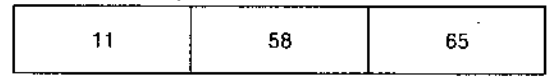


Fig. 9 (*j*) *Merging of subarrays shown in fig. 9 (i) and fig. 9 (g).*

Finally, the two sorted sub arrays of size three each shown in fig. 9 (*f*) and fig. 9 (*j*) are merged to give the sorted array

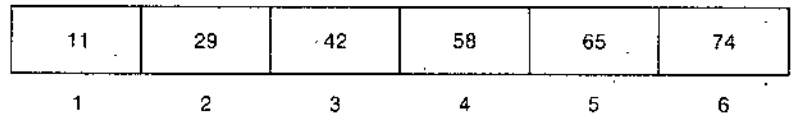


Fig. 9 (*k*) *Resultant array A having elements in ascending order.*

The algorithm for merging is given below :

6.3.5 Algorithm Merge (A, Low, Mid, High)

Given two ordered (ascending order) subarrays stored in an array A with LOW, MID and HIGH as array indices; where the LOW through the MID elements and the MID+1 through the HIGH elements represent the left and right sorted subarrays respectively. TEMPARR is a temporary array used in the merging process which is of the same size as that of array A. The variables I and J denote the index (cursor) associated with the first and second subarrays, respectively. K is an index variable associated with the array TEMPARR.

1. [Initialize]

   I ← LOW

   J ← MID + 1

   K ← LOW

2. [Compare the corresponding elements and store the smallest]

   Repeat while (I ≤ MID and J ≤ HIGH)

```
   {
      If (A [I] ≤ A[J]) Then
         {
            TEMPARR[K] ← A[I]
               I ← I+1
         }
```

```
    Else
       {
          TEMPARR[K]←A[J]
             J ← J+1
       }
    K ← K+1
}
```

3. [Copy the remaining elements]

```
If (I ≤ MID) Then
{
   Repeat while(I ≤ MID)
      {
         TEMPARR[K]←A[I]
         I ← I + 1
         K ← K + 1
      }
}
Else
{
   Repeat while(J ≤ HIGH)
      {
         TEMPARR[K] ← A[J]
         J ← J+1
         K ← K+1
      }
}
```

4. [Copy elements from TEMPARR to original array A]

```
   Repeat for I = LOW, LOW +1, ------, HIGH.
         A[I] ← TEMPARR[I]
```

5. End

Note that the timing performance of this algorithm is O(n) where n denotes the sum of the sizes of the two subtables to be merged.

Given an array having N elements. Let us consider this array to be a set of N arrays, each of which contains a single element. Obviously, an array which contains a single element is sorted. The following algorithm perform a merge sort :

6.3.6 Algorithm mergesort (A, lB, UB)

Given an array A, it is required to sort recursively its elements between positions LB and UB (both inclusive). MIDDLE denotes the position of the middle element of the current subarray.

NOTES

NOTES

1. [Test base condition for subarray of size one]
   If (LB < UB) Then
   {
      [Calculate mid-point position of current subarray]
      MIDDLE ← Integral part of ((LB + UB)/2)
      [Recursively sort first subarray]
      CALL MERGESORT(A, LB, MIDDLE)
      [Recursively sort second subarray]
      CALL MERGESORT (A, MIDDLE+1, UB)
      [Merge two ordered subarrays]
      CALL MERGE(A, LB, MIDDLE, UB)
   }
2. End

The algorithm MERGESORT is initially called (invoked) as given below :

CALL MERGESORT (A, 1, N)

where N denotes the number of elements (that is, SIZE) of the original array to be sorted.

The following function in 'C' implement the above concept :

```
/* recursive function definition merge_sort() */

void merge_sort(float a[],int lb,int ub)
{
   void merge(float [],int,int,int); /* function prototype */
   int middle; /* local variable */
   if(lb<ub)
     {
       middle=(lb+ub)/2; /* divide the array into two halves */
       merge_sort(a,lb,middle); /* function call for left half */
       merge_sort(a,middle+1,ub); /* function call for right half */
       merge(a,lb,middle,ub); /* function call merge() */
     }
}

/* function definition merge() */

void merge(float a[],int low,int mid,int high)
{
   float temparr[SIZE]; /* local variable deaclared */
   int i,j,k;
   i=low;   /* i is cursor for first segment */
   j=mid+1; /* j is cursor for second segment */
   k=low;   /* k is cursor for resultant segment */
   while( i<=mid && j<=high)
```

```
    {
      if(a[i]<=a[j])
        {
          temparr[k]=a[i];
          i++;
        }
      else
        {
          temparr[k]=a[j];
          j++;
        }
      k++;
    }
  if(i<=mid) /* if elements in first segment are left */
    {
      for(;i<=mid;i++)
        {
          temparr[k]=a[i];
          k++;
        }
    }
  else       /* if elements is second segment are left */
    {
      for(;j<=high;j++)
        {
          temparr[k]=a[j];
          k++;
        }
    }
  /* copy the elements from array temparr[] to array a[] */
  for(i=low;i<=high;i++)
    a[i]=temparr[i];
}
```

(*vi*) Radix Sort

Given an array A having N positive integers. We want to sort the integers in ascending order. As the *base* or *radix* of decimal number system is 10, we require ten pockets (buckets). In general, integers consisting of more than one digit are sorted. In such a case, an ascending-order sort can be done by performing several individual digit sorts in order. That is, each column is sorted in turn starting with the lowest-order (right-most) column and proceeding through the other columns from right to left, that is, first on unit place digit, then on tens place digit, then on hundredth place digit, and so on. For example, consider the following array A having 7 elements :

NOTES

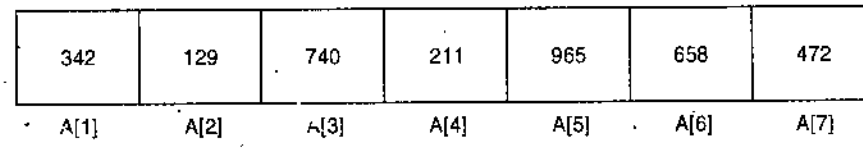


Fig. 10 (*a*) *Array a having 7 elements.*

After the first pass on the unit digit position of each number we have the pockets :

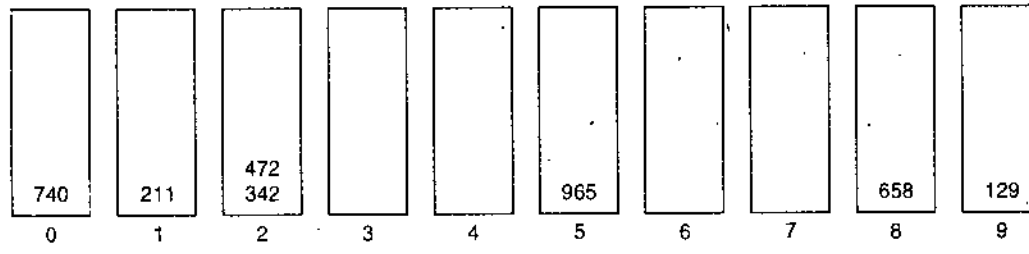


Fig. 10 (*b*) *Status of pockets after pass 1.*

Now by collecting the integers from pockets into array A we have :

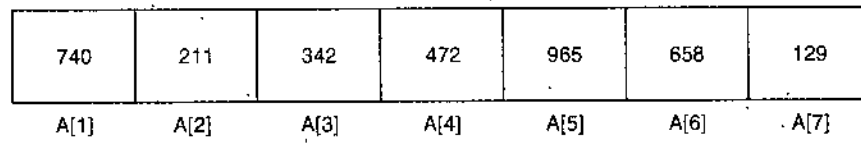


Fig. 10 (*c*) *Array A after pass 1 sorted on unit digit.*

After the second pass on the tens digit position of each number we have the pockets :

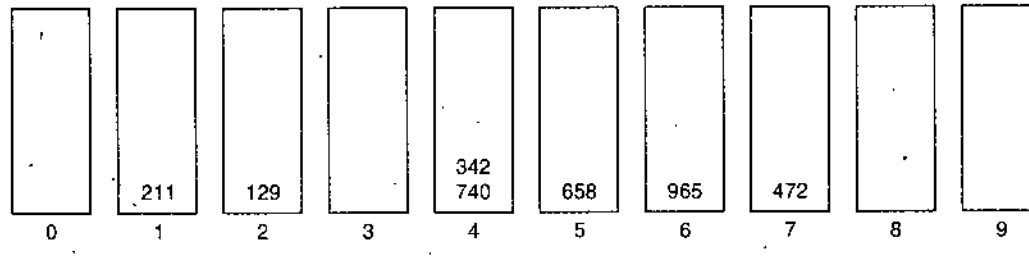


Fig. 10 (*d*) *Status of pockets after pass 2.*

Now by collecting the integers from pockets into array A we have :

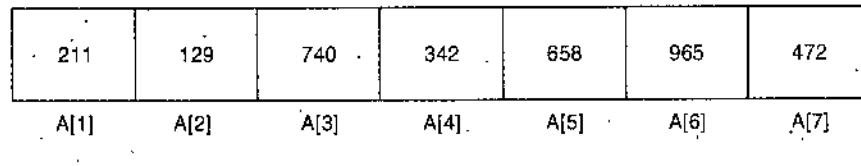


Fig. 10 (*e*) *Array A after pass 2 sorted on tens digit.*

After the third pass on the hundredth digit position of each number we have the **pockets :**

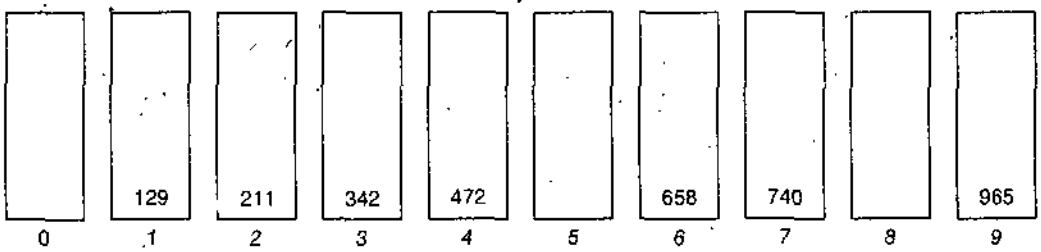


Fig. 10 (*f*) *Status of pockets after pass 3.*

Now by collecting the integers from pockets into array A we have :

Fig. 10 (*g*) *Array A after pass 3 sorted on hundredth digit.*

As the maximum number of digits in the given integers is 3 so no further pass is required. Therefore, the sorted array is

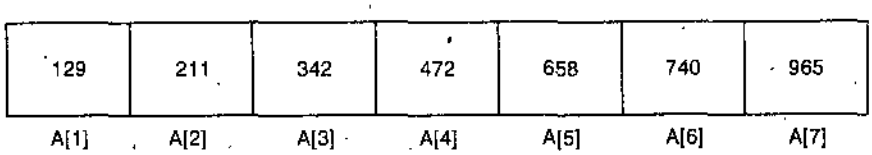


Fig. 10 (*h*) *Sorted array A.*

The algorithm for Radix sort is given below :

Given an array A having N positive integers with index beginning at 1. We are to arrange the integers in ascending order. BIGGEST denotes the largest integer in the given array. The variable MAXDIGITS is used to store the number of digits in the largest number in order to perform the maximum number of passes. DIVISOR is used for dividing given integers and R for storing remainder. PASS denotes the pass counter. I, J and K denote array indices. POCKET is a two dimensional array of size 10 by N used for storing integers during passes. COUNT is a one dimensional array of size 10 used for storing the numbers of integers, that is, count in different pockets. Indices of POCKET and COUNT array start at 0,0 and 0 respectively.

1. [Initialise]
   BIGGEST ← A[1]
2. [Find the largest integer among array elements]
   Repeat for I = 2, 3, ...,N
   {
   If (A[I] > BIGGEST) *Then*
   BIGGEST ← A[I]
   }

NOTES

3. MAXDIGITS ← 0
4. [Find number of digits in largest number]
   Repeat while(BIGGEST > 0)
   {
      MAXDIGITS ← MAXDIGITS + 1
      BIGGEST ← Integral part of (BIGGEST/10)
   }
5. [Initialise divisior for least significant digit of integers]
   DIVISOR ← 1
6. [Perform passes]
   Repeat for PASS = 1, 2, ..., MAXDIGITS
   {
      [Initialise count for all pockets]
      Repeat for I=0, 1,2...,9
         count[I] ← 0
      [Put integers in pockets according to current significant digit]
      Repeat for I = 1, 2, ...,N
         {
            [Find remainder , here MOD used for finding remainder on integer division]
            R ← (Integral value of (A[I]/DIVISOR)) MOD 10
            POCKET[R, COUNT[R]] ← A[I]
               COUNT[R] ← COUNT[R] + 1
         }
      [Collect integers from buckets into array A]
         K ← 1
      Repeat for I = 0, 1, 2, ...,9
         {
            Repeat for J = 0, 1, 2, ...,COUNT[I] - 1
               {
                  A[K] ← POCKET[I,J]
                     K ← K + 1
               }
         }
      DIVISOR ← DIVISOR × 10
   }
7. End

The following function in C implements the above concept :

```
/* function definition radix_sort() */

void radix_sort(int a[],int n)
{
   int pocket[10][SIZE],count[10]; /* local variables */
```

```
int i,j,k,r,pass,biggest,divisor,maxdigits=0;

/* find the biggest number among integers */
biggest=a[0];
for(i=1;i<n;i++)
  {
    if(a[i]>biggest)
      biggest=a[i];
  }
/* find the number of digits in the biggest number */
while(biggest>0)
  {
    maxdigits++;
    biggest /= 10;
  }
/* sorting */
divisior=1; /* divisor for least significant digit of integers */
for(pass=1;pass<=maxdigits;pass++)
  {
    /* initialise count for all pockets */
    for(i=0;i<10;i++)
      count[i]=0
    /* put integers in pockets according to current significant
       digit */
    for(i=0;i<n;i++)
    {
      r = (a[i] / divisor) % 10;
      pocket[r][count[r]++] = a[i];
    }
    /* collect integers from pockets into array 'a' */
    k=0;
    for(i=0;i<10;i++)
      {
        for(j=0;j<count[i];j++)
          a[k++]=pocket[i][j];
      }
    printf("\n\nArray after pass %d is\n\n",pass);
    display(a,n); /* function call */

    divisor *= 10; /* for next significant digit of integers */
  }
}
```

NOTES

(*vii*) Shell Sort

Shell sort or **diminishing increment sort** is named after its developer Donald Shell. It is more significant improvement on simple insertion sort. This method

NOTES

sorts seperate subfiles of the original (given) file. These subfiles have every Kth, element of the original file. The value of K is called an **increment**. For example, if K is 5, the subfile having elements A[1], A[6], A[11], ... is just stored. Five subfiles each having one fifth of the elements of the original file are sorted in this manner. These subfiles are given below (reading across) :

Subfile 1	⟶	A[1]	A[6]	A[11]	.....
Subfile 2	⟶	A[2]	A[7]	A[12]	.....
Subfile 3	⟶	A[3]	A[8]	A[13]	.....
Subfile 4	⟶	A[4]	A[9]	A[14]	.....
Subfile 5	⟶	A[5]	A[10]	A[15]	.....

The K subfiles are divided in such a way, so that the ith element of the jth subfile is given by A[(i–1) × k + j – 1]

These K subfiles are sorted usually by simple insertion sort. Now the value of K is decremented and the file is again partitioned into a new set of subfiles. These larger subfiles are sorted and this process is applied again with an even smaller value of K. Finally, the value of K is set of 1 so that the subfile having the entire file is sorted. A decreasing sequence of increments is fixed in the beginning of the entire process. The last value in the sequence must be 1. For example, consider the file having 6 elements and we want to sort the elements in ascending order (Note that the output of one pass becomes the input of the next pass.

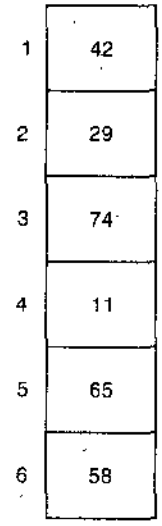


Fig. 11 ***(a)*** *Original file having 6 elements.*

Pass 1 : Starting with increment = 4

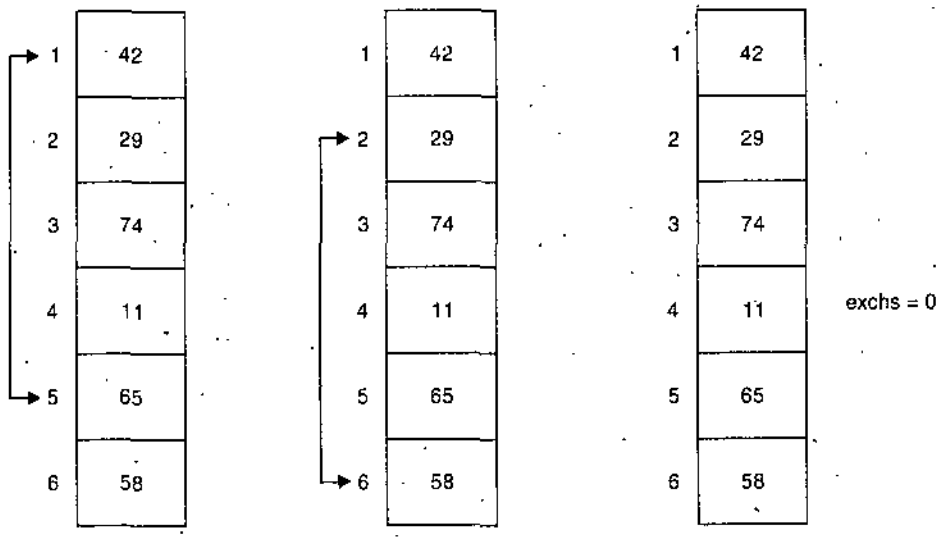


Fig. 11 (*b*) *File after action of Pass 1.*

Pass 2 : Starting with increment = 1

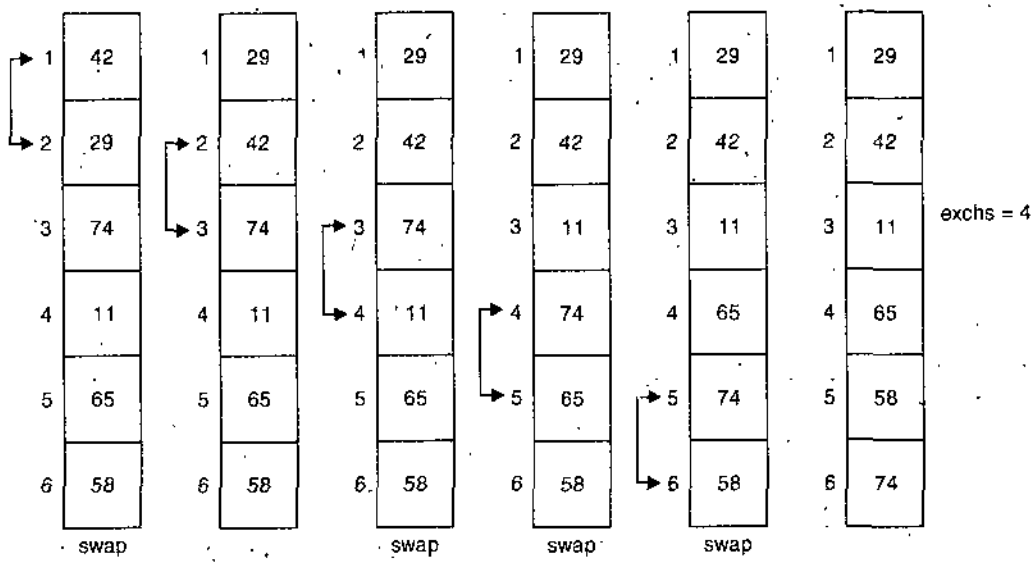


Fig. 11 (*c*)

NOTES

NOTES

As the number of exchs = 4 so repeat this process

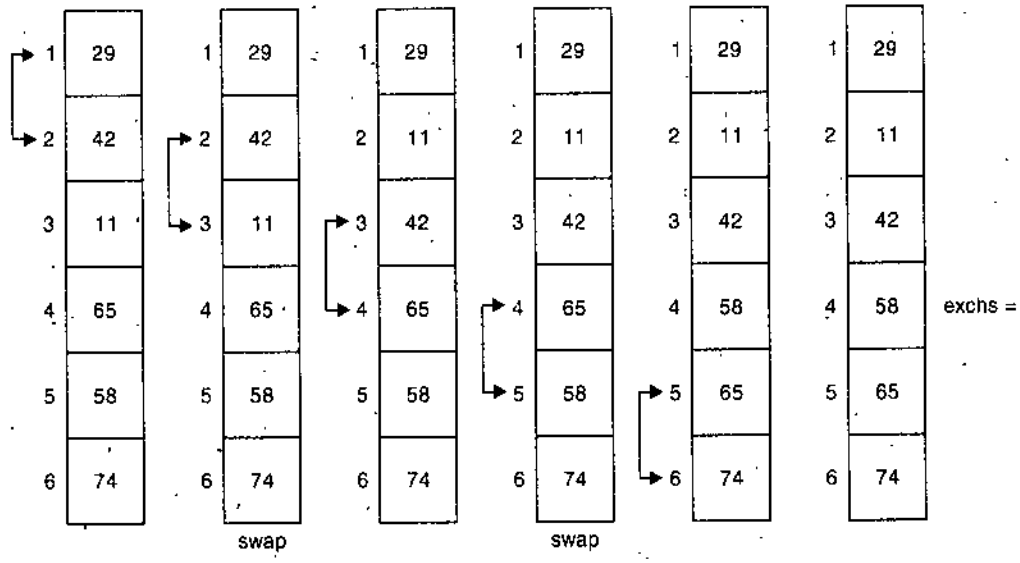


Fig. 11 (*d*)

As the number of exchs = 2 so repeat this process

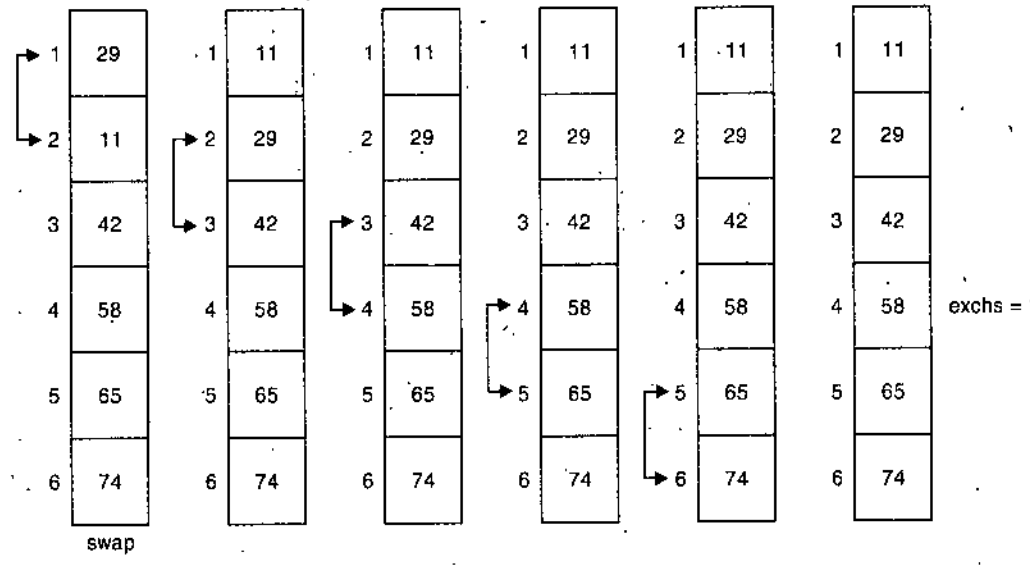


Fig. 11 (*e*)

As the number of exchs = 1 so repeat this process

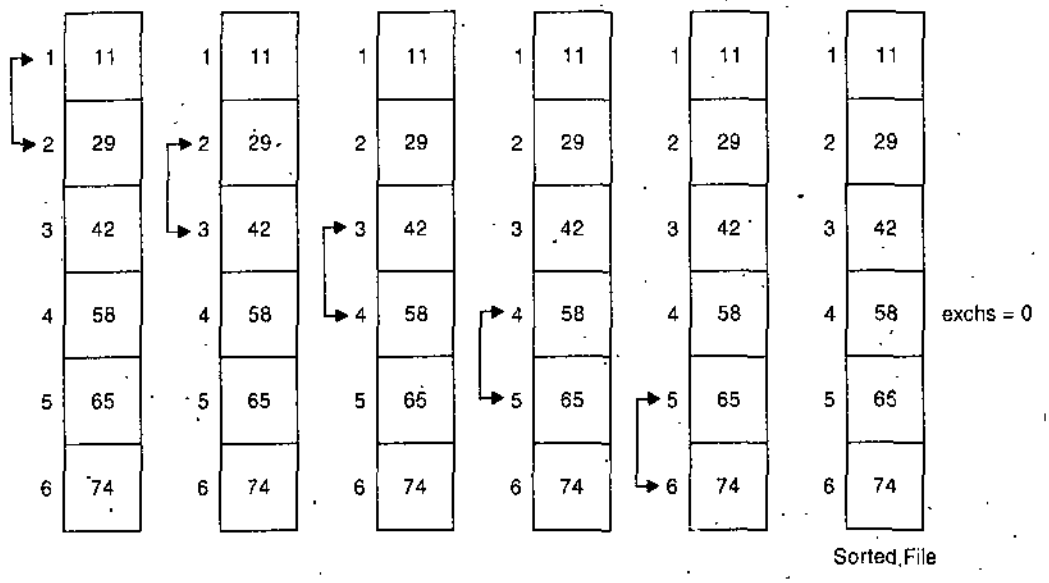


Fig. 11 (f)

As the number of exchs = 0 so file is sorted now.

We must note that if a file is partially sorted using an increment K and is subsequently partially sorted using an increment j, the file remains partially sorted on the increment K. That is, subsequent partial sorts so do not disturb earlier ones.

Several studies have been made of Shell sort, but no one has been able to prove that any choice of increments is greatly superior to all others. However, we must avoid choosing increments as powers of 2, such as 8, 4, 2 and 1 etc., as the same keys compared in one pass would be compared in next one. If we select the *increments* which one relatively prime (that is, have no common divisors other than 1) then this guarantees that successive iterations intermingle (mix) subfiles so that the entire file is indeed almost sorted when **increment** equals 1 on the last iteration. In general the shell sort is recommended for moderately sized files of several hundred elements.

The algorithm for shell sort is given below :

Given an array A having N elements. We want to arrange the elements in ascending order. I, J denote array indices. INCREMENT denotes the number of increment (gap) between the elements sorted which finally reduces to 1, to ensure that the array is completely sorted. Variables EXCHS and TEMP are used for storing the number of exchange during a pass and swapping the elements. PASS denotes the pass counter (Note INCREMENT depends on the user's choice)

1. PASS ← 0

   INCREMENT ← 1

NOTES

NOTES

2. Repeat while (INCREMENT ≤ N)
   INCREMENT ← INCREMENT × 3 + 1
3. PASS ← PASS + 1
   INCREMENT ← Integral part of (INCREMENT/3)
4. EXCHS ← 0
5. Repeat for I = 1, 2, 3, ..., N–INCREMENT
   {
      If (A[I] > A [I+INCREMENT]) Then
         {
            EXCHS ← EXCHS + 1
            TEMP ← A[I]
            A[I] ← A[I+INCREMENT]
            A[I+INCREMENT] ← TEMP
         }
   }
6. If (EXCHS ≠ 0) Then
      goto step 4
7. If (INCREMENT ≠ 1) Then
      goto step 3
8. End

The following function in C implements the above concept :

```
/* function definition shell_sort() */

void shell_sort(int arr[],int n)
{
   int i,j,increment,exchs,temp,pass=0; /* local variables */
   increment=1;

   /* choose sequence of increments */
   while(increment<=n)
     increment = increment*3 + 1;
   do
     {
      pass++;
      increment /= 3;
      printf("Iteration %d :\n",pass);
      do
      {
        exchs=0;
        for(i=0;i<n-increment;i++)
          {
            if(arr[i]>arr[i+increment])
              {
                exchs++; /* increment number of exchanges */
```

```
                /* swapping */
                        temp   = arr[i];
                        arr[i] = arr[i+increment];
                    arr[i+increment] = temp;
                }
            }
          for(j=0;j<n;j++)
            printf("%d ",arr[j]);
          printf("exchs = %d\n",exchs);

      }while(exchs);
    }while(increment != 1);
}
```

NOTES

(viii) Heap Sort

Ans- B

Let us discuss same basic concepts which are essential for understanding heap sort.

6.3.7 Definitions

A **max tree** (**min tree**) is a tree in which the value in each node is greater (less) than or equal to those in its childern (if any). Some max trees are shown in figure 12, and some min trees are shown in figure 13.

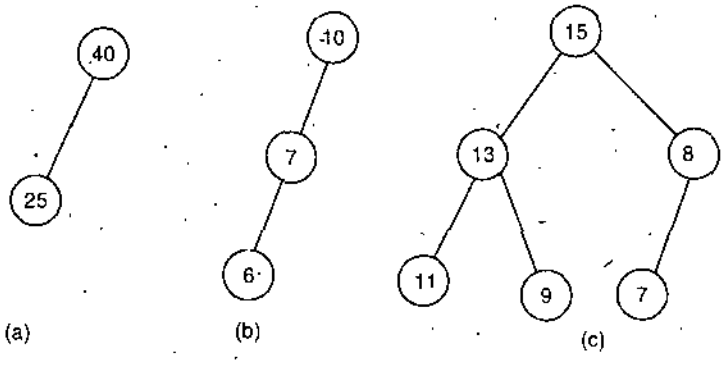


Fig. 12. *Max trees.*

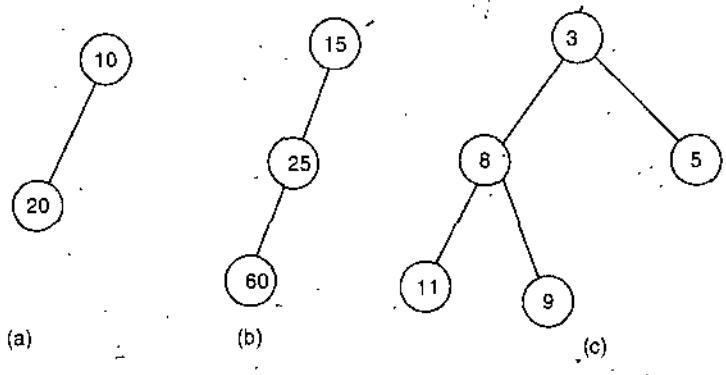


Fig. 13. *Min trees.*

NOTES

Note that max trees and min trees are binary trees, it is not necessary for a maxtree or mintree to be binary. Nodes of a max or min tree may contain more than two childern.

A **max heap** (**minheap**) of size N is a max (**min**) tree that is an almost complete binary tree of N nodes. A max heap is also called a **descending heap** or a **descending partially ordered tree.**

Generally a max heap represented by an array A of size N satisfies the property A[J] ≤ A[I] for 2 ≤ J ≤ N and I = Integral part of (J/2) assuming that array starts with index 1.

It is clear from this definition of a max heap that the root of the tree (or the first element of the array) contains the largest element in the heap. Also note that any path from the root to a leaf (or indeed, any path in the tree that includes no more than one node at any level) is an ordered list in descending order. Similarly we can define an **ascending heap** (or a **min heap**) as an almost complete binary tree such that the content of each node is greater than or equal to the content of its father.

The max tree of figure 12 (*b*) is not a max heap. The other two max trees are max heaps. The min tree of figure 13 (*b*) is not a min heap. The other two are.

Since a heap is an almost complete binary tree, it can be efficiently represented in memory using one dimensional array. Here, we will be discussing max heaps.

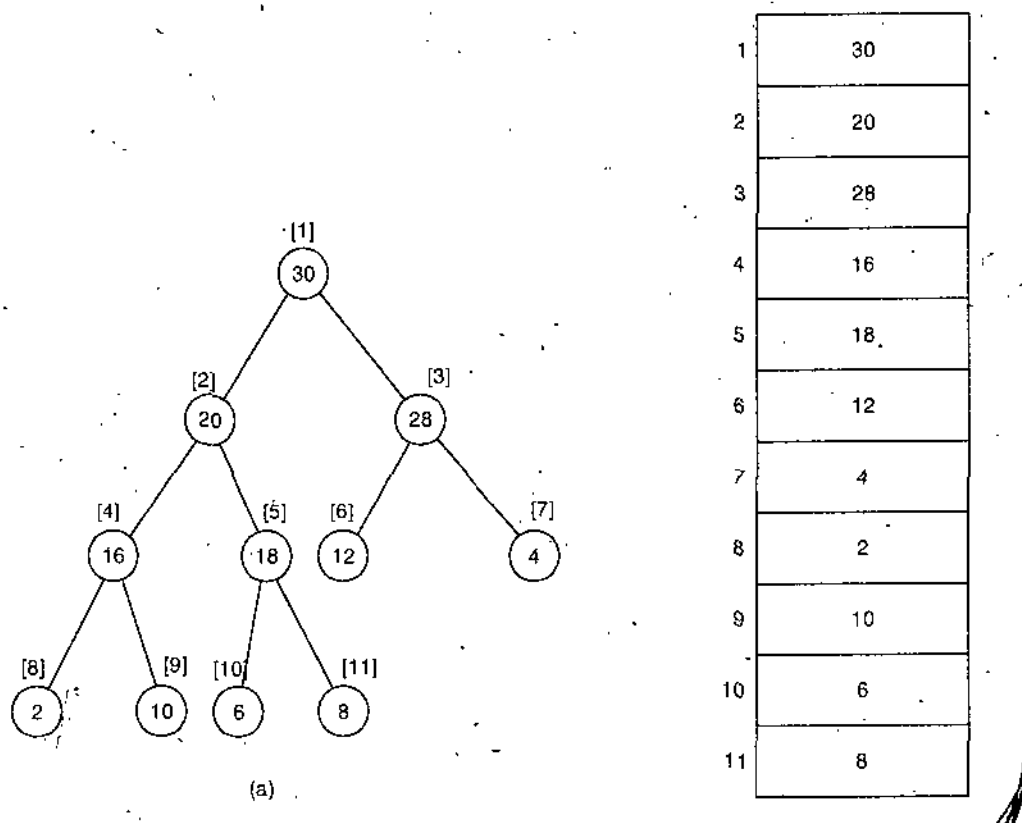


Fig. 14 *(a) Max heap of 11 elements (b) Sequential representation of max heap*

6.3.8 Insertion into a Max Heap

NOTES

Figure 15 (*a*) shows a max heap with five elements. When an element is added to this heap, the resulting six-element heap must have the structure shown in figure 15 (*b*).

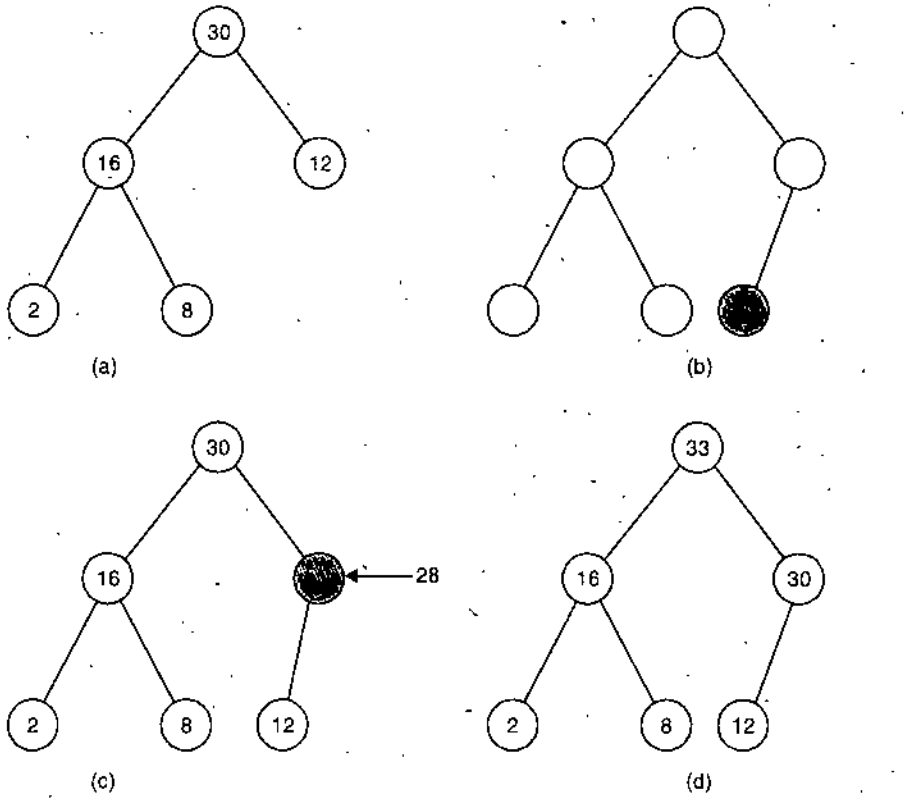


Fig. 15. *Insertion into a max heap.*

Because a heap is an almost complete binary tree. If we want to insert 10, it may be inserted as the left child of 12. If instead, the value of the new element is 28, the element cannot be inserted as the left child of 12 (as in this case, we will violate the max tree property). Therefore, the element 12 is moved down to its left child (see Figure 15(*c*)) and we check whether placing 28 at the old position of 12 results in a max heap. Since the parent element (30) is at least as large as the element 28 being inserted, we can insert the new element at the position shown in figure.

Next suppose that the new element has value 33 rather than 28. In this case the element 12 moves down to its left child as shown in figure 15 (*c*). The element 33 cannot be inserted into the old position of element 30 is moved down to its right child and the element 33 inserted in the root of the heap (figure 15 (*d*)).

This method of insertion just explained above makes a single pass from a leaf towards the root. At each level we do O(1) work, so we should be able to implement this method to have complexity O(height) = O(log n).

NOTES

6.3.9 Deletion from a Max Heap

When an element is to be deleted from a max heap, it is taken from the root of the heap. For example, a deletion from the max heap of figure 16 (*a*) results;

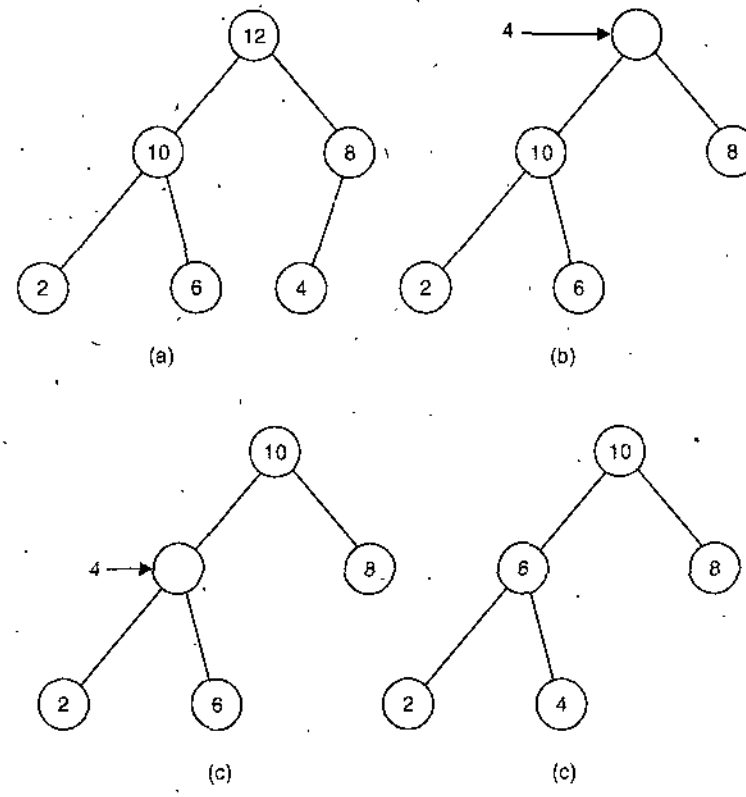


Fig. 16 *Deletion from a max heap*

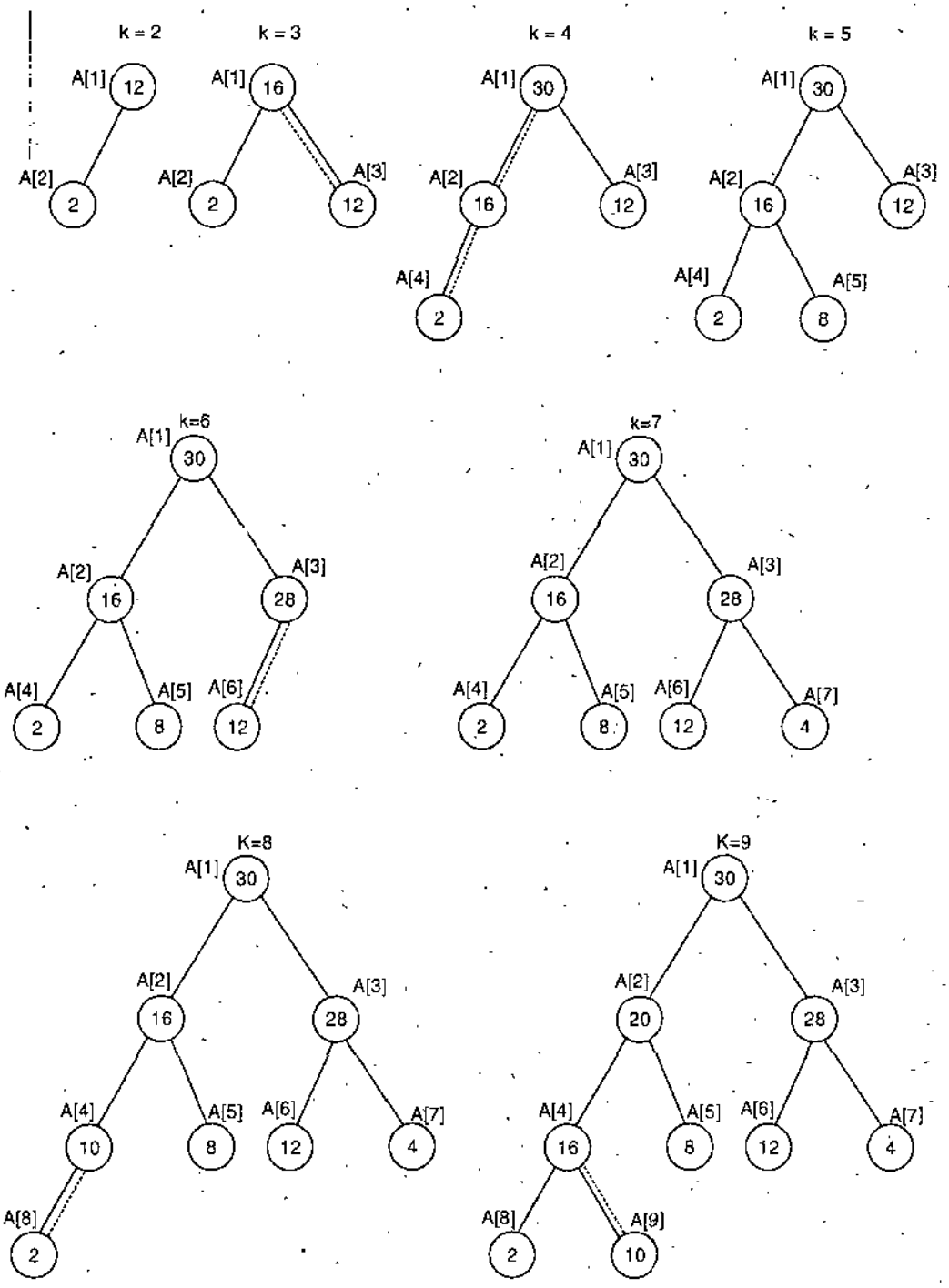
in the removal of element 12. Since the resulting max heap has only five elements, the binary tree of figure 16 (*a*) needs to be readjusted to give a complete binary tree with five elements. For this redjusting, we remove the element in position six, that is, the element 4. Now we have the structrue shown in figure 16 (*b*), but the root is vacant and the element 4 is not in the heap. If the element 4 is inserted into the root the resulting binary tree is not a max tree. The element at the root should be the largest from among the element 4 and the elements in the left and right children of the root. This element is 10. It is moved into the root there by creating a vacancy in position two. The element at position two should be the largest from among the element 4 and the elements in the left and right children at position four and five. This element is 6 (see figure 16 (c)). It is moved into position two, and the element 4 is inserted into position five. The resulting heap is shown in figure 16 (*d*).

The deletion method explained above makes a single pass from the heap root down towards a leaf. At each level O(1) work is done, so the time complexity of these operations is O(height) = O(log n).

6.3.10 Sorting using a Heap—an Example

NOTES

Let A be an array having N elements. Assume that N = 11 and the values of the elements in A[1:11] are {12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18}. We want to arrange the elements in ascending order using heap sort method. Figure 17 illustrates the creation of a heap of size 11 from the original array A. Here index variable K denotes the number of insertions which is to be performed. The dotted lines in that figure indicate an element being shifted down the tree.



NOTES

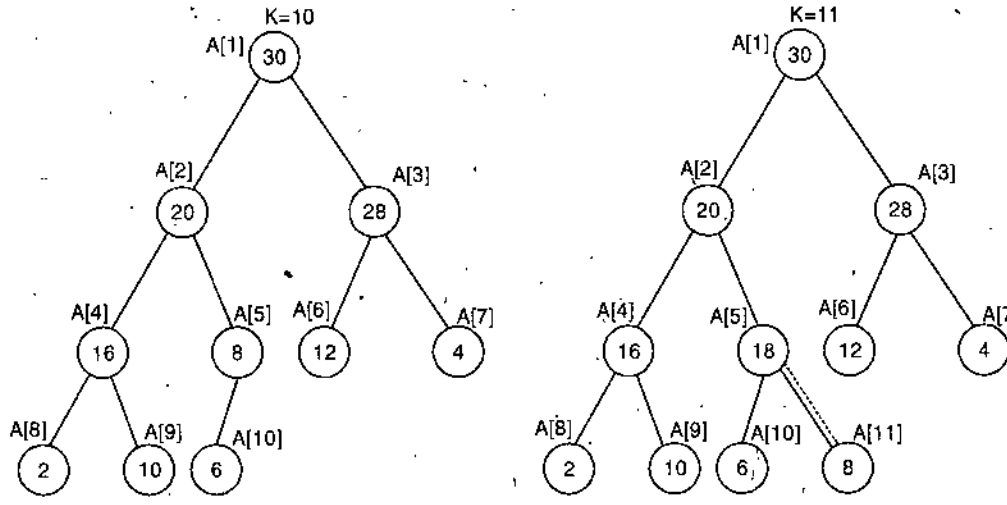
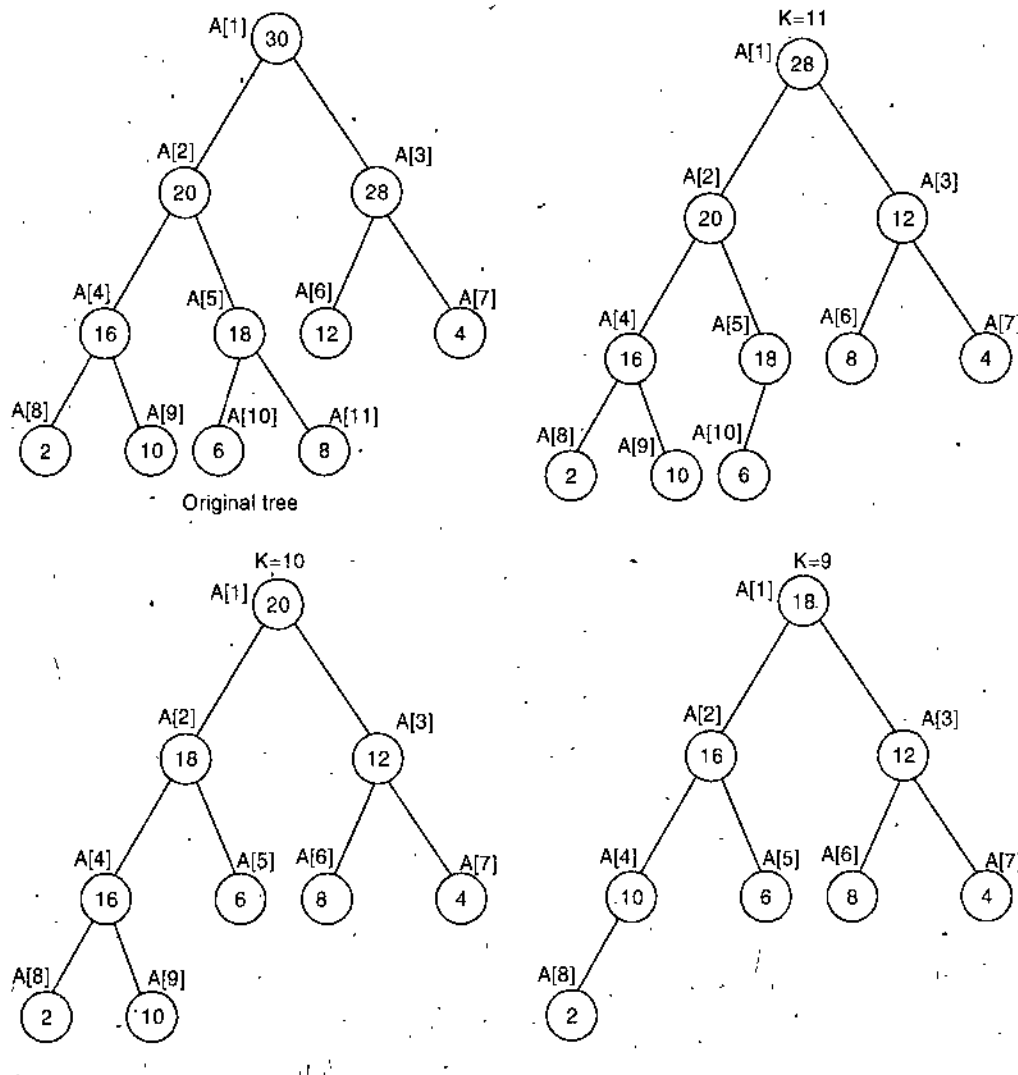


Fig. 17. *Creating a heap of size 11.*

Figure 18 illustrates the adjustment of the heap as A[1] is repeatedly selected and placed into its proper position in the array and the heap is readjusted, until



NOTES

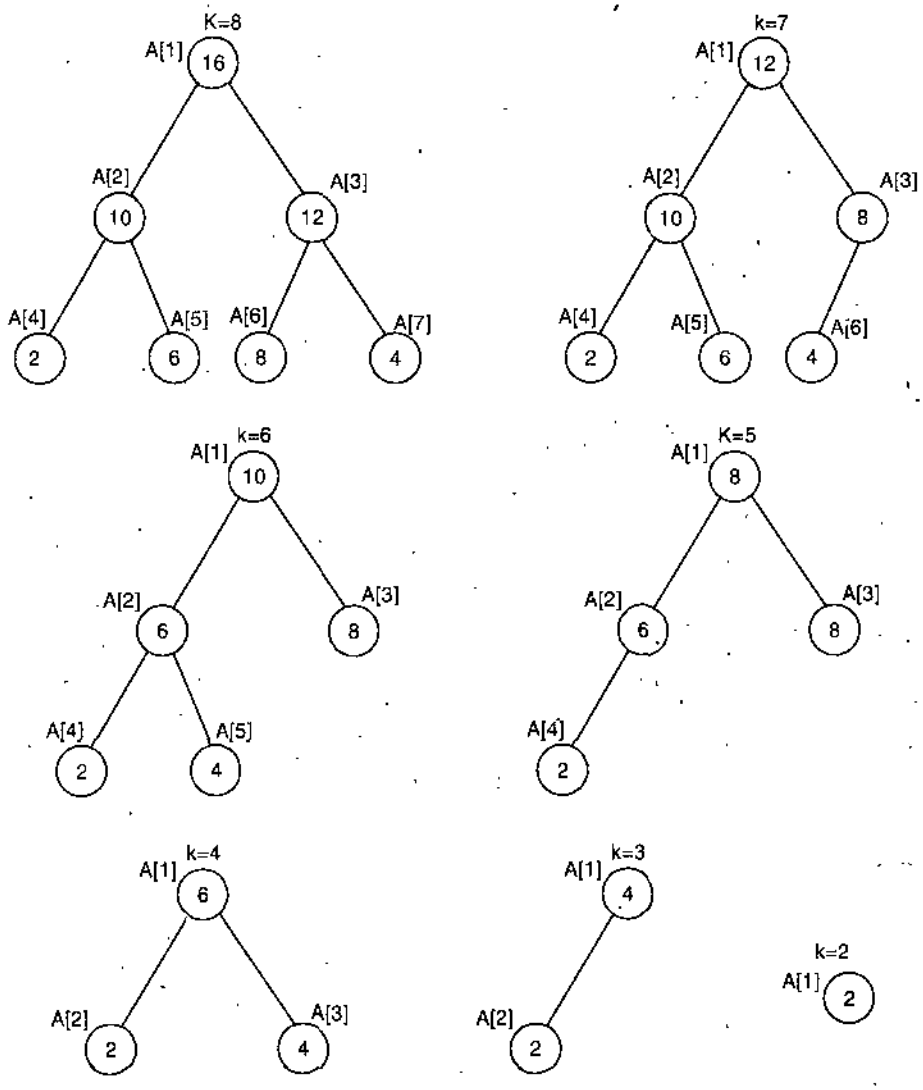


Fig. 18. *Adjusting a heap.*

all the heap elements are processed. K denotes the pass index. Note that after an element has been "deleted" from the heap, it remains in the array, it is merely ignored in the subsequent processing.

6.3.11 Algorithm Create heap (A, N)

This algorithm produces as output a max heap. Initially the heap has one element. Now elements are inserted into the existing heap such that a new max heap is formed after insertion. This procedure is repeated until all elements in the given array form a max heap.

The general approach of create heap is as follows :

1. Repeat while there still is another element to be placed in the max heap upto step 5
2. Take element to be placed at leaf level
3. Find position of parent for this element

NOTES

4. Repeat while the element has a parent and the element is greater than its parent

   Move parent down to position of element

   Find position of new parent for the element

5. Insert element into its proper place (position)

The algorithm CREATE HEAP is given below :

Given an unsorted array A having N elements. This algorithm creates a max heap. The index variable K denotes the number of insertions which is to be performed. The integer variable J denotes the index of the parent of element A[I]. Variable VALUE stores the element being inserted into an existing heap.

1. [Build max heap]

   Repeat for K = 2, 3, ... N up to step

2. [Initialize]

   I ← K

   VALUE ← A[K]

3. [Find position of parent of new element]

   J ← Integral value of (I/2)

4. [Place the new element in existing max heap]

   Repeat while (I > 1 and VALUE > A[J])

   ```
   {
      [Interchange elements]
        A[I] ← A[J]
      [Find position of next parent]
        I ← J
        J ← Integral value of (I/2)
      If (J < 1) Then
        J ← 1
   }
   ```

5. [Place the new element into its proper position]

   A[I] ← VALUE

6. End

6.3.12 Algorithm Heap Sort (A, N)

This algorithm takes as input a maxheap stored in array A having N elements. The element with the largest key is currently in A[1] and it can be written out directly. This is done by interchanging A[1] and A[N], and then it restructures a new max heap having only N-1 elements. This is done in the same way as in algorithm CREATE HEAP. Again the restructural max heap stores the second largest element in A[1]. This element can now be exchanged (swapped) with

element A[N-1]. A new max heap is then restructured for N-2 elements. By repeated application of these steps, the given array can be sorted. The general approach for heap sort is as follows :

1. Create the initial max heap.
2. Repeat for N-1 times up to step 5
3. Swap (exchange) first element with last unsorted element
4. Find position (index) of the largest child of the new element.
5. Repeat for the unsorted element in the maxheap and while the current element is greater than the first element

```
{
   Interchange elements and find the next left child
   Find position of the next largest child
   Place the element into its proper position
}
```

NOTES

The algorithm HEAP SORT is given below :

Given an array A having N elements and the algorithm CREATE HEAP which has been previously described, this algorithm sorts the array into ascending order. The variable K denotes the pass number. Variables I and J denote the array indices, where J is the index of the left child of I. Variable TEMP is used for swapping and VALUE is used for storing the element being swapped at each pass.

1. [Create the initial max heap]

   CALL CREATE HEAP(A, N)

2. [Sorting]

   Repeat for K = N, N – 1, ...,2 upto step 6

3. [Exchange elements]

   TEMP ← A[K]

   A[K] ← A[1]

   A[1] ← TEMP

4. [Initialize the pass]

   I ← 1

   VALUE ← A[1]

   J ← 2

5. [Find position of the largest child of new element]

```
If (J+1 < K) Then
{
   If ( A[J + 1] > A[J]) Then
      J ← J + 1
}
```

NOTES

```
6.  [Now reconstruct the new max heap]
    Repeat while ( J ≤ K–1 and A[J] > VALUE)
    {
       A[I] ← A[J]
       [Obtain the next left child]
          I ← J
          J ← I × 2
       [Obtain the position of next largest child]
       If (J+1 < K) Then
          {
             If (A[J + 1] > A[J]) Then
                J ← J + 1
             Else
                {
                   If (J > N) Then
                      J ← N
                }
          }
          [Copy the element in its proper position]
             A[I] ← VALUE
    }
7.  End.
```

6.4 SUMMARY OF SORTING METHODS

Some of the sorting methods discussed here are summarized in Table 1 Note that the entries in the table are approximate. The parameter m denotes the number of digits in a key. It is used in the radix sort.

Table 1. Comparison of Sorting Methods (entries are approximate)

Algorithm	*Average*	*Worst Case*	*Space Usage*
SELECTION	$n^2/4$	$n^2/4$	In place
BUBBLE SORT	$n^2/4$	$n^2/2$	In place
MERGE SORT	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra n entries
QUICK SORT	$O(n\log_2 n)$	$n^2/2$	Extra $\log_2 n$ entries
HEAP SORT	$O(n\log_2 n)$	$O(n\log_2 n)$	In place
RADIX SORT	O(m+n)	O(m+n)	Extra space for links

It is difficult to assert that a particular sorting technique is *always* superior to other methods for every key set. Certain properties of a given key set play an important role in the determination of which sorting technique should be

preferred. Properties such as the number, size, distribution, and orderness of keys often dictate which method should be used. The amount of memory available in performing the sort may also be an important factor.

In summary, the selection or bubble sorts can be used if the number of records in the table is small. If n is large and the keys are short, the radix sort can perform well. With a large n and long keys, quick, sort, heap sort, or a merge sort can be used. If the table is, initially, almost sorted, then quick sort should be avoided.

NOTES

6.5 SUMMARY

- The searching techniques are Binary Search and Sequential Search.
- A method which traverses data sequentially to locate item is called Linear or Sequential Search.
- Binary search technique can be applied only on the sorted data.
- Searching an ordered array is called *Interpolation Search.*
- Bubble sort is a sort that exchanges the neighbor elements i and $i+1$ of a sequence starting from left going to right.
- Quick sort is the best sorting algorithm when one does not have infinite memory space Quick sort originally proposed by C.A.R. Hoare, Computer Journal, April 1962.
- A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions.
- The most common algorithm used in an external sorting, that is for the problem in which data is stored in disks or magnetic tapes, merge sort is an excellent sorting method.

6.6 TEST YOURSELF

Answer the following questions :

1. Explain the algorithm for **selection sort** and give a suitable example.
2. Explain the algorithm for **exchange sort** with a suitable example.
3. Write an algorithm to sort the N elements of an array in ascending order using Bubble Sort technique.
4. Write C procedure to implement :

   (*a*) Shell sort

   (*b*) Radix sort

   Each phase of the above algorithms for the input data set in the sequence :

   F = {42, 23, 74, 11, 66, 57, 94, 36, 99, 87, 70, 81, 61}
5. Which of the sorting algorithm has best performance in terms of storage and time complexity ? Justify your answer.

❑❑❑