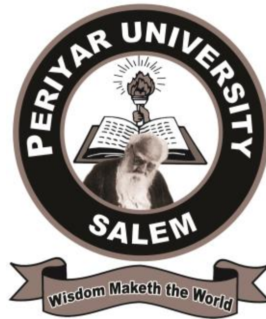


PERIYAR INSTITUTE OF DISTANCE EDUCATION (PRIDE)

PERIYAR UNIVERSITY

SALEM - 636 011.

B.Sc. COMPUTER SCIENCE

FIRST YEAR

PAPER - II : PROGRAMMING LANGUAGE C AND DATA STRUCTURE

Prepared by :

Mrs. M.Kaladevi

Lecturer in Computer Science

Vivekanandha College of Arts and Sciencs for (Women),

Elayampalayam, Tiruchengode,

Namakkal(Dt.).

B.Sc. COMPUTER SCIENCE
FIRST YEAR
PAPER - II : PROGRAMMING LANGUAGE C AND DATA STRUCTURE

Unit – I

Overview Of C: History Of C – Importance Of C – Basic Structure Of C Programs. Constants, Variables and Data Types. Operators and Expression. **Managing Input And Output Operations:** Reading And Writing A Character –Formatted Input And Output. **Decision Making And Branching:** Simple If, If-Else, Nesting Of If-Else, Else-If Ladder, Switch Statements – Goto Statements. **Decision Making And Looping:** While Statement - Do Statement- For Statement.

Unit – II

Structure And **Unions**. **Arrays:** Definition – One Dimensional Arrays – Declaration Of One-Dimensional Arrays – Initialization Of One-Dimensional Arrays – Two-Dimensional Arrays- Initializing Two Dimensional Arrays – Multidimensional Arrays – Dynamic Arrays.

Unit – III

Character Arrays And Strings: Introduction – Declaring And Initializing String Variables – Reading Strings From Terminal – Writing Strings To Screen – String Handling Functions – **Pointers** – **Files** – Opening/Closing Files – Files – Input/Output – Error Handling During I/O Operations – Random Access To Files – Command Line Arguments.

Unit IV

Data Structure: Definition – Categories Of Data Structures. **Arrays:** Array Operations – Merging Of Two Arrays – Two Dimensional Arrays.

Stacks: Definition – Operations On Stack – Representation Of A Stack As An Array – Representation Of A Stack As An Linked List – Evaluation Of Expression: Infix To Prefix Conversion – Infix To Postfix Conversion.

Queues: Definition – Operations On Queue – Representation Of Queue As An Array – Representation Of Queue As A Linked List – Circular Queues.

Linked List: Definition – Operation On Linked List – Circular List – Doubly Linked List – Operations On Doubly Linked List – Polynomial Addition.

Unit V

Trees: Definition & Terminology – Binary Tree – Reversal Of A Binary Tree: In-Order, Pre-Order And Post-Order. Representation Of A Binary Trees In Memory – Linked Representation Of Binary Trees – Array Representation Of Binary Trees – Operations On A Binary Search Tree:

Searching Operation – Insertion Operation And Deletion Operation. **Forest Tree:** Conversion Of Forest Tree To Binary Tree.

Graphs: Definition & Terminology – Graph Representations – **Graph Travels:** Depth First Search & Breadth First Search. Shortest Path Algorithm (Using Dijikstra's Algorithm).

TEXT BOOKS:

1."**Programming In ANSI C" By E. Balagurusamy**
TMH, New Delhi, 2nd Edition

2. "Data Structure through C" Yashavant Kanethkar.

INTRODUCTION

Dear Students

This package consists of five units dealing with concepts of **Programming Language C and Data Structure.** The first unit deals with the fundamental concepts of C programming language and the history of C language.

The second unit deals with the derived data types like structure & union, arrays and functions in details with the necessary sample programs.

The third unit explores the derived data type as pointer. You can discuss about the different pointer concept and its usage. You can also learn about character arrays & string handling functions. In this unit also deals file different file concepts and related file handling functions.

The fourth unit describes the need for studying data structures and also describes the linear data structures like arrays, stacks, queues and linked list in detail with algorithms and sample programs.

The fifth unit, describes the non–linear data structure like trees and graphs in details with algorithms and sample program.

PRIDE would be happy if you make use of this learning material to enrich your knowledge and skills.

UNIT – I

UNIT STRUCTURE

1.1.Introduction
1.2.Objectives
1.3.Overview Of C
  1.3.1. History Of C
  1.3.2. Importance Of C
  1.3.3. Basic Structure Of C Programs
  1.3.4. Self Assessment Questions
1.4.Constants, Variables and Data Types
  1.4.1. Constants
  1.4.2. Variables
  1.4.3. Data Types
  1.4.4. Self Assessment Questions
1.5.Operators and Expression
  1.5.1. Introduction
  1.5.2. Arithmetic Operators
  1.5.3. Relational Operators
  1.5.4. Logical Operators
  1.5.5. Assignment Operators
  1.5.6. Increment And Decrement Operators
  1.5.7. Conditional Operators
  1.5.8. Bit Wise Operators
  1.5.9. Special Operators
  1.5.10. Arithmetic Expressions
  1.5.11. Evaluation Of Expressions
  1.5.12. Precedence of Arithmetic Operators
  1.5.13. Type Conversions In Expressions
  1.5.14. Operator Precedence And Associativity
  1.5.15. Mathematical Functions
  1.5.16. Self Assessment Questions
1.6.Managing Input And Output Operations
  1.6.1. Introduction
  1.6.2. Reading And Writing A Character

1.6.3. Formatted Input And Output
1.6.4. Self Assessment Questions
1.7. Decision Making with Branching and looping
1.7.1. Branching
1.7.1.1. Introduction
1.7.1.2. Decision making with IF statement
1.7.1.3. Simple If Statement
1.7.1.4. The If-Else Statement
1.7.1.5. Nesting Of If-Else
1.7.1.6. The Else-If Ladder Statement
1.7.1.7. The Switch Statements
1.7.1.8. Goto Statements
1.7.2. Looping
1.7.2.1. Introduction
1.7.2.2. The While Statement
1.7.2.3. The Do-While Statement
1.7.2.4. The For Statement.
1.7.2.5. Jumps in Loop
1.7.3. Self Assessment Questions
1.8. Summary
1.9. Unit questions
1.10. Answers for Self Assessment Questions

UNIT – II

UNIT STRUCTURE

2.1. Introduction
2.2. Objectives
2.3. Structure And Unions
    2.3.1. Introduction
    2.3.2. Structure Definition or Template Declaration
    2.3.3. Declaration of Structure Variable
    2.3.4. Giving Values To Members
    2.3.5. Structure Initialization
    2.3.6. Comparison Of Structure Variable
    2.3.7. Arrays Of Structures
    2.3.8. Arrays Within Structures
    2.3.9. Structures Within Structures
    2.3.10. Structures an Functions
    2.3.11. Size Of Structure
    2.3.12. Unions
    2.3.13. Self Assessment Questions
2.4. Arrays
    2.4.1. Definition
    2.4.2. One Dimensional Arrays
    2.4.3. Declaration Of One Dimensional Arrays
    2.4.4. Initialization Of One-Dimensional Arrays
    2.4.5. Two-Dimensional Arrays
    2.4.6. Initializing Two Dimensional Arrays
    2.4.7. Multidimensional Arrays
    2.4.8. Dynamic Arrays
    2.4.9. Self Assessment Questions
2.5. Functions
    2.5.1. Introduction
    2.5.2. Need for user defined functions
    2.5.3. The form of C functions
    2.5.4. Return values and their types
    2.5.5. Calling a function

2.5.6. Category of functions

2.5.7. Nesting of function

2.5.8. Recursion

2.5.9. Functions with arrays

2.5.10. Self Assessment Questions

2.6. Summary

2.7. Unit Questions

2.8. Answers for Self Assessment Questions

UNIT – III

UNIT STRUCTURE

Introduction

Objectives

Character Arrays & Strings

- Introduction
- Declaring And Initializing String Variables
- Reading Strings From Terminal
- Writing Strings To Screen
- String Handling Functions
- Self Assessment Questions

Pointers

- Introduction
- Understanding pointers
- Accessing the address of a variable
- Declaring and initializing pointer
- Accessing a variable through its pointer
- Pointer expressions
- Pointers and array
- Pointers and character strings
- Pointer and functions
- Pointer and structures.
- Self Assessment Questions

Files

- Introduction
- Opening/Closing Files
- Input/Output files
- Error Handling During I/O Operations
- Random Access To Files
- Command Line Arguments
- Self Assessment Questions

Summary

Unit Questions

Answers for Self Assessment Questions

UNIT – IV

UNIT STRUCTURE

4.1.Introduction

4.2.Objectives

4.3.Data Structure

4.3.1. Definition

4.3.2. Categories Of Data Structures

4.3.3. Self Assessment Questions

4.4.Arrays

4.4.1. Introduction

4.4.2. Array Operations

4.4.3. Merging Of Two Arrays

4.4.4. Two Dimensional Arrays.

4.4.5. Self Assessment Questions

4.5.Stacks

4.5.1. Definition

4.5.2. Operations On Stack

4.5.3. Representation Of A Stack As An Array

4.5.4. Representation Of A Stack As An Linked List

4.5.5. Evaluation Of Expression

4.5.5.1.Introduction

4.5.5.2.Infix To Prefix Conversion

4.5.5.3.Infix To Postfix Conversion

4.5.6. Self Assessment Questions
4.6.Queues
4.6.1. Definition
4.6.2. Operations On Queue
4.6.3. Representation Of Queue As An Array
4.6.4. Representation Of Queue As A Linked List
4.6.5. Circular Queues
4.6.6. Self Assessment Questions
4.7.Linked List
4.7.1. Definition
4.7.2. Operation On Singly Linked List
4.7.3. Circular Linked List
4.7.4. Doubly Linked List
4.7.5. Operations On Doubly Linked List
4.7.6. Polynomial Addition
4.7.7. Self Assessment Questions
4.8.Summary
4.9.Unit questions
4.10. Answers for Self Assessment Questions

UNIT – V

UNIT STRUCTURE

5.1 Introduction
5.2 Objectives
5.3 Trees
5.3.1 Definition A & Terminology
5.3.2 Binary Tree
5.3.3 Reversal Of A Binary Tree
5.3.3.1.Introduction
5.3.3.2.In-Order Traversal
5.3.3.3.Pre-Order Traversal
5.3.3.4.Post-Order Traversal
5.3.4 Application Of Binary Tree
5.3.5 Representation Of A Binary Trees In Memory
5.3.5.1.Linked Representation Of Binary Tree

5.3.5.2. Array Representation Of Binary Trees

5.3.6 Operations On A Binary Search Tree

5.3.6.1. Introduction

5.3.6.2. Searching Operation

5.3.6.3. Insertion Operation And Deletion Operation

5.3.7 Forest Tree

5.3.8 Self Assessment Questions

5.4 Graphs

5.4.1 Definition & Terminology

5.4.2 Graph Representations

5.4.3 Graph Traversals

5.4.3.1. Depth First Search

5.4.3.2. Breadth First Search

5.4.4 Shortest Path Algorithm (Using Dijikstra's Algorithm)

5.4.5 Self Assessment Questions

5.5 Summary

5.6 Unit questions

5.7 Answers for Self Assessment Questions

UNIT – I

1.1. Introduction

This unit provides you with *fundamental concepts of C language and its programming structures that* will be used throughout this chapter. It also deals with process of the various data types, *constants, variables, operators, expressions*, and decision making with *branching and looping* in C programming. It assures you to understand general C programming concepts.

1.2. Objectives

After studying this unit, you should be able to:

- ❖ Understand the framework of a C program.
- ❖ Understand the various data types available in C and qualifiers that can be applied to each.
- ❖ Understand to create variables of each type, define constant and the **typedef** statement.
- ❖ Understand and employ the control sequences used in formatted I/O and utilize the various escape sequences provides by C.
- ❖ Understand about operators and expression concepts.
- ❖ Understand about, how to making the decision with branching and looping statement.

1.3. Overview Of C

1.3.1. History Of C

‘**C**’ is one of the most popular programming languages; it was developed by **Dennis Ritchie** at **AT & T’s Bell Laboratories** at **USA** in **1972**. It is an upgraded version of two earlier languages, called ‘**Basic Combined Programming Language’ (BCPL)** and **B**, which were also developed at Bell Laboratories. ‘C’ was developed along with the UNIX operating system. This operating system, which was also developed at Bell Laboratories. C is running under a number of operating systems including MS-DOS.

The root map for all modern computer languages are started through the ALGOL language, in early 1960’s after the COBOL was being used for commercial applications, FORTRAN was being developed for scientific applications. At this stage, people started thinking about the single language, which can perform all possible applications. A committee was formed to develop a new language called **Combined Programming Language (CPL)** at Cambridge University.

It seemed to abstract too general another language called '**Basic Combined Programming Language' (BCPL)** was developed by **Martin Richards** at **Cambridge University**. With some additional features than CPL.

At the same time a language called '**B**' was developed **by Ken Thompson** at **AT & T's Bell Laboratories**. But like BCPL and B turned out to be very specific. Dennis Ritchie developed a language with some additional features of BCPL and B which is very simple, relatively good programming efficiency and good machine efficiency called '**C**' language.

1.3.2. Importance Of C

- 'C' is a general purpose, structured programming language.
- 'C' is powerful, efficient, compact and flexible.
- 'C' is highly portable (i.e. It can be run in different operating systems environments).
- 'C' is robust language whose rich set of built in functions and operators can be used to write any complex program.
- 'C' has the ability to extend itself. We can continuously add our own functions to the existing library functions.
- 'C' is well suited for writing system software as well as application software.
- 'C' program can be run on different operating systems of the different computers with little or no alteration.
- 'C' is a middle level language, i.e. it supports both the low-level language and high level language features.
- 'C' language allows reference to a memory allocation with the help of pointers, which holds the address of the memory location.
- 'C' language allows dynamic memory allocation i.e. a program can request the operating system to allocate or release memory at runtime.
- 'C' language allows manipulation of data at the lowest level i.e. bit level manipulation. This feature is extensively useful in writing system software program.
- 'C' is widely available, commercial 'C' compilers are available on most PC's.
- 'C' program are fast and efficient.
- 'C' has rich set of operators.
- 'C' can be applied in systems programming areas like Compilers, Interpreters and Assemblers etc.

1.3.3. Basic Structure Of C Programs

A ‘C’ program may contain one or more sections given in Fig. 1.3.1.

- **Documentation Section:** It consists a set of comment lines used to specify the name of program, the author and other details etc.
- **Comments:** Comments are very helpful in identifying the program features and underlying logic of the program. The lines begins with ‘/*’ and ending with ‘*’ are known as comment lines. These are not executable, the compiler is ignored anything in between /* and */.
- **Preprocessor Section**: It is used to link system library files, for defining the macros and for defining the conditional inclusion.

  Example:

  #include<stdio.h>, #define A 10, #if def, #endif…etc.
- **Global Declaration Section:** The variables that are used in more than one function throughout the program are called global variables and declared outside of all the function i.e., before **main ( ).**
- Every ‘C’ program must have one main () function, which specify the starting of ‘C’ program. It contains the following two parts.
- **Declaration part:** This part is used to declare all the variables that are used in the executable part of the program and these are called local variables.
- **Executable part:** It contains at least one valid ‘C’ statement. The execution of a program begins with opening brace ‘{‘ and ends with closing brace ‘}’. The closing brace of the main function is logical end of the program.

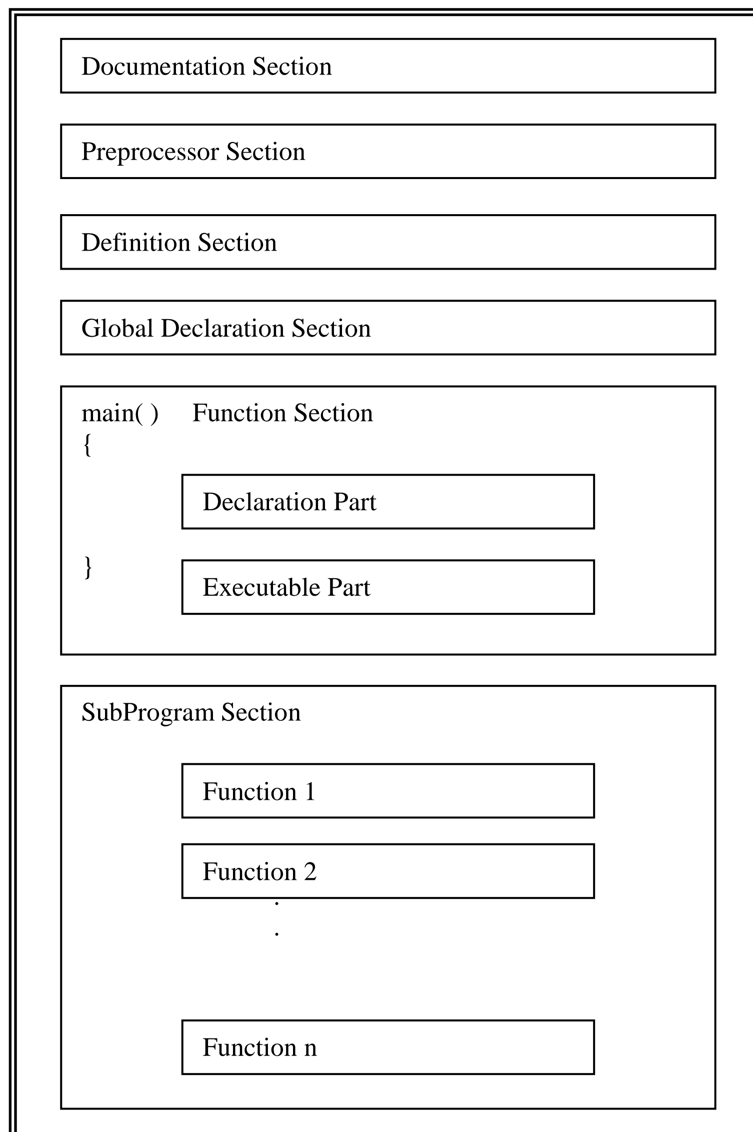


Fig. 1.3.1 Overview of a C program

Programming Rules:

The following rules should follow while writing a 'C' program.

- All statements in 'C' program should be written in lower case letters. Upper case letters are only used for symbolic constants
- Blank spaces may be inserted between the words. It is not used while declaring a variable, keyword, constant and function.
- The program statement can write anywhere between the two braces following the declaration part.
- The user can also write one or more statements in one line separating them with a semicolon (;)

1.3.4. Self Assessment Questions

Fill in the blank

1. C language was developed by ____________________.

2. C language is _______________________programming language.

True / False

1. All statements in 'C' program should be written in upper case letters.

2. In C language, each line of program ends with semicolon.

Multiple Choice

1. C language was developed at AT & T's Bell Laboratories at USA in

a) 1976 b) 1972

c) 1975 d) 1973

Short Answer

1. Define local variable.

1.4. Constants, Variables and Data Types

1.4.1. Constants

Constants in C refer to *fixed values* that do not change during the execution of a program. Types of constant are as shown in Fig.1.4.1.

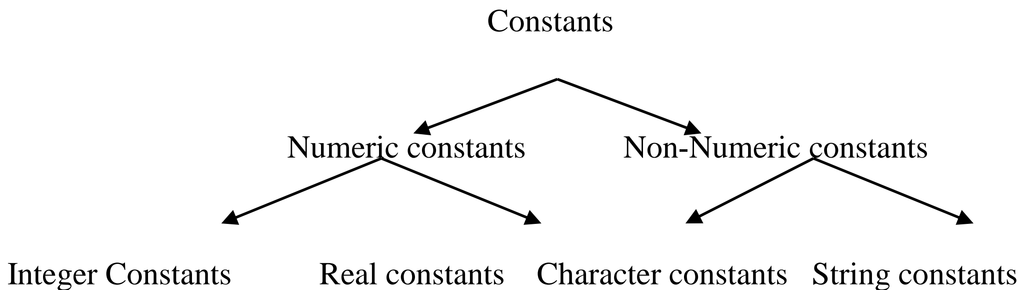


Fig.1.4.1 C Constants

Numeric Constants

Integer Constants

A *whole number* is called integer constants. An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal, octal and hexadecimal integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by optional minus sign. Valid **examples** are:

123   -231   0   253444

Embedded spaces, commas, and non-digit characters are not permitted between digits. For **example**

36   5689   20.000 $777456 are invalid numbers

An octal integer constant consists of any combination of digit from the set 0 through 7, with leading 0.

Valid **examples** are:

046   123   0   0675

A hexadecimal integer constant consists of any combination of digit from the set 0 through 9 and A through F and preceded by 0x or 0X

Valid **examples** are:

0x3Bf7   0X27   0x

Real Constants

Number containing with *decimal point* is called real constants. There are two types of notation, namely, decimal notation and Exponential (or scientific) notation.

In decimal notation a whole number followed by a decimal point and the fractional part, which is an integer.

Valid **examples** are:

0.376   .57   -.46   23.68

The *general form* of an Exponential notation is:

mantissa   e   exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase.

Valid **examples** are:

0.47E2 12e-7   1.8e+3   7.3E2   -6.0e-2

Non-Numeric Constants:

Character constants

A single character constant contains a *single character* enclosed within a pair of single quote marks. Valid **examples** are:

‘6’   ‘s’   ‘W’   ‘;’   ‘ ’

String constants

A string constant is a *sequence of characters* enclosed between double quotes. The characters may be alphabets, digits special characters and blank spaces.

Valid **examples** are:

"Hello" "3566" "%---$" "22-7"

Backslash character constants

C supports some special backslash character constants that are used in output methods. The characters combinations are known as escape sequences.

Table 2.6.3 Backslash character constants

Constants	Meaning
' \ b'	back space
' \ f '	form feed
' \ n'	new line
' \r'	carriage return
' \ t'	horizontal tab
' \ ''	single quote
' \ "'	double quote
' \ \'	back slash

Symbolic constants

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. For **example** of such a constant is 3.142 representing the value of the mathematical constant "pi". We face two problems in the subsequent use of program. They are:

1. Problem in modification of the program.
2. Problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculation. In this case, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce incorrect outputs.

Understandability

Assignment of a symbolic name to numeric constants frees us problems like same value means different things in different places. For **example**, the number 40 may mean the number of students at one place and the "pass marks"

at another place of the same program. We may use the name STRENGTH to denote the number of students and PASS_MARK to denote the pass marks required in subject. Constant values are assigned to these names at the beginning of the program.

A constant is declared as follows:

```
#define symbolic-name value of constant;
```

Valid **examples** are:

```
#define STRENGTH = 40;
#define PASS_MARK = 40;
#define PI = 3.1459;
```

Rules for forming the symbolic constants are:

- ❖ Symbolic names take the same form as variable names. But, they are written in CAPITALS.
- ❖ No blank space between the pound sign '#' and word define is permitted.
- ❖ '#' must be the first character in the line.
- ❖ A blank space required between #define and symbolic name and between the symbolic name and the constant.
- ❖ #define statement must not end with a semicolon.
- ❖ After declaration of symbolic constants, they should not be assigned any other value within a program.
- ❖ Symbolic names are NOT declared for data types.
- ❖ #define may appear anywhere in the program but before it referenced in the program.

1.4.2. Variables

A *variable* is an identifier that denotes a storage location used to store a data value. A variable may take different values at different times during the execution of the program.

Rules for naming the variables

1. Variable names may consist of alphabets, digits, the underscore (_) and dollar characters.
2. They must not begin with a digit.
3. Uppercase and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.
4. It should not be a keyword.
5. White space is not allowed.

6. The length of the variable names cannot exceed 8 characters and some of the 'C' compilers can be recognized up to 31 characters.
7. No commas or blank spaces are allowed within a variable name

Declaration Of Variables

Variables are the names of the storage locations. A variable must be declared before it is used in the program. A variable can be used to store a value of any data type. After designing the variable names, we must declare them to the complier. Declaration does three things:

1. It tells the compiler what the variable name is
2. It specifies what type of data the variable will hold.
3. The place of declaration decides the scope of the variables.

Primary Type Declaration:

The *general form* of declaration of a variable is:

data_type variable1, variable2,……, variableN;

where

data_type – is the type of the data.

variable1, variable2,….. variableN – are the list of variables.

Variables are separated by commas. A declaration statement must end with a semicolon. Some valid declarations are:

```
int     rollno;
float   average;
double  pi;
byte    b;
char    c1, c2;
```

User-Defined Type Declaration:

'C' language provides a feature to declare a variable of the type of user-defined type declaration. Which allows users to define an identifier that would represents an existing data type and this can later be used to declare variables. **typedef** cannot create a new data type. The *general form* is

typedef data_type identifier;

where

typedef – is the user defined type declaration.

data_type – is the existing data type.

identifier – refers to the 'new' name given to the data type.

Some valid declarations are:

typedef int marks;

marks m1, m2, m3;

Another user-defined data type is defined as

enum identifier{value1, value2,…….valueN};

where

enum - is the key word.

identifier - is a enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants).

After this definition, we can declare variables to be in this 'new' type as

enum identifier variable1, variable2……;

The enumerated variables variable1, variable2……. can only have one of the values value1, value2………….

Example:

```
enum day {Monday, Tuesday,…….., Sunday};    // enum definition
enum day week_st, week_end;                  // enum declaration
week_st = Monday;          // valid
week_st = May;             // invalid
```

Declaring a Variable as Constant:

The value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization.

Example:

const int class_strength = 40;

Giving Values To Variables

A variable must be given a value after it has been declared that before it is used in an expression. This can be achieved in *two ways*:

1. By using an assignment statement
2. By using a read statement

Assignment Statement

A simple method of giving value to a variable is through the assignment statement as

variableName = value;

For **Example:**

rollno = 1;

c1 = ‘ x ‘;

Another method to assign a value to a variable at the time of its declaration as

data_type variableName = value;

For **Example**:

```
int     rollno  =   1;
float   average =   68.66;
```

The process of giving initial values to variables is known as the initialization. The following are *valid C statements*:

```
float   x,  y,  z;          // declares three float variables
int     m =  3,  n  = 6;// declares and initializes two int variables
```

Reading data from Keyboard

We may also give values to variables through the keyboard using the **scanf** function. The *general form* of **scanf** is:

scanf(“control string”, &variable1, variable2…..);

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name’s address.

Example:

scanf (“%d”, &number);

When this statement is encountered by the computer, the execution stops and waits for the value of the variable number to be typed in. Since control string “%d” specifies that an integer value is to be read from the terminal. Once the number is typed in and the ‘Return’ key is pressed, the computer then proceeds to the next statement.

Scope Of Variables

‘C’ variables are classified into two kinds:

- Global/ External variables
- Local variables

The global/ external variables are declared before the function **main( )**. These are available for all the functions inside the program.

Example:

```
int a, b =2;                    fun()
main()                          {
{                                       int sum;
```

```
  …….                                    Sum = a +b;
  fun();                              }
}
```

The integer variables **a, b** are **global/external** variables, as they are declared before the function **main( )**. These variables are later used in the function **fun( ).**

Variables declared and used inside functions are called **local variables**. They are not available for use outside the functions definition. Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible (ie. Usable) is called its **scope.**

Example:

```
fun( )
{
 int i, j;
………
}
```

1.4.3. Data Types

The size and type of values that can be stored in variable is called as **Data type**. Data types in C under various *categories* are shown in Fig.1.4.2.

The user-defined types, derived types such as arrays, functions, structures and pointers are discussed in concern chapters.

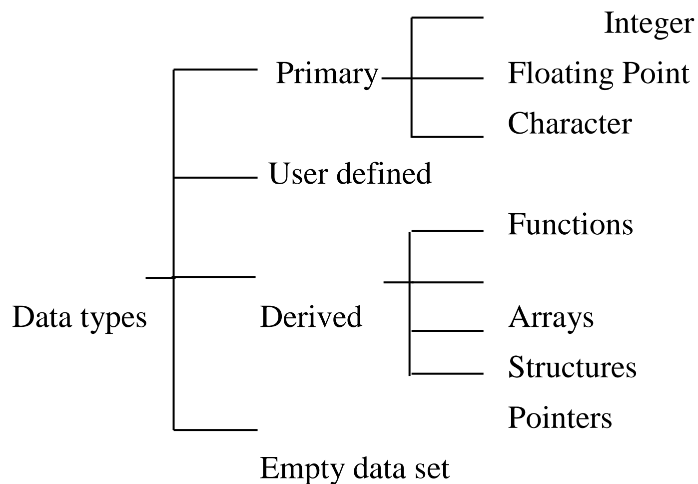


Fig 1.4.2. Data types in C under various *categories*

Primary Data Types:

The primary data types otherwise is called *as fundamental data types*. It can be classified into three types are

- **Integer data type**
- **Float type**
- **Character type**

Integer Types

Integer types can hold whole numbers. C supports *three types* of integers are **short int, int** and **long int**, in both **signed** and **unsigned** forms as in Fig.2.6.2.

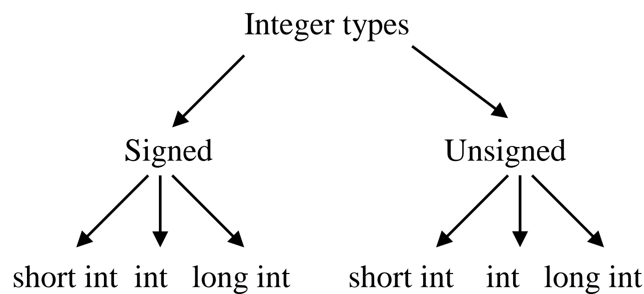


Fig.2.6.3 Integer data types

The integer occupies one word of storage typically 16 or 32 bits. The size of the integer depends upon the system. If we use 16 bit word length, the size of integer value is limited to the range –32768 to + 32767.

Floating Point Types

Integer types can hold only whole numbers and therefore we use another type known as **floating point or real type** to hold numbers containing fractional parts such as 27.59 and -1.375. There are *two kinds* of floating point storage in C as shown in Fig.2.6.3

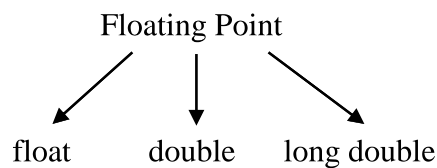


Fig.2.6.3 Floating-point data types

The **float** type values are single-precision numbers while the double types represent double-precision numbers. To extend the precision further, we may use long double. Table 2.6.2 gives the size of these *two types*.

Table 2.6.2 Type And Size Of Floating Point

Type	Size
float	4 bytes
double	8 bytes

Example:

1.23 7.56923e5

Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematical functions such as **sin, cos** and **sqrt** return **double** type values.

Character Type

In order to store character constants in memory, C provided a character data type called **char**. The char type assumes a size of 1 *byte* so it can hold only a *single character*. The qualifier **signed or unsigned** may be explicitly applied to char. While **unsigned char** have values between 0 and 255, **signed char** have values from –128 to 127.

Empty Data Set

The **void** is the empty data type in C language. This is generally specified with the function, which has no arguments.

1.4.4. Self Assessment Questions

Fill in the blank

1. ____________ function is used to read data through keyboard.
2. The global / external variables are declared before the _____________ function.

True / False

1. The scanf( ) is used to display the values on the screen.
2. The void data type represents empty.

Multiple Choice

1. Which of the following is not a valid identifier

a) averave b) sum_S

c) Total d) 2ABC

Short Answer

1. What is meant by variable?

----

2. Define Data type.

1.5.Operators and Expression

1.5.1. Introduction

C supports a rich set of operators. An **operator** is a symbol that tells the computer to perform certain *mathematical or logical* manipulations. Operators are used in programs to manipulate data and variables.

C operators can be classified into a *number of types* are:

- ❖ Arithmetic operators
- ❖ Relational operators
- ❖ Logical operators
- ❖ Assignment operators
- ❖ Increment and decrement operators
- ❖ Conditional operators
- ❖ Bitwise operators
- ❖ Special operators
- ❖ Special operators

1.5.2. Arithmetic Operators

C provides all the basic arithmetic operators are listed in Table1.51. The operators + , **-** , * , and **/** all work the same way as they do in other languages. These can operate on any built-in numeric data type of C. We *cannot* use these operators on *Boolean type*. The unary minus operator, in effect, multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

Table1.5.1 Arithmetic Operators

Operator	**Meaning**
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Remainder)

Integer division truncates any fractional part. The modulo division produce the remainder of an integer division. Arithmetic operators are used as

a - b   a + b

a * b a / b

a % b - a + b

Integer Arithmetic

When both the operands in single arithmetic expression such as a + b are integers, the expressions is called an ***integer expression***, and the operation is called ***integer arithmetic***. Integer arithmetic yields an integer value. In above **examples** if a and b are integers the a =2 and b = 2 we have the following results:

a – b = 0

a + b = 4

a * b = 4

a / b = 1 (decimal part truncated)

a % b = 0 (remainder of integer division)

For modulo division (%), the sign of the result is always the sign of the first operand.

Real arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. The floating- point modulus operator returns the floating-point equivalent of an integer division. What this means is that the division is carried out with both floating-point operands, but the resulting divisor is treated as an integer, resulting in a floating-point remainder. The operator % cannot be used with real operands.

Example Program: Program for Arithmetic operator works on Floating values

```
#include <stdio.h>
void main( )
{
        int a = 10, b= 3;
        printf(" a + b =  % d\n", (a + b));
        printf(" a - b =  % d\n " , (a - b));
        printf(" a * b = % d\n " , (a * b));
        printf(" a / b =  % d\n" , (a / b));
        printf(" a modulo b = % d\n" , (a % b));
}
```

Output Of Program

a + b = 13

a - b = 7

a * b = 30

a / b = 3

a modulo b = 1

Mixed-mode Arithmetic

When one of the operand is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then the other operand is converted to real and the real arithmetic is performed. The result will be a real. Thus

16 / 5 produce the result 3.1

Whereas

16 / 5 produce the result 1

1.5.3. Relational Operators

We often compare two quantities, and depending on their relation, take certain decisions. For **example**, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators. C supports six relational operators as shown in Table 1.5.2.

Table 1.5.2 Arithmetic Operators

Operator	Meaning
<	is less than
<=	is less than equal
>	is greater than
>=	is greater than equal
= =	is equal to
! =	is not equal to

A simple relational expression contains only one relational operator and is of the following form:

ae -1 relational operator ae – 2

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Example Program: Program to use various relational operators and display their return values.

```
#include<stdio.h>
void main( )
{
int a = 5 , b = 5, c = 50;
printf("\n Condition  :  Return values ");
printf("\n a != b          :  %d", a !=  b);
printf("\n a = = b         :  %d", a == b);
printf("\n a >= c          :  %d", a >=  c);
printf("\n a <= c          :  %d", a <=  c);
getch( );
}
```

Output Of Program

```
Condition   :  Return values
    a != b    :  0
    a = = b   :  1
    a >= c    :  0
    a <= c    :  1
```

In the above program, the condition is true it return 1and the condition false it returns 0.

1.5.4. Logical Operators

C has three logical operators as shown in Table 1.5.3. The logical operators && and || are used when we want to form compound conditions by combining two or more relations.

Example:

a > b && x == 10

An expression combines two or more relational expression is called as *logical expression* or *compound relational expression*. Logical expression also yields a value of true or false.

Table 1.5.3. Logical Operators

Operator	Meaning
&&	is logical AND
||	is logical OR
!	is logical NOT

Example Program: Program to demonstrate logical operator.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int c1, c2, c3;
    clrscr( );
    printf(" Enter values of c1, c2 & c3: ");
    scanf("%d%d%d", &c1, &c2, &c3);
    if( (c1 < c2)  &&  (c1 < c3) )
    printf("\c1 is less than c2  & c3");
    if( ! (c1 < c2))
    printf("\n c1 is greater than c2");
    if( (c1 < c2)  | | (c1 < c3) )
    printf("\nc1 is less than c2 or c3 or both");
    getch( );
}
```

Output Of Program

```
Enter values of c1, c2 & c3: 45 32 98
c1 is greater than
c1 is less than c2 or c3 or both
```

1.5.5. Assignment Operators

Assignment operators are used to assign the value of an expression to a variable. The *form*

v op= exp;

Where v is a variable, exp is an expression and op is a C binary operator. The operator op=is known as the shorthand assignment operator.

The shorthand assignment operators are illustrated in Table 1.5.4.

Table 1.5.4. Shorthand Assignment Operators

Statement with simple Assignment operator	Statement with Shorthand operator
a = a + 1	a + = 1
a = a - 1	a - = 1
a = a * (n + 1)	a * = n + 1
a = a % b	a % = b
a = a / (n + 1)	a / = n + 1

The use of shorthand assignment operators has *three advantages*:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read

Use of shorthand operator results in a more efficient code.

Example Program: Program to demonstrate assignment operator.

```
#include<stdio.h>
void main( )
{
        int i, j, k;
        k = (i = 4, j = 5);
        printf(" k  =  %d", k);
        getch( );
}
```

Output Of Program

```
k = 5
```

The above program prints the value of variable that it contains but that variable in terms assigned by two assignment variables. Such as k = (i= 4, j= 5) means first i value is assigned to k, then j value assigned and replaced the value of i. so the latest assignment value only contains the variable k.

1.5.6. Increment And Decrement Operators

The increment and decrement operators:

++ and - -

The operator ++ adds 1 to the operand while -- subtracts 1. Both are unary operators and are used in the following form:

```
++m;    or    m++;
--m;    or    m--;
```

We use the increment and decrement operators extensively in for and while loops.

Example:

```
m = 5;      y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements

```
m = 5;
y = m++;
```

Then, the value of y would be 5 and m would be 6. Prefix operator adds 1 to the operand and then the result is assigned to the variable on left. Postfix operator first assigns the value to the variable on left and then increments the operand.

Similar is the case, when we use ++(or --) in subscripted variables. That is the statement a[i++] = 10 is equivalent to

```
a[ i ]  =  10
i       =  i + 1
```

Example Program: Program for using increment and decrement operators.

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

Output Of Program

```
a++ = 10
++a = 12
 --a  = 11
 a--  = 11
```

where

a + + - post increment, first do the operation and then increment.

+ + a - pre increment, first increment and then do the operation.

- - a - pre decrement , first decrement and then do the operation.

a - -   - post decrement , first do the operation and then decrement.

Note : Do not use increment and decrement operator on floating point variables.

1.5.7. Conditional Operators

The conditional pair ? : is a ternary operator available in C. This operator is used to construct conditional expressions of the *form*

exp1 ? exp2 : exp3

Where exp1,exp2, exp3 are expressions.

The operator ? : works as follows : exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the conditional expressions. If exp1 is false, exp3 is evaluated and is value becomes the value of the conditional expression.

For **Example**:

```
a    =    10;
b    =    15;
x    =    (a > b)  ?  a : b
```

In this example, the value of x is the value of b.

Example Program:

```
#include<stdio.h>
void main( )
{
        int a, b, c, d;
        a = 5; b = 2; c = 3;
        d = ( a > b ) ? a : b + c;
        printf("Output is : %d", d);
        getch( );
}
```

Output Of Program

Output is : 5

1.5.8. Bit Wise Operators

C has a special operators is known as *bitwise* operators or manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left as shown in Table 1.5.5.

Table 1.5.5. Bitwise Operators

Operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise Exclusive OR
~	one's Complement
<<	shift left
>>	shift right

1.5.9. Special Operators

C supports some special operators of interest such as **comma operator, sizeof operator, pointer operators (& and *)** and **member selection operators (. and ->)**. The pointer and member selection operators are discussed in the pointers concepts.

The comma Operator

The comma operator can be used to separate the statement elements such as variables, constants or expression etc., and this operator is used to link the related expressions together, such expressions can be evaluated from left to right and the value of right most expressions is the value combined expression.

Example:

```
value = (x  = 10, y= 5, x + y);
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to **value**.

The sizeof operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

```
m= sizeof(sum);
n = sizeof(long int);
k = sizeof(123L);
```

1.5.10. Arithmetic Expressions

An arithmetic expression is a combination of variables, constants, and operators. **Example** of C expression is shown in Table 1.5.6.

Table 1.5.6. Expressions

Algebraic Expression	C Expression
a b - c	a * b - c
$\frac{ab}{c}$	a * b / c

1.5.11. Evaluation Of Expressions

Expressions are evaluated using assignment statement of the *form*

variable = expression;

variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side.

Examples of evaluation statements are

```
l = x*y-z ;
m = y/z*x ;
```

The blank space around an operator is optional.

1.5.12. Precedence of Arithmetic Operators

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. There are *two distinct priority levels* in C:

High priority * / %

Low priority + -

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators are applied and second pass; the low priority operators are applied.

Consider the following evaluation statement:

x = a - b / 3

When a = 9 and b = 6, the statement becomes

x = 9 – 6 / 3

and evaluated as

First pass

x = 9 – 2 (6 / 3 evaluated)

Second pass

x = 7 (9 - 2 evaluated)

Introducing parentheses into expression can change the order of evaluation. Parentheses may be nested, and in such cases, the expression will

proceed out from innermost set of parentheses. Every opening parenthesis has a matching closing one. Parentheses allow us to change the order of priority.

1.5.13. Type Conversions In Expressions

Automatic Type Conversion

C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the higher type before the operation proceeds. The result is of the higher type. Table 1.5.7. provides a reference chart for type conversion.

Table 1.5.7. Automatic Type Conversion Chart

	char	**byte**	**short**	**int**	**long**	**float**	**double**
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. The following changes are occurs in final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long to int causes dropping of the excess higher order bits.

Casting a Value

We need to store a value of one type into a variable of another type. In such situation, we must cast the value to be stored by proceed it with the type name in parentheses. The *general form* of a cast is:

(type_name) expression

Where type_name is one of the standard C data types. The expression may be constant, variable or an expression.

Examples of casts and their actions are

X = (int) 7.5 7.5 is converted to integer by truncation

A = (int) 21.3 / (int)4.5 Evaluated as 21/4 and the result would be 5

1.5.14. Operator Precedence And Associativity

Each operator in C has precedence associated with it. The operator at the higher level of precedence is evaluated first. The operator of the same level of precedence is evaluated from left to right or from right to left, depending on level. This is known as the associativity property of an operator. Table 1.5.8 provides a complete lists of operators, their precedence levels, and their rules of association.

Table 1.5.8. C Operators precedence and associativity

Operator	Meaning	Associativity	Rank
( )	Function call	Left to Right	1
[ ]	Array element reference		
-	Unary minus	Right to left	2
++	Increment		
--	Decrement		
!	Logical Negation		
~	One's complement		
*	Pointer reference (indirection)		
&	Address		
sizeof	Size of an object		
(type)	Casting		
*	Multiplication	Left to Right	3
/	Division		
%	Modulus		
+	Addition	Left to Right	4
-	Subtraction		
<<	Left shift	Left to Right	5
>>	Right shift		
<	Less than	Left to Right	6
< =	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
= =	Equality	Left to Right	7
! =	Inequality		

<table>
<tr><td>&</td><td>Bitwise AND</td><td>Left to Right</td><td>8</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>Left to Right</td><td>9</td></tr>
<tr><td>|</td><td>Bitwise Or</td><td>Left to Right</td><td>10</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to Right</td><td>11</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to Right</td><td>12</td></tr>
<tr><td>?:</td><td>Conditional operator</td><td>Right to Left</td><td>13</td></tr>
<tr><td>=
*= /= %=
+= -= &=
^= |=
<<= >>=</td><td>Assignment operator</td><td>Right to Left</td><td>14</td></tr>
<tr><td>,</td><td>Comma operator</td><td>Left to right</td><td>15</td></tr>
</table>

1.5.15. Mathematical Functions

C support basic math function through **#include <math.h>**. Table 1.5.9 lists the math functions defined **math.h** .

Example:

```
int   a = mod(10,15);
```

Table 2.7.9 Math Function

Function	Action
	Trigonometric
sin(x)	Returns the sine value of the angle x in radians
cos(x)	Returns the cosine value of the angle x in radians
tan(x)	Returns the tangent value of the angle x in radians
asin(y)	Returns the angle whose sine is y
acos(y)	Returns the angle whose cosine is y
atan(y)	Returns the angle whose tangent is y
atan2(x,y)	Returns the angle whose tangent is x/y
Hyperbolic	
sinh(x)	Returns the Hyperbolic sine value of the angle x in radians
cosh(x)	Returns the Hyperbolic cosine value of the angle x in radians
tanh(x)	Returns the Hyperbolic tangent value of the angle x in radians
Other Functions	
pow(x,y)	Returns x raised to y(xy)
exp(x)	Returns e raised to x(ex)

log(x)   Returns the natural logarithm of x

sqrt(x)   Returns the square root of x

ceil(x)   Returns the smallest whole number greater than or equal to x (rounding up)

floor(x) Returns the largest whole number less than or equal to x (Rounded down)

mod(x,y) Returns the remainder of x/y

abs(x) Returns the absolute value of x.

Note: x and y are double type parameters.

1.5.16. Self Assessment Questions

Fill in the blank

1. ______________ are used in programs to manipulate data and variables.

2. ______________ mathematical function return the smallest whole number greater than or equal to parameter x.

True / False

1. The logical operators do not return true and false value.

Multiple Choices

1. Suppose m=5,y=m++, what is value of m and y?

a) 5 & 6   b) 6 & 5   c) 5 & 5   d) 6 & 6

2. What is value of 5%2?

a) 2   b) 3   c) 0   d) 1

Short Answer

1. Define integer arithmetic.

----

2. What is called arithmetic expression?

1.6. Managing Input And Output Operations

1.6.1. Introduction

Most computer program that take some data as input and display the processed data, often known as **information** or **results.** In 'C' language, *two types of Input/Output statements* are available. All input/output operations are carried out through function calls such as **scanf** and **printf.** These functions are

collectively known as the **standard I/O library**. Each program that uses a standard input/output function must contain the statement

#include <stdio.h>

at the beginning. The file name **stdio.h** is an abbreviation for **standard input-output header** file. **#include <stdio.h>** tells the compiler to search for a file named **stdio.h** and place its content at this point in the program. The contents of the header file become part of the source code when it is compiled.

1.6.2. Reading And Writing A Character

Reading a Character using getchar( ) or **getc( )function**

Reading *a single character* can be done by using the function **getchar( )** or **getc( ).** The *general form* of **getchar( )** or **getc( )** function is

```
variable_name = getchar( );           variable_name = getc( );
            or                                  or
char variable_name = getchar( );      char variable_name = getc( );
```

where

variable_name is a valid C name that has been declared as **char** type.

When this statement is executed, the computer waits until a key is pressed and then assigns this character as a value to **getchar( )** function or **getc( )** function. The character value of **getchar( )** or **getc( )** is in turn assigned to the **variable_name** on the left.

For **Example**

```
char name;                          char name;
name = getchar( );                  name = getc( );
        or                                  or
char name = getchar( );             char name = getc( );
```

will assign the character 'D' to the variable **name** when we press the key D on the keyboard.

Writing a Character using putchar( ) or **putc( ):**

Function **putchar( )** or **putc( )** for writing characters one at a time to the screen. The *general form* of **putchar( )** or **putc( )** function is

```
putchar(variable_name);
        or
putc(variable_name);
```

where

variable_name is a valid C name that has been declared as **char** type variable containing a character. This statement displays the character contained in the variable_name at the screen.

For **Example**

```
putchar(name);
        or
putc(name);
```

will display the character 'D' contained in the **name** at the screen.

Example Program1: Program for converting a character from lower to upper and vice versa.

```
#include<stdio.h>
#include<ctype.h>
void main( )
{
        char ch;
        printf(" Enter any alphabet either in lower or upper case….");
        ch = getchar( );
        if (islower(ch))
                putchar(toupper(ch));
        else
                putchar(tolower(ch));
        getch( );
}
```

Output Of Program:

```
Enter any alphabet either in lower or upper case….s
S
```

The gets( ) and puts( ) function

The **gets( )** is used to read the string (is a group of characters) from the standard input device(keyboard). The *general form* of **gets()** function is

```
gets(array_variable);
```

where

array_variable is a valid C variable declared as one dimension char type.

For **Example**

```
gets(s);
```

The **puts( )** is used to display/write the string to the standard output device (monitor). The general form of **puts( )** function is

```
puts(array_variable);
```

where

array_variable is a valid C variable declared as one dimension char type.

For **Example**

```
s = 'Harsh';
puts(s);
```

Example Program2: Program for illustrating **gets( )** and **puts( )**.

```
#include<stdio.h>
#include<ctype.h>
void main( )
{
        char name[20];
        printf(" Enter the name:");
        gets(name);
        puts(" \n Name is :");
        puts(name);
        getch( );
}
```

Output Of Program:

```
Enter the name: Harshini
Name is : Harshini
```

Character Test Functions

C language many of character test functions (See Table 1.6.1) that are used to test the character taken from the input. These character functions are contained in the file **ctype.h** and therefore the statement

```
#include < ctype.h >
```

must be included in the program.

Table 1.6.1 Character test function

Function	Test
isalnum (c)	is c an alphanumeric character?
isalpha (c)	is c an alphabetic character?
isdigit (c)	is c a digit?
islower (c)	is c a lower case letter?
isprint (c)	is c a printable character?
ispunct (c)	is c a punctuation mark?
isspace (c)	is c a white space character?
isupper(c)	is c an upper case letter?
tolower (c)	convert c to lower case letter?
toupper (c)	convert c to upper case letter?

1.6.3. Formatted Input And Output

Formatted Input:

Formatted input refers to input data that has been arranged in a particular format. For **example,** consider the following data:

14.67 345 Harsh

This contains three pieces of data, that is arranged in a format, such data can be read to the format of its appearance, as the first data should be read into a variable **float**, the second into **int**, and the third into **char**. This is possible in C using the formatted input (scanf( )) statement. The **scanf( )** function is used to read information from the standard input device(keyboard). The general form of **scanf( )** is

scanf("control string", &variable1, variable2…..);

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name's address where the data is stored.

The control string contains the *field or format specification,* which direct the interpretation of input data. It may include

- Field or format specification, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs, or new lines.

The data type character indicates the type of data that is to be assigned to the variable associated with corresponding variable. The field width specifier is optional.

Inputting Integer Numbers

The field specification for reading an integer number is:

% w d

The **%** indicates that a conversion specification follows. **w** is an integer number that specifies the field width of the number to be read and d, known as data type character, indicates that the number to be read is in **integer mode**.

Example 1:

```
scanf("%2d %5d, &n1,  &n2);
```

Data line:

40   45323

The value 40 is assigned to n1 and 45323 to n2. Suppose the input data is as

45323   40

The variable n1 will be assigned 45 (because of %2d) and n2 will be assigned 323. The value 40 that is unread will be assigned to the first variable in the next **scanf** call.

This error may be eliminated if we use field specification without the field width specifications. That is, the statement

```
scanf("%d %d, &n1,  &n2);
```

will read the data

45323   40

correctly and assign 45323 to n1 and 40 to n2.

Example 2:

The **scanf** may skip reading further input. An input field may be skipped by specifying * in the place of field width.

```
scanf("%d %*d %d", &a, &b);
```

will assign the data

10   20   30

as follows:

10   to   a

20   skipped (because of *)

30   to   b

The data type character **d** may be preceded by '**l**' to read long integers. What happened if enter a floating-point number instead of an integer? The fractional part may be omitted away. Spaces, tabs or new lines must separate input data items. Punctuation marks do not count as separators.

Inputting Real Numbers

The field width of real numbers is not to be specified and therefore **scanf** reads real numbers using the simple specification **%f** for both the notations, namely, **Decimal point notation** and **exponential notation**.

Example:

scanf("%f %f %f", &x, &y , &z);

With the input data

456.56 24.56E-2 789

will assign the value 456.56 to x, 0.245 to y and 789.0 to z.

The number to be read is of double type, then the specification should be %lf instead of simple %f. a number may be skipped using %*f specification.

Inputting Character strings

A **scanf** function can input strings containing more than one character. The specification for reading character string is

%wsor %wc

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when argument is a pointer to a **char** variable. When we use **%wc** for *reading a string*, the system will wait until the **w**th character is keyed in. The specification **%s** terminates reading at the encounter of a blank space.

Reading Mixed data types:

The **scanf** statement to input a data line containing mixed mode data. For example

scanf("%d %c %f %s", &n, &ch, &fl, &str);

will read the data

10 k 4.64 hai

and assign the values to the variables in the order in which they appear.

Rules for writing the scanf statement:

- The control string must be preceded with % sign and must be given within double quotation.
- All function arguments, except the control string, must be pointers to variables.

- If there is a number of input data items must be separated by commas and must be preceded with & sign except for string input.
- The control string and the variables going to input should match with each other.
- It must be terminate with semicolon.
- The scanf reads the data values until the blank space in numeric input or maximum number of character have been read, or an error is detected.

Formatted Output:

The **printf** statement provides certain features to effectively control the alignment and spacing of print outs on the screen. The *general form* of **printf** statement is

printf("control string" , variable1, variable2…..);

Control string consists of the **three types** of items:

- Characters that will be printed on the screen as they appear.
- Format specifications that define the output format for display of each item.
- Escape sequence characters such as \n, \t, and \b.

where

control string - indicates how many arguments follow and what their types.

Arguments variable1, variable2….. are the variables whose values are formatted andprinted according to the specifications of the control string. The arguments should match in number, order and type with the format specification

A simple format specification has the following form:

% w.p type-specifier

where

w – is an integer number that specifies the total number of columns for the output value

p – is another integer number that specifies the number of digits to the right of the decimal point or the number of characters to be printed from string.

Both w and p are optional.

Output of Integer Numbers

The format specification for printing an integer number is

% w d

where

w - specifies the minimum field width for the output

d - specifies that the value to be printed to an integer

If the field width is not specified, it will be printed in full, without blank space in the right side.

Example:

printf("%d", 1234);

1	2	3	4

If the number is greater than specified field width, it will be printed in full.

Example:

printf("%2d", 1234);

1	2	3	4

If the number is less than specified field width, the number will be printed with right justified.

Example:

printf("%6d", 1234);

		1	2	3	4

The number can be left justified using the hyphen (-) after % character in the field width specification.

Example:

printf("%-6d", 1234);

1	2	3	4		

To pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier.

Example:

printf("%06d", 1234);

0	0	1	2	3	4

Long integers may be printed by specifying ld in the place of d in the format specification.

Output of Real Numbers

The format specification for printing a real number in decimal notation is

% w.p f

where

w – is an integer number that specifies the total number of columns for the output value

p – is another integer number that specifies the number of digits to be displayed after the decimal point .

The value, when displayed, is rounded to p decimal places and printed right justified in the field of w columns.

Example:

The value x = 39.6755

printf(“%7.4f”, x);

3	9	.	6	7	5	5

The precision is 6 decimal places. Leading blanks and trailing zeros will appear as necessary. The negative numbers will be printed with the minus sign.

Example:

The value x = 39.6755

printf(“%7.3f”, -x);

-	3	9	.	6	7	6

Padding the leading blanks with zeros and printing with left-justification is also possible by introducing 0 or – before the field width specification.

Example:

The value x = 39.6755

printf(“%07.2f”, x);

0	0	3	9	.	6	8

Example:

The value x = 39.6755

printf(“%-7.2f”, x);

3	9	.	6	8		

If the field width and precision is not specified, it will be printed in full, without blank space in the right side.

Example:

The value x = 39.6755

printf(“%f”, x);

3	9	.	6	7	5	5

The *format specification* for printing a real number in exponential notation is

% w.p e

Examples:

The value x = 39.6755

printf("%9.2e", x);

	3	.	9	7	e	+	0	1

The value x = 39.6755

printf("%10.4e", x);

3	.	9	6	7	6	e	+	0	1

Printing of a Single Character

The format specification for printing a single character in a desired position is

% w c

The character will be displayed right-justified in the field of w columns. We can make display left justified by placing a – before the integer w. the default value for w is 1.

Example:

The value x = 'A'

printf("%5c", x);

				A

The value x = 'A'

printf("%-5c", x);

A				

Printing of String

The format specification for printing strings to that of real numbers is

% w.p s

where

w specifies the field width for display

p instructs that only the first p characters of string are to be displayed.

The display is right justified. The character will be displayed right justified in the field of w columns. We can make display left justified by placing a – before the integer w. the default value for w is 1.

Examples:

The value x = ‘ Good Luck’

printf(“%s”, x);

<table><tr><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td><td>u</td><td>c</td><td>k</td><td></td></tr></table>

printf(“%10s”, x);

<table><tr><td></td><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td><td>u</td><td>c</td><td>k</td></tr></table>

printf(“%-10s”, x);

<table><tr><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td><td>u</td><td>c</td><td>k</td><td></td></tr></table>

printf(“%.6s”, x);

<table><tr><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td><td></td><td></td><td></td><td></td></tr></table>

printf(“%5”, x);

<table><tr><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td><td>u</td><td>c</td><td>k</td><td></td></tr></table>

printf(“%10.6s”, x);

<table><tr><td></td><td></td><td></td><td></td><td>G</td><td>o</td><td>o</td><td>d</td><td></td><td>L</td></tr></table>

Mixed Data Output

It is permitted to mix data types in one print statement.

Example:

printf(“%d %f %s %c”, a, b, c , d);

is valid.

The **printf** uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, incorrect results will be output.

Commonly use **printf** format are

Code	Meaning
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either exponent type or floating point type depending on value

%i	print a signed decimal integer
%o	print an octal integer without leading zero
%s	print a string
%u	print a unsigned decimal integer
%x	print an hexadecimal integer without leading 0x

Enhancing the readability of output

If we print the variables always with the field width and format specification the output will be unambiguous and can be difficult to identify and read. The clarity and correctness of output is all most important, so follow the given steps to enhance the readability of output.

- Specify the blank spaces in between the data items whenever necessary and applicable.
- Specify the appropriate headings and names for variables.
- Give the blank lines as and when required.
- Specify new line character whenever necessary.
- Print special text messages depending on necessary.

1.6.4. Self Assessment Questions

Fill in the blank

1.All input and output functions are collectively known as the _______________.

2. We can make display _________ justified by placing a – before the integer w.

True / False

1. The puts( ) function is used to read the string from the standard input device.
2. The function getc( ) and getchar( ) both are used read a single character.

Multiple Choice

1. Which format specification is used to print double type data?

   a) %f b) %d c) %lf d) %df

Short Answer

1. What is the purpose using control string in scanf function?.

1.7. Decision Making with Branching and looping

1.7.1. Branching

1.7.1.1. Introduction

We have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

Control or decision making statements are

1. if statements
2. switch statement
3. Conditional operator statement

1.7.1.2. Decision Making with IF statement

The **if statement** is a decision making statement and is used to control the flow of execution of statements. The *general form* is

```
if ( test expression)
```

It allows the computer to evaluate the expression first and then, based on the value of expression is 'true' or 'false', it transfers the control to a particular statement (See Fig 1.7.1.1)

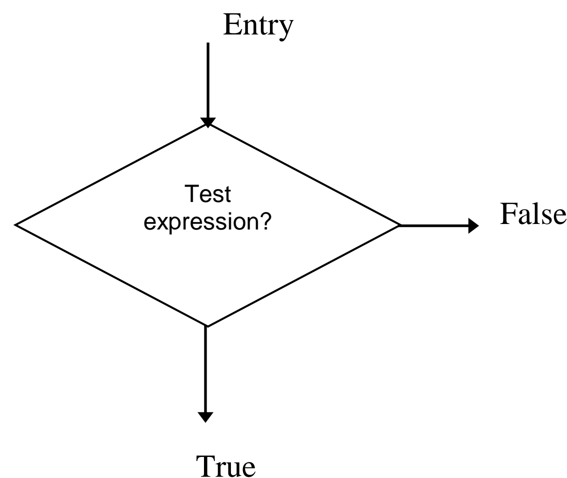


Fig. 1.7.1.1. Two-way branching

The *if statement* may be implemented in different forms based on condition to be tested.

1. Simple if statement
2. if …else statement
3. Nested if…else statement
4. else if ladder.

1.7.1.3. Simple If Statement

The *general form* is

```
if ( test expression)
{
statement-block;
}
statement-x;
```

The ‘statement-block’ may be single or group of statements. If the test expression is true, the statement-block will be executed; otherwise the execution will to the statement-x. (See Fig 1.7.1.2)

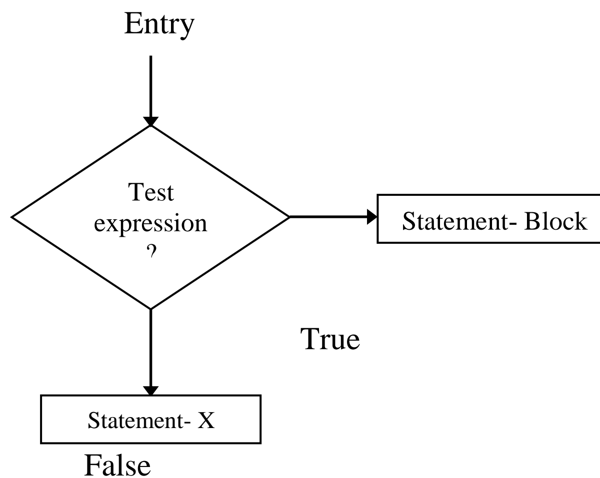


Fig.1.7.1.2.Flowchart of simple if control

Example Program: To find a smallest value among two numbers

```
/*
        Smallest Among Two numbers Using if statement
*/
#include<stdio.h>
void main( )
 {
```

```
        int a = 0,b=0;
        int small = 0;
        printf("\nEnter the two values");
        scanf("%d", &a);
        scanf("%d", &b);
        small = a;
        if ( small > b )
          small = b;
        printf("\nSmallest Among Two No.is : %d" , small);
        getch( );
    }
```

Output Of Program

```
Enter the two values
23
17
Smallest Among Two No.is : 17
```

1.7.1.4. The If-Else Statement

The *general form* is

```
if ( test expression)
{
statement-block1;
}
else
{
statement-block2;
}
statement-x;
```

If the test expression is true, the statement-block1 will be executed; otherwise, the statement-block2 will be executed, not both. In both the cases, the control is transferred to the statement-x. (See Fig.1.7.1.3)

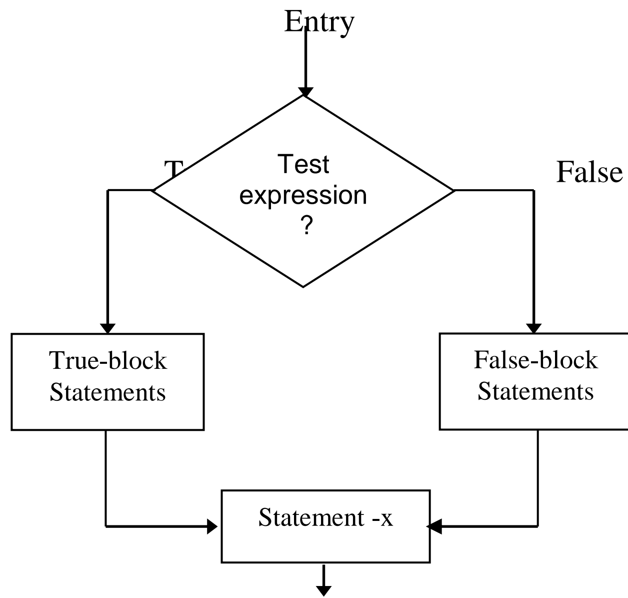


Fig.1.7.1.3. Flowchart of if….else control

Example Program: To Find a smallest value among three numbers.

```
/*
  Smallest Among Three numbers Using if – else statement
*/
#include<stdio.h>
void main( )
 {
        int a = 0,b=0,c=0;
        int small = 0;
        printf("\n Enter the three values");
        scanf("%d", &a);
        scanf("%d", &b);
        scanf("%d", &c);
        small = a;
        if ( small > b )
           small = b;
        else
           small = c;
        printf("\n Smallest Among Three No.is : %d " , small);
        getch( );
 }
```

Output Of Program

Enter the three values

23 17 56

Smallest Among Three No.is : 17

1.7.1.5. Nesting Of If-Else Statement

The *general form* is

```
if ( test expression1)
{
        if ( test expression2)
        {
        statement-block1;
        }
        else
        {
        statement-block2;
        }
}
else
{
statement-block3;
}
statement-x;
```

If the test expression1 is false, the statement-block3 will be executed; otherwise it continues to perform the second test. If test expression2 is true, the statement-block1 will be executed; other wise the statement-block2 will be executed and then control is transferred to the statement-x. (See Fig.1.7.1.4)

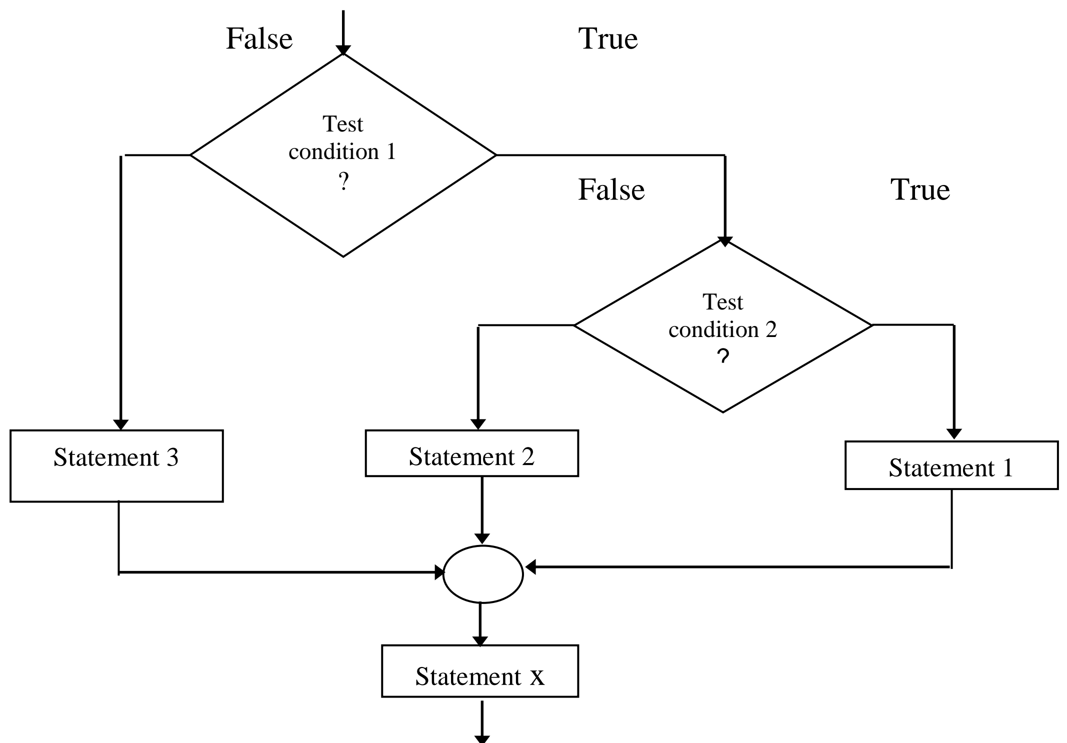


Fig.1.7.1.4. Flowchart of Nesting Of If…Else Statement

Example Program: To Find a largest value among three numbers

```
/*
  Largest Among Three numbers Using nested if-else statement

*/
#include<stdio.h>
void main( )
 {
        int a = 0,b=0,c=0;
        int large = 0;
        printf("\nEnter the three values");
        scanf("%d", &a);
        scanf("%d", &b);
        scanf("%d", &c);
        if ( a > b )
        {
```

```
        if (  a > c )
        {
                large = a;
        }
        else
        {
                large = c;
            }
}
else
{
        if (c > b)
        {
          large = c;
        }
        else
        {
            large = b;
            }
}
printf("\nLargest Among Three Number is : %d", large);
getch( );
}
```

Output Of Program

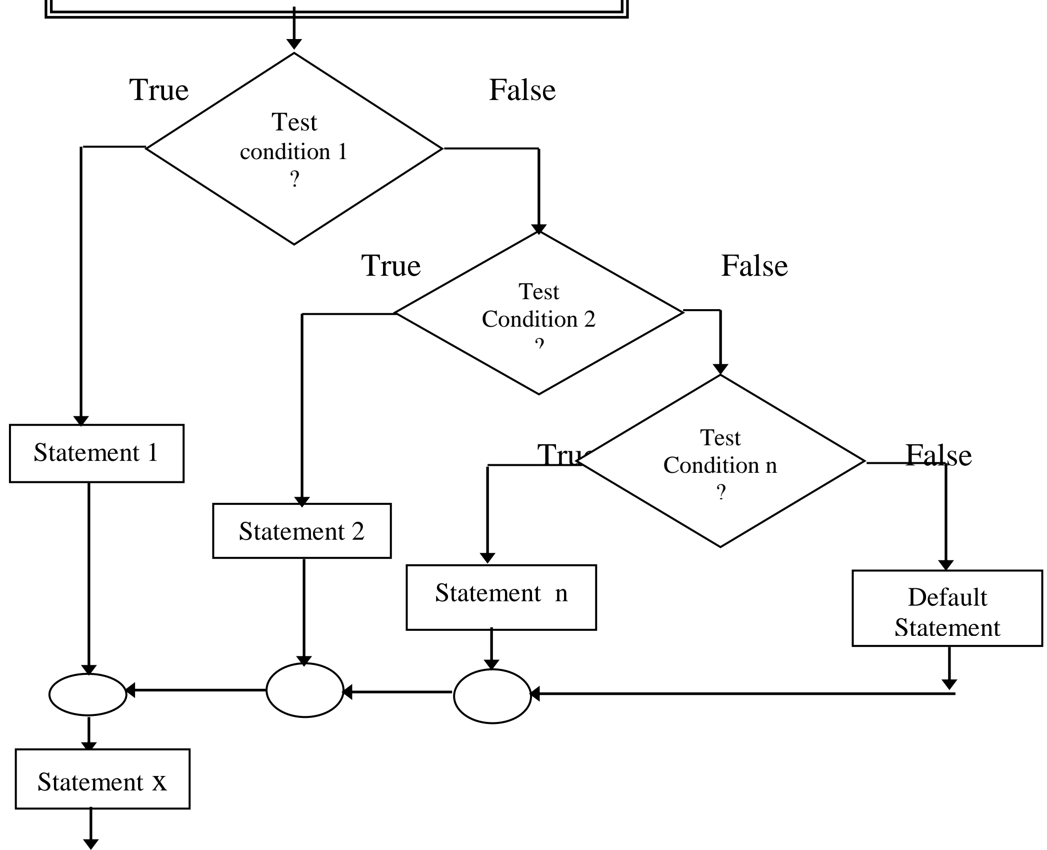
```
Enter the three values
23      17      56
Largest Among Three Number is : 56
```

1.7.1.6. The Else-If Ladder

This construct is known as the **else if** ladder. The test conditions are evaluated from the top, downwards. As soon as true test-condition is found, the statement associated with it is executed and control is transferred to the statement-x. When all test n conditions become false, then the else containing default-statement will be executed. (See Fig.1.7.1.5)

The *general form* is

```
if ( test expression1)
          statement-block1;
else if ( test expression2)
          statement-block2;
          ------------------------
          else if (test expression n)
                    statement-block-n;
          else
                    default -statement;
statement-x;
```



Next Statement

Fig.1.7.1.5 Flowchart of Else – if Ladder Statement

Example Program: To Find a largest value among three numbers

```
/*
Largest Among Three numbers Using else  - if  ladder statement

*/
#include<stdio.h>
void main( )
 {
        int a = 0,b=0,c=0;
        int large = 0;
        printf("\nEnter the three values");
        scanf("%d", &a);
        scanf("%d", &b);
        scanf("%d", &c);
        large = a;
        if ( large < b )
                large = b;
        else if (large < c)
                large = c;
        printf("\nLargest Among Three No.is : %d" ,large);
         getch( );
  }
```

Output Of Program

```
Enter the three values
23      17      56
Largest Among Three No.is : 56
```

1.7.1.7. The Switch Statements

The *general form* of the switch statement is

```
switch ( expression)
{
        case value-1:
                        block-1
                        break;
        case value-2:
                        block-1
                        break;
        ------------
        ------------
        default:
                        default-block
                        break;
}
statement-x;
```

The expression is an integer expression or characters. value-1, value-2---- are constants or constant expressions and are known as *case labels.* Each of these values should be unique within a **switch** statement. block-1, block-2, ------ are statement lists and may contain zero or more statement. There is no need to put braces around these blocks, case labels end with a colon (:).

When the switch is executed, the value of expression is compared with the values value-1, value-2,………if a case is found then the block of statements that follows the case are executed.

The break statement at the end of each block signals the end of the particular block and control is transferred to the statement-x.

The default is an optional case. When the values of expression not match with any of the case, the default case will be executed. If default statement not present, no action takes place when all matches fail and the control goes to the statement-x. (See Fig.1.7.1.6)

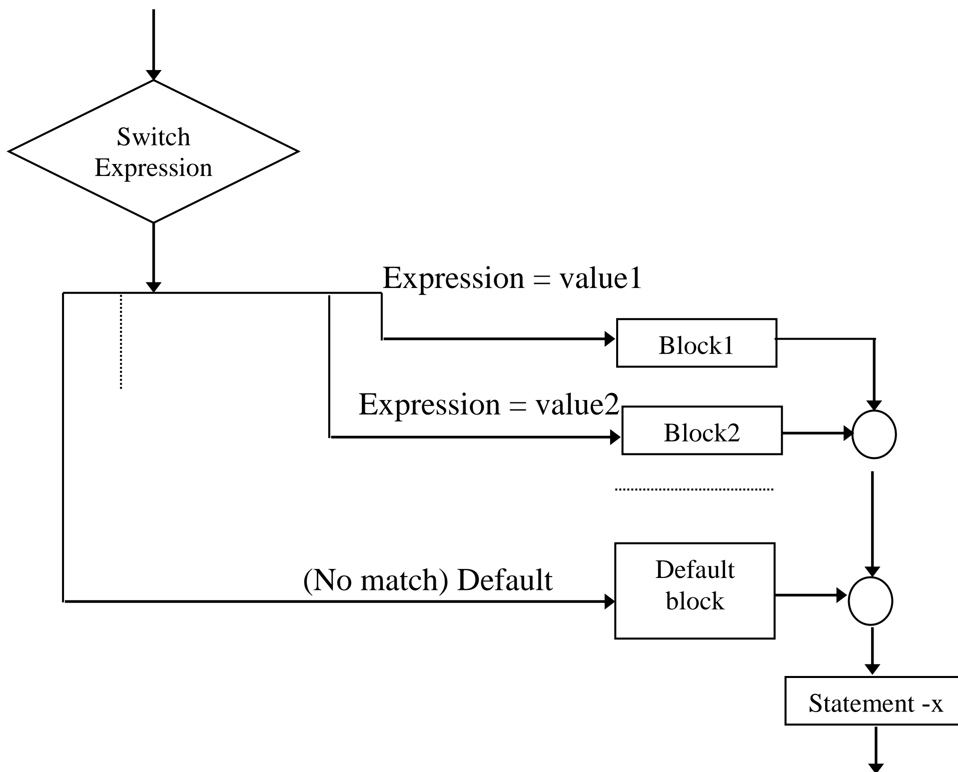


Fig.1.7.1.6 Flowchart of Switch statement

Example Program1: To demonstrate **switch-case** statement

```
/*
    Demonstration of switch-case statement
*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int choice = 0;
        printf("\nEnter the choice value");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:
                printf("I am in case 1\n");
                break;
        case 2:
```

```
                printf("I am in case 2\n");
                break;
        case 3:
                printf("I am in case 3\n");
                break;
        default:
                printf("I am in default case\n");
        }
  getch( );
  }
```

Output Of Program

Enter the choice value

2

I am in case 2

Enter the choice value

5

I am in default case

Example Program 2: To find whether the given number is even or odd using **switch-case** statement

```
/*
  Program for Even or Odd number using switch-case statement
*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n = 0;
        printf("\nEnter the number n");
        scanf("%d", &n);
        switch(n % 2)
        {
        case 0:
                printf("The given number  %d is even", n);
```

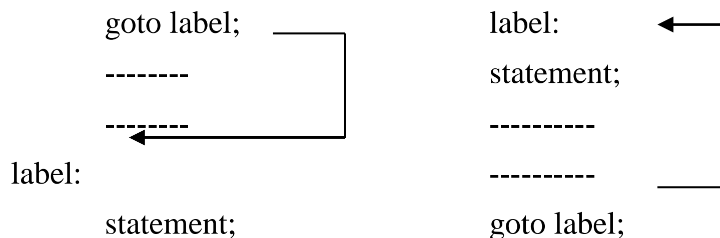
```
            break;
        case 1:
            printf("The given number  %d  is odd", n);
            break;
        }
    getch( );
    }
```

Output Of Program

```
Enter the number n
35
The given number 35 is odd
```

1.7.1.8.The Goto Statements

C supports the goto statement to branch unconditionally from one place to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The *general forms* of goto and label statements are



The label: can be anywhere in the program either before or after the goto label; statement. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is called as a backward jump. On the other hand, if the label: is placed after the goto label; some statement s will be skipped and jump is called as forward jump.

Example Program: To find the sum of positive numbers using **goto** statement

```
/*  Program for sum of positive number using goto statement

*/
    #include<stdio.h>
```

```
#include<conio.h>
void main( )
{
        int n = 0,sum =0, i;
        printf("\nEnter the 5 numbers ");
        for(i=1;i<=5;i++)
        {
                scanf("%d", &n);
                if (n < 0)
                    goto  endsum ;
                else
                     sum = sum + n;
        }
 endsum:   printf("\nSum of positive numbers is : %d" ,sum);
         getch( );
 }
```

Output Of Program

```
Enter the 5 numbers
12
34
56
-78
Sum of positive numbers is :102
```

1.7.2. Looping

1.7.2.1. Introduction

The process of repeatedly executing a block of statements is known as looping. The statements in block may be executed any number of times, from zero to infinite number is called an infinite loop. The program loop consists of *two segments* are

- ❖ Body of the loop
- ❖ Control statement (tests certain conditions and then directs the repeated execution of the statements in the body of the loop)

A Control structure may be classified either into *two types* are

- ❖ Entry-controlled loop: The control conditions are tested before the start of the loop execution. If conditions are not satisfied, the body of the loop will not be executed. (See Fig.1.7.2.1)
- ❖ Exit-controlled loop: The test is performed at the end of the body of the loop and therefore body is executed unconditionally for the first time. (See Fig.1.7.2.2)

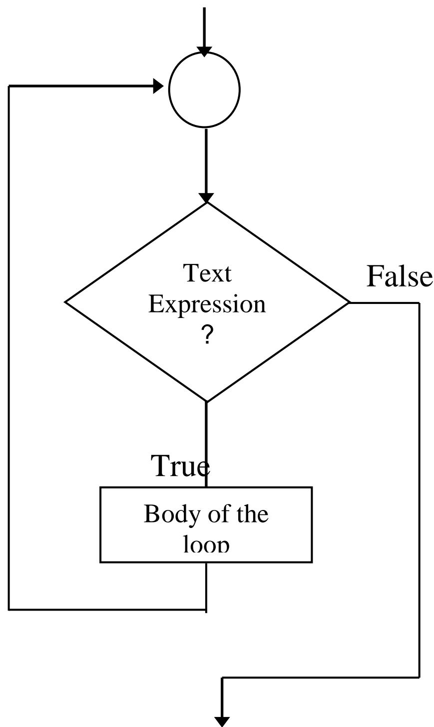


Fig.1.7.2.1 Entry control

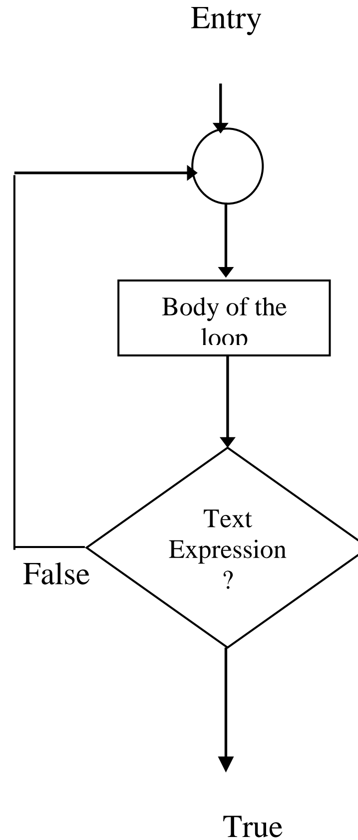


Fig.1.7.2.2 Exit control

A looping process will follow four steps:

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

They are three types looping construct are:

1. while construct
2. do construct
3. for construct

1.7.2.2. The While Statement

The *general form* is

```
Initialization;
while ( test condition )
{
        Body of the loop
}
```

The *while* is an entry-controlled loop statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. These processes of repeated execution of the body continue until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

Example Program: To reverse the given number using **while loop**

```
/*
      Reverse of the given number using while loop statement

*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int number= 0, digit = 0 , rev =0;
        printf("\nEnter the number");
        scanf("%d", &number);
        while (number != 0)
        {
                        digit = number %10;
                        rev = rev*10+digit;
                        number = number / 10;
                }
        printf("\nReverse of the given number is: %d " , rev);
        getch( );
  }
```

Output of Program

Enter the number

1234

Reverse of the given number: 4321

1.7.2.3. The do – while Statement

The *general form* is

```
Initialization;
do
{
        Body of the loop
}
while ( test condition )
```

The *do-while* is an exit-controlled loop statement. On **do** statement; the body of the loop will be executed first. At the end of the loop, the test conditions in the while statement is evaluated. If condition it true, the program proceed to continues to evaluate the body of the loop once again. This process continues as long as condition is true. When the condition become false, the loop will be terminated and control goes to statement after the while statement.

Example Program: To find the summation of 'n' numbers using **do–while** statement

```
/*
Summation of 'n' numbers using do–while statement
*/
#include<stdio.h>
#include<conio.h>
viod main( )
 {
        int n = 0, i = 1, sum = 0;;
        printf("\nEnter the value of  n");
        scanf(" %d", &n);
        do
        {
```

```
                sum = sum + i;
                i = i + 1;
        }
        while (i <= n);
        printf("\nSummation of n numbers is : %d" ,sum);
        getch( );
}
```

Output Of Program

```
Enter the value of n
11
Summation of n numbers is: 66
```

1.7.2.4. The For Statement.

The **for** loop is an entry-controlled loop. The *general form* is

```
for (initialization ; test condition ; increment)
{
Body of the loop
}
```

The execution of the **for** statement is as

1. *Initialization* of the control variables is done using assignment statements.
2. The value of the control variable is tested using the *test condition*.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement.

Example Program: To find the summation of 'n' numbers using **for** statement

```
/*
Summation of n numbers using for statement
*/
#include<stdio.h>
#include<conio.h>
void main( )
 {
        int n = 0, i = 1, sum = 0;;
```

```
        printf("\nEnter the value of  n");
        scanf(" %d", &n);
        for(i = 1 ; i <= n ; i++)
        {
                sum = sum + i;
        }
        printf("\nSummation of n numbers is : %d" ,sum);
        getch( );
    }
```

Output Of Program

```
    Enter the value of n
    10
    Summation of n numbers is: 55
```

Additional features of for loop

The **for** loop has several capabilities that are not found in other loop constructs. For example *more than one variable* can be *initialize*d at a time in the **for** statement.

```
    p = 1;
    for (n = 0 ; n<17; ++n)
```

can be rewritten as

```
    for (p =1, n = 0 ; n<17; ++n)
```

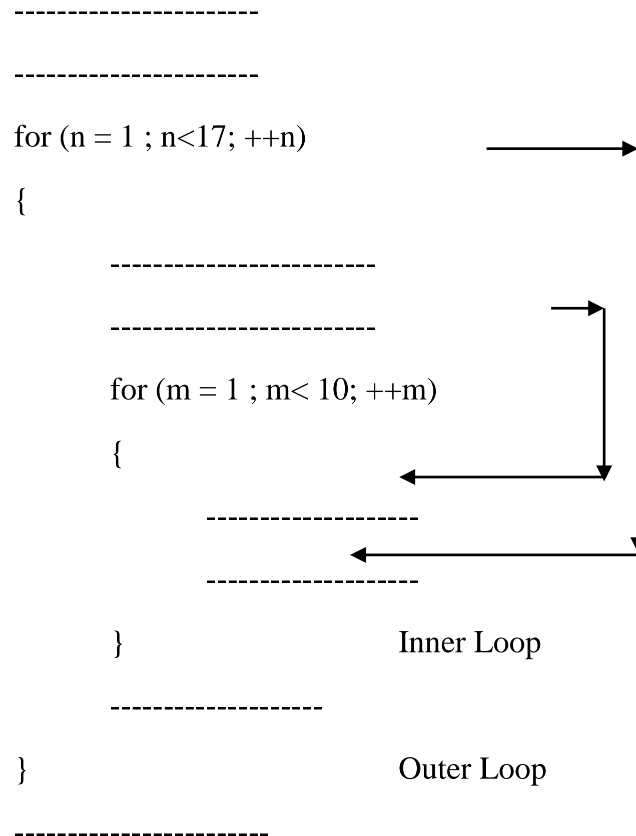
Increment section may also have more than one part. For example

```
    for (n = 0, m = 50  ; n<17; ++n, --m)
```

The *test condition* may have any compound relation and testing need not be limited only to the control variable.

Nesting of for loops

Nesting of loops, that is one **for** statement within another **for** statement, is allowed in C. For **example**



1.7.2.5. Jumps in loops

C permits a jump from one statement to the end or beginning of a loop as well as jump out of a loop.

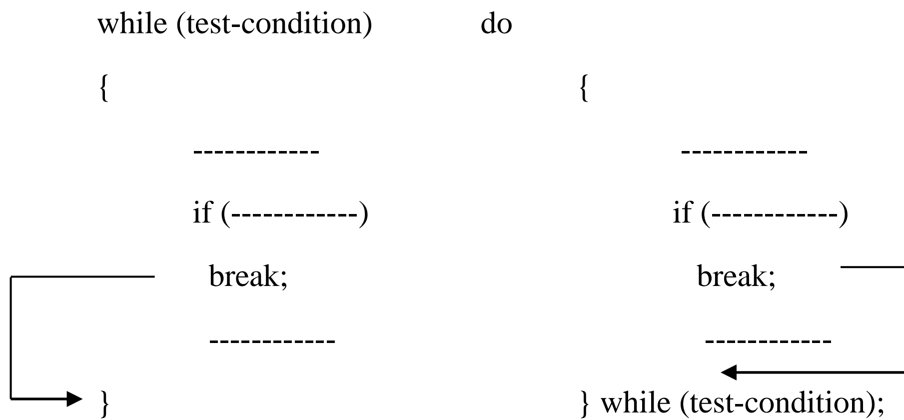
Jumping Out of a Loop

An early exit from a loop can be accomplished by using **break** statement. The *general form* is

```
break ;
```

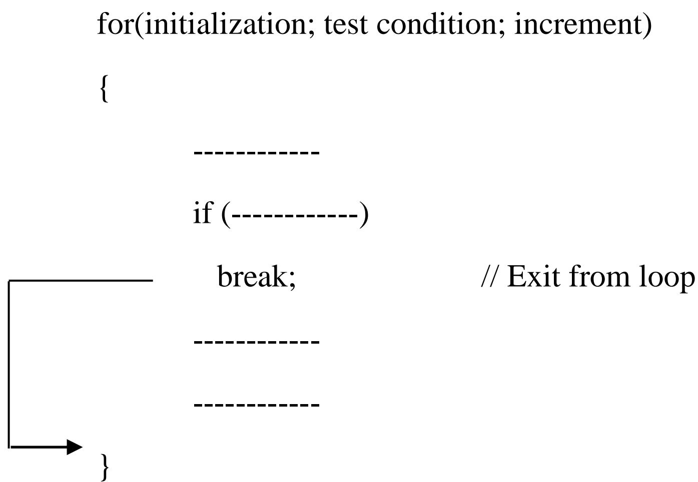
The break statement can be used within while, do, for loops. When the break statement is encountered inside a loop, the loop is immediately exited. When the loops are nested, the break would exit from containing it. The use of the **break** statement in loops is illustrated in Fig 1.7. 2.3.

A goto statement can transfer the control to any place in a program; it is useful to provide branching within a loop is illustrated in Fig.1.7.2.4.

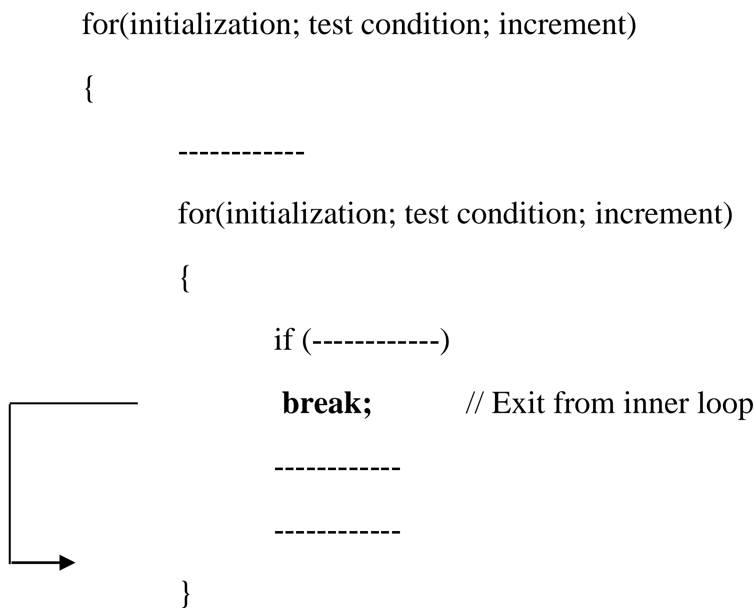

while (test-condition)
do
{
{

----
if (------------)
if (------------)
break;
break;

}
} while (test-condition);


for(initialization; test condition; increment)
{
----
if (------------)
break;
// Exit from loop

}


for(initialization; test condition; increment)
{
----
for(initialization; test condition; increment)
{
if (------------)
break;
// Exit from inner loop

}

}

Fig 1.7. 2.3 Bypassing and continuing in loops

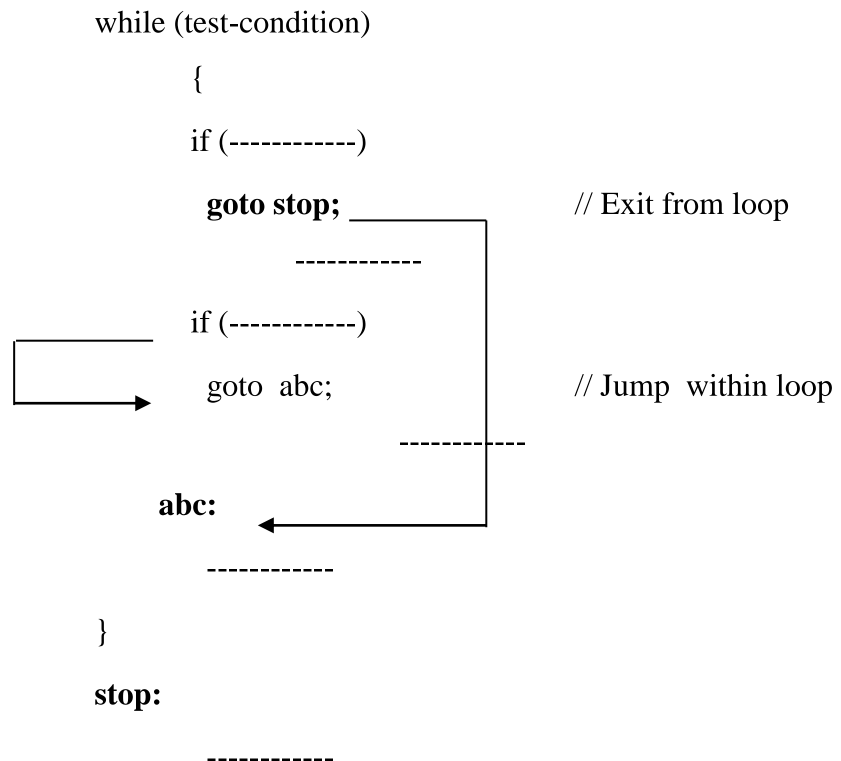


Fig 1.7.2.4 Jumping within and exiting from the loops with goto statement

Example Program: To find the sum of positive numbers using **break** statement

```
/*
  Program for sum of positive number using break statement

*/
        #include< stdio.h>
        #include<conio.h>
        void main( )
        {
```

```
        int n = 0,sum =0, i;
        printf("\nEnter the 5 numbers ");
        for(i=1;i<=5;i++)
        {
                scanf("%d", &n);
                if (n < 0)
                    break;
                else
                      sum = sum + n;
        }
        printf("\nSum of positive numbers is : %d", sum);
        getch( );
}
```

Output Of Program

```
Enter the 5 numbers
12
34
56
-78
Sum of positive numbers is :102
```

Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions by using **continue** statement. The *general form* is

continue;

The use of the **continue** statement in loops is illustrated in Fig 1.7.2.5. In **while** and **do** loops, continue causes the control go to directly test condition and then to continue the iteration process. In the case of **for** loop, the *increment* section of the loop is executed before the *test condition* is evaluated.

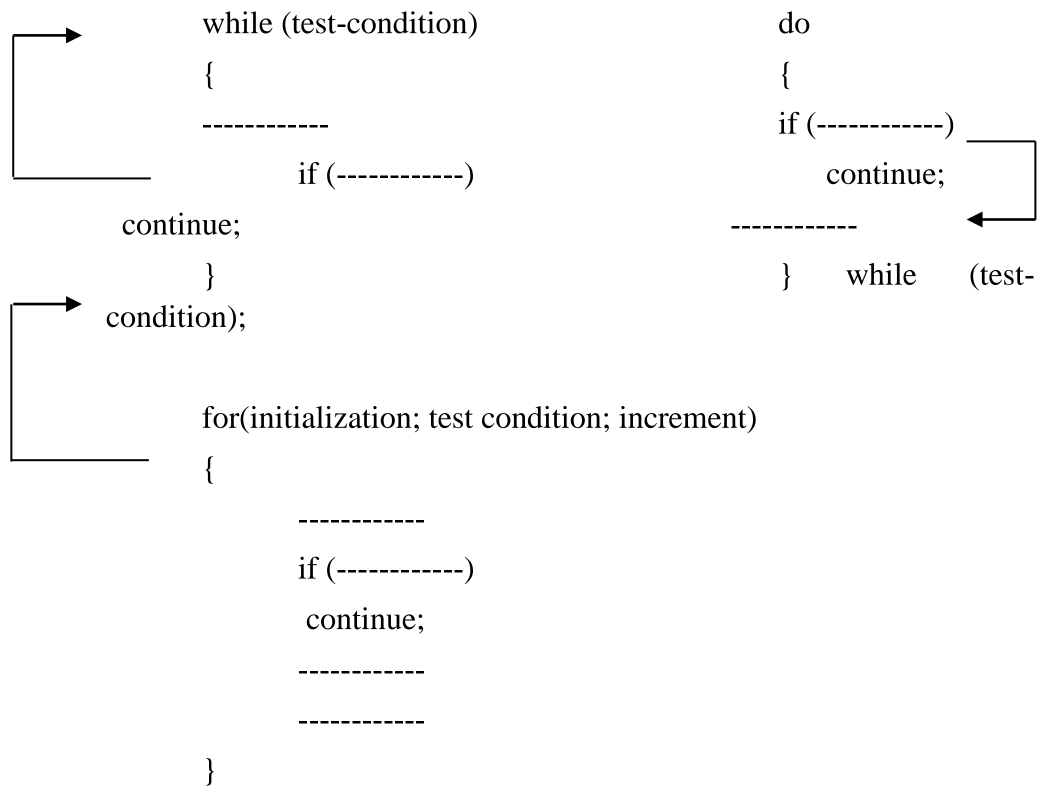


Fig 1.7.2.5 Bypassing and continuing in loops

Example Program: To find the sum of positive numbers using **continue** statement

```
/*
 Program for sum of positive number using continue statement

*/
        #include< stdio.h>
        #include<conio.h>
        void main( )
        {
                int n = 0,sum =0, i;
                printf("\nEnter the 5 numbers ");
                for(i=1;i<=5;i++)
                {
                        scanf("%d", &n);
                        if (n < 0)
                            continue;
```

```
                else
                      sum = sum + n;
            }
            printf("\nSum of positive numbers is : %d" sum);
            getch( );
      }
```

Output Of Program

```
Enter the 5 numbers
12      34      56      -78     10
Sum of positive numbers is :112
```

1.7.3. Self Assessment Questions

1. if statements returns either__________ or __________.

2.One loop within a loop is called __________________.

True / False

1. When the break statement is encountered inside a loop, the loop is immediately exited

2. for loop is an exit controlled loop statement.

Multiple Choices

1. Which is one of exit controlled loop?

a) do-while b) while loop

c) for loop d) none of the above

2. The segment of a program is

```
x=1; sum=0;
while ( x<=5)
{
      sum=sum+x;
      x++;
}
```

what is the value of sum?

a) 13 b) 14 c) 10 d) 15

Short Answer

1. List out branching control statement.

----

2. Define infinite loop

1.8. Summary

In this unit we have introduced some of the most fundamental parts of C language and the framework of C program. You have also discussed about how to use variable, expression, and operators. You have also learnt how the precedence rules work with arithmetic statements and how the integer and floating point conversions take place.

Among input/output statements you have learnt to use scanf( ) and printf(), and how to handle input/output of different types of variables using these.

You have learnt how to create and use decisions making looping and branching statements in variety of situations

1.9. Unit questions

1. Explain the structure of C program.
1. Define data types. Explain its types with example.
2. Discuss various types constant with example.
3. Explain in detail about

a) Formatted Input /Output statements b) Unformatted Input / Output statement

4. Distinguish between if-else and switch-case statement.
5. Briefly explain conditional or decision making statement.
6. Find the output of the following expressions.

   a) 4 + 8/2 * 7 + 4 b) 4%3 * 5/2 + (5*5)

7. Write a program to find the sum of digits of an integer using while-loop.
8. Write a program to print a name using gets( ) and puts( ) function.
9. Briefly explain different types operator in C.

1.10. Answers for Self Assessment Questions

Answer 1.3.4

Fill in the blank

1. Dennis Ritchie   2. Structured programming language

True / False

1. False   2. True

Multiple Choice

1. b

Short Answer

1. Variables declared and used inside functions are called local variables.

Answer 1.4.4

Fill in the blank

1. scanf ( ) function   2. main( ) function

True / False

1. False   2. True

Multiple Choice

1. d

Short Answer

1. A variable is an identifier that denotes a storage location used to store a data value.

2. The size and type of values that can be stored in variable is called as **Data type**.

Answer 1.5.16

Fill in the blank

1. Operators.   2. ceil( ) function

True / False

1. False

Multiple Choice

1. a   2. d

Short Answer

1. When both the operands in single arithmetic expression such as a + b are integers, the expressions is called an integer expression, and the operation is called integer arithmetic.

2. An arithmetic expression is a combination of variables, constants, and operators

Answer 1.6.4

Fill in the blank

1. standard I/O library 2. left

True / False

1. False 2. True

Multiple Choice

1. %lf

Short Answer

1. The type of data that the user going to accept via scanf function. This can be formatted and always preceded with a '%' sign.

Answer 1.7.3

Fill in the blank

1. true or false 2. Nested loop

True / False

1. True 2. False

Multiple Choice

1. a 2. d

Short Answer

1. if statements, switch statement, Conditional operator statement

2. The statements in block may be executed any number of times, from zero to infinite number is called an infinite loop.

NOTES

UNIT – II

2.1 Introduction

Structure and union, which is method for placing data of different types. A structure is convenient tool for handling a group of logically related data items. But arrays can be used to represent a group of data items that belong to the same type.

In this unit we shall introduce three different derived data types such as structure & union, arrays and functions. We discuss how structure & union, arrays and functions can be defined and used in C.

2.2 Objectives

After studying this unit, you should be able to

- Create a structure & union and reference each of its members.
- Create and manipulate various types of array.
- Create and manipulate function in C.
- Understand the term scope, local and global and how each will affects variables within functions.

2.3 Structure And Unions

2.3.1 Introduction

C provides a constructed data type known as structure, which is a method for packing data of different types. A structure contains one or more data items of different data type in which the individual elements can differ in type. The individual element in structure is called members.

Example:

You might want to process information on students, in the category of names and marks. Here we can declare the structure '**student**' with the fields, **names** and **marks** and we can assign their appropriate data types. These fields are called *members of the structure.*

2.3.2 Structure Definition or Template Declaration

The structure creates a format that may be used to declare structure variables. The *general form* for defining the structure as

```
struct structure_name
{
 type   structure_element 1;
 type   structure_element 2;
------------
type   structure_element n;
};
```

where

struct – is the keyword that declares a structure.

struct_name – is the name of the structure and is called the structure tag.

type – specifies the data type of the structure elements.

structure_element 1,structure_element 2… - are the structure elements or members. Each element may be belonging to a different type of data.

Example:

```
struct student
{
 char   student_name;
 int     student_rno;
 int     student_mark1;
 int      student_mark2;
};
```

Rules for defining a structure

- ❖ A structure must be end with semicolon (;).
- ❖ A structure appears at the top of the source program.
- ❖ While entire declaration is considered as a statement, each member is declared independently for its name and type in separate statement inside the structure.
- ❖ The tag name, such as struct_name can be used to declare the structure variables of its type.
- ❖ The structure elements or members must be accessed with structure variable with dot (.) operator.

2.3.3 Declaration of Structure Variable

We can declare structure variables using the tag name any where in the program. The *general form of structure variable* declaration is

```
struct struct_name struct_var1, struct_var2…….struct_varn;
```

where

struct – is the keyword

struct_name - is the name of the structure

struct_var1, struct_var2…..- are the structure variables

Example:

struct student s1, s2 …. sn;

C also supports to combine both the template declaration and variables declaration in one statement. The *general form* is

```
struct structure_name
{
        type   structure_element 1;
        type   structure_element 2;
        ------------
        type   structure_element n;
} struct_var1, struct_var2…….struct_varn;
```

Example:

```
struct student
{
int      student_rno;
char    student_name;
int      student_mark1;
int      student_mark2;
}s1, s2 …. sn;
```

2.3.4 Giving Values To Members

We can assign values to the members of a structure in a number of ways. The link between a member and a variable is established using the member operator '.' Which is also known as dot operator.

Example

```
s1.student_rno;
```

is the variable representing the roll number of student and it treated like ordinary variable.

We assign values to the members of student:

```
s1.student_rno = 1000;
strcpy(s1.student_name, 'Harsh');
s1.student_mark1 = 100;
s1.student_mark2 = 99;
```

we can also use scanf to give the values through the keyboard.

```
scanf("%d\n", s1.student_rno);
scanf("%s\n", s1.student_name);
```

are valid input statements.

2.3.5 Structure Initialization

Like any other data type, a structure variable can be initialized, but this initialization can be made at the compile time. This type of initialization can be done in two ways.

Example1:

```
struct student
{
int     student_rno;
char    student_name;
int     student_mark1;
int     student_mark2;
} s1 = {1000, 'Harsh',100, 99};
```

There is one-to-one correspondence between the members and their initializing values.

Example2:

```
struct student
{
int     student_rno;
char    student_name;
int     student_mark1;
int     student_mark2;
```

```
};
 main()
 {
 struct student s1 ={1000, 'Harsh',100, 99};
 }
```

2.3.6 Comparison Of Structure Variable

Two variables of the same structure type can be compared the same way as ordinary variables. If student1 and student2 belong to the same structure, then the following operations are valid:

Operation	Meaning
student1 = student2	Assign student2 to student1.
student1 = = student2	Compare all members of student1 and student2 and return 1 if they are equal, 0 otherwise.
student1 ! = student2	Return 1 if all the members are not equal, 0 otherwise.

2.3.7 Arrays Of Structures

We may declare an *array of structures*, each element of the array representing a structure variable. For **example**

```
struct   class student[100];
```

defines an array called *student*, that consists of 100 elements. Each element is defined to be of the type struct class.

```
struct student
{
    int      student_rno;
    char    student_name;
    int      student_mark1;
    int      student_mark2;
};
main()
{
    struct student s1[2]= {{1000, 'Harsh',100, 99},{1001, 'Raj',78, 97}};
}
```

This declares the s1 as an array of two elements s1[0] and s1[1] and initializes their members as

```
s1[0] .rno = 1000;
s1[0].name = 'Harsh';
.
.
s1[1].mark2= 97;
```

2.3.8 Arrays Within Structures

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single or multi dimensional arrays of type int or float. For **example**

```
struct student
{
        int     student_rno;
        char    student_name[20];
        int     student_mark[2];
} s1[2];
```

Here, the member mark contains two elements, mark[0] and mark[1].These elements can be accessed using appropriate subscripts.

For **example**

```
s1[1].mark[1];
```

would refer to the marks obtained in the second subject by the second student.

Example Program: Program to illustrate structure with arrays

```
#include<stdio.h>
main( )
{
        struct student
        {
                int     student_rno;
                char    student_name[10];
                int     student_mark[2];
        } s1[2];
        int i, j;
        printf("\nEnter roll no, name and marks");
        for( i = 0; i < 2; i++)
        {
```

```
            scanf("%d%s",                          &s1[i].student_rno,
        s1[i].student_name);
            for(j = 0; j<2; j++)
            {
                scanf("%d",
            &s1[i].student_mark[j]);
            }
        }
        printf("R.No   Name   Mark1   Mark2");
        for( i = 0; i < 2; i++)
        {
            printf("\n%d\t%s",                          s1[i].student_rno,
        s1[i].student_name);
            for(j = 0; j<2; j++)
            {
                printf("\t%d",s1[i].student_mark[j
            ]);
            }
        }
        getch( );
    }
```

Output Of the Program

```
Enter roll no, name and marks
1000
Ramu
100
90
2000
Gopi
89
78
R.No   Name   Mark1   Mark2
1000   Ramu    100      90
2000   Gopi     89      78
```

2.3.9 Structures Within Structures

Structure within a structure means nesting of structures. For **example**

```
struct salary
{
        char name[20];
        char dept[10];
        struct
        {
                int dearness;
                int house_rent;
        } allowance;
    } employee;
```

The salary structure contains a member named allowance which itself is a structure with two members. The members contained in the inner structure namely dearness and house_rent can be referred to as

```
employee.allowance.dearness
employee.allowance.house_rent
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator.

An inner structure can have more than one variable. For **example**

```
struct salary
{
        --------------
        struct
        {
                int dearness;
                int house_rent;
        }
        allowance;
        arrears;
}employee;
```

The inner structure has *two variables*, allowance and arrears. We can also use tag names to define inner structures. For **example**

```
srtuct pay
```

```
{
        int dearness;
        int house_rent;
};
struct salary
{
        char name[20];
        char dept[10];
        struct pay allowance;
        struct pay arrears;
};
struct salary employee[100];
```

pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

2.3.10 Structures and Functions

C supports the passing of structure values as arguments to functions. There are **three methods** by which the values of a structure can be transferred from one function to another.

- The **first method** is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables.
- The **second method** involves passing of a copy of the entire structure to the called function.
- The **third approach** employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function.

In this section, we discuss about the second method, while the third approach using pointers is discussed in the pointer chapter.

The *general form* of sending a copy of a structure to the called function is

```
function_name ( structure_variable_ name)
```

the *called function* takes the following *form*:

```
data_type function_name(structure_name)
struct_type st_name;
{
--------
--------
return(expression);
}
```

Example Program:

```
#include<stdio.h>
struct std
{
        int a;
        char b;
};
void  fun( struct std  s);
void main( )
{
        struct std a1;
        a1.a = 10;
        a1.b = ‘H’;
        fun(a1);
        getch( );
}
void fun(struct std s);
{
        printf(“The value of a is :%d\n”, s.a);
        printf(“The value of b is :%c\n”, s.b);
}
```

Output Of Program:

```
The value of a is :10
The value of b is :H
```

2.3.11 Size Of Structure

We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The *expression*

Sizeof (struct a)

will evaluate the number of bytes required to hold all the members of the structure **a**.

2.3.12 Unions

Unions are a same concept of structures and therefore follow the *same syntax as structures.* However, there is major distinction between them in terms of storage. In structures each member has its own storage location, whereas all the members of a 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. Like structures a union can be declared using the keyword union as

```
union item
{
int a;
float b;
char c;
} code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. Fig.2.3.1 shows how all three variables share the same address. The compiler allocates a piece of storage that is *large enough to hold the largest variable type* in the union.

To access a union member, we can use same syntax that we use for structure members. That is,

```
code.a
code.b
code.c
```

Unions may be used in all places where a structure is allowed.

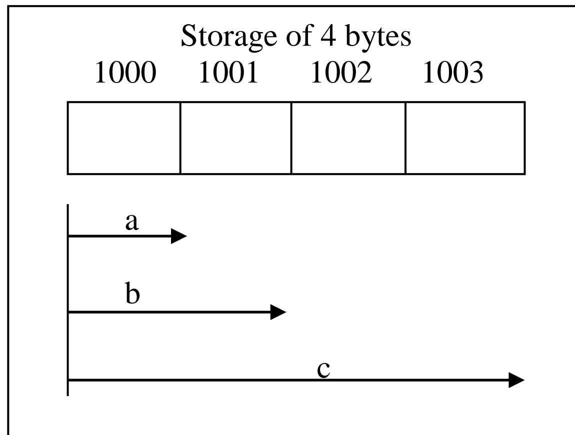


Fig.2.3.1 Sharing of storage location by union members

Example Program 1: Program for summation of three numbers using Union.

```
#include <stdio.h>
union  sum
{
        int a;
        int b[2];
};
main( )
{
        union sum  s;
        int s1;
        s.a = 10;
        s.b[0] = 20;     s.b[1] = 30;
        s1 = s.a + s.b[0] + s.b[1];
        printf("The sum of three numbers is : % d\n", s1);
        getch( );
}
```

Output Of Program:

The sum of three numbers is : 70

Example Program 2: Program to find number bytes reserved for union and structure.

```
#include <stdio.h>
main( )
{
```

```
union  result
{
        int  mark;
        char grade;
};
struct res
{
        char name[15];
        int age;
        char sex;
        union result perf;
}res1;
printf("Size of Union :  %d\n", sizeof(res1.perf));
printf("Size of Structure : %d\n", sizeof(res1));
getch( );
}
```

Output Of Program:

Size of Union : 2

Size of Structure : 20

2.3.13 Self Assessment Questions

Fill in the blank

1.________________ keyword is used to declare the structure.

2. The link between a member and a variable of structure is established using the ________________member operator.

True / False

1. All structure elements are stored in contiguous memory locations.

2. An array should be used to store dissimilar elements and a structure to store similar element.

Multiple Choice

1. Given the statement

    student.result.mark = 60;

a) structure mark is nested within structure result

b) structure result is nested within structure student

c) structure student is nested within structure result

d) structure student is nested within structure mark

Short Answer

1. How structure elements can be accessed?

----

2. Define union.

2.4 Arrays

2.4.1 Definition

An **array** is a group of related data items that *share a common name*. An array of value can be accessed by *index or subscript* enclosed in square brackets after array name. For **example**, a[5] represents the 5th element in an array.

The subscript can begin with number 0. The value of each subscript can be expressed as an integer constant or integer variable or an integer expression.

Arrays can be *classified* into

- One-Dimensional arrays
- Two-Dimensional arrays
- Multi-Dimensional arrays

Features of arrays:

- An array is derived data type. It is used to represent a collection of elements of the same type.
- The elements can be accessed with base address (index) and the subscripts define the position of the element.
- In array the elements are stored in continues memory location. The starting memory location is represented by the array name and it is known as the base address of the array.
- It is easier to refer the array elements by simply incrementing the value of the subscript.
- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.

2.4.2 One Dimensional Arrays

A list of data items can be stored under a one variable name using only one subscript and such a variable is called a *single-subscripted* variable or *one-*

dimensional array. For **example,** if we want to represent a set of four numbers, by array variable **a** then we may declare the variable **a** as

```
int    a [ 4 ];
```

and computer reserves four memory locations as

a[0]	
a[1]	
a[2]	
a[3]	

The values to the array elements can be assigned as

```
a[0]   =  10 ;
a[1]   =  20 ;
a[2]   =  30 ;
a[3]   =  40 ;
```

This would cause the array **a** to store the values as

a[0]	**10**
a[1]	**20**
a[2]	**30**
a[3]	**40**

This element may be used in programs just like any other C variable. For **example**, the following are **valid statements**:

```
b = a[0] + 10;
a[3] = a[0] + a[2];
```

2.4.3 Declaration Of One Dimensional Arrays

Declaration of arrays

The *general form* of array declaration is:

```
type arrayname[size];
```

The *type* specifies the type of element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array.

Examples:

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

declares the **a** to be an array containing 10 integer elements. Any subscript 0 to 9 are valid.

```
float  b[5];
```

declares the **b** to be an array containing 5 real elements. Any subscript 0 to 4 are valid.

```
char  c[8];
```

declares the **c** as a character array containing 7 character elements. Any subscript 0 to 6 are valid. Suppose we read the following string constant into the string variable **c**.

"WELCOME"

Each character of the string is treated as an element of the array **c** and is stored in the memory as follows:

'W'
'E'
'L'
'C'
'O'
'M'
'E'
'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element c[8] holds the null character '\0' at the end.

2.4.4 Initialization Of One-Dimensional Arrays

Initialization of arrays

To put values into array created is known as *initialization.* This is done using array subscripts as

arrayname[subscript] = value;

Example:

```
a[0]    =       28;
------------------
------------------
a[3]    =       57;
```

C creates arrays starting with subscript of 0 and ends with a value one less than the *size* specified.

We can also initialize arrays automatically when they are declared, as

type arrayname[ ] = { list of values };

The array initializer is a list of values separated by commas and surrounded by curly braces.

Example:

```
int a[ ] = { 10, 20,30,40};
char c[ ] = {'W','E','L','C','O','M','E'};
```

Loops may be used to initialize large size arrays.

Example:

```
----------------
----------------
for(int   i = 0;    i<10;     i++)
{
```

```
        a[i] = i;
    }
    -----------------
    ------------------
```

Example Program

```
/*  Sorting the n given number in Descending order */
#include<stdio.h>
void main( )
{
     int  a[ ] = {10 , 45 , 100, 37, 25};
     int  i, j, t;
     for (i=0; i<5; i++)
     {
          for (j=i+1; j<5; j++)
          {
               if (a[i] <= a[j] )
               {
                    t = a[i];
                    a[i] = a[j];
                    a[j] = t;
               }
          }
     }
     printf("Descending Order\n");
     for (i=0;i<5;i++)
     {
          printf("%d\n",a[i]);
```

```
        }
        getch( );
}
```

Output Of Program

```
Descending Order
100
45
37
25
10
```

2.4.5 Two-Dimensional Arrays

We may create a two-dimensional array as

```
type array_name  [row_size][column_size];
```

The **type** specifies the data type of element that will be contained in the array, such as int, float, or char and C place each size in its own set of brackets. Each dimension of the array is indexed from *zero to its maximum size minus one*:

- The first index selects the row and
- The second index selects the column within that row.

Example:

```
int    table[3] [3];
```

This creates a table that can store 9 integer values, three across (row) and three down (column).

	C1	C2	C3
R1			
R2			
R3			

The individual elements are identified by index or subscript of an array from the above **example**.

table[0][0]   table[0][1]   table[0][2]

table[1][0]   table[1][1]   table[1][2]

table[2][0]   table[2][1]   table[2][2]

2.4.6 Initializing Two Dimensional Arrays

A two-dimensional array may be initialized by following their declaration with a list of initial values enclosed in braces.

For **example**,

```
int table[2] [3] = {0,0,0,1,1,1};     //initialize the elements row by row
                 or
int table[2] [3] = {{0,0,0},{1,1,1}}; //separate the element of each row by braces
                 or
int table[2] [3] = {
                    {0,0,0},
                    {1,1,1}
                    };
```

If the values are missing in an initializer, they are automatically set to zero. For **example**:

```
int table[2] [3] = {
                    {1,1},
                    {2}
                    };
```

will initialize the first two elements of the row to one, the first element of the second row to two, and *all other elements to zero*.

Example Program: Program for Addition of Two Matrix using Two-Dimensional Array

```
/* Matrix Addition Using Two Dimensional Array */
#include<stdio.h>
void main( )
{
        int a[ 5 ][ 5 ] , b[ 5 ][ 5 ] ,c[ 5 ][ 5 ] ;
        int n = 0, i = 0, j = 0;
        printf("Enter the order of matrix");
        scanf("%d", &n);
        printf("\nEnter the A matrix");
        for(i=0; i<n; i++)
        {
            for(j=0; j<n; j++)
            {
                scanf("%d", &a[i][j] );
            }
        }
        printf("\nEnter the B matrix");
        for(i=0; i<n; i++)
        {
            for(j=0; j<n; j++)
            {
                scanf("%d", &b[i][j] );
            }
        }
        printf("\nThe A matrix\n");
```

```
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
         printf("%d\t", a[i][j]);
    }
    printf("\n");
}
printf("\nThe B matrix\n");
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
         printf("%d\t", b[i][j]);
    }
    printf("\n");
}

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
   {
          c[i][j] = a[i][j] + b[i][j];
    }
}
printf("\nThe Resultant Matrix is\n");
for(i=0; i<n; i++)
```

```
        {
          for(j=0; j<n; j++)
          {
              printf("%d\t", c[i][j]);
           }
          printf("\n");
        }
  getch( );
 }
```

Output Of Program

```
Enter the order of matrix
2
Enter the A matrix
2
2
2
2
Enter the B matrix
2
2
2
2
The A matrix
2    2
2    2
The B matrix
2    2
```

2   2

The Resultant Matrix is

4   4

4   4

2.4.7 Multidimensional Arrays

C allows arrays of three or more dimensions. The compiler determines the exact limit. The *general form* of a multidimensional array is

```
type array_name  [s1][s2]-------[sn];
```

where

s_n is the size of the nth dimension.

Example:

```
int d[3][5][12];
```

where

d is a three dimensional array declared to contain 180 integer type elements.

2.4.8 Dynamic Arrays

C language requires the number of elements in an array to be specifies at compile time. But we may not be able to do so always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. The process of allocating memory at **run time** is known as **dynamic memory allocation** or it is called as **dynamic arrays**.

Although C does not inherently have this facility, there are four library routines known as memory management functions that can be used for allocating and freeing memory during program execution. They are listed in Table 2.4.1.

Table 2.4.1 Memory Allocation Functions

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initialize them to zero and then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

Allocating a Block of Memory

A block of memory may be allocated using the function **malloc.** The malloc function reserves a block of memory of specified size and return a pointer of type void. This means that we can assign it to any type of pointer. The general form is

ptr = (cast-type *) malloc(byte-size);

Where

ptr - is a pointer of type cast-type.

malloc – returns a pointer to an area of memory with size byte-size.

Example:

x = (int *) malloc(100 * sizeof(int));

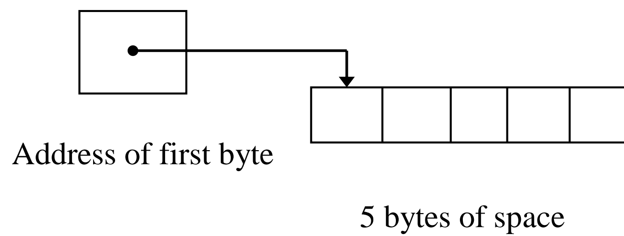
On successful execution of this statement, a memory space equivalent to "100 times the size of an int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of int.

Similarly, the statement

cptr = (char *) malloc(5);

allocates 5 bytes of space for the pointer cptr of type char. This illustrated below:

cptr



We may also use malloc to allocate space for complex data types such as structure.

Example

```
st_var = (struct store *) malloc(sizeof(struct store));
```

Where st_var is a pointer of type struct store.

Example Program: Program to print values from memory address

```
#include<stdio.h>
#include<stdlib.h>
main( )
{
    int *a, *n, size;
    printf("Enter the size..");
    scanf("%d", &size);
    n = (int *)malloc(size * sizeof(int));
    printf("Address of the first byte is %u \n", n);
    printf("Enter the values");
    for(a = n; a < n + size; a++)
        scanf("%d", a);
    printf("Printing the values \n");
    for(a = n; a < n + size; a++)
        printf("%d is stored in address %u\n", *a, a);
    getch( );
}
```

Output Of The Program

Enter the size..5

Address of the first byte is 2402

Enter the values 1 2 3 4 5

Printing the values

1 is stored in address 2402

2 is stored in address 2404

3 is stored in address 2406

4 is stored in address 2408

5 is stored in address 2410

Allocating Multiple Block of Memory

The calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. calloc allocates multiple block of storage, each of the same size, and then sets all bytes to zero. The general form of calloc is

```
ptr = (cast-type *) calloc(n, elem-size);
```

The above statement allocates contiguous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

Example:

```
-----------
-----------
struct student
{
        char name[20];
        float age;
        long int id_num;
```

```
};
typedef struct student rec;
rec * st_ptr;
int class_size = 30;
st_ptr = (rec *)calloc(class_size, sizeof(rec));
----------
----------
```

rec is of type struct student having three members: name, age and id_num. the calloc allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the st_ptr. For **example**

```
if(st_ptr = = NULL)
{
printf("Available memory not sufficient");
exit(1);
}
```

Releasing the Used Space

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With dynamic run time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. We may release that block of memory for future use, using the free function:

```
free(ptr);
```

ptr is a pointer to a memory block which has already been created by malloc or calloc.

Altering the Size of a Block

The previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much

larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with the help of the function realloc. This process is called as the reallocation of memory. For **example**

```
ptr = malloc(size);
```

Then reallocation of space may be done by the

```
ptr = realloc(ptr, newsize);
```

This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte os the new memory block. The newsize may be larger or smaller than the size.

Example Program: Program to altering the allocated memory

```
#include<stdio.h>
#include<stdlib.h>
main( )
{
        char *p;
        p = (char *)malloc(6);
        strcpy(p, “MADRAS”);
        printf(“Memory contains: %s\n”, p);
        p =(char *)realloc(p, 7);
        strcpy(p, “CHENNAI”);
        printf(“Memory now contains: %s\n”, p);
        free(p);
        getch( );
}
```

Output Of The Program

```
Memory contains: MADRAS
Memory now contains:CHENNAI
```

2.4.9 Self Assessment Questions

Fill in the blank

1. To put values into array created is known as _______________.

2. In two dimensional array the second index refers ________________.

True / False

1. Index value of arrays can begin with the number 1.

2. An array should be used to store dissimilar elements .

Multiple Choice

1. Dimension of the array is indexed from

a) zero to its maximum size b) one to its maximum size minus one c) one to its maximum size d) zero to its maximum size minus one

Short Answer

1. Define arrays.

2.5 Functions

2.5.1 Introduction

C functions can be classified into *two types*.

❖ **Library functions:**

Library functions are not required to be written by us. For **example** *printf* and **scanf** and other library functions are sqrt, cos, strcat etc.,

❖ **User-defined functions:**

A user-defined function has to be developed by user at the time of writing a program. For **example** *main* is a user-defined function.

2.5.2 Need for user defined functions

Every program must have a main function to indicate where the program has to begin its execution.

While it is possible to write any complex program under the main( ) function and it leads to a **number of problems**, such as

- The program becomes too large and complex
- The users can't go through at a glance.
- The task of debugging, testing and maintenance becomes difficult.

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These subprogram is called 'functions' are much easier to understand, debug, and test.

The **advantages** of using subprograms are

- The length of the source program can be reduced by dividing it into the smaller functions.
- By using functions it is very easy to locate and debug an error.
- The user defined functions can be used in many other source programs whenever necessary.
- Functions avoid coding of repeated programming of the similar instructions.
- Functions facilitate top-down programming approach.

How functions works?

- Once a function is called, it takes some data from the calling function and return back some value to the called function.
- Whenever function is called control passes to the called function and working of the calling function is temporarily stopped, when the execution of the called function is completed then control returns back to the calling function and execute the next statement.

- The function operates on formal and actual arguments and sends back the result to the calling function using return statement.

2.5.3 The form of C functions

The *general form* of functions is

```
datatype    function_name(argument list)
argument declaration;
{
local variable declarations;
-----------
-----------        // body of the function
-----------
return(expressions);
}
```

where

datatype – is the type of data that the function is going to return to its main program.

function_name – is the name of the function.

argument list - is the list of arguments that are transferred to the function from the main program.

The argument list must be separated by commas and has no termination (semi colon) after the parenthesis. The argument list and its associated argument declaration parts are optional.

The declaration of local variables is required only when any local variable are used in the function.

The function can have any number of executable statements. The function that does nothing may not include any executable statement at all.

For **example**

```
do_nothing( )
{ }
```

The **return** statement is used to for returning a value to the *calling function*. This is optional, when no value is being transferred to the calling function.

2.5.4 Return values and their types

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. While it is possible to pass to the called function any number of values, the called function can only return one value per call at the most.

The *general form* of return statement is

```
return;
or
return(expression);
```

The **first form** does not return any value; it acts much as the closing brace of the function. When return is encountered, the control is immediately passed back to the calling function.

For **example**

```
if(error)
return;
```

The **second form** of return with an expression returns the value of the expression. For **example**

```
add(x,y)                                add(x,y)
int x,y;                                int x,y;
{                                       {
      int   z;          or                    return(x + y) ;
      z = x + y;                        }
```

```
        return(z) ;
    }
```

A function may have *more than one return* statement. For **example**

```
if x < y
    retrun(x);
else
    return(y);
```

2.5.5 Calling a function

A function can be called by using the function name in a statement. For **example**

```
main()
{
    int a;
    a = add(10, 20);
    printf("%d\n", a);
}
```

When the compiler encounters a function call, the control is transferred to the function add(x, y). This function is then executed line by line and a value is returned when a return statement is encountered. This value is assigned to **a**.

A function that does not return any value may not be used in expressions; but can be called to perform certain tasks specified in the function.

2.5.6 Category of functions

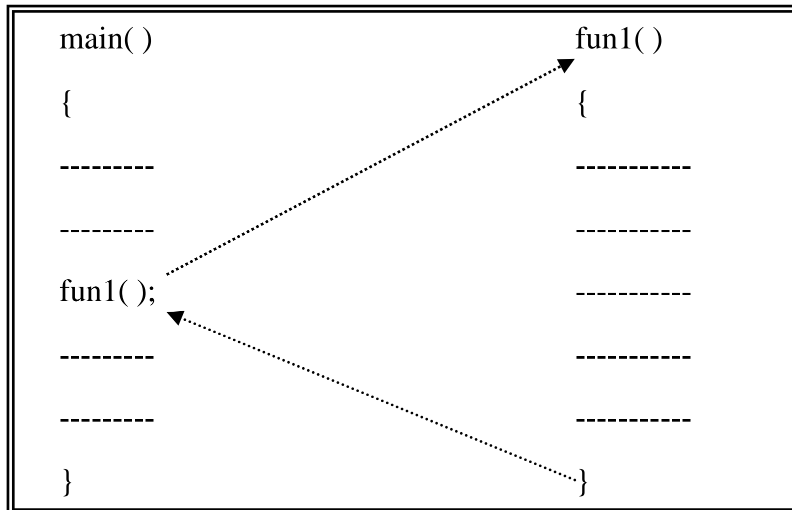
A function, depending on whether arguments are present or not and whether a value is returned or not, it can be classified into *four types* are

- Functions with no arguments and no return values
- Functions with arguments and no return values
- Functions with arguments and return values
- Functions with no arguments and return values

Functions with no arguments and no return values

In this type, *no data transfer takes place between the calling function and called function.* That is the called function does not receive any data from the calling function and does not send back any value to the calling function.

The *general form* is



Note:

The *dotted line* indicates that, there is only transfer of control but no data transfer. The *solid line or continuous* indicates that, there is transfer of data.

Example Program: Function call without parameter

```
#include<stdio.h>
main( )
{
        message( );
        printf("Main Message");
}

message( )
{
        printf("Function Message\n");
```

}

Output Of The Program

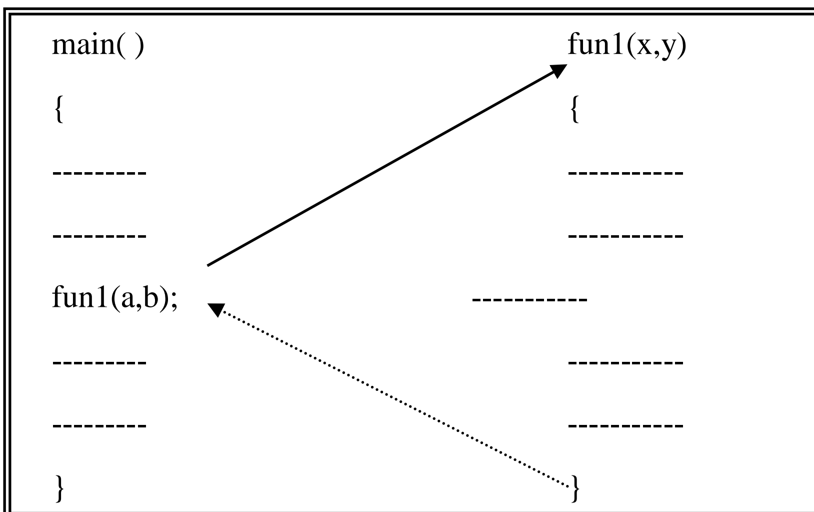
Function Message

Main Message

Functions with arguments and no return values

In this type, data is transferred *from calling function to called function*. That is the called function does receive some data from the calling function and does not send back any values to the calling function.(one way communication)

The *general form* is



Example Program: Function with arguments and no return values

```
#include<stdio.h>
main( )
{
        add( 10, 20);
        getch( );
}
add(int a, int b )
{
    int         c;
```

```
        c = a + b;
        printf("\nAddition of two number is :%d", c);
    }
```

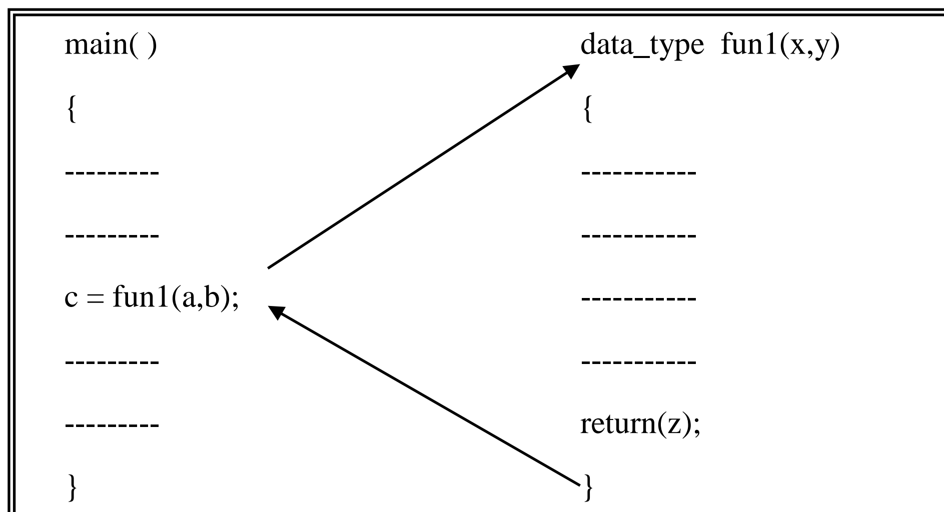
Output Of The Program

Addition of two numbers is :30

Functions with arguments and return values

In this type, *data is transferred between calling function and called function*. That is the called function does receive some data from the calling function and send back a value to the calling function.(one way communication)

The *general form* is



Example Program: Function with arguments and return values

```
#include<stdio.h>
main( )
{
        int x;
        x = add( 10, 20);
        printf("\nAddition of two number is :%d", x);
```

```
        getch( );
}
int add(int a, int b )
{
    int         c;
    c = a + b;
    return(c);
}
```

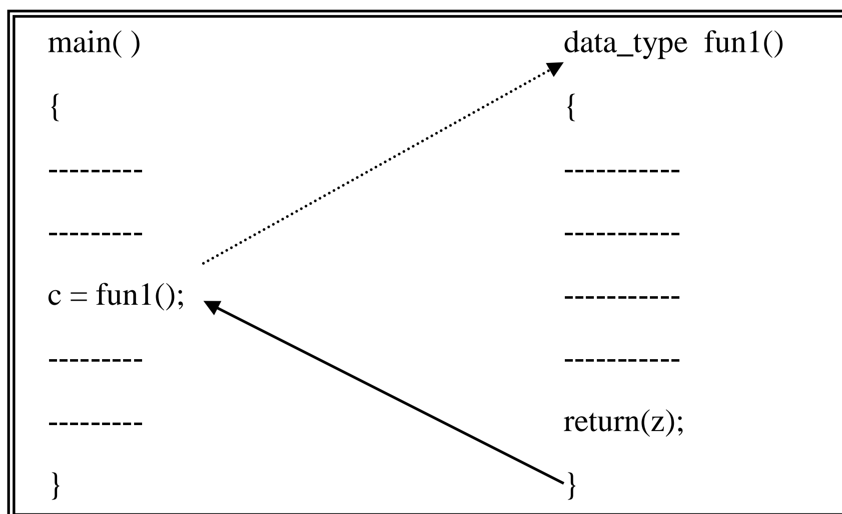
Output Of The Program

Addition of two numbers is :30

Functions with no arguments and return values

In this type, *calling function cannot pass any data to the called function but the called function may send some value to the calling function.*(one way communication)

The *general form* is



Example Program: Function with no arguments and return values

```
#include<stdio.h>
main( )
{
```

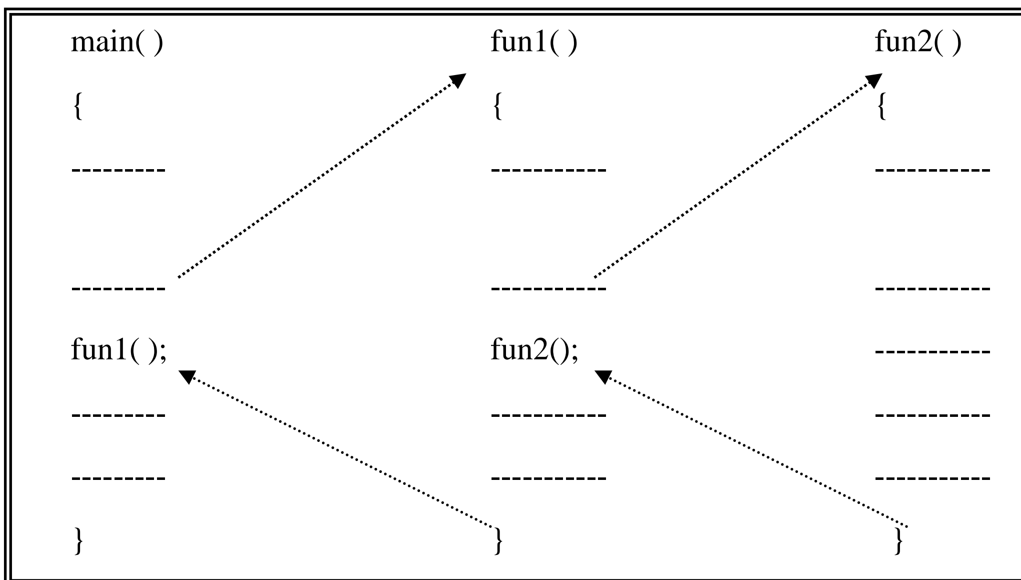
```
        int x;
        x = add( );
        printf("\nAddition of two number is :%d", x);
        getch( );
    }
    int add( )
    {
        int     a, b, c;
        a = 10;
        b = 20;
        c = a + b;
        return(c);
    }
```

Output Of The Program

Addition of two numbers is :30

2.5.7 Nesting of function

C permits to write *one function within another function* is called **nesting of functions.** **main**( ) function can call function1, which calls function2, ……… and so on. The *general form* is



2.5.8 Recursion

A *function calls itself* is called as recursion. In order to write a recursive program, the user must satisfy the following.

- The problem must be analyzed and written in recursive form.
- The problem must have stopping condition.

The **general form** is

```
function1( )
{
        function1( );
}
```

The above function the function1( ) is called itself continuously, so the above function is in recursive manner.

Example Program: Program to find the factorial of a given number.

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

```
        printf("\nEnter the number:");
        scanf("%d", &a);
        printf("The factorial of %d is : %d", a, recur(a));
        getch( );
}
recur(int x)
{
        int fact;
        if( x = = 1)
                fact = 1;
        else
                fact = x * recur(x – 1);
        return(fact);
}
```

Output Of Program:

```
Enter the number 5
The factorial of 5 is : 120
```

In the above program the variable **a** is read through the keyboard. The user defined function recur( ) is called from the main( ) function. Here the condition checked x = 1, if condition satisfied then control transfer to the main program and prints the value. Otherwise else part is executed and calculated the factorial value it returns the value to the main program. The function recur( ) is called as a recursive function.

2.5.9 Functions with arrays

In C, it is possible to pass the values of an array to a function. To pass an array to a called function, it is sufficient to list the name of the array without any subscripts and the size of the array as arguments. For **example**, the call

```
smallest( a , n);
```

will pass all elements contained in the array a of size n. the called function expecting this call must be appropriately defined. The **smallest** function header might look like

```
float smallest(array, size)
float array[ ];
int size;
```

The function smallest is defined to take *two arguments*, the array name and the size of the array to specify the number of elements in the array.

Example Program: Program to find smallest among n numbers

```
#include<stdio.h>
main( )
{
        int i, n, s, a[10];
        printf("\nEnter the value of n");
        scanf("%d", &n);
        printf("Enter the %d numbers of values", n);
        for(i = 0; i <n; i++)
            scanf("%d", &a[i]);
    s = smallest(n, a);
    printf("Smallest among %d number is :%d", n , s);
    getch( );
  }
  int smallest(int n , int x[])
  {
          int i, small =  x[0];
          for(i = 1; i < n; i++)
          {
```

```
                if( small > x[i] )
                        small = x[i];
        }
        return(small);
}
```

Output Of The Program

```
Enter the value of n
5
Enter the 5 numbers of values
45
37
28
68
40
Smallest among 5 number is : 28
```

2.5.10 Self Assessment Questions

Fill in the blank

1. Every called function must contain a ______________ statement.

2. A function calls itself is called as _________________.

True / False

1. The same variable names ca used in different functions without any conflict.

Multiple Choice

1. Each function must contain return statement, which returns

a) only one value b) three values

c) more than one value d) none of the above

Short Answer

1.What is meant by functions with no arguments and no return values?

2.6 Summary

The first lesson of this unit you have learnt the use of structures, which allow us to combine several variables of different types into a single entity. We have also examined the features of structures, which lead to programming convenience.

The second lesson of this unit you have learned how to handle arrays in a variety of forms. We have learnt how to declare arrays, how to initialize them and how to access array elements using subscript.

The third lesson of this unit you have learnt how to use function, how to write them, how to functions interact with one another how to send them information using arguments and how to use them to return values. You also learnt how recursive functions work.

2.7 Unit Questions

1. Write a program to print student name and marks using structure.
2. Compare structure and union in C.
3. Explain structure concept in C with suitable example.
4. Define array. Explain different types of arrays with example.
5. Write a program to multiple two matrices using two-dimensional array.
6. Write a program to sort the given n numbers in descending order using array.
7. Discuss the categories of functions used in C.
8. What is called recursion? Explain it needs.
9. Write a program to generate fibonacci series upto n numbers using recursion.

10.Write a program, which performs the following tasks:

a) initialize an integer array of 10 elements in main( )

b) pass the entire array to a function modify( )

c) in modify( ) multiply each element of array by 2

d) return the control to main( ) and print the new array elements in main( )

2.8 Answers for Self Assessment Questions

Answer 2.3.13

Fill in the blank

1. struct keyword 2. dot (.) operator

True / False

1. True 2. False

Multiple Choice

1. True 2. False

Short Answer

1. After declaring the structure type, variables and members, the member of the structure can be accessed by using the structure variable along with the dot(.) operator.

2. Unions are a same concept of structures and therefore follow the *same syntax as structures.* However, there is major distinction between them in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location

Answer 2.4.9

Fill in the blank

1.initialization 2. column

True / False

1. False 2. False

Multiple Choice

1. d)

Short Answer

1.An **array** is a group of related data items that *share a common name*. An array of value can be accessed by *index or subscript* enclosed in square brackets after array name.

Answer 2.5.9

Fill in the blank

1. return statement 2. recursion

True / False

1. True

Multiple Choice

1. a)

Short Answer

1. No data transfer takes place between the calling function and called function.

NOTES

UNIT – III

3.1. Introduction

In this unit we introduce character arrays & string, pointers and file concepts in C language. An array of character is called character array or string. Character strings are often used to build meaningful and readable programs. We shall discuss about the common operation performed on character strings.

Pointer is a important feature of C language. In the second lesson of this unit, we examine the pointers in details and learn how to use them in program.

In the last lesson of this unit, we shall discuss about file concept in detail.

3.2. Objectives

After studying this unit, you should be able to:

- Understand string variables and the utilization of the NULL character (\0)
- Understand the various string handling functions.
- Understand the concept of a pointer and create pointers to each type of variable, to character strings, to functions and to structures.
- Open files in three modes (read, write and append), understand where each mode places the file pointer and appreciate how each affects the data previously contained in the file.
- Understand the various file handling functions and command line arguments concept in C.

3.3. Character Arrays & Strings

3.3.1 Introduction

A string is an *array of characters*. Any group of characters defined between double quotation marks (" ") is a string constant.

Example

“welcome”

If we want to include a double quote in the string, then we may use it with a back slash. For **example**

printf(“\”welcome !\” ”);

will output the string

”welcome !”

The *common operations* performed on character strings are:

- Reading and Writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

3.3.2 Declaring And Initializing String Variables

A string variable is any valid C variable name and is always declared as an array. The *general form* of declaration of a string variable is

char string_name[size];

The size determines the number of characters in the string_name.

Example

char name[20];

char dept[10];

When the compiler assigns a character string to a character array, it automatically supplies a null character (‘\0’) at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the *two forms*:

char name[5] = ”Raja”;

```
char name[5] = {'R','a','j,'a','\0'}
```

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized.

Example

```
char string [ ] = {'R','a','j,'a','\0'}
```

Defines the array string as a *five elements* array.

3.3.3 Reading Strings From Terminal

Reading words

The input function **scanf** can be used with *%s format specification* to read in a string of characters. **Example:**

```
char address[15];
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds (A white space includes blanks, tabs, carriage returns, form feeds and new lines). Therefore, if the following line of text is typed in at the terminal,

NEW YORK

Then only the string "NEW" will be read into the array address, since the blank space after the word "NEW" will terminate the string.

Example Program: Program for reading words from terminal.

```
#include<stdio.h>
void main( )
{
    char address[50];
    printf("Enter the address\n");
    scanf("%s", address);
    printf("The address is %s", address);
```

```
        getch( );
    }
```

Output Of Program

```
Enter the address
NEW YORK
The address is NEW
```

Reading a line of text

We can use **getchar** function repeatedly to successive single characters from input and place them into a character array. Thus, an entire line of text can be red and stored in an array. The reading is terminated when the new line character (‘\n’) is entered and the null character is then inserted at the end of the string.

Example Program: Program for reading words from terminal.

```
#include<stdio.h>
void main( )
{
    char address[50], ch;
    int n = 0;
    printf(“Enter the address\n”);
    do
    {
        ch = getchar( );
        address[n] = ch;
        n++;
    } while (ch ! = ‘\n’);
    n = n – 1;
    address[n] = ‘\0’;
```

```
        printf("The address is %s", address);
        getch( );
}
```

Output Of Program

```
Enter the address
NEW YORK
The address is NEW YORK
```

3.3.4 Writing Strings To Screen

The **printf** function with %s format is used to print strings to the screen. The format **%s** can be used to display an array of characters that is terminated by null character.

For **example**

```
printf("%s", name);
```

can be used to display the entire contents of the array name.

We can also specify the precision with which the array is displayed. For **example**, the specification

%10.4

indicates that the first four characters are to be printed in a field width of 10 columns.

Example Program: Program for display the string under various format specifications.

```
#include<stdio.h>
void main( )
{
        char address[]  = "NEW YORK";
        printf("\nThe address is\n ");
        printf(" %10s\n", address);
        printf(" %4s\n", address);
```

```
printf(" %10.5s\n", address);
getch( );
}
```

Output Of Program

```
Enter the address
NEW YORK
The address is
   NEW YORK
 NEW YORK
          NEW Y
```

3.3.5 String Handling Functions

The C library supports a large number of string handling functions that can be used to manipulate the strings.

strlen ( ) Function

The **strlen ( )** function is used to count and return the number of characters in the string. The *general form* is

var = strlen(string);

where

var - is the integer variable, which accept the length of the string.

string - is the string constant or string variable in which the length is going to be found. The counting ends with first null ('\0') character.

strcpy ( ) Function

The **strcpy ( )** function is used to copy the contents of one string to another and almost works like string assignment operator. The *general form* is

strcpy(string1, string2);

where

string1 and string2 are character array variable or string constant. It will assign the contents of string2 to string1.

For **example**

strcpy(name, “Harsh”);

will assign the string “Harsh” to the string variable name.

strcpy(name1, name2);

will assign the contents of the string variable name2 to the string variable name1. The size of the array name1 should be large enough to receive the contents of name2.

strcmp ( ) Function

The **strcmp ( )** function is used to compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first non matching characters in the strings.. The *general form* is

strcmp(string1, string2);

where

string1 and string2 are character array variable or string constant.

For **example**

strcmp(name1, name2);

strcmp(name1, “John”);

strcmp(“Raja”, “raja”);

strcat ( ) Function

The **strcat ( )** function is used to combine or join or concatenate two strings together. The general form is

strcat(string1, string2);

where

string1 and string2 are character array variable or string constant.

When the function strcat is executed, string2 is appended to string1.

For **example**

strrev ( ) Function

The strrev ( ) function is used to reverse a strings . The general form is

strrev(string1);

where

string1 is character array variable or string constant.

3.3.6 Self Assessment Questions

Fill in the blank

1. A string is an _____________ of characters.

2. The ___________function is used to copy the contents of one string to another.

True / False

1. The length( ) function is used to find the string length.

2. When the compiler assigns a character string to a character array, it automatically supplies a null character (‘\0’) at the end of the string.

Multiple Choice

1.Which is more appropriate for reading in a multi-word string?

a) gets( )		b) printf( )		c) scanf( )		d) puts( )

2. The array char name[10] can consist of maximum of

a) 10 characters		b) 9 characters

c) 11 characters		d) none of the above

Short Answer

1. What is the purpose of strrev( ) function?

Pointers

o Introduction

Generally, computer uses memories for storing instruction and values of the variables with in the program, but the **pointers** have *memory address* as their values. The memory address is the location where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

The computer's memory is a sequential collection of storage cells as shown in Fig. In, computer's memory each cell is one byte, and it has a number called address, this address is numbered consecutively starting from zero to last address (depends on the memory size).

While declaring variable, the computer allocates appropriate memory to hold the value of the variable. Since every memory has a unique address, this address location will have its own address number somewhere in the memory area.

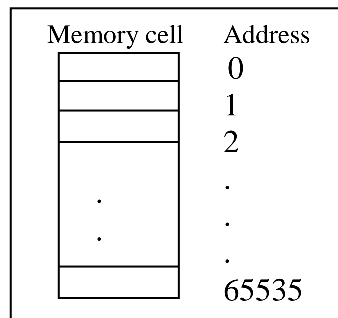
o Understanding pointers

The computer's memory is a sequential collection of storage cells as shown in Fig.3.4.1. Each cell, commonly known as a byte, has a number called address associated with it. The addresses are numbered consecutively starting from zero. The last address depends on memory size. For **example** *64K memory* will have its last address as *65535*.

Whenever declare variable the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Consider following statement

```
int  x = 100;
```

This statement instructs the system to find a location for integer variable **x** and puts the value 100 in that location.(See Fig 3.4.2)



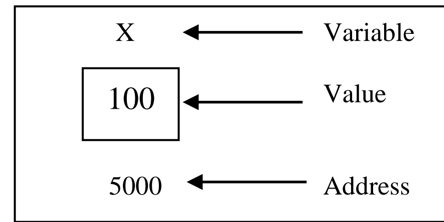


Fig. 3.4.1 Memory Organization Fig. 3.4.2 Representation of a variable

Memory addresses are simply numbers; they can be assigned to some variables, which can be stored in memory, like any other variable. Such variable that hold memory addresses are called **pointers**.

Suppose, we assign the address of **x** to a variable p. the link between the variable p and x can be visualized as shown in Fig.3.4.3.

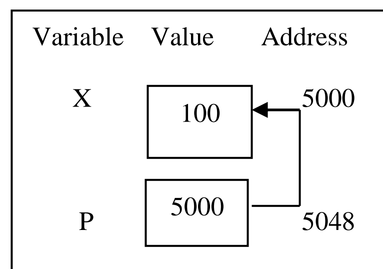


Fig. 3.4.3 Pointer as a variable

- **Accessing the address of a variable**

The operator **&** immediately preceding a variable returns the address of the variable associated with it. For **example**, the statement

p = &q;

would assign the address 4000(the location of q) to the variable p. the & operator can remembered as 'address of'.

The & operator can be used only with a simple variable or an array element. The following are **illegal use** of address operator:

1. & 450 (pointing at constant)
2. int x[10];

& x (pointing at array names)

3.&(x+y) (pointing at expressions)

Example Program: Program for accessing addresses of variables

```
#include<stdio.h>
main( )
{
    int a;
    char c;
    float f;
    a = 100;
    c = 'A';
    f = 20.56;
    printf("%d is stored at address %u\n", a, &a);
    printf("%c is stored at address %u\n", c, &c);
    printf("%f is stored at address %u\n", f, &f);
    getch( );
}
```

Output Of The Program

```
100 is stored at address 65494
A is stored at address 65497
20.559999 is stored at address 65498
```

- **Declaring and initializing pointer**

Declaration

The *general form* for declaration of a pointer variable is

```
data_type    *pointer_name;
```

where

data_type – is specifies the type of data to which the pointer points

asterisk (*) – that tells the variable pointer_name is a pointer variable

pointer_name- is specifies the name of the pointer.

For **example**

```
int  *p;     //declares the variable p as a pointer variable that points to an
                  integer data type.
float  *f         //declares x as a pointer to a floating point variable.
char  *c          //declares c as a pointer to a character variable
```

Initialization:

Once a pointer variable has been declared, it can be made to point to a variable using an *assignment statement* such as

```
p = &q;
```

which causes p point to q. That is, p now contains the address of q. This is known as **pointer initialization**. Before a pointer is initialized, it should not be used.

The pointer variable can be initialized in its declaration itself. For **example**

```
int  x,  *p = &x;          //valid initialization
```

It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x.

- **Accessing a variable through its pointer**

Once the pointer is declared and assigned to the address of another variable, the variable can be accessed through its pointers. This is done by using another unary operator *(asterisk), usually known as the indirection operator. Another name for the indirection operator is the de-referencing operator.

For **example**

```
int  *p , x, a;   // declares two integer variable x & a and one pointer
                     variable p
x  = 15;          // assigns the value 15 to variable x
p = &x;           // assigns the address of x to the pointer variable p
a = *p;           // it returns the value of the variable x, because p
                     contains the address of the variable x
```

Another **example**

The statement p = &x assigns the address of x to p and y = *p assigns the value pointed to by the pointer p to y.

```
*p = 25;
```

This statement puts the value 25 at the memory location whose address is the value of p. we know that the value of p is the address of x and therefore the old value of x is replaced by 25.

Example Program: Program for accessing variables using pointers

```
#include<stdio.h>
main( )
{
    int a, b;
    int *ptr;
    a = 30;   ptr = &a;
    b = *ptr;
    printf("Value of a is %d\n", a);
    printf("%d is stored at address %u\n", a , &a);
    printf("%d is stored at address %u\n", *&a , &a);
    printf("%d is stored at address %u\n", *ptr , ptr);
    printf("%d is stored at address %u\n", b , &*ptr);
    getch( );
```

}

Output Of The Program

Value of a is 30

30 is stored at address 65502

30 is stored at address 65502

30 is stored at address 65502

30 is stored at address 65502

- **Pointer expressions**

Pointer variables can be used in expressions. Here, the pointers are preceded by the * (indirection operator) symbol.

Example

```
c = *a  +  * b; can be written as       c = (*a)  + (*b);
s = s  +  *a/ * b;        can be written as       s = s + (*a) / (*b);
*p =  *a**b;              can be written as       *p = (*a) * (*b);
```

Note that there is a blank space between / and * in above expression. The following is wrong.

```
s = s  +  *a/* b;
```

The symbol /* is considered as the *beginning of a comment* and therefore the statement *fails*. So we leave the space between / and * operator.

C allows us to **add or subtract** integers from *one pointer to another*. In addition to that the pointer can also be compared using relational operators. But comparisons can be used mainly in handling arrays and strings.

Example

```
p1  +  4        p2  - 2        and    p1 – p2        are valid
p1 > p2         p1 = = p2      and    p1 ! = p2      are valid
```

We may also use shorthand operators with pointers.

Example

```
p1 + +;
```

```
- - p2;
s  +=  *p2;
```

We may not use pointers in division and multiplication.

Example

p1 / p2   p1 * p2   p1 / 3   p2*6   are invalid

Similarly, two pointers *cannot* be added. That is, p1 + p2 is illegal.

Example Program: Program for pointer expression

```
#include<stdio.h>
main( )
{
    int a, b , x, y;
    int *ptr1, *ptr2;
    a = 10; b = 3;
    ptr1 = &a;
    ptr2 = &b;
    x =  *ptr1 + *ptr2 – 2;
    y =  *ptr1 * *ptr2 + 1;
    printf(“Address of a is %u\n”, ptr1);
    printf(“Address of b is %u\n”, ptr2);
    printf(“a = %d, b = %d\n”, a, b);
    printf(“x = %d, y = %d\n”, x, y);
    *ptr2 = *ptr2 + 3;
    *ptr1 = *ptr2 * 2;
    printf(“a = %d, b = %d\n”, a, b);
    printf(“x = %d, y = %d\n”, x, y);
    getch( );
}
```

Output Of The Program

Address of a is 65496

Address of b is 65498

a = 10, b = 3

x = 11, y = 31

a = 12, b = 6

x = 11, y = 31

- **Pointers and array**

Array is a collection of similar data items, which is stored under *common name*. When an array is declared, the compiler allocates a base address and sufficient amount of storage locations for all elements of an array. The base address is the location of the first element (index 0) of the array.

The compiler also defines the array names as a constant pointer to the first element.

For **example**

int x[5]   ={1, 2, 3, 4, 5};

Suppose the base address of x is 2000 and assuming that each integer requires 2 bytes, the five elements will be stored as

The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 2000, the location where x[0] is stored. That is

x = &x[0] = 2000

If we declare p as integer pointer, then we can make the pointer p to point to the array x by the assignment

p = x;

This is equivalent to

p = &x[0];

Now, we can access every value of x using p++ to move from one element to another. The relationship between p and x is

p = &x[0]   (=2000)

p+1 = &x[1]   (=2002)

p +2= &x[2]   (=2004)

p +3= &x[3]   (=2006)

p +4= &x[4]   (=2008)

The address of an element is calculated by using its index and the scale factor of the data type. For **example**

Address of x[2] = base address + (2 x scale factor of int)

= 2000 + (2 x 2)

= 2004

When handling arrays, instead of using array indexing, we can use pointers to access array elements.

For **example**

*(p + 2) gives the value of x[3].

This is much faster than array indexing.

Example Program: Program to print the array values and address of the array elements using pointer

```
#include<stdio.h>
main( )
{
    int arr[5] = { 10, 20, 30, 40, 50};
    int i, *ptr_arr;
    ptr_arr = arr;
    for(i = 0; i < 5; i++)
    {
        printf("Address = %u\t", &arr[i]);
        printf("Element = %d\t", arr[i]);
```

```
                printf(%d\t", *(arr+i));
                printf("%d\t", ptr_arr[i]);
                printf("%d\t\n", *(ptr_arr+i));
            }
            getch( );
        }
```

Output Of The Program

```
Address = 65494 Element = 10   10   10   10
Address = 65496 Element = 20   20   20   20
Address = 65498 Element = 30   30   30   30
Address = 65500 Element = 40   40   40   40
Address = 65502 Element = 50   50   50   50
```

In the above program, the variable arr[] is declared as integer array, the elements are displayed using different syntax. 1. arr[i], 2. *(arr + i), 3. ptr_arr[i] , 4. *(ptr_arr + i)

Pointers can be used to manipulate two-dimensional arrays. An element in a two-dimensional array can be represented by the pointer expression as

((a+i)+j)   or   *(*(p+i)+j)

The base address of the array a is &a[i][j] and starting at this address, the compiler allocates contiguous space for all the elements, row-wise. That is, the first element of the second row is placed immediately after the last element of the first row, and so on.

- **Pointers and character strings**

A string is an array of characters, terminated with a null character. Like one-dimensional arrays, we can use a pointer to access the individual characters in a string.

Example Program:

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

```
main( )
{
    char *s1 = "abc";   char s2[ ] = "xyz";
    printf("%s %16lu \n", s1, s1);
    printf("%s %16lu \n", s2, s2);
    s1 = s2;
    printf("%s %16lu \n", s1, s1);
    printf("%s %16lu \n", s2, s2);
    getch( );
}
```

Output Of The Program

```
abc     12976536
xyz     13041628
xyz     13041628
xyz     13041628
```

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *city;
city = "Salem";
```

These statements will declare city as a pointer to character and assign to city the constant character string "Salem".

One important use of pointers is in handling of a table of strings. For **example** array of strings:

```
char name[3][20];
```

This array that the **name** is a table containing three names, each with a maximum length of 20 characters (including null character). Total storage requirements for the name table are 60.

Instead of making each row with fixed number of characters, we can make it a pointer to a string of varying length. For **Example**

```
char *city[3] = {    "Chennai"
                     "Covai"
                     "Salem"
                };
```

declares city to be an array of three pointers to characters, each pointer pointing to a particular city as shown below:

```
city[0] -> Chennai
city[1] -> Covai
city[2] -> Salem
```

The following statement would print out all three cities:

```
for(i = 0; i< 3; i++)
printf("%s\n", city[i]);
```

To access the jth character in the ith name, we may write as

```
*(city[i] + j)
```

- **Pointer and functions**

When we pass addresses to a function using pointers to pass the addresses of variable is known as *call by reference*. The function, which is called by reference can change the value of the variable used in the call.

Example Program: Program for pointer and function

```
main()
{
        int   x;
        x = 20;
        change(&x);
        printf("%d\n", x);
```

```
        getch( );
}
change(p)
int *p;
{
        *p = *p +10;
}
```

Output Of The Program

30

When the function change( ) is called, the address of the variable x, not its value, is passed into the function change( ). Inside change( ), the variable p is declared as a pointer and therefore p is the address of the variable x. The statement

*p = *p +10;

means ‘add 10 to the value stored at the address p’. Since p represents the address of x, the value of x is changed from 20 to 30.

Pointer to Functions

A function, like variable, has an address location in the memory. To declare a pointer to a function, which can be used as an argument in another function. The *pointer to a function* is declared as

type (*fprt) ( );

This tells the compiler that fprt is a pointer to a function, which returns type value.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

For **example**

```
double (*p1) ( ), mul( );
p1 = mul;
```

declare p1 as a pointer to a function and mul as a function and then make p1 to point to the function mul.

To call function mul, we may use the pointer p1 with the list of parameters. That is

```
(*p1) (x, y)
```

is equivalent to

```
mul(x, y)
```

- **Pointer and structures**

The *general form* is

```
struct structure_name
{
        type   structure_element 1;
        type   structure_element 2;
        ------------
        type   structure_element n;
} struct_var , *ptr;
```

where struct_var is the structure type variable and ptr as a pointer to data objects of the type struct structure_name.

For **example**

```
struct student
{
    int      student_rno;
    char     student_name;
    int      student_mark1;
    int      student_mark2;
} s1[2] ,  *s2 ;
```

This statement declares s1 as an array of two elements, each of the type struct student and s2 as a pointer to data objects of the type struct student.

The assignment

```
s2 = s1;
```

would assign the address of the zeroth element of s1 to s2. That is pointer s2 will now point to s1[0]. Its members can be accessed using the following notation

```
s2 - > student_name
s2 - > student_rno
s2 - > student_mark1
s2 - > student_mark2
```

The symbol -> is called the arrow operator and is made up of a minus sign and a greater than sign.

We also use the notation

```
(*s2).rno
```

to access the member rno. The parentheses around *s2 are necessary because the member operator “. “ has a higher precedence than the operator *.

While using structure pointers, we should take care of the precedence of operators.

For **example**

```
++ p -> c;
```

increments c, not p. however,

```
(++p) -> c;
```

increments p first, and then links c.

Example Program: Program for pointers to structure variables

```
#include<stdio.h>
main( )
{
```

```
struct student
{
        int      rno;
        char    *name[20];
        int      mark;
} *s2 ;
printf("Enter student Roll no., Name, Mark\n");
scanf("%d%s%d", &s2->rno, s2->name, &s2->mark);
printf("Student Roll no.:%d", &s2->rno);
printf("Student Name:%s",  s2->name);
printf("Student Mark:%d", &s2->mark);
getch( );
}
```

Output Of The Program

```
Enter student Roll no., Name, Mark
2000
Harshinni
100
Student Roll no.:2000
Student Name:Harshinni
Student Mark:100
```

- **Self Assessment Questions**

Fill in the blank

1. The address of an element is calculated by using its ______________ and the scale factor of the _______________.

2.When we pass addresses to a function using pointers to pass the addresses of variable is known as ___________________.

True / False

1. Address of a floating-point variable is always a whole number.

Multiple Choice

1.Which of the following is the correct way of declaring a float pointer

a) float ptr; b) float *ptr;

c) *float ptr; d) none of the above

2. If int s[5] is a one dimensional array of integers, which of the following refers to the third element in the array?

a) *(s+2) b) *(s+3) c) s+ 3 d) s+2

Short Answer

1. What is mean by pointer expression?

----

2. What is the purpose of using & operator in pointers?

3.4. Files

3.4.1 Introduction

A **file** is a collection of related information, that is permanently stored on the disk and allows us to access and alter the information whenever necessary. We can perform the following *basic operations* on files.

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Closing a file

There are large numbers of standard library functions available for *performing disk files*.

- High level file I/O functions
- Low level file I/O functions

High-level file I/O functions do their own buffer management, whereas in low level file I/O functions buffer management has to be done by the programmer. Some of the high level file I/O functions listed in Table.

3.4.2 Opening/Closing Files

Defining a file

We want to store data in a file in the secondary memory; we must specify the following things about file, to the operating system. They include

- **File name**

  Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and optional period with the extension. For **example**

  Input.data

  store

  stud.c

  Empl.out

- **Data structure**

  Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type.

- **Purpose**

When we open a file, we must specify what we want to do with the file. For **example**, we may write data to the file or read the already exiting data.

Opening a file

The *general form* for declaring and opening a file is

```
FILE    *fp;

fp  =  fopen("filename, "mode");
```

The **first statement** declares the variable fp as a "pointer to the data type FILE". The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all information about the file and it, is used as a communication link between the system and the program.

The **second statement** also specifies the purpose of opening this file. The mode does this job. **Mode** can be one of the following

- r open the file for reading only.
- w open the file for writing only.
- a open the file for appending or adding data to it.

The filename and mode is treated as *string*, so it should be enclosed in double quotation marks (" ").

When trying to open a file, one of the *following things* may happen.

- When the mode is ***'writing'***, a file with specified name is created if the file does not exist. The contents are deleted, if the file already exists.
- When the purpose is ***'appending'***, the file is opened with current contents safe. A file with specified name is created if the file does not exist.
- If the purpose is ***'reading'*** and if it exists, then the file opened with current contents safe; otherwise an error occurs.

Example

```
FILE  *p1, *p2;

p1 = fopen("stud.out", "r");

p2 =fopen("results", "w");
```

The file *stud.out* is opened for reading and a result is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If *stud.out* file does not exist, an error will occur.

Closing a file

A file must be closed after all the operation of the file has been completed. The *general form* for closing a file is

```
fclose(file_pointer);
```

This would close the file associated with the FILE pointer file_pointer.

Example

```
-----------
-----------
FILE *p1;
p1 = fopen("input", "w");
-----------
-----------
fclose(p1);
-----------
```

This program opens a file and closes a file after all operations on the files are completed. Once a file is closed, its file pointer can be reused for another file. All files are closed automatically whenever a program terminates.

3.4.3 Input/Output files

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table.

The getc and putc functions

The function **getc** is used to read a character from a file that has opened with read mode r.

Example

```
c = getc(fp1)
```

would read a character from the file whose file pointer is fp1.

The function **putc** is used to write the character contained in the character variable to the file associated with FILE pointer that has been opened with write mode w.

Example

```
putc(c, fp2);
```

would write a character contained in the variable c to the file associated with FILE pointer fp2.

The file pointer moves by one character position for every operation of **getc** or **putc.** The **getc** will return an end-of-file marker **EOF**, when end of the file has been reached. Therefore the reading should be terminated when **EOF** is encountered

Example Program: Program for writing to and reading from a file.

```
#include<stdio.h>
main( )
{
    FILE *f1;
    char c;
    printf("Data input\n");
    f1 = fopen("Test", "w"); /* Open the file Test*/
    while((c = getchar( ) ) != EOF) /* Get a character from keyboard */
        putc(c, f1);                /* Write a character to Test file  */
    fclose(f1);                     /* Close the file Test*/
    printf("Data Output\n");
    f1 = fopen("Test", "r");
    while((c = getc(f1)) != EOF)    /* Read a character from Test file
        */
        printf("%c", c);            /* Display a character on screen */
```

```
    fclose(f1);
    getch( );
}
```

Output Of The Program

```
Data Input
Dog
Cat
Rat
^Z
Data Output
Dog
Cat
Rat
```

We enter the input data via keyboard and the program writes it, character by character, to the file Test. Entering an **EOF** character, which is *control-Z*, indicates the end of the data. The file Test is closed at this signal.

The file Test is reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when **getc( )** encounters the end-of-file mark **EOF**.

The getw and putw functions

The **getw** and **putw** are *integer-oriented* functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. The *general forms* of **getw** and **putw** are

```
putw(integer, fp);
getw(fp);
```

Example Program: Program to illustrate the use of **putw** and **getw** functions.

```
#include<stdio.h>
main( )
```

```
{
        FILE *f1, *f2, *f3;
        int number, i;
        printf(“Contents of Data file \n”);
        f1 = fopen(“DATA”, “w”);
        for(i = 1; i <=20; i++)
        {
                scanf(“%d”, &number);
                if( number = = -1) break;
                putw(number, f1);
        }
        fclose(f1);
        f1 = fopen(“DATA”, “r”);
        f2 = fopen(“ODD”, “w”);
        f3 = fopen(“EVEN”, “w”);
        while((number = getw(f1))  != EOF)
        {
                if (number % 2 == 0)
                putw(number, f3);
                else
                putw(number, f2);
        }
        fclose(f1);
        fclose(f2);
        fclose(f3);
        f2 = fopen(“ODD”, “r”);
        f3 = fopen(“EVEN”, “r”);
        pintf(“\nContents of ODD file\n”);
        while((number = getw(f2)  != EOF)
                printf(“%4d”, number);
        pintf(“\nContents of EVEN file\n”);
        while((number = getw(f3)  != EOF)
                printf(“%4d”, number);
        fclose(f2);
```

```
        fclose(f3);
        getch( );
    }
```

Output Of The Program

Contents of Data file

1 2 3 4 5 6 7 8 9 10 -1

Contents of ODD file

1 3 5 7 9

Contents of EVEN file

2 4 6 8 10

The fprintf and fscanf functions

These functions are used to handle a group of mixed data simultaneously. The functions **fprintf** and **fscanf** perform I/O operations those are identical to the familiar **printf** and **scanf** functions except that they work on files. The *general form* of **fprintf** is

```
fprintf(fp, “control string”, list);
```

where

fp - is a file pointer associated with a file that has been opened for writing.

control string - contains output format specifications for items in the list.

list - may include variables, constants and strings.

Example

```
fprintf(fp1, “%s%d”, name, age);
```

Here, name is an array variable of type char and age is an int variable.

The *general form* of **fscanf** is

```
fscanf(fp, “control string”, list);
```

would cause the reading of the items in the list from file specified by fp, according to the specifications contained in the control string.

Example

```
fscanf(fp2, “%s%d”, name, &age );
```

fscanf returns the number of items that are successfully read. When the end of file is reached, it returns the value **EOF**.

Example Program: Program to Handling of files with mixed data types

```
#include<stdio.h>
main( )
{
        FILE *f1;
        int rno, i;
        char name[20];
        f1 = fopen(“STUDENT”, “w”);
        printf(“Input Student data\n\n”);
        printf(“Roll No.        Name\n”);
        for(i = 0; i < 3; i++)
        {
                fscanf(stdin, “%d%s”, &rno, name);
                fprintf(f1, “%d%s\n”, rno, name);
        }
        fclose(f1);
        fprintf(stdout, “\n\n”);
        f1 = fopen(“STUDENT”, “r”);
        printf(“Output Student data\n\n”);
        printf(“Roll No.        Name\n”);
        for(i = 0; i < 3; i++)
        {
                fscanf(f1, “%d%s”, &rno, name);
                fprintf(stdout, “%d\t\t%s\n”, rno, name);
        }
        fclose(f1);
        getch( );
}
```

Output Of The Program

```
Input Student data
Roll No.  Name
1000    aaa
```

2000   bbb

3000   ccc

Output Student data

Roll No.   Name

1000   aaa

2000   bbb

3000   ccc

In the above program the data is read using function **fscanf** from the file **stdin**, which refers to the terminal and it is then written to the file that is being pointed to by file pointer **f1**.

After closing the file STUDENT, it again reopened for reading. The data from the file, along with the item values are written to the **stdout**, which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file.

3.4.4 Error Handling During I/O Operations

It is possible that an error may occur during I/O operations on a file. Typical *error situations* are

- Trying to read beyond the end-of-file mark.
- Device overflow.
- Trying to use a file that has not been opened.
- Trying to perform an operation on a file, when the file is opened for another type of operation.
- Opening a file with an invalid filename.
- Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output.

C library has *two functions*, **feof** and **ferror** that can help us to detect I/O errors in the files.

feof function

The **feof** function can be used to *test for an end of file condition*. It takes a **FILE** pointer as its only argument and return a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If fp is a file pointer to file that has just been opened for reading, then *the statement*

```
if(feof(fp))
printf("end of data");
```

would display the message "end of data" on reading the end of file condition.

ferror function

The **ferror** function reports the status of the file indicated. It also takes a **FILE** pointer as it's argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. *The statement*

```
if(ferror(fp))
printf("an error has occurred");
```

would print the error message, if reading is not successful.

We know that whenever a file is opened using fopen function, afile pointer is returned. If the file cannot be opened for some reason, then the function returns a null pointer.

Example

```
if(fp = NULL)
printf("file could not be opened");
```

Example Program: Program to illustrate error handling in file operations.

```
#include<stdio.h>
main( )
{
        char filename[20];
        FILE *f1, *f2;
        int i, number;
        f1 = fopen("TEST", "w");
        for(i = 10; i<= 100; i +=10)
                putw(i, f1);
        fclose(f1);
        printf("\nInput File name\n");
        open_file:      scanf("%s", filename);
        if((f2 = fopen(filename, "r")) == NULL)
```

```
        {
                printf("Cannot open the file \n");
                printf("Type filename again\n");
                goto open_file;
        }
        else
        for(i = 1; i <= 20; i++)
        {
                number = getw(f2);
                if(feof(f2))
                {
                        printf("Ran out of data\n");
                        break;
                }
                else
                        printf("%d\n", number);
        }
        fclose(f2);
}
```

Output Of The Program

```
Input File name
TETS
Cannot open the file
Type filename again
TEST
10
20
30
40
50
60
70
80
90
```

100

Ran out of data

3.4.5 Random Access To Files

In case of random access to file, we can access only a particular part of a file and not in reading the other parts. This is achieved with the help of the **fseek, ftell,** and **rewind** functions available in the I/O library.

ftell function:

The **ftell** function is used to *specify the current position of a file*. It takes the file pointer and returns a number of types long, that corresponding to the current position. The *general form* is

```
n  =  ftell(fp);
```

where n would give the relative offset (in bytes) of the current position.

rewind function:

The rewind takes the file pointer and *resets the position to the start of the file*. For **example**

```
rewind(fp);
n = ftell(fp);
```

would assign 0 to n because the file position has been set to the start of the file by rewind. The first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file.

fseek function:

The **fseek** function is used to move the file position to a desired location within the file. The *general form* is

```
fseek(file_pointer, offset, position);
```

where

file_ pointer is a pointer to the file

offset is a number or variable of type long, and

position is an integer number

The offset specifies the number of positions to be moved from the location specified by position. The position can take one of the following three values.

Value	Meaning
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, it moves forwards or negative, it moves backwards. The following examples illustrate the operation of the **fseek** function:

Statement	Meaning
fseek(fp, 0L, 0);	Go to beginning .
fseek(fp, 0L, 1);	Stay at the current position
fseek(fp, 0L, 2);	Go to end of the file, past the last character of the file.
fseek(fp, m, 0);	Move to (m+1) th byte in the file.
fseek(fp, m, 1);	Go forward by m bytes.
fseek(fp, -m, 1);	Go backward by m bytes from the current position.
fseek(fp, -m, 2);	Go backward by m bytes from the end.

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns –1.

Example Program: Program to illustrate of fseek & ftell functions

```
#include<stdio.h>
main( )
{
        FILE *f1;
        int n;
        char c;
        f1 = fopen("RANDOM","w");
        while(( c = getchar( ) ) != EOF)
                putc(c, f1);
        printf("No. of characters entered = %d\n",ftell(f1));
        fclose(f1);
        f1 = fopen("RANDOM","r");
        n = 0;
```

```
        while(feof(f1) == 0)
        {
                fseek(f1, n, 0); /* Position to (n+1) th character */
                printf("Position of %c is %d\n", getc(f1), ftell(f1));
                n = n + 5;
        }
        putchar('\n');
        fseek(f1, -1, 2);         /* Position to the last character */
        do
        {
                putchar(getc(f1));
        }
        while(fseek(f1, -2, 1));
        fclose(f1);
    }
```

Output Of The Program

```
    ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
    No. of characters entered = 26
    Position of A is 0
    Position of F is 5
    Position of K is 10
    Position of P is 15
    Position of U is 20
    Position of Z is 25
    Position of    is 30
```

3.4.6 Command Line Arguments

Command line argument is a *parameter supplied to a program* when the program invoked. This parameter may represent a filename the program should process. For **example** if we want to execute a program to copy the contents of a file named xfile to another file name yfile, then we may use command line like

```
c>program    xfile    yfile
```

program is the filename where the executable code of the program is stored.

In C program should have one main function and that it marks the beginning of the program. The main function take *two arguments* called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an *argument counter* that counts the number of arguments on the command line. The **argv** is an *argument vector* and represents an array of character pointers that point to the command line arguments. The size of the array will be equal to the value of **argc**. For the above **example**, **argc** is three and **argv** is an array of *three pointers to strings* as

```
argv[0]  -------> program
argv[1]  ------->xfile
argv[2]  ------->yfile
```

In order to access the command line arguments, we must declare the main function and its parameters as

```
main(argc, argv)
int argc;
char *argv[];
{
        -------
        -------
}
```

the *first parameter* in the command line is always the **program name** and therefore argv[0] always represents the program name.

Example Program: Program to illustrate command line arguments

```
#include<stdio.h>
main(int argc, char argv[ ])    /* Main with arguments          */
{
FILE *f1;
int i;
char word[15];
f1 = fopen(argv[1], "w");       /* Open the file with name argv[1] */
printf(" \nNo. Of argument in command line = %d\n", argc);
for(i = 2; i < argc; i++)
        fprintf(f1, "%s",argv[i]);
```

```
fclose(f1);
/* Writing content of the file to screen */
printf("content of  %s file \n", argv[1]);
f1 = fopen(argv[1], "r");
for(i = 2; i <argc; i++);
{
        fscanf(f1, "%s", word);
        printf("%s", word);
}
fclose(f1);
printf("\n");
/* Writing argument from memory */
for(i = 0; i < argc; i++)
        printf("%*s\n", i*5, argv[i]);
}
```

Output Of The Program

```
C:\TC>file6 TEST aaa bbb ccc
No. Of argument in command line = 5
content of  TEST file
aaabbbccc
C:\TC\FILE6.EXE
 TEST
      aaa
           bbb
                ccc
```

3.4.7 Self Assessment Questions

Fill in the blank

1. The ___________________is used to move the file position to a desired location within the file.

2. Command line argument is a ________________supplied to a program when the program invoked.

True / False

1. If a file is opened for writing already exists its contents would be overwritten.

2. The **getw** and **putw** are integer-oriented functions

Multiple Choice

1. fscanf( ) function is used to reading of the items from

a) file	b) keyboard
c) monitor	d) All of the above

Short Answer

1. What is mean by file?

3.5.Summary

In the first lesson of this unit we have examined strings, which are nothing but arrays of characters. We have seen how the strings are stored in memory, how initialize strings, and how to access elements of a string using subscript. We learnt two new functions to input and output strings: **gets( )** and **puts( );** more functions **strlen( ), strcpy( ), strcat( )** and **strcmp( )** which can manipulate strings. We also explored topic of two-dimensional array of characters.

In the second lesson of this unit we have examined pointers, which is the variable contains address. We have seen how the pointers can be processed, and how to use the pointer arithmetic.

In the third lesson of this unit we have examined files, which is the collection related information. We have seen how the files accessed in sequential and random access method with the help various file handling functions.

3.6.Unit Questions

1.List out various string handling functions. Explain each function with example.

2. How to reading and writing strings in C language? Explain.

3. Define pointers. Explain how are pointers accessed, declared and initialized?

4. Write a program for arrays with pointers.

5. Discuss the structures & pointers with example program.

6. What are the advantages while using pointers?

7. Explain command line arguments.

8. Discuss random access file with related functions.

9. How the error will be handled during input / output operation? Explain.

10. Discuss reading out of or writing to it is accomplished using the standard I/O routines.

3.7.Answers for Self Assessment Questions

Answer 3.3.6

Fill in the blank

1. array of characters 2. strcpy ( )

True / False

1. False 2. True

Multiple Choice

1. c) 2. b)

Short Answer

1. The strrev( ) function is used to reverse a string.

Answer 3.4.12

Fill in the blank

1. index, data type
2. call by reference

True / False

1. True

Multiple Choice

1. b) 2. d)

Short Answer

1. Pointer variables can be used in expressions is called as pointer expression.
2. The operator **&** immediately preceding a variable returns the address of the variable associated with it.

Answer 3.5.6

Fill in the blank

1. **fseek** function 2. parameter

True / False

1. True 2. True

Multiple Choice

1. a)

Short Answer

1. A **file** is a collection of related information, that is permanently stored on the disk and allows us to access and alter the information whenever necessary.

NOTES

UNIT – IV

4.1. Introduction

This unit provides you with *fundamental concepts of Data structure.* It deals with various linear data structure like array, stack, queue, and linked list. And also deals the applications of data structure and its implementation. It assures you to understand general data structure concepts.

4.2. Objectives

After studying this unit, you should be able to:

- Understand the data structure and its types..
- Understand the array operations and its types
- Understand the various operation on stack and its application.
- Understand the various operation on stack and its application.
- Understand the various operation on queue and its application.
- Understand the various operation on linked list and its usage.

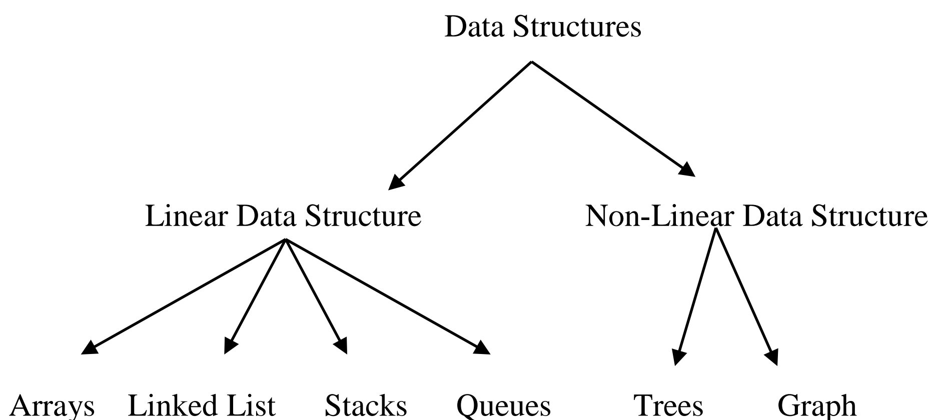
4.3. Data Structure

4.3.1. Definition

A collection of data elements, whose arrangement is characterized by accessing functions that are used to store and retrieve individual data elements are called data structure.

4.3.2. Categories Of Data Structures

The data structure can be classified into **two types** as



Linear Data Structure

In linear data structure, all elements are formed in a sequence or maintain in a linear ordering. Linear data structures are

- **Arrays**

  Array is the sequential organization, and using only index can access the elements of an array, but the size of an array must be previously specified at the array definition. The elements of array can be stored in consecutive memory location

- **Linked list**

  A linked list is a collection of n number of data items of same type on nodes; each node contains two fields as one is element or item and another is address of the next node.

- **Stacks**

  Stack is an ordered collection of homogeneous data elements, where the insertion and deletion operations take place at one end called top of the stack. That is, in stack the last inserted element can be deleted first. It operates last in first out (LIFO) fashion.

- **Queues**

  Queue is an ordered collection of homogeneous data elements, in which the element insertion and deletion operations takes place at two end called front and rear end. It operates first in first out (FIFO) fashion.

Non-Linear Data Structure

In non-linear data structure, all the elements are distributed on a plane that is these have no such sequence of element as in case of linear data structure.

- **Trees:** A tree is a non-linear data structure in which the data items are arranged.
- **Graphs:** A **graph G** consists of a set of vertices V and a set of edges (links) E. then G can be written as,

$$G = (V, E)$$

where

$$V = \{ v_1, v_2, \ldots\ldots v_n\}$$

$$E = \{e1, e_2, \ldots\ldots e_n\}$$

4.3.3. Self Assessment Questions

Fill in the blank

1. A data structure is said to be _____________ if its elements form a sequence.

2. Tree is a ___________________ data structure.

True / False

1. An array is a non-linear data structure.

Multiple Choice

1. The stack structure is

a) Last In First Out b) First In First Out

c) Last In Last Out d) None of the above

Short Answer

1. Define data structure.

4.4. Arrays

4.4.1. Introduction

An **array** is a group of related data items that *share a common name*. An array of value can be accessed by *index or subscript* enclosed in square brackets after array name. For **example**, a[5] represents the 5th element in an array. An array may contain all integers or all characters but not both.

An array may containing **n** number of elements is referenced using an index that varies from 0 to n-1. The subscript can begin with number 0. The value of each subscript can be expressed as an integer constant or integer variable or an integer expression. For **example,** the elements of an array arr[n] containing n elements are denoted by arr[0], arr[1], . . . arr[n-1], where **0** is the lower bound and **n-1** is the upper bound of the array.

In general, the lowest index of an array is called its **lower bound** and the highest index is called its **upper bound**. The number of elements in the array is called its range.

Arrays can be *classified* into

- One-Dimensional arrays
- Two-Dimensional arrays
- Multi-Dimensional arrays

One Dimensional Array

A list of data items can be stored under a one variable name using only one subscript and such a variable is called a *single-subscripted* variable or *one-*

dimensional array. For **example,** if we want to represent a set of four numbers, by array variable **a** then we may declare the variable **a** as

```
int     a [ 4 ];
```

and computer reserves four memory locations as

a[0]	
a[1]	
a[2]	
a[3]	

The values to the array elements can be assigned as

a[0] = 10 ;

a[1] = 20 ;

a[2] = 30 ;

a[3] = 40 ;

This would cause the array **a** to store the values as

a[0]	**10**
a[1]	**20**
a[2]	**30**
a[3]	**40**

This element may be used in programs just like any other C variable. For **example**, the following are **valid statements**:

```
b = a[0] + 10;
a[3] = a[0] + a[2];
```

4.4.2. Array Operations

There are several operations that can be performed on an array. These operations are

- Traversal - Processing each element in the array.
- Search – Finding the location of an element with a given value.
- Insertion – Adding a new element to an array
- Deletion – Removing an element from an array
- Sorting – Organizing the elements in some order

- Merging – Combining two arrays into a single array
- Reversing – Reversing the elements of an array

Example Program: Program for array operations

```
#include<stdio.h>
#define MAX 5
void insert(int *, int pos, int num);
void delete(int *, int pos);
void reverse(int *);
void display(int *);
void search(int *, int num);
void main( )
{
    int  arr[5], i, value;
    printf("\nEnter the array elements");
    for( i = 1; i <= 5; i++)
    {
       scanf("%d", &value);
       insert(arr,  i, value);
    }
    printf("\nElements of Array");
    display(arr);
    delete(arr, 5);
    delete(arr, 2);
    printf("\nAfter Deletion");
    display(arr);
    insert(arr, 2, 20);
    insert(arr, 5, 50);
    printf("\nAfter Insertion");
    display(arr);
    reverse(arr);
    printf("\nAfter Reversing");
    display(arr);
    search(arr, 20);
    search(arr, 60);
```

```
        getch( );
}
/* Insert an element num at given position pos into an array arr*/
void insert(int *arr, int pos, int num)
{
        /* shift elements to right */
        int i;
        for(i = MAX – 1 ; i >= pos; i--)
                arr[i] = arr[i-1];
        arr[i] = num;
}
/* Delete an element num from the given position pos */
void delete(int *arr, int pos)
{
        /* skip to the desired position */
        int i;
        for(i =  pos;  i < MAX  ; i++)
                arr[i - 1] = arr[i];
        arr[i - 1] = 0;
}
/* Reverse the entire array */
void reverse(int *arr)
{
        int temp, i;
        for(i = 0;  i < MAX / 2   ; i++)
        {
                temp  = arr[i];
                arr[i] = arr[MAX – 1- i];
                arr[MAX – 1- i] = temp;
        }
}
/*Searches array for a given element num */
void search(int *arr, int num)
{
```

```
        /* Traverse the array */
        int i;
        for(i = 0;  i < MAX  ; i++)
        {
            if( arr[i] = = num)
            {
                printf("\nThe element %d is  present at %dth position",

                num, i +1);
                 return;
            }
        }
        if(i = = MAX)
           printf("\n The element %d is not present in the array", num);
    }
    /* Display the contents of a array */
    void display(int *arr)
    {
        int  i;
        printf("\n");
        for(i = 0;  i < MAX  ; i++)
            printf("%d\", arr[i]);
    }
```

Output Of The Program

```
Enter the array elements
11
12
13
14
15
Elements of Array
11      12      13      14      15
After Deletion
11      13      14      0       0
```

After Insertion

11 20 13 14 50

After Reversing

50 14 13 20 11

The element 20 is present at 4th position

The element 60 is not present in the array

In the above program we created an array with *5 integer* elements. Then the base address of this array is passed to functions like **insert( ), delete( ), reverse( )** and **search( )** to perform different array operations.

The **insert( )** function takes *two arguments*, the position **pos** at which the new number has to be inserted and the number **num** that has to be inserted. In this function, first through a loop, we have shifted the numbers, from the specified position, one place to the right of their existing position. Then we have placed the number **num** at the vacant place.

The **delete( )** function deletes the element present at the given position pos. While deleting, we have shifted the numbers placed after the position from where the number is to be deleted, one place to the left of their existing positions. The place that is vacant after deletion of an element is filled with 0.

The **reverse( )** functions, we have reversed the entire array by swapping the elements like arr[0] with arr[4], arr[1] with arr[3] and so on. Swapping should continue for MAX / 2 times only.

The **search( )** function searches the array for the specified number. In this function, the 0th element has been compared with the given number num. If the element compared to be same then the function displays the position at which the number if found. Otherwise the comparison is carried out until either the list is exhausted or a match is found. If match is not found then the function displays the relevant message.

In the **display( )** function, the entire array is traversed. As the list traversed the function displays the elements of the array.

4.4.3. Merging Of Two Arrays

Merging of arrays involves *two steps are sorting the array* that are to be merged, and adding the sorted elements of both the arrays to a new array in a sorted order.

Example Program: Program to demonstrate merging of two arrays into a third array

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

```
#define MAX1 5
#define MAX2 6
int * arr;
int *create(int);
void sort(int*, int);
void display(int* , int);
int* merge(int *, int *);
void main( )
{
        int *a, *b, *c;
        printf("\nEnter elements for first array:\n");
        a = create(MAX1);
        printf("\nEnter elements for second array:\n");
        b = create(MAX2);
        sort(a, MAX1);
        sort(b, MAX2);
        printf("\nFirst array:");
        display(a, MAX1);
        printf("\nSecond array:");
        display(b, MAX2);
        printf("\n After Merging:");
        c = merge(a, b);
        display(c, MAX1+MAX2);
        getch( );
}
/* Create array of given size, dynamically */
int* create(int size)
{
        int *arr, i;
        arr = (int *)malloc(sizeof(int) * size);
        for(i = 0; i < size; i++)
        {
                printf("Enter the element %d       ", i +1);
                scanf("%d", &arr[i]);
```

```
        }
        return arr;
}
/* Sorts array in ascending order */
void  sort(int *arr, int size)
{
        int temp, j, i;
        for(i = 0; i < size; i++)
        {
                for(j = i+1;  j < size; j++)
                {
                        if(arr[i] >arr[j])
                        {
                                temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                        }
                }
        }
}
/* Display the contents of array */
void display(int *arr, int size)
{
        int i;
        for(i = 0; i < size; i++)
                printf("\t%d", arr[i]);
}
/* Merge two arrays of different size */
int* merge(int *a, int *b)
{
        int *arr;
        int i, j, k;
        int size = MAX1 + MAX2 ;
        arr = (int *)malloc(sizeof(int) * size);
```

```
        k = 0; j = 0;
        for(i = 0; i <= size;i++)
        {
                if( a[k] < b[j])
                {
                        arr[i] = a[k];
                        k++;
                        if(k >= MAX1)
                        {
                                for(i++; j < MAX2; j++, i++)
                                        arr[i] = b[j];
                        }
                }
                else
                {
                        arr[i] = b[j];
                        j++;
                        if(j >= MAX2)
                        {
                                for(i++ ; k <MAX1; k++, i++)
                                        arr[i] = b[k];
                        }
                }
        }
        return arr;
}
```

Output Of The Program

```
Enter elements for first array:
Enter the element 1   56
Enter the element 2   36
Enter the element 3   76
Enter the element 4   46
Enter the element 5   96
```

Enter elements for second array:

Enter the element 1   37

Enter the element 2   27

Enter the element 3   97

Enter the element 4   57

Enter the element 5   47

Enter the element 6   87

First array: 36   46   56   76   96

Second array: 27   37   47   57   87   97

After Merging: 27   36   37   46   47   56   57   76   87   96   97

The above program the **create( )** function is used to create an array of integers, function **sort( )** is used to sort the elements of an array in ascending order, function **merge( )** is used to add elements of two arrays to the new array and function **display( )** is used to display the contents of an array.

All arrays created dynamically because of the size of each array is different from the other.

Inside the function **merge( ),** a for loop is executed for size number of times. Here size is the sum of the number of element presents in the arrays that are pointed by a and b respectively. Before placing the element in the new array arr that is pointed by we have sequentially compared the elements of two arrays which are pointed by a and b respectively. The element that is found to be smaller is added first to the array that is pointed by arr.

4.4.4. Two Dimensional Arrays

A *two dimensional array* is a collection of elements placed in *m rows* and *n columns*. The syntax used to declare a 2-D array includes *two subscripts*, of which one specifies the number of rows and the other specifies the number of columns of an array. Each dimension of the array is indexed from *zero to its maximum size minus one*:

- The first index selects the row and
- The second index selects the column within that row.

Example:

```
int   table[3] [3];
```

This creates a table that can store 9 integer values, three across (row) and three down (column).

		Columns		
		C1	C2	C3
	R1			
Rows	R2			
	R3			

The individual elements are identified by index or subscript of an array from the above **example**.

table[0][0] table[0][1] table[0][2]

table[1][0] table[1][1] table[1][2]

table[2][0] table[2][1] table[2][2]

Initializing Two Dimensional Arrays

A two-dimensional array may be initialized by following their declaration with a list of initial values enclosed in braces.

For **example**,

```
int table[2] [3] = {0,0,0,1,1,1};        //initialize the elements row by row
```

or

```
int table[2] [3] = {{0,0,0},{1,1,1}}; //separate the element of each row by braces
```

or

```
int table[2] [3] = {
                     {0,0,0},
                     {1,1,1}
                     };
```

If the values are missing in an initializer, they are automatically set to zero. For **example**:

```
int table[2] [3] = {
                     {1,1},
                     {2}
                     };
```

will initialize the first two elements of the row to one, the first element of the second row to two, and *all other elements to zero*.

Row major and Column major arrangement

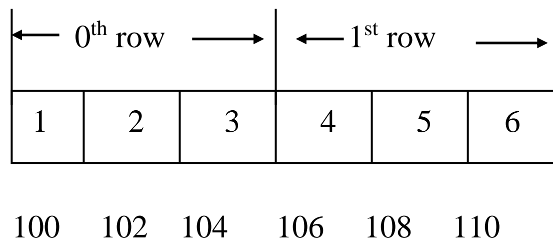
In memory all elements are stored in consecutive units of memory. This leads to *two possible arrangements of elements in memory* are

- Row major arrangement
- Column major arrangement

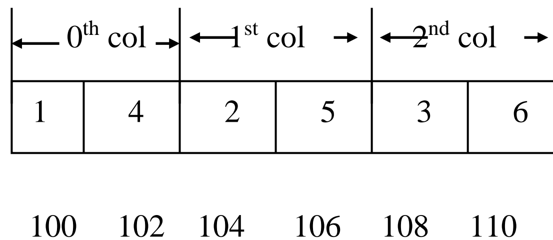
Fig 4.4.1 illustrates these two arrangements for a 2-D array. For **example**

```
int  a[2][3] = {
                {1, 2, 3},
                {4, 5, 6}
               };
```

Row major arrangement



Column major arrangement



Note: Each integer occupies two bytes

Fig 4.4.1 Possible arrangements of 2-D array

Since the array elements are stored in adjacent memory locations we can access any element of the array once we know the base address (starting address) of the array and number of rows and columns present in the array.

Example Program: Program for Addition of Two Matrix using Two-Dimensional Array

```
/* Matrix Addition Using Two Dimensional Array   */
#include<stdio.h>
void main( )
{
```

```
int a[ 5 ][ 5 ] , b[ 5 ][ 5 ] ,c[ 5 ][ 5 ] ;
int n = 0, i = 0, j = 0;
printf("Enter the order of matrix");
scanf("%d", &n);
printf("\nEnter the A matrix");
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
        scanf("%d", &a[i][j] );
    }
}
 printf("\nEnter the B matrix");
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
        scanf("%d", &b[i][j] );
    }
}
printf("\nThe A matrix\n");
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
         printf("%d\t", a[i][j]);
    }
    printf("\n");
}
printf("\nThe B matrix\n");
 for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
   {
```

```
            printf("%d\t", b[i][j]);
        }
        printf("\n");
    }

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    printf("\nThe Resultant Matrix is\n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            printf("%d\t", c[i][j]);
        }
        printf("\n");
    }
getch( );
}
```

Output Of Program

```
Enter the order of matrix
2
Enter the A matrix
2
2
2
2
Enter the B matrix
2
2
```

2

2

The A matrix

2   2

2   2

The B matrix

2   2

2   2

The Resultant Matrix is

4   4

4   4

4.4.5. Self Assessment Questions

Fill in the blank

1. An array is a collection of _______________ elements stored in _____________ memory location.

2. Index of an array containing n elements varies from _______ to ___________.

True / False

1. Merging refers to processing elements of an array.

2. Each dimension of the array is indexed from 0 to its maximum size minus one.

Multiple Choice

1. To traverse an array means

a)To process each element in an array   b) To delete an element from an array

c)To insert an element into an array   d) To combine two arrays into single array

Short Answer

1. What is purpose of insertion operation in array?

4.5. Stacks

4.5.1. Definition

Stack is an *ordered collection of homogeneous data elements*, where the insertion and deletion operations take place at one end called *top of the stack*. That is, in stack the last inserted element can be deleted first. It operates *last in first out (LIFO)* fashion. The structure of stack is as shown in Fig.4.5.1

Example

Stack of plates: It will be placed on a table in a party where a guest always picks up a fresh plate from the top.

Pile of coins.

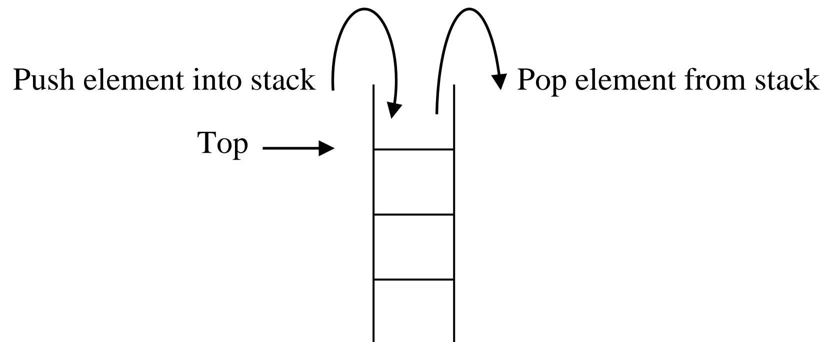
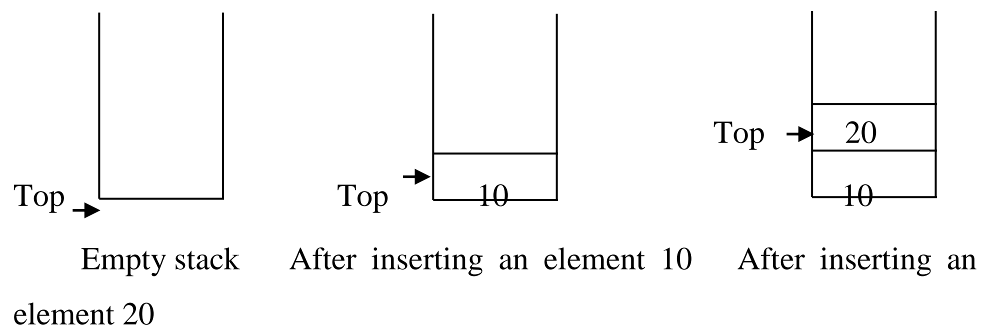


Fig.4.5.1 Structure of a Stack

4.5.2. Operations On Stack

The stack has *two basic operations* such as

- **Push**: Inserting a new element into the top of stack. For every push operation the top is incremented by 1.



Empty stack After inserting an element 10 After inserting an element 20

Algorithm:

StackPUSH(Stack , Max_size)

Step 1: [check for stack overflow]

If Top > Max_size –1 then

Print "Stack is full" and

else

Step 2: [Increment Top]

Set Top = Top +1

Step 3: [Insert new element into the Top position]

Set Stack[Top] = item

End StackPUSH

- **Pop**: Deleting an element from the top of stack. After every pop operation the top pointer is decremented by 1.

Algorithm:

StackPOP(Stack , Max_size)

Step 1: [check for stack underflow]

If Top < 0 then

Print "Stack is empty" and

else

Step 2: [Delete the element from the Top position]

Set item = Stack[Top]

Step 3: [decrement Top]

Set Top = Top - 1

End StackPOP

Exceptional conditions

- **Overflow:** Attempt to insert an element, when the stack is full is said to be overflow.
- **Underflow:** **A**ttempt to delete an element, when the stack is empty is said to be underflow.

Stack operations can be implemented either by *using an array* (static implementation) or *by using pointer* (linked list or dynamic implementation)

4.5.3. Representation Of A Stack As An Array

In the array representation of stack, first allocate memory block of required size to accommodate the capacity of the stack, then starting from first locations. Let us see a program that implements a stack using an array.

Program: Program to implement a stack using an array.

```
#include<stdio.h>
#include<conio.h>
#define MAXSIZE 10
void push( );
int pop( );
int stack[MAXSIZE];
int top = -1;
void main( )
{
    int choice;
    char ch;
    do
    {
        clrscr( );
        printf("\n1.Push");
        printf("\n2.Pop");
        printf("\n3.Exit");
        printf("\nEnter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:
                push( );
```

```
                break;
            case 2:
                printf("\nThe deleted element is %d", pop( ) );
                break;
            default:
                printf("\nYou entered wrong choice");
            }
            printf("\nDo you wish to continue (Y / N)");
            fflush(stdin);
            scanf("%c",&ch);
            }
            while(ch = ='Y' || ch = ='y');
}
void push( )
{
        int item;
        if ( top = = MAXSIZE –1)
        {
            printf("\n The Stack is Full");
            exit(0);
        }
        else
        {
                printf("\nEnter the new element");
                scanf("%d", &item);
                top = top + 1;
                stack[top] = item;
            }
}
int pop( )
{
            int item;
            if( top = = -1)
            {
```

```
                printf(" The Stack is Empty");
                getch( );
                exit(0);
            }
            else
            {
                item = stack[top];
                top = top – 1;
            }
            return(item);
        }
```

Output Of The Program

```
1.Push
2.Pop
3.Exit
Enter your choice 1
Enter the new element 10
Do you wish to continue (Y / N) y
1.Push
2.Pop
3.Exit
Enter your choice 1
Enter the new element  20
Do you wish to continue (Y / N) y
1.Push
2.Pop
3.Exit
Enter your choice 2
The deleted element is 20
Do you wish to continue (Y / N) n
```

In this program we have defined an *array* called *stack* with the **MAXSIZE**. The **MAXSIZE** indicates the length of the array. The **push( )** and **pop( )** functions are used to add and delete items from the top of the stack. The variable **top** is act as index of the array stack. In this implementation each stack is associated with a top pointer, which is **–1** for an *empty stack* and **MAXSIZE – 1** for a *full stack*.

4.5.4. Representation Of A Stack As An Linked List

The stack using array is very easy and convenient, but it allows only to represents fixed size stacks. In several applications the size of the stack may vary during the program execution. In such situation the linked list representation of the stack is very useful, and the single linked list structure is sufficient to represent stack.

Initially the **list is empty**, so the top pointer is **NULL**. The push function takes a pointer to an exiting list as the parameter and a data value to be pushed as the second parameter, creates a new node by using data value and adds it to the top of the existing list.

A pop function takes a pointer to existing list as first parameter, and a pointer to a data object in which the popped value is to be returned as a second parameter. Thus it retrieves the value of the node pointed to by the top pointer, takes the top point to the next node, and destroys the node that was pointed to by the top. Let us see a program that implements a stack using an array.

Declaration for linked list implementation

```
struct node;
typedef struct node *stack;
int isempty(stack s);
stack createstack(void);
void make empty(stack s);
void push(int x, stack s);
int top(stack s);
void pop(stack s);
struct node
{
int element;
struct node * next;
};
```

Routine to check whether the stack is empty

```
int isempty(stack s)
{
    if (s->next == NULL)
    return(1);
}
```

Routine to create an empty stack

```
stack create( )
{
    stack s;
    s = malloc(sizeof(struct node));
    if (s == NULL)
    error(“Out of space”);
    makeempty(s);
    return s;
}
void makeempty(stack s)
{
    if ( s == NULL)
        error(“create stack first”);
    else
        while( ! isempty(s))
                pop(s);
}
```

Routine to push an element onto a stack

```
void push(int x, stack s)
{
        struct node *tempnode;
        tempnode = malloc(sizeof(struct node));
        if( tempnode == NULL)
            error(“ Out of space”);
        else
```

```
        {
            tempnode->element = x;
            tempnode->next = s->next;
            s->next = tempnode;
        }
}
```

Routine to return top element in a stack

```
int top(stack s)
{
    if( !isempty(s))
        return s->next->element;
    error("Empty stack");
    return 0;
}
```

Routine to pop from a stack

```
void pop(stack s)
{
    struct node *tempnode;
    if( !isempty(s))
        error("Empty stack");
    else
    {
        tempnode = s->next;
        s->next = s->next->next;
        free(tempnode);
    }
}
```

4.5.5. Evaluation Of Expression

4.5.5.1. Introduction

Evaluation of arithmetic expression is the one of the applications of stack. There are different types of notations to represent arithmetic expression. They are

- Infix notation
- Postfix notation
- Prefix notation

While writing arithmetic expression, the operator symbol is usually placed between two operands. For **example**

A + B * C

A * B – C

A + B / C – D

This way of representing arithmetic expressions is called **infix notation**. While evaluating an infix expression follows the operator precedence is used:

- Highest Priority: Exponentiation ($)
- Next Highest priority: Multiplication (*) and division (/)
- Lowest priority: addition (+) and subtraction (-)

The expressions with in a pair of parentheses are always evaluated earlier than other operations.

In **prefix notation** the operator comes before the operands. For **example**

+ A B

In **postfix notation**, the operator follows two operands. For **example**

A B+

4.5.5.2. Infix To Prefix Conversion

Let us now see a program that would accept an expression in infix form and convert it to a prefix form.

Program: Program for infix to prefix conversion

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAX 50
struct infix
{
        char target[MAX];
        char stack[MAX];
        char *s, *t;
        int top, i;
};
```

```
void initinfix(struct infix *);
void setexpr(struct infix *, char *);
void push(struct infix *, char);
char pop(struct infix *);
void convert(struct infix *);
int priority(char c);
void show(struct infix);
void main( )
{
	struct infix  q;
	char expr[MAX];
	initinfix(&q);
	printf("\nEnter an expression in infix form");
	gets(expr);
	setexpr(&q, expr);
	convert(&q);
	printf("\nThe prefix expression is :");
	show(q);
	getch( );
}
/* initializes elements of structure variable */
void initinfix(struct infix *pq)
{
	pq->top = -1;
	strcpy(pq->target, "");
	strcpy(pq->stack, "");
	pq->i = 0;
}
/* reverse the given expression */
void setexpr(struct infix *pq, char *str)
{
	pq->s = str;
	strrev(pq->s);
	pq->i = strlen(pq->s);
```

```
        *(pq->target +pq->i ) ='\0';
        pq->t = pq->target + (pq->i -1);
}
/* adds operator to the stack */
void push(struct infix *pq, char c)
{
        if(pq->top == MAX-1)
                printf("\nStack is full\n");
        else
        {
                pq->top++;
                pq->stack[pq->top] = c;
        }
}
/* pos an operator from the stack */
char pop(struct infix *pq)
{
        if(pq->top == -1)
        {
                printf("\nStack is full\n");
                return -1;
        }
        else
        {
                char item = pq->stack[pq->top];
                pq->top--;
                return item;
        }
}
 /* coverts the infix expr. Into prefix expr form */
void convert(struct infix *pq)
{
        char opr;
        while(*(pq->s))
```

```
{
        if (*(pq->s ) == ' ' || *(pq->s) == '\t')
        {
                pq->s++;
                continue;
        }
        if(isdigit(*(pq->s )) || isalpha(*(pq->s) ))
        {
                while(isdigit(*(pq->s))                ||
                isalpha(*(pq->s) ))
                {
                *(pq->t) = *(pq->s);
                pq->s++;
                pq->t--;
                }
        }
        if(*(pq->s) == ')')
        {
                push(pq, *(pq->s));
                pq->s++;
        }
        if(*(pq->s) == '*' || *(pq->s) == '+' || *(pq->s) == '/' ||
         *(pq->s) =='%' || *(pq->s) == '-' || *(pq->s ) == '$')
        {
                if(pq->top != -1)
                {
                        opr = pop(pq);
                        while(priority(opr)
                        >priority(*(pq->s)))
                        {
                                *(pq->t) = opr;
                                pq->t--;
                                opr = pop(pq);
                        }
```

```
                push(pq, opr);
                push(pq, *(pq->s));
            }
            else
                push(pq, *(pq->s));
            pq->s++;
        }
        if(*(pq->s) == '(')
        {
            opr = pop(pq);
            while(opr != ')')
            {
                *(pq->t) = opr;
                pq->t--;
                opr = pop(pq);
            }
            pq->s++;
        }
    }
    while(pq->top != -1)
    {
        opr = pop(pq);
        *(pq->t) = opr;
        pq->t--;
    }
    pq->t++;
}
/* return the priority of the operator */
int priority(char c)
{
    if ( c == '$')
        return 3;
    if ( c == '*' ||  c == '/' || c == '%')
        return 2;
```

```
    else
    {
        if ( c == '+' ||  c == '-')
            return 1;
        else
            return 0;
    }
}
/* display  the prefix form of given expr */
void show(struct infix pq)
{
    while(*(pq.t))
    {
        printf("%c", *(pq.t));
        pq.t++;
    }
}
```

Output Of The Program

Enter an expression in infix form4$2*3-3+8/4/(1+1)

The prefix expression is :_+-*$4233//84+11

4.5.5.3. Infix To Postfix Conversion

Let us now see a program that would accept an expression in infix form and convert it to a postfix form.

Program: Program for infix to postfix conversion

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAX 50
struct infix
{
    char target[MAX];
    char stack[MAX];
    char *s, *t;
    int top;
```

```
};
void initinfix(struct infix *);
void setexpr(struct infix *, char *);
void push(struct infix *, char );
char pop(struct infix *);
void convert(struct infix *);
int priority(char);
void show(struct infix);
void main( )
{
        struct infix  q;
        char expr[MAX];
        initinfix(&q);
        printf("\nEnter an expression in infix form");
        gets(expr);
        setexpr(&q, expr);
        convert(&q);
        printf("\nThe prefix expression is :");
        show(q);
        getch( );
}
/* initializes elements of structure variable */
void initinfix(struct infix *p)
{
        p->top = -1;
        strcpy(p->target, "");
        strcpy(p->stack, "");
        p->t =p->target;
        p->s= "";
}
/* sets s to point to given expr*/
void setexpr(struct infix *pq, char *str)
{
        pq->s = str;
```

```
}
/* adds operator to the stack */
void push(struct infix*pq, char c)
{
    if(pq->top == MAX-1)
    printf("\nStack is full\n");
    else
    {
        pq->top++;
        pq->stack[pq->top] = c;
    }
}
/* pos an operator from the stack */
char pop(struct infix *p)
{
    if(p->top == -1)
    {
        printf("\nStack is empty\n");
        return -1;
    }
    else
    {
        char item = p->stack[p->top];
        p->top--;
        return item;
    }
}
```

/ coverts the infix expr. Into prefix expr form */*

```
void convert(struct infix *p)
{
    char opr;
    while(*(p->s))
    {
        if (*(p->s ) == ' ' || *(p->s) == '\t')
        {
            p->s++;
            continue;
        }
        if(isdigit(*(p->s )) || isalpha(*(p->s) ))
        {
            while(isdigit(*(p->s)) || isalpha(*(p->s) ))
            {
                *(p->t) = *(p->s);
                p->s++;
                p->t++;
            }
        }
        if(*(p->s) == ')')
        {
            push(p, *(p->s));
            p->s++;
        }
        if(*(p->s) == '*' || *(p->s) == '+' || *(p->s) =='/' ||
        *(p->s)== '%' || *(p->s) == '-' || *(p->s) == '$')
        {
            if(p->top != -1)
            {
                opr = pop(p);
                while(priority(opr) >priority(*(p->s)))
                {
                    *(p->t) = opr;
```

```
                    p->t++;
                    opr = pop(p);
                }
                push(p, opr);
                push(p, *(p->s));
            }
            else
                push(p, *(p->s));
            p->s++;
        }
        if(*(p->s) == '(')
        {
            opr = pop(p);
            while(opr != ')')
            {
                *(p->t) = opr;
                p->t++;
                opr = pop(p);
            }
            p->s++;
        }
    }
    while(p->top != -1)
    {
        char opr = pop(p);
        *(p->t) = opr;
        p->t++;
    }
    *(p->t) = '\0';
}
/* return the priority of the operator */
int priority(char c)
{
    if ( c == '$')
```

```
                return 3;
        if ( c == '*' ||  c == '/' || c == '%')
                return 2;
        else
        {
                if ( c == '+' ||  c == '-')
                        return 1;
                else
                        return 0;
        }
}
/* display  the prefix form of given expr */
void show(struct infix p)
{
        printf("%s", p.target);
}
```

Output Of The Program

Enter an expression in infix forma+b

The prefix expression is :ab+

4.5.6. Self Assessment Questions

Fill in the blank

1. The data structure stack is also called____________ structure.

2. In _____________ notation the operator follows the two operands.

True / False

1. In stack new element or deletion of an exiting element always takes place at the same end.

Multiple Choice

1. Pushing an element to stack means

a) Removing an element from the stack b) Adding a new element to the stack c) Searching a given element from the stack d) Sorting the elements in the stack

Short Answer

1. What is mean by postfix expression?

4.6. Queues

4.6.1. Definition

Queue is a linear data structure that permits insertion of new element at one end is called **rear** and deletion of an element at the other end is called **front**. Queue has **First-In-First-Out (FIFO)** structure.

The structure of a queue is shown in Fig .4.6.1. The name 'queue' comes from the everyday use of the term.

Example

People waiting at a ticket counter in theater.

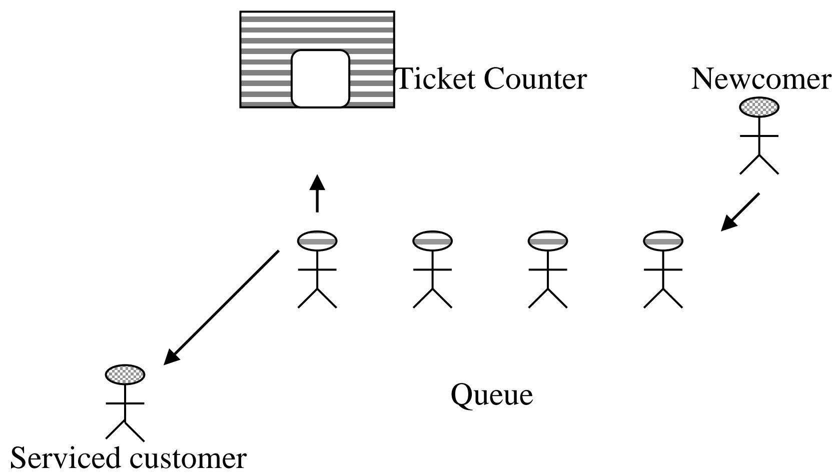


Fig .4.6.1 Sample Structure of Queue

4.6.2. Operations On Queue

The queue has two basic operations such as

- **Insertqueue**

  Inserting a new element into the queue.

```
Algorithm
  InsertQueue( )
  Step 1: [check for queue overflow]
              If rear = Maxsize – 1 then
                    Print " Queue is full" and
             else
  Step 2: [Increment rear]
              Set rear = rear +1
  Step 3: [Insert a new element into the queue]
              Set queue[rear] = item
  End InsertQueue
```

- **Deletequeue**

  Deleting an element from the queue.

Algorithm

```
  DeleteQueue( )
  Step 1: [check for queue underflow]
              If front > rear  then
                    Print " Queue is empty" and
             else
  Step 2: [Delete an element from the queue in front position]
              Set item  = queue[front]
  Step 3: [Decrement front pointer]
              Set front = front  - 1
  End DeleteQueue
```

Exceptional conditions

- **Overflow:** Attempt to insert an element, when the queue is full is said to be overflow.

- **Underflow:** Attempt to delete an element from the queue, when the queue is empty is said to be underflow

Types of Queues

- Linear Queue
- Circular Queue
- Dequeue (double ended Queue)

We have seen so far is referred to as the **linear queue**.

- Linear queue has front end and rear ends.
- Insertion takes place at rear end and deletion takes place at front end.
- Linear queue can be traversed in only one direction (front to rear).
- If the front pointer is in the front position and rear pointer is in the last position, the queue is said to be full queue.
- If the rear is -1 and front is 0, then the queue is said to be empty queue.

4.6.3. Representation Of Queue As An Array

Queue being a linear data structure can be represented in various ways such as **arrays** and **linked lists**. Representing a queue as an array would have same problem that we discussed in case of stacks.

An array is a data structure that can store a *fixed number of elements*. The size of an array should be fixed before using it

Queue, on the other hands keeps on changing as we remove elements from the front end or add new elements at the rear end. Declaring an array with maximum size would solve this problem. Fig.4.6.2 shows representation of a queue as an array.

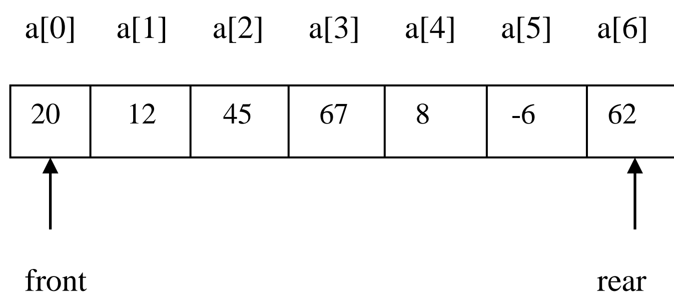


Fig.4.6.2. Representation of a queue as an array

Let us see a program that implements queue as an array.

Program: Program to implement queue as an array.

```
#include<stdio.h>
#define MAXSIZE 10
```

```
int front = -1, rear = -1, choice;
int queue[MAXSIZE];
void main( )
{
        do
        {
            printf("\n1. Insert\n");
            printf("2. Delete\n");
            printf("3. Display\n");
            printf("4. Exit\n");
            printf("Enter your choice\n");
            scanf("%d", &choice);
            switch(choice)
            {
                case 1: insert( );
                            break;
                case 2: delete( );
                            break;
                case 3: display( );
                             break;
                case 4: return;
            }
        }
        while(choice != 4);
}
/* inserting element into queue */
insert( )
{
        int num;
        if( rear == MAXSIZE – 1)
        {
            printf("Queue is full\n");
            return;
        }
```

```
        else
        {
            printf("Enter element\n");
            scanf("%d", &num);
            rear = rear + 1;
            queue[rear] = num;
            if( front == -1)
                front = front + 1;
        }
        return;
}
/* deleting element from the queue */
delete( )
{
        int num;
        if( front == -1)
        {
            printf("Queue is empty\n");
            return;
        }
        else
        {
            if( front == rear)
                    front = rear = -1;
            else
            {
                    num = queue[front];
                    printf("Deleted element is %d", queue[front]);
                    front = front +1;
            }
        }
        return(num);
}
/* display the content of the queue */
```

```
display( )
{
        int i;
        if(front == -1)
        {
                printf("Queue is empty\n");
                return ;
        }
        else
        {
                printf("\nThe status of the queue\n");
                for(i = front; i <= rear; i++)
                        printf("%d\t", queue[i]);
        }
        printf("\n");
}
```

Output Of The Program

```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice
1
Enter element
10
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice
1
Enter element
20
1. Insert
```

2. Delete
3. Display
4. Exit
Enter your choice
2
Deleted element is 10
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice
3
The status of the queue
20
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice
4

Here we have used an array **queue[]** to maintain the queue. We have also declared two variables front and rear to monitor the two ends of the queue. The initial values of front and rear are set to –1 which indicate that the queue is empty. The functions **insert( )** and **delete( )** are used to perform add and delete operations on the queue.

While *adding a new element to the queue*, first it would be check whether such addition is possible or not. Since array indexing begins with 0, the maximum number of elements that can be stored in the queue are **MAXSIZE –1**. If many elements are already present in the queue then it is reported to be full. If the element can be added to the queue then the value of the variable rear is incremented by 1 and the new value is stored in the array.

While *deleting elements from the queue*, first it would be check whether they are any elements available for deletion. If not then the queue is reported as empty. Otherwise, an element is deleted from the **queue[front].**

4.6.4. Representation Of Queue As A Linked List

Queue can also be represented using linked list. Space for the elements in a linked list is allocated dynamically; hence it can grow as long as there is enough memory available for dynamic allocation. Fig.4.6.3 shows representation of a queue as a linked list.

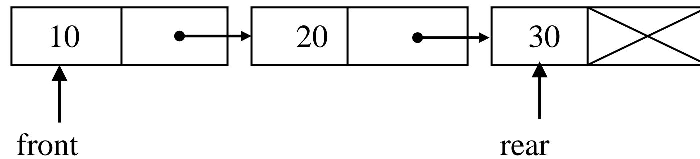


Fig.4.6.3. Representing of a queue as a linked list

Let us see a program that implements the queue as a linked list.

Program: Program to implements the queue as a linked list.

```
#include<stdio.h>
struct node
{
    int element;
    struct node *next;
};
struct queue
{
    struct node *front;
    struct node *rear;
};
void initqueue(struct queue *);
void addq(struct queue *, int);
int delq(struct queue *);
void main( )
{
    struct queue a;
    int i, num, choice;
    do
    {

        printf("\n1. Insert\n");
        printf("2. Delete\n");
```

```
            printf("3. Initialize\n");
            printf("4. Exit\n");
            printf("Enter your choice\n");
            scanf("%d", &choice);
            switch(choice)
            {
               case 1: printf("Enter element\n");
                       scanf("%d", &num);
                       addq(&a,  num);
                       break;
               case 2: num = delq(&a);
                       if ( num == 0)
                       printf("Deleted element is %d", num);
                       break;
               case 3:  initqueue(&a);
                       break;
               case 4: return;
            }
        }
        while(choice != 4);
}
/* initializes data member */
void initqueue(struct queue *q)
{
        q->front = q->rear = NULL;
        printf("Queue is initialized\n");
}
/* adds an element to the queue */
void addq(struct queue *q, int item)
{
        struct node *temp;
        temp = (struct node *) malloc(sizeof(struct node));
        if(temp == NULL)
                printf("\nQueue is full");
```

```
		else
		{
			temp->element = item;
			temp->next = NULL;
			if (q->front == NULL)
			{
				q->rear = q->front = temp;
				return;
			}
			else
			{
				q->rear->next = temp;
				q->rear = q->rear->next;
			}
		}
	}
	/* removes an element an element from the queue */
	int delq(struct queue *q)
	{
		struct node *temp;
		int item;
		if(q->front == NULL)
		{
			printf("\nQueue is empty");
			return NULL;
		}
		item = q->front->element;
		temp = q->front;
		q->front = q->front->next;
		free(temp);
		return item;
	}
```

Output Of The Program

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

3

Queue is initialized

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

1

Enter element

10

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

1

Enter element

20

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

2

Deleted element is 10

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

2

Deleted element is 20

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

2

Queue is empty

1. Insert

2. Delete

3. Initialize

4. Exit

Enter your choice

4

4.6.5. Circular Queues

The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full, even though there might be empty slots at the beginning of the queue. To overcome this limitation we can implement the queue as a circular queue.

Now if we go on adding elements to the queue we may reach the end of the array. We cannot add any more elements to the queue since we have reached the end of the array. Instead of reporting the queue is full, if some elements are deleted then there might be empty slots at the beginning of the queue. In such case these slots would be filled by new elements being added to the queue. Fig.4.6.4 shows the pictorial representation of a circular queue.

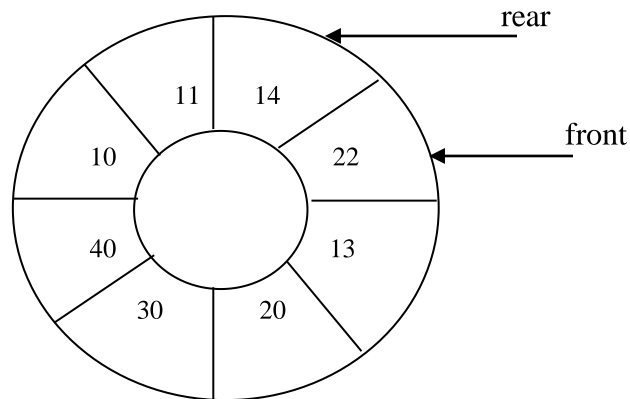


Fig.4.6.4. Pictorial representation of a circular queue

Program: Program to implement the circular queue

```
#include<stdio.h>
#define MAX 10
void  addq(int *, int, int *, int *);
int delq(int *, int *, int *);
void display(int *);
void  main( )
{
        int arr[MAX];
        int i, front, rear;
        /* initialize data member */
        front = rear = -1;
        for(i = 0; i <MAX; i++)
                arr[i] = 0;
        addq(arr, 14, &front, &rear);
        addq(arr, 22, &front, &rear);
        addq(arr, 13, &front, &rear);
        printf("\nElements in the circular queue\n");
        display(arr);
        i =delq(arr, &front, &rear);
        printf("item deleted: %d\t", i);
        i =delq(arr, &front, &rear);
        printf("item deleted: %d\t\n", i);
        printf("\nElements in the circular queue after deletion\n");
```

```
        display(arr);
        addq(arr, 21, &front, &rear);
        addq(arr, 18, &front, &rear);
        addq(arr, 9, &front, &rear);
        printf("\nElements in the circular queue after addition\n");
        display(arr);
        getch( );
}
/* adds an element to the queue */
void addq(int *arr, int item, int *pfront, int *prear)
{
        if ((*prear == MAX – 1 && *pfront == 0) ||
                (*prear + 1 == *pfront))
        {
                printf("\n Queue is full");
                 return ;
        }
        if (*prear == MAX – 1)
                *prear = 0;
        else
                (*prear)++;
        arr[*prear] = item;
        if(*pfront == -1)
        *pfront = 0;
}
/* removes an element from the queue */
int delq(int *arr, int *pfront, int *prear)
{
        int  data;
        if(*pfront == -1)
        {
                printf("\n Queue is empty");
                 return  NULL;
        }
```

```
        data = arr[*pfront];
        arr[*pfront] = 0;
        if(*pfront == *prear)
        {
            *pfront = -1;
            *prear = -1;
        }
        else
        {
            if(*pfront == MAX -1)
                *pfront = 0;
            else
                (*pfront)++;
        }
        return data;
    }
    /* displays element in a queue */
    void display(int * arr)
    {
        int i;
        printf("\n");
        for(i = 0; i <MAX; i++)
            printf("%d\t", arr[i]);
        printf("\n");
    }
```

Output Of The Program:

```
Elements in the circular queue
14    22    13    0    0    0    0    0    0    0
item deleted: 14        item deleted: 22
Elements in the circular queue after deletion
0    0    13    0    0    0    0    0    0    0
Elements in the circular queue after addition
0    0    13    21    18    9    0    0    0    0
```

Here the arr is used to stored the elements of the circular queue. The function **addq( )** and **delq( )** are used to add and remove the elements from the queue. The function **display( )** displays the existing elements of the queue. The initial value of front and rear are set -1, which indicates that the queue is empty.

In main( ), first we have called the **addq( )** function to insert elements in the circular queue. In this function, *following cases are considered before adding element into the circular queue.*

- First we have checked whether or not the array is full. The message "queue is full " gets displayed if front and rear are in adjacent locations with rear following front.
- If the value of front -1 then indicated the queue is empty and element to be added would be the first element in the queue. The values of front and rear in such case are set to 0 and the new element gets placed at the 0th position.
- It may also happen that some of the position at front end of the array is vacant. This happens if we have deleted some elements from the queue, when the value of rear is MAX – 1 and value of front is greater than 0. in such case the value of rear is set to 0 and element to be added at this position.
- The element is added at rear position in case the value of front is either equal to or greater that 0 and the value of rear is less than Max – 1.

Fig.4.6.5 shows the pictorial representation of a circular queue after adding elements.

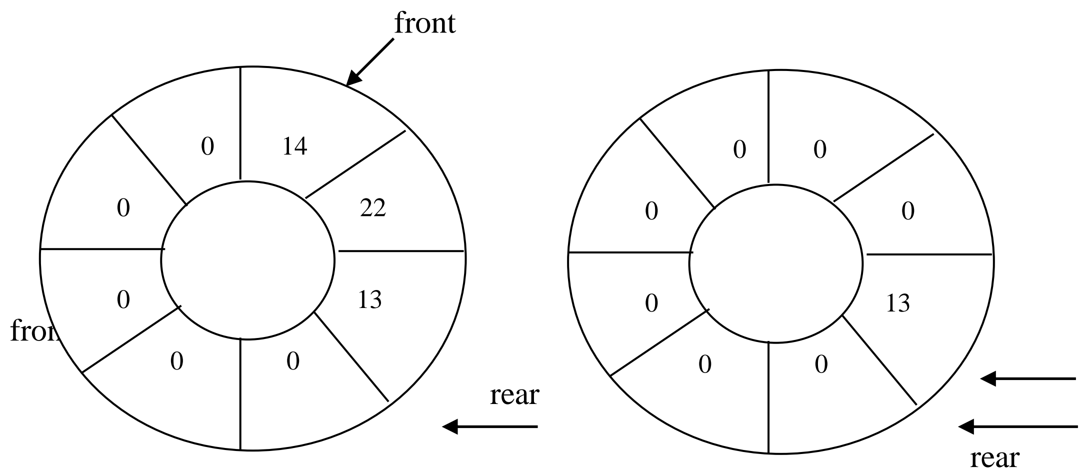


Fig.4.6.5. Circular queue After addition**Fig.4.6.6.** Circular queue After deletion

Next we have called delq( ) function twice to remove 2 elements from the queue. *The following condition is checked while deleting an element.*

- First we have checked whether or not the queue is empty. The value of front in our case is 4, hence an element at the front position would get removed.
- Next, we have checked if the value of front has become equal to rear. If it has, then the element we wish to remove is the only element of the queue. On removal of this element the queue would empty and hence the values of front and rear are set to -1.

On deleting an element from the queue the value of front is set to 0 if it is equal to MAX – 1. Otherwise front is simple incremented by 1. Fig.4.6.6 shows the pictorial representation of circular queue after deleting two elements from the queue.

4.6.6. Self Assessment Questions

Fill in the blank

1. Queue is a ______________ data structure.

True / False

1. The end at which a new element gets added to a queue is called rear end.

Multiple Choice

1. The end at which a new element gets removed from a queue is called

a) frontb) rear c) top d) bottom

2. Queue is also called

a) Last In First Out data structure b) First In First Out data structure

c) Last In Last Out data structure a) First In Last Out data structure

Short Answer

1. What is advantage to represent queue as a linked list?

4.7. Linked List

4.7.1. Definition

List is an ordered set of elements the *general form* of the list is

$$L = \{A_1, A_2, A_3, \text{---------}A_n\}$$

Where

A_1 - First element of the list

A_n - Last element of the list

N – is the size of the list

If the element at position i is A_i, then its successor is A_{n+1} and its predecessor is A_{n-1}. Linked list consists of series of nodes. Each node contains the two fields namely data field and link field i.e. data field contains actual element and link field is used to point the address of next element. The pointer of the last node points to NULL. The structure of the node is given below. The Linked list structure and sample structure is shown in Fig.4.7.1

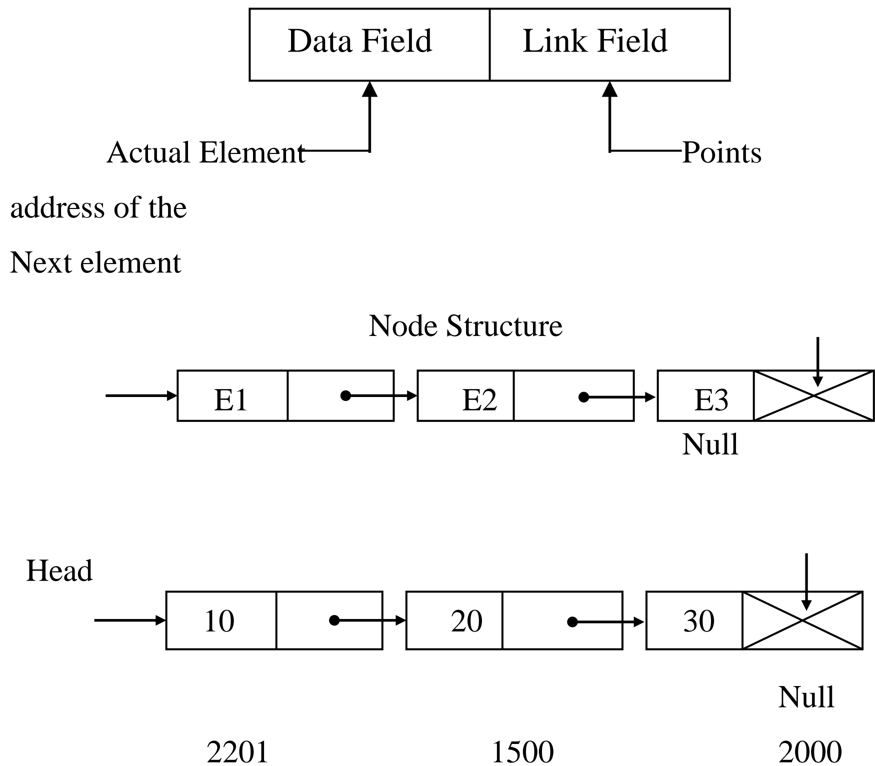


Fig. 4.7.1 Linked list structure and sample structure

Linked list nodes are not stored contiguously in memory, but they are logically contiguous. The address stored in linked list is of *three types* as follows.

- External address: The address of the first node, which is stored in the head pointer.
- Internal address : The address of the inner nodes, which is stored in the link field of predecessor node.
- Null address : Address stored by the Null values, which is of the last node in the link field.

Types of Linked List

A linked list can have a partial or full link among its nodes. The nodes can have only single link to predecessor or can have double links to both its predecessor and successor. Some times it can have a type of link where the last node and first node are connected to maintain the serializability. Accordingly, they are *three different types* of linked lists. They are

- Singly Linked List
- Doubly Linked List
- Circular Linked List

4.7.2. Operations On Singly Linked List

Linked list in which each node contains only one link field pointing to next node. It is referred as **singly linked list** or **linear linked list**. Singly linked list structure is as shown in Fig.4.7.1. A singly linked list can be traversed only one direction. That is head to null. The singly linked list has the following *properties*.

- Referred as linear linked list
- Each node has single link to its next node.
- Traversed only one direction, from head to null

Basic operations on singly linked list

- List creation
- Node insertion
- Node deletion
- List traversal

Declaration for linked list

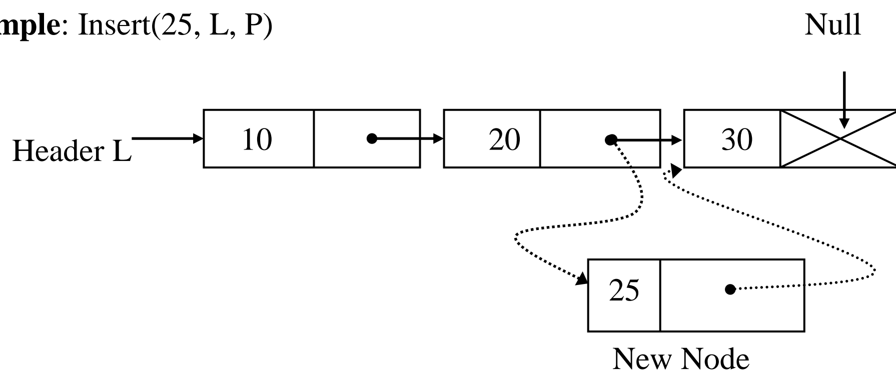
```
struct node;
typedef struct Node *List;
typedef struct Node *Position;
```

```
int IsLast(List  L);
int IsEmpty(List  L);
position Find(int  x , List  L);
void Delete(int  x ,  List  L);
position FindPrevious(int  x , List  L);
position FindNext(int  x , List  L);
void Insert(int  x , List  L , Position P);
void DeleteList(List L);
struct Node
{
      int   element;
      Position  Next;
}
```

Routine to Insert an Element in the List

```
void Insert(int  x , List  L , Position P)
{
      Position     NewNode;
      NewNode = malloc(sizeof(struct  Node));
      if( NewNode != NULL)
      {
                NewNode  -> element = x;
                NewNode -> Next = P -> Next;
                P -> Next = NewNode;
      }
}
```

Example: Insert(25, L, P)



Routine to Check whether the List is Empty

```
int IsEmpty( List  L)  /* return 1 if  L is empty */
{
      if( L -> Next = = NULL)
      return( 1 );
}
```

Routine to Check whether the Current position is Last

```
int IsLast(Position   P ,  List  L) /* return 1 if  P is the last position in L
      */
{
      if(P->Next = = NULL)
         return(1);
}
```

Routine to Find whether the Element in the List

```
Position Find( int x , List L)
{
      /* Return the position of x in L; Null if x is not found */
      Position P;
      P = L-> Next;
      while(P != NULL && P->Element != x)
                  P = P->Next;
      return P;
}
```

Routine to Find Previous

```
Position FindPrevious(int x, List L)
{
      /* Return the position of the predecessor */
```

```
    Position P;
    P = L;
    while( P->Next != NULL && P->Next->element != x)
            P = P->Next;
    return P;
}
```

Routine to Find Next

```
Position FindNext(int x, List L)
{
    /* Return the position of its successor  */
    Position P;
    P = L;
    while( P->Next != NULL && P->element != x)
            P = P->Next;
    return P;
}
```

Routine to Delete an element from the List

```
void Delete(int x , List L)
{
/* Delete the first occurrence of  x from the List */
Position P, Temp;
P = FindPrevious(x , L);
if ( !IsLast(P, L))
{
    Temp = P->Next;
    P->Next = Temp->Next;
    free(Temp);
```

```
}
}
```

Routine to Delete the List

```
void DeleteList(List L)
{
Position P, Temp;
P = L->Next;
L->Next = NULL;
while (  P != NULL)
{
    Temp = P->Next;
    free(P);
    P = Temp;
}
}
```

4.7.3. Circular Linked List

The linked lists that we have seen so far are often known as linear linked lists. All elements of such a linked list can be accessed by first setting to the first node in the list and then traversing the entire list using pointer. Although a linear linked list is useful data structure, it has several short comings. For **example,** given a pointer p to a node in a linear list, we cannot reach any of the nodes that precede the node to which p is pointing.

This disadvantage can be overcome by making a small change to the structure of a linear linked list such that the link field in the **last node contains a pointer back to the first node** rather than a **NULL**. Such a list is called a **circular linked list** and is illustrated in Fig.4.7.2

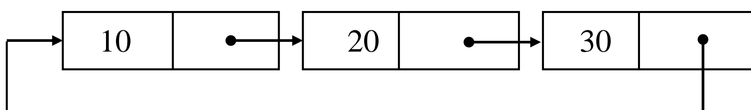


Fig.4.7.2 Circular linked list

From any point in such a list it is possible to reach any other point in the list. A circular linked list does not have a first or last node. We must, therefore, establish a first and last node by convention. The circular linked list can be used to represent a stack and queue.

The following program implements a queue as a circular linked list.

Program: Program to implements queue as a circular linked list.

```
#include<stdio.h>
#include<alloc.h>
/*structure containing  a data part and link part */
struct node
{
      int data;
      struct node *link;
};
void addcirq(struct node **, struct node **, int);
int delcirq(struct node **, struct node **);
void cirdisplay(struct node *);
void main( )
{
      struct node *front, *rear;
      front = rear = NULL;
      addcirq(&front, &rear, 10);
      addcirq(&front, &rear, 20);
      addcirq(&front, &rear, 30);
      printf("\n\nBefore deletion:");
      cirdisplay(front);
      delcirq(&front, &rear);
      delcirq(&front, &rear);
      printf("\n\nAfter deletion:");
      cirdisplay(front);
}
/* adds a new element at the end of the queue */
void addcirq(struct node **f, struct node **r, int item)
{
```

```
        struct node *q;
        /*create new node */
        q = malloc(sizeof(struct node));
        q->data = item;
        /*if the queue is empty */
        if( *f == NULL)
                *f = q;
        else
                (*r)->link = q;
        *r = q;
        (*r)->link = *f;
}
/* removes an element from front of queue */
int delcirq(struct node **f, struct node **r)
{
        struct node *q;
        int item;
        /* if queue is empty */
        if( *f == NULL)
             printf("Queue is empty");
        else
             {
                     if(*f == *r)
                     {
                          item = (*f)->data;
                          free(*f);
                          *f =NULL;
                          *r= NULL;
                     }
                     else
                     {
                          /* delete the node */
                           q= *f;
                          item = q->data;
```

```
                *f = (*f)->link;
                (*r)->link = *f;
                free(q);
            }
            return(item);
        }
        return NULL;
    }
    /* display whole of the queue */
    void cirdisplay(struct node *f)
    {
        struct node *q = f, *p = NULL;
        /* traverse the entire linked list */
        while( q != p)
        {
            printf("%d\t", q->data);
            q = q->link;
            p = f;
        }
    }
```

Output Of The Program

Before deletion: 10 20 30

After deletion: 30

The pointers front and rear point to the first node and last node respectively. To begin with both the pointers front and rear are initialized to NULL. The functions defined in the program are discussed as follows.

Function addcirq( )

This function accepts three parameters. First parameter receives the address of the pointer to the first node (i.e. address of front), the second parameter receives the address of the pointer to the last node (i.e. address of read). The third parameter is the item that holds the data that we need to add to the list.

Then memory is allotted for the new node whose address is stored in pointer q. Then the data, which is present in item, is stored in the data part of the new node.

Next a condition is checked, whether the new node is being added to an empty list. If the list is empty then the address of the node is stored in front. This is done through the statement

*f=q;

After this statement

*r=q;

is executed which stores the address of the new node into rear. Thus both front and rear point to the same node.

Now the statement

(*)->link=*f;

is executed which stores the address of the front node in the link part of the rear node. This is done, because it is the property of a circular linked list that the link part of the last node should contain the address of the first node.

If the new node that is to be added is not the first node then the address present in the link part of the last node is overwritten with the address of the new node, which is done through the statement

(*r)->link=q;

Now the address of the new node is stored in the pointer rear through the statement

*r=q;

and the address of the first node is stored in the link part of the new node. This done through the statement

(*r)->link=*f;

Fig.4.7.3 shows how to add a new node in the circular queue maintained as a linked list.

Function delcirq( )

This function receives two parameters. The first parameter is the pointer to the front and the second is the pointer to the rear. Then a condition is checked whether the list is empty or not. If the list is empty then the control returns back to calling function.

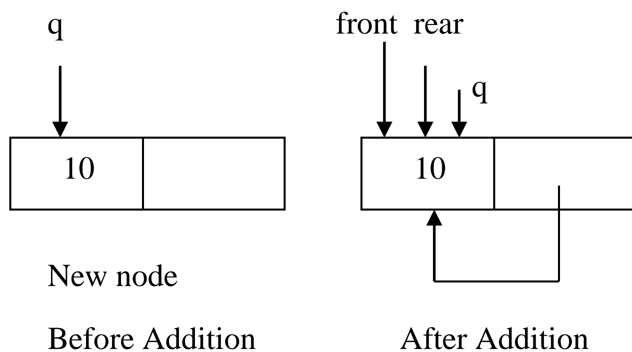
If the list is not empty then it is checked whether the front and rear pointer to the same node or not. If they point to the same node then the memory occupied by the node is released and front and rear both are assigned a NULL value.

If front and rear are pointing to different nodes then the address of the first node is stored in a pointer q. Then the front pointer is made to point to the next node in the list. i.e. to the node which is pointed by (*f)->link. Now the address of the front is stored in the link part of last node. Then memory occupied by the node being deleted is released.

Fig.4.7.4 shows how the deletion of a node from the circular queue maintained as a linked list happens.

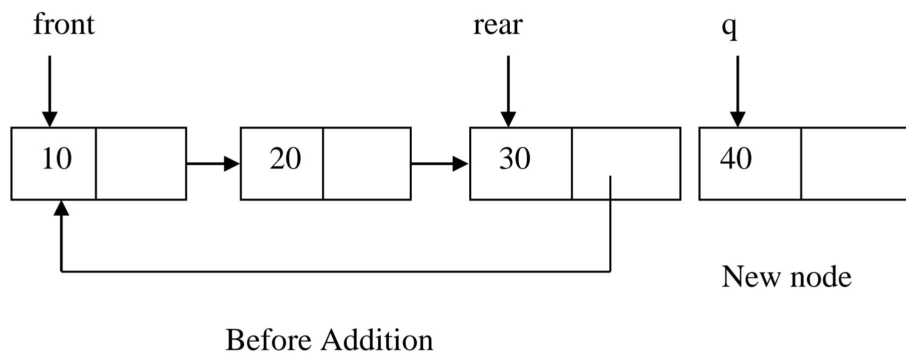
1. Addition of node to empty list

front = rear = NULL

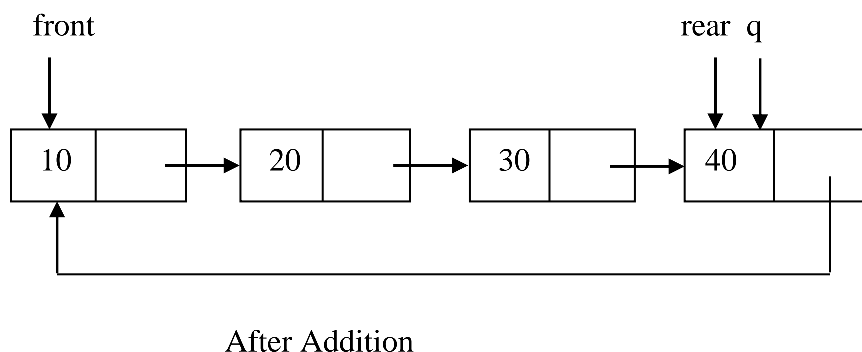


Before Addition After Addition

2. Addition of node to non-empty list



Before Addition



After Addition

Fig 4.7.3 Addition of a node in circular linked list

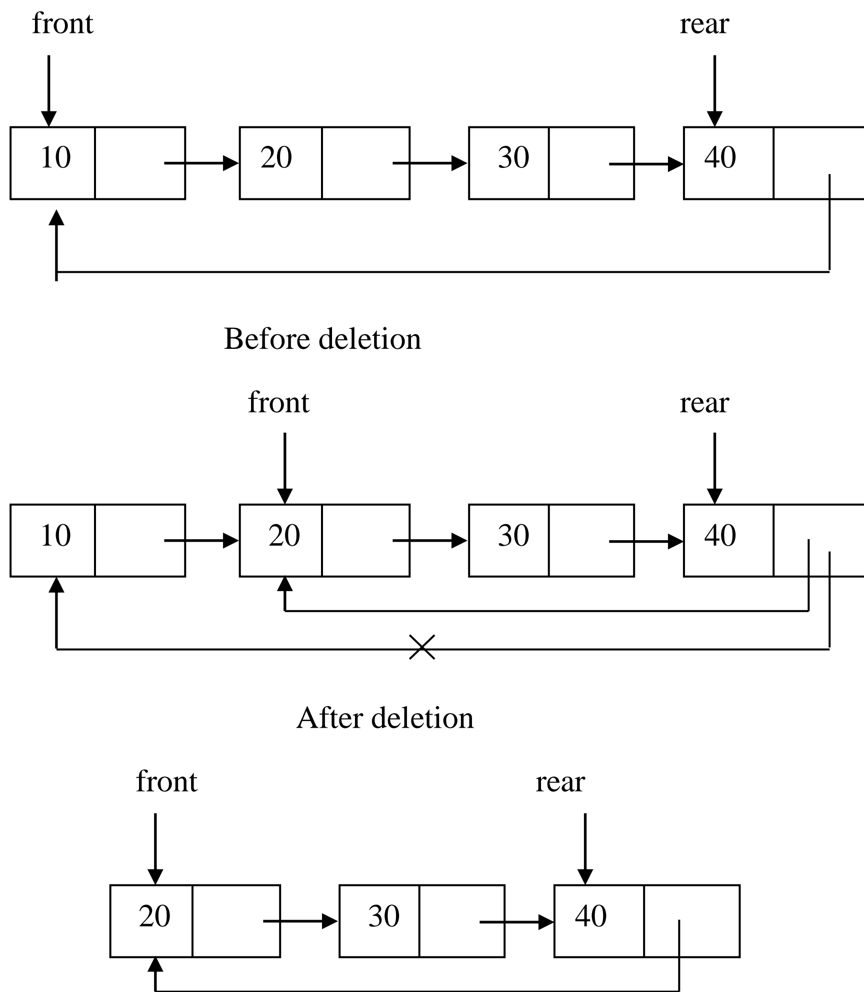


Fig.4.7.4 Deletion of a node from circular linked list

Function cirdisplay( )

This function receives the pointer to the first node in the list as a parameter. Then q is also made to point to the first node in the list. This is done because the entire list is traversed using q. Another pointer p is set to NULL initially. Then through the loop the circular linked list is traversed till the time we do not reach the first node again. We would make the circle and reach the first node when q equals p.

First time through the loop p is assigned the address of the first node after q has been moved to the next node. Had we done this before the loop then the condition in the loop would have failed the first time itself.

4.7.4. Doubly Linked List

In the linked lists that we have used so far each node provides information about where is the next node in the list. It has no knowledge about where the previous node lies in memory. If we are at say the 15th node in the

list, then to reach the 14th node we have to traverse the list right from the first node.

To avoid this we can store in each node not only the address of next node but also the address of the previous node in the linked list. This arrangement is often known as a 'Doubly Linked List' and is shown in Fig.4.7.5. the following program implements the doubly linked list.

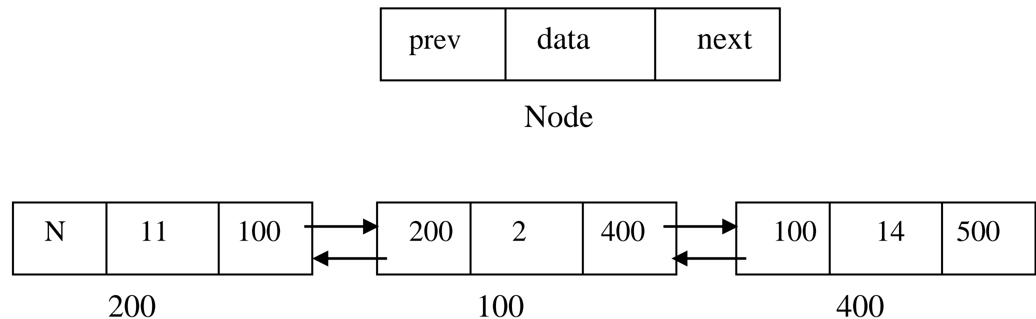


Fig.4.7.5 Double linked list

Program: Program to implements doubly linked list

```
#include<stdio.h>
#include<alloc.h>
/* structure representing a node of the doubly linked list */
struct dnode
{
        struct dnode *prev;
        int data;
        struct dnode *next;
};
void d_append(struct dnode ** , int);
void d_addatbeg(struct dnode **, int);
void d_addafter(struct dnode *, int, int);
void d_display(struct dnode *);
int d_count(struct dnode *);
void d_delete(struct dnode **, int);
void main( )
{
        struct dnode *p;
        p = NULL;
```

```
        d_append(&p, 10);
        d_append(&p, 20);
        d_append(&p, 30);
        d_display(p);
        printf("\nNo. Of elements in the list = %d\n", d_count(p));
        d_addatbeg(&p, 11);
        d_addatbeg(&p, 22);
        d_display(p);
        printf("\nNo. Of elements in the list = %d\n", d_count(p));
        d_addafter(p, 4, 66);
        d_addafter(p, 2, 76);
        d_display(p);
        printf("\nNo. Of elements in the list = %d\n", d_count(p));
        d_delete(&p, 20);
        d_delete(&p, 66);
        d_display(p);
        printf("\nNo. Of elements in the list = %d\n", d_count(p));
}
/* add a new node at the end of the doubly linked list */
void d_append(struct dnode **s, int num)
{
        struct dnode *r, *q = *s;
        /* if the linked list is empty */
        if( *s == NULL)
        {
                /* create a new node */
                *s = malloc(sizeof(struct dnode));
                (*s)->prev = NULL;
                (*s)->data = num;
                (*s)->next = NULL;
        }
        else
        {
                /* traverse the linked list till the last node is reached */
```

```
                while( q->next != NULL)
                        q = q->next;
                /* add a new node at the end */
                r = malloc(sizeof(struct dnode));
                r->data = num;
                r->next = NULL;
                r->prev = q;
                q->next = r;
        }
}
/* adds a new node at the beginning of the list */
void d_addatbeg(struct dnode **s, int num)
{
        struct dnode *q ;
        /* create a new node */
        q = malloc(sizeof(struct dnode));
        /* assign data and pointer to the new node */
        q->prev = NULL;
        q->data = num;
        q->next= *s;
        /* make new node the head node */
        (*s)->prev = q;
        *s = q;
}
/* adds a new node after the specified number of nodes */
void d_addafter(struct dnode *q, int loc, int num)
{
        struct dnode *temp ;
        int i;
        /* skip to desired portion */
        for(i = 0; i < loc; i++)
        {
                q = q->next;
                /* if end of the list is encountered */
```

```
		if ( q == NULL)
		{
			printf("\nThere are less than %d elements", loc);
			return;
		}
	}
	/* insert new node */
	q = q->prev;
	temp = malloc(sizeof(struct dnode));
	temp->data = num;
	temp->prev = q;
	temp->next = q->next;
	temp->next->prev = temp;
	q->next = temp;
}
/*display the contents of the list */
void d_display(struct dnode *q)
{
	printf("\n");
	/* traversethe entire list */
	while(q != NULL)
	{
		printf("%2d\t", q->data);
		q = q->next;
	}
}
/* counts the number of nodes in the list */
int d_count(struct dnode *q)
{
	int c = 0;
	/* traverse the entire list */
	while(q != NULL)
	{
		q = q->next;
```

```
            c++;
        }
        return c;
}
/* deletes the specified node from the list */
void d_delete(struct dnode **s, int num)
{
        struct dnode *q = *s;
         /* traverse the entire list */
        while(q != NULL)
        {
            /* if node to be deleted is found */
            if(q->data == num)
            {
            /* if node to be deleted is the first node */
            if(q == *s)
            {
                *s = (*s)->next;
                (*s)->prev = NULL;
            }
            else
            {
                /* if node to be deleted is the last node */
                if(q->next == NULL)
                    q->prev->next = NULL;
                else
                {
            /* if node to be deleted is the intermediate node */
                    q->prev->next =  q->next;
                    q->next->prev = q->prev;
                }
                free(q);
            }
            return; /* return back after deletion */
```

```
                }
                q = q->next; /* go to next node */
        }
        printf("\n%d not found", num);
}
```

Output of The Program

```
10    20    30
No. Of elements in the list = 3
22    11    10    20    30
No. Of elements in the list = 5
22    11    76    10    20    66    30
No. Of elements in the list = 7
22    11    76    10    30
No. Of elements in the list = 5
```

4.7.5. Operations On Doubly Linked List

Function d_append( )

The **d_append( )** function adds a node at the end of the existing list. It also checks the special case of adding the first node if the list is empty.

This function accepts *two parameters*. The first parameter ***s** is of type struct dnode*** which contains the address of the pointer to the first node of the list or a NULL value in case of empty list. The second parameter **num** is an *integer*, which is to be added in the list.

To begin with we initialize **q** which is of the type *struct dnode** with the value stored at **s**. This is done because using **q** the entire list is traversed if it is non-empty.

If the list is empty then the condition

```
if (*s==NULL)
```

Gets satisfied. Now memory is allocated for the new node whose address is stored in ***s** (i.e. p). Using s a NULL value is stored in its **prev** and **next** links and the value of **num** is assigned to its data part.

If the list is non-empty then through the statements

```
while(q->next!=NULL)
        q=q->next;
```

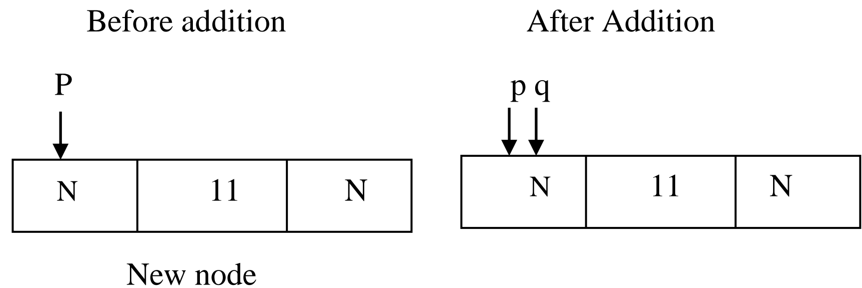
q is made to point to the last node of the list.

Addition of new node at the end

1. **Addition to an empty linked list**

Related function: d_append( )

p = *s = NULL



2. **Addition to an existing linked list**

Related function :d_append( )

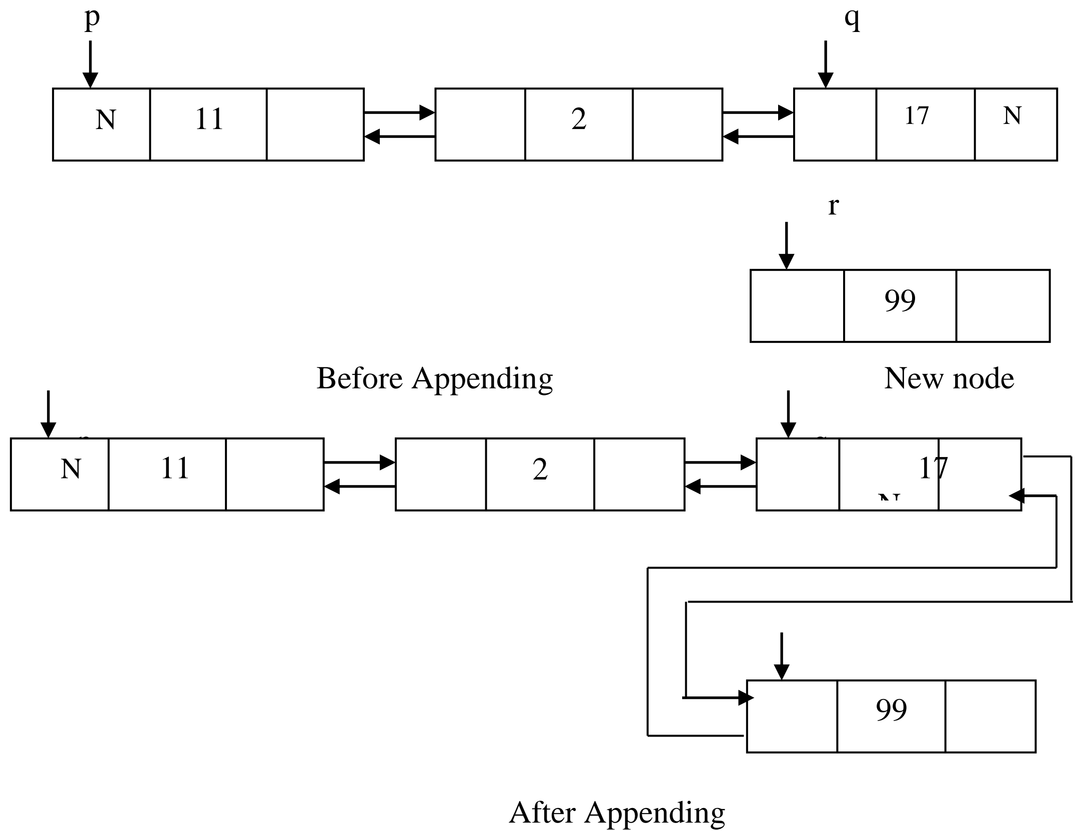


Fig.4.7.6 Doubly linked list addition of a node

Then memory is allocated for the node whose address is stored in **r**. A Null value is stored in the next part of this node, because this is going to be last node. Now what remains to be done is to link this node with rest of the list. This is done through the statements

r->prev=q;

q->>=r;

The statement r->prev=q makes the **prev** part of the new node **r** to point to the previous node **q**. The statement q->next=r makes the next part of **q** to the last node r. This is shown in Figure.

Function d_addatbeg( )

The **d_addatbeg( )** function adds a node at the beginning of the existing list. This function accepts two parameters. The first parameter **s** is of type *struct dnode*** which contains the address of the pointer to the first node and the second parameter **num** is an *integer*, which is to be added in the list.

Memory is allocated for the new node whose address is stored in **q**. Then num is stored in the data part of the new node. A NULL value is stored in **prev** part of the new node as this is going to be the first node of the list. The next part of this new node should contain the address of the first node of the list. This is done through the statement

q->next=*s;

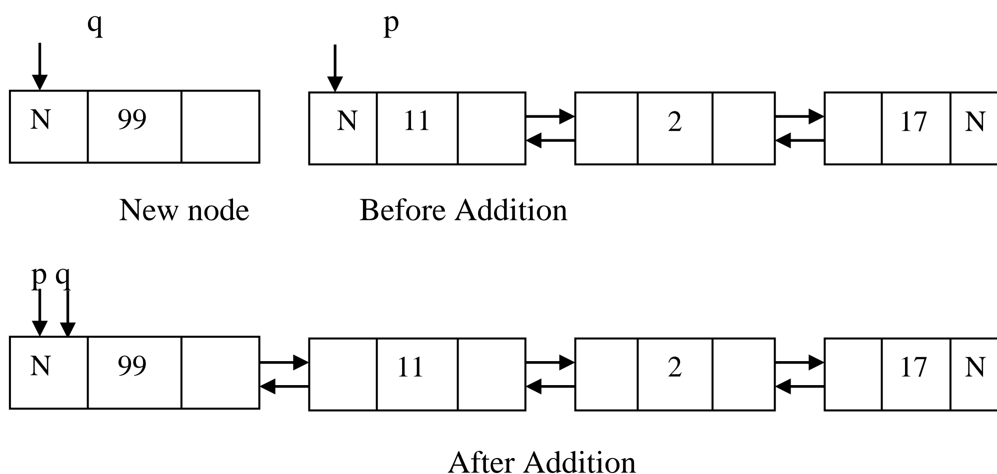
Now what remains to be done is to store the address of this new node into the **prev** part of the first node and make this new node the first node in the list. This is done through the statements

(*s)->prev=q; *s=q;

These operations are shown in Fig.4.7.7.

Addition of new node at the beginning

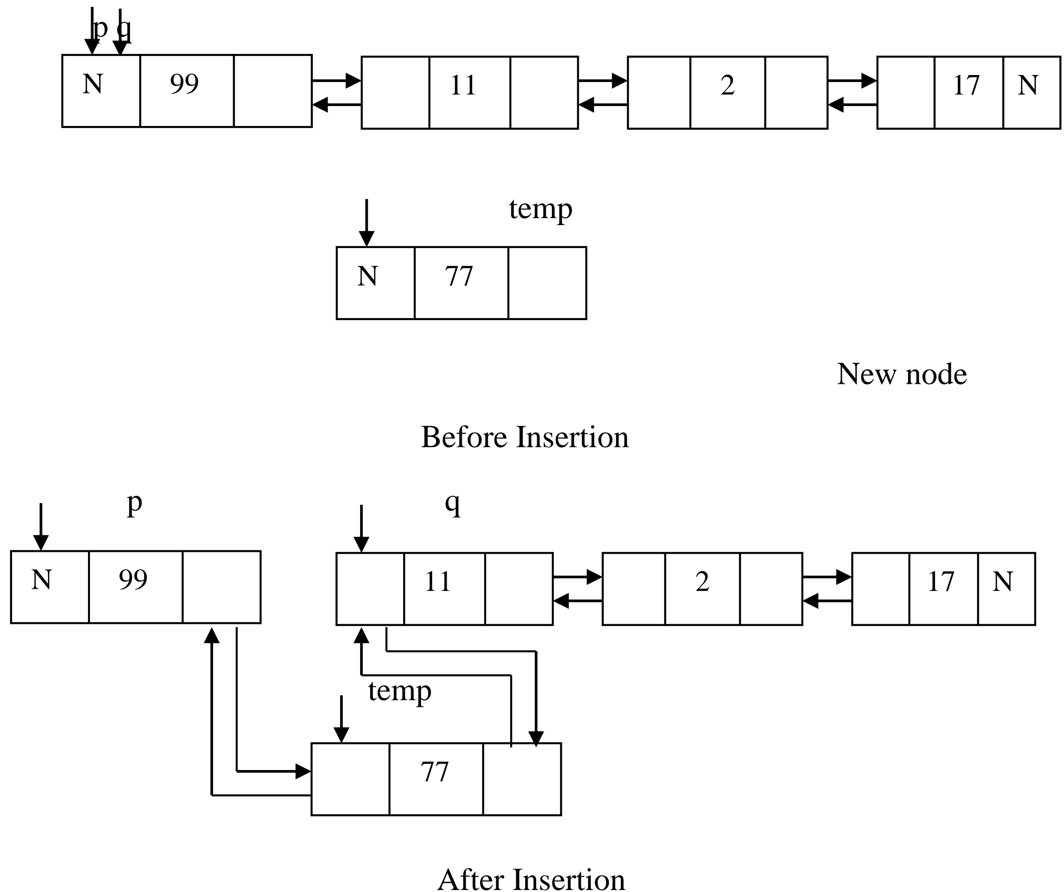
Related function: d_addatbeg( )



After Addition

Insertion of new node after a specified node

Related Function : d_addafter



After Insertion

Fig.4.7.7 Working of d_addatbeg( ) and d_addafter( )

Function d-addafter( )

The **d_addafter( )** function adds a node at the specified position of an existing list. This operation of adding a new node in between two existing nodes can be better understood with the help of Fig.4.7.7.

This function accepts three parameters. The first parameter **q** points to the first node of the list. The second parameter **loc** specifies the node number after which the new node must be inserted. The third parameter **num** is an integer, which is to be added to the list.

A loop is executed to reach the position where the node is to be added. This loop also checks whether the position loc that we have mentioned. Really occurs in the list or not. When the loop ends, we reach the **loc** position where

the node is to be inserted. By this time **q** is pointing to the node before which the new node is to be added.

The statement

q = q->prev;

Makes **q** to point to the node after which the new node should be added. Then memory is allocated for the new node and its address is stored in **temp**. The value of **num** is stored in the data part of the new node.

The **prev** part of the new node should point to **q**. This is done through the statement

temp->prev = q;

Then **next** part of the new node should point to the node whose address is stored in the **nex**t part of node pointed to by **q**. This is achieved through the statement

temp->next=q->next;

Now what remains to be done is to make **prev** part of the next node(node pointed by q->next) to point to the new node. This is done through the statement

temp->next->prev=temp;

At the end, we change the next part of th **q** to make it point to the new node, and this is done through the statement

q->next=temp;

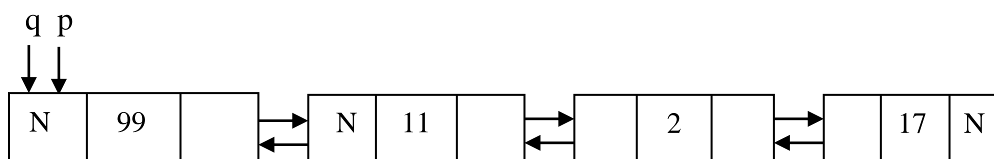
Function d_delete( )

The function **d_delete( )** deletes a node from the list if the data part of that node matches num. This function receives *two parameters*. The first parameter is the *address of the pointer to the first node* and the second parameter is the *number* that is to be deleted.

Deletion of node

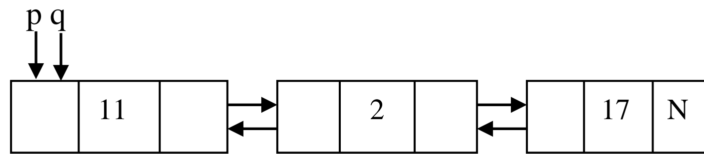
1. **Deletion of first node**

Related function: d_delete( )



Node to be deleted 99

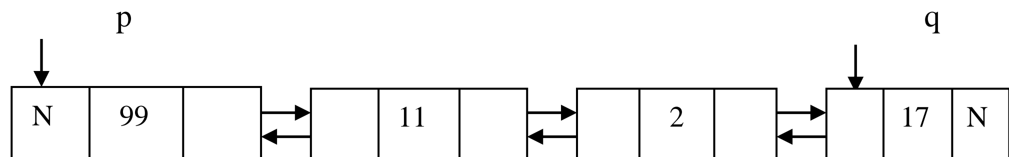
Before deletion



After Deletion

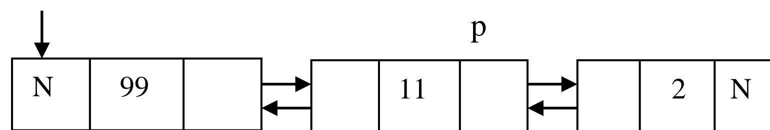
2. **Deletion of last node**

Related function: d_delete( )



Node to be deleted 17

Before Deletion



After Deletion

Fig.4.7.8 Working of d_delete( ) function

We run a loop to traverse the list. Inside the loop the data part of each node is compared wit the **num** value. If **num** value matches the data part in the node then we need to check the position of the node to be deleted.

If it happens to be the first node, then the first node is to point to the **next** part of the first node. This is done through the statement

```
*s=(*s)->next;
```

Then, a value NULL is stored in **prev** part of the second node, since it is now going to become the first node. This is done through the statement

```
(*s)->prev=NULL;
```

If the node to be deleted happens to be the last node,then a value NULL is stored in the next part of the second last node. This is done through the statements

```
If(a->next==NULL)
        q->prev->next=NULL;
```

If the node to be deleted happens to be any intermediate node, then the address of the next node is stored in the **next** part of the previous node and the

address of the previous node is stored in the **prev** part of the next node. This is done through the statement

```
q->prev->next=q->next;
q->next->prev=q->prev;
```

Finally the memory occupied by the node being deleted is released by calling the function **free( )**. Fig4.7.8 shows the working of the **d_delete( )** function.

4.7.6. Polynomial Addition

Polynomials like $5x^4 + 2x^3 + 7x^2 + 10x - 8$ can be maintained using a linked list. To achieve this each node should consist of three elements namely coefficient, exponent and a link to the next term.

While maintaining the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. Once we build a linked list to represent a polynomial we can use such lists to perform common polynomial operations like addition and multiplication.

Program: Program to add two polynomials

```
#include<stdio.h>
#include<malloc.h>
struct polynode
{
    float coeff;
    int exp;
    struct polynode *link;
};
void poly_append(struct polynode **, float, int);
void display_poly(struct polynode *);
void poly_add(struct polynode *,struct polynode *, struct polynode **);
void main( )
{
    struct polynode *first, *second, * total;
    first = second = total = NULL;
    poly_append(&first, 1.4, 5);
    poly_append(&first, 1.5, 4);
    poly_append(&first, 1.7, 2);
    poly_append(&first, 1.8, 1);
```

```
        poly_append(&first, 1.9, 0);
        display_poly(first);
        poly_append(&second, 1.5, 6);
        poly_append(&second, 2.5, 5);
        poly_append(&second, -3.5, 4);
        poly_append(&second, 4.5, 3);
        poly_append(&second, 6.5, 1);
        printf("\n\n");
        display_poly(second);
        poly_add(first, second, &total);
        printf("\n\n");
        display_poly(total);
}
/* adds a term to a polynomial */
void poly_append(struct polynode **q, float x, int y)
{
        struct polynode *temp;
        temp = *q;
        /* creates a new node if the list is empty */
        if( *q == NULL)
        {
            *q = malloc(sizeof(struct polynode));
            temp = *q;
        }
        else
        {
            /* traverse the entire list */
            while(temp->link != NULL)
                    temp = temp->link;
            /* create new nodes at intermediate */
            temp->link = malloc(sizeof(struct polynode));
            temp = temp->link;
        }
        /* assign coeffiecient and exponent */
```

```
        temp->coeff = x;
        temp->exp = y;
        temp->link = NULL;
}
/* displays the contents of linked list that representing polynomial */
void display_poly(struct polynode *q)
{
        /* traverse the entire list */
        while(q != NULL)
        {
            printf("%.1fx^%d :",q->coeff,q->exp);
            q = q->link;
        }
        printf("\b\b\b"); /* erase last colon */
}
/* add two polynomials */
void poly_add(struct polynode *x, struct polynode *y, struct polynode **s)
{
        struct polynode *z;
        /* if both list are empty */
        if( x == NULL && y == NULL)
            return;
        /* traverse till one of the list ends */
        while( x != NULL && y != NULL)
        {
            /* create a new node if the list is empty */
            if( *s == NULL)
            {
                *s = malloc(sizeof(struct polynode));
                z = *s;
            }
            /* create a new node at intermediate */
            else
            {
```

```
        z->link = malloc(sizeof(struct polynode));
        z = z->link;
    }
    /* store a term of the larger degree polynomial */
    if( x->exp < y->exp)
    {
        z->coeff = y->coeff;
        z->exp = y->exp;
        y = y->link ; /* go to next node */
    }
    else
    {
        if( x->exp > y->exp)
        {
            z->coeff = x->coeff;
            z->exp = x->exp;
            x = x->link ; /* go to next node */
        }
        else
        {
            /* add the coeffiecients, when exponents are equal */
            if( x->exp == y->exp)
            {
                z->coeff = x->coeff +y->coeff;
                z->exp = x->exp;
                x = x->link ; /* go to next node */
                y = y->link ;
            }
        }
    }
}
/* assigning remaining terms of the second polynomial to the results */
while ( x != NULL)
{
```

```
        if (*s == NULL)
        {
            *s = malloc(sizeof(struct polynode));
            z = *s;
        }
        else
        {
            z->link = malloc(sizeof(struct polynode));
            z = z->link;
        }
        z->coeff = x->coeff;
        z->exp = x->exp;
        x = x->link;
    }
    while ( y != NULL)
    {
        if (*s == NULL)
        {
            *s = malloc(sizeof(struct polynode));
            z = *s;
        }
        else
        {
            z->link = malloc(sizeof(struct polynode));
            z = z->link;
        }
        z->coeff = y->coeff;
        z->exp = y->exp;
        y = y->link;
    }
    z->link = NULL;
}
```

Output Of The Program

1.4x^5 :1.5x^4 :1.7x^2 :1.8x^1 :1.9x^0 :

1.5x^6 :2.5x^5 :-3.5x^4 :4.5x^3 :6.5x^1 :

1.5x^6 :3.9x^5 :-2.0x^4 :4.5x^3 :1.7x^2 :8.3x^1 :1.9x^0 :

In this program the poly_append( ) function is called several time to build the two polynomials which are pointer to by the first and second. The poly_add( ) is called to carry out the addition of two polynomials. The resulting polynomial are displayed by using the function display_poly( ).

4.7.7. Self Assessment Questions

Fill in the blank

1. All nodes in the linked list stored in ________________ memory locations.

True / False

1. Linked list is used to store similar data item.
2. The link part of the last node in a linked list must contain NULL.

Multiple Choice

1. Doubly linked list facilitates movement from one node to another

a) from left to right	b) from right to left
c) from either direction	c) none of the above

Short Answer

1. What is mean by singly linked list?

4.8. Summary

In this unit we have introduced linear data structure like array , stack, queue and linked list.

The first lesson of this unit ,you have discussed about how to make the arrays. You have also learnt about how various operation performed an different types of array.

The next lesson of this unit, you have learnt the stack ,queue and linked list. You have learnt how the various operation performed on the stack, queue and linked list. You have also learnt about representation stack and queue using array and linked list.

4.9. Unit questions

1.Develop an algorithm to perform various operation on arrays.

2. Write down the procedure for implementing stack and its operation.

3. Discuss the application of stack.

4. How an infix expression is converted to postfix expression?

5. Write down an algorithm for insert and delete an element from a queue.

6. Formulate an algorithm to count number of nodes in the linked list and to free all nodes.

7. Explain how queue can be implemented using arrays.

8. Write the procedure for implementation of polynomial addition using linked list.

9. Explain doubly linked list with the neat diagram.

10. What is the difference between circular queue and queue ? Explain with procedure.

4.10. Answers for Self Assessment Questions

Answer 4.3.3

Fill in the blank

1. linear 2. non-linear

True / False

1. false

Multiple Choice

1. a)

Short Answer

1. A collection of data elements, whose arrangement is characterized by accessing functions that are used to store and retrieve individual data elements are called data structure.

Answer 4.4.4

Fill in the blank

1. related & consecutive 2. 0 to n

True / False

1. False 2. True

Multiple Choice

1. a)

Short Answer

1. Adding a new element to an array

Answer 4.5.6

Fill in the blank

1. LIFO 2. prefix notation

True / False

1. True

Multiple Choice

1. b)

Short Answer

1. An operators precedes the two operands is called postfix expression

Answer 4.6.6

Fill in the blank

1. Linear data structure

True / False

1. True

Multiple Choice

1. a) 2. b)

Short Answer

1. Space for the elements in a linked list is allocated dynamically.

Answer 4.7.7

Fill in the blank

1. non-contiguous

True / False

1. True 2. True

Multiple Choice

1. c)

Short Answer

1. A singly linked list is one type linked list , it allows traversal of the list only one direction

NOTES

UNIT – V

5.1 Introduction

The data structures that we have seen so far (such as linked lists, stacks, queues) were linear data structures. As against this trees and graphs are non-linear data structures. Trees are encountered frequently in every day life.

In tree structure, each node may points to several other nodes (which may then points to several other nodes etc.,) For example, suppose we wish to use a data structure to represent a person and all his or her descendants. Although the nodes in a general tree may contain any number of pointers to the other tree nodes, a larger number data structures have at most two pointers to the other tree nodes. This type of a tree is called a binary tree.

Graphs are data structures which have wide ranging of applications in real life like analysis of electrical circuits, finding shortest routes, statistical analysis, etc.,.

5.2 Objectives

After studying this unit, you should be able to:

- ❖ Understand trees and its terminologies.
- ❖ Understand the concepts of binary tree, various way of traversal in binary tree and representation of binary tree in memory.
- ❖ Understand about the various operation performed on binary tree.
- ❖ Discuss about the forest trees and conversion of forest trees in to binary trees.
- ❖ Understand the graphs and its terminologies.
- ❖ Understand about the various representation of graphs in memory, various way graph traversal.
- ❖ Understand the Shortest Path Algorithm (Using Dijikstra's Algorithm) with related illustrations.

5.3 Trees

5.3.1 Definition & Terminology

The data structure we so far discussed is linear data structures, since they have a *linear relationship between its adjacent elements*. There are many applications in real life situations that make use of non-linear data structures such as graphs and trees. Trees are very flexible, versatile and powerful data structures that can be used to represent data items possessing hierarchical relationships. In a linear data structure, each node has a link which points to another node, whereas in non-linear data structure, each node may point to several other nodes.

Tree definition:

A **tree** is a finite set of one or more nodes such that there is a specially designated node called the root, and zero or more non empty sub trees T_1, T_2, T_3……T_k each of whose roots are connected by a directed edge from root R. Fig.5.3.1 shows a simple tree structure. The tree data structure grows downward from up (level 0) to bottom (level n).

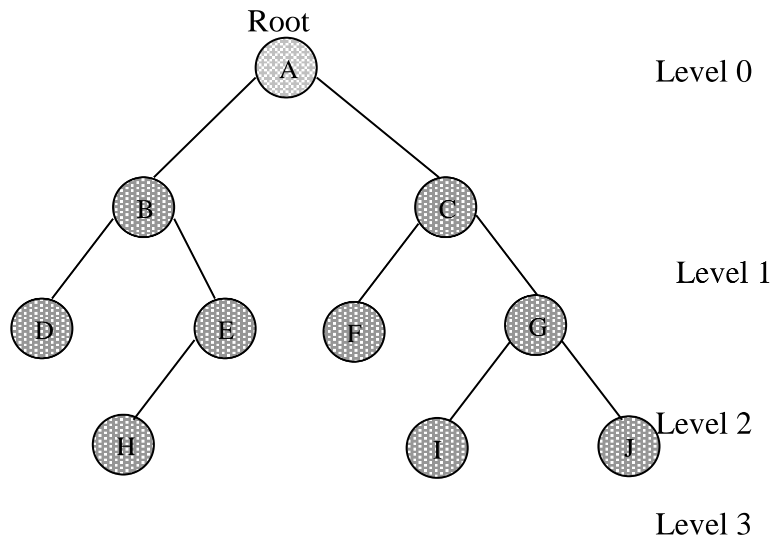


Fig 5.3.1 Tree Structure

Tree terminology

❖ **Root:**

A node, which doesn't have a parent. That is the root is the first and top most node in the hierarchical arrangement of data items. In Fig.5.3.1 node A is the root of the tree.

❖ **Node:**

Each data item in the tree is called node. It is basic data structure that specifies the data information and has links to other data items. The data items A, B, C, D, E, F, G, H, I, J all are nodes of tree in Fig.5.3.1.

❖ **Leaf or Terminal node:**

A node that doesn't have children is called leaf or terminal node. In Fig.5.3.1 D, H, F, I, J are leaf nodes.

❖ **Siblings:**

Children of the same parents are called siblings. In above Fig.5.3.1 the siblings are

Siblings(A) = { B, C}

Siblings(B) = { D, E}

Siblings(C) = { F, G}

Siblings(E) = { H}

Siblings(G) = { I, J}

- **Path:**

A path is a sequence of consecutive edges from the source node to the destination node. There is exactly only one path from each node to root. In Fig.5.3.1. path from A to Leafs D, H, F, I,J.

Path 1: A → B → D

Path 2: A → B → E → H

Path 3: A → C → F

Path 4: A → C → G → I

Path 5: A → C → G → J

- **Length:**

The length is defined as the number of edges on the path.

- **Degree:**

The number of sub trees of a node is called its degree.

Degree of A is 2	Degree of F is 0
Degree of B is 2	Degree of G is 2
Degree of C is 2	Degree of H is 0
Degree of D is 0	Degree of I is 0
Degree of E is 1	Degree of J is 0

- **Edges:**

An edge is connection lines that connect two adjacent nodes of a tree. That is line drawn from one node to another nod is called as edge

- **Level:**

The entire tree structure is leveled; it starting from root node is always at level 0. Then its immediate children are at level 1 and so on. In general, if node is at level n, then its children will be at level n + 1.

- **Depth:**

For any node n, the depth of n is the length of the unique path from root to n. The depth of the root is zero. In Fig.5.3.1 the depth of node F is 2.

- **Height:**

For any node n, the height of the node n is the length of the longest path from n to the leaf. The height of the leaf is zero. In Fig.5.3.1 the height of node F is 0.

Note:

Height of the tree is equal to the height of the root.

Depth of the tree is equal to the height of the tree.

5.3.2 Binary Tree

Binary tree is a tree in which no node can have more than two children. The binary tree having the following *features*.

- ❖ The maximum degree of any node is at most two. That means the degree of a binary node is either zero or one or two.
- ❖ All nodes to the left of the binary tree are referred as left sub tree and all nodes to the right of a binary tree are referred as right sub trees.

The **binary tree** can be representing in *two ways*.

- ❖ Linear Representation (Using Arrays)
- ❖ Linked Representation (Using Pointers)

Types of binary tree

Strict binary tree:

Every non-leaf node consists of non-empty left sub tree and right sub tree. A strict binary tree is shown in Fig. 5.3.2. In this tree, all the non-terminal nodes such as B, C and E are having non-empty left and right sub-trees namely {D}, {F, G}, {H, I} respectively.

Strict binary trees are used to represent any expression in order to evaluate them. A non-leaf node represents every operator, whereas every operand is in a leaf node. The expression tree is shown in Fig.5.3.3.

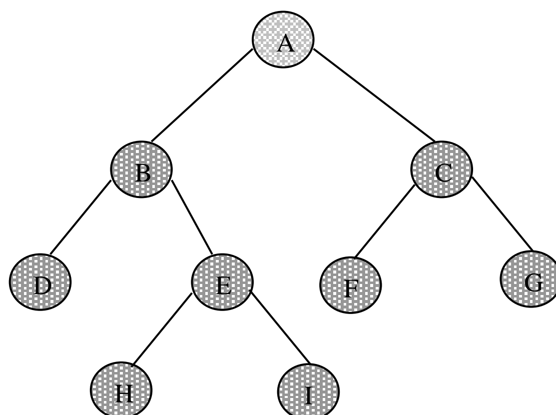


Fig 5.3.2 Strict Binary Tree

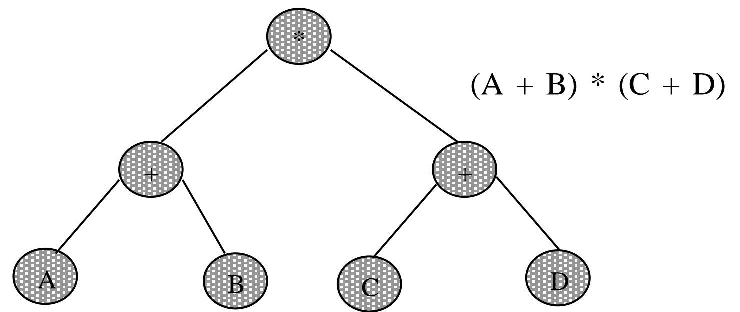


Fig 5.3.3 Expression Tree

Full binary tree:

All leaves are at same level and every non-leaf node has exactly two children. (See Fig.5.3.4)

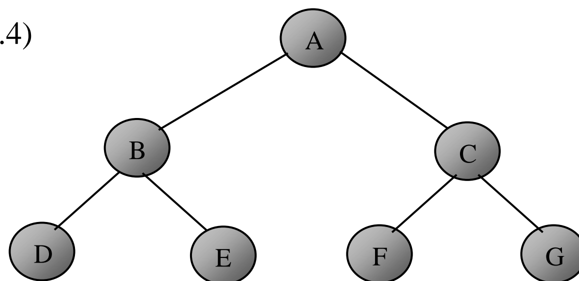


Fig 5.3.4 Full Binary Tree

Complete binary tree:

Every non-leaf node has exactly two children but all leaves are not necessarily at same level. (See Fig. 5.3.5)

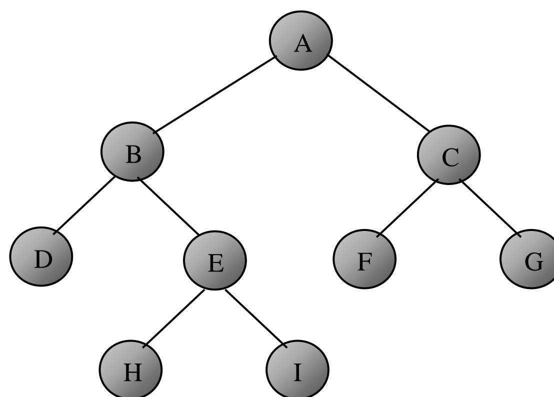


Fig 5.3.5 Complete Binary Tree

Left skewed binary tree:

A binary tree with only left sub trees (See Fig.5.3.6)

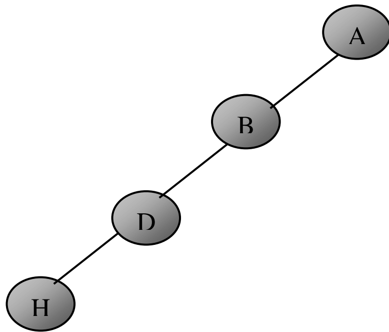


Fig.5.3.6 Left skewed binary tree

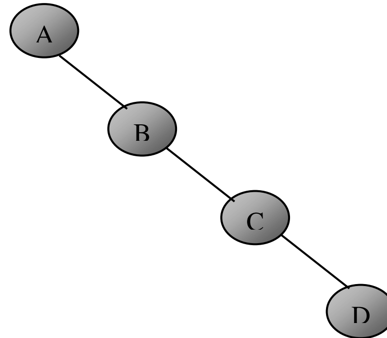


Fig.5.3.7 Right skewed binary tree

Right skewed binary tree:

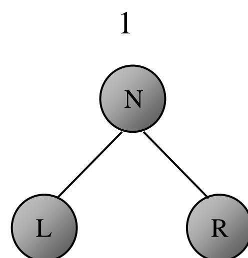
A binary tree with only right sub tree.(See Fig.5.3.7)

5.3.3 Traversal Of A Binary Tree:

5.3.3.1 Introduction

The **traversal of a binary tree** involves *visiting each node in the tree exactly once*. There are many applications that require traversal of binary trees. For **example**, when a tree is used to represent an arithmetic expression, it needs to be traversed to evaluate that expression. There are *three methods* commonly used for binary tree traversal. These *methods are*

- In-order traversal (L N R)
- Pre- order traversal (N L R)
- Post-order traversal (L R N)



Here

N - Root

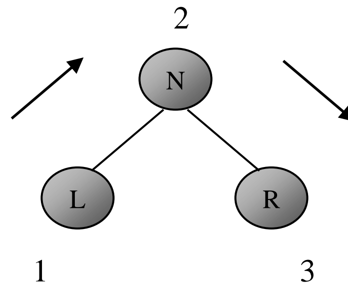
L – Left sub tree

R – Right sub tree

5.3.3.2 In-Order Traversal

The *in-order traversal* of a binary tree is performed as

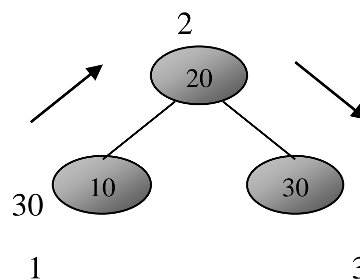
- Traverse the left sub-tree in in-order (L)
- Visit the root (N)
- Traverse the right sub-tree in in-order (R)



Recursive Routine for In-order Traversal

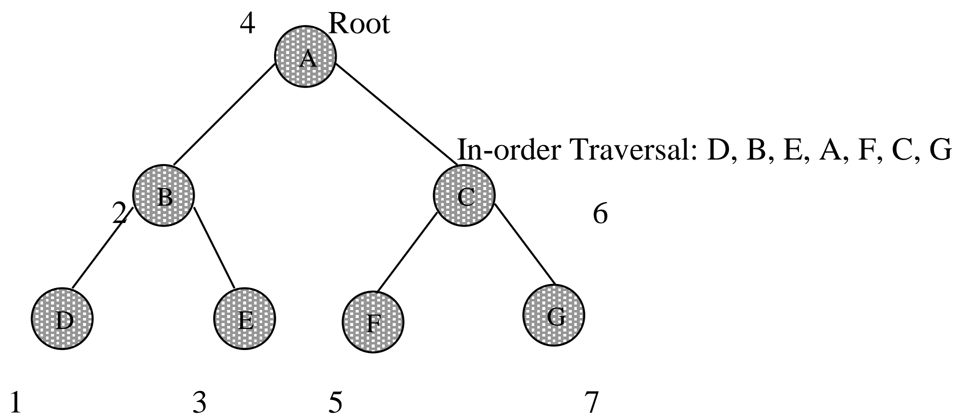
```
void inorder(struct rec * Tree )
{
    if  (tree != NULL)
    {
        inorder(tree -> left);
        printf("%d\n", tree->value);
        inorder(tree -> right);
    }
}
```

Example 1



In-order Traversal: 10, 20, 30

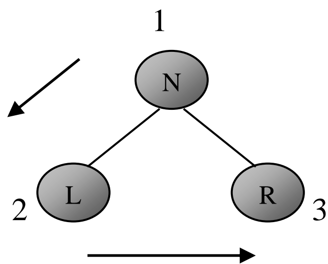
Example 2



5.3.3.3 Pre-Order Traversal

The *pre-order traversal* of a binary tree is performed as

- Visit the root (N)
- Traverse the left sub-tree in pre-order (L)
- Traverse the right sub-tree in pre-order (R)

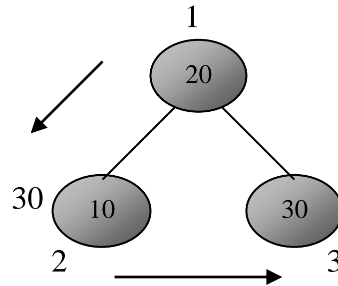


Recursive Routine for pre-order Traversal

```
void preorder(struct rec * Tree )
{
        if  (tree != NULL)
        {
                printf("%d\n", tree->value);
                preorder(tree -> left);
                preorder(tree -> right);
```

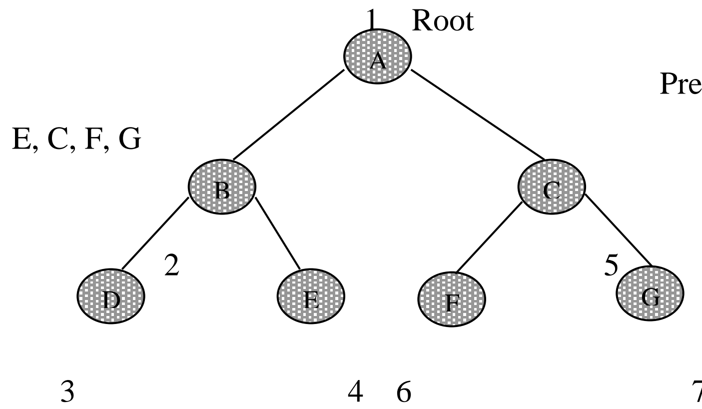
```
            }
        }
```

Example 1



Pre-order Traversal: 20, 10, 30

Example 2

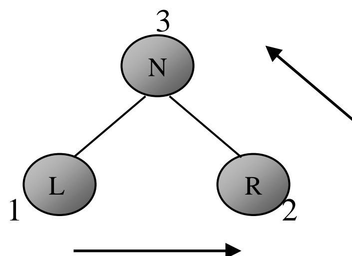


Pre-order Traversal: A, B, D, E, C, F, G

5.3.3.4 Post-Order Traversal

The *post-order traversal* of a binary tree is performed as

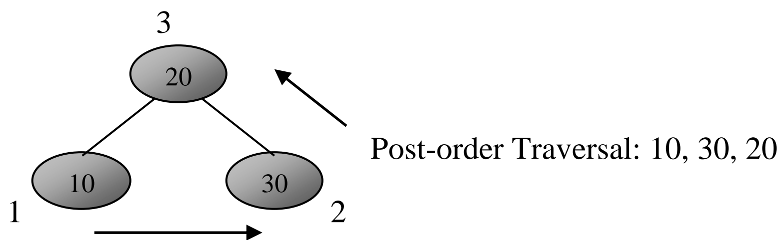
- ❖ Traverse the left sub-tree in pre-order (L)
- ❖ Traverse the right sub-tree in pre-order (R)
- ❖ Visit the root (N)



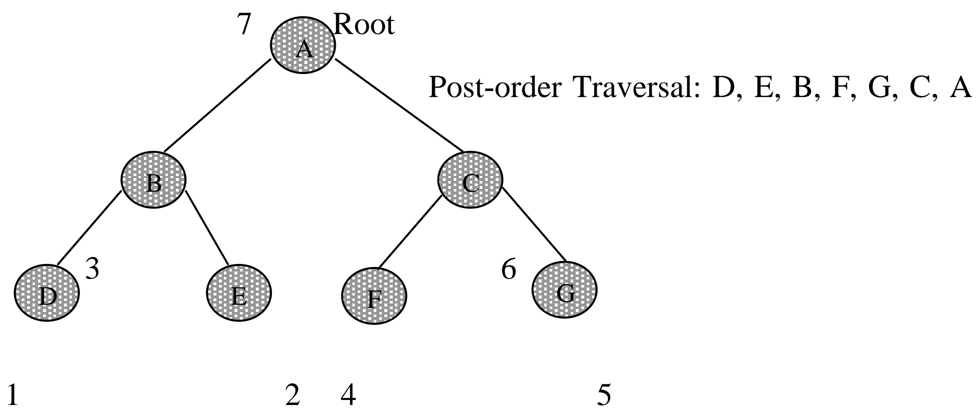
Recursive Routine for post-order Traversal

```
void postorder(struct rec * Tree )
{
    if  (tree != NULL)
    {
        postorder(tree -> left);
        postorder(tree -> right);
        printf("%d\n", tree->value);
    }
}
```

Example 1



Example 2



Example Program: Program for Binary Tree Traversal

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
typedef struct tree *node;
```

```
node insert(int, node T);
void inorder(node T);
void preorder(node T);
void postorder(node T);
struct tree
{
	int data;
	struct tree *right, *left;
}*root;
void main( )
{
	node T = NULL;
	int data, ch, i =0, n;
	clrscr( );
	printf("\nEnter the number of elements in the tree");
	scanf("%d", &n);
	printf("\n The elements are:\n");
	while( i < n )
	{
		scanf("%d",&data);
		T = insert(data, T);
		i++;
	}
	printf("1.Inorder\t. 2.Preorder\t3.Postorder\t4.Exit");
	do
	{
		printf("\nEnter your choice:");
		scanf("%d",&ch);
		switch(ch);
		{
		case 1: printf("Inorder Traversal of the given tree\n");
			inorder(T);
			break;
		case 2: printf("Preorder Traversal of the given tree\n");
```

```
                preorder(T);
                break;
        case 3: printf("Postorder Traversal of the given tree\n");
                postorder(T);
                break;
        default: printf("Exit");
                exit(0);
        }
    }while(ch < 4);
    getch( );
}
node insert(int x, node T)
{
    struct tree *newnode;
    newnode = malloc(sizeof(struct tree));
    if( newnode = = NULL)
        printf(" Out of space\n");
    else
    {
        if ( T = = NULL)
        {
            newnode->data = x;
            newnode->left = NULL;
            newnode->right = NULL;
            T= newnode;
        }
        else
        {
            if (x < T->data)
                T->left = insert(x, T->left);
            else
                T->right = insert(x, T->right);
        }
    }
```

```
        return T;
}
void inorder(node T);
{
        if ( T!= NULL)
        {
                inorder(T->left);
                printf(%d\t", T->data);
                inorder(T->right);
        }
}
void preorder(node T);
{
        if ( T!= NULL)
        {
                printf(%d\t", T->data);
                preorder(T->left);
                preorder(T->right);
        }
}
void postorder(node T);
{
        if ( T!= NULL)
        {
                postorder(T->left);
                postorder(T->right);
                printf(%d\t", T->data);
        }
}
```

Output Of Program:

```
Enter the number of elements in the tree
The elements are:
30
20
```

40

25

1.Inorder 2.Preorder 3.Postorder 4.Exit

Enter your choice:1

Inorder Traversal of the given tree

20 25 30 40

Enter your choice:2

Preorder Traversal of the given tree

30 20 25 40

Enter your choice:3

Postorder Traversal of the given tree

25 20 40 30

Enter your choice:4

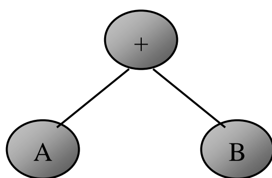
Exit

5.3.4 Application Of Binary Trees:

There are many applications that require traversal of binary trees. For **example**, when a tree is used to represent an arithmetic expression, it needs to be traversed to evaluate that expression. And can also be used in binary search tree. In this, we see about the how to evaluate arithmetic expressions by using binary tree traversal.

Evaluation of arithmetic expressions:

1.Arithmetic Expression: A + B Traversing Expression Trees

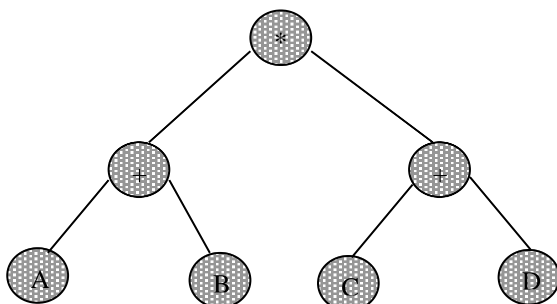


Pre-order : +AB

In-order : A+B

Post-order: AB+

2. (A + B) * (C + D)

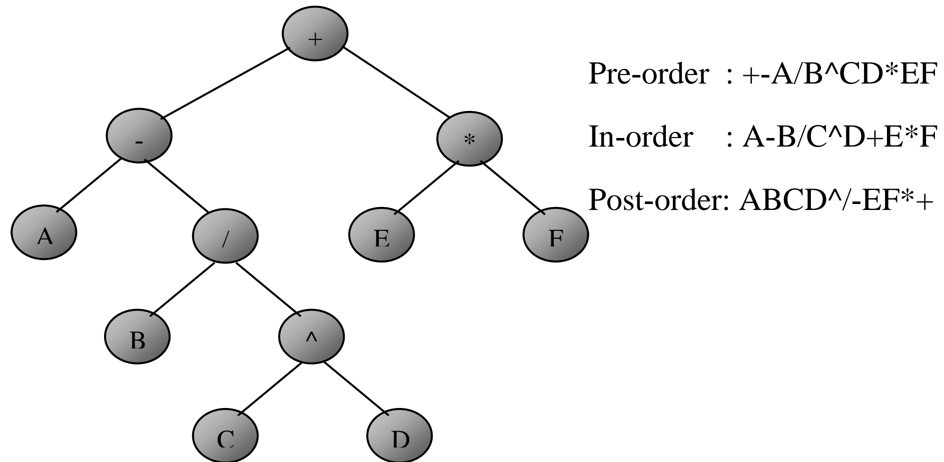


Pre-order : *+AB+CD

In-order :A+B*C+D

Post-order:AB+CD+*

3. A – B / (C ^ D) + (E * F)



Pre-order : +-A/B^CD*EF

In-order : A-B/C^D+E*F

Post-order: ABCD^/-EF*+

5.3.5 Representation Of A Binary Trees In Memory

5.3.5.1 Linked Representation Of Binary Tree

The elements are represented using pointers. Each node in linked representation has *three fields* namely

- Pointer to the left sub-tree – contain address of the left child
- Data field
- Pointer to the right sub-tree – contain address of the right child

Left child	Data	Right child

In leaf nodes, both pointer fields are assigned as NULL. Linked representation of binary tree (in fig 5.3.8) as shown in Fig. 5.3.9.

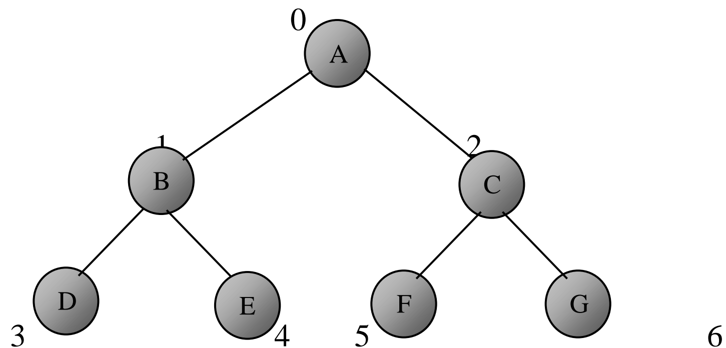


Fig 5.3.8 Binary tree

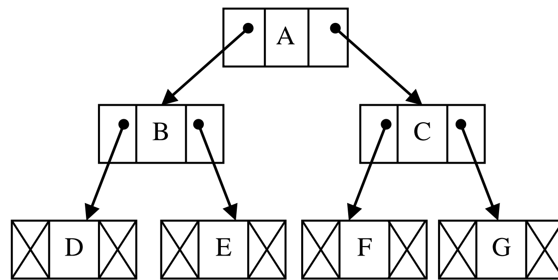


Fig 5.3.9 Linked Representation

Advantages

- Insertions and deletion involve no data movement and no movement of nodes.

Disadvantages

- Difficult to determine the parent node of n
- More memory space is required to store pointers.

5.3.5.2 Array Representation Of Binary Trees

The elements are represented using arrays; this is otherwise called as **linear representation**. For any element is in position i, the left child is in position 2i, the right child is in position (2i + 1) and the parent is in position (i/2). An array representation of binary tree (in Fig 5.3.8) as shown in Fig.5.3.10.

A	B	C	D	E	F	G

Fig 5.3.10 Array or Linear Representation

To identify parent and children, the following phenomenon is followed

For any node n , 0 <= n <= maxsize-1 then we have,

1. Parent(n)

For a node n, where n>0 then

parent(n) = floor((n - 1) / 2)

For **example** in Fig.5.3.8, consider a node D with index 3

Now

Parent(D) = floor((3 – 1) / 2)

= floor(2 / 2)

= floor(1)

= 1

The node index 1 that is B is the parent of D

2.left child(n)

The left child of node numbered as n is at (2n + 1)

For **example**

Left child (A) = left child(0) = 2 x 0 + 1 = 1

that is the node index 1 nothing but B is the left child of A.

3. Right child (n)

The right child of node n is (2n + 2)

For **example**

Right child (A) = Right child(0) = 2 x 0 + 2 = 2

that is the node index 2 nothing but C is the right child of A.

5.3.6 Operations On A Binary Search Tree

5.3.6.1 Introduction

Binary Search Tree

Binary search tree is a special binary tree in which every node x in the tree, the values of all the keys in its left sub-tree are smaller than the key value in x and the values of all keys in its right sub-tree are larger than the key value in x.

Characteristics

- Every node will have a value.
- No two nodes should have same value.
- The value of left sub-tree is less than the value of its parent node.

  value (left) < value (parent)

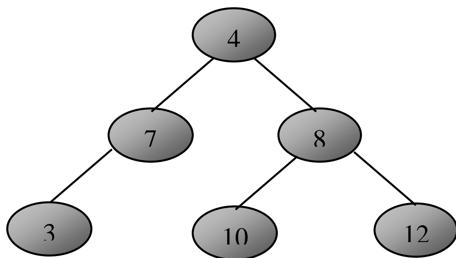
- ❖ The value of right sub-tree is greater than the value of its parent node.

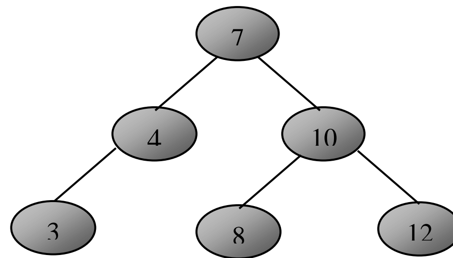
    value (right) < value (parent)

- ❖ Left and right sub-trees are it-self a binary search trees.

Comparison between binary tree and binary search tree

Binary Tree	Binary Search Tree
1.A tree is said to be a binary tree if it has at most two children.	1. A binary search tree is a binary tree in which the key values in the left node is less than the root and the key values in the right node are greater than the root.
2. It doesn't have any order.	2. It should have order.
3. Example	3. Example





Note:

- ❖ Every binary search is a binary tree.
- ❖ All binary trees need not be the binary search trees.

Operation Of Binary Search Tree

The operation of binary search tree are

- ❖ Searching a node
- ❖ Node Insertion
- ❖ Node Deletion

5.3.6.2 Searching Operation

The most important operation on binary search tree is **searching a node**. While searching a node in a binary search tree take node value to the searched start comparison from root node. If root is NULL then terminate the search. If the root is not NULL, it means the tree is not empty so that compare search node value with root node value if it is match then node under search is the root node. If the value is less then root continue search on left sub-tree in similar fashion else value is greater then the root continue search on right sub-tree in the similar fashion

To search the node in the tree

- Check whether the root is NULL

  if root is NULL then

  return NULL
- Other wise, check the value x with the root node value i.e T->element

  If x is equal to T->element, return T

  If x is less than T->element, traverse the left of T recursively

  If x is greater than T->element, traverse the right of T recursively

Routine for Search Operation from the Binary Search Tree

```
int BSTsearch(int x, searchTree T)
{
        if (T = = NULL)
                return NULL;
        else if ( x < T->element)
                BSTsearch(x, T->left);
        else if (x > T->element)
                 BSTsearch(x, T->right);
        else
                return T; // return the position of the search element
}
```

5.3.6.3 Insertion Operation And Deletion Operation

Insertion Operation

While inserting a new node into binary search tree, check whether the tree is empty or not. If tree is empty insert the new node as root node and make

its children as NULL. If the tree is not empty, compare the new node value with the nodes starting from root in order to find the suitable node position by following the binary search tree characteristics.

To insert the element x into the tree

- Check with the root node T
- If it is less than the root

  Traverse the left sub-tree recursively until it reaches

  the T-> left equals to NULL. Then x is placed in T-> left.
- If x is greater than the root

  Traverse the right sub-tree recursively until it reaches

  the T-> right equals to NULL. Then x is placed in T-> right.

Routine to Insert Element into a Binary Search Tree

```
searchTree insert( int x, searchTree T)
{
        if (T = = NULL)
        {
                T= malloc(size_of(struct TreeNode);
                T->element = x;
                T->left = NULL;
                T->right = NULL;
        }
        else
        if ( x < T->element )
                T->left = insert(x, T->left);
        else if(x > T->element )
                T->right = insert(x, T->right);
        else if ( x = = T->element)
        {
                printf("Duplicate node");
                exit(0);
        }
        return T ;
}
```

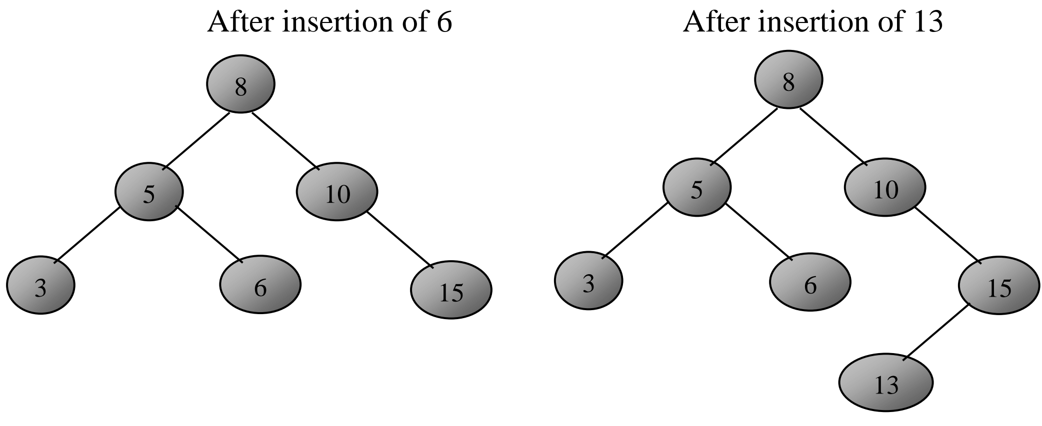
Example:

To insert 8, 5, 10, 15, 3, 6, 13 into binary search tree

Steps	Description	Tree Construction
1.	Insert 8 as root. Tree is empty, . so that 8 is inserted as root node The left and right child are set to NULL	
2.	Insert 5, Assume x = 5 x < root i.e ., 5 < 8 left sub-tree is null. Hence insert 5 as left sub-tree	
3.	Insert 10 , Assume x =10 x > root i.e ., 10 > 8 right sub-tree is NULL Hence insert 10 as right sub-tree	
4.	Insert 15, Assume x =15 x > root i.e., 15 >8 Traverse towards right Right sub-tree is not NULL x > right sub-tree i.e., 15 >10 Hence Insert 15 as right sub-tree of 10	
5.	Insert 3, Assume x = 3 x < root i.e., 3 < 8 Traverse towards left Left sub-tree is not NULL X < left sub-tree i.e., 3 < 5	

Hence insert 3 as left sub tree of 5

6. Similarly the rest of the elements are traversed



Deletion Operation

While deleting a node from a tree, the memory is to be released. Deletion can be at three different places.

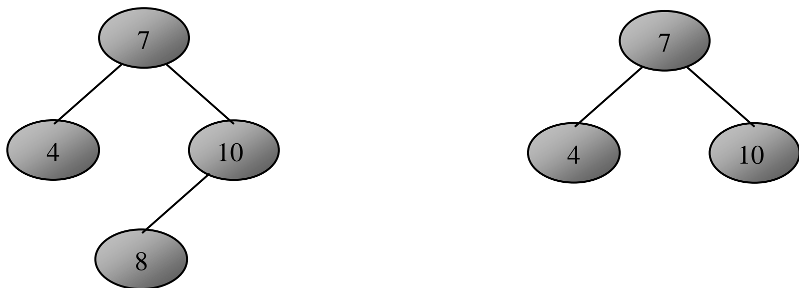
❖ **Deleting a leaf node**

If the node is a leaf node, it can be deleted immediately. It need the following things

- Search the parent of the leaf node.
- Make the parent link to the leaf node as NULL.
- Release the memory from the deleted node.

Example

Deleting the leaf node 8 from a binary search tree, which has no child.



Before deleting the leaf node 8 After deleting the leaf node 8

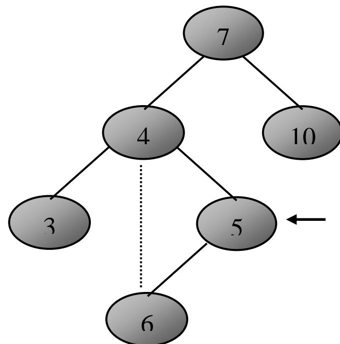
❖ **Deleting the node with one child**

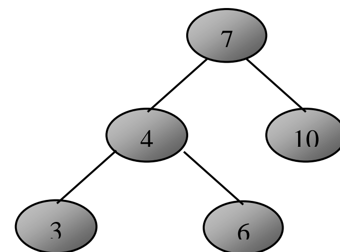
If the node has one child, it can be deleted by adjusting its parent that point to its child node. It need the following things

- ➢ Search the parent of the node to be deleted.
- ➢ Assign the parent link to the child node of the node to be deleted.
- ➢ Release the memory from the deleted node.

Example :

Deleting the node 5 from a binary search tree, which has one child.





Before deleting the node 5 After deleting the node 8

❖ **Deleting the node with two child**

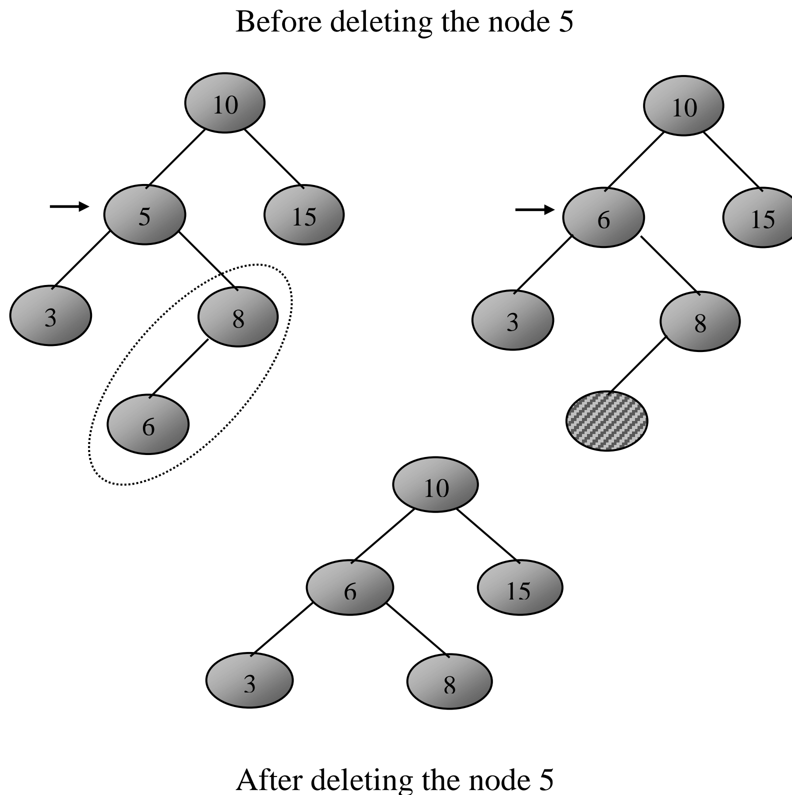
It is difficult to delete a node, which has two children. It need the following things

- ➢ Search the parent of the node to be deleted.
- ➢ Copy the content of the in-order successor to the node to be deleted
- ➢ Delete the in-order successor node. If the in-order successor has only one child, follow the steps for deleting anode with one child.
- ➢ Release the memory from the deleted node.

Example 1

Deleting the node 5 from a binary search tree, which has two children. The minimum element at the right sub-tree of node 5 is node 6. Now the value 6 is replaced in the position of value 5. Since the position of 6 is the leaf node delete immediately.

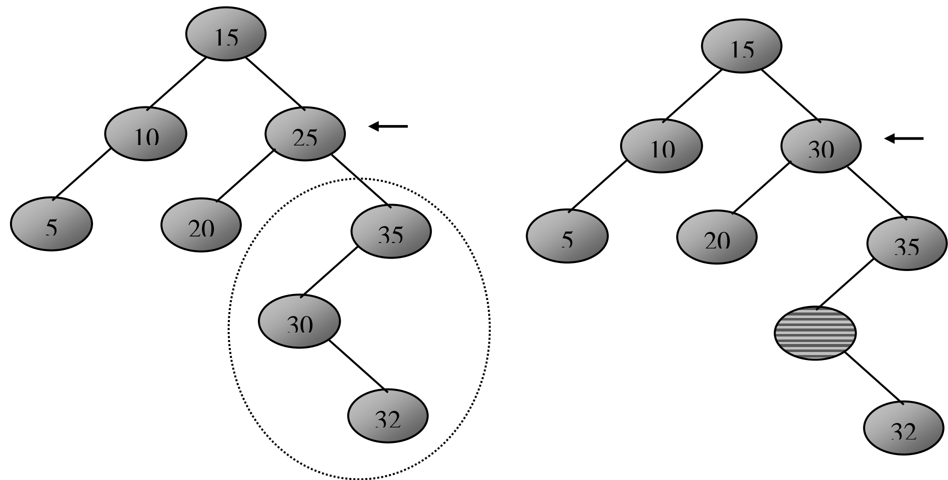
Before deleting the node 5



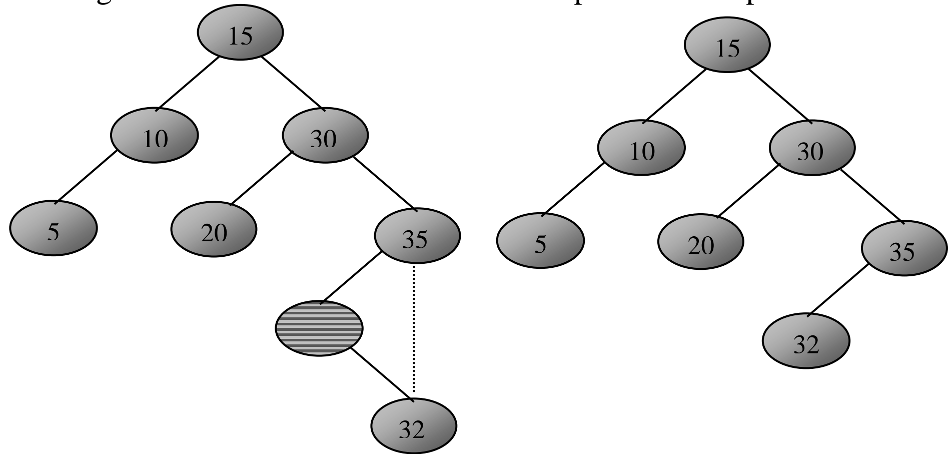
After deleting the node 5

Example 1

Deleting the node 25 from a binary search tree, which has more than one child. The minimum element at the right sub-tree of node 25 is node 30. Now the value 30 is replaced in the position of value 25. Since node 30 has one child, the pointer currently pointing to this node is made to pointer to its child node 32.



Before deleting the node 25 Node 30 is replaced in the position of 25



Since node 30 has one child, the pointer currently

After deleting the node 25

pointing to this node is made to pointer to its child node 32.

Routine for Deleting Node from Binary Search Tree

```
searchTree delete(int x, searchTree T)
{
        int tempnode;
        if( T = = NULL)
                printf("Tree is empty and element  not  found");
        else
        if(x < T->element) // Traverse towards left
```

```
            T->left = delete(x, T->left);
        else
        if(x > T->element) // Traverse towards right
            T->right = delete(x, T->right);
        else if (x = = T->element)
            dealloc(sizeOf(T));
        else    // two children
        if(T-> left && t->right)
        {
            // Replace with smallest element in right sub-tree
            tempnode = searchMin(T->right);
            T->element = tempnode->element;
            t->right = delete(T->element, T->right);
        }
        else    // one or zero child
        {
             tempnode = T;
            if (T->left = = NULL)
                T = T->right;
            else if ( T->right = =  NULL)
                T = T->left;
            Free(tempnode);
        }
        return T;
}
```

5.3.7 Forest Tree

Definition

A forest is a set of n >= 0 disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. For example, in Fig 5.3.11 . if we remove **A** we get a forest with three trees (See Fig. 5.3.12)

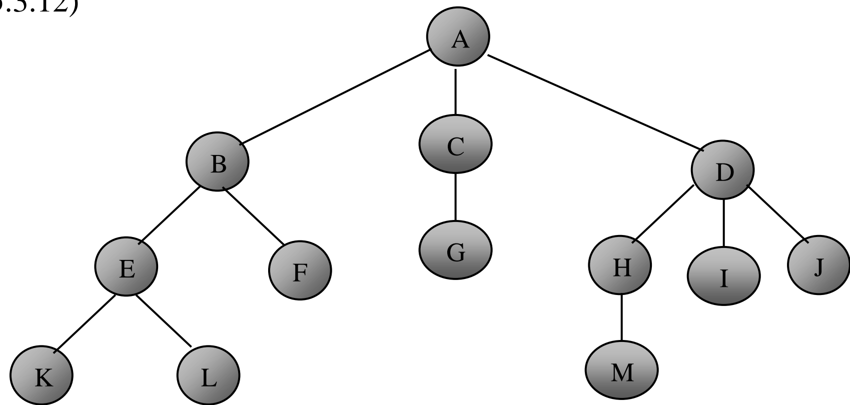


Fig. 5.3.11 A Sample Tree

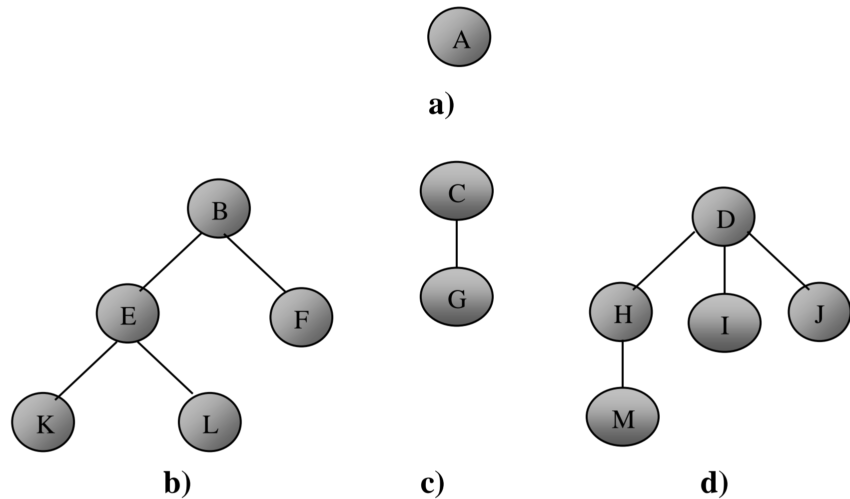


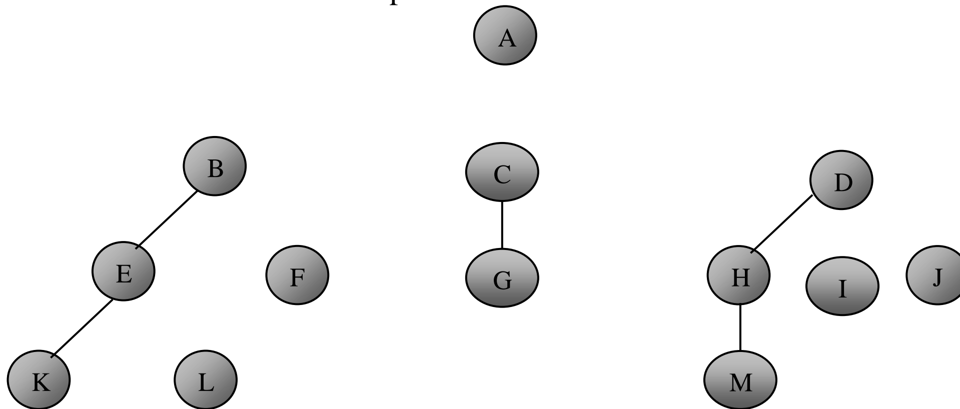
Fig. 5.3.12 Forest Trees for Sample Tree in Fig. 5.3.11

Conversion Of Forest Tree To Binary Tree

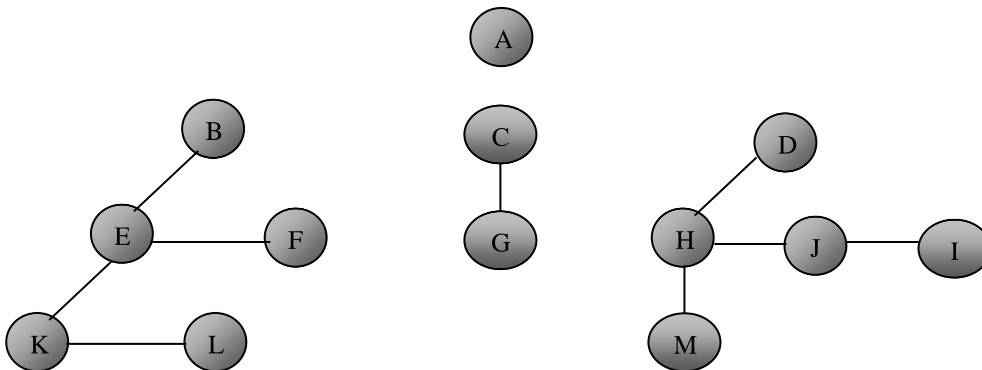
Now we have the forest then these can all be transformed into a single binary tree by first obtaining the binary tree representation of each of the trees in the forest and then linking all the binary trees together through the sibling field of the root nodes. For instance, the forest with the trees in Fig.5.3.12 yields the binary tree representation shown in Fig.5.3.13 .

Following procedure illustrates the conversion from a forest to binary trees.

Step1: Delete all the branches except the left most branch from the tree.

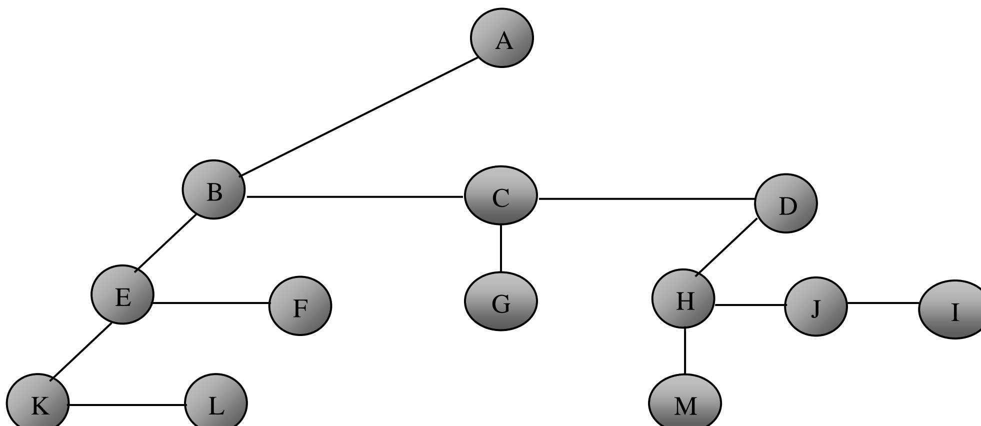


Step 2: Connect the nodes from left to right to make a binary tree.



Step 3:

If two or more trees in a group connect the root node of a second tree to the root node of a first tree as a right child, connect the root node of a third tree to the root node of a second tree as a right child and so on.



We define this information in a formal way as follows. If T1, T2, ... Tn is a forest trees, then binary tree corresponding to this forest, denoted by B(T1, T2, ... Tn):

i) is empty if n = 0

ii) has root equal to root(T1); has left sub tree equal to B(T11, T12, . . . T1m) where T11, T12, ... T1m are the sub trees of root(T1) ; and has right sub tree b(T2……Tn).

Finally we get the actual binary tree representation of forest trees is shown in Fig.5.3.13.

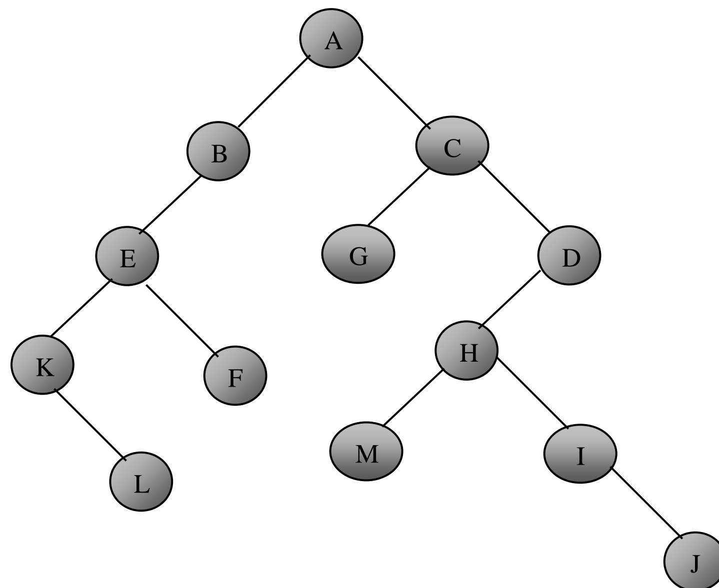


Fig. 5.3.13 Binary Tree for the forest trees in Fig.5.3.12

5.3.8 Self Assessment Questions

Fill in the blank

1. A tree is said to be a binary tree if it has at most _________ children.

2. Remove the root from tree it forms ______________ trees.

True / False

1. A binary tree whose non-leaf nodes have left and the right child is a complete binary tree.

Multiple Choice

1. Traverse from left, root and right is called as

a) Preorder traversal b) Inorder traversal

c) Postorder traversal d) All of the above

Short Answer

1. What is the purpose of tree traversal?

----

2. Define binary search tree.

Graphs

5.3.9 Definition & Terminology

Definition

A **graph G** consists of a set of vertices V and a set of edges (links) E. then G can be written as,

G = (V, E)

where

$V = \{ v_1, v_2, \ldots\ldots\ldots v_n \}$

$E = \{ e_1, e_2, \ldots\ldots\ldots e_n \}$

The graph can be divided into **two types**.

❖ **Directed Graph or Digraph**

Directed graph is a graph which consist of directed edges, where each edge in E is unidirectional. It is also called digraph. If (v, w) is a directed edge then (v, w) # (w, v). (See Fig.5.4.1)

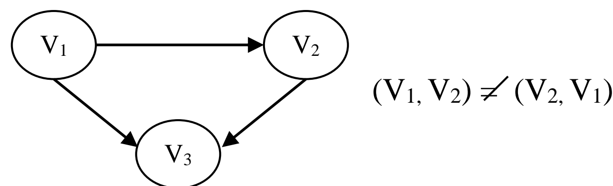


Fig. 5.4.1 Directed Graph

❖ **Undirected Graph**

An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge the (v, w) = (w, v). (See Fig.5.4.2)

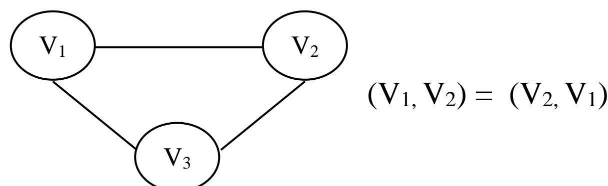
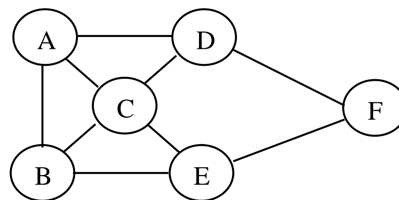


Fig. 5.4.2 Undirected Graph

Terminologies

❖ **Adjacent vertices**

Vertex A is said to be adjacent to vertex B if there is an edge between A and B. Let us consider the graph in Fig.5.4.3.



Here

Adjacent of A = {B, C, D}

Adjacent of B = {A, C, E}

Adjacent of C = {A, B, D, E}

Adjacent of D = {A, C, F}

Adjacent of E = {B, C, F}

Adjacent of F = {D, E}

Fig.5.4.3 Graph G

❖ **Path**

A path from vertex w is a sequence of vertices, each adjacent to the next. Let us consider the graph in Fig.5.4.3 the following is the paths for A to F.

Path 1: A -> B -> E -> F

Path 2: A -> C -> E -> F

Path 3: A -> D -> F

Path 4: A -> C -> D -> F

❖ **Cycle**

A cycle is a path in which first and last vertices are the same. In graph G in Fig.5.4.3 the following is the cycle

Cycle 1: A -> B -> E -> F ->D -> A

Cycle 2: A -> C -> E -> F -> D -> A

Cycle 3: C -> E -> F -> D -> C

Cycle 4: A -> B -> C -> A

❖ **Degree**

The number of edges incident on a vertex determines its degree. The degree of vertex V is written as degree (V). In a directed graph, the degree is categorized into in-degree and out-degree

In-degree (V) = the number of edges coming into that vertex V.

Out- degree (V) = the number of edges going out of that vertex V.

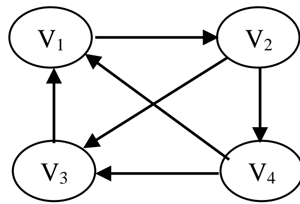


Fig .5.4.4 Directed Graph

In above directed graph in Fig 5.4.4

In-degree of vertex V_1 is 2

Out-degree of vertex V_1 is 1

❖ **Length**

The length of the graph is the number of edges on the path, which is equal to n - 1, where n represents the number of vertices. The length of path A -> B -> E -> F in Fig.5.4.3 is **3** i.e., {(A, B) , (B, E) , (E, F)}.

If there is a path from vertex to **itself**, then the path length is 0.

❖ **Loop**

If the graph contains an edge (v, v) from a vertex to itself then the path is referred to as a loop.

❖ **Simple path**

A simple path is a path such that all vertices on the path, except possibly the first and last are distinct.

❖ **A cyclic graph**

A directed graph, which has no cycles, is called as acyclic graph. It is abbreviated as DAG (Directed Acyclic Graph). Fig. 5.4.5 shows an acyclic graph.

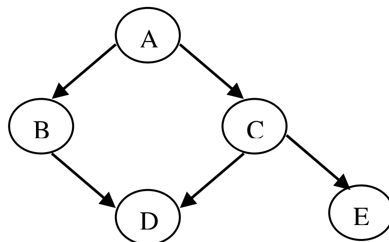


Fig. 5.4.5 Acyclic Graph

❖ **Weighted graph**

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph. (See Fig.5.4.6)

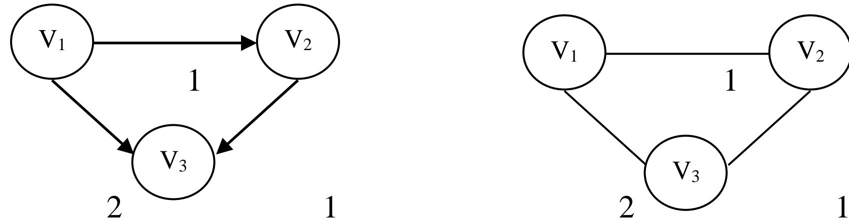


Fig. 5.4.6 Weighted Graph

❖ **Complete graph**

A complete graph is a graph in which there is an edge between every pair of vertices. The complete graph with n vertices will have n (n - 1) / 2 edges. (See Fig .5.4.7)

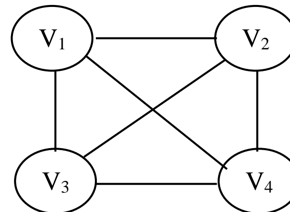


Fig .5.4.7 Complete Graph

Number of vertices n is 4

Number of edges is = 4 (4 – 1) / 2 = (4 * 3) / 2 = 6

i.e., there is a path from every vertex to every other vertex. A complete digraph is a strongly connected graph.

❖ **Strongly connected graph**

If there is a path from every vertex to every other vertex in a directed graph is called strongly connected graph (See Fig.5.4.8). Otherwise it is said to be weakly connected graph (See Fig. 5.4.9).

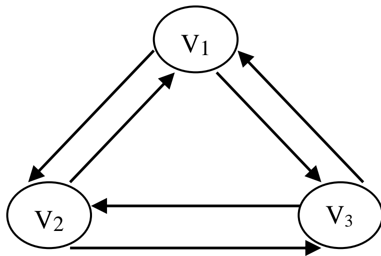


Fig. 5.4.8 Strongly Connected Graph

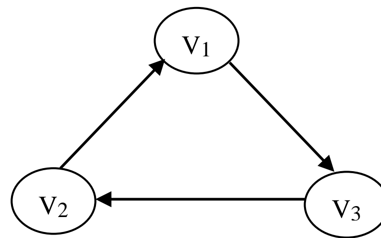


Fig. 5.4.9 Weakly Connected Graph

5.3.10 Graph Representations

A graph can be represented by

- Adjacency Matrix (Using Array)
- Adjacency List (Using Pointers)

Adjacency Matrix Representation

One simple way to represents a graph is to use a two dimensional array. This is known as adjacency matrix representation.

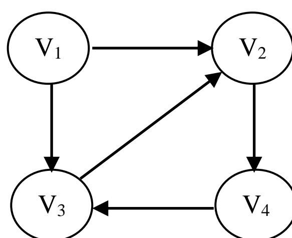
The adjacency matrix A for a graph G = (V, E) with n vertices is an n x n matrix, such that

Aij= 1, if there is an edge Vi to Vj

Aij = 0, if there is no edge.

For **example** Fig.5.4.10 shows the directed graph and corresponding adjacency matrix representation, and Fig.5.4.11 shows the undirected graph and corresponding adjacency matrix representation.

Adjacency Matrix Representation for Directed Graph



ij	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	0	1
V_3	0	1	0	0
V_4	0	0	1	0

Fig .5.4.10 Directed Graph & Adjacency Matrix A for directed Graph G

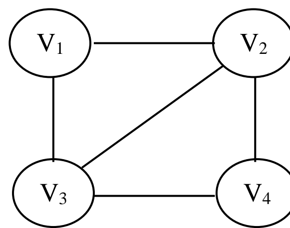
For **example**

$V_1,V_2 = 1$ since there is an edge V_1 to V_2

$V_1,V_3 = 1$, there is an edge V_1 to V_3

V_1,V_1 & $V_1,V_4 = 0$, there is no edge

Adjacency Matrix Representation for Undirected Graph



ij	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	1	0	1	1
V_3	1	1	0	1
V_4	0	1	1	0

Fig .5.4.11 Undirected Graph & Adjacency Matrix A for Undirected Graph G

Advantage

- ❖ Simple to implement.

Disadvantage

- ❖ Takes $O(n^2)$ space to represents the graph.
- ❖ It takes $O(n^2)$ time to solve the most of the problem.

Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices. The rows of the adjacency matrix are represented as n linked lists. There is one list of each vertex in the graph. The nodes in list i represent the vertices that are adjacent from vertex i. Each list has a head node. The head nodes are sequential providing easy random access to the adjacency list for any particular vertex.

Adjacency List Representation for Directed Graph

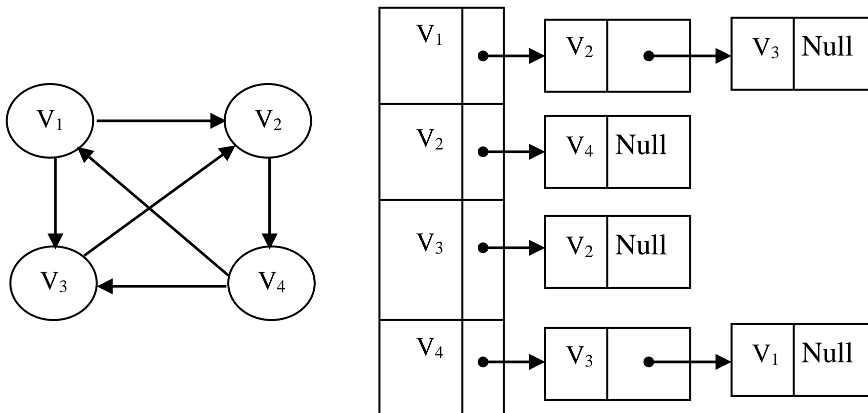


Fig .5.4.12 Directed Graph & Adjacency List for directed Graph G

Adjacency List Representation for Undirected Graph

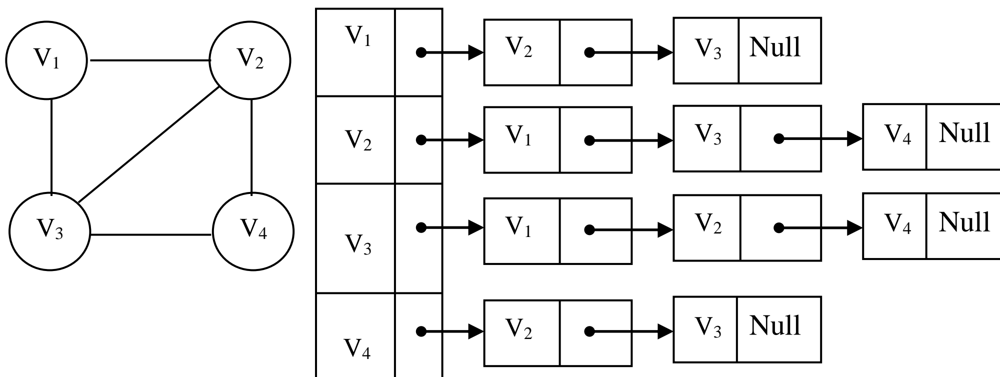


Fig .5.4.13 Undirected Graph & Adjacency List for Undirected Graph G

Disadvantage

It takes O(n) time to determine whether there is an arc from vertex I to vertex j. since there can O(n) vertices on the adjacency list for vertex i.

5.3.11 Graph Traversals

5.3.11.1 Introduction

Traversal means visiting all nodes exactly once. There are two ways to traverse a graph.

- Depth First Traversal or Depth First Search
- Breadth First Traversal or Breadth First Search

5.3.11.2 Depth First Search

The algorithm for depth first search of an undirected graph is as follows:

- Depth first works by selecting one vertex v of G as a start vertex; v is marked as visited.
- Select an unvisited vertex w adjacent to v.
- Repeat steps 1 and 2 till all adjacent vertices of w are visited.
- On reaching a vertex whose all-adjacent vertices have been visited go back to the last vertex visited which has an unvisited vertex adjacent to it and go back to step 1.
- Terminate the search when no unvisited vertex can be reached from any of the visited ones.

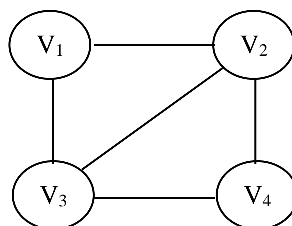
Routine for Depth First Traversal

```
/*  give an undirected graph G = <V,E> with n vertices and an array
reachable from v. G and visited are global */
    void DFT( vertex v)
    {
            visited[v] = True;
            for each w adjacent to v
            if (visited[w] = false)
                    DFT(vertex v);
    }
```

Illustration of Depth First Search or Traversal

Illustration 1:

The undirected graph in Fig.5.4.14 (a) is represented by its adjacent matrix shown in Fig.5.4.14 (b) The process of depth first search on that graph is described in stages is given below. Related Depth First Spanning Tree is shown in Fig.5.4.15.



ij	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	1	0	1	1
V_3	1	1	0	1
V_4	0	1	1	0

(a) (b)

Fig .5.4.14 Undirected Graph & Adjacency Matrix A for Undirected Graph G

Implementation:

1. Let V_1 be the source vertex. Mark it to be visited.

2. Find the immediate adjacent unvisited vertex V_2. Mark it to be visited.

3. From V_2 the next unvisited adjacent vertex is V_3 mark it to be visited.

4. From V_3 the next unvisited adjacent vertex is V_4 mark it to be visited.

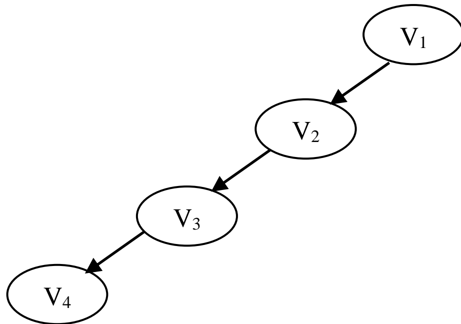
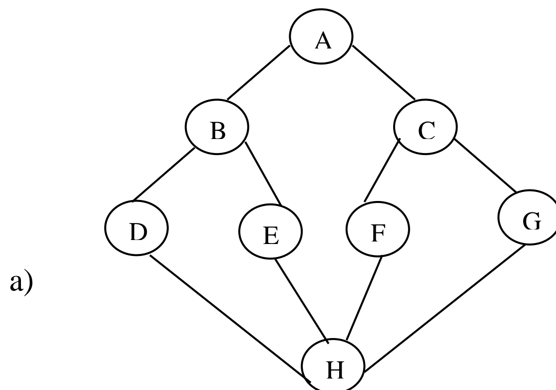


Fig.5.4.15 Depth First Spanning Tree

Illustration 2:

The undirected graph in Fig.5.4.16 (a) is represented by its adjacent matrix shown in Fig.5.4.16(b) The process of depth first search on that graph is described in stages is given below. First initialize all vertices to zero.

a)



b)

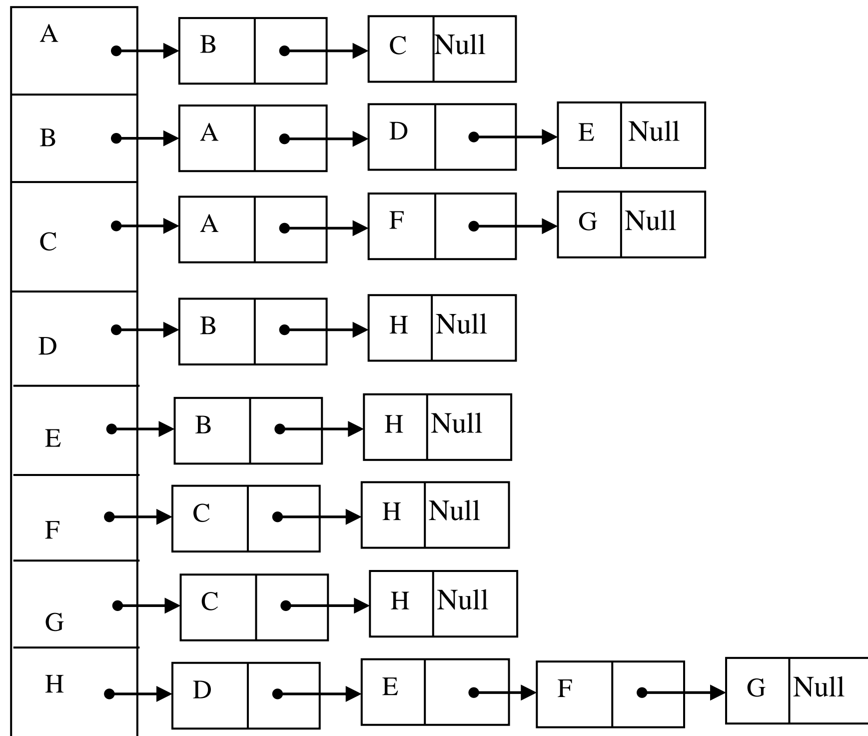


Fig .5.4.16 Undirected Graph & Adjacency List L for Undirected Graph G

Visited(A) = 0 Visited(B) = 0 Visited(C) = 0 Visited(D) = 0

Visited(E) = 0 Visited(F) = 0 Visited(G) = 0 Visited(H) = 0

If a depth first search is initiated from vertex A then the vertices of G are visited in the order A, B, D, H, E, F, C, G. We may easily verify that DFT(A) visits all vertices connected to A.

Status		Adjacent Vertices			
Visited(A)	= 1	B	C	-	-
Visited(B)	= 1	A (visited)	D	E	-
Visited(D)	= 1	B (visited)	H		
Visited(H)	= 1	D (visited)	E	F	G
Visited(E)	= 1	B (visited)	H (visited)	-	-
Visited(F)	= 1	C	H	-	-
Visited(C)	= 1	A (visited)	F (visited)	G	-
Visited(G)	= 1	C (visited)	H (visited)	-	-

Vertex A, then the visited vertices are A, B, D, H, E, F, C, G.

5.3.11.3 Breadth First Search

Starting vertex V and marking as visited, breadth first search differs from depth first search in that all visited vertices adjacent to V are visited next, then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex A, then vertex B, C next vertices D, E, F and G and finally H.

Routine for Breadth First Traversal

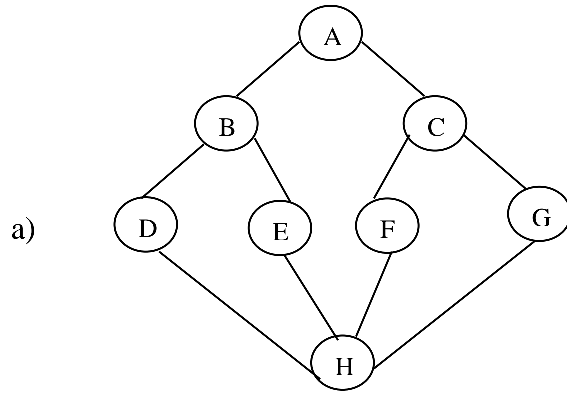
```
/* A breadth first search of G is carried out beginning at vertex V. All
vertices are marked as visited(V) = 1. The graph G and array visited are
global and visited is initialized to zero */

void BFT( vertex v)
{
        visited[v] = True;
        Initialize Q to be empty  /* Q is a queue */
        for each w adjacent to v
        {
        if (visited[w] = false) then  /* add w to the queue */
        {
                call addq(w, Q);
                visited(w) = True;
        }
        if Q is empty then return
                call deleteq(v, Q);
        }
}
```

Illustration of Breadth First Search or Traversal

The undirected graph in Fig.5.4.17(a) is represented by its adjacent List shown in Fig.5.4.17(b) The process of breadth first search on that graph is described in stages is given below.

a)



b)

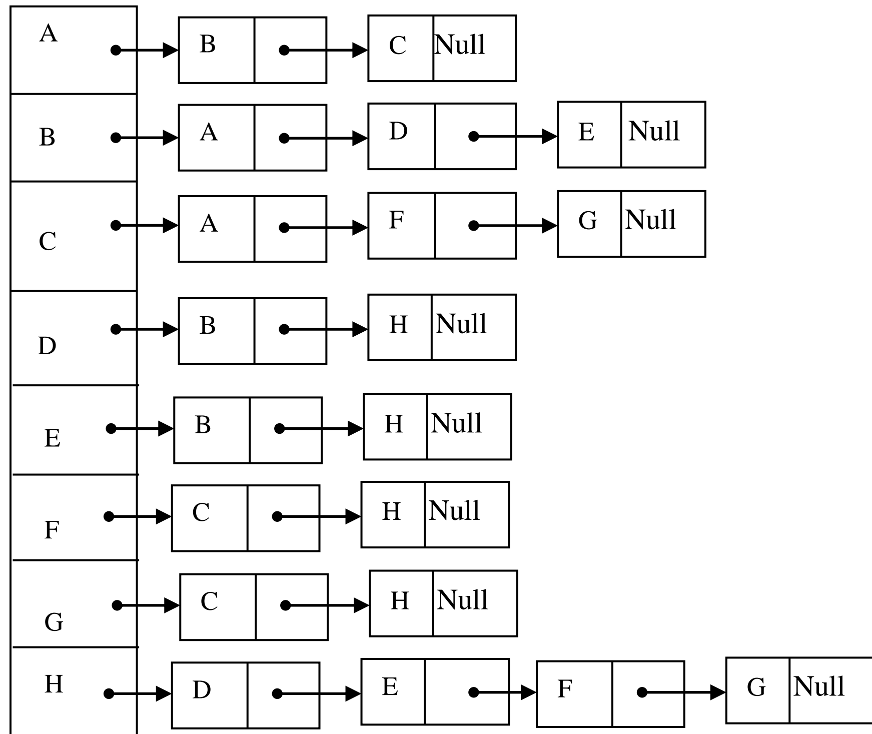


Fig .5.4.17 Undirected Graph & Adjacency List L for Undirected Graph G

Process of Breadth First Search

Visited(A) = 0 Visited(B) = 0 Visited(C) = 0 Visited(D)=0 Visited(E) = 0

Visited(F) = 0 Visited(G) = 0 Visited(H) = 0

Queue

Initialize Queue as empty

Visited(A) = 1 B C

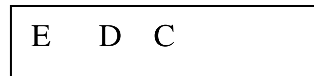
Visited(B) = 1 add B to Queue

Visited(C) = 1 Add C to Queue

Delete B from Queue

Visited(B) = 1 A D E

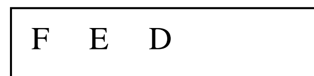
Visited(D) = 1 Add D to Queue



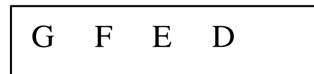
Visited(E) = 1 Add E to Queue

Delete C from Queue

Visited(C) = 1 A F G



Visited(F) = 1 Add F to Queue



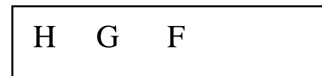
Visited(G) = 1 Add G to Queue

Delete D from Queue

Visited(D) = 1 B H

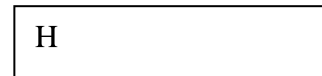
Visited(H) = 1 Add H to Queue

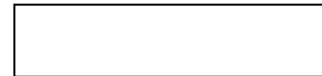
Delete E from Queue



Delete F from Queue

Delete G from Queue





Delete H from Queue

Empty Queue

If a breadth first search is initiated from vertex A then the vertices of graph G are visited in the order A, B, C, D, E, F, G, H. We may easily verify that BFT(A) visits all vertices connected to A.

5.3.12 Shortest Path Algorithm (Using Dijikstra's Algorithm)

The shortest vertex path algorithm determines the minimum cost of the path from source to every other.

The cost of the path $V_1, V_2,$ ------ V_N is $\sum_{I=1}^{N-1} C_{i,\ i+1.}$ This is referred as weighted path length.

The unweighted path length is merely the number of the edges on the path, namely N – 1.

Two types of shortest path problems, exist namely,

- ❖ The single source shortest path problem
- ❖ The all pairs shortest path problem

The single source shortest path algorithm finds the minimum cost from single source vertex to all other vertices. Dijkstra's algorithm is used to solve this problem, which follows the greedy technique. This is applied to weighted graph.

All pairs shortest path problem finds the shortest distance from each vertex to all other vertices. To solve this problem dynamic programming technique known as Floyd's algorithm is used.

In this topic we discuss about only a single source shortest path using Dijkstra's algorithm.

Dijkstra's algorithm

The **single source shortest path** algorithm finds the minimum cost from single source vertex to all other vertices. **Dijkstra's algorithm** is used to solve this problem, which follows the **greedy technique**. This is applied to **weighted graph**.

This algorithm proceeds in stages, at each stage it selects a vertex v, which has the smallest dv among all the unknown (unvisited) vertices, and declares that as the shortest path from S to V and mark it to be known.

We should set dw = dv + Cvw, if the new value for dw would be an improvement.

Where C v w is the cost of minimum shortest path between the vertices v and w.

For a graph G(V, E), Dijkstra's algorithm keeps two sets of vertices.

S = {the set of vertices whose shortest path from the source have already determined}

U = V - S = {the set of remaining undetermined vertices}

The other data structures used for this algorithm are

D = array of distance estimates of shortest path to each vertex

Pi = array of predecessors for each vertex

Routine for Dijkstra's Algorithm

```
void dijkstra(Graph G, Table T)
{
    int i;
    vertex v, w;
    Read Graph(G, T); // Read graph from adjacency list
    //Table Initialization
    for( i= 0; i < Numvertex; i++)
    {
            T[i].known = False;
            T[i].Dist = Infinity;
            T[i].path = NotA vertex;
    }
    T[start].dist = 0;
    for( ; ;)
    {
            V = Smallest unknown distance vertex;
            if( V= = NotA vertex)
                    break;
            T[V] .known = True;
            for each w adjacent to v
                if ( !T[w].known)
                {
                    T[w].Dist = Min( T[w].Dist, T[v].Dist + C v w )
                    T[w].path = v;
                }
    }
}
```

Illustration of Dijkstra's Algorithm

Consider a graph shown in Fig.5.4.18. Assume that the source node is V_1 and fix it as start. The following steps describe how the shortest path between the source and any other node in graph G are identified.

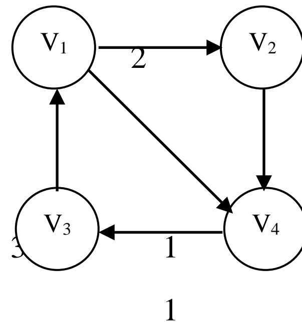
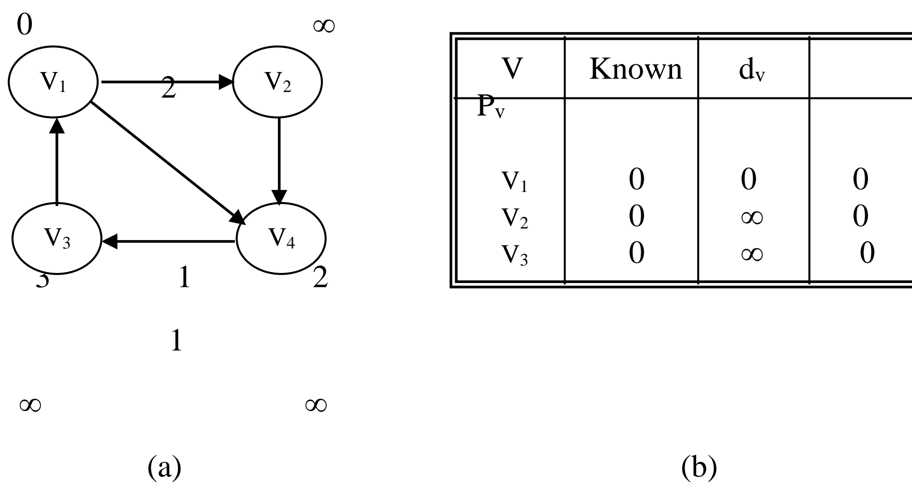


Fig 5.4.18 The directed graph G



V	Known	d_v	P_v
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0

(b)

Fig. 5.4.19 Initial Graph & Initial table configuration of graph G

Vertex V_1 is choose as source and is declared as known vertex. Then the adjacent vertices of V_1 are found and its distance are updated as follows.

$T[V_2].Dist = Min[\, T[V_2].Dist ,\, T[V_1].Dist + C_{V1, V2}\,]$

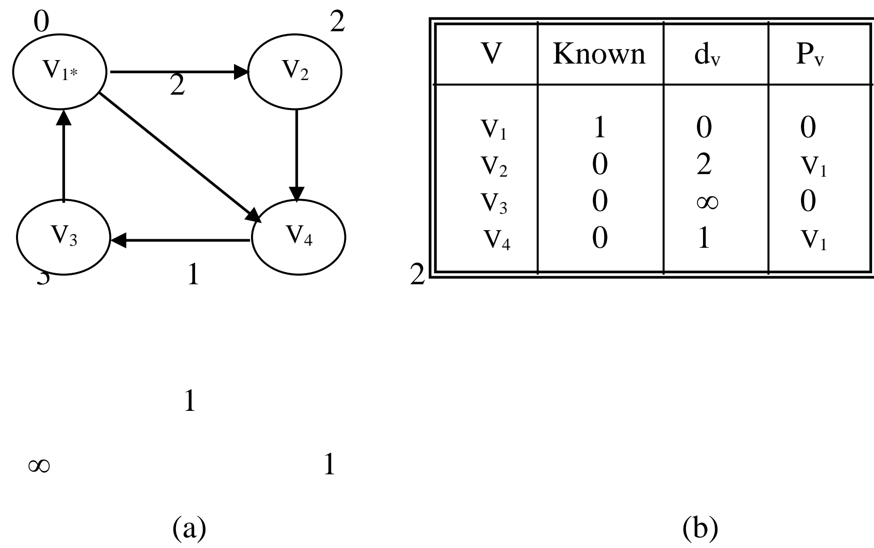
$= Min[\, \infty ,\, 0 + 2]$

$= 2$

$T[V_4].Dist = Min[\, T[V_4].Dist ,\, T[V_1].Dist + C_{V1, V4}\,]$

$= Min[\, \infty ,\, 0 + 1]$

$= 1$



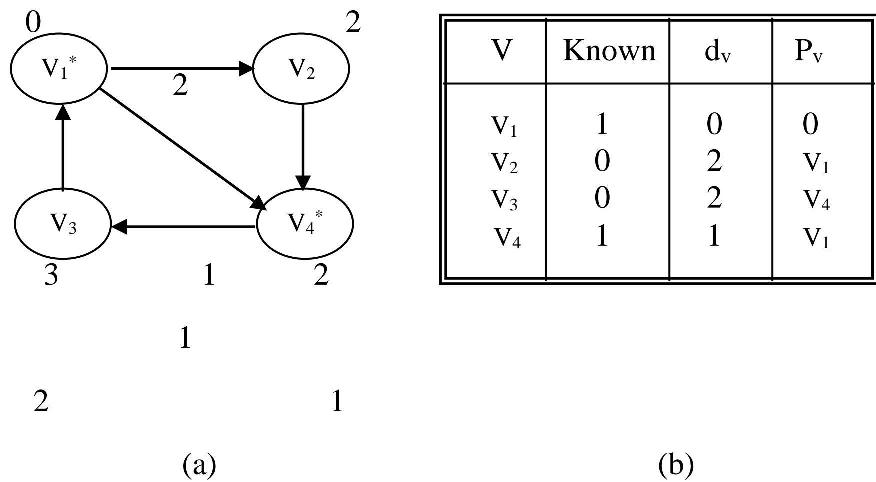
V	Known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	∞	0
V_4	0	1	V_1

(a) (b)

Fig. 5.4.20 After V_1 is declared known

Now $S = \{ V_1 \}$ and $U = \{V_2, V_3, V_4\}$

$$
\begin{aligned}
T[V_3].Dist &= \text{Min}[\, T[V_3].Dist\,,\ T[V_4].Dist + C_{V4,V3}\,] \\
&= \text{Min}[\, \infty\,,\ 1 + 1\,] \\
&= 2
\end{aligned}
$$



V	Known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1

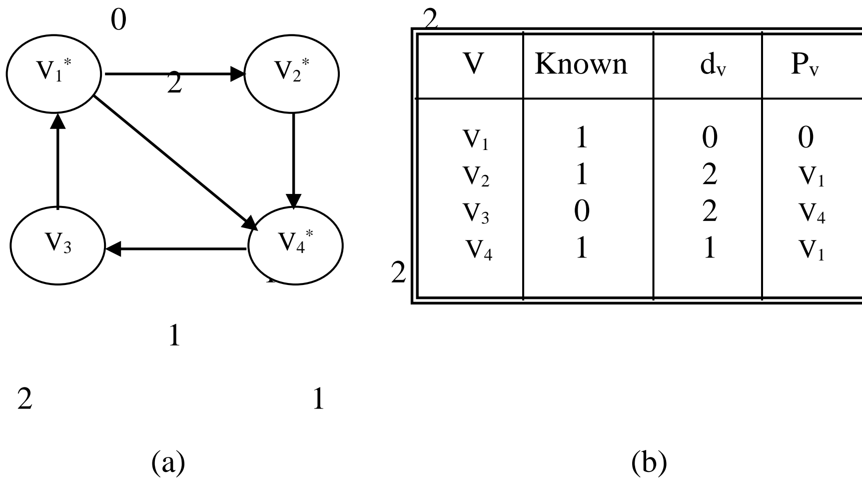
(a) (b)

Fig. 5.4.21. After V_4 is declared known

Now $S = \{ V_1 , V_4\}$ and $U = \{V_2 , V_3 \}$

The next minimum vertex is V_4 and mark it as visited.

Since the adjacent vertex V_4 is already visited, select next minimum vertex V_2 and mark it as visited.

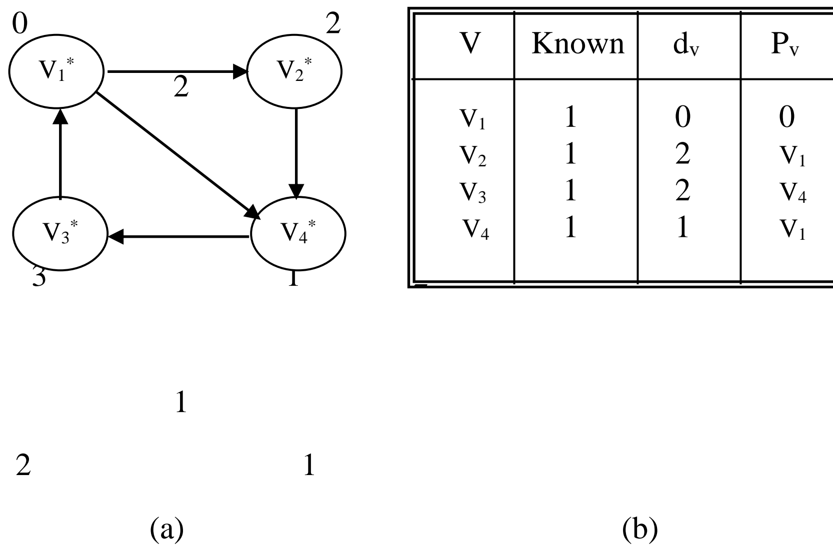


V	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	2	V_4
V_4	1	1	V_1

(a)   (b)

Fig. 5.4.22. After V_2 is declared known

Now $S = \{ V_1 , V_4, V_2 \}$ and $U = \{ V_3 \}$



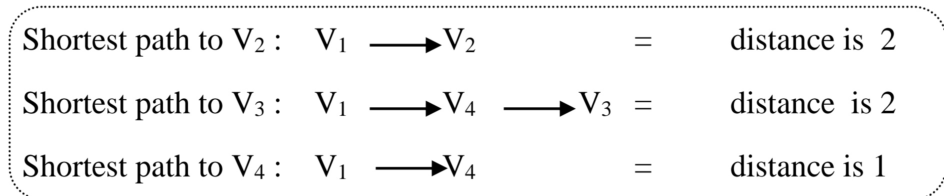
V	Known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	2	V_4
V_4	1	1	V_1

(a)   (b)

Fig. 5.4.23. After V_3 is declared known

Now $S = \{ V_1 , V_4, V_2 , V_3\}$ and $U = \{ \}$

After V_3 is declared known and algorithm terminated, because shortest distance from the source node (root) to all vertices are found. The final table shown in fig5.4. b) states that starting from the source vertex V_1, the following paths are known to be shortest to all other vertices.



This algorithm always works as long as no edge has a negative cost. If any edge has negative cost, this algorithm could not produce right answer. The running time of this algorithm depends on how the table is manipulated.

5.3.13 Self Assessment Questions

Fill in the blank

1. A directed graph, which has no cycles, is called as _______________.

2. Children of the same parents are called ______________

True / False

1. Dijkstra's algorithm is used to find the shortest path of the graph.

Multiple Choice

1. The number of edges coming into the vertex V is called as

a) In-degree b) Out-degree

c) Both a) and b) d) None of the above.

Short Answer

1. Define Shortest path algorithm.

5.4 Summary

In this unit we have introduced some of the most fundamental terminologies of trees and graphs.

The first lesson of this unit ,you have discussed about how to make the binary trees from normal trees. You have learnt how the various operation performed on the trees. You have also learnt about tree traversal and representation trees in memory. Finally you have learnt bout concept of forest tree, conversion of forest trees into binary tree and its related algorithms.

The second lesson of this unit, you have learnt the various terminologies of graphs. You have learnt how the various operation performed on the graphs. You have also learnt about graph traversal and representation graphs in memory. Finally you have seen the shortest path algorithm.

5.5 Unit questions

1. Define binary trees. Discuss various types of binary trees with example.
2. Explain binary search tree with algorithm.
3. Write a algorithm for various binary tree traversal. Explain.
4. How the binary tree represented in memory ? Explain.
5. Define Forest tree. Discuss conversion of forest tree to binary tree with example.
6. Define graph. Discuss terminologies of graphs.
7. How graph can be represented in memory? Explain.
8. Write a algorithm for depth first traversal. Discuss its with illustration.
9. Write a algorithm for breadth first search. Discuss its with illustration.
10. Write Dijikstra's Algorithm for finding shortest path. Explain with illustration.

5.6 Answers for Self Assessment Questions

Answer 5.3.8

Fill in the blank

1. two children 2. Forest tree

True / False

1. True

Multiple Choice

2. b)

Short Answer

1. The traversal of a binary tree involves visiting each node in the tree exactly once.

2.Binary search tree is a special binary tree in which every node x in the tree, the values of all the keys in its left sub-tree are smaller than the key value in x and the values of all keys in its right sub-tree are larger than the key value in x.

Answer 5.4.5

Fill in the blank

1. acyclic graph 2. siblings

True / False

1. True

Multiple Choice

1. a)

Short Answer

1. The shortest vertex path algorithm determines the minimum cost of the path from source to every other.

NOTES

NOTES