UNIT-1

C-Programming Fundamentals

Data types - variables - Operations - Expressions and Statements - Conditional Statements - Functions, Recursive functions, Arrays, Single and Multi-Dimensional Arrays

1. Datatypes

The Basic Datatypes in C are int, char, float and Double. The Quantifiers Short, long, Signed and Unsigned Apply to Integer whereas long apply to Double

Integer Datatypes

Datatype	Description	Required Memory	min value	max value
int	Represent Integers: signed / unsigned	2 Bytes	-32,768 / 0	32767 / 65,535
int	Represent Integers: Signed / Unsigned	4 Bytes	-2,147,483,648 / 0	2,147,483,647 / 4,294,967,293
short int	Represent Integers: Signed / Unsigned	2 Bytes	-32,768 / 0	32767 / 65,535
long int	Represent Integers: Signed / Unsigned	4 Byte	-2,147,483,648 / 0	2,147,483,647 / 4,294,967,215

* Some compilers Use 48, 64 (or) more bits

Datatype	Description	Required memory	min value	max Val
Char	Signed single character	1 Bytes	-128	127
char	Unsigned Single character	1 Byte	0	255

Floating Point Data types :-

Data type	Require memory	minvalue	Max value
float	4 Bytes	3.4e-38	3.4e+38
double	8 Bytes	1.7e-308	1.7e+308
long double	10 Bytes	3.4e-4932	1.1e-4932

Variables

A variable is an Identifier that is used to represent some specified type of Information within a designated Portion of program. A variable may have different values at Different times during the execution i.e. It is the Named Memory location

Rules for Naming the Variable

→ A variable Name can be Any combination of 1 to 8 Alphabets Digits (or) Underscore.

→ The first character must be an Alphabet (or) an Underscore (_).

→ No commas (or) blank spaces are allow within a special symbol, a underscore can be used a variable

Variable Declaration:-

Variable Names, we must Declare them in a Program and this Declaration tells the compiler what the Variables Name and type of the Data that the Variable will hold.

Syntax:- data type $V_1, V_2, V_3 \ldots V_n$;

Description:- data type - is the type of the Data.

$V_1, V_2, \ldots V_n$ - are the list of Variables.

Example:

```
Int code;
char Sex;
float Price;
char name [10]
```

Initializing Variables:-

Initialization of variables can be done Using the Assignment Operator (=). The variables can be Initialized while Declaration Itself.

Syntax:-

```
Variable = Constant ;
   (or)
datatype Variable = Constant ;
```

Description:-

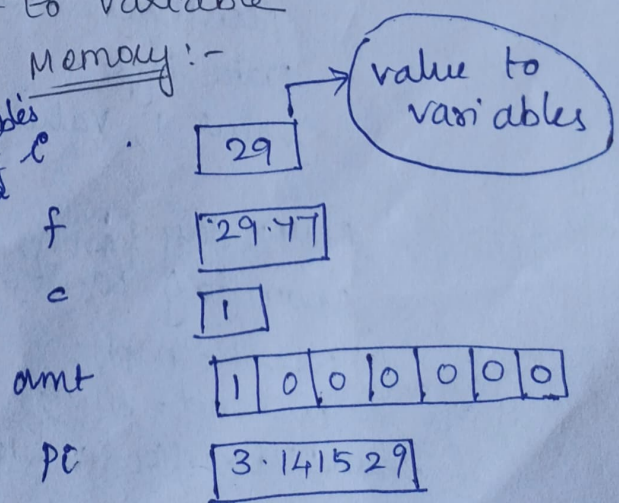
i, f, c are the type of int, float, char Data types.

program:-

```
Variable ← int i = 29; → Value to variable
  type     float f = 29.77;
           char c = 'i';
           long amt;
           double pi;.
```

(→ Variable Identifier)

Memory:-



User defined variables :-

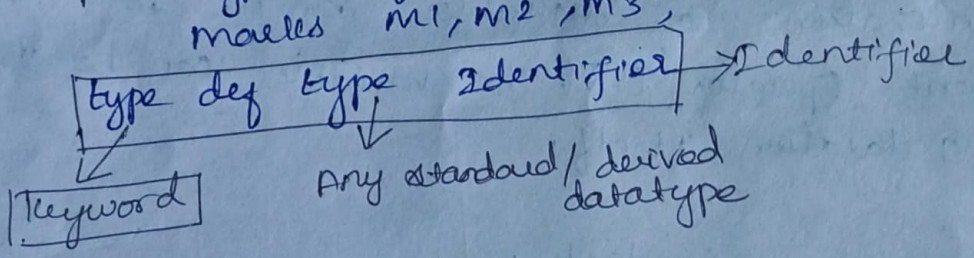
(a) type Declaration:

C Language Provides a feature to Declare variable of the type of user Defined type Declarat the typedef is a keyword. in the C Programmin languages.

Syntax :- typedef data type Identifier;

Description :- type def is the user Defined type declaration data type is the existing Datatype Identifier is the Identifier Refers to the New Name given to the Data type.

Example: type def int marks;
marks m1, m2, m3;



Enumerated Datatype

The C language Provides Another User defined dat type called Enumerated Datatype

Syntax: enum identifier {value 1, value 2, ... Value

Description: Identifier is the User Defined Enumerate Data type.
value 1, value 2 ... value n are the enumeration constants

example: enum day {mon, tue, wed ... sun};
enum day w-st, w-end;
w-st = mon;
w-end = sun;
enum Identifier {value 1, value 2 .... value n};

User defined enumerated Data type

enumerated constants

Scope of variable.

Scope of a variable Implies the Availability of Variabl within the program. Two types of scopes.

(i) Local variables:

The vaubles which are Defined inside a function a function Block (or) Inside a Compound statement of a function subprogram are called local variables.

eg: function ()
{
int i, j;
/* body of the function */

The integer variables i, j are Defined Inside the block of function (). Hence i, j are called local variables.

(ii) Global / External variables:-

An operator is a symbol that tells the compiler to perform Specific mathematical (or) Logical functions. C language is rich in Build in operators and provides the following types of operators.

1) Arithmetic operators
2) Relational operators
3) Logical operators
4) Bitwise operators
5) Assignment operators
6) Miscellaneous operators

1) Arithmetic Operators:

All the Arithmetic operators supported by the c language. Assume variable A holds 10 and variable B holds 20 then.

operator	Description	example (A=10, B=20)
+	Add two operands	A+B = 30
−	subtracts second operand from the first	A−B = −10.
*	Multiplies both operands	A*B = 200
/	Divides Numerator by de-Numerator	B/A = 2
%	Modulus operator and Remainder of after an Integer Division	B%A = 0
++	Increment operator Increases the Integer value by one	A++ = 11
−−	Decrement operator decreases the Integer value by one	A−− = 9

2) Relational operators:-

All the Relational Operators supported by C
Assume Variable A holds 10 and variable B holds 2
then

Operator	pescription	examp
==	Check if the values of two operands are equal (or) not. If yes then the condition becomes true.	(A==B) is not true

!=	Checks if the values of two operands are Equal (or) Not. If the values are Not. If the values are not Equal then the condition becomes true	(A!=B) is true
>	Check if the values if left operand is operator then the value of Right operand If yes, then the condition Becomes true	(A>B) is Not true
<	check if the value of left operand is left thanthe. value of Right operand. If yes then the condition becomes true.	(A<B) is true.
>=	Chick if the value of left operand is greater than (or) Equal to the value of Right Operand If yes then the condition becomes true.	(A>=B) is not true
<=	checks if the value of left operand is less than (or) Equal to the Right	(A<=B) is true

Relational operator Eg. Program:-

```
#include <stdio.h>
#include <conio.h>
void main()
printf("\n condition : Return Values \n");
printf("\n 5!=5 : %5d", 5!=5);
printf("\n 5==5; %5d", 5==5);
printf("\n 5>=5; %5d", 5>=5);
printf("\n 5<=5; %5d", 5<=5);
printf("\n 5!=3; %5d", 5!=3);
}
```

Output:-

Condition : Return Values.

5 != 5 : 0

5 == 5 : 1

5 >= 5 : 1

5 <= 50 : 1

5 != 3 : 1

3) Logical operators

Logical operators are used to combine the Result two or more Conditions 'c' has the following L operators.

operator	Meaning	Example	Return Va
&&	Logical AND	(9>2)&&(17>2)	1
||	Logical OR	(9>2)||(17==7)	1
!	Logical NOT	29!=29	0

&& eg:

(exp1) && (exp2)

|| eg:

(exp1) || (exp2)

! eg:

! (exp1)

example:

i is an Integer variable, value is 7

f is a float variable, value is 5.5

c is a character variable, that represents character 'w'

Expression	Interpretation	value
(i >= 6) && (c == 'w')	True	1
(f < 11) && (i > 100)	false	0
(c != 'p') || (i <= 100)	True	1

Program :-

```
# include <stdio.h>
#include <conio.h>
void main ()
{
int c1, c2, c3;
clrscr();
printf("Enter Values of c1, c2 & c3: ");
scanf("%d %d %d", &c1, &c2, &c3);
```

```
if((c1<c2) && (c1<c3))
printf("\n c1 is lesser than c2 and c3");
if((c1<c2) && (c1<c3))
if(c1<c2)
printf("\n c1 is greater than c2");
if((c1<c2) || (c1<c3))
printf("\n c1 is less than c2 (or) c3
                (or) both");
getch();
}
```

output:

```
Enter values of c1, c2 & c3 : 45 32 9
c1 is greater than c2
c1 is less than c2 or c3 (or) both
```

4) Assignment Operators:

Assignment operators are used to assign a value (i) an expression (ii) a value of variables for another variable.

Syntax: Variable = expression (or) value;

Description: Variable is any valid 'c' variable, assigned value can be anything.

Eg:

x = 10; x = a+b; x=5

prgm

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

output

k=5

(i) Compound Assignment:-

Apart from assignment operator (=), 'c' also provides compound Assignment operators to Assign a value to a variable in order to Assign a New Value.

operator	example	Meaning
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

(ii) Nested (or) Multiple Assignments :-

C language has got distinct features in Assignment called Nested (or) Multiple Assign

Syntax:

Var 1 = Var 2 = Var 3 = ... Var n = single variable (or) expres

Description: Var 1, Var 2, Var 3 .... Var n are values 'c' variables

example:

```
i=j=k=l;
x=y=z=(i+j+k);
```

5) Bitwise operators :-

Bitwise operators are used to manipulate th data at bit level. It operates (or) Integers only. May Not be applied to float (or) Real.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift Right
~	One's complement

Bitwise AND (&)

This operator is Represented as '&' a operates on two operands of Intoge type

truth table:

&	0	1
0	0	0
1	0	1

eg:

$x = 7 = 0000\ 0111$

$y = 8 = 0000\ 1000$

$x \& y = 0000\ 0000$

AND

00 → 0

01 → 0

10 → 0

11 → 1

Bitwise OR (|) :-

Similar to AND operator in all Aspects the truth table for Bitwise OR. But this operator gives if either of the operand Bit is '1' then Result is '1' both operands are '1' then ; also given '1'.

|	0	1
0	0	1
1	1	1

Truth table.

ex:

$x = 7 = 0000\ 0111$

$y = 8 = 0000\ 1000$

$x | y = 0000\ 1111$

OR

00 → 0

01 → 1

10 → 1

11 → 1

Bitwise Exclusive OR (^)

Similar to AND operator in all Aspects but Integer gives if either of the operand bit is high (1) then it gives high (1) Results.

A	0	1
0	0	1
1	1	0

truth table

eg:

$x = 13 = 0000\ 1101$

$y = 8 = 0000\ 1000$

$x^{\wedge}y = 0000\ 0101$

XOR

00 → 0

01 → 1

10 → 1

11 → 0

truth table:

&	0	1
0	0	0
1	0	1

eg:

$x = 7 = 0000\ 0111$

$y = 8 = 0000\ 1000$

$x \& y = 0000\ 0000$

AND

00 → 0

01 → 0

10 → 0

11 → 1

Bitwise OR (|) :-

Similar to AND operator in all Aspects the truth table for Bitwise OR. But this operator gives if either of the operand Bit is '1' then Result is '1' both operands are '1' then ; also given '1'.

|	0	1
0	0	1
1	1	1

Truth table.

ex:

$x = 7 = 0000\ 0111$

$y = 8 = 0000\ 1000$

$x | y = 0000\ 1111$

OR

00 → 0

01 → 1

10 → 1

11 → 1

Bitwise Exclusive OR (^)

Similar to AND operator in all Aspects but Integer gives if either of the operand bit is high (1) then it gives high (1) Results.

A	0	1
0	0	1
1	1	0

Truth table

eg:

$x = 13 = 0000\ 1101$

$y = 8 = 0000\ 1000$

$x \wedge y = 0000\ 0101$

XOR

00 → 0

01 → 1

10 → 1

11 → 0

eg:

a	b	a|b	a&b	a^b	~a
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Program :-

```
#include <stdio.h>
#include <conio.h>
void main()
{
char c1, c2, c3;
printf("Enter values of c1 and c2 : ");
scanf("%c, %c", &c1, &c2);
c3 = c1 & c2;
printf("\n Bitwise AND i.e. c1 & c2 = %c", c3);
c3 = c1 | c2;
printf("\n Bitwise OR i.e. c1 | c2 = %c", c3);
c3 = c1 ^ c2;
printf("\n Bitwise XOR i.e. c1 ^ c2 = %c", c3);
c3 = ~c1;
printf("\n Ones Compliment of c1 = %c
c3 = c1 << 2;
printf("\n left shift by 2 bits c1 << 2 = %
c3);
c3 = c1 >> 2;
printf("\n right shift by 2 bits c1 >> 2 =
getch();
}
```

Output:

c_1 c_2

Enter values of c_1 and c_2 : 5 9

Bitwise AND i.e $c_1 \& c_2 = 1$

Bitwise OR i.e. $c_1 | c_2 = 13$

Bitwise XOR i.e. $c_1 ^ c_2 = 12$

Ones compliment of $c_1 = -6$

left shift by 2 bits $c_1 << 2 = 18$

Right shift by 2 bits $c_1 >> 2 = 4$

A) Expressions

An Expression represent data Item such as variables, constant and are Interconnected with operators as per the syntax of the language. An Expression evaluated using Assignment operation.

Syntax: Variable = expression;

Description: Any 'c' valid variable and Expression

Example: $x = a * b - c$;

Examples of Algebraic Expressions and C Expressions

Algebric Expression	C expression
$a+b*c$	$a + b * c$
ax^2+bx+c	$a * x * x + b * x + c$
$\left(\frac{4ac}{2a}\right)$	$(4 * a * c) / (2 * a)$
$\left(\frac{2x^2}{b}\right) - c$	$((2 * x * x) / b) - c$

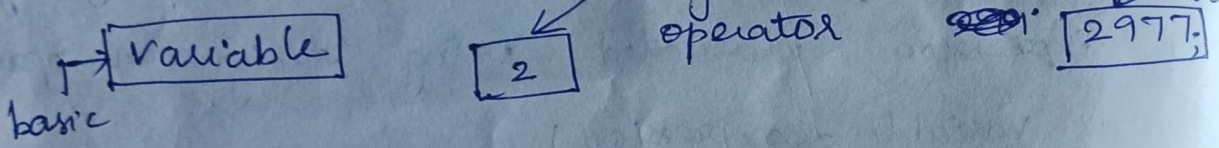
Statements :-

Statements can be defined as a set of declaration (or) sequence of Action. Statement cause the computer to Perform some Action.

Assignment Statements:

Here the Assignment operator is used for Assigning values to the variable.

eg: basic = 2977;
dob = 2977;
flag = 0;
i = a = k = 2977;



Null statements :

A statement without Any characters and if only semicolon is called Null statement

eg: ; (null statement)

Block of statements

Block contains several statements that are Enclosed within a pair of braces {}. There can be of any Expression, Assignments and Keywords etc.

eg: { ⟶ opening Brace
int a = 2977;
float b = 29.77;

printf ("%d %f", a, b);

y → close Brace

Expression Statement :

These consists of Expressions and can be Arithmetic, Relational (or) logical.

Eg: a=29;
b=a+77;
fun(a,b);

5) Conditional Statement

The Basic Decision statements in Computer is Selection structure. The Decision is described to computer as conditional statement that can be answered True / False.

(i) Sequential structure:

In which Instructions are Executed in sequence

example i = i+1;
j = j+1;

(ii) Selection Structure:

Here the sequence of the Instructions are Executed in sequence Determined by using the Result of the condition.

eg: if (x>y)
i = i+1;
else
j = j+1;

(iii) Iteration structure

In which statements are repeatedly executed these forms program loops.

eg: for (i=1; i<=5; i++)
{
i = i+1;
y

'c' language Provides the following conditional (de making) statements.

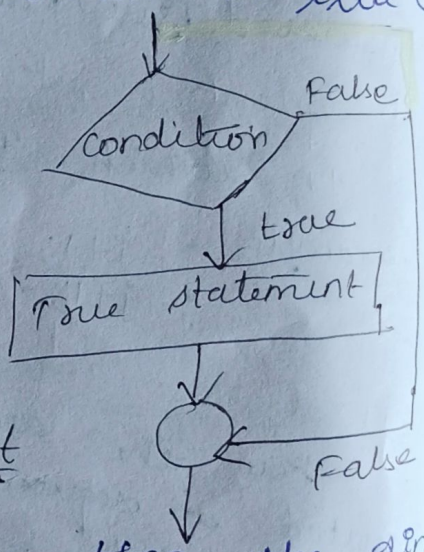
→ if statement
→ if .... else statement
→ nested if ... else statement
→ if ..... else ~~Ladder~~ Ladder

If statement :-

The "If" Statement is a decision Making st It is used to control the flow of executio of the statements and also used to test logically whether the condition is true false

Syntax :-

```
if (condition is true)
{
True statements;
}
```



Properties of an "if" statement

→ If the condition is true, then the sim or compound condition statements are Exe

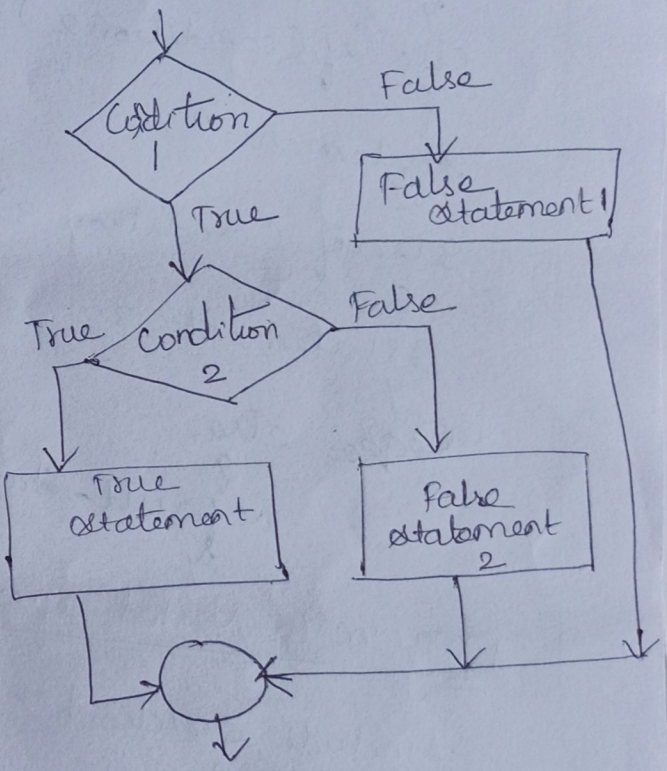
→ If the Condition is false it doesnot anything.

eg:
```
#include <Stdio.h>
printf ("\n Enter the Number<10....");
Scanf ("%d", &i);
if (i<10)
printf ("\n The entered Number i'd is <10
```

output: Enter the Number <10.....5
The entered Number 5 is <10.

Nested if.....else statement:

when a series of if....else statements occur in a Program, we can write an Entire if...else statement in another if...else statement. called Nesting and the statement is called Nested if.



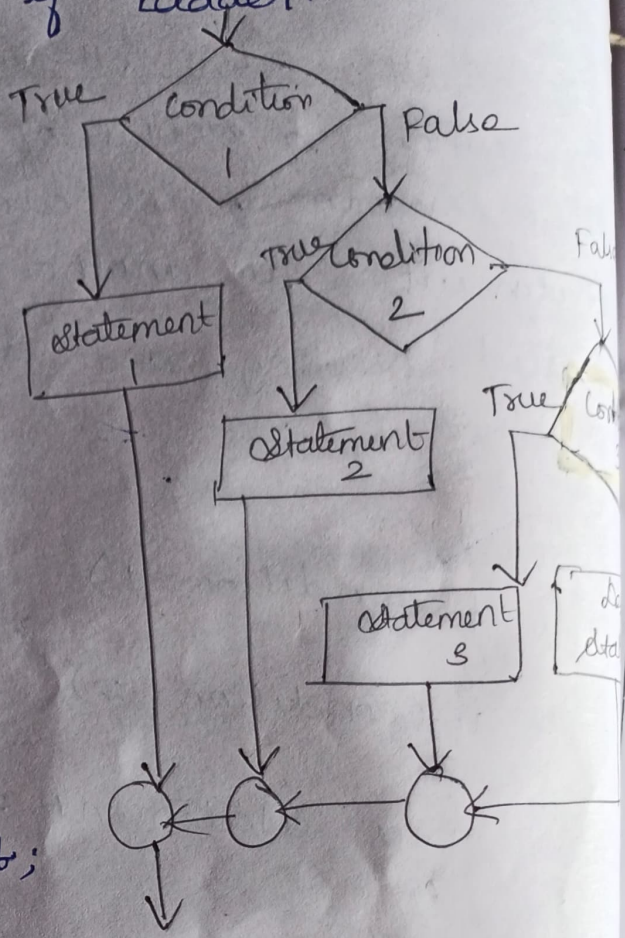
Syntax:

```
if (condition 1)
{
if (condition 2)
 {
 True statement 2;
  }
else
{
false statement 2;
}
} }
else
 {
false statement 1;
 }
```

The if....else ladder

Nested if Statements can become quite complex. If there are more than three Alternatives and Identation is not consistent. It may be different for you to Determine the logical structure of

if statement. In situations you can used the nested if as the else if ladder.



Syntax:-

```
if (condition 1)
{
statement 1;
}
else if (condition 2)
{
statement 2;
}
else if (condition 3)
{
statement 3;
}
else
{
default-Statement;
}
```

program to Nested if....else statement

```
#include <stdio.h>
main()
{
int n;
printf("\n Enter a Number....");
scanf("%d", &n);
if (n==15)
printf("\n play foot ball");
else
{
if (n==10)
printf("play cricket");
else
printf("don't play");
}}
```

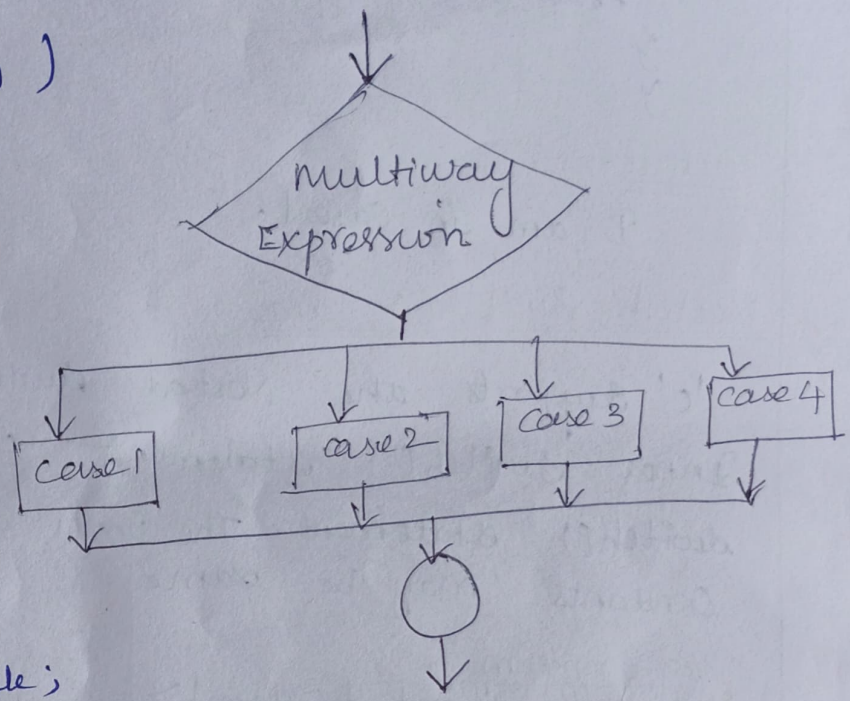
output:
Enter a Number
play cricket

switch statement :-

The switch statement is used to pickup (or) execute a variable group of statements from several available group of statements :-

syntax :-

```
switch (expression)
{
case constant 1:
  block 1;
  break;
case constant 2:
  block 2;
  break;
  :
default:
  default Block;
  break;
}
```



Rules for writing switch() statement :-

→ The expression in switch statements must be an Integer value or a character constant.

→ No Real Numbers are used in an expression.

→ A two case constants are Identical.

→ The case keyword must terminate with colon (:)

eg pgm :-

```
#include <stdio.h>
#include <conio.h>
main()
{
int a=1;
switch(a)
{
case 1:
  printf("I am in case 1 \n");
  break;
```

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

output:

I am in case 1.

Nested switch() case

'C' supports the Nested switch() statements. The Inner switch() statement can be a part of an switch() statement. The Inner and outer switch() Constants may be same.

Eg program:

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int a,b;
clrscr();
printf ("Enter a number:");
scanf ("%d", &a);
switch (a)
{
case 0:
   printf ("The Number is Even. \n");
   break;
case 1:
   printf ("The Number is odd \n");
   break;
default:
      b=a%2;
switch(b)
{
case 0:
  printf ("The Number is Even. \n");
```

```
break;
default:
printf("The Number is odd.\n");
}
}
getch();
}
```

output:
Enter a Number: 89
The numbe is odd

o/p 2:
enter a Number: 48
The Number is even

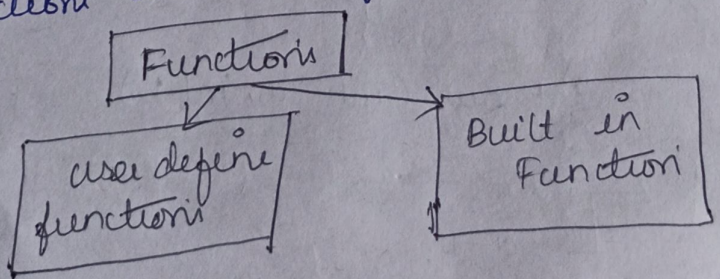
Comparison of switch() case and Nested if:-

switch ( ) case	Nested if:
(1) The switch() can test only constant values	The If can Evaluate Relational (or) logical expression.
(2) character constants are automatically converted to Integers.	character constants are automatically converted to Integers.
(3) In switch() case statement Nested if can be used	In Nested if statements switch() case can be used.

b) Functions:-

A function is a set of Instructions that are perform specified tasks with repeatedly occurs in main program. Function is a set of Instructions that provide Modularity to the software.

Functions are classified in to two types.



Declaring, Defining And Accessing Functions :-

(1) Userdefined function :-

The function defined by the User Accordi to the Requirements are called User-defined functions.

The Users can Modify the function Accord to Requirement.

Need for User-defined functions :-

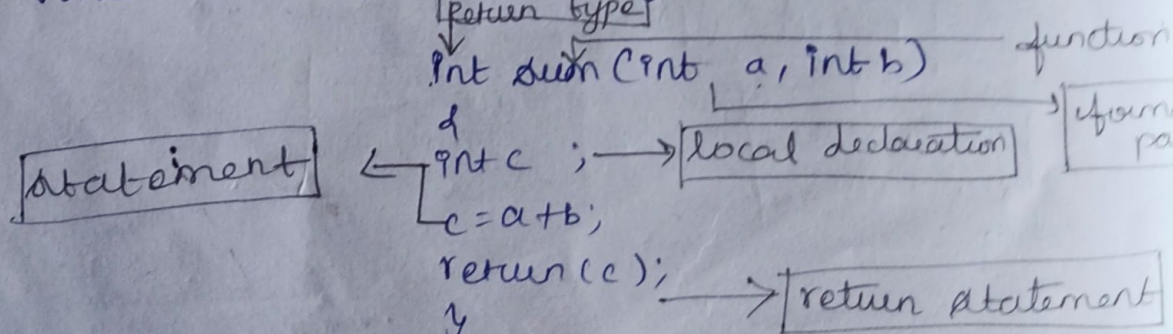
(i) The program becomes too large and complex.
(ii) The Users can not go through at a glance.
(iii) The task of debugging, testing and Maintonance becomes difficult.

Advantages of userdefined functions :-

(i) By using functions it is very Easy to locate and debug an Error.
(ii) Functions facilitate top-down preprocessing.
(iii) The User-defined is called it takes some da from the calling function and Return back Value to the called function.

How function work :-

Once a function is called it takes some d from the calling function and Return back some Value to the called function.



Elements of User-defined functions :-

following three Elements:

(a) function definition
(b) function declaration
(c) function call

function definition:

It is the Proces of specifying and Establishing the user defined function by specifying all of its.

Function Declaration :-

Like the Normal variables in a Program the function can also be declared befor they are defined and involced.

Syntax: return_type function name (parameter)

Description: return type → is the return of the function
function_name → is the Name of the function
parameters List → are the list of parameter that the function can convey.

ex: int add (int x, int y, int z);

Syntax:-

```
datatype functionname (Parameters List)
parameters declaration ;
{
local variable declaration ;
....
body of the function;
.....
return (expression);
}
```

Function call

The function can be called by simply specifying the name of the function, Return value and parameters if present

Syntax:- function_name ();
function_name (parameter);

Return value = function_name (parameter);

ex:
```
fun();  /* function without arguments and Return value */
fun(a,b); /* function with Argument */
c=fun(a,b) /* function with arguments and Return Va
```

Eg: function call without Parameter :-

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

output:
function Messag
Main Message

Function call with Parameter:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int k;
clrscr();
k=add(20,20);
printf("The value of k is %d\n",k);
getch();
}
int add(int x, int y)
{
int z;
z=x+y;
return(z);
}
```

output

The value of k is 30.

1) Recursive Functions :

Recursive functions is a Programming technique that allows Programmer to express operations in terms of in C this takes the form of a function that call.

Syntax:

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

In order to write a Recursive Progam, that User satisfy the following.

→ The Problem must be Analysed and written in satisfy the following:

→ The Problem must have the stopping condition

Eg:

Calculating the factorial of an Integer Number

$n! = 1 \times 2 \times 3 \times \dots \times n$ (when 'n' is an Integer)

express this by $n! = n*(n-1)!$

$5! = 5*(5-1)! = 5*4!$

$4! = 4*(4-1)! = 4*3!$

then the final Answer:

$1! = 1*(1-1)! = 1*0! = 1*1 = 1$

$2! = 2*(2-1)! = 2*1! = 2*1 = 2$

$3! = 3*(3-1)! = 3*2! = 3*2 = 6$

Program:

```
#include <stdio.h>
main()
{
int a;
printf("Enter the number:");
Scanf("%d", &a);
```

```
printf ("The factorial of %d = %d", a, rec(a));
}
rec(int x);
int f;
{
if (x == 1)
return (1);
else
f = x * rec(x-1);
return(f);
}
```

Output

Enter the Number
The factorial of 5 = 12

8. Arrays:

An arrays is a collection of similar Data It that are stored under a Common Name. A valu in an Array is Identified by Index (or) subscri enclosed in square Brackets with Array Nam

'n' → Elements Array

'Q' → Element.

Q[0], Q[1], Q[2], Q[3], ... Q[n-1].

Needs of an Array:-

→ Enables us to define a set (or) similar dat Term known as An array.

→ Suppose you have a set of Marks that you want to read into the computers.

```
printf ("Enter Marks: \n");
scanf ("%i", &Marks 1);
printf ("Enter Marks 2: \n");
scanf ("%i", &Marks 2);
printf ("Enter Marks 3: \n");
printf ("%i", &Marks 3);
...
```

Arrays can be classified into

→ One-Dimensional Arrays
→ Two-Dimensional Arrays.
→ Multi-Dimensional Arrays.

One-Dimensional Arrays:

The collection of data Items can be stored under a one variable Name Using Only one Subscript such as variable is called the one-dimensional Array.

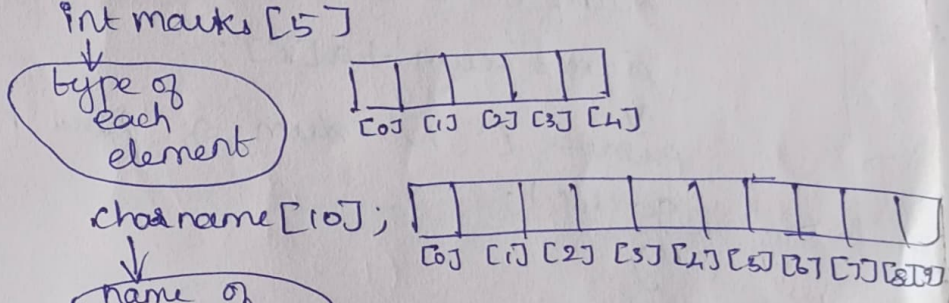
Array Declaration

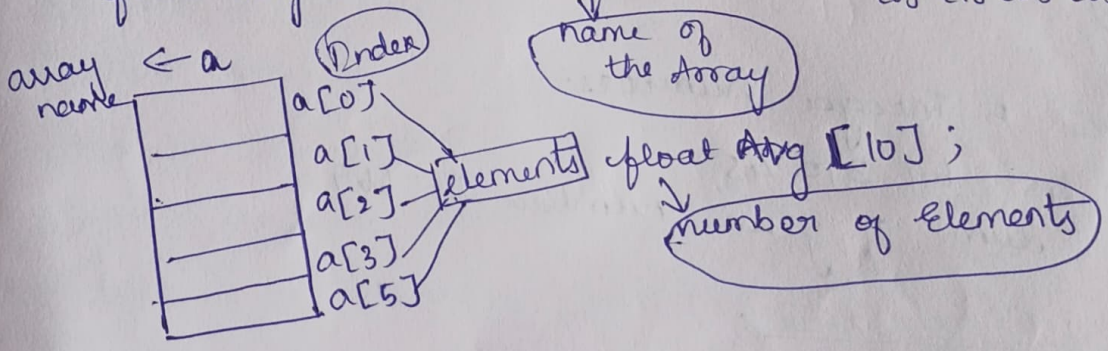
Syntax:-

data_type array_variable[Size (or) Subscript];

Eg:

int a[5];
int marks[5];
char name[10];
float avg[10]





Types of Arrays:-

An Array is a derived datatype.
→ It is used to represent a collection of elements of the same datatype.
→ In Array, the elements are stored in continous memory location.

Processing of an Array:-

The Entire Array cannot be accessed with single operation. So the Array Elements must be Accessed on an Element-by-Element Basices.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a[5], sum=0, i;
clrscr()
 printf("Enter 5 integer numbers\n");
for(i=1; i<=5; i++)
 {
scanf("%d", &a[i]);
sum = sum + a[i];
}
printf("The sum of given numbers is : %d \n", sum);
getch();
}
```

Output:

```
Enter 5 integer numbers.
120 45 69 390 45
The sum of given numbers is : 669
```

Array Initialization:

The values can be initialised to an Array, when are declared like ordinary variables, otherwise they hold garbage values.

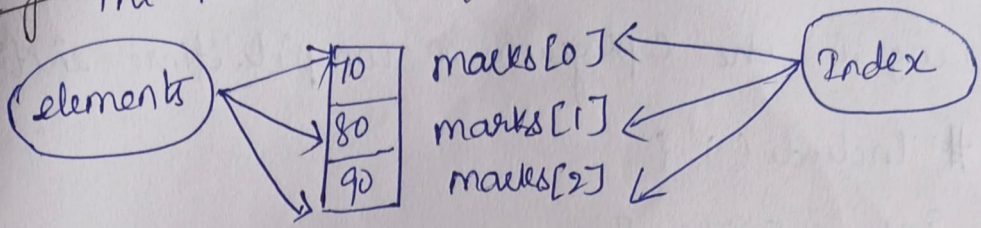
Two ways:

(i) At compile time
(ii) At Run time

At compile time:

Syntax :- data_type array_name[size] = { list of values };

eg: Int marks[3] = {70,80,90};



marks[0] = 70;
marks[1] = 80;
marks[2] = 90;

At Runtime:

The Array can be explicitly Initialized at Runtime.

eg:
```
while (i<=10)
  {
  if (i<5)
  sum[i]=0;
    else
    sum[i]=sum[i]+1
  }
```

Enter the 3 student rollno and mark : 6683 85

0 student roll no 3977 mark 80
1 student roll no 1776 mark 95
2 student roll no 6682 mark 82
3 student roll no 6683 mark 85

Data stored in memory location ⇒

	col 1	col 2
row1	3977	80
row2	1776	95
row3	6682	82
row4	6683	85

Single dimensional Arrays:

The syntax is as follows:-

datatype arrayname [size]

example: int a[5]

Initialization

An array can be initialized in two ways, which are as follows

* compile time Initialization
* Runtime Initialization

example:

following is the c program on compile time initialization

```
#include() {
int a[5]= {10,20,30,40,50};
int i;
printf("elements of the array are");
for(i=0;i<5;i++)
printf("%d",a[i]);
}
```

output:

Elements of the array are

10 20 30 40 50.

runtime initialization

following is the c program on runtime initialization

```
#<stdio.h>
main() {
int a[5], i;
printf("enter 5 elements");
for(i=0;i<5;i++)
scanf("%d",&a[i]);
printf("elements of the array are");
for(i=0;i<5;i++)
printf("%d",a[i]);
}
```

output:-

enter 5 elements 10 20 30 40 50

elements of the array are: 10 20 30 40 50

example :

following is another c program for one dimension array.

```
#include <stdio.h>
int main (void) {
 int a[4];
 int b[4]={1};
 int c[4]={1,2,3,4};
 int i;
printf ("\n Array a:\n");
for (i=0; i<4; i++)
 printf("array[%d]:%d\n", i, a[i]);
 printf("\n Array b:\n");
for(i=0; i<4; i++)
 printf ("arr[%d]:%d\n", i, b[i]);
 printf("\n Array c:\n");
for(i=0; i<4; i++)
 printf("arr[%d]:%d\n", i, c[i]);
 return 0;
}
```

output

```
Array a:
arr[0]:8
arr[1]:0
arr[2]:54
arr[3]:0
Array b:
arr[0]:1
arr[1]:0
arr[2]:0
arr[3]:0
Array c:
arr[0]:1
arr[1]:2
arr[2]:3
arr[3]:4
```

Multi-Dimensional Arrays:-

Similarly like one and two dimensional Arrays 'C' language allows Multi-dimensional Arrays. The Dimension with three (or) more called Multi-Dimensional Arrays.

Syntax:

data-type array-name [size 1][size 2]....[size n]

eg: int a[3][3][3]

float table, b[4][4][4][4]

→ where a is the three dimensional array declared as Integer type and can be 27 elements

→ b is the 4 dimensional Array declared as float type and can be 256 elements.

m(i,j)

	0	1	2	3
0	9	3	20	82
1	9	2	20	07
2	13	8	20	13
3	17	7	19	76

consider 4×4 matrix

$$\begin{bmatrix} 9 & 3 & 20 & 82 \\ 9 & 2 & 20 & 7 \\ 13 & 8 & 20 & 13 \\ 17 & 7 & 19 & 76 \end{bmatrix}$$

Mathematics the Notation M[i,j] is used in 'C' the Equivalent Notation is M[i][j]

```
int m[4][4] = { {9,3,19,82};
                {9,2,20,07};
                {13,8,20,13};
                {17,7,19,76} };
```

int M[4][4] = {9,3,19,82,9,2,20,07,13,8,20,13,17,7,19,76};

Example program:-

```
#include <stdio.h>
#include <conio.h>
void main ()
 {
 int i,j,k;
 int a[2][3][4]
 int b[3][4];
 int c[4];
 int int=0;
 clrscr();
 for (i=0; i<2; i++)
 for (j=0; j<3; j++)
 for (k=0; k<4; k++)
 {
 a[i][j][k]=int;
 } int++;
 print_onedim(a);
 print_twodim(a);
 }

print onedim(a)

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

```
print -twodim (int a[][4])
{
 int j;
 for (j=0 ; j<3; j++)
 print -onedim (a[j]);
 print ("\n");
 }

 print - threedim (int a[][3][4])
  {
  int j;
 printf ("Each two dimension matrix\n");
  for (j=0 ; j<2; j++)
   print -twodim (a[j]);
    getch ();
      }
```

Output:

```
0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11
Each two dimension Matrix
```

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23

Unit-2

C-Programming - Advanced Features

Structures - Union - Enumerated Data Types - Pointers: Pointers to Variables, Arrays and Functions - File Handling - Preprocessor Directives.

---

1) Structures:-

Need for structures:-

→ Arrays can store many values of similar Data types.

→ Data in the Array is of the same composition and Nature as far as type is concerned.

→ All these Data types cannot be expressed in a single array.

Structure Definition:-

⇒ A structure contains one (or) more Data Items of Different Datatype in which the Individual elements can differ in type. A simple structures may contains the Integer elements, float elements and character elements etc, the Individual structure Element are called Members.

⇒ structures are declared by using the "struct" keyword.

Uses of Structures:-

→ It is possible to pass structure Elements to a function. This is similar to passing an ordinary variable to a function.

→ It allows Grouping together of Diffrent type elements.

→ It also possible to create structures Pointers

Declaration of structure

```
Struct structure-name
{
data-type member1; data-type member2; ...
data_type membern;
};
```

we can create variable of structure

Struct structure_name V_1, V_2, V_3;

V_1, V_2, V_3 are variable

Example:-

Consider a student database consisting of student Name, Age, Weight and Pay

```
Struct stud_data
{
char name[30]; int age;
float weight;
int pay;
};
Struct stud_data S1, S2, S3
```

The above example declares a structure "stud_data" with 4 members. Note that the members are of Different Datatypes. where S_1, S_2, S_3 are calle structure variable.

Rules for Declaring a Structure:

→ A structure must end with a semicolon.

→ Each structure Element must be terminated.

→ Usually a structure appears at the top of down program

→ The structure Element must be Accessed by structure variable with dot (.) operator.

eg:

```
Struct stud_data raja, srini, value;
```

define raja, srini, value as variables of type struct. stud_data.

```
Struct structure_name variable_name1, variable_name2...;
```

for the above example 38 byte are allocated afor each variable

name - 30 bytes (30*1) Age - 2 bytes
weight - 4 bytes pay - 2 bytes

Example:-

```
Struct stud_data
{
char name[30]; int age;
float weight; int pay;
} rain, srini, valu;
```

is also valid.

Example program 1:

```
#include<stdio.h>
#include<conio.h>
Struct employee
{
char name[10]; int idno;
float salary;
} e;
main()
{
printf("enter the name\n");
scanf("%s", &e.name);
printf("Enter the idnumber\n");
scanf("%d", &e.idno);
```

```
printf(" enter the Salary \n");
scanf("%f", &e.salary);
printf("Name: %s\n", e.name);
printf("ID Number: %d\n", e.idno);
printf("Salary: %f\n", e.salary);
getch();
}
```

Output:

```
Enter the name
Priyanga
Enter the id number
100
enter the Salary
20000
Name: G. Priyanga
ID Number: 100
Salary: 20000.00000
```

Initialization of structure :-

The members of a structure can be Initialized to constant values by enclosing the values to be Assigned within the braces After the structures definition. Thus the Declaration.

```
Struct date
{
int date;
int month;
int year;
} Indepence = {15, 08, 1947};
```

Accessing And giving values to structure members

After Declaring the structure type, variables and members. The members of the structure can be Accessed by using the structure variable along with (.) operator.

Syntax: Variable_name . member_name

eg: Struct book
{
int id;
char name[20];
};
Struct book b1;

for Accessing the structure members from the above example:

Struct book
{
int id;
char name[20];
};
Struct book b1;

For Accessing the structure members of from the above example:

b1.id;
b1.name; where 'b1' is the structure variable

Example Program:

```
#include <stdio.h>
Struct book
{
int id;
char name[20];
char author[15];
};
main()
{
Struct book b1; /* Accessing the member values
```

```
printf("\n Enter the book id, Book Name and Author of the
                Book for the structure variable \n");
scanf("%d \n %s \n %s \n", &b1.id, b1.name, b1.author);
                /* printing the member values */
printf("\n Book id is = %d", b1.id);
printf("\n Book Name is = %s", b1.name);
printf("\n Book Author is = %s", b1.author);
}
```

Structure within structure :-

A structure can be declared within Another structure. Sometimes it is Required to keep a compound Data Items within Another compound Data Item is called structure within structure (or) its meaning Nesting of structures.

Syntax :- Struct Structure_name1
{
declaration 1; declaration 2; .... declaration N;};

Struct Struct_name2
{
declaration 1; declaration 2;
Struct structure_name1 variable_name1;
:
declaration n;
};

Example program

```
#include <stdio.h>
Struct date
{
int date, month, year;
}dob;
```

```
Struct str_data
{
char name[20];
Struct data dob;
} s;
main ()
{
Struct str_data S={"Priya", {28,01,2000}};
printf ("\n Name %.s", s.name);
printf ("\n\n Date of Birth :%d - %.d - %.d", s.dob.date,
                                    s.dob.month, s.dob.year);
}
```

output

Name : Priya
Date of Birth : 28.01.2000

Arrays of Structure

The 'c' language Permits to Declare an Array of Structure Variable. Use the Array of Structure Variable to store them in one Structure Variable.

Syntax:-

```
Struct structure_name
{
declaration 1, declaration 2;
:::::
declaration;
} Variable_name [size];
```

Example Program:-

```
#include <stdio.h>
#include <conio.h>
Struct student
{
char name[10];
```

```
int number m1, m2, m3, total;
float Avg;
}n[10];
main(.)
{
int i, n;
Clescr();
printf("Enter the Number of students");
scanf("%d", &n);
for (i=1; i<=n; i++)
{
printf("Student- details %d \n\n", i);
printf("Enter the student name \n");
scanf("%s", &n[i].name);
printf("Enter the student roll number and Marks\n");
scanf("%d %d %d %d", &n[i].number, &n[i].M1,
                     &n[i].M2, &n[i].m3);
n[i].total = n[i].m1 + n[i].m2 + n[i].m3;
n[i].Avg = n[i].total/3;
}
printf("student details:");
for(i=1; i<=n; i++){
printf("Name : %s \n", n[i].name);
printf("Rollno: %d \n", n[i].number);
printf("total : %d \n", n[i].total);
printf("Average : %f \n", n[i].number);
printf("Total : %d \n", n[i].total);
printf("Average : %f \n", n[i].Avg);
}
getch();
}
```

output:-

```
Student details
Enter the Student name
Priya
Enter the Student roll number and marks 100 97 98
Name : Priya
Roll no: 100
Total: 295
Average: 98.0000
Employee details
Enter the Employee name :
Akila
Enter the ID number 101
Enter the Employee salary 20000
Employee Name : Akila
Employee ID Number : 101
Employee Salary : 20000
```

Structure with Functions:-

When a structure is used as a Parameter to a fuction the entire structure is Passed to the function using the call by value Method.

Example Program :-

```
#include <stdio.h>
#include <conio.h>
void struct fun();
Struct student
{
int rollno;
char name[10];
float marks;
} s;
```

```
main ()
{
  clrscr();
  printf("%d", &s.rollno);
  printf("Enter the roll number\n");
  scanf("%d", &s.rollno);
  printf("Enter the student name\n");
  scanf("%s", &s.name);
  printf("Enter the student marks\n");
  scanf("%f", &s.marks);
  Struct fun(s); // structure variable as a parameter
  getch();
}
void Structfun(Struct student s) // function Definition
{
printf("Roll number: %d\n", s.rollno);
printf("Name: %s\n", s.name);
printf("marks: %.f\n", s.marks);
}
```

Output:

```
Enter the roll number 101
Enter the student name
priya
Enter the student marks
100
Roll number : 101
Name : Priya
Marks : 100.000
```

Pointers to structures:-

Sometimes it is useful to Assign Pointers to structures. Declaring Pointers to structures is Basically the same as Declaring a Normal Pointer.

EX:-

```
type def Struct {
Char firstName[20];
Char LastName[20];
Char SSN[10];
float gpa;
}student;
```

Example program:-

```
#include <stdio.h>
#include<stdlib.h>
Struct name{
int a;
float b;
Char c[80];
};
int main(){
Struct name *ptr; int i, n;
printf("Enter n;");
scanf("%d", &n);
ptr = (Struct name*) malloc(n*sizeof(Struct Name));
for(i=0; i<n; ++i)
  {
printf("Enter String, integer, and floating Number
                                respectively: \n");
scanf("%s%d%f", &(ptr+i)->c, &(ptr+i)->a,
                                &(ptr+i)->b);
}
printf("Displaying Information : \n");
for(i=0; i<n; ++i)
printf("%s \t %d \t %.2f \n", (ptr+i)->c,
                    (ptr+i)->a, (ptr+i)->b));
return 0; }
```

output:

Enter n:2
Enter String, integer and floating number respectively programming : 2

3.2
Enter string, integer and floating Number respectively : structure
6

2.3.
Displaying Information
programming 2 3.20
structure 6 2.30

Structure	Arrays
(1) Different Datatype	(1) Same Data type
(2) Dynamic Memory Allocation	(2) Static Memory Allocation
(3) Structure is not a pointer	(3) Array is a base pointer. It points to a particular Memory location.
(4) It uses the dot (.) operator to Access the structure members.	(4) It uses the subscript to Access the Array Elements.

(2) Union:-

Union are Derived Data types they are Declared like Structure the Difference between Union and Structure is in terms of storage like structures, Union also declared by using the keyword Union.

characteristic of Union:-

⇒ size allocated is equal to the largest data member of the Union.

→ Only one Union member can be accessed at a time.

→ collection of variables of Different Datatypes.

→ The keyword "union" is used to declare a Union.

Syntax:-

```
Union union_name
{
Union member1;
Union member2;
.....
Union membern;
};
Union union_variable;
```

Let us compare & Discuss the following structure

```
struct name
{
int S.no;
char na[5];
};
Struct name P;
```

```
Union name
{
int sno;
char na[5];
};
Union name P;
```

Example Program: Employee Details using Union:-

```
#include <stdio.h>
#include <conio.h>
union Employee
{
char name[10]; int id no;
float salary;
}e;
main()
{
printf("Enter the name\n");
scanf("%s", &e.name);
printf("Name : %s\n", e.name);
printf("Enter the id number\n");
scanf("%d", &e.idno);
printf("ID Number : %d\n", e.idno);
printf("Enter the salary\n");
scanf("%f", &e.salary);
printf("salary : %f\n", e.salary);
}
```

output:-

```
Enter the name
priyanga
Name : priyanga
Enter the id number
100
ID Number : 100
Enter the salary
20000
```

Comparison of Union and Structures:

S.No	Union	structure
1.	Union allocate the memory equal to the memory required by the number of the union.	structure Allocate the memory equal to the total memory required by the members.

2.	The Union, one block is used by all the member of the Union	In case of structure each member have their own memory space
3	The members of the union cannot be manipulated simultaneously	The members of the structure can be manipulated simultaneously

Example: employe details:-

```
#include <stdio.h>
Union job
{
char name[32]; float salary; int worker_no;
} u;
int main ()
{
printf ("Enter the name: \n");
scanf ("%s", su.name);
printf ("Enter salary: \n");
scanf ("%f", su.salary);
printf ("Enter salary: \n");
printf ("Displaying \n Name: %s\n", u.name);
printf ("salary: %f", u.salary);
return 0;
}
```

Output:

Enter Name:

Priya

Enter salary

20,000

Displaying

Name: Priya

Salary: 20000

3. Enumerated Data type

The ANSII provides Userdefined Datatype which is called Enumerated Datatype. i.e, the programmer can create their own Datatype and define what values the variables of these Data-types can hold.

Syntax:

keyword → enum tagname ← Name of the Enumeration

```
{
enum 1;
enum 2;
:
enum n;
}
```
← list of Identifier

```
};
```

example:

```
enum operator
{
plus;
minus;
multiplication;
division;
};
enum operation opr1, opr2;
opr 1 = plus;
opr 2 = division;
```

By Default the values in the enum are Assigned from 0. In the above mentioned example plus = 0, minus = 1 and so on.

Example program:-

```
#include <stdio.h>
#include <conio.h>
void main()
{
enum week = { dum = 1, Mon, Tues, Wed, Thur, Fri, Sat };
clscr();
printf("Mon = %d \n", Mon);
printf("Tue = %d \n", Tue);
printf("Wed = %d \n", wed);
```

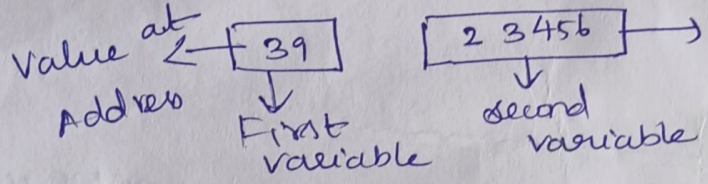
```
Printf("sat = %d\n", sat);
getch();
}
```

Op Mon = 2
Tue = 3
Wed = 4
Sat 7.

4) Pointer : pointers to variables

A Pointer is a variable, it may contain the memory address of Another variable. Pointer Can have Any name that is legal for other variable. It is declared in the same manner like other variable. It is Always denoted by '*' operator. A pointer is a variable whose value is also an Address. ~~of P~~

two Attributes: Address & values:



Features of Pointers:-

→ Pointers are Used for saving memory space.

→ Pointers are Efficient in handling Data and Associated with Array.

→ pointer Reduce length and complexity of the program.

Advantages of Using pointers:

→ pointers are more compact and Efficient code.

→ pointers can be used to Achieve clarity and simplicity

→ pointer Enables us to Access the Memory Directly

pointer Declaration:

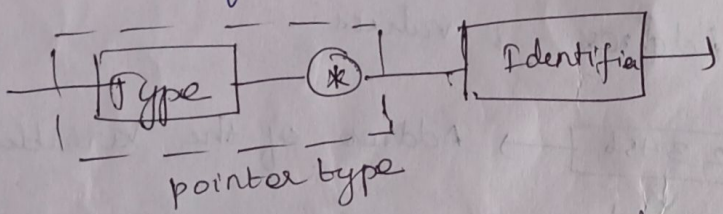
pointer variable that contains the Address of Another Variable like Normal variable (direct address of memory location.

Syntax: data_type *pointer_name;

Description: data_type → specifies the type of Data to which the pointer points.

pointer_name → specifies the name of pointer

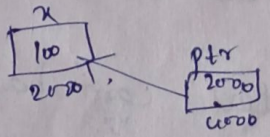
Example:-
```
int *a;
char *b;
float *c;
```



Accessing Variable through pointers:

To access address of a variable to a pointer, we use unary operator '&' that returns the address of that variable.

```
#include <stdio.h>
int main()
{
int x, y; //declare variables
int *ptr; // pointer initialization
x=100;
ptr = &x
y = *ptr;
printf("value of x is %d\n", x);
printf("%d is the address of %u\n", *ptr, ptr);
```



```
printf("%d is the addres of %u\n", ptr, &ptr);
printf("%d is the addres of %u\n", y, &y);
return 0;
}
```

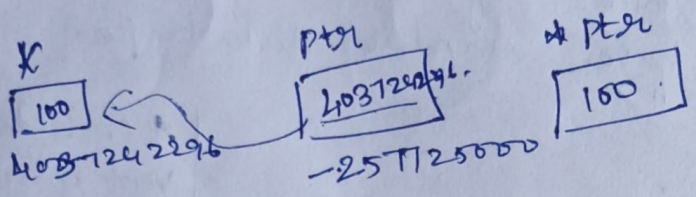
O/P

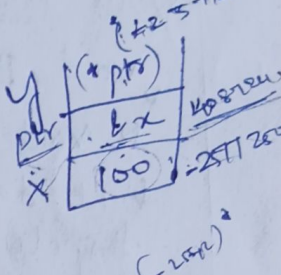
The value of x is 100

100 is the addres of 4037242296.

-2577250 is the addres of 4087242304

100 is the addres of 4037242280.



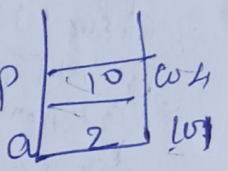



```
#include<stdio.h>
main(){
int a; a=2; int *p;
```

pointer variable stores the address of another variable

```
p=&a; //p=101
printf("a=%d\n", a); //a=2
printf("&a=%d\n", &a) //&a=101
printf("p=%d\n", p); //p=101
printf("*p=%d\n", *p); //*p=*(&a)=*(101)=2.
```

Returns value



y

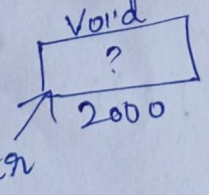
Pointer to Arrays and functions

Types of pointer:

1) ~~Void~~ Null pointer:

void point can point to any type of data. The type of data inside the block can be char, int, float (or) any other type.

```
Void *ptr;
```




```
Void *Name of pointer
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
int *p=NULL; // Null pointer is used when address doesn't have any address
clrscr();
printf("\n Null Pointer" , *p)
getch();
}
```

O/P

Null pointer

2) Void Pointer:

* Void pointer is also called generic pointer.
* Void Pointer = void keyword = Hold address of any variable
* It does not have standard data type.

```
#include<stdio.h>
Void main()
{
int i=2;
Void *p=&i;
printf("\n value of p : %d", *(int *)p);
getch();
}
```

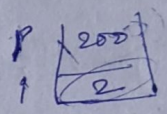
(Type conversion void → int)

O/P value of P: 2

```
#include<stdio.h>
void main()
{
Void *p=NULL;
clrscr();
printf("\n Memory size:%d", sizeof(p));
getch();
}
```

O/P

Memory size = 2



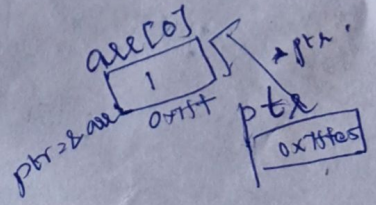
Pointer to an Array:

data_type (*var_name) [size_of_array];

```
#include <stdio.h>
int main()
{
int arr[5] = {1, 2, 3, 4, 5};
int *ptr=&arr;
printf("%p\n", ptr);
return 0;
}
```

Output

0x7ffe5cee5700



pointers to an array point the address of memory block of an array variable.

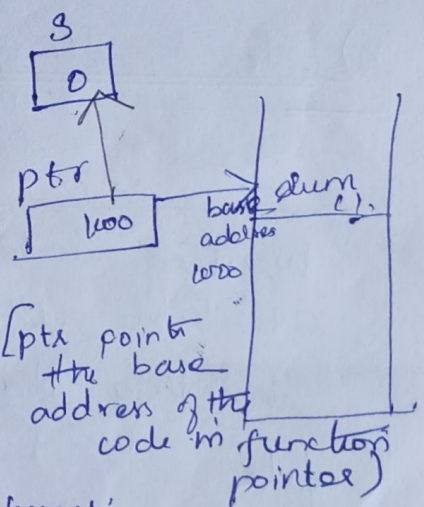
pointer to function

A pointer to a function points the address of the executable code of the function

```
return_type of function (* pointer_name)(Datatype of arguments)
```

```
#include<stdio.h>
int sum(int, int);
void main()
{
int s=0;
int (*ptr)(int, int)=&sum;
s=(*ptr)(2,3);
printf("sum=%d\n", s);
}
int sum(int a, int b)
{
return a+b;
}
```

o/p
sum = 3.



[ptr point the base address of the code in function pointer]

Whereas:
ptr points the address of the another variable

Application of function pointer

callback function:-

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
#include <stdio.h>
void sum(int a, int b)
{ printf("sum = %d\n", a+b);
```

```
void sum(int a,int b)
{
printf("sum=%d\n",a-b);
}
void display(void (*fptr)(int,int))
{
fptr(5,1);
}
void main()
{
display(sum);
display(sub);
}
```

O/P
sum = 6
sub = 4

5) File handling in C:

Description of function	function in Use
Writing data into an available file	fopen()
Writing data into an available file	fprintf()
Reading the data available in a file	fscanf()
writing any character into the program file	fputc()
used to close the program file	fclose()
used to set the file pointer to the intended file position	fseek()
Writing an integer into an available file	fputw()
used to read an integer from the given file	fgetw()

a+	open a file for both appending and reading the content
ab+	open a file for both appending and reading the content in binary mode.

fclose

once we write/read a file in a program, we need to close it (for both binary and text files). To close the files we utilise the fclose() function in a program.

```
fclose(fptr);
```

fptr - file pointer that is associated with that file that needs to be closed in a program

C Fprintf() and fscanf()

Writing File: fprintf() function

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

syntax:

```
int fprintf(FILE *stream, const char *format [, argument... ]
```

example:

```
#include <stdio.h>
main(){
FILE *fp;
fp = fopen("file.txt", "w"); // opening the file
fprintf(fp, "Hello file by fprintf...\n"); // writing data into file
fclose(fp); // closing file
}
```

Reading File: fscanf() function:

The fScanf() function is used to read set of characters from file. It reads a word from the file returns EOF at the end of file

Syntax:

1. int fscanf (FILE *stream, const char *format [, argument, ...])

example:

```
#include <stdio.h>
main() {
FILE *fp;
char buff[255]; // creating char array to store data to file
fp= fopen ("file.txt", "r");
while (fscanf (fp, "%s", buff) != EOF) {
printf ("%s", buff);
}
fclose (fp);
}
```

output:

Hello file by fprintf

fputc() and fgetc()

Writing File: fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

1. int fputc (int c, FILE *stream)

example:

```
#include <stdio.h>
main() {
FILE *fp;
fp= fopen ("file1.txt", "w"); // opening file
fputc ('a', fp); // writing single character into file
fclose (fp); // closing file
}
```

file1.txt

a

Reading File: fgetc() function :

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

1. Int fgetc(FILE *stream)

Example:

```
#include <stdio.h>
#include <conio.h>
Void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

myfile.txt

This is simple text message

C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fputs() functions.

Writing File: fputs() function

The fputs() function writes a line of characters into file. It writes string to a stream.

Syntax:

~~1. Int fputs() function writes a line of characters into file. It outputs string to a stream.~~

1. int fputs(const char *s, FILE *stream)

Example:

```
#include <stdio.h>
#include <conio.h>
Void main() {
FILE *fp;
clrscr();
fp= fopen ("myfile2.txt","w");
fputs ("hello c progranming",fp);
fclose(fp);
getch();
}
```

myfile2.txt

hello c progranming

Reading File: fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax

```
int fgets(const char *s, FILE *stream)
```

Example:

```
#include <stdio.h>
#include <conio.h>
void main() {
FILE *fp;
char text[300];
clrscr();
fp= fopen("myfile2.txt","r");
printf("%s", fgets(text,200,fp));
fclose(fp);
getch();
}
```

output

hello c programming

fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to read the data into file at desired location.

Syntax:

1. int fseek(FILE *stream, long int offset, int whence)

There are 3 constants used in the fseek() for whence:
SEEK_SET, SEEK_CUR and SEEK_END.

example:

```
#include <stdio.h>
void main(){
FILE *fp;
fp = fopen("myfile.txt","w+");
fputs("This is javatpoint", fp);
fseek(fp, 7, SEEK_SET);
fputs("sonoo jaiswal", fp);
fclose(fp);
}
```

myfile.txt

```
This is sonoo
      jaiswal
```

C rewind() function:

The rewind() function sets the file pointer at the beginning of the stream. It is useful to have to use stream many times.

Syntax:

1. void rewind(FILE *stream)

example:

```
#include <stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp = fopen("file.txt", "r");
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
rewind(fp); // moves the file pointer at beginning of the file
while((c=fgetc(fp))!=EOF){
```

```
printf("%c", c);
}
fclose(fp);
getch();
}
```

output:

```
This is a simple text this is a
simple text
```

C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. we can use SEEK_END constant to move the file pointer at the end of file.

syntax:

```
long int ftell(FILE *stream)
```

example:

```
#include <stdio.h>
#include <conio.h>
void main(){
    FILE *fp;
    int length;
    clrscr();
    fp=fopen("file.txt","r");
    fseek(fp,0,SEEK_END);
    length=ftell(fp);
    fclose(fp);
    printf("size of file: %d bytes",length);
    getch();
}
```

output:

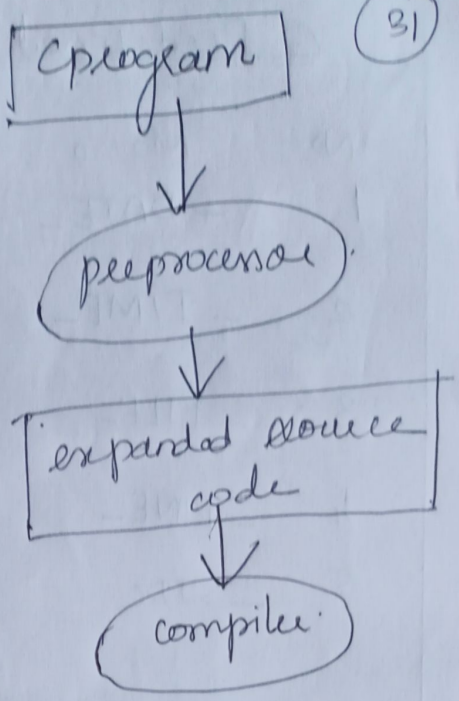
```
size of file: 21 bytes
```

6) C preprocessor Directives:

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro processor because it allows us to add macros.

The All preprocessor directives starts with hash (#) symbol.

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma



C Macros:

A macro is a segment of code which is replaced by the value of Macro. Macro is defined by #define directive.

There are two types they are:

1. Object-like Macros
2. Function-like Macros.

Object-like Macros:

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. for example.

1. #define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function like macro looks like function call.

ex: #define MIN(a,b) ((a)>(b)? (a) : (b))

Here, MIN is the macro name

C Predefined Macros:

No	Macro	Description
1	__DATE__	represents current date in "MMM DD YYYY" format
2	__TIME__	represents current time "HH:MM:SS" format
3	__FILE__	represents current file name
4	__LINE__	represents current line number
5	__STDC	It is defined as 1 when compiler compiles with the ANSI standard.

C predefined macros example

File: simple.c

```
#include <stdio.h>
int main() {
printf("File: %s\n", __FILE__);
printf("Date: %s\n", __DATE__);
printf("Time: %s\n", __TIME__);
printf("Line: %d\n", __LINE__);
printf("STDC: %d\n", __STDC__
return 0;
}
```

output

```
File: simple.c
Date: Dec 6 2015
Time: 12:28:46
Line: 6
STDC: 1
```

1. C #include

#include preprocessor is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

There are two variants to use #include directive.

1. #include <file name>
2. #include "file name"

The #include<filename> tells the compiler to look for the directory where system header files are held. In UNIX it is "/usr/include directory".

The #include "filename" tells the compiler to look in the current directory from where program is running.

example
```
#include <stdio.h> // preprocessor directive
int main () {
printf("Hello. C");
return 0;
}
```

output

Hello C

2) C #define:

The #define preprocessor directive is used to define constant (or) micro substitution. It can use any basic data type.

syntax:

#define token value

example:

```
#include <stdio.h>
#define PI 3.14
main () {
printf("%f", PI)
}
```

output:

~~Maximum between 10 and 20 is 10~~
3.140000

3. C #undef:

The #undef preprocessor directive is used to undefine the constant (or) macro defined by #define.

Syntax:

1. #undef token

example:

```
#include <stdio.h>
#define PI 3.14
#undef PI
main() {
printf("%f", PI);
}
```

output.

Compile time Error: PI is undeclared

Used for reading the current position of a file	ftell()
Sets an intended file pointer to the file beginning itself	rewind()

Operations Done in File Handling

The process of File Handling enables a user to update, create, open, read, write, and ultimately delete the file/content in the file that exists on the C program's local file system.

primary operations that you can perform on a file in C program:

1. opening a File that already exists
2. Creating a new File.
3. Reading content/data from the existing file.
4. Writing more data into the file.
5. Deleting the data in the file (or) the file altogether.

opening a File in the program - to create and edit data

We open a file with the help of the fopen() functio that is defined in header file - stdio.h.

Syntax for opening file

```
ptr = fopen("openfile", "openingmode");
```

example:

fopen("E:\\myprogram\\recentprogram.txt", "w");

(fopen) creates a new file with name E:\\myprogram location.

"w" refers to writing mode. It allows a programmer to overwrite/edit and create the contents in a program file

opening Modes of C in Standard I/O of a Program

Mode in program	Meaning in Mode.
r	- open a file for reading the content.
rb	- open a file for reading the content in binary mode.
w	- open a file for writing the content.
wb	- open a file for writing the content in binary mode.
a	- open a file for appending the content. Meaning, the data of the program is added to the file's end in a program.
ab	- open a file for appending the content in binary mode. Meaning, the data of the program is added to the file's end in a program in a binary mode.
r+	- open a file for both writing and reading the content
rb+	- open a file for both writing and reading the content in binary mode.
w+	- open a file for both writing and reading
wb+	- open a file for both writing and reading the content in binary mode

4) C #ifdef

The #ifdef preprocessor directive checks if macros is defined by #define. If yes, It executes the code other wise #else code is executed, if present.

Syntax:-

```
#ifdef MACRO
// code
#endif
```

Syntax with #else:-

```
#ifdef MACRO
// successful code
#else
// else code
#end if.
```

C #ifdef example:

```
#include <stdio.h>
#include <conio.h>
#define NOINPUT
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
{printf("Enter a:");
scanf("%d", &a);
#end if
printf("value of a : %d\n", a);
getch();
}
```

Output

Value of a: 2

5. C #ifndef:

The #ifndef preprocessor directive checks if macros is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
// code
#endif
```

Syntax with #else:

```
#ifndef MACRO
// Successful Code
#else
// else code
#endif
```

c) #ifndef example:

```
#include<stdio.h>
#include<conio.h>
#define INPUT
Void main(){
int a=0;
#ifndef INPUT
 a=2;
 #else
 printf("Enter a:");
 scanf("%d",&a);
#endif
 printf
 getch();
 }
```

output

```
Enter a:5
Value of a:5
```

b) c #if:

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #else if or #else (or) #endif code is executed.

Syntax:

```
#if expression
 // code
#endif
```

Syntax with #else

```
#if expression
 // if code
#else
 // else code
#endif
```

```
#include<stdio.h>
#include<conio.h>
#define NUMBER 0
Void main(){
#if (NUMBER==0)
 Printf("Value of Number is:%d", NUMBER);
#endif
 getch();
}
```

output:

```
Value of Number is: 0
```

7) C #else

The #else preprocessor directive evaluates the expression or condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax

```
#if expression
  // if code
#else
  // else code
#endif
```

Syntax with #elif

```
#if expression
  // if code
#elif expression
  // elif code
#else
  // else code
#end if
```

example

```
#include<stdio.h>
#include<conio.h>
#define NUMBER 1
void main() {
#if NUMBER==0
printf("Value of Number is : %d",NUMBER);
#else
print("Value of Number is non-zero");
#endif
getch();
}
```

output

Value of Number is non-zero

(8) C #error

The #error preprocessor directive indicate error. The compiler gives fatal error if #error directive is found and skips further compilation process.

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
  float a;
  a=sqrt(7);
  printf("%f",a);
#endif
```

output

2.645751

9). C #pragma:

⇒ The #pragma preprocessor directive is used to provide additional Information to the compiler.

⇒ The #pragma directive is used by the compiler to offer machine (or) operating system feature.

Syntax:

1). #pragma token

```
#include <stdio.h>
#include <conio.h>
void func();
#pragma startup func
#pragma exit func
void main(){
printf("\n I am in main");
getch();
}
void func(){
printf("\n I am in func");
getch();
}
```

output

I am in func
I am in main
I am in func

Unit-II Linear Data Structures

Abstract Data types (ADTs) - List ADT - Array Based Implementation - Linked list - Doubly linked lists - Circular Linked List - stack ADT - Implementation of Stack - Applications - Queue ADT - Priority Queues - Queue Implementation - Applications.

1) Abstract Datatypes (ADTs)

An Abstract Datatype (ADT) is defined as a mathematical model with a collection of operations defined in the model

⇒ set of integers, together with the operations of union, intersection and set difference form an example of an ADT.

⇒ An ADT consists of data together with functions that operate on that data

Advantages of ADT

1. Modularity
2. Reuse
3. code is easier to understand

2) The List ADT

* List is an ordered set of elements

The general form of the list is $A_1, A_2, \cdots, A_N$

A_1 - First element of the list

A_2 - 2nd element of the list.

N - size of the list

If the element at position i is A_i, then its successor is A_{i+1} and its predecessor is A_{i-1}

Various operations performed on List:

1. Insert(X, 5) – Insert the element X after the position 5.
2. Delete(X) – The element X is deleted
3. Find(X) – Returns the position of X
4. Next(i) – Returns the position of its successor element i+1
5. Previous(i) – Returns the position of its predecessor i-1.
6. Print list – Contents of the list is displayed.
7. Make empty – Makes the list empty.

Implementation of List ADT

1. Array based Implementation
2. Linked List based Implementation.

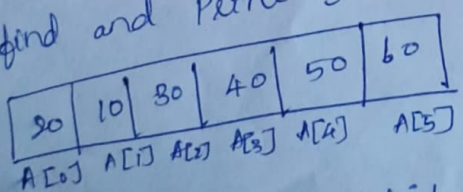
1) Array Implementation of List:

⇒ Array is a collection of a specific number of same type of data stored in consecutive memory locations

⇒ Array is a static data structure i.e, the memory should be allocated in advance and the size is fixed.

⇒ Insertion and Deletion operation are expensive as it requires more data movements.

⇒ find and Print list operations takes constant time.



The basic operations performed on a list of elements are

a. Creation of List
b. Insertion of data in the List
c. Deletion of data from the List
d. Display all data's in the List
e. Searching for a data in the List

Declaration of Array

```
#define maxsize 10
int list[maxsize], n;
```

Create operation

Create operation is used to create the list with 'n' number of elements. If array's maxsize then elements cannot be inserted into the list. otherwise the array elements stored in the consecutive array locations (i.e) list[0], list[1] & so on.

```
Void create()
{
int i;
printf("\n Enter the number of elements to be added in the list: \t");
scanf("%d", &n);
printf("\n Enter the array elements: \t");
for(i=0; i<n; i++)
scanf("%d", &list[i]);
}
```

if n=6, the output of creation is as follows: list[6]

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Insert operation

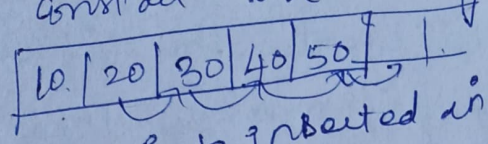
Insert operation is used to insert an element at particular position in the existing list. Inserting the element in the last position of an array is easy. But inserting the element at a particular position in an array is quite difficult.

Routine to insert an element in the array:-

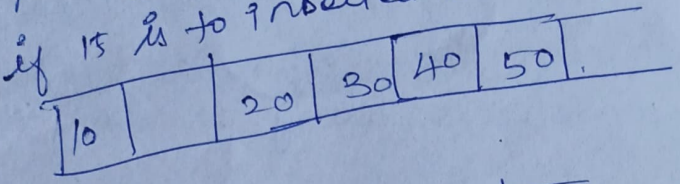
```
Void insert()
{
int i, data, pos;
printf("\n Enter the data to be inserted: \t");
scanf("%d", &data);
printf("Array overflow");
for(i=n-1; i>=pos-1; i--)
list[i+1] = list[i];
list[pos-1] = data;
```

```
n=n+1;
Display(); }
```

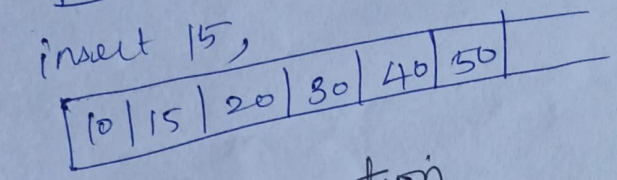
Consider an array with 5 elements [max elements = 10]



if 15 is to inserted in 2nd position.



insert 15,



Deletion Operation

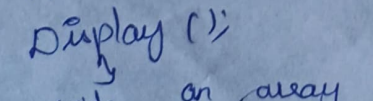
⇒ Deletion is the process of removing an element from the array at any position.

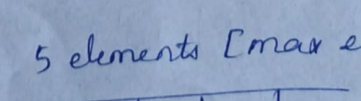
⇒ Deleting an element from the end is easy. If an element is to be deleted from any particular position, it requires all subsequent element from that position is shifted one position toward left

Routine to delete an element in the array:

```
void Delete()
{
int i, pos;
printf("\n Enter the position of the data to be deleted: ");
scanf("%d", &pos);
printf("\n The data deleted is: %d", list[pos-1]);
for(i=pos-1; i<n-1; i++)
  list[i] = list[i+1];
n=n-1;
Display();
```

Consider an array with 5 elements [max elements = 10]





Display operation

Traversal is the process of visiting the elements in a array. Display() operation is used to display all the elements stored in the list. The elements are stored from the index 0 to n-1. Using a loops, the elements are viewed in list.

Routine to traverse/display elements of the array:

```
void display()
{
int i;
printf("\n ***** elements in the array *****\n");
for(i=0; i<n; i++)
  printf("%d\t", list[i]);
}
```

Search operation

Search() operation is used to determine whether a particular element is present in the list or not. Input the search element to be checked in the list

Routine to search an element in the array:

```
void search()
{
int search, i, count=0;
printf("\n Enter the element to be searched:\t");
scanf("%d", &search);
for(i=0; i<n; i++)
{
if (search == list[i])
  count++;
}
if (count==0)
printf("\n Element not present in the list");
else
printf("\n Element present in the list");
}
```

Advantages :

1. The elements are faster to access using random access
2. Searching an element is easier.

Limitations of array implementation

⇒ An array store its node in consecutive memory location.
⇒ Insertion and deletion operation in array are expensive
⇒ The number of elements in array is fixed & it is not possible to change the number of elements

Applications of arrays:

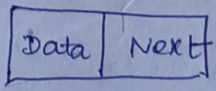
Arrays are particularly used in progams that require storing large collections of similar type of data

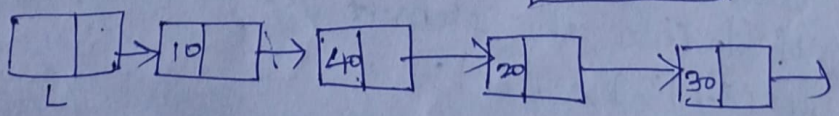
---

4) Linked list Implementation

Linked List :-

A linked list is an ordered collection of elements. Each element in the list is referred as a node. Each node contains two fields namely.





Advantages of Linked List :-

1. Insertion and deletion of elements can be done efficiently
2. It uses dynamic memory allocation,
3. memory utilization is efficient compared to arrays.

Disadvantages:

1. Linked list does not support random access,
2. Memory is required to store next field.
3. Searching takes time compared to arrays.

Types of Linked list:-

1. Singly linked list (or) one way list.
2. Doubly linked list (or) Two way list.
3. Circular linked list.

Dynamic allocation:

The process of allocating memory to the variables during execution of the program (or) at run time is known as dynamic memory allocation. Dynamic memory allocation give best performance in situation in which we do not know memory requirements in advance.

Memory allocation/deallocation functions

function	Task
malloc()	Allocates memory and returns a pointer the the first byte of allocation.
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory.
free()	frees previously allocated memory
realloc()	Alters the size of previously allocated memory

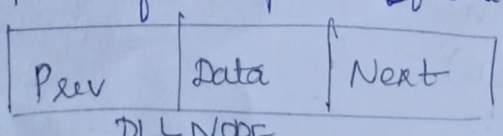
5) Doubly linked list:

A doubly linked list is a linked list in which each node has three fields namely Data, Next, Prev.

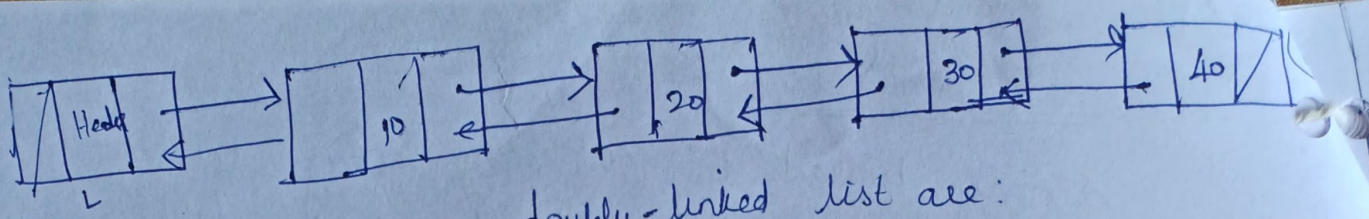
Data – This field stores the value of the element

Next – This field points to the successor node in the list

prev – The field points to the predecessor node in the list



DLL NODE

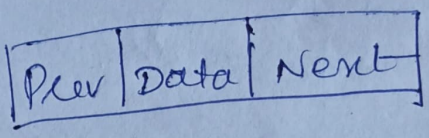


Basic operations of a doubly-linked list are:

1. Insert – Inserts a new element at the end of the list
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list

Declaration of DLL Node

```
typedef struct node *position;
struct node
{
 int data;
 position prev;
 position next;
};
```



Creation of list in DLL

Initially the list is empty, Then assign the first node as head.

```
newnode->data = X;
newnode->next = NULL;
newnode->prev = NULL;
L = newnode;
```

If we add one more node in the list, then create a newnode and attach that node to the end of the list.

```
L->next = newnode;
newnode->prev = L;
```

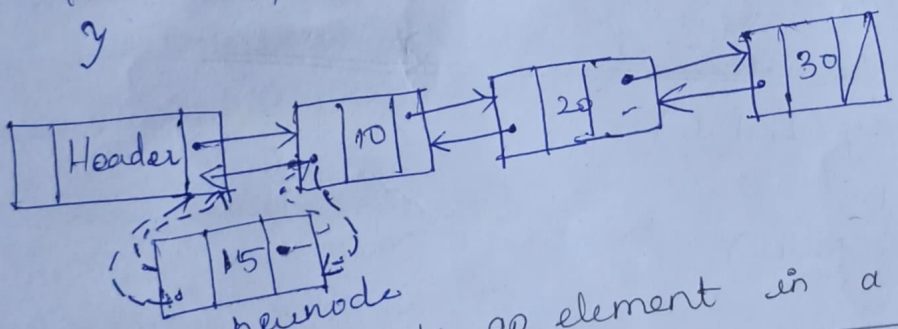
Routine to insert an element in a DLL at the beginning

```
void insert (int x, list L, position P) {
struct node *Newnode;
if (pos==1)
 P=L;
Newnode = (struct node*) malloc (sizeof (struct node));
if (Newnode != NULL)
Newnode->data = x;
```

```
Newnode = (struct node*) malloc (sizeof (struct node));
if (Newnode != NULL)
Newnode → data = x;
Newnode → Next = L → next;
L → next → prev = Newnode;
L → next = Newnode;
Newnode → prev = L;
}
```

Routine to insert an element in an DLL any position:

```
Void Insert (int x, list L, position P)
{
struct node *Newnode;
Newnode = (struct node*) malloc (sizeof (struct node));
if (Newnode != NULL)
Newnode → data = x;
Newnode → next = P → next;
P → next → prev = Newnode
P → next = Newnode;
Newnode → prev = P;
}
```



Routine to insert an element in a DLL at the end.

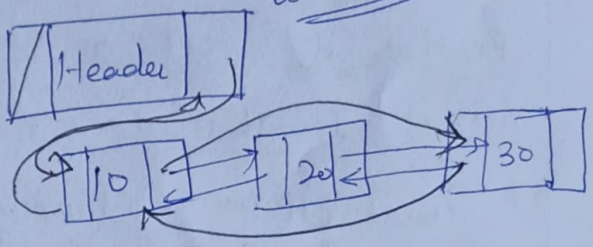
```
Void insert (int x, list L, position P)
{
P = L;
newnode = (struct node*) malloc (sizeof (struct node));
printf ("In Enter the data to be inserted In");
scanf ("%d", &newnode → data);
while (P → next != NULL)
P = P → next;
newnode → next = NULL;
P → next = newnode;
```

```
        newnode->prev = p
        }
```

Routine for deleting an element:

```
void Delete (int x, List L)
{
Position p, temp;
P = Find (x, L);
if (P == L->next)
 temp = L;
 L->next = temp->next;
 temp->next->prev = L;
 free(temp);
 else if(IsLast(P, L))
   {
   temp = P;
   P->prev->next = NULL;
   free (temp);
   }
 else {
   temp = P;
   P->prev->next = p->next;
   p->next->prev = P->prev;
   free (temp);
```

delete 20



Routine to display the elements in the list:

```
void Display (List L)
{
 P = L->next;
 while (P != NULL)
 {
  printf ("%d", p->data);
  p = p->next;
  }
  printf ("NULL");
    }
```

Routine to search whether an element is present in the list

```
void find()
 {
 int a, flag=0, count=0;
 if (LL == NULL)
```

```
printf ("\n The list is empty");
else
{
printf ("\n Enter the element to be searched");
scanf ("%d", &a);
for (P=L; p!==NULL; p=p->next)
{
count ++;
if (P->data ==a)
{
flag=1
printf ("\n The element is found");
printf ("\n The position is %d", count);
break; }}
if (flag==0)
printf ("\n The element is not found");
}}
```

Advantages of DLL:

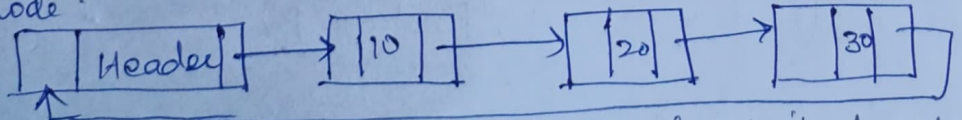
The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL only one link field is there which stores next node which makes accessing of any node difficult

Disadvantages of DLL:

The DLL has two pointer fields. One field is prev link and another is next link field. Because of these two pointer fields, more memory space is used by DLL compared to SLL.

6) Circular linked list

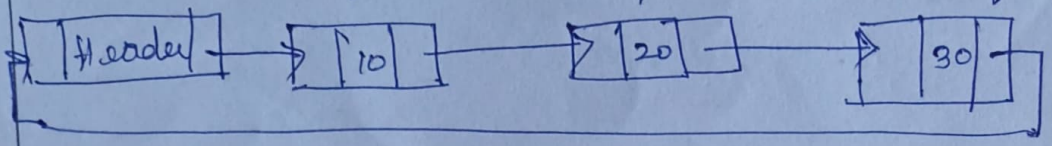
Circular linked list is a linked list in which the pointer of the last node points to the first node.



Types of CLL: ⇒ Circular singly linked list
⇒ Circular doubly linked list

Singly linked circular list :-

A singly linked circular list is a linked list in which the last node of the list points to the first node.

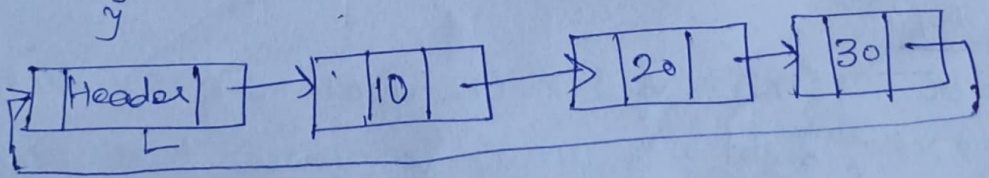


Declaration of node :-

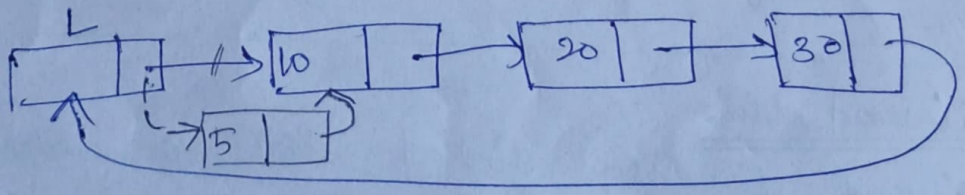
```
typedef struct node *position
struct node
{
int data;
position next;
};
```

Routine to insert an element in the beginning

```
void insert_beg(int X, List L)
{
position Newnode;
Newnode = (struct node*)malloc(sizeof(struct node));
if(Newnode != NULL)
{
Newnode -> data = X;
Newnode -> next = L -> next;
L -> next = Newnode;
}
}
```



insert (5, L)



Routine to insert an element in the middle.

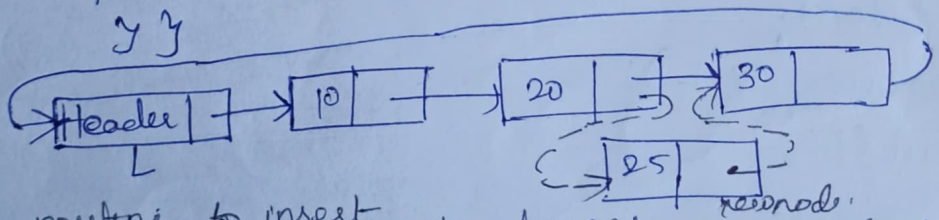
```
void insert_mid(int x, List L, Position p)
{
position Newnode;
Newnode = (struct node *) malloc (sizeof (struct node));
```

```
printf("\n The list is empty")
else
{
printf("\n Enter the elements to be searched");
scanf("%d",&a);
for(P=L; P!=NULL; P=P->next)
{
count++;
if(P->data==a)
{
flag=1;
Printf("\n The element is found");
Printf("\n The position is %d", count);
break;
}
}
if(flag==0)
Printf("\n The element is not found");
}
}
```

~~Advantages of DLL~~

~~The DLL has two pointer fields.~~

```
if(Newnode!=NULL)
{
Newnode->data=x;
Newnode->next=p->next;
p->next=Newnode;
}}
```

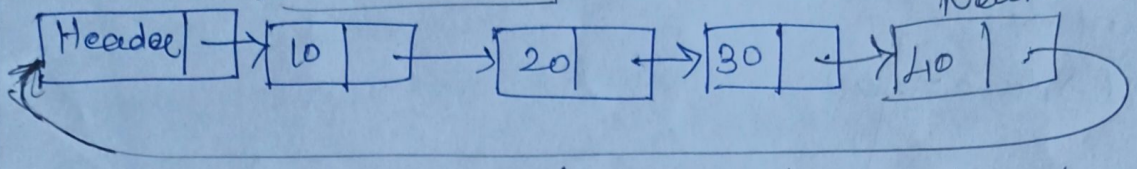


Routine to insert an element in the last.

```
Void Insert_last(int x, List L)
position Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
P=L;
```

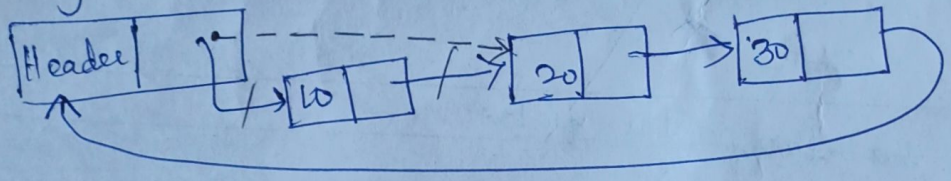
```
While (P->next != L)
P = P->next;
Newnode->data = X;
P->next = Newnode;
Newnode->next = L;
}
```

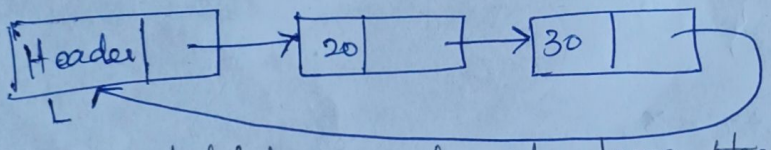
Insert (H0, L)



Routine to delete an element from the begining

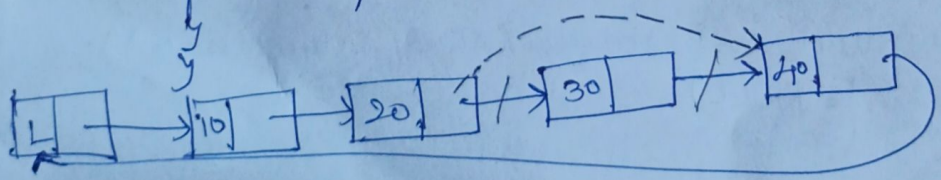
```
Void del_first (List L)
{
position temp;
temp = L->next;
L->next = temp->next;
free (temp);
}
```



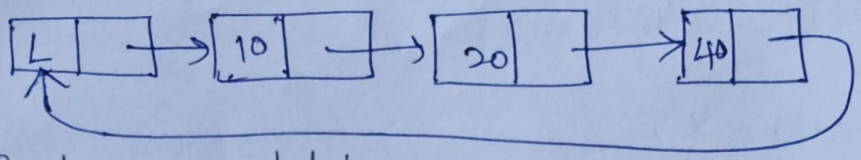


Routine to delete an element from the middle

```
void del_mid (int X, List L)
{
position P, temp;
P = findprevious (X, L);
if (!Islast (P, L))
{
temp = P->next;
P->next = temp->next;
free (temp);
}
}
```

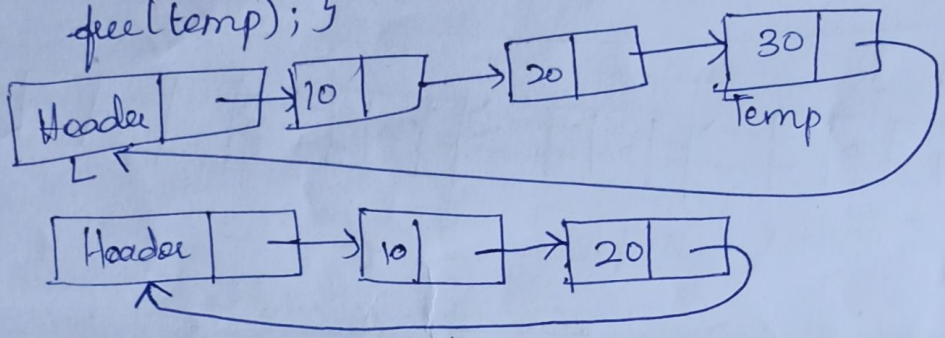


Delete(30,L)



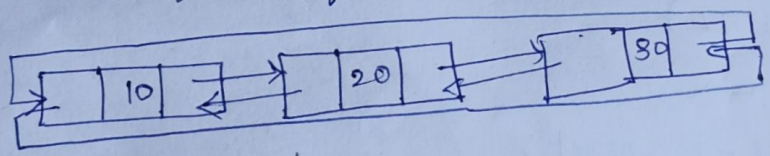
Routine to delete an element at the last position

```
void del_last(List L)
{
position p, temp;
p=L;
while (p->next->next!=L)
p=p->next;
p->next=L;
free(temp); }
```



Doubly linked circular list:

A doubly linked circular list is a doubly linked list in which the next link of last node points to the first node and prev link of the first node points to the last node of the list.



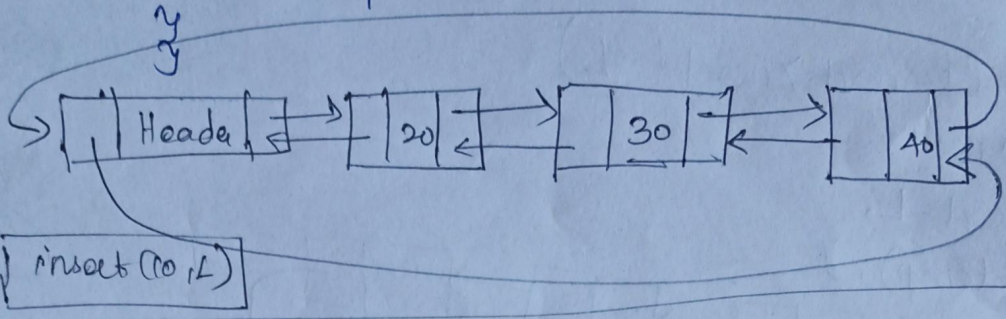
Declaration of node:

```
typedef struct node* position;
struct node
{
int data;
position next;
position prev;
};
```

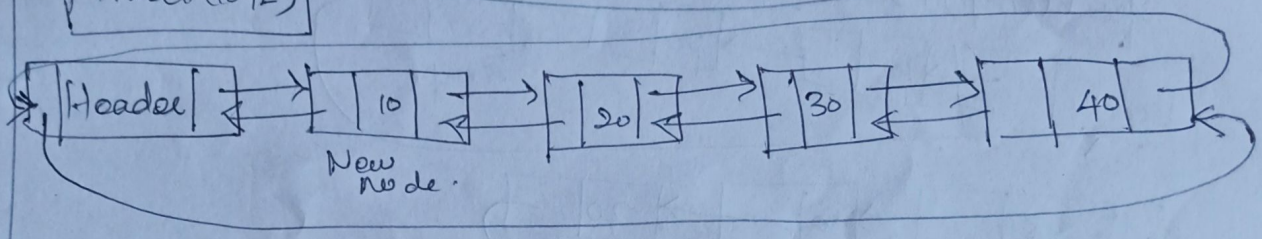
Routine to insert an element in the bagining

```
void insert_bag(int x, List L)
{
position Newnode;
```

```
Newnode = (struct node *)malloc (sizeof (struct node));
if (Newnode != NULL)
{
    Newnode->data = X;
    Newnode->next = L->next;
    L->next->prev = Newnode;
    L->next = Newnode;
    Newnode->prev = L;
}
```

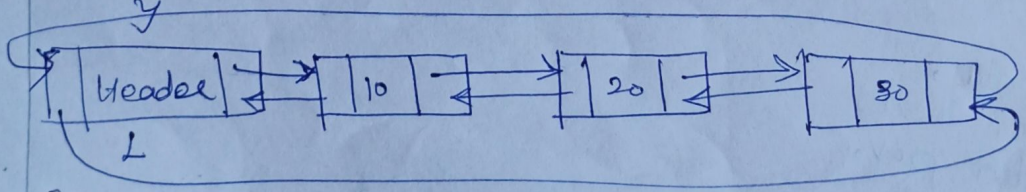


insert (10, L)

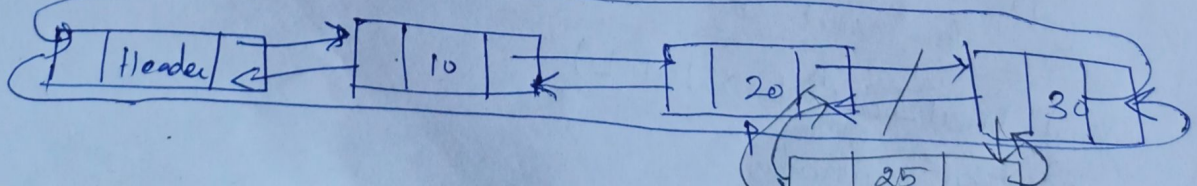


Routine to insertan element in the middle:

```
Void insert_mid (int x, List L, Position p)
{
position Newnode;
Newnode = (struct node*)malloc (sizeof (structnode));
if (Newnode != NULL)
{
Newnode->data = X;
Newnode->next = p->next;
p->next->prev = Newnode;
p->next = Newnode;
Newnode->prev = p;
}
}
```

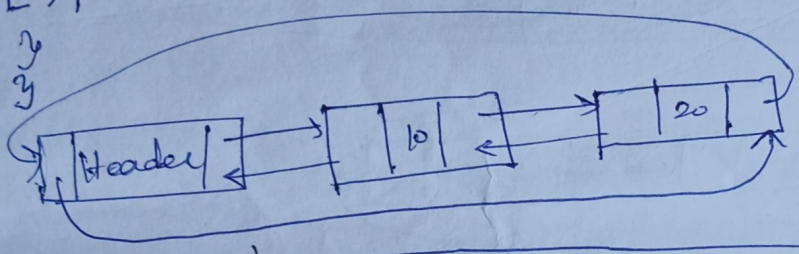


Insert (25, L, P)

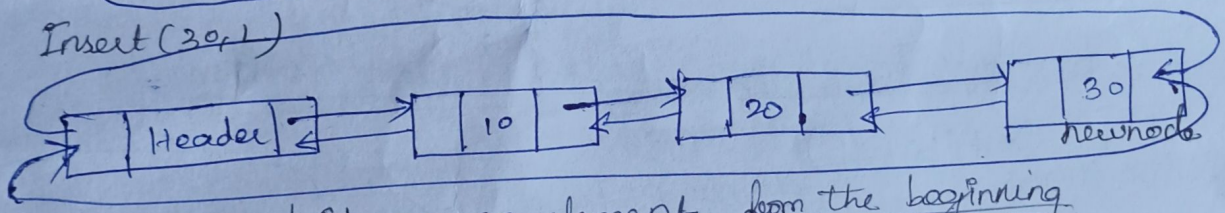


Routine to insert an element in the last

```
void insert_last (int x, list L)
{
position Newnode, p;
Newnode = (struct node*) malloc (sizeof (struct node));
if (Newnode != NULL)
{
p=L;
while (p->next != L)
p=p->next;
Newnode -> data = x;
p->next = Newnode;
Newnode->next = L;
Newnode -> prev = p;
L-> prev = newnode;
}
}
```

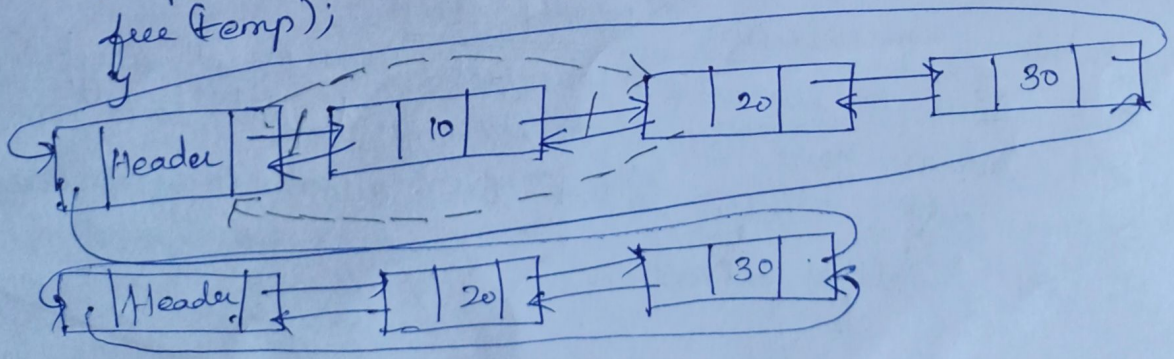


Insert (30, L)



Routine to delete an an element from the beginning

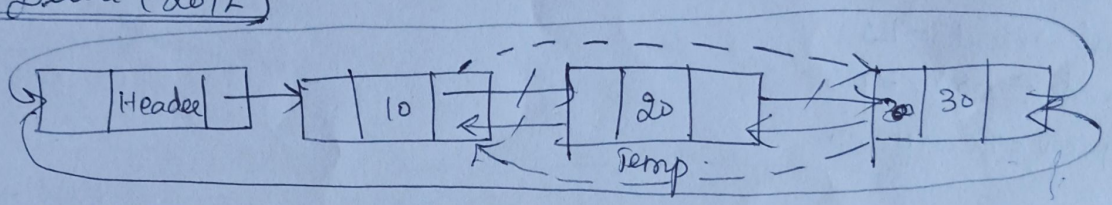
```
void del-first (List L)
{
position temp;
if (L->next != NULL)
{
temp = L->next;
L->next = temp->next;
temp->next->prev = L;
free (temp));
}
```

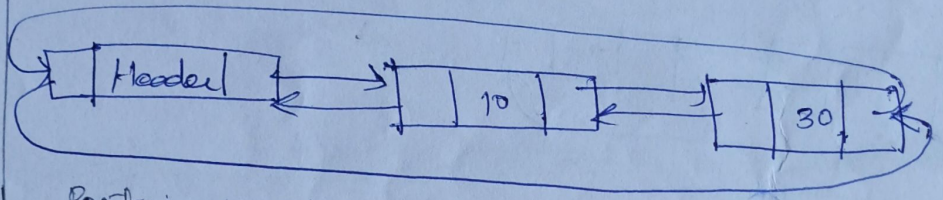


Routine to delete an element from the middle.

```
void del_mid (int X, List L)
{
position p, temp ;
p = FindPrevious (X);
if (!IsLast(p,L))
 {
temp = p->next;
p->next = temp->next;
 temp->next->prev = p.
  free(temp); } }
```

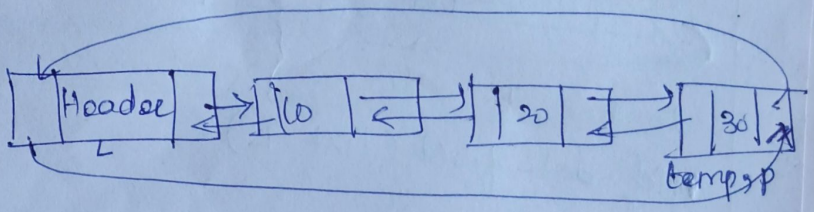
Delete (20, L)

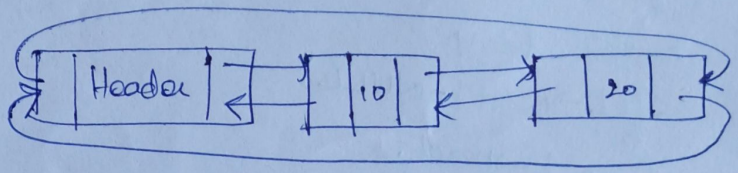




Routine to delete an element at the last position

```
void del_last (List L)
{
position p, temp;
p = L;
while (p->next != L)
 p = p->next;
  temp = p;
  p->next->prev = L;
   L->prev = p->prev;
  free (temp);
  }
```





Advantages of circular linked list

⇒ It allows to traverse the list starting at any point
⇒ It allows quick access to the first and last records
⇒ circularly doubly linked list allows to traverse the list in either direction.

7) Stack

Stack is Linear Data Structure that follows Last In First Out (LIFO) principle.

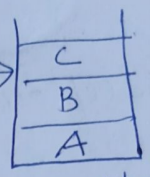
Insertion and deletion can be done at only one end of the stack called TOP of the stack.

example:- Pile of coins, stack of Tray

Stack ADT:

Top pointer:

* It will always point to the last element inserted in the stack.



stack model

* for empty stack, top will be pointing to -1. (TOP = -1)

Operation on Stack (Stack ADT):

Two fundamental operations performed on the stack are PUSH & POP.

(a) PUSH:

It is the process of inserting a new element at the Top of the stack.

for every push operation:

1. Check for full stack (overflow).
2. Increment Top by 1. (TOP = TOP+1).
3. Insert the element x in the Top of the stack.

(b) POP:

It is the process of deleting the Top element of the stack.

for every pop operation:

1. Check for empty stack (underflow).
2. Delete (pop) the Top element x from the stack.
3. Decrement the Top by 1. (TOP = TOP-1).

Exceptional conditions of stack.

1. stack overflow.

⇒ An Attempt to insert an element x when stack is full is said to be stack overflow.

⇒ For every Push operation, we need to check this condition.

Routine to delete an element

1. Stack overflow:

* An Attempt to insert an element X when the stack is full, is said to be stack overflow.
* For every push operation, we need to check this condition.

2. Stack underflow

* An attempt to delete an element when the stack is empty, is said to be stack underflow.
* for every pop operation, we need to check this condition

8) Implementation of stack.

stack can be implemented in 2ways.

1. Static Implementation (Array implementation of stack)
2. Dynamic Implementation (Linked list implementation of stack)

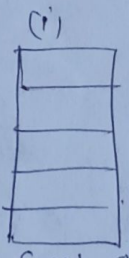
Array Implementation of stack.

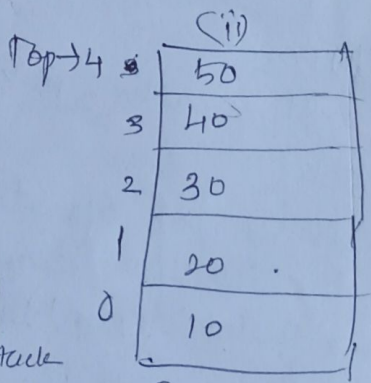
⇒ Each stack is associated with a Top pointer
⇒ for Empty stack, Top = -1.
⇒ stack is declared with its maximum size.

Array Declaration of stack:

```
#define ArraySize 5
int S[Array size];
(or)
int S[5];
```

(i) Stack Empty Operation





⇒ Initially stack is Empty.
⇒ with Empty stack Top pointer points to -1.
⇒ It is necessary to check for empty stack before deleting (Pop) an element from the stack.

(ii) Stack full operation

⇒ As we keep inserting the elements, the stack gets filled with the elements

⇒Hence it is necessary to check whether the stack is full or not before inserting a new element into the stack

(iii) Push operation :-

⇒ It is the process of inserting a new element at the Top of the stack.

⇒ It takes two parameters. Push(x,s) the element x to be inserted at the Top of the stack s.

⇒ Before inserting an Element onto the stack, check for full stack.

⇒ If the stack is already full, Insertion is not possible.

⇒ otherwise, Increment the Top pointer by 1 and then insert the element x at the Top of the stack.

(iv) Pop operation

⇒ It is the process of deleting the Top element of the stack.

⇒ It takes only one parameter. Pop(x). The element x to be deleted from the Top of the stack

⇒ Before deleting the Top element of the stack, check for empty stack.

⇒ If the stack is empty, deletion is not possible.

⇒ otherwise, delete the Top element from the stack and then decrement the Top pointer by 1.

(v) Return top Element :

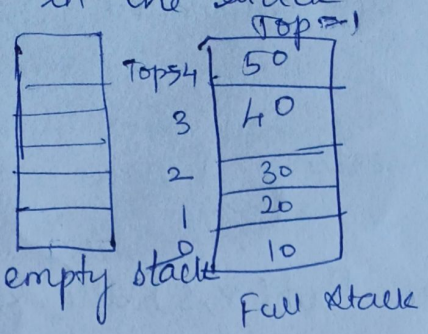
⇒ pop routine deletes the Top element in the stack.

⇒ If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack.

⇒ To do this, first check for Empty stack

⇒ If the stack is empty, then there is no element in the stack

⇒ Otherwise, return the element which is pointed by the Top pointer in the stack.

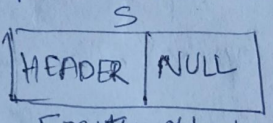


Linked List implementation of stack

* stack elements are implemented using SLL (singly Linked List) concept.
* Dynamically, memory is allocated to each element of the stack as a node.

(i) stack Empty operation:

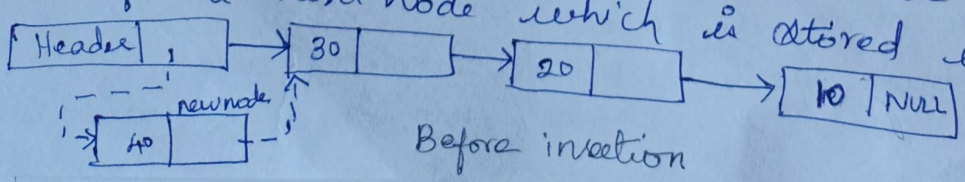
* Initially stack is Empty.
* With linked list Implementation, Empty stack is represented as S→next = NULL.
* It is necessary to check for Empty stack before deleting (pop) an element from the stack.



Empty stack.

(ii) Push Operation:

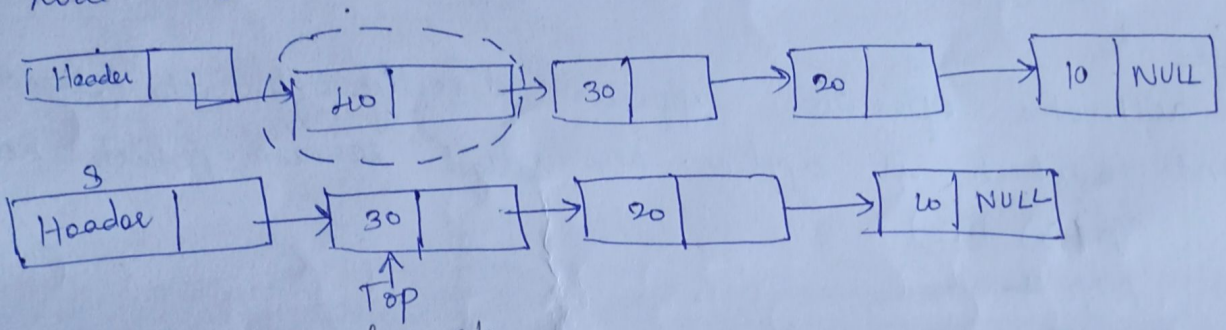
* It is the process of inserting a new element at the Top of the stack.
* With Linked List implementation, new element is always inserted at the front list (i.e.) S→next
* It takes two parameter. Push(x,s) the element x to be inserted at the Top of the stack s.
* Allocate the memory for the newnode to be inserted.
* Insert the element in the data field of the newnode
* Update the next field to the newnode with the address of the next node which is stored in the S→next



Before insertion

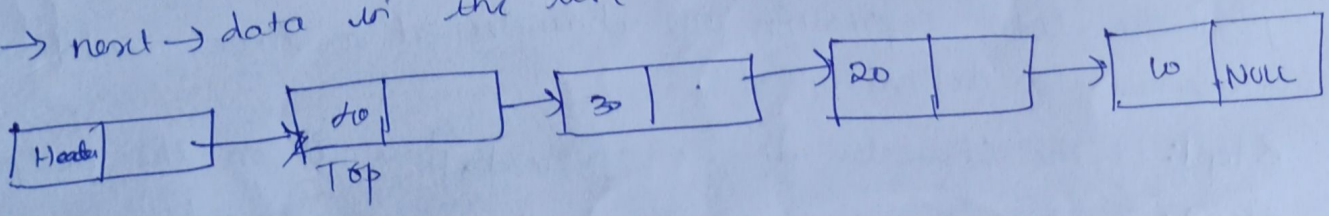
Pop operation:

⇒ It is the process of deleting the Top element of the stack
⇒ with Linked list implementation, the element at the front of the List (i.e) s→next is always deleted.
⇒ It takes only one parameter. pop(x). The element x to be deleted from the front of the list.
⇒ Before deleting the front element in the front element in the list, check for Empty stack.
⇒ If the stack is empty, deletion is not possible.
⇒ Otherwise, make the front element in the list as "temp".
⇒ update the next field of header.
⇒ Using free() function, Deallocate the memory allocated for temp node.



(iv) Return Top Element

⇒ POP routine deletes the front element in the list
⇒ If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack
⇒ TO do this, first check for empty stack.
⇒ If the stack is empty then there is no element in the stack.
⇒ otherwise, return the element present in the S→next→data in the list.



(v) Applications of stack:

1. Evaluating the arithmetic expressions.
   * conversion of Infix to Postfix Expression
   * Evaluating the Postfix Expression
2. Balancing the Symbols.

2. Balancing the symbols.
3. Function call
4. Tower of Hanoi
5. 8 Queen Problem

Evaluating the Arithmetic Expression

There are 3 types of Expressions
- Infix Expression
- Postfix Expression
- Prefix Expression

Infix:
The arithmetic operator appears between the two operands to which it is being applied.
A/B+C

postfix:
The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation.
(A/B)+C)
AB/C+

Prefix:
The arithmetic operator is placed before the two operands to which it applies. Also called polish notation.
((A/B)+C)
+/ABC

Evaluating Arithmetic Expressions

1. Convert the given infix expression to postfix expression.
2. Evaluate the postfix expression using stack.

Algorithm:
Read the infix expression one character at a time until it encounters the delimiter "#".

Step 1: If the character is an operand, place it on the output.

Step 2: If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3: If the character is left parathesis, push it onto the stack.

Step 4: If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Infix expression: A * B + (C - D / E) #

Read char	stack	output
A	⊔	A
*	*	A
B	+ / *	AB
+	+	AB*
(	(/ +	AB*
C	C / (/ +	AB*C
-	- / C / (/ +	AB*C
D	- / C / (/ +	AB*CD
/	/ / - / C / (/ +	AB*CD
E	/ / - / C / (/ +	AB*CDE
)	+ / - / C / (/ +	AB*CDE/-
#	⊔	AB*CDE/-+

Postfix expressions :- AB*CDE/-+

Evaluating the Postfix Expression

Algorithm to evaluate the obtained Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter - #"

Step 1: If the character is an operand, push its associated value onto the stack.

Step 2: If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

Postfix expression :- AB*CDE/-+

operand	value
A	2
B	3
C	4
D	4
E	2

char Read	stack
A	2
B	3 2
*	6
C	4 6
D	4 4 6
E	
/	2 4 6
-	2 6
+	8

Towers of Hanoi

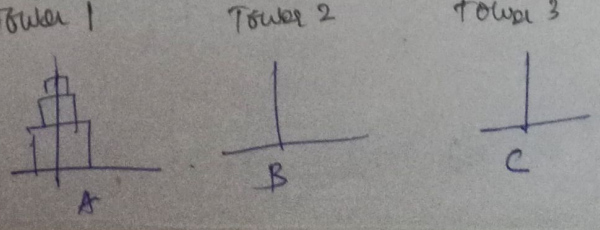
Towers of Hanoi can be easily implemented using recursion. Objective of the problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

Rule 1: Only one disk could be moved at a time.

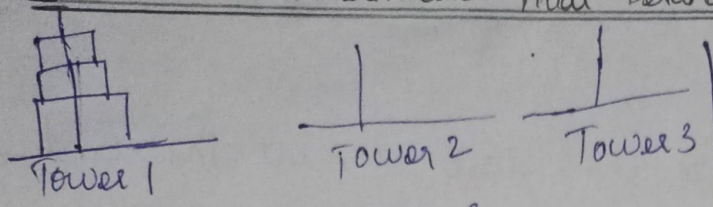
Rule 2: No larger disk could ever reside on a pillar on top of a smaller disk.

Rule 3: A 3rd pillar could be used as intermediate to store one (or more) disks, when they were being moved from from source to destination.

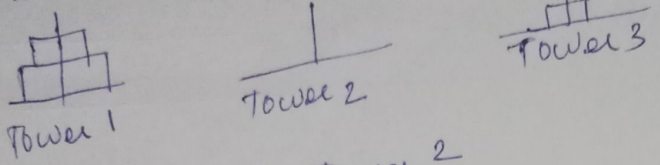
Initial setup of Tower of Hanoi



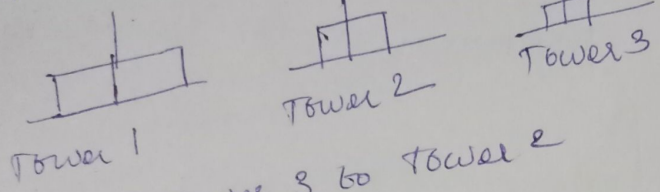
Source Pillar Intermediate Pillar Destination Pillar



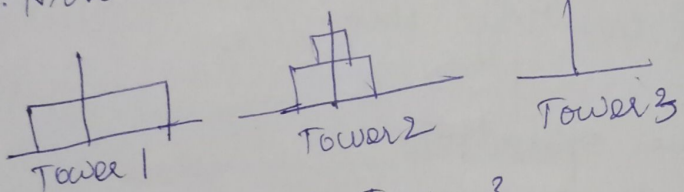
1. Move Tower 1 to Tower 3



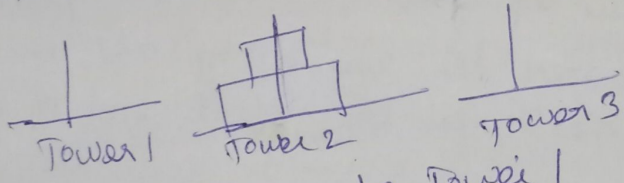
2. Move Tower 1 to Tower 2



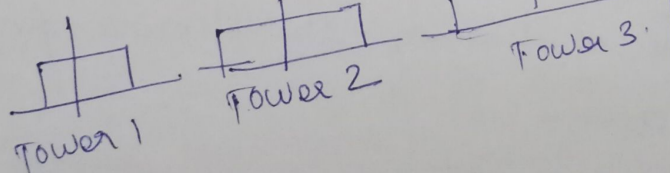
3. Move Tower 3 to Tower 2



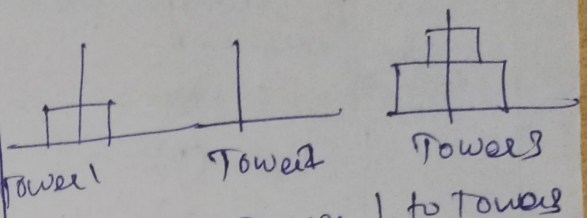
4. Move Tower 1 to Tower 3



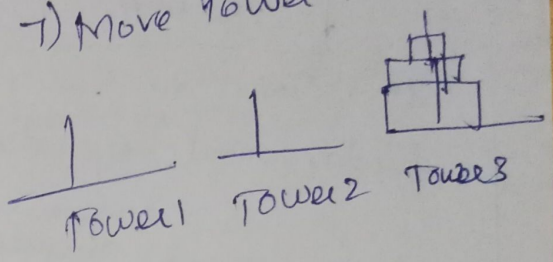
5. Move Tower 2 to Tower 1



6) Move tower 2 to Tower 2



7) Move Tower 1 to Tower 3



* Since disks are moved from each tower in LIFO manner, each tower may be considered as stack. least Number of moves required solving the problem according to our algorithm is given by

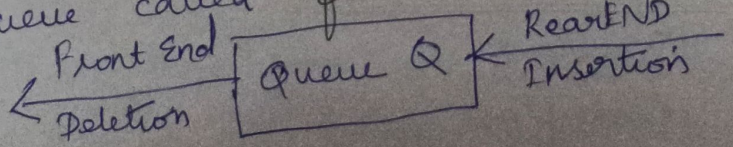
$$O(N) = O(N-1) + 1 + O(N-1)$$
$$= 2^N - 1$$

11) Queues:

Queue is a Linear Data Structure that follows First in First Out (FIFO) principle.

Insertion of element is done at one end of the que called "Rear" end of the queue.

Deletion of element is done at other end of the Queue called "front" end of the queue.



Front Pointer:-

It always points to the first element inserted in the Queue

Rear Pointer:-

It always points to the last element inserted in the Queue.

For Empty Queue -

Front (F) = -1

Rear (R) = -1

Operations on Queue:

fundamental operations performed on the queue are

1. Enqueue
2. Dequeue

(i) Enqueue operation:

⇒ It is the process of inserting new element at the rear end of the Queue.

⇒ for every Enqueue operation

* Check for full queue
* If the Queue is full, Insertion is not possible
* Otherwise, increment the rear end by 1 and then insert the element in the rear end of the Queue.

(ii) Dequeue operation

⇒ It is the process of deleting the element from the front end of the queue.

⇒ for every Dequeue operation

* Check for empty Queue.
* If the queue is empty, Deletion is not possible.
* Otherwise, delete the first element inserted into the queue and then increment the front by 1.

Exceptional conditions of Queue

* Queue Overflow
* Queue Underflow

(i) Queue Overflow

*An Attempt to insert an element X at the Rear end of the Queue when the Queue is full is said to be Queue Overflow

*for every Enqueue Operation, we need to check this condition

(ii) Queue Underflow:-

*An Attempt to delete an element from the front end of the Queue when the Queue is empty is said to be Queue underflow.

*for every DeQueue operation, we need to check this condition

13) Implementation of Queue:-

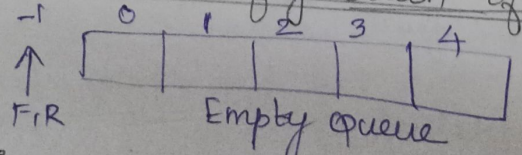
Queue can be implemented in two ways.

1. Implementation using Array (Static Queue)
2. Implementation using Linked list (Dynamic Queue)

Array Declaration of Queue:-

```
#define ArraySize 5
int Q[ArraySize];
  (or)
int Q[5];
```

Initial configuration of Queue:-



(i) Queue Empty Operation:

- Initially Queue is Empty.
  - with empty Queue, front(f) and Rear(R) point to -1.
- It is necessary to check for Empty Queue before deleting (DeQueue) an element from the Queue(Q).

(ii) Queue full operation

- As we keep inserting the new elements at the Rear end of the Queue, the Queue becomes full.

* when the Queue is full, Rear reaches its maximum array size.

* for every Enqueue operation, we need to check for full Queue condition.

(iii) Enqueue Operation:

- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, Enqueue(X,Q), The element X to be inserted at the Rear end of the Queue Q.
- Before inserting a new Element into the Queue, check for full que.
- If the Queue is already full, 'insertion is not possible.
- otherwise, Increment the Rear pointer by 1 and then insert the element X at the Rear end of the Queue.
- If the Queue is Empty, Increment both front and Rear pointer by 1 and then insert the element X at the Rear end of the Queue.

(iv) DeQueue Operation:

⇒ It is the process of deleting a element from the front end of the Queue.

⇒ It takes one parameter, DeQueue(Q). Always front element in the Queue will be deleted.

⇒ Before deleting an Element from the Queue, check for Empty Queue.

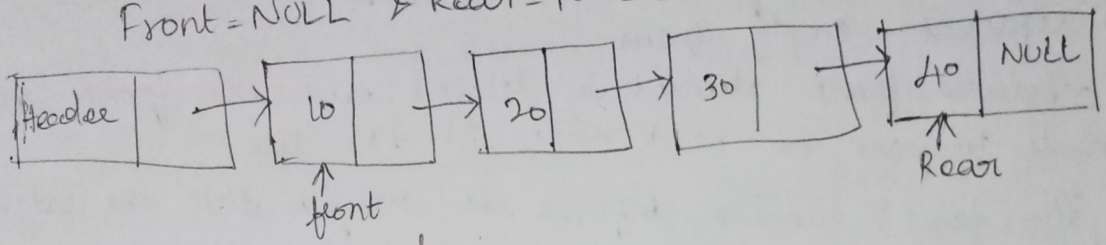
⇒ If the Queue is empty, deletion is not possible.

⇒ If the queue has only one element, then delete the element and represent the empty queue by updating front = -1 and Rear = -1.

⇒ If the Queue has many Elements, then delete the element in the front and move the front pointer to next element in the Queue by incrementing front pointer by 1.

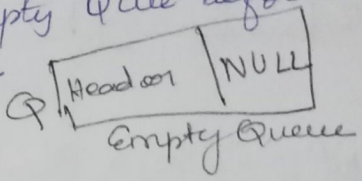
Linked List Implementation of Queue

* Queue is implemented using SLL (Singly Linked List) node.
* Enqueue operation is performed at the end of the linked list and Dequeue operation is performed at the front of the linked list.
* With linked list implementation, for Empty Queue Front = NULL & Rear = NULL



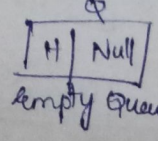
(i) Queue Empty Operations :-

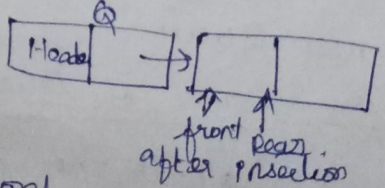
* Initially queue is Empty.
* With linked list implementation, Empty Queue is represented as S→next = NULL.
* It is necessary to check for Empty Queue before deleting the front element in the Queue.



(ii) Enqueue Operation

* It is the process of inserting a new element at the Rear end of the queue.
* It takes two parameters, Enqueue(int x, Queue Q), The element X to be inserted into the Queue Q.
* Using malloc() function allocate memory for the newnode to be inserted into the Queue.
* If the Queue is Empty, the newnode to be inserted will become first and last node in the list. Hence front and Rear points to the newnode.
* Otherwise insert the newnode in the Rear→next and update the Rear pointer.

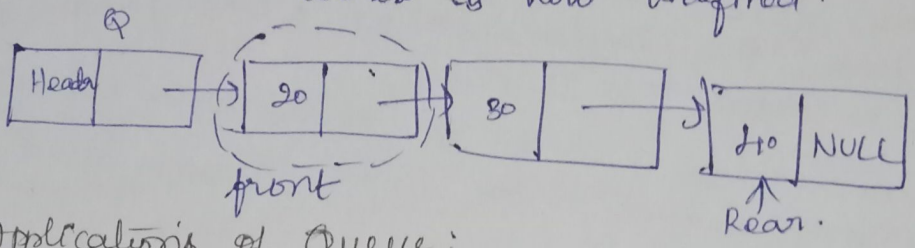




(iii) Dequeue Operation :

* It is the process of deleting the front element from the Queue.
* It takes one parameter, Dequeue(Queue Q). Always the element in the front (i.e) element pointed by Q→next is deleted always.
* element to be deleted is made "temp".

* If the Queue is Empty, the deletion is not possible.
* ~~If the Queue is Empty, then deletion is not possible~~
* If the Queue has only one element, then the element is deleted and front and Rear pointer is made NULL to represent empty Queue.
* Otherwise, front element is deleted and the front pointer is made to point to next node in the list.
* The free() function informs the compiler that the address the temp is pointing to, is unchanged but the data present in the address is now undefined.



14) Applications of Queue:

1) Serving requests on a single shared resource like a printer, CPU task scheduling etc.

2) In real life, call center phone systems will use Queues, to hold people calling them in an order, until a service representative is free

3) Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, first come first served.

4) Batch processing in operating systems

5) Job scheduling Algorithms like Round Robin Algorithm uses Queue.

Drawbacks of Queue (Linear Queue)

* With the array implementation of Queue, the element can be deleted logically only by moving front+1.
* Hence the Queue space is not utilized fully.

12) priority Queue:

priority queue is an abstract data type that is similar to a queue, and every element has some priority value associated with it. The priority of the elements in a priority queue determines the order in which elements are served.

Properties of priority Queue:

* Every item has a priority associated with it
* An element with high priority is dequeued before an element with low priority.

Operations of a priority queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue:

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right.

However, if the element is not in the correct place then it will be compared with the parent node.

2) Deletion in a priority Queue:

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue.

3) peek in a priority queue

This operation helps to return the maximum element from max heap or the minimum element from min heap without deleting the node from the priority queue.

Difference between priority Queue and Normal Queue:

There is no priority attached to elements in a queue, the rule of first-in-first-out (FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

How to implement priority queue

priority queue can be implemented using the following data structures.

- Arrays
- linked list
- Heap data structure
- Binary Search tree.

1) Implement priority Queue Using Array:

enqueue(): This function is used to insert new data into the queue. O(1)

dequeue(): This function removes the element with the highest priority from queue O(n)

peek()/top(): This function is used to get the highest priority element in the queue without removing it from the queue. O(n)

2) Implement Priority Queue using linked list:

push(): This function is used to insert new data into the queue. O(n)

pop(): This function removes the element with the highest priority from the queue. O(1)

peek()/top(): This function is used to get the highest priority element in the queue without removing it from the queue. O(1)

3) Implement Priority Queue Using Heaps:-

insert(p): inserts a new element with priority p.

extractMax(): Extracts an element with maximum priority.

remove(i): Removes an element pointed by an iterator i.

getMax(): Returns an element with maximum priority

changePriority(i,p): changes the priority of an element pointed by i to p

insert() - O(logn) ; remove() - O(logn) ; peak() - O(1)

4) Implement priority Queue Using Binary Search Tree:

A self-Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc. can also be used to implement a priority Queue.

Binary Search Tree	peek()	insert()	delete()
Time complexity	O(1)	O(logn)	O(logn)

Applications of priority Queue:

- CPU scheduling.
- Stack implementation.
- Data compression in Huffman code.
- finding kth largest/smallest element

Unit 4

Non-Linear Data Structures.

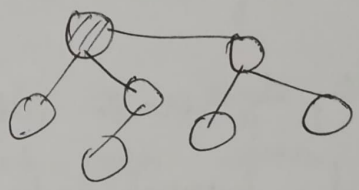
Trees – Binary Trees – Tree Traversals – Expression Trees – Binary Search Tree – Hashing – Hash functions – Separate chaining – Open Addressing – Linear Probing – Quadratic probing – Double Hashing – Rehashing.

1) Trees

A tree is a finite set of one or more nodes such that –

(i) There is a specially designated node called root.

(ii) The remaining nodes are partitioned into $n >= 0$ disjoint sets $T_1, T_2, T_3, \ldots T_n$ where $T_1, T_2, T_3, \ldots T_n$ are called subtrees of the root



Basic concepts in Trees:

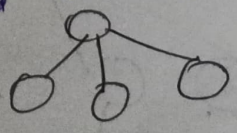
1. Root:
Root is a unique node in the tree to which further subtrees are attached.

2. Parent node: The node having further subbranches is called parent node.

3. Child nodes: The child nods in above given tree.

4. leaves:
These are the terminal nodes of the tree

5. Degree of the node:
The total number of subtrees attached to that node is called the degree of a node.



6. Degree of tree:
The maximum degree in the tree is degree of tree.

7. Level of trees:

The root node is always considered at level zero. The adjacent nodes to root are supposed to be at level 1 and so on.

8. Height of the tree:

The maximum level is the height of the tree, sometimes height of the tree is also called depth of tree.

9. Predecessor:

while displaying the tree, if some particular node occurs previous to some other node then that node is called predecessor of the other node.

10. Successor:

Successor is a node which occurs next to some node.

11. Inta and external nodes:

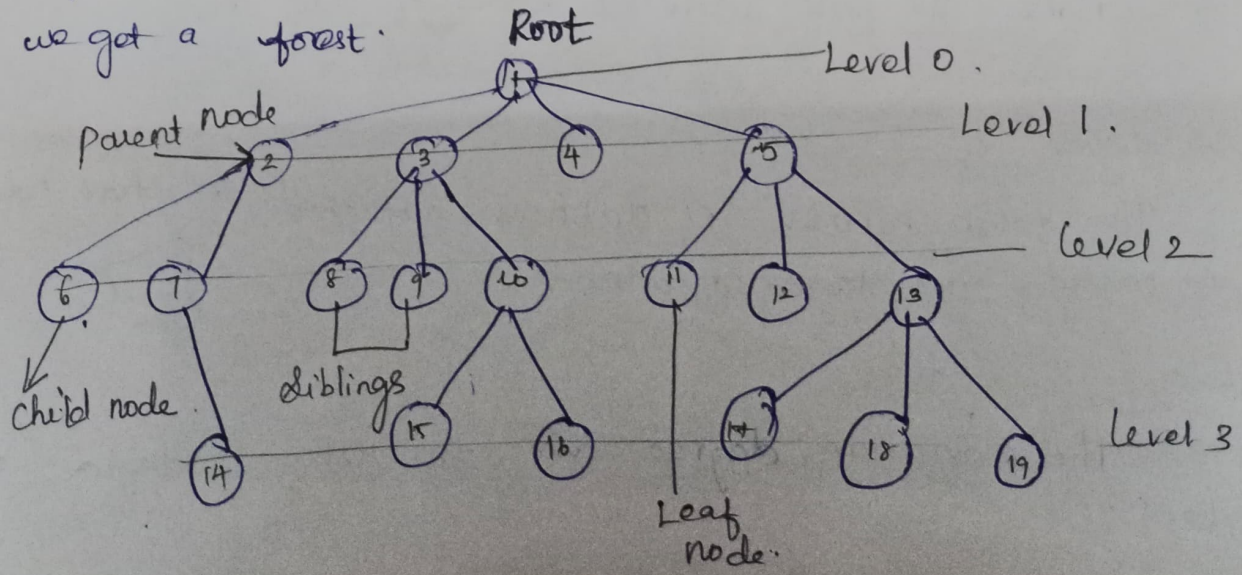
Leaf node means a node have no child node. As leaf nodes are not having further links, we call leaf nodes external nodes and non-leaf nodes are called internal nodes.

12. Sibling:

The nodes with common parent are called siblings or brothers.

13. forest:

forest is a collection of disjoint trees. from given tree if we remove its root then we remove its root then we get a forest.



properties of a tree

Number of edges: An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edge. There is only one path from each node to any other node of the tree.

Depth of a node: The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

Height of a node: The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

Height of the Tree: The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

Degree of a Node: The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the node in the tree.

Basic operation of Tree:

Create - Create a tree in data structure.

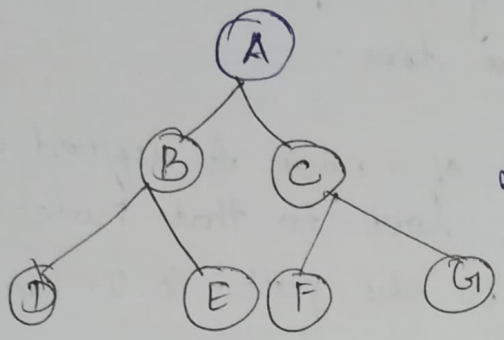
Insert - Inserts data in a tree.

Search - Search especific data in a tree to check it is present or not.

preorder Traversal - perform Travelling a tree in a pre-order manner in data structure.

In Order Traversal - perform Traveling a tree in an in-order manner.

Post order Traversal - Perform Traveling a tree in a post-order manner.



Types of Tree data Structure

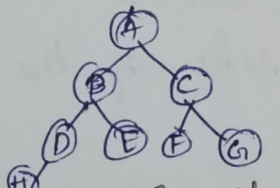
1. General tree: A general tree data structure has no restriction on the no. of nodes. It means that a parent node can have any number of child nodes.

2. Binary tree:

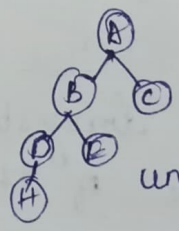
A node of a binary tree can have a maximum of two child nodes. In the given tree diagram, node B, D and F are left children, while E, C, and G are the right children.

3. Balanced tree:

If the height of the left subtree and the right subtree is equal (or) differs



Balanced Tree



unbalanced tree.

4. Binary search tree

BST are used for various searching and sorting algorithms. The examples include AVL tree and red-black tree. It is a non-linear data structure. It shows that the value of the left node is less than its parent, while the value of the right node is greater than its parent.

Applications of Tree data Structure

1. Spanning trees: It is the shortest path tree used in the routers to direct the packets to the destination

2. Binary search tree: It is a type of tree data structure that helps in maintaining a sorted stream of data.

1. full Binary tree
2. Complete Binary tree
3. Skewed Binary tree
4. Strictly Binary tree
5. Extended Binary tree

3. Storing hierarchical data: Tree data structures are used to store the hierarchical data, which is used in compilers.

4. Syntax tree: The syntax tree represents the structure of the program's source code, which is used in compilers.

5. Trie: It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.

6. Heap: It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

2) Binary Trees:

A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.

Binary Tree node contains the following parts:

1. Data
2. pointer to right child.
3. pointa to left child

Basic operation on Binary Tree:

* Inserting an element
* Removing an element
* Searching for an element.
* Traversing an element.

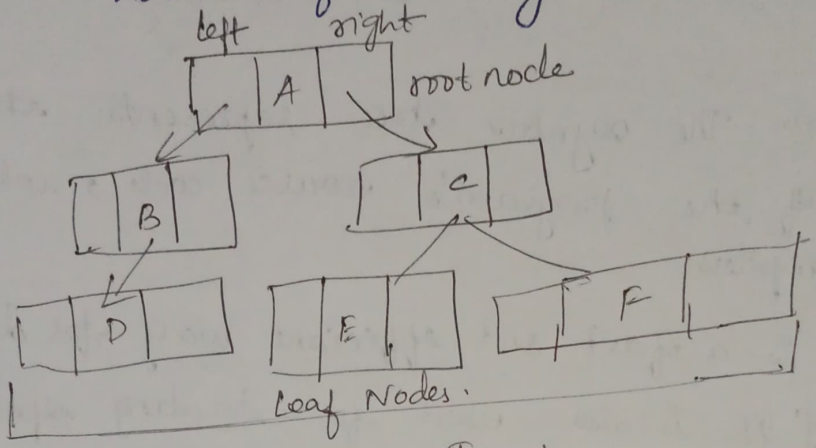
Auxiliary Operation on Binary Tree:

* Finding the height of the tree
* Find the level of the tree
* Finding the size of the entire tree

Each node in a binary tree has these three elem

1. Data item that is the data stored in the

2. Address of the left child.
3. Address of the right child.



Terminologies in Binary Trees:-

1. Nodes – Nodes are the building blocks of any data structu They majorly contain some data and link to the next/ previous nodes. In the case of binary trees, they contain the address of the left and the right child respectively.

2. Root – The topmost node in a tree is known as the root node. A tree can have at most one root node.

3. Parent Node – A node that has a succeeding node is known as a parent node.

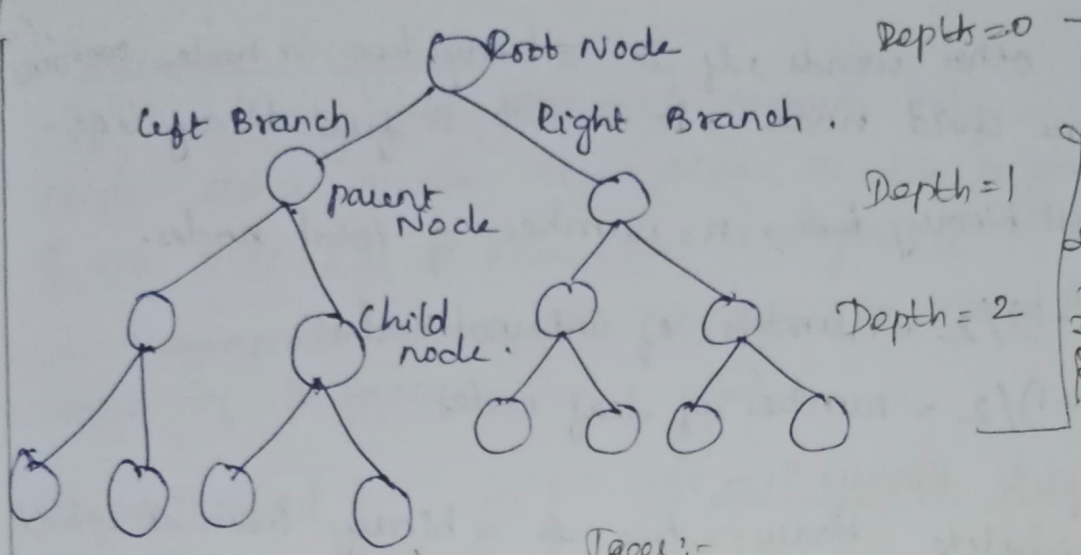
4. Child Node – A node that has a preceding node is known as a child node. A node can be both parent and child depending on the node.

5. Leaf Node – A node with no children.

6. Internal Node – A node that has at least one child node is known as an internal node.

7. Depth of Binary Tree – The number of edges from a node is known as an internal node.

8. Height of a Binary Tree – The number of edges from the deepest node in the tree to the root node.



properties of Binary Trees:-

1. If there are n nodes in a perfect binary tree, its height is given by $-\log n$. This is because, for a given node in a binary tree, there can be at most 2 child nodes. This further drives us to the explanation that at each level or height of a binary tree, the number of nodes will be almost equal to half of the number of nodes present in the next level. To put it simply, in a binary tree, the number of nodes of each level is almost double the number of nodes at the previous level.

2. The minimum number of nodes possible at the height h of a binary tree is given by h+1.

3. If a binary tree has L number of leaf nodes, its height is given by L+1.

4. At each level i of a binary tree, the number of total nodes is given by 2^{1i}

Types of Binary Trees:

Binary Trees are of many types, and each of these trees has its own properties and characteristics.

1. Full Binary Tree

A full binary tree, also known as a proper binary tree, is a tree in which each internal node has either zero or two children nodes is known as a full binary tree.

In other words, if in a binary tree a node contains only one child node, it is not a full binary tree.

In a full binary tree, n = number of total nodes.

$(n-1)/2$ = number of internal nodes

$(n+1)/2$ = number of leaf nodes

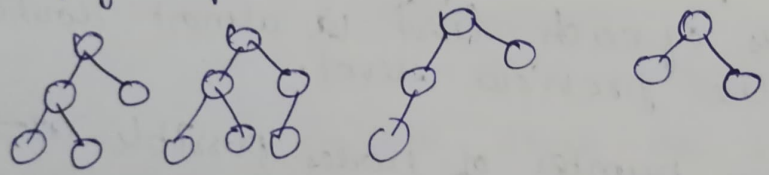
2. Complete Binary Tree:

A complete binary tree is a binary tree in which all the elements are arranged without missing any sequence.

In a complete binary tree –

* All the levels are completely filled except the last level that may or may not be completely filled.
* elements are filled from left to right.

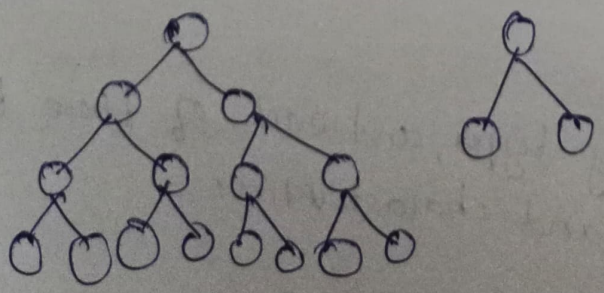
following trees are complete binary trees since they have no empty spaces in them.



3. Perfect Binary Trees:

If in a tree all the internal nodes have exactly two children nodes, it is known as perfect binary tree.

In a perfect binary tree, all the leaf nodes are on the same level.



Total no. of nodes.

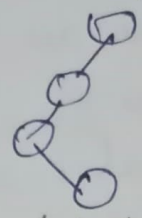
$2^h - 1$

h = height.

4) Degenerate Binary Tree:

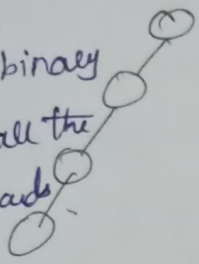
If in a binary tree each node contains only one child node either on the left side or the right side of the tree, it is known as a degenerate binary tree.

Degenerate binary tree is equal to linked lists in terms of performance.



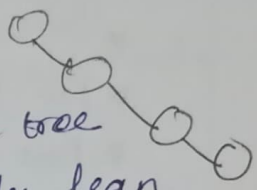
(a) Left-Skewed

A degenerate binary tree in which all the nodes lean towards the left side of tree.



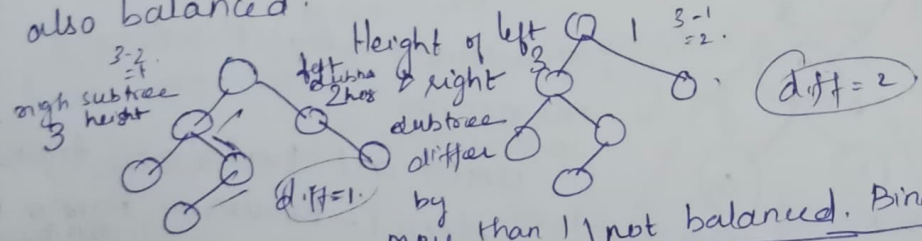
Right Skewed

A degenerate binary tree in which all the nodes lean towards the right side of the tree.



5) Balanced Binary Trees:

Binary tree is said to be balanced if the height of the left and the right subtree differ by 0 or 1. In balanced Binary Tree, both the left and right trees are also balanced.



Balanced Binary tree

not balanced Binary tree

Benefits of Binary Trees:

1. Binary trees are used in converting different Prefix and postfix expressions.
2. Binary trees are also used in graph traversal algorithms like Dijkastra's algorithm.

Time complexity	Space complexity
Searching = Insertion = Deletion = O(n)	Searching = Insertion = Deletion = O(n)

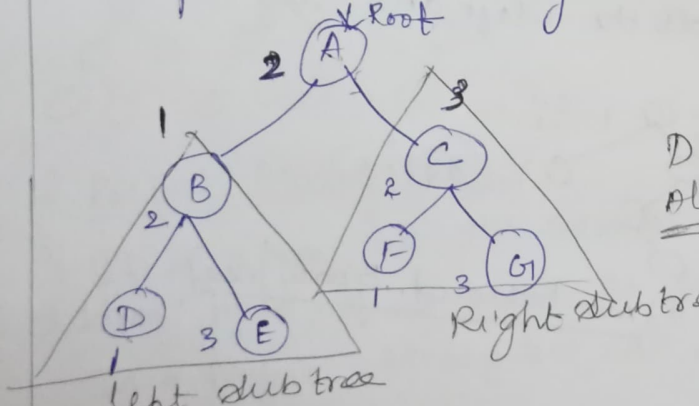
(3) Tree Traversal.

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree -

* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

In-Order Traversal:

* In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
* If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



D → B → E → A → F → C → G

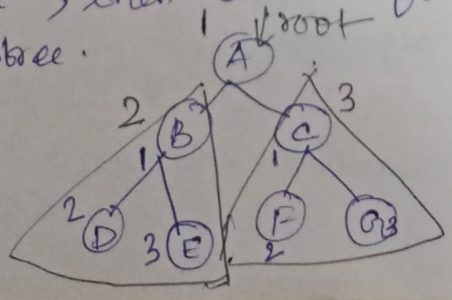
Algorithm

until all nodes are traversed

step 1 - Recursively traverse left subtree.
step 2 - visit root node.
step 3 - Recursively traverse right subtree.

Pre-order Traversal:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



A → B → D → E → C → F → G.

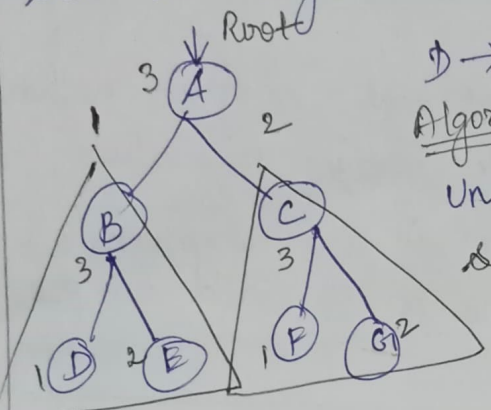
Algorithm:-

Until all nodes are traversed.

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

Post-Order Traversal:

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



D → E → B → F → G → C → A

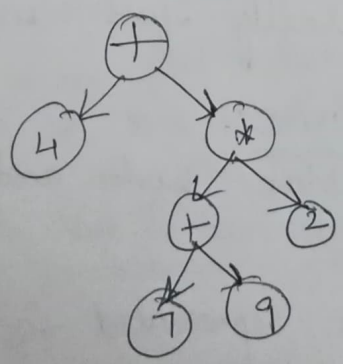
Algorithm

Until all nodes are traversed.
Step 1 - Recursively traverse left subtree.
Step 2 - Recursively traverse right subtree.
Step 3 - Visit root node.

4) Expression Trees:

Expression trees are those in which the leaf nodes have the values to be operated, and internal nodes contain the operator on which the leaf node will be performed.

example:
4+((7+9)*2) will have an expression tree as follows



Algorithm to construct an Expression tree:

If T is not NULL:
  If T->data is an operand:
    return T.data
  A = solve(T.left)
  B = solve(T.right)
  -> calculate operator for "T.data" on A and B, and call recursively
  return calculate(A, B, T.data)

How to construct an expression tree?

⇒ To construct an expression Tree for the given expression, we generally use stack Data structure.

⇒ Initially we iterate over the given postfix expression and follow the steps as given below.

* If we get an operand in the given expression, then push it in the stack. It will become the root of the expression tree as it child, and push them in the current node.

* Repeat step-1 and step-2 until we do not complete over the given expression.

* Now check if every root node contains nothing but operands and every child node contains only values.

* If an operator gets two values in the expression, then add in the expression tree as its child and push them in the current node.

(5) Binary search tree.

→ The binary search tree is an advanced used for analyzing the node, its left and right branches, which are modeled to a tree structure and returning the value. The BST is devised on the architecture of a basic binary search algorithm; hence it enables faster lookups, insertions, and removals of nodes. This makes the program really fast and accurate.

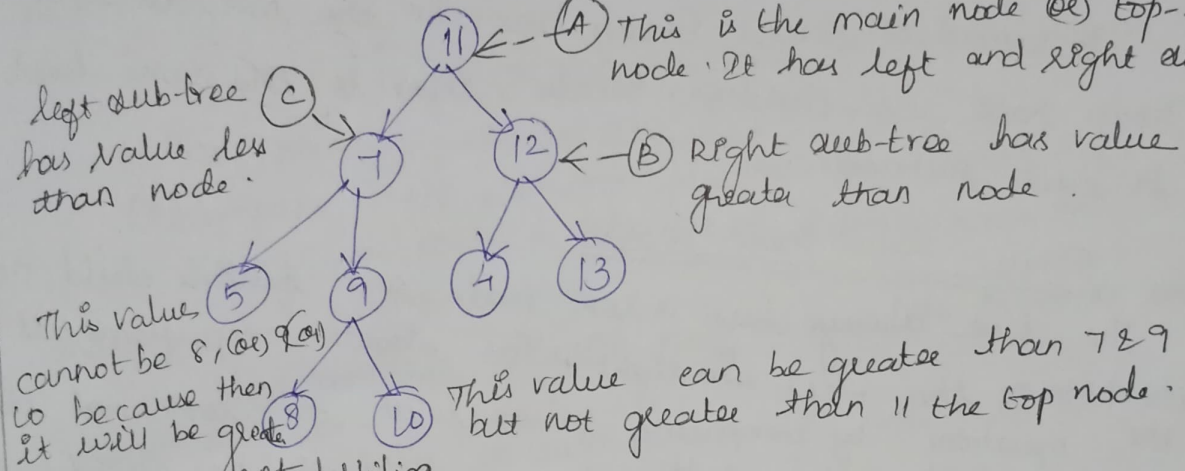
Attribute of Binary Search Tree:

A BST is made of multiple node and consists of the following attributes.

* Nodes of the tree are represented in a parent-child relationship.

* Each parent node can have zero child nodes or a maximum of two subnodes (or) subtrees on right side.

* Every sub-tree, also known as a binary search tree has sub-branches on the right and left of themselves.
* All the nodes are linked with key-value pairs.
* The keys of the nodes present on the left subtree are smaller than the keys of their parent node.
* Similarly, the left subtree nodes keys have lesser values than ~~their~~ their parent node's keys.



(A) There is the main node (or) parent level 11. Under it, there are left and right nodes/branches with their own key values.

(B) The right sub-tree has key values greater than the parent node.

(C) The left sub-tree has values than the parent node.

need a Binary Search Tree:

⇒ The two major factors that make binary Search Tree an optimum solution to any real-world problems are speed & accuracy.

⇒ Due to the fact that the binary search is in a branch search is in a branch-like format with parent-child relations, the algorithm knows in which location of the tree the elements need to be searched. This decreases the number of key-value comparisions the program has to make to locate the desired element.

⇒ The algorithm ~~activities~~ efficiently supports operations like search, insert and delete.

⇒ BST is commonly utilized to implement complex search robust game logics, graphics and auto-complete activities

Types of Binary Trees:-

Three kinds of binary trees are:

Complete binary tree: All the levels in the trees are full of last level's possible exceptions. Similarly, all the

Full binary tree: All the nodes have 2 child nodes except the leaf.

Balanced (or) Perfect Binary tree: In the tree, all the nodes have two children. Besides, there is the same level of each subnode.

Binary Search Tree Works:

The tree always has a root node and furtha child nodes whether on the left or right. The algorithm performs all the operations by comparing values with the root and its further child nodes in the left (or) right sub-tree accordingly.

BST primary offers the following three types of operations for your usage.

Search: Searches the element from the binary tree.

Insert: adds an element to the binary tree.

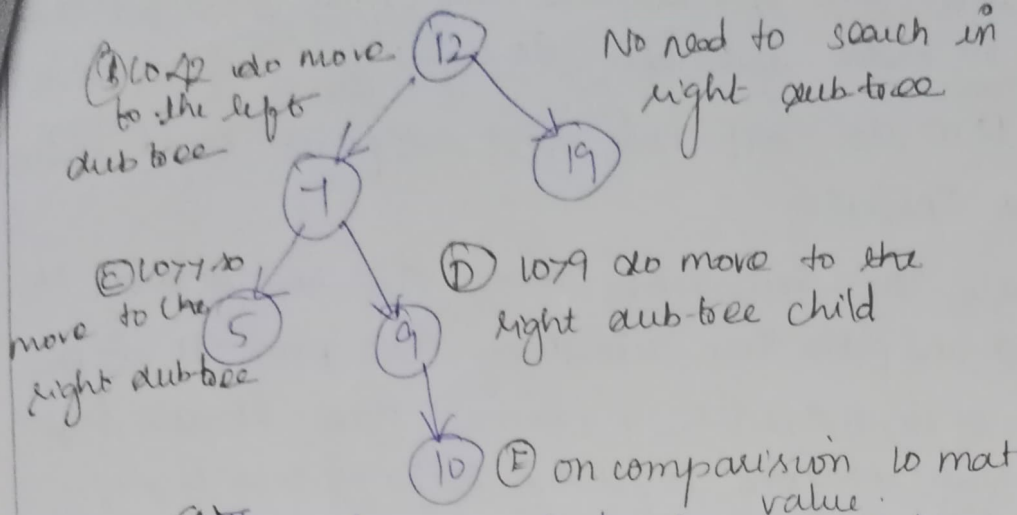
Delete: delete the element from a binary tree.

Each operation has its own structure and method of execution/analysis, but the most complex of all is the Delete operation.

Search operation:

Always initiate analyzing tree at the root node and then move further to either the right (or) left sub tree of the root node depending upon the element to be located is either less (or) greater than the root.

(A) Elements to be searched in the tree is 10.



(A) The element to be searched is 10.

(B) Compare the element with the root node 12, $10 < 12$, hence you move to the left subtree. No need to analyze the right-subtree

(C) Now compare 10 with node 7, $10 > 7$, so move to the right-subtree.

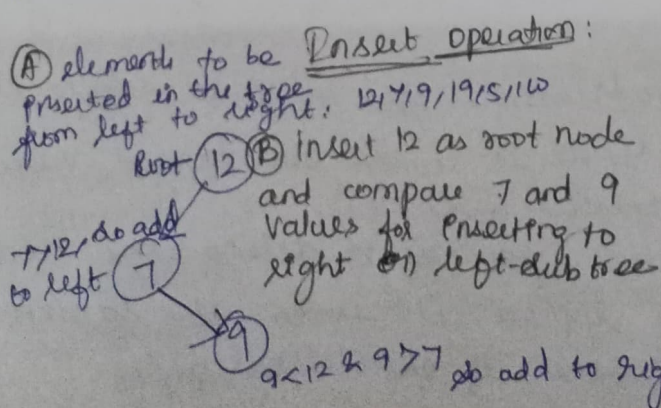
(D) Then compare 10 with next node, which is 9, $10 > 9$, look in the right subtree child.

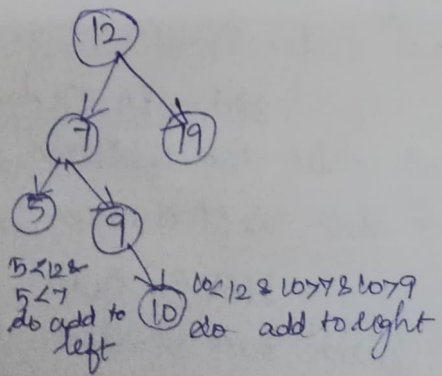
(E) 10 matches with the value in the node $10 = 10$, return the value to the user.

Insert Operation:

This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it less than the root it is added to the right subtree, and if is lesser than the root, It is added to the left subtree.

(A) elements to be inserted in the tree from left to right: 12, 7, 9, 19, 5, 10





(A) There is a list of 6 elements that need to be inserted in a BST in order from left to right.

(B) Insert 12 as the root node and compare next values 7 and 9 for inserting.

(C) Compare the remaining values 19, 5 and 10 with the root node 12 and place them accordingly. 19 > 12 place it as the right child of 12, 5 < 12 & 5 < 7, hence place it as left child of 7. Now compare 10, 10 is < 12 & 10 > 7 & 10 is > 9, place 10 as right subtree of 9

Delete Operations

for deleting a node from a BST, there are some cases, i.e. deleting a root or deleting a leaf node. Also, after deleting a root, we need to think about the root node.

to delete a leaf node, we can just delete it, we can just delete it, but if we want to delete a root

Case 1 - Node with 0 children: this is the easiest situation you just need to delete the node which has no further children on the right or left

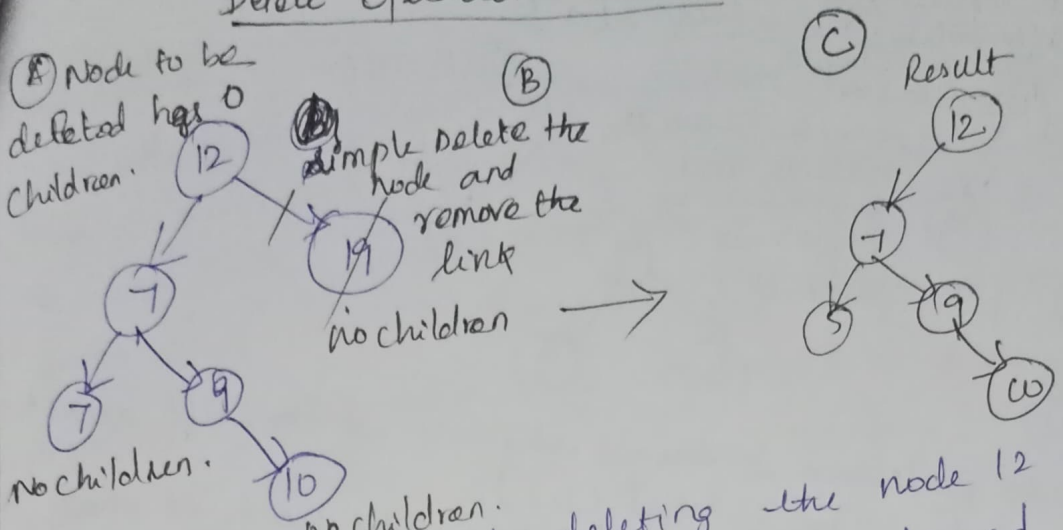
Case 2 - Node with one child: Once you delete the node, simply connect its child node with the parent node of the deleted value.

case 3 - Node with two children: this is the most difficult situation, and it works on the following two rules.

3(a) - In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of deleted node.

3(b) - In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node.

Delete Operation - Case 1



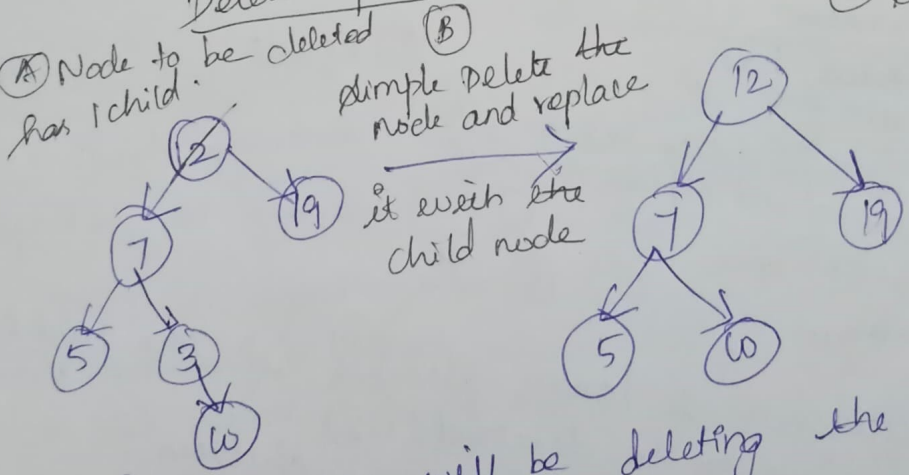
(A) Here you will be deleting the node 12 that has two children.

(B) The deletion of the node will occur based upon the

(C) Delete the node 12 and replace it with 10 as it is the largest value on the left subtree

(D) View the new structure of the BST after deleting 12.

Delete operation - case 2.



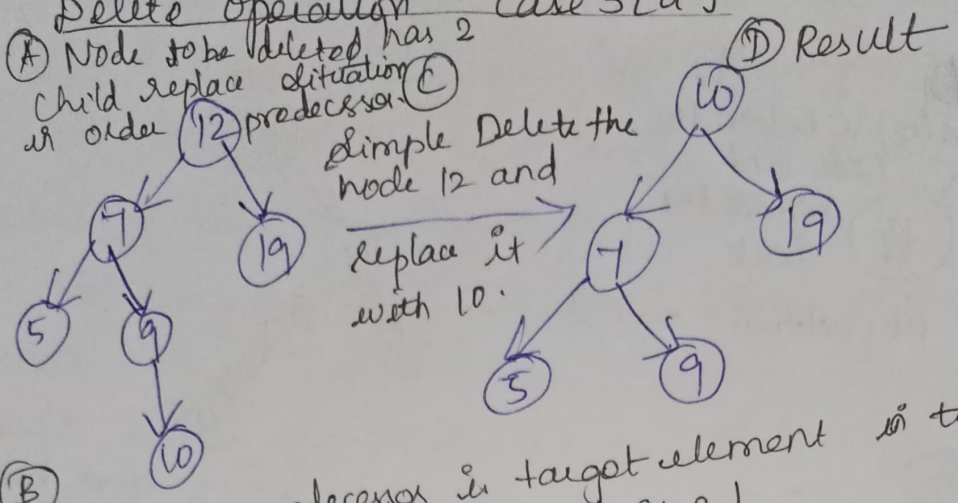
(A) Here you will be deleting the node 12 that has two children.

(B) The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on left subtree 12 will replace it

(C) Delete the node 12 and replace it with 10 as it is the largest value on the left subtree.

(D) View the new structure of the BST after deleting 12.

Delete Operation - case 3[a].



(B) In order predecessor is target element of the node to be deleted (1,2) in the left sub-tr

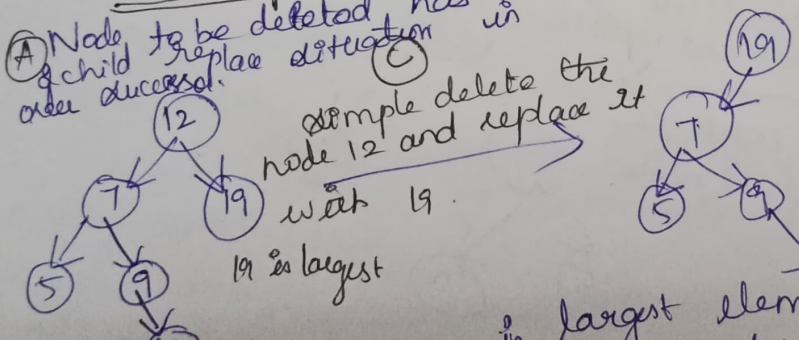
(A) Here you will be deleting the node 12 that ha two children.

(B) The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.

(C) Delete the node 12 and replace it with 10 as it is the largest value on the left subtree.

(D) View the new structure

Delete operation - Case 3(b)



(B) In order successor is largest element in the right sub-tree of the node to be deleted (1,2) children.

(A) 1 Delete a node 12 that has two

(B) 2 The deletion of the node will occur based upon the In order successor rule, which means that the largest element on the right subtree of 12 will replace it it is the largest value on the right subtree.

(C) 3 Delet the node 12 and replace it with 19 as

(D) 4 View the new structure of the BST after deleting 12.

6) Hashing:-

Hashing is technique (or) process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of mapping depends on the efficiency of the hash function used.

H(x) hash function maps the value x at the index x%10 in array. for example if the list of value is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively

List = [11, 12, 13, 14, 15]

$H(x) = [x \% 10]$

11%10 = 1, 12%10 = 2, 13%10 = 3, 14%10 = 4, 15%10 = 5

0	1	2	3	4	5
	11	12	13	14	15

Hash Table

11% = 0.11 0.011.

7) Hash Functions:

Hash function is a function that maps any big number (or) string to a small integer value.

Hash function takes the data item as an input and returns a small integer value as an output.

The small integer value is called as a hash value. Hash value of the data item is then used as an index for storing it into the hash table.

Types of hash function

1. Mid Square Function.
2. Division Function
3. Folding Hash Function

properties of Hash function:

* It is efficiently computable.
* It minimizes the number of collision.
* It distributes the key uniformly over the table.

(8) Separate chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly linked list which is known as a chain.

Here, all those elements that hash into the same slot index are inserted into linked list. Now, we can use a key k to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to k then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining. If two different elements have the same hash value then we store both the elements in the same linked list one after the other.

example: Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

m = Number of slots in hash table.
n = Number of keys to be inserted in hash table.
Load factor $\alpha = n/m$
Expected time to search = $O(1+\alpha)$
Expected time to delete = $O(1+\alpha)$
Time to insert = $O(1)$
Time complexity of search insert and delete is $O(1)$ if α is $O(1)$

(4) Open Addressing

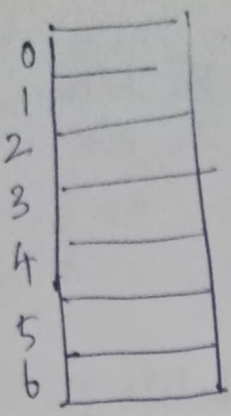
Like separate chaining, open addressing is a method for handling collisions. In open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the number of keys. This approach is also known as closed hashing. This entire procedure is based upon probing.

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

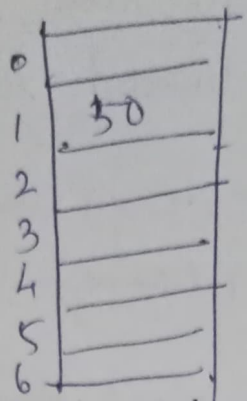
Search(k): Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): * Delete operation is interesting. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".

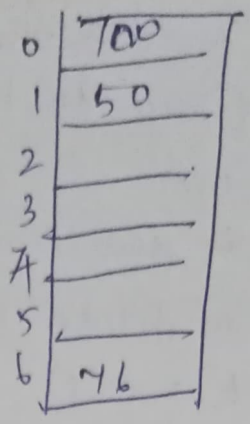
* The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.



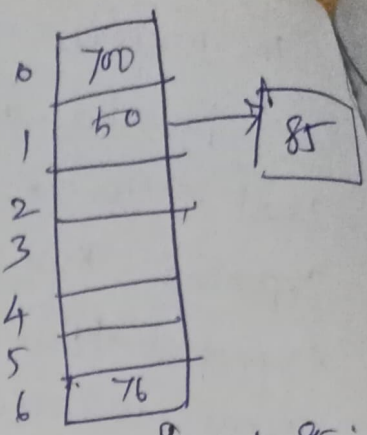
Initial Empty Table



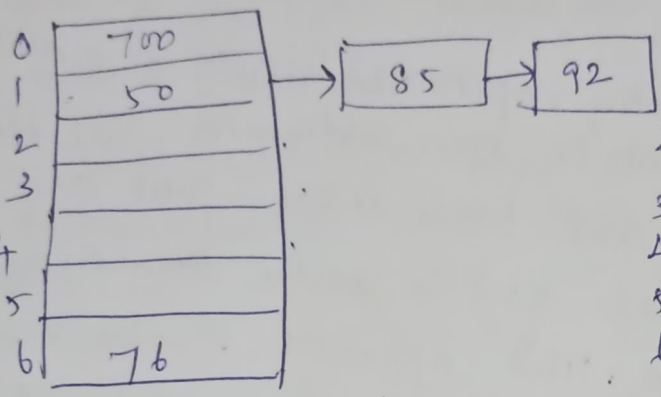
Insert(50)



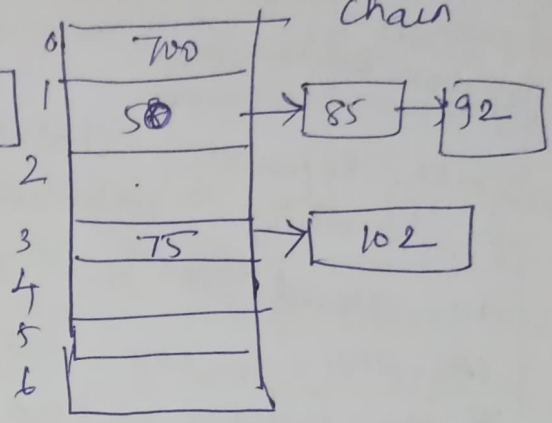
Insert 700 & 76.



Insert 85: collision occurs, add to chain



Insert 92 collision occurs, add to chain.



Insert 75 & 102

Advantages:

* Simple to implement.
* Less sensitive to the hash function function or load factors.
* It is mostly used when it is unknown how many & how frequently keys may be inserted or deleted.

Disadvantages:

* Wastage of space.
* Uses extra space for links.
* If the chain becomes long, then search time

Performance of chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table.

Different ways of open addressing:

- Linear Probing
- Quadratic probing.
- Double Hashing:

(i) Linear Probing

In Linear Probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows:

rehash(key) = (n+1) % table_size

for example: The typical gap between two probes is 1 as seen in the example below:

Let hash(x) be the slot index computed using a hash function and S be the table size.

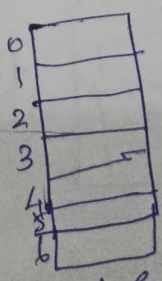
If slot hash(x) % S is full, then we try (hash(x)+1) % S)

If (hash(x)+1) % S is also full, then try (hash(x)+2) % S

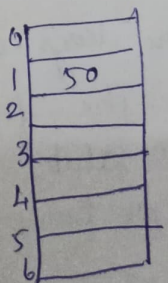
If (hash(x)+2) % S is also full, then we try (hash(x)+3) %

Example:

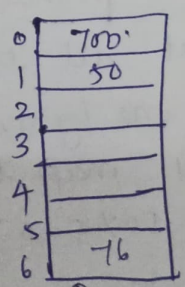
Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.



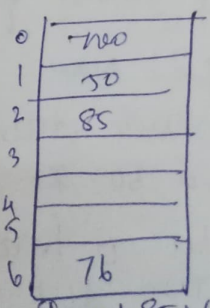
Initial empty table



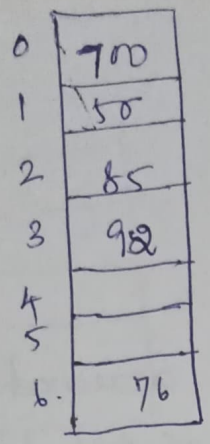
Insert (50)



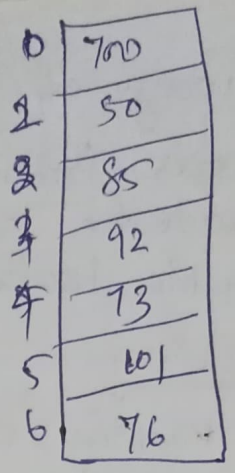
Insert 700 & 76



Insert 85: collision occurs, insert 85 at next free slot.



Insert 92, collision occcurs 50 is there at index 1. Insert at next free slot.



Insert 73 & 101

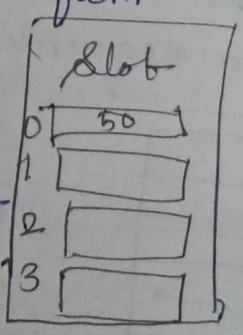
Challenges in Linear probing:

Primary clustering: One of the Problems with linear Probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.

Secondary clustering:- Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

example:- let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted 50, 70, 76, 93.

step1: First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.

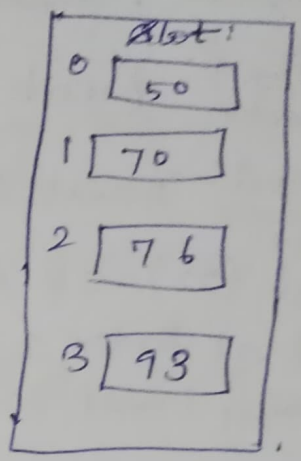


step2: Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. so Insert it into slot number 0.

10
5 | 50
50
0

Step 3:- The next key is 70. It will map to slot number 0 because 70%.5=0 but 5 is already at slot number 0 so, search for next empty slot and insert.

Step 4:- The next key is 76. It will map to slot number 1 because 76%.5=1 but 70 is already a slot number 1 so, search for the next empty slot and insert it.



Step 5: The next key is 93 it will map to slot number 3 because 93%.5=3 so insert it into slot number 3

(ii) Quadratic Probing:

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which can solve the problem of clustering that was discussed above. This method is also known as the mid-square method. In this method we look for the i^2th slot in the ith iteration. We always start from the original hash location. If only the location is occupied then we check the other slot

Let hash(x) be the slot index computed using hash function.

If slot hash(x)%.S is full, then we try (hash x)+1
%.S.

If (hash(x)+1*1)%.S is also full, then we try (hash(x)+2
%.S

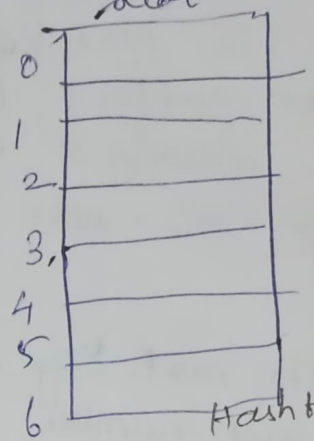
If (hash(x)+2*2)%.S is also full, then we try (hash(x)+3*3)%.S.

try ....

Example:

Let us consider table size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30 and 50.

Step 1: Create a table of size 7 for slot

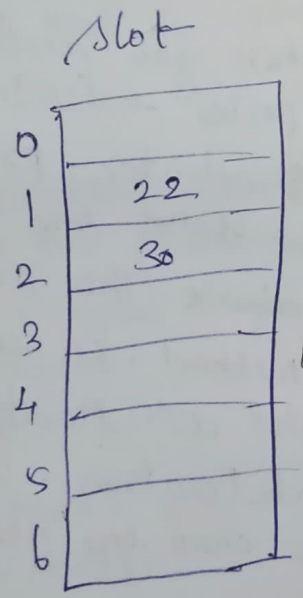


Hash table.

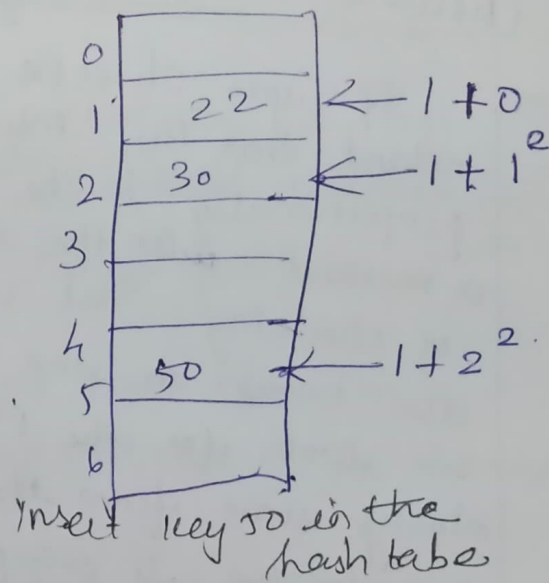
Step 2: Insert 22 and 30.

Hash(22) = 22 % 7 = 1, since the cell at index 1 is empty, we can easily insert 22 at slot 1.

Hash(30) = 30 % 7 = 2, since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 & 30 in the hash table.



Step 3: Inserting 50.

* Hash(50) = 50 % 7 = 1
* In our hash table slot 1 already occupied, so, we will search for slot $1+1^2$, i.e 1+1=2
* again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e 1+4=5.
* Now, cell 5 is not occupied so we will place 50 in slot 5.

12) Double Hashing:-

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the ith rotation.

Let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1* has 2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S.

Example: Insert the keys 27, 43, 92, 72 into the Hash Table of size 7. where first hash-function is $h1(k) = k \bmod 7$ and second hash-function is $h2(k) = 1 + (k \bmod 5)$

Step 1: Insert 27

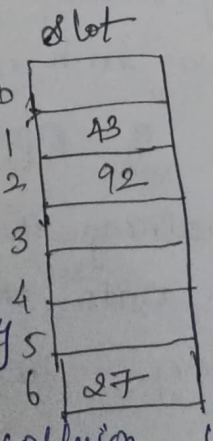
27 % 7 = 6, location 6 is empty so insert 27 into 6 slot

Step 2: Insert 43

43 % 7 = 1, location 1 is empty so insert 43 into 1 slot

Step 3: Insert 92:

92 % 7 = 6, but location 6 is already being occupied and this is a collision so we need to resolve this collision using double hashing.



$h_{new} = [h_1(92) + i * (h_2(92))] \% 7$

$= [6 + 1 * (1 + 92 \% 5)] \% 7$

$= 9 \% 7 = 2$ Now, as 2 is an empty slot so we can insert 92 into 2nd slot.

Step 4:

Insert 72.

* $72 \% 7 = 2$, but location is already being occupied and this is collision
* so we need to resolve this collision using double hashing

$h_{new} = [h_1(72) + i * (h_2(72))] \% 7$

$= [2 + 1 * (1 + 72 \% 5)] \% 7$

$= 5$, Now, as 5 is an empty slot, so we can insert 72 into 5th slot.

Slot	
0	
1	43
2	92
3	
4	
5	72
6	27

Comparision of the above three:

* Linear Probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
* Quadratic probing lies between the two in terms of cache performance and clustering.
* Double hashing has poor cache performace but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Performance of Open Addressing:

Like chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table.

m = Number of slots in the hash table.
n = Number of keys to be inserted in the hash table.
load factor $\alpha = n/m$ (< 1)
Expected time to search / insert / delete $< 1/(1-\alpha)$
So search, Insert and Delete take $(1/(1-\alpha))$ time.

Rehashing

* Rehashing is collision resolution technique
* Rehashing is a technique in which the table is resized (i.e.) the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.
  - => when table is completely full.
  - => with quadratic probing when the table is filled half.
  - => when insertions fail due to overflow.

In such situations, we have to transfer the entries from old table to the new table by recomputing their positions using hash function.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49 and 87. the table size is 10 and will use hash function.

$$H(key) = key \bmod table\ size$$

$37 \% 10 = 7$ $55 \% 10 = 5$ $17 \% 10 = 7$
$90 \% 10 = 0$ $22 \% 10 = 2$ $49 \% 10 = 9$

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, The becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be.

H(key) key mod 23.

37 % 23 = 14 17 % 23 = 17

90 % 23 = 21 49 % 23 = 3

55 % 23 = 9 87 % 23 = 18.

22 % 23 = 22

Now, the hash table is sufficiently large to accommodate new insertions.

Unit V

Sorting and Searching Techniques

Insertion Sort – Quick Sort – Heap Sort – Merge Sort – Linear Search – Binary Search.

1) Insertion Sort:

⇒ The insertion sort works by taking elements from the list one by one and inserting them in the correct position into a new sorted list.

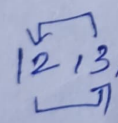
⇒ Insertion sort consists of N-1 passes, where N is the Number of elements to be sorted.

The ith pass will insert the ith element A[i] into its rightful place among A[1], A[2], ..., A[i-1].

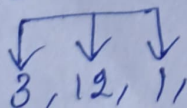
After doing this insertion the elements occupying A[1] ... A[i] are in sorted order.

How insertion sort algorithm works:-

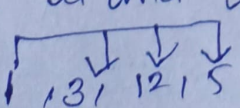
Step1: checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.

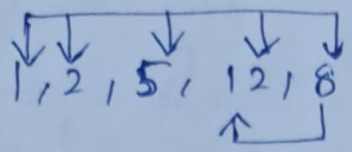


Step2: checking third element of array with elements before it and inserting it in proper position, In this case, 1 is inserted in position of 3.



Step3: checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.



Step 4: 

Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.

Step 5: 0, 1, 3, 8, 12

sorted array in Ascending Order.

Insertion sort routine:

```
Void Insertion_sort(int a[], int n)
{
int i, j, temp;
for(i=0; i<n-1; i++)
  {
  temp = a[j];
  for (j=i; j>0 && a[j-1]>temp ; j--)
    {
    a[j] = a[j-1];
    }
  a[j] = temp;
  }}
```

Program for Insertion Sort:

```
#include <stdio.h>
void main(){
int n, a[25], i, j, temp;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
for(i=0; i<n; i++)
  scanf("%d", &a[i]);
for(i=0; i<n; i++) {
  temp = a[i];
```

```
for (j=i ; j>0 && a[j-1]>temp ; j--)
 {
 a[j] = a[j-1];
  }
 a[j] = temp; }
printf (" sorted list in ascending order: \n");
for (i=0; i<n; i++)
    printf ("%d \n", a[i]); }
```

output:

Enter number of elements
6
Enter 6 Integers
20 10 60 40 30 15
Sorted list in ascending order:
10
15
20
30
40
60

Advantage of Insertion sort:

- Simple Implementation
- Efficient for (quite) small data sets.
- Efficient for data sets that are already substantially sorted.

Disadvantage of Insertion sort:

It is less efficient on list containing more number of elements.

As the number of elements increases the performance of the program would be slow.

Insertion sort needs a large number of element shifts.

2) Quick Sort

Quick sort is based on divide and conquer strategy. It works in the following steps:

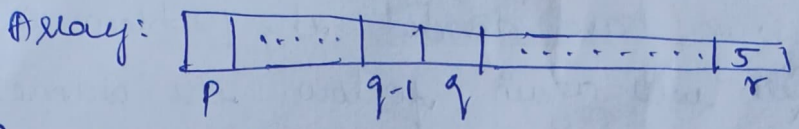
1. It selects an element from within the array known as the pivot element.
2. Then it makes use of the partition algorithm to divide the array into two sub arrays. One sub-array has all the values less than the pivot element. The other sub-array has all the values higher than the pivot element.
3. In the next step, the quicksort algorithm calls itself recursively to sort these two sub arrays.
4) Once the sorting is done, we can combine both the sub arrays into a single sorted array.

The most important part of quicksort is the partition algorithm. The partition algorithm puts the ~~partition~~ element into either of the two subarrays depending on the pivot point. We can choose pivot point in many ways:

* Take the first element as the pivot point.
* Take the last element of the array as the pivot point.
* Take random element as the pivot element in every recursive call.
* Take the middle element of the array as the pivot element.

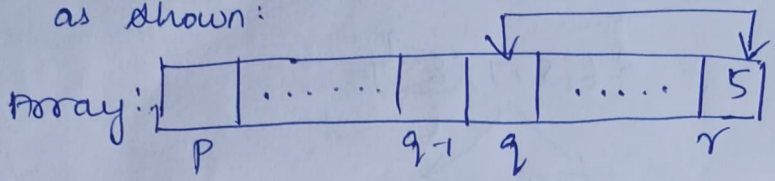
Partition Process in Quick Sort:-

It picks up an element called the pivot element and places that element at its correct position. Let us take the last element of the array to be the pivot element.



Let Pivot = Array[r] and let the correct position of the pivot element 5 be Array[q]. Then, all the elements from Array[p] to Array[q-1] will be less than 5 and all elements from Array[q+1] to Array[r] will be greater than 5.

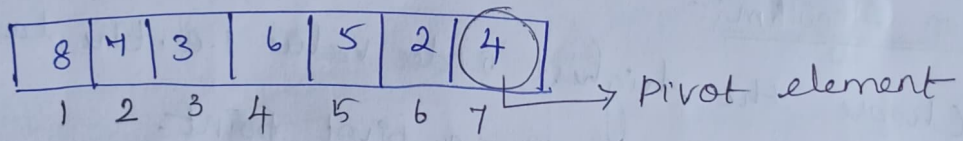
Thus, we will move 5 from Array[r] to Array[q] as shown:



And this process will recursively repeat for all the elements until the array is completely.

Working of Quicksort

consider an array having the following elements: 8,7,3,5,2,4. Let us choose the last element i.e. 4 as the pivot element.

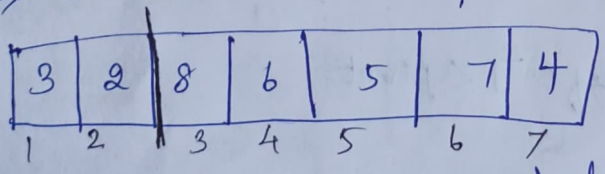


At position 1, we have 8 and 8>4, therefore, nothing will happen.

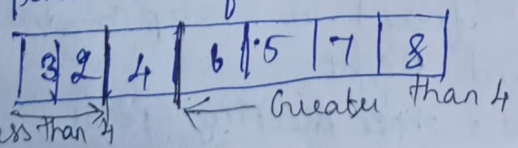
At position 2, there is 7 and 7>4, so do nothing.

At position 3, the element present is 3 and 3<4.

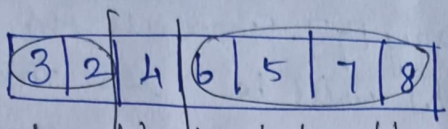
So, we will move 3 to position 1 as shown:



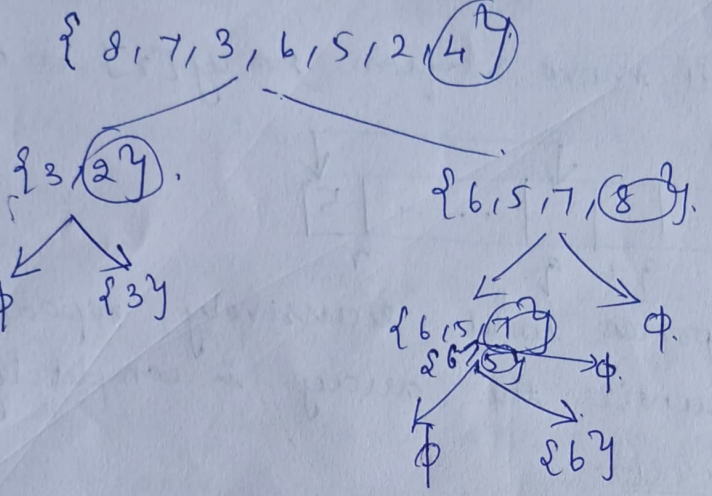
After traversing the whole array and comparing the pivot point with every element, we have found the right position for 4 which is at position 3.

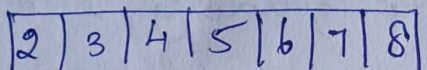


with this step, we have divided the problem into two sub-problems which will again follow the same procedure to sort the elements recursively.



sub problem 1 sub problem 2.





once we have applied the partition procedure on all the elements, the final sorted array will be:

Partition Algorithm:-

Step 1: Choose the highest index value i.e. the last element of the array as a pivot point.

Step 2: point to the 1st and last index of the array using two variables.

Step 3: left points to the low index and Right points to the high.

Step 4: While Array [left] < pivot
move Right

Step 5: while Array [Right] > pivot
move Left

Step 6: If no match found in step 5 and step 6, swap left and Right.

Step 7: If left ≥ Right, their meeting point is the new pivot

Quicksort Algorithm

Step 1 - Array [Right] = pivot

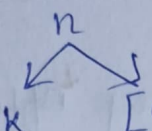
Step 2 - Apply partition algorithm over data items using pivot element

Step 3 - Quicksort (left of pivot)

Step 4 - quicksort (right of pivot)

Complexity of Quicksort

In quicksort, we sort the array by dividing it into sub-problems.



we divide a problem of size 'n' into two subproblems of size k and (n-1-k). Thus, the recursive function for the time complexity is: $T(n) = T(k) + T(n-k-1) + \Theta(n)$. Here, $\Theta(n)$ is added because of the time taken by the partition process

Scenario	Time complexity
Worst case	$O(n^2)$
Average case	$O(n \log n)$
Best case	$O(n \log n)$

3 ways Quicksort

* The first sub-array contains all the elements less than the pivot element.
* The second sub-array contains all the elements equal to the pivot element is repeated.
* The third sub-array contains all the values greater than the pivot element.

Advantages of Quick Sort:

* Fast and efficient as it deals well with a huge list of items.
* No additional storage is required.

Disadvantages of Quick Sort:

* The difficulty of implementing the partitioning algorithm.

Heap Sort:

* Heap Sort is a comparison-based sorting technique based on binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning.

* Heap sort is an in-place algorithm.

* Typically 2-3 times slower than well-implemented Quick sort

Advantages of heapsort:

Efficiency - Sorting algorithm is very efficient.

Memory usage - Memory usage is minimal because apart from from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work

Simplicity - It is simpler to understand.

Applications of HeapSort:

* HeapSort is mainly used in hybrid algorithms like the introsort.

* Sort a nearly sorted array.

* k largest (or smallest) elements in an array.

Types of Heap:

Min-heap:- The key present at the root node is smaller than or equal to keys all the nodes present in the children nodes.

Max-Heap:-
The key which is present at the rootnode is greater than (or) equal to the keys of all the children nodes of the tree. The same property recursively applicable for all the subtree of the tree:

Min Heap

The key which is present at root node of the tree is lesser than (or) equal to the keys present at the children nodes.

The minimum key element is present at the root in a Min-Heap.

Min-Heap uses the ascending property.

The smallest element has the priority in the construction of a min-Heap.

Max Heap

The key which is present at the root node of the tree is greater than (or) equal to the key present at the children nodes.

The maximum key element is present at the root in a Max Heap

Max-Heap uses the descending property.

The largest element has the priority in the construction of a Max-Heap.

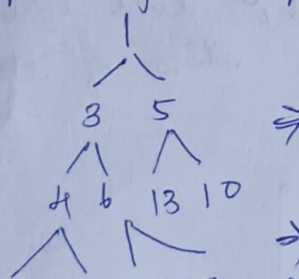
Heapify:

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the first index of the non-leaf node whose index is given by $n/2 - 1$. Heapify uses

How does Heapify work?

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:



⇒ The task to build a Max-Heap from above array

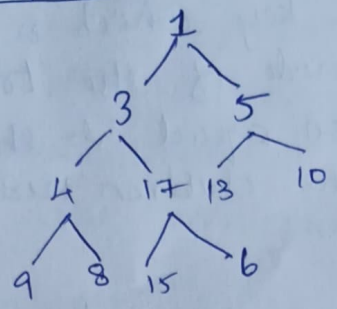
⇒ Total Nodes = 11

⇒ Last Non-leaf node index = $(11/2) - 1 = 4$

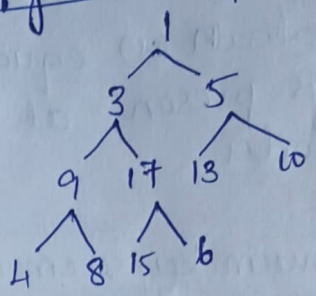
⇒ Therefore, last non-leaf node = 4

to build heap, heapify only the nodes: [1, 3, 5, 4, 6] in reverse order

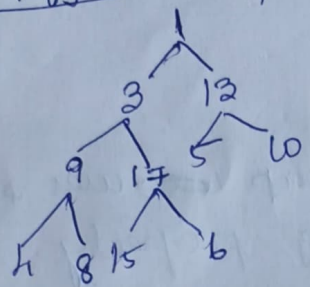
Heapify 6: Swap 6 and 17.



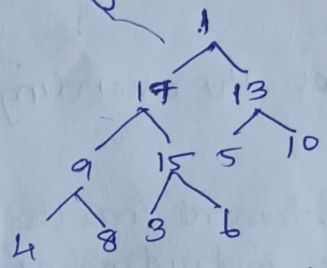
→ Heapify 4: Swap 4 and 9



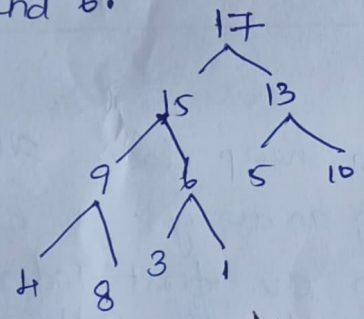
Heapify 5: Swap 13 and 5



Heapify 3: first Swap 3 & 17, again Swap 3 and 15



Heapify 1: First Swap 1 and 17, again swap 1 and 15, finally swap 1 and 6.



Heap Sort Algorithm

* Build a max heap from the input data.
* At this point, the maximum element is stored at the root of the heap. Replace it with with the last item of the heap followed by reducing the size of the heap by 1. finally, heapify the root of the tree
* Repeat step 2 while the size of the heap is greater than 1.

Time complexity: O(N log N) Auxiliary space: O(1)

4) Merge Sort Algorithm

The Merge Sort Algorithm is a sorting algorithm that is based on the Divide and conquer paradigm. In this algorithm, the array is initially divided in two equal halves and they are combined in a sorted manner.

Merge Sort Working Process:

Think of it as recursive algorithm continously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Illustration:

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}.

* At first, check if the left index of array is less than the right index, if yes then calculate its mid point

l - left Index r = right Index.

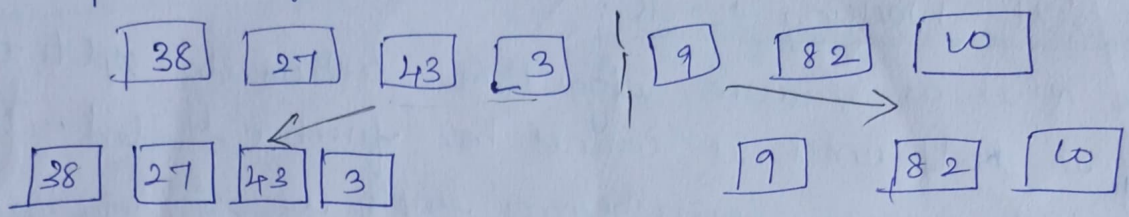
38	27	43	3	9	82	10

Is l < r yes m = l + (r - 1)/2

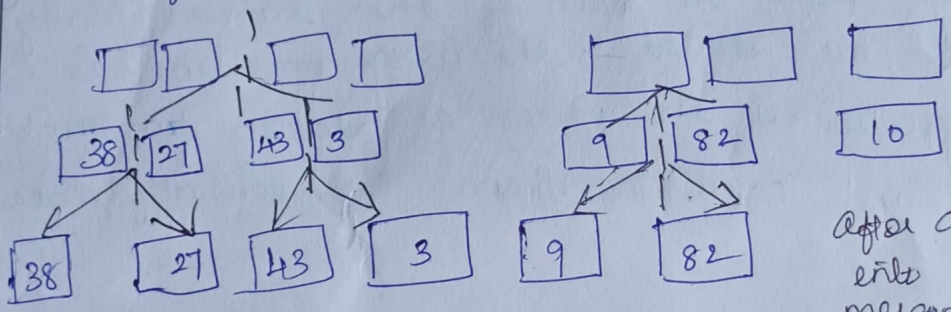
* Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

* Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

* Now, again find that is left index is less than the right index for both array, if found yes, then again calculate mid points for both the arrays.



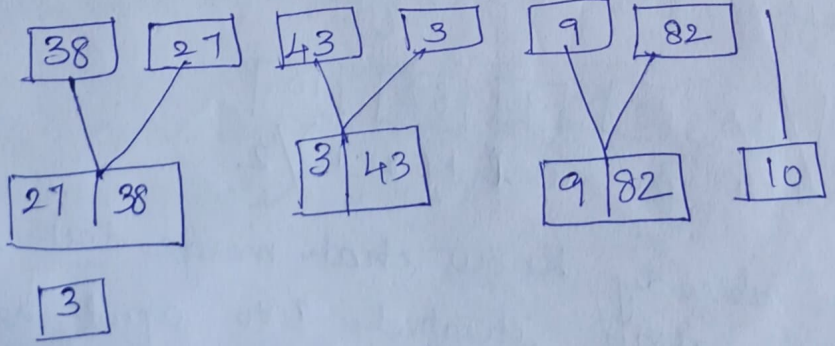
* Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



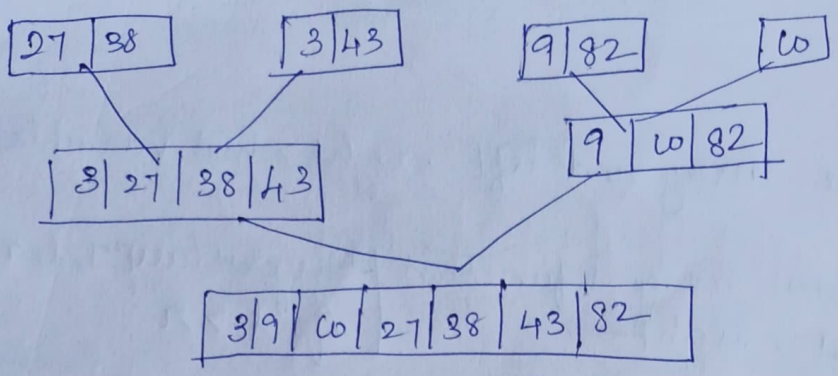
* After dividing the array into smallest units start merging the elements again based on comparison of size of elements.

After dividing the array into smallest units start merging them based on comparision of elements.

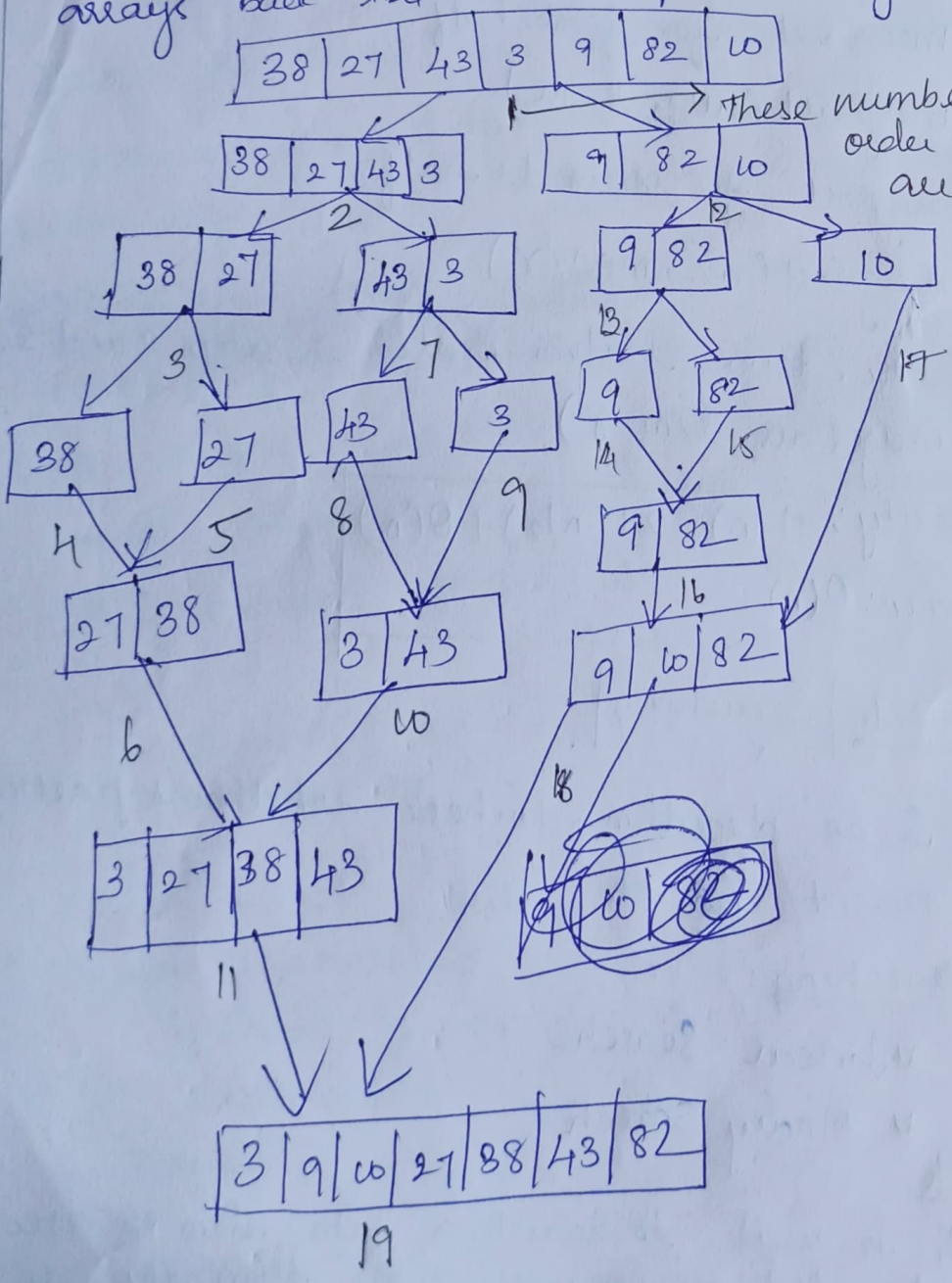
* firstly, compare the element for each list and then combine them into another list in a sorted manner.



* After the final merging, the list looks like this.



If we take closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge process come into action and start merging arrays back till the complete array is merged.



Algorithm

Step 1: Start

Step 2: declare array and left, right, mid variable.

Step 3:: perform merge function [Mergesort(arr[], l, r) if l > r

if left > right return

⇒ find the middle point to divide the array into two halves

middle m = l + (r+1) / 2.

⇒ call mergeSort for first half:

call mergeSort(arr, l, m)

⇒ call mergesort for second half:

call mergeSort(arr, m+1, r)

⇒ Merge the two halves sorted in steps 2 and 3:

Call mergl(arr, l, m, r)

Step 4: stop.

Time complexity: $T(n) = 2T(n/2) + \theta(n)$

Auxiliary Space: $O(n)$

5) Linear Search [Searching].

Searching is an algorithm, to check whether a particular element is present in the list

Types of searching:

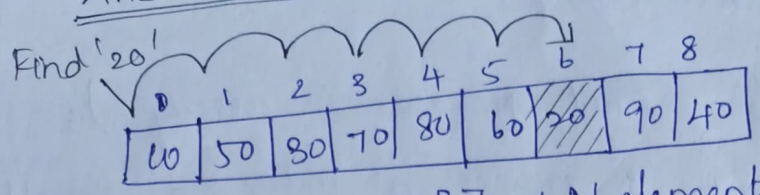
- Linear Search
- Binary Search

Linear Search

Linear Search is used to search a data item in the given list in the sequential manner, starting from the first element. It is also called a sequential search.

* It is easiest searching algorithm.

Linear Search



Given an array arr[] of N elements, the task is to write a function to search a given element X in arr[].

Examples:

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}, x = 110;
(indexes: 0 1 2 3 4 5 6)

output: 6

Explanation: Element x is present at index 6

follow the below idea to solve the problem.

⇒ Iterate from 0 to N-1 and compare the value of every index with x if they match return index.

⇒ follow the given steps to solve the problem

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[].
* If x matches with an element, return the index.
* If x doesn't match with any of the elements, return -1.

```
#include <stdio.h>
int search(int arr[], int N, int x)
{
int i;
for(i=0; i<N; i++)
  if (arr[i]==x)
    return i;
  return -1;
}
int main(void){
int arr[] = {2,3,4,10,40};
```

```
int x = 10;
int N = sizeof(arr);
int result = search(arr, N, x);
(result == -1)
? printf("Element is not present in array");
: printf("Element is present at index %d", result);
return 0;
}
```

o/p

Element is present at index 3.

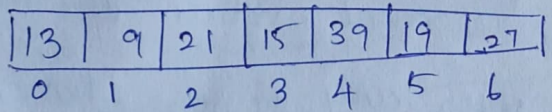
Time complexity: O(N)

Auxillary space: O(1)

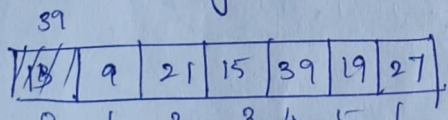
Example of Linear search Algorithm:

Consider an array of size 7 with elements 13, 9, 21, 15, 39, 19 and 27 that starts with 0 and ends with size minus one, 6.

Search element = 39.

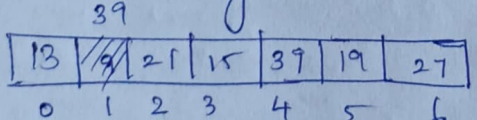


Step 1: The searched element 39 is compared to the first element of an array, which is 13.



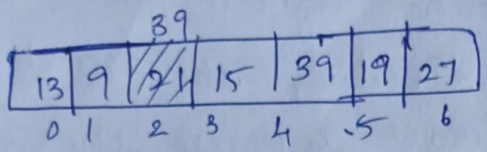
* The match is not found, you now move on to the next element and try to implement a comparison

Step 2: Now, search element 39 is compared to the second element of an array 9.



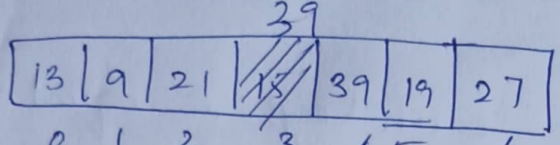
* As both are not match, you will continue the search.

Step 3: Now, each element 39 is compared with the third element, which is 21.



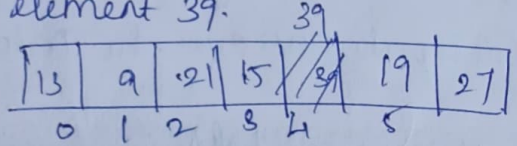
* Again, both the elements are not matching, you move onto the next following element.

Step 4: Next, each element 39 is compared with the fourth element, which is 15



* As both are not matching, you move on to the next element.

Step 5: Next, each element 39 is compared with the fifth element 39.



* A perfect match is found, you stop comparing any further elements and terminate the linear search Algorithm and display the element found at location 4.

Best case =	O(1) operations
Worst Case =	O(n) operations
Average case =	O(n) operations

Application of Linear Search Algorithm:

* Linear search can be applied to both single-dimensional and multidimensional arrays.
* Linear search is easy to implement and effective when the array contains only a few elements.
* Linear search is also efficient when the search is performed to fetch a single search in an unordered list

6) Binary Search:

Binary search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Logn).

Binary search Algorithm

* Begin with the mid element of the whole array as a search key.
* If the value of the search key is equal to the item then return an index of the search key.
* Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
* Otherwise, narrow it to the upper half.
* Repeatedly check from the second point until the value is found or the interval is empty.

Binary search Algorithm can be implemented in the following two ways.

1. Iterative method

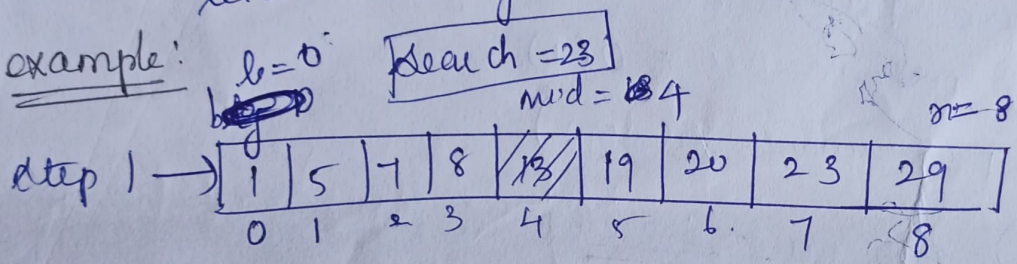
```
binarySearch(arr, x, low, high)
  Repeat till  low = high
      mid = (low + high) / 2
      if (x == arr[mid])
        return mid.
      else if (x > arr[mid]) // x is on the right side.
        low = mid + 1
      else
        high = mid - 1 // x is on the left side
```

Recursive Method (The recursive method follows divide and conquer approach).

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid.
        else if x > arr[mid] // x is on the right side
            return binarySearch(arr, x, mid+1, high)
        else
            return binarySearch(arr, x, low, mid-1)
```

example:

Search = 23



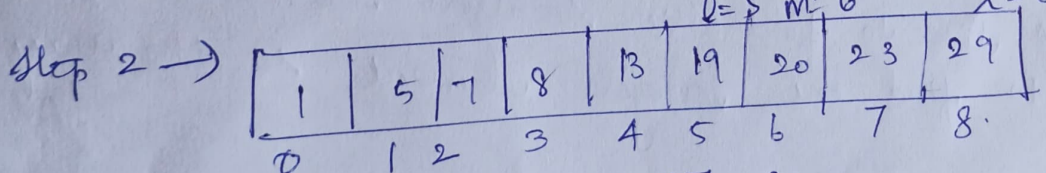
a[mid] = 13

13 < 23.

l = mid + 1 = 4 + 1 = 5

r = 8.

m = (l + r)/2 = 5 + 8/2 = 13/2 = 6.



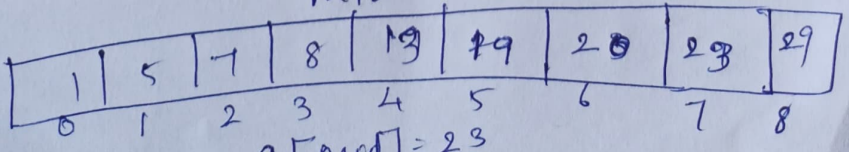
a[mid] = 20

20 < 23

l = mid + 1 = 6 + 1 = 7

r = 8

mid = (l + r)/2 = 15/2 = 7.



a[mid] = 23

23 = 23

loc = mid

return location 7.

Step-by-step Binary Search Algorithm:

1. Compare X with the element.
2. If X matches with the middle element, we return the mid index.
3. Else if X is greater than the mid element, then X can only lies in the right half subarray after the mid element. So we recur for the right half.
4. Else (X is smaller) recur for the left half.

Time complexity: $O(\log n)$
Auxiliary space: $O(\log n)$
Worst case performance = $O(\log n)$
Best-case performance = $O(1)$
Average performance $O(\log n)$

Application of Binary Search:

1. This algorithm is used to search element in a given sorted array with more efficiency.
2. It could also be use for few other additional operations like to find the smallest element in the array or to find the largest element in the array.