Language

- MACHINE LANGUAGE
- ASSEMBLY LANGUAGE
- HIGH LEVEL LANGUAGE
  Eg. C, C++

* C

=> Developed in 1972 by Denis Ritchie in BELL LABS OF AMERICA.

=> In early 80's → TURBO C
In late 80's → ANSI C
(AMERICAN NATIONAL STD. STATE)

* Tokens Used in C

=> The smallest individual unit of a program is known as Token.

1) keywords
2) Strings
3) Identifier
4) Constants
5) Operators

1) keywords :- These are pre-defined, reserved words used in programming that have special meanings to the compiler. There are 43 keywords in C but generally we use 32.

Keywords in C

auto	const	double	float	int	short	struct	unsign
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

2) **Identifiers** :- It refers to the names given to entities such as variables, func^{n}s, structures etc.

Rules for Naming The Identifiers

=> It may be written as a comb^n of alphabets, digits & underscore (_).

=> The first letter should be either a letter or an underscore.

=> Cannot use keywords as identifiers

=> Max^m 31 characters are allowed.

3) **Variables** :- It is a container to hold data which can be changed.

Rules for Naming a Variable

=> Can only have letters, digits & underscore

=> All the rules are same as that of identifiers.

Constants :- It remains const. throughout the entire program. The value cannot be changed.

Literals :- These are the data used for fixed values. They can be directly used in the code.
Eg. 1, 2, 5, 'c' etc.

String :- These are the sequence of characters enclosed in double-quote marks.
Eg. "Hello", "123", "B-39" etc.

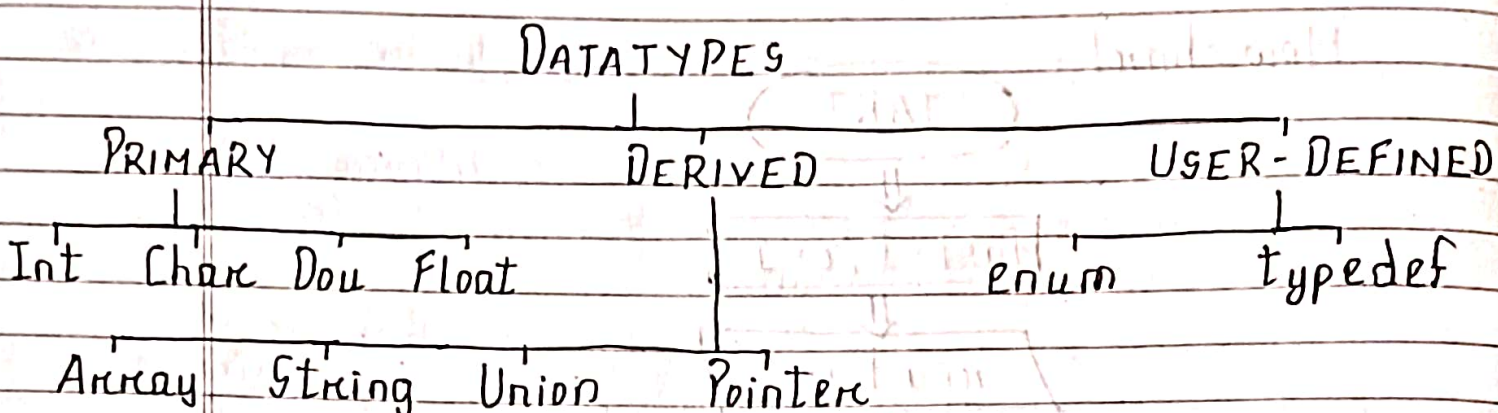
* Datatypes

=> Data types are declaration of variables.

=> This determines the size of data associated with variables.

Basic Data Types

TYPE	SIZE (IN BYTES)	FORMAT SPECIFIERS	RANGE
int	2	%d	-32768 to 32767
char	1	%c	-128 to 127
float	4	%f	
double	8	%lf	
short int	2	%hd	-32768 to 32767



Enum

⇒ It is a user-defined datatype in C.

⇒ Mainly used to assign names to integral consts.

For eg. enum week {Mon, Tue, Wed};
enum week day;

Typedef

⇒ Used to provide existing data types with a new name.

Syntax typedef existing-name alias-name;

Eg. typedef unsigned int unit;

Typecast / Casting Operator

=> If we have

```
int a = 10, b = 4;
float c = a/b;
```

Then output will be 2.00 whereas actual value is 2.50 because int / int = int.

=> If we want the accurate result of c & at the same time do not want to take either a or b as float then we have to use casting operator as follows :-

```
c = (float) a/b;
```

OR

```
c = a / (float) b;
```

=> By the casting operator the datatype of given variable is converted to the specific datatype.

Syntax `(datatype) variable name;`

Operator

=> Symbol which operates the operands mathematically or logically.

4 + 5 = 9

operand → operator

=> It is of 2 types :-

a) A/q involvment
- Unary (single operand)
- Binary (12 operand)
- Ternary (3 operand)

b) A/q operation
- Arithmetic
- Relational
- Logical
- Assignment
- Increment & Decrement
- Conditional
- Bitwise
- Special

1) Arithmetic Operator

=> Those operator involved in arithmetic operation.

Eg - +, -, *, /, %.

a+b a-b a%b
a*b a/b

2) Relational Operator

=> Those operator used for comparison of 2 operands.

Eg. <, >, >=, <=, !=, ==

⇒ Every relational operator return TRUE or FALSE.
If comparison satisfy TRUE otherwise FALSE.

```
a = 10, b = 8;
c = a > b;          c = TRUE
```

3) Logical Operator

3 types :- i) LOGICAL AND (&&)

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

ii) LOGICAL OR (||)

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

iii) LOGICAL NOT (!)

A	!A
0	1
1	0

4) Assignment Operator

=> Those operator which right operand assigns left operand is called assignment operator.

2 Types :- Simple (=)
Compound (+=, -=, *=, %=)

=> In compound assignment operator, 1st left operand operates first right operand with given arithmetic operator.

a = 5
a += 4 a = a+4
a = ? = 9

5) Increment & Decrement

=> Used for increasing & decreasing operand by 1.

++ ⟶ Increment
-- ⟶ Decrement

2 Types :- Post increment & decrement
Pre increment & decrement.

=> When increment & decrement operator comes before operand, it is called pre increment & decrement.

```
++a;
--a;
```

=> The priority of pre increment & decrement is highest than assignment operator. First the operand will increase, then it will be assigned to the operand.

```
a = 10;
b = ++a;
```

Output a = 10
b = 10

```
a = 10;
b = --a;
```

Output a = 9
b = 9

=> When increment & decrement operator comes after operand, it is called post increment & decrement.

```
a++;
b--;
```

=> The priority of assignment operator is higher than post increment & decrement. First the operator will assign, then it will be increased.

a = 5 b = a++	a = 5 b = a--
Output a = 6 b = 5	Output a = 4 b = 5

6) Conditional Operator (? :)

=> Syntax cond^{n} ? statement : statement ;

```
a = 10, b = 20
x = (a > b) ? a : b ;
```

Output x = 20

7) Bitwise Operator

=> Those operator bitwise operates the operands is called bitwise operator.

i) Bitwise AND (&)

```
a = 4, b = 5
c = a & b ;
c = ?
```

c = 4

a = 100
b = 101
$(100)_2$

iii) Bitwise OR (|)

a = 4 , b = 5 ;
c = a | b ;

a = 1 0 0
b = 1 0 1
$(1\ 0\ 1)_2$

c = 5

iii) Bitwise XOR

a = 4 , b = 5
c = a ^ b
(same bits = 0)

a = 1 0 0
b = 1 0 1
$(0\ 0\ 1)_2$

c = 1

iv) Bitwise right shift

Given no. of bits will be eliminated from right side & same no. of zeroes will be added on left side.

a = 4
b = a >> 1

b = 100 >> 1
= 010
= 2

v) Bitwise left shift

Given no. of bits will be put in right side as zeroes.

a = 4
b = a << 1

b = 100 << 1
= 1000
= 8

Control Statements

=> 3 Types :- 1) Decision control
2) Iterative control
3) Jump

1) Decision Control Statement

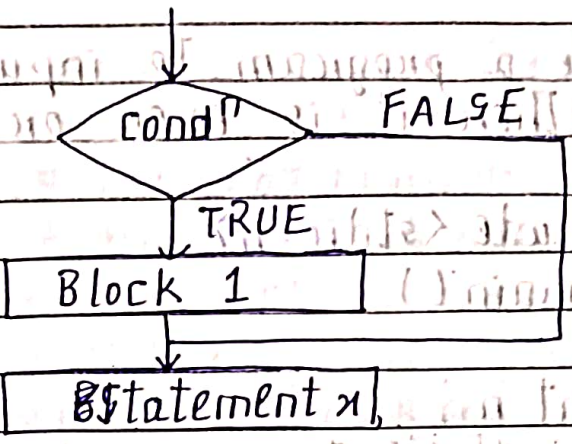
=> Helps to jump from one part of the program to another depending on whether a particular cond^n is satisfied or not.

a) simple if

Syntax

```
if (cond^n)
{
    Statement 1;
    ---
    Statement n;
}
Statement x;
```

Flowchart

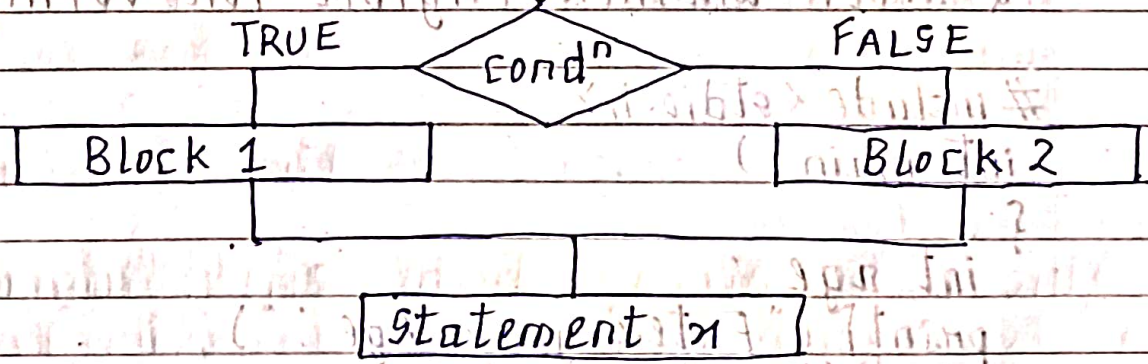


b) if-else

Syntax

```
if (cond^n)
{
  statement block 1;
}
else
{
  statement block 2;
}
statement x;
```

Flowchart



Q- Write a program to input a no. & check whether it is even or odd.

A-
```
#include <stdio.h>
int main()
{
   int n;
   printf("Enter a number : ");
   scanf("%d", &n);
   if (n % 2 == 0)
   {
     printf("%d is even", n);
   }
   else
   {
     printf("%d is odd", n);
   }
   return 0;
}
```

Q- Write a program to input the age of a person & check whether eligible for voting or not.

A-
```
#include <stdio.h>
int main()
{
  int age;
  printf("Enter your Age : ");
  scanf("%d", &age);
```

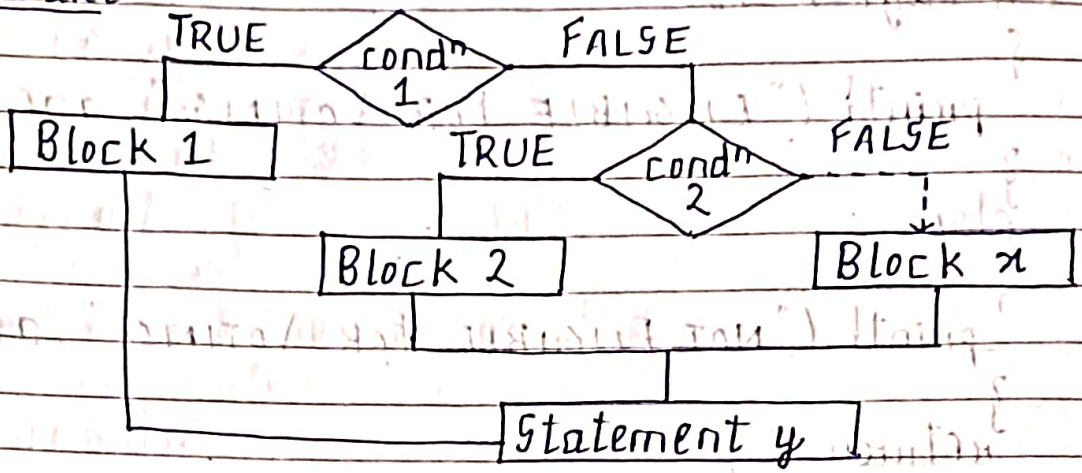
```
if (age >= 18)
{
  printf ("ELIGIBLE FOR VOTING", age);
}
else
{
  printf ("NOT ELIGIBLE FOR VOTING", age);
}
return 0;
}
```

c) if-else-if

Syntax

```
if (cond^n)
{
  statement block 1;
}
else if (cond^n)
{
  statement block 2;
}
------
else
{
  statement block x;
}
statement y;
```

Flowchart



Q- Write a program to input the marks of ~~the~~ 4 ~~diff~~ subjects & find the division acc. to the following cond^n's :-

Avg. Mark	Division
0-29	FAIL
30-49	THIRD
50-59	SECOND
60-above	FIRST

A-

```
#include <stdio.h>
int main()
{
  int a,b,c,d;
  float avg;
  printf("Enter the marks of 4 subjects : ");
  scanf("%d %d %d %d", &a, &b, &c, &d);
  avg = a+b+c+d;
```

```
avg = avg * 4 ;
if (avg <= 29)
{
  printf ("FAIL", avg);
}
else if (avg <= 49)
{
  printf ("THIRD", avg);
}
else if (avg <= 59)
{
  printf ("SECOND", avg);
}
else
{
  printf ("FIRST", avg);
}
return 0;
}
```

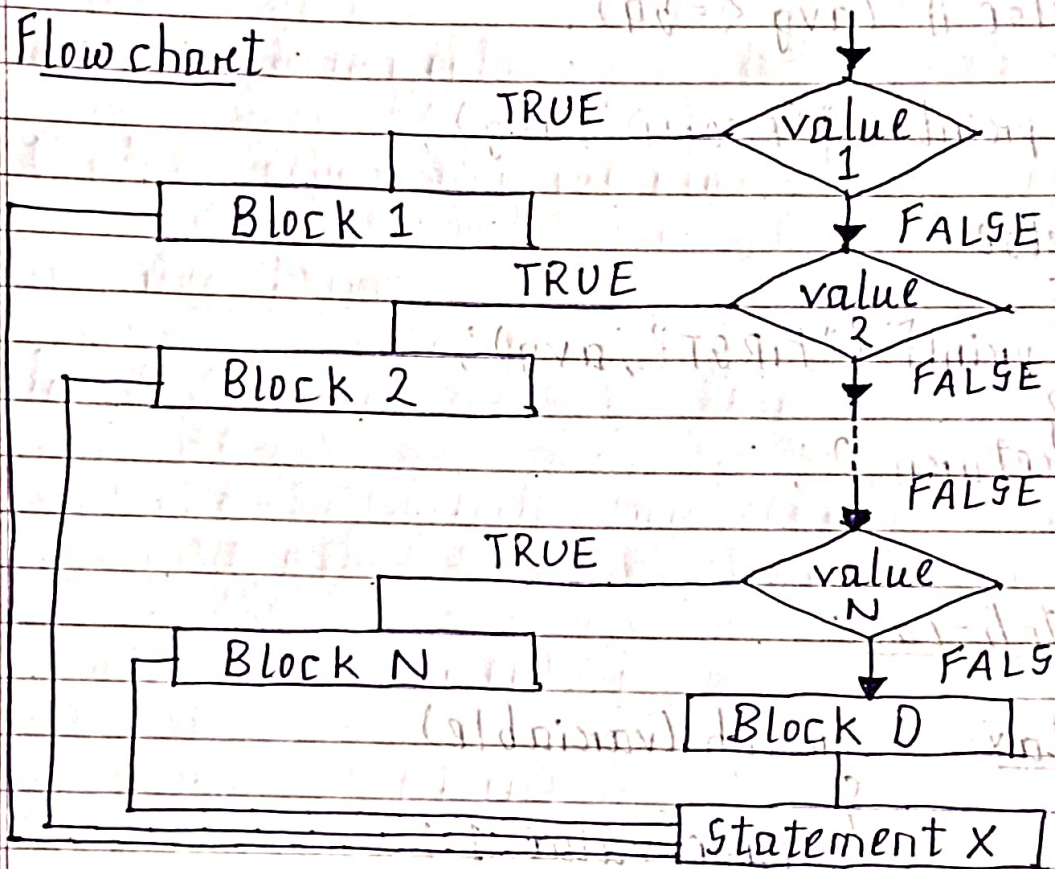
d) switch-case

Syntax

```
switch (variable)
{
  case value 1 :
      statement 1 ;
      break ;
  case value 2 :
      statement 2 ;
      break ;
```

```
-----
case value N:
    statement block N;
    break;
default:
    statement block D;
    break;
}statement X;
```

Flow chart



Q- Write a program to check whether the entered character is vowel or not.

A-

```
#include <stdio.h>
int main()
{
    char ch;
    printf("Enter a Character : ");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'a':
        case 'A':
            printf("%c is VOWEL", ch);
            break;
        case 'e':
        case 'E':
            printf("%c is VOWEL", ch);
            break;
        case 'i':
        case 'I':
            printf("%c is VOWEL", ch);
            break;
        case 'o':
        case 'O':
            printf("%c is VOWEL", ch);
            break;
```

```
case 'u':
case 'U':
    printf("%c is VOWEL", ch);
    break;

default:
    printf("%c IS NOT A VOWEL", ch);
}
return 0;
}
```

2) Iterative Statements

⇒ Used to repeat the execution of a sequence of statements until the specified expression becomes false.

a) for loop

Syntax :-
```
for (initialisation ; criteria ; increment/decrement)
{
    Statement 1 ;
    ---
    Statement n ;
}
```

initialization

It is the initial value of the counter

variable that counts the no. of times the loop will be executed.

Criteria

=> It is the cond" criteria upto which the loop will be executed.

increment / decrement

=> It is the step value how much the counter variable will be increased / decreased after each loop execution to achieve the criteria.

Q- Write a program to input any no. & find out factorial.

A-
```
#include <stdio.h>
int main()
{
  int i, fact = 1, num;
  printf("Enter a no.: ");
  scanf("%d", &num);
  for(i=1; i<=num; i++)
  {
    fact = fact * i;
  }
  printf("Factorial = %d", fact);
  return 0;
}
```

b) **while Loop**

Syntax :

```
statement x ;
while (cond^n)
{
    statement block ;
}
statement y ;
```

Q- Write a program to calculate the sum of no.s from m to n.

A-

```
#include<stdio.h>
int main()
{
  int m, n, i, sum = 0 ;
  printf("Enter the value of m : ");
  scanf("%d", &m);
  i = m ;
  printf("Enter the value of n : ");
  scanf("%d", &n);
  while (i <= n)
  {
     sum = sum + i ;
     i = i + 1 ;
  }
  printf("The sum of no.s from %d to %d = %d",
          m, n, sum);
  return 0 ;
}
```

c) do-while loop

Syntax :-

```
statement x;
do
{
    statement block;
}
while (cond^n);
statement y;
```

Q- Write a program to calculate the avg. of first n numbers.

A-

```
int main()
{
  int n, i=0, sum=0;
  float avg;
  printf("Enter the value of n: ");
  scanf("%d", &n);
  do
  {
     sum = sum+i;
     i = i+1;
  }
  while (i<=n);
  avg = (float) sum/n;
  printf("The sum of first %d no.s = %d", n, sum);
  printf("The avg. of first %d no.s = %.2f", n, avg);
  return 0;
}
```

Break Statement

It is used to come out of the switch statement. If we do not give break, then it will go on executing other cases even if the cond^n is not satisfied until it finds a break.

Continue Statement

It causes the control to return to the conditional statement without executing the statements after it.

Q-1 Write a program to input any no. & display its all the factors.

A-

```
int main ()
{
  int num, i;
  printf ("Enter a number : ");
  scanf ("%d", &num;
  printf ("Factors are : ", num);
  for (i=1; i<=num; i++)
  {
    if (num % i == 0)
    {
      printf ("%d", i);
    }
  }
  return 0;
}
```

Q-2 Write a program to input any no. & find out the sum of the digits.

A-
```
int main ()
{
  int num, i, sum = 0;
  printf ("Enter a number : ");
  scanf ("%d" &num);
  for (i = num ; i != 0 ; i /= 10)
  {
    sum += i % 10 ;
  }
  printf ("The sum of digits : %d", sum);
  return 0 ;
}
```

Q-3 Write a program to input any no. & display its multiplication table.

A-
```
int main ()
{
  int n, i ;
  printf (" Enter a number : ");
  scanf ("%d", &n);
  for (i = 1 ; i <= 10 ; ++i)
  {
    printf (" %d * %d = %d \n", n, i, n * i);
  }
  return 0 ;
```

Q-4 Write a program to input any no. & reverse it.

A-

```
int main()
{
  int n, rev = 0, digit;
  printf("Enter a no. : ");
  scanf("%d", &n);
  while (n != 0)
  {
    digit = n % 10;
    rev = rev * 10 + digit;
    n = n / 10;
  }
  printf("Reverse of the no. : %d", rev);
  return 0;
}
```

Q-5 Write a program to input any no. & check whether it is palindrome or not.

A-

```
int main()
{
  int n, original, rem, rev = 0;
  printf("Enter a number : ");
  scanf("%d", &n);
  original = n;
  while (n != 0)
  {
    rem = n % 10;
```

```
    rev = rev * 10 + rem ;
    n /= 10 ;
  }
  if (original == reverse)
  {
    printf ("%d is a Palindrome Number", original);
  }
  else
  {
    printf ("%d is not a Palindrome Number", original);
  }
  return 0;
}
```

Q-6 Write a program to input any no. & check whether it is prime no. or not.

A-

```
int main ()
{
  int n, i, temp = 0 ;
  printf ("Enter a number : ");
  scanf ("%d", &n);
  for (i = 2 ; i <= n/2 ; ++i)
  {
    if (n % i == 0)
    {
      temp ++ ;
      break ;
    }
```

```
if (temp == 0)
{
printf ("%d is a Prime number", n);
}
else
{
printf ("%d is not a Prime number", n);
}
return 0;
}
```

Q-7 Write a program to input a no. & check whether it is perfect no. or not.

A:

```
int main()
{
int n, i, sum = 0;
printf ("Enter a number: ");
scanf ("%d", &n);
for (i = 1; i <= n/2; i++)
{
  if (n % i == 0)
  {
    sum += i;
  }
}
if (sum == n && n > 0)
{
printf ("%d is a Perfect no.", n);
}
```

```
else
{
printf("%d is not Perfect number", n);
}
return 0;
}
```

Q-8 Write a program to input base & exponent & find its power.

A-
```
int main()
{
int n, bs, exp, pow = 1, i;
printf("Enter base & exponent : ");
scanf("%d %d", &bs, &exp);
for (i=1; i<= exp; i++)
{
pow = pow * bs;
}
printf("Power = %d ", pow);
return 0;
}
```

Q-9 Write a program to display first 10 nos of Fibonacci series.

A-
```
int main()
{
int n = 10, i, a = -1, b = 1, c;
printf("First 10 terms of Fibonacci series : ");
```

```
for(i=1 ; i<=n ; ++i)
{
   c = a+b ;
   printf ("%d," , c) ;
   a = b ;
   b = c ;
}
return 0 ;
}
```

Q-10 Write a program to input any year & check whether it is leap year or not.

A-

```
int main ()
{
 int yr ;
 printf("Enter a year : ") ;
 scanf ("%d" &yr) ;
 if ((yr %. 4 == 0 && yr %. 100 != 0 ) || (yr %. 400 == 0)
 {
  printf ("%d is a leap year", yr) ;
 }
 else
 {
  printf("%d is not a leap year ", yr) ;
 }
 return 0 ;
}
```

* Functions

=> Func^n is a self-contained program segment that carries out some specific user defined task.

=> It is a named block of statement that performs certain user defined task.

=> A Func^n is a complete program in itself.

=> All C programs start at main Func^n (). The main Func^n called library Func^n as well as other user defined Func^n.

=> The Func^n which calls other Func^n's is known as the calling Func^n & the Func^n which is being called by the calling Func^n is known as the called Func^n.

Why are Func^n's Needed?

→ Dividing the program into separate well-defined Func^n's facilitates each Func^n to be written & tested separately. This simplifies the process of getting the total program to work.

→ Understanding, coding & testing multiple separate Func^n's is easier than doing the same for one big Func^n.

Func^n declaration

Syntax: return-data-type Func^n-name (data-type variable 1, data-type variable 2, ...);

=> return-data-type specifies the datatype of the value that will be returned to the calling func^n as a result of the processing performed by the called func^n.

=> Func^n-name is valid name for the func^n. Naming a Func^n follows the same ~~value~~ rules that are followed while naming variables.

=> (data-type variable 1, data-type variable 2...) is a list of variables of specified data types.

Func^n definition

Syntax: return type Func^n name (datatype variable 1, datatype variable 2,...)

{

statements

---

return (variable);

}

Func^n call

Syntax :- Func^n name (variable 1, variable 2, ...);

=> Func^n name & the no. & the type of arguments in the func^n call must be same as that given in the func^n declaration & func^n definition.

=> When the func^n is called the control jumps to the func^n definition part with the value of actual parameters.

Passing parameters to Func^ns

1) Call by Value

=> The value of the variables are passed by the calling func^n to the called func^n.

=> When arguments are passed by value, the called func^n creates new variables of the same datatype as the arguments passed to it.

=> The values of the arguments passed by the calling func^n are copied into the newly created variables.

=> Values of the variables in the calling func^n remains unaffected when the arguments are passed using the call-by-value technique.

Advantage : Arguments can be passed as variables literals or expressions

Disadvantage : Copying data consumes additional storage space.

2) Call By Reference

=> The addresses of the variables are passed by the calling func^n to the called func^n.

=> Here, we declare the func^n parameters as references rather than normal variables

=> A func^n recieves an implicit reference to the argument, rather than a copy of its value.

=> Thus, the func^n can modify the value of the variable & that change will be reflected in the calling func^n as well.

Local Variable

=> The variable which is accessible by the func" in which it has been declared is called local variable.

=> No other func" except that particular func" can access it.

Global Variable

=> The variable which is accessible by almost all the func"s of the program is known as global variable.

=> It is declared outside of all func"s.

Recursive Function

=> The func" which falls under the process of recursion is known as recursive func".

=> Recursive func" is called by itself.

=> Here the calling func" & the called func" both are the same single func".

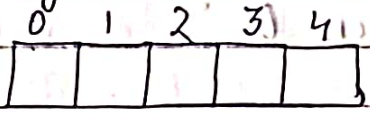
* Arrays

=> It is a data structure in which similar types of elements are stored in contagious way.

=> In C, it is of 2 types :
i) Single dimensional array
ii) Multi dimensional array

Syntax datatype arrayname [size];

Eg. int no [5];



The size denotes the max^m no. of elements that can be stored in the array.

Single Dimensional Array

=> Single index is used to access the element.

=> Index = 0 to size - 1.

Q-1 Write a program to input any 10 no.s to an array & display them.

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

```
int a[10], i ;
printf ("Enter the elements : ");
for(i=0 ; i<10 ; i++)
{
  scanf ("%d", &a[i]);
}
printf ("Output : ");
for (i=0 ; i<10 ; i++)
{
  printf ("%d\t ", a[i]);
}
return 0 ;
}
```

Q-2) Write a program to input any 10 no.s to an array & find the sum of all no.s

A-

```
#include <stdio.h>
int main()
{
 int a[10], i, sum = 0 ;
 printf ("Enter the elements : ");
 for (i=0 ; i<10 ; i++)
 {
   scanf ("%d", &a[i]);
 }for (i=0 ; i<10 ; i++)
 {
    sum += a[i] ;
 }
```

* Storage Class

=> Storage class decides in which location the variable would be stored.

=> The storage class informs us about the following facts :-

1) Storage → Where the value would be stored.

2) Default initial value → What would be the value of the variable, if we don't assign anything.

3) Scope → Where the value of the variable is available.

4) Life → How long the value exists in the variable.

Syntax storage class datatype variable name;

=> It is of 4 types :- 1) Automatic
2) Static
3) Register
4) External

1) Automatic

=> keyword → 'Auto'

=> Default storage class of a variable.

i) Storage → Memory

ii) Default initial value → Garbage value (any unexpected value)

iii) Scope → Local to the block in which it has been defined.

iv) Life → Till the control remains in the block in which it has been defined.

```
void main()
{
  auto int i=1;
  {
    auto int i=2;
      auto int i=3;
      printf("%d\n",i);
    }
  printf("%d\n",i);
  }
  printf("%d\n",i);
  getch();
}
```

Output

```
3
2
1
```

2) Static

=> Syntax ⟶ static int i;

ii) Storage → Memory

iii) Default initial value → 0

iii) Scope → Local to the block in which it has been defined.

iv) Life → The value of the variable persist bet" diff. func" call.

```
void main()
{
 void increment ();
 increment ();
 increment ();
 increment ();
 getch ();
}
void increment ()
{
  static int i = 1;
  printf ("%d\n", i);
  i++;
}
```

Output

1
2
3

The value of the static variable does not erase from the memory. It persist in memory even if the func" is no longer active. When it enters into the same func" once again, it would take the value that it had last time.

3) Register

keyword → register
syntax → register int c;

i) Storage → CPU

ii) Default initial value → Garbage value

iii) Scope → Local to the block in which it has been defined.

iv) Life → Till the control remains in the block in which it has been defined.

The variables store inside the CPU register are accessed in faster way than the variables stored in a memory register.

The frequently used variables of a program are declared as register storage class so that they can be accessed easily & improve the speed of execution.

We cannot declare all types of variables as register storage class because generally the CPU registers are 16 bit register. If we declare a lengthy variable then they would be stored in memory register & would be treated as automatic storage class.

4) External

Keyword → extern
Syntax → extern int i;

i) Storage → Memory

ii) Default initial value → 0

iii) Scope → Global

iv) Life → Until the execution of program does not comes to an end.

```
int x;
void main()
{
 void increment();
 void decrement();
 printf("%d\n", x);
 increment();
 increment();
 decrement();
 decrement();
 getch();
}
void increment()
{
  x++;
```

Output:

```
0
1
2
1
0
```

```
    printf ("%d\n", x);
}
void decrement ()
{
    x--;
    printf ("%d\n", x);
}
```

Declaring a variable as external would save a lot of memory space & would avoid unnecessary passing of the variable as parameters into diff func"s. Once we declare them, they remain in memory throughout the entire program.

* Sorting

1) Bubble Sort

=> Sorts the array elements by repeatedly moving the largest element to the highest index pos" of the array segment.

=> Consecutive adjacent pairs of elements in the array are ~~combined~~ compared with each other

=> Called Bubble sorting because elements 'bubble' to the top of the list.

Technique

STEP I :- A[0] & A[1] are compared, then A[1] with A[2] A[2] with A[3] & so on. Finally, A[N-2] is compared with A[N-1]. Involves n-1 comparisons & places the biggest element at the highest index of the array.

STEP II Again the same process will be repeated till n-2 comparisons & places second biggest element.

STEP III After n-1 A[0] & A[1] are compared so that A[0] < A[1]. After this, all elements are sorted.

Algorithm

1) Repeat for i = 0 to n-1
2) Repeat for j = 0 to n-i
3) If A[J] > A[j+1]
   swap a[j] & a[j+1]
   [End of inner loop]
   [End of outer loop]
4) Exit

```
#include <stdio.h>
int main()
{
  int a[50], i, j, temp, n;
  printf("Enter the no. of elements in an
          array : ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  for(i=0; i<n; i++)
  {
    for(j=0; j<n-i-1; j++)
    {
      if(a[j] > a[j+1])
      {
        temp = a[j];
        a[j] = a[j+1];
        a[j+1] = temp;
      }
    }
  }
  printf("Sorted array is : ");
  for(i=0; i<n; i++)
  {
    printf("%d\t", a[i]);
  }
  return 0;
}
```

2) Insertion Sort

=> The sorted array is built one element at a time.

=> The main idea is that it inserts each item into its proper place in the final list.

Technique

Let there are n elements in the array. Initially the index 0 is in the sorted set. Rest of the elements are in unsorted set.
The 1st element of the unsorted set has array index 1
During each iteration of the algorithm, the 1st element in the unsorted set is picked up & inserted into the correct pos^{n} in the sorted set.

Algorithm

1) Repeat for I = 2 to n.
   k = A[I]
   J = I - 1
2) while J > 0 && A[J] > k
   A[J+1] = A[J]
   J = J - 1
   A[J+1] = k
3) Exit

Q- WAP to sort an array using insertion sort.

A-

```
#include <stdio.h>
int main()
{
  int a[50], i, j, n, temp;
  printf("Enter the no of elements in an
          array : ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  for(i=1; i<n; i++)
  {
    temp = a[i];
    j = i-1;
    while((temp<a[j] && (j>=0))
    {
      a[j+1] = a[j];
      j--;
    }
    a[j+1] = temp;
  }
  printf("Sorted array : ");
  for(i=0; i<n; i++)
  {
    printf("%d\t", a[i]);
  }
}
```

3) Selection Sort

=> Works by finding the smallest value & placing it in the 1st pos". It then finds the 2nd smallest value & places it in 2nd pos" & so on.

Algorithm

1) Repeat for (I = 0 to n-1
   pos = I
2) Repeat for J = I+1 to n
3) If A[pos] > A[J]
   pos = J
4) If pos != I
   temp = A[I]
   A[I] = A[pos]
   A[pos] = temp

Q- WAP to sort an array using selection sort.

A-
```
#include <stdio.h>
int main()
{
  int a[50], i, j, k, temp, pos;
  printf("Enter the no. of elements in an
        array : ");
  scanf("%d", &k);
  printf("Enter the elements : ");
```

```
for (i = 0 ; i<n ; i++)
{
  scanf ("%d", &a[i]);
}
for (i=0 ; i<n-1 ; i++)
{
   pos = i ;
  for (j = i+1 ; j<n ; j++)
  {
    if (a[pos] > a[j])
        pos = j ;
  }
  if (pos != i)
   {
        temp = a[i] ;
        a[i] = a[pos] ;
        a[pos] = temp ;
    }
}
printf ("Sorted arrays : ");
for (i=0 ; i<n ; i++)
{
   printf ("%d\t", &a[i]);
}
return 0 ;
}
```

① Recursion

Q- WAP to calculate the factorial of a no.

A-
```
#include <stdio.h>
int fact (int);
int main ()
{
    int n, fact;
    printf ("Enter the no: ");
    scanf ("%d", &n);
    fact = fact (n);
    printf ("Factorial: %d", fact);
    return 0;
}
int fact (int k)
{
    if (k == 1)
        return 1;
    else
        return (n * fact (n-1));
}
```

Q- WAP to calculate GCD of 2 nos using recursion

A-
```
#include <stdio.h>
int GCD (int, int);
int main ()
{
```

```
    int a, b, res;
    printf("Enter 2 nos : ");
    scanf("%d %d", &a, &b);
    res = GCD(a,b);
    printf("GCD : %d", res);
    return 0;
}
int GCD(int x, int y)
{
    int rem;
    rem = x % y;
    if (res == 0)
        return y;
    else
        return(GCD(y, rem));
}
```

Q- ~~Write~~ WAP to print the fibbonacci series using recursion.

A-

```
#include <stdio.h>
int fibonacci(int);
int main()
{
    int n, c=0, res;
    printf("Enter the no. of terms : ");
    scanf("%d", &n);
    printf("Fibonacci series : ");
    for(c=0; c<n; c++)
    {
```

```
        res = fibonacci(n);
        printf("%d\t", res);
    }
    return 0;
}
int fibonacci(int k)
{
    if (k == 0)
        return 0;
elseif (k == 1)
        return 1;
    else
        return (fibonacci(n-1) + fibonacci(n-2);
    }
```

② String

Q- Write a program to find the length of string.

A-
```
#include <stdio.h>
int main()
{
    char str[100], c=0, length;
    printf("Enter the string : ");
    gets(str);
    while (str[c] != '\0')
        c++;
    length = c;
    printf("Length : %d", length);
} return 0;
```

Q- WAP to convert the lower case characters of a string into upper case.

A-
```
#include <stdio.h>
int main()
{
 char str[100], upper_str[100];
 int i=0;
 printf("Enter the string: ");
 gets(str);
 while(str[i]!='\0')
 {
  if(str[i]>='a' && str[i]<='z')
        upper_str = str[i]-32;
  else
      upper_str = str[i];
   i++;
 }
 upper_str[i] = '\0';
 printf("Upper case: ");
 puts(upper_str);
 return 0;
}
```

Q- WAP to find whether a string is a palindrome or not.

A-
```
#include <stdio.h>
int main()
```

```
{
char str[100];
int i=0, j, length=0;
printf("Enter the string: ");
gets(str);
while(str[i]!='\0')
{
    length++;
    i++;
}
i=0;
j=length-1;
while(i<=length/2)
{
    if(str[i]==str[j])
    {
        i++;
        j--;
    }
    else
        break;
}
if(i>=j)
    printf("PALINDROME");
else
    printf("NOT A PALINDROME");
return 0;
}
```

Q- WAP to reverse a string.

A-

```
#include <stdio.h>
int main()
{
  char str[100];
  int len = 0, i;
  printf("Enter a string : ");
  gets(str);
  while (str[len] != '\0')
      len++;
  for (i = 0; i < len/2; i++)
  {
     char temp = str[i];
     str[i] = str[len-i-1];
     str[len-i-1] = temp;
  }
  printf("Reversed string : ");
  puts(str);
  return 0;
}
```

③ Arrays

Q- WAP to enter n no. of elements in an array & reverse it.

A-
```
#include <stdio.h>
int main()
{
  int a[50], i, n;
  printf("Enter the no. of elements in an
        array : ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  printf("Reversed array : ");
  for(i=n-1; i>=0; i++)
  {
    printf("%d\t", a[i]);
  }
  return 0;
}
```

Q- WAP to enter n no.s of elements in an array & copy it to another array.

A-

```
#include <stdio.h>
int main()
{
  int a[50], b[50], n, i;
  printf("Enter the no of elements in an
          array : ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  for(i=0; i<n; i++)
  {
      b[i] = a[i];
  }
  printf("Copied array : ");
  for(i=0; i<n; i++)
  {
    printf("%d\t", b[i]);
  }
  return 0;
}
```

Q- WAP to input the nos in a 3 x 4 matrix & find the sum of each row separately.

A-

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

```
int a[3][4], sum[3] = 0, i, j;
printf("Enter the elements of 3 x 4 matrix:")
for(i=0; i<3; i++)
{
  for(j=0; j<4; j++)
  {
    scanf("%d", &a[i][j]);
  }
}
for(i=0; i<3; i++)
{
  for(j=0; j<4; j++)
  {
    sum[i] += a[i][j];
  }
}
for(i=0; i<3; i++)
{
  printf("Sum of each row: %d", sum[i]);
}
return 0;
}
```

Q- WAP to input the nos in a 3x4 matrix & find the sum of each column separately.

A-

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

```
int a[3][4], sum[4] = 0 ; i, j ;
printf ("Enter the elements in 3 x 4 matrix : ")
for (i = 0 ; i < 3 ; i++)
{
 for (j = 0 ; j < 4 ; j++)
 {
  scanf ("%d", &a[i][j]);
 }
}
for (j = 0 , j < 4 ; j++)
 {
  for (i = 0 ; i < 3 ; i++)
  {
   sum[j] += a[i][j] ;
  }
 }
for (j = 0 ; j < 4 ; j++)
 {
  printf ("Sum of each column : %d ", sum[j]);
 }
return 0 ;
}
```

Q- WAP to input the no.s in 3 x 3 matrix & find its diagonal elements.

A-

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

```
int a[3] b[3], i, j;
printf ("Enter the elements in 3 x 3 matrix : ");
for (i=0 ; i<3 ; i++)
{
  for (j=0 ; j<3 ; j++)
  {
    scanf ("%d", &a[i][j]);
  }
}
printf ("Diagonal elements : ");
for (i=0 ; i<3 ; i++)
{
  printf ("%d\t", a[i][i]);
}
return 0;
}
```

Q- WAP to input the nos in 3x3 matrix & transpose it

A-

```
#include <stdio.h>
int main()
{
  int a[3][3], i, j ; b[3][3] ;
  printf ("Enter the elements in 3 x 3 matrix :");
  for (i=0 ; i<3 ; i++)
  {
    for (j=0 ; j<3 ; j++)
    {
      scanf ("%d ", &a[i][j] );
```

```
    }
    for (i=0 ; i<3 ; i++)
    {
      for (j=0 ; j<3 ; j++)
      {
          b[j][i] = a[i][j] ;
      }
      printf ("\n");
    }
  printf ("Transpose of the matrix : ");
  for (i=0 ; i<3 ; i++)
  {
    for (j=0 ; j<3 ; j++)
    {
      printf ("%d ", b[i][j]);
    }
    print ("\n");
  }
  return 0;
}
```

Q- WAP to enter 2 matrix. Add them & display the resultant matrix.

A-

```
#include <stdio.h>
int main ()
{
  int a[50][50] , b[50],[50], rows, col, i, j;
  printf ("Enter the no. of rows & col : ");
  scanf ("%d %d", &i, &j);
```

```
printf("Enter the elements for 1st matrix: ");
for(i=0; i
for(rows=0; rows<i; rows++)
{
  for(col=0; col<j; col++)
  {
    scanf("%d", &a[rows][cols]);
  }
}
printf("Enter the elements of 2nd matrix: ");
for(rows=0; rows<i; rows++)
{
  for(col=0; col<j; col++)
  {
    scanf("%d", &b[rows][col]);
  }
}
int res[50][50];
printf("Resu
for(rows=0; rows<i; rows++)
{
  for(col=0; col<j; col++)
  {
    res[i][j] = a[i.
    res[rows][col] = a[rows][col] + b[rows][col];
  }
  printf("Resultant matrix: ");
  for(rows=0; rows<j; rows++)
  {
    for(col=0; col<j; col++)
    { printf("%d\t", res[rows][col]);
}}
```

* Pointer

=> Pointer is a variable which can store the address of another variable.

=> An ordinary value can store an ordinary value but a pointer variable would store a value which is the address of another variable.

Syntax datatype * pointer variable name ;

Eg. int *p ; int no, i ;
char *q ; char ch, letter ;
float *r ; float sal, avg ;

=> The datatype of an ordinary variable is decided by the type of value it would stored but the datatype of the pointer variable is decided by the type of variable to which it points (whose memory address it would stored).

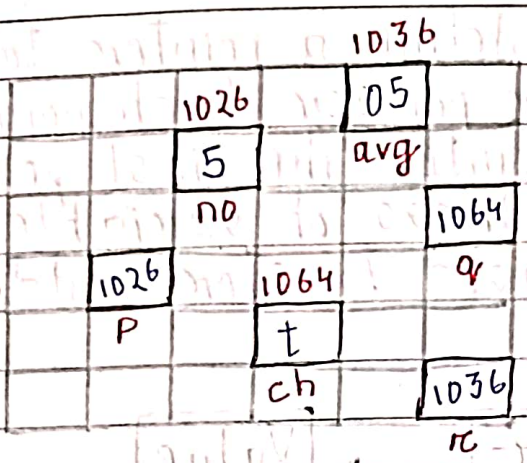
To assign the address of a variable to a pointer of a variable

Syntax →

pointer variable = & ordinary variable ;

Eg. p = &no ; → *p = no ;
q = &ch ; → *q = ch ;
r = &avg ; → *r = avg ;

Referencing a variable — Defencing a variable



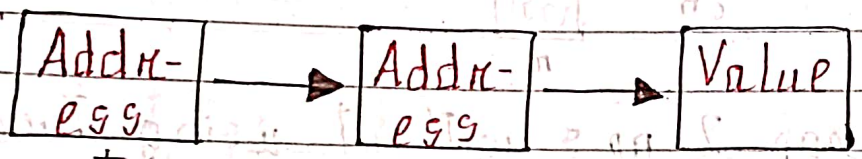
Q- WAP to swap 2 no.s without using 3rd variable & using pointers.

A-
```
void main()
{
 int a, b;
 int *A1, *B1;
 A1 = &a;
 B1 = &b;
 printf("Enter the values of a & b : ");
 scanf("%d %d", &a, &b);
 printf("Before swap : a = %d, b = %d", a, b);
   *A1 = *A1 + *B1;
   *B1 = *A1 - *B1;
   *A1 = *A1 - *B1;
```

printf ("After swap : *a = %d, *b = %d",
*a, *b);
getch ();
}

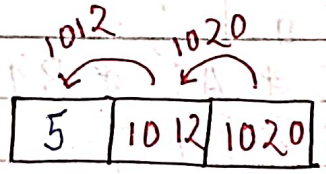
Pointer to Pointer

In C, we can also define a pointer to store the address of another pointer. Such pointer is known as double pointer. The first pointer is used to store address of variable whereas second pointer is used to store address of first pointer.



Syntax datatype ** pointer variable

```
int i;
int *p;
int **q;
p = &i;  →  *p = i
q = &p;  →  *q = p
```



*(*q) = *p

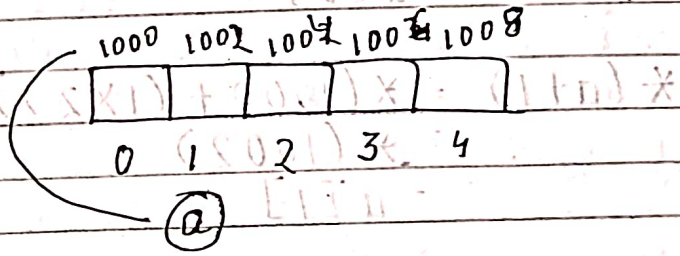
=> **q = *p = i

```
#include <stdio.h>
void main()
{
 int a = 10;
 int *p;
 int **pp;
 p = &a;
 pp = &p;
 printf("Address of a = %x", p);
 printf("Address of p = %x", pp);
 printf("Value stored at p = %d", *p);
 printf("Value stored at pp = %d", **pp);
}
```

Array to Pointer

We cannot use a pointer to point an array name because the array name suggest the base address of the array from where it start.

```
int a[5];
int *p;
p = &a;
```



We cannot do so because here a means 1000. If we do p = &a, it will be p = &1000, which will be like pointing to a const.

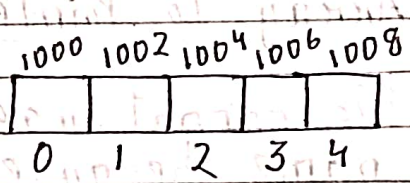
We can use a pointer to point an array element.

p = &a[0] => *p = a[0]
p = &a[1] => *p = a[1]

Also we can use the pointer directly to point array element without taking a pointer variable as the array name is an address itself.

Syntax → *(array name + index no)
= array name [index no];

Eg. *(a+0) = a[0];
*(a+1) = a[1];
*(a+2) = a[2];



*(a+0) = *(1000 + (0 x 2))
= *(1000)
= a[0]

*(a+1) = *(1000 + (1 x 2))
= *(1002)
= a[1]

*(a+2) = *(1000 + (2 x 2))
= *(1004)
= a[2]

Pointer to String

Pointer will treat string as it treats other types of arrays because string is a character array.

Without pointer variables

```
void main()
{
char city[] = "Bhubaneswar";
int i;
for(i=0; city[i]!='\0'; i++)
{
  printf("%c\t", city[i]);
  printf("%c\n", *(city+i));
}
getch();
}
```

Output

B
H
U
B
A
N
E
S
W
A
R

Output

B	H	U	B	A	N	E	S	W	A	R

Using pointer variable

```
void main()
{
char city[] = "Bhubaneswar");
char *p;
p = city;
while(*p!='\0')
{
  printf("%c", *p);
  p++;
}
getch();
}
```

Pointer to Func^n

A pointer can point to a Func^n in C. However the declaration of pointer variable must be same as Func^n.

```
#include <stdio.h>
int add();
int main();
{
   int res;
   int (*ptr)();
   ptr = &add;
   res = (*ptr)();
   printf("Sum : %d", res);
}
int add()
{
   int a, b;
   printf("Enter 2 no.s : ");
   scanf("%d %d", &a, &b);
   return a+b;
}
```

* Structure

=> Structure is a user-defined datatype. It is used to group diff. types of datatype in a single group.

Syntax → struct structurename
{
datatype item 1 ;
datatype item 2 ;
datatype item 3 ;
}

Eg. struct st
{
int roll ;
char name [10] ;
int age ;
}

=> To access the structure we have to use the variable name & a dot (.) operator with each member name.

```
void main ()
{
  struct emp
  {
    char name [10] ;
```

```
    int age;
    float salary;
};
struct emp e;
printf("Enter the name, age & salary : ");
scanf("%s %d %f", &e.name, &e.age,
        &e.salary);
printf("name = %s\n age = %d\n salary = %.2f\n
        e.name, e.age, e.salary);
getch();
}
```

Structure & Function

=> We can pass the structure variable as parameter into a func^n.

=> It can be done in 2 ways :-
i) By passing individual ~~stru~~ members
ii) By passing the entire structure

Array of Structure

We can declare a array of structure variables where each element of the array is a structure variable of same structure type.

```
void main()
{
  struct st
  {
    int roll;
    char name[10];
    int mark;
  };
  struct st s[3];
  int i;

  for(i=0; i<3; i++)
  {
    printf("Enter the roll, name & mark: ");
    scanf("%d %s %d", &s[i].roll, &s[i].
          name, &s[i].mark);
  }
  printf("Output: ");

  for(i=0; i<3; i++)
  {
    printf("roll = %d\t name = %s\t mark=%d\n",
          s[i].roll, s[i].name, s[i].mark);
  }
}
```

File Handling

Through program files we can handle a data file. We can create, a file, can store data in it, We can read data from it & also can close the file. This is called file handling.

fopen() → used to open a file
fclose() → used to close a file.
getc() → used to read a char from a file.
putc() → used to store a char in a file.
getw() → used to read an integer in a file.
putw() → used to store an integer in a file.
fscanf() → used to read a set of data values from a file.
fprintf() → used to store a set of data values in a file

To work upon a file of disk we need a communication link bet^n the memory & file. This link can be established by creating a buffer in memory. (Buffer is a specified memory area).

Syntax → FILE * buffername;

Mode

It is the purpose of opening a file.

"r" → always opens an existing file for reading only.

"w" → always creates a new file for writing only. If any file exists in the same name then it will be destroyed & a new file is created.

"a" → always opens an existing file for appending. (adding at the end only).

"rt" → always opens an existing file for both reading & writing.

"wt" → always creates a new file for both writing & reading.

"at" → always opens an existing file for both appending & writing.

Before opening a file in a particular mode, we have to close the currently existing mode as the buffer is the communication link bet^n file & memory.

Q- WAP to create a file & store 10 no.s in it. Display the contents of the file.

A-

```
void main()
{
  FILE *ptr;
  int no, i;
  ptr = fopen("Num.txt", "w");
  for(i=1; i<=10; i++)
  {
     printf("Enter a no :");
     scanf("%d", &no);
     putw(no, ptr);
  }
  fclose(ptr);
  printf("Reading data from the file\n");
  ptr = fopen("Num.txt", "r");
  while((no==getw(ptr))!=EOF)
  {
    printf("%d\t", no);
  }
  fclose(ptr);
}
```

Dynamic Memory Allocation

3 func's :- i) malloc()
ii) calloc()
iii) free()

a) Malloc()

=> It is used to ~~store~~ allocate memory dynamically.

Syntax :-

```
datatype * pointer variable = (datatype *)
        malloc (n * sizeof (datatype));
                  ↑ no. of elements
```

Here we need to typecast because the malloc() by default create void type of memory. So, we need to convert it to the type of our requirement by typecast operator.

b) Calloc()

=> Here we take 2 arguments.

Syntax :-

```
datatype * pointer variable = (datatype *)
        calloc (n, sizeof (datatype));
```

c) Free()

It is used to deallocate or release the memory space that is allocated dynamically.

Syntax :-

```
Free (pointer variable);
```

Difference bet^{n} Malloc() and Calloc()

	Malloc()	Calloc()
1	Creates a single block of memory of a specific size.	Assigns multiple blocks of memory to a single variable.
2	No. of arguments is 1.	No. of arguments is 2.
3	mallo() is faster.	calloc() is slower.
4	Has high time efficiency.	Has low time efficiency.
5	Indicates memory allocation	Indicates contiguous allocation

Merge Sort

Merge sort is a sorting algorithm that uses the divide, conquer & combine algorithmic paradigm

Divide means partioning the n-element array to be sorted into 2 sub-arrays of $n/2$ elements.

Conquer means sorting the 2 sub-arrays recursively using merge sort.

Combine means merging the 2 sub-arrays of size $n/2$ to produce the sorted array of n elements.

Algorithm

MERGE (ARR, BEG, MID, END)

1) SET I = BEG, J = MID + 1, INDEX = 0
2) Repeat while (I <= MID) AND (J <= END)
   IF ARR[I] < ARR[J]
      SET TEMP[INDEX] = ARR[I]
      SET I = I + 1
   ELSE
      SET TEMP[INDEX] = ARR[J]
      SET J = J + 1
   [END OF IF]

```
    SET INDEX = INDEX + 1
[END OF LOOP]
3) [Copying the remaining elements of right sub-array]
    IF I > MID
        Repeat while I <= END
            SET TEMP [INDEX] = ARR [J]
            SET INDEX = INDEX + 1, SET J = J + 1
        [END OF LOOP]
[Copy the remaining elements of left sub-array]
    ELSE
        Repeat while I <= MID
            SET TEMP [INDEX] = ARR [I]
            SET INDEX = INDEX + 1, SET I = I + 1
        [END OF LOOP]
    [END OF IF]
4) [Copy the contents of TEMP back to ARR]
    SET k = 0
5) Repeat while k < INDEX
        SET ARR [k] = TEMP [k]
        SET k = k + 1
    [END OF LOOP]
6) END

MERGE-SORT (ARR, BEG, END)

1) IF BEG < END
        SET MID = (BEG + END) / 2
        CALL MERGE-SORT (ARR, BEG, MID)
        CALL MERGE-SORT (ARR, MID + 1, END)
        MERGE (ARR, BEG, MID, END)
    [END OF IF]
```

Q- Write a program to implement merge sort.

A-
```
#include <stdio.h>
void merge (int a[], int, int, int);
void merge_sort (int a[], int, int);
void main()
{
int arr[size], i, n;
printf ("Enter the no. of elements in array : ");
scanf ("%d", &n);
printf ("Enter the elements : ");
for (i=0; i<n; i++)
{
  scanf ("%d", &arr[i]);
}
merge_sort (arr, 0, n-1);
printf ("The sorted array : ");
for (i=0; i<n; i++)
{
  printf ("%d \t", arr[i]);
}
void merge (int arr[], int beg, int mid, int end)
{
int i = beg, j = mid+1, index = beg, temp[size], k;
while (i<=mid) && (j<=end)
{
  if (arr[i] < arr[j])
  {
    temp[index] = arr[i];
    i++;
  }
```

```
    else
    {
        temp[index] = arr[j];
        j++;
    }
    index++;
}
if (i > mid)
{
    while (j <= end)
    {
        temp[index] = arr[j];
        j++;
        index++;
    }
}
else
{
    while (i <= mid)
    {
        temp[index] = arr[i];
        i++;
        index++;
    }
}
for (k = beg; k < index; k++)
{
    arr[k] = temp[k];
}
}
```

```
void merge_sort (int arr[], int beg, int end)
{
  int mid;
  if (beg < end)
  {
    mid = (beg + end)/2;
    merge_sort (arr, beg, mid);
    merge_sort (arr, mid + 1, end);
    merge (arr, beg, mid, end);
  }
}
```

Quick Sort

Based on the divide & conquer algorithm that picks an element as a pivot & partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Technique

1) Select an element pivot from the array elements.
2) Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot & all elements greater than the pivot element come after it. After such a partitioning, the pivot is placed in its final pos". This is called partition operation

3) Recursively sort the 2 sub-arrays thus obtained.

Q- Write a program to implement quick sort algorith

A-

```
#include <stdio.h>
int partition (int a[], int beg, int end);
void quick_sort (int a[], int beg, int end);
void main()
{
  int arr[size], i, n;
  printf("Enter the no. of elements in the array: ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for (i=0; i<n; i++)
  {
    scanf("%d", &arr[i]);
  }
  quick_sort (arr, 0, n-1);
  printf("Sorted array : ");
  for (i=0; i<n; i++)
  {
    printf("%d\t", arr[i]);
  }
}
int partition (int a[], int beg, int end)
{
  int left, right, temp, loc, flag;
  loc = left = beg;
  right = end;
  flag = 0;
```

```
while (flag != 1)
{
 while ((a[loc] <= a[right]) && (loc != right))
 right--;
 if (loc == right;
 flag = 1;
 else if (a[loc] > a[right])
 {
    temp = a[loc];
    a[loc] = a[right];
    a[right] = temp;
    loc = right;
 }
 if (flag != 1)
 {
  while ((a[loc] >= a[left]) && (loc != left))
   left++;
   if (loc == left)
   flag = 1;
  else if (a[loc] < a[left])
  {
    temp = a[loc];
    a[loc] = a[left];
    a[left] = temp;
    loc = left;
  }
 }
}return loc;
}
```

```
void quick_sort (int a[], int beg, int end)
{
  int loc;
  if (beg < end)
  {
    loc = partition (a, beg, end);
    quick_sort (a, beg, loc-1);
    quick_sort (a, loc+1, end);
  }
}
```

DATA STRUCTURES

* LINKED LIST

=> Linked list is a very commonly used linear data structure which consists of group of nodes in a sequence.

=> Each node holds its own data & the address of the next node hence forming a chain like structure.

Advantages

They are dynamic in nature which allocates the memory when required.

Insertions & deletion operations can be easily implemented.

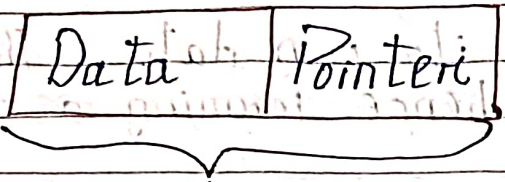
Disadvantages

The memory is wasted as pointers require extra memory for storage.

No element can be accessed randomly, it has to access each node sequentially.

What is a Node?

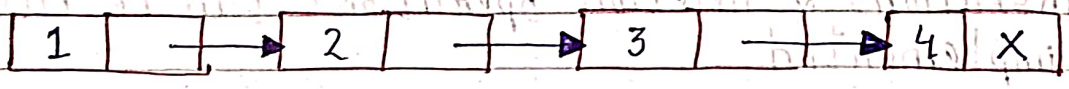
A node in a linked list holds the data value & the pointer which points to the location of the next node in the linked list.



Types of Linked List

1) Singly ~~list~~ Linked List
2) Doubly Linked List
3) Circular Linked List

1) Singly Linked List




```
//A Linked list node
  struct node
  {
     int data;
     struct node * next;
  } *p, *q, *r;
```

Traversing a linked list

```
void createList()
{
  if (start == NULL)
  {
    int n;
    printf("Enter the no. of nodes : ");
    scanf("%d", &n);
    if (n != 0)
    {
      int data;
      struct node *newnode;
      struct node *temp;
      newnode = malloc(sizeof(struct node));
      start = newnode;
      temp = start;
      printf("Enter the no. to be inserted : ");
      scanf("%d", &data);
      start -> info = data;

      for (int c = 2; c <= n; c++)
      {
        newnode = malloc(sizeof(struct node));
        temp -> next = newnode;
        printf("Enter the no. to be inserted : ");
        scanf("%d", &data);
        newnode -> info = data;
        temp = temp -> next;
      }
}
```

```
    printf ("The list is created");
}
else
    printf ("The list is already created");
}
//Func" to traverse the linked list
void traverse()
{
  struct node * temp;
  if (start != NULL)
  {
    temp = start;
    while (temp != NULL)
    {
      printf ("Data : %d ", temp->data);
      temp = temp->link;
    }
  }
}
//Func" to insert at the front of the linked list
void insertAtFront()
{
  int data;
  struct node * temp;
  temp = malloc (sizeof (struct node));
```

```
printf("Enter the no. to be inserted : ");
scanf("%d", &data);
temp -> info = data;
temp -> next = start;
start = temp;
}
//Func^n to insert at the end of the linked list

void insertAtEnd()
{
 int data;
 struct node *temp, *head;
 temp = malloc(sizeof(structnode));
 printf("Enter no. to be inserted : ");
 scanf("%d", &data);
 temp -> info = 0 link = 0;
 temp -> info = data;
 head = start;
 while (head -> link != NULL)
 {
    head = head -> link;
 }
 head -> link = temp;
}

//Func^n to insert at any specific pos^n in the
  linked list
```

```
void insertAtPosition()
{
  struct node *temp, *newnode;
  int pos, data, i = 1;
  newnode = malloc(sizeof(struct node));
  printf("Enter pos" & data : ");
  scanf("%d %d", &pos, &data);
  temp = start;
  newnode → info = data;
  newnode → link = 0;
  while (i < pos - 1)
  {
    temp = temp → link;
    i++;
  }
  newnode → link = temp → link;
  temp → link = newnode;
}
// Func" to delete from the front of the linked list

void deleteFirst()
{
  struct node *temp;
  if (start != NULL)
  {
    temp = start;
    start = start → link;
    free(temp);
  }
}
```

```
//Func^n to delete from the end of the linked list

void deleteEnd()
{
  struct node *temp, *prevnode;
  if (start != NULL)
  {
    temp = start;
    while (temp -> link != NULL)
    {
      prevnode = temp;
      temp = temp -> link;
    }Free (temp);
    prevnode -> link = 0;
  }
}

//Func^n to delete from any specific pos^n from the
  linked list

void deletePosition()
{
  struct node *temp, *position;
  int i = 1, pos;
  if (start != NULL)
  {
    printf ("Enter position : ");
    scanf ("%d", &pos);
```

```
position = malloc (size of (struct node));
temp = start;
while (i < pos - 1)
{
  temp = temp → link;
  i++;
}
position = temp → link;
temp → link = position → link;
free (position);
}
}
```

// Func° to reverse the linked list

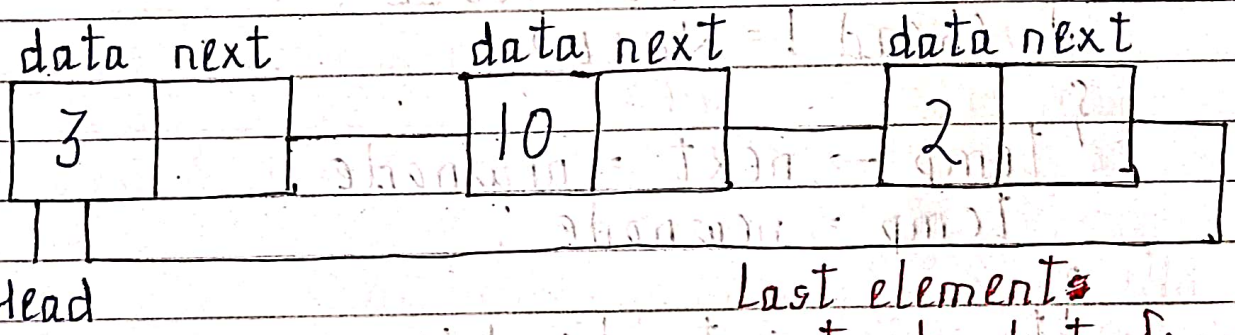
```
void reverse ()
{
struct node *t1, *t2, *temp;
t1 = t2 = NULL;
while (start != NULL)
{
  t2 = start → link;
  start → link = t1;
  t1 = start;
  start = t2;
}
start = t1;
temp = start;
```

```
printf ("Reverse Linked list : ");
while (temp != NULL)
{
  printf ("%d", temp -> info);
  temp = temp -> link;
}
}
```

```
printf ("Reverse Linked list : ");
while (temp != NULL)
{
  printf ("%d", temp -> info);
  temp = temp -> link;
}
}
```

2) Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



Program to implement Circular Linked list

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

struct node
{
   int data;
```

```
    struct node *next;
} *head;

//create a new node

void create()
{
  struct node *newnode, *temp;
  head = NULL;
  newnode = (struct node *) malloc (sizeof (struct
            node));
  printf ("Enter The data : ");
  scanf ("%d", &newnode -> data);
  newnode -> next = NULL;
  if (head != NULL)
  {
    temp -> next = newnode;
     temp = newnode;
  }
  temp -> next = head;

//display The node

void display()
{
  struct node *temp = head;
  while (temp != head)
  {
     temp = temp -> next;
    printf ("%d", temp -> data);
  }
```

```
        printf("\n");
    }
//insert at beginning

void beginsert()
{
  struct node *temp, *ptr;
  int item; head = NULL;
  ptr = malloc(sizeof(struct node));
  if (ptr != NULL)
  {
    printf("Enter the data:");
    scanf("%d", &item);
    ptr -> data = item;
    temp = head;
    while (temp -> next != head)
    {
      temp = temp -> next;
      ptr -> next = head;
      temp -> next = ptr;
      head = ptr;
    }
  }
}

//insert at last

void lastinsert()
```

```
{
 struct node *ptr, *temp;
 head = NULL;
 int item;
 ptr = malloc (size of (struct node));
 if (ptr != NULL)
 {
  printf ("Enter data : ");
  scanf ("%d", &item);
  ptr → data = item;
  temp = head;
  while (temp → next != head)
  {
   temp = temp → next;
  }
  temp → next = ptr;
   ptr → next = head;
  }
}

//insert at position

void insertafter ()
{
  struct node *newNode, *temp;
  head = NULL
  int item ; pos ;
newNode = malloc (sizeof (struct node));
  printf ("Enter the pos : ");
  scanf ("%d", & pos);
```

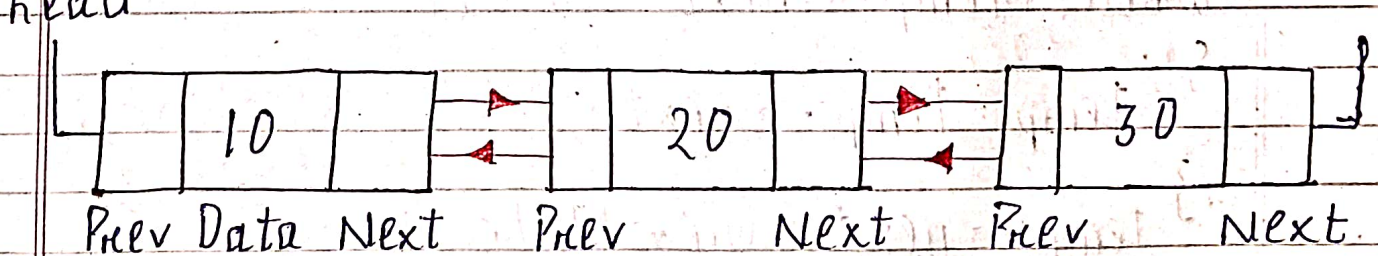
```
temp = head;
for (i=1; i<pos-1 && temp->next != head; i++)
{
  temp = temp->next;
}
newNode->next = temp->next;
temp->next = newNode;
}
//delete from first

void begdelete()
{
 struct node *ptr;
 ptr = head;
 while (ptr->next != head)
 {
   ptr = ptr->next;
 }
 ptr->next = head->next;
 free (head);
 head = ptr->next;
}

//delete from last

void deletelast()
{
 struct node *ptr, *prev;
 ptr = head;
```

```
while (ptr -> next != head)
{
    prev = ptr;
    ptr = ptr -> next;
}
prev -> next = head;
free (ptr);
}

//delete from specific position

void deleteafter()
{
struct node *ptr, *temp;
int c = 1, pos;
printf ("Enter the pos : ");
scanf ("%d", &pos);
while (c < pos-1 && ptr -> next != head)
{
    ptr = ptr -> next;
    c++;
}
temp = ptr -> next;
ptr -> next = temp -> next;
free (temp);
}
```

```
//reverse the list

void reverseList()
{
 struct node *prev, *current, *nextNode;
 prev = NULL;
 current = head;
 while (current != head)
 {
   nextNode = current->next;
   current->next = prev;
   prev = current;
   current = nextNode;
 }
 head->next = prev;
   head = prev;
}
```

3) Doubly Linked List

In a doubly linked list, each node contains a data part & 2 addresses, one for the previous node & one for the next node.



// create a new node

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
  int data;
  struct node *prev;
  struct node *next;
} *head;
void create()
{
  struct node *ptr, *temp;
  int num;
  printf("Enter data : ");
  scanf("%d", &num);
  ptr = malloc(sizeof(struct node));
  ptr -> data = num;
  ptr -> prev = NULL;
  ptr -> next = NULL;
  temp = head;
  while (temp -> next != NULL)
  {
    temp = temp -> next;
  }
  temp -> next = newNode;
  newNode -> prev = temp;
}
```

// insert at front

```
void beginsert()
{
  int data; head = NULL;
  struct node *temp;
  temp = malloc (size of (struct node));
  printf ("Enter data :");
  scanf ("%d", &data);
  temp → data = data;
  temp → prev = NULL;
  temp → next = head;
  head = temp;
}
```

// insert at last

```
void lastinsert()
{
  struct node *Newnode;
  newNode = malloc (size of (struct node));
  ~~newNode → data~~ =
  int data; head = NULL;
  printf ("Enter data : ");
  scanf ("%d", &data);
  newNode → data = data;
  newNode → prev = NULL;
  newNode → next = NULL;
```

```
if (head != NULL)
{
  struct node * temp = head;
  while (temp -> next != NULL)
  {
    temp = temp -> next;
  }
  temp -> next = newNode;
  newNode -> prev = temp;
}
}

//insert at position

void insertafter()
{
  struct node * newNode, * temp;
  int data, pos, c = 1
  newNode = malloc (size of (struct node));
  printf ("Enter the pos & data : ");
  scanf ("%d %d", &pos, &data);
  newNode -> data = data;
  newNode -> next = NULL;
  newNode -> prev = NULL;
  while (temp != NULL && c < pos - 1)
  {
    temp = temp -> next;
    c++;
  }
```

```
newNode->next = temp->next;
newNode->prev = temp;
if (temp->next != NULL)
{
  temp->next->prev = newNode;
}
temp->next = newNode;
}

//delete from first

void deletefirst()
{
 struct node *temp;
 temp = head;
 if (head->next != NULL)
 {
  head = head->next;
  head->prev = NULL;
 }
 free(temp);
}

//delete from last

void deletelast()
{
 struct node *temp;
 temp = head;
```

```
if (head -> next != NULL)
{
  while (temp -> next != NULL)
  {
    temp = temp -> next;
  }
  temp -> prev -> next = NULL;
}
free (temp);
}

//delete from position

void delete after ()
{
 struct node * temp = head;
 int pos, c = 1;
 while (temp != NULL && c
 printf ("Enter pos : ");
 scanf ("%d", & pos);
 while (temp != NULL && c < pos)
 {
   temp = temp -> next;
   c++;
 }
 if (temp -> next != NULL)
 {
   temp -> next -> prev = temp -> prev;
 }
```

```
    temp -> prev -> next = temp -> next;
    free (temp);
}
//Reverse the list

void reverse ()
{
  struct node * current = head;
  struct node * temp = NULL;
  while (current != NULL)
  {
    temp = current -> prev;
    current -> prev = current -> next;
    current -> next = temp;
    current = current -> prev;
  }
  if (temp != NULL)
  {
    head = temp -> prev;
  }
}

//display the list

void display ()
{
  struct node * ptr temp;
  ~~temp~~ = head;
  ptr
```

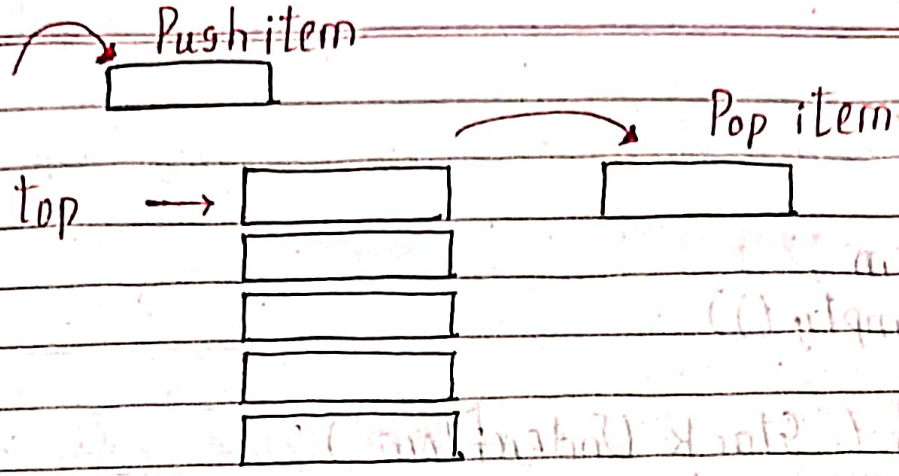
```
while (ptr != NULL)
{
  printf("%d", ptr->data);
  ptr = ptr->next;
}
```

* STACK

A stack is an ordered list in which insertion & deletion are done at one end, called top. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.

Basic Operations on Stack

- push() to insert an element into stack.
- pop() to remove an element from the stack.
- top() returns the top element of the stack.
- isEmpty() returns true if stack is empty else false
- size() returns the size of stack.



Push

Adds an item to the stack. If the stack is full, then it is said to be an Overflow condⁿ.

```
void push (int item)
{
  if (isFull())
  {
    printf ("Stack Overflow");
    return;
  }
  top = top + 1;
  stack [top] = item;
}
```

Pop

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If stack is empty, it is said to be an Underflow Condⁿ.

```
int pop()
{
  int item;
  if (isEmpty())
  {
    printf("Stack UnderFlow");
    exit(1);
  }
  item = stack[top];
  top = top - 1;
  return item;
}
```

isEmpty

```
int isEmpty()
{
  if (top == -1)
  {
    return 1;
  }
  else
    return 0;
}
```

isFull()

```
int isFull()
{
  if (top == MAX-1)
  {
    return 1;
  }
  else
    return 0;
}
```

Display The stack

```
void display ()
{
 int i ;
 if (isEmpty ())
 {
  printf ("Stack is empty");
  return ;
 }
printf ("Stack elements : ");
for(i = top ; i >= 0 ; i--)
{
  printf ("%d \n", stack [i]);
}
 printf ("\n");
}
```

Linked List Implementation of Stack

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

struct node
{
  int data ;
  struct node *ptr;
} *top, *top1, *temp ;

int count = 0 ;
```

```
void push (int data)
{
 if (top == NULL)
 {
 top = malloc (1 * sizeof (struct node));
 top -> ptr = NULL;
 top -> data = data;
 }
 else
 {
 temp = malloc (1 * size of (struct node));
 temp -> ptr = top;
 temp -> data = data;
 top = temp;
 }
 count ++;
}

int pop()
{
 top1 = top;
 if (top1 != NULL)
 {
 top1 = top1 -> ptr;
 int popped = top -> data;
 free (top);
 top = top1;
 count --;
 return popped;
 }
}
```

```
void display ();
{
  top1 = top;
  if (top1 != NULL)
  {
    printf ("The stack is ");
    while (top1 != NULL)
    {
      printf ("%d ", top1 -> data);
      top1 = top1 -> ptr;
    }
    printf ("NULL");
  }
```

Arithmetic Notations

1) **Infinix Notation** : It is a way of writing mathematical expressions where the operator is placed bet^n the two operands.
   Eg. "3+4"

2) **Postfix Notation** : The operator appears after the operands.
   Eg. "34+"

3) **Prefix Notation** : "+34"

Rules for the Conversion from infix to postfix

1) Scan all the symbols from left to right.
2) If the reading symbol is an operand, then append it to the postfix expression.
3) If the reading symbol is left parenthesis '(', then push it onto the stack.
4) If the reading symbol is right parenthesis ')', then pop all the contents of the stack until the respective left parenthesis is popped & append each popped symbol to postfix expression.
5) If the reading symbol is an operator (+, -, *, /), then push it onto the stack. However, first, pop the operators which are already on the stack that have higher or equal prerecedence than the current operator & append them to the postfix.
6) If the input is over, pop all the remaining symbols from the stack & append them to the postfix.

Q. K+L-M*N+(O^P)*W/U/V*T+Q

INPUT	STACK	POSTFIX EXPRESSION
K		K
+	+	K
L	+	KL
-	+ -	KL+
M	-	KL+M
*	- *	KL+M
N	- *	KL+MN
+	+	KL+MN*-
(	+(	KL+MN*-
O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^
W	+*	KL+MN*-OP^W
/	+/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V/
T	+*	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*+Q

KL+MN*-OP^W*U/V/T*+Q+

Evaluation of postfix expression using stack

1) Scan the expression from left to right.

2) If we encounter any operand in the expression. Then push the operand in the stack.

3) When we encounter any operator in the expression Then we pop the corresponding operands from the stack.

4) When we finish with the scanning of the expression, the final value remains in the stack.

Q- 2 3 4 * +

A-

Input	Stack
234*+	
34*+	2
4*+	3 2
*+	4 3 2
+	12 2

Result - 14

Rules for the Conversion from infix to prefix

1) Reverse the infix expression.
2) Make every '(' as ')' & every ')' as '('.
3) Convert expression to postfix form & reverse it.

Q- (A+(B^C)*D+E^5

A- 5^E+D*)C^B+A(
= 5^E+D*(C^B+A)

INPUT	STACK	EXPRESSION
5		5
^	^	5
E	^	5E
+	+	5E^
D	+	5E^D
*	+*	5E^D
(	+*(	5E^D
C	+*(	5E^DC
^	+*(^	5E^DCB
B	+*(^	5E^DCB
+	+*(+	5E^DCB^
A	+*(+	5E^DCB^A
)	+*	5E^DCB^A+

5E^DCB^A+*+
↓
+*A^BCD^E5

QUEUE

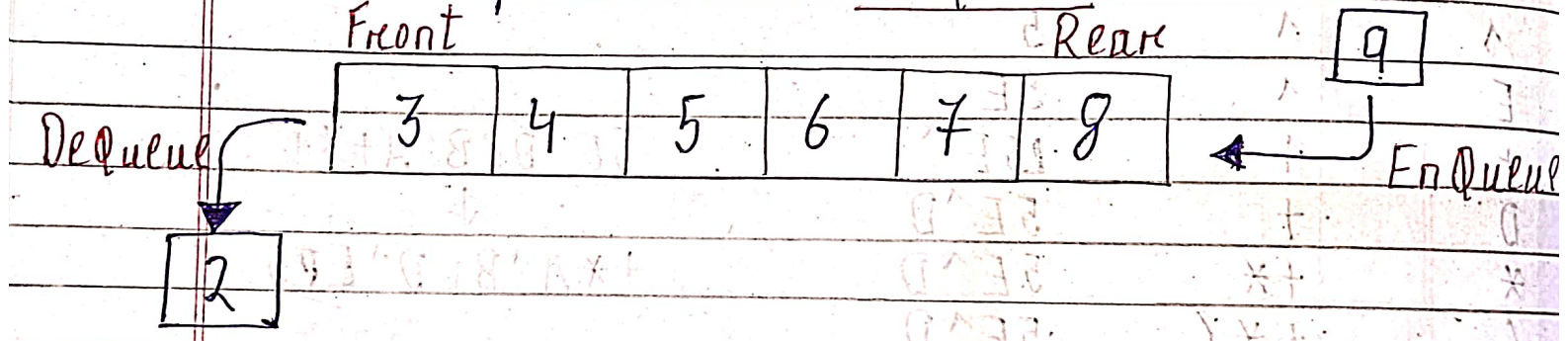
* QUEUE

A queue is an ordered list in which insertions are done at one end (rear) & deletions are done at other end (front).

The first element to be inserted is the first one to be deleted. Hence, it is called FIRST IN FIRST OUT (FIFO) or LAST IN LAST OUT (LIFO) list.

When an element is inserted in a queue, the concept is called EnQueue.

When an element is removed from the queue, the concept is called DeQueue.



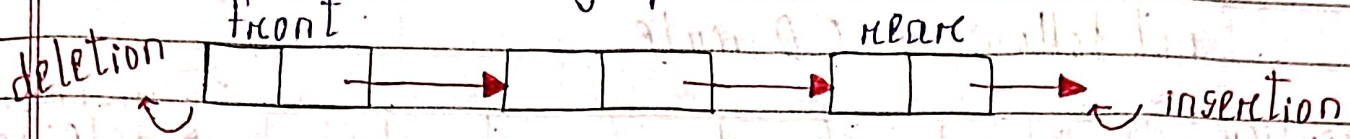
Types of Queue

There are 4 diff types of queues :-

i) Simple
ii) Circular
iii) Priority
iv) Double Ended

ii) Simple Queue

Insertion takes place at rear & removal occurs at front. It strictly performs FIFO rule.



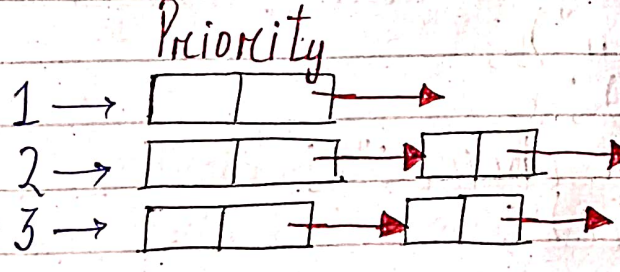
iii) Circular Queue

The last element points to the first element making a circular link.

The main advantage over simple queue is if the last option is full & the first option is empty, we can insert an element in the first option.

iii) Priority Queue

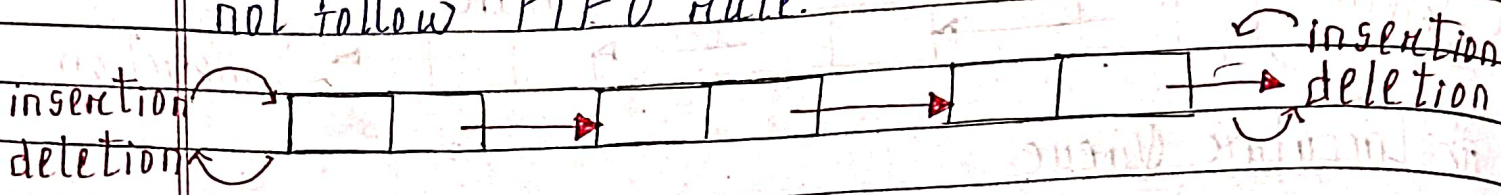
It is a special type of queue in which each element is associated with a priority & is served according to its priority.



Insertion occurs based on the arrival of the values & removal occurs based on priority.

iv) Deque (Double Ended Queue)

Insertion & removal of elements can be performed from either front or rear. Thus, it does not follow FIFO rule.



Implement Queue using Array

```
#include <stdio.h>
# define MAX 50
void insert();
void delete();
void display();
int queue[MAX];
int rear = -1;
int front = -1;
int main()
{
  int choice;
  while (1)
  {
  printf("1. Insert element to queue\n");
  printf("2. Delete element from queue\n");
  printf("3. Display\n");
  printf("4. Exit\n");
  printf("Enter your choice:");
  scanf("%d", &choice);
```

```
switch (choice)
{
  case 1:
     insert ();
     break ();
  case 2:
     delete ();
     break ();
  case 3:
     display ();
     break ();
  case 4:
     exit (1);
  default:
     printf ("WRONG CHOICE \n");
  }
}
}

void insert ()
{
  int item;
  if (rear == MAX-1)
     printf ("QUEUE OVERFLOW");
  else
     if (front == -1)
     {
        front = 0;
        printf ("Enter the element: ");
```

```
    scanf ("%d", &item);
    rear++;
    queue[MAX] = item;
  }
}

void delete()
{
  if (front == -1 || front > rear)
  {
    printf ("QUEUE UNDERFLOW");
    return;
  }
  else
  {
    printf ("Element deleted is %d \n", queue[front
    front++;
  }
}

void display()
{
  int i;
  if (front == -1)
    printf ("Queue is Empty");
  else
    printf ("Queue is : \n");
    for (i=front; i<=rear; i++)
      printf ("%d", queue[i]);
    printf ("\n");
}
```

Circular Queue

```
#include <stdio.h>
#define MAX 6
int queue[max];
int front = -1;
int rear = -1;

void enque (int element)
{
  if (front == -1 && rear == -1)
  {
     front = 0;
     rear = 0;
     queue[rear] = element;
  }
  else if ((rear + 1) % max == front)
  {
    printf("QUEUE OVERFLOW");
  }
  else
  {
     rear = (rear + 1) % max;
     queue[rear] = element;
  }
}
```

```
int dequeue()
{
 if ((front == -1) && (rear == -1))
 {
  printf ("QUEUE UNDERFLOW");
 }
 else if (front == rear)
 {
  printf ("The dequed element is %d", queue[front]);
  front = -1;
  rear = -1;
 }
 else
 {
  printf ("The deque element is %d", queue[front]);
  front = (front + 1) % max;
 }
}

void display ()
{
 int i = front;
 if (front == -1 && rear == -1)
     printf ("Queue is empty");
 else
     printf ("Elements are : ");
     while (i <= rear)
     {
       printf ("%d", queue[i]);
       i = (i+1) % max;
     }
}
```

gain

```
int main()
{
  int x;
printf("Enter the element : ");
scanf("%d", &x);
enqueue(x);
dequeue();
display();
}
```

Implementation of Circular Queue using Linked List

```
#include <stdio.h>
struct node
{
  int data;
  struct node *next;
};

struct node *front = -1;
struct node *rear = -1;

void enque(int x)
{
 struct node *newnode;
 newnode = malloc(sizeof(struct node));
 newnode->data = x;
 newnode->next = NULL;
```

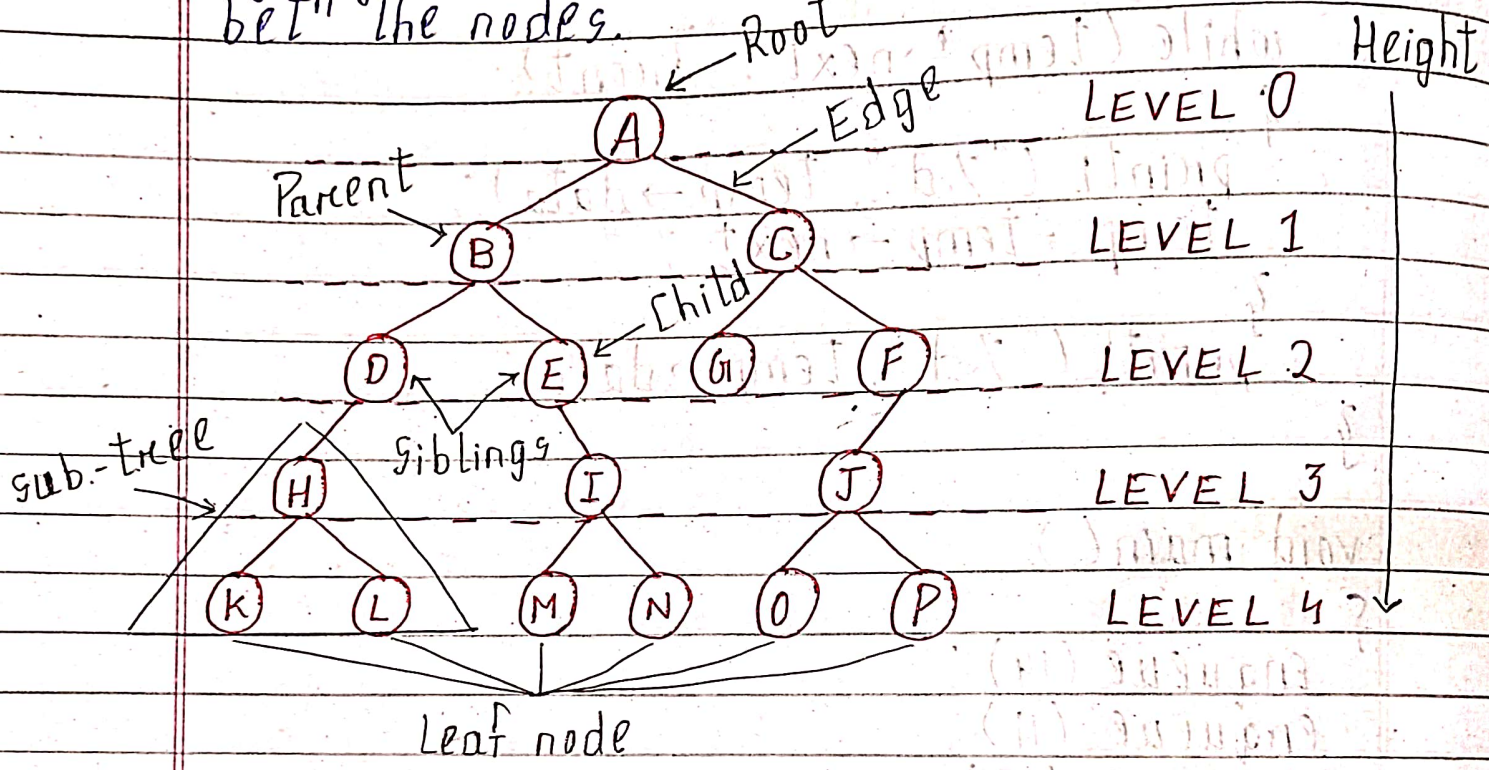
```
if (rear == -1)
{
  front = rear = newnode;
  rear -> next = front;
}
else
{
   rear -> next = newnode;
   rear = newnode;
   rear -> next = front;
}
}

void dequeue()
{
  struct node *temp;
  temp = front;
  if (front == -1 && rear == -1)
     printf("Queue is empty");
  else if (front == rear)
     front = rear = -1;
     free(temp);
  else
     front = front -> next;
     rear -> next = front;
     free(temp);
}
```

```
void display ()
{
  struct node *temp;
  temp = front;
  printf ("The elements are : ");
  if (front == -1 && rear == -1)
    printf ("Queue is empty");
  else
    while (temp -> next != front)
    {
      printf ("%d", temp -> data);
      temp = temp -> next;
    }
    printf ("%d", temp -> data);
}

void main()
{
  enqueue (14);
  enqueue (11);
  enqueue (13);
  display ();
  dequeue ();
}
```

TREES

A tree data structure is a hierarchical structure that is used to store represent & organise data in a way that is easy to navigate & search.

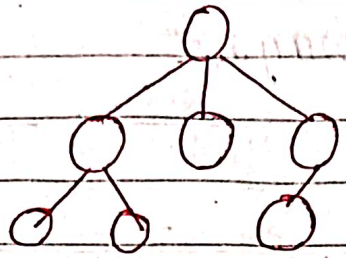
It is a collection of nodes that are connected by edges & has a hierarchical relationship bet^n the nodes.



Types of Trees

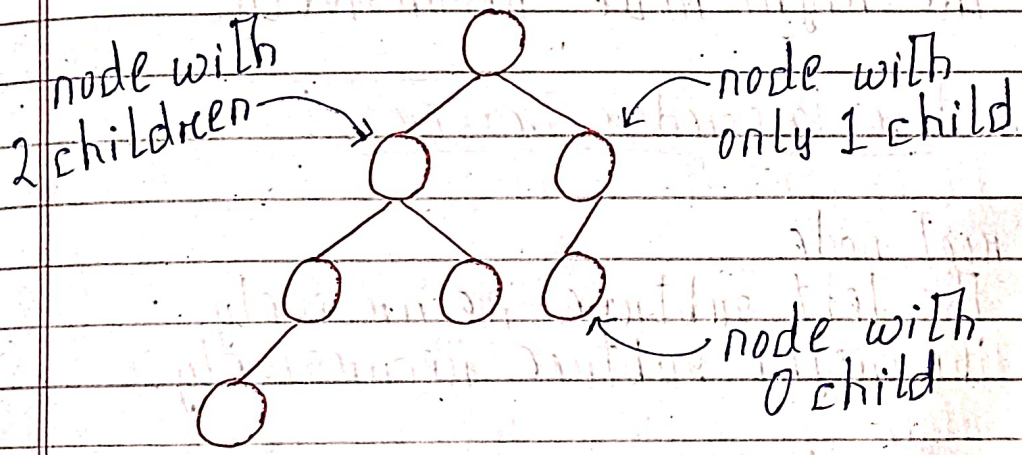
1) General Trees

Unordered data structures where the root has min. 0 & max^{m} n subtrees.



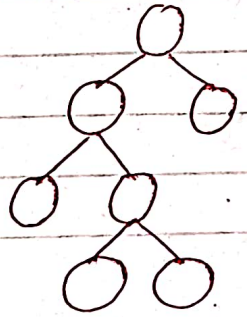
2) Binary Trees

Each node has either 0 child or 1 child or 2 children.



Full Binary Trees

Every node has either 0 or 2 children.



* Tree Traversal

It refers to the process of visiting each node in a tree data structure exactly once.

It can be done in 3 ways :-
i) Inorder
ii) Preorder
iii) Postorder

Algorithm for Inorder Traversal

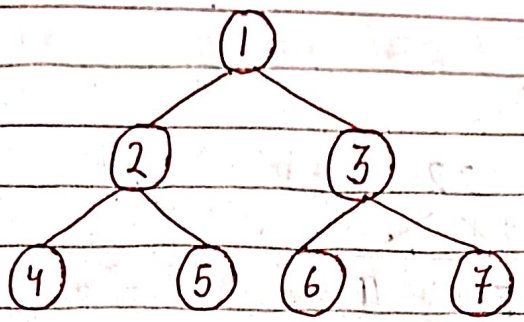
Traverse the left subtree recursively.
Visit the root node.
Traverse the right node recursively.

Algorithm for Preorder Traversal

Visit the root node.
Traverse the left subtree recursively.
Traverse the right subtree recursively.

Algorithm for Postorder Traversal

Traverse left subtree recursively.
Traverse right subtree recursively.
Visit the root node.

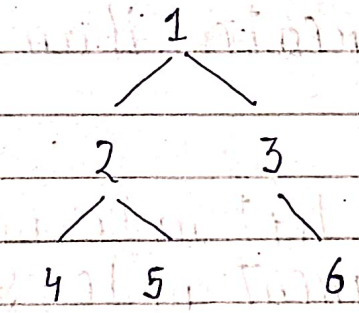


Inorder → 4 2 5 1 6 3 7
Preorder → 1 2 4 5 3 6 7
Postorder → 4 5 2 6 7 3 1

Q- in[] = {4, 2, 5, 1, 3, 6}
pre[] = {1, 2, 4, 5, 3, 6}

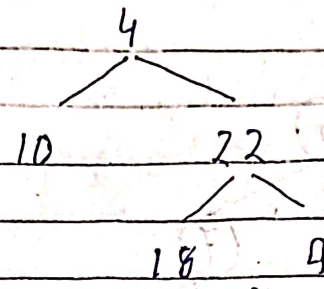
Find postorder & represent it in tree.

A- post[] = {4, 5, 2, 6, 3, 1}



Q- postorder = [10, 18, 9, 22, 4]
inorder = [10, 4, 18, 22, 9]

A-



pre = [4, 10, 22, 18, 9]

* Binary Search Tree (BST)

It is a node-based binary tree data structure which has the following properties :-

The left subtree of a node contains only nodes with keys lesser than the node's key.

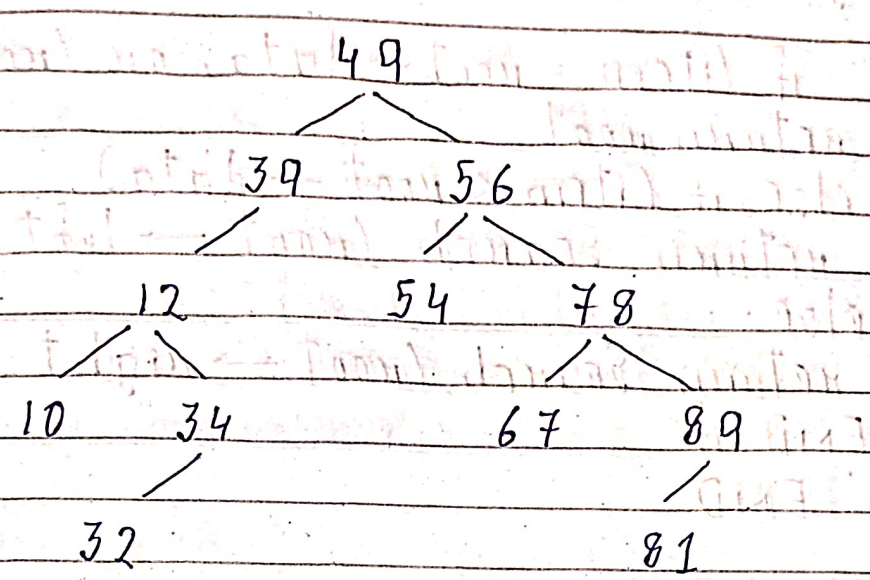
The right subtree of a node contains only nodes with keys greater than the node's key.

In BST, the value of left node must be smaller than the parent node & the value of right node must be greater than the parent node.

Q- Create a BST using the following data elements :-

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

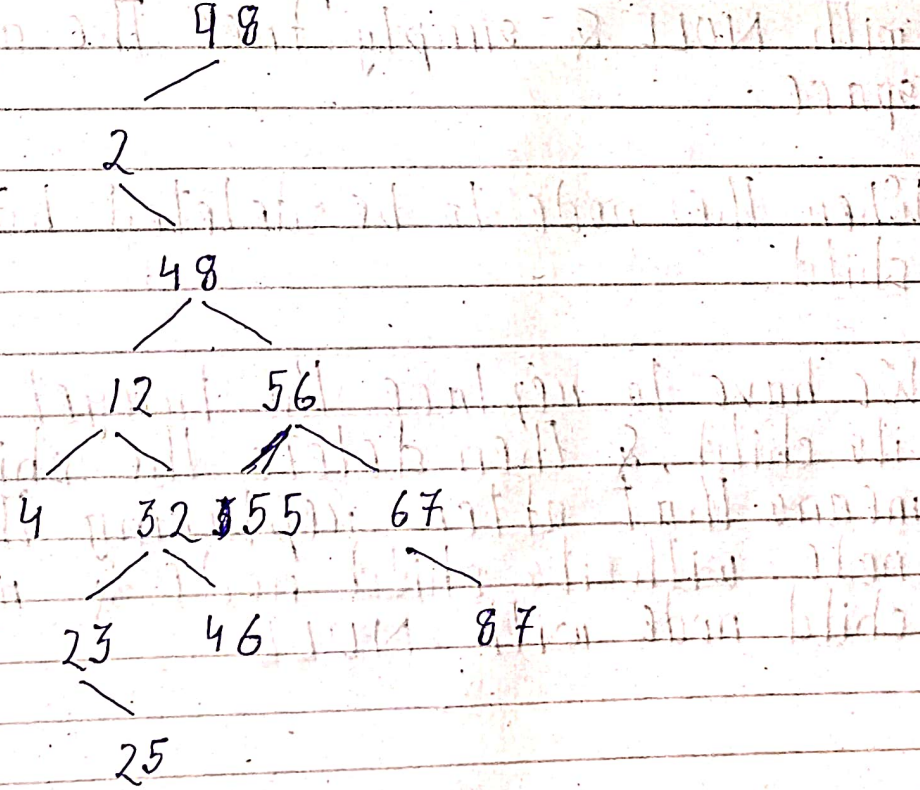
A-



Q- Create a BST with the input given below:

98, 2, 48, 12, 56, 32, 4, 67, 23, 87, 25, 55, 46

A-



Searching in BST

STEP 1 : if (item = root → data) or (root = NULL)
return root
else if (item < root → data)
return Search (root → left, item)
else
return Search (root → right, item)
END if
STEP 2 : END

Deletion in BST

i) When the node to be deleted is the leaf node.

Here, we have to replace the leaf node with NULL & simply free the allocated space.

ii) When the node to be deleted has only one child

We have to replace the target node with its child, & then delete the child node. It means that after replacing the target node with its child node & replace the child node with NULL.

iii) When the node to be deleted has 2 children

→ First, find the inorder successor of the node to be deleted.
→ After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
→ At last, replace the node with NULL & free up the allocated space.

Insertion in BST

If the root node to be inserted is less than the root node, then search for an empty location in the left subtree. Else search for the empty location in the right subtree & insert the data.

* Balanced Binary Tree

It is defined as a binary tree in which height of the left & right subtree of any node differ by not more than 1.

Cond^n's :-

i) difference bet^n left & right subtree for any node is not more than 1.
ii) The left subtree is balanced.
iii) The right subtree is balanced.

NOTE Tree is said to be balanced if balance factor is in bet^n -1 to 1.

AVL Tree: is height balanced tree in which each node is associated with a balance factor.

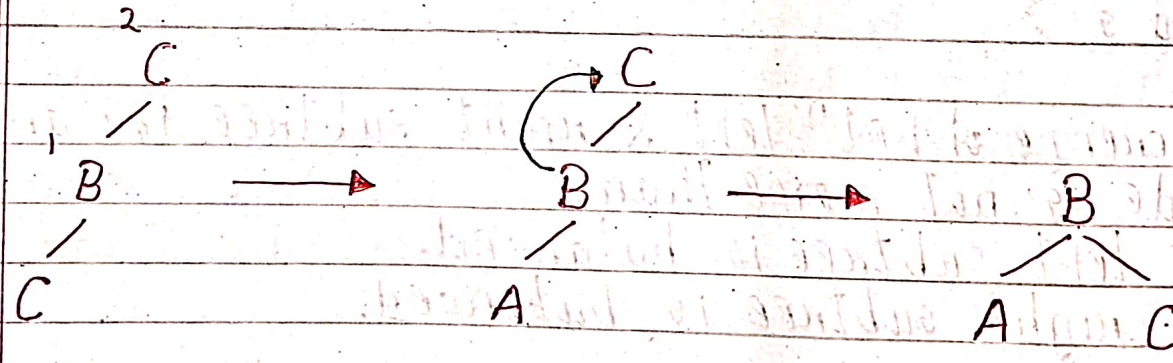
Balance factor = height - height
left right

AVL Rotations

4 types :- 1) LL
2) LR
3) RL
4) RR

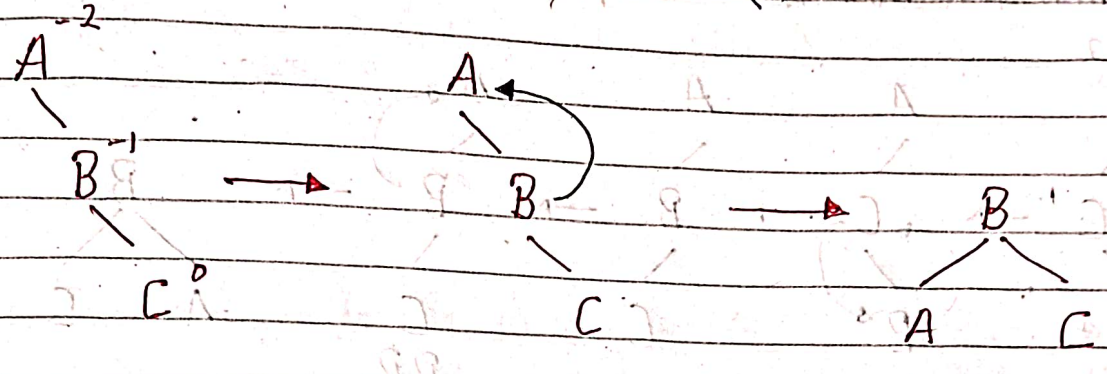
1) LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance 2.



2) RR Rotation

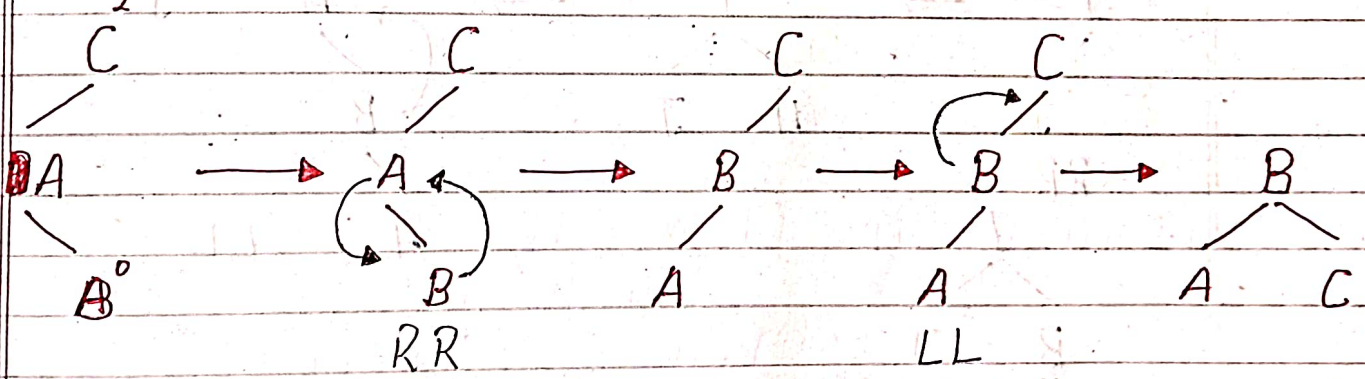
When BST becomes unbalanced, due to a node is inserted into the right subtree, then we perform RR rotation, RR rotation is anti-clockwise rotation which is applied below a node having balance -2.



3) LR Rotation

LR Rotation = RR Rotation + LL Rotation

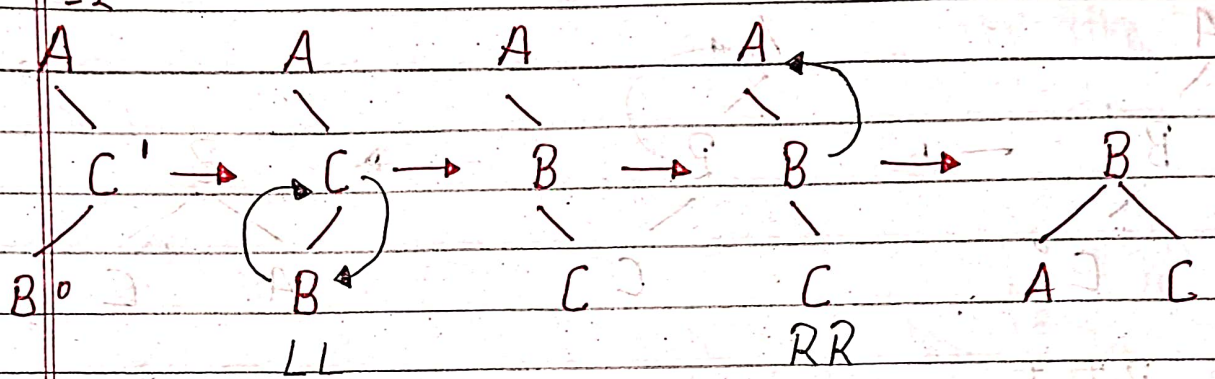
First RR rotation is performed on subtree & then LL rotation is performed on full tree.



4) RL Rotation

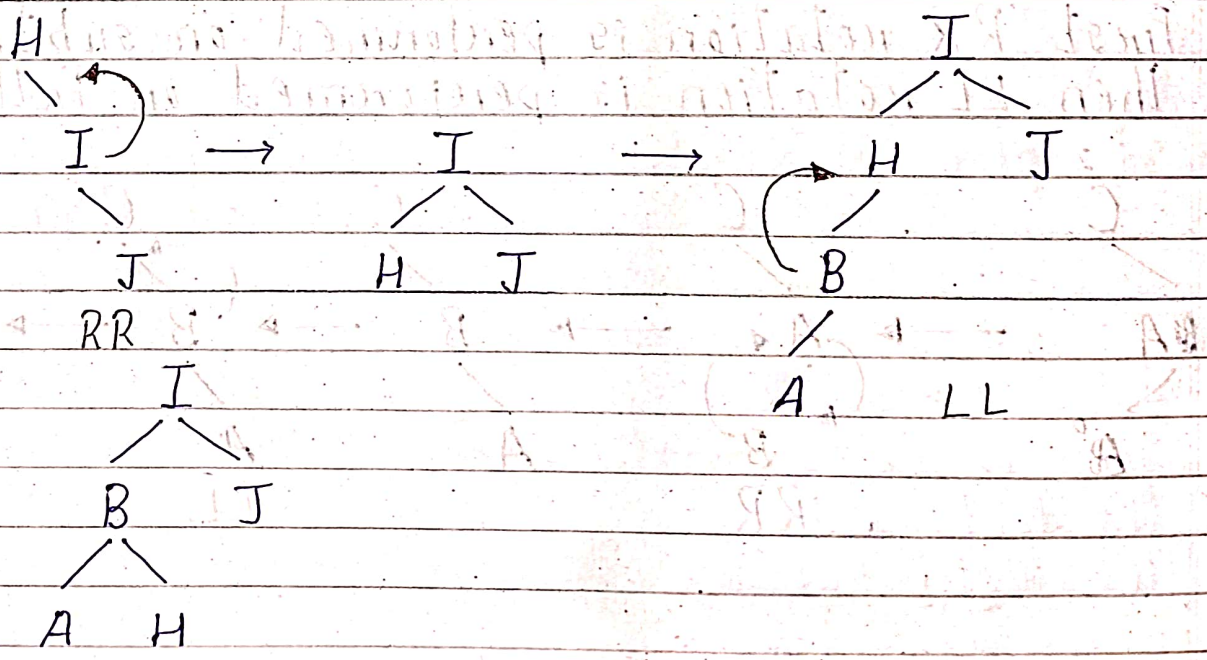
RL = LL + RR
Rotation Rotation Rotation

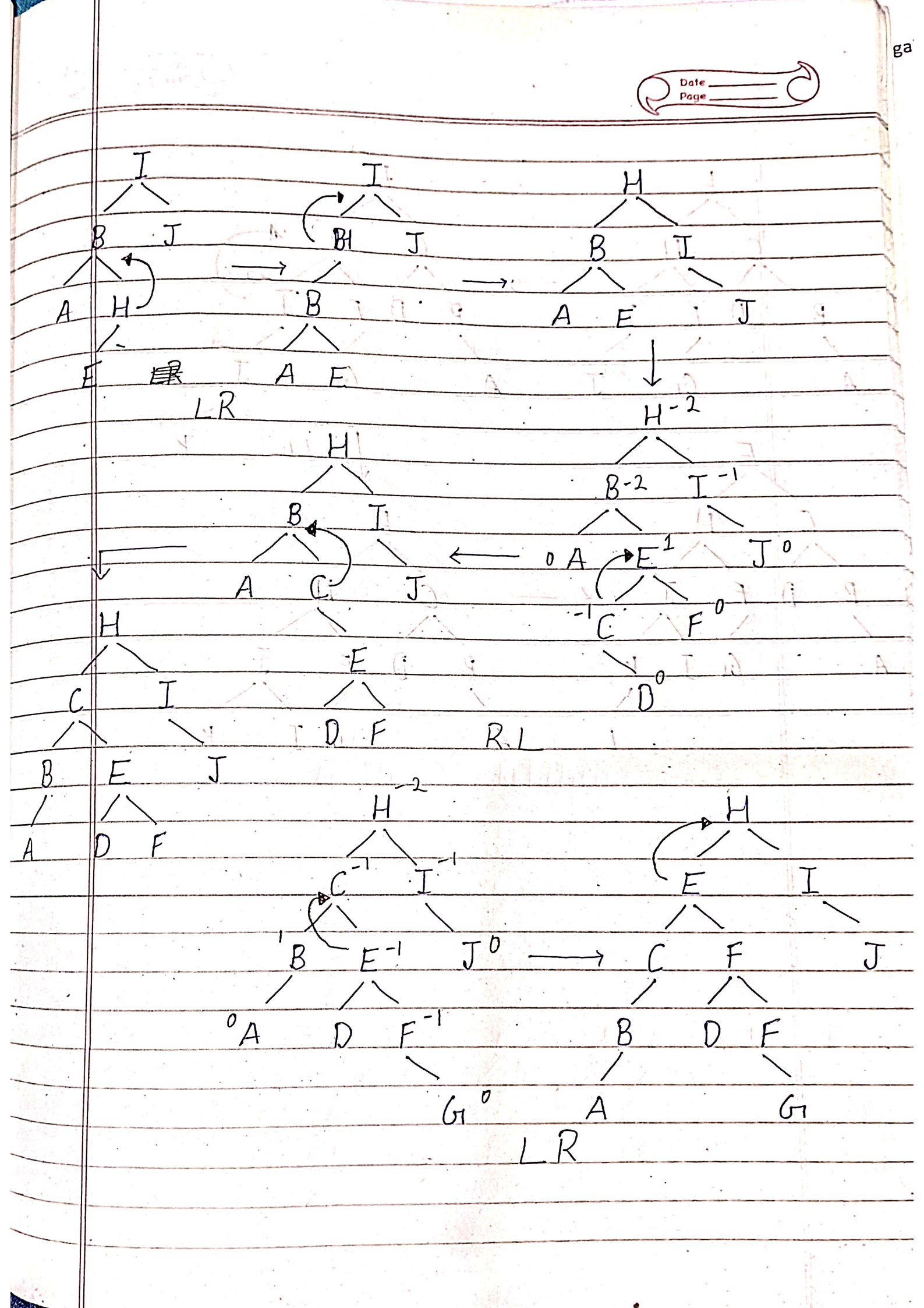
First LL rotation is performed on subtree & Then RR rotation is performed on full tree.

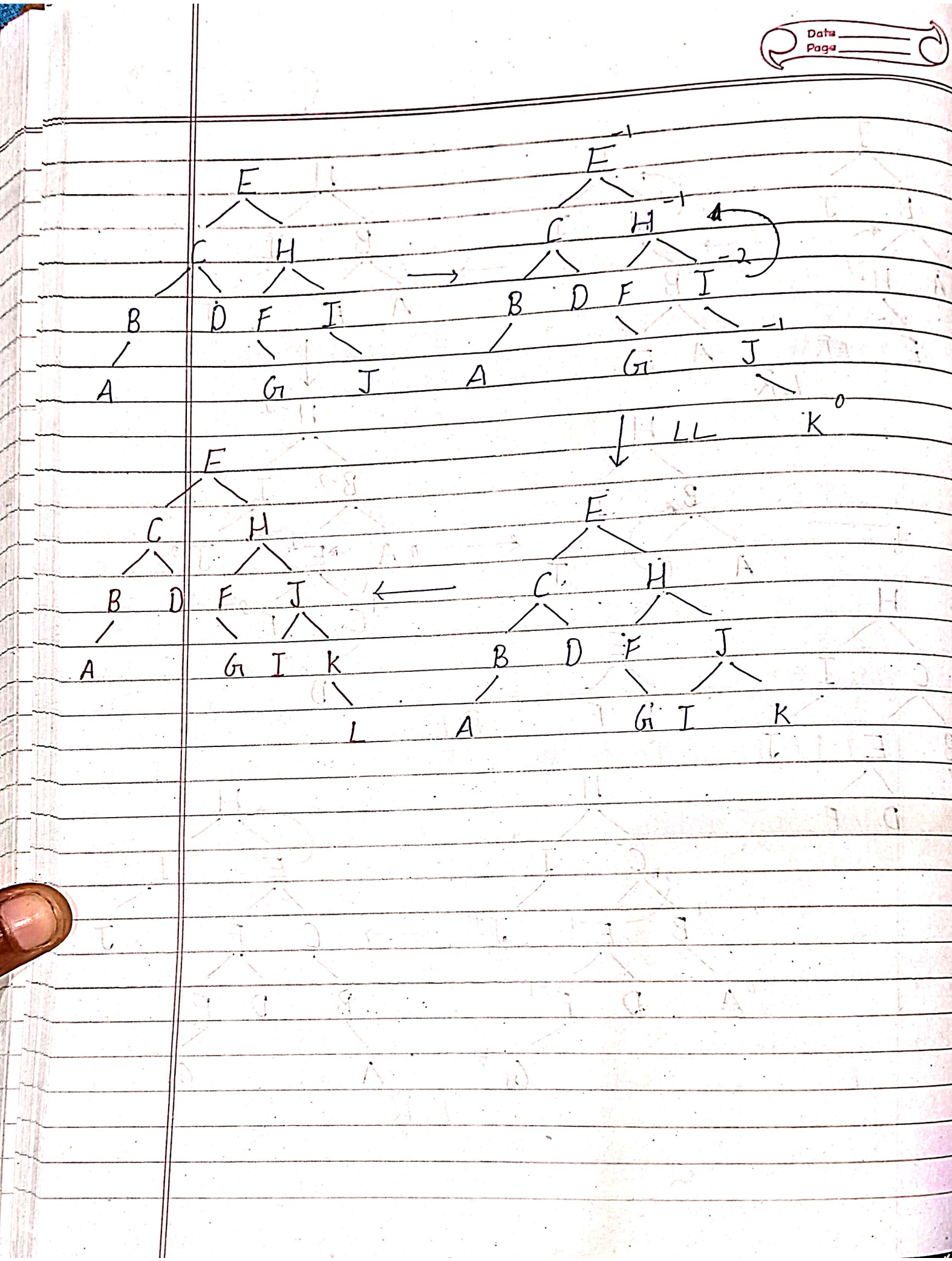


Q- Construct an AVL tree using the following elements -
H, I, J, B, A, E, C, F, D, G, K, L

A-




I
B J
A H
E
LR
I
B H J
B
A E
H
B I
A E J
H-2
B-2 I-1
0 A E 1 J 0
-1 C F 0
D 0
RL
H
C I
B E J
A D F
H
B I
A C J
E
D F
H-2
C-1 I-1
B E-1 J 0
0 A D F-1
G 0
H
E I
C F J
B D F
A G
LR


E
C
H
B
D
F
I
A
G
J
E
C
H
B
D
F
I
A
G
J
K
LL
E
C
H
B
D
F
J
A
G
I
K
L
E
C
H
B
D
F
J
A
G
I
K