

2023

Data Structure and Algorithms

Dr. Babasaheb Ambedkar Open University



Data Structure and Algorithms

Course Writer

Dr. Kamesh R. Raval Assistant Professor,
Som Lalit Institute of Computer Application

Prof.(Dr.) M. T. Savliya Professor and Head,
Computer Engineering Department,
Vishwakarma Government Engineering College,
Ahmedabad

Content Reviewer and Editor

Prof. (Dr.) Nilesh K. Modi Professor & Director
School of Computer Science,
Dr. Babasaheb Ambedkar Open University

Copyright © Dr. Babasaheb Ambedkar Open University – Ahmedabad

ISBN-

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad
While all efforts have been made by editors to check accuracy of the content, the representation of facts, principles, descriptions and methods are that of the respective module writers. Views expressed in the publication are that of the authors, and do not necessarily reflect the views of Dr. Babasaheb Ambedkar Open University. All products and services mentioned are owned by their respective copyrights holders, and mere presentation in the publication does not mean endorsement by Dr. Babasaheb Ambedkar Open University. Every effort has been made to acknowledge and attribute all sources of information used in preparation of this learning material. Readers are requested to kindly notify missing attribution, if any.



Data Structure and Algorithms

Block-1: Introduction to Programming and C Language

UNIT-1

Introduction to Programming Languages 02

UNIT-2

C-Language and Console I/O Operations 17

UNIT-3

Keywords, Variables, Datatypes and Operators 33

UNIT-4

Decision Making with Branching and Looping 52

Block-2: C Programming Concepts

UNIT-1

Working with Functions 73

UNIT-2

Working with Arrays and Strings 91

UNIT-3

Structures and Unions 108

UNIT-4

Pointers 122

Block-3: Introduction to Data Structure

UNIT-1

Basics of Data Structures 135

UNIT-2

Array and Stack 157

UNIT-3

Queue 191

UNIT-4

Linked List 214

Block-4: Sorting & Searching, Nonlinear Data Structure

UNIT-1

Sorting 249

UNIT-2

Searching 278

UNIT-3

Tree 296

UNIT-4

Graph 328

Block-1

**Introduction to Programming and
C Language**

Unit 1: Introduction to Programming Languages

1

Unit Structure

- 1.1. Learning Objectives
- 1.2. Introduction
- 1.3. Hardware and Software
- 1.4. Programming Concepts
- 1.5. Programming Languages
- 1.6. Let's sum up
- 1.7. Check your Progress: Possible Answers

1.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Define Computer system, Data and Information
- Role of Hardware and Software
- Types of Software
- Logic building with algorithms and flowcharts
- Different types of languages

1.2 INTRODUCTION

Before going in to the depth of the subject, we will discuss few terms which is related to the subject Data and File Structure. In this section we will discuss some basic terminologies and concepts.

SYSTEM

System or Computer is a machine which can perform arithmetic and logical operations at a very fast speed. It can store the data and retrieve the data or information as per users' request or instruction.

The term 'Computer' is derived from the word 'Compute' means doing calculations. But today, most of the work done by the Computer in which no computation is involved. For example, we can listen music, watching movies, doing chat with friends, making resume using word processor, sending a mail, searching something on Internet etc., do not have any type of computation. So, we can define today's computer as a data processor which takes data as input and produce information as an output.

DATA

Data is unstructured raw materials and unstructured facts which will provide necessary inputs to the computer system. Here unstructured means value or set of values are not in structured format or not processed.

Data can be available in different formats. For example, numbers (34, 28, 76 etc.), numbers with decimal points (3.14, 78.65, 25.001 etc.), characters ('A', 'a', 'E' etc.),

group of characters which also known as strings (“BAOU”, “Computer Science”, “Data and File Structure” etc.), Date, ISBN number of the book and many other types of formats are available.

Data is a value or set of values which is not processed. For example, “Ram’s presence in the class on some date”, “Shyam’s marks of Maths subject” etc. By mean of one day attendance, we cannot predict that Ram is regular attending the class or not. Similarly, if we have marks of the Shyam of any one subject we cannot predict intelligence of Shyam.

INFORMATION

Information is structured data, or we can say - processed data is information. For example, if we record attendance of the Ram every day for whole semester and after processing it if we find his attendance is 78.20% is an information. If we collect marks of the Shyam for all the subjects of particular semester and generate the grade sheet after processing the marks is information. So, information is nothing but collection of processed data, which produces some meaningful output.

From the above discussion it is clear that data is unstructured facts and input to the system, while information is the processed data in meaningful or summarized format produced as an output by the system. Following figure will clarify the terms ‘data’ and ‘information’ more clearly.

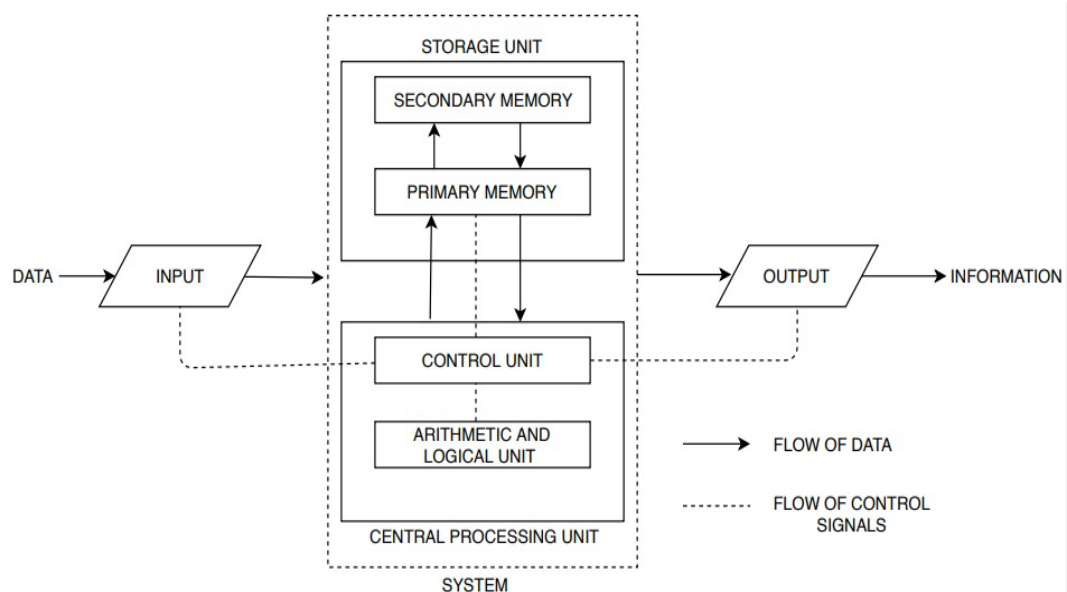


Figure-1.1 System Diagram

Diagram shown in Figure-1 is describing how system act as a data processor, which takes data from the input devices, processed it and produces information as an output. System uses storage unit, which consists of primary memory to store intermediate results and data at the time of processing and secondary storage to store information.

1.3 HARDWARE AND SOFTWARE

To input the data to the Computer system we use input devices like Keyboard, Mouse, Scanner etc. Similarly, to retrieve information from the Computer system we use output devices like Monitor, Printer, Speaker etc. The physical peripherals of the computer system, which we can see or touch are called devices or hardware of the computer. To operate all these devices series of instructions are provided to the Computer system is called software. Software is collection of programs which provides necessary instructions to fetch the data, to process the data and produce information and control the various devices of the Computer system. Fig.1.2 shows classification of the Computer system, which consists of Hardware and Software. Software can again be classified in to System software and Application software.

Those software's which directly deals with the hardware of the computer are known as System software. System software's are design to handle various computer peripherals. Following software can be considered as a system software.

1. Computer BIOS
2. Device drivers
3. Operating System
4. Utility programs like Anti-Virus software, Backup utility, Disk Checker etc.
5. Compilers, Interpreters, Linkers and Loaders

Application software's are designed to solve specific problem or perform specific task. Application software is a type of computer software that employs the capabilities of a computer directly to performs a user specified tasks, for example, calculating salary of employees, managing inventory, keeping student details etc. Following software's can be considered as application software.

1. Word processor

2. Payroll system
3. On-Line Examination software
4. Library management system
5. Student information system

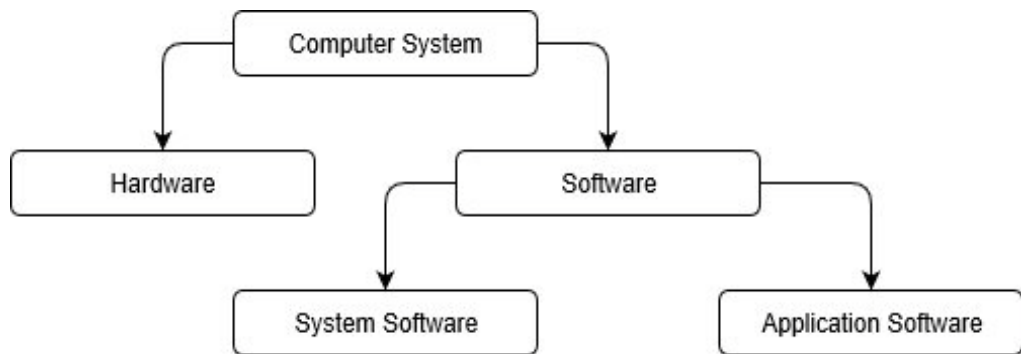


Figure-1.2 Hardware and Software system

As we have discussed earlier that the to function various hardware devices, software is essential and to design a software we need to make programs. Programs can be defined as set of instructions written by programmer in a logical sequence to solve a specific problem.

1.4 PROGRAMMING CONCEPTS

ALGORITHMS

The finite set of steps, which provide a chain of actions for solving a definite nature of problem is called Algorithm.

Algorithm is a process of writing series of steps or instructions to solve a specific problem. Algorithm do not depend on computer language, but it is just a set of instructions written to solve a problem in logical sequence. Usually algorithm can be written in natural languages like 'English'. Later on, programmer or developer of specific language will translate the logic expressed by an algorithm into a specific computerized language called computer program.

For example, if we want to explain computer, "How to check the number given by the user is even or odd?" we need to write the following steps:

Step:1 Start

Step:2 Define num, remainder

Step:3 Print "Enter Any Number"

Step:4 Accept value for num
Step:5 remainder = num % 2
Step:6 If remainder = 0 Then
 Print "Given Number is Even"
Step:7 If remainder = 1 Then
 Print "Given Number is Odd"
Step:8 Stop

Algorithm begins with 'Start' and ends with 'Stop' instructions. In the Step:2 we have declared two variables num and remainder. Step:3 will prompt to the user to input a number and Step:4 indicated the number inputted by the user is stored in the num variable. Step:5 will compute the remainder after dividing number by 2 and stores it in another variable called remainder. After computing remainder will check its value. If the value of the remainder is 0 then Step:6 will print the message that the given number is Even. Step:7 will check the if the value in the remainder variable is 1 then it will print the message that given number is odd.

Algorithm can be written in many ways. There are not fixed rules there, but it expresses the logic of how to solve a problem. Here some more examples are given to practice different kind of algorithms. It is possible that same problem can be solved using two or three different methods. In this case two or more different algorithms are also possible. For example, in the above problem, we know that when we divide the number by 2 possible remainder can be either 1 or 0. We know that those numbers which are not even are odd numbers. So, we can rewrite the logic discussed above as:

Step:1 Start
Step:2 Define num, remainder
Step:3 Print "Enter Any Number"
Step:4 Accept value for num
Step:5 remainder = num % 2
Step:6 If remainder = 0 Then
 Print "Given Number is Even"
 Else
 Print "Given Number is Odd"
Step:8 Stop

Both algorithms are correct, but in first algorithm computer needs to evaluate two conditions where as in second algorithm computer needs to check only one condition, we can say second algorithm is better or faster.

Some more examples of the algorithm are discussed below.

<p>Example:1 Algorithm to calculate Area:</p> <p><i>Step:1 Start</i> <i>Step:2 Define length, breadth, area</i> <i>Step:3 length=150</i> <i>Step:4 breadth=200</i> <i>Step:5 area=length * breadth</i> <i>Step:6 Print value of area</i> <i>Step:7 Stop</i></p>	<p>Example:2 Algorithm to find sum of two numbers</p> <p><i>Step:1 Start</i> <i>Step:2 Define Num1, Num2, Sum</i> <i>Step:3 Accept value of Num1</i> <i>Step:4 Accept value of Num2</i> <i>Step:5 Sum=Num1+Num2</i> <i>Step:6 Print value of Sum</i> <i>Step:7 Stop</i></p>
<p>Example:3 Algorithm to find greatest number from given two numbers</p> <p><i>Step:1 Start</i> <i>Step:2 Define Num1, Num2</i> <i>Step:3 Accept value of Num1</i> <i>Step:4 Accept value of Num2</i> <i>Step:5 If Num1 > Num2 go to Step 6</i> <i>Else go to Step 7</i> <i>Step:6 Print value of Num1</i> <i>Step:7 Print value of Num2</i> <i>Step:8 Stop</i></p>	<p>Example:4 Algorithm to check the Given year is a Leap year or not.</p> <p><i>Step:1 Start</i> <i>Step:2 Define year, remainder</i> <i>Step:3 Accept value of year</i> <i>Step:4 remainder = year % 4</i> <i>Step:5 If remainder=0 go to Step 6</i> <i>Else go to Step 7</i> <i>Step:6 Print "Year is a Leap Year"</i> <i>Step:7 Print "Year is not a Leap Year"</i> <i>Step:8 Stop</i></p>
<p>Example:5[A] Algorithm to find greatest number from given two numbers</p> <p><i>Step:1 Start</i> <i>Step:2 Define A, B, C, MAX</i> <i>Step:3 Print "Enter Three Numbers"</i> <i>Step:4 Accept value of A, B, C</i> <i>Step:5 MAX =A</i> <i>Step:6 If B> MAX then MAX=B</i> <i>Step:7 If C>MAX then MAX=C</i> <i>Step:8 Print MAX</i> <i>Step:9 Stop</i></p>	<p>Example:5[B] Algorithm to find greatest number from given two numbers</p> <p><i>Step:1 Start</i> <i>Step:2 Define A, B, C</i> <i>Step:3 Print "Enter Three Numbers"</i> <i>Step:4 Accept value of A, B, C</i> <i>Step:5 If A > B Then</i> <i>If A > C Then</i> <i>Print A</i> <i>Else</i> <i>Print C</i> <i>Else</i> <i>If B>C Then</i> <i>Print B</i> <i>Else</i> <i>Print C</i> <i>Step:6 Stop</i></p>
<p>Example:6 Algorithm to find factorial of given number:</p> <p><i>Step:1 Start</i> <i>Step:2 Define Fact, I, N</i></p>	<p>Example:7 Algorithm to find reverse of given number:</p> <p><i>Step:1 Start</i> <i>Step:2 Define Num, Remainder, RNum</i></p>

<p>Step:3 Print "Enter Any Number"</p> <p>Step:4 Accept value of N</p> <p>Step:5 Fact=1</p> <p>Step:6 I=1</p> <p>Step:7 Fact=Fact * I</p> <p>Step:8 If I < N then go to Step 7 Else go to Step 9</p> <p>Step:9 Print Fact</p> <p>Step:10 Stop</p>	<p>Step:3 Print "Enter Any Number"</p> <p>Step:4 Accept value of Num</p> <p>Step:5 RNum=0</p> <p>Step:6 Remainder = Num % 10</p> <p>Step:7 Num = Num / 10</p> <p>Step:8 RNum = Rnum * 10 + Remainder</p> <p>Step:9 If Num > 0 Then go to Step:6</p> <p>Step:10 Print RNum</p> <p>Step:11 Stop</p>
--	--

FLOWCHARTS

Pictorial representation of an algorithm is called Flowchart. To draw the flowchart, we have to know the algorithm, as well as certain symbols which are given in the following table.


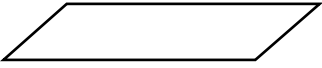

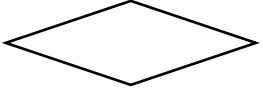


Sr. No.	Action	Symbol Name	Symbol used in Flowchart
1	Start / Stop (Terminals)	Oval	
2	Input / Output	Parallelogram	
3	Processing	Rectangle	
4	Conditional Statement	Dimond	
5	Connector	Circle	
6	Flow Line	Arrow	

Table-1.1 Flowchart Symbols

We have discussed the algorithm to check the given number is Even or Odd. The flowchart representation of the same problem can be described as below.

Flowchart of the problem 'Check the given number is Even or Odd' is shown in the Figure.3 given below. In the Flowchart terminal symbols are used for the Step Start and Stop. Defining variables and calculating remainder are processing steps, so that we have used rectangle symbol. Printing messages and accepting value from the

user are kind of Input / Output statements and to indicate it parallelogram is used. To evaluate the condition 'Is the value of the remainder variable is 0?', Diamond is used, which has two branches. Computer follows the appropriate branch depending upon condition is True or False.

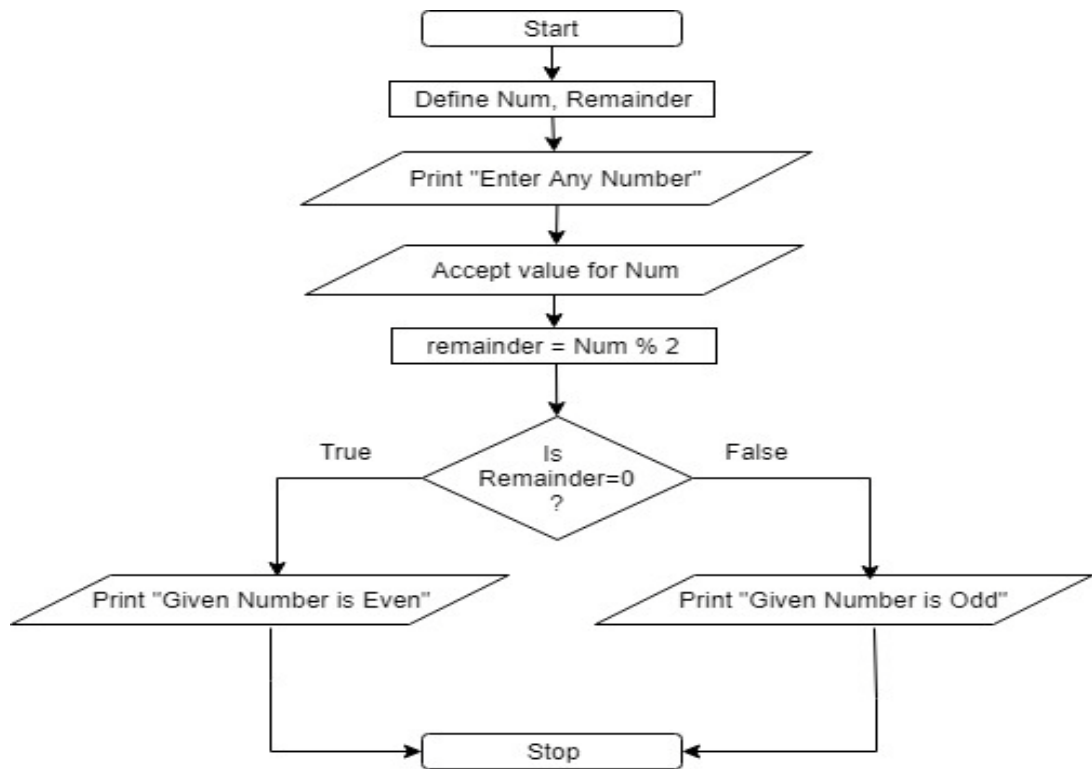


Figure-1.3Flowchart – Given number is Even or Odd

Connector is used if the flowchart is long and cannot be accommodated in a single page. How the connector is used is shown in the following Figure.4.

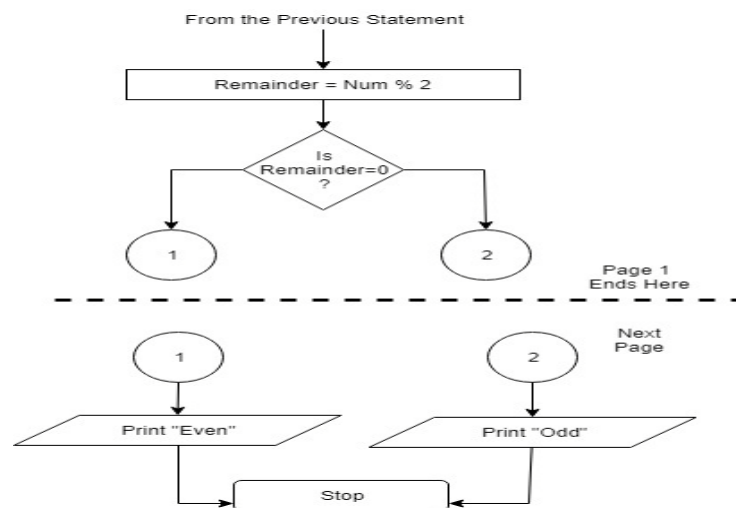
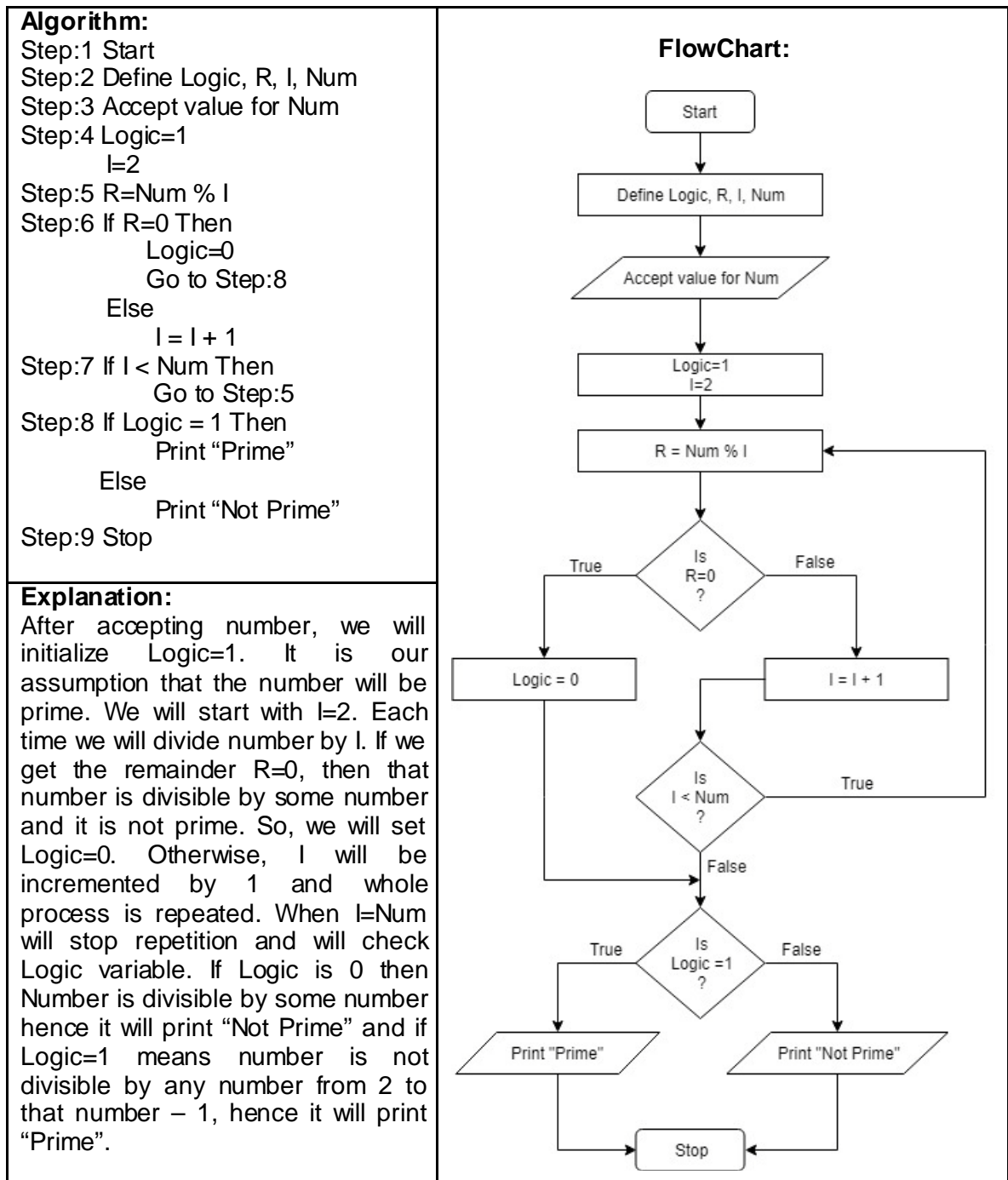


Figure-1.4Use of connectors in Flowchart

Another algorithm and a Flowchart of the problem to check whether the given number is prime or not is discussed below:



Exercise: 1.1 Write algorithms for following problems:

1. To find sum of all even numbers from 40 to 60.
2. To generate Fibonacci series up to N number. i.e. (1, 1, 2, 3, 5, 8, 11, N)

3. To check given number is Palindrome or not. (A number and its reverse are same then that number is Palindrome number. For example, 12321 is Palindrome but 1234 is not Palindrome).

Exercise: 1.2 Draw Flowchart for following problems:

4. To find factorial of a given number.
5. To find greatest number from given 3 numbers.
6. To check given number is Perfect number. (A number is Perfect if sum of all its factor is same as that number. For example, 6 (1+2+3=6) where 1,2,3 are factors of given number 6. Another Perfect number is:28 (1+2+4+7+14).

1.5 PROGRAMMING LANGUAGES

To communicate with the Computer, we need programming languages. It is specifically design to give instructions to the Computer system to perform I/O operations, computations and logical operations that control behavior of the system. Programming languages consist of syntax (refers to grammatical rules) and semantics (refers to the meaning of the vocabularies) which provides meaningful instruction to the system. Programming languages provide platform to developer to write the program, which expressed in the algorithm which can be executed on system. Depends on the enhancement and advances done in the programming language it can be classified by following generations.

1. Machine language
2. Assembly language
3. High-level language (3rd generation languages)
4. Very high-level languages (4th generation languages)

MACHINE LANGUAGE

Machine language is a language, which machine can understand directly. To understand the instruction written in this language, machine do not need any kind of translator. As we know machine is an electronic device and made by number of electronic components which can be in 2 states (charged means 1 or discharged means 0). So, it a language of only two symbols 1 and 0. This language is also known as binary language. Here the program has number of instructions, and each

instruction has unique binary pattern string. Because of each instruction directly written in machine language, translation is not needed. So, it is faster language than all other languages. It is also known as Low-level language.

It is difficult for the programmer to memorize all instructions of the system in the form of binary strings. Hence, it is difficult to learn machine language. Another drawback of the machine language is, it is machine dependent language. So, program written for one machine cannot be executed on another machine having different architecture.

ASSEMBLY LANGUAGE

Rather writing instruction into the string of binary as in machine language, Assembly language use mnemonic symbols such as to add two number instruction 'ADD' is used. Similarly, 'MOV', 'SUB', 'MUL' kind of mnemonic instruction (instead of Binary) make this programming language easier and more readable, than machine language. Here programmer will write the program using mnemonic codes. After writing the program all instructions will translated into machine language using assembler.

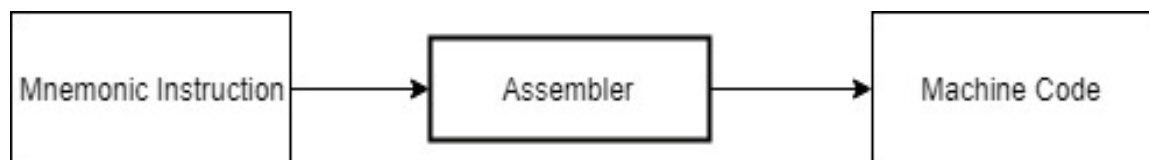


Figure-1.5 Assembly language and Assembler

Assembler is a simple translator, which takes one assembly instruction and convert it into one corresponding machine code(instruction). So, it not reducing the length of the program. Furthermore, assembly programs are also machine dependent (as Machine language).

HIGH-LEVEL LANGUAGES

Procedural and Object-Oriented languages are high-level languages. C, C++, Java, Visual Basic .NET, C#.NET are High-level languages. Basically, it is easier for the programmer to write the program in the high-level languages. High-level programs are more readable compare to assembly language code. To translate the instruction written in the high-level languages, compilers and interpreters are used. Compilers

and Interpreters translates the High-level instruction to the machine understandable code, before the execution starts.

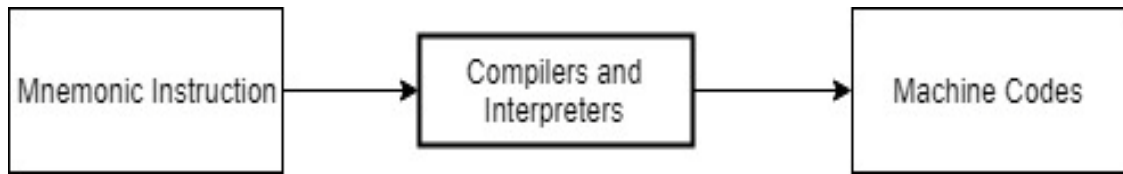


Figure-1.6 High-level code and Compiler / Interpreters

Compare to the assembler, compilers are doing its work in more efficient way. Compilers can take a single high-level instruction and can convert it into one or more instructions in the low-level language. So, compiler reduces number of instructions that programmer needs to write. Hence compiler makes high-level programs more compact. High-level languages are portable and platform independent. So, program written for a computer can be executed in other machines even they have different architecture or operating systems.

In the following figure we have tried to show how one line of high-level code is compiled and translated into 3 lines. In the C-language we have instruction 'I++', which will increment the value of variable 'I' by one. Now variables are declared in the random-access memory, and do not allow to change the value of any variable stored in RAM. It allows only two operations read and write. When programmer writes 'I++' instruction in the C-language, compilers generates 3 line of codes as shown in the figure. Computer will read the value of variable 'I' from the RAM and copy it to some register AX. The 1 will be added to the register AX. Finally, the incremented value of the AX register will be written back to RAM (i.e. variable I will be overwritten by incremented value). Thus, compiler save efforts of the higher-level language programmer, as well as allow programmers to do programming even if they do not have prior knowledge of the computer hardware or architecture.

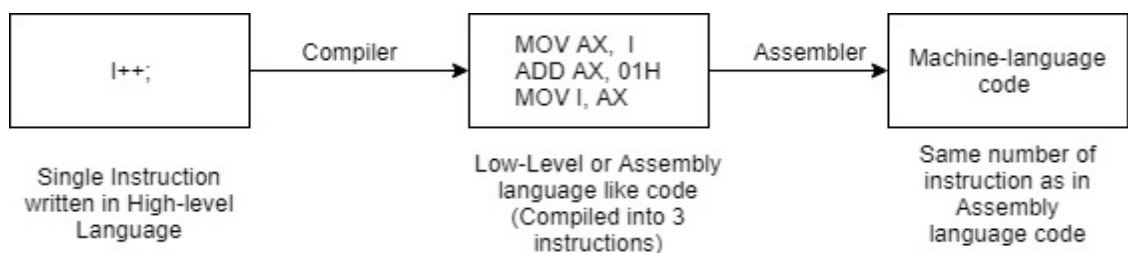


Figure-1.7 Compilation of high-level language code

It is clear that High-level languages reduces programming efforts by reducing number of lines, more readable and platform independent but it takes much time to translate it into machine code and these languages are slower compare to the Machine language and Assembly language.

4th Generation Language

4th Generation languages like SQL (Structured Query Language) is much simpler and easy to understand. It uses syntax nearer to English like language. For example, to fetch the name of the Employees from the Emp table having salary more than 25000, we can write “SELECT NAME FROM EMP WHERE SALARY > 25000”. To perform the same task into the procedural (3rd GL) requires number of instructions like declaration of number of variables, running of the loop for each record, if condition to verify salary > 25000 and so on.

COMPILER Vs. INTERPRETER

Compilers	Interpreters
Compiler reads the whole source code of the program and translate it into the object code. Once the whole program is translated then and then it starts execution.	Interpreters reads one line at a time, translate it into the object code and execute it.
If program has error in the source code, execution will not be started.	If program has error then instruction written before the line having error will be executed.
Takes more compilation time	Takes less compilation time.
Execution of the program will be faster as whole source code is translated in prior to the execution.	Execution of the program will be slower as after execution each line, interpreter fetch the next instruction, translate it and then execute it.
C, C++ like languages uses compiler.	Unix shell programming uses interpreter.

Table-1.2 Compiler Vs. Interpreter

Exercise 1.5

1. Differentiate Assembler and Compiler
2. Differentiate Compiler and Interpreter
3. List the advantages and disadvantages of Assembly language.
4. Explain High-level languages in brief.
5. _____ is a 4GL language.
6. 4GL are _____ than 3GL language. (Slower / Faster).
7. _____ is a translator which translate and execute instruction line by line.

1.6 LET US SUM UP

In this chapter we have learnt definition of computer system, data and information. We have seen that the computer system consists of hardware and software. We have also discussed application and system software. Then we have focused on programming concepts and how to build logic using algorithms and draw flowcharts. Finally, we have discussed various programming languages like machine language, assembly language, high-level languages and 4th GL languages. Finally, we have ended our discussion with how compiler and interpreters differently help in the execution of our program.

1.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Check Your Progress 1:

5. SQL
6. Slower
7. Interpreter

Unit 2: C-Language and Console I/O Operations

2

Unit Structure

- 2.1. Learning Objectives
- 2.2. Introduction to C-Programming Language
- 2.3. Printing white spaces
- 2.4. Console I/O Operations
- 2.5. Let's sum up
- 2.6. Check your Progress: Possible Answers

2.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- History and Features of C-Language
- Program structure of C-Languages
- Programming rules and Execution of C-Programs
- Printing white spaces in the program
- Formatted and Unformatted IO Functions

2.2 INTRODUCTION TO C-PROGRAMMING LANGUAGE

C-Language is most popular programming language developed and designed by Dennis Ritchie in 1972 at AT&T Bell Laboratory, USA. It is a general-purpose programming language can be used to develop both application programs and well as system programs. Several important features of the C-Language are taken from its successor B-Language which is developed by Ken Thomson in 1970 at Bell Laboratories USA, and BCPL (Basic Combined programming language) which is developed by Martin Richards in 1967 as shown in the Fig.1 given below.

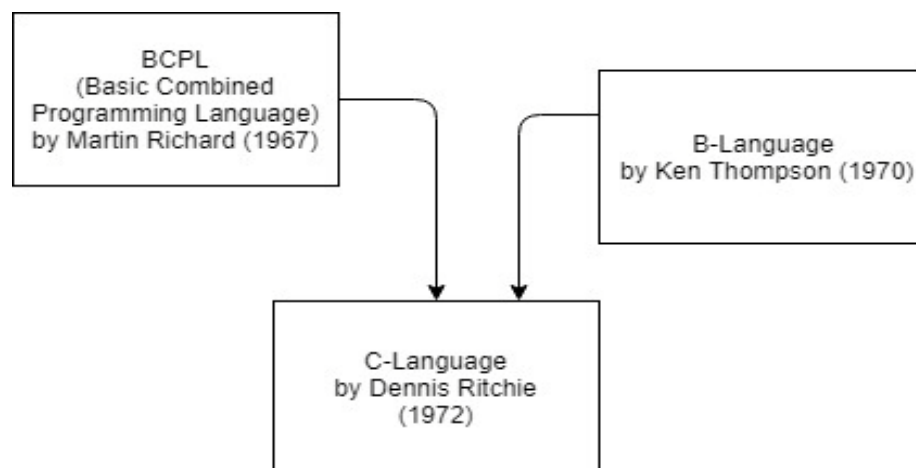


Figure-2.1 Development of C-Language

FEATURES OF C-LANGUAGE

- C-Language is general purpose programming language, can be used to develop application software as well as system software. Many utility programs like hardware drivers, compilers are developed in this language. UNIX operating system is also written in C-Language.

- C-Language can run on any operating systems like MS-DOS, Windows, UNIX, Linux and UBUNTU operating systems.
- C-Language is a procedure-oriented language. C-Programs are modular and allow user to create functions for specific task. Functions makes program more readable. It reduces complexity and increases reusability of the code.
- Development of C-Programs are faster. It is easy to write the program, debug, test, and maintain the program.
- C has character, integer, float, double data types. C also allows programmer to define user defined data type. So, C-Language is also called Structural programming language.
- C-language supports, if statement, if ... else statement, if ... else if.... else and switch...case kind of conditional statements. C-Language also supports While, For and Do...While loop.
- C-Language has powerful set of operators.

PROGRAM STRUCTURE OF C-LANGUAGE

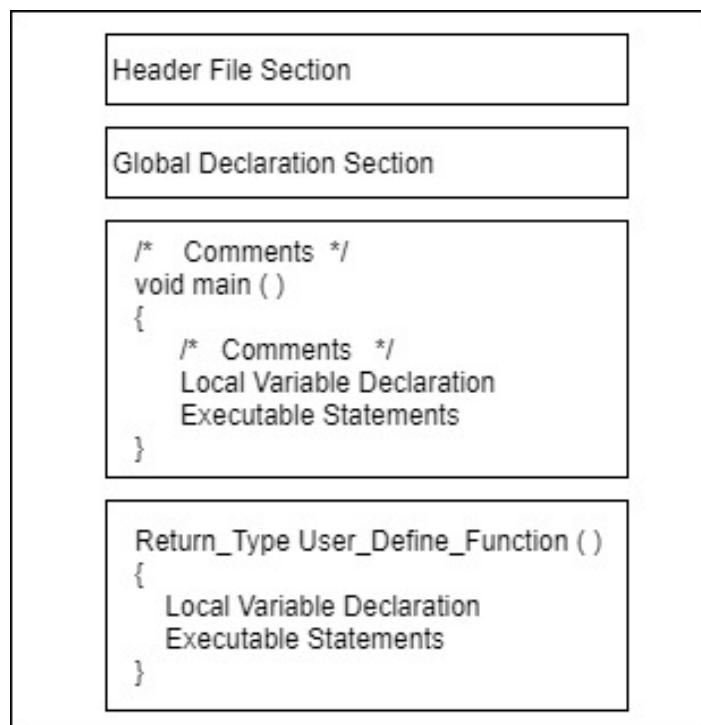


Figure-2.2 Program structure of C-Language

1. Header File Section: As we have discussed earlier C-Language supports functions. C-Language has 30 or more header files, which include more than 145 library functions. This section of the C-Program allows programmer to include

header files. Once the header file is included in the program, all library functions specified in that header file can be used in the program. Header files are stored with '.h' extension. For example, if you have included header file '#include <stdio.h>' (Standard Input Output Header File) then you can use 'printf ()' and 'scanf ()' functions. If you have included header file '#include <conio.h>' (Console Input Output Header File) then you can use 'clrscr ()' and 'getch ()' functions. More details about the header files and its function please refer Table:1.

- 2. Global Declaration Section:** Variables declared in this section are called global variables. Global variables are created in the memory when program starts its execution and remain into the memory unless the program is not terminated. Global variables can be accessible anywhere in the program.
- 3. Main Function:** In C-Programming language, N number of functions you can create, but the function from which you can start execution of the program should have function name 'main ()'. System will always start the execution of the program from the main function. Because of the main () function do not returns any value, we have mentioned 'void' return type for the main function. '{' indicate start of the main function and '}' is used to close main () function.
- 4. Local Variables:** Variable declared inside the function is called local variable. The life of the local variable is limited. When the function is called and it is loaded into the memory local variables are created and after the execution when the function is destroyed all the local variables will be deleted from the memory. The scope of the local variable is limited to that function only. You cannot access local variable of one function into other function. Unlike global variable (declared outside of function) local variable cannot be accessible throughout the program.
- 5. Executable Statements:** Executable statements are set of statements written to solve the specific problem. That includes IO statements, computational statements, conditional statements or looping statements.
- 6. Comments:** Comments are the statements that can be written anywhere in the C-Program. Comment statements are simply skipped by the compiler and it will not be translated into the machine language, means those statements are not be executed by the system. Comments are the statements, which written in the program to increase the readability of the program. There are two types of

comments: (1) Single line comment which is denoted by // and (2) Multiline comment which is denoted by /*..... */ For Example:

//This is single line comment

/* This is

Multiline

Comment */

7. **User Defined Function:** If you are not be able to locate proper library function for particular task, you can define your own function in the C-Language. Such functions are known as user defined function (UDF). You can create as many as UDFs into a program.

PROGRAMMING RULES:

1. A C-program can have multiple functions, but in each program must have 'main ()' function. System always starts the execution of the program by 'main ()' function.
2. Each statement of the C-Program ends with semicolon (;). However conditional statements, for and while loop and in the function definition, statements do not end with semicolon.
3. Generally, in the C-program all the statements should be written in the lower case. However uppercase strings can be used for symbolic constants.
4. The opening and closing braces should be balanced. i.e. number of opening braces and number of closing braces are same.
5. It is not necessary that each line has single statement in C-program. You can write more than one C-program statements in the same line. For example,

X = Y + Z;

P = 5 * X;

Can be written as,

X = Y + Z; P = 5 * X;

EXECUTION OF C-PROGRAM

A C-program must pass through number of phases like writing a program with editor, compiling and linking the program and finally loading and executing the program. In this section we will do the details discussion of each phase of the program.

- **Writing a C-program:** The C-Program can be written in the C-editor. The file has to be saved with '.C' extension. '.C' is the default extension. Some editors use '.CPP' as a default extension to the file. Make sure, .CPP is a C++ file, and editor use C++ compiler to compile the file at the compilation time. Because C++ compiler can also compile C-programs, used feel that C-programs can be stored as either '.C' or '.CPP' extension.
- **Compiling and Linking:** After writing the C-Program, we need to compile the C-Program by pressing (Alt + F9). Different C-Language editor provides different shortcut key for compilation option. At the time of the compilation is any syntax error is there then error message is displayed. User must have to solve the error and again needs to compile the program. If the program does not have any syntax error the compilation process will be completed with success message and it will generate object file having extension '.obj' and it will be stored in the secondary memory such as disk. After successful completion of compilation process linking process will be started, which will link the C-programs with the header files, included in the C-program. For example if we are using sqrt () function in the C-Program and we have included the header file '#include <math,h>' then the linking process will bring the code of sqrt () function to the object code file from math.h header file. As an output of the linking process '.EXE' file will be generated.
- **Execution of the program:** Once the compilation and linking process is completed successfully '.EXE' file is generated. One we press Ctrl + F9 key (Shortcut key can be different for different editor) a software loader is invoked and it will put the '.EXE' file into the memory for execution. Once the program is loaded into the main memory CPU of the system will start the execution of the program line by line.

FIRST C-PROGRAM

Open the C-language editor and type the following program:

Program 2.1 Printing Hello, World!!! On Console

```
#include<stdio.h>
void main ()
{
    /* This is my First C-Program */
    printf ("Hello, World!!!");
}
```

OUTPUT:

Hello, World!!!

Once the program is written into the editor, save the program with 'Hello.C', and compile the program with Alt+F9. If any error is there, correct the source code, eliminate the error and compile it again. After successful compilation press Ctrl+F9 to execute the program. In output we will get "Hello, World!" string printed on the Console.

Exercise: 2.1 Fill in the blanks:

1. _____ keys are used to compile the C-Program.
2. Compilation is a process which converts source code into _____ code.
3. _____ is used as single line comment.
4. Stdio.h stands for _____.
5. _____ variables are accessible throughout the program.
6. Scope of the _____ variable is limited to that function only.
7. From _____ function execution of the program starts.
8. _____ will put the compiled code into the memory for execution.
9. Functions used in the C-program, are brought into the object code by _____.
10. _____ are the statements added in to the C-Program to increase readability of the program and they will not be executed by the system.

2.3 PRINTING WHITE SPACES

	White space character	Symbol
Function printf() is a formatted output function, provides functionality to format the output in the desire format on the Console. In the next section of this chapter we will discuss all I/O functions in		

the greater details. White spaces such as space, tab, back space, new line etc. are needed to print the output in the desire format. Refer the following table which is listing for which white space character which symbols are used. Sr No.		
1	Back Space	\b
2	Horizontal tab	\t
3	Vertical tab	\v
4	New line	\n
5	Backslash	\\
6	Question mark	\?
7	Alert bell	\a
8	Carriage return	\r
9	Form feed	\f

Table-2.1 White space characters

In the following table we have discussed how to use white space symbols in the printf() statement and how the output will be displayed.

Sr. No.	printf() statements	Output
1	printf("Hello World!");	Hello World!
2	printf("Hello "); printf("World!");	Hello World!
3	printf("Hello\n "); printf("World!");	Hello World
4	printf("Hello"); printf("\nWorld!");	Hello World

5	printf("Hello\nWorld");	Hello World
6	printf("Hello\tWorld");	Hello World
7	printf("Hello\n\nWorld");	Hello\nWorld
8	printf("\a");	It will generate Beep sound as alert bell.

Table-2.2 White space in printf() function and its output

2.4 CONSOLE I/O OPERATIONS

As we have already discussed, programs are set of instruction, written in particular language which takes input from the user, process it and produce information as output. To take the data IN to the program we need console in, and to give information OUT to the user we need console out operation. In this section we will focus on console input / output operations.

When we design a C-Program, variables of the program can be initialized in two ways. Either we can initialize them with some fixed value which is shown in the program 2.2 or we can initialize them by taking the value from the user which is shown in the program 2.3

Program 2.1 Assigning static value to int variable and print it.

```
#include<stdio.h>
void main ()
{
    int x = 5;
    printf ("Value of X is: %d", x);
}
```

Output:

Value of X is:5

In this program we have declared one variable 'x' of type integer and we have initialized it with a fixed value 5. This type of assignment is known as static assignment or static initialization. printf () is a function, which used to print output on the screen. A printf () function is used to print any string, character, integer or floating values on the console screen, it known as console output function.

Program 2.2 Accepting integer value from the user and printing it to Console

```
#include<stdio.h>
void main ()
{
    int x;
```

```

printf ("Enter Any Number:");
scanf ("%d", &x);
printf ("Value of X is:%d", x);

}

```

Output:

```

Enter Any Number: 100
Value of X is:100

```

In this program we have declared a variable x same in the previous program. By using a printf () function we prompt to the user to 'Enter Any Number:'. After printf () statement we have return a scanf () statement. Here console wait for the user input. Suppose user enters 100, then scanf () statement will accept this value from the console and it will store it in the variable 'x'. Finally, in the last statement when we print the output 'Value of X is:100' on the screen. From this program we can say that printf () function is for giving output on the console and scanf () function is for taking input from the console. Both printf () and scanf () function are available in the stdio.h header file so that we must have to include header file stdio.h into our program if we are using printf() or scanf() functions in our program. Consider the following program which will accepts two numbers from the user and print the sum of it.

Program 2.3 Accepting two integer values and printing sum of it

```

#include <stdio.h>
void main ()
{
    int x, y, sum;
    printf ("Enter First Number:");
    scanf ("%d", &x);
    printf ("Enter Second Number:");
    scanf ("%d", &y);
    sum = x + y;
    printf ("Sum of %d and %d = %d", x, y, sum);
}

```

Output:

```

Enter First Number: 5
Enter Second Number: 7
Sum of 5 and 7 is = 12

```

TYPES OF I/O FUNCTIONS

There are many I/O functions are there in the C-Language, which can be classified in two different types:

1. Formatted functions
2. Unformatted functions

Formatted functions allow us, to format the input or output is to be formatted as per our requirements. We can make our input and output statements more readable, properly formatted and more user friendly with formatted I/O functions. In Unformatted I/O functions, formatting is not possible. Various unformatted I/O functions are explained below in more details.

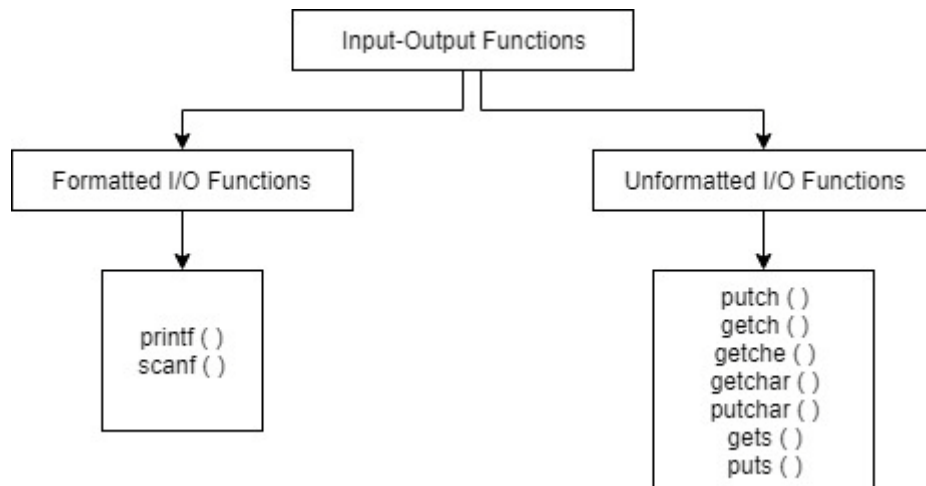


Figure-2.3 Formatted and Unformatted I/O Functions

1. Formatted I/O Functions:

Functions `printf()` and `scanf()` are known as formatted I/O functions. Both functions are available in the header file `stdio.h`. So, it is necessary to include header file `stdio.h` if program used these functions. `printf ()` functions is basically used to put output on the console, and `scanf()` function is used to take (read) input from the console. Function `printf()` and `scanf()` uses a format string to print different types of data which is shown in the following table.

Sr. No.	Type of data	Description	Format String
1	char	Single character	<code>%c</code>
2	int	Integer number	<code>%d</code> or <code>%i</code>
3	unsigned int	Positive Integers only	<code>%u</code>
4	long int	Long integer number	<code>%ld</code>
5	float	Floating point number	<code>%f</code>
6	double	Long floating-point number	<code>%lf</code>
7	string	Group of characters of Text	<code>%s</code>
8	octal number	Number represented in octal	<code>%o</code>
9	Hexa decimal	Lower case	<code>%hx</code>
10	Hexa decimal	Upper case	<code>%p</code>

In the program 2.1 to 2.3 we have used printf() and scanf() functions to print 'int' (integer) numbers. How the printf() and scanf() functions can be used to input or output characters, strings and floating point numbers are shown in the following programs.

Program 2.4 Printing floating point number with desire decimal points

```
#include<stdio.h>
void main ()
{
float pi;
printf ("Enter the value of PI:");
scanf ("%f", &pi);
printf ("Value of PI is: %f", pi);
printf ("\n Value of PI in 2 Decimal point is: %.2f", pi);
printf ("\n Value of PI in 3 Decimal point is: %.3f", pi);
}
```

Output:

```
Enter the value of PI: 3.14
Value of PI is: 3.140000
Value of PI in 2 Decimal point is: 3.14
Value of PI in 3 Decimal point is: 3.140
```

In the above program variable 'pi' is declared of type float. We prompt to the user to input the value of pi, by a printf() statement, and wait for the user input by scanf() statement. Here we are reading a floating-point value from the console so that format sting we have used is "%f" and variable we have taken is float. Suppose user has entered 3.14 value on the console. 3 consecutive printf() statements will print the value of pi variable. If we use "%f", then it will print full floating-point number that is 3.140000 (6 decimal point). But we want to print the floating-point number with 2 or 3 decimal points then we can use ".2f" or ".3f" format string for floating-point numbers. More discussion about different datatypes, we will do in the chapter:3.

Program 2.5 Input Output statements for character and string

```
#include<stdio.h>
void main ()
{
char ch, name[10] ;
printf ("Enter any character:");
scanf ("%c", &ch);
printf ("Enter your name:");
scanf ("%c", name);
}
```

```

    printf ("Character you have entered is: %c", ch);
    printf ("\nYour Name is: %s", name);
}

```

Output:

```

Enter any character: A
Enter your name: abcde
Character you have entered is: A
Your Name is: abcde

```

In the above program variable 'ch' is character variable and 'name' is string (group of characters having capacity to maximum 10 characters). See carefully both scanf() statements in the character we have mention '&ch' while in the case of string we have used only 'name'. & is not there in the scanf() statement which takes string.

2. Unformatted I/O Functions:

In this category some functions start with word 'put'. Those function are output function which is used to put character or string on the console. Similarly, some functions start with word 'get'. Those functions are Input functions and used to take character of string input from the console.

Functions getchar() and putchar(): Function getchar() used to take (input) a character from the console. Here function allows user to enter string, but only first letter of the string will be stored in character variable and rest of the characters will be ignored. Putchar() function is used to print (output) character on the console. Both functions are in stdio.h header file. Following program shows the use of both the functions.

Program 2.5 Use of getchar() and putchar() functions

```

#include<stdio.h>
void main()
{
    char ch;
    printf("Enter a character:");
    ch=getchar();
    printf("You have Entered:");
    putchar(ch);
}

```

Output:

```

Enter a character: ABCD
You have Entered: A

```

Functions getch(), getche() and putch(): Function getch() is used to take a character from the user. It allows user to input only one character. User is not allowed to enter second character (string). The inputted character will not be displayed on the console. This function can be used to accept password, where the text entered by the user is not displayed on the console. Both functions are available in the conio.h header file.

Program 2.6 Use of getch() and putch() functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch;
    printf("Enter a character:");
    ch=getch();
    printf("You have Entered:");
    putch(ch);
}
```

Output:

Enter a character:

You have Entered: A

Similarly, function getche() takes only one character from the user and that character will be displayed on the console.

Functions gets() and puts(): Function gets() and puts() is used to accept string from the console and print a string on the console. We can also accept string by a formatted function scanf() with format string "%s", but it can store one word only, while gets() function can store sentence (more than one words). The same thing we have tried to show in the program given below.

Program 2.7 Use of gets and puts functions

```
#include<stdio.h>
void main()
{
    char x[20], y[20];
    printf("Enter any String:");
    scanf("%s",x);
    fflush(stdin);
    printf("Enter same string again:");
    gets(y);
}
```

```

printf("\nString is X is:%s",x);
printf("\nString in Y is:");
puts(y);
}

```

Output:

Enter any String:BAOU University

Enter same string again:BAOU University

String is X is:BAOU

String in Y is:BAOU University

Exercise: 2.2 Fill in the blanks:

1. To print a string with formatted I/O function printf(), _____ format string is used.
2. _____ unformatted I/O function takes a single character and it will not be displayed on the console.
3. To use getch() function _____ header file has to be included in the program.
4. To take more than one word as a string _____ function is used.
5. _____ function will take only a single character and character will be displayed on the string.

Exercise: 2.3 Find the output of following statements:

1. `printf(5+"Good Morning\n");`
2. `printf("%c", "Programming Language"[3]);`
3. `#include<stdio.h>`

```

{
    char ch='A';
    printf("The character is:%c",ch);
    printf("\nAnd its ASCII value is:%d",ch);
}

```

4. `#include<stdio.h>`

```

{
    char ch=97;
    printf("The character is:%c",ch);
    printf("\nAnd its ASCII value is:%d",ch);
}

```

2.5 LET US SUM UP

In this chapter we have learnt history of the C-Language. We have also learnt C-Language programming rules and basic program structure of the C-Language. Then we have focused on how to write, compile and execute C-Programs. Finally, we have ended our discussion with formatted and unformatted I/O functions, how to use them in the program with different examples. We hope students will have now sufficient knowledge make C-Programs and use different I/O functions.

2.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 2.1

- 1 Alt + F9
- 2 Object
- 3 //
- 4 Standard Input Output Header File
- 5 Global
- 6 local
- 7 main()
- 8 Loader
- 9 linker
- 10 Comments

Exercise: 2.2

- 1 %s
- 2 getch()
- 3 conio.h
- 4 gets()
- 5 getche()

Exercise: 2.3

- 1 Morning
- 2g
- 3 The character is: A
And its ASCII Value is:65
- 4 The character is: a
And its ASCII Value is:97

Unit 3: Keywords, Variables, Datatypes and Operators

3

Unit Structure

- 3.1. Learning Objectives
- 3.2. Keywords
- 3.3. Variables
- 3.4. Constants
- 3.5. Datatypes
- 3.6. Operators
- 3.7. Type conversion and casting
- 3.8. Let's sum up
- 3.9. Check your Progress: Possible Answers

3.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand keywords
- Declare and use variables and constants
- Understand different datatypes their size and range
- Use of various operators in the expression

3.2 KEYWORDS

Keywords are reserved words for the compiler. Keywords are those words which are using to instruct the computer. These words have specific meaning, and cannot be used of other purposes. There are 32 keywords are there in the C-Language which are shown in the table given below.

auto	break	case	char
const	continue	default	do
double	else	extern	float
for	float	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	While

Table-3.1 Keywords of C-Language

3.3 VARIABLES

Variables are used to store values. When variables are declared at the time of program execution, system reserves certain amount of space in the memory. Variables are data names which can store the data values, in the memory and its value can be changes at any time. Variable can be declared with the help of datatype. When we assign the value to the variable is called initializing the variable. Once we initialize the variable in the program, its value may change at different times while the program is in the running mode. Following program demonstrate how the different types of variables can be declared in initialized in the C-program.

Program 3.1 Declaring variable and initializing it.

```
#include<stdio.h>
void main()
{
    int age;//Declaring 3 variables age, height and gender
    float height;
    char gender;
    age=21;// Initializing 3 variables with values 21, 5.7 and 'M'
    height=5.7;
    gender='M';
}
```

In the above program we have declared 3 variables age of type integer, height of type float and gender to type character and then we have initialized age by value 21, height by value 5.7 and gender by value 'M'. The following program shows how can we change the value of the variable.

Program 3.2 Over writing the value of the variable.

```
#include<stdio.h>
void main()
{
    int age;
    age=21;
    printf("Age is:%d", age);
    age=22;
    printf("\nNow Age is:%d", age);
}
```

OUTPUT:

Age is:21

Now Age is:22

In the program 3.2 when the system is executing first printf() statement at that time the value in the age variable is 21. After executing this statement, the value of the

age is changed to 22. So, the next printf() statement will print the value of age 22. That means, whenever we are assigning a new value to the variable, older value will be removed or flushed out and new value will be stored in the variable. This is also known as variable overwritten. Variable allows programmer to change the value at any time, so that is called variable (whose value can vary time to time).

RULES FOR DEFINING VARIABLES

- A variable must start with alphabets or underscore (_). It should not be start with numeric character.
- Variable name should not be a keyword.
- Length of the variable is depending on the compiler but generally it should not be more than 31 characters.
- Uppercase and lowercase letters are allowed in the variable name but C-language is case sensitive language. Therefore, variable names radius, Radius, or RADIUS will be treated as separate variables.
- Space is not allowed in the variable name.
- Special symbols except underscore (_) is not allowed in the variable name.

3.4CONSTANTS

Unlike variable constants do not allow user to change the value. The value of the Constant will remain fixed and cannot be changed during program execution. Constants are not reserving the space into the main memory, but at the time of compilation physical value is replaced by its name. So, during the execution, CPU do not have fetch the value of the constant into the memory, constants are evaluated faster than variable. In the case of variable CPU has to fetch the value of the variable from the memory at the time of evaluating the expression. So, it takes some time to read the memory and hence variables are slower than constants. The following program shows how to declare and use of a constant in the program.

Program 3.3 Using constants in the program.

```
#include<stdio.h>
#define MAX 10
void main()
{
    printf("The value of the MAX is:%d", MAX);
}
```

OUTPUT:

The Value of the MAX is:10

Constants are initialized at the time of declaration, and then the value of the constant remain unchanged during the program. Once constant is declared and initialized, we cannot change the value of it. Usually constants are written in the upper case. Infact this just convention and not the rule. Constants can be integer or real number, character or string type. In the above program we have declare MAX variable using pre-processor directive. Another way to define the constant variable is shown in the program given below:

Program 3.4 Using real number constant in the program.

```
#include<stdio.h>
const float PI=3.14;
void main()
{
    float radius, area;
    printf("\nEnter Radius of the Circle:");
    scanf("%f", &radius);
    area=PI*radius*radius;
    printf("\n Area of the Circle is:%.2f",area);
}
```

In the above program we have declared a real number constant PI using keyword 'const' and initialized it with value 3.14. We have taken two variables radius and area from which radius we are accepting from the user, calculating area and printing it on the screen.

3.5 DATATYPES

Data types are used in defining the variables. For example, if we write 'int x;' statement then 'int' is a data type and 'x' is a variable. Here int data type is indicating that the variable 'x' is used to store integer values. So, the datatype is used to indicate what type of data we are going to store in the variable. We have already discussed that, variable reserves space in the memory to store data values, data type is used to specify the how much space has to be given to that variable. For example, in the above declarative statement variable 'x' will reserves 2 Bytes of space in the memory.

If we declare 'char ch;' then in this declarative statement 'ch' is the variable of type 'char'. Data type 'char' indicate 1 Byte, hence variable ch will occupies 1 Bytes (=8

bits) of space in the memory. That means with this variable we can represent $2^8 = 256$ different symbols. If we count all the symbols of our keyboard that it is not exceeded by 256. So, 1 Byte of memory is sufficient to store one symbol (character) of our keyboard. We can also store the number in the variable 'ch'. In the case of number 256 is further divided by 2 to store the positive and negative values, furthermore we need one permutation to store or represent 0. Therefore, in the case of numeric values the range of the character variable is -128 to 127.

Do not confuse with character and numeric data types. In the memory everything is stored as string of binaries. Because of binary can be computed of the numbers every character has unique number called ASCII (American Standard Code for Information Interchange) value.

As we have discussed earlier in the case 'int x;' declarative statement x occupies 2 Bytes (=16 bits) of data in the memory. By using 16 bits we can represent $2^{16} = 65,536$ different numbers. If we divide all numbers in two categories positive and negative, plus if we keep one permutation to represent number 0, then range of the integer data type is -32,768 to 32,767. That means, variable 'x' can store maximum 32,767 and minimum -32,768 numbers.

Different data types, their size, range and format strings are given in the following table.

Sr. No	Data Type	Size (Bytes)	Range	Format String
1	char	1	-128 to 127	%c
2	unsigned char	1	0 to 255	%c
3	int or short	2	-32,768 to 32,767	%d or %i
4	unsigned int	2	0 to 65,535	%u
5	long	4	-2,147,483,648 to 2,147,483,647	%ld
6	unsigned long	4	0 to 4,294,967,295	%lu
7	float	4	3.4e-38 to 3.4e+38	%f or %g
8	double	8	1.7e-308 to 1.7e+308	%lf
9	long double	10	3.4e-4932 to 1.1e+4932	%lf

Table-3.2 Data types, their size, range and format strings

Difference between int and unsigned int:

	signed int	unsigned int
Size	2 Bytes	2 Bytes
Range	-32,768 to 32,767	0 to 65,535
Format string	%d or %i	%u
Example:	int i; i=32767;	unsigned int i; i=65535;

Difference short int and long int:

	short int	long int
Size	2 Bytes	4 Bytes
Range	-32,768 to 32,767	-2,147,483,648 to 2,147,483,647
Format string	%d or %i	%ld
Example:	int i; short int j; i=32767; j=32767;	long int i; i=350000;

Data types are listed in the above tables are basic or primitive data type. They are readily available in the C-Language, so some authors are also called then built-in data types. Arrays, Pointers and Functions (which is discussed in the next chapters) are called derived data types. They are extending the capabilities of primitive data types. While by using structure, unions, typedef or enumeration user can create new data types so they are called User Defined Datatypes. Classification of the different data types are shown in the following figure 3.1.

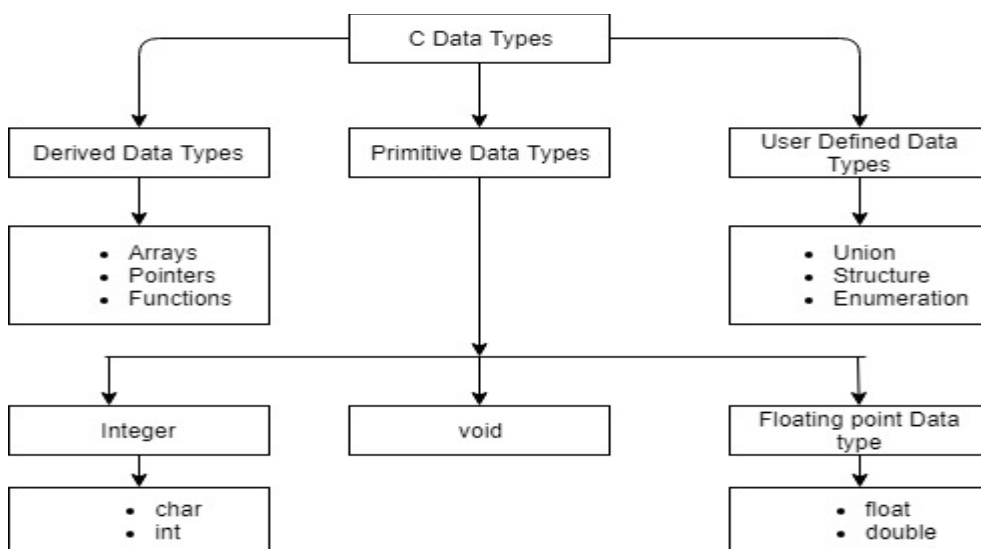


Figure-3.1Category wise C-Language Data Types

Exercise: 3.1 Fill in the blanks:

1. Format string for unsigned int variable is _____.
2. Format string for long int variable is _____.
3. Range of short int variable is from _____ to _____.
4. Range of unsigned int variable is from _____ to _____.
5. Range of unsigned short int variable is from _____ to _____.

3.6 OPERATORS

Operators are the special symbols which act upon data. For example, consider the following expression:

```
X= 2+3*5-7%2;
```

The expression given above has number of operators. +, *, -, % and = are called operators, while 2, 3, 5, 7 and X are called operands. Based on how many operands are used by the operators, they can be classified in three different categories:

UNARY OPERATORS

Unary operators are those who can take only one operand as an input. For example, negation, increment, decrement and sizeof operators takes single operator as an input, so they are unary operators. Consider the following example:

Program 3.5 Example of sizeof operator.

```
#include<stdio.h>
void main()
{
    printf("%d", sizeof(long int));
}
```

OUTPUT:

4

In the above example we have taken sizeof operator to check the size of long integer variable. As we know the size of long integer is 4 Bytes, we will get 4 as output. Instead of 'long int'. instead of 'long int' you can pass 'int', 'char' or 'float' and sizeof operator will gives you the size occupies by the variable of that data type. Because of in the 'sizeof' operator at a time we can pass only one data type it is called unary operators.

Program 3.6 Example of increment and decrement operators.


```

#include<stdio.h>
void main()
{
    int x=5;
    printf("\nX is: %d", x);
    x++;
    printf("\nAfter Increment X is:%d",x);
    x--;
    printf("\nAfter Decrement X is:%d",x);
}

```

OUTPUT:

X is:5

After Increment X is:6

After Decrement X is:5

In the above example, we have declared a variable 'x' of type integer and initialized it with value 5. When we are printing the value of x, it will print 'X is:5'. In the next statement when 'x++' is executed then the value of X will be increased by 1 and X will become 6. So in the next printf() statement it will print '**After Increment X is:6**'. Similarly, X--; statement will decrease the value of variable X by 1. So, the value of X which 6 will be decrease and will be turn to 5 again. So last printf() statement will print: '**After Decrement X is:5**'.

Program 3.6 Example of pre-increment operators.

```

#include<stdio.h>
void main()
{
    int x,y;
    x=5;
    y=++x;
    printf("\nValue of X:%d\nValue of Y:%d",x,y);
}

```

OUTPUT:

Value of X:6

Value of Y:6

Program 3.7 Example of post-increment operators.

```

#include<stdio.h>
void main()
{
    int x,y;
    x=5;
    y=x++;
    printf("\nValue of X:%d\nValue of Y:%d",x,y);
}

```

OUTPUT:

Value of X:6

Value of Y:5

If we are comparing above 2 programs then there is a one small difference is there. In program 3.6 we have written `y=++x;` instruction. Here `++` operator is called pre-increment operator. In this case value of `x` will be incremented by 1 and then it will be assigned to variable `y`. So, variables `x` and `y` both have same incremented value that is 6. In program 3.7 we have written `y=x++;` instruction. Here `++` operator is called post-increment operator. In this case variable will incremented later and value of `x` variable that is 5 is assigned to the variable `y`. So, in this case variable `y` will be 5 and then `x` will be increased by 1. Therefore, it will print value of `x` is 6 and `y` is 5. The same thing is also applicable to decrement operators.

BINARY OPERATORS

Binary operators are those operators which takes two operands as input. For example, we can write `5*7` or `5+7`. Here multiplication (`*`), Addition (`+`) always need at least two operands, so they are called binary operators. Consider the following program which focus on division and modulo binary operators:

Program 3.8 Example of division and modulo operators.

```
#include<stdio.h>
void main()
{
    int d,r;
    d=38/5;
    r=38%5;
    printf("Division is:%d", d);
    printf("\nRemainder is:%d", r);
}
```

OUTPUT:

Division is:7

Remainder is: 3

When we divide `38/5` and if the answer is stored in the integer variable (`d` in our case) then `d` will be 7 (as $7 * 5=35$). When we write `38%5` then `%` (Mod or Modulo) operator will gives remainder that is ($38 - 35 =3$).

TERNARY OPERATOR

The Ternary operator is an operator that takes 3 arguments. Conditional operator (?:) is a ternary operator. It is basically used to evaluate condition. Consider the following programs in which we have used a conditional operator to find greater number from given 2 numbers.

Program 3.9 Example of conditional operators-1.

```
#include<stdio.h>
void main()
{
    int n1,n2,max;
    printf("Enter First Number:");
    scanf("%d", &n1);
    printf("Enter Second Number:");
    scanf("%d", &n2);
    max=(n1>n2 ) ?n1 : n2;
    printf("Greater Number is:%d",max);
}
```

OUTPUT:

```
Enter First Number:5
Enter Second Number:7
Greater Number is:7
```

In the statement 'max=(n1>n2)?n1:n2;' first part that is n1>n2 is a conational part. Variable n1 we have written in the True part. Conditional part and True part are separated by '?'. Variable n2 is written in the False part. True and False parts are separated by ':'. Here system is checking the condition that is: 'Is n1 > n2?', if TRUE then True part will be executed and value of n1 variable will be copied to max variable. But if the condition is FALSE then False part will be executed and value of n2 variable will be copied to max variable. In short in the max variable greater number from n1 or n2 will be copied. Consider the next program in which we are finding the greatest number from given 3 numbers.

Program 3.10 Example of conditional operators-2.

```
#include<stdio.h>
void main()
{
    int n1,n2,n3,max;
    printf("Enter Any 3 Numbers:\n");
    scanf("%d%d%d", &n1,&n2,&n3);
    max=(n1>n2) ? (n1>n3)?n1:n3 : (n2>n3)?n2:n3;
    printf("Greatest Number is:%d", max);
}
```

```

}
OUTPUT:
Enter Any 3 Numbers:
5
9
7
Greatest Number is:9

```

Here in this program we are evaluating condition $n1 > n2$. If TRUE then $n1$ is bigger than $n2$ and now competition is between $n1$ and $n3$. Therefore, in the True part we have nested conditional statement that is $n1 > n3$ if again TRUE then $n1$ is the greatest number else $n3$ is the greatest number. If first condition $n1 > n2$ is FALSE then $n2$ is bigger and in the False part we have nested condition operator with $n2 > n3$. If this condition is TRUE then $n2$ is the greatest otherwise $n3$ is the greatest number.

Now, based on the behaviour of the operator it can be further classified in the following categories.

TYPES OF OPERATORS

Sr. No.	Type of Operator	Symbolic Representation
1	Increment/ Decrement operators	++, --
2	Size of operator	sizeof
3	Arithmetic operators	+, -, *, / and %
4	Shift operators	<<, >>
5	Relational operators	>, >=, <, <=, ==, !=
6	Bitwise operators	&, ^, and
7	Logical operators	&&, , and !
8	Conditional operators	?:
9	Assignment operators	=, +=, -=, *=, /=, %=
10	Comma operator	,

Table-3.3Types of operators

We have already discussed Increment / Decrement, sizeof, and Arithmetic operators. Now we will discuss all other operators as shown in the table given above.

Relational Operators:

Relational operators are basically used to compare two values or operands. After comparing they will return Boolean value either TRUE or FALSE (1 or 0). There are six relational operators are there as shown in the following table.

Operator	Name of the operator	Example	Return Value
>	Greater Than	5 > 3 3 > 5 5 > 5	1 0 0
>=	Greater or Equal	5 >= 3 3 >= 5 5 >= 5	1 0 1
<	Less Than	5 < 3 3 < 5 5 < 5	0 1 0
<=	Less or Equal	5 <= 3 3 <= 5 5 <= 5	0 1 1
==	Equal	5 == 3 5 == 5	0 1
!=	Not Equal	5 != 3 5 != 5	1 0

Table-3.4 Relational operators, its Example and Return value

Shift operators:

Shift operators are used to perform binary shift operations on data. There are two binary shift operators are there Left shift (<<) and right shift (>>). In the following program we have tried to explain shift operators.

Program 3.11 Example of shift operators.

```
#include<stdio.h>
void main()
{
    char x, y;
    x=23 << 1;
    y=23 >> 2;
    printf("X is:%d\nY is:%d",x,y);
}
```

OUTPUT:

X is: 46

Y is: 5

In the above program `x=23<<1` statement is written. Character data type use 1Byte (8 bits). Binary of 23 in 1 Byte is:0001 0111 if we shift all bits one time then binary will become 0010 1110 (make sure left most bit will be deleted, right most side '0' is appended and all bits are shifted from right to left by 1 position) is a binary of 46. Similarly, in the statement `y=23>>2`; binary of 23 that is: 0001 0111 (2 bits 00 will be

added at left side, and 2 bits '11' from the right side will be removed and all bits are shifted twice from left to right) will make 0000 0101 is a binary of 5.

Bitwise Operators:

There are 3 bitwise operators are there in C-Language. AND (&), OR (|) and XOR (^) consider the following truth tables:

A	B	A & B (A and B)
0	0	0
0	1	0
1	0	0
1	1	1
Truth Table of AND		

A	B	A B (A or B)
0	0	0
0	1	1
1	0	1
1	1	1
Truth Table of OR		

A	B	A ^ B (A xor B)
0	0	0
0	1	1
1	0	1
1	1	0
Truth Table of XOR		

Table-3.5 Truth table for AND, OR and XOR

By looking to the table, we can say that AND returns 1 if A and B both inputs are 1. OR returns 1 if either A or B (any one input) is 1, and XOR returns 1 if A and B inputs are not similar. Consider the following program which will gives you the idea of how to use truth table.

Program 3.12 Example of bitwise operators.

```
#include <stdio.h>
void main()
{
    char x = 23; // 0001 0111
    char y = 43; // 0010 1011
    char and, or, xor;
    and = x & y; // 0000 0011 (3)
    or = x | y; // 0011 1111 (63)
    xor = x ^ y; // 0011 1100 (60)
    printf("Variable AND is:%d, OR is: %d, and XOR is:%d", and, or, xor);
}
```

OUTPUT:

Variable AND is:3, OR is: 63, and XOR is:60

In the above program 3 variables and, or and xor are computer by bitwise &, | and ^ operators on x is 23 and y is 43.

Logical Operators:

There are 3 logical operators are there in C-Language. Logical AND (&&), Logical OR (||) and Logical NOT (!). Usually logical operators are used between two or more

conditions. Consider the following table, in which we have mention different conditions and its outputs in another column. Logical AND (&&) will return 1 if both the conditions are TRUE, Logical OR (||) will return 1 if any one condition is TRUE. In rest of the cases both operators will return 0. NOT operator will simply return the inverted output. That mean it will return 0 if the condition is TRUE and return 1 if the condition is FALSE.

Sr. No	Condition	Output
1	printf(“%d”, 35 >45);	0
2	printf(“%d”, ! (35 >45));	1
3	printf(“%d”, 45 >35)	1
4	printf(“%d”, ! (45 >35));	0
5	printf(“%d”, (35 >45) && (45 < 55));	0
6	printf(“%d”, (35 < 45) && (45 < 55));	1
7	printf(“%d”, (35 > 45) (45 < 55));	1
8	printf(“%d”, (35 > 45) (45 > 55));	0

Assignment operators:

Normally operator is ‘=’ is used to assign the value to any variable, so it is called assignment operator. Other assignment operators are +=, *=, -=, and %=. Operator += is nothing but it is a combination of addition and assignment. For example, if we write expression X+=5, then the value of the variable X is increased by 5 or we can say it is short hand instruction of X = X + 5. Consider the following program in which we have used *= operator with variable X which X*=5 a short instruction of (X = X * 5).

Program 3.13 Example of assignment operator *.=.

```
#include<stdio.h>
void main()
{
    int x = 20;
    x *= 5; // (x=x*5 or x = 20 * 5 )
    printf(“Now value of x is: %d”, x);
}
```

OUTPUT:

Now value of x is: 100

PRECEDENCE OF THE OPERATORS

Precedence of the operators are important while we are using more than one operator in the expression. For example, if we have written a statement ‘int x = 5 + 7 * 2 – 9 / 3 % 2’ then different user predicts different values for x. Some user will do

addition first, some will do multiplication, some will do modulo operation first and finally every time we will get different values for variable 'x'. So, it is important to know how the computer system will evaluate the expression. Consider the following table in which we have listed all the operators in the sequence from higher priority to lower priority. An expression can have number of different operators and it is also possible that they belong to different category. For example, consider the following expression they have Arithmetic, Relational, Logical and Assignment operators written in a single expression, but it is very easy to calculate the output with the help of the table given below:

```
int x = 2 + 3 * 5 % 2 < 5 + 7 - 9 % ( 3 + 1 ) && 7 - 11 % ( 2 + 1 ) >= 5 * 7 % 2;
```

Sr. No.	Operators	Operations	Priority
1	()	Parenthesis or Function Call	1 st
2	[]	Square brackets or Array expressions	
3	- >	Pointer to Structure operator	
4	.	Dot operator	
5	+	Unary plus	2 nd
6	-	Unary minus or Negation	
7	++	Increment	
8	--	Decrement	
9	!	Not operator	
10	~	Ones Complement	
11	*	Pointer operator	
12	&	Address operator	
13	sizeof	Size of operator	
14	*	Multiplication	3 rd
15	/	Division	
16	%	Modulo	
17	+	Addition	4 th
18	-	Subtraction	
19	<<	Left shift	5 th
20	>>	Right shift	
21	<	Less than	6 th
22	<=	Less than or equal to	
23	>	Greater than	
24	>=	Greater than or equal to	
25	==	Equal to	7 th
26	!=	Not equal to	
27	&	Bitwise AND	8 th

Sr. No.	Operators	Operations	Priority
28	^	Bitwise XOR	9 th
29		Bitwise OR	10 th
30	&&	Logical AND	11 th
31		Logical OR	12 th
32	?:	Conditional operator	13 th
33	=, +=, -=, *=, %=, &=, ^=, =, <<=, >>==	All assignment operators	14 th
34	,	Comma operator	15 th

3.7TYPE CONVERSION AND TYPE CASTING

When the expression is written in which different types of variables are used then how can we predict the final output (the value return by an expression) will be in which data type? For that we have to understand type conversion.

For example, consider the following program which will have 4 variables, and from which 3 variables are at the right-hand side of the = operator.

```
#include<stdio.h>
void main()
{
    char x=4;
    int y =25;
    float z =5.0;
    float ans;
    ans = z / (y * x);
    printf("\nAnswer is : %f", ans);
}
```

OUTPUT:

Answer is: 0.050000

So, answer is computed in the float. That is just because of from the 3 variables at right-hand side the largest variable is z (because it is float and it occupies 4 Bytes, while x is character and y is integer which occupies 1 and 2 Bytes), so the answer is computer in float. This is called type conversion. Now consider following program:

```
#include<stdio.h>
void main()
{
    int x=7, y=3;
    float ans;
    ans=x/y;
    printf("Answer is: %f", ans);
}
```

OUTPUT:

Answer is: 2.000000

It is surprising that that instead of output 2.333333 it shows output is 2.000000. The rule is same. At the right-hand side of the = operator of the expression has 2 variables and both are integer. So, if we find greater variable from them, we get integer only, and final answer will be in the integer which will remove all 3s after decimal points. If we want to get the exact answer then from 2 variables x and y at least one has to be float. In the next program we have provided solution for this problem:

```
#include<stdio.h>
void main()
{
    int x=7, y=3;
    float ans;
    ans=(float)x/y;
    printf("Answer is: %f", ans);
}
```

OUTPUT:

Answer is: 2.333333

When we have written (float)x instead of x, then for this statement only system is considering variable x is of type float. Now at the right-hand side from 2 variables x is float and y is integer. Bigger variable is float and so the answer is converted into float. ***The process of converting data type of any variable at the time of expression execution is called type casting.***

Exercise: 3.2 Fill in the blanks:

1. && is _____ operator.
2. To find the size, occupy by a variable in the memory, _____ operator is used.
3. Operator = is used for _____ and == is used for _____.
4. In the expression, we can change the type of the specific variable it is called _____.
5. X+=5; means _____.

3.8 LET US SUM UP

In this chapter we have studied constant, variables, how they are differentiated. How many different data types are there and how to use proper data type? We have also studied different types of operators, their categories and precedence. Finally, we have also seen how the expression is evaluated using type conversion and how we get desired output using type casting.

3.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 3.1

1. %u
2. %ld
3. -128 to 127
4. 0 to 65535
5. 0 to 255

Exercise: 3.2

1. logical
2. sizeof
3. assignment, comparison
4. type casting
5. $X = X + 5;$

Unit 4: Decision Making with Branching and Looping

4

Unit Structure

- 4.1. Learning Objectives
- 4.2. Introduction
- 4.3. Branching instructions
- 4.4. Conditional branching with if and switch...case statements
- 4.5. Loop controls
- 4.6. Break and continue statements
- 4.7. Let us sum up
- 4.8. Check your progress: Possible answers

4.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Use unconditional branching instruction goto in the program
- Understand different types of conditional branching statement if
- Use of switch...case statement
- Learn how the repetition of certain instructions can be done with different types of loops
- Nesting of loops

4.2 INTRODUCTION

We have discussed many C-Language programs in previous chapters. If we observe those programs carefully then we come to know that all programs are executing all its instructions in a sequential order. Once the program starts its execution then it will execute the first instruction then second instruction and in the same order, when the last instruction is executed, program execution will be terminated. All instructions will be executed. But sometime we have to break this sequence. For example in the program of 'Checking the Number Even or Odd' we need to write two printf() statements showing the message 'Given number is Even' and 'Given number is Odd'. But we know that both the statements should not be executed. From both the messages any one message will be printed depending up on the number entered by the user is either even or odd. So, we need some instructions which will be used to control the flow of the program. Such instructions are known as branching instructions. Similarly, sometimes we need to execute some statements has to be executed repeatedly in the program. For that purpose, we need to use loops. In this chapter we will focus on such instructions.

4.3 BRANCHING INSTRUCTIONS

Depends on the instruction perform branching in the program is based on the conditional statement or not branching instruction can be divided in the 2 categories:

1. Unconditional branching instruction
2. Conditional branching instruction

[1] Unconditional branching

Branching instructions which perform branch (break sequential execution) in the program without inspecting any condition is called unconditional branching instruction. Statement 'goto' is used for unconditional branching in the program. Consider the following program.

4.1 Example of goto statement

```
#include<stdio.h>
void main()
{
    printf("\nOne");
    printf("\nTwo");
    goto mylabel;
    printf("\nThree");
    printf("\nFour:");
mylabel:
    printf("\nEnd");
}
```

OUTPUT:

One

Two

End

Here system will execute first two printf() statement in sequential order and print One and Two on the screen. Third statement is goto statement which is unconditional jump or branch instruction, which transfer the flow control on the label mylabel: instruction: and then next instruction will print End. Because of goto unconditional jump statement which has to print Three and Four will be skipped and not be executed. Unconditional jump degrades the performance of the program so it is not advisable to use goto statement more frequently.

[2] Conditional branching

Those branching instructions which evaluate the condition in prior to jump the flow control on specific statement is called conditional branching statement. Following instructions are used as conditional branching.

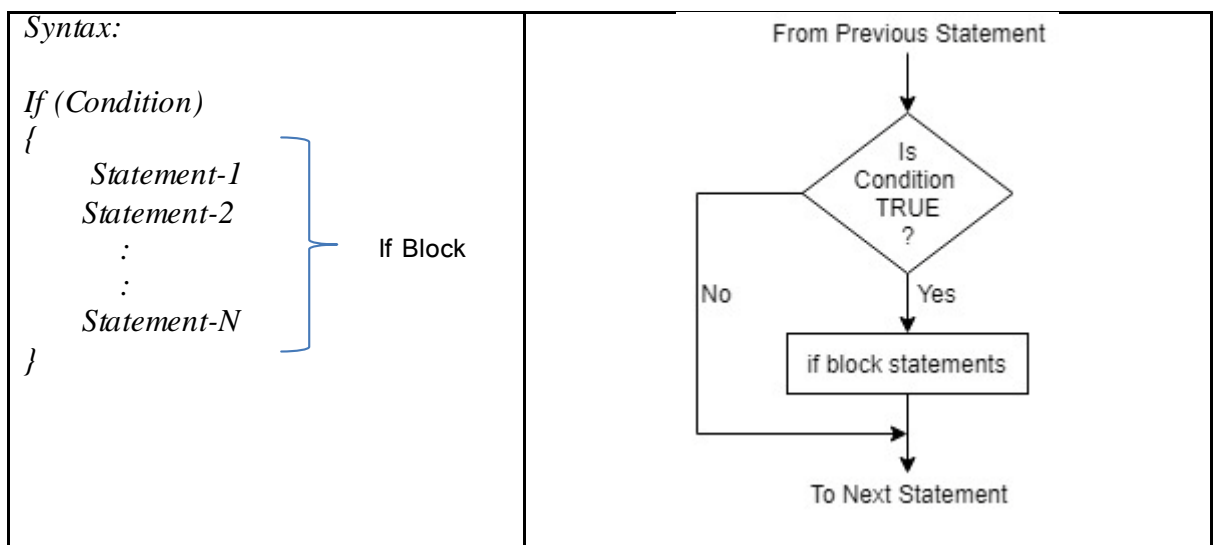
1. If statement
2. If ... else statement
3. If ... else if ... else statement
4. Nested if statements
5. switch ... case statement.

4.4 CONDITIONAL BRANCHING WITH IF AND SWITCH...CASE STATEMENTS

In this section we will discuss different types of if statements available in the C-Language.

THE if STATEMENT

Keyword **if** is used in the C-Language to write if conditional statement. We need to specify logical condition with if statement. Statement or block of statements associated with if statement will be executed when the logical condition specified with if statement is TRUE. If the condition is FALSE then statement or block of statements associated with if will be skipped and will not be executed.



The syntax of the if statement and flow diagram is shown in the above Figure 4.1. Example if condition is given in the following program.

4.2 Example of if statement

```
#include<stdio.h>
void main()
{
    int num, remainder;
    printf("Enter Any Number:");
    scanf("%d",&num);
    remainder=num%2;
    if(remainder==0)
    {
        printf("\nGiven Number is Even:");
    }
}
```

```

    }
    if(remainder==1)
    {
        printf("\nGiven Number is Odd:");
    }
}

```

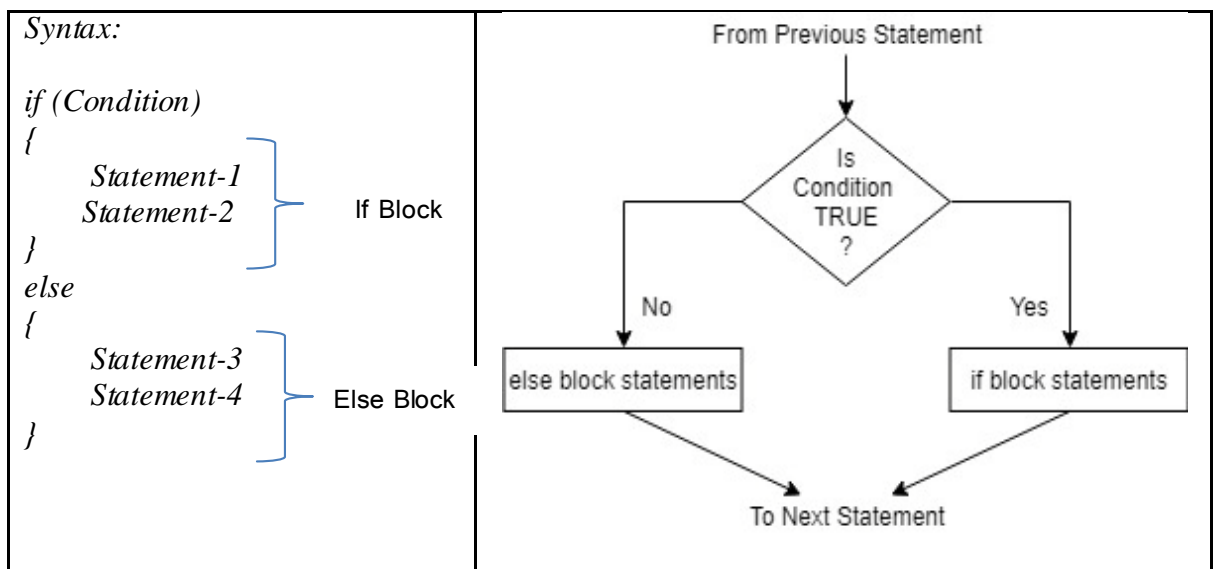
OUTPUT:

Enter Any Number: 28
 Given Number is Even:

In the above program we have taken two variables num and remainder. We prompt to the user to input the number and stores that number in then num variable. We divide the num variable by 2 and stores the remainder in the variable. Then we check if the value in the remainder variable is 0 then only, we print the message 'Given number is Even'. Similarly, if the value in the remainder variable is 1 then only, we print the message 'Given number is odd'. Make sure the printf() statement placed in the if condition will be executed if the condition specified is TRUE, otherwise it will be skipped.

THE if ... else STATEMENT

In the above program we have tested the condition where the value of the remainder variable is 0 or 1 twice. We know that if calculate the remainder after dividing any number by 2 then remainder must be either 0 or 1. Once we check "is remainder is 0?" If yes then the number is Even, and if no then the remainder must be 1 and number is odd. We do not have to check the second condition. For such type of case we can use if ... else statement.



The syntax and flow diagram of if ... else is shown in the above figure. In this statement only one condition is evaluated. Statement 1 and 2 will be executed if the condition is TRUE, otherwise statement 3 and 4 will be executed. In this statement either if block will be executed (in the case condition is TRUE) or else block will be executed (in case condition is FALSE). Both blocks will never be executed and in the same way it also not possible that any of the block is not execute.

4.3 Example of if ... else statement

```
#include<stdio.h>
void main()
{
    int num, remainder;
    printf("Enter Any Number:");
    scanf("%d",&num);
    remainder=num%2;
    if(remainder==0)
    {
        printf("\nGiven Number is Even:");
    }
    else
    {
        printf("\nGiven Number is Odd:");
    }
}
```

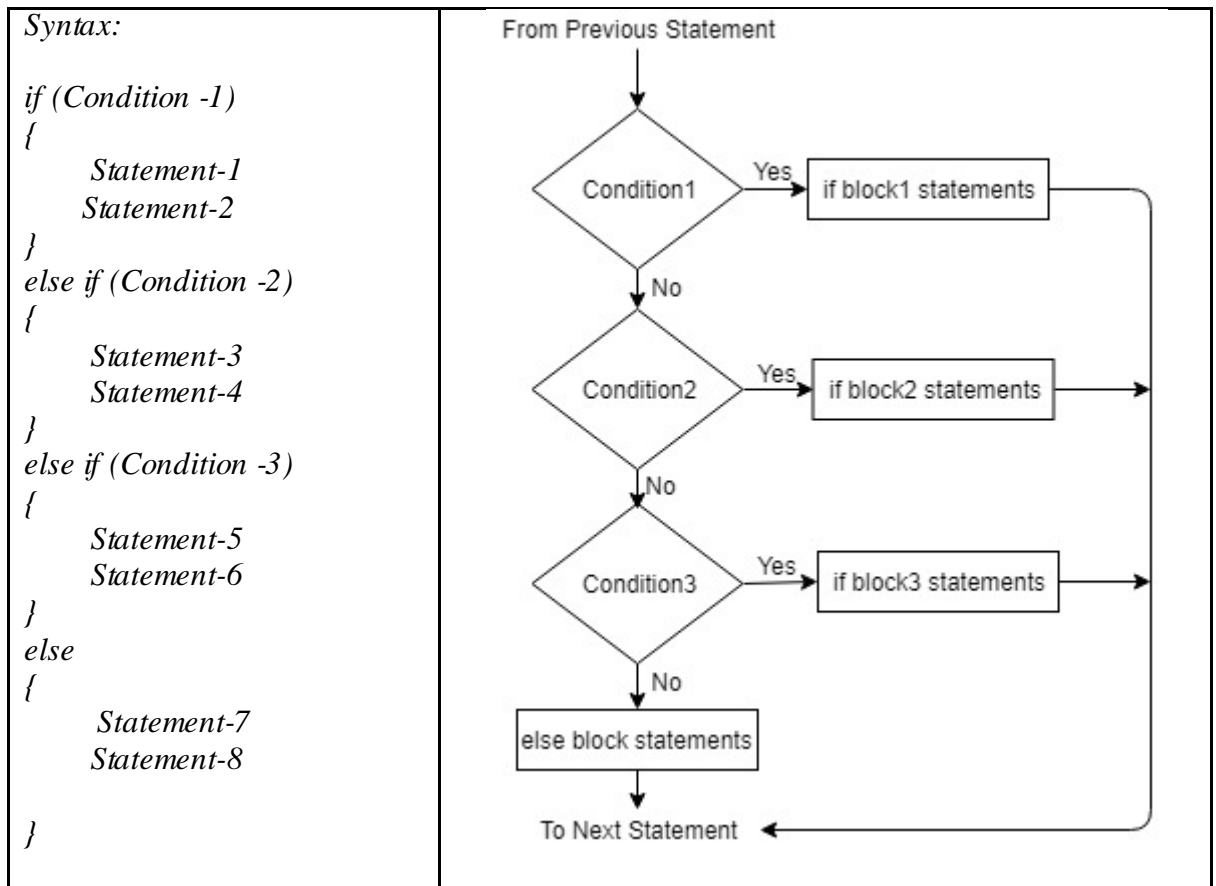
OUTPUT:

```
Enter Any Number: 28
Given Number is Even:
```

THE if ... else if ... else STATEMENT

If we want to evaluate multiple conditions then if ... else if ...else statement is used. We can specify more than 1 condition like condition1, condition2, condition3 as so on.

System is evaluating condition1 if it is TRUE then it will execute first block and skip rest of the conditions. Condition2 will be evaluated when the first condition is FALSE. Similarly, else block will be executed when all conditions are FALSE. Syntax, Flow diagram and example of if ... else if ... else is shown below.



4.4 Example of if ... else if ... else statement

```

#include <stdio.h>
void main()
{
    int marks;
    printf("\nEnter Marks:");
    scanf("%d", &marks);
    if(marks > 75)
        printf("\nDistinction");
    else if(marks > 60)
        printf("\nFirst Class");
    else if(marks > 50)
        printf("\nSecond Class");
    else if(marks > 35)
        printf("\nPass");
    else
        printf("\nFail");
}

```

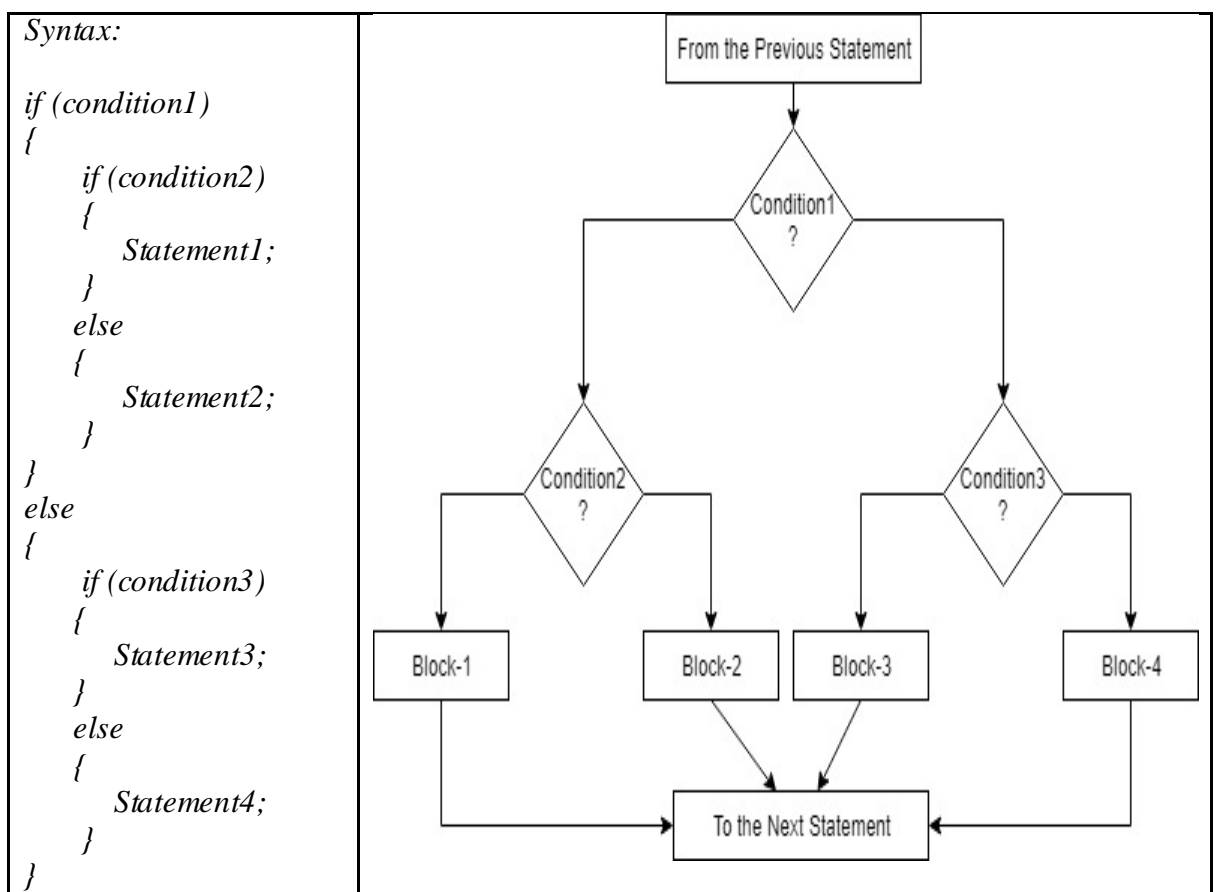
In the above program system will check the first condition that is 'Is marks > 75' if TRUE then will print Distinction. Now system will not evaluate rest of the condition

and they will be simply skipped by the system. If first condition is FALSE then and then system will evaluate second condition that is 'Is marks > 60?'

Else part of the if condition should not have condition. Because else means if all the conditions are FALSE. Opening and closing curly brackets are not needed if there only one statement is there in any block.

THE Nested if ... else STATEMENT

Sometimes we need to write one if statement into one more if statement. When we write one if statement inside one more if statement then it is called nested if statement. The syntax, flow diagram and example of the nested if statement are given below.



4.4 Example of nested if ... else statement

```

#include<stdio.h>
void main()
{
    int num1, num2, num3;
    printf("Enter Any 3 Numbers:");
}

```

```

scanf("%d%d%d",&num1, &num2, &num3);
if(num1 > num2)
{
    if(num1 > num3)
    {
        printf("Greatest Number is:%d",num1);
    }
    else
    {
        printf("Greatest Number is:%d",num3);
    }
}
else
{
    if(num2 > num3)
    {
        printf("Greatest Number is:%d",num2);
    }
    else
    {
        printf("Greatest Number is:%d",num3);
    }
}
}

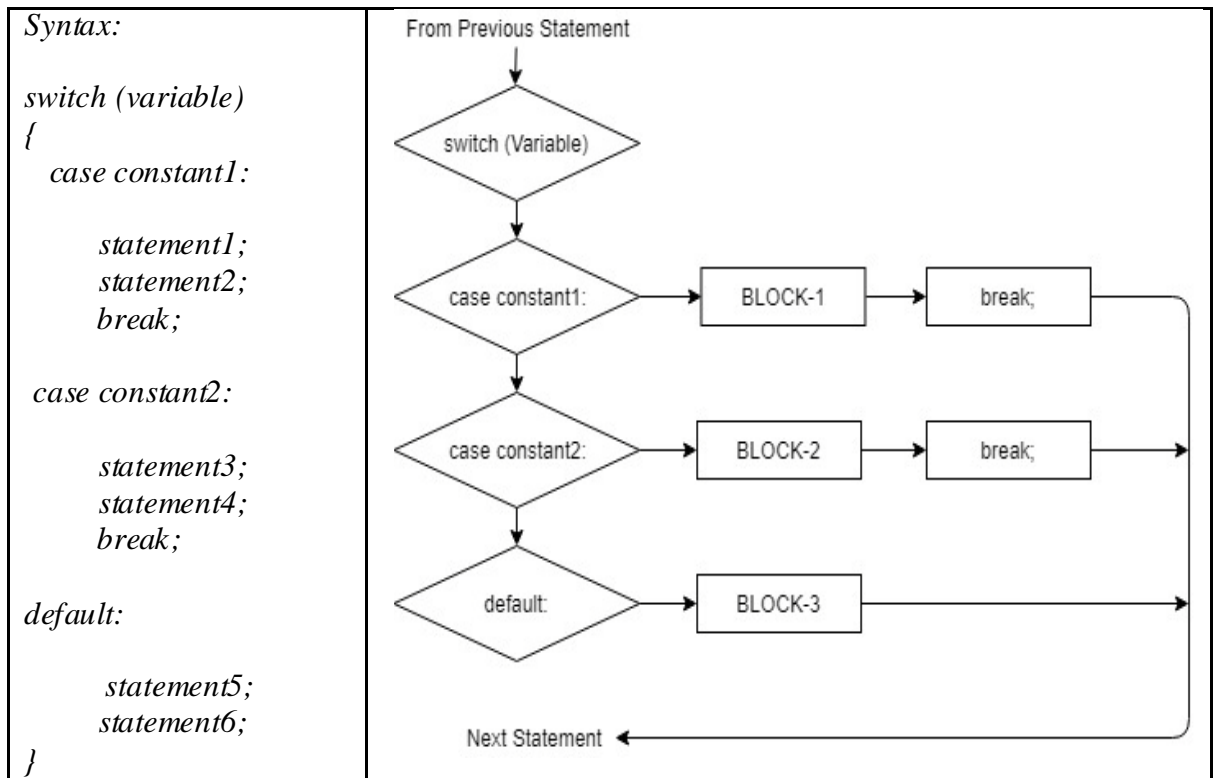
```

In the above program we are accepting 3 values from the user and store them in the num1, num2 and num3 variables respectively. We are comparing is num1 > num2 if yes then num1 is greater than num2 and now there is a competition is between num2 and num3. So, we have written one more if condition, that if num1 > num3 is also true then num1 is the greatest number and has to be printed. If no then num3 is greatest and it has to be printed. If our first condition num1 > num2 is false then num2 is greater than num1 and now there is a competition is there between num2 and num3. To test who is greater among num2 and num3 we have written one more if condition to test num2 > num3 in the else part of the first if condition.

THE switch ... case STATEMENT:

If condition can be used to test equality (==) as well as inequality (<, <=, >, >=, and !=). But switch ...case statement is used only to test equality. We need to pass the expression with the switch statement, which is evaluated and matched with multiple cases. When system found any case to be matched then all the statements written in that case will be executed. If no case is matched then all the statements written in the 'default:' case will be executed. In the switch ... case statement different cases

are separated by 'break' keyword. Syntax, flow diagram and examples of the switch ... case statement are given below.



4.5 Example of switch ...case statement

```

#include<stdio.h>
void main()
{
    int x;
    printf("Enter Any Number from 1 to 5:");
    scanf("%d",&x);
    switch(x)
    {
        case 1:
            printf("One");
            break;
        case 2:
            printf("Two");
            break;
        case 3:
            printf("Three");
            break;
        case 4:
            printf("Four");
            break;
        case 5:
            printf("Five");
            break;
    }
}

```

```

    default:
        printf("Invalid Input");
    }
}

```

In the above program, we have taken a variable 'x' and value of will be entered by the user. Variable 'x' we have passed in the switch statement, which will match the value of 'x' is either 1, 2, 3, 4, or 5 with different cases. If the value of x is matched with particular case then the 'printf()' statement written in that case will be executed, and next statement 'break;' will send the program control out of switch ...case statement. That means no other case is evaluated if one case is matched. If we are not writing the 'break;' statement in each case then all the cases which is below matched case will be executed. So, if in the above program we are not writing 'break;' and we are inputting 3 then output will be "ThreeFourFiveInvalid Inupt" generated. Case 'default:' will be executed when no case is matched. For example, in the above program is we enter 7 then it will print "Invalid Input". Consider the next example of switch ...case statement which takes a character from the user and print the character entered by the user is Vowel or Consonant.

4.6 Example of switch ...case statement with character variable

```

#include<stdio.h>
void main()
{
    char ch;
    printf("\nEnter any lower-case alphabet:");
    scanf("%c",&ch);
    switch(ch)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Entered character is Vowel");
            break;
        default:
            printf("Entered character is Consonant");
    }
}

```

OUTPUT:

```

Enter any lower-case alphabet: a
Entered character is Vowel

```

4.5 LOOP CONTROLS

In some programs, we need to execute a statement or block of statements repeatedly. Loop is used for the same purpose. Loops are available in all middle level to high level programming languages. In the C-Language we have 3 different types of loop. These are:

1. While loop
2. For loop
3. Do ...while loop.

All the loops discussed above are used for the same purpose that is, repeatedly execution of the statement(s). In the While loop and For loop the condition is evaluated before execution of the body part so they are called entry control loop. While in the do ... while loop the condition is evaluated after execution of the body part of the loop, so it is called the exit control loop. While loop and For loop are same but just the way of writing of the loops are different.

When we write a loop in to the program, mainly we have to take care of 3 important instructions.

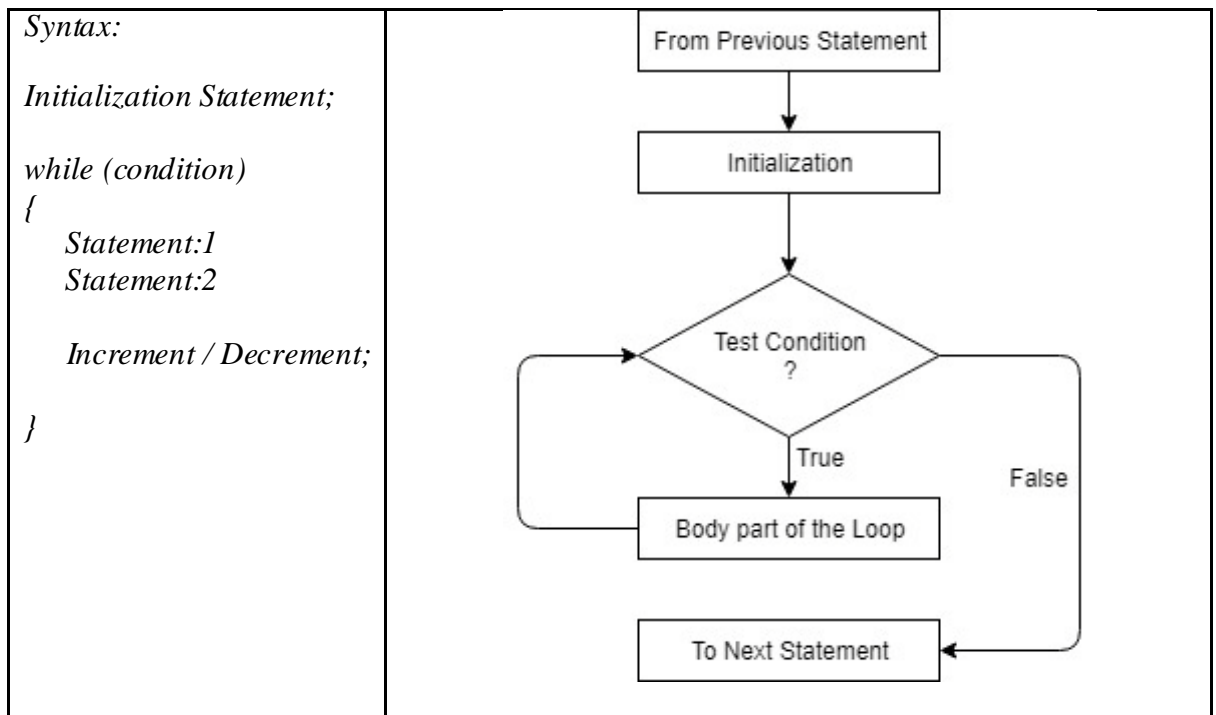
1. Initialization (Starting point of the loop)
2. Condition (Ending point of the loop)
3. Increment / Decrement (Journey from start point to end point)

Before starting any loop, we need to initialize one or more variables. For example, in the program of printing integer numbers from 1 to 10, we need to initialize variable with 1. After initialization condition is evaluated, if the condition is True then and then the body part of the loop is executed. When the condition becomes false then system will stop repeating execution of the body part of the loop. For example, when variable is smaller than 10 keep printing numbers, when variable becomes 11 stop printing numbers. Third instruction is, we need to increment the value of the variable by 1, because if we not do then variable never reaches to 11 and loop will run for ever.

WHILE LOOP:

While loop is an Entry control loop (condition is evaluated before executing body of the loop). In this loop we write the initialization instruction before loop. 'while'

keyword is used, and along with 'while' we need to specify condition. Body part of the loop will be executed repeatedly if the condition is true. Increment / decrement statement has to be specified within the body of the loop. Syntax, flow diagram and example of while loop are given below.



4.7 Example of while loop

```

#include<stdio.h>
void main()
{
    int i;
    i=1; //Initialization of variable i to 1
    while(i<=10) //Conditional Statement
    {
        printf("\n%d", i);
        i++; //Increment
    }
}

```

If we execute the above program then it will print from 1 to 10. Similarly, if we want to print all even numbers from 1 to 20 then we can do it by placing an if condition in the loop as shown in the next program.

4.7 Example of while loop

```

#include<stdio.h>
void main()

```



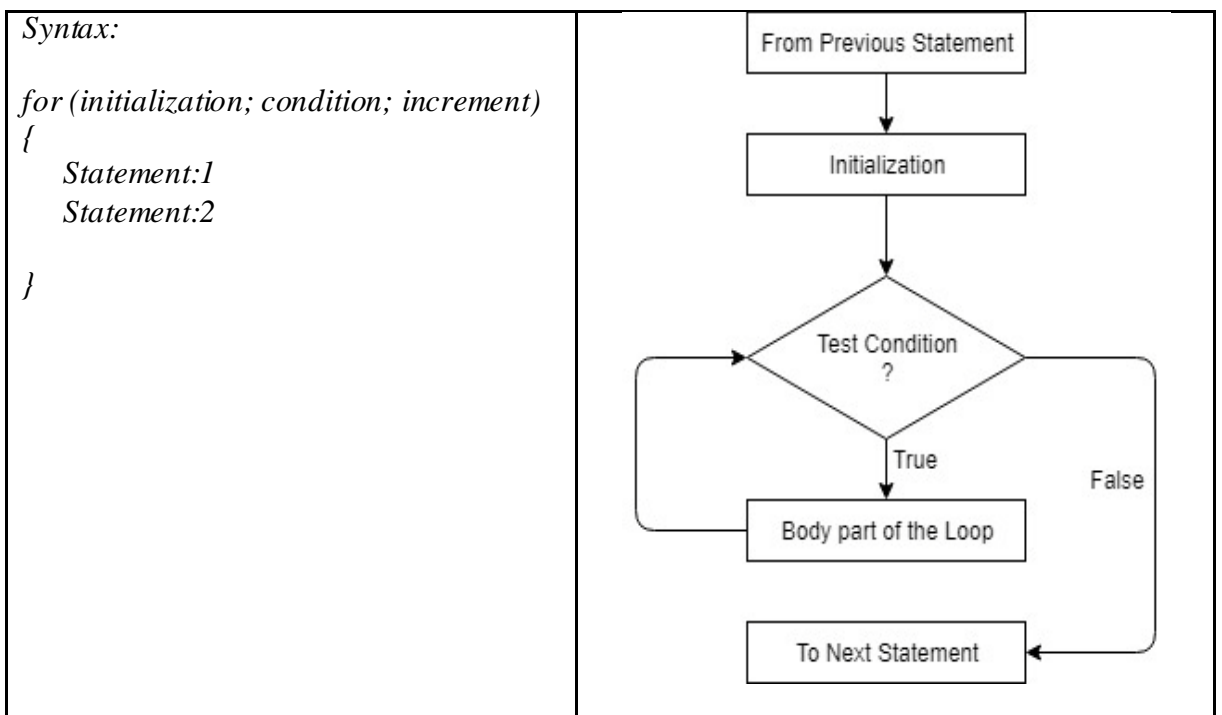
```

{
int i;
  i=1; //Initialization of variable i to 1
  while(i<=10) //Conditional Statement
  {
    if(i%2==0)
    {
printf(“\n%d”, i);
    }
    i++; // Increment
  }
}

```

FOR LOOP:

For loop is similar to while loop. It is entry controlled loop and it will evaluate the condition prior to execution of the body part of the loop. It executes and repeats the body part of the loop if the condition is TRUE. When the condition becomes FALSE the flow control will come out of the loop and the next statement written after the loop will be executed. The difference between while loop and for loop is its syntax. In the while loop initialization, condition and increment / decrement statements are written in 3 different statement where as in for loop all 3 things can be written in one statement. So compare to while loop, for loop program looks smaller. In the while loop ‘while’ keyword is used, similarly in the for loop ‘for’ keyword is used. The syntax, flow diagram and example of the for loop is given below:



4.7 Example of for loop

```
#include<stdio.h>
void main()
{
    int i;
    for(i=1;i<=10;i++)
    {
        printf("\n%d",i);
    }
}
```

Program written above will print the numbers from 1 to 10 on the console. In the next program we will try to find factorial of a given number. In this program we are taking a variable 'fact', which is initialize by 1. Each time we will multiply it with variable 'i' which starts from 1 and go up to n (a number entered by the user) by a for loop.

4.8 Finding factorial of a given number

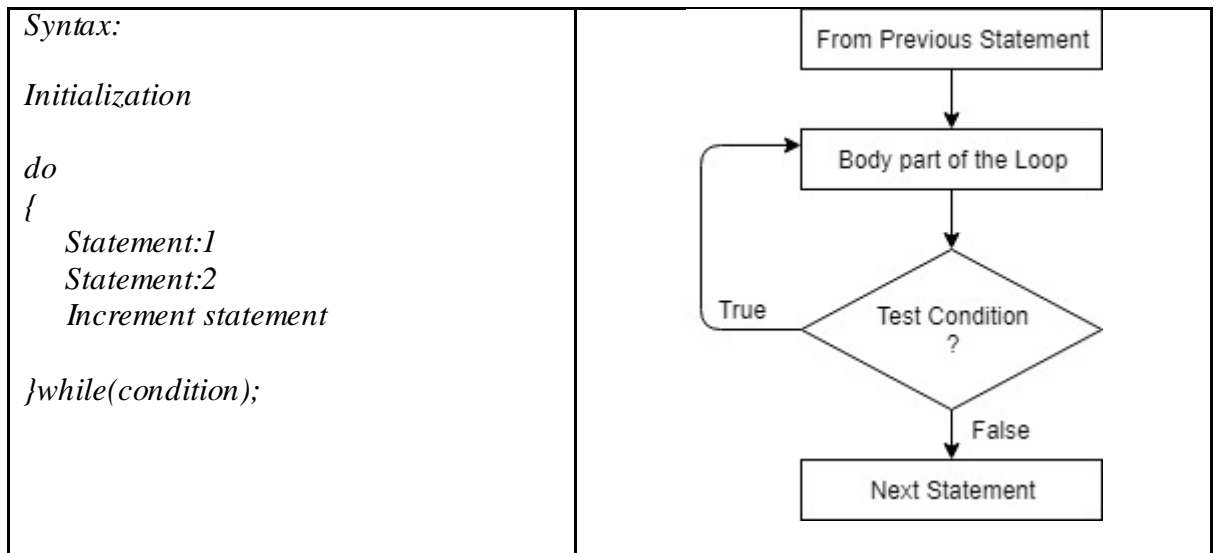
```
#include<stdio.h>
void main()
{
    int i,fact, num;
    printf("Enter Any Number:");
    scanf("%d", &num);
    for(i=1,fact=1;i<=num;i++)
    {
        fact=fact*i;
    }
    printf("Factorial of number %d is =%d", num,fact);
}
```

OUTPUT:

```
Enter Any Number:5
Factorial of number 5 is =120
```

DO...WHILE LOOP:

Do...while loop is exit control loop. Here program flow control enters into the loop without checking any condition. After the execution of the body part of the loop condition is evaluated. If condition is 'TRUE' then second iteration of the loop will start, but if condition is 'FALSE' then body part of the loop will not be executed. That means, in this loop body part of the will be executed at least once. Another difference between while loop, for loop and do...while loop is do...while loop ends with semicolon (;). The syntax, flow diagram and example of the do...while loop is given below.



4.9 Count positive, negative, zeros entered by the User

```
#include<stdio.h>
void main()
{
    int num, positive=0,negative=0,zero=0;
    do
    {
        printf("Enter Number[Enter 999 to Exit]:");
        scanf("%d", &num);
        if(num!=999)
        {
            if(num>0)
                positive++;
            else if(num<0)
                negative++;
            else
                zero++;
        }
    }while(num!=999);
    printf("You have entered %d positive, %d negative,%d zeros",positive, negative,zero);
}
```

OUTPUT:

```
Enter Number[Enter 999 to Exit]:5
Enter Number[Enter 999 to Exit]:10
Enter Number[Enter 999 to Exit]:-60
Enter Number[Enter 999 to Exit]:20
Enter Number[Enter 999 to Exit]:0
Enter Number[Enter 999 to Exit]:23
Enter Number[Enter 999 to Exit]:999
You have entered 4 positive, 1 negative,1 zeros
```

NESTED LOOPS:

In some programs we need to take a loop inside one more loop. When in the program we take a loop, inside one more loop is called a nested loop. We can write nesting loop for any kind of loop we have discussed above. We can write nested for loop, nested while, or nested do...while loop. The following program demonstrate nesting of loop.

4.10 Printing Pyramid

```
#include<stdio.h>
void main()
{
    int num, row, column;
    printf("Enter Any Number:");
    scanf("%d", &num);
    for(row=1;row<=num;row++)
    {
        for(column=1;column<=row;column++)
        {
            printf("* ");
        }
        printf("\n");
    }
}
```

OUTPUT:

Enter Any Number:5

```
*
* *
* * *
* * * *
* * * * *
```

4.11 Printing Triangle

```
#include<stdio.h>
void main()
{
    int num, row, column,space;
    printf("Enter Any Number:");
    scanf("%d", &num);
    for(row=1;row<=num;row++)
    {
        for(space=1;space<=num -row;space++)
        {
            printf(" ");
        }
        for(column=1;column<=row;column++)
        {
            printf("* ");
        }
    }
}
```

```

    printf("\n");
}
}

```

OUTPUT:

Enter Any Number:5

```

*
**
***
****
*****

```

4.6 BREAK AND CONTINUE KEYWORDS:

Break: Keyword 'break' is used to terminate the premature loop. For example, if we are writing a program to check given number is prime or composite, we start dividing number by 2, 3, 4... up to that number-1. If we get a single number by using, we can divide the number entered by the user then we have to exit the loop. When the system will execute the 'break' statement flow control will terminate the loop and comes out of the loop.

4.11 Program to check given number is prime or not.

```

#include<stdio.h>
void main()
{
    int logic=1, num, i;
    printf("Enter Any Number:");
    scanf("%d", &num);
    for(i=2;i<num;i++)
    {
        if(num%i==0)
        {
            logic=0;
            break;
        }
    }
    if(logic==1)
        printf("\nGiven Number is Prime:");
    else
        printf("Given Number is Not Prime:");
}

```

Continue: When the 'continue' statement will be executed then flow of the control is transferred to the header of loop. In this case statements written below 'continue' will not be executed.

4.12 Use of continue statement in the program.

```
#include<stdio.h>
void main()
{
    int i;
    for(i=1;i<=10;i++)
    {
        if( i>=4 && i<=8)
            continue;
        printf("\n%d",i);
    }
}
```

OUTPUT:

1
2
3
9
10

Excercise:1

1. _____ loop is exit control loop.
2. If we take a loop, inside one more loop is called _____.
3. For loop is _____ control loop.
4. To terminate premature loop _____ keyword is used.
5. Switch...case statement is used to test only _____.
6. Statement if(-5) is always _____.
7. _____ loop will be executed at least once.
8. Keyword _____ performs unconditional jump.
9. A loop never ends, is called _____.

4.7LET US SUM UP

In this chapter we have studied flow control of the program. We have started with unconditional jump statement 'goto', then we have learnt different types of if conditional statements. We have seen how the equality can be tested with switch...case statement. We have also discussed different types of loops, and how to use them in the C-Program. Finally, we have ended our discussion with 'break' and 'continue' statements.

4.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 3.1

1. Do...while
2. Nested loop
3. Entry
4. break
5. Equality
6. True
7. Do...while
8. goto
9. infinite loop

Block-2

C Programming Concepts

Unit 1: Working with Functions

1

Unit Structure

- 1.1. Learning Objectives
- 1.2. Introduction
- 1.3. System defined functions
- 1.4. User defined functions
- 1.5. How to make user defined functions?
- 1.6. Categories of user defined functions
- 1.7. Variable scope and lifetime
- 1.8. Storage classes
- 1.9. Recursion
- 1.10. Let's sum up
- 1.11. Check your Progress: Possible Answers

1.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand function and its type
- Learn the basic structure of the User Defined Functions(UDFs)
- Know scope and lifetime of Local and Global variable
- Learn storage classes
- Understand the concept of recursion

1.2 INTRODUCTION

When the length of the program is large, it is difficult to manage it. Large sized program is not readable and difficult to understand as well. The best way to handle this problem is modularity. Complex and lengthy code of the program is logically divided into small and easier blocks of codes called modules or functions. Functions are small subprograms which perform dedicated task only, while collectively handle complex task. Which will be manageable and easy to understand. C-Language supports functions.

Set or group of executable statements is called function. There are two types of functions are there:

1. System Defined Functions
2. User Defined functions

1.3 SYSTEM DEFINED FUNCTIONS

System defined functions are also known as library functions or primitive functions. These functions are located in to some library (header file). In this, user is not responsible to write the logic of the function, but user is just using it by including proper header file. These functions are designed by somebody else, and readily available. As a programmer, we should know “how to use this function?” and “The function is available in which header file?”. For example, the following program in which `sqrt()` function is used to find the square root of the given number.

Example 1.1 Finding square root of given number

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
    int x=49,y;
    clrscr();
    y=sqrt(x);
    printf("Square root of %d is = %d",x,y);
    getch();
}
```

This program will print the output 'Square root of 49 is = 7'. Now in this program **printf()**, **clrscr()**, **sqrt()**, and **getch()** are the system defined functions. printf() function is located in the header file called stdio.h, clrscr() and getch() functions are located in the conio.h, and sqrt() function is located in the header file called math.h. In this example, we are not writing any logic to find square root of a given number, but we are including proper header file, math.h and we are calling sqrt function. We are passing our data 49 which is stored into variable x, and function will return, it's square root that is 7 in to variable y. These type of function, in which we are not writing any logic or coding, function are already written into some header file is called system defined function. We just have to include the header file and call the functions to perform desire tasks.

1.4 USER DEFINED FUNCTIONS (UDFs)

User defined functions (UDFs) are those, which are not readily available in the header file. Here programmer is responsible to write the logic (coding) of that function and then, that can be used in the main or any other functions. Dividing the program into various functions is called modular approach. By this our program will be readable. By dividing the program in to various functions will reduce the complexity of the code. Not only that, but it will speed up the process of execution because it saves lots of memory. Another advantage of dividing program into functions is reusability of the code. Once function will be written and tested that can be called several time, and it will give the proper answer every time.

User defined function	Library function
These functions are created by users.	These functions are available in the library of C software.
User defined functions are subprograms to carry out the different tasks of the main program. These functions can be stored in user defined header file.	With the standard C compilers, standard libraries are provided with the different standard functions used to do some common tasks is known as library functions.
These functions can be accesses by including header files in a 'C' program like following preprocessor directive #include "test.h"	These functions can be accesses by including header files in a 'C' program like following preprocessor directive #include <stdio.h>
Example: sum();	Example: printf();

1.5 HOW TO MAKE USER DEFINED FUNCTIONS

Structure of defining function:

```

<Return Type> function_Name (<Argument List>)
{
    Statement 1;
    Statement 2;
    .
    .
    .
    Statement N;
}

```

1. Return Type:

Whenever we want to define A function, we have to mention the return type of the function. Some functions are not returning the value; in this case we have to mention 'void' in the return type section. Consider the following example, in this 'displaymsg()' function does not return anything to any other function.

Example 1.2 Displaying message using User defined function

```
#include<stdio.h>
#include<conio.h>
void displaymsg()
{
    printf("\nHello");
}
void main()
{
    int i;
    for(i=0;i<10;i++)
    {
        displaymsg();
    }
}
```

In this example, we have create a function called 'displaymsg()', which is not returning anything to any other function, So, its return type is 'void'. 10 times main function is calling this function (because we are calling function in the loop and loop is repeated for 10 times), so as an output 10 times 'Hello' will be printed on the screen.

Now consider a program, which will take two numbers from the main function, the numbers will be added, and sum will be return by the sum() function to main() function. Obviously, here sum of two integer numbers will be an integer. Therefore, the return type of the user defined function sum() is int.

Example 1.3 User defined function to do sum of two integers

```
#include<stdio.h>
#include<conio.h>
int sum( int num1, int num2)
{
    int result;
    result = num1 +num2;
    return result;
}
void main()
{
    int i,j,k;
    i=5;
    j=7;
    k=sum(i,j);
    printf("\nAnswer is: %d",k);
}
```

In this above example, 'main()' function passes two integer numbers 'i' and 'j', that are 5 and 7 to the sum function. Sum function takes (copies) those values, in to its own two variables called num1 as num2 respectively. Now sum function will add these numbers, in to variable result. This variable result (12) will be return to the main program. In the main program, we have declared an additional variable 'k', which takes the value, which is returned by sum() function, and that will be printed on the screen.

In the similar way function can also return 'float', 'char' or any other type of data.

Points to Remember:

- It is not necessary that function always returns a value. Sometime, functions are designed to perform some task in which, returning value is not necessary, in this case the return type of the function will be 'void'.
- Function can also return any value, that can be either int, or float, or char or pointer (which is discussed in the later chapter) or any other type of data.
- Function can return only one value.
- Function is transferring the control to the caller function.
- If no return type in mentioned in the user defined function, then 'int' will be taken as a default return type.

Now consider the following example:

Example 1.4 Nesting of user defined functions

```
#include<stdio.h>
#include<conio.h>
void function2()
{
    printf("\nYou are in Function2");
}
void function1()
{
    printf("\nYou are in Function1");
    function2();
    printf("\nYou are Back to Function1");
}
void main()
{
    clrscr();
    printf("\nYou are in the Main Function");
```

```

        function1();
        printf("\nYou are back to Main");
    getch();
}

```

OUTPUT:

```

You are in the Main Function
You are in Function1
You are in Function2
You are Back to Function1
You are back to Main

```

In the above program, there are three functions are there, (1) Function1 (2) Function2 and, (3) main. System starts with the main() function. System is responsible to execute all the lines, which is written in the main function. First of all, it will clear the screen as we are calling clrscr() function. Then it will print "You are in the Main Function". Then main function is calling Function1(). So, system must have to execute, all the lines which is return in the Function1(). Here, main function is calling Function1(). System will now start to execute, one by one line of Function1(). First line of the Function1() will print "You are in the Function1", then system will execute the second line of the Function1(). Here Function1() is calling to the Function2(). So, system jumps to the Function2(). In function2(), there is only one line is there, and system will print "You are in Function2". After execution of Function2(), system will turn back to the Function1(), because Function1() has called Function2(), and it will continue to execute third line of Function1(). Here, System will print "You are Back to Function1". After executing all the lines of function1(), system will turn back to its caller function that is main(), and we will get the message "You are back to Main". So, we can say function must pass the control to its caller function.

2. Function Name:

Every function will have its own unique identity, and this identity we can provide by giving a unique name to it. So that, other functions can call this function.

Naming rule for function:

- Function name must start with either letter (a to z, A-Z) or underscore (_).
- Function name must be unique. Once the name is given to one function, that name we cannot use again for the other function.

- Function name cannot be keyword (if, else, include, for, switch, case, while, do, int, float etc).
- Special symbols such as (+, -, *, /, \$, @, #) cannot be used in the function name.
- Space is not allowed in the function name.

3. Argument List:

When we are calling a function sometime it is essential to pass, some initial data to it. For example, sum function cannot do the sum, if we are not giving two integers to it. So, the initial data we are passing to the function, at the time of calling it, is called argument list.

1.6 CATEGORIES OF UDFs

1 No argument no return value:

Consider the following program, in which main function is calling to another function called 'sayhello()' and this function, will put the message on the screen (console) 'Hello World!!!'. Here main() function is not passing any argument to our user defined function 'sayhello()', and 'sayhello()' function is not returning any value to the main function. Here sayhello() function does not takes any argument, so we have written void in the argument list, as well as it is not returning any value, so we have written 'void' in the return type:

Example: 1.5 User defined function with no argument and no return

```
#include<stdio.h>
#include<conio.h>
void sayhello(void);
void main()
{
    clrscr();
    sayhello();
    getch();
}
void sayhello(void)
{
    printf("Hello World:");
}
```


1. Arguments but no return value:

Now consider the following program, in which main() function passes two numbers (arguments), to the function sum(). Function sum() takes two numbers from the main() function, and it will do the summation of these numbers, and resultant value is printing on the console. In this example, our user defined function takes two arguments from the main() function, but it is not returning anything to the main() function. So, the return type of the sum function is void.

Example: 1.6 User defined function with arguments but no return

```
#include<stdio.h>
#include<conio.h>
void sum(int, int);
void main()
{
    clrscr();
    sum(5,7);
    getch();
}
void sum(int x, int y)
{
    printf("Sum is : %d", (x+y));
}
```

2. No arguments but return value:

Now consider the example, in which main() function is calling a function called 'PIvalue()' and every time this function will return 3.14 (float). Because of function is not taking any value from the main() function, we have written void in the argument list, and every time while it is called, it returns 3.14. So, return type of the function is float.

Example: 1.7 User defined function with no arguments but returns value

```
#include<stdio.h>
#include<conio.h>
float PIvalue(void);
void main()
{
    float area, radius=10;
    clrscr();
    area=PIvalue() * radius * radius;
```

```

    printf("Area of the circle is:%.2f",area);
    getch();
}
float PValue(void)
{
    return 3.14;
}

```

3. Arguments and return value:

Consider a program in which main() function passes two integer numbers 5 and 7 to the function sum(). Here function sum() takes two arguments (two numbers of type integer) and it will return the sum of these numbers that is again integer.

Example: 1.8 User defined function with arguments and with returns value

```

#include<stdio.h>
#include<conio.h>
int sum(int, int);
void main()
{
    int ans;
    clrscr();
    ans=sum(5,7);
    printf("\nResult is:%d",ans);
    getch();
}
int sum(int x, int y)
{
    return (x+y);
}

```

1.7 VARIABLE SCOPE AND LIFETIME

There are two types of variables are there:

1. Local Variables
2. Global Variable

Those variables, which are declared inside the function is called local variable for that function. They are created when the function is called and destroyed automatically, when the function is exited. So, its lifetime is same as how much time the function resides in the memory. Its scope is limited to the function; Outside of this function we cannot access, that variable. Means we cannot access any variable

declared locally in one function, from any other function. When function has been called, the variable will be declared and after the execution of the function the variable will be flushed out from the memory.

Variables that are both alive and active throughout the entire program are known as global variables. Global variables are defined outside the function. Unlike local variables, global variables can be accessed by any function in the program. Because global variable can be accessible everywhere in the program the scope of global variable is whole program (not limited to the function as local variable). The scope of the variable is not to any function, but entire program. Global variables are created in the memory when the program starts, and they will remain in the memory till program ends. Therefore, the life time of the global variable is same as life of the program. Because of it occupies space in the memory for long period of time, we do not have to use global variables unless and until they are needed.

Local variables can be accessed within the function only; other function cannot access this variable. Where, global variables can be accessed anywhere in the program.

Local variable	Global variable
Local variables are defined inside the function.	Global variable are defined outside the function.
They are created when the function is called and destroyed automatically when the function is exited.	Variables those are both alive and active throughout the entire program.
Example: <pre>void main() { int a=10; ----- ----- }</pre>	Example: <pre>int a=10; void main() { ----- ----- }</pre>

Now consider the following example:

```
#include<stdio.h>
#include<conio.h>
```

```

void printdata(void);
int globalvar;    //This is global variable
void main()
{
    int localvar;//This is local variable
    localvar=5;
    globalvar=10;
    printf("\nLocalvar=%d",localvar);
    printf("\nGlobalvar=%d",globalvar);
    printdata();
}
void printdata()
{
    printf("\nGlobalvar is=%d",globalvar); //Accessing global var in user defined
function
}

```

In this program, because of we are writing user defined function below main() function, prototype is essential, and that we have declared after including the header files.

In the user defined function if we are accessing global variable, then it is fine and valid, but if we are trying to print the value of the local variable, it gives error, as the variable is declared inside the main() function, it is a local variable of main() function and cannot be accessible by other function.

Resolving conflicts:

Consider the example given below:

```

#include<stdio.h>
#include<conio.h>
void function1()
{
    int a;
    a=5;
    printf("\nIn the Function1 a is=%d",a);//This will print 5
}
void main()
{
    int a;
    a=10;
    printf("\nIn the main a is=%d",a); //This will print 10
    function1();
}

```

In this example variable 'a' is declared twice. Once it is declared in the main, and again it is declared in the function1. But still this program works fine, because the

scope of the variables is in the different functions. Function1 can access only that variable, which is declared inside function1(), and main() function, can access only that variable, which is declared in the main() function. Function main() cannot access variable 'a' which is declared in the function1(), and function1() cannot access that variable which is declared in the main(). So, there is no conflict is here.

But consider a scenario in which we have two variables, one is global and another is local variable. Can we give the same name to both? See the example given below which has two variables having same name 'a'. One is global variable, and another is local variable.

What happened if we are assigning or printing the value to variable 'a' in the main? Consider following example:

```
#include<stdio.h>
#include<conio.h>
int a=10;
void main()
{
    int a=5;
    printf("\n A is:%d",a);
}
```

In this program global 'a' is initialized with 10, which is also accessible in the main. [As we know global variables are accessible in all the functions of the program]. In the main program we have one more variable which is local variable, having same name 'a'. This is initialized with 5. What happened, if we are printing the value of 'a'? It will print 5 (value of local variable) or 10(value of global variable)?

Answer is it will print 5 (value of local variable). If you want to access global variable, in the main() function, then the scope resolution operator (::) is used.

```
printf("\nGlobal A is: %d", ::a);
```

This statement will print the value of global variable 'a', which is 10.

1.8 STORAGE CLASSES

The storage class determines duration of the storage, linkage and scope. In the C-language, we make modular program where the program is divided into the number of blocks. Some variables are accessible in particular block or some variable are accessible throughout the program. In the C-language four storage classes are there, these are:

[1] Automatic variables: Automatic variables are declared inside a function in which they are to be utilized. They are created, when the function is called and destroyed automatically, when the function is exited. So, the name is automatic. By default, when we are declaring any local variable, they are automatic variable.

```
void main()
{
    int x;
    auto int y;
}
```

In the above function both x and y are automatic variables.

[2] Static Variable: The value of the static variables persists until the end of the program. A variable is declared using 'static' keyword. For example:

```
static int x;
static float y;
```

A static variable may be internal or an external type, depending on the place of declaration. Internal static variables are those which are declared inside the function. The scope of the internal static variables extends up to the end of the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in the memory throughout program.

A static variable is initialized only one, when the program is compiled. It is never initialized again.

Example:

```
void main()
{
    int i;
    for(i=0;i<5;i++)
    {
        funcstatic();
    }
}
```

```

}
void funcstatic()
{
    static int x=0;
    x=x+1;
    printf("\n%d",x);
}

```

This program will print 0,1,2,3,4. But if instead of static, x is auto then will get 1,1,1,1,1

[3] Register variable: Normally, variables are created in the main memory of the system. But we can tell the compiler to keep the value of the variable, in the register instead of main memory. This type of variables is known as register variables. Register are faster than main memory, so if we are declaring loop control variables as a register variable, then it will speed up the process of the program. This can be done as follows:

Example:
register int counter;

If the variable is created in the register rather than main memory then, the access of that variable as well as change in the value of that variable will become faster.

[4] Extern variable: extern variables are external variables. Those variables which are declared in another file and not in the same file of our program are called external variable. To test the external variable, create a file called 'extvar.c' and type following code:

```
int j=4;
```

Now, create another file called 'testext.c' in the same directory and type following code.

```

#include<stdio.h>
#include "extvar.c"
void main()
{
    extern int j;
    printf("Value of External variable J is:%d ",j);
}

```

OUTPUT:

Value of External variable J is: 4

1.9 Recursion:

In normal case the function calls to the other function and other function will return the value, but sometime function calls itself. When the calls itself, it called the recursion.

So, we can define recursion is a process of calling function itself. And the function calls itself is known as recursive function.

Consider the program in which function 'fact()' which gives factorial of given number, is calling itself.

Example 1.9 Write a recursive function which will return the factorial of given number:

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
    int num,ans;
    clrscr();
    printf("\nEnter any number:");
    scanf("%d",&num);
    ans=fact(num);
    printf("\nFactorial of %d is: %d",num,ans);
    getch();
}
int fact(int tmp)
{
    if(tmp==1)
        return 1;
    else
        return (tmp * fact (tmp-1));
}
```

In this example, we can find function fact(), is calling to the same function in the statement '**return (tmp * fact (tmp-1));**'. This is called recursion.

Now, consider another program in which will print Fibonacci series up to some number. Here function fibo() calls itself, so we can say that this function is recursive function.

Example: 1.10 Write a recursive program to generate Fibonacci series up to some number.

```
#include<stdio.h>
#include<conio.h>
int fibo(int,int,int);
void main()
{
    int num;
    clrscr();
    printf("\nEnter any number:");
    scanf("%d",&num);
    printf("\n1\n1");
    fibo(1,1,num);
    getch();
}
int fibo(int x, int y, int num)
{
    int sum=x+y;
    if(sum > num)
        return 0;
    printf("\n%d",sum);
    x=y;
    y=sum;
    fibo(x,y,num);
}
```

In the above program, fibo() function calls itself, we can say it is a recursive function.

Exercise:1 Fill in the blanks:

1. _____ variables are declared in the register rather than main memory.
2. The process of function call itself is called _____.
3. The scope of _____ variable is limited to that function only.
4. The variable declared outside of the function is _____.
5. Default return type of the function is _____.

1.10LET US SUM UP

In this chapter we have learnt two types of functions that is system defined and user defined function. We have seen how can we create user defined function for our program. Then we have seen the scope and lifetime of local and global variable. We have learnt storage classes, and finally we have focused on recursion.

1.11 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 1

8. register
9. recursion
10. local
11. global
12. int

Unit 2: Working with Arrays and Strings

2

Unit Structure

- 2.1. Learning Objectives
- 2.2. Introduction to Arrays
- 2.3. Understanding Arrays
- 2.4. One-dimensional Arrays
- 2.5. Two-dimensional Arrays
- 2.6. Strings
- 2.7. String functions
- 2.8. Let us sum up
- 2.9. Check your Progress: Possible Answers

2.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Learn declaring and initializing arrays
- Understand basic terminologies of arrays
- Work with One-dimensional and Two-dimensional arrays
- Handle strings
- Understand various string functions

2.2 INTRODUCTION TO ARRAYS

Arrays are finite and ordered collection of homogeneous data. Homogeneous means same type of data. As arrays are collection of the data, we can store more than one element in the array under one common name, but make sure all the elements are of same type

2.3 UNDERSTANDING ARRAYS

Defining Array

If in the C-Language, you write '*int x [10];*' in the declarative section of the program then you have a single variable (array) named 'x' which can store 10 elements of same type ('int' in this case) for you.

Initializing Array

We can initialize arrays in three different ways:

3. Initializing array at the time of its declaration

```
int x [10] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50};
```

```
char y [5] = {'A', 'E', 'I', 'O', 'U'};
```

in the above examples, 'x' is an integer array which is initialized by 10 data elements i.e. 5, 10, 15, 20, 25, 30, 35, 40, 45 and 50. First data element '5' will be stored on the first 0th position of the array x. That means x [0] is '5'. Similarly, 10 is stored on 1st position, 15 is stored on 2nd position and so on. Finally, data element 50 will be stored on 9th position. In C-Language array index starts from 0 so last element we can find on the position Size – 1. Here the size of array 'x' is 10 so last data element '50' will be stored on position 9.

In the another example we have declared an array 'y' of type character, which is initialize by five character data elements 'A' , 'E' , 'I' , 'O' and 'U'. First data element 'A' will be stored on 0th position and 'U' will be stored on 4th position.

4. Declaring array and initializing it with some static values:

```
int x [5];
```

```
char y [5];
```

```
x [0] =5;
```

```
x [1] = 10;
```

```
x [2] = 15;
```

and so on. Similarly, array 'y' can be initialized as,

```
y [0] = 'A';
```

```
y [1] = 'E';
```

and so on.

5. Declaring and initializing array by user input

```
int x [10], i;
```

```
for (i=0; i< 10; i++)
```

```
{
```

```
    printf (" Enter Any Number:");
```

```
    scanf("%d", & x[i]);
```

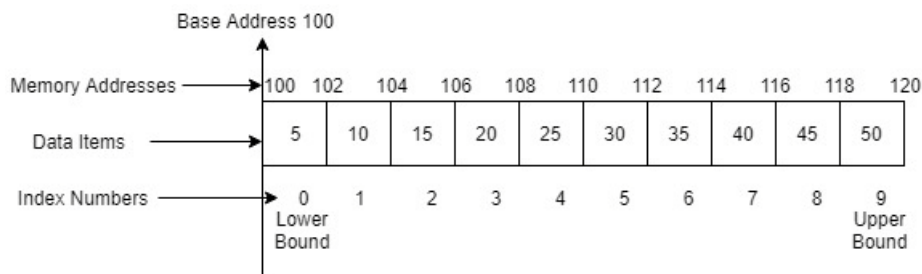
```
}
```

Terminologies

6. **Type:** Type of an array represents type of data can be stored in the array. For example, array can be of type int, float, char, double etc.

7. **Size:** Size of an array represent number maximum data elements that can be accommodated in the array. For example, if we declare 'int x [10]' then array 'x' can accommodate maximum 10 elements. So, we can say size of array x is 10.
8. **Index:** Each element stored in the array has unique reference number is called index number. It is denoted in the square brackets like []. As we have discussed in the above example $x[0] = 5$, means data element 5 is stored in at the index number 0 of the array 'x'.
9. **Range:** Range is a set of all valid index numbers. We know that in C-Language array index starts from 0, it is called Lower bound of the array. If we declare array 'int x [10];' then last element we can place on 9th position. It is called upper bound of an array. Set of all possible integers from lower bound to upper bound is called range of an array. So, the range of array x is 0 to 9.
10. **Base:** Starting memory location of an array is called base address of an array. In C-Language array name itself refers base address of an array.

Fig.2.1 Array Terminologies



In the fig.2.1 memory representation of an integer single dimension array is shown. Array of 10 integer values are stored in the consecutive memory location starting 100 to 120. 100 is the starting address of the array is called base address. Because type of an array is integer and we know each integer occupies 2 bytes of space in the memory first data element of the array i.e. '5' will be stored from memory address 100 to 102. Second value 10 will be stored from 102 to 104 and so on. Index number of first data element 5 is 0 is called lower bound of the array and similarly last element is stored on position 9 is called upper bound of an array.

Let is we have an array having Lower bound LB and upper bound UB. Then the index number of i^{th} element will be always: $\text{Index}(X_i) = \text{LB} + i - 1$. For example, in C-Language index number of 5th element will be $0 + 5 - 1 = 4$.

Similarly, size of an array is: $\text{Size} = \text{UB} - \text{LB} + 1$. In C-Language if UB of an array is 9 the size = $9 - 0 + 1 = 10$.

The following Fig. 2.1 will explain all the terms discussed above. Array can be One – dimensional (Linear), Two – dimensional (Matrix) or Multi – dimensional. One – dimensional and Two – dimensional arrays are discussed below in details.

2.4 ONE-DIMENSIONAL ARRAY

One-dimensional array is an array which requires only one index or subscript number to reference any element stored in the array. Array presented in the Fig.2.1 is one-dimensional array. Value and Address of any element having index 'i', in the array 'X' can be found by following equations:

$$\text{Value of element can be accessed } X[i] \quad (1)$$

$$\text{Address of } X[i] = \text{Base address of an array} + i * \text{size of element} \quad (2)$$

For example, in the integer array whose base address is 100, address of element located on index number 4 is: $100 + 4 * 2 = 108$. Here 100 is the base address of an array, 4 is the index number of an element and because of the type of an array is integer size of element will be 2.

1.1 Operations on arrays

Different types of operations can be performed on the array such as, Traversing, Insertion, Deletion, Searching, Sorting, Merging etc.

Traversing:

Accessing each element of an array is called traversing operation. Consider the following program in which we are printing all the values stored by the user.

Program:2.1 Array Traversal

```
#include <stdio.h>
void main()
{
    int x[10]={2,9,11,28,34,45,57,78,83,90};
    int i;
    printf("\nArray X contains: ");
    for(i=0;i<10;i++)
    {
```

```

        printf("%d\t",x[i]);
    }
}

```

Output:

Array X contains: 2 9 11 28 34 45 57 78 83 90

In the above program we have declare an integer array 'x' having 10 integer values. Using variable 'i' and a for loop we have printed each element of the array. Accessing all the elements of the array for printing or any other calculation purpose is called traversing the array.

Insertion in the array:

Consider an array having some values are inserted and other cells are empty (having value 0). Now suppose user want to insert a new data element on potion 4. In this case first we copy 4th element into some variable (tmp). On the 4th position we will place, the number user wants to insert. Finally, we copy the value of tmp variable to num variable. The same process is repeated till end of the array. Mean on the 4th position, new elements will be inserted, 4th element will be placed on 5th position, 5th element will be placed on 6th position and so on. So, insertion, requires all elements to be shifted at right hand side.

Program 2.2 Insertion in the Array

```

#include<stdio.h>
void main()
{
    int arr[10]={5, 10, 15, 20, 25, 30, 0, 0, 0, 0};
    int num, i, tmp, pos;
    printf("Enter Position:");
    scanf("%d", &pos);
    printf("Enter Element:");
    scanf("%d",&num);
    printf("Array Before Insertion:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
    for(i=0;i<10;i++)
    {
        if(i>=pos -1)
        {
            tmp=arr[i];
            arr[i]=num;
            num=tmp;
        }
    }
}

```



```

    }
    printf("\nArray After Insertion:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
}

```

OUTPUT:

Enter Position:4

Enter Element:18

Array Before Insertion:

5 10 15 20 25 30 0 0 0 0

Array After Insertion:

5 10 15 18 20 25 30 0 0 0

Deletion in the array:

Deletion is the reverse process than insertion. When any value is deleted from the array then all the right-hand side elements are shifted to the left-hand side by 1 position. The program of deletion from an array is given in the following example.

Program 2.3 Deletion in the Array

```

#include<stdio.h>
void main()
{
    int arr[10]={5, 10, 15, 18, 20, 25, 30, 0, 0, 0 };
    int i, tmp, pos;
    printf("Enter Position:");
    scanf("%d", &pos);
    printf("Array Before Deletion:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
    for(i=0;i<9;i++)
    {
        if(i>=pos-1 )
        {
            arr[i]=arr[i+1];
        }
    }
    arr[9]=0;
    printf("\nArray After Deletion:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
}

```

OUTPUT:

Enter Position:4

Array Before Deletion:

5 10 15 18 20 25 30 0 0 0

Array After Deletion:

5 10 15 20 25 30 0 0 0 0

Deletion in the array:

In the case of searching, user will input the element and we have to compare each element of an array (starting from the position 0) with the element entered by the user. Whenever, we find the exact match we have to stop the process and the position of that element into the array is displayed to the user. The process of searching an element in the array by comparing each element of the array (starting from 0) is called Linear search.

Program 2.4 Searching an element in the Array (Linear search)

```
#include<stdio.h>
void main()
{
    int arr[10]={5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };
    int i, num;
    printf("Enter Search Element:");
    scanf("%d", &num);
    printf("Array:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
    for(i=0;i<10;i++)
    {
        if(arr[i]==num )
        {
            printf("\nElement Found on: %d position", i+1);
            break;
        }
    }
    if(i==10)
    {
        printf("\nSearch element do not exists in the Array ");
    }
}
```

OUTPUT:

Enter Search Element:25

Array:

5 10 15 20 25 30 35 40 45 50

Element Found on: 5 position

Sort an array:

Sorting is the process of arranging array elements either in the ascending (from lower to higher) or descending (from higher to lower). Consider the following program, to sort elements of an array in the ascending order:

Program 2.5 Sorting an array

```
#include<stdio.h>
void main()
{
    int arr[10]={5, 23, 78, 90, 30, 76, 28, 20, 48, 2 };
    int i, j, tmp;

    printf("Array Before Sorting:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
    for(i=0;i<10;i++)
    {
        for(j=i;j<10;j++)
        {
            if (arr[j]<arr[i])
            {
                tmp=arr[i];
                arr[i]=arr[j];
                arr[j]=tmp;
            }
        }
    }
    printf("\nArray After Sorting:\n");
    for(i=0;i<10;i++)
        printf("%d\t", arr[i]);
}
```

OUTPUT:

Array Before Sorting:

5 23 78 90 30 76 28 20 48 2

Array After Sorting:

2 5 20 23 28 30 48 76 78 90

The algorithm we have used to sort an array is called selection sort. Other sorting algorithms are also there like Bubble sort, insertion sort etc. We will discuss them in the separate chapter 'Searching and Sorting' in the 'Data Structures'.

Here we are sorting elements of an array in the ascending order. If you wish to sort them in the descending order then change the if condition we have written in the program from: *if (arr[j]<arr[i])* to *if (arr[j]>arr[i])*.

2.5 TWO-DIMENSIONAL ARRAY

In the above examples we have declared array like: 'int x[10];'. When we declare array like this, then array 'x' has 10 rows or we can say the array is one-dimensional (rows). But if we declare array like: 'int y[3][3];' then array 'y' has 3 rows and in each row, 3 columns are there. That means array 'y' can store 9 elements. In

this case we can say that array 'y' is 2-dimensional array (rows, columns). To represent matrix, we need a 2-dimension array. 2-dimesional array can be represented as follows:

Figure 2.2 Representation of two-dimensional array

Columns →	0	1	2
Row 0	1 [0][0]	2 [0][1]	3 [0][2]
Row 1	4 [1][0]	5 [1][1]	6 [1][2]
Row 2	7 [2][0]	8 [2][1]	9 [2][2]

In the above array we have stored the values 1, 2, 3, ...9. Value 1 can be accessed by `y[0][0]`. Similarly 2 can be accessed as `y[0][1]`, 3 can be accessed by `y[0][2]`, 4 can be accessed by `y[1][0]` and so on. Usually two-loops (nested) are used to access matrix (2-dimnssional array). Consider the following program, which will do the addition of 2 matrices (2-D Arrays) of size 3*3.

Program 2.6 Addition of 2 Matrices

```
#include<stdio.h>
void main()
{
    int x[3][3], y[3][3],z[3][3];
    int i,j;
    printf("Enter Elements for Matrix1:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("Enter value for X[%d][%d]:",i,j);
            scanf("%d",&x[i][j]);
        }
    }
    printf("Enter Elements for Matrix2:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("Enter value for Y[%d][%d]:",i,j);
            scanf("%d",&y[i][j]);
        }
    }
}
```

```

    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            z[i][j]=x[i][j]+y[i][j];
        }
    }
    printf("Matrix X:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\t",x[i][j]);
        }
        printf("\n");
    }
    printf("Matrix Y:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\t",y[i][j]);
        }
        printf("\n");
    }
    printf("Matrix Z:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\t",z[i][j]);
        }
        printf("\n");
    }
}

```

OUTPUT:

Matrix X:

```

1  2  3
4  5  6
7  8  9

```

Matrix Y:

```

11 12 13
14 15 16
17 18 19

```

Matrix Z:

```

12 14 16
18 20 22
24 26 28

```

2.6 STRINGS

Set of characters is called string. In many high-level programming language data type string is available. Unfortunately, C-language do not have such data type. But as we have seen in the previous topic, that more than one element can be stored in the array. Here, we will do the same thing. We will take a character array, which is able to store one or more characters or we can say simply string. Refer the following program in which we are accepting string from the user, and printing it on the console.

Program 2.7 Accepting sting from the user and printing it on the console

```
#include<stdio.h>
void main()
{
    char str[10];
    printf("Enter Any String:");
    scanf("%s",str);
    printf("String the you have entered is: %s");
}
```

OUTPUT:

```
Enter Any String:BAOU University
String the you have entered is: BAOU
```

One method is to accept the string is, use scanf() function with "%s" format specifier. But in this method string will accept space. When we have entered string "BAOU University", it stores only "BAOU" in the character array str. If we want to store the string with spaces then we have use gets() and puts() function. The same program is written in the next example, but instead of printf() and scanf() function we are using puts() and gets() functions.

Program 2.7 Accepting and printing string using gets() and puts() functions

```
#include<stdio.h>
void main()
{
    char str[10];
    printf("Enter Any String:");
    gets(str);
    printf("String the you have entered is:");
    puts(str);
}
```

OUTPUT:

```
Enter Any String:BAOU University
String the you have entered is: BAOUUniversity
```

As we have seen, the input and out of the above program is same. That means, gets() and puts() function accept space. If you want to accept the space and enter also in the string then we have accepted one by one character and put it in the array, unless user is not entering EOF. To input EOF value user have to press Ctrl + Z keys.

Program 2.8 Accepting string till EOF

```
#include<stdio.h>
void main()
{
    char str[100], ch;
    int i=0;
    printf("Enter Any String:\n");
    while((ch=getchar())!=EOF)
    {
        str[i]=ch;
        i++;
    }
    str[i]=EOF;
    printf("\nString you have entered is:\n");
    i=0;
    while((ch=str[i])!=EOF)
    {
        printf("%c",ch);
        i++;
    }
}
```

OUTPUT:

Enter Any String:
 C-Language is most popular
 general purpose programming language.
 I like C-Language.
 ^Z(Press Ctrl + Z keys Here)

String you have entered is:
 C-Language is most popular
 general purpose programming language.
 I like C-Language.

2.7 STRING FUNCTIONS

C-language supports large number of string functions available in the string.h header file. The functions are listed in the following table.

Functions	Description
int strlen(char *)	To find the length of the sting
strcpy(char *destination, char *source)	To copy source string to the destination

	array.
strncpy(char *destination, char *source, int n)	To copy n character from source to destination
int strcmp(char *str1, char *str2)	If str1 > str2 then it returns 1 If str2 > str1 then it returns -1 If str1=str2 then it returns 0
int stricmp(char *str1, char *str2)	Comparing two string same as strcmp by ignoring case (Here 'A' and 'a' are same)
int strncmp(char *str1, char *str2, int n)	Comparing two string str1 and str2 up to n characters, and returns integer value same as strcmp() function
strlwr(char *str)	To convert string in the lower case
strupr(char *str)	To convert string in the upper case
strcat(char *str1, char *str2)	To concatenate two strings. Str2 is appended in the string str1.
strrev(char *)	To reverse the given string

consider the following examples to learn how the functions discussed above can be used into the C-programs:

Program 2.9 Finding length of the string

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str1[10];
    int i;
    printf("Enter String:");
    scanf("%s",str1);
    i=strlen(str1);
    printf("Length of the string is %d",i);
}
```

OUTPUT:

```
Enter String:India
Length of the string is 5
```

In the next program we will try to check whether given string is palindrome or not. A string is palindrome if the string and its reverse sting is same.

Program 2.10 Check whether given string is palindrome or not

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str1[10],str2[10];
    int i;
    printf("Enter String:");
```



```

scanf("%s",str1);
strcpy(str2,str1);
strrev(str2);
i=strcmp(str1,str2);
if(i=0)
    printf("Given string is Palindrome");
else
    printf("Given string is not Palindrome");
}

```

OUTPUT:

Enter String:madam

Given string is Palindrome

Consider the next example in which, we have excepted string from lower case and converted it into the upper case without using string.h header file. To do this, we have to know that “how string ends?”. When user input the string, for example user has entered “abcd” then in the array string stored like “abcd\0”. ‘\0’ is a special character called NULL character. Here in the array on the 0th position character ‘a’, on the 1st position character ‘b’ and in the same way on the 4th position NULL character ‘\0’ is being stored. To convert the character from lower to upper, we need to subtract 32 from each character. Because the ASCII difference between lower case letters and uppercase letters is 32.

Program 2.11 Convert the lower-case letter string to upper-case

```

#include<stdio.h>
void main()
{
    char str[10];
    int i;
    printf("Enter String:");
    scanf("%s",str);
    for(i=0;str[i]!='\0';i++)
        str[i]=str[i]-32;
    printf("Upper case string is:%s",str);
}

```

OUTPUT:

Enter String:baou

Upper case string is:BAOU

Array of Strings:

We know that the string itself is an array of characters. Here we are discussing array of strings. That can be managed by taking 2-dimentional array of type character. If

we want to store 3 strings in the array we need to declare it like 'char strs[3][10]'. Array 'strs' can store 3 strings, can have maximum length of 10 characters. That can be represented in the following form:

A	B	C	\0						
X	Y	Z	\0						
B	A	O	U	\0					

Excercise:1 Fill in the blanks:

1. String ends with _____ character.
2. Array index starts with _____.
3. To convert the lower-case string into the upper-case string, _____ string function is used.
4. To reverse the given string _____ string function is used.
5. strlen() function is available in _____ header file.
6. The ASCII difference of lower-case letters and its upper-case letters is _____.
7. Array of string can be represented by _____.
8. Matric of 3*3 can be represented by _____.
9. In the array of size 10, index number of last element will _____.
10. To compare n characters, from two stings _____ function is used.

2.8LET US SUM UP

In this chapter we have learnt how to store multiple elements in the array. We have learnt one-dimensional and two-dimensional arrays, operations that can be performed in the array. We have also learnt how stings can be stored busing array. In this chapter we have also focused on different types of string functions and their uses. We hope student now have sufficient knowledge of the different kind of arrays.

2.9CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 2.1

- 1 \0
- 2 0
- 3strupr()
- 4strrev()

5 string.h

6 32

7 two-dimension array

8 two-dimension array

9 9

10 strncmp()

Unit 3: Structures and Unions

Unit Structure

- 3.1. Learning Objectives
- 3.2. Introduction
- 3.3. Structures
- 3.4. Array of structures
- 3.5. Nested structures
- 3.6. Unions
- 3.7. Let's sum up
- 3.8. Check your Progress: Possible Answers

3.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand keywords
- Declare and use variables and constants
- Understand different datatypes their size and range
- Use of various operators in the expression

3.2 INTRODUCTION

Consider following program:

```
#include<stdio.h>
void main()
{
    int x;
    x=50;
    x=28;
    printf("The value of variable X is = %d",x);
    getch();
}
```

Output:

The value of variable X is=28

In the above program when we have declared variable x, we are initializing it by value 50. Then we have written 'x=28'. This statement will change (overwrite) the value of x by 50 to 28. Finally, when we are printing the value of x, it will print 28. So, we can say that one variable can store only one value. If we are trying to assign second value to it, it will overwrite the older value.

What happened if we want to store many values in the one variable? We have to take an array. 'int x[10]; ' This statement will declare array (single variable) x, which is long enough to store 10 values. But the drawback, of the array is all elements stored in the array have same data type. As we have discussed in the above chapter 'Array is homogenous (same type of data) collection. Here in the array x, we can store 10 integer numbers only.

What happened, if we want to store different data element to single variable? For example, if we want to store the details of the students in which 'roll no' is

integer, 'Name' is character array, and 'height' of the student in float. Obviously, one array cannot store these data items, which are of different data types.

In this case, we can define our own data type, which is able to store all the data of different data types. This data type is known as user defined data type. To design user defined data type 'Structure' is used.

3.3 STRUCTURES

Structure is known as user defined Data Type. A 'Structure' can be defined as a collection of related entities that may be of different data types accessed using a common name. Array is also a collection of data elements. But the data elements of the array are of same type (homogeneous), where structure is a collection of data elements of different types (heterogeneous).

So, 'Structure' is nothing but group of data items stored as a single entity. Each data item of a structure is called field. A field is the smallest meaningful element, which has a name and storage requirement. Field can also be referred as data members of the structure.

Defining Structure:

To use a structure, we first need to define a structure template. The template identifies the name and members of the structure. We can define the structure by using keyword 'struct'.

```
struct struct_name
{
    Datatype member1;
    Datatype member2;
    :
    :
    Datatype member;
};
```

In the syntax given above, the keyword 'struct' is followed by the name of the structure. The name of the structure is an identifier, and that can be treated as a user defined data type. By using that user can create the variables of this (user defined) data type. The variables to be grouped together in a structure are placed between

curly braces. The structure template is terminated by semi colon (;) at the end of the closing curly brace.

Example:

```
struct student
{
    int rollno;
    char name[10];
    float height;
};
```

In the example given above 'student' is the name of the structure. 'rollno', 'name' and 'height' are the member of this structure.

Structure is user defined data type. Data type never being used to store value, because it does not occupy space in the memory. So, we cannot assign value to the data member of the structure directly. If we want to store the value then, we need to create the variable(s) of it. For example, we cannot store the value in int; '**int=5**' is an invalid statement. But we are creating variable of 'int' and then, in that variable we are storing the value. For example, '**int x; x=5;**' are valid statements.

Accessing structure variables:

Once the structure template is defined, it can be used as a data type (user defined data type). The syntax, to declare a variable of structure is as follows.

```
struct struct_name variable_name;
```

For example,

```
struct student x,y;
```

Here x and y are the variable of type 'struct student'. Each variable will occupy 16 Bytes of space in the memory. 2 Bytes for rollno (int), 10 bytes for name (char array), and 4 bytes for height (float).

Operations on structure variables:

Various operations can be performed on the structure variable like, initializing structure variable by initializing its data members, copying one structure variable to another, comparing structure variables etc.

[1] Initializing structure variable:

After creating structure variable, we have initialized its data member, and also have to access the data members. To access data element of the structure variable (.) dot-operator is used.

For example, if we want to store height of the student x then we can write:

Syntax:

```
structure_variable.data_item_of_structur=value;
```

Example:

```
x.height=4.9;
```

In the similar way if we want to print the height of student 'X' then we can write:

```
printf("%.2f",x.height);
```

Program 3.1 Initializing structure variables

```
#include<stdio.h>
#include<string.h>
struct student
{
    int rollno;
    char name[10];
    float height;
};
void main()
{
    struct student x,y;

    /* Details of student x */

    x.rollno=51;
    strcpy(x.name,"ABC");
    x.height=5.2;

    /* Details of student y */

    y.rollno=52;
    strcpy(y.name,"XYZ");
    y.height=4.9;

    /* Printing Details of both students */

    printf("\nStudent X:");
    printf("\n-----\n");
    printf("Roll No:%d\nName:%s\nHeight:%.2f",x.rollno,x.name,x.height);
    printf("\n\nStudent y:");
```

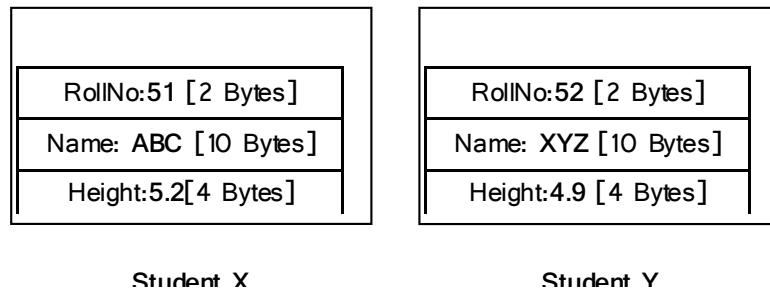


```

printf("\n-----\n");
printf("Roll No:%d\nName:%s\nHeight:%.2f",y.rollno,y.name,y.height);
}

```

In the above program we have created 2 variables x and y. Both are of same type 'struct student' and they will occupy 16 Bytes of space in the memory as shown below:



[2] Copying structure variable :

Consider the following program, in which we have declared two variables of type structure student, that are x and y. we are initializing the value for student x. Now, if the data of the student y is similar to x, we can simply write y=x. This operation will copy all the data of structure variable x to another structure variable y.

Program 3.2 Copying structure variables

```

#include<stdio.h>
#include<string.h>
struct student
{
    int rollno;
    char name[10];
    float height;
};
void main()
{
    struct student x, y;
    x.rollno=1;
    strcpy(x.name,"abc");
    x.height=4.9;

    y=x;

    printf("\nRollno of Student Y is:%d", y.rollno);
    printf("\nName of the Student Y is:%s", y.name);
}

```

```

        printf("\nHeight of the Student Y is:%.2f", y.height);
    }

```

In the above example we are copying the data of student x to student y.

[3] Comparing structure variables:

Structure variables cannot be compared directly, but can be compared by comparing its data elements for example, if we have two variables x and y of type 'struct student' we can not compare them directly by writing the statement '*if(x==y)*', but we can compare them by comparing their data elements like '*if(x.height==y.height)*' or '*if(x.height >y.height)*'. Consider the following example:

Program 3.3 Comparing structure variables

```

#include<stdio.h>
#include<string.h>
struct student
{
    int rollno;
    char name[10];
    float height;
};

void main()
{
    struct student x, y;
    x.rollno=1;
    strcpy(x.name,"abc");
    x.height=4.9;
    y=x;
    if(x.height==y.height)
        printf("Variables X and Y are Equal");
    else
        printf("Variables X and Y are not Equal");
}

```

OUTPUT:

Variables X and Y are Equal

3.4 ARRAY OF STRUCTURE

If we need many variables (of same type) then normally, we are taking array. For example, '*int x[10];*'. This statement will declare array 'x', which can store 10 integer numbers in it. Here x is an array of primitive data type (int). If we want to create an array of structure (user defined data type), then it is also possible. The syntax is as follows:

```
struct student x[5];
```

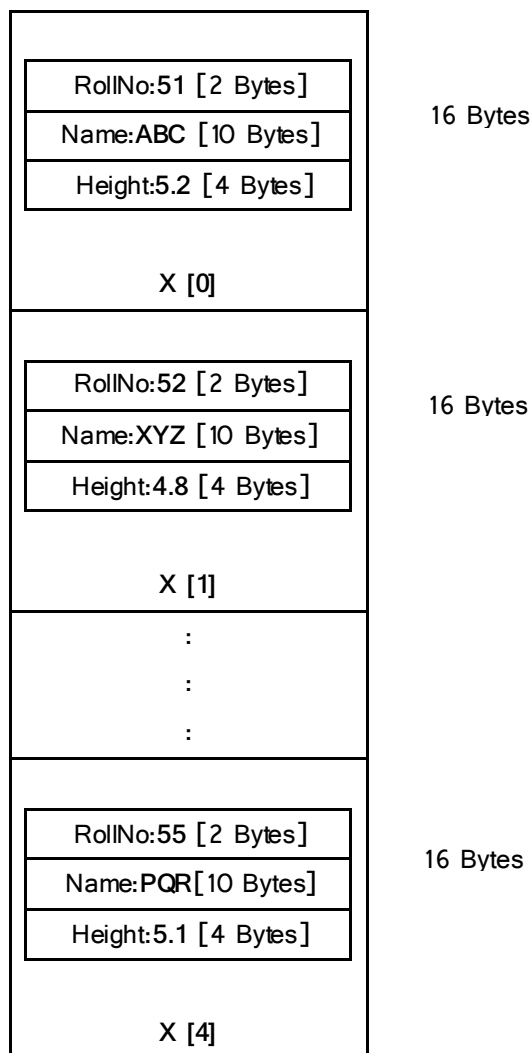
Here x is an array which can store the details of 5 students. The size of x is 80 Bytes (16Bytes for 1 student and there are 5 students). Because of array, we can access the data of the student by using index number of an array. For example, first student will be represented like x[0], Second student will be represented like x[1] and so on.

Now the specific details of the student (data members) can be accessible like:

```
x[0].roll no=51;           //This will set roll no of the first student by 51
x[0].height=5.2;         //This will set height of the first student by 5.2
x[1].rollno=52           //This will set roll no of the Second student by 52
```

Similarly, we can print the data of the first student like:

```
printf("Roll No:%d",x[0].rollno);
printf("Name of the Student:%s",x[0].name);
printf("Height of the Student:%.2f",x[0].height);
```



Consider the following program, in which the details of five students are stored in the array. Using loop, we are accepting values for all the students and also displaying the student details on the console, from the array using loop.

Program 3.3 Comparing structure variables

```
#include<stdio.h>
struct student
{
    int rollno;
    char name[10];
    float height;
};
void main()
{
    struct student x[3];
    int i;
    for(i=0;i<3;i++)
    {
        printf("\nEnter the Details of Student:%d",i+1);
        printf("\nRollNo:");
        scanf("%d",&x[i].rollno);
        printf("Name:");
        scanf("%s",x[i].name);
        printf("Height:");
        scanf("%f", &x[i].height);
    }
    printf("\n RollNo  Name  Height");
    printf("\n-----");
    for(i=0;i<3;i++)
        printf("\n%d\t\t%s\t%.2f",x[i].rollno,x[i].name,x[i].height);
}
```

OUTPUT:

```
Enter the Details of Student:1
RollNo:1
Name:ABC
Height:6.2
```

```
Enter the Details of Student:2
RollNo:2
Name:XYZ
Height:5.4
```

```
Enter the Details of Student:3
RollNo:3
Name:PQR
Height:5.8
```

```
RollNo Name Height
-----
```

1	ABC	6.20
2	XYZ	5.40
3	PQR	5.80

Now, consider the following c-program in which, we have an array of the students. Here, we take rollno, name and marks of 3 subjects. Obviously, to accept the marks for 3 different subject we have taken an array. Finally, after accepting data from the student program will calculate the total and print it on the screen. In this example, we have taken an array of marks inside the structure, and we have also taken an array of the students (structure).

Program 3.4 Array inside the structure and also array of structures

```
#include<stdio.h>
struct stu
{
    int rollno;
    int m[3];
    int total;
};

void main()
{
    struct stu x[5];
    int i,j;
    for(i=0;i<5;i++)
    {
        printf("\nEnter Roll No:");
        scanf("%d",&x[i].rollno);
        printf("\nEnter Marks for 3 Subject");
        x[i].total=0;
        for(j=0;j<3;j++)
        {
            printf("\nStu[%d]-Subject[%d]",i+1,j+1);
            scanf("%d",&x[i].m[j]);
            x[i].total=x[i].total+x[i].m[j];
        }
    }
    printf("\nRollNo Marks");
    printf("\n/-----/\n");
    for(i=0;i<5;i++)
    {
        printf("\n%d \t\t %d",x[i].rollno,x[i].total);
    }
}
```

3.5 NESTED STRUCTURES

Sometime we need to take structure in side a structure. When we have structure within a structure, then it is called a nested structure. For example, higher level languages have date as datatype, but in the C-language, we don't have date datatype. If we want to store the details of the employee with data members employee code, name, data of birth and date of joining, then date of birth and date of joining can be a variable of type date, which consists of day, month and year. In the program we have structure employee and inside that structure we have a structure date.

Program 3.5 Example of nested structure

```
#include<stdio.h>
struct employee
{
    int empcode;
    char name[10];
    struct date
    {
        int dd;
        int mm;
        int yy;
    }dob, doj;
};
void main()
{
    struct employee x;
    x.empcode=1;
    strcpy(x.name,"ABC");
    x.dob.dd=22;
    x.dob.mm=8;
    x.dob.yy=1976;
    x.doj.dd=15;
    x.doj.mm=7;
    x.doj.yy=2000;
    printf("Employee Details:");
    printf("\nEmployee Code:%d",x.empcode);
    printf("\nName:%s",x.name);
    printf("\nDOB:%d-%d-%d",x.dob.dd,x.dob.mm,x.dob.yy);
    printf("\nDOJ:%d-%d-%d",x.doj.dd,x.doj.mm,x.doj.yy);
}
```

OUTPUT:

Employee Details:

Employee Code:1

Name:ABC

DOB:22-8-1976
DOJ:15-7-2000

Let if we have an array of the structure 3.5 of 5 employees, that is 'struct employee x[5]' that what will be size of x? compute it. Here variable of type date occupies 6 bytes of space (2 bytes for dd, 2 bytes for mm and 2 bytes for yy because of dd, mm and yy all are of type int). Therefore, both variables dob and doj occupies 6 bytes of space. Empcode needs 2 bytes and name occupies 10 bytes. Therefore, one employee needs (2 bytes for Empcode, 10 bytes for name, 6 bytes for dob, 6 bytes for doj) =24 Bytes. If we have 5 employees then we need $24*5=120$ Bytes of space in the memory for our array 'x'.

3.6 UNIONS

Union is a concept; similar to structure and therefore follows the same syntax as structure. Union is also called user defined data type, same as structure. However, there are major distinctions between them in terms of storage. In structures, each member has its own storage location, whereas all the members of the union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Similar to the structures, a union can be declared using the keyword **union** as follows”

```
union abc
{
    int a;
    float b;
    char c;
} x;
```

This declares a variable x of type union abc. The union contains three members, each with different data type. However, we can use only one of them at a time. This is due to the fact, that only one location is allocated for a union variable, irrespective of its size.

The compiler allocates a piece of storage that is large enough to hold the largest variable in the union. In the declaration above, the member b requires 4 bytes which is the largest among the members. Here variable a, b and c shares the 4 bytes of space.

To access union members, we can use the same syntax that we use for structure variable that is variable_name.datamember_name.

Here, union variable x will store 4 bytes of space. Because the maximum space required to store data is 'b' which is of type float (4 bytes).

In the case of structure, it will occupy 2 bytes for 'int a' + 4 bytes for 'float b' + 1 byte for 'char c' = 7 bytes.

So, in the structure we can store the values in a, b, and c at the same time. In the case of union, because of we are reserving only 4 bytes of space, only one data item we can store at a time, that can be anything from int, float or char.

There are a smaller number of applications are there of union compare to the structure. Union variable occupies less memory than structure, on other hand only one data element can be initialized. Structure variable occupies more memory than union, but it allows to initialize all of its data member.

Exercise: 3.1 Fill in the blanks:

1. To access data members of the structure variable _____ operator is used.
2. The variable of _____, occupies memory, is sum of memory occupies by its data member.
3. In the _____ only one data member can be initialized at a time.
4. Variable of type _____ occupies memory is same, as the memory occupies by its largest element.

3.7LET US SUM UP

In this chapter we have studied how can we construct our data types. We have learnt how structure and union can be declared, initialized and use into our C-programs. We have also learnt array of the structure and nested structure. Structure is heterogenous data structure, and useful in the upcoming chapters of data structures. I hope the discussion we done in this chapter will help a lot, to the student to understand chapter of data structures.

3.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 3.1

6. . (dot operator)
7. structure
8. union
9. union
10. 0 to 255

Unit 4: Pointers

4

Unit Structure

- 4.1. Learning Objectives
- 4.2. Introduction
- 4.3. What is Pointer?
- 4.4. Pointers on array
- 4.5. Pointers on structures
- 4.6. Dynamic memory allocation
- 4.7. Returning multiple values from the function
- 4.8. Let us sum up
- 4.9. Check your progress: Possible answers

4.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know what is pointer? How to declare and initialize it.
- Understand how pointer works on arrays
- Use of pointer variable on structure variable
- Know different functions of dynamic memory allocation
- Returns multiple values from the function using pointer
- Advantages of the pointer variable

4.2 INTRODUCTION

As we have discussed in the chapter 'Working with the Functions', there are two types of variables are there [1] Local variables and [2] Global variables. The variable declared inside the function, is a local variable, and we can access that within that function only. Directly, it is not possible to access those variables which are declared in some other functions. Consider the following example, in which the function is designed to swap two variables which is declared in the main() function.

```
#include<stdio.h>
void swap(int, int); //Prototype declaration
void main()
{
    int a=5,b=7;
    swap(a,b); //Calling function swap to swap the values of variable a and b
    printf("\nA is:%d",a);
    printf("\nB is:%d",b);
}
void swap(int x, int y) // Functions Swap()
{
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

In the above program, we have declared two variables, 'a' and 'b' in the main() function. We want to swap these variables. So that, we have created 'swap' function. This function takes two arguments.

When main () function is calling to the swap function, swap function creates two variables, 'x' and 'y' and the values passed by the 'main()' function (value of variable 'a' and 'b') is being copied in to variable 'x' and 'y' respectively.

Logic which is written inside 'swap()' function will swap the value of variable 'x' and 'y'. After execution of the 'swap ()' function, the function 'swap ()' will be removed from the memory and all the local variables of it will be deleted from the memory. So, we get the following output.

Output:

A is: 5

B is: 7

Value of variables a and b remains unchanged even after calling 'swap ()' function.

So, we can say the value of the local variable of one function cannot be changed by another function. If we want to change the value of the local variable, of one function, by another function, we need a pointer. In the above example, we are passing values of the variables, to the function 'swap ()'. Instead of the values, now will pass the address (References) of the variables to the 'swap ()' function. For that, we need to learn the concepts of address of the variable and pointer.

4.3 WHAT IS POINTER?

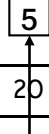
Before discussing pointer, we should know 'what is an address of the variable?'. We know that, when we declare the variable, it reserves space in the main memory that is Random Access Memory (RAM). Main memory of the system can be considered as an array of cells (memory words). As we have seen in the chapter of 'Arrays', the values stored in the array can be accessed by index number. Similarly, in the case of main memory each cell on the memory has unique number, which called the address or reference of the memory location.

We have tried to show the structure of the main memory. It looks like:

1	6	11	16	21	26	31	36
2	7	12	17	22	27	32	37
3	8	13	18	23	28	33	38
4	9	14	19	24	29	34	39
5	10	15	20	25	30	35	40

Each cell in the table can store the data, and the unique number represents memory address (reference). Now, when we declare any variable of type 'int', we know that it needs 2 Bytes of space in the main memory. Operating system here, check the memory and allocate 2 consecutive bytes of memory to the variable. Make sure the memory will be given from that memory space which are unallocated (not allocated to any running process) or free. This is same as in a running class if new student, comes then that student reserves his/her sit from the set of available (free) seats. Newly arrived student can not acquire that seat which already acquire by some student. Now, think every seat has some unique number is written. Then that, seat number is called reference or address. The person who sits in the chair is called value. In the same manner when we declare a variable of type int, the 2 Bytes of space will be reserves in the main memory and when we initialize it, initialization value will be written in that space. So, after execution of the statement 'int x = 5;' memory looks like:

1	6	11	16	21	26	31	36
2	7	12	17	22	27	32	37
3	8	13	18	23	28	33	38
4	9	14	5	24	29	34	39
5	10	15	20	25	30	35	40



 Variable x is declared here

As, we can see the variable 'x' reserves the space on cell 18, and in the memory cell 18 has value 5, we can say the 5 is the value of variable x and 18 is the address of the variable x.

Pointer:

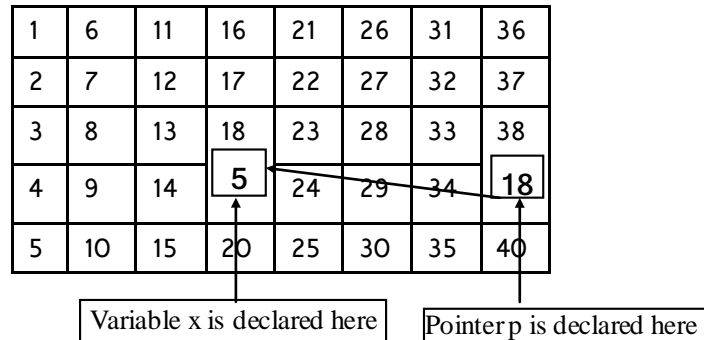
Pointer is a special variable, which can store the address of another variables.

So, pointer is nothing it is a variable. Pointer is that variable which is able to store the address of another variable. So, now basically we have two types of variables, normal variables are used to store the value, and pointer variable are used to store memory addresses. But how system can distinguish both variable? the answer is when we declare a pointer variable we will use *.

```
int x; // normal or regular variable, used to store value or data
int *p; // pointer variable, used to store address
```

Now, how can we store the address, in the pointer variable? The answer is we can find the address of any variable using '&' operator (recall we have already used & operator to provide the address of our variable to scanf () function) . If we want to store the address of variable x in the pointer variable p, then:

```
p=&x; //Initializing pointer variable. We are storing address of variable x in the pointer p
x=5; //Initializing normal variable x with value 5.
```



Now, you can see from the above figure that variable x has value 5 and address 18. Because of pointers are the variable, it also reserves space in the memory. Our pointer variable p is created on address 38, so we can say that the 38 is the memory address of the pointer variable p. We have written p=&x; statement, that stores address of x that is 18 in to the pointer variable p. So, pointer p has value 18, which an address of variable x, we can say that pointer p is pointing to the variable x. The addresses shown in the figure are just assumption and addresses can defer system to system. Now, consider the following program:

Program 4.1 Normal variable and pointer variable

```
#include<stdio.h>
void main()
{
    int *p, x; // x is normal and p is pointer variable
    x=5;
    p=&x;
    printf("\n Value of variable X is: %d",x);
    printf("\n Address of variable X is: %u",&x);
    printf("\n Value in the pointer variable P is: %u", p);
    printf("\n Address of the variable P is: %u", &p);
    printf("\n Pointer P is pointing to: %d", *p);
}
```

OUTPUT:

```
Value of variable X is: 5
Address of variable X is: 6356744
```

Value in the pointer variable P is: 6356744
Address of the variable P is: 6356748
Pointer P is pointing to: 5

To print the address of the variable, %u format string is used in the printf () statement. The meaning of *p in the last printf statement is value stored at the address, stored in pointer variable p. Variable p has address 6356744(an address of variable x) and value stored at this address is obviously 5 (value of variable x).

Now, after this discussion, we turn back to our original problem, that is deriving a function to swap two local variables of the main () function.

Program 4.2 Swapping values of local variable

```
#include<stdio.h>
#include<stdio.h>
void swap(int, int); //Prototype declaration
void main()
{
    int a=5,b=7;
    swap(&a,&b); //Calling function swap to swap the addresses of variable a and b
    printf("\nA is:%d",a);
    printf("\nB is:%d",b);
}
void swap(int *x, int *y) // Swap() accepts addresses of variable a and b into pointers x and y
{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;
}
```

OUTPUT:

A is: 7
B is: 5

Now, in the above program we have made a small change, compare to our previous program of swap. From the main () function, rather than passing the values, we are passing the addresses of variable a and b. We know that, to store the address we need pointer variables. So, we have taken pointer arguments for function swap ().

4.4 POINTERS ON ARRAYS

We know that the pointer is a special variable which can store the address of another variable. We also know that the array name itself represents base address

of an array. We can store the base of an array into the pointer variable, and can access all the elements of an array through the pointer consider the following example in which we have declared an array locally in the main () function. From the main function, we are calling a display () function, and we are passing array name (base address of an array) to the display () function. Function display () then access all the elements of an array and print them on the console.

Program 4.2 Swapping values of local variable

```
#include <stdio.h>
void main()
{
    int x[10]={5, 10, 15, 20, 25, 30, 35, 40, 45, 50};
    display(x); //Passing the base address of the array X
}
void display(int *p)
{
    int i;
    for(i=0;i<10;i++)
    {
        printf("\n%d",p[i]);
    }
}
```

In this program function main () is calling display () function, and passes the base address of the array x, which is locally declared in the main () function. Because we are passing address of array x, and address can be stored in the pointer variable, we have taken pointer p as an argument for display function. Now p has same value of as x (base address), so we can access all elements of x using pointer variable p. Now consider another example where we are passing string (base address of a character base array) to the function upper (), and it is changing all the letters from lower-case to upper-case.

Program 4.3 Program to change the case from lower to upper.

```
#include <stdio.h>
void upper (char *); //prototype declaration
void main()
{
    char name[10];
    printf("Enter Your University Name in Lower-case:");
    scanf("%s",name);
    upper(name); //passing the base address of array name
    printf("Your University name is: %s", name);
}
void upper(char *p)
{
```



```

while (*p!=\0')
{
    *p=*p-32;
    p++;
}
}

```

OUTPUT:

Enter Your University Name in Lower-case: baou
Your University name is: BAOU

In this example, we are passing the base address of array name at the time of calling function upper () function. Because we are passing address of the character array and address can be stored in the pointer variable, we have taken pointer of character. For each character stored in the array we are subtraction 32. When the ASCII value is decreased by 32, then that lower-case letter will be converted to upper-case. *p = *p - 32 statement will decrease the value by 32, while p++ statement gives a jump of 1byte to the pointer variable p, so that we can access another character. If we declare int *p, then p++ statement gives a jump of 2 Bytes to the pointer variable p.

4.5 POINTER ON STRUCTURE

We know that, the data members of the structure variable can be accessible by dot operator (.). But we you have a pointer variable of the structure having address of some structure variable, then using pointer variable you can also access all the data members of the structure. In this case instead of using (.) dot operator, you need to use (->) point-to operator. Consider the following example:

Program 4.3 Program to change the case from lower to upper.

```

#include<stdio.h>
struct student
{
    int rollno;
    float marks;
};
void main()
{
    struct student s; // Variable of structure
    struct student *p;// Pointer variable of structure
    p=&s;           // Initializing pointer variable to address of s
    p->rollno=51;  // Accessing data members of structure using pointer
}

```

```
p->marks =78.90;
printf("\nRoll No: %d", s.rollno); //Accessing data element using variable name
printf("\nMarks: %.2f", s.marks);
}
```

OUTPUT:

Roll No: 51

Marks: 78.90

So, always remember that if you have a variable name, though variable name you want to access data member of the structure the (.) dot operator is used, and if you have pointer of structure variable and you want to access data members of structure variable then (->) point to operator is used.

4.6 DYNAMIC MEMORY ALLOCATION

Up to here whatever programs we have done, that are based on some algorithm. By considering the algorithm, we will come to know, how many variables we have to declare? to implement the logic. But some time, as a programmer we have to write a program, in which how many variables has to be declared?is not known, in advance. Here we have to write a program, that is able to declare the variables, as it is needed. This type of variable declaration is known as dynamic memory allocation.

Whenever we are declaring a variable, memory allocation is done. By mean of memory allocation, memory is allocated to that variable in the primary memory. For example, when we write 'int x;', 2 Bytes of memory allocated to the variable 'x'. This is called static memory allocation. In the case of static memory allocation, complier reserves the space for all variables, we are using in the program first, and then the execution of the program starts. In the case of dynamic memory allocation, program starts it execution first and when needed, it reserves the space for the variable.

To allocate memory to the variable dynamically, we have to use some functions, which is available in the 'alloc.h' header file (In the case of UNIX based software or open source C-Editor 'stdlib.h' is the header file). So, it is essential to include 'alloc.h' / 'stdlib.h' header file, while we are using following functions of dynamic memory allocation.

malloc()

This function allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

Syntax:

```
ptr = (Cast_type *) malloc (byte-size);
```

Example:

```
int *p = (int *) malloc (sizeof (int));
```

calloc()

This function allocates space for an array of elements, initialize them to zero and then returns a pointer to the starting address of the array.

Syntax:

```
ptr = (Cast_type *) calloc (no_of_elements,byte-size);
```

Example:

```
int *p = (int *) calloc (10,sizeof (int));
```

In this example, calloc() function will declare integer array of size 10, and it will return the starting address of the array, that we are storing in the pointer p.

realloc()

This function is used to modify the size of previously allocated space.

Syntax:

```
ptr = realloc (ptr , new-byte-size);
```

free()

This function is used to free the memory, to allocated space previously by using malloc() function.

Syntax:

```
free(ptr);
```

Example:

```
int *p = (int *) malloc (sizeof (int));  
free(p);
```

4.7 RETURNING MULTIPLE VALUES FROM THE FUNCTION

We have already discussed that; function can return only one value. Function can not return more than one value. But with the help of the pointer, we can return more than one values from the function. Consider the following example, in which we have

taken 2 variables, initialized with 12 and 4. We want to compute addition, multiplication, subtraction and division of these values. If function can return only one value then how can we return 4 different values from the function? Consider the following program, in which calc () function is returning 4 values, to the main () function.

Program 4.4 Returning multiple values from the function.

```
#include<stdio.h>
void calc(int, int, int *, int *, int *,int *);
void main()
{
    int x, y, sum, sub, mul, div;
    x=12;
    y=4;
    calc(x, y, &sum, &sub, &mul, &div);
    printf("\nAddition is:%d" ,sum);
    printf("\nSubtraction is:%d" ,sub);
    printf("\nMultiplication is:%d" ,mul);
    printf("\nDivision is:%d" ,div);
}
void calc(int x, int y, int *p, int *q, int *r, int *s)
{
    *p=x+y;
    *q=x-y;
    *r=x*y;
    *s=x/y;
}
```

OUTPUT:

```
Addition is:16
Subtraction is:8
Multiplication is:48
Division is:3
```

USE OF THE POINTER:

1. When we have the local variable, which we want to change by some other function, we need pointer.
2. If we want to return multiple values from the function, then pointer is useful.
3. In the case of, dynamic memory allocation, variables are created at runtime, they do not have name, but in this case dynamic memory allocation function returns address of that variable, and we can operate dynamically created variables by its pointers.

4. It allows passing of an array and string to the function very efficiently.

Exercise: 4.1

10. To find the address of a variable _____ operator is used.

11. To access the data elements of the structure, by its pointer _____ operator is used.

12. _____ is a special variable used to store the address of another variable.

13. _____ operator is used to declare pointer variable.

14. _____ function of dynamic memory allocation is used to create a variable.

15. Dynamic memory allocation function _____ is used to create an array.

16. _____ function is used to delete the variable created by malloc() function.

4.8 LET US SUM UP

In this chapter we have studied pointer variable. We have started with why we need pointer variables, and then we have learnt to declare and initialize pointer variable. We have also seen that how pointer variable can be efficiently used with array and structures. Finally, we have seen different dynamic memory allocation functions and advantages of the pointer variable. We hope, students know easily use pointer variables in the C-programs, and have clear understanding about pointer variable.

4.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Exercise: 4.1

10. &

11. -> (point-to operator)

12. Pointer

13. *

14. malloc()

15. calloc()

16. free()

Block-3

Introduction to Data Structure

Unit 1: Basics of Data Structures

Unit Structure

- 1.1. Learning Objectives
- 1.2. Introduction
- 1.3. Data and Information
- 1.4. Data Types
- 1.5. Storage Representation
- 1.6. Classification of Data Structures
- 1.7. Primitive Data Structures
- 1.8. Non-primitive Data Structures
- 1.9. Let us sum up
- 1.10. Check your Progress
- 1.11. Further Reading
- 1.12. Assignments

1.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Differentiate between data and information.
- Categorize the various data types and describe the storage.
- Classify the various types of data structures and compare them.
- List the various primitive data structures and describe their storage format.
- List the various non-primitive data structures.
- Define the array and show the its addressing functions.

1.2 INTRODUCTION

Understanding of the data structures is very essential for writing computer programs from very small and simple to very large and complex (modern softwares). Purpose of any computer program is to process the user data. Process represents the task to be performed by a program and forms the algorithms for a program. The data to be processed by a program is represented in different forms called data structures using the relations among the data items. From this, we can conclude that

Program = algorithms + data structures

The data represented by data structures are stored in computer memory so that algorithms in a program can read it for processing and write the updated data back in memory. The way of storing data structure in computer memory is called its memory organization. Hence, the data structures and their organization in computer memory are very important issues for writing programs. Efficiency of a program is also influenced by choice of the data structure and its representation in memory. Choosing improper data structure creates many problems.

This unit introduces the basic concepts of data structures, various types of data structures, memory representation of various data structures and operations on data structures. It also covers the array with addressing functions.

1.3 DATA AND INFORMATION

We use computer programs to process the data i.e. data are supplied as input to programs. These *data* are nothing but single values or set of values. Hence, data is defined as values or group of values. Following are some examples of data.

- 1) Program
- 2) 67.8
- 3) 14/03/1968
- 4) 72, 55, 67, 62, 89
- 5) 10 Aklavya 74

In above examples, first three data are single values. Fourth is an example of set of values. The fifth example shows the composite data. Composite data are used to represent the real life entities like students, employees etc. Fifth data is an example of entity student with attributes roll number, name and his result.

Data are row facts and does not possess any meaning. For example, 67.8 is just a number. In real life, we assign meaning to these row facts to use them meaningfully. Let us say, 67.8 is a weight of a person. Thus, we have assigned meaning like “a weight of a person” to 67.8 to represent it as meaningful data. Once data is assigned meaning it is called *information*. Hence, Information is defined as data with meaning. In case of composite data, the attributes of entity themselves are meaning. In our previous example, once we say it is student record with roll number, name and result, it is information. The data shown in previous example are shown below with their meaning i.e. as information.

- | | |
|-----------------------|---|
| 1) Program | “set of instructions” |
| 2) 67.8 | “weight of a person’ |
| 3) 14/03/1968 | “birthdate of a person” |
| 4) 72, 55, 67, 62, 89 | “marks of five subjects” |
| 5) 10 Aklavya 74 | “student details : roll number, name, result” |

Information is not only assigning meaning to data, but any processed data is also known as information. Making total of marks and deciding the class like pass, first, distinction is an example of information. Consider that we have table or set of records representing student’s roll number, name and marks of five subjects, by processing these data we can find various information like,

- Finding the total of marks for each student and their result.
- Generating merit list based on total of all the subjects.
- Count number of students failed.
- Finding the highest marks in each subject.

1.4 DATA TYPES

Data are of various types. For example, 24 is a numeric type, "Program" is non-numeric i.e. string, 14/03/1968 is date type. Similarly we can have various data with different types, like graphics, array, character, alpha-numeric etc. Programming languages support basic data types and user-defined data types to represent variety of data types. C supports basic data types like char, int, float and double and user-defined data types like array, structure, union etc.

The basic data types are useful for representing single numeric and non-numeric data like integers, real numbers, single characters and strings. Many application demands to represent more complex data types like dates, set of values, lists or tabular values. User defined data types are either used or person has to create data type using basic and user defined data types supported by language to meet the need of the application. Date type is not supported by C language, so we can use structure to store a date with three integer members. Same way, to represent a table with multiple columns, we can use array of structures in C where each member of structure represent a column. Processing of these data types are done by defining the various operations based on the need of the application.

1.5 STORAGE REPRESENTATION

Data to be processed by a program is usually stored in primary memory of computer which consists of RAM(Read/Write Memory) and ROM(Read Only Memory). Normally, user programs use RAM for storing its data. But how do we store data in computer memory? Before answering this question, it is better to understand how the computer memory is organized.

Computer memory is just like a one dimensional array where each location stores 8-bits i.e. a byte and identified by a unique number. This unique number is

called a memory address of a respective memory location which stores a byte. Fig. 1.1 shows the 256 byte memory with address range 0 to 255. The address of first location is 0 and last location is 255. For, 1 KB (Kilo Byte) memory address range is 0 to 1023.

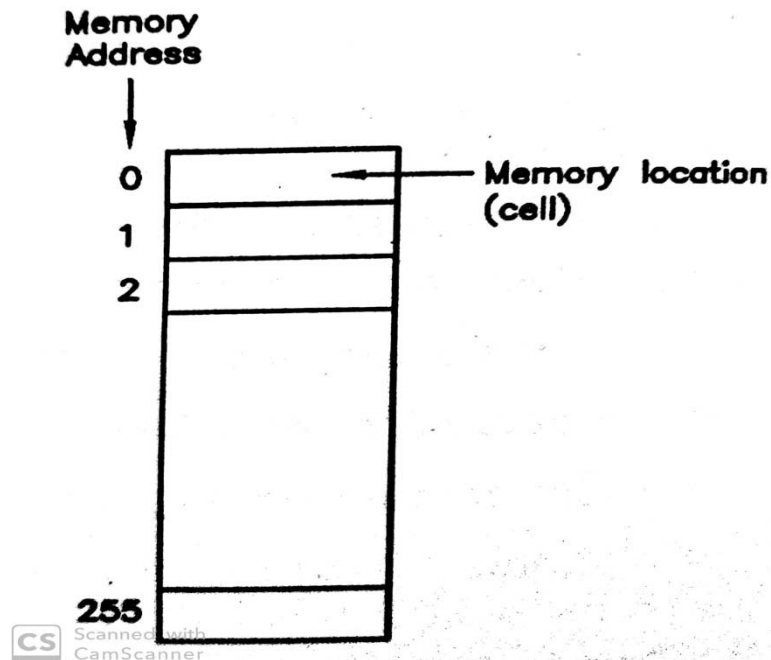


Fig. 1.1 : 256 Byte Memory

Each memory location is nothing but a **cell** which stores an 8-bit value or a byte. Hence, basic unit of computer memory is byte. Is it possible always to store a data value or data item into a byte? Answer is yes, if data item is a character as it is stored as 8-bit ASCII value. But answer is no, if data item is an integer or float. Because an integer occupies two bytes while a float needs four bytes. Then, how an integer or a float is stored? Naturally, integer is stored using two consecutive bytes and float is stored using four consecutive bytes in memory as shown in fig. 1.2.

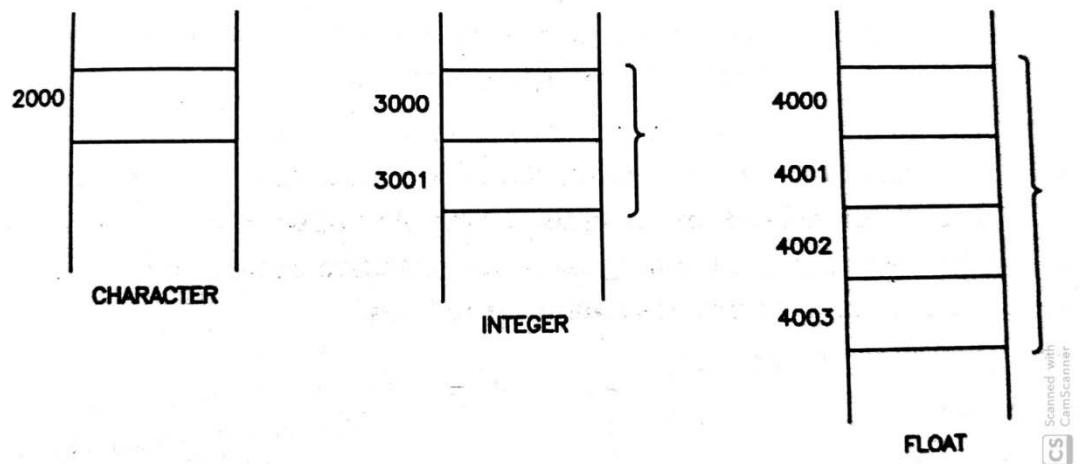


Fig. 1.2 : Storage of character, integer and float

As shown in figure, character is represented at memory address 2000 and it occupies only one byte. However, in case of integer and float they occupy more than one byte. What then is address of it? Whenever multi-byte data value (or object) is under consideration and its address is referred, we always refer the address of first byte only. Hence, address of integer in fig 1.2 is 3000 and for float it is 4000. System will automatically read required number of bytes from given address to form correct value i.e. in case of integer, system reads a byte from given address, second byte from next address and combines them to form 16-bit value.

Previous example deals with basic data items like a character, an integer and a float. But what's about real life information like set of values or information about person. Set of integers may be stored as data structure like an **array**. Person information may consist of his/her name, age, address etc. This collection of related data items is called a **record** of a person where each data is called a **field**. For, example age is a field which is stored as an integer. In other terms, **field** is a collection of cells as it may occupy one or more cells. We will discuss about arrays and records with their storage representation in more detail later.

1.6 CLASSIFICATION OF DATA STRUCTURES

Data structure is defined as logical relation among the data items. There are various possible ways to logically relate the data items. These possible ways to

logically relate the data items are collectively called *data structures*. The choice of proper data structure relates the items in such a manner that the data items are easily and efficiently accessible for various operations.

The data structures are classified as ***primitive*** data structures and ***non-primitive*** data structures. The primitive data structures are those data structures which are directly processed by machine by its instructions. The basic data types supported by programming languages are primitive data structures. Following data types are examples of primitive data structures.

Primitive data structures :

- Integers
- Reals (float)
- Characters
- Booleans
- Pointers
- Strings

Non-primitive data structures represent more complex structural relationship among the data items and require algorithms or special code for performing operations as they are not directly operated by machine.

The non-primitive data structures are further classified into *linear* data structures and *non-linear* data structures. The logical relation among the data items in linear data structures is linear (like one dimension, one-by-one), while it is not linear in case of non-linear data structures. Examples of non-primitive data structures are as follows.

Non-Primitive data structures :

- Linear data structures
 - Arrays
 - Stacks
 - Queues
 - Linked lists
- Non-linear data structures
 - Trees
 - Graphs

The important issue about the data structures is their storage representation or memory organization. The representing a data structures in computer memory is called its *storage structure* or *memory organization*. Data structure holds logical relationship among the data items which are maintained also in its storage structure regardless of which type of storage structure is used to represent it in memory. It is possible to use more than one possible ways to represent a data structure in memory i.e. using different storage structure. For example, stack can be implemented using sequential allocation (Array) or dynamic allocation (Linked list). Storage structure of primitive data structure depends on the architecture of particular computer system. For non-primitive data structures, user can choose suitable storage representation based on need of the application.

Another important issue is operations on data structures. The first and most important operation is to create a data structure in memory. Allocations of space in computer memory for a data structure is called **create** operation. Once the use of data is over, we should remove it from memory which is called **destroy** operation. Consider the following declaration in C.

```
int x;
```

This is example of create operation as this statement allocates memory to integer variable **x**. Once it is created, it can be assigned value as

```
x = 10;
```

If **x** is local to some function, it is removed from memory when function ends and if it is global variable; it is removed from memory when program ends. This is same as destroy operation.

The basic operations also include **select** (to access) and **update** (to change) operations. For example statement,

```
a = x * 5 + 2;
```

requires to read value of **x** from memory to evaluate expression $x*5+2$ to compute value of **a** which is 52 ($=10*5+2$). The reading value of **x** from memory is select operation. Later on we may need to change or update value of **x** using

```
x = 14;          /* change x to 14 */
```

or

```
x = x + 2;    /* update x by 2 */
```

In case of non-primitive data structures, the most important operations are **insert** and **delete** operations apart from above four basic operations. Insert operation add new data item in a data structure, while delete operation removes a particular data item from data structure. In addition to insert and delete operations, we can perform variety of other operation on non-primitive data structures depending on type of data structure.

Unit-3 and Unit-4 cover the various data structures, their storage structures and operations on them.

1.7 PRIMITIVE DATA STRUCTURES

Primitive data structures include : Integers, Reals, Characters, Booleans, Pointers and Strings. Programming languages provides in-built facilities to deal with these data structures. In this section, we will take the overview of these data structures and their storage representations. We will take C language as an example to understand, how these data structures are supported in programming languages.

Integers :

Integers represent numerical values which are whole quantities. For example, number of students in a class, number of mango trees in a garden, number of pages in a book etc. Integers are normally represented using radix-complement form. Computer uses binary numbers (radix 2 numbers) and radix-complement for binary machine is called **2's complement**. If we take an 8-bit integer, the range of numbers supported is -128 to +127. For example,

Number	2's complement representation
+5	0000 0101
-7	1111 1001

The MSB (Most Significant Bit) represents the sign of the number. If MSB is 0, number is positive and if MSB is 1, number is negative. In above example, for +5, MSB is 0, while for -7, MSB is 1. Rests of the 7-bits represent the magnitude of the

number. The format used to represent the integer numbers in sign-magnitude form is shown in fig. 1.3.

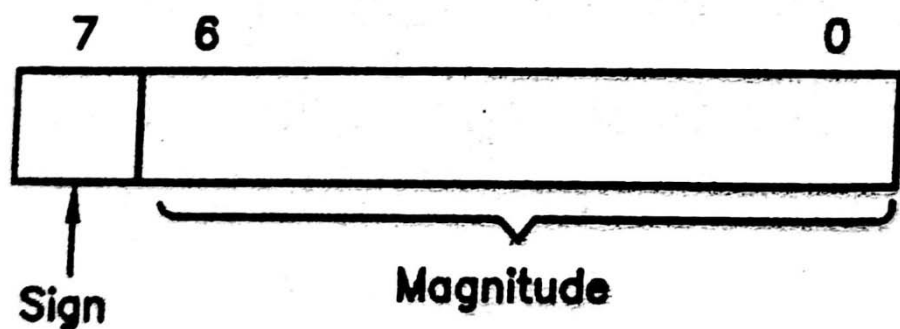


Fig. 1.3 : Integer representation using sign-magnitude form

To represent integer number in 2's complement form,

- 1) Take binary equivalent of number after removing sign i.e. unsigned value.
- 2) Add remaining leading 0s to make it 8-bit.
- 3) If number is positive, 8-bit representation in step 2 is answer.
- 4) If number is negative, take 2's complement (1's complement plus 1) of step 2

Example : Representation of +5

- ✓ Binary of 5 is 101
- ✓ Adding leading 0s, it is 0000 0101
- ✓ 2's complement representation of **+5** is **0000 0101** ■

Example : Representation of -7

- ✓ Binary of 7 is 111
- ✓ Adding leading 0s, it is 0000 0111
- ✓ 2's complement of 0000 0111 is

1111 1000 1's complement

1 plus 1

1111 1001 2's complement

- ✓ 2's complement representation of **-7** is **1111 1001** ■

Radix complement representation is very useful as it allows flexibility in arithmetic operations. For example, addition and subtraction both are performed using addition only. Consider following examples using 2' complement.

Example : 5 + 7

$$\begin{array}{r}
 +5 \quad 0000 \ 0101 \\
 +7 \quad 0000 \ 0111 \\
 \hline
 +12 \quad 0000 \ 1100
 \end{array}$$

Answer is +12 as MSB in 0000 1100 is 0 and decimal equivalent of 0000 0011 is 12. ■

Example : 5 – 7 (It is same as +5 + (-7))

$$\begin{array}{r}
 +5 \quad 0000 \ 0101 \\
 -7 \quad 1111 \ 1001 \\
 \hline
 -2 \quad 1111 \ 1110
 \end{array}$$

Answer is -2 as MSB in 1111 1110 is 1 and 2's complement of 1111 1110 is 0000 0010 which is equal to decimal 2. ■

For unsigned integers, all 8-bits are used to represent the magnitude and hence the range of numbers represented is 0 to 255.

In general, the range of numbers for n-bit signed representation is -2^{n-1} to $+2^{n-1}-1$ and for unsigned integers 0 to 2^n-1 . For example, 16-bits signed representation represents range $-32768(2^{15})$ to $+32767(2^{15}-1)$, while unsigned numbers represents range 0 to $65536(2^{16}-1)$.

C language uses the various types of integers to provide the different range for representing small to large quantities.

Reals :

Real numbers represent the numbers with fractional parts i.e. digits after decimal parts. They are used to represent contiguous quantities like temperature of day, weight of a person, distance between two points etc. For example, today's temperature is 42.5 degree centigrade. Normally, real numbers are shown in *fixed point format* as

$$\pm d_1 d_2 d_3 \dots d_n . f_1 f_2 f_3 \dots f_n$$

where $d_1 d_2 d_3 \dots d_n$ is decimal part while $f_1 f_2 f_3 \dots f_n$ is fractional part. In 42.5, 42 is decimal part and .5 is fractional part. Real number can alternatively, stored in *scientific* or *exponent* format as

$$\pm f_1 f_2 f_3 \dots f_n \times R^E$$

where $f_1 f_2 f_3 \dots f_n$ is normalized fraction part known as *mantissa*, R is the base of the number and E is the exponent. The normalized fraction part means the decimal point is set to the left of the first non-zero digit. For example, 123.45 can be represented in scientific form as 1.2345×10^2 but it is not normalized. The normalized scientific form of 123.45 is 0.12345×10^3 .

The scientific format is normally used to represent the numbers which are too small or too big. For example, 0.000000587 is very small number and written in scientific format as 0.587×10^{-8} . Similarly, 15000000000 is too big number and written in scientific format as 0.15×10^{11} .

Let us understand how the real numbers are stored in computer memory i.e. their storage structure. The storage structure for the real numbers is given in fig. 1.4

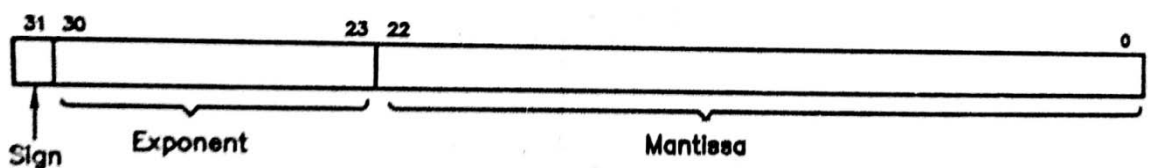


Fig. 1.4 : Storage structure for real numbers

It contains three parts: *sign*, *exponent* and *mantissa*. The *sign* is a single bit and represents sign of a number. If it is 0, number is positive and if it is 1, number is negative. The *exponent* is 8-bit and hence it can be -128 to +127. It can be represented using sign-magnitude form used for integers, but normally *biased-exponent* form is used as it is more convenient for floating point arithmetic. In *biased-exponent* form number is biased by middle number in unsigned range with same size i.e. 128 (middle number in 0 to 255 range for 8-bit number). For example, if the exponent value is -4, then it is stored as $-4+128 = 124$. This will effectively puts numbers in a range -128 to +127 in 0 to 255. The *mantissa* is normalized fractional

part. For normalization, condition is that fractional part must be in range $R^{-1} \leq \text{Mantissa} < 1$, where R is radix of the number. For binary representation, it is $0.5 \leq \text{Mantissa} < 1$. The following example clarifies the process.

Example : Representation of +24.5.

- ✓ Covert 24.5 to binary which is 10111.1
- ✓ Covert 10111.1 to normalized binary floating-point number. It is 0.1011111×2^5
- ✓ Mantissa is 101111. Extend it to 23 bits by appending 0s on right. It becomes 10111100000000000000000
- ✓ Exponent is 5. Add 128 to make biased exponent. It becomes 133 whose binary is 10000101
- ✓ Sign is 0 as number is positive
- ✓ The final representation of 23.5 is

0 10000101 10111100000000000000000 ■

C language supports floating point numbers with single precision floating point and double precision floating point. The higher precision means more number of digits after decimal point for more accuracy, but with more storage requirements.

Characters :

Character set includes alphabets, digits and special symbols called punctuation marks (+, -, *, >, ! etc.). Each character is stored in computer memory by its code depending on the coding system used in particular computer. Majority of the computers including PCs uses ASCII (American Standard Code for Information Interchange) coding system. ASCII is widely accepted and include

- Alphabets both uppercase (A,B,...,Z) and lowercase (a,b,...,z)
- Digits (0,1,2,...,9)
- Punctuation marks (+,-,*,>!,.... etc)
- Control characters like CR(Carriage Return), LF(Line Feed), HT(Horizontal Tab) etc.

ASCII is 7-bit code and represents $2^7 = 128$ different characters. For example, ASCII code for A is 65 (41 in hex), while ASCII code for 0 is 48 (30 in hex).

C language uses **char** data type to represent the characters. For example,
char alpha;

defines a variable **alpha** which can store a character. To assign character to variable **alpha**, we can use

```
alpha = 'A';    or    alpha = 65;
```

Either directly ASCII value is assigned or character itself is given in single quotes. Once character is given in single quotes, it automatically assigns the corresponding ASCII value of a character. Hence, above both statements are one and same. The print statement

```
printf("The ASCII value of %c is %d\n",alpha,alpha);
```

prints

```
The ASCII value of A is 65
```

because %c prints the character stored in **alpha** while %d prints ASCII value of character stored in **alpha**. C allows variety of operations to be performed on character data types. For example,

```
alpha = alpha + 3;
```

increments ASCII value from 65 to 68 and if above print statements is performed again the output will be

```
The ASCII value of D is 68
```

Booleans :

Boolean data types are used to represent data which is of type *true/false* or *yes/no*. For example, result of a student is either pass (true) or fail (false). In general, we can use Boolean data types for any information having only two possible states like gender of a person is male or female. Many programming languages support boolean data type directly. JAVA supports Boolean data types whose values can be only *true* and *false*. C language does not support Boolean data types, but using enumerated data types it is possible to create them. The statement

```
enum boolean {false,true};
```

creates data type **boolean** which can take only values *false* and *true*. Now we can create variables of **boolean** type and play with them like other variables. The statement

```
boolean status = true;
```

defines a variable **status** with value *true*. Now it can be used in program to perform operation like

```
if(status == true) {  
    /* put code for action */  
    ....  
}
```

Pointers :

Pointers are link or reference to data structures. The data structures referred by pointers can be simple like any primitive data type integer, real, character etc. or complex like non-primitive data structures like linked listed and trees. Fig. 1.5(a) shows a pointer to integer while fig. 1.5(b) shows a pointer to whole linked list. Pointers provide universal way of accessing data structures regardless of whether they are small and simple or large and complex. There are many situations where use of pointers provides more flexibility to access and modify the data structures.

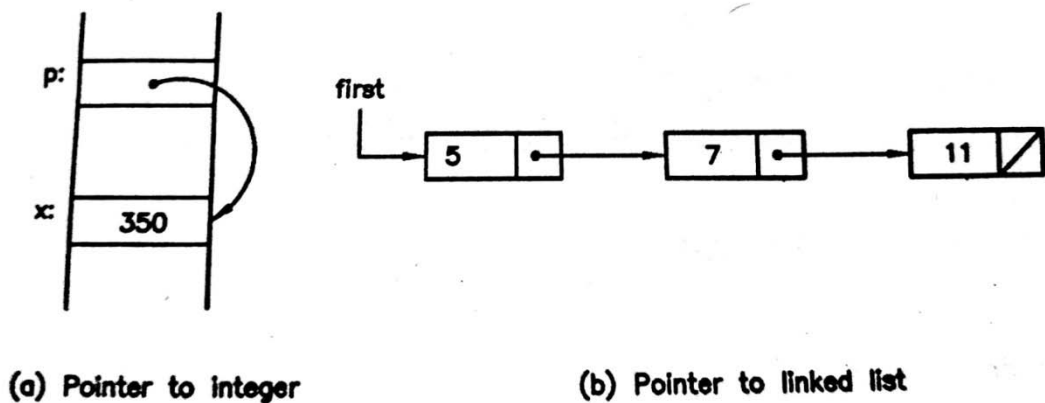


Fig. 1.5 : Example of pointers

Many times it happens that a single data item may be required at many places in a program. One of the ways is to provide separate copy of same data item everywhere which leads to maintaining multiple copies of a data item plus needs more memory. Pointer provides efficient solution for this situation. Data item is created only once and each part of program requiring this item creates a separate

pointer to same item. Thus, a single data item is referred by multiple pointers avoiding multiple copies. Consider following C statements.

```
int x = 10;  
int *p = &x;  
int *q = p;
```

The first statement defines integer variable **x** with value 10 which is pointed by both **p** and **q** pointers. Any time **x** can be accessed using either **p** or **q** pointers. It is shown in fig. 1.6.

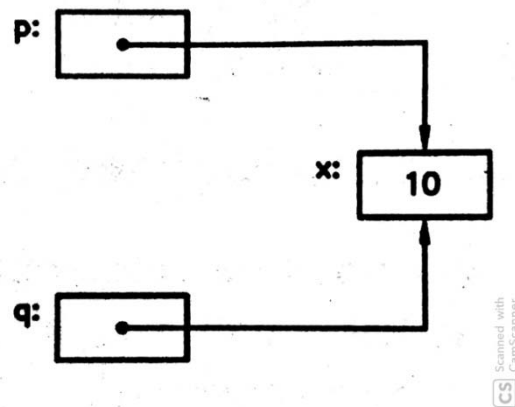


Fig. 1.6 : Multiple links to a data item

Pointers are very useful in parameter passing to functions. Normally parameters are passed using *call by value* in which a duplicate copy of parameters is created by called function with values provided by caller function from actual to formal parameters. The limitation of this approach is that the called function can change only their copy of parameters, but can not change the original values in caller. Many times it is required that caller should directly change the values in caller function. It is achieved only using pointers. Instead of passing the values, caller passes the addresses (pointers) to the original values to called function as parameters and using these pointers called function directly accesses the original values of caller function.

Pointers are very useful in non-primitive data structures in which frequently insertion and deletion of items are necessary. Example of such data structures are

linked list and tree. By changing only one or two links a new data item can easily be inserted or deleted.

C users are already familiar with the pointers. Apart from using pointers to access the data item it points, we can also perform number of operations on C pointers like assigning one pointer to another pointer, adding or subtracting integer from pointer, subtracting one pointer from another pointer. We will study two different methods for accessing data items from complex data structures. One of the methods is *computed address* method in which pointer or link to only first item is stored while pointer or link to other items are calculated using some computations with reference to pointer or link to first item. The example of such data structure is arrays which we will study later in this chapter. Second method is *pointer or link* method in which pointer or link to each data item in data structure is separately stored and using those, items are accessed directly. Examples of such data structures are linked list and trees which we will study in chapter 3 and 4.

Strings :

Strings represent non-numeric information like name of person, city name, address, title of a book etc. The storage structure for strings is same as characters as string is a sequence of characters. Hence, a string is stored in computer memory as series of ASCII codes of the characters in a string. Variety of operations can be performed on strings like finding length of a string, joining two strings, comparing two strings, finding substring from a string etc.

1.8NON-PRIMITIVE DATA STRUCTURES

Non-primitive data structures are representing complex relationship among the data items and there are no directly machine instructions available to process them. Hence to process them, algorithms are designed to perform desired operations. The efficiency of these operations depends on how the data items are organized in data structure as the organization of data items directly affects the ways individual items are accessed from data structure. The non primitive data structure is called efficient if it preserves the original relationships among the data items and their storage structure is such that which allows easy and fast access to individual

data items. The data items which make the non-primitive are either primitive or in many cases even non-primitive.

The non-primitive data structures are classified broadly into two categories : *linear* and *non-linear*. The linear data structures are those in which adjacency among the data items are linear, while in non-linear adjacency among the data items are not linear.

The arrays, stacks, queues and linked lists are examples of linear data structures. Arrays represent the ordered set of data items. Stacks and Queues are also linear lists in which insertion and deletions operations are performed in LIFO (Last In First Out) and FIFO (First In First Out) manner respectively. The linked lists are more general lists and allow insertion and deletion at any point.

The trees and graphs are examples of non-linear data structures. The trees are used to represent the hierarchical relationships. The directory structure in computers is an example of trees. The graphs are used to represent relations in network fashion. The railway network or airline network is an example of graphs.

We will study these data structures with their storage representation, operations on them and their applications in subsequent units.

The comparison between primitive and non-primitive data structures is as follows:.

Primitive data structures	Non-primitive data structures
They are simple and represent basic data items.	They are complex and represent complex relationship among the data items.
Storage structure is dependent on machine architecture or programming language.	Storage structure is decided by the user based on application need.
Direct machine instructions are available to process them.	Direct machine instructions are not available to process them. Algorithms are designed to perform operations on them.

Create, destroy, select and update are common operations performed on them.	Apart from four common operations : create, destroy, select and update, the two more frequent operations performed on them are insert and delete.
The basic data types provided by programming languages are primitive data structures.	User defined data types provided by programming languages are non-primitive data structures.
Integers, Reals, Characters, Booleans, Pointers and Strings are primitive data structures.	Arrays, Stacks, Queues, Linked lists, Trees and Graphs are non-primitive data structures.

1.9 LET US SUM UP

Data : Data are raw facts and does not possess any meaning.

Information : Once the data is assigned meaning, it is called information.

Data structure : Data structure is defined as logical relation among the data items.

Primitive data structures : Data structures which are directly processed by machine by its instructions. The basic data types supported by programming languages are primitive data structures.

Non-primitive data structures: Non-primitive data structures represent more complex structural relationship among the data items and require algorithms or special code for performing operations as they are not directly operated by machine.

Storage structure : The representing a data structures in computer memory is called its storage structure or memory organization.

1.10 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. data structures are directly processed by processor.
2. Pointer is an example of data structure.

3. Data structures represent the relationship among the
4. Smallest unit of memory is called
5. Table showing the student marks is an example of data structure.
6. and are non-linear data structures.
7. Representation of data structure in computer memory is called or
8. Characters are stored in memory using codes.
9. Allocating memory space for data structure is called operation.
10. Non-primitive data structures are categorized as and data structures.
11. User defined data types supported by programming languages are normally data structures.

➤ **True-False**

1. Temperature of a day is an example of data.
2. Machine instructions can directly process the non-primitive data structures.
3. C structure is an example of primitive data structure.
4. All the user defined data types in C are primitive data structures.
5. Insert and delete operations are frequently used in linked lists.
6. Stack is an example of non-linear data structure.
7. Stack is an example of FIFO.
8. An algorithm is needed to perform operations on a non-primitive data structure.

➤ **Answer in brief**

1. What is data structure? Mention two major categories of data structures.
2. What is memory cell? How many bits does it store?
3. List the primitive data structures.
4. Give the two major characteristics of non-primitive data structures.
5. What is difference between data structure and storage structure?
6. How can you create Boolean data types in C?
7. What do you understand by update operation on data structures? Give example.

➤ **Answer the following**

1. List the various basic operations on data structures? Give suitable example for each of them.
2. Discuss the sign-magnitude form of integer representation with suitable illustrations.
3. Describe the storage representation of floating point numbers.
4. Find the storage representation of +41.23.
5. Compare the primitive and non-primitive data structures.
6. Explain the computed and linked address methods with their advantages and disadvantages.
7. Write short notes.
 1. Primitive data structures
 2. Non-primitive data structures
 3. Operations on data structures

1.11 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

1.12 ASSIGNMENTS

1. Develop a C program to print the binary storage of given integer.
2. Develop a C program that separates sign, exponent and mantissa of a given floating point number.
3. Develop a C program which accepts the string of digits and converts it into integer number.
4. Develop a C program which accepts an integer number and converts it into a string of digits.

Unit 2: Array and Stack

2

Unit Structure

2.1 Learning Objectives

2.2 Introduction

2.3 Arrays

2.4 Stacks

2.5 Operations on Stacks

2.6 Applications of Stacks

2.7 Let us sum up

2.8 Check your Progress

2.9 Further Reading

2.10 Assignments

2.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define the array and explain the addressing function.
- Use the addressing function of an array to compute the location of a given element of the array.
- Define a stack and list the real life examples of stack.
- Perform the push and pop operations performed on stack.
- Define recursion and compare its with iterative approach.
- Use the stack to evaluate the postfix expressions.

2.2 INTRODUCTION

A data structure in which relationship among the data items is linear is known as linear data structures. The linear data structures include arrays, stacks, queues and linked list. All the linear data structures are non-primitive data structures and there are no direct machine instructions provided by processor to process them. For performing any operation i.e. processing on them, we need to write appropriate algorithm.

This unit covers the two important linear data structures: arrays and stacks. We will learn the concepts of array with the addressing functions for one-dimensional and two-dimensional arrays. Stack allows operations only at one end and follows the LIFO (Last In First Out) order. Stack is covered with important operations: push and pop with its applications in recursion and in evolution of postfix expressions.

2.3 ARRAYS

Arrays are used to store an ordered set of data elements. All the data elements stored in an array should be of the same type i.e. all the elements are homogeneous. An array provides a single name to a group of ordered elements of same type. Array uses the static memory allocation as memory required is known at compile time. The total memory required by an array is allocated as single contiguous block so it is also example of sequential allocation. The name of an array indicates the memory location of the first element in an array. To access each

element from array, array name and its position(subscript) relative first to first element are provided to compute the location of required element. This computation is called the *addressing function* of an array. Thus, array is an example of computed address method. In this section, we will study the storage structure and addressing functions of one-dimensional and two-dimensional arrays with proper illustrations. We will also consider the C arrays as example.

One-dimensional Arrays :

One-dimensional array is also known as vector. Let us consider an one-dimensional array **A(L:U)** of **N** elements each of size **S**, where,

- A** : name of the array
- L** : lower bound (subscript for first element)
- U** : upper bound (subscript for last element)
- N** = Total no. of elements = $U - L + 1$
- S** : Size of each elements in bytes

Assuming **L=1, U=N** storage structure for the array **A(1:N)** is shown in fig. 2.1. The **N** elements are denoted as **A[1], A[2], ..., A[N]** where **A** is name of the array and integer in bracket is the position of an element, known as *subscript*.

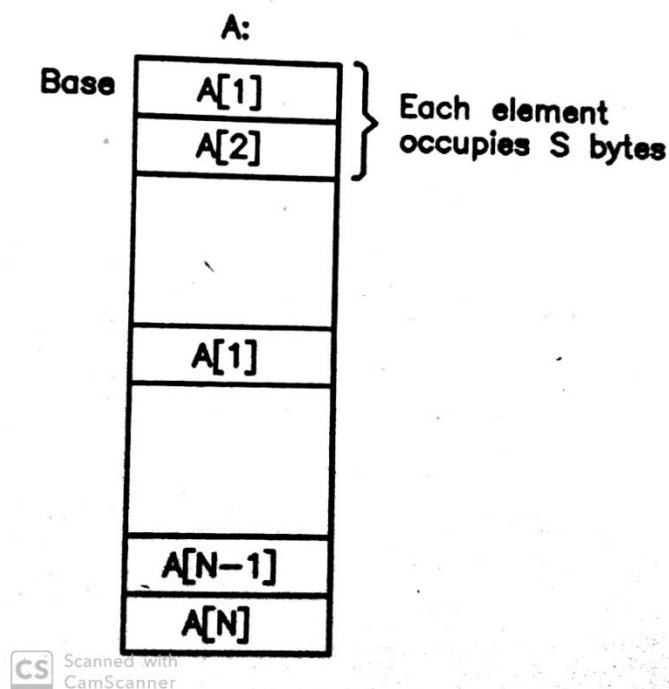


Fig. 2.1 : Storage structure for array A(1:N)

If an address or memory location of first element is denoted as **Base**, then addressing function to find location of I th element ($1 \leq I \leq N$) is as below.

$$\text{Loc}(A[I]) = \text{Base} + (I - 1) * S$$

The following examples clarify the concept.

Example : Give the storage structure for an array $A(1:5)$ with address of all the elements if size of each element is 2 bytes. Assume that memory location of first element is 2000.

Here, $L = 1$, $U = 5$, $N = 5 - 1 + 1 = 5$ and $\text{Base} = 2000$.

Using above addressing function, the addresses of elements are:

$$I = 1, \text{Loc}(A[1]) = 2000 + (1 - 1) * 2 = 2000$$

$$I = 2, \text{Loc}(A[2]) = 2000 + (2 - 1) * 2 = 2002$$

$$I = 3, \text{Loc}(A[3]) = 2000 + (3 - 1) * 2 = 2004$$

$$I = 4, \text{Loc}(A[4]) = 2000 + (4 - 1) * 2 = 2006$$

$$I = 5, \text{Loc}(A[5]) = 2000 + (5 - 1) * 2 = 2008$$

The storage structure is given in fig. 2.2

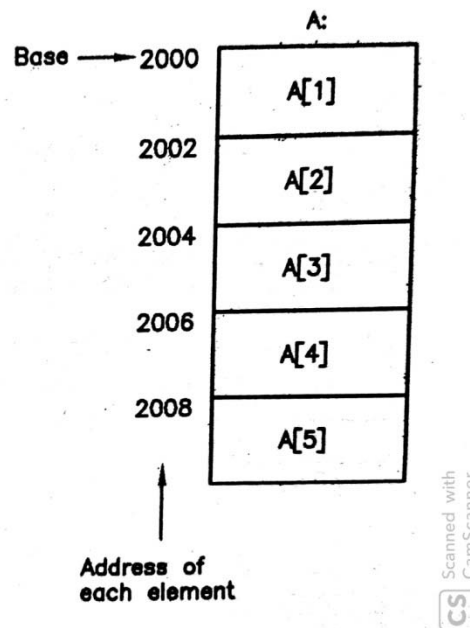


Fig. 2.2 : Storage structure for array $A(1:5)$

Example : Given an array $A(3:7)$ with size of each element 4 bytes and base address 2000, find the address of third element in the array.

Here, $L = 3$, $U = 7$, $N = 7 - 3 + 1 = 5$ and $\text{Base} = 2000$. Data elements are denoted as $A[3], A[4], A[5], A[6]$ and $A[7]$. Hence, third element is denoted as $A[5]$ with $I = 5$, its location is computed as

$$\text{Loc}(A[5]) = 2000 + (5 - 3) * 4 = 2000 + 8 = 2008 \quad \blacksquare$$

In above examples, we used lower subscript 1 and 3 respectively. Normally, 0 and 1 are the best candidates for starting any sequence for human users. For this reason, programming languages either starts lower subscript from either 0 or 1 for the arrays. The PASCAL is an example of programming language which starts lower subscript from 1, while C, C++ and JAVA are examples of programming languages which start lower subscript from 0. An array of N elements in PASCAL is shown in fig. 2.3(a) and same array in C is shown in fig. 2.3(b).

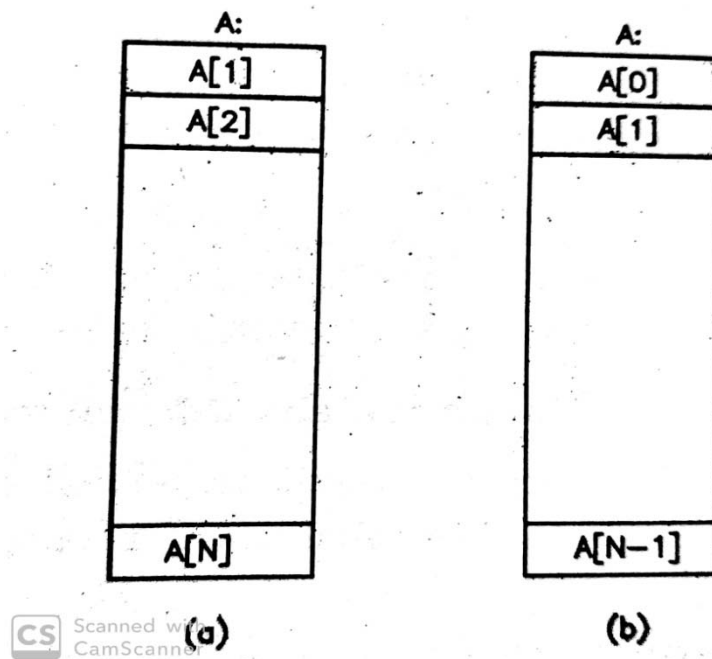


Fig. 2.3 : Array $A(N)$ in (a) PASCAL (b) C

Following C statement defines array of 10 integers.

```
int a[10];
```

Considering size of one integer 2 bytes it is allocated 20 bytes block of memory statically at compile time only, where a is the address of the first element i.e. $a[0]$. C

allows user to access **ith** element with notation **a[i]** where **a** is the name of the array i.e. base address and **i** denotes the position. Internally C converts **a[i]** to

a + i * sizeof(int);

which is the addressing function. C uses **sizeof** operator to find size of integer data in bytes. Compare it with formulae (1.1), **Base** is replaced by **a**, **S** is replaced by **sizeof(int)** and only **i** is used instead of **(I – L)** as **L = 0** in C language.

Once array is defined in C, we can store the values in them and perform desired operation. The examples of operations are finding a minimum or maximum from an array, finding average of all elements of an array, sorting into ascending or descending order etc. Just like integer array, we can create array with any data types and perform similar operations.

It is also possible in C using pointers to allocate memory to array at run time i.e. dynamic allocation as follows.

```
int *a;
int n;
scanf("%d",&n);
a = (int *)malloc(n*sizeof(int));
```

Once array is created in above manner, we can use the same subscript notation **a[i]** to access the elements or use totally pointer manipulations. Refer C Programming book for more details on pointers and using them with arrays. Although memory is allocated at run time in this case, still it is single block and it is sequential only.

The major limitations of the arrays are that once the array is created, it is not possible to insert and delete the elements from them.

Two-dimensional Arrays :

Two-dimensional array is made of rows and columns. Following array **A(1:3, 1:4)** is an example of an array consists of 3 rows and 4 columns.

	4	5	6	7
A =	1	2	3	4
	5	6	7	8

where **A[I][J]** denotes the element in **I**th row and **J**th column.

Remember that computer memory is one-dimensional array of locations. The question is, how can we store two-dimensional array in one-dimensional memory? There are two ways :

- 1) Consider two-dimensional array as vector of rows and store it row by row. It is called *row-major order* representation and shown in fig. 2.4(a).
- 2) Consider two-dimensional array as vector of columns and store it column by column. It is called *column-major order* representation and shown in fig. 1.10(b).

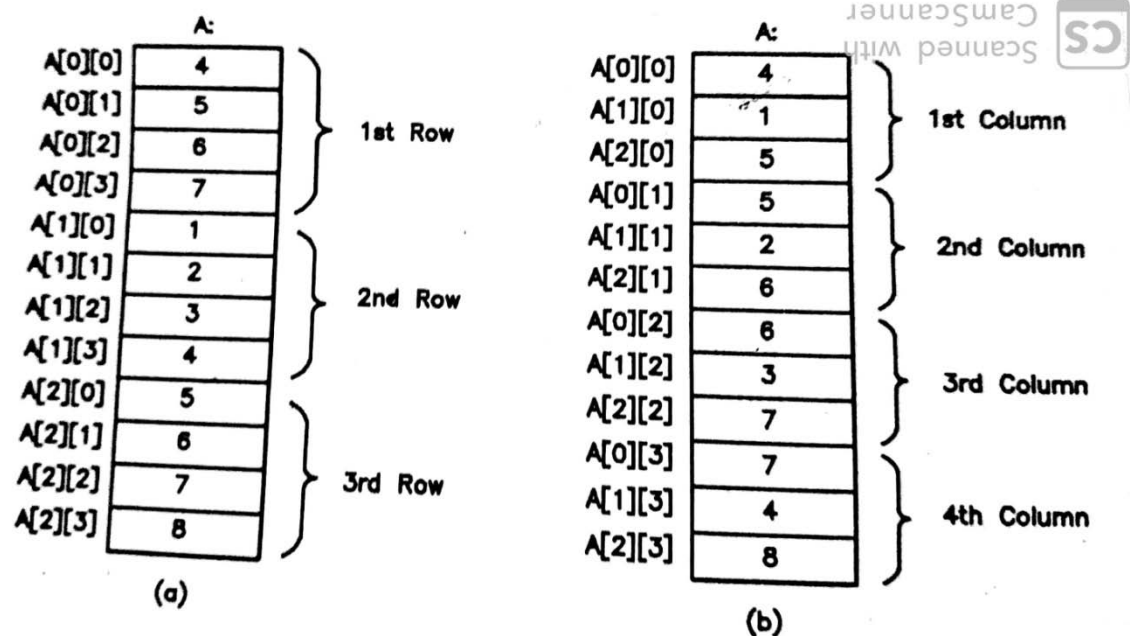


Fig. 2.4 : (a) row major order (b) column major order representation of A(3,4)

Let us find the addressing functions for both row-major and column-major order representation of two-dimensional arrays. Consider an array $A(L_1:U_1, L_2, U_2)$ of $M \times N$ elements with each elements is of S bytes where,

A : name of the array

L₁ : lower bound of row

U₁ : upper bound of row

L₂ : lower bound of column

U₂ : upper bound of column

M = No. of rows = No. of elements in a column = $U_1 - L_1 + 1$

N = No. of columns = No. of elements in a row = $U_2 - L_2 + 1$

If an address or memory location of first element is denoted as **Base**, then addressing function for row-major order representation to find location of element in **I**th row ($1 \leq I \leq M$) and **J**th column ($1 \leq J \leq N$) is given below.

$$\text{Loc}(A[I][J]) = \text{Base} + ((I - L_1) * N + (J - L_2)) * S$$

An addressing function for column-major order representation to find location of element in **I**th row ($1 \leq I \leq M$) and **J**th column ($1 \leq J \leq N$) is given below.

$$\text{Loc}(A[I][J]) = \text{Base} + ((J - L_2) * M + (I - L_1)) * S$$

Example : Find the address of $A[2][3]$ element if $\text{Base} = 2000$ and $S = 2$ bytes for (a) row-major order (b) column-major order.

(a) row-major order :

$$\begin{aligned} \text{Loc}(A[2][3]) &= 2000 + ((3 - 1) * 3 + (2 - 1)) * 2 \\ &= 2000 + (6 + 1) * 2 \\ &= 2014 \end{aligned}$$

(b) column-major order :

$$\begin{aligned} \text{Loc}(A[2][3]) &= 2000 + ((2 - 1) * 4 + (3 - 1)) * 2 \\ &= 2000 + (4 + 2) * 2 \\ &= 2012 \blacksquare \end{aligned}$$

C language follows the row-major order representation for multidimensional arrays. C start both row and column subscripts from 0. An $M \times N$ array **a** of integers is represented in C as

$$\begin{aligned} &a[0][0], a[0][1], \dots, a[0][N-1], a[1][0], a[1][1], \dots, a[1][N-1], \dots, a[M-1][0], \\ &a[M-1][1], \dots, a[M-1][N-1] \end{aligned}$$

and elements are referred as $a[i][j]$. Internally C converts $a[i][j]$ to

$$a + (i * N + j) * \text{sizeof}(\text{int})$$

which is nothing but the addressing function. Compare it with formulae (1.2). Note that C uses array name as base address and lower bounds for both rows and columns are 0.

Example : Find the address of element $f[1][2]$ in C array

float $f[3][4]$;

Assume base address 2000.

$$\text{Here, } M = U1 - L1 + 1 = 2 - 0 + 1 = 3$$

$$N = U2 - L2 + 1 = 3 - 0 + 1 = 4$$

$$\begin{aligned}\text{Loc}(f[1][2]) &= 2000 + (1*4+2) * \text{sizeof}(\text{float}) \\ &= 2000 + 6 * 4 \\ &= 2024\end{aligned}$$

Note that float occupies 4 bytes. ■

2.4 STACKS

A stack is a linear data structure with restriction that it allows operations to be performed only at one end known as top of the stack. The operations are performed in Last In First Out order only. The major operations performed on stack are **push** and **pop**. A push operation inserts a data item in a stack and pop operation removes a data item from a stack. Push and pop both operation are performed only at one end i.e. at top of the stack and they are performed in an opposite order meaning a data value last pushed is popped first (LIFO).

Before we go into details of the stack, let us consider a real life example of a stack. Consider an example of a stack of plates on a table in marriage function. Guest will take a plate only which is on top (pop operation), while whenever a catering person puts more plates, they are placed on top of the existing plates only (push operation). Fig. 2.5 shows the stack of plates with top of the stack. After catering person puts plates on a stack, next guest will take a plate only which was placed last following LIFO order.

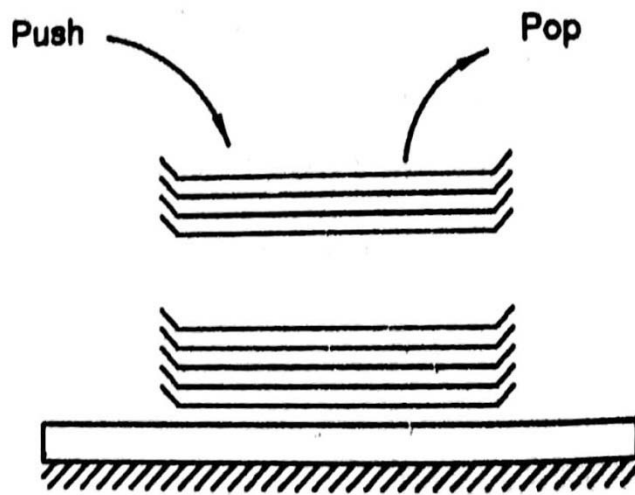


Fig. 2.5 : Stack of plates on a table.

Another example is a parcel or a container in which multiple pieces of the same items are placed on one another. Let us consider a container of books containing 10 copies of a same book. While placing the books in a container, we put books on one another, like first book at bottom, next book on top of it and it is followed for all 10 books with 10th book on top. When we want to remove books from container, the first book we can remove is the book that was placed last. The second book we can remove is the book we placed second last in a container. Fig. 2.6 shows the container of a book with push and pop operation.

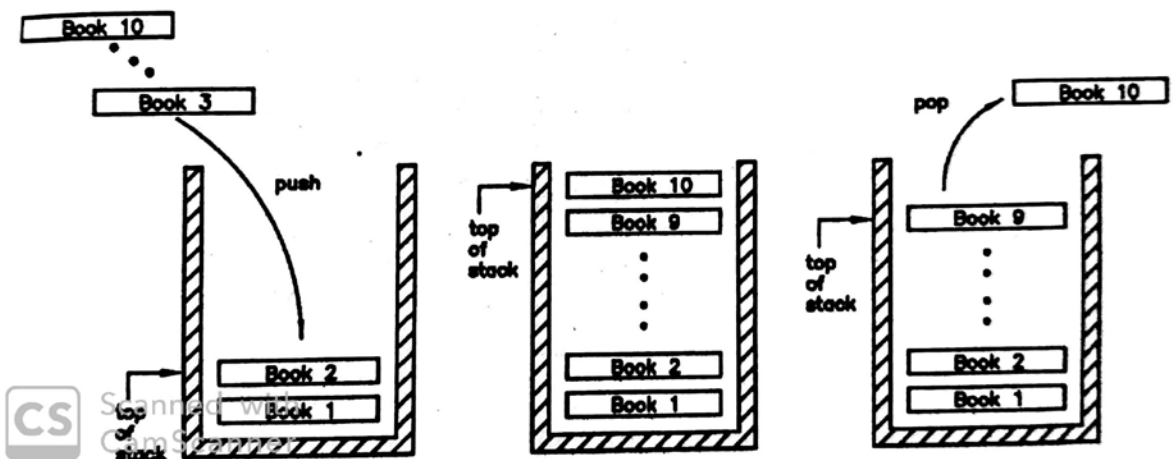


Fig. 2.6 : Container of books with push and pop operation.

Representation of Stacks :

A general representation of a stack is shown in fig. 2.7 with bottom and top representing bottom end and top end of a stack. The top of the stack is represented by a variable called **top** which contains the position of the last item placed in a stack. Sometimes it is also called top pointer as it points to the last item in a stack. The insertion(push) and deletion(pop) is performed only at top of the stack as shown in fig. The insertion increments the top of stack, while deletion decrements the top of the stack.

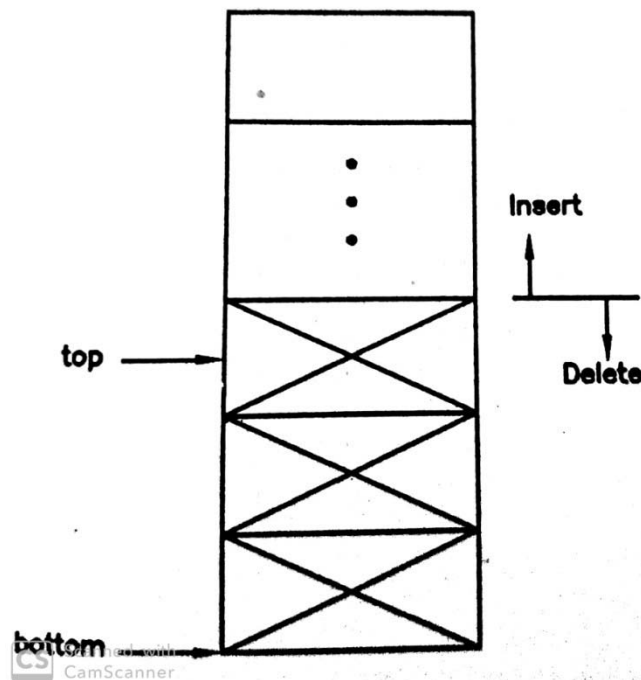


Fig. 2.7 : General representation of a stack.

One of the ways to implement the stack is using an array. Let us consider a stack with total size $N = 5$. That means maximum number of data items at a time resides in a stack is 5. Fig. 2.8(a) shows the stack initially when it is empty i.e. there is no items in a stack. We can use array index as position of the data item in a stack and top of the stack contains the index or position of the last data item in a stack. Initially, as the stack is empty, **top** contains 0 assuming that array index starts from 1. Fig. 2.8(b) shows the stack after 3 three data items have been pushed on to the stack and fig. 2.4(c) shows the stack when all the positions are filled and **top = 5** which equal to size N .

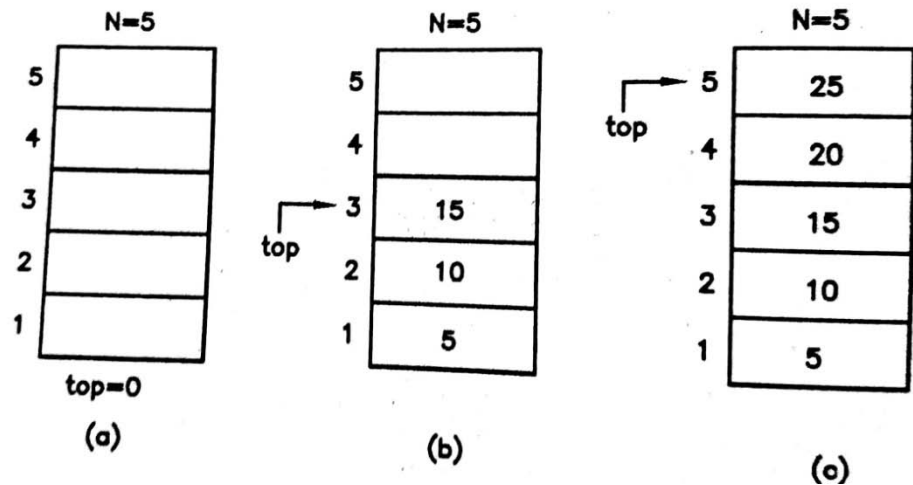


Fig. 2.8 : Stack using array with size $N = 5$

If we use C array to implement stack, initial value of the **top** is -1 as C starts array index from 0 and the last position is 4 (which is 1 less than size i.e. $N-1$). Fig. 2.9 (a), (b) and (c) shows the stack initially, after 3 items pushed and when it is full using C array. In rest of the chapters we will use only C arrays to implement the stack.

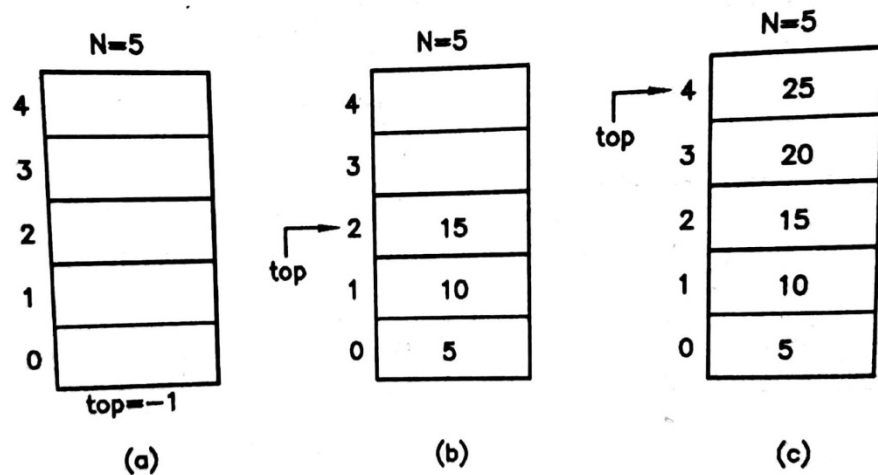


Fig. 2.9 : Stack using C array with size $N = 5$

2.5 OPERATIONS ON STACKS

Two major operations performed on stack are:

- push (insert)
- pop (delete)

The push operation inserts a data item into the stack at the top of the stack and pop operation removes a data item from the top of the stack.

Push operation first increments the top of the stack and then inserts data value at new position. But what happens if stack is already full i.e. top points to the last position ($top = N-1$)? In this situation, we can not perform push operation and this situation is called “Stack Overflow”. Hence, an attempt to push the data item on to stack when stack is full is known as “**Stack Overflow**”. Similarly, pop operation removes the data item from the current top position and then decrements the top of stack. What happens if we try to pop a data item from stack when stack is empty ($top = -1$)? This situation is known as “Stack underflow”. An attempt pop the data item from empty stack is known as “**Stack Underslow**”. Following example will clarify the idea.

Example : Consider a stack with size $N = 3$. Perform the following series of operations.

Push 10, Push 20, Pop, Push 30, Push 40, Pop, Pop, Pop, Pop, Push 50

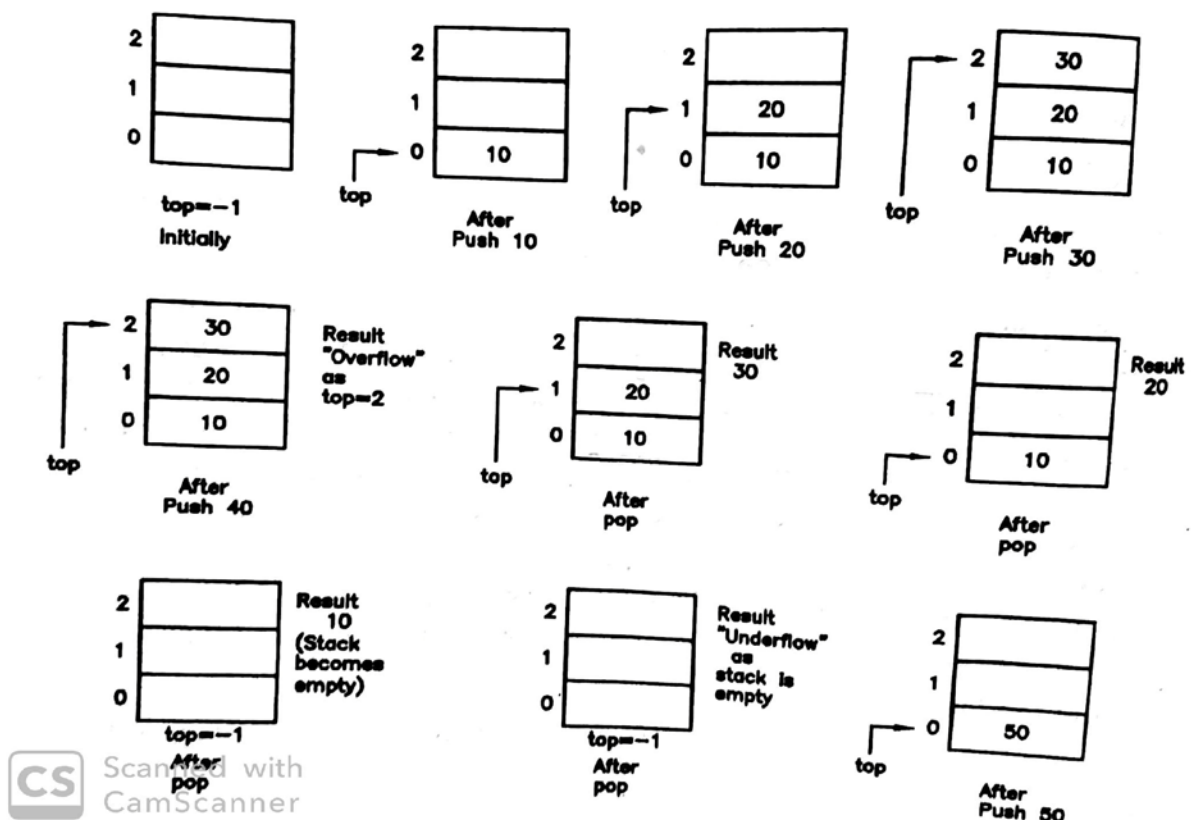


Fig. 2.10 : Stack position after each operation

Fig. 2.10 shows the position of stack after each operation. Observe that performing operation “Push 40” results in overflow as stack was already full. Similarly, last pop operation results into underflow as there were no items in stack.

After understanding the concepts behind the stack and its operations : push and pop, we are now able to formalize the algorithms for both the operations. The algorithms are presented as C functions. We assume here that, user of these algorithms define the stack as array of size N and implements top of the stack as integer variable whose address is passed as argument to both these operations so that the change made by the algorithms are directly reflected to a top of the stack maintained by user.

Following are important steps to write an algorithm to push a data item on to a stack.

1. If **top** is already at last position ($N - 1$), stack is full and we cannot push the new data and return message “overflow”.
2. Otherwise, increment top pointer.
3. Insert new data item.

The C function to push a data item into a stack is given as follows.

```

/* Push operation accepts the following arguments
   s   : Array implementing stack
   top : Pointer to top of the stack
n   : Size of the stack
   val : Data item to be pushed
*/
void push(int s[], int *top, int n, int val)
{
    /* Check the overflow */
    if(*top >= n - 1) {
        printf("Stack overflow\n");
        return;
    }

    /* increment the top to next free position */
    *top = *top + 1;

    /* insert the new value */
    s[*top] = val;

    /* return */
    return;
}

```

To push a data item on the stack, above function is called as

```
push(s,&top,N,data);
```

where **s** is a stack of size **N**, having top of stack defined as **top** whose address is passed to the function and **data** is the new data item to be pushed on to the stack.

Following are important steps to write an algorithm to pop a data item from a stack.

1. If **top** is -1, the stack is empty and return message “underflow”.
2. Delete a data item from top and store into temporary variable.
3. Decrement the **top** pointer.
4. Return the deleted data item.

Assume that our stack stores only positive integers, pop operation returns -1 whenever overflow occurs. The C function to pop a data item from stack is

```
/* Pop operation accepts the following arguments
   s   : Array implementing stack
   top : Pointer to top of the stack
and returns
   integer value (data item or -1)
*/
int pop(int s[], int *top)
{
    int val;

    /* check underflow */
    if(*top <= -1) {
        printf("Stack overflow\n");
        return(-1);
    }

    /* get the value from stack */
    val = s[*top];

    /* decrement the top of the stack */
    *top = *top - 1;

    /* return the data item */
    return(val);
}
```

To pop a data item from the stack this function is called as

```
data = pop(s,top);
```

where the popped value is returned into variable **data** if stack is not empty, otherwise it will be -1.

Note that to use the above functions push() and pop() in program, you have to define the stack as an integer array and initialize the top of the stack to -1 as C starts index from 0. For example,

```
int s [N];    /*stack of size N */
int top = -1  /* top of stack */
```

Program1 :

Write a menu driven C program to implement the stack.

```
#include <process.h>

#define N 4

void push(int [],int *,int,int);
int pop(int [],int *);
void display(int [],int);

void main()
{
    int s[N];    /* stack of size N */
    int top = -1; /* top of stack */
    int data,choise;

    while(1) {

        clrscr();
        printf("    1. Push\n");
        printf("    2. Pop\n");
        printf("    3. Display\n");
        printf("    4. Exit\n");
        printf("    Enter your choise : ");
        scanf("%d",&choise);

        switch(choise) {
            case 1: printf("Enter a value to push : ");
                    scanf("%d",&data);
                    push(s,&top,N,data);
                    break;
            case 2: data = pop(s,&top);
                    if(data != -1)
                        printf("Poped %d\n",data);
                    break;
        }
    }
}
```

```

        case 3: display(s,top);
                break;
        default: exit(0);
    }

    getch();
}

/* Function to push a data item on to stack */
void push(int s[], int *top, int n, int val)
{
    /* Check the overflow */
    if(*top >= n - 1) {
        printf("Stack overflow\n");
        return;
    }

    /* increment the top to next free position */
    *top = *top + 1;

    /* insert the new value */
    s[*top] = val;

    /* return */
    return;
}

/* Function to pop a data item from stack */
int pop(int s[], int *top)
{
    int val;

    /* check underflow */
    if(*top <= -1) {
        printf("Stack underflow\n");
        return(-1);
    }

    /* get the value from stack */
    val = s[*top];

    /* decrement the top of the stack */
    *top = *top - 1;

    /* return the data item */
    return(val);
}

/* Function to display contents of stack */

```

```

void display(int s[],int top)
{
    if(top <= -1)
        printf("Stack Empty\n");
    else {
        /* Display contents of stack */
        printf("\t\t\tTop --> ");
        while(top >= 0) {
            printf("%d\n",s[top]);
            printf("\t\t\t\t");
            top = top - 1;
        }
    }
}

```

In above implementation, we used array i.e. sequential allocation to implement a stack. It can also be implemented using linked allocation with linked list.

2.6 APPLICATIONS OF STACKS

This section discusses two major applications of stack. The first application is the recursion and second application is the evaluation of postfix expressions. Recursion is one of the most popular application of the stack and many programming languages including C supports recursion as programming technique. Programming language uses infix expression which are not suitable for efficient evaluation for computing its value. Infix expressions are internally converted to postfix expression by compilers so that they can be efficiently evaluated to their final values using stack.

Recursion :

Recursion is a technique which defined task or process in term of itself. In term of programming languages, recursion is a technique in which function calls itself. The meaning of both is one as a task or process in when implemented in programming is represented as function. The most popular example of recursion is factorial. It is defined as follows.

$$\text{Factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{Factorial}(n - 1), & \text{Otherwise} \end{cases}$$

Observer here that n factorial is defined in terms of $n - 1$ factorial. Similarly, $n - 1$ factorial is defined in terms of $n - 2$ factorial and so on. Finally, when $n = 1$, it is defined in terms of factorial of 0 which is 1. For example,

$$\begin{aligned}4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 \times 0! \\ &= 4 \times 3 \times 2 \times 1 \times 1\end{aligned}$$

There are two important issues in recursion. First issue is the base or terminating condition. The base or terminating condition in factorial is the fact that it is 1 when $n = 0$ i.e. $0! = 1$. As recursion causes the chain of processes, if it is not terminated then it is likely to go into infinite loop. Thus, terminating condition terminates one of the process (in our case when $n = 0$) so that the whole chain will reverse back and completes the computation which in turn finally returns us the factorial of given number. The second important issue is that task or process when defines itself, each time it should lower the value so that we can move towards the solution. In our example, the n factorial is defined in terms of $n - 1$ factorial and hence every time n is reduced by 1 which finally reached to 0 and the base condition is achieved.

The C code for the factorial of n is as follows.

```
int fact(int n)
{
    int f = 1;

    if(n == 0)
        return(f);
    f = n * fact(n - 1);
    retrn(f);
}
```

Fig. 2.11 shows the working of the function `fact()` for $n = 3$. Each invocation of function is numbered with first call made by user/main function is 1, second call made from call 1 is 2 and so on. This calls create a chain of function calls and finally when last call i.e. call 4 observes the base condition i.e. $n = 0$, it returns back to its

caller (with call 3 and reverse chain is created. That reveals the important fact that in recursive process calls follow LIFO order (Calls returns in opposite order of making calls) which is the behavior of the stack. But what is stored here in stack and retrieved back? In initial part of body of each call, current value of n is pushed on to stack, so that after making next inner call from it, n remains safe and once the inner call returns, to compute $n * (n - 1)!$, n is read from stack. Finally, in end part of body values pushed on stack are popped. In reality, recursive calls stores and retrieves arguments, return address as well as local variables. For simplicity, in fig. 2.11, only stack with n is shown. Call 1 pushes $n = 3$ on to the stack, call 2 pushes $n = 2$, call 3 pushes $n = 1$, call 4 pushed $n = 0$. Call 4 observes the base condition and returns with result 1 to call 3. But before it returns to call 3, it pops back $n = 0$ from stack, so that call 3 sees $n = 1$ on top which it has pushed. It uses it from stack to compute result which is $1 * 1 = 1$ and returns back to call 2 with result 1 after popping $n = 1$ from stack. Call 2 sees $n = 2$ on top which it uses to compute $2 * 1 = 2$ and returns with result 2 to call 1 after popping $n = 2$ from stack. Call 1 sees $n = 3$ on top which it uses to compute $3 * 2 = 6$, and returns result 6 to user after popping $n = 3$ from stack.

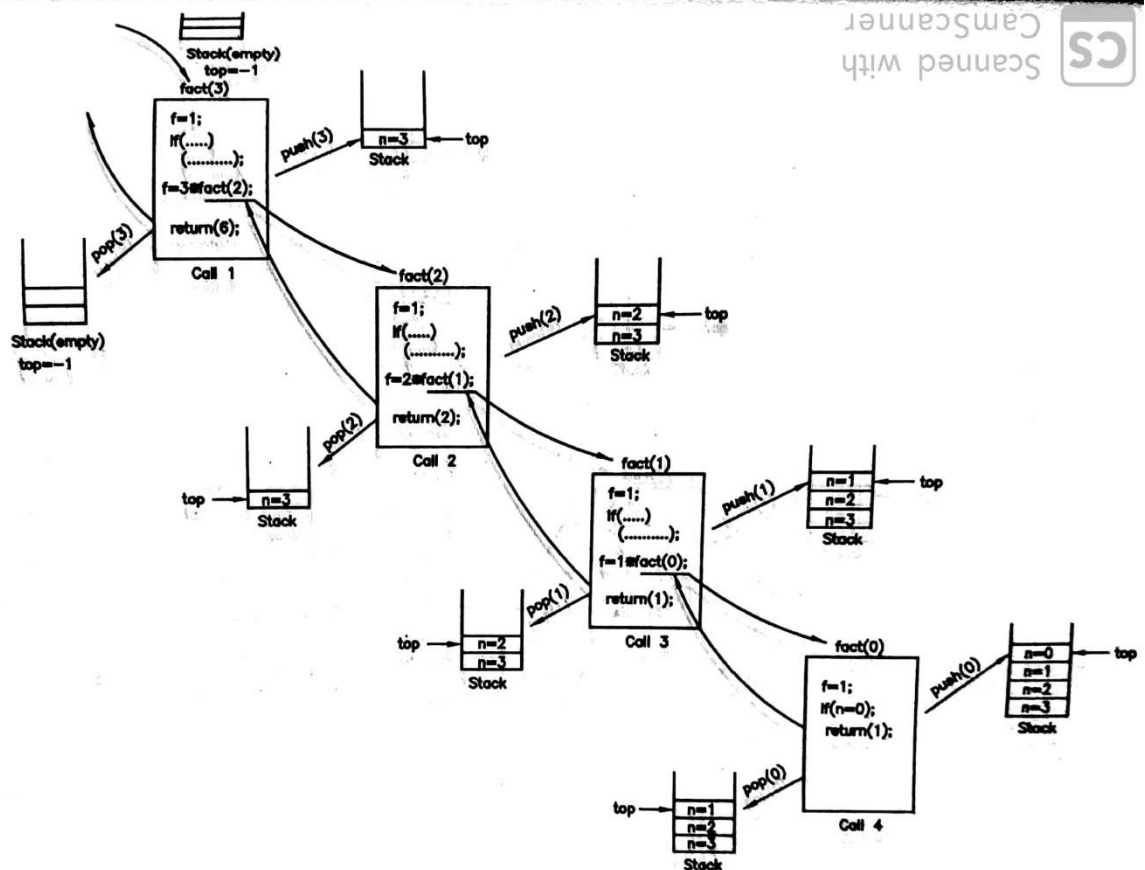


Fig. 2.11 : Working of fact() for n = 4 with stack

Taking above discussion in consideration, we can represent, the recursive process in general form using flowchart as shown in fig. 2.12. As shown in fig., initial part stores the arguments, return address and local variables on stack. The body part tests the base condition and if it is not reached then next recursive call to same process is made. If base condition is reached then process moves into last part which restores all the values form stack and using return address, returns back to the caller.

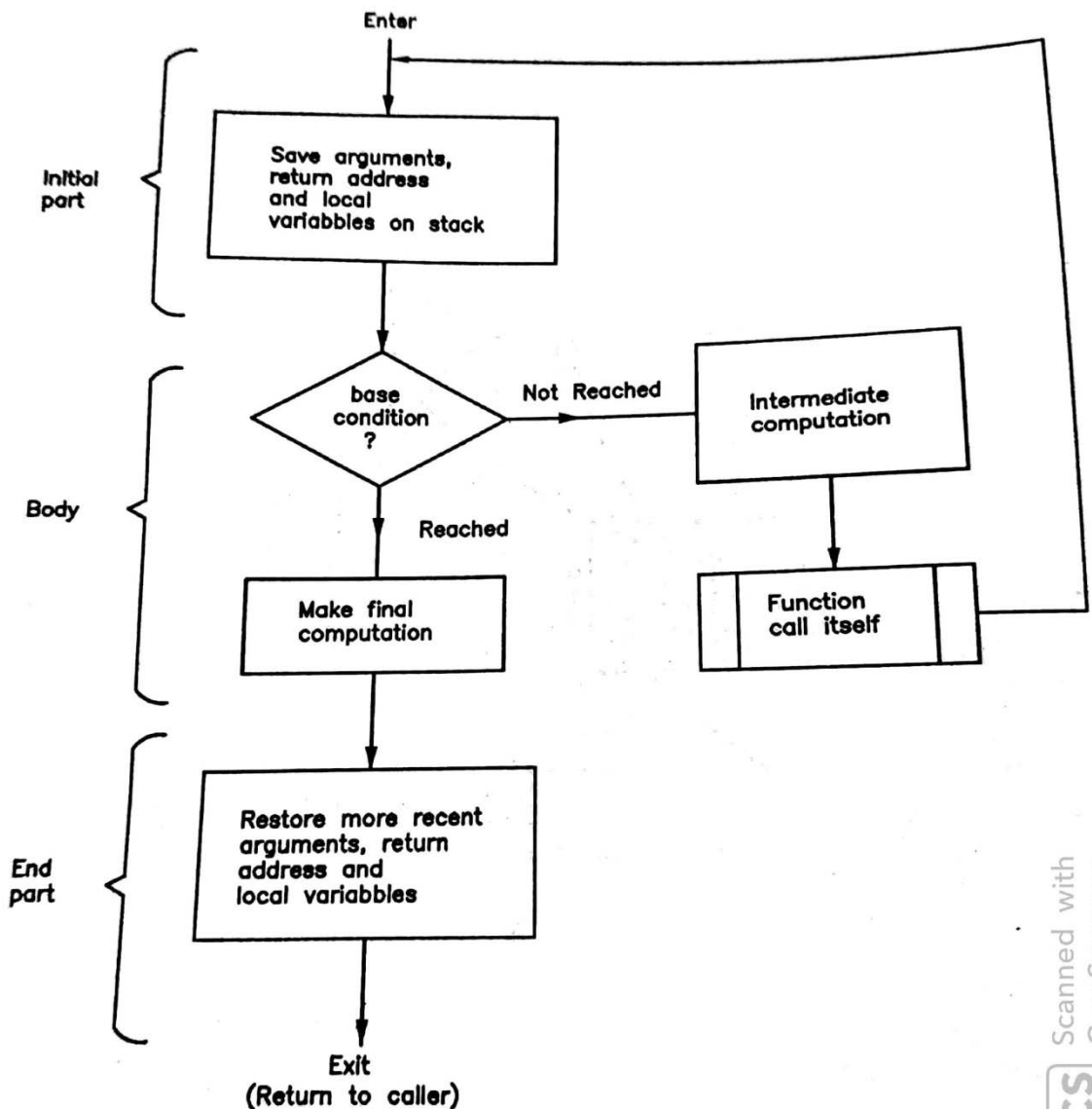


Fig. 2.12 : Recursive process

Program2 :

Write a Fibonacci function using recursion and use it in main to generate N Fibonacci numbers.

```
#include <stdio.h>
int fact(int);

void main()
{
    int f,n;

    /* Get the number */
    puts("Enter a number");
    scanf("%d",&n);

    /* Find the factorial and print */
    f = fact(n);
    printf("Factorial of %d is %d\n",n,f);
}

/* Recursive function to compute factorial */
int fact(int n)
{
    int f = 1;
    if(n == 0)
        return(f);
    f = n * fact(n-1);
    return(f);
}
```

Recursive processes discussed above can also be solved using iterative method. Iterative methods repeat the same task or process with modified values of variables each time. This is same as performing looping in programming. For example, for loop in C is an example of iterative process. Consider the following for loop making sum of 1 to 10.

```
sum = 0;
for(i=1;i<=10;i++) {
    sum = sum + i;
}
```

In above example, Initially sum is initialized to 0 and i is initialized to 1. As long as i is less than or equal to 10, current value of i is added to sum. Then i is modified and again the process is repeated. Hence, this process has three important parts : initialization, body (task or process) and modification. The body is executed

as long as condition is true. The general form of the iterative process is shown in fig. 2.13 using flowchart.

CS
Scanned with
CamScanner

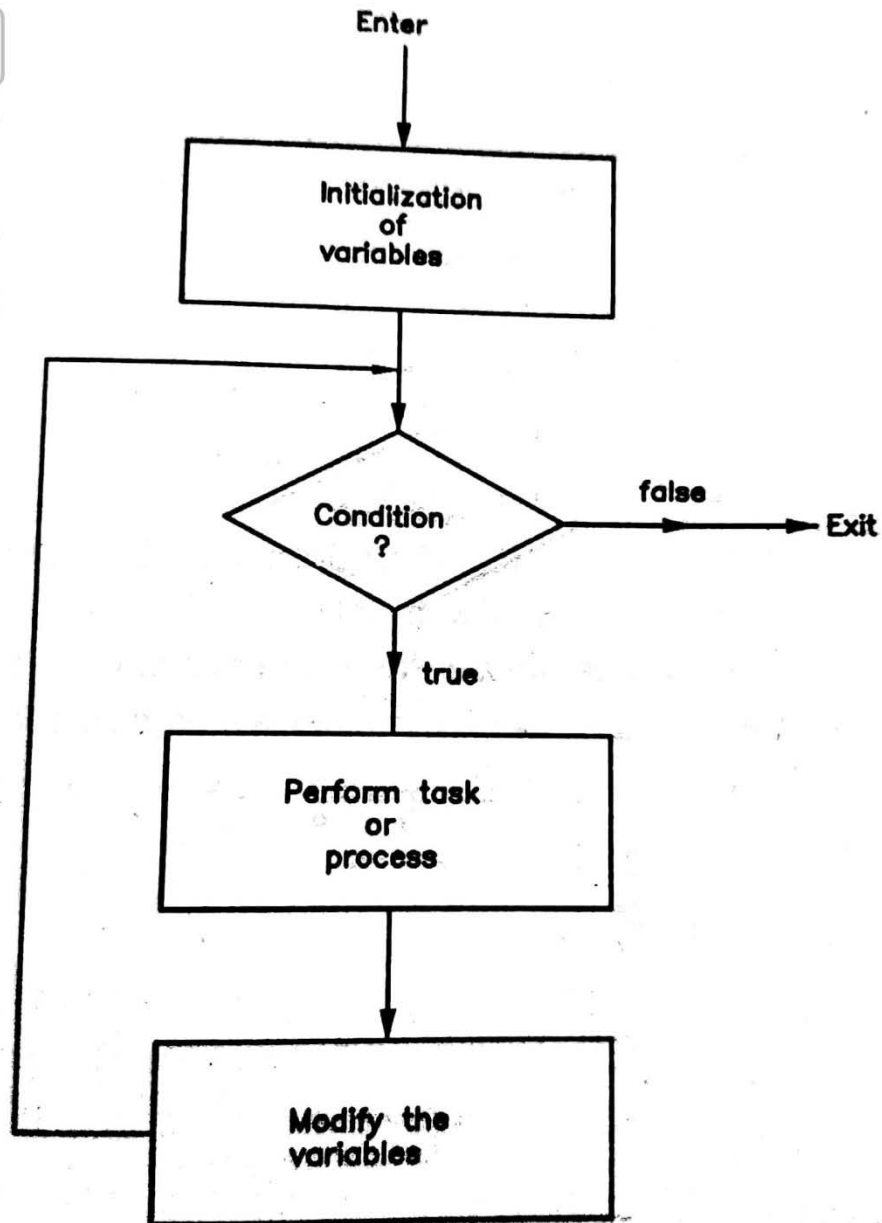


Fig. 2.13 : Iterative process

The following C code gives the factorial function using the iterative method.

```
int fact(int n)
{
    int f;

    for(f=1;n>0;n--) {
        f = f * n;
    }
}
```

```

    }
    return(f);
}

```

Postfix expressions and their evaluation :

We are aware of the expressions used in programming languages, which are, infix expressions having following form.

Operand OperatorOperand

Let us consider how C infix expressions are evaluated. To evaluate an arithmetic expression properly, we have to use the precedence and associativity of the operators involved in expression. The precedence and associativity of arithmetic operators in C language are given below.

Precedence	Associativity
* /	left to right
+ -	left to right

The precedence of * and / is same but higher than + and -. The precedence of + and - is same. In both cases the associativity is left to right. This means if in same expression two or more operators with same precedence appears than they are evaluated from left to right as they occurs in expression. Consider an expression

$$a + b/c - d * e \quad \text{where } a = 10, b=6, c=2, d=8 \text{ and } e=13$$

It is evaluated as follows.

	$a + b/c - d * e$
Step 1:	$10 + \underline{6/2} - 8 * 13$
Step 2:	$10 + 3 - \underline{8 * 13}$
Step 3:	$\underline{10 + 3} - 104$
Step 4:	$13 - 104$
Step 5:	-91

Here, / and * are done first as their precedence is higher. Then + and - are done left to right. It is shown in fig. 2.114.

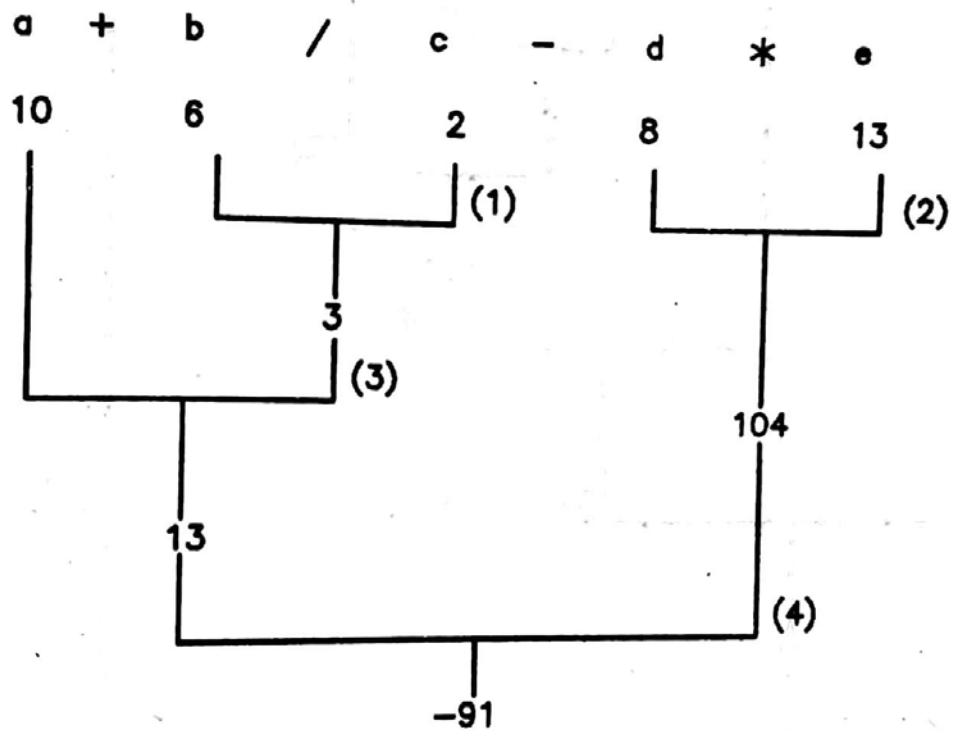


Fig.. 2.14 : Evaluation of $a + b / c - d * e$

Many times we want the priority of an operator to be changed as per our requirements. For example, $a+b*c$ is evaluated always with $*$ has higher precedence and the result is $2+5*10 = 52$ if $a = 2$, $b = 5$ and $c = 10$. But if we want a and b to be added first and its sum to be multiplied with c , then expression must be written as $(a+b)*c$. As $()$ has got highest precedence, the sub-expression in $()$ is done first. Hence, the answer is $(2+5)*10 = 70$. The $()$ allows user to write an expression with his/her own meaning. If $()$ are nested one inside other, then they are done from inner most to outer. Fig. 2.15 shows the evaluation of expression

$$a + (b / ((c + d) * e)) - f \quad \text{where } a=9, b=216, c=3, d=6, e=12, f=10$$

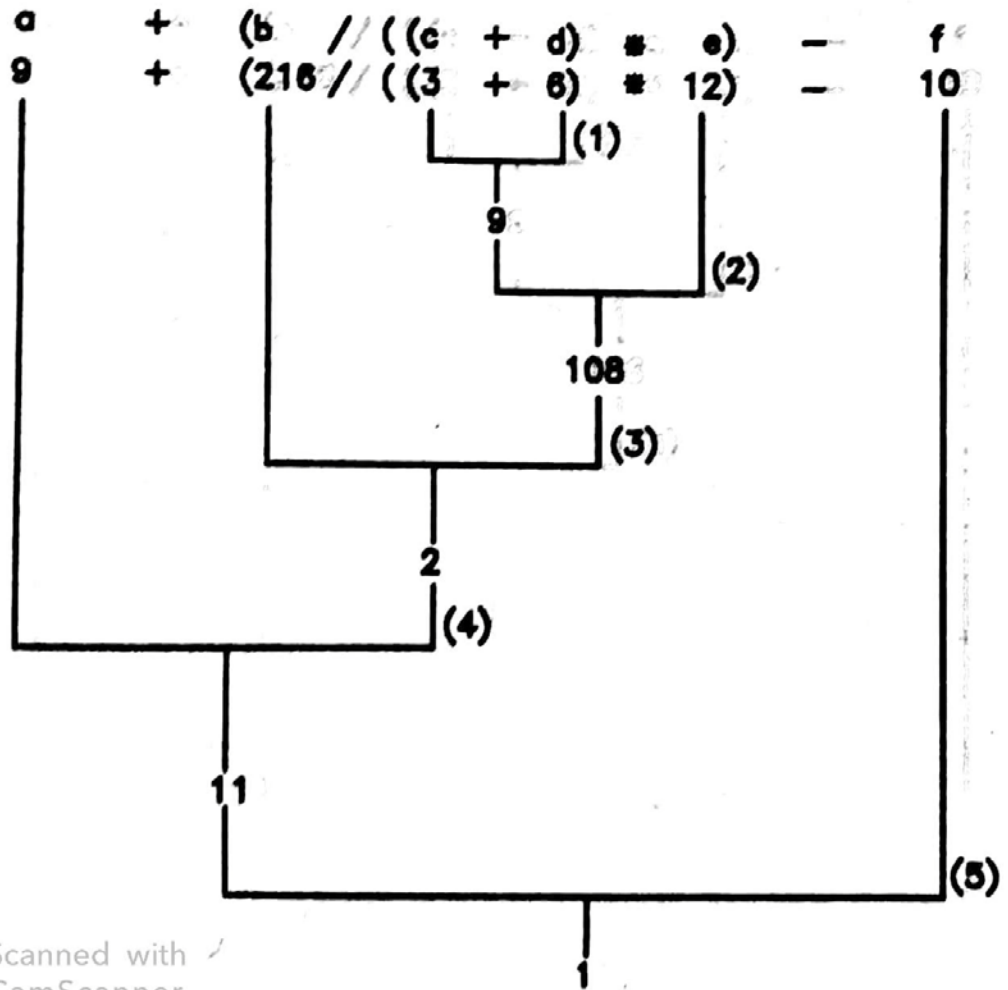


Fig. 2.15 : Evaluation of $a + (b / ((c + d) * e)) - f$

In fig. 2.14 and 2.15, the order in which operations performed are indicated using numbers in bracket. Consider fig. 2.11, the first operation performed is $c + d$, then second operation is $T * e$ where T is answer of $c + d$. To determine which operation is to be performed next, repeatedly expression is scanned from left to right and the operator having highest precedence amongst all is considered with their operands to perform the operation. This makes the process very slow, as complex expressions require more number of scanning of expression. We can solve this problem by first converting infix expression to postfix expression (also known as polish notation) before evaluation. Postfix expression has following form.

Operand Operand Operator

The advantage of the postfix expression is that whole expression is evaluated by scanning only once from left to right with the help of the stack. Before we learn,

how to evaluate postfix expression using stack, let see how infix expression are converted into postfix expression. While converting from infix to postfix, we have to consider precedence, associativity and (). Consider following infix expression and their postfix conversions.

Infix	Postfix
$a + b$	$a b +$
$a + b * c$	$b c * a +$
$(a + b) * c$	$a b + c *$
$a + b / c - d * e$	$b c / a + d e * -$
$a + (b / ((c + d) * e)) - f$	$a b c d + e * / + f -$

The validity of the postfix expression is checked easily by calculating its rank. Rank of operator (+, -, *, /) is -1 and operand is 1. Rank of expression is sum of rank of all the operators and operands. If it 1, the postfix expression is valid, otherwise it is invalid. All example given above are valid as rank of all the postfix expressions is 1.

Let us now learn how to evaluate postfix expression using stack in only single scan from left to right. While scanning postfix expression from left to right

- If symbol is operand, push it on to the stack.
- If symbol is operator, pop two operands from top of stack and perform operation indicated by operator and push the answer on stack.

When all the symbols are over, the value in stack is the answer.

Example : Evaluate expression : $a + b / c - d * e$ where $a = 10$, $b=6$, $c=2$, $d=8$ and $e=13$.

Postfix expression : $b c / a + d e * -$

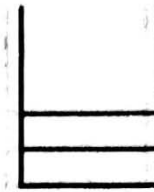
The postfix expression is scanned form left to right one symbol at a time. The stack at each symbol is shown in fig. 2.17. The final answer is -91. ■

Symbol

Operation

No

No



Initially

top = -1

b

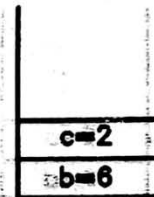
push(b)



top = 0

c

push(c)



top = 1

/

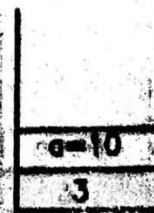
pop(c), pop(b)
b/c, push(b/c)



top = 0

a

push(a)



top = 1

Scanned with
CamScanner



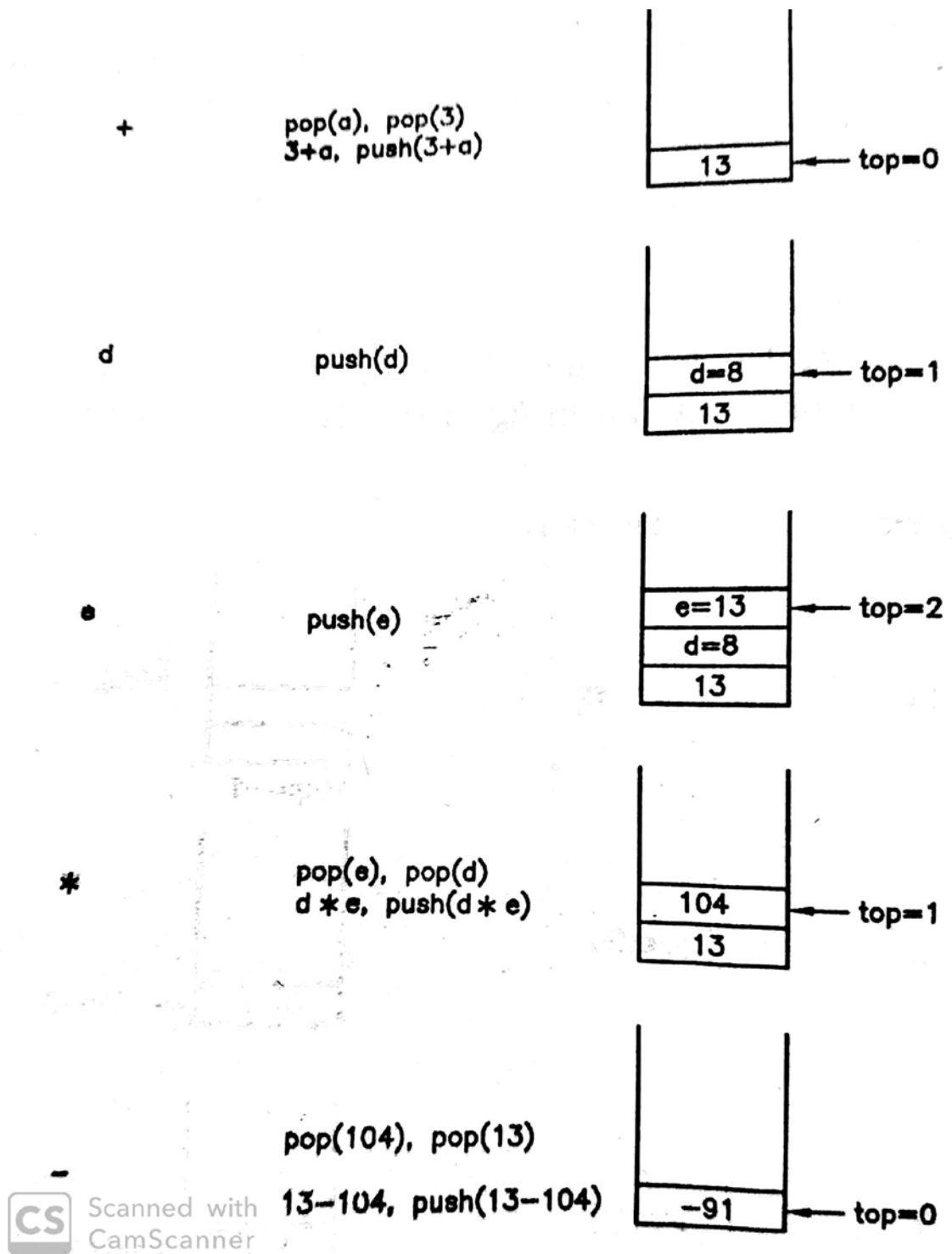


Fig. 2.17 : Evaluating postfix expression using stack.

2.7 LET US SUM UP

Array : Arrays are used to store an ordered set of data elements. All the data elements stored in an array should be of the same type i.e. all the elements are

homogeneous. An array provides a single name to a group of ordered elements of same type.

Addressing function : Addressing function is an expression used to compute the memory location of a given element into an array.

Row-major order : Storing a two-dimensional array row-by-row.

Column-major order : Storing a two-dimensional array column-by-column.

Stack : A stack is a linear data structure with restriction that it allows operations to be performed only at one end known as top of the stack. It follows LIFO (Last In First Order).

Top of the stack : It is an pointer to point to the last element in a stack.

Push : Push is an operation to insert an element into stack at the top of the stack.

Pop : Pop is an operation to delete an element from the top of the stack.

Stack overflow : If an attempt is made to push an element into a stack when it is full, it is called overflow.

Stack underflow : If an attempt is made to delete an element from empty stack, it is called underflow.

Recursion : Recursion is a technique which defined task or process in term of itself. In term of programming languages, recursion is a technique in which function calls itself.

2.8 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. Array is an example of data structure.
2. Addressing function of an array is used to compute of an element into the array.
3. C uses order for storing two-dimensional array.
4. Stack follows order for insert and delete operations.
5. Trying to push a new data item in a stack which if full result in condition.

6. A condition occurs when pop operation is performed on empty stack.
7. operation on a stack decrements the top of the stack.
8. data structure allows operation only at one end, while data structure allows operations at both the end.
9. Expressions used in programming languages are expressions.

➤ **True-False**

1. All the elements in an array must be of same type.
2. C uses column-major order to represent the two-dimensional array.
3. Array is an example of linked address method.
4. Stack is linear data structure while queue is non-linear data structure.
5. Recursion gets into infinite loop if terminating condition if not implemented properly.
6. Initial value of top of stack is only -1, whenever array is used to implement stack.

➤ **Answer in brief**

1. What do you mean by row-major order representation of an array? Give example.
2. What is an addressing function for an array? What is use of it?
3. Give the addressing function for one-dimensional array and briefly describe it.
4. Why is stack linear data structure?
5. What do you mean by underflow and overflow conditions in stack?
6. What is top of stack? Give its significance.
7. What is recursion? Give two examples.
8. What is an advantage of postfix expression?

➤ **Answer the following**

1. Explain the row-major and column-major representations of a two-dimensional array.
2. Assume that C array is declared as

```
int a[10];
```

Find the address of a[5] if the base address is 5000.

3. Assume that C array is declared as

```
double a[4][6];
```

Find the address of a[2][3] if the base address id 3000.

4. What is stack? Explain the operations performed on stack.
5. Give the steps to perform push and pop operations on stack.
6. Explain the push and pop algorithms using C.
7. Explain at least one application of stack.
8. What is recursion? What are the important issues associated with stack.
9. Explain iterative process with example.

10. Convert following expressions to postfix and evaluate using stack

$(a + b) / (c - d)$ where $a = 10, b = 5, c = 4, d = 1$

$a - (c / (d + e)) * f$ where $a = 50, c = 25, d = 3, e = 2, f = 5$

11. Write short notes

1. Addressing function of arrays
2. Representation of two-dimensional arrays
3. Stack
4. Recursion

2.9 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

2.10 ASSIGNMENTS

1. Write a menu driven program for stack of integers with the push() and pop() functions provided in the unit.

2. Develop a C program which implements a stack of characters with push and pop operation. Use it to accept a string from keyboard and print it in reverse.
3. Develop a C program to implement double stack. Double stack uses one array with two top pointers one moves from start towards end and second moves from end towards start. This is same as implementing two stacks. Push and pop can be performed in any one based on choice. Whenever both pointers are at adjacent location, stack is full.
4. Develop a C program having a recursive function to print first N Fibonacci numbers.
5. Develop a C function which prints given string in reverse using recursion. Use it in main to reverse a string enter from keyboard.

Unit 3: Queue

3

Unit Structure

- 3.1 Learning Objectives
- 3.2 Introduction
- 3.3 Queues
- 3.4 Operations on Queues
- 3.5 Circular Queues
- 3.6 Operations on Circular Queues
- 3.7 Priority Queue
- 3.8 Let us sum up
- 3.9 Check your Progress
- 3.10 Further Reading
- 3.11 Assignments

3.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define queue with its representation.
- List and perform the various operations on the linear queue.
- List the limitations of the linear queue.
- Define the circular queue and list its benefits.
- List and perform the various operations on the circular queue.
- Define the priority queue with its functioning.

3.2 INTRODUCTION

Another very important linear data structure is Queue which allows operations at both the ends and follows FIFO (First In First Order). This unit discusses the basic concepts of queues with its applications in real life. The most important operations performed on queue are insert and delete. Insert operation is performed at rear end while delete operation is performed on front end as queue allows operations on both the ends. To overcome the limitations of linear queues, circular queues are used where operation is performed from start once you reach to the end. At last chapter ends with discussion of the fundamentals of priority queues. The C implementations of the queue operations are provided for the better understanding.

3.3 QUEUES

A queue is a linear data structure which allows insertion and deletion operations to be performed at different end. The insertion operation performs the operation at rear end while deletion operation is performed at front end. To keep track of front and rear end, queue maintains two pointers known as **front** and **rear** pointers. The delete operation deletes a data item from the front end and increments the front pointer. The insert operation inserts new data item at the rear end and increments the rear pointer. Whenever we refer a queue, we refer a linear or simple queue. We will discuss about circular queue also.

Let us understand the real life examples of a queue. These include a queue at the petrol pump, a queue at the railway reservation center, a queue at the cinema house for ticket booking etc. In case of an example of railway reservation center, there is a separate queue at each window and once ticket booking clerk completes request for one person, he considers the request for a next person in front of the queue and once he/she is served leaves the queue which is same as deleting an item from a queue. A new person who wants to book the ticket from same window joins the queue after last person at rear end which is same as insertion in a queue.

Representation of Queues :

A general representation of a queue (simple or linear) is shown in fig. 3.1 with **front** denotes the pointer to front end and the **rear** denotes the pointer to rear end. The insertion operation is performed at rear end and the delete operation is performed at front end.

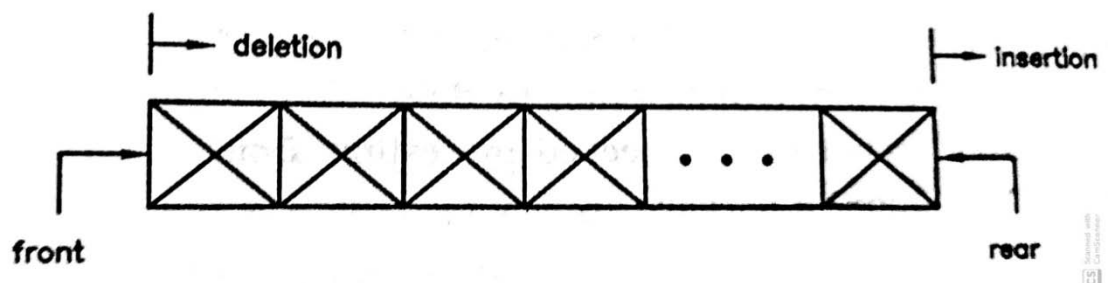


Fig. 3.1 : General representation of a queue

One of the ways to implement a queue is to use the array. Let us consider the queue with maximum size $N = 5$. The **front** and **rear** pointers are implemented as integer variables holding the index of the data item at the front end and rear end respectively. Assuming that C array is used for implementation, initially when queue is empty, the **front** and **rear** pointers are set to -1 as shown in fig. 3.2(a). Fig. 3.2(b) shows the queue after 3 data items inserted and fig. 2.14(c) shows the queue when there is only one data item in a queue. Observe that **front** and **rear** both points at same position in this case. Remember that front and rear can never exceed $N - 1$.

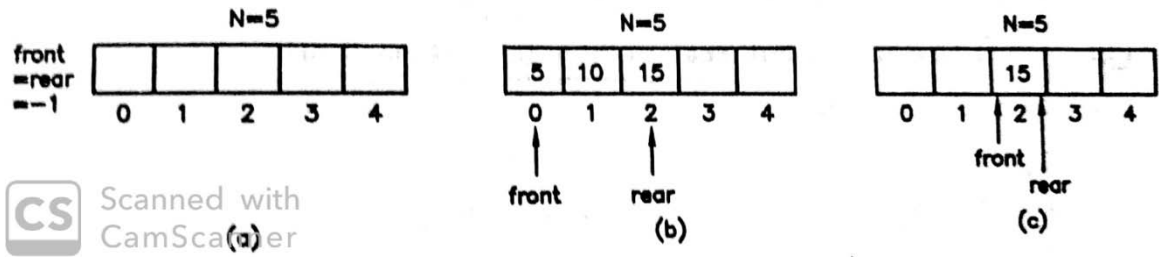


Fig. 3.2 : Queue using C array with N = 5

3.4 OPERATION ON QUEUES

There are two important operations performed on a queue :

- Insert
- Delete

The insert operation insert a new data item at rear end and delete operation deletes a data item from the front end.

Let us consider the example with series of insert and delete operations to understand how the front and rear pointers are manipulated. Fig. 3.3 shows the series of insert and delete operations on queue using C array with size N = 3.



Scanned with
CamScanner

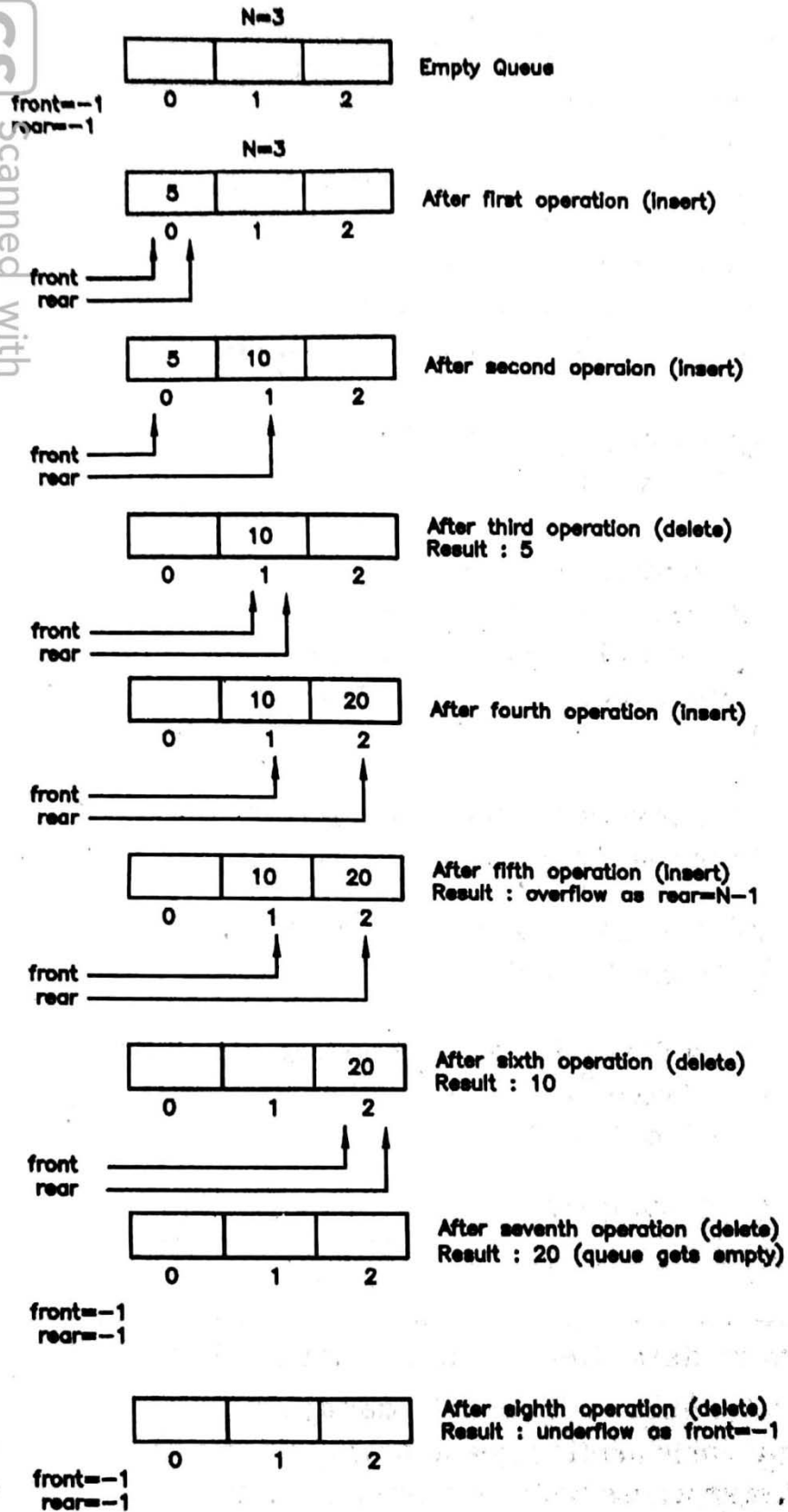


Fig. 3.3 : Insert and delete operations on a queue.

Observe in fig. 2.15 that the fifth operation results in an overflow even though there is one vacant space in queue at position 0. Once rear pointer reaches to last position i.e. $N - 1$, we can insert more elements in a queue even though space is available. This is the major drawback of the linear or simple queue as it does not allow the reuse of the space. We can reuse it only after getting it completely empty. Last operation shows that trying to delete from empty queue results in underflow. Underflow condition is checked by testing **front = -1** (rear must also be -1). Overflow condition is checked by testing **rear = N - 1**.

After understanding the concepts behind the queue and its operations : insert and delete, we are now able to formalize the algorithms for both the operations. The algorithms are presented as C functions. We assume here that, user of these algorithms define the queue as array of size N and implements front and rear pointers as integer variables whose addresses are passed as arguments to both these operations so that the change made by the algorithms are directly reflected to pointers maintained by user.

Following are important steps for writing an algorithm for inserting a data item in a linear or simple queue.

1. If **rear** is already at last position ($N - 1$), we can not insert new data item and return with message "overflow".
2. Otherwise increment **rear** pointer to next position
3. Insert new data item.
4. If it was first insert (before insert **front = rear = -1**), then increment **front** also as new data item will be only item after insert, both front and rear should be same.

The C function to insert a data item into a queue is as follows.

```
/* qinsert operation accepts the following arguments
   q      : Array implementing queue
   front  : Pointer to front end
   rear   : Pointer to rear end
```

```

        n      : Size of the queue
        val    : Data item to be inserted
*/
void qinsert(int q[], int *front, int *rear, int n, int val)
{
    /* check for overflow */
    if(*rear == n - 1) {
        printf("overflow\n");
        return;
    }

    /* increment the rear pointer */
    *rear = *rear + 1;

    /* insert new data item */
    q[*rear] = val;

    /* Is first insert? Then set front pointer */
    if(*front == -1)
        *front = 0;

    /* return */
    return;
}

```

To insert new data item into a queue, this function is called as

```
qinsert(q,&front,&rear,N,data);
```

where **q** is the array implementing queue having front and rear pointers defined as integers **front** and **rear** whose addresses are passed to function, **N** denotes the size of the queue and **data** is the value to inserted in a queue.

Following are the important steps for writing an algorithm for deleting a data item from a linear queue.

1. If queue is empty (**front = -1**), we can not delete an item and return with message "underflow".
2. Otherwise delete an item from front end.
3. If it was only a item in queue, make queue empty by setting **front = rear = -1**, otherwise increment **front** to point to next item in a queue.

The C function to delete a data item from a queue is as follows. Assume that queue stores only positive integers and returns -1 when there is an underflow.

```
/* qdelete operation accepts the following arguments
   q      : Array implementing queue
   front  : Pointer to front end
   rear   : Pointer to rear end
and returns
   integer value (data item or -1)
*/
int qdelete(int q[], int &front, int &rear)
{
    int data;

    /* check for underflow */
    if(*front == -1) {
        printf("underflow\n");
        return(-1);
    }

    /* delete an item and store in temporary variable */
    data = q[*front];

    /* set the front pointer */
    if(*front == *rear)
        *front = *rear = -1; /* make queue empty */
    else
        *front = *front + 1; /* forward pointer */

    /* return */
    return(data);
}
```

To delete a data item from a linear queue, this function is called as

```
data = qdelete(q,&front,&rear);
```

where **q** is the array implementing queue having front and rear pointers defined as integers **front** and **rear** whose addresses are passed to function. Variable **data** receives data item deleted or -1 if queue is empty.

Program 1 uses above functions to write a menu driven for linear queue of positive integers. It also implements function to display the whole contents of queue.

Program 1 :

Write a menu driven program to perform insert and delete operations on a linear queue of positive integers.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <process.h>

#define N 4

void qinsert(int [],int *,int *,int,int);
int qdelete(int [],int *,int *);
void display(int [],int,int);

void main()
{
    int q[N];      /* queue of size N */
    int front = -1, rear = -1;    /* front and rear pointers */
    int data,choise;

    while(1) {
        clrscr();
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choise : ");
        scanf("%d",&choise);

        switch(choise) {
            case 1: printf("Enter a value to insert : ");
                    scanf("%d",&data);
                    qinsert(q,&front,&rear,N,data);
                    break;
            case 2: data = qdelete(q,&front,&rear);
                    if(data != -1)
                        printf("Deleted %d\n",data);
                    break;
            case 3: display(q,front,rear);
                    break;
            default: exit(0);
        }

        getch();
    }
}

/* Function to insert a data item in a queue */
```

```

void qinsert(int q[], int *front, int *rear, int n, int val)
{
    /* check for overflow */
    if(*rear == n - 1) {
        printf("overflow\n");
        return;
    }

    /* increment the rear pointer */
    *rear = *rear + 1;

    /* insert new data item */
    q[*rear] = val;

    /* Is first insert? Then set front pointer */
    if(*front == -1)
        *front = 0;

    /* return */
    return;
}

/* Function to delete a data item in a queue */
int qdelete(int q[], int *front, int *rear)
{
    int data;

    /* check for underflow */
    if(*front == -1) {
        printf("underflow\n");
        return(-1);
    }

    /* delete an item and store in temporary variable */
    data = q[*front];

    /* set the front pointer */
    if(*front == *rear)
        *front = *rear = -1; /* make queue empty */
    else
        *front = *front + 1; /* forward pointer */

    /* return */
    return(data);
}

/* Function to display contents of queue */
void display(int q[], int front, int rear)
{
    if(front == -1)

```

```

        printf("\n\t\tQueue Empty\n");
    else {
        printf("\n\t\tFront( = %d)-->",front);
        while(front <= rear) {
            printf("%d ",q[front]);
            front++;
        }
        printf("<-- Rear(= %d)\n",rear);
    }
}

```

3.5 CIRCULAR QUEUES

The major drawback of the linear queue is that once the rear pointer reaches to end of the queue, we can not insert more data items, even though there is space. We can reuse the location only after queue gets empty. To overcome this problem, we can use circular queue. In circular queue, both the ends of the queue are joined together so that once the pointer, either front or rear reaches to the end of the queue, they can move to the start of the queue. This way we can reuse the same locations again and again. In this section, we will discuss the concept of circular key with its representation and working as well as we will redesign the insert and delete algorithms to suit to the circular nature of the queue.

Representation of Circular Queues :

A general representation of a circular queue is provided in fig 3.3. Insertion and deletion operation are performed in same fashion as the linear queue with only difference is whenever front or rear pointer reaches to last position, they are sent back to the starting position.

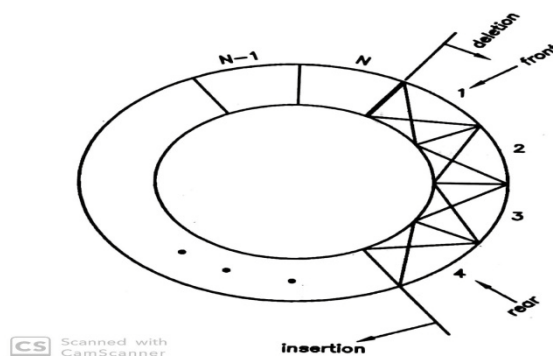


Fig. 3.3 : General representation of circular queue

Consider a C array with size $N = 8$ is used to implement a circular queue, fig. 3.4 shows the circular queue at different time. Fig. 3.4(a) shows empty circular queue with **front** and **rear** pointers are set to -1. Fig. 3.4(b) shows circular queue after 3 subsequent insert operations. Assuming that circular queue after some arbitrary number of insert and delete operations is like fig. 3.4(c) and we want to insert a new data item. The new data item will be inserted at first position as the rear end already at last position which is shown in fig. 3.4(d). This way we can reuse the same location repeatedly.

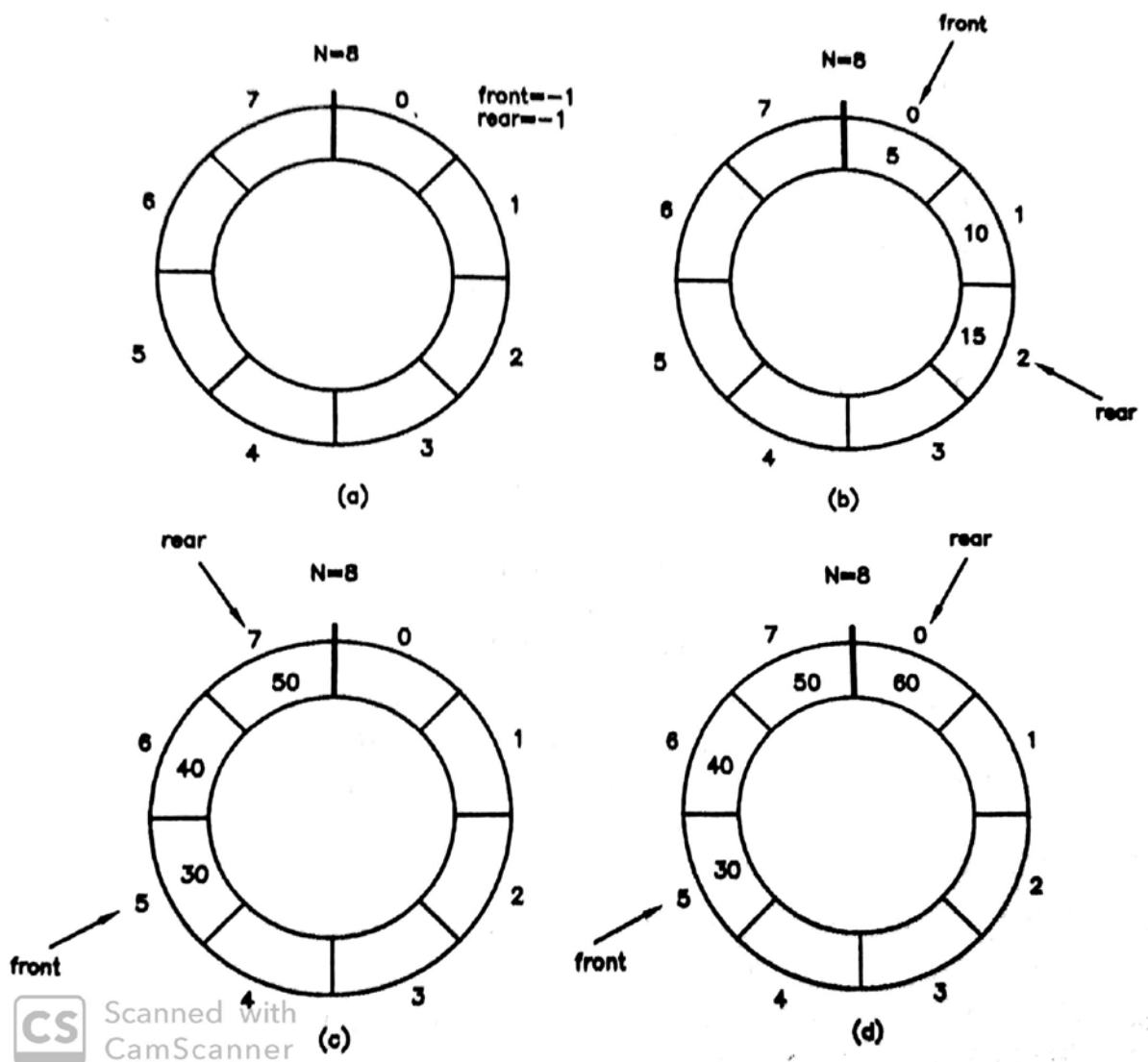


Fig. 3.4 : Circular queue with $N = 8$ in different situations

Attempt to delete from an empty circular queue results in an underflow situation which can be detected by testing the front or rear pointer for -1. How can we detect overflow while inserting in a circular queue that is full. There are two

situations when circular queue is full. One when front pointer points to first position and rear pointer points to last position which is shown in fig. 3.5(a). Second when rear and front pointer are adjacent with rear pointer follows front pointer that is shown in fig. 3.5(b).

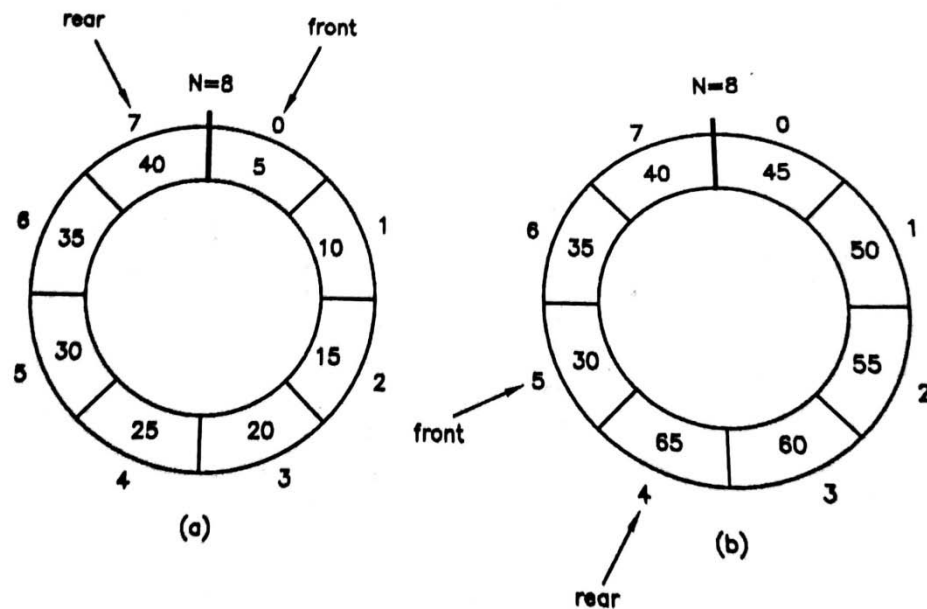


Fig. 3.5 : Overflow conditions in circular queue

3.6 Operation on circular Queues

The insert and delete are the operations performed on a circular queue. We will write the algorithms for both as C functions only with same assumptions that C array is used to implement the circular queue and front and rear pointers are implemented as integer variables whose addresses are passed to functions so that direct updating in original variables are possible.

Following are important steps for writing an algorithm for inserting a data item in a circular queue.

1. If **front** pointer is at first position and **rear** pointer is at last position, or **rear** and **front** pointers are adjacent i.e. **rear** pointer plus 1 is

front pointer, then we can insert into circular and return with message “overflow”.

2. Otherwise if **rear** pointer is at last position, place it to first position else increment it to move to next position.
3. Insert new data item.
4. If it was first insert (before insert **front = rear = -1**), then increment **front** also as new data item will be only item after insert, both front and rear should be same.

The C function to insert a data item into a circular queue is as follows.

```
/* cqinsert operation accepts following arguments
   q    : Array implementing queue
   front : Pointer to front end
   rear  : Pointer to rear end
   n     : Size of the queue
   val   : Data item to be inserted
*/
void cqinsert(int q[], int *front, int *rear, int n, int val)
{
    /* check for overflow */
    if((*front == 0 && *rear == n - 1) || (*rear + 1 == *front))
    {
        printf("overflow\n");
        return;
    }

    /* set rear pointer */
    if(*rear == n - 1)
        *rear = 0;
    else
        *rear = *rear + 1;

    /* Insert new data item */
    q[*rear] = val;

    /* Is first insert? Then set front pointer */
    if(*front == -1)
        *front = 0;

    /* return */
    return;
}
```

To insert a new data item in a circular queue, above function is called as

```
cqinsert(q,&front,&rear,N,data);
```

where **q** is the array implementing queue having front and rear pointers defined as integers **front** and **rear** whose addresses are passed to function, **N** denotes the size of the queue and **data** is the value to inserted in a queue.

Following are important steps to write an algorithm to delete a data item from a circular queue.

1. If **front** pointer contains -1, the circular queue is empty and return message "underflow."
2. Delete the item from front and store in temporary variable.
3. If it was only data item (**front = rear**), set **front = rear = -1** and return deleted item.
4. Otherwise if **front** is at last position, set it to first position, else increment **front** to point to next position and return deleted item.

The C function to delete a data item form a circular queue is as follows. Assume that queue stores only positive integers and returns -1 when there is an underflow.

```
/* cqdelete operation accepts following arguments
   q      : Array implementing queue
   front  : Pointer to front end
   rear   : Pointer to rear end
   n      : Size of the queue
and returns
   ineteger number (data value or -1)
*/
int cqdelete(int q[], int *front, int *rear, int n)
{
    int data;

    /* check for underflow */
    if(*front == -1) {
        printf("underflow\n");
        return(-1);
    }
}
```

```

    }

    /* delete a data item */
    data = q[*front];

    /* was it only item in queue? */
    if(*front == *rear) {
        *front = *rear = -1;
        return(data);
    }

    /* set front pointer? */
    if(*front == n - 1) {
        *front = 0;
    }
    else
        *front = *front + 1;

    /* return */
    return(data);
}

```

To delete a data item from a circular queue, above function is called as

```
data = cqdelete(q,&front,&rear,N);
```

where **q** is the array of size **N** implementing circular queue having front and rear pointers defined as integers **front** and **rear** whose addresses are passed to function. Variable **data** receives data item deleted or -1 if queue is empty.

Program 2 uses above functions to write a menu driven for circular queue of positive integers. It also implements function to display the whole contents of queue.

Program 2 :

Write a menu driven program to perform insert and delete operations on a circular queue of positive integers.

```

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <process.h>

```

```
#define N 4
```

```

void cqinsert(int [],int *,int *,int,int);
int cqdelete(int [],int *,int *,int);
void display(int [],int,int,int);

void main()
{
    int q[N];      /* queue of size N */
    int front = -1, rear = -1;    /* front and rear pointers */
    int data,choise;

    while(1) {
        clrscr();
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choise : ");
        scanf("%d",&choise);

        switch(choise) {
            case 1: printf("Enter a value to insert : ");
                    scanf("%d",&data);
                    cqinsert(q,&front,&rear,N,data);
                    break;
            case 2: data = cqdelete(q,&front,&rear,N);
                    if(data != -1)
                        printf("Deleted %d\n",data);
                    break;
            case 3: display(q,front,rear,N);
                    break;
            default: exit(0);
        }

        getch();
    }
}

/* Function to insert a data item in a circular queue */
void cqinsert(int q[], int *front, int *rear, int n, int val)
{
    /* check for overflow */
    if((*front == 0 && *rear == n - 1) || ((*rear + 1) == *front))
    {
        printf("overflow\n");
        return;
    }

    /* set rear pointer */
    if(*rear == n - 1)
        *rear = 0;
}

```

```

else
    *rear = *rear + 1;

/* Insert new data item */
q[*rear] = val;

/* Is first insert? Then set front pointer */
if(*front == -1)
    *front = 0;

/* return */
return;
}

/* Function to delete a data item in a circular queue */
int cqdelete(int q[], int *front, int *rear, int n)
{
    int data;

    /* check for underflow */
    if(*front == -1) {
        printf("underflow\n");
        return(-1);
    }

    /* delete a data item */
    data = q[*front];

    /* was it only item in queue? */
    if(*front == *rear) {
        *front = *rear = -1;
        return(data);
    }

    /* set front pointer? */
    if(*front == n - 1)
        *front = 0;
    else
        *front = *front + 1;

    /* return */
    return(data);
}

/* Function to display contents of queue */
void display(int q[], int front, int rear, int n)
{
    int i;

```

```

if(front == -1)
    printf("\n\t\tQueue Empty\n");
else {
    if(front <= rear) {
        printf("\n\t\tFront( = %d) -->",front);
        while(front <= rear) {
            printf("%d ",q[front]);
            front++;
        }
        printf("<-- Rear(= %d)\n",rear);
    }
    else {
        printf("\n\t\tPos 0 -->");
        i = 0;
        while(i<=rear) {
            printf("%d ",q[i]);
            i++;
        }
        printf("<-- Rear(= %d)",rear);
        if((rear + 1) != front)
            printf(" - - ");
        i = front;
        printf("Front ( = %d)-->",front);
        while(i<=n - 1) {
            printf(" %d ",q[i]);
            i++;
        }
        printf("<-- Pos N-1\n");
    }
}
}
}

```

Above implementations of the linear and circular both were using sequential allocation. We can also implement them with linked allocation. Linked allocation is discussed in next unit.

3.7 PRIORITY QUEUE

A priority queue is a linear queue in which insertion and deletion are made at any position depending on the priority of the data item. These types of queues are normally used by operating systems to implement a queue of ready processes with different level of priorities. Whenever processor completes current process, new process is selected for execution from the ready queue based on priority. Consider a priority queue denoting ready processes with priorities 1,2 and 3 where 1 is highest

priority and 3 is lowest priority. It is shown in fig. 3.6. Process to execute next is selected from queue with priority 1 processes first. Process with priority 2 is selected only if there is no process with priority 1 is waiting in queue. Similarly, processes with priority 3 are selected only if there is no process in queue with priority 1 or 2. The queue also follows the FIFO order within priority level so that whenever new ready process is created, it joins the queue after the last process with same priority and before the first process with next lower priority.

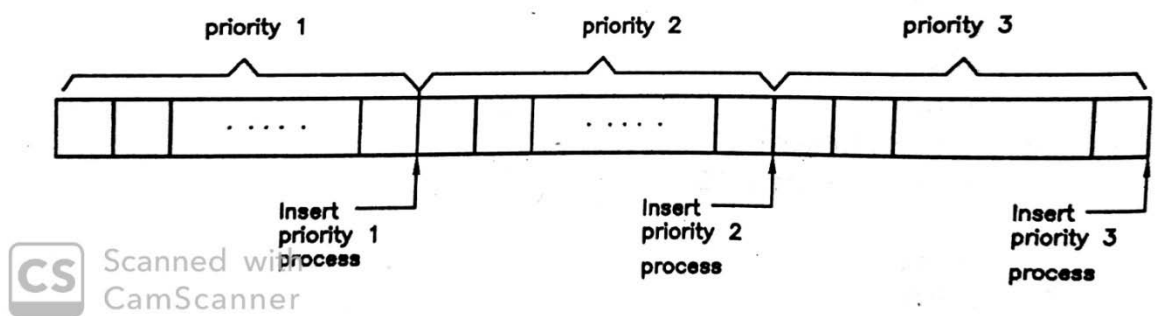


Fig. 3.6 : Priority queue with three different priority levels.

The above implementation uses a single queue to implement multiple queues with different priorities. The problem with this approach is that the insertion operation is done in middle of the queue i.e. after last process with same priority and before the first process with next lower priority. Inserting process in queue in middle is difficult operation. To solve this problem, we can use multiple queues rather than one queue, separate for each priority level as shown in fig. 3.7.

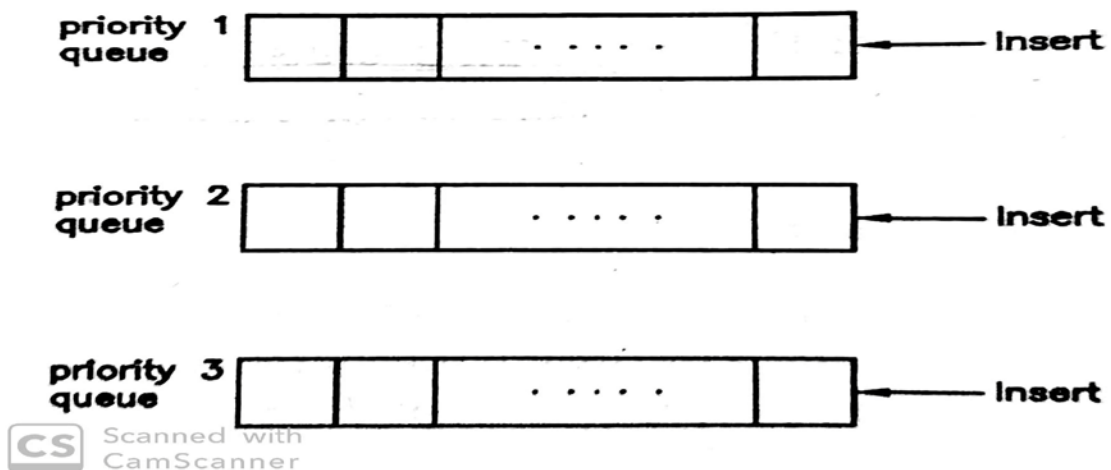


Fig. 3.7 : Priority queue using multiple queue

In this approach, whenever new process is ready to join (to be inserted), it directly joins at the end of queue whose priority is same as the priority of processes to join. Whenever process is to be selected for execution (to be deleted), first the all processes with priority 1 are selected in FIFO manner, then with priority 2 if there is no processes in queue for priority 1 and so on.

3.8LET US SUM UP

Queue : A queue is a linear data structure which follows FIFO order and allows operation at both the ends. Insertion is performed at rear end and deletion is performed at front end.

Front : Pointer to front end of the queue where delete operation is performed.

Rear : Pointer to rear end of the queue where insert operation is performed.

Circular queue : In circular queue, both the ends of the queue are joined together so that once the pointer, either front or rear reaches to the end of the queue, they can move to the start of the queue.

Priority queue : A priority queue is a linear queue in which insertion and deletion are made at any position depending on the priority of the data item.

3.9CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. Insertion operation is performed at end in queue.
2. When rear pointer points to position N (last) in circular queue, next insertion is performed at position
3. data structure allows operation only at one end, while data structure allows operations at both the end.
4. If front and rear end points to same position, there are at maximum elements present in linear queue.
5. does not allow reuse of space until it gets empty.

➤ **True-False**

1. Stack is linear data structure while queue is non-linear data structure.
2. Circular queue does not follow LIFO order.
3. Overflow never occurs in circular queue as it allows reuse of space.
4. Pressing keys on keyboard and simultaneously displaying them on display is an example of order.
5. Overflow in circular queue is detected by testing $\text{rear} = N - 1$ and $\text{front} = 0$ only.

➤ **Answer in brief**

1. What is major difference between stack and queue?
2. What is major drawback of linear queue? How can you overcome it?
3. Give the advantage of circular queue over linear queue.
4. Name at three examples of linear queue.
5. Give at least three example of simple queue.
6. Why does queue need two pointers?
7. Give the application of priority queue.

➤ **Answer the following**

1. Give the representation of queue and explain the operations performed on it.
2. Give the algorithm in C for insert into a linear queue.
3. Give the steps to delete an element from linear queue.
4. Explain the concept of circular queue and give the steps to delete a data item from it.
5. Give the steps to insert an element into a circular queue.
6. Give the algorithm for circular queue delete.
7. Write short notes
 1. Queue
 2. Insert operation in linear queue
 3. Circular queue
 4. Delete operation in circular queue
 5. Priority queue

3.10 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

3.11 ASSIGNMENTS

1. Develop a C program which implements a queue of student information like roll number, name and marks. New record is inserted at the end while records are taken from front to process them.
2. Develop a C program to implement the priority queue with 5 different priority levels.

Unit 4: Linked List

4

Unit Structure

- 4.1 Learning Objectives
- 4.2 Introduction
- 4.3 Singly Linked Lists
- 4.4 Operations on Singly Linked Lists
- 4.5 Circular Lists
- 4.6 Doubly Linked Lists
- 4.7 Operations on Doubly Linked Lists
- 4.8 Let us sum up
- 4.9 Check your Progress: Possible Answers
- 4.10 Further Reading
- 4.11 Assignment

4.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define a singly linked list with its node structure.
- State the merits and demerits of the singly linked list.
- List and perform the various operations on singly linked list.
- Define the circular list and perform operations on it.
- Define a doubly linked list with its node structure.
- List and perform the various operations on doubly linked list.

4.2 INTRODUCTION

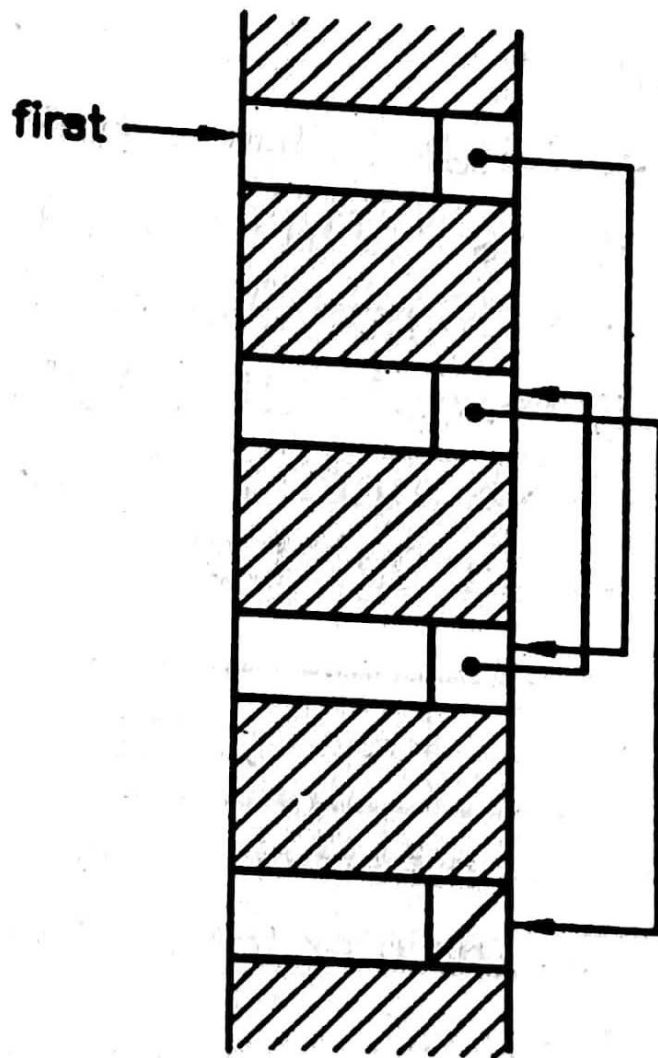
We have studied three linear data structures arrays, stacks and queues in previous units. Fourth and very important linear data structure is linked list. In unit 2 and 3 , we have used static and sequential memory allocation using arrays to implement stacks and queues. This approach is suitable for fixed size data but is not suitable for applications in which data size is varying with time and frequent insert and delete operations are required. These types of applications are dynamic in nature. The major issue in such situation is memory allocation. Allocating fixed amount of memory using arrays will cause program to fail when size of data exceeds the size allocated during the execution or allocation of larger memory for safety wastes the memory. The solution is to use the dynamic memory allocation to allocate the memory as per the need during the execution using linked allocation. The best data structure in such situation is linked list. Telephone directory is one such example, where the list of telephone numbers and the owners can be maintained as linked list and subject to frequent change as new numbers are added in list and some numbers are deleted as the connection is cancelled.

This unit covers the singly linked list, circular list and doubly linked list with basic concepts, representation and operations.

4.3 SINGLY LINKED LIST

Array is an example of static memory allocation as memory required is allocated at the compile time. The major problem in sequential allocation is insertion and deletion. How can we insert a new element between first and second element? One of the ways is to shift down all the elements from second to last, make the space between first and second and then insert. Similarly, to delete an element, all the elements below it are to be shifted up. The process of shifting elements down or up is time consuming and makes program slow. This is apparent in large arrays.

The linked list solves all the above problems, i.e. size as well as insert and delete. Linked list uses the dynamic memory allocation and hence the more memory can be allocated as and when needed as well as unused memory can be returned back once the elements are deleted. The major advantage comes from the non-sequential allocation of memory i.e. not necessarily continuous block of memory. This results into optimal use of the available memory. But, as elements are not represented sequentially, how do we know where the next element is? Each element in linked list must store the address of next element to know where it is. It is shown in fig. 4.1. Knowing the address of the first element in list, we can access all the elements, as each element knows address of next element. Last element stores null in address part to denote the end of the list. These concepts will be clearer in subsequent sections.



Scanned with
CamScanner

Fig. 4.1 : Non-sequential memory allocation for linked list

The Linked list of fig. 4.1 is normally represented as shown in fig. 4.2. It is known as *singly linked list* as each node contains only address of the next node called successor.

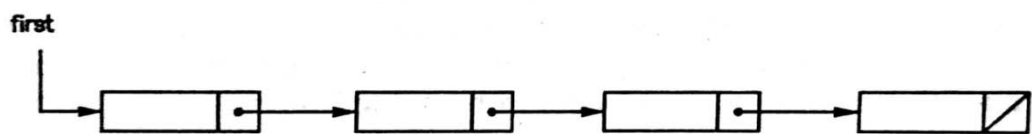


Fig. 4.2 : Singly linked list

The **first** is an address of the first element in the list. If **first** is NULL, the list contains no elements and called empty list. Fig. 4.3 shows the some examples of singly linked lists.

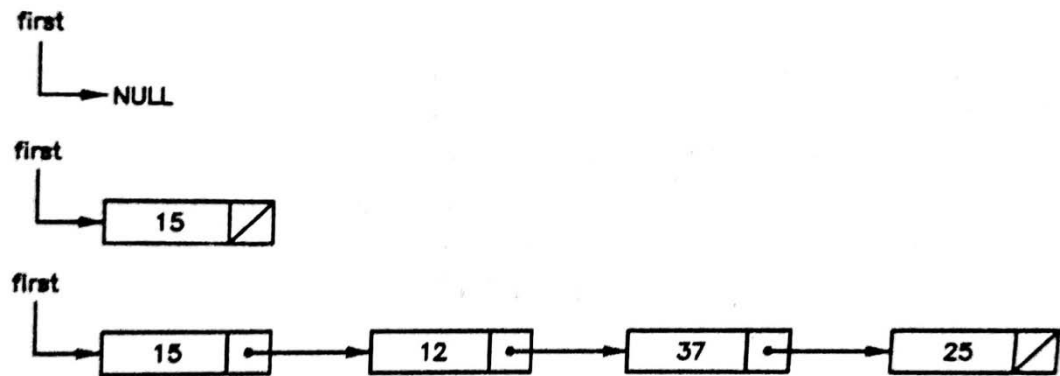
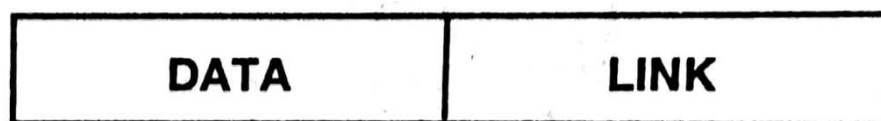


Fig. 4.3 : Examples of singly linked list

Each node in the singly linked list follows the node structure as shown in fig. 4.4. The node contains two fields : DATA and LINK. DATA represents the information associated with the node. It may be single value like an integer, float etc. or composite value like student record. LINK field contains the address of the next node (successor) in the list. LINK field is NULL for last node as it has no successor (see last node in examples in fig. 4.3).



DATA : Information associated with node

LINK : Address (pointer to) next node

Fig. 4.4 : Node structure of singly linked list

To work with the linked list using programming language like C, first we need to define the node structure and then using operations like insert and delete, list is manipulated. C structure is used to define the node. For the singly linked list representing the integer numbers, the node structure is defined as follows.

```
struct SNODE {
    int data;          /* DATA field representing integer */
};
```



```

        struct SNODE *next;    /* LINK field representing link to next node */
};

```

To represent the student list, node structure is defined as follows.

```

struct STUDENT {
    int    roll_no;
    char   name[20];
    int    marks;
};

struct SNODE {
    struct STUDENT data; /* DATA field representing a student info. */
    struct SNODE *next; /* LINK field representing link to next node */
};

```

Observe that DATA field itself consists of three values **roll_no**, **name** and **marks** packed in structure called STUDENT. It is also possible to represent above node structure directly without using structure for STUDENT as follows.

```

struct SNODE {
    int    roll_no;          /* DATA field1 – roll number */
    char   name[20];        /* DATA field2 – name of student */
    int    marks;           /* DATA field3 – marks of student */
    struct SNODE *next;     /* LINK field representing link to next node */
};

```

4.4 OPERATIONS ON SINGLY LINKED LIST

The insertion and deletion are most frequent operations performed on linked lists as new values are required to be inserted in linked list and values, which are no longer required, are deleted from the linked list. A new node can be inserted in singly linked list either at front, end or in middle. Same is the case for deletion. However, when we need to develop an algorithm to perform insertion or deletion our criteria for insertion or deletion must be known.

We will use following node structure through out this section for singly linked list.

```
typedef struct SNODE {
    int data;
    struct SNODE *next;
} SNODE;
```

Insertion into and deletion from front:

Let us first discuss how to insert a node in front of the singly linked list i.e as first node. Assume that the node to be inserted is pointed by pointer **new**. There are two possibilities:

1. Insertion in empty list. List has no nodes before insertion.
2. Insertion in non-empty list. List already contains one or more nodes.

In first case, the new node becomes only the node after insertion and in second case, the new node will become the first node of updated list. This is shown in fig. 4.5 where a pointer **first** point to the existing linked list in which node is to be inserted and pointer **new** points the node to be inserted.

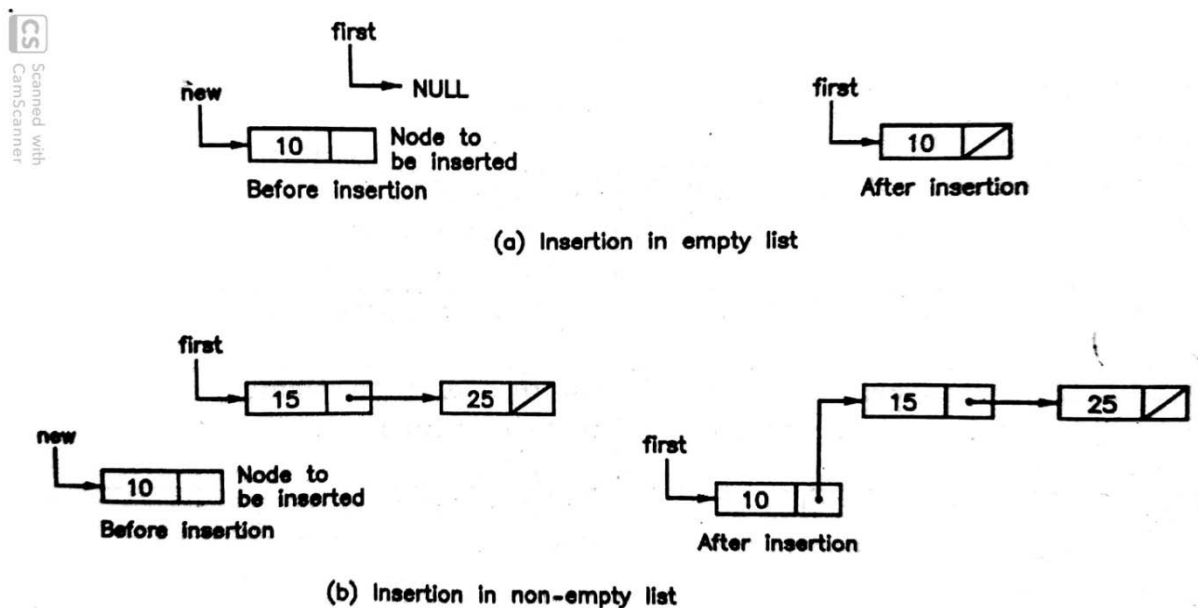


Fig. 4.5 : Insertion in front of singly linked list

In both cases, **first** pointer is copied into the LINK part of the new node pointed by **new** pointer. Observe that in first case, the existing list is empty (**first** pointer is NULL) which is copied in LINK field (pointer **next**) of the new node. As after insertion, new node will be the only node in list, its LINK field is to be NULL. In second case, copying pointer to first node of existing list i.e. pointer **first** in LINK field of the new node joins the existing list behind the new node and new node becomes first node. After insertion in both cases, new node will become the first node and hence pointer **new** becomes the updated **first** pointer.

Considering above discussion, following are important steps to write an algorithm to insert a new node in front of singly linked list.

1. Allocate memory for new node dynamically.
2. Copy the data to be inserted in DATA field of new node.
3. Join the existing list by copying the address of first node into the LINK field of the new node.
4. Return the address of new node as updated address of list.

The C function to insert the node in front of the singly linked list is written as follows. The linked list consists of integer values and hence the node structure described in previous section with structure SNODE is used.

```

/* Insert(front) operation accepts following arguments
   first   : Pointer to first node of existing list
   val     : Data to be inserted
and returns
   new     : Pointer to first node after insertion
*/
NODE *insert_front(NODE *first, int val)
{
    NODE *new;

    /* Allocate memory for new node */
    new = (struct SNODE *)malloc(sizeof(struct SNODE));

    /* Initialize data part of new node by value val */
    new->data = val;

    /* Insert the node in front */
    new->next = first;

```

```

    /* return the address of updated list */
    return(new);
}

```

To insert a node, above function is called as

```

first = insert_front(first,val);

```

where the first argument **first** is the pointer to the first node of existing list and second argument **val** is the value to be inserted. After performing an insertion operation, function returns the address of the newly inserted node (i.e. pointer **new**) which is copied into the **first** again and hence, after the above statement, the first points to the updated list with newly inserted node is first node.

Let us now understand how to delete a first node (deletion from front) of the singly linked list. There are three possibilities:

1. Deleting from an empty list. List has no nodes before deletion.
2. Deleting from list having only one node.
3. Deleting from list having more than one node.

In case 1, no action is required. Case 2 deletes only the node in list and makes the list empty. In case 3, after deleting first node, second node becomes the first node. These are shown in fig. 4.6.

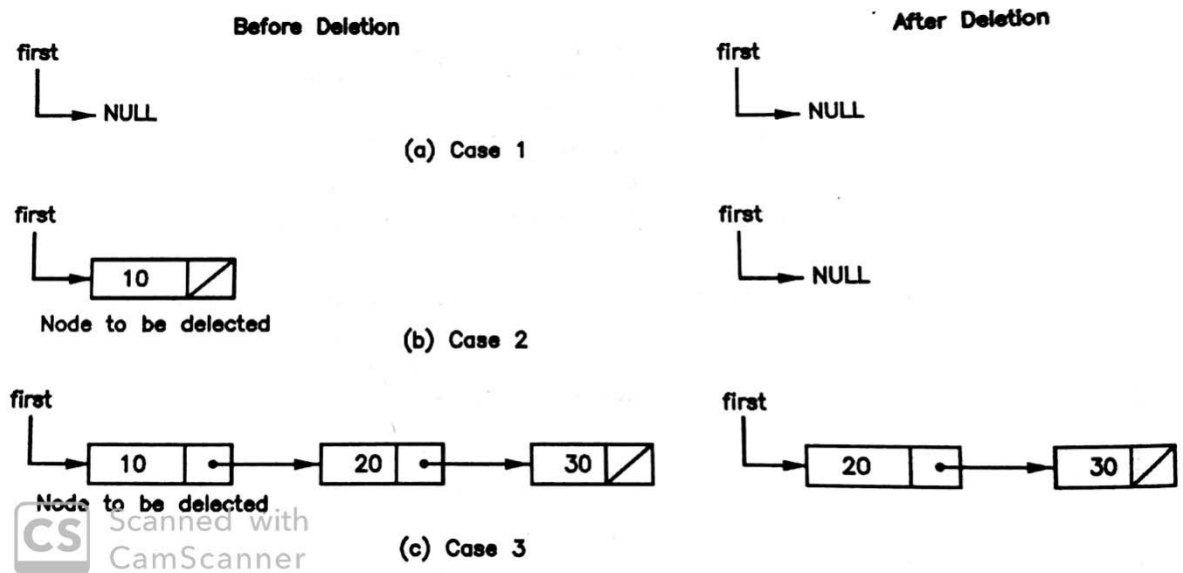


Fig. 4.6 : Deletion from front of singly lined list

Last two cases can be combined together in implementation. As in case 2, after deletion list becomes empty and updated value for pointer **first** is NULL (which is LINK field of node pointed by **first** before deletion) and in case 3, the updated value for pointer **first** is the address of second node, which is also the LINK field of the node pointed by **first** before deletion).

Following are important steps to delete n node form the front of the singly linked list.

1. If list is empty, we can not delete the node and return with null.
2. Otherwise, store the address of first node (pointer first) into temporary pointer.
3. Delete the first node by copying LINK of **first** to **first**.
4. Remove the deleted node from memory using temporary pointer.
5. Return the updated address of linked list.

The C function to delete a node from front of singly linked list is shown below.

```

/* Delete(front) operation accepts following arguments
   first   : Pointer to first node of existing list
and returns
   Pointer to node : Pointer to first node after deletion
*/
NODE *delete_front(NODE *first)
{
    NODE *temp;

    /* Is list empty? */
    if (first == NULL) {
        printf("List is Empty\n");
        return(NULL);
    }

    /* If list is not empty, store first in temp */
    temp = first;

    /* update first pointer */
    first = first->next;

    /* remove first node */
    free(temp);

    /* return the address of updated list */
    return(first);
}

```

```
}
```

To delete a node from front, above function is called as

```
first = delete_front(first);
```

where argument **first** demotes the pointer to first node before deletion and address returned by function is stored again in **first** to point to the updated list.

Following program uses above two functions to write a menu driven program to insert and delete a node in front of the singly linked list. It also provides a function for displaying contents of list.

Program 1 :

Write a menu driven program to insert and delete node from the front of a singly linked list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <process.h>

typedef struct SNODE {
    int data;
    struct SNODE *next;
} NODE;

NODE *insert_front(NODE *, int);
NODE *delete_front(NODE *);
void display(NODE *);

void main()
{
    NODE *first = NULL;
    int val,choise;

    while(1) {
        clrscr();
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choise : ");
```

```

scanf("%d",&choise);

switch(choise) {
    case 1: printf("Enter a value to insert : ");
            scanf("%d",&val);
            first = insert_front(first,val);
            break;
    case 2: first = delete_front(first);
            break;
    case 3: display(first);
            break;
    default: exit(0);
}

getch();
}
}

```

```

/* Function to insert node in front of list */
NODE *insert_front(NODE *first,int val)
{
    NODE *new;

    /* Allocate memory for new node */
    new = (NODE *)malloc(sizeof(NODE));

    /* Initialize data part of new node by value val */
    new->data = val;

    /* Insert the node in front */
    new->next = first;

    /* return the address of updated list */
    return(new);
}

```

```

/* Function to delete node from front of list */
NODE *delete_front(NODE *first)
{
    NODE *temp;

    /* Is list empty? */
    if (first == NULL) {
        printf("List is Empty\n");
        return(NULL);
    }

    /* List is not empty */
    temp = first; /* store first in temp */

```

```

        /* update first pointer */
        first = first->next;

        /* remove first node */
        free(temp);

        /* return the address of updated list */
        return(first);
    }

/* Function to display entire list */
void display(NODE *first)
{
    /* is list empty? */
    if(first == NULL)
        printf("List is Empty\n");

    /* Display contents of list */
    while(first != NULL) {
        printf("%d ",first->data); /* Print data */
        first = first->next;      /* go to next node */
    }
    printf("\n");
}

```

Insertion into and deletion from end :

Previous example inserts a node or deletes a node from the front of the singly linked list. Same way insertion and deletion can be performed at the end of the list also. To insert or delete a node at the end of the list is difficult as compared to front, because to perform the operation at the end of a list needs traversal from first node to last node in order to reach the end of the list. Inserting new node at the end of the list require to copy address of the new node i.e pointer **new** in the LINK field of the last node as shown in fig. 4.7. But to access the LINK field of the last node we need address of last node which is stored in LINK field of the previous node. Same argument applies to previous node and finally we have to start from the first node to reach the last node (node with LINK field NULL). Once the last node is reached (node with pointer **cur** in fig. 3.9), insertion operation is performed by just copying pointer **new** into the LINK field.

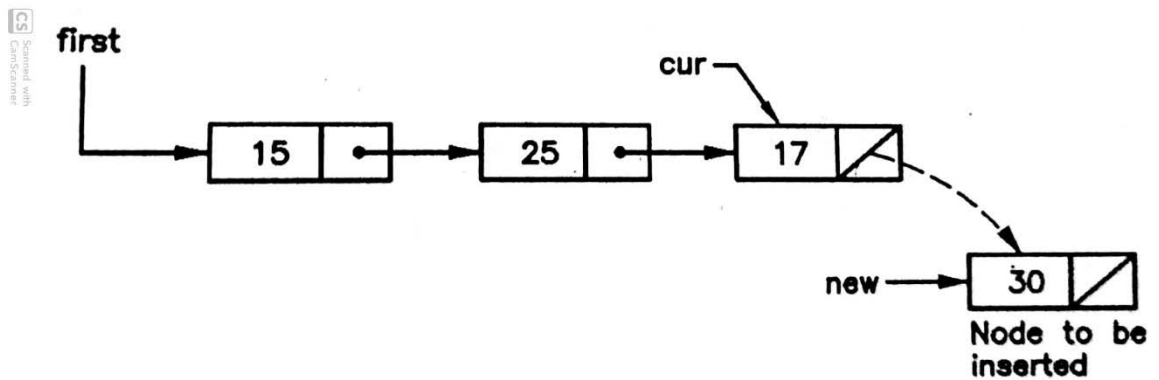


Fig. 4.7 : Insertion at the end of singly linked list

Given an address of the first node (pointer **first**), following code reaches the end of the list.

```
cur = first;
while(cur->next != NULL)
cur = cur->next;
cur->next = new;
```

First statement initialize pointer **cur** with pointer **first**. Each time in loop, if the LINK field of node pointed by pointer **cur** is not null, the **cur** is updated by the address of next node i.e. **cur** moves to the next node. When finally **cur** reaches to last node, LINK field is NULL and loop terminates. The next operation we can perform is insertion by copying address of new node (pointer **new**) in the LINK field of **cur** node.

Following are important steps to write an algorithm to insert a new node at the end of the list.

1. Allocate a memory dynamically for new node
2. Initialize new node by copying value to be inserted in DATA field and NULL in the link field.
3. If the list is empty, then new node becomes only node after insertion and returns its address.
4. Otherwise, initialize current pointer by first and forward by updating it by its own LINK field until last node is encountered.
5. Insert new node by copying its address in link field of current node.
6. Return address of first node.

The C function to insert a new node at the end of the singly linked list is as follows.

```

/* Insert(end) operation accepts following arguments
   first   : Pointer to first node of existing list
   val     : Data to be inserted
and returns
   Pointer to node : Pointer to first node after insertion
*/
NODE *insert_end(struct SNODE *first, int val)
{
    NODE *new,*cur;

    /* Allocate memory for new node */
    new = (struct SNODE *)malloc(sizeof(struct SNODE));

    /* Initialize new node */
    new->data = val;
    new->next = NULL;

    /* Is list empty? */
    if(first == NULL)
        return(new);

    /* Reach to the end of the list, if list is not empty */
    cur = first;
    while(cur->next != NULL)
        cur = cur->next;

    /* Insert the node */
    cur->next = new;

    /* return the address of updated list */
    return(first);
}

```

Deleting last node needs the address of not only last node, but last and its previous as deleting last node needs to store NULL in the LINK field of previous node as shown in fig. 4.8. To locate last node and previous node, we have use the same concept used in insertion to traverse from first node to last node. This time we will use two pointers **cur** and **pred** (current and predecessor) where **pred** follows the **cur**. When **cur** reaches to last node, **pred** points to second last node. Then deletion operation is performed by putting NULL in LINK field of node pointed by **pred**.

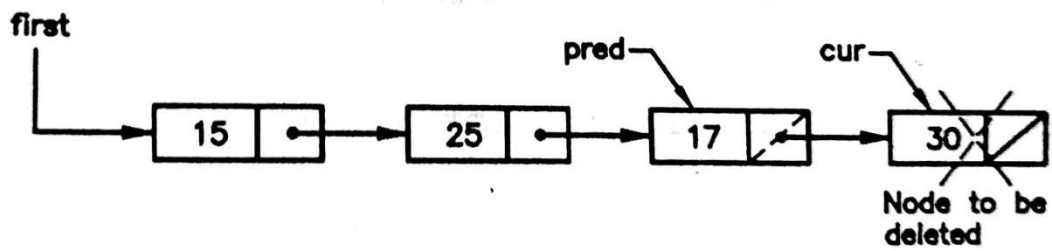


Fig. 4.8 : Deletion at the end of singly linked list

The code is given as (assuming there are minimum two nodes in list)

```

prev = NULL;
save = first;
while(save->next != NULL) {
    prev = save;
    save = save->next;
}
prev->next = NULL;

```

Following are important steps to delete a last node from the singly linked list.

1. If list is empty, node can not be deleted and return with null.
2. If there is only one node in list (LINK field of first is NULL), then delete the node and return with null as list becomes empty now.
3. Initialize predecessor with null and current with first and update current by its LINK field and predecessor by current until last node is encountered.
4. Remove last node from list by copying NULL in LINK of predecessor and delete current node (last node) from memory.
5. Return address of first node.

The C function to delete a node from end of the singly linked list is as follows.

```

/* Delete(end) operation accepts following arguments
first   : Pointer to first node of existing list
and returns
          Pointer to node : Pointer to first node after deletion
*/
NODE *delete_end(NODE *first)
{
    NODE *pred,*cur;

```

```

/* Is list empty? */
if (first == NULL) {
printf("List is Empty\n");
return(NULL);
}

/* only one node in list */
if(first->next == NULL) {
free(first);
return(NULL);
}

/* List with more than one node */
pred = NULL;
cur = first;
while(cur->next != NULL) {
pred = cur;
cur = cur->next;
}
/* delete last node from list */
pred->next = NULL;
/* Release memory */
free(cur);

/* return */
return(first);
}

```

The insert and delete functions performing operations at end of the list are called in same fashion as the insert and delete function to perform operation in front of the list.

Insertion in middle:

The above examples perform operations either in front or at the end. It is also possible to design algorithms and implemented them in C where operations are performed in middle. Let us consider an example to insert a node after node with specified value. It is ensured that node with specified value is available in the list and list is not empty. For example, in fig. 4.9, new node is to be inserted after node with value 24. This requires that address of the node with value 24 is known i.e. pointer **cur**. To complete the operation, first the address of node after **cur** node is copied in the LINK field of the new node and address of new node i.e. pointer **new** is copied into the LINK field of the node **cur**.

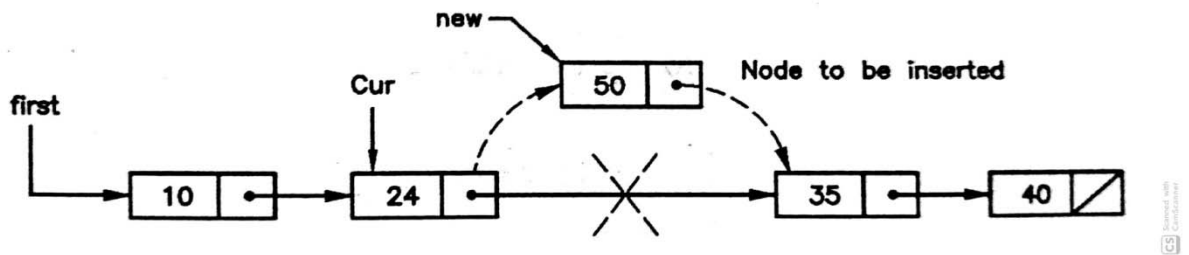


Fig. 4.9 : Inserting node in middle of singly linked list

Following steps are required to perform the insertion of new node after the node with certain value in a singly linked list.

1. Allocate memory dynamically for new node.
2. Initialize new node by copying value to be inserted in the DATA field.
3. Search for the node with given value. To find its address initialize current node with **first** pointer, compare DATA field of current with given value and forward current pointer to next node in list if they are not equal.
4. Insert new node by copying LINK field of current into LINK field of new node and address of new node into LINK field of current.

The C function for this insertion is as follows. Remember our assumption that list is not empty and having at least one node with value equal to specified value.

```

/* Insert(middle) accepts the following arguments
   first   : Pointer to first node
   sp_val  : Value of a node after which the new node is to be inserted
   val     : Value of new node to be inserted
*/
void insert_mid(NODE *first, int sp_val, int val)
{
    NODE *new;

    /* Allocate memory */
    new = (NODE *)malloc(sizeof(NODE *));

    /* Initialize node */
    new->data = val;

    /* search for the node with specified value */
    cur = first;
    while(cur->data != sp_val)

```

```

        cur = cur->next;

/* Insert the node */
new->next = cur->next;
cur->next = new;
}

```

If this function is called as,

```
insert_mid(first, 20,45);
```

then it inserts a new node with value 45 after the node having value 20 in its DATA field in a singly linked list pointer by pointer **first**.

4.5 CIRCULAR LISTS

Each node in singly linked list stores address of next node in its LINK field except last node. Last node contains NULL pointer in its LINK field to denote that it is last node. If NULL pointer in LINK field is replaced by address of first node, then singly linked list is known as *circular linked list* or *circular list* as shown in fig. 4.10.

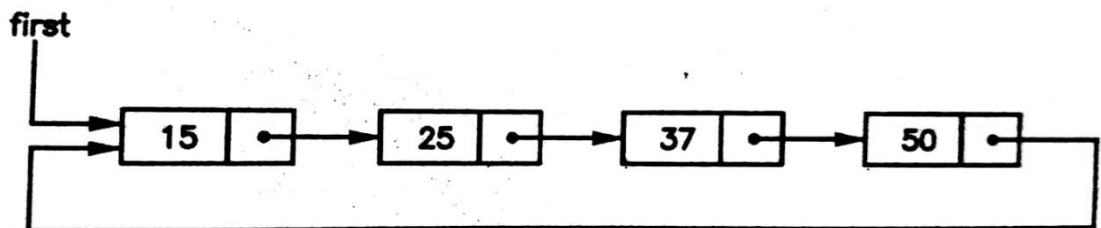


Fig. 4.10 : Circular linked list

Circular linked list allows us to move from last node to first node and hence we can circle around list. The advantage is that we can start traversing the list from any given node not only from first node. The disadvantage is that if proper care is not taken to identify when circle is over, there are chances that one will in infinite loop. To solve this problem, a special node called *Head node* is used which always act, as head of the list as well as list will never be empty as at least head node is always present in the list. The head node does not store any user data and hence the DATA field of head node is always shown with shadow and LINK part points to head itself

or points first user node. If LINK field of the head node contains address of head node, the list is considered as empty. Fig. 4.11 shows the structure of circular list with head node.

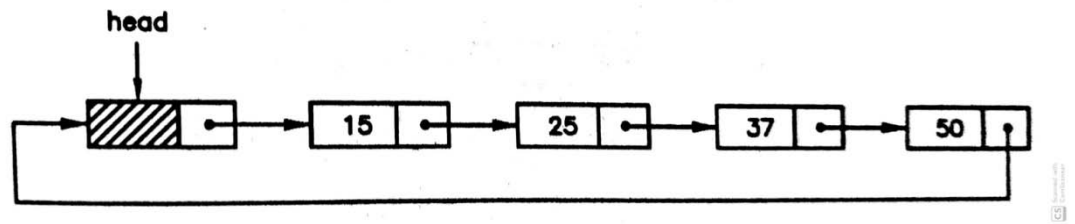
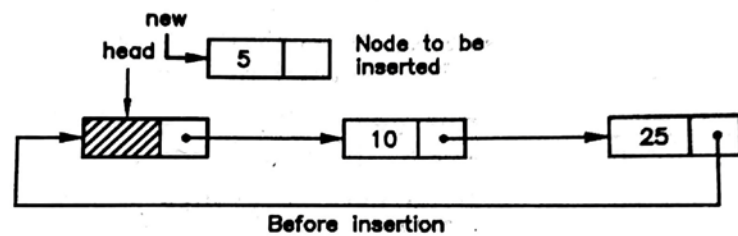
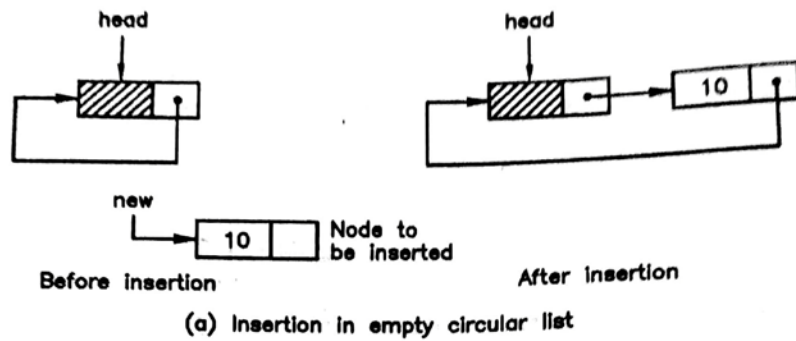


Fig. 4.11 : Circular list with head node

We can perform all the operations on circular list that we performed on singly linked list with consideration that last node should always point to head node after operation also. Let us discuss algorithms to insert and delete the nodes in front i.e. immediately after head node in circular list.

To insert a node, if list is empty i.e. LINK of head is pointing to head, the new node will join after head node and link of new node points to head. If list is not empty, new node will be inserted between head and existing first node by updating pointer. Fig. 4.12 shows both the cases before and after insertion. Observe that insertion in both cases is accomplished by copying LINK field of head into LINK field of **new** node and address **new** into LINK of head node.



CS Scanned with CamScanner

Fig. 4.12 : Insertion in circular list as first node

The steps for insertion algorithm are as follows.

1. Allocate memory dynamically for new node.
2. Initialize DATA field of new node by given value.
3. Insert new node by copying LINK of head into LINK of **new** and **new** into LINK of head.

The C code for insert algorithm is as follows.

```

/* Insert operation accepts following arguments
   head : Pointer to head node
   val   : Value to be inserted.
*/
void cir_insert(NODE *head,int val)
{
    NODE *new;

```



```

/* Allocate memory for new node */
new = (NODE *)malloc(sizeof(NODE));

/* Initialize new node */
new->data = val;

/* Insert new node */
new->next = head->next;
head->next = new;

/* return */
return;
}

```

To insert a node, above function is called as

```

cir_insert(head,val);

```

where **head** is pointer to head node of circular list and **val** is data to be inserted.

Let us now see how we can delete a first node from the circular list. If list is empty, then deletion can not be performed. Otherwise, to delete a node, simply copy the LINK field of first node to the LINK field of head node. If list has only one node, then that node contains address of head in its link as it is last node. Deletion will copy it into LINK field of head node which makes the list empty. In second case, the second node will become first node after head node. These are shown in fig. 4.13.

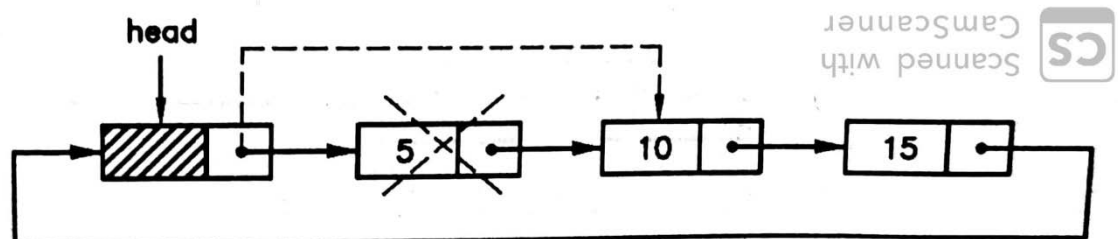


Fig. 4.13 : Deleting first node from circular list

The steps for deletion algorithm are as follows.

1. If list is empty, then we can not delete the node and return.
2. Otherwise, copy the address of first node into temporary pointer.
3. Remove the node by copying LINK of first node to LINK of head node.
4. Delete node from memory using temporary pointer.

The C function to delete a node from circular list is as follows.

```
/* Delete operation accepts following arguments
   head  : Pointer to head node
*/
void cir_delete(NODE *head)
{
    NODE *temp;

    /* Is list empty? */
    if(head->next == head) {
        printf("List is Empty\n");
        return;
    }

    /* get address of first node */
    temp = head->next;

    /* Remove the node from list */
    head->next = temp->next;

    /* delete node from memory */
    free(temp);

    /* return */
    return;
}
```

To delete a node, it is called as

```
cir_delete(head);
```

where **head** is the pointer to the head node of the circular list.

You can write complete program demonstrating the insert and delete operations as we did in case of singly linked list. However, when program starts we need to create empty circular list as below before we start performing operations.

```
NODE *head;
head = (NODE *)malloc(sizeof(NODE));
head->next = next;
```

We can also similar way develop the algorithms to insert and delete the nodes from end of the circular list. It is also possible to develop insert and deletion algorithms based on other criteria like singly linked list. These are left as exercise for the readers.

4.6 DOUBLY LINKED LIST

Major limitation of singly linked list is that it allows traversal in only one direction i.e. from first to last but not in reverse direction. You cannot go to previous node from the current node. Another problem with singly linked list is that delete operation is costly, as it needs address of previous node, which can be found only by traversing from first node to desired node. This limitations are overcome by using two pointers at each node : one pointing to previous node and another pointing to next node. If each node in list uses two pointers in linked list, then it is called *doubly linked list*. Fig. 4.14 shows the node structure for the doubly linked list.



LPTR : Pointer to previous node (predecessor)

DATA : Information associated with node

RPTR : Pointer to next node (successor)



Scanned with
CamScanner

Fig. 4.14 : Node structure of doubly linked list

The DATA part stores the information associated with node while LPTR and RPTR are pointers to predecessor and successor nodes respectively. In order to keep track of both the ends, two special pointers **L** and **R** are used where **L** is pointer to left most node and **R** is the pointer to right most node. Fig. 3.18 shows the various examples of doubly linked list. Fig. 4.15(a) shows the empty doubly linked list as **L = R = NULL**. When ever doubly linked list contains only one node both **L** and **R**

points to same node i.e. $L = R$ which is shown in fig. 4.15(b). A list with more nodes is shown in fig. 3.18(c).

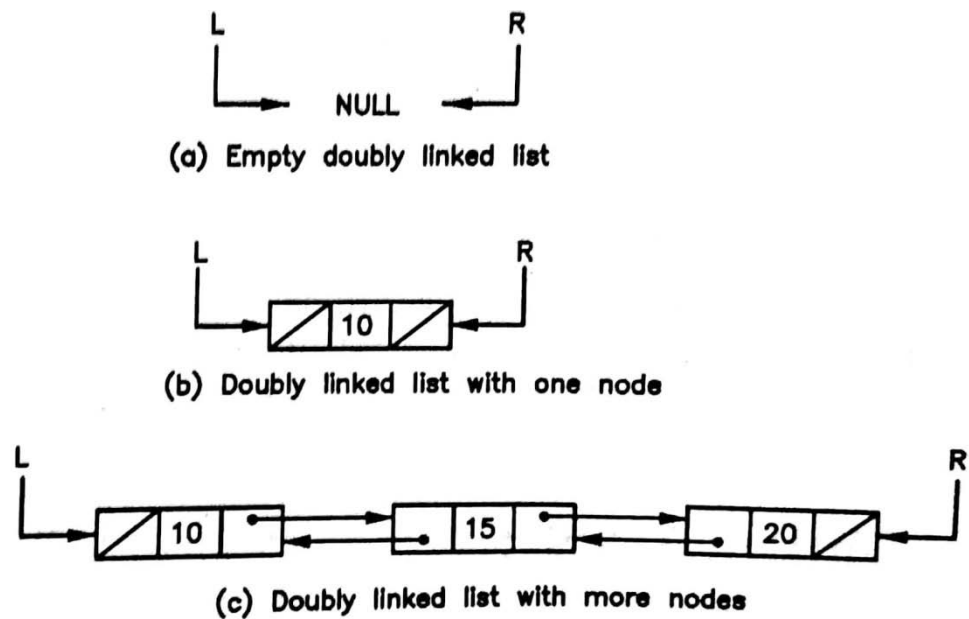


Fig. 4.15 : Examples of doubly linked list

Doubly linked list allows traversal in both the direction because 1) each node contains pointer to its predecessor as well as successor 2) Pointer to left most node L enables us to traverse from left to right and 3) Pointer to right most node R enables us to traverse from right to left. In case of deletion, as each node points to predecessor and successor, without requiring any additional traversal, it can be performed. The only disadvantage of doubly linked list is it requires addition space to store two pointers.

To implement doubly linked list using C language, we need to first define the node structure as below

```
typedef struct D_NODE {
    struct D_NODE *lptr;    /* LPTR: Pointer to predecessor */
    int data;              /* DATA : information associated with node */
    struct D_NODE *rptr    /* RPTR: Pointer to successor */
} DNODE;
```

4.7 OPERATIONS ON DOUBLY LINKED LISTS

We can perform variety of operations on doubly linked list just like singly linked list. The most common operations are insertion and deletion. They are performed again with certain criteria. For example, to insert the new node as left most node or delete the left most node. Similarly, we can insert new node as right most node or delete the right mode node. We can also perform insertion and deletion based on other criteria. Apart from insertion and deletion, it is also possible to perform other operations. In this section, we will study the algorithms to insert at delete at left end.

Let us first consider operation to insert new node as left most node. There are two possibilities:

1. Insertion in empty list. List has no nodes before insertion.
2. Insertion in non-empty list. List already contains one or more node.

Fig. 4.16 shows both of above cases before and after insertion.

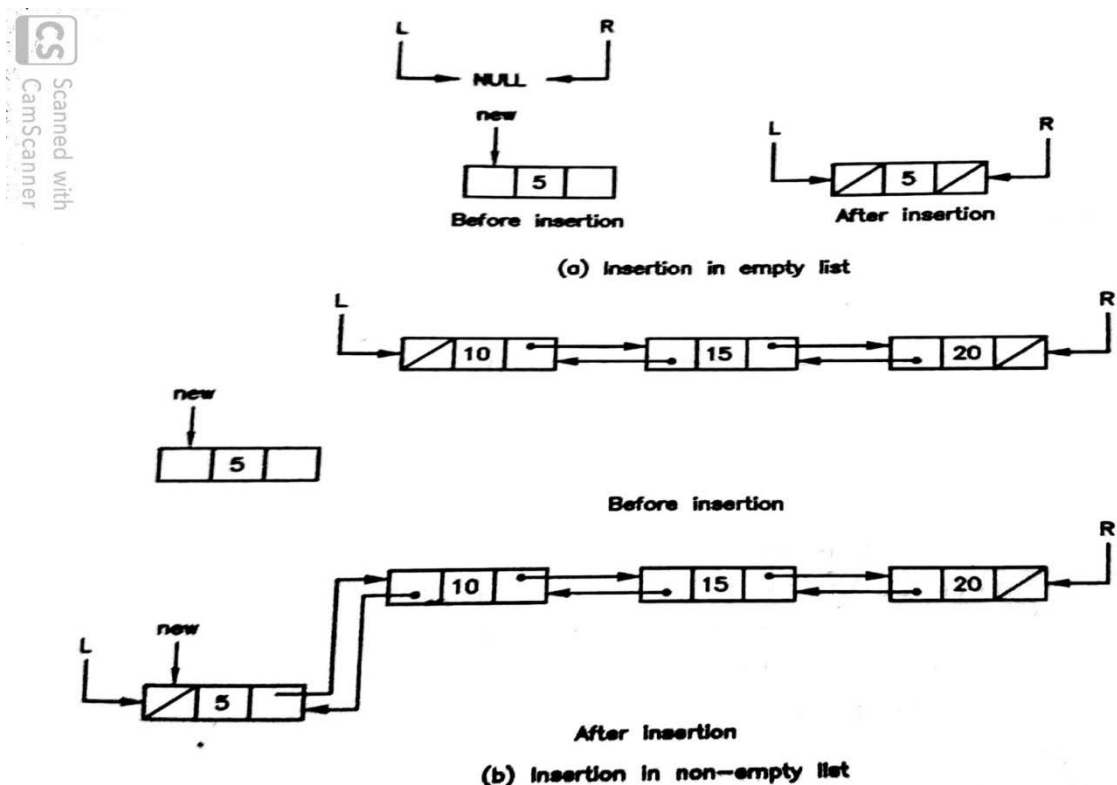


Fig 4.16 : Insertion in doubly linked list as left most node

The steps to write an algorithm to insert as left most node in doubly linked list as follows.

1. Allocate the memory dynamically for new node.
2. Initialize DATA field of new node by given value.
3. If list is empty (L is NULL), make LPTR and RPTR fields of new node NULL, copy **new** to both **L** and **R** and return.
4. Otherwise, to insert node perform following.
 - a. Make LPTR of new node NULL.
 - b. Copy existing **L** to RPTR of new node.
 - c. Copy **new** into LPTR of existing **L**
 - d. **new** becomes updated **L**.

The C function to insert new node as left most node in doubly linked list is as follows. Here, It is assumed that L and R pointers are defined globally and accessible to insert function and hence not passed as arguments.

```

/* Insert(left) accepts following arguments
   L, R (global) : Pointer to left most and right most nodes
   val : Data value to be inserted as left node
*/
void insert_left(int val)
{
    DNODE *new;

    /* Allocate memory for new node */
    new = (DNODE *)malloc(sizeof(DNODE));

    /* Initialize data part of new node by value val */
    new->data = val;

    /* Is list empty */
    if(L == NULL) {
        new->lptr = new->rptr = NULL;
        L = R = new;
        return;
    }

    /* Insert in non-empty list */
    new->lptr = NULL;
    new->rptr = L;

```

```
L->lptr = new;  
L = new;  
  
/* return */  
return;  
}
```

This function is called as follows to insert the node as left most node.

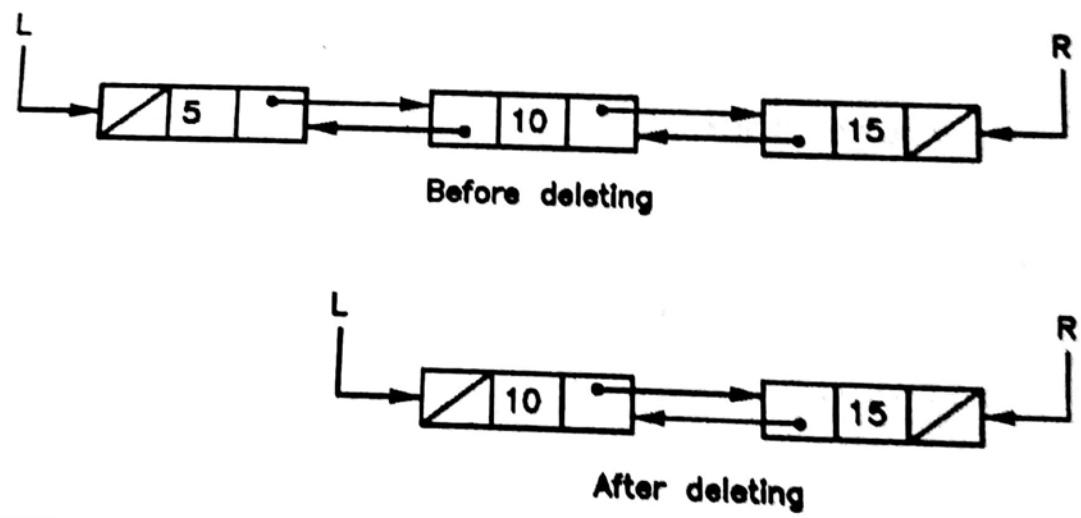
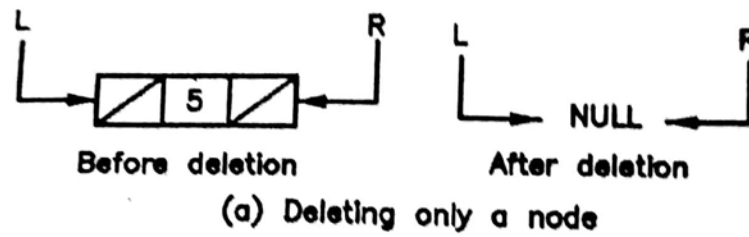
```
insert_left(val);
```

where **val** denotes the value of node to be inserted.

Let us now consider the deletion of left most node. There are three possibilities:

1. List is empty and deletion can not be done.
2. List contains only one node. Deleting that will make list empty.
3. List contains more than one nodes and left most node gets deleted which makes next node left most node.

Fig. 4.17 shows last two case from above before and after deletion.



CS Scanned with CamScanner

Fig. 4.17 : Deleting left most node from doubly linked list

The steps for writing an algorithm to delete left most node is as follows.

1. If list is empty (L is NULL), return with message "List is Empty".
2. If list contains only one node (L equals R), delete node pointed by L from memory and make L and R both NULL.
3. Otherwise to delete left most node perform following.
 - a. Store L in temporary pointer.
 - b. Copy RPTR of L (address of second node) to L.
 - c. Make LPTR of new L, NULL.
 - d. Delete node from memory using temporary pointer.

The C function to delete the left most node is as follows.

```
/* Delete(left) accepts following arguments
   L, R (global) : Pointer to left most and right most nodes
```



```

*/
void delete_left()
{
    DNODE *temp;

    /* Is list empty? */
    if (L == NULL) {
        printf("List is Empty\n");
        return;
    }

    /* Only one node */
    if(L == R) {
        free(L);
        L = R = NULL;
        return;
    }

    /* delete left most */
    temp = L;
    L = L->rptr;
    L->lptr = NULL;
    free(temp);

    /* return */
    return;
}

To delete the left most node above function is called as

delete_left();

```

Students are advised to write complete menu driven program using above insert and delete operations. Before you start operations following initialization is needed.

```

DNODE *L, *R;

L = R = NULL;

```

To design algorithms for insertion and deletion from right most position in doubly linked list, in above algorithms only roles of left and right pointers need to be changed.

4.8 LET US SUM UP

Singly linked list : Singly linked list is a linear data structure which uses dynamic memory allocation. Each node in singly linked list has two parts DATA and LINK.

DATA part contains information associated with node and LINK part points to next node. LINK is NULL in last node.

first : first is a pointer to first element in a singly linked list. If first is NULL, list is empty.

Circular linked list : If last node in a singly linked list points to first node then it is called circular linked list.

Head node : Head node is a special node having no data value. Its next part points to the first node in a circular linked list. When head node points to itself, list is empty. It is used to avoid the looping in circular linked list.

Doubly linked list : Doubly linked list is used to traverse in both the directions as each node contains two pointers LPTR and RPTR. LPTR points to previous node in the list and RPTR points to next node in the list.

L : L is a pointer to left most node in the doubly linked list.

R : R is a pointer to right most node in the doubly linked list.

4.9 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. Node in a singly linked list has only pointer to node.
2. node is not used to store user data.
3. Deletion of particular node inlist is faster than list.
4. Last node pointing to first node is known as list.
5. When singly linked list is empty, the first pointer is
6. is the condition for circular list to become empty.
7. When pointer to left most node L in doubly linked list is NULL, pointer to right most node R must be
8. If $L = R$ in doubly linked list, List contains..... nodes.

➤ **True-False**

1. Array is an example of static and sequential memory allocation.
2. Static memory allocation is useful when data size is unknown.
3. Linked list makes best use of memory.
4. Singly linked list allows traversal in both directions.
5. No node in circular list contains NULL pointer in its LINK field.

6. Doubly linked list takes more space.
7. Deletion operation in singly linked list needs only address of node to be deleted.
8. Array can not be created dynamically.
9. Linked list is more useful for data where frequent insertion and deletion are required.
10. When pointer first is NULL, singly linked list is empty.

➤ **Answer in brief**

1. Explain the node structure for a singly linked list.
2. Give the major limitations of singly linked list.
3. What is disadvantage of doubly linked list?
4. What is head node? Give its usefulness.
5. Why is deletion operation costlier in singly linked list?
6. Discuss the condition for empty list in case of singly, circular and doubly linked list.
7. Compare static and dynamic memory allocation.
8. Compare sequential and non-sequential memory allocation.
9. Compare array and linked list.
10. Compare singly and doubly linked list.

➤ **Answer the following**

1. Give and explain algorithm to insert a node in front of singly linked list.
2. Develop an algorithm to insert node as third node in the singly linked list. Assume that list has minimum two nodes.
3. Develop an algorithm to insert new node as last node in the circular list. Give complete C function for it.
4. Give the steps to write an algorithm for inserting a node as right most node in doubly linked list.
5. Develop an algorithm to delete a node with given value from doubly linked list.
6. Write an algorithm to count number of nodes in a circular list.
7. Write an algorithm to delete all the nodes after node having given value.

4.10 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

4.11 ASSIGNMENT

1. Develop a C function `count_odd()` to count only nodes having odd value as data in singly linked list. Use it in `main()` to show its working.
2. Develop a C function `reverse_list()` to reverse a singly linked list. Use it in `main()` to show its working.
3. Develop a C function `insert_sort()` to insert nodes in singly link list such that list remains in ascending order. Demonstrate its use with `main()`.
4. Develop a C function `split()` to divide given singly linked list into two lists, one containing only nodes with odd values and second containing only nodes with even values from original list. Use it in `main()` to show its working.
5. Develop a C function `copy_list()` to create a duplicate copy of singly linked list. Use it in `main()` to show its working.
6. Develop a menu driven C program to insert and delete nodes at the end of circular list. Program should also provide option for displaying list.
7. Develop a menu driven C program to insert and delete the node from right most position in doubly linked list. Program should also provide option for displaying contents of doubly linked list.

Block-4

**Sorting & Searching, Nonlinear
Data Structure**

Unit 1: Sorting

1

Unit Structure

- 1.1. Learning Objectives
- 1.2. Introduction
- 1.3. Selection Sort
- 1.4. Bubble Sort
- 1.5. Quick Sort
- 1.6. Merge Sort
- 1.7. Radix Sort
- 1.8. Let us sum up
- 1.9. Check your Progress
- 1.10. Further Reading
- 1.11. Assignment

1.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define the sorting and its use.
- Design algorithms for the various sorting techniques.
- Compare the various sorting technique in terms of their efficiency.
- Write the C code for various sorting techniques and use them in programming.

1.2 INTRODUCTION

In previous units, we have studied variety of primitive and non-primitive data structures with different operations on them. Apart from the operations discussed in previous units, another more common operation that can be performed on any data structure is sorting. The sorting operation is used to place the data in either ascending or descending order. Sorting is very common in business applications that use lots of data for different purpose. For example, banking application stores the information regarding the large number of customer accounts, which we can sort in order of account number or in order of name of customer.

There are many different sorting methods available to satisfy the varying performance needs of different applications. This chapter discusses the various sorting methods like selection sort, bubble sort, quick sort, radix sort, merge sort and heap sort.

1.3 SELECTION SORT

The selection sort is very simple and easy method. For a given array of size N , it starts by finding the position or index of the smallest number. Once the index of the smallest value is determined, the smallest value is exchanged with the first value. Thus, in first pass the smallest number is placed at first position. In second pass, array is considered from second to last position, the index of the smallest among them (second smallest for entire array) is again determined, and the smallest number is exchanged with the second value which places second smallest to its

position. Third pass considers the array from third to last position and repeats the same process. For array of size N , total of $N - 1$ passes are needed to put whole array in ascending order.

Let us understand the selection sort process with the help of example. Fig. 1.1 shows all the passes of the selection sort. Initially array is unsorted and we start pass-1 with finding index of smallest value. The smallest value is 9 with index 3. It is marked with arrow in fig. 1.1(a). Then it is exchanged with value at index 0 (first value) and the array after pass-1 is shown in fig. 1.1(b) where solid line in array indicates that the values above it are already sorted. Pass-2 finds the index of next smallest only from values below solid line. It is value 15 with index 3 marked by arrow in fig. 1.1(b). The value 15 is exchanged with value at index 1 (second value) and the array after pass-2 is shown in fig. 1.1(c). The solid line placed after second value denotes that first two numbers are in order. Performing subsequent passes in same manner sort whole array (fig. 1.1(d) to (h)). In our example as there are total 7 values, we need 6 passes as last value automatically comes into its position.

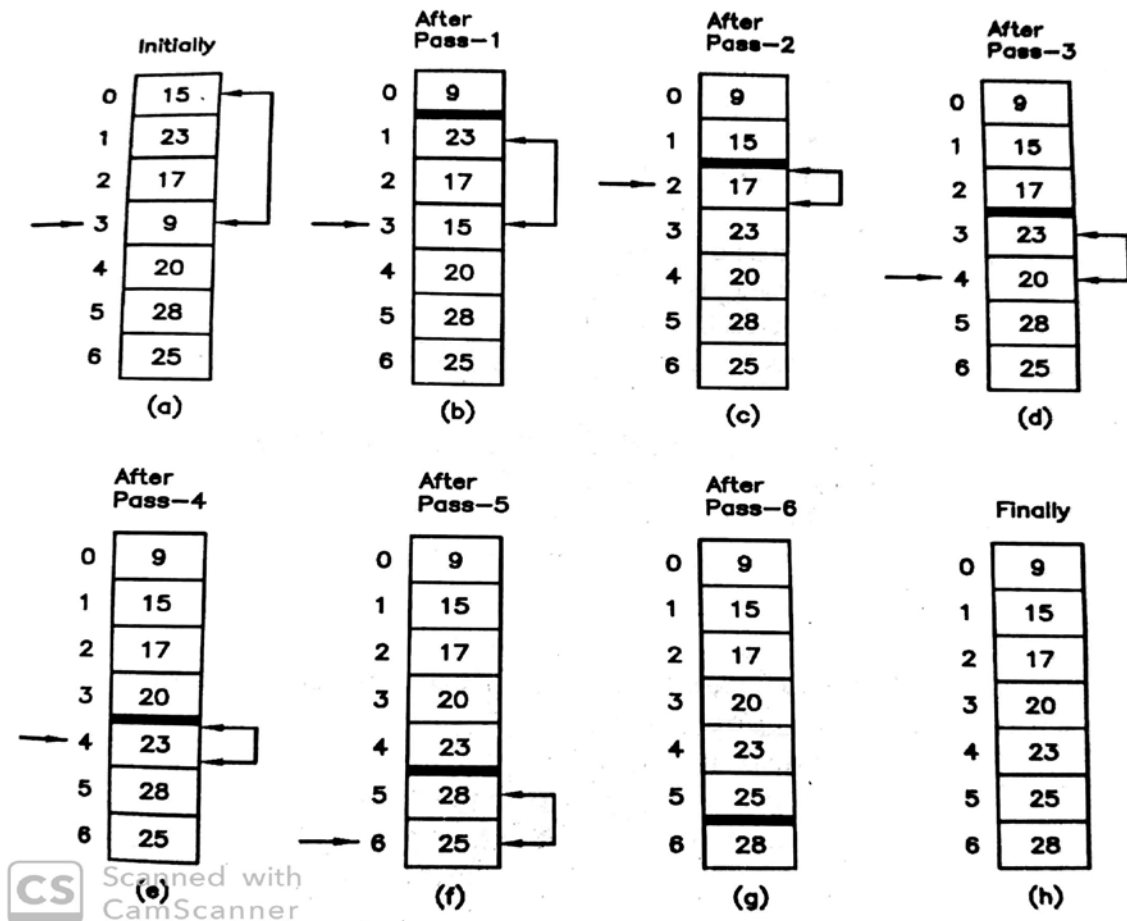


Fig. 1.1 : Passes of selection sort

The steps of selection sort algorithm are as follows. The variable **n** denotes the size of the array.

1. Repeat following steps for passes **p = 0** to **n - 1**
 - a. Find the index of next smallest as **min_pos**, from values at index **p** to **n - 1**
 - b. IF **min_pos** is not equal to pass **p**, then exchange the values at **min_pos** and **p**

The C code for performing selection sort is as follows.

```
/* Selection sort accepts the following arguments
   a : Array to be sorted
   n : Size of the array
   and returns
   sorted array
*/
void sel_sort(int a[], int n)
{
    int p,i,min_pos,temp;

    /* perform selection sort */
    for(p=0;p<n-1;p++) {
        /* Initialize */
        min_pos = p;

        /* find the index of next smaller */
        for(i=p+1;i<n;i++) {
            if(a[i] < a[min_pos])
                min_pos = i;
        }

        /* place next smaller to its position */
        if(min_pos != p) {
            temp = a[p];
            a[p] = a[min_pos];
            a[min_pos] = temp;
        }
    }
}
```

To sort the given array **a** with size **n**, it is called as

```
sel_sort(a,n);
```

Once it returns, the array **a** is sorted in ascending order. Program 5.1 demonstrates the selection sort.

The major advantage of the selection sort is it performs the exchange operation only once in each pass. However, it always need $N - 1$ passes.

Program 1 :

Write a program to implement the selection sort algorithm.

```
#include <stdio.h>

void sel_sort(int [],int);

void main()
{
    int a[50],n,val,pos,i;

    /* Get the size of the array */
    printf("How many elements ? ");
    scanf("%d",&n);

    /* Get the elements of array */
    for(i=0;i<n;i++) {
        printf("a[%d] = ",i);
        scanf("%d",&a[i]);
    }

    /* sort the array */
    sel_sort(a,n);

    /* print the sorted array */
    printf("Sorted array -->\n");
    for(i=0;i<n;i++)
        printf("a[%d] = %d\n",i,a[i]);
}

/* Fuction to perform selection sort */
void sel_sort(int a[], int n)
{
    int p,i,min_pos,temp;

    /* perform selection sort */
    for(p=0;p<n-1;p++) {
        /* Initialize */
        min_pos = p;

        /* find the index of next smaller */
        for(i=p+1;i<n;i++) {
```

```

        if(a[i] < a[min_pos])
            min_pos = i;
    }

    /* place next smaller to its position */
    if(min_pos != p) {
        temp = a[p];
        a[p] = a[min_pos];
        a[min_pos] = temp;
    }

    /* Print array after each pass */
    printf("After Pass %d : ",p+1);
    for(i=0;i<n;i++)
        printf("%d ",a[i]);
    printf("\n");
}
}

```

The result of execution of the above program is shown below.

How many elements ? 7

a[0] = 15

a[1] = 23

a[2] = 17

a[3] = 9

a[4] = 20

a[5] = 28

a[6] = 25

After Pass 1 : 9 23 17 15 20 28 25

After Pass 2 : 9 15 17 23 20 28 25

After Pass 3 : 9 15 17 23 20 28 25

After Pass 4 : 9 15 17 20 23 28 25

After Pass 5 : 9 15 17 20 23 28 25

After Pass 6 : 9 15 17 20 23 25 28

Sorted array -->

a[0] = 9

a[1] = 15

a[2] = 17

a[3] = 20

a[4] = 23

a[5] = 25

a[6] = 28

1.4 BUBBLE SORT

The bubble sort is also simple and easy to understand. The basic principle of bubble sort is that the lighter values moves up in each pass as like the air bubble moves up in the water. Bubble sort compares two values and if they are out of order (first value is greater than second value) then they are exchanged (placed in order). Assuming array of N elements (0 to $N - 1$), during pass-1 it starts comparing values at first and second position and if they are out of order, they are exchanged. Then second and third values are compared and if they are out of order they are exchanged. Similarly, third and fourth, fourth and fifth up to the second last and last values are compared and if any of them found out of order, they are placed in order by exchanging them. At the end of pass-1, the largest value goes to last ($N - 1$) position. Pass-2 considers the array of size $N - 1$ (0 to $N - 2$) and repeats the process which places second largest value to its position ($N - 2$). Pass-3 repeats the same process on values from 0 to $N - 3$. This way all the required passes are performed to complete the sorting of whole array. Bubble sort keeps track of how many exchanges are performed during each pass. Next pass is performed only if at least one exchange is performed. If no exchange is performed during any pass means data is sorted and process completes.

Fig. 1.2 shows the pass-1 of the bubble sort for an array with $N = 7$. First, the values at first and second position i.e. 15 and 23 are compared, but as they are in order, exchange is not required. Then values at second and third position i.e. 23 and 17 are compared and they are exchanged as $23 > 17$. Then values at third and fourth position i.e. 23 and 9 are compared and exchanged as $23 > 9$. Continuing this way gets the largest value 28 at the last position. Similar way, we can perform the second pass, after reducing size of array by 1 i.e. consider size $N = 6$ (0 to 5) only. Fig. 1.3 shows the array with all the passes where it is visible that how lighter values moves up in each pass. During pass-4, no exchange is performed and process completes.

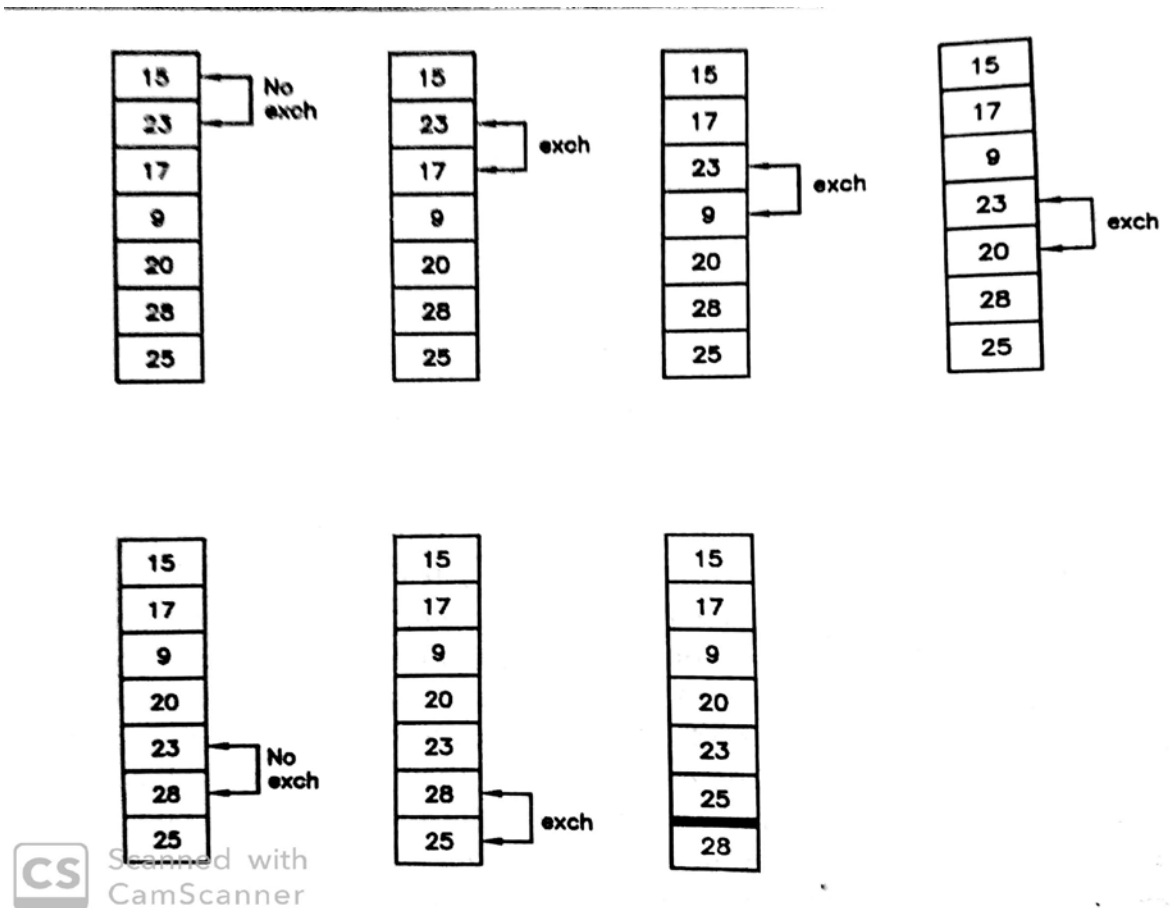


Fig. 1.2 : Pass-1 of bubble sort

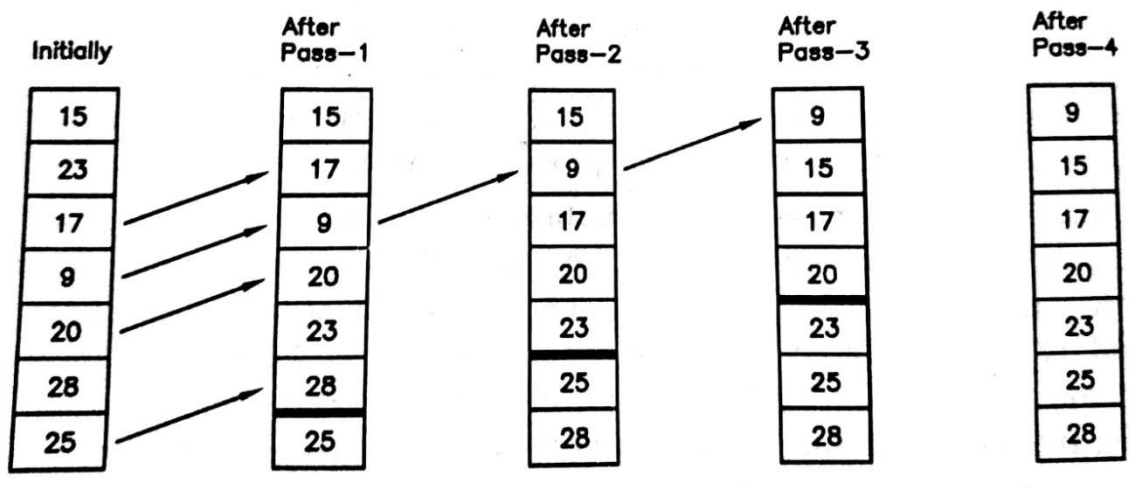


Fig. 1.3 : Passes of bubble sort

The steps for bubble sort algorithm are as follows. The variable n denotes the size of the array.

1. Initialize end with $n - 1$

2. Repeat following steps for passes **p = 0** to **n - 1**
 - a. Initialize **n_exch** with 0
 - b. Repeat following steps for position **i = 0** to **end**
 - i. Compare **ith** and **(i+1)th** values. If they are out of order exchange them and increment **n_exch**
 - c. If any exchange is performed (**n_exch != 0**), then reduce **end** by 1 else complete the process.

The C code for the bubble sort function is as follows.

```

/* Bubble sort accepts the following arguments
a : Array to be sorted
   n : Size of the array
and returns
   sorted array
*/
void bubble_sort(int a[], int n)
{
    int end,p,i,n_exch,temp;

    /* Initialize */
    end = n - 1;

    /* perform bubble sort */
    for(p=0;p<n;p++) {
        n_exch = 0;

        /* Place values in order */
        for(i=0;i<end;i++) {
            if(a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                n_exch++;
            }
        }

        /* Is sorted? */
        if(n_exch != 0)
            end--; /* reduce the size */
        else
            break; /* sorting completes */
    }
}

```

To sort the given array **a** with size **n**, it is called as

```
bubble_sort(a,n);
```

Once it returns, the array **a** is sorted in ascending order. Program 5.2 demonstrates the selection sort.

Comparing to selection sort, bubble sort performs more exchange operations as it exchanges the values once they are out of order. However, the advantage is that it takes maximum $N - 1$ passes, while selection sort takes always $N - 1$ passes. In bubble sort, the subsequent passes are not performed if no exchange occurs in current pass.

Program 2 :

Write a program to implement the bubble sort algorithm.

```
#include <stdio.h>

void bubble_sort(int [],int);

void main()
{
    int a[50],n,val,pos,i;

    /* Get the size of the array */
    printf("How many elements ? ");
    scanf("%d",&n);

    /* Get the elements of array */
    for(i=0;i<n;i++) {
        printf("a[%d] = ",i);
        scanf("%d",&a[i]);
    }

    /* sort the array */
    bubble_sort(a,n);

    /* print the sorted array */
    printf("Sorted array -->\n");
    for(i=0;i<n;i++)
        printf("a[%d] = %d\n",i,a[i]);
}

/* Fuction to perform bubble sort */
void bubble_sort(int a[], int n)
{
    int end,p,i,n_exch,temp;
```

```

/* Initialize */
end = n - 1;

/* perform bubble sort */
for(p=0;p<n;p++) {
    n_exch = 0;

    /* Place values in order */
    for(i=0;i<end;i++) {
        if(a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            n_exch++;
        }
    }

    /* Print array after each pass */
    printf("After Pass %d : ",p+1);
    for(i=0;i<n;i++)
        printf("%d ",a[i]);
    printf("\n");

    /* Is sorted? */
    if(n_exch != 0)
        end--; /* reduce the size */
    else
        break; /* sorting completes */
}
}

```

The result of execution of the above program is shown below.

How many elements ? 7

a[0] = 15

a[1] = 23

a[2] = 17

a[3] = 9

a[4] = 20

a[5] = 28

a[6] = 25

After Pass 1 : 15 17 9 20 23 25 28

After Pass 2 : 15 9 17 20 23 25 28

After Pass 3 : 9 15 17 20 23 25 28

After Pass 4 : 9 15 17 20 23 25 28

Sorted array -->

a[0] = 9

a[1] = 15

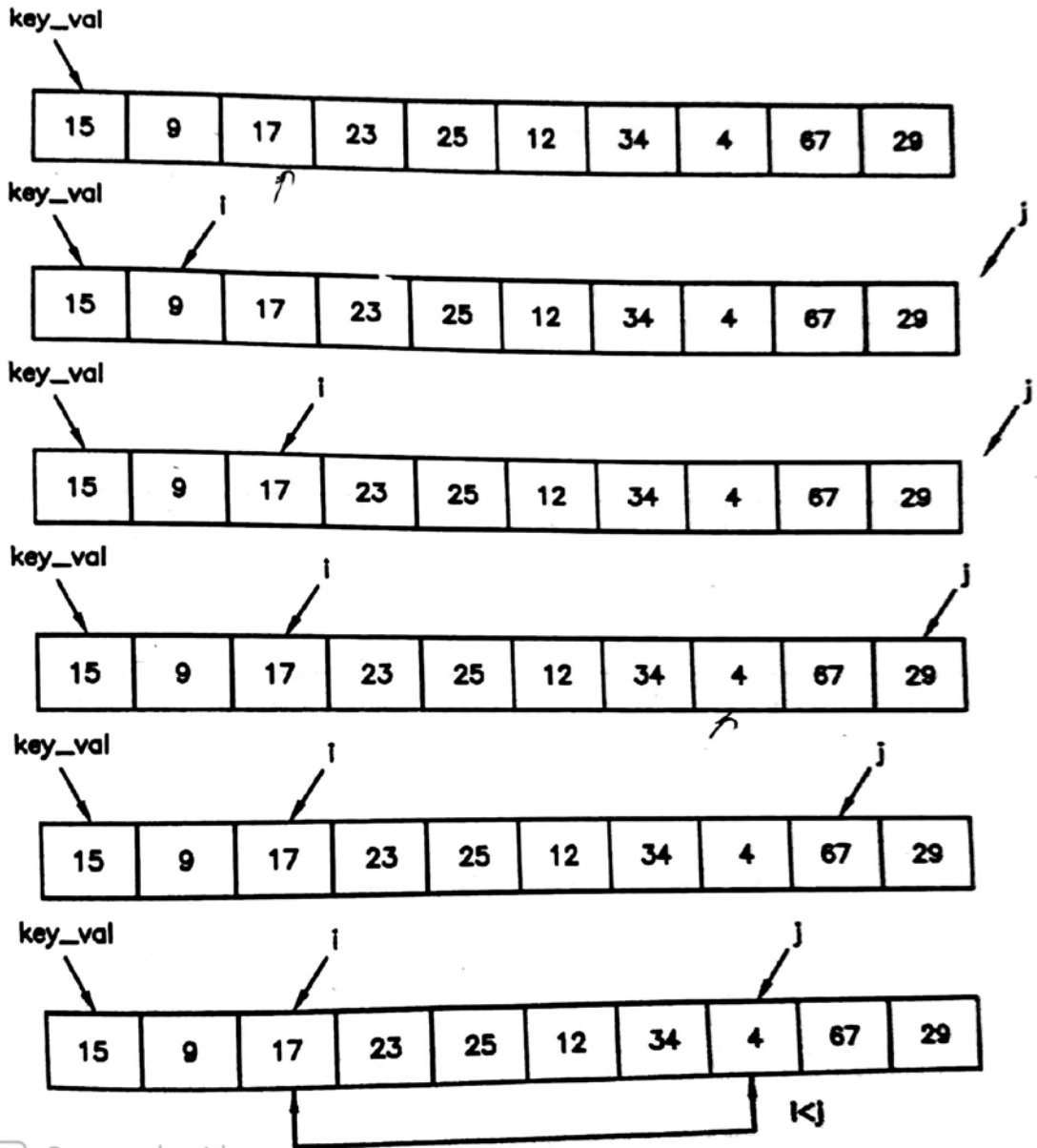
a[2] = 17

a[3] = 20
a[4] = 23
a[5] = 25
a[6] = 28

1.5 QUICK SORT

The quick sort is very efficient method for larger lists. The basic principle is that each time one key value is placed to its position-dividing list into two parts where left part contains values smaller than key value and right part contains values greater than key value. Process is then repeated for both the partitions. The dividing lists into smaller lists and repeating process continues until all the values comes to its place. Quick sort is also known as partition exchange sort as each times it divides list into two partitions.

Let us understand the process of quick sort with the help of example. Fig. 1.4 shows how key value 15 moves to its position. This requites two variables *i* and *j* as index variables. Variable *i* is initialized to next position after key value and *j* is initialized to after the end. Increment *i* as long as value at index *i* is less than key value. Once *i* stops decrement *j* as long as value at index *j* greater than key value. After this process if $i < j$, exchange the values at index *i* and *j*. As shown in fig., values 17 and 4 are exchanged. Then increment *i* to next position and repeat the process of incrementing *i* and decrementing *j* again. Once they stops, if $i < j$, exchange values at *i* and *j*. In fig., values 23 and 12 are exchanged. Repeat same process until, $i > j$. Once $i > j$, exchange the key value with value at index *j*. In fig. key value 15 and value 12 are exchanged. This places key value 15 to its place and list is partitioned into two parts as shown in fig. Then we have repeat the whole process separately for each list.



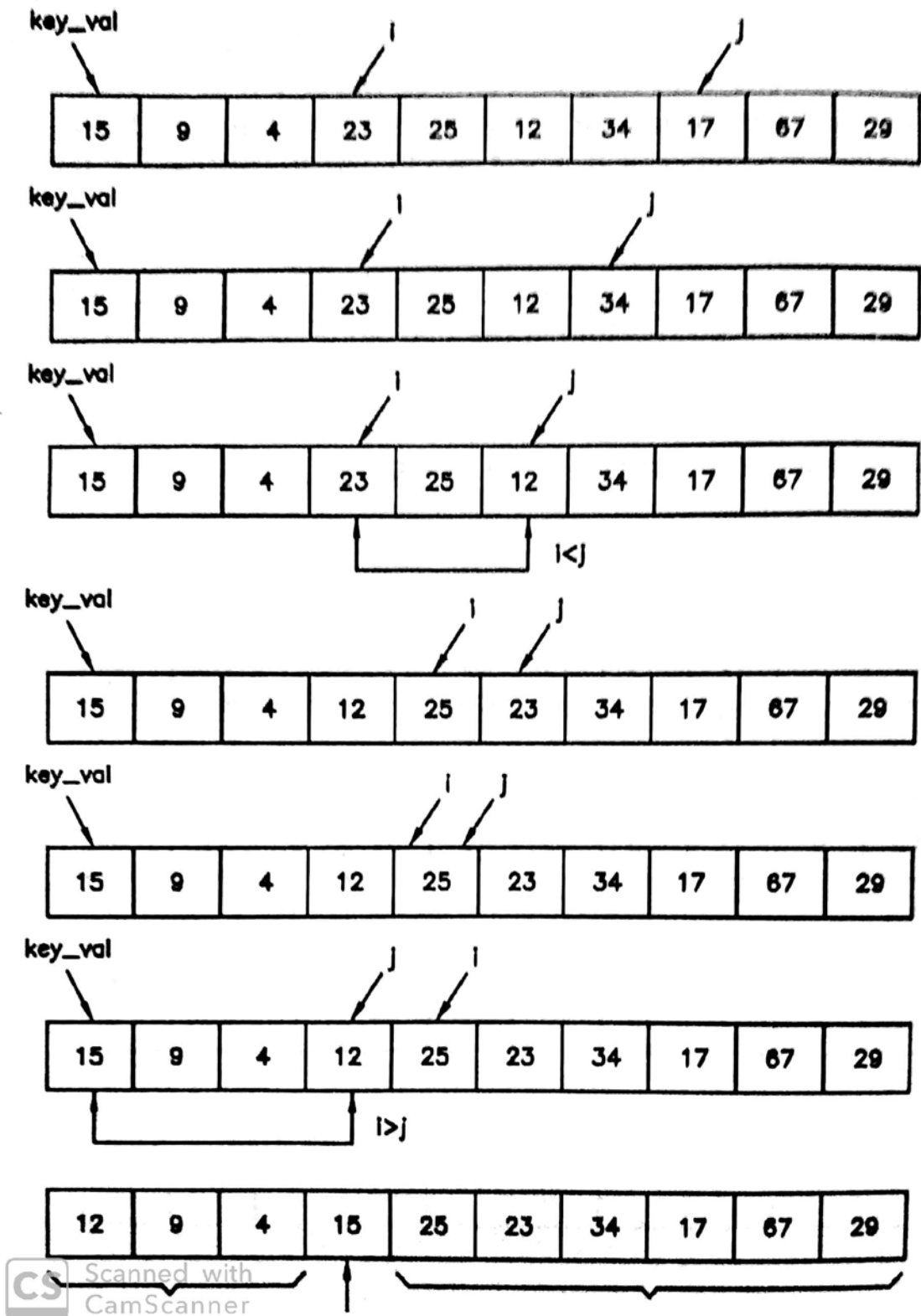


Fig. 1.4 : Pass-1 of quick sort

The steps for quick sort algorithm are as follows. Assume that variables **start** is 0 and **end** is $n - 1$ where **n** is the size of the array.

1. Initialize **flag = 1**
2. if **start < end** perform following steps
 - a. Initialize **i = start**, **j = end+1** and **key_val** to value at **start**
 - b. Repeat following steps while **flag** is 1
 - i. Increment **i**
 - ii. Repeat while value at **i** is less than **key_val**
 1. Increment **i**
 - iii. Decrement **j**
 - iv. Repeat while value at **j** is greater than **key_val**
 1. Decrement **j**
 - v. If **i** is less than **j**, exchange values at index **i** and **j**, else make **flag = 0**
 - c. Exchange values at index **start** and **j**
 - d. Call process for left partition (**start** to **j - 1**)
 - e. Call process for right partition (**j + 1** to **end**)

The C code for quick sort is as follows.

```
/* Quick sort accepts the following arguments
a : Array to be sorted
n : Size of the array
and returns
sorted array
*/
void quick_sort(int a[], int start, int end)
{
    int flag = 1, i, j, key_val, temp;

    /* perform quick sort */
    if(start < end) {
        i = start;
        j = end+1;
        key_val = a[start];
        while(flag) {
```

```

        i++;
        while(a[i] < key_val)
            i++;
        j--;
        while(a[j] > key_val)
            j--;
        if(i < j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
        else
            flag = 0;
    }

    /* place the key to its position */
    temp = a[start];
    a[start] = a[j];
    a[j] = temp;

    /* sort the left partition */
    quick_sort(a,start,j-1);
    /* sort the right partition */
    quick_sort(a,j+1,end);
}
}

```

To sort the given array **a** with size **n**, it is called as

```
quick_sort(a,0,n-1);
```

Once it returns, the array **a** is sorted in ascending order.

Program 3 :

Write a program to implement the quick sort algorithm.

```

#include <stdio.h>

void quick_sort(int [],int,int);

void main()
{
    int a[50],n,val,pos,i;

    /* Get the size of the array */
    printf("How many elements ? ");
    scanf("%d",&n);

```

```

/* Get the elements of array */
for(i=0;i<n;i++) {
    printf("a[%d] = ",i);
    scanf("%d",&a[i]);
}

/* sort the array */
quick_sort(a,0,n-1);

/* print the sorted array */
printf("Sorted array -->\n");
for(i=0;i<n;i++)
    printf("a[%d] = %d\n",i,a[i]);
}

/* Fuction to perform quick sort */
void quick_sort(int a[], int start, int end)
{
    int flag = 1,i,j,key_val,temp;

    /* perform quick sort */
    if(start < end) {
        i = start;
        j = end+1;
        key_val = a[start];
        while(flag) {
            i++;
            while(a[i] < key_val)
                i++;
            j--;
            while(a[j] > key_val)
                j--;
            if(i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
            else
                flag = 0;
        }

        /* place the key to its position */
        temp = a[start];
        a[start] = a[j];
        a[j] = temp;

        /* sort the left partition */
        quick_sort(a,start,j-1);
        /* sort the right partition */
        quick_sort(a,j+1,end);
    }
}

```

```
}  
}
```

The result of execution of the above program is shown below.

How many elements ? 10

a[0] = 15

a[1] = 9

a[2] = 17

a[3] = 23

a[4] = 25

a[5] = 12

a[6] = 34

a[7] = 4

a[8] = 67

a[9] = 29

Sorted array -->

a[0] = 4

a[1] = 9

a[2] = 12

a[3] = 15

a[4] = 17

a[5] = 23

a[6] = 25

a[7] = 29

a[8] = 34

a[9] = 67

1.6MERGE SORT

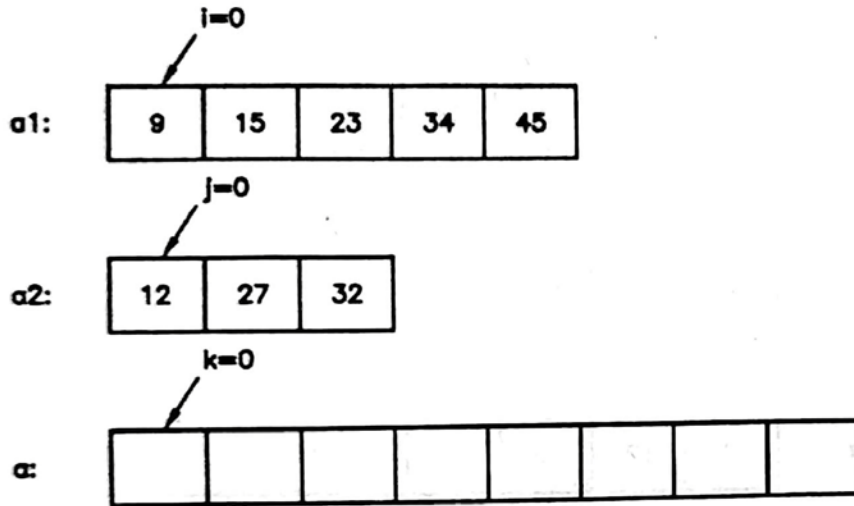
Merging means combining two sorted arrays or lists into a single sorted array or list. This is sometime also referred to as external sort as it does not sort the unsorted list but combines sorted lists into a larger sorted list. It maintains two pointers **i** and **j** one for each sorted list, initially pointing to first values in each of the sorted list. Third pointer **k** is used to point to next free location in combined sorted list. Merge sort compares the values at **i**th position in first list and **j**th position in second list and copies whichever is smaller at **k**th position into a combined list (if equal any one can be copied). It then increments corresponding pointer either **i** or **j** from which value has been copied as well as **k**. This process is repeated until one of the lists is over. Then rest of the values from another list are copied to combined list one by one till the end of that list.

Let us understand the merge sort with the help of example. Assume that following two sorted arrays are to be combined into one larger sorted array.

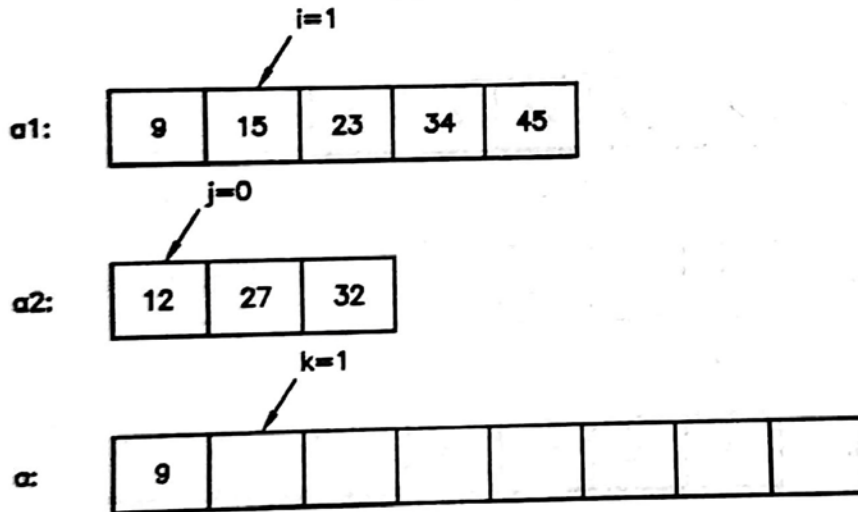
a1 : 9,15,23,34,45

a2 : 12,27,32

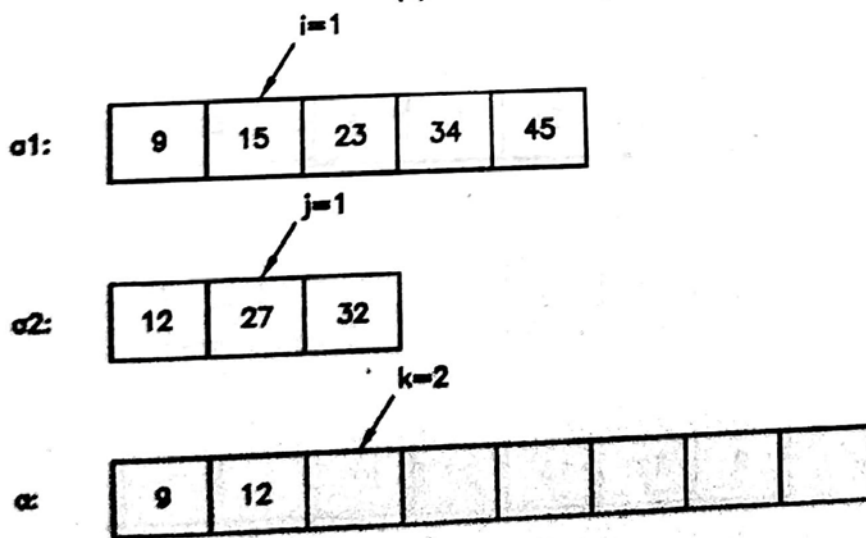
Merging of above arrays is shown in fig. 1.5. Fig. 1.5(a) shows the initial position with pointers **i**, **j** and **k** are 0. As **a1[i]** is less than **a2[j]** ($9 < 12$), **a1[i]** (value 9) is copied to **a[k]** as and pointer **i**(becomes 1) and **k**(becomes 1) are incremented as shown in fig. 1.6(b). The same process is repeated as shown in fig. 1.5(c) to (g) where the array **a2** is over. Then rests of the values from **a1** (34 and 45) are copied one by one as shown in fig. 1.5(h) and (i) to array **a**. The array **a** in fig. 1.5(i) is the final result.



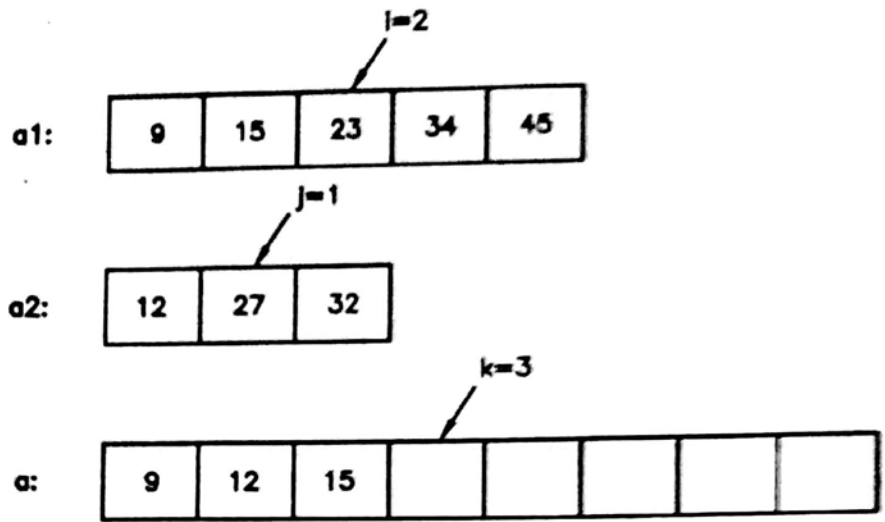
(a) Initially



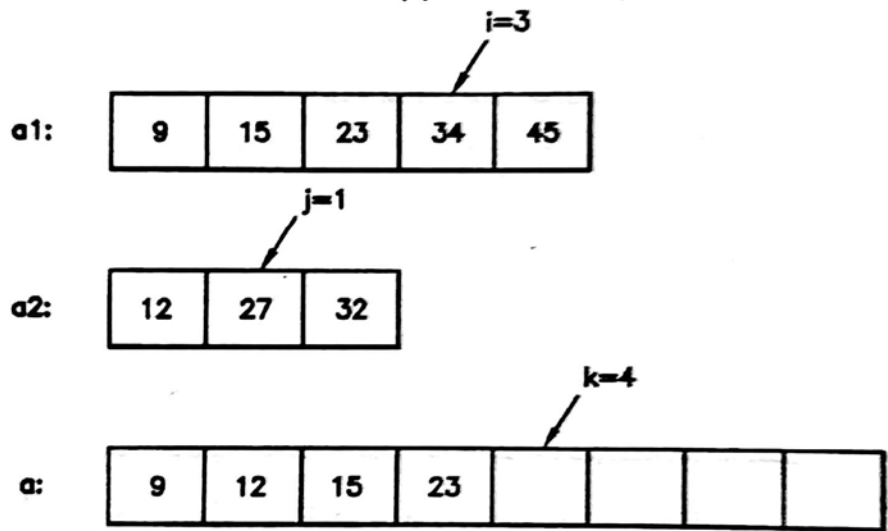
(b) After first operation



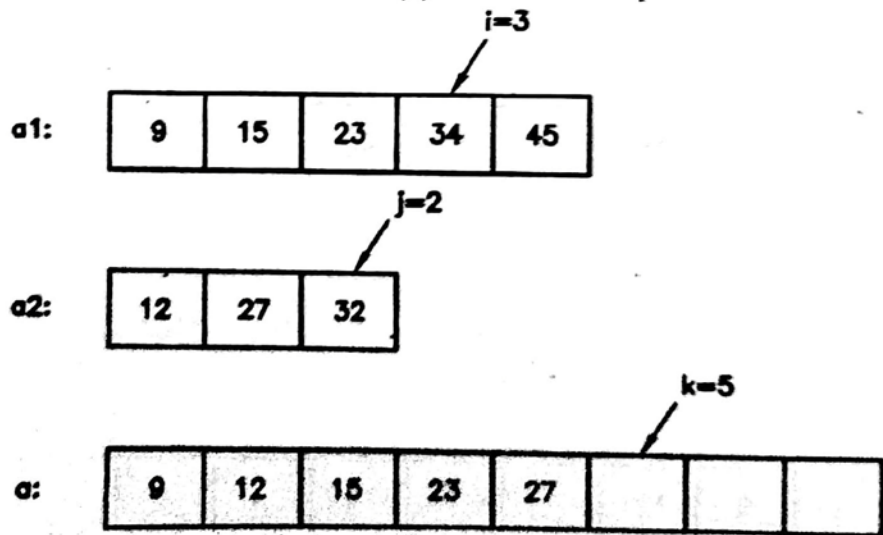
(c) After second operation



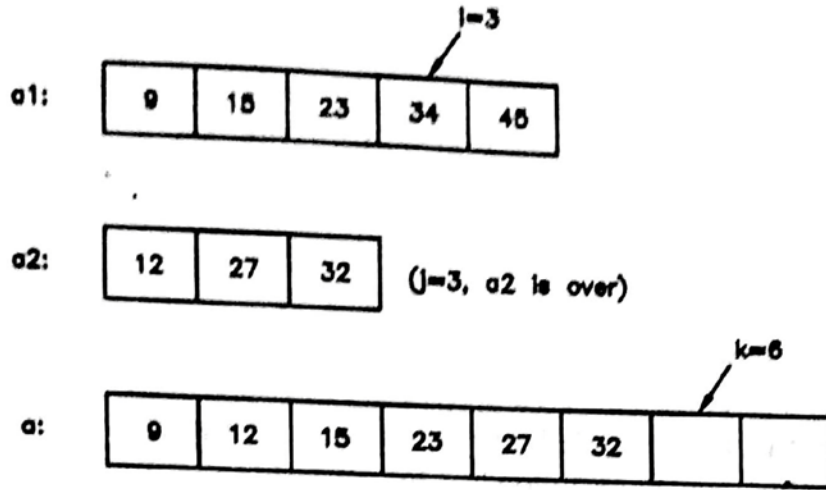
(d) After third operation



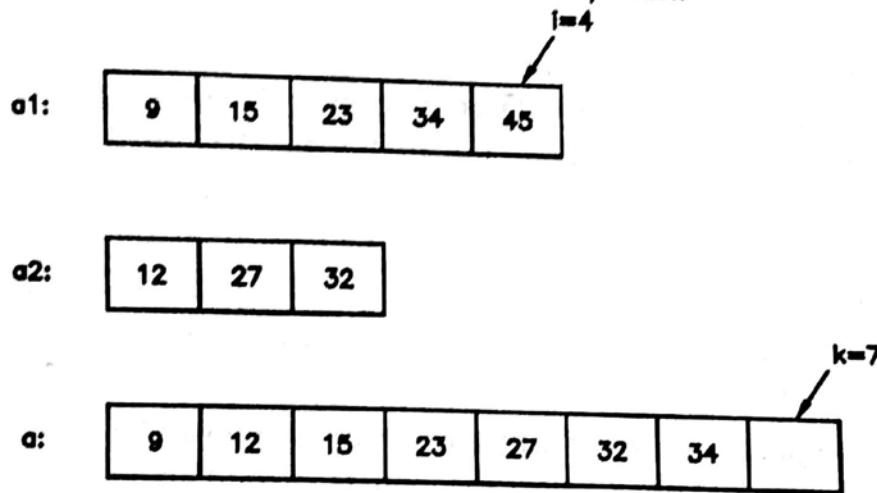
(e) After fourth operation



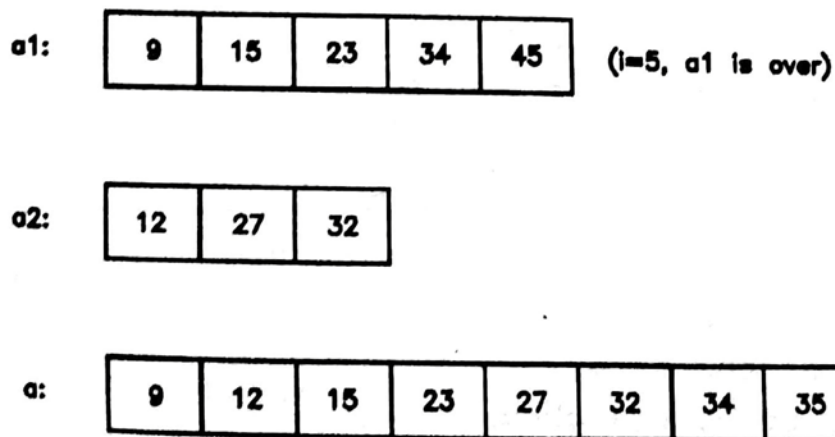
(f) After fifth operation



(g) After sixth operation



(h) After seventh operation



(i) After eight operation

Fig. 1.5 : Simple merge

The above method is also called the simple merge as it merges the two sorted arrays or tables into one. Following C program implements the simple merge to merge two sorted arrays. Program accepts two sorted arrays and combines them into one array side by side i.e. first array starts from the position **first** and second array starts from the position **second**. Position **third** denotes the end of the second array.

Program 4 :

Write a C program to implement the simple merge to merge two sorted arrays.

```
#include<stdio.h>
#include<conio.h>
#define N 50

void merge(int *,int,int,int);

void main()
{
    int a[N],first=0,second,third,i;
    clrscr();

    /* Enter two sorted arrays */
    printf("How many values in array1 ? : ");
    scanf("%d",&second);
    printf("Enter the values of array1\n");
    for(i=0;i<second;i++)
        scanf("%d",&a[i]);

    printf("How many values in array2 ? : ");
    scanf("%d",&third);
    third=third+second;
    printf("the values of array2\n ");
    for(i=second;i<third;i++)
        scanf("%d",&a[i]);

    /* merge the array */
    merge(a,first,second-1,third-1);

    /* print combined array */
    printf("Sorted values\n");
    for(i=0;i<third;i++)
        printf("%d\n",a[i]);

    getch();
}
```

```

/* Function to merge two array */
void merge(int *a,int first,int second,int third)
{
    int L=-1,i=first,j=second,*temp,count;

    /* compare and copy smaller */
    while(i<second && j<=third)
    {
        if(a[i]<=a[j])
        {
            L++;
            temp[L]=a[i];
            i++;
        }
        else
        {
            L=L+1;
            temp[L]=a[j];
            j++;
        }
    }

    /* copy remaining unsorted elements */
    if(i>=second)
    {
        while(j<=third)
        {
            L=L+1;
            temp[L]=a[j];
            j++;
        }
    }
    else
    {
        while(i<second)
        {
            L++;
            temp[L]=a[i];
            i++;
        }
    }

    /* copy from temporary array to original */
    for(count=0;count<=L;count++)
        a[first+count]=temp[count];

    return;
}

```

Simple merge can be used to combine multiple sorted arrays into a large sorted array. For example, if we want to combine eight sorted arrays, we can use simple merge to combine each pair of the arrays reducing them into four arrays. In second pass, we can combine pairs from four arrays resulting into two arrays and finally in third pass two arrays are combined to single large sorted array. This forms the basis for the two-way merge sort in which we can consider each element as sorted array in an unsorted array and combine two elements. Then the sorted arrays of size two are combined and the process is repeated until all the elements of the array are in sorted order. Following C function implements the two-way merge sort.

```

/* Function to perform two-way merge sort */
void two_way_merge_sort(int arr[],int start,int end)
{
    Int size, middle;

    /* get the size of current part */
    size=end-start+1

    /* sorted part */
    if(size<=1)
        return;

    /* Get middle of current part */
    middle=start+(size/2)-1;
    /* recursively sort first part */
    two_way_merge_sort(arr,start,middle);
    /* recursively sort second part */
    two_way_merge_sort(arr,middle+1,end);
    /* merge two ordered parts */
    simple_merge(arr,start,middle+1,end);

    return;
}

```

To sort the given array **a** with size **n**, it is called as

```
two_way_merge_sort (a,0,n-1);
```

Students are advised to use above two-way merge sort function in C program and test it with sample data.

1.7 RADIX SORT

The radix sort is another sorting method, which is very useful specifically when the data values in list are smaller. For list containing decimal numbers, radix sort works by distributing values in 10 buckets (one for each decimal digit i.e. bucket 0 to bucket 9) and collecting back repeatedly. Let us understand the radix sort with help of following list of values.

15,23,17,9,25,12,34,4,67,29,78,58

In above list, the maximum number of digits in any value is 2 and hence two passes are needed to sort them. Assuming the leading zero for values, which are of single digit. For example, 4 is 04. Our initial list will become

15,23,17,09,25,12,34,04,67,29,78,58

During pass-1, the values are distributed from first to last based on the least significant digit of a value i.e. right most digit. First value 15 goes into bucket 5 as the least significant digit in 15 is 5. The next value 23 goes into bucket 3, 17 goes into bucket 7 and so on. When ever multiple values comes into a single bucket, they are placed on top of each other. The distribution of all the values of the list based on least significant digit is shown in fig. 1.6(a).

Values					12	23	04	25		67	58	29
Bucket	0	1	2	3	4	5	6	7	8	9		

Fig. 1.6(a) Pass-1 of radix sort

After distributing the values, we have to collect them back in a list from left to right (i.e. bucket 0 to bucket 9) and bottom to left in a same bucket. The collection is as follows.

12,23,34,04, 15,25,17,67,78,58,09,29

During pass-2, we have again distribute the above list based on next least significant digit i.e. second right most digit. The first value 12 goes into bucket 1, 23 goes into bucket 2, 34 into bucket 3 and so on. Fig. 1.6(b) shows the distribution of above list based on second right most digit.

Values	09	17	29							
	04	15	25							
	04	12	23	34		58	67	78		
Bucket	0	1	2	3	4	5	6	7	8	9

Fig. 1.6(b) Pass-2 of radix sort

Collecting them from buckets gives the following sorted list

04,09,12,15,17,23,25,29,34,58,67,78

Our example list is sorted in two passes of distribution and collection. In general, to determine the number of passes, we have to find the largest number from the list and count the number of digits of that number. For example, in our list the largest value is 78 having two digits. Distribution in each pass is performed based on the next least significant digit i.e. from right to left.

1.8 LET US SUM UP

Selection sort : The selection sort is very simple and easy method. For a given array of size N, it starts by finding the position or index of the smallest number. Once the index of the smallest value is determined, the smallest value is exchanged with the first value.

Bubble sort : The basic principle of bubble sort is that the lighter values moves up in each pass as like the air bubble moves up in the water.

Quick sort : The quick sort is very efficient method for larger lists. The basic principle is that each time one key value is placed to its position-dividing list into two parts where left part contains values smaller than key value and right part contains values greater than key value.

Merge sort : Merging means combining two sorted arrays or lists into a single sorted array or list. This is sometime also referred to as external sort as it does not sort the unsorted list but combines sorted lists into a larger sorted list.

Radix sort : The radix sort is another sorting method, which is very useful specifically when the data values in list are smaller. For list containing decimal

numbers, radix sort works by distributing values in 10 buckets (one for each decimal digit i.e. bucket 0 to bucket 9) and collecting back repeatedly.

1.9 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. Lighter values move up in each pass in sort.
2. sort performs only one exchange operation in each pass.
3. sort combines the sorted lists.
4. sort needs maximum $N - 1$ passes.
5. sort uses recursion.

➤ **True-False**

1. If sorted data is passed to bubble sort, it will come out in one pass only.
2. Selection sort always needs $N - 1$ passes.
3. Merge sort is external sort.
4. Quick sort uses the divide and conquer method as it divides list into two parts in each pass.

➤ **Answer in brief**

1. Give the principle of the selection sort.
2. Why do we use 10 buckets in radix sort?
3. Why quick sort is also called partition sort?
4. Compare selection and bubble sort.

➤ **Answer the following**

1. Explain the two-way merge sort.
2. Sort the following data using bubble sort.
10, 3, 56, 24, 78, 34, 47, 36
3. Sort the following data using radix sort.

24, 89, 34, 6, 123, 56, 234

- Sort the following data using bubble sort.

15,23,17,9,20,25,23

- Merge the following two tables using simple merge.

T1 : 34 68 200 456

T2 : 25 57 97 157

- Write short notes.

- Merge sort
- Two-way merge sort
- Radix sort
- Quick sort
- Selection sort
- Bubble sort

1.10 FURTHER READING

- Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
- Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
- Data Structures Using C, Reema Thareja, Oxford
- Data Structures and Program Design in C, Kruse, Pearson
- Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
- Website : <https://www.geeksforgeeks.org/data-structures/>

1.12 ASSIGNMENT

- Develop a C function to create the heap for a list of numbers. Use it to perform the heap sort.
- Assume that we want to store student details roll number, name and marks using structure. Develop a program which accepts records of N students and sorts them in order of roll number using bubble sort. Use sorted records to search the details of a student once the roll number is given using binary search method.

Unit 2: Searching

2

Unit Structure

- 2.1 Learning Objectives
- 2.2 Introduction
- 2.3 Sequential Search
- 2.4 Binary Search
- 2.5 Hash Tables
- 2.6 Hashing Functions
- 2.7 Collision-Resolution Techniques
- 2.8 Let us sum up
- 2.9 Check your Progress
- 2.10 Further Reading
- 2.11 Assignment

2.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define the searching and state its need.
- Develop an algorithm to perform sequential search and analyze its performance.
- Develop an algorithm to perform binary search and analyze its performance.
- Define hash table and hashing function.
- Demonstrate the use need and use of the collision resolution technique.

2.2 INTRODUCTION

Search operation is used to find whether the desired value is present in set of value or not. If it is present, the search operation provides us its position in that set of values which we can use later to get that value. The search operation is very common in many applications For example, in banking application we need to search the customer details using the account number as key. This unit discusses the most common search operations namely; sequential (or linear) and binary search as well as hashing. Linear search is very simple technique but inefficient for large set of data. To solve the performance issues of the linear search, binary search is used. However, binary search needs data in sorted order. The performance of both linear and binary search depends on the size of the data. The hash searching is independent of the size of data and most efficient searching method. This unit covers the linear, binary and hash search with collision resolution techniques.

2.3 SEQUENTIAL SEARCH

Sequential search also known as linear search is very simple. It simply sequentially compares the given value with each value in list of values one by one from start to end until value is found or the list is ended. If the value is found, the position of the value is returned. Assume that, we have following list of values stored in an array

50, 45, 42, 78, 65, 86, 34, 40, 70

and we want to search the value 78. The value 78 is compared to first value in array which is 50 and it does not match. Then it is compared with second value 45 which also does not match. Third value 42 also does not match to 78. Fourth value in array is 78 which same as value we are searching. The position at which the value match or found is 4. Thus we can say that given value is found in array at position 4.

As a second example, let us say value to be searched is 58. Comparing 58 with values in array one by one from start to end, none of the value in array matches to 58 and we can say “value not found”.

Following are the steps to write an algorithm for sequential search. Assume that array stores **n** values and value to be searched is stored in variable **val**

1. Repeat following steps for all **n** values in array
 - a. If value matches to next value, return the position of the value.
2. Return message “Value not found”

The C code for the sequential search is as follows.

```
/* seq_search operation accepts following arguments
   a      : Array of values
   n      : Size of the array
   val    : Value to be searched
and returns
   pos    : position at which value is found or -1 if not found
*/
int seq_search(int a[], int n, int val)
{
    int pos;

    /* search value sequentially */
    for(pos=0;pos<n;pos++) {
        if(val == a[pos])
            return(pos);
    }

    /* value not found, return -1 */
    return(-1);
}
```

To search the value, this function is called as

```
p = seq_search(a,n,val);
```

where **a** is the array having **n** elements in which we want to search value **val**. If found position is stored in **p**, otherwise p is -1.

Sequential search works on both sorted as well as unsorted data, but mainly used when data is unsorted. The major advantage of the sequential search is its simplicity as it is easy to understand and implement. The major drawbacks of sequential search are

- The search time is not unique. If value is in start it takes less time, but if value is near to end it takes more time.
- As the size of data increases, the search time also linearly increases.

Program 1 demonstrates the use of sequential search.

Program 1 :

Write a program to find a given integer in an array using sequential search.

```
#include <stdio.h>

int seq_search(int[], int, int);

void main()
{
    int a[50],n,val,pos,i;

    /* Get the size of the array */
    printf("How many elements ? ");
    scanf("%d",&n);

    /* Get the elements of array */
    for(i=0;i<n;i++) {
        printf("a[%d] = ",i);
        scanf("%d",&a[i]);
    }

    /* Get the elements to search */
    printf("Enter an element to search : ");
    scanf("%d",&val);
    /* Search the value to search */
    pos = seq_search(a,n,val);
```

```

        /* Print the result */
        if(pos == -1)
            printf("Value not found\n");
        else
            printf("Value %d is found at position %d\n",val,pos+1);
    }

    /* Fuction to perform sequential search */
    int seq_search(int a[], int n, int val)
    {
        int pos;

        /* search value sequentially */
        for(pos=0;pos<n;pos++) {
            if(val == a[pos])
                return(pos);
        }

        /* value not found, return -1 */
        return(-1);
    }

```

The result of executing above program is shown below.

Result 1:

```

How many elements ? 9
a[0] = 50
a[1] = 45
a[2] = 42
a[3] = 78
a[4] = 65
a[5] = 86
a[6] = 34
a[7] = 40
a[8] = 70
Enter an element to search : 78
Value 78 is found at position 4

```

Result 2:

```

How many elements ? 9
a[0] = 50
a[1] = 45
a[2] = 42
a[3] = 78
a[4] = 65
a[5] = 86
a[6] = 34

```

a[7] = 40
a[8] = 70
Enter an element to search : 58
Value not found

2.4 BINARY SEARCH

Binary search is more efficient than the sequential search, but it requires the sorted data. It first computes the **middle** which is index of the middle value in the list as

$$\text{middle} = (\text{start} + \text{end}) / 2$$

where **start** is the index of first value in the list and the **end** is the index of the last value in the list. Then it compares the value to be searched with value at index **middle**. If both matches, the value is found and the index **middle** is returned. If both does not match, then list is divided into two parts called upper half and lower half. All the values in upper half is less than the value at index **middle** and all the values in lower half are greater than the value at index **middle**. If value to be searched is less than value at index **middle**, then we have to repeat the same process with the upper half (**start** to **middle - 1**). Otherwise if value to be searched is greater than value at index **middle**, then we have to repeat the same process with the lower half (**middle + 1** to **end**). This process of dividing list into two parts is repeated until either the value is found or the list will have single value not matching the value to be searched.

Let us understand the process of binary search with the help of example. Assume that the sorted list is as follows

34, 40, 42, 45, 50, 65, 70, 78, 86

and we want search the value 42. Initially **start** is 0 and **end** is 8 (as total values are n=9). The **middle** = $(0+8)/2 = 4$. The value to be searched is less the value at index **middle** ($42 < 50$) as shown in fig. 2.1(a). The process is to be repeated with the upper half for which **start** = 0 and **end** = **middle - 1** = $4 - 1 = 3$. The new **middle** = $(0+3)/2 = 1$. The value 42 is greater than the value 40 at index **middle** as shown in

fig. 2.1(b). The next search is to be in lower part for which **start = middle + 1 = 1 + 1 = 2** and **end = 3**. The new **middle = (2+3)/2 = 2**. The value at index **middle(=2)** matches the 42 and we will return with 2 as the index of value to be searched as shown in fig. 2.1(c).

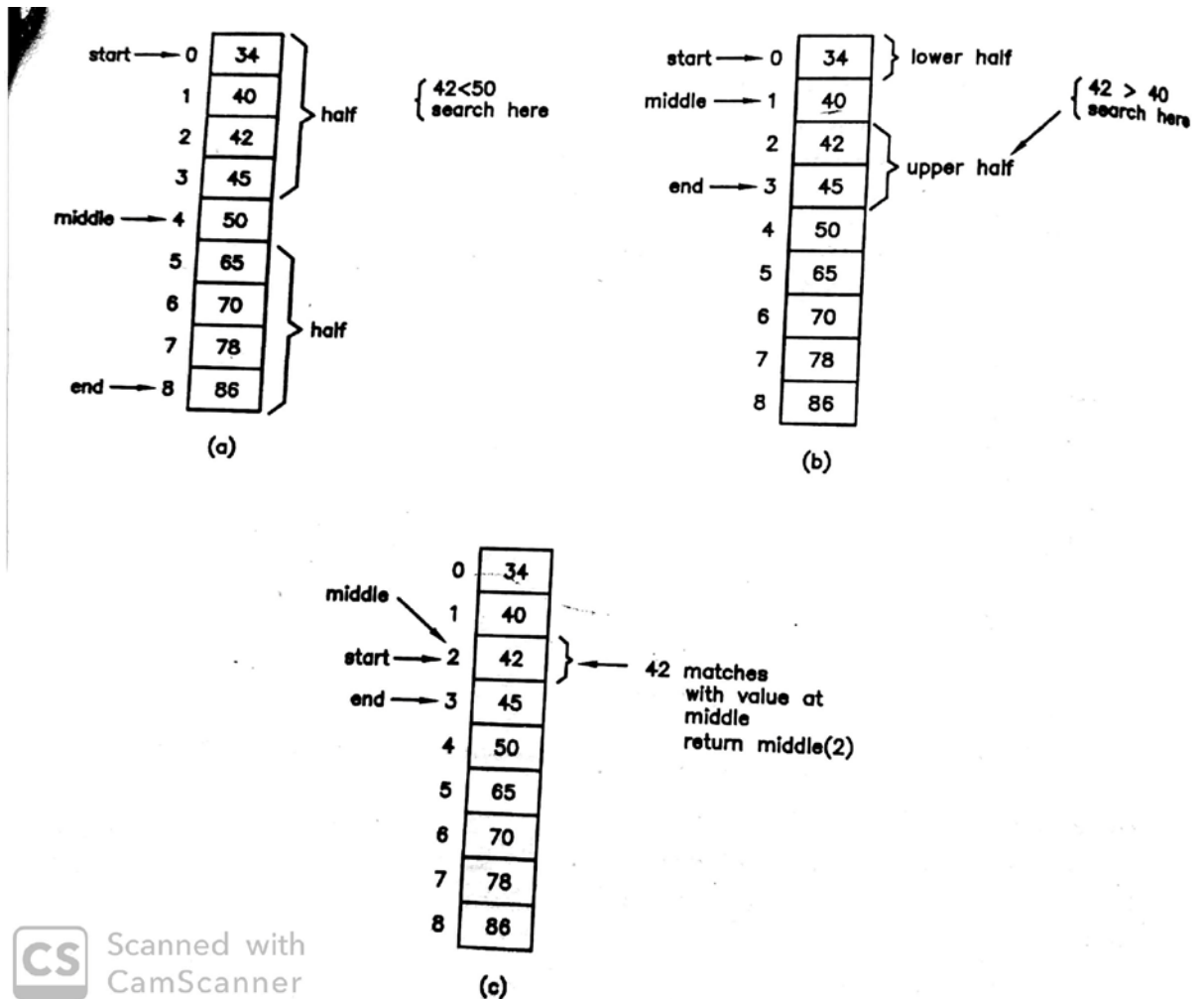


Fig. 2.1 : Binary search for finding 42

Following are the steps to write an algorithm for sequential search. Assume that array stores **n** values and value to be searched is stored in variable **val**

1. Initialize **start = 0** and **end = n - 1**
2. Repeat following steps till **start <= end**
 - a. Compute **middle = (start+end)/2**
 - b. if value at **middle** is less than **val** , then update **end** by **middle - 1**
 Otherwise if value at **middle** is grater than **val**, update **start** by **middle + 1**
 Otherwise, value at **middle** matches the **val** and return **middle**

3. Return with -1 as value is not found.

The C code for the sequential search is as follows.

```
/* bin_search operation accepts following arguments
   a      : Array of values
   n      : Size of the array
   val    : Value to be searched
and returns
   pos    : position at which value is found or -1 if not found
*/
int bin_search(int a[], int n, int val)
{
    int start,end,middle;

    /* Initialize */
    start = 0;
    end = n - 1;

    /* search value */
    while(start <= end) {
        /* Compute index of middle element */
        middle = (start+end)/2;
        /* Compare value with element at middle */
        if(val < a[middle])
            end = middle - 1; /* upper half */
        else if(val > a[middle])
            start = middle + 1; /* lower half */
        else
            return(middle); /* found */
    }

    /* value not found, return -1 */
    return(-1);
}
```

To search the value, this function is called as

```
p = bin_search(a,n,val);
```

where **a** is the sorted array having **n** elements in which we want to search value **val**.
If found position is stored in **p**, otherwise p is -1.

The major advantage of the binary search is that every time it reduces the search area to half so that even for larger data size the search time remains almost constant for searching any value regarding its place. Only limitation of the binary search is that it needs sorted data.

Program 2 :

Write a program to find a given integer in an array using binary search.

```
#include <stdio.h>
int bin_search(int [],int,int);

void main()
{
    int a[50],n,val,pos,i;

    /* Get the size of the array */
    printf("How many elements ? ");
    scanf("%d",&n);

    /* Get the elements of array */
    for(i=0;i<n;i++) {
        printf("a[%d] = ",i);
        scanf("%d",&a[i]);
    }

    /* Get the elements to search */
    printf("Enter an element to search : ");
    scanf("%d",&val);

    /* Search the value to search */
    pos = bin_search(a,n,val);

    /* Print the result */
    if(pos == -1)
        printf("Value not found\n");
    else
        printf("Value %d is found at position %d\n",val,pos+1);
}

/* Fuction to perform binary search */
int bin_search(int a[], int n, int val)
{
    int start,end,middle;

    /* Initialize */
    start = 0;
```

```

end = n - 1;

/* search value */
while(start <= end) {
    /* Compute index of middle element */
    middle = (start+end)/2;
    /* Compare value with element at middle */
    if(val < a[middle])
        end = middle - 1;    /* upper half */
    else if(val > a[middle])
        start = middle + 1; /* lower half */
    else
        return(middle);    /* found */
}

/* value not found, return -1 */
return(-1);
}

```

The result of execution of above program is shown below.

```

How many elements ? 9
a[0] = 34
a[1] = 40
a[2] = 42
a[3] = 45
a[4] = 50
a[5] = 65
a[6] = 70
a[7] = 78
a[8] = 86
Enter an element to search : 42
Value 42 is found at position 3

```

2.5 HASH TABLES

The linear and binary searching both are easy to understand and implement, but their searching time is dependent on size of the array or table i.e. number of entries. For linear searching the search time is in order of n and binary search it is $\log_2 n$, where n is number of entries in table. The more complex but faster searching techniques are based on the hash tables in which search time is independent of number of entries in a table and known as hash table methods. The basic principle of the hash table methods is to map the key to a position into a table i.e. address using

functions called hashing functions. The key is the field into a table using which records in a table are searched. For example, in a table of employees, employee number is the key. Hashing function directly maps the key using certain mathematical operations to a table position where record is stored and later can be accessed by again calculating position from a key. The range of key values forms the key space while range of table positions forms the address space. Usually, address space is smaller than the key space and hence multiple keys are mapped to same position which results into collisions as we can not store more than one record at same position. To solve this problem, collision resolution techniques are used. This section introduces the concepts of hashing functions with some commonly known hashing functions. Then, two commonly used collision- resolution techniques are introduced.

2.6 HASHING FUNCTIONS

The general form of the hashing function is given as

$$p = h(k)$$

where **k** is the key value and **p** is the address or table position produced as a result of key-to-address transformation using hashing function **h**. Table position (or address) **p** is known as *hash value*. Thus hashing function maps the key to a hash value. The **k** is one of the values from key set **K**, while **p** is one the values from address set **P**. Let us consider the employee table shown in fig. 5.9 storing employee information like employee number (e_no), employee name (e_name) and employee salary (e_salary). Table uses the 4-digit employee number as key. Hence the key range is 1000 to 9999, which results into key set $K = \{1000, 1001, 1002, \dots, 9998, 9999\}$. The hashing function **h** maps the given key value from **K** to a hash value in **P**, where **P** denotes the set of hash values. Assume that following hash function is used to map key **k** to hash value **p**.

$$h(k) = k \text{ mod } 11 + 1$$

If key $k = 1025$, then the hash value $p = 3$ meaning that the record with key value 1025 is stored in table at position 3 as shown in fig. 2.2. Later when this record is to be searched, we can apply the same function to map $k = 1025$ to $p = 3$ and access the record directly. Later if we have to store the record with key 1113, it also

results into position 3 and the collision occurs. The problem of collision is solved using collision-resolution techniques.

Key
↓

	e_no	e_name	e_salary
1			
2			
3	1025	Brijesh	25000
4			

Fig. 2.2 : Employee table with hashing

In employee information, key is numeric and easy to manipulate by mathematical operations. But if the keys are non-numeric like name of persons, then it is first converted into numeric and then hashing function is applied. For example, name can be represented as series of ASCII values to form a number which can be used to map to hash value. This type of conversion before applying hashing function is called *preconditioning*. Let us take the overview of some commonly used hashing functions.

Division function : This is one of the widely used hash function and it is given as

$$h(k) = k \text{ mod } m + 1$$

where **k** is key value and **m** some integer. The mod operation denotes the remainder of dividing k by m. The hash value is remainder plus 1. For example, if employee number is 1025 and divisor m is 11, then hash value is

$$\begin{aligned} h(1025) &= 1025 \text{ mod } 11 + 1 \\ &= 3 \end{aligned}$$

As we have earlier discussed, many keys are mapped to same hash value, resulting into collisions. To reduce the number of collisions, m should be chosen as prime number or number without small divisors. In general, m is a prime number which is equal to size of hash table. For $m = 11$, the size of hash table is 11 as the function calculates the hash values in range 1 to 11 only.

Midsquare function : This is another widely used hash function in which key is multiplied by itself to get its square and the hash value is chosen as appropriate number of digits from the middle of square. For example, if the key is 1025, then its square is 1050625. If three middle digits are chosen as hash value, then for key 1025, the hash value is 506.

Folding function : In this method, a large key is separated into number of parts of equal lengths except last one. Then these parts are added together to form the address or hash value by ignoring last carry. Consider a key 105678 which is divided into three parts 10, 56 and 78. Their addition with last carry ignored is $10+56+78 = 44$. Hence the hash value for key 105678 is 44.

There are many other hash functions available.

2.7 COLLISION-RESOLUTION TECHNIQUES

Hashing function maps more than one key to same table position which is called collision as we can not store another record at same position when it is already occupied by some record. These types of records are called colliding records. The colliding record is then stored some where else in table. The collision-resolution technique finds appropriate empty table position by searching the table to store the colliding record. There are two techniques used for this purpose known as sequential probing and separate chaining. Let us take the overview of both of these techniques.

Sequential probing : This technique also called linear probing, looks for the next available empty table position by sequentially searching the table positions starting from the hash value or table position for which collision has occurred. Considering hash table of size m and the resulting hash value is p (for which collision has occurred), it searches the table positions as follows.

$p+1, p+2, \dots, m - 1, m, 1, 2, \dots, p - 1$

Let us consider our example of employee table. Assume that we have used the division hashing function with $m = 11$ and the table already stores the records as shown in fig. 2.3(a).

	Key	e_no	e_name	e_salary
1				
2				
3		1025
4		1026
5				

(a)

	Key	e_no	e_name	e_salary
1				
2				
3		1025
4		1026
5		1113

(b)

Fig. 2.3 : Sequential probing

The next record to be stored in table is having key 1113. The hash value for $k = 1113$ is $p = 3$. The table position 3 is already occupied by the record with key 1025 resulting into collision. The sequential probing technique then looks for position 4 which is also occupied by record with key 1026. The next position 5 is empty and the colliding record with key $k = 1113$ is stored there as shown in fig. 2.3(b). Later when this record is to be searched, we have to start searching it from its table position equal to its hash value until key matches or the empty record is found. If the search ends with empty record, this record is not available into the table. As we have stored record with key 1113 at table position 5 (not at its position 3), if any record now while storing results into hash value 5 will collide. This is major limitations of the sequential probing which results into increase in search time.

Separate chaining : This technique also called overflow chaining, maintains separate link list for a group of colliding records. This requires an addition pointer field in each record to point to next record in chain. Only first record in chain is stored in original table, while all the colliding records for same position are stored in

separate table called overflow area using linked list. Fig. 2.4 shows the hash table with overflow area for the employee information. Table stores the record with key 1025 (hash value 3 using division function) at table position 3 in original table. The subsequent records with keys 1113, 2147 results into same hash value 3. They are stored in overflow area using linked list. Observe that new colliding record is always stored at front of the list in overflow area.

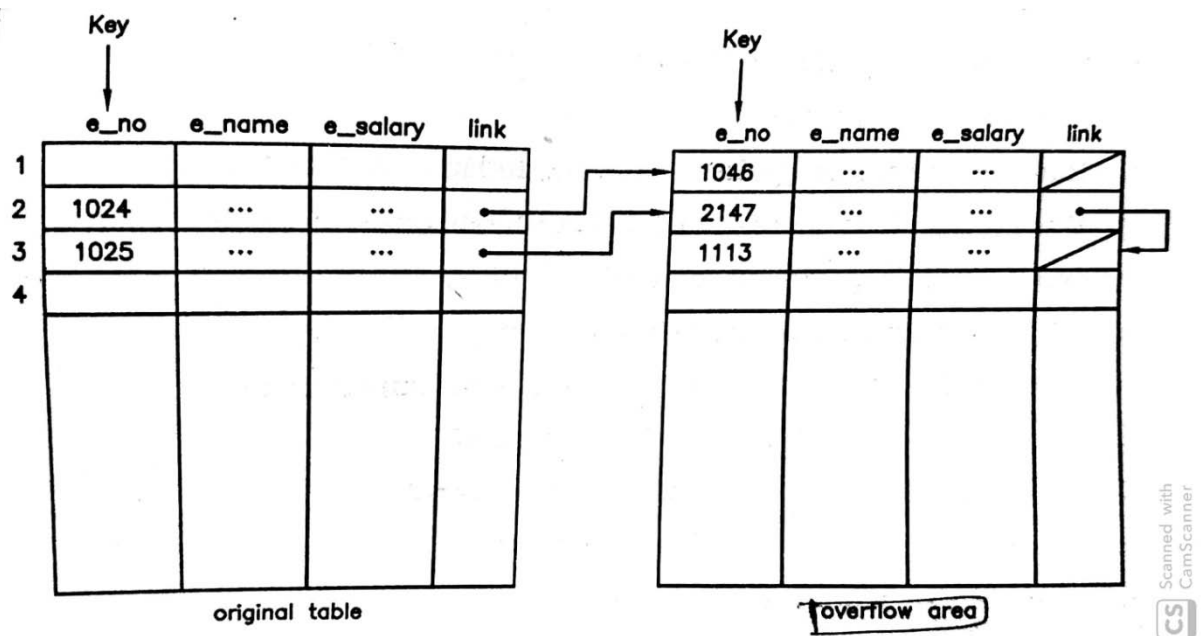


Fig. 2.4 : Overflow chaining

Later when record is to be searched, it is searched in the linked list using its table position. The only drawback of separate chaining is that it needs extra memory for each record to store the pointer.

2.8 LET US SUM UP

Search : Search is a method to find a given element into the set of values either ordered or unordered.

Sequential Search : Sequential search also known as linear search is very simple. It simply sequentially compares the given value with each value in list of values one by one from start to end until value is found or the list is ended. Its time complexity is $O(n)$ where n is the number of elements in the list.

Binary Search : Binary search is more efficient than the sequential search, but it requires the sorted data. It compares the given value with middle value in the list and if value matches then search is complete. Otherwise, if the given value is less than middle then it repeats the same process in upper half of the list else if the given value is greater than middle value then it repeats the same process in lower half of the list. Its time complexity is $O(\log n)$.

Hash Search : The more complex but faster searching techniques are based on the hash tables in which search time is independent of number of entries in a table and known as hash table methods. The basic principle of the hash table methods is to map the key to a position into a table i.e. address using functions called hashing functions.

Hashing function : It is a function which maps a given key to the position into the hash table. There are number of different hash functions available.

Collision : When a hashing function maps more than two keys to a same position into the hash table, it is called collision.

Collision-Resolution : The collision-resolution technique finds appropriate empty table position by searching the table to store the colliding record. There are two techniques used for this purpose known as sequential probing and separate chaining.

Sequential probing : This technique also called linear probing, looks for the next available empty table position by sequentially searching the table positions starting from the hash value or table position for which collision has occurred.

Separate chaining : This technique also called overflow chaining, maintains separate link list for a group of colliding records. This requires an addition pointer field in each record to point to next record in chain.

2.9 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. search compares given value to list of values one by one.
2. search needs data to be sorted.
3. search, search time increase with size of data.

4. Binary search starts comparison from value at position.
5. When two keys result into same hash value, it is called

➤ **True-False**

1. Linear search is faster than binary search.
2. Linear search works for both sorted and unsorted data.
3. Binary search divides list into half every time.
4. Hash table method makes searching independent of table size.
5. Binary search is more complex than hash table methods.

➤ **Answer in brief**

1. What is the advantage of binary search?
2. Give the basic principal of hashing.
3. What is hashing function?
4. What is collision? How does it occur?
5. What is sequential probing?
6. What is overflow chaining?

➤ **Answer the following**

1. Explain the linear search with its algorithm.
2. Give and explain the binary search method.
3. What hashing? Explain its with example.
4. What is hashing function? Discuss the various hashing functions.
5. When collision does occur? What are the solutions to resolve the collision?
6. Write short notes.
 1. Sequential search
 2. Binary Search
 3. Hashing
 4. Hash functions
 5. Collision resolution techniques

2.10 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill

2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

2.11 ASSIGNMENT

1. Assume that a system stores the transactions of a retail store including item number, item name, and price. Write a program to search a given item when item number is given using linear search.
2. Develop a program which accepts the names of the persons in alphabetical order. Use the binary search to verify whether given name is present in list or not.
3. Implement division hashing function so that given a key value, it returns a hash value. Use it in main() to demonstrate its use.

Unit Structure

- 3.1 Learning Objectives
- 3.2 Introduction
- 3.3 Trees
- 3.4 Binary Trees
- 3.5 Sequential Representation
- 3.6 Linked Representation
- 3.7 Operations on Binary Trees
- 3.8 Insertion
- 3.9 Traversals
- 3.10 Search
- 3.11 Deletion
- 3.12 Let us sum up
- 3.13 Check your Progress
- 3.14 Further Reading
- 3.15 Assignment

3.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define a tree and give the structure of binary tree.
- Demonstrate the sequential and linked representations of a tree.
- List the various operations on binary trees.
- Develop the algorithms for various operations like insertion, traversals, search and deletion on binary trees.
- Perform the various operations on binary trees.

3.2 INTRODUCTION

We have studied so far the linear data structures in which relationship among the data items is linear or one-dimensional. However, there are many applications which cannot represent complex relationship among the data items just in linear manner. A non-primitive data structure in which relationship among the data items is non-linear is known as non-linear data structure. The most common and useful example of non-linear data structure is tree. The non-linear data structures like trees are very useful in representing data or information in which relationship among the elements are more complex. Specifically where ever hierarchical relationship is maintained, non-linear data structure like tree is very useful and allow easy manipulation of the information. One of the most known examples of such relationship is directory structure in operating systems.

This unit starts with introduction of the trees and then covers the mainly binary trees, their sequential and linked representations and finally the various operation on the binary trees like insertion, traversals, search and deletion.

3.3 TREES

Let us define a tree. Students are advised to go through the concepts of graphs given in next unit for better understanding.

A **tree** is defined as acyclic directed graph having

- A root node with indegree 0 and
- All other nodes with indegree 1

A tree is one type of graph and hence at least one node is required to define a tree. That means a single node is a tree having only root node. Fig. 3.1 shows an example of a tree. As shown in fig, top most node is called **root node**, the nodes with outdegree 0 are **leaf nodes** and all other nodes are **intermediate nodes**. Considering level of root node 0, *level* of any node is length of the path from the root node. For example, level of node A is 0, nodes B and C is 1, D, E, F, G and H is 2 and I, J, K and L is 3.

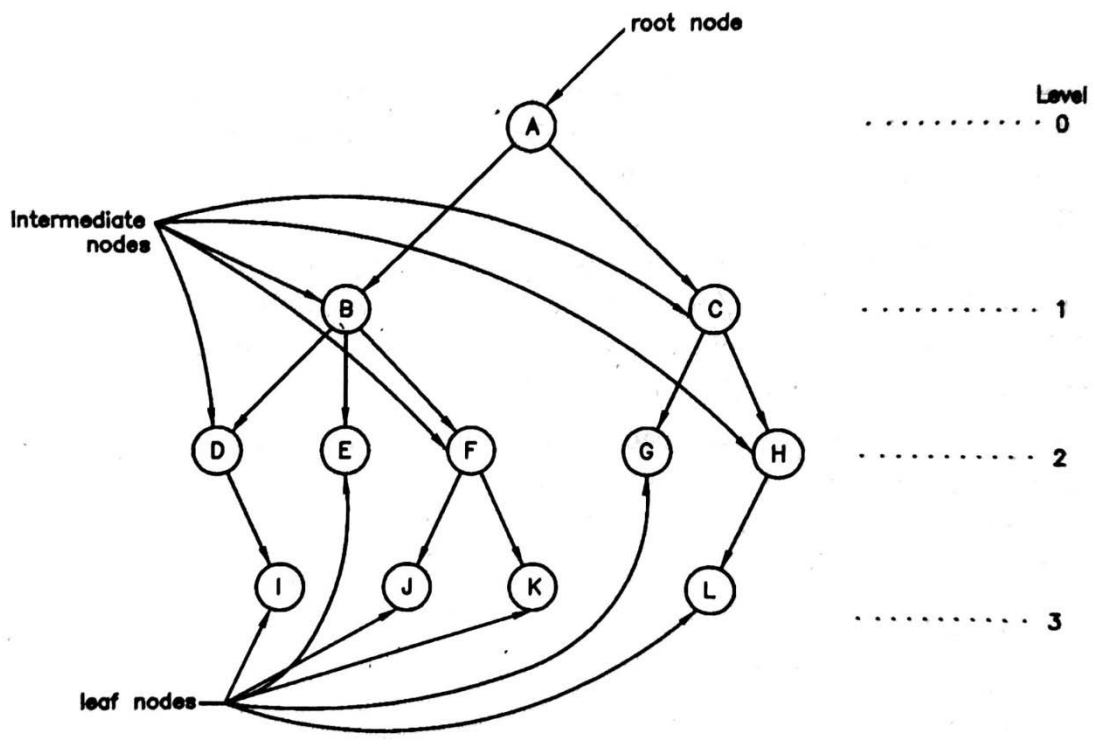


Fig. 3.1 : Example of a tree

Observe that tree is a recursive structure i.e. part of tree is also having same structure as whole tree. Assume that from tree in fig. 3.1, root node A and edges (or links) from A to B and A to C are removed, then we will get two separate **subtrees** as shown in fig. 3.2. These set of separate trees is also referred to as **forest** of trees. The structure of both the subtrees is same as original tree as both have their root nodes, intermediates nodes and leaf nodes connected by links. It results into two subtrees as A has two children, one with root node B and second with root node C.

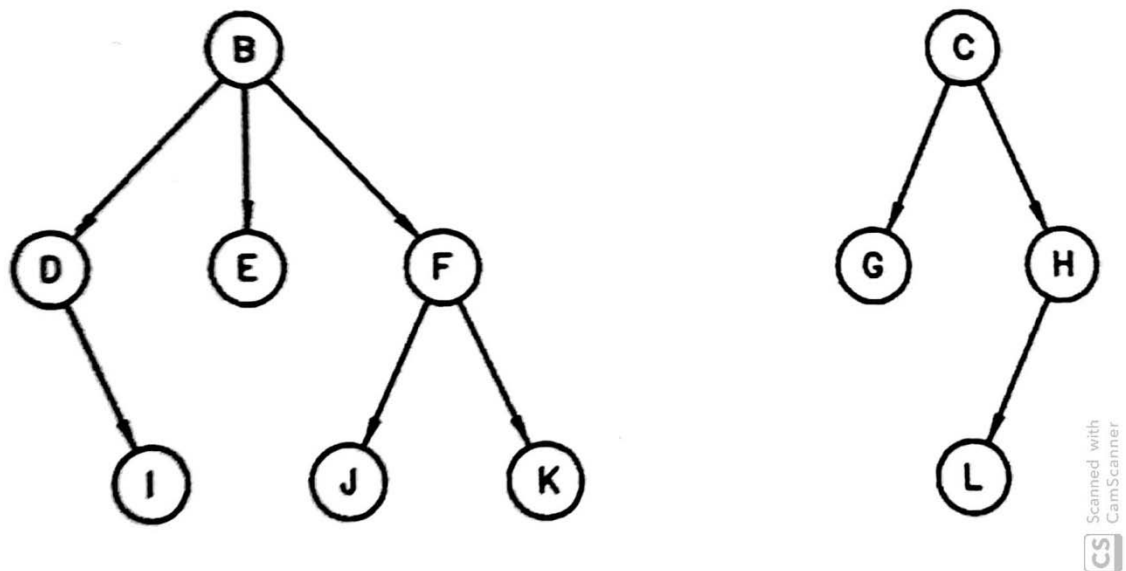


Fig. 3.2 : Subtrees of a tree in fig. 3.1

Similarly, B has three subtrees each with root node D, E and F respectively. Node C has two subtrees with root nodes G and H. We can repeat this, until you find subtrees having only root node. This reveals an important fact that each node in a tree is root node of some subtree. The number of subtrees for any node is equal to the outdegree of a node. For example, node A has two subtrees as its outdegree is 2, while B has three subtrees as its outdegree is 3.

Another important concept is parent-child relationship between nodes. In above example, A is parent of B and C as both of them are directly accessible from A. On other way, B and C are children of A. That means for two nodes A and B, if node B is directly accessible (by traversing a single edge or link) from A, then B is **child** of A and A is **parent** of B.

Trees are categorized based on maximum number of children allowed at each node or we can say maximum outdegree of each node. If in a tree at each node, number of children (or outdegree) is 0 to m then it is called **m-ary** or **m-way** tree. Tree in fig. 3.1 is an example of 3-ary ($m = 3$) tree as B has maximum three children. Further, if we restrict the number of children at each node in m-ary tree to either 0 or m and number of nodes at each level in a tree to m^L (where L is level number of node with $L = 0$ for root node), then tree is called **complete m-ary** tree. Fig.

3.3 shows complete 3-ary tree. When $m = 2$, tree is 2-ary tree which popularly known as binary tree. We will study binary trees and their operations in subsequent sections.

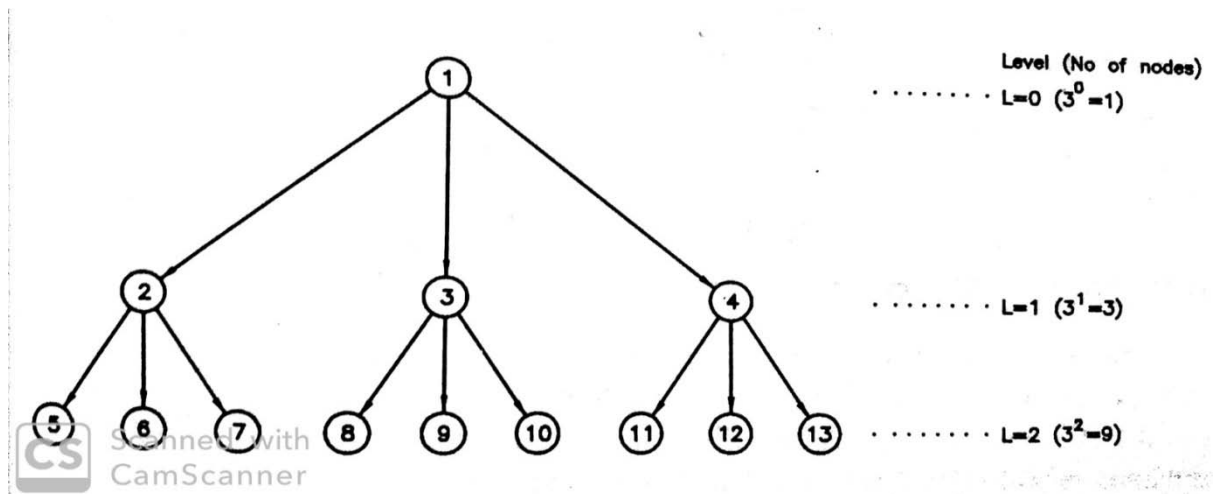


Fig. 3.3 : Complete 3-ary tree

Trees are very useful to represent hierarchical information. One of the best example of tree structure is the directory structure of operating systems. The top of the directory structure is denoted by root directory '/' which forms the root node of a tree. The root directory contains files and subdirectories. Each subdirectory again contains files and subdirectories. This can be expanded to any level. All the files are same as leaf nodes while subdirectories forms the intermediate nodes which further can be expanded. Fig. 3.4 shows the typical directory structure of an operating system.

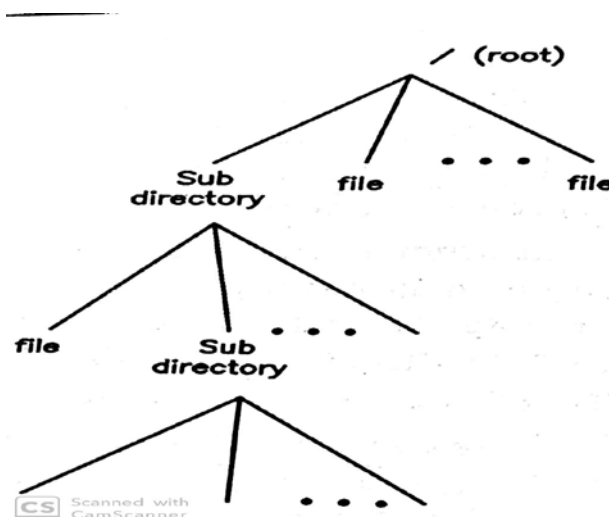


Fig. 3.4 : Directory structure

3.4 BINARY TREES

Binary tree is defined as m-ary tree with $m = 2$. That means number of children (or subtrees) for any node in a binary tree is less than or equal to 2. Fig. 3.5 shows various binary trees. As any node in binary tree has maximum two children, we will refer them as *left child* and *right child*. For example, in fig. 3.5(a) B is left child of A and C is right child of A. For node B, D is left child and right child is not present. In fig. 3.5(c) node B has no left child while D is its right child.

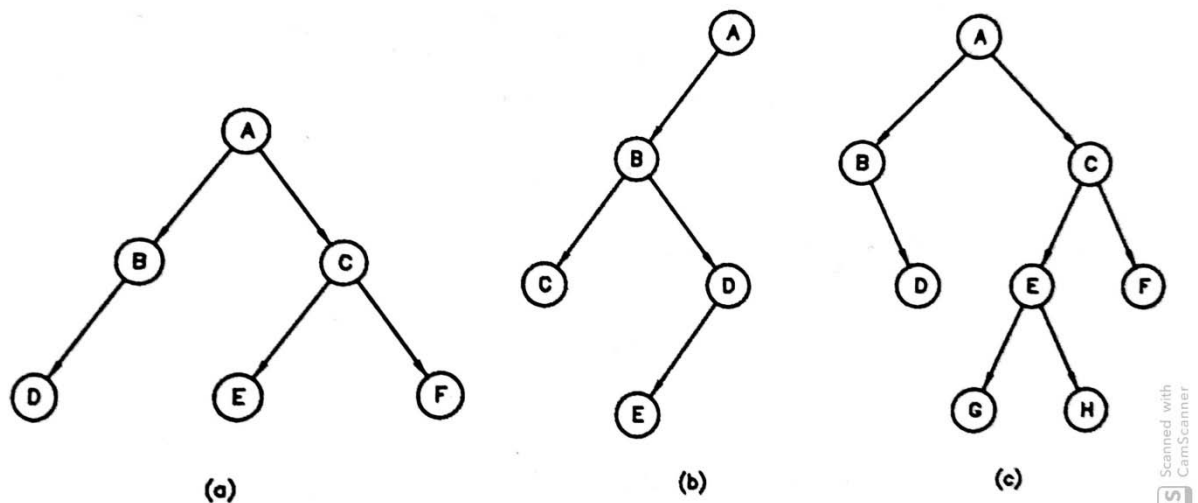


Fig. 3.5 : Various binary trees

If number of children at each node is either 0 or 2 and number of nodes at each level is 2^L (with $L = 0$ for root) then it is complete binary tree. Fig. 3.6 shows complete binary tree with total number of levels 4 i.e. 0 to 3. The number of nodes at each level is power of 2. Observe that both conditions to define complete binary tree together results into situation that all the leaf nodes have 0 child, while all other nodes including root node have 2 children. Observe that total number of nodes in fig. 3.6 is 15 which is $2^4 - 1$ where 4 is total number of levels. In general, total number of nodes in a complete binary tree is $2^{\text{no. of levels}} - 1$.

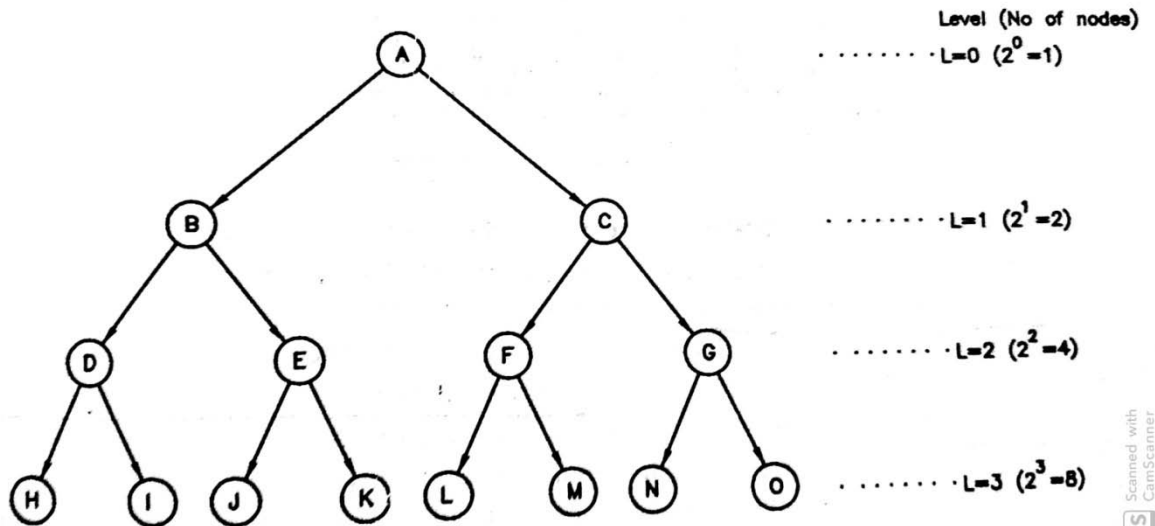
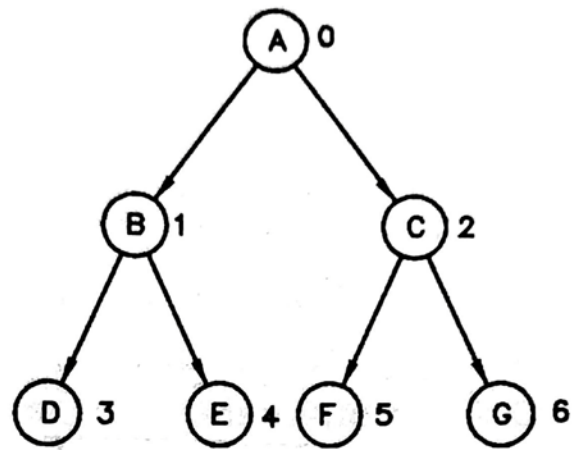


Fig. 3.6 : Complete binary tree with total levels = 4

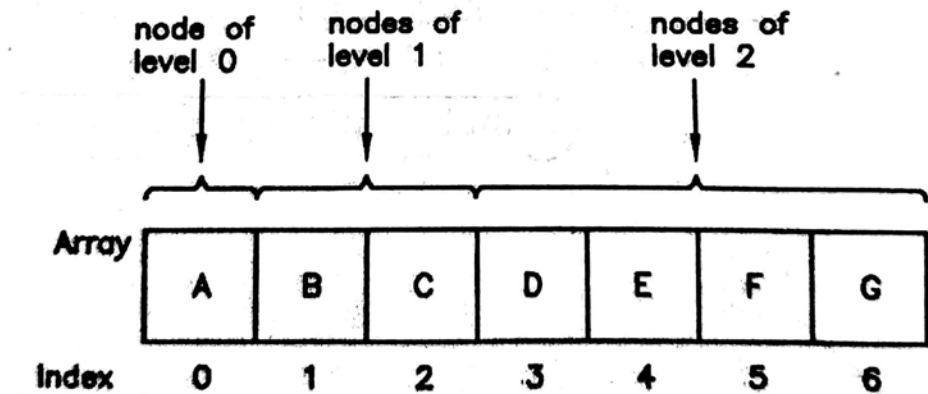
The most important issue now is how to store binary tree in computer memory i.e. its storage representation. There are two possible representations: sequential and linked representation. The sequential representation uses the arrays to store the information of nodes in binary trees while linked representation assigns memory dynamically and uses pointers to maintain the relationships. Let us understand both the representations with their merits and demerits with appropriate examples.

3.5 SEQUENTIAL REPRESENTATION

Arrays are used to store binary trees in sequential representation as array is always assigned contiguous memory. Let us first consider the storage representation of complete binary tree as it always contain all the nodes at each level which allows storing node data sequentially in a array level wise from top to bottom and left to right at each level. Fig. 3.7(a) shows a complete binary tree with 3 levels. A number at node shows its sequential position. Total number of nodes in a tree are 7 ($2^3 - 1$). The sequential storage structure-using array of size 7 is shown in fig. 3.7(b). First node in array is A as it is root node (level 0), next two nodes are B and C which are the nodes at level 1 and last four nodes D, E, F and G are nodes at level 2.



(a) Complete binary tree



(b) Sequential representation

Fig. 3.7 : Sequential representation of complete binary tree

The important question is once a binary tree is stored in array, how can we identify parent of a given node and vice versa. For this purpose, we have to use the array index. As we are using C arrays in our case index of first element is 0. Using array index with simple arithmetic operations we can identify parent child relationship. To find the parent of a node at I th position (index) following operations are used.

$$\begin{aligned}
 \text{Parent}(I) &= I/2 && \text{if } I \text{ is odd} \\
 &= I/2 - 1 && \text{if } I \text{ is even}
 \end{aligned}$$

The operation $\dots\dots\dots$ denotes truncated division in which only integer part is kept while fractional part is removed.

Example : Find the parent of nodes B and C for a tree in fig. 3.7

The index of node B is 1, the index of its parent is

$$\text{Parent}(1) = 1/2 = 0$$

As node at index 0 is A, parent of B is A. Similarly, for node C (index of C is 2)

$$\text{Parent}(2) = 2/2 - 1 = 0$$

Which also node A. Thus A is parent of nodes at index 1 and 2 which are nodes B and C respectively. ■

To find the children of node at *l*th position (index), following operations are used.

$$\begin{aligned}\text{Child}(l) &= 2 * l + 1 \quad \text{for left child} \\ &= 2 * l + 2 \quad \text{for right child}\end{aligned}$$

Example : Find the left and right child of the node C in a tree in fig. 4.11

The index of node C is 2, so the index of its left child is

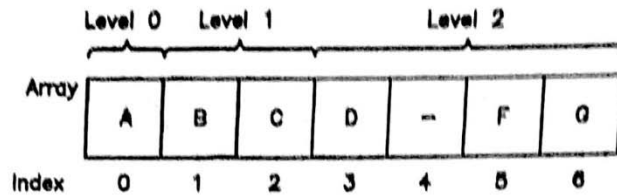
$$\text{Child}(2) = 2 * 2 + 1 = 5$$

The node with index 5 is F and hence F is the left child of C. The index of its right child is

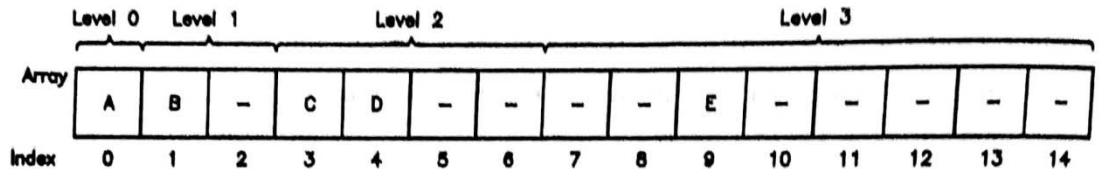
$$\text{Child}(2) = 2 * 2 + 2 = 6$$

which is the index of node G. Thus, node G is the right child of node C. ■

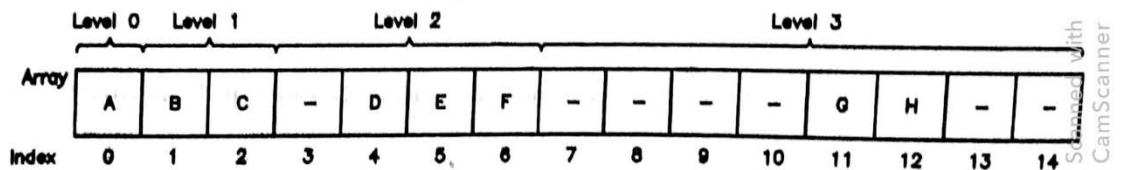
In a normal binary tree, it is not necessary that all the nodes at every level are present which means many nodes may not have its left child or right child or both. In such case how can we sequentially represent a binary tree? To represent binary tree sequentially, we will use array as like in complete binary tree with array locations corresponding to nodes, which are absent, are left blank. This will allow us to use the same operations defined for finding parent and child in complete binary tree without any modification for normal binary tree also. Fig. 3.8 shows the sequential representation of all the binary trees given in fig. 3.5.



(a) Sequential representation of binary tree in fig 4.9(a)



(b) Representation of binary tree in fig. 4.9(b)



(c) Representation of binary tree in fig 4.9(c)

Fig. 3.8 : Sequential representation of binary trees

For a binary tree in fig. 3.5(a), maximum nodes can be 7 as total levels are 3 ($2^3 - 1$). Nodes at each level 0, 1 and 2 can be 1, 2 and 4 respectively. For first two levels all the nodes are present, but for level 2, second node is absent (node B has no right child), which leaves location with index 4 blank. This is shown in fig. 3.8(a). Fig. 3.8(b) shows the sequential representation of a binary tree in fig. 3.5(b). In this case, you can observe that so many locations are blank as tree has 4 levels (maximum nodes can be 15), but so many nodes are absent (1 at level 1, 2 at level 2 and 7 at level 3). Fig. 3.8(c) shows the sequential representation of a binary tree given in fig. 3.5(c).

As earlier stated to find the parent or child for a node, use the same operations we mentioned in complete binary tree. If result of operation is an index with blank location indicates that the node is absent. For example, in fig 4.12(a), if you try to find right child of node B whose index is: $Child(1) = 2 * 1 + 2 = 4$. The location at index 4 is blank which indicates B does not have right child.

The sequential representation is easier to understand and we know how to work with arrays, but it has two major drawbacks.

1. Array is a static structure i.e. it has fixed size. Trees are normally representing dynamic information which changes frequently cause us to perform insertion and deletion of nodes. Insertions and deletions in array are very time consuming as lots of data values are needed to be shifted. Another major problem is that as array is assigned memory statically at compile time, once you reach to maximum limit, more nodes can not be inserted.
2. While representing normal binary trees, lots of space is wasted as, we have to leave blank locations for absent nodes. See in fig. 4.12(b), only 5 locations are used out of 15. As the number levels increases in a binary tree, wastage becomes more and more.

To overcome these problems, we have to use linked representation, which will solve both of the problems as it uses linked structure insertions and deletions are performed easily by changing the links i.e. manipulating pointers, and memory is assigned dynamically as and when needed so it does not waste the space.

3.6 LINKED REPRESENTATION

In linked storage representation of a binary tree, each node stores

- Data or information regarding that node
- Address of its left child which root node of its left subtree. Simply referred as pointer to left subtree. If left child (left subtree) is not present, then it is NULL.
- Address of its right child which root node of its right subtree. Simply referred as pointer to right subtree. If right child (right subtree) is not present, then it is NULL.

Node structure for a node in a binary tree is shown in fig. 3.9. It is same as node structure of doubly linked list. Doubly linked list was linear data structure so two pointers left and right were used to point to previous and next nodes in one-

dimensional way where as tree is non-linear data structures where two pointers left and right are used to point to their left and right subtrees in two-dimensional way.



LPTR : Pointer to left subtree
DATA : Data or information of node
RPTR : Pointer to right subtree

Fig. 3.9 : Node structure for a binary tree

The C structure to define this node is as follows.

```
typedef struct T_NODE {  
    struct T_NODE *lptr;           /* Pointer to left subtree */  
    int data;                      /* Data associated with node */  
    struct T_NODE *rptr;         /* Pointer to right subtree */  
} TNODE;
```

We will use above C structure while discussing the algorithms to define the various operations on binary trees.

Considering above node structure, we can create linked storage representation of a whole binary tree. Fig. 3.10(b) shows the linked storage representation of a binary tree given in fig. 3.10(a). Special pointer **R** is a pointer to (address of) root node of a binary tree. Knowing address of root node, we can access whole tree by using left and right pointers. If **R** is NULL, tree is empty i.e. having no nodes. NULL in left or right pointer indicates absence of corresponding subtree. Note that for leaf nodes both LPTR and RPTR are NULL.

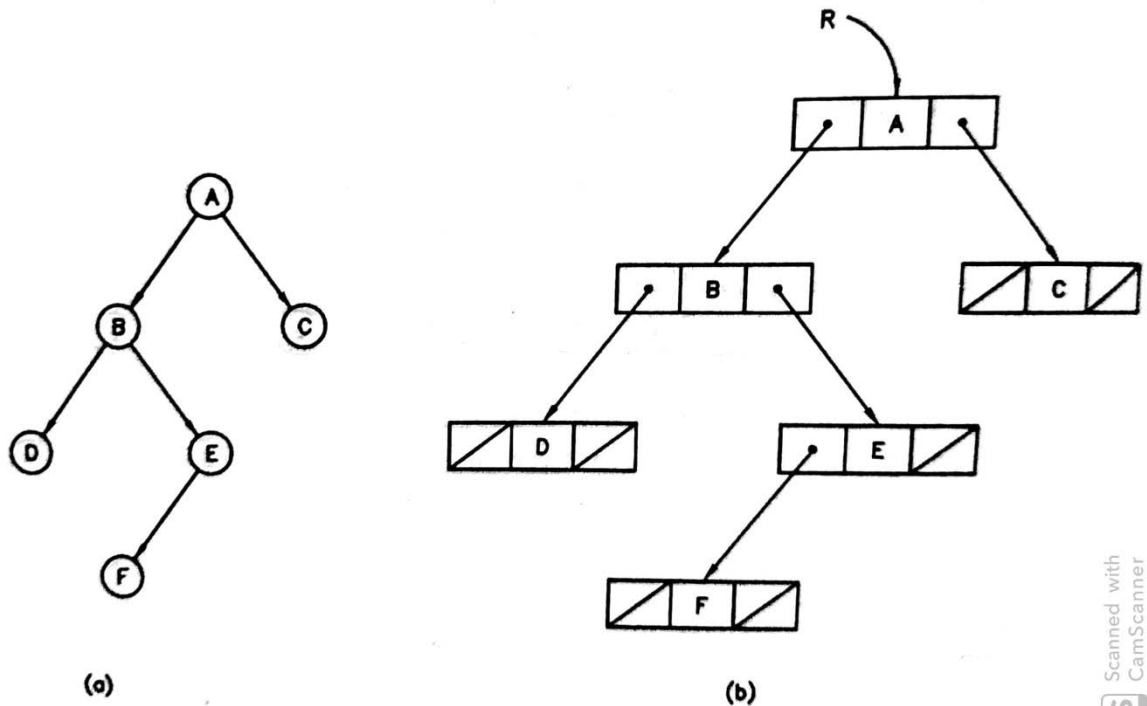


Fig. 3.10 : Linked storage representation of a binary tree

After learning storage representation of binary trees, important issue is how to perform various operations on them. Next section discusses various operations on binary trees. Now onwards we will use only linked storage representation of binary trees through out the chapter as algorithms for all the operations on binary trees are defined using linked storage only.

3.7 OPERATIONS ON BINARY TREES

We can perform variety of operations on binary trees as like other data structures also. This section discusses the important and common operations, which include

- Insertion
- Traversals
 - Preorder
 - Inorder
 - Postorder
- Search
- Deletion

The important issue before we can perform operations on binary trees is to have binary trees which requires create operation to create binary trees. The insertion operation acts as create operation, which allows one new node to be inserted in a binary tree at a time and this way builds a binary tree. Once a binary tree is created, we want to process the information or data stored as nodes of binary tree. This requires visiting or traversing the nodes of a binary tree in particular manner. There are three various traversal operations namely: preorder, inorder and postorder are defined for this purpose. In many occasions, we need to check whether the node given data or information is present in the binary tree. This can be accomplished by search operation. The deletion operation deletes a desired node from a binary tree, as it is no longer required. Lets us now learn all these operations in detail with proper examples.

3.8 INSERTION

Insertion operation is very useful as it creates a binary tree. Before we understand how to perform insertion, we must keep in mind that insertion should follow some order. For example, if we are creating a binary tree of numbers, then all the nodes in left subtree should have numbers less than that of number in root node and all the nodes in right subtree should have numbers greater than that of root node. This is criteria is followed at each subtree. This type of tree is some times referred as *binary search tree*. Here, we assume that duplicate values are node allowed i.e. no two nodes will have same numbers. Fig. 3.11(a) shows an example of such a binary tree. Observe that all the values to the left of root node (having value 15) are less than 15 and all the values to the right of root node are greater than 15. This is true equally for any subtree. As an example take a left subtree (having root node with value 12) have all the values on the left are less than 12 and all the values on right are greater than 12.

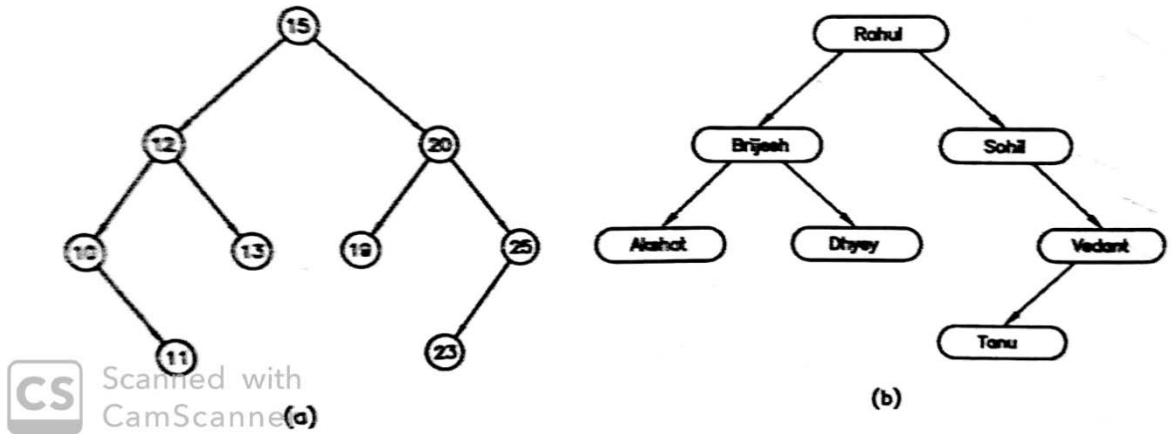


Fig. 3.11 : Binary trees with order (Binary search trees)

Similarly, if a binary tree is to be created with non-numeric information like name of persons then all the names to the left of root node are lexically (dictionary order) less than that of name associated to root node and all the names to the right of root node are lexically greater than that of name associated to root node. Fig. 3.11(b) shows an example of such a binary search tree. Now onwards we will refer binary search tree as simply binary tree.

Let us now discuss how to perform insertion operation to insert the nodes in a binary tree. There are two possibilities, either insertion is made in empty binary tree (if it is first node to be inserted) or insertion in non-empty binary tree i.e. tree already contains some nodes. If the insertion is in non-empty binary tree, care must be taken so that after inserting new node in a binary tree, the order is maintained. The steps to perform the insertion operation are listed below.

1. If binary tree is empty (first insertion), then insert the node as root node and return its address.
2. If binary tree is not empty, then
 - a. If new value to be inserted is less than value of root node and the left subtree of the root node is empty, then insert it as left leaf, otherwise repeat the process with root of the left subtree.
 - b. If new value to be inserted is greater than value of root node and the right subtree of the root node is empty, then insert it as right leaf, otherwise repeat the process with root of the right subtree.

The following examples will clear the concept of insertion operation.

Example : Create a binary tree using following data.

15, 12, 20, 10, 13, 19, 11, 25, 23

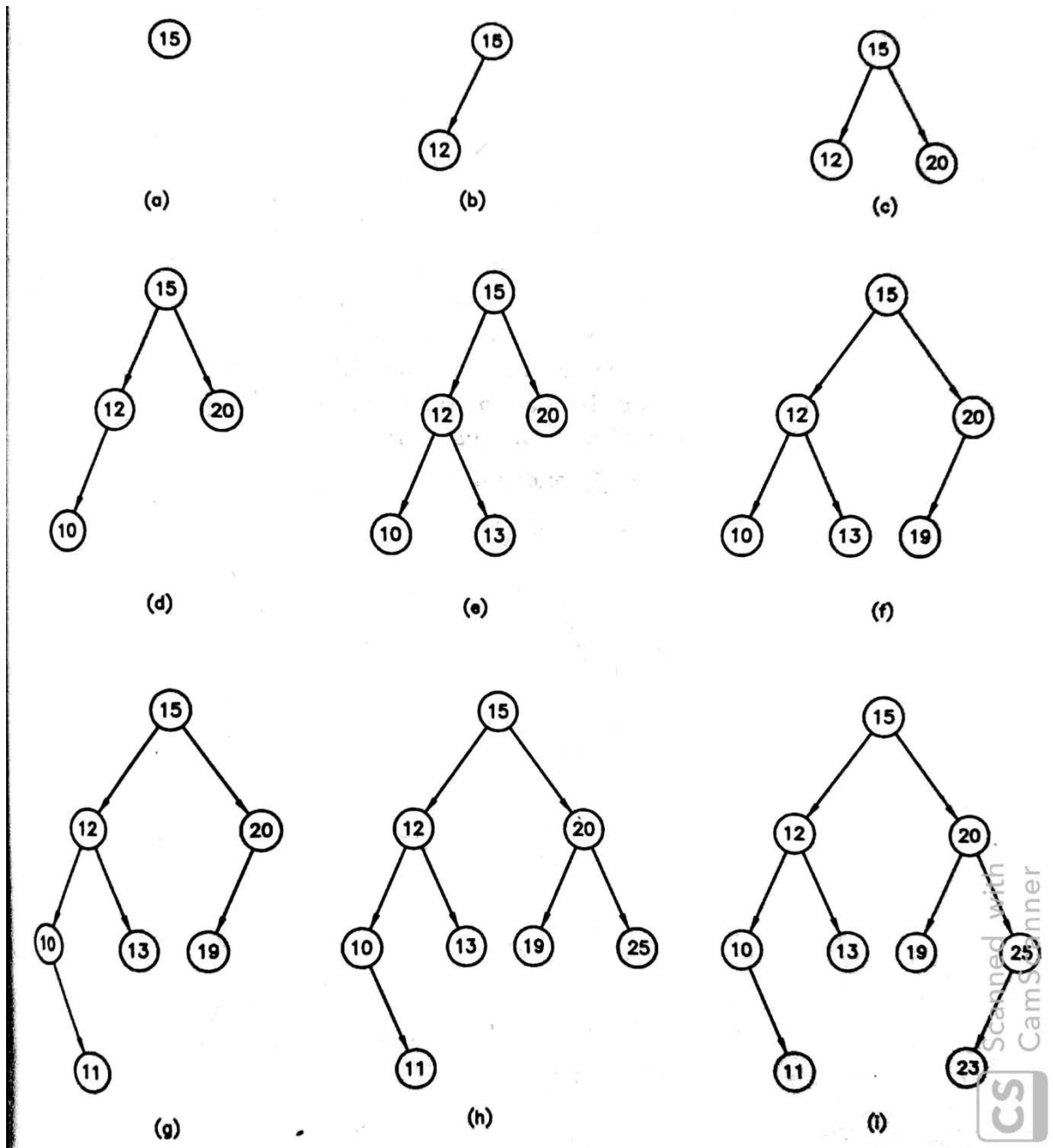


Fig. 3.12 : Creation of a binary tree using insertion

The stepwise building of the binary is shown in fig. 3.12. A new node to be inserted is shown with thick round. Initially tree is empty so the first value 15 will be inserted as root node (fig. 3.12 (a)). Second value is 12 which is less than root node value 15 and also left subtree of node 15 is empty, so it is inserted as left leaf of the

node with value 15 (fig. 3.12 (b)). Third value is 20 which is greater than 15 and right subtree of node 15 is empty, so it is inserted as right leaf of node 15 (fig 3.12 (c)). Fourth value is 10 which is less than 15, but left leaf is present so the process is repeated with root node of left subtree of node 15 which is 12. The value 10 is less than 12 and left subtree of 12 is empty so it is inserted as left leaf of node 12 (fig. 3.12 (d)). Fifth value is 13 which is less than 15, but left leaf of node 15 is present, so the process is repeated with root node of left subtree of node 15 which is 12. The value 13 is greater than 12 and right subtree of 12 is empty so it is inserted as right leaf of node 12 (fig. 3.12 (e)). Sixth value is 19 which is greater than root node value 15, but its right subtree is present whose root node is 20. The value 19 is less than 20 and left subtree of node 20 is empty so that 19 is inserted as left leaf of the node 20 (fig. 3.12 (f)). Continuing this way inserts 11 as right leaf of node 10 (fig. 3.12 (g)), 25 as right leaf of node 20 (fig. 3.12 (h)) and 23 as left leaf of node 25 (fig. 3.12 (i)).



Example : Create a binary tree of following alphabets.

D, B, E, A, C, F

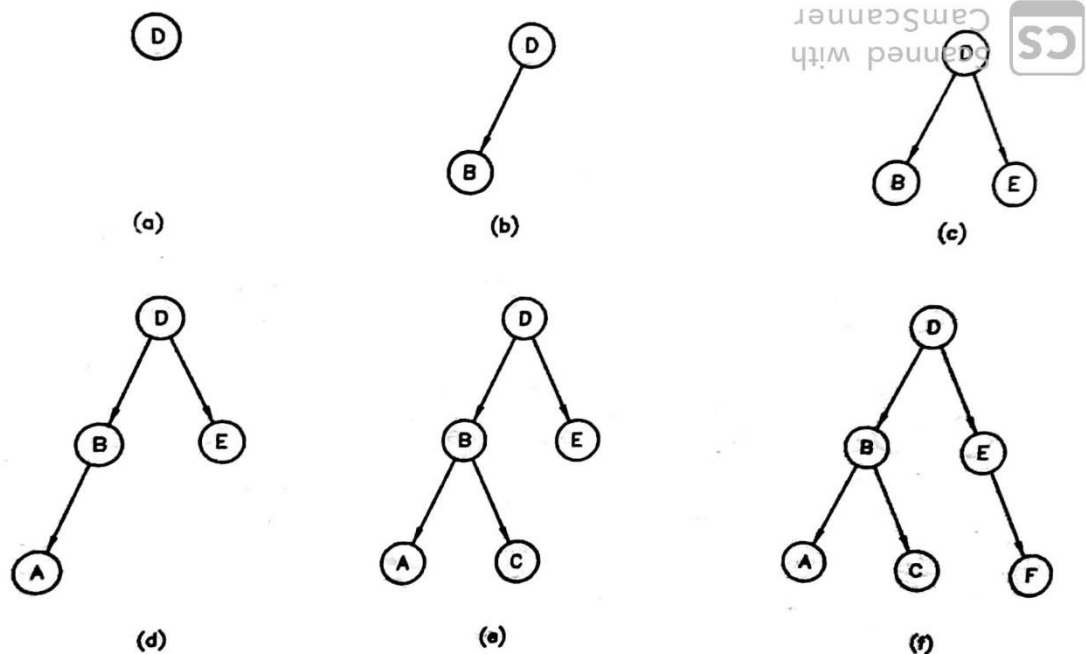


Fig. 3.13 : Binary tree of alphabets

Following the insertion process in similar manner as previous example, the resulting binary tree is shown in fig. 313. The difference is only that instead of numbers, here alphabets are compared. ■

3.9 TRAVERSALS

In order to process the information associated with the nodes, we need to visit the nodes of a binary tree. The traversal is a very common and important operation to visit each node of a binary tree in some particular order. There mainly three orders in which traversal operations are performed. They are

- Preorder
- Inorder
- Postorder

Let us understand each of these traversal operations with suitable examples.

Preorder :

Preorder traversal visits the nodes in following order.

1. Visit the root node.
2. Visit the left subtree in preorder manner.
3. Visit the right subtree in preorder manner.

In steps 2 and 3, we use the recursion as subtrees are also trees and hence the process is repeated in same manner for left and right subtrees. Again, the subtrees of both left and right subtrees are visited recursively. Recursion stops when corresponding subtree is empty.

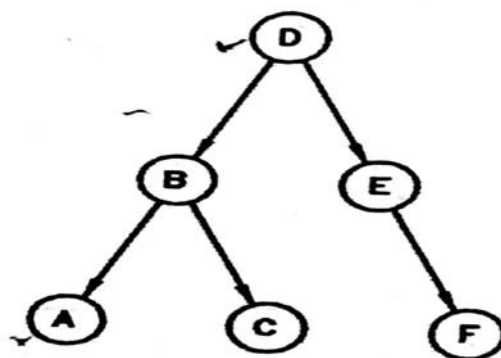


Fig. 3.14 : Binary tree

Consider a tree given in fig. 3.14, its preorder traversal sequence is given as follows.

D B A C E F

For above preorder sequence, the recursive process is traced as below. Whenever either a left subtree or right subtree is found empty, the sequence stops on that branch.

- Visiting root node produces D
- Visiting left subtree in preorder produces B, A and C
 - B is root node
 - A is representing left subtree
 - A is root node
 - Left subtree is empty
 - Right subtree is empty
 - C is representing right subtree
 - C is root node
 - Left subtree is empty
 - Right subtree is empty
- Visiting right subtree in preorder produces E and F
 - E is root node
 - Left subtree is empty
 - F is representing right subtree
 - F is root node
 - Left subtree is empty
 - Right subtree is empty

Inorder :

Preorder traversal visits the nodes in following order.

1. Visit the left subtree in inorder manner.
2. Visit the root node.
3. Visit the right subtree in inorder manner.

In above process, again the binary tree is traversed in recursive manner only. The inorder sequence of the binary tree in fig. 3.14 is as below.

A B C D E F

Observe that the sequence produces the data in ascending order. That means the inorder traversal of a binary search tree (ordered tree) sorts the data into ascending order. This is also some times known as *binary tree sort*.

Postorder :

Preorder traversal visits the nodes in following order.

1. Visit the left subtree in postorder manner.
2. Visit the right subtree in postorder manner.
3. Visit the root node.

In above process, again the binary tree is traversed in recursive manner only. The inorder sequence of the binary tree in fig. 3.14 is as below.

A C B F E D

In above sequence, A, C and B represents left subtree, F and E represents right subtree and finally D is root node. The recursive trace of the postorder traversal can be written in same manner as preorder and inorder with left, right and root as order.

Following C Program gives the C code for the insertion and all of the three traversal methods and demonstrates their use with main() function.

Program 1 :

Write a program to demonstrate insertion and traversal operations on a binary search tree.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <process.h>

typedef struct T_NODE {
    struct T_NODE *lptr;    /* Pointer to left subtree */
    int data;              /* Data associated with node */
    struct T_NODE *rptr;    /* Pointer to right subtree */
} TNODE;

TNODE *insert(TNODE *, int);
void preorder(TNODE *);
```



```

void inorder(TNODE *);
void postorder(TNODE *);

void main()
{
    TNODE *R = NULL;
    int val,choise;

    while(1) {
        clrscr();
        printf("1. Insert\n");
        printf("2. Preorder Traversal\n");
        printf("3. Inorder Traversal\n");
        printf("4. Postorder Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choise : ");
        scanf("%d",&choise);

        switch(choise) {
            case 1: printf("Enter a value to insert : ");
                    scanf("%d",&val);
                    R = insert(R,val);
                    break;
            case 2: preorder(R);
                    break;
            case 3: inorder(R);
                    break;
            case 4: postorder(R);
                    break;
            default: exit(0);
        }

        getch();
    }
}

/* Function to insert the node */
TNODE *insert(TNODE *R, int val)
{
    TNODE *new;

    /* Is tree empty */
    if(R == NULL) {
        new = (TNODE *)malloc(sizeof(TNODE));
        new->data = val;
        new->lptr = new->rptr = NULL;
        return(new);
    }

    /* Insert as left child */

```

```

        if(val < R->data)
            R->lptr = insert(R->lptr,val);

        /* Insert as right child */
        if(val > R->data)
            R->rptr = insert(R->rptr,val);

        /* return */
        return(R);
    }

/* Function to perform preorder traversal */
void preorder(TNODE *R)
{
    /* visit the root node */
    if(R != NULL)
        printf("%d ",R->data);
    else {
        printf("Tree is Empty\n");
        return;
    }

    /* Visit left subtree */
    if(R->lptr != NULL)
        preorder(R->lptr);

    /* Visit right subtree */
    if(R->rptr != NULL)
        preorder(R->rptr);

    /* Return */
    return;
}

/* Function to perform inorder traversal */
void inorder(TNODE *R)
{
    /* Is tree empty? */
    if(R == NULL) {
        printf("Tree is Empty\n");
        return;
    }

    /* Visit left subtree */
    if(R->lptr != NULL)
        inorder(R->lptr);

    /* Visit root node */
    printf("%d ",R->data);
}

```

```

        /* Visit right subtree */
        if(R->rptr != NULL)
            inorder(R->rptr);

        /* Return */
        return;
    }

/* Function to perform postorder traversal */
void postorder(TNODE *R)
{
    /* Is tree empty? */
    if(R == NULL) {
        printf("Tree is Empty\n");
        return;
    }

    /* Visit left subtree */
    if(R->lptr != NULL)
        postorder(R->lptr);

    /* Visit right subtree */
    if(R->rptr != NULL)
        postorder(R->rptr);

    /* Visit root node */
    printf("%d ",R->data);

    /* Return */
    return;
}

```

3.10 SEARCH

Many times, we may require to find whether a node with desired value is present in a binary tree or not. This can be accomplished by performing search operation. Search operation becomes easy as binary (search) tree stores values in order with all the values less than root on left and all the values greater than root on right. To perform the search operation, first the given value is compared with root node value and if it matches the root node value then node is found. Otherwise if value is less than root node value, branch to left subtree and repeat the process with root node of left subtree, else (if it greater than root node value) branch to the right subtree and repeat the process with root node of right subtree. The search

completes when either node is found or subtree to branch is empty (in this case not is not available in binary tree).

Let us consider the following binary tree having values inorder to understand the search operation.

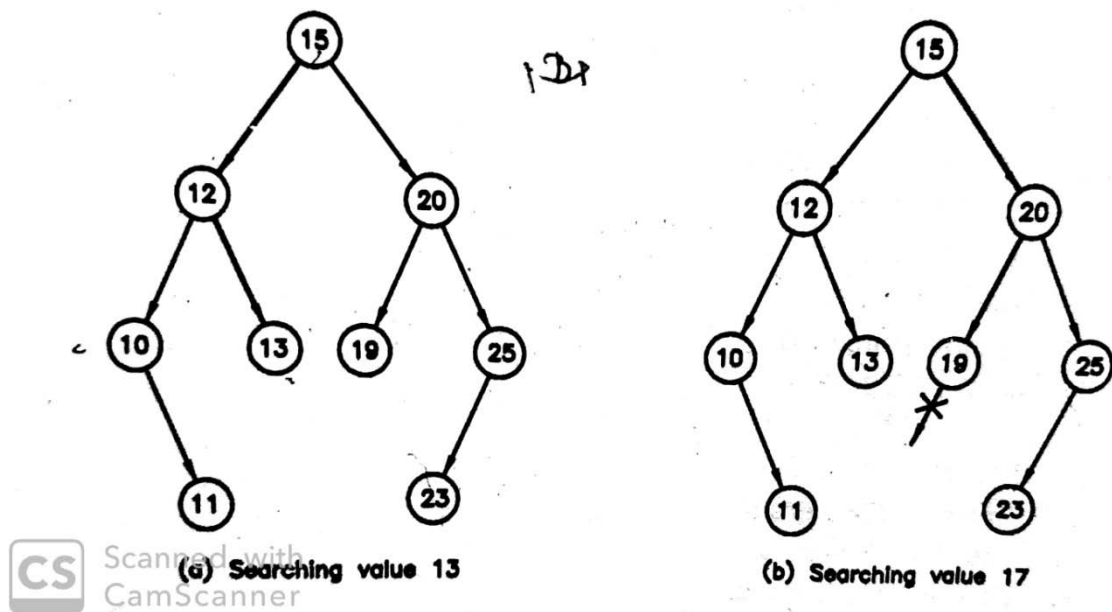


Fig. 3.15 : Search operation on binary tree

Let us first we want to search 13. The root not value 15 does not match to 13, but 13 is less than 15, so we branch to left. The root node of left subtree is 12 which does not match with 13, but 13 is greater than 12, so we branch to right. The root node of right subtree is 13 that match to our value 13 and hence node is found and we can return the address of this node. It is shown in fig. 3.15(a) with search path with thicker line. As a second example, let us search for value 17. The search path is shown in fig. 3.15(b) with last node on path is 19. Now 17 is less than 19, so we need to branch left, but left subtree is empty so the node is declared as not found and return with NULL.

The steps for search algorithm are as follows.

1. Initialize **flag** to false and pointer **ptr** to root node.
2. Repeat following steps while **flag** is false and **ptr** is not NULL
 - a. If **value** matches to value of node pointed by **ptr**, make **flag** true
 - b. Otherwise, if value is less than value of node pointed by **ptr**

then update pointer **ptr** by lptr of **ptr** node

else update pointer **ptr** by rptr of **ptr** node

3. If **flag** is true, then node is found and return pointer **ptr**, else node is not found and return NULL pointer.

The C function to perform search operation is shown below.

```
/* Search operation accepts following arguments
   R      : Address of root node
   val    : Value of the node to be searched
and returns
   Pointer to node if found, otherwise NULL
*/
TNODE *search(TNODE *R, int val)
{
    int flag;
    TNODE *ptr;

    /* Initialization */
    flag = 0;
    ptr = R;

    /* Search the node by comparison */
    while((flag == 0) && (ptr != NULL)) {
        if(val == ptr->data)
            flag = 1;
        else if (val < ptr->data)
            ptr = ptr->lptr;
        else
            ptr = ptr->rptr;
    }

    /* Check the status */
    if(flag == 0) {
        printf("Node not found\n");
        return(NULL);
    }
    else
        return(ptr);
}
```

To call this function use

```
temp = search(R, v);
```

where **R** is the pointer to root node, **v** is the value to search and **temp** is temporary pointer in which address of the node if found is returned.

3.11 DELETION

Deleting a node from binary search tree is one of the most complex operations, as order of the tree should be maintained even after deletion operation. There are three possibilities when a node is to be deleted from a binary tree.

1. Node to be deleted is leaf node i.e. it has no subtrees

This is very simple case and handled easily. Consider a binary tree given in fig. 3.16(a) in which node to be deleted is 32. As it is left leaf node, it is simply deleted by removing left link of its parent node 34. This is accomplished by making left pointer (LPTR) of node 34 NULL. The binary tree after deletion is shown in fig. 3.16(b). Observe that tree remains ordered after deleting the node.

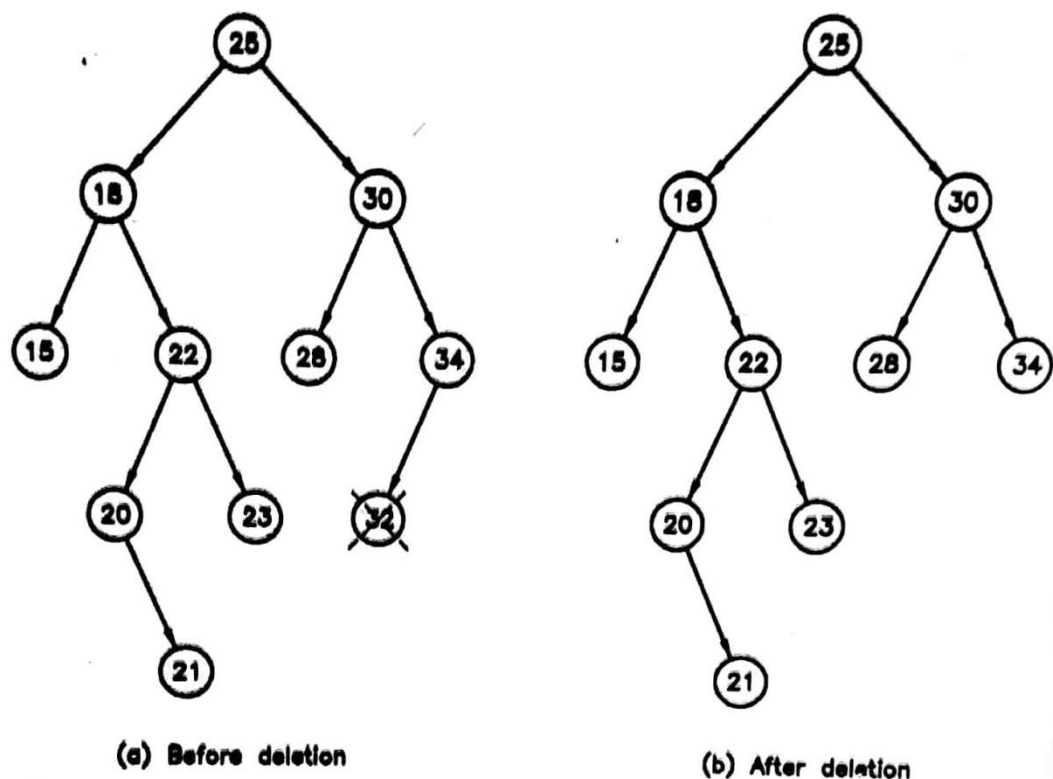


Fig. 3.16 : Deleting a leaf node

Similarly, if node to be deleted is right leaf of its parent, then it is deleted by removing right link of its parent i.e. making right pointer (RPTR) of parent NULL.

2. Node to be deleted has one of its subtrees present.

This is also relatively easy case to handle. Consider a binary tree shown in fig. 3.17(a) in which we want to delete a node with value 34. Its left subtree is non-empty. To delete node 34, we have to append its non-empty left subtree to grand parent node (parent of 34). This is accomplished by copying the address of node 32 in right pointer (RPTR) of node 30 (as node 34 was right child of node 30). Fig. 3.17(b) shows the binary after deletion. Again observe that order of the tree is maintained after deletion too.

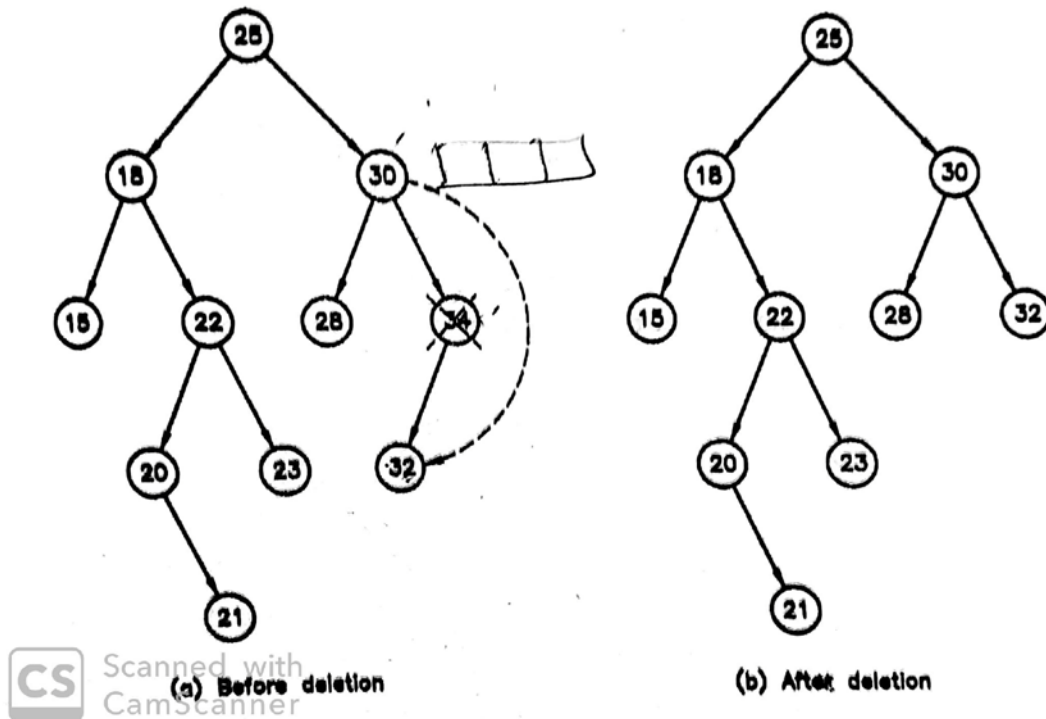


Fig. 3.17 : Deleting a node with one of the subtrees non-empty

Similarly, if node to be deleted is left child of its parent, then one of the non-empty subtrees is to be appended as left link to its grand parent by copying address of non-empty subtree to left pointer (LPTR) of grand parent.

3. Node to be deleted has both subtrees present.

This is the complex case and requires more operations to complete it. As the node to be deleted has both its subtrees present, the question is which node should take its place? At the same time the order should be maintained after the deletion also. Consider a binary tree in fig. 3.18(a) from which node with value 18 (marked with arrow) is to be deleted.

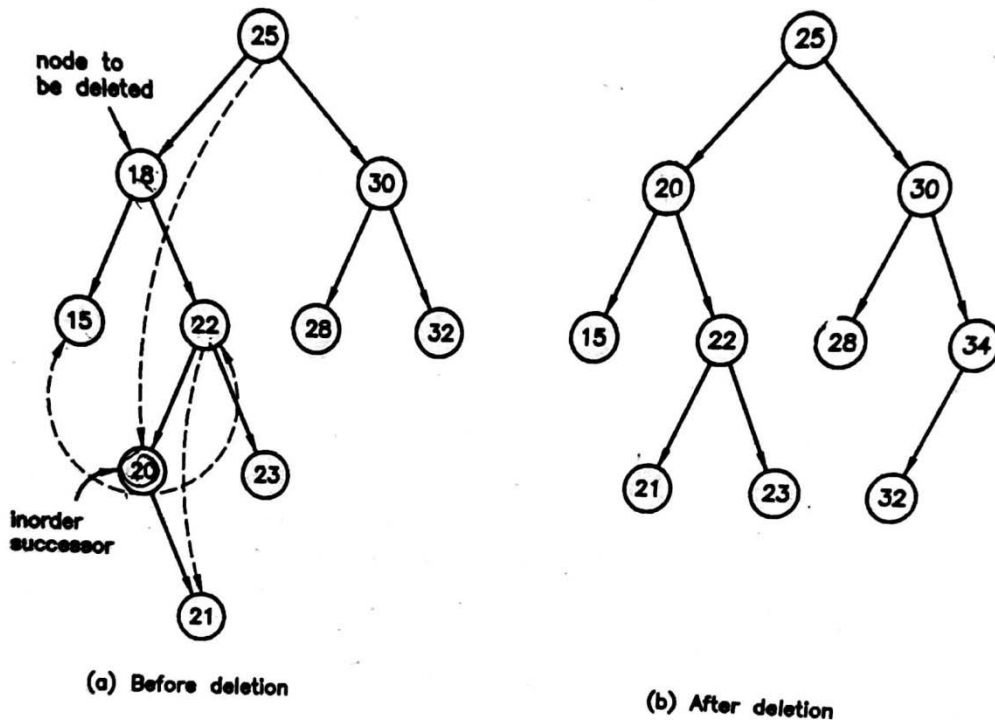


Fig. 3.18 : Deleting a node with both subtrees non-empty

The inorder traversal of the binary tree in fig. 4.22 is

15 18 20 21 22 23 25 28 30 32 34

where if we delete the node 18, the sequence should look like

15 20 21 22 23 25 28 30 32 34

That means above is the inorder traversal of the same binary after deleting node 18 and as sequence is in sorted order, the tree consists of above sequence is ordered binary tree. Observing above sequence tells that node 18 is replaced with node 20 which is the inorder successor of the node 18 in original sequence. A node is called inorder successor of other node if it is immediate next to other node in inorder traversal of a binary tree. This requires that first we need to identify an inorder successor of the node to be deleted. For any node, it is very easy to identify inorder successor, as inorder successor is always the left most node of its right subtree. The node 20 in fig. 3.18(a) is marked as inorder successor of node 18 as node 20 is the left most node in the right subtree of node 18. Node 20 will replace the node 18 means node 20 has to be deleted from its place. If it (inorder successor) has its right subtree non-empty, then it is appended to its grand parent. In our case node 21 (right subtree of node 20) is appended to node 22. Finally, to replace the

node 18 with node 20, the left and right subtrees of node 18 are appended to node 20 as left and right subtrees and node 20 is to be appended as left child of the node 25 (parent of 18) as node 18 is the left child of the node 25. The binary tree after deleting node is shown in fig. 3.18(b). ■

Considering all of three above possibilities and the discussion made for them, the steps to write an algorithm to delete a node from binary search tree is as follows.

1. If node to be deleted is leaf node, simply delete it and return.
2. If node to be deleted has one of the subtree non-empty, then delete it by appending its non-empty subtree to its grand parent node and return.
3. If node to be deleted has both subtrees non-empty, then perform following operations.
 - a. Find the inorder successor of the node to be deleted.
 - b. If inorder successor has non-empty right subtree, then append it to its grand parent.
 - c. Replace node to be deleted with its inorder successor by performing following operations.
 - i. Append left subtree of node as left subtree of inorder successor (Copy LPTR of node to LPTR of its inorder successor).
 - ii. Append right subtree of node as right subtree of inorder successor (Copy RPTR of node to RPTR of its inorder successor).
 - iii. Append inorder successor to parent of node to be deleted. (Copy address of inorder successor to LPTR of parent of node to be deleted if node was left child, otherwise (if right child) copy address of inorder successor to RPTR of parent).

3.12 LET US SUM UP

Tree : A **tree** is defined as acyclic directed graph having a root node with indegree 0 and all other nodes with indegree 1.

M-ary tree : If in a tree at each node, number of children (or outdegree) is 0 to m then it is called **m-ary** or **m-way** tree.

Complete M-ary tree : If the number of children at each node in m-ary tree to either 0 or m and number of nodes at each level in a tree to m^L (where L is level number of node with L = 0 for root node), then tree is called **complete m-ary** tree.

Binary tree : It is defined as m-ary tree with $m = 2$. That means number of children (or subtrees) for any node in a binary tree is less than or equal to 2.

Binary Search Tree : A binary tree with all the nodes in its left subtree having less value than root node and all the nodes in its right subtree having greater value than root node is called binary search tree.

Traversal : Traversal is a way to visit each node of a tree exactly once in predefined order. There are three types of traversals : Preoder, Inorder and Postorder.

3.13 CHECK YOUR PROGRESS

➤ **Filling the blanks**

1. Trees and graphs are data structures.
2. nodes are required at level 4 in complete binary tree.
3. Directed acyclic graph is called
4. node in tree, has no child.
5. A complete binary tree with total of 5 levels need array with size
6. Inorder traversal visits binary tree in _____ , _____ and _____ order.
7. If node to be deleted has both its subtrees present, then it is replaced by its

➤ **True-False**

1. Tree is a recursive structure.
2. A tree can not have cycle.

3. Each node in a tree is root node of some subtree.
4. All the nodes except leaf nodes in complete binary tree have 2 children.
5. Windows Explorer is an example of a tree.
6. Sequential storage does not need space for absent node.
7. All the nodes in left subtree must contain less value than root node in binary tree with order.
8. Inorder successor of a node is always the node which is left most node in its right subtree.

➤ **Answer in Brief**

1. Draw a complete 4-ary tree for 4 levels.
2. What is root node? How does it differ from other nodes?
3. What is m-ary tree? Give an example.
4. How many total nodes are there in binary tree with total 3 levels?
5. Give at least two examples of tree structure.
6. What are the drawbacks of sequential representation of binary trees?
7. Give the node structure for a binary tree for linked storage. Define it using C structure.

➤ **Answer the following**

1. Construct a binary search tree for following data.
50, 45, 42, 78, 65, 86, 34, 40, 70
And perform all the traversals on it.
2. What is sequential storage representation for binary trees? How can you identify parent and child relation?
3. Describe the insertion operation with suitable example.
4. What is traversal? List the various traversal methods and discuss one of them with algorithm.
5. Explain the delete operation in detail with proper examples.
6. Compare the sequential and linked representation of binary tree.
7. Write short notes on
 1. Tree
 2. Binary tree
 3. Sequential representation of a tree

4. Insertion in binary tree
5. Traversal operations
6. Delete operation

3.14 FURTHER READING

1. Introduction to Data Structures: With Applications, Tremblay and Sorenson, McGrawHill
2. Fundamentals of Data Structures in C, Sartaj Sahni, Universities Press
3. Data Structures Using C, Reema Thareja, Oxford
4. Data Structures and Program Design in C, Kruse, Pearson
5. Data Structures with C (Schaum's Outline Series), Seymour Lipschutz, McGrawHill Education
6. Website : <https://www.geeksforgeeks.org/data-structures/>

3.15 ASSIGNMENT

1. Develop a C function to count the number of nodes in a binary tree. Use it in main to demonstrate its use.
2. Develop a C function to create a duplicate copy of a given binary tree. Use it in main to demonstrate its use.
3. Develop a C function, which implements the delete operation on binary search tree. Use it in main to demonstrate its use.
4. Develop a C function to count leaf nodes in a given binary tree and use it in main to demonstrate its use.

Unit 4: Graph

4

Unit Structure

- 4.1 Learning Objectives
- 4.2 Introduction
- 4.3 Concepts of Graphs

4.1 LEARNING OBJECTIVES

After studying this unit student will be able to:

- Define a graph and its concepts.
- Show the matrix representation of a graph.
- Create the adjacency list for a graph and show its representation.
- Develop the algorithm for the breadth first search.
- Develop the algorithm for the depth first search.

4.2 INTRODUCTION

We have studied the tree in last unit and seen that tree is an acyclic graph. Graph is a non-linear data structure which can be used to represent the most complex relationship among the elements of a data. Graph consists of mainly two components: vertices and edges. Railway network is an best example of a graph where each station is a node or vertex and the track between two stations is an edge connecting two nodes. Once the data is represented a graph, we can perform various operations on it to reveal useful information or solve the specific problem.

This unit starts with the basic concepts of the graph and then discusses the matrix and adjacency list representations of the graphs. Then unit covers the two most important algorithms on the graph: Breadth First Search (BFS) and Depth First Search (DFS).

4.3 CONCEPTS OF GRAPHS

Graph is a non-linear data structure, which consists of nodes and edges. Let us consider a simple example of a square shown in fig. 4.1 which consists of four points which are its vertices and four line segments connecting these points or vertices. The vertices of the square here form the nodes and line segments connecting these vertices form the edges.

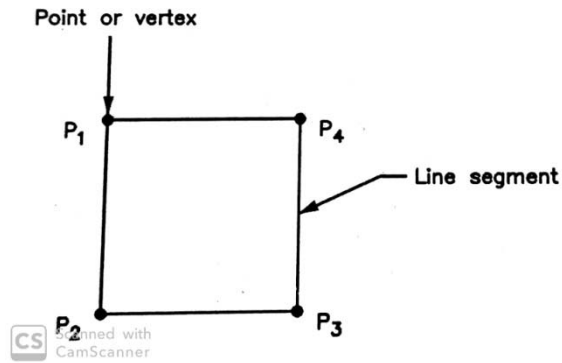


Fig. 4.1 : Square as a graph

Another important example of graph is railway network where each station is a node and the tracks connecting two stations is an edge. Thus, whole railway network is an example of a big graph with number of stations connected by tracks. Fig. 4.2 shows the small part of railway network. Similarly, airline network is also an example of a graph.

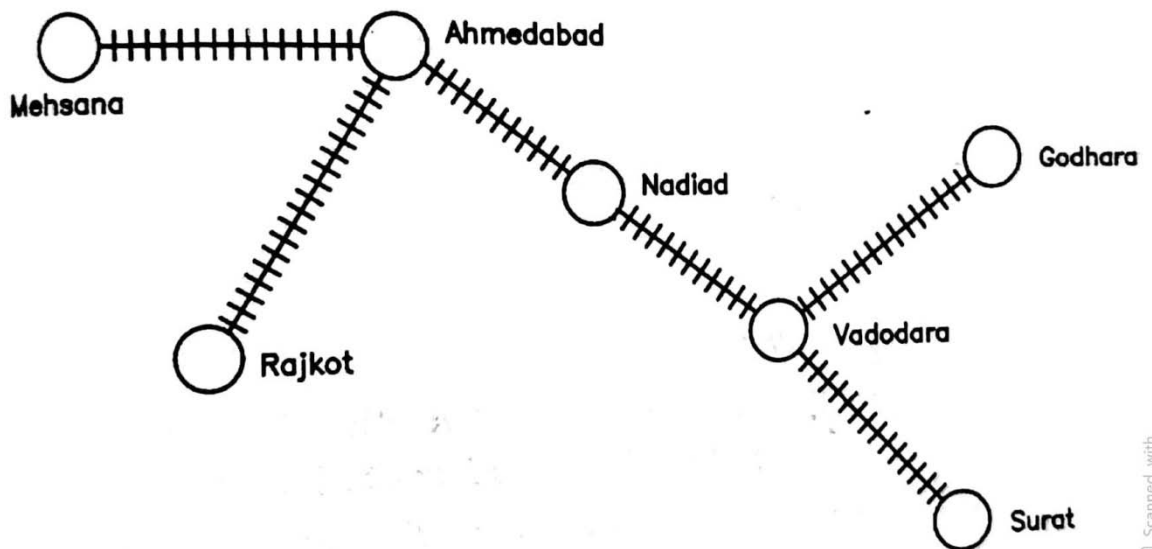


Fig. 4.2 : Railway network

Let us now formally define a graph.

A graph $G = \{V, E\}$ consists of

- Non empty set of nodes or vertices V
- Set of edges E represented as pairs of nodes.

From the above definition, we can say that graph must have at least one node because, set of nodes V can not be empty, but set of edges E can be empty. Fig. 4.3 shows various examples of graphs.

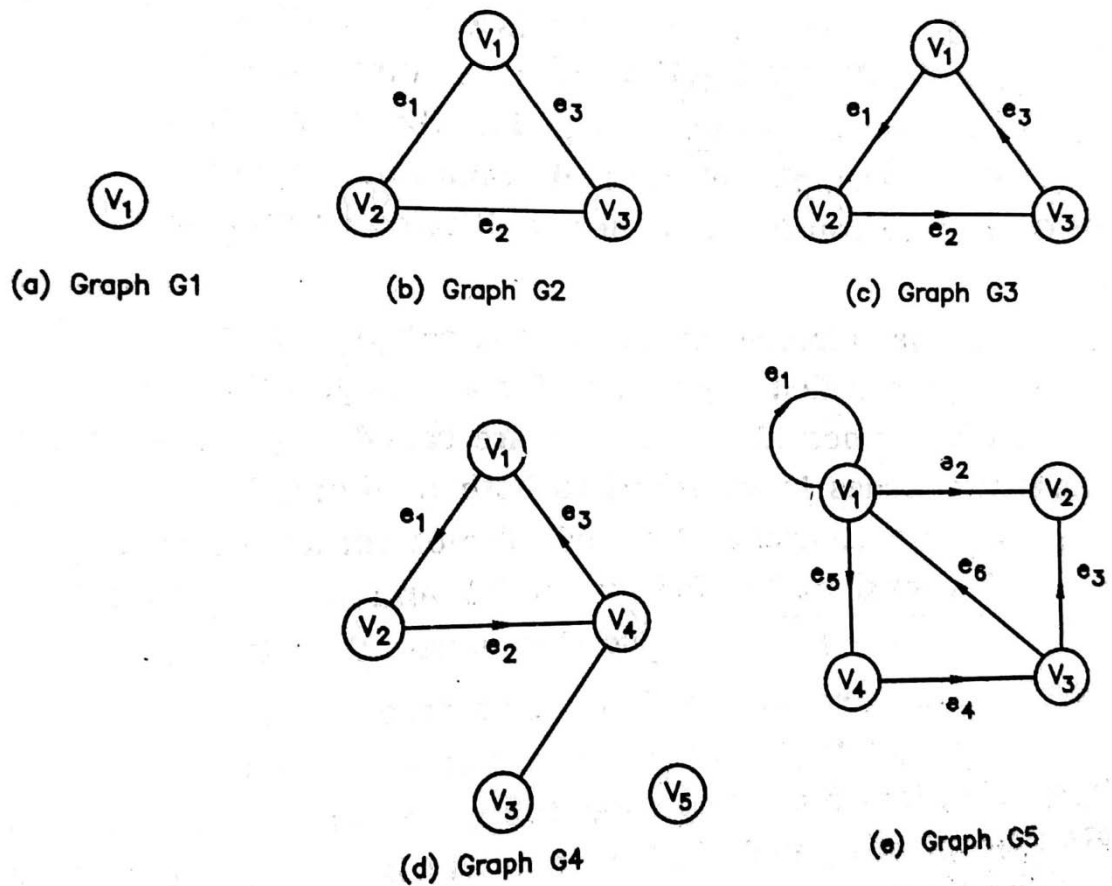


Fig. 4.3 : Examples of graphs

The graph G1 consists of

$$V = \{v_1\} \text{ and}$$

$$E = \{\}$$

while graph G2 consists of

$$V = \{v_1, v_2, v_3\} \text{ and}$$

$$E = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$$

All the edges in graph G2 are having no specific direction. If edge does not have specific direction, then it is called **undirected edge**. For example, edge $e_1=(v_1,v_2)$ in graph G2 is undirected and hence it can be represented also as $e_1=(v_2,v_1)$. That means, in general undirected edge between nodes v_i and v_j is shown either as (v_i,v_j) or (v_j,v_i) . If an edge contains specific direction, then it is called **directed edge**. All the edges in graph G3 are directed where an arrow shows direction. In case of directed edges, it is represented as ordered pair of nodes say (v_i,v_j) and it is not same as (v_j,v_i) . They are two different edges. For example, edge $e_1=(v_1,v_2)$ in graph G3. Thus, graph G3 consists of

$$V = \{v_1, v_2, v_3\} \text{ and}$$

$$E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$$

If a graph contains all the undirected edges, then it is called **undirected graph**. A graph with all the directed edges is called **directed graph** or in short **digraph**. A graph containing some of the edges directed and some of the edges undirected is called **mixed graph**. In fig. 4.3, G2 is an example of undirected graph, G3 and G5 are examples of directed graph and G4 is an example of mixed graph. Graph G1 can be considered either as directed or undirected as it does not contain edges.

If an edge whether directed or undirected $e=(v_i,v_j)$ connects nodes v_i and v_j , then nodes v_i and v_j called adjacent nodes and edge e is incident to v_i and v_j both. In general, two nodes connected by an edge are called **adjacent nodes** and edge connecting these two nodes is **incident** to both of them. For example, nodes v_1 and v_2 in graph G2 are adjacent. If a node is not connected to any other node, then it is called **isolated node**. Node v_1 in G1 and node v_1 in G4 are isolated nodes. If all the nodes in a graph are isolated nodes, then it is called **null graph**.

If an edge e is directed edge and connects ordered pair of nodes (v_i,v_j) , then node v_i is initial or originating node for edge e and is v_j terminal or ending node for edge e . In general, for a directed edge, the node from which edge comes out is called **initial** or **originating** and node in which it ends is called **terminal** or **ending** node. For edge e_1 in graph G3, v_1 is the initial node and v_2 is the terminal node. If an edge originates and terminates in same node, then it is called **loop** or **sling**.

Direction of edge in loop has no meaning. Edge e_1 in graph G_5 is an example of loop.

In a directed graph, number of edges coming to a node (terminating in a node) is called **indegree** of a node and number of edges coming out (originating from a node) is called **outdegree** of a node. The sum of indegree and outdegree is defined as **total degree** of a node. For example, in graph G_5 for node v_3 , indegree = 1, outdegree = 2 and total degree = 3. The degree of loop is considered as 2 (1 for in and 1 for out). Considering this, for node v_1 in graph G_5 , indegree = 2, outdegree = 3 and total degree = 5. For undirected graph, only total degree is defined for any node as number of incidents to a node. In graph G_2 , number of incidents to node v_2 is 2 which is its total degree.

A sequence of edges in a directed graph in which terminal node of an edge is initial node for a next edge in sequence, then it is called **path**. The number of edges in a path is called **length of path**. Following are some examples of paths in graph G_5 of different length.

$$P_1 = \{(v_1, v_4), (v_4, v_3)\}$$

$$P_2 = \{(v_1, v_4), (v_4, v_3), (v_3, v_2)\}$$

$$P_3 = \{(v_1, v_4), (v_4, v_3), (v_3, v_1)\}$$

$$P_4 = \{(v_1, v_1), (v_1, v_4)\}$$

If path starts from and ends in a same node i.e. initial node of first edge and terminal node of last edge in sequence is same, then it is called **cycle**. Path P_3 in above examples is an example of cycle as path starts from v_1 and ends in v_1 . A directed graph with no cycles is called **acyclic** graph.