Unit-1

Fundamentals of Computer

* What is a Computer

A computer can be defined as an electronic device i.e., designed to except data perform the required Mathematical and logical operations at high speed and output the result

Characteristics of computer :-

1) Speed :

Computers can perform Millions of operations per second. The speed of computer is usually given in Nano seconds and pico seconds

1 Nano Second = 1×10^{-9}

1 pico second = 1×10^{-12}

2) Accuracy :

A computer is a very fast, reliable and robost electronic device. It always gives Accurate result provided the correct data and set of instructions are input to it.

3) Automation :

Besides being very fast and accurate, computers are automatible devices that can perform a task without any user intervention.

4) Deligence :

∴ unlike humans, computer never get tired of a repeatative task. It can continuously

work for hours without creating errors.

5) Versatility :

Versatility is the quality of being flexible Today computers are used in our daily life in different fields

6) Memory :

Computers have internal or primary memory as well as External or Secondary memory. While the Internal memory of computers is very Expensive and limited in size. The Secondary Storage is cheaper and of bigger capacity

7) Economical :

Using computers reduces man power requirements and efficient way of performing various tasks. Hence computers save time, energy and money.

* Generations of computers :-

"Generation" indicates a state of improvement in the product development process. When applied to computing it refers to the different advancements of new computer technology.

First Generation (1940 - 1956) :

First Generation computers used a very large number of "Vaccum tubes" for circuitary and Magnetic drums for memory. This used

Mechine language which is the lowest level programming language consists of 1's and 0's

"UNIVAC" (Universal Automatic Computer) and Electronic Numerical Integration and Calculato (ENIAC) are Examples of first Generation computing devices.

Advantages:

* They were the fastest calculating devices of their time

Disadvantages:

* They generated a lot of heat
* They consume a lot of Electricity
* They were very Bulky in size
* Vaccum tubes had limited life
* They were very Expensive

Second Generation (1956 - 1963):

These computers were Manufactured usi "transistors". Transistors were smaller, faster, cheaper and reliable. At this time Computer used assembly language which allowed In- structions in words. At this time high level Programming languages such as. COBAL, FORTRAN ALGOL fo also being developed.

Advantages:

* They consumed less Electricity and gene rated less heat has compare to first generation computers

* They were faster, cheaper, smaller and more reliable.
* Computers had faster primary memory and larger secondary memory

Disadvantages:

* Production of computers difficult and Expensive.

Third Generation (1964-1971):

Third Generation computers were manufactured using Integrated circuits. These were smaller, less Expensive more reliable and faster in operations consumed less power and generated less heat. These computers had a few mega bits of main memory and Magnetic disk that could store a few tens of mega bits of data ~~for~~ per disk drive

Some more high level programming language such as PASCAL, BASIC were developed for

Advantages:

* They were smaller, cheaper and more reliable than ~~there~~ their processors.
* These computers had faster and larger primary and secondary storage.
* These were used for scientific and business applications.

Disadvantages:

* These computers were difficult to maintain
* They got heated very quickly

Fourth Generation: (1971-1989)

Micro Processor launched the fourth generate of computers with thousands of integrated circuits, built on to a single Silicon chip.

Many new operating systems were developed including MS-dos, Microsoft windows, UNIX operating systems.

Advantages:

* they used as general purpose computers.
* They were smaller, cheaper, faster and reliable than their processors.

disadvantages:

* They were not intelligent systems.

Fifth Generation (present at):

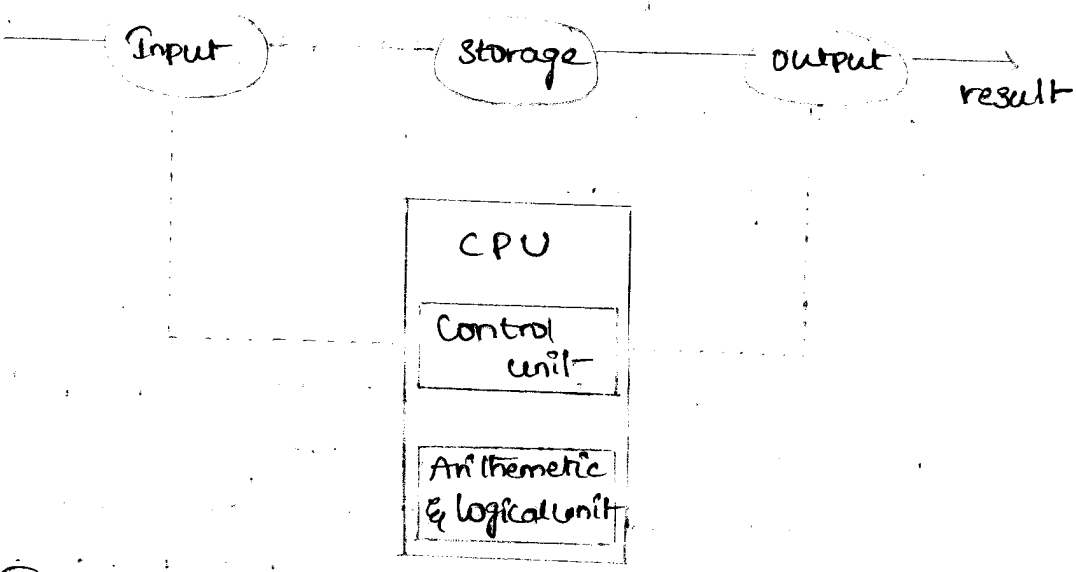
These are completely based on the "artificial intelligence" (AI). These computers are still in development these are certain application such as voice recognization, gaming, Expert Systems, Natural language computers, ~~Neural~~ Neutral Network, robotics computers.

* Basic Computer Organisation :

A computer is an Electronic device that basically performs 5 major operations.

1. Accepting data or instructions
2. Storing data
3. processing data
4. Displaying result
5. Controlling and coordinating of all operations inside a computer

Block diagram of a computer :



Input :-

This is the process of Entering data or instructions into the computer systems.

The data and instructions can be entered by using different input devices such as keyboard, Mouse. Input devices convert the input data into binary codes.

Storage :-

It is the process of Saving data and

instructions in the computer. A computer has two types of storage areas.

Primary Storage:

It is used to store the data and parts of programs primary storage space is very expensive and therefore, limited in capacity, it is voltaile. If the computer is switched off information stored gets based.

Eg: RAM.

Secondary Storage:

It is cheaper, non-voltaile and use to permanently store data and programs. It has limited storage capacity.

Eg: Magnetic disk

Processing:

The process of performing operations on the data as per the instructions specified by the user is called processing.

Data and instructions are taken from the primary memory and transfered to the "ALU"

ALU:

Which perforns all sorts of calculations, intermediate results stored in the main memory. The processing completes the result is then transfered to the main memory.

Output :-

output is the process of giving the result of data processing to the outside world.

Control Unit :-

It Manages and Controls all the component of the computer system. Control unit- desides th which instructions will be selected and opera tions performed. It takes the step by step processing of all operations

* Uses Of Computers :-

Government :

Computers are used in Government organisa tions to keep records, revence service Interna records.

Traffic control :

In the US computers are used by the gover ment for city planning and traffic control.

Legal System :

Lawers use computers to look throw millions of individual cases and keep track of app -ments.

Retail bussiness :

Computers are used in Retail shops to Enter orders, calculate cost and print recei

Sports :

computer are used to identify weak Player and strong players in the control room to display action replays.

Music :

computer to generate a variety of sounds, background music in movies, TV shows.

Movies ;

Computers are used to create sets, special effects, animations

Business and Industry :

Computers are used mainly for data processing which include task's such as word processing, analyzing data, Entering records.

Hospitals ;

Hospitals uses computers to record every information about patients. Computer controlled devices are widely used to monitor pulse rate, Blood pressure.

Weather forecasting :

Computers are used in weather forecasting for finding air press, temperature and weather of particular area.

Education ;

A computer is a powerful teaching aid and acts as another teacher in the class room. Teachers may use picture graphics and graphical presentations. students can also give online exams and get instant results

Online Banking :

The world today is moving towards a

Flatbed plotter :

In a flat-bed plotter the paper is spread on the flat rectangular surface of the plotter and the pen is moved over it.

Flatbed plotters are less expansive and are used in many small computing system.

* Algorithm :—

Algorithm is a step by step processor for solving problems using a computer it is necessary to evolve a details and precise step by step methods of solutions

Examples : ①

Step-1 : Start
Step-2 : Take a,b values
Step-3 : Compare a,b values (using if Condition)
Step-4 : If a<b then
Step-5 : Print b is big
Step-6 : Hence Else
Step-7 : Print a is big
Step-8 : Stop

② Addition of two numbers

Step-1 : Start
Step-2 : Read a,b values
Step-3 : Add a,b values
Step-4 : Store that adding value in c (c←a
Step-5 : Print the c value
Step-6 : Stop

Problem Solving using Computers :

To Solve a problem using a Computer to the following steps

* The given problem is Analysed
* The Solution method is broken down into a Sequence of elementary tasks
* Based on these analysis an algorithm to Solve the problem is formulated
* The Algorithm should be precised and unambiguous Based on our discussions we realise that algorithm formulation is difficult and time consuming
* An Algorithm expressed using a precise Notation is called a Computer program.
* The Computer program is fed to the Computer
* The Computer's processing unit interprets the instructions in the programs, executes them and sends the result to the output unit

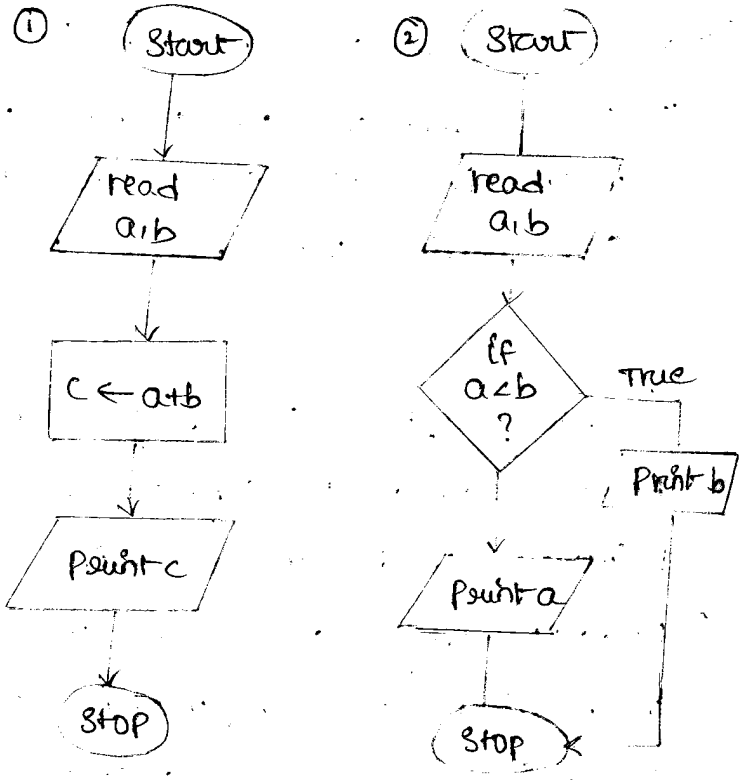
Flow chart :-

We express the algorithm in a pictorial Form called a flow chart. The flow chart is Primarily used as an Aid in formulating and understanding algorithm. The Sequencing of instructions and repetation of groups of instructions may be quickly seen by inspecting a flowchart.

The following shapes are used for various blocks in a flowchart

1. Rectangles with rounded ends are used to indicate START and STOP
2. Parallelograms are used to represent input and output operations
3. Diamond shaped boxes are used to indicate questions asked or conditions tested based on whose answers appropriate exits are taken by processor the exits from the diamond shaped boxe are labeled with the answers to be questions
4. Rectangles are used to indicate any processing operation such as storage and Arithematic

Eg:



* Programming Languages :-

A programer has to remember all the operation codes of the computer and know in detail what each code does & how it effects various registors in the processor. computer programing languages are developed with the primary objective of facilitating a large number of people to use computers without the need to know in detail the internal structure of the computer. Languages are matched to the type of operations so to be perform in algorithms for various applications.

Languages are also designed to be machine independent.

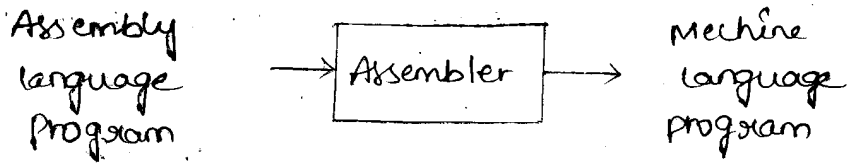
1) Assembly Language :

The first step in the evaluation of programming languages was the development of what is known as an Assembly language. In an assembly languages Mnemonics are used to represent operation codes, and strings of characters to represent addresses. The symbolic addresses of operands used in Assembly language should also we converted to absolute Numeric addresses.

The Translator which does this is known an Assembler. The input to an Assembler

is the assembly language program and is known as the source program. It's output is the equilant Mechine language program and is known as the object program.

The Assembler is the system program which is supplied by the computer manufacturer.



(Source program) ——————→ (Object program)
one-to-one

Advantages :

The main advantage of using an Assembly language for programing is the efficiency of the Mechine language program resulting from it As all hardware feature available in the processor of the computer, such as registors, stacks

Disadvantages :

* It is Mechine dependent. Thus programs return for one model of a computer cannot be executed on another model. It is not protable from one Mechine to another.
* An Assembly language programmer must be an expert who knows all about the logical structure of the computer.
* Writing Assembly language program is

rogram and is
n. It's output
guage program
program.
stem program which
manufacturer.

Mechine
language
program

ject program)

ing an Assembly
the efficiency
ram resulting
ature available
puter, such as

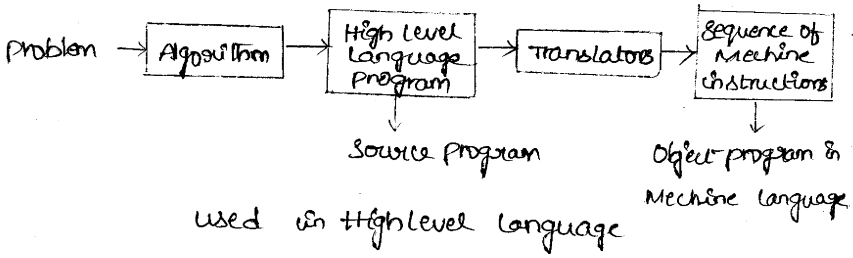
Thus programs
computer cannot
odel. It is not
to another.
grammer must
all about the
computer.
program is

difficult and time consuming

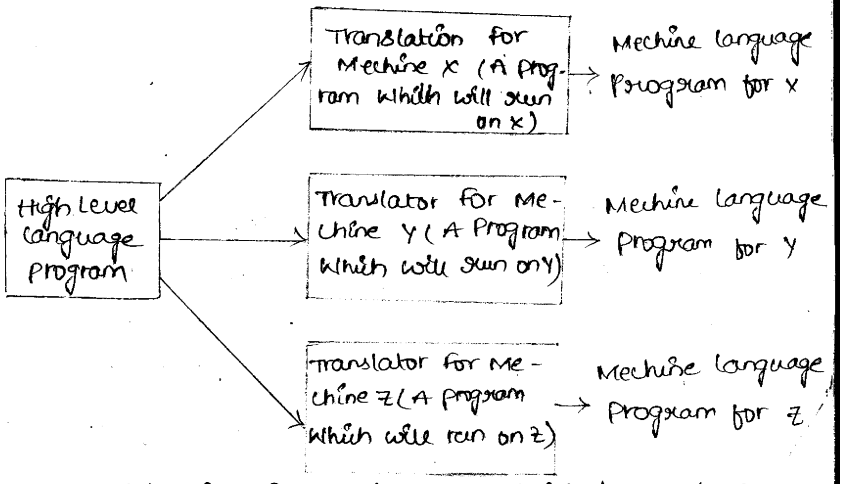
2) Higher level programming language:

High level languages are designed independent of the structure of a specific computer. This facilitates executing a program written in such as language on different computers. Associated with each high level language is a elaborate computer program which translates it into the Mechine language of the computer in which it is to be executed

The translator program is Normally written in the Assembly language of that computer



Used in High Level Language



Mechine independence of High Level Language

C - Language

Introduction :- (1)

'C' Language was developed by Dennis and Ritche in 1972 at AT and T (American telegraph and telecommunication) Bell Laboratories.

Features :-

1. portable :

The c language exe file can easily carry one or more from one system to another system.

2. Flexible :

We can easily perform any modifications (or) change in c program (i.e., editing is possible in C program)

3. Builten functions :

Several predefined functions are available in C Language.

Eg : Mathematical functions (Powerc(), Sqrt())
String functions (Strlen())

4. General purpose :

The c language can be used for both purpose. Those are Scientific application Purpose and bussiness application purpose.

5. Easy to Learn :

The c language is almost in English, So every one can easily learn

6. Character Set of C Language :

Alphabets : a to z and A to Z

Digits : 0 to 9

Wide Space characters:

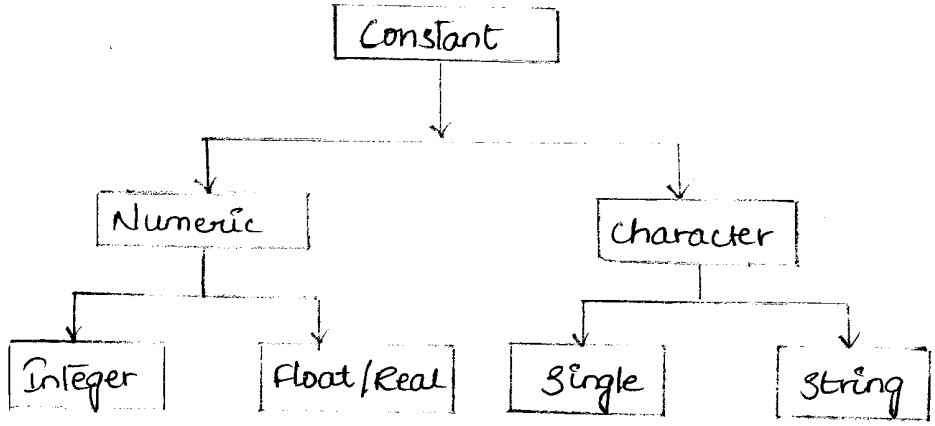
New line (\n)

Horizontal tab (\t)

Backspaces (\b)

Constants :—

Constant is Nothing but a quantity cannot be changed. It can be stored in some memory location. It is classified into two types.



1. Integer :

Without decimal point Numbers are called integer Numbers.

Eg: 2, 23, -25 etc.

Rules

1) It Must have atleast a Single digit

2) It Must Not have decimal point Numbers.

3) We can use +ve, -ve Numbers

4) We don't give blank space, commas between the digits.

5) The range is -32768 to 32767

2. Float (or) Real :

The decimal point Numbers are called float Constants. In c language the real constant can be represented in two ways

(i) Factorial Form

(ii) Exponential form

(i) Factorial Form Rules :

1) It Must have a decimal point Numbers.

2) We can use +ve and -ve Numbers

3) We don't give blank spaces, commas b/w digits

Eg: 2.5, 33.5, 5.5 etc.

(ii) Exponential Form :

In Exponential Form the real constant is expressed as two parts, the part before 'e' is called "Mantisa" and the Part After 'e' is called "exponential" part

Rules

1) The Mantisa and exponential parts are seperated with the letter 'e'

2) We can use the +ve (or) -ve Numbers to Mantisa and exponential parts

3) The range of the real constant in exponential Form is -3.4 e 38 to 3.4 e 38

Character Constant :

a) Single character constant :

A Single character enclosed with in a pair of Single quote mark is called as Single

Character constant. The character may be alphabet (or) digits (or) any special symbol. The range of character constant is -128 to 127

Eg: '5', 'a' etc.

b) String Constant :

A group or collection of characters enclosed within a pair of double quote marks is called as String constant.

Eg: "MCA", "Student", etc.

Variable :—

Variable is Nothing but a name which is given some memory location where the constant is stored. The value of the variable may be changed during the program execution.

Rules :

1) We can take alphabets or digits or only one special symbol i.e., '_' as a variable name.
2) In the variable name the first letter always must be a character or alphabet.

Eg: If we take $2x+3y=15$, in this equation 2,3 are constants and x,y are variables.

* Data Type :—

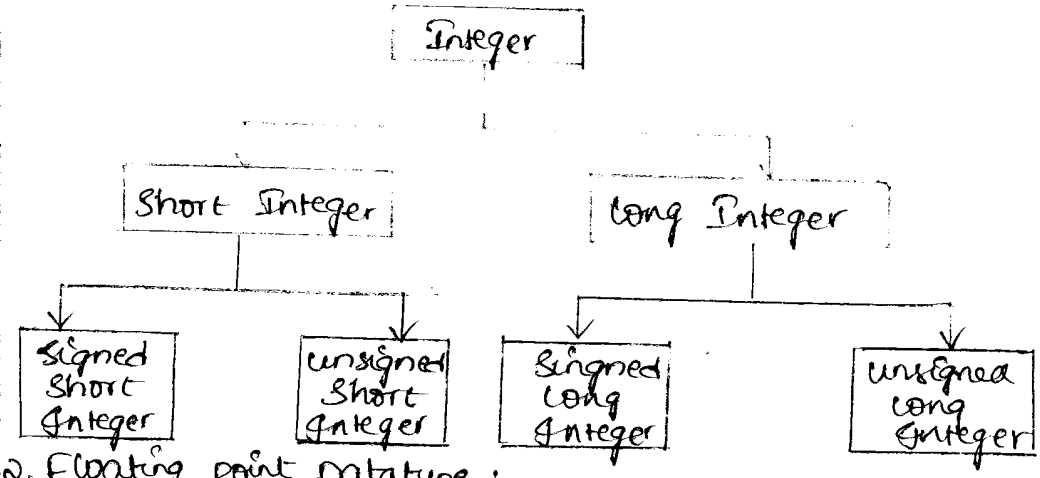
Data type may be used to identified the type of the data. In C language the datatypes are mainly classified into three types

1. Integer
2. Float (or) Real
3. character

1. Integer Datatype :

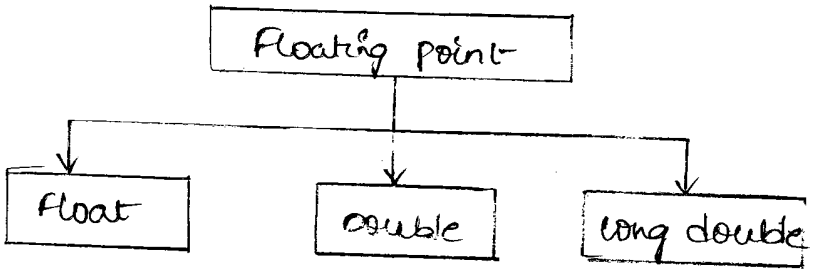
Integer Datatypes are classified as a short integer, long integer.

It is shown in the following figure.



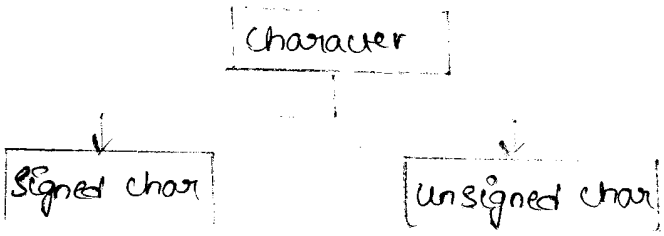
2. Floating point Datatype :

It is divided into mainly three types those are float, double, long double. It is showned by following figure.



3. Character Datatype :

It is divided into mainly two types those are Signed char, unsigned char. It is shown in the following figure



S.No	Datatype	Range	bytes	Formate Specifiers
1.	Signed Short int (or) Short int (or) int	-32768 TO 32767	2	%d, %i (b)
2.	unsigned Short int (or) unsigned int	0 to 65535	2	%u
3.	Signed long int (or) long int (or) long	-2147483648 to 2147483647	4	%li, %ld
4.	unsigned long int (or) unsigned long	0 to 4294967295	4	%lu
5.	Float	-3.4e	4	%f (or) %a
6.	double	-1.7e 308 to +1.7e 308	8	%lf (or) %la
7.	long double	-1.7e 4932 to 1.7e 4932	10	%lf
8.	Signed char (or) char	-128 to 127	1	%c
9.	unsigned char	0 to 255	1	%c

Keywords :

Keywords are sometimes called it is reserved words. This words are used as a specific purpose

Eg: int, float, if, for, etc.

In 'C' language 32 keywords are available

Declaration and initialization of variable :

Declaration of variables :

1) The variable must declare in the declaration part of the program.
2) The variable must declare before using that variable.
3) If we declare any variable it provides two things a) what is variable name
   b) What data (or) value the variable is hold

Syntax : Datatype variablename ;

Eg : int a;
     float p;
     char ch;

Initialization of variable :

1) The value will be assigned into the variable is called initialization of variable.
2) Here we use '=' Operator
3) The declaration and initialization of the variable both are in the same line (or) in different lines.

In the same line :

Syntax : Datatype variablename = constant ;

Eg : int a=2;
     float p=5.6 ;

In the different line :

Syntax : variable name = Constant ;

Eg : int a;
     a = 2;

* Identifiers :-

Identifier is Nothing but a name which is given to variable or Array or function. Here also we follow the same rules of variables.

* Operators :-

The operators are used to perform different types of operations. These operators are used with Operands. The C language contains 7 types of Operators. Those are

1. Arithemetic operator
2. Relational operator
3. Logical operator
4. Assignment operator
5. Conditional operator
6. Comma operator
7. Bitwise operator

1. Arithmetic Operator :

Arithmetic operators performs arithmetic Operation These are divided into two types

(i) Binary arithmetic operator

(ii) Unary arithmetic operator

(i) Binary Operator :

These operators are used in between two operands.

Syntax :

Operand$_1$ Binary arithmetic operator Operand$_2$

Operator	Meaning	Example int a=2, b=3
+	Addition	2+3 or a+b;
-	Substraction	2-3 or a-b;
*	Multiplication	2*3 or a*b;
/	Devision	2/3 or a/b;
%	Modular division	2%3 or a%b;

(ii) Unary Operator :

These operators are used with only one operand.

Operator	Meaning
-	Minus
++	Increment
--	Decrement
&	Address operator
( )	Size of

a) Minus (-) :

This operator change the size of an operand.

Syntax : -operand;

Eg : -2, int a=5;

-a;

b) Increment operator :

It adds '1' to its operand we can write pre increment or post increment like below.

Syntax : ++variable ; (or) variable++ ;

Operator	Meaning	Process
++x	pre increment	x = x+1
x++	post increment	x = x+1

Pre increment and post increment both are perform same operation but a small different is there that is illustrated with the following example.

Post increment Eg :

```
int x = 2;
int y;
y = x++;
```

According to the above statement first x value is assigned into y variable. Then y value is 2, when the cursor coming to the next line x is incremented then x value is 3

Pre increment Eg :

```
int x = 2;
int y;
y = ++x;
```

According to the above example, first x is incremented after that value assigned into y variable then y = 3 and x = 3

c) Decrement operator (--) :

It substracts '1' from its operand we can write pre decrement (or) post decrement like below.

Syntax : --Variable ; (or) Variable -- ;

Operator	Meaning	Process
--x	Pre decrement	x = x-1
x--	Post decrement	x = x-1

Predecrement and post decrement both are Perform same operation but a small difference is their that is illustrated with the following example

Post decrement Eg :

```
int x = 2;
int y;
y = x--;
```

According to the above statement first x value is assigned into y variable, then y value is 2. When the cursor coming to the next line x is decremented then x value is 1.

Pre decrement Eg :

```
int x = 2;
int y
y = --x;
```

According to the above statement first x is decremented after that the value is assigned into y variable then y value is 1 and x = 1.

d) Address operator ;

If we use this operator before any variable name then it gives the address of that variable

Syntax : & variable name;

Eg: int a;
printf("%d", &a);

e) Size Of Operator (()):

This operator gives the number of bits occupied by any variable or constant

Syntax: sizeof (variable name / constant);

Eg: int a;
sizeof (a);
It gives two bytes.

2. Relational operator:

These operators are used to compare two operands. If the Expression result is true it returns '1' otherwise '0'.

Syntax: operand1 relational operator operand2

Operator	Meaning	Example	Return Value
<	Less than	2<4	1 (True)
>	Greater than	5>6	0 (False)
<=	Less than or equal to	2≤4	1 (True)
>=	Greater than or equal to	5≥6	0 (False)
==	Equal to	5==5	1 (True)
!=	Not Equal to	5!=4	1 (True)

Eg: int a=2, b=4;
a>b
it returns value 0
a<b
it returns value 1

3. Logical Operator:

These operators Combine two or more relational expressions. These operators are

Operator Name	Operator
Logical AND	&&
Logical OR	||
Logical NOT	!

(i) Logical AND (&&) :

* The operator before expression and after Expression both are true then the final result is true (1)
* The operator before expression or after Expression any one is false then the final result is false (0)

Syntax :

Relational expression1 && relational Expression2 && relational Exp3 .....

Eg : 1. (3>2) && (2<5) => 1

2. (3==2) && (2<5) => 0

(ii) Logical OR (||) :

* The Operator before expression (or) after Expression any one is true then the final result is true (1).
* The Operator before expression (or) and after expression both are false.

Syntax :

Relational expression1 || relational expression2 || relational Exp3 || .....

Eg : (3>2) || (2<5) || .....

(iii) Logical NOT :

* The relation is true the Operator consists false
* The relation is False the Operator Converts True.

Syntax : ! Relational expression

Eg : 1. ! (3>2) ⇒ 0

2. ! (3==2) ⇒ 1

4. Assignment Operator :

The right side value will be assigned into the left side ~~value~~ variable.

Syntax : variable name = variable / constant / expression ;

Eg :
```
int a=2;
int b;
b=a;
```

5. Comma Operator :

This operator is used to separate a list of variable.

Eg : int a,b,c ;

6. Conditional (or) Ternary Operator :

? and : are both are called conditional operator.

Syntax : exp1 ? exp2 : exp3 ;

If the exp1 is true, after the ? the expression 2 is executed if the exp1 is false after the : the exp3 is executed.

Eg : The following example is for finding the biggest of two Numbers

```
int P=3, q=2;
(P>q) ? printf("P is big") : printf("q is big");
```

7. Bitwise Operator ;

These operators perform manipulation or chang on each bit of data (ie, this operators perform directly Manipulations on bits). These operator are used for testing (or) Shifting (or) Complementing.

Bitwise operators are classified into three types.

(a) Bitwise logical
(b) Bitwise Shift
(c) Bitwise complement

(a) Bitwise Logical ;

These operators are used for testing. The bitwise logical operators are

(i) Bitwise AND(&)
(ii) Bitwise OR (|)
(iii) Bitwise EX-OR (^)

(i) Bitwise AND(&) ;

If the corresponding bits in both operands then the resultant bit is '1' otherwise '0'

Truth Table ;

Input		output
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Syntax : Operand1 & Operand2 ;

Eg: int a=13, b=7, c;

c = a&b

a = 0000 0000 0000 1101

b = 0000 0000 0000 0111

c = a&b = 0000 0000 0000 0101

The result of c in decimal is '5'

Bitwise OR (|) :

If the corresponding bit in any operand is 1, then the resultant bit is 1, otherwise if the corresponding bit in both operands '0' then the resultant bit is 0.

Input		output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Syntax: Operand1 | Operand2

Eg: int a=13, b=7, c;

c = a|b;

a = 0000 0000 0000 1101

b = 0000 0000 0000 0111

c = 0000 0000 0000 1111

The result of c value is 15

Bitwise Ex-OR (^) :

If the corresponding bits in both operands '1' and '0' then the resultant bit is 0, otherwise 1

Input		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Syntax : Operand$_1$ ^ Operand$_2$

Eg: int a=13, b=7, c;

c = a^b;

a = 0000 0000 0000 1101

b = 0000 0000 0000 0111

c = 0000 0000 0000 1010

The result c value is '10'

Bitwise Shift Operator :

These operators shift or move bits towards leftside and Right side. It is divided into two types

(i) Bitwise left shift operator (<<)

(ii) Bitwise Right shift operator (>>)

(i) Bitwise Left shift Operator :

It is denoted with '<<' this operator shift or move bits to leftside & Right side

Will be adding '0' (or) this Operator shift or move the bits 'n' times to leftside and Right will be adding zero's

Syntax : Operand << operand

Eg : (c = b<<2

The original bits of b is)

① int a=13, b=7

a<<b

a = 0000 0000 0000 1101

b = 0000 0000 0000 0111

1st time Shifting - 0000 0000 0000 1010

2nd time Shifting - 0000 0000 0000 0100

3rd time Shifting - 0000 0000 0110 1000

In left shift we can find the final result in decimal form by using following formula

Operand * 2^n

⇒ $13 * 2^3$

⇒ 104

② c = b<<2

The Original bits of b is 0000 0000 0000 0111

1st time Shifting 0000 0000 0000 1110

2nd time Shifting 0000 0000 0000 1100

The resultant of binary format is converted into decimal Number by using the following formula.

Operand * 2^n

$b * 2^n$

7×2^2

7*4

28

The resultand c value is '28'

(iii) Bitwise Right Shift Operator :

This Operator shift or move bits to Right-side and left-side will be adding zero's (or)

This Operator shift or moves bits n time to Right side and left will be adding zero.

Syntax: Operand >> Operand

Eg: c = b>>2

The original bits of b is 0000 0000 0000 0111

1st time shifting 0000 0000 0000 0011

2nd time shifting 0000 0000 0000 0001

The resultant of binary format is converted into decimal number by using the following formula

Operand $/2^n$

$b/2^n$

$7/2^2$

$7/4$

1.7

1

The resultant c value is '1'

Bitwise compliment Operator (~):

This is also called as one's compliment this operator is represented with '~' symbol. In this the bit '1' is inverted or changed as a '0' and the bit '0' is inverted or

changed as '1'

Syntax: ~ operand

Input	output
a	~a
1	0
0	1

Eg: ~a

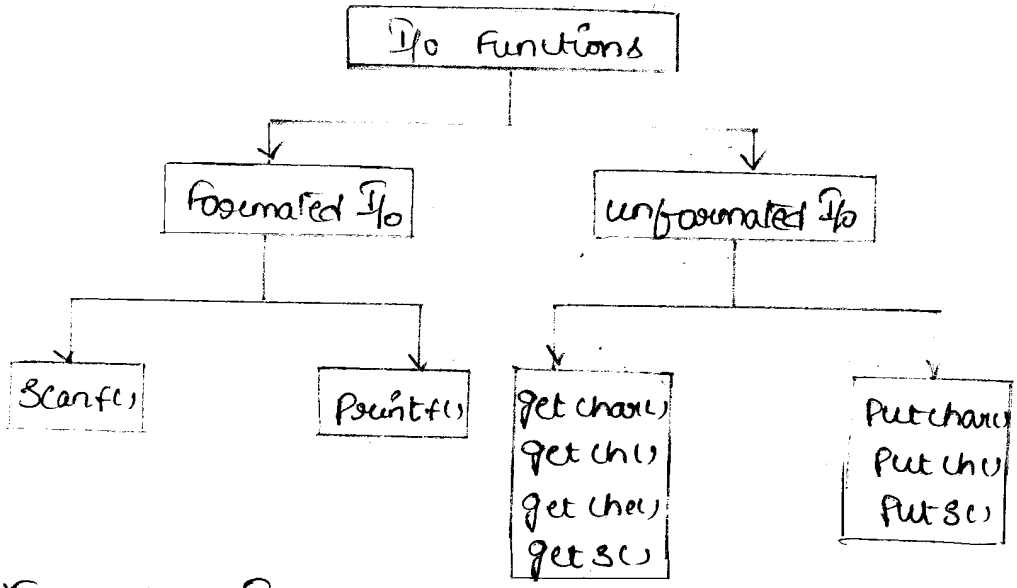
The original bit of a is 0000 0000 0000 1101

~a is 0000 0000 0000 0010

* Input/Output (I/O) functions :—

The input function reads the data from the keyboard the output function (print) writes the data on the output device.

I/O functions are mainly classified into two categories



(i) Formatted I/O functions :—

This function reads and writes (print) all

datatype values. In these functions we use formate specifier

Formated output function:

Printf(): This functions writes/prints all datatype value on the output device.

Syntax: printf ("<formate string>", <list of variables>)

In the formate string we may use the following 3 objects

Formate Specifiers:

%d - For printing integer value
%f - for printing float value
%c - For printing character value

Alphabets, digits, Special symbols:

a to z, A to Z, <, >, = etc.

Wide space characters:

\n, \t, \d etc.

Eg: int a = 2;
Float b = 4.5,
Char ch = a
Printf ("%d %f %c", a, b, ch);

We can use formate specifiers and list of variables.

The Number of formate specifiers are equal to Number of variables.

In the printf statement the list of variables is optional.

Eg: printf("Enter any two Numbers");

Formatted Input function:

Scanf(): This function reads all datatype values from the keyboard.

Syntax: Scanf(<"Formate String">,&<list of variables>);

* In the formate string we use the formate specifiers

%d - reading integer value
%f - reading float value
%c - reading character value

Eg:
```
int a;
float b;
char ch;
Scanf("%d,%f,%c", &a, &b, &c);
```

* We can use formate specifiers and list of variables.
* The Number of formate specifiers are equal to Number of variables.
* Before each and every variable we should give '&'

NOTE:

If the Scanf() Statement or printf() statement how many successfull read/write the value must be written

Eg:
```
int a;
float b;
char ch;
```

scanf("%d %f, %c", &a, &b, &ch);

If the above scanf() reads 3 variables successfully it returns value 3. Suppose it reads only two variables successfully it returns values if it is not successfully reads any value It returns a Negative Number. The same process is also applied to printf()

(ii) Unformated I/O functions:

These function reads/writes (print) only character data

* In this we don't use formate specifiers
* The unformated I/o functions are classified into 2 types
  (i) character I/o
  (ii) String I/o

Character I/o function:

This function reads/write only a single character (character data) at a time.

Character Input functions:

a) getchar(): This function reads at a time a single character.

Syntax: variable = getchar();

Eg: char ch;
    ch = getchar();

b) getch(): This function reads at a time a single character

Syntax: variable = getch()

Eg: char ch;
ch = getch();

According to the above statement getch() reading character is assigned into 'ch' variable

getche(): This function reads at a time single character from the keyboard

Syntax: variable = getche()

Eg: char ch;
ch = getche();

In the above examples if we giving any character that will be taken by these functions and stores into 'ch' variable.

NOTE:

A difference between getchar() and getch() is: If we using getchar() the reading character will be displayed for output device and it waits for Enterkey. If we are using getch() the reading character will not be displayed on the output device and it must not wait for Enterkey.

Character output functions:

a) putchar(): This function writes at a time a single character

Syntax: putchar(variable);

Eg: ① char ch = 'a'
putchar(ch);

② char ch;
ch = getchar();
putchar(ch);

b) putch() : This function writes at a time a single character

Syntax: putch(variable);

Eg: charch = a;
putch(ch);

① char ch;
ch = getch();
putch(ch);

String I/O functions :

These functions reads / writes a string.

gets() : This is a string input function it reads a string.

Syntax: gets(string variable);

Eg: char ch[10];
gets[ch];

puts() : This is a string output function it writes a string.

Syntax: puts(string variable)

Eg: char ch[10] = "First MCA";
puts(ch);

Some Common library functions :-

1. clrscr() :

* It means clear the screen
* This function clears previous program output for displayed the currently executing program output

Syntax: clrscr();

* This function in coin.h header file

2. Exit :

* This function permanently terminate or stop the

currently executing program.

Syntax: Exit(Any integer number);

Eg: Exit(1);

* This function in process.h header file.

3. Sleep():

* This Function temporarily stop or terminate the currently executing program.

Syntax: Sleep(number of milliseconds);

Eg: Sleep(10,000);

* This Function in DOS.h header file.

4. System():

* This Function Executes various DOS Commands

Syntax: System(any DOS command);

Eg: System(dir);

* This function in DOS.h header file.

Type Conversion (or) Type Casting :-

Suppose the assignment operator Right side expression result or variable may not be match with left side variable datatype. In such a case the right side expression result or variable value are constant is promoted or demoted.

It is Explained with the following example.

```
float p;
int a=5, b=2;
P=a/b;
```

According to the above statement p value is 2.0 (The a/b result is 2.5) but the a/b result

is 2.5 so, the result is denoting we require type casting

Syntax: Function (datatype) Expression

Eg: float p;
int a=5, b=2;
p = (float) a/b;

Now the result p value is 2.5

Structure of 'c' program :-

```
Comment line section
Link Section
Global variable declaration
main()
{
  declaration part
  executable part
}
Sub program
{
  function1()
  {
  ...
  }
  function2()
  {
  ....
  }
  .....
  .....
  function n()
  {
  }
}
```

After writing the program we should give the following steps.

* Save : To save the program we use F_2 functi button.

* Compile : To save the program we use Alt + C o Alt + F9

* Run : To execute the program we use Alt + R Control + F9

* verify the output : To verify the output we use Alt + F5

* D

Decision Control Statements

1. The Decision control statement sometimes is called as the decision making statement.
2. In the decision control statement depend upon the condition a single statement or group of statements are executed
3. The following are the decision control statements
   (i) If statement
   (ii) If else statement
   (iii) Nested ifelse statement (if-else-if ladder)
   (iv) Switch case statement

(i) **If Statement :**

In If statement, if the condition is true the if block statements are executed.

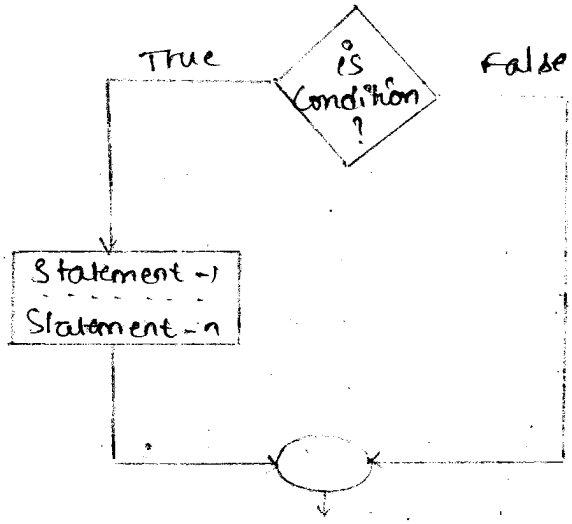
2. If the condition is false the if block statement are not executed

Syntax:
```
If (condition)
{
  Statement-1;
  Statement-2;
  ---------
  ---------
  Statement-n;
}
```

Eg: This Example is for the given number is checking Negative number

```
if (n<0)
{
  Printf("The Number is Negative");
}
```

Flowchart :



(ii) If Else Statement :

1. In the If Else statement if the condition is true the if block Statements are Executed
2. The condition is false the else block Statements are executed.

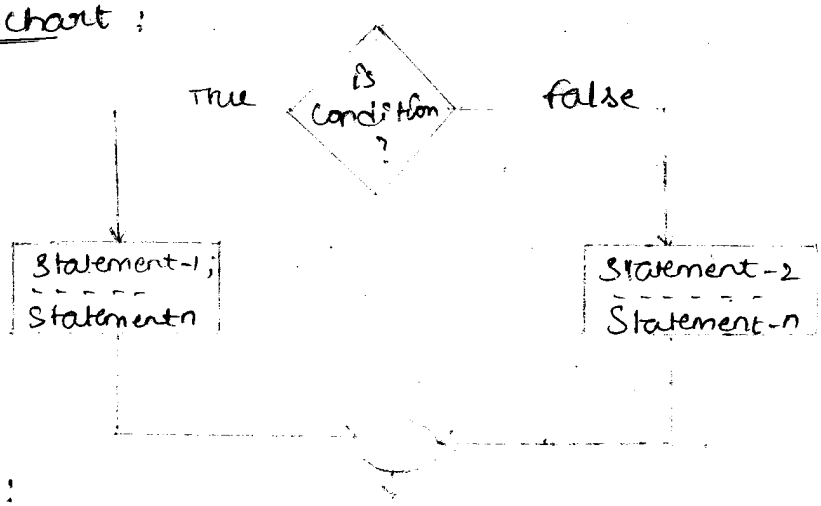
Syntax:

```
if (condition)
{
  Statement -1;
  ---------
  Statement -n;
}
else
{
  Statement -1;
  ---------
  Statement -n;
}
```

Eg: This example is for the given number is checking Negative or positive number.

```
if (n<0)
{printf("number is negative");
}
else
{printf("number is positive");
}
```

Flowchart :



NOTE :

After the If Statement and Else Statement we don't give Semicolon (;).

(iii) Nested If Else Statement :

1. The Nested If Else the number of condition are checked where the condition is true that corresponding block statements are executed
2. If all conditions fail or false then the final else block statements are executed

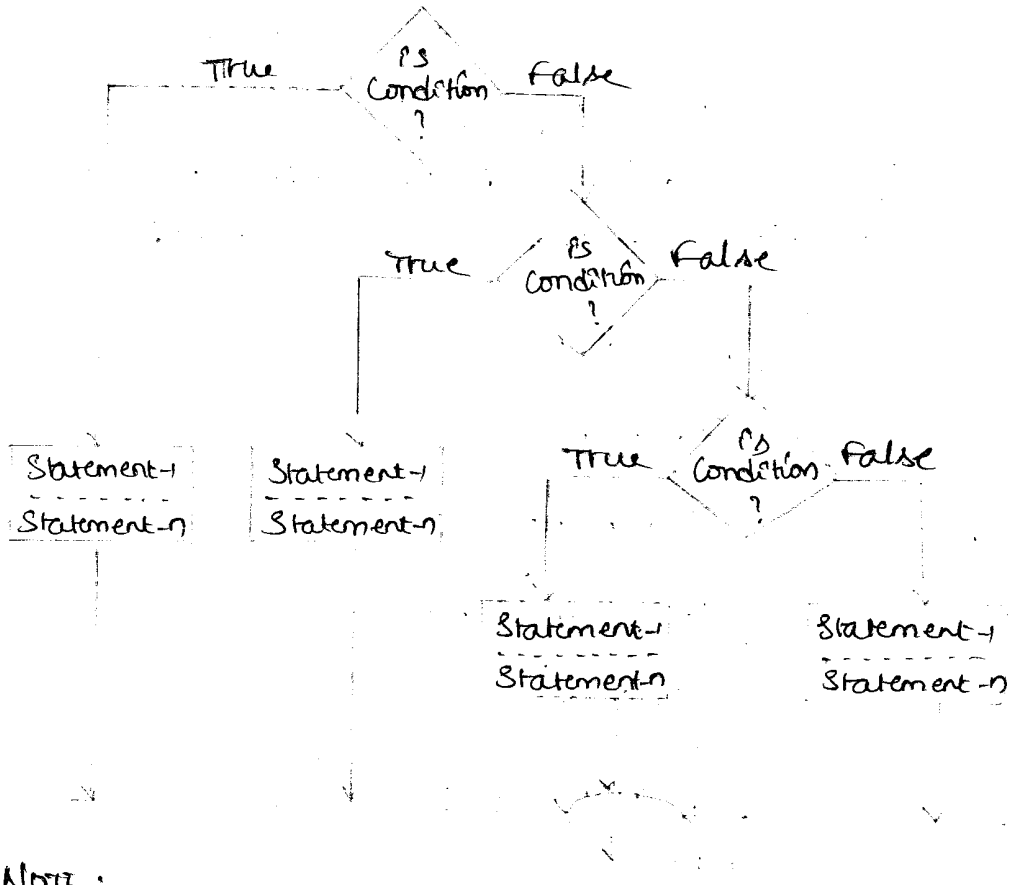
Syntax :

```
if (condition)
{
  Statement-1;
  ---------
  Statement-n;
}
else if (condition)
{
  Statement-1;
  ---------
  Statement-n;
}
else
{
  Statement-1;
  ---------
  Statement-n;
}
```

Eg: This example is for finding the biggest of 3 Numbers.

```
if(a>b && a>c)
{
 Printf("a is big");
}
else if (b>a && b>c)
{
 Printf("b is big");
}
 else
{
 Printf("c is big");
 }
```

Flowchat :



Note :

1. We should maintain the gap in between Else if
2. A If Statement Contains another if Statement

(or) a if Statement can be used within another if Statement is called as Nested if

Syntax:

```
if (condition)
{
  Statement-1
  Statement-n
}
  if (Condition)
  {
    Statement-1
    - - - - -
    Statement-n
  }
}
```

(iv) Switch Case Statement:

1. It is used to Select a Single Statement or a group of statement among several alternatives or choices.
2. In this the variable or expression result is compared with the constant and where it is matching the following case statements are executed.
3. If the variable or expression result is not Matching with anyone of the constant then the default statements are executed.
4. In place of constant we can take integer or character but not float
5. In Switch statement we can use the break statement it is breaking the switch block execution.

6. If we don't use break statement in that time variable or Expression result is matching with from the constant to default all statements are executed

Syntax:

```
Switch (variable/Expression)
{
 case Const 1 :
        Statement-1;
        ------
        Statement-n;
        break;
 case Const 2 :
        Statement-1;
        ---------
        Statement-n;
        break;
 default :
        Statement-1;
        ---------
        Statement-n;
}
```

Eg:

```
Switch(p)
{
 case 1 :
        c = a+b;
        Printf("addition = %d", c);
        break;
 case 2 :
        c = a-b;
        Printf("Substraction = %d", c);
        break;
 default :
        Printf("Invalid choice");
}
```

Nested Switch :

One Switch Statement used within another Switch Statement (or) one Switch block is used within another switch block is called Nested Switch.

Syntax :

```
Switch (variable / Expression)
{
case constant :
Switch (variable / Expression)
{
 case const1 :
      Statement -1;
      - - - - - - -
      Statement -n;
      break;
 case const2 :
      Statement -2;
      - - - - - - -
      Statement -n;
      break;
  - - - - - - - - - -
 default
      Statement -1;
      - - - - - - -
      Statement -2;
} ---------- /* Inner Switch closing */
case constant2 :
     Statement -1;
     - - - - - - - - -
      Statement -n;
      break;
 default
     Statement -1;
     - - - - - - -
      Statement -n;
} ---------- /* Outer Switch closing */
```

~~The diff~~

The difference b/w Switch and Nested If Statements

Switch	Nested If
1. We don't take the same Constant value to two or more cases (i.e., two or more cases don't have the same Constant value)	1. The same condition may be Repeated Number of times.
2. We can use the Nested If Statement in Switch block.	2. We can use the Switch Statement in Nested If block.

Loop Control Statements :—

1. It is also called as Iterative Statements.
2. A Single Statement or a group of Statements are executed for Certain number of times untill the condition is true or (untill the Condition is true a Single Statement or group of Statements are executed repeatedly
3. If the condition is false the loop is terminated or Stop
4. There are 3 loop Control Statements
   (i) For loop
   (ii) While loop.
   (iii) Do While loop

(i) For Loop :

1. A Single Statement or a group of Statements

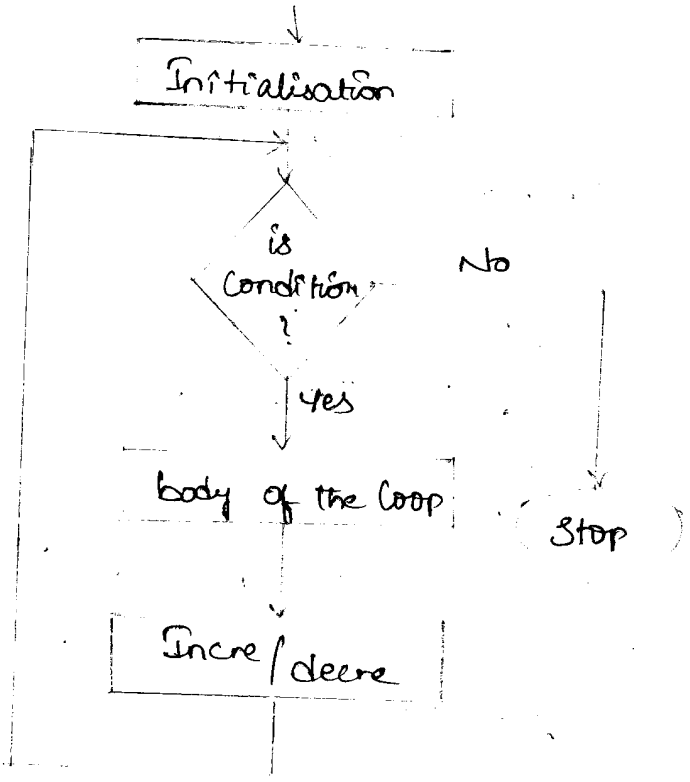
are executed for certain Number of times until the condition true or untill the condition is true a single statement or group of statements are executed

2. If the condition is false the loop is terminated or stop

Syntax:

```
for ( Initialisation ; test condition ; Incre/decre)
{
    Statement -1;
    .....
    .....
    Statement -n;
}
```

Flowchat ;



Eg: This Example is for display from 1 to 5 Natural Numbers

```
for(i=1; i<=5; i++)
{
  Printf("%d", i);
}
```

2) This Example is for display from 5 to 1 Natural number.

```
for (i=5; i>=1; i--)
{
  Printf("%d", i);
}
```

NOTE:

The 1st example is called as increment loop, the 2nd example is called as decrement loop

Other Examples:

1. We can write initialisation outside of the for statement

```
i=1;
for( ; i<=5; i++)
{
  Printf("%d", i);
}
```

2. We can write the for loop without test condition.

```
for (i=1;    ; i++)
{
  Printf("%d", i);
}
```

This loop is also treated as Infinite loop. To break the infinite loop we should write the

following

3.
```
for(i=1;    ; i++)
{
  if (i==6)
    break;
  else
    printf("%d", i);
}
```

4. We can write the incre/decre in the body of the loop or inside of the loop.

```
for(i=1; i<=5;   );
{
 printf("%d", i);
 i++;
}
```

5. If we omit or leave initialisation, test condition, incre/decre that loop is also treated as infinite loop

```
for ( ; ; )
{
  .....
  .....
  .....
}
```

6. We can write many counter variable initialisation and incre/decre in the same for statement with the help of comma operation

```
for (i=1, j=5; i<=5; i++, j--)
{
  printf("%d %d", i, j);
}
```

NOTE

1. We don't write Semicolon (;) after the for statement.
2. The intial or starting time the condition is failure then the loop is not executed at least a single time

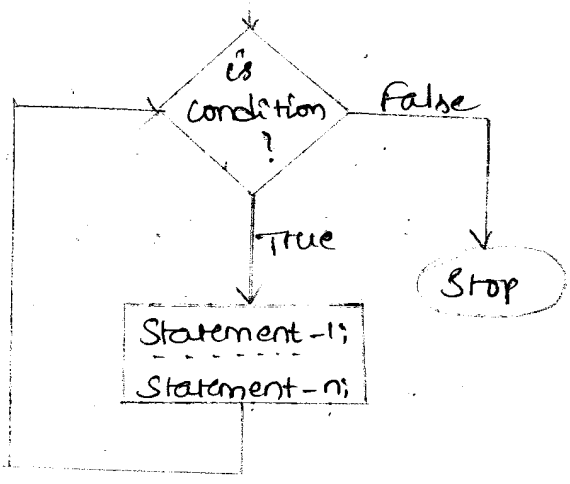
(iii) While Statement :-

1. A single Statement or group of Statements can executed repeatedly for certain number of times untill the condition is true or untill the condition is true the single or group of Statements are executed
2. If the condition is false the loop will be terminated or stop

Syntax:

```
While( test condition )
{
    Statement-1;
    - - - - - -
    Statement-n;
}
```

Flow chart:



Eg: Display from 1 to 5 Natural numbers

```
while (i<=5)
{
  Printf("%d", i);
  i++;
}
```

2. It display from 5 to 1 Natural numbers.

```
while (i>=1)
{
  Printf("%d", i);
  i--;
}
```

Other Eqs:

1.
```
while (true)
{
  Statement-1;
  .....
  Statement-n;
}
```

2.
```
while (1)
{
  Statement-1;
  .....
  Statement-n;
}
```

The above 2 loops are infinite loops.

(iii) do while Statement;

1. Without checking any test condition the body of the loop is executed atleast a single time
2. After Executing the single time the condition is check if the condition is true the loop is reexecuted. This process is continues untill the condition is true.
3. If the condition is false the loop will be

terminated or Stop

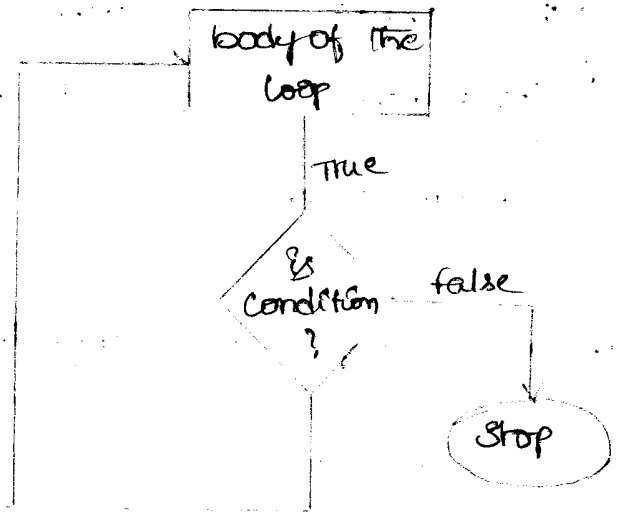
4. In this the condition will be checked at the end of the loop

5. Here the while statement contains semicolon (;)

Syntax:

```
do
{
  Statement_1;
  ---------
  Statement_n;
}
while (test condition);
```

Flowchat:



Eg: 1. Display 1 to 5 Natural numbers

```
i=1;
do
{
Printf("%d", i);
i++;
}
while (i<=5);
```

2. Display 5 to 1 Natural Numbers

```
i=5;
do
{
Printf("%d", i);
i--;
}
while (i>=1);
```

Other Eqs:

1.
```
do
{
Statement-1;
.....
Statement-n;
}
while(True);
```

2.
```
do
{
Statement-1;
.....
Statement-n;
}
while(1);
```

The above two loops are infinite loops.

Difference b/w while and dowhile:

while	dowhile
1. Depend upon the Condition only the loop is executed.	1. Without depend upon the Condition the loop is executed atleast Single time
2. Here the condition will be check at the Starting of the loop	2. Here the condition will be checked at the end of the loop
3. While Statement must not Contain SemiColon	3. While Statement must Condi! Contain SemiColon

Break :

1. It is used to terminate or stop the loop or switch statement
2. It is used with any decision control statement.
3. When the Break statement is executed the control goes to the first statement after the loop.

Syntax : Break ;

Eg: This example is for 1 to 5 Natural numbers

```
for(i=1; i<=6; i++)
{
  if (i==6)
  break;
  else
    Printf("%d", i);
}
Printf("After the loop");
```

control goes to

Continue :

1. It is quite opposite of Break Statement.
2. It is used with any decision control statement.
3. When the statement is executed the control goes to begining of the loop. (The control goes to the incre/decre part)

Syntax: Continue;

Eg: This Example is for display below 10 Even Numbers

```
For (i=1; i<10; i++)
{
  if (i%2==1)
    Continue;
  else
    Printf("%d", i);
}
```

Control goes to

NOTE:

The Break and continue statements are also called as conditional statements because the decision statements are Needed to these.

GOTO:

1. This is unconditional Statement (i.e., Generally it is no need of any condition)
2. When the GOTO Statement is executed the Control goes to corresponding label.

Syntax: goto label;

```
goto label;
Statement-1;
.....
Statement-n;
Label; <-----
Statements;
```

Forward Jump

```
Label1: <-----
Statement-1;
.....
Statement-n;
goto label;
Statements;
```

Backward Jump

Eg: This example is for display the given number is Even or odd

```
if (n%.2 == 0)
  Printf("%d", i);
else
  goto add;
  Odd;
  Printf("The number is odd");
```

NOTE :

Goto is purely an unconditional statement

UNIT - 2

Arrays

Array :-

Array is Nothing but a group of similar data element can be shared a common name. The group of similar data elements means all are Integers or all are Float or all are characters.

* The similar data elements are stored in continuous memory location.
* Array is classified into two types
  1. Single dimensional array
  2. Multi dimensional array

1. Single dimensional array :-

In single dimensional array a group of similar data elements can be used in a single variable name and using single subscript such a variable (or) array is called as "single dimensional array"

Declaration of single dimensional array :

Syntax : datatype variable name [size];

Eg : int a[5];
float b[10];

Here Size is indicates maximum how many Numbers can be stored in array.

Initialisation of Single dimensional array and how to store the data element in this array :

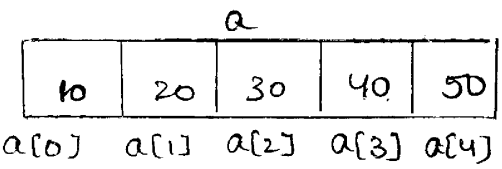
A Single dimensional array initialisation is just like as org ordinary variable initialisation

Eg: int a[5] = {10, 20, 30, 40, 50};

(or)

int a[] = {10, 20, 30, 40, 50};

Example of How to Store in memory :



How to access Single dimensional array data elements

To access Single dimensional array each and every data element we use array name is followed by a Square bracket (or) Subscript enclosed with in integer, number. That number indicates array index.

Eg:

a[0] → It access 0th position data element i.e. 10

a[1] → It access 1st Position data element. i.e. 20

a[2] → It access 2nd Position data element i.e. 30

a[3] → It access 3rd Position data element i.e. 40

How to Read and print Single dimensional array data elements :

For Reading and printing Purpose in Single dimensional array only one for loop

is taken

Eg: For Reading

```
int a[5], i;
for (i=0; i<5; i++)
scanf("%d", &a[i]);
```

For printing

```
int a[5], i;
For (i=0; i<5; i++)
Printf ("%d", a[i]);
```

2. Multi dimensional array :-

Multi dimensional array is Nothing but a group of similar data elements can be given a single variable name using Multiple Subscripts is called as Multi-dimensional array.

Eg: 2D, 3D

(i) Two dimensional array :-

Two dimensional array is Nothing but a group of similar data elements can be given a single variable name using two subscripts is called two dimensional array.

Declaration of two dimensional array :

Syntax: datatype variablename [row size][Column Size];

Eg: int a[3][2];

float b[2][2];

Here Row size indicates Number of rows, Column size indicates Number of Columns.

Initialisation of two dimensional array and how to store data Elements in this array :

Eg for Initialisation ;

```
int a[3][3] = { 1,2,3,4,5,6,7,8,9 };
```

(or)

```
int a[3][3] = { 1,2,3,
                4,5,6,
                7,8,9
              };
```

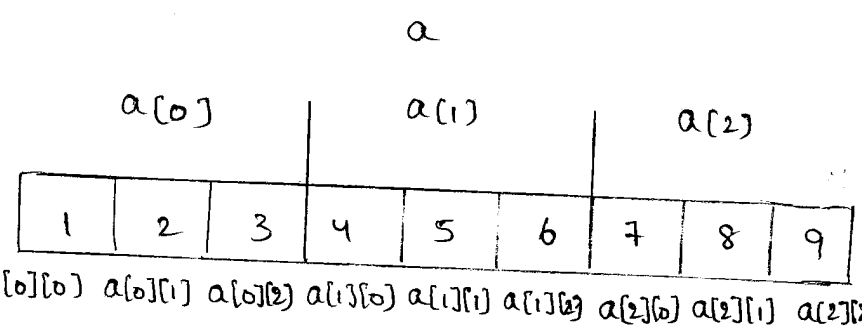
(or)

```
int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

(or)

```
int a[3][3] = {
                {1,2,3},
                {4,5,6},
                {7,8,9},
              };
```

Eg of how to store in memory ;



How to access two dimensional array data elements

To access two dimensional array each and every data element we use array name is followed by two subscripts enclosed within two integer Numbers. These Numbers indicate first one row index, second one column inde

Eg: a[0][0] → It refers or access 0th row & 0th column (i.e., 1)

a[0][1] → It refers or access 0th row and 1st column i.e., 2

a[0][2] → It refers or access 0st row and 2st column Element i.e., 3

a[1][1] → It refers or access 1st row and 1st column Element i.e., 5

How to Read and print two dimensional data Elements :

For Reading and printing purpose in two dimensional array we need two loops. (i.e. first loop indicates the row, second loop indicates the column)

Eg: For Reading

```
int a[3][3], i,j;
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
scanf("%d", &a[i][j]);
}
}
```

For printing

```
int a[3][3], i,j;
For (i=0; i<3; i++)
{
For(j=0; j<3; j++)
{
Printf("%d", a[i][j]);
}
}
```

(iii) Three Dimensional array :-

Three dimensional array is Nothing but a group of Similar data Elements can be given a Single variable name using three Subscripts such a array is called as Three

dimensional array.

Declaration of three dimensional array :

Syntax:

Datatype variable name [size1] [size2] [size3];

```
int a[3][3][3];
Float b[2][2][2];
```

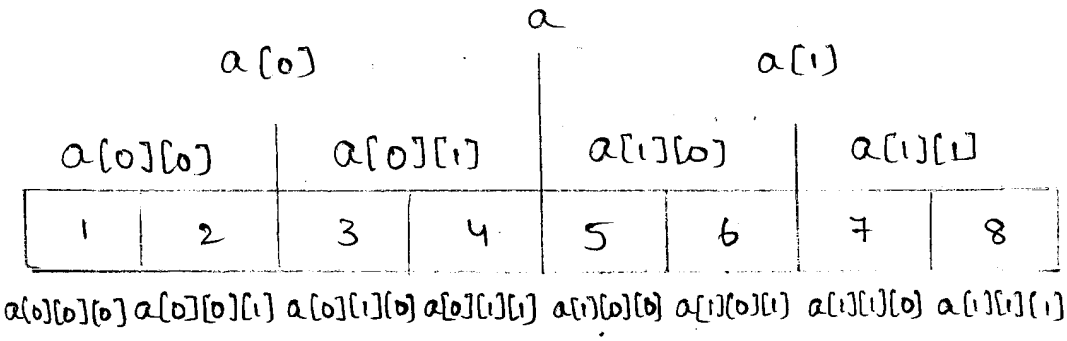
Here $Size_1$ indicates how many matrices or Number of matrices, $Size_2$ indicates Number of rows of each matrix, $Size_3$ indicates Number of columns of each matrix.

Initialization of three dimensional array and how to store the data in 3D array:

Eg for Initialisation :

```
int a[2][2][2] = {
                    {
                     {1,2},
                     {3,4}
                    },
                    {
                      {5,6},
                       {7,8}
                     }
                    }
```

Eg of how to store in memory :



How to access 3D array data elements:

To access 3D array each and every data Element to use array name is followed by three subscripts enclosed within three integer Numbers. 1st Number indicates Matrix, 2nd number indicates row, 3rd Number indicates column.

Eg:

a[0][0][0] → It refers or access 0th matrix, 0th row, 0th column Element ie., 1

a[0][0][1] → It refers or access 0th matrix, 0th row, 1st column Element ie., 2

a[1][1][1] → It refers or access 1st matrix, 1st row, 1st column Element ie., 8 etc.,

How to read and print 3D array data Elements:

For Reading and printing purpose in 3D array we need three loops. ie., First loop indicates Matrix, Second loop indicates row, third loop indicates column.

Eg For Reading:

```
int a[2][2][2], i, j, k;
for (i=0; i<2; i++)
{
  for (j=0; j<2; j++)
  {
    for (k=0; k<2; k++)
    {
      scanf("%d", &a[i][j][k]);
    }
  }
}
```

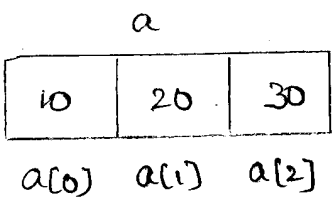
Eg For Printing :

```
int a[2][2][2], i, j, k;
for (i=0; i<2; i++)
{
  for(j=0; j<2; j++)
  {
    for(k=0; k<2; k++)
    {
      printf("%d", a[i][j][k]);
    }
  }
}
```

Characteristics of arrays :-

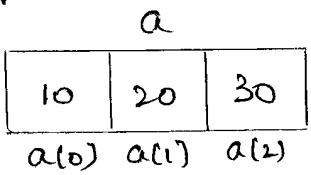
* Instead of taking five variables to five values or data Elements or Numbers of data types, it is bigger to take an array that may be an integer array (or) float-array (or) character array.
* In array all similar data Elements can be shared a common name.

Eg:

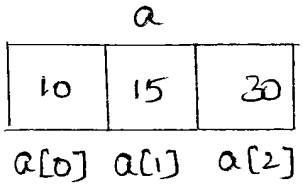


* In array an array index plays major roles
* In array we can change any data element without distributing to other data Elements.

Eg: Suppose the array contains that data Elements like



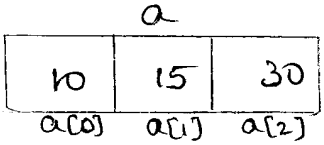
Suppose we want to assign a[1]=15 then the array is like below.



* We can assign the array data Element to an ordinary variable or array variable of same data type.

Eg: int a[5], b[5], c;

Suppose the array contains data Elements like Below.



The a[1] data Element can be assigned like Below.

```
c=a[1];
  (or)
a[2]=a[1];
  (or)  b[0]=a[1]
```

* You can calculate the number of bytes occupied by an array using the following formula.

Total bytes = datatype * size

Eg: int a[5];

Total bytes = 2*5

= 10 bytes

Strings

String :-

String is Nothing but a group of (collection) characters is terminated or closed with (\0) (Null character). Here Null character also occupied one byte

Declaration of string :-

Syntax: datatype string variable (or) name [size];

Eg: char str[6];

Initilise and how to store string in memory:

The string Initilisation is also done just like as ordinary variable initialisation.

Eg :-

Model -1 : character by character Initialisation.

```
char str[6] = {'I','N','D','I','A','\0'}
```

(or)

```
char str[] = {'I','N','D','I','A','\0'}
```

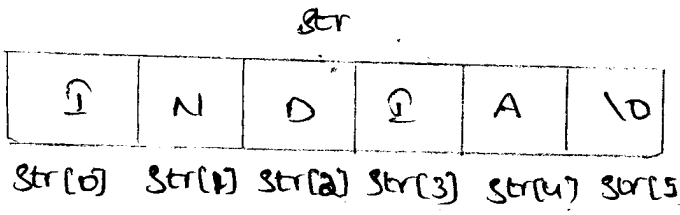
Model-2 : At a time the entire string can be initialisation.

```
char str[6] = "INDIA"
```

(or)

```
char str[] = "INDIA"
```

Example of how to store in memory :



How to access characters from string :

TO access character from string (or any character from string) we use a string name is followed by single subscript enclosed with an integer number. The Number indicates array index.

Eg : str[0] → It refers or access 0th character i.e., I

str[1] → It access 1st character i.e., N

⋮

str[4] → It access 4th character i.e., A

Reading and printing :

Model-1 : If we want to read and print character by character we need to take one loop.

Eg :
```
char str[6];
int i;
For(i=0; i<6; i++)
Scanf("%c", &str[i]);
```

Eg of printing :
```
char str[6];
int i;
For(i=0; i<6; i++)
Printf("%c", str[i]);
```

Model-2 : Here we take formatted I/o functions scanf(), printf() with formatted specifier %s

Eg of Reading :

```
Char Str[6];
Scanf("%s", Str);
```

Eg of printing

```
Char Str[6];
Printf("%s", Str);
```

Model-3 : Here we are taking unformatted I/o functions getS(), putS()

Eg of Reading

```
Char Str[6];
getS(Str);
```

```
Char Str[6];
PutS(Str);
```

Note :

String is sometimes called as a single dimensional character array.

Array of String or 2D character array :-

A collation of (or) group of strings is called as array of strings. Here first subscript indicates Number of strings and second subscript indicates Number of characters of each string.

Declaration of array of string :

Syntax :

Datatype String name [no. of strings] [each string no. of characters];

Eg: Char Str[5][10];

The above statement indicates '5'

Strings and each string is '10' characters.

Initialization and how to store in memory:

Array of strings initialization is also done just like as ordinary variable initialization.

Eg: char Str[5][10] = { "Sri",
"Sai",
"Rama",
"Raju",
"Krishna kumar"
}

Example of store in memory:

Str

Sri \0	Sai \0	Rama \0	Raju \0	Krishna kumar \0
Str[0]	Str[1]	Str[2]	Str[3]	Str[4]

(or)

The above is also reffered as

	0	1	2	3	4	5	6	7	8	9
Str[0]	S	r	i	\0						
Str[1]	S	a	i	\0						
Str[2]	R	a	m	a	\0					
Str[3]	R	a	j	u	\0					
Str[4]	K	r	i	s	h	n	a	k	u	m	a	r	\0

How to access:

To access the individual string from

array. We use String name is followed by a single subscript enclosed within an integer number is reffered as array index

Eg: Str[0] → It refers or access 0th String (i.e. Sri)

Str[1] → It refers or access 1st String (i.e. Sai)

⋮

Str[4] → It refers or access 4th String (i.e. Kiran Kumar)

Reading and printing:

For reading and printing we need a single loop.

Example of reading:

Model-1:

```
char str[5][10];
int i;
for(i=0; i<5; i++)
scanf("%s",&str[i]);
```

Model-2:

```
char str[5][10];
int i;
for(i=0; i<5; i++);
gets(str[i]);
```

Example of printing:

Model-1:

```
char str[5][10];
int i;
for(i=0; i<5; i++)
  printf("%s", str[i]);
```

Model-2:
```
char str[5][10];
int i;
for(i=0; i<5; i++)
puts(str[i]);
```

* String Handling Functions (or) String Library Functions

These functions sometimes is called as String predefined functions. We must include <string.h> header file to these functions.

1. Strlen():

* This function is String length
* This function counts Number of characters of the given string excluding NULL character (\0).

Syntax: n = strlen(string)

Here String may be either String variable (or) any String constant.

Eg ① char str1[10] = "Hello";
int n;
n = strlen(str1);

② int n
n = strlen("Hello");

The above function counts Number of characters i.e 5 and will be stored in to n variable.

2. Strcpy():

* This function is String copy.
* This function copies one string contents into another string i.e. one string is

copied into another string

Syntax: strcpy (Str1, Str2)

Here Str2 may be either string variable or string constant.

Eg: ① char Str1[10];
char Str2[10] = "Hello";
strcpy (Str1, Str2);

② char Str1[10]
strcpy (Str, "Hello");

The above function copies Str2 into Str1 i.e., Both Str1 and Str2 contains same string

3. Strcat():

* This function is string concatination
* This function concatinates or joins two strings.

Syntax: Strcat (Str1, Str2).

The Str2 can be joined at the end of the Str1. Here Str2 may be either string variable or constant

Eg: ① int n;
char Str1[10] = "our";
char Str2[10] = "India";
n = Strcat (Str1, Str2);

② char Str1[10] = "our";
Strcat (Str1, "India");

According to the above function Str1 is "OurIndia" Str2 is India

4. StrCmp() :

* This Function is String comparision.
* This Function compares the given two strings

Syntax: n = StrCmp (Str1, Str2);

This Function follows case sensitive (ie., upper case and lower case will be treated as differentiate)

→ If Str1 and Str2 both are same it return '0'

→ If Str1 is less than St2 it returns Negative value

→ If Str1 is greater than Str2 it returns Positive value.

Eg: ① Char Str1[10] = "Their"
Char Str2[10] = "There"
int n;
n = Strcmp (Str1, Str2);

② int n;
n = Strcmp ("Their", "There");

According to the above function it returns negative value so Str1 < Str2

5. StriCmp() :

* This Function is String ignore comparision
* This Function also compares the given two Strings but should not follows case Sensitive. (ie., Upper case and Lower case both are Same).

Syntax: n = StriCmp (Str1, Str2);

Eg: ① char str1[10] = "RAMU";
char str2[10] = "ramu";
int n;
n = strcmp (str1, str2);

int n;
n = strcmp ("RAMU", "ramu");

According to the above value both strings are same so it returns zero.

6. Strrev() :

* This Function is String reverse
* This function reverse the given string

Syntax: Strrev(String);

Eg: char str1[10] = "INDIA";
Strrev(str1);

According to the above Function str1 is "AIDNI"

7. Strlwr() :

* This function is string lower
This function converts or changes the given upper case into lower case.

Syntax: Strlwr(String);

char str1[10] = "INDIA";
Strlwr(str1);

According to the above function str1 is "India"

8. Strupr() :

* This Function is String upper.

* This function converts or changes the given lower case string into upper case.

Syntax: strupr (string);

Eg: char str1[10] = "india";
strupr (str1);

According to the above function, str1 is "INDIA".

9. sscanf() and sprintf():

sscanf():

This function reads one string and writes into another string.

Syntax: sscanf (str1, format specifier, str2);

Eg: char in[] = "hello";
char out[10];
sscanf(in, "%s", out);

sprintf():

This function writes any data type or all datatypes into the string or onto the string.

Syntax: sprintf (string, format specifier, list of variables);

int a = 3;
char ch = a;
float f =
char ~~out~~ in[10];

Display the String with different Formates :-

there two Formates are there

1. Width Specifier
2. pression Specifier

1. Width Specifier ;

Syntax: %.ws

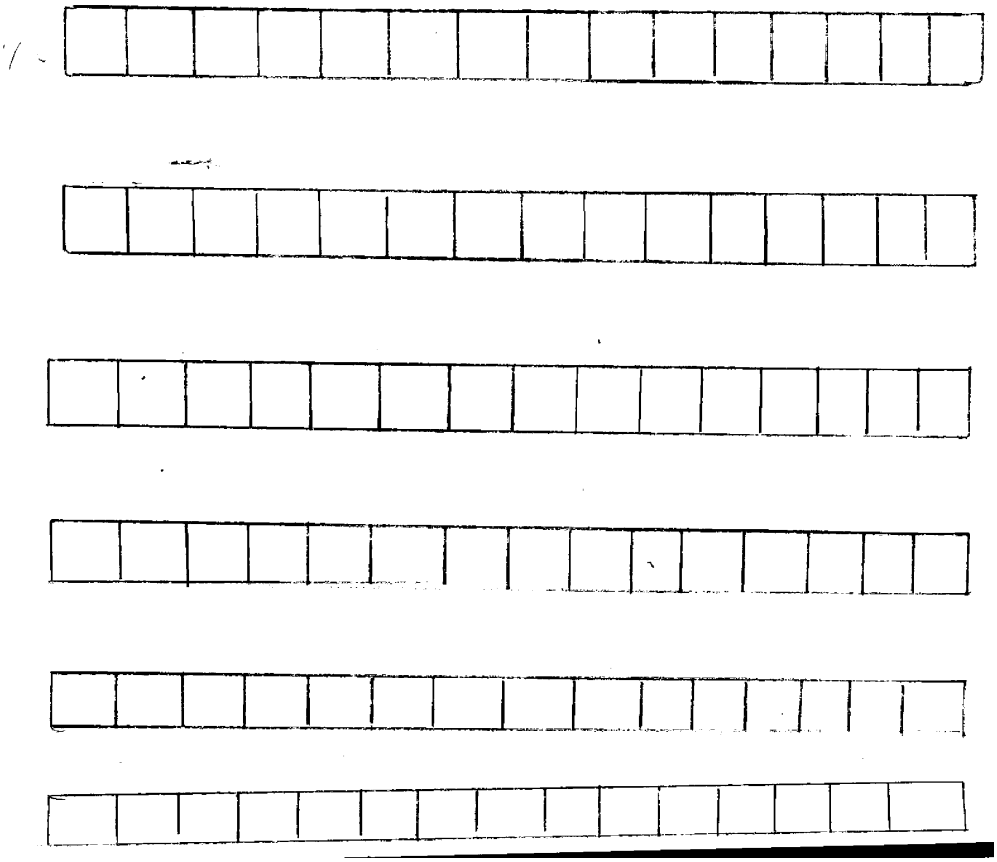
w indicates the width it is either Negative or positive

2. pression Specifier ;

Syntax: %.wps

w indicates width and p indicates pression depends on the p, first characters will be displayed

char str[] = "NEW DELHI";



Functions

Function Definition :-

* Function is a Sub program of one or more statements
* It Performs a special task when it calls

Advantages :-

1. When we write a function it avoids re-writing the same code over and over (many time).
2. The large program or problem is divided into smaller programs (or) problems. It is easy for finding Errors.
3. Generally the Function is classified into two types.
    1. Library Function
    2. user defined functions

1) Library Functions :

It is also called as predefined functions

Eg: 1) Mathematical Functions : Sqrt(), Pow() etc.

2) String Functions : Strcmp(), Strlen() etc.

2) User defined Functions :

These functions are defined by the user

Eg: main()

User defined Function Declaration and definition :

The user defined function declaration must be done in main function and

Function calling also must be done in main function.

The function definition must be done are return after main() function closing

Syntax:

```
main()
{
  returntype functionname(datatype1 variable1,
                datatype2 variable2....) /* Function Declaration */
  ..........
  ..........
  Variable = Function name(value1, value2.....)   /* Function calling */
  ..........
  ..........
}
returntype functionname(datatype1 variable1.....)   /* Function definition (or) Called Function */
{
  local variables;
  Statement-1;
  .......
  Statement-n;
  return (result data);
}
```

actual arguments

Formal arguments

Control goes to

Function Body

Function Declaration:

It tells three things

1) What is the return type (i.e., what data type it must be return)
2) What is the function name.
3) How many arguments and the corresponding datatypes.

Function calling (or) Calling function ;

1) It contains the function name following with arguments.
2) These arguments are called actual arguments.
3) After executing the statement the control goes to function definition

Function definition (or) called function :

1) It is similar to function declaration.
2) These arguments are called formal arguments.
3) It contains local variables and set of statements.

Return type :

1) It is optional
2) After executing Return statements the control goes back to function call or calling function with resulted data or without resulted data.

Types of functions :-

According to the arguments and return type the function was classified into 4 types.

1) Function with no arguments and no return type.
2) Function with arguments and no return type.
3) Function with no arguments and with return type.
4) Function with arguments and with return type.

1) Functions with no arguments and no return type:

The data is not passed from calling function and the resultant data will not be sent back from called function.

Syntax:

```
main()
{
  void functionname();
  .....
  ....
  Function name()
  .....
  .....
}
void function name()
{
  .....
  .....
}
```

Eg: This Example is for addition of two numbers

```
main()
{
  void add();
  clrscr();
  add();
}
void add()
{
  int a,b,c;
  Printf("Enter a,b,c values");
  Scanf("%d %d", &a, &b);
  c = a+b;
  Printf("addition = %d", c);
}
```

2) Function with arguments and no return type:

The data is passed from calling function and the resulted data will not be sent

back from Called Function.

Syntax:

```
main()
{
  void Function name (datatype, Variable, ....);
  .....
  .....
  function name (value1, value2 ... );
  .....
}
void Function name (datatype, Variable, ....)
{
  .....
  .....
}
```

Eg: This Example is for addition of two Numbers.

```
main()
{
  void add (int, int);
  int a,b;
  clrscr();
  Printf(" Enter a,b values");
  Scanf("%d %d", &a, &b);
  add (a,b);
}
void add(int p, int q)
{
  int r;
  r= p+q;
  Printf(" addition = %d", r);
}
```

3) Function with no argument and with return type:

The data is not passed from calling

function and the resultant data will be sen back from called function.

3) Function with no arguments and with returntype

Syntax:

```
Main()
{
returntype functionname();
.....
.....
variable = functionname();
.....
}
returntype functionname()
{
.....
.....
return(result data)
}
```

Eg: This example is for addition of two numbers

```
Main()
{
int add();
int c;
clrscr();
c=add();
Printf("addition = %d", c);
}
int add()
{
int a,b,r;
Printf("Enter a,b values \n");
Scanf("%d", &a,&b);
r = a+b;
return(r);
}
```

4) Functions with arguments and with return type;

The data is passed from calling function and the resulted data will be send back from called function

Syntax:

```
Main()
{
returntype functionname(datatype1, variable1,...);
.....
.....
Variable = functionname (value1, value2 ...);
.....
.....
}
returntype functionname(datatype1, variable1,...);
{
.....
.....
return(result data);
}
```

Eg: The example is for addition of two numbers

```
Main()
{
int add (int, int);
int a,b,c;
clrscr();
Printf("Enter a,b values \n");
Scanf("%d%d", &a, &b);
c= add (a,b);
Printf ("addition = %d", c);
}
int add (int p, int q)
{
return (p+q);
}
```

Call by value and Call by reference :-

Call by Value :

* In call by value the actual arguments zerox will be copied into formal arguments
* Even though we are performing changes in formal arguments that will not be effected on actual arguments

Eg: This Example is for swaping of two numbers

```
Main()
{
  int a,b;
  void swap(int,int);
  clrscr();
  Printf("Eneter ab values");
  Scanf("%d %d", &a, &b);
  Printf("Before swaping a=%d b=%d", a, b);
  Swap(a,b);
  Printf("After swapping a=%d b=%d", a,b);
  getch();
}
void swap(int p, int q)
{
  int temp;
  temp = p;
  P=q;
  q = temp;
}
```

Call by Reference :

* In call by reference the actual arguments addresses will be copied into formal arguments

* We are performing changes in formal arguments that will be effected on actual arguments.

Eg: This Example is for swapping of two Numbers

```
Main()
{
  int a,b;
  void swap(int *, int *);
  clrscr();
  Printf("Enter a,b values");
  Scanf("%d %d", &a, &b);
  Printf("Before swapping a=%d b=%d", a,b);
  swap(&a, &b);
  Printf("After swapping a=%d b=%d", a,b);
  getch();
}
void swap(int *p, int *q)
{
  int temp;
  temp = *p;
  *p = *q;
  *q = temp;
}
```

Functions as arguments :-

Not only passing the values and addresses we can also pass function (or) functions as a arguments.

Eg: This Example is for Perform square and cube of given number

```
Main()
{
  int n;
```

```
int sqr(int);
void cube(int,int);
clrscr();
Printf("Enter n value \n");
Scanf("%d", &n);
Cube(n, Sqr(n));
getch();
}
int Sqr(int p)
{
return (p*p)
}
void cube (int p, int q)
{
int s
s = p*q;
Printf("cube = %d", s);
}
```

decision making:

Functions and looping statements :—

We can use the decision statements in functions (or) in decision statements we can use functions

This Example is for finding the biggest of two numbers

```
main()
{
int a,b;
void big (int ,int);
clrscr();
Printf ("Enter a,b values");
Scanf("%d %d", &a, &b);
big(a,b);
getch();
}
```

```
void big (int p, int q)
{
  if (p>q)
    Printf ("%d is big", p);
  else
    Printf ("%d is big", q);
}
```

Functions and Looping Statements:

We can use the looping statements in functions (or) in looping statements we can use functions.

This Example is for finding the factorial of the given number.

```
Main()
{
  int n,f;
  int fact(int);
  clrscr();
  Printf("Enter n values");
  Scanf("%d", &n);
  f = fact(n);
  Printf("factorial = %d", f);
  getch();
}
  int fact(int n)
  {
    int i, f=1;
    for (i=1; i<=n; i++)
      f = f*i;
    return (f);
  }
```

Pointers to functions (or) function pointer :-

* We can assign the address of any function to function pointer or pointers to function.
* Here we call the original function with function pointer
* The function pointer is declared according to the original function (i.e., It contains return type and arguments).

Eg: This Example is for printing a message with function pointer.

```
Main()
{
 int disp();
 void (*p)();
 clrscr();
 P= disp(); → /* The original class function
                 address will be assign into
                 function pointer */
 (*p)(); → /* The original function is
              calling through function pointer*/
 getch();
}
 void disp()
 {
  Printf("Welcome to OLR");
 }
```

Arrays with functions :-

* The arrays elements passed the function by specifying (or) giving array name in function call. In this the actual array

Argument address will be copied into formal arguments array. If we are performing the changes in formal argument array that will be effected on actual argument array.

Eg: Bubble sort

```
Main()
{
  int a[20], i, n;
  void bsort (int[], int);
  clrscr();
  Printf("How many numbers do you want \n");
  Scanf("%d", &n);
  Printf("Enter elements into array \n");
  for(i=0; i<n; i++)
     Scanf("%d", &a[i]);
     bsort (a,n);
  for(i=0; i<n; i++)
     Printf("%d", a[i]);
     getch();
}
void bsort (int b[20], int n)
{
  int i, j, temp;
  for( i=0; i<n-1; i++)
  {
    for (j=0; j<n-1; j++)
```

```
        {
            if (b[j]>b[j+1])
            {
                temp = b[j];
                b[j] = b[j+1];
                b[j+1] = temp;
            }
        }
    }
}
```

Recursion:-

The function itself calling until the condition is true is called recursion. It has two types

1. Direct Recursion
2. Indirect Recursion

1. Direct Recursion:

The function itself calling until the condition is true.

2. Indirect Recursion:

In this function 1 calling function 2 and the function 2 calling function 1

Eg: This example is for factorial of the given number.

```
main()
{
    int n,f;
    int fact(int);
    clrscr();
    Printf("Enter n values\n");
    Scanf("%d", &n);
    f = fact(n);
```

```
    Printf("factorial = %d", f);
    getch();
}
int fact(int n)
{
    int p;
    if(n==1)
        return(1);
    else
        p = n*fact(n-1);
        return(p);
}
```

Storage classes :—

The characteristics of the variable is called a Storage class. The Storage class define several characteristics.

1) Storage :

Where the variable is stored either memory or registered.

2) Default initial value :

Default initial value (what is the initial value, if the initial value is not assigned)

3) Scope :

How long a variable keeps the particular value.

4) Life time (or) Live :

How long a variable is valid.

There are four Storage classes are given below

1. Automatic or Auto Storage class
2. Register Storage class.

3. Static Storage class
4. External Storage class

1. Automatic or Auto Storage class:

1) If we use Automatic Storage class along with the variable is called auto variable
2) These variables are declared inside the function. When we call the function these variables are created, when we exit from the function these variables are destroyed.
(A variable is defined or declared without any storage class is called Auto variable).

The characteristics or features of auto variable.

Storage: In memory
Default Initial value: Garbage value
Scope: within the block.
Life time: within the block

Syntax:
```
main()
{
  auto int n;
  .....
  .....
}
```

Eg:
```
main()
{
  int a = 2, b = 3;
  {
    int c;
  }
  c = a+b; /* Invalid */
}
```

2) Register storage class :

1) If use Register Storage class along with the variable is called Register variable.

2) Generally the frequently accessable variable is declared as Register variable.

Eg : loop counter variables

3) These variables values are stored in Register

The characteristics or features of Register variables.

Storage → In Register

Default initial value → Garbage

Scope → with in the block

life time → with in the block

Eg :

```
main()
{
  Register int i;
  for(i=0; i<n; i++);
  Printf("%d", i);
}
```

3) Static Storage class :

1) If use Static Storage class along with the Variable is called Static variable.

2) These variables are declared Internal (inside the function) or (External (or outside the function)

3) The internal Static variables maintained data between different function calls.

The characteristics or features of Static variable.

Storage → In Memory

Default initial value → Zero

Scope → With in the Block

Life time → value of the variable maintained data between different function call

Eg:
```
main()
{
  void increment(); /* Function declaration */
  clrscr();
  increment();
  increment(); /* Function callig */
  increment();
}
void increment()
{
  static int x;
  Printf("%d", x++);
}
```

4) External storage class :

1) If use External Storage class along with the variable is called External variable.

2) These variables are sometimes called as global variables.

3) These variables valid until the end of the Program. So these variables can be accessed by any Function in the program.

The characteristics or features of External variable

Storage → In Memory

Default initial value → zero

Scope → Global

Life time → until the end of the program.

Eg: ①

```
int i;
main()
{
 void increment();
 void descrement();
 clrscr();
 i=2;
 Printf("%d",i)
  increment();
 } descrement();
 void increment()
 {
  Printf("%d", i++);
 }
 void descrement()
 {
   Printf("%d", i--);
  }
```

②

```
main()
{
 void Function();
  x=2;
  Printf("%d", x);
}
 int x;
 void Function1()
{
  Printf("%d", x++);
}
```

In the above program the main() Functic Cannot use x Variable because it has been declared after main() function. This problem can be solved by declare the variable in the above main or with storage class extern it is shown Below.

```
main()
{
void Function1();
extern int x=2;
Printf("%d", x);
}
int x;
void Function1()
{
Printf("%d", x++)
}
```

(or)

```
main()
{
void Function1();
Extern int x=2;
Printf("%d", x);
}
void Function1()
{
extern int x;
Printf("%d", x++);
}
int x;
```

Note :-

If we declare Extern variable the compiler

doesnot locate memory for these variables. because it is already allocated in another place where it is declared has global variables.

2. Multiple programs or source files can share variables for this we must declare global variables in one file and Explicity defined them with 'Extern' in other files.

Eg:

File1.c

```
int M;
main()
{
.....
}
```

File2.c

```
Extern int M;
function1()
{
.....
}
function2()
{
.....
}
```

Macro :-

Macro is a given piece of code or value. Macros can be defined with argument or without arguments.

Difference between the Macros and Functions.

Macro	Functions
1) Macro is Pre-processor	1) It is compiled
2) No datatype checking	2) Datatype checking is done.
3) Source code length increases or code length increase.	3) Code length remains same

4) Use of macros can read to Side effect	4) No Side Effects
5) Speed of Execution is faster.	5) Speed of Execution is slow.
6) Before compilation Macro name is replaced by Macro value.	6) During function call the control transfer to function definition
7) Small code appears many times	7) Large code appears many times
8) Macros doesnot check compile time Errors	8) Function checks compile time Errors

Pointers

Definition:-

A pointer is a variable which holds the address of another variable. There are two pointer operators

1. Address operator (&)
2. Indirectional operator (or)
   Differencing operator (or)
   Star operator (*)

The address operator gives address of the variable.

The Indirectional operator gives the value at address

Declaration and Initialisation:

The pointer variable declaration is as follows

Syntax: Datatype * variable;

Eg: int *p;
    char *ch;
    float *q;

The initialisation will be done in the same line (i.e., the declaration and initialisation both) or in different lines

① int x = 2;
   int *p = &x;

② int x=2, *p;
   *p = &x;

The following Examples are Invalid

① int *p = &i, i;

② float q;
int *p;
p = &q;

In the above Example ① before declare the i has been initialised.

In the Example ② The pointer variable and ordinary variable must required same datatype

The following operations can be performed with pointers.

1) Addition of two pointers.
2) Multiplication of two pointers or Multiplication any constant with pointer
3) Division of two pointers or division with any constant with pointer

Eg: ① int *p, *q;

1) p = p+q ; /* Invalid */
2) p = p*q ; or p = p*2 /* Invalid */
3) p = p/q ; or p = p/2 /* Invalid */
4) p = 0-q ; /* Invalid */
5) p = p<<2 /* Invalid */

Note :-

We can perform Substraction of two pointers or Substract any constant with pointer and we can add any constant with pointer is Possible.

Arithematic operations with pointers :-

We can perform two Arithematic operations with pointers

1) Unary Arithematic Operation : Increment (++), Decrement (--),

2) Binary Arithematic operation : Addition (+), Substraction (-).

The following Examples are unary Arithematic operations:

Example	Holding Address of Pointer	Operator		after Operation		bytes
int x=2; int *p=&x;	2023	++	--	2025	2021	2
char c='s'; char *ch=&x;	2023	++	--	2024	2021	1
float p=34; float *q=&p;	2023	++	--	2027	2019	4
long int x=2; long int *p=&x;	2023	++	--	2027	2019	4

The following Examples are binary arithematic operations

① int x=9, *p;

p=&x;

p=p+2; → /* The p pointer variable holding address is 2027 */

p=p-3; → /* The p pointer variable holding address is 2017 */

② int x=4, *P;

P= &x;

P= P+x; → /* The P address is 2027 */

P= P-x; → /* The P address is 2017 */

Pointers and Arrays :-

It has two types

1. Pointers and Single dimensional array
2. pointers and Multi dimensional array

1) Pointers and Single dimensional array :

* The array name by itself is starting address or pointer.
* The array name indicates zeroth (0th) Element address.

Eg : Model -1

```
Main()
{
int a[5] = {10,15,20,25,30},i;
clrscr();
for(i=0; i<5; i++)
{
  printf("%d address is %u \n", *(a+i),
                                 (a+i));
}}.
```

It is represented with the following diagram

a

10	15	20	25	30
2000	2002	2004	2006	2008

*(a+0) is refered the 0th Element address i.e. 2000

(a+1) is reffered the 1st Element address i.e. 2002

(a+2) is reffered the 2nd Element address is 2004

⋮

*(a+0) is reffered the 0th Element i.e., 10

*(a+1) is reffered the 1st Element i.e., 15

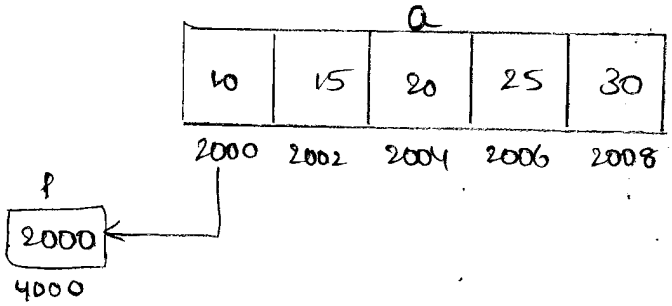
*(a+2) is reffered the 2nd Element i.e., 20

⋮

Model-2:

```
Main()
{
  int a[5] = {10,15,20,25,30}, i, *p;
  clrscr();
  p=a or &a[0];
  for(i=0; i<5; i++)
  {
    printf("%d address is %u \n", *(p+i),
                                   (p+i))
  }
}
```

It is represented to the following diagram



(P+0) is reffered 0th Element address i.e., 2000

(P+1) is reffered 1st Element address i.e., 2002

(P+2) is reffered 2nd Element address i.e. 2004

⋮

*(P+0) is reffered 0th Element i.e. 10

*(P+1) is reffered 1st Element i.e. 15

⋮

2) Pointers and two dimensional array :

The two dimensional array can be treated as matrix. In two dimensional array the first subscript indicates the rows and the second subscript indicates the columns.

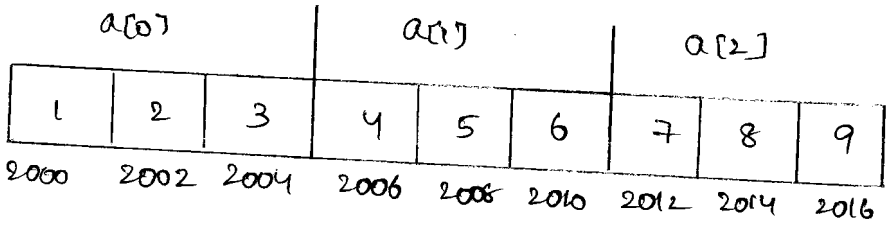
1) The array name by itself is a address or Pointer.

2) The Expression array name [i], it points or reffers i^{th} row starting address.

3) The Expression array name [i]+j, it points or reffers the i^{th} row, j^{th} column address.

Eg: It is represented by the following diagram

a



a[0] → reffers or is represented the 0th row starting address i.e., 2000

a[1] → Is represented the 1st row starting address i.e., 2006

a[2] → Is represented the 2nd row starting address i.e., 2012

a[0]+0 → It is represented the 0th row, 0th column address i.e., 2000

a[0]+1 → It is represented the 0th row, 1st column address i.e., 2002

a[2]+2 → It is represented the 2nd row, 2nd column address i.e., 2016

If I want to use * before the above Expressions, It reffers or access the corresponding elements

Eg:

*(a[2]+2) or *(*(a+2)+2) → It reffers or access the Element is 9.

Model-1;

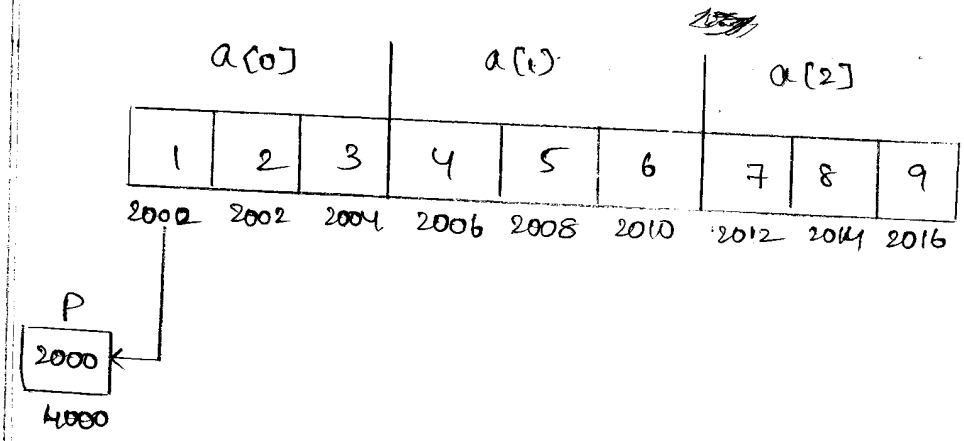
```
Main()
{
  int a[3][3] = {1,2,3,
                 4,5,6,
                 7,8,9
                };
  int i,j;
  clrscr();
  for(i=0; i<3; i++)
  {
    for(j=0; j<3; j++)
    {
      Printf("%d address is %u \n",
                *(*(a+i)+j), *(a+i)+j);
    }
  }
}
```

Model-2:

```
Main()
{
  int n[3][3] = {1,2,3,
                 4,5,6,
                 7,8,9
                };
  int i,j;
```

```
        clrscr();
        P=a or a[0];
        for(i=0; i<9; i++)
        {
          Printf("%d address is %u\n", *(P+i),
                                        (P+i));
        }
    }
```

The Model is represented by the following diagram.



* Array of pointers :—

A Collection of addresses or pointers is called an array of pointers (i.e., The array holds a Collection of addresses or pointers.) These addresses may be any individual ordinary variable addresses or any array Element addresses.

Syntax: Datatype *array name [size];

Eg: int *a[5];

```
    float *b[5];

        Main()
    {
      int a[4]={1,2,3,4}
```

```
int *p[4], i;
clrscr();
for(i=0; i<4; i++)
{
  p[i] = &a[i];
}
for (i=0; i<4; i++)
{
  Printf("%d address is %u", *p[i], p[i]);
}
}
```

It is represented in the following diagram

a

1	2	3	4
2000	2002	2004	2006

p

2000	2002	2004	2006
4000	4002	4004	4006

* pointers and Strings :-

String is Nothing but a collection of (or) group of characters is terminated with null character. The string name by itself is a starting address or pointer

It is represented with the following diagram.

Str

I	N	O	I	A	\0
2000	2001	2002	2003	2004	2005

(Str+0) → Is reffered or pointing the starting address i.e., 2000

(Str+1) → It is reffered or pointing the first location address ie 2001

If we are using * before the above Expressions, It access the corresponding characters

*(Str+0) → It reffers or access the ~~first~~ 0th character ie., I

*(Str+1) → It reffers or access the first character ie, N

Eg:
```
Main();
{
  char str[6] = "INDIA";
  int i;
  clrscr();
  for (i=0; *(str+i) != \0; i++)
  {
    Printf ("%c address is %u", *(str+i),
                                (str+i));
  }
}
```

* Pointers and array of strings :-

A collection or group of strings is called as Array of strings In this the first subscript indicates Number of strings and second subscript indicates Number of characters of each string. The string name array Name by itself is a starting address or pointer

It is represented in the following diagram

Str:

One \0	Two \0	Three \0	Four \0
2000	2010	2020	2030

(Str+i) → It reffers or pointing the i^{th} string address

(Str+0) → It reffers or pointing the 0^{th} string address i.e. 2000

(Str+1) → It reffers or pointing the 1^{st} string address i.e., 2010

⋮

By using the above Expressions, we can access the corresponding strings with Format-Specifier %s

Eg:

```
Main()
{
 Char Str[4][10]={"one","two","Three","Four"};
  Int i;
  clrscr();
  For(i=0; i<4; i++)
  {
   Printf("%s address is %u", (Str+i),
                              (Str+i));
  }
}
```

Note:

In the pointers and array of strings According to the above Example 40 bytes of memory was allocated But we are utilising 15 Bytes remaining is wastage.

Since the array of pointers to string concept was implemented. It reduce some of the wastage of memory space.

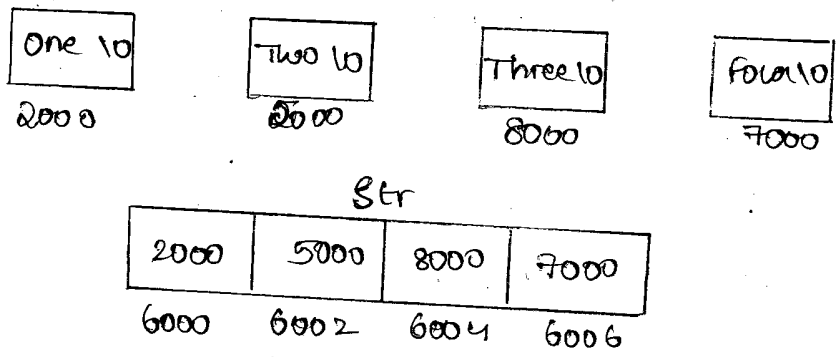
* Array of pointers to string :

A collection of string addresses or pointers is called as array of pointers to string.

Syntax: Datatype *Stringname [size];

Eg : char *str[] = {"one", "two", "three", "four"}

It is represented in the following figure



If I want

str[0] → It reffers or pointing the string is "one"

str[1] → It reffers or pointing the string is "two"

⋮

str[0] → It reffers or pointing the string "one" address is 2000

str[1] → It reffers or pointing the string "two" address is 5000

⋮

Eg :

```
main()
{
    char *str[] = {"one", "two", "three", "four"};
    int i;
    clrscr();
```

```
        for(i=0; i<4; i++)
        {
          printf("%s address is %u", str[i],
                                      str[i]);
        }
        getch();
    }
```

Note:

In pointers and Array of Strings we are wasting some memory space.

Eg: char str[4][10] = {"one", "two", "Three", "Four"};

The above statement allocated 40 bytes but we are utilising 15 bytes so the remaining Bytes are wastage. (25)

Since the array of pointers to Strings concept was Implemented. It reduce some of the wastage. According to the above Example diagram It was allocated 23 bytes.

* Pointer to pointer :-

A Pointer variable holds the address of another pointer variable.

Syntax: Datatype ** variable

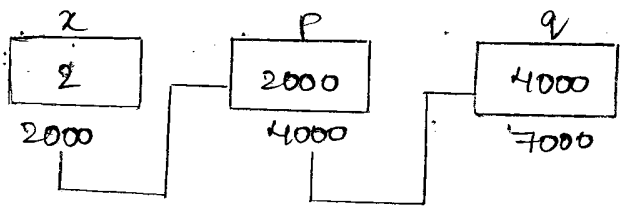
Eg: ① int **q;

② int x=2, *p, **q;

p = &x;

q = &p;

According to the above second Example the memory was allocated like below.



If I want to access the x address with q pointer variable I use *q ie., 2000

If I want to access the x value with q pointer variable I use **q ie., 2

Void pointer :-

A specific datatype of pointer variable cannot holds the address of another variable.

Eg: float *p;
int x=2;
p= &x;

* The void pointer can holds the address of any datatype of the variable.
* Before accessing the value of the variable with void pointer we must perform type-casting Because void datatype cannot be determine any bytes.

Syntax: void * variable;

Eg:
```
main()
{
  void *p;
  int x=2;
  float f=4.3;
  p= &x
  printf("%d", *((*int) p));
  p= &f;
```

```
    Printf("%d", *((*float)p));
    getch();
}
```

Dynamic Memory Allocation

Dynamic memory allocation :-

The runtime memory allocating is called as dynamic memory allocation.

Advantages :

* Save the memory space
* utilising heap memory
* Processing speed up

TO allocate memory at own time we have 3 functions.

1. malloc()
2. Calloc()
3. realloc()

(1) malloc() :

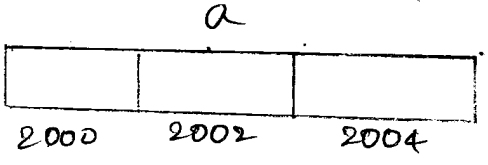
Syntax : (void *) malloc(unsigned int size);

It allocates maximum of 64 Kb at a time since the unsigned int maximum value is 65535.

Eg : Allocates memory for 3 elements

```
int *a;
a = (int *) malloc (3* sizeof (int));
                (or)
a = (int *) malloc ( 3* 2);
                (or)
a = (int *) malloc ( 6);
```

It is represented as the following diagram.



(2) Calloc():

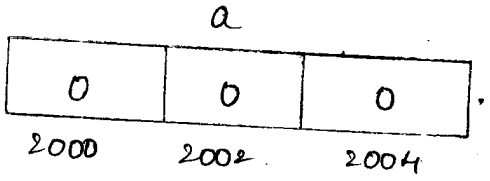
Syntax:

(void *)Calloc(unsigned int no of elements, unsigned int size);

Eg: Allocation of memory for 3 elements

```
int *a;
a = (int *) calloc (3,2);
        (or)
a = (int *) calloc (3, sizeof(int));
```

It is represented as the following diagram



Difference between malloc() and calloc():

1. Arguments are different
2. malloc() memory blocks initialised with garbage value and calloc() memory blocks initialised with zero's

(3) Realloc():

Using the above two functions we cannot expand the memory ones reallocated since to expand the memory realloc() was implemented.

Syntax:

(void *) realloc(void * block; unsigned int size);

The first argument is previously obtained memory block through malloc() or calloc(). Second argument is the total size (old + new).

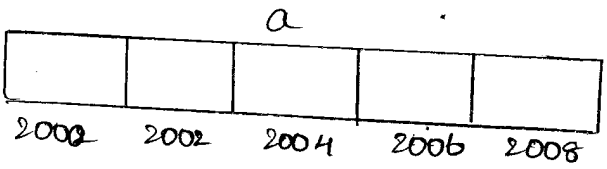
Eg: int *a;

a = (int *) realloc (a, 5*2);

(or)

a = (int *) realloc (a, 5* size of (int));

using the above realloc() we are expanding 4 bytes it is represented as the following figure.



Note :-

1. If malloc() or calloc() previously returns null (without any memory space). Then the realloc() works just like an malloc() or calloc().
2. If sufficient space is not available after the block or memory block then realloc() allocates a new memory block of the required size and it copies the contents or elements to the newly allocated block. and remove the old block.

Free() :-

This function is used to free or Remove the allocated memory space in heap memory

Syntax :

(void *) free(void * block);

Eg: free(a);

Dynamic Memory Allocation of Multi-dimensional array :-

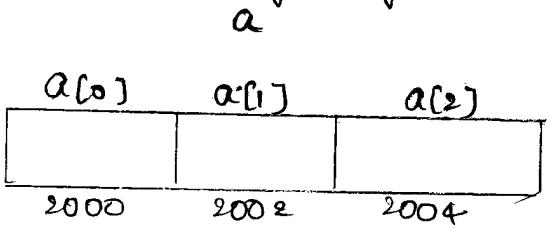
In this No direct way to allocate memory

Space. In two dimensional array first we allocate to rows after that we allocate memory to each row number of columns.

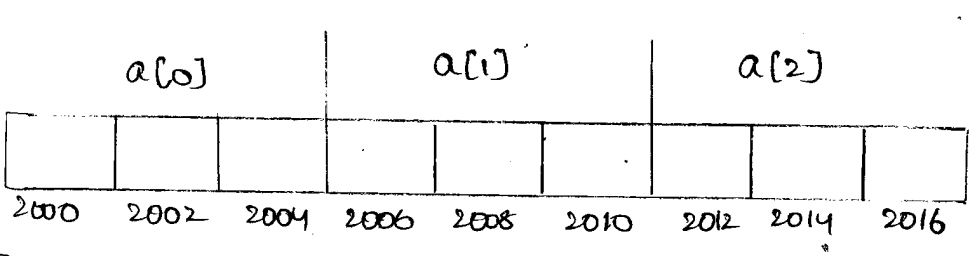
Eg: int **a;

```
a = (int **) malloc (6);
        (or)
a = (int **) malloc (3 * size of (int));
        (or)
a = (int **) malloc (3 * 2);
        (or)
a = (int **) malloc (3 * size of (int *));
for (i=0; i<3; i++)
{
  a[i] = (int *) malloc (3 * size of (int));
}
```


First it allocates memory to rows we shown is the following figure.



After it allocates memory for each row of columns.



Self referential structures :-

A Self referential structure contains a Pointer member that points to a structure of

the same structure type. For Example,

```
struct node
{
    int data;
    struct node *nextptr;
};
```

(4)

Defines a type struct node a structure of type struct node has two members - integer member data and pointer member nextptr. Member nextptr points to a structure of type struct node, a structure of the same type as the one being declared here, Hence the term "self refferential structure". Member nextptr is reffered to as a link ie, next ptr can be used to "tie" a structure of type struct node to another structure of the same type.

Self Refferential structures can be linked together to form useful data structure such as lists, Queues, Stacks and trees

1) Command line arguments :-

1. The arguments are passed or supplied to a program when the program is executing
2. In the command line argument the main can receive or contains two arguments. They are : arg c, arg v.
3. The command line arguments information is passed to the program through the arguments (arg c, arg v).

Arg c :

1. This is an argument counter
2. It indicates or specifies the total number of arguments passed from command prompt

Arg v :

1. This is an argument vector and represents array of pointers to string.
2. The size of this array will be equal to the value of arg c.
3. Syntax of command line argument is :

   C:\> program name file1 file2

   Eg : C:\> copy abc.txt xyz.txt

   argv(0) → copy

   argv(1) → abc.txt

   argv(2) → xyz.txt

In command line argument we must declare the main function like below.

```
main(int argc, char *, argv[]).
main()
{
  int - - - - - - -
    - - - - - - - -
    - - - - - - -
}
```


Eg: This program is copying of one file into another file.

```
main(int argc, char * argv[])
{
  FILE *fp, *ft;
  char ch;
  clrscr();
  fp = fopen(argv[1], "r");
  ft = fopen(argv[2], "w");
  if (fp == NULL)
  {
    Printf("The file was not existed");
    exit(1);
  }
  else
  {
    while(!feof(fp))
    {
      ch = fgetc(fp);
      fputc(ch, ft);
    }
  }
  fclose(fp);
  fclose(ft); } fcloseall();
}
```

Structures and Unions Unit-3

Structures :-

* A group of different related data items or member variables is called structure.
* The structure is reffered with a single name
* Sometime it is called as "Hetrogenious" datatypes

Declaration :-

keyword ← Struct [<Structure name>] → tag

{

Structure (or) Body / Structure templets (or) data items

Datatype variable$_1$, variable$_2$ ....;
Datatype variable$_1$, variable$_2$ ....;
.....
.....

} [<Structure variable>];

Eg: ①

```
Struct Student
{
  int no;
  char name[10];
  char address[10];
};
```

②

```
Struct Employee
{
  char name[10];
  char addr[10];
  float salary;
};
```

```
main()
{
  struct employee e;
  .....
  .....
}
```

How to initialise :-

It is Initialise like below.

Eg-① struct student

```
{
  int no;
  char name[10];
  char address[10];
} s = {1, "Sai", "GMD"};
```

Eg-② struct Employee

```
{
  char name[10];
  char addr[10];
  float salary;
};
main()
{
  struct employee e = {"Raj", "HYD", 6000};
  .....
  .....
}
```

How to store :-

The structure datatypes are stored in a sequential or continuous memory location, according to the above Example-1.

s

1	Sai	GMD
no	name	address
2000	2002	2012

How to access Structure Data Items :

The Structure Data Items are access with a Structure variable is followed by . (Dot operator and member variable or Data item.

Syntax: Structure variable . member variable

According to the above Example -1.

S.no → It gives or access the number '1'.

S.name → It gives or access the name 'Sai'.

S.address → It gives or access the address 'GMD'

* Array of Structures :-

In array Every Element is a Structure type (i.e., in array Each memory location Contains a Structure) It is declared as follows.

```
Struct Student
{ int no;
  char name[10];
  char address[10];
} S[3] = { 1, "Sai", "GMD"
           2, "Rama", "H40"
           3, "Raj", "KKD"
         };
```

It is stored in memory as a Sequential like below.

S

S[0]			S[1]			S[2]		
no	name	address	no	name	address	no	name	address
1	Sai	GMD	2	Rama	H40	3	Raj	KKD

It is accessed like below

Structure variable [Index] . member variable

Eg : S[0].no → It gives the number '1'

S[0].name → It gives the name 'Sai'

S[0].address → It gives the address 'Gmo'

⋮

S[2].address → It gives the address 'KKO'

Note :

Array within Structure means a Structure contains an array.

Eg : The above Student Structure contains 2 arrays are name, address.

* Nested Structure :-

A Structure within the another Structure is called as Nested Structure. There are two ways of declaring such a Structure.

Eg : ①

```
Struct date
{
  int day;
  int month;
  int year;
}birth;
Struct employee
{
  char name[10];
  Struct date birth;
  float salary;
} e;
```

Eg:② Struct employee

```
{
  char name[10];
  Struct date
  {
    int day;
    int month;
    int year;
  } birth;
  float salary;
}e;
```

We can access the inner most structure data items like e.birth.daf, e.birth.month, e.birth.year

* Pointer to Structure (or) Structure pointer :-

A Structure is reffered with a pointer variable is called pointer to structure or Structure pointer variable. It is declared as follows.

```
Struct student
{
  int no;
  char name[10];
  char address[10];
} s = {1, "sai", "ongo"}, *ptr;
*Ptr = &s;
```

The structure data item or Numbers using structure pointer variable in two ways

1. using '→' (Arrow or Indirection operator)

2. Using '.' (dot (or) Period Operator)

Eg:① ptr → no → It gives or access '1'

ptr → name → It gives or access 'Sai'

ptr → address → It gives or access 'Gmd'

Eg:② (*ptr).no → It gives '1'

(*ptr).name → It gives 'Sai'

(*ptr).address → It gives 'Gmd'.

* Structures and Functions :-

The Structure datatem values (or) member variable values will be passed as arguments to Functions. There are three methods.

Method - 1:

Here the datatem values individually will be passing to function.

Eg-①:

```
main()
{
  Struct Student
  { int no;
    char name[10];
    char address[10];
  } s = { 1, "Sai", "Gmd" };
  void s display (int, char[], char[]);
  clrscr();
  s display (s.no, s.name, s.address);
  getch();
}
```

```
void display (struct student *ptr)
{
  printf("%d %s %s", ptr->no, ptr->name,
                      ptr->address);
}
```

Method -2:

Here the entire to value be passing the function.

```
main()
{
  struct student
  {
    int no;
    char name[10];
    char address[10];
  } s = {1, "sai", "GMD"};
  void display (struct student);
  clrscr();
  s display(s)
  getch();
}
void display (struct student std)
{
  printf("%d %s %s", std.no, std.name,
                      std.address);
}
```

According to the above two methods the actual arguments xerox will be the copied into formal arguments.

Method -3:

Here the Structure address will be passed into function.

```
main()
{
  Struct Student,
  {
    int no;
    char name[10];
    char address[10];
  } s = {1, "sai", "Gmo"};
  void s display (Struct Student, *);
  clrscr();
  s display (&s);
  getch();
}
void display (Struct Student *ptr)
{
  Printf ("%d %s %s", ptr → no, Ptr → name,
                      Ptr → address);
}
```

* Union :-

In union all the data items or member variables use the same memory location (i.e., The Compare Allocates memory of largest datatype variable and can be shared by all data items.

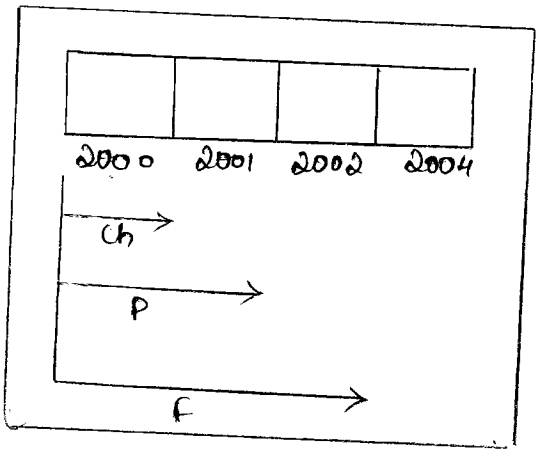
Syntax: union [<union name>]

```
{
    datatype variable1, variable2 ....;
    datatype variable1, variable2 ....;
    ..........
    ..........
} [<union variable>];
```

Eg:

```
union item
{
    int p;
    float f;
    char ch;
} it;
```

How to allocate memory:



During accessing only the currently stored variable value will be accessed

Eg: For Example the Following Statement

```
it.p = 320;
it.f = 420.21;
Printf ("%d", it.p); /* It gives wrong value */
```

Because the currently stored value is float variable value so we can access like below.

```
PrintF ("%F", ft.F);
```

Difference between structures and union :-

1. Memory Allocation :

* The structure allocates memory for all data items.

* In union the only largest data type of the variable memory will be allocated and all the data items can stored the memory location.

Eg :

```
Struct Student
{
  int no;
  char name[10];
  char addr[10];
};
```

```
union Student
{
  int no;
  char name[10];
  char addr[10];
};
```

The memory size of structure 22 bytes and the memory size of union is 10 bytes

2. Operations and members (or) member variable accessing :

In structure all the data items or member variables can be accessed at a time (or) at the same time. But in union the currently stored variable can be accessed

3. Declaration :

The structure is declared with 'Struct' keyword, the union is declared with 'union keyword.

```
Struct [<Structure names>]/union
{
  Data type variable1, variable2;... ;
  Data type variable1, variable2... ;
  ------
  ------
}
```

4. In union if we assign the new value to any member variable then replace the previous member variable value. But in structure not like that.

Union Of Structures :

We can use the structure in union or union in structure.

Eg: ① Struct date

```
{
  int day;
  int month;
  int year;
}
union person
{
  char name[10];
  Struct date birth;
  float salary;
} P;
```

② union date

```
{ int day;
  int month;
  int year;
}
Struct person
{
  char name[10];
  Struct date birth;
  float salary;
}
```

Features :-

1. We can assign a structure variable to another structure variable at the same time by using '=' operator.
2. Nested structure is possible.
3. Array of structures is possible.
4. Pointer to structure or structure pointer is possible.
5. Structures and Functions is possible

Bitwise operators :-

Computers represent all data internally as sequence of bits. Each bit can assume the value '0' or the value 1. On most systems, a sequence of 8 bits form a byte. The standard storage unit for a variable of type char. Other datatype are stored in larger number of bytes. The bitwise operators are used to manipulate the bits of integral operands (char, short, int and long, both signed and unsigned). Unsigned integral are normally used with the bitwise operators.

Operator	Description
& Bitwise AND	The bits in the result are said to 1 if the corresponding bits in the two operands are both 1.
| Bitwise Inclusive OR	The bits in the result are set to 1 if atleast one of the corres

	ponding bits in the two Operands is 1
^ Bitwise Exclusive OR	The Bits in the result are set to 1 if Exactly one of the Corresponding bits in the two operands is 1
<< Left Shift	Fill from the Right with 0 bits
>> Right Shift	The Method of filling from the left is Machine dependent.
~ Ones Complement	All 0 Bits are set to 1 and all 1 Bits are set to 0

Bit Fields :-

C enables programmers to Specify the number of bits in which an unsigned or int member of a Structure or union is Stored. This is reffered to as a bit field. Bit fields enable better memory utilisation by storing data in the minimum number of bits required. Bit field members must be declared as int or unsigned

Files

File :-

File is a collection of data. The data may be either text or records. The file information is stored on hard disk permanently.

File Operations :-

The File follows following operations They are :

(1) Opening the File
(2) Read/Write the file (or) I/O Operations on the file.
(3) Closing the File.

1) Opening the File :

Before performing any file read and Write operations, the file must be opened. If we want to open file we the function "fopen()".

Syntax :

```
FILE *fp;
fp = fopen("filename", "mode");
```

The above syntax 'FILE' is a structure, '*fp' is a file pointer in the fopen(). The first argument is physical 'filename'. And second argument 'mode' is any of the following.

r : This is open a text file for reading (ie., read mode).

w : This is created text file for writing (ie., write mode)

a : This is append to a text file (ie., append mode). (or) If the file was not existed then it performs write operation

r+ (Read + write) : This is open a text file for Read and write

w+ (write + Read) : This is create a text file for write and read

a+ (Append + Read) : This is append + reading file.

rb : This is open a binary file for reading.

wb : This is create a binary ~~tree~~ file for writing.

r+b : This is append binary file for Read and write

w+b : This is create binary file for write and Read

ab : This is append to binary file.

a+b : This is append + read binary file

Eg : FILE *ft;

ft = fopen("Sample.txt", "w");

Read and write operations of the file (I/O Operations of the File) :-

Reading From the files :-

The data will be read from the file, for reading data we use following functions.

(1) getc() :

This function is used to read a character from a file. That has been opened in read mode.

Syntax : int getc(FILE *fp);

Here 'fp' is a file pointer.

Eg : char ch;

ch = getc(fp);

(2) fgetc() :

It is same as above. Instead of getc() we can use fgetc()

(3) fgets() :

This function read a string from file.

Syntax :

char[] fgets(char[], int length, FILE *fp);

Here first argument is string, second argument is length of the string and third argument is the file pointer.

Eg :- char STR[12];

fgets(STR, 12, fp);

(4) getw() :

This is used to read integer value from file.

Syntax : int getw(FILE *fp);

Eg:- int p;

p = getw(fp);

(5) fscanf():

This function reads mixed data from the file.

Syntax: int fscanf(FILE *fp, "<Formate String>", "<List of variables>");

Eg: char str[10];
int p;
fscanf(fp, "%s %d", str, &p);

(6) fread():

This function read only datatype from the file to a buffer or block.

Syntax:

size_t fread(void *buffer, size_t size, size_t n, FILE *fp);

Here size_t defined as type def unsigned int size_t. Here, the buffer receives data from the file, size is length of each item read in bytes (the no. of bytes of each item) and n is the number of items read.

Eg: int p;
double k;
fread(&p, sizeof(int), 1, fp);
fread(&k, sizeof(double), 1, fp);

Writing to files :-

The data will be write to the file, for writing we use the following functions.

(1) putc():

This is used to write a character to a file. That has been opened in write mode.

Syntax: int putc(char ch, FILE *fp);

Eg: char ch = 'a';
putc(ch, fp);

(2) fputc():

It is same as above instead of putc(), we use fputc().

(3) fputs():

This function write a string to a file.

Syntax:
int fputs(char[], int length, FILE *fp);

Eg: char str[] = "WELCOME TO C";
fputs(str, 10, fp);

(4) putw():

This is used to write integer value to a file.

Syntax: int putw(int p, FILE *fp);

Eg: int p=10;
putw(p, fp);

(5) fprintf():

This function writes mixed data to file.

Syntax:

fprintf(FILE *fp, "<Formate string>", <list of variables>);

Eg: char str[] = "welcome to c";
int p=20;
Printf(fp, "%s %d", str, p);

(6) fwrite() :

This Function write any data type to file.

Syntax :

```
size_t fwrite(void * buffer, size_t size, size_t n,
                                        FILE *fp);
```

Here buffer is used to store data, size is length of each item write in a byles and n is number of items write.

Eg:
```
int p=10;
double k=20;
fwrite(&p, sizeof(int), 1, fp);
fwrite(&k, sizeof(double), 1, fp);
```

5) fclose() :

A file must be closed after read and write operations have been completed. This ensured that all outstanding information associated with the file is flusheed out from the buffers and all links to the file are broken.

Syntax : fclose(FILE *fp);

Eg: fclose(fp);

(4) fcloseall() :

It closes all opened files.

Other File Functions :-

(1) feof() :

This Function can be used to test for end of the file. When the file pointer reaches to end it returns 'EOF'.

Syntax :- feof (FILE *fp);

Eg : if(!feof(fp))

(2) ftell() :

This function gives the current position of the file pointer. That means it gives the no. of bytes of the current location of the file pointer.

Syntax :- long int ftell (FILE *fp);

Eg : long int n;
n = ftell(fp);

(3) fseek() :

This function is used to move the file pointer to a desire location in the file.

Syntax : int fseek (FILE *fp, long int offset, int position);

Here, fp is a file pointer, offset is the no. of bytes to be moved from the location specified by position (it may be negative or positive long int)

The position can take out of the following.

Constant : Seek_set, Seek_cur, Seek_end

Value : 0, 1, 2

Station in the File : Beginning of the file, Current position of the file pointer, End of the file.

Eg :

1. fseek (fp, 0L, 0) go to the beginning
2. fseek (fp, 0L, 2) go to the end of the file
3. fseek (fp, 0L, 1) Stale at the current position
4. fseek (fp, M, 0) move to M+1 bytes in the file.

for fseek().

5. fseek(fp,M,1) go forward by M bytes from the current position

6. fseek(fp,-M,1) go backward by M bytes from the current position.

Means 0 (L indicates long constant).

(4) rewind():

This function resets or moves the file pointer to the starting of the file.

Syntax: rewind(FILE *fp);

Eg: rewind(fp);

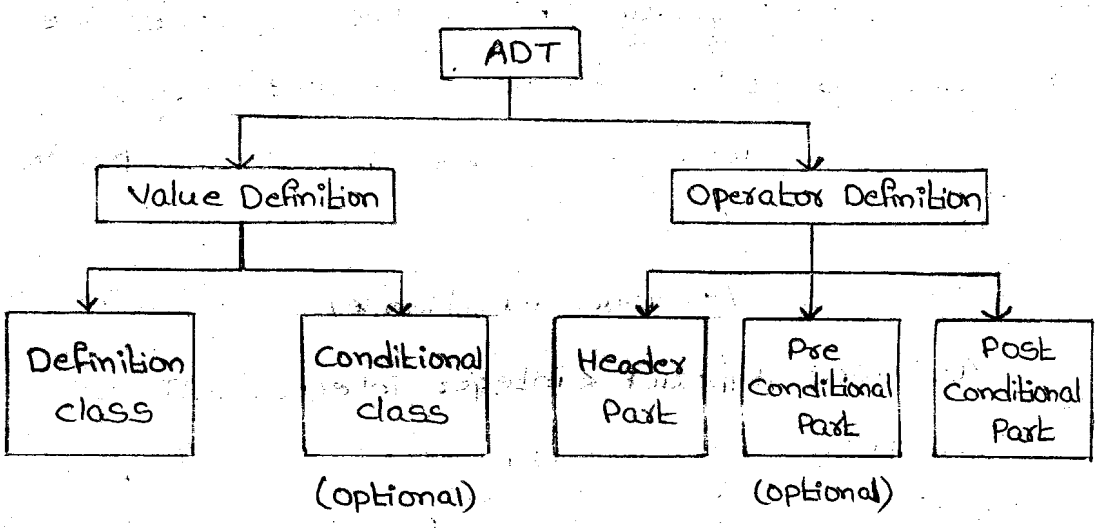
* Explain the functions for file handling.

A) Above all functions (from 'file definition to 'rewind();')

* Explain the Random file accessing functions

A) ftell(), fseek(), rewind(),

Abstract Datatype (ADT):



Abstract datatype is useful to specify the logical properties of datatype. The term 'ADT' refers to mathematical concept that defines data type.

The ADT cannot concern with space or time efficiency as they are implementation issues. ADT is not at all concern with implementation details.

Methods of specifing ADT's:

The methods we adopt for declaring ADT is semi-formal and borrows from 'c' notation and extends the notation whereever necessary.

To illustrate the concept of ADT consider the ADT RATIONAL which is a mathematical concept.

A Rational number is a number that can be expressed as a quotient of two numbers.

We can perform addition, multiplication and testing for equality of rational numbers. We can create a rational number from 2 integers.

The following is the initial specification of ADT.

```
/* value definition */
abstract typedef <integer, integer> RATIONAL;
Condition RATIONAL [1] <>0;

/* Operator definition */
abstract RATIONAL make rational (a,b)
int a,b;
precondition b<>0;
Postcondition make rational [0]==a;
              make rational [1]==b;

① abstract RATIONAL add (a,b)
   RATIONAL a,b;
   Postcondition add [1]==a[1] * b[1];
                 add [0]==a[0]*b[1]+a[1]*b[0];

② abstract RATIONAL mult(a,b)
   RATIONAL a,b;
   Postcondition mult[0]==a[0]*b[0];
                 mult[1]==a[1]*b[1];

③ abstract equal (a,b)
   RATIONAL a,b;
```

Postcondition equal==(a[0]*b[1])==(b[0]*a[1]);

An ADT consists of two parts

1) Value definition

2) Operator definition

1) Value definition:

It defines the collection of values for the ADT and it consists of 2 parts they are

a) Definition class

b) Condition class

Eg: The value definition for ADT RATIONAL states that it consists of 2 integers.

The conditional class states that among the two integers the second one does not equals to zero.

The keywords "ABSTRACT" type def we define or introduce a value definition and the keyword 'condition' is used to specify any conditions on newly defined type.

In value definition, the definition class is compulsory whereas the conditional class is optional.

2) Operator definition:

The immediate following of the value definition is the operation definition. It consists of 3 parts, they are

a) Header part

b) Pre conditions

c) Post conditions.

For example, the operator definition of ADT rational includes the operation of creation (i.e; make rational) addition, multiplication as well as test of equality.

In the above example, the header of the definition is the first 2 lines which is similar to 'c' function header.

The keyword 'ABSTRACT' indicates that this is not a 'c' function but it is an ADT operator definition.

The postcondition specifies what the operation will perform. In postcondition, the name of the function [mult] is used to denote the result of the operation.

Mult [0] → Represents result of numerator.

Mult [1] → Represents result of denomenator.

The preconditions states that the make rational cannot be applied if the second parameter is zero. In general, the precondition specifies any restrictions that must be satisfied before the operation can be applied.

* Sequence of value definition:

In devoloping the specifications for various datatypes we use set of theoritical notations to specify the values of an ADT.

A sequence is nothing but simply an ordered set of elements. A sequence 's' is sometimes written as the enumerations of its elements, such as

$$S :: <S_0, S_1, S_2 \text{---} S_{n-1}>$$

If 's' contains n elements then s is said to be length of n.

* Different length character strings for an ADT:

There are 4 basic operations that any system supports on string. They are

1) Length
2) Concat
3) Substr
4) Pos.

1) Length:

A function that returns the current length of a string.

2) Concat:

A function that returns the concatenation of its two input strings.

3) Substr:

A function that returns a sub string of a given string.

Eg: implementation ↔ Substr (3, 7).

4) POS:

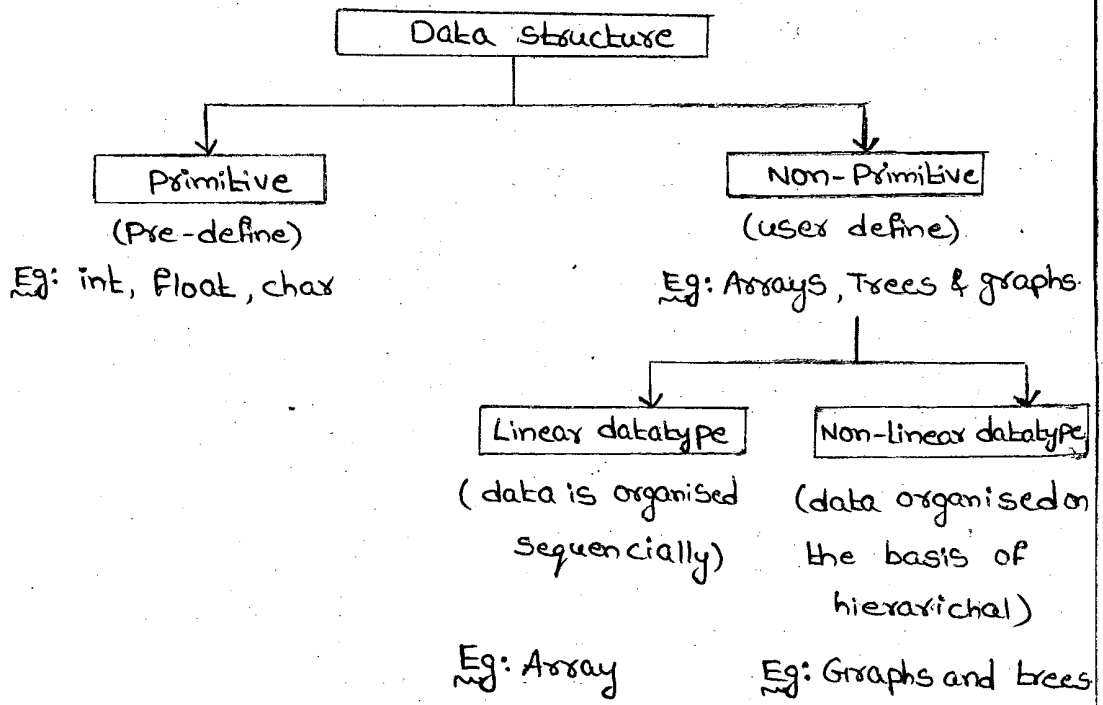
A function that returns the first position of one string as a substring of another.

⑥

```
abstract typedef <<char>> STRING;
abstract length (S);
STRING S;
Postcondition length == len(S);
abstract STRING concat (S1, S2)
STRING S1, S2;
Post condition concat == S1+S2;
abstract STRING substr (S1, i, j)
STRING S1;
int i, j;
Precondition 0<=j<len(S1);
             0<=j<len(S1)-1;
Post condition substr == sub (S1, i, j);
```

* Data Structures:

The logical or mathematical model of a particular organisation of a data is called "data structure."



* **STACK:**

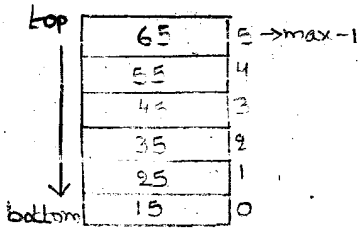
An ordered collection of data items is called "Stack". We can store stacks in memory in 2 ways.

1) Arrays (Sequential representation)
2) Linkedlists (Random)

The elements can be added as well as deleted at one end that is called TOP. It follows the principle of LIFO (Last in first out).

Stack has mainly 3 operations they are

a) Traverse (Read)
b) Push (Insertion)
c) Pop (Deletion).



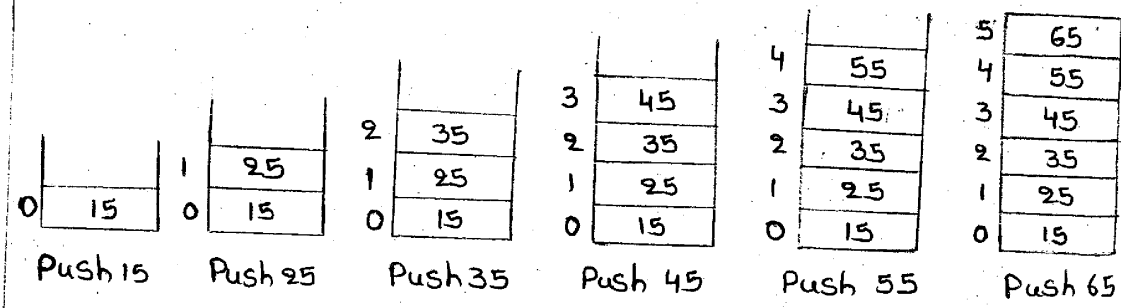
Stack full and empty conditions:

```
if top==max-1
printf ("stack is full");
if top==-1
printf ("stack is empty");
```

1) **Push:**

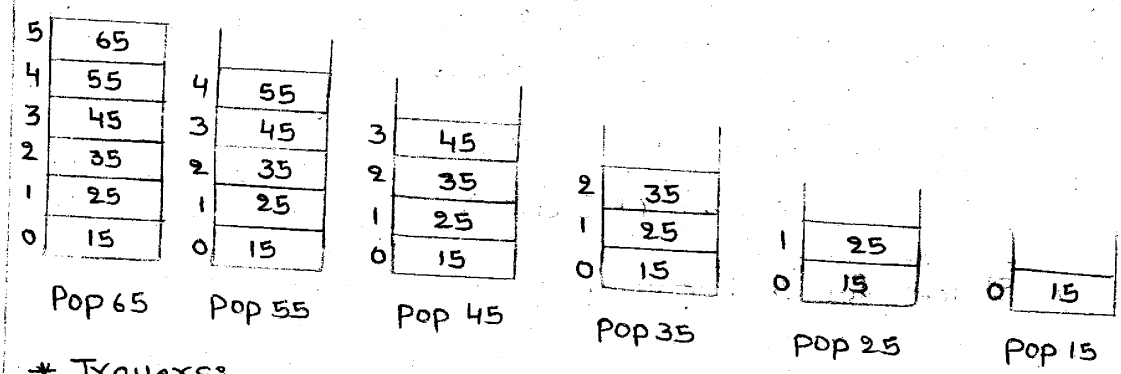
```
top=-1
if top==max-1
printf ("stack is full");
top=top+1
Push 15.
```

(8)



2) Pop:

```
if top == -1
Printf("stack is empty");
Pop 65
top = top-1
```



* Traversing:

Moving from top to bottom in stack is called Traversing. Printing the ith position of data where "i" is the position of address.

* Push():

This is used to insert data into the stack array. Top is treated as global variable.

* Pop():

This is used to delete data by LIFO Principle.

* Applications of stack:

The Arithmetic expression can be represented in three ways they are

1) Infix Notation
2) Prefix Notation
3) Postfix Notation.

1) Infix Notation:

If the operator is placed in between 2 operands then it is called "Infix expression." The expressions which are used in mathematical operations are infix expressions.

Infix expressions can be converted into postfix and prefix.

Eg: a+b.

2) Prefix Notation:

If an operator is placed before the operand then it is called prefix expression. This is also called as "Polished Notation."

Eg: +ab.

3) Postfix Notation:

If an operator is placed after the operands then it is called "postfix expression." This is also called as "Reverse polished notation" (or) Suffix notation.

Eg: ab+.

Conversion of Infix into Prefix and Postfix Notations:

To convert an Infix expression into postfix expression we have to follow the precedency of operators.

Operator	Precedency
$, ^	1st priority
*, /	2nd priority
+, −	3rd priority

Algorithm:

Step 1: Scan infix string from left to right one character at a time.

Step 2: If we get any operand from infix expression put it in postfix notation.

Step 3: If we get any operator from infix expression then push it into stack. But, before pushing it into the stack check that this pushing operator within the top position operator of the stack. If the pushing operator less or equal than the top position operator, remove the top position operator from the stack and put in expression and decrease the top.

Again check the pushing operator within the top if it is less or equal, do the same as above. This procedure will be continue untill the pushing operator is greater. If

it is greater, then the pushing operator is pushed into stack.

Step 4 : If the pushing operator is "(" put it into stack without checking.

Step 5 : If the pushing operator is ")" remove the stack elements one by one and put them in postfix expression, upto we get "(" in the stack. After that remove the "(" from the stack.

* Convert the given infix expression (A$B*C-D+E/F/(G+H)) into postfix notation.

Step	Symbol	Stack	Expression
1	(	(	
2	A	(	A
3	$	($	A
4	B	($	AB
5	*	(*	AB$
6	C	(*	AB$C
7	-	(-	AB$C*
8	D	(-	AB$C*D
9	+	(+	AB$C*D-
10	E	(+	AB$C*D-E
11	/	(+/	AB$C*D-E
12	F	(+/	AB$C*D-EF
13	/	(+/	AB$C*D-EF/
14	(	(+/(	AB$C*D-EF/
15	G	(+/(	AB$C*D-EF/G
16	+	(+/(+	AB$C*D-EF/G
17	H	(+/(+	AB$C*D-EF/GH
18	)	(+/	AB$C*D-EF/GH+
19	)	—	AB$C*D-EF/GH+/+

* Evaluation of postfix expression:

Method:

Step 1: Scan the expression from left to right till the operator is encountered.

Step 2: Scan the backone to obtain immediate two left operands.

Step 3: Perform the indicated operation.

Step 4: Replace the operands and the operator as result.

Algorithm:

Step 1: Scan the expression from left to right.

Step 2: If we get an operand then push it into stack.

Step 3: If we get an operator then pop the top 2 operands from the stack.
The value poped is an operand 2 and the second value is poped is at operand 1.
This can be achieved by using following statements.

Step4: $OP_2 = pop()$

$OP_1 = pop()$

Step5: Perform the operation

ie; $Res = OP_1$ op OP_2

Where op_1 and OP_2 are operands

op is an operator.

Step 6: Push the result on to the stack.

step 7: Repeat the above procedure till the end

of input is encountered.

Eg: 5 6 2 + * 12 4 / -

Step	Symbol	Stack
1	5	5
2	6	5,6
3	2	5,6,2
4	+	5,8
5	*	40
6	12	40,12
7	4	40,12,4
8	/	40,3
9	-	37

* Conversion of infix to prefix Notation:

Algorithm:

Step 1: Scan an infix string from right to left one character at a time.

Step 2: If we get any operand, then from infix expression put it in prefix notation.

If we get any operator from infix then push it into stack.

But, before pushing to stack check the pushing operator with the top position operator of the stack.

If the pushing operator is lessthan the top operator of the stack than the top operator will be shifted to an expression This method will continue until the, pushing operator is greater. If the pushing operator

is greater or equal then push this operator into stack.

Step 4: If the pushing operator is ")" without checking push it into stack.

Step 5: If the pushing operator is "(" then remove the stack elements one by one and put them in prefix expression upto we get "(". After that the "(" is also removed from the stack.

Eg1: ((A+B) * (C-D)).

Step	Symbol	Stack	Expression
1	)	)	
2	)	))	
3	D	))	D
4	-	))-	D
5	C	))-	CD
6	(	)	-CD
7	*	)*	-CD
8	)	)*)	-CD
9	B	)*)	B-CD
10	+	)*)+	B-CD
11	A	)*)+	AB-CD
12	(	)*	+AB-CD
13	(	-	*+AB-CD

Eg 2: (A$ B* C-D+E/F/(G+H)).

Step	Symbol	Stack	Expression
1	)	)	
2	)	))	

3	H	))	H
4	+	))+	H
5	G	))+	GH
6	(	)	+GH
7	/	)/	+GH
8	F	)/	F+GH
9	/	)//	F+GH
10	E	)//	EF+GH
11	+	)+	//EF+GH
12	D	)+	D//EF+GH
13	−	)+−	D//EF+GH
14	C	)+−	CD//EF+GH
15	*	)+−*	CD//EF+GH
16	B	)+−*	BCD//EF+GH
17	$	)+−*$	BCD//EF+GH
18	A	)+−*$	ABCD//EF+GH
19	(	—	+−*$ABCD//EF+GH

* Queue:

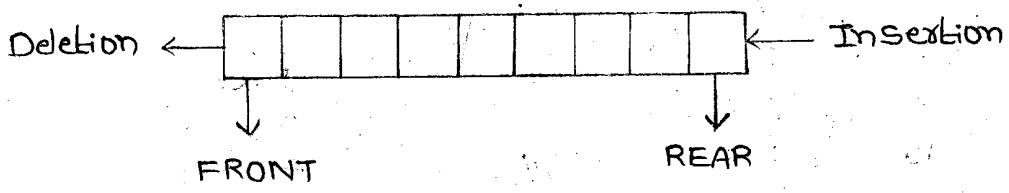
A queue is an ordered collection of data. In queues, the data is inserted at one end and deleted from another end.

In real life, queues form at many places such as petrol stations, theatres, railway ticket and reservation counters etc.

The queue will follow the principles of FIFO (First In First Out). That means that the information retrive from a queue comes in the same order or sequence as we place them on the queue. The purpose of the queue is to provide some form of buffering. We use queues in computer system

(16) For process management, buffer between the fast computer and a slow printer.

The following figure shows a queue representation.



* Representation of queues:

The queue will be represented in two ways.

1) Using Arrays
2) Linked Lists.

One of the most common way to implement the queue is using an array.

The insertion will be done from one end i.e; REAR and the deletion will be done from another end i.e; FRONT.

The rules for manipulating these variables are [FRONT, REAR].

1) Each time information is added (insert) to the queue we increment REAR.
2) Each time information taken from the queue (delete) we increment FRONT.
3) When FRONT = REAR, the queue is EMPTY.
4) Whenever REAR = MAX [size] the queue is FULL.

The array implementation of queue contains several drawbacks. The max queue size has to be set at compile time rather than at

runtime. So, the space can be coasted if we don't use the full length of array.

* Operations on queues:

A queue has basically 2 operations, they are

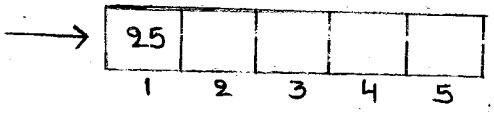
1) Adding new items to the queue.
2) Deleting / Removing items from the queue.

The following is an example of new items in the queue.

Insertion:

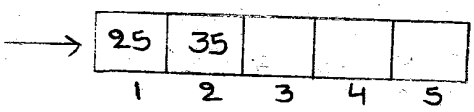
FRONT = REAR = 0

Step-1: Insert 25.



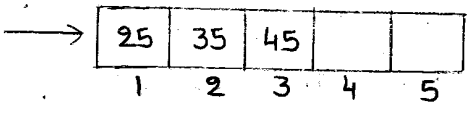
FRONT = 0 , REAR = 1

Step-2: Insert 35



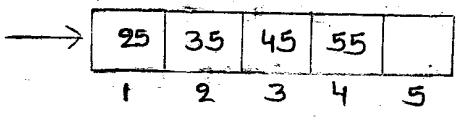
FRONT = 0 , REAR = 2

Step-3: Insert 45



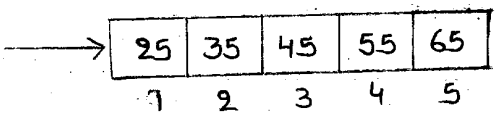
FRONT = 0, REAR = 3

Step-4: Insert 55



FRONT = 0, REAR = 4

Step-5: Insert 65

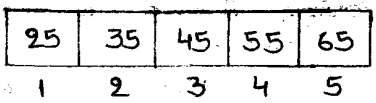


FRONT=0 , REAR= MAX [size] = 5

∴ Queue is Full.

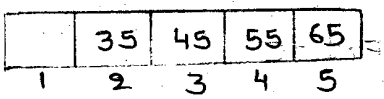
Deletion:

~~Step-N:~~



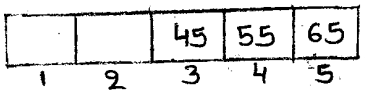
FRONT = 0, REAR= 5.

Step-1: Delete 25



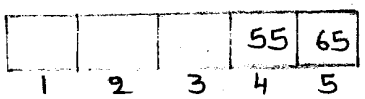
FRONT = 1 , REAR=5

Step-2: DRlete 35



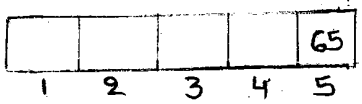
FRONT= 2, REAR=5

Step-3: Delete 45



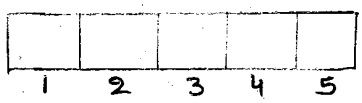
FRONT= 3 , REAR= 5

Step-4: Delete 55



FRONT=4, REAR=5

Step-5: Delete 65



FRONT= 5 , REAR=5

i.e; FRONT= REAR

∴ Queue is empty.

* Circular queue:

A circular queue is a queue in which all locations are treated as circular. In ordinary queue the deleted portion is wasted. So, to overcome the drawback of queue the circular queue was implemented. In this technique the deleted portion also be utilised or reused.

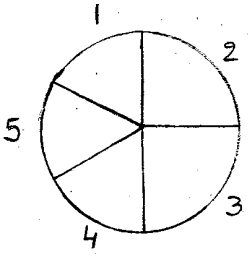
Operations of circular queue:

We can perform the following 3 operations on circular queue.

1) Insertion
2) Deletion
3) Reuse of deleted position.

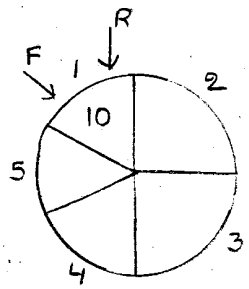
1) Insertion:

The initial stage of the circular queue is formed with the maximum size. We declared or defined, as its front and rear positions are initialised to zero. Then we call the queue is empty.



Front = 0
Rear = 0

Insert or add 10 to queue. Now the element value 10 is inserted or placed at the first position of the queue. Now the front and rear also becomes to 1.

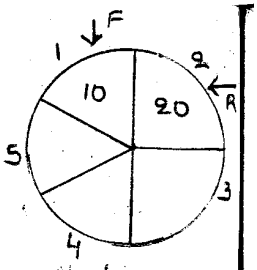


Front = 1
Rear = 1

When we are inserting the value of second element 20 in the queue the rear

(20)

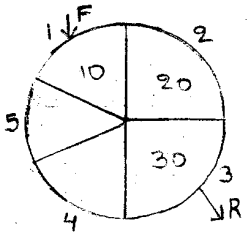
Position changes from 1 to 2 and front position remains as it is.



Front = 1

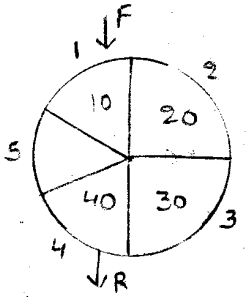
And the 3rd element 30 to the queue when we add the third element in the queue the rear position will changes from 2 to 3 and front remains at 1 only.

Rear = 2

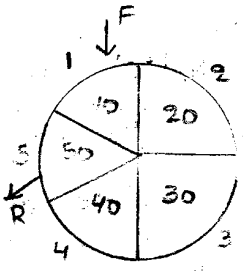


Front = 1, Rear = 3

Similarly, we add the 4th and 5th elements 40 and 50 respectively into the queue and same operations mentioned above will be held. And the rear position will shifts from 3 to 4 and then from 4 to 5. And front remains unchanged.



Front = 1
Rear = 4



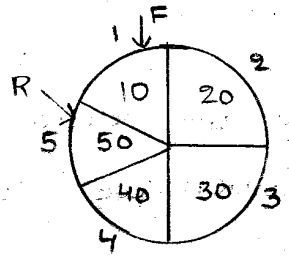
Front = 1
Rear = 5

if (Front = 1 && rear = max)
then circular queue is full.

2) Deletion:

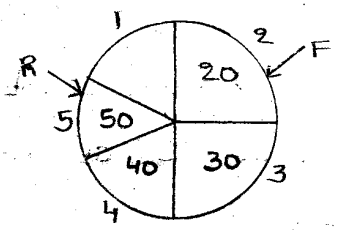
While deleting elements from the queue we adopt the principle of FIFO. When we delete the element the front position is increased by 1. So, the front changes to 2, and the rear is remains at 5.

Before Deletion



Front = 1
Rear = 5

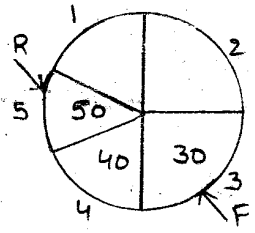
After Deletion



Front = 2
Rear = 5

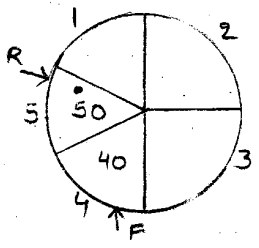
∴ 10 is deleted.

When we delete second element value 20, the front position is increased by 1. i.e, front equals to 3 and rear remains unchanged.



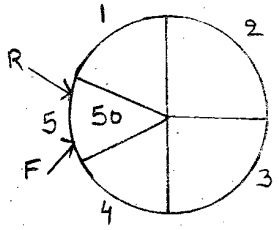
Front = 3
Rear = 5
∴ 20 is deleted.

When we delete the third element value 30, then the front position is increased by 1 i.e, front equals to 4. At this stage, rear is 5.

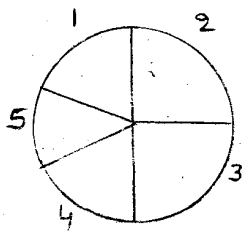


Front=4, Rear=5
∴ 30 is deleted

Similarly, when we delete the 4th and 5th elements the front position is increased by 1 respectively as above and changes to 4th and 5th. And the rear remains at 5 only in both cases.



Front = 5
Rear = 5



Front=0
Rear=0

If circular rear + 1 = front

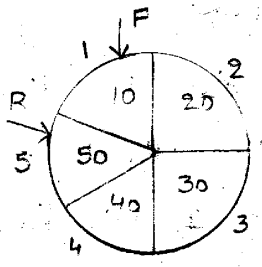
Then the circular queue is empty.

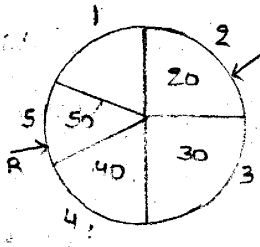
3) Reuse of deleted element / position:

If we want to reuse the deleted position when rear is at maximum position we initialise Rear=1.

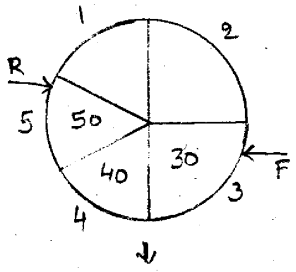
i.e, if rear=max then rear is reassigned with 1 while deleting if front=max+1 then front is initialised with 1.

Insertion:

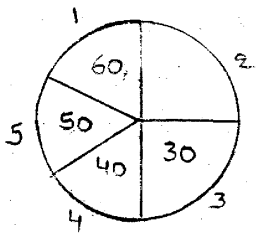




→ Deleted 10



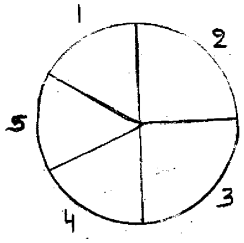
Deleted 20



→ Add 60

```
if (rear = max) then
    rear = 1
```

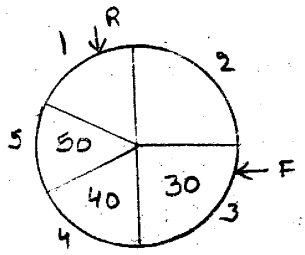
Deletion:

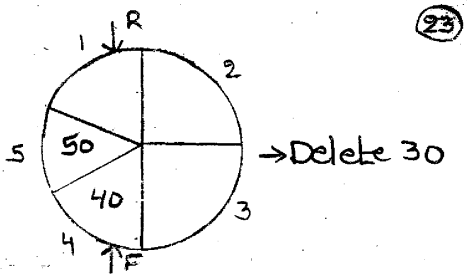


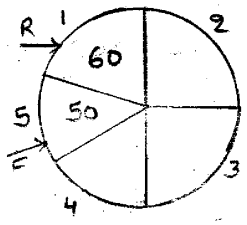
Front = 0
Rear = 0

```
if (front == 0)
```

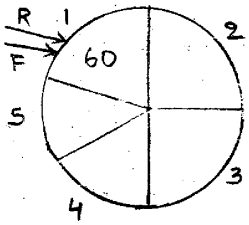
Then queue is empty.







Delete 40



Delete 50

if (front == max)

Then front = 1.

* Priority Queue:

A collection of elements in which each element can have certain priority is known as "priority queue." In this queue, the elements are deleted and processed (display) comes from the following rules

1) The highest priority element can be deleted first or more.

2) If 2 elements having equal priorities then the elements can be deleted on the orders of insertion.

3) Priority queue can be represented in 2 ways in memory. They are

1) Arrays.

2) Linked Lists.

1) Array Representation of Priority Queue:

The priority queue is represented by

(24) a 2-dimensional array and each row of 2-D array is treated as a circular queue.

Suppose if we have any row and in that row all the elements contains same priority. For example, the priority queue is represented in the form of pq[F][R].

pq[0] contains the highest priority in all elements.

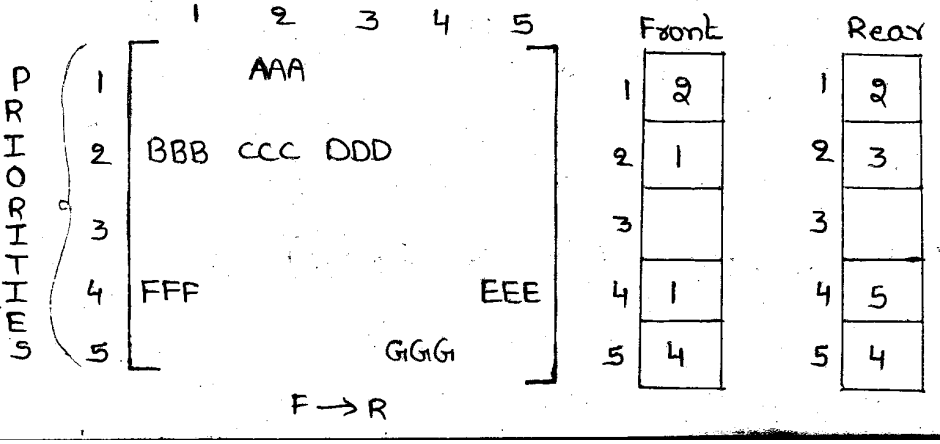
pq[1] will contains the next highest priority in those elements and so on.

For example; suppose, Front[0] and Rear[0] will indicate the front position and rear position of the priority queue which is indicated as (pq[0]) i.e; 0th row of front at rear position.

→ Front[1], Rear[3] will indicate the front and rear positions of the queue.

→ i.e; in 1st row the front is at first position and rear at 3rd position.

→ The priority queue will be represented in the following form.



* Insertion Algorithm:

The algorithm adds an item with priority number 'k' to the priority queue mentioned by a 2-dimensional array of queue.

Step 1 : Insert an item in the k^{th} row of a queue at its rear position.

Step 2 : Exit

* Deletion Algorithm:

Step 1 : The front of array front[k] != NULL.

Step 2 : Delete and process the front element in the row k of a queue.

* Linked list representation of priority queue:

Single / one-way linkedlists:

a) Each node in the list will contain 3 items of information. They are

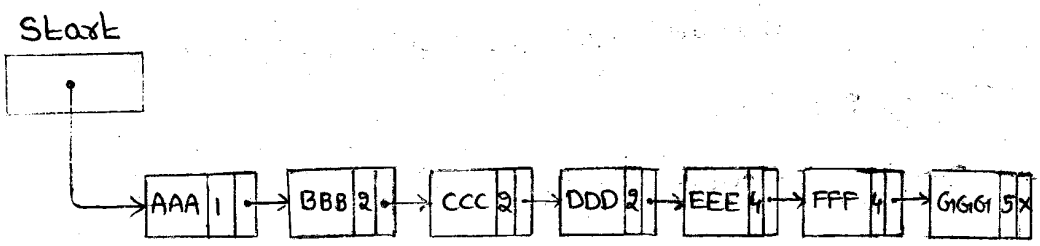
1) Information field (INFO)
2) Priority Number (PRN)
3) Linked Number (LINK)

INFO	PRN	LINK
XXX		

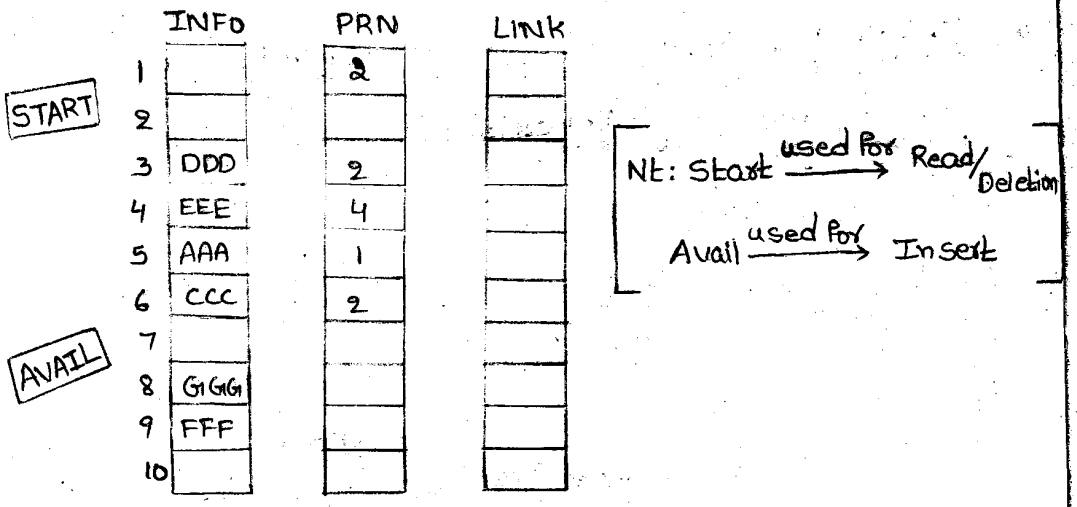
b) A node x preceeds a node y in the list when x has higher priority than y or when both have the same priority but x was added to the list before y. That means, the order in the one way lists corresponds to the order of the priority queue.

(3) The priority numbers will operate in the usual way that is, the lower is the priority number, the higher is the priority.

It will be represented in the following figure.



The above diagram shows a schematic diagram of a priority queue with 7 elements.



The above figure shows the way priority queue may appear in memory using linear arrays.

① INFO ② PRN ③ LINK

The main property of one way list representation of a priority queue is that the elements in the queue that should be processed first always appears at the beginning of the one way list.

Accordingly it is very easy to delete and process/display an element from priority queue.

* Algorithm for Deletion:

Step 1 : Set Item := INFO[START]
// this saves the data in the first node //
Step 2 : Delete first node from the list.
Step 3 : Process/Display item.
// displays the deleted item //
Step 4 : Exit

* Algorithm for Insertion:

The item will be added with it's priority number to a priority queue which is maintained in memory as a one-way list.

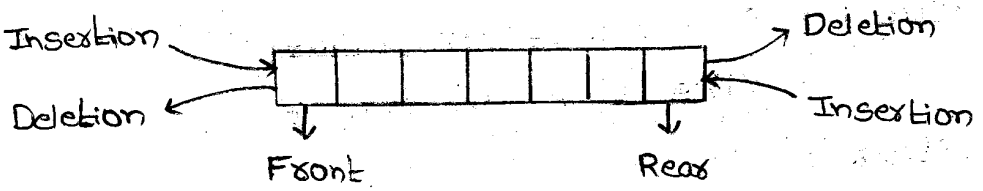
Step 1 : Traverse the one-way list untill finding a node x whose priority number x is k. Insert item infront of the node x.
Step 2 : If no such node is found insert the item as the last element of the list.

* Advantages of priority Queue:

1) The property of priority queue is a time saving system.
2) Programs of higher priority are processed first.
3) Programs with the same priority forms a standard queue.

Dequeue:

A Dequeue is a linear list in which elements can be added or removed at either ends, but not in the middle.



The term DEQUEUE refers to the name "DOUBLE ENDED QUEUE." It is a very flexible structure when compared to stacks and queues.

DEQUEUE can also be maintained in circular manner. There are various ways of representing a Dequeue. Unless it is stated or implied we assume our queue maintained by a circular array with pointers left and right.

There are 2 variations of a Dequeue.

1) Input Restricted Dequeue
2) Output Restricted Dequeue.

1) Input Restricted Dequeue:

In input Restricted Dequeue, it allows deleting the elements or nodes at both ends of the list where as it will allow to insert only at one end.

2) Output Restricted Dequeue:

The output restricted queue will allow to insert at both ends but, deletion are allowed only at one end.

* Linked list:

A linked list or one-way list is a linear collection of data elements called NODES. The linear order is given by means of pointers. Each node is divided into 2 parts.

The first part contains information of the element and the second part contains the address of the next node in the list and is called as the linkfield or Next pointer field.

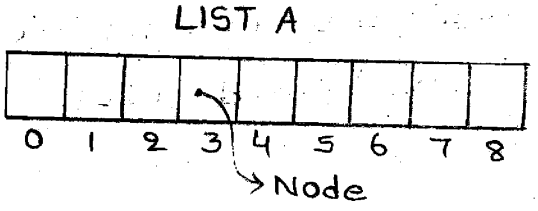


Fig (a)

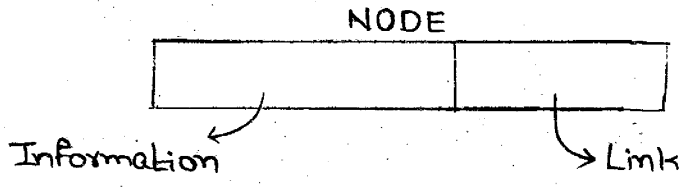


Fig (b)

The simplest way to organise the list is in the form of an array. For example, if there are 6 items in a list A then it can be represented as the above fig(a). and each node is represented as in fig(b).

An array is also referred to as a linear list because the individual elements are in sequencial location in memory i.e, the elements

are physically arranged in adjacent memory.

The array representation is useful to access a particular element by its position in arrays. This type of allocation is called as sequencial allocation.

When an array is declared it allocates fixed size of memory, that means the size of array cannot be changed at runtime. Some-times we may use less space than the allocated space. In this case, memory is wasted. To overcome with this problem we can use a structure called linked lists.

A linked list is very flexible dynamic Data Structure. Items (data) can be added to it or deleted from it dynamically. In linked lists, each item is allocated space as it is added to the list. A link (pointer) is kept with each item. to indicate the next item in the list.

Thus the adjacent elements in the list does not occupy adjacent space in the memory i.e; the elements are logically adjacent but not physically.

This type of allocation is called linked allocation. The following are the drawbacks of linked allocations.

1) We cannot access directly to a particular

element.

2) Additional memory is required to store pointers.

However, these limitations are less significant in compare to that powerful capabilities.

* Types of linked lists:

These are of 2 types, they are

1) Single linked lists
2) Double linked lists.

A Single linked list is a linear collection of data items. The linear order is given by means of pointers. These types of links are often refer as "Linear Linked Lists."

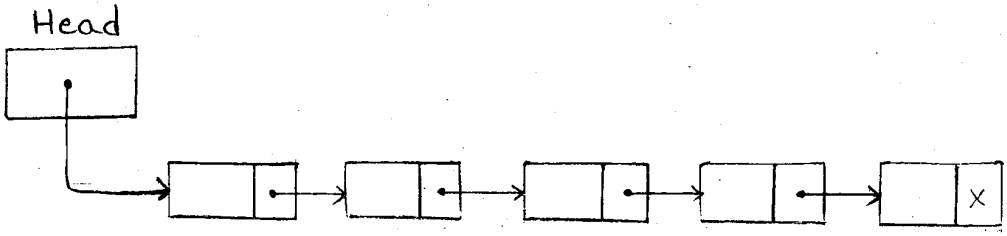
Each item in the list is called as a node and a collection of nodes are called "linked Lists." Each node of the list has 2 fields.

* Information field.
* Link field / Next pointer field.

Information field contains the data of any type i.e; int, char etc being stored in the list.

Linked field / Next pointer field contains address of next data in the list. The last node in the list contains a null pointer to indicate that it is the end of the list.

The following figure shows a linkedlists with 5 nodes.



In the above figure each node is pictured with 2 parts. The left part represents the information part of node which may contain data items.

Eg: name, address, salary ----

The right part represents the next pointer field of the node and there is an arrow drawn from it to the next node in the list. The pointer of the last node contains a special value which is called the "NULL pointer."

The null pointer is denoted by [X] shown in the diagram to signal the end of the list.

In actual practice zero or a negative number is used for the null pointer.

A linked list is also contains a list pointer variable called "Head" which contains the address of the first node in the list. We need this address at the beginning/start to trace through the list. If the list has no nodes then it is called as "Null list" or "empty list" and is denoted by Null pointer in the variable Head.

i.e., Head = null or Head → NULL

As items are added to a list memory for a node is dynamically allocated.

A single linked list is the simplest example of a class of datastructures refer to as self referencial structure. It will be shown in the following form.

Simple self referencial structure:

```
Struct node
{
  int info;
  Struct node * ptr;
}
```

* Operations of lists:

There are 3 operations to perform on lists. They are

1) Traversing
2) Insertion
3) Deletion.

1) Traversing the list:

Accessing the data from each node untill the end of the list is called traversing the list.

2) Insertion:

usually, there are 3 cases for insertion. They are

a) At the beginning of the list.

b) At the end of the list.

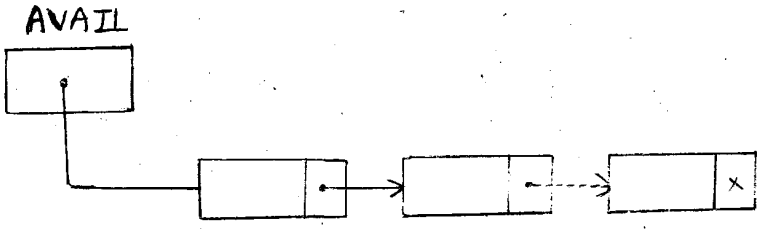
c) At the middle of the list. [Insertion after a specified node].

3) Deletion:

We can remove an element from the list without destroying the integrity of the list itself (without destroying the linking process).

After a series of memory allocations and deallocations there are blocks of free memory scattered through out the available space inorder to reuse this memory a special list called "AVAIL LIST" or "free pool" is maintained.

An external free list pointer refered as "AVAIL". It points to the first block in the free list. It will be shown in the below figure. When a new block of memory is requested the memory allocation algorithm will scan the free list.



Free storage list.

* Traversing a linked list:

Each node in a single linked lists contains a pointer to the next element. Hence they can

only be traverse in one direction from starting to ending. We cannot traverse in a backward direction in this list.

Algorithm:

// current pointer is a temporary pointer variable, head is a pointer variable pointing to the first node of the list. //

Step 1 : current ptr ← Head.

Step 2 : While current ptr ≠ NULL do.

Begin.

Step 3 : Print the information in the field info of current ptr.

Step 4 : current ptr ← link

end.

* Inserting element into a list:

When an insertion is performed on a list it is difficult to adjust all the pointers properly once a single linked list pointer points to the corong place then the entire list will become worthless.

The insertion can be done in three places:

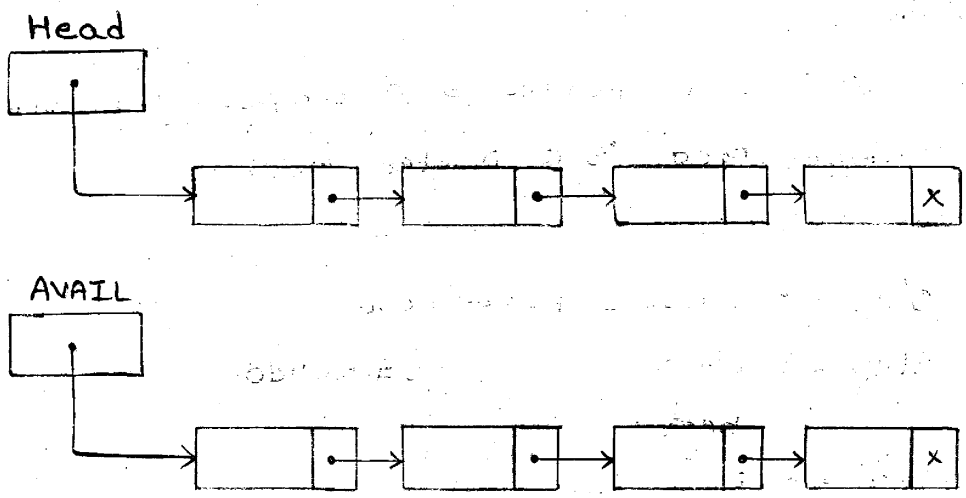
1) At the beginning of the list.
2) At the end of the list.
3) At the middle of the list.

1) Inserting at the beginning of the list:

Suppose a node 'N' is to be inserted

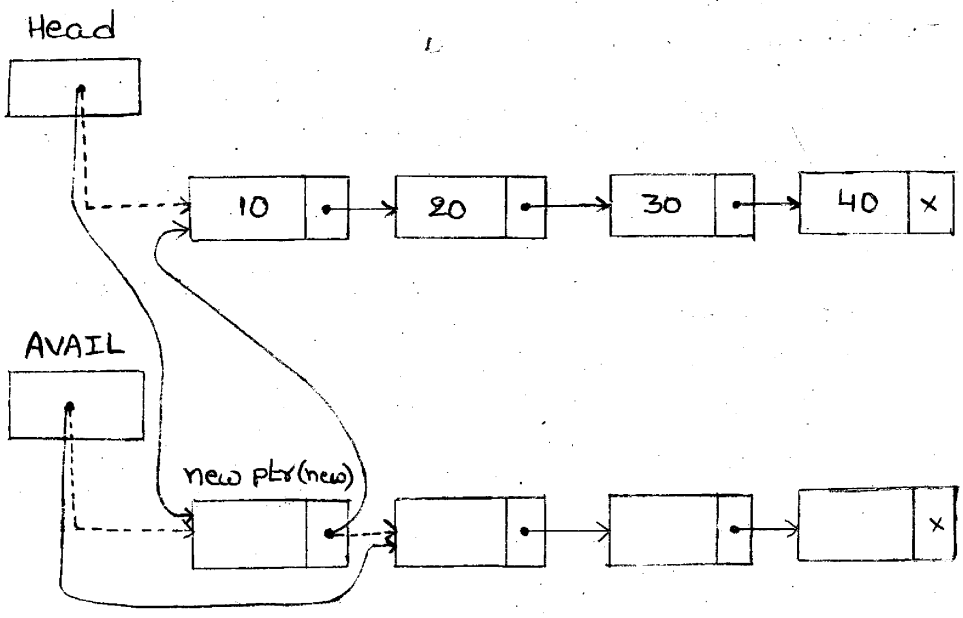
into the list at the beginning is shown in the following figure.

Before Insertion



Free memory list

After Insertion



The simplest way for inserting any item at the beginning of the list is

* Allocates space for a new node.
* Copy the inserting item into that node.

* Make the new nodes next pointer point to the current head of the list.
* Make the head of the list point to the nearly allocated node.

Algorithm:

```
Step 1 : If AVAIL = NULL then
         begin
            Print "space is not available in free
            memory"
            return
         end
Step 2 : New ← Avail.
Step 3 : Avail ← ptr(Avail).
Step 4 : Info(new) ← Item.
Step 5 : ptr(new) ← head.
Step 6 : Head ← new.
Step 7 : Exit.
```

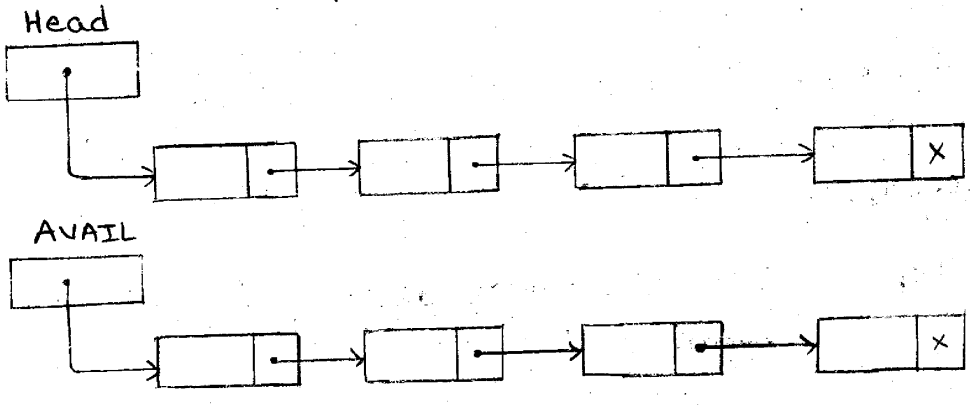
In the above algorithm head is a pointer variable pointing to the starting node and NEW variable is used to add a new node.

2) Insertion at the end of the list:

If we want to insert an element at the end of the list the following conditions should be adopted.

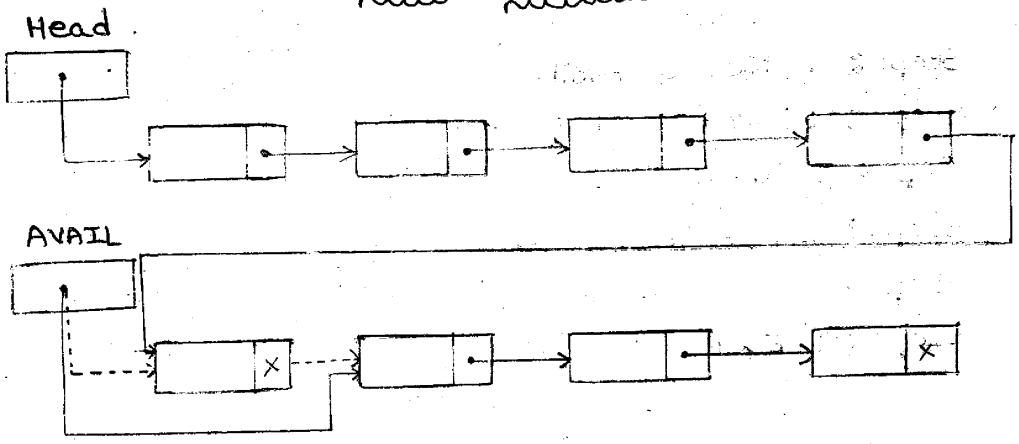
* If the list is empty then the new node becomes the first node itself.
* If the list is not empty then it adds at the end of the list and the node address field (ptr) will be assigned "NULL".

Before Insertion



Free memory list

After Insertion



Algorithm:

```
Step 1 : If AVAIL = NULL then
         begin
            Print "Space is not available"
            return
         end
Step 2 : New ← Avail
Step 3 : Avail ← Ptr[Avail]
Step 4 : Info[new] ← Item.
Step 5 : Ptr(new) ← Null
Step 6 : If Head = Null then
         begin
            Head ← new
         end
```

Step 7 : else

Step 8 : begin

Temp(ptr) ← Head

end

Step 9 : While ptr(temp ptr) ≠ NULL do

Tempptr ← ptr(temp ptr)

Step 10: Ptr(Tempptr) ← new.

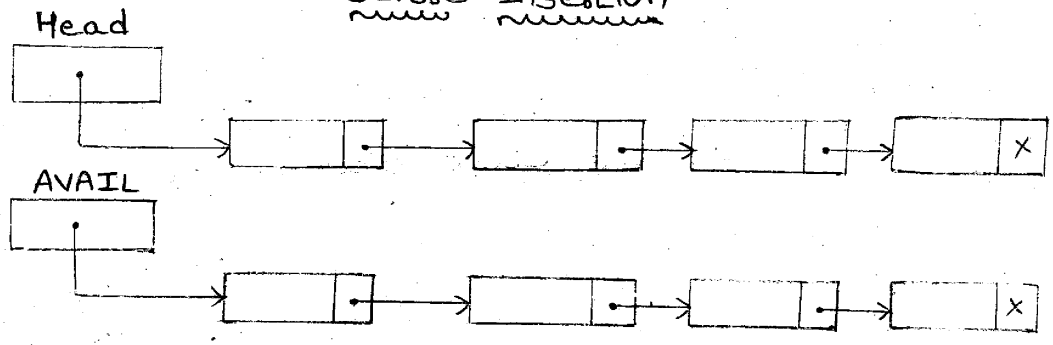
Step 11 : Exit.

3) Inserting at the middle of the list :

Suppose if we want to insert an element 'N' in a list after a specified node A.

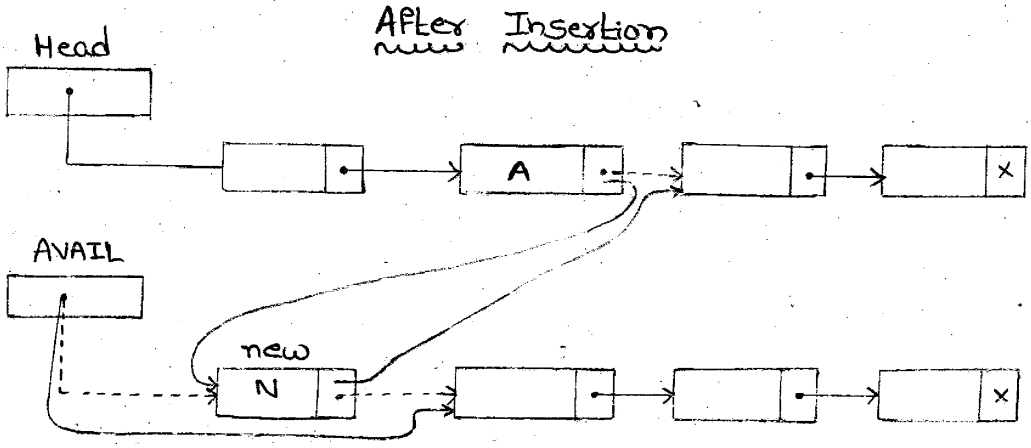
Let P be a pointer of A. If we want to insert at the beginning then P is NULL, list is empty. Otherwise P can be points to any node. This can be shown in the bellow figure.

Before Insertion



Free memory list

After Insertion



(b) Algorithm:

```
Step 1 : If AVAIL = NULL then
         begin
             Print "Space is not Available"
             return
         end
Step 2 : New ← Avail
Step 3 : Avail ← ptr(Avail)
Step 4 : info(new) ← item
Step 5 : If P = NULL then
         begin
             ptr(new) ← head
             head ← new
         end
Step 6 : else
         begin
             ptr(new) ← ptr(p)
             ptr(p) ← new
         end
Step 7 : Exit
```

In the above algorithm head is the pointer variable pointing to starting node and new variable is used to add new node.

'p' is the location of node A in a linked list. If the node is to be inserted at first position then we take P=NULL.

Otherwise, we assume P is pointing to node A in the list.

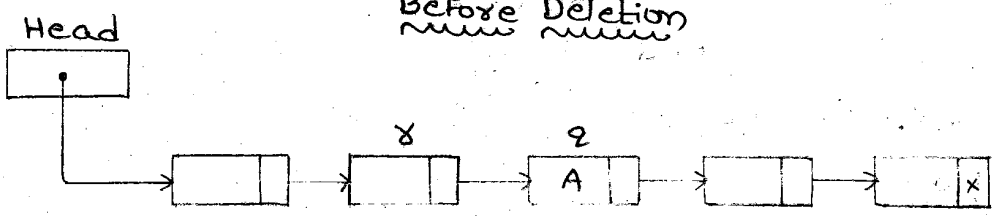
* Deletion from a list:

To delete a list element in a

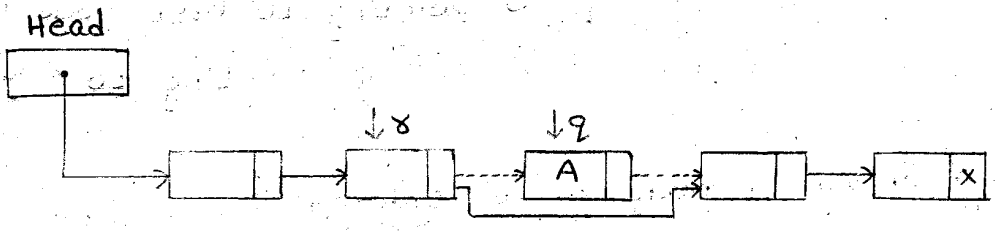
Single linked list the following procedure is to be adopted.

The deleted node of next node address is assigned to the deleted before node address field and the deleted node is added to the free list at the beginning as shown in below figure.

Before Deletion



After Deletion



In the above figure we have taken 2 pointer variables 'q' and 'r'.

q is pointing to the deleting node 'A' and r is pointing to the before node of deleting node.

Algorithm:

```
Step 1 : If Head=NULL
         begin
             print "List is empty"
         return
         end
Step 2 : If r=null then
         begin
             q ← Head
             print "Deleted node is info(q)"
```

Head ← ptr(Head) or ptr(q)

end.

Step 3 : else

begin

ptr(r) ← ptr(q)

Print "Deleted element is", info(q)

ptr(q) ← Avail

Avail ← q

end

Step 4 : Exit

Note:

If we want to delete the first node in a linked list then 'q' is pointing to first node then r=null that means r is not pointing to any node.

* Inserting a node in an ordered list (sorting):

Suppose, if we want to insert an element into a list whose elements are in a sorted order. If the list is empty then the new node becomes the first node and now the list contains only one node.

Otherwise, set the suitable place for new node to be inserted. While searching all preceeding elements are less than or equal to the new element and the succeeding elements are greater than the new element.

Algorithm:

Step 1 : If AVAIL = NULL then

```
        begin
            printf "space is not available in free
                memory list."
            return
        end
Step 2 : new ← Avail
Step 3 : Avail ← link (avail)
Step 4 : Info (new) ← item
Step 5 : if (head = null) then
        begin
            link (new) ← head
            head ← new
        end
Step 6 : else
            if (info (new) <= info (head))
            begin
                link (new) ← head
                head ← new
            end
Step 7 : else
            begin
                q ← head
                r ← head
                while (link (q) ≠ null) and info (q) <= info (new)
                begin
                    r ← (q)
                    q ← link (q)
                end
            end
Step 8 : if info (q) > info (new)
            begin
                link (new) ← link (r)
```

```
        link(r) ← new
    end.
```

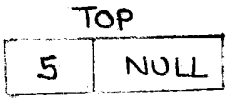
Step 9 :

```
else
    begin
        link(new) ← link(q)
        link(q) ← new
    end.
```

Step 10 : Exit.

* Stack operations using linked list (Linked Stack):

If the list is empty then the new node becomes the first node and the list contains only one node. Otherwise the new node is added before the existing list. Suppose, if we want to push 5 into the linked stack, if the list is empty then this is taken as 1st node.



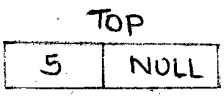
Again if we coant to add another element like 10 then it will be added at the beginning of the existing list.

Now the top pointer variable is moved to the starting node (newly inserted node). Likely wise every new node is added to the existing list.

In the linked stack the pop operation is started from the top pointer and after deleting the node the top pointer variable is

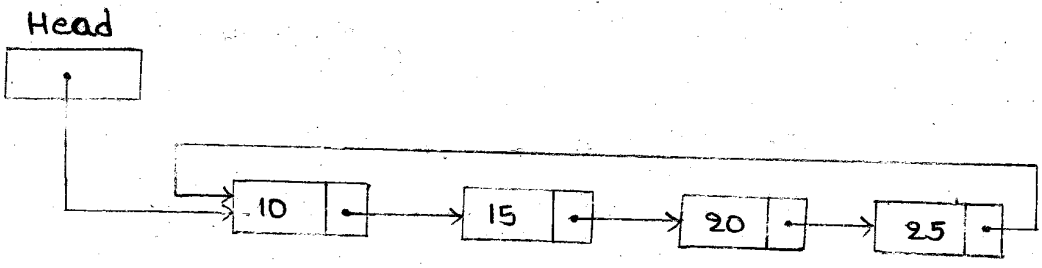
moved to the next node.

Eg: Suppose if we want to perform the pop operation on the above diagram then the top pointer variable points to the second node.



* Circular linked list:

The list in which the last node always points to the first node is called the circular linked list. This type of list can be constructed / built by just replacing the null pointer which points to the first node. The following figure shows the circular form of the list.



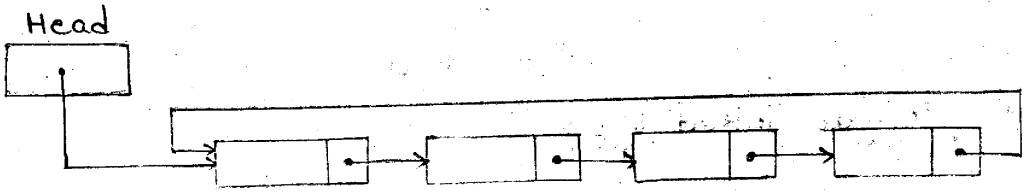
Actually, there is no first or last node. Since, the structure is circular. However the header pointer continue to represent the list by pointing to the first node.

Advantages of linked lists:

* Any node can traverse from starting to any other node in the list.
* There is no need of null pointer to signal / indicate the end of the list. Hence, all pointers will contain a valid address.

A circular list can be implemented

with or without a header node. If the header node is present in the list then the last node of the list points to the header node. This type of list after refered as header circular linked list.



Header circular linked list.

* Operations on circular linked lists:

1) Insertion:

The insertion of a new node is to an existing linked list can be done in 3 ways.

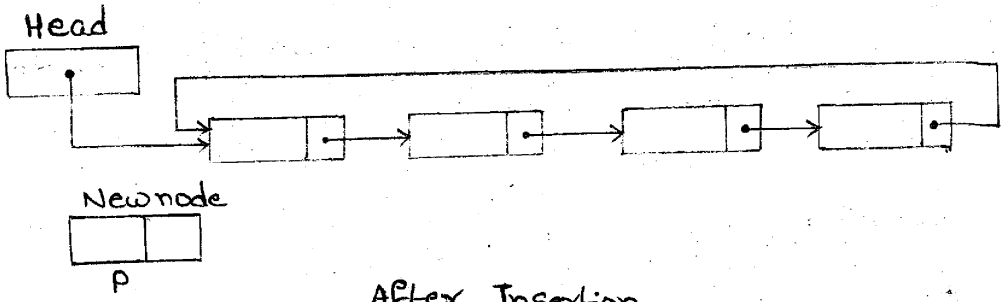
a) At the beginning of the list.

b) At the middle of the list.

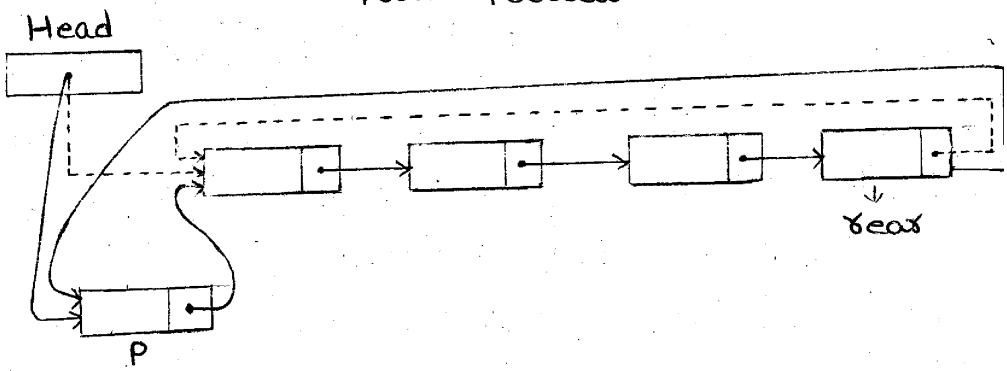
c) At the end of the list.

a) Inserting at the beginning of the list:

Before Insertion



After Insertion



Algorithm:

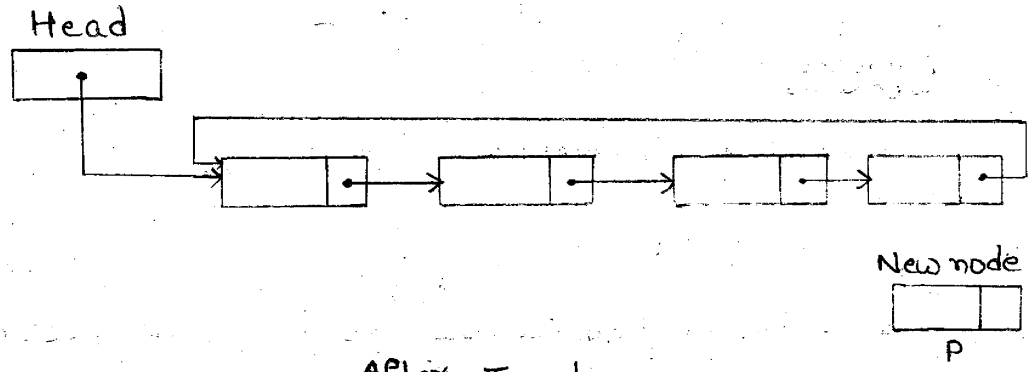
P → next ← head

head ← P

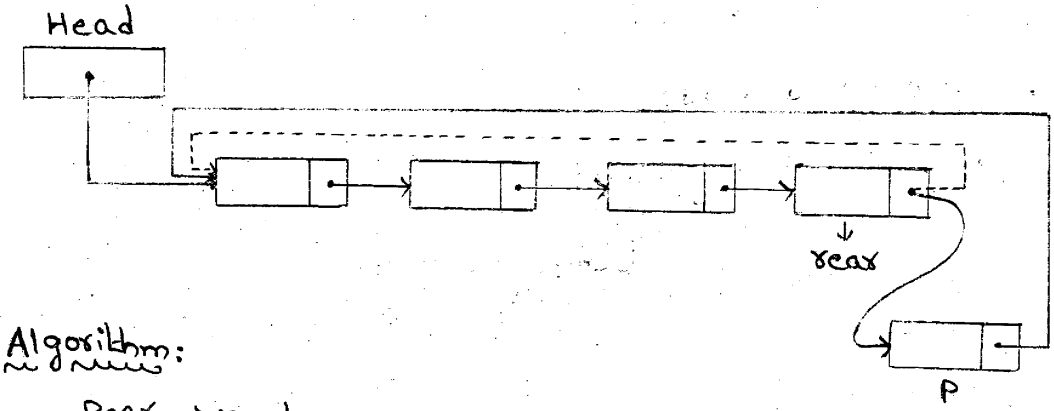
rear → next ← P

b) Inserting at the end of the list:

Before Insertion



After Insertion



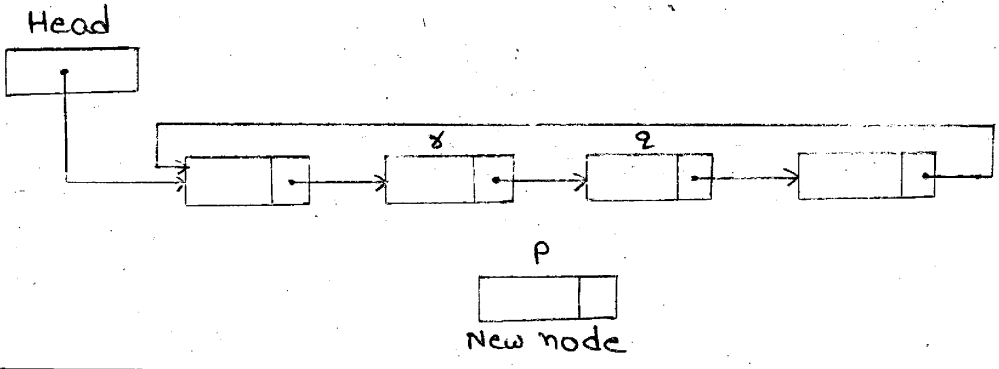
Algorithm:

Rear → next ← new

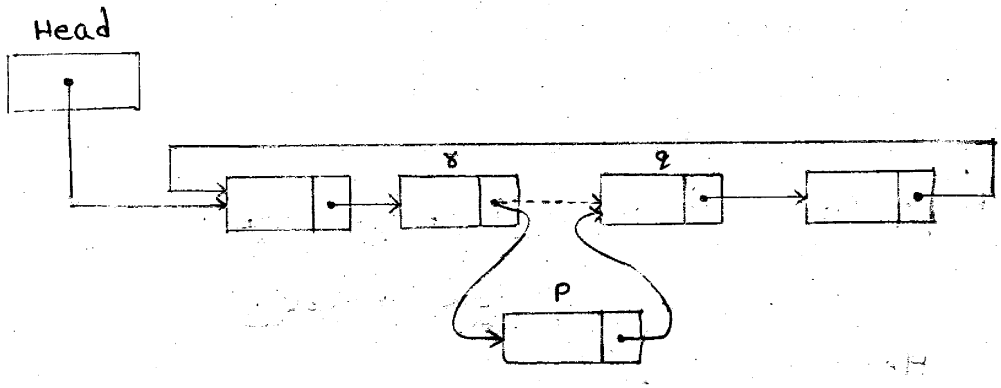
new → next ← head.

c) Inserting at the middle of the list:

Before Insertion



After Insertion



Algorithm:

P → next ← r → next

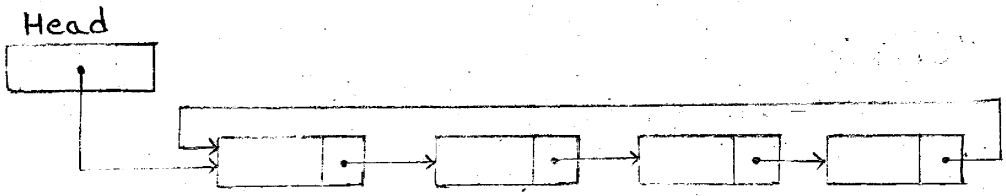
r → next ← P

In the above diagram P is a new node and r is pointing to the inserting node position before the new node, and q is pointing the inserting position after the new node.

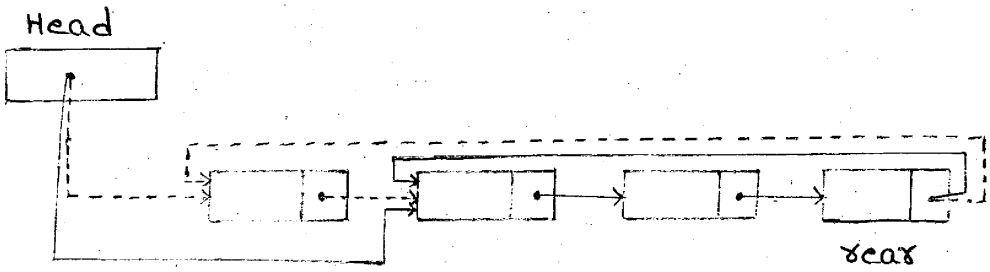
* Deleting a Node:

1) At the beginning:

Before Deletion



After Deletion



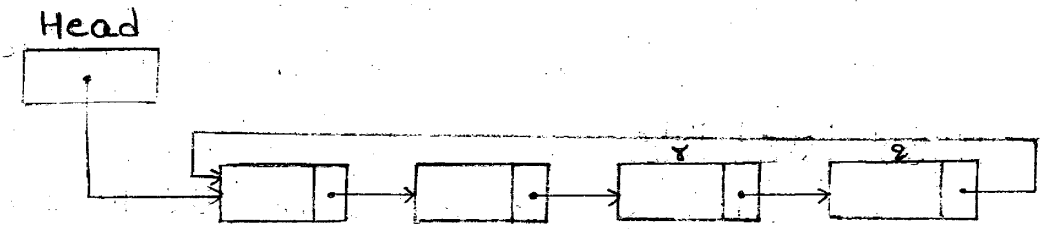
Algorithm:

head ← headnext / topnext

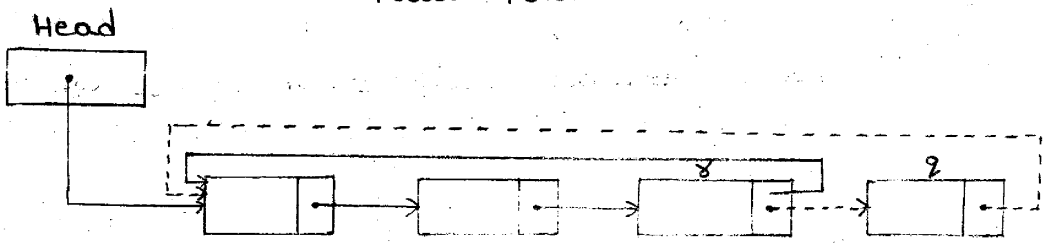
rear → next ← head

2) At the end of the list:

Before Deletion



After Deletion



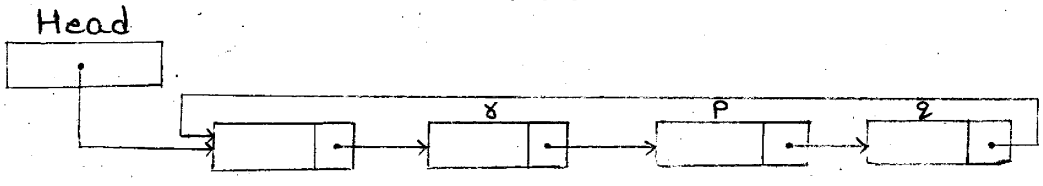
Algorithm:

r → next ← head.

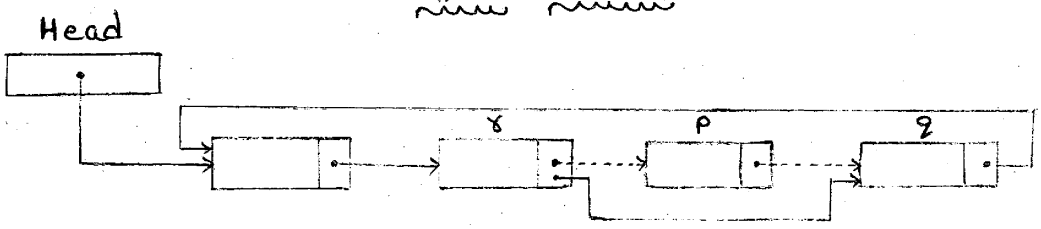
In the above diagram, q is the last node and r is just before the q or last node.

3) At the middle of the list:

Before Deletion



After Deletion



(30) Algorithm:

$r \rightarrow next \leftarrow P \rightarrow next$

(or)

$r \rightarrow next \leftarrow q$

In the above diagram P points the deleting r points to the before node of deleting node and q points to the next node of the deleting node.

* Procedure for circular linked list:

Head and Rear are the pointer variables of the structure.

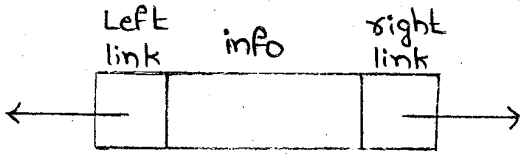
```
# include <stdio.h>
  Struct node
  {
   int data;
   Struct node *next;
   *head=null, *rear=null;
  }
  Void create(int x)
  {
   Struct node *p;
   P=(Struct node *) malloc (Size of (Struct node));
   P→data=x;
   if (head==null)
   {
    head=p;
   }
   else
   {
    rear→next=p;
   }
   rear=p;
   rear→next=head;
  }
```

* Double Linked lists:

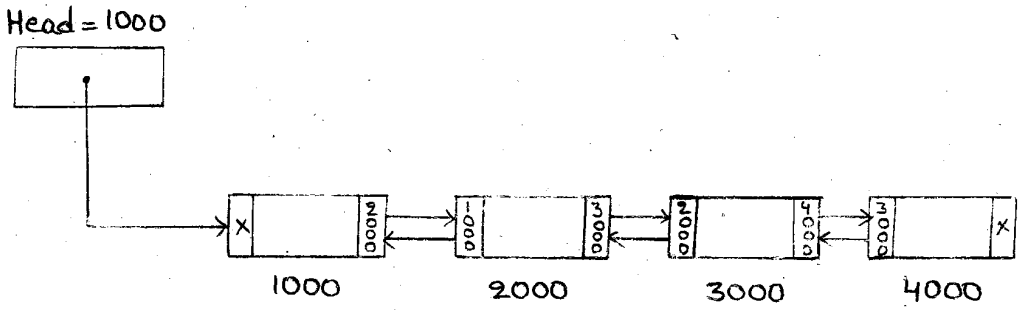
A Single linked lists has the disadvan-tage that we can only traverse in one direction (i.e; forward direction) but many applications required for searching backwards and forwards through Sections of a list or within the entire list.

For example, Searching for a common word in a telephone directory would probably need much scanning backward and forward through a small region of its whole list. So the backward list becomes very useful.

The useful requirement of a single linked list is wayed a path for creating double linked list. The difference between the two linked list is that the Single linked list have pointer going in only one direction. Where as the double linked list will consist the pointers going both to next element and previous element in the list.



Double Linked List Node



Double linked list with 4 nodes.

The main advantage of double linked list is that they permit traversing of a list in both directions that means giving a pointer to any list element it is possible to get any other elements on the list.

In double linked lists to travel to the end of the list we simply follow the next pointer (right pointer) and to move to the beginning of the list i.e; backward direction we follow the previous pointer (i.e; left pointer link).

The structure of double linked list is as follows

```
Struct node
{
  Struct node * leftptr;
  int data;
  Struct node * rightptr;
}
```

When compared to a single linked list an extra pointer is adds to the double linked lists and also occupies addiontional space. At the same time, it also increases the maintanance of insertions and deletions because there are more pointers to edges.

In the double linked list, each node contains 3 fields / parts. The first part indicates the previous node address and second part indicates the information or data. Third part indicates next node address.

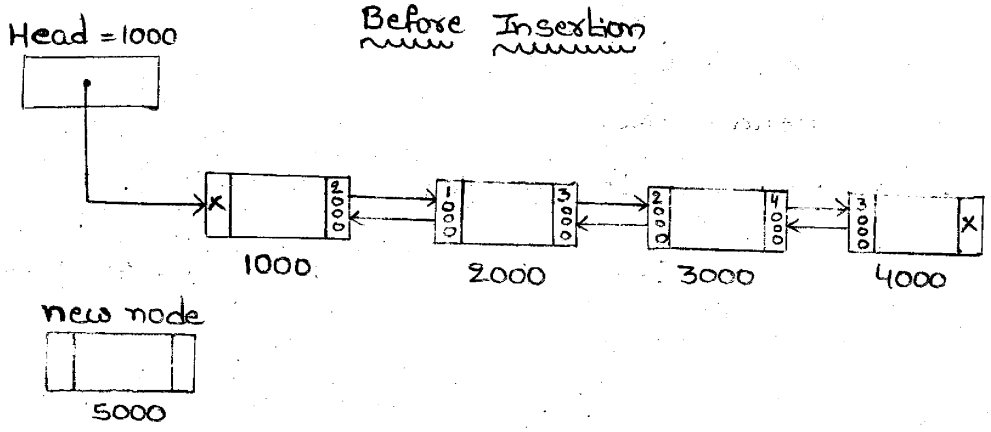
* Operation on Double linked lists:

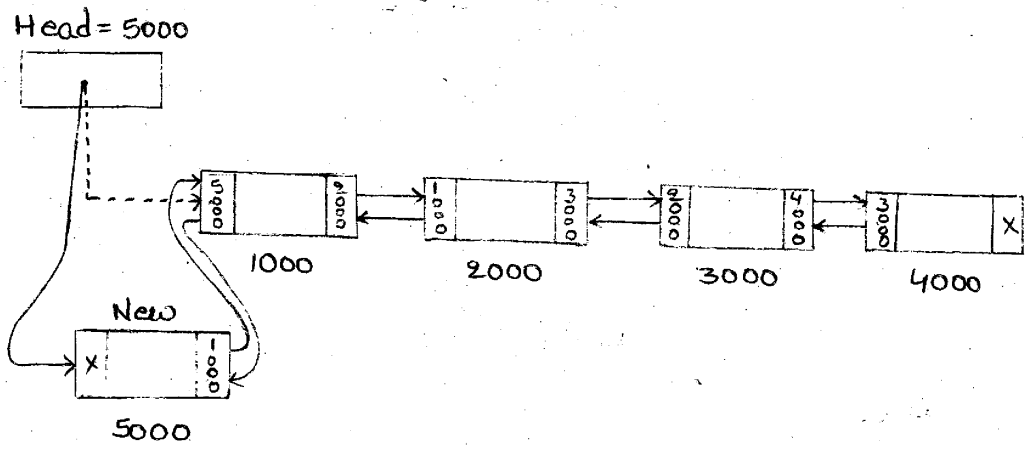
1) Insertion of an element / node into a double linked list:

The insertion can be done in 3 ways they are:

a) Insertion at the beginning.
b) Insertion at the middle.
c) Insertion at the end.

a) Insertion at the beginning:





The new node can be inserted before the first node using the following steps.

Step-1 : Adjust the new node pointer.

Step-2: First node address is assigned to new node right pointer.

Step-3: The new node address is assigned to starting (or) First node left pointer.

Step-4: The first node pointer variable will be adjusted. It is shown in the above figure and its algorithm is as follows.

Algorithm:

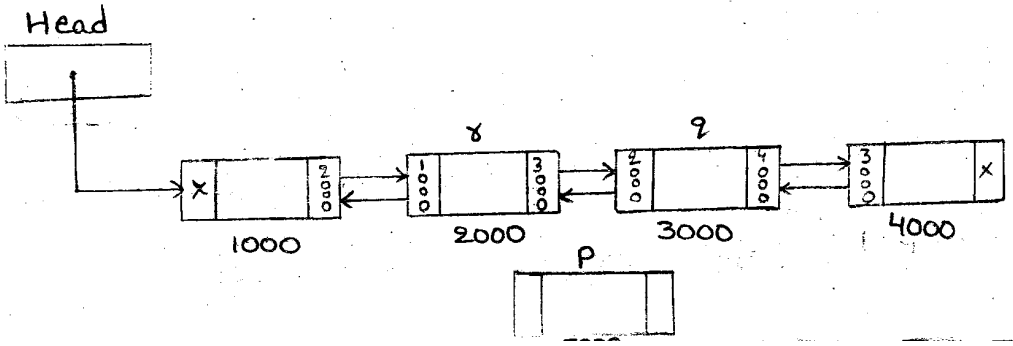
new → rptr ← head
head → lptr ← new
new → lptr → null
head ← New

b) Insertion at the middle:

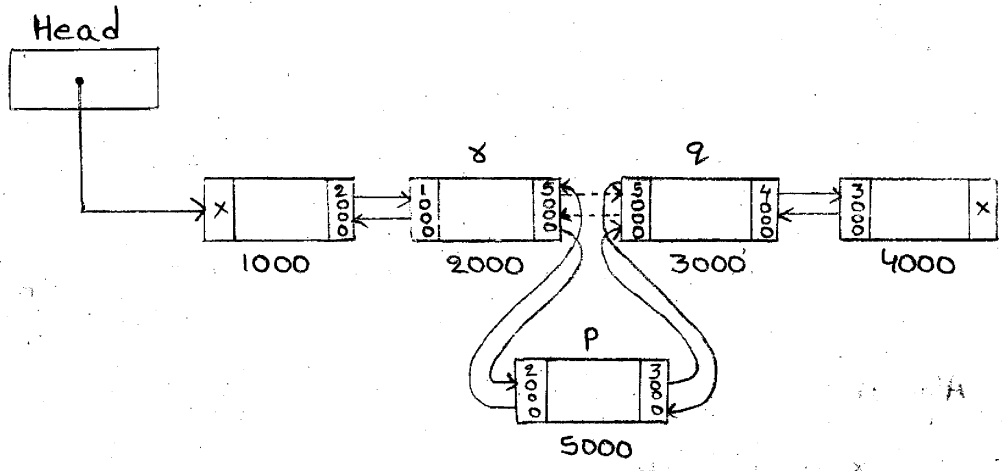
If the new node is to be inserted in the middle then we represent the new node with p and its previous node with r and its next node is represented with q.

When we want to insert p in the middle of the list (in between r and q) and assign the address of r → rptr to p ← rptr and assign address of p ← rptr and assign address of p → lptr ← q → lptr, finally address of q → lptr ← p.

Before Insertion



After Insertion



Algorithm:

$P \rightarrow rptr \leftarrow r \rightarrow rptr$

$r \rightarrow rptr \leftarrow P$

$P \rightarrow lptr \leftarrow q \rightarrow lptr$

$q \rightarrow lptr \leftarrow P$.

c) Insertion at the end:

When a new node is inserted at the end of the list the following steps are adopted.

Step-1 : We assign the p address to $r \rightarrow rptr$

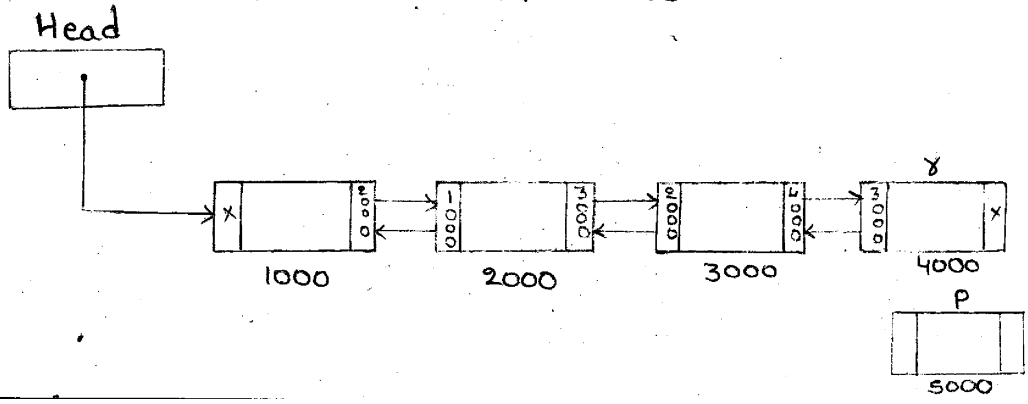
i.e, $r \rightarrow rptr \leftarrow p$.

Step-2 : Assign the address of r to $p \rightarrow lptr$ i.e,

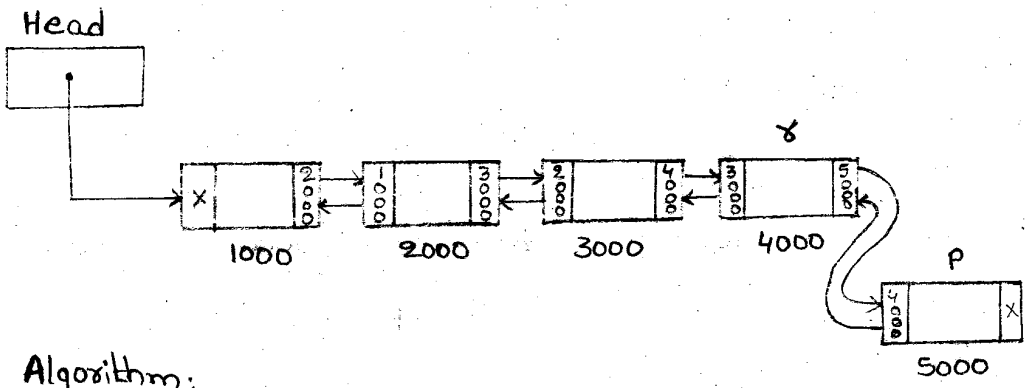
$p \rightarrow lptr \leftarrow r$.

Step-3 : Assign the address of $p \rightarrow rptr \leftarrow null$.

Before Insertion



After Insertion



Algorithm:

r → rptr ← P

P → lptr ← r

P → rptr ← null

* Deletion of Double Linked Lists:

If the list is empty and we are trying to delete an element then it should print the appropriate message informing that the list is empty.

If there is only one node in the list then delete that node and assign "null" to head. If the list has more than one nodes then assign the pointer variable of the next node to the previous node right pointer and the previous node address to the next node left pointer.

It has 3 types for deletion, they are

1) At the beginning
2) At the middle
3) At the end

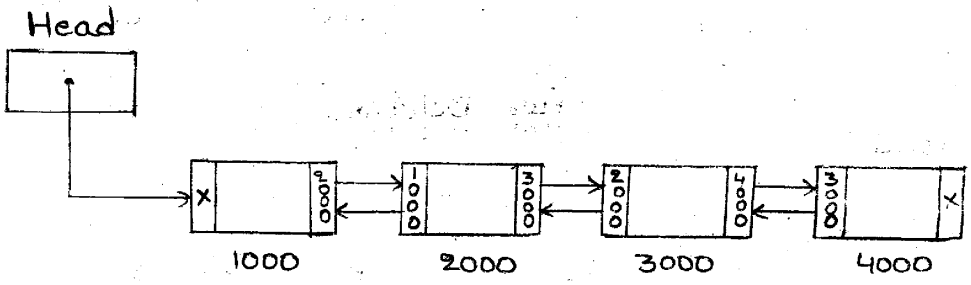
1) Deletion at the beginning:

If we want to delete the beginning of the node the head pointer variable moves to

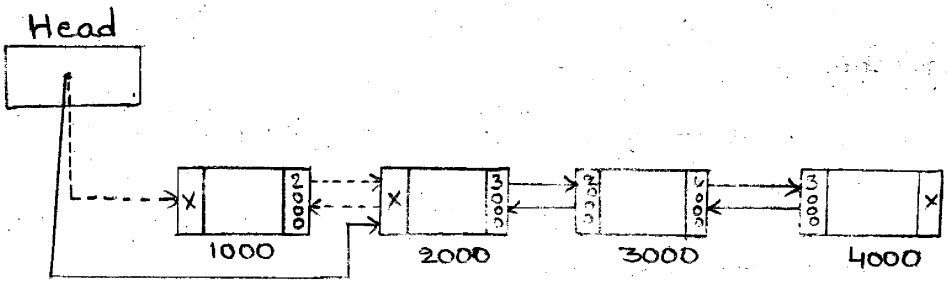
the next node and null is assigned to head left pointer.

It is shown in following figure.

Before Deletion



After Deletion



Algorithm:

head ← head → rptr

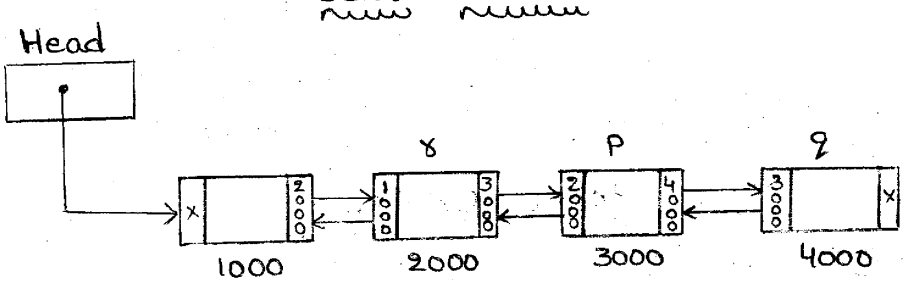
head → lptr ← null.

2) Deletion at the middle:

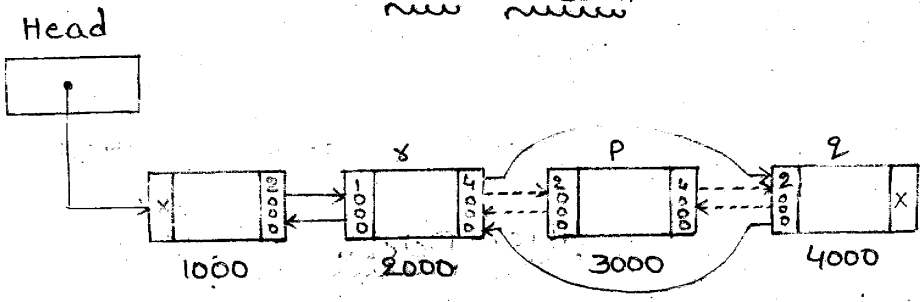
If we assume that the pointer variable P is pointing to deleted node, the r Pointer variable is pointing to the before node of deleting node and the q pointer variable is pointing to the next node of the deleting node.

When we delete the p node, we assign q to r right pointer and r is assigned to q's left pointer.

Before Deletion



After Deletion



Algorithm:

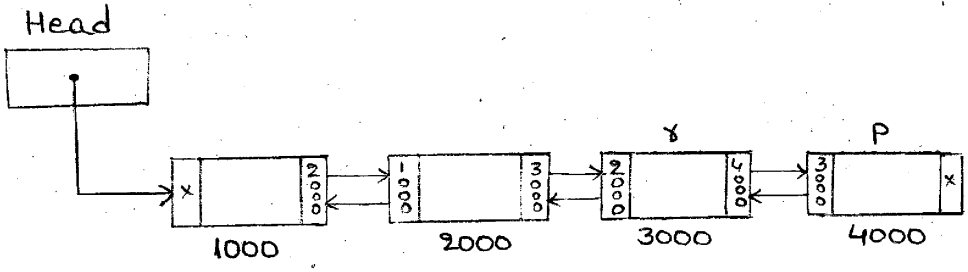
$x \rightarrow rptr \leftarrow q$

$q \rightarrow lptr \leftarrow x$.

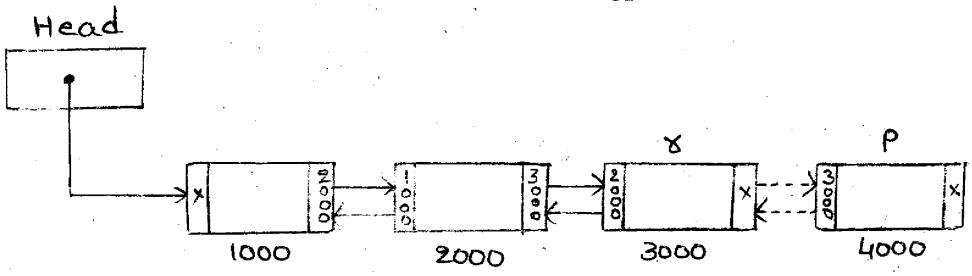
3) Deletion at the end:

If we want to delete a node at the end of the list then assign the null pointer to the 'x' right pointer which is shown below.

Before Deletion



After Deletion



Algorithm:

x → xptr ← null.

* Header Linked Lists:

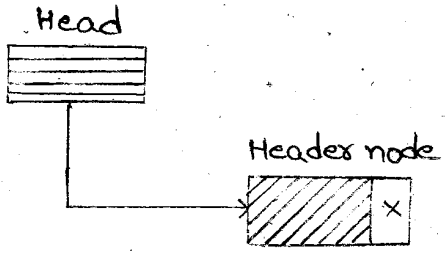
It is a linked list which always contain a special node at the beginning of the list called "header node."

This node does not hold any information in the datafield, and this node acts as a "dummy node."

The datafield of the header node is shaded to indicate that it does not contain any meaningful information. In the header linked list the header node address field always points to the first logical node.

There are distinct advantages in using a header node. specially at the time of processing an empty list. In the linked list, the front pointer variable / header pointer variable should be made null and the list becomes empty.

But, in case of header linked lists there is no need to set header variable to null. Since it always points to the header node that means even an empty list must contain this extra node.

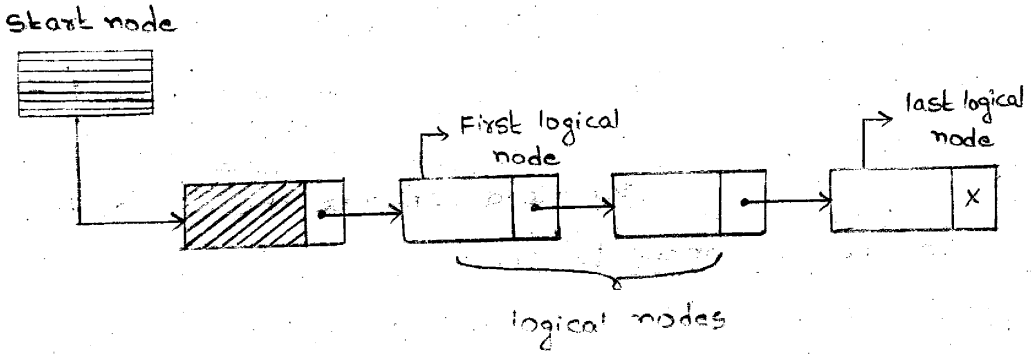


There are two header linked lists. They are:

1) Grounded Header List
2) Circular Header List.

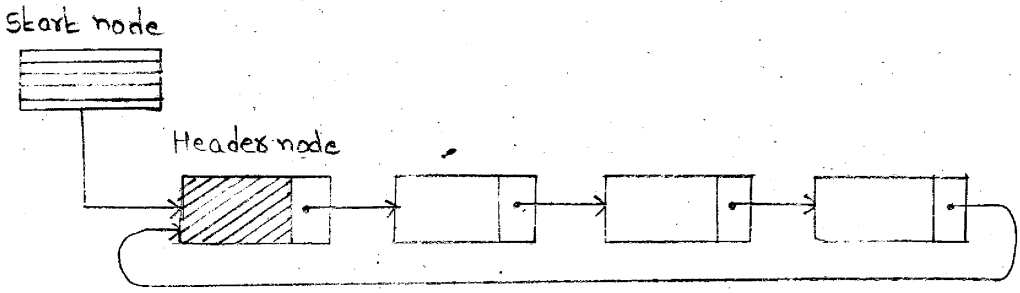
1) Grounded Header List:

Grounded Header list always contains a null pointer at the last node.



2) Circular Header List:

Circular Header list is a header list where the last node points back to the header node.



UNIT-2

Trees

* **Definition of Tree:**

Tree is a connected graph without any circuits. The tree can be used in compiler constru-ctions, operating system design and so on.

Tree is sometimes called as "Inverted Tree"

General Tree:

It is a collection of 'n' elements. When $n>0$ one of the element is designed as a "ROOT". And the remaining elements are partitioned into disjoint subsets. Each subset itself is a tree.

(v.imp)

* **Binary Tree:**

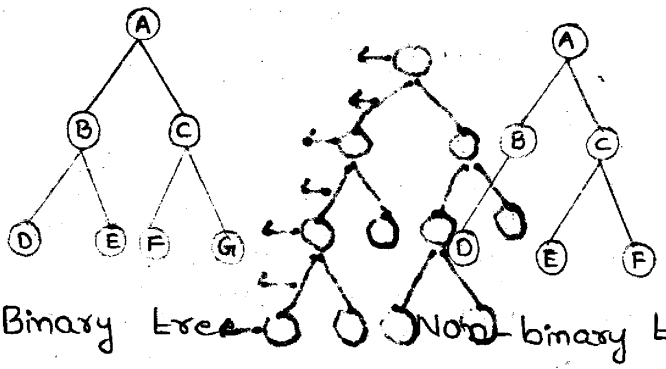
It is an ordered collection of 'n' (where $n \geq 0$) elements.

i) When $n=0$, it is called empty tree.

ii) When $n>0$, one of the element is taken as root and the remaining elements are partitioned into two disjoint subsets.

Each subset is a "Binary Tree".

Eg:



Binary tree Non-binary tree.

In the above tree, A is the parent of B,C and B,C are the childrens of A.

Parent nodes are called preceeders of children node and children nodes are called "successors of parent.

If the children having same parent then the children are called Siblings.

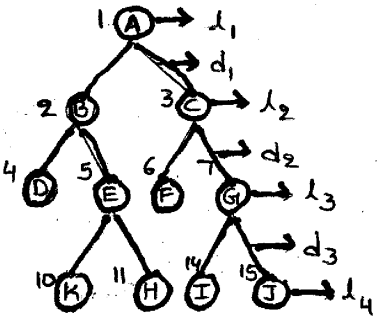
* Representation of Trees in memory:

Trees can be represented in memory in two ways they are:

1) Array Representation.
2) Linked Representation.

1) Array Representation of binary tree (or) Sequencial representation of binary tree:

* The Sequencial representation uses 1-D array the root of binary tree is stored in the first location of array.
* If a node is in j^{th} location, of an array then its left child is stored in the location $2*j$ and its right child is stored in the location $(2*j)+1$.
* The maximum size that is required for an array to store in a tree is $2^{d+1}-1$.



level = 4
Depth = 3

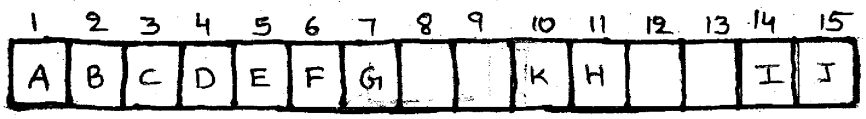
$$2^{d+1}-1 = 2^{3+1}-1$$

$$= 2^4 - 1$$

$$= 16-1$$

$$= 15$$

∴ Memory location = 15



* Advantages of Trees:

* NO overhead of maintaining pointers.
* For a specified child the parent can be easily computed. If the child node is in the j^{th} location then its parent node is in $\frac{j}{2}^{th}$ location.
* If you want to find c's parent in the above then 'c' is in 3rd position. So its parent is in $\frac{3}{2}^{th}$ location i.e; 1. So the c's parent A is at location 1.

Disadvantages:

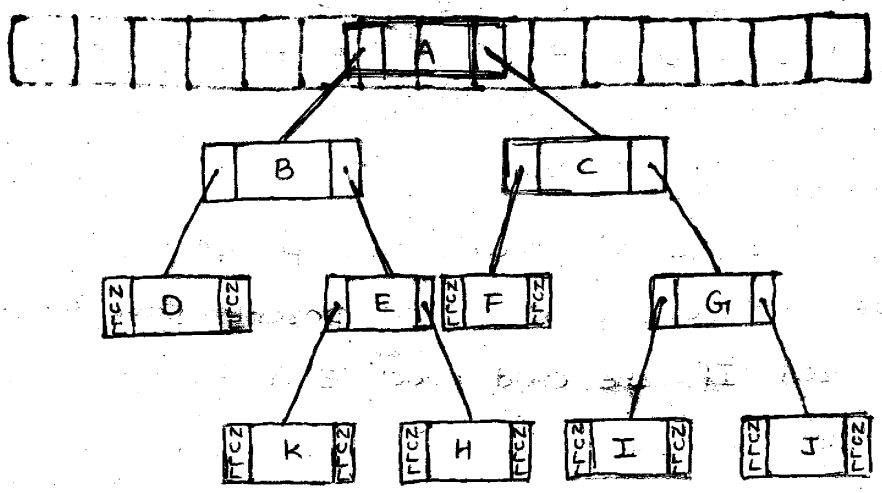
* The arrays are linear data structures. Thus, growing and decreasing of a tree cannot be efficiently managed.
* In the array representation, the number of memory locations are not filled. So the memory is wasted.

2) Linked Representation of a binary Tree:

Linked representation of tree in memory is represented using pointers. Since each node in a binary tree can have max 2 children. The linked representation of node has 2 pointer fields. Those are left child and right child (like

double linked list).

If a node does not have any child that corresponding pointer field is made as a null pointer, as shown in the following figure.



A binary tree with 'n' nodes contains (n+1) null pointers called linked representation binary tree.

* Wastes space due to null pointer.

* Identifing a parent node is difficult.

Binary tree structures are very useful for processing arithmetic expressions because they contains binary operands.

* Binary Search Tree/Sort Tree (BST):

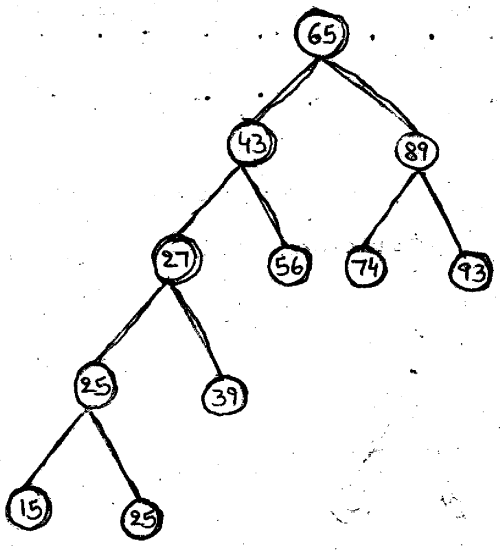
A binary Search tree is also called ordered binary tree. The data values of all the nodes in the left sub tree are less than that of the root node.

The data values of all the nodes in the right sub tree are greater than or

equals to the root node, the left and right subtrees are themselves binary search tree.

Eg: Construct BST using the following elements

65, 43, 89, 27, 25, 39, 93, 56, 25, 74, 15.



* Trees and their applications:

Non Recursive or Iterative traversal of a tree using stack operation:

There are 3 different trees to traversal. They are

1) Preorder Traversal
2) Inorder Traversal
3) Postorder Traversal

1) Preorder Traversal:

It follows the principle of DLR (Data, left, Right). To traversal in this order the following steps are to be adopted.

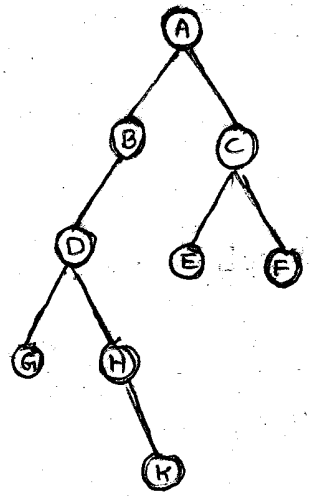
Step - 1: Set a pointer to the root node and process it, and repeat the following steps.

Step-2 : If the node has a right child push it into the stack.

Step-3 : After processing the node proceed down towards through its left child and follow the step-2. Untill a node with no left child is processed.

Step-4 : When a node with no left child is processed pop the top element from the stack and set a particular pointer to it and go back to step-2.

Eg:



Step	Process	Stack
1	A	C
2	B	C
3	D	C,H
4	G	C,H
5	H	C,K
6	K	C
7	C	F
8	E	F
9	F	-

Preorder traversal for above tree is

"ABDGHKCEF"

2) Inorder traversal:

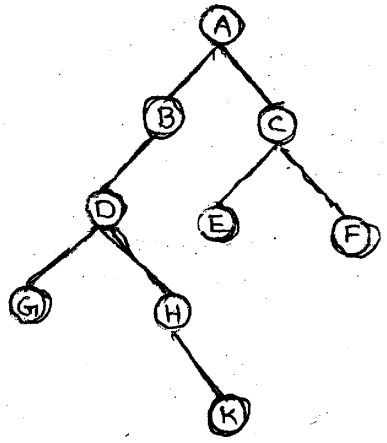
It follows the principle of LDR (Left, Data, Right). To traverse in this order, the following steps are to be adopted.

Step-1: Set a pointer to root node, proceed down towards the left by pushing each node onto the stack untill a node with no left child is pushed.

Step-2: When a node with no left child is pushed then pop the top element from the stack and process it.

Step-3: If the node process has right child then set a pointer to that right child and go back to step-1.

Eg:



Step	Stack	Process
1	A	–
2	A,B	–
3	A,B,D	–
4	A,B,D,G	–
5	A,B,D	G
6	A,B	D
7	A,B,H	–

8	A,B	H
9	A,B,K	-
10	A,B	K
11	A	B
12	-	A
13	C	-
14	C,E	-
15	C	E
16	-	C
17	F	-
18	-	F

Inorder traversal for above tree is

"GDHKBAECF."

3) Post order traversal:

It follows the principle of LRD (Left, right, Data). To traverse in this order the following steps are to be adopted.

Set a pointer to the root, proceed down towards the left and then perform the following steps.

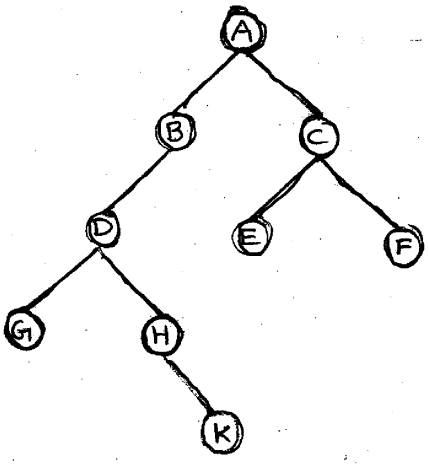
Step-1 : Push the node onto the stack.

Step-2 : If the node is having a right child push it's negative onto the stack.

Step-3 : Repeat the above steps until a node with no left child is pushed.

Step-4 : Pop and process the positive node. If a negative node is popped set a pointer to it and go back to Step-2.

Eg:



Step	Stack	Process
1	A	–
2	A,-C	–
3	A,-C,B	–
4	A,-C,B,D	–
5	A,-C,B,D,-H	–
6	A,-C,B,D,-H,G	–
7	A,-C,B,D,-H	G
8	A,-C,B,D,H,-K	–
9	A,-C,B,D,H,K	–
10	A,-C,B,D,H	K
11	A,-C,B,D	H
12	A,-C,B	D
13	A,-C	B
14	A,C	–
15	A,C,-F	–
16	A,C,-F,E	–
17	A,C,-F	E
18	A,C,F	–
19	A,C	F
20	A	C
21	–	A

Post order traversal for above tree is

"GKHDBEFCA."

Threaded binary tree:

In the binary tree the most of pointer fields right and left contains null elements. Hence some places in memory will be wasted unnecessarily so to avoid the wastage of space, the space will be used by some other type of information.

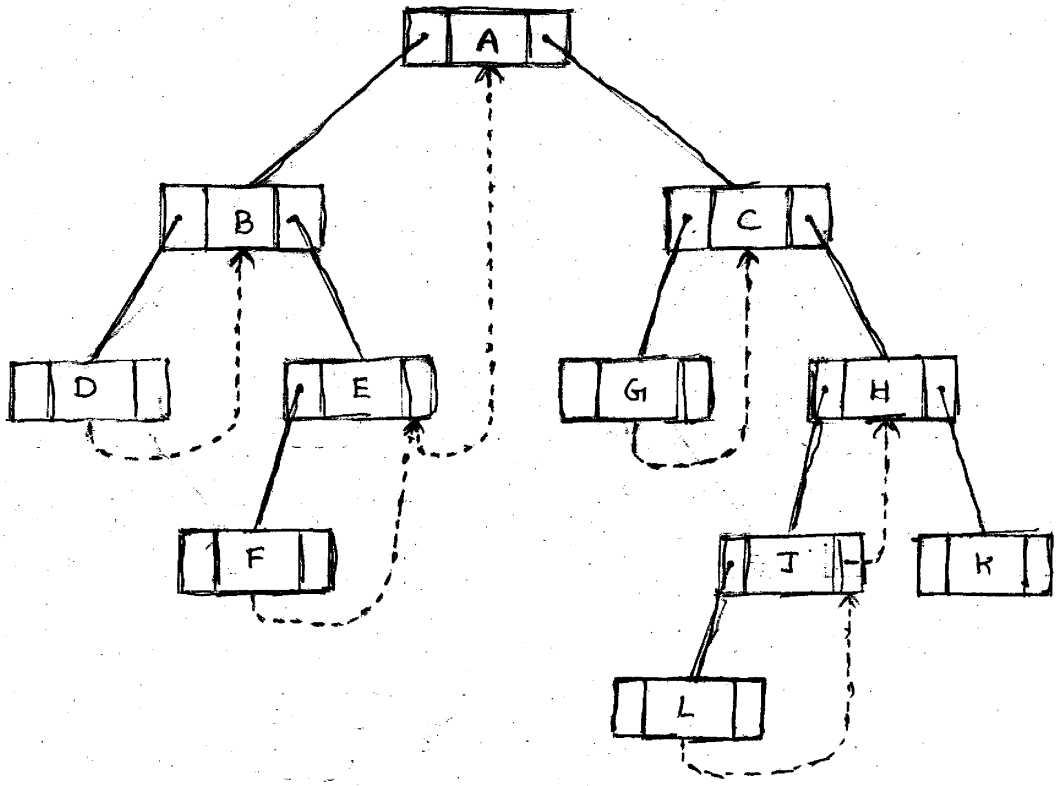
We will replace the null entries by special pointers which points to node higher in the tree. These special pointers are called threading and binary trees with such a pointers are called threaded trees (or) threaded binary trees.

One extra bit block field may be used in memory to construct a thread in memory, the thread may be denoted by negative integers and the ordinary pointers by positive integers.

The threaded binary tree can be divided into two types.

1) one way threading.
2) Two way threading.

In one way threading, the thread will appear at the right side of the node and it will point to the next node in inorder traversal of tree.

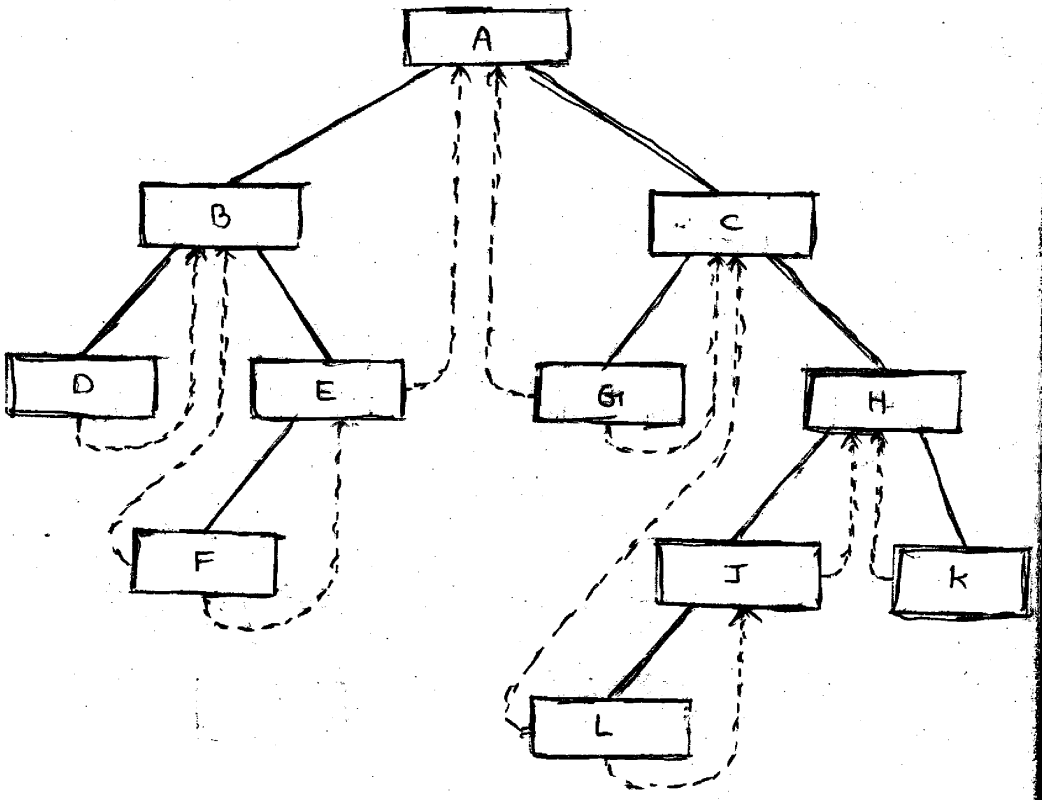


One-way Threading

D → B → F → E → A → G → C → L → J → H → K.

In two way threading, the thread will appear at the left side and right side of the node. The left side thread points to the previous (or) preceeding node and the right side points to the next or successor node in the inorder traversal of tree.

The left pointer of the first node and the right pointer of the last node will contains the null value when tree does not have the header node. If it has a header node then they points to the header node.



Two - way Threading.

$\otimes \leftarrow D \rightleftarrows B \rightleftarrows F \rightleftarrows E \rightleftarrows A \rightleftarrows G \rightleftarrows C \rightleftarrows L \rightleftarrows J \rightleftarrows H \rightleftarrows K \rightarrow \otimes$

Every binary threaded tree is a binary search tree. But every binary such tree such not be a binary threaded tree.

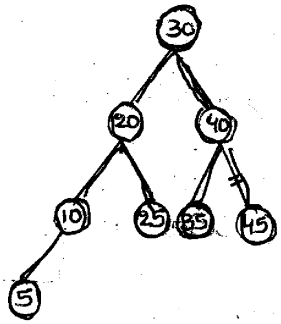
* Deletion of a node from binary search tree:

The deletion of a node from binary search tree follows into three categories.

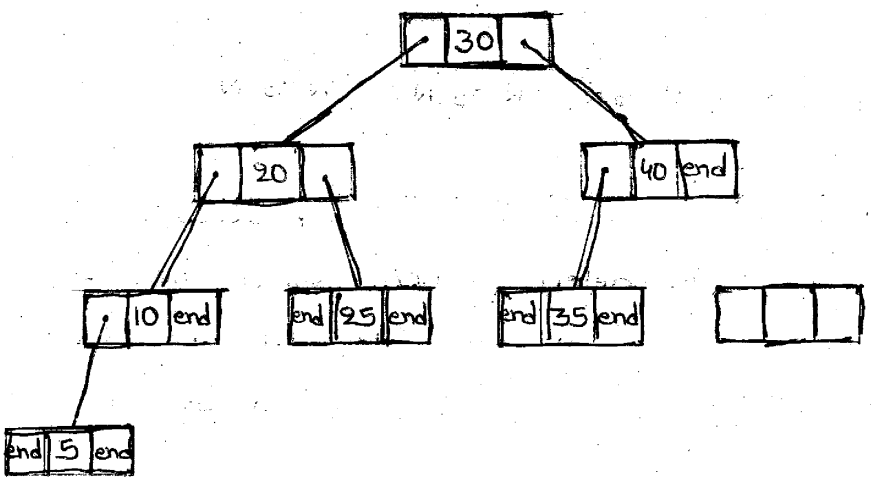
1) Deleting a node with no children.
2) Deleting a node with one children
3) Deleting a node with two children.

1) Deleting a node with no children: To delete a node which has no children to the Parent pointer of that node.

Before Deletion



After Deletion

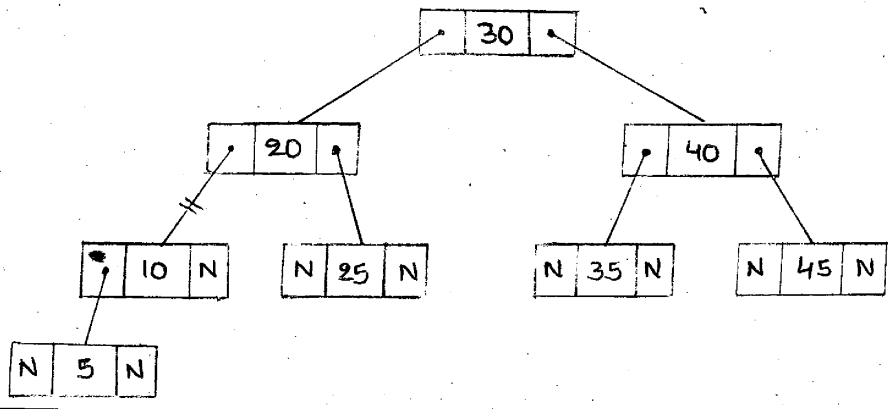


2) Deleting a node with one children:

To delete a node with one children the process is:

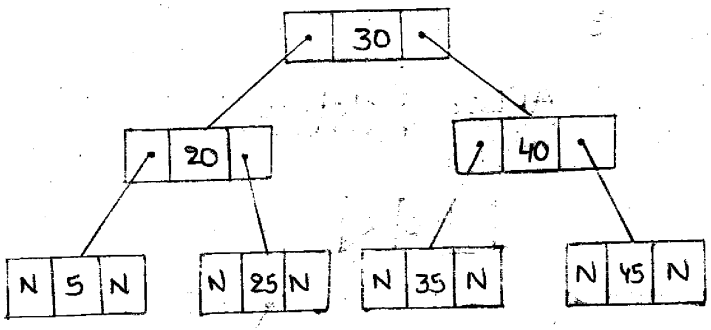
1) To identify parent pointer of that node.
2) Then pointer assign the address of the child of the deleted node to it's parent.

Before Deletion



For example, from the above diagram we want to delete node containing value 10 which have a single child i.e, node 5.

After Deletion



3) Deletion of a node with two children:

To delete a node which has two children we have to adopt the following steps:

1) First identify the parent node of the deleted node.

2) Put a pointer variable (Par) at parent and put another pointer variable (Temp) at the deleted node right child.

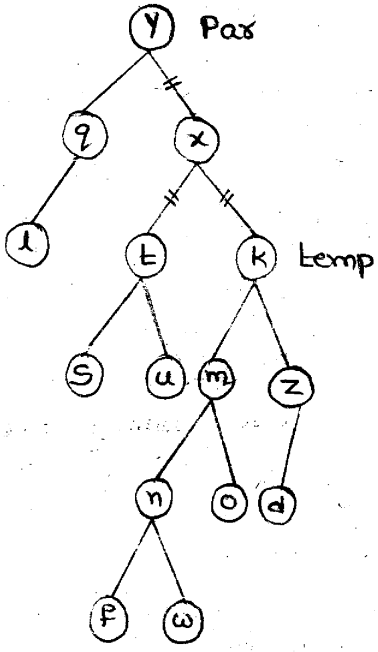
3) Traverse the temp pointer variable upto "temp → left != NULL."

4) Add the deleted node left child at the temp left i.e; "temp → left ← Parent → left".

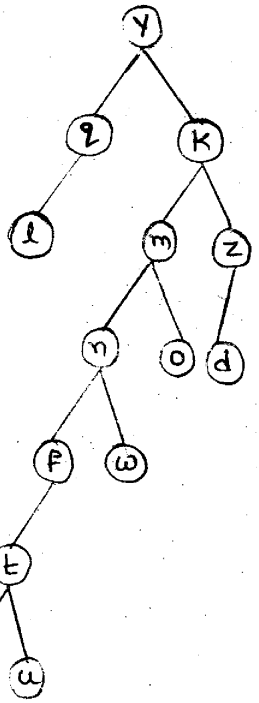
5) Finally add the deleted node right child to the parent of the deleted node.

6) Now we can remove the deleted item from a binary search tree.

Before Deletion



After Deletion



* Huffman Algorithm:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable length codes to input characters, lengths of the

assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are prefix codes, means the codes (bit sequences are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.

There are mainly two major parts in Huffman coding.

1) Build a Huffman tree from input characters.
2) Traverse the Huffman tree and assign codes to characters.

Steps to build Huffman tree:

Input is array of unique characters only with their frequency of occurrences and output is Huffman tree.

1) Tree at a leaf node for each unique character and build a min heap of all leaf nodes (Min heap is used as priority queue. The value of frequency field is used to compare to nodes in min heap. Initially, the least frequent character is at root).
2) Extract two nodes with the minimum frequency from the min heap.
3) Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as it's right child. Add this node to the min heap.

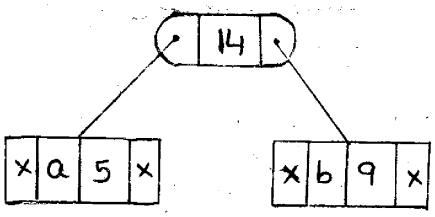
4) Repeat Steps - 2 and 3 untill the heap contains only one node. The remaining node is the root node and the tree is complete.

Eg:

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step-1: Build a min heap that contains six nodes where each node represents root of a tree with single node.

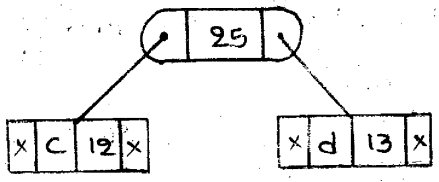
Step-2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5+9=14. It is denoted by



Now min heap contains 5 nodes where 4 nodes are roots of the trees with single element each, and one heap node is root of tree with 3 elements.

Character	Frequency
c	12
d	13
Internal node	14
e	16
f	45

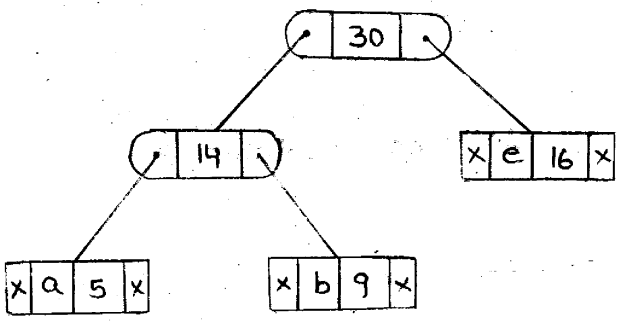
Step-3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12+13=25. It is denoted by



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one node.

Character	Frequency
Internal node	14
e	16
Internal node	25
f	45

Step-4: Extract two minimum frequency nodes. Add a new internal node with frequency 14+16=30

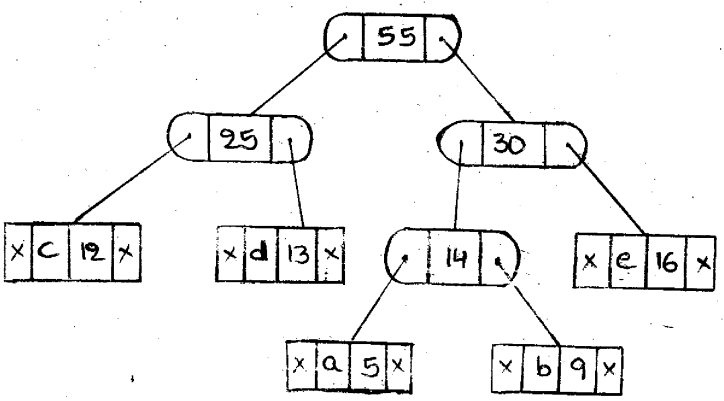


Now min heap contains 3 nodes.

Character	Frequency
Internal node	25
Internal node	30
f	45

Step-5: Extract two minimum frequency nodes.

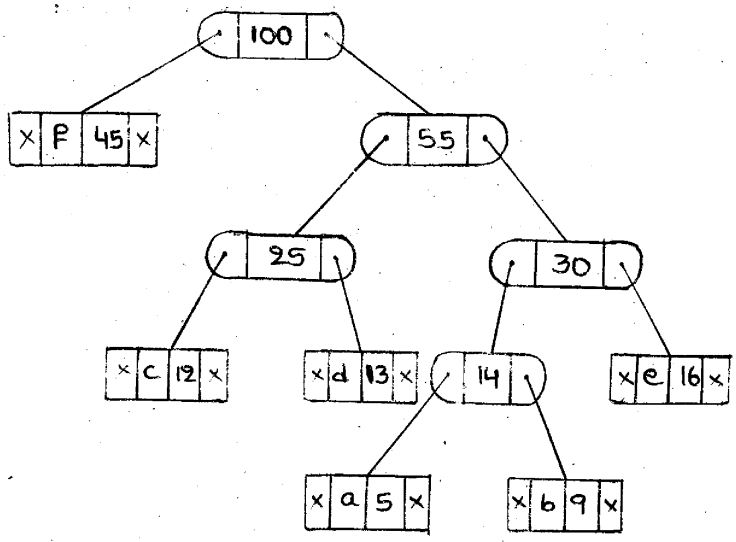
Add a new internal node frequency 25+30=55



Now min heap contains 2 nodes.

Character	Frequency
F	45
Internal node	55

Step-6: Extract two minimum frequency nodes. Add a new internal node with frequency 45+55=100.



Now min heap contains only one node.

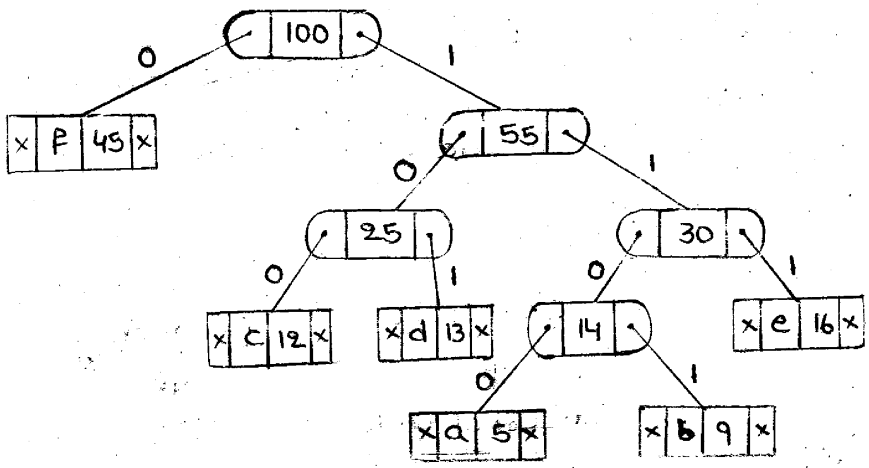
Character	Frequency
Internal node	100

Since the heap contains only one node, the algorithm stops.

Steps to print codes from Huffman tree:

1) Traverse the tree formed starting from the root.

(20) 2) Maintain an auxiliary array. While moving to the left child, write '0' to the array. While moving to the right child, write '1' to the array. Print the array when a leaf node is encountered.



The code are as follows:

Character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Graphs

Graph:

Graph is a set of vertices and edges i.e., G1=(V,E). We can represent a group using an array of vertices and 2-dimensional array of edges.

Vertex:

Each node of the graph is represented as a vertex. We can represent them using an array identified by index.

Edge:

Edge represents a path between two vertices or a line between two vertices. We can use a 2-dimensional array to represent edges.

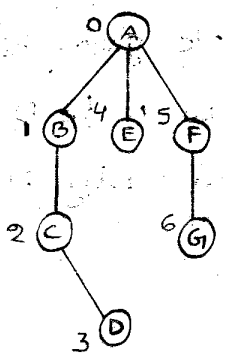
Adjacency:

Two nodes or vertices are adjacent if they are connected to each other through an edge.

Path:

Path represents a sequence of edges between the two vertices.

Eg:



Basic operations:

Following are the basic primary operations of a graph.

1) Add vertex:

Adds a vertex to the graph.

2) Add edge:

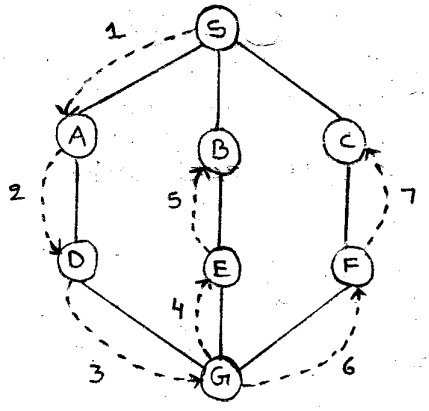
Adds an edge between the two vertices of the graph.

3) Display vertex:

Displays a vertex of the graph.

* 1) Depth First Search (DFS):

DFS algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule-1:

Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

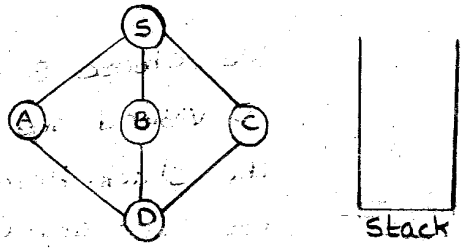
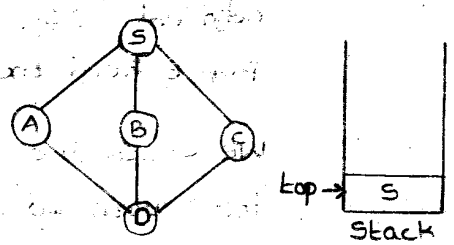
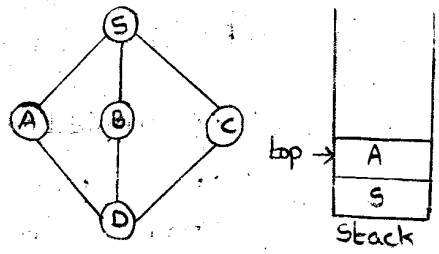
Rule-2:

If no adjacent vertex is found, pop up

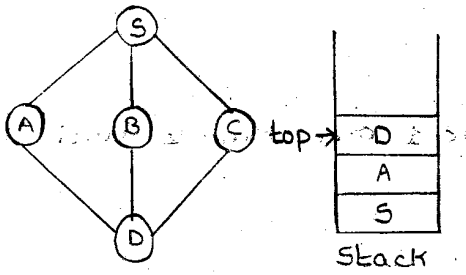
a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule-3:

Repeat Rule 1 and Rule 2 until the stack is empty.

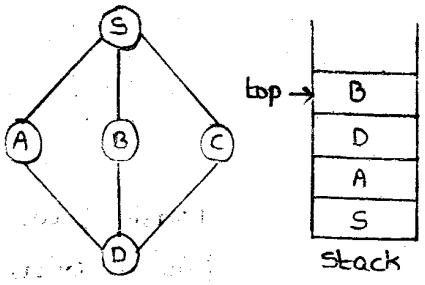
Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4.



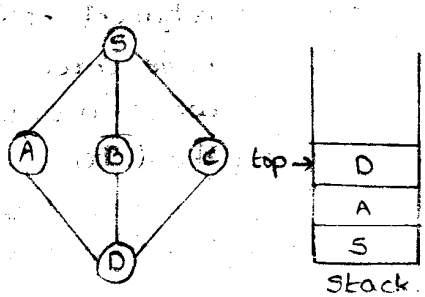
Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.

5.



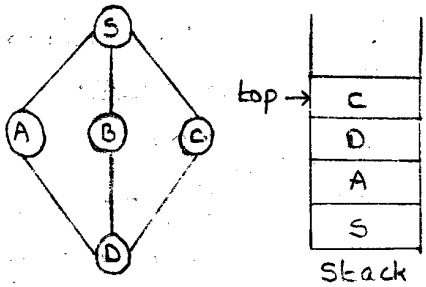
We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.

6.



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.

7.



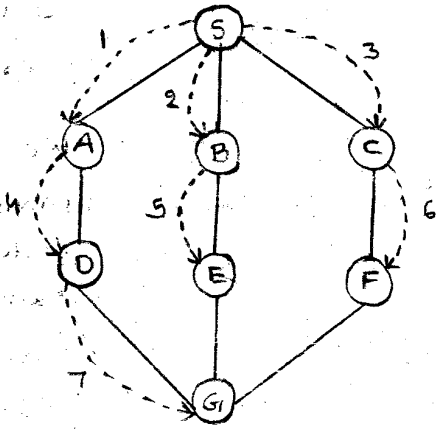
only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited

adjacent node. In this case, there's none and we keep popping until the stack is empty.

2) Breadth First Search (BFS):

BFS algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search. When a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule-1:

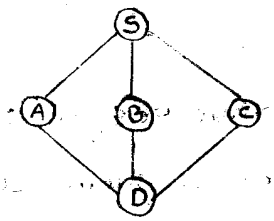
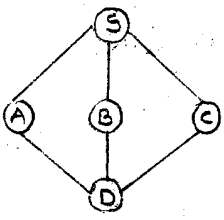
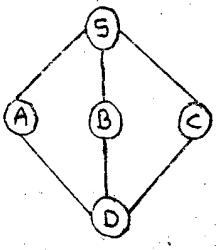
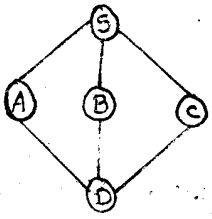
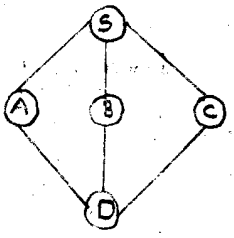
Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

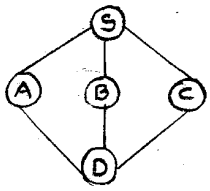
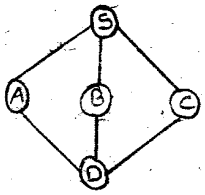
Rule-2:

If no adjacent vertex is found, remove the first vertex from the queue.

Rule-3:

Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1.	 Queue	Initialize the queue.
2.	 S | Queue	We start from visiting S (starting node), and mark it as visited.
3.	 S A | Queue	We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4.	 S B A Queue	Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5.	 S C B A Queue	Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.

6.		S C B Queue	Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7.		S P C B. Queue	From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

As this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the Program is over.

Sorting And Searching

Practical general sorting algorithms are almost always based on an algorithm with average time complexity O(nlogn), of which the most common are heap sort, merge sort and quick sort and so on. Finally, they may also be unstable, and stability is often a desirable property in a sort.

* Bubble sort:

Basically the bubble sort is slow, but its conceptually the simplest of the sorting algorithms.

Let we assume that there are N numbers and the positions are numbered from '0' on the left to N-1 on the right. Now we have to start at the left end of the line and compare two numbers in positions '0' and '1'. If the one of the left (0^{th} position) is bigger, we have to swap them. If the one on the right is taller we don't do any thing then we have to move over one position and compare the numbers in position 1 and 2, again if the one on the left is bigger, we have to swap them.

Here are the rules we are following.

1) Compare two numbers.

2) If one on the left is bigger, swap them.

3) Move one position right.

Eg:	0	1	2	3	4	5	6	7	8	9
	77	99 (swap)	44	55	22	88	11	00	66	33
	77	44	99	55 (swap)	22	88	11	00	66	33
	77	44	55	99	22 (swap)	88	11	00	66	33
	77	44	55	22	99	88	11	00	66	33
	⋮									
1st pass:	77	44	55	22	88	11	00	66	33	99
2nd pass:	44	55	22	77	11	00	66	33	88	99
3rd pass:	44	22	55	11	00	66	33	77	88	99
4th pass:	22	44	11	00	55	33	66	77	88	99
5th pass:	22	11	00	44	33	55	66	77	88	99
6th pass:	11	00	22	33	44	55	66	77	88	99
7th pass:	00	11	22	33	44	55	66	77	88	99

→ Sorted order.

If we observe clearly the above example after first pass, we have made n-1 comparisions and some where between '0' and (n-1) swaps. The number at end of the array was sorted and can't be moved again.

Now as shown in 2nd pass we have to come back and start another pass from the left end of the line. Again, you go towards the right, comparing and swap. When appropriate however this time you can stop one number

short of the end of the line at position n-2, because we know the last position at n-1, already contains the biggest number.

Bubble sort defination:

* Bubble sort the side by side elements are compared that means j^{th} position element is compared with $(j+1)^{th}$ position element.
* If j^{th} position element is greater than $(j+1)^{th}$ position element. Then perform swapping otherwise no need of swapping. This process is continued untill last but one element.

* Selection sort:

The selection sort improves on the bubble sort by reducing the number of swaps necessary from $O(N^2)$ to $O(N)$. But, the number of comparisions remains $O(N^2)$.

The concept involved in the selection sort is making a pass through all the elements and picking the shortest one. This shortest elements is then swapped with the elements on the left end of the line at position '0'.

Now the left most players is sorted and won't need to be moved again. In this algorithm the sorted elements accumilated on the right.

In this next pass, we have to sort at position 1 and finding the minimum, swap

with position 1. This process will continue untill all the elements are sorted.

Eg:

77	99	44	55	22	88	11	00	66	33

↑ ——— ↑ Swap

00	99	44	55	22	88	11	77	66	33
00	11	44	55	22	88	99	77	66	33
00	11	22	55	44	88	99	77	66	33
00	11	22	33	44	88	99	77	66	55
00	11	22	33	44	55	99	77	66	88
00	11	22	33	44	55	66	77	99	88
00	11	22	33	44	55	66	77	88	99

——————→ Sorted

* Heap sort :

Heap sort is a comparision based sorting techinque based on a "binary Heap" data structure.

Binary Heap:

A binary heap is a complete binary tree where value in a parent node is greater (or smaller) than the values in it's two children nodes. The former is called as "max heap" and the latter is called "min heap". The heap can be represented by binary tree or array.

Since a binary heap is a complete binary tree, it can be easily represented as array and array based representation is ~~space~~ sufficient. If the parent node is stored at

index I, the left child can be calculated by 2*I+1 and right child by 2*I+2 (assuming the indexing starts at 0).

Heap sort algorithm for sorting in increasing order:

Step-1: Build a max heap from the input data.

Step-2: At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

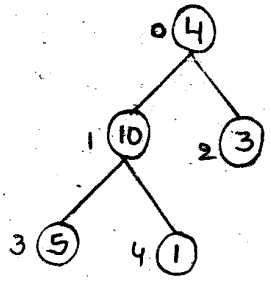
Step-3: Repeat above steps while size of heap is greater than 1.

Eg: Input elements: 4, 10, 3, 5, 1.

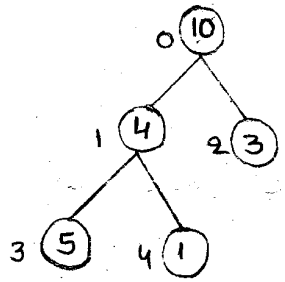
Array:

0	1	2	3	4
4	10	3	5	1

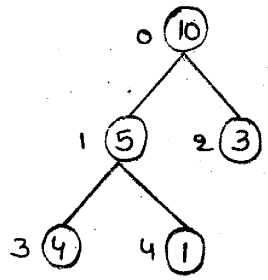
Build heap



Step-1: Build max heap.

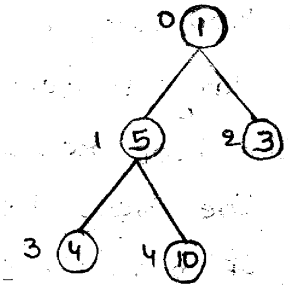


0	1	2	3	4
10	4	3	5	1



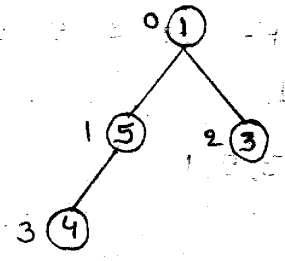
0	1	2	3	4
10	5	3	4	1

Swap last element with heap element.



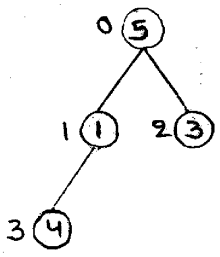
0	1	2	3	4
1	5	3	4	10

Remove the last element.

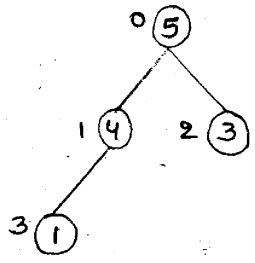


0	1	2	3	4
1	5	3	4	10

Step-2 : Build max heap.

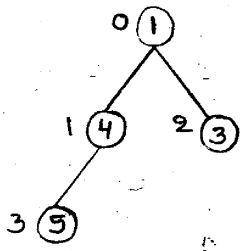


0	1	2	3	4
5	1	3	4	10



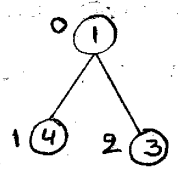
0	1	2	3	4
5	4	3	1	10

Swapping last element with heap element.



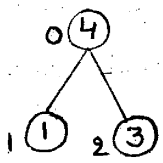
0	1	2	3	4
1	4	3	5	10

Remove the last element.



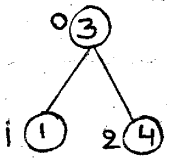
0	1	2	3	4
1	4	3	5	10

Step-3: Build max heap



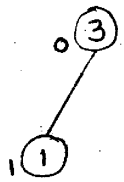
0	1	2	3	4
4	1	3	5	10

Swap last element with heap element.



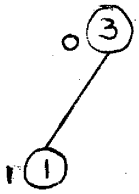
0	1	2	3	4
3	1	4	5	10

Remove last element.



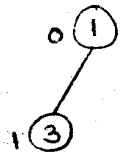
0	1	2	3	4
3	1	4	5	10

Step-4: Build max heap.



0	1	2	3	4
3	1	4	5	10

Swap the last element with heap element.



0	1	2	3	4
1	3	4	5	10

Remove the last element.

0	1	2	3	4
1	3	4	5	10

As only one element exists the algorithm ends. The heapify procedure calls itself recur-sively to build heap.

* Insertion sort:

Insertion sort also executes $O(N^2)$ times. But its about twice as fast as the bubble sort and some what faster than the selection sort.

To begin the insertion sort, start with our numbers linked up in random order. It is easier to think about the insertion sort if we begin in the middle of the process, when the numbers are half sorted.

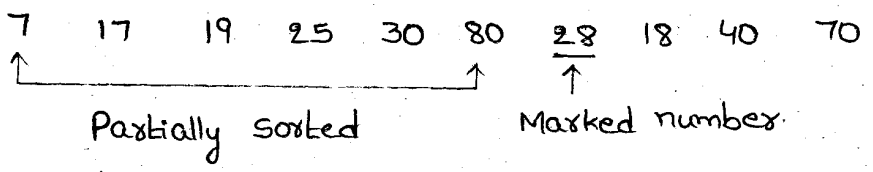
Partial sorting:

At this point there is an imaginary marker somewhere in the middle of the line. The numbers to the left of this marker are partially sorted. That means each number is bigger to its right.

Marked number:

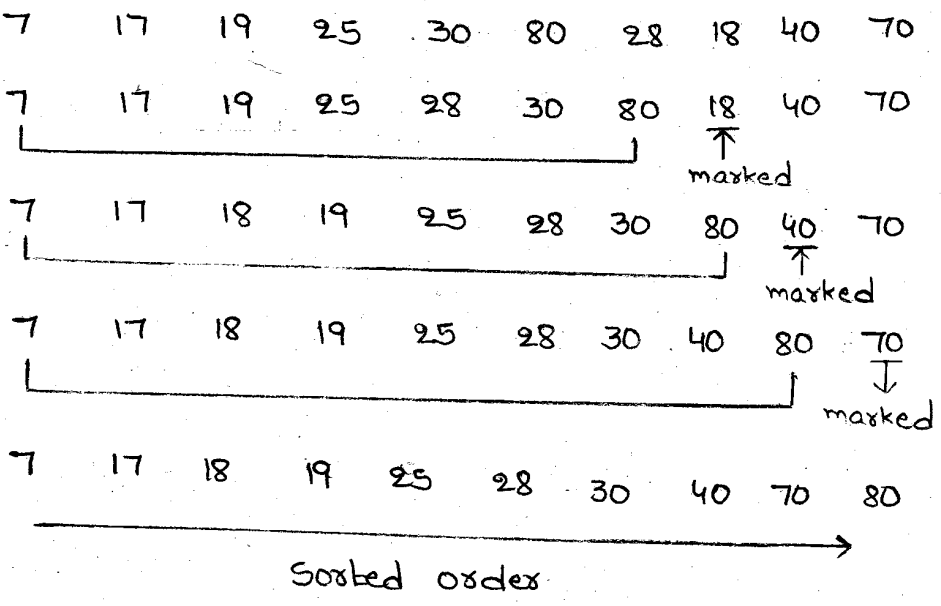
The number where the marker is, we will call as marked number. All the numbers to the right of the marked numbers are

unsorted. This is shown in following.



Now we are going to do is insert the marked numbers in the appropriate place in the sorted group. For this we need to shift some of the sorted numbers to the right i.e, the biggest number moves to the marked numbers spot and so on. This process is repeated untill all the unsorted numbers have been inserted.

Eg:

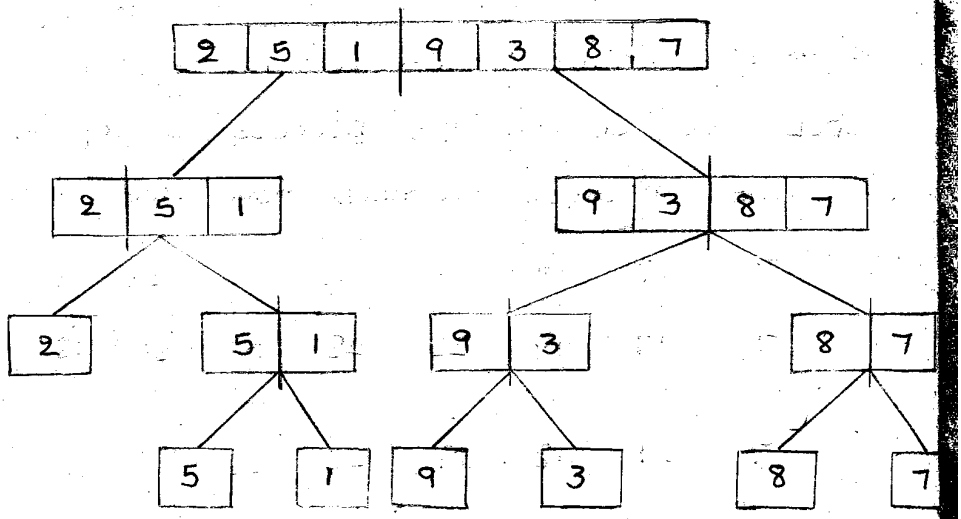


* Merge Sort:

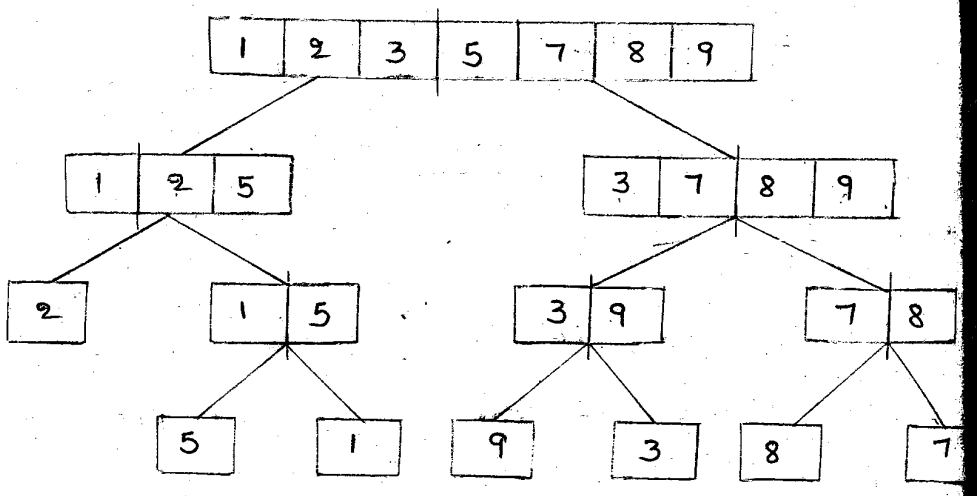
The idea behind the merge sort is very simple. If you are given a pile of cards to sort divide the pile into half, sort each half independently. We would also apply the merge sort algorithm again to each half pile. Apply the merge sort recursively to each half pile

until we encounter the smallest possible pile, a pile of one object. We then merge the sub piles together one stage at a time until we reach the top level again at which point the entire list is sorted.

Consider the below figure, initially the numbers to be sorted are 2 5 1 9 3 8 7.



Sub division of the list in merge sort:



Merge the sublist.

Algorithm for merge sort:

Step-1 : Merge sort (Array (first ---- last))

Step-2 : Begin

Step-3 : If array contains any one element then Return array.

Step-4 : else

Step-5 : Middle = ((Last + first)/2) rounded down to the nearest integer.

Step-6 : Left half array = merge sort (Array (first ---- middle)).

Step-7 : Right half array = merge sort (Array (middle+1 ---- last)).

Step-8 : Result array = merge (Left half array, Right half array).

Step-9 : Return result array

Step-10 : End if

Step-11 : End mergesort.

* Searching algorithm:

Searching:

Searching is nothing but finding an element weather it is existing or not in the list.

Linear search:

* It is also called as sequential search.
* In linear search the finding element or key element is compared with from 0th position to ending position.
* In the finding element is equal or matching to any one of the element then the search

was found or completed and we can stop the process.

* If the finding element is not equal or not matching to any one of the element then we can decide the element may not found or the search was not found.

Algorithm:

'a' is an array with 'n' elements and fine is the searching element.

Step-1 : Start

Step-2 : i ← 0

Step-3 : Repeat steps from 3 to 5 untill i<n.

Step-4 : if a[i] = key then

Step-4.1 : Pos ← i+1, break;

Step-5 : i ← i+1

Step-5.1 : if pos>0 then

Step-6 : Print elements found.

Step-7 : else

Step-7.1 : Print element not found

Step-8 : Stop.

* Binary Search:

* For performing binary search the elements must be acsending order.

* We set "Low" variable at 0th position and "high" variable at ending position. After finding the mid position by using following formula.

$$Mid = \frac{(low+high)}{2}$$

* If the key element is equal to mid position element then we decide the element was found and stop the process otherwise go the next step.
* Suppose if the key element is less than the mid position element then high position is changed to mid-1 like below.

high = mid - 1.

Otherwise goto next step.

* Suppose if the key element is greater than the mid position element the low position is changed to mid+1 like below.

low = mid + 1.

The above processes are repeated untill the element is found. Suppose if the process was completed but we did not find the element then we can decide the element was not found.

Algorithm:

b search (a, h, find)

Step - 1 : Start

Step - 2 : set low ← 0, high ← n-1

Step - 3 : Repeat steps from 3 to 7 untill low ≤ high.

Step - 4 : mid ← (low+high)/2.

Step - 5 : if key = a[mid] then

Step - 5.1 : pos ← i+1, break;

Step-6 : else if key < a[mid] then

Step-6.1 : high ← mid-1

Step-7 : else

Step-7.1 ; low ← mid+1

Step-8 : if key = a[mid] then

Step-8.1 : Print element was found.

Step-9 : else

Step-9.1 : Print element was not found.

Step-10 : Stop.

* Quick Sort:

Quick Sort is the most popular algorithm. It is the fastest algorithm, operating in the O(N* log N). There are three basic steps in this algorithm.

1) Initialize i and j. at first and last element position initialize pivot variable with the list first position.

2) Take the ith position element compare with pivot position element. If the ith position element is '<' or '=' to pivot position element, then increment continue this process until the ith position element is greater than the pivot element.

3) Take the j^{th} position element compare with pivot position element. If j^{th} position element is '>' the pivot position element then decrement j. This process is continue until the j^{th} position

element is '<' pivot position.

If there is no possibility to move i and j then exchange i and j elements.

If i and j are crosses (or) i=j then there is no need to go forward of i and j. Then exchange pivot position element with jth position element. Then the list is splitted into 2 parts. The pivot element of the before part is one list and the pivot position after element is another list.

Eg: 42 89 63 12 94 27 78 3 50 36

Again the above process can be applied repeatedly to each individual list until the new list cannot be formed.

Eg:

(42) Pivot	89 i	63	12	94	27	78	3	50	36 j
(42) Pivot	36	63 i	12	94	27	78	3 j	50	89
(42) Pivot	36	3	12	94 i	27 j	78	63	50	89
(42) Pivot	36	3	12	27 j	94 i	78	63	50	89
(27) Pivot	36 i	3	j 12	(42)	(94) i Pivot	78	63	50	j 89
(27) i Pivot	12	3 j	36	42	(89) i Pivot	78	63	j 50	94
3	12	27	36	42	(50) i Pivot	78	j 63	89	94
					i (78) Pivot	63 j			
					63	78			

Hence the final sorting order is

3 12 27 36 42 50 63 78 89 94

Algorithm: Qsort (a,0,n-1)

Here 'a' is an array and '0' is lower boundary and "n-1" is upper boundary.

Step-1: Start

Step-2: if m<n then

Step-2.1: Set i ← 0, pivot ← 0, j ← n;

Step-3: Repeat steps from 3 to 6 untill i<j.

Step-4: Repeat steps from 4 to 4.1 untill a[i] ≤ a[pivot].

Step-4.1: i ← i+1

Step-5: Repeat steps from 5 to 5.1 untill a[i] > a[pivot].

Step-5.1: j ← j-1;

Step-6: if i<j then.

Step-6.1: Swap (a[i], a[j]).

Step-7: Swap (a[j], a[pivot]).

Step-8: Call Qsort (a,m,j-1).

Call Qsort (a, j+1, n)

Step-9: Stop.

* Efficiency of all sorting:

Method	Worst	Average	Best case
Bubble sort →	n^2 →	n^2 →	n^2
Insertion sort →	n^2 →	n^2 →	n^2
Selection sort →	n^2 →	n^2 →	n^2
Heap sort →	$n \log n$ →	$n \log n$ →	$n \log n$
Merge sort →	$n \log n$ →	$n \log n$ →	$n \log n$
quick sort →	n^2 →	$n \log n$ →	$n \log n$