classmate
Date
Page

DSA

* Steps for program development & execution
  1.> edit
  2.> compile
  3.> linking
  4.> loading
  5.> execution

Edit:

first.c

```
int div (int a, int b)
{
int c;
c = a/b;
return (c);
}
main()
{
int x=10, y=5, z;
z = div(x,y);
printf("%d", z);
}
```

IDE → Integrated development environment [Turbo C]

Console

Compile converts HLL → LLL

first.obj

first.exe

main	div
div	main
x	✓ ✓

uses defined function

* Compiler v/s Interpreter

- Compiler translates program once
- fast
- Program runs independently for eg. .exe file
- Interpreter every time is translated whenever is executes.
- slow
- Interpreter runs with
  - first.gs → Browser
  - first.gsp → web browser

** C is a compiled language
Java is slower than compiled language

Linking: Linking of a library

stdio.lib
[Having machine code]

include <stdio.h>
[Having Prototype]

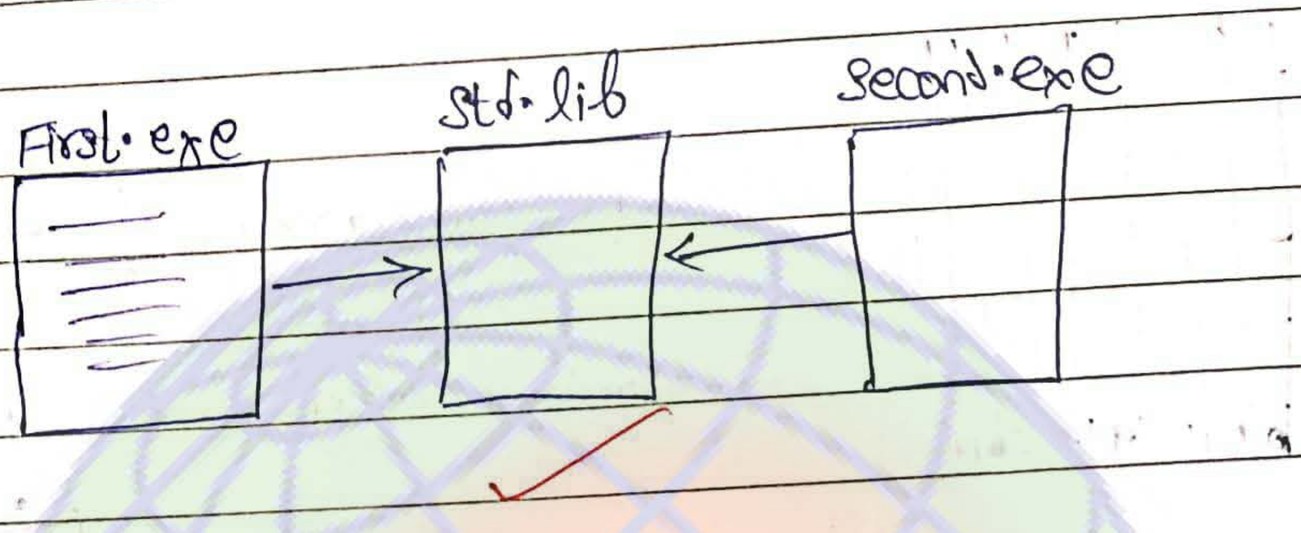
first.obj
first.exe
dir
main

★ (1) Static library
(2) Statically linked library
(3) Dynamically linked library

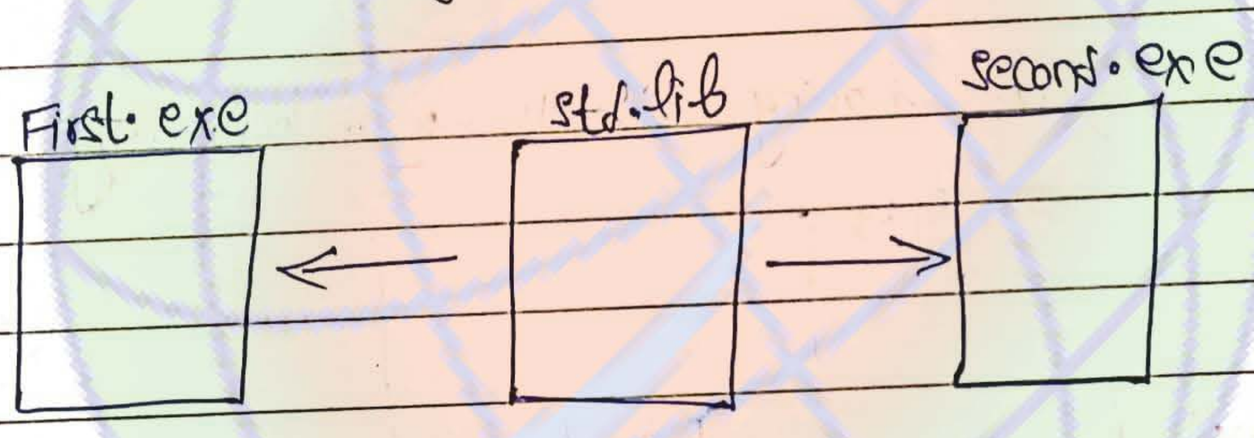
(1) Static
- Everything from the library
- Program can run independently.

Linked library

- There is no copy of the library
- But the program is dependent on the library



linked library

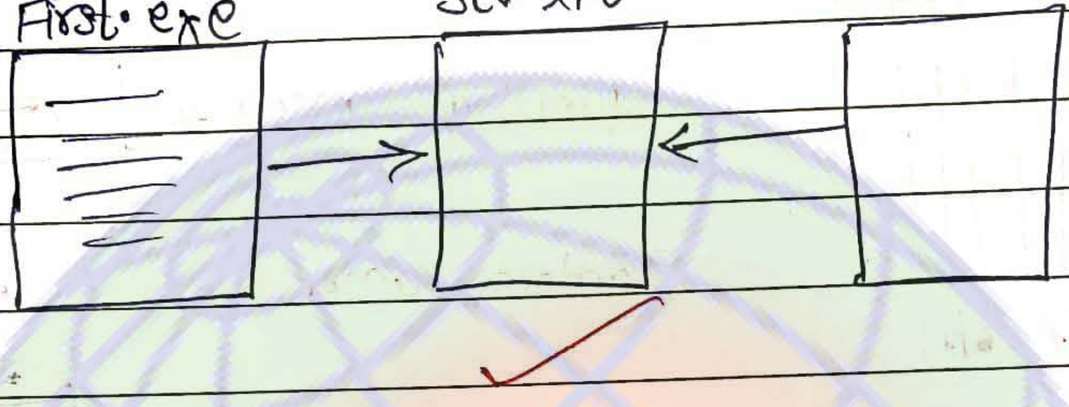


Static library

Statically	Dynamically
• edit, compile, linking, loading { library is attached to the program before execution	• Execution i.e. Run time DLL [Dynamically link library] { library is attached to the program during DLL }
• Anything happening before execution	• with the help of this we can achieve multi-tasking.
• Compiler design time [for int takes 2 bytes]	

Loading: The memory is divided into smallest unit. i.e. byte called as cell & size is 1 byte.

R$ Linked library
- There is no copy of the library
- But the program is dependent on the library



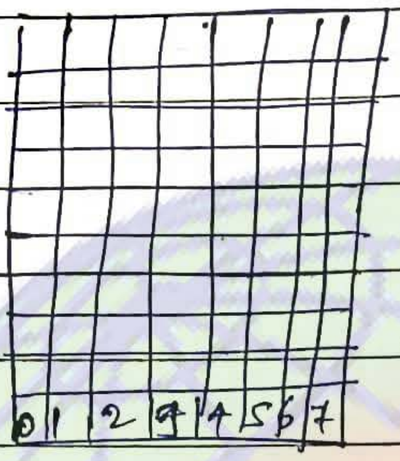
linked library

Static library

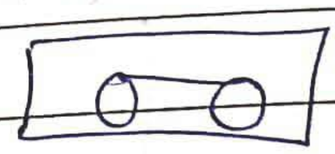
Statically	Dynamically
• edit, compile, linking, loading { library is attached to the program before execution	• Execution i.e. Run time DLL [Dynamically link library] { library is attached to the program during DLL }
• Anything happening before execution	• with the help of this we can achiere multi-tasking.
• compiler design time [for int takes 2 byte]	

Loading: • The memory is divided into smallest unit. i.e. byte called as cell & size of 1 byte.

- Address are linear
- Every byte is having it's address
- Every byte is accessed with the help of it's address

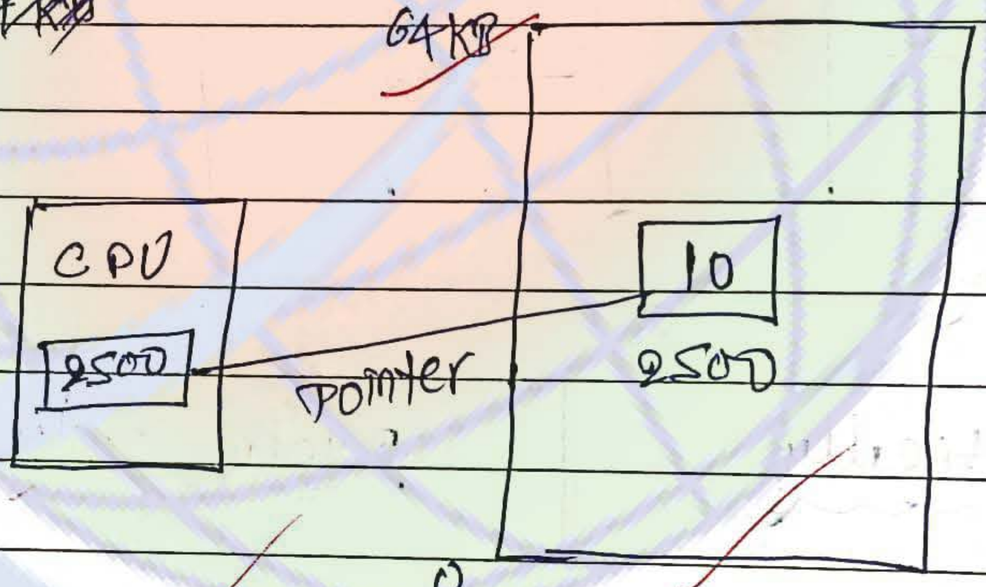


Sequential access →



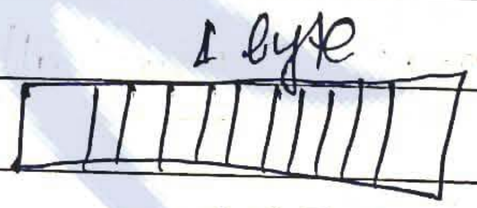
Random access →

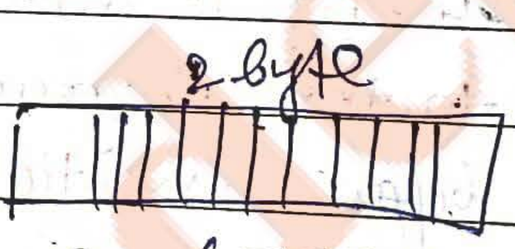
Earlier, 8085 µp having 64 KB memory, why



Sol^n

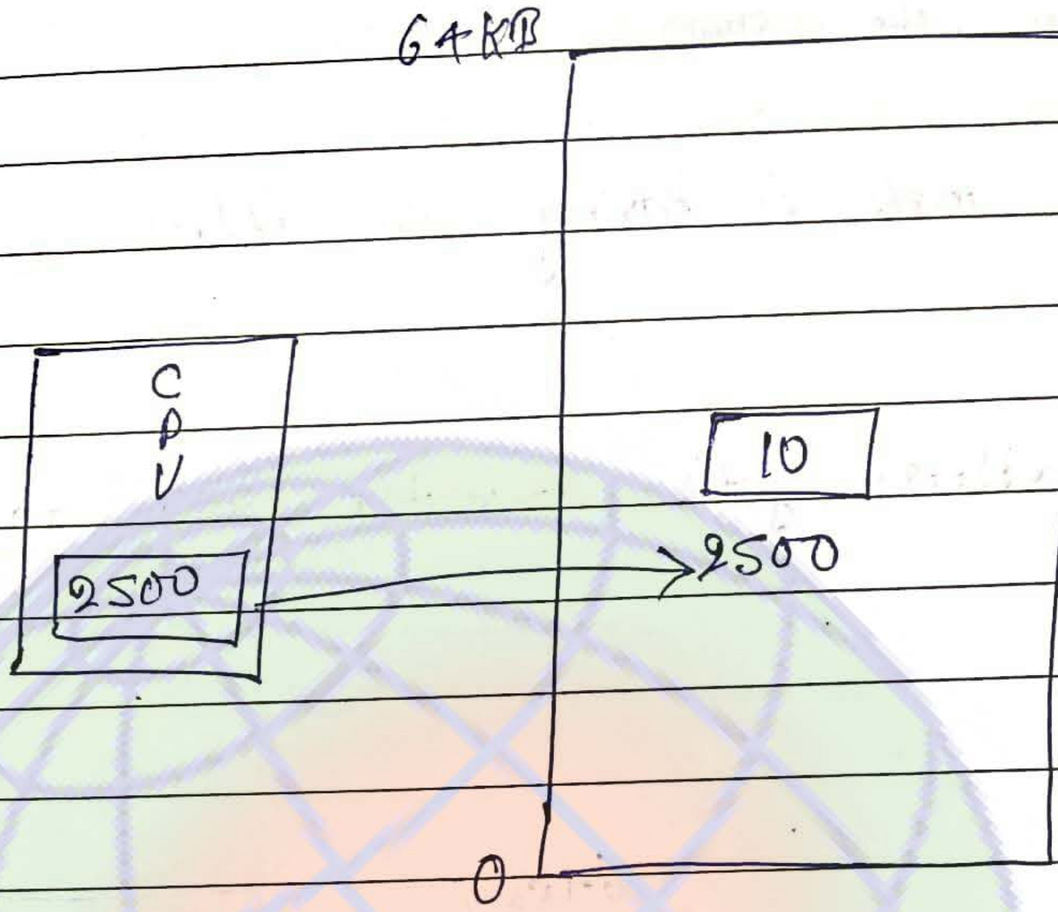
16
2



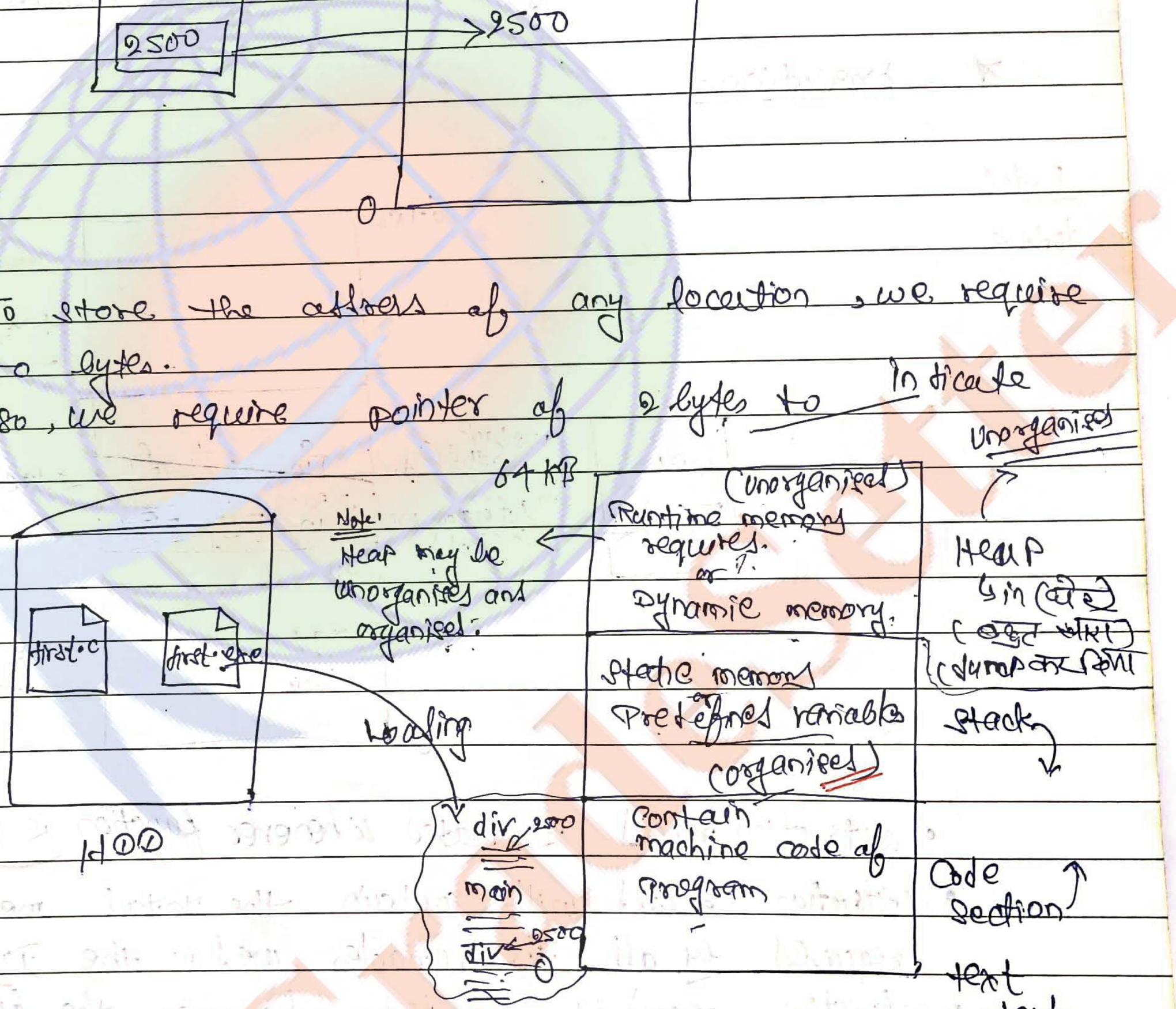


- To store the address of any location we need 2 Bytes.
  i.e. pointer of 2 Bytes from 0-64 kilobytes.
- when a program is loaded into main memory every instruction is having it's address.
- At the place of function call writing the address of function at the place of function call & calling

Binding [Relocation]
- so, relocatable address areas then converted into absolute address



To store the address of any location, we require zero bytes.
So, we require pointer of 2 bytes to indicate 64KB



So, main memory is divided into three major sections -

(a) Code section.
(b) Stack section.
(c) Heap section.

Relocation = ?

(1) when the pro

(2) every inst is having the address in main memory.

* Relative addres gets converted into absolute addres.

* Execution :-

First C

#include <

- activation record is created whenever function is called.
- Activation record will contain the total memory requires by all the variables used in the program.
- activation record is created whenever the function is called
- Activation record is delocated when the function terminate.
- Topmost activation record is belong's to currently executing function.

- stack of activation record's or stack frames.
- one function can not access the activation record of another function.

(*) (1) main()
{
A();
B();
C();
}

A()
{
}

B()
{
}

C()
{
}

⇒ Total activation records creates = 4
⇒ maximum activat record at a time = 2

{ main
A/B/C }

(2) main()
{
A();
}

A()
{
B();
}

B()
{
C();
}

C()
{
}

⇒ Total Activ records created = 4
maximum " " at a time = 4

Note: maximum depends on maximum chain of calls or maximum depth of calls.

2

3.7

```
main()
{
1 —
2 —
3 fun();
6 —
}
```

```
fun()
{
4 —
5 —
}
```

★ Activation record :-

(1) It will contain the memory for parameter

fun(int a, int b) ← (1)

(2) It " " " " " local variables

```
fun(      )
{
int c; ← (2)
```

(3) Contain values can be manages in two ways.

(1) Immediate access

mov B, 10

(2) Direct access

```
LXI  500
mov  B, M
```

M
10
500

C = 2*a + 3*b ;
(constants)

So,

(3) It will contain the memory for constants.

(4) It will contain the memory for result of expression

(5) " " " " " return address

dir
main
dir

Activation record

(1) Parameter
(2) Local variables
(3) constants
(4) result of expression
(5) Return address.

Note:

```
(1)
fun ( int a, int b )
{
(2)  int c;
     c = 2*a + 3*b;
                  (4)  (3)
     return(c);
}
```

Q.) main ( )
{
int x, y, Z ;
— —
≡
}

This is static variables but, the memory allocation is dynamic in nature.

Note: memory is not allocated at compile time.

★ Attributes of variable:-

static { 1.> Name — name
2.> Type/size — int, float.
3.> value → garbage value, initialize
always dynamic → 4.> Address → memory addr
static { 5.> scope → main, fn → यहाँ पर accessible है
6.> lifetime/extent → life time - how long यह is alive

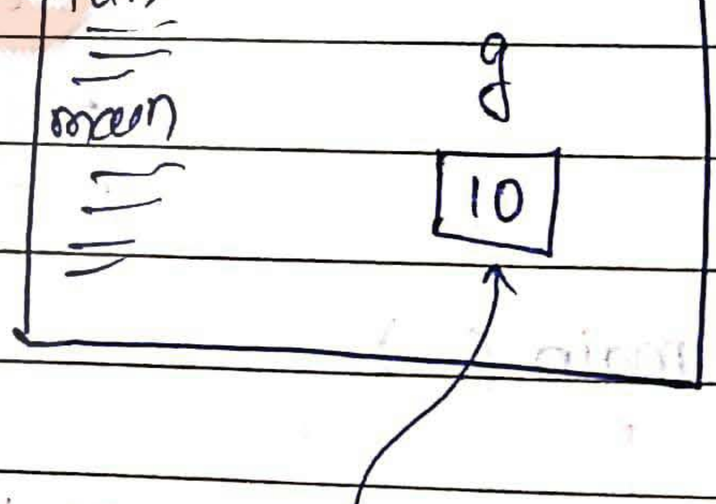
★ methods of memory allocation:-

1.> static
2.> Stack dynamic ← we studies in our previous page,
3.> Heap dynamic

(1) Static methods of memory allocation:-

global variable → int g = 10; ← static memory allocation

```
fun()
{ int x, y;
  Pf(g);  ——— 12
  g++;
}
main()
{ int a, b;
  g = g+2;
  Pf(g);  ——— 12
  fun();
  Pf(g);  ——— 17
}
```



• main is allocates at the loading time in the code section. This is static memory allocation.

• 'C' language supports static method of memory allocation for

(a) Global variables
(b) static variable.

```
fun()
{
static int x=10;
x++;
pf(x);
}
main()
{
fun();   —11
fun();   —12
}
```

x
[~~45~~ 12]
↑ static के case में ये code section में होगा, यानि delete नहीं होगा।

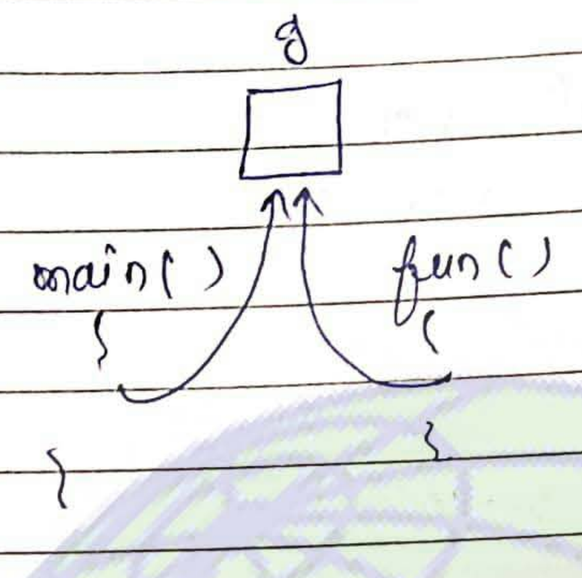
Note: (a) fun()
```
{
int x=10;
x++;
pf(x);
}
main()
{
fun();   —11
fun();   —11
}
```

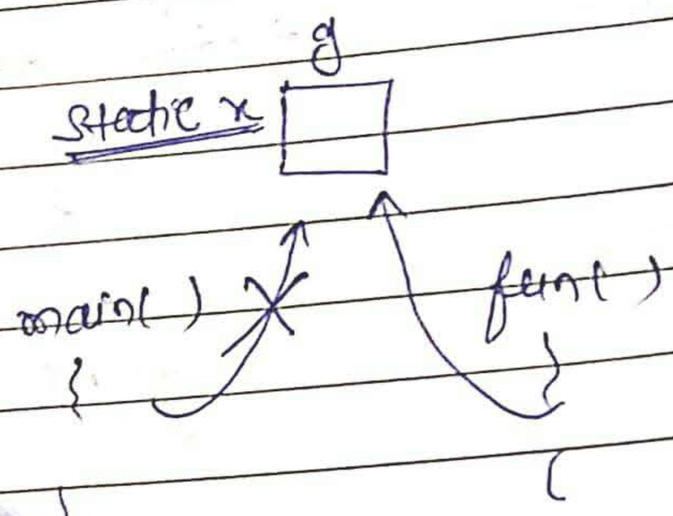
(b)

Global	static
code section same loading time	• loading time
diff • for all • accessible by all functions	• only for local • accessible by only one function.

बहुत important है इस बात का ध्यान रखें।

garbage value = Any value





Note:

- The memory which are allocated in code section is automatically initialize to zero(0).
- Code section = initialize to zero ← बकी
  Stack section = garbage value.

global and static variables of C are initialize to zero.//

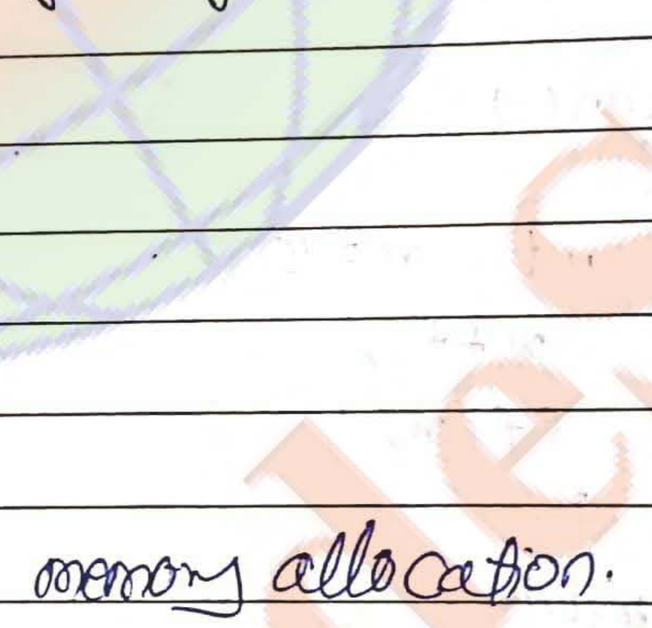
(3) Heap method of memory allocation.

```
main()
{
int *p, q;
p = (int *) malloc (2)
*p = 10;
```

fun(n) (Indirect access)
↑ These is used to allocate heap in memory.

(direct access) →

pointer uses for storing address of data.



```
main()
{
int *p, a;
cout<<"enter size";
cin>>a;
p=(int *) malloc (a);
*p=10;
```

* Heap management

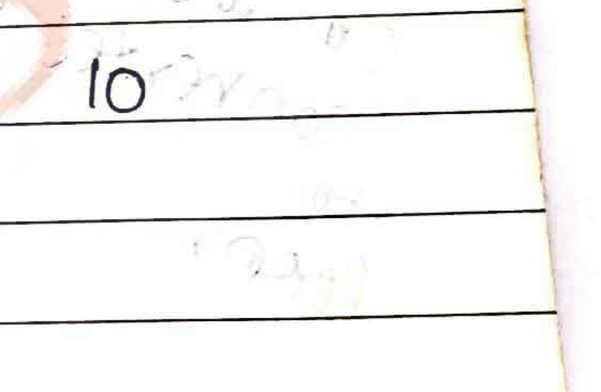
```
p= malloc(n);   ← to allocate
free(p);        ← to deallocate/release
```

Heap memory is managed by two list
malloc ← when we allocate the memory
free ← when we free the memory.

free list - ~~1 2 3 4 5 6~~ 7 8 9 10

used list - 1, 2, 3, 4 , 5, 6

```
main
{
p = malloc(4);
q = malloc(2);
{
free(p);
p=NULL;
```



Note:
Every program has it's own code section, stack and Heap.

Recursion:-

90% question of C are either (1) recursion or (2) pointer.

Tail Recursion: → tail में function calling अंत में होता है।

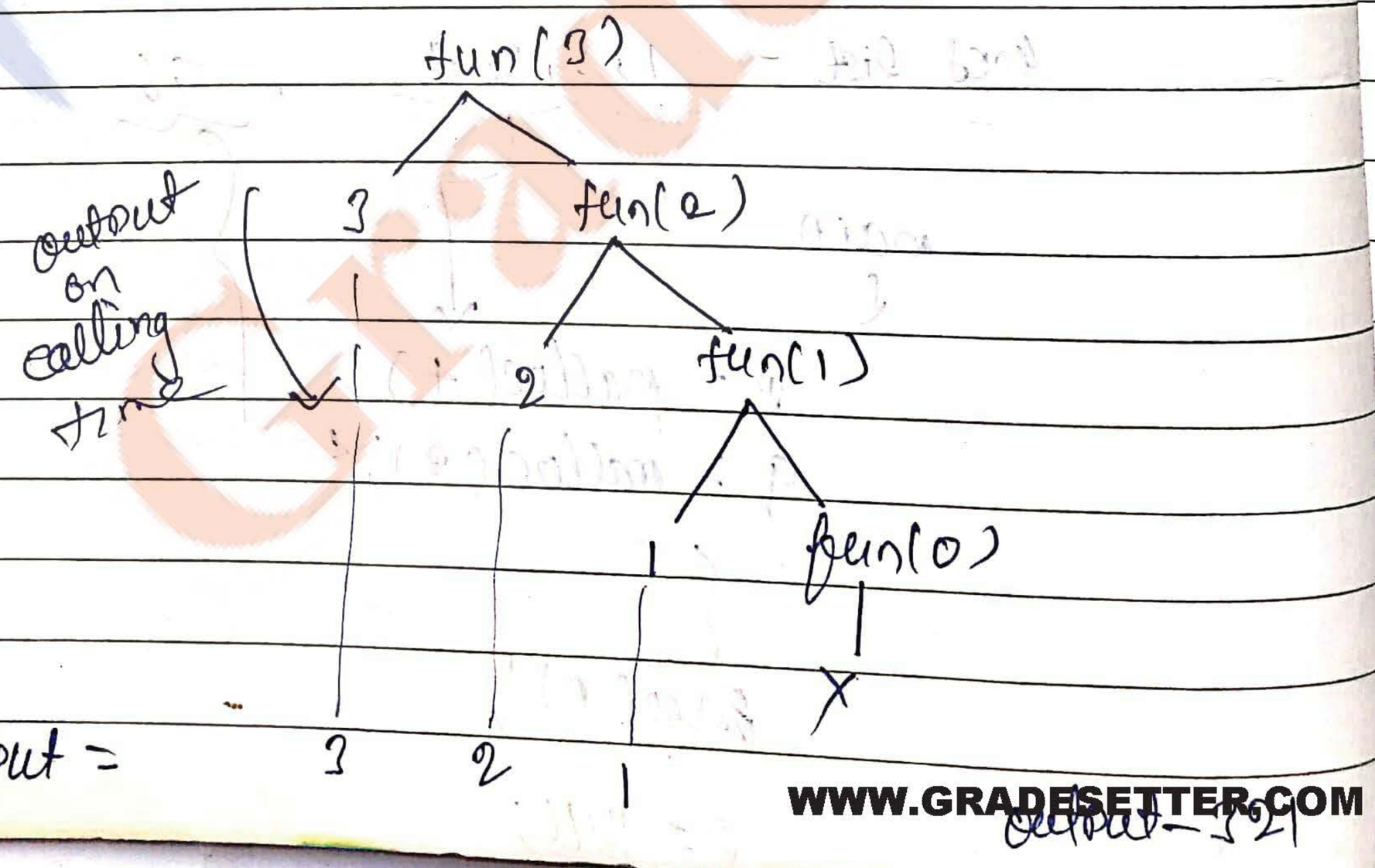
```
void fun(int n)
{
  if(n>0) ← Base condition or termination condition.
  {
    Printf("%d", n);
    fun(n-1);
  }
}

main()
{
  int a=3;
  fun(a);
}
```

1)
2)
3)

```
n=3
while(n>0)
{ Pf(n)
  n--;
}
```

tail recursion can be easily converted to loop.

output on calling time

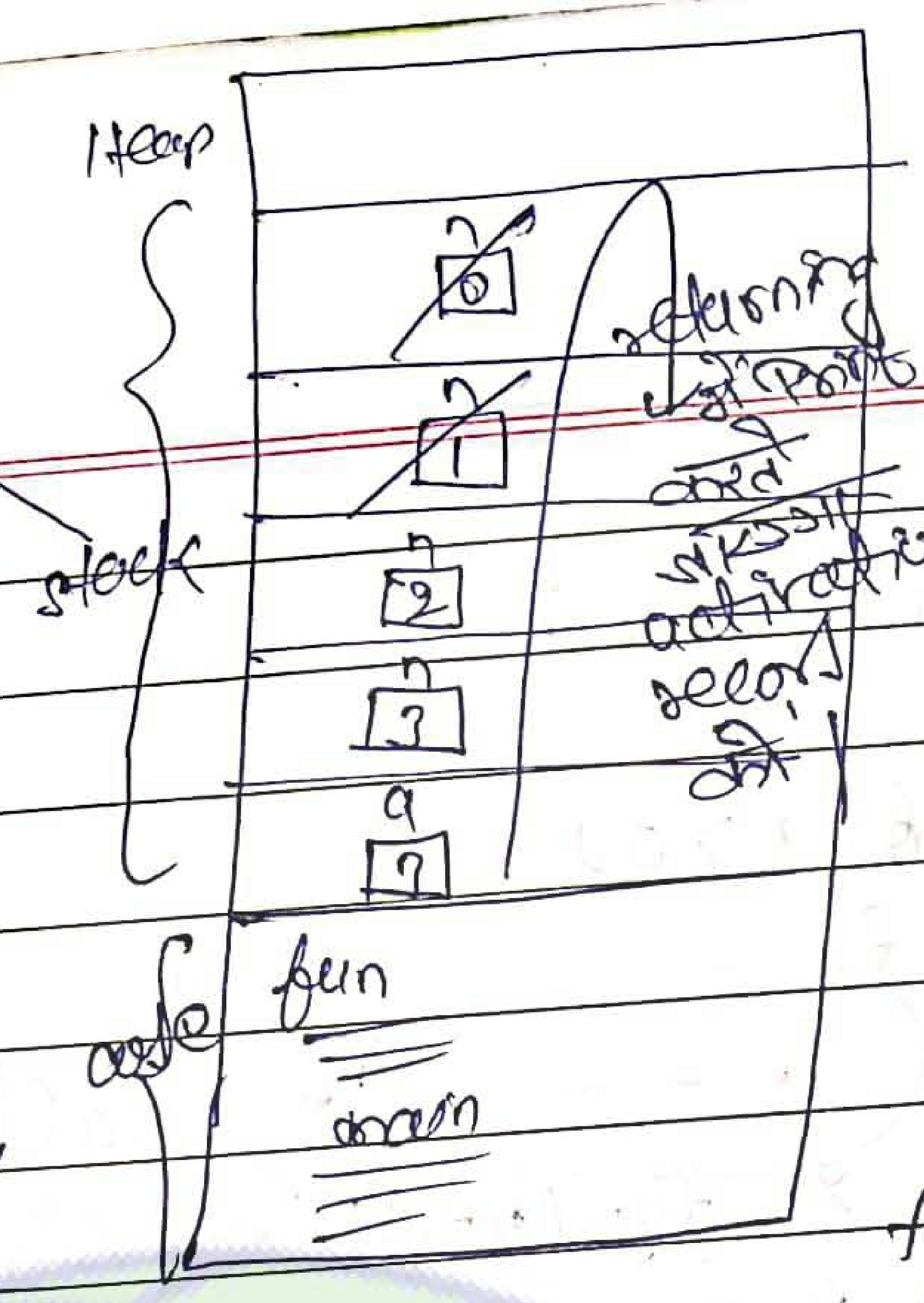


Output = 3 2 1

Output - 321

Note: for supporting recursion stack is introduced.

n=3
while(n>0)
{
}

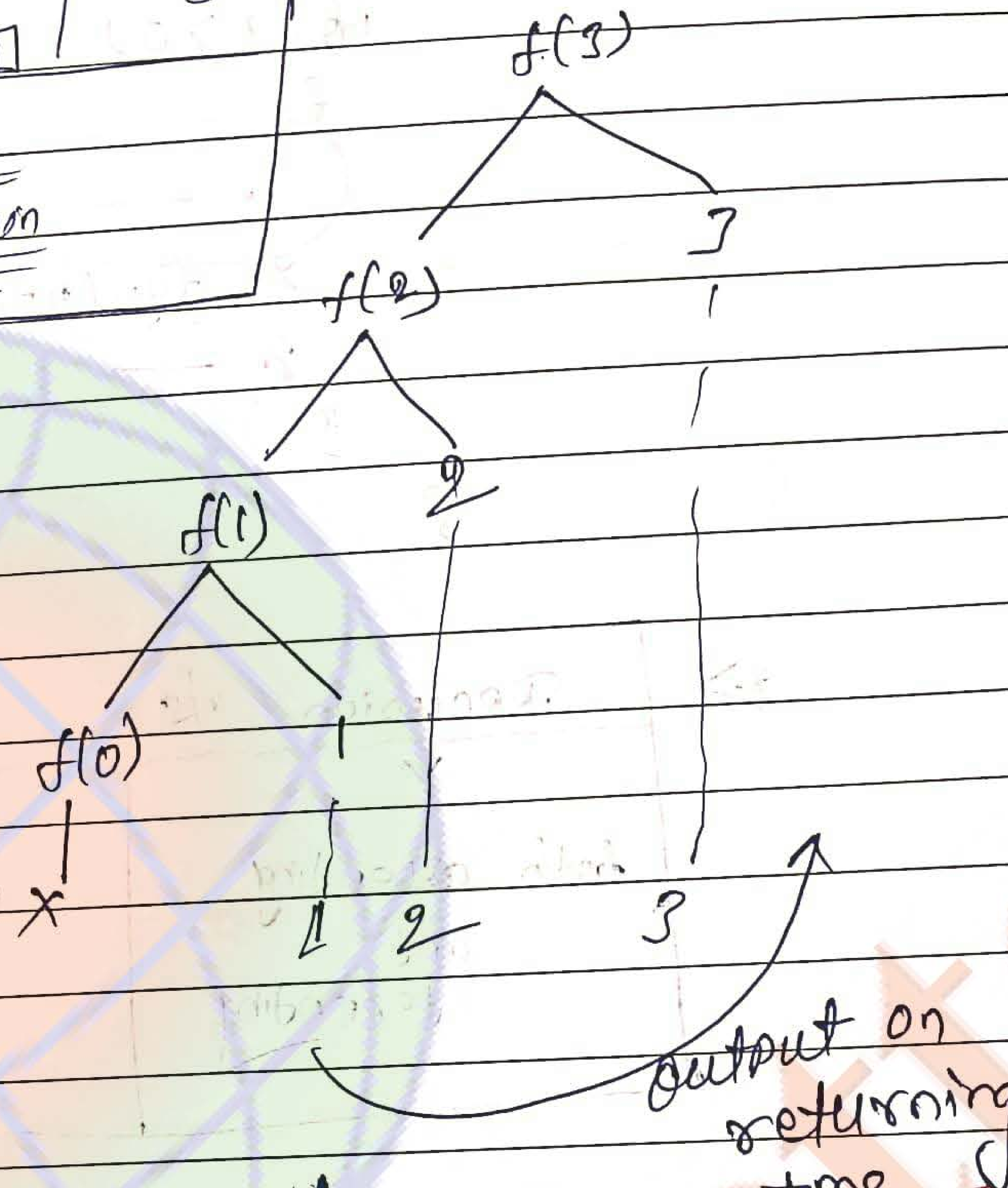


Head Recursion

```
fun (int n)
{
    if (n>0)
    {
        fun(n-1);
        pf(n);
    }
}
main()
{
    int a=3;
    fun(a);
}
```

2.> function calling पहले होगा। print returning time पे होगा

using method recursion concept



O/P:-

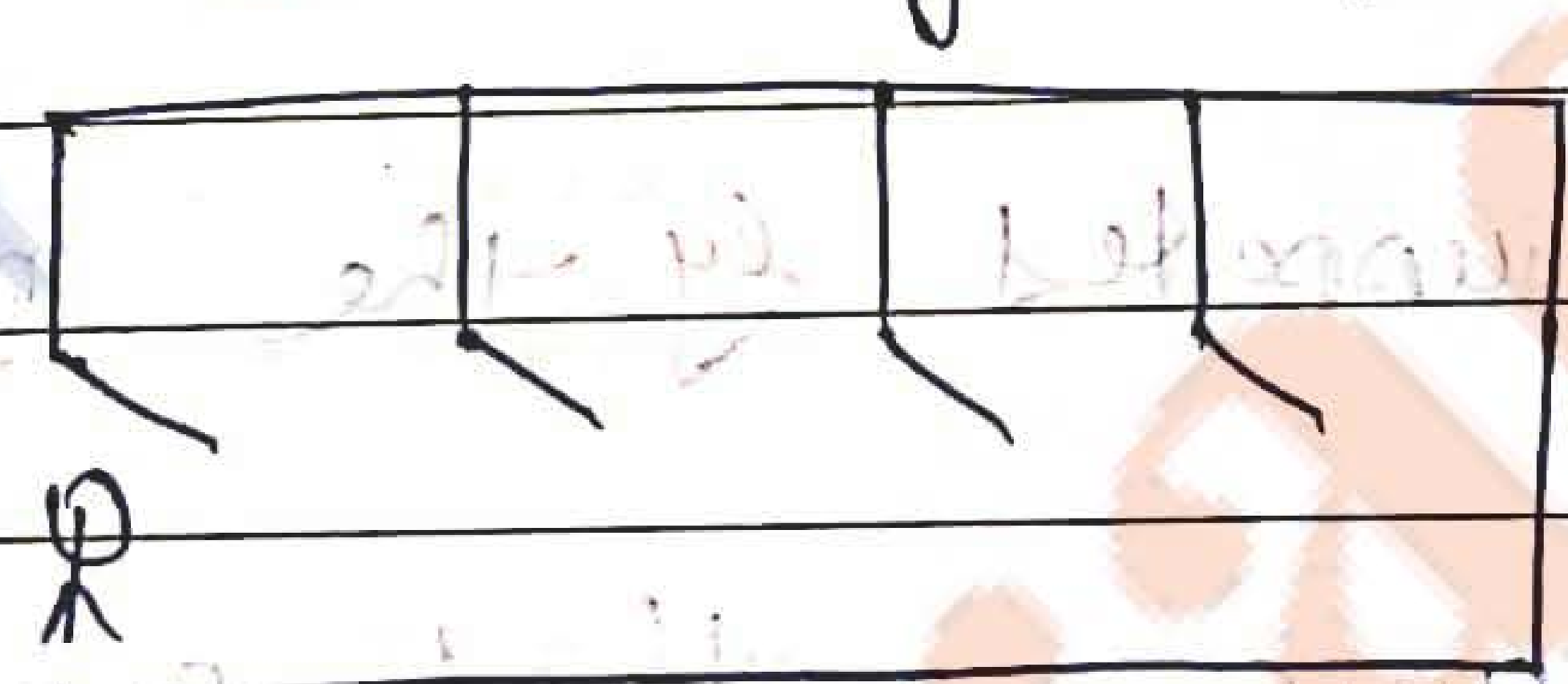
"अगर बाद में Printf होगा तो returning time पे Print होता जाएगा॥

output = 1 2 3

"इसान से पहले function"

V.V.I Note:

1.> Two phases in recursion
(I) calling time / increasing time / ascending
(II) Returning time / decreasing time / descending



1.> switch on bulb
2.> Goto next room
O/P → 1, 2, 3, 4

1.> Goto next room
2.> switch on bulb
O/P → 4, 3, 2, 1

2.> fun(int n)
{
if(n>0)
{
1. ——— ← calling time
2. fun(n-1); ← otherwise returning time
3. ———
}
}

before recursive function.

3.>

Recursion v/s	loop
both ascending and descending	only ascending

4.> recursion and loop are both converted to each other and vice-versa are also true

⇒ Recursion ⇌ loop

5.> Recursion is supported by the mathematics.

6.> Stack dynamic method supports recursion.

7.> In recursion always activation record in stack is created.

8.> mostly in recursion, less recursion is used, because it uses fully

* $n! = 1 * 2 * \dots * n$

$n! = n * (n-1) * \dots * 2 * 1$

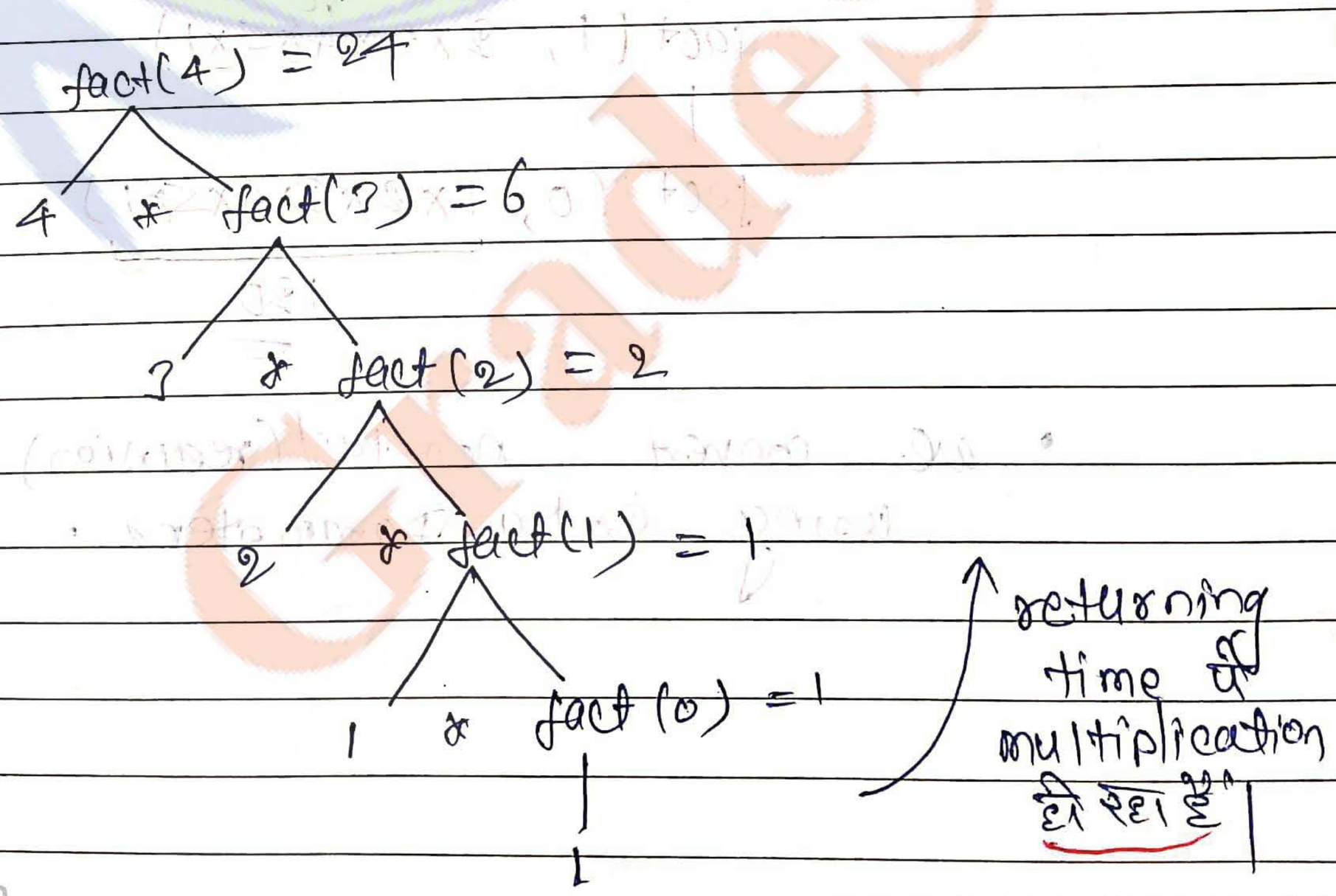
$n! = n * (n-1)!$

$$fact(n) = \begin{cases} 1 & n = 0 \\ n * fact(n-1) & \text{if } n > 0 \end{cases}$$

or

```
int fact (int n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

eg:-



Date
Page

★ Converting recursion into tail recursion using auxiliary variable.

```
int fact (int n, int x)
{
    if (n==0)
        return 1;
    else
        return fact (n-1, n*x);
}
```

fact (5, 1)
|
fact (4, 5x1)
|
fact (3, 4x5x1)
|
fact (2, 3x4x5x1)
|
fact (1, 2x3x4x5x1)
|
fact (0, 1x2x3x4x5x1)

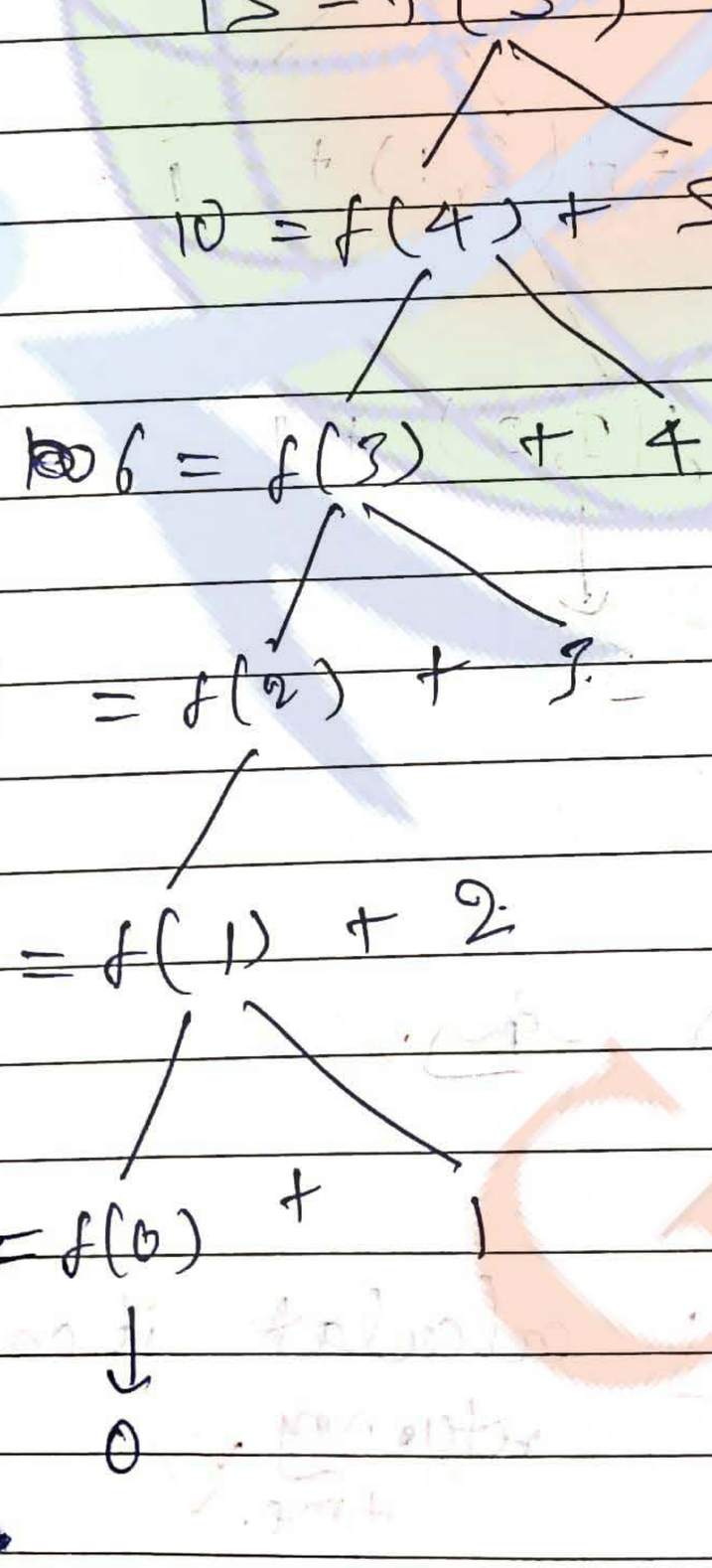
120

- We convert non-tail (recursion) to tail using extra parameters.

Date
Page

*

```
int f(int n)
{
  if(n==0)
  {
    return 0;
  }
  else
  {
    return f(n-1)+n;
  }
}
```



$\frac{n(n+1)}{2}$

(1)

```
int f( int m, int n)
{
if (n=0)
{
return m';
}
else
{
return f(m, n-1)+1';
}
{
```

$G = f(5,1)$ (m, n)

$= f(5,0) + 1$

5

$= 6$

Choose any random number and try to find the number.)

$7 = f(5,2)$

$= f(5,1) + 1$

$= f(5,0) + 1$

5

(2)

$S = f(3,2)$

$= f(3,1) + 1$

$= f(3,0) + 1$

3

$= m + n$ Ans ✓

Note: calculate it on returning time. ✓✓

(2.)

```
int f(int m, int n)
{
    if (m<n)
    {
        return 0;
    }
    else
    {
        return f(m-n, n)+1;
    }
}
```

f(3, 2)

1 = f(3, 2)

0 = f(1, 2) + 1

f(1, 2)

0

integer division

3 = 1.1

2 = 1

(integer division)

m / n

4 = f(4, 1)

3 = f(3, 1) + 1

2 = f(2, 1) + 1

1 = f(1, 1) + 1

f(0, 1) + 1

0

(3) int Trial (int a, int b, int c)
{
if ((a >= b) && (c < b))
return b;
else if (a >= b) return Trial (a, c, b);
else return Trial (b, a, c);
}

middle element.

I = trial (4, 2, 3)
a b c

/

I = trial (4, 3, 2)
a b c

/

middle of 2, 3, 4
i.e 3 Ans

middle = 3 Ans

4) int f(int m, int n)
{
if (m == n)
return m;
if (m > n)
return f(m-n, n);
else
return f(m, n-m);
}

Ans GCD

RI

f(4,5)

f(4,1)

f(3,1)

f(2,1)

f(1,1)

Ans 6, CD

f(27,15) = 3

↓

f(27, 15)

↓

f(27-15, 15)

↓

f(12, 7)

f(9, 7)

f(6, 7)

f(3, 7)

7

eg:) int f(int n)
{
static int r=0;
if (n<=0) return 1;
if (n>2)
{
r=n;
return f(n-2)+2;
}
return f(n-1)+r;
}
f(5);

r=5 8 = f(5) =10 Ans

6 = f(3) + 2 =16+2=18

1 = f(2) + r
11+5 = 16

f(1) + r = 6+5=11
6 5

0 = f(0) + r = 1+5=6

1

RI
NVH
only write r in calling time not write final value

eg6>
```
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[] = {12, 7, 13, 4, 11, 6};
  pf (f(a,6));
}
```

4, 6

f(12, 6) = 15

12 + f(7,5) = 12+3=15

7 - f(13,4) = 7-4=3

13 - f(4,7) = 13-9=4

4 + f(11,2) = 4+5=9

11 - f(6,1) = 11-6=5

6 + f(0,0) 6+0=6

Date
Page

eg:-
```
foo (int n, int r)
{
  if (n>0)
  return n%r + foo (n/r, r);
  else
  return 0;
}
```

foo(345,10); → ?

f(345,10) = 12 (n, r)

Ans = 12

345/10
34

345 + foo(34,10) = 5+7=12
5

34/10
3

34 + foo(3,10) = 4+3=7
4

3/10
0.3

3 + f(0,10) = 3+0=3
3

0

Note:

n%r = remainder

n/r → integer division

eg8.)
```
int f(int j)
{
  static int i=50;
  int k;
  if(i==j)
  {
    printf("something");
    k=f(i);
    return 0;
  }
  else
    return 0;
}
```

f(50) →

j

f(50)

|

something

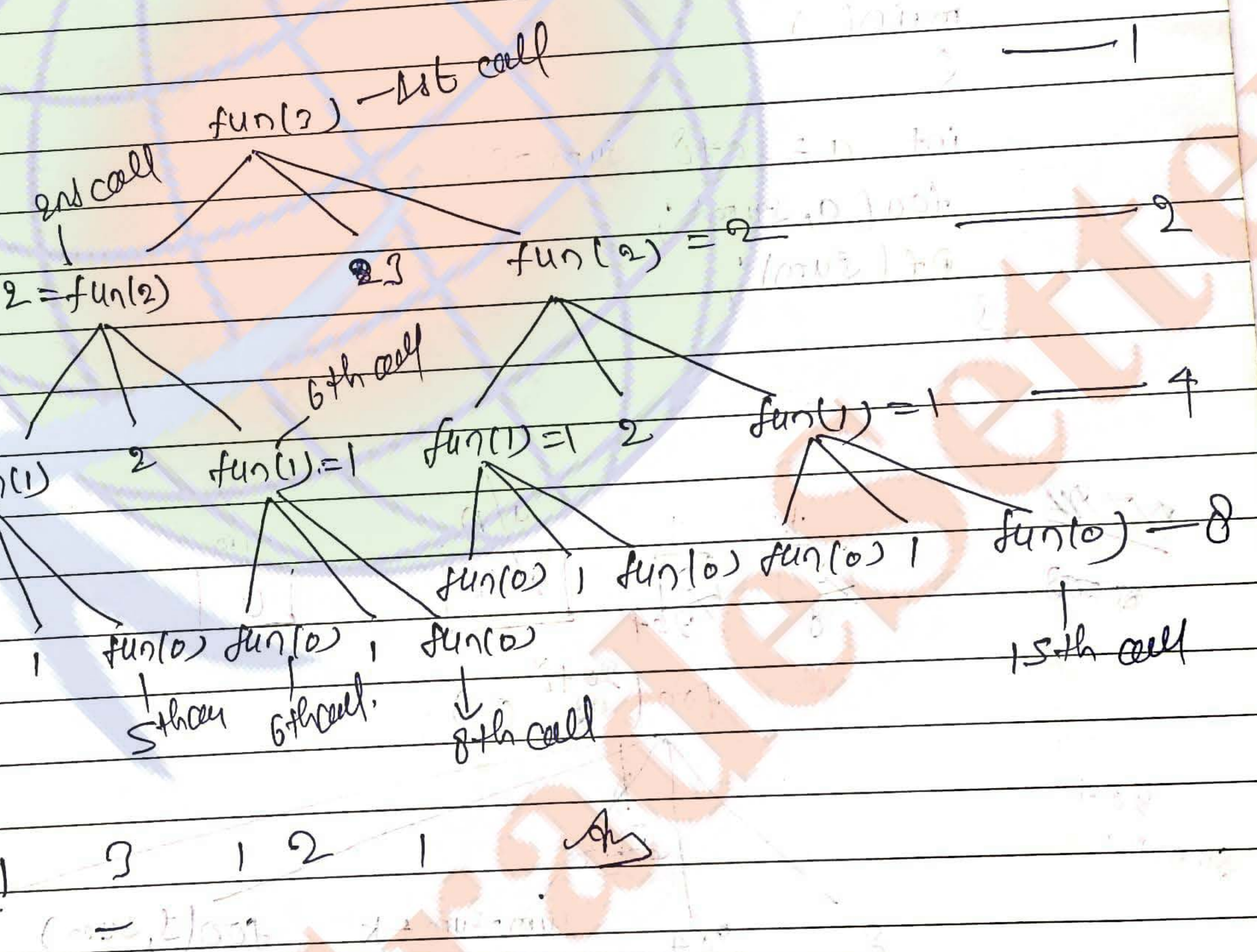
i = 50

k

An infinite loop

4/06/17

eg:-
```
fun (int n)
{
    if (n>0)
    {
        fun(n-1);
        pf(n);
        fun(n-1);
    }
}
fun(3)
```



Ans

No of call = 7

$n = 2$

$n = 3$

$15 = 1+2+2^2+2^3 = 2^4-1 = 16-1 = 15$

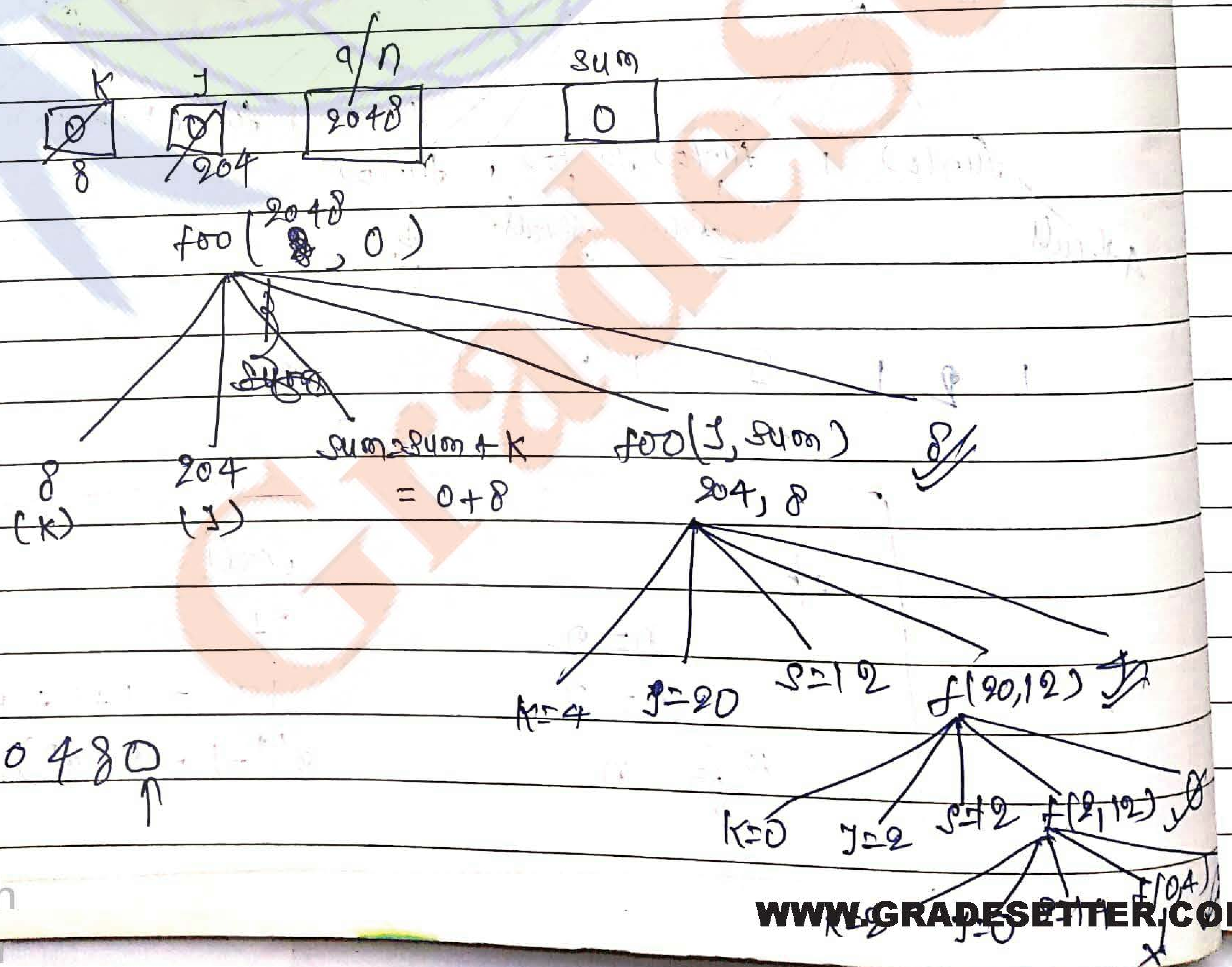
So for n

$2^{n+1}-1 = O(2^n)$

gate Q2.>

```
foo (int n, int sum)
{
  int k=0, j=0;
  if (n==0)
    return;
I.    k = n%10;
II.   j = n/10;
III.  sum = sum+k;
IV.   foo (j, sum);
V.    pf (k);
}
main( )
{
int a = 2048, sum = 0;
foo(a, sum);
pf(sum);
}
```

gate Q3.>



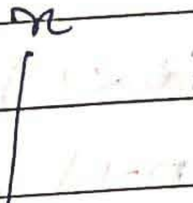
204
sum='

801
Ans 20480

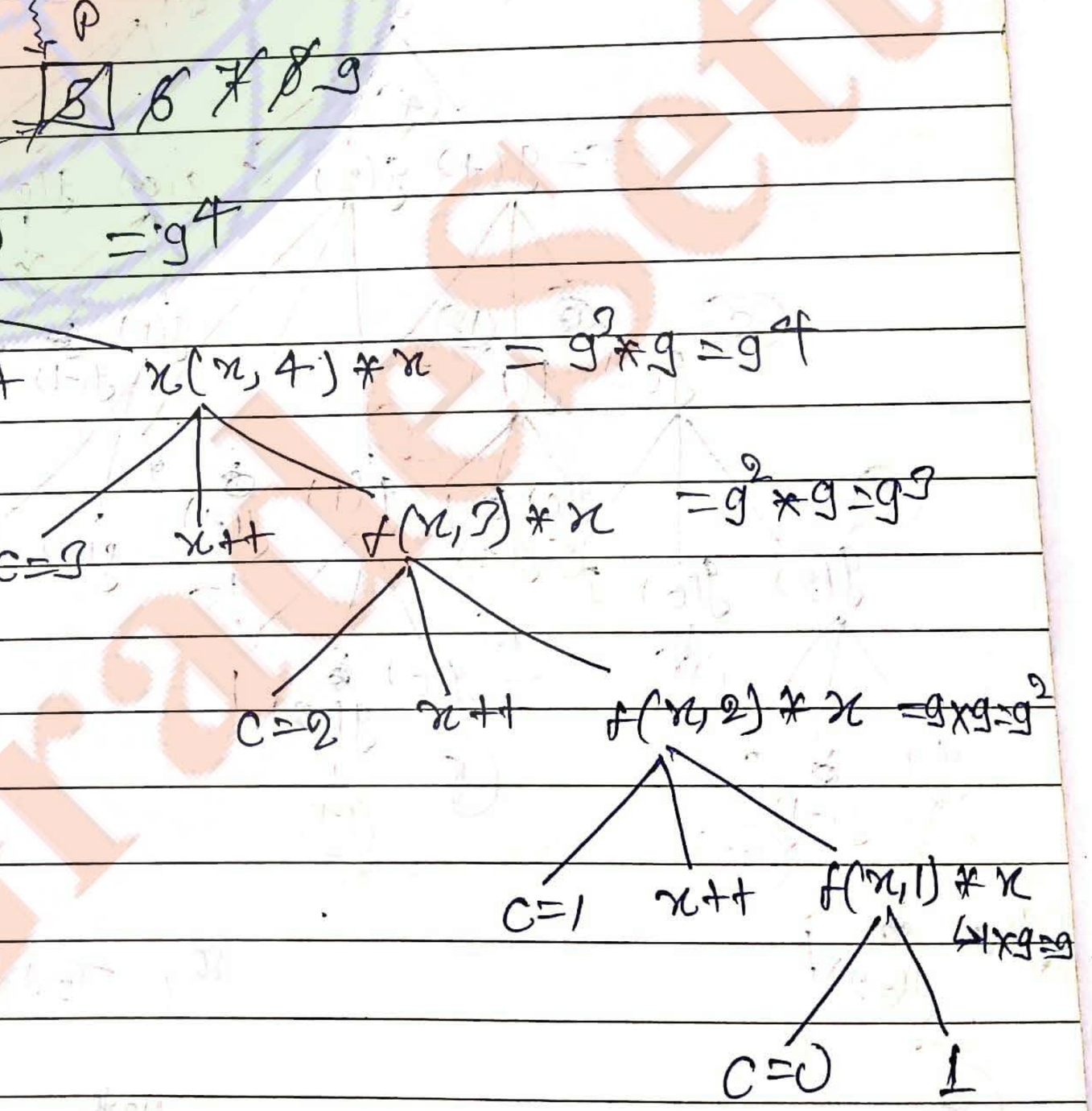
call by value, call by reference,
call by address.

gate

```
eg:> int f(int &x, int c)
     {
       c=c-1;
       if (c==0)
         return 1;
       x=x+1;
       return f(x,c) * x;
     }
     main()
     {
       int p=5;
       f(p,p);
```



reference variable



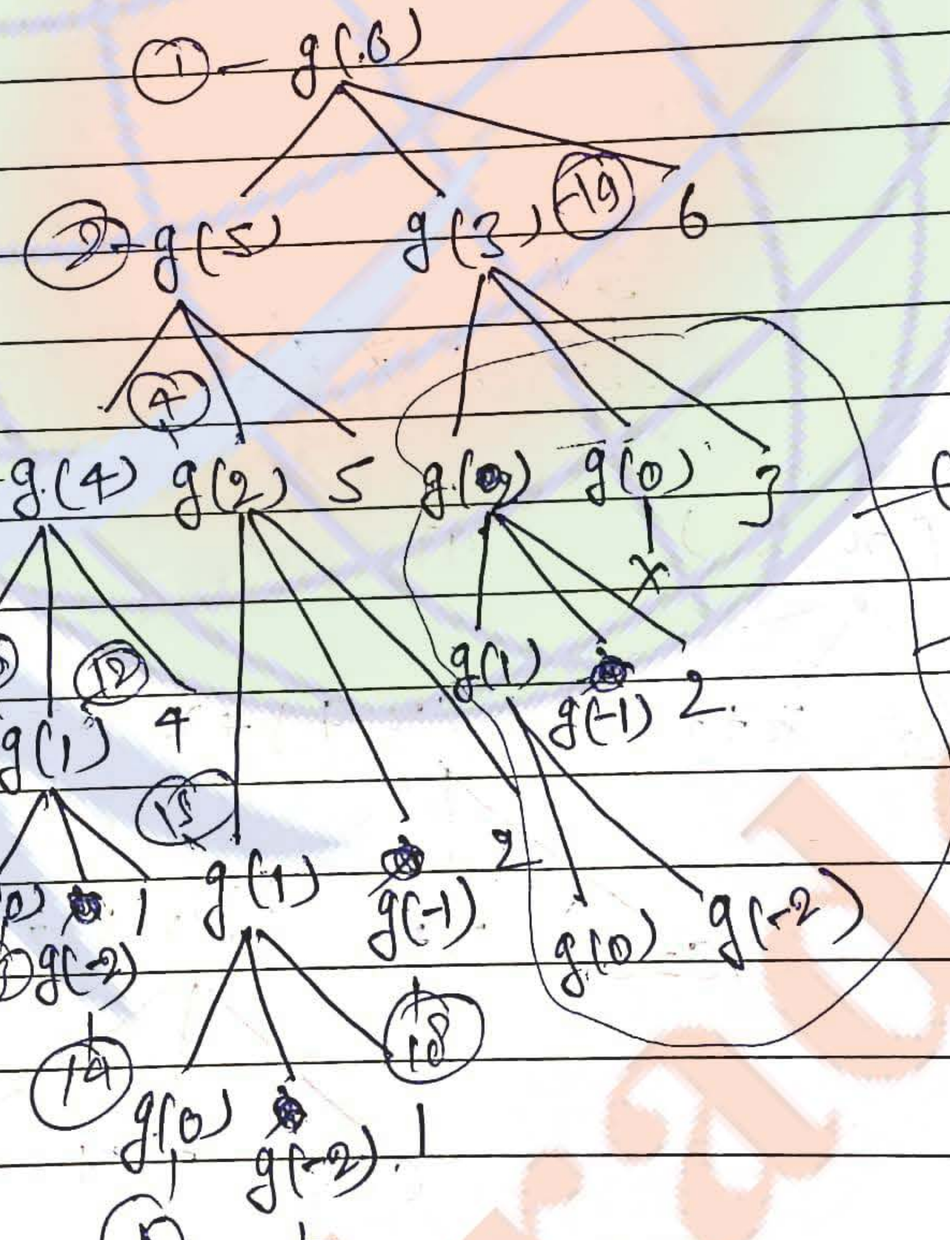
Ans g^4

Date
Page

4) (call by reference) based //

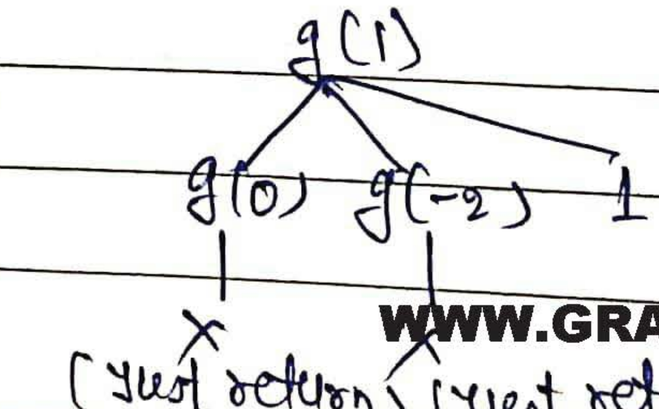
```
void g(int n)
{
  if (n<1) return;
  g(n-1);
  g(n-3);
  Pf(n);
}
g(6)
```

find the no. of calls.



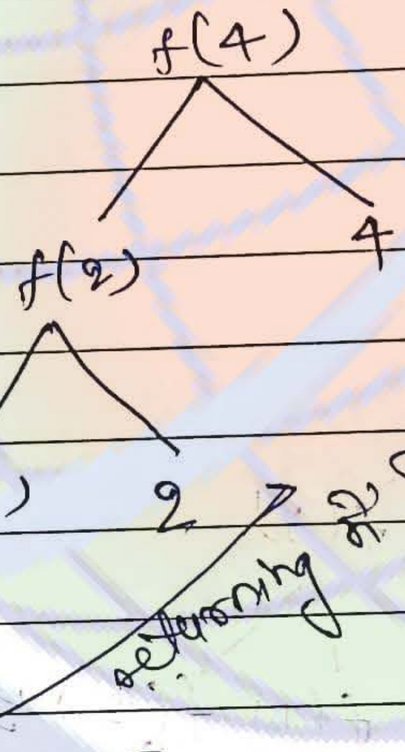
80, Total calls = ~~10+6=20~~ 19+6=25

Note:



eg5:>
```
f(int n)
{
 if (n>0)
 f(n-2);
 printf(n);
}
f(4);
```

Here,
code reading is important



Ans = 0 2 4

eg6:> $fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2) + fib(n-1) & \text{if } n>1 \end{cases}$

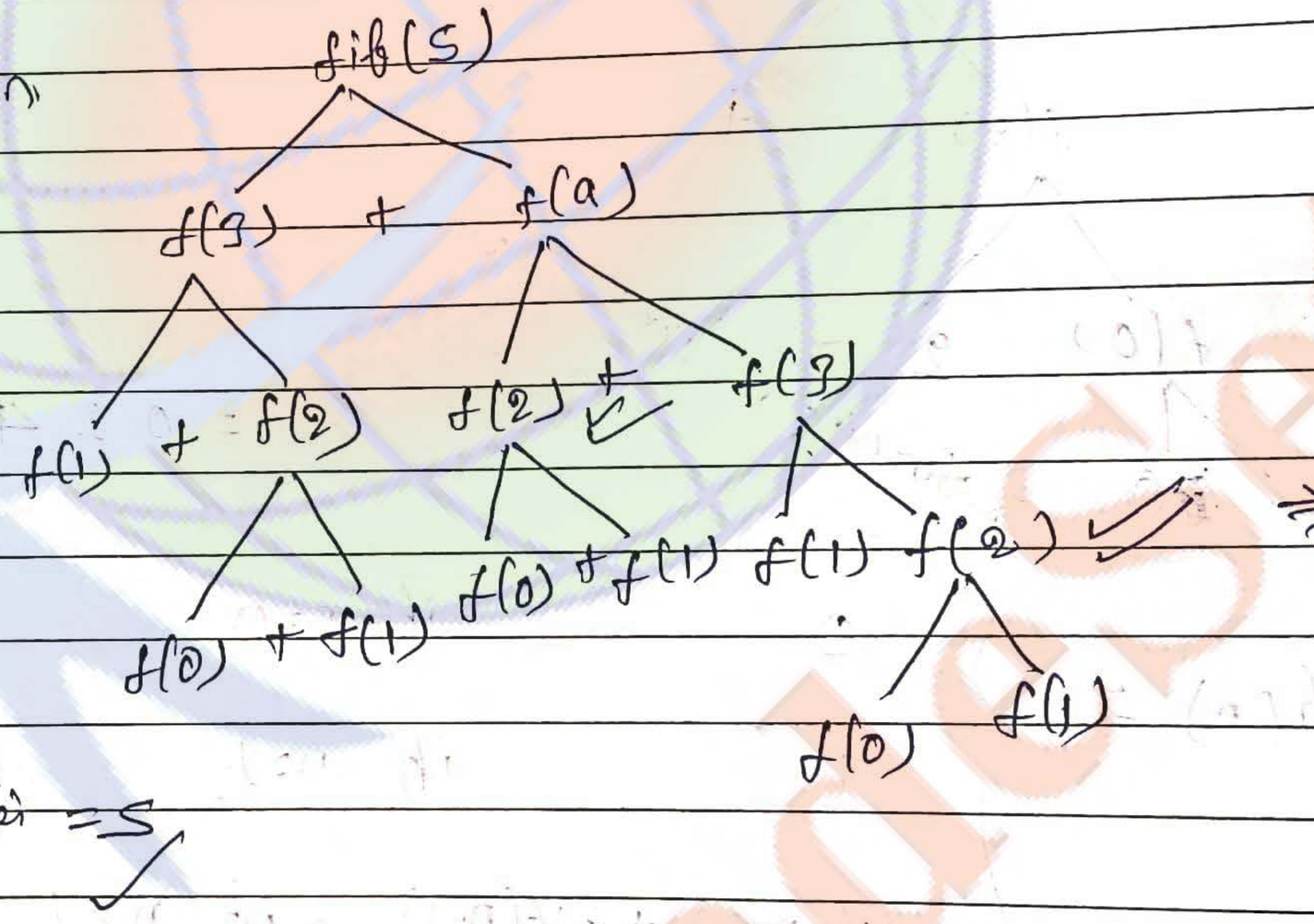
n	0	1	2	3	4	5	6	7	8
fib(n)	0	1	1	2	3	5	8	13	21

first two number

```
int fib (int n)
{
  if (n<=1)
     return n;
  else
     return fib (n-2) + fib(n-1);
}
```

eg a) fib(5),

recursive version



total call = 15 ✓

the maximum height is 5 so, maximum stack size is = 5 ✓

⇒ excessive recursion also tree recursion. ✓

Linear recursion - if calling is only one time ✓

tree recursion - recursive function calling itself more than one times. ✓

Excessive recursion - If the recursive function is calling itself multiple times for the same parameter. ✓

Now, we are trying to remove excessive calls:-

⇓

using memorisation

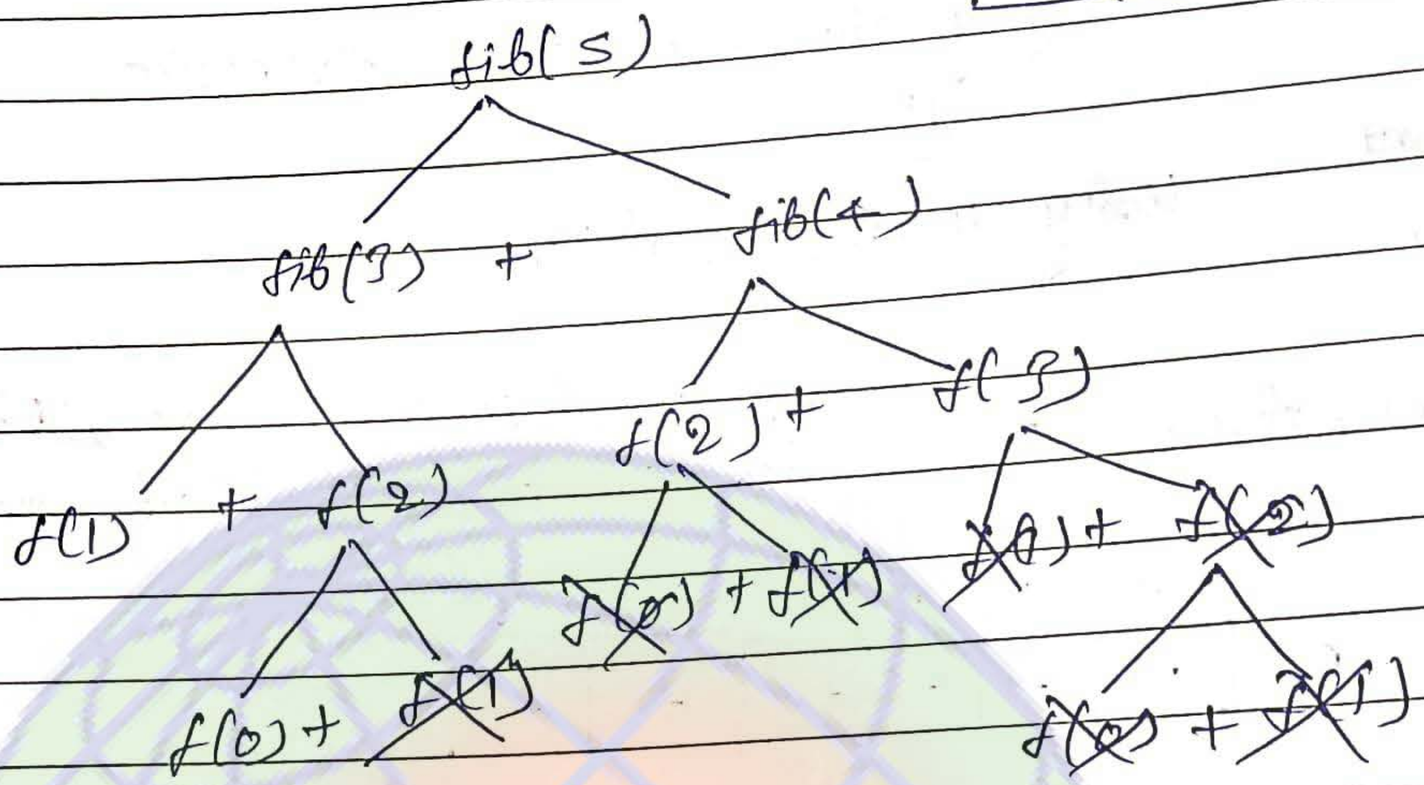
☆ memorisation → used to store the results of the function call to avoid excessive calls.

	0	1	2	3	4	5
F	-1	-1	-1	-1	-1	-1

```
int fib (int n)
{
  if (n<=1)
  {
    F[n] = n;
    return n;
  }
  else
  {
    if (F[n-2] == -1)
      F[n-2] = fib (n-2);
    if (F[n-1] == -1)
      F[n-1] = fib (n-1);
    return F[n-2] + F[n-1];
  }
}
```

So, now the call is as follows.

	0	1	2	3	4	5
F	0	1	1	2	3	5



Recursive version.

So, No. of calls is = 8

max. stack size = 4

★ Note Iterative version

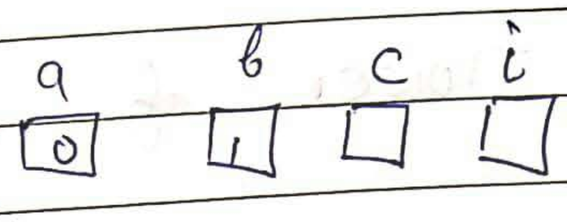
iterative version

```
int fib( int n)
{
 int a=0, b=1, c, i;
 for (i=0; i<n; i++)
 {
  c = a+b;
  a = b;
  b = c;
 }
 return a;
}
```

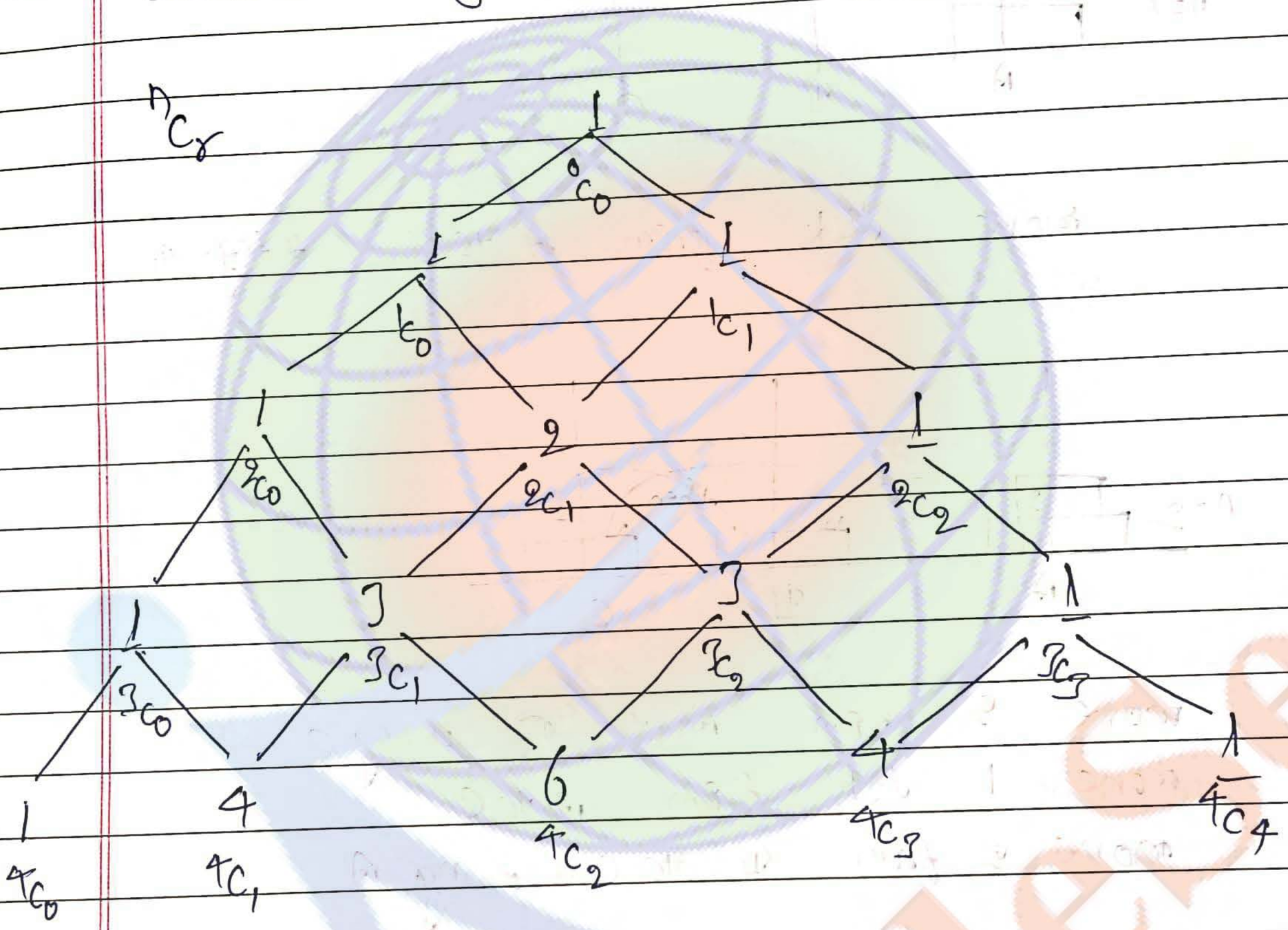
i	a	b	c
0	0	1	

table बना कर solve करें

fib(5)



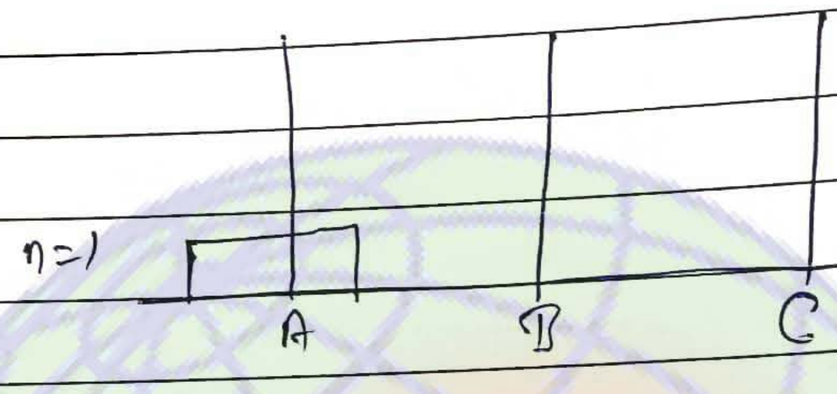
★ Pascles triangle:-



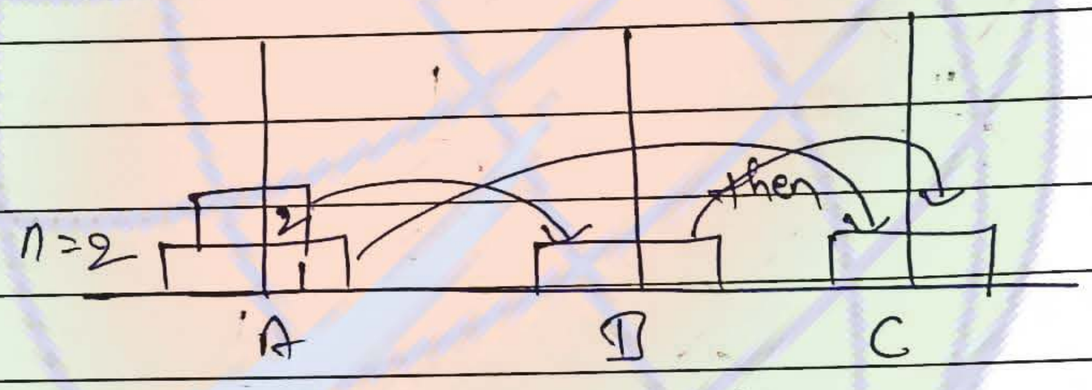

```
int comb (int n, int r)
{
    if (r==0 || r==n)
        return 1;
    else
        return comb(n-1, r-1) + comb(n-1, r);
```

This is recursive definition for finding any combination value.

★ Tower of Hanoi

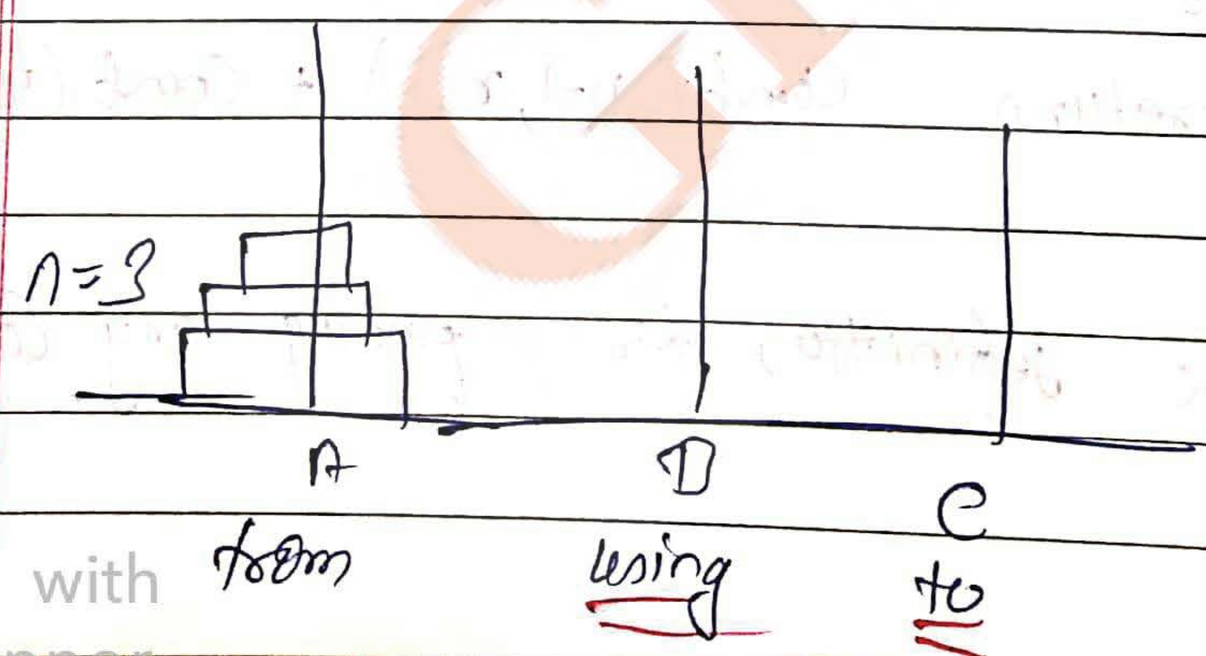


move disk from A to C using B



move 2 from A to B using C
move 1 disk from A to C
move 2 from B to C using A

(i) One disk should move at a time
(ii) larger disk should not be kept over smaller disk



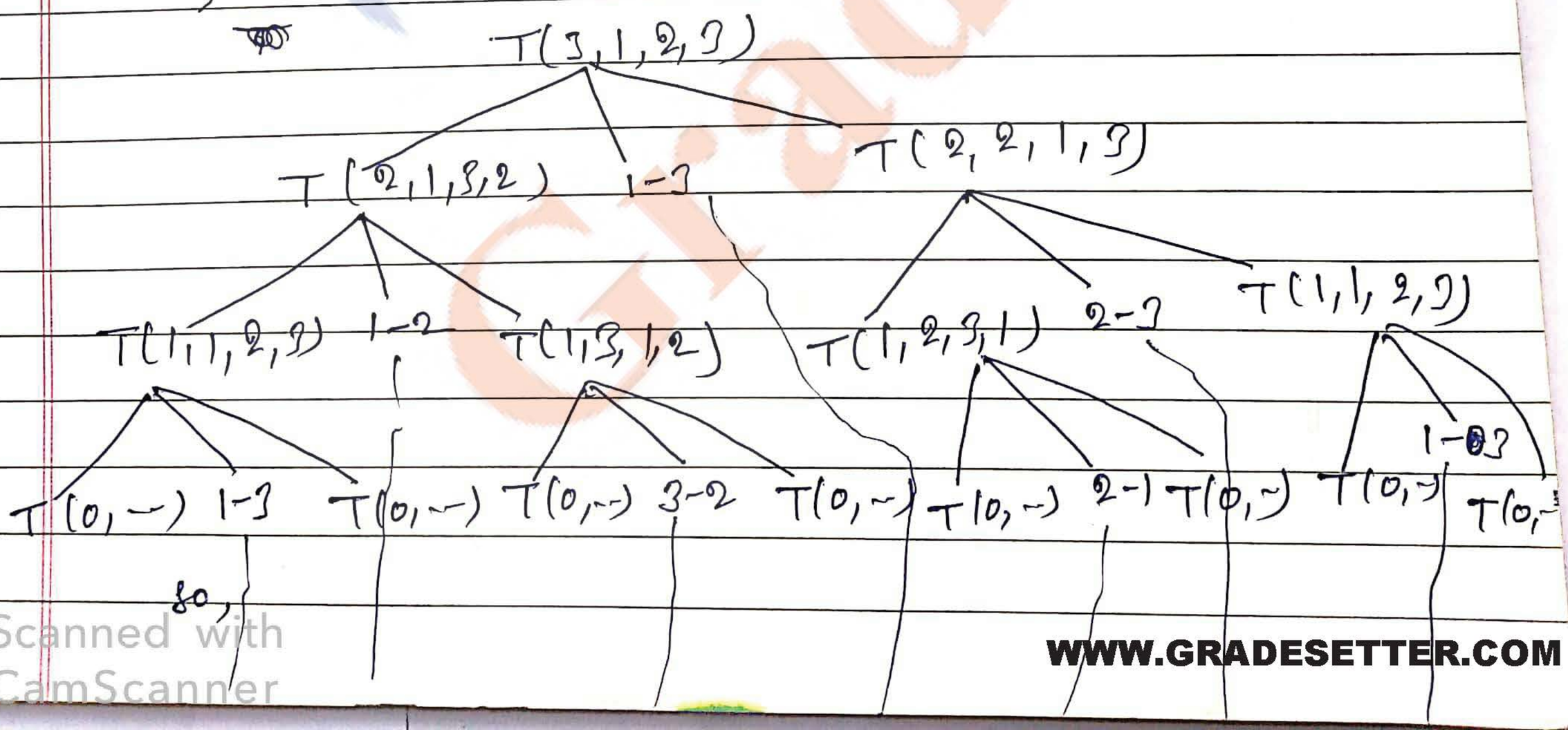
1.> move 2 disks from A to B using C
2.> move a disk from A to C
3.> move 2 disk from B to C using A.

Now,

for n disks
1.> move n-1 disks from A to B using C
2.> move a disk from A to C
3.> move n-1 disks from B to C using A

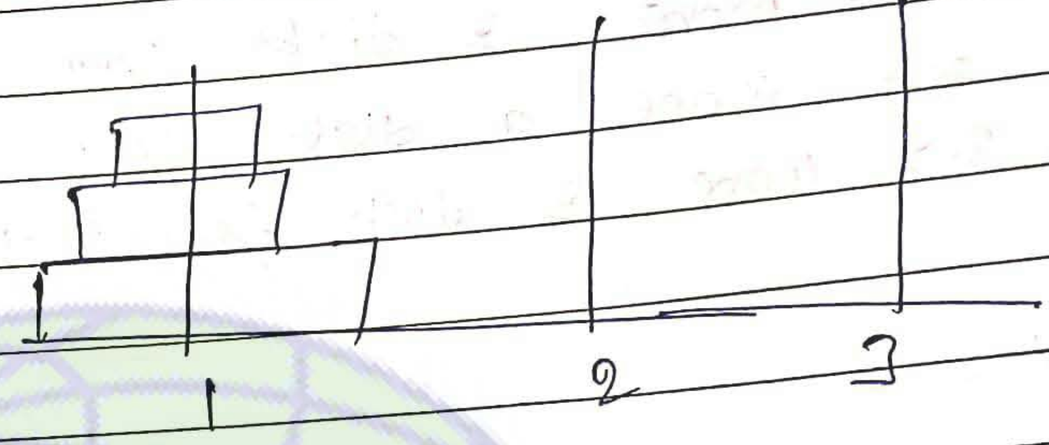
```
TOH (int n, int A, int B, int C)
{
  if (n>0)
  {
    TOH (n-1, A, C, B);
    Printf ("move a disk from %d to %d", A, C);
    TOH ( n-1, B, A, C);
  }
}
```

Note
TOH(n, from, using, to)



so,

1-3
1-2
3-2
1-3
2-1
2-3
1-3



Pointers:

① data variable = int a = 10 ;
address variable = int *p

a
10
200

2.> declaration: int a = 10;
int *p;
initialization: p = &a;
dereferencing: printf ("%d", *p); — 10
printf ("%d", p); — 200

dereferencing means p is pointing to that address and take the value

a
10
200/201

p
200
250/250

Note:
Pointer will take address of 1st byte of the data

3.> Size of Pointer

64 KB

int p1; — 2
char p2; — 2
float p3; — 2

0 — 65575

- Pointer is the address of the data.
- Pointer is like a key.
-

2 bytes

0 - 65575

Date
Page

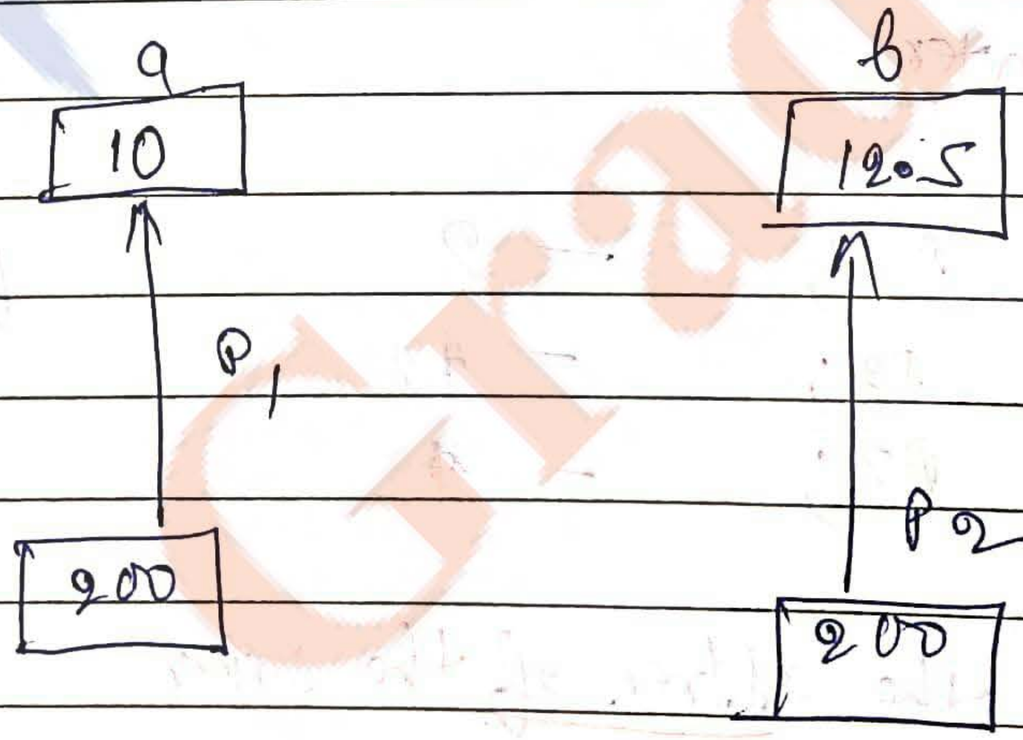
- Every pointers takes 2 bytes, it can refer to memory ranging to memory 64 KB.
- Pointer is only taking address not data.
- Size does not depend on the datatype of pointer
- Pointer is unsigned integer (only the value from 0 to 65535)

Q) int a = 10;
int *p1;
p1 = &a;

float b = 12.5;
float *p2
p2 = &b;

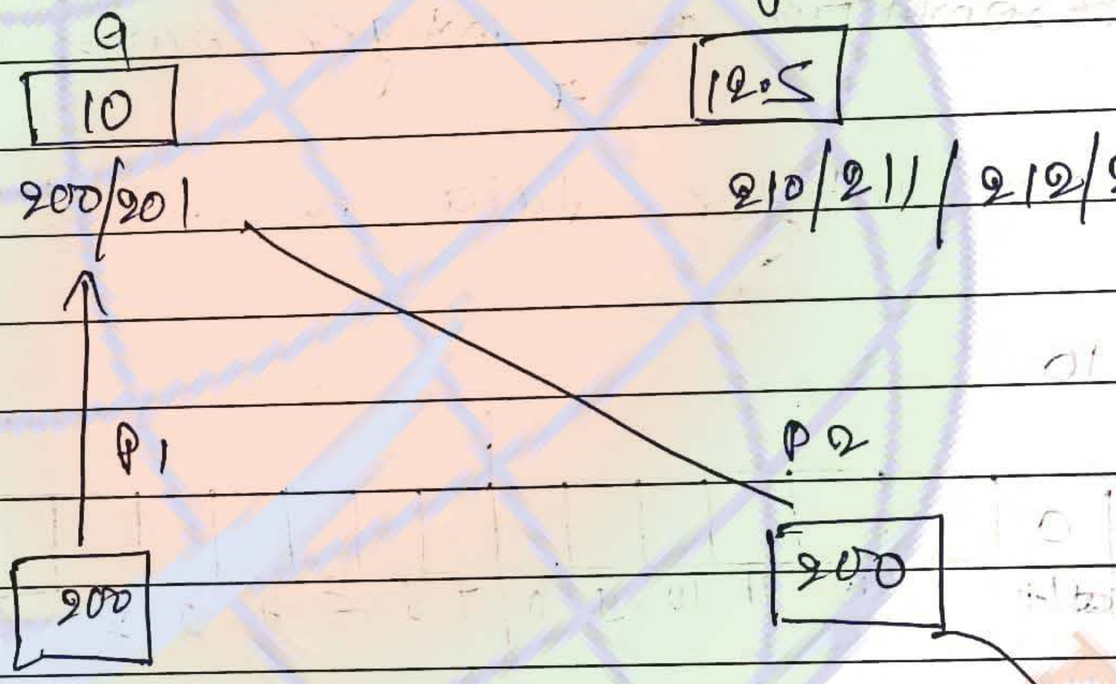
~~p1 = &b;~~

- A pointer of one type can store the address of data of same type only.



De-referencing :-
when the pointer is deference, it will access the data depending on data type of pointer.
so, - integeger pointer will access two byte.
and - float pointer will access four byte.

Here P_2 is float, but accessing a.

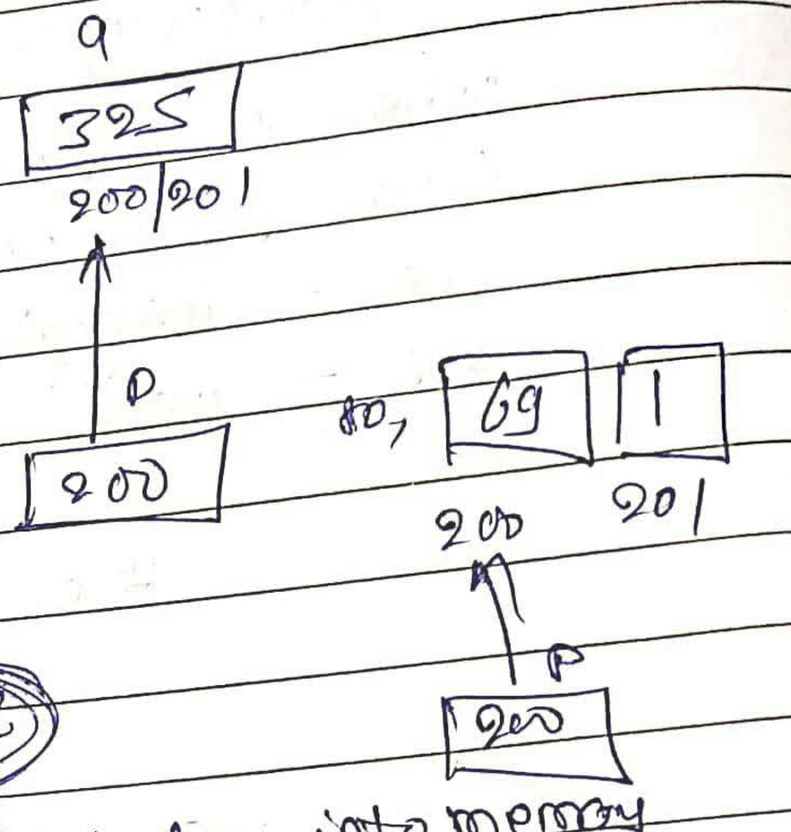



```
P2 = &a;
Pf ("%f", *P2);
```

Note: there is trying to access four bytes from a, but only two byte is present so its not true show error.

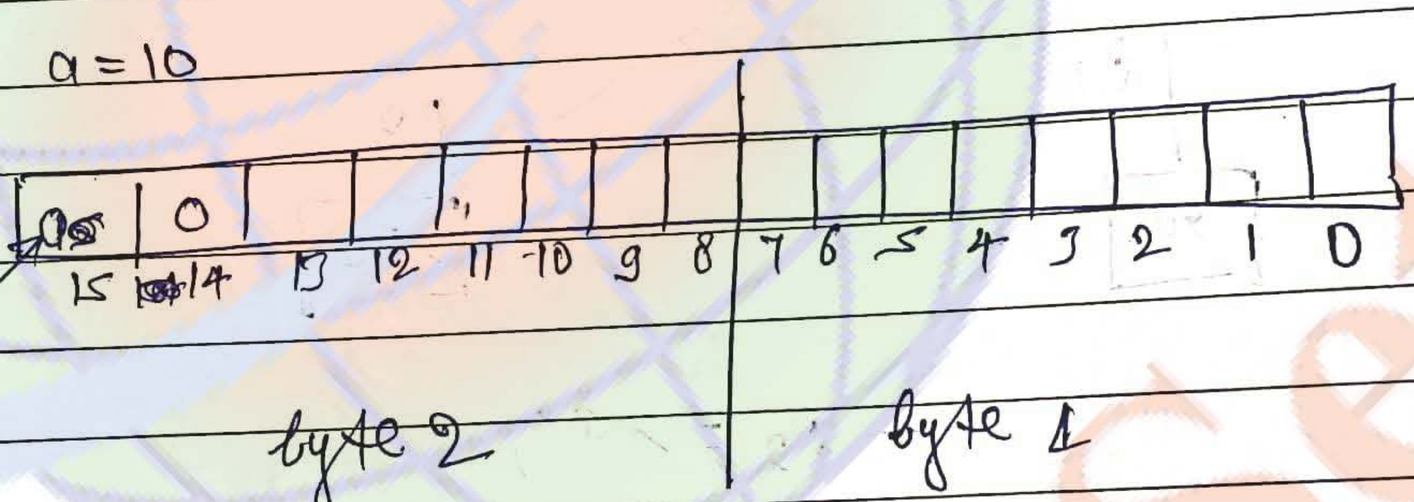
error - 2501

```
int a = 325;
char *p;
p = &a;
Pf ("%.c", *p);
```

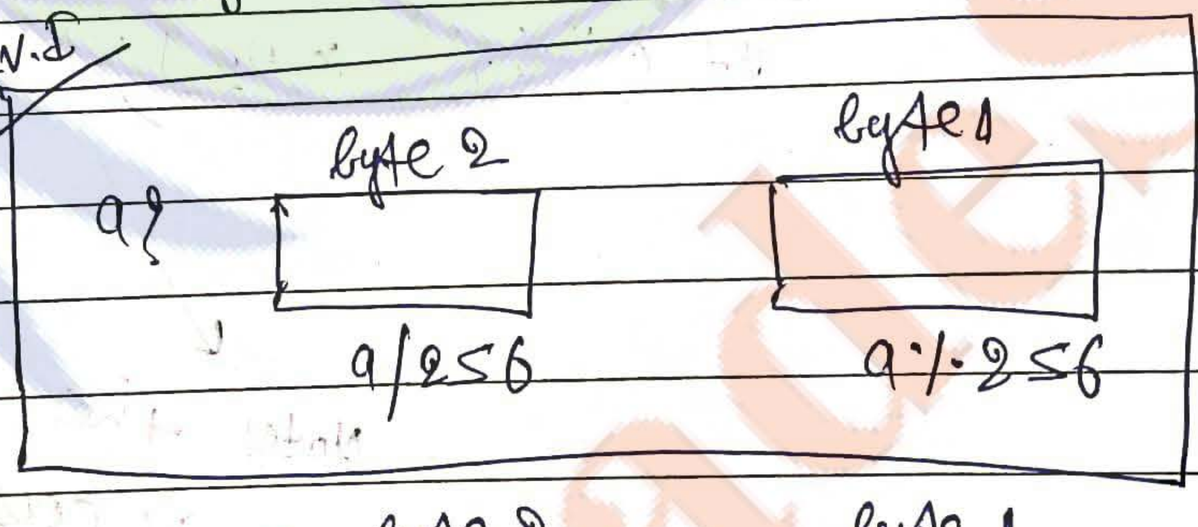


5) Representation of or integer data into memory

How to integer data is stored in memory:-



Note:
C, C++ using signed bit concept for -ve number
Java, C# using 2's complement for -ve number.



257/256

257/256

a = 257

1

1

a = 325

1

69

only paper work (on paper)

a = ?

byte2

byte1

a/256 (divide)

a%256 (mod)

a = 520

2

8

a = 325

byte2

byte1

1

69

(big)

(little)

so, we are using two method to store in the memory.

generally used

little endian

69

1

200

201

big endian

1

69

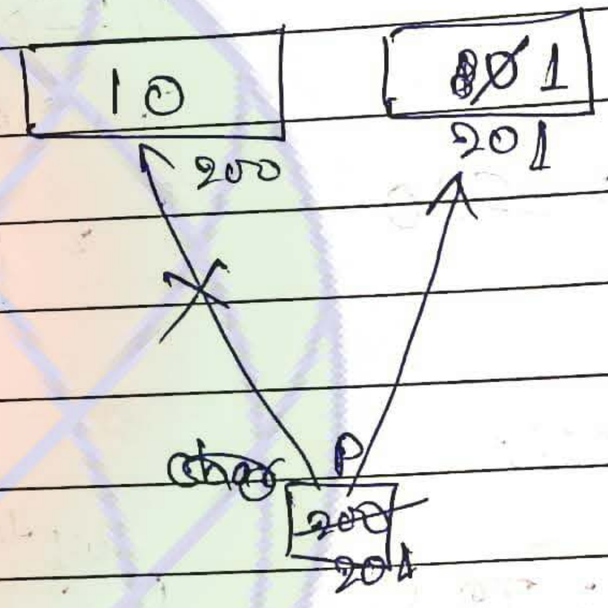
200

201

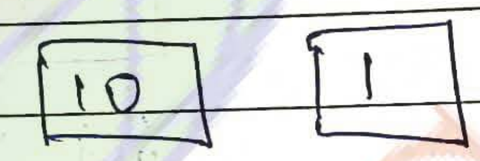
Eg.>
```
int a = 10;
char *p;
p = &a;
p++;
(*p)++;
pf("%d", a);
```

8 4 2 1
1 0 1 0

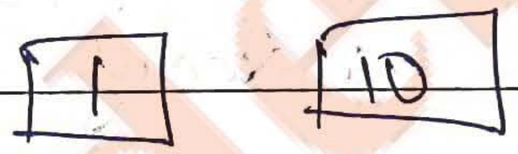
little endian form →



little endian form →



& inactual



9/256 9%256

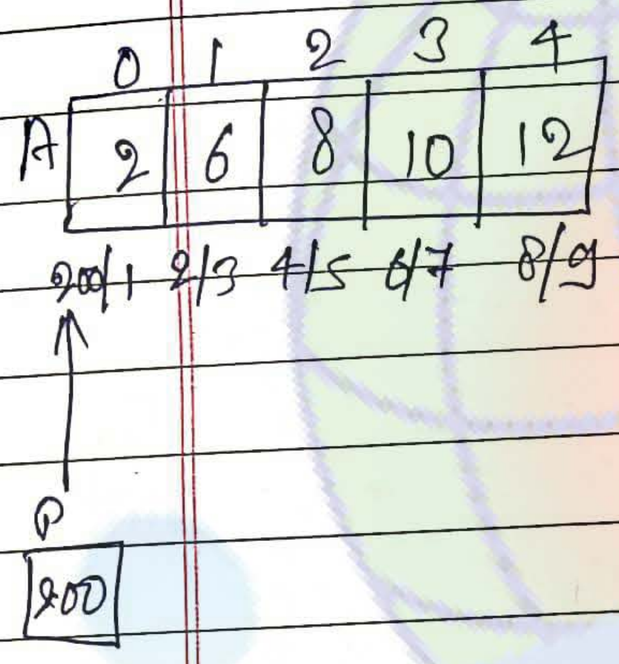
1×256 + 10 = 266 Ans

(divide को multiply, mod को sum करेंगे)

6) Pointer arithmetic :-

(a) Pointer to an array :-

```
int A[5] = {2,6,8,10,12};
int *P;
P = A;
Printf ("%d", P[2]); — 8
```



A[2] — 8

2[A] — 8

*P — 2

*A — 2

P — 200

P+2 — 204

*(P+2) — 8

Note:

P = A;
P = &A[0];
} same

→ All arithmetic operations are not allowed on pointers

1.) P++ : moving to next element

eg. if int = 2 byte memory
float = 4 byte memory

2.) P-- : moving to Previous element

3.) P+3 : moving to three element away/ give address of element 3 location away in forward direction

4.) P-3 : moving to three element Previous or gives address of element 3 location away in backward direction

```
int A[5] = {2, 6, 8, 10, 12};
int *p, *q;
p = A;
p = &A[0];
```

2	6	8	10	12

200/1 ↑ p 206/7 ↑ q

(5) d = q - p

(3) distance ⇒ Indicate no. of elements

200 ↑ p — 206 ↑ q = (206 − 200)/2 = 6/2 = 3 (elements)

Note: only above operation allowed in pointers for int

(1) p + q

(2) p * q

(3) p / q

(7) Note:

*P → dereferencing ✓

P++ → Pointer increment ✓

(10) ② ①
* ++P ;
← right to left

*(++P) —— 6

11.) ++ *P ; —— 7
←

(data increment) ++ (*P) → Increment the data where P is pointing. ✓
"take the data and increment the data")

most Commonly used:
① ②
11.) * P++ ;

a.) *P — take the data and
b.) P++; — move the pointer.
} precedence of ++ is higher
post increment is higher.

↓
data ki bina, the Pointer increment.
("take the data and move pointer)

eg.) int A[5] = {2, 6, 8, 10, 12};
int *P, *q;
{P = A;
or P = &A[0];
for (i=0; i<5; i++)
{

```
pf("%d", *p); } or pf("%d", *p++);
p++;
}
```

Ans Print all element
and pointer will be after last element.

or

Note:
```
for (i=0 ; i<5 ; i++)
{
pf ("%d", *(p+i)); or *p[i]
}
```
∴ *(p+i) → p[i]

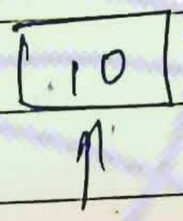
Print all element
and
pointer will remains on base address.

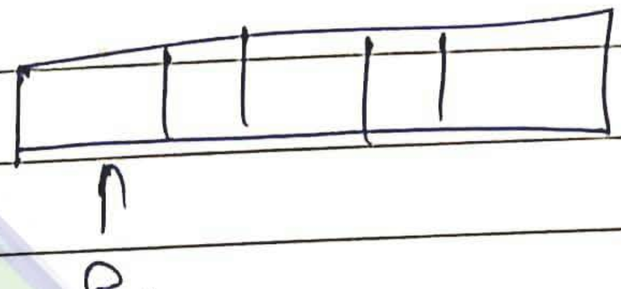
(8) Why pointer's have datatype

Ans Reasons to pointer have data types:-
(i) Dereferencing
(ii) - Pointer arithmetic

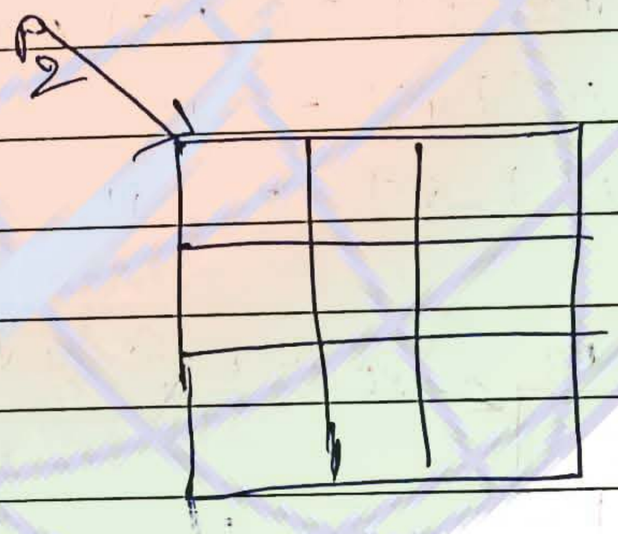
Double pointer :-

(1) int *P1; — single pointer





int **P2; — Double pointer.



int a; — in stack

int A[5]; — in stack

int A[3][4] — in stack

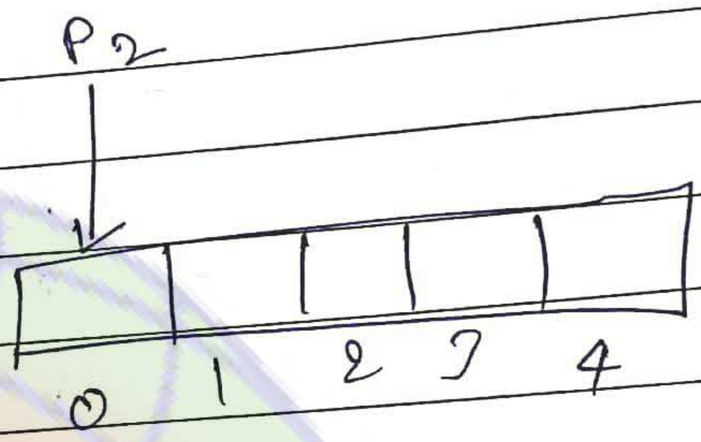
Note

single int int *P1;

P1 = (int *) malloc(2)

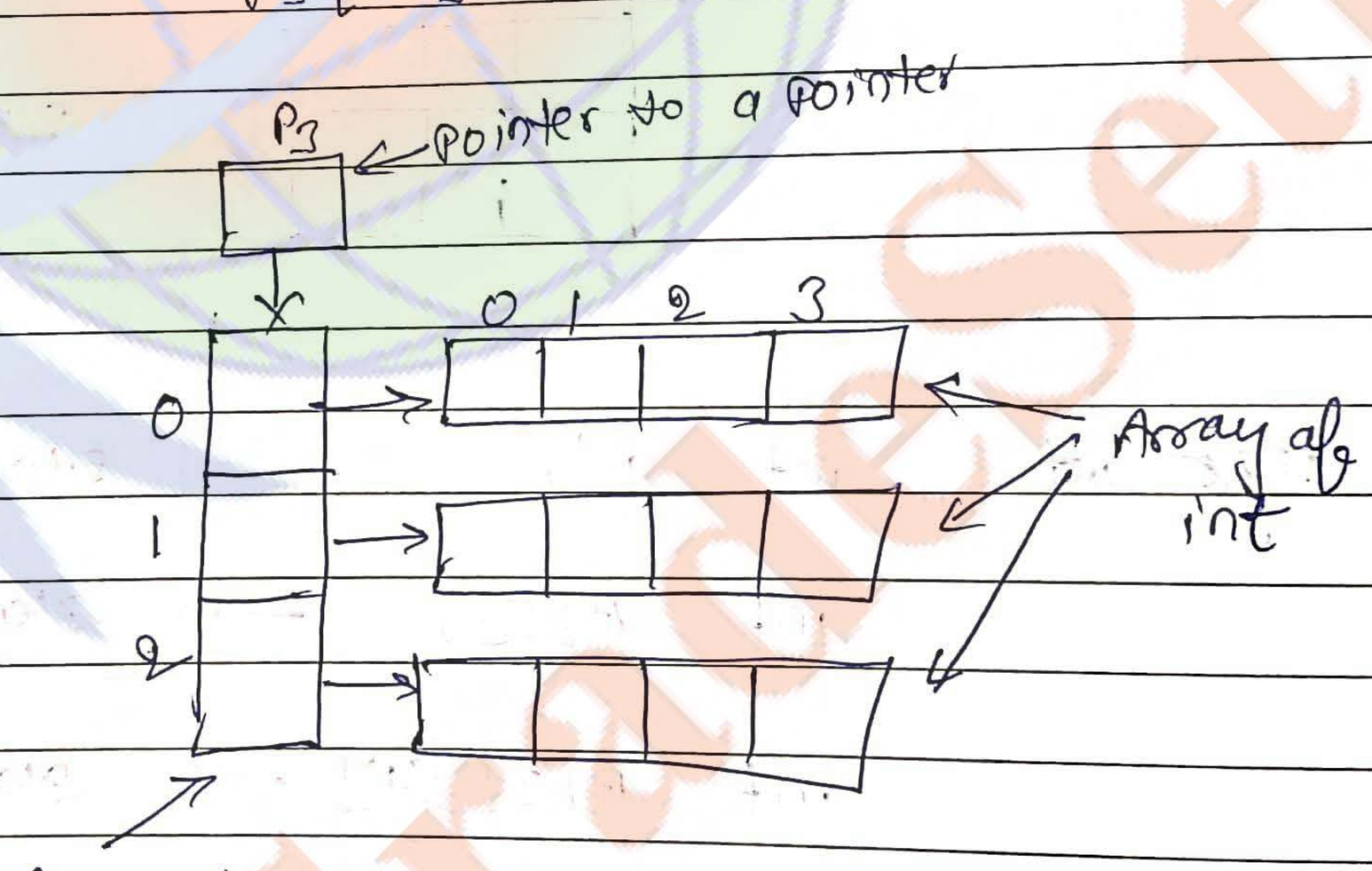
Array of int

```
int *p2;
p2 = (int *) malloc(10);
```



2D Array 3x4

```
int **p3;
p3 = (int **) malloc (3*2);
p3[0] = (int*) malloc( 4*2);
p3[1] = //
p3[2] = //
```



```
main()
{
int n;
cout << "what is array size";
cin >> n;
```

X int A[n]; (not allowed) → stack में array fixed होना चाहिये।

✓ int *B = new int[n]; ← heap में new से बनते है।

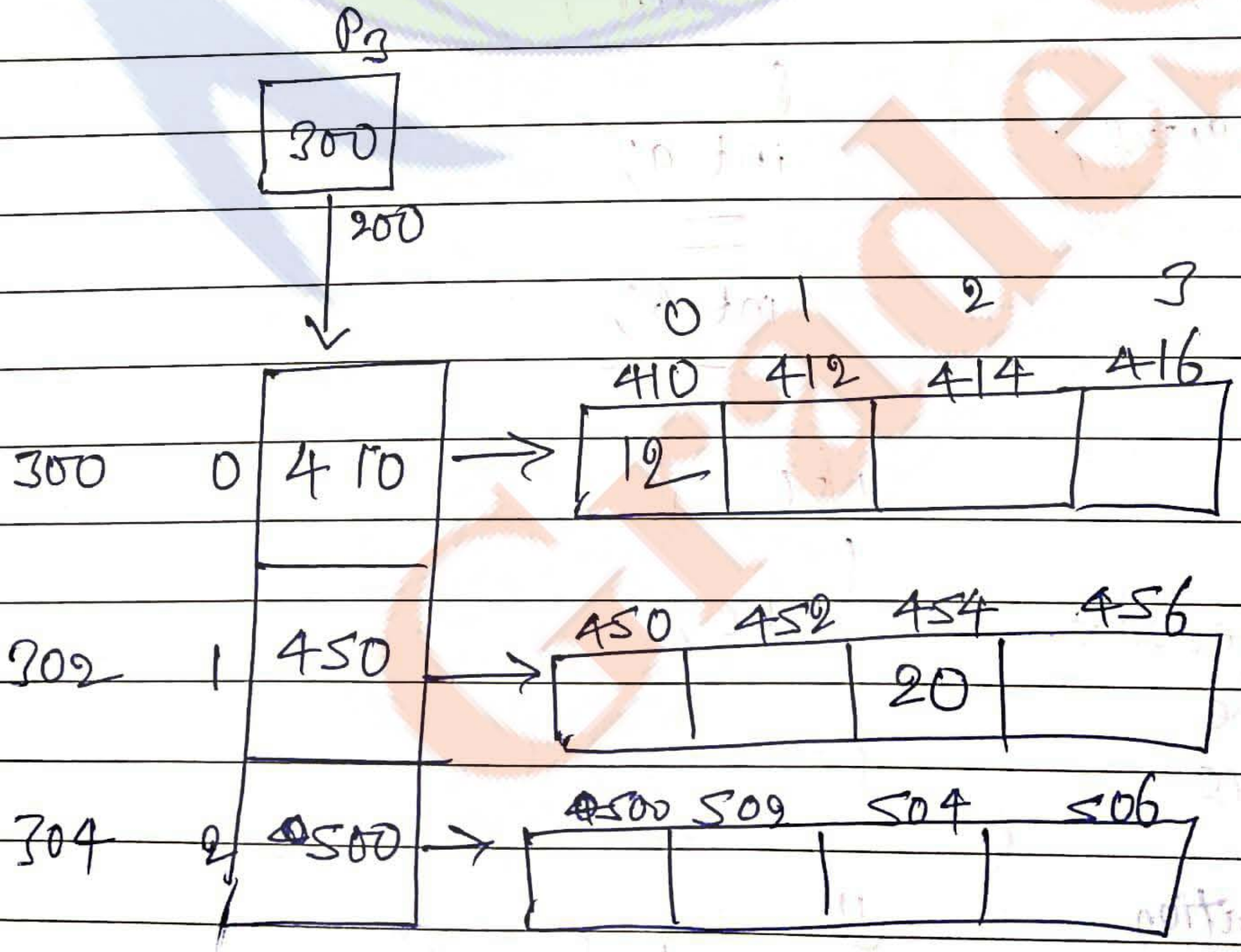
```
}
```

Note

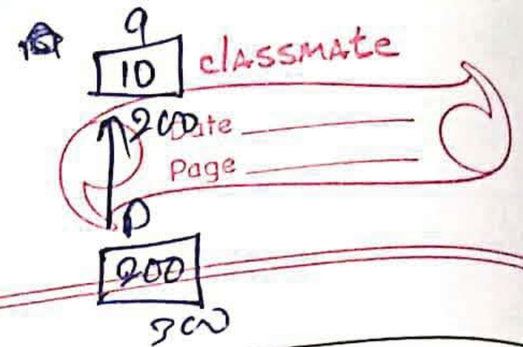
stack के array का size = static है ✓

heap " " " " = dynamic है। ✓

(2)



a = 10
*a = 200
p = 200
*p = 300
*(*p) = 10



Printf ("%d", p3); — 300
" " *p3 — 200
*p3 — 410
**p3 — 12 (Double pointer data तक पहुंचेगा)

&p3[1] → p3+1 — 302

p3[1] → *(p3+1) — 450

&p3[1][2] → *(p3+1)+2 — 454

p3[1][2] → *(*(p3+1)+2) — 20

Note: C-language

```
main()
{
int a, b, c;
}
```

Here, all the variable should be declared once

&

Here activation record is static.

C++ language

```
main()
{
int a;
int b;
lfb( );
{ int c;
}
```

You may declare at any time

Here activation record

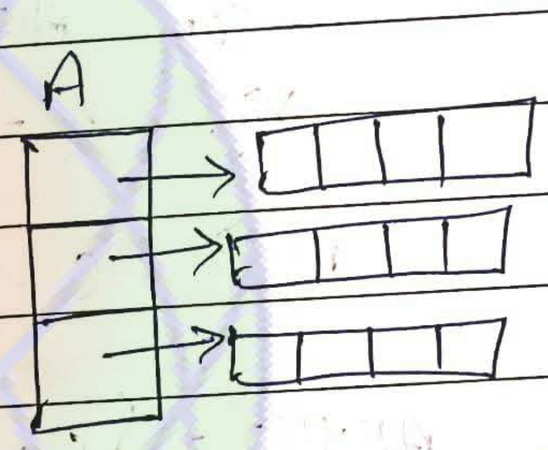
(1) methods of declaring 2D array

1.> int A[3][4];

both row & column are static

A[i][j] = n;
(left side)

2.> int *A[3];
A[0] = new int[4];
A[1] = "
A[2] = "

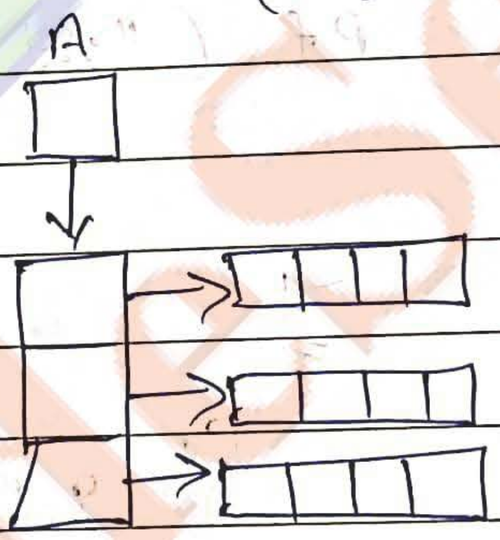


rows are static.
& size of rows are dynamic

A[i][j]=15
A[2]=9
(left side)

3.> int **A;

row → A = new int*[3];
column → A[0] = new int[4];
A[1] = "
A[2] = "



both row & column are dynamic.

Note:
x = 10 (write)
y = x (read x)

A[i][j] = x;
A[i] = k;
A = q;
(left side)

Q.> which are allowed or which are not allowed on the left hand side of assignment

1.> int *A[5];
2.> int B[2][4];
3.> int **C;

✗ a. B[c]
✓ b. C[c]
✓ c. A[c]
✓ d. A[c][y]
✓ e. C[c][y]
✗ f. D
✓ g. C.
✗ h. A
✓ i. A = B;
✗ j. B = A;
✓ k. C = B

only see left side

eg:> int A[][] = {{2,5,4}, {5,6,7}, {8,9,1}};
int *p[5] = A;
pf ("%d", *(*(p+1)+2));

Ans

{{2,5,4}, {5,6,7}, {8,9,1}}

(p+1) + 2

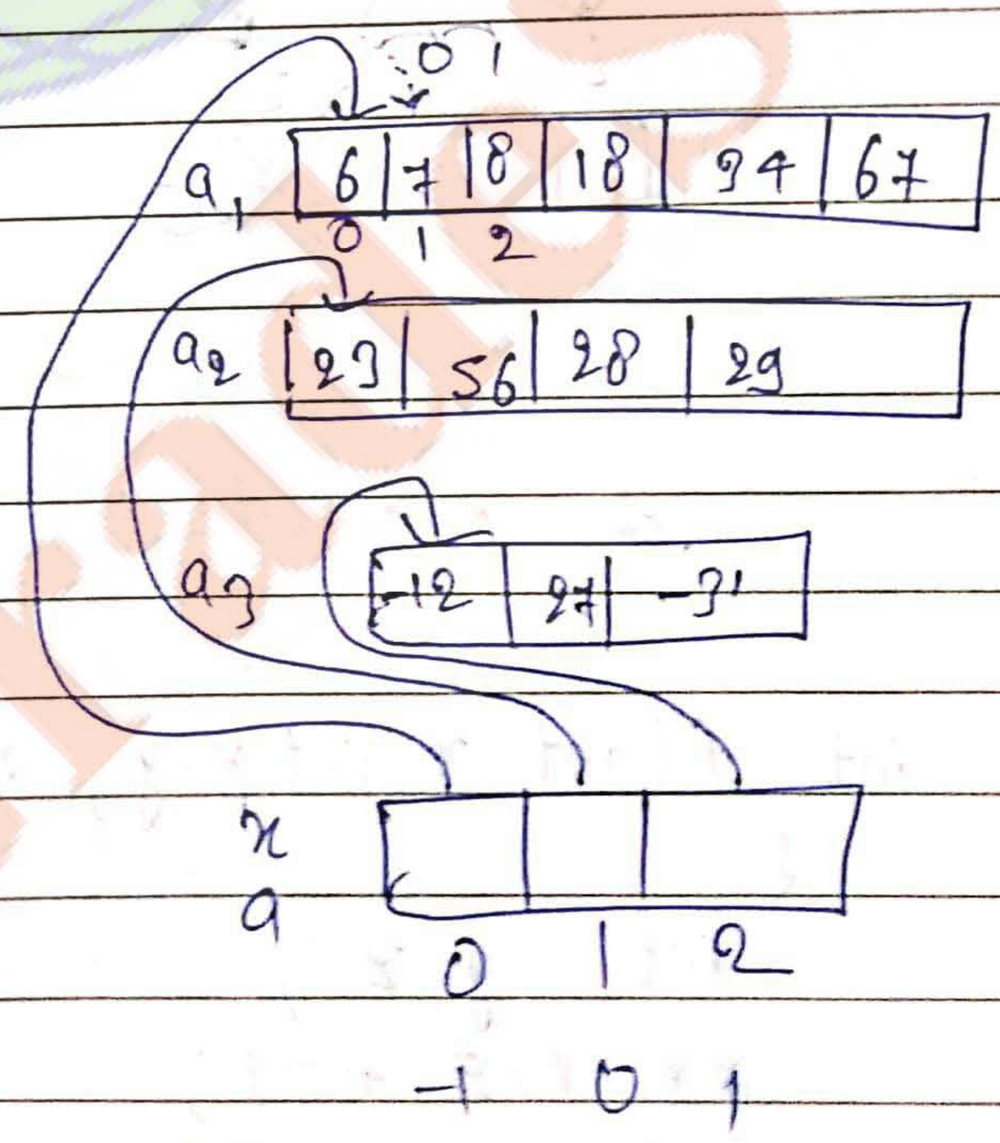
	0	1	2
0	2	5	4
1	5	6	7
2	8	9	1

∴ *(*(p+1)+2) = p[1][2]

1 2

gate
Eg.>
```
int a1[] = {6, 7, 8, 10, 34, 67};
int a2[] = {27, 56, 28, 29};
int a3[] = {-12, 27, -31};
int *x[] = {a1, a2, a3};
void print {int *a[]}
  {
0 — Printf (a[0][2]);
-12 — Printf (*a[2]);
-1 — Printf (*++a[0]);
27 — Printf (*(++a)[0]);
0 — Printf (a[-1][+1]);
  }
main()
{
  Print(x);
}
```

Ans
2



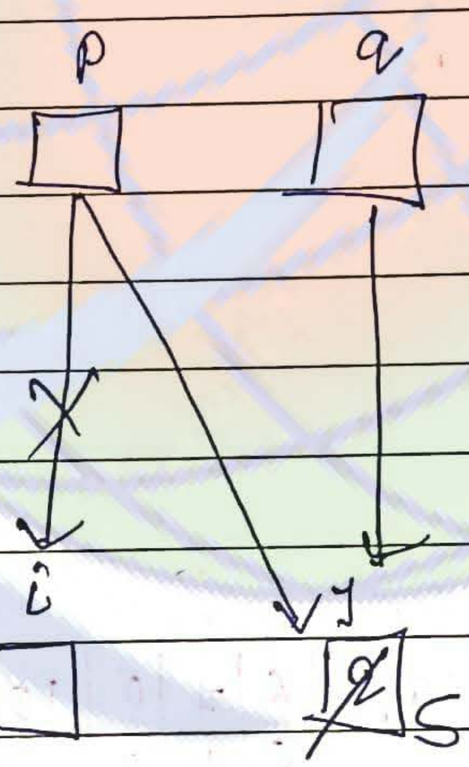
9. 8, -12, 7, 27, 8

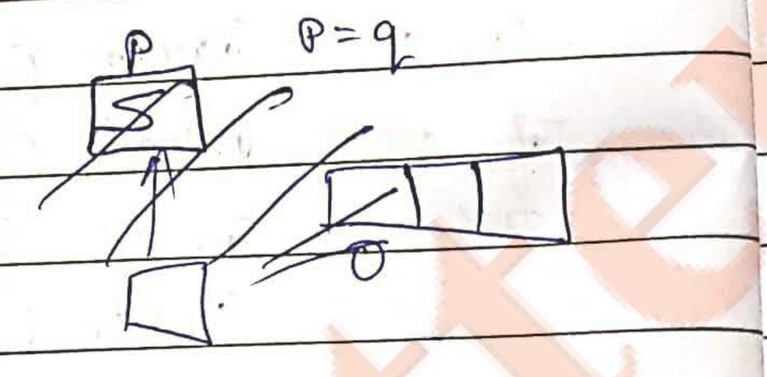
A

gate
eg 9.)

```
f(int *p, int *q)
{
p=q;
*p=5;
}
main()
{
int i=0, j=2;
f(&i, &j);
Pf(i,j);
}
```

soln



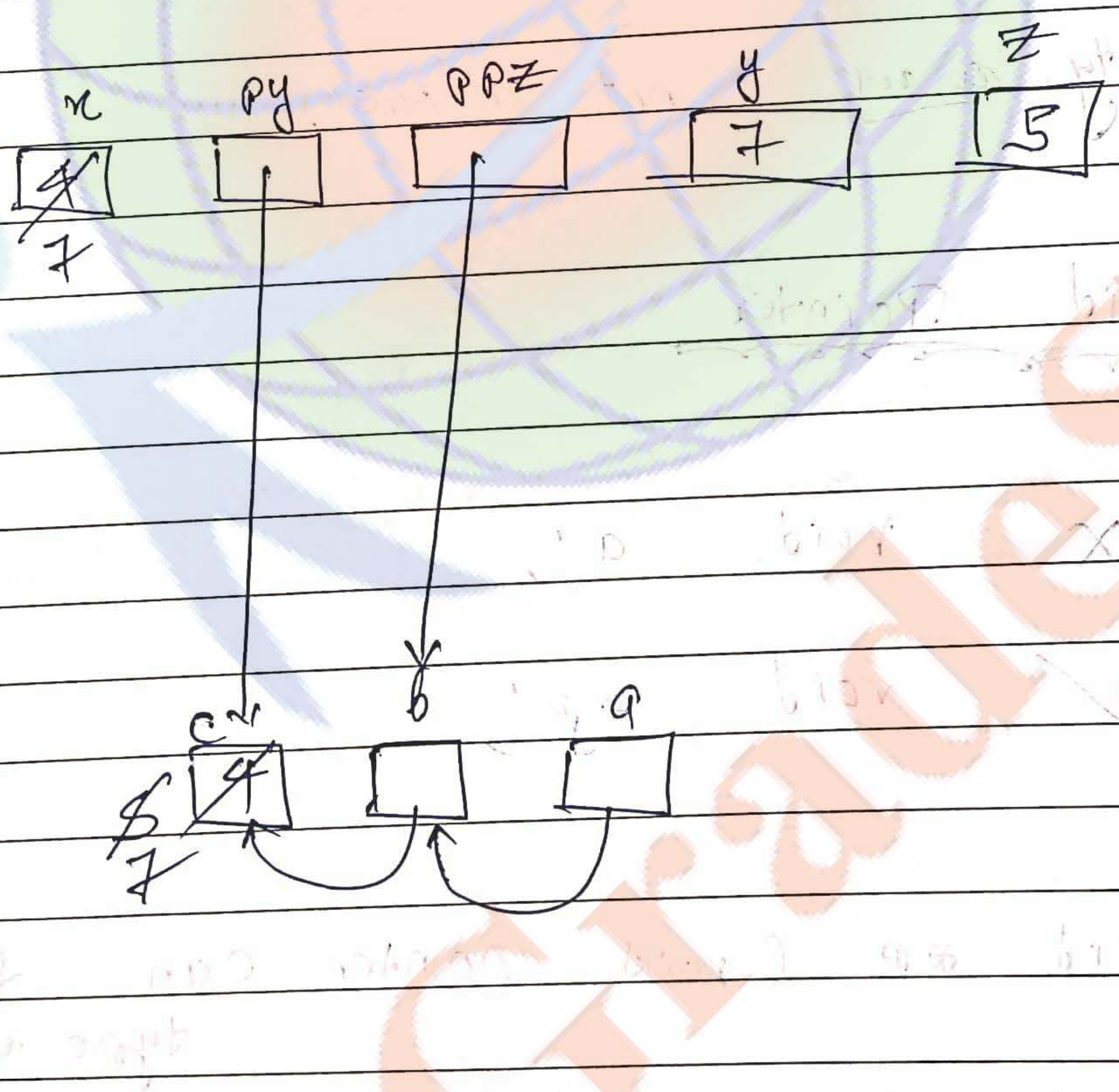


Ans 0, 5. Ans.

gate
eg 10.)

```
int f(int x, int *py, int **ppz)
{
int y, z;
**ppz += 1;
z = **ppz;
```

```
* py += 2;
  y = *py;
  x += z;
  return x+y+z;
}
void main()
{
int c, *b, **a;
c=4; b=&c; c=&b;
Pf(f(c, b, a));
}
```



x, 7+7+5 = 19 Ans

x+y+z

eg 4:

```
f(int *p)
{
}
```

```
g(int p[])
{
X int q[];  (Not allowed)
}
```

```
main()
{
  int A[5];
X int B[];  (Not allowed)
  f(A);
  g(A);
}
```

Note
⇒ empty array is not allowed;

Void Pointer:

X void a;

✓ void *p;

- void *p (void pointer can store any type of data).

```
int   x=10;
float y=12.5;
char  z='A';
void  *p;
```

```
p = &x;
p = &y;
p = &z;
```

- void pointer can not be de-referenced.

```
X  pf(*p);
```

- pointer arithmetic is not allowed on void pointer.

- void *malloc(int size)

```
int *p;
p = (int *) malloc(10);
```

★ Pointer to a function :-

```
void display()
{
pf("Hi");
}
main()
{
decl → void (*fp)();
init → fp = display;
func call → (*fp)();
}
```

- Declaration of a function should be same as prototype of a function.

Date
Page

# int add (int x, int y)
{
return x+y;
}

int max (int a, int b)
{
return a>b ? a:b;
}

main()
{
int (*fp) (int, int);
fp = add;
(*fp) (10,5); ✓
fp = max;
(*fp) (10,5);
}

- A function pointer can point on all those function which has same prototype.

1.) int f(int)
2.) g(int) — by default 'int' ई
3.) void h() → by default 'void' ई on
4.) void p(void)

int (*m) (int);
void (*n) (void);

~~a.~~ m = f;
~~b.~~ m = h;
~~c.~~ m = g;
~~d.~~ n = p;
~~e.~~ n = h;
~~f.~~ n = g;

★ Declarations -

Array { int A[5];
int *B[5]; → five pointers.

single pointer { int *c;
int (*c)[5];

int fun(int, int); — a function which takes two integer and returns integer.

int *fun(int); — fun which takes int as parameter and returns pointer to an integer

int (*fun)(int); — Pointer to a fun
(Bracket is from left to right)

Note: declaration is from right to left

int **fun(float); - fun which takes float

Common Problems using pointers:-

1. memory leak:-

```
int *p = (int *) malloc(10);
p = NULL;
```

↓

memory leak

2. Uninitialized pointer

```
int *p;
*p = 25;
```

3. dangling pointer

```
int *p = new int[5];
int *q = p
free(q);
```

gate

eg:-

```
int *p1()
{
int *p;
return p;
}
```

↓ uninitialized pointer

```
int *p2()
{
int x = 10;
return &x;
}
```

↓ dangling pointer

```
int *p3()
{
int *p = new int[7];
return p;
}
```

↓ No Problem

(iv) Toeplitz:-

$$A = \begin{bmatrix} 2 & 4 & 6 & 8 \\ 3 & 2 & 4 & 6 \\ 5 & 3 & 2 & 4 \\ 7 & 5 & 3 & 2 \end{bmatrix}$$

4x4

n x n

$A[i][j] = A[i-1][j-1]$

No. of elements $= n - row + 1$

$n - 1 \rightarrow col$

$2n - 1$

0	1	2	3	4	5	6
2	4	6	8	3	5	7

↑ row ↑ col

if $(i <= j)$

$Addr(A[i][j]) = L_0 + \underbrace{[j - i]}_{index} * \omega$

if $(i > j)$

$Addr(A[i][j]) = L_0 + MS + [i - j - 1] * \omega$

```
Union Test
{
  char a';
  int b';
  float c';
}
```

Free Union

```
Union Test t;
t.a = 'A';
t.b = 500;
```

Decrement Union

```
Record : Test (int ch)
{
case ch=1
    char a;  ——(1)
case ch=2
    int b';  ——(2)
case ch=3
    float c;  ——(4)
}
```

```
Var test(2) t';
X t.a='A'; X
t.b=25; ✓
```

* In decrement we are allows only fixed member to use.

```
{
}
```

decriment = 2 byte, takes

// C language supports free union

// C language allows decriment union for using fixed values.

Arrays:

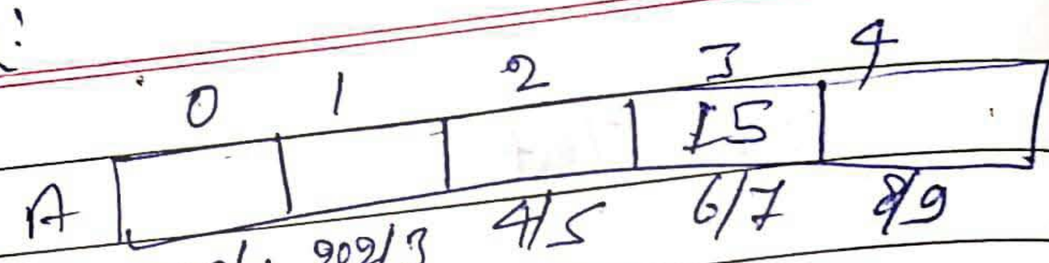
(1)

```
int A[5];
A[3] = 15;
```

Addr[A[3]] = Lo + 3*2

= 200 + 6

= 2006




```
int a = 30;
```

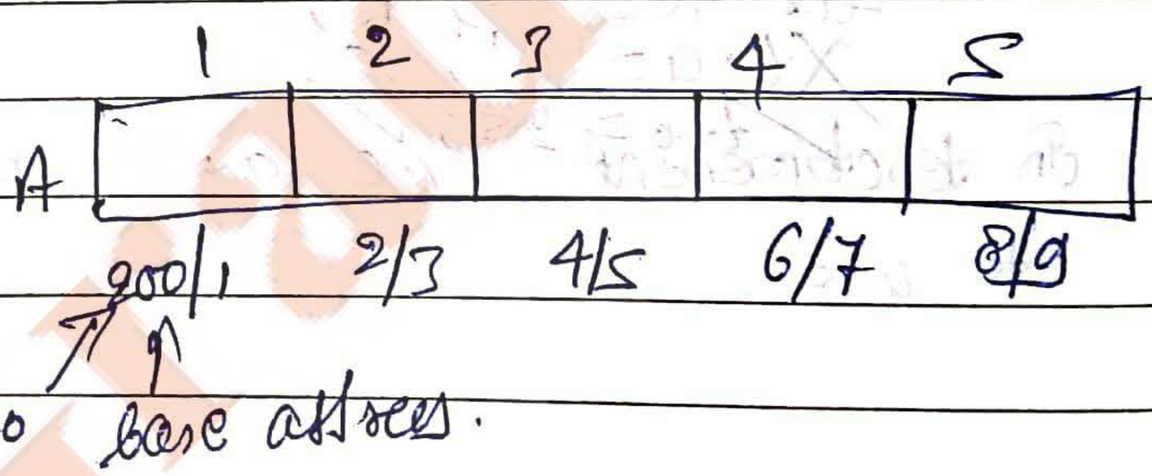
* How compiler handle array, and generate relative location.

- The address are defined based on base address

Addr (A[i]) = Lo + i * w

Lo → base address
i → Index
w → size of data type

(2) Array index starting from 1.



var A[1...5] : Integer

Addr(A[i]) = Lo + (i-1) * w

Addr (A[1]) = Lo + 0
= 200

Addr (A[2]) = Lo + 1 * 2
Addr (A[3]) = Lo + 2 * 2
Addr (A[i]) = Lo + (i-1) * w

(3) Declare an array, by any starting bound

var A[lb ---- ub] Type

Addr (A[i]) = Lo + (i - lb) * w

eg.> int A[-5, ---- 5]

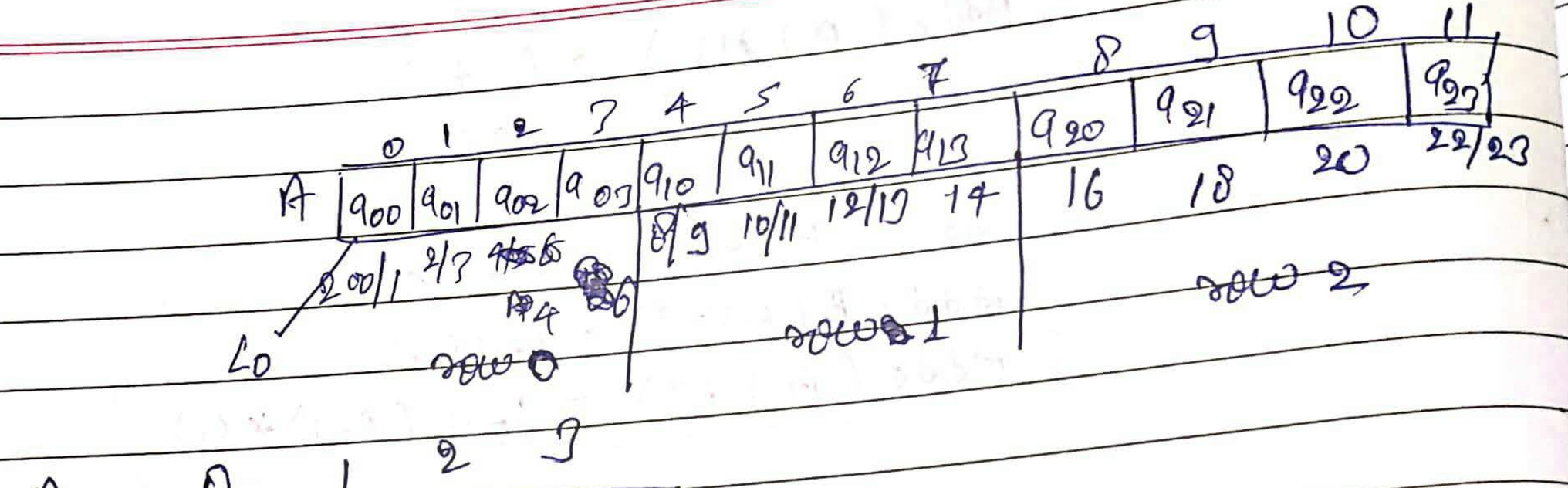
addr (A[1]) =
Lo = 100
w = 2

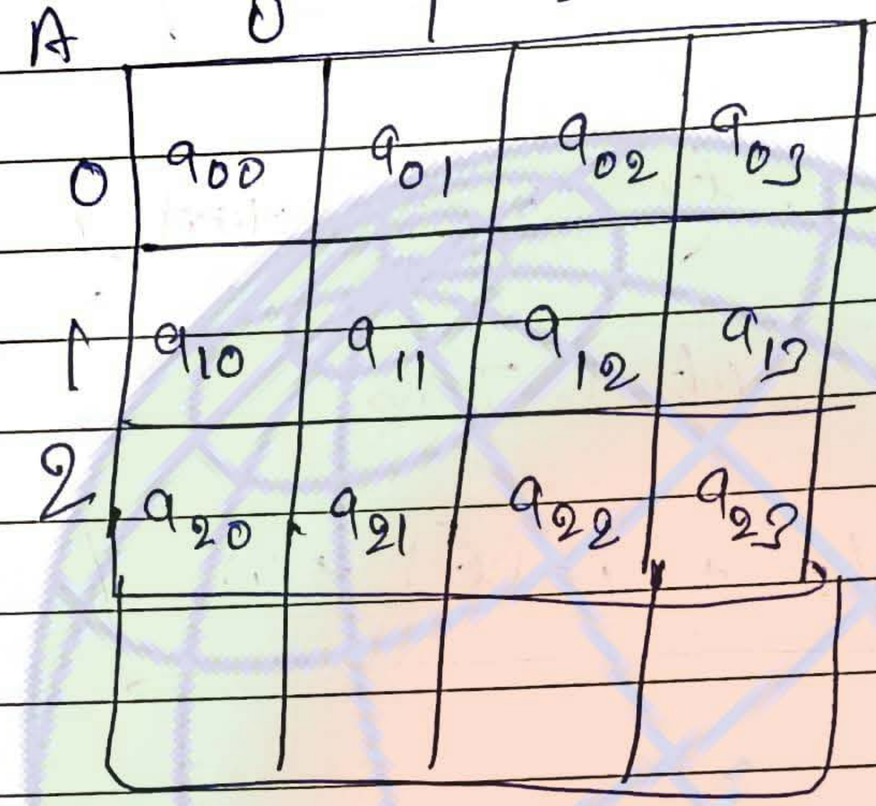
Addr (A[1]) = 100 + (1+5) * 2
= 112

eg.> Addr (A[2]) = 112
Addr (A[3]) = 116
Lo = ? Lo = ?

(4) 2D array :-

int A[3][4];





1.> Row major mapping
2.> Column major mapping

Row measure:

Addr (A[i][j])

Addr (A[1][3]) $= L_0 + [1 \times 4 + 3] \times 2$

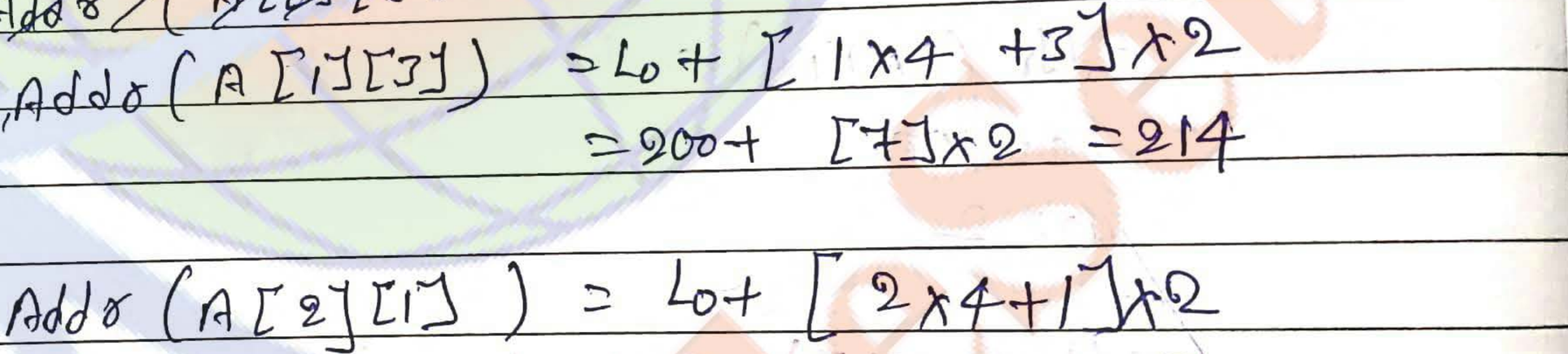
$= 200 + [7] \times 2 = 214$

Addr (A[2][1]) $= L_0 + [2 \times 4 + 1] \times 2$

$= 200 + [9] \times 2 = 218$

Addr [A[i][j] $= L_0 + [i \times n + j] * w$

column measure:



Addr (A[1][2]) = L_0 + (2*3 + 1) * 2

= 200 + 7 * 2 = 214

Addr (A[2][3]) = L_0 + [3 * 3 + 2] * 2

= 200 + [11] * 2 = 222

Addr (A[i][j]) = L_0 + [j * m + i] * w

Note' - Both are equally efficient
- In C language row measure is used.
- Row measure is also known as Lexical analysis.

In C language row

Gate Q1> C = A x B

1.) A in Row measure
B in column measure
2.) both in row measure
3.) both in column measure
4.) Independent of mapping/representation.

Gate Q2.) var A[1--10][1--15] Type

L_0 = 100, w = 1

addr (A[i][j]) =

If Lexical mapping/ row measure is used.

Ans
- addr [A(i)[j]) = 100 + [10i + 15j] * 1

= 1004

= 84 + 15i + j

(5)

- Row major index starting

int A[1 — m][1 — n]

Addr(A[i][j]) = b + [(i-1) * n - j - 1] * w

- Row major for any index

Type A[lb1 — ub1][lb2 — ub2]

Addr(A[i][j]) = b + [(i - lb1) * (ub2 - lb2 + 1) + (j - lb2)] * w

6) 3D Array

int A[m][n]

new Col

int A[m][n]

new Column

Date
Page

If compiler has generated ~~three~~ intermediate
(eg) Find datatype and dimensions for
int A[10][20]

t1 = i * 1024
t2 = j * 32
t3 = 8 * 4
t4 = t1 + t2
t5 = t4 + t3
t6 = A[t5]

(7) = Type = A[D1] ~~A~~ [D2][D3][D4]

Addr (A[c1, c2

Date
Page

Horner's Rule:

$$P(x) = 4x^3 + 3x^2 + 9x + 7$$

$$P(x) = 4 \times x \times x \times x + 3 \times x \times x + 9 \times x + 7$$

3 multiplications, 2 multiplications, 1 multiplication $= 3+2+1$

$\left(O(n^2) \right)$

A/Q rule:-

$$P(x) = 7 + 9x + 3x^2 + 4x^3$$
$$= 7 + x(9 + 3x + 4x^2)$$
$$= 7 + x(9 + x(3 + 4x))$$

$\rightarrow 3$

$O(n)$

eg:> $P(x) = 5x^4 + 3x^2 + 5$

=> 4

Note The no. of multiplication is equal to atmost degree of polynomial.

eg:-> $P(x) = x^3 + 2x^2 + 9x + 7$

$$P(x) = 7 + 9x + 2x^2 + x^3$$
$$= 7 + x(9 + 2x + x^2)$$
$$= 7 + x(9 + x(2 + x))$$

2

Date ____
Page ____

q.) $P(x) = 3x^5 + 4x^3 + 8x + 7$

A $P(x) = 7 + 8x + 4x^3 + 3x^5$

$= 7 + x(8 + 4x^2 + 3x^4)$

$= 7 + x(8 + x^2(4 + 3x^2))$

~~$= 7 + x(8 +$~~ 4 ⑤

Now

$t = x^2$

$= 7 + x(8 + t(4 + 3t))$

1 1 1 1 = ④

Note: If one temporary variable is allowed then no. of multiplication will decrease.

eg.) $P(x) = x^4 + 3x^2 + 8$

$P(x) = 8 + 3x^2 + x^4$

$= 8 + x^2(3 + x^2)$

$t = x^2$

$= 8 + t(3 + t)$

$1 + 1 = \underline{2}$

// "t" भी तो calculate करना पड़ेगा। इसलिए t भी जुड़ेगा।

L1 → learn again

classmate
Date
Page

Addr(A[i1][i2][i3][i4] = L0 + (i1 * D2 * D3 * D4 + i2 * D3 * D4 + i3 * D4 + i4) * w

= L0 + [i4 + i3 * D4 + i2 * D3 * D4 + i1 * D2 * D3 * D4] * w

= L0 + [i4 + D4 (i3 + D3 (i2 + i1 * D2))] * w

+ 1 = 4

★ Matrices:

n x n

	1	2	3	4
1	a_{11}	a_{12}	a_{13}	a_{14}
2	a_{21}	a_{22}	a_{23}	a_{24}
3	a_{31}	a_{32}	a_{33}	a_{34}
4	a_{41}	a_{42}	a_{43}	a_{44}

if (i=j) → Diagonal

if (i=1) → 1st row

if (i<j) → Upper triangle

if (i>j) → lower triangular

if (i-j=1) → diagonal below main diagonal

if (i-j=-1) → " above " "

if (i+j=n+1) → cross diagonal

if (i+j>n) → right lower triangle

gate Q) V is a ~~matrix~~ nxn

$\forall$ V[i][j] = i-j

then

sum of all elements = ?

11=0 12=-1

21=1 22=0

=0

Ans = 0

Ans

Q>

```
c = 100;
for ( i=1; i<=n; i++)
  {
  for ( j=1; i<=n; i++)
    {
      t = A [i][j] + c;
      A[i][j] = A[j][i];
      A[j][i] = t - c;
    }
  }
```

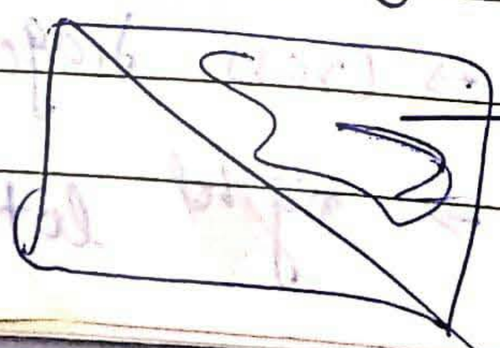
Ans i=1, j=1

t = 100

try by taking small matrix

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Note:



- transpose of a matrix can not be done within a matrix.

★ Special matrix:—

★ (1) Diagonal

(2) lower triangul matrix

(3) upper "

(4) Symmetric

(5 triplity

6) ~~tri-diagonal~~ tri-diagonal matrix

① diagonal:-

	1	2	3	4
1	2	0	0	0
2	0	5	0	0
3	0	0	8	0
4	0	0	0	10

A =

n x n

if ($i \neq j$) then A[i][j] = 0

No. of non-zero element = n

	0	1	2	3
A	2	5	8	10

if ($i = j$)

Addo (A[i][j]) = Lot [i-1] + ω

index

if ($i != j$)

0

```
class Diagonal
{
  int *A, n;
public
  Diagonal (int d)
  {
    n=d;
    A = new int[n];
  }
void set (int i, int j, int x)
  {
    if (i==j)
      A[i-1]=x;
  }
int get (int i, int j)
  {
    if (i==j)
      return A[i-1];
    else
      return 0;
  }
```

(2) lower triangular matrix

* 1 2 3 4

1	a_{11}	0	0	0
2	a_{21}	a_{22}	0	0
3	a_{31}	a_{32}	a_{33}	0
4	a_{41}	a_{42}	a_{43}	a_{44}

lower triang

n x n

if (i < j) then A[i][j] = 0

then No. of non-zero elements = $1+2+3+\dots+n = \frac{n(n+1)}{2}$

	0	1	2	3	4	5	6	7	8	9
A	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}
	row 1	row 2		row 3			row 4			

Row major :

Row - major

$$Addr(A[3][3]) = L_0 + \left[\underbrace{1+2}_{i} + \frac{2}{j}\right] * w$$

$$= L_0 + \underset{\text{index}}{5} * w$$

$$Addr(A[4][3]) = L_0 + \left[\underbrace{1+2+3}_{i} + \frac{2}{j}\right] * w$$

$$= L_0 + 8 * w$$

$$\boxed{Add(A[i][j]) = L_0 + [1+2+ \cdots + i-1 + j-1] * w}$$

$$= L_0 + \left[\frac{i(i-1)}{2} + j-1\right] * w$$

↳ row major formula for lower triangle

Now

Column major

	0	1	2	3	4	5	6	7	8	9
A	a_{11}	a_{21}	a_{31}	a_{41}	a_{22}	a_{32}	a_{42}	a_{33}	a_{43}	a_{44}
	col 1				col 2			col 3	col 4	

$$Addr(A[i][j]) = L_0 + [N + N-1 + N-2 + \cdots + N-(j-2) + i-j] * w$$

10×10

A[10][7] = Lo + 10+9+8+7+6+5

= Lo + 10 + 10-1 + 10-2 + 10-3 10-4+10-5

A[10][7] = Lo + 10 + 10-1

3) Lower Upper triangular matrix :-

	1	2	3	4
1	a_{11}	a_{12}	a_{13}	a_{14}
2	0	a_{22}	a_{23}	a_{24}
3	0	0	a_{33}	a_{34}
4	0	0	0	a_{44}

if $(i > j)$ then $A[i][j] = 0$

No. of non-zero elem : $\frac{n(n+1)}{2}$

column major

$$Addr(A[i][j]) = Lo + \left[\frac{j(j-1)}{2} + (i-1)\right] * w$$

Row major $Addr(A[i][j]) = Lo + [N + N-1 + N-2 + \cdots N-(j-2) + j-i] * w$

Note: column major of lower triangle is same as row major of upper triangle.

&

Row " " " " "

" column " " "

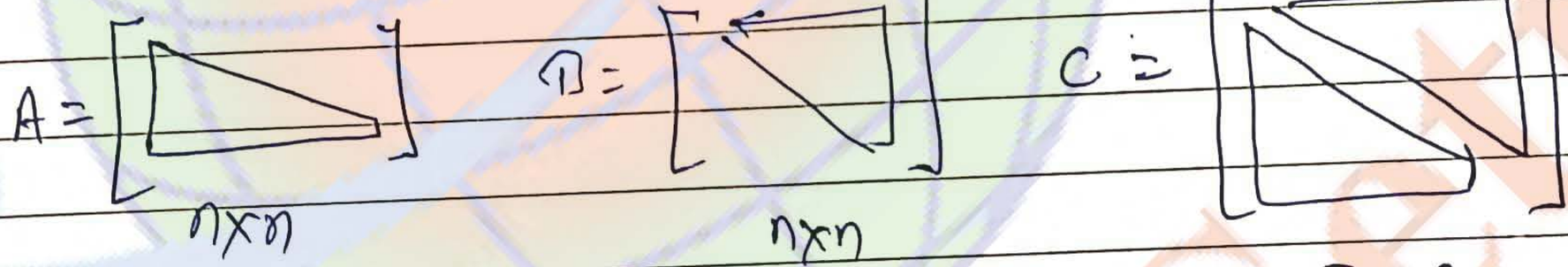
4) Symmetric matrix:-

	1	2	7	4
1	2	4	8	10
2	4	0	5	7
7	8	5	9	10
4	10	7	10	15

If $(A[i][j]) == A[j][i]$

$$\frac{n(n+1)}{2}$$

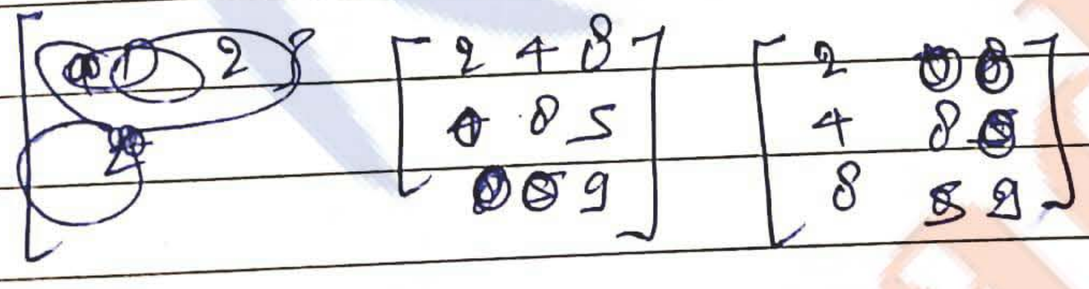
Sol Q:-



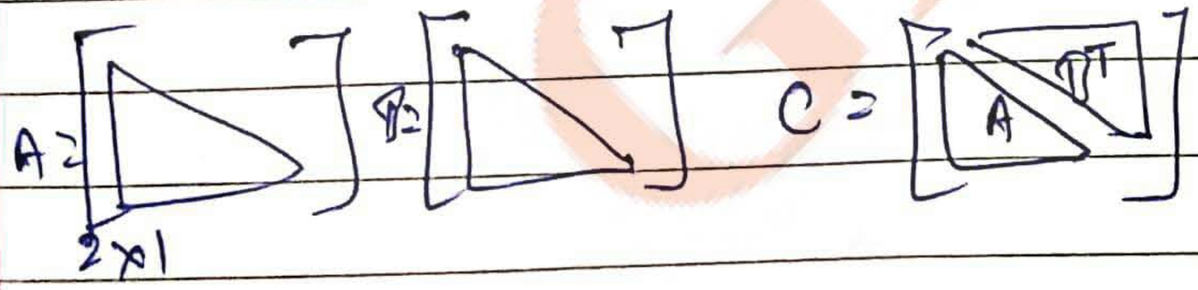
$\Rightarrow m \times (n+1)$

$A[i][j] = C[i][j]$

$B[i][j] = C[i][j+1]$



Q>

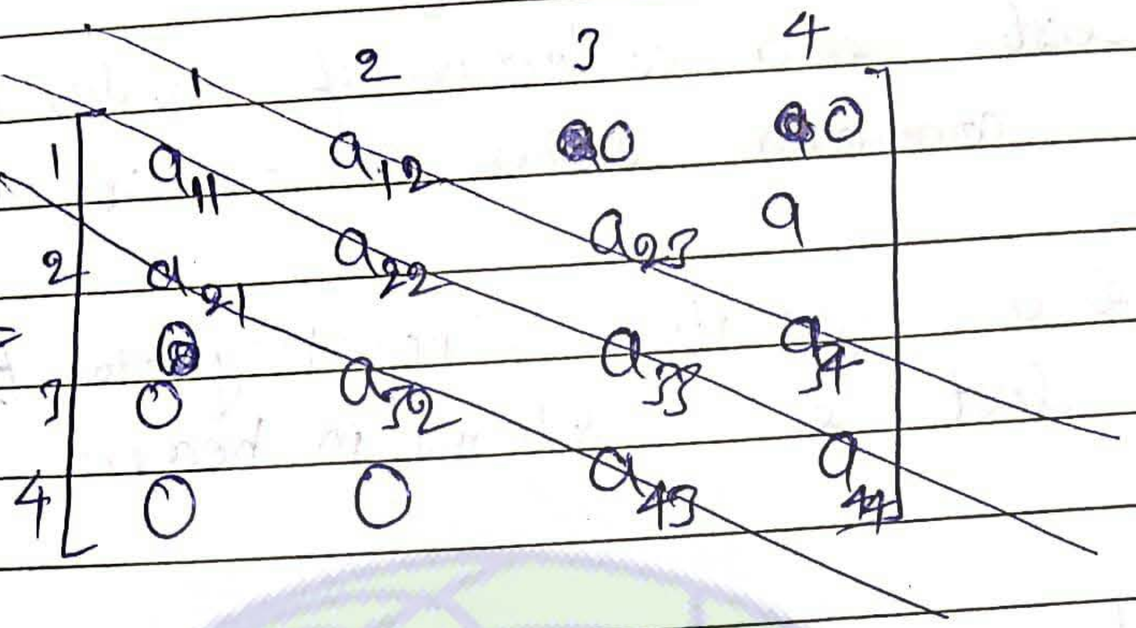


$A[i][j] = C[i][j]$

$B[i][j] = C[j][i+1]$

$m \times (n+1)$

(6) Tri-diagonal matrix' -



$i-j = \{-1, 0, 1\}$

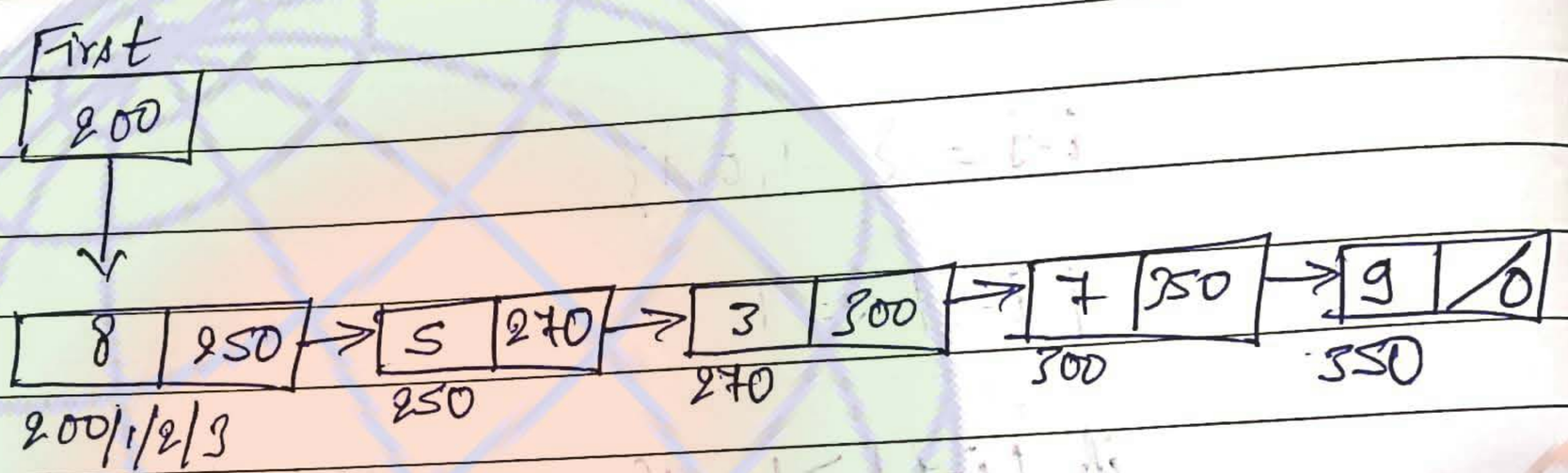
$|i-j| <= 1$

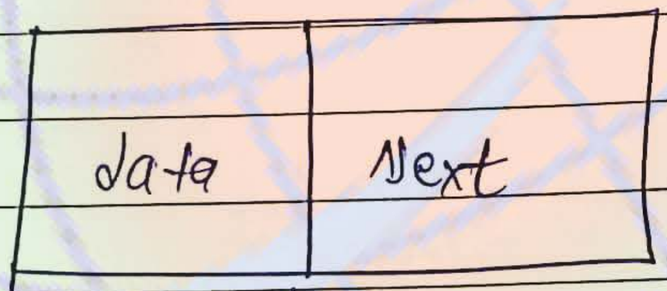
if $|i-j| > 1$ then $A[i][j] = 0$

Note: row major (1 formula)
col - major (1 formula)
diagonal - by - diagonal
(lower - diag + main diag + upper diag)
(3 formula)

Linked list

- Linked list is a collection of nodes, where each node contain data & pointer to next node.
- First is a pointer, Pointing to first node.
- Linked list is stored in heap.





struct Node
{
int data; —(1)
struct Node * next; —(2)
};

size = 4 bytes

↓

It's a self referential structure.

```
struct Node *first;
struct Node c;
c.data = 10;
c.next = NULL;
```

c

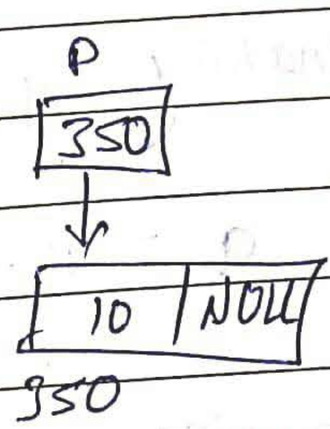
10	NULL

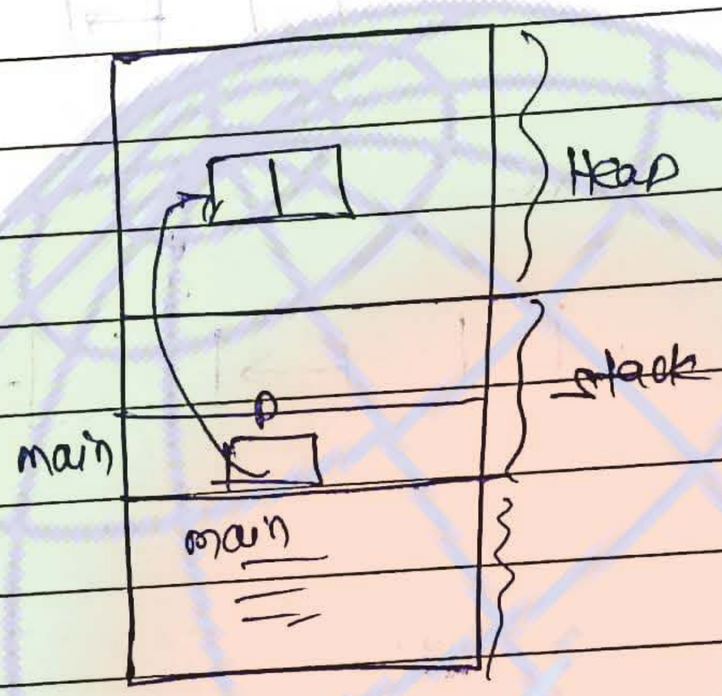
(3)

— Creating and accessing a node in stack.

creating a node in heap

```
main()
{
struct Node *p;
p=(struct Node *)malloc(size of (struct Node));
p -> data = 10;
p -> next = NULL;
```





Note In C++, we use

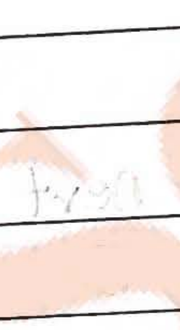
C++

```
struct Node *p;
p = (struct Node ---
p->
p->
p = new node
```

(2) *

```
struct Node *p;
```

Not pointing
```
if (p==NULL)
if (!p)
```

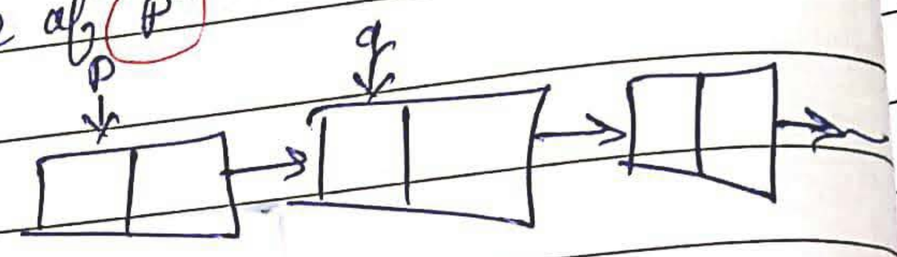


pointing
```
if (p!=NULL)
if (p)
```

3) struct node *p;

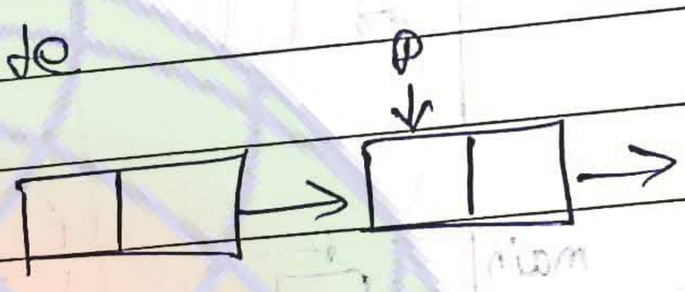
//making q point on next node of p

q = p -> next;



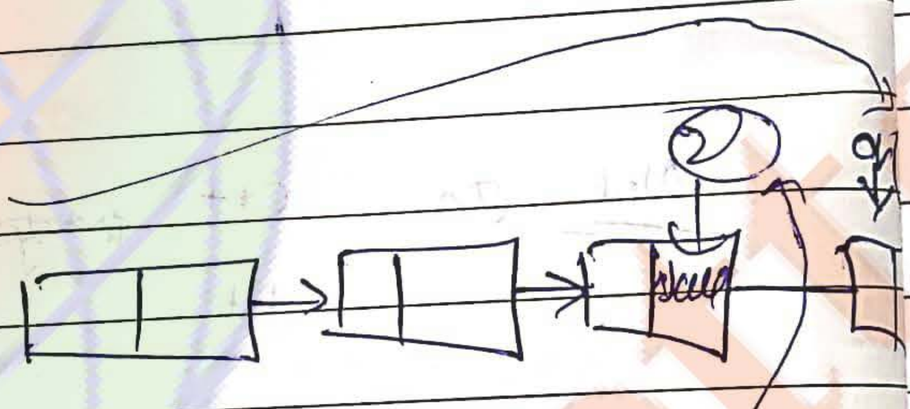
//moving p to next node

p = p -> next;



4) struct node **p;

(1) q = p -> next -> next -> next;
read



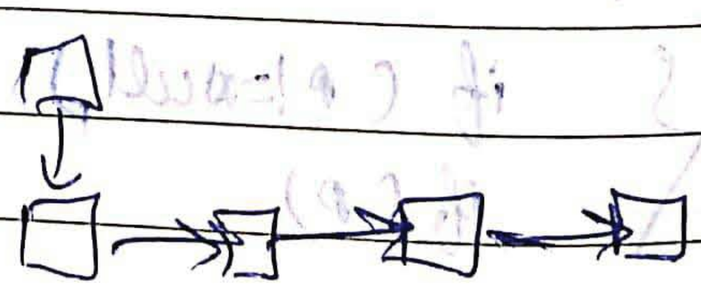
(2) p -> next -> next -> next = NULL;
write

(Addr) / L-value / write = (data) / r-value / read

write = read

5) Traverse/display linked list:-

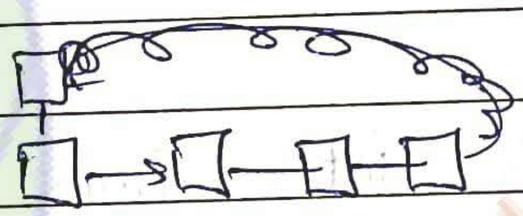
struct Node *p;
p = first;
while (p != NULL)
{



```
Printf ("%d", P -> data);
P = P -> next;
}
```

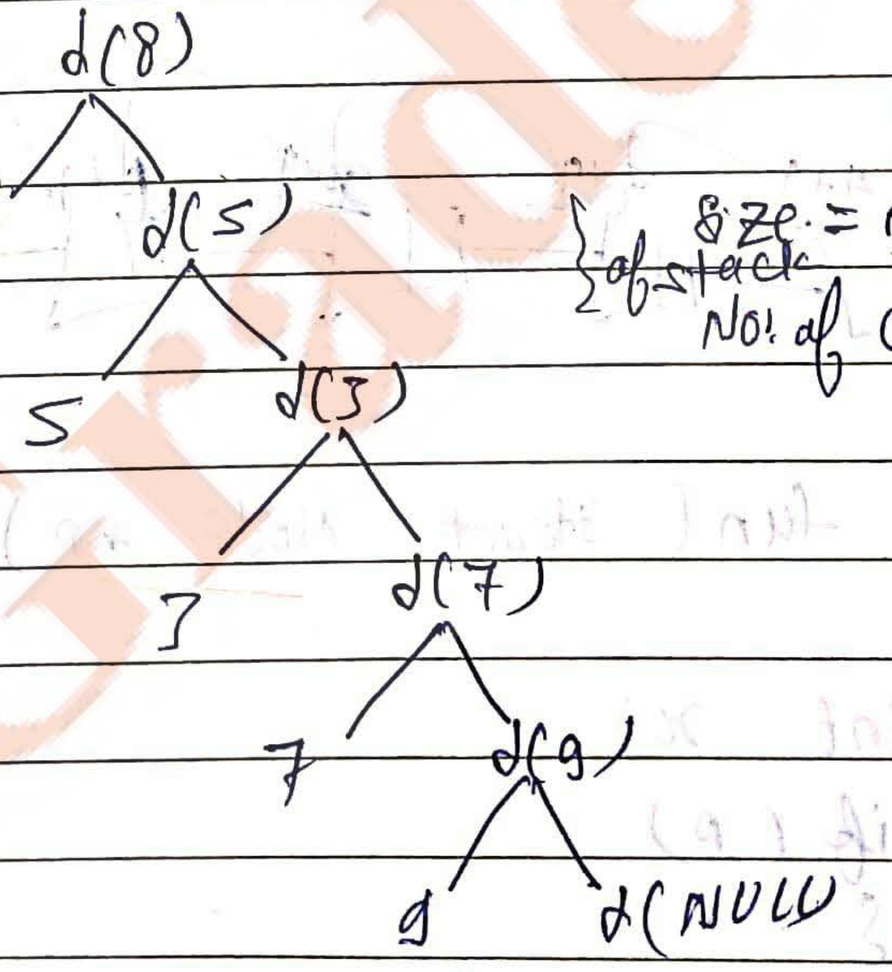
* For n-loop, body of loop will repeat for n-times.
* And condition will be checked for n+1 times.

6)
```
display (struct Node *P)
{
  if (P != NULL)
  {
  Printf ("%d", P -> data);
  display (P -> next);
  }
}
display (first);
```



time function
↓
$f(n) = n+1$

$O(n)$



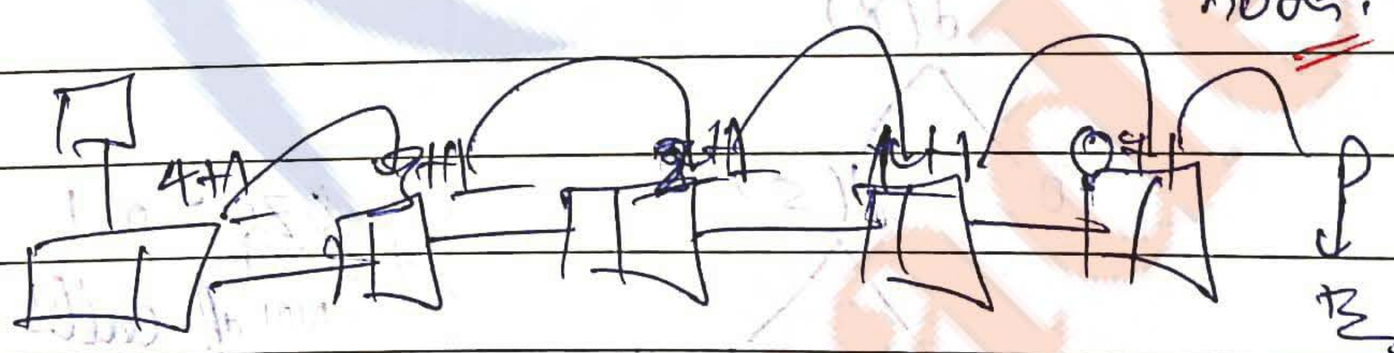
Size = n+1
No. of cells = n+1

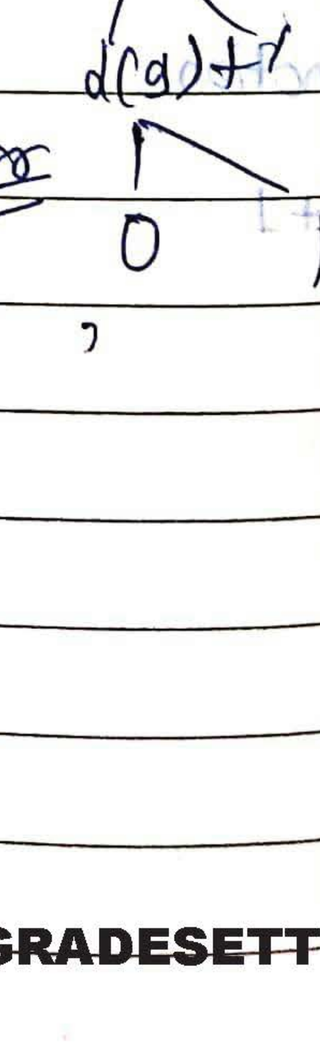
eg:- display (struct Node *p)
{
if (p != NULL)
{
display(p -> next);
pf(p -> data);
}
}

This will display the linked list in reverse

eg:- int fun (struct Node *p)
{
if (!p)
return 0;
return fun (p -> next) + 1;
}

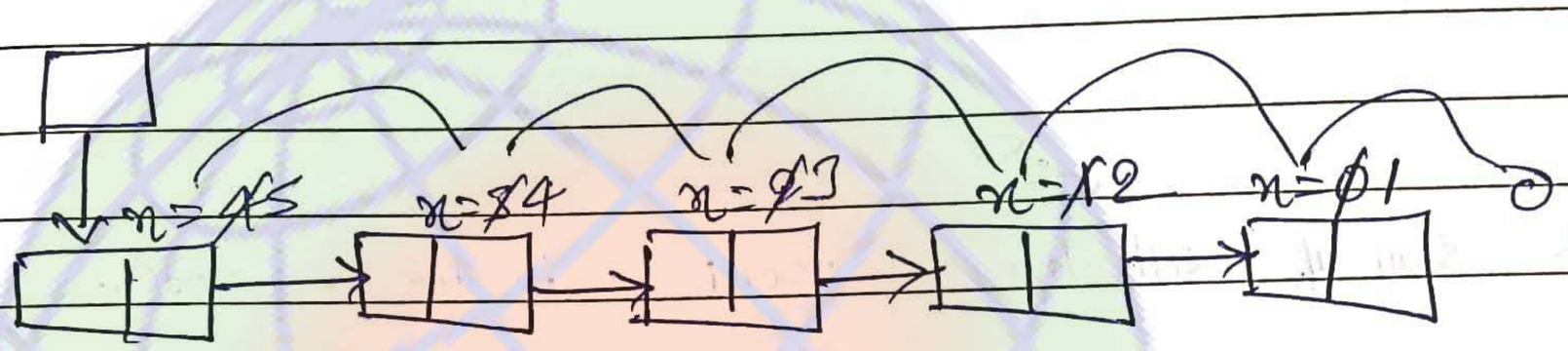
Ans 5 , This is for counting node.





eg:- int fun (struct Node *p)
{
int x;
if (p)
{
x = fun (p -> next);

```
x++ ;
return x ;
}
return 0 ;
}
```



This is also counting no. of nodes

or

$S = f(8) = x++ = 5$

$x = f(5)$ $x++ = 4$

$3 = x = f(3)$ $x++ = 3$

$2 = x = f(2)$ $x++ = 2$

$1 = x = f(1)$ $x++ = 1$

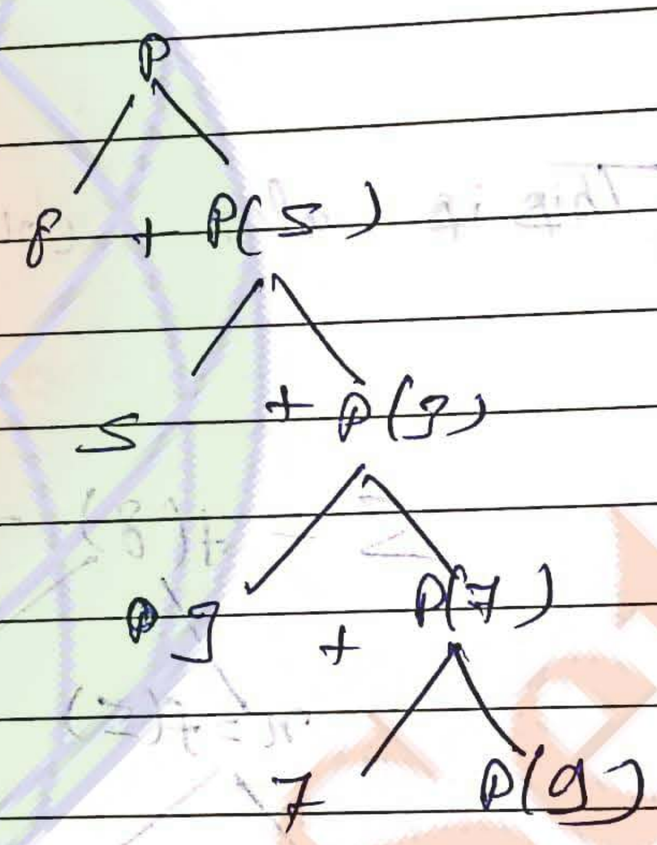
$0 = x = f(0)$

0

eg:-

```
int fun ( struct Node *P)
{
  if (P)
    return fun ( P -> next) + P -> data;
  return 0;
}
```

Ans Sum of all the present in the k nodes.



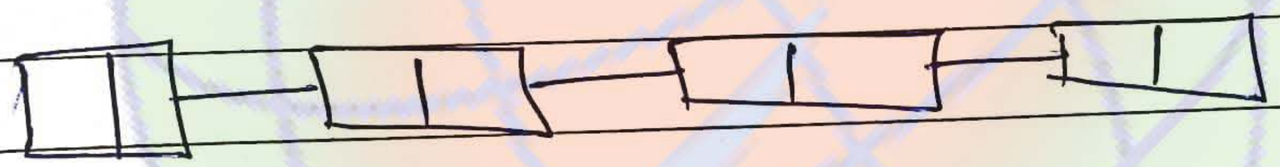
or

```
int fun (struct Node *P)
{
  int x;
  if (P)
  {
    x = fun ( P -> next
    x = x + P -> data;
    return x;
  }
  return 0;
}
```

Linear Search:-

```
int fun (struct Node *p, int x)
{
    if(p)
    {
        if (p->data == x)
            return p;
        return fun(p->next, x);
    }
    return NULL;
}
```

A A for searching the data



minimum call = 1 (for first)
maximum calls = n+1 (not in the)
maximum calls = n (if n the last)

best O(1)
worst O(n)

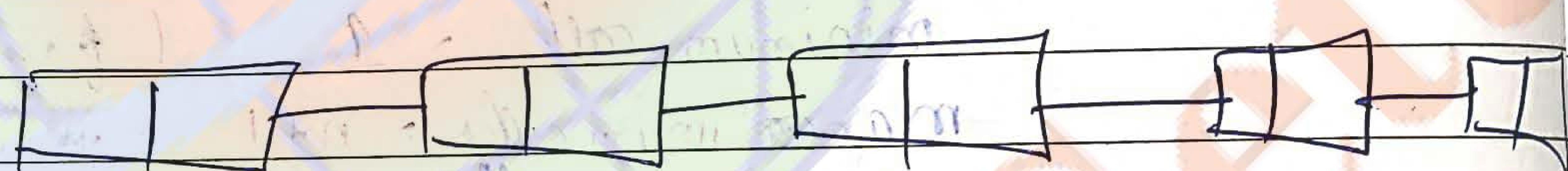
Counting occurances of x

```
int fun (struct Node *p, int x)
{
    if(p)
    {  if (p->data == x)
            return fun(p->next, x)+1;
        return fun(p->next, x);
```

}
return Null;
}

3.)

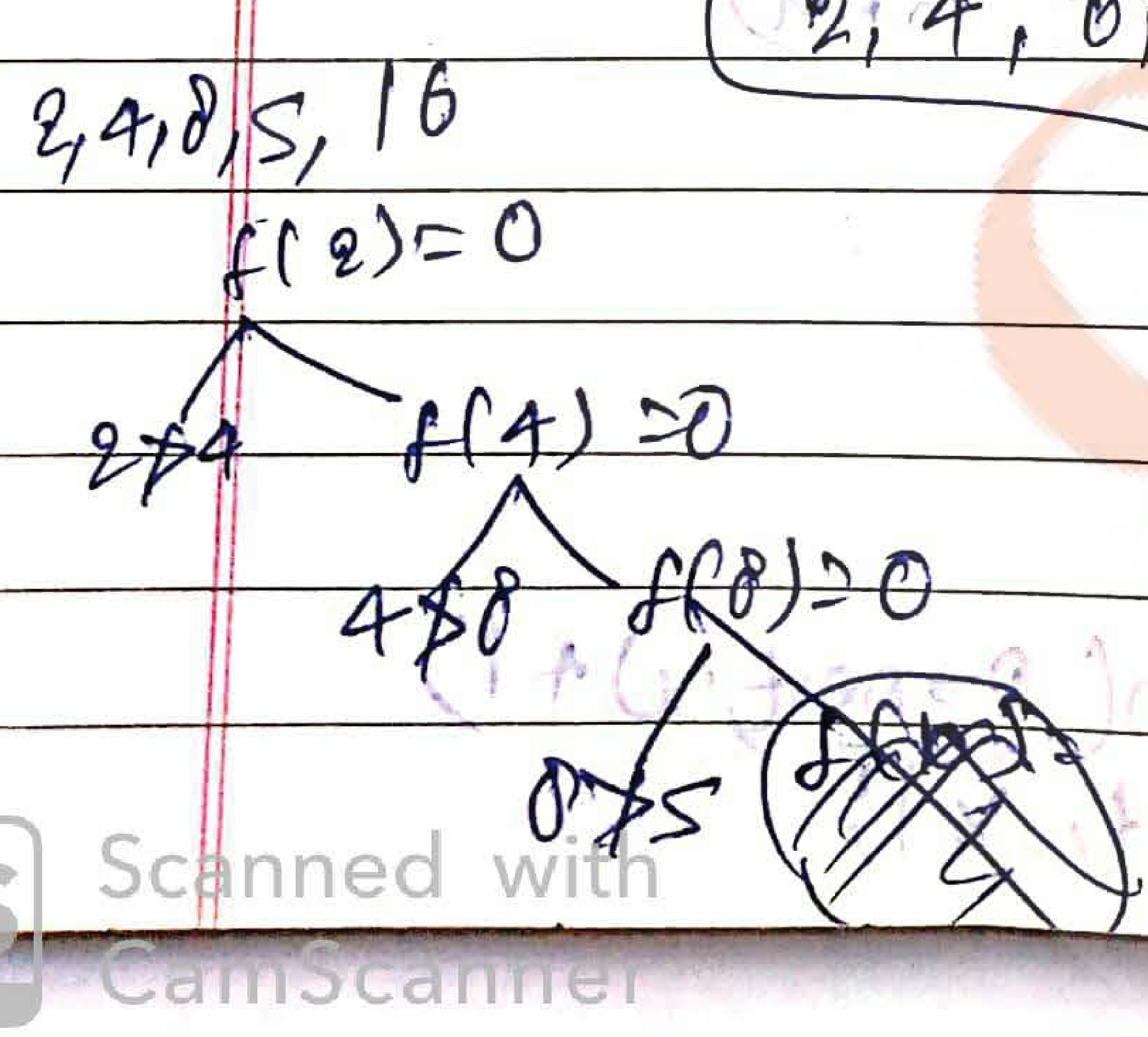
```
int fun(struct Node *p)
{
    if (!p)
        return 1;
    if (p -> next && p->data > p -> next ->data)
        return 0;
    return fun(p -> next);
}
```

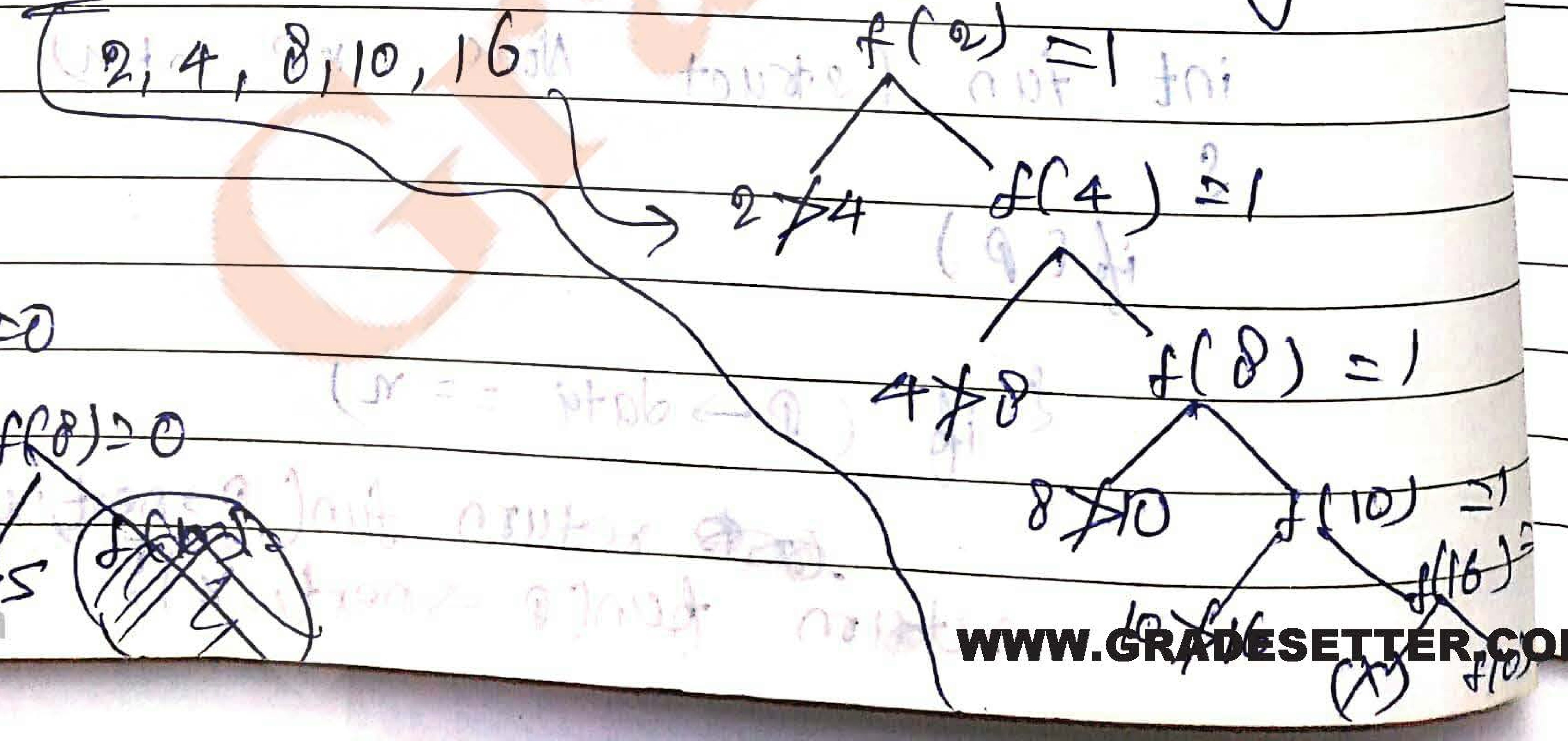


A ~~shorted in ascending order.~~

~~//checking the~~

//Contradictory checking of ensuring the number is in ascending order.





4) int fun(struct Node *p)
{
if (!p)
return 0;
x = fun (p -> next);
return x > p -> data ? x : p -> data;
}

A~ Hint> c = a > b ? a : b

if (a > b)
{
c = a;
}
else
{
c = b;
}

if (x > p -> data)
return x;
else
return p -> data;

This is for finding greatest number.

- The first value recores in x is 0, from the last call.

GATE
S:> struct Node *p, *q;
p = first;
q = first -> next;
while (p && q)
{
t = p -> data;
p -> data = q -> data;
q -> data = t;
p = q -> next;
q = p ? p -> next : NULL;
}

Ans 1 thing

```
if (P! = NULL)
    q = p -> next;
else
    q = NULL;
```

Ans 5 → 8 → 7 → 3 → 9

(a)
```
P = first;
while (P -> next)
{
    P = P -> next;
}
```

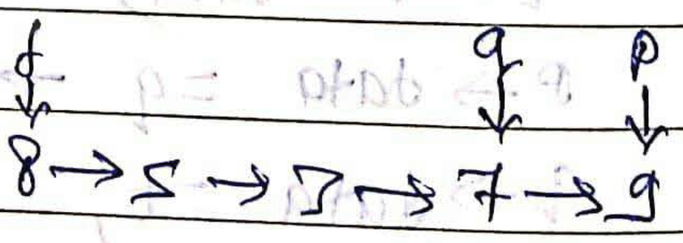
//making P more till last node.//

(b)
```
P = first;
while(P)
{
    q = P;
    P = P -> next;
}
```

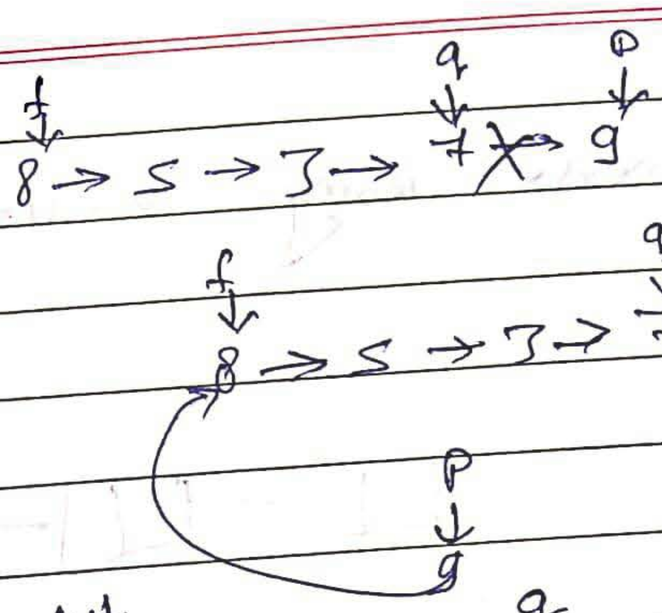
//taking q till last node

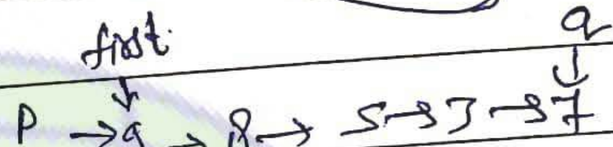
(c) more to head

```
P = first;
while (P -> next)
{
    q = P;
    P = P -> next;
}
```



q -> next = NULL;
p -> next = first
first = p;





★ Reversing of link list -

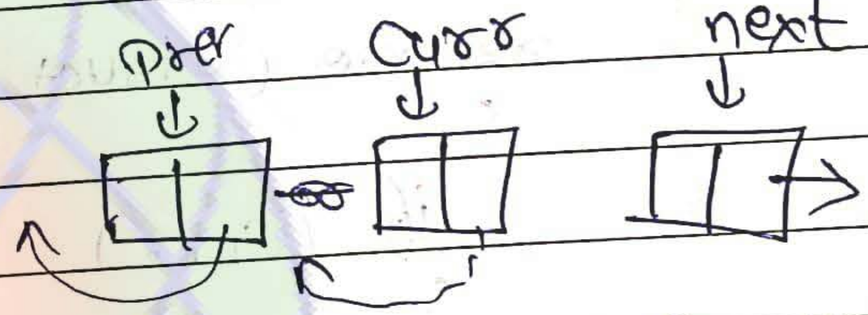
gateQuest

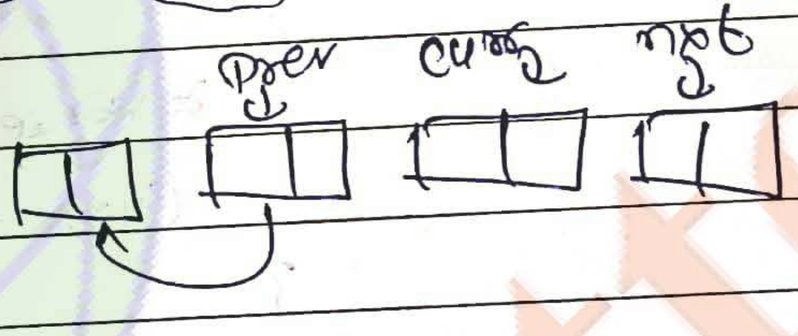
a)

```
Prev = NULL;
curr = first;
next = first -> next
while (curr != NULL)
{
  { curr -> next = prev;
    prev = curr;
    curr = next;
    next = next ? next -> next : NULL;
  }
}
first = Prev.
```

reversing the link

slide





or

```
Prev = NULL;
Curr = NULL
next = first;
while (next != NULL)
{
  slide { prev = curr;
          curr = next;
          next = next -> next;
  reverse the link { curr -> next = Prev;
  }
}
first = curr;
```

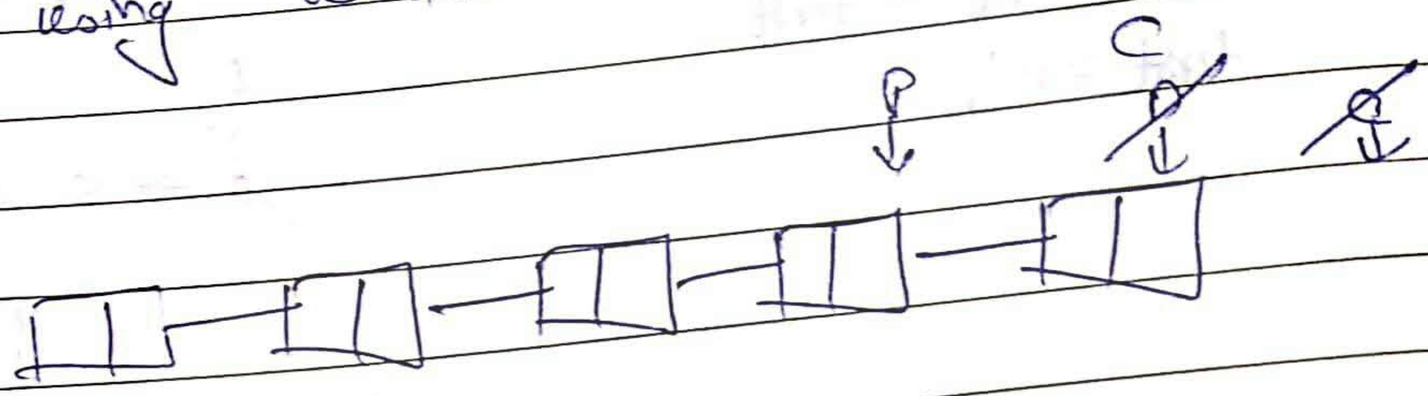
// In terms of space, loop is better.

// Here firstly the sliding then reversing the link is taken place.

// three pointers are required for reversion.

(test) Total swaps

* Reverse using recursion:-

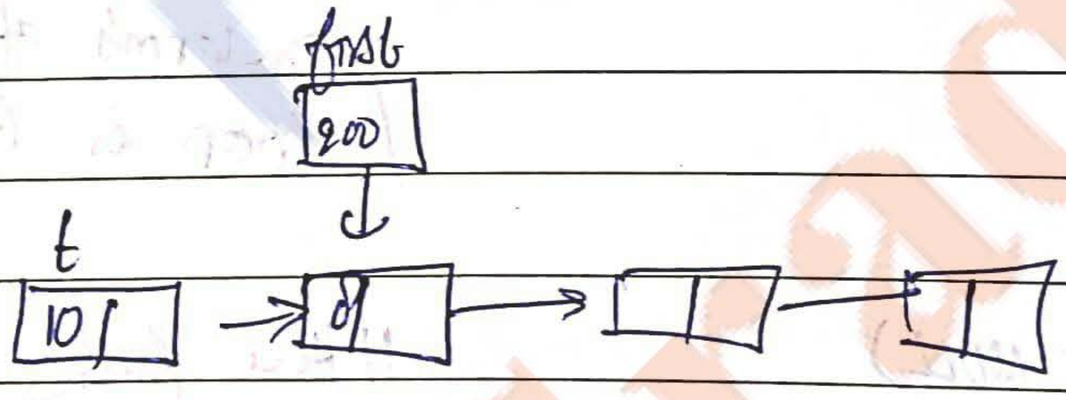



```
reverse (NULL, first);
reverse (struct Node *Prev, struct Node *Curr)
{
  if (Curr)
  {
    reverse (Curr, Curr->next);
    Curr->next = Prev;
  }
  first = Prev;
}
```

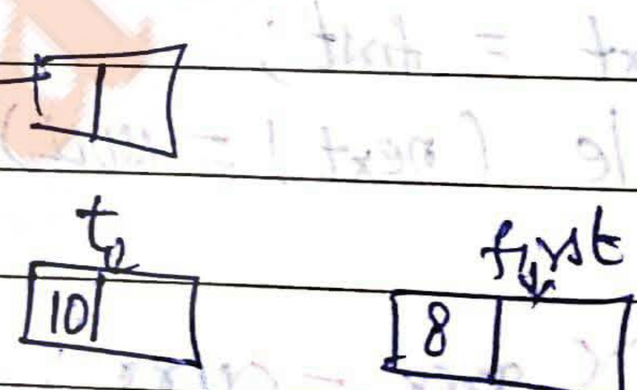
// two pointers are required.
Total pointer = 2x(n+1)

★ Insert operations:-

(1) Inserting before 1st node:-



f(n)=4 t = new node;
O(1) t->data = x;
O(n°) t->next = first;

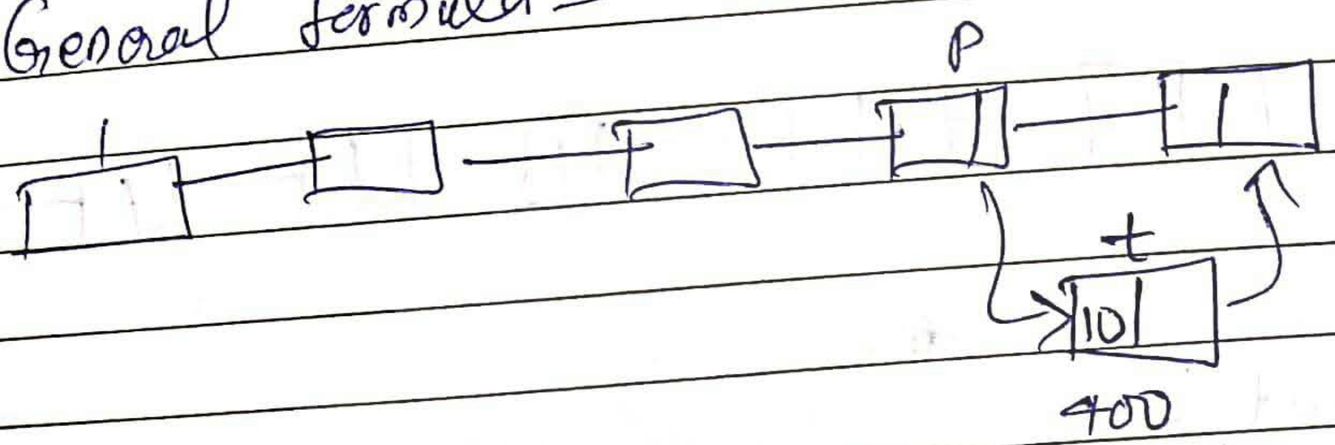


first = t;

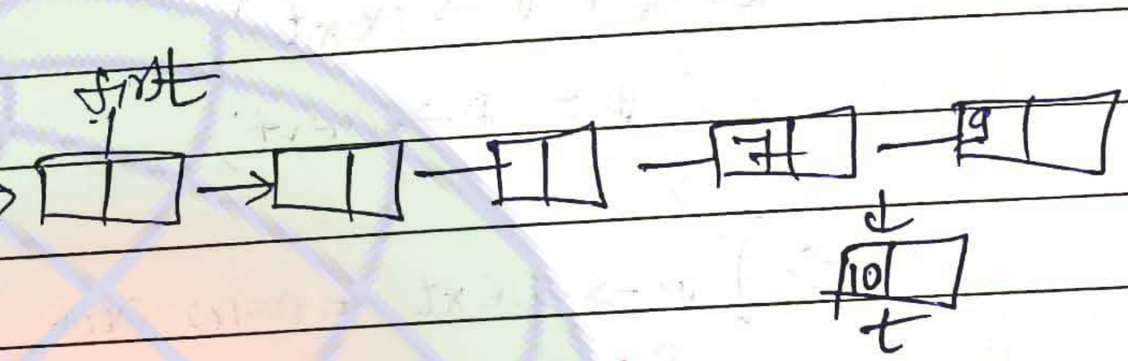
// one extra pointer is required.

Note Insertion at nodes, will also work like

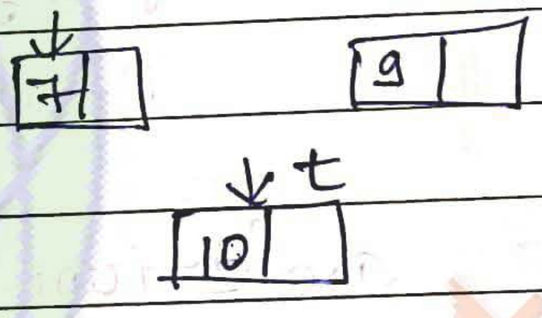
2) General formula :-

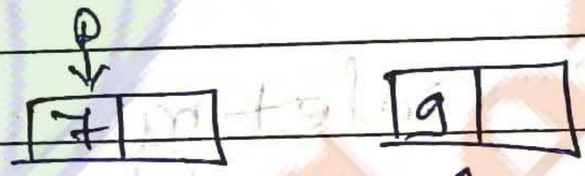


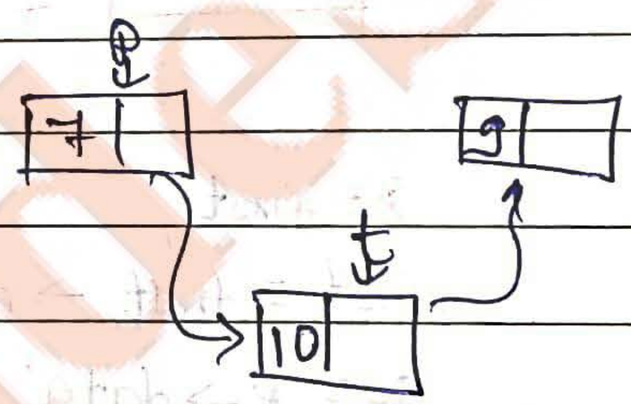

```
t = newNode;
t -> data = x;
```




```
p = first
for (i=0; i<pos-1; i++)
{
    p = p -> next;
}
t -> next = p -> next;
p -> next = t;
```





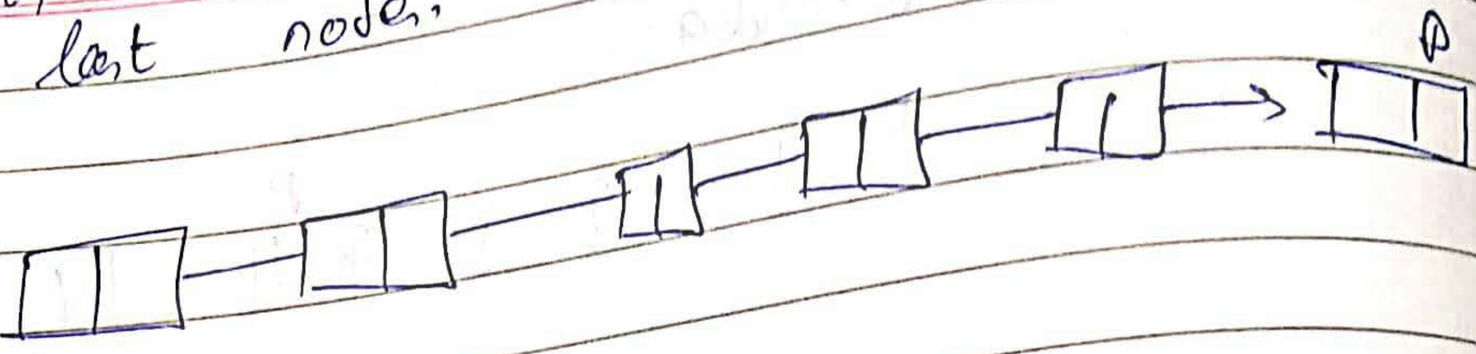


// two extra pointer is required here
p -> for links
t -> for creating new nodes.

// time: O(n)

Note: This will also work on adding nodes on the last node.

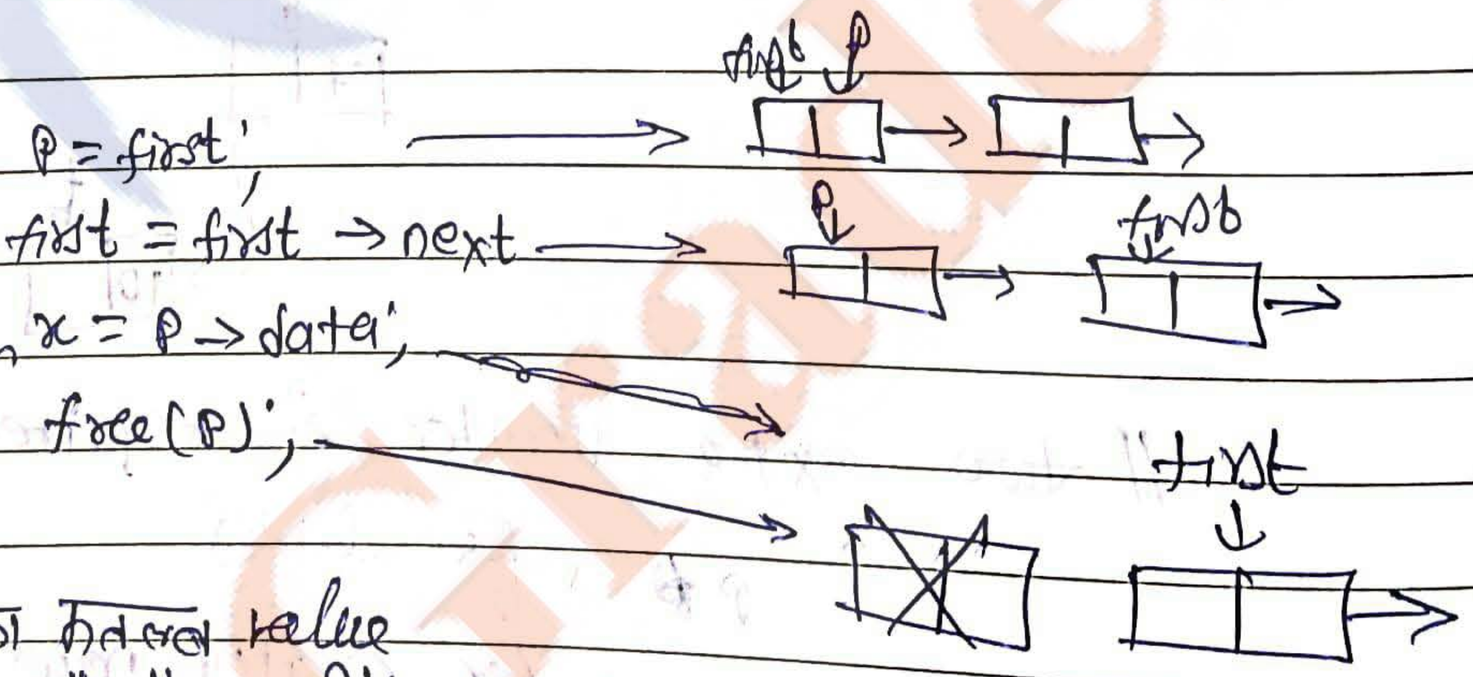
3.> Insertion
At last node.




```
first = p
while (p -> next)
{
    p = p -> next;
}
-> [p -> next = new node;
    p = p -> next;
    p -> data = x;
    p -> next = NULL;
```

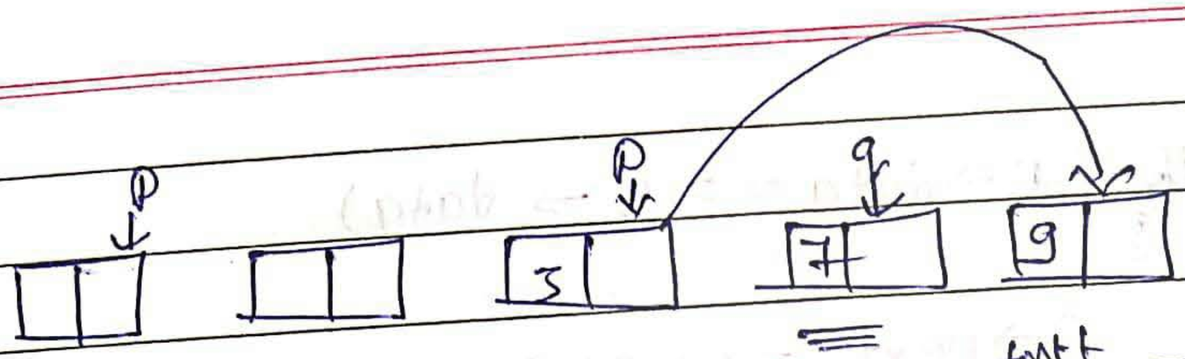
★ Deletion:-

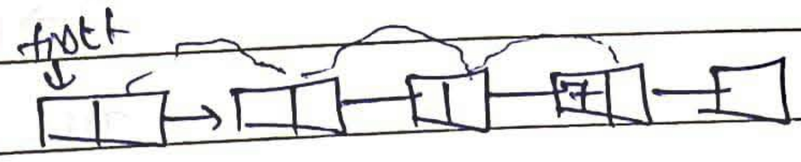
(1) Deleting first node.



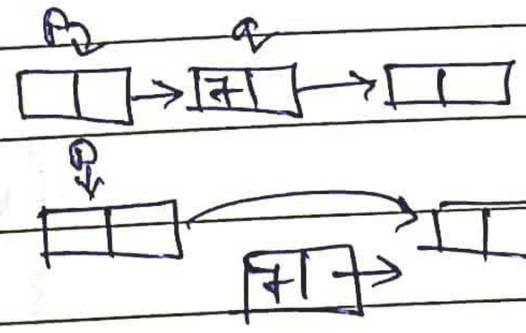
// One extra pointer is required
// O(1) time

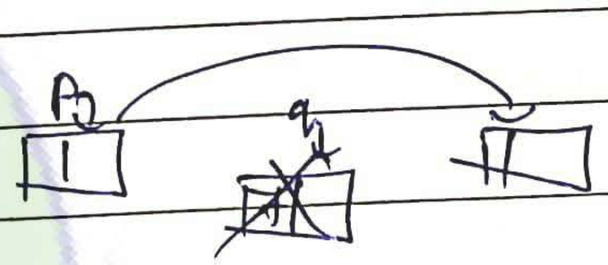
(2)






```
p = first;
for(i=0; i<pos-2; i++)
    p = p->next;
q = p->next;
p->next = q->next;
x = q->data;
free(q);
```





// No. of pointers = 2 ✓

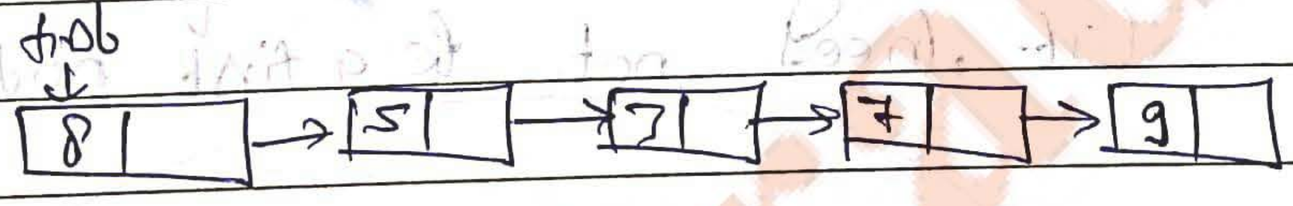
// time : O(n) → Here 'n' is position. ✓
(linear ✓

// links modified = 1

3) ~~Last node~~

Note: First node, second node and last node can be deleted using single pointer.

eg:



// The code will
Delete duplicates from sorted linked list

3→5→5→10→12→12→12→16→16→20

```
fun()
{
  p = first;
  q = first->next;
  while(p&&q)
  {
```

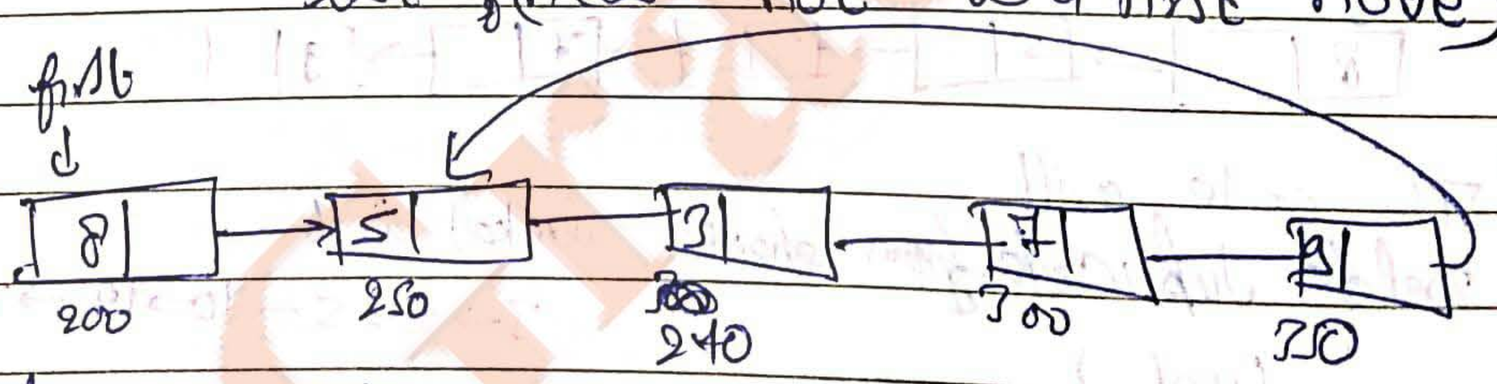
```
if ( p->data == q -> data)
{
p->next =q-> next;
free (q);
 q = p->next;
}
else
{
p=q;
q=q->next
{
}
}
```

// time: O(n)

☆ Circular linked list.

① ★ Loop:-

If last node ~~is pointing~~ is pointing on any other nodes of a linked list. (need not be a first node)



② Algorithm for finding, the given linked list is linear or having the loop.

(a) Store the address of newly visited node and

Visit becomes Null

check where the nodes ei visited or not

time = $O(n^2)$

Addr	200	250	240	700	750

n nodes
searching for n nodes:
↑ linear O(n)
↑ binary search O(log n) (apply on only shorted order)

Note: If we know that the addresses are in shorted order, then we will use binary
$O(n \log n)$

(a) ~~Hash tables~~

(b) Hashing

h(x) = 0
O(1)

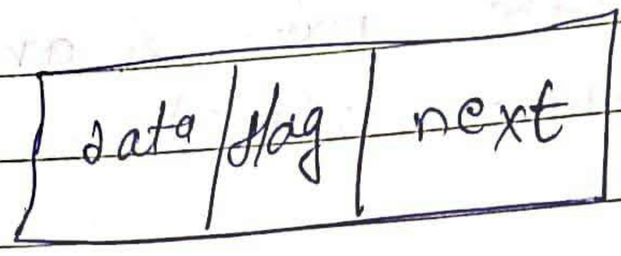
200
250
240
700
750

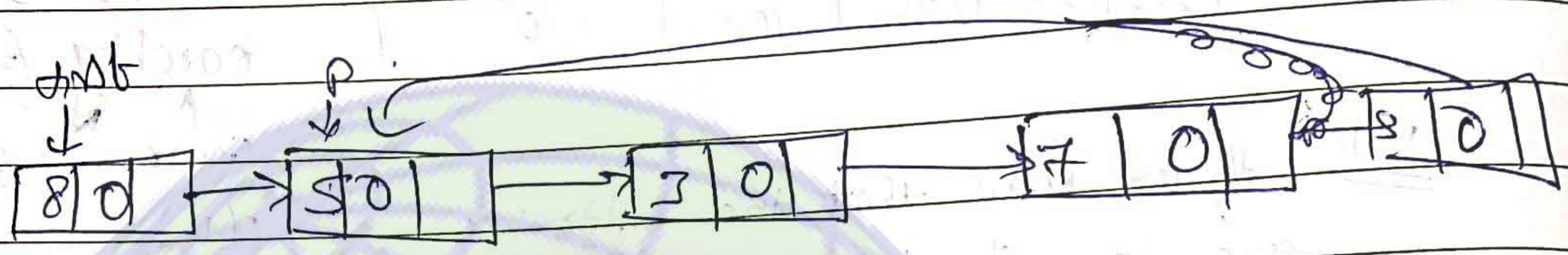
hash table

// time : $O(n)$

Note:
$n \times n \rightarrow$ linear
$n \times \log n \rightarrow$ Binary, demerit ⇒ addres should be shorted.
$n \times 1 \rightarrow$ Hash, demerit ⇒ hashing consume lot of space

(c)

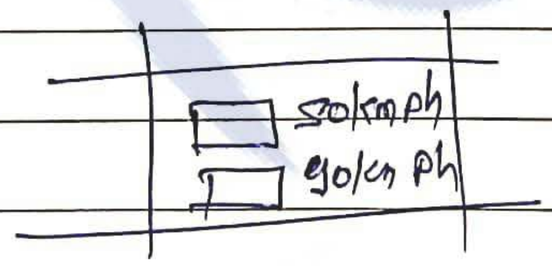


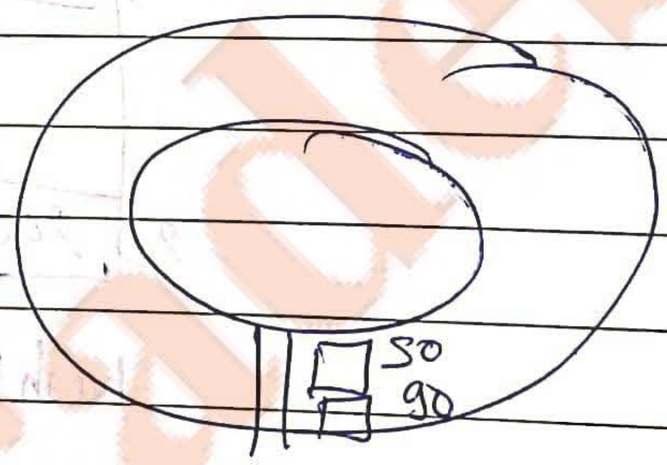



```
struct Node
{
int data;
struct Node * next;
 int flag;
};
```

size = 6 byte

(d) floyd's algorithm for circular path -





// time: O(n)

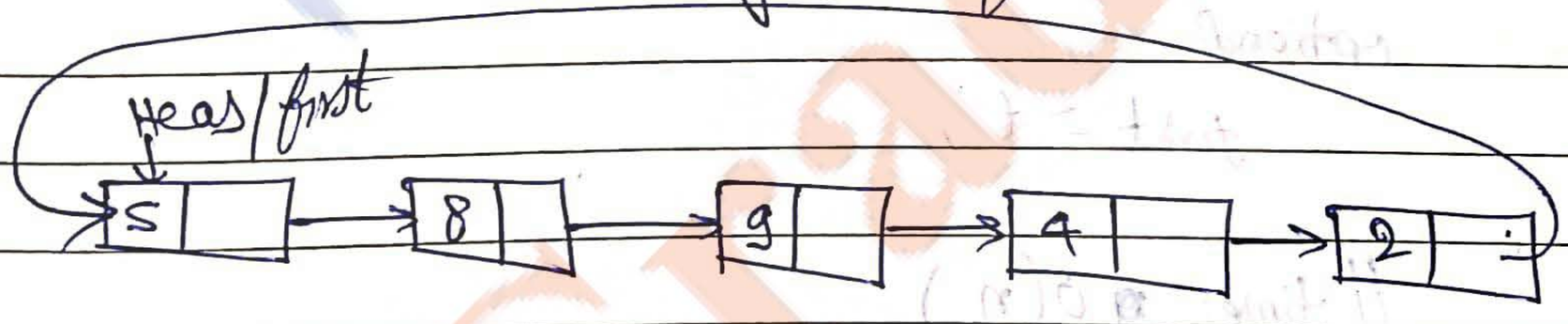
// one pointer is faster and the other is slower, as it's circular, they will meet at one time //

// If ~~null~~ null, it means linear

```
p = q = first;
do
{
p = p -> next;
q = q -> next;
q = q ? q -> next : q;

} while (p != q && p != q);
if (p == q)
    pf ("Loop");
else
    pf ("linear");
```

★ Circular linked list:-

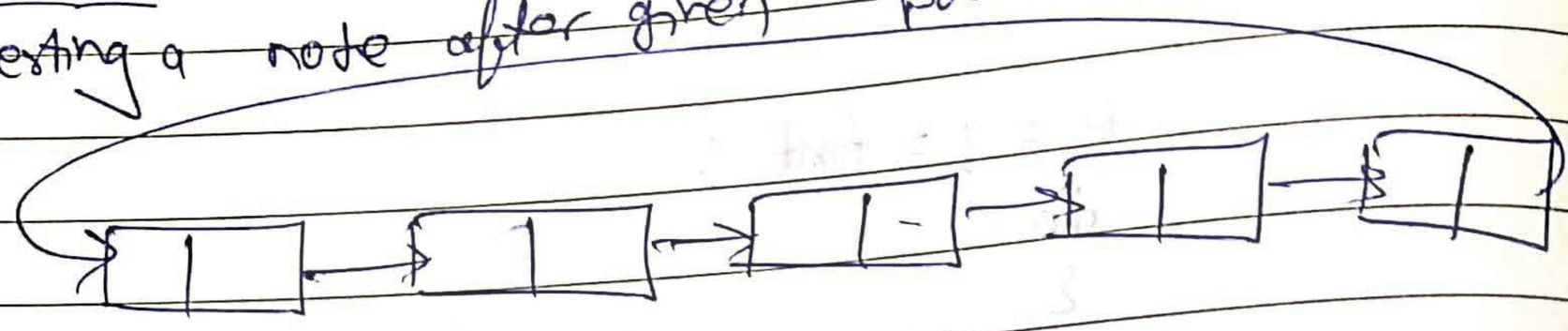
- circular linked list is a loop where Last node is pointing on first node.




```
p = first
do
{
printf ("%d", p -> data);
p = p -> next;
}
while (p != first);
```

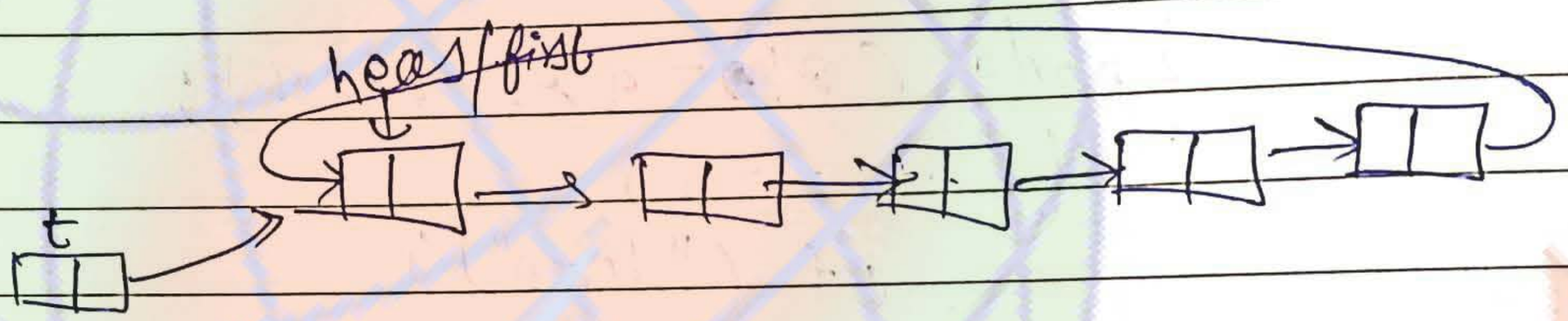
(1) Insertion:

(a) Inserting a node after given position



// procedure is same as inserting in a linear linked list.

(b) Inserting before head node :




```
p=first;
while(p->next!=first)
    p=p->next;
t=new node;
t->data=x;
t->new=first;
p->next=t;
optional
    first=t;

// time: O(n)
```

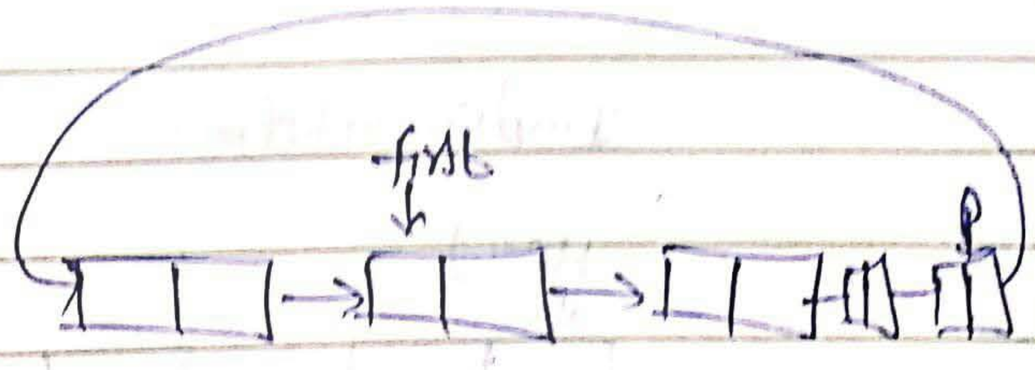
(c) Inserting after first node takes constant time. //

(2) Deletion:

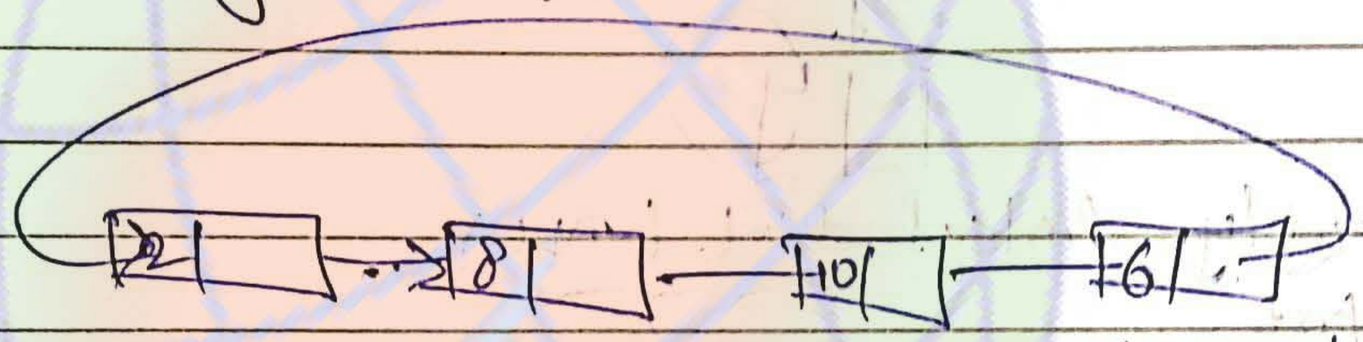
(a) // procedure is same as linear linked list.

(b) Deleting first node:-

```
p = first;
while (p != first)
    p = p->next;
x = first->data;
first = first->next;
free (p->next);
p->next = first;
```

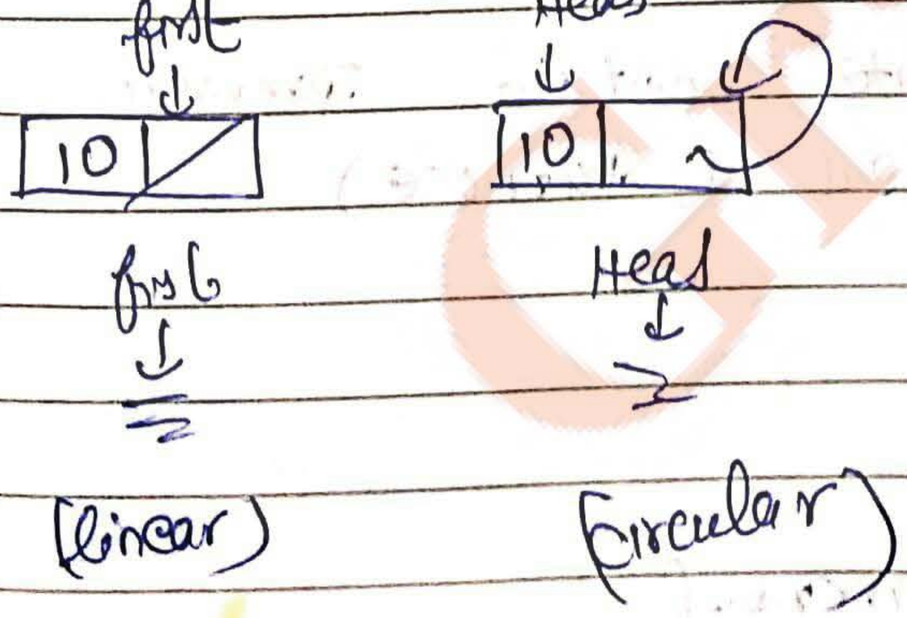


(c) Deleting after first node takes constant time.

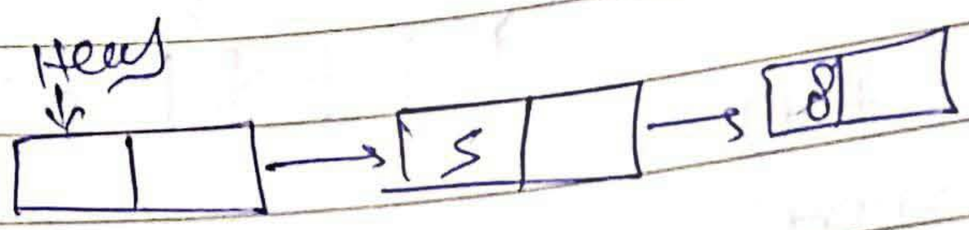


- If a circular linked list is used for implementing linked list already having some elements
- there are no pointers on linked list.
- If a pointer to is given, then where it should point such insertion and deletion in a queue takes constant time

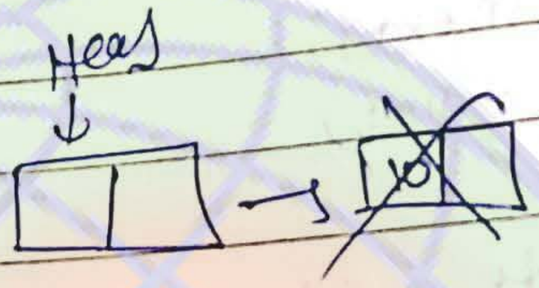
* -

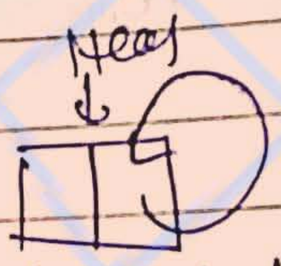


Implementation:

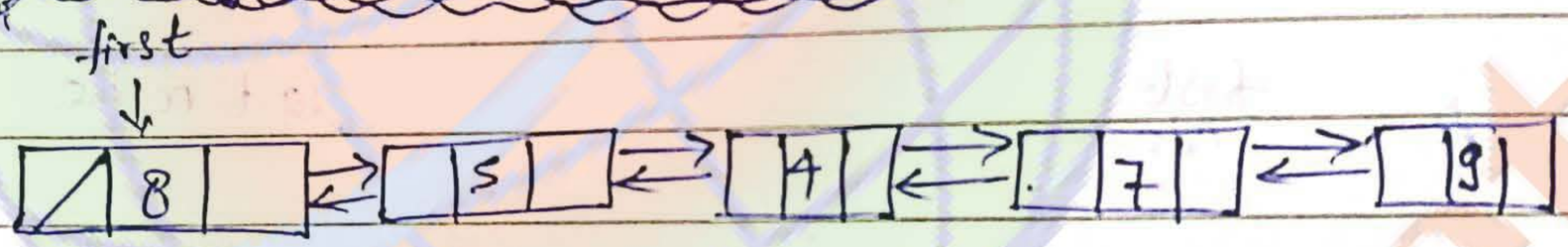


(extra heas)





Doubly linked list (DLL)




```
Struct Node
{
struct Node *Prev;
int data;
struct Node* next;
};
```

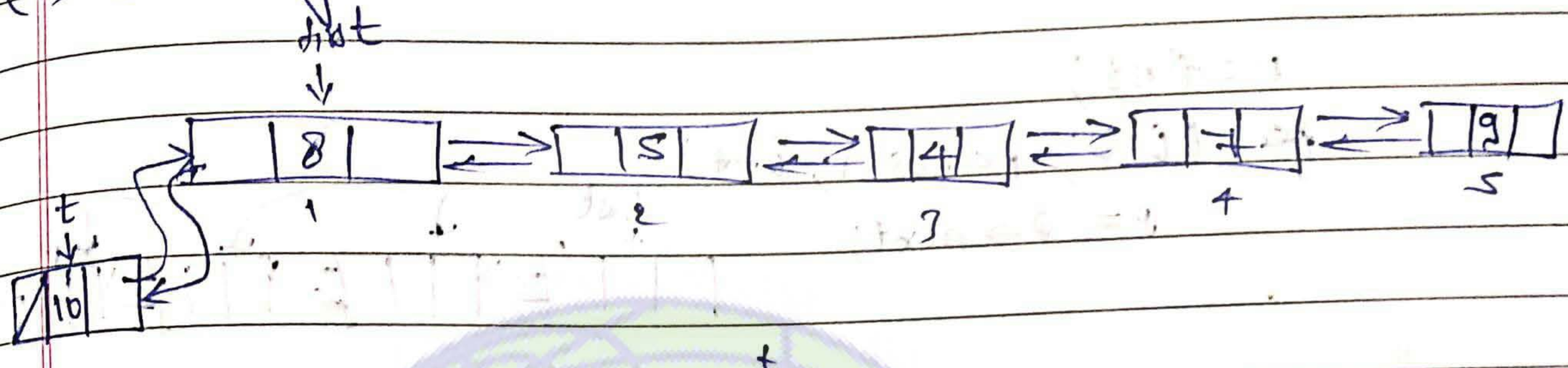
- Every node will point to Previous.
- Bidirectional traversal (Purpose)

Note:

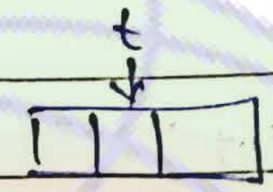
circular link list
↓
Purpose
↓
circular traversal

(i) Insertion:-

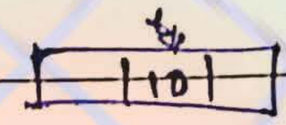
(a) Inserting at first position:-



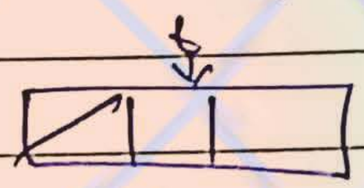
t=new Node;



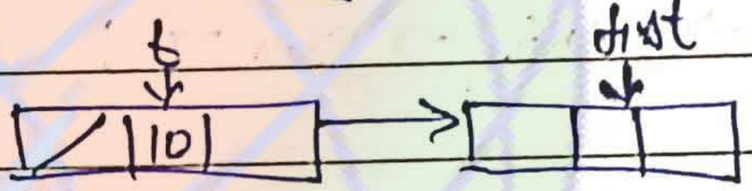
t->data = x;



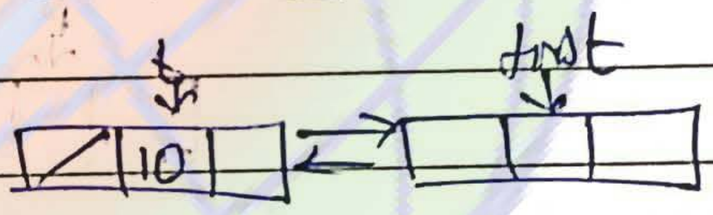
t-> Prev = NULL;



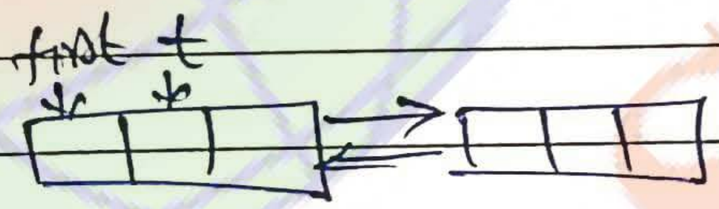
t-> next = first;



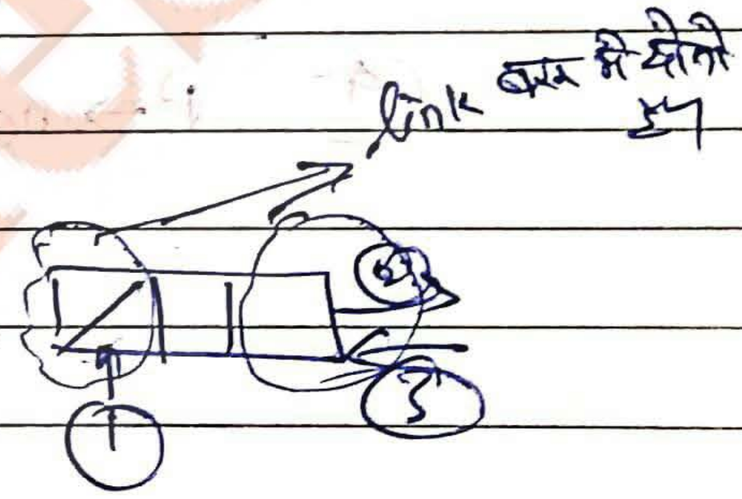
first-> Prev = t;



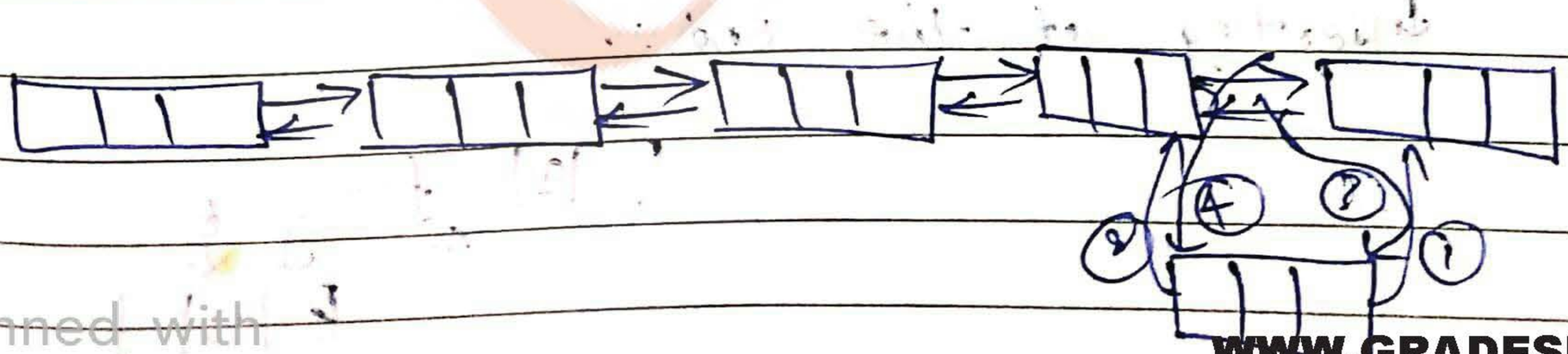
first = t;



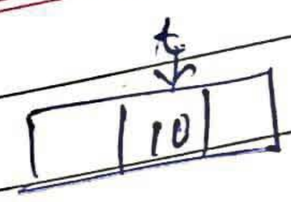
// time: O(1)
// one extra pointer
// links modified: 3

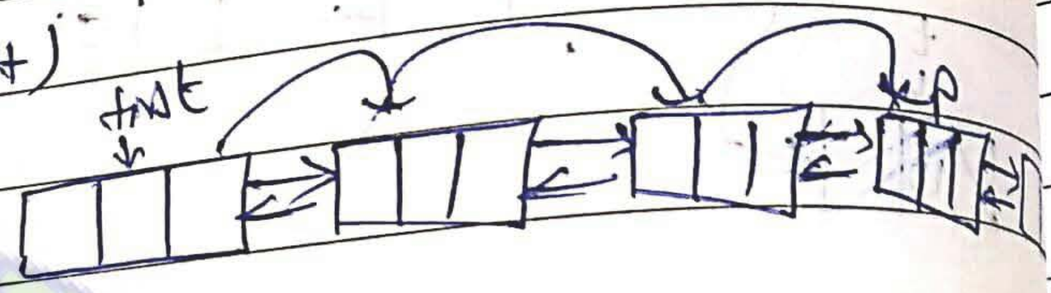


(b) Inserting a node at any position

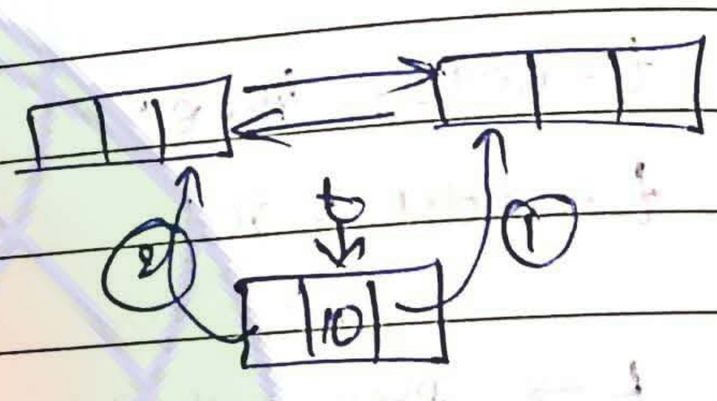


```
t=new Node;
t->data=x;

p=first;
for(i=0; i<pos-1; i++)
    p= p->next;
```

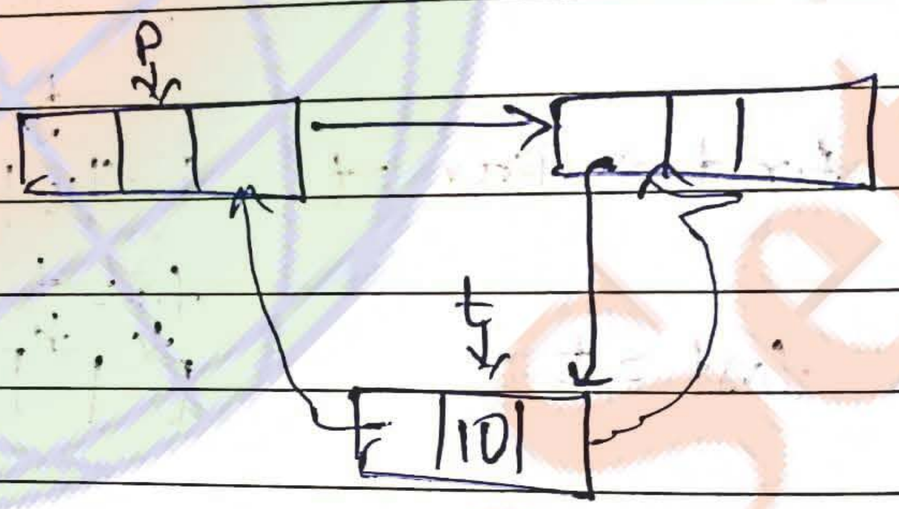




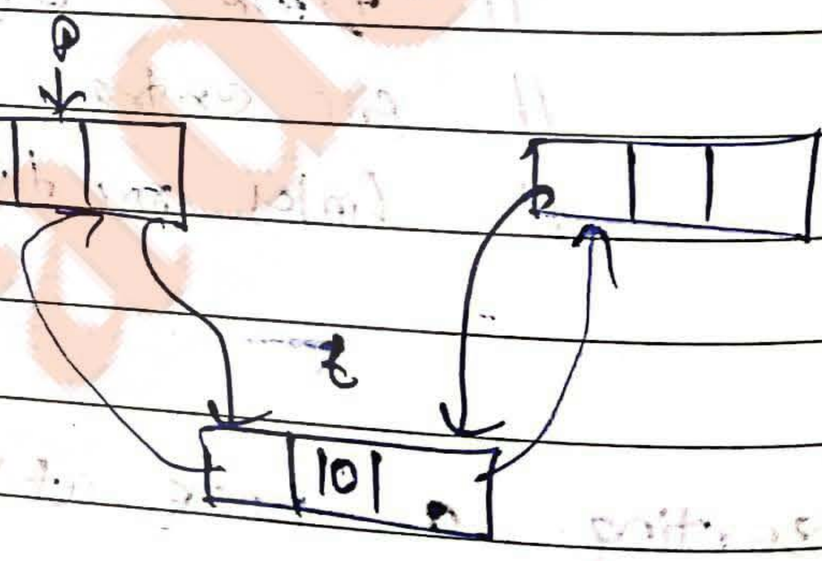
① t->next = p->next;
② t->prev = p;



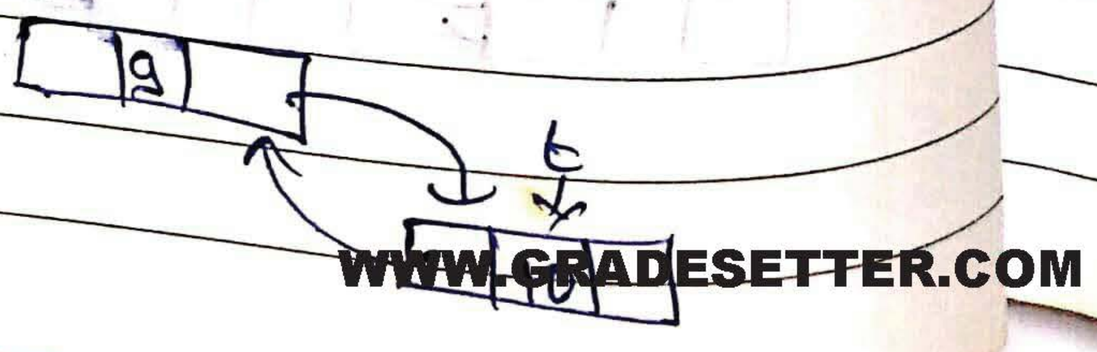
③ p->next->prev = t;



④ p->next = t;



(c) Inserting at the end :-



Same as (b)

(7) if (p → next)

p → next → prev = t;

Same as b.

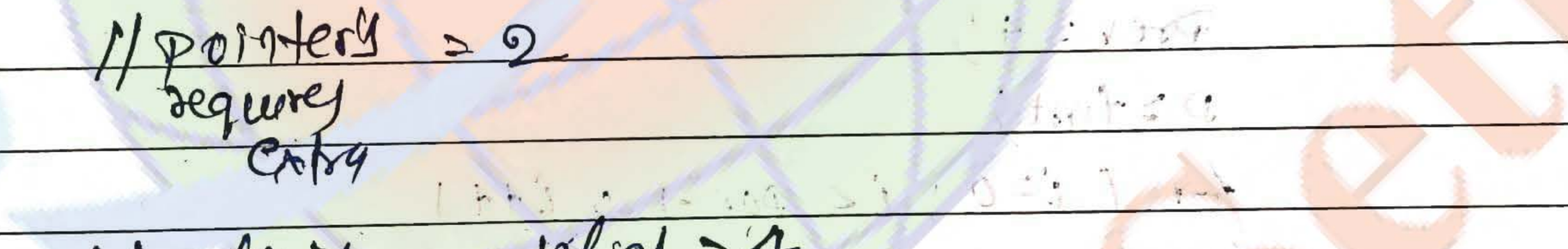
// time = O(n)

// pointers = 2
required extra

// links modified = 4

(2) Deletion:-

(a) Deleting first node: (But at least two nodes is required)

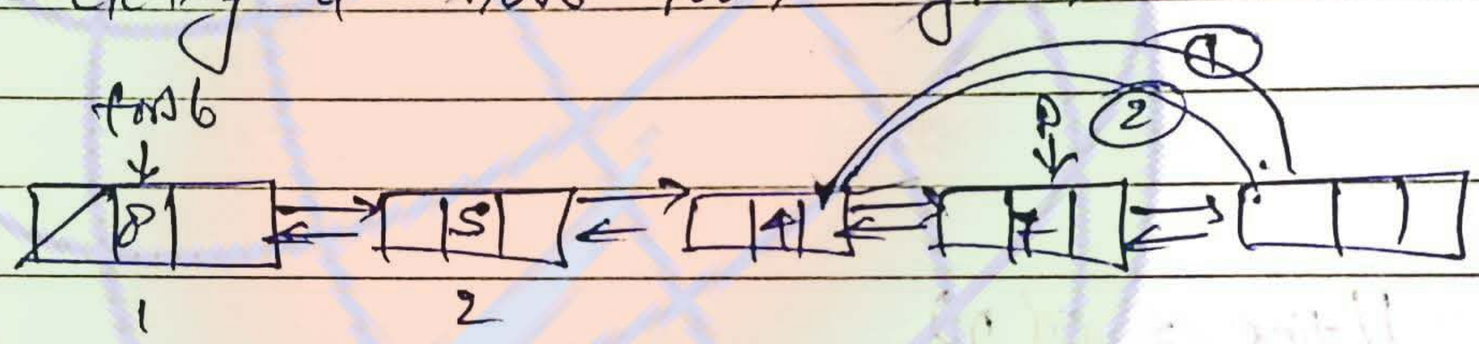


x = first → data;
first = first → next;
free (first → prev);
first → prev = NULL;

(b) Deleting first node

```
x = first -> data;
p = first;
first = first -> next;
free (p);
if (first)
    first -> prev = NULL;
```

(c) Deleting a node from given position:-

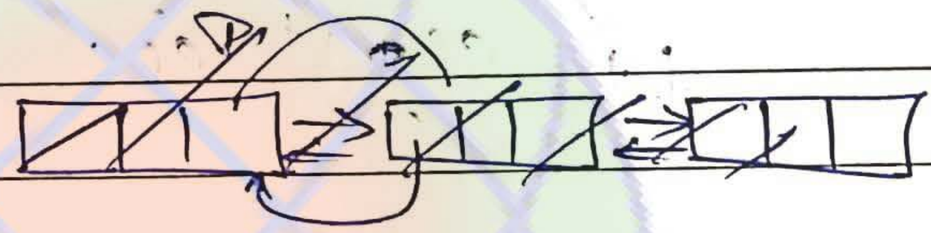



```
prev = 4;
p = first;
for (i=0; i< pos-1 ; i++)
        p = p -> next;
① p -> prev -> next = p -> next;
② if (p -> next)
        p -> next -> prev = p -> prev;
x = p -> data;
free (p);
// time : O(n)
// No. of Pointer required = 1
// No of link modified = 2
```

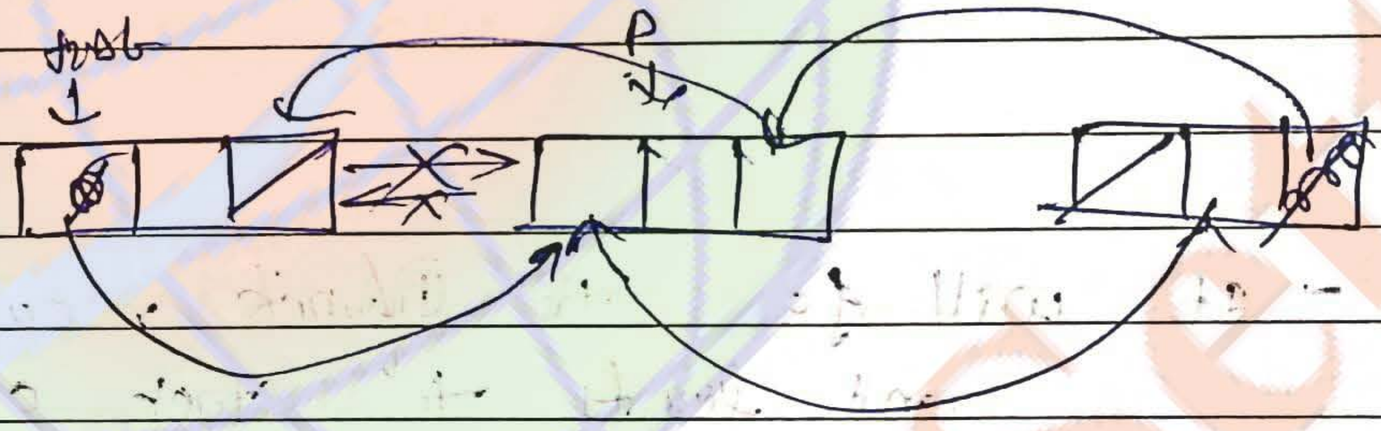
Date
Page

Q>
```
p = first;
while (p)
{
    q = p -> next;
    p -> next = p -> prev;
    p -> prev = q;
    p = p -> prev;
}
```

Sol:



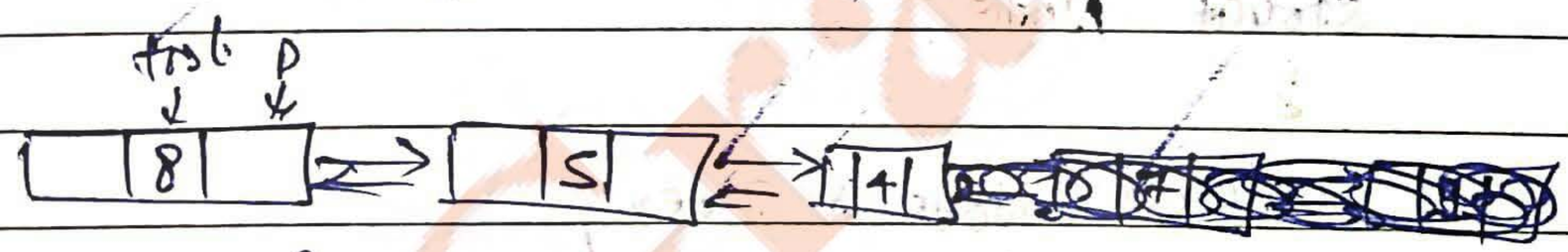
→ Link list is reversed



Interchanging the pointer, link list get reversed.

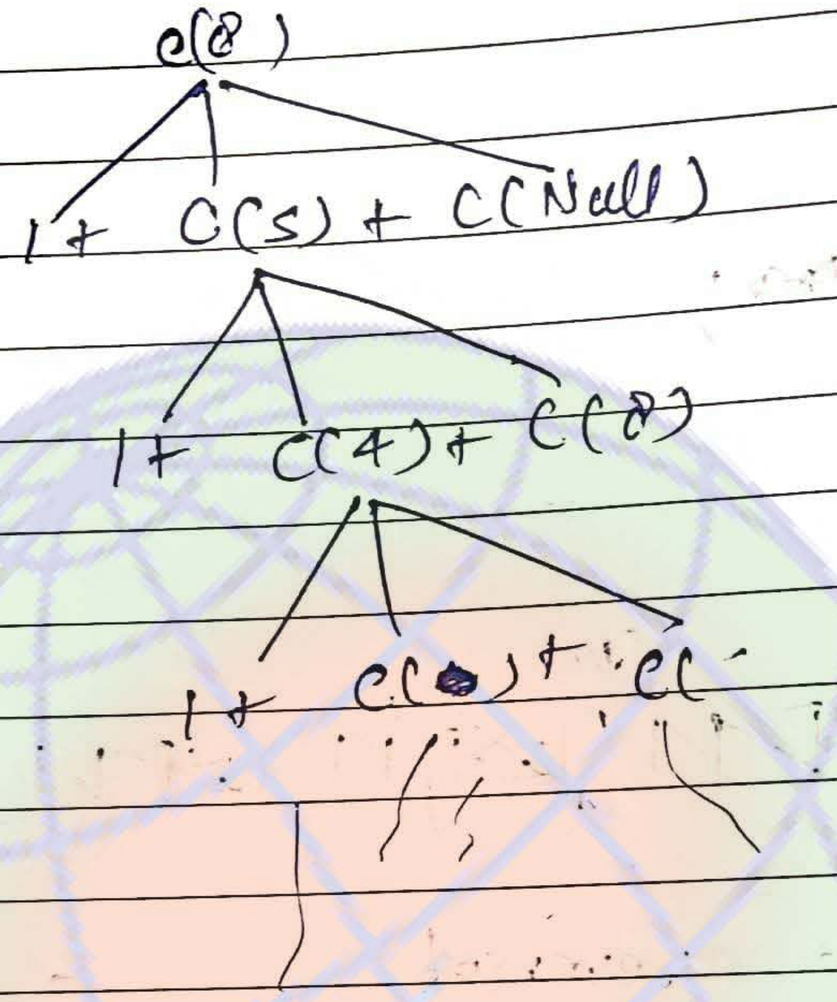
- First remains on the same node, so it will pointing on the last node of reversed node.

Code Q>




```
int Count (struct Node * p)
{
    if (p)
    {
        return 1 + count(p -> next) + count(p -> prev);
    }
    return 0;
}
```

A It will go to infinite loop.



- It will go to Infinite recursive calling and leads to stack overflow

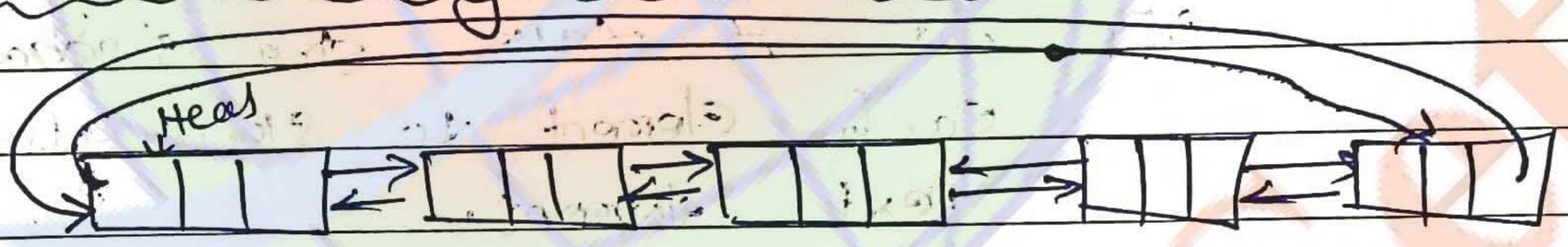
gate
c)

Prev	data	flag	next

```
struct Node
{
  struct Node * prev;
  int data;
  int flag;
  struct Node * next;
};
```

Ans
```
int count(struct Node *P)
{
    if(P)
    {
        if(P->flag ==0)
        {
            P->flag =1;
            return 1+count(P->next)+count(P->prev)
        }
    }
    return 0;
}
```

★ Circular doubly linked list:-



Benefit: circular & bi directional

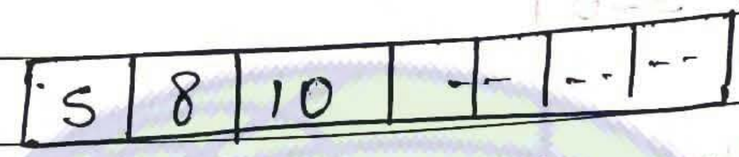
- Procedure of insertion deletion will be same as proceedure for insert & delete of doubly linked list.

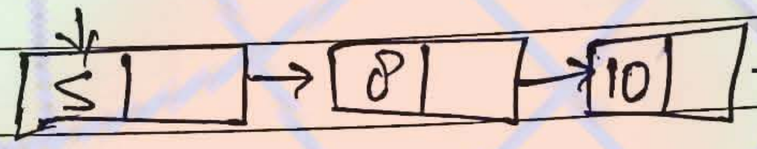
(a) Inserting nodes before Head

```
t = new node;                    // links modifies = 4
t->data = x;
P = Head->prev;
t->next = Head;
t->prev = P;
P->next = t;
Head->prev = t;
```

Operations on List's (Array/Linked list)

5, 8, 10, 16, 12, 4, 9





// link list is better, due to it's dynamic size, but it takes extra space for each element to store address of next element.

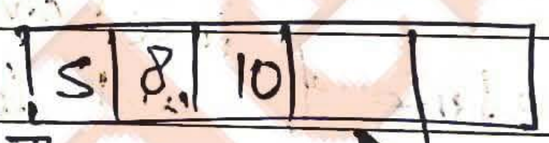
1.) Insert
2.) Delete
3.) Search
4.) Find
5.) Sort
6.) Reverse
7.) Shift
8.) Split
9.) length
10.) ~~Update~~ Append
11.) Merge

Set Operations.

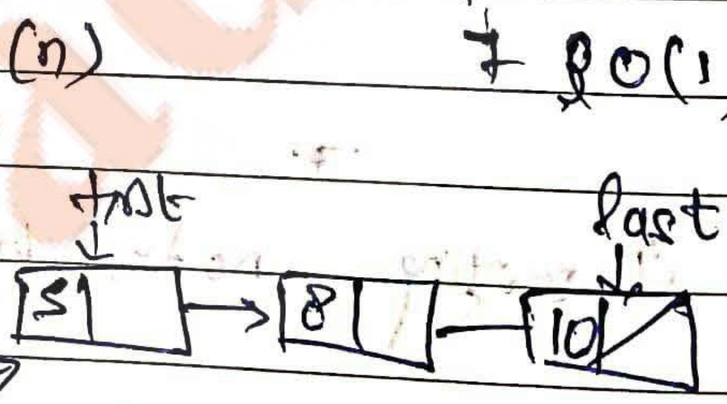
12) Union - n^2
13) Intersection - n^2
14) Difference - n^2

} Set membership

Insert

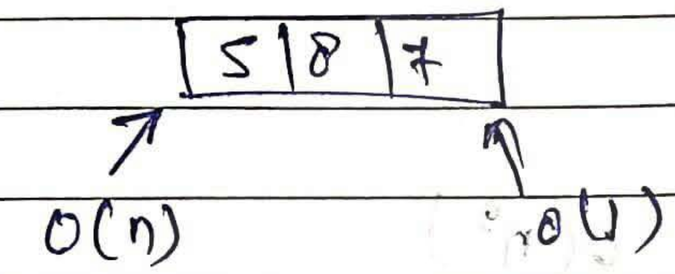


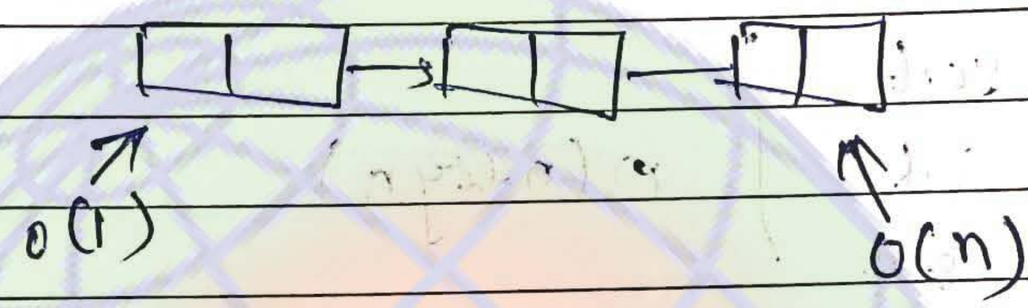
$\rightarrow O(n)$ $\rightarrow O(1)$



$\rightarrow O(1)$ $\rightarrow O(n)$

② Delete





Note:

③ Search

	Array	Linked list
(a) Linear search	O(n)	O(n)
(b) Binary search	O(log n)	O(n log n)

Can be performed if the elements is sorted.

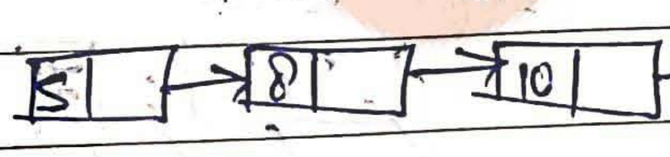
we can not directly reach the middle element, we have to traverse the nodes.

// Binary search is not efficient on linked list.

④ Find: finding an element at a given position.

0	1	2	3	4
A	5	8	10	6	7

O(1)



O(n)

5@) short :-

(a) Bubble sort
(b) Insertion sort
(c) selection sort
} $O(n^2)$

(d) Heap sort
(e) Quick sort
(f) merge sort
} $O(n \log n)$

* → These shorts are on the basic of linked list.
→ These two are useful on linked list.
→ Insertion & merge sort work on nodes.

gate 05

| 8 | 5 | 3 | 4 |

| 8 | → | 5 | →

6) Reverse

array $O(n)$
linkedlist $O(n)$

7) shift

left shift : 1

| 5 | 8 | 7 | 6 | 2 |

left rotate

| 5 | 8 | 7 | 6 | 2 | 5 |

Date
Page

WELCOME

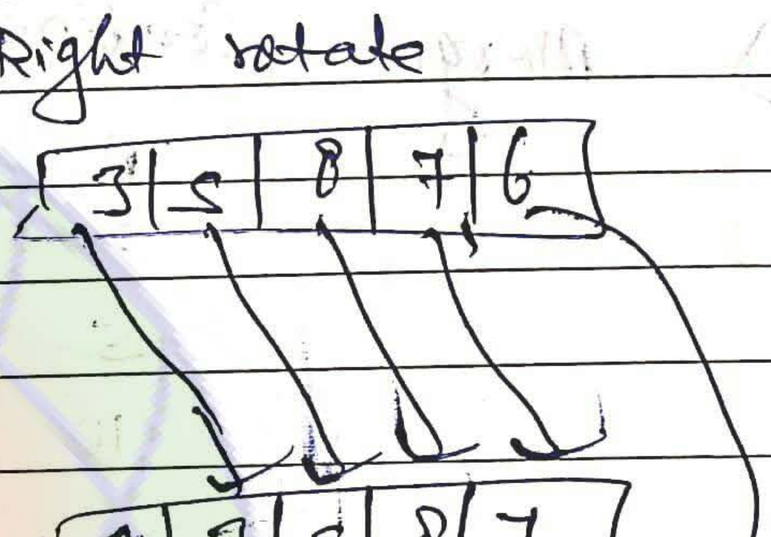
Rotation

ELCOMEW

Right Shift

you have to put something, so put '0'.

Right rotate



gate Q)

A[1 —— n]

reverse (A, 1, k)
reverse (A, k+1, n)
reverse (A, 1, n)

Ans

A

1	2	3	4	5	6	7	8	9	10

5 6 7 8 9 10 11 12
1 2 3 4 5 6 7 8

K

7 6 5 12 11 10 9 8
1 2 3 4 5 6 7 8

n-k times left rotation

8 9 10 11 12 5 6 7
1 2 3 4 5 6 7 8

→ left rotation by 7 elements

(8) Split :- Splitting an array into two or more

(9) Length - is the measuring of the length

10) Append - Adding more elements on the tail of the array.

11) Merge : Combining two shorted list into single shorted list.

A	B	C
2	3	2
5	6	3
10	9	5
m	12	6
	n	9
		10
		12

$O(m+n) = O(n)$

// For merging list should be in shorted way.

Set Operation

12) Union

A | 3 | 5 | 10 | 12 | 16 | — n

B | 4 | 5 | 9 | 12 | 20 | — n^2

~~C | 3 | 4 | 5 | 9 | 10 | 12 | 16 | 20~~

C | 3 | 5 | 10 | 12 | 16 | 4 | 9 | 20

Note: Union can be done with the help of merging if the elements is shorted time : O(n)

$f(n) = n^2 + n$

$O(n^2)$

13) Intersection

A | 3 | 5 | 10 | 12 | 16 | n

B | 4 | 5 | 9 | 12 | 20 | n^2

पहले हो दे रत लो common
ई then array में
insert करेंगे

C | 5 | 12

$f(n) = n^2 + n$

$O(n^2)$

14) Difference : non common element

A [ ]

B [ ]

C [ ] time : $O(n^2)$

(15) set

- element does belong to the set or not
-

$\Rightarrow$ // time : $O(n)$

Union

Note: Using merging for set operation

If two sets A and B are sorted, then merging can be used for performing set operations.

It will take $O(n)$ times

for Union.

// while merging, don't copy duplicate value.

Date
Page

Intersection

for intersection, while merging take only equal elements.

(a) Code for merging:-

```
i=0 ; j=0; k=0;
while ( i<m && j<n)
  {
  if ( A[i] < B[j])
    {
    C [k++] = A[i++];
    }
  else
    {
      C [k++] = B [j++];
    }
  }
for ( ; i<m ; i++)
    C [k++] = A [i];
for ( ; j< n; j++)
    C [k++] = B [j];
```

(b) Code for Union by merging.

```
i=0 ; j=0; k=0;
while ( i<m && j<n)
{
if ( A[i] < B[j])
  {
  C [k++] = A [i++];
```

```
else if (A[i] > B[j])
    {
    C[k++] = B[j++];
    }
else
    {
    C[k++] = A[i++];
    j++;
    }
}
for ( ; i<m; i++)
    C[k++] = A[i];
for ( ; j<n; i++)
    C[k++] = B[j];
```

(c) Intersection using merging.

```
i=0; j=0; k=0;
while (i<m && j<n)
{
if (A[i] < B[j])
    {
    i++
    }
else if (A[i] > B[j])
    {
    j++;
    }
else
    C[k++] = A[i++];
    j++;
}
```

Date
Page

Stack

① - Stack is a data structure which works on disciplines LIFO (last in first out)
- Elements are inserted and deleted from same end.

- Stack can be implemented using
  (a) Array and
  (b) linked list
  hiding details, showing only required.

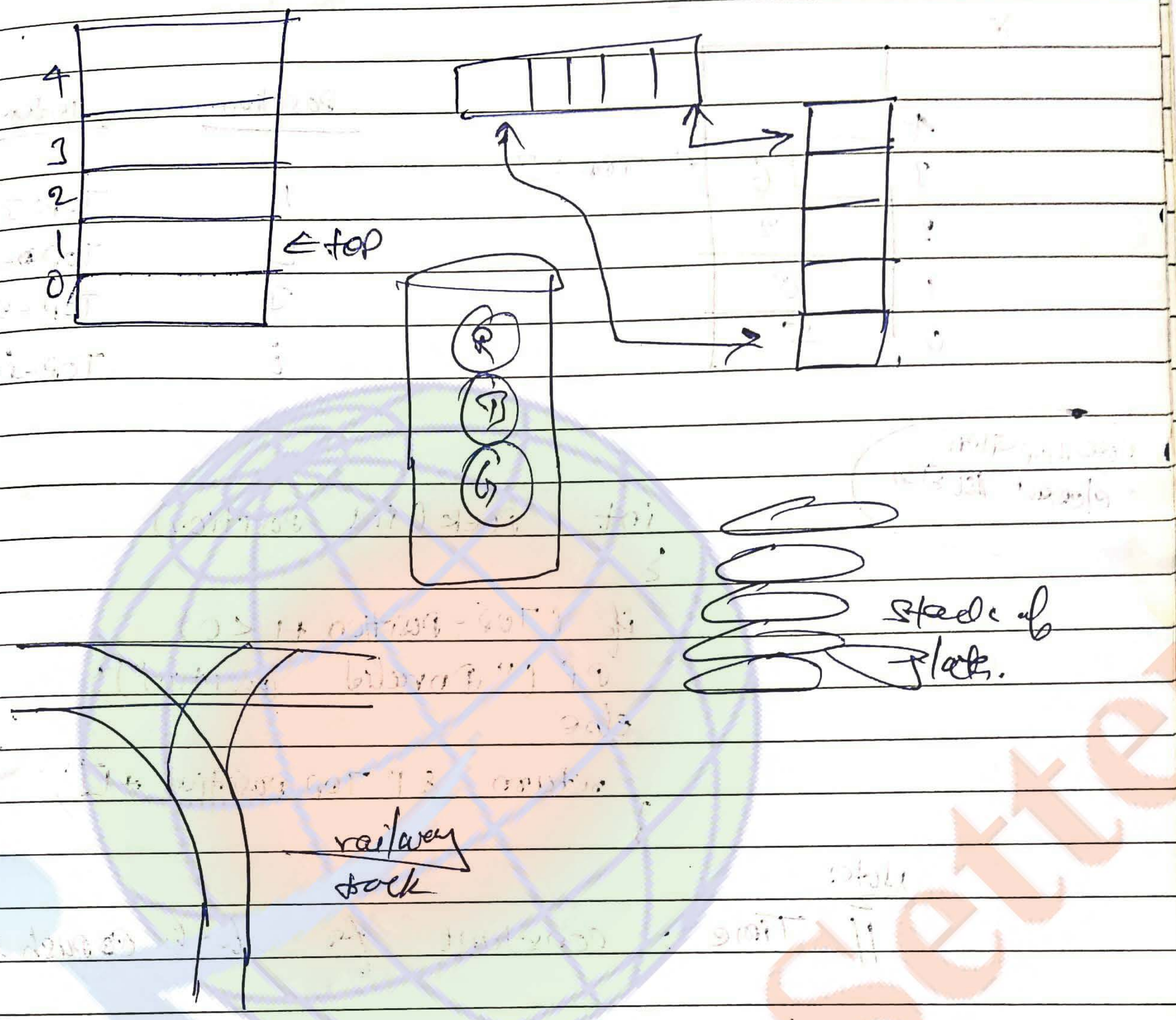
② ADT (Abstract data type of stack):-

Data:
1> space for storing elements
2> Top-pointer, to point on topmost element

Operation:
Push(x)
POP( )
Peek( ) - finding an element at given position
StackTop( ) - knowing what is the topmost element on it
isFull( ) - } check if the stack is full or
isEmpty( ) - } empty.

Note:
Data type: - defines, representation of data and Operations on data.

3> Implementing stack Using Array:



Initially	Empty	Full
Top = -1	if (Top == -1)	if (Top == size - 1)

```
push (int x)
{
if(TOP == size-1)
    Pf (" stack is Full");
else
    {
      Top++;
      s[TOP] = x;
    }
}
```

```
int POP()
{
int x=0;
if (Top == -1)
  Pf ("stack is empty");
else
  {
    x = s [TOP];
    Top --;
  }
return x;
```

size = 5

*

4	
3	16
2	12
1	8
0	5

← TOP = 3

Position	Index
1	TOP = 3
2	TOP - 1 = 2
3	TOP - 2
i	TOP - i + 1

** Note: top वाला element list होगा

```
int Peek (int Position)
{
  if (TOP - Position + 1 < 0)
    Pf ("Invalid position");
  else
    return S[TOP-Position+1];
}
```

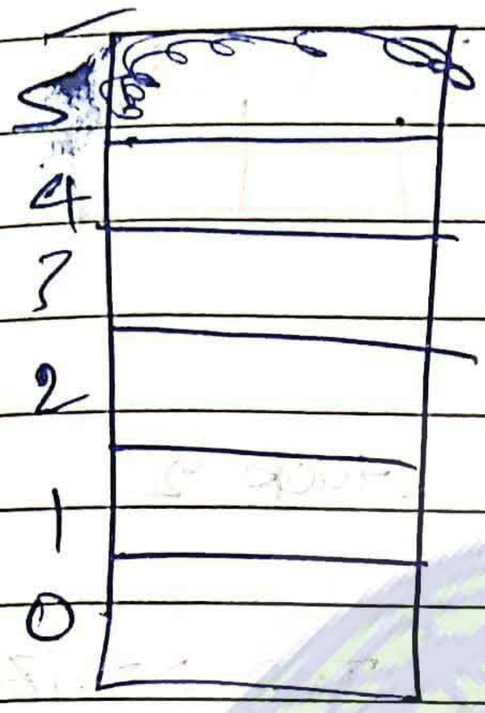
Note:

// Time : constant for both Push & Pop

- Full - ei the state
- overflow - result of an operation

```
push (int x)
{
  if (TOP >= -size-1)
    Pf ("stack ei full");
  else
  {
    ++top
    S[top] = x;
  }
}
```

(4) Using zeroth (0th) index of an array as a top pointer:-



size 5

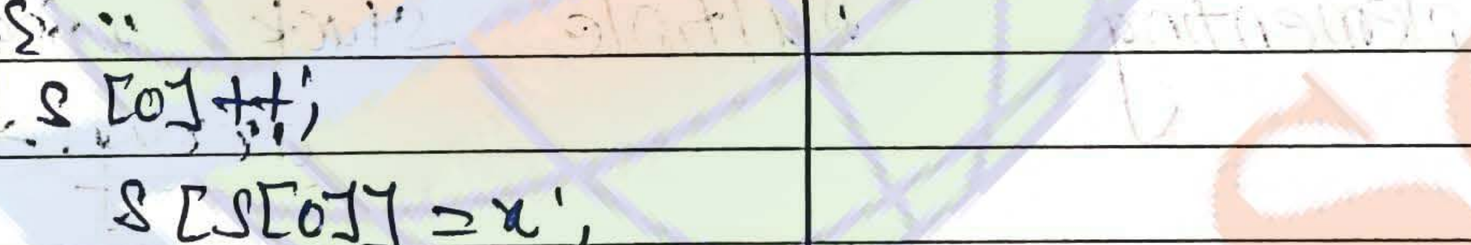
Initially	Empty	Full
S[0] = 0	If (TOP == -1)	If (TOP == size-1)

```
push(int x)
{
if (S[0] == size-1)
    pf("stack is full");
else
{
S[0]++;
S[S[0]] = x;
}
}
```

```
int pop
```

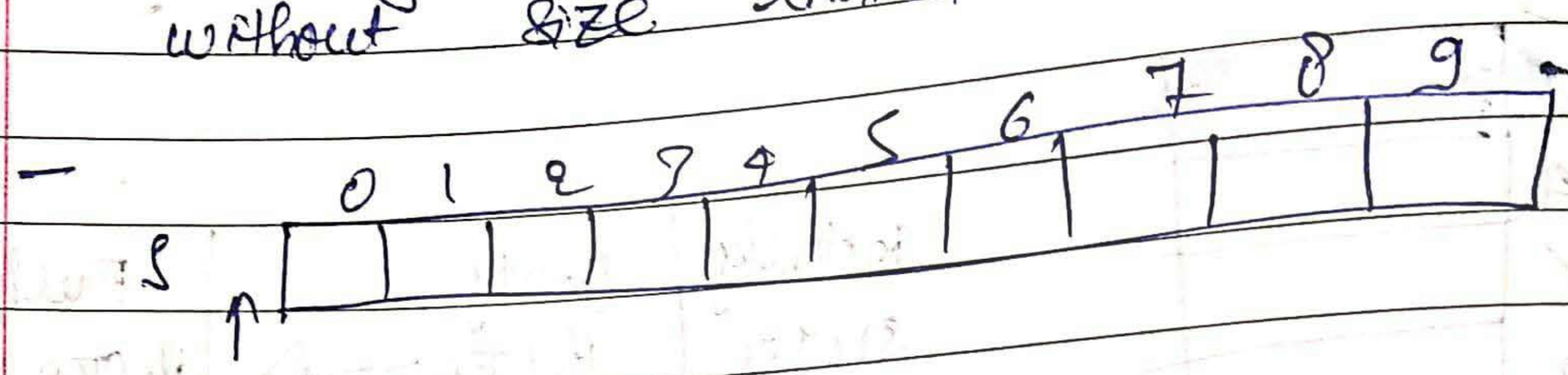
(5) Implementing two stacks using single array:-

size = 10



- Single array is used for implementing two stacks. from either ends and the array is

- uses by both the stack without size limit.



Stack 1	Stack 2
Empty: if (Top == -1)	if (Top == size)
Full: if (Top 1 == TOP2 - 1)	if (Top 2 == TOP 1 - 1)
	if (TOP 1 == TOP 2 - 1)

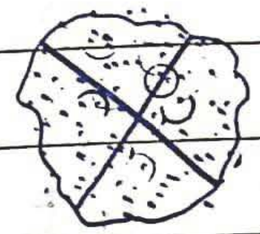
☆ Implementing multiple stack using single array:

Array S of size - m
No. of stacks - n
Array of Tops - T[n]
Array of Boundries - B[n+1]

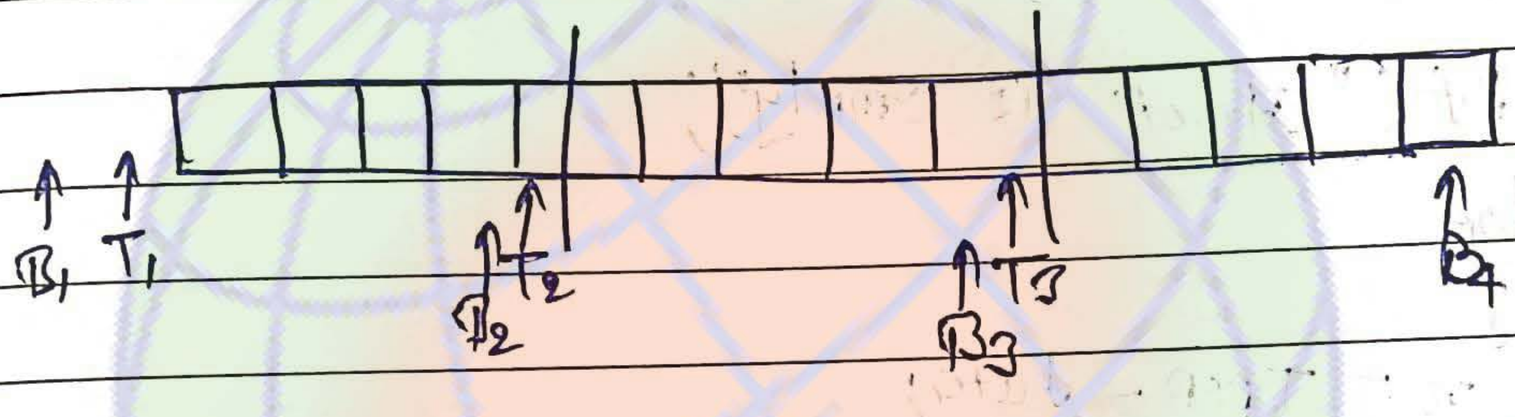
```
for (i=0; i<n; i++)
{
    T[i] = m/n * i - 1;
    B[i] = m/n * i - 1;
}
B[n] = m-1;
```

Empty and full stack :

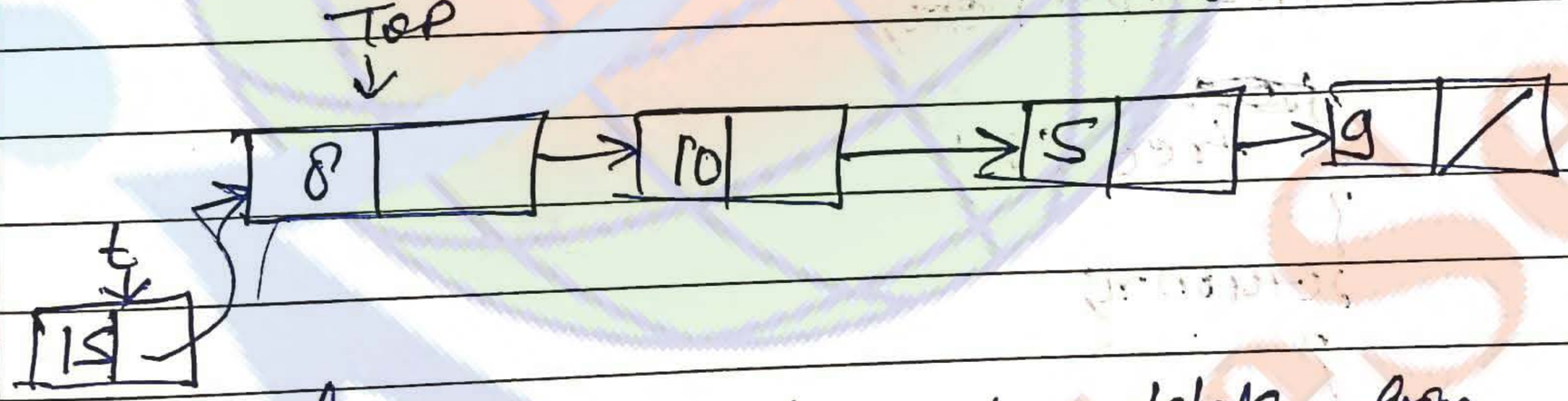
empty : if (T[i] == B[i])



Full : if (T[i] == B[i+1])



* Implementation of stack using linked list :-



- always insert and delete from left hand side, it will take constant time.

```
Push (int x)
{
t = new Node;
if (t == NULL)
  Pf ("stack is full");
else
  {
    t -> data = x;
```

```
        t->next = Top;
        Top = t;
    }
}

int pop()
{
    int x=0;
    if (Top==NULL)
        Pf("Stack is empty");
    else
    {
        x = Tpop->data;
        T = Top;
        Top=Top->next;
        free(T);
    }
    return x;
}

Peek(
```

Note: Both Push and Pop takes constant time.

Infix to Postfix Conversion:

(1) Infix: operand operater operand

eg. a + b

prefix: operator operand operand

eg. +ab

Postfix: operand operand operater

eg. ab+

Note: right to left

operate	precedence (lower)
=	0
+, -	1
*, /	2
^	3
( )	4

left to right

increasing prece ↓ higher

(2) infix a+b*c

Postfix a+b*c

1

2

(a+(b*c))

(a+[bc*])

postfix = abc*+

prefix: (a+(b*c))

a+[*bc])

prefix : +a*bc

(2) $a+b+c+d*e$

$a+b+c+(d*e)$

$(a+b)+c+(d*e)$

$((a+b)+c)+(d*e)$

$(((a+b)+c)+(d*e))$

Prefix:

$+++abc*de$

Post:

$ab+c+de*+$

(3) $(a+b)*(c-d)/e$

Post fix:-

$\frac{(a+b)*(c-d)}{e}$

$\Rightarrow \frac{(ab+)(cd-)*}{e} = \frac{(ab+)(cd-)*}{e}$

$= (ab+)(cd-)*e/$

Prefix $*/+ab-cde$

(4) $a+b-c*d/e$

Postfix:

$a+b-cd*e/$

$\Rightarrow ab+-cd*e/$

$\Rightarrow ab+cd*e/-$

prefix

$a+b$

$-+ab/*cde$

$a+b-*\frac{cd}{e}$

$a+b-/*cde$

$\Rightarrow +ab-/*cde$

$\Rightarrow -+ab/*cde$

(5) $a = b = c = 10$ (Right to Left)

$a + b + c$ (left to right)

$a^{(b)^c} \rightarrow a \wedge b \wedge c$ (Right to left)

Associativity

→ If two or more operaters are having same presidence in an expression, then they are evaluates based on associativity.

→ operates can have two types of associativity

(a) L to R → eg. + - * /.

(b) R to L → eg. (=) Assignment operate

(1) R-L : $a = b = c = 10$

Prefix : $= a = b = c 10$

Postfix : $abc 10 = = =$

(2) RL → $a^{(b)^c} \rightarrow a \wedge b \wedge c$

prefix : $\wedge a \wedge bc$ post = $abc \wedge \wedge$

Note ① $a * b - c * d$

$ab * cd * -$

② $a * b + c$

abc

If operats are continous, then it will affects.

Note

Note Unary operaters

④ Unary operater has highest precedence, and their associativity is Right to Left

⑤

Precedence is same, associativity → R to L

eg ① $-a + \sqrt{b}$

Post

$a - b\sqrt{+}$

Pre

$+ - a * \sqrt{b}$

eg ② $\log(a+b)!$

Post

$ab+!\log$

Pre

$\log!(+ab)$

(3) $-3*a+b\log\sqrt{c}$

Post.

$-3*a+bc\sqrt{\log}$

$3-*a+bc\sqrt{\log}$

$3-a*+bc\sqrt{\log}$

$3-a*bc\sqrt{\log}*+$ // $bc = b*c$

Note

$ab+cd$

$ab*+cd*$

$ab*cd*+$

(4) $a*b+c-d/e*f$

Post :- Df associate è R to L

$ab*+c-d/ef*$

$ab*+c-def*/$

$ab*c+-def*/$

$ab*c+def*/-$

$ab*cdef*/-+$

Now

Consider addition and subt. have higher
then *, -
as assoc is L to R

b c + - d
b c + d -

a b c + d - * c * f /
a b c + d - * c f / *

* ① a + b * c - d / e

② a + b c * + d / e -

a b c * + d e / -

③ a + b + c / + d ^ e - 3 * a ! + (c / f) * x

1	( )
2	-1
3	^
4	*, /
5	+, -

1

⇒ 5 scans

(5 बार ही expression
को scan करेंगे)

classmate
Date
Page

☆ Procedure for Infix to Postfix Conversion Using Stack:-

- this Procedure will by Scanning an expression only one time.

Procedure:-

(1) take a infix expression
(2) " " Stack
(3) scan through infix expres taking one symbol at a time.
(4) If a symbol is an operand, append into post fix
(5) If a symbol is an operator, then
(a) If the presedence of operator is greater then the presedence of symbol in stack, then push it into stack
(b) else
Pop out all greater or equal symbols from the stack.

Symbol	Stack	Postfix
a	-	a
+	+	a
b	+	ab
*	* +	ab
c	* +	abc
-	-	abc *+
d	-	abc*+d
/	/-	abc*+d/
e	/-	abc *+d/e
		abc*+de/-