

Introduction

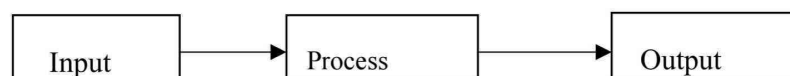
History of Computer:

- This word is taken from the **Latin** language. The computer word can be slipped into 2 parts. One is **comp** and another is **puter**.
- **Comp** means **automatic**, **puter** means **calculation**. Computer is nothing but an automatic calculation (or) electronic device.
- **Abbreviation of computer:**
 - C – Common
 - O – Operating or oriental
 - M - Machine
 - P - Particularly
 - U - Used for
 - T - Technical or trade
 - E - Education
 - R - Research

Computer Definition: -

Computer is an electronic device, which is used to process and to store the large amount of data. A computer can also perform arithmetic and logical operations at a high speed according to user instructions; it does not process any intelligence of its own. That means, it does not have any thinking (or) giving (or) decision taking of its own. That means, it does not have any thinking (or) giving (or) decision taking of its own.

A computer system accepts data and instructions as input and process that data according to the given user instructions to provide results as output.

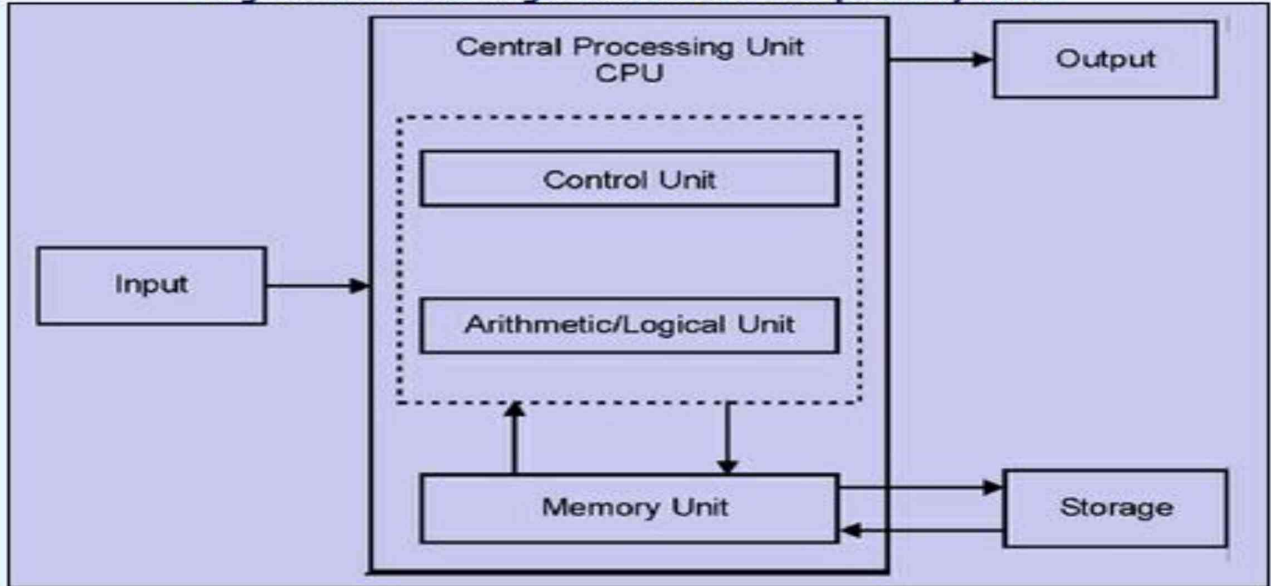


Basic Organization of Computer:-

The computer mainly consists of the following components. They are:

- Input device
- Output device
- Central Processing Unit (CPU)

Figure 1.1: Basic Organization of a Computer System



Input device:-

- The input devices allow the user to input (or) store data and instructions to the computer.
- There are varieties of input devices. They are **Keyboard, Mouse, Joy Stick, Track ball, Digitalcameras** and so on.

Central Processing Unit:-

- CPU stands for central processing unit. It is the most important component of a computer.
- It is the brain of the computer and all processing take place in the CPU.
- It typically consists of three parts. They are:
 - *ALU (Arithmetic & logical unit)*
 - *Control Unit*
 - *Memory Unit*

ALU(Arithmetic Logical unit): -

- ALU stands for Arithmetic & logical unit. It performs all arithmetic operations like Add,Sub, Mul, div etc., and logical operations like >, <, <=, >=,.
- Whereas the logical operations are used to compare two numbers and generate either true or false values.

Control Unit: -

- It controls all the operations of the CPU and peripheral devices.
- It controls the flow of data between the CPU and Input-Output devices.
- The CU decides which instructions will be executed & operations performed. It takes care of the step – by – step processing of all operations are performed in the computers

Memory Unit

- The storage area of the computer is called as memory. The memory can be divided in totwo types.
- ✓ Primary memory
- ✓ Secondary memory

Primary Memory:

- This memory is used for temporary storage. The contents of memory will be availableas long as the power is in ‘on’ state.
- This memory allows the CPU to store and retrieve data quickly.

ROM:-

- ROM stands for Read Only Memory.
- It is used to start up the computer system.
- Primary storage space is very expensive & limited capacity and also known as“Volatile memory”.
- The contents stored in the ROM are used for reading purpose only.

RAM:-

- RAM stands for **Random Access memory**.
- It is used to do some intermediate manipulations.
- RAM is a temporary memory.

Secondary (Storage) Memory:-

- This memory allows storing the data permanently.
- Secondary storage is also known as “Secondary memory” or “Auxiliary memory”. Itbasically overcomes all the drawbacks of primary

storage area.

- The contents will be available permanently either power is 'on' or 'off' state.
- The mostly used secondary devices are **floppy disks, Hard disk, CD's, DVD's, magneticTape** etc.,

Output Devices:-

- The output device is used to present the data (or) information from the memory.
- Since the computer all data & results in binary form that's why the result cannot be directly given to the user.
- This device converts the binary information to human understandable language.**Ex: Monitor, Printers, Speakers** etc.

Variables:

- A Variable is an entity (or) identifier whose value can be changed during program execution and is known to the program by a name.
- A Variable definition associates a memory (or) storage location to the Variable name. It can hold only one value at a time during the program execution.
- Variable names are identifiers used to name Variables.
- They are symbolic names assigned to the memory locations.

RULES IN VARIABLES:

- A Variable name consists of letters, digits and underscore.
- Variable name must begin with an alphabet or under _score character.
- The maximum number of characters used in forming a Variable must not exceed 31 characters.
- Variable name does not allow any special characters even blank spaces also.
- Any keywords can not use as variable names.
- C is case sensitive for instance variable names such as rate, Rate and RATE are treated as different.

Valid Variable name:-

Ex:- sum, sno, fact

Invalid Variable names:-

Eg:- 9_empno, data_of_join, float, \$ symbols

Variable declaration syntax:-

Data type Var_name1, Var_name2,.....,Var_nameN;

Variable Initialization

Syntax:- Data type Var_name=constant value;

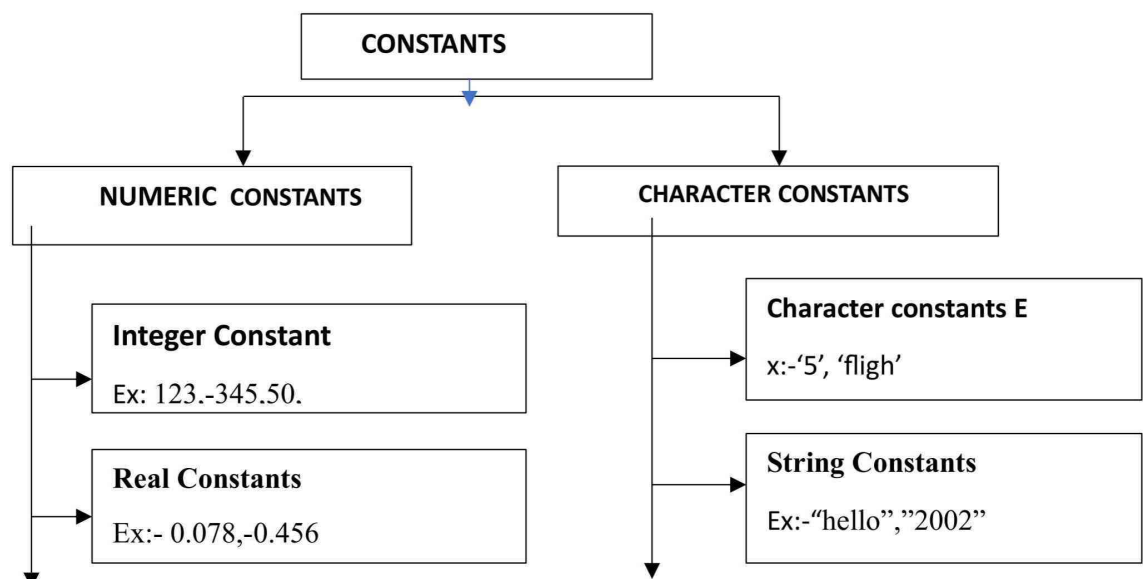
Eg:- int sno=33;

float fees=6500;

Constants:-

- Constants are Literals in C are a sequence of characters (like digits, letters and other characters) that represent constant values to be stored in variables.
- Constants in C refer to fixed values that do not change during the execution of a program.

C supports several types of constants. They are..



- Literals are constants to which symbolic names are associated for the purpose of readability and easy of handling. C provides the following three ways of defining Constants:

1. # (Hash) define preprocessor directive
2. Enumerated data types
3. Const Keyword

- The Variables in C can be created and initialized with a constant value at the point of its definition.

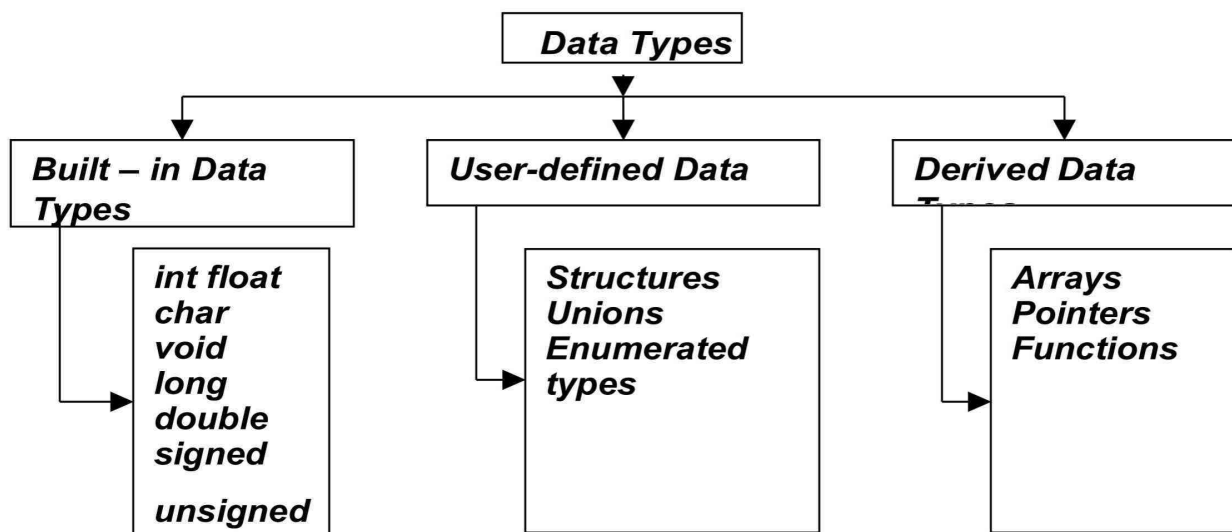
Eg: - **float Pi=3.142;**

- Defines a Variable name Pi and is assigned with floating point numeric constant value 3.142. It is known as that the constant value that not changes.
- In the above case the Variable Pi is considered as constant whose value does not change through out the life at the program. The qualifier used in C to define such Variable is the Const qualifier.
- The syntax of defining Variables with the Const Qualifier is

Syntax:- Const datatype variable=value

Data types:

- Every variable in C has a data type.
- Data types specify the size and type of values that can be stored.
- The variety of data types available allows the programmer to select required type to the needs of the application.
- C language can have 3 types of data types like....
 - **Built- in data types**
 - **Derived data types**
 - **User – defined data types**



Built-in -data Types

Operators and Expressions

- C Operators are special characters (or) symbols which instruct the compiler to perform Operations on some Operands. Operation instructions are specified by Operators, which Operands can be Variables, expressions are literal values.
- Some operators Operate on a single Operand they are indicated before Operands and they are called prefix Operators. Others indicated after the Operand is called post-fix Operators.
- Most Operators are embedded between the two Operands and they are called binary '+' Operators. An expression a+b uses the binary '+' Operators. C has even Operator that takes three Operands called ternary Operator.
- Operators are used in programs to manipulate data and variables. C operators can be classified as several types like...
 - *Arithmetic Operators*
 - *Relational Operators*
 - *Logical Operators*
 - *Assignment Operators*
 - *Increment and decrement Operators*
 - *Conditional Operator*
 - *Bitwise Operators*
 - *Special Operators*

Arithmetic Operators: -

- The Arithmetic Operators are used to perform Arithmetic Operations on numeric values. They have both unary and binary categories.
- These operators are used to construct mathematical expressions as in algebra. C provides the basic arithmetic operators this can be operate on any built-in numeric data types and we cannot use these operators on Boolean types.

Relational Operators:-

Relational operators are also called as comparison operator. Compare between the two operands.

<u>Operator</u>	<u>Meaning</u>
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Ex: - $x > y$, $x < y$, $x >= y$, $x <= y$, $x == y$,

LOGICAL OPERATORS:-

- Logical operators combine two (or) more relational expressions is termed as a “logical expression (or) compound relational expression”.
- Any expression that evaluates to zero denotes a false logical condition and that evaluation to non-zero value denotes a true logical condition. Logical operators are useful in combining one or more conditions (relations).

<u>Operator</u>	<u>Meaning</u>
&&	Logical AND
	Logical OR
!	Logical NOT

Logical AND (&&):-

If the both conditions are True then the result is True. Otherwise then the result is False.

Syntax:- (operand_1 && operand_2)

<i>p</i>	<i>Q</i>	<i>P&&Q</i>
T	T	T
T	F	F
F	T	F
F	F	F

Logical OR (||): -

If the both conditions are True then the results is True and any one condition is True then the result is also True. Otherwise the results is False.

Syntax: - (operand_1 || operand_2)

<i>p</i>	<i>Q</i>	P Q
T	T	T
T	F	T
F	T	T
F	F	F

Logical NOT (!): -

Syntax: -! (Operand_1 && operand_2)

p	!p
T	F
F	T

Assignment Operator (=):-

- Assignment operators are used to assign the value of an expression to a variable we

have seen the usual assignment operator like “=”

- “=” is an assignment operator, which is used to assign right part of the expression (or) value in to left side of the variable. It has the following.
- Assignment operators are used to assign the value of an expression to a variable we have seen the usual assignment operator like “=”
- “=” is an assignment operator, which is used to assign right part of the expression (or) value in to left side of the variable. It has the following.

Syntax:-

Variable = expression (or) value;

- The left hand side has to be a variable and the right hand side has to be a valid expression.

Ex:- a=5;

C language has a set of short hand assignment operators and its form like...

Syntax:-

variable operator=expression;

Increment And Decrement Operators:-

INCREMENT OPERATOR:

This is used to increase the value one by one. It has two types:

* Post-fix Increment operator

* pre-fix Increment operator

POST-FIX INCREMENT OPERATOR:

“++” symbol is used to represent Post-fix Increment operator. This symbol is used after the operand.

In this operator, value is first assign to a variable and then incremented the value.

EX:- int a=10,b;

```
b=a++;
```

O/P: a=11

```
b =10
```

In the above example first the value of “a” is assign to the variable “b”, then increment the value, so the value of b variable is “10”.

PRE-FIX INCREMENT OPERATOR:

++” symbol is used to represent Pre-Fix operator, this symbol is used after the operand. In this operator value is incremented first and then assigned to a variable.

EX:- int a=10,b;

```
b=++a;
```

O/P: a=11;

```
b=11;
```

In the above example first the increment is done then the value of “a” variable is assigned to the variable “b”, so the value of “b” variable is “11”.

DECREMENT OPERATOR:

“--” symbol is used to decrease the value by one. It has two types:

1.post-fix decrement operator

2.pre-fix decrement operator

POST_FIX DECREMENT OPEATOR:

“--” symbol is used to represent post-fix decrement operator, this symbol is used after the operand. In this operator, value is first assigned to a variable and then decrement the value.

EX: int a,b;

```
a=10;
```

```
b=a--;
```

In the above example first the value of “a” is assign to the variable ” b”, then decrement the value. So the value of “b” variable is “10”.

PRE-FIX DECREMENT OPERATOR:

“--” Symbol is used to represent the pre-fix decrement operator. This symbol is used after the operand. In this operator, value is decremented first and then decremented value is used in expression.

EX: int a,b;

 a=10;

 b=--a;

In the above example first the value of “a” is decrement then assign to the variable “b”. So the value of b variable is “9”.

Conditional Operator:-

An alternative method to use if-else Construct is the conditional Operator statement. It is also called the ternary Operator, which Operates on three Operands.

The character pair (**? And :**) is a ternary operator available in C. this operator used to construct conditional expressions.

False
┌───────────┴───>
Syntax:-exp1 ? exp2 : exp3;
└───────────┬───────────┘
 True

Here the exp1 is evaluated first, if it is true then the value of exp2 is the result; otherwise the exp3 is the result.

EX: - int a=50, b=60;

 Z= (a+b)? a : b;

Bitwise Operators:

Bitwise operators are used to perform bit-level operations. Bitwise operators in 'C' are given below

Operators	Meaning of operator
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
<<	Bitwise Left shift
>>	Bitwise Right shift

Bitwise AND (&) operator:

The output of bitwise AND is 1 if corresponding bits of two operands is 1. If either bit of an operand is zero, the result of corresponding bit is evaluated to zero.

example:

Bit operation of 8 and 4.

$$8 = 1000$$

$$4 = 0100$$

$$\begin{array}{r}
 1000 \\
 \& 0100 \\
 \hline
 0000 = 0 \text{ (In decimal)}
 \end{array}$$

Bitwise OR ($|$) operator:

The output of bitwise OR is one if at least one corresponding bit of two operands is 1.

example:

Bitwise OR operation between 8 and 4

$$\begin{array}{r} 1000 \\ | 0100 \\ \hline 1100 = 12 \text{ (in decimal)} \end{array}$$

Bitwise XOR (exclusive OR):

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by (\wedge).

example:

Bitwise exclusive-OR operation between 8 and 4

$$\begin{array}{r} 1000 \\ \wedge 0100 \\ \hline 1100 = 12 \end{array}$$

Bitwise Complement operator (\sim):

Bitwise Complement operator is a unary operator (works on only one operand). It changes 1 to 0 and 0 to 1.

example:

Bitwise Complement operation of 8 is

$$\begin{aligned} 8 &= 1000 \\ \sim 8 &= 0111 \Rightarrow 7 \text{ (in decimal)} \end{aligned}$$

Bitwise Right shift operator (>>) :-

Right shift operator shifts all bits towards right by certain number of specified bits.

example:

Shifting the number 8 by 2 bits i.e., $8 >> 2$

$8 = 1000$

$8 >> 2 = 0010$

$8 >> 3 = 0001$

left shift operator (<<) :-

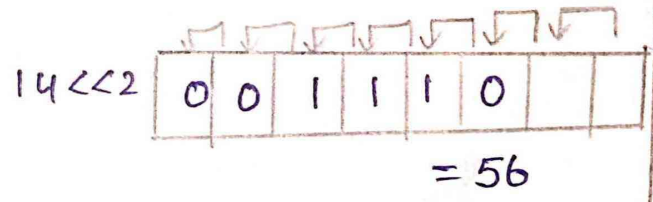
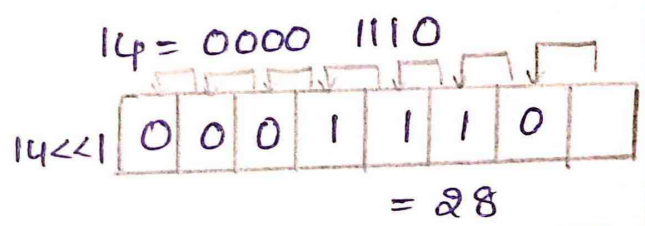
Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by left shift operator are filled with zero.

example:

Shifting the number 14 by 2 bits i.e., $14 << 2$

$14 = 0000 1110$

$14 << 2 = 001110$



C-tokens

C tokens are the basic building blocks in C language which are constructed together to write a C program.

Tokens :- An individual smallest unit of a program is called "Tokens". By using the tokens making the program very easily.

In 'C' Tokens are classified into 'six' types. They are

- Keywords
- Identifiers
- Constants
- Strings
- Special symbols
- Operators

* Keywords := Keywords are Reserved (or) Pre-defined (or) fixed words in a C-Compiler.

- Once we used the keyword which cannot change during the execution of the program.
- All keywords must be written in lowercase.
- Since keywords are reserved names for compiler, they can't be used as variable name.
- In 'C' there are '32' keywords.
- Eg: auto, break, char, do, for, int, ifelse, float, while etc

* Identifiers := Each program elements in a C-program are given a name called Identifiers. Names given to identify variables, functions and arrays are examples for identifiers.

Eg: `int x = 10;`

here `x` is a name given to integer type variable in above statement.

• Rules for Identifiers:-

The following are the Rules for Identifier.

- First character should be an alphabet or underscore.
- Identifiers does not includes backspace.
- It cannot allow the duplicate value.
- It cannot takes the keyword as the variable name.
- Length of identifiers is 31 characters.
- **variables**: The name of memory location is called variables.

- It is used to store any datatype value.
- variables are changeable, we can change the value of variable during the program execution.

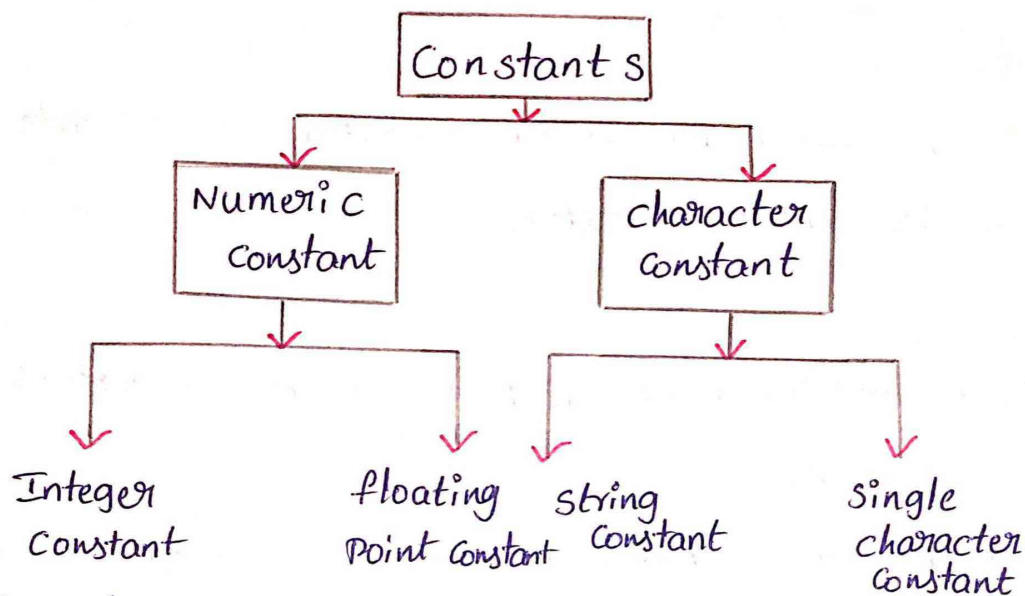
Types of variable :

- * Local variable
- * Global variable
- * static variable

* **Constants** := Constants are fixed value. That cannot be changed during program execution.

It is similar to a variable but it cannot be modified by the program once defined.

There are two types of constants in C.



• **Examples :-**

Integer : 250, 25, 101 etc.

floating point : 2.34, 93.95, 1045.00 etc.

String : "SITAMS", "MCA", etc

single character : 'a', 'b', 'x' etc

* **Strings** := Sequence of characters enclosed with double quotations is known as strings.

“

* Special symbols :=

In 'C' generally, the special symbols have some special meaning. Some of the special symbols that are used in C program. They are:

#, { }, , (), ;, : etc.

* Operators :=

Operator is a symbol by using operator to operate with operands is called operator.

Eg: $5 + 3 = 8$

- here 5, 3, 8 are operands and +, = are operators in the above statement.

The following are the different types of operators in C.

- Arithmetic operators - Eg: +, -, *, /, %
- Logical operators - Eg: &&, ||, !
- Relational operators
(or) Comparison operators - Eg: <, >, <=, >=, ==, !=
- Assignment operators - Eg: =
- Increment & Decrement operators - Eg: ++, --
- Bitwise operators - Eg: &, ||
- Special operators - Eg: , { }, ()
- Conditional operators - Eg: :, ?

Type Conversion:

Type conversion is the process of converting one data type to another data type. The type conversion is only performed to those data types where conversion is possible. Type conversion is performed by a compiler. In type conversion the destination data type can't be smaller than the source data type.

Types of type conversion:

There are 2 kinds of type conversion.

1. Implicit type conversion
2. Explicit type conversion

Implicit Type Conversion:

In implicit type conversion, the value of one type is automatically converted to the value of another type.

Ex: #include <stdio.h>

```
int main()
```

```
{ int x = 10;
```

```
  char y = 'a';
```

```
  x = x + y;
```

```
  float z = x + 1.0;
```

```
  printf("x = %d, z = %.f", x, z);
```

```
  return 0;
```

```
}
```

Output:

x = 107

z = 108.000000

Explicit Type conversion:

In explicit type conversion, we manually convert values of one data type to another type.

Ex: #include <stdio.h>

```
int main()
```

```
{ float a = 1.5;
```

```
  int b = (int)a;
```

```
printf ("a = %.f\n", a);  
printf ("b = %.d\n", b);  
return 0;  
}
```

Output:
a = 1.500000
b = 1

Type casting:

Type casting is the process of converting a variable from one data type to another datatype.

Types of Type casting:

There are two types of type casting

1. Implicit Type Casting
2. Explicit Type casting

Implicit Type Casting:-

The conversion of a smaller data type to a larger data type is known as Implicit type casting.

Ex:-

```
#include <stdio.h>  
#include <conio.h>  
void main ( )  
{  
    int a=15, b=2;  
    float div;  
    div = a/b;  
    printf ("the result is %.f", div);  
    getch();  
}
```

Explicit type casting:

The conversion of larger data type to a smaller data type is known as explicit type casting.

Ex:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    double n = 25.5;
    int res = (int)n;
    printf ("The value is res %d", res);
    getch();
}
```

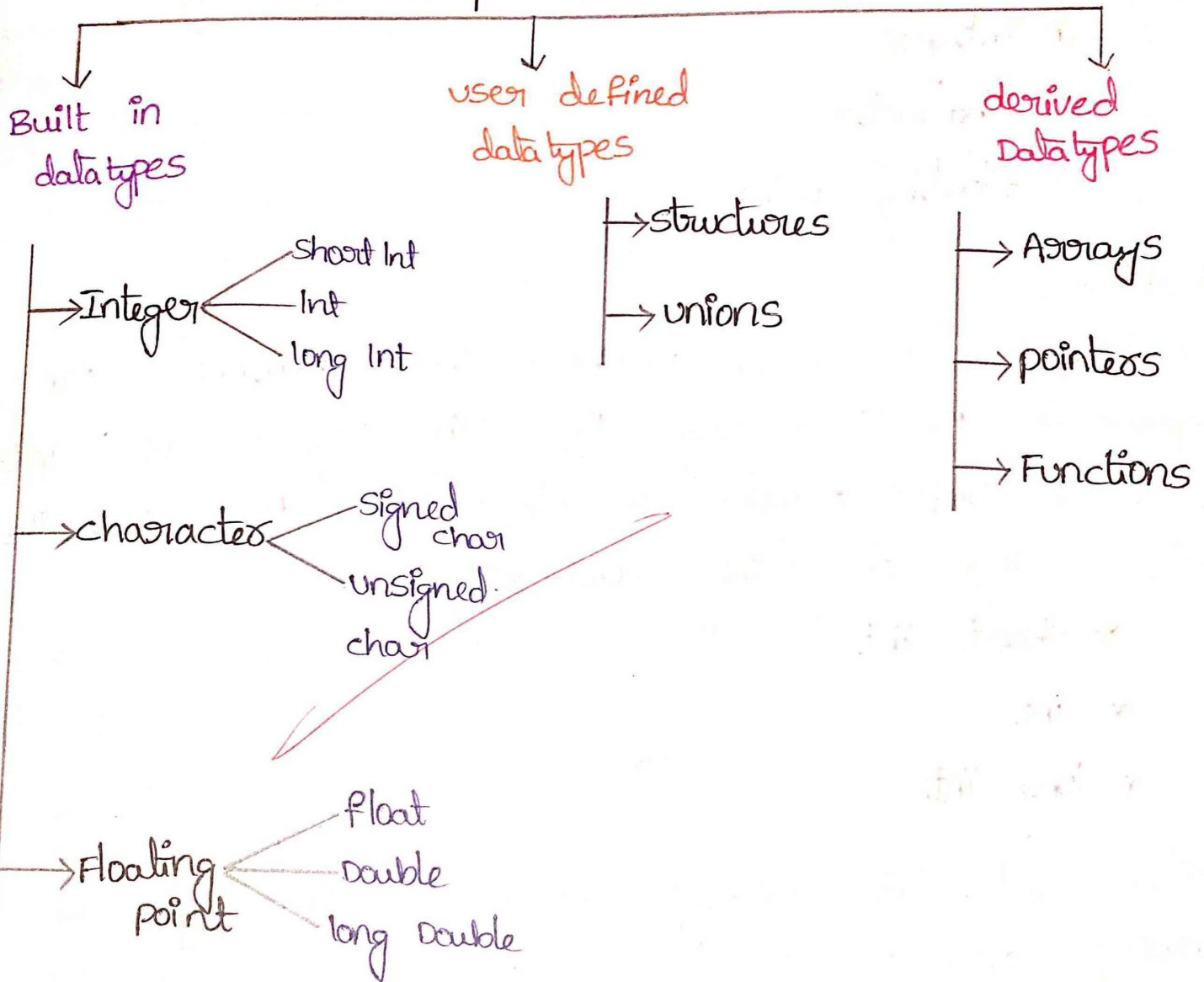
DATA TYPES IN 'C'

① Definition of Datatype :-

Any data value which can presents in a variable is called datatype

Datatypes are classified into three types

Datatypes



Built in Datatype:-

Built in datatype means already to develop the code by software.

* Built in datatype is also called as pre-defined data type [or] Basic datatype [or] primitive datatype [or] primary datatype.

Built in datatype basically divided into 3 types.

- ① Integer
- ② character
- ③ floating point

* Integer:-

Integers or whole numbers those numbers are represent in C-language by using key words "int". In C, we have 3 types of integer data to control the various range of integer numbers.

* short int

* int

* long int

⇒ Short int :- This datatype requires "one byte" of memory and it can store numbers ranging from -128 to +127. short int can be divided into 2 types.



signed short int :-

signed short int can store positive and negative values.

unsigned short int :-

unsigned short int can be stored only positive values.

* If it is unsigned it can store only positive numbers 0 to 255

⇒ int :-

This data type requires "two bytes" of memory and it can store numbers ranging from -32,768 to 32,767. If it is unsigned it can store only positive numbers 0 to 65,535.

⇒ long int :-

This data type requires "4 bytes" of memory and it can store -2,147,483,648 to +2,147,483,647. If it is unsigned, it can store only positive numbers from 0 to 4,294,967,295

Example :- $i=1, j=10$

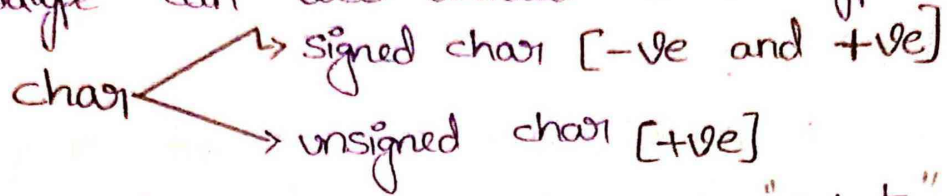
Character :-

This datatype is used to store 'single character' in character variable. char datatype can be enclosed with 'single quotes'. character can be stored in "1 byte".

This datatype can be represented with a keyword "char".

* this type can store ^{range} value -128 to +127. if it is unsigned it can be store only positive values from 0 to 255.

character datatype can also divided into 2 types.



* signed & unsigned char can store the "1 byte" of memory.

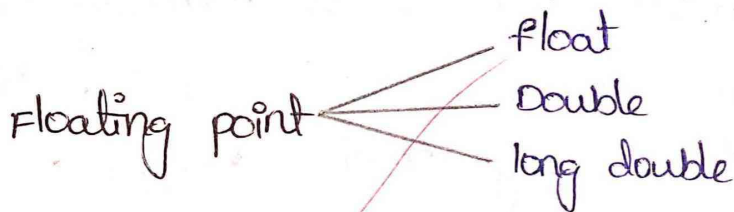
Ex:- 'A', 'B', '\$'.

Floating point :-

floating point numbers are stored in "four bytes" of memory with 6 digits precision.

These numbers are represented by using a keyword "float". The ranging from 3.4×10^{-38} to 3.4×10^{38} .

* floating point can be divided into 3 types.



⇒ Double :-

Double data type occupies "8 bytes" of memory with 14 digits precision.

* These numbers are represented by using a keyword "double".

* The Double datatype can range from 1.7×10^{-308} to 1.7×10^{308} to extend the precisions.

long double:-

'c' supports "long double datatype". it requires "10 bytes" of memory and it can store numbers ranging from $3.4e-4932$ to $3.4e+4932$.

Examples:-

- * 10.74 can be stored in a variable using float datatype.
- * 10.793554 can be stored in a variable using double data type.
- * 10.743554898 can be stored in a variable using long double datatype.

User defined datatypes:-

user defined datatype is used to create new datatype from the user requirement.

user defined datatype can be classified into 2 types.

① Structures

② Unions.

⇒ Structures:-

A group of one (or) more variables of different datatypes organised in a single name (or) unit is called as a structure.

* structure can require lots of memory space.

Syntax:-

```
struct structure_name  
{
```

```
<datatype> member 1 ;  
<datatype> member 2 ;  
  ⋮  
<datatype> member n ;  
};
```

Ex.:-

```
struct student  
{  
  int sno;  
  char name[20];  
  int marks ;  
  char address [50];  
};
```

Unions :-

A union is a user defined datatype which may hold members of different sizes and type.

* union uses a single memory location to hold more than one variable.

* union can be defined by the keyword "union"

* unions can be required less memory space.

Syntax:-

```
union union_name  
{  
  <datatype> member 1 ;  
  <datatype> member 2 ;  
  ⋮  
}
```

Derived datatype :-

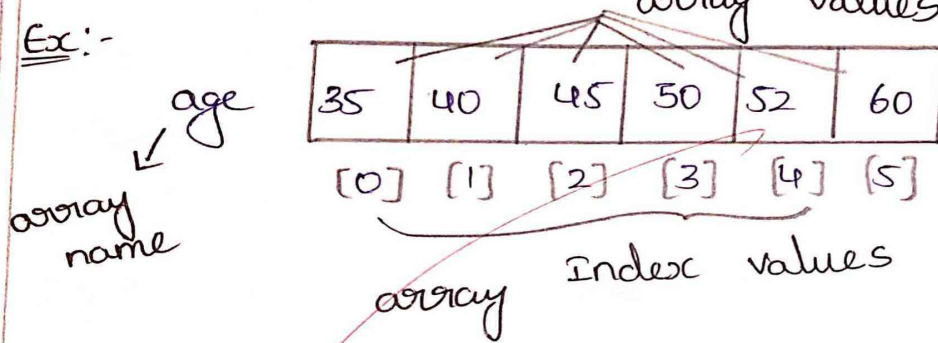
derived datatype can be classified into 3 types.

- * Arrays
- * pointers
- * Functions.

⇒ Arrays :-

An array is a group of logically related data items can be stored with in the common name with different address for their continuous memory allocation.

Ex:-



pointers :-

pointers is a variable which can hold address of another variable. It can represent $[*p]$

Ex:-

datatype

$\text{int } a = 10;$ → variable to assign value.
 $\text{int } (*p);$ → Another variable.

$\text{int } *p = \&a$ → address of variable

Functions: function is a subprograms segments that carries out a well defined tasks.

Syntax:-

```

function (a, b)
{
    // body of the loop
}
fun (10, 5)

```

function definition
 formal arguments
 actual arguments.

write a c program for built-in datatype.

```

#include <stdio.h>
void main()

```

```

{
    int a;
    float b;
    char c;
    double d;

```

```

    a = 10;
    b = 20.50;
    c = 'H';
    d = 10.2023231;

```

O/P:-

```

integer = 10
float = 20.50
character = H
double = 10.2023231

```

```

printf ("In Integer = %d", a);
printf ("In float = %.f", b);
printf ("In character = %c", c);
printf ("In double = %.f", d);
?

```

Strings functions

String:

String is Collection (or) group of (or) sequence of characters is called string.

A string is a sequence of characters terminated with null character '\0'

Syntax:-

```
Char name[];
```

Eg:-

```
char str[] = "College"
```

Index → 0 1 2 3 4 5 6
String →

c	o	l	l	e	g	e	\0
---	---	---	---	---	---	---	----

Address →

It can represent a string of maximum '7' characters '7' place including '\0' or '0'.

String handling functions:-

1. Strlen()
2. Strcpy()
3. Strcat()
4. Strcmp()
5. Strupr()
6. Strlwr()
7. Strrev()

1. Strlen():-

Strlen() is a function returns the length of the given string. It returns the integer value.

Syntax:

```
Strlen("string");
```

Ex:-

```
String s1 = "program";
```

```
int n = Strlen[program];
```

```
Length = 7;
```

Program:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
    char * str = "SITAMS";
    int strlen;
    strlen = strlen(str);
    printf("%d", strlen);
}
```

output: 6

2. strcpy():

strcpy() function is used to copy one string to another string. It takes two arguments like source and destination.

It copies the source string values into destination string.

Syntax:

```
strcpy (Destination-string, Source string);
```

Program:

```
#include <stdio.h>
#include <string.h>
void main()
{
    char * Destination-string;
    char * source-string = "SITAMS";
    strcpy(Destination-string, source-string);
    printf("%s", Destination-string);
}
```

output = SITAMS.

3. strcat()

strcat() is used to concatenation of two string. It takes two string₁, string₂ but it will store the result in one string.

Syntax:

```
strcat (string1, string2)
```


Program:

```
#include <stdio.h>
#include <string.h>
void main()
{
    char * string1 = "good";
    char * string2 = "morning";
    strcat(string1, string2);
    printf("%s", strcat);
}
```

output: good morning.

4. strcmp;

strcmp() function is used to compare two strings. It takes two arguments like string1, string2. If both strings have same values it will return '0'.

Syntax:

```
int strcmp(string1, string2);
```

Program:

```
#include <stdio.h>
#include <string.h>
void main()
{
    char * string1 = "Chittoor";
    char * string2 = "chittoor";
    int = return;
    return = strcmp(string1, string2);
    printf("%d", return);
}
```

Output = 0

5.struprc):

struprc() is used to convert the lowercase into uppercase values.

Syntax:

```
struprc("string");
```

Program:

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

```

    uppercase
    strlen(c);
    printf (" %d", strlen);
}

```

Output: CHITTOOR

6. Strlwr()

This function is used to Convert the uppercase string to lowercase string.

Syntax:

```
Strlwr() = ("string");
```

Ex: SITAMS

```
char s[7] = ("SITAMS");
```

```
Strlwr();
```

↳ char s[7] = ("sitams");

7. Strrev:-

This function is used to reverse the given value or string.

Syntax:

```
Strrev() = ("string");
```

Program:

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

```
#include <string.h>
```

```
void main()
```

```
{ int rev;
```

```
char * string = "Gopi";
```

```
Strrev();
```

```
printf ("%d", rev);
```

```
getch();
```

```
}
```

Output: iPOG

Standard functions:-

Standard functions are functions that have been pre-defined by 'C' and put into standard libraries.

Ex:- printf(), scanf().. etc.

what we need to do to use them is to include the appropriate header files.

Ex:-

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

It contained in the header files are the prototypes of the standard functions. The function definitions (the body of the functions) has been compiled and put into a standard C library which will be linked by compiler during compilation.

Printf:-

- ↳ Printf (Print formatted) in C
- ↳ It writes out a Cstring to standard output
- ↳ printf() function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen

use:-

- %d - value of an integer variable
- %c - value of an character variable
- %f - value of an float variable
- %s - value of a string variable

scanf:-

- ↳ It is used to take input from the users
- ↳ C function to read input from the standard input until encountering a whitespace or newline
- ↳ scanf function takes the format string and variables with their address as parameter
- ↳ scanf reads data according to the format specifier.

getch():-

It takes the program to hold the output screen for some time until the user passes a key from the keyboard to exit the console screen.

↳ getch() function doesn't take any parameters.

↳ getch() reads a single character from the keyboard.

puts:-

↳ A 'C' library function that writes a string to stdout or standard output.

↳ Declaration is `int puts
(const char *str)`

↳ Helps to display a string on a standard output device.

↳ Returns a non-negative value if successful; if unsuccessful, it will return Null.

Return 0:-

↳ A way to exit a function.

→ Return 0, in this case, means that the main() function terminates normally and returns to the windows operating system.

clrscr :-

↳ The clrscr in C is a built-in function that is used for clearing the screen of the console output during the execution of the 'C' program.

↳ This function is defined in the conio.h header file. We need to include the "conio.h" file for using the clrscr in 'C' programs.

Syntax:-

`clrscr();`

Control Structure

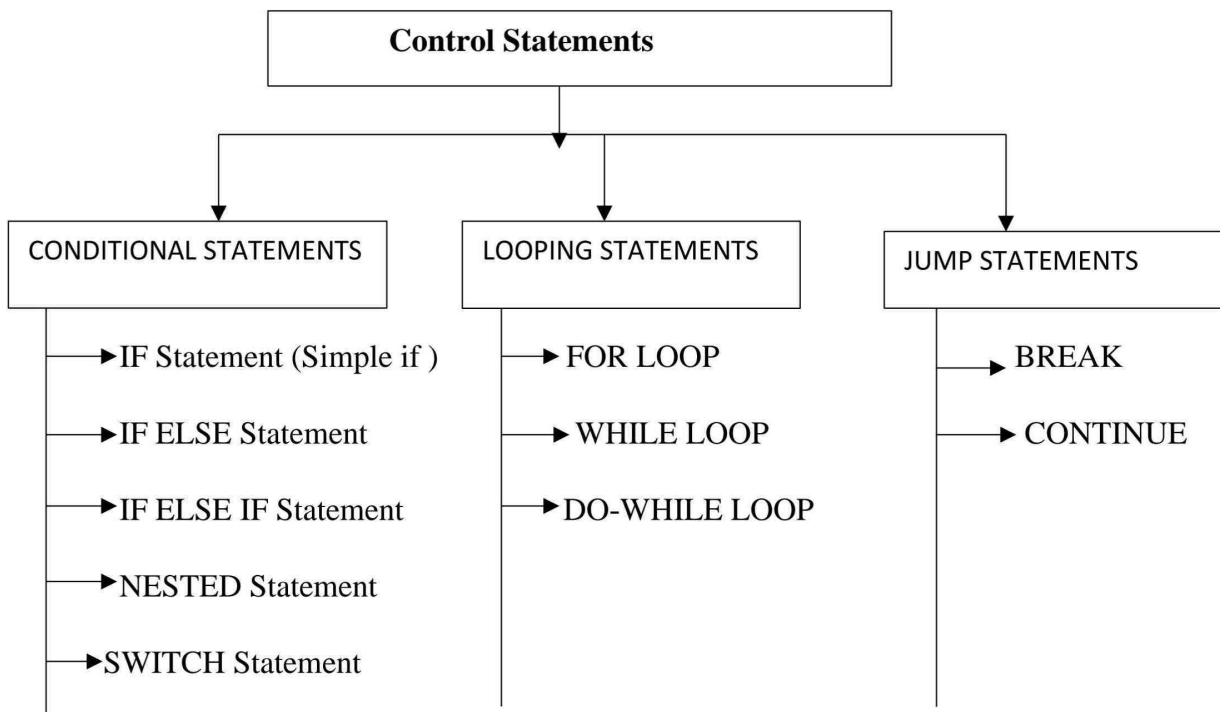
Simple Sequential programs, Conditional Statements (if, if-else, switch statements), Loops (for, while, do-while)

Break and Continue statements

1. Explain about Control Structures (or) Control Statements in C?

CONTROL STATEMENT:

- The C program is a set of statements which are normally executed sequentially. However, we have a number of situations we may have to change the order of execution of statements based on the certain conditions (or) repeat a group of statements until certain specified conditions are met.
- This involves a kind of decision making then the computer to execute certain statements.
- Control Structures are mainly can be classified as 2 types
 1. Branching statements (Conditional statements)
 2. Looping statements (Iterative statements)
 3. Jump statements



CONDITIONAL STATEMENT or BRANCHING STATEMENTS:-

- When a program breaks the sequential flow and jumps to another part of the code its called “Branching statement”.
- When the branching is based on a particular condition it’s know as “Conditional Branching”.If branching takes place without any decision it’s know as “Unconditional Branching”.
- C language possesses such decision making capabilities are know as “Control (or) Decision making Statements”. They are...
 - **If statements**
 - **Switch statement**

IF STATEMENT(Simple if) :-

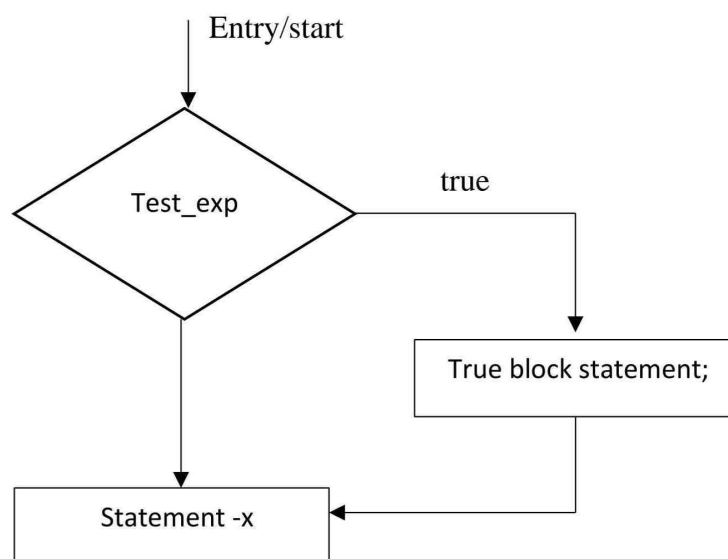
- Simple if is one of the decision- making statement but it is “one- way decision making statement”.

Syntax:-

```
if (Test_Expression)
{
    Statements;
    -----
    -----
}
Statement – X;
```

- The Test_Expression should always be enclosed in parentheses.
- If Test_Expression is true then the statements are executed otherwise control will passes to the next statement i.e (statement –x) followed by

FLOW CHART:



EXAMPLE 1:

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    int age;
    clrscr();
    printf("enter the person age");
    scanf("%d",&age);
    if(age>=18)
    {
        printf("Eligible for vote");
    }
    getch();
}
```

Example 2:

```
// Program to display a number
#include <stdio.h>
#include<conio.h>
int main()
{
    int n;
    printf("enter a number");
    scanf("%d",&n);
    if(n>0)
    {
        printf("the number is:%d",n);
    }
    return 0;
}
```

If-else statement:-

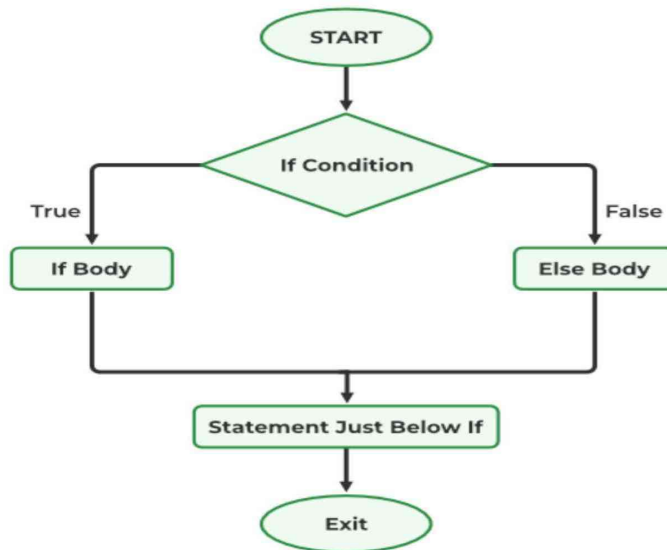
- It is an extension of the "Simple if statement but it is a two – way decision making statement.
- The if-else statement will execute a single or group of statements when the test expression is true. It performs else block statements when the test expression fails.

Syntax:

```
if(Test_Expression)
{
    True block of statements;
    -----
    -----
}
else
{
    False block of statements;
    -----
    -----
}
Statement- x;
```

- If the Test_Expression is true (non-zero) the if part is executed and control passes to the next statement following the if construct.
- Otherwise the else part is executed and control is passed to the next statement (statement –x).

FLOW CHART:



Example Program:

```

void main()
{
    int m1,m2,m3,total;
    float avg;
    clrscr();
    print("enter any 3 subjects marks");
    scanf("%d%d%d",&m1,&m2,&m3);
    total=m1+m2+m3;
    avg=total/3;
    if(m1>=35 && m2>=35 && m3>=35)
        printf("result is pass");
    else
        print("result is fail");
        print("total marks is %d", total);
        printf("average marks is %f", avg);
    getch();
}
  
```

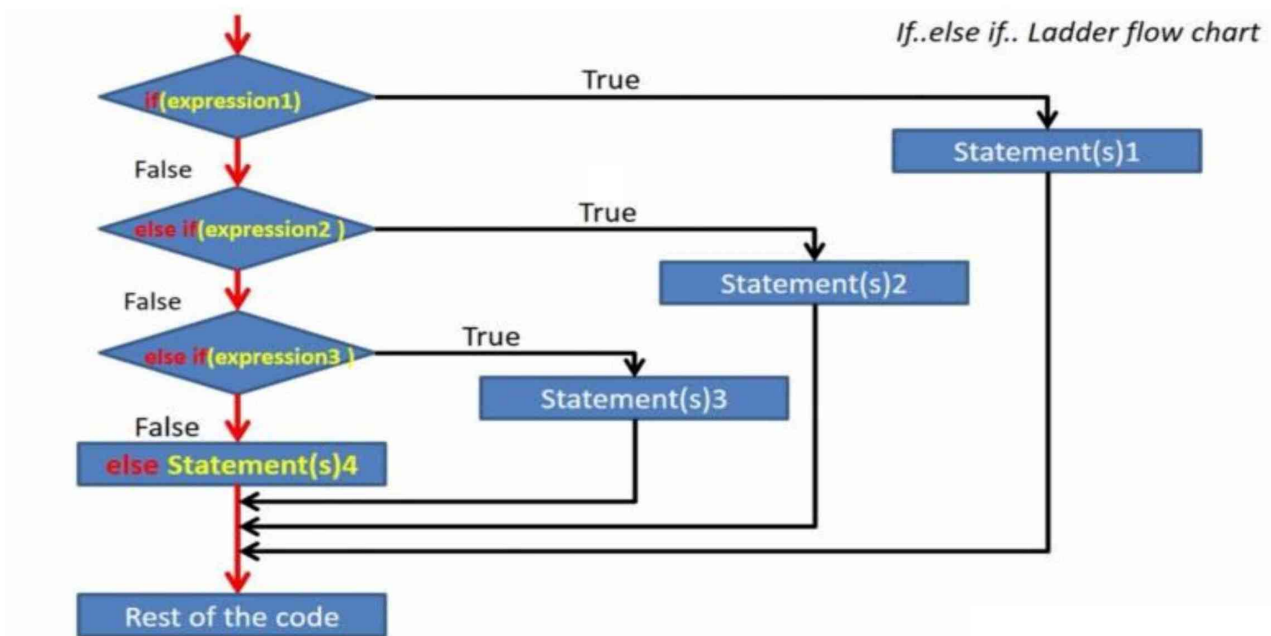

Else – If ladder (or) If-else-if Statement:-

- Multi way decisions arrive when there are multiple conditions and different actions to be taken under each condition.
- It can be return by using if-else construct as follows.

Syntax : if..else.. ladder statement

```
if (expression1) ← First check this expression
{
  // statement(s)
}
else if (expression2) ← Check this expression only if above expression
                       is false
{
  // statement(s)
}
else if (expression3) ← Check this expression only if above expression is
                       false
{
  // statement(s)
}
.
else
{
  // statement(s) ← Execute these statements only if all the above
                    expression checks are false.
}
```

FLOW CHAT:



- The conditions are evaluated from the top of the ladder to downwards. As soon as true condition is found, that associated statements executed and the control is transfer to the statement-x.
- When all conditions becomes false, then final else containing default statement will be executed.

EXAMPLE PROGRAM:

```
int main()
{
    int a, b, c;
    printf("Enter the numbers a, b and c: ");
    scanf("%d %d %d", &a, &b, &c);
    if (a > b && a > c)
    {
        printf("%d is the largest number.", a);
    }
    else if (b > a && b > c)
    {
        printf("%d is the largest number.", b);
    }
    else
    {
        printf("%d is the largest number.", c);
    }
    return 0;
}
```

NESTED IF-ELSE-STATEMENTS:-

- With in If statements we will define more than one If statements called “Nested If – else Statement”

Syntax:-

```
if(Test_condition -1)
{
    if(Test_condition -2)
        Statement block-1;
    else
        Statement block-2;
}
else
    statements block-3;
```

- If the Test_condition -1 is false, then the statement block- 3 will be executed. Otherwise it continues to perform the second condition.
- If the Test_condition –2 is true, then the statement block- 1 will be evaluated otherwise statement block – 2 will be evaluated and control is transferred to the statement –x.

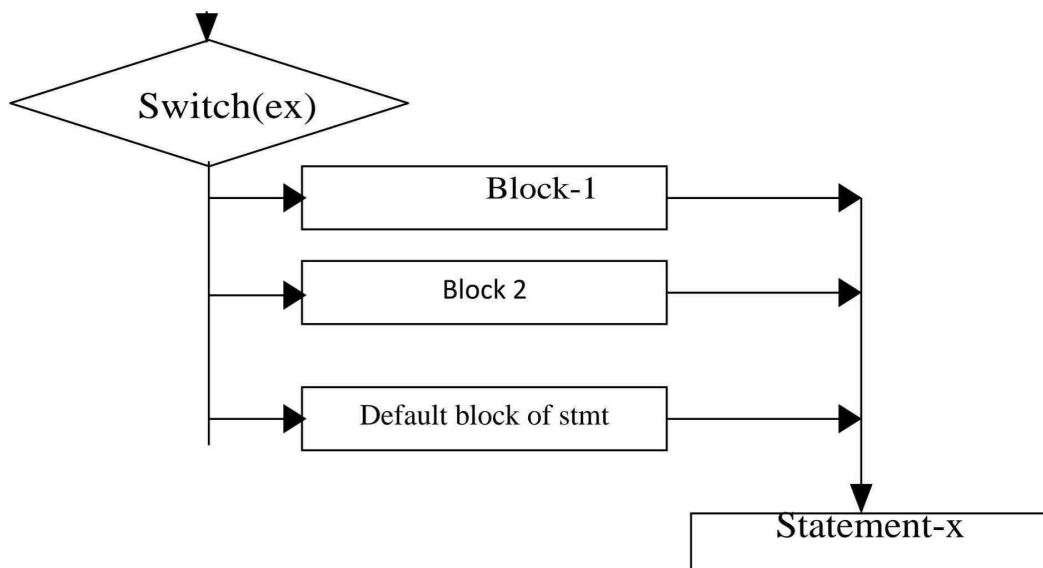
EXAMPLE :

```
void main()
{
    int n;
    printf("enter a number")
    scanf("%d",&n);
    if(n>0)
    {
        If(n%2==0)
            Printf("even number");
        else
            printf("odd number");
    }
    else
        printf("number is not suitable");
}
```

SWITCH STATEMENT:-

- ❖ The switch statements provide clear way to dispatch to different parts of a code based on the value of single variable or expression.
- ❖ It is multi-way decision making construct that allows choosing of a statement or group of statements among several alternatives.
- ❖ The control flow in the switch statement executes the specified statements when the constant values are matched with the expression.
- ❖ The switch statement is mainly used to replace multiple if-else sequences which are hard and maintain.

FLOW CHAT:



Syntax:-

```
Switch(Expression)
{
Case value-1:
    Block-1 of Statements;
    break;
case value-2:
    Block-2 of Statements;
    break;
-----
-----

default:
    Default block of Statements;
    break;
}
Statement-x;
```

- ❖ The expression following the switch keyword is an integer valued expression. The value of the expression decides the sequence of statements to be executed.

Each statements of sequence begins with the keyword case followed by a constant integer or character (value-1, value-2,...) control is transferred to the block of statements(block-1,block2,.....) following the case label whose constant is equal to the value of the expression in the switch statement.

- The default part is an optional case, it will be executed if the value of the expression does not match with any of the case values.
- The keywords break at the end of each block signal the end of a particular case and exit from the switch statement.

Notes:

- If we do not use the BREAK statement, all statements after the matching label are also executed.
- The default clause inside the SWITCH statement is optional.

EXAMPLE :-

```
#include <stdio.h>
int main()
{
    int week;
    printf("Enter week number(1-7): ");
    scanf("%d", &week);
    switch(week)
    {
        case 1:
            printf("Monday");
            break;
        case 2:
            printf("Tuesday");
            break;
        case 3:
            printf("Wednesday");
            break;
        case 4:
            printf("Thursday");
            break;
        case 5:
            printf("Friday");
            break;
        case 6:
            printf("Saturday");
            break;
        case 7:
            printf("Sunday");
            break;
        default:
            printf("invalid number.");
    }
    return 0;
}
```

2.Explain About Looping Statements?

- The process of repeatedly executing a block of statements is known as “looping”.
- A program consists of 2 segments
 - Body of the loop
 - Control statement
- The control statement tests certain conditions and then dissects the repeated execution of the statements contained in the body of loop.
- Depending on the position of the control statements in loop can be classified as
 - Entry – controlled loop
 - Exit – controlled loop
- C language provides for 3 constructs for performing loop operations. They are
 - For loop
 - While loop
 - Do-while loop

For loop Statement:-

- The for loop is another “entry – control loop” that provides a more concise loop control structure.
- The for loop is useful while executing statements a fixed number of times. The control flow of the for loop is

Syntax:

```
for (initialization ; test condition; increment /decrement)  
{  
    Statement -1;  
    -----  
    Statement – n;  
}  
Statement – x;
```

- The for statement is compact way to express a loop. All the three parts of the loop are in close proximity with the for statement.
- The initialization part is executed first but only once. Next the test condition is executed. If the test evaluates to false, then the next statement after the for-loop is executed (means the statement – x is executed).

- If the test expression evaluates to true, then after executing the body of the loop, the increment and decrement part is executed.
- The test evaluated again and the whole process is repeated as long as the test expression evaluates to true.

Nested for loop (Nesting of for loops):-

- Nesting of loops means one for loop statement with in another for loop statement is allowed in C language.

Syntax:-

```

for(initialization; test_condition; increment/decrement)
{
    for(initialization; test_condition; increment/decrement)
    {
        -----
        ----- // body of inner loop
    }
    ----- // body of outer loop
}

```

Statement – X;

The loops should be properly indented so as to enable the reader to easily determine which those statements are contained with in each for loop statement.

SYNTAX -2 ;

```

Outer_loop
{
    Inner_loop
    {
        // inner loop statements.
    }
    // outer loop statements.
}

```

EXAMPLE :

```

include <stdio.h>
int main()
{
    int n;    // variable declaration
    printf("Enter the value of n :");
    for(int i=1;i<=n;i++)    // outer loop

```

```

    {
    for(int j=1;j<=10;j++) // inner loop
    {
        printf("%d\t",(i*j)); // printing the value.
    }
    printf("\n");
}

```

WHILE LOOP:-

- The simplest of all looping statements in C is the “while loop statement” and while is an entry-controlled loop statement.
- The while loop is used when the number of iterations to be performed are not known in advance. The Control flow in while loop is start from the expression and execute the specified statement within the loop as long as it remains true.

Syntax of while loop is..

Syntax:

```

Initialization;
While (Test_condition)
{
    Statement - 1;
    -----
    Statement- n;
    Increment/decrement
    statement;
}
Statement- x;

```

- The Test_condition is evaluated and if the condition is true, then the body of loop is executed and the Test_condition is once again evaluated and if it's true, the body of loop is executed once again.
- The process of repeated execution of the body continues until the Test_condition finally becomes false and control is transferred out of the loop means statement -x is executed.

EXAMPLE:

```
void main()
{
    int i ;
    i=1;
    while(i>0)
    {
        Printf(“display numbers”,i);
        i++;
    }
}
```

Do – while loop Statement:-

- The while loop construct makes a test condition before the loop is executed. On some occasions it might be necessary to execute the body of the loop before the test condition is performed. Such situations can be handled with the help of “do while statement”.
- Some times it is desirable to execute the body of a loop at least once even if the test condition evaluates to false during the first iteration. In effects, this requires testing of termination expression at the end of the loop rather than the beginning as in the while loop.
- So the do-while loop is called a bottom expression loop. The loop is executed repeatedly as long as the test condition remains true.

Syntax :

Initialization;

do

{

Statement - 1;

.....

Statement – n;

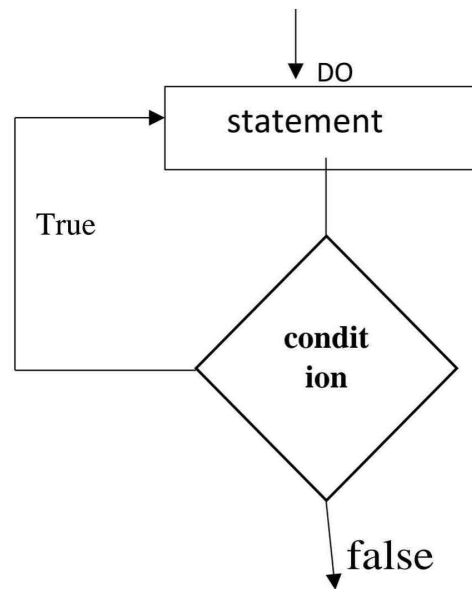
Increment / decrement statement;

}

while (Test_condition);

- In the above syntax the body of the loop evaluated first. At the end of the loop the Test_condition is evaluated. If it's true to evaluated the body of loop once again.
- This process continues as long as when the condition become false the loop will be terminated and the control goes to the after the “do – while statement” means statement – x is executed.
- Since the condition evaluated at the bottom of the loop, the do – while construct provides an exit – control loop and body of loop is always executed at least once..

FLOWCHART



Explain about Break and Continue statements with an example?

Jumps in loops:-


- Some times when executing a loop it becomes desirable to skip a part of the loop (or) to leave the loop as soon as certain conditions occur.
- C permits a jump from one statement to another statement to the end or beginning of a loop as well as a jump out of a loop.
- In C language jumping statements 2 types
 - Break statement
 - Continue

Break statement:-

- We have already seen the use of the break in the switch statement. The break statement can also be used within loops (like for, while, do while).
- A break statement constructs terminates the execution of loop and the control is transferred to the statement immediately following the loop (means control passes out side of the loop).
- The term break refers to the act of breaking out of a block of code. The control flow in for, while and do-while loop statements will be terminated by a break statement.

Syntax:

```
for(expression1;expression2;expression3)
{
    .....
    .....
    if(<condition>)
    {
        break
    }
    Statement – x;
```



- When the loops are nested the break would only exit from the loop containing it. It means “each break statement exit only a single loop”.

Continue statement:-

Continue statement is used to skip the current transaction and executed the next iteration is called as a CONTINUE statement.

- During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions and the execution continues with the next loop operation.
- C supports another statement “continue statement”. It causes the loop to be continued with the next iteration after skipping any statements in between the continue statement falls the compiler.
- The continue statement skips the remainder of the current iteration and initiate the execution of the next iteration. If this statement encountered in a loop, the rest of the statements in the loop are skipped and the control passes to the condition which is evaluated. If the condition is true, the control enters in to the loop again.

Syntax:

```
for(expression1; expression2;expression3)
{
    .....
    .....
    if(<condition>)
        continue;
    .....
    .....
}
Statement – x;
```

EXAMPLE :

```
#include <stdio.h>
int main()
{
    int i, n;
    printf("Print odd numbers till: ");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        if(i%2=0)
        {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To create an array, define the data type (like int, string...etc) and specify the name of the array followed by square brackets [].

There is two types of arrays Single dimensional array and Multidimensional array

Single dimensional array

One-Dimensional Array is a group of elements having the same data type which are stored in a linear arrangement under a single variable name.

To insert values in single dimensional array

```
Int myNumbers[] = {25, 50, 75, 100};
```

0	1	2	3
25	50	75	100

Access the Elements of an Array

To access an array element, refer to its index number.

Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

This statement accesses the value of the first element [0] in myNumbers:

Example

```
#include <stdio.h>
int main()
{
    int numbers[5] = {10, 20, 30, 40, 50};
    for(int i=0; i<5; i++)
    {
        printf("numbers[%d] = %d\n", i, numbers[i]);
    }
    return 0;
}
```

Output

```
Numbers[0] = 10
Numbers[1] = 20
Numbers[2] = 30
Numbers[3] = 40
Numbers[4] = 50
```

Multidimensional Arrays

A multi-dimensional array is an array with more than one level or dimension. For example, a 2D array, or two-dimensional array, is an array of arrays, meaning it is a matrix of rows and columns (think of a table)

Arrays can have any number of dimensions.

Two-Dimensional Arrays

A 2D array is also known as a matrix (a table of rows and columns).

To create a 2D array of integers, take a look at the following example:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

The first dimension represents the number of rows [2], while the second dimension represents the number of columns [3]. The values are placed in row-order, and can be visualized like this:

	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

Access the Elements of a 2D Array

To access an element of a two-dimensional array, you must specify the index number of both the row and column.

This statement accesses the value of the element in the **first row (0)** and **third column (2)** of the **matrix** array.

Example

```
Int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

```
Matrix[0][0] = 9;
```

```
Printf("%d", matrix[0][0]); // Now outputs 9 instead of 1
```

Loop Through a 2D Array

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following example outputs all elements in the matrix array:

Example

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

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

```
Main()
```

```
{
```

```
Int a[3][3],b[3][3],c[3][3];
Int i,j;
Printf("element for a matrix is:");
For(i=0;i<=2;i++)

{
    For(j=0;j<=2;j++)
    {
        Scanf("%d",&a[i][j]);
    }
}
Printf("element for b matrix is:");
For(i=0;i<=2;i++)
{
    For(j=0;j<=2;j++)
    {
        Scanf("%d",&b[i][j]);
    }
}

Printf("matrix addition is");
For(i=0;i<2;i++)
{
    For(j=0;j<=2;j++)
    {
        C[i][j]=a[i][j]+b[i][j];
    }
}
```



```
    }  
  }  
  For(i=0;i<=2;i++)  
  {  
    For(j=0;j<=2;j++)  
    {  
      Printf("%d\t",c[i][j]);  
    }  
    Printf("\n");  
  }  
}
```

Output:

Element for a matrix is:1

2

3

4

5

6

7

8

9

Element for b matrix is:1

2

3

4

5

6

7

8

9

Matrix addition is

2 4 6

8 10 12

14 16 18

FEATURES OF POINTERS:

- 1)Pointers save memory space.
- 2)Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location.
- 3)Memory is accessed efficiently with the pointers. The pointer assigns and releases the memory as well. Hence it can be said the Memory of pointers is dynamically allocated.
- 4)Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
- 5)An array, of any type, can be accessed with the help of pointers, without considering its subscript range.
- 6)Pointers are used for file handling.
- 7)Pointers are used to allocate memory dynamically.
- 8)In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.
- 9)A pointer is a variable that stores the memory address of another variable as its value.

POINTERS:

- 1.Pointer is a variable that stores a memory address.
- 2.pointers are used to store the address of other variables or memory items.
- 3.pointers are very useful for another type of parameter passing.
- 4.pointers are essential for dynamic memory allocation.
- 5.pointer variable points to a data type (like int) of the same type, and is created with the * operator

SYNTAX:

Datatype variable name;

Datatype *variable name;

Example

```
int myAge = 43;
int* ptr = &myAge;
printf("%d\n", myAge);
printf("%p\n", &myAge);
printf("%p\n", *my Age);
```

ARITHMETIC OPERATIONS WITH POINTERS:

Arithmetic operations in c is used to calculate math operations they are as follows

1) addition (+):

This operator is used to add two variables

2)substraction (-)

This operator is udes to find difference between two variables

3)multiplication (*)

This operator is used to multiply two variables

4)division (/)

This operator is used to find remainder

5)modulo (%)

This operator is used to find quotient

Example program of arithmetic operations with pointers

```
#include<studio.h >
#include<conio.h>
Void main()
{
Int a=25,b=10,*p,*j;
```

```
p=&a;
j=&b;
Printf("\n addition a+b=%d",*p+*j);
Printf("\n subtraction a-b=%d",*p-*j);
Printf("\n multiplicarin a*b=%d",p*j);
Printf("\n division a/b=%d",*p/*j);
Printf("\n mod a%b=%d",*p%*j);
}
```

OUT PUT:

addition:35

substraction:15

Multiplication:250

division:2

modulo:5

ARRAYS WITH POINTERS:

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it

SYNTAX:

Datatype variablename[size]:{array elements};

Example

```
int a[4] = {25, 50, 75, 100};
```

```
int i;
```

```
for (i = 0; i < 4; i++)  
{  
    printf("%d\n", a[i]);  
}
```

OUTPUT:

```
25  
50  
75  
100
```

Instead of printing the value of each array element, let's print the memory address of each array element.

Example:

```
int a[4] = {25, 50, 75, 100};  
  
int i;  
  
for (i = 0; i < 4; i++) {  
    printf("%p\n", &a[i]);  
}
```

OUTPUT:

```
0x7ffe70f9d8f0  
0x7ffe70f9d8f4  
0x7ffe70f9d8f8  
0x7ffe70f9d8fc
```

Two Dimensional Array of pointers in C:

A Two Dimensional array of pointers is an array that has variables of pointer type. This means that the variables stored in the 2D array are such that each variable points to a particular address of some other element.

SYNTAX :

```
datatype *variable name[size][size];
```

EXAMPLE:

```
int *arr[5][5];
```

pointer array of 5 rows and 5 columns.

The element of the 2D array is been initialized by assigning the address of some other element.

In the example, we have assigned the address of integer variable 'n' in the index (0, 0) of the 2D array of pointers.

```
int n;
```

```
arr[0][0] = &n;
```

position (0, 0)

Below is the implementation of the 2D array of pointers.

Example:

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

```
int main()
```

```
{
```

```
    int arr1[5][5] = { { 0, 1, 2, 3, 4 },
```

```
                    { 2, 3, 4, 5, 6 },
```

```
                    { 4, 5, 6, 7, 8 },
```

```
                    { 5, 4, 3, 2, 6 },
```

```
                    { 2, 5, 4, 3, 1 } };
```

```
    int* arr2[5][5];
```

```
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        arr2[i][j] = &arr1[i][j];
    }
}
printf("The values are\n");
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        printf("%d ", *arr2[i][j]);
    }
    printf("\n");
}

return 0;
}
```

Output:

The values are

0 1 2 3 4

2 3 4 5 6

4 5 6 7 8

5 4 3 2 6

2 5 4 3 1

ARRAYS OF POINTERS:

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

pointer_type: Type of data the pointer is pointing to.

array_name: Name of the array of pointers.

array_size: Size of the array of pointers.

Note: It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing `*array_name` in the parenthesis will mean that `array_name` is a pointer to an array.

Example:

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;
    int* ptr_arr[3] = { &var1, &var2, &var3 };
    for (int i = 0; i < 3; i++)
    {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }
}
```

```
    }  
    return 0;  
}
```

Output

Value of var1: 10 Address: 0x7fff1ac82484

Value of var2: 20 Address: 0x7fff1ac82488

Value of var3: 30 Address: 0x7fff1ac8248c

Explanation:

As shown in the above example, each element of the array is a pointer pointing to an integer. We can access the value of these integers by first selecting the array element and then dereferencing it to get the value.

Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.

Syntax:

```
char *array_name [array_size];
```

After that, we can assign a string of any length to these pointers.

Example:

```
char* arr[5] = { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }
```

This method of storing strings has the advantage of the traditional array of strings. Consider the following two examples:

Example :

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

```
int main()
```

```
{
    char str[3][10] = { "Geek", "Geeks", "Geekfor" };
    printf("String array Elements are:\n");
    for (int i = 0; i < 3; i++)
    {
        printf("%s\n", str[i]);
    }
    return 0;
}
```

Output

String array Elements are:

Geek

Geeks

Geekfor

In the above program, we have declared the 3 rows and 10 columns of our array of strings. But because of predefining the size of the array of strings the space consumption of the program increases if the memory is not utilized properly or left unused. Now let's try to store the same strings in an array of pointers.

POINTERS TO POINTERS

A pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer.

SYNTAX

```
int **p;
```

Consider the following example.

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

```
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a;
    pp = &p;
    printf("address of a: %x\n",p);
    printf("address of p: %x\n",pp);
    printf("value stored at p: %d\n",*p);
    printf("value stored at pp: %d\n",**pp);
}
```

Output

address of a: d26a8734

address of p: d26a8738

value stored at p: 10

value stored at pp: 10

Functions & Structures

Introduction:

- A function is a subprogram segment that carries out some specific well-defined tasks. Every C program consists of one or more functions. And all of those functions must be called as main function.
- A function is a subprogram that Modular Programming or, Black box analogy, Procedural Programming statements that can be processed independently.
- A function can be invoked by calling with its name the communication between caller (calling function) and callee (called function) takes place through parameters.
- The function can be designed, developed and implemented independently by different programmers.
- The independent functions can be grouped to form a software library.
- Functions are independent because variable names and labels defined within its body or local to it.
- The use of functions offers flexibility in the design, development and implementation of the programmer to solve complex problems.

Syntax: -

```
void myFunction() {  
    // code to be executed  
}
```

Function definition and Declaration: -

Definition: -

- Function itself is referred to as function definition.
- First line of a function definition is called as “function declaratory or declaration” and followed by the “function body”.
- Within function declaration to use same function name, no of arguments, arguments type and return types etc.
- This type of declaration’s cannot allow other “function declarations”.

Syntax: -

Return type function name (arguments list)

```
{  
    Statement1;  
    Statement2;  
    -----  
}
```

```
Statement N;  
}
```

Declaration: -

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

Syntax: -

A function declaration has the following parts –

return type function name (parameter list);

Ex: -

- For the above defined function max (), the function declaration is as follows –

```
int max(int num1, int num2);
```

- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max (int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Return statement: -

- The return statement in a function need not be at the end of the function.
- It can occur any where in the function body and as soon as it's encountered execution control will be return to the caller(calling function).

Syntax: -

```
return_type function_name (parameterslist)  
  
{  
-----  
Return statement;  
-----  
}
```

Types of functions: -

In C language functions can be classified as 2 types. They are

1. Built-in functions(library)
2. User defined functions

Built-in functions: -

- These functions are developed by the other software professionals and that's having some
- pre-defined meaning. According to our purpose we are calling our programs.
- In C language those functions are comes with the C library. These functions are also called pre-defined functions.
- Examples: - sqrt (), pow(), round(), strlen(), strcpy() etc.

User defined functions:-

- C formats the use of user-defined functions apart from the library functions. These functions are developed by the programmers according to our own purpose.
- This functions are used to when a set of instructions as kept as a block. When never we need the instructions we will call the user-defined functions from the main function or program.

Syntax:-

Return_type function_name (list of arguments or parameters)

```
{  
Statement1;  
Statement2;  
-----  
Statement N;  
Return statement;  
}
```

Return Statement :-

The return statement is used to return from a function. A function may or may not return a value. A return statement returns a value to the calling function and assigns to the variable in the left side of the calling function. If a function does not return a value, the return type in the function definition and declaration is specified as void.

The function can return only one value at a time. The function declared as void may not contain a return statement that specifies a value. Since a void function has no return value it means no return statement with a void function.

The general form of return statement is
Syntax:- `return (expression);` (or) `return return_value;`
Syntax:- `return a;`
`return (a+b);`

1. Returning control from function that does not return value.

`return;`

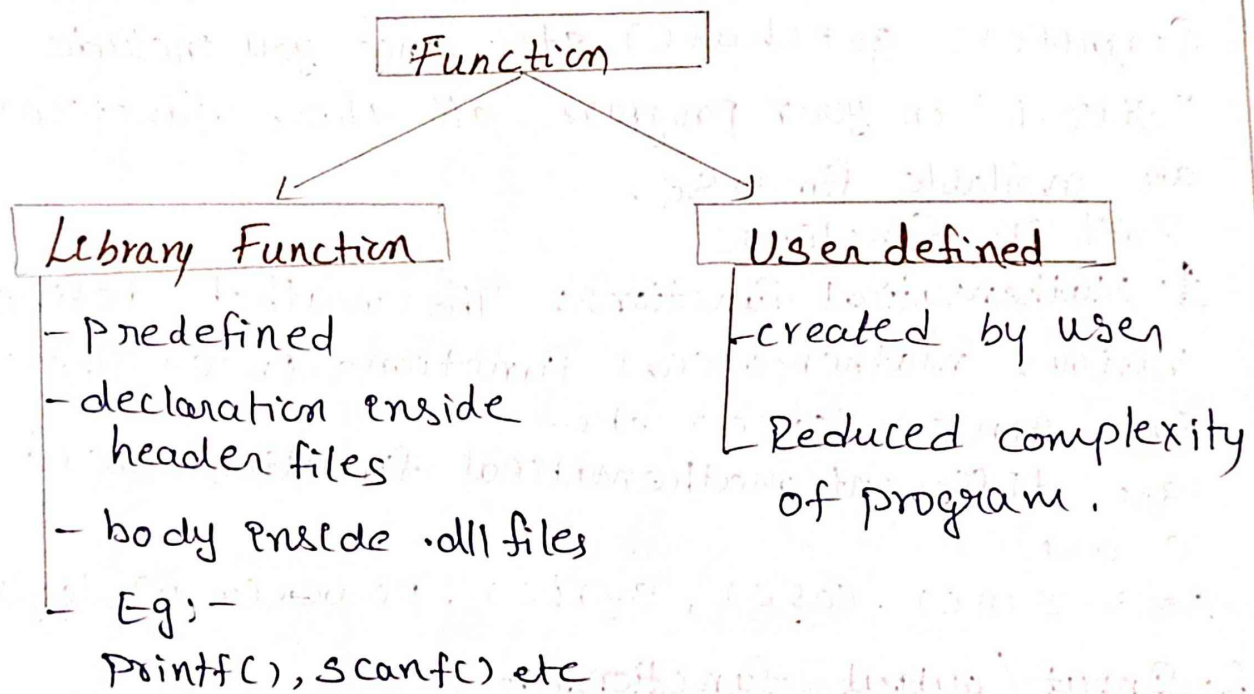
2. Returning control from function that return value:

`return <value>;`

Types of Functions in C Programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming.

1. Built-in function (or) Standard library functions (or) pre-define function.
2. User defined functions.



1. Built-in functions:-

Built in functions in C, often referred to as library functions or standard library functions are predefined functions that are part of the C standard library. These functions provide the standard library functions are built-in functions in C programming to handle tasks such as mathematical calculations, input and output operations, string manipulation, memory allocation etc.

These functions are defined in the header file. When you include the header file, these functions are available for use.

For example:-

The `printf()` is a standard library function to send formatted output to the screen.

The function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as `scanf()`, `fprintf()`, `getchar()` etc. Once you include "stdio.h" in your program, all these functions are available for use.

Built-in functions:

1. **Mathematical functions:** The `math.h` defines various mathematical functions in C. There

Ex:- `sin()`, `cos()` etc.

are different mathematical functions used in C are

Ex:- `sin()`, `cos()`, `sqrt()`, `power(x, y)`, `log(x)`

2. **Input/output functions:-**

These input/output functions are used to read input from the user or write output to the screen.

Ex:- `printf()`, `scanf()`, `getc()`.

3. **String manipulation functions:-**

These string manipulation functions are used to manipulate string in C.

Ex:- `strlen()`, `strcpy()`, `strcat()`.

4. **Time function:** These time functions are used to retrieve the current time or perform time-related calculations.

Ex:- `time()`, `localtime()`, `strftime()`

2. User defined functions:-

These functions are created by programmer according to their requirement. These functions are not predefined or built into the programming language but are written by the programmer to modularize code, improve code organization and make the program more understandable.

Syntax for user defined function in C:-

```
returntype function-name (parameter list)
{
    Statement-1;
    Statement-2;
}
```

For example :- Suppose you want to create a function for add two number then you create a function with name sum() this type of function is called user defined function.

Defining a function: Defining of function is nothing but give body of function that means write logic inside function body.

Syntax:-

```
return-type function-name (parameter)
{
    function body;
}
```

Function Declarations:

A function declaration is the process of tells the compiler about a function name. The actual body of the function can be defined separately.

```
return-type function-name (parameter);
```

Calling a function:

When we call any function control goes to function body and execute entire code. For call any function just write name of function and if any parameter is required then pass parameter.

Syntax:-

function-name ();

Example of UDF function:-

How user-defined function works:-

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

```
void functionName ()
```

```
{
```

```
-----
```

```
}
```

```
int main ()
```

```
{
```

```
-----
```

```
functionName ();
```

```
-----
```

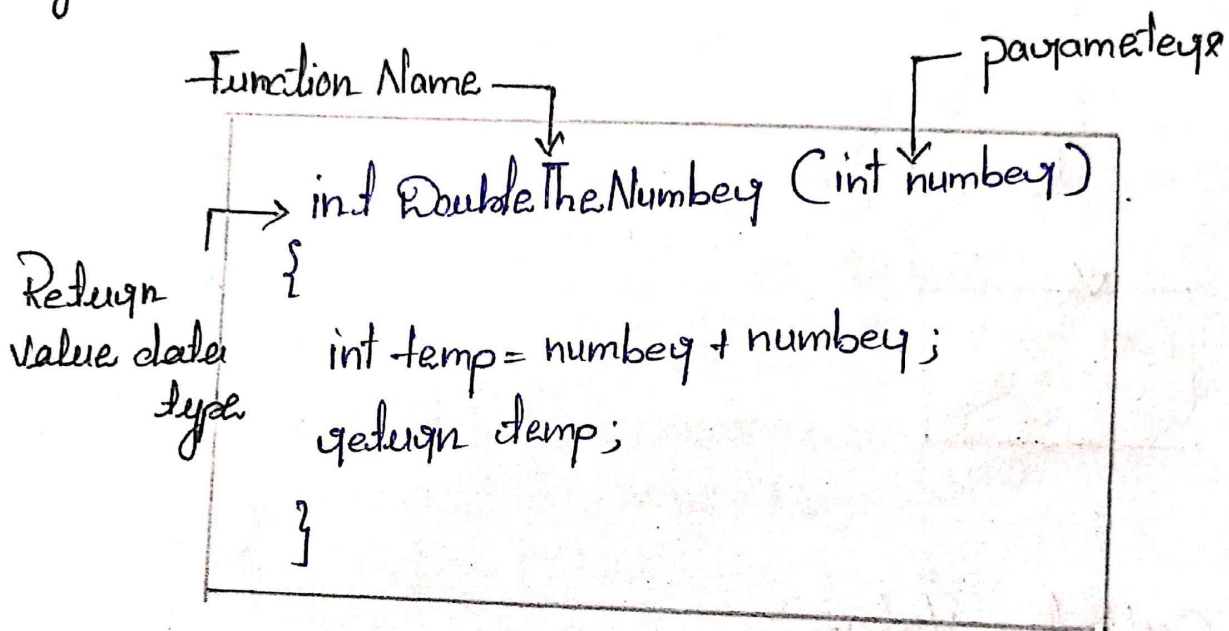
```
}
```

Call by Value vs Call by Reference

Function :-

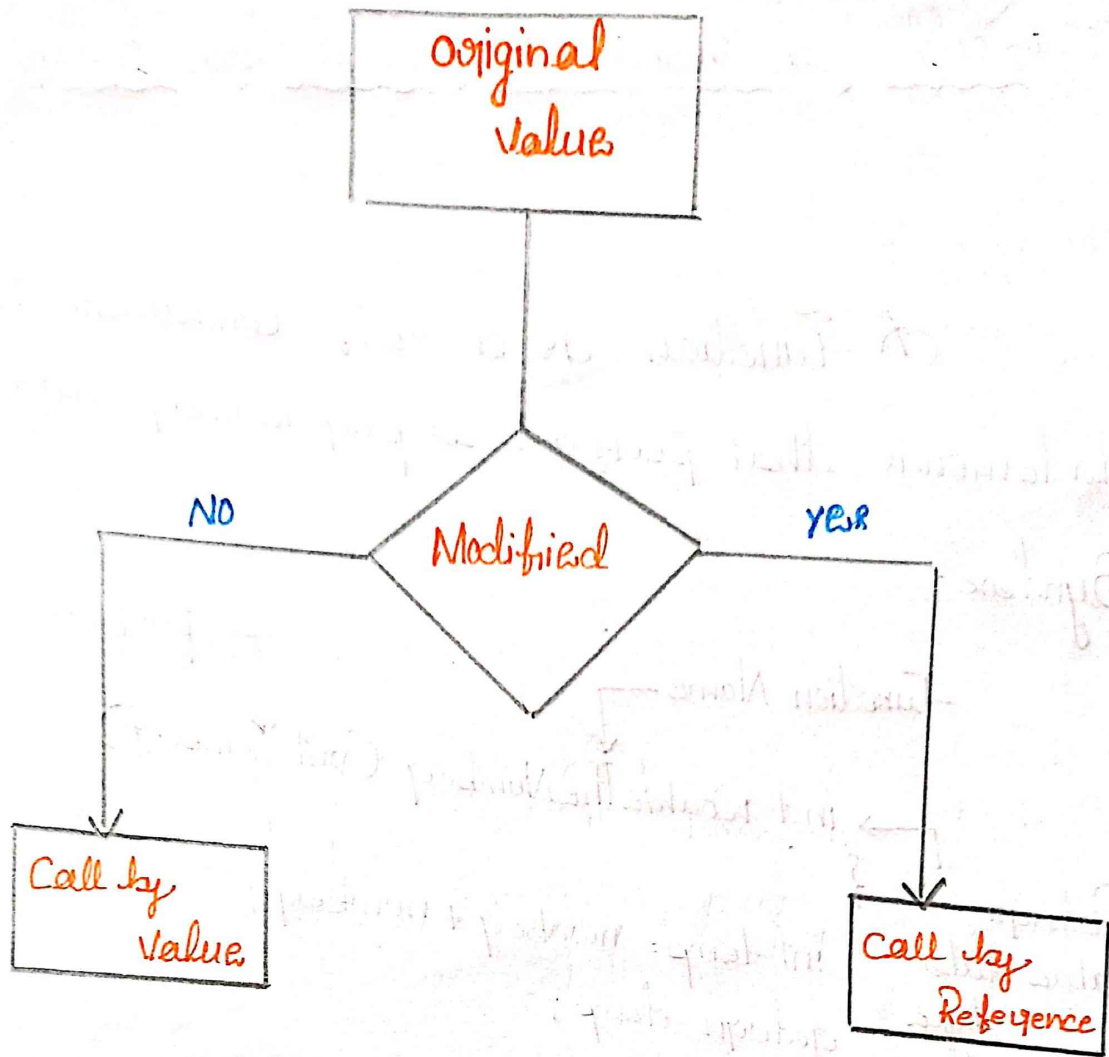
A function is a self-contained block of statements that perform a particular task.

Syntax :-



There are two methods to pass the data into the function in C language i.e.

- * call by value
- * call by Reference



Call by Value :-

- * In call by value method, the value of the actual parameter is copied into the formal parameter. We can say that the value of the variable is used in the function call in the call by value method.
- * In call by value method, we cannot modify the value of the actual parameter by the formal parameter.

* In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

* The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in C with example given below.

```
#include <stdio.h>
#include <conio.h>
void robo (int x)
{
    x = x + 10;
    printf ("x = %d", x);
}
int main()
{
    int a = 10;
    robo (a);
    printf ("a = %d", a);
    getch();
    return 0;
}
```

Output :-

∴ x = 20, a = 10

- * 'qobo (a)' is called, and within the 'qobo' -fun 'x' is modified to 'a+10', which is '20'.
- * In the 'main' function, printf ("x = %d", x); prints the modified value of x.
- * Then printf ("a = %d", a); prints the original value of 'a', which is '10'.

Call by Reference

- * In call by reference, the address of the variable is passed into the function call as the actual parameter.
- * The value of the actual parameter can be modified by changing the formal parameter since the address of the actual parameter is passed.
- * In call by reference, the memory allocation is similar for both formal parameter and actual parameter. all the operations in the function are performed on the value stored at the address of the actual parameter, and the modified value gets stored at the same address.

Consider the following example for the call by Reference.

```
#include <stdio.h>
#include <conio.h>
void swap (int *, int *);
int main ()
{
    int x, y;
    x = 10;
    y = 20;
    printf ("before swap x = %d y = %d", x, y);
    swap (&x, &y);
    printf ("after swap x = %d y = %d", x, y);
    getch ();
    return 0;
}

void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Output :-

Before swap x = 10 y = 20 after swap x = 20 y = 10

* Initial values of 'x' and 'y' are set to 10 and 20 respectively.

* `printf ("before swap x = %d y = %d", x, y);` prints these initial values.

* `Swap (&x, &y);` is called which swaps the values of 'x' and 'y'.

* `printf ("after swap x = %d y = %d", x, y);` prints the values after the swap.

* Before swap $x = 10$ $y = 20$.

After swap $x = 20$ $y = 10$.

Recursion :=

Recursion is the process of calling a function itself repeatedly until a particular condition is met. A function that calls itself directly or indirectly is called a recursive function and such kind of function calls are called recursive calls.

Syn:-

```
return type  function_name (args)
{
  statement - 1;
  ≡
  -
  function_name ( );
}
```

Ex:-

```
void main ( )
{
  printf (" welcome /n ");
  void main ( );
}
```

The above example void main () itself is called main () again and again. The example will produce an output.

o/p:
welcome
welcome
welcome
⋮
⋮
⋮

• Example program:

Factorial of a given number by using recursion.

```
#include <stdio.h>
void main ( )
{
  int i, fact = 1, number;
  clrscr();
  printf ("enter n value:");
```

```
scanf("%d", &n);  
for (i=1; i<=n; i++)  
{  
fact = fact * i;  
}  
printf("factorial of %d is: %d", number, fact);  
getch();  
}
```

o/p :- enter n value : 5
120

advantages :-

- Recursive solution is shorter and simpler than non-recursive.
- code is clearer and easier to use.

disadvantages :-

- for some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack.
- It is difficult to find bugs while using global variables.

Features of Structures

J. Kishwanth
MCA

Structure :- [Definition]

→ A structure is a collection of one or more variables of different datatypes, grouped together under a single name.

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

Features :-

→ To copy elements of one array to another array of same datatype elements are copied one by one. It is not possible to copy all the elements at a time. Whereas in structure it is possible to copy the contents of all structure elements of different datatypes to another structure variable of its type using assignment (=) operator.

→ Nesting of structures is possible i.e., one can create structure within the structure. Using this feature one can handle complex data types.

→ It is also possible (to) to pass structure elements to a function.

→ one can pass individual structure elements or entire structure by value or address.

→ It is also possible to create structure pointers.

→ we can create a pointer pointing to structure elements. For this it requires ['→' operator] <->.

Declaration and initialization of Structure:-

Structure can be declared as given below:

```
struct struct_type  
{  
    type variable1;  
    type variable2;  
};
```

Structure declaration always starts with structure keyword. Here struct-type is known as tag. The struct declaration is enclosed with in a pair of curly braces. using struct and tag user can declare structure variables like variable1, variable2 and so on. These are the members of the structure. After defining structure we can create variables as given below:

```
struct struct_type v1, v2, v3.
```

Here v1, v2 and v3 are variables of structure struct-type. This is similar to declaring variable of any data type.

```
int v1, v2, v3
```

Here v1, v2, v3 are variables of integer data type.

The declaration defines the structure but this process doesn't allocate memory. The memory allocation takes

place only when variables are declared.

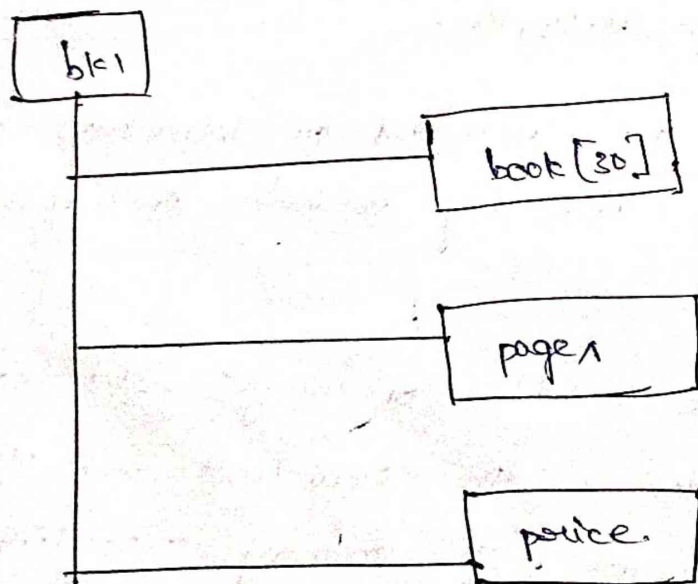
```
struct book1  
{  
    char book[30];  
    int pages;  
    float price;  
};
```

```
struct book1 bk1;
```

In the above example a structure of type book1 is created. It consists of three members book[30] of char data type, pages of int type and price of float data type. Explain various members of a structure.

```
struct book1 bk1;
```

The above line creates variable bk1 of type book1 and it reserves total 36 bytes (30 bytes for book[30], 2 bytes for integer and 4 bytes for float) through bk1 all



the three members of structure can be accessed.
In order to initialize structure elements with certain values following statement is used

```
struct book b1 = ("Srinivasa", 500, 385.00);
```

All the members of structure are related to variable b1:

struct - variable . member or b1 . book

The period (.) sign is used to access the structure members we can directly assign values to members as given below

```
b1.book = "Srinivasa";
```

```
b1.pages = 500;
```

```
b1.price = 385.00;
```

Example for working of structure! -

Ex! - (write a program to display size of structure elements)
(by using () of operator)

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

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

```
main()
```

```
{
```

```
struct book
```

```
{
```

```
char book[20];
```

```
int pages;
```

```
float price;
```

```
};
```

```
struct book1 bkl;
```

```
clrscr();
```

```
printf("\n size of structure element ");
```

```
printf("\n Book : %d", sizeof (bkl.book));
```

```
printf("\n pages : %d", sizeof (bkl.pages));
```

```
printf("\n price : %d", sizeof (bkl.price));
```

```
printf("\n Total Bytes : %d", sizeof (bkl));
```

```
}
```

Output: ←

size of structure element

Book : 30

pages : 2

price : 4

Total Bytes : 36

Explanation: ←

In the above program structure book1 is defined with three member variables char book[30], int pages & float price respectively. The bkl is an object of the structure book1. Using 'sizeof()' operator their sizes are displayed. The sizes are 30, 2 and 4 respectively. The total size of one record is 36, i.e. size of all member variables of structure.

Ex: 2: Write a program to define a structure and initialize its member variables.

```
main()
{
    struct book1
    {
        char book[30];
        int pages;
        float price;
    };
    struct book1 bkl = {"c++", 300, 285.00};
    clrscr();
    printf("In Book name : %s", bkl.book);
    printf("In no. of pages : %d", bkl.pages);
    printf("In Book price : %f", bkl.price);
}
```

Output:-

Book name: c++

No. of pages: 300

Book price: 285.00

Explanation:-

In the above program the structure book1 is defined with its member variables char book[30], int pages and float price. The bkl is an object of the structure book1. The statement struct book1 bkl = {"c++", 300, 285.00} define object bkl and initialize

the variables with the values enclosed in the curly braces respectively. using printf() statement contents of individual fields are displayed.

Structure and functions

Introduction :-

A function is a subprogram segment that carries out some specific well defined tasks. every C program consists of one or more functions and all of those functions must be called as main function.

Derived data types are formed from fundamental data type. structures are one such user-defined data type. The structures can have many fundamental data types known as structure members grouped into a single user-defined data type.

structure-function can be effectively used while writing code, structure can be passed by arguments to the functions, this can be done in three ways.

They are

- passing the members of the structures as an argument
- passing the entire structure as an argument.
- passing the address of the structure as argument

syntax :-

```
void myfunction ( )  
{  
    _____  
}
```

To pass structure members to functions :-

sometimes we don't want to pass the entire structure to the function. we want to pass only a few members of the structure. we can use the dot (.) operator to access the individual

member of the structure and pass them to the function.

syntax :-

```
struct student
{
    int roll no;
    char name [10];
    float marks;
}
```

} member of functions

Return structure from a function :-

Return a variable from a function such as return, return a we can also return multiple variables in the form of a single structure variable.

A structure can be returned from a function using the return keyword. structures can be passed into functions either by reference or by value.

syntax :-

Return-type functionname (datatype paramname....);

Pass structure by reference :-

Passing the parameter as a value will make a copy of the structure variable, passing it to the function. Imagine we have a structure with a huge no. of structure members. making a copy of all the members and passing it to the function a lot of time and consumes a lot of memory.

syntax :-

```
struct student  
{  
  ---  
  ---  
}
```

Array structure as function :-

An array is a collection of similar data types we know that even a structure is a data type from previous knowledge.

- structure - function can be used to write code effectively.
- The structure can be passed as a parameter to the function.
- An entire structure can be passed to a function or individual members of the structure can be passed into the function.
- A structure can be returned from a function using the return keyword.
- Structure can be passed into functions either by reference or by value.
- An array of structure can also be passed to a function.

Example :-

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

```
}  
char name[50];  
int age;  
};
```

```
void display (struct student s)
```

```
int main ()
```

```
{
```

```
struct student s;
```

```
printf ("Enter name ");
```

```
scanf ("%c", &s, name);
```

```
printf ("Enter age ");
```

```
scanf ("%d", &s, age);
```

```
display (s);
```

```
return 0;
```

```
}
```

```
void display (struct students)
```

```
{
```

```
printf ("displaying info");
```

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

```
printf ("age %d", s.age);
```

```
}
```


Array of Structures

Structure is a collection of variables of different data types, and an array is a collection of variables of the same data type. Combining these concepts gives us a Array of Structures, which allows for a collection of structures.

Declaration of array of structure

```
struct student st[5];
```

Accessing Structure Fields in Array

We can access individual members of a structure variable

Syntax

```
array_name[index].member_name
```

Example

```
#include<stdio.h>
#include <string.h>
/* Declaration of structure */
struct student
{
    int rollno;
    char name[10];
};
int main()
{
    int i;
    /* Declaration of array of structure */
    struct student st[5];
    printf("Enter Records of 5 students");
    for(i=0;i<5;i++)
    {
        printf("\nEnter Rollno:");
        scanf("%d",&st[i].rollno);
        printf("\nEnter Name:");
        scanf("%s",&st[i].name);
    }
    printf("\nStudent Information List:");
    for(i=0;i<5;i++)
    {
        printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
    }
}
```

```
    return 0;  
}
```

Output

Enter Records of 5 students

Enter Rollno:1

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

Data structures

Introduction

- *A data structure is a storage that is used to store and organize data.*
- A data structure is not only used for organizing the data. It is also used for processing and retrieving data.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure

- The arrangement of data in a sequential manner is known as a linear data structure.
- The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues.
- In these data structures, one element is connected to only one another element in a linear form.

Non-Linear Data Structure

- When one element is connected to the 'n' number of elements known as a non-linear data structure.
- The best example is trees and graphs. In this case, the elements are arranged in a random manner.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structures provide reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

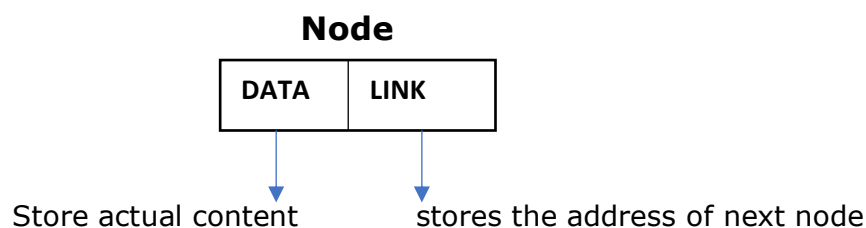
Linked list

Linked list is an ordered collection of homogeneous data elements called nodes where the linear order is maintained by means of links (or) pointers

Linked list is a data structure which is collection of nodes that contains of two parts

1. Data part
2. Link to next node

Structure of Node: -



Types of Linked list

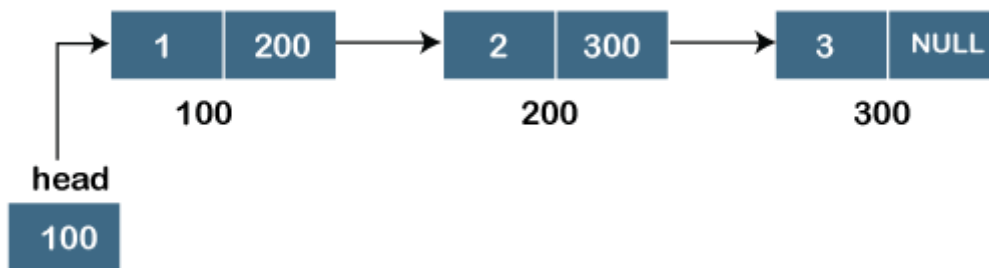
The following are the types of linked list:

- [Single Linked list](#)
- [Doubly Linked list](#)
- [Circular Linked list](#)
- [Doubly Circular Linked list](#)

Single Linked list

The single linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a **head pointer**.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

1. struct node
2. {
3. **int** data;
4. struct node *next;
5. }

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

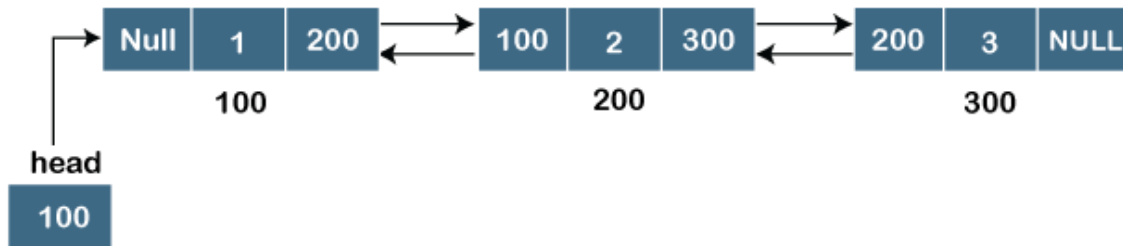
As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part.

UNIT-3

(or)

In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the **address of the next** while the other part of the node stores the **previous node's address**. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

1. struct node
2. {
3. **int** data;
4. struct node *next;
5. struct node *prev;
6. }

In the above representation, we have defined a user-defined structure named **a node** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the **struct node**

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the **singly linked list** and a **circular linked** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular

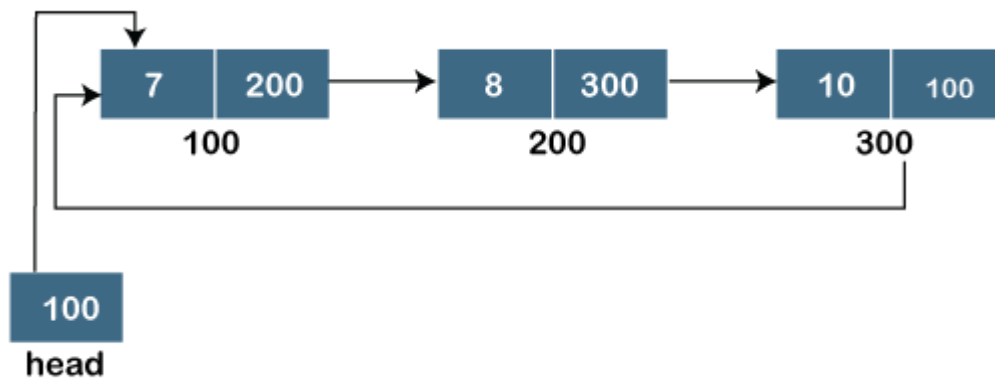
UNIT-3

linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

Representation of the node in Circular linked list

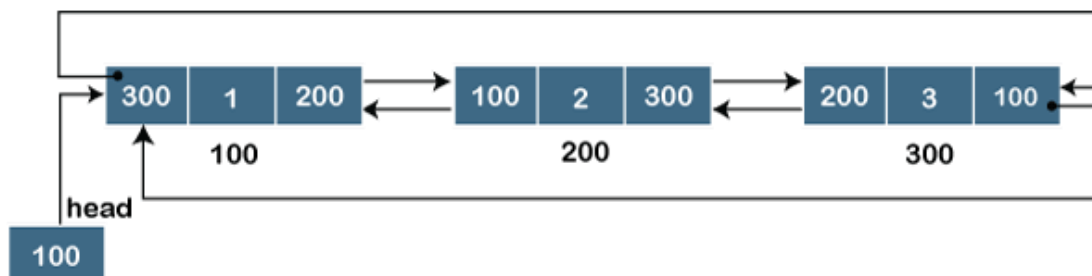
1. struct node
2. {
3. **int** data;
4. struct node *next;
5. }

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



Doubly Circular linked list

The doubly circular linked list has the features of both the **circular linked list** and **doubly linked list**.



The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The

UNIT-3

main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

Representation of the node in Doubly Circular linked list

1. struct node
2. {
3. **int** data;
4. struct node *next;
5. struct node *prev;
6. }

What is a Stack?

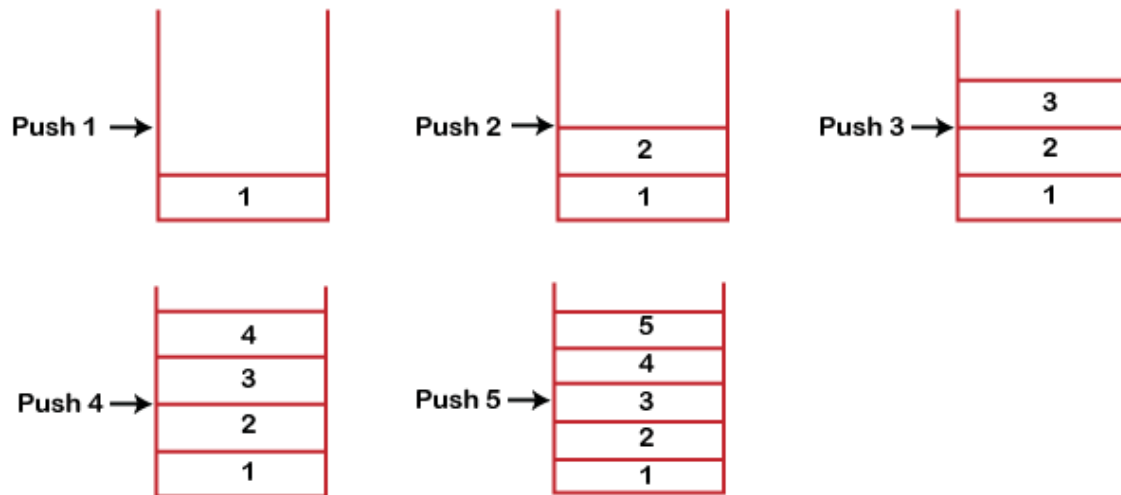
A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, **a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.**

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.

UNIT-3



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

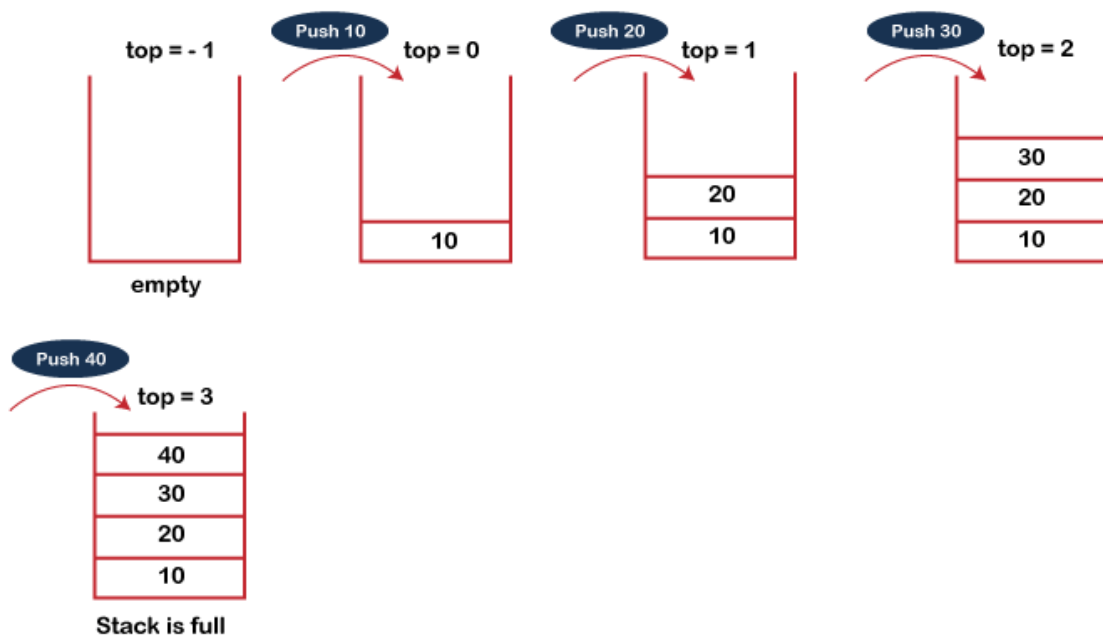
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the **underflow** condition occurs.

UNIT-3

- If the stack is not empty, we first access the element which is pointed by the **top**
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflowed when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

1. begin
2. **if** top = n then stack full
3. top = top + 1
4. stack (top) := item;
5. end

Time Complexity : $O(1)$

UNIT-3

implementation of push algorithm in C language

```
1. void push (int val,int n) //n is size of the stack
2. {
3.     if (top == n )
4.         printf("\n Overflow");
5.     else
6.     {
7.         top = top +1;
8.         stack[top] = val;
9.     }
10.}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
1. begin
2.     if top = 0 then stack empty;
3.     item := stack(top);
4.     top = top - 1;
5. end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
1. int pop ()
2. {
3.     if(top == -1)
```

UNIT-3

```
4.  {
5.    printf("Underflow");
6.    return 0;
7.  }
8.  else
9.  {
10.   return stack[top - - ];
11.}
➤ }
```

Here, We have implemented stacks using [arrays in C](#).

```
#include<stdio.h>
#include<stdlib.h>
#define Size 4
int Top=-1, inp_array[Size];
void Push();
void Pop();
void show();
int main()
{
    int choice;

    while(1)
    {
        printf("\nOperations performed by Stack");
        printf("\n1.Push the element\n2.Pop the
element\n3.Show\n4.End");
        printf("\n\nEnter the choice:");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: Push();
                    break;
            case 2: Pop();
                    break;
            case 3: show();
                    break;
            case 4: exit(0);

            default: printf("\nInvalid choice!!!");
        }
    }
}
```

UNIT-3

```
}

void Push ()
{
    int x;

    if (Top==Size-1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter element to be inserted to the stack:");
        scanf("%d",&x);
        Top=Top+1;
        inp_array[Top]=x;
    }
}

void Pop ()
{
    if (Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element:  %d",inp_array[Top]);
        Top=Top-1;
    }
}

void show ()
{
    if (Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for(int i=Top;i>=0;--i)
            printf("%d\n",inp_array[i]);
    }
}
}
```

Output:

Operations performed by Stack

1.Push the element

2.Pop the element

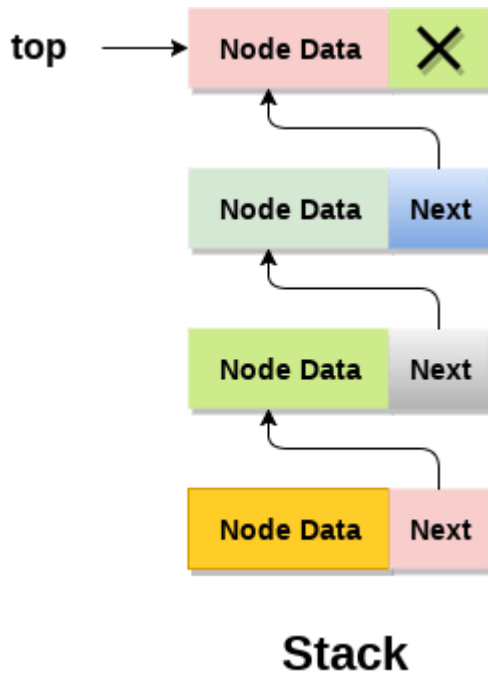
UNIT-3

```
3.Show
4.End
Enter the choice:1
Enter element to be inserted to the stack:10
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End
Enter the choice:3
Elements present in the stack:
10
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End
Enter the choice:2
Popped element: 10
Operations performed by Stack
1.Push the element
2.Pop the element
3.Show
4.End
Enter the choice:3
Underflow!!
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflowed if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Pause

Unmute

Loaded: 51.61%

Fullscreen

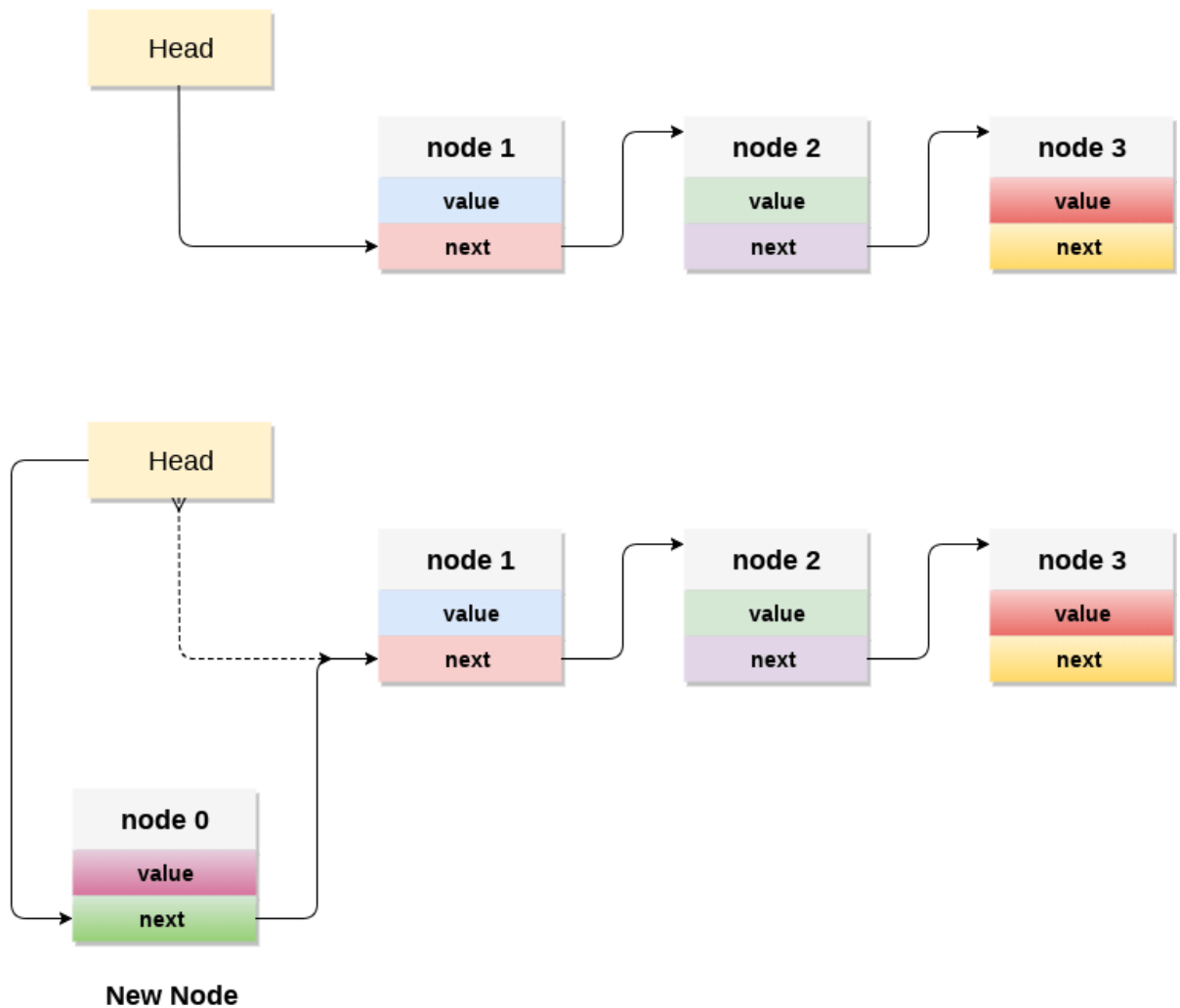
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

UNIT-3

Time Complexity : $O(1)$



C implementation :

1. **void** push ()
2. {
3. **int** val;
4. struct node *ptr =(struct node*)malloc(sizeof(struct node));
5. **if**(ptr == NULL)
6. {
7. printf("not able to push the element");
8. }
9. **else**
10. {
11. printf("Enter the value");
12. scanf("%d",&val);

UNIT-3

```
13.     if(head==NULL)
14.     {
15.         ptr->val = val;
16.         ptr -> next = NULL;
17.         head=ptr;
18.     }
19.     else
20.     {
21.         ptr->val = val;
22.         ptr->next = head;
23.         head=ptr;
24.
25.     }
26.     printf("Item pushed");
27.
28. }
29. }
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

30. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
31. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : $O(n)$

C implementation

```
32. void pop()
33. {
```

UNIT-3

```
34. int item;
35. struct node *ptr;
36. if (head == NULL)
37. {
38.     printf("Underflow");
39. }
40. else
41. {
42.     item = head->val;
43.     ptr = head;
44.     head = head->next;
45.     free(ptr);
46.     printf("Item popped");
47.
48. }
49. }
```

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.

Queue underflow (isempty): When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow

Implementation of Queue in C

Queues in C can be implemented using Arrays, Lists, Structures, etc. Below here we have implemented queues using [Arrays in C](#).

Example:

UNIT-3

```
#include <stdio.h>
# define SIZE 100
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
main()
{
    int ch;
    while (1)
    {
        printf("1.Enqueue Operation\n");
        printf("2.Dequeue Operation\n");
        printf("3.Display the Queue\n");
        printf("4.Exit\n");
        printf("Enter your choice of operations : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
            case 3:
                show();
                break;
            case 4:
                exit(0);
            default:
                printf("Incorrect choice \n");
        }
    }
}

void enqueue()
{
    int x;
    if (Rear == SIZE - 1)
        printf("Overflow \n");
```

UNIT-3

```
else
{
    if (Front == - 1)
        Front = 0;

    printf("Element to be inserted in the Queue\n : ");
    scanf("%d", &x);
    Rear = Rear + 1;
    inp_arr[Rear] = x;
}
}

void dequeue()
{
    if (Front == - 1 || Front > Rear) //Rear===-1
    {
        printf("Queue is empty \n");
        return ;
    }
    else
    {
        printf("Element deleted from the Queue: %d\n", inp_arr[Front]);
        Front = Front + 1;
    }
}

void show()
{
    if (Front == - 1)
        printf("Empty Queue \n");
    else
    {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
            printf("%d ", inp_arr[i]);
        printf("\n");
    }
}
```

Output:

- 1.Enqueue Operation
- 2.Dequeue Operation
- 3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue: 10

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue: 20

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 3

Queue:

10 20

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 2

Element deleted from the Queue: 10

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations: 3

Queue:

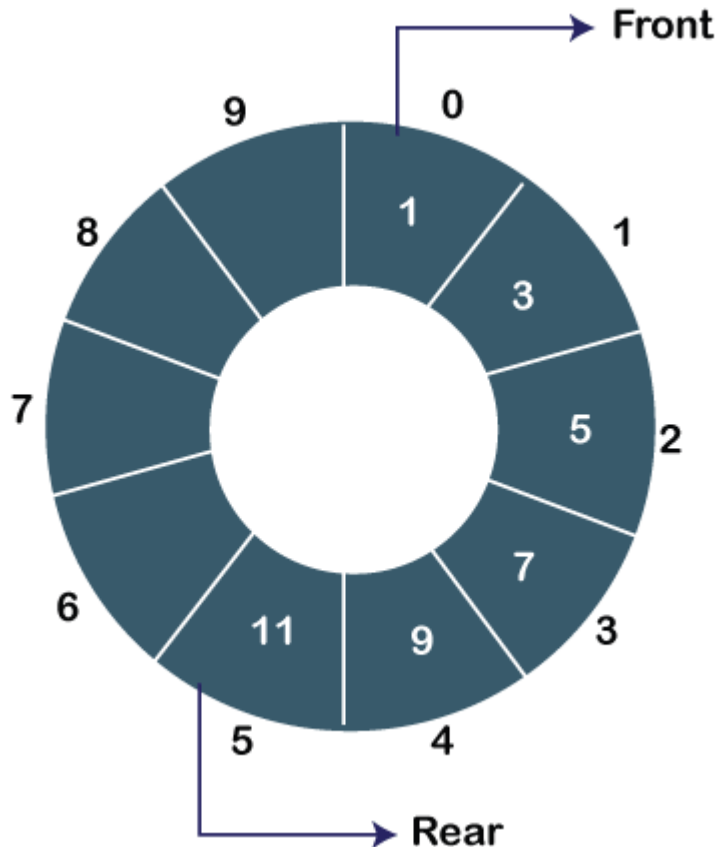
20

Types of Queue

- **Circular Queue**

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:

UNIT-3



Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enqueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end

-
- **Algorithm to insert an element in a circular queue**
-

- **Step 1:** IF $(REAR+1)\%MAX = FRONT$
Write " OVERFLOW "
Goto step 4
[End OF IF]
- **Step 2:** IF $FRONT = -1$ and $REAR = -1$
SET $FRONT = REAR = 0$
ELSE IF $REAR = MAX - 1$ and $FRONT \neq 0$

UNIT-3

SET REAR = 0

ELSE

SET REAR = (REAR + 1) % MAX

[END OF IF]

- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

- **Priority Queue**

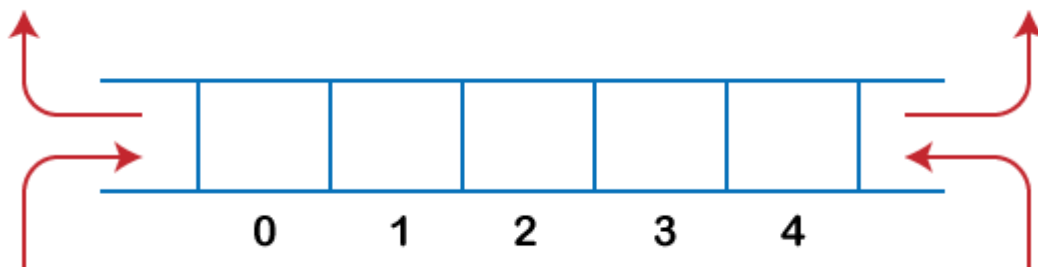
A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure

Deque

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.



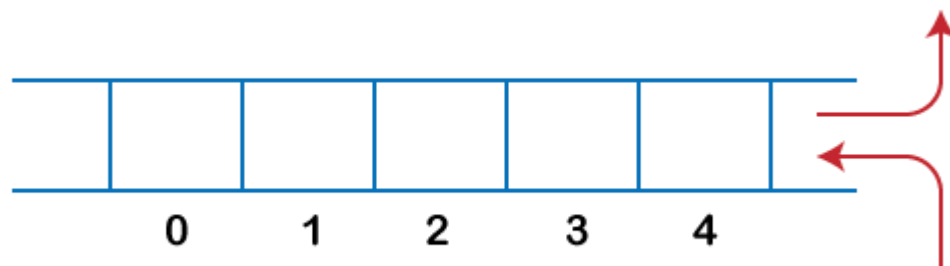
Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Let's look at some properties of deque.

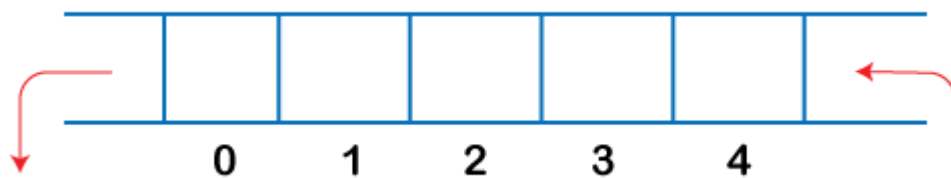
UNIT-3

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.



In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

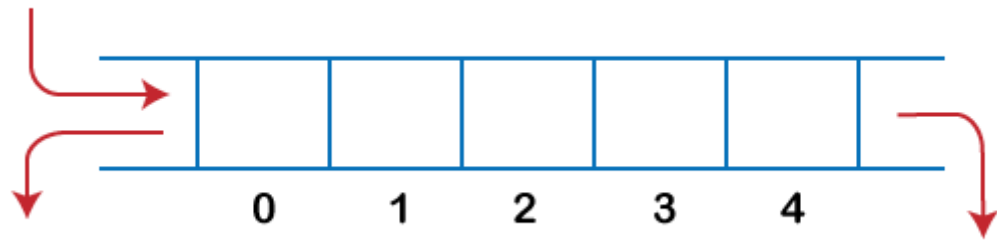


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

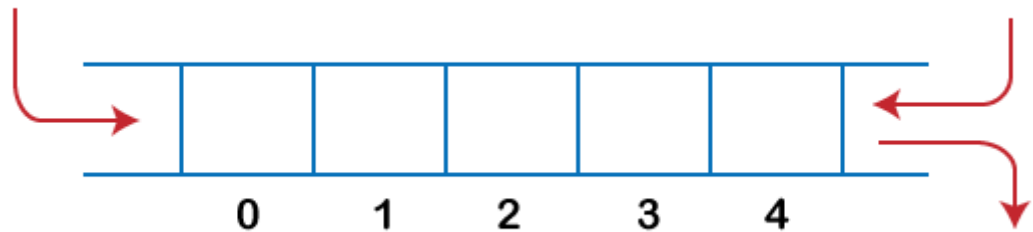
1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the

UNIT-3

ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

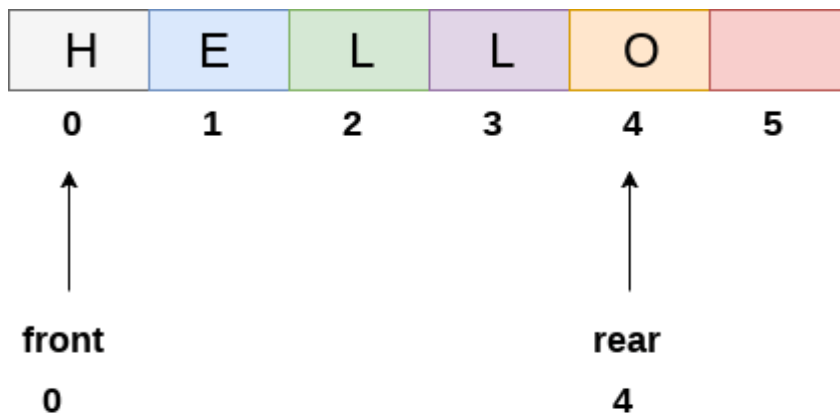
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.

UNIT-3

- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Array representation of Queue

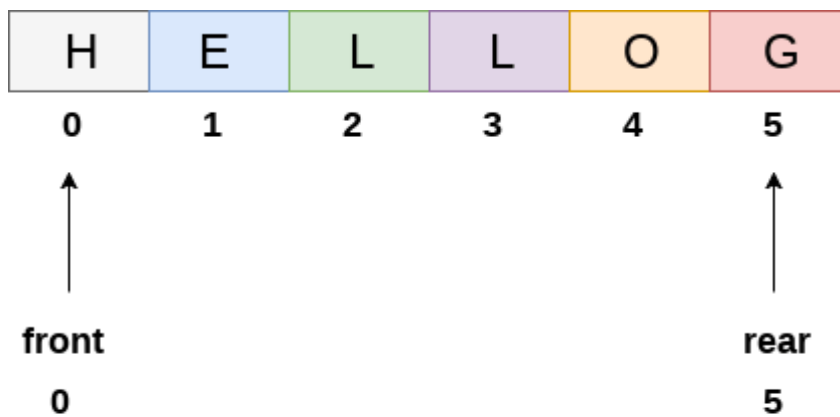
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue

The above figure shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.

UNIT-3



Queue after inserting an element

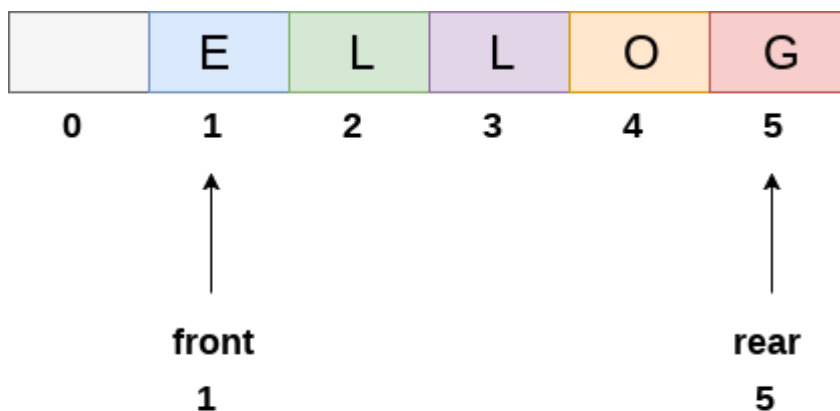
After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.

Pause

Unmute

Loaded: 9.56%

Fullscreen



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to $\text{max} - 1$. if so, then return an overflow error.

UNIT-3

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

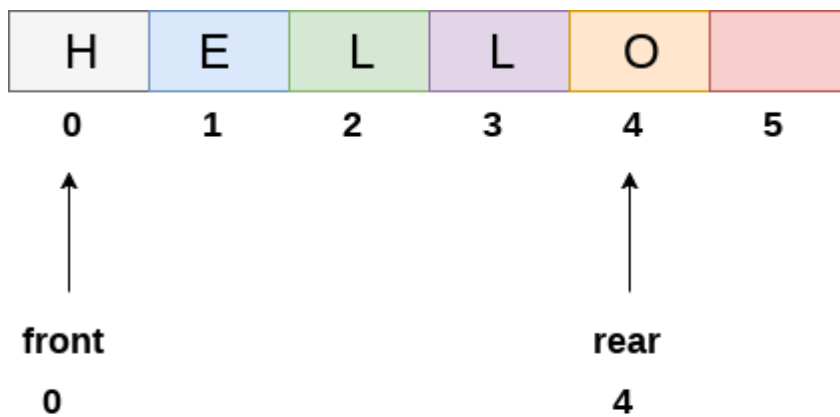
- **Step 1:** IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** exit

[Next →](#) [← Prev](#)

Array representation of Queue

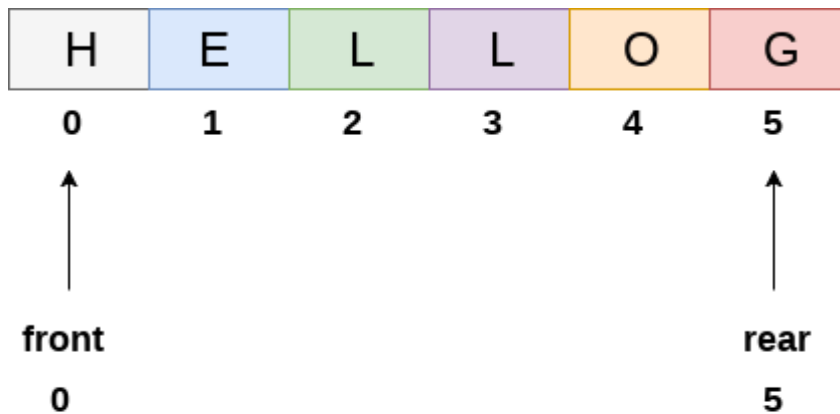
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.

UNIT-3



Queue

The above figure shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

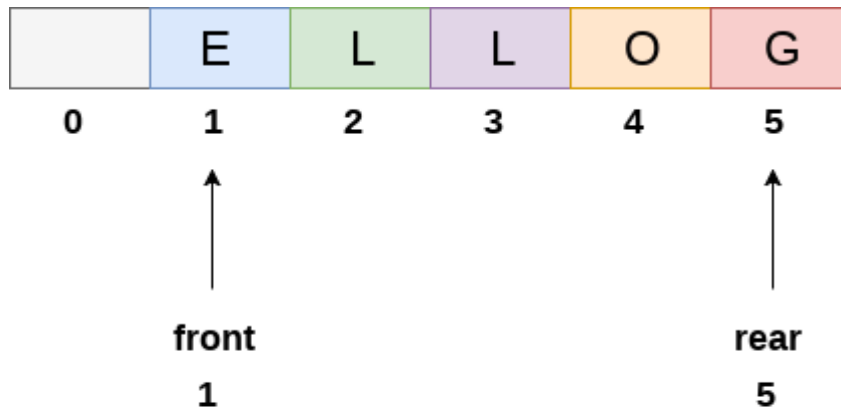
After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.

Pause

Unmute

Loaded: 25.77%

Fullscreen



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM

- **Step 4:** EXIT

C Function

```
1. void insert (int queue[], int max, int front, int rear, int item)
2. {
3.     if (rear + 1 == max)
4.     {
5.         printf("overflow");
6.     }
7.     else
8.     {
9.         if(front == -1 && rear == -1)
10.        {
11.            front = 0;
12.            rear = 0;
13.        }
14.        else
15.        {
16.            rear = rear + 1;
17.        }
18.        queue[rear]=item;
19.    }
20.}
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]

UNIT-3

SET FRONT = FRONT + 1

[END OF IF]

- **Step 2:** EXIT

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

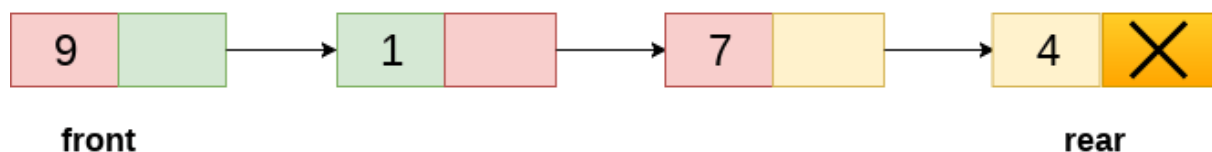
The storage requirement of linked representation of a queue with n elements is $o(n)$ while the time requirement for operations is $o(1)$.

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

UNIT-3

Firstly, allocate the memory for the new node ptr by using the following statement.

1. `Ptr = (struct node *) malloc (sizeof(struct node));`

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

1. `ptr -> data = item;`
2. `if(front == NULL)`
3. `{`
4. `front = ptr;`
5. `rear = ptr;`
6. `front -> next = NULL;`
7. `rear -> next = NULL;`
8. `}`

In the second case, the queue contains more than one element. The condition `front = NULL` becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

1. `rear -> next = ptr;`
2. `rear = ptr;`
3. `rear->next = NULL;`

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR

SET REAR -> NEXT = NULL

[END OF IF]

- **Step 4:** END

C Function

```
1. void insert(struct node *ptr, int item; )
2. {
3.
4.
5.     ptr = (struct node *) malloc (sizeof(struct node));
6.     if(ptr == NULL)
7.     {
8.         printf("\nOVERFLOW\n");
9.         return;
10.    }
11.    else
12.    {
13.        ptr -> data = item;
14.        if(front == NULL)
15.        {
16.            front = ptr;
17.            rear = ptr;
18.            front -> next = NULL;
19.            rear -> next = NULL;
20.        }
21.        else
22.        {
23.            rear -> next = ptr;
24.            rear = ptr;
25.            rear->next = NULL;
26.        }
27.    }
28.}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition

UNIT-3

front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

1. ptr = front;
2. front = front -> next;
3. free(ptr);

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

Unit 5

Trees

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

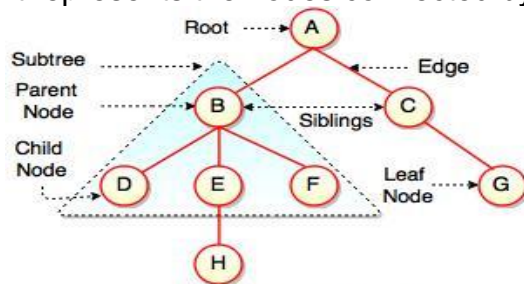


Fig. Structure of Tree

The above figure represents structure of a tree. Tree has 2 subtrees.

A is a parent of B and C.

B is called a child of A and also parent of D, E, F.

Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.

Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

In the above figure, D, F, H, G are **leaves**. B and C are **siblings**. Each node excluding a root is connected by a direct edge from exactly one other node parent → children.

- **Levels of a node**
- Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:

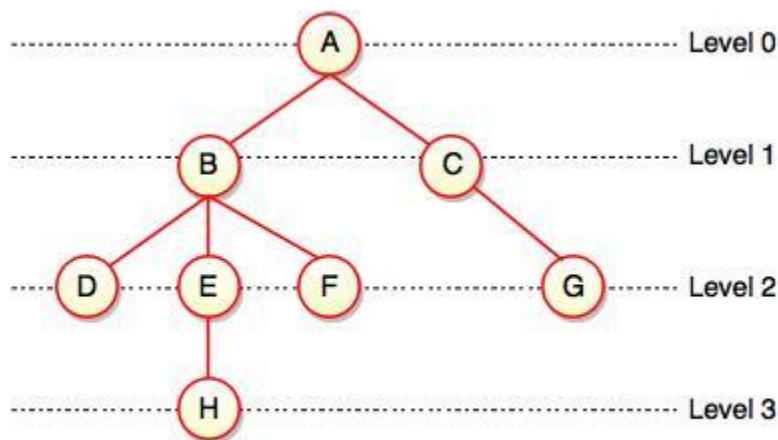
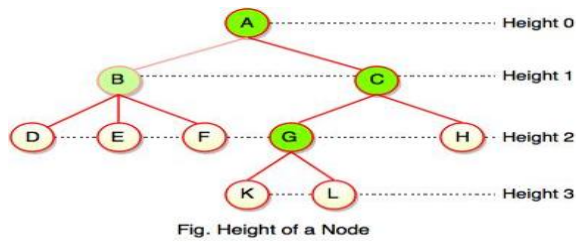


Fig. Levels of Tree

Note:

- If node has no children, it is called **Leaves** or **External Nodes**.
- Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.
- A tree can be empty with no nodes or a tree consists of one node called the **Root**.

- **Height of a Node**



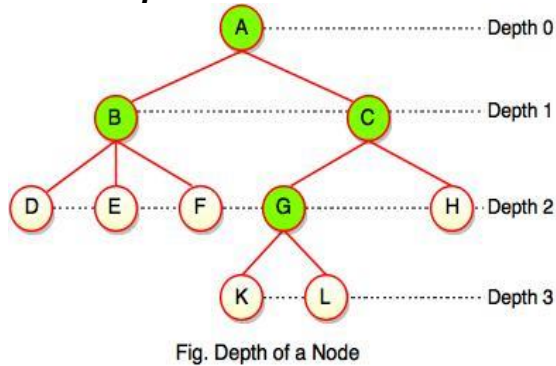
As we studied, height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.

In the above figure, A, B, C, D can have height. Leaf cannot have height as there will be no path starting from a leaf. Node A's height is the number of edges of the path to K not to D. And its height is 3.

Note:

- Height of a node defines the longest path from the node to a leaf.
- Path can only be downward.

• **Depth of a Node**



While talking about the height, it locates a node at bottom where for depth, it is located at top which is root level and therefore we call it depth of a node.

In the above figure, Node G's depth is 2. In depth of a node, we just count how many edges between the targeting node & the root and ignoring the directions.

Note: Depth of the root is 0.

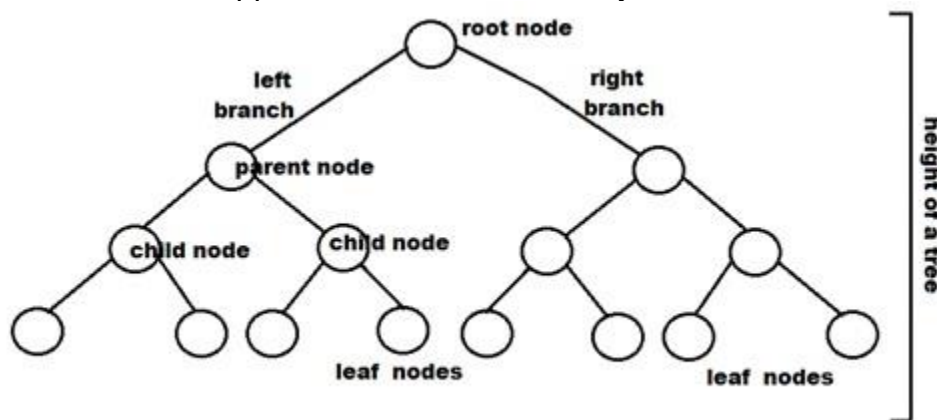
Advantages of Tree

- Tree reflects structural relationships in the data.
- It is used to represent hierarchies.
- It provides an efficient insertion and searching operations.
- Trees are flexible. It allows to move subtrees around with minimum effort.

Binary trees

A **binary tree** is a tree-type non-linear [data structure](#) with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.



Terminologies associated with Binary Trees

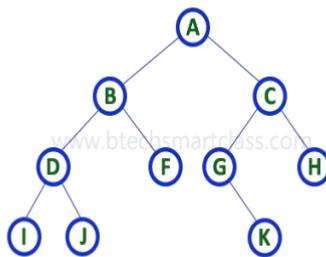
- **Node:** It represents a termination point in a tree.
- **Root:** A tree's topmost node.
- **Parent:** Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child:** A node that straightway came from a parent node when moving away from the root is the child node.
- **Leaf Node:** These are external nodes. They are the nodes that have no child.
- **Internal Node:** As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree:** The number of edges from the tree's node to the root is.
- **Height of a Tree:** It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...



To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2n + 1$.

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

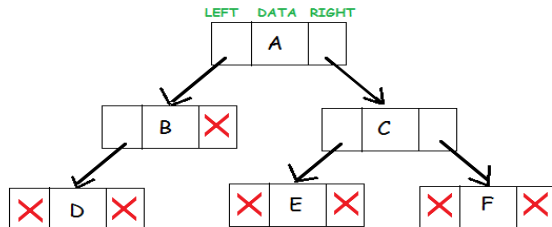


The above example of the binary tree represented using Linked list representation is shown as follows...

Binary Tree Components

There are three **binary tree components**. Every **binary tree** node has these three components associated with it. It becomes an essential concept for programmers to understand these three **binary tree components**:

1. Data element
2. Pointer to left subtree
3. Pointer to right subtree



Source

These three **binary tree components** represent a node. The data resides in the middle. The left pointer points to the child node, forming the left sub-tree. The right pointer points to the child node at its right, creating the right subtree.

Types of Binary Trees

There are various **types of binary trees**, and each of these **binary tree types** has unique characteristics. Here are each of the **binary tree types** in detail:

1. Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.
- Full binary tree is also called as **Strictly Binary Tree**.

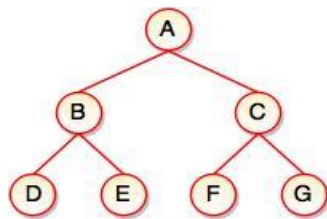


Fig. Full Binary Tree

- Every node in the tree has either 0 or 2 children.
- Full binary tree is used to represent mathematical expressions.

2. Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.
- Complete binary tree is also called as **Perfect Binary Tree**.

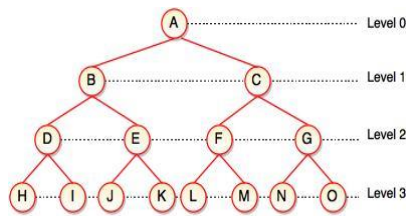


Fig. Complete Binary Tree

- In a complete binary tree, every internal node has exactly two children and all leaf nodes are at same level.
- For example, at Level 2, there must be $2^2 = 4$ nodes and at Level 3 there must be $2^3 = 8$ nodes.

3. Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.



Fig. Left Skewed Binary Tree

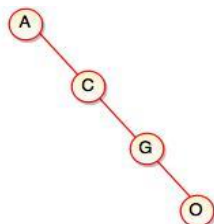


Fig. Right Skewed Binary Tree

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

4. Extended Binary Tree

- Extended binary tree consists of replacing every null subtree of the original tree with special nodes.
- Empty circle represents internal node and filled circle represents external node.
- The nodes from the original tree are internal nodes and the special nodes are external nodes.
- Every internal node in the extended binary tree has exactly two children and every external node is a leaf. It displays the result which is a **complete binary tree**.

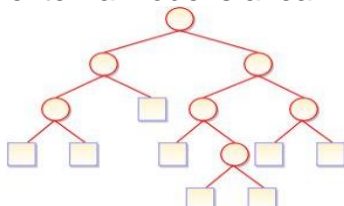
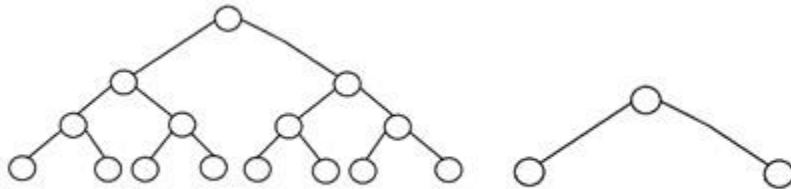


Fig. Extended Binary Tree

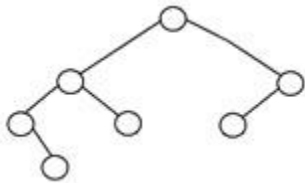
4. Perfect Binary Tree

A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect [binary tree](#) having height 'h' has $2^h - 1$ nodes. Here is the structure of a perfect binary tree:



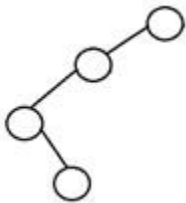
5. Balanced Binary Tree

A binary tree is said to be 'balanced' if the tree height is $O(\log N)$, where 'N' is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:



6. Degenerate Binary Tree

A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:



Benefits of a Binary Tree

- The search operation in a binary tree is faster as compared to other trees
- Only two traversals are enough to provide the elements in sorted order
- It is easy to pick up the maximum and minimum elements
- Graph traversal also uses binary trees
- Converting different postfix and prefix expressions are possible using binary trees

Differences between General Tree and Binary Tree

General Tree

- General tree has no limit of number of children.
- Evaluating any expression is hard in general trees.

Binary Tree

- A binary tree has maximum two children
- Evaluation of expression is simple in binary tree.

Application of trees

- Manipulation of arithmetic expression
- Construction of symbol table
- Analysis of Syntax
- Writing Grammar
- Creation of Expression Tree

Binary Tree Traversals

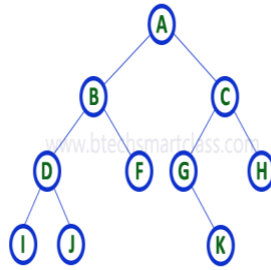
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between the left child and right child. In this traversal,

- the left child node is visited first, then
- the root node is visited and later
- we go for visiting the right child node.

This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

Algorithm

Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal,

- the root node is visited first, then
- its left child and
- later its right child.

This pre-order traversal is applicable for every root node of all subtrees in the tree. In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

Algorithm

Until all nodes are traversed -

- Step 1** - Visit root node.
- Step 2** - Recursively traverse left subtree.
- Step 3** - Recursively traverse right subtree.

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal,

- left child node is visited first, then
- its right child and then
- its root node.

This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I-J-D-F-B-K-G-H-C-A

Algorithm

Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Recursively traverse right subtree.

Step 3 - Visit root node.