VAAGDEVI INSTITUTE OF TECHNOLOGY AND SCIENCE

Peddasettiplli (V), Proddatur-516360

Year/Semester/Branch: I B.Tech I Sem (CSE)

Name of the Subject : C-Programming & Data Structures (20A05201T)

Lecture Notes, Short Answer Questions, Multiple Choice Questions & Long Answer Questions

Prepared by,

G. B. Hima Bindu, B. Tech, M. Tech.,(Ph.D),

Assistant Professor,

Dept. of CSE, VITS, Proddatur.

LECTURE NOTES (2 MARKS)

INDEX

SHORT ANSWER QUESTIONS (2 MARKS)

INDEX

MULTIPLE CHOICE QUESTIONS

INDEX

LONG ANSWER QUESTIONS (10M)

INDEX

PREVIOUS

UNIVERSITY

EXAMINATIONS

QUESTION

PAPERS

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B.Tech -CSE – I Sem L T P C

3 0 0 3

(20A05201T) C-PROGRAMMING & DATA STRUCTURES

(Common to All Branches of Engineering)

UNIT-1

Introduction to C Language - C language elements, variable declarations and data types, operators and expressions, decision statements - If and switch statements, loop control statements - while, for, do-while statements, arrays.

UNIT – 2

Functions, types of functions, Recursion and argument passing, pointers, storage allocation, pointers to functions, expressions involving pointers, Storage classes – auto, register, static, extern, Structures, Unions, Strings, string handling functions, and Command line arguments.

UNIT-3

Data Structures, Overview of data structures, stacks and queues, representation of a stack, stack related terms, operations on a stack, implementation of a stack, evaluation of arithmetic expressions, infix, prefix, and postfix notations, evaluation of postfix expression, conversion of expression from infix to postfix, recursion, queues - various positions of queue, representation of queue, insertion, deletion, searching operations.

UNIT – 4

Linked Lists – Singly linked list, dynamically linked stacks and queues, polynomials using singly linked lists, using circularly linked lists, insertion, deletion and searching operations, doubly linked lists and its operations, circular linked lists and its operations.

UNIT-5

Trees - Tree terminology, representation, Binary trees, representation, binary tree traversals. binary tree operations, Graphs - graph terminology, graph representation, elementary graph operations, Breadth First Search (BFS) and Depth First Search (DFS), connected components, spanning trees. Searching and Sorting – sequential search, binary search, exchange (bubble) sort, selection sort, insertion sort.

Text Books:

1. The C Programming Language, Brian W Kernighan and Dennis M Ritchie, Second Edition, Prentice Hall Publication.

2. Fundamentals of Data Structures in C, Ellis Horowitz, SartajSahni, Susan Anderson-Freed, Computer Science Press.

3. Programming in C and Data Structures, J.R.Hanly, Ashok N. Kamthane and A. AnandaRao, Pearson Education.

4. B.A. Forouzon and R.F. Gilberg, “COMPUTER SCIENCE: A Structured Programming Approach Using C”, Third edition, CENGAGE Learning, 2016.
5. Richard F. Gilberg & Behrouz A. Forouzan, “Data Structures: A Pseudocode Approach with C”, Second Edition, CENGAGE Learning, 2011.

Reference Books:

1. Pradip Dey and Manas Ghosh, Programming in C, Oxford University Press, 2nd Edition 2011.
2. E. Balaguruswamy, “C and Data Structures”, 4th Edition, Tata Mc Graw Hill.
3. A.K. Sharma, Computer Fundamentals and Programming in C, 2nd Edition, University Press.
4. M.T. Somashekara, “Problem Solving Using C”, PHI, 2nd Edition 2009.

Course Outcomes:

1. Analyse the basic concepts of C Programming language. (L4)
2. Design applications in C, using functions, arrays, pointers and structures. (L6)
3. Apply the concepts of Stacks and Queues in solving the problems. (L3)
4. Explore various operations on Linked lists. (L5)
5. Demonstrate various tree traversals and graph traversal techniques. (L2)
6. Design searching and sorting methods (L3)

UNIT-1

Introduction to C Language - C language elements, variable declarations and data types, operators and expressions, decision statements - If and switch statements, loop control statements - while, for, do-while statements, arrays.

1. Introduction to C Language:

Overview of C:

C language is one of the most popular computer languages today because it is structured high level, machine independent language.

History of C:

- C was evolved from ALGOL, BCPL(Basic combined programming language), B, and C by Dennis Ritchie at the Bell laboratories in 1972.
- It uses many concepts from these languages and added the concept of data types and other powerful features
- The history and development of C is illustrated in fig

1960 -> ALGOL -> International Group
1967 -> BCPL -> Martin Richards
1970 -> B -> Ken Thompson
1972 -> C ->Dennis Ritchie

Fig: History of C

Compiler: Compiler converts the high level language to machine understandable language (i.e., low level language).

Importance of C:

- It is a robust language whose rich set of built – in functions and operators can be used to write any complex problem
- The C compiler combines the capabilities of assembly language with the features of high level language and therefore it is well suited for writing both system software and business packages
- Programs written in C are efficient and fast due to its variety of data types and powerful operators
- There are only 32 keywords and its strength lies in its built in functions
- C is highly portable. This means that C program written for one computer can be run on another with little or no modification
- C is well suited for structured programming.
- Important feature of C is its ability to extend itself

A Simple C Program:

/*simple program to print hello message*/

```
#include<stdio.h>
#include<conioh>
void main()
{
clrscr();
printf("Hello! Welcome to C Programming\n");
getch();
}
```

1.1.C Language Elements -C Tokens:

- Individual words and punctuation marks are called tokens
- In C programs the smallest individual units are known as C tokens
- C has six types of tokens as shown in fig.

- C programs are written using these tokens and the syntax of the language

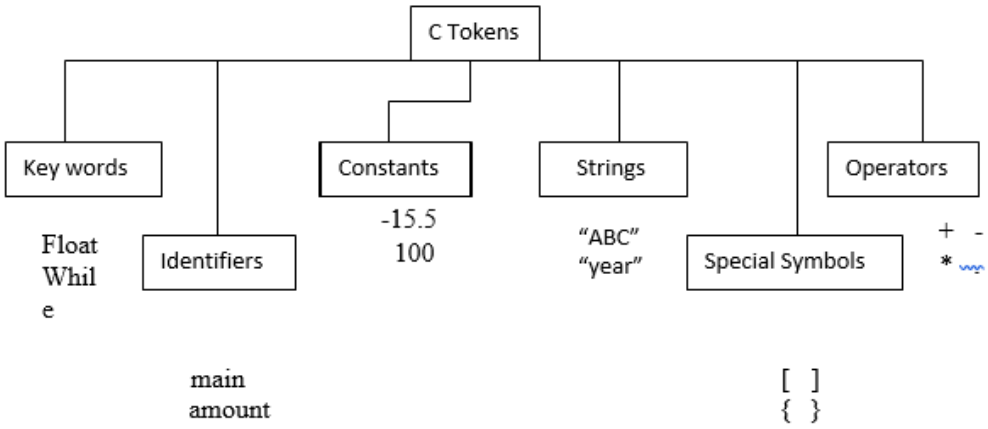


Fig. C Tokens and Examples

1.2. **1.2.1. Variable:** A variable is a data name that may be used to store a data value. A variable may take different values at different execution.

Declaration of variables:

Declaration does two things:

1) Tells the compiler what the variable name is
2) Specifies what type of data the variable will hold Primary type declaration:

datatype v1,v2,v3,… .... ,vn;

- Variables are separated by commas
- A declaration statement must end with a semicolon
- Ex: int count;
  int number, total;
  double ratio;

The conditions to specify variable are:

1. They must begin with a letter or underscore
2. The length should not be more than eight characters
3. Upper case and lower case letters are significant variable Total is not the same as total or TOTAL
4. It should not be a keywordWhite space is not allowed

Ex: John _value mark sum1 x1 are valid

1.2.1.1. (area) % 25^{th} are not valid

Identifiers and keywords

1. Every C word is classified either as a keyword or an identifier
2. All keywords have fixed meanings and these meaning cannot be changed
3. Keywords are basic building blocks for program statements
4. The list of all keywords are listed in table
5. All keywords must be written in lower case

Table: Keywords

Auto	Double	Int	Struct
Break	Else	long	Switch
Case	Enum	register	Typedef
Char	Extern	return	Union
Const	Float	short	unsigned
Continue	For	signed	Void
Default	Goto	sizeof	Volatile
Do	If	static	While

Identifiers refer to the names of variables, functions and arrays

These are user defined names and consist of a sequence of letters and digits, with a letter as a first character.

Both upper case and lower case letters are permitted

Lower case letters are commonly used

The underscore character is also permitted in identifiers usually used as a link between two words in long integers

Rules of identifiers:

1. First character must be an alphabet (or underscore)

2. Must consist of only letters, digits or underscore
3. Only first 31 characters are significant
4. Cannot use a keyword
5. Must not contain white space

1.2.2. Data types

C language is rich in its data types. Storage representation and machine instructions to handle constants differ from one machine to machine. C supports three classes of data types

1) Primary (or fundamental) data types
2) Derived data types (arrays, functions, pointers, structures and unions)
3) User – defined Data types

1) Primary Data types:

C compiler supports five fundamental data types

- integer(int)
- floating point(float)
- character(char)
- double – precision point(double)
- void

and extended data types such as long int, long double

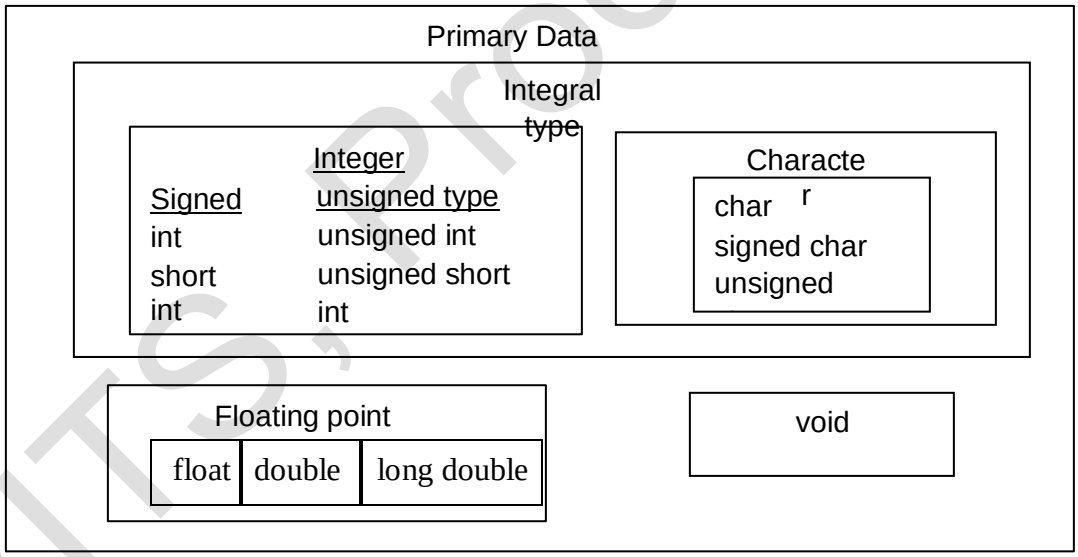


Fig. Primary data types in C

Table: Size and range of basic data types on 16 bit machine

Data type	Range of values
char	-128 to 127
int	-32768 to 32767
float	3.4e-38 to 3.4e+38
double	1.7e-308 to 1.7e+308

Integer types:

- Integers are whole numbers with a range of values supported by a particular machine
- Integer occupy one word of storage
- Word size of machines vary(16 or 32 bits)
- If we use a 16 bit, the range is -32768 to 32767 (that is, -2^{15} to $+2^{15}-1$)
- A signed integer uses one bit for sign and 15 bits for the magnitude of the number
- 32 bit store an integer ranging from 2,147,483,648 to 2,147,483,647
- C has three classes of integer storage
    - short
    - int
    - long int

  in both signed and unsigned forms
- Short int represents small integer and requires half the amount of storage as a regular int
- Unsigned integers use all the bits for the magnitudes the number and are always positive then in 16 bit the range is 0 to 65535
- Default declaration is a signed number

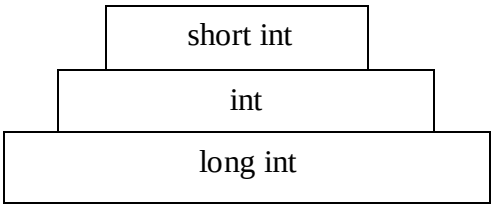


Fig. Integer types organizes from the smallest to the largest

Table: size and range of data types on a 16 – bit machine

Type	Size		Range
	Bit	bytes	
char or signed char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int or signed int	16	2	-32768 to 32767
unsigned int	16	2	0 to 65535
short int or signed short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int or signed long int	32	4	-2,147,483,648 to 2,147,483,647
unsigned long int	32	4	0 to 4,294,967,295
Float	32	4	1.2E-38 to 3.4E+38
Double	64	8	2.3E-308 to 1.7E+308
Long double	80	10	3.4E-4932 to 1.1E+4932

Floating point types:

- Floating point(or real) numbers are stored in 32 bits (on all 16 or 32 bit machines), with 6 digits precision
- Floating point number are defined in C by the keyword **float**
- When the accuracy provided by float is not sufficient the type double used
- A double data type uses 64 bits giving precision of 14 digits
- These are known as double precision number
- To extend the precision further, we may use long double which uses 80 bits

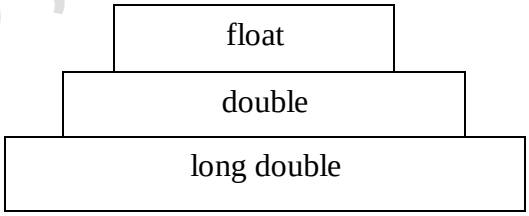


Fig. floating point types

Void type:

- The void type has no values
- It is usually used to specify the type of functions
- The type of a function is said to be void when it does not return any value to the calling function

Character types:

- A single character can be defined as a character(char) type data

- Characters are usually stored in 8 bits (one byte) of internal storage
- Unsigned chars have values between 0 and 255, signed chars have values from

  -128 to 127

Table: Data types and their keywords

Data type	Keyword equivalent
character	char
unsigned character	unsigned char
signed character	signed char
signed integer	signed int (or int)
signed short integer	signed short int (or short int or short)
signed long integer	signed long int (or long int or long)
unsigned integer	unsigned int (or unsigned)
unsigned short integer	unsigned short int (or unsigned short)
unsigned long integer	unsigned long int (or unsigned long)
floating point	float
double – precision floating point	double extended double – precision floating point long double

3) User defined type declaration:

There are two types of user – defined data types

- Type definition
- Enumerated

Type definition:

- It allows user to define an identifier that would represent an existing data type
- It can later be used to declare variables typedef type identifier;
  type refers to an existing data type and identifier refers to the new name given to the data type
- The new type is new only in name, but not the data type
- Typedef cannot create a new type
- Ex: typedef int marks;
  marks name1,name2;
- By this it increases the readability of the program

Enumerated data type:

- It is 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 of this new type
- It is defined as follows:

  enum identifier{value1,value2,value3,…….,valuen}; enum identifier v1,v2,v3,…..... ,vn;
- The variables v1,v2,…….,vn can have one of the values value1,value2,……..

  v1=value3; v5=value1;
- An ex: enum day{mon,tue,wed,thu,fri,sat,sun};

  enum day week_st,week_end;

  week_st=mon;

  week_end=fri;

  if(week_st==tue)

  week_end=sat;
- The compiler automatically assigns integer digits with 0 to all enum constants
- That is enum constant value1 is assigned 0, value2 is assigned 1 and so on
- The automatic assignments can be overrided by explicit values

  enum day{mon=1,tue,…..... , sun};

  here next constants increase successively by 1
- The definition and declaration can be combined in one statement as
- Ex: enum day{mon,… ,sun}

  week_st,week_end;

Additional Topics:

Constants:

Constant refers to fixed values that do not change during the execution of a program

Several types of constants in C are,

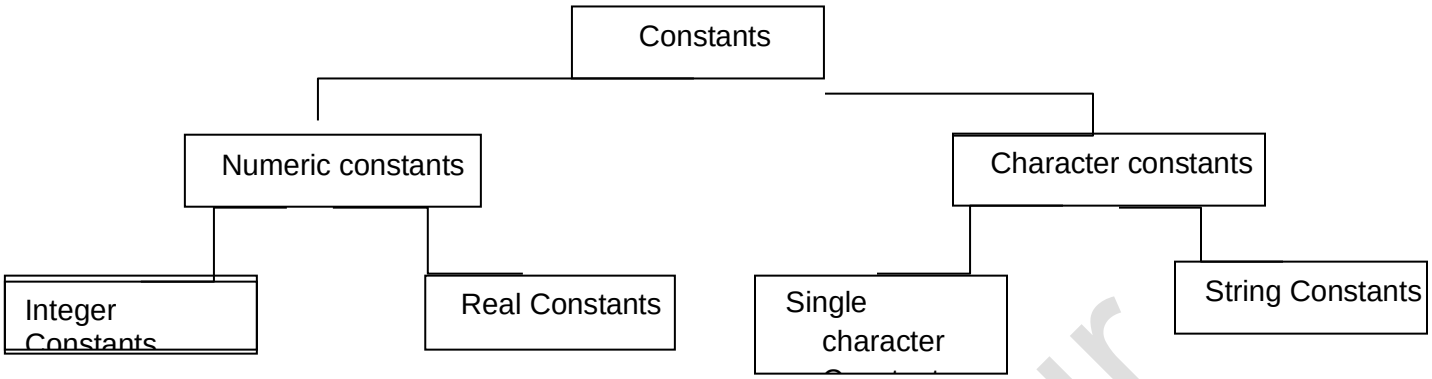


Fig. Basic types of C constants

Integer constants:

An integer refers to a sequence of digits. Three types of integers are:

- Decimal integer
- Octal integer
- Hexadecimal integer

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign

Ex: 123 -321 0 654321 +78 are valid

15 50 20,000 $1000 are illegal

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

Ex: 037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as hexadecimal integer and includes alphabets A through F or a through f

Ex: 0X2 0x9F 0Xbcd

To store larger integer constants we can append qualifier such as U,L and UL to the constants

Ex: 56789U or 56789u (unsigned integer)

98761247UL or 98761234ul(unsigned long integer)

9876543L or 9876543l (long integer)

Real constants:

The quantities represented by numbers containing fractional parts are called real(or floating point) constants

Ex: 0.0083 -0.75 435.36 +247.0

are in decimal notation

215. .95 -.71 7.5 are also valid

A real number may also be expressed in exponential(or scientific) notation For ex: 215.65 -> 2.1565e2

e2 means multiply by 10^2

The general form is:

Mantissa e exponent

Mantissa is either a real number in decimal notation or an integer e written in lowercase or uppercase exponent it is an integer with an optional plus or minus sign

ex: 0.65e4 12e-2 1.5e+5 -1.2E-1

Single character constants:

It contains a single character enclosed with in a pair of single quote marks

Ex: '5' 'X' ';' ' '

Here 5 is not a number it is character. Characters constants have integer values known as ASCII values

Ex: printf("%d",'a');
gives 97
printf("%c",'97');
prints the letter a

String constants:

A string constant is a sequence of characters enclosed in double quotes

"Hello!" "1987" "well done" "5+3" "x" 'x' is not equal to "x"

String constants does not have an equivalent integer value

Backslash character constants:

These are also known as escape sequences

Although it consists of two characters it represents as only one character

Table: Backslash character constants

Constant	Meaning
'\a'	Audible alert(bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote

'\?'	question mark
'\\'	backslash
'\0'	null

Structure of a C program:

- C program can be viewed as a group of building block called **function**
- A function(also called as subroutine or procedure) is that include one or more statements to perform a specific task
- To write a c program, we first create functions and then put them together
- A C Program may contain one or more sections as shown.

```
Documentation Section Link Section
Definition Section
Global Declaration Section
main() /*function section*/
{
        Declaration part Executable part
}
Sub program section /*User defined function*/
  Function 1
Function 2
    .
    .
Function n
```

- The **documentation section** consists of comment lines giving the name of the program, the author and other details, which the programmer would like to use later
- The **link section** provides instructions to the compiler to link function from the system library
- The **definition section** defines all symbolic constants
- There are some variables that are used in more than one function such variables are called **global variables** and are declared in the **global declaration section** that is outside of all the functions
- This section also declares all the user defined function
- Every C program must have one main() function section. This section contains two parts
  - Declaration part - Executable part
- The **declaration part** declares all the variables used in the executable part
- There is atleast one statement in the executable part

- These two parts must appear between the opening and the closing braces
- The program execution starts at the opening brace and ends at the closing brace
- The closing brace of the main function section is the logical end of the program
- All statements in the declaration and executable parts end with a semicolon(;)
- The **subprogram** section contains all the user defined functions in the main function
- User defined functions are generally placed immediately after the main function, although they may appear in any order.

Simple C programs to exchange the values of two variables:

```
/*C Program to exchange the values of two variables using temp variable*/ #include<stdio.h>
#include<conio.h>
void main()
{
int a,b,t;
clrscr();
printf("enter the values of and b\n"); scanf("%d%d",&a,&b);
t=a;
a=b;
b=t;
printf("the values after exchange are: a=%d, b=%d",a,b);
getch();
}
```

```
/*C Program to exchange the values of two variables using temp variable*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,t;
clrscr();
printf("enter the values of and b\n");
scanf("%d%d",&a,&b);
a=a+b;
b=a-b;
a=a-b;
printf("the values after exchange are: a=%d, b=%d",a,b);
getch();
}
```

Counting the number of factors of a given integer

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

```
#include<conio.h>
void main()
{
int n,i,c=0;
clrscr();
 printf("enter the value of n");
 scanf(""%d",&n);
 for(i=1;i<=n;i++)
{
if(n%i==0)
{
c=c+1;
}
}
printf("the number of factors of given number %d are: %d",n,c);
}
```

1.3.Operators

- An **operator** indicates an operation to be performed on data that yields a value
- Using various operators in C one can link the variables and constants.
- An **operand** is a data item on which operators perform the operations
- C is rich in use of different operators

Type of operator	Symbolic Representation
1. Arithmetic operators	+, -, *, / and %
2. Relational operators	>, <, ==, >=, <= and !=
3. Logical operators	&&, ||, and !
4. Increment & decrement operator	++ and –
5. Assignment operator	=
6. Bitwise operator	&, |, ^, >>, <<, and ~
7. Special operator	,, sizeof, pointer operators(& And *), Member selection(. &->
8. Conditional operator	? :

1. Arithmetic operators:

Operator	Meaning	Example
+	Addition or unary plus	2+2=4

-	Subtraction or unary minus	5-3=2
*	Multiplication	2*5=10
/	Division	10/2=5
%	Modulo division	11%3=2(remainder)

Integer Arithmetic:

When both operands are integers then the operation is called integer arithmetic a = 15/7 = 2.1 = 2

Real arithmetic:

An arithmetic operation involving only real operands is called real arithmetic x = 6.0/7.0 = 0.857143

the operator % cannot be used with real operands Mixed mode arithmetic:

When one operand is real and the other is integer the expression is called a mixed – mode arithmetic expression. If either operand is of real type, then only the real operation is performed and the result is always real number

15/10.0 = 1.5

2. Relational operators:

These operators provides the relationship between two expression. If the relation is true then it return a value 1 otherwise 0 for false relation

Operator	Description or Action	Example	Return value
>	Greater than	5>4	1
<	Less than	10<9	0
>=	Greater than or equal to	11>=5	1
==	Equal to	2==3	0
!=	Not equal to	3!=3	0

```
/* program to use various relational operators and display their return values*/
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   printf("\n Condition : Return values\n");
   printf("\n10!=10 : %5d",10!=10);
   printf("\n10==10 : %5d",10==10);
   printf("\n10>=10 : %5d",10>=10);
   printf("\n 10<=100 : %5d",10<=100);
   printf("\n10!=9 : %5d",10!=9);
```

```
    getch();
}
```

Output:

Condition : Return
values 10!=10 : 0
10==10 : 1
10>=10 : 1
10<=100 : 1
10!=9 :1

3. Logical operators:

The logical relationship between the two expressions are checked with logical operators. The operators could be constants, variables and expressions. After checking the conditions it provides logical true(1) or false(0) status.

Operator	Description or Action	Example	Return Value
&&	Logical AND	5>3&&5<10	1
||	Logical OR	8>5||8<2	1
!	Logical NOT	8!=8	0

```
/* program to use various logical operators and display their return
values*/
#include<stdio.h>
#include<conio.h> void main()
{
        clrscr();
        printf("\n Condition : Return values\n");
        printf("\n5>3 && 5<10 : %5d",5>3 && 5<10);
        printf("\n8>5 || 8<2 : %5d",8>5 || 8<2);
        printf("\n!(8==8) : %5d",!(8==8));
        getch();
}
```

Output:

Condition : Return values 5>3 && 5<10 : 1
8>5 || 8<2 : 1
!(8==8) : 0

4. Increment(++) and Decrement(--) operators:

- The operator ++ adds 1 to the operand, while – subtracts 1

- Both are unary operators and take the following form:

  ++m; or m++;

  --m; or m--;

  ++m is equivalent to m=m+1; (or m+=1)

  --m is equivalent to m=m-1; (or m-=1;)
- If "++" or "—" are used as a suffix to the variables name then the post increased/decreased operations takes place
- Ex:

  take x=10;

  y=20;

1) z=x*y++;

   a=x*y; then,

   z=200

   a=210

2) z=x*++y;

   a=x*y; then,

   z=210

   a=210

/*program to show the effect of increment operator as suffix*/ #include<stdio.h>

```
#include<conio.h>
void main()
{
int a,z,x=10,y=20;
clrscr();
z=x*y++;
a=x*y;
printf("\n%d \t%d",z,a);
getch();
}
```

/*program to show the effect of increment operator as prefix*/ #include<stdio.h>

```
#include<conio.h>
void main()
{
int a,z,x=10,y=20;
```

```
clrscr();
z=x*++y;
a=x*y;
printf("\n%d \t%d",z,a);
getch();
}
```

5. Assignment operator:

- Assignment operators are used to assign the result of an expression to a variable
- Assignment operator is "="
- C has a 'short hand' assignment operators of the form v op = exp;

  v is a variable

  op is a binary arithmetic operator

  exp is an expression

  v op = exp; is equivalent to v = v op (exp);

  Ex: x+=y+1; is same as x=x+(y+1);

  x+=3; is equal to x=x+3;

6. Bitwise operators:

- Bitwise operators are used for manipulation of data at bit level
- This may not be applied to float or double and can be applied only for int, char, short, long int etc

Operator	Meaning
>>	Right shift
<<	Left shift
^	Bitwise XOR(excluiveOR)
~	One's complement
&	Bitwise AND
|	Bitwise OR

Ex 1. x=8

x>>2

000000000001000 -> 8

0000000000000010 ->2

2. a=8

b=4

c=a&b

0000000000001000 -> 8

0000000000000100 -> 4

0000000000000000 -> 0

C=0

```
/*program to shift input data by two bits right*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int x,y; clrscr();
    printf("Read the integer from the keyboard (x):"); scanf("%d",&x);
    x>>2;
    y=x;
    printf("the right shifted data is = %d",y);
    getch();
}
```

Output:

Read the integer from keyboard(x):8

The right shifted data is = 2

7. Special operator:

- There are three types of special operators
  , , sizeof, pointer(& and *) and member selection operators(.and->)
- Comma operator is used to separate two or more expressions
- It has the lowest priority among all operators and are evaluated left to right
  value = (x=10, y=5, x+y)
  value = 15
- Size of operator gives the bytes occupied by a variable int sum;
  m=sizeof(sum); then m=2
- The remaining will be discussed in the later topics

```
/* program to illustrate the use of comma (,) operator*/
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
```

```
printf("Addition = %d\n subtraction = %d",2+3,5-4);
getch();
}
```

Output:

Addition = 5

Subtraction = 1

```
/* program to use & and sizeof() operator and determine the size of integer and float variables*/
#include<stdio.h>
#include<conio.h>
void main()
{
int x=2;
float y=2;
clrscr();
printf("\nsizeof(x) = %d bytes",sizeof(x));
printf("\nsizeof(y) = %d bytes",sizeof(y));
printf("\nAddress of x=%u and y=%u",&x,&y);
getch();
}
```

Output:

sizeof(x)=2

sizeof(y)=4

Address of x=4066 and y=25096

8. Conditional operator:

- The conditional operator contains a condition followed by two statements or values
- If condition is true the first statement is executed otherwise the second statement
- Conditional operator? and : are sometimes called "ternary operators" because they take three arguments

  Syntax:

  ```
  condition ? (expression1) : (expression2)
  ```

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

  then

  x=15

  it can be written using if…else

```
    if(a>b)
       x=a;
    else
       x=b;
```

/*program to use the conditional operator with two values*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    printf("Result=%d",2==3?4:5);
    getch();
}
```

Output:

Result=5

/*program to use the conditional operator with two statements*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    3>2?printf("True"):printf("False");
    getch();
}
```

Output:

True

4. Expressions

- An expression is a combination of operators and operands which reduces to a single value.
- Operator: it indicates an operation to be performed on data that yields a value
- Operand: It is a data item on which an operation is performed
- A **simple expression** contains only one operator
  For ex: 3+5 is a simple expression which yields a value 8,
  -a is also single expression
- A **complex expression** contains more than one operator
  For ex: 6+2*7

- An expression can be divided into six categories based on,
    - The number of operators,
    - Position of the operands and operators
    - Precedence of operator
- The categories are,

Primary	Unary
Postfix	Binary
Prefix	ternary

1. **Primary expressions:**
    - In C operand in the primary expression can be a name, a constant, or an parenthesized expression
    - Name: It is an identifier for a variable.
    - Constant: A constant is the one whose value which cannot be changed during program execution
    - Any value enclosed in within parentheses must be reduced to a single value
    - A complex expression can be converted into primary expression by enclosing it with parentheses
      Ex: (3*5+8); (c=a=5*c);

2. **Postfix expressions:**
    - The postfix expression consists of one operand and one operator comes after the operand
    - Ex: Post increment: In this the variable is increased by '1' a++ results in the variable increment by '1'
      Post decrement: In this, the variable is decreased by '1'
      a—results the variable decreased by '1'
3. **Prefix expressions**
    - Prefix expressions consists of one operand and one operator, the operand comes after the operator.
    - Ex: prefix expressions are
      Prefix increment: ++a
      Prefix decrement: --a
    - The only difference between prefix and postfix operators is in the prefix operator, the effect take place before the expression
4. **Unary expressions**
    - An unary expression is like a prefix expression consists of one operand and one operator and the operand comes after the operator

- Ex: ++a; -b; -c; +d;

5. Binary expressions

- Binary expressions are the combinations of two operands and an operator
- Any two variables added, subtracted, multiplied, or divided is a binary expression
- Ex: a+b; c*d;

6. Ternary expressions:

- Ternary expressions is an expression which consists of a ternary operator pair "?:"
- Ex: exp1?exp2:exp3;

In the above example, if exp1 is true exp2 is executed else exp3 is executed

5. Precedence and associativity:

- Every operator has a precedence value
- An expression containing more than one operator is known as complex expression
- Complex expressions are executed according to precedence of operators
- Associativity specifies the order in which the operators are evaluated with the same precedence in a complex expression
- Associativity is of two ways i.e., left – to – right and right – to – left
- The precedence and associativity of various operators in C are as shown in the following table

Operators	Operation	Associativity	Precedence
( )	Function call		
[ ]	Array expression or square bracket	Left to Right	1st
->	Structure operator		
.	Structure operator		
+	Unary plus		
-	Unary minus		
++	Increment	Right to Left	2nd
--	Decrement		

	!	Not operator		
	~	One's complement		
	*	Pointer operator		
	&	Address operator		
	Sizeof	Size of an object		
	type	Type cast		
	*	Multiplication		
	/	Division	Left to Right	3rd
	%	Modular division		
	+	Addition	Left to Right	4th
	-	Subtraction		
	<<	Left shift	Left to Right	5th
	>>	Right Shift		
	<	Less than		
	<=	Less than or equal to	Left to Right	6th
	>	Greater than		
	>=	Greater than or equal to		
	==	Equality	Left to Right	7th
	!=	Inequality		
&		Bitwise AND	Left to Right	8th
^		Bitwise XOR	Left to Right	9th
|		Bitwise OR	Left to Right	10th
&&		Logical AND	Left to Right	11th
||		Logical OR	Left to Right	12th

? :	Conditional operator	Right to Left	13th
=, *=, -=, &=, += ^=, !=, <<=, >>=	Assignment operators	Right to Left	14th
,	Comma operator	Left to Right	15th

Evaluation of expressions:

- Expressions are evaluated using an 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 values before evaluation is attempted
- Examples of evaluation
  statements are x=a*b-c;
  y=b/c*a;
  z=a-b/c+d;

```
/*Program to illustrate the use of variables in expressions and their evaluation*/
#include<stdio.h>
#include<conio.h>
void main()
{
        float a,b,c,x,y,z;
        clrscr();
        a=9; b=12; c=3;
        x=a-b/3+c*2-1;
        y=a-b/(3+c)*(2-1);
        z=a-(b/(3+c)*2)-1;
        printf("x=%f\n",x);
        printf("y=%f\n",y);
        printf("z=%f\n",z);
        getch();
}
```

Output:

x=10.000000
y=7.000000

z=4.000000

2. Conditional Statements(or Decision statements):

- A program is nothing but the execution of sequence of one or more instructions
- Depending upon on certain situations it is desirable to alter the sequence of the statements in the program
- Based on the conditions we change the order of the execution of statements
- It involves decision making condition to see whether a particular condition is satisfied or not
- To alter the flow of program
    - Test the logical condition
    - Control the flow of execution as per the selection
    - The conditions can be placed in the program using conditional statements
- C language supports the following control statements
    - The if statement
    - The if – else statement
    - The Nested if – else statement
    - The if – else – if ladder
    - The switch() case statement

The if statement:

The general form of a simple if statement is

Syntax:

```
if(condition)   /*no semicolon*/
{
 Statement – block;
}
Statement x;
```

- The statement – block may be a single statement or a group of statements
- If the condition is true, the statement block will be executed; otherwise the statement – block will be skipped and the execution will jump to the statement x
- When the condition is true both the statement – block and the statement x are executed in sequence.
- This is as shown in flowchart
- The statements following the if statement are normally enclosed with curly braces
- The curly braces indicates the scope of if statement
- The default scope is one statement

- The statement – block contain one statement may or may not give the curly braces, but it is a good practice to use curly braces even with a single statement

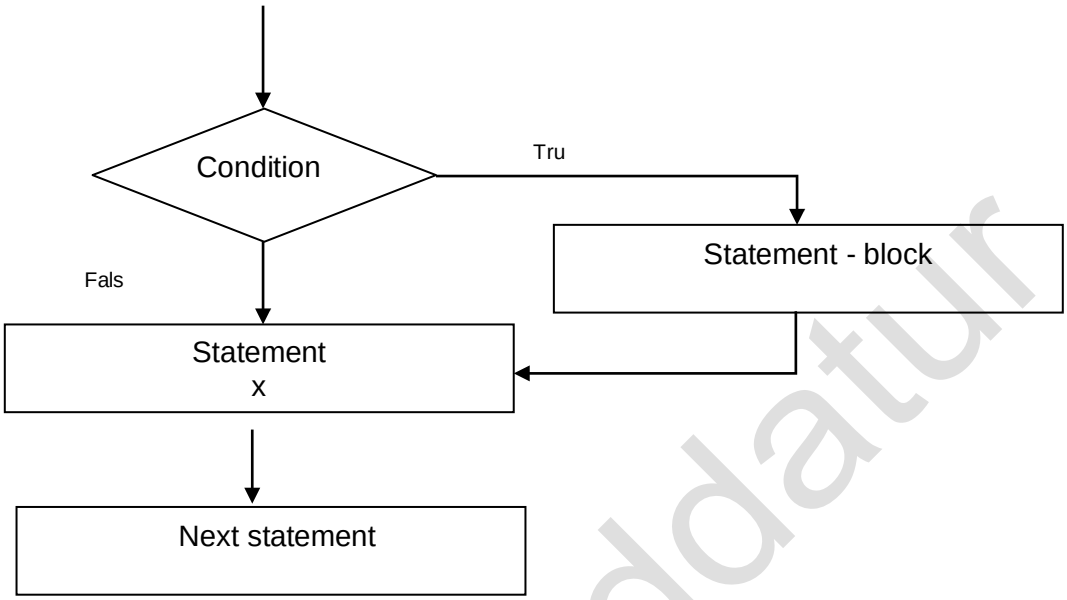


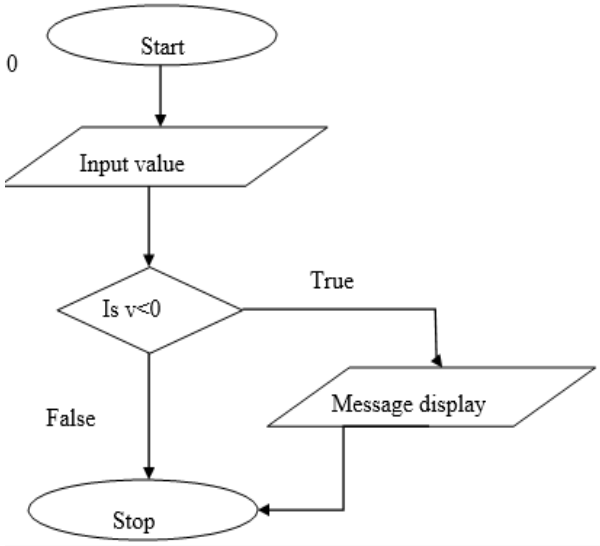
Fig: Flow chart of simple if control

```
/*program to demonstrate the use of if statement*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int v;
        clrscr();
        printf("enter the number:\n");
        scanf("%d",&v);
        if(v<10)
                printf("\nNumber entered is less than 10");
        getch();
}
```

Output:

```
enter the number:
8
Number entered is less than 10
```

Flow chart:



The if … else statement:

- It is observed that if statement executes only when the condition following if is true
- It does nothing when the condition is false
- But, if……else statement takes care of true as well as false conditions
- The if……else statement is an extension of the simple if statement. The general form is

```
if(condition)    /*no semicolon*/
{
        True – block statement(s)
}
else
{
        False – block statement(s)
}
Statement x;
```

- If the condition is true, then the true block statements immediately following the if statements are executed; otherwise, the false – block statements are executed
- In either case, either true – block or false – block will be executed, not both
- This is shown in fig

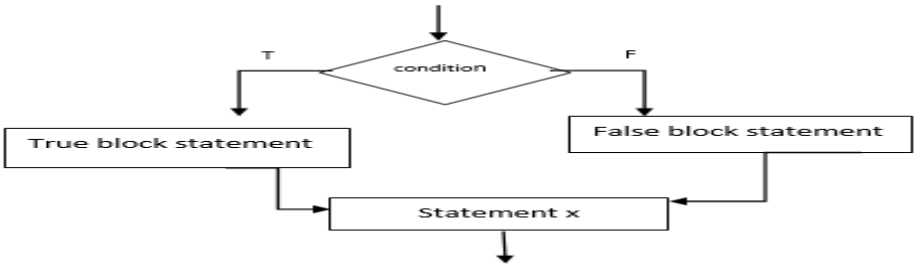


Fig: Flowchart of if.....else control

- In both the cases, the control is transferred subsequently to the statement x
- The else statement cannot be used without if
- No multiple else statements are allowed with one if

/* Demonstration of if…else statement is given in the following programs. Read the values of a, b, c through the keyboard add them and after addition check it is in range 100 & 200 or not. Print separate message for each*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,c,d;
        clrscr();
        printf("enter three numbers a b c:");
        scanf("%d%d%d",&a,&b,&c);
        d=a+b+c;
        if(d<=200 &&d>=100)
                printf("\n sum is %d which is in between 100 & 200",d);
        else
                printf("\n sum is %d which is out of range",d);
         getch();
}
```

Output:

```
100    20   10
sum is 130 which is in between 100 & 200
```

/* program to find the roots of a quadratic equation by using if else condition*/

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

```
        float x1,x2;
        clrscr();
        printf("\n enter values for a,b,c:");
        scanf("%d%d%d",&a,&b,&c);
        if(b*b>4*a*c)
        {
            x1=-b+sqrt(b*b-4*a*c)/2*a;
            x2=-b-sqrt(b*b-4*a*c)/2*a;
             printf("\nx1=%f x2=%f",x1,x2);
        }
      else
           printf("\nRoots are imaginary");
          getch();
  }
```

Output:

enter values of a,b,c: 5 1 5

Roots are imaginary

Nested if – else statement:

- When a series of decisions are involved, we may have to use more than one if….else statement in nested form as shown below
- The logic of execution is illustrated in fig.
- If the condition is false the statement 3 will be executed
- Otherwise it continues to perform the second test
- If the condition 2 is true, the statement 1 will be evaluated and then the control is transferred to the statement x

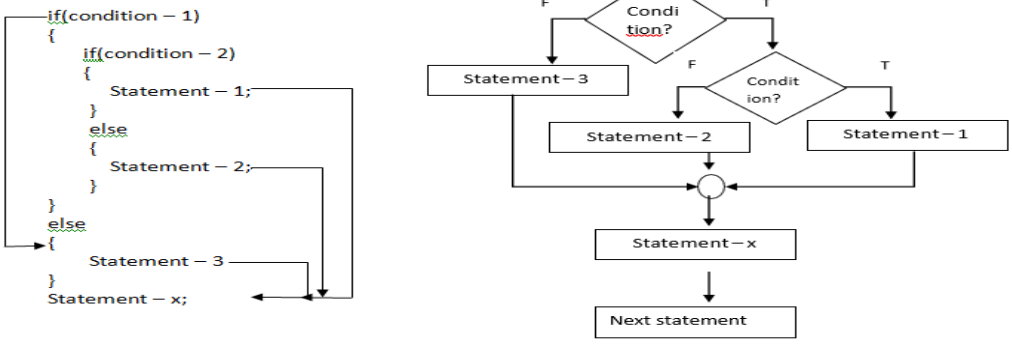


Fig. Flowchart of nested if....else statements

- A commercial bank has introduced an incentive policy of giving bonus to all deposit holders.
- The policy is as follows: A bonus of 2 percent of the balance held on 31st December is given to everyone, irrespective of their balance and 5 percent is given to female account holders if their balance is more than rs.5000. this logic can be coded as follows:
- …….

```
if(person is female)
{
if(balance>5000)
     bonus=0.05*balance;
else
       bonus=0.02*balance;
}
else
{
   bonus=0.02*balance;
}
balance=balance+bonus;
…….
```

- In C, an else is linked to the closest non terminated if
- Consider alternative

```
if(person is female)
if(balance>5000)
   bonus=0.05*balance;
else
   bonus=0.02*balance;
balance=balance+bonus;
```

- The else is associated with the inner if and there is **no else option for the outer if**
- So in this case it will not calculate bonus for male account holders
- Consider alternative

```
if(person is female)
{
 if(balance>5000)
       bonus=0.05*balance;
}
else
       bonus=0.02*balance;
 balance=balance+bonus;
```

- In this case, else is associated with the outer if and therefore bonus is calculated for the male account holders and does not calculate for female account holders, whose balance is equal to or less than 5000.

```
/*selecting the largest of three numbers*/
 #include<stdio.h>
#include<conio.h>
void main()
{
float a,b,c; clrscr();
printf("enter three values\n"); scanf("%f%f%f",&a,&b,&c);
printf("\nlargest value is");
if(a>b)
{
if(a>c)
printf("%f\n",a);
else
printf("%f\n",c);
}
else
{
if(c>b)
printf("%f\n",c);
else
printf('%f\n",b);
}
getch();
}
```

Output:

```
enter three values
23445          67379          88843
Largest value is 88843.000000
```

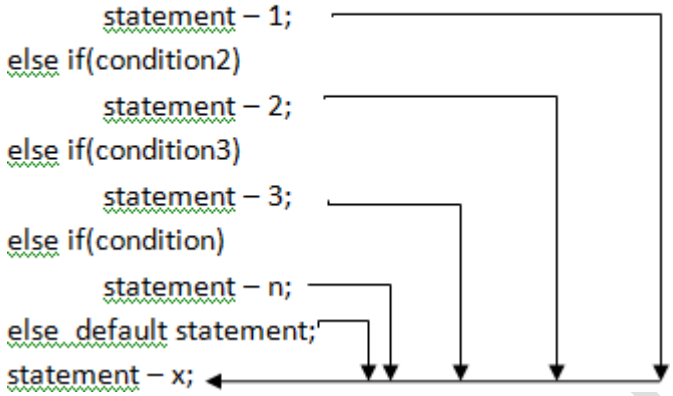
a) else – if ladder

- There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if

- The following general form:

Syntax:

```
if(condition1)
        statement – 1;
else if(condition2)
        statement – 2;
else if(condition3)
        statement – 3;
else if(condition)
        statement – n;
else  default statement;
statement – x;
```



  - This construct is also known as the else if ladder
  - The conditions are evaluated from top(of the ladder), downwards
  - As soon as true condition is found, the statement associated with it is executed and the control is transferred to the statement x.
  - When all the n conditions become false, then the final else containing the default – statement will be executed.
  - Figure shows the logic execution of else if ladder statements

/*write a program to calculate energy bill. Read the starting and ending meter reading. The charges are as follows.

No. of units consumed	Rate in (Rs.)
200 – 500	3.50
100 – 200	2.50
Less than 100	1.50 */

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int initial,final,consumed;
    float total;
    clrscr();
    printf("\n Initial and final readings:");
    scanf("%d%d",&initial,&final);
    consumed=final-initial;
    if(consumed>=200 && consumed<=500)
    total=consumed*3.50;
    else if(consumed>=100 && consumed<=199)
        total=consumed*2.50;
    else if(consumed<100)
```

```
        total=consumed*1.50;
    printf("Total bill for %d unit is %f",consumed,total);
    getch();
}
```

Output:

Initial & final readings:

800 850

Total bill for 50 unit is

75.000000

b) Switch statement:

- The switch statement is a multi way branch statement
- We can use an if statement to control the selection, but when the number of alternatives increases then the complexity of program increases
- The switch statement tests the value of a given variable(or expression) against list of case values and when a match is found, a block of statements associated with the case is executed
- The general form of switch statement is as shown below: Syntax:

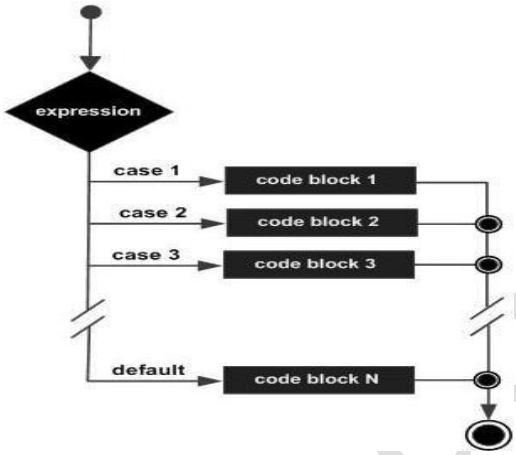
```
switch(expression)
{
    case value-1: block – 1
                break;
     case value-2: block –2
                    break;
     ……….
     ……..
    default: default block break;
}
statement-x;
```

- The expression is an integer expression or characters
- value-1, value-2, are constants or constant expressions(evaluable to an integer constant) and are known as **case labels**
- The case labels end with **:** and each of these values should be unique within the switch statement
- block – 1, block – 2,……. Are statement lists and may contain zero or more statements and there is no need to put braces around these blocks
- when the switch is executed, the value of expression is successfully compared against the values value-1, value-2,…….
- If a case found whose value matches with the value of the expression, then the block of statements that follows the case are executed
- The break statement at the end of each block signals the end of a particular

case and causes an exit from the switch statement, transferring the control to the statement-x following the switch

- The default is an optional case, will be executed if the value of the expression does not match with the any of the case values
- If the default statement is not present, no action takes place if all matches fail and control goes to the statement-x

- The selection process of switch statement is illustrated in the flowchart as shown in fig.



program to convert years into 1. Minutes 2. Hours 3. Days 4. Months 5. Seconds using switch() statement/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    long ch,min,hrs,ds,mon,yrs,se;
    clrscr();
    printf("\n[1] MINUTES");
    printf("\n[2] HOURS");
    printf("\n[3] DAYS");
    printf("\n[4] MONTHS");
    printf("\n[5] SECONDS");
    printf("\n[0] EXIT");
    printf("\n enter your choice");
    scanf("%ld",&ch);
     if(ch>0&&ch<6)
      {
         printf("enter years:");
         scanf("%ld",&years);
      }
      mon=yrs*12;
```

```
        ds=mon*30;
        ds=ds+yrs*5;
        hrs=ds*24;
        min=hrs*60;
        se=min*60;
        switch(ch)
        {
              case 1: printf("\n Minutes: %ld",min);
                      break;
              case 2: printf("\n Hours: %ld",hrs);
                      break;
              case 3: printf("\n Days: %ld",ds);
                      break;
              case 4: printf("\n Months: %ld",mon);
                      break;
              case 5: printf("\n seconds: %ld",se);
                     break;
              case 0: printf("\n Terminated choice"):
                       exit(0);
                       break;
              default: printf("\n Invalid choice");
        }
          getch();
    }
```

Output:

[1] MINUTES
[2] HOURS
[3] DAYS
[4] MONTHS
[5] SECONDS
[0] EXIT

enter your choice: 4

enter years:2

Months: 24

3. **Iterative statements (or loop control statements or repetition statements):**
   - A loop is defined as a block of statements which are repeatedly executed for a certain number of times
   - A program loop therefore consists of two segments, one known as the "body of the loop" and the other known as the "control statement"

- The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop
- Depending on the position of the control statement in the loop, a control structure may be classified either as the "entry – controlled loop", or as the "exit controlled loop".
    - Entry controlled loop: In this, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. It is also known as "**pre – test loop**"
    - Exit controlled loop: In this, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. It is also known as "**post – test loop**"
    - The test conditions should be carefully stated in order to perform the desired number of loop executions and should not set up an infinite loop

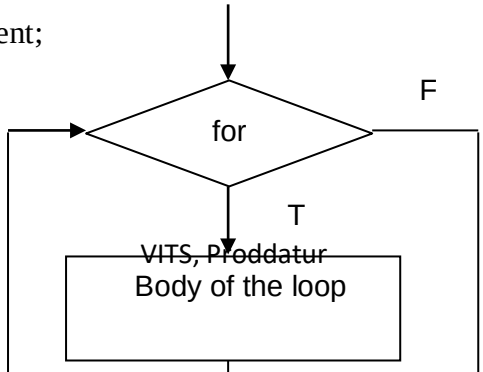
Steps in loop:

- *Loop variable:* it is a variable used in loop
- *Initialization:* It is the first step in which starting and final value is assigned to the loop variable. Each time the updated value is checked by the loop itself
- *Incrementation/Decrementation:* It is the numerical value added or subtracted to the variable in each round of the loop
- C language supports the three types of loop control statements
    a. The while statement (entry controlled loop)
    b. The do statement (exit controlled loop)
    c. The for statement (entry controlled loop)

3.1. **The for statement:**

- The for loop allows to execute a set of instructions until a certain condition is satisfied.
- Condition may be predefined or open ended
- The for loop is an entry controlled loop
- The for loop statement comprises of three actions
- The three actions are placed in the for statement itself
- The three actions are:
    - Initialization Condition
    - Increment/decrement are included in one statement
- The expressions are separated by semi – colons(;)
- The general syntax of the loop is

```
for(initialization;condition;increment/decrement)/*no semicolon*/
{
    Body of the loop
}
Next statement;
```



Flow chart for

For loop Explanation:

1. The initialization sets a loop to an initial value. This statement is executed only once
2. The condition is a relational expression that determines the number of iterations desired or it determines when to exit from the loop. The for loop continues to execute as long as conditional test is satisfied. When the condition becomes false
   the control of the program exits from the body of the loop and executes next statement after the body of the loop
3. The increment/decrement decides how to make changes in the loop
   - The body of the loop may contain either a single statement or multiple statements
   - In case there are only one statement after the for loop braces may not be necessary. Good to practice to use braces even for a single statement
   - for(; ;) will give infinite loop
   - for(a=0;a<=20;) will also produce infinite loop
   - for(a=0;a<=10;a++) printf("%d",a); will displays value from 0 to 10
   - for(a=10;a>=0;a--) printf("%d",a); will displays value from 10 to 0

```
/*program to display numbers from 1 to 15 using for loop*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1;i<=15;i++)
    printf("%5d",i);
    getch();
 }
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Nested for loops:

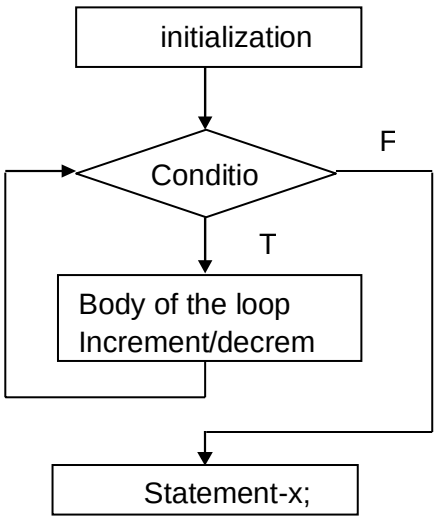
- Nesting of loops, that is, one for statement within another for statement is allowed in C
- For ex, two loops can be nested as follows:

```
…….
…….
for(i=1;i<10;i++) /* outer loop begins*/
{
……..
……..
for(j=1;j!=5;j++)  /* inner loop begins*/
{
……
……
}   /*inner loop ends*/
…….
…….
}   /*outer loop ends*/
      ……..
      …….
```

- The number of iterations in this type of structure will be equal to the number of iterations in the outer loop multiplied by the number of iterations in the inner loop
- The nesting may continue up to any desired level

/*writing a program to illustrate an example based on nested for loops*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    clrscr();
    for(i=1;i<=3;i++)/*outer loop*/
    {
       for(j=1;j<=2;j++)/*inner loop*/
          printf("\n i*j:%d",i*j);
    }
    getch();
}
```



Output:

1*1:1

1*2:2

2*1:2

2*2:4

3*1:3

3*2:6

3.2. **The while loop:**

- This loop is another entry controlled loop
- The syntax is as follows initialization; while(test condition)
  {
  Body of the loop Increment/decrement;
  }
  Statement-x; flow chart for while loop
- The test condition may be any expression
- The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed
- When the condition becomes false the execution will be out of the loop
- Steps of while loop are as follows:
  1. The test condition is evaluated and if it is true, the body of the loop is executed
  2. On execution of the body, test condition is repetitively checked and if it is true the body is executed
  3. The process of execution of the body will continue till the test condition becomes false
  4. The control is transferred out of the loop
- The braces are needed only if the body of the loop contains more than one statement

/*write a program to print the string "you have learnt C program" 9 times using while loop*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int x=1;
   clrscr();
   while(x<10)
   {
      printf("\n you have learnt C program");
   }
   getch();
 }
```

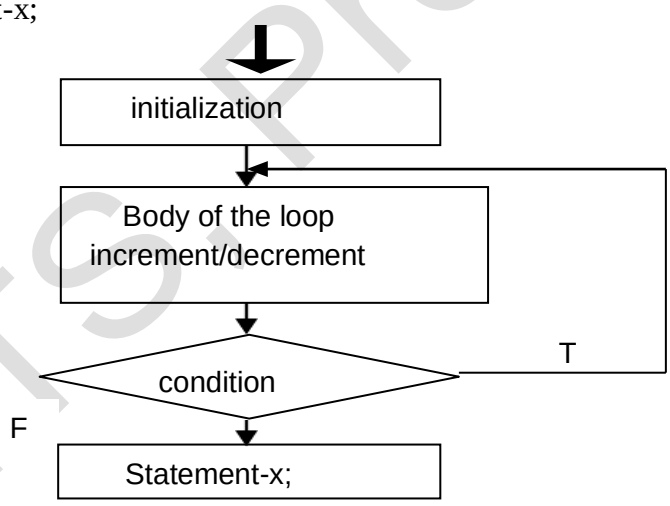
Output:

you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program
you have learnt C program

3.3. The do –while loop:

- The do – while loop is an exit controlled loop and therefore the body of the loop is "always executed atleast once"
- The syntax is as follows: initialization

```
do
{
   Body of the loop;
   increment/decrement;
}
while(condition);
statement-x;
```



flow chart of do – while loop

- On reaching the do statement, the program proceeds to evaluate the body of the first
- At the end of the loop, the test condition in the while statement is evaluated
- If the condition is true; the program continues to evaluate the body of the loop once again
- The process continues as long as condition is true
- When the condition is becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement

/*illustrating do – while loop by considering the false condition*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i=7;
        clrscr();
        do
        {
           printf("\n this is a program of do while loop");
           i++;
        }while(i<=5);
        getch();
}
```

Output:

this is a program of do while loop

4. Arrays:

Definition:

An array is a collection of two or more adjacent memory cells, called Array elements, that are associated with a particular symbolic name.

(or)

An array is a fixed – size sequenced collection of elements of the same data type

Declaring an array:

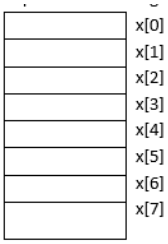
To set up an array in memory, we must declare both the name of the array and the number cells associated with it.

The syntax:

datatype variable_name[size];

Ex: double x[8];

In above ex, the computer reserves eight storage locations as shown below



The values to the array can be assigned as follows

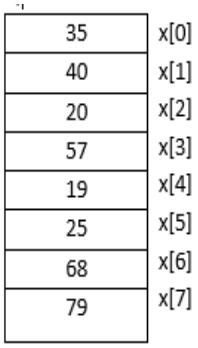
```
x[0]=35;
x[1]=40;
x[2]=20;
x[3]=57;
```

x[4]=19;

x[5]=25;

x[6]=68;

x[7]=79;



To process the data stored in an array, we reference each individual element by specifying the array name and identifying the element desired (for ex, element e of array x). The subscripted variable x[0] (read as x sub zero) may be used to reference initial or 0^{th} element of the array x, x[1] the next element and x[7] the last element. The integer enclosed in brackets is the array subscript and its value must be in the range from **"zero to one less than the number of memory cells in the array"**

Array initialization:

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either the following stages

1. At compile time
2. At run time

1. Compile time initialization:

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

The general form of initialization of array is:

datatype array_name[size]={list of values};

the values in the list are separated by commas. For ex, the statement

int number[3]={0,0,0};

will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized.

The remaining elements will be set to zero automatically.

For instance,

float total[5]={0.0,15.75,-10};

will initialize the first three elements to 0.0,15.75,-10.0 and the remaining two elements to zero.

The size may be omitted. In such case, the compiler allocated enough space for all initialized elements.

For ex, the statement

```
int counter[]={1,1,1,1};
```

will declare the counter array to contain four elements with initial values 1.

Character arrays may be initialized as,

```
char name[]={'J','O','H','N','\0'};
```

If we have more initializes than the declared size, the compiler will produce an error. That is,

```
int number[3]={10,20,30,40};
```

will not work. It is illegal in C.

2. Run time initialization:

An array can be explicitly initialized at run time. This approach is applied for initializing large arrays.

For ex,

```
------------------------------
------------------------------
for(i=0;i<100;i=i+1)
{
    if(i<50)
        sum[i]=0.0;
    else
 sum[i]=1.0;
}
------------------------------
------------------------------
```

The first 50 elements of the array sum are initialized to zero and while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as scanf to initialize an array.

For ex,

```
int x[3];
scanf("%d%d%d",&x[0],&x[1],&x[2]);
```

will initialize array elements with the values entered through the keyboard.

For ex,

```
int x[100];
for(i=0;i<100;i++)
scanf("%d",&x[i]);
```

Array subscripts:

We use subscript,

- to differentiate between the individual array element

- to specify which array element is to be manipulated

Array subscript may be any integer expression. To create a valid reference, the value must be 0 and one less than the declared size. We should understand the distinction between an array subscript values an array element value is essential.

Consider array x below:

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

The subscripted variable x[i] references a particular element of this array

If i is 0,

The subscript value is 0, and x[0] is referenced.

The value x[0] is 16.0

If i is 2,

The subscript value is 2, and x[2] is referenced

The value x[2] is 6.0

If i is 8,

The subscript value is 8, and we cannot predict the value x[8] because the subscript value is out of allowable range

Syntax:

aname[subscript];

ex: b[i+1]

Table: Code fragment that manipulates array x

Statement	Explanation
i=5;	
printf("%d%.1f",4,x[4]);	displays 4 and 2.5 (value of x[4])
printf("%d%.1f",I,x[i]);	displays 5 and 12.0 (value of x[5])
printf("%.1f",x[i]+1);	displays 13.0 (value of x[5] plus 1)
printf("%.1f",x[i]+i);	displays 17.0 (value of x[5] plus 5)
printf("%.1f",x[i+1]);	displays 14.0 (value of x[6])
printf("%.1f",x[i+i]);	invalid. Attempt to display x[10]
printf("%.1f",x[2*i]);	invalid. Attempt to display x[10]
printf("%.1f",x[2*i-3]);	displays -54.5 (value of x[7])
printf("%.1f",x[(int)x[4]]);	displays 6.0 (value of x[2])
printf("%.1f",x[i++]);	displays 12.0 (value of x[5]) then assign 6 to i
printf("%.1f",x[--i]);	displays 5(6-1) to i and then displays 12.0 (value of x[5])
x[i-1]=x[i];	Assigns 12.0 (value of x[5]) to x[4]
x[i]=x[i+1];	Assigns 14.0 (value of x[6]) to x[5]
x[i]-1=x[i]	illegal assignment statement

Processing an Array:

To process the elements of an array in sequence, starting with element zero. For ex, Scanning the data into the array (or) printing its contents, we can accomplish this processing easily using an indexed for loop, a counting loop whose loop control variable runs from zero to one less than the array size. Using the loop counter as an array index(subscript) gives access to each array element.

The number of passes through the loop will therefore equal the number of array elements to be processed.

```
/*program to read the values into array and display the values from the array*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],i;
    clrscr();
    printf("enter the values of array elements\n");
    for(i=0;i<10;i++)
          scanf("%d",&a[i]);
    printf("The values stored in array are:\n";
    for(i=0;i<10;i++)
        printf("%d\n",i,a[i]);
    getch();
}
```

```
/*write a C program to store the squares of the integers 0 through 10 in array (e.g., square[0] is 0,square[1] is 1, square[2] is 4……, square[10] is 100)
#include<stdio.h>
#include<conio.h>
void main()
{
   int square[11],i;
   clrscr();
   for(i=0;i<11;i++)
     square[i]=i*i;
   for(i=0;i<11;i++)
        printf("square[%d]=%d",i,a[i]);
   getch();
}
```

Passing arrays to functions:

We can pass a one dimensional 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 ex, the call

```
largest(a,n);
```

the called function expecting this call must be appropriately defined
the largest function header might look like

float largest(float array[],int size)

it is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. It is not necessary to specify the size of array here.
Let us consider, the problem of finding the largest value in an array of elements. The program is as follows:

```
main()
{
   float largest(float a[],int n);
   float value[4]={2.5,-4.75,1.2,3.67};
   printf("%f\n",largest(value,4));
}
float largest(float a[],int n)
{
      int i;
      float max;
      max=a[0];
      for(i=1;i<n;i++)
      if(max<a[i])
        max=a[i];
      return(max);
   }
```

The name of the array represents the address of its first element. By passing the array name means passing the address of the array to the called function. So, any changes in the array in the called function will be reflected in the original array.

Pass by value: passing the values as parameters to the functions is known as pass by value.

Pass by address: passing address of parameters to the functions is referred to pass by address(or pass by pointers).

Two Dimesnsional and Multi Dimensional arrays:

Two Dimensional Arrays:

Declaring two dimensional array is as follows:

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

two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For ex,

```
int table[2][3]={0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one.

The initialization is done row by row.

The above statement can be equivalently written as

int table[2][]={{0,0,0},{1,1,1}};

by surrounding the elements of the each row by braces.

When the array completely initialized values, explicitly, we need not specify the size of the first dimension. That is the statement

int table[][3]={ {0,0,0},{1,1,1}}; is permitted

In initialization if the values are missing they are automatically set to zero.

Consider,

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

will initialize first two elements of the first row to one, the first element of the second row to two, and all other elements to zero. When all the elements are to be initialized to zero, the following short-cut method may be used.

int m[3][5]={{0},{0},{0}};

Two-dimensional arrays are used to represent tables of data, matrices and other two dimensional objects.

Consider the following table

	Item1	Item2	Item3
Sales Girl #1	310	275	365
Sales Girl#2	210	190	325
Sales Girl#3	405	235	240
Sales Girl#4	260	300	380

It contains total of 12 values, three in each line. It can be considered as a matrix consists of four rows and three columns.

Each row represents values of sales by a particular sales girl and each column represents the item.

	Column 0	Column 1	Column 2
	[0][0]	[0][1]	[0][2]
Row 0--->	310	275	365
	[1][0]	[1][1]	[1][2]
Row 1--->	10	190	325
	[2][0]	[2][1]	[2][2]
Row 2--->	405	235	240
	[3][0]	[3][1]	[3][2]
Row 3--->	310	275	365

Fig. Representation of a two dimensional array in memory.

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 with that row.

/*write a program to illustrate two dimensional array*/

```
#inlcude<stdio.h>
#include<conio.h>
void main()
{
    int a[3][3],i,j;
    clrscr();
    printf("enter the elements of array row wise\n");
    for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      scanf("%d",&a[i][j]);
    printf("the elements of array are:\n");
    for(i=0;i<3;i++)
    {
       for(j=0;j<3;j++)
           printf("a[%d][%d]=%d\n",i,j,a[i][j]);
        printf("\n");
    }
    getch();
}
```

Output:

Enter the elements of array rowwise

1 2 3 4 5 6 7 8 9

The elements of array are:

a[0][0]=1

a[0][1]=2

a[0][2]=3

a[1][0]=4

a[1][1]=5

a[1][2]=6

a[2][0]=7

a[2][1]=8

a[2][2]=9

Like simple arrays, we can also multi-dimensional arrays to functions. The rules are simple,

1. The function must be called by passing only the array name
2. In the function definition, we must indicate that, the array has two dimensions by including two sets of brackets
3. The size of the second dimension must be specified
4. The prototype declaration should be similar to the function header.

The function below calculates the average of the values in a two dimensional matrix

```
double average(int x[][N],int M,int N)
{
                    int i,j;
                    double sum=0.0;
                    for(i=0;i<M;i++)
                      for(j=0;j<N;j++)
                       sum+=x[i][j];
                    return(sum/(M*N));
}
```

Multi Dimensional Arrays:

C allows arrays of three or more dimensions. The exact limit is determined by the compiler.

The general form of multidimensional array is,

type array_name[s1][s2][s3][s4][s5]……[sm];

where si is the size of the i^{th} dimension.

Ex:

```
int survey[3][5][12];   /*three dimensional*/
float table[5][4][5][3];    /*four dimensional*/
```

survey contains 180 integer type elements

table contains 300 elements of float – point type

i) Question and Answers: 2 Marks

1. What is a compiler? Give an example. (Jan – 2020)

Ans: Compiler converts the high level language to machine understandable language(i.e., low level language).

Example: gcc , Microsoft Visual Studio

2. Illustrate the use of break in loop statements with an example. (Jan – 2020, Nov/ Dec – 2019)

Ans: The break is a keyword in C which is used to bring the program control out of the loop.

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
   int i;
   for(i = 0; i<10; i++)
   {
      printf("%d ",i);
```

```
    if(i == 5)
    break;
  }
  printf("came outside of loop i = %d",i);
}
```

3. What is the use of typedef in C? Give an example. (Jan – 2020)

Ans: It allows user to define an identifier that would represent an existing data type. It can later be used to declare variables typedef type identifier; type refers to an existing data type and identifier refers to the new name given to the data type. The new type is new only in name, but not the data type. Typedef cannot create a new type

Ex: typedef int marks;
marks name1,name2;

4. List out the basic data types and their sizes in C. (Dec – 2019, Jun - 2017)

Ans:

Data type	Range of values
char	-128 to 127
int	-32768 to 32767
float	1.2e-38 to 3.4e+38
double	2.3e-308 to 1.7e+308

5. Differentiate between while and do-while loop. (Dec – 2019)

Ans: **While loop** is executed only when given condition is true. Whereas, **do-while loop** is executed for first time irrespective of the condition. After executing **loop** for first time, then condition is checked.

6. What is an enumerated data type? Give an example. (Nov/ Dec – 2019)

Ans: It is 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 of this new type. It is defined as follows:

enum identifier{value1,value2,value3,…….,valuen};

enum identifier v1,v2,v3,… .....,vn;

The variables v1,v2,…….,vn can have one of the values value1,value2,……..

v1=valu3;

v5=value1;

7. Identify and eliminate the error present in the statement: CS > EC ? ANS = CS : ANS = EC(Jun/Jul – 2019, Dec/Jan – 2016).

Ans: ANS = CS > EC ? CS : EC

8. Define an array. Write the formula to find the address of element in 1-D array.

Ans:
An array is a collection of two or more adjacent memory cells, called Array elements, that are associated with a particular symbolic name.
Address of element = base address + subscript * datatype size.

9. List out relational and logical operators.

Ans: Relational operators >, <, ==, >=, <= and !=
Logical operators &&, ||, and !

10. Give the syntax of switch statement. (May/June – 2019)

Ans: Syntax:

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

11. Distinguish between continue and break statements.(Nov/Dec – 2018)

Ans: The major **difference between break** and **continue statements in C** language is that a **break** causes the innermost enclosing **loop** or switch to be exited immediately. Whereas, the **continue statement** causes the next iteration of the enclosing for , while , or do **loop** to begin.

12. Predict the result for the following code const int i = 5; i++; printf("i value = %d",i) (Nov/Dec – 2018)

Ans: Error

13. Write syntax of If-else statements. (Dec – 2016)

Ans:
```
if(condition)            /*no semicolon*/
{
      True – block statement(s)
 }
 else
 {
     False – block statement(s)
 }
 statement x;
```

14. What is the size required to store any array of 25 integers? (Jun - 2016)

Ans: 50 bytes

15. Write a program in C to find the sum of numbers present in an array. (Nov/Dec – 2018)

Ans:
```
#include <conio.h>
 int main()
{
   int a[1000],i,n,sum=0;
   printf("Enter size of the array : ");
   scanf("%d",&n);
   printf("Enter elements in array : ");
   for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }
   for(i=0; i<n; i++)
   {
     sum+=a[i];
   }
    printf("sum of array is : %d",sum);
   return 0;
}
```

ii) MCQs:

1._____ converts the high level language to machine understandable language(i.e., low level language).

a. **Compiler** b. Assembler c. Both A & B d. None

2. A _____ is a data name that may be used to store a data value.

a. Constant b. **Variable** c. Datatype. d. None

3. _____ have fixed meanings and these meaning cannot bechanged.

a. Identifiers b. Variables c. **keywords** d. None

4. _____ allows user to define an identifier that would represent an existing data type.

a. Integer b. Float c. Enum d. **typedef**

5. An_____ indicates an operation to be performed on data that yields a value.

a. Operand b. **Operator** c. Both a & b d. None

6. _____ operators are used to assign the result of an expression to a variable

a. **Assignment** b. Relational c. Logical d. None

7. _____ operators are used for manipulation of data at bit level.

a. Relational b. Arithmetic c. **Bitwise** d. Conditional

8. An _____ is a combination of operators and operands which reduces to a single value.

a. Operators b. Operands c. Precedence **d. Expression**

9. Complex expressions are executed according to _____ of operators.

a. Associativity b. **Precedence** c. Binary d. Ternary

10. The _____ statement is a multi way branch statement.

a. If b. **Switch** c. If…else d. Nested if else

11. A _____ is defined as a block of statements which are repeatedly executed for a certain number of times.

a. Conditional b. Operational c. **Loop** d. None

12. An _____ is a collection of two or more adjacent memory cells, called Array elements, that are associated with a particular symbolic name.

a. Function b. Pointer c. Condition d. **Array**

13. We use _____ to differentiate between the individual array element.

a. Superscript b. **Subscript** c. Both A & B d. None

14. The name of the array represents the _____ of its first element.

a. Subscript b. Value **c. Address** d. None

15. _____-dimensional arrays are used to represent tables of data, matrices.

a. **Two** b. One c. Three d. Four

iii) Question and Answers (10 Marks):

1. **Explain the arithmetic operators in C. Write a C program to test a given number is a positive or negative with and without using conditional operator. (Dec – 2019)**

Ans: Arithmetic operators: - 4M

Operator	Meaning	Example
+	Addition or unary plus	2+2=4
-	Subtraction or unary minus	5-3=2
*	Multiplication	2*5=10
/	Division	10/2=5
%	Modulo division	11%3=2(remainder)

Integer Arithmetic:

When both operands are integers then the operation is called integer arithmetic a = 15/7 = 2.1 = 2

Real arithmetic:

An arithmetic operation involving only real operands is called real arithmetic x = 6.0/7.0 = 0.857143

the operator % cannot be used with real operands Mixed mode arithmetic:

When one operand is real and the other is integer the expression is called a mixed – mode arithmetic expression. If either operand is of real type, then only the real operation is performed and the result is always real number

15/10.0 = 1.5

C program without using conditional operator: 3M

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int num,test;
  clrscr();
  printf("Enter the number:\n");
scanf("%d", &num);
test=(num>0)?1:0;
if(test==1)
```

```
  printf("%d is a positive number\n",num);
 else
   printf("%d is a negative number\n",num)'
 getch();
}
```

C program using conditional operator: 3M

```
#include<stdio.h>
#inclde<conio.h>
void main()
{
  int num;
  clrscr();
  printf("enter the number:\n");
scanf("%d", &num);
if(num>0)
   printf("%d is a positive number\n",num);
else
  printf("%d is a negative number\n",num);
getch();
}
```

2. **Write the syntax of for, while, and do while loops. Write a C program to print the sum of first n natural numbers. (Dec – 2019)**

Ans: **Syntax of for loop: 2M**

```
for(initialization;condition;increment/decrement)/*no semicolon*/
{
Body of the loop
}
Next statement;
```

Syntax of while loop: 2M

```
    initialization;
    while(test condition)
     {
        Body of the loop
        Increment/decrement;
     }
     Statement-x;
```

Syntax of do while loop: 2M

```
initialization;
    do
    {
       Body of the loop;
       increment/decrement;
    }
    while(condition);
    statement-x;
```

C program to print the sum of first n natural numbers: 4M

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int n, sum = 0;
  clrscr();
  printf("enter the number of values:\n");
  scanf("%d",&n);
  for(i=1;i<=n;i++)
    sum=sum+i;
  printf("the sum of first n natural numbers: %d",sum);
  getch();
}
```

3. **Discuss two – dimensional array. Write a C program to calculate the sum of two matrices and obtain the transpose of the resultant matrix. (Dec – 2019)**

Ans: **Two – dimensional array: 3M**

Declaring two dimensional array is as follows:

type array_name[row_size][column_size];

two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For ex,

int table[2][3]={0,0,0,1,1,1};

initializes the elements of the first row to zero and the second row to one.

The initialization is done row by row.

The above statement can be equivalently written as

int table[2][]={{0,0,0},{1,1,1}};

by surrounding the elements of the each row by braces.

When the array completely initialized values, explicitly, we need not specify the size of the first dimension. That is the statement

int table[][3]={ {0,0,0},{1,1,1}}; is permitted

In initialization if the values are missing they are automatically set to zero.

Consider,

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

will initialize first two elements of the first row to one, the first element of the second row to two, and all other elements to zero. When all the elements are to be initialized to zero, the following short-cut method may be used.

int m[3][5]={{0},{0},{0}};

Two-dimensional arrays are used to represent tables of data, matrices and other two dimensional objects.

C program to calculate the sum of two matrices and obtain the transpose of the resultant matrix. 7M

```
#include <stdio.h>
int main()
{
  int r, c, a[100][100], b[100][100], sum[100][100], transpose[100][100],i, j;
  printf("Enter the number of rows (between 1 and 100): ");
  scanf("%d", &r);
  printf("Enter the number of columns (between 1 and 100): ");
  scanf("%d", &c);
  printf("\nEnter elements of 1st matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);
    }
  printf("Enter elements of 2nd matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element b%d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]);
    }
  // adding two matrices
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
    }
  // printing the result
```

```
  printf("\nSum of two matrices: \n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
      if (j == c - 1) {
        printf("\n\n");
      }
    }
// // computing the transpose
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    transpose[j][i] = sum[i][j];
}
// printing the transpose
  printf("\nTranspose of the matrix:\n");
  for (int i = 0; i < c; ++i)
  for (int j = 0; j < r; ++j) {
    printf("%d  ", transpose[i][j]);
    if (j == r - 1)
    printf("\n");
  }
  return 0;
}
```

4. Explain in detail about datatypes in C. (Dec – 2019)

Ans: Data types

C language is rich in its data types. Storage representation and machine instructions to handle constants differ from one machine to machine. C supports three classes of data types

1) Primary (or fundamental) data types
2) Derived data types (arrays, functions, pointers)
3) User – defined Data types

Primary Data types: 5M

C compiler supports five fundamental data types

- integer(int)
- floating point(float)
- character(char)
- double – precision point(double)
- void

and extended data types such as long int, long double

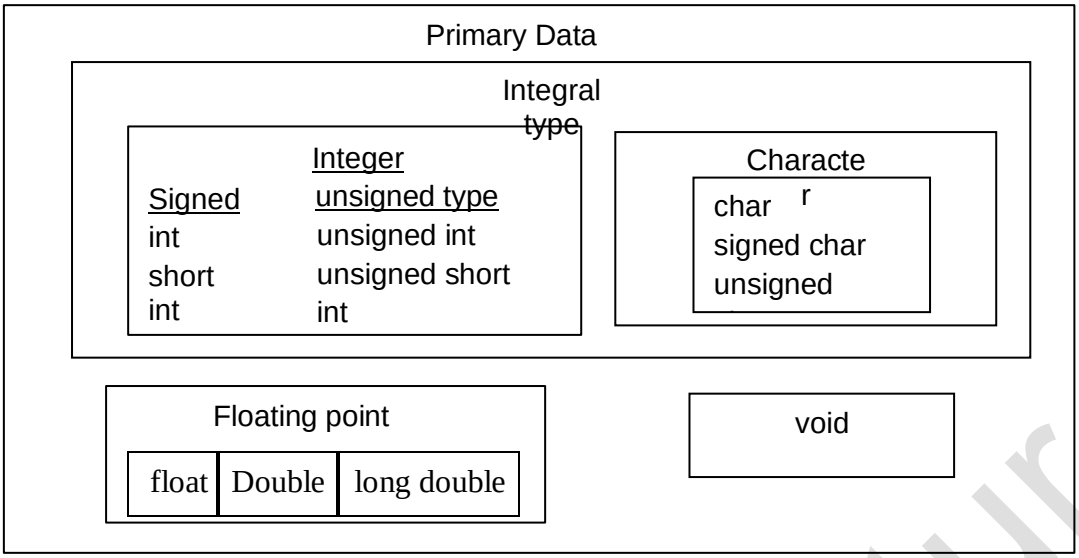


Fig.Primary data types in C

Table: Size and range of basic data types on 16 bit machine

Data type	Range of values
char	-128 to 127
int	-32768 to 32767
float	3.4e-38 to 3.4e+38
double	1.7e-308 to 1.7e+308

Integer types:

- Integers are whole numbers with a range of values supported by a particular machine
- Integer occupy one word of storage
- Word size of machines vary(16 or 32 bits)
- If we use a 16 bit, the range is -32768 to 32767 (that is, -2^{15} to $+2^{15}-1$)
- A signed integer uses one bit for sign and 15 bits for the magnitude of the number
- 32 bit store an integer ranging from 2,147,483,648 to 2,147,483,647
- C has three classes of integer storage
  - short
  - int
  - long int in both signed and unsigned forms
- Short int represents small integer and requires half the amount of storage as a regular int
- Unsigned integers use all the bits for the magnitudes the number and are always positive then in 16 bit the range is 0 to 65535
- Default declaration is a signed number

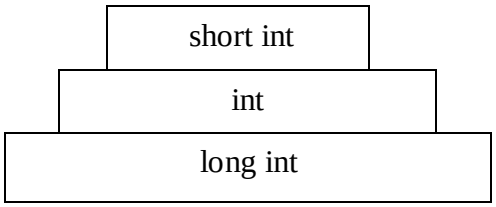


Fig. Integer types organizes from the smallest to the largest

Table: size and range of data types on a 16 – bit machine

Type	Size		Range
	Bit	Bytes	
char or signed char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int or signed int	16	2	-32768 to 32767
unsigned int	16	2	0 to 65535
short int or signed short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int or signed long int	32	4	-2,147,483,648 to 2,147,483,647
unsigned long int	32	4	0 to 4,294,967,295
Float	32	4	1.2E-38 to 3.4E+38
Double	64	8	2.3E-308 to 1.7E+308
Long double	80	10	3.4E-4932 to 1.1E+4932

Floating point types:

- Floating point(or real) numbers are stored in 32 bits (on all 16 or 32 bit machines), with 6 digits precision
- Floating point number are defined in C by the keyword **float**
- When the accuracy provided by float is not sufficient the type double used
- A double data type uses 64 bits giving precision of 14 digits
- These are known as double precision number
- To extend the precision further, we may use long double which uses 80 bits

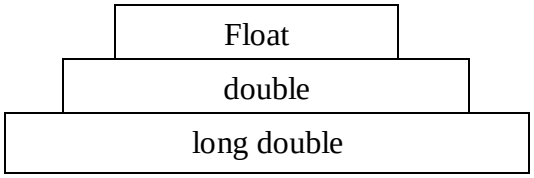


Fig. floating point types

Void type:

- The void type has no values
- It is usually used to specify the type of functions
- The type of a function is said to be void when it does not return

any value to the calling function

Character types:

- A single character can be defined as a character(char) type data
- Characters are usually stored in 8 bits (one byte) of internal storage
- Unsigned chars have values between 0 and 255, signed chars have values from

  -128 to 127

Table: Data types and their keywords

Data type	Keyword equivalent
character	char
unsigned character	unsigned char
signed character	signed char
signed integer	signed int (or int)
signed short integer	signed short int (or short int or short)
signed long integer	signed long int (or long int or long)
unsigned integer	unsigned int (or unsigned)
unsigned short integer	unsigned short int (or unsigned short)
unsigned long integer	unsigned long int (or unsigned long)
floating point	float
double – precision floating point	double extended double – precision floating point long double

User defined type declaration: 5M

There are two types of user – defined data types

- Type definition
- Enumerated

Type definition:

- It allows user to define an identifier that would represent an existing data type
- It can later be used to declare variables

  typedef type identifier;

  type refers to an existing data type and identifier refers to the new name given to the data type
- The new type is new only in name, but not the data type

- Typedef cannot create a new type
- Ex: typedef int marks;
  marks name1,name2;
- By this it increases the readability of the program

Enumerated data type:

- It is 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 of this new type
- It is defined as follows:
  enum identifier{value1,value2,value3,…….,valuen};
  enum identifier v1,v2,v3,… ,vn;

- The variables v1,v2,…….,vn can have one of the values value1,value2,…….. v1=value3; v5=value1;
- An ex: enum day{mon,tue,wed,thu,fri,sat,sun};
  enum day week_st,week_end;
  week_st=mon;
  week_end=fri;
  if(week_st==tue)
  week_end=sat;
- The compiler automatically assigns integer digits with 0 to all enum constants
- That is enum constant value1 is assigned 0, value2 is assigned 1 and so on
- The automatic assignments can be overrided by explicit values
  enum day{mon=1,tue,… , sun};
  here next constants increase successively by 1
- The definition and declaration can be combined in one statement as Ex: enum day{mon,…,sun} week_st,week_end;

5. Write a C Program for calculating the sum of 'N' odd numbers using for loop. (Dec – 2019) 10M

Ans: C Program for calculating the sum of 'N' odd numbers using for loop:

```
#include <stdio.h>
int main()
{
   int i, n, sum=0;
   /* Input range to find sum of odd numbers */
   printf("Enter upper limit: ");
   scanf("%d", &n);
   /* Find the sum of all odd number */
   for(i=1; i<=n; i+=2)
```

```
    {
       sum += i;
    }
    printf("Sum of odd numbers = %d", sum);
    return 0;
}
```

6. Explain the syntax and use of switch statement with suitable example. (Dec – 2019)

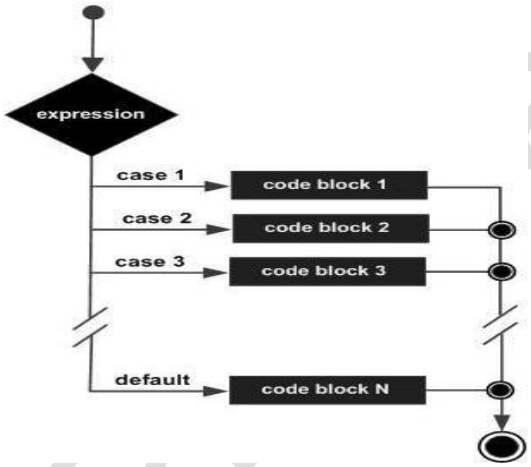
Ans: Switch: 5M

- The switch statement is a multi way branch statement
- We can use an if statement to control the selection, but when the number of alternatives increases then the complexity of program increases
- The switch statement tests the value of a given variable(or expression) against list of case values and when a match is found, a block of statements associated with the case is executed
- The general form of switch statement is as shown below:
- Syntax:

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

  - The expression is an integer expression or characters
  - value-1, value-2, ... are constants or constant expressions(evaluable to an integer constant) and are known as **case labels**
  - The case labels end with **:** and each of these values should be unique within the switch statement
  - block – 1, block – 2,……. Are statement lists and may contain zero or more statements and there is no need to put braces around these blocks
  - when the switch is executed, the value of expression is successfully compared against the values value-1, value-2,…….

- If a case found whose value matches with the value of the expression, then the block of statements that follows the case are executed
- The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch
- The default is an optional case, will be executed if the value of the expression does not match with the any of the case values
- If the default statement is not present, no action takes place if all matches fail and control goes to the statement-x
- The selection process of switch statement is illustrated in the flowchart as shown in fig.



/*program to convert years into 1. Minutes 2. Hours 3. Days 4. Months 5. Seconds using switch() statement*/ 5M

```
#include<stdio.h>
#include<conio.h>
void main()
{
  long
  ch,min,hrs,ds,mon
  ,yrs,se; clrscr();
  printf("\n[1] MINUTES");
  printf("\n[2] HOURS");
  printf("\n[3] DAYS");
  printf("\n[4] MONTHS");
  printf("\n[5] SECONDS");
  printf("\n[0] EXIT");
  printf("\n enter your choice"); scanf("%ld",&ch);
  if(ch>0&&ch<6)
  {
```

```
        printf("enter years:"); scanf("%ld",&years);
      }
      mon=yrs*12;
      ds=mon*30;
      ds=ds+yrs*5;
      hrs=ds*24;
      min=hrs*60;
      se=min*60;
      switch(ch)
      {
        case 1: printf("\n Minutes: %ld",min);
                break;
        case 2: printf("\n Hours: %ld",hrs);
                break;
        case 3: printf("\n Days: %ld",ds);
                break;
        case 4: printf("\n Months: %ld",mon);
                break;
        case 5: printf("\n seconds: %ld",se);
                break;
        case 0: printf("\n Terminated choice"):
      default: printf("\n Invalid choice");
      }
      getch();
  }
```

Output:

[1] MINUTES
[2] HOURS
[3] DAYS
[4] MONTHS
[5] SECONDS
[0] EXIT
enter your choice: 4
enter years:2
Months: 24

Unit – 2

Functions, types of functions, Recursion and argument passing, pointers, storage allocation, pointers to functions, expressions involving pointers, Storage classes – auto, register, static, extern, Structures, Unions, Strings, string handling functions, and Command line arguments.

2.1. Functions:

- A function is a self contained block of code that performs a particular task

2.2. Types of functions:

- C functions can be classified in two categories, namely, library functions and user – defined functions
- main is an example of user defined functions
- printf and scanf belong to category of library functions

Library functions(or predefined):

- A primary goal of software engineering is to write error free code
- One way to reach this goal is reusing program fragments that have already been written and tested
- C promotes reuse by providing many predefined functions that can be used to perform mathematical computations
- C's standard math library defines a function named sqrt that perfornes square root Computation

1. Defining a function

- A function definition, also known as "function implementation" shall include the following elements
    1. Function name
    2. Return type
    3. List of parameters
    4. Local variable declaration
    5. Function statement
    6. A return statement
- All the six elements are grouped into two parts, namely,
    a. Function header(first three elements); and
    b. Function body(second three elements)
    - Function header: The function header consists of three parts, return type, function name, formal parameter list
    - Note that the a semicolon is not used at the end of the function header

- Name and type:
  - The return type specifies the type of value that the function is expected to return to the program calling the function
  - If the return type is not explicitly specified , C will assume that is an integer type
  - If the function is not returning any thing, then we need to specify the return type as void
  - The function name is any valid C identifier
  - Care must be taken to avoid duplicating library routine names or operating system commands
- Formal parameter list:
  - The parameter list declares the variables that will receive the data sent by the calling program
  - They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as formal parameters
  - The parameters are also known as arguments

  Syntax:

  returntype functionname(argument list) /*no semicolon*/

  {

  Local declarations ; Executable statements; return statement;

  }

- Function body:
  - The function body contains declaration contains declarations and statements
  - The body enclosed in braces, contains three parts, in the order given below
    1. Local declarations that specify variables needed by the function
    2. Function statements that perform the task of the function
    3. A return statement that returns the value evaluated by the function
  - For a function with void return type, we can omit the return statement

2. Accessing a function:

- A function can be accessed by making a call to the particular function as, Syntax: functionname(list of arguments);
- Note that the function call will be terminated by semicolon
- The function that has function call statement is called as calling function and the function definition which was called by function call is known as called function
- The function call will have the actual parameters(or arguments) which is passed to called function
- List of argument are given with their data type and separated by comma

3. Function Prototype(or function declaration):

- Like variable, all functions in a C program must be declared, before they are invoked
- A function declaration (also known as function prototype) consists of four parts
  - Return type(or function type)
  - Function name
  - Parameter list
  - Terminating semicolon

Syntax:

returntype functionname(parameter list);

Ex:

void drawcircle(void);

- A prototype declaration may be placed in two places in a program
  a. Above all the functions(includein main)
  b. Inside a function definition
- Global Prototype: when we place the declaration above all the functions(in the global declaration section), the prototype is referred to as a global prototype
- Local prototype: when we place the declaration in function definition(in the local declaration section), the prototype is called a local prototype
- Global prototypes are available for all functions in the program
- Local prototype are primarily used by the functions containing them
- The place of declaration of function defines a region in a program in which the function may be used by other functions. This region is known as the 'scope' of the function

2.3. Recursion:

- Recursion means, where a function calls itself
- Ex: main()
  {
  printf("this is an example of recursion");
  main();
  }
- When executed, this program will produce an output something like this this is an example of recursion
  this is an example of recursion this is an example of recursion this is an example of recursion
  …….
- Execution is terminated abruptly; otherwise the execution will continue indefinitely

2.4. Argument passing:

- There are two types of parameter passing mechanisms: call – by – value and call – by - reference

- Call by value and call by reference (also known as pass-by-value and pass-by-reference).
- These methods are different ways of passing (or calling) data to functions.
- **Call by Value**
  - If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function. Let's take a look at a call by value example:

```
#include <stdio.h>
void call_by_value(int x)
{
printf("Inside call_by_value x = %d before adding 10.\n", x);
x += 10;
printf("Inside call_by_value x = %d after adding 10.\n", x);
}
int main()
{
 int a=10;
printf("a = %d before function call_by_value.\n", a);
call_by_value(a);
printf("a = %d after function call_by_value.\n", a);
return 0;
}
```

  - The output of this call by value code example will look like this:
  - a = 10 before function call_by_value.
  - Inside call_by_value x = 10 before adding 10.
  - Inside call_by_value x = 20 after adding 10.
  - a = 10 after function call_by_value.

Ok, let's take a look at what is happening in this call-by-value source code example. In the main() we create a integer that has the value of 10. We print some information at every stage, beginning by printing our variable a. Then function call_by_value is called and we input the variable a. This variable (a) is then copied to the function variable x. In the function we add 10 to x (and also call some print statements). Then when the next statement is called in main() the value of variable a is printed. We can see that the value of variable a isn't changed by the call of the function call_by_value().

- **Call by Reference**
  - If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example:

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

```
#include<conio.h>
void call_by_reference(int *y)
{
printf("Inside call_by_reference y = %d before adding 10.\n", *y);
(*y) += 10;
printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
 int main()
{
 int b=10;
 clrscr();
printf("b = %d before function call_by_reference.\n", b);
call_by_reference(&b);
printf("b = %d after function call_by_reference.\n", b);
return 0;
}
```

The output of this call by reference source code example will look like this:

- b = 10 before function call_by_reference.
- Inside call_by_reference y = 10 before adding 10.
- Inside call_by_reference y = 20 after adding 10.
- b = 20 after function call_by_reference.

Let's explain what is happening in this source code example. We start with an integer b that has the value 10. The function call_by_reference() is called and the address of the variable b is passed to this function. Inside the function there is some before and after print statement done and there is 10 added to the value at the memory pointed by y. Therefore at the end of the function the value is 20. Then in main() we again print the variable b and as you can see the value is changed (as expected) to 20.

- **When to Use Call by Value and When to use Call by Reference?**
  - One advantage of the call by reference method is that it is using pointers, so there is no doubling of the memory used by the variables (as with the copy of the call by value method). This is of course great, lowering the memory footprint is always a good thing. So why don't we just make all the parameters call by reference?
  - There are two reasons why this is not a good idea and that you (the programmer) need to choose between call by value and call by reference. The reason is: side effects and privacy. Unwanted side effects are usually caused by inadvertently changes that are made to a call by reference

2.5. POINTERS

Definition: A pointer variable is a memory variable that can store the address of another variable declared for the same type. A variable is identified as pointer if it is declared for its type and is preceded with the indirection operator (*****).

Features of Pointers:

- Pointer saves the memory space.
- Execution time with pointer is faster because data is manipulated with the address i.e., direct access to memory location.
- The memory is accessed efficiently with the pointers
- The pointer assigns the memory space and it also releases
- Dynamic memory allocation and de-allocation is possible
- Pointers are used with data structures.
- They are useful for representing two-dimensional and multi-dimensional arrays

Pointer Declaration:

The general form for pointer declaration is given below:

Data type *variable;

Where data type can be any fundamental data type or user defined data type and the variable name should follow the rules of identifier and ***** is called **the indirection operator or dereferencing operator.**

For eg. int *x;
float *y;
char *z;

1. In the above example x is declared as integer pointer and it can store only the address of another variable declared of type int.

In the above example y is declared as float pointer and it can store only the address of another variable declared of type float.

In the above example z is declared as character pointer and it can store only the address of another variable declared of type char.

2. The indirection operator (*) is also called dereference operator when a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

3. Normal variables provide direct access to their own values and a pointer provides indirect access to the values of the variable whose address it stores.

4. The indirection operator(*) is used in two different ways with pointers,

a) Declaration
b) Dereference

5. When pointer is declared, the star indicates that it is a pointer, not a normal variable.

6. When the pointer is dereferenced, the indirection operator indicates that the value at that memory location stored in the pointer is to be accessed rather that the address itself.

Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as initialization. The general form for assigning the address of a variable to a pointer variable is given below:

Pointervariable=**&**variable;

Where pointer variable and variable should be declared of the same type and **& is the reference operator.**

Ex: int quantity;
int *p;
p=&quantity;

we can also combine like this,
int *p=&quantity;

the only requirement is the variable quantity must be declared before initialization.

Remember, this is initialization of p and not *p. The & is the address operator and it represents the address of the variable.The %u is used with printf() function for printing the address of a variable.

Note: Pointer variable declared of any type occupies the same memory space.

```
/* sample program showing the usage of pointer declaration, assigning of address to pointer variable  and also printing the address of variables using pointers */
#include <stdio.h>
main()
{
/* pointer variable and variable declaration */
int *p,q;
char *r,s;
float *z,y;
/* variables intialised with values */
q=12;
s='a';
y=23.45;
/* assigning the address of the variables to the pointer variables */
p=&q;
r=&s;
z=&y;
/* printing the address of the variables using pointers */
printf("address of the variable q = %u\n",p);
printf("address of the variable s=%u\n",r);
printf("address of the variable y=%u\n",z);
/* printing the address of the variables without using pointers */
printf("address of the variable q = %u\n",&q);
printf("address of the variable s=%u\n",&s);
printf("address of the variable y=%u\n",&y);
}
```

```
/* to illustrate the use of indirection operator '*' to access the value pointed to by a pointer */
#include <stdio.h>
#include <conio.h>
void main()
```

```
{
/* pointer variable and variable declaration */
int x,y;
int *ptr;
clrscr();
x=10;
ptr=&x;
y=*ptr;
printf("value of x is %d\n\n",x);
printf("%d is stored at addr %u\n",x,&x);
printf("%d is stored at addr %u\n",*&x,&x);
printf("%d is stored at addr %u\n",*ptr,ptr);
printf("%d is stored at addr %u\n",ptr,&ptr);
printf("%d is stored at addr %u\n",y,&y);
*ptr=25;
printf("\n Now=x=%d\n",x);
getch();
}
```

Output:

```
Value of x is 10
10 is stored at addr 4104
10 is stored at addr 4104
10 is stored at addr 4104
4104 is stored at addr 4106
10 is stored at addr 4108
Now x=25
```

Note: x=*(&x)=*ptr=y

&x=&*ptr

Pointers and One dimensional Array:

Array name itself is an address or pointer. It points to the address of the first element of the array.

The elements of the array together with their addresses can ve displayed by using array name itself. Array elements are always stored in contiguous memory locations. To understand this much better below we will see an example illustrating this.

```
/* program to display the elements of the array using array as pointer and also to print the addresses of these locations. */
#include <stdio.h>
main()
{
int num[5]={20,30,40,50,60};
int i;
i=0;
while(i<5)
```

{
printf("element=%3d address=%8u\n',*(num+i),num+i);
i++;
}}

Explanation:

If we suppose assume that the memory is allocated for the above declaration as shown below:

Array name with index	num[0]	num[1]	num[2]	num[3]	num[4]
Value	20	30	40	50	60
Address	3000	3002	3004	3006	3008

then in the while loop when i =0, the condition becomes true and when the printf statement given below gets executed

printf("element=%3d address=%8u\n',*(num+i),num+i);

then the array name num will be initialized with the **starting address** or **base address** or **address of the first element of the array** then

***(num+i) will be interpreted as *(3000+0) which is further interpreted as *(3000+0*no. of bytes allocated for that data type) which is finally interpreted as *(3000) which is value at address i.e., 20**

and

(num+i) will be interpreted as (3000+0) which is further interpreted as (3000+0*no. of bytes allocated for that data type) which is finally interpreted as (3000) which is the address i.e., 3000

likewise when i get incremented with 1 then

***(num+i) will be interpreted as *(3000+1) which is further interpreted as *(3000+1*no. of bytes allocated for that data type) which is finally interpreted as *(3002) which is value at address i.e., 30**

and

(num+i) will be interpreted as (3000+1) which is further interpreted as (3000+1*no. of bytes allocated for that data type) which is finally interpreted as (3002) which is the address i.e., 3002

and so on.

```
/* program to copy the elements of the array to another array using pointer */
#include <stdio.h>
main()
{
int num[5]={20,30,40,50,60},dnum[5],*aptr;
int i;
/* assigning the address of the array to the pointer variable */
aptr=num;  /* starting address or address of the first element of the array num is assigned to the pointer variable aptr*/
i=0;
```

```
while(i<5)
{
dnum[i]=*aptr;  /* value at address is assigned to the array dnum */
aptr++; /* pointer arithmetic operation is performed */
i++;
}
}
```

Explanation:

If we suppose assume that the memory is allocated for the above declaration as shown below:

Array name with index	num[0]	num[1]	num[2]	num[3]	num[4]
Value	20	30	40	50	60
Address	3000	3002	3004	3006	3008

then in the while loop when i =0, the condition becomes true and when the statement given below gets executed

dnum[i]=*aptr;

then the pointer variable aptr which is holding the starting address or base address or address of the first element of the array i.e.,

*(aptr) will be interpreted as *(3000+0) which is further interpreted as

*(3000+0*no. of bytes allocated for that data type) which is finally interpreted as *(3000) which is value at address i.e., 20 which is initialized to the array dnum with index 0.

likewise when i gets incremented with 1 and aptr gets incremented by 1 then

aptr++ interpreted as aptr=aptr+1, as aptr holds the starting address or address of first element of the array that value is susbstitute and is further interpreted as

(3000+1*no. of bytes allocated for that data type) which is finally interpreted as (3002) which is assigned to aptr and in the next iteration the condition becomes true and at that time aptr holds the address 3002 and *aptr will be interpreted as *(3002) and assigns the value at address to the array dnum with index 1

and so on.

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

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements are stored as follows:

Elements	value	address
x[0]	1	1000
x[1]	2	1002
x[2]	3	1004
x[3]	4	1006
x[4]	5	1008

x=&x[0]=1000

Consider, p as an integer pointer,

```
p=x;
```

equivalent to p=&x[0];

Now, can access every value of x using p++ relationship between p and x is shown as:

p=&x[0](=1000)
p+1=&x[1](=1002)
p+2=&x[2](=1004)
p+3=&x[3](=1006)
p+4=&x[4](=1008)

ex:

address of x[3]=baseaddress+(subscript*scalefactorofdatatype)

ex: address of x[3]=baseaddress+(3*scalefactorofint)

=1000+(3*2)=1006

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

Pointers and Two dimensional arrays:

Array name itself is an address or pointer. It points to the address of the first element of the array.

To understand this much better below we will see an example illustrating this.

```
/* program to display the elements of the array using array as pointer and just as an array */
#include <stdio.h>
main()
{
int num[3][3]={2,4,3,6,8,5,3,5,1};
int i,j;
for(i=0;i<2;i++)
for(j=0;j<2;j++)
{
printf("element=%3d element=%3d\n',*(*(num+i)+j),num[i][j]);
i++;
}
}
```

Explanation:

If we suppose assume that the memory is allocated for the above declaration as shown below:

Array name with index	num [0][0]	num [0][1]	num [0][2]	num [1][0]	num [1][1]	num [1][2]	num [2][0]	num [2][1]	num [2][2]
Value	2	4	3	6	8	5	3	5	1
Address	3000	3002	3004	3006	3008	3010	3012	3014	3016

printf("element=%3d element=%3d\n',*(*(num+i)+j),num[i][j]);

In the above statement ,*(*(num+i)+j) is interpreted as shown below:

Imagine 2-D array as collection of several 1-D arrays. The only thing that the compiler needs to remember about 1-D array is its base address.

According to that the base address of the 1-D arrays are stored in num[0],num[1] and num[2].

If suppose that num[1] gives the base address of the second array then num[1]+2 gives the address of the third element in that array. In this case it is the address 3010.
The value at this address can be obtained through the expression *(num[1]+2). We already know that num[1] can be expressed as *(num +1). So, the above expression becomes *(*(num+1)+2) which is the same as num[1][2].

The base address of array a is &a[0][0] 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. Suppose we declare an array a as follows:

int a[3][4]={{15,27,11,35},{22,19,31,17},{31,23,14,36}};

the elements of a row will be stored as:

----------row 0---------- ----------row1---------- ----------row2----------

15	27	11	35	22	19	31	17	31	23	14	36

If we declare p as an int pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+4*i+j)

consider

a[2][3] is given by *(p+284+3)=*(p+11)

Operations on pointers:

Arithmetic operations with pointers:

Arithmetic operations on pointer variables are also possible. Increase, decrease, prefix and postfix operations can be performed with help of pointers. The effect of these operations are shown in the below given table:

Initial Data type		Address	After operation		Address Operations		Required Bytes
Int	i=2	4046	++	--	4048	4044	2
Char	c='x'	4053	++	--	4054	4053	1
Float	f=2.2	4058	++	--	4062	4054	4
Long	I=2	4060	++	--	4064	4056	4

Below is given a simple example illustrating the usage of pointers in performing arithmetic operations:

```
/* sample program to illustrate the use of pointers in performing arithmetic operations */
#include <stdio.h>
main()
{
int a,b,res;
int *c,*d;
clrscr();
printf("enter values for a and b:");
scanf("%d %d",&a,&b);
c=&a;  /* address of a assigned to c */
d=&b; /* address of b assigned to d */
```

```
res=*c + *d;  /* pointer variable used with indirection operator that gives value at address*/
printf("sum of a and b is %d\n",res);
}
```

Below is given a table that describes the contents of the variables for the above program(Values and addresses of locations are assumed)

Name of the variable	A	b	c(address)	*c(value at address)	d	*d(value at address)
Contents of the variable	10	20	2050	10	3050	20
Address of the variable	2050	3050	2666		3666	

Explanation: Value is read into the variable a and b through scanf statement and the address of the variables are assigned to the pointer variables with the statements

c=&a;

d=&b;

The statement res=*c + *d , *c gives the value of the addressed location a and *d gives the value of the addressed location b and these two values are added and the result is stored in the variable res, which is then printed.

Note: if the pointer variable is preceeded with indirection operator and is used in any part of the program other than the declaration, then it gives the value of the variable whose address is assigned to this pointer variable.

If the pointer variable is not preceeded with indirection operator and is used in any part of the program other than the declaration, then it gives the address of the variable whose address is assigned to the pointer variable.

Arithmetic operations on pointers without using indirection operator:

Below is given a simple example illustrating the usage of pointers in performing arithmetic operations:

Note: For pointer arithmetic, arithmetic operators * and / cannot be used.

```
/* sample program to illustrate the use of pointers in performing arithmetic operations */
#include <stdio.h>
main()
{
int a,b;
int *c,*d;
clrscr();
a=5;
b=6;
c=&a; /* address of a assigned to c */
d=&b; /* address of b assigned to d */
c=c+2; /* use of pointer variable with no indirection operator */
```

```
printf("value of c=%u\n",c);
printf("value of d=%u\n",d);
}
```

Below is given a table that describes the contents of the variables for the above program(Values and addresses of locations are assumed)

Name of the variable	A	b	c(address)	D	c=c+2
Contents of the variable	5	6	2050	3050	2054
Address of the variable	2050	3050	2666	3666	2666

Explanation:

In the arithmetic statement c=c+2, where pointer variable is used without indirection operator then the statement computation is done as shown below:

c=c+2*No. of bytes allocated for that data type

as the pointer variable c holds the address of the variable a i.e., 2050 on sUbStitutine this in the above statement, we get

c=2050+2*2

c=2054, similarly we form for the next statement and the result is shown in the table.

Array of pointers:

Array of pointers is nothing but collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. Below is given an example showing how addresses are stored in array of pointers.

```
/* program to store the addresses of the isolated variables in array of pointers*/
#include <stdio.h>
main()
{
int *parr[3];
int i=10,j=20,k=30,m;
parr[0]=&i;
parr[1]=&j;
parr[2]=&k;
printf("elements are:\n");
for(m=0;m<3;m++)
printf("%3d\n",*(parr[m]);
}
```

Explanation:

If we suppose assume that the memory is allocated for the above declaration as shown below:

Array name with index	i	j	k	parr [0]	parr [1]	parr [2]
Value	10	20	30	3000	3002	3004
Address	3000	3002	3004	3006	3008	3010

parr[0] will hold the address of the variable i and parr[1] will hold the address of the variable j and parr[2] will hold the address of the variable k. and the expression *(parr+0) will give the value at that address.

Pointers to pointers:

Pointer variable holding the address of another pointer variable is called as pointer to pointer. Below is given a program that illustrates the concept of pointer to pointer.

```
/* program to illustrate the usage of pointer to pointer*/
#include <stdio.h>
main()
{
int **p; /* pointer to pointer declaration */
int *q; /* pointer declaration */
int a=2; /* variable declaration */
q=&a;
p=&q;
printf(value of a = %d %d %d\n",a,*q,**p);
}
```

Explanation:

If we suppose assume that the memory is allocated for the above declaration as shown below:

Array name with index	a	p	q
Value	2	3004	3000
Address	3000	3002	3004

In the printf statement **p gives the value of the variable a as p holds the address of q and q holds the address of a.

2. 6. Storage allocation:

The C language supports two kinds of memory allocation through the variables in C programs:

- *Static allocation* is what happens when you declare a static or global variable. Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is started (part of the exec operation), and is never freed.
- *Dynamic memory allocation* is a technique in which programs determine as they are running where to store some information. You need dynamic allocation when the

amount of memory you need, or how long you continue to need it, depends on factors that are not known before the program runs. To dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc()

malloc():

- The name "malloc" stands for memory allocation.
- The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

- ptr = (castType*) malloc(size);

Example

- ptr = (float*) malloc(100 * sizeof(float));

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float.

realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

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

Here, ptr is reallocated with a new size x.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

2.7. Pointers to function:

A function, like a variable, has a type and all address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

type (*fptr)();

This tells the compiler that fptr is a pointer to a function, which returns type value. The parentheses around *fptr are necessary.

Consider the statement,

type *gptr();

would declare gptr as a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

For ex,

```
double mul(int,int);
double (*p1)();
p1=mul;
```

to call the function mul, we may now use the pointer p1 with the list of parameters, that is

(*p1)(x,y)/*function call*/

is equivalent to

mul(x,y)

note the parentheses around *p1

```
/* program to display address of user defined function*/
 #include <stdio.h>
#include <conio.h>
void main()
{
int show();
clrscr();
show();
printf("%u",show);
getch();
}
show()
{
printf("n Adress of function show() is:");
}
```

Output:

Address of function show() is: 530

```
/*Program to call function – using pointer*/
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
int show(); /*function prototype*/
int (*p)(); /*pointer declaration*/
p=show; /*Assign address of show() to p*/
(*p)(); /*Function call using pointer*/
printf("%u",show); /*Displays address of function*/
getch();
}
show()
{
clrscr();
printf("\nAddress of function show() is:");
}
```

Output:

```
Address of function show() is: 531
/*Program to display the address of library functions*/
#include<stdio.h>
#include<conio.h>
void main()
{
 void (*p)();
 p=clrscr()'
 (*p)(); /*clears the screen*/
 printf("\nAddress of printf() is %u",printf);
 printf("\nAddress of scanf() is %u",scanf);
 printf("\n Address of clrscr() is %u",clrscr);
 getch();
}
```

Output:

```
Address of printf() is 3518
Address of scnaf() is 5726
Address of clrscr() is 6035
```

2.8. Expressions involving pointers:

Write a c program to illustrate pointer expressions.

```
#include <stdio.h>
int main()
{
   int a = 20, b = 10;
   int add, sub, div, mul, mod;
   int *ptr_a, *ptr_b;
   ptr_a = &a;
   ptr_b = &b;
   add = *ptr_a + *ptr_b;
```

```
    sub = *ptr_a - *ptr_b;
    mul = *ptr_a * *ptr_b;
    div = *ptr_a / *ptr_b;
    mod = *ptr_a % *ptr_b;
    printf("Addition = %d\n", add);
    printf("Subtraction = %d\n", sub);
    printf("Multiplication = %d\n", mul);
    printf("Division = %d\n", div);
    printf("Modulo = %d\n", mod);
    return 0;
}
```

2.9. Storage Classes:

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a

register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>
 /* function declaration */
void func(void);
static int count = 5; /* global variable */
main() {
   while(count--) {
      func();
   }
   return 0;
}
/* function definition */
void func( void ) {
   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined

variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
  count = 5;
  write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
  printf("count is %d\n", count);
}
```

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c.

2.10. STRUCTURES

Definition: Structure is a very useful derived/user defined data type supported in c that allows grouping one or more variables of different data types with a single name.

Or

A structure is a collection of one or more variable of different data types, grouped together under a single name.

By using structures we can make a group of variables, arrays, pointers etc

Features of Structures:

1. To copy elements of one array to another array of same data type elements are copied one by one. It is not possible to copy all the elements at a time. Whereas in structure it is possible to copy the contents of all structure variable of its type using assignment (=) operator. It is possible because the structure elements are stored in successive memory locations.
2. Nesting of structures is possible i.e., one can create structure within the structure. Using this feature one can handle complex data types.

3. It is also possible to pass structure elements to a function. This is similar to passing an ordinary variable to a function one can pass individual structure by value or address.
4. It is also possible to create structure pointers. In the pointers we have seen pointing a pointer to an integer, pointing to float, pointing to char.

In a similar we can create a pointer pointing to structure elements.

For this it requires -> operators .

Declaration and Initialization of structures:

Structures must be defined first for their format that may be used later to declare structure variable.

The general syntax of structure is given below:

```
Struct <tagname>
{
   datatype membername1;
   datatype membername2;
};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside.
3. The tag name such as student can be used to declare structure variables of its type, later in the program.

Declaring structure variables:

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword struct
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

Structure can be declared and initialized either above the main function or within the main function. The process of defining, declaring and initializing of structure is illustrated below:

Above main function	**Within main function**

struct student
{
char name[20],course[5];
int rollno;
}s1={“ARYA”,”BTECH”,123};
 	
Or
struct student
{
char name[20],course[5];
	main()
	{
	struct student
	{
	char name[20],course[5];
	int rollno;
	} s1={“ARYA”,”BTECH”,123};
	}
	Or
	main()

int rollno;
}s1;
struct student s1={“ARYA”,”BTECH”,123};
 	
main()
{
 	
}
Explanation:
In the above example student is the structure
name and s1 is the structure variable declared
and the values are initialized to the structure
variable within flower braces
	{
	struct student
	{
	char name[20],course[5];
	int rollno;
	} s1;
	struct student s1={“ARYA”,”BTECH”,123};
	}
	Explanation:
	In the above example student is the structure
	Name and s1 is the structure variable declared and the values are initialized to the structure an variable within flower braces

The use of tag name is optional here for ex:

```
struct
{
 -----
 -----
}book1,book2,book3;
```

declares book1, book2, book3 as structure variable representing three books, but does not include a tag name.

However, this approach is not recommended for two reasons,

1. Without tag name, we cannot use it for future declarations
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the main, along cases, the definition is global and can be used by other functions as well.

Accessing members of the Structure:

The member’s of the structure can be accessed with a variable declared of that structure type and with dot operator(.)

The general form is shown below:

Structurevariablename.membername

For eg.,

```
struct student
{
  char name[20],course[5];
  int rollno;
}s1;
main()
```

```
{
printf("enter the name,course and rollno:");
scanf("%s%s%d",s1.name,s1.course,&s1.rollno);
printf("name=%s\n",s1.name);
printf("course=%s\n",s1.course);
printf("rollno=%d\n",s1.rollno);
}
```

Explanation:

In the above example a **structure by name student** is declared with **members name and course declared as character array and rollno declared of typed as int.**

The scanf statement shows how data is read for the members of the structure through structure variable likewise the printf statement shows how to print the data of the members of the structure.

Here is how we would assign values to the members of book1:

```
strcpy(book1.title,"BASIC");
strcpy(book1.author,"balaguru");
book1.pages=250;
book1.price=120.50;
```

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

```
scanf("%s",book1.title);
scanf("%d",&book1.pages);
```

are valid input statements.

Nested Structures:

Structure defined within another structure is said to be nested structures. Below are given examples of the different ways of making a structure nested.

struct employee
{
int empid;
char designation[15];
struct salary
{
int basic;
int da;
}s1;
}emp;
	Struct date
	{
	int day;
	int month;
	int year;
	};
	struct employee
	{
	int empnum;
	char name[20];
	float basic;
	Struct date joindate;
	}emp1;

In case of nested structures the accessing of data for members is91done **by91accessing the outer structure variable followed by the dot operator and then the inner structure variable followed by the dot operator and then the member name.**	

Explanation:	**Explanation:**
For example in the above structure to read data for the member basic. It is written as scanf("%d",&emp1.s1.basic); Where emp1 is the outer structure variable and s1 is the inner structure variable and basic is the member name.	For example in the above structure to read data for the member day. It is written as scanf("%d",&emp1.joindate.day); Where emp1 is the outer structure variable and joindate is the inner structure variable and day is the member name.

Array of Structures

Declaration

As we know that structure is a derived data type that can group one or more variables of different data types. If the same structure is required for more than once then we declare the variable as array of structures.

The declaration of array of structures is illustrated below:

```
struct student
{
  int rollno;
  int marks1;
};
struct student s1[3];
```

In the above example s1 is a variable declared as array of structures, where s1 is the variable name and the value 3 enclosed in square brackets gives the size of the array variable.

Accessing the structure with array of structure variables

The following segment of code illustrates the process of accessing the structure with array of structure variables.

```
for(i=0;i<3;i++)
{
        scanf( "%d",&s[i].rollno);
        scanf("%d",&s[i].marks1);
}
```

Here s1 is the structure variable declared as an array,rollno and marks1 are members of the structure and I is subscript variable used to vary the index of the array variable.

The members of the structures are accessed by giving **the structure variable name followed by subscript variable enclosed in square brackets followed by dot operator and then the member name.**

Arrays within structures

Declaration

Structure is a derived data type that can group one or more variables of different data types with a single name, if one of these variables or all of these variables is declared as an array variable then it becomes as use of arrays within structures.

Example that illustrates the declaration of array within structure is given below:

```
struct student
{
```

```
  int rollno;
  int marks[3];
};
struct student s1;
```

in the above example variable marks is declared as array of type int.

Accessing data for array declared within structure

The following segment of code illustrates the process of accessing arrays within structure.

```
for(i=0;i<3;i++)
   scanf("%d",&s1.marks[i]);
```

here s1 is the structure variable, marks is the member of structure declared as array variable and I is subscript variable used to vary the index of the array variable.

The members of the structures declared as array are accessed by giving **the structure variable name followed by dot operator and then the member name with subscript variable enclosed in square brackets.**

Passing structures to Functions:

Structure variable can be passed as argument to function in two ways that is by value and by reference. Below we will see with example the procedure of passing structure variable as arguments by value and by reference.

By Value

```
struct employee
{
int empid;
char designation[15];
float salary;
}emp;
main()
{
readvalue(emp); /* function call  with structure variable passed as argument */
}
readvalue(e)
struct employee e;
{
printf("enter employee id, designation, salary:");
scanf("%d%s%f",&e.empid,e.designation,&e.salaray);
}
```

Explanation:

In the above example the structure variable is passed as argument in the function call and received in an **argument by name e** declared of the same structure in the called function and the procedure of reading data to the member of the structure is the same as what we have discussed above.

By Reference

```
struct employee
{
int empid;
char designation[15];
float salary;
}emp;
main()
{
readvalue(&emp); /* function call  with structure variable passed as argument */
}
readvalue(e)
struct employee *e;
{
printf("enter employee id, designation, salary:");
scanf("%d%s%f",&e->empid,e->designation,&e->salaray);
printf("employee id =%d\n",(*e).empid);
printf("designation=%s\n",(*e).designation);
printf("salary=%f\n",(*e).salary);
}
```

Explanation:

In the above example the structure variable is passed as reference argument in the function call and received in an **argument by name e** declared of the same structure **as pointer** in the called function.

The procedure of reading data to the member of the structure is done using &pointervariable->membername for members declared of type int or float and if the member is a string then the data is accessed using pointervariable->membername

The procedure of printing data of the member of the structure is done using (*pointervariable).membername.

2.11. Unions:

Unions are derived data types that allows to group one or more variables of different data type with single name.

Syntax:

```
union <tagname>
{
  Data type member name;
  Data type member name;
};
```

Example:

```
union type
{
```

```
int x;
char y;
float z;
}u;
```

Structures and Unions:

Structures	**Unions**
Structures are derived data types that allows to group one or more variables of different data type with a single name.	Unions are derived data types that allows to group one or more variables of different data type with single name.

Syntax:
struct <tagname>
{
Data type member name;
Data type member name;
};
	Syntax:
	union <tagname>
	{
	Data type member name;
	Data type member name;
	};

Size allocated to the structure variable is the **sum of the bytes allocated to the members of the structure.**	Size allocated to the union variable is the **maximum number of bytes allocated to any member of the structure**.
Example: struct college { int collegeid; char collegaddress[20]; }c; Here c is the structure variable declared which takes a size of 22 bytes i.e., 2 bytes for the member college id and 20 bytes for the member collegeaddres.	**Example:** union type { int x; char y; float z; }u; Here u is the union variable declared which takes a size of 4 bytes i.e., 2 bytes for member x, 1 byte for member y and 4 bytes for member z and the maximum of these is 4 bytes which will be the size allocated to the union variable.

Union of Structures:

A structure variable declared as a member of the union is called as union of structure.

e.g.,

```
union x
{
        Struct student s;
};
```

2.12. Strings:

Defining and Initialization of strings:

Definition: A string is a sequence of characters that is treated as a single data item.

Any group of characters (except double quote sign) defined between double quotation marks is a string constant.

Ex: “man is obviously made to think”

If we want to include a double quote in the string to be printed, then we may use it with a back slash as shown below.

“\”Man is obviously made to think,\” said pascal.”

For ex:

printf(“\”Well Done!”\”);

will output the string

“Well Done!”

While statement

printf(“Well Done!”);

will output the string

Well Done!

The common operations performed on character strings include:

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

Declaring string variables:

C does not support string as a data type. It allows as “**character arrays**” or “**array of characters**”. The general form for declaration,

char string_name[size];

the size determines the number of characters in the string_name.

Ex: char city[10];

char name[30];

when we assign string to a character array it automatically supplies a null character(‘\0’) at the end of the string. The **size** should be **maximum number of characters in the string plus one**.

Initialization of string variables:

Character arrays may be initialized in when they are declared.

Ex: char city[9]=”NEW YORK”;

Char city[9]={‘N’,’E’,’W’,’ ‘,’Y’,’O’,’R’,’K’,’\0’};

The string NEW YORK is 8 characters and one element space is provided for the null terminator.

C permits us to initialize a character array without specifying the number of elements.

Ex:

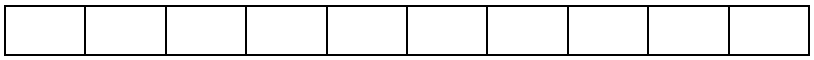
char string[]={‘G’,’O’,’O’,’D’,’\0’};

define the array as a five element array.

We can declare the size much larger than the string size in the initialize.

Ex: char str[10]=”GOOD”;

The storage will look like



Consider,

char str2[3]=”GOOD”; is illegal results in a compile time error

we cannot separate the initialization from declaration. That is,

char str3[5];

str3=”GOOD”;

is not allowed. Similarly,

char s1[4]=”abc”;

char s2[4];

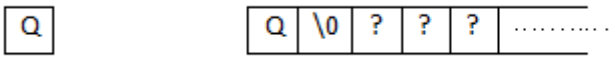
s2=s1;/*error*/

is not allowed

An array name cannot be used as the left operand of an assignment operator.

NULL Character:

The difference between the representations of the character ‘Q’ and the string “Q”



Character ‘Q’ string “Q” represented by its initial address and ends with null character

Reading and writing a string:

Reading a string using scanf function:

We use %s format specification to read in string of characters

Ex: char address[10];

scanf(“%s”,address);

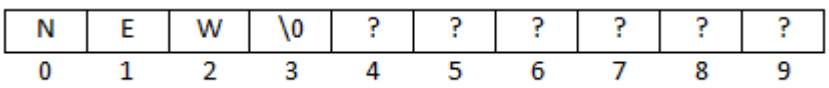
problem with the scanf function is that it terminates its input on the first white space it finds.

A white space includes blanks, tabs, carriage returns, form feeds and new lines.

For ex:

NEW YORK

NEW only read into the array address. The scanf function automatically terminates the string that is read with a null character. Therefore the character array should be large enough to hold string plus the null character. The & is not required. Te address array is created in the memory as shown below:



Note that the unused locations are filled with garbage. If we want to read the entire line

char adr1[5],adr2[5];

scanf("%s%s",adr1,adr2);

with the line of text

New York

Will assign the string "New" to adr1 and "York" to adr2.

We can also specify the field width using the form %ws

Ex: scanf("%ws",name);

Here, two things may happen.

1. The width w is equal to or greater than the number of characters typed in the entire string will be stored in the string variable.
2. The width w is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider,

```
char name[10];
scanf("%5s",name);
```

Input string RAM will be stored as:

R	A	M	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Input string KRISHNA will be stored as:

K	R	I	S	H	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading line of text:

scanf cannot be used for reading a text containing more than one word. C supports a format specification known as the **edit set conversion code** %[..] can be used to read a line containing a variety of characters, including white spaces.

Ex:
```
char line[80];
scanf("%[^\n]",line);
printf("%s",line);
```

Character Operations:

Using getchar and gets functions:

getchar():

We can use this function repeatedly to read successive single characters from the input and place them into character 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. The getchar function takes the following form.

```
char ch;
ch=getchar();
```

function has no arguments.

gets(str):

This is a function which reads a string of text containing white spaces

```
gets(str);
```

it reads the characters into str from the keyboard until a new-line character is encountered and

then appends a null character to the string
ex: char line[80];
gets(line);
printf("%s",line);
last two statements can be combined as:
printf("%s",gets(line));

putchar and puts functions:

putchar:
putchar is a function which is used to print one character at a time on the screen

```
char ch;
ch=getchar();
putchar(ch);
```

puts(str):
we will use this function to print a string on to the screen
ex: char line[80];

```
gets(line);
puts(line);
```

- **Arithmetic operations on characters:**

A character constant or character variable is used in an expression, it is automatically converted into an integer value.

For ex: x='a';
Printf("%d",x);

Displays the number 97(ASCII equivalent of a).

Consider,

x='z'-1;
z is 122
x=122-1=121

may use character constants in relational expressions. For ex,

ch>='A'&&ch<='Z'

would test the character contained in the variable ch is an upper-case letter

ex: x='7'-'0'
= 55-48
=7

Ex1: char x='a';
printf("%d",x);

output: 97

Ex2: char x='a';
printf("%c",x);

output: a

Ex3: char x='a'+1;

printf(“%d”,x);

output: 98

Ex4: char x=’z’-‘a’

printf(“%d”,x);

output:25

C converts a string of digits into their integer values by using the function.

x=atoi(string);

ex: number=”1988”;

year=atoi(number);

number is a string variable which is assigned the string constant “1988”. The function a to i converts the string “1988” to its numeric equivalent 1988 and assigns it to the integer variable year. If no valid conversion then it returns 0.

number=1988

str=itoa(number);

this itoa function works exactly opposite to atoi.

String function conversions are stored in the header file <stdlib.h>

/*Program to print the alphabet set a to z and A to Z in decimal and character form*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
      char c;
   for(c=65;c<=122;c++)
   {
                        if(c>90&&c<97)
                        continue;
                        printf(“|%4d-%c”,c,c);
          }
              printf(“\n”);
}
```

Output:

|65-A|66-B|67-C|68-D|69-E|70-F|71-G|72-H|73-I|74-J|75-k|76-L|77-M|78-N|79-O|80-P|81-Q|82-R|83-S|84-T|85-U|86-V|87-W|88-X|89-Y|90-Z|97-a|98-b|99-c|100-d|101-e|102-f|103-g|104-h|105-i|106-j|107-k|108-l|109-m|110-n|111-o|112-p|113-q|114-r|115-s|116-t|117-u|118-v|119-w|120-x|121-y|122-z

2.13. String handling functions

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

The following table provides most commonly used string handling function and their use...

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strrev()	strrev(string)	Reverses the characters in a given string.

2.14. Command line arguments:

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
```

```
  }
  else {
    printf("One argument expected.\n");
  }
}
```

Output:

arg.exe testing
The argument supplied is testing

arg.exe testing1 testing2
Too many arguments supplied.
arg.exe
one argument expected.

i) Question & Answers: (2 marks)

1. What is a command line argument? How do you pass a command line arguments to a C-program? (Jan – 2020)

Ans: The command line arguments are handled using **main () function arguments** where **argc refer**s to the number of **argument**s passed, and **argv []** is a pointer array which points to each argument passed to the program.

2. Let int a[5] = { 1, 3, 5, 7, 9}. What is the value output of: printf("%d", *(a+2)). Justify your answer. (Jan – 2020)

Ans: 5
Array name is base address, so a+2 will result in address of 3 element.

3. List out any four string handling functions in C. (Dec – 2019)

Ans: strcmp(), strcat(), strlen(), strcpy()

4. List out the storage classes in C. (Dec – 2019)

Ans: auto, extern, static, and register

5. Write a recursive function to compute factorial of an integer. (Dec - 2019)

Ans:
```
int factorial(int i)
    {
      if(i<=1)
          return 1;
     else
       return i * factorial(i-1);
  }
```

6. Differentiate between structures and unions. (Dec – 2019)

Ans: Structures will occupy more memory and whereas union will occupy less memory compared to that of structures.

7. Write a C function to interchange two values using pointers.(Nov/Dec – 2019)

Ans:
```
void swap(int *num1, int *num2)
  {
      int temp;
```

```
    temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

8. In what way call by value mechanism differs from call by reference mechanism? (Jun/ Jul – 2019)

Ans: In **Call by value**, a copy of the variable is passed whereas in **Call by reference**, a variable itself is passed.

9. What is a pointer? Differentiate between ∗ and & with pointers. (May/ Jun – 2019)

Ans: A **pointer** in **C is** a variable that holds the memory address of another variable.
* is a declaration or dereference operator and & is an address of operator.

10. Write the syntax to read string from the keyboard. (Nov/ Dec – 2018)

Ans: scanf("%s", str);

11. Define function prototype. Give the general syntax of function prototype.

Ans: A function prototype is simply the declaration of a function that specifies function's name, parameters and return type.
Syntax: returntype functionname(parameter list);

12. What is the problem with getchar ()? (Dec – 2016)

Ans: getchar() is a function, used to read only one character at a time.

13. Declare a struct name containing field's first_name, middle_name, last_name within a struct student. (Dec – 2016)

Ans:
```
struct student
{
    struct
    {
      char first[20];
      char middle[20];
      char last[20];
     }name;
  }s;
```

14. Define precedence and order of evaluation. (Dec – 2016)

Ans: Complex expressions are executed according to precedence or priority of operators and order of evaluation depends on precedence and associativity. Associativity specifies the order in which the operators are evaluated with the same precedence in a complex expression. Associativity is of two ways i.e., left – to – right and right – to – left

15. Give syntax to create a pointer to function. (Dec/Jan – 2015/2016)

Ans: int (*fpFunc)(int x,int y);

ii) MCQs:

1. A _____ is a self contained block of code that performs a particular task.
a. **Function** b. Array c. Union d. Pointer
2. _____ means, where a function calls itself.
a. Non Recursion b. Function call c. Prototype d. **Recursion**
3. A _____ variable is a memory variable that can store the address of another variable declared for the same type.
a. Array b. Union **c. Pointer** d. Structure
4. _____ name itself is an address or pointer.
a. Typedef b. **Array** c. Pointer d. Structure
5. _____of pointers is nothing but collection of addresses.
a. Pointer b. Structure c. Cast d. **Array**
6. _____ is a default storage class.
a. Static b. **Auto** c. Extern d. Register
7. The _____ storage class is used to define local variables that should be stored in a register instead of RAM.
a. Static b. **Register** c. Auto d. Extern
8. _____ is a very useful user defined data type supported in c that allows grouping one or more variables of different data types with a single name.
a. Array b. Union c. Pointer d. **Structure**
9.A _____ is a sequence of characters that is treated as a single data item.
a. Character b. Array c. **String** d. Integer
10. _____ is a function which reads a string of text containing white spaces.
a. **gets** b. scanf c. getchar d. read
11. _____ appends string2 to string1.
a. Strcpy b. Strcmp c. **Strcat** d. Strupr
12. Every string ends with _____
a. **Null Character** b. Bold character c. Zero Character d. Single Character
13. A _____ is a function compares two strings
a. Strcpy b. **Strcmp** c.Strcat d. Strupr
14. When we place the declaration above all the functions, the prototype is referred to as a _____ protoype.
a. Local b. Static c. Personal d. **Global**
15. A function can be accessed by making a _____ to the particular function.
a. Signal b.**Call** c. Rise d. Trigger

iii) Question and Answers: 10 Marks

1. What is a pointer? What are the problems with the pointers? Write a C program to print the element of a one – dimensional array using pointers.

Ans: Pointer: 3M

A pointer variable is a memory variable that can store the address of another variable declared for the same type. A variable is identified as pointer if it is declared for its type and is preceded with the indirection operator (*****).

Problems with the pointers: 2M

1. Undefined behavior
2. Garbage

C program to print the element of a one – dimensional array using pointers: 5M

```
#include <stdio.h>
int main() {
   int data[5];
   printf("Enter elements: ");
   for (int i = 0; i < 5; ++i)
      scanf("%d", data + i);
   printf("You entered: \n");
   for (int i = 0; i < 5; ++i)
      printf("%d\n", *(data + i));
   return 0;
}
```

2. Discuss in brief about the prototype of functions. Write recursive and non recursive functions in C to calculate nth Fibonacci number.

Ans: **Function Prototype(or function declaration): 2M**

- Like variable, all functions in a C program must be declared, before they are invoked
- A function declaration (also known as function prototype) consists of four parts
  - Return type(or function type)
  - Function name
  - Parameter list
  - Terminating semicolon

Syntax:

returntype functionname(parameter list);

Ex:

void drawcircle(void);

Recursive function to find nth Fibonacci term: 4M

```
long fibo(int num)
{
   if(num == 0)      //Base condition
      return 0;
   else if(num == 1) //Base condition
      return 1;
   else
      return fibo(num-1) + fibo(num-2);
}
```

Non Recursive function to find nth Fibonacci term: 4M

```
int fib(int n)
{
 /* Declare an array to store Fibonacci numbers. */
 int f[n+1];
 int i;

 /* 0th and 1st number of the series are 0 and 1*/
 f[0] = 0;
 f[1] = 1;

 for (i = 2; i <= n; i++)
 {
    /* Add the previous 2 numbers in the series
       and store it */
    f[i] = f[i-1] + f[i-2];
 }

 return f[n];
}
```

3. Write a C program to interchange two values by using pointers. (July – 2019) 10M

Ans:

```
#include <stdio.h>
int main()
{
  int x, y, *a, *b, temp;
  clrscr();
  printf("Enter the value of x and y\n");
  scanf("%d%d", &x, &y);
```

```
  printf("Before Swapping\nx = %d\ny = %d\n", x, y);
  a = &x;
  b = &y;
  temp = *b;
  *b = *a;
  *a = temp;
  printf("After Swapping\nx = %d\ny = %d\n", x, y);
  return 0;
}
```

4. Define structure. Why we need structures? (July – 2019)

Ans: Definition: 2M

Structure is a very useful derived/user defined data type supported in c that allows grouping one or more variables of different data types with a single name.

Need of structures: 8M

C has built in primitive and derived data types. Still not all real-world problems can be solved using those types. You need custom data type for different situations.

For example, if you need to store 100 student record that consist of name, age and mobile number. To code that you will create 3 array variables each of size 100 i.e. name[100], age[100], mobile[100]. For three fields in student record it say seem feasible to you. But, think how cumbersome it would be to manage student record with more than 10 fields, in separate variables for single student.

To overcome this we need a user defined data type. In this tutorial I am going to explain how easily we will deal with these situations using structures in C programming language.

5. Differentiate between call by value and call by reference. Demonstrate the difference by writing functions to swap two elements. (July – 2019)

Ans:

- Call by value and call by reference (also known as pass-by-value and pass-by-reference).
- These methods are different ways of passing (or calling) data to functions.
- **Call by Value: 5M**
    - If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function. Let's take a look at a call by value example:

```
#include <stdio.h>
void call_by_value(int x, int y)
{
int t;
printf("Inside call_by_value x = %d and y=%d before swapping.\n", x,y);
```

```
    t=x;
    x=y;
    y=t
    printf("Inside call_by_value x = %d and y = %d after swapping.\n", x,y);
    }
int main()
{
    int a=10,b=20;
    printf("a = %d and b=%d before function call_by_value.\n", a,b);
    call_by_value(a,b);
    printf("a = %d and b=%d after function call_by_value.\n", a,b);
    return 0;
}
```

- **Call by Reference: 5M**
  - If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example:

```
#include <stdio.h>
#include<conio.h>
void call_by_reference(int *x,int *y)
{
printf("Inside call_by_reference x=%d and y = %d before swapping.\n",*x, *y);
t=*x;
*x=*y;
*y =t;
printf("Inside call_by_reference x=%d and y = %d after swapping.\n", *x,*y);
}
int main()
{
int a=20, b=10;
clrscr();
printf("a=%d and b = %d before function call_by_reference.\n", a,b);
call_by_reference(&a,&b);
printf("a=%d and b = %d after function call_by_reference.\n", a,b);
return 0;
}
```

6. Discuss in detail about storage classes.

Ans: A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class 2M

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class 2M

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class 3M

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>
 /* function declaration */
void func(void);
static int count = 5; /* global variable */
main() {
   while(count--) {
      func();
   }
   return 0;
}
/* function definition */
void func( void ) {
   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class 3M

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
   count = 5;
```

```
  write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
  printf("count is %d\n", count);
}
```

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c.

Unit -3

Data Structures, Overview of data structures, stacks and queues, representation of a stack, stack related terms, operations on a stack, implementation of a stack, evaluation of arithmetic expressions, infix, prefix, and postfix notations, evaluation of postfix expression, conversion of expression from infix to postfix, recursion, queues - various positions of queue, representation of queue, insertion, deletion, searching operations.

3.1. Data Structures:

- *A data structure is an aggregation of atomic and composite data into a set with defined relationships.*

Atomic and Composite data

- **Atomic data** are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data.
- The opposite of atomic data is composite data.
- **Composite data** can be broken into subfields that have meaning.

3.2. Overview of data structures:

- The data structure can be divided into two basic types: preliminary data structure and secondary data structure.
- The primitive data structures are the basic data types such as int, char, float where the non primitive data structures are the data structures which are basically derived from primitive data structures.
- They can be further categorized into linear and non linear data structures.
- **Linear data structures** are the data structures in which the data is arranged in a list or a straight sequence.

  For ex: arrays, lists, stacks, queues
- **Non linear data structures** are the data structures in which data may be arranged in hierarchical manner.

  For ex: trees, graphs

Linear data structures:

Arrays:

An array is a group of homogeneous data which shares the same name.

List

- List is basically the collection of elements arranged in a sequential manner.
- In memory we can store the list in two ways –
  - List of sequentially stored elements – using arrays
  - List of elements with associated pointers – using linked lists

Stack

- A stack is a last in – first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

Queue

- A Queue is a list in which data can be inserted at one end, called the rear and deleted from the other end, called the front. It is a first in first out(FIFO) restricted data structure.

Non Linear data structures:

Trees

- A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches that connect the nodes.

Graphs

- A graph is a collection of nodes, called vertices, and line segments , called arcs or edges, that connect pairs of nodes.

3.3. STACKS:

A Stack is a linear list in which all additions and deletions are restricted to one end called to ***top***.

- It is also known as restrictive data structure.
- If you insert a data series into a stack and then remove it, the order of the data is reversed.

  Data input as {5,10,15,20} is removed as {20,15,10,5}
- This reversing attribute is why stacks are known as the **Last in First Out** (LIFO) data structure.
- Any situation we can add or remove an object at the top is a stack.
- If you want to remove any object other than the one at the top, you must first remove all objects above it.
- A graphic representation of stack is shown in fig.

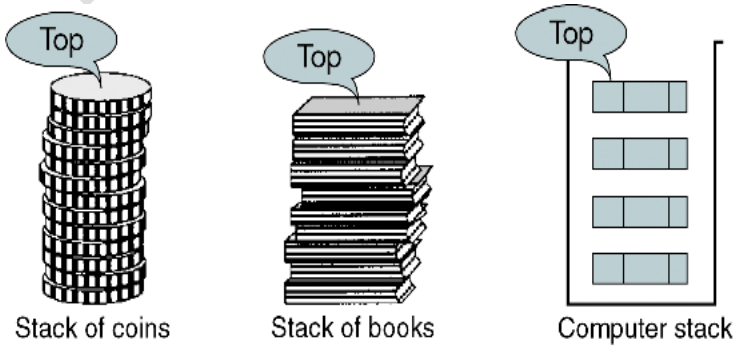


FIGURE Stack

Definition:

A stack is a last – in – first – out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

3.4. Representation of stack:

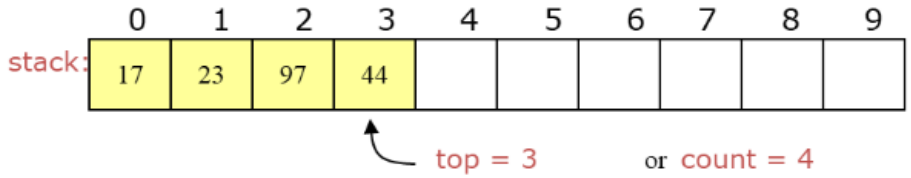
- Stack can be represented by using Arrays or Linked lists.

4.1. Array representations of stacks:

Several data structures can be used to implement a stack. Here we implement a stack as an array.

a. Pushing:

- Initially consider the elements in the stack are: 17, 23, 97, now top = 2 or count = 3
- After pushing the element 44 into the stack the top or count will be incremented by 1, then top = 3 or count = 4.
- For every push operation the top or count will be incremented by 1.

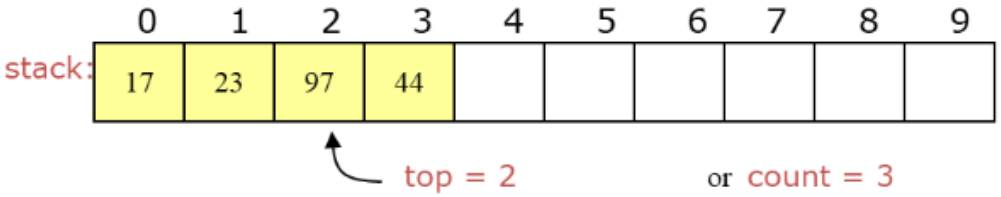


- If the bottom of the stack is at location 0, then an empty stack is represented by **top = -1 or count = 0**
- To add (push) an element, either:
  - Increment top and store the element in stack[top], or
  - Store the element in stack[count] and increment count

b. Popping:

- To remove (pop) an element, either:
  - Get the element from stack[top] and decrement top, or
  - Decrement count and get the element in stack[count]

After Popping,



When you pop an element, do you just leave the "deleted" element sitting in the array?
The surprising answer is, "it depends" on language.

- There are two stack errors that can occur:
  - **Underflow**: trying to pop (or peek at) an empty stack
  - **Overflow:** trying to push onto an already full stack

3.5. Stack related terms:

- **Push, Pop, Stack top, Overflow, Underflow.**

3.6. Operations on stack:

Three basic operations are push, pop, and stack top.

a. **Push:** It is used to insert data into the stack.
b. **Pop:** removes data from a stack and returns the data to the calling module.
c. **Stack top:** returns the data at the top of the stack without deleting the data from the stack.

a. Push:

- Push adds an item at the top of the stack
- After the push, the new item becomes the top
- The only problem with this simple operation is that there is room for the new item
- If there is not enough room, the stack is an **overflow** state and the item cannot be added.
- Figure shows the push stack operation

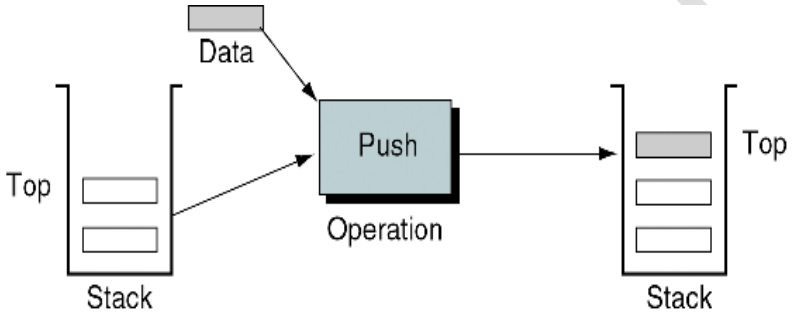


FIGURE Push Stack Operation

b. Pop:

- When we pop a stack, we remove the item at the top of the stack and return it to the user.
- After removing the top item, the next older item in the stack becomes the top.
- When the last item in the stack is deleted, the stack must be set to its empty state.
- If pop is called when the stack is empty, it is in an **underflow** state.
- The pop stack operation is shown in fig.

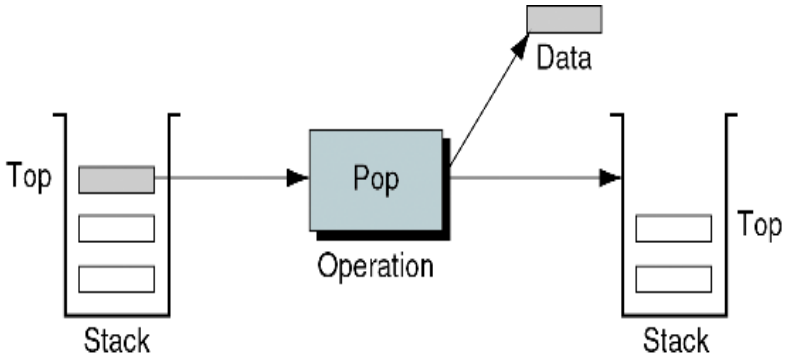


FIGURE Pop Stack Operation

c. Stack top:

- The stack top copies the item at the top of the stack; that is, it returns the data in top element to the user but does not delete it.
- This operation is nothing but as reading the stack top.
- Stack top can also result in underflow if the stack is empty.
- The stack top operation is shown in figure.

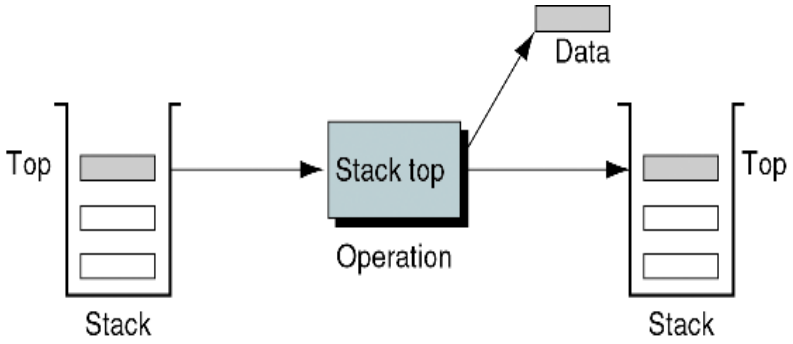


FIGURE Stack Top Operation

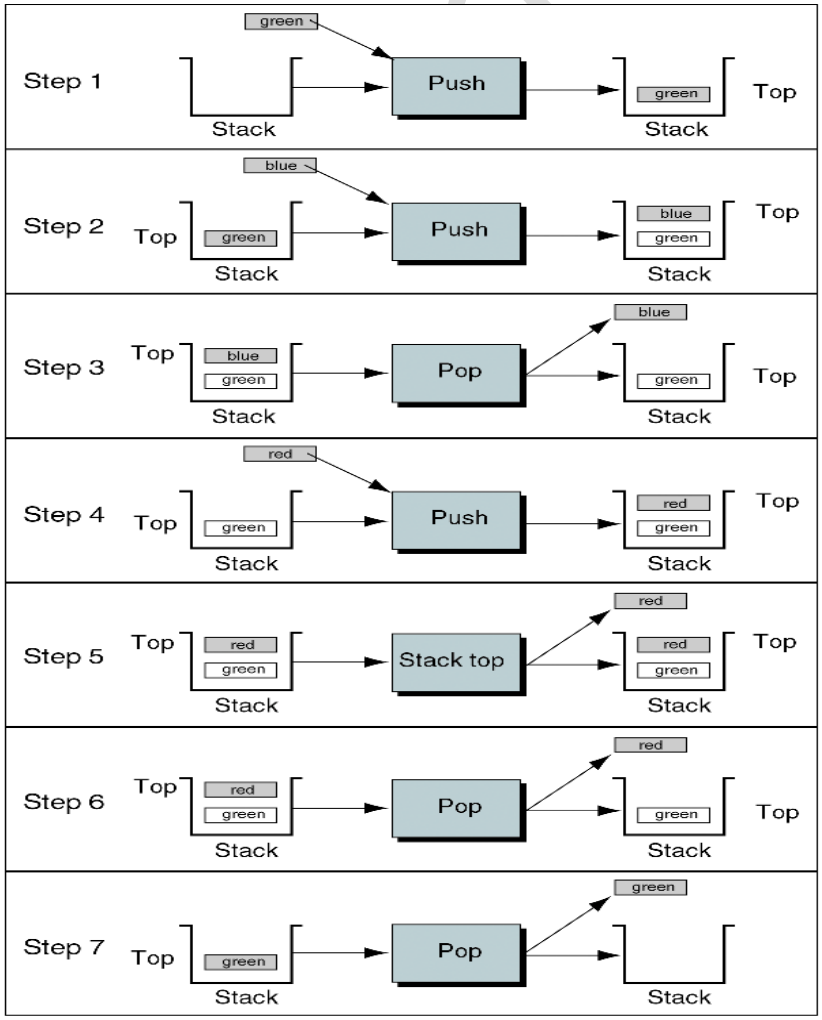


FIGURE Stack Example

3.7. Implementation of a stack:

Push operation:

```
int stack[MAX];
int top=-1;
void push(int ele)
{
                    if(top>MAX)
                    {
                     printf("stack is full\n");
                     exit(0);
                    }
                    stack[++top]=ele;
}
```

Pop operation:

```
void pop()
{
                    if(top==-1)
                    {
                     printf("stack is empty\n");
                     exit(0);
                    }
                    a=stack[top--];
                    printf("the element popped is %d",a);
}
```

Display elements in stack:

```
void display()
{
                    int k;
                    if(top==-1)
                    {
                     printf("stack is empty\n");
                     exit(0);
                    }
                    printf("\n\tElements present in the stack are:\n\t");
                    for(k=0;k<=top;k++)
                    printf("%d\t",stack[k]);
}
```

3.8. Evaluation of Arithmetic Expressions

- Expressions are evaluated using an 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 values before evaluation is attempted
- Examples of evaluation statements are
  x=a*b-c;
  y=b/c*a;
  z=a-b/c+d;

```
/*Program to illustrate the use of variables in expressions and their evaluation*/
#include<stdio.h>
#include<conio.h>
void main()
{
float a,b,c,x,y,z;
clrscr();
a=9;
b=12;
c=3;
x=a-b/3+c*2-1;
y=a-b/(3+c)*(2-1);
z=a-(b/(3+c)*2)-1;
printf("x=%f\n",x);
printf("y=%f\n",y);
printf("z=%f\n",z);
getch();
}
```

Output:

```
x=10.000000
y=7.000000
z=4.000000
```

3.9. Infix

Infix notation: X + Y. Operators are written in-between their operands.

3.10. Prefix

Prefix notation: + X Y. An **expression** is called the **prefix expression** if the operator appears in the **expression** before the operands.

3.11. Postfix

Postfix notation: X Y +. A **postfix expression** is a collection of operators and operands in which the operator is placed after the operands.

3.12. Evaluation of postfix expression

- Fundamental principles followed while evaluating the postfix expression :
  1.Read the expression from left to right.
  2.If there comes an operand , push it into the stack.
  3.If there comes an operator , pop operand 1 and operand 2 and then :
    A . Push '('
    B . Push 'operand 2'
    C . Push 'operator'
    D . Push 'operand 1'
    E . Push ')'

Advantages of Postfix Expression:

- Postfix notation is easier to work with. In a postfix expression operands appear before the operators, there is no need of operator precedence and other rules. In postfix expression the topmost operands are popped off and operator is applied and the result is again pushed to the stack and thus finally the stack contains a single value at the end of the process.
- In postfix expression, there are no parentheses and therefore the order of evaluation will be determined by the positions of the operators and related operands in the expression.

Algorithm: Evaluate a Postfix Expression

```
 Pre                          a valid expression
  Post                        postfix value computed
  Return value of expression
createstack (stack)
/* scan the input string reading one element */
while (not end of input) {
                             symb = next input character;
                             if (symb is an operand)
                                push(opndstk, symb)
                             else {
                             /* symb is an operator */
                                opnd2 = pop(opndstk);
                                opnd1 = pop(opndstk);
                                value = result of applying symb to opnd1 and opnd2;
                                push(opndstk, value);
                             } /* end else */
} /* end while */
return (pop(opndstk));
```

Example:

Postfix Expression: 6 2 3 + - 3 8 2 / + * 2 $ 3 +

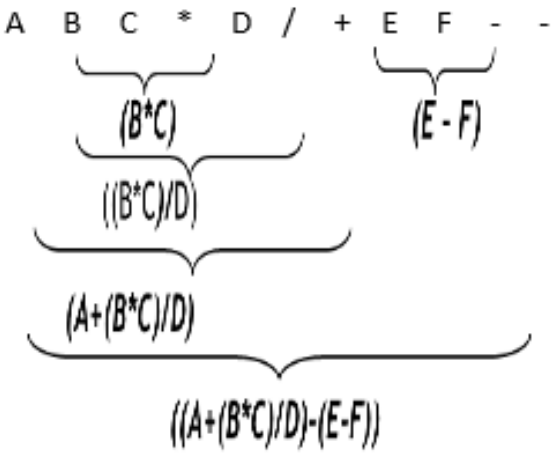
symb	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Algorithm for Postfix Expression :

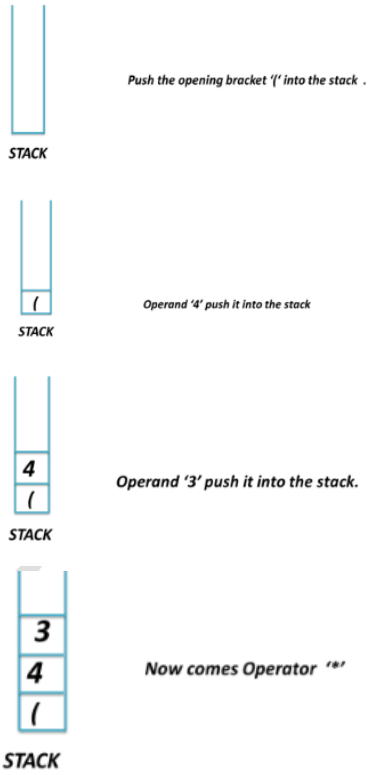
1. Read the element.
2. If element is an operand then:
   Push the element into the stack.
3. If element is an operator then :
   Pop two operands from the stack.
   Evaluate expression formed by two operand and the operator.
4. If no more elements then:
   Pop the result
   Else
   Goto step 1.
   Push the result of expression in the stack end.

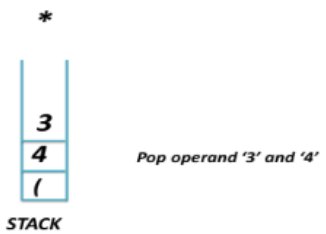
How to evaluate postfix (manually)

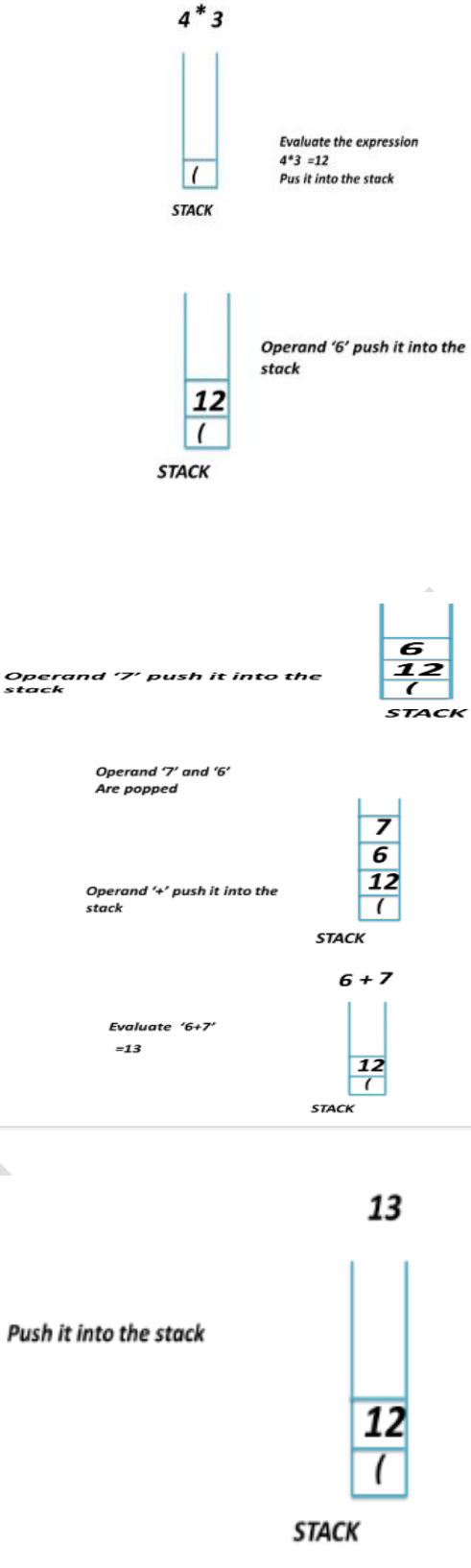
Going from left to right, if you see an operator, apply it to the previous two operands (numbers)
Example:



Equivalent infix: A+ B * C / D– (E– F)
Ex: 43*67+5-+)






4 * 3
(
STACK
Evaluate the expression
4*3 =12
Pus it into the stack
12
(
STACK
Operand '6' push it into the stack
Operand '7' push it into the stack
6
12
(
STACK
Operand '7' and '6'
Are popped
Operand '+' push it into the stack
7
6
12
(
STACK
6 + 7
Evaluate '6+7'
=13
12
(
STACK
13
Push it into the stack
12
(
STACK

Operand '5' push it into the stack

Operands '5' and '13' are popped

Now operator '-' occurs

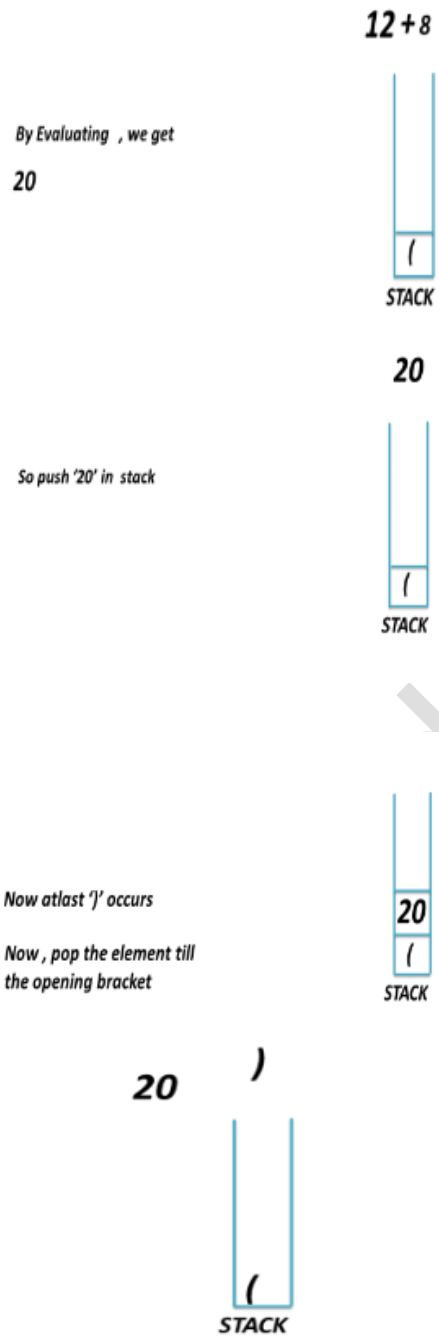
Evaluating "13-5"

We get '8'

8

Now, push the resultant value i.e '8' in the stack

Operator '+'



C Implementation

```
#include<stdio.h>
#include<conio.h>
#define SIZE 50
char s[SIZE];
int top=-1;
void push(char elem)
{
   s[++top]=elem;
```

```
}
char pop()
{
   return(s[top--]);
}
int cal(int elem1,char op,int elem2)
{
   switch(op)
   {
   case '+':return(elem1+elem2);
   case '-':return(elem1-elem2);
   case '*':return(elem1*elem2);
   case '/': return(elem1/elem2);
   }
}
void main()
{
   char pofx[50],ch;
   int i=0,opnd2,opnd1,val;
   clrscr();
   printf("\n\nRead the Postfix Expression ? ");
   scanf("%s",pofx);
   while((ch=pofx[i++])!='\0')
   {
                                    if(isdigit(ch))
                                    {
                                    ch=pofx[i-1]-'0';
                                    push(ch);
                                    }
                                    else
                                    {
                                      opnd2=pop();
                                      opnd1=pop();
                                      val=cal(opnd1,ch,opnd2);
                                      push(val);
                                    }
    }
    val=pop();
    printf("the result of evaluation of postfix expression is:%d",val);
getch();
}
```

3.13. Conversion of expression from infix to postfix

- **Infix notation:** Infix notation is the common arithmetic and logical formula notation, in which operators are written infix-style between the operands they act on

  E.g. A + B
- **Postfix notation:** In Postfix notation, the operator comes after the Operand.
- For example, the Infix expression A+B will be written as AB+ in its Postfix Notation.
- Postfix is also called 'Reverse Polish Notation'
- One disadvantage with the infix notation is that we need to use parentheses to control the evaluation of the operators.
- In postfix notation we do not need parentheses.
- Infix to postfix can be done in two ways:
  - Manual transformation
  - Algorithmic transformation

i) Manual Transformation:

The rules for manually converting infix to postfix expressions are as follows:

1. Fully parenthesize the expression using any explicit parentheses and the arithmetic precedence – multiply and divide before add and subtract.
2. Change all infix notations in each parenthesis to postfix notation, starting from the innermost expressions. Conversion to postfix notation is done by moving the operator to the location of the expressions closing parenthesis.
3. Removing all parentheses

Example:

A + B * C

Step1 results in

(A + (B * C))

Step2 moves the multiply operator after c

(A (B C *) +)

and then moves the addition operator to between the last two closing parentheses. This change is made because the closing parenthesis for the plus sign is the last parenthesis. We know have

(A (B C *) +)

Finally , step 3 moves the parentheses

A B C * +

More complex example

This example is not only longer but it already has one set of parentheses to override the default evaluation order.

(A + B) * C + D + E * F – G

Step1 adds parentheses

(((((A + B) * C) + D) + (E * F)) – G)

Step2 then moves the operators .

(((((A B +) C *) D +) (E F *) +) G -)

Step3 removes the parentheses.

A B + C * D + E F * + G –

ii) Algorithmic Transformation

- The manual operation would be difficult to implement in a computer
- Another technique is that we can easily implement with a stack.
- Simple example:

A * B converts to A B *

- We can read the operands and output them in order.
- The problem becomes how to handle the multiply operator; we need to *postpone* putting it in the output until we have read the right operand B.
- In this case we push operator into a stack and, after the whole infix expression has been read, pop the stack and put the operator in the postfix expression
- Complex Expression:

A * B + C converts to A B * C +

- If we were to simply put the operators into the stack as we did earlier and then pop them to the postfix expression after all of the operands had been read, we would get the wrong answer.
- Some how we must pair the two operators with their correct operands.
- One possible rule might be to postpone an operator only until we get another operator .
- Then before we push the second operator, we could pop the first one and place it in the output expression.
- This logic works in this case, but it won't for others.
- Consider the following Example:

A + B * C Converts to A B C * +

- Infix expression use a precedence rule to determine how to group the operands and operators in an expression.
- We can use the same rule to convert infix to postfix
- When we need to push an operator into the stack, if its priority is higher than the operator at the top of the stack, we go ahead and push it into the stack.
- Conversely , if the operator at the top of the stack has a higher priority than the current operator, it is popped and placed in the output expression. Using this rule with the above expression, we would take the following actions:
    1. Copy operand A to output expression
    2. Push operator + into stack
    3. Copy operand B to output expression
    4. Push Operator * into stack (priority of * is higher than +.)
    5. Copy operand C to output expression
    6. Pop operator * and copy to output expression
    7. Pop Operator + and copy to output expression
- Priority
    - Priority 2: * /
    - Priority 1: + -

- Priority 0: (

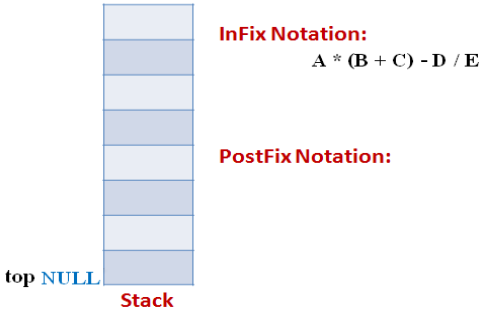
- One more complex example is

A + (B * C) – D / E

★ Let the incoming the Infix expression be:

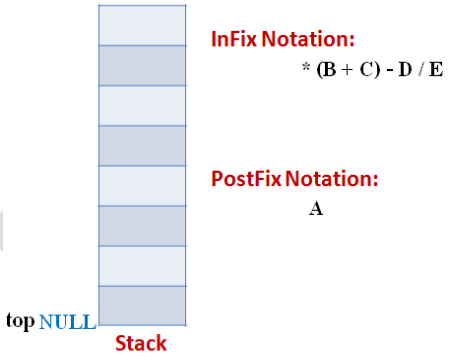
A * (B + C) – D / E

Stage 1: Stack is empty and we only have the Infix Expression.



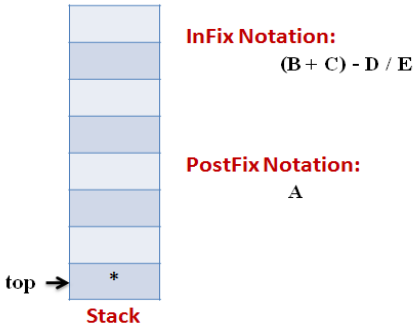
Stage 2

★ The first token is **Operand A** Operands are Appended to the Output as it is.



Stage 3

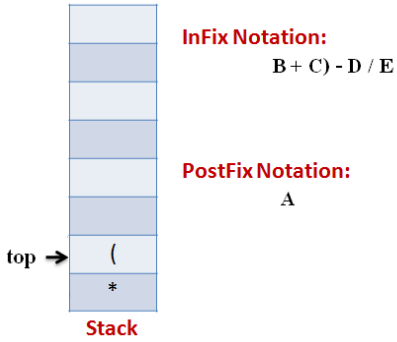
★ Next token is ***** Since **Stack is empty (top==NULL)** it is **pushed into the Stack**



Stage 4

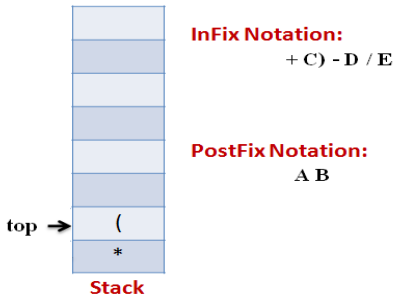
★ Next token is **(** the precedence of open-parenthesis, when it is to go inside, is maximum.

★ But when another operator is to come on the top of **'('** then its precedence is least.



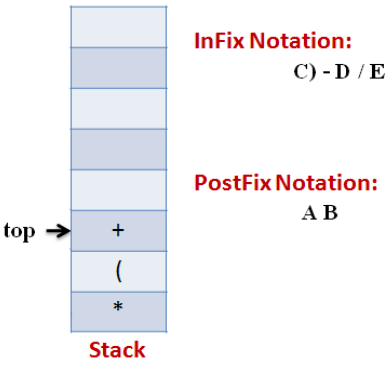
Stage 5

★ Next token, **B** is an operand which will go to the Output expression as it is



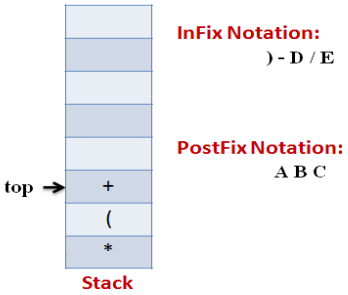
Stage 6

★ Next token, + is operator, We consider the precedence of **top element in the Stack**, **'('**. The outgoing precedence of open parenthesis is the least (refer point 4. Above). So + gets **pushed into the Stack**



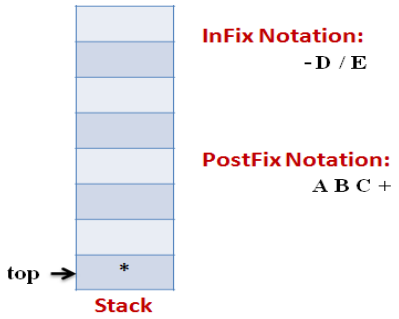
Stage 7

★ Next token, **C**, is appended to the output



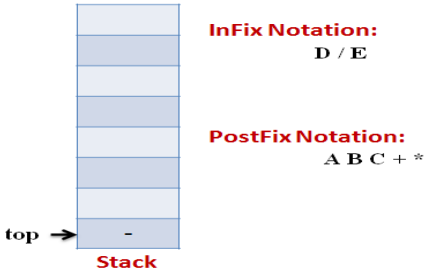
Stage 8

★ Next token **)**, means that **pop all the elements from Stack** and **append them to the output** expression till we read an opening parenthesis.



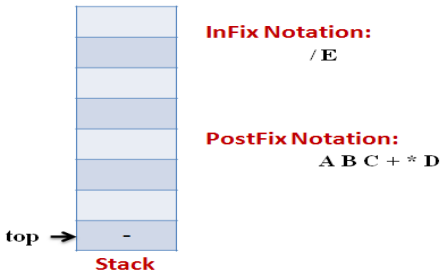
Stage 9

★ Next token, **-**, is an operator. The precedence of operator on the top of Stack **'*'** is more than that of Minus. So we **pop multiply** and **append it to output** expression. Then **push minus in the Stack**.



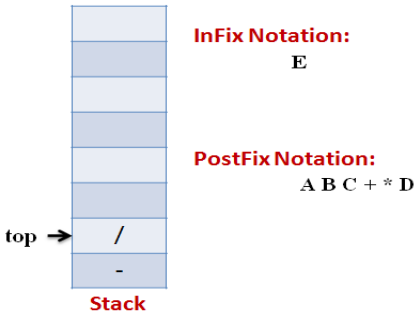
Stage 10

★ Next, Operand '**D**' gets **appended to the output**.



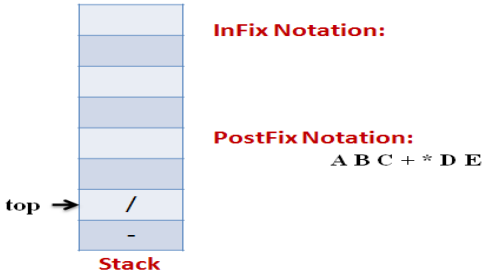
Stage 11

★ Next, we will insert the **division** operator into the Stack because its precedence is more than that of minus.



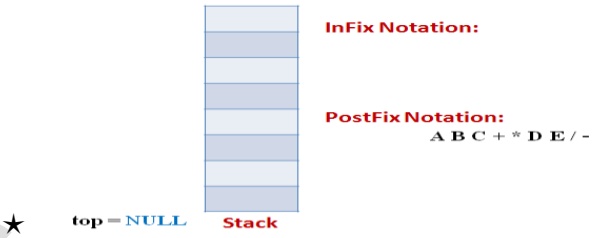
Stage 12

★ The last token, **E**, is an operand, so we **insert it to the output** Expression as it is.



Stage 13

★ The input Expression is complete now. So we **pop the Stack** and **Append it to the Output Expression** as we pop it.



Algorithm: Convert infix to postfix

Algorithm inToPostFix (formula)
Convert infix formula to postfix

Pre formula is infix notation that has been edited to ensure that there are no syntactical errors

Post Postfix formula has been formatted as a string

Return Postfix formula

1 Create stack (stack)
2 loop (for each character in formula)
1 if (Character is open paranthesis)
1 pushStack (stack, character)
2 elseif (character is close paranthesis)
1 popStack (stack. Character)
2 loop (Character not open paranthesis)
1 concatenate character to postFixExpr
2 popStack (stack, character)

```
            3 end loop
        3 elseif ( character is operator )
            Test priority of token to token at top of stack
            1 popstack ( stack, topToken)
            2 loop ( not emptyStack (stack) AND priority ( character ) < = priority ( topToken ) )
                1 popStack ( stack, tokenOut )
                2 concatenate tokenOut to postFixExpr
                3 stackTop ( stack,, topToken )
            3 end loop
            4 pushStack ( stack, token )
        4 else
            character is operand
            1 Concatenate token to postFixExpr
        5 end if
3 end loop
        Input formula is empty. Pop stack to postfix
4 loop ( not emptyStack ( Stack ) )
        1 popStack ( stack, character )
        2 concatenate token to postFixExpr
5 end loop
6 return postFix
end inToPostFix
```

C Implementation

```
#include<stdio.h>
#include<conio.h>
#include <ctype.h>
#define SIZE 50
char s[SIZE];
int top=-1;
void push(char elem)
{
   s[++top]=elem;
}
char pop()
{
   return(s[top--]);
}
int pr(char elem)
{
```

```
    switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
}
void main()
{
    char infx[50],pofx[50],ch;
    int i=0,k=0;
    clrscr();
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
    while((ch=infx[i++])!='\0')
    {
          if( ch == '(') push(ch);
          else
             if( ch == ')')
                   {
                      while( s[top] != '(')
                            pofx[k++]=pop();
                      pop(); /* Remove ( */
                   }
                   else
                    if(isalnum(ch)) pofx[k++]=ch;
              else
              {       /* Operator */
                      while( pr(ch) <= pr(s[top]))
                            pofx[k++]=pop();
                      push(ch);
              }
             }
    while( s[top] != '#')
          pofx[k++]=pop();
    pofx[k]='\0';
    printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n",infx,pofx);
    getch();
}
```

Output:
Read the infix expression?
(a+b)*(c-d)/e
Given Infix Expn: (a+b)*(c-d)/e Postfix Expn: ab+cd-*e/

3.14. Recursion: A function calling itself is called as recursion.

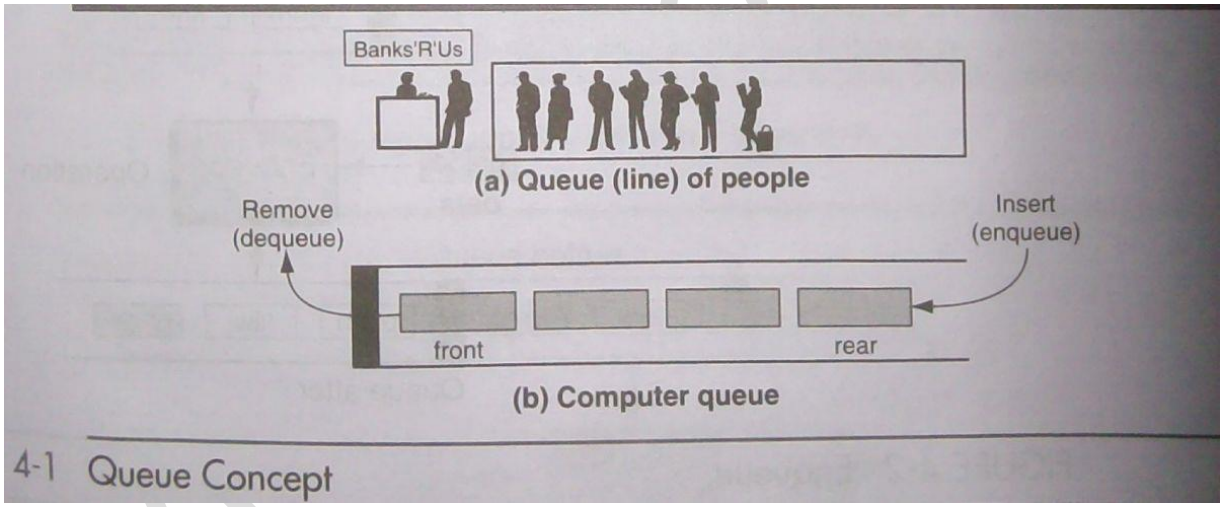
3.15. Queues – various positions of queues

- A Queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- These restrictions ensure that the data are processed through the queue in the order in which they are received.
- In the other words, a queue is a first in first out (FIFO) structure.
- A queue is same as a line

Ex:

a. A line of people waiting for the bus at a bus station is a queue,
b. A list of calls put on held to be answered by a telephone operator is a queue and
c. A list of waiting jobs to be processed be a computer is a queue

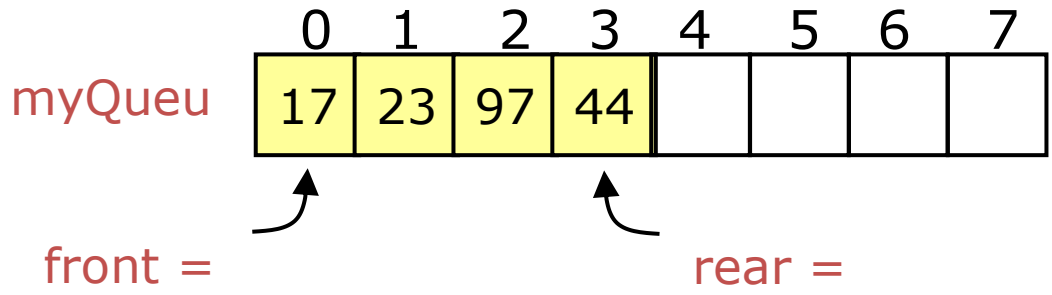
- Fig shows two representations of a queue
  a. Queue of a people and
  b. Computer queue



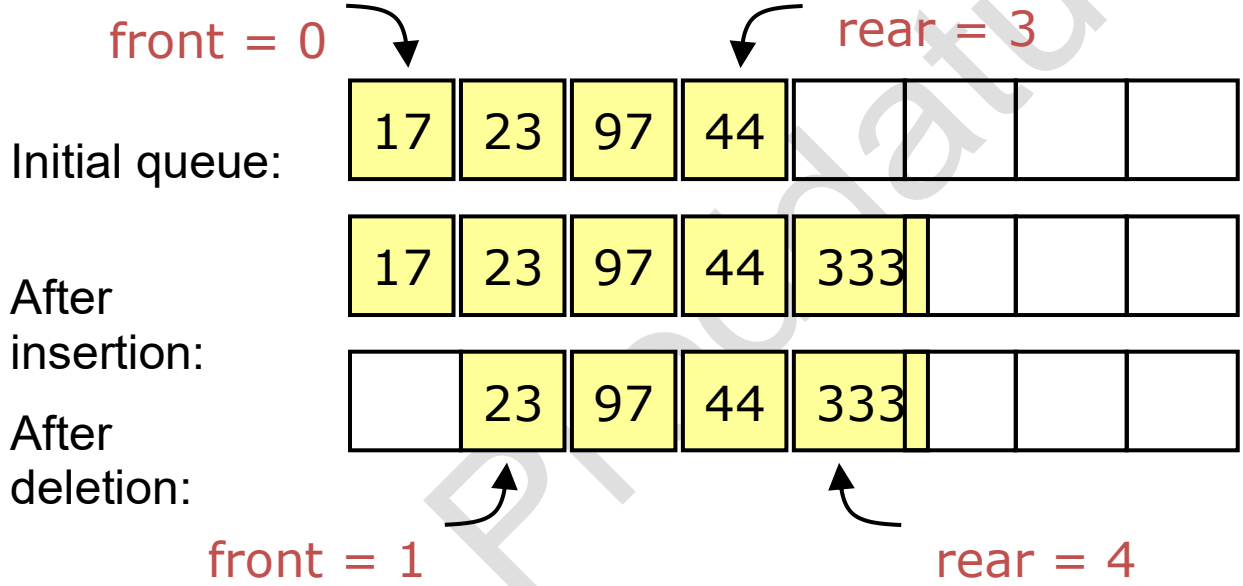
- Both people and data enter the queue at the rear and progress through the queue until they arrive at the front
- Once they are at the front of the queue, they leave the queue and are served.

3.16. Representation of Queue:

A queue is linear, sequential list of items that are accessed in the order First in First Out(FIFO). The first item inserted in a queue is also the first one to be accessed. This is accomplished by inserting at one end (the rear) and deleting from the other (the front).



To insert: put new element in location 4, and set rear to 4
To delete: take element from location 0, and set front to 1



Notice how the array contents "crawl" to the right as elements are inserted and deleted
This will be a problem after a while!

Array implementation of queues for Enqueue:

```
int rear=front=-1;
enqueue(int ele)
{
    if(rear==MAX-1)
    printf("queue overflow\n");
    else
    {
            if(front==-1)
                    front=0;
            rear=rear+1;
            queue[rear]=ele;
    }
```

```
}
```

Array implementation of queues for dequeue:

```
dequeue()
{
    if(front==-1||front>rear)
    printf("queue underflow\n");
    else
    {
            printf("element deleted from queue: %d",queue[front]);
            front=front+1;
    }
}
```

Array implementation of queues for display:

```
display()
{
    if(front==-1||front>rear)
    printf("queue is empty\n");
    else
    {
            for(i=front;i<=rear;i++)
            printf("%d",queue[i]);
  }
}
```

3.17. Operations (Insertion, Deletion, Searching):

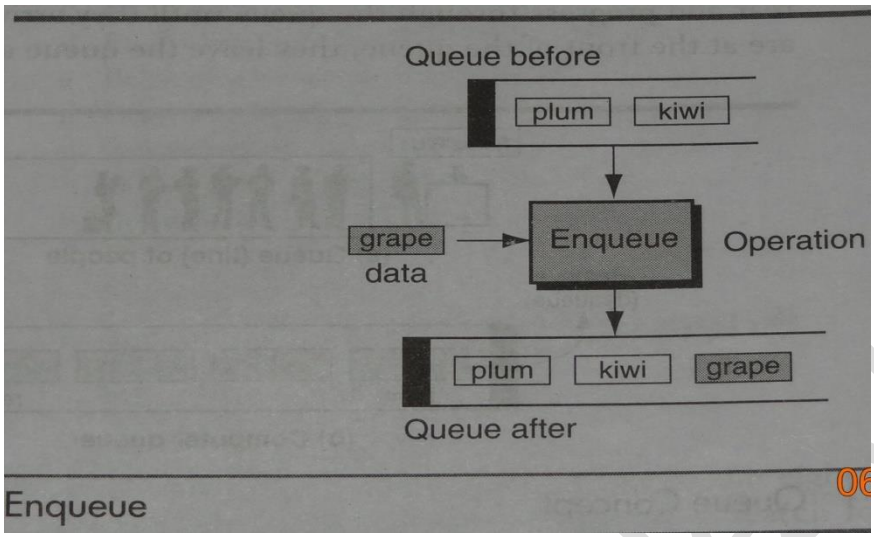
- Four basic operations:
    - Data can be inserted at the rear
    - Deleted from the front
    - Retrieved from the rear
    - Retrieved from the front
- Difference of stack and queue is implementation needs to keep track of the front and the rear of the queue; whereas stack only worry about one end the top.
- **Definition:** A queue is a list in which data can be inserted at one end, called the rear and deleted from the other end, called the front. It is a first in first out (FIFO) restricted data structure.

a. Enqueue

- The queue is insertion operation is known as enqueue
- After the data have been inserted into the queue, the new element becomes rear.
- The only potential problem with enqueue is running out of room for the data
- If there is not enough room for another element in the queue, the queue is an overflow state.

Enqueue inserts an element at the rear of the queue

Figure shows the enqueue operation

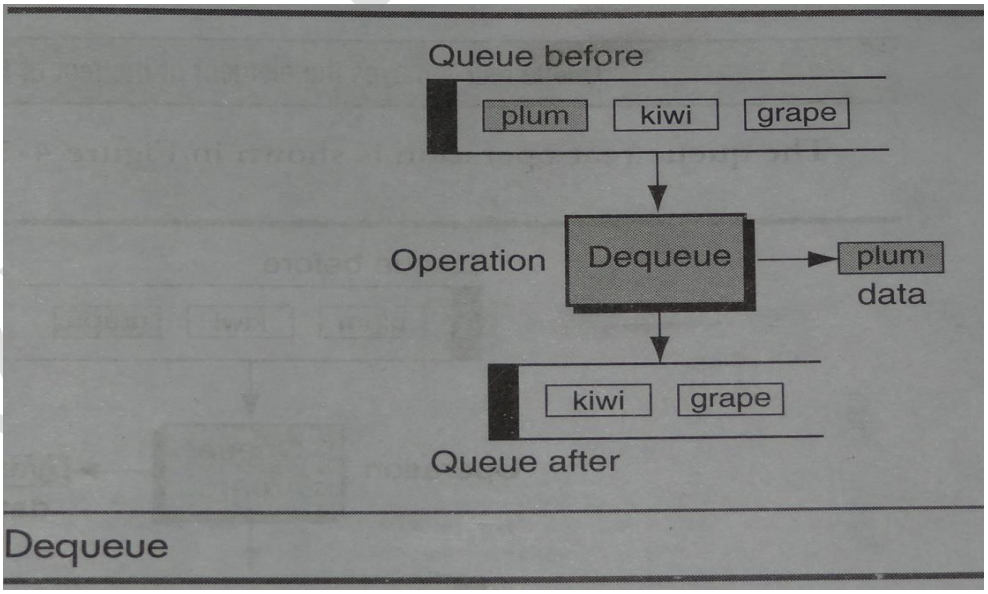


b. Dequeue

- The queue delete operation is known as dequeue
- The data at the front of the queue are returned to the user and removed from the queue
- If there are no data in the queue when a dequeue is attempted, the queue is in an underflow state

Dequeue deletes an element at the front of the queue

The dequeue operation is shown in fig.

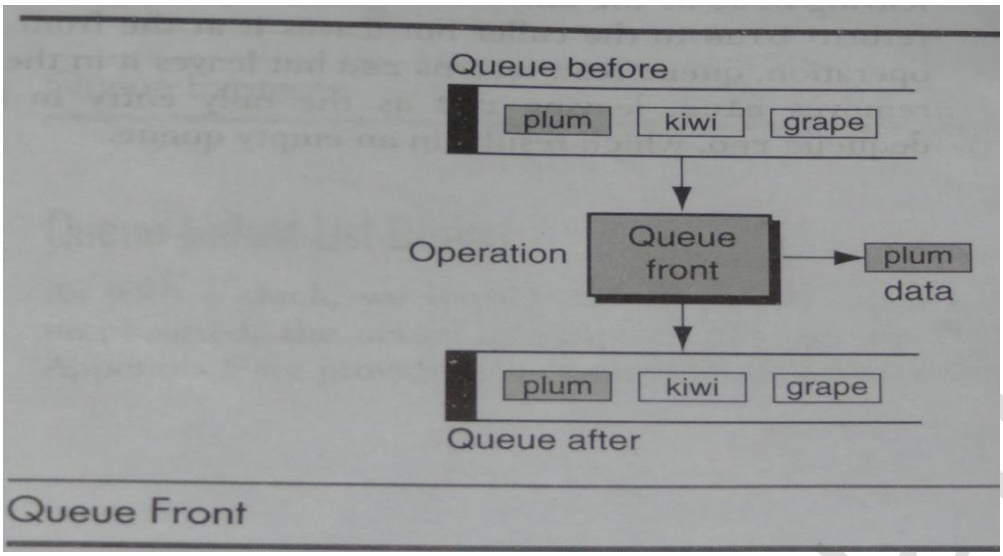


c. Queue front:

- Data at the front of the queue can be retrieved with queue front
- It returns the data at the front of the queue without changing the contents of the queue
- If there are no data in the queue when a queue front is attempted, then the queue is in underflow state.

Queue front retrieves the element at the front of the queue

- The queue front operation is as shown in figure

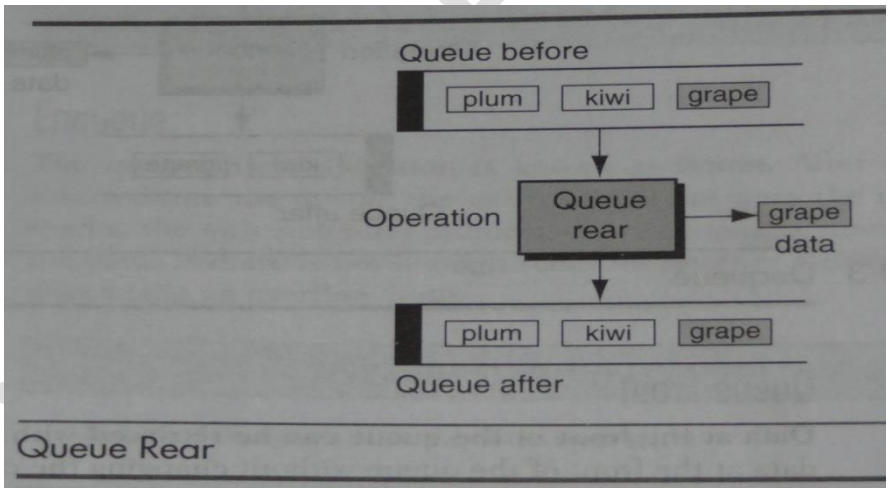


d. Queue Rear

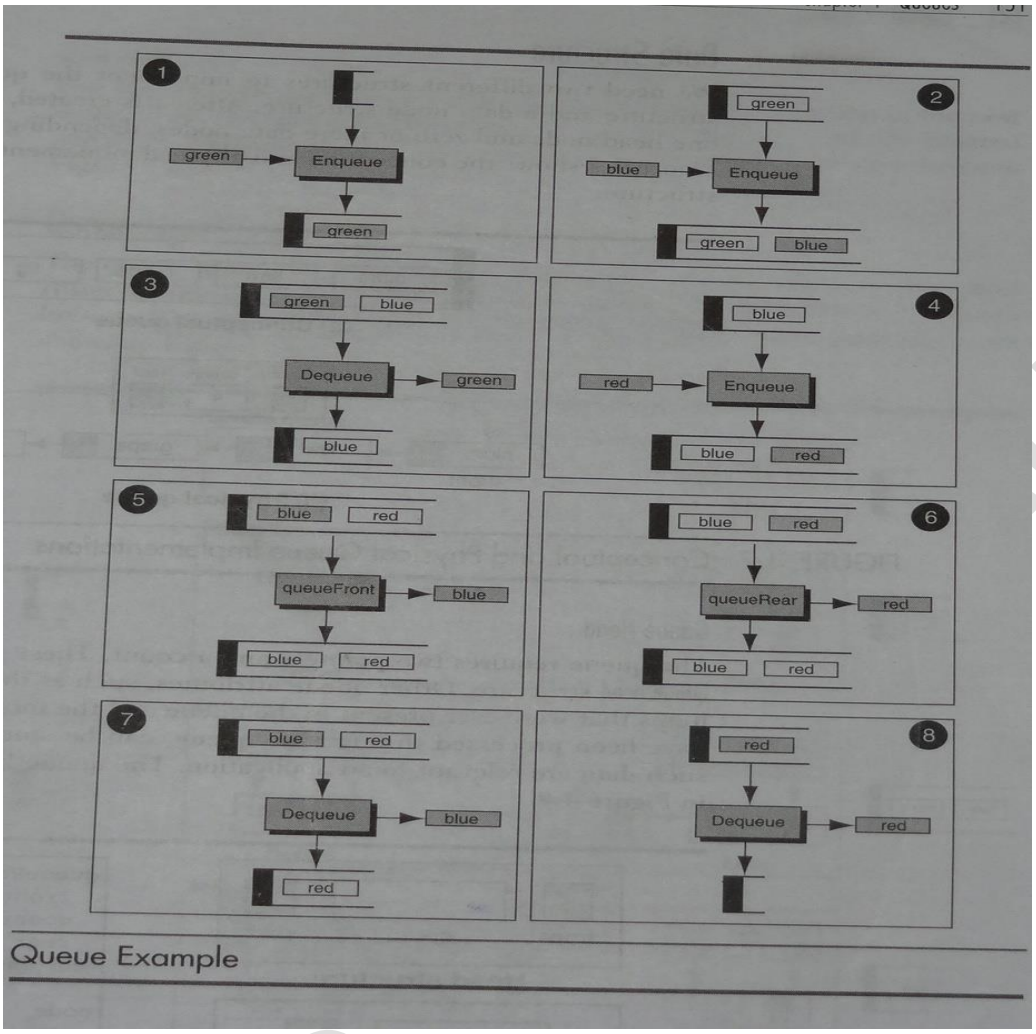
- The queue rear retrieves the data at the rear for the queue
- It is known as queue rear
- As with queue front, if there are no data in the queue when a queue rear is attempted, the queue is an underflow state

Queue rear retrieves the element at the rear of the queue

- The queue rear operation is shown in figure



Queue example



Queue Example

e. Searching:

To search for a specific item in the Queue is a sequential process. The comparison starts from the beginning of the list until the target item is found or until the end of the list is reached. Therefore, the
first matched item is returned.

i) Question and Answers: 2 Marks

1. Distinguish between linear and non linear data structures. (Nov/ Dec – 2019)

Ans: In a **linear data structure**, **data** elements are arranged in a **linear** order where each and every elements are attached to its previous and next adjacent. In a **non-linear data structure**, **data** elements are attached in hierarchically manner.

2. In how many ways a stack can be stored in memory? List them. (Nov/Dec – 2019)

Ans: A **stack can** be implemented in 3 **ways** simple array based, using dynamic **memory**, and Linked **list** based.

3. Find the value for the following postfix expression: 1 2 + 3 4 * -. (Nov/Dec – 2019)

Ans: -9

4. List the various types of linear and non linear data structures. (Nov/Dec – 2019)

Ans: Linear data structures,
Array, List. Stack, Queue.
Non Linear data structures,
Trees, Graphs.

5. PUSH(A), PUSH(B), POP(), PUSH(C), PUSH(D), POP(), POP() if these operations are performed on a empty stack, show the final contents of the stack. (Nov/Dec – 2019)

Ans: A

6. State the different ways of representing an expression. Give example for each. (May/Jun -2019)

Ans: Infix: The operator comes between operands
A+B
Prefix: The operator comes before operands
+AB
Postfix: The operator comes after operands
AB +

7. List the applications of stacks. (Jun/ Jul – 2019)

Ans: Infix to postfix conversion
Evaluation of postfix expression.

8. What are the basic operations that can be performed on stacks? (May/Jun – 2019, Dec - 2018)

Ans: Push: Insertion of an element into stack
Pop: Deletion of an element from stack
Stack Top: Retrieving an element from top of stack

9. Give any two operations of queues. (May/ Jun – 2019)

Ans: Enqueue: Insertion of an element into queue
Dequeue: Deletion of an element from queue

10. What is linear data structure? Discuss difference between FIFO and LIFO concepts. (Dec – 2018)

Ans: In a **linear data structure**, **data** elements are arranged in a **linear** order where each and every elements are attached to its previous and next adjacent.
FIFO – First In First Out means the element inserted first will be removed first.
LIFO – Last In First Out means the element inserted last will be removed first.

11. Define queue. Write some of the applications. (Nov/ Dec – 2018)

Ans: Queue is a restricted data structure where insertion is done at rear end and deletion is done at front end. The application of queue is, Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

12. Convert ((A + B) * C - (D - E)) $ (F + G) to postfix. (Nov/ Dec – 2018, Nov/Dec - 2017)

Ans: A B + C * D E - - F G + $

13. Assume that the operators +, -, x are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x, +, -. What is the postfix expression corresponding to the infix expression a + b x c - d ^ e ^ f? (May/ Jun – 2017)
Ans: a b c * + d e f ^ ^ -

14. Write the procedure to evaluate a post fix expression. (Dec – 2017)

Ans: 1.Read the expression from left to right.
2.If there comes an operand , push it into the stack.
3.If there comes an operator , pop operand 1 and operand 2 and then :
A. Push '('
B. Push 'operand 2'

C. Push ‘operator’
D. Push ‘operand 1’
E. Push ‘)’

15. What is overflow and underflow condition in a stack.

Ans: Overflow: It is an error condition, when you try to insert an element into a full stack,
Underflow: It is an error condition, when you try to delete an element from empty stack.

ii) MCQs

1.What is the logic for "Queue is Full" ?

a. **rear==SIZE-1** b. top==SIZE-1 c.rear==0 d. front++

2. front=rear=-1 implies

a. **Queue is Empty** b. Inserting an element c. Queue is Full d. None

3. What is the purpose of enqueue?

a. Display b. Deletion c. **Insertion** d. none

4. Evaluation of the postfix expression 53+62/*35*+ is

a. 71 b. **39** c. 100 d. 65

5. A ____ is a linear list in which all additions and deletions are restricted to one end called top.

a. List b. Queue c. **Stack** d. None

6. To remove an element from stack the operation is _____.

a. Push b. **Pop** c. Stack top d. None

7. _____ expression can be defined as an expression in which **all the operators are present after the operands**.

a. Prefix b. **Postfix** c. Infix d. None

8. A _____ is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.

a. Stack b. **Queue** c. List d. None

9. The queue delete operation is known as _____

a. Enqueue b. **Dequeue** c. Traversal d. Display

10. _____ adds an item at the top of the stack.

a. Stack top b. **Push** c. Pop d. Display

11. The _____ copies the item at the top of the stack; that is, it returns the data in top element to the user but does not delete it.

a. **Stack top** b. Push c. Pop d. display

12. In _____ the operators are written in between the operands they act on.

a. Prefix b. Postfix c. **Infix** d. None

13. The queue insertion operation is known as _____.

a. Dequeue b. **enqueue** c. Traversal d. Display

14. The queue is full and trying to insert an element then the state of queue is _____.

a. Underflow b. Normal c. **Overflow** d. Both A & C

15. The _____ retrieves the data at the rear of queue.

a. **Queue rear** b. Queue front c. Both A & B d. None

iii) Question and Answers: 10 Marks

1. Define Stack and Queue and explain in detail about the operations of stack using arrays.

Ans:

Stack: 2M

A Stack is a linear list in which all additions and deletions are restricted to one end called to ***top***.

Queue: 2M

A Queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.

Operations of stack using arrays:

Push: 2M

- Push adds an item at the top of the stack
- After the push, the new item becomes the top
- The only problem with this simple operation is that there is room for the new item
- If there is not enough room, the stack is an **overflow** state and the item cannot be added.
- Figure shows the push stack operation

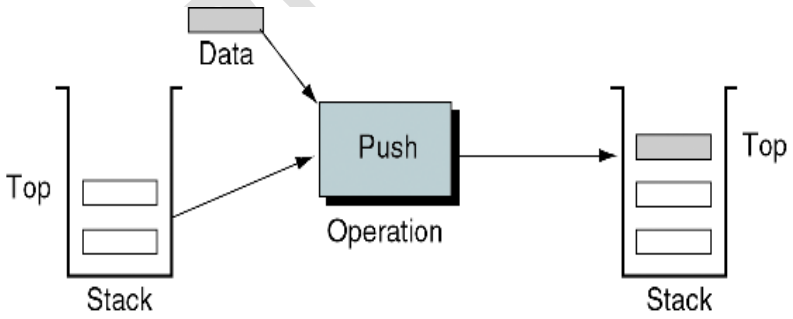


FIGURE Push Stack Operation

```
int stack[MAX];
int top=-1;
void push(int ele)
{
        if(top>MAX)
        {
                printf("stack is full\n");
                exit(0);
        }
        stack[++top]=ele;
}
```

b. Pop: 2M

- When we pop a stack, we remove the item at the top of the stack and return it to the user.
- After removing the top item, the next older item in the stack becomes the top.
- When the last item in the stack is deleted, the stack must be set to its empty state.
- If pop is called when the stack is empty, it is in an **underflow** state.
- The pop stack operation is shown in fig.

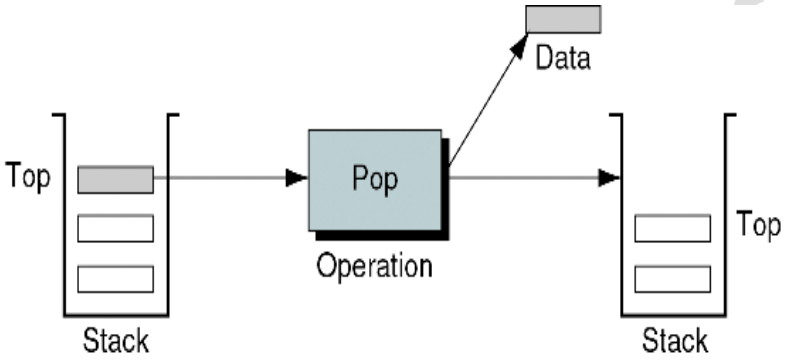


FIGURE : Pop Stack Operation

```
void pop()
{
                        if(top==-1)
                        {
                         printf("stack is empty\n");
                         exit(0);
                        }
                        a=stack[top--];
                        printf("the element popped is %d",a);
}
```

c. Stack top: 2M

- The stack top copies the item at the top of the stack; that is, it returns the data in top element to the user but does not delete it.
- This operation is nothing but as reading the stack top.
- Stack top can also result in underflow if the stack is empty.

The stack top operation is shown in figure.

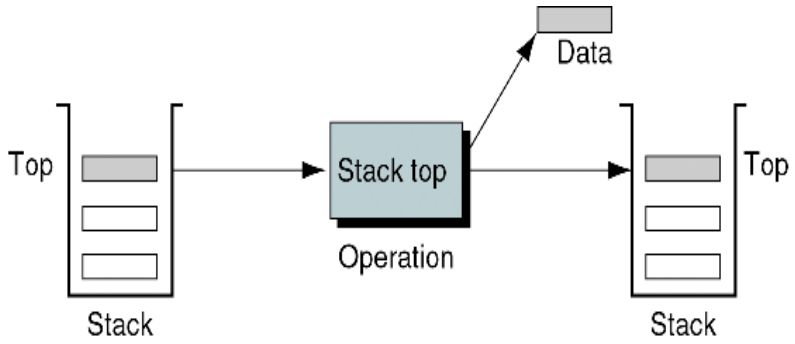


FIGURE Stack Top Operation

2. Write a procedure to convert any infix expression into postfix form. Using step by step procedure convert the following infix expression into postfix form: A+(B*C)-((D*E+F)/G). (Dec – 2019)

Ans:

Procedure: 5M

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Convert the following infix expression into postfix expression: 5M

A+(B*C)-((D*E+F)/G)

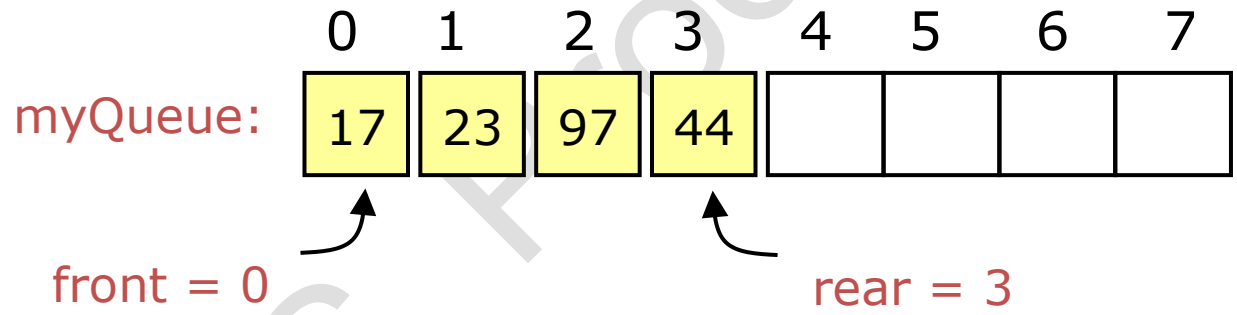
Input Expression	Stack	Postfix Expression
A		A
+	+	A
(	+(	AB
*	+(*	AB
C	+(*	ABC
)	+	ABC*
-	-	ABC*+
(	-(	ABC*+

(	-((	ABC*+
D	-((	ABC*+D
*	-((*	ABC*+D
E	-((*	ABC*+DE
+	-((+	ABC*+DE*
F	-((+	ABC*+DE*F
)	-(	ABC*+DE*F+
/	-/	ABC*+DE*F+
G	-/	ABC*+DE*F+G
	-	ABC*+DE*F+G/
		ABC*+DE*F+G/-

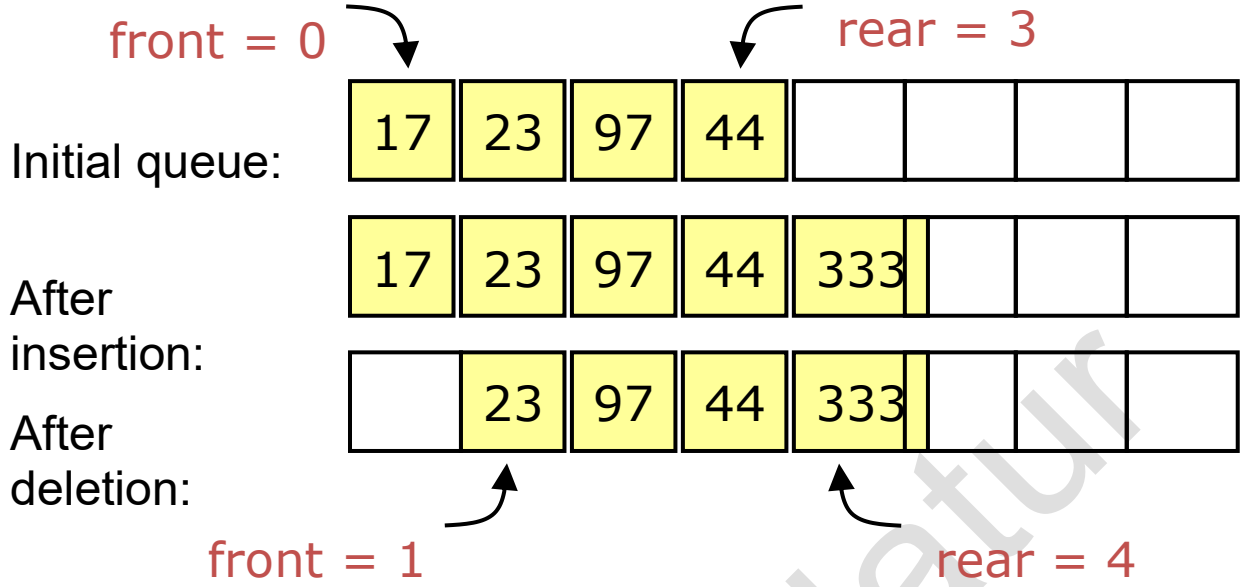
3. Give brief description about the queue storage using arrays. (Nov/Dec – 2019) 10M

Ans:

A queue is linear, sequential list of items that are accessed in the order First in First Out(FIFO). The first item inserted in a queue is also the first one to be accessed. This is accomplished by inserting at one end (the rear) and deleting from the other (the front).



To insert: put new element in location 4, and set rear to 4
To delete: take element from location 0, and set front to 1



Notice how the array contents "crawl" to the right as elements are inserted and deleted
This will be a problem after a while!

Array implementation of queues for Enqueue:

```
int rear=front=-1;
enqueue(int ele)
{
    if(rear==MAX-1)
    printf("queue overflow\n");
    else
    {
            if(front==-1)
                    front=0;
            rear=rear+1;
            queue[rear]=ele;
    }
}
```

Array implementation of queues for dequeue:

```
dequeue()
{
    if(front==-1||front>rear)
    printf("queue underflow\n");
    else
    {
            printf("element deleted from queue: %d",queue[front]);
            front=front+1;
```

```
    }
}
```

Array implementation of queues for display:

```
display()
{
    if(front==-1||front>rear)
    printf("queue is empty\n");
    else
    {
            for(i=front;i<=rear;i++)
                    printf("%d",queue[i]);
  }
}
```

4. How can we say stack follows LIFO principle? Justify your answer. (Nov/Dec – 2019) 10 M

Ans:

LIFO is short for "Last In First Out". The last element pushed onto the **stack** will be the first element that gets popped off. If you were to pop all of the elements from the **stack** one at a time then they would appear in reverse order to the order that they were pushed on.

5. Write an algorithm to convert infix expression into postfix expression. (Jun/July – 2019) 10M

Ans:

Algorithm: Convert infix to postfix

```
1 Create stack ( stack)
2 loop (for each character in formula )
        1 if ( Character is open paranthesis)
                1 pushStack (stack, character)
        2 elseif ( character is close paranthesis )
                1 popStack ( stack. Character )
                2 loop ( Character not open paranthesis )
                        1 concatenate character to postFixExpr
                        2 popStack (stack, character )
                3 end loop
        3 elseif ( character is operator )
                Test priority of token to token at top of stack
                1 popstack ( stack, topToken)
                2 loop ( not emptyStack (stack) AND priority ( character ) < = priority ( topToken ) )
                        1 popStack ( stack, tokenOut )
                        2 concatenate tokenOut to postFixExpr
```

```
                3 stackTop ( stack,, topToken )
            3 end loop
            4 pushStack ( stack, token )
        4 else
            character is operand
            1 Concatenate token to postFixExpr
        5 end if
3 end loop
        Input formula is empty. Pop stack to postfix
4 loop ( not emptyStack ( Stack ) )
        1 popStack ( stack, character )
        2 concatenate token to postFixExpr
5 end loop
6 return postFix
end inToPostFix
```

6. Write the steps to evaluate a postfix expression. Evaluate the following postfix expression using stack: 8 3 4 + - 4 9 3 / + * 2 ^ 3 +. (Nov/Dec – 2019)

Ans:

Steps to evaluate the postfix expression: 4M

1.Read the expression from left to right.

2.If there comes an operand, push it into the stack.

3.If there comes an operator, pop operand 1 and operand 2 and then :

A . Push '('

B . Push 'operand 2'

C . Push 'operator'

D . Push 'operand 1'

E . Push ')'

Evaluation of given expression: 6M

8 3 4 + - 4 9 3 / + * 2 ^ 3 +

symb	opnd1	opnd2	value	opndstk
8	-	-	-	8
3	-	-	-	8, 3
4	-	-	-	8, 3, 4
+	3	4	7	8, 7
-	8	7	1	1
4	8	7	1	1, 4

9	8	7	1	1, 4, 9
3	8	7	1	1, 4, 9, 3
/	9	3	3	1, 4, 3
+	4	3	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
^	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Unit - 4

Linked Lists – Singly linked list, dynamically linked stacks and queues, polynomials using singly linked lists, using circularly linked lists, insertion, deletion and searching operations, doubly linked lists and its operations, circular linked lists and its operations.

Linked Lists:

Linear List:

- It is a list in which operations, such as retrievals, iterations, changes and deletions can be done any where in the list i.e.,
  - At beginning
  - At middle
  - At end of the list
- We use many different types of general lists in our daily lives, like list of employees, student lists, and our favorite songs
- When we process our song list, we need to be able to search for a song, add a new song, or delete one way we gave away.
- We refer general linear lists as **lists.**

Types of linked lists:

- Singly linked lists
- Doubly linked lists
- Circularly linked lists

4.1. Singly Linked List:

Basic List Operations:

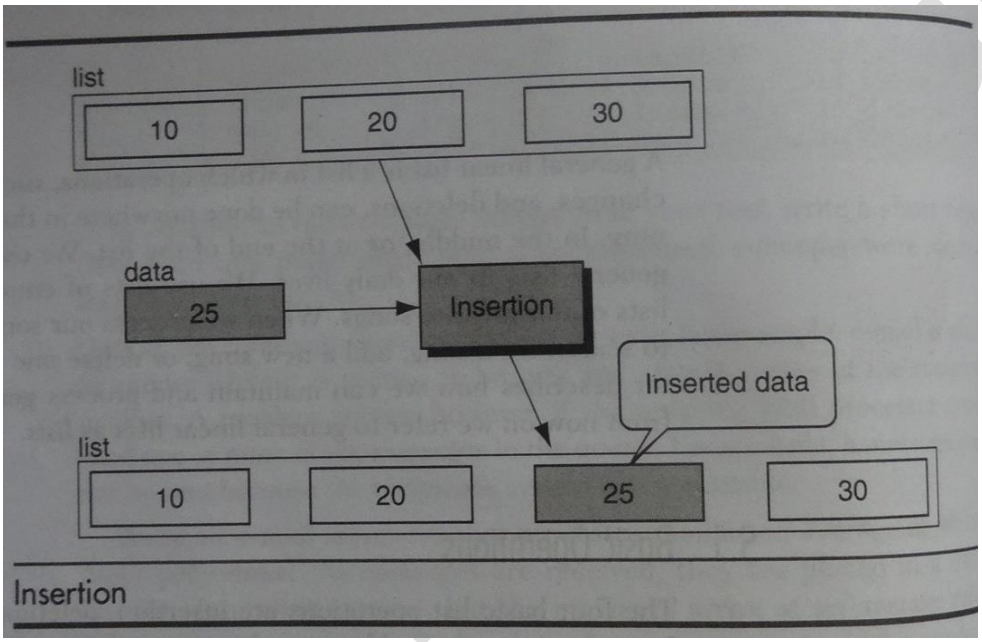
- Four basic list operations are
  - Insertion
  - Deletion
  - Retrieval
  - traversal
- **Insertion**: it is used to add a new element to the list
- **Deletion**: it is used to remove an element from the list
- **Retrieval**: it is used to get the information related to an element without changing the structure of the list
- **Traversal:** it is used to traverse the list while applying a process to each element

Insertion:

- List insertion can be ordered or random
- **Ordered lists** are maintained in sequence according to the data or, when available a key that identifies the data
- A **key** is one or more fields within a structure that identifies the data
- Examples of keys are

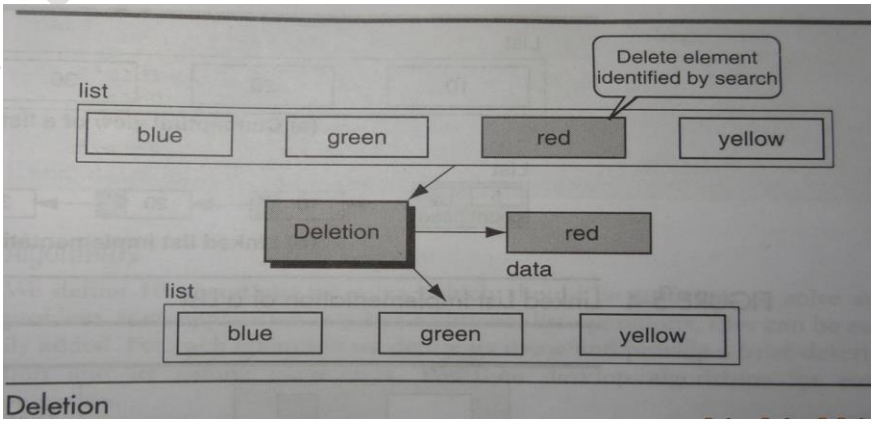
- Social security number
- Universal product code

- In **Random lists** there is no sequential relationship between two elements
- There are no restriction on inserting data into a random lists, computer algorithms generally insert data at the end of the list
- Thus random lists are sometimes called **chronological lists**
- Data must be inserted into ordered lists so that the ordering of the list is maintained
- To determine where to insert e use a **search algorithm**
- We may insert at begin, middle or end of the list
- Mostly the data are inserted somewhere in the middle of the list



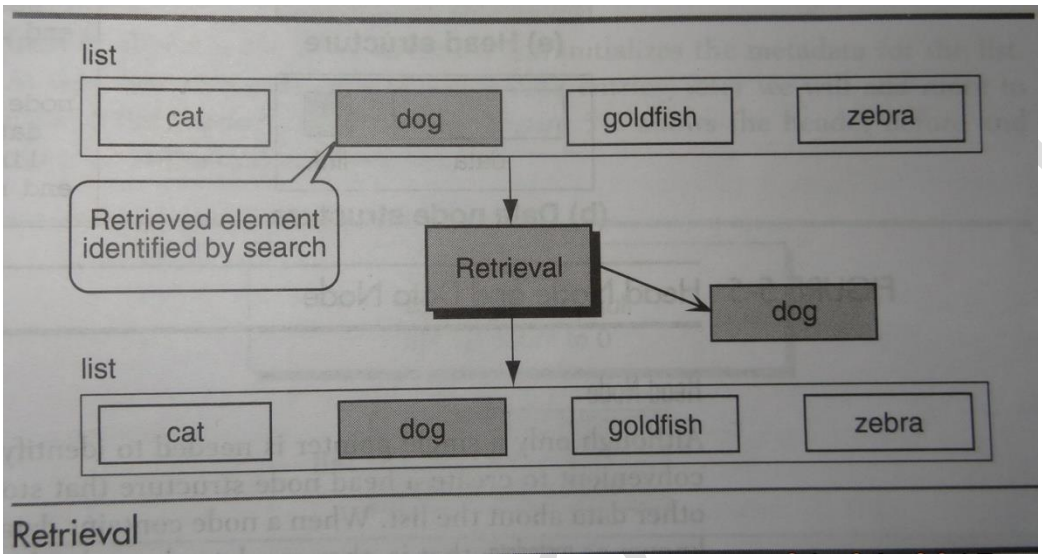
Deletion:

- Deletion from a list requires that the list be searched to locate the data being deleted
- Once located, the data are removed from the list
- Fig. depicts a deletion from a list



Retrieval:

- List retrieval requires that data be located in a list and presented to the calling module without changing the contents of the list
- Both insertion and deletion, a search algorithm can be used to locate the data to be retrieved from a list.
- Retrieving data from a list is shown in fig.

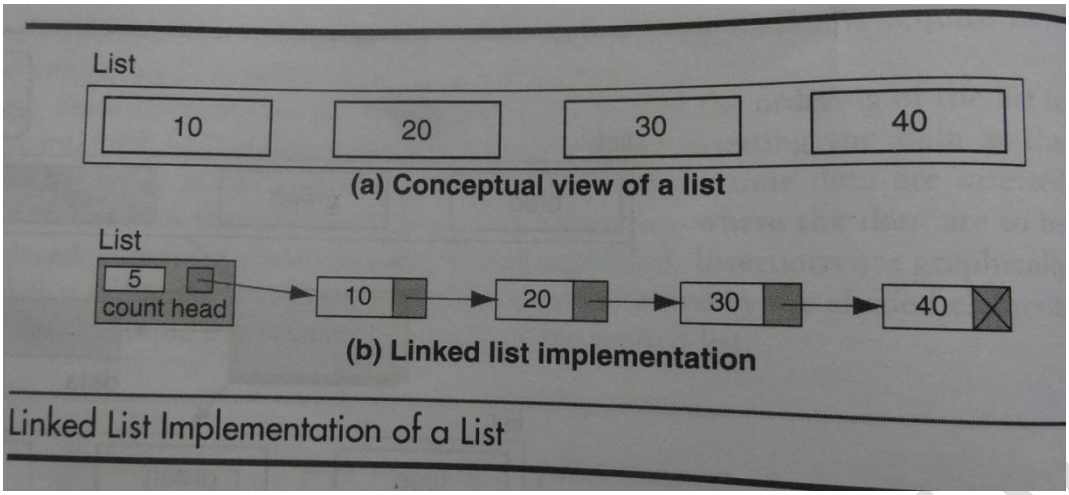


Traversal:

- It processes each element in a list in sequence
- It requires a **looping algorithm** rather than a search
- Each execution processes one element in the list
- The loop terminates when all elements have been processed.

Singly linked implementation

- Several data structures can be used to implement a list, we use a **linked list**
- A **linked list** is a good structure for a list because data are easily inserted and deleted at the beginning, in the middle, or at the end of the list
- Fig shows conceptual view of a list and its implementation as a linked list



Linked List Implementation of a List

Head node:

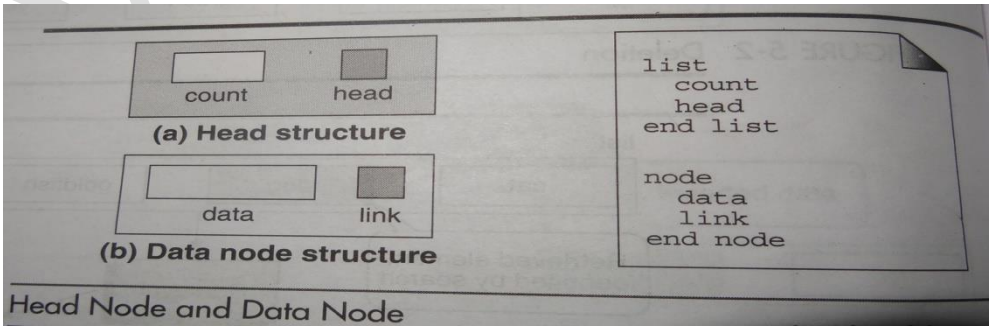
- Only a single pointer is needed to identify the list
- It stores the head pointer and other data about the list
- When node contains data about a list, the data are known as metadata; that is, they are data about the data in the list
- For ex, the head structure in fig contains one piece of metadata: count, an integer that contains the number of nodes currently in the list

Data node:

- The data type for the list depends entirely on the application
- A typical data type is shown below

```
data
                    key
                    field 1
                    field 2
                    ---
        ---
        filed N
end data
```

- We include a key field for application that require searching by key. The datatypes must be tailored to the application being created



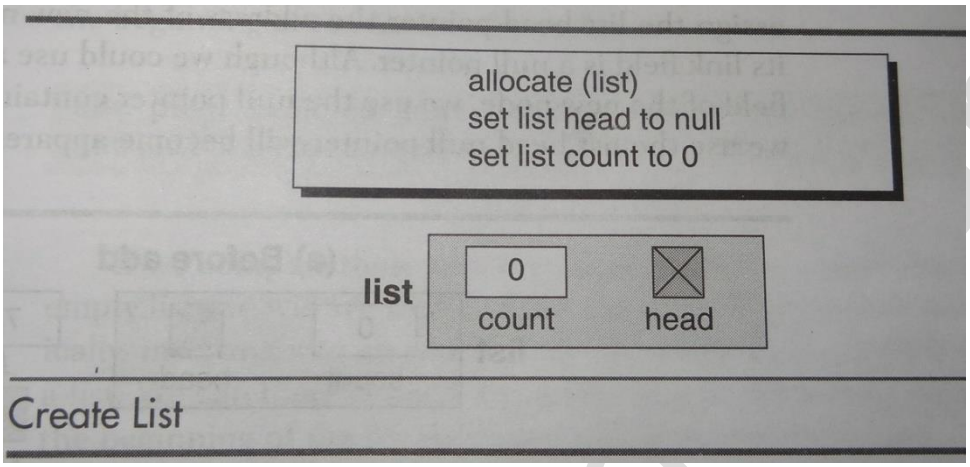
Head Node and Data Node

Algorithms:

- Ten operations for a list, would be sufficient to solve any problem
- We may easily extend these operations

1. **Create list:**

- Create list allocates the head structure and initializes the metadata for the list
- At this time there are only two metadata entries; later we will add more to expand the capabilities of the list
- Fig shows the header before and after it is initialized by create list



- The pseudo code for create list is shown in algorithm

Algorithm: create list

Algorithm createlist(list)
Initializes metadata for list
Pre list is metadata structure passed by reference
Post metadata initialized

1. allocate(list)
2. set list head to NULL
3. set list count to 0

end **createlist**

---

2. **Insert Node:**

- Insert node adds data to a list
- We need only its logical predecessor to insert a node in the list
- Given the predecessor, there are three steps to the insertion
  1. Allocate memory for the new node and move data to the node
  2. Point the new node to its successor
  3. Point the new node's predecessor to the new node
- These steps appear to be simple, but a little analysis is needed
- To insert a node into a list, we need to know the location of the node that precedes the new node
- This node is identified by a predecessor pointer that can be in one of two states
  1. It can contain the address of a node or it can be null. When the predecessor pointer is null, it means that there is no predecessor to the data being added.

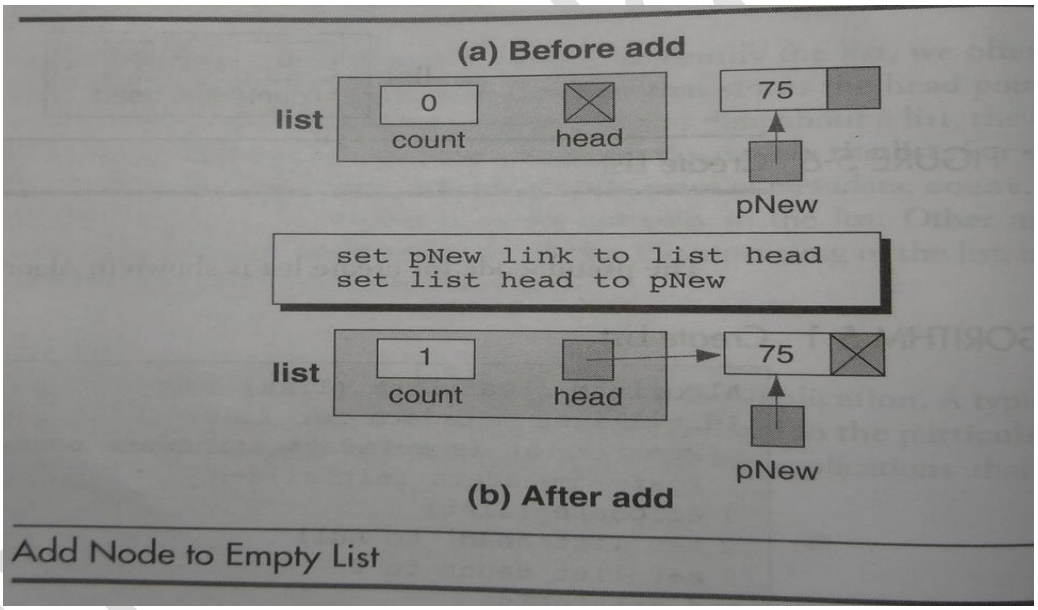
Conclusion is that we are either **adding to an empty list or are at the beginning of the list**

2. If the predecessor is not null, we are adding somewhere after the first node i.e., in the middle of the list
3. Or at the end of the list

- **Insert into empty list:**
  - when the head pointer of the list is null, the list is empty
  - This is shown in fig.
  - All is necessary to add a node to an empty list is to assign the list head pointer the address of the new node and make sure that its link filed is a null pointer
  - We could use a constant to set the link field to the new node, we use the null pointer contained in the list head
  - The pseudo code statements to insert a node an empty list are shown here

    Set pnew link to list head (Null Pointer)

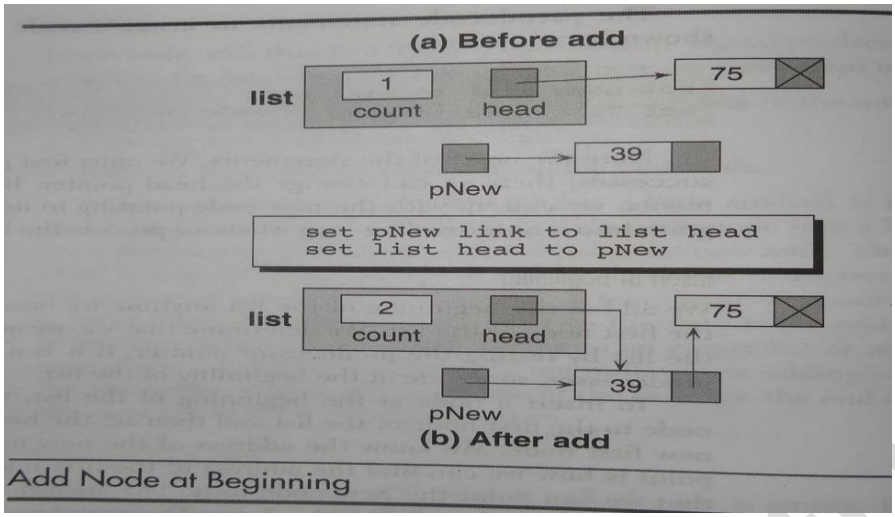
    Set list head to pnew (First node)
  - If we reverse these statements, we end up with the new node to point to itself



- **Insert at beginning**
  - If it is a null pointer, there is no predecessor, so we are at the beginning of the list
  - To insert a node at the beginning of the list, we simply point the new node to the first node of the list and then set the head pointer to point to the new first node

    Set pnew link to list head (to current first node)

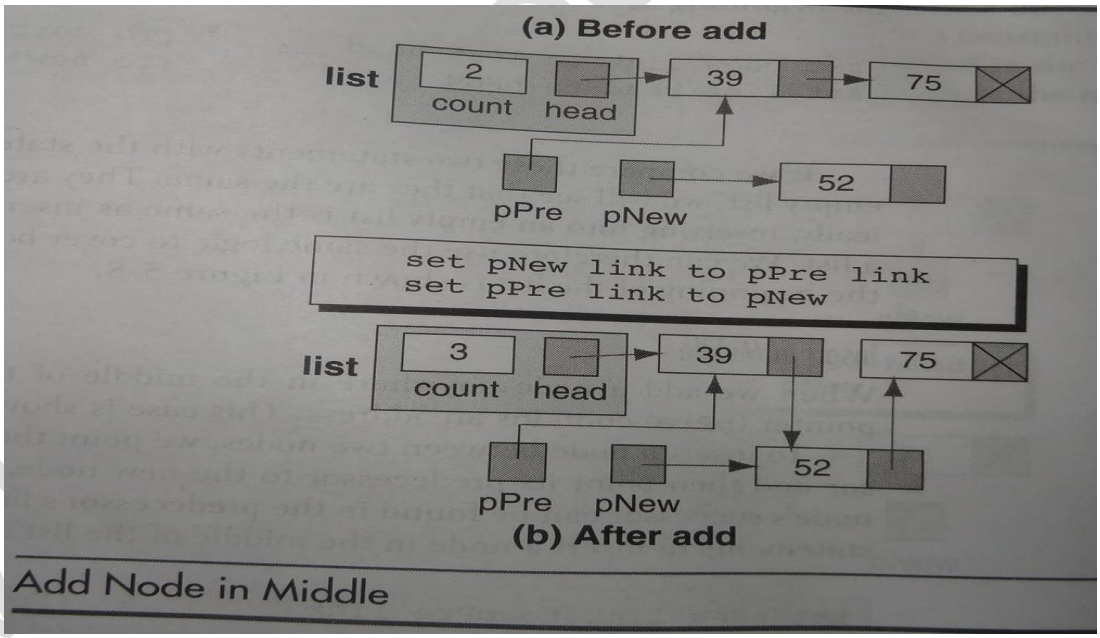
    Set list head to pnew (to new first node
  - We can think this both situations(at beginning and empty list) as adding at the beginning of the list as the code is logically same for both.



- **Insert at middle**
  - When we add a node anywhere in the middle of the list, the predecessor pointer (ppre) contains an address
  - This case is shown in fig
  - The pseudo code statements to insert a node in the middle of the list are shown

    Set pnew link to ppre link

    Set ppre link to pnew



- **Insert at end**
  - When we are adding at the end of the list, we only need to point the predecessor to the new node and to set the new nodes link field to a null pointer.

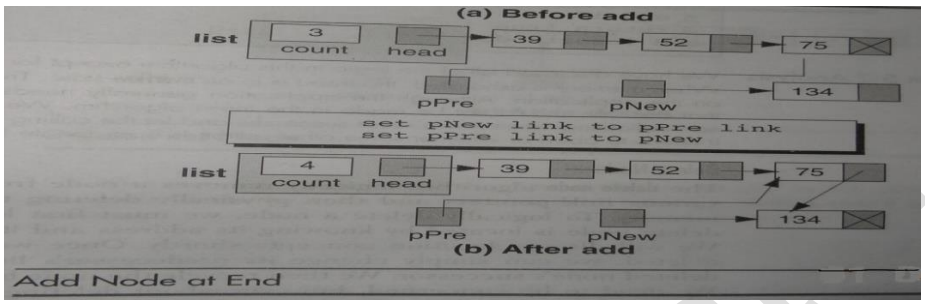
    Set pnew link to null pointer

    Set ppre link to pnew (predecessor to new)
  - We know that the last node in the list has a null link pointer

- We use this pointer rather than a null pointer constant, the revised code becomes same as the code for inserting in the middle
- It is as shown

  Set pnew link to ppre link (new to null)

  Set ppre link to pnew (predecessor to new)
- Fig shows the logic



Insert node algorithm

- The algorithm is complete, it returns a Boolean true if it was successful and false if there was no memory for the insert

Algorithm: Insert node

Algorithm insertnode(list, ppre, dataIn)

Inserts data into a new node in the list

Pre list is metadata structure to a valid list
ppre is pointer to data's logical predecessor
datain contains data to be inserted

Post data have been inserted in sequence

Return true if successful, false if memory overflow

1. Allocate (pnew)
2. Set pnew data to datain
3. if(ppre null)

adding before first node or empty list

   1. set pnew link to list head
   2. set list head to pnew

4. else

adding in middle or at end

   1. set pnew link to ppre link
   2. set ppre link to pnew

5. end if
6. return true

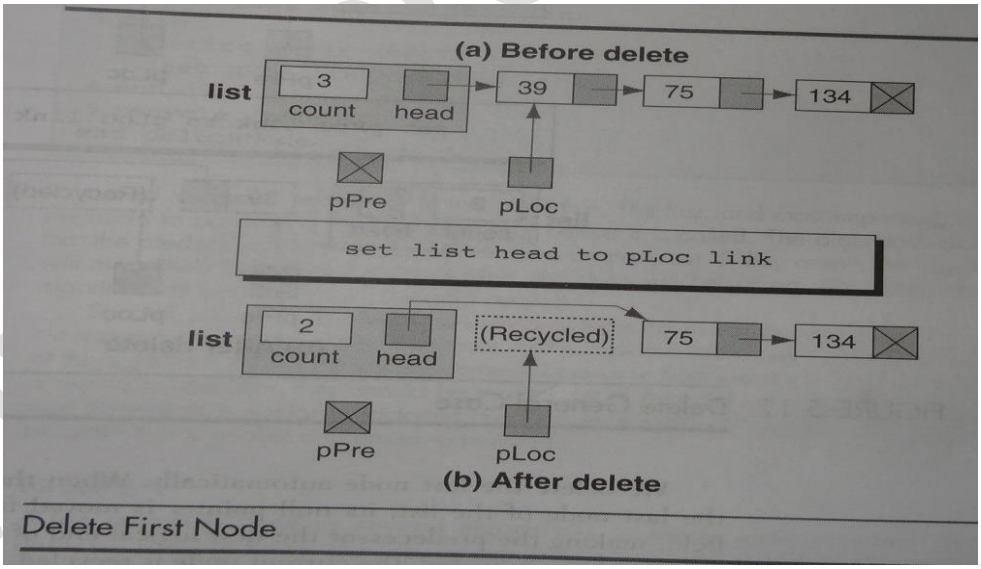
end **insert node**

3. Delete node

- The delete node algorithm logically removes a node from the list by changing various link pointers and then physically deleting the node from dynamic memory

- To logically delete a node , we must first locate the node itself
- A delete node is located by knowing its address and its predecessors address
- Once we locate we simply change it's predecessor's link field to point to the deleted node's successor
- We then recycle the node back to dynamic memory
- Deleting the only node results in an empty list. So in this case the head is set to a null pointer
- The delete also can have,
  - Delete the only node
  - The first node
  - A node in the middle of the list
  - Or the last node of a list
- The above four can be made as:
  - Delete the first node
  - Delete any other node
- In all the cases the node to be deleted is identified by a pointer ploc

- **Delete first node**
  - When we delete the first node, we must reset the head pointer to the first node's successor and then recycle the memory for the deleted node
  - If the predecessor is a null pointer then we say we are deleting the first node
  - This is shown in fig



  - The pseudo code is

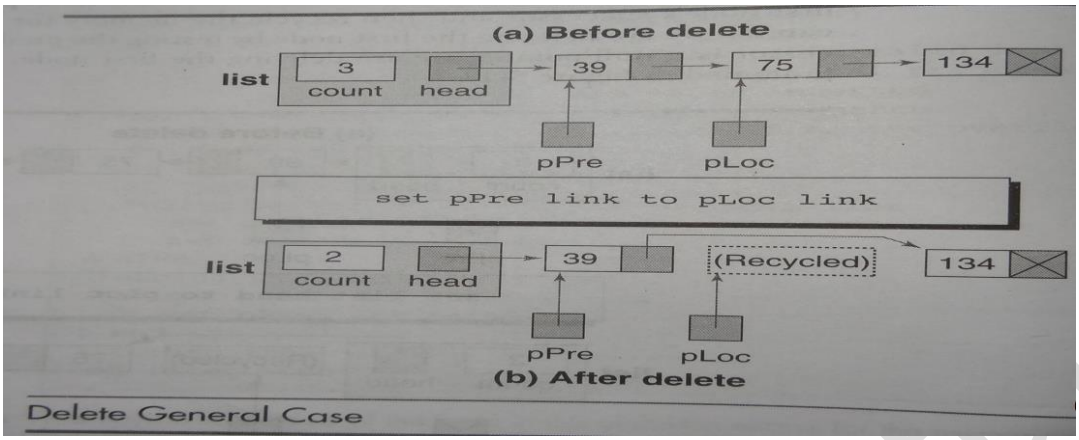
    Set list head to ploc link

    Recycle(ploc)
  - This logic can be applied only when we are deleting the only node in the list

General delete case:

- The same logic can be applies to delete any node either in the middle or at the end of the list

- i.e., the predecessor node to the successor of the node being deleted
- The logic is as shown in fig



- The pseudo code is

Set ppre link to ploc link

Recycle (ploc)

Delete node algorithm

Algorithm: list delete node

Algorithm deletenode(list, ppre, ploc, dataout)

Deletes data from list & returns it to calling module

Pre list is metadata structure to a valid list

pre is a pointer to predecessor node

ploc is a pointer to node to be deleted

dataout is variable to receive deleted data

Post data have been deleted and returned to caller

1. Move ploc data to dataout
2. if(ppre null)

deleting first node

   1. set list head to ploc link

3. else

deleting other nodes

   1. set ppre link to ploc link

4. end if
5. recycle(ploc)

end deletenode

---

4. List search

- A list search is used by several algorithms to locate data in a list
- To insert data, we need to know the logical predecessor
- To delete data, we need to find the node to be deleted and its logical predecessor
- To retrieve data from a list, we need to search the list and find the data
- We must use a sequential search because there is no physical relationship among the nodes

- To search a list on a key, we need a key field
- For **simple lists the key and the data can be the same field**
- For complex structures, we need a separate key field
- Data node structure may be

    data
        key
        field 1
        field 2

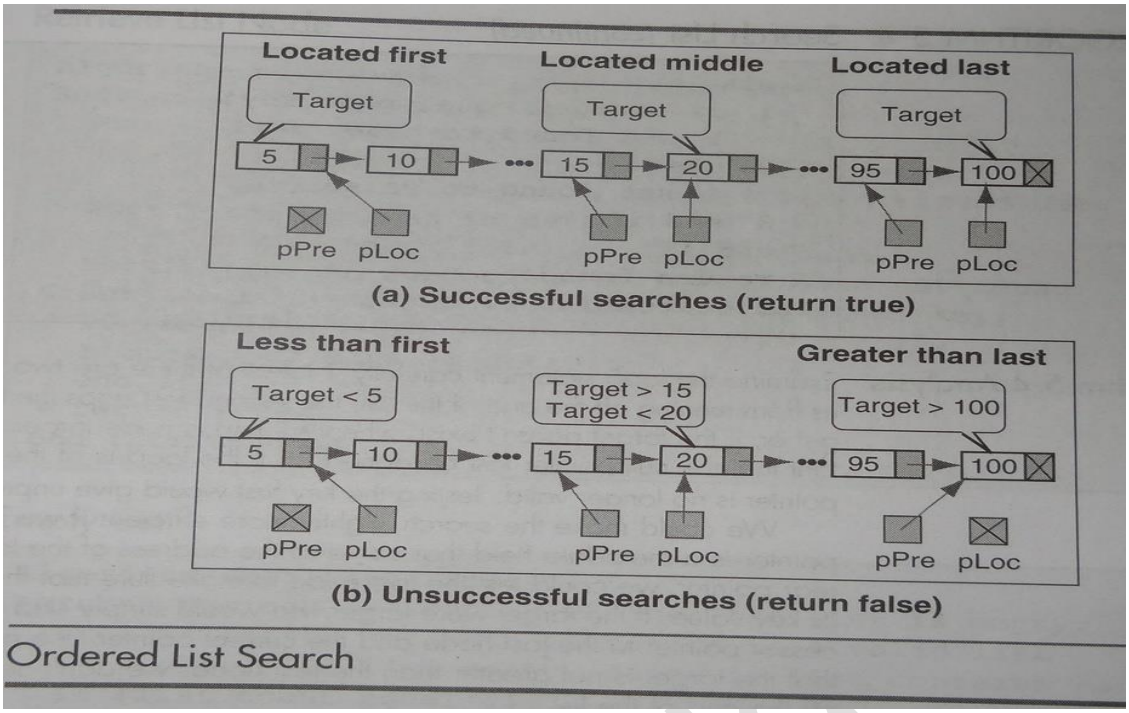
        -----
        field N
    end data

- **Sequential search in ordered list** is simply called as **ordered list search**
- Given a target key, the ordered list search attempts to locate the requested node in the list
- If a node in the list matches the target value, the search returns true; if there are no key matches it returns false
- The predecessor and current pointers are set according to the rules in

Condition	pPre	pLoc	Return
Target < first node	Null	First node	False
Target = first node	Null	First node	True
First < target < last	Largest node < target	First node > target	False
Target = middle node	Node's predecessor	Equal node	True
Target = last node	Last's predecessor	Last node	True
Target > last node	Last node	Null	False

List Search Results

- We start at beginning and search the list sequentially until the target value is no longer greater than the current nodes key
- At this point the target value is either less than or equal to the current nodes key while the predecessor is pointing to the node immediately before the current node
- And we set if it is equal to current node to true or false if it is less and terminate the search



Algorithm: searchlist

Algorithm searchlist(list, ppre, ploc, target)
Searches list and passes back address of node containing target and its logical predecessor
Pre list is metadata structure to a valid list
ppre is pointer variable for predecessor
ploc is pointer variable for current node
Post ploc points to first node with equal or greater key or null if target > key for last node
ppre points to target node smaller than key or null if target<key of first node or null if target<key of first node

1. Set ppre to null
2. Set ploc to list head
3. loop(ploc not null and target >ploc key)
    1. Set ppre to ploc
    2. Set ploc to ploc link
4. end loop
5. if (ploc null)
    1. Set found to false
6. else
    1. if(target equal ploc key)
        1. set found to true
    2. else
        1. set found to false
    3. end if
7. end if
8. return found

end **searchlist**

5. Retrieve node

- Retrieve node uses search node to locate the data in the list
- If data is found it moves the data to output area in the calling module and returns true
- If they are not found, it returns false
- The pseudo code is shown in algorithm

Algorithm: Retrieve node

```
Algorithm retrievenode(list, key, dataout)
Retrieves data from a list
Pre     list is metadata structure to a valid list
        key is target of data to be retrieved
        dataout is variable to receive retrieved data
Post    data placed in dataout
        -or- error returned if not found
        1. set found to searchlist(list, ppre, ploc, key)
        2. if(found)
                1. move ploc data to dataout
        3. end if
        4. return found
end retrievenode
```

6. Empty list

- Processing logic depends on there being data in a list
- We determine empty list by a simple module that returns a Boolean indicating that there are data in the list or that it is empty

Algorithm: Emptylist

```
Algorithm emptylist(list)
                        Returns Boolean indicating whether the list is empty
                        Pre     list is metadata structure to a valid list
                        Return true if list empty, false if list contains data
    1. If(list count equal 0)
        1. return true
    2. else
        1. return false
end emptylist
```

7. Full list

- Full list appears to be simple as empty list, but it is complex algorithm to implement
- Very few language come s with this capability to test how much memory is left in dynamic memory bit c does not
- The pseudo code is shown in algorithm

Algorithm: Full list

Algorithm fulllist(list)

Returns Boolean indicating whether or not the list is full
Pre list is a metadata structure to a valide list
Return false if room for new node; true if memory full

1. if(memory full)
    1. return true
2. else
    1. return false
3. end if
4. return true

end **fulllist**

---

8. List count

- List count is also one line module, the calling module has no direct access to the list structure for that it is necessary

Algorithm: List count

Algorithm listcount(list)

Returns integer representing number of nodes in list
Pre list is metadata structure to a valid list
Return count for number of nodes in list

1. return (listcount)

end listcount

---

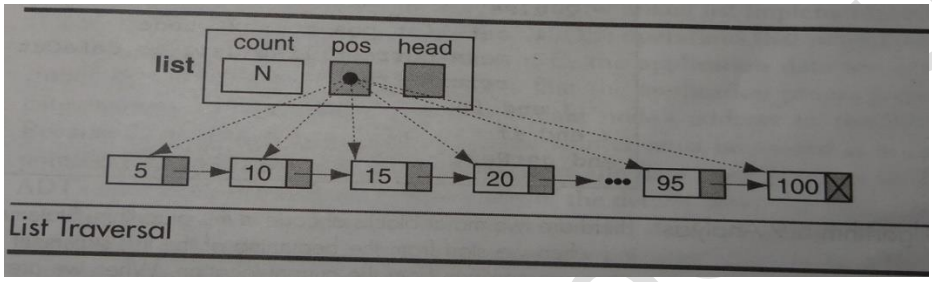
9. Traverse list

- Algorithms that traverse a list start at the first node and examine each node in succession until the last node has been processed
- This logic is used by different types of algorithms
    - Such as changing a value in each node
    - Printing the list
    - Summing a field in the list or
    - Calculating the average of a field
- Any application that requires the entire list to be processed uses a traversal
- To traverse the list, we need a walking pointer, a pointer that moves from node to node as each element is processed
- Assuming a list with a head structure, the following pseudo code uses a walking pointer to traverse the list
- Each loop modifies the pointer to move to the next node in sequence as we traverse the list

```
Set pwalker to list head
loop(more nodes)
    process(pwalker data)
set pwalker to next link
```

end loop

- We begin by setting the walking pointer to the first node in the list
- Then using a loop, we continue until all of the data have been processed.
- Each loop calls a process module and passes it the data and then advances the walking pointer to the next node
- When the last node has been processed, the loop terminates
- We have two possible approaches in designing the traverse list implementation
  1. User controls the loop, calling traverse to get the next element in the list
  2. Traverse module controls the loop calling user supplied algorithm to process the data
- We implement the first option as it provides flexibility
- Fig. shows a graphic representation of a list traversal



- We need to remember where we are in the list from one call to the next, so we need to add a current position pointer to the next as the traversal algorithm is called
- Each call also need to know whether we are starting from the beginning of the list or continuing from the last node processed
- This information is shown in algorithm

Algorithm: Traverse list

Algorithm Traverse(list, fromwhere, dataout)

Traverse a list. Each call returns the location of an element in the list

Pre list is a metadata structure to a valid list
fromwhere is o to start at the first element
dataout is reference to data variable

Post dataout contains data and true returned – or – if end of list, returns false

Return true if next element located false if end of list

1. if(emptylist)
   1. return false
2. if(fromwhere is beginning)

start from list

   1. set list pos to list head
   2. move current list data to dataout
   3. return true
3. else

continue from pos

1. if(end of list)

end of list

1. return false
2. else

1. set list pos to next node
2. move current list data to dataout
3. return true

3. end if

4. end if

end traverse

---

10. Destroy list

- When a list is no longer needed, then the list should be destroyed
- Destroy list deletes any nodes still in the list and recycles their memory
- It then sets the metadata to a null list condition
- The pseudo code for destroy list is shown in algorithm

Algorithm: Destroy list

Algorithm destroylist(plist)

Deletes all data in list
Pre list is metadata structure to a valid list
Post all data deleted

1. loop(not at end of list)
   1. set list head to successor node
   2. release memory to heap
2. end loop

no data left in list. reset metadata

3. set list pos to null
4. set list count to 0

end destroylist

Singly linked list program in C

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
int data;
struct node *link;
}dnode;
typedef struct
{
```

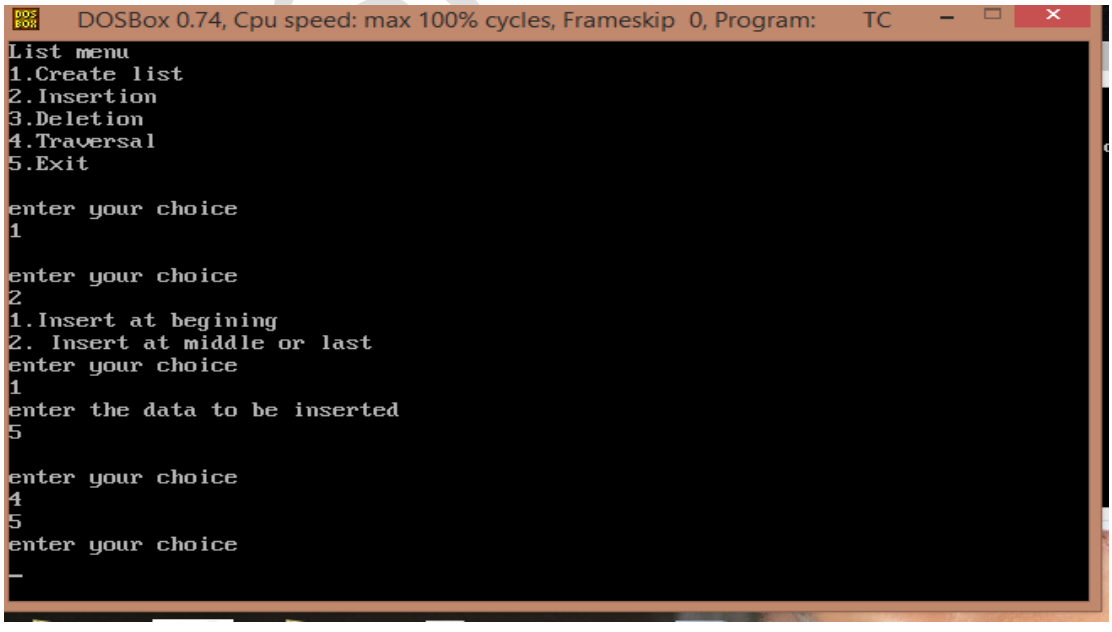
```
int count;
dnode *head;
dnode *rear;
}list;
dnode *pnew,*ppre,*clist,*ploc;
list *plist;
void createlist()
{
plist=(list*)malloc(sizeof(list));
plist->head=NULL;
plist->rear=NULL;
plist->count=0;
}
void insert()
{
pnew=(dnode*)malloc(sizeof(dnode));
printf("enter the data to be inserted\n");
scanf("%d",&pnew->data);
pnew->link=NULL;
if(ppre==NULL)
{
pnew->link=plist->head;
plist->head=pnew;
if(plist->count==0)
plist->rear=pnew;
}
else
{
pnew->link=ppre->link;
ppre->link=pnew;
if(pnew->link==NULL)
plist->rear=pnew;
}
plist->count++;
}
int search()
{
int targ;
```

```
ppre=NULL;
ploc=plist->head;
if(plist->count==0)
printf("list is empty\n");
else
{
printf("enter the data\n");
scanf("%d",&targ);
while(ploc->link!=NULL&&targ>ploc->data)
{
ppre=ploc;
ploc=ploc->link;
}
if(ploc==NULL)
{
printf("Data not found in the list");
return 0;
}
else
{
if(targ==ploc->data)
{
return 1;
}
else
{
printf("element not found\n");
return 0;
}
}
}
}
void del()
{
if(ppre==NULL)
plist->head=ploc->link;
else
ppre->link=ploc->link;
```

```
if(ploc->link==NULL)
plist->rear=ppre;
plist->count--;
free(ploc);
}
void traversal()
{
int i;
if(plist->count==0)
printf("list is empty\n");
else
{
clist=plist->head;
for(i=plist->count;i>0;i--)
{
printf("%d\t",clist->data);
clist=clist->link;
}
}
}
void main()
{
int ch;
clrscr();
printf("List menu\n");
printf("1.Create list\n2.Insertion\n3.Deletion\n4.Traversal\n5.Exit\n");
do
{
printf("\nenter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:createlist();
    break;
case 2: printf("1.Insert at begining\n2. Insert at middle or last\n");
    printf("enter your choice\n");
    scanf("%d",&ch);
    switch(ch)
```

```
    {
    case 1: ppre=NULL;
                                insert();
                                break;
    case 2:if(search())
          {
                                insert();
          }
          break;
    }
    break;
case 3: traversal();
    search();
    del();
    traversal();
    break;
case 4:traversal();
    break;
case 5:exit(0);
}
}while(ch!=5);
getch();
}
```

Output:



4.2. Dynamically linked stacks and queues:

Linked representations of stacks:

Several data structures can be used to implement a stack. Here we implement a stack as a linked list.

Data structure:

- To implement linked list stack, we need two different structures,
  - Head node
  - Data node
- The head structure contains metadata that is, data about data – and a pointer to the top of the stack.
- The data structure contains data and a link pointer to the next node in the stack.
- The conceptual and physical implementations of the stack are shown in fig.

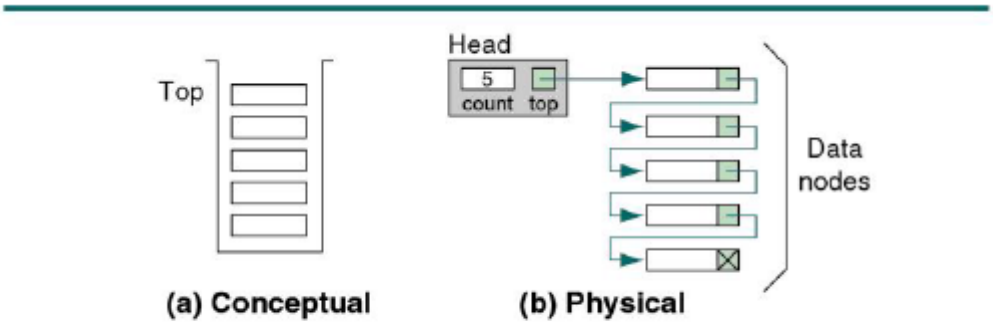


FIGURE Conceptual and Physical Stack Implementations

Stack Head node:

- Head stack requires only two attributes:
  - Top pointer
  - Count the number of elements in the stack
- These two elements are placed in a structure
- These two metadata items allow the user to determine average number of items processed through the stack in a given period. This is used only if a statistic were required for some reason.
- The basic head structure is shown in figure.

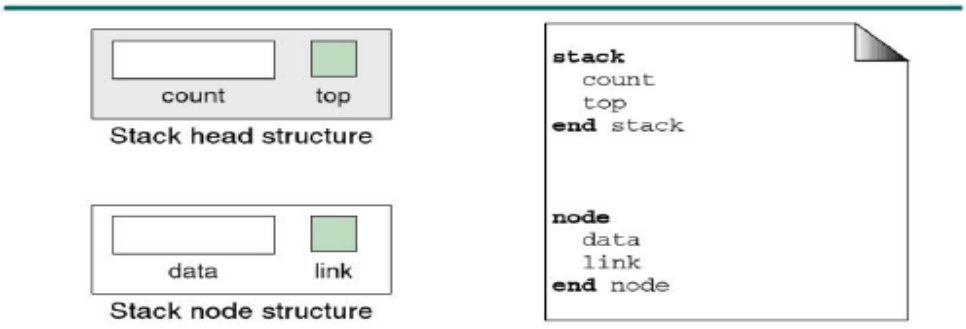


FIGURE 3-7 Stack Data Structure

Stack Data node:

- The rest of the data structure is a typical linked list data node
- The stack data node looks like any linked list node which contains,
  - Data
  - Link pointer to the other data nodes, making it a self – referential data structure.
- In self – referential structure, each instance of the structure contains a pointer to another instance of the same structure.
- The stack data node is as shown in above figure.

Stack algorithms:

- The eight stack operations discussed in this section should be sufficient to solve any basic stack problem.
- If any additional operations are required we could easily add.
- Implementation of stack depends on implementation language, it is usually implemented with a **stack head structure** in C
- The below figure show the four most common stack operations are: create stack, push stack, pop stack, destroy stack.
- Operations such as stack top are not shown in the figure because they do not change the stack structure.

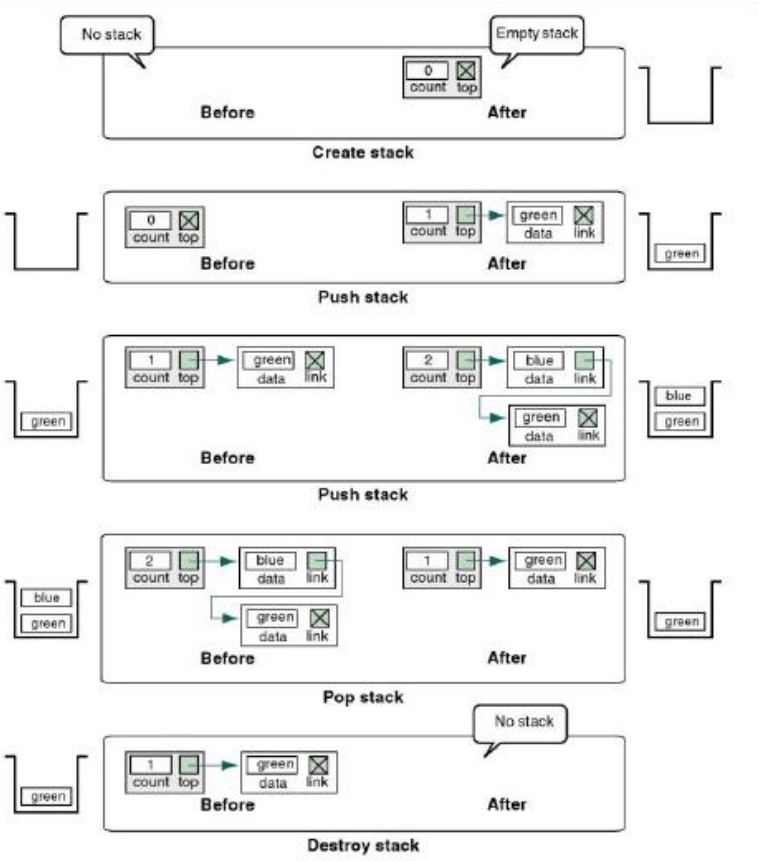


FIGURE 3-8 Stack Operations

a. Create stack:

- Create stack allocates memory for the stack structure and initializes its metadata(a data about data).
- The pseudo code is shown in algorithm

Create stack algorithm:

Algorithm createstack
Creates and initializes metadata structure
Pre Nothing
Post structure created and initialized
Return Stack head

1. Allocate memory for stack head
2. Set count to 0
3. Set top to NULL
4. Return stack head

end create stack

---

b. Push Stack:

- Push stack inserts an element into the stack

- The first thing we need to do when we push data into a stack is find memory for the node.
- We must therefore allocate a node from dynamic memory
- Once the memory is allocated, we simply assign the data to the stack node and then set the link pointer to point to the node currently indicated as the stack top.
- We also need to update the stack top pointer and add 1 to the stack count field
- Figure traces a push a stack operation in which a new pointer (PNew) is used to identify the data to be inserted into the stack.

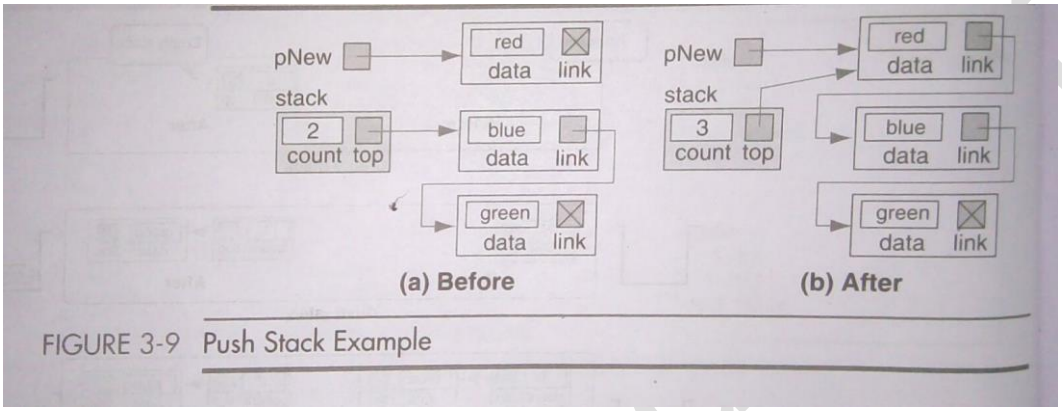


FIGURE 3-9 Push Stack Example

- To develop the insertion algorithm using a linked list, we need to analyze three different stack conditions:
  1. Insertion into an empty stack
  2. Insertion into a stack with data, and
  3. Insertion into a stack when the available memory is exhausted
- The third is an error condition
- When we insert into a stack that contains data,
- The new nodes link pointer is set to point to the node currently at the top , and

The stack's top pointer is set to point to the new node.

Push stack design: Algorithm

Algorithm pushstack (stack,data)
Insert(push) one item into the stack
Pre stack passed by reference
Data contain data to be pushed into stack
Post data have been pushed into stack

1. Allocate new node
2. Store data in new node
3. Make current top node the second node
4. Make new node the top
5. Increment stack count

end pushstack

---

c. Pop stack:

- Pop stack sends the data in the node at the top of stack back to the calling algorithm
- It then adjusts the pointers to logically delete the node, it is physically deleted by recycling the memory, that is returning it to dynamic memory.
- After count is adjusted by subtracting 1, the algorithm returns the status to the caller, if the top was successful, it returns true; if the stack is empty when pop is called, it returns false.
- The operations for pop stack are traced in figure.

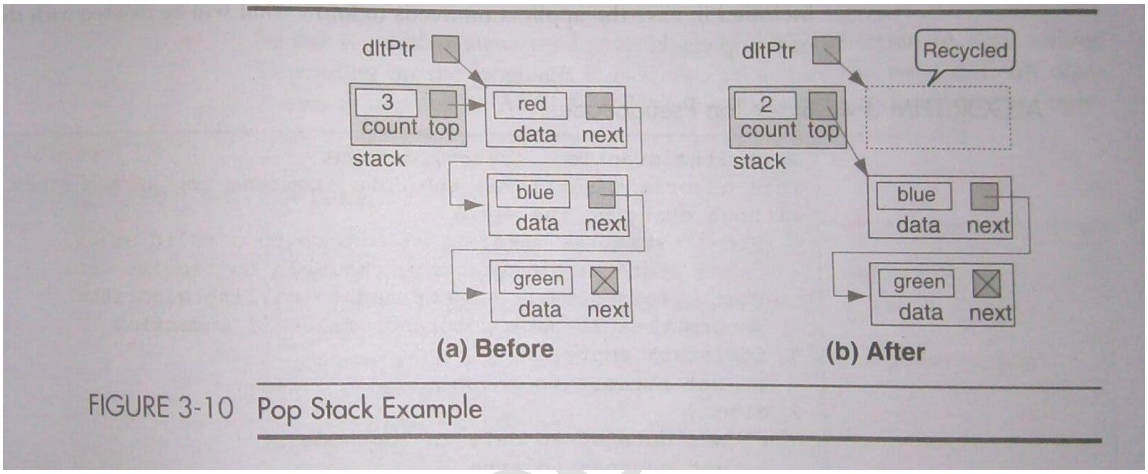


FIGURE 3-10 Pop Stack Example

Algorithm: Pop stack

Algorithm popstack(stack,dataout)
This algorithm pops the item on the top of the stack and returns it to the user
Pre Stack passed by reference
dataout is a reference variable to receive data
Post Data have been returned to calling algorithm
Return true if successful; false if underflow

1. if(stack is empty)
    1. Set success to false
2. else
    1. Set dataout to data in top node
    2. Make second node to the top node
    3. Decrement stack count
    4. Set success to true
3. end if
4. Return success end popstack

d. Stack top

- The stack top algorithms send the data at the top of stack back to the calling module without deleting the top node.
- This is included in case the application needs to know what will be deleted with the next pop stack

Algorithm: stack top pseudo code

Algorithm stackTop (stack,dataout)
This algorithm retrieves the data from the top of the stack without changing the stack
Pre stack is metadata structure to a valid stack
datout reference variable to receive data
Post Data have been returned to calling algorithm
Return true if data returned, false if underflow

1. if(stack empty)
    1. Set success to false
2. else
    1. Set dataout to data in top node
    2. Set success to true
3. endif
4. return success

end stack Top

---

e. Empty stack

- Empty stack is provided to implement the structured programming concept of data hiding: if the entire program has access to the stack head structure, it is not needed.
- If the stack is implemented as a separately compiled program to be linked with other programs, the calling program may not have access to the stack head node.
- In these cases it is necessary to provide a way to determine whether the stack is empty.
- The pseudo code for empty stack is shown in algorithm

Algorithm: Empty stack

Algorithm emptystack (stack)
Determines if stack is empty and returns a boolean
Pre Stack is metadata structure to a valid stack
Post Returns stack status
Return true if stack empty, false if stack contains data

1. if (stack count is 0)
    1. return true
2. else
    1. return false
3. end if

end emptystack

---

f. Full stack

- It is another structured programming implementation of data hiding.
- Depending on the language, it may also be one of the most difficult algorithms and no direct way to implement.
- The pseudo code for full stack is shown in algorithm

Algorithm: Full stack

Algorithm fullstack(stack)
Determined if stack is full and returns a Boolean
Pre stack is metadata structure to a valid stack
Post returns stack status
Return true if stack full, false is memory available

1. if(memory not available)
    1. return true
2. else
    1. return false
3. end if

end fullstack

---

g. Stack Count:

- Stack count returns the number of elements currently in the stack. It is another implementation of the data – hiding principle of structured programming.
- The pseudo code is shown in algorithm.

Algorithm: stack count

Algorithm stackcount (stack)
Returns the number of elements currently in stack
Pre stack is metadata structure to a valid stack
Post returns stack count
Return integer count of the number of elements in the stack

1. return (stack count)

end stackcount

---

h. Destroy stack

- It deletes all data in a stack

Algorithm: Destroy stack

Algorithm destroystack (stack)
This algorithm releases all nodes back to the dynamic memory
Pre stack passed by reference
Post stack empty and all nodes deleted

1. if (stack is not empty)
    1. loop (stack not empty)
        1. delete top node
        2. end loop
2. end if
3. delete stack head

end destroystack

C Implementation

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

```
#include<conio.h>
#include<stdlib.h>
#define NULL 0
struct data
{
        int info;
        struct data *next;
};
struct stack
{
        int count;
        struct data *top;
}*head;
typedef struct stack hnode;
typedef struct data dnode;
dnode *pnew,*dptr,*list;
void createstack();
void push();
void pop();
void print();
void main()
{
        int ch;
        clrscr();
        do
        {
                printf("\n1.create stack");
                printf("\n2.push");
                printf("\n3.pop");
                printf("\n4.print");
                printf("\n5.exit");
                printf("\nEnter choice:\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: createstack();
                                break;
                        case 2: push();
                                break;
                        case 3: pop();
                                break;
                        case 4: print();
                                break;
```

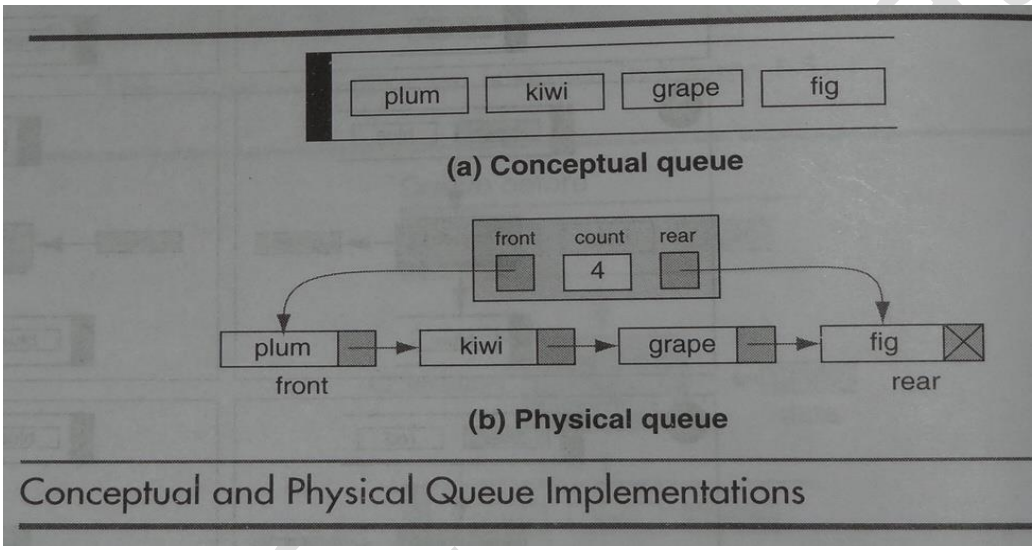
```
                        case 5: exit(0);
                                break;
                }
        }while(ch!=5);
}
void createstack()
{
        head=(hnode *)malloc(sizeof(hnode));
        head->count=0;
        head->top=NULL;
}
void push()
{
        pnew=(dnode *)malloc(sizeof(dnode));
        printf("Enter the element");
        scanf("%d",&pnew->info);
        pnew->next=head->top;
        head->top=pnew;
        head->count++;
        printf("the number of elements in the stack after pushing are:%d\n",head->count);
}
void pop()
{
        if(head->top==NULL)
                printf("\nStack is Empty\n");
        else
        {
                dptr=head->top;
                printf("the popped element is %d\n",dptr->info);
                head->top=dptr->next;
                head->count--;
        printf("the number of elements in stack after popping are:%d\n",head->count);
        }
}
void print()
{
        int i;
        list=head->top;
        if(list==NULL)
                printf("\nStack is empty\n");
        else
        {
        printf("\n the stack elements are");
```

```
            for(i=head->count;i>0;i--)
            {
                printf("\n%d",list->info);
                list=list->next;
            }
        }
}
```

Linked representations of Queues:

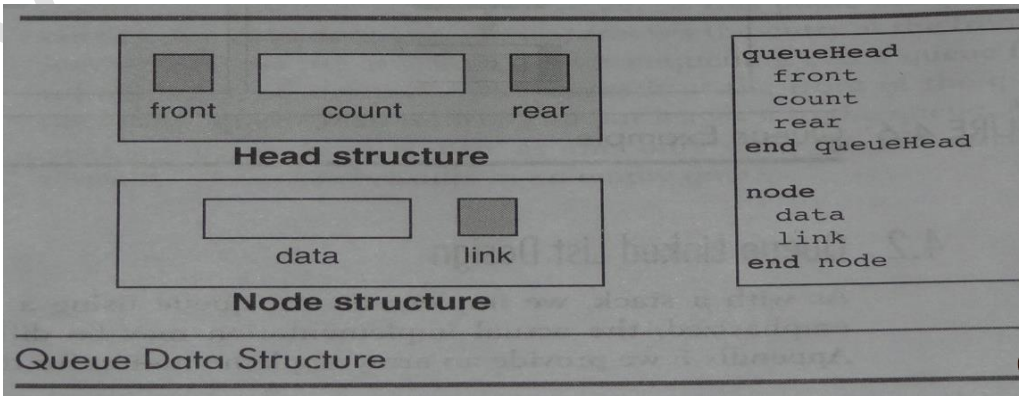
Data Structure:

- Two different data structures are needed to implement the queue:
  - a queue head structure
  - a data node structure
- After it is created, the queue will have one head node and zero or more nodes.
- Fig shows the conceptual and physical implementations for the queue structure



Queue head:

- It requires two pointers and a count
- These fields are stored in the queue head structure.
- Other attributes, such as maximum number etc., can be stored in the head node.
- The queue head structure is as shown in fig.

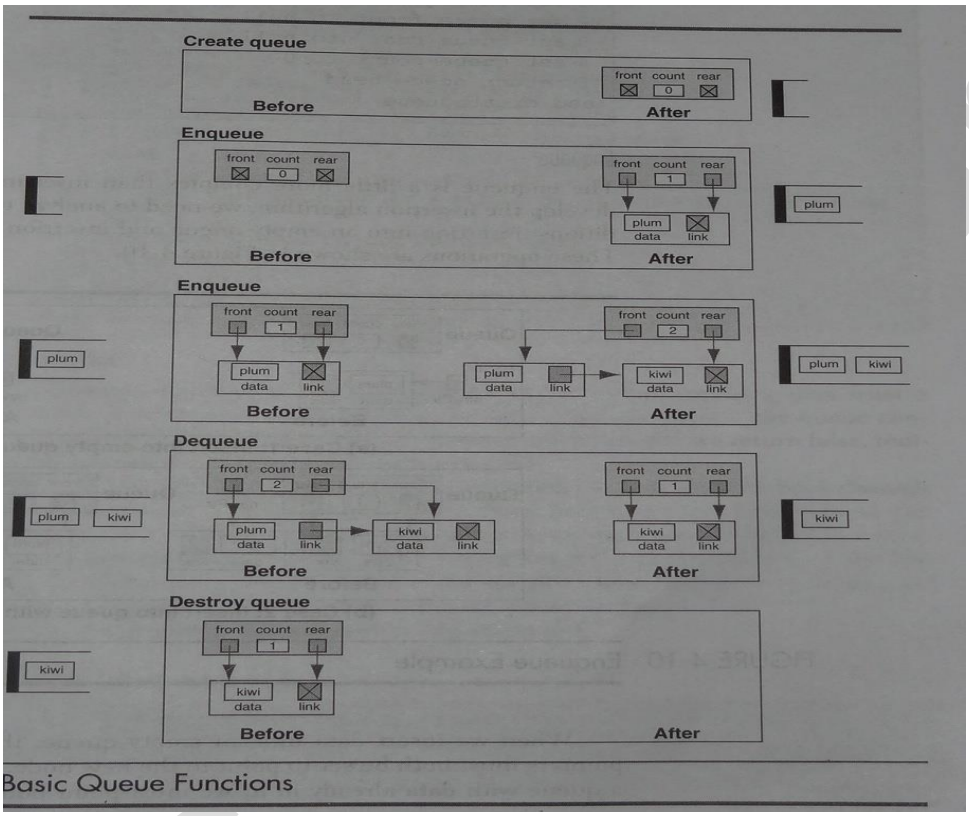


Queue data node:

- The queue data node contains the user data and a link field pointing to the next node, if any.
- These nodes are stored in dynamic memory and are inserted and deleted as requested by using program.
- Its structure is also shown in above fig.

Queue Algorithms:

- The four basic queue operations are shown in fig.



Create queue:

Algorithm createQueue

Creates and initializes queue structure.

	Pre	queue is a metadata structure
	Post	metadata elements have been initialized
	Return	queue head

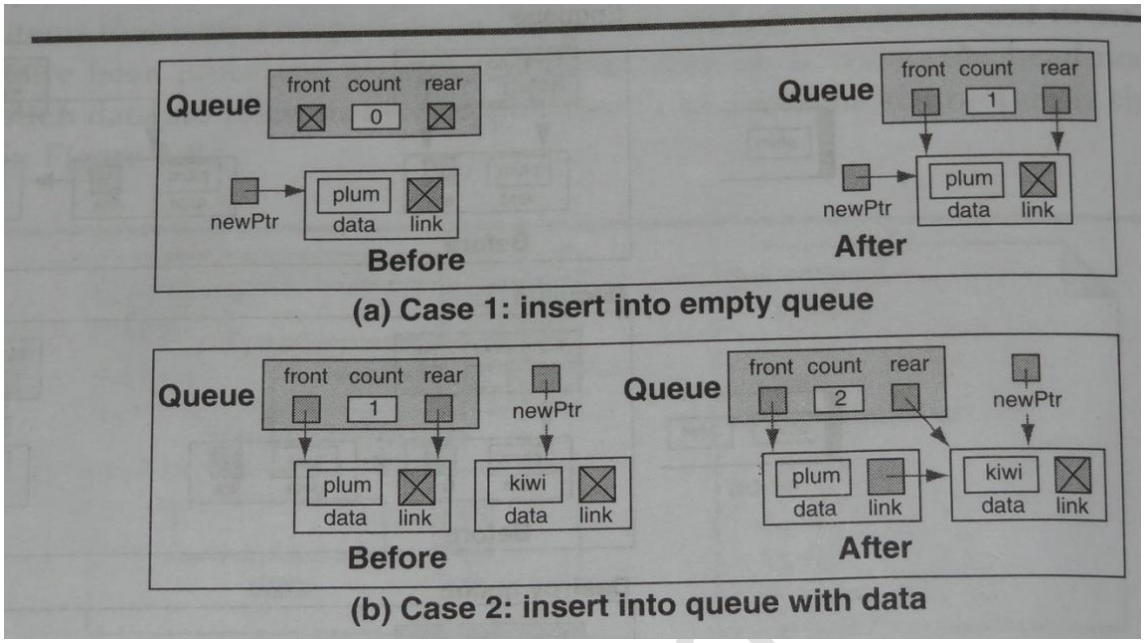
1 allocate queue head
2 set queue front to null
3 set queue rear to null
4 Set count to zero
5 return queue head
end createQueue

Enqueue:

- The enqueue is a little more complex
- To develop the insertion algorithm, we need to analyze two different queue conditions:

- Insertion into an empty queue
- Insertion into a queue with data

- These are operations are shown in fig.



- When we insert data into an empty queue, the queue's front and rear pointers must both be set to point to the new node.
- When we insert data into a queue with data already in it, we must point both the link field in the last node and the rear pointer to the new node.
- If the insert was successful, we return a true; if there is no memory left for the new node, we return a false.
- The pseudo code is shown in algorithm

Insert data into Queue

Algorithm enqueue (queue, dataIn)
This algorithm inserts data into a queue.
    Pre queue is a metadata structure
    Post dataIn has been inserted
    Return true if successful, false if overflow
1 if (queue full)
    1 return false
2 end if
3 allocate (new node)
4 move dataIn to new node data
5 set new node next to null pointer
6 if (empty queue)
    Inserting into null queue
    1 set queue front to address of new data

7 else

Point old rear to new node

1 set next pointer of rear node to address of new node

8 end if

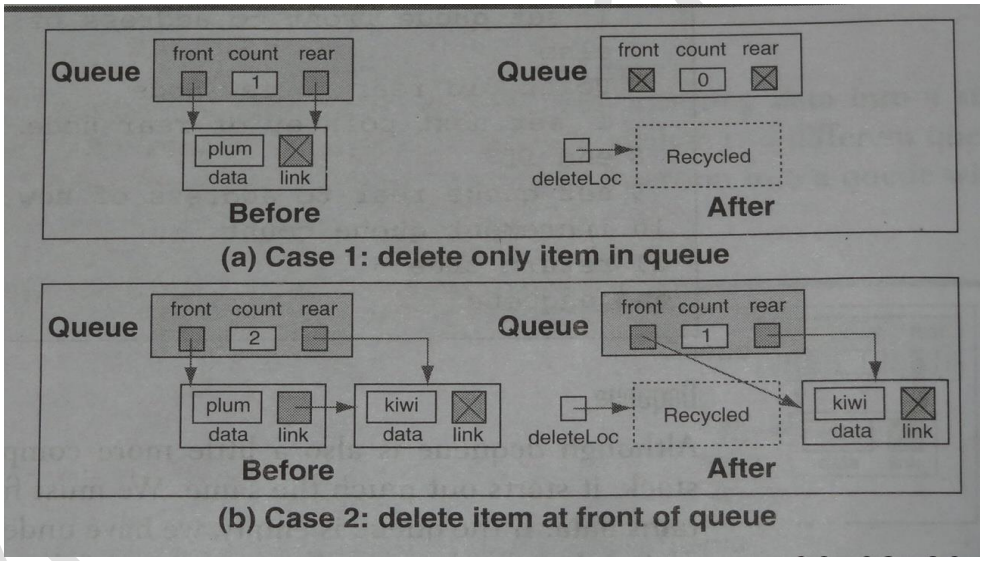
9 set queue rear to address of new node

10 increment queue count

end enqueue

Dequeue

- Dequeue is also a little more complex.
- We must ensure that the queue contains data.
- If the queue is empty, we have underflow and we return false, indicating that the dequeue was not successful.
- Otherwise, we retrieve the item at the front pointer and move the front pointer to the next item.
- If we dequeue the last item the front pointer automatically becomes null, but we must also set to the rear pointer to null.
- These cases are shown in fig.



- The pseudo code is shown in algorithm.

Algorithm dequeue (queue, item)

This algorithm deletes a node from a queue.

Pre queue is a metadata structure

Item is a reference to calling algorithm variable

Post data at queue front returned to user through item and front element deleted

Return true if successful, false if underflow

1 if (queue empty)

1 return false

2 end if

3 move front data to item
4 if (only 1 node in queue)
    Deleting only item in queue
    1 set queue rear to null
5 end if
6 set queue front to queue front next
7 decrement queue count
8 return true
end dequeue

Retrieving queue data:

- The only difference between the two retrieve queue operations are: Queue front, Queue rear is which pointer used either front or rear.
- Here we are looking at queue front
- The logic is same as dequeue but it doesn't delete any element.
- It first checks for an empty queue and returns false
- Otherwise it passes back the data and returns true.
- The pseudo code is shown in algoritm.

Retrieve data at front of queue

Algorithm queueFront (queue, dataOut)
Retrieves data at the front of the queue without changing queue contents.
Pre queue is a metadata structure
    dataout is a reference to calling algorithm variable
Post data passed back to caller
Return true if successful, false if underflow
1 if (queue empty)
    1 return false
2 end if
3 move data at front of queue to dataOut
4 return true
end queueFront

- To implement queue rear, the algorithm is as shown
- **Retrieving data at Rear of queue**

Algorithm queueRear (queue, dataOut)
Retrieves data at the Rear of the queue without changing queue contents.
Pre queue is a metadata structure
    dataout is a reference to calling algorithm variable
Post data passed back to caller
Return true if successful, false if underflow
1 if (queue empty)
    1 return false
2 end if

3 move data at Rear of queue to dataOut
4 return true
end queueRear

Empty Queue

- Empty queue returns true if the queue is empty and false if the queue contains data.
- There are several ways to test for an empty queue.
- Checking the queue count is the easiest
- The pseudo code is shown in algorithm

Queue Empty

Algorithm emptyQueue (queue)
This algorithm checks to see if a queue is empty.
Pre queue is a metadata structure
Return true if empty, false if queue has data
1 if (queue count equal 0)
    1 return true
2 else
    1 return false
end emptyQueue

Full Queue

- Full queue is another structured programming implementation of data hiding.
- Depending on language, it may be one of the most difficult algorithms to implement provides no direct way to implement it.
- Te pseudo code is show in algorithm

Full Queue

Algorithm fullQueue (queue)
This algorithm checks to see if a queue is full. The queue is full if memory cannot be allocated for another node.
Pre queue is a metadata structure
Return true if full, false if room for another node
1 if (memory not available)
    1 return true
2 else
    1 return false
3 end if
end fullQueue

Queue count

- Queue count returns the number of elements currently in the queue by simply returning the count found in the queue head node.
- The Pseudo code is shown in algorithm

Queue count

Algorithm queueCount (queue)

This algorithm returns the number of elements in the queue.

Pre queue is a metadata structure

Return queue count

1 return queue count

end queueCount

Destroy Queue

Destroy queue deletes all data in the queue. The pseudo code is shown in algorithm

Algorithm destroyQueue (queue)

This algorithm deletes all data from a queue.

Pre queue is a metadata structure

Post all data have been deleted

1 if (queue not empty)

    1 loop (queue not empty)

        1 delete front node

    2 end loop

2 end if

3 delete head structure

end destroy Queue

C Implementation

queueadt.h

```
struct node
{
 int data;
 struct node *link;
}*newptr,*dptr,*list;
struct queue
{
  struct node *front;
  struct node *rear;
  int count;
}*head;
typedef struct node dnode;
typedef struct queue hnode;
void createqueue()
{
 head=(hnode*)malloc(sizeof(hnode));
 head->front=NULL;
 head->rear=NULL;
 head->count=0;
```

```
}
void enqueue()
{
  newptr=(dnode *)malloc(sizeof(dnode));
  printf("enter the data to be inserted:\n");
  scanf("%d",&newptr->data);
  if(queueempty())
   head->front=newptr;
  else
   head->rear->link=newptr;
  head->rear=newptr;
  newptr->link=NULL;
  head->count++;
}
void dequeue()
{
 if(queueempty())
 {
    head->rear=NULL;
    printf("Queue is underflow\n");
   // exit(0);
 }
 else
 {
 dptr=head->front;
 printf("the element deleted is:%d",dptr->data);
 head->front=dptr->link;
 head->count--;
 }
}
void display()
{
  int c;
  if(queueempty())
 {
    printf("Queue is underflow\n");
    //exit(0);
 }
 else
 {
  list=head->front;
  c=head->count;
  printf("the queue elements are:\n");
```

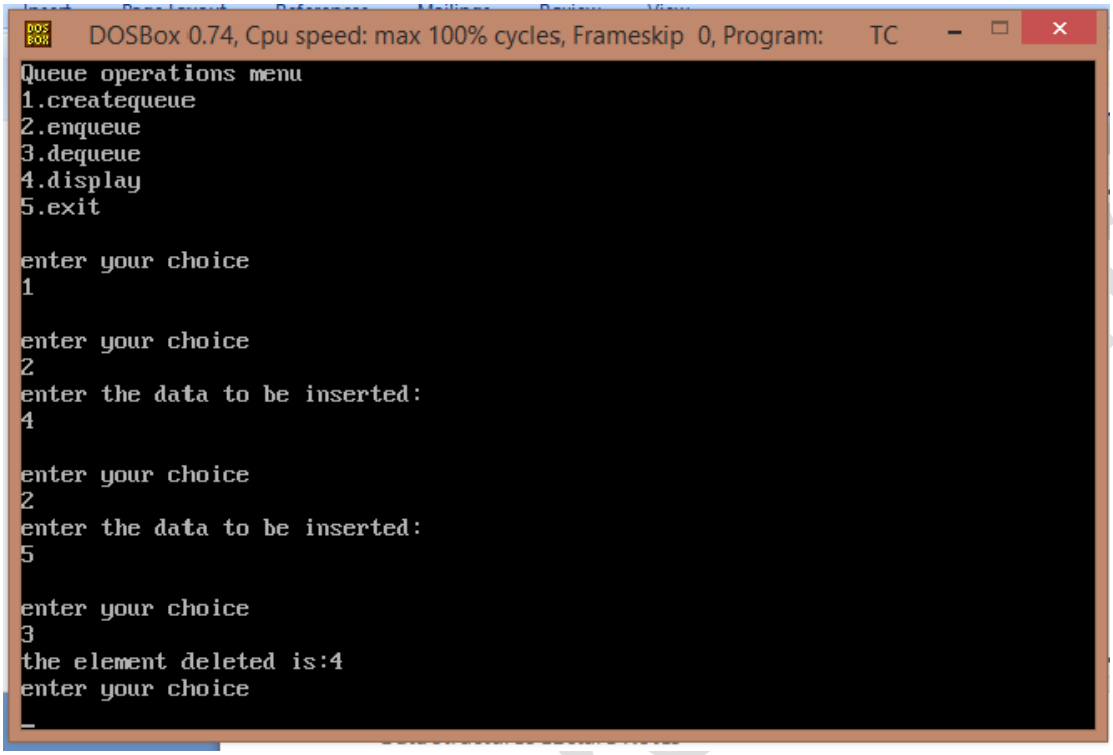
```
  while(c>0)
  {
   printf("%d ",list->data);
   list=list->link;
   c--;
  }
  }
}
int queueempty()
{
if(head->count==0)
return 1;
else
return 0;
}
```

program

```
#include<stdio.h>
#include<conio.h>
#include"queueadt.h"
void main()
{
  int ch;
  clrscr();
  printf("Queue operations menu\n");
  printf("1.createqueue\n2.enqueue\n3.dequeue\n4.display\n5.exit\n");
  do
  {
        printf("\nenter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:createqueue();
             break;
        case 2:enqueue();
                break;
        case 3:dequeue();
                break;
        case 4:display();
                break;
        case 5:exit(0);
        default: printf("not a valid choice\n");
    }
```

```
    }while(ch!=5);
  getch();
}
```

Output



4. 3. Polynomials using singly linked lists:

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

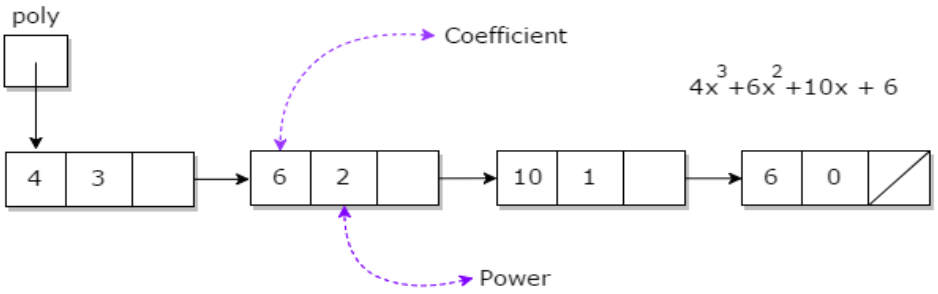
An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



C Implementation:

```
struct node
{
   int cof;
   int exp;
   struct node *link;
};
struct node * create(struct node *q)
 {
  int i,n;
  printf("enter the number of nodes");
  scanf("%d",&n);
  struct node *ptr=(struct node *)malloc (sizeof(struct node));
  for(i=0;i<n;i++)
  {
              printf("entre the coefficient and exponent respectivly");
              scanf("%d%d",&ptr->cof,&ptr->exp);
              ptr->link=NULL;
              q=insert(ptr,q);
  }
  return q;
 }
struct node * insert(struct node *ptr,struct node *p)
 {
  struct node *temp,*b;
  if(p==NULL)
  p=ptr;
  else
  {
     if((p->exp)<(ptr->exp))
     {
                    ptr->link=p;
                    p=ptr;
     }
     else
     {
```

```
        temp=p;
        while((temp!=NULL)||((temp->link->exp)<(ptr->exp)))
        temp=temp->link;
        b=temp->link;
        temp->link=ptr;
        ptr->link=b;
      }
  }
  return p;
  }
void display(struct node *ptr)
 {
   struct node *temp;
  temp=ptr;
  while(temp!=NULL)
  {
                printf("%d x ^ %d + ",temp->cof,temp->exp);
                temp=temp->link;
  }
 }
 int main()
  {
  printf("enter the first polynomial");
  struct node *p1=NULL,*p2=NULL;
  p1=(struct node *)malloc(sizeof(struct node));
  p2=(struct node *)malloc(sizeof(struct node));
  p1=create(p1);
  printf("enter second polynomial");
  create(p2);
  display(p1);
  display(p2);
  getch();
  return 0;
 }
```

4.4. Polynomials using circularly linked lists:

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
int coeff,exp;
struct node *link;
}dnode;
```

```
typedef struct
{
int count;
//node *pos;
dnode *head;
dnode *rear;
}list;
dnode *pnew,*ppre,*clist,*ploc,*last;
list *plist;
void createlist()
{
plist=(list*)malloc(sizeof(list));
plist->head=NULL;
plist->rear=NULL;
plist->count=0;
}
void insert()
{
pnew=(dnode*)malloc(sizeof(dnode));
printf("enter the coeff and exp data to be inserted\n");
scanf("%d%d",&pnew->coeff,&pnew->exp);
pnew->link=NULL;
if(ppre==NULL)
{
pnew->link=plist->head;
plist->head=pnew;
if(plist->count==0)
plist->rear=pnew;
last=plist->rear;
last->link=pnew;
}
else
{
pnew->link=ppre->link;
ppre->link=pnew;
if(pnew->link==plist->head)
{
plist->rear=pnew;
}
}
plist->count++;
}
int search()
```

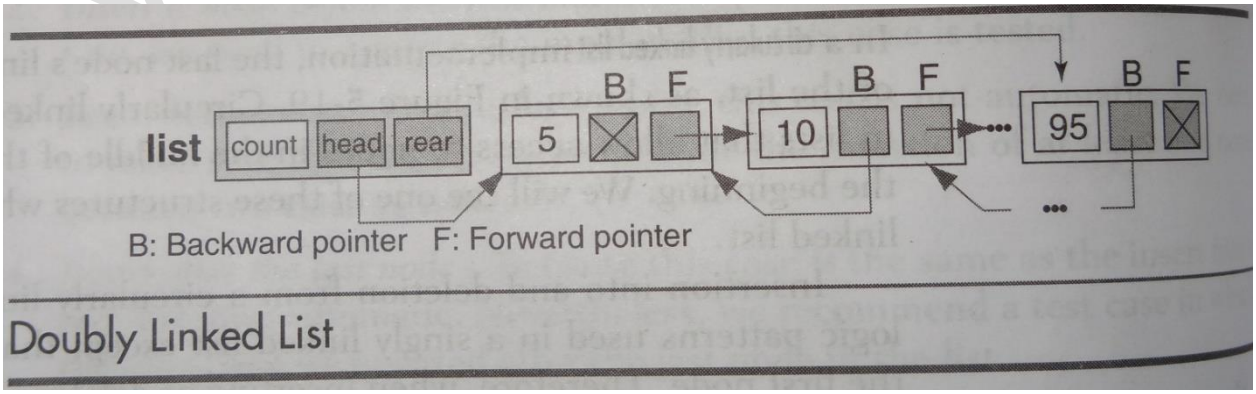
```
{
int targ;
ppre=NULL;
ploc=plist->head;
if(plist->count==0)
printf("list is empty\n");
else
{
printf("enter the data\n");
scanf("%d",&targ);
while(ploc->link!=plist->head&&targ>ploc->coeff)
{
ppre=ploc;
ploc=ploc->link;
}
if(ploc==NULL)
{
printf("Data not found in the list");
return 0;
}
else
{
if(targ==ploc->coeff)
{
printf("element found\n");
return 1;
}
else
{
printf("element not found\n");
return 0;
}
}
}
}
void del()
{
if(ppre==NULL)
{
plist->head=ploc->link;
last=plist->rear;
last->link=plist->head;
}
```

```
else
ppre->link=ploc->link;
if(ploc->link==plist->head)
{
plist->rear=ppre;
last=plist->rear;
last=plist->head;
}
plist->count--;
free(ploc);
}
void traversal()
{
int i;
if(plist->count==0)
printf("list is empty\n");
else
{
clist=plist->head;
for(i=plist->count;i>0;i--)
{
printf("coeff=%d\t exp=%d\n",clist->coeff,clist->exp);
clist=clist->link;
}
}
}
void main()
{
int ch;
clrscr();
printf("List menu\n");
printf("1.Create list\n2.Insertion\n3.Deletion\n4.Traversal\n5.Exit\n");
do
{
printf("enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:createlist();
        break;
case 2:printf("1. Insertion at beginning\n 2.Insertion at middle or end\n");
    printf("enter your choice\n");
    scanf("%d",&ch);
```

```
        switch(ch)
        {
        case 1:traversal();
                ppre=NULL;
                insert();
                traversal();
                break;
        case 2:  traversal();
                    if(search())
                {
                ppre=ploc;
                insert();
                traversal();
                }
                break;
                }
        break;
case 3: traversal();
            if(search())
            del();
            traversal();
            break;
case 4:traversal();
            break;
case 5:exit(0);
}
}while(ch!=5);
getch();
}
```

4.5. Doubly Linked lists and its operations:

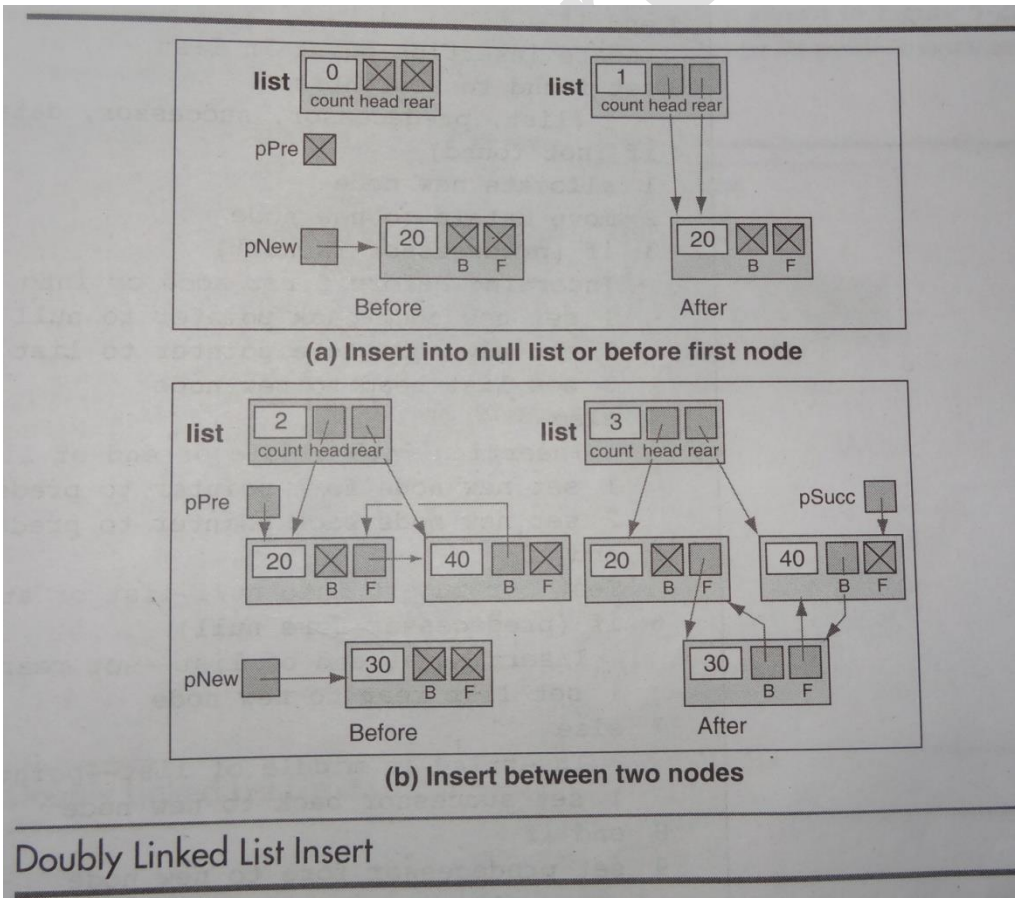
- A doubly linked list is a linked list structure in which each node has a pointer to both its successor and predecessor
- Fig shows representation



- There are three pieces of meta data in the head structure:
    - a count head
    - a head pointer
    - a rear pointer
- A rear pointer is not required in all doubly linked list, it makes some of the list algorithms such as insert and search more efficient
- Each node contains two pointers:
    - a backward pointer to its predecessor
    - a forward pointer to its successor
- Another variation of doubly linked list is the doubly circular linked list
- In this the last forward pointer points to the first node of the list and the backward pointer of the first node points to the last node
- If there is only one node in the list both forward and backward pointer points to the node itself

Insertion

- Inserting follows same as inserting node into the singly linked list but we need connect both forward and backward pointers
- A null doubly linked list head and rear pointers are null
- To insert a node into a null list, we simply set the head rear pointers to point to the new node and set the forward and backward pointers of the new node to null
- The results of inserting node into a null list are shown in fig.



- Fig(b) shows the case for inserting between two nodes
- The new node need to be set to point to both its predecessor and its successor, and they need to be set to point to the new node
- Because the insert is in the middle of the list, the **head structure is unchanged**
- Inserting at the end of the list requires that the new node's backward pointer to be set to point to its predecessor
- Because there is no successor, the forward pointer is set to null
- The rear pointer in the head structure must also be set to point to the new rear node

Algorithm insertdbl(list,dataIn)

This algorithm inserts data into a doubly linked list

Pre	list is metadata structure to a valid list
	dataIn contains the data to be inserted
Post	the data have been inserted in sequence
Return	0: failed – dynamic memory overflow
	1: successful
	2: failed – duplicate key presented

1. If(full list)
   1. return 0
2. end if

   locate insertion point in list
3. set found to searchlist(list, predecessor, successor, dataIn, key)
4. if(not found)
   1. allocate new node
   2. move dataIn to new node
   3. if(predecessor is null)
      1. set new node back pointer to null
      2. set new node fore pointer to list head
      3. set list head to new node
   4. else

   inserting into middle or end of list
      1. set new node fore pointer to predecessor fore
      2. set new node back pointer to predecessor
   5. end if

   Test for insert into null list or at end of list
   6. if(predecessor fore null)

   inserting at end of list – set rear pointer
      1. set list rear to new node
   7. else

   inserting in middle of list – point successor to new
      1. set successor back to new node
   8. end if

9. set predecessor fore to new node
10. return 1

5. end if

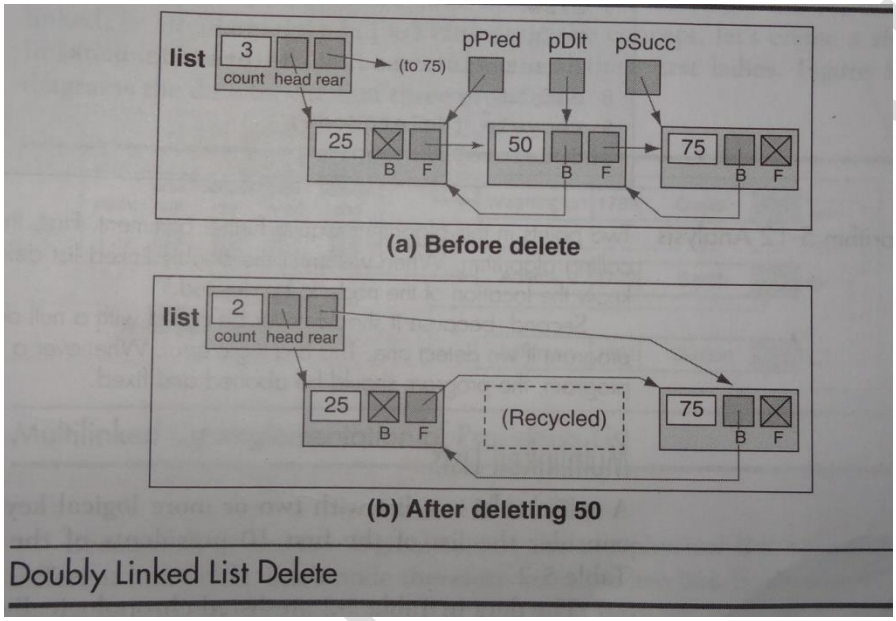
duplicate data key already exists

6. return 2

end insertdbl

Deletion:

- Deleting requires that the deleted nodes predecessor, if present, be pointed to the deleted node's successor and that the successor, if present, be set to point to the predecessor
- As show in fig



- Once we locate the node to be deleted, we change its predecessors and successors pointers and recycle the node

Algorithm deleteDbl(list, deleteNode)

This algorithm deletes a node from doubly linked list

Pre list is metadata structure to a valid list
deleteNode is a pointer to the node to be deleted

Post node deleted

1. If(deletedNode null)
    1. abort("impossible condition in delete double");
2. end if
3. if(deleteNode back not null)

point predecessor to successor

1. set predecessor to deleteNode back
2. set predecessor fore to deleteNode fore

4. else updatehead pointer
    1. set list head to deleteNode fore
5. end if
6. if(deleteNode fore not null)

point success to predecessor

1. set successor to deleteNode fore
2. set successor back to deleteNode back

7. else

point rear to predecessor

1. set list rear to deleteNode back

8. endif
9. recycle(deleteNode)

end deleteDbl

Doubly linked list program in C:

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
int data;
struct node *b;
struct node *f;
}dnode;
typedef struct
{
int count;
//node *pos;
dnode *head;
dnode *rear;
}list;
dnode *pnew,*ppre,*succ,*clist,*ploc;
list *plist;
void createlist()
{
plist=(list*)malloc(sizeof(list));
plist->head=NULL;
plist->rear=NULL;
plist->count=0;
}
void insert()
{
pnew=(dnode*)malloc(sizeof(dnode));
printf("enter the data to be inserted\n");
scanf("%d",&pnew->data);
pnew->f=NULL;
if(ppre==NULL)
{
pnew->b=NULL;
```

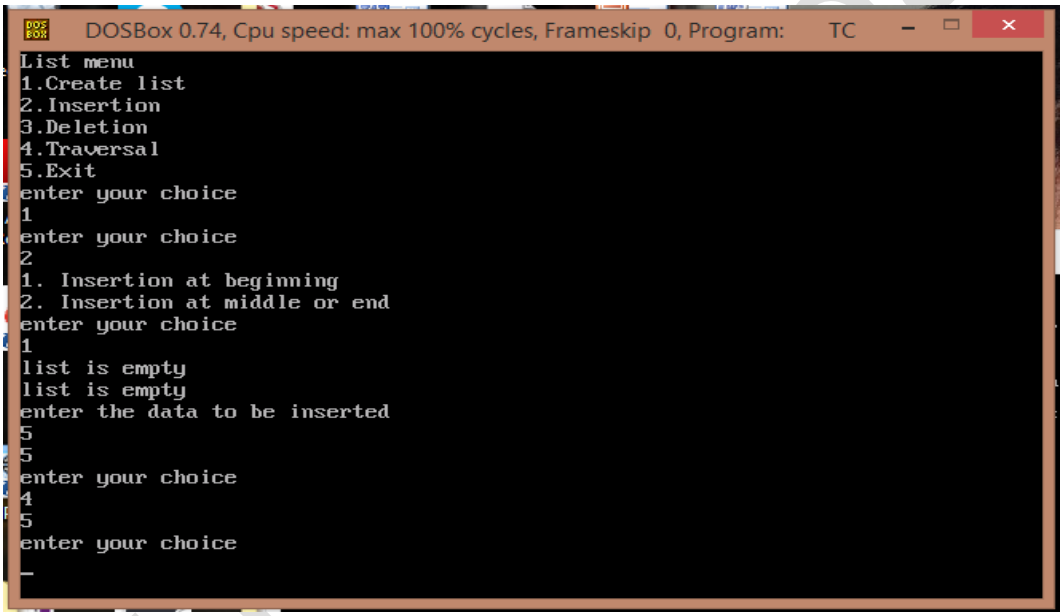
```
pnew->f=plist->head;
plist->head=pnew;
if(plist->count==0)
plist->rear=pnew;
}
else
{
pnew->f=ppre->f;
pnew->b=ppre;
}
if(ppre->f==NULL)
plist->rear=pnew;
else
succ->b=pnew;
ppre->f=pnew;
plist->count++;
}
int search()
{
int targ;
ppre=NULL;
ploc=plist->head;
if(plist->count==0)
printf("list is empty\n");
else
{
printf("enter the data\n");
scanf("%d",&targ);
while(ploc->f!=NULL&&targ>ploc->data)
{
ppre=ploc;
ploc=ploc->f;
}
if(ploc==NULL)
{
printf("Data not found in the list");
return 0;
}
else
{
if(targ==ploc->data)
{
printf("element found\n");
```

```
return 1;
}
else
{
printf("element not found\n");
return 0;
}
}
}
}
void del()
{
if(ploc->b!=NULL)
{
ppre=ploc->b;
ppre->f=ploc->f;
}
else
plist->head=ploc->f;
if(ploc->f!=NULL)
{
succ=ploc->f;
succ->b=ploc->b;
}
else
plist->rear=ploc->b;
plist->count--;
free(ploc);
}
void traversal()
{
int i;
if(plist->count==0)
printf("list is empty\n");
else
{
clist=plist->head;
for(i=plist->count;i>0;i--)
{
printf("%d\t",clist->data);
clist=clist->f;
}
printf("\n");
```

```
}
}
void main()
{
int ch;
clrscr();
printf("List menu\n");
printf("1.Create list\n2.Insertion\n3.Deletion\n4.Traversal\n5.Exit\n");
do
{
printf("enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:createlist();
        break;
case 2: printf("1. Insertion at beginning\n2. Insertion at middle or end\n");
          printf("enter your choice\n");
          scanf("%d",&ch);
          switch(ch)
          {
          case 1:traversal();
          if(search())
          {
          ppre=NULL;
          succ=ploc;
          insert();
          }
          traversal();
          break;
          case 2:traversal();
          if(search())
          {
          ppre=ploc;
          succ=ploc->f;
          insert();
          }
          traversal();
          break;
          }
          break;
case 3: traversal();
          if(search())
```

```
        {
        succ=ploc->f;
        del();
        }
        traversal();
        break;
case 4:traversal();
        break;
case 5:exit(0);
}
}while(ch!=5);
getch();
}
```

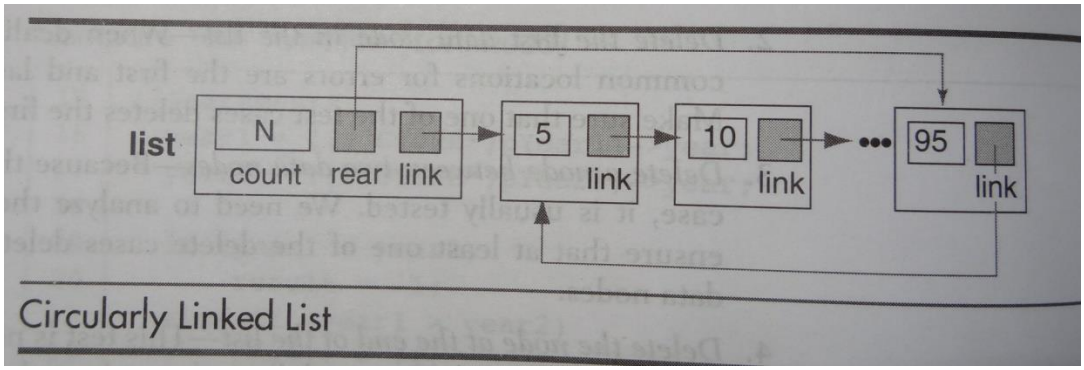
Output:



4.6. Circularly linked list and its operations:

- In a circularly linked list implementation the last node's link points to first node of the list as shown in fig.

- It is primarily used in lists that allow access to nodes in the middle of the list without starting at the beginning
- Insertion into and deletion from a circular linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node
- Therefore, when inserting or deleting the last node, in addition updating the rear in header we must also point the link field to the first node



- Given that we can directly access a node in the middle of the list through its data structure, we are then faced with a problem when searching the list
- If the search target lies before the current node, how do we find it?
- In a singly linked list implementation, however, we automatically continue the search from the beginning of the list
- The ability to continue also presents another problem
- If the target does not exist, in singly linked list we stop **when it reaches the end of the list or we target was less than the current node's data**
- With a circular list, we save starting node's address and stop when we have circled around to it, as shown in the code below:

Loop(target not equal to pLoc key AND pLoc link not equal to startAddress)

Circular linked list program in C:

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
int data;
struct node *link;
}dnode;
typedef struct
{
int count;
//node *pos;
dnode *head;
dnode *rear;
}list;
dnode *pnew,*ppre,*clist,*ploc,*last;
list *plist;
void createlist()
{
plist=(list*)malloc(sizeof(list));
plist->head=NULL;
plist->rear=NULL;
plist->count=0;
```

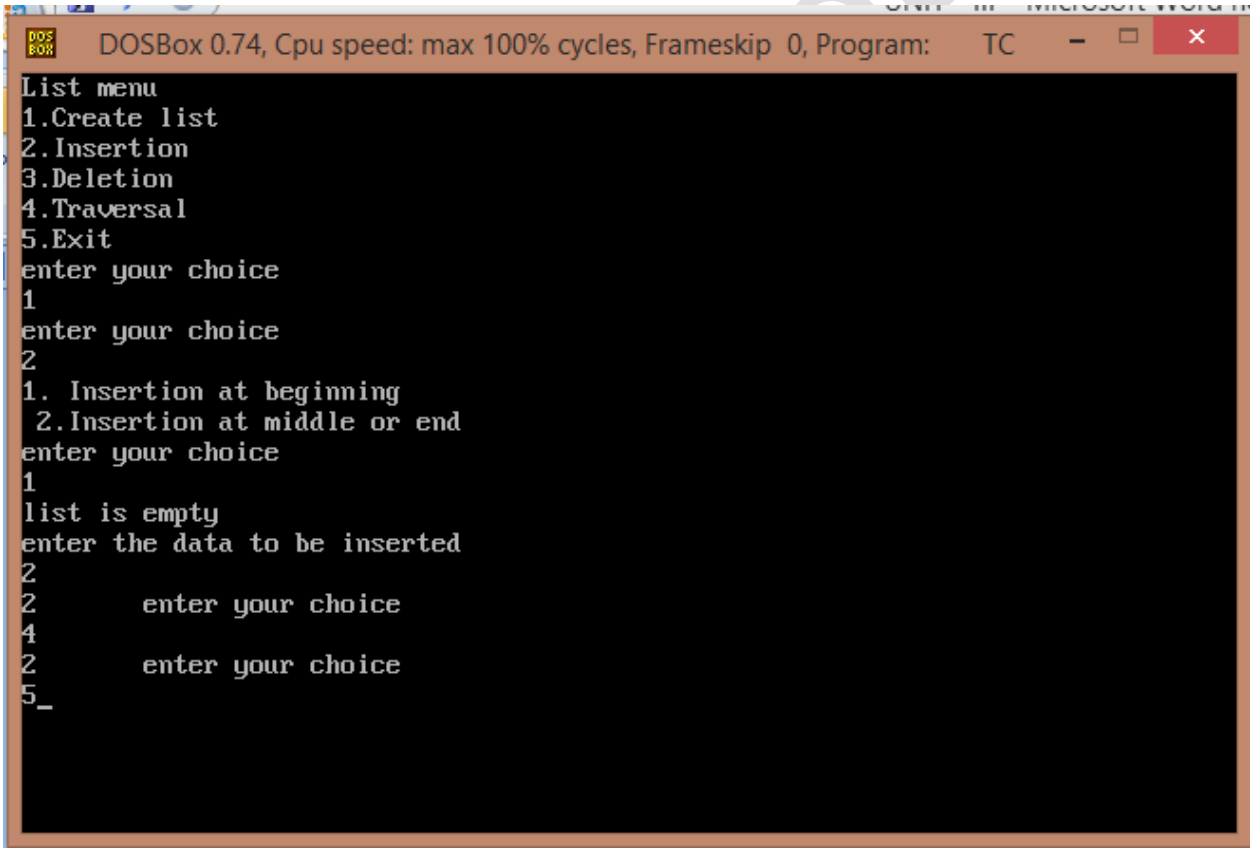
```
}
void insert()
{
pnew=(dnode*)malloc(sizeof(dnode));
printf("enter the data to be inserted\n");
scanf("%d",&pnew->data);
pnew->link=NULL;
if(ppre==NULL)
{
pnew->link=plist->head;
plist->head=pnew;
if(plist->count==0)
plist->rear=pnew;
last=plist->rear;
last->link=pnew;
}
else
{
pnew->link=ppre->link;
ppre->link=pnew;
if(pnew->link==plist->head)
{
plist->rear=pnew;
}
}
plist->count++;
}
int search()
{
int targ;
ppre=NULL;
ploc=plist->head;
if(plist->count==0)
printf("list is empty\n");
else
{
printf("enter the data\n");
scanf("%d",&targ);
while(ploc->link!=plist->head&&targ>ploc->data)
{
ppre=ploc;
ploc=ploc->link;
}
```

```
if(ploc==NULL)
{
printf("Data not found in the list");
return 0;
}
else
{
if(targ==ploc->data)
{
printf("element found\n");
return 1;
}
else
{
printf("element not found\n");
return 0;
}
}
}
}
void del()
{
if(ppre==NULL)
{
plist->head=ploc->link;
last=plist->rear;
last->link=plist->head;
}
else
ppre->link=ploc->link;
if(ploc->link==plist->head)
{
plist->rear=ppre;
last=plist->rear;
last=plist->head;
}
plist->count--;
free(ploc);
}
void traversal()
{
int i;
if(plist->count==0)
```

```
printf("list is empty\n");
else
{
clist=plist->head;
for(i=plist->count;i>0;i--)
{
printf("%d\t",clist->data);
clist=clist->link;
}
}
}
void main()
{
int ch;
clrscr();
printf("List menu\n");
printf("1.Create list\n2.Insertion\n3.Deletion\n4.Traversal\n5.Exit\n");
do
{
printf("enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:createlist();
                                    break;
case 2:printf("1. Insertion at beginning\n 2.Insertion at middle or end\n");
      printf("enter your choice\n");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:traversal();
             ppre=NULL;
             insert();
             traversal();
             break;
      case 2:  traversal();
                 if(search())
             {
             ppre=ploc;
             insert();
             traversal();
             }
             break;
```

```
                }
        break;
case 3: traversal();
        if(search())
        del();
        traversal();
        break;
case 4:traversal();
        break;
case 5:exit(0);
}
}while(ch!=5);
getch();
}
```

Output:



i) Question and Answers: 2 Marks

1. Write advantages of doubly linked lists over singly linked lists. (Jun/ Jul – 2019)

Ans: 1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.

2. List the advantages and limitations of linked list over arrays in representing a group of items. (Jun/Jul – 2019, Dec-2018)

Ans:

	Array	Linked List
Strength	• Random Access (Fast Search Time) • Less memory needed per element • Better cache locality	• Fast Insertion/Deletion Time • Dynamic Size • Efficient memory allocation/utilization
Weakness	• Slow Insertion/Deletion Time • Fixed Size • Inefficient memory allocation/utilization	• Slow Search Time • More memory needed per node as additional storage required for pointers

3. Write the procedure for deleting an element from the list. (Nov/Dec – 2018)

Ans:

1. Move ploc data to dataout
2. if(ppre null)

deleting first node

   a. set list head to ploc link

3. else

deleting other nodes

4. set ppre link to ploc link
5. end if
6. recycle(ploc)

4. What are the applications of linked list? (Nov/Dec – 2018)

Ans: 1. Maintaining directory of names
2. Performing arithmetic operations on long integers
3. Manipulation of polynomials by storing constants in the node of linked List
4. Representing sparse matrices

5. Draw the node structure of a double linked list. Explain the various fields present in it. (Jun – 2017)

Ans: The node consists of three fields,

1. The back pointer which consists of address of its previous node.
2. The data field consists of information about data.
3. The fore pointer consists of address of its next node.

6. What is the use of a header node? (May/Jun – 2017)

Ans: The data in the head are known as metadata; that is, they are data about the data in the list.

7. Define singly linked list. (Nov/Dec – 2016)

Ans: It is a list which consists of a single link storing address of next node, in which operations, such as retrievals, iterations, changes and deletions can be done any where in the list i.e.,

- At beginning
- At middle
- At end of the list

8. List basic operations of a singly linked list.

Ans: Four basic list operations are

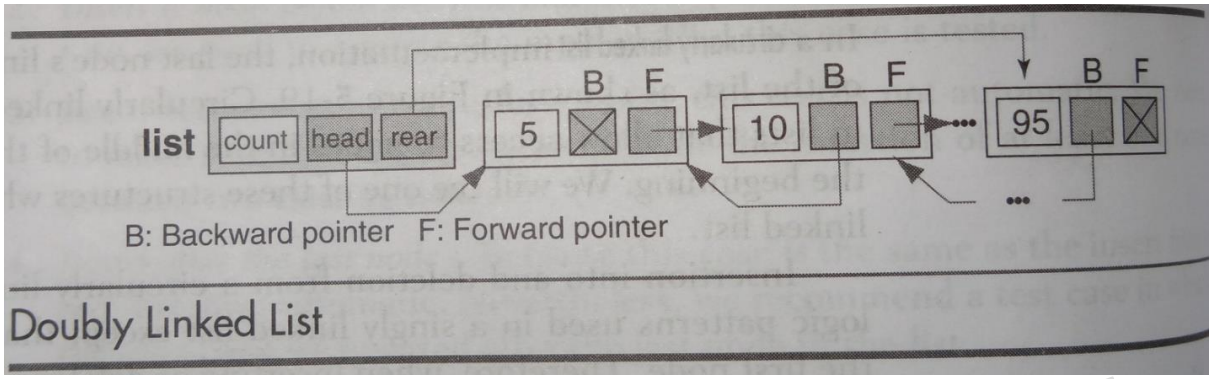
- Insertion
- Deletion
- Retrieval
- Traversal

9. What is a list search?

Ans: A list search is used by several algorithms to locate data in a list.

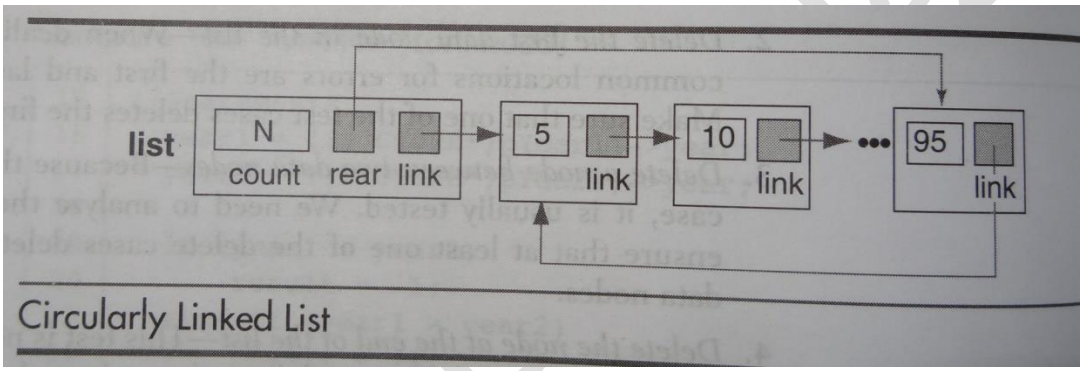
10. Define doubly linked lists.

Ans: A doubly linked list is a linked list structure in which each node has a pointer to both its successor and predecessor.



11. Define circularly linked list.

Ans: In a circularly linked list implementation the last node's link points to first node of the list as shown in fig.



12. Write Procedure for destroy list.

Ans:

5. loop(not at end of list)
   3. set list head to successor node
   4. release memory to heap
6. end loop

no data left in list. reset metadata

7. set list pos to null
8. set list count to 0

13. Discuss in brief about operation list count.

Ans: List count is also one line module, the calling module has no direct access to the list structure for that it is necessary

Algorithm listcount(list)

Returns integer representing number of nodes in list

Pre list is metadata structure to a valid list
Return count for number of nodes in list

2. return (listcount)

end listcount

---

14. What is the condition for empty list?

Ans: Processing logic depends on there being data in a list. We determine empty list by a simple module that returns a Boolean indicating that there are data in the list or that it is empty.

list count equal 0

15. What is meant by insertion in a list?

Ans: Insertion is adding item into a list. It can be done either into empty list, at beginning, at middle and at end.

ii) MCQs

1. _____ is used to add an element into the list.
a. **Insertion** b. Deletion c. Traversal d. Retrieval
2. _____ is used to remove an element from the list.
a. Insertion b. **Deletion** c. Traversal d. Retrieval
3. _____ is used to get the information related to an element without changing the structure of the list
a. Insertion b. Deletion c. Traversal d. **Retrieval**
4._____ is used to traverse the list while applying a process to each element.
a. Insertion b. Deletion c. **Traversal** d. Retrieval
5. A _____ is a good structure for a list because data are easily inserted and deleted at the beginning, in the middle, or at the end of the list
a. Arrays b. Stacks c. Queues d. **Linked list**
6. _____ node contains data about a list, the data are known as metadata; that is, they are data about the data in the list.
a. **Head Node** b. Data Node c. Both A & B d. None
7. The _____ node algorithm logically removes a node from the list by changing various link pointers and then physically deleting the node from dynamic memory.
a. Insert b. **Delete** c. Traversal d. Retrieval
8. A list _____ is used by several algorithms to locate data in a list.
a. Traversal b. Retrieval c. **Search** d. None
9. A _____ is a linked list structure in which each node has a pointer to both its successor and predecessor.
a. Singly b. Circularly c. Loop d. **Doubly**
10. In _____ linked list implementation the last node's link points to first node in list.
a. Singly b. **Circularly** c. Loop d. Doubly
11. When _____ is NULL the insertion is done at beginning.
a. Successor b. Descendent c. **Predecessor** d. None
12. _____ uses search node to locate the data in the list.
a. Insert b. Delete c. **Retrieve** d. None
13. _____ adds data to a list.
a. **Insert** b. Delete c. Retrieve d. None
14. _____ a list start at the first node and examine each node in succession until the last node has been processed.
a. Insert b. Delete c. Retrieve d. **Traversal**
15. _____ operation returns the number of elements present in the list.
a. Traversal b. Retrieval c. **List count** d. None

iii) Question and Answers: 10 Marks

1. List the operations that can be performed on single linked list. In how many ways a node can be deleted from single linked list? Explain. (Nov/Dec – 2019)

Ans:

Four basic list operations are: **2M**

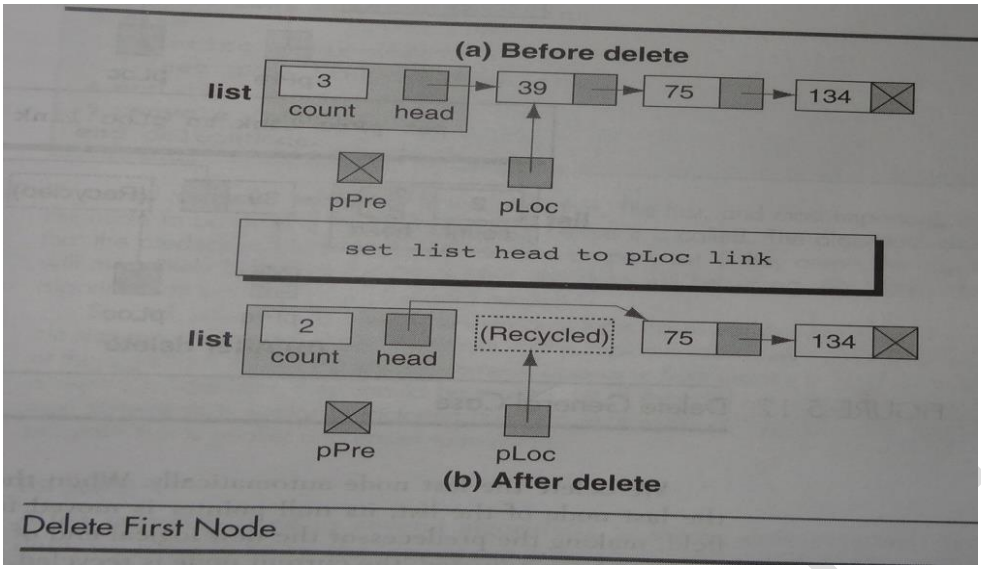
- Insertion
- Deletion
- Retrieval
- traversal

- **Insertion**: it is used to add a new element to the list
- **Deletion**: it is used to remove an element from the list
- **Retrieval**: it is used to get the information related to an element without changing the structure of the list
- **Traversal:** it is used to traverse the list while applying a process to each element

Delete node: 8M

- The delete node algorithm logically removes a node from the list by changing various link pointers and then physically deleting the node from dynamic memory
- To logically delete a node , we must first locate the node itself
- A delete node is located by knowing its address and its predecessors address
- Once we locate we simply change it's predecessor's link field to point to the deleted node's successor
- We then recycle the node back to dynamic memory
- Deleting the only node results in an empty list. So in this case the head is set to a null pointer
- The delete also can have,
  - Delete the only node
  - The first node
  - A node in the middle of the list
  - Or the last node of a list
- The above four can be made as:
  - Delete the first node
  - Delete any other node
- In all the cases the node to be deleted is identified by a pointer ploc

- **Delete first node**
  - When we delete the first node, we must reset the head pointer to the first node's successor and then recycle the memory for the deleted node
  - If the predecessor is a null pointer then we say we are deleting the first node
  - This is shown in fig



- The pseudo code is

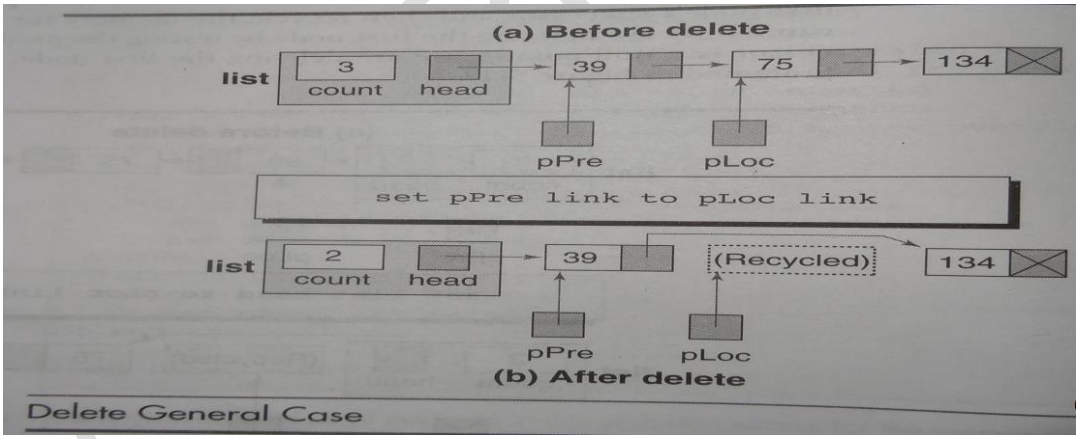
  Set list head to ploc link

  Recycle(ploc)

- This logic can be applied only when we are deleting the only node in the list

General delete case:

- The same logic can be applies to delete any node either in the middle or at the end of the list
- i.e., the predecessor node to the successor of the node being deleted
- The logic is as shown in fig



- The pseudo code is

  Set ppre link to ploc link

  Recycle (ploc)

Delete node algorithm

Algorithm: list delete node

Algorithm deletenode(list, ppre, ploc, dataout)

Deletes data from list & returns it to calling module

Pre list is metadata structure to a valid list

pre is a pointer to predecessor node

ploc is a pointer to node to be deleted
dataout is variable to receive deleted data
Post data have been deleted and returned to caller

1. Move ploc data to dataout
2. if(ppre null)

deleting first node

   1. set list head to ploc link

3. else

deleting other nodes

   1. set ppre link to ploc link

4. end if
5. recycle(ploc)

end deletenode

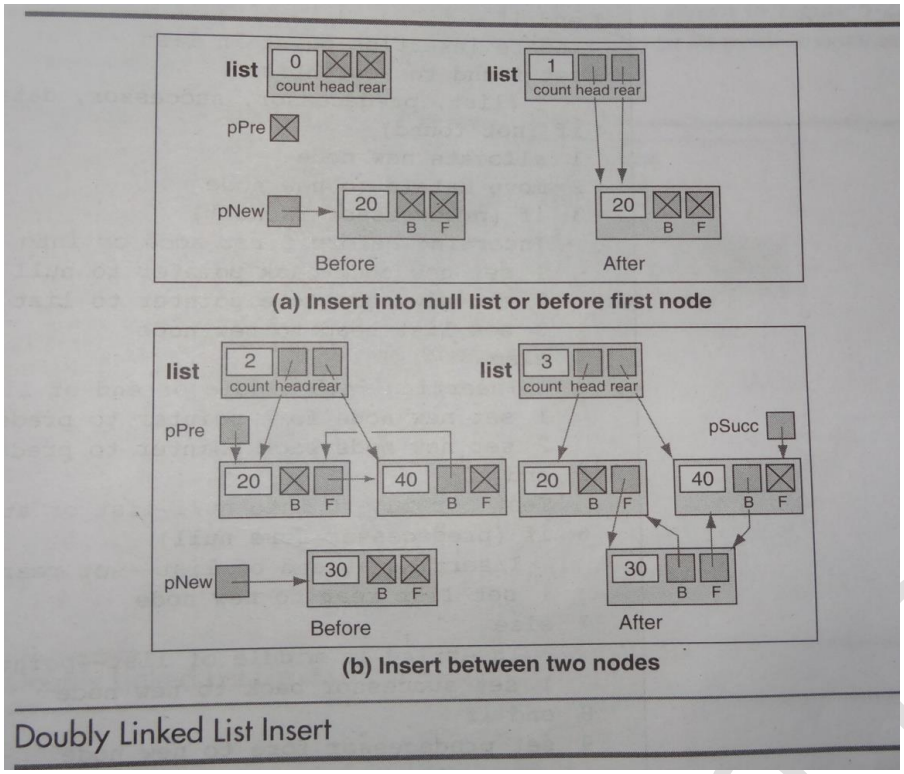
---

2. Write and explain algorithms for inserting and deleting an element from doubly linked list. (Dec – 2019)

Ans:

Insertion – 5M

- Inserting follows same as inserting node into the singly linked list but we need connect both forward and backward pointers
- A null doubly linked list head and rear pointers are null
- To insert a node into a null list, we simply set the head rear pointers to point to the new node and set the forward and backward pointers of the new node to null
- The results of inserting node into a null list are shown in fig.



- Fig(b) shows the case for inserting between two nodes
- The new node need to be set to point to both its predecessor and its successor, and they need to be set to point to the new node
- Because the insert is in the middle of the list, the **head structure is unchanged**
- Inserting at the end of the list requires that the new node's backward pointer to be set to point to its predecessor
- Because there is no successor, the forward pointer is set to null
- The rear pointer in the head structure must also be set to point to the new rear node

Algorithm insertdbl(list,dataIn)
This algorithm inserts data into a doubly linked list

Pre	list is metadata structure to a valid list
	dataIn contains the data to be inserted
Post	the data have been inserted in sequence
Return	0: failed – dynamic memory overflow
	1: successful
	2: failed – duplicate key presented

1. If(full list)
   1. return 0
2. end if

   locate insertion point in list
3. set found to searchlist(list, predecessor, successor, dataIn, key)
4. if(not found)
   1. allocate new node
   2. move dataIn to new node

3. if(predecessor is null)
    4. set new node back pointer to null
    5. set new node fore pointer to list head
    6. set list head to new node
4. else

inserting into middle or end of list

    3. set new node fore pointer to predecessor fore
    4. set new node back pointer to predecessor
5. end if

Test for insert into null list or at end of list

6. if(predecessor fore null)

inserting at end of list – set rear pointer

    2. set list rear to new node
7. else

inserting in middle of list – point successor to new

    2. set successor back to new node
8. end if
9. set predecessor fore to new node
10. return 1

5. end if

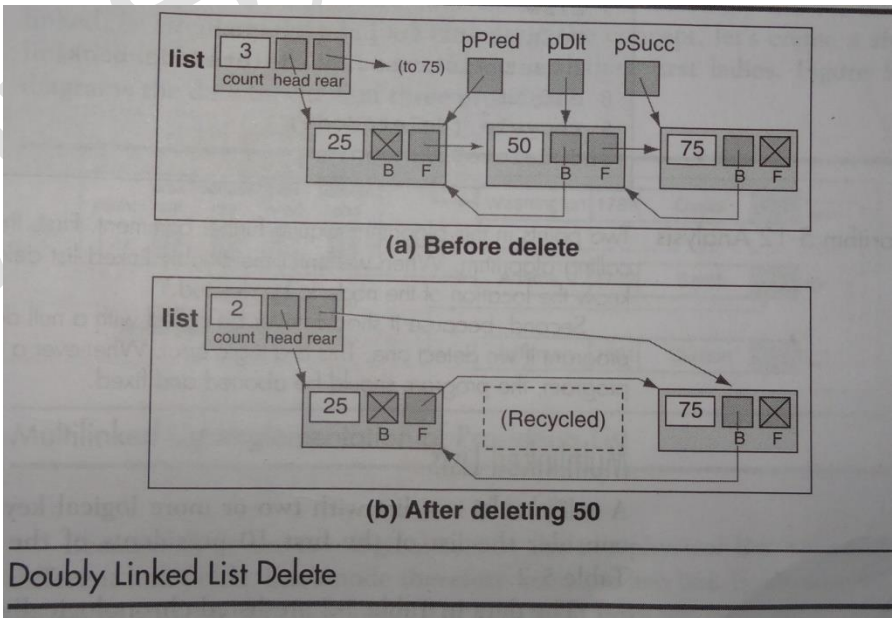
    duplicate data key already exists

6. return 2

end insertdbl

Deletion: 5M

- Deleting requires that the deleted nodes predecessor, if present, be pointed to the deleted node's successor and that the successor, if present, be set to point to the predecessor
- As show in fig



- Once we locate the node to be deleted, we change its predecessors and successors pointers and recycle the node

Algorithm deleteDbl(list, deleteNode)

This algorithm deletes a node from doubly linked list

Pre list is metadata structure to a valid list
deleteNode is a pointer to the node to be deleted

Post node deleted

1. If(deletedNode null)
    1. abort("impossible condition in delete double");
2. end if
3. if(deleteNode back not null)

point predecessor to successor

1. set predecessor to deleteNode back
2. set predecessor fore to deleteNode fore

4. else updatehead pointer
    1. set list head to deleteNode fore
5. end if
6. if(deleteNode fore not null)

point success to predecessor

1. set successor to deleteNode fore
2. set successor back to deleteNode back

7. else

point rear to predecessor

1. set list rear to deleteNode back

8. endif
9. recycle(deleteNode)

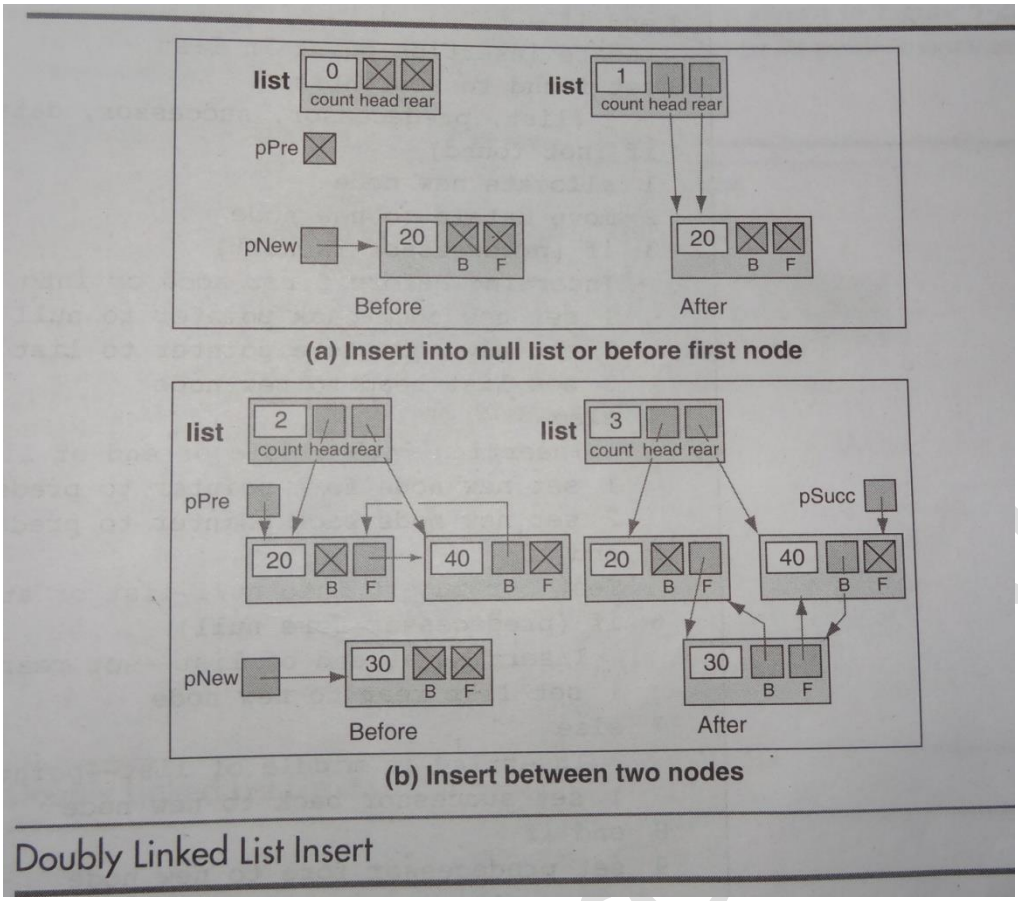
end deleteDbl

3. In how many ways a new node can be inserted into a doubly linked list? Explain them with neat sketch. (Dec – 2019)

Ans:

Insertion – 10M

- Inserting follows same as inserting node into the singly linked list but we need connect both forward and backward pointers
- A null doubly linked list head and rear pointers are null
- To insert a node into a null list, we simply set the head rear pointers to point to the new node and set the forward and backward pointers of the new node to null
- The results of inserting node into a null list are shown in fig.



- Fig(b) shows the case for inserting between two nodes
- The new node need to be set to point to both its predecessor and its successor, and they need to be set to point to the new node
- Because the insert is in the middle of the list, the **head structure is unchanged**
- Inserting at the end of the list requires that the new node's backward pointer to be set to point to its predecessor
- Because there is no successor, the forward pointer is set to null
- The rear pointer in the head structure must also be set to point to the new rear node

Algorithm insertdbl(list,dataIn)

This algorithm inserts data into a doubly linked list

Pre	list is metadata structure to a valid list
	dataIn contains the data to be inserted
Post	the data have been inserted in sequence
Return	0: failed – dynamic memory overflow
	1: successful
	2: failed – duplicate key presented

1. If(full list)
   1. return 0
2. end if

locate insertion point in list

3. set found to searchlist(list, predecessor, successor, dataIn, key)
4. if(not found)
    1. allocate new node
    2. move dataIn to new node
    3. if(predecessor is null)
        1. set new node back pointer to null
        2. set new node fore pointer to list head
        3. set list head to new node
    4. else

    inserting into middle or end of list
        1. set new node fore pointer to predecessor fore
        2. set new node back pointer to predecessor
    5. end if

    Test for insert into null list or at end of list
    6. if(predecessor fore null)

    inserting at end of list – set rear pointer
        1. set list rear to new node
    7. else

    inserting in middle of list – point successor to new
        1. set successor back to new node
    8. end if
    9. set predecessor fore to new node
    10. return 1
    11. end if

        duplicate data key already exists
    12. return 2

end insertdbl

4. What is a Circular linked list and illustrate it with appropriate example. Write procedures for insertion and deletion operations on circular linked list.

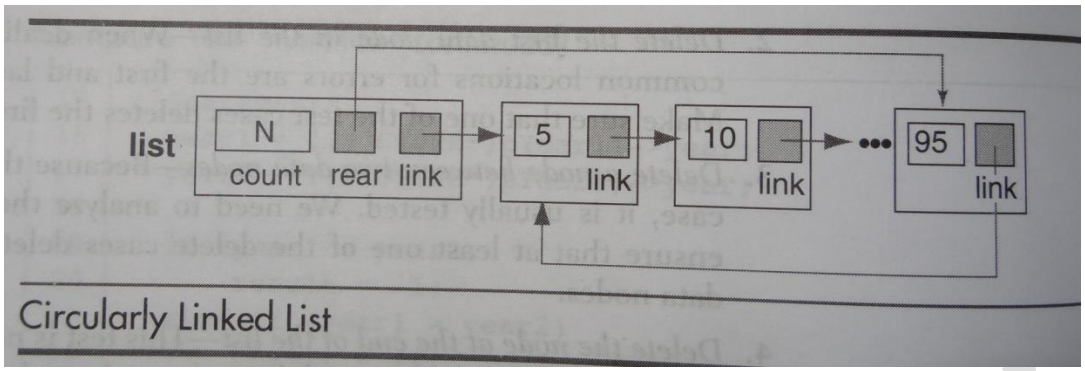
Ans:

Circular linked List: 2M

- In a circularly linked list implementation the last node's link points to first node of the list as shown in fig.

- It is primarily used in lists that allow access to nodes in the middle of the list without starting at the beginning
- Insertion into and deletion from a circular linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node
- Therefore, when inserting or deleting the last node, in addition updating the rear in header we must also point the link field to the first node

Example: 2M



- Given that we can directly access a node in the middle of the list through its data structure, we are then faced with a problem when searching the list
- If the search target lies before the current node, how do we find it?
- In a singly linked list implementation, however, we automatically continue the search from the beginning of the list
- The ability to continue also presents another problem
- If the target does not exist, in singly linked list we stop **when it reaches the end of the list or we target was less than the current node's data**
- With a circular list, we save starting node's address and stop when we have circled around to it, as shown in the code below:

Loop(target not equal to pLoc key AND pLoc link not equal to startAddress)

Insertion operation procedure: 3M

```
void insert()
{
pnew=(dnode*)malloc(sizeof(dnode));
printf("enter the data to be inserted\n");
scanf("%d",&pnew->data);
pnew->link=NULL;
if(ppre==NULL)
{
pnew->link=plist->head;
plist->head=pnew;
if(plist->count==0)
plist->rear=pnew;
last=plist->rear;
last->link=pnew;
}
else
{
pnew->link=ppre->link;
ppre->link=pnew;
if(pnew->link==plist->head)
```

```
{
plist->rear=pnew;
}
}
plist->count++;
}
```

Deletion operation procedure: 3M

```
void del()
{
if(ppre==NULL)
{
plist->head=ploc->link;
last=plist->rear;
last->link=plist->head;
}
else
ppre->link=ploc->link;
if(ploc->link==plist->head)
{
plist->rear=ppre;
last=plist->rear;
last=plist->head;
}
plist->count--;
free(ploc);
}
```

5. Write an algorithm to perform the following operation on singly linked list.
(i) Insert node at the beginning of list.
(ii) Insert new node at middle.
(iii) Delete a node in the middle and last.
(iv) Count the number of nodes.

Ans:

(i) Insert node at the beginning of list. 2M

1. if(ppre null)

adding before first node or empty list

1. set pnew link to list head
2. set list head to pnew

(ii) Insert new node at middle. 2M

1. If(ppre not null)

adding in middle or at end

   1. set pnew link to ppre link
   2. set ppre link to pnew

(iii) Delete a node in the middle and last. 4M

Algorithm: list delete node

Algorithm deletenode(list, ppre, ploc, dataout)

1. Move ploc data to dataout
2. if(ppre null)

deleting first node

   1. set list head to ploc link

3. else

deleting other nodes

   1. set ppre link to ploc link

4. end if
5. recycle(ploc)

end deletenode

(iv) Count the number of nodes. 2M

Algorithm listcount(list)

Returns integer representing number of nodes in list

Pre list is metadata structure to a valid list

Return count for number of nodes in list

1. return (listcount)

end listcount

6. Explain in detail about polynomials using singly linked lists. 10M

Ans:

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

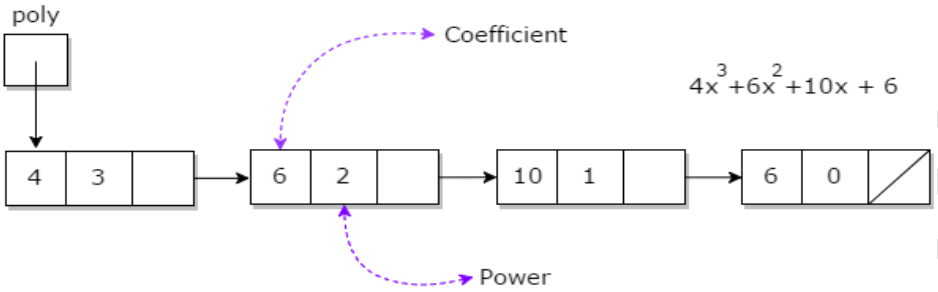
An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



C Implementation:

```
struct node
{
   int cof;
   int exp;
   struct node *link;
};
struct node * create(struct node *q)
 {
  int i,n;
  printf("enter the number of nodes");
  scanf("%d",&n);
  struct node *ptr=(struct node *)malloc (sizeof(struct node));
  for(i=0;i<n;i++)
  {
              printf("entre the coefficient and exponent respectivly");
              scanf("%d%d",&ptr->cof,&ptr->exp);
              ptr->link=NULL;
              q=insert(ptr,q);
  }
  return q;
 }
struct node * insert(struct node *ptr,struct node *p)
 {
  struct node *temp,*b;
  if(p==NULL)
  p=ptr;
  else
  {
```

```
    if((p->exp)<(ptr->exp))
    {
                    ptr->link=p;
                    p=ptr;
    }
    else
    {
       temp=p;
       while((temp!=NULL)||((temp->link->exp)<(ptr->exp)))
       temp=temp->link;
       b=temp->link;
       temp->link=ptr;
       ptr->link=b;
    }
 }
 return p;
 }
void display(struct node *ptr)
 {
  struct node *temp;
 temp=ptr;
 while(temp!=NULL)
 {
            printf("%d x ^ %d + ",temp->cof,temp->exp);
            temp=temp->link;
 }
}
int main()
 {
 printf("enter the first polynomial");
 struct node *p1=NULL,*p2=NULL;
 p1=(struct node *)malloc(sizeof(struct node));
 p2=(struct node *)malloc(sizeof(struct node));
 p1=create(p1);
 printf("entr secon dpolynimial");
 create(p2);
 display(p1);
 display(p2);
 getch();
 return 0;
}
```

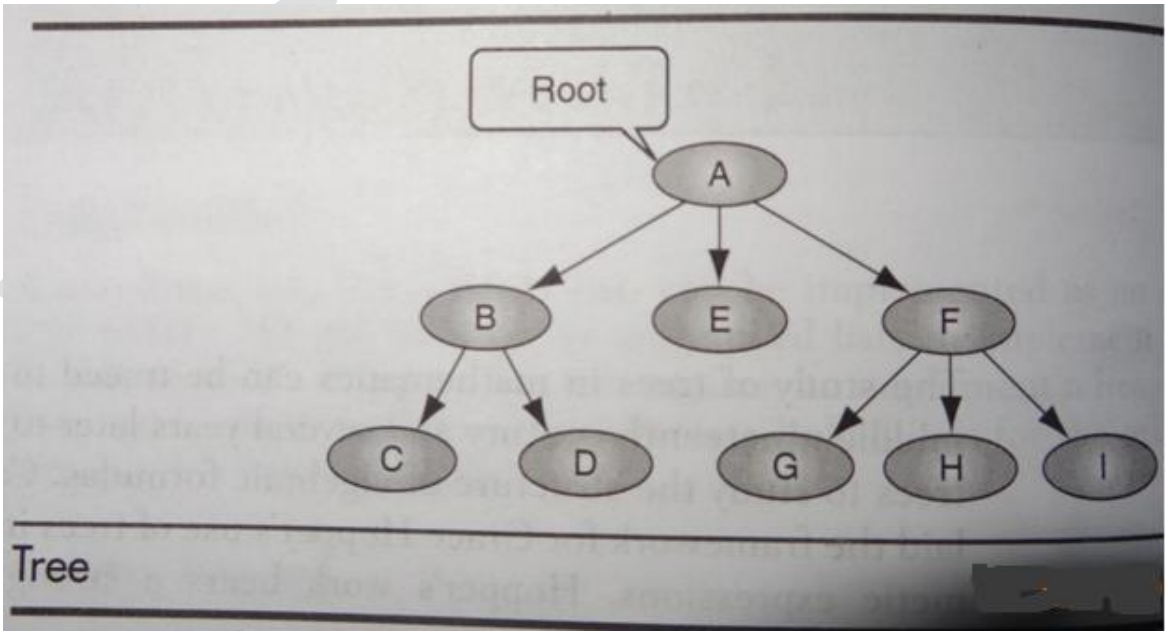
Unit – 5

Trees - Tree terminology, representation, Binary trees, representation, binary tree traversals. binary tree operations, Graphs - graph terminology, graph representation, elementary graph operations, Breadth First Search (BFS) and Depth First Search (DFS), connected components, spanning trees. Searching and Sorting – sequential search, binary search, exchange (bubble) sort, selection sort, insertion sort.

5.1. Trees:

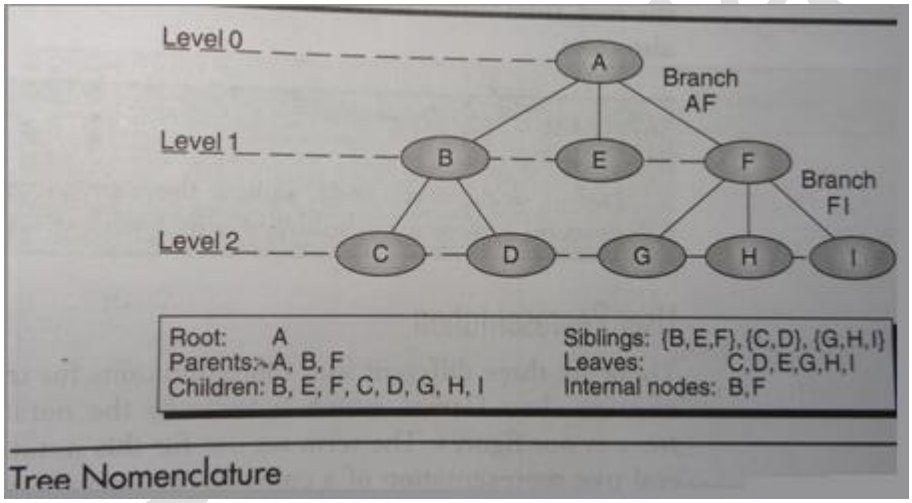
Basic tree concepts

- A **tree** consists of a finite set of elements called **nodes**, and a finite set of directed lines, called **branches**, that connect the nodes
- The number of branches associated with a node is the **degree** of the node
- When the branch is directed toward the node, it is an **indegree** branch; when the branch is directed away from the node, it is an **outdegree** branch
- The sum of the indegree and outdegree branches is the degree of the node.
- **A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes**
- If the tree is **not empty**, the first node is called the **root.** The indegree of the root is, by definition, zero
- The indegree of the root is, by definition, zero
- With the exception of the root, all of the nodes in a tree must have an indegree of exactly one; that is they may have only **one predecessor**
- All nodes in the tree can have zero, one, or more branches leaving them; that I, they may have an outdegree of zero, one, or more (zero or more successors)
- Fig. is a representation of a tree



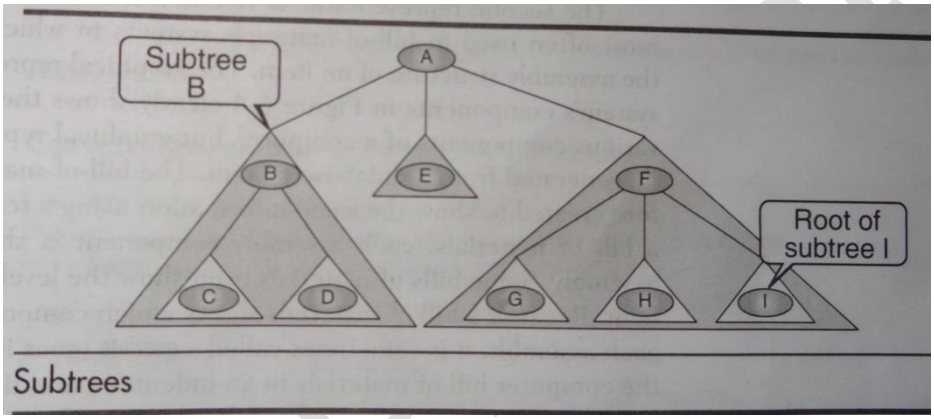
5.2. Tree Terminology:

- In addition to root, many different terms are used to describe the attributes of a tree
- A **leaf** is any node with an outdegree of zero, that is, a node with no successors
- A node that is not a root or a leaf is known as an **internal node** because it is found in the middle portion of a tree
- A node is a **parent** if it has successor nodes that
- is if it has an outdegree greater than zero
- Conversely, a node with a predecessor is a **child**
- A **child** has an **indegree of one**
- Two or more nodes with the same parent are **siblings**
- An **ancestor** is any node in the path from root to the node
- An **descendent** is any node in the path below the parent node; that is, all nodes in the paths from a give node to a leaf are descendents of that node
- Fig. shows the usage of these terms



- A **path** is a sequence of nodes in which each node is adjacent to the next one
- Every node in the tree can be reached by following a unique path starting from the root
- In fig. the path form the root to the leaf I is designated as AFI
- It includes two distinct branches, AF and FI
- The **level** of a node is its distance from the root
- Because the root has a zero distance from itself, the root is at level 0
- The children of the root are at level 1, their children at level 2, and soforth
- Note the relationships between levels and siblings and in fig. above
- Siblings are always at the same level but all nodes in a level are no necessarily siblings
- For ex, at level2 C and D are siblings, as are G, H and I
- However D and G are not siblings because they have different parents
- The **height** of the tree is the level of the leaf in the longest path from the root + 1
- By definition the height of the empty tree is -1
- Fig. above contains nodes at three levels 0,1, and 2
- Its height is 3

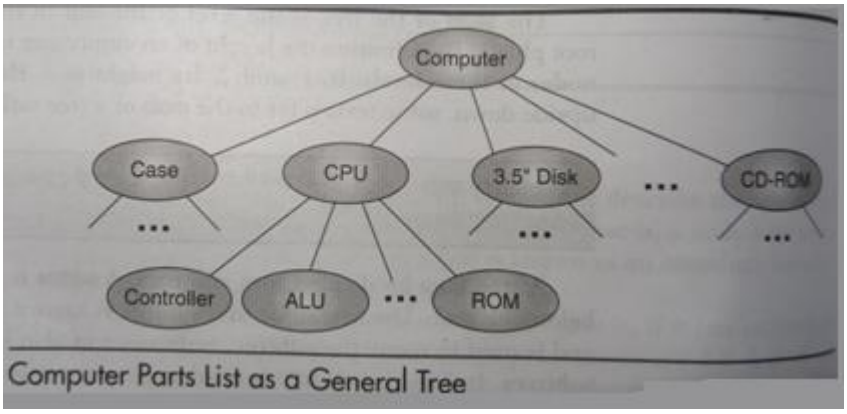
- Because the tree is drawn upside down, some texts refer to the depth of a tree rather than its height
- **The level of a node is its distance from the root. The height of a tree is the level of the leaf in the longer path from the root plus 1**
- A tree may be divided into subtrees
- A subtree is any connected structure below the root
- The first node in a subtree is known as the root of the subtree and is used to name the subtree
- Subtrees can also be further divides into subtees
- In fig below, BCD is a subtree as are E and FGHI
- Note that by this, a single node is a subtree
- Thus, the subtree B can be divided into two subtrees C and D, and the subtree F contains the subtrees G, H and I



- The concept of subtrees leads us to a recursive definition of a tree: a tree is a set of nodes that either:
  1. Is empty or
  2. Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees

5.3. Representation:

- There are three different user representations of trees are
  1. **Organization chart format**
- It is basically the notation we use to represent trees in our figures
- The term we use for this notation in **general tree**
- The general tree representation of computers components is shown in fig.



Computer Parts List as a General Tree

2. Indented list

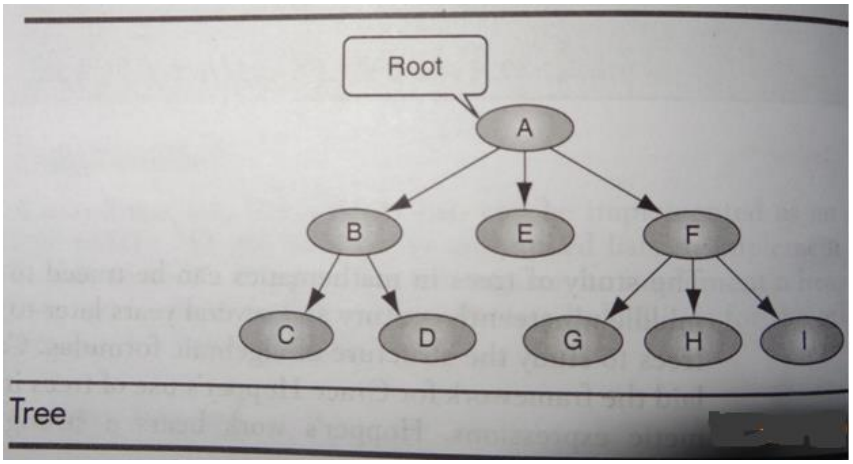
- You will find this used in bill of materials systems in which a parts list represents the assembly structure of an item
- In fig. above clearly shows the relationship among the various components of a computer, but graphical representations are not easily generated from a database syste,
- The bill – of – materials format was created to show the same information using a textual parts list format
- In a bill of materials, each assembly component is shown indented below to assembly
- Some bills of materials even show the level number of each component
- Because a bill of materials shows which components are assembled into each assembly it is sometimes called a **goezinto(goesinto)**list
- Table shows the computer bill of materials in an indented parts list format

Part number	Description
301	Computer
301-1	Case
...	...
301-2	CPU
301-2-1	Controller
301-2-2	ALU
...	...
301-2-9	ROM
301-3	3.5" Disk
...	...
301-9	CD-ROM
...	...

Computer Bill of Materials

3. Parenthetical listing

- It is used with algebraic expressions
- When a tree is represented in parenthetical notation, each open parenthesis indicates the start of a new level; each closing parenthesis completes the current level and moves up one level in the tree
- Consider the tree & its parenthetical notation is A(B(CD)EF(GHI))



- To convert a general tree to its parenthetical notation, we use the code as in below algorithm

Algorithm convertoparen(root,output)
Convert a general tree to parenthetical notation

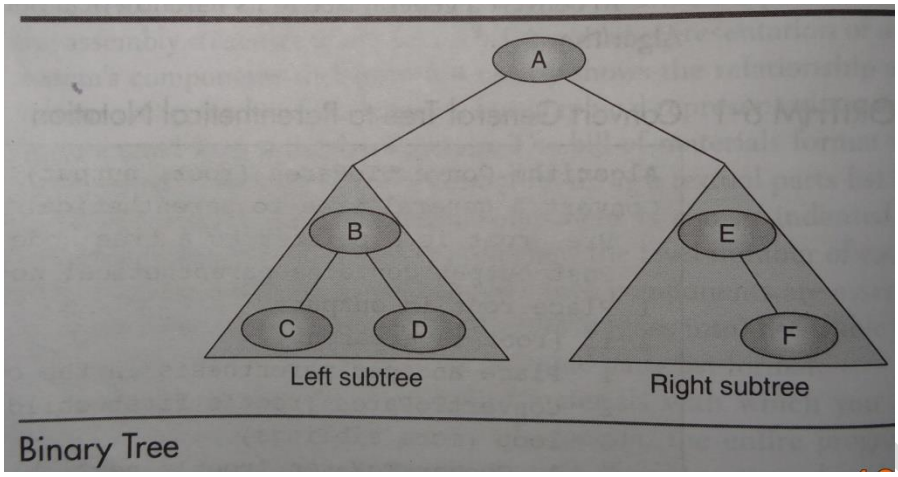
Pre root is a pointer to a tree node
Post output contains parenthetical notation

1. Place root in output
2. If(root is a parent)
    1. Place an open parenthesis in the output
    2. Converttoparen(root's first child)
    3. Loop(more siblings)
        1. Converttoparen(roots next child)
    4. End loop
    5. Place close parenthesis in the output
3. End if
4. Return

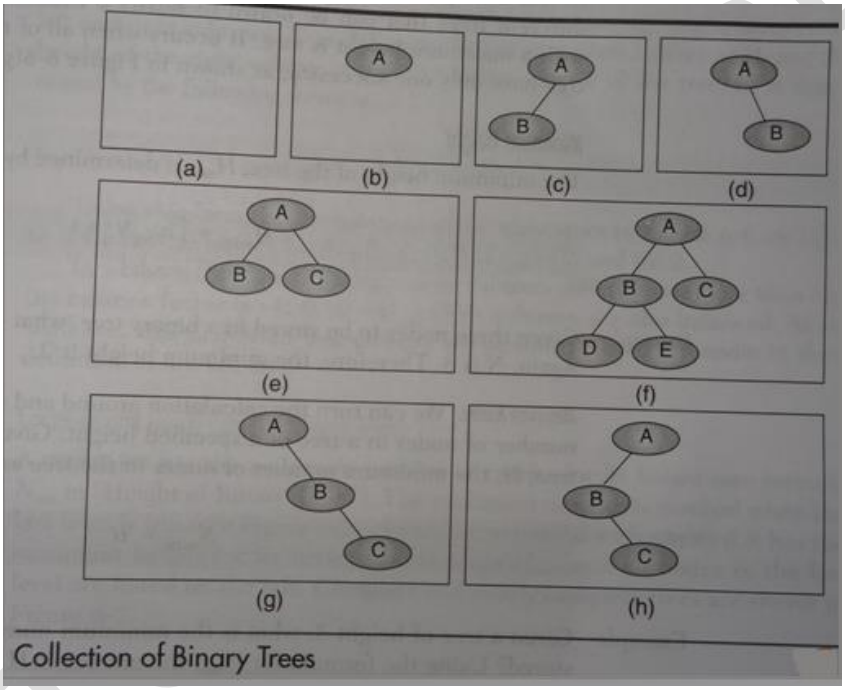
End converttoparen

5.4. Binary Trees and its representation:

- A binary tree is a tree in which no node can have more than two subtrees, the **maximum outdegree** for a node is **two**
- In other words, a node can have zero, one or two subtrees
- The subtrees are designated as the left subtree and right subtree
- Fig. shows a binary tree with its two subtrees
- Note that each subtree is itself a binary tree



- To better understand the structure of binary tree, see the fig.



- Fig. above contains eight binary trees; the first of which is a null tree
- A **null tree** is a tree with no nodes, as shown in fig(a)
- **A node in a binary tree can have no more than two subtrees**

Properties

- Several properties for binary trees that distinguish them from general trees

Height of binary trees

- The height of binary trees can be mathematically predicted

Maximum height

- Given that we need to store N node in a binary tree, the maximum height H_{max} is

$$H_{max}=N$$

- Ex: given,

Nodes = 3

What is maximum height?

In this ex N is 3

Maximum height = 3

- There are four different trees that can be drawn to satisfy a maximum height = 3
- A tree with maximum height is rare
- It occurs, when all nodes in the entire tree have only one successors shown in fig(g) and fig(h)

Minimum Height

- The minimum height of the tree, H_{min} is determined by the following formula:

$$H_{min} = [\log_2 N]+1$$

- Ex: Given,

  Nodes = 3

  What is minimum height?

  N=3

  The minimum height = 2

Minimum nodes. We can turn the calculation around and determine the minimum number of nodes in a tree of a specified height

- Given a height of the binary tree, H, the minimum number of the nodes in the tree are given as

$$N_{min} = H$$

- Ex: Given a tree of height = 3

  $N_{min}=H$

  N_{min} = 3 nodes

as seen in fig(g) and fig(h)

Maximum Nodes

- The formula for the maximum number of nodes is derived from the fact that each node can have only two descendents
- The height of the binary tree H, the maximum number of nodes in the tree is given as

$$N_{max} = 2^H - 1$$

- Ex: H=3

N_{max} = 7

Balance

- The distance of a node from the root determines how efficiently it can be located
- The children of any node in a tree can be accessed by following only one branch path, the one that leads to the desired node
- The nodes at level 1, which are children of the root, can be accessed by following only two branches from the root
- It stands to reason, therefore, that the sorter the tree, the easier it is to locate any desired node in the tree

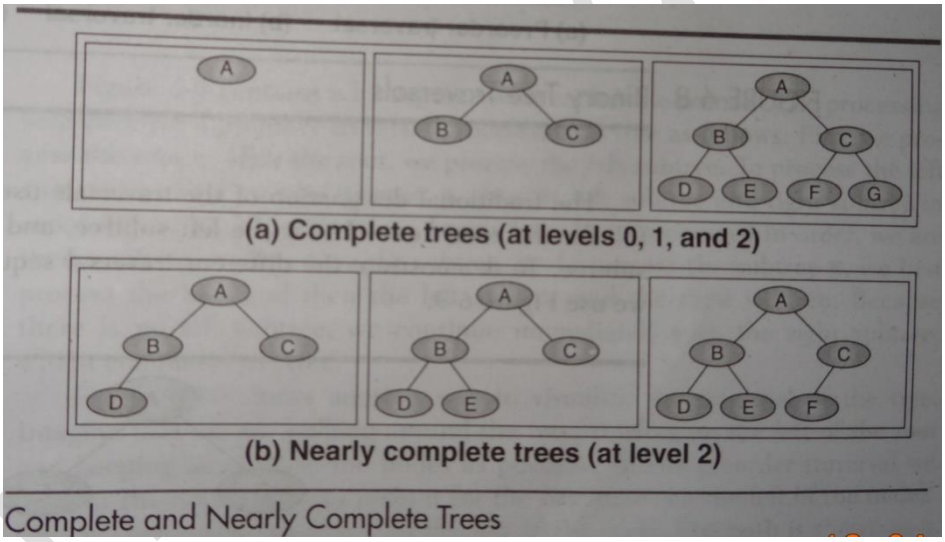
- The concept leads us to a very important characteristic of a binary tree – its balance
- To determine whether a tree is balanced, we calculate its balance factor
- The balance factor of a binary tree is the difference in height between its left and right subtrees
- If we define the height of the left subtree as H_L and the height of the right subtree as H_R, the balance factor of the tree, B, is determined by the following:

$$B=H_L - H_R$$

- Using this formula, the balances of the eight trees in fig are a. 0 by definition b. 0, c. 1, d. -1, e. 0, f. 1, g. 2, and h.2
- In a balanced binary tree, the height of its subtree differs by no more than one(its balance factor is -1, 0, or +1), and its subtrees are also balanced

Complete and Nearly complete binary trees

- A **complete tree**has the maximum number of entries for its height(see N_{max} in "height of binary trees")
- The maximum number is reached when the last level is full, see fig.
- A tree is considered **nearly complete** if it has the minimum height for its nodes(see formula H_{min}) and all nodes in the last level are found on the left
- Complete and nearly complete trees are shown in fig.



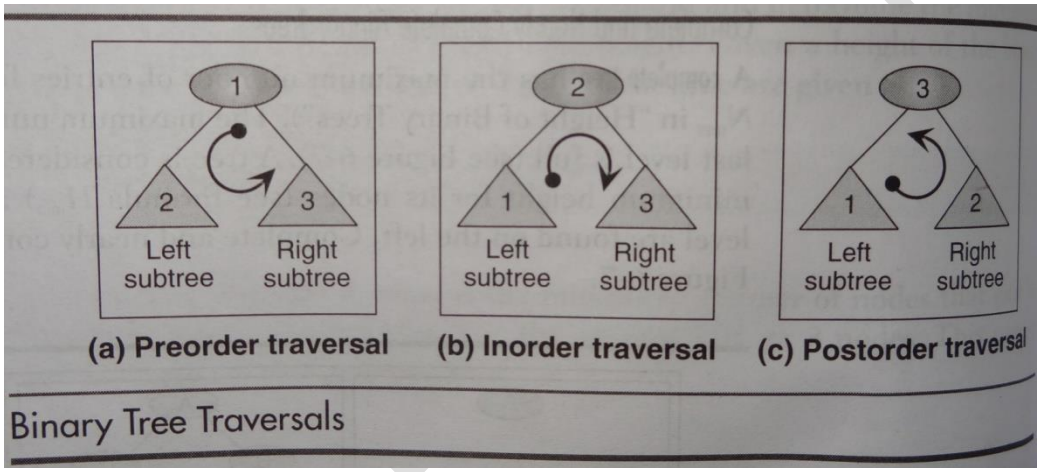
5.6. Binary tree traversals

- A **binary tree traversal** requires that each node of the tree be processed once and only once in a predetermined sequence
- The two general approaches to the traversal sequence are:
    a. Depth first b. breadth first
- In the depth first traversal the processing proceeds along a path from the root through one child to the most distant descendant of that first child before processing a second child
- In other words, in the depth first traversal, we process the descendants of a child before going on to the next child

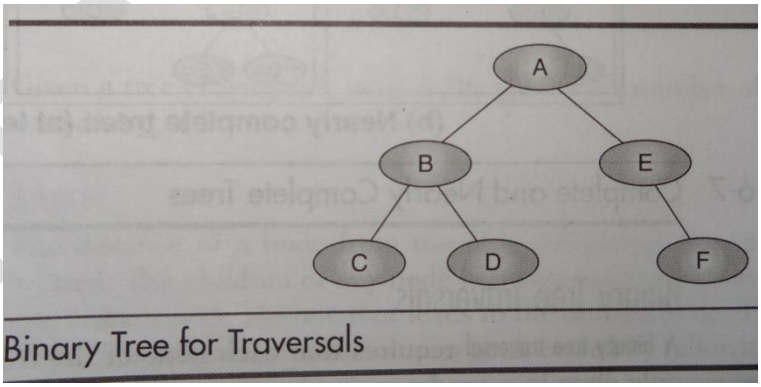
- In a breadth first traversal, the processing proceeds horizontally from the root to all of its children, then to its children's children, and so forth until all nodes have been processed
- In other words, in the breadth first traversal each level is completely processed before the next level is started

Depth first traversal

- Given a binary tree consists of a root, left subtree, and a right subtree, we can define six different depth – first traversal sequences
- Computer scientists have assigned three of these sequences standard names in the literature, the other three are unnamed but are easily derived
- The standard traversals are shown in fig.
- The traditional designation of the traversals uses a designation of node(N) for the root, left(L) for the left subtree and right(R) for the right subtree



- To demonstrate the different traversal sequences for a binary tree we use fig



Preorder Traversal(NLR)

- In the preorder traversal, the root node is processed first, followed by the left subtree and then the right subtree
- It draws its name from the latin prefix pre, which means to go before
- Thus, the root goes before the subtrees
- In the preorder traversal, the root is processed first, before its subtrees

- Given the recursive characteristic of trees, it is only natural to implement tree traversals recursively
- First we process the root, then the left subtree, and then the right subtree
- The left subtree inturn processed recursively as the right subtree
- The code for the preorder traversal is shown in algorithm

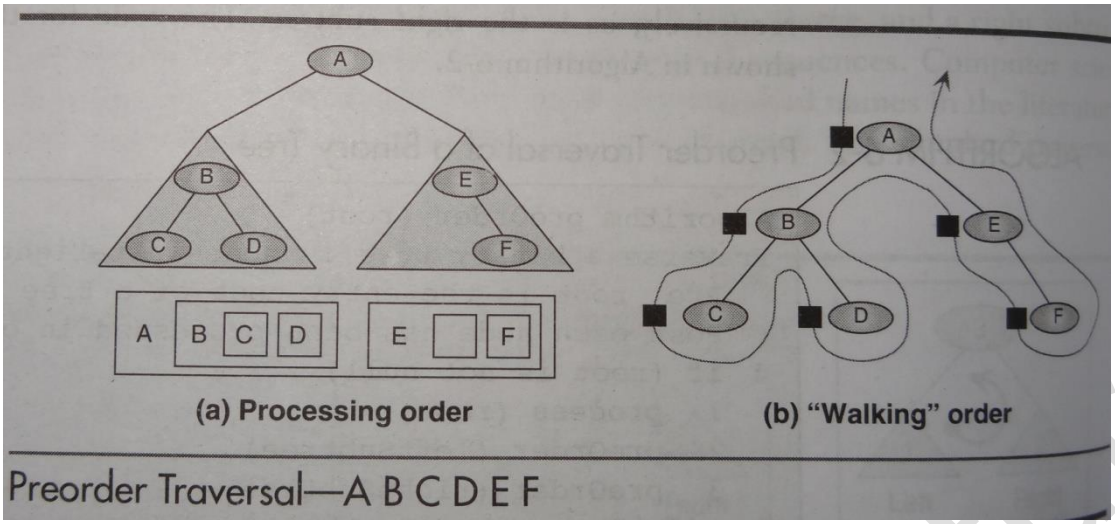
Algorithm preorder(root)
Traverse a binary tree in node – left – right sequence

Pre	root is the entry node of a tree of subtree
Post	each node has been processed in order

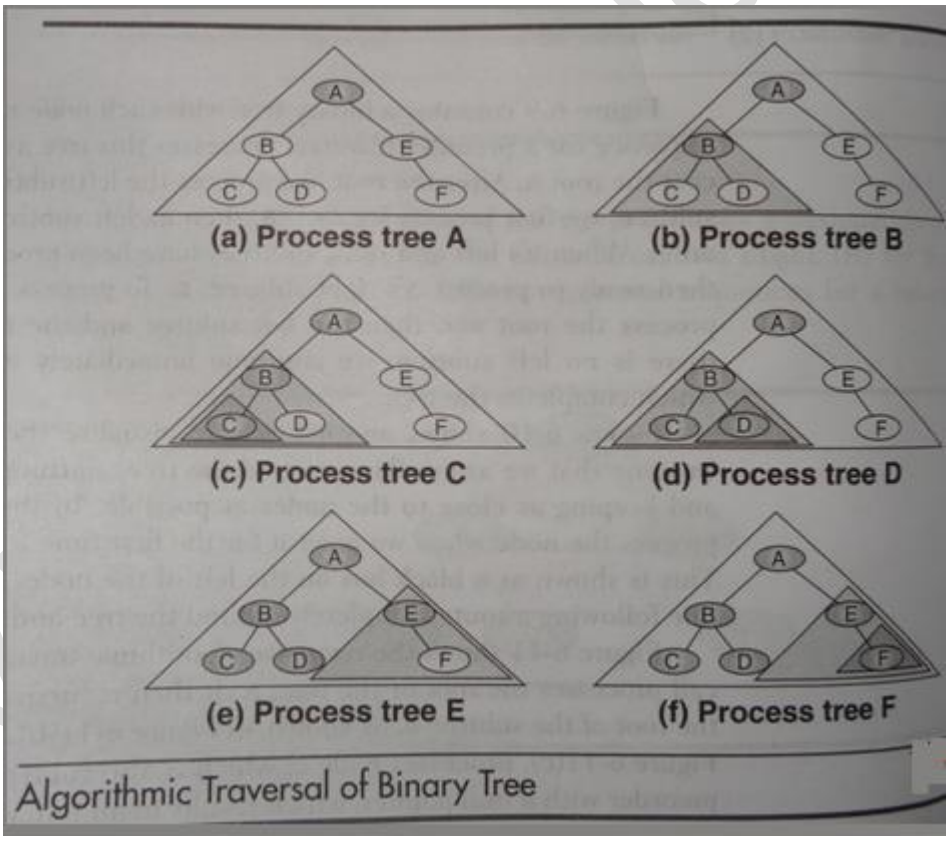
1. If(root is not null)
    1. Process(root)
    2. Preorder(leftsubtree)
    3. Preorder(rightsubtree)
2. End if

End preorder

- Fig. above contains a binary tree with each node named
- The processing sequence for a preorder traversal processes this tree as follows
    1. We process the root A
    2. We process the left subtree first
    3. To process the left subtree, we process its root B, then its left subtree and right subtree in order
    4. When B's left and right subtrees have been processed in order we are then ready to process A's right subtree, E
    5. To process the subtree E; we first process the root and then the left subtree and right subtree
    6. Because there is no left subtree, we continue immediately wit right subtree which completes the tree
- Fig. below shows another way to visualize the traversal of the tree
- Imagine that we are walking around the tree, starting on the left of the root and keeping as close to the nodes as possible



- In the preorder traversal we process the node when we meet it for the first time
- This is shown as a black box on the left of the node
- The path is shown as a line following a route completely around the tree and back to the root
- Fig. next shows the recursive algorithmic traversal of the tree



Inorder Traversal(LNR):

- The inorder traversal processes the left subtree first, then the root, and finally the right subtree
- The meaning of the prefix in is that the root is processed in between the subtrees
- Once again we implement the algorithm recursively, as shown below

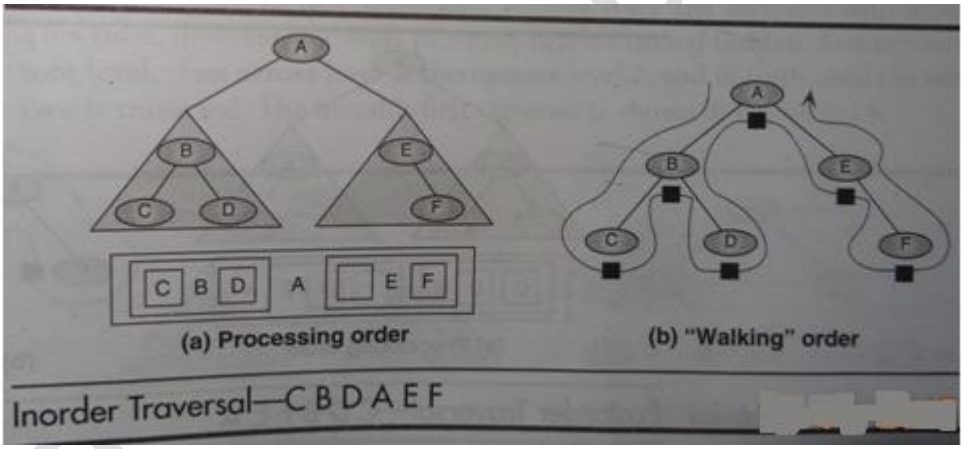
Algorithm Inorder(root)
Traverse a binary tree in left – node– right sequence

Pre root is the entry node of a tree of subtree
Post each node has been processed in order

1. If(root is not null)
    1. Inorder(leftsubtree)
    2. Process(root)
    3. Inorder(rightsubtree)
2. End if

End Inorder

- Because the left subtree must be processed first, we trace from the root to the far left leaf node before processing any node
- After processing the left subtree, C, we process its parent node, B
- We are now ready to process the right subtree, D
- Processing D completes the processing of the root's left subtree, and we are now ready to process the root, A, followed by its right subtree
- Because the right subtree, E, has no left child, we can process its root immediately followed by its rights subtree F
- The complete sequence for inorder processing is shown in fig.



- To walk around the tree in inorder sequence, we follow the same path but process each node when we meet it for the second time
- The processing route is shown in fig(b)
- **In the inorder traversal, the root is processed between its subtrees**

Post order traversal(LRN)

- The last of the standard traversal is the postorder traversal
- It processes the root node after (Post) the left and right subtrees have been processed
- It starts by locating the far – left leaf and processing it
- It then processes its right sibling including its subtrees(if any). Finally it process the root node
- **In the postorder traversal, the root is processed after its subtrees**
- The recursive postorder traversal logic is shown in algorithm

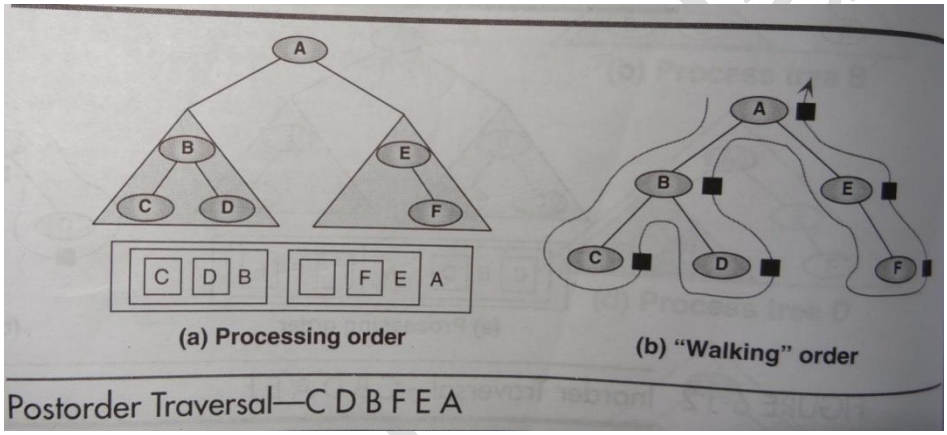
Algorithm postorder(root)
Traverse a binary tree in left– right – node sequence

Pre root is the entry node of a tree of subtree
Post each node has been processed in order

1. If(root is not null)
    1. Postorder(leftsubtree)
    2. Postorder(rightsubtree)
    3. Process(root)
2. End if

End postorder

- In the tree walk for a postorder traversal we move the processing block to the right of the node so that we process it as we meet the node for the third time
- The postorder traversal is shown in fig.
- Note that we took the same path in all three walks; only the time of the processing changed



Breadth first traversals

- In the breadth first traversal of a binary tree, we process all of the children of a node before proceeding with the next level
- In other words, given a root at level n, we process all nodes at level n, we process all nodes at level n before proceeding with nodes at level n+1
- To traverse tree in depth – first order, we used a stack
- To traverse a tree in breadth first order, we use a queue
- The pseudocode for a breadth first traversal of our binary tree is shown in algorithm

Algorithm breadthfirst(root)
Process tree using breadth first traversal

Pre root is node to be processed
Post tree has been processed

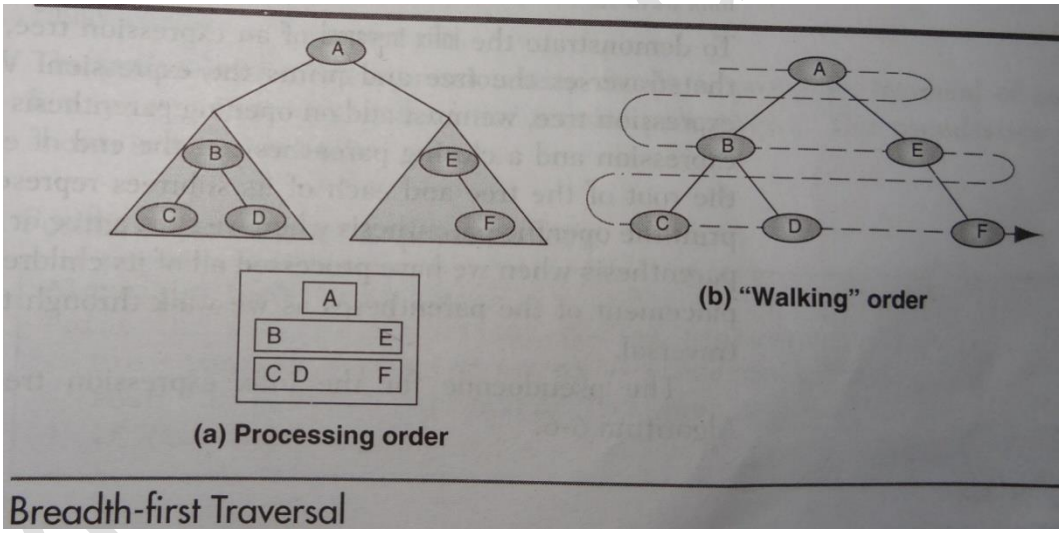
1. Set current node to root
2. Createqueue(bfqueue)
3. Loop(current node not null)

1. Process(current node)
2. If(left subtree not null)
    1. Enqueue(bfqueue, leftsubtree)
3. End if
4. If(right subtree not null)
    1. Enqueue(bfqueue, rightsubtree)
5. End if
6. If(not emptyqueue(bfqueue))
    1. Set currentnode to dequeue(bfqueue)
7. Else
    1. Set current node to null
8. End if

4. End loop
5. Destroyqueue(bfqueue)

End breadthfirst

- Like depth first traversals, we can trace the traversal with a walk
- This time, however, the walk proceeds in a horizontal fashion, first across the root level, the across level 1, then across the level2, and so forth until the entire tree is traversed
- The breadth first traversal is shown in fig.



5.7. Binary tree operations:

- **Searching:** For searching element, we have to traverse all elements (assuming we do breadth first traversal).
- **Insertion:** For inserting element as left child, we have to traverse all elements.
- **Deletion:** For deletion of element, we have to traverse all elements to find delete element (assuming we do breadth first traversal).

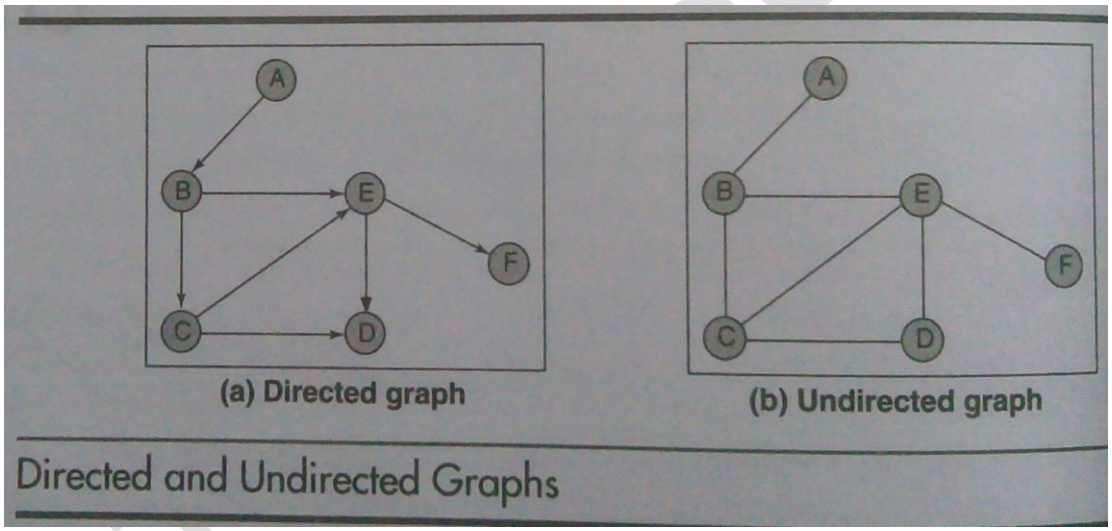
5.8. Graph:

- In the graph each node may have multiple predecessors as well as multiple successors
- Graphs are very useful structures

- They can be used to solve complex routing problems, such as designing an routing airlines among the airports they serve
- A **graph** is a collection of nodes, called vertices, and a collection of segments, called lines, connecting pairs of vertices

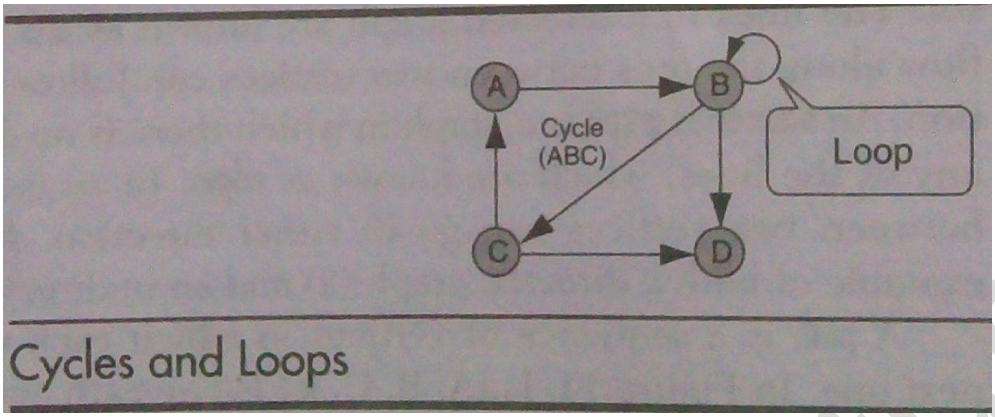
5.9. Graph Terminology:

- In other words, a graph consists of two sets, a set of vertices and a set of lines
- Graphs may be either directed or undirected
- A **directed graph or digraph** for short, is a graph in each line has a direction(arrow head) to its successor
- The lines in a directed graph are known as **arcs**
- An **undirected graph** is a graph in which there is no direction on any of the lines, which are known as **edges**
- Fig contains an example of both a directed graph (a) and an undirected graph (b)
- A **path** is a sequence of vertices in which each vertex is adjacent to the next one
- In fig. {A, B, C, E} is one path and {A, B, E, F} is another
- Note that is both directed(you may travel in indicated direction) and undirected graph(you may travel in either direction) have paths



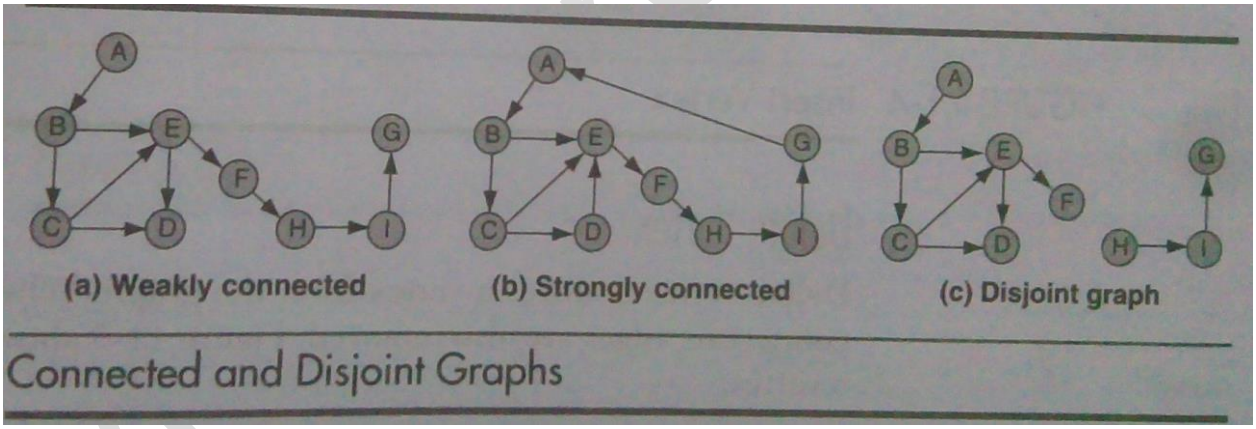
- Two vertices in a graph are said to be adjacent vertices (or neighbors) if there is a path length 1 connecting them
- In fig(a) B is adjacent to A, whereas E is not adjacent to D; on the other hand, D is adjacent to E
- In fig(b), E and D are adjacent, but D and F are not
- A **cycle** is a path consisting of atleast three vertices that starts and ends with the same vertex
- In fig(b) B, C, D, E, B is a cycle
- The same vertices in fig(a) is not a cycle
- A **loop** is a special case of a cycle in which a single arc begins and ends with the same vertex

- In a loop the end points of the line are the same



Cycles and Loops

- Two vertices are said to be connected if there is a path between them
- A graph is said to be connected if, ignoring direction, there is a path from any vertex to any other vertex
- A directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph
- A digraph is **weakly connected** if atleast two vertices are not connected
- A graph shows is a **disjoint graph** if it is not connected
- Fig. shows



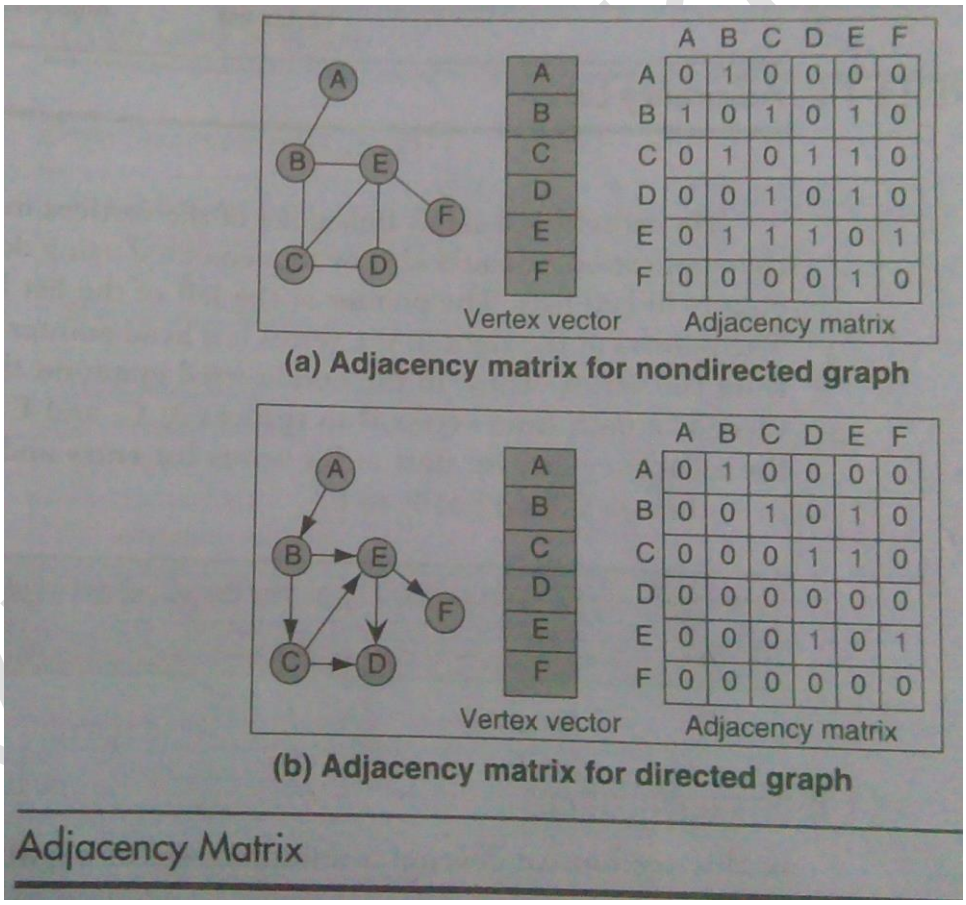
Connected and Disjoint Graphs

- The degree of a vertex is the number of lines incident to it
- In fig(a) degree of vertex B=3 E=4
- The outdegree of a vertex in a digraph is the number of arcs leaving the vertex;
- The indegree is the number of arcs entering the vertex
- In fig(a) the indegree of vertex B is 1 and its outdegree is 2;
- In fig(b) the indegree of vertex E is 3 and its outdegree is 1
- A **tree is a graph** in which each vertex has only one predecessor; however **a graph is not a tree**

5.10. Graph Representation

a. Adjacency matrix

- The adjacency matrix uses a vector(1-D array) for the vertices and a matrix (2 – D array) to store the edges (See fig)
- If vertices are adjacent – that is, if there is an edge between them – the matrix intersect has a value of 1; if there is no edge between them, the intersect is set to 0.
- If the graph is directed, the intersection in the adjacency matrix indicates the direction
- For ex, if fig(b) there is an arc from source vertex B to destination vertex C
- In the matrix, this is arc is seen as 1, in the intersection from B (on the left) to C (on the top)
- Because there is no arc from C to B
- The intersection from C to B is 0
- In fig(a) the edge from B to C is bidirectional that is you can traverse in either direction
- So, from B to C is 1 as well as C to B is also 1
- The matrix reflects the fact that you can use the edge to go either way.



In adjacency matrix representation, we use a vector to store the vertices and a matrix to store the edges

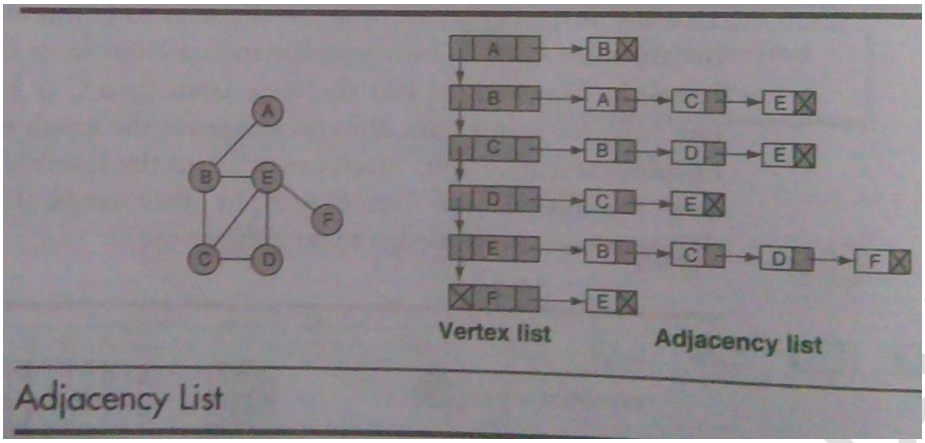
Limitation of adjacency matrix

- I need to know the size of the graph before the program starts

- Only one edge can be stored between any two vertices. Some network structures require multiple lines between vertices

b. Adjacency list

- The adjacency list uses a two – D ragged array to store the edges
- An adjacency list is shown in fig.



- The vertex list is a singly linked list of the vertices in the list
- It could be implemented using a doubly or circularly linked lists
- The pointer at the left of the list links the vertex entries
- The pointer at the left of the list links the vertex entries
- The pointer at the right in the vertex is a head pointer to a linked list of edges from the vertex
- Thus, in the nondirected graph on the left in fig, there is a path from vertex B to vertex A, C, and E
- To find these edges in the adjacency list, we start at B's vertex list entry and traverse the linked list to A, then to C, and finally to E

In the adjacency list, we use a linked list to store the vertices an a 2 – D linked list to store the arcs

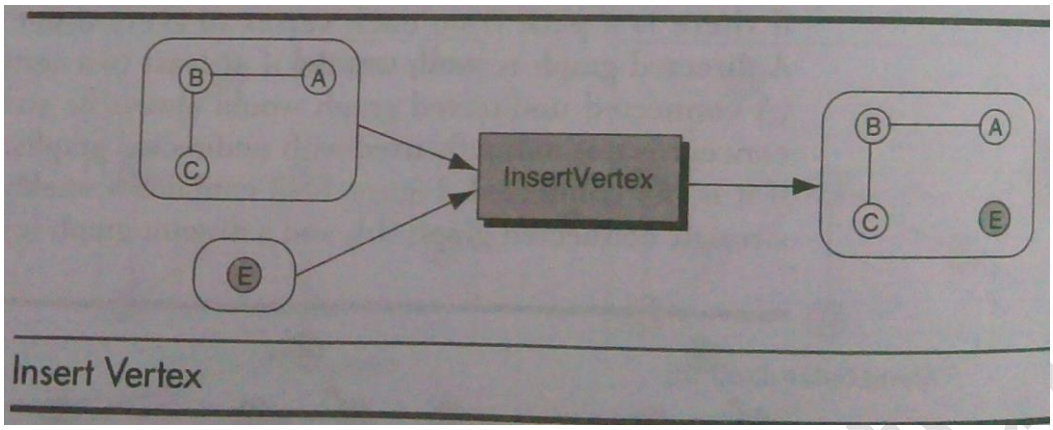
5.11. Elementary graph operations

- Six primitive graph operations that provide the basic modules needed to maintain a graph
  - Insert a vertex
  - Delete a vertex
  - Add an edge
  - Delete an edge
  - Find a vertex
  - Traverse a graph

Insert a vertex

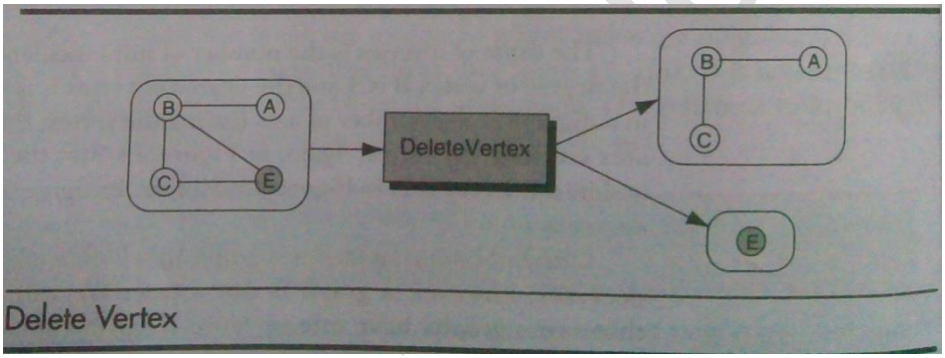
- It adds a new vertex to a graph
- When it is inserted, it is disjoint; that is, it is not connected to any other vertices in the list
- Inserting a vertex is just the first step in the insertion process
- After insertion, it must be connected

- Fig. shows a graph before and after a new vertex is added



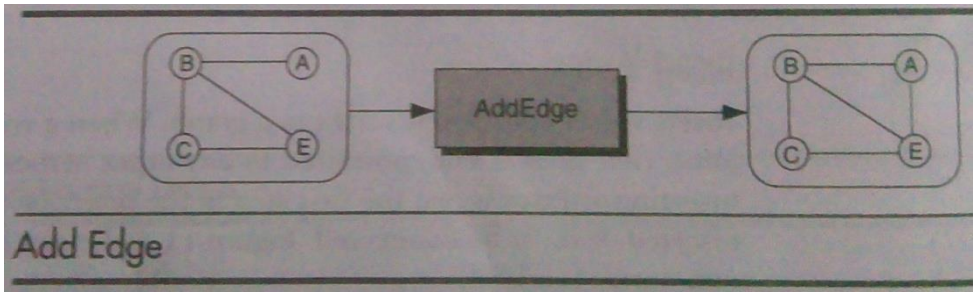
Delete a vertex

- Delete vertex removes a vertex from the graph
- When a vertex is deleted, all connecting edges are also removed
- Fig shows deleting a vertex



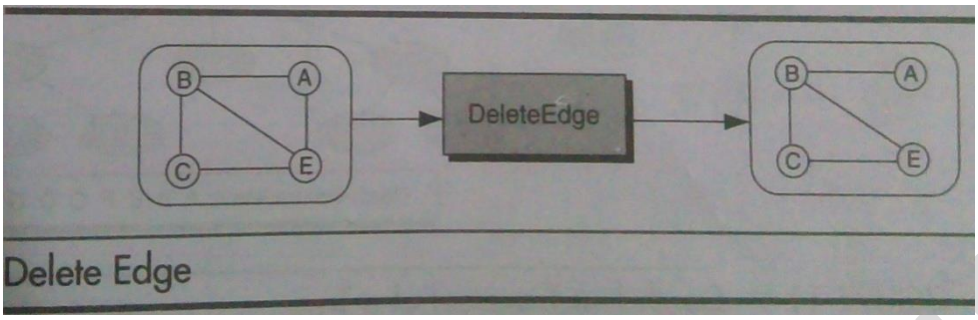
Add edge

- Add edge connects a vertex to a destination vertex
- If a vertex requires multiple edges, add an edge must be called once for each adjacent vertex
- To add an edge, two vertices must be specified
- If the graph is a digraph, one of the vertices must be specified as the source and one as the destination
- Fig shows an examples of adding an edge {A, E}, to the graph



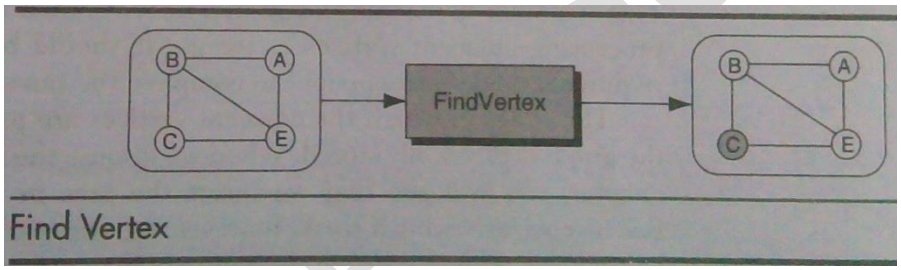
Delete Edge

- Delete edge removes one edge from a graph
- Fig. shows an example that deletes the edge {A, E} from the graph



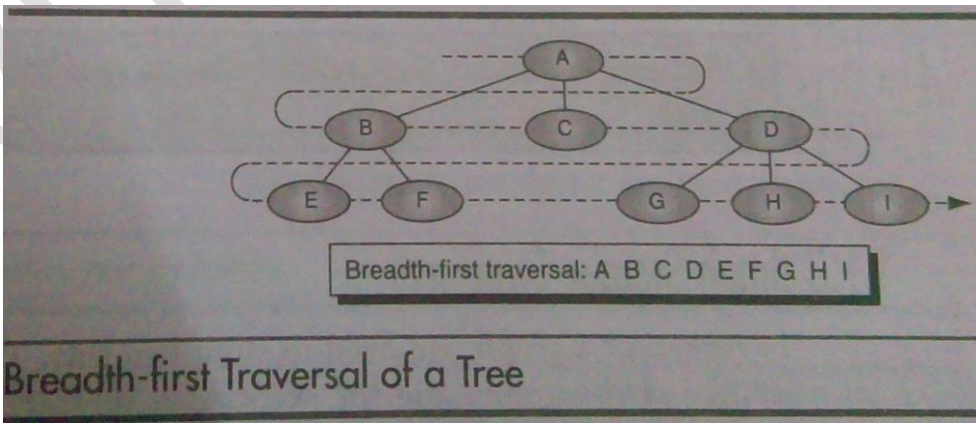
Find vertex

- Find vertex traverses a graph, looking for a specified vertex
- If the vertex is found its data are returned
- If it is not found, an error is indicated
- In fig find vertex traverses the graph, looking for vertex C



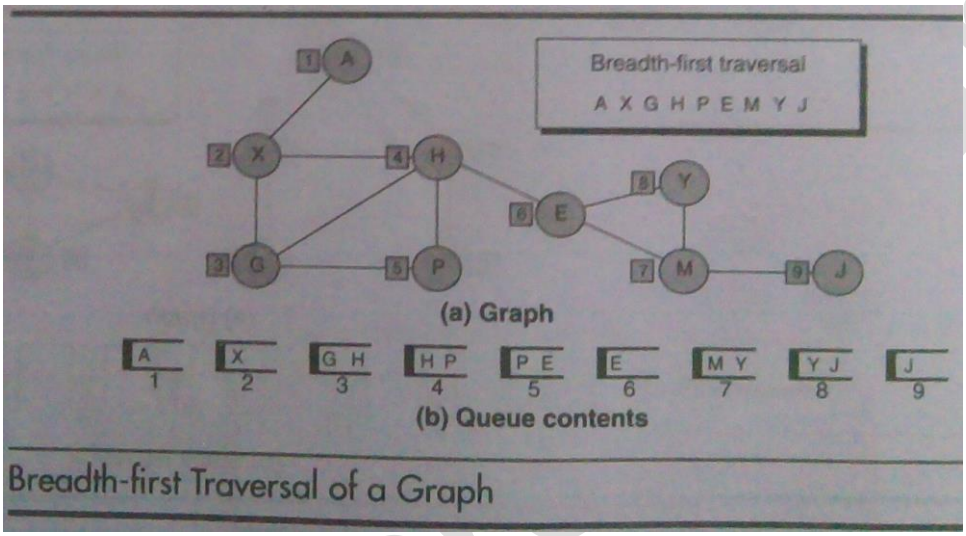
5.12. Breadth first search

- In the BFT of a graph, we process all adjacent vertices of a vertex before going to the next level
- Looking at the tree in fig, we see that its BFT starts at level 0 and then process all the vertices in level 1 before going to process the vertices in level 2



- The BFT of a graph follows the same concept

- We begin at starting vertex (A); after processing it we process all of its adjacent vertices (BCD)
- After we process all of the first vertex's adjacent vertices, we pick its first adjacent vertex(B) and process all of its vertices, and so forth until we are finished
- We show that the BFT uses a queue of its adjacent vertices in the queue
- Then select the next vertex to be processed, we delete a vertex from the queue and process it
- Lets trace the graph in fig.



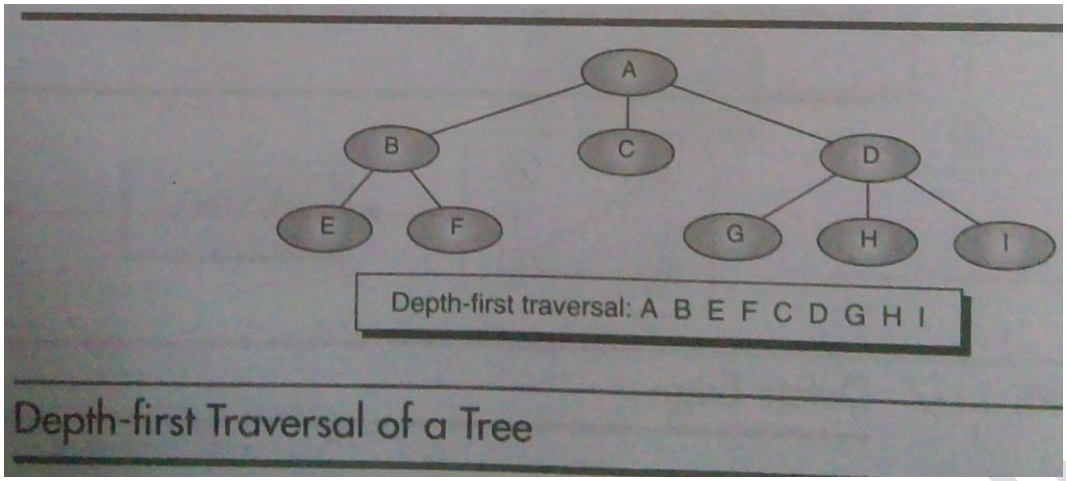
Breadth first traversal of graph

1. We begin by enqueing vertex A in the queue
2. We then loop, dequeing the queue and processing the vertex from the front of the queue. After processing the vertex, we places of its adjacent vertices into the queue. Thus at step 2 in fig (b), we dequeue vertex X, process it, and then place vertices G and H in the queue. We are then ready for step 3, in which we process vertex G
3. when the queue is empty, the traversal is complete

In the BFT, all adjacent vertices are processed before processing the descendents of a vertex

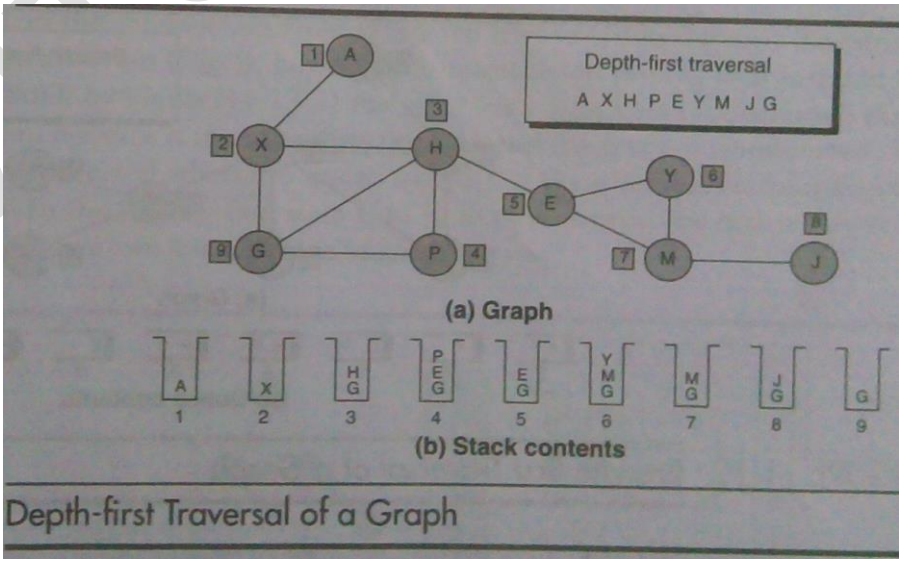
5.13. Depth first Search

- In the depth first traversal, we process all a vertex's descendents before we move to an adjacent vertex
- This is easy when the graph is a tree
- In fig. we show the tree preorder traversal processing sequence



Depth-first Traversal of a Tree

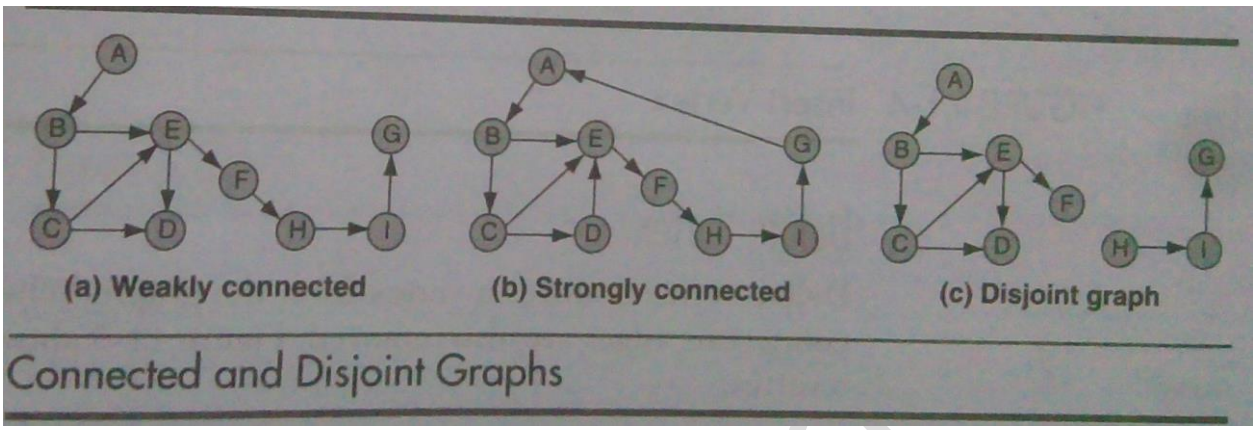
- The depth first traversal of a graph starts by processing the first vertex of the graph
- After processing the first vertex, we select any vertex adjacent to the first vertex and process it
- We process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries
- The logic requires a stack (recursion) to complete the traversal
- The traversal processes adjacent vertices in descending, or last – in – first out (LIFO) order
- Trace a DFT through a graph in fig.
- The number in the box next to a vertex indicates the processing order
- The stacks below the graph show the stack contents as we work our way down the graph and then as we back out
  1. We begin by pushing the first vertex A, into the stack
  2. We then loop, pop the stack, and, after processing the vertex, push all of the adjacent vertices into the stack, process it and then push G and into the stack, giving the stack contents for step 3 as shown in fig (B) – HG
  3. When the stack is empty the traversal is complete



Depth-first Traversal of a Graph

5.14. Connected components:

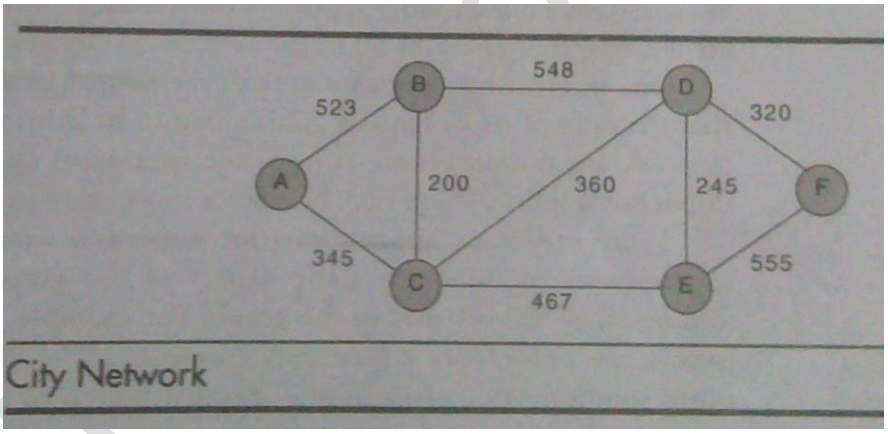
- A directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph
- A digraph is **weakly connected** if atleast two vertices are not connected
- A graph shows is a **disjoint graph** if it is not connected
- Fig. shows



Connected and Disjoint Graphs

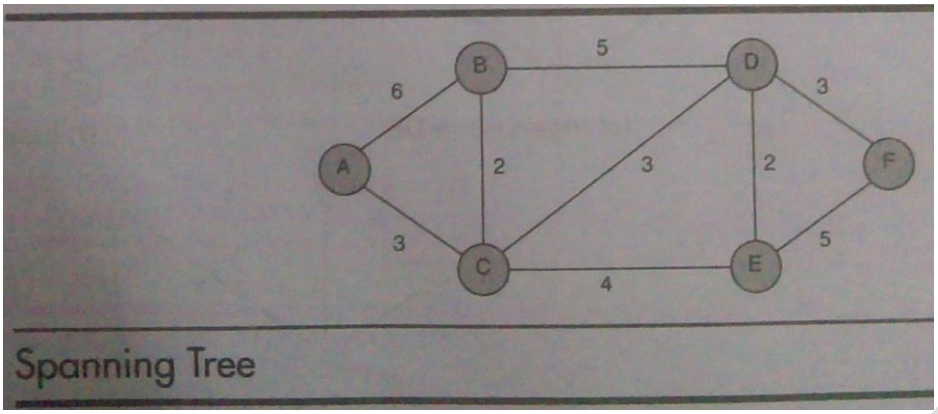
5.15. Spanning Trees:

Network: A Network is a graph whose lines are weighted spanning tree



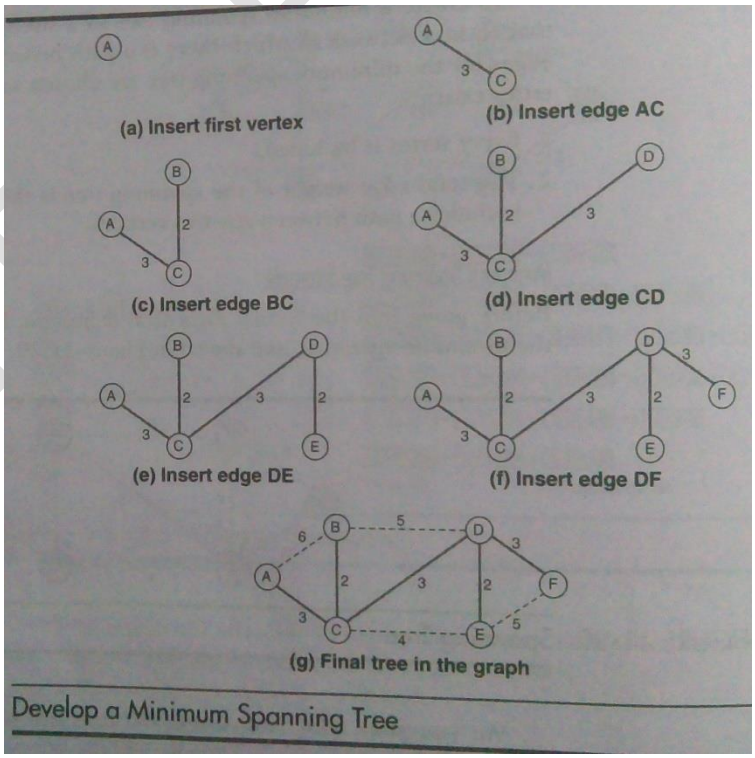
City Network

- We can derive one or more spanning tree from a connected network
- A spanning tree is a tree that contains all the vertices in the graph
- One interesting algorithm the minimum spanning tree of a network such that the sum of its weights is guaranteed to be minimal
- If the weights in the network are unique, there is only one minimum spanning tree
- If there are duplicate weights, there are may be one or more minimum spanning trees
- There are many applications for minimum spanning trees, all with the requirement to minimize some aspect of the graph



- For ex, given a network of computers, we can create a tree that connects all of the computers
- The minimum spanning tree gives us the shortest length of cable that can be used to connect all computers ensuring that there is a path between any two computers
- A spanning tree contains all of the vertices in a graph. A minimum spanning tree is a spanning tree in which total weight of the lines is guaranteed to be the minimum of all possible trees in the graph
- To create a minimum spanning tree in a strongly connected network that is, in a network in which there is a path between any two vertices – the edges for the minimum spanning tree are chosen so that the following properties exist:
  1. Every vertex is included
  2. The total edge weight of the spanning tree is the minimum possible that includes a path between any two vertices

Minimum spanning tree example is in Fig.



5.16. Searching and Sorting:

Searching:

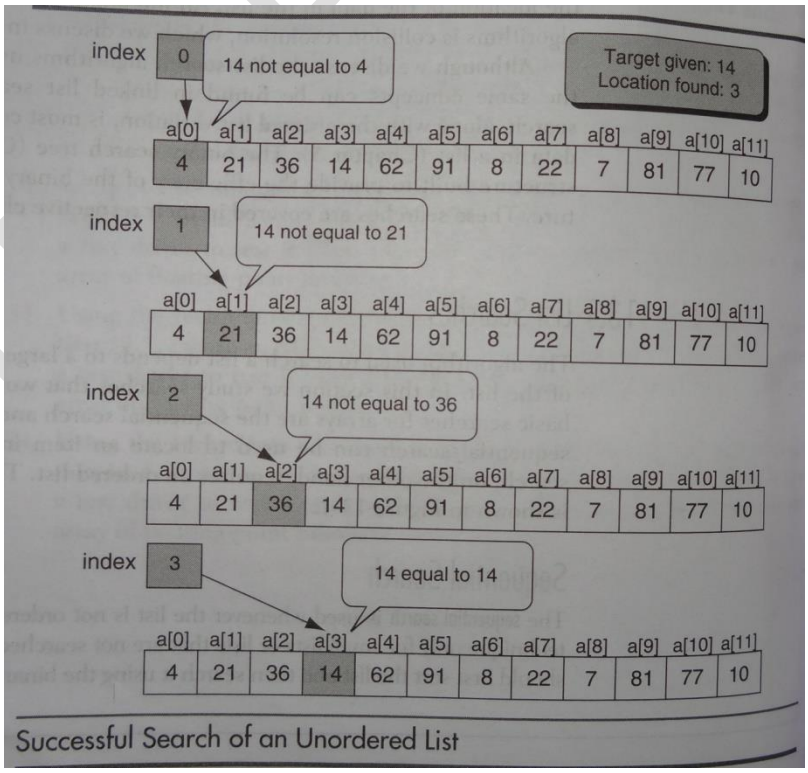
- One of the most common and time – consuming operation in computer science is searching, the process used to find the location of a target among a list of objects
- There are two basic search algorithms:
    - The sequential search (including three variants)
    - The binary search

Sorting:

- Sorting is one of the most common data processing applications
- Data may be sorted in either ascending or descending sequence
- If the order of the sort is not specified it is assumed to be ascending order

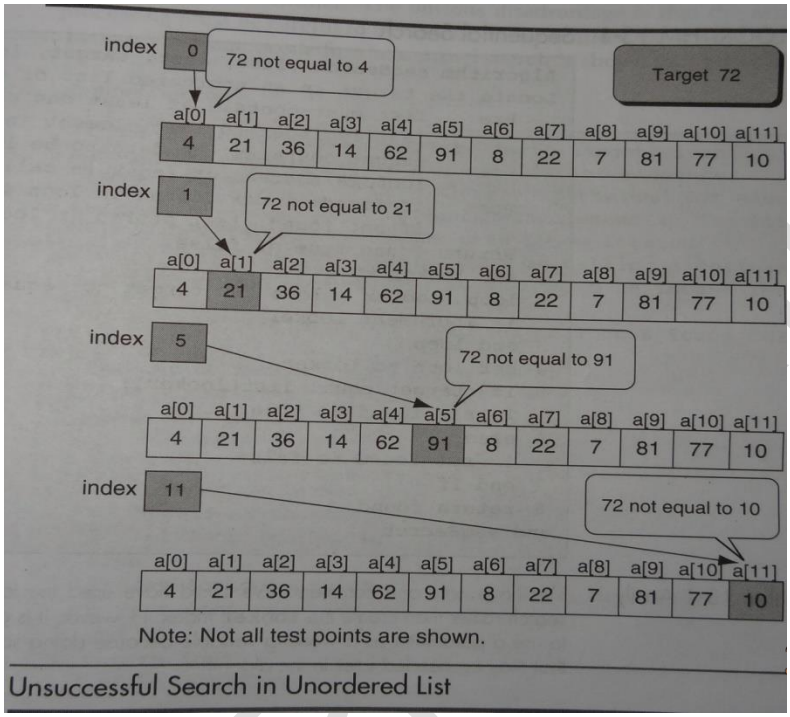
5.17. Sequential Search:

- Sequential search start searching for the target from the beginning of the list and continues searching one by one sequentially in the list until we find the target or we are sure that it is not in the list
- It is used whenever list is **unordered**
- We can apply this technique for small lists or lists that are not searched often
- In other cases we need to sort the list and then apply binary search
- It gives two possibilities:
    - Either we find it or
    - We reach the end of the list
- In fig we trace the steps to find the value 14



Successful Search of an Unordered List

- We first check the data at index 0, then 1 and then 2 before finding 14 on the fourth element (index 3)
- But what if the target were not in the list
- In that case we would have to examine each element until we reach the end of the list
- Fig. traces the search for target 72
- At the end of the list, we discover that the target does not exist



Unsuccessful Search in Unordered List

Sequential search Algorithm

- It needs to tell the calling algorithm two things:
    1. Did it find the data it was looking for? And,
    2. If it did, at what index are the target data found
- To answer the above questions we require four parameters.
    1. The list we are searching
    2. An index to the last element in the list
    3. The target, and
    4. The address where the found element's index location is to be stored
- To tell calling algorithm whether data found, we return a Boolean –true if found it or false if we didn't find it

Algorithm seqsearch(list, last, target, locn)
Locate the target in an unordered list of elements
Pre list must contain at least one element
last is index to last element in the list
target contains the data to be located
locn is address of index in calling algorithm
Post if found index stored in locn & found true
if not found: last stored in locn & found false
Return found true or false

1. Set looker or index 0
2. Loop(looker<last AND target not equal list[looker])
    1. increment looker
3. end loop
4. set locn to looker
5. if(target equal list[looker])
    1. set found to true
6. else
    1. set found to false
7. end if
8. return found

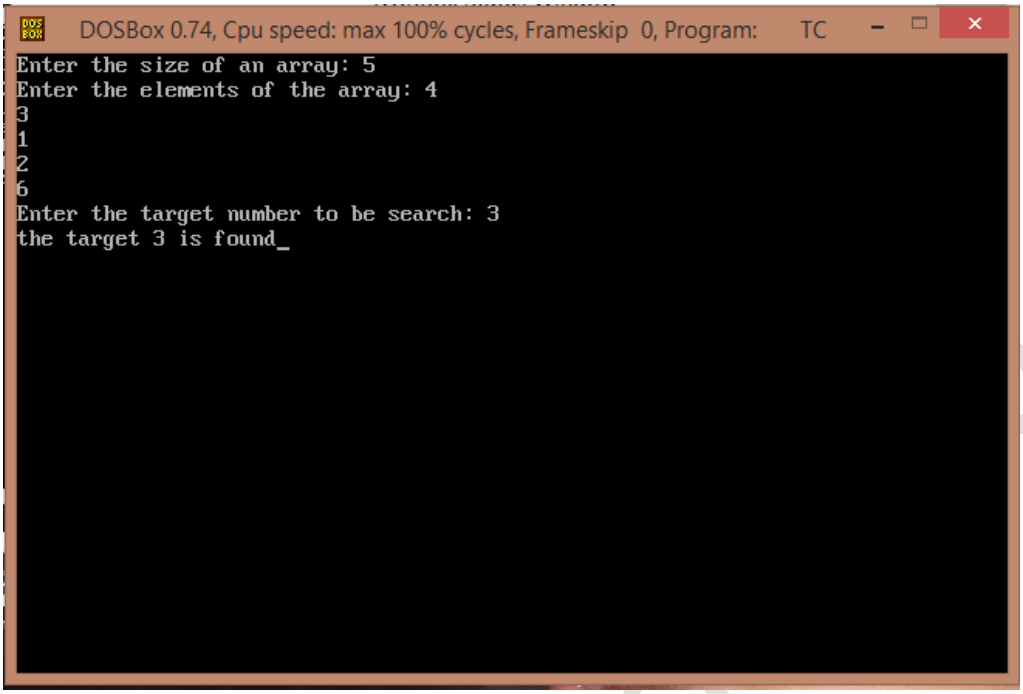
end seqsearch

Sequential or linear search C implementation

```
#include<stdio.h>
#include<conio.h>
int linsearch(int list[],int l, int tar)
{
  int looker=0;
  while(looker<l && tar!=list[looker])
        looker++;
  if(tar==list[looker])
        return 1;
  else
        return 0;
}
void main()
{
   int a[20],i,n,tar;
   clrscr();
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements of the array: " );
   for(i=0;i<n;i++)
   {
         scanf("%d",&a[i]);
   }
   printf("Enter the target number to be search: ");
   scanf("%d",&tar);
   linsearchnrec(a,n-1,tar);
   if(linsearch(a,n-1,tar))
   printf("the target %d is found",tar,n+1);
   else
```

```
    printf("the target %d is not found",tar);
    getch();
}
```

Output:

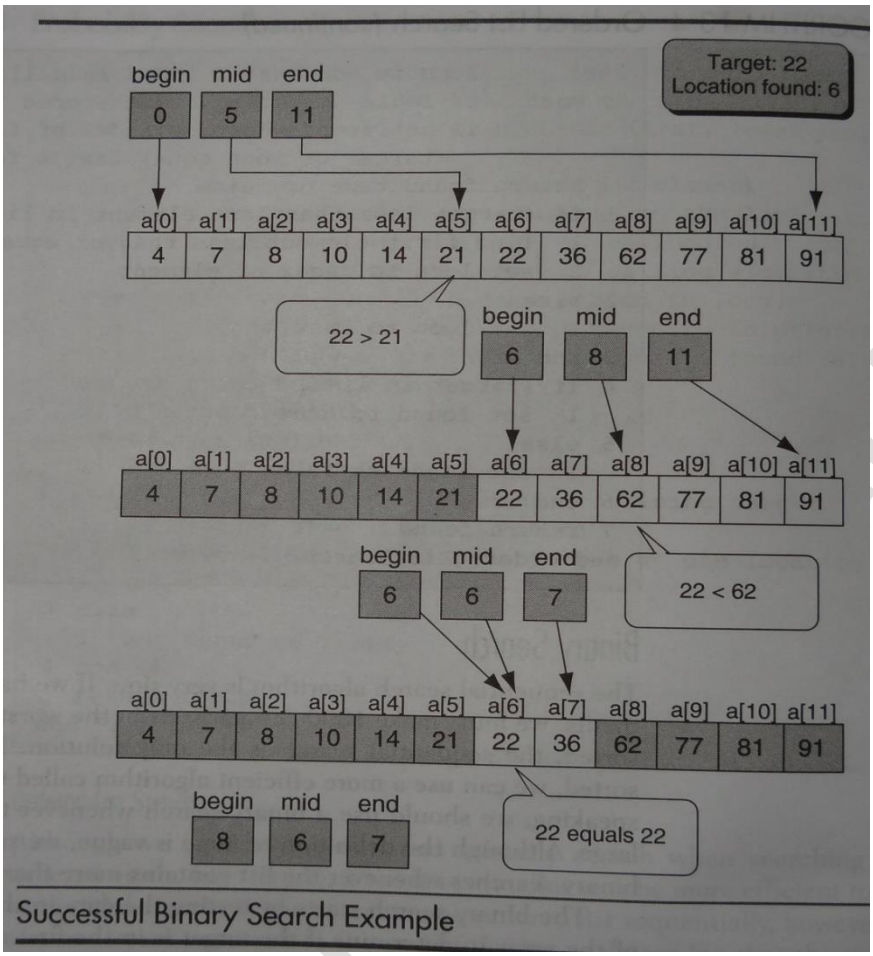


5.18. Binary Search:

- The sequential search algorithm is very slow
- If we have an array of 1000 element, we must make 1000 comparisons in the worst case
- If array is not sorted sequential search is the only the solution
- If the array is sorted the more efficient algorithm we can use is binary search
- Suggestion is use binary search when we have more than 16 element in list
- The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list
- If it is in the first half, we do not need to check the second half
- If it is in the second half, we do not need to check the first half
- In other words, we eliminate half the list from further consideration with just one comparison
- We repeat this process, eliminating half of the remaining list with each test, until we find the target or determine that it is not in the list
- To find the middle of the list, we need three variable: one to identify the middle, one to identify the beginning, one to identify the end of the list.
- We analyze two cases:
  - The target is in the list
  - The target is not in the list

Target found

- Fig. traces the binary search for a target of 22 in a sorted array



- We call our three indexes begin, mid, and end
- Given begin as 0 and end as 11, we can calculate mid as follows:

mid=(begin+end)/2 =(0+11)/2=5

- At index location 5, we discover that the target is greater than the list value (22>21)
- Therefore we eliminate the array locations 0 through 5
- To narrow our search, we set mid + 1 to begin and repeat the search
- The next loop continues mid with the new value for begin(6)and determines that the midpoint is now 8

mid=(6+11)/2=17/2=8

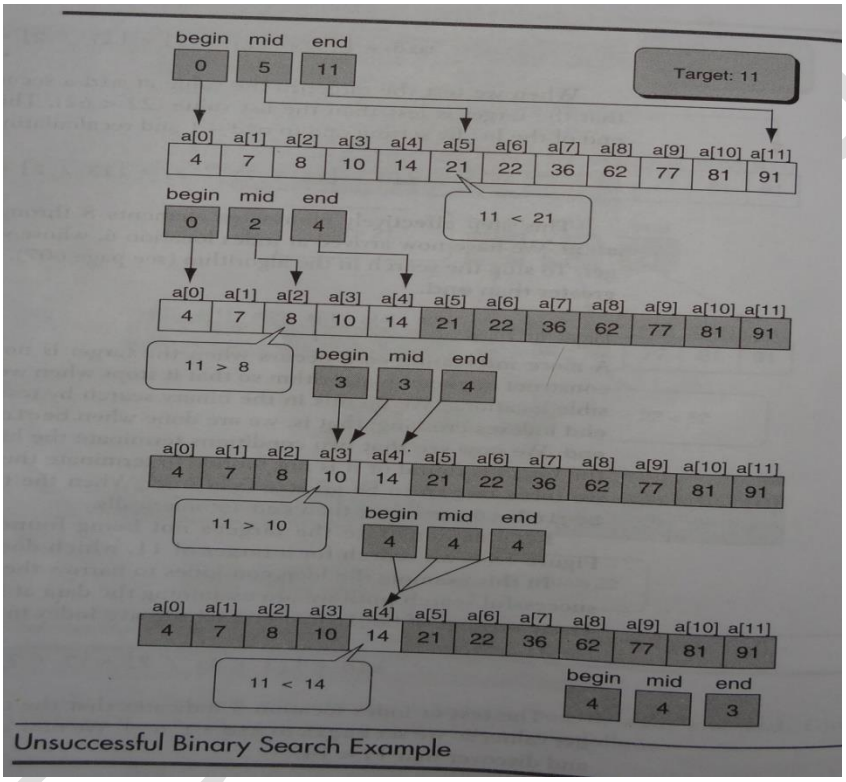
- When we test the target to the value at mid a second time
- The target is less than the list value(22<62)
- This time we adjust the end of the list by setting end to mid – 1 and recalculating mid

mid = (6+7)/2 = 13/2 = 6

- It effectively eliminates 8 through 11 from consideration
- We now arrived at index location 6, whose value matches our target
- We force begin to be greater than end

Target not found

- We must construct our search algorithm so that it stops when we have checked all possible location
- We do this in the binary search by testing for the begin and end indexes crossing; that is, we are done when begin becomes greater then end
- We now see two conditions terminate the binary search algorithm, the target is found or it is not found
- To terminate the loop when it is found, we force begin to be greater than end
- When target is not in the list, begin becomes larger than end automatically
- In fig. we search for a target of 11, which doesn't exist in the array



- In this example the loop continues until we examine the data at index location 3 and 4
- These setting of begin and end set the mid index

mid = (3+4)/2 = (7/2) = 3

- The test at index location 3 indicates that the target is greater than the list value, so we set begin to mid+1, or 4
- We now test the data at location 4 and discover 11<14

mid = (4+4)/2 = 8/2 = 4

- At this point the target should be between two adjacent values, in other words, it is not in the list
- End is set to mid-1 which makes begin greater than end, the signal that the value we are looking for is not in the list

Binary search Algorithm

Algorithm binarysearch(list, last, target)

Search an ordered list using binary search

Pre list is ordered, it must have atleast 1 value
last is index to the target element in the list
target is the value of element being searched

Post Found: return index and set true
Not Found: set found to false

Return found true or false

1. Set begin to 0
2. Set end to last
3. Loop(begin<=end)
    1. Set mid to (begin+end)/2
    2. if(target>list[mid])
    look in upper half
        1. set begin to (mid+1)
    3. else if(target<list[mid])
    look in lower half
        1. set end to mid-1
    4. else
    found: force exit
        1. set begin to (end + 1)
    5. end if
4. end loop
5. if(target equal list[mid])
    1. set found to true
6. else
    1. set found to false
7. end if
8. return found

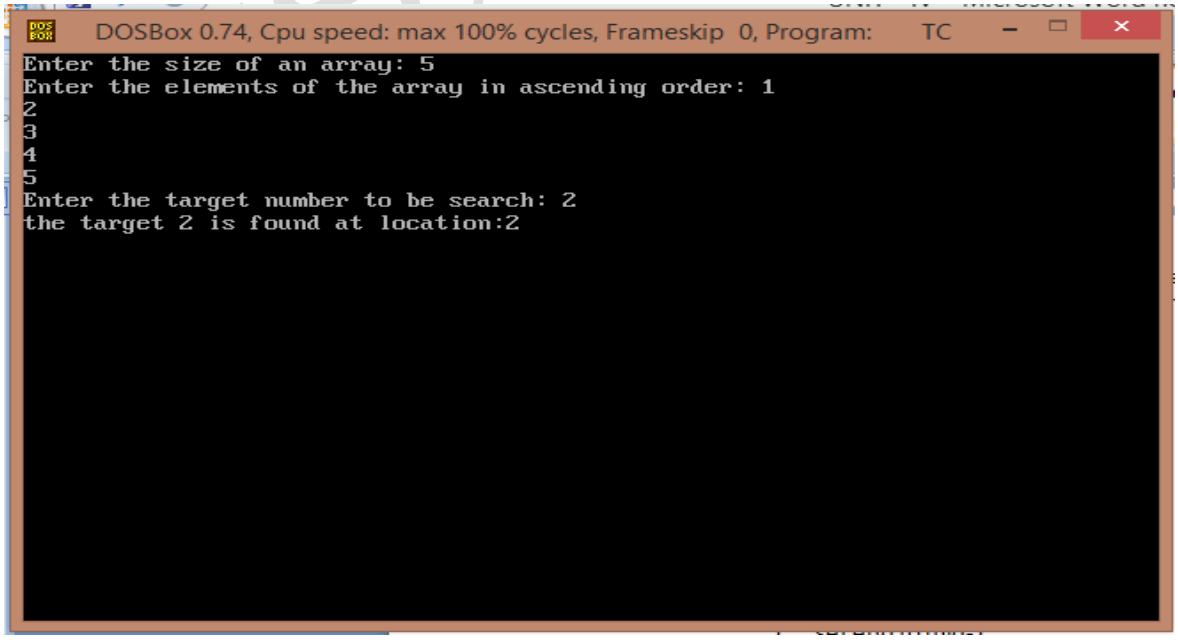
end binary search

Binary search C Implementation

```
#include<stdio.h>
#include<conio.h>
int m;
int binsearch(int list[],int b,int e, int tar)
{
 while(b<=e)
 {
        m=(b+e)/2;
        if(tar>list[m])
                b=m+1;
        else if(tar<list[m])
                e=m-1;
```

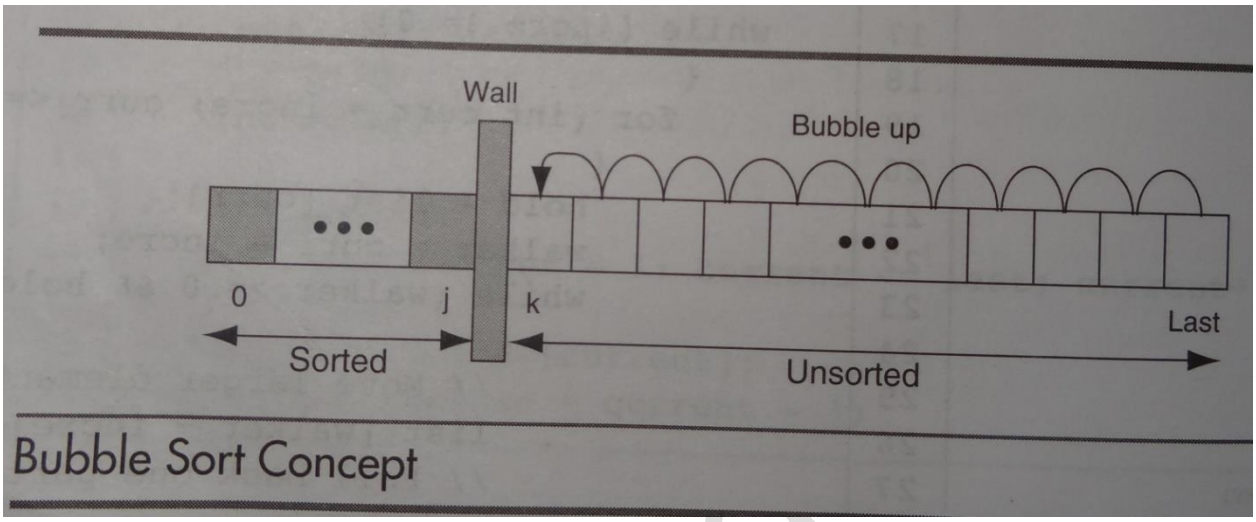
```
        else
                b=e+1;
  }
  if(tar==list[m])
  return 1;
  else
  return 0;
}
void main()
{
   int a[20],i,n,tar;
   clrscr();
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements of the array in ascending order: " );
   for(i=0;i<n;i++)
   {
        scanf("%d",&a[i]);
   }
   printf("Enter the target number to be search: ");
   scanf("%d",&tar);
   if(binsearch(a,0,n-1,tar))
   printf("the target %d is found at location:%d",tar,m+1);
   else
   printf("the target %d is not found",tar);
   getch();
}
```

Output:



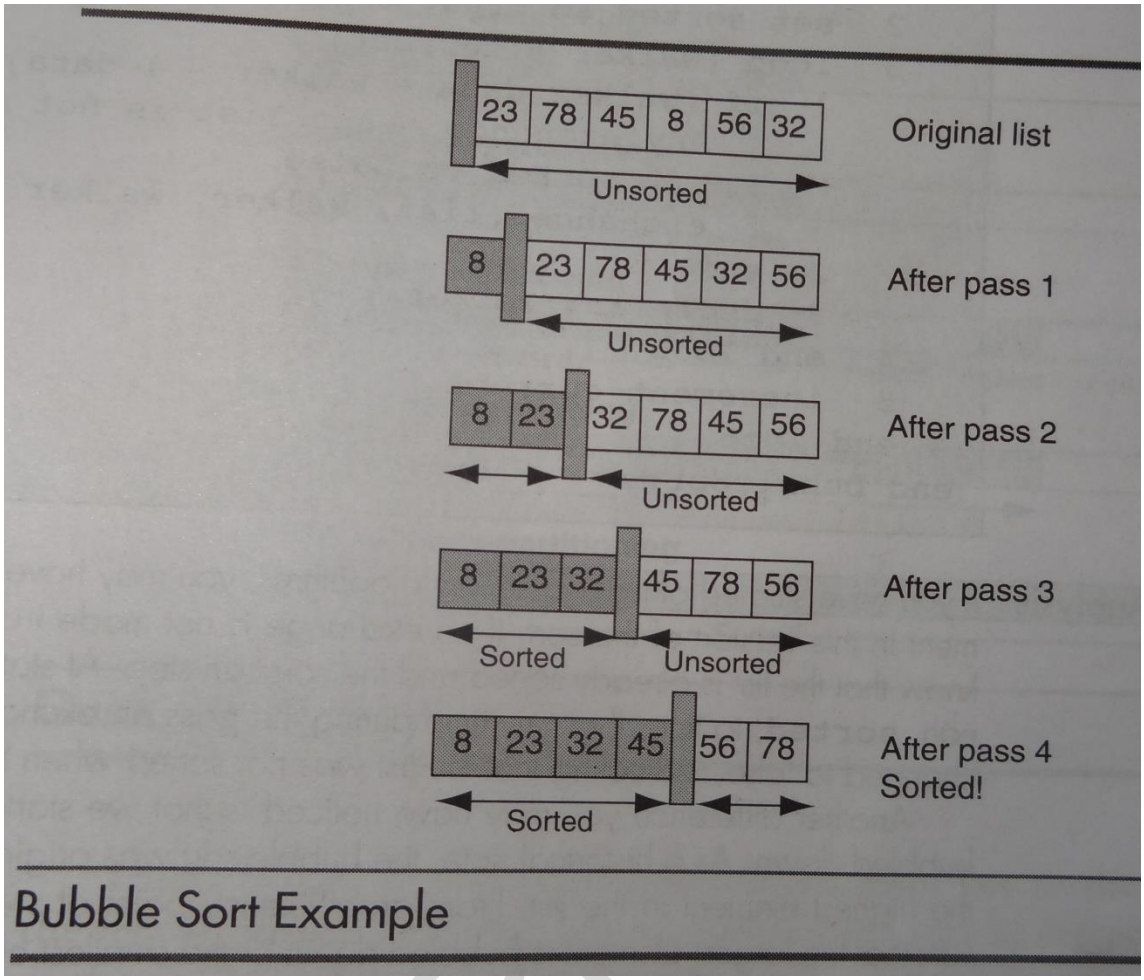
5.19. Exchange (Bubble) Sort:

- In this, the list at any moment is divided into two sublists : sorted and unsorted
- The smallest element is bubbled from the unsorted sub list and moved to the sorted sublist
- After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones(fig.)



Bubble Sort Concept

- Each time an element moves from the unsorted to sorted sublist, one sort pass is completed
- Given a list of n elements, the bubble sort requires up to n-1 passes to sort the data
- Fig. below shows how the wall moves one element in each pass
- Look at the first pass, we start with 32 and compare it with 56
- Because 32 is less than 56, we exchange the two and step down one element
- We then compare 32 and 8
- Because 32 is not less than 8, we do not exchange these elements
- We step down in element and compares 45 and 8
- They are out of sequence, so we exchange them and step down again
- Because we moved 8 down, it is now compared with 78, and these two elements are exchanged
- Finally, 8 is compared with 23 and exchanged
- This series exchanges places 8 in the first location and wall is moved up one position



Bubble sort algorithm

- The bubble sort is quite simple
- In each pass through the data, the smallest element is bubbled to the beginning of the unsorted segment of the array

Algorithm bubblesort(list, last)
Sort an array using bubble sort
Adjacent elements are compared and exchanged until list is completely ordered

Pre list must contain atleast one item
last contains index to last element in the list
Post list has been rearranged in sequence low to high

1. Set current to 0
2. Set sorted to false
3. Loop(current <= last AND sorted false)

Each iteration is one sort pass

1. Set walker to last
2. Set sorted to true
3. Loop(walker>current)
    1. if(walkerdata<walker-1data)

any exchange means list is not sorted
1. set sorted to false
2. exchange(list, walker, walker – 1)

2. end if
3. decrement walker

4. end loop
5. increment current

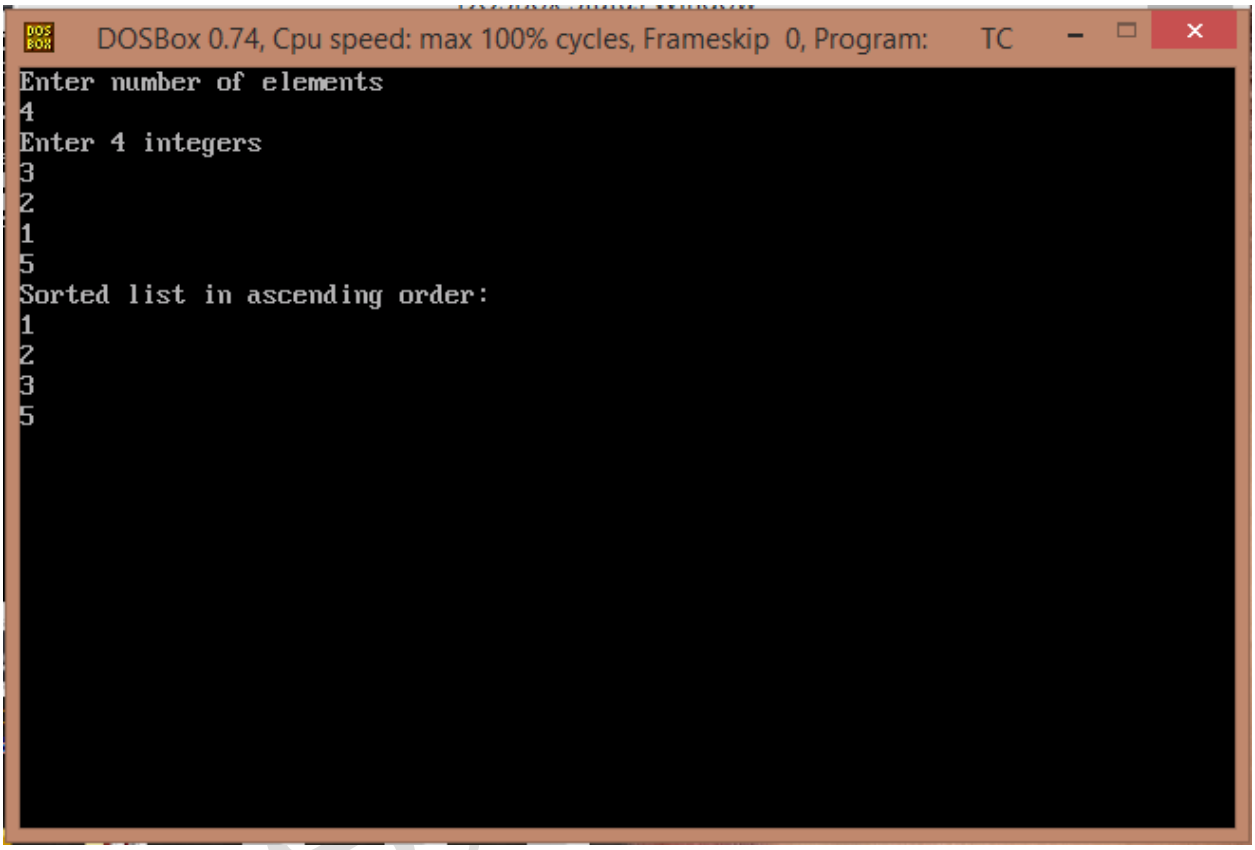
4. end loop

Bubble sort implementation in C

```
#include <stdio.h>
void exchange(int list[],int w1,int w2);
void bubblesort(int list[],int last)
{
 int c=0,s=0,walk;
 while(c<=last&&s==0)
 {
 walk=last;
 s=1;
 while(walk>c)
 {
  if(list[walk]<list[walk-1])
  {
   s=0;
   exchange(list,walk,walk-1);
  }
  walk--;
 }
 c++;
 }
}
void exchange(int list[],int w1,int w2)
{
 int swap=0;
 swap=list[w1];
 list[w1]=list[w2];
 list[w2]=swap;
}
void main()
{
  int arr[100], n,i;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
```

```
  for (i = 0; i < n; i++)
  scanf("%d", &arr[i]);
  bubblesort(arr,n-1);
  printf("Sorted list in ascending order:\n");
  for ( i = 0 ; i < n ; i++ )
    printf("%d\n", arr[i]);
}
```

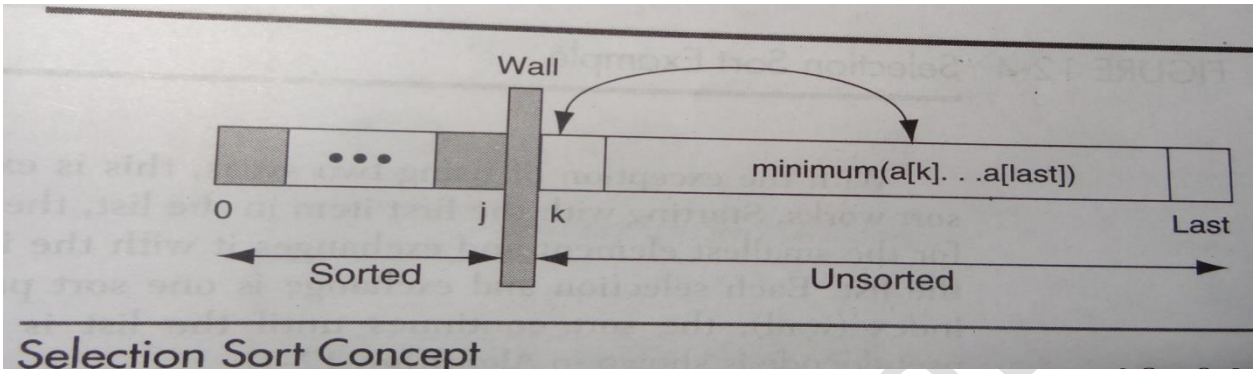
Output:



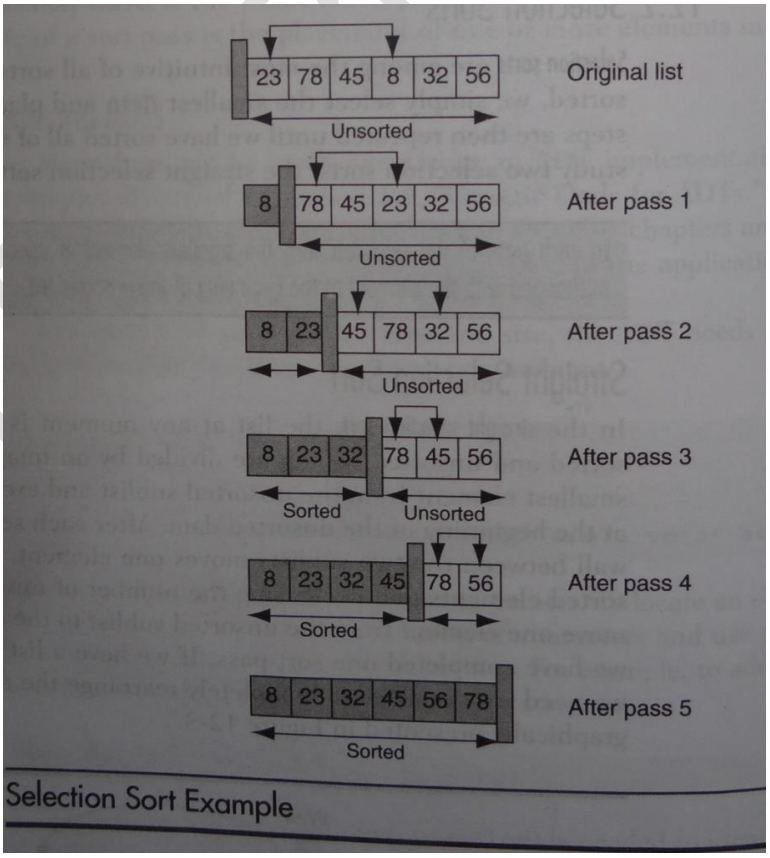
5.20. Selection sort:

- Selection sorts are among the most intuitive of all sorts
- Given a list of data to be sorted, we simply select the **smallest item** and place it in a **sorted list**
- These steps are then repeated until we have sorted all of the data
- In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list
- The list at any moment is divided into two sublists, sorted and unsorted, which are divided by imaginary wall
- We select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data
- After each selection and exchange, the wall between the two sublists moves one element, increasing the number of sorted elements and decreasing the number of unsorted ones

- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one sort pass
- If we have a list of n elements, therefore we need n-1 passes to completely rearrange the data
- The selection sort is presented in fig.



- Fig traces our set of six integers as we sort them
- It shows how the wall between the sorted and unsorted sublists moves in each pass
- You see that the array is sorted after five passes, one less than the number of elements in the array
- Thus, if we use a loop to control the sorting, our loop has one less iteration than the number of elements in the array



Selection sort algorithm

- Starting with the first item in the list, the algorithm scans the list for the smallest element and exchanges it with the item at the beginning of the list
- Each selection and exchange is one sort pass
- After advancing the index(wall), the sort continues until the list is completely sorted
- The algorithm is

Algorithm selectionsort(list, last)
Sorts list array by selecting smallest element in unsorted portion of array and exchanging it with element at beginning of the unsorted list

Pre list must contain atleast one item
last contains index to last element in the list
Post list has been rearranges smallest to largest

1. Set current to 0
2. Loop(until last element sorted)
    1. Set smallest to current
    2. Set walker to current – 1
    3. Loop(walker <= last)
        1. if(walker key < smallest key)
            1. set smallest to walker
        2. increment walker
    4. end loop

    smallest selected: exchange with current element
    5. exchange(current, smallest)
    6. increment count
3. end loop

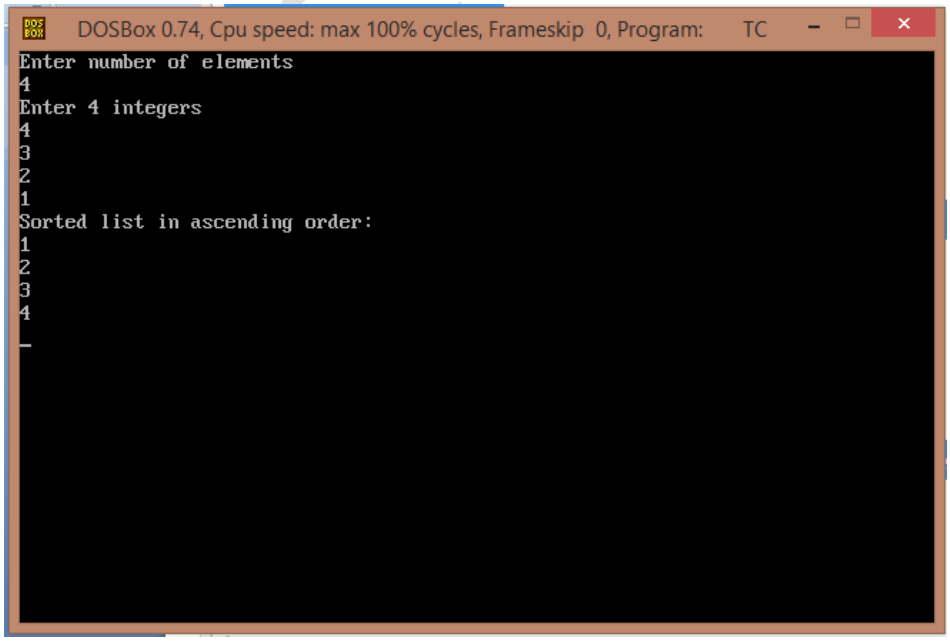
end selection sort

Selection sort implementation in C

```
#include <stdio.h>
void exchange(int list[],int w1,int w2);
void selectionsort(int list[],int last)
{
 int c=0,small,walk;
 while(c<=last)
 {
 small=c;
 walk=c+1;
 while(walk<=last)
 {
  if(list[walk]<list[small])
  {
   small=walk;
  }
```

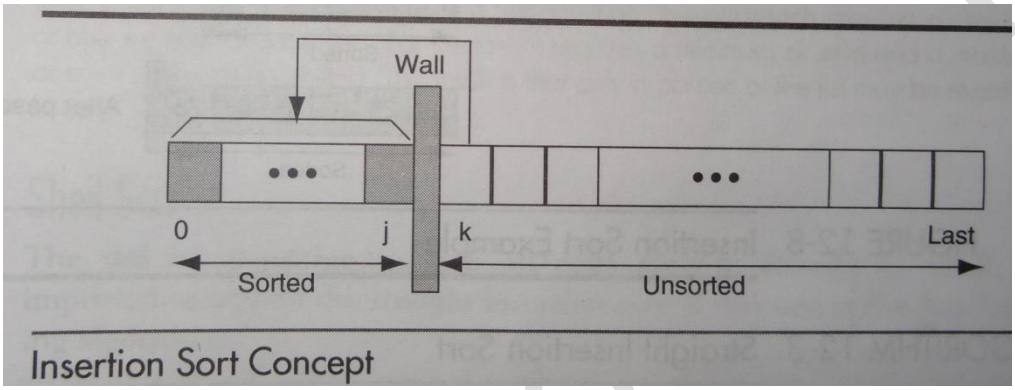
```
   walk++;
  }
  exchange(list,c,small);
  c++;
 }
}
void exchange(int list[],int w1,int w2)
{
 int swap=0;
 swap=list[w1];
 list[w1]=list[w2];
 list[w2]=swap;
}
void main()
{
  int arr[100], n,i;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
  scanf("%d", &arr[i]);
  selectionsort(arr,n-1);
  printf("Sorted list in ascending order:\n");
  for ( i = 0 ; i < n ; i++ )
     printf("%d\n", arr[i]);
}
```

Output:



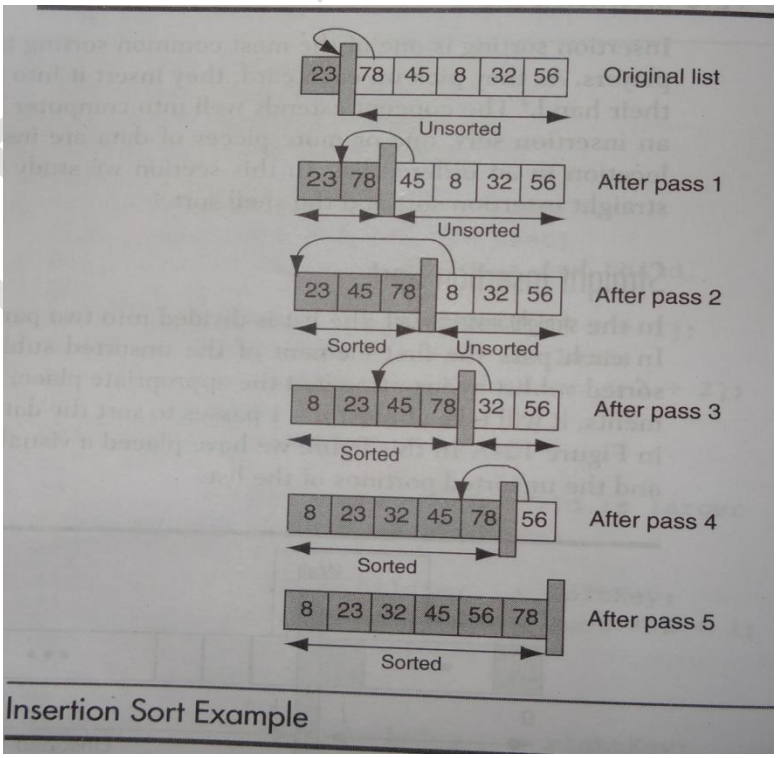
5.21. Insertion Sort:

- It is one of the most common sorting techniques used by card player
- As they pick up each card, they insert it into the proper sequence in their hand
- In each pass of sort, one or more pieces of data are inserted into their correct location in an ordered list
- As usual the list is divided into two parts: sorted and unsorted
- In each pass the first element of the unsorted sublist is transferred to the sorted sublist is transferred to the sorted sublist by inserting it at the appropriate place
- If we have a list of n elements, it will take at most n-1 passes to sort the data
- This concept is shown in fig we have placed a visual wall between the sorted and the unsorted partitions of the list



Example: fig. traces the insertion sort through a list of six numbers

- Sorting these data requires five sort passes
- Each pass moves the wall one element from the unsorted sublist and inserted into the sorted sublist



Insertion sort algorithm

- Each execution of the outer loop inserts the first element from the unsorted list into the sorted list
- The inner loop steps through the sorted list, starting at the high end, looking for the correct insertion location
- The pseudo code is shown in algorithm

Algorithm insertionsort(list, last)

Sort list array using insertion sort. The array is divided into sorted and unsorted lists. With each pass, the first element in the unsorted list is inserted into the sorted list

Pre list must contain atleast on element
last is an index to last element in the list

Post list has been rearranged

1. Set current to 1
2. Loop(until last element sorted)
    1. Move current element to hold
    2. Set walker to current – 1
    3. Loop(walker >=0 AND hold key < walker key)
        1. Move walker element to right one element
        2. Decrement walker
    4. End loop
    5. Move hold to walker + 1 element
    6. Increment current
3. End loop

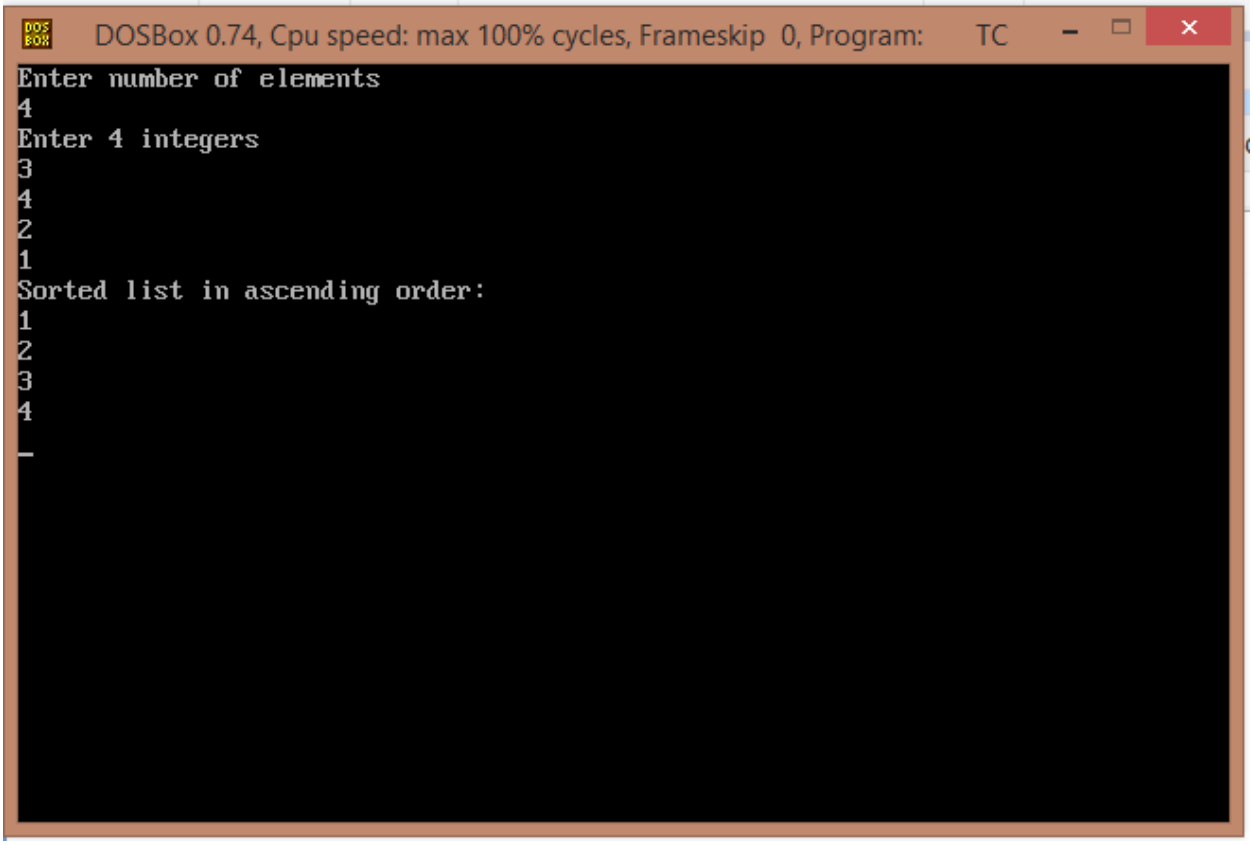
End insertionsort

Insertion sort implementation in C

```
#include <stdio.h>
#include <conio.h>
void insertionsort(int list[],int last)
{
 int c=1,hold,walk;
 while(c<=last)
 {
 hold=list[c];
 walk=c-1;
 while(walk>=0&&hold<list[walk])
 {
  list[walk+1]=list[walk];
  walk--;
 }
 list[walk+1]=hold;
c++;
}
```

```
}
void main()
{
  int arr[100], n,i;
  clrscr();
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
  scanf("%d", &arr[i]);
  insertionsort(arr,n-1);
  printf("Sorted list in ascending order:\n");
  for ( i = 0 ; i < n ; i++ )
     printf("%d\n", arr[i]);
}
```

Output



i) Question and Answers: 2 Marks

1. Differentiate between full and complete binary tree. (Dec – 2019)

Ans: A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) **is** a **tree** in which every node other than the leaves has two children. A **complete binary tree is** a **binary tree** in which every level, except possibly the last, **is** completely filled, and all nodes are as far left as possible.

2. In what way a linear search differs from binary search? (Dec – 2019, Nov/Dec - 2019)

Ans: **Linear search** is a **search** that finds an element in the list by **searching** the element sequentially until the element is found in the list. On the other hand, a **binary search** is a **search** that finds the middle element in the list recursively until the middle element is matched with a searched element.

3. What is meant by searching? (Nov/Dec – 2019)

Ans: **Searching** is the process of **finding** a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of **finding** a particular item in a collection of items.

4. Define height, level and degree of a tree. (Nov/Dec – 2019)

Ans: The **height** of the tree is the level of the leaf in the longest path from the root + 1.

- The **level** of a node is its distance from the root
- Degree is the number of branches associated with a node is the **degree** of the node. The sum of the indegree and outdegree branches is the degree of the node.

5. Define Sorting? (Nov/ Dec – 2019, Nov/ Dec – 2018)

Ans: Sorting is arranging the elements in a particular order, it may be either ascending or descending order.

6. Define graph and its key terms. (Jun/Jul – 2019)

Ans: A **graph** is a collection of nodes, called vertices, and a collection of segments, called lines, connecting pairs of vertices.
Digraph, Undirected graph, Arcs, edges, path, Cycle, Loop.

7. What is the main idea behind insertion sort? (Jun/Jul – 2019)

Ans: It is one of the most common sorting techniques used by card player. As they pick up each card, they insert it into the proper sequence in their hand. In each pass of sort, one or more pieces of data are inserted into their correct location in an ordered list. In each pass the first element of the unsorted sublist is transferred to the sorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.

8. Define Spanning tree. (June/July – 2019)

Ans: A spanning tree is a tree that contains all the vertices in the graph. A Network is a graph whose lines are weighted spanning tree.

9. State any two properties of a binary tree. (Jun/Jul – 2019)

Ans: **Height of binary trees**

Given that we need to store N node in a binary tree, the maximum height H_{max} is

$$H_{max}=N$$

The minimum height of the tree, H_{min} is determined by the following formula:

$$H_{min} = [\log_2 N]+1$$

Maximum Nodes

The height of the binary tree H, the maximum number of nodes in the tree is given as

$$N_{max} = 2^H - 1$$

Given a height of the binary tree, H, the minimum number of the nodes in the tree are given as

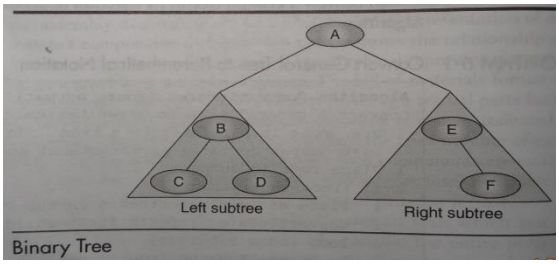
$$N_{min} = H$$

10. How can we say that binary search is better than linear search? (Jun/Jul – 2019)

Ans: The **binary search** algorithm is more efficient than the **linear search** algorithm because it takes less time to **search through** the list.

11. Define a binary tree and give an example. (May/Jun – 2019).

Ans: A binary tree is a tree in which no node can have more than two subtrees, the **maximum outdegree** for a node is **two**. In other words, a node can have zero, one or two subtrees.



12. Differentiate sorting by insertion and sorting by selection. (May/Jun – 2019).

Ans: The insertion sort inserts the values in a presorted file to sort a set of values. On the other hand, the selection sort finds the minimum number from the list and sort it in some order.

13. Distinguish between trees and binary trees. (Dec – 2017).

Ans: The main **difference between tree and binary tree** is that **tree** arranges data in a structure similar to a **tree** in a hierarchical manner while a **binary tree** is a type of **tree** in which a parent node can have a maximum of two child nodes.

14. Give brief description about the technique behind the linear search. (Dec – 2017)

Ans: Linear search start searching for the target from the beginning of the list and continues searching one by one sequentially in the list until we find the target or we are sure that it is not in the list

15. Enumerate the steps to delete an edge from an undirected graph. (Nov/ Dec – 2017)

Ans: It removes one arc from the adjacency list.

- To identify an arc, we need to vertices.
- The vertices are identified by their key.
- The algorithm therefore searches the vertex list for the start vertex and then searches its adjacency list for the destination vertex.
- After locating and deleting the arc, the degree in the form and to vertices is adjusted and the memory recycles.

ii) MCQs

1. _____ search start searching for the target from the beginning of the list and continues searching one by one sequentially.

a. Linear b. Sequential c. **Both a & b** d. None

2. The _____ search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list.

a. Linear b. **Binary** c. Sequential d. None

3. _____ is used to arrange the elements in a list in either ascending or descending order.

a. Search b. **Sort** c. ordering d. None

4. In _____ algorithm we simply select the smallest item and place it in a sorted list**.**

a. Bubble sort b. Insertion sort c. **Selection sort** d. None

5. _____is the most common sorting techniques used by card player.

a. Bubble sort b. **Insertion sort** c. Selection sort d. None

6. A _____ consists of a finite set of elements called **nodes**, and a finite set of directed lines, called branches, that connect the nodes.

a. Graph b. **Tree** c. List d. Array

7. The number of branches associated with a node is the _____ of the node.

a. Height b. Predecessor c. Successor d. **Degree**

8. A ___ is any node with an outdegree of zero, that is, a node with no successors.

a. Parent b. Child c. Root d. **Leaf**

9. A _____ is a sequence of nodes in which each node is adjacent to the next one.

a. Ancestor b. Descendent c. **Path** d. Siblings

10. The _____ of a node is its distance from the root.

a. Height b. Weight c. **Level** d. Degree

11. A _____ is a tree in which no node can have more than two subtrees, the maximum out degree for a node is two.

a. Graph b. **Binary tree** c. Sub tree d. None

12. In preorder traversal, which traversing order is followed, _____.

a. LNR b. LRN c. **NLR** d. NRL

13. A _____ is a collection of nodes, called vertices, and a collection of segments, called lines, connecting pairs of vertices.

a. Tree b. **Graph** c. Array d. List

14. A _____ is a special case of a cycle in which a single arc begins and ends with the same vertex.

a. **Loop** b. Round c. Circle d. None

15. The _____ of a vertex in a digraph is the number of arcs leaving the vertex.

a. Indegree b. **Outdegree** c. Degree d. Both a & b

iii) Question and Answers: 10 Marks

1. What is meant by tree traversal? List the different traversals used in binary trees. Find the different traversal with an example binary tree.

Ans:

Tree traversal: 2M

A **tree traversal** requires that each node of the tree be processed once and only once in a predetermined sequence.

List different traversals used in binary tree: 2M

1. Depth first traversal:

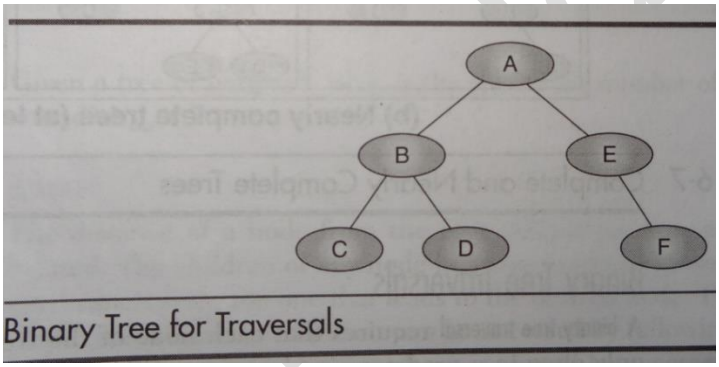
a. Preorder traversal

b. Postorder traversal

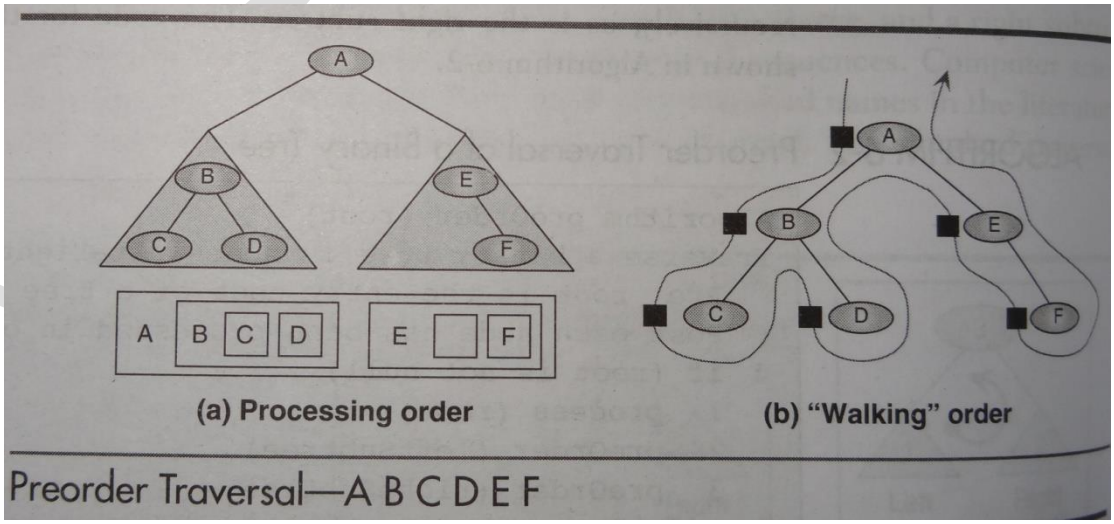
c. Inorder traversal

2. Breadth first traversal

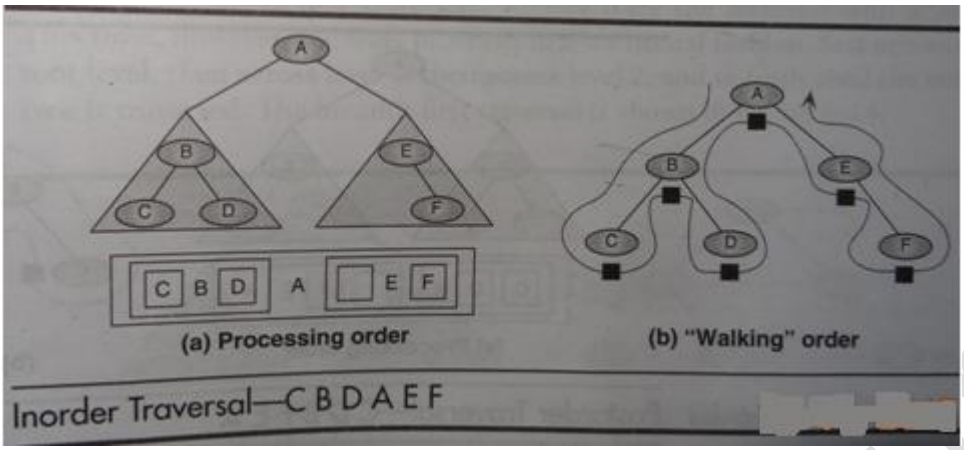
Example binary tree:



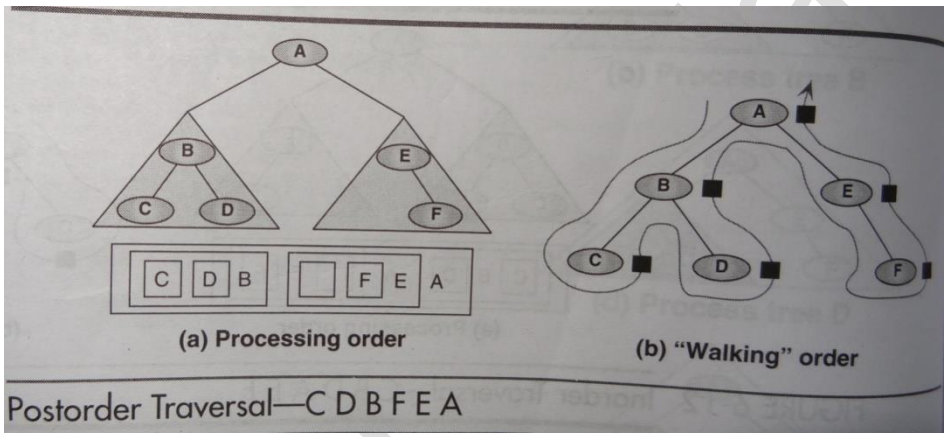
Preorder Traversal: 2M



Inorder Traversal: 2M



Postorder Traversal: 2M



2. Apply binary search to find the element 12.11 from the set {1.1, 2.33, 4.9, 12.11, 13.33, 14.44, 15.55, 16.66}. Show each step clearly. 10M

Ans:

Given set,
{1.1, 2.33, 4.9, 12.11, 13.33, 14.44, 15.55, 16.66}
The element given to search is, 12.11
total elements =8
Calculate middle index,

m=(b+e)/2

b= 0 e=7

m=3

the element present at index 3 is 12.11

compare the search element with middle element,

12.11 == 12.11

As both are equal the element is found at index 3.

3. Write and explain algorithm for binary search. (Jun/July -2019) 10M

Ans:

Binary search Algorithm

Algorithm binarysearch(list, last, target)
Search an ordered list using binary search

Pre list is ordered, it must have atleast 1 value
last is index to the target element in the list
target is the value of element being searched
Post Found: return index and set true
Not Found: set found to false
Return found true or false

1. Set begin to 0
2. Set end to last
3. Loop(begin<=end)
    1. Set mid to (begin+end)/2
    2. if(target>list[mid])
        1. set begin to (mid+1)
    3. else if(target<list[mid])
        1. set end to mid-1
    4. else
        1. set begin to (end + 1)
    5. end if
4. end loop
5. if(target equal list[mid])
    1. set found to true
6. else
    1. set found to false
7. end if
8. return found

end binary search

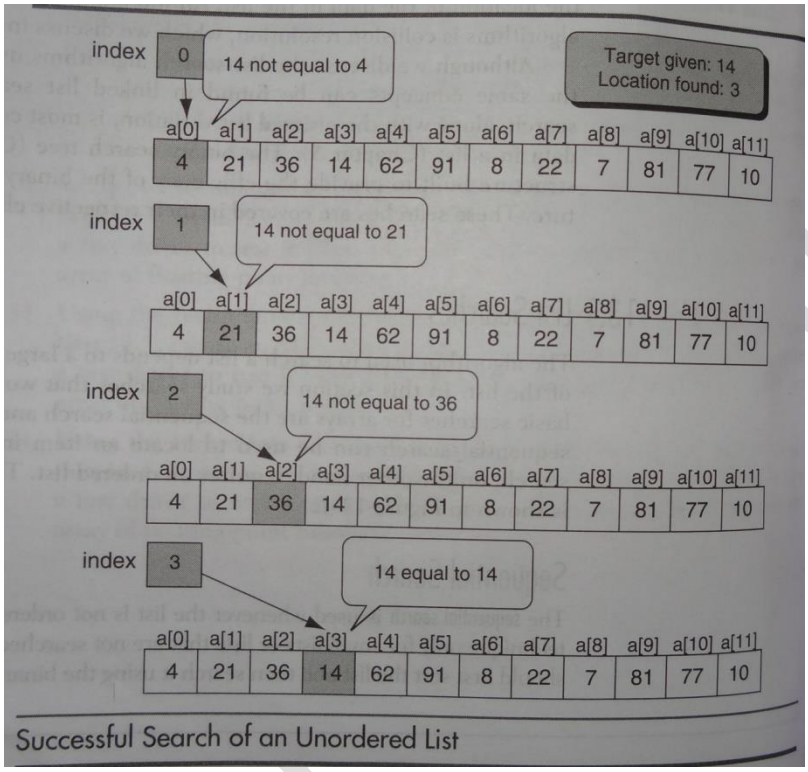
4. Discuss sequential search procedure with example. (Dec – 2018) 10M

Ans:

Sequential or linear search

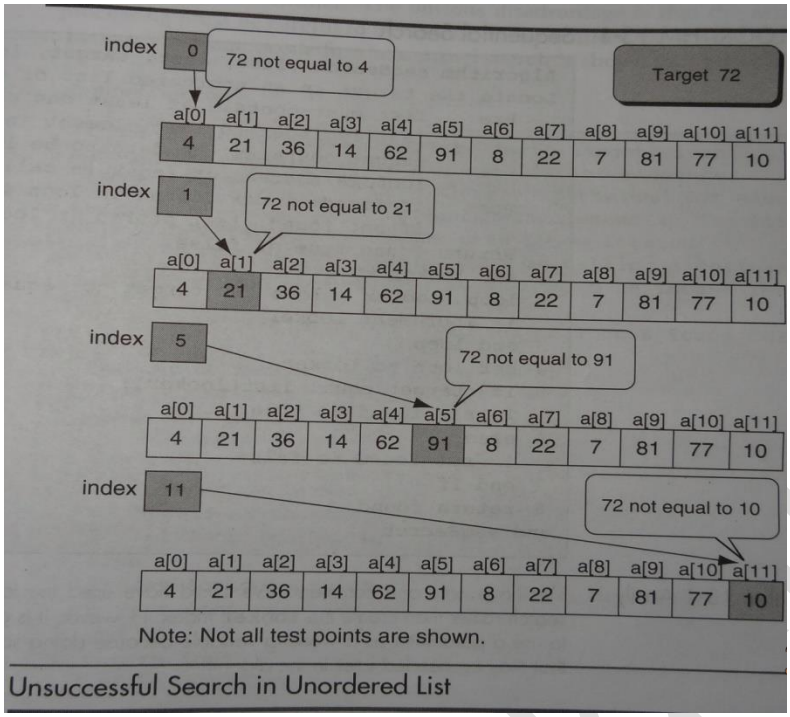
- Sequential search start searching for the target from the beginning of the list and continues searching one by one sequentially in the list until we find the target or we are sure that it is not in the list
- It is used whenever list is **unordered**

- We can apply this technique for small lists or lists that are not searched often
- In other cases we need to sort the list and then apply binary search
- It gives two possibilities:
    - Either we find it or
    - We reach the end of the list
- In fig we trace the steps to find the value 14



Successful Search of an Unordered List

- We first check the data at index 0, then 1 and then 2 before finding 14 on the fourth element (index 3)
- But what if the target were not in the list
- In that case we would have to examine each element until we reach the end of the list
- Fig. traces the search for target 72
- At the end of the list, we discover that the target does not exist



5. An array contains the elements shown below

3 13 7 26 44 23 19 57

Sort the array using bubble sort and show the contents of the array at each step. (Dec – 2018) 10M

Ans:

Given elements,

3 13 7 26 44 23 19 57

Pass 1:

3 7 13 19 26 44 23 57

Pass 2:

3 7 13 19 23 26 44 57

Pass 3:

3 7 13 19 23 26 44 57

Pass 4:

3 7 13 19 23 26 44 57

Pass 5:

3 7 13 19 23 26 44 57

Pass 6:

3 7 13 19 23 26 44 57

Pass 7:

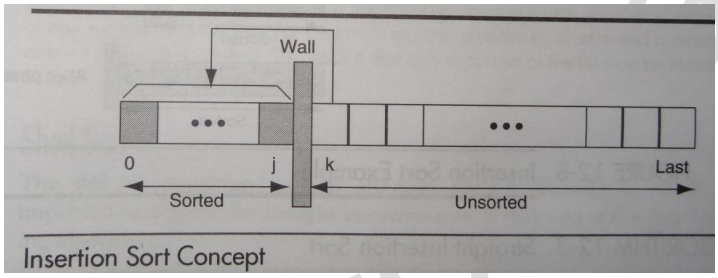
3 7 13 19 23 26 44 57

6. Discuss in detail about insertion sort and algorithm with example.

Ans:

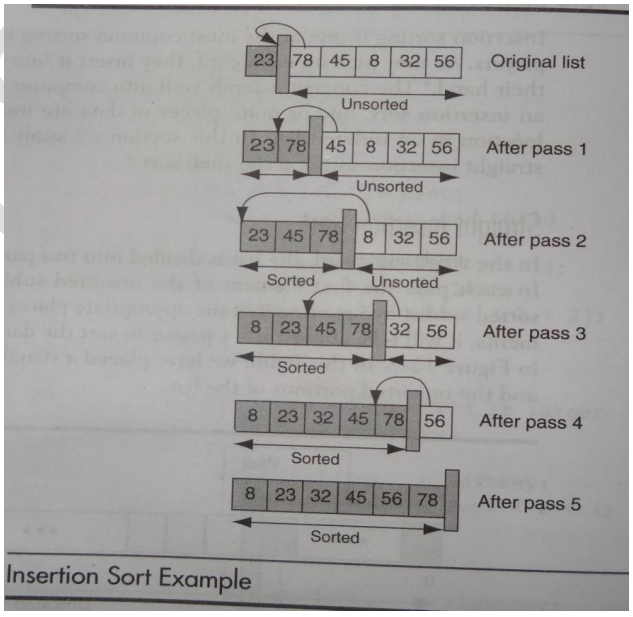
Insertion sort: 5M

- It is one of the most common sorting techniques used by card player
- As they pick up each card, they insert it into the proper sequence in their hand
- In each pass of sort, one or more pieces of data are inserted into their correct location in an ordered list
- As usual the list is divided into two parts: sorted and unsorted
- In each pass the first element of the unsorted sublist is transferred to the sorted sublist is transferred to the sorted sublist by inserting it at the appropriate place
- If we have a list of n elements, it will take at most n-1 passes to sort the data
- This concept is shown in fig we have placed a visual wall between the sorted and the unsorted partitions of the list



Example: fig. traces the insertion sort through a list of six numbers 5M

- Sorting these data requires five sort passes
- Each pass moves the wall one element from the unsorted sublist and inserted into the sorted sublist



Code: 15A05101 **R15**

B.Tech I Year II Semester (R15) Supplementary Examinations December 2019

COMPUTER PROGRAMMING

(Food Technology)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) List the advantages and disadvantages of flowcharts over algorithms.
(b) List out the basic data types and their sizes in C.
(c) Differentiate between while and do-while loop.
(d) List out any four string handling functions in C.
(e) Write the syntax and one example for any two dynamic memory allocation functions in C.
(f) List out the storage classes in C.
(g) Write a recursive function to compute factorial of an integer.
(h) Differentiate between structures and unions.
(i) Write the syntax and one example for any two formatted console I/O statements in C.
(j) Write the syntax of fopen and fclose functions.

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 With a neat diagram, explain the software development method.

OR

3 Explain the arithmetic operators in C. Write a C program to test whether a given number is positive or negative with and without using conditional operator.

UNIT – II

4 Write the syntax of for, while and do-while loops. Write a C program to print the sum of first n natural numbers.

OR

5 Discuss two-dimensional array. Write a C program to calculate the sum of two matrices and obtain the transpose of the resultant matrix.

UNIT – III

6 What is a pointer? What are the problems with the pointers? Write a C program to print the elements of a one-dimensional array using pointers.

OR

7 Discuss in detail about the scope of functions with suitable examples.

UNIT – IV

8 Discuss in brief about the prototype of functions. Write recursive and non-recursive functions in C to calculate the n^{th} Fibonacci number defined below:

$f(n) = f(n-2) + f(n-1), \forall\, n \geq 2$ and $f(0) = f(1) = 1$

OR

9 How structures are passed to functions? Discuss bit fields and enumerations in brief.

UNIT – V

10 Write a C program to open a text file. Read the contents of the file and write the content into a new file by converting all the lower case letters into upper case letters.

OR

11 Discuss the following pre-processor directives:
(i) #define. (ii) #if. (iii) #else. (iv) #include. (v) #elif.

*****

Code: 15A05101 **R15**

B.Tech I Year I Semester (R15) Supplementary Examinations November/December 2019

COMPUTER PROGRAMMING

(Common to CE, EEE, CSE, ECE, ME, EIE and IT)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) Describe in one or two sentences about the phases in software development.

(b) Draw a flow chart for finding the second maximum number in a set of three given integers.

(c) If the base address of a two dimensional integer array A with 10 rows and 10 columns is 1100, find the address of the following elements. Size of integer is 2 bytes.
(i) A[5][7]. (ii) A[0][0]. (iii) A[9][1]. (iv) A[7][7].

(d) Illustrate the use of break statement with suitable code snippet.

(e) Write a C function to interchange two values using pointers.

(f) What is a type qualifier? What is its use? Give one example.

(g) What is an enumerated data type? Give an example.

(h) Give example for declaring and accessing bit fields.

(i) What do you mean by formatted I/O? Give an example.

(j) Write a C function to read two strings, compare them and print the result.

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 (a) Illustrate the working of bitwise operators in C.

(b) Write an algorithm and draw flow chart for computing the GCD of two given integers.

OR

3 Explain in detail about the data types in C.

UNIT – II

4 (a) Write a C program for calculating the sum of first 'N' odd numbers using for loop.

(b) Write a C function that takes a single dimensional integer array as input and prints the largest number among the elements of the array. Access the elements of the array using pointer arithmetic.

OR

5 (a) Write a C program for computing the sum of first 'n' terms of the following series using for loop.
$1 + 1/x^2 + 1/x^4 + 1/x^6 + \ldots\ldots\ldots\ldots\ldots$

(b) Explain the syntax and use of switch statement with suitable example.

UNIT – III

6 (a) Write a C function countEven(int*, int) which receives an integer array and its size, and returns the number of even numbers in the array.

(b) Write a brief note on storage classes in C.

OR

7 (a) Write a C program to declare memory for an array of integers dynamically and initialize the array with -1.

(b) Illustrate the scope of variables in C with suitable example.

Contd. in page 2

UNIT – IV

8 (a) Write a C program to determine mean and grade based on mean of the marks obtained by the students in three subjects. Grade is defined in below table.

Mean	Grade
90 <= x <= 100	A+
80 <= x < 90	A
70 <= x < 80	B
60 <= x < 70	C
x < 60	D

(b) Illustrate the use of typedef with suitable example.

OR

9 Provide an implementation of a function POINTshow(struct Rect, struct Point) that returns the position of the point with respect to the rectangle (i.e. Inside, outside or on). The rectangle corner points are stored in the structure Rect and the point coordinates are stored in a structure Point. Assume that the rectangle sides are parallel to the x and y axes.

UNIT – V

10 Write a C program to read a text file and print the following information. Provide the name of the file to read as command line argument:
(i)Total number of characters.
(ii) Total number of lines.
(iii) Total number of vowels and consonants.
(iv) Total number of words.

OR

11 Explain the following library functions with suitable examples:
(i) fopen. (ii) fread. (iii) fseek. (iv) fscanf. (v) fprintf

*****

Code: 15A05201 **R15**

B.Tech II Year II Semester (R15) Supplementary Examinations December 2019

DATA STRUCTURES

(Electronics & Communication Engineering)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) List the areas where data structures can be applied.
(b) Mention the merits and demerits of arrays.
(c) Define circular queue full condition.
(d) What is a hash function? Give an example.
(e) What is a skewed tree? Give example.
(f) Differentiate between full and complete binary tree.
(g) Which sorting is called as stable sorting? Give the reason.
(h) Define time complexity. Write down the best, worst and average case time complexity for exchange sort algorithm.
(i) What is meant by collision? When it happens? Give example.
(j) In what way a linear search differs from binary search?

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 What is sparse matrix? Explain array and linked list representation of a sparse matrix.

OR

3 (a) Write an algorithm to find the largest element present in an array.
(b) Write and explain algorithms for inserting and deleting an element from double linked list.

UNIT – II

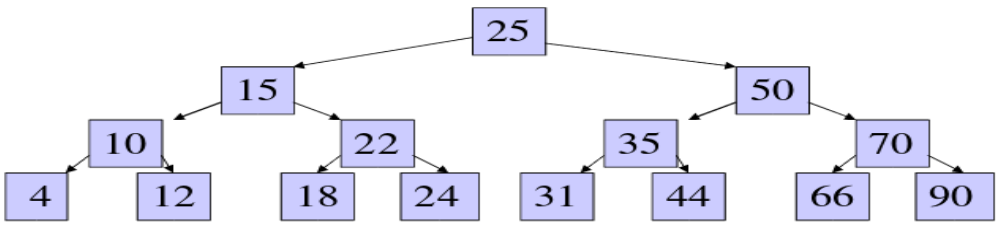
4 Write a procedure to convert any given infix expression into postfix form. Using step by step procedure convert the following infix expression into postfix form: A + (B * C) – ((D*E+F)/G).

OR

5 (a) Write and explain the different types of double ended queues.
(b) Give brief description about the queue storage using arrays.

UNIT – III

6 What is meant by tree traversal? List the different traversals used in binary trees. Find the different traversals for the following binary tree.



OR

7 (a) Write and explain the properties of a red – black tree.
(b) With the help of suitable example, explain topological sorting.

Contd. in page 2

UNIT – IV

8 Write the procedure to sort a set of elements by using quick sort. Apply quick sort mechanism to sort the data {J, N, T, U, E, X, A, M}.

OR

9 Apply max heap sort on the list of elements 14, 12, 9, 8, 7, 10, 18, 20, 30.

UNIT – V

10 Define hashing. With the help of suitable example, explain different types of hashing.

OR

11 With the help of suitable example, explain linked list collision resolution and bucket hashing.

*****

Code: 15A05201 **R15**

B.Tech II Year I Semester (R15) Regular & Supplementary Examinations November/December 2019

DATA STRUCTURES

(Electrical & Electronics Engineering)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) Distinguish between linear and non linear data structures.

(b) What is a sparse matrix? What are its disadvantages?

(c) In how many ways a stack can be stored in memory? List them. Which is the best one?

(d) Define hashing. List their advantages.

(e) Find the value for the following postfix expression: 1 2 + 3 4 * - .

(f) In what way a max-heap differs from min-heap.

(g) Define time complexity. What is the worst, average and best case time complexity for bubble sort?

(h) What is external sorting? Why do we need it?

(i) What is meant by searching? Write the best and worst case time complexity for linear search.

(j) How can we say that binary search is better than linear search?

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 What is meant by performance analysis? What are the factors used to measure the performance of algorithm? Write and explain the various asymptotic notations.

OR

3 In how many ways a new node can be inserted into a double linked list? Explain them with neat sketch.

UNIT – II

4 (a) How can we say that a stack follows LIFO principle? Justify your answer.

(b) Explain in detail about the circular queues.

OR

5 Write the procedure to convert an infix expression into postfix form. Convert the following infix expression into post fix form by using stack:
M + (N * O) – ((P * Q + R) / S)

UNIT – III

6 (a) Distinguish between tree and binary tree? Write about the array representation of binary tree with an example.

(b) Why do we need height balanced binary search trees? Explain with simple example.

OR

7 What is shortest path problem in a graph? Describe an algorithm to solve it. Illustrate your algorithm by taking an example.

Contd. in page 2

UNIT – IV

8 Describe the method of merge sort? Apply this to sort the following data:
{ J, N, T, U, A, E, X, M, I, N, A, T, I, O, N}

OR

9 (a) Write an algorithm for sorting set of elements by using quick sort? Explain with an example.
(b) Give brief description about sorting elements using shell sort technique.

UNIT – V

10 Write and explain any five hashing techniques with suitable examples.

OR

11 (a) Write the idea behind binary searching? Illustrate with example.
(b) With the help of an example, explain open addressing collision resolution method..

*****

Code: 15A05201

SS

B.Tech II Year I Semester (R15) Regular & Supplementary Examinations November/December 2019

DATA STRUCTURES

(Computer Science & Engineering)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) In what way a recursive algorithm differs from iterative algorithms.
(b) List the various types of linear and non linear data structures.
(c) PUSH(A), PUSH(B), POP(), PUSH(C), PUSH(D), POP(), POP() if these operations are performed on a empty stack, show the final contents of the stack.
(d) List the applications of queues.
(e) Distinguish between full and complete binary trees.
(f) Define height, level and degree of a tree.
(g) What is meant by sorting? Write the best, average and worst case time complexity for insertion sort.
(h) State the logic behind the shell sort.
(i) Define collision? List the collision resolution techniques.
(j) List the benefits of hash tables.

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 Write and explain the working of circular linked list with example.

OR

3 List the operations that can be performed on single linked list. In how many ways a node can be deleted from single linked list? Explain.

UNIT – II

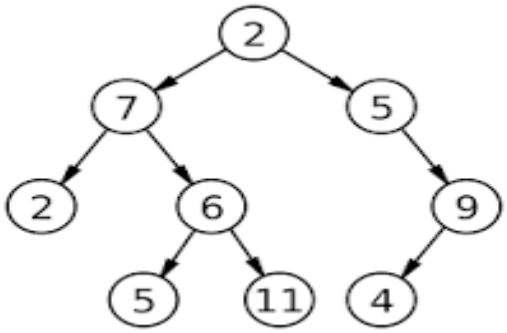
4 Write the steps to evaluate a postfix expression. Evaluate the following postfix expression using stack: 8 3 4 + - 4 9 3 / + * 2 ^ 3 +

OR

5 (a) Define queue. In how many ways a queue can be stored in memory? Explain any one method with suitable example.
(b) How can we insert an element into a circular queue? Demonstrate with example.

UNIT – III

6 Write the recursive algorithms for different binary tree traversal techniques. Find the tree traversals for the following binary tree:



OR

7 Define binary search tree. Explain with example deletion of an element from a binary search tree.

Contd. in page 2

UNIT – IV

8 Sort the following set of elements by using min – heap property: {33, 22, 11, 44, 55,66, 32, 23,39, 100}.

OR

9 (a) What is meant by external sorting? Why we need it? Explain.

(b) Sort {U, N, I, V, E, R, S, I, T, Y} using merge sort.

UNIT – V

10 Define hashing. Discuss various hashing functions with suitable examples.

OR

11 (a) Explain chaining with example.

(b) Apply binary search to find the element 12.11 from the set {1.1, 2.33, 4.9, 12.11, 13.33, 14.44, 15.55, 16.66}. Show each step clearly.

*****