21 Programming & Data Structure 08

Avg mark:
13 marks

- Teaching schedule.
  - → Basic: (operators, loops, function, switch)
  - → Storage classes, scope
  - → pointers String
  - → Array
  - → Recursion
  - → Structure, unions
  - → Dynamic Memory Allocation
- Data structure
  - → Linked list
  - → Stack
  - → Queue
  - → Tree (BST, AVL)
  - → Hashing

Text Books

1. "C Test Your Aptitude" By Venugopal & N Chandra Kanth
2. "C Language" By Dennis Ritchie
3. DS By Sahani
4. DS By Mark allen weiss.

- Implementation of queue using stack & stack from queue
- enum datatype
- compile time & runtime
- Hashing question
- alltest

Precedence table

Basic of C programing

- Operators:-

Operators	Precedence	Associativity
()	1(High)	L → R
↑	2	R → L
*1 %	3	L → R
+ -	4	L → R
. =	5(low)	R → L

- Associativity
If two or more operators are having same precedence then the expression will be evaluated by using associativity

ex: () () ()

2^3^4

Expression	Result
5/2	2
5.0/2	2.5
5.0/2.0	2.5
2/5	0
2.0/5	0.4
2.0/5.0	0.4

Note

- If both are integer the o/p will be integer & if any one is float the o/p will be float

- Relational & logical operators. (↳ <, <=, >, >= ; ↳ &&, ||)

1. All the relational & logical operators return 0 or 1. If the expression is true then it return 1 & if the expression is false then it return 0.
2. All non-zero are consider as true & zero(0) is consider as false.

True	False.
1	0.0
10	0
-10	\0
6.5	NULL
-3.5	
0.6	

\0 ASCII value is 0.

Eg 1: a=5>4;

(T) → 1

a=1

Eg 2: a=(5>4)+(5>3)

(T) (T)

1 + 1

a=2

Eg 3: a=5>4>3

(T)

1 > 3

(F)

0

a=0

Expression	Assigned to int	Assigned to float
1. 5	5	5.0
2. 5/2	2	2.0
3. 5.0/2	2	2.5
4. 5.0/2.0	2	2.5
5. 2/5	0	0.0
6. 2.0/5	0	0.4
7. 2.0/5.0	0	0.4

Eg: int a = 2*3/4 + 2.0/5 + 8/5

6/4 + 2.0/5 + 8/5

1 + 0.4 + 1

2.4

int a = 2.4 ⇒ (2)

- Modulus operator (%):

1. 15 %. 7 = 1
2. -15 %. 7 = -1
3) 15 %. -7 = 1
4. -15 %. -7 = -1
5. -15 % +25 = -15
6. +15 % -25 = 15 (smaller than divisor)
7. +15.5 %. 7 = error
8. -15.7 %. 7 = error

+7) -15 (-2
-14
-1

-7) +15 (-2
+14
+1

-7) -15 (+2
-14
-1

Note

- modulus always give numerators sign
- modulus doesn't work on float value, it work only on the Integers.
- If the value is small without sign, then it give same value as o/p.

Ex: 15.0/2 * 2 %. 7

7.5 * 2 %. 7

15.0 %. 7

error

ex: void main() {
int a=5;
if (a=8) { (8 → non zero (True))
printf("Raj");
}
else {
printf("kishore");
}
printf("%d", a);
}

a 8

• assignement operator assign the value & return the assign value

Format specifies.
- %d int
- %f float
- %c chars
- %s string
- %u address

Printf

compiler if not present write return 0 at end

1. Ex: void main()
{
int x;
x = printf("Hello");
printf("%d", x);
}

O/p: 5

x ← value 5

O/p: Hello 5

5 ← no. of character

Note
- printf always return integer i.e no of symbols displayed on the screen by that particular printf's only
- If we want we can collect that value

2. Ex: void main()
{
int a;
a = printf("Made %d Easy", printf("delhi"));
printf("%d", a);
}

a ← 9

O/p: delhi Made %d Easy 9

Made 5 Easy

return 5 → these 5 is take care by %d

Ex3: write a program to read a two proper integers & print them on the screen.

```
void main(){
    int x,y,sn;
    pf("Enter two proper integers");
    sn = scanf("%d,%d", &x,&y);
    if(sn==2){
        pf("proper inputs given that is %d %d", x,y);
    }else{
        printf("wrong i/p given");
    }
}
```

Scanf

- Scanf() always returns integer i.e no. of proper i/p given A/C to format specifiers.

I/p	sn
*,9	0
#,0	1
5.5,$	0
10,20	2

Ternary Operator

condition ? if condition is True : if condition is false

note

- In the ternary operator there should be equal no. of colon's & question marks otherwise it give error.
- Every question mark followed by colon not immediately but following.
- Every colon should match with the just before question marks.

Ex1: a = 3>4 ? 10 : 8>7 ? 20 : 30;

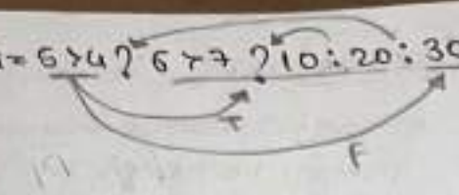
o/p: a=20

T, F

8>7 ? 20 : 30

T, F

Ex$_2$: a = 6>4 ? 6>7 ? 10 : 20 : 30; O/P: a=20



6>7 ? 10 : 20

Ex$_3$: a = 2>3 ? 5>4 ? 10 : 30 : 8>7 ? 30 : 40
(10) (30)

2>3 ? 10 : 30

⇒ (30)//

Ex$_4$: a = 6>7 ? 2>3 ? 10 : 6>8 ? -2 : 3 : 0 ? -1 : 6>8 ? 10 : 3>4 ? 10 : 20
(3) (20)
(3) (20)

6>7 ? 3 : 0 ? -1 : 20

0 ? -1 : 20

(20)//

Ques] write a program to find max. of 3 no. using ternary operator?

```
int a,b,c,max;
max = (a>b&&a>c)?a : (b>a&&b>c)?b:c;
```
or
```
a>b&&a>c?:b>c?b:c;
```

Ques] write a program to find min of 4 no.

```
int a,b,c,d,min;
min = a<b&&a<c&&a<d?a:b<c&&b<d?b:c<d?c:d;
```

H.W] write the above two program without using (&&) and operator

max of 3 variable: max: a>b?a>c?a:c:b<c?b:c

min value out of 4 variable : `a<b?a<c?a<d?a:d:c<d?c:d:b<c?b<d?b:d:c<d?c:d;`

Gate Ques: which combinatn of the integer variable x, y & z make variable 'a' get the value 4 in the following expression.

`a = x>y? x>z? x:z : y>z? y:z;`

three method (my method)

```
a<b?a<c?a<d?a
          :d
       c<d?c
          :d
    b<c?b<d?b
          :d
       c<d?c
          :d
```

a) x=6, y=5, z=3
b) x=6, y=3, x=5
c) x=5, y=4, z=5
d) x=3, y=4, z=2

Pre/post Increment/ Decrement

Pre	Post	Pre	Post
a=10;	a=10;	a=10;	a=10;
b=++a;	b=a++;	b=--a;	b=a--;

a	b	a	b	a	b	a	b
10 11	11	10 11	10	10 9	9	10 9	10

b=a++;
- 1st assign the value & increment the value in next line

a++ & ++a are same until & unless you are not assign the value to any other variable.

Eg: a=10;
pf(++a);
o/p: 11

a=10;
pf(a+1);
o/p: 10;

Ex: `t---*(a++)++t-1----`

post increment applied only on variable 'a'

Ex1: a=10;
f(++a)
P.V = 11
a.V = 11

a=10;
f(a++);
P.V = 10
a.V = 11

a=10;
f(a+1);
Passing value (P.V) = 11
assign value (a.V) = 10

Gate 2014:

```
void main()
{
  int m=10;
  int n,n1;
  n= ++m;
  n1= m++;
  n--; // 10
  --n1; // 10
  n-=n1; n=n-n1 → n=10-10
  printf("%d",n); //
}
```

m 12 | n 11 | n1 11

O/P=0 //

Ex1: int a=5

printf(++a, ++a, ++a);

8 7 6

6 7 8

bad program

Ex2: int a=5;

printf(a+1, a+2, a+3);

6 7 8

6 7 8

good program

Ex: int a=1;

pf(++a * a++ * ++a);

→ Compiler dependent O/P

Bad program

Note

1. Program should work based on logic not based on compilers
2. Incrementing same variable multiple time in a single line is consider as bad programming.

Ex.: void main()

```
{
  int a=5;
  printf("%d", ++(a++)); //error
  printf("%d", (a+1)++); //error.
}
```

++5

6++

Note

- the pre increment & post increment will work only on the variable but not on the constant, But not on the expression

Loops

1. For
2. while
3. dowhile

Ex. WAP to print 1 to 10 using above loops

1)
```
void main(){
  int i;
  For(i=1;i<=10;i++)
    Printf("%d",i);
}
```

2)
```
void main(){
  int i=1;
  while(i<=10){
    printf("%d",i++);
  }
}
```

3)
```
void main(){
  int i=1;
  do{
    printf("%d",i++);
  }while(i<=10);
}
```

also be written in

1.
```
void main(){
  int i;
  For(i=1;i<=10;printf("%d",i++));
}
```

i<=10 — if true go

3 ≤10
2 ≤10
1 ≤10

i = 1, 2, 3 ... 11

O/p: 1, 2, ... 10

Ex.
```
void main(){
  int a=5;
  while(a++<=7);
  printf("%d",a);
}
```

a: 5 → 6 → 7 → 8

O/p: 8

Ex.
```
void main(){
  int a=1
  do{
    while(a++<1);
  }while(a++<=2);
  Printf("%d",a);
}
```

O/p: 5

a: x → 2 → 3 → 4 → 5

Note

```
int a(){
  ....
  ....
  ....
  return x;
}
```

For(a(); a(); a())

- a() — It will execute exactly once & return value is not having any impact
- a() — It will execute min once & return value is having impact
- a() — It will execute min zero time & return value is having no impact

H-W ① input n=6

o/p should be

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

```
void main(){
  int n=6,i;
  for(i=1;i<=n;i++){
    for(int j=1;j<=i;j++){
      Printf("%d",j);
    }
  }
}
```

② if i/p=6

```
      6
    5 6
  4 5 6
3 4 5 6
2 3 4 5 6
1 2 3 4 5 6
```

```
void main(){
int n=6;i;

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

③ n=7

o/p :

```
1 . . . . . . . . . . . 1
1 2 . . . . . . . . . . 2 1
1 2 3 . . . . . . . . 3 2 1
1 2 3 4 . . . . . . 4 3 2 1
1 2 3 4 5 . . . . 5 4 3 2 1
1 2 3 4 5 6 . . 6 5 4 3 2 1
1 2 3 4 5 6 7 7 6 5 4 3 2 1
```

```
void main(){
    int n=7,i,j;
    for(i=1; i<=n ;i++){
        j=1;
        for( ;j<=i;j++){
            printf("%d", j);
        }
        for( ;j<=2(n-i);j++){
            printf(" ")
        }
        for( ;j<=2*n;j++){
            printf("%d", 2n-j+1);
        }
    }
}
```

O/P

```
1 . . . . . . . . . . . . 1
1 2 . . . . . . . . . . 2 1
1 2 3 . . . . . . . . 3 2 1
1 2 3 4 . . . . . . 4 3 2 1
1 2 3 4 5 . . . . 5 4 3 2 1
1 2 3 4 5 6 . . 6 5 4 3 2 1
1 2 3 4 5 6 7 7 6 5 4 3 2 1
```

• Switch case :-

1. case and label should have space
2. case label must end with colon (":")
3. case lable must be constant
4. case lable can't be floating point no.
5. Empty switch case is allowed
6. only one "default". is allowed & it can be placed any where in the switch
7. Default will be execute only when no case is making
8. after every case there is a requirement of break otherwise it will execute all further cases. till the occurs of break or till the end of switch.

note

▸ After entering the switch case the contro is directly transfer to the matching case; without executing any stmt b/w them.

ex:- switch(i){
    |z|+10 → there will not execute
    case 10:
    case 15
}

Ex:

```
void main(){
    int x;
    scanf("%d", &x);
    switch(x){
        case 0: x=x+1;
        case 1: x=x+2;
        default: x=x+3;
    } printf("%d", x);
}
```

I/p	O/p
x=0	0 7 6
x=1	7 6
x=2	5

Ex:

```
void main(){
    int x;
    scanf("%d", &x);
    Switch(x){
        case 0: x=x+1;
                break;
        default: x=x-1;
        case 1: x=x+10;
                break;
        case 2: x=x+100;
    }
    Printf("%d", x);
}
```

Keyword

Should have gap

I/p	O/p
x=0	0 1
x=1	11
x=2	102
x=3	3 12

Ex: void main()
{
int i;
for(i=1; i<=25; i++)
switch(i) // 1 // 14 // 20
{
case 0: i+=5;
case 1: i+=3;
case 2: i+=4;
default: i+=6;
break;
}
printf("%d", i); // 57
}
}

1≤25 | 14≤25 | 20≤25 | 26 ≤25

i → 1 → 8 → 13 → 19 → 25 → 26

Print value

at the end the of one iteration & starting of another we will increment it

- the sum of all value of printf is 57 //

Ex: void main()
{
int i, j, k=0;
j = 2*3/4+2.0/5+8/5; // 2
k -= -1 - j // -1
for(i=0; i<5; i++)
switch(i+k)
{
case 0:
case 1: printf("%d", i+k);
case 2: printf("%d", i+k);
default: printf("%d", i+k);
}
}
}

k → 0 → -1

for loop

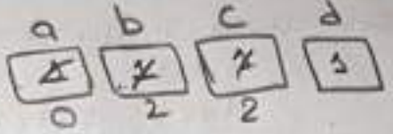
value of switch: -1 0 1 2 3

i = 0 1 2 3 4

no of time printf will be call: 1 3 3 2 1 → 10 //

- the no. of time printf() statement execute is 10

• Short circuit evaluation



Ex: a=1, b=1, c=1, d=1

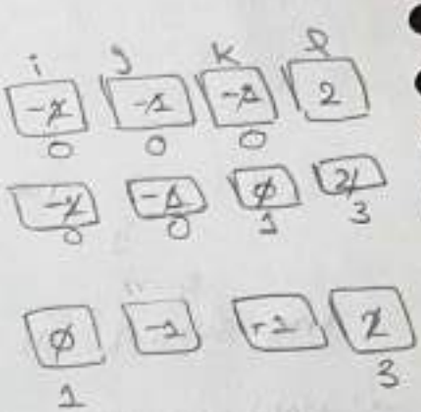
a=(b++>1 || c++>1) && d++>1;

↳ not evaluated

printf(a,b,c,d);

O/p: 0 2 2 1

Ex: int i=-1, j=-1, k=-1, l=2, m;



m=((i++ && j++ && k++) || (l++));

pf(i,j,k,l,m);

I/p	O/p i, j, k, l, m
orginal	0 0 0 2 1
if k=0	0 0 1 3 1
if i=0	1 -1 -1 3 1

Function

• the purpose of function is code reutilization

• the purpose of loop is code repeatation.

• the purpose of header file is function reutilization

• if we want to re use the function then we need to create header file of that particular functn & include that in the required program.

- the function will have the following attributes.
  - → Function prototype / declaration
  - → Function definition
  - → Function calling

```
int add(int, int);
void main()
{
  int x=10, y=20, z;
  z=add(x,y); //Functn calling
  printf("%d", z);
}
int add(int a, int b)
{
  int c;
  c=a+b
  return c;
}
```

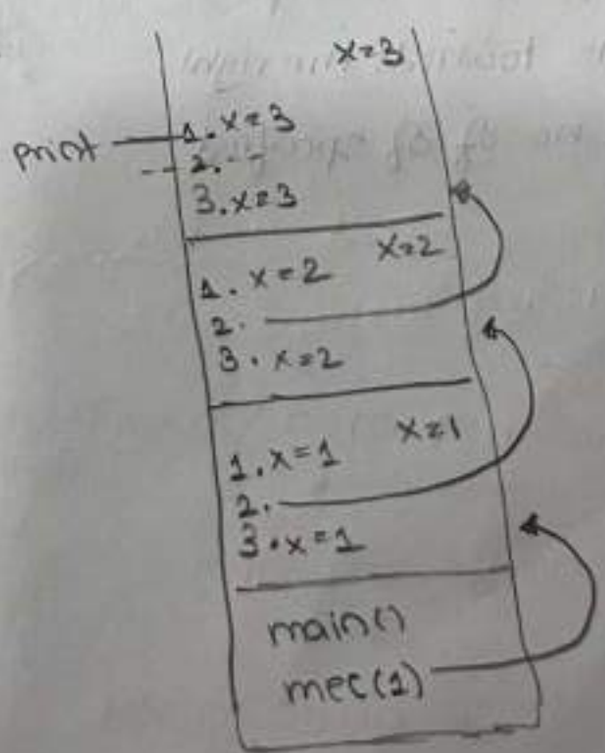
- actual parameter
- Functn definition
- formal parameter & they are called as local variable in the add function

- Prototype contain
  1) name of the functn
  2) no. of arguments
  3) type of arguments
  4) return type of argument

Ex.
```
Void mec(int x)
{
  Printf("%d",x) // 1
  if(x<=2)
    mec(x+1);
  printf("%d", x);
}
Void main()
{
  mec(1); // 123321
}
```

m(1)
↳ pf(1)
m(2) → pf(2)
pf(1) m(3) → pf(3)
pf(2) pf(3)

what is o/p printed?
123321



Recursion

- funcn calling itself is know as recursion
- Recursion should definitly have the termination condition (Base condition) otherwise it will lead to stack overflow
- recursion internal make use of stack
- every time recursive call is made activation record will be create in the stack

- Activation Record: the stack frame (memory) allocated to a recursive call is called as activation record.
- the creation & the Deletion of the activation record depend on the actual funcn calling sequence.

Bitwise operators

operator	meaning
&	Bitwise AND
|	Bitwise OR
>>	Right shift
<<	Left shift.

Ex:

```
void main()
    int a=25 ,b=12;
    printf("%d",a&b);
    printf("%d",a|b);
}
```

25: 00011001
12: 00001100
a&b 8: 00001000

25: 00011001
12: 00001100
a|b 29: 00011101

- Right shift(">>")
- right-shift operator shift all the bits towards the right by certain no of of specified bits.

Eg: 212 : 11010100

→ 0000 0000 1101 0100 } why 16bit b/c Integer take 2Bytes.

212>>2 : 0000 0000 0011 0101
⇒53

212>>7 :
↳ 0000 0000 0000 0001
⇒1

- left shift (" << "):
- left shift operator shift all bits towards left by certain no. of Specified bit.

Ex: 212 = 0000 0000 1101 0100

212<<2: 0000 0011 0101 0000
→ 848

212<<3: 0000 0110 1000 0000
→ 1696

Ex:-
```
Void main()
{
    int x=4096;
    while(x)
    {
        if(x & 1)
            printf("Hello");
        x >>= 1;
    }
}
```

How many time string "Hello" will be printed — 1

x	
~~4096~~	1
~~2048~~	2
~~1024~~	3
~~512~~	4
~~256~~	5
~~128~~	6
~~64~~	7
~~32~~	8
~~16~~	9
~~8~~	10
~~4~~	11
2	12
1	13 → print the Hello only once time

H.W] write a program to store Date, month, year of a person in a single Integer variable (Read & print)?

int dob; → 2B → 16bit

Date	month	year
1 to 31	1 to 12	i.e 2000
5bit	4bit	-1900
		100
		7bit

write the code

- Pre-processor statements.
- Preprocessor will make only substitution & it will not make any calculation
- It will recognize those statement by hash symbol
- It will done before compilation of the program

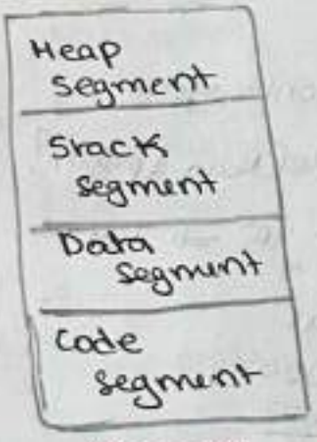
Ex: # define f1 x*x*x

```
Void main()
    int x=3,y;
      y=f1// x*x*x
      Printf("%d",y);// 3*3*3
  y
```

O/p: 27

Memory Segments

- Main memory is divided into various segment



Memory

- **Code segment**

→ In the code segment of the memory all the general instruction which will not change throughout the program execution will be store
Ex: Editors

- **Stack segment**

→ In the stack segment all the recursive functn call, local variable & formal parameter will be store

- **Data segment**

→ In the Data segment all the global variable & static variable will be store.

- **heap segment**

→ In the heap segment all the Dynamic memory allocation will be done, the memory allocation by using the standard functn like malloc(), calloc(), realloc().

Ex: int x=10;

```
void main()
{
    int x=20;
    {
        int y=30;
        printf("%d,%d", x,y);//
                                10,30
    }
    printf("%d", x);//20
}
```

Note

- for the global variable the memory will be allocated at complile time, if it is not initialized then default contain 0.
- For the local variable the memory will allocated at run time if it is not initialized then default contain garbage value.
- local variable is having high precedence as compare to global variable

Storage classes

- In the C language every variable have some characteristics like data type, scope, lifetime & default value.
- Storage classes mainly deal with scope & life time
- scope:
  the region in which variable is available & accessable is called as scope of a variable

auto — it's default for local variable
register — used for local variable; it's not default.
static, extern — they are for both local & global.

Type	Scope	Life time	default value	Storage	memory segment
auto	Body (Block)	Body	Garbage value	RAM	stack segment
register	Body	Body	G.V	Register	CPU register
static	Body	program	0	RAM	Data (Static area) segment
extern	program	program	0	RAM	Data segment

imp

- auto.

Syntax: (Keyword)

auto datatype variable_name;

Ex: auto int x;

- By Default all the local variable are auto variable
- If it is not initialized the default contain garbage value
- For every functn call auto variable is re-created & re-initialized
- It has a local scope & life time ... is within the Body

Ex:
```
void main()
{
    test(); // 10
    test(); // 10
    test(); // 10
}

void test()
{
    auto int x=10;
    printf("%d", x);
    x++;
}
```

x [1011] 1000

x [1011] 2000

x [1011] 3000

- object with the auto class are not available to the linker
- you can't use for global variable.

- auto, register, static } no linker

extern → is available to linker

- you can't use auto & register for global variable

- **Register**.

Syntax: → keywork

register data-type variable-name; → not a cmd. it's a request

Ex:
register int x;

- the keycoord (register) is request by the compiler to allocate the memory in cpu registers
- If the free cpu register are available the it will be allocated to the register otherwise it will allocated to RAM only
- register variable are faster as compare to auto variable Hence they will be used in loops & functn
- By Default register variable will behave like auto variable

- **Static**.

Syntax:

Static data_type variable_name

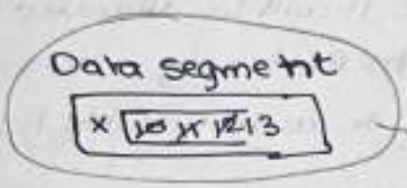
Ex: Static int x;

- the varible persistenting pervious state value even though the destruction of several functn call is called as
- Static variable is created & memory will be allocated only once in the data segment at compile time
- If it is not initialize the default contain zero.
- It has a local scope but Life time is entire program.

Ex: void main()
{
test();
test();
test();
}

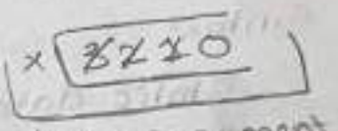
void test()
{
static int x=10;
printf("%d",x);
x++;
}

o/p: 10,11,12



Ex: void main()
{
1. static int x=3;
2. printf("%d", x--);
3. [if(x) main();]
}

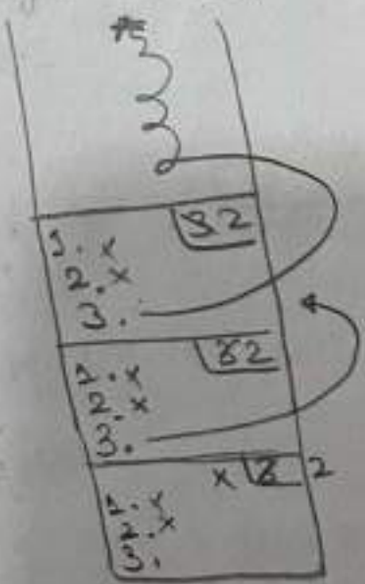
o/p: 3 2 1



Same program without static.

void main()
{
1. int x=3;
2. printf("%d", x--);
3. [if(x) main();]
}

o/p: 3 3 3 .....



Abnormal termination with stack overflow msg

```
int f(int n){
    static int i=1;
    if(n>5) return n;
    n=n+i;
    i++;
    return f(n);
}
void main(){
    printf("%d", f(1)); //7
}
```

i → 1 → 2 → 4

Note: returns means coming out of the Activation record

F(1) → 1+1
F(2) → 2+2
F(4) → 4+3
F(7) → 7 //

```
void main(){
    while(true){
        print("3");
    }
}
```

O/P: 3 3 3 ....

Infinite loop

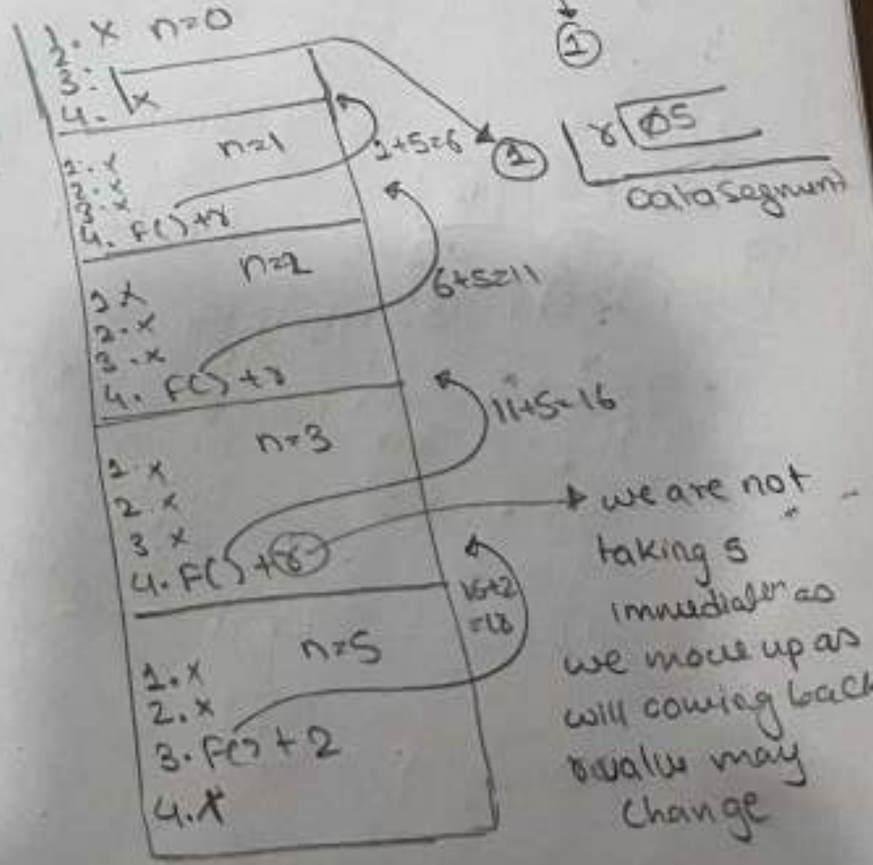
Ques)

```
int f(int n){
    static int x;
    if(n<=0) return 1;
    if(n>2){
        x=n;
        return f(n-2)+2;
    }
    return f(n-1)+x
}
```

What is the return value of f(5) 18 ?

x 5

F(5) → 18
16 → F(3)+2
11 → F(2)+5
6 → F(1)+5
F(0)+5
1



Q)
```
int incr(int i)
  static int count;
  count = count+i;
  return count;
}
void main(){
  int i,j;
  for(i=0;i<=4;i++)
    j=incr(i);
  Printf("%d",j);
}
```

count: 0 → 0 → 1 → 3 → 6 → 10

j: ϕ → 0 → 1 → 3 → 6 → 10

i=0 → j=0; i=1 → j=1; i=2 → j=3; i=3 → j=6; i=4 → j=10

what is o/p printed 10 ?

Q)
```
void main(){
  int i; x=5, y=10;
  for(i=1; i<=2; i++){
    y=y+f(x)+g(x);
    Printf("%d",y);
  }
}
```

```
int f(int x){
  int y;
  y=g(x);
  return(x+y);
}
int g(int x){
  static int y=5;
  y=y+7;
  return(x+y);
}
```

i: 1 | x: 5 | y: 10 | static y: 5 → 12 → 19

i=0

y = 10 + f(5) + g(5) → y = 12 + 7 → 19

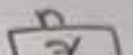
f(5): y = g(5) → return 5+17 → 22

g(5): y = 5+7 = 12; x = 5+12 = 17

return 5+19 = 24

local y: 17

Q) int x()?
Static int n =7;
return n--;
}
Void main()?
for(x();x();x())
Printf("%d",x());
}



O/p printed ?

a) 4 1
b) 6 3 0
c) 6 3
d) 5 2

GOB
41
Pgno
132

Solve it

#include <studio.h>
int total(int v)?
Static int count=0;
while(v)?
count+=v&1
v>>=1;
}
return count;
}

Void main()?
Static int x=0
int q=5;
for(;i>0;i--)?
x=x+total(i);
}
printf("%d\n",x);
}

• Extern.

• no memory will be allocated for extern variable they make use of memory of outside variable (global variable)

Ex1: int a=8
main()
{
extern int a;
pf(a);
a=20;
pf(a);
}
op: 8 20

these don't have memory

a is a local variable in the main functn & it will make use of memory of a global variable.

Ex2: main()
{
extern int a;
pf("Hello");
}
o/p: Hello.

Ex3:
main()
{
extern int a;
pf(a); ?
}
no variable declare outside
o/p: error

no variable outside so it will throw error

Ex4: int a=5;
main()
{
extern int a;
extern int a;
extern int a;
pf(a);
}

multiple declaration of same variable in the same functn is allowed only if it is extern.

o/p: 5

Ex5: main()
{
extern int a;
pf(a);
}
int a=10;
o/p: 10

if these variable not there then these will not work

Ex6: main()
{
int a;
int a;
int a;
}
X

o/p: error

Ex4 main()
{
extern int a=10;
pf(a);
}
O/p: error
Initialization of a extern variable is not allowed only assignement is allowed

Ex5: int a;
main()
{
extern int a;
pf(a);
}
O/p: 0

Declaration: D
Initialization: I
Assignment: A

Ex1 int a=20; → D,I
main()
{
int a=30; → D,I
int b; → D
b=20 → A
B();
}
B()
{
static int c=30 → D,I
a=10; → A
}

Global	B()
a 20	c 30
data segment	data segment

∴ purpose of declaring the variable is to allocate the memory.

Ex2 int a=30; → D,I
main()
{
static int a=30; → D,I
B();
}
B()
{
static int a=40; → D,I
C();
}
C()
{
extern int a; → D
}

global	main()	B()
a 30	a 30	a 40

data segement.

Ex3: a=10; → D,I
main()
{
a; → D
a=20; → A
B();
}
B()
{
a=20; → D,I
b=40; → D,I
b=50; → A
}

Scope

- It is type categorized into two
  - Static scoping (Compile time)
  - Dynamic scoping (Run time)

Scoping rules are applicable **only** to the "Free variable

- Free variable
  - ↳ the variable which is not declare in the functn but used in the functn is called as Free variable

- Static scoping (faster than Dynamic scoping)
  - ↳ In the static scoping free variable are resolved by using global variable

- Dynamic scoping
  - ↳ In the Dynamic scoping free variable are resolved from pervious call functn.

Note

1. In the worst case Dynamic scoping also behave like static scoping if no variable is present in the pervious calling functn then global variable will be printed. (used).
2. C language By default follow static scoping

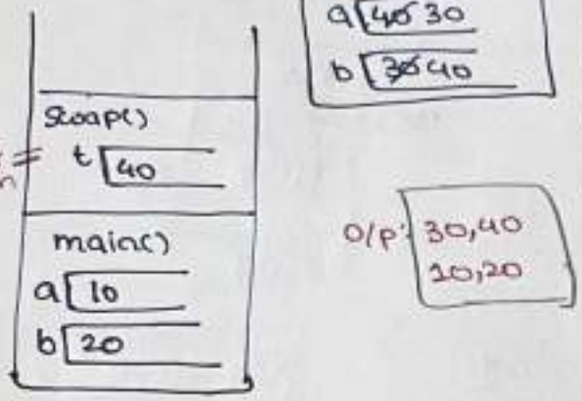
Ex:
```
int a=40, b=80;
void main(){
    int a,b;
    a=10, b=20;
    swap();
    pf(a,b);
}
```

```
swap(){
    int t;
    t=a;
    a=b;
    b=t;
    pf(a,b);
}
```

Free variable

What is o/p using Static scoping & Dynamic scoping?

Static



Dynamic

Ex: int a=5;
main()?
int a=10;
B();
}

B()?
int a=20;
C();
}

C()?
int a=30;
D();
}

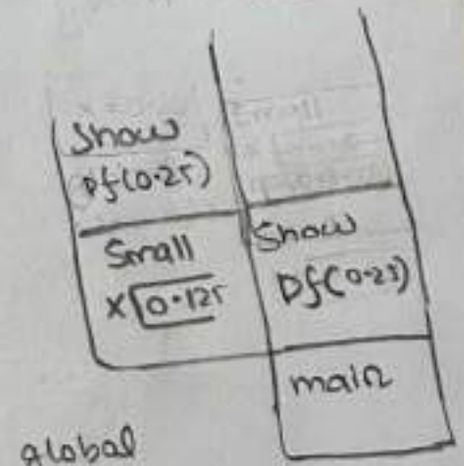
D()?
int a=50;
pf(a);
}

Static	Dynamic
5	30
	20
	10
	5

show doesn't have its own variable so go back but main also doesn't have x so refer the gloable

Ex.: variable x: real
Procedure show()
begin
pf(x)
end
procedure small()
begin
variable x: real
x = 0.125;
show();
end
begin
x = 0.25
show
small
end
main() func

Static



o/p 0.25
0.25

Dynamic

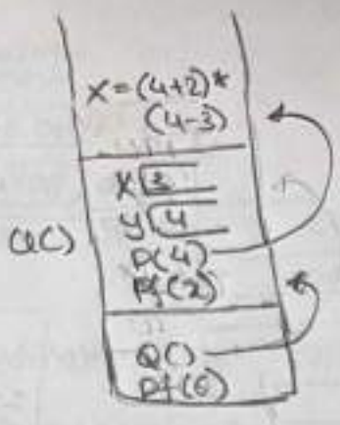
o/p: 0.25
0.125

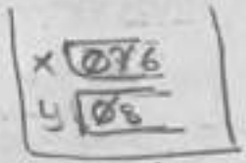
Qus] variable x,y: integer

```
procedure pc variable n: integer)
begin
    x = (n+2) * (n-3);
end

Procedure Q()
begin
    variable x,y: integer
    x=3
    y=4
    pc(y);
    Printf(x);
end

begin
    x=7, y=8;
    Q();
    Printf(x);
end
```

main

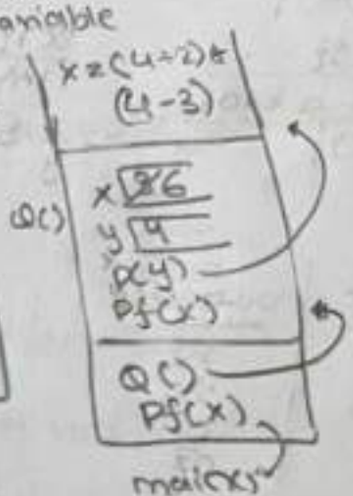
Static

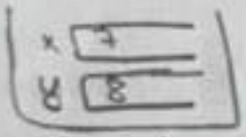




O/P: 3,6

Dynamic

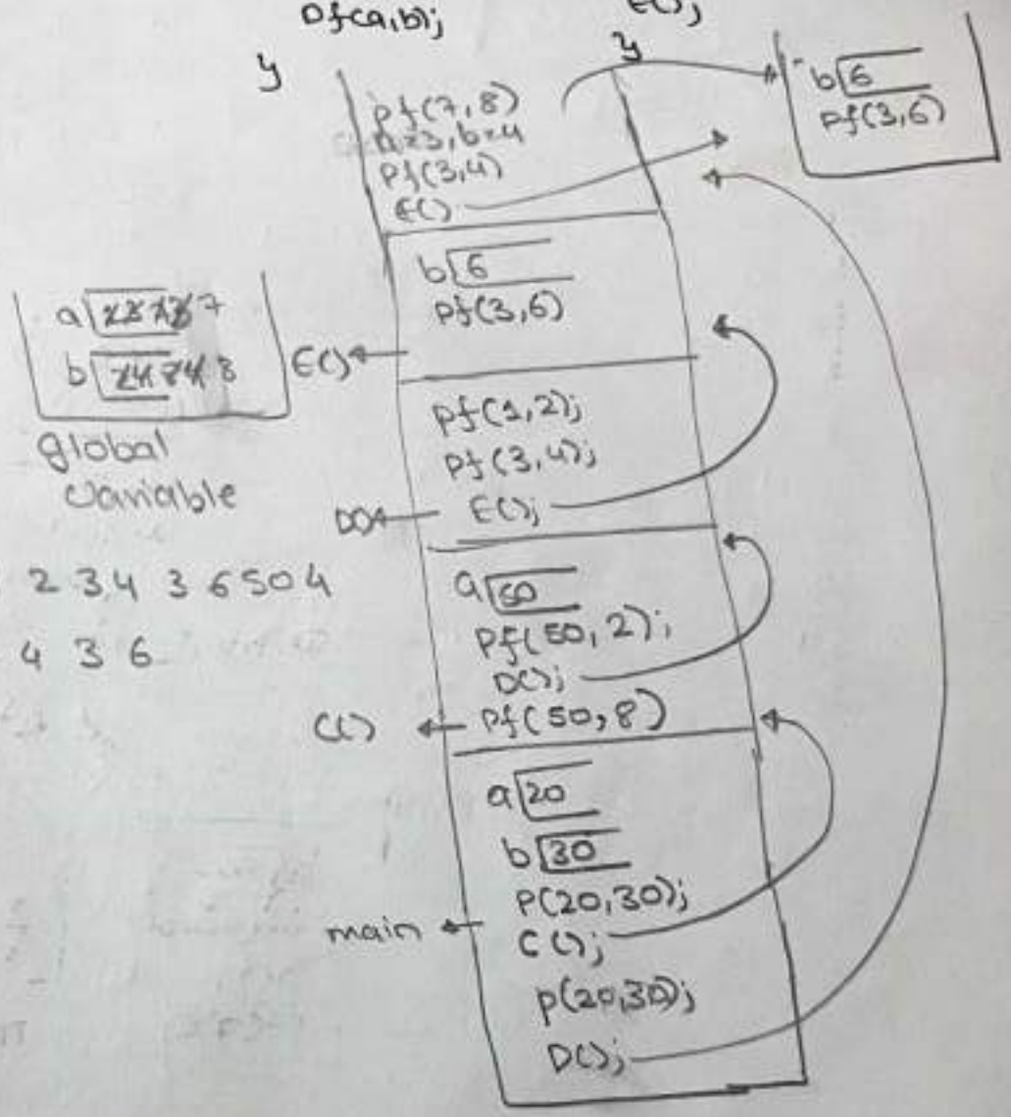




OP: 6,7

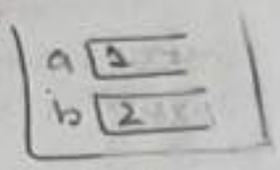
Q1

```
int a=1,b=2;
main()
{
    int a=20, b=30;
    pf(a,b);
    c();
    pf(a,b);
    D();
}
E()
{
    int b=6;
    pf(a,b);
    a=7,b=8;
}
C()
{
    int a=50
    pf(a,b);
    D();
    pf(a,b);
}
D()
{
    pf(a,b);
    a=3,b=4;
    pf(a,b);
    E();
}
```

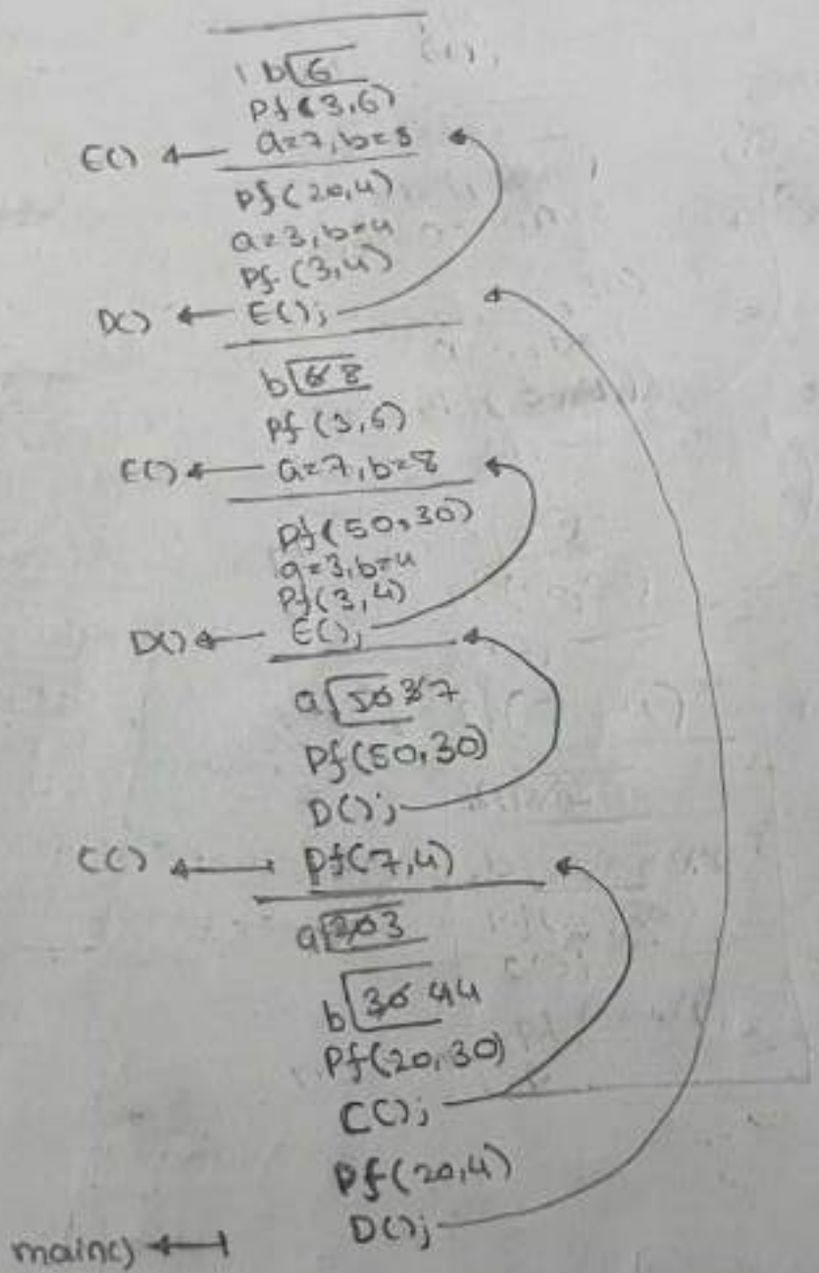


O/p 20 30 50 2 1 2 3 4 3 6 5 0 4
20 30 7 8 3 4 3 6

Dynamic



O/p 20 30
50 30
50 30
3 4
3 6
7 4
20 4
20 4
34
36



Q] int a,b;

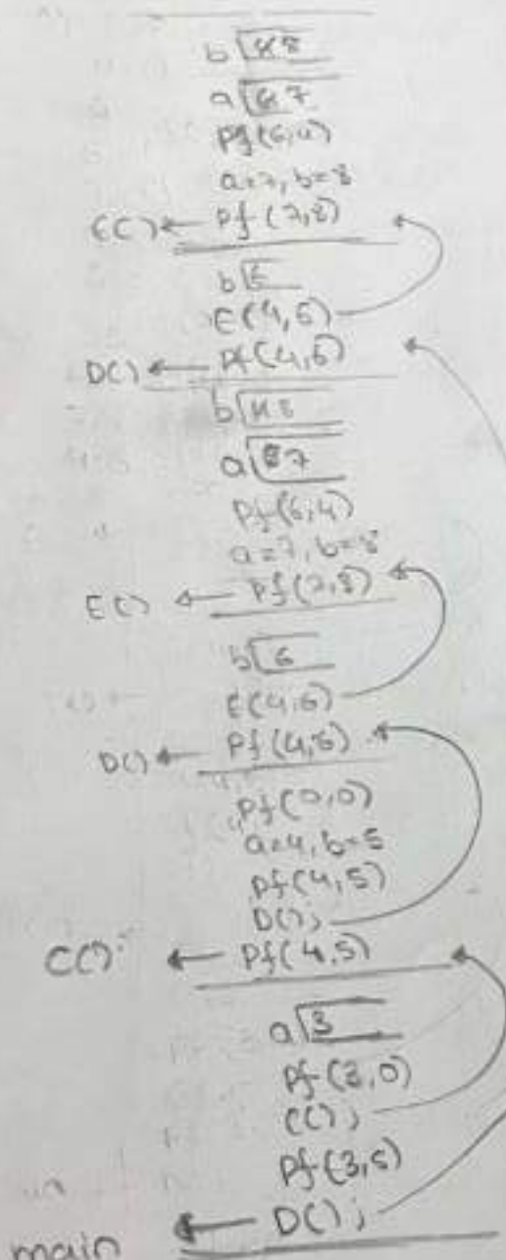
```
main(){
    int a=3
    pf(a,b);
    c();
    pf(a,b);
    D();
}
```

```
c(){
  pf(a,b);
  a=4,b=5
  pf(a,b)
  D();
  pf(a,b);
}
```

```
D(){
  int b=6;
  E(a,b);
  pf(a,b);
}
```

```
E(int b,int a){
  pf(a,b)
  a=b+a;} no use
  b=a-b;
  a=7,b=8
  pf(a,b);
```

Static

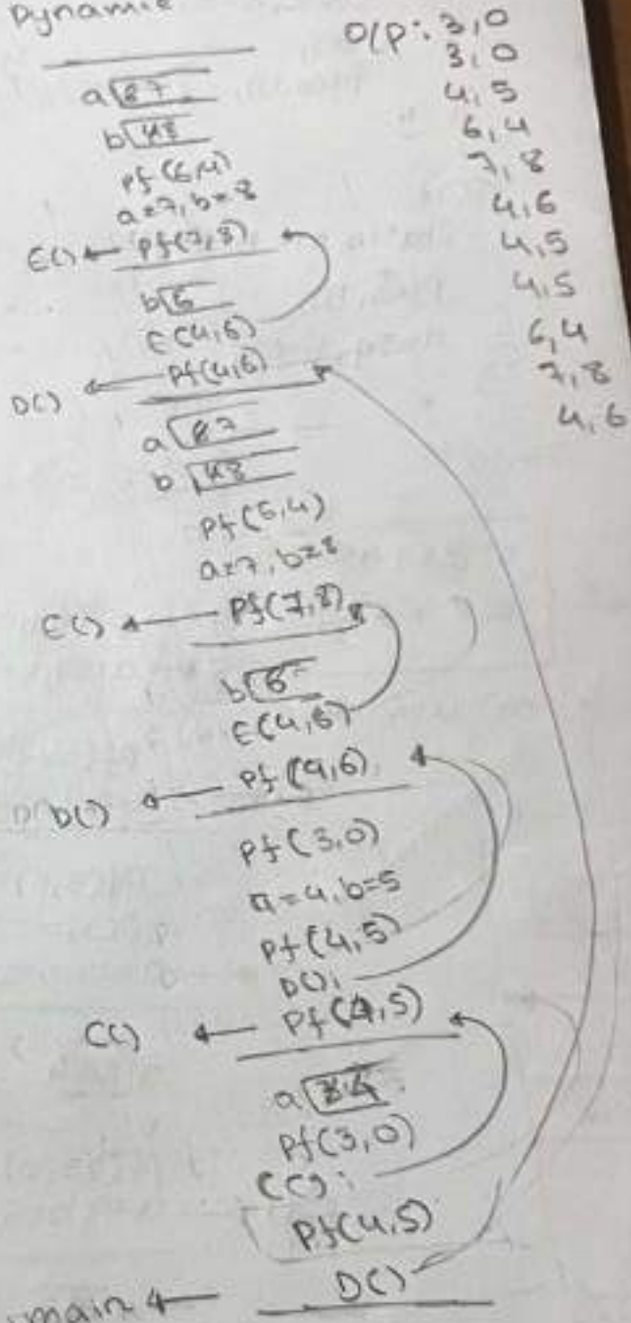


O/p: 3,0
0,0
4,5
6,4
7,8
4,6
4,5
3,5
6,4
7,8
4,6

a: 04
b: 05

global variable

Dynamic



O/p: 3,0
3,0
4,5
6,4
7,8
4,6
4,5
4,5
6,4
7,8
4,6

a: 0
b: 05

global variable

Q] int a=5, b=6;

```
main()
{
  auto int a=1;
  pf(a,b);
  C();
  a=2, b=3;
  E();
  pf(a,b);
}
```

```
E()
{
  Static int b=27;
  pf(a,b);
  a=59, b=69;
}
```

```
C()
{
  register int a=25;
  Static int b;
  D();
  pf(a,b);
  a=4, b=5;
}
```

```
D()
{
  extern int a
  pf(a,b)
  E();
  a=44, b=55;
}
```

extern is refering to global variable so if you change in extern then globle is changes

Static

a: ~~5~~ ~~59~~ ~~59~~
b: ~~6~~ ~~56~~ 69

global variable

O/p: 1,6
5,6
5,27
25,0
~~2,69~~
2,69

pf(2,69)
E() ← a:59, b=69

pf(5,29)
E() ← a=59, b=69

pf(5,6)
E();
D ← a=44, b=55

a 25 4
D();
pf(25,0)
C() ← a=4, b=5

a 1 2
pf(1,6)
C();
a=2, b=3
E();
pf(1,69)

main ←

C()
Static b
05

D()
extern a
544

E()
Static b
27 69

Dynamic

a: 5

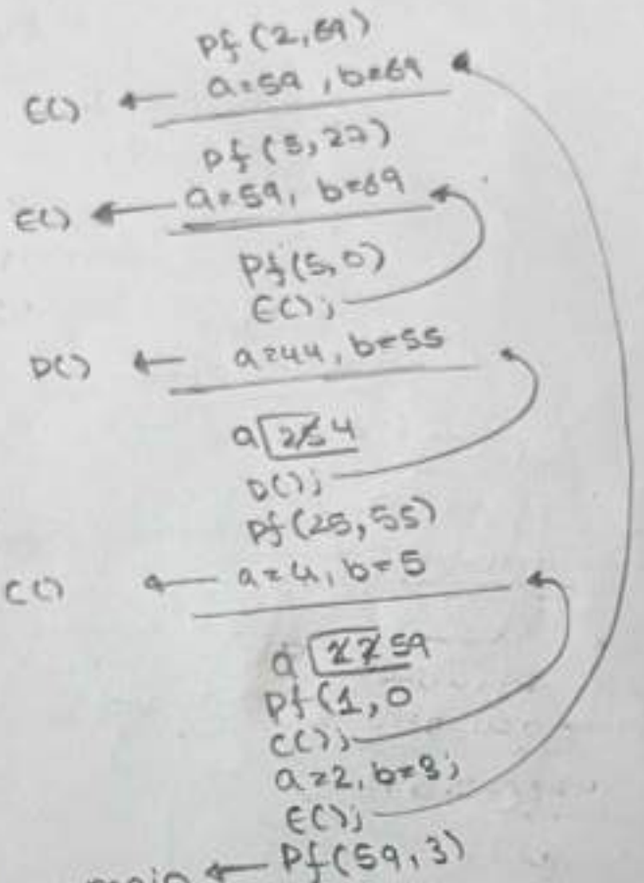
b: 63

global

C()
static b
~~0~~ 55

D()
extern a
~~2~~ 44

E()
static b
~~2~~ 69



O/P: 1,0
5,0
5,27
25,55
2,69
59,3

Q] int a=10, b=20;

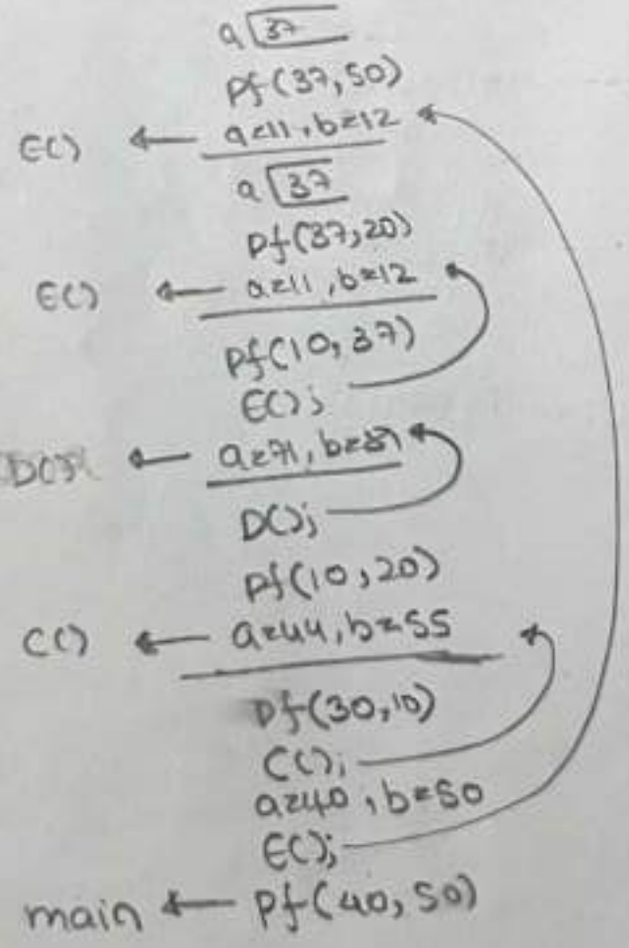
```
main()
{
  static int a=30;
  pf(a,b);
  C();
  a=40, b=50;
  E();
  pf(a,b);
}
```

```
C()
{
  extern int a;
  D();
  pf(a,b);
  a=44, b=55;
}
```

```
D()
{
  static int b=37;
  pf(a,b);
  E();
  a=71, b=81;
}
```

```
E()
{
  int a=37;
  pf(a,b);
  a=11, b=12;
}
```

o/p: 30 10
10 37
37 20
71 10 20
37 50
40 50 ×12



D()
static
b 37 81

C()
extern
a 10 44

main
static
a 30 40

a: 10 71 44
b: 20 55 50

global variable

Dynamic

O/p: 30, 20
10 37
37 37
71 20
37 50
40 12

global variable

a: 10 37 44 41

b: 20 55 50 12

main
Static
a [30 40]

C()
extern
a [10 37 44]

D()
Static
b [37 12 81]

a [37 41]
pf(37, 50)
E() ← a=11, b=12

a [37 1]
pf(37, 37)
E() ← a=11, b=12

pf(10, 37)
E();
D() ← a=71, b=81

D();
pf(71, 20)
C() ← a=44, b=55

pf(30, 20)
C();
a=40, b=50
E()
main() ← pf(40, 12)

Pointer

1. pointer is a variable which is used for to store the address
2. the two different operation are used in the pointer
   a) & → address of operator
   b) * → indirection operator / dereference operator / object-at location / value at location
3. pointer variable takes same byte of memory irrespective of its datatype (It can be integer pointer, character pointer, float-pointer).
4. Pointer variable can dereference (access) no. of bytes depending on its data type
5. Integer pointer can dereference "two" bytes of data / character pointer can dereference "one" byte of data & float pointer can dereference "4" byte of data
6. pointer variable can be declared by using ("*")
7. Address can't be negative
8. Address size will change from system to system depending on O.S.

- Pointer:
  - Basic
  - pointer to pointer
  - pointer to array
  - Array of pointer
  - multi-dimensional Array
  - pointer to string
  - pointer to structure & funct^n

- pointer variable takes same byte of memory w.r.t data-type
- & pointer variable can dereference/access no. of byte depending on the datatype

- Basic :

Ex: int *b;

b is a pointer variable of type integer where we can store address of some (any) integer

(or)

b is an integer pointer

int *b;
- Second first
- operand has high precedence
i.e b is a integer variable

Ex: int a=10; a [10] 100

int b;
b=&a // (x) b [ ] 200

∴ we can't store address of a variable in the normal integer variable.

Ex:
int a=10; a [10] 100

int *p=&a; p [100] 200

Declaration & Initialization

int *p; → Declaration
p=&a → Ass.

pf(a) = 10
pf(&a) = 100
pf(p) = 100
pf(&p) = 200
pf(*p) = 10.

"Variable name means inside value"

Ex: int a=10; a [10] 100

pf(a) = 10
pf(&a) = 100
pf(*&a) = 10

note
a = *&a

Ex: int a=10;
int *b;
b=&a;

a [10] 100
b [100] 200

pf(b) = 100
pf(a) = 10
pf(*b) = 10

pf("%d", a*b); // error
a → 10, b → 10

Compiler ask b if the left * is your or not do "b" said Check my declaration

pf("%d", a**b); // 100
a → 10, *b → *10

Ex: int a=10;
int *b= &a;

Pf(b)= 100

Pf(*b) = 10

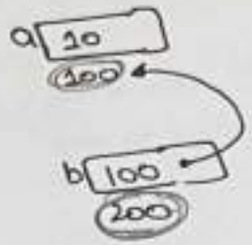
Pf(**b)= error

Pf(*&b)= 100

b

Pf(**&b)= 10.

b



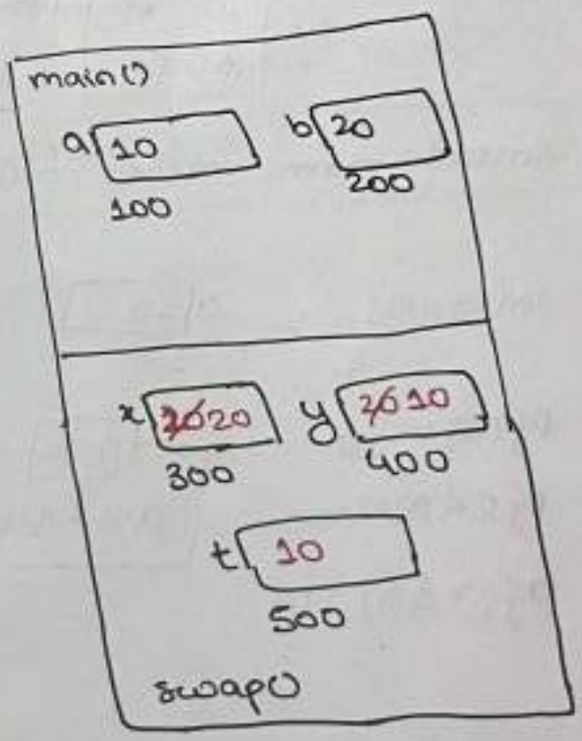
- parameter passing techniques:-
  - Call by value
  - Call by reference or Call by Address

Call By Value

```
Void main()?
  int a=10; b=20;
    Swap(a,b);
  printf("%d %d",a,b);
}
Void Swap(int x, int y)?
    int t;
    t=x;
    x=y;
    y=t;
}
```

actual parameter

Formal Parameter

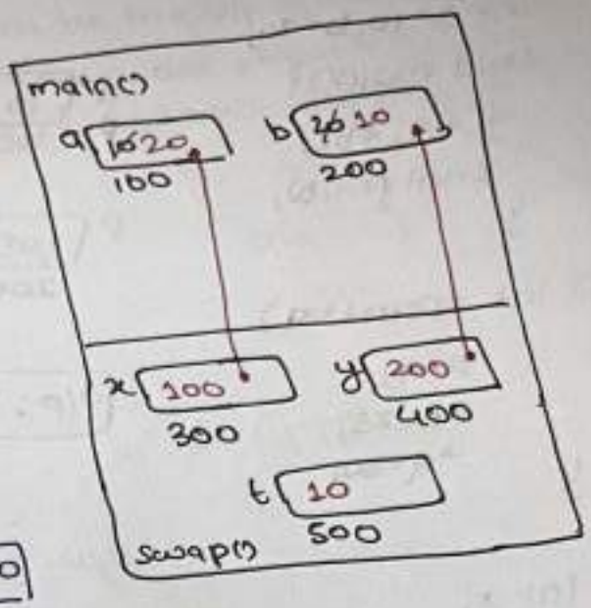


o/p : 10,20

note — Whatever the value change in the formal parameter are not reflected into actual parameter

Call by Reference

```
void main()
{
  int a=10,b=20;
  swap(&a,&b);
  printf("%d %d",a,b);
}
void swap(int *x,int *y)
{
    int t;
    t=*x;//10
    *x=*y;//20
    *y=t;//10
}
```



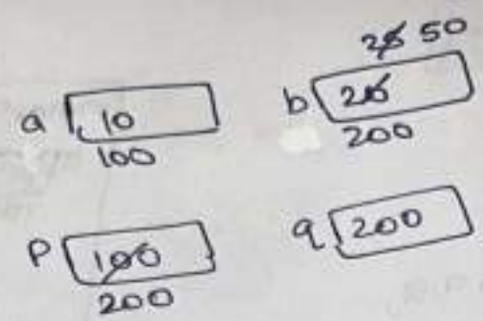
O/p: 20,10

Note

- If the Address is been passed than receiver should be pointer variable
- C language support only call by value but we can achive call by reference By passing the address
- In the above program actual parameter are getting change b/c of call by reference.
- In the question if the call by reference is given then whenever parameter is passed in the functn we need to understand implict that address is been passed receiver is a pointer variable & actual parameter will be change

Ex:

```
int a=10, b=20;
void main()
{
  f(&a, &b);
  printf(a,b);
}

f(int *p, int *q)
{
  p=q;
  *p=25;
  *q=50;
}
```

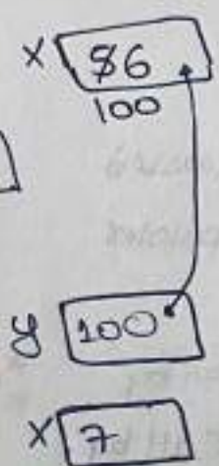


O/P: 20,50

Q)

```
int x;
void Q(int z)
{
  z=z+(x);
  print(z); //12
}
void p(int *y)
{
  int x=*y+2; // 5+2
  Q(x); //7
  *y=x-1; //6
  print(x); //7
}
main()
{
  x=5
  P(&x);
  print(x)//6
}
```

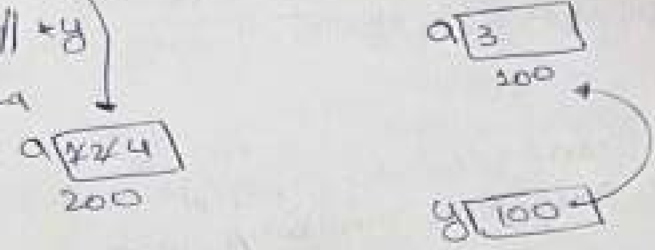
free variable



O/P: 12

Q) a = 3;
D.S ← void n(a) { x [200]   What will be the o/p of the program when parameter are passed by reference & dynamic scoping.

```
  x = x * a; // *x = *x * a
                = 2x2
                = 4
  printf(a); // *x
}
void m(y) {  [4] → // *y
D.S ← a = 1;
  a = y - a; // a = *y - a
  n(a); // 200
  printf(a); // (4)
}
void main() {
  m(a); // 100
}
```

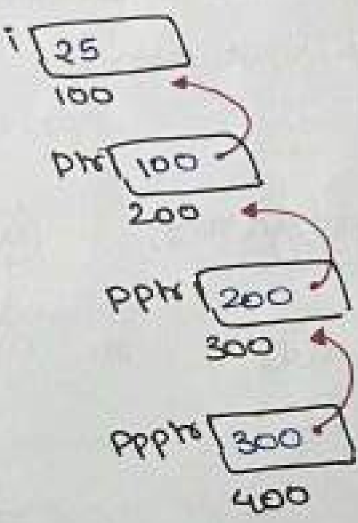


o/p ⇒ 4, 4

Pointer to pointer

Ex: void main() {

```
  int i;
1. int *ptr;
2. int **pptr;
3. int ***ppptr;
  i = 25;
  ptr = &i;
  pptr = &ptr
  ppptr = &pptr
}
```

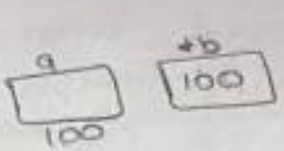


1. i = 25
2. ptr = 100
3. &ptr = 200
4. &i = 100
5. *ptr = 25
6. **ptr = Error
7. pptr = 200
8. & pptr = 300
9. *pptr = 100
10. **pptr = 100
11. ppptr = 300
12. &ppptr = 400
13. *ppptr = 200
14. **ppptr = 100
15. ***ppptr = 25
16. ****ppptr = Error
17. ***&ppptr = 25

1. these is called as single pointer, in these we need to store Address of the integer & it can access upto 1 level
2. It is called as double pointer or "pointer to pointer", In these we need to store address of single pointer & it can access upto 2 level
3. It is called as triple pointer or "pointer to pointer to pointer", In these we need to store address of double pointer & it can access upto 3 level.

Gate:

```
void main()?
  int a;
  int *b = &a;
  scanf("%d", b);   // can be replace by &a
  printf("%d", *b+10); // value entered + 10
```

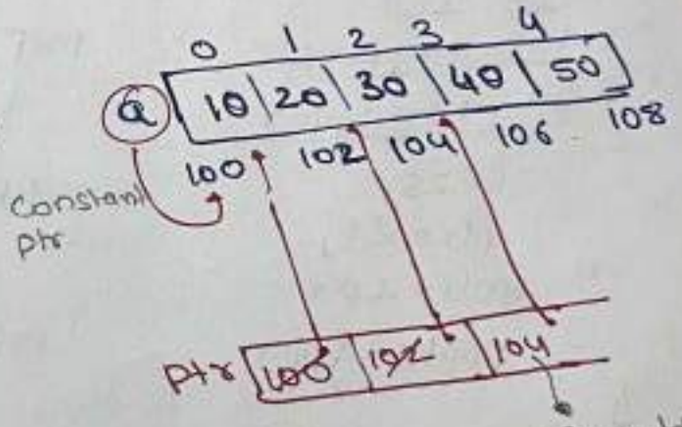


Note

In the scanf we need to provide address of the variable in which the data should be store.

- pointer to array:
- Array name without subscript ([]), contains constant base address of the array, this base address can't be modified
- Array name is considered as constant pointer
- Array name means base address.

```
void main()?
  int a[5] = {10, 20, 30, 40, 50};
  int *ptr;
  ptr = a; // ptr = &a[0]
  ++ptr;
  ++ptr;
  -- *ptr;
  printf("%d", *ptr); // 29
}
```



pf(a) = 100

pf(ptr) = 100

++ptr ✓

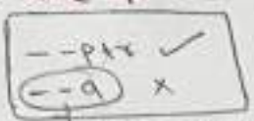
++a X

ptr increase with the respect of the size it is stored

unary operator association is right to left

Note

1. when the ptr. variable is incremented then it will increment with respect to the size of element to which it is pointing to
2. unary associativity is from right to left
3. array name is given as constant ptr b/c that is the only source to the memory that how the array contain can be access.
4. In the above example 'ptr' & 'a' both are ptr to array but 'ptr' is a pointer variable & 'a' is the pointer constant.



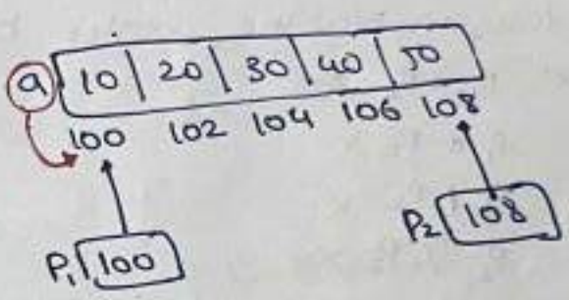
here we are try to assign the increment value

pf("%d", a+2); → 102 ✓ allow b/c here we are not assign the value, we are just trying to print the value

Pointer Arithmetic

Ex: void main()

```
{
    int a[5] = {10,20,30,40,50};
    int *p1 = a;
    int *p2 = a+4;
    printf("%d", p2-p1);
}
```



O/p: 108-100 → 8 → 8/2 → 4

1) we can add integer constant to an ptr. it means skip the element in the forward direction

Ex: a+1

$P_1 = P_1 + 2$

$= 100 + 2$

$\neq 100 + 2*2$ ↳ size of element

$\neq 104.$

2) we can subtract integer constant from the pointer it means skiping the element in the backward direction.

Ex_2: $P_2 = P_2 - 2$

$= 108 - 2$

$\neq 108 - 2*2$ ↳ size of Element

$\neq 104$

3. we can perform subtraction blw 2 pointer, these is allowed only when both are pointing to same array & $P_1 \leq P_2$ & "size of the element should be same".

Ex: $P_2 - 9$
$= 108 - 100$
$= 8$
$\Rightarrow \frac{8}{2} = 4$

[these operation gives no of elements from P_2 before P_1]

Ex: $P_2 - P_1$
$\Rightarrow 108 - 100$
$\Rightarrow 8$
$\Rightarrow \frac{8}{2}$ → size of Element
$\Rightarrow 4$

4. we can't add floating point no. to the pointer b/c it has no meaning.

Ex: $P_1 + 2.5$ ✗
$P_2 - 2.5$ ✗

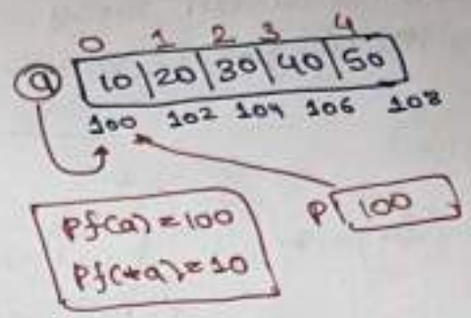
5. we can't perform addition, multiplication & division blw the pointer b/c it has no meaning

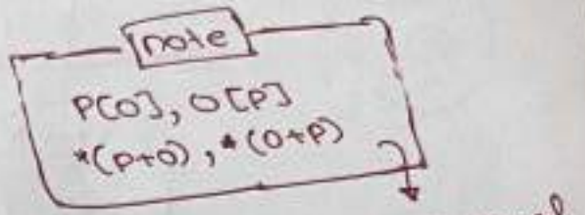
Ex: $P_1 + P_2$ ✗
$P_1 * P_2$ ✗
P_1 / P_2 ✗
$P_2 \% P_2$ ✗

Passing an array to the function

6. in the clanguage if we want to pass an array elements to the functn. then it is not a good idea to pass all the array element instead by simply passing the base address of the array we can access all the array element

Ex: Void main()
{
int a[5] = {10,20,30,40,50};
display(a,5); // 100,5
}
Void display(int *p, int n)
{
for(i=0; i<n; i++)
printf("%d", p[i]);
}
o/p : 10,20,30,40,50

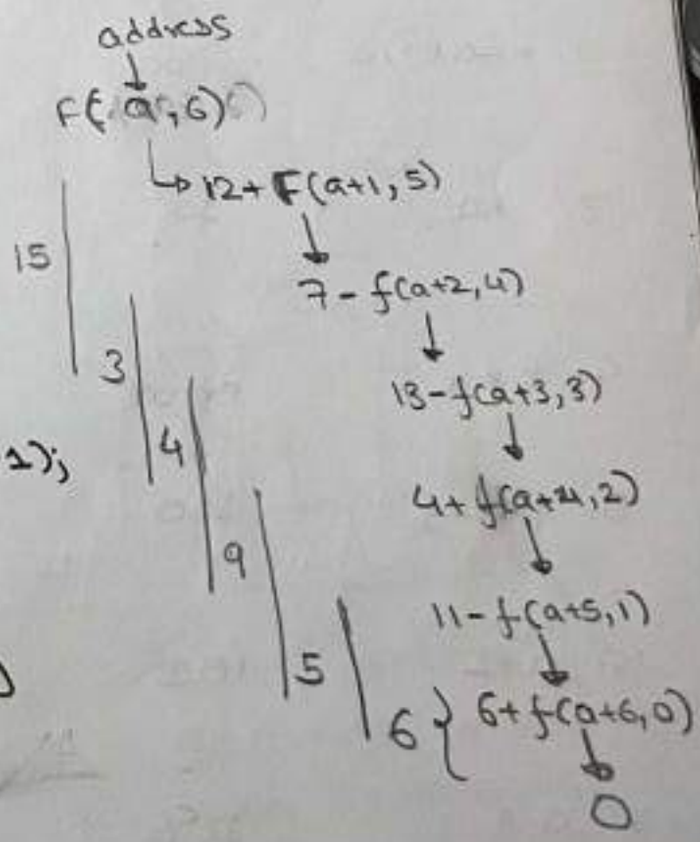




p[0]	p[1]
*(P+0)	*(p+1)
*(100+0)	*(100+2)
*(100)	*(102)
10	20

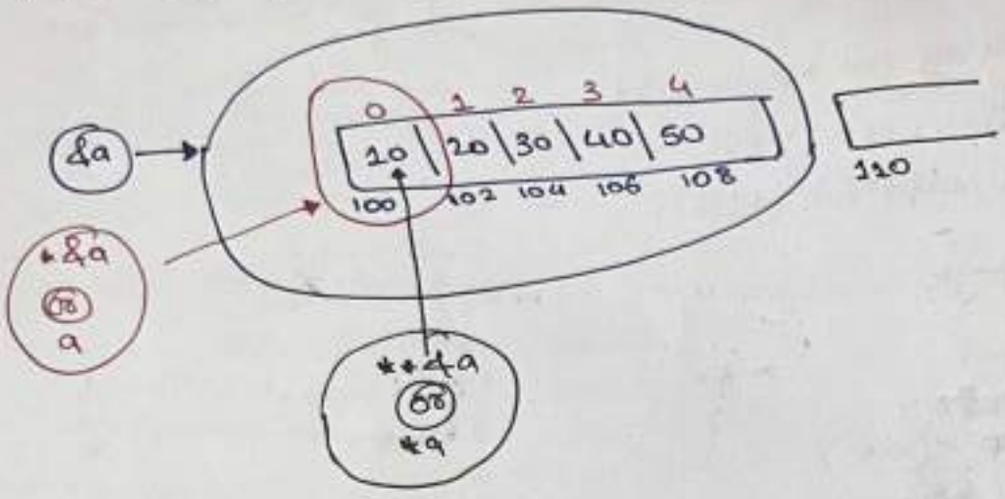
Gate

Q] int f(int *a, int n)
{
if(n<=0) return 0;
else if(*a %2==0)
return *a + F(a+1, n-1);
else
return *a - f(a-1, n-1);
}
main()
{
int a[6]={12,7,13,4,11,6};
printf("%d", f(a,6));
}
o/p : 15



• 1-Dimensional Array:

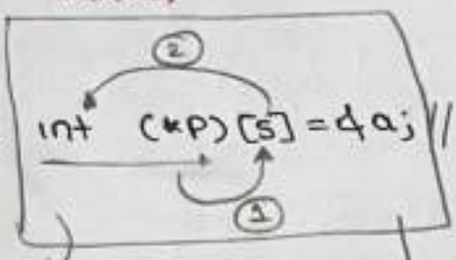
Eg: int a[5] = {10, 20, 30, 40, 50};

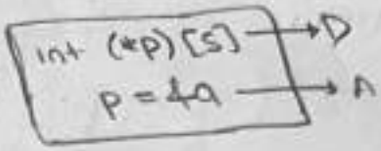


Expression	Print value	Type	meaning
1. &a	100	int (*)[];	Baseaddress of the entire 1D array
2. *&a (or) a	100	int *;	Base address of the zeroth element of the array
3. *a	10	int	value of the zeroth element of the array
4. **a	error	?	referring Illegeal location
5. &a+1	110	int (*)[];	Base address of the next continous 1D array
6. a+1	102	int *;	Base address of the 1st element of array
7. a+3	106	int *;	Base address of the 3rd element of a array

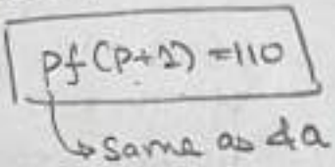
	Expression	print value	Type	meaning
8.	*(a+3)	40	int	value of the 3rd element of an array
9.	a[2] ⇒ *(a+2)	30	int	value of the 2nd element of an array

Ex: int (*p)[5] = &a; // p is a pointer to a array of 5 element where every element is an Integer





Both Declaration & Initialization



Ex: *(&a+1)-a

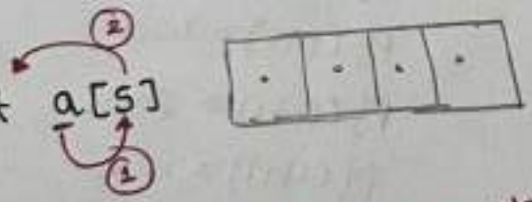
⇒ 110-100

⇒ 10

Array of pointers

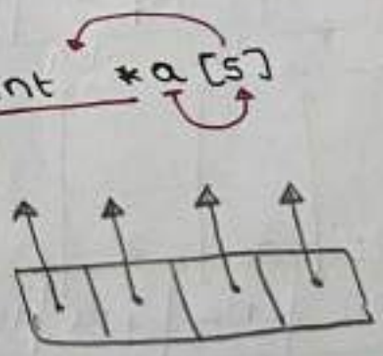
- Just like we had array of integers similarly we also have array of pointer

Eg int a[5]



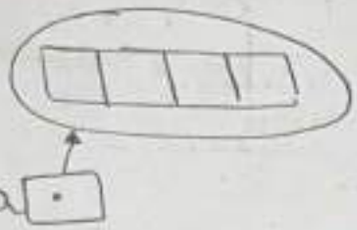
a is array of 5 element where every element is integers

- Ex: int *a[5]



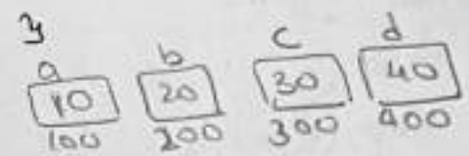
a is a array of 5 element where every element is integer pointer

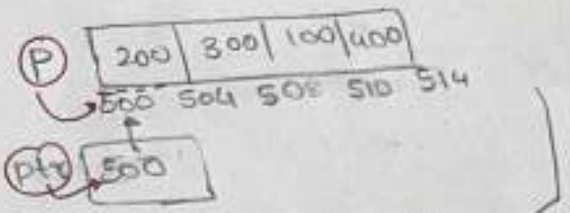
Ex: `int (*a)[5];`



a is a pointer to array of 5 elements where every element in that array is an integer.

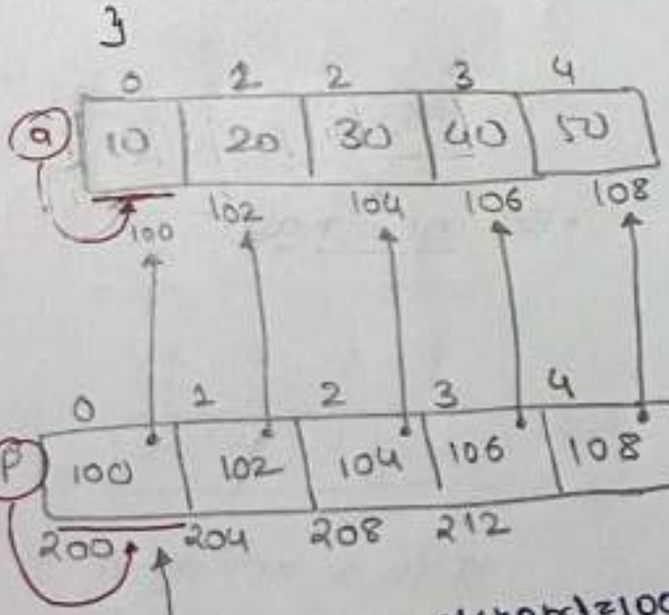
Ex:
```
void main()
{
int a=10, b=20, c=30, d=40;
int *p[4]={&b,&c,&a,&d};
int **ptr=p;
```





pf(a) = 10
pf(P) = 500
pf(ptr) = 500
pf(p+1) = 504
pf(ptr+1) = 504
pf(*ptr) = 200
pf(*p) = 200

Ex: main()
```
{ int a[5] = {10,20,30,40,80};
int *p[5] = {a, a+1, a+2, a+3, a+4};
int **ptr;
ptr = p;
```



pf(a) = 100
pf(p) = 200
pf(ptr) = 200
pf(p+1) = 204
pf(a+1) = 102
pf(ptr+1) = 204
pf(ptr-p) = 200-200
⇒ 0 = 0 (4) size of

pf(*ptr) = 100
pf(*p) = 100
pf(*a) = 10
pf(*p-a) = 100-100
= 0 = 0 (2) 5.0€

pf(*ptr-a)
⇒ 100 - 100
= 0 = 0 (2) 8.0€

Ex: void main()

```
int a[5] = {0,1,2,3,4};
int *p[5] = {a, a+1, a+2, a+3, a+4};
int **ptr = p;
ptr++; //504
```

1,1,1 ← `Pf("%d %d %d", ptr-p, *ptr-a, **ptr);`

```
*ptr++; // *(ptr = ptr+1)
```

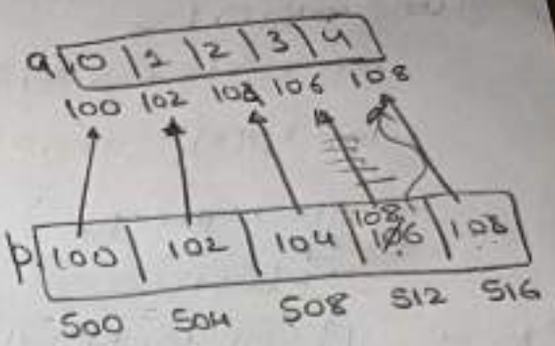
2,2,2 `Pf("%d %d %d", ptr-p, *ptr-a, **ptr);`

```
*++ptr; // *(ptr = ptr+1)
```

3,3,3 `Pf("%d %d %d", ptr-p, *ptr-a, **ptr);`

```
++*ptr; *ptr = *ptr + 1
```

3,4,4 `Pf("%d %d %d", ptr-p, *ptr-a, **ptr);`



ptr: ~~500~~ ~~504~~ ~~508~~ 512

504 − 500 ⇒ $\frac{4}{4} = 1$

102 − 100 ⇒ $\frac{2}{2} = 1$

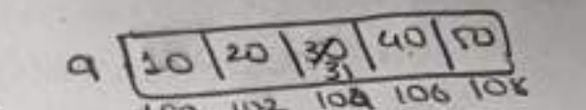
1 1

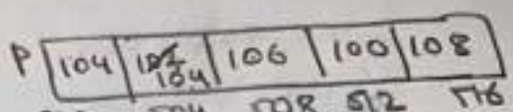
⇒ 508 − 500	104 − 100	2
⇒ $\frac{8}{4} = 2$	$\frac{4}{2} = 2$	

⇒ 512 − 500	106 − 100	3
⇒ $\frac{12}{4} = 3$	$\frac{6}{2} = 3$	

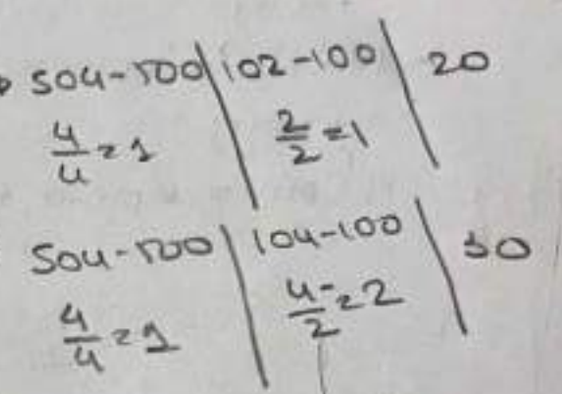
512 − 500	108 − 100	4
$\frac{12}{4} = 3$	$\frac{8}{2} = 4$	

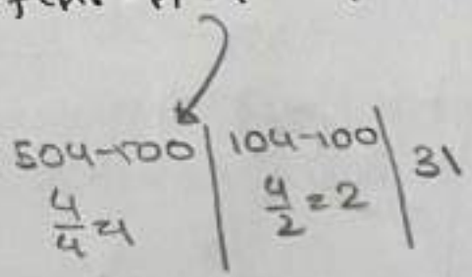
Q] void main()
{
int a[5] = {10, 20, 30, 40, 50};
int *p[5] = {a+2, a+1, a+3, a, a+4};
int **ptr = p;
**(ptr++); 500
1,1,20 pf(ptr-p, *ptr-a, **ptr);
*++*ptr;
1,2,30 pf(ptr-p, *ptr-a, **ptr);
++**ptr;
1,2,31 pf(ptr-p, *ptr-a, **ptr);
}





ptr 500/504





note

- if the array is initialized remaining all will be store as 0.
- if it is not initialized then it will contain garbadge value.

Ex: void main()
{
int a[5] = {};
pf(a[2]); o/p: 0
}

Ex1. void main()
{
int a[5];
pf(a[2]); o/p: Garbage value
}

void main()
{
int a[5] = {10, 20};
pf(a[1], a[3]);
}
o/p: 20, 0

- multi dimensional Array.
- array name without subscript contains base address of 0th element

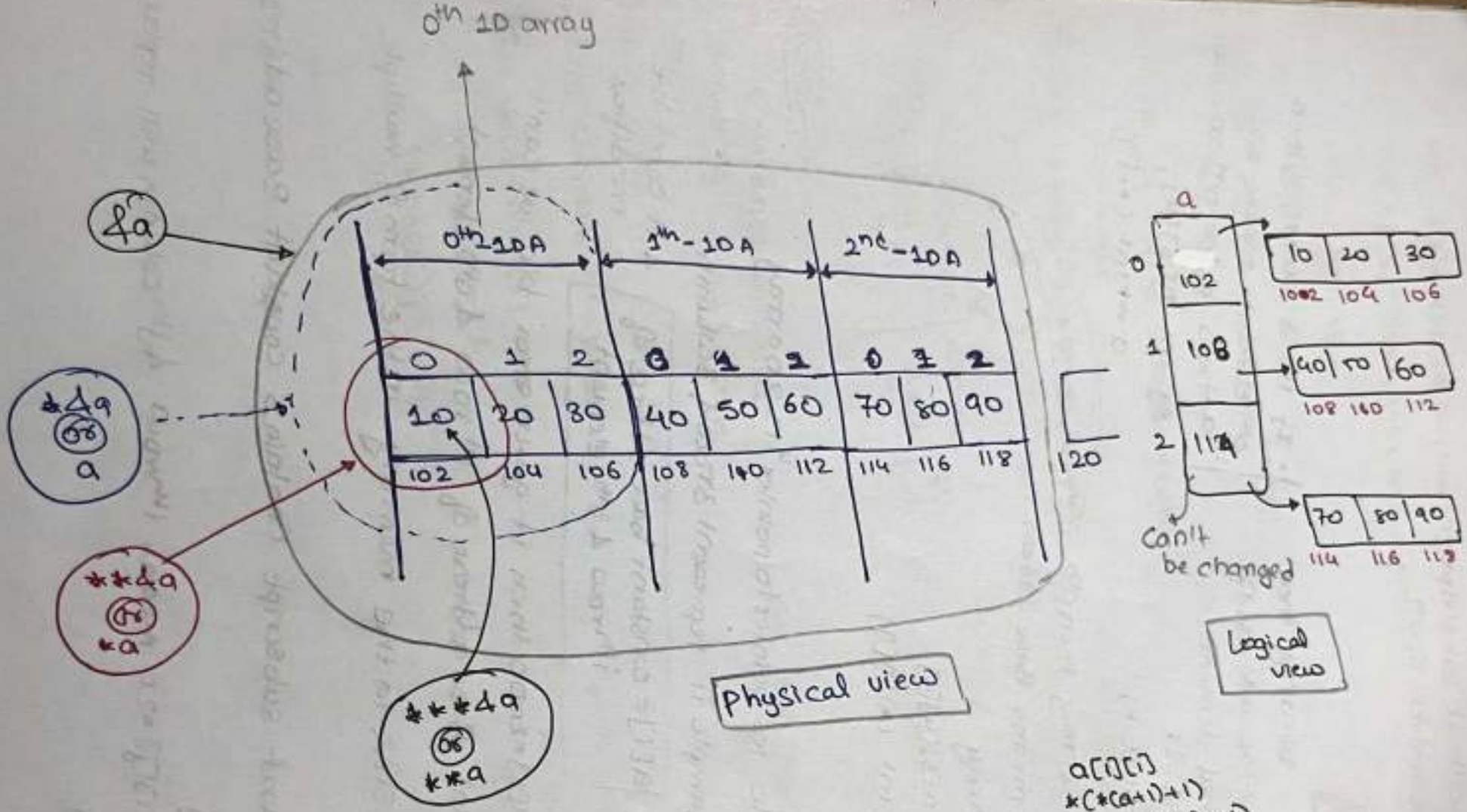
0th element

- If it is 1D array then it mean Base address of the normal element
  eg: int a[5];
  a ⟹ int *;
- If its 3D array then 0th element means Base address of 2D array
  Ex: int a[3][4][5]
  a ⇒ int (*)[][];

- If its a 2D array then 0th element means base address of the 0th 1D array
  eg: int a[3][4];
  a ⇒ int (*)[];

2D array

1. the best example for "pointer to pointer" is 2D array
2. In the above example it represent 3 row & 3 column
3. in the 2D array **a[i] = constant base address of ith row & 0th element** & it can't be modified
4. if we perform a[1]=2000 then it gives error bic we can't modified constant Base address of 1st row & 0th element
5. 2D array will be store in the memory in the form of multiple 1D array
6. array name without subscript contains constant Base address of 0th 1D array
7. If we perform a[i][j]=20 means ith row & jth column will replace the value with 20.



a[][]

((a+i)+j)

((102+i)+j)

→ 0th - 1D array

((108)+j)

→ 1st - 1D array

*(108+j)

→ 108+j → 0th element of 1D array

*(110)

50 → 1st element of 1D array

Expression	print value	Type	meaning.
1. &a	102	int(*)[][]	Base address of the entire 2D array
2. *&a OR a	102	int(*)[]	Base address of the 0th 1D array
3. *a	102	int *	B.A of the 0th element of the 0th 1D array
4. **a	10	int	value of 0th element of the 0th 1D array
5. ***a	error	?	referring unknow locatn
6. &a+1	120	int(*)[][]	B.A of the next continous 2D array
7. a+1	108	int(*)[]	B.A of the 1st 1D array
8. *(a+1)	108	int *	B.A of 0th element of the 1st 1D array
9. *a+1	104	int *	B.A of the 1st element of 0th 1D array
10. *(a+2)+1	116	int *	B.A of the 1st element of 2nd 1D array
11. a[0]	102	int *	B.A of the 0th element of 0th 1D array
12. *(*(a+1)+2)	60	int	value of 2nd element of 1st 1D array
13. **&a	10	int	value of 0th element of 0th 1D array
14. *(a+2)	114	int *	B.A of the 0th element of 2nd 1D array

Ex: *(&a+1)-a

→ 118 - 100

→ $\frac{18}{6}$ = 3

↳ SOE 1D array

Ex: **(&a+1) - *a

118 - 100

→ $\frac{18}{2}$ = 9

↳ SOE

Ex: int (*p)[3][3] = &a;

- p is the pointer to the 2D array of size 3×3 where every element is an integer

ex: pf("%d", p+1) // 120

expand → a[2][2]

((a+2)+2)

((102+2)+2)

↳ 0th element of 1D array

((112)+2)

↳ 2nd 2D array

*(112+2)

↳ 0th element of 2nd 2D array

*(118)

↳ 2nd element of 2nd 2D array

90 //

H.W

1) int a[4][5] = {0, 1, ... 15}; B.A = 200, SOE = 2B

print all expression

2) int a[5][6] = {0, 1, 2, ... 29}, BA = 3000, SOE = 4B

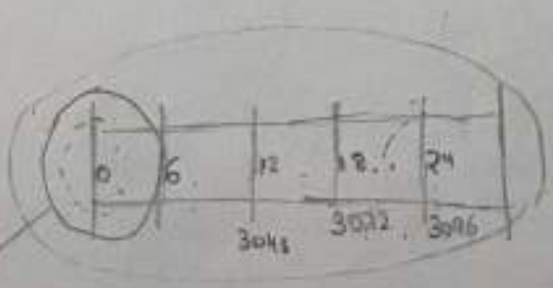
a) pf(*(a+4)) : 3096

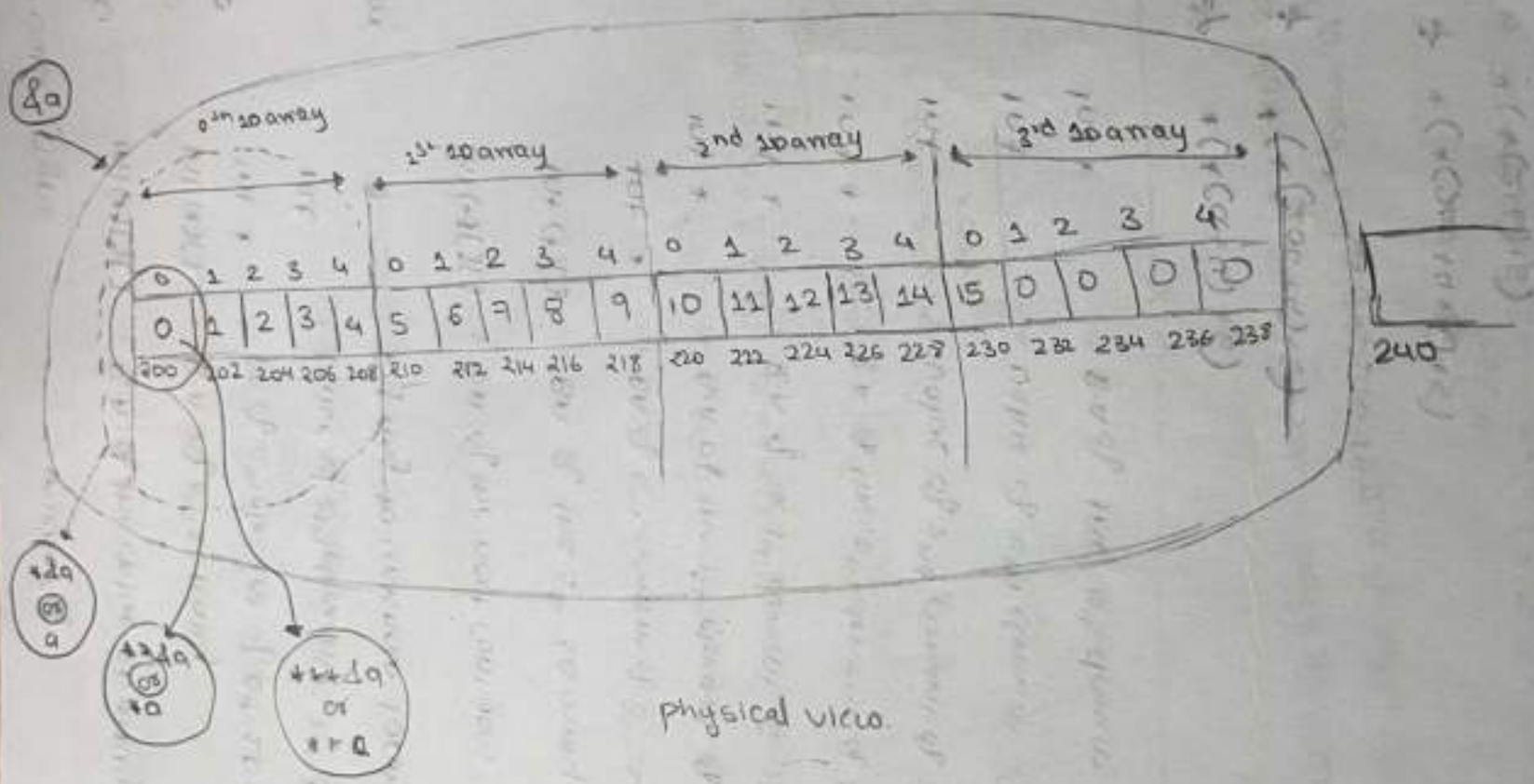
b) pf(*(a+3)+2) : 3080

c) expand a[4][5]

d) pf(*(*(a+2)+3)); 15

4 address for these is 3060





physical view.

Expression	print value	Type	meaning
1. &a	200	int(*)[][]	→ B.A of the entire 2D array
2. *&a/a	200	int(*)[]	→ B.A of 0th 1D array
3. *a	200	int *	→ BA of 0th element of 0th 1D array
4. **a	10	int	→ value of 0th element of 0th 1D array
5. ***a	error	?	→ Refering to unknown location
6. &a+1	240	int(*)[][]	→ B.A of the next continous 2D array
7. a+1	210	int(*)[]	→ BA of the 1st 1D array
8. *(a+1)	210	int *	→ BA of 0th element of 1st 1D array
9. *a+1	202	int *	→ B.A of the 1st element of 0th 1D array
10. *(a+2)+1	222	int *	→ B.A of the 1st element of 2nd 1D array
11. a[0]	200	int *	→ B.A of the 0th element of 0th 1D array
12. *(*(a+1)+2)	7	int	→ value of 2nd element of 1st 1D array
13. ***&a	0	int	→ value of 0th element of 0th 1D array
14. *(a+2)	220	int *	→ B.A of the 0th element of 2nd 1D array

---

(2) Expand a[4][5]

⇒ *(*(a+4)+5)

⇒ *(*(3000+4)+5)

↳ 0th - 1D array

⇒ *(*(3000+4*2)+5)

↳ S.O.E

⇒ *(*(3096)+5)

↳ 3rd - 1D array

⇒ *(3096+5)

↳ 0th element of 3rd 1D array

⇒ *(3096+5*4)

⇒ *(3116)

↳ 5th element of 3rd 1D array

⇒ 28

Ques
gate2020

```
int main()?
  int a[4][5]={{1,2,3,4,5},
               {6,7,8,9,10},
               {11,12,13,14,15},
               {16,17,18,19,20}};
  printf("%d", *(*(a+*(a+2)+3)));
}
```

o/p: 19 //

((a+3)+3) ⇒ a[3][3] = 19

gate
Ques

```
void main()?
  int a[5][3]={10,20,30,---100};
  printf("%d", ((a==a[0]) && (a[0]==*a)));
}
```

a == a[0] → 100 == *a → 100 → T
a[0] == *a → *a → T

What is o/p printed?

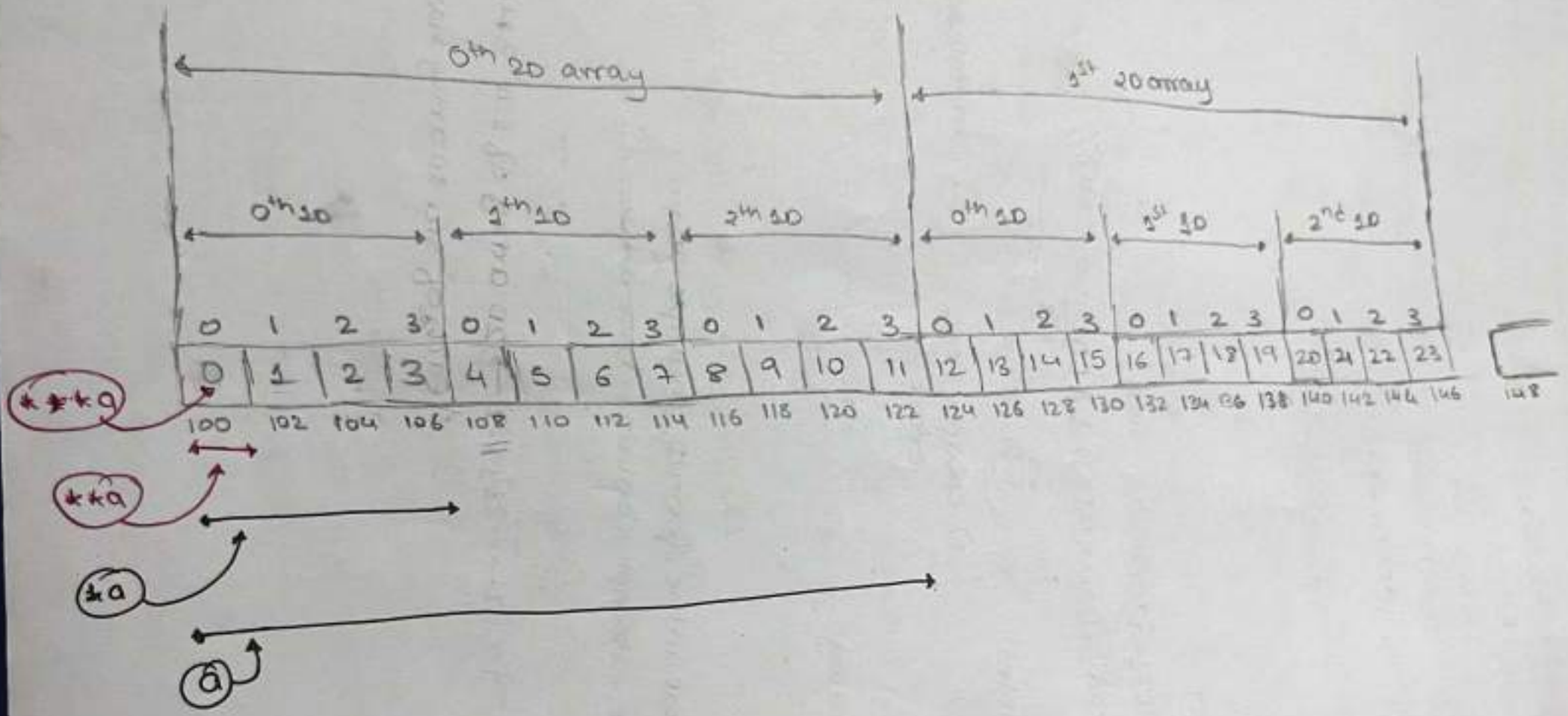
a) 1 b) 0 c) Compile time error d) Run time error.

- Comparing with prior value will comparing

3D array

- 3D array will be store in the form of multiple 2D array
- array name without subscript contains base address of the 0th 2D array

Ex: int a[2][3][4] = {0,1,2.---23}; // 2 → 2D array of size 3*4
Each of size 1D array has 4 element


0th 2D array
1st 2D array
0th 1D
1st 1D
2nd 1D
0th 1D
1st 1D
2nd 1D
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148
***a
**a
*a
a

Expression	Print Value	Type	meaning
1. &a	100	int(*)[][][]	→ B.A of the entire 3D array
2. *&a/a	100	int (*)[][]	→ B.A of 0th 2D array
3. *a	100	int (*)[]	→ B.A of 0th 1D array of 0th 2D array
4. **a	100	int *	→ B.A of 0th element of 0th 1D array of 0th 2D array
5. ***a	0	int	→ value of 0th element of 0th 1D array of 0th 2D array
6. *****a	error	?	→ refering to unknow location
7. &a+1	148	int(*)[][][]	→ B.A of the continous next 3D array
8. a+1	124	int(*)[][]	→ BA of the 1st 2D array
9. *(a+1)	124	int (*)[]	→ B.A of 0th 1D array of 1st 2D array
10. **(a+1)	124	int *	→ B.A of the 0th element of 0th 1D array of 1st 2D array
11. *a+1	108	int (*)[]	→ B.A of the 1st 1D array of 0th 2D array
12. **a+1	102	int *	→ B.A of the 1st element of 0th 1D array of 0th 2D array
13. *(a+1)+1	132	int(*)[]	→ B.A of the 1st 1D array of 1st 2D array
14. *(*(*(a+1))+2)	128	int *	→ B.A of the 2nd element of 0th 1D array of 1st 2D array
15. *(*(a+1)+1)+3	138	int *	→ B.A of the 3rd element of the 1st 1D array of 1st 2D array.
16. *(*a+2)+2	120.	int *	→
17. *****&a	0	int	→
18. *(*(*a+1)+2)			
19. *(*(*(a+1)+2)+1)			

Ex: a[2][2][2]

⇒ *(*(*(a+1)+1)+1)

⇒ *(*(*(100+2)+1)+1)
↳ 0th - 2D array

⇒ *(*(*(124)+1)+1)
↳ 1st - 2D array

⇒ *(*(124+1)+1)
↳ 0th 1D array, 1st 2D array

⇒ *(*(132)+1)
↳ 1st 1D Array, 1st 2D array

⇒ *(132+1)
↳ 0th element, 1st 1D array, 1st 2D array

⇒ *(134)
↳ 1st element, 1st 1D array, 1st 2D array

⇒ 17

Ex: a[1][2][3]

((*(a+1)+2)+3)

⇒ *(*(*(100+1)+2)+3)
↳ 0th - 2D array

⇒ *(*(*(124)+2)+3)
↳ 1st 2D array

⇒ *(*(124+2)+3)
↳ 0th 1D array, 1st 2D array

⇒ *(*(140)+3)
↳ 2nd 1D array, 1st 2D array

⇒ *(140+3)
↳ 0th element of 2nd 1D array of 1st 2D array

⇒ *(146)
↳ 3rd element, 2nd 1D array of 1st 2D array.

Note

① `int (*p)[2][3][4] = &a;` // p is a pointer to a 3D array of size 2x3x4 where every element is integer.

`Pf("%d",p+1);` // 148

② *(&a+1) - a
②

③ **(&a-1) - *a
⑥

④ **(&a+1) - a
24

⇒ 148 - 100

⇒ $\frac{48}{24} = 2$

Ex:

```
void main()
{
  int a[2][3][2] = {{{1,2},{9,8},{23,7}},
                    {{2,24},{21,43},{15,44}}};
  Printf("%d %d %d", (a[1]-a[0]), (a[1][0]-a[0][0]), (a[1][0][0]-
                                                      a[0][0][0]));
}
```

what is o/p printed?

a) 3,3,1
b) 3,6,1
c) 6,6,1
d) 1,1,1

a[1]-a[0]
*(a+1) - *a
⇒ 112 - 100
⇒ $\frac{12}{4} = 3$ //

a[1][0] - a[0][0]
((a+1)+0) - **a
**(a+1) - **a
⇒ 112 - 100
⇒ $\frac{12}{2} = 6$ //

①

H.W

Ex: ① int a[3][4][5] = {0,1,2 ..... 59};

B.A = 3000, SOE = 28

Print all expressions. (16)

② int a[4][5][6] = {0,1, ... 119};

BA = 4000, SOE = 48;

a) pf(*(a+2)+3) = ?

b) pf(*(*(a+3)+2)+1) = ?

c) pf(*(*(a+1)+3)+4) = ?

d) expand a[3][2][3].

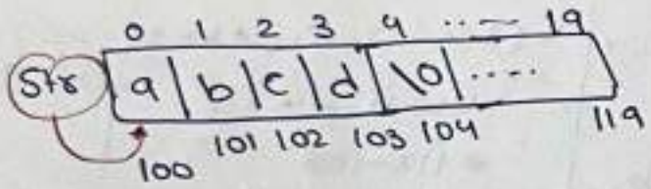
Strings

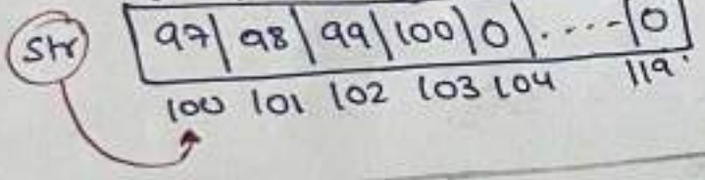
1. Character array is called string.
   String always end with a "NULL" character
2. null character representation is "\0" & ASCII value "0".
3. String name always give Base address.
4. every characters of a string store in the memory sequential in continous location in the form of there ASCII value

Ex: void main()
{
Char str[20] = {'a','b','c','d','\0'}; ← not compulsory
Pf("%c%c%c%c", str[0], str[1], str[2], str[3]);
}
o/p: abcd

str[3]
*(str+1)
*(100+1)
*(101)
98

ASCII
A → 65
a → 97





Pf(str) = 100
Pf(str+1) = 101

① %c → will print character → we should provide ASCII value

② %d → will print ascii value

str[0]
*(str+0)
*str
97

③ If we lead str[0] in any question we need consider ASCII value of the respective character.

②
```
void main()
{
  char str[20] = "abcd";
  printf("%s", str);
}
```
(str) → 100

0	1	2	3	...	19
a	b	c	d	....	
100	101	102	103	...	119

O/P: abcd

for %s we need to provide B.A from the BA provided it will printed all the character till the null character. untill the NULL character

- Initialization of string:

1. `char S[6] = "INDIA";`
2. `char S[] = "INDIA";`
3. `char *p = "INDIA"; // string constant`

```
char S[7] = "INDIA";
Pf("%s", S);
O/P: INDIA@@@...
```

- Explanation

1. Explicitly we need to provide size for null character otherwise it will print some additionally garbage data/value.
2. Implicitly it calculate the size & allocate the memory accordingly
3. the meaning of 3rd statement means we are requesting the compiler to allocate the memory for the string "INDIA" & B.A of the 1st character of the string will be store in the pointer variable P.

- It is consider as string constant & content of string constant can't modified

```
S[2] = 'A';
```

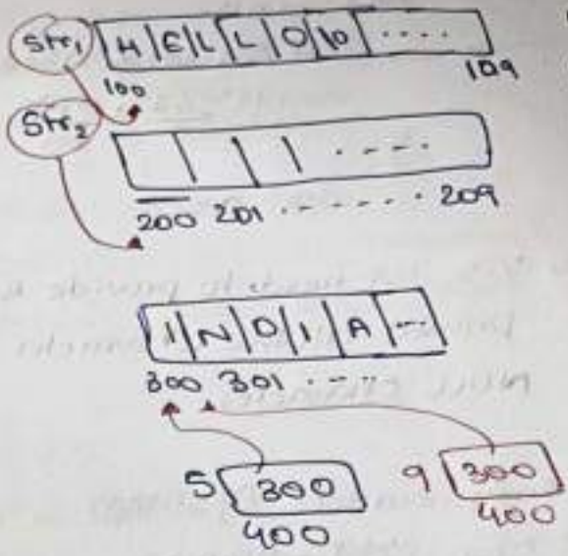
0	1	2	3	4	5
I	N	D	I	A	\0
100	101	102	103	104	105

```
P[2] = 'A';   (X)
```

P → 100

200

① main()
{
char str1[10] = "Hello";
char str2[10];
char *s = "INDIA";
char *q;
str2 = str1; //X
//q = s; we can't modify constant Base address
}



② Main()
{
Char str1[10] = "Hello";
Char str2[10] = "Hello";
if (str1 == str2) // we are comparing B.A of 2 Different String b/c they are different O/p will be "Bye"
Pf ("Hai");
else
Pf ("Bye");
}

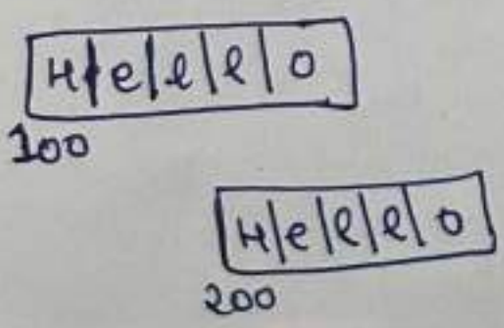
O/p

③ main()
{
if (*"Hello" == *"Hello")
Pf ("Hai");
else Pf ("Bye");
}

- If the string is given in double quots it means BA is given

- O/p : Hai

- we are comparing 1st character of both string b/c they are same so o/p will be Hai
- we are not comparing the remaining character



- General functn available in <String.h>

1. strcpy <T, S>
   - target string B.A
   - source string B.A

2. strupr(s)
   - convert all the character to upper case.

3. strlwr(s)
   - Convert all the character to lowercase

4. strrev(s)
   - reverse the given string

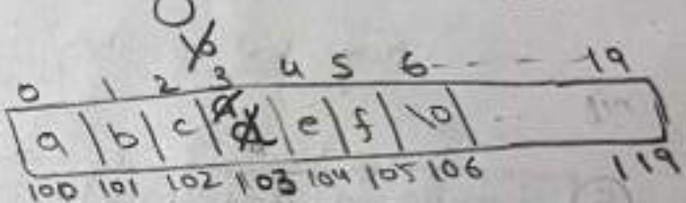
5. Strlen(s)
   - the below functn return the actual length of the string
   - Ex: char *s = "INDIA";
     Strlen(s); // (5) → length
     If you calculate size the (6) → size including null character

Ex: void main()

```
Char str[20]="abcdef";
Pf("%s",str)); // abcdef
Str[3]=97;
Pf("%s",str); abcadef
Str[3]=0;
Pf("%s",str); abc
Pf("%s", *Str); // (97) // error
Pf("%c", Str+3); // 100+3=103
Pf("%c", *(S+3)); // 0
```

Str[3]='0';
Pf("%s",str); // abc0ef
↳ error
Pf("%s", Str+3); // 100+3=103|0ef

S[3]



It was expecting ASCII value but got some address

gate:

Q] char p[20]

```
char *s = "string";
int length = strlen(s); //6
for(int i=0; i<length; i++)
    p[i] = s[length-i];
printf("%s", p);
```

0	1	2	3	4	5	6
s	t	r	i	n	g	\0

p → | \0 | | | |

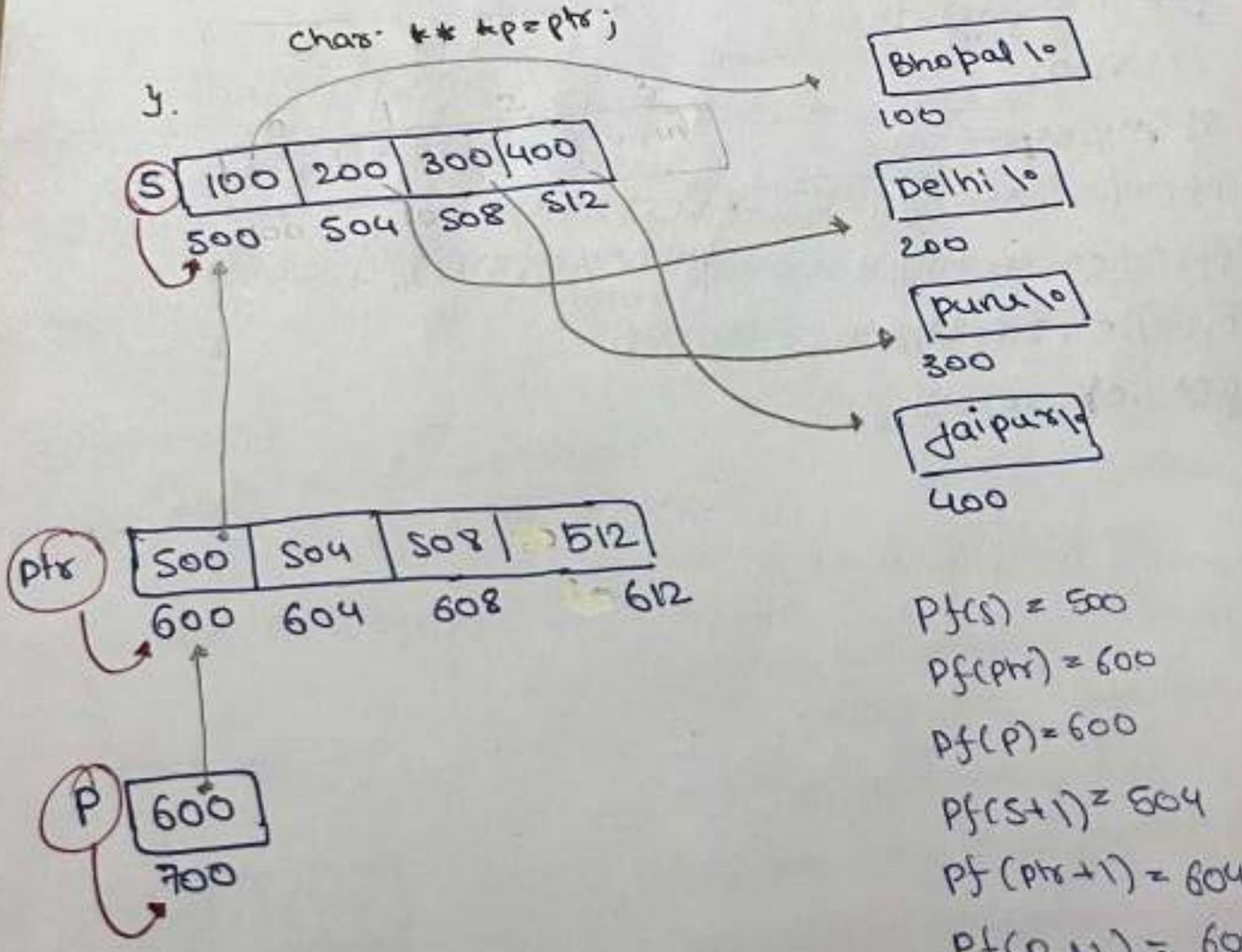
Array of pointer to strings

Ex: void main()

```
char *s[4] = {"Bhopal", "Delhi", "Pune", "Jaipur"};
```

s is a array of 4 element where every element is a character pointer

```
char **ptr[4] = {s, s+1, s+2, s+3};
char ** *p = ptr;
```



Pf(s) = 500

Pf(ptr) = 600

Pf(p) = 600

Pf(s+1) = 504

Pf(ptr+1) = 604

Pf(p+1) = 604

Ex: void main() {

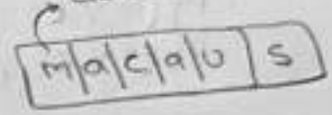
```
char *s[5] = {"London", "Birmingham", "Houston", "milpitas",
              "macau"};

char **ptr[5] = {s, s+1, s+2, s+3, s+4};
char ***p = ptr;
Pf ("%s", ***++p); // Birmingham
Pf ("%s", *--*++p+3); minghom
Pf ("%s", *(p[2])+3); don
Pf ("%s", p[-1][-1]+2); ondon
Pf ("%s"; **p+2); // Birmingham
Pf ("%c", ***++p+2); // O
Pf ("%c", s[1][2]);
```

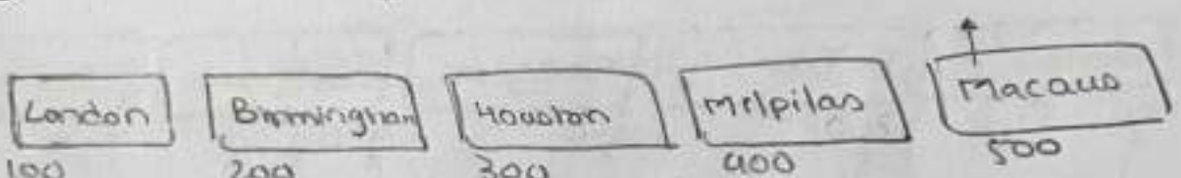
}

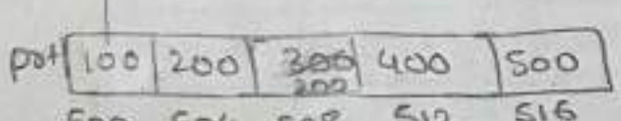
((s+1)+2)

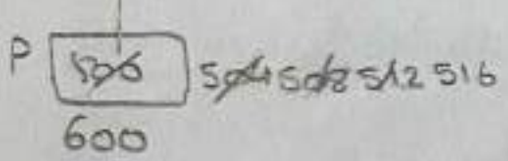
Unary operators have high precedence than Binary

add +2 to get "O"

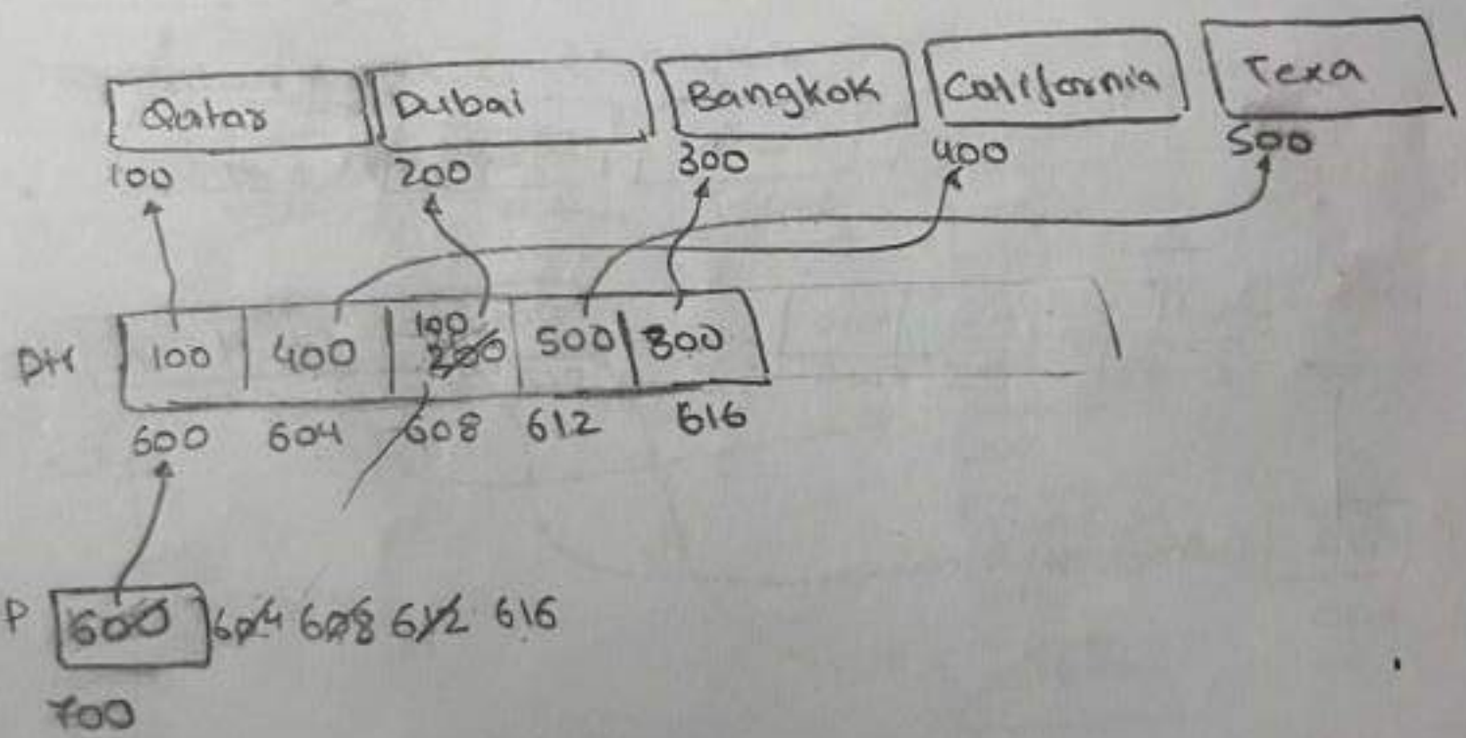






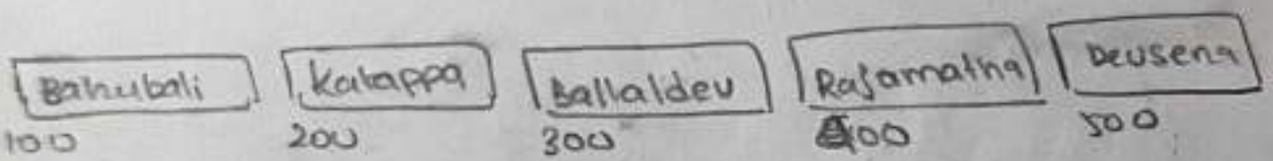


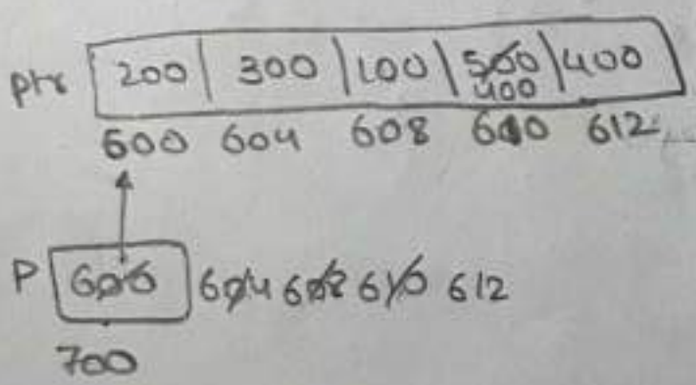
```
void main()
{
  char *s[5] = {"Qatar", "Dubai", "Bangkok", "California", "Texas"};

  char **ptr[5] = {s, s+3, s+1, s+4, s+2};

  char ***p = ptr;

  Pf("%s", **p+1);  // Quatar
  Pf("%s", *--*++p+3);  // ifornia
  Pf("%s", **++p);  // Texa
  Pf("%s", p[-1][-1]+1);  // error
  Pf("%s", *(p[-2])+3);  // iforni a
  Pf("%s", ***++p+2);  // D
  Pf("%c", s[1][2]);  // b
```



Ex: void main()

```
char *S[5]={"Bahubali","Kalappa","Ballaldev","Rajamatha",
                                              "Devsena"};
char **ptr[5] ={S+1,S+2,S,S+4,S+3};
char ***p=ptr;
Pf("%.5s",**p++);//kalappa
Pf("%.5s",*(p[-2])+3);//error or Garbage value
Pf("%.5s",**++p);//bahubali
Pf("%.5s",*--*++p+3);//amatha
Pf("%.5s",P[-1][-1]+2);//error
Pf("%.6s",*S[1][2]);t
Pf("%.6s",***++p+2);T
```





multi dimensional arrays on Strings

Ques] void main()
{
char mess[6][30] = {"Koi bhi Lakshya",
"Bada nahi",
"Jeeta wo ahi",
→ "Jo dara, nahi",
"hara wo ahi",
"Jo Lada nahi"};

Pf("%.s", mess+3); // Jo dara nahi

Pf("%.c", *(*(mess+2)+3)); // t

Pf("%.s", *(mess+4)); // hara wahi

}

H.W

Q. write a program to implement string copy function

```
void main()
{
    char str1[20] = "made easy";
    char str2[20];
    strcopy(str2, str1);
    printf("%.s", str2);
}
void strcopy(char *t, char *s)
{
    while(*t++ = *s++);
}
```

} after performing the last operation (copying the "\0" & then return 0) while returning "0" it get false

Assignement value assign value & return that value

Type casting

- In order to perform any operation blw two variable type of both the variable should be same other wise we need to perform type casting

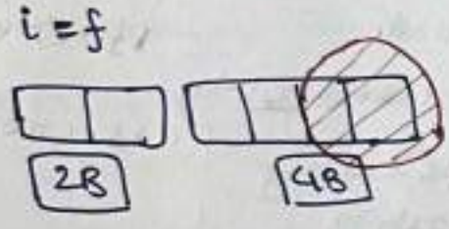
Ex
- int i; → 2B
- char c; → 1B
- float f; → 4B

- Implicit Type Casting: {Coersion}
  - No problem in size
  - problem in type
  - no error
  - type casting required
  - Compiler will do the type casting

F = (float) i;
↳ temporarily treated as float variable

- Explicit Type Casting:

i = f



1) problem in size
2) problem in type
3) error ✓
4) typecasting required
5) user will do type casting
6) no error after type casting
7) data loss will happen

L = (int) f;
↳ treated as int.

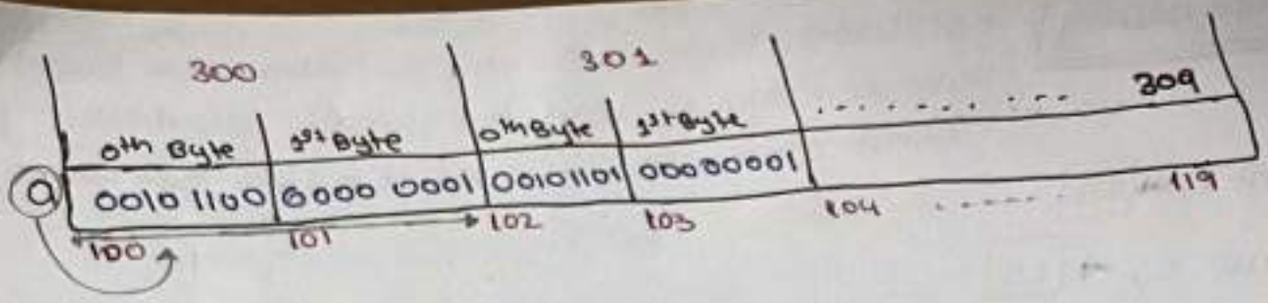
Typecasting in pointers

int a[10] = {300, 301, 302 ..... 309};

Ex: $(300)_{10} = (\underbrace{0000\ 0001}_{H.V}\ \underbrace{00\ 10\ 1100}_{L.V})_2$

we use in our system

Little Endian ⇒ L.V = L.A & H.V = H.A
Big Endian ⇒ L.V = H.A & H.V = L.A



b 100

int *b = a;

4B 4B

- no problem in size
- no problem in type
- no type casting required
- no error

pf(*b) = 2B data

b+1 = 102

It has capacity to access only 1 Byte

c 100

char *c = a;

4B 4B

- no problem in size
- problem in type
- type casting required

char *c = (char*)a;

type casting to Character pointer

- no error after type casting

pf(*c) = 1B data

c+1 = 101

f 100

float *f = a;

4B 4B

- no problem size
- problem in type
- error will come
- type casting required

float *f = (float*)a;

type casting floating pointer

- no error after type casting

pf(*f) = 4B data

f+1 = 104

e 100

void *e = a;

4B 4B

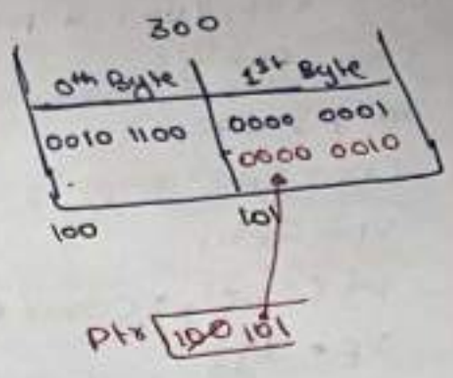
- no problem in size
- no problem in type
- no error
- no type casting required

pf(*e) = error

e+1 = error

w.B
page no.
128
023

```
{
int i=300;
Char *ptr=(char*)&i;
*++ptr=2;
printf("%d",i);
}
```



$i = (0000\ 0010\ 0010\ 1100)_2$

$\Downarrow$

$(556)_{10}$

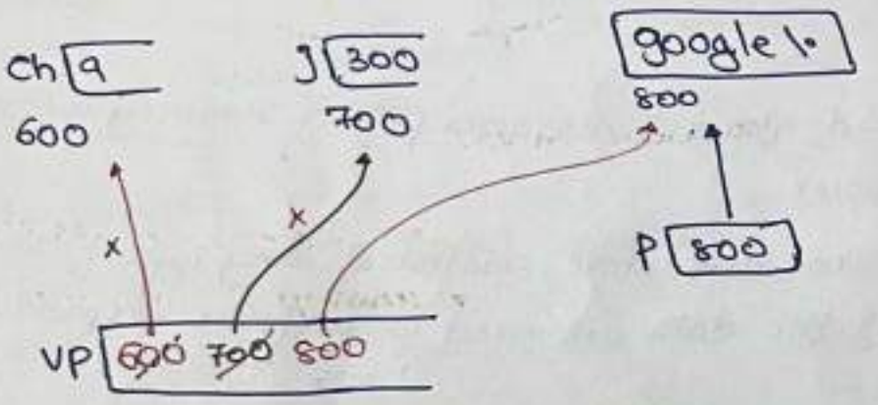
Void pointer

- just like we had integer & character pointer similarly we can also have void pointer.
- in the void pointer we can store address of any type of variable but will accessing the data we need to perform respective type casting A/C
- type casting is required b/c void pointer doesn't know how many Byte of data it has to dereference
- void pointer are used in device driver programming.

note

- the main purpose of type casting is that it will clarify that how many Byte of data it has to dereference

```
void main()
{
  void *vp; // vp is a pointer variable of type void where we can store address of any type
  char ch='g';
  int j=300;
  char *p="google";
  vp=&ch;
  pf("%c",*(char *)vp); // g
  vp=&j;
  pf("%d",*(int *)vp); // 300
  vp=p;
  pf("%s",(char *)vp+3); // 800+3 → 803 → gle
}
```



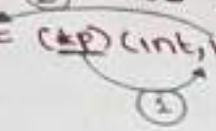
Pointer to function

```
void test();
void main()
{
  1. void (*p)();
  2. p=test;
  3. (p)();
}
void test()
{
  pf("Hello");
}
```

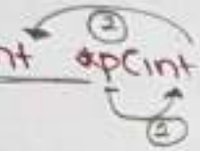
1. these is known as declaring functn pointer, P is a pointer to a functn which doesn't take any arguments and return void
2. assigning the B.A of the test functn to the pointer variable P
3. calling the test functn by using pointer variable P

- Terminalogy for various expression:

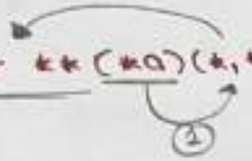
1. `int (*p)(int,int);` // p is a pointer to the functn which takes two integer argument & return integer



2. `int *p(int *);` // p is a function which take integer pointer as a argument & return integer pointer

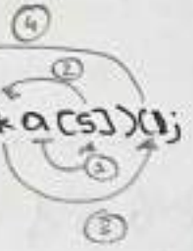


3. `char **(*a)(*,*);` // a is a pointer to a function which takes two pointer as a argument & return character double pointer.

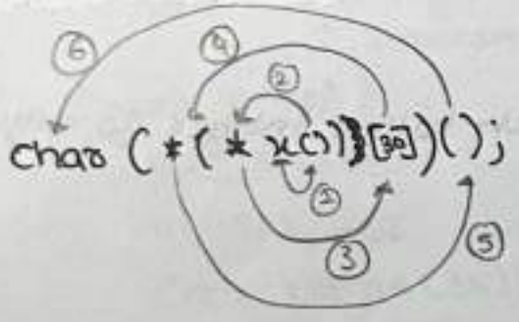


Ex:

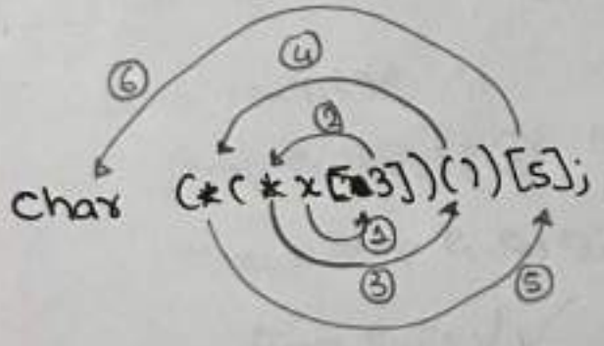
4. `int (*a[5])();` // a is an array of 5 element & every element is pointer to a function which don't take any argument & return integer.



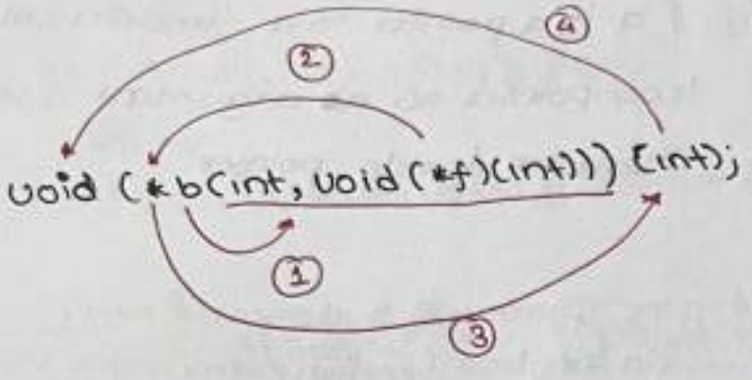
5. `char (*(*x())[30])();` // x an is funct^n which doesn't take any argument to an pointer of an array of 30 element & every element is pointer to the function which doesn't take any argument & return char



6. `char (*(*x[3])())[5];` x is an array of 3 element, where each element is a pointer to a function which doesn't take any argument & return pointer to an array of 5 element which return characters.



Ex: void (*b(int, void (*f)(int)))(int);



•B is an functn which is taking two argument a, b & return pointer, & those pointer is pointing to function which is taking integer as argument & return void

a: a is an integer

b: f is an pointer to functn which takes integer as argument & return void.

Structure

- Structure is used to create user define datatype.
- In the structure to operators are used.
  - dot (.) : Structure variable to member variable
  - Arrow ("→") : pointer variable to member variable

Note

- Structure contains member variable
- size of the structure is sum of all members variable
- we can't define a structure without any members variable
- we can define a structure inside the funct^n.
- we can't define the funct^n inside the structure
- we can create existing struct as member to other struct
- we can't create same structure variable as members to the Structure b/c logical size is undefine
- we can create same structure pointer as member to the structure if that is called as self referencial structure (used in linked list).

Ex:

```
Keyword        name of Structure
   ↑                ↗
Struct emp{
    int id;→2B
    int sal;→2B
    char name[36];→36B      } 40B  | member variables.
};
typedef Struct emp Emp;

void main(){
    emp e1,e2,e3;// ✗  Struct keyword is missing
    Struct emp e1,e2,e3;// ✓  these are called Structure variable
    Emp e4,e5;// ✓   Emp will be replace by Struct.emp.
```

$e_1.id = 1429$ // ✓

$e_1.sal = 2500$ // ✓

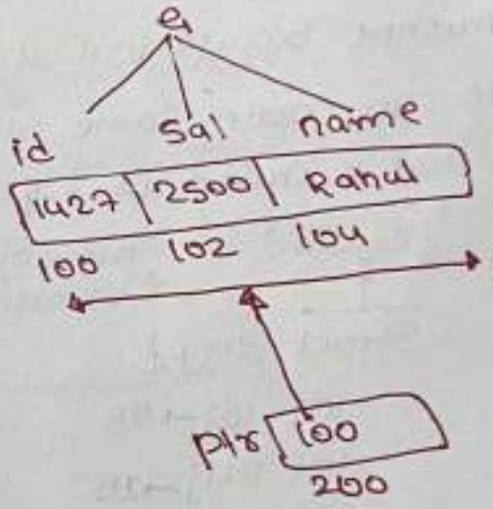
$e_1.name = Rahul$ // ✗ not allowed since we can't modified constant Base address.

$e_1.name = "Rahul"$ // ✗ ——— // ——— same reason

Strcpy($e_1.name$, "Rahul"); // ✓

Emp *ptr; // ✓ ptr is a pointer variable of type struct Emp wherewe can store address of any struct variable

ptr = &e_1 // ✓



Pf(ptr); // 100 ✓

Pf(e_1); // ✗

Pf($e_1.sal$); // ✓ 2500

Pf(*ptr) // ✗

Pf((*ptr).sal); // ✓ 2500

Pf((ptr[0]).sal); // ✓ 2500

Pf(ptr → sal); // ✓ 2500

all are same

$e_2 = e_1$; // the variable operation is allowed but not suggestiable to perform practically.

$e_1 * e_2$; // ✱

$e_1 + e_2$; // ✗

}

H.W

Difference b/w typedef & #define

① Struct node {
    int data; → 2B
    Struct node n1; // ④
};
refer to point no. ⑦

② Struct node {
    int data;
    Struct data *n1; // ⑤
refer to point no. 8.

[As we are creating pointers we are clearly define that pointer take 4B i.e used in Linked list]

③ void main() {
    Struct emp {
        int id;
        float b;
    };
    mec();
}

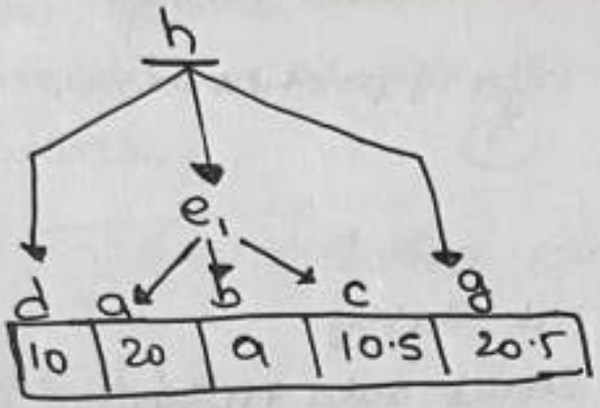
void mec();
    Struct emp e1, e2, e3; // ⑥

- Structure is defined inside the main & we are creating Structure variable outside the main.
- If we want to create structure variable outside the main() then we need to define the structure outside the main.

Eg. Struct S1 {
    int a;
    char b;
    float c;
};
Struct S2 {
    int d;
    Struct S1 e;
    float g;
};

Struct S_2 h = {10, {20, 'a', 10.5}, 20.5}

d a b c g



pf(h.d) = 10

pf(h.e1.a) = 20

Ex: Struct test?

```
    int i;
    Char *c;
};
Struct test st[5] = {5, "become", 4, "better", 6, "jungle", 8, "ancestor",
                     7, "brother"};
```

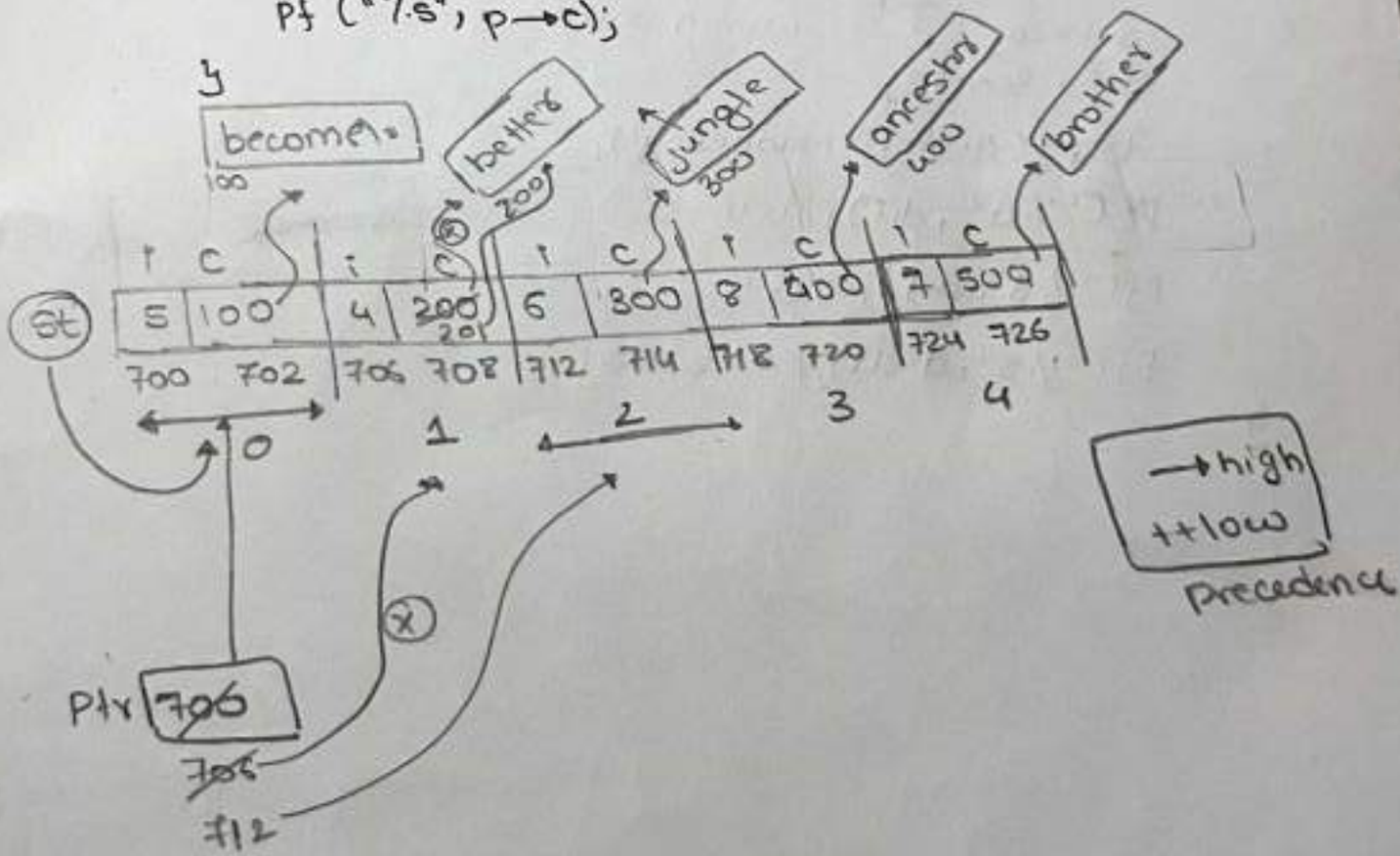
ptr → 706 = 4508

```
void main()?
    Struct test *p = st;
        p++;
        ++p->c;
        pf("%s", p++->c);//etter
        pf("%c", *++p->c);//u
        pf("%d", p[0].i);//6
        pf("%s", p->c);
```

(b)

o/p:

etter, u, 6, ungle.



Unions

- the size of the union will be the max size of the member variable

- In the structures we can access all the member any time but in case of union we can only access one member variable & all the remaining member variable will contain garbage value.
- In the union same memory space will be shared by all the member variable

Eg: union data { → name of union

Keyword

```
int i;  → 2B
char str[20]; → 20B
float f; → 4B
```

} 20B

};

```
main(){
    union data d; // 20B
                  → union variable
    d.i = 20;
    d.f = 300.5;
    Strcpy(d.str, "madeeasy");
    Pf("%d", d.i); // G.V
    Pf("%f", d.f); // G.V
    Pf("%s", d.str); // made easy
}
```

d — 20MB

~~20~~ ~~300.5~~ madeeasy

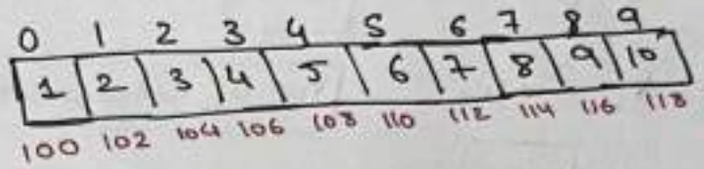
Array

- array is used to store homogenous/similarly type of element
- all the element will be store in continous location
- array support random access

- 1D Array

int a[0 ..... 9] = {1,2,3,4,5,6,7,8,9,10}

lower bound — upper bound



$Loc(a[8]) = 100 + (8-0)*2$ → S.O.E, → lower bound

$\Rightarrow 100 + 16$

$\Rightarrow 116$

$Loc(a[4]) = 100 + (4-0)*2$ → S.O.E, → L.B

108

Q1) a[-636 ..... +817], B.A = 4700, SOE = 9

$Loc[a[799]) = 4700 + (799 - (-636)) * 9$

$\Rightarrow 4700 + 1435 * 9$

$\Rightarrow 17615$

2) a[-843 ..... +916]

B.A = 6800

S.O.E = 8

$6800 + (847 - (-843)) * 8$

$\Rightarrow 6800 + 1690 * 8$

$\Rightarrow 6800 + 13520$

$\Rightarrow 20320$

note • array index always start with 0. otherwise we need to perform one additional subtraction operation

a[lb.....ub], B.A, SOE = 'S'

loc(a[i]) = B.A + (i - lb) * S

2D array

a[1.....4][1.....5]

lb_1 ub_1 lb_2 ub_2

Rows Columns

no. of rows (n-r) = $ub_1 - lb_1 + 1$ 4-1+1=4

no of columns (n-c) = $ub_2 - lb_2 + 1$ 5-1+1=5

• 2D array will be store in the form of 1D array & it is store in.

2 Different way

→ row-major order

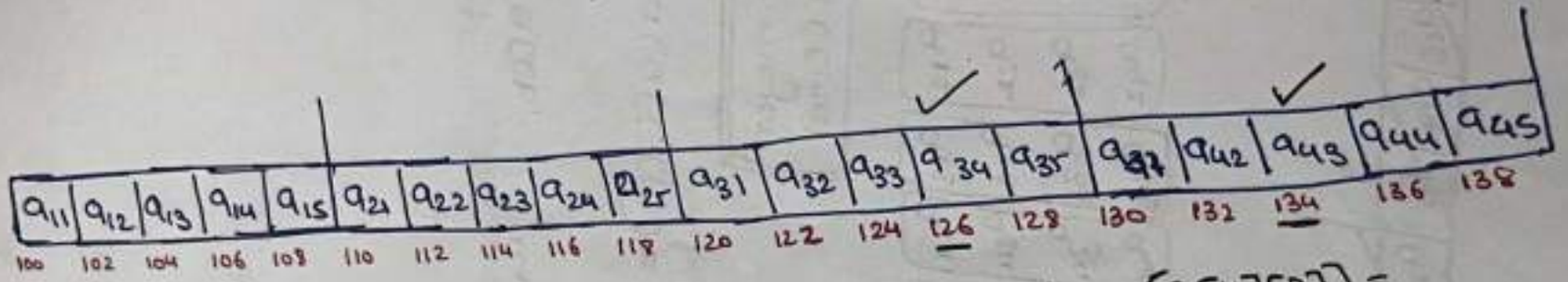
→ column-major order

• C language By default follow row major order

Row major order (R.M.O):

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

4×5



Ex: $loc(a[3][4]) = 100 + [(3-1)*5 + (4-1)]*2$ → 5·0·6

lb_1 → n·c → lb_2

$\Rightarrow 100 + [10+3] \times 2$

$\Rightarrow 126$

Ex. $a_{43} = loca(a[4][3]) =$

$\Rightarrow 100 + [(4-1)*5 + (3-1)]*2$

$\Rightarrow 100 + 34$

$\Rightarrow 134$

Ex: $a[-157 \ldots 968][-468 \ldots 819]$

B·A = 7200

S·O·S 7

① $loca(a[839][727])$

$\Rightarrow 7200 + [(839-(-157))*1288 + (727-(-468))]*7$

$7200 + [(996*1288) + 1195]*7$

$7200 + 12,82,848 + 1195$

Q) a[-699 ..... 713] [-543 ..... 547]. BA = 6500, SOE = 6

loca(a[691][547])

→ 6500 + [(691 - (-699)) * 1141 + (547 - (-543))] * 6.

→ 6500 + [1390 * 1141 + 1090] * 6

→ 6500 + 1587080 * 6

→ 9528980

note:

a[lb1 ..... ub1][lb2 ..... ub2], B.A, SOE = S, RMO

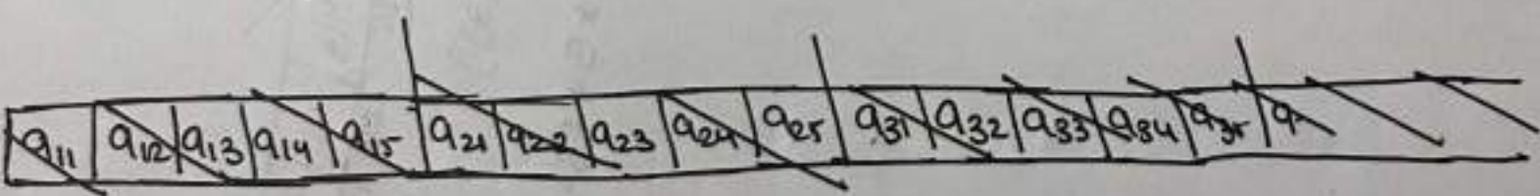
loca(a[i][j]) = BA + [(i - lb1) * (ub2 - lb2 + 1) + (j - lb2)] * S

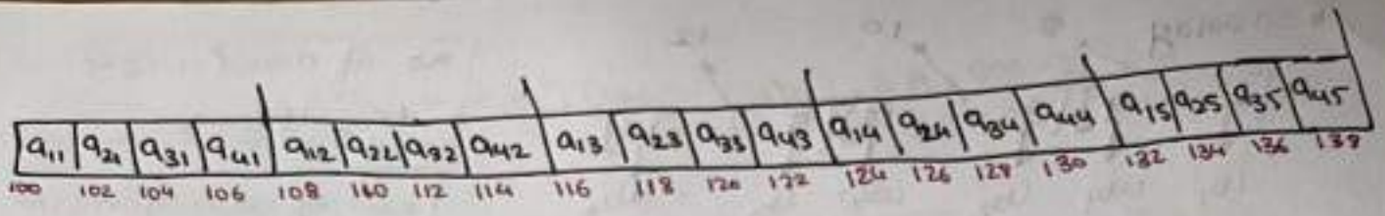
Colum major order (CMO)

a[lb1 ..... ub1][lb2 ..... ub2], B.A, SOE = S, CMO

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

loca(a[i][j]) = BA + [(j - lb2) * (ub1 - lb1 + 1) + (i - lb1)] * S





$$\text{Loca}(a[3][5]) = 100 + [(5-1)*4 + (3-1)]*2$$

(lb_2, n·r, lb_1)

$$\Rightarrow 100 + [16+2]*2$$

$$\Rightarrow 136$$

Ex: a[-611 ---- +711][-811 ---- +911].

BA = 7850, SOE = 9

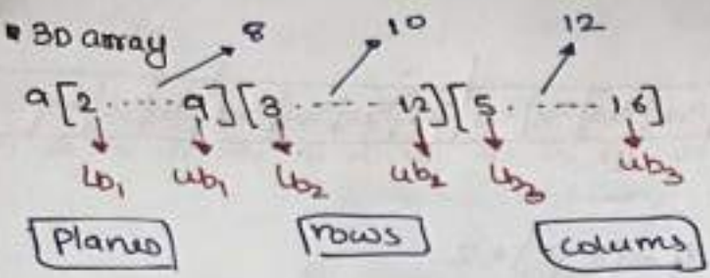
$$\text{Loca}[a[633][844]] = 7850 + [(844-(-811))*1323 + (633-(-611))]*9$$

$$\Rightarrow 7850 + [1655*1323 + 1244]*9$$

$$\Rightarrow 7850 + [2189565 + 1244]*9$$

$$\Rightarrow 19725131$$

• 3D array



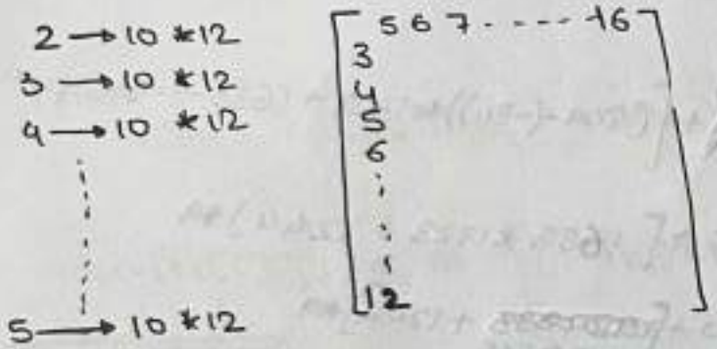
no. of rows (n.r) = $ub_2 - lb_2 + 1$

no. of column (n.c) = $ub_3 - lb_3 + 1$

R.M.O

• 8 ⇒ 2D arrays
↳ each of size = 10 * 12

• 10 ⟶ 1D arrays
↳ each array has 12 elements



Ex:

$$loc(a[5][7][9]) = 100 + [(5-2)\,10*12 + (7-3)*12 + (9-5)]*2$$

lb_1 → nr. · nr → nc. · lb_2 → nc. · lb_3 → S.O.E

$$\Rightarrow 100 + [30*12 + 4*12 + 4]*2$$

• 924

$$loc(a[7][11][14] = 100 + [(7-2)\,10*12 + (11-3)*12 + (14-5)]*2$$

$$\Rightarrow 100 + [50*12 + 8*12 + 9]*2$$

⇒ 1510

Note

a[lb1 ... ub1][lb2 ... ub2][lb3 ... ub3], B.A, size = s

R.M.O:

$$Loc(a[i][j][k]) : BA + [(i-lb_1)*(ub_2-lb_1+1)*(ub_3-lb_3+1) + (j-lb_2)*(ub_3-lb_3+1) + (k-lb_3)]*s$$

(n·s, n·c, n·c)

C.M.O:

$$Loc(a[i][j][k]) : BA + [(i-lb_1)*n \cdot r*nc + (k-lb_3)*n \cdot r + (j-lb_2)]*s$$

Lower triangular matrix (L.T.M)

- Lower triangular matrix is possible only for square matrix.

Ex: a[1 ..... 4][1 ..... 4]

lb_1 ub_1 lb_1 ub_1

i<j → 0

i≥j → stored (or) non-zero.

R.M.O.

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_{4\times4}$$

∴ no of non-zero element $= \frac{n*(n-1)}{2}$

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}
100	102	104	106	108	110	112	114	116	118

loc(a[4][3]) =

$$100 + \left[\frac{(4-1)(4-1+1)}{2} + (3-1) \right] * 2$$

(n, ub_1; ub_1)

$= 116.$

i.e loc(a[3][2]) ⇒ $i \geq j$

$$100 + \left[\frac{(3-1)(3-1+1)}{2} + (2-1) \right] * 2$$

(ub_1)

$$\Rightarrow 100 + \left[\frac{2 \times 3}{2} + 1 \right] * 2$$

$$\Rightarrow 108.$$

Ex: a[137 ···· 961][137 ···· 961], B.A = 4900, SO.E = 7.

$$\text{loc}(a[893][745]) = 4900 + \left[\frac{(893-137)(893-137+1)}{2} + (745-137) \right] * 7$$

$$\Rightarrow 4900 + \left[\frac{756 \times 757}{2} + 608 \right] * 7$$

$$\Rightarrow 4900 + [378 * 757 + 608] * 7$$

$$\Rightarrow 4900 + [286114] * 7$$

$$\Rightarrow 2012178$$

note: a[lb$_1$ ... ub$_1$][lb$_1$ ... ub$_2$] ; B.A, SOE = $; R.M.O, L.T.M

$$loc(a[i][j]) = B.A + \left[\frac{(i-lb_1)(i-lb_1+1)}{2} + (j-lb_1)\right] * \$$$

$i \geq j$

C.M.O :-

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_{4\times4}$$

a_{11}	a_{21}	a_{31}	a_{41}	a_{22}	a_{32}	a_{42}	a_{33}	a_{43}	a_{44}
100	102	104	106	108	110	112	114	116	118

$$loc(a[4][2]) = 100 + \left[\frac{(4-1+1)*(4-1+1+1)}{2} - \frac{(4-2+1)(4-2+1+1)}{2} + (3-2)\right] * 2$$

(lb_1, lb_1, ubi, lb_1 | ub_1, j, ub_1, j | i, j)

$$\Rightarrow 100 + \left[\frac{4*5}{2} - \frac{3*4}{2} + \right] * 2 \Rightarrow 100 + (10 - 6 + 1) * 2$$

$$\Rightarrow 110$$

$$loc(a[4][3]) = 100 + \left[\frac{(4-1+1)*(4-1+1+1)}{2} - \frac{(4-3+1)*(4-3+1+1)}{2} + (4-3) \right] * 2$$

(ub, lb, ub, lb; ub, j, ub, j; i, j)

$$\Rightarrow 116.$$

Ex: $a[117....489][117....489]$ $BA = 8800$, $SOE = 9$.

$loca[a[473][389]]$

$$\Rightarrow 8800 + \left[\frac{(489-117+1)(489-117+1+1)}{2} - \frac{(489-389+1)(489-389+1+1)}{2} + (473-389) \right] * 9$$

$$\Rightarrow 8800 + \left[\frac{373*374}{2} - \frac{101*102}{2} + 84 \right] * 9$$

69751

$$\Rightarrow 590956$$

note : a[lb1 ... ub1][lb1 ... ub1], BA, SO ∈ S . LT.M, C.MO.

Loc(a[i][j]) = BA + [(ub1 - lb1+1)(ub1 - lb1+1+1) / 2

- (ub1 - j + 1)(ub1 - j + 1 + 1) / 2

- (i - j)] * S

i ≥ j

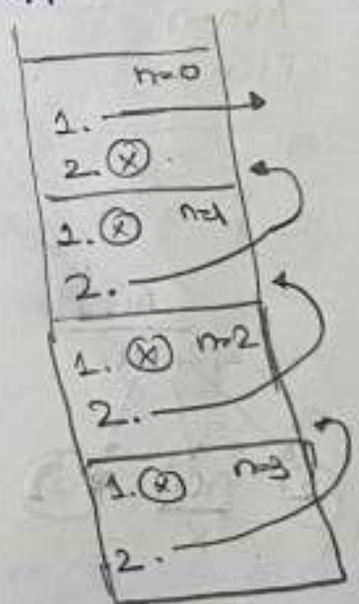
Recursion • Recursion are categorized into 4 type

• Tail Rec. : In the program very last statement is only one recursive call.

Ex:
```
TR(int n){
  if(n==0) return;
  else{
    Pf(n);
    return TR(n-1);
  }
}
```

what is the o/p of TR(3)?

o/p: 3 2 1



Note

i) Break; → }// come out of bracket

2) return → function

3) exit(0) → program, successful exit

4) exit(1) → program, unsuccessful exit

5) exit() → program, not informing anything

- In tail Recursion we are mis.s using concept of recursion b/c it is wasting a lot of stack space
- the advantage is that we can easyily write equivalent non-Recursion program By using for loop & while loop

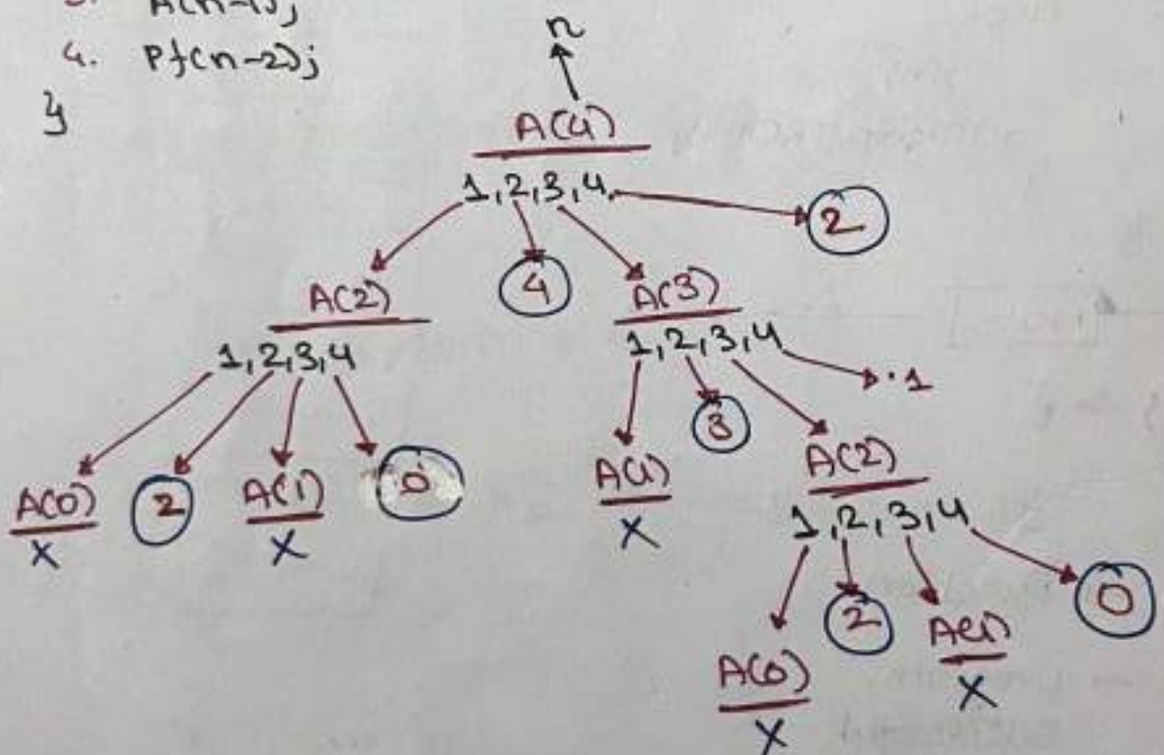
non-Tail Recursion

- in the program after the recursive call there are some statement to execute. then it is called as non-tail Recursion

Ex① A(int n){
if(n<1)return;
else{
1. A(n-2);
2. Pf(n);
3. A(n-1);
4. Pf(n-2);
}
}

what is o/p of A(4)?

o/p: 2,0,4,3,2,0,1,2.



Q] A(int n) {

if (n ≤ 1) return;

else {

1. Pf(n-2)
2. A(n-1);
3. Pf(n-1)
4. A(n-2);
5. Pf(n);
6. A(n-3);
7. Pf(n-2);

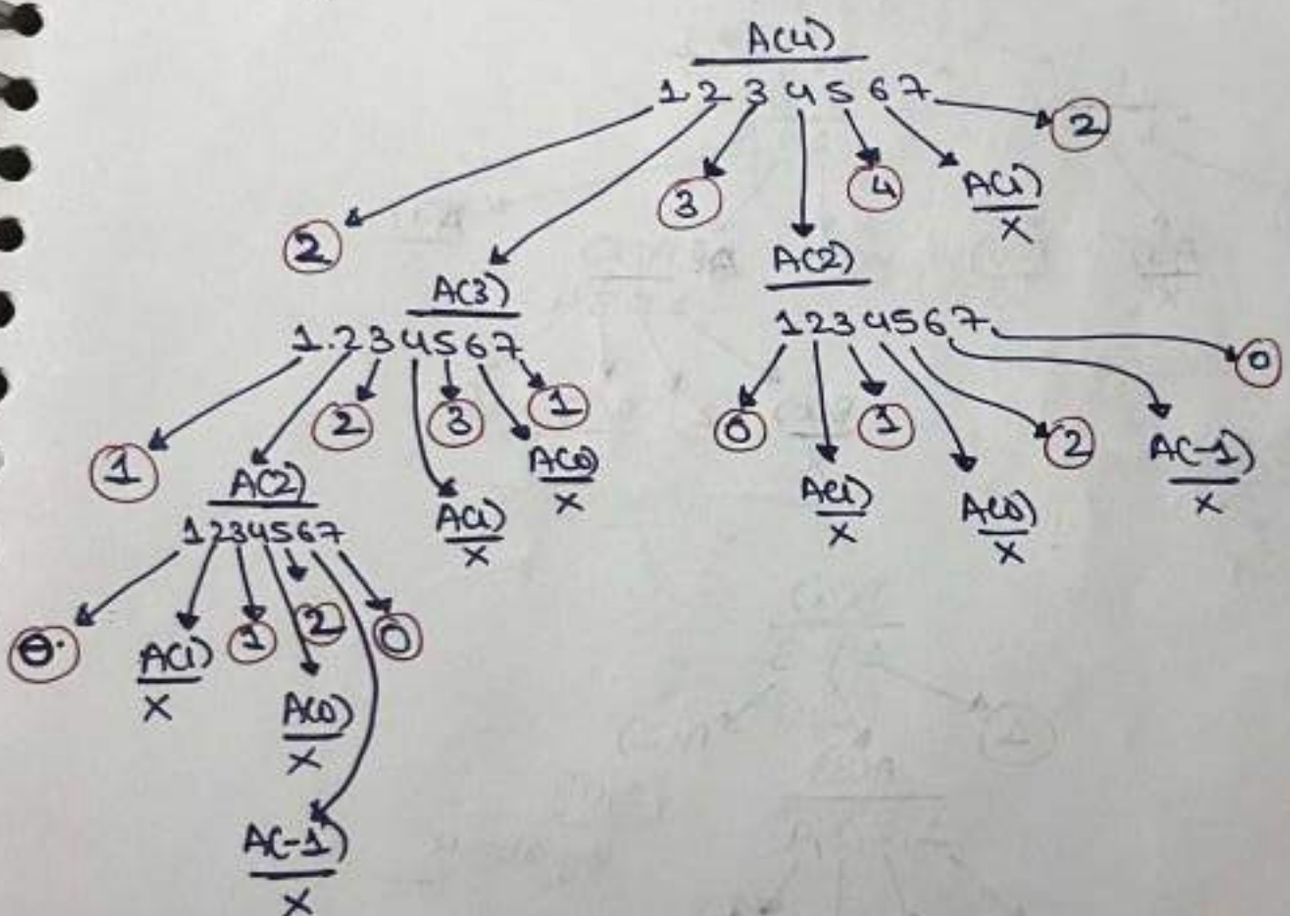
}

}

O/p: 2,1,0,1,2,0,2,3,1,3,0,1,2,0,4,2

Note

- in non tail Rec. writting an equivalent non-Recursive program is possible but difficult.



Indirect Recursion

- Two or more function calling each other is called indirect recursion.

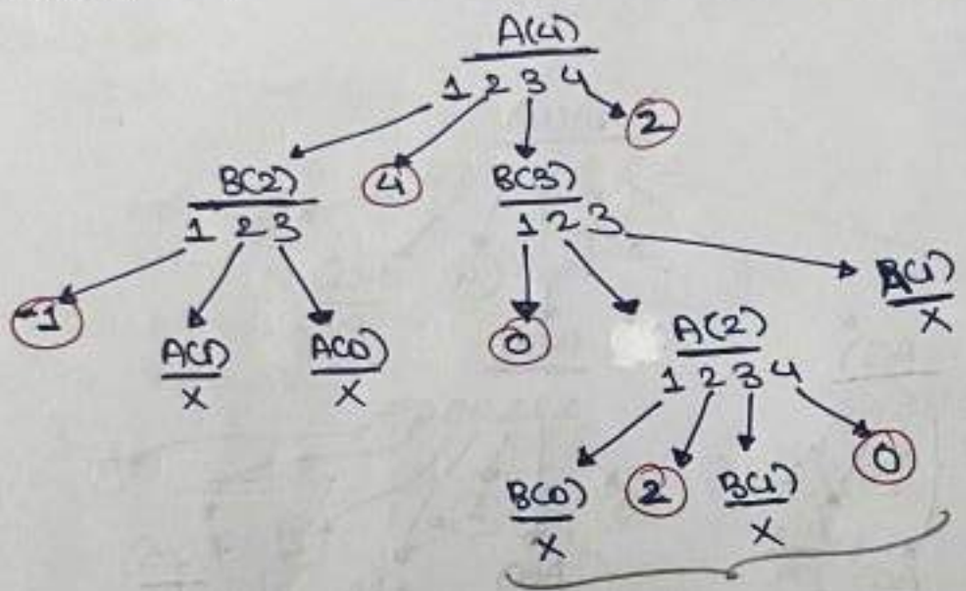
```
A(int n){
    if (n<=1) return;
    else{
    1. B(n-2);
    2. Pf(n);
    3. B(n-1);
    4. Pf(n-2);
    }
}
```

```
B(int n){
    if(n<=1) return;
    else{
    1. Pf(n-3);
    2. A(n-1);
    3. A(n-2);
    }
}
```

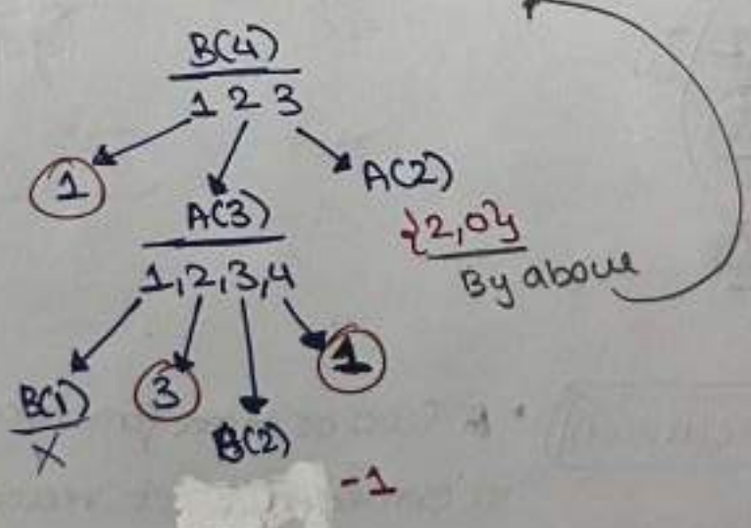
O/p: -1, 4, 0, 2, 0, 2

what is o/p of A(4)?

→



→



O/p: 1, 3, -1, 1, 2, 0

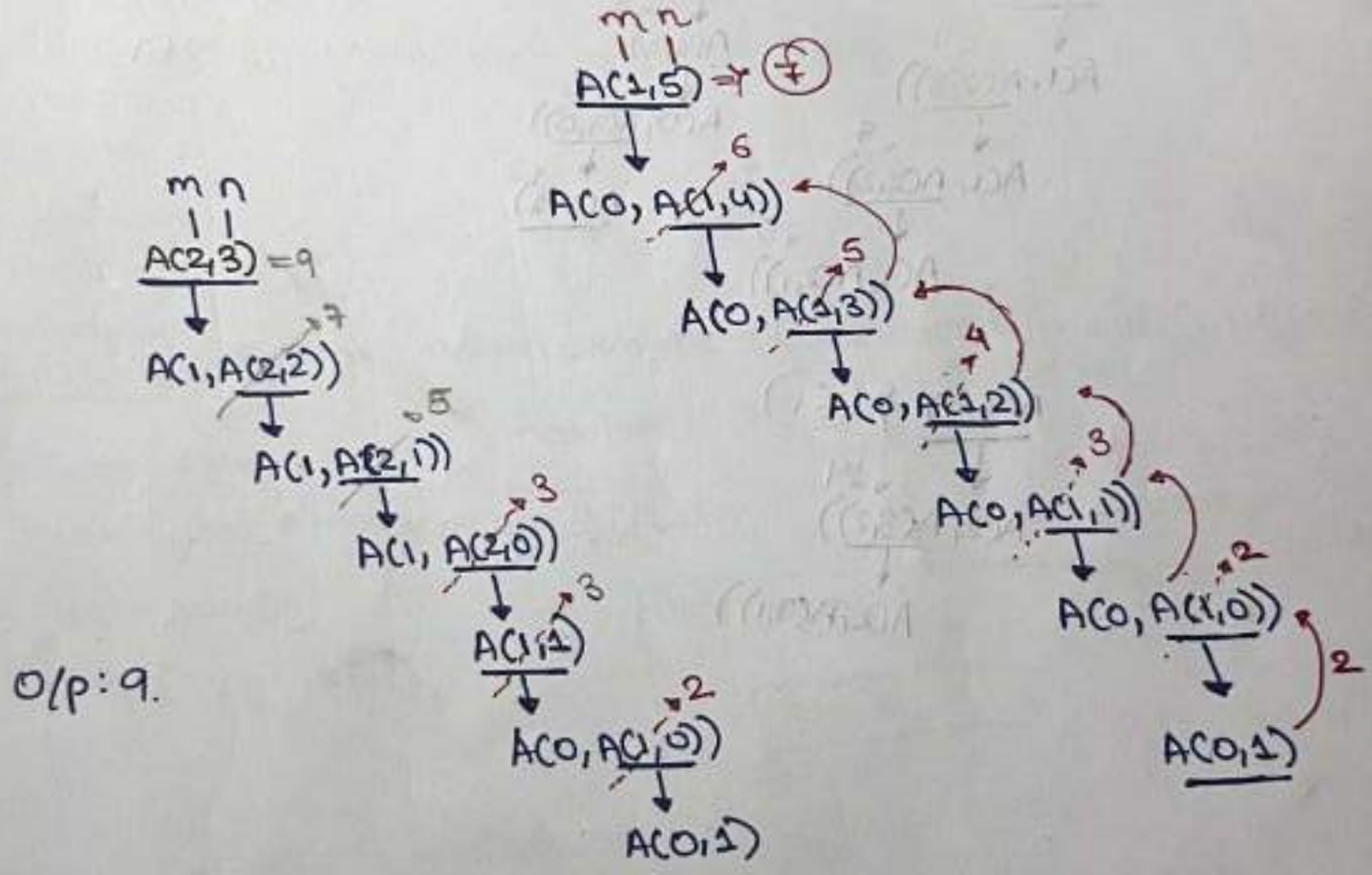
Nested Recursion

(Caller mains function).

- A recursion function which is passing itself as a parameter then it is called as nested recursion.

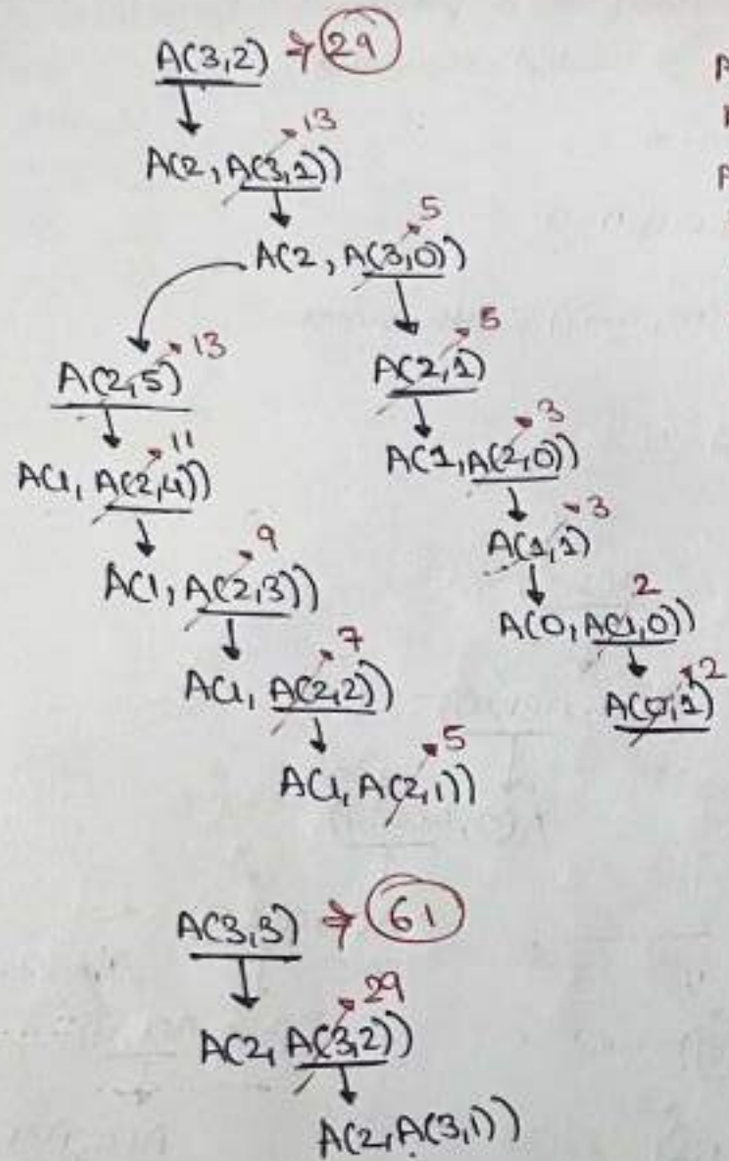
$$A(m,n) = \begin{cases} n+1, & \text{if } m=0 \\ A(m-1,1), & \text{if } n=0 \\ A(m-1, A(m, n-1)), & \text{otherwise} \end{cases}$$

what is o/p of A(1,5)=? (7)



o/p: 9.

H.W ① A(3,2) = 29 ② A(3,3) = 61



A(2,1) = 5
A(2,3) = 9
A(2,5) = 13
A(2,7) = 17
A(2,9) = 21
A(2,11) = 25
A(2,13) = 29
A(2,29) ⇒ 2×29+3
⇒ 58+3
⇒ 61

- Dynamic memory allocation (D·M·A)
- array follow static memory allocation which means that we should the size of array prior to the declaration.
- If the requirement is Dynamical allocating the memory at runtime, then we need to use the concept of DMA.
- By using the concept of DMA we can allocate the memory whenever we want & however we want at runtime
- these will allocated at the heap segment.
- the lifetime of Dynamic Memory (DMA) is entire program.

→ Following funct^n are used in the DMA

1. malloc();
2. calloc();
3. realloc();
4. free();

Malloc() Syntax: Pointer_variable = (type-cast *) malloc (sizeof(size));

not funct^n. it's just an operator

Ex: float *ptr;

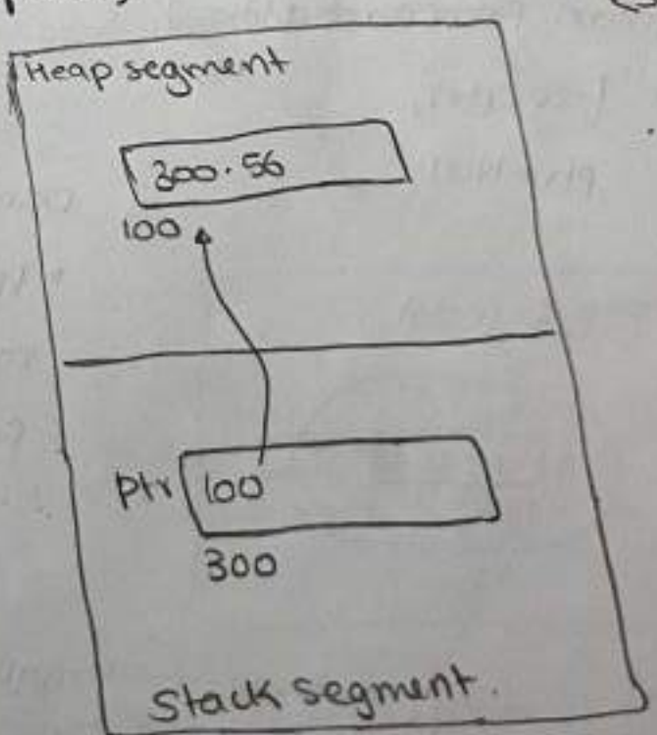
ptr = (float *) malloc (sizeof(float));

*ptr = 300.56;

printf ("%f", *ptr);

//malloc use
#include <stdlib.h>

//malloc only take Byte as an argument.



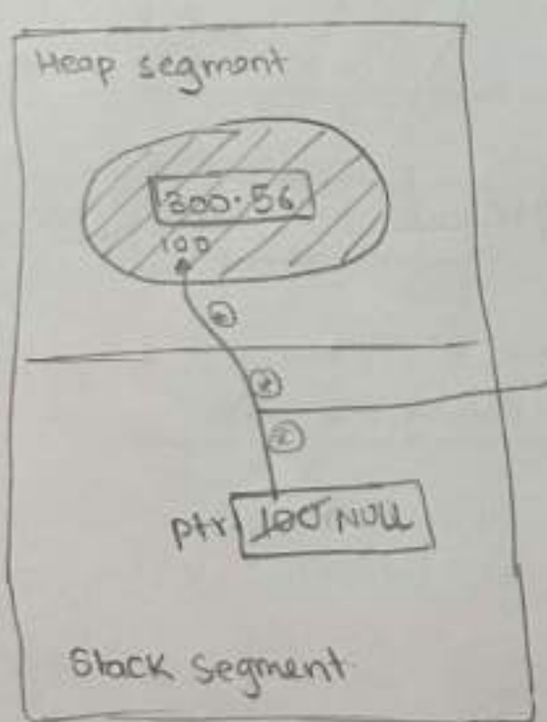
- malloc Dynamical allocate memory in the heap segment & it take one argument that is in Byte
- Sizeof is an operator and it calculate the size in Byte
- malloc allocate the memory in heap segment & return its Base address & these Base address to an integer pointer.
- B/c it is returning B.A receiver should be an pointer, b/c it is returning it is integer pointer then respectively type casting is required to access the required no. of Bytes.
- whatever the memory allocation in the stack segment can be accessed using variable names the memory is allocated in the Heap segment can only be accessed through pointer
- malloc return NULL if it is not able to allocate memory if memory is full

Free()

- Free() deallocates the memory in heap segment & it takes one argument that is B.A.

Syntax: Free(Base address);

Ex: free(ptr);
ptr = NULL;



Dangling pointer:

- the pointer pointing to deallocated memory (unknown location) is called as dangling pointer

after ptr = NULL;

dangling ptr is removed too.

- After completion of the usage of dynamically allocated memory it must be delocated otherwise it is called "memory leakage".

Data Structure

- Data Structure is a way of organize the data element by considering its relationship
- Data Structure mainly deal with
  1. how efficiently data can be organise & store in the computer memory
  2. how efficiently data can be retrieves & manipuated

Program = Algorithm + Data Structure

Data Structure = organized Data + operation

- Data Structure is divided into 2 type
  - Linear
    - Arrays
    - Linked List
    - Stack
    - Queue
  - non-Linear
    - graphs
    - tree's (B-ST, AVL)

- Abstract Data type (ADT): In order to simplify the process of solving the problem data structure combined along with a operation is called as "ADT".

- ADT mainly deal with
  - → Declaration of Data
  - → Declaration of operation

Linked List

- its an Data Structure used to store data element, the successive element are connected by pointer & last element will contain "null"
- we can increase the size of Linked list dynamically untill the memory is full
- if the requirement is frequently inserting & deleting at any's position then Linked list is best suitable data structure
- the main Drawback of Linked list is that it only allow Sequential access.

TC = SLL

	B.C	W.C
Insert	O(1)	O(n)
Delection	O(1)	O(n)

Construction of Linked list:

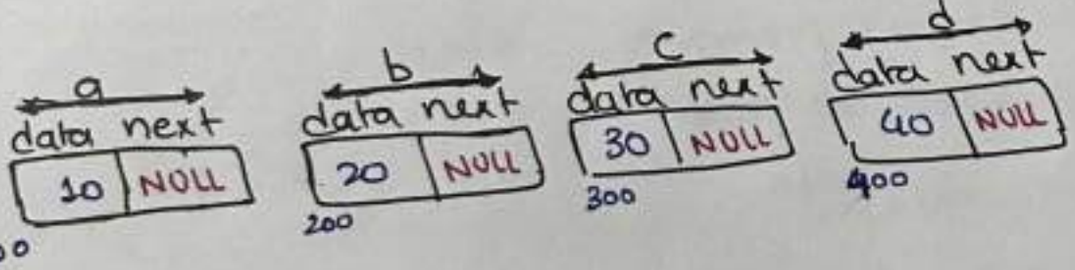
① Struct node{
    int data; //2B
    Struct node * next; //4B
};
typedef struct node node;

② node a = {10, null};
node b = {20, null};
node c = {30, null};
node d = {40, null};

→ Decalaration & Initialization

a,b,c,d are Structure variable

- in a S.L.L a pointer is given, point to some node. so to delete that node, what is the TC.

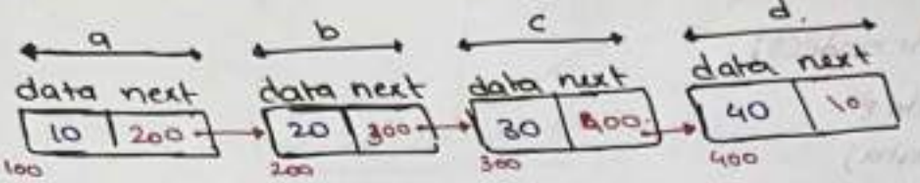


note

- In the S.L.L last node next pointer is always NULL
- In the S.L.L we can traverse only in one direction.

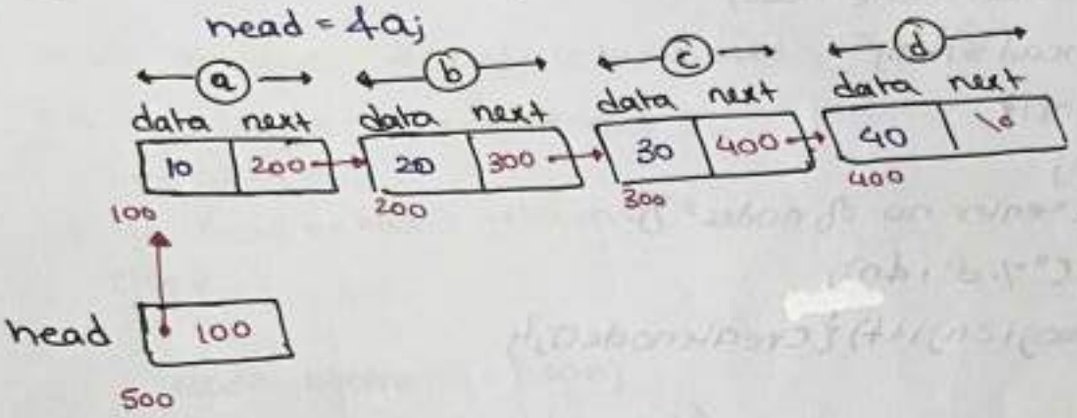
Note: TC : O(n)

③ a.next = &b;
b.next = &c;
c.next = &d;



④ node *head = NULL;

head = &a;



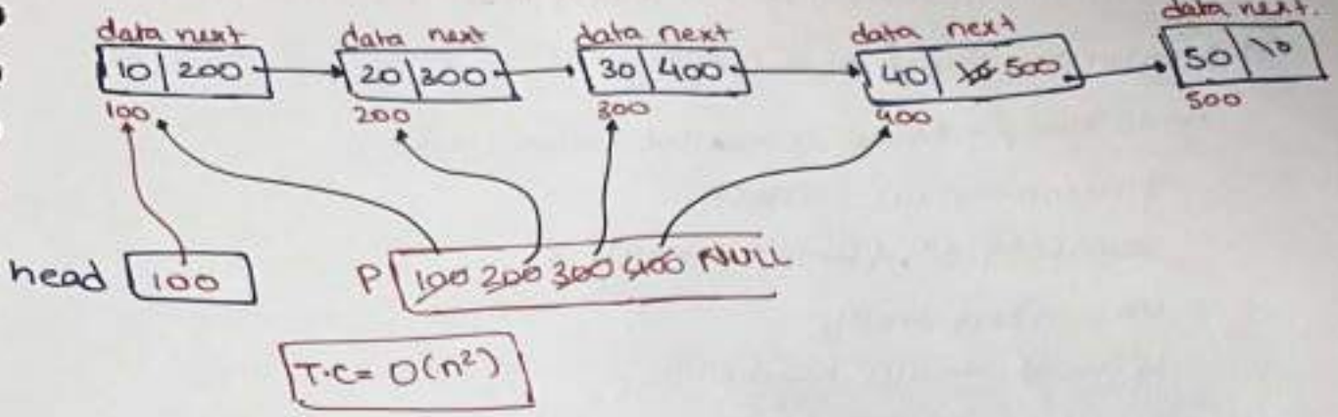
Note

1. pf(head) = 100
2. pf(head→data) // 10
3). if(head==NULL)
// if above condition is true then linked list is empty
4) pf(head→next); //200
5) pf(head→next==null) // if above condition is true then L.L have only one element.
6). pf(head→next→next→next→data);
// 40

- Creation of linked list is done by self referential structure variable
- If the requirement is frequent insertion & deletion in various position then linked list very much useful compare to of any array.

- Write a program to create L.L Dynamical with n nodes.

```
#include <studio.h>
# Include <studl lb.h>
void createnode();
Struct node {
    int data;
    Struct node *next;
};
typedef Struct node node;
node *head = null;
Void main(){
   int i,n;
   printf("enter no. of nodes");
   scanf("%d", &n);
   for(i=0;i<n;i++){createnode();}
}
void Createnode(){
   node *temp = (node*) malloc (sizeof(node));
   if (temp== NULL) exit(1);
   printf ("enter data element");
   Scanf("%d", &temp→ data);
   temp→next = NULL;
   if (head == NULL)
      head = temp;
   else {
      node *p = head;
      while(p→ next!=NULL) {
          p=p→ next;
      }
      p→next = temp;
      p= NULL; // optional
   }
   temp = NULL; // optional
}
```



T.C = $O(n^2)$

W.A.P to count #f node in the given L.L. T.C: $\Theta(n)$

```
int ListLength(node *head){
    int count=0;
    if(*head==NULL) return 0;
    else{
        node *temp= head;
        while(temp->next!=null){
            count++;
            temp=temp->next;
        }
        Return count;
    }
```

WAP to insert new node at the begin of L.L T.C: $O(1)$

```
InsertAtBegin(node *head){
    node *temp=(node *)malloc(sizeof(node));
    // if(temp==NULL) exit(1); // if not written then will cause.
    Scanf("%d",&temp->data);
    temp->next=NULL; // Segmentation error (or) NULL pointer dereferencing
    (null->next)
    if(head==null) head=temp;
    else{
        temp->next=head;
        head=temp;
    }
}
```

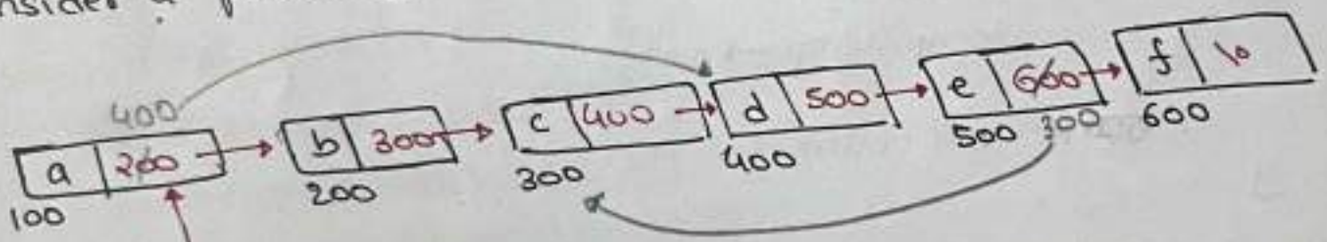
Creating node

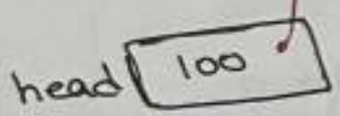
1st connect external link then break internal link

W.A.P insert the new node at the end of given L.L T.C. O(n)

```
void insertAtEnd (node *head)
{
    node *temp = (node *) malloc (sizeof(node));
    if (temp==null)  exit(1);
    scanf("%d", &temp->data);
    temp->next = null;
    if (head == null) head = temp;
    else {
        node *curr = head;
        while (curr->next != null){
            curr = curr->next;
        }
        curr->next = temp;
    }
}
```

Q] consider a following LL with given statement.

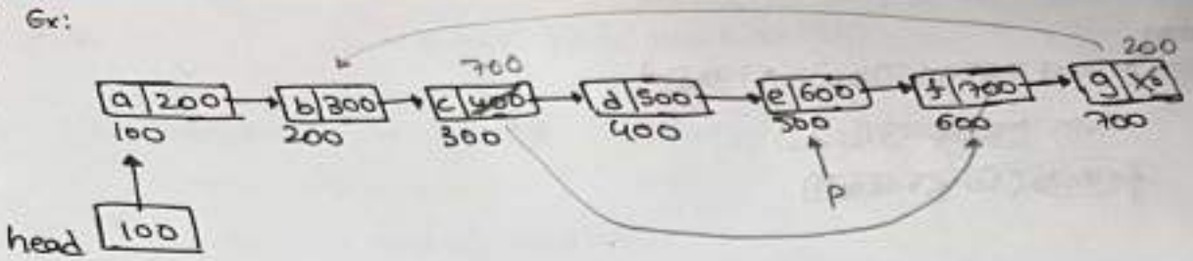





```
Struct node *p;
p = head->next->next;//300
p->next->next->next = p
head->next = p->next;
pf("%c", head->next->next->next->next->data);
```


O/p : d//

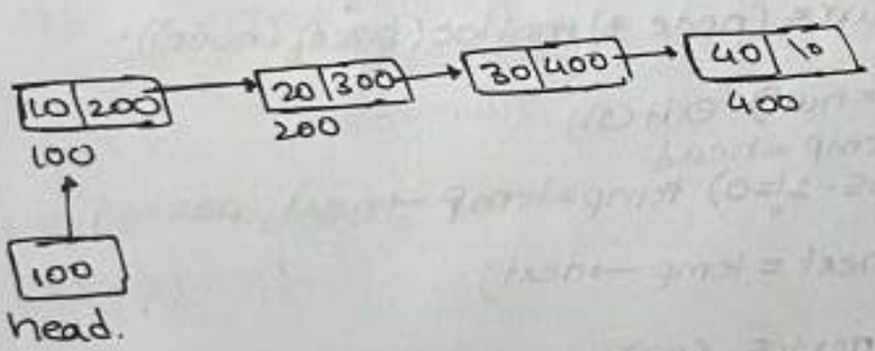
Ex:




```
Struct node *p;
p=head->next->next->next->next;
head->next->next->next=p->next->next;
p->next->next->next=head->next
pf("%c", p->next->next->next->next->next->next->next->data);
```

y O/P: C

Ex: WAP to insert the new node at the given position




```
int ListLength(node *head){
    int count=0; node *temp=head;
    while(temp==null){
        count++;
        temp=temp->next;
    }
    return count;
}
```

```
insertAtpos( node *head , int pos){
    int length = listlength(head);
    if (pos<=0 || pos > length +1){
        Pf("invalid position");
        return;
    }
    if (pos == 1){insertAt Begin(head); return;}
    if (pos == length +1){
        insertAtEnd(head);
        return;
    }
    else{
        node *curr = (node *) malloc(sizeof(node));
        if(curr == null) exit(1);
        node *temp = head
        while(pos-1!=0) temp=temp->next; pos--;
        Curr->next = temp->next;
        temp->next = curr;
    }
}
```

Scanf("%d", &Curr->data);

Q. WAP to delete the last node. TC O(n)

```
deleteAtEnd (node *head){
    node *p=head;
    int len = ListLength(head);

    if(p==null) return;
    if(p->next==null){ free(p);
        head = null;
    }
    while (len-1!=0){p=p->next; len--;}
    free (p->next);
    p->next=null;
}
```

Q] WAP to delete the 1st node.

```
deleteAtBegin (node *head){
if (head==null) return;
if( head->next==null){
        free(head);
        head=null;

} node *p=head;
p=p->next;

free(head);
head=p;
}
```

Q1 WAP to delete element of a given position. TC: O(n)

```
deleteAtpos (node * head, int pos){  int len = listlength(head);
    if( pos<=0 || pos>length) { Pf("Invalid"); return;}
    if(head==null) return;
    if (head→next==null) {
        free(head); head=null;
    }
    if(pos==len( ) { deleteAtEnd(head); return;}
    if( pos==1){deleteAtBegin(head); return;}
    else{ node *p=head;
        while(pos-1!=0) {
            p=p→next;
            pos--;
        }
                    node *temp= p→next
        }
        p→next=p→next→next;
        free(temp);
    }
}
```

Q. WAP to reverse a given LL. T.C : O(n)

```
node *reverse(node *head){

    if(head==null) return;
    if(head->next==null) return;
    node *curr=head,
         *prev=null;
    while(curr->next!=null || curr==null){   node *temp=
        curr->next=                          curr->next;
        prev=curr;
        curr=temp;
    }
    return prev;
}
```

Q] WAP to find a wheather a cycle exist or doesn't exist.

```
LL cyclepresentornot(node *head){

    if(head==null || head->next==null)
        pf("no cycle");
    node *slow=head, *fast=head->next;
    while(slow!=fast){
        slow=slow->next;
        fast=fast->next->next;
    }
    if(slow==fast)
        pf("cycle is present");
    else pf("no cycle present");
}
```

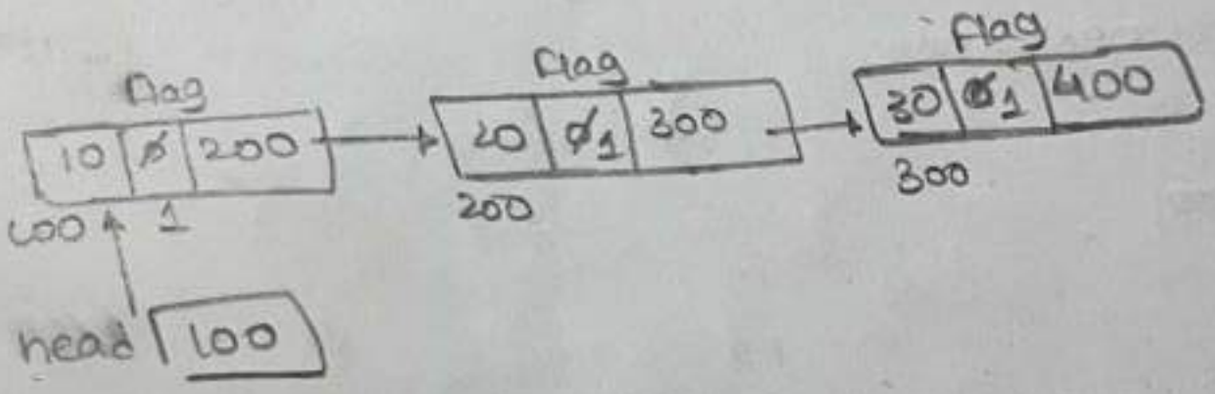
Solution : 2

[Changing the struct]

```
Struct node{
        int data;
        int flag;
        Struct node *next;
};
```

Inevery node initialize flag value '0'

while creating L.L



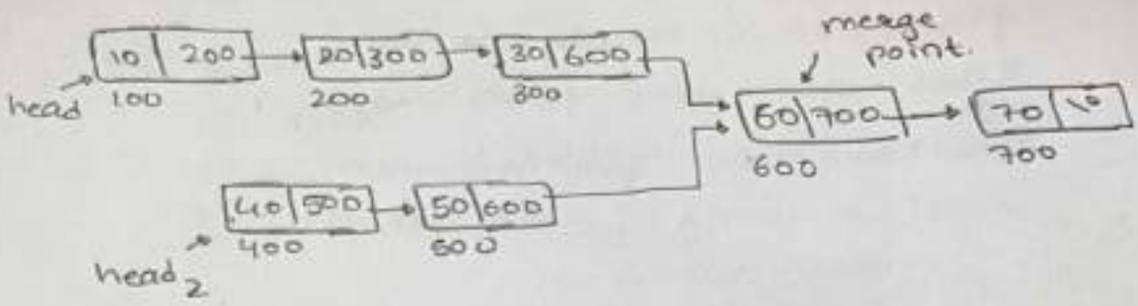
Q/ WAP to find middle of LinkedList

```
node* middleLL(node *head){
        if(head==null) return null;
        if(head->next==null) return head;
        node *slow=head, *fast=head->next;
        while(fast!=null && fast->next!=null){
            slow=slow->next;
            fast=fast->next->next;
        }
        return slow;
}
```

Q/ WAP to find middle of the LL using listlength().

```
node* middleLL(node *head){
        int mid=listlength()/2;
        if(head==0) return null;
        if(len==1) return head;
        if(head==null) return null;
        if(head->next==null) return head;
        int mid=listlength()/2; node *temp=head;
        while(mid--!=0){
            temp=temp->next;
        }
        return temp;
}.
```

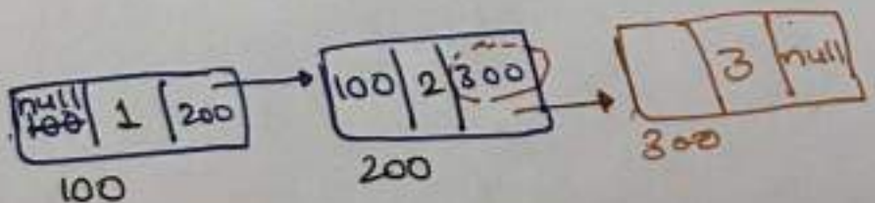
Q] WAP to find mergepoint in give two L.L :- T.C O(m+n) or O(len1+len2)



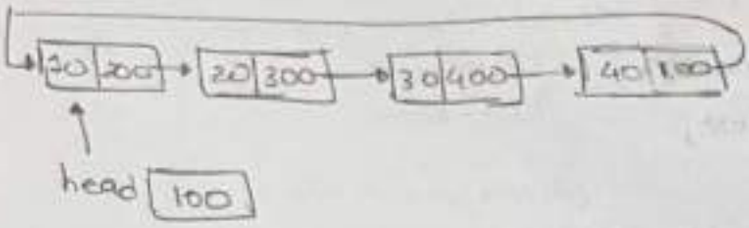

```
mergepoint (node *head1 , node *head2)?
    if( head1==null || head2==null) return.
    if(head1→next==null && head2→next==null) return
    int len1=listlength(head1); len2=listlength(head2); diff=0;
    if(len1>len2) diff=l1-l2
    else diff=l2-l1
    while( len1>len2 && diff!=0) head1=head1→next; diff--;
    while( len2>len && diff-- !=0) head2=head2→next;
    while(head1!=head2)?
        head1=head1→next;
        head2=head2→next;
    }
    return head1;
}
```

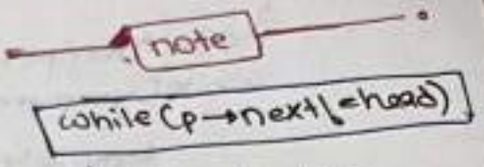
- In a doubly linked list organization, insertion of record in end involve modification of one pointer for existing list.

ex:

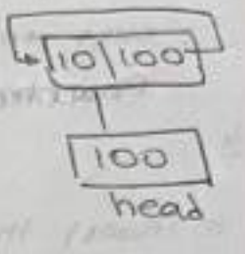


- **Circular single linkedlist (C.L.L or S.L.L):**
- In CSLL, last node next pointer will point to head node.





↳ termination condition



- WAP to find length of a C.L.L

```
Int ListLength CSLL(node *head){
    if(head == null) return 0;
    if(head->next == head) return 1;
    int len = 1; node *temp = head;
    while(temp->next != head){
        temp = temp->next;
        len++;
    }
    return len;
}
```

- WAP to insert new node at the begin of the C.L.L.

```
insert atBeginCLL(node *head){
    node *curr = (node *) malloc(sizeof(node));
    if(curr == null) exit(1);
    scanf("%d", curr->data);
    curr->next = null;
    if(head == null){
        head = curr;
        curr->next = head;
    }
}
```

```
if(head->next == head){
        head->next=curr;
        curr->next = head;
}
node *temp=head;
while(temp->next!=head){
        temp = temp->next;
}
temp->next = curr;
curr->next = head;
temp=null;
}
```

- WAP to insert the new node at the end of the C.L.L.

```
InsertAtendCLL(node *head){
        node *curr=(node *)malloc(sizeof(node));
        if(curr!=null) exit(0);
        scanf("%d", curr->data);
        curr->next=null;
        curr->prev=null;
        if(head==null){head=curr;
                curr->next=head;
        }node *temp=head;
        while(temp->next!=null){
                temp=temp->next;
        }
        temp->next=curr;
        curr->next=head;
        temp=null;
}
```

• WAP to delete the last node of the C.L.L.

```
deleteAtEnd(node *head){
    if(head == null) return;
    if(head->next == head){
        free(head);
        head = null;
    }
    node *temp = head;
    while(temp->next->next != head){
        temp = temp->next;
    }
    Free(temp->next);
    temp->next = head;
    temp = null;
}
```

• WAP to delete the 1st node of the C.L.L.

```
DeleteAtBeginCLL(node *head){
    if(head == null) return
    if(head->next == null){
        head->next = null;
        Free(head);
        head = null;
    }
    node *curr = head; node *temp = head
    curr->prev->next = curr->next;
    curr->prev = null;
    head = curr->next;
    curr->next = null;
    free(curr);
}
```

```
while(temp->next != head){
    temp = temp->next;
}
temp->next = curr->next;
head = curr->next
curr->next = null;
free(curr);
}
```

there code Section is for Circular Double Linked List.

Note T.C: C.S.L.L

	Begin	End
Insert	O(n)	O(n)
Delete	O(n)	O(n)

- **Double Linked list (D.L.L)**

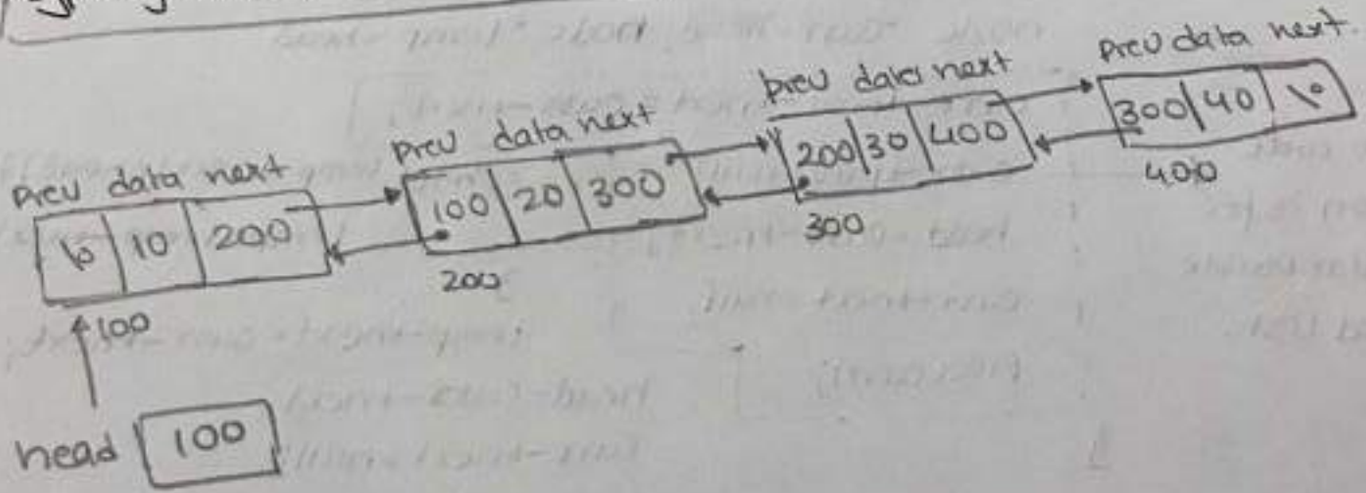
T.C: D.L.L

	Begin	End
Insert	O(1)	O(n)
Delete	O(1)	O(n)

- in double L.L ever node contains 2 pointers pointing to previous node & next node
- we can traverse in both direction,
- insertion & deletion will take more amount of time b/c of more pointer updation.
- the main application D.L.L is, it is used in tree programing

```
Struct DLLnode{
    Struct DLLnode *prev;
    int data;
    Struct DLLnode *next;
};
```

```
typedef Struct DLLnode node;
```



- WAP to insert new node at the end of D.L.L

```
insertAtEndDLL (node *head)?

    node *curr = (node *) malloc(sizeof(node));
    if(curr==null) exit(1);
    scanf("%d", curr->data);
    curr->prev = null;
    curr->next = null;
    if(head==null){
        head=curr
    }
    if(head->next==null){
        head->next=curr;
        curr->prev=head
    }
    node *temp=head;
    while(temp->next!=null){
        temp=temp->next;
    }
    temp->next=curr;
    curr->prev=temp;
    temp=null;
}
```

- WAP to insert a node at the beglaning of the D.L.L

```
Insertthenode ATBegin (node *head)){
    node *curr= (node *) malloc (sizeof(node))
    if(curr=null) exit(1);
    scanf("%d", curr->data);
    curr->next=null;
    curr->prev=null;
    if(head==null) return;
    if (head->next!=null)?
        free(head);
        head=null;
}
node *curr=head;
head=curr->next
curr->next=null;
free(curr);
}
```

- WAP to delete 1st node of D.L.L.

```
deleteAtBeginDLL(node *head)
    if(head==null) return
    if(head->next==null){free(head); head=null;}
    node * temp=head; *temp_next=temp->next;
     while(temp_next->next!=null){
       temp = temp->next
       temp_next = temp_next->next;
     }
     temp->next=null; temp_next->prev=null;
     Free(temp_next);
}
```

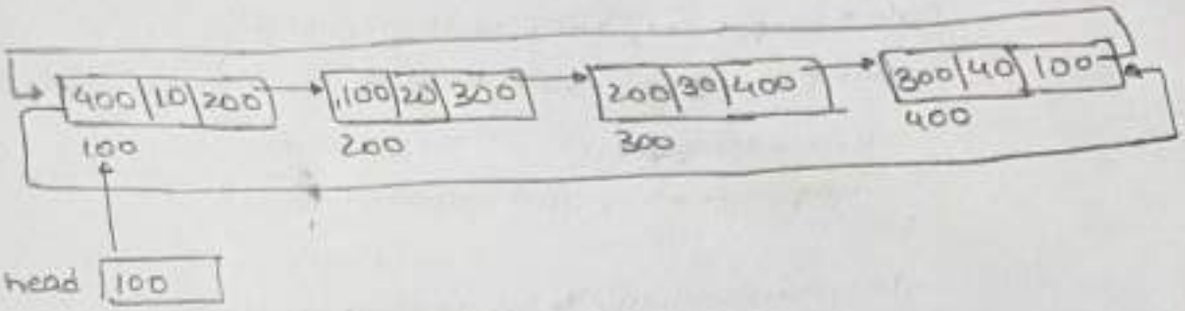
- WAP to delete the last node of a D.L.L.

```
deleteATEnd(node *head)
    if(head==null) return.
    node *temp1=head; *temp2=head->next;
    while(temp2->next!=null){temp1=temp1->next;
          temp2=temp2->next;
    }
    if(temp2==null)
    if(head->next==null){
          free(head);
          head=null;
    }
    temp1->next=null;
    free(temp2);
}
```

- Circular Double Linked List (C.D.L.L)
- C.D.L.L 1st node previous will point to last node & last node next pointer will point to 1st node

T.C. of C.D.L.L

	Begin	End
insert	O(1)	O(1)
delete	O(1)	O(1)



- WAP to insert at Begin & End in CDLL

Begin:

```
InsertAtBeginCDLL(node *head){
    node *curr = (node *)malloc(sizeof(node));
    if(curr==null) exit(1);
    scanf("%d", curr->data);
    curr->next = null;
    curr->prev = null;
    if(head == null) head = curr; head->next=head;
    if (head->next==head){
        curr ->next = head;
        head->next = curr;
        head->prev = curr;
        head= curr;
    }
    node *temp=head;
    temp->prev->next=curr
    curr->prev= temp->prev;
    temp->prev= curr;
    curr->next =temp;
    head = curr;
}
```

End: InsertionAtEnd(node *head)

```
node *curr = (node *) malloc (sizeof(node));
if curr == null) exit(2);
Scanf("%d", curr → data); curr → next = null;
curr → prev = null;
if(head == null) head = curr; head → next = head;
node *temp = head;
while (temp → next != head) temp = temp → next;
temp → next = curr;
curr → prev = temp;
curr → next = head;
head → prev = curr;
temp = null;
}
```

*WAP to delete from Begin & End of a CDLL.

Begin:

```
DeleteATBeginCDLL (node *head)
if (head == null) return;
if(head → next == head)
        head → next = null; free(head);
}
node *temp = head;
head → prev → next = head → next.
head → prev = null
head = temp → next;
temp → next = null
Free (temp);
}
```

end:

```
DeleteATEndCDLL( node *head)?
    if (head==null) return
    if (head->next==null) { free(head); head=null;}
    node *prev_node = head->prev;
     head->prev = prev_node->prev;
     prev_node->next =null;
     prev_node->prev = null;
     free (prev_node);
}
```

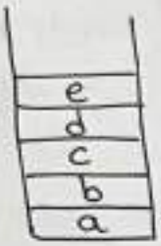
Note:

- next level Data Structure after Linked list is "Skip list".

Stack

- Stack is a Data structure used to store Data element with the restriction of first in Last out. or last in first out
- the element are inserted and deleted from same end called "Top"
- One side open & one side clore.

Ex:



Insertion: a,b,c,d,e

deletion: e,d,c,b,a

- Application of stack:
  - recursive funct^n call
  - balancing paranthesis.
  - HTML & XML tag checking.
  - Towers of Hanoi
  - page back in brocoser
  - Infix to postfix conversion
  - postfix evaluation
  - Fibonacci Series

- Stack ADT:
- Declaration of data:
  1. Space to store Stack element
     - array
     - Linkedlist.
  2. a variable "Top" pointing to Top most element in stack

- Declaration of operation:
  1. push(x):
     - ↳ element to insert
     - the above operation insert the element in the stack with the help of top
  2. pop(): the operation delete the element from stack using "Top"
  3. IsFull(): the operation return true if the stack is full
  4. Is empty(): the operation return true if stack is Empty

- Implement stack:
  - using array;

Initially
Top=-1

overflow condition
Top==N-1

underflow condition
Top==-1

Push:
Top: -1 0 1 2 3 4
Top++;
S[Top]=x;

Pop:
Top: 4, 3 2 1 0 -1
x=S[Top--];

- WAP to perform push op^n in the stack;

```
                stack  size  Top  element to insert
push(S,n,Top,x){
    if(Top==N-1) pf("Overflow condition"); return;
    else{ S[++Top]=x;}
}
```

- WAP to perform pop op^n in the stack;

```
int pop(S,N,Top){
    int x=0;
    if(Top==-1) pf("underflow condition"); return -1;
    else x=S[Top--];
    return x;
}
```

gate.

Q] A program attemps to generate as many string as possible of the string "abcd" by pushing the characters a,b,c,d. in the same order onto a stack, but it may pop off the top character. which one of the following string can't be generated using the program?

(a) abcd. (b) dcba (c) cbad (d) cabd

a,b,c,d pop when the element is in stack

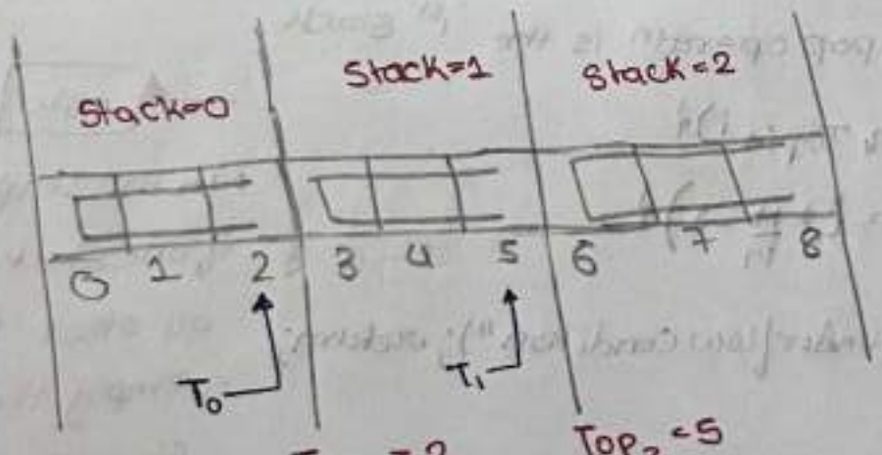
pop one by one

push till c & the pop & at last push & pop d

can't be generate

- Implementing multiple stacks in a single arrays:
- In order to execute any program we required stack, if we want to execute multiple program then we need multiple stack in the memory. these concept is ~~[illegible]~~ similarly to implementing multiple Stack in a single array.

array size (N) = 9, no. of stack(m) = 3, size of each stack $= \left(\frac{N}{m}\right) = \frac{9}{3} = 3$



Initial top of i^{th} stack (Top$_i$) $= i * \frac{N}{m} - 1$

Initially ⇒ Top$_0$ = -1

Top$_1$ = 2

Top$_2$ = 5

T$_0$ = -1, 0, 1, 2

T$_1$ = 3, 4, 5

Initial Top of 0th stack (Top_0) = $0 * \frac{9}{3} - 1 = -1$

Initial Top of 1st stack (Top_1) = $1 * \frac{9}{3} - 1 = 2$

Initial Top of 2nd stack (Top_2) = $2 * \frac{9}{3} - 1 = 5$

NOTE

- i^{th} stack overflow condition arises when "Top$_i$" reaches to initial Top of $(i+1)^{th}$ stack

- WAP to perform push operation in the i^{th} stack.

Stack size, no of stack, Top, stack no., element to be inserted

```
push(S, N, M, Top_i, i, x){
    if(Top_i == (i+1)*N/M - 1){
        Pf("overflow condition");
        return;
    }
    else{
        Top_i++;
        S[Top_i] = x;
    }
}
```

- WAP to perform pop operation is the i^{th} stack

```
int pop(S, N, M, Top_i, i){
    if (Top_i == i*N/M - 1){
        Pf("underflow condition"); return;
    }
    else{
        return S[Top_i--];
    }
}
```

note

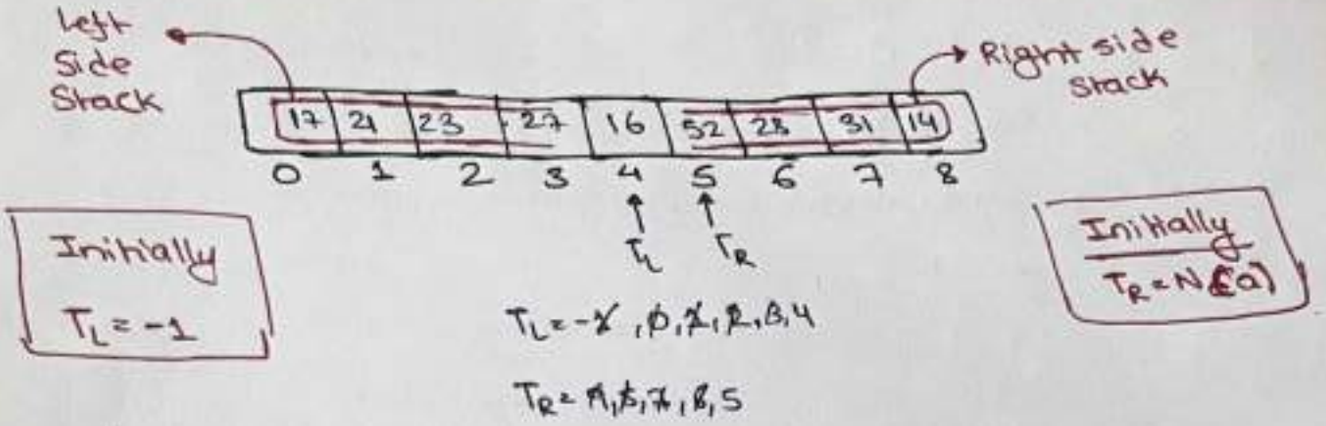
- In the implementation if one stack is full & all other stack are empty that still it is showing overflow condition.

- even though some empty slot are available they are not efficiently utilized.

- to avoide these problem we will use another approach.

- Implementing multiply stacks in a single array efficient ways.

array size (N) = 9 no. of stacks (M) = 2



- In above implementation we are using two stack in a single array growing in the opposite direction.

- WAP to perform push & pop operation in both the stack.

overflow condition

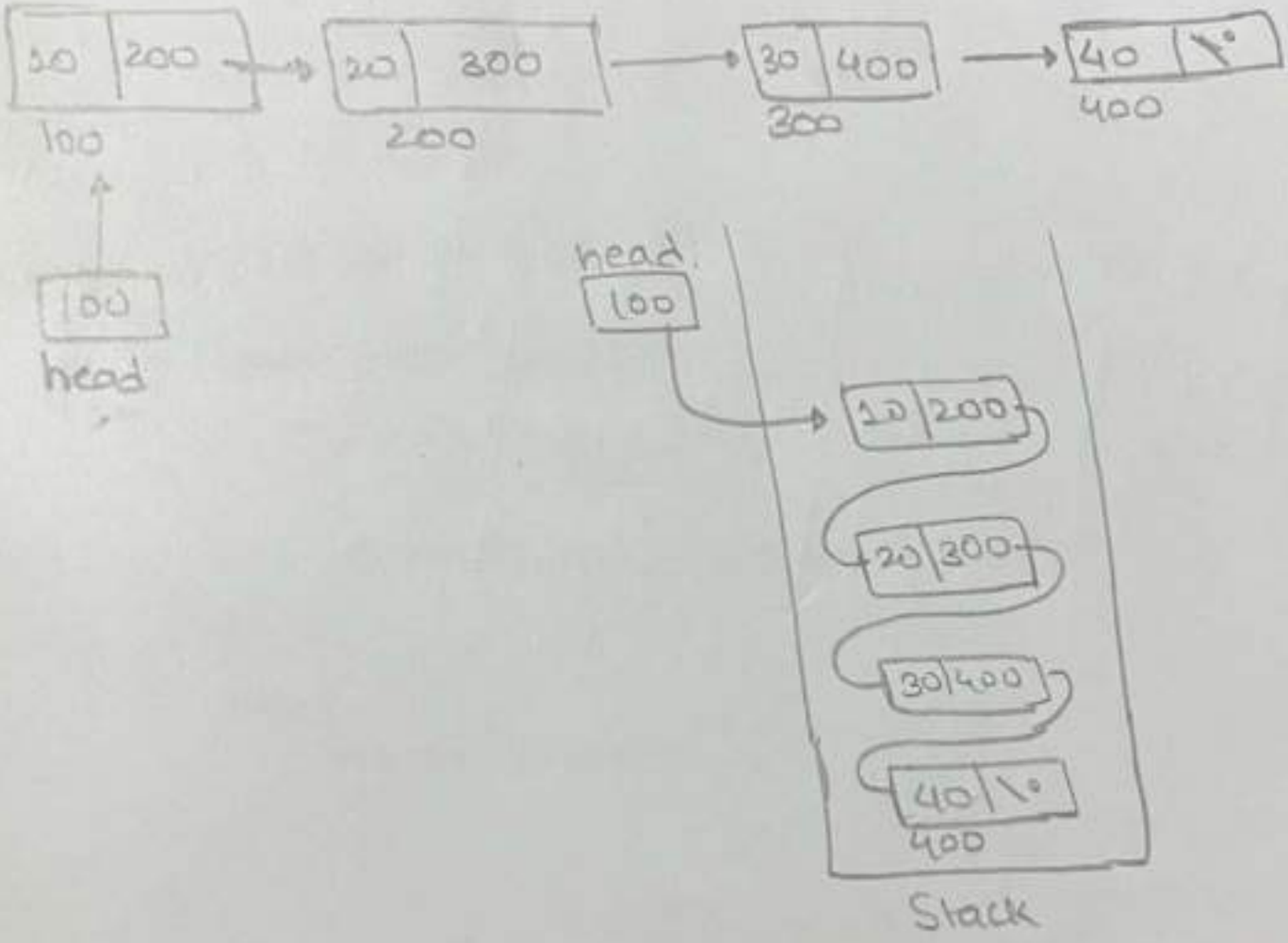
$T_R - T_L = 1$

$T_R = T_L + 1$

$T_L = T_R - 1$

for both the stack.

- Implementing Stack using Linked list



• : Insertion & deletion will always happen from the head node.

• WAP to perform push operation in the stack using linkedlist.

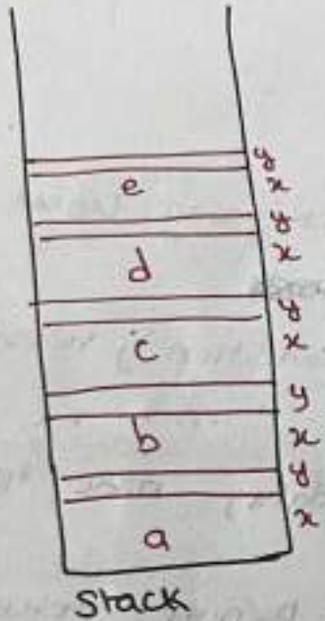
```
                                  → element to insert
push(node *head, int x){
        node *temp=(node *) malloc (sizeof(node));
        if(temp==null){("stack overflow condition"); exit(1);}
        else{
            temp→data=x;
            temp→next=head;
            head=temp;
        }
        temp=null;
    }
```

• WAP to perform pop operatn in the stack using Linked list? TC:O(1)

```
int pop(node *head){
    if(head==null){ pf("underflow condition"); return -1; }
    else{
        int x=head→data;   node *p=head.
        head=head→next;
        free(p); p=null;  return x;
    }
}
```

- Find avg. life time of an element in the stack:
- The time for insert / delete an element take 'x' & there is a delay of "y" b/w every such operation.
- The life time is considered as time elapsed from end of Push to start of pop operation that removes an element from stack.
- Find avg. life time of an element.

pg 3.8 Pg no. 356



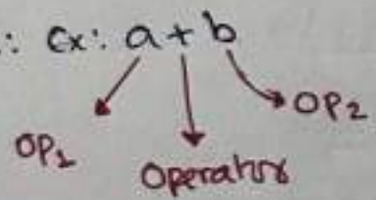
element = a, b, c, d, e

element	Life time
e	y
d	2x+3y (y+x+y+x+y)
c	y+x+y+x+y+x+y+x+y ⇒ 4x+5y
b	6x+7y
a	8x+9y.

Infix, prefix, postfix

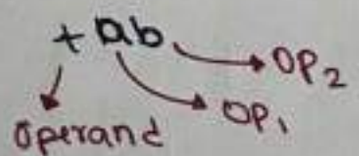
- Binary operation b/w 2 operand.

Infix: Ex: a+b

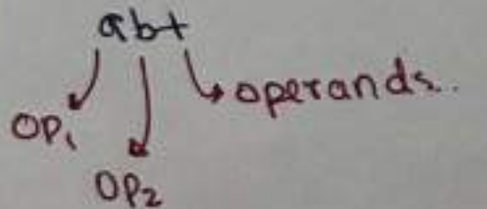


Prefix:

- Binary operatn before 2 operands.



Postfix: Binary operatn after 2 operand



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

Prefix: A+B*(C+D)/F+D*E

⇒ A+B*+CD/F+D*E

⇒ A++*B+CD/F+D*E

⇒ A+/*B+CDF+D*E

⇒ A+/*B+CDF+*DE

⇒ ++A/*B+CDF*DE

Postfix: A+B*(C+D)/F+D*E

⇒ A+B*CD+/F+D*E

⇒ A+BCD+*/F+D*E

⇒ A+BCD+*F/+D*E

⇒ A+BCD+*F/+DE*

⇒ ABCD+*F/++DE*

⇒ ABCD+*F/+DE*+

Infix: a+b*c-d/e↑f*g+h

Prefix: a+b*c-d/e^f*g+h

⇒ a+b*c-d/^ef*g+h

⇒ a+*bc-d/^ef*g+h

⇒ a++*bc-/d^efg+h

⇒ a++*bc-*/d^efgh

⇒ -+a*bc*/d^efgh+h

⇒ +-+a*bc*/d^efghh

Postfix: a+b*c-d/e^f*g+h

⇒ a+b*c-d/ef^*g+h

⇒ a+bc*-d/ef^*g+h

⇒ a+bc*-def^/*g+h

⇒ a+bc*-def^/g*+h

⇒ abc*+-def^/g*+h

⇒ abc*+def^/g*-+h

⇒ abc*+def^/g*-h+

4.co Infix: f*(cd-b)^(a/c))^(e+g)*h

Prefix: f*(cd-b)^(a/c))^(e+g)*h

⇒ f*(-db^ac)^+eg*h

⇒ f*^-db/ac^+eg*h

⇒ f*^^-db/ac+eg*h

⇒ *f^^-db/ac+eg*h

⇒ **f^^-db/ac+egh.

postfix: f*(cd-b)^(a/c))^(e+g)*h

⇒ f*(db-^ac/)^(eg+)*h

⇒ f*db-ac/^^eg+*h

⇒ fdb-ac/^*eg+^*h

⇒ fdb-ac/^eg+^*h*

- Prefix to postfix conversion:
  - In order to convert from prefix to postfix we need

1) operators followed by
2) two immediate operands.

then, convert it into postfix.

ex: prefix: *+cd-ba

postfix: *cd+-ba

*cd+ba-

cd+ba-*

ex: prefix: *^/+cae*db^fg

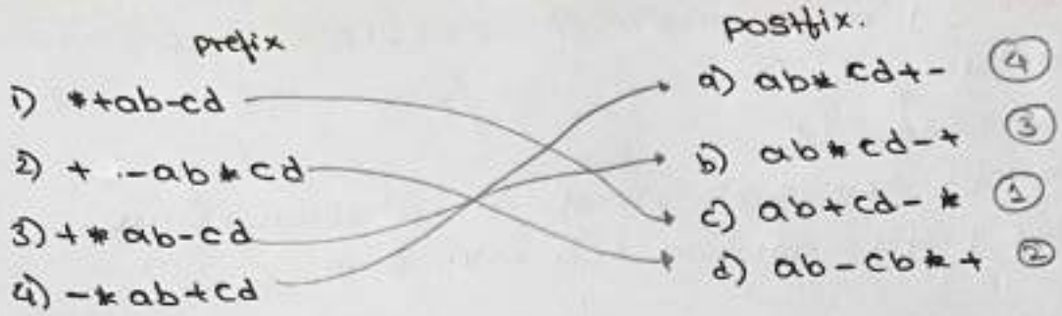
postfix: *^/ca+edb*fg^

⇒ *^ca+e/db*fg^

⇒ *ca+e/db*^fg^

⇒ ca+e/db*^fg^*

Q] match the following



- Reverse polish notation
- Postfix to infix conversion [we use bracket]
- to convert from postfix to infix
  1) two operands followed by
  2) immediate operator.

  then convert it into infix expression.

postfix : ab*cd/-

Infix : (a*b)cd/-

⇒ (a*b)(c/d)-

⇒ ((a*b)-(c/d))

- Prefix to infix conversion
  1) Operator followed by
  2) two immediate operands

  then convert the prefix into infix expression

prefix : *+ad-cd

⇒ *(a+d)(c-d)

⇒ ((a+d)*(c-d))

Note

- In order to evaluate Infix expression we need to scan multiple time jumping from one place to other place. A/c to the precendence of the operator.

the Best solution is

1) Convert infix expression into postfix in a single scan.
2) evaluating postfix expression in a single scan

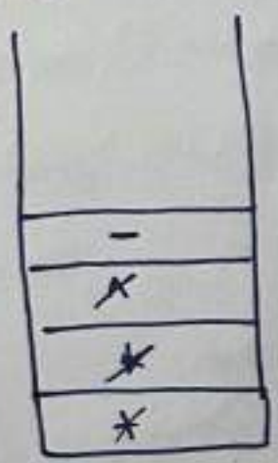
- convert Infix to postfix:
(using operator stack):

Top of stack	Next operator	operator
Low	high	push
high	low	pop
same	same(L→R)	pop
Same	same(R→L)	push

- If any braces come treat it as new stack until the closing braces

Ex: Infix:- a+b*c^d-e

Postfix: abcd^*+e-



Stack

max stack size used (M.S.U)

3

Infix: a+b*c-d/e^f*g+h

postfix: abc*+def^/g*-h+

Stack

max stack size used ≥ 3

Ex: when we will reach symbol "f" while something during conversion from infix to postfix what will be the content of stack

Infix string: a*b-c/e*f+g

postfix: ab*ce/f

Ex: Infix ((a+b)*(c-d))

postfix: ab+cd-*

M.S.U = 4

Ex: Infix:

(((P+Q)*(R+S))/T)+(A*(B+C))

Postfix: PQ+RS+*T/ABC+*

m.s.u = 5.

*postfix evaluation: (using operand stack)

procedure:

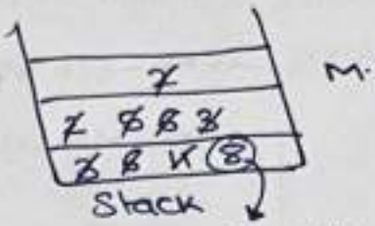
1) if (symbol == operand)
   push (symbol);

2) if (symbol == operator)
   a) op_2 = pop();
   b) op_1 = pop();
   c) push (op_1 <operator (symbol)> op_2);

3) At the end, Result will be in the stack.

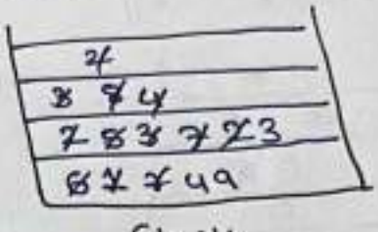
Postfix: 3 2 * 5 + 6 2 / -

OP_1	operator	OP_2	Result
3	*	2	6
6	+	5	11
6	/	2	3
11	−	3	(8) //



Postfix: 6 2 3 + - 3 8 2 / + * 2 ^ 3 + : 52

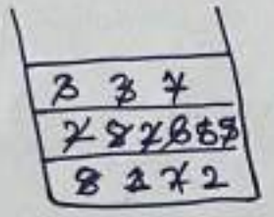
OP_1	operator	OP_2	Result
2	+	3	5
6	−	5	1
8	/	2	4
3	+	4	7
1	*	7	1
7	^	2	49
49	+	3	52



Q] 8 2 3 ^ / 2 3 * + 5 1 * -

↓ 2

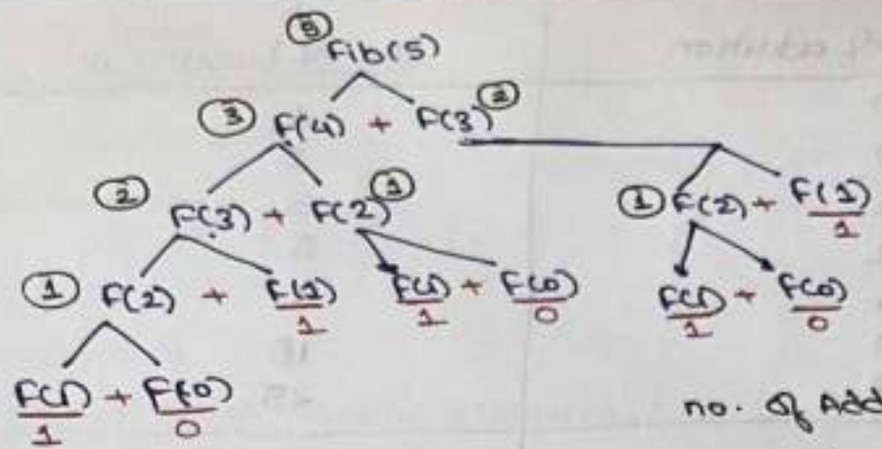
4 m.s.u → (3) //



- Fibonacci Series.

n	0	1	2	3	4	5	6	7	8	9	10
Fib(n)	0	1	1	2	3	5	8	13	21	34	55

$$Fib(n) = \begin{cases} n, \text{ if } n==0 \text{ or } n==1 \\ Fib(n-1) + Fib(n-2), \\ \quad 0 \text{ otherwise} \end{cases}$$

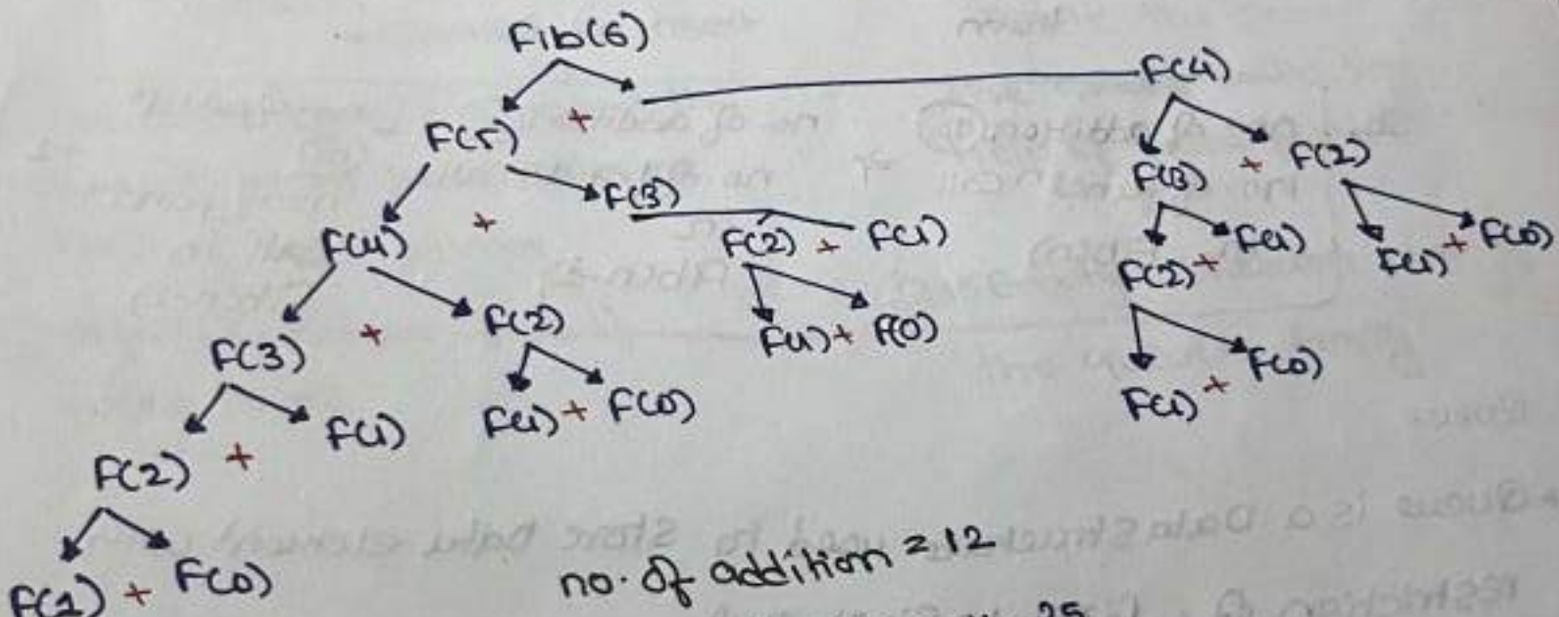


no. of Addition = 7

no. of funct^n call = 15

Note

- Funct^n call log is preorder & Funct^n evaluation is postorder.



no. of addition = 12

no. of funct^n call = 25.

Fib(n)	no. of addition	no. of funct^n call.
0	0	1
1	0	1
2	1	3
3	2	5
4	4	9
5	7	15
6	12	25
general^n formula	F(n) + F(n-1) + F(n-2)	

no. of addition required for individual +1 (extra for combining them.

So, no. of addition (OR) no. of funct^n call in Fib(n) = no. of addition (OR) no. of funct^n call in Fib(n-1) + no. of addition (OR) no. of funct^n call in Fib(n-2) +1

Queue

- Queue is a Data Structure used to store Data element with restriction of First In First Out.
- the element are inserted from one end & deleted from another end
- it uses two pointer
  1) Front → used for deleting an elements.
  2) Rear → used for inserting an element.

- Applicatn of Queue.
  1. Printer Spooler Daemon
  2. Ticket Reservation.
  3. Job scheduling.

- Queue A.D.T :-

1) Declaration of data:
   a) space to store Queue elements.
      - array
      - Linked list
   b) two pointers front & Rear.

2) Declaration of operation:
   a) enqueue(x) ↳ element to insert

   the above operation insert the element in the queue with the help of rear pointer.

   c) isfull(): Return true if queue is full.

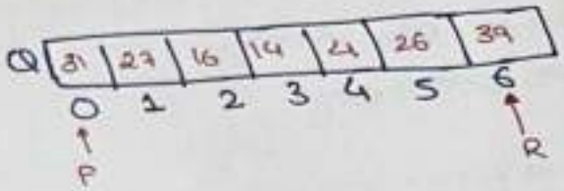
   b) dequeue()
      Delete the element from the queue with the help of front pointer.

   d) isEmpty(): Return true if the queue is Empty

- Implementing Queue :

① using array:

int Q[N], N=7



Initially
front = -1
rear = -1

enqueue:

f = -1
R = -1, 0, 1, 2, 3, 4, 5, 6

R++;
Q[R] = x

Overflow condition!
R == (N-1)

underflow condition
f == -1

- WAP to enqueue

① Initially check for overflow condition

② if it is not overflow & if it is 1st insertion then increment both front & rear otherwise increament only rear pointer & then insert the element in the queue using rear pointers.

Queue, size, front, Rear, element to insert.

enqueue(Q, N, F, R, x){

if(R == (N-1)) Pf("overflow condition"); return;

if(R == -1){
F++; R++; } 1st insertion.

}else{
R++;
}
Q[R] = x;
}

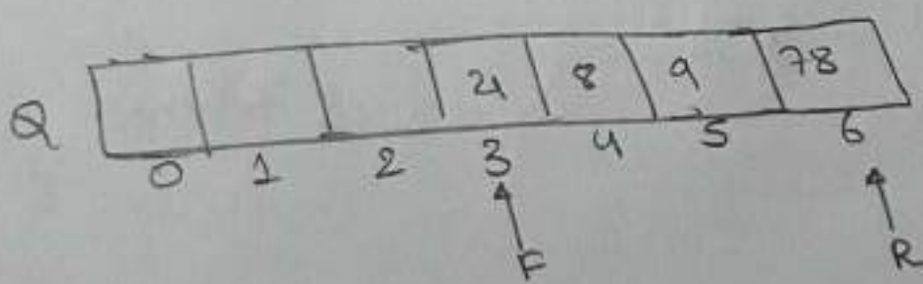
• WAP to dequeue.

```
int dequeue(Q,N,F,R){
    if(F == -1){
        pf("underfloco condition");
        return;
    } int y=0;
    if(F==    R   ){ y=Q[F];
        F=-1; R=-1;
    }
    else{
        y=Q[F++];
    }
    return y;
}.
```

note

- in the above implementation of linear queue. when the rear pointer reaches to right most pointer & front pointer is some where in the middle then still it is showing overflow condition.
- even though queue is showing some empty slot still they are not efficiently utilized.
- to avoid these problem we will go for circular queue.

int Q[N] N=7



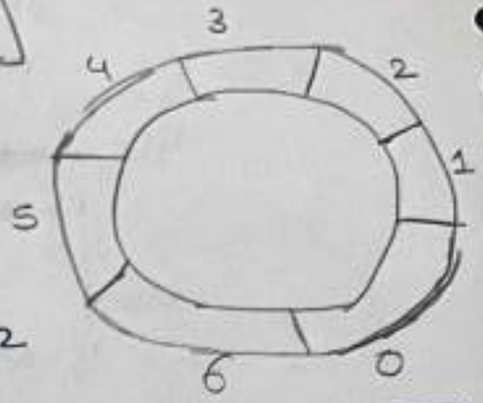
F=-1, 0, 1, 2, 3

R=-1, 0, 1, 2, 3, 4, 5, 6

- Circular Queue

int Q[N], N=7

Initially
F=-1
R=-1



R=(R+1)%N
F=(F+1)%N

enqueue:

F= -1, 0, 1, 2, 3

R= -1, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2

(R+1)%N==F
(6+1)%7==3
0==3
(F)

(R+1)%N==F
(1+1)%7==3
2==3
(F)

(R+1)%N==F
(0+1)%7==3
1==3
(F)

(R+1)%N==F
(2+1)%7==3
3==3
(T)

Overflow.

overflow condition
(R+1)%N==F

underflow condition
F==-1

- WAP to perform enqueue operatn in circular queue.

```
enqueue Circular(Q,N,F,R,x){
  if((R+1)%N==F) Pf("overflow condition"); return;
  if(R==-1){
     F++; R++;
  }else{
     R=(R+1)%N;
  }
  Q[R]=x;
}
```

- WAP to perform dequeue operatn in circular queue.

```
int dequeue(Q,N,F,R){
  if(F==-1) Pf("underflow condition"); return;
  int y=0; y=Q[F]
  if(F==R){
     y=Q[F];
     F=-1, R=-1;
  }
  else{
     F=(F+1)%N;
  }
  return y;
}
```

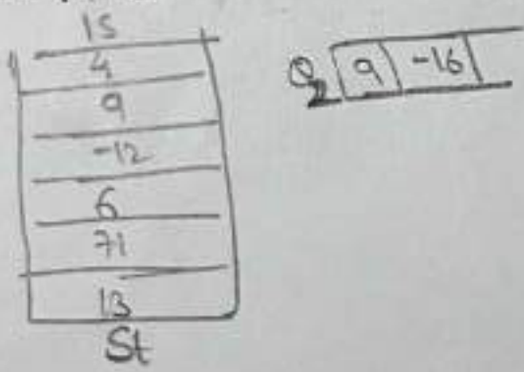
Q] Conside the following program which uses · 3 data structure. Queue(Q1), Queue(Q2), & stack(St).

- the program take' Q1' value as input & perform some operation

Q_1 = 13, 71, 6, -12, 9, 0, 4, 15, 12, 8, -16, 0, 18, 13

```
xyz(Queue Q1)?
    int s=0;
    while(!isEmpty(Q1))?
        x=dequeue(Q1);
        if(x==0)?
            st=pop(st);
            enqueue(Q2,s);
            s=0;
        }
        push(st,x);
    }
}
```

1) what is sum of all value of stack after completion of above program. 148



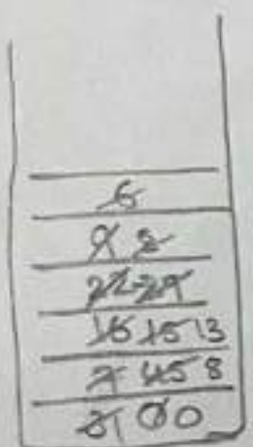
2) what is m.s.v = -12

3) what is sum of all value of Q_2 = -7

4) what is max Q_2 size used = 2.

Q1] Q_1 = 31, 7, 16, 22, 9, 0, 45, 15, -29, 8, 16, 0, 8, 15

```
ABC (Queue Q1)?
    while(!isEmpty(Q1))?
        x = dequeue(Q1);
        if (x==0)?
            pop(st);
            pop(st);
            while(!isEmpty(st)) enqueue(Q2, pop(st));
        }
        push(st, x);
    }
}.
```



Q_2

16	7	31	-29	15	45	0

① What is sum of all value of stack after completion of above program : 21

② What is M.S.U : 6

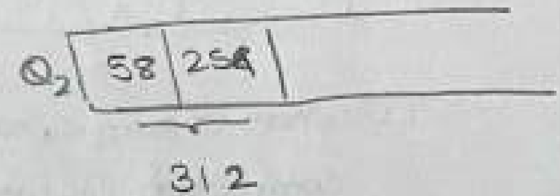
③ What is sum of all value of Q_2 ? 85

④ What is max Q_2 size used : 7.

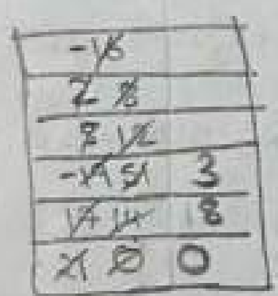
Ex:

Q_1 = 21, 19, -19, 8, 2, 0, 14, 51, 12, 8, -16, 0, 18, 3

```
Abc (Queue Q1){
    Static int s;
    while (!isEmpty(Q1)){
        x=dequeue(Q1);
        if (x==0){
            while(!isEmpty(St))
                S+=pop(St);

            S+=S;
            enqueue(Q2, S);
        }

        push(St, x);
    }
}.
```



Static S



St

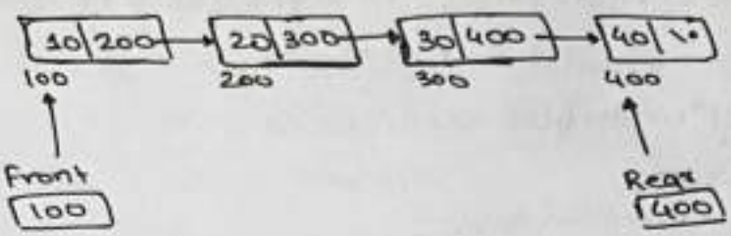
1. what is sum of all value of stack (21)

2. what is M.S.U (6)

3. what is sum of all value of Q_2 (312)

4. what is max. Q_2 size used. (2)

• using linked list :



Initially
- Front = null
- Rear = null

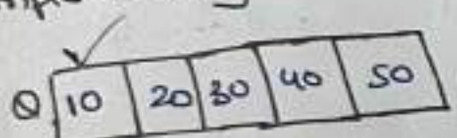
• WAP to perform enqueue operation using Linked list.

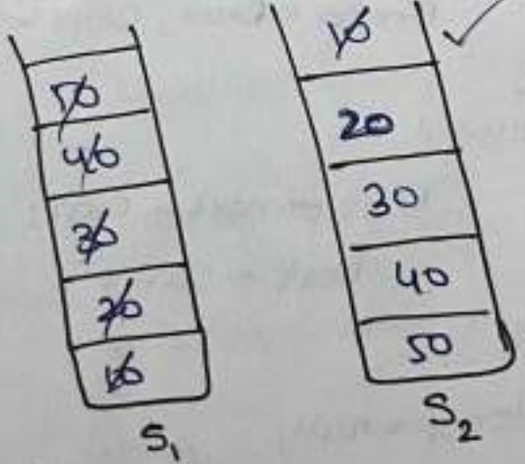
```
enqueueLL(node * front, node *rear, int x){
    node *curr = (node *)malloc(sizeof(int));
    if(curr==null) pf("overflow condition"); return;
      curr->data=x;
      curr->next=null;
    if(rear==null){
        Front = Curr; Rear = Curr;
    }
    else{
        Rear->next = Curr;
          Rear = Curr;
    }
    temp=null;
}
```

• WAP to perform dequeue operation

```
int dequeueLL(node* Front, node *rear){ node *p = Front;
        if(Front==null){
            pf("underflow condition");
            return;
        } int y= Front->data;
        if(Front == Rear){
            Front=null; Rear=null;
        }
        else{
            Front=Front->next;
        }
        free(p)
        p=null;
        return y;
    }
```

• Implementing Queue using stacks:





- WAP to perform enqueue & dequeue operation.

```
int dequeue (S1, S2){
    if (S2 is NOT Empty){
        return pop(S2);
    else if (S1 is Empty){
        pf("Queue is Empty");
        return -1;
    }
    else{
        while (S1 is not Empty){
            x=pop(S1);
            push(S2,x);
        }
        return pop(S2);
    }
}
```

```
enqueue(S1,x){
    push(S1,x);
}
```

- Consider the above pseudo code to implement queue using stack. Let 'n' insert & m(≤n) delete operation be performed in an arbitary order on an empty queue. & let 'x' & 'y' be the no. of push & pop operation performed respectively. which one of the following is true for all 'm' & 'n'?

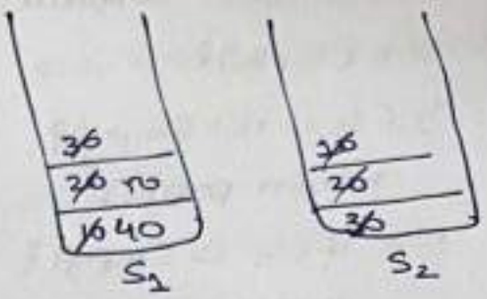
(a) $n+m \le x \le 2n$ & $2m \le y \le n+m$.

(b) $n+m \le x < 2n$ & $2m \le y \le 2n$

(c) $2m \le x < 2n$ & $2m \le y \le n+m$

(d) $2m \le x < 2n$ & $2m \le y \le 2n$.

n=5
m=3



max = push = x = 5+5 = 10
= 2n

push=5
pop=3

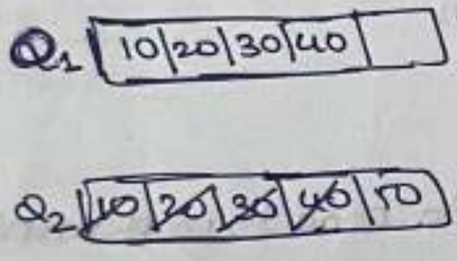
push=3
pop=3

max = pop = y = 5+3 = 8 = n+m

min = push = x = 5+3 = 8 = n+m

min = pop = y = 3+3 = 6 = 2m.

- Implementing stack using Queues:

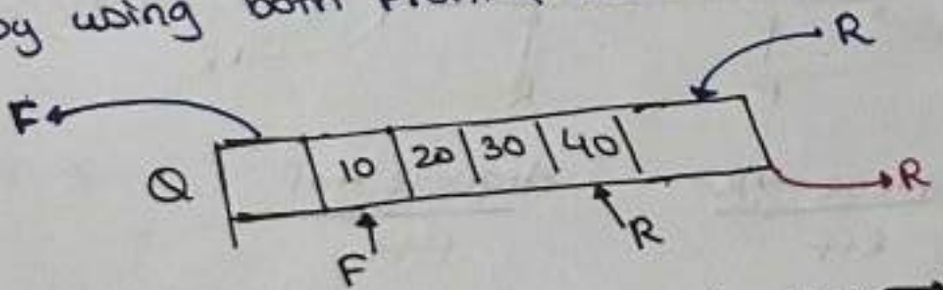


- write psuedo code for push & pop operatn.

```
int pop(Q1,Q2){
  if(Q1 is empty){
    if(Q2 is empty){
      pf("stack is empty");
      return -1;
    }
    else{
      while(Q2 does not contain one element){
        x= dequeue(Q2);
        enqueue(Q1,x);
      }
      return dequeue(Q2);
    }
  else{
    while(Q1 doesnot contain one element){
      x=dequeue(Q1);
      enqueue(Q2,x);
    }
    return dequeue(Q1);
  }
}
```

```
push(Q1,Q2,x){
  if(Q1 is Empty)
  enqueue(Q2,x);
  else
  enqueue(Q1,x);
}
```

- Input Restricted Queue:
- enqueue operatn are restricted but the dequeue operatn can be done by using both Front & Rear.



Enqueue → R

R++

dequeue → F

F++

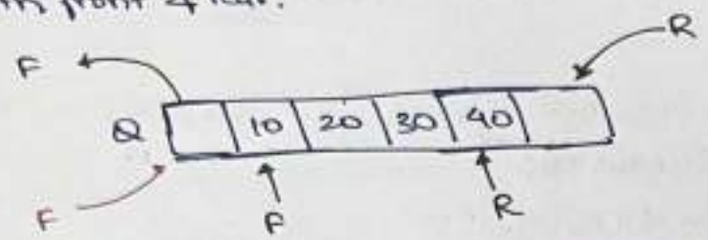
dequeue → R

R--

- it is not following queue property. it is following both FIFO & LIFO (Last in First out)

- O/p Restricted Queue.

- dequeue operation is restricted but enqueue operation can be done by using both front & rear.



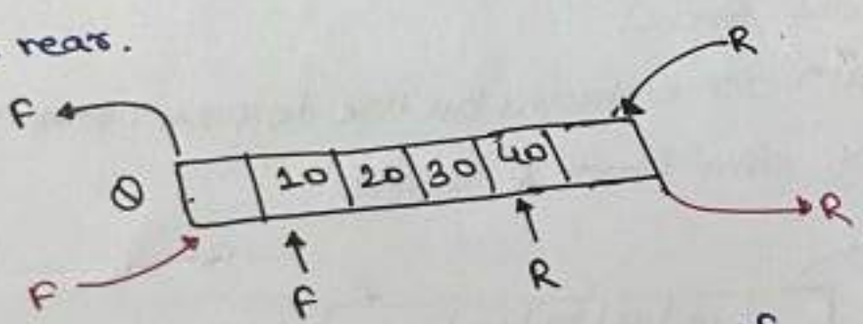
Enqueue → R
R++

dequeue → F
F++

Enqueue → F
F--

- it is not following queue property, it is following both first in firstout & last in firstout.

- Double Ended Queue.
- enqueue & dequeue operation can be done by using both front & rear.



Enqueue → R
R++

dequeue → F
F++

enqueue → F
F--

dequeue → R
R--

- it is not following queue property it is following both FIFO & LIFO

- priority Queue:

element: 85, 30, 16, 10, 25, 70, 90, 45.

1) Ascending priority Queue (delete min)

10, 16, 25, 30, 45, 70, 85, 90

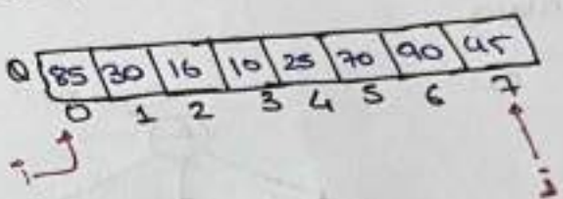
2) Descending priority Queue (delete max)

90, 85, 70, 45, 30, 25, 16, 10

- Implementing Ascending priority Queue (delete min):

element : 85, 30, 16, 10, 25, 70, 90, 45

int Q[N]; N=8



Initially.

i = -1

j = -1

'j' will always point to last element.

enqueue:

i = -1 0

j = -1 0 1 2 3 4 5 6 7

J++

Q[j] = x

Dequeue:

1. scan the entire array & find the position of minimum element
2. y = Q[min] // 10
3. Q[min] = Q[j];
4. J--;
5. return y;

T.C = Θ(n)

B/C of unsorted array

using sorted array then

enqueue

T.C = O(n)

dequeue

T.C = O(1)

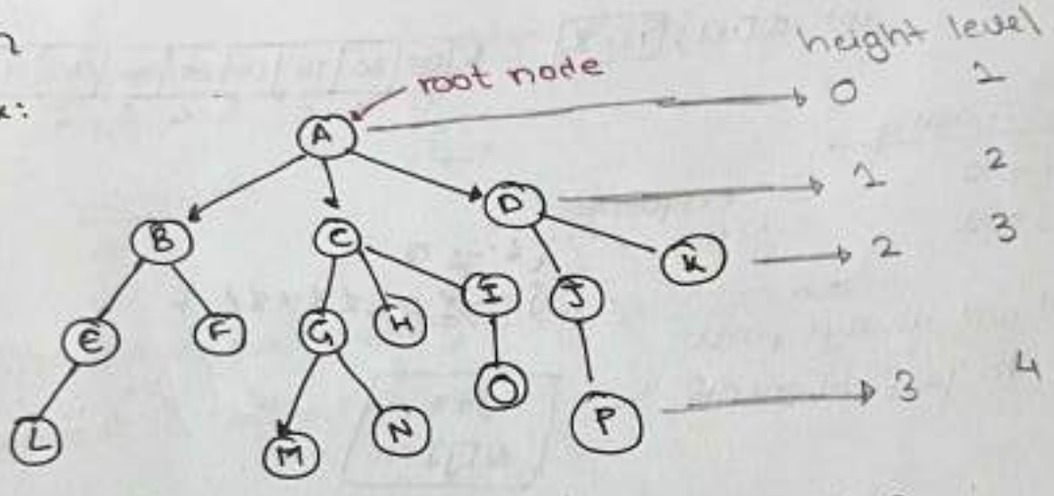
Tree

- Tree is non-linear data structure used to store the data element by following (parent-child) hierarchial Relationship b/w the data element

- Type of tree
  - n-ary tree
  - Binary tree.

- Degree of tree:
  the degree of the tree represent the max. no. of child node : any node can contain

Ex:



1) Leaf node: L, F, M, M, H, O, P, K
2) Internal node: A, B, E, C, G, I, D, J
3) Ancestors of 'P' : J, D, A

- height of the tree: 3
- level of the tree: [height+1] 4

- Siblings of "B": C, D
- Left most child of 'C': G
- Right sibling of "J" NO
- Right sibling of 'E': F
- Right sibling of 'A' : NO

Height of the tree

the longest path from root not to any leaf node. (root node height is general consider as "0").

- Representation of tree:
  - array
  - linked list (*)

- to find max element in binary tree T.C: O(n)

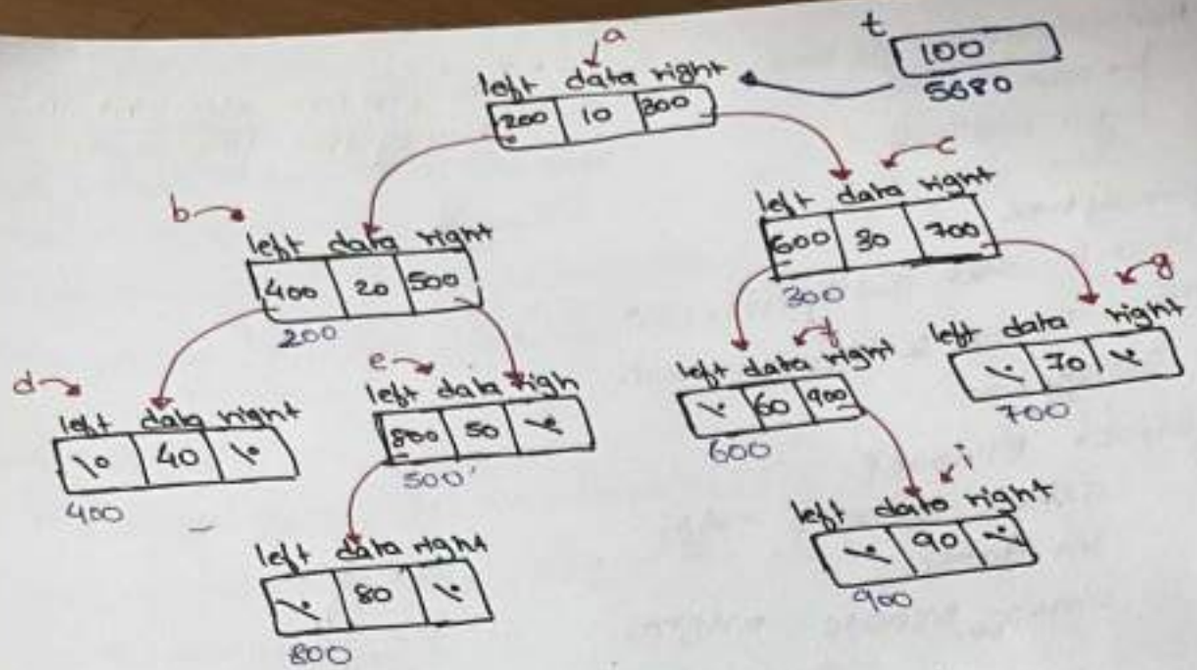
- Binary tree
- A tree in which every node can contain atmost 2 child is known as Binary tree.

①
```
Struct BTnode{
    Struct BTnode *left;
    int data;
    Struct BTnode *right;
};
Typedef Struct BTnode node
```

②
```
node a = {Null, 10, Null};
node b = {Null, 20, Null};
```

③
```
a.left = &b;
a.right = &c;
b.left = &d;
b.right = &e;
```

④
```
node *t = null
t = &a;
```



note

- pf(t) = 100
- if (t == null)
  - ↳ if above condition is true then it means that Binary tree is empty
- (t → left) is given
  - ↳ it means that left subtree is given
- (t → right) is given
  - ↳ it means that right subtree is given
- pf(t → data); // (10)
- pf(t → left); // (200)
- if(t → left == null && t → right);
  - ↳ if above condition is true that means Binary tree has only one node & that node is known as leaf node.
- (t) is given
  - ↳ it means that entire tree is given

- pf(t → left → right → data); // (50)

- In the tree program will be done using recursion bc every node is having some behaviours.

Ex: write a program to find no. of leaf node in the given Binary tree.

```
int NL(node *t){ if(t==null) return 0;
    if(t->left==null && t->right==null) return 1;

    return (NL(t->left) + NL(t->right));
```

Ex: WAP to find height of the Binary tree.

```
int HL(node *t){
    if(t==null) return 0;

    return max(HL(t->left),HL(t->right)) +1;
}
```

Ex: WAP to Find whealther Binary tree is a strict Binary tree.

```
int StrictBT( node *t){
    if(t==null)return 1;
    if((t->left!=null && t->right==null)||(t->right!=null &&
                                           t->left==null))
        return 0;
    if(t->left==null && t->right==null) return 1;
    return (Strict(t->left) && Strict(t->right));

}
```

- WAP to find no. of internal node in the Binary tree.

```
int NI(node *t){
    if(t==null) return 0;
    if(t->left==null && t->right==null) return 0;
    return (1 + NI(t->left) + NI(t->right));
}
```

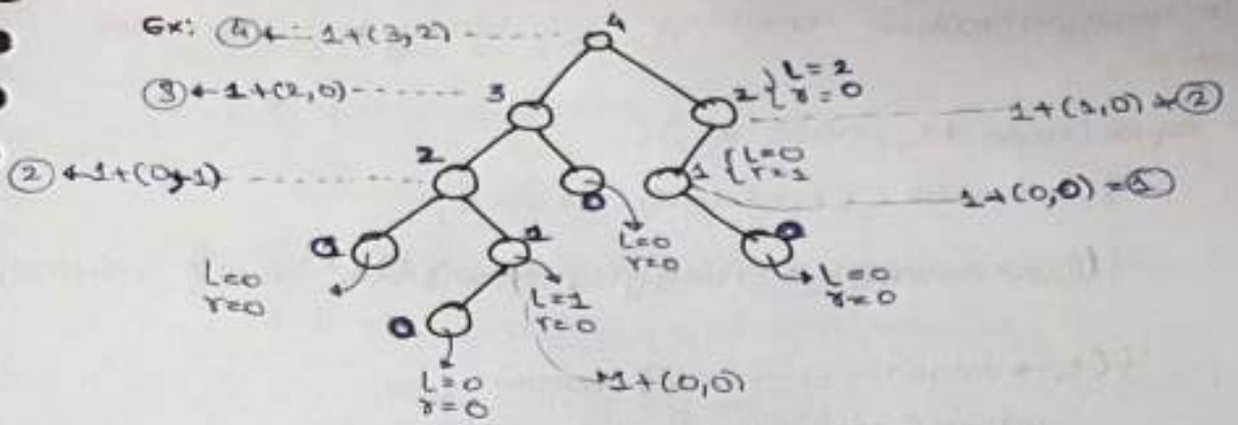
- WAP to find total no. of node in the given Binary tree

```
int Totalnode(node *t){
    if(t==null) return 0;
    return ( Totalnode(t->left) + Totalnode(t->right) +1);
}
```

to make these good program
you should add
these line
∴ `if(t->left==null && t->right==null) return 1;`

- WAP to find height without using max funct^n.

```
int height (node *t){
    if(t==null) return 0;
    if(t->left==null && t->right==null) return 0;
    int l= height(t->left);
    int R= height(t->right);
    return (1+(l>r ? l :r));
}
```



T·C = O(n)

- WAP to find level in BT.

```
int levels(node *t){
    if(t==null) return 0;
    if(t->left==null && t->right==null) return 1;
    else {
        int l= levels(t->left);
        int r= levels(t->right);
        return (1+(l>r?l:r));
    }
}
```

T·C = O(n)

- WAP to check whether remaining two binary tree are equal or not.

```
int equal(node *t1, node *t2){
    if(t1==null && t2==null) return 1;
    if((t1==null && t2!=null) || (t2==null && t1!=null)) return 0;

    if(t1->value == t2->value)
        return(equal(t1->left, t2->left) &&
               equal(t1->right, t2->right))

}
```

- Strict Binary tree (S.B.T)
- a tree in which every node has exactly zero or two child node is called as Strict Binary tree

- WAP to convert the given Binary tree into its mirror image. Swapping data.

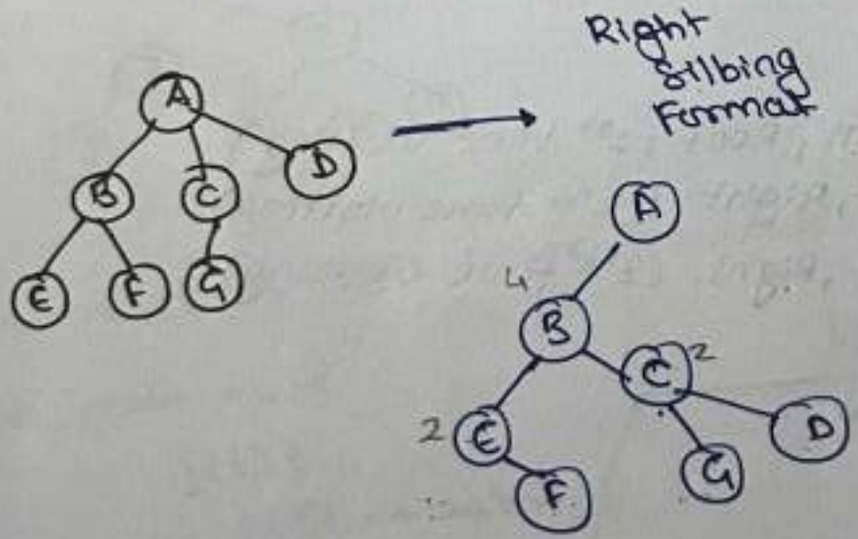
```
void MI(node *t){
    if(t==null) return;
    if(t->left==null && t->right==null) return;
    if(t->left!=null && t->right!=null){
        int *temp= t->left->data;
        t->left->data = t->right->data;
        t->right->data = temp;
    }
    return (MI(t->left) && MI(t->right));

}
```

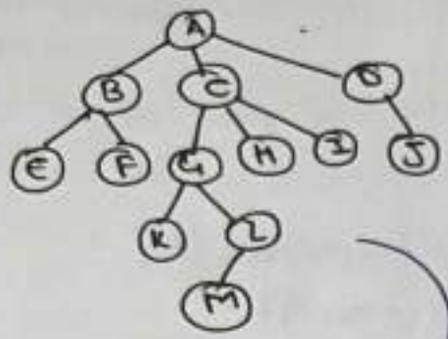
:. Swapping node not data.

```
void MI(node *t)
{
    if(t==null) return;

    if(t->left==null && t->right==null) return;
    if(t->left!=null && t->right!=null)
    {
        node *temp = t->left;
        t->left = t->right;
        t->right = temp;
    }

    MI(t->left);
    MI(t->right);
}
```

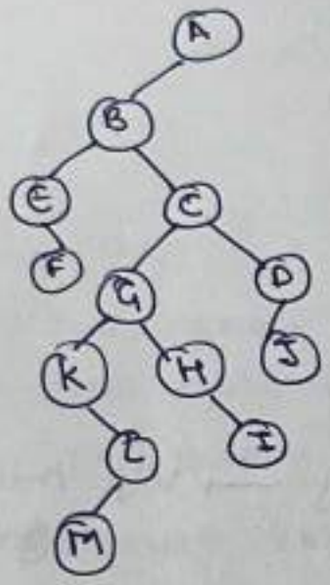
Pg: 151
Q: 12

[Arbitary tree ⟶ Binary tree ⟶ left most child, Right sibling.]





Right silbing format.

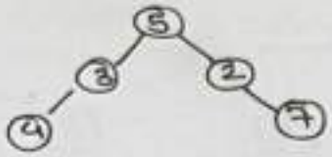


- Tree traversals.

postorder : left, Right, Root (1st time visiting)
Inorder : left, Root, Right (2nd time visiting)
preorder : Root, left, Right. (3rd time visiting)

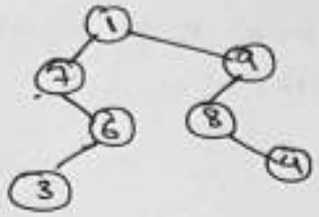
```
Preorder (node *t) {
    pf(t) {
        pf(t→data);
        Preorder (t→left);
        Preorder (t→right);
    }
}
```

Ex:



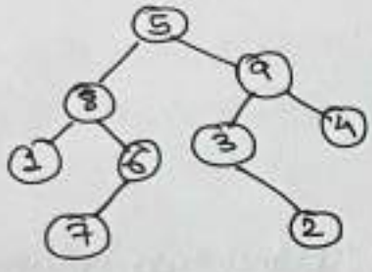
Preorder: 5,3,4,2,7
Inorder: 4,3, 5,2,7.
postorder: 4,3,7,2,5

Ex:



Preorder: 1,7,6,3,9,8,4
Inorder: 7,3,6,1,8,4,9
postorder: 3,6,7,4,8,9,1

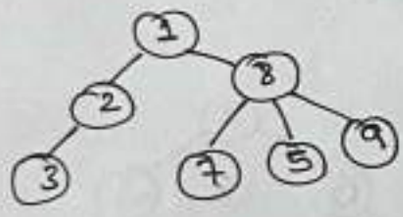
Ex:



Preorder: 5,8,1,6,7,9,3,2,4
Inorder: 1,8,7,6,5,3,2,9,4
postorder: 1,7,6,8,2,3,4,9,5

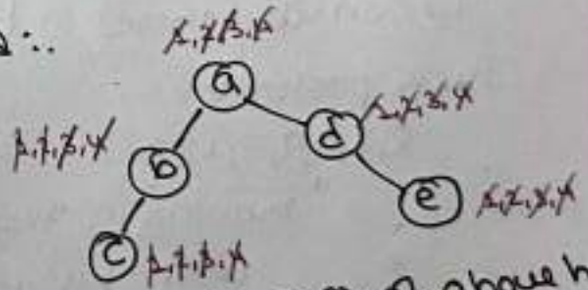
Inorder: 3,2,1,7, 8 5,9

Ex:



- Double order & Triple order Traversal :..

```
DO(node *t){
    if(t){
1.      pf(t->data);
2.      DO(t->left);
3.      pf(t->data);
4.      DO(t->right);
    }
}
```

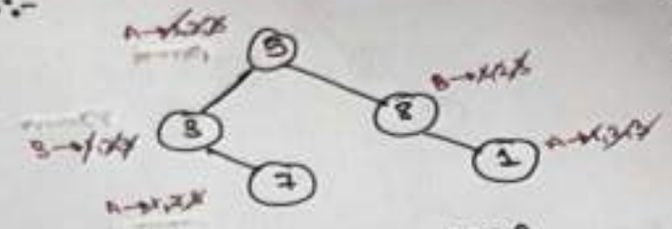


what is DO() of above tree
∴ a, b, c, b, a, d, d, e, e

Double order

- Indirected Recursion on trees:-

```
A(node *t)?
    if(t)?
      1. Pf(t → data);
      2. B(t → left);
      3. B(t → right);
    }
}
B(node *t)?
      if(t)?
        1. A(t → left);
        2. Pf(t → data);
        3. A(t → right);
      }
}
```



what is o/p of A(5)?

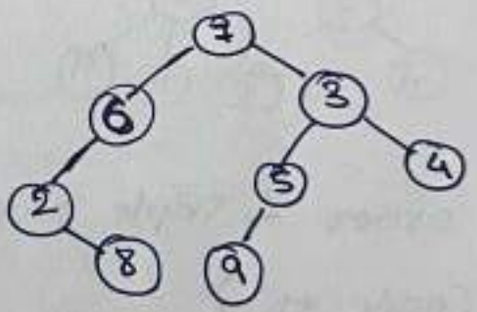
5,3,7,8,1

- Constructing unique Binary tree
- A unique Binary tree can be construct if any two traversal are given But one of them should be Inorder.

Ex: postorder: 8,2,6,9,5,4,3,7

Inorder: 2,8,6,7,9,5,3,4.

to construct these



Chuck postorder

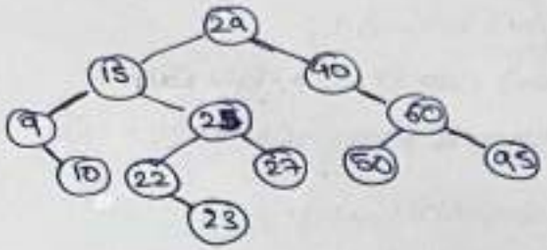
8,2,6,9,5,4,3,7

moving in these direction will fill us root node & then compare with inorder.

Gate:

Post order: 10,9,23,22,27,25,15,50,95,60,40,29.

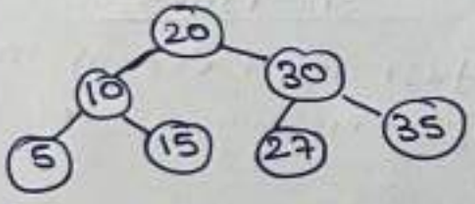
Inorder: 9,10,15,22,23,25,27,29,40,50,60,95



- Binary search tree (B.S.T):
  - Binary search tree is also a Binary tree But with the restriction that all the element in the left subtree should be smaller to the parent & all the subtree on the right should be greater than parent. (these rule is applicable for all node).
  - BST is used for searching an element.
  - All the element in the BST should be unique.

Ex:

element : 20,10,15,30,35,27,5.



Preorder : 20,10,5,15,30,27,35

Inorder : 5,10,15,20,27,30,35

Postorder : 5,15,10,27,35,30,20.

Note

- Inorder traversal of BST is a sorted order.
- only Inorder traversal of the BST is given, can we construct unique Binary tree? No.
- only postoreder traversal of BST is given. Can we construct unique Binary tree? yes.
- last element in the preorder & last element in the inorder is same (these is true only for complete Binary tree).

*WAP to search an element in an BST.

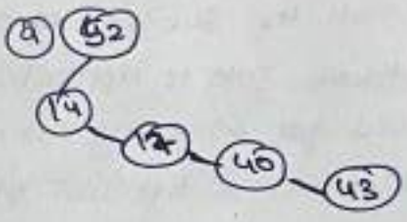
*search

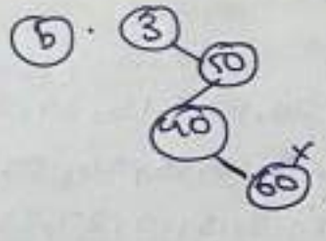
```
node (node *t, int x){
    if(t==null) return;
    if(x==t->data) return t;
    if(x>t->data)
        return(search(t->right,x))
    else return search(t->left,x);
}
```

Node
wc: O(n)
↳ left skewed or right skewed BST

Ex: consider the no. (1 to 100) in BST
when searching for element 43, which of the following sequence can't be the sequence of node examined?

a) 52, 14, 19, 40, 43
b) 3, 50, 40, 60, 43
c) 10, 65, 31, 48, 37, 43
d) 81, 61, 52, 14, 41, 43





Short trick

1 a) just insert element in a sequence from the min (10) to max (75)

10 43 57 55 60 64 75 } no issue

2 b) check for adjacent if it is lower then the current then all following should be lower than that or if adjacent is greater than all following should be greater.

▸ Search for 55

a) 10, 75, 64, 43, 60, 57, 55
b) 90, 12, 68, 34, 62, 45, 55
c) 9, 85, 47, 68, 43, 57, 55
d) 79, 14, 72, 56, 16, 53, 55

Smaller

90 12 68 34 62 45 55

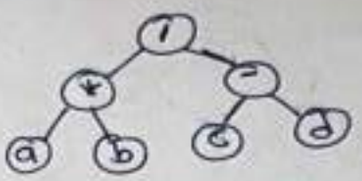
greater

• Construct uniqui binary tree of given reverse polish notation.

postfix: ab*cd-/ → convert it

Infix: (a*b)(c-d)/

↳ ((a*b)/(c-d))



expression tree.

gate:

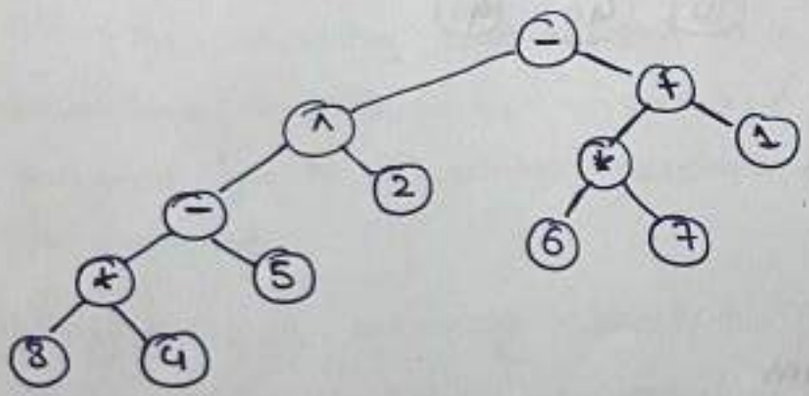
postfix: - 3 4 * 5 - 2 ^ 6 7 * 1 + -

Infix: (3*4) 5 - 2 ^ (6*7) 1 + -

((3*4) - 5) 2 ^ (6*7) 1 + -

(((3*4) - 5) ^ 2) (6*7) 1 + -

* (((((3*4) - 5) ^ 2) - ((6*7) + 1)



• Deletion in the BST

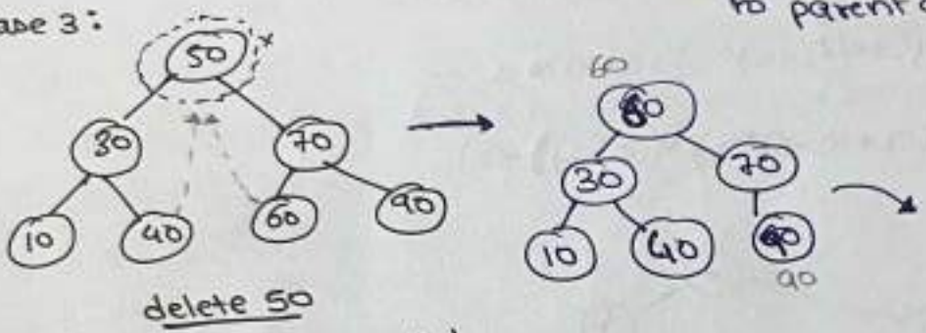
Case 1:

▸ If it is a leaf node then simple delete that node & update in its parent pointer as null.

Case 2:

▸ Child of 30 will be connected to parent of 30

Case 3:



▸ new root will be updated with inorder predecessor largest element in the left subtree) or inorder successor (Smallest element in the Right subtree.

Inorder:
10 30 40 50 60 70 90

Inorder predecessor (Largest element L.S.T)

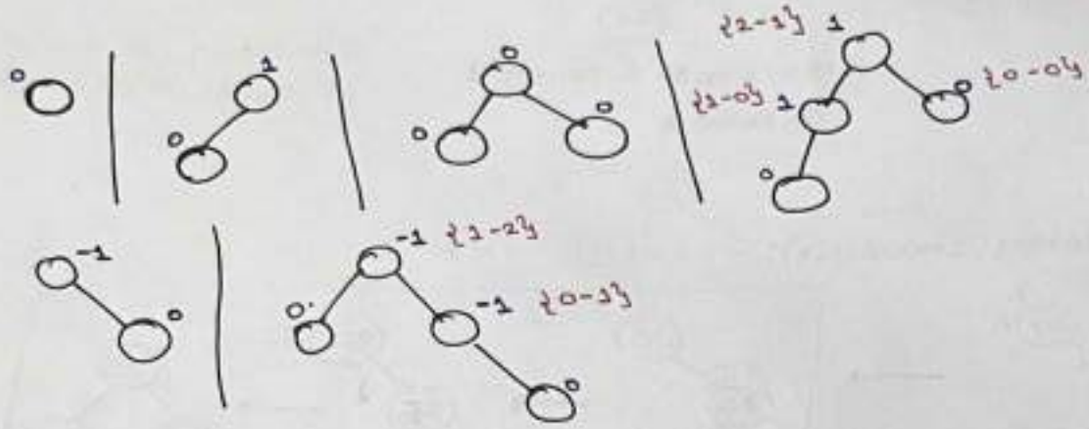
{ Right child will be always null }

Inorder Successor (Smallest successor)

{ left child will be always null }

- AVL Tree's (Height Balanced B.S.T):-
  (Adelson, Valsky, Landis).

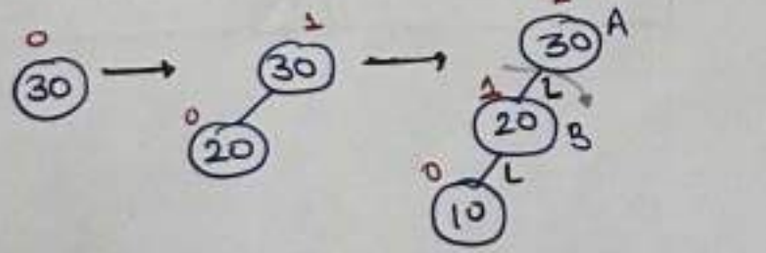
[Balanced Factor = height of L.S.T − height of R.S.T] ∴ Accepted balance factors = 0, -1, 1



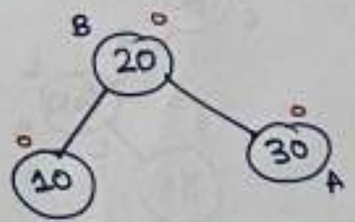
- Construction of AVL Tree's
- the procedure is same as constructing BST by inserting one element at a time
- when the new node is inserted, the Balance factor of all its ancestor's will be updated.
- if any node is becoming inbalance's then we need to apply proper rotation technique to make it balance

- Rotation technique:
- LL Imbalance:

{Left, Left}

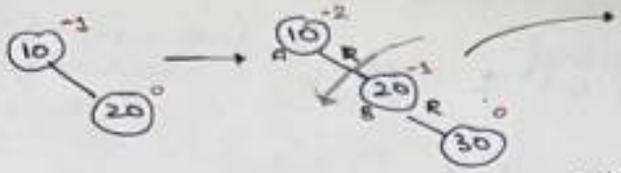


[Make node 'A' as child of node

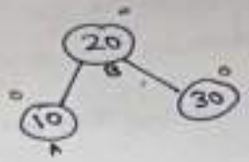


// Internally it involve single Rotation.

- RR Imbalance:

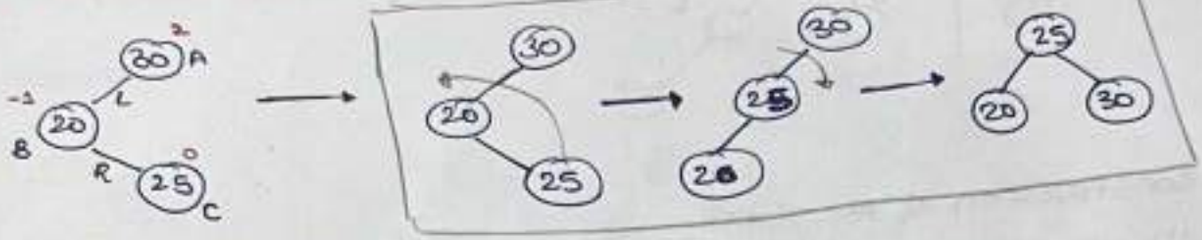


// make node A as child of node B



// internally it involve single rotation.

- LR Rotation (Imbalance):

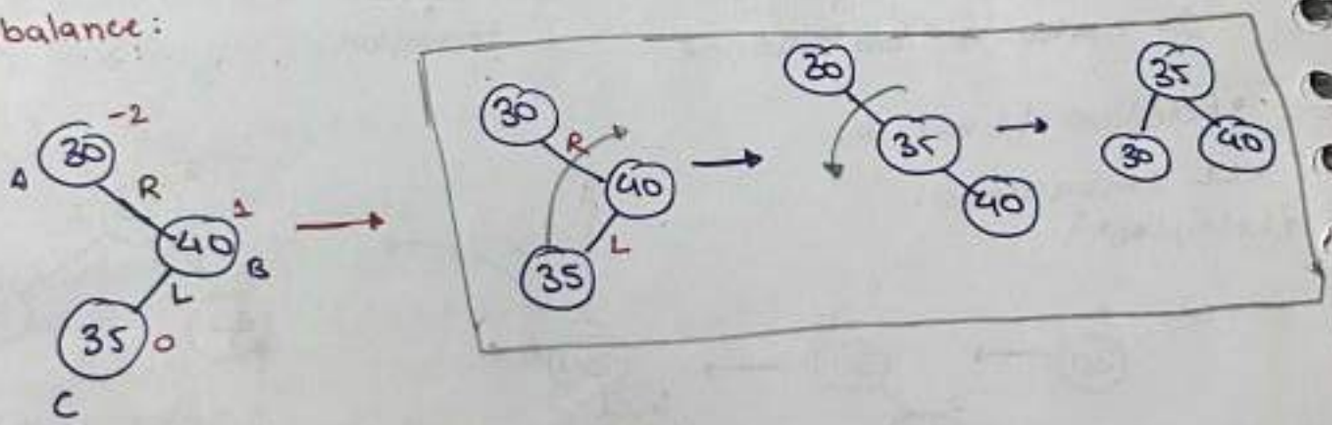


// Make node A & B as a child of node C.

// internally it involve double rotation.

{ go from towards newly insert node if there is an imbalance}

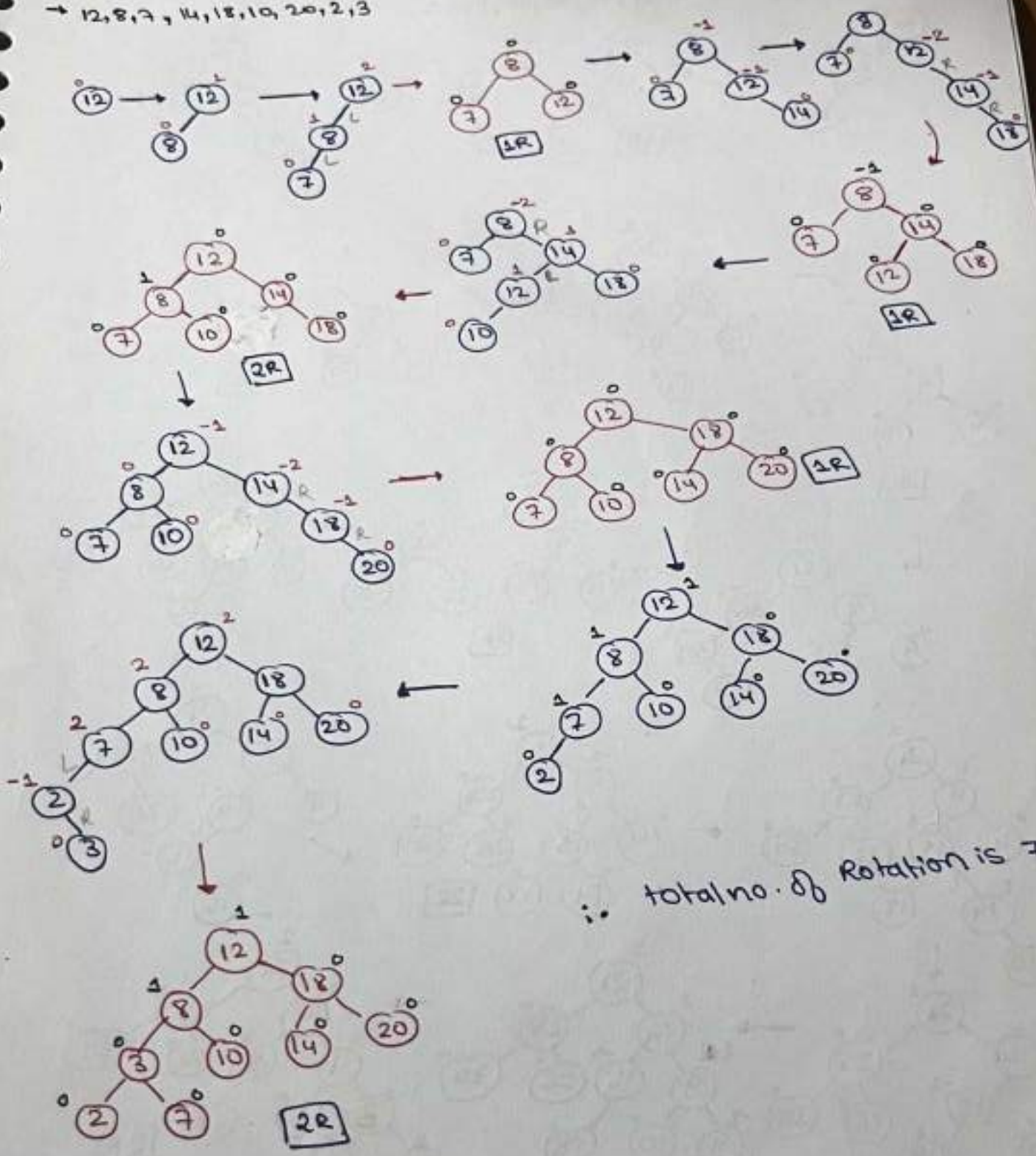
- RL Imbalance:



// make node A & B as a child of node C.

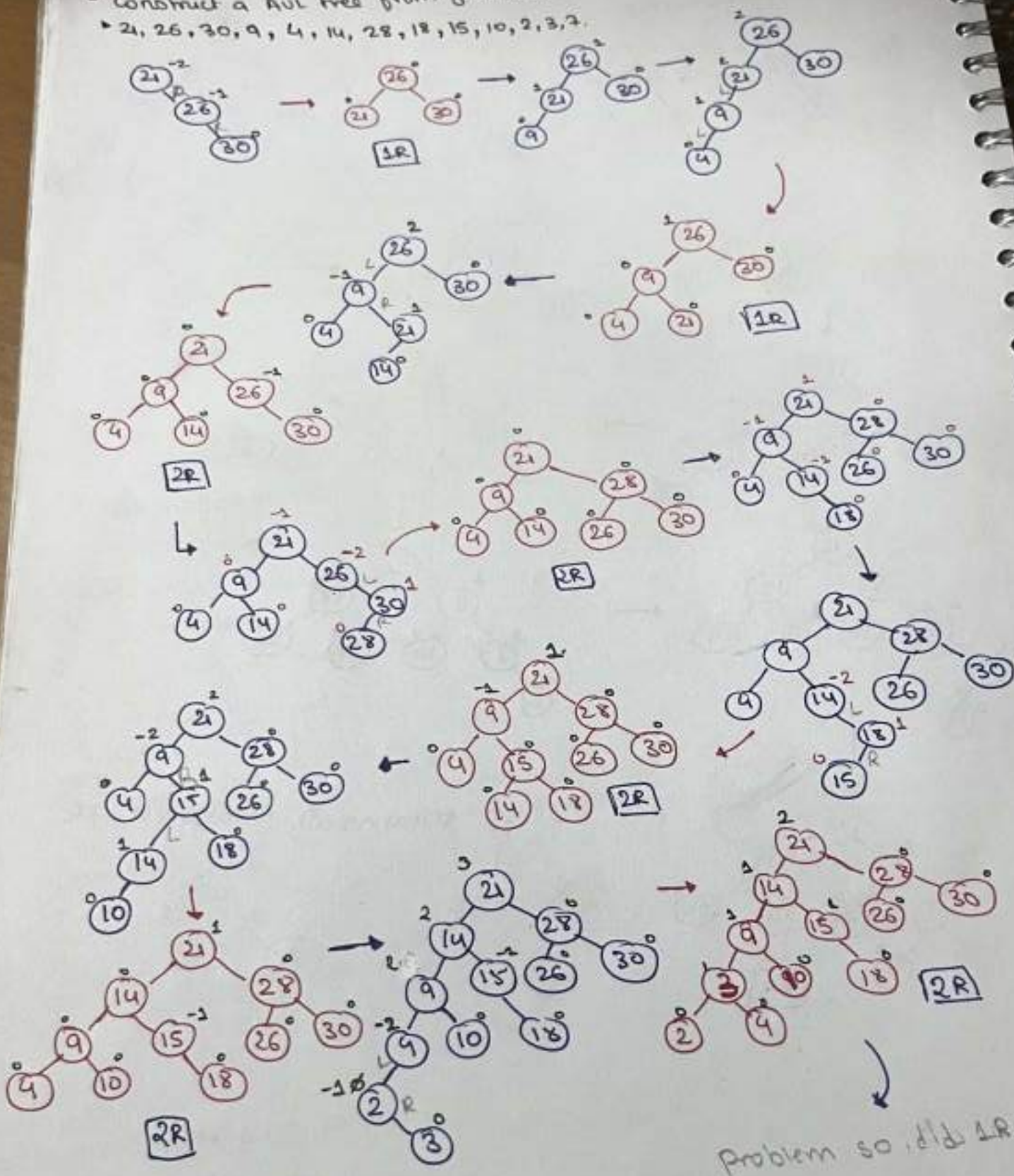
// internally it involve double rotation.

- Construct the AVL tree with given node
  → 12, 8, 7, 14, 18, 10, 20, 2, 3

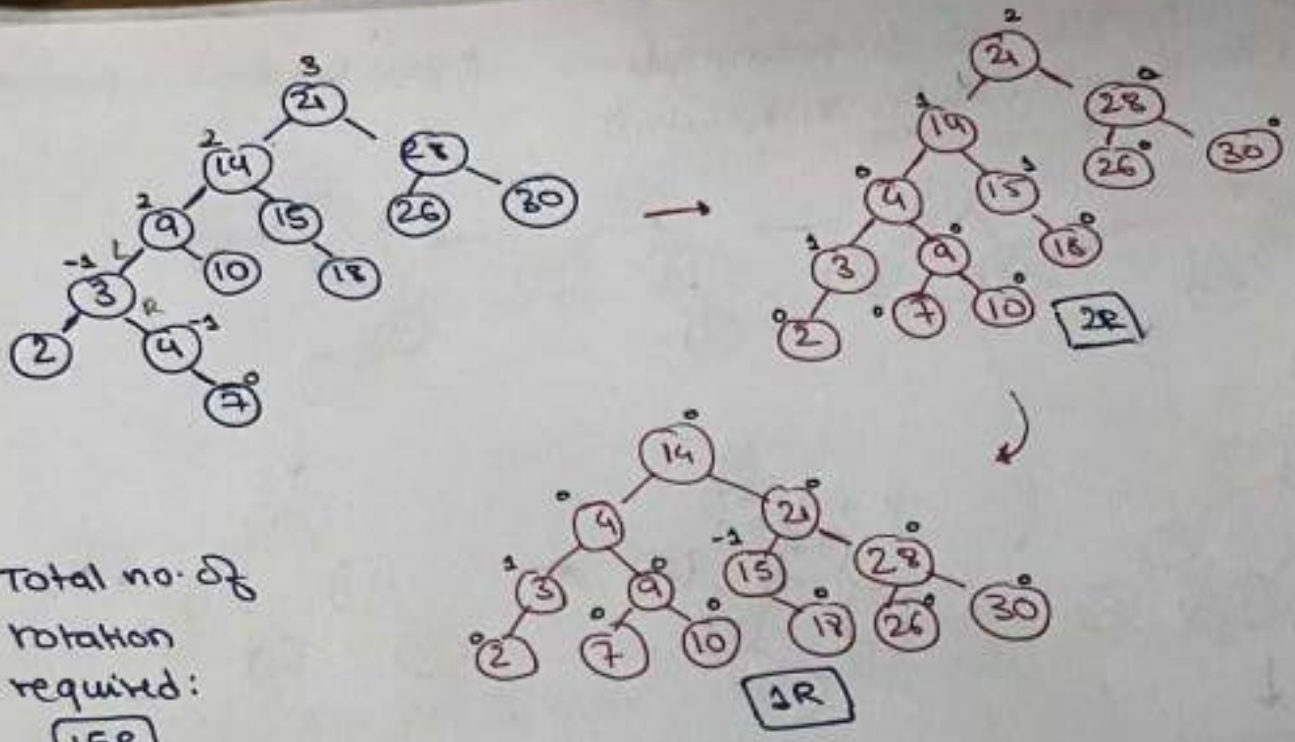


∴ total no. of Rotation is 7R.

- Construct a AVL tree from given node
- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 2



Problem so did 1R Rotation in last.

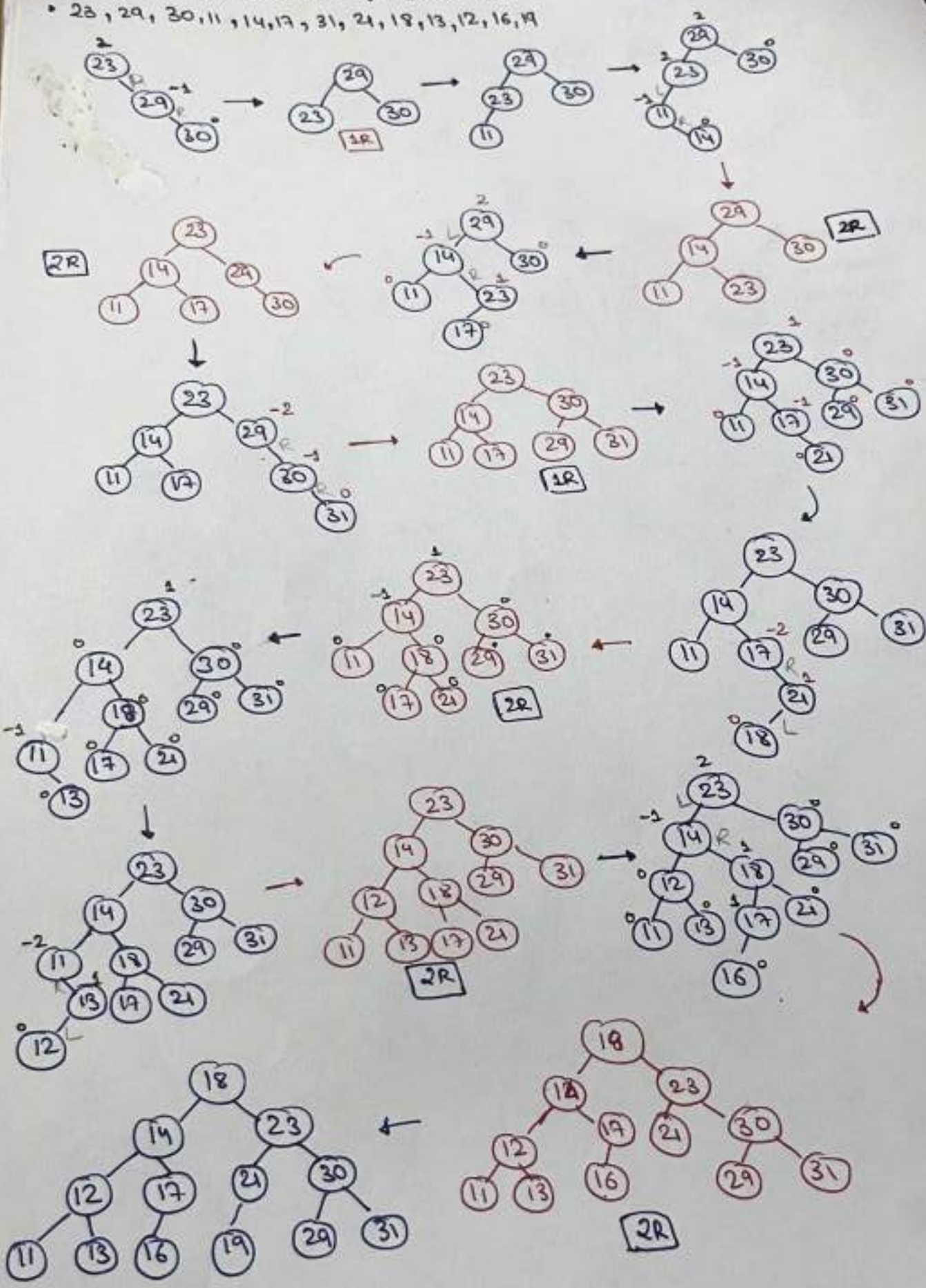


// Total no. of rotation required:

15R

- Construct the AOL tree for given node
- 23, 29, 30, 11, 14, 17, 31, 21, 18, 13, 12, 16, 19

// Total Rotation: 12 Rotation.



note: max. no of nodes in the binary tree (AVL tree also) with height 'h'

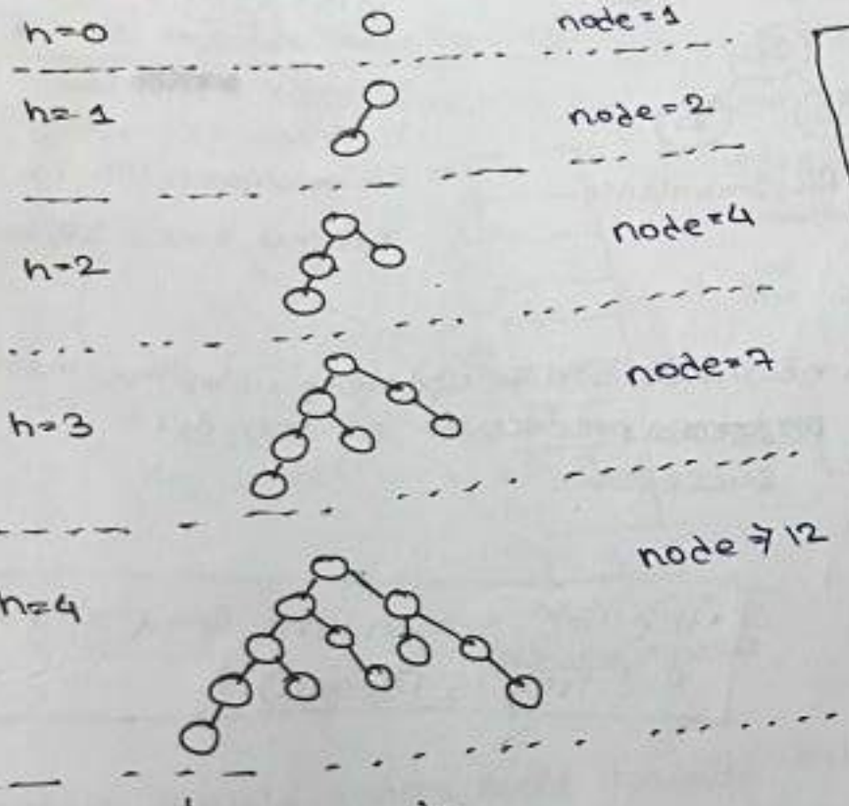
$2^{h+1}-1$

h=0 ⟶ 0 $= 2^{0+1}-1 \Rightarrow 2-1 = 1$

h=1 ⟶  $= 2^{1+1}-1 = 2^2-1 = 3$

h=2 ⟶  $= 2^{2+1}-1 \Rightarrow 2^3-1 \Rightarrow 7$

* Finding minimum no of node in AVL tree with height 'h':



Formula ⇒

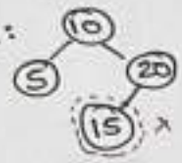
$min(h) \Rightarrow min(h-1) + min(h-2) + 1$

height(h)	min. node
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143
10	232

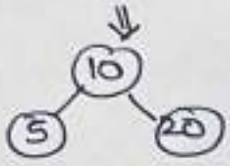
min. no. of node in AVL tree with height(h) = min. no. of node with height(h-1) + min. no of node with height(h-2) + 1

- Deletions in AVL Tree:
- Deletion in the AVL tree is same as BST but after deletion check the balance factor of every node. if any node is having imbalance then apply the proper rotation & make it balance.

case 1:

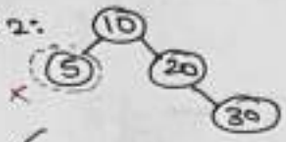


delete 15

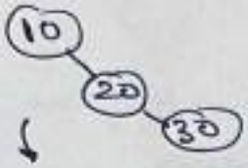


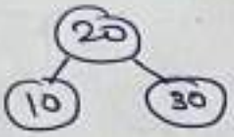
no balancing required

case 2:



delete 5

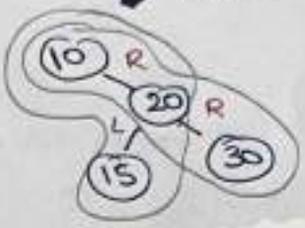




// RR Imbalance

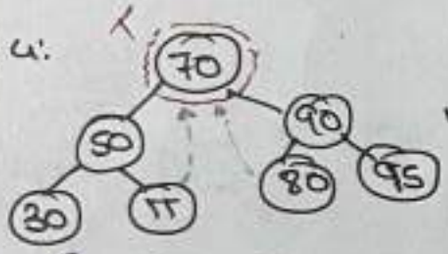
case 3:

delete 5.

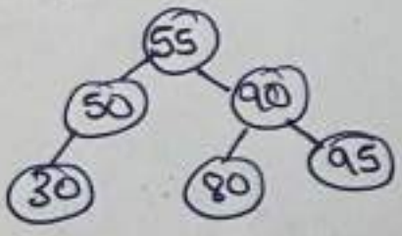


in above [redacted] case it is better to apply RR imbalance b/c RR internally takes 1 Rotation

Case 4:



delete 70

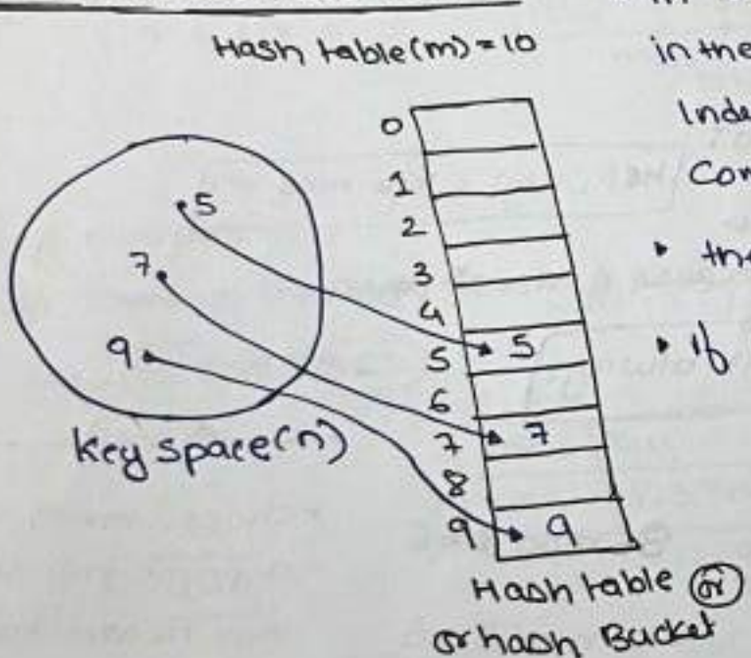


- new node will be updated, with the inorder [redacted]) predecessor or inorder successor

- the wc search time complexity in AVL tree is O(logn).

- Hashing:
  - its a technique used for searching an element
  - Hashing is mathematical computation over a key.
  - the goal of a hashing is to design the hash function which give min. no. of collisions
  - W.C Time complexity O(1) {99%}
  - the following component are used in hashing
    - → key space(n)
    - → hash functn
    - → Hash table (or) Hash Bucket(m).

- **Direct Address table (D.A.T):-**
  - In D.A.T Key will be Directly Store in the Hash table in the respective Index without any mathematical Computation.
  - the W.C Time complexity is O(1)
  - if one key have lager size i.e key = 2^{1000}, 9999 then we need to maintain the large size Hash table.

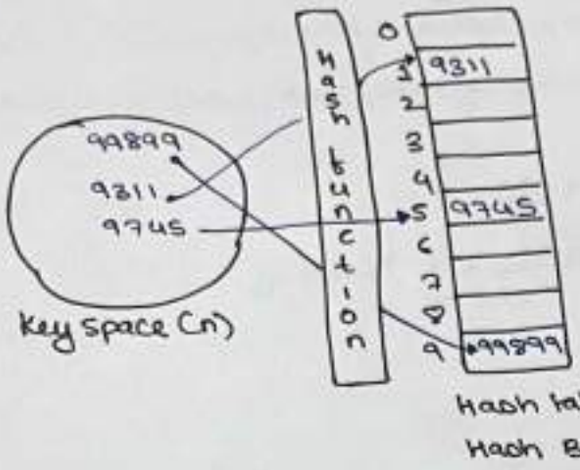


- even though empty space available in the Hash table. its not efficiently utilized, to avoid this problem., the concept of Hash functn is introduced.

• hash function.

H.F(key) = key mod m

m = 10 (0 ---- 9)



Hash table (m)
Hash Bucket (m)

• By using the concept of Hashfunction we can comfortly store large key
• if Both the key are mapped to the same index then it is called as collision.

• to handle collision, various Collision resolution technique are used.

▪ Various Hash functn.:

(1) Division modulo method:

H.F(key) = key mod m

• Do not choose 'm' value as exactly power of 2', because if $m = 2^k$, then

HF(key) = LSB 'k' bit always

note

• Choose the m value as a prime no. which is not nearer to power of 2. then it will give min. no. of collision

Ex: $m = 8 = 2^3 \neq 2^k; k = 3$

1] key = 86 0 1 0 1 0 1 1 0 — 86 mod 8 = 6

2] key = 102 = 0110 0110 — 102 mod 8 = 6

3] key = 118 = 0111 0110 — 118 mod 8 = 6

4] key = 126 = 0111 1110 — 126 mod 8 = 6

Ex: $m = 7 \neq 2^k \neq 2^3$

1) 86 mod 7 = 2
2) 102 mod 7 = 4
3) 118 mod 7 = 6
4) 126 mod 7 = 0

• mid square method :-

$m = 1000 (0 \cdots 999)$

Ex: $(986)^2 =$ 9 7 2 1 9 6

5 4 3 2 1 0

Index	key
721	986

Hash table(m)

• we choosen 3digit bit only b/c there are largest no of 3digit bit than one digit & two digit.

you can choose any 3digit via

- these is a good hash funct^n
- counter ex^n is difficult to generate.

• Digit Extraction method.

$m = 1000 (0 \cdots 999)$

k = 9 3 4 6 8 1

5 4 3 2 1 0

Index	key
948	934681

Hash table(m)

• its a bad hash funct^n

- Counter example is easy to generate.

• Folding method:

① Shift Boundary method (Fold Boundary method)

key: 123 986 321

```
  123
  321
+ 444
```

Index	key
444	123986321

- its a bad hash funct^n
- counter ex^n is easy to generate.

② Fold Shifting method:

key = 321 123 321

```
  321
  123
  321
+ 765
```

Index	key
765	321123321

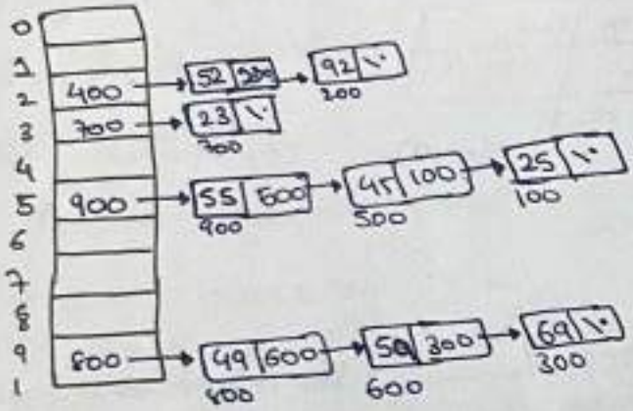
- its a bad hash funct^n
- counter ex^n is easy to generate.

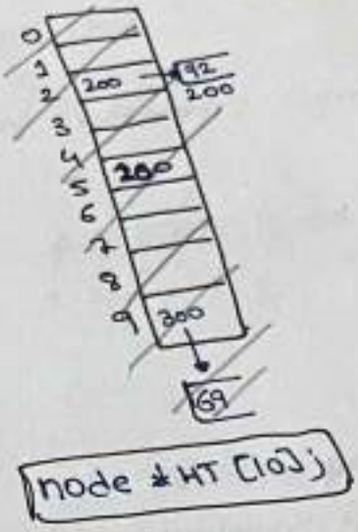
- Collision Resolution Techniques (C.R.T) :-

1. → Chaining (key are store outside)
2. → open addressing (key is store inside.)
   - a. → linear
   - b. → Quadratic } probing
   - c. → Double hashing

- Chaining:
  - chaining is implemented using LinkedList
  - key will be store outside the has table.

key(n): 25, 92, 69, 52, 46, 59, 23, 49, 55





- the main advantage of chaining is that we can handle any no. of collision until the memory is full
- even though empty slot are available in hash table we are using outside space
- the longest chain possible is n keys.

	B.C	WC
Insertion	O(1)	O(1)
deletion	O(1)	O(n)
Searching	O(1)	O(n)

• open Addressing:

↳ Linear probing: $H.F(key) = key \bmod m$ $\quad m = 10 (0 \dots 9)$

$L.P(key, i) = (HF(key) + i) \bmod m$

$i = 0, 1, 2, \dots 9 (m-1)$

Ex: keys (n): 25, 38, 43, 68, 79, 46, 58, 65, 20

1) L.P(25,0) = 5+0 = 5
2) LP(38,0) = 8+0 = 8
3) LP(43,0) = 3+0 = 3
4) LP(68,0) = 8+0 = 8, 8+1 = 9 } 2
5) LP(79,0) = 9+0 = 9, 9+1 = 10%.10 = 0 } 2
6) LP(46,0) = 6+0 = 6
7) LP(58,0) = 8+0 = 8, 8+1 = 9, 8+2 = 10%.10 = 0, 8+3 = 11%.10 = 1 } 3
8) LP(65,0) = 5+0 = 5, 5+1 = 6, 5+2 = 7 } 2
9) LP(20,0) = 0+0 = 0, 0+1 = 1, 0+2 = 2 } 2

// Total no. of collision : 9 //.

79	0
58	1
20	2
43	3
	4
25	5
46	6
65	7
38	8
68	9

Hash table.

• Load factor (α) :-

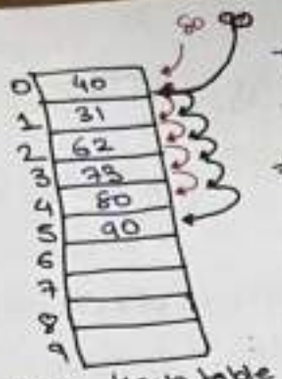
the no. of keys getting Store in oneslot is know as load factor.

In "m" Slot ⟶ "n" keys are Stored.

In 1 Slot ⟶ ?

$\frac{1}{m} * n = \frac{n}{m} \Rightarrow \boxed{\alpha = \frac{n}{m}}$

Ex: Keys(n) = 62, 31, 40, 73, 80, 90



Hash table

Total no. of collision is (4) //

* primary clustering.

- primary clustering.
- if the two keys are mapped on to same starting location in the hash table then both follow the same path in the linear manner & b/c of that search Time complixity will increase
- to avoid this problem we will use Quadratic probing.

note

- deletion will create problem to other keys but we can manage by storing some special symbol like "#"

	B.C	W.C
Insertion	O(1)	O(m)
deletion	O(1)	O(m)
Searching	O(1)	O(m)

- if more no. of deletion occur then perform re-hashing

* Quadratic probing: Simple term : $(H.F(key) + c_2 * i^2) \bmod m$

$$Q.P(Key, i) = (H.F(Key) + c_1 * i + c_2 * i^2) \bmod m$$

$c_1 = 1$, $c_2 = 1$ $\quad i = 0, 1, 2, \ldots, q(m-1)$

Ex: key(n) = 25, 98, 57, 75, 97, 18, 78, 46.

1) Q.P(25, 0) ⇒ 5+0+0 = 5
2) Q.P(98, 0) ⇒ 8+0+0 = 8
3) Q.P(57, 0) ⇒ 7+0+0 = 7
4) Q.P(75, 0) → 5+0+0 = 5
   5+1*1+1*1² = 5+1+1 = 7
   5+1*2+1*2² = 5+2+4 = 11 % 10 = 1
5) Q.P(97, 0) = 7+0+0 = 7
   = 7+1+1 = 9

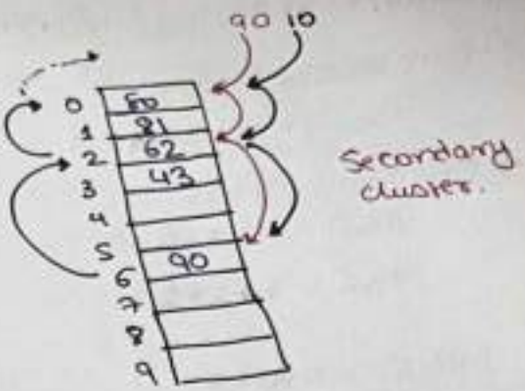
0	18
1	75
2	
3	
4	78
5	25
6	46
7	57
8	98
9	97

Total no. of collisions (6) //

⑥ keys(n) = 62, 81, 43, 80, 90, 10

Q. P(90,0) = 0+0+0 = 0
(90,1) = 0+1+1 = 2
(90,2) = 0+2+4 = 6

Q. P(10,0) = 0+0+0 = 0
(10,1) = 0+1+1 = 2
(10,2) = 0+2+4 = 6
(10,3) = 0+3+9 = 12 % 10
= 2
(10,4) = 0+4+16 = 20 % 10
= 0
⋮



- Secondary clustering.
- if the two key are mapped on to same starting location in the hash table then they both follow same path in the quadratic manner. b/c of these search time complexity will increase.
- to avoid this problem we will use Double hashing.

	B.C	W.C
Insertion	O(1)	O(m)
deletion	O(1)	O(m)
Searching	O(1)	O(m)

- Double hashing.

$H.F_1(Key) = k \bmod m$

$H.F_2(Key) = (k \bmod m-2)+1.$

$D.H(Key, i) = (HF_1(Key) + i * H.F_2(Key)) \bmod m$ $\quad i = 0,1,2,\cdots 9(m-1)$

	B.C	W.C
InBertion	O(1)	O(m)
Deletion	O(1)	O(m)
Searching	O(1)	O(m)

Q] Keys(n) = 25, 98, 57, 75, 97, 18, 78, 65, 46.

1) DH(75,0) = 5+0=5
(75,1) = 5+1*2=9 } 2

→ DH(97,0) = 7+0=7
(97,1) = 7+1*2=9
(97,2) = 7+2*2=11%10
=1 } 2

→ DH(18,0) = 8+0=8
(18,1) = 8+1*3=11%10
=1
(18,2) = 8+2*3=14%10
=4 } 2

→ DH(78,0) = 8+0=8
(78,1) = 8+1*7=15%10
=5 } 2

→ DH(78,2) = 8+2*7=22%10
=2

→ DH(65,0) = 5+0=5
(65,1) = 5+1*2=7
(65,2) = 5+2*2=9
(65,3) = 5+3*2=11%10
=1
(65,4) = 5+4*2=13%10
=3 } 4

→ DH(46,0) = 6+0=6

0	
1	97
2	78
3	65
4	18
5	25
6	46
7	57
8	98
9	75

Hash table.

Total no. of collision:- 11

Note

- If n≤m, then by using perfect hashing we can achive worst search time complexity as "O(1)". (universal hashing (no collision)).

- The expected no. of probes in an unsuccessful search of open addressing technique is.

$\frac{1}{1-\alpha}$, where 'α' is load factor.

- the expected no. of probes in successful search of open addressing technique is

$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ where 'α' is load factor

$\alpha = n/m$

- In a hash-table with chaining & uniform hashing, the expected no. of comparisions in an unsuccessful search is given by α, $\alpha = \frac{n(\text{record})}{n(\text{slot})}$