

Data Structures

Scope Of a Variable

- ① Static Scoping
- ② Dynamic Scoping

int a = 500, b = 600;

Main()

```
{ int a, b;
  a = 10; b = 20;
  swap();
  printf("%d %d", a, b);
}
```

Swap()

```
{ int t;
  t = a;
  a = b;
  b = t;
  printf("%d %d", a, b);
}
```

O/p :-
600 500
(means variables are not in this fn.)

Static Scoping :-

O/p :- 600 500 10 20

When the free variables are taken from global variables.

Dynamic Scoping :-

O/p :- 20 10 20 10

When the free variables are taken from previous function call till we reach main().

If we don't get variables even in main, then we refer dynamic scoping global variable.

* C language follows Static Scoping, older languages follow dynamic scoping.

O/p in C :-
600 500 10 20

* int a = 10, b = 20;

main()

```
{ int a = 1, b = 1;
  A();
}
```

A()

```
{ int a = 2, b = 2;
  B();
}
```

B()

```
{ int a = 3, b = 3;
  printf("%d %d", a, b);
}
```

04.08.12

% 4
% 2:50

```

• boolean b=true;
  procedure p
  begin
    pf(b);
  end
begin
  boolean b=false;
  p;
end

```

b
true
1000
b
false
2000
no name for this fn.
C: it is the main fn.

• static scoping: true
 • dynamic scoping: false

```

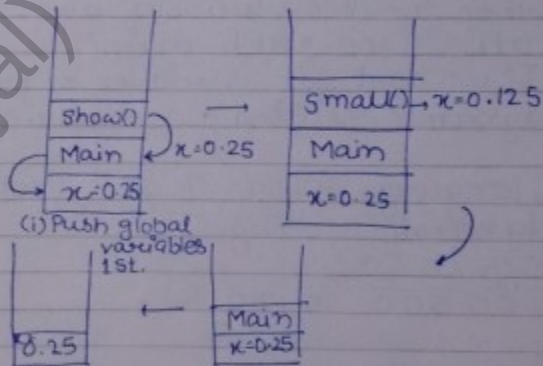
• var x:real
  procedure show()
  begin
    pf(x);
  end
  procedure small()
  begin
    var x:real
    x=0.125;
    show();
  end
  begin
    x=0.25
    show();
    small();
  end
end

```

no concept of stack.
 Since at compile time, we know only global variable.
 (at compile time)

• Static scoping: 0.25 0.25
 • dynamic scoping: 0.25 0.125
 (at run time)

For dynamic:-



```

• var x,y:integer
  procedure p (var n:integer)
  begin
    x=(n+2)/(n-3);
  end
begin
  x=7; y=8;

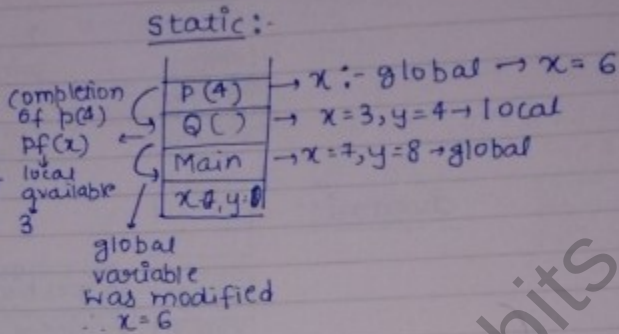
```

• Static: 3 6
 • dynamic: 6 7

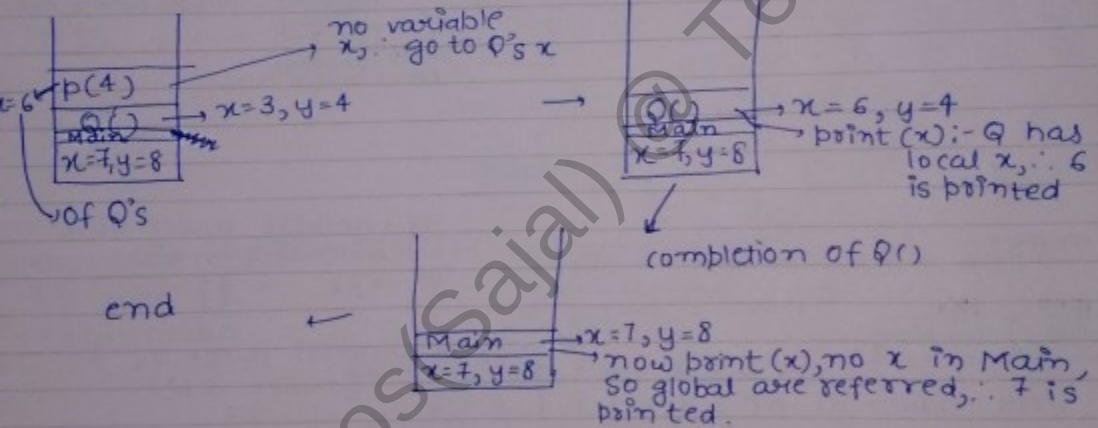
```

Q();
pf(x);
end
procedure Q
begin
  var x,y:integer
  x=3,y=4;
  p(y);
  pf(x);
end

```



dynamic:-



```

Q. var x:integer
procedure two
begin
  pf(x);
end
procedure one
begin
  var x:integer
  x=5;
  two;
end

```

```

begin
  r=2;
  two;
  one;
  two;
end

```

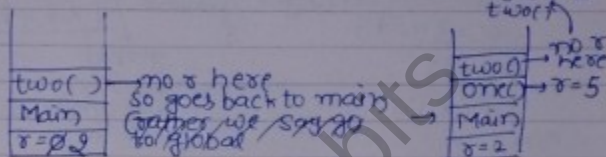
changed by main 2 $\boxed{\emptyset}$ ← global 1000

Static: 2 2 2

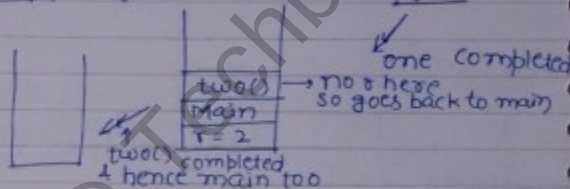
↑
pointed by pf(r) of two (global variable)

↑
pointed by pf(r) of two (when called by one)

dynamic:



∴ O/p :- 2 5 2



• int a; — $\boxed{}$ 1000 big endian

• a=55; — $\boxed{0000\ 0000\ 0011\ 0111}$ 1000 1001

1000 1001
 $\boxed{0011\ 0111}\ \boxed{0000\ 0000}$

↓
little endian

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

55 → nearest power of 2 (smaller is 2⁵)

$$2^5 \rightarrow 32 \rightarrow \begin{array}{r} 55 \\ -32 \\ \hline 23 \end{array}$$

23 → nearest power of 2 (smaller is 2⁴)

$$2^4 \rightarrow 16 \rightarrow \begin{array}{r} 23 \\ -16 \\ \hline 7 \end{array}$$

7 → nearest power of 2 (smaller is 2³)

$$2^3 \rightarrow 4 \rightarrow \begin{array}{r} 7 \\ -4 \\ \hline 3 \end{array}$$

3 → nearest power of 2 (smaller is 2¹)

$$2^1 \rightarrow 2 \rightarrow \begin{array}{r} 3 \\ -2 \\ \hline 1 \end{array}$$

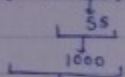
1 → nearest power of 2 (smaller or equal)

$$2^0 \rightarrow 1 \rightarrow \begin{array}{r} 1 \\ -1 \\ \hline 0 \end{array}$$

pf(a) → 55

pf(&a) → 1000

pf(*(&a)) → 55



value at address 1000

pf("%d", a) → 55

pf("%u", &a) → 1000

unsigned
(for addresses)

pf("%d", *(&a)); → 55

• int a, →

• a = 55; →

• a → 55

• b → 1000

• *b → 55

• *a → error (compilation error)

• &b → 2000

• *(&b) → 1000

& → referencing operator

* → dereferencing

int a = 10; b = -1;

printf("%u", a); → 10

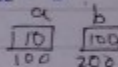
printf("%u", b); → 2's complement

of -1 (if range of int is upto 4294967295) then the o/p is 4294967295

printf("%d", b); → -1

int b = &a; // we can't write like this, // because writing int b, // we committed that we will // store a value at 'b'

we write :- int *b = &a;



printf("%u", b) → 1000

we use 'u' because we declared as pointer

printf("%d", *b) → 55

printf("%u", &b) → 2000

int a = 10;

int b = (int)&a;

pf("%d", a); → 10

pf("%d", &a); → 1000

pf("%d", b); → 100

pf("%d", *b); → compile time error

Q1. int a = 10; →

int *b = &a; →

int **c = &b; →

because b is itself a pointer

int ***d = &c; →

d is storing an address which is a pointer to a pointer.

• Single pointer :- storing address of a value.

• double pointer :- storing address of a pointer.

- ★ If variable is holding a value, then to get the value → no ^{req.}
- ★ " " " " " single pointer, then to get the value → 1 ^{req.}
- ★ " " " " " ~~two~~ pointers, " " " " " → 2 ^{req.}
- double

Parameter Passing Techniques

1. Call by value / Call by Copy
2. Call by reference
3. Call by copy-restore
4. Call by Name
5. Call by Need
6. Call by Text

Call by Value :-

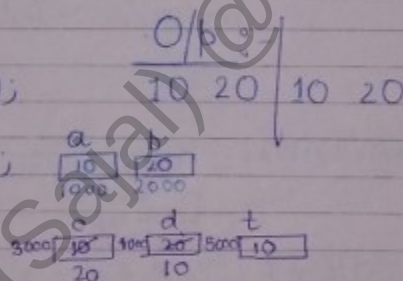
```
main()
{
    int a = 10; b = 20;
    printf("%d / %d", a, b);
    swap(a, b);
    printf("%d / %d", a, b);
}
```

```
swap(int c, int d)
{
    int t;
```

```
    t = c; // c = 20
    c = d; // d = 10
    d = t;
}
```

• Without using temp. variable:-

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



★ In C compiler, by default call by value is done,

i.e. `swap(a, b)` → call by value

★ If any compiler supports call by reference by default

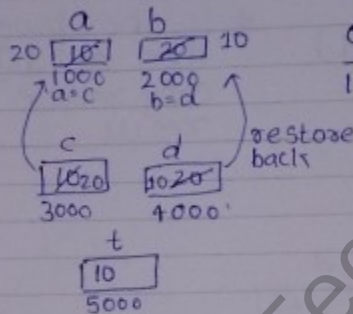
then `swap(a, b)` → call by reference

It will pass addresses of `a` & `b`, not values of `a` & `b`.

Call by Copy - Restore

```
main()
{int a=10,b=20;
 printf(a,b);
 swap(a,b);
 printf(a,b);
}
swap(int *c, int *d)
{int t;
 t = *c;
 *c = *d;
 *d = t;
}
```

★ executing the program in a compiler having call by copy-restore by default.



O/p:-
10 20 | 20 10

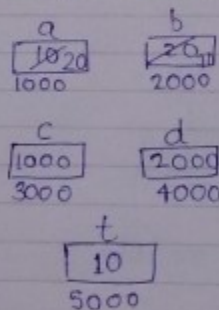
5/3x5
3/8x5

★ Changes to actual parameter takes place at the end using call by copy restore.

★ Changes to actual parameter takes place on the fly using call by reference.

Call by reference

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



O/p:-
10 20 | 20 10

Q. Consider the following program:-

```
main()
{
    int a = 5;
    Test(a, a);
    printf(a);
}

Test(int c, int d)
{
    c += d * 2;
    d *= c - 3;
}
```

★ Call by reference ~~or~~ call by copy restore gives diff. answer if then program contain aliasing.
 ★ If the program contains aliasing then call by reference & call by copy restore may generate same result

Call by Value:-

a	c	d
5 1000	515 2000	560 3000

$$c += d * 2 \rightarrow c = c + d * 2$$

$$d *= c - 3 \rightarrow d = d * (c - 3)$$

in this case, always R.H.S. is calculated first irrespective of the priorities.

O/p:-
5

Call by Copy-Restore:-

a	c	d
5 1000	515 2000	560 3000

restore

restore

My answer:-

O/p:- 60 15/60

whichever is restored last is the output.

Call by reference:-

a	c	d
5 1000	1000 2000	1000 3000

$$*d = (*d) * 2;$$

$$(*c) = (*c) + (*d) * 2;$$

$$(*d) = (*d) * ((*c) - 3);$$

O/p:-
180

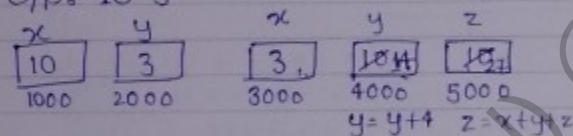
Q. Consider the following programs:-

```
main()
{
    int x=10, y=3;
    fun(y, x, x);
    printf("%d", y);
}

fun(int x, int y, int z)
{
    y = y + 4;
    z = x + y + z;
}
```

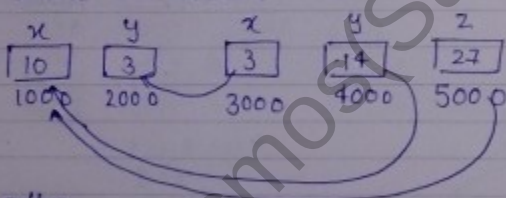
Call by Value:-

O/p:- 10 3



Call by Copy restore:-

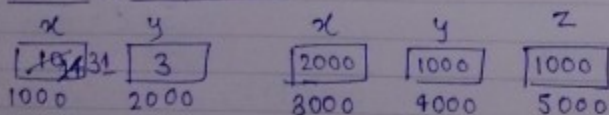
same as above



O/p:-

14/27 3

Call by reference



$$\begin{aligned} *y &= *y + 4 \\ &= 10 + 4 \\ &= 14 \end{aligned}$$

$$\begin{aligned} *z &= (*x) + (*y) + (*z) \\ &= 10 + 14 + 14 \\ *z &= 38 \end{aligned}$$

O/p:-

31 3

Aliasing:- x/y

• int x; $\rightarrow$ 1000

✓ int &y = x; $\rightarrow$ means y address is same as x address.
 * C doesn't support aliasing.

Q. Imp.

main()

{ int a, b;

a = 2; b = 3;

P(a+b, a, a);

printf(a, b);

this is done
in temporary
variable & its
address is passed.

}

P(int x, int y, int z)

{ y = y + 1;

z = z + x;

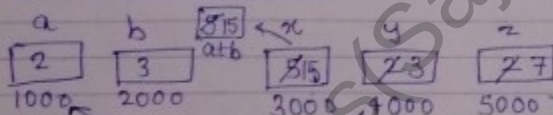
x = x + y + z;

}

Call by Value:-

O/p:- 2, 3

Call by Copy restore:-



a = 3/7

a + b = 15

if a = 3, then b = 12

if a = 7, then b = 8

My answer

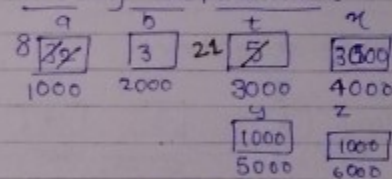
$\therefore$ O/p:-

3/7 12/8 $\rightarrow$ My answer

but answer:-

3/7 3

Call by reference:-



*y = *y + 1
= 2 + 1
= 3

*z = (*z) + (*x)
= 3 + 5 = 8

*x = (*x) + (*y) + (*z)
= 5 + 8 + 8
= 21

O/p:-
8 3

```

Q. main()
{
    int a=10, b=20;
    swap(a,b);
    printf(a,b);

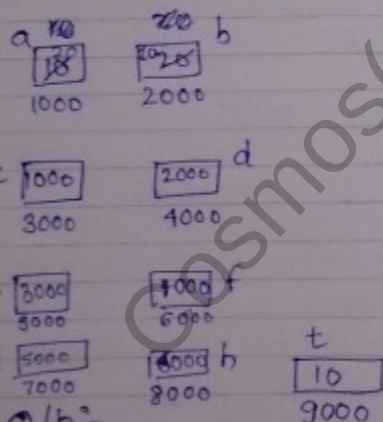
    swap(int*c, int*d)
    {
        interchange(c,d);
        printf(c,d);
    }

    interchange(int*e, int*f)
    {
        A(e,f);
    }

    A(int*g, int*h)
    {
        int t;
        t=g; // t=***g;
        g=h; // ***g=***h;
        h=t; // ***h=t;
    }
}

```

Call by reference:-



O/p:-
20 10

Q. Consider following C-program;

```
#include <stdio.h>
void f(int *p, int *q)
```

```
{ p = q;
```

```
*p = 2;
```

```
}
```

```
int i = 0, j = 1;
```

```
main()
```

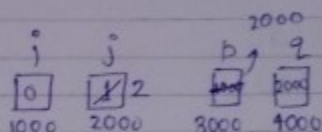
```
{ f(&i, &j);
```

```
printf("i/d", i, j);
```

```
return(0);
```

```
}
```

∴ Output:- 0 2



*p = 2

printf

Q. What is the o/p printed by the following C-program:-

```
main()
```

```
{ int c, *b, **a;
```

```
c = 4;
```

```
b = &c;
```

```
a = &b;
```

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

```
}
```

```
f(int x, int *py, int **ppz)
```

```
{ int y, z;
```

```
**ppz += 1;
```

```
z = **ppz;
```

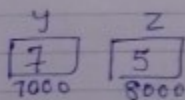
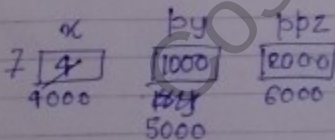
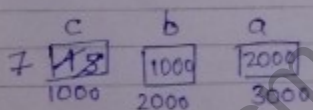
```
*py = z;
```

```
y = *py;
```

```
x += 3;
```

```
return(x+y+z);
```

```
}
```



7 + 7 + 5 = 19

O/p:-

19

**ppz = **ppz + 1;

= 4 + 1 = 5

z = **ppz

= 5

*py = *py + 2

= 5 + 2 = 7

y = *py;

y = 7

x = x + 3

= 4 + 3 = 7

Q. Consider the following C program:- (Static Scoping, by default as its a C program)

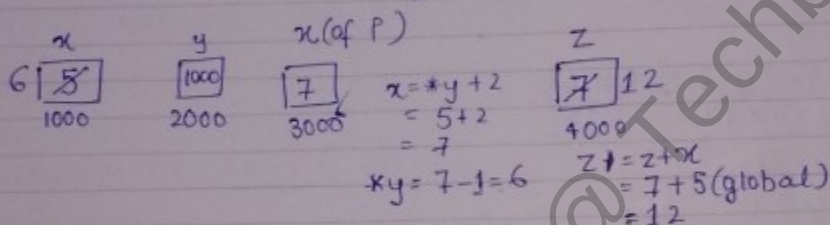
```
#include <stdio.h>

int x;

void Q(int z)
{
    z = x;
    printf(z);
}
```

```
void P(int *y)
{
    int x = *y + 2;
    Q(x);
    *y = x - 1;
    printf(x);
}
```

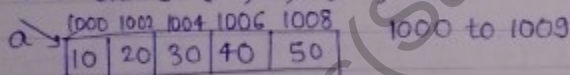
```
main()
{
    x = 5;
    P(&x);
    printf(x);
}
```



O/p:-

12, 7, 6

Q. `int a[5] = {10, 20, 30, 40, 50};`



`a = 1000`

`*a = 10`

`a+1 = 1000 + 1 * 2 = 1002` [as integer is of 2 bytes]

`*a+1 = *(1002) = 20 = a[1]`

`*++a`

`*a+4 = 50 = a[4]`

★ at time of preprocessing `a[4]` is converted to `*(a+4)`.

`a[50] = *(a+50) = *(base address + 50 * 2)`

↑ random access is possible due to pointers.

- `a+0` → skipping 0 elements → `a[0]`
- `a+1` → " 1 " → `a[1]`
- `a+2` → " 2 " → `a[2]`

```

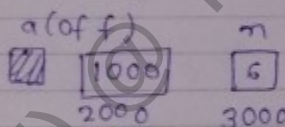
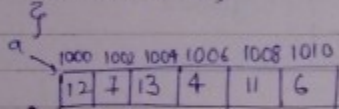
Q. #include <stdio.h>
int f(int *a, int n)
{ if (n <= 0)
    return 0;
  else
    if (*a % 2 == 0)
      return *a + f(a+1, n-1);
    else
      return *a - f(a+1, n-1);
}

```

```

main()
{ int a[] = {12, 7, 13, 4, 11, 6};
  printf("%d", f(a, 6));
  return 0;
}

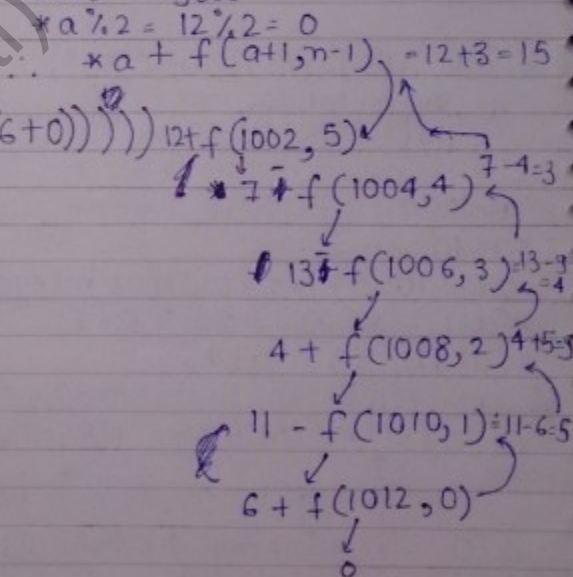
```



O/p:-

15

$$\begin{aligned}
 & 12 + (7 - (13 - (4 + (11 - (6 + 0)))))) \\
 &= 12 + (7 - (13 - (4 + 1))) \\
 &= 12 + (7 - (13 - 5)) \\
 &= 12 + (7 - 8) \\
 &= 12 + (-1) \\
 &= 11
 \end{aligned}$$



Q. What is the O/p of the following C-program?

```
#include <stdio.h>
int f(int n, int k)
{ if (n == 0)
    return 0;
  else
    if (n % 2)
      return f(n/2, 2*k) + k;
    else
      return f(n/2, 2*k) - k;
}
```

```
main()
{ printf("%d", f(20, 1));
}
```

O/p :-
9

Diagram illustrating the recursive calls for $f(20, 1)$:

```

      f(20, 1)
      /
    f(10, 2)
    /
  f(5, 4)
  /
f(2, 8)
/
f(1, 16)
/
f(0, 32)

```

Arrows indicate the return values being passed back:

- $f(0, 32) \rightarrow 0$
- $f(1, 16) \rightarrow 16 - 8 = 8$
- $f(2, 8) \rightarrow 8 + 4 = 12$
- $f(5, 4) \rightarrow 12 - 2 = 10$
- $f(10, 2) \rightarrow 10 + 1 = 11$
- $f(20, 1) \rightarrow 11 - 2 = 9$

Q. Main()

```
{ int a[] = {10, 20, 30, 40, 50};
  int *p[] = {a, a+1, a+2, a+3, a+4};
  int **ptr = p;
  ptr++;
  printf("%d %d %d", ptr-p, *ptr-a, **ptr);
  *++ptr;
```

```
  printf("%d %d %d", ptr-p, *ptr-a, **ptr);
  ++(*ptr);
  printf("%d %d %d", ptr-p, *ptr-a, **ptr);
  *ptr++;
  printf("%d %d %d", ptr-p, *ptr-a, **ptr);
}
```

?

ptr++ = ptr
after semicolon
is encountered
then only ptr
is incremented

a

10	20	30	40	50
1000	1002	1004	1006	1008

b

1000	1002	1004	1006	1008
2000	2002	2004	2006	2008

O/p:

2	2	20
4	4	30
4	6	40

ptr

2000

3000

→ ptr++, → ptr = ptr + 1 →

ptr

2002

3000

(it gives no. of elements)

1st printf → ptr - p = 2002 - 2000 = 2 → 2 means only 1 element
 *ptr - a = 1002 - 1000 = 2 → 2 means only 1 element
 **ptr = 20

→ *(++ptr) → ptr

2004

3000

2nd printf → ptr - p = 2004 - 2000 = 4 → 4 means '2' no. of elements
 *ptr - a = 1004 - 1000 = 4 → 4 means '2' no. of elements
 **ptr = 30

→ ++(*ptr) = *ptr - 1004, ++(1004) = 1006

ptr

1006

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

b

2000	2002	2004	2006	2008
1000	1002	1006	1008	1008

ptr

2004

3000

→ 3rd printf → ptr - p = 2004 - 2000 = 4 → 4 means '2' no. of elements
 *ptr - a = 1006 - 1000 = 6 → 6 means '3' no. of elements
 **ptr = 40

→ *ptr++; → *ptr will take first → 1006

ptr++ will not take place till is encountered
 *ptr takes place 1st, after execution ptr is incremented

b

2000	2002	2004	2006	2008
1000	1002	1008	1006	1008

ptr

2006

3000

(*ptr) = (*ptr) * X;
 after this statement ptr will take place

★ Note:- '*' and '++' having same priority, so we will go to associativity, & associativity is right to left.

→ 4th printf statement → $ptr - p = 2006 - 2000 = 6 \rightarrow 6$ means 3 no. of elements
 $*ptr - a = 1006 - 1000 = 6 \rightarrow 6$ means 4 no. of elements
 $**ptr = 5040$

★ If two addresses undergoes arithmetic opn.

O/p

1 1 20
 2 2 30
 2 3 40
 2 4 50 3 3 40

Q. #include <stdio.h>
 main()

```
{ int a[] = {0, 1, 2, 3, 4};
  int *p[] = {a, a+1, a+2, a+3, a+4};
  int **ptr = p;
  **ptr++;
  printf("%d %d %d", ptr - p, *ptr - a, **ptr);
  *++*ptr;
  printf("%d %d %d", ptr - p, *ptr - a, **ptr);
  ++**ptr;
  printf("%d %d %d", ptr - p, *ptr - a, **ptr);
}
```

a

0	1	2	3	4
1000	1002	1004	1006	1008

b

1000	1002	1004	1006	1008
2000	2002	2004	2006	2008

ptr
 2000
 3000

→ **ptr++;
 ↳ from right to left
 ptr++ will take place 1st
 $\therefore ptr++ \Rightarrow ptr = 2000$
 & *ptr = 1000
 & **ptr = 0
 & after ; , ptr++, $\therefore ptr = 2002$

1 2 2
1 2 3

1st printf:- $ptr - p = 2002 - 2000 = 2 \rightarrow 2$ means '1' element
 $*ptr - a = 1002 - 1000 = 2 \rightarrow "$ " " "
 $**ptr = 1$

$\rightarrow *++*ptr \rightarrow$ from right to left

$*ptr \rightarrow 1002$

$++*ptr \rightarrow (*ptr) = (*ptr) + 1$

$\therefore p \rightarrow$

1000	1004	1004	1006	1008
2000	2002	2004	2006	2008

$*1004 = 2$

2nd printf:- $ptr - p = 2002 - 2000 = 2 \rightarrow 2$ means '1' element
 $*ptr - a = 1004 - 1000 = 4 \rightarrow 4$ " '2' elements
 $**ptr = 2$

$\rightarrow ++**ptr \rightarrow$ from right to left

$*ptr = 1004$

$**ptr = 2$

$++**ptr =$ ~~$*$~~ $**ptr = **ptr + 1$

$\therefore$

a	1000	1002	1004	1006	1008
	0	1	3	5	7

$\therefore$ 3rd printf $\rightarrow ptr - p = 2002 - 2000 = 2 \rightarrow 2$ means '1' element
 $*ptr - a = 1004 - 1000 = 4 \rightarrow 4$ " '2' element
 $**ptr = 3$ [$*ptr = 1004 \rightarrow **ptr = 3$]

★ Subtracting two pointers means $*(**ptr)$
 gives the total size of the elements $*(**ptr++)$,
 that can reside in b/w them.
 if the elements are integers, then
total no. of int = total size b/w them
 2

Note 1:- For the pointer variable, we can add integer constants.
e.g. $p = p + 5$ $p_1 + i$ (skipping i elements in forward direction).

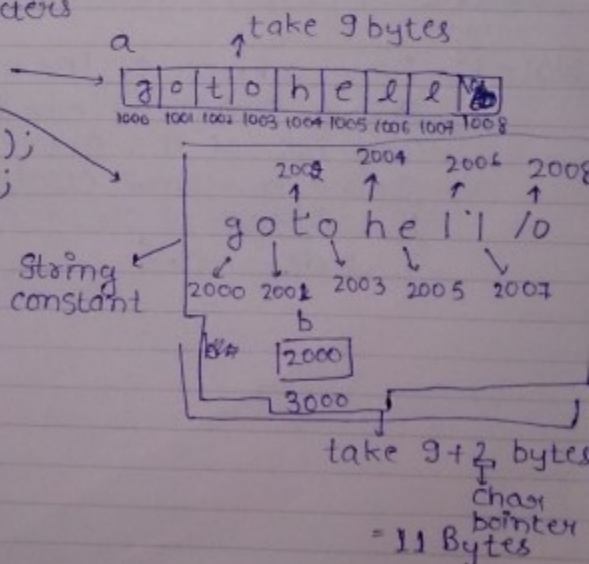
Note 2:- We can subtract integer constant from pointer variable
e.g. $p_1 = 1014$
 $p_1 = p_1 - 1 = 1014 - 2 \times 1 = 1012$

$p_1 - 2$ (skipping 2 elements in backward direction)
e.g. $p = 1000$ $q = p$ given no. of elements in
 $q = 1008$

Note 3:- Subtraction b/w 2 pointers ~~is~~ possible (p, q) iff
 $p - q$ [$p \geq q$ should be pointers of the same array]
where ' $p - q$ ' indicates no. of elements b/w two pointers including 1st one & excluding last one.

Note 4:- Multiplication, division & addition is not possible b/w two pointers because it has no meaning.

Q. main() Array of characters
{ char a[] = "gotohell";
char *b = "gotohell";
a[3] = printf("%s", a);
printf("%s", b);
a[3] = 'w';
b[3] = 'w';
printf("%s", a);
printf("%s", b);
}



* $a = a + 5$ → error
We can't change the base address of the array.

* but $b = b + 3$ → no error
because b is character pointer & can be changed.

- ★ `printf("%s", 1000)` → means it will print characters starting from 1000 address till null character (null character is not printed.)
- ★ `printf("%s", *1000)` → it will print only one character at 1000 address.

`printf("%s", a)` → goto hell
`printf("%s", a+4)` → hell
`printf("%s", *(a+1))` → l
`printf("%s", b)` → goto hell
`printf("%s", b+5)` → ell
`printf("%s", *(b+5))` → e
`printf("%s", *(a+3))` → o

• `a[3] = *(a+3)`

`*a[3] = 'o'`

`a`

g	o	t	h	e	l	l	\0
---	---	---	---	---	---	---	----

• `b[3] = *(b+3)`

`printf("%s", a);` → goto hell

`printf("%s", b);` → goto hell

Note 1:- Base address of the array can't be changed by string constant can be changed.
 Contents of the array can be changed, but string constant content can't be changed,
 if contents of string constant are changed, result is undefined.

Date

05.08.12

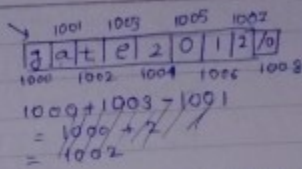
main()

```
char p[] = "gate2012";
printf("%s", p + p[3] - p[1]);
```

g

O/P:

te2012 2012



$$p[3] = *(p+3) = 'e'$$

$$e - a = 1004 - 1000 = 4$$

Warning in compilation takes place (program) when there is no loss of data but it doesn't follow semantic rules.

eg. int a;

char b;

a = b; // only warning, implicitly it is converting a = (int) b
// coercion (implicit type conversion)

// b = a; → this generate error message, because loss of data
// we are storing 2 byte data into 1 byte storage, & hence
// error.

b = (char) a; // correction, explicit conversion.

★ printf("%c", 'a'); → 'a'

★ printf("%d", 'a'); → 97

★ Lower Address → Higher Byte
Higher " → Lower Byte } → Big endian

★ Lower Address → Lower Byte
Higher " → Higher " } → Little endian

★ char s = 'a';

s = s + 1; // s now contains 'b'

★ char s = 255;

s = s + 1; // s will be 0.

★ char s;

s = 'a' + 'b'; → s = 97 + 98 = 195

s = 'b' - 'a'; → s = 98 - 97 = 1

Semantic rule

Lower bit
Variable using
lower memory
can be stored in
a variable using
more memory
but not vice
versa.

* (i+p) = i[p] ✓

★ printf ("%s", p); → gate2012

★ printf ("%s", p+5); → 012

★ printf ("%c", *(p+5)); → 0

★ printf ("%c", p); → error message, because we are using address with %c

★ printf ("%s", *p); → error message, as *p means g.

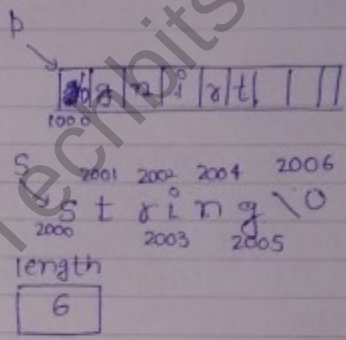
★ printf ("%c", *p); → g

Q. main()

```

{ char p[20];
  char *s = "string"; // string constant
  int length = strlen(s);
  for (int i=0; i<length; i++)
  { p[i] = s[length-i]; }
  printf ("%s", p);
}

```



Q. O/p :- Nothing (because first character is \0 itself)

★ sizeof ("abcd"); → 5 (including \0)
 ★ strlen ("abcd"); → 4
 (because sizeof means the size of "abcd" in the memory, i.e. 5 Byte.)

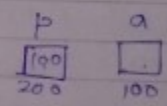
Q. main() → let Address to be of 4 Bytes

```

{ char *p;
  printf ("%d %d", sizeof(p), sizeof(*p));
}

```

Q. O/p :- 4 1
 *p means p is pointing to a character, which takes 1 B.
 sizeof(p) means p store address, address takes 4 B.



Q. main()

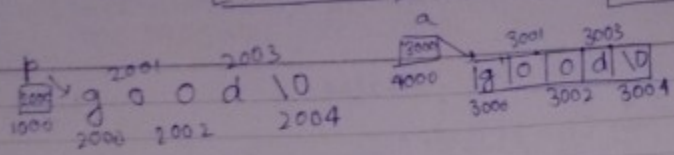
```

{ char *p = "good";
  char a[] = "good";
  printf ("%d %d %d", sizeof(p), sizeof(*p), strlen(p));
  printf ("%d %d", sizeof(a), strlen(a));
}

```

Note:-

void a = 10; → size of (b) → 'b' is a pointer which holds an address & hence takes 4B.



O/P:-

4 1 4
4 5 4 4

When we take address to be taking 4 Bytes.

size of (*b) → p is pointing to a character, which is of 1B

size of (p) → p stores an address

void a;
int b;
float c;
char d;
a = b;
a = c;
a = d;

★ Any datatype can be stored in void variable & void can be stored in any variable (with loss of data).
 * (without loss of data)
 b = (int)a; // have
 c = (float)a; // do explicit
 d = (char)a; // by casting

→ no error, without typecasting this is possible advantage of void type.

if:- int a[5];

size of (a); → size of the array :- 10B

char a[] = "good";

size of (a); → size of the array :- 5B (including null character)

main()
{ int i = -1, j = -1, k = 0, l = 2, m;
m = ((i++ && j++ && k++) || l++); m = 1
printf("%i %i %i %i %i", i, j, k, l, m);
}

O/P:-
0 0 1

My answer:- -1 -1 1 2 1

But ans ex:- m = ((i++ && j++ && k++) || l++); l++

left to right because && is binary operator

-1 && -1 && 1 && 2 = true = 1
O/P:- 0 0 0 2 1, l++ wasn't done.

★ if (0 || 0)
printf("hi");
else
printf("bye");

O/P:- hi

★ else A = 10
B = 0

if (A = B)
printf("hi");
else
printf("bye");

O/P:- bye

* if k=0

then o/p:- 0 0 1 3 1

is false

(i++ && j++ && k++)

∴ we need to execute l++

* If i=0, j=2, k=3 & l=2

∴ o/p:- 1 2 3 3 1

Imp.

* int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

10001000

10	20	30	40	50	60	70	80	90	100
0	1	2	3	4	5	6	7	8	9

int *b, = (int *) a;

char *c; // c waiting for an address of character.

float *d;

void *e;

c = (char *) a;

1000
3000

d = (float *) a;

1000
4000

printf(d) → 1000

printf(*d) → 10 20 (because d is a float pointer & will print 4 Bytes which is the size of float)

0000 0000 0000 0100, 0000 0000 0000 1010
100 20 10

0000 1010 0000 0000 0000 0100 0000 0000
1000 1001 1002 1003

it is in little endian & hence has to be reversed
if it is in big endian, then
O/p will be:- 10 20

printf(c) → 1000

printf(*c) → only 1 Byte will be retrieved, ∴ only 1000 address value is printed, ∴ o/p:- 10

printf(e) → 1000

printf(*e) → error.

* if c++, then c = 1001

* if d++, then d = 1004

* if b++, then b = 1002

* if e++, then error.

* No compilation error,
but windows stopped
the program execution.

a	b
2	2
10 00	20 00

Q.

```
int a=2;
int *b=(int *)a; → 

|      |
|------|
| 2    |
| 2000 |

 → 2 is an integer pointer now, which is integer value before
*if (cb) printf(b) → 2
printf(*b) → garbage (value at address 2, 3)
++b → 4 (because 2 is now integer pointer & hence ++b will increment it to 4).
++a → 3 (because value in a is integer value.)
```

Q. int a=2;

```
float *b=(float *)a; → 

|      |
|------|
| 2    |
| 2000 |

 → integer value '2' is converted to floating pointer
printf(b) → 2
* printf(*b) → garbage (value at address 2, 3, 4, 5)
++b → 6 (because floating pointer increment it by 4).
++a → 3
```

Q. int a=2;

```
char *b=(char *)a; → 

|      |
|------|
| 2    |
| 2000 |

 → integer value '2' is converted to character pointer
printf(b) → 2
* printf(*b) → garbage (value at address 2)
++b → 3 (increment by 1 only because character pointer)
++a → 3
```

Q. float a=10,00,000; → takes 4 Bytes

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

```
printf(b) → 1000
```

```
* printf(*b) → since b is an integer pointer, & hence will print 2 Bytes out of 4 Bytes (hence data loss)
```

a
10,00,000
1000 - 1003

b
1000
1000

★ if address lines are of 2 Bytes

then float a=10,00,000;

```
int *b=(int *)&a
```

we are storing a (4 Bytes data) into 2 Bytes integer pointer & hence the 4 Byte a (converted to integer pointer) is stored in 2 Byte 'b' & hence address loss.

Static Variable

```
main()
```

```
{ static int var = 5;
```

```
printf("%d", var--);
```

```
if (var)
```

```
main();
```

```
}
```

O/p:-

54321

- ★ Waiting 'Static' means at the time of compilation time memory is allotted. (Static binding) [compile time memory is allocated]
- ★ if 'Static' isn't there then the memory is allotted at run time. (dynamic binding) [memory is allocated at run time]

- ① For the static variable memory will be created only once during its life time (compile time only).
(Re Compiling the program will allocate different memory location to the static variable.)
- ② For the static variable initialization will be done once.
(By default)
- ③ By default Static & Global variable are initialized to zero.

Q. void main()

```
{ static int i = 5;
```

```
if (--i) 4
```

```
{ main();
```

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

```
}
```

```
}
```

O/p:-

0000

main()

i = 5

main()

4

main()

i = 4

main()

3

main()

i = 3

main()

2

main()

i = 2

main()

i = 1

main()

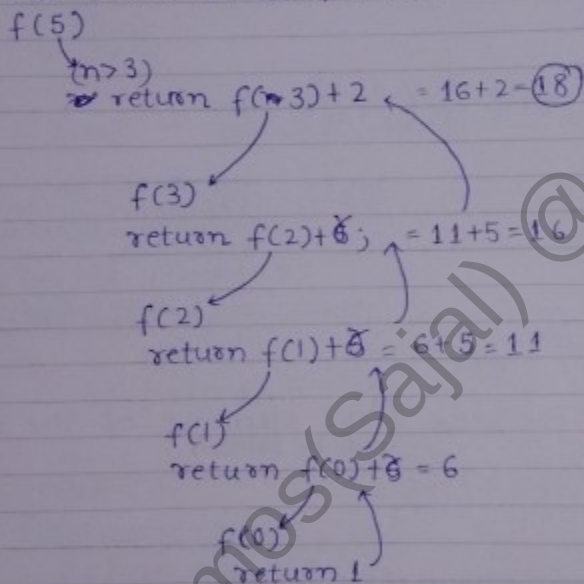
i = 0

```

Q) main() int f(int n)
{
    { static int x=0;
      if(n<=0)
        return 1;
      if(n>3)
        { x=n;
          return f(n-2)+2;
        }
      return f(n-1)+x;
    }
}

```

What is the value of $f(5)$?



x
☒ 5
 1000

Q/p:-
18

Note:- If we use register instead of static, then x is stored in register for faster access.

```

#include <stdio.h>
main()
{ struct a
  { char c[7];
    char *str;
  };
  struct b

```

```

{ char *c;
  struct a s1;
}

```

```

struct b s2 = {"Raipur", "Kampur", "Jaipur"};
printf("%s/%s", s2.c, s2.s1.st);
printf("%s/%s", ++s2.c, ++s2.s1.st);
}

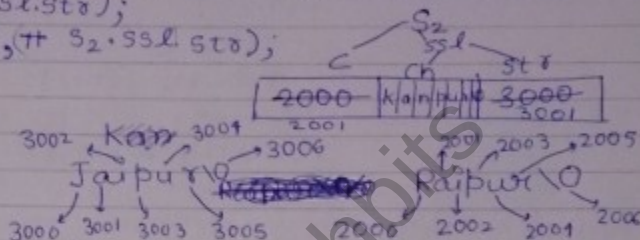
```

O/p:-

Raipur Jaipur
aipur aipur

★ int a=10;
int *b=a; //no error

a b
1000 2000



Structures & Union

```

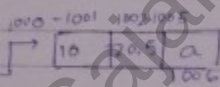
struct s1
{ int a;
  float b;
  char c;
}

```

```

struct s1 x;
x = {10, 20.5, 'a'};

```



NOTE:- Whenever we reference a struct, many variables are referenced & hence selection is req. to select the intended variable using

★ printf(x) → // means nothing. first element of the struct will be printed.

★ printf(x.a) → 10

★ printf(x.b) → 20.5

★ printf(x.c) → a

★ if struct s1

```

{ int a;
}

```

then s2.a not required s2 will also print a.

Both are possible

```

struct s1
{ int a;
  float b;
  char c;
}

```

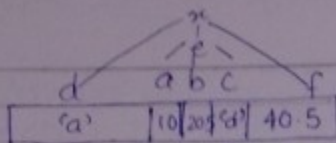
```

struct s2
{ char d;
  struct s1 e;
  float f;
}

```

← we can declare struct within struct.

s1 → 123



struct s₂ = { 'a', 10, 20.5, 'd', 40.5 }

struct x = { 'a', 10, 20.5, 'd', 40.5 }

x.d = 'a'

x.e.a = 10

x.e.b = 20.5

x.e.c = 'd'

x.f = 40.5

Q struct s
{
int a;
float b;
};

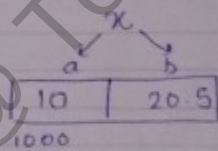
1. struct s, x = { 10, 20.5 };

2. struct s, *y = &x;

3. printf(y) → 1000

printf(&y, a); → 10

or printf(y, a); → 10



1000
2000

Q main()

{ struct s

{ char *z;

int i;

struct s, *p;

};

struct s, a[] = { { "Nagpur", 1, a+1 },

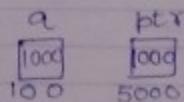
{ "Raipur", 2, a+2 },

{ "Kampur", 3, a+3 } };

struct s, *ptr = a;

printf("%s/%s/%s/%s", a[0].z, ptr->z, a[2].b->z);

(*ptr).z a[2].(*p).z



1000
2000

Nagpur/0
2000



1000

3000
2006

O/P:-

Nagpur Nagpur Nagpur

2000 1 1000 3000 2 1012 1000 3 1000

3000 Raipur/0 3006 4000 400

3006 4000 400

★ preprocessor will change $p \rightarrow c$ into $(*p).c$

main()

```
{ struct s;
```

```
{ char *z;
```

```
int i;
```

```
struct s, *p;
```

```
};
```

```
struct s, a[] = { {"Nagpur", 1, a1}, {"Raipur", 2, a2}, {"Kampur", 3, a3} };
```

```
struct s, *pt = a;  $\rightarrow (*pt).z$ 
```

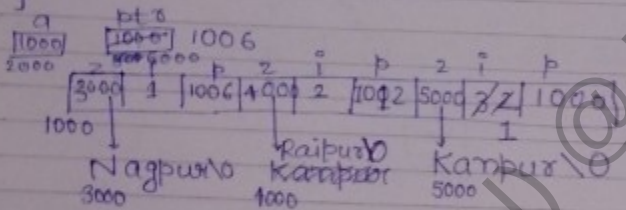
```
printf ("%s", ++(pt->z));
```

```
printf ("%s", a[(++pt->i)-1].z);
```

```
printf ("%s", a[--(pt->p->i)].z);
```

```
printf ("%d", --a[2].i);
```

```
}
```



O/p:-
agpur
Kampur
Kampur
Kampur
2 1

Q. #include <stdio.h>

```
struct Test
```

```
{ int i;
```

```
char *c;
```

```
};
```

```
struct Test st[] = { 5, "become", 4, "better", 6, "jungle", 8, "ancestor", 7, "brother" };
```

main()

```
{ struct Test *p = st;
```

```
p = p+1;
```

(*p).c

```
printf ("%s", ++(p->c));
```

```
printf ("%c", *(p->c));
```

```
printf ("%d", p[0].i);
```

```
printf ("%s", p->c);
```

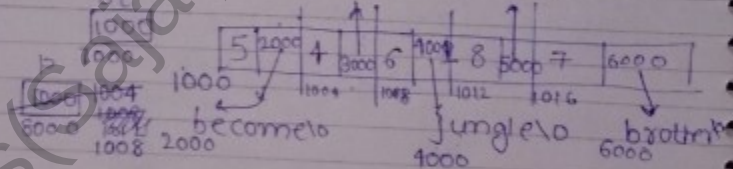
```
}
```

*(*p).c ++

O/p:-

etter, u, 6, ungle

p[0] = *(p+0)



```

Q. void main()
{
    char s[] = "hello";
    char R[] = "hello";
    if (S == R)
        printf("equal");
    else
        printf("not equal");
}

```

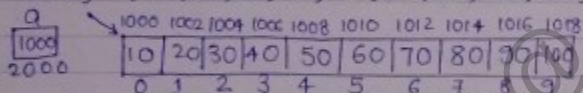
O/p:-
Not equal
because S & R both stores
diff. base addresses.

★ if *s == *R ('h' == 'h')
O/p:- equal

★ Arrays

1-D (1-Dimensional)

int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}



$$a+5 = 1000 + 2 \times 5 = 1010$$

$$L(a[5]) = 1000 + 2 \times (5 - 0) = 1010$$

$$L(a[9]) = 1000 + 2 \times (9 - 0) = 1018$$

if array starts with index '1'

$$L(a[5]) = 1000 + 2 \times (5 - 1) = 1008$$

$$L(a[9]) = 1000 + 2 \times (9 - 1) = 1016$$

in C compiler it is like
 $1000 + 2 \times 5$
(So C compiler saves '1' subtraction)

This means before
5th element, how many
elements are there.

Q. Array A is defined as follows:-

- A[-5...+5]
- Base address (BA) = 999
- Size of element = 100B

Location of a[-1], a[+5]

$$L(a[-1]) = 999 + 100 \times (-1 - (-5)) = 999 + 100 \times 4 = 1399$$

$$L(a[5]) = 999 + 100 \times (5 - (-5)) = 999 + 100 \times 10 = 1999$$

Total no. of elements:-

$$\boxed{\text{Last Index} - \text{1st Index} + 1}$$

In General :-

- $A[lb \dots ub]$
- $BA \rightarrow$ Base Address
- $C \rightarrow$ Size of 1 element

then location of i^{th} element :-

$$Loc(a[i]) = B.A. + C \times (i - lb)$$

Two-Dimensional Arrays

int a[4][4];

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

★ Every multidimensional array is stored in the form of multidimensional or 1-dimensional array.

Row-Major Order :-

1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022	1024	1026	1028	1030
a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	a_{41}	a_{42}	a_{43}	a_{44}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$$L(a[i][j]) = B.A. + C \times ((i - lb_1) \times (ub_2 - lb_2 + 1) + (j - lb_2))$$

$$L(a[4][3]) = 1000 + 4(4-1) + (3-1) \times 2$$

$$= 1000 + 28$$

$$= 1028$$

$$Loc(a[2][4]) = 1000 + 4(2-1) + (4-1) \times 2$$

$$= 1000 + 7 \times 2$$

$$= 1014$$

Total Rows

Last Row No. -
1st Row No. + 1

Total Columns

Last Column No. -
1st Column No. + 1

array starting
from $a[1][1]$

Q. Consider the following 2-D array :-

$$a[5 \dots 25, 50 \dots 75]$$

$$B.A = 0$$

$$C = 58$$

$$\begin{aligned} L(a[20][70]) &= 0 + (70-50)(20-50) + (70-50) \times 5 \\ &= (15 \times 25 + 20) \times 5 \\ &= 395 \times 5 \\ &= 1975 \end{aligned}$$

$$\begin{aligned} \text{Loc}(a[25][75]) &= 0 + ((25-50) \times (75-50+1) + (75-50)) \times 5 \\ &= (20 \times 26 + 25) \times 5 \\ &= 445 \times 5 \\ &= 2225 \end{aligned}$$

In General (Row Major Order)

$$A[lb_1 \dots ub_1, lb_2 \dots ub_2]$$

$$BA, C$$

$$\text{Loc}(a[i][j]) = BA + ((i-lb_1) \times (ub_2-lb_2+1) + (j-lb_2)) \times C$$

Date _____

11.08.12

```
• int a[4][4];
```

$$a_{11} \quad a_{12} \quad a_{13} \quad a_{14}$$
$$a_{21} \quad a_{22} \quad a_{23} \quad a_{24}$$
$$a_{31} \quad a_{32} \quad a_{33} \quad a_{34}$$
$$a \quad a_{41} \quad a_{42} \quad a_{43} \quad a_{44}$$
$$a[t_1, ub_1][t_2, ub_2]$$

a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

$$Loc(a[i][j]) = B.A. + ((j - lb_2) \times (ub_1 - lb_1 + 1) + (i - lb_1)) \times C$$

$$\begin{aligned} \cdot \text{Loc}(a[3][4]) &= 1000 + ((4-1) \times 4 + (3-1)) \times 2 \\ &= 1000 + (12 + 2) \times 2 \\ &= 1000 + 28 \\ &= 1028 \end{aligned}$$

$$\begin{aligned}\cdot \text{LOC}(a[i][j]) &= 1000 + ((3-1) \times 4 + (4-1)) \times 2 \\ &= 1000 + (8+3) \times 2 \\ &= 1000 + 22 \\ &= 1022\end{aligned}$$

$$\begin{aligned} \cdot \text{Loc}(a[4][1]) &= 1000 + ((1-1) \times 4 + (4-1)) \times 2 \\ &= 1000 + 6 = 1006 \end{aligned}$$

Q. Consider the array (using Column Major Order)

$$A[-5..+20][25..75]$$

• BA = 999

• $C = 10 \text{ B}$

$$\begin{aligned} \text{Loc}(A[0][74]) &= 999 + ((74-25) \times (20-(-5)+1) + (0-(-5))) \times 10 \\ &= 999 + (49 \times 26 + 5) \times 10 \\ &= 999 + 12790 \\ &= 13789 \end{aligned}$$

$$\begin{aligned}\text{Loc}(A[-2][30]) &= 999 + ((30-25) \times 26 + (-2+5)) \times 10 \\ &= 999 + (130+3) \times 10 \\ &= 999 + 1330 = 2329\end{aligned}$$

$$\begin{array}{r} 5 \\ 49 \\ \times 26 \\ \hline 294 \\ 980 \\ \hline 1274 \\ + 5 \\ \hline \end{array}$$

Lower Triangular Matrix

int a[4][4]

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

*it is valid only for Square matrices.

$$A[i][j] = 0 \text{ if } i < j$$

Storing using Row Major :-

a	1000	02	04	06	08	10	12	14	16	18
	a ₁₁	a ₂₁	a ₂₂	a ₃₁	a ₃₂	a ₃₃	a ₄₁	a ₄₂	a ₄₃	a ₄₄
	1st	2nd		3rd			4th			

$$\begin{aligned} \text{LOC}(a[4][3]) &= 1000 + ((4-1) \times ((1+2+3) \times (4-1) + (3-1))) \times 2 \\ &= 1000 + 18 \\ &= 1000 + ((4-1)(1+2+3) + (3-1)) \times 2 \\ &= 1000 + (8) \times 2 \\ &= 1016 \end{aligned}$$

$$\text{LOC}(a[i][j]) = \boxed{B.A. + ((i-lb_i)NNS + (j-lb_j)) \times C}$$

$(i-lb_i) \rightarrow$ Natural No Sums upto from 1 to $(i-lb_i)$

$\rightarrow (4-1)$ sum of natural nos. (i.e. sum of natural nos. upto 3.)

$$\begin{aligned} \text{LOC}(a[4][2]) &= 1000 + ((1+2+3) + (2-1)) \times 2 \\ &= 1000 + (6+1) \times 2 = 1014 \end{aligned}$$

Q. Consider matrix defined as follows:-

$$A[-25] \dots +25, -25, \dots, +25]$$

$$BA = 0$$

Row Major Order, lower Triangular Matrix

$$C = 100B$$

$$\begin{aligned} \text{LOC}(A[-20][-21]) &= 0 + ((-20+25)NNS + (-21+25)) \times 1000 \\ &= 0 + ((1+2+3+4+5) + 4) \times 100 \\ &= 0 + (19) \times 100 \\ &= 1900 \end{aligned}$$

$$\frac{5 \times 4^3}{2} = 1$$

$$\begin{aligned} \text{Loc}(A[0][0]) &= 0 + ((0+25) \text{NNS} + (0+25)) \times 100 \\ &= 0 + \left(\frac{25 \times 26^{13}}{2} + 25 \right) \times 100 = 35,000 \end{aligned}$$

Column Major Order:-

00	02	04	06	08	10	12	14	16	18
a ₁₁	a ₂₁	a ₃₁	a ₄₁	a ₂₂	a ₃₂	a ₄₂	a ₃₃	a ₄₃	a ₄₄

My answer

$$\text{Loc}(a[i][j]) = B.A + \left(\frac{14 \text{ to } j}{1 \text{ to } 10} \right) \text{reverse order sum} + \left(\frac{8-j}{1 \text{ to } 10} \right) \times C$$

$$\begin{aligned} \text{Loc}(a[4][3]) &= 1000 + ((4+3) + (4-3)) \times 2 \\ &= 1000 + (7+1) \times 2 \\ &= 1010 \quad 1020 \quad 1016 \end{aligned}$$

★

$$\text{or } \text{Loc}(a[4][3]) = 1000 + \left(\frac{4 \times 5}{2} \right) - (4-3+1) \text{NNS} + (4-3) \times 2$$

we actually want to go to 3rd column

a ₁₁	a ₂₀	a ₃₀	0
a ₂₁	a ₂₂	0	0
a ₃₁	a ₃₂	a ₃₃	0
a ₄₁	a ₄₂	a ₄₃	a ₄₄

we are at here (at the end)

we are at here actually after a₄₂ [after subtracting]

$$\begin{aligned} &= 1000 + (10 - (2+1) + 1) \times 2 \\ &= (1000 + 8) \times 2 \\ &= 1016 \end{aligned}$$

$$\frac{4 \times 5}{2}$$

$$\begin{aligned} \text{Loc}(a[4][1]) &= 1000 + \left(\frac{4 \times 5}{2} - (4-1+1) \text{NNS} + (4-1) \right) \times 2 \\ &= 1000 + (10 - 10 + 3) \times 2 \\ &= 1000 + 6 = 1006 \end{aligned}$$

int * a = 2;
pf(*a);

$A[lb_1, ub_1, lb_2, ub_2]$

skip all
columns

go to
the starting
of the desired
column

go to
the desired
row

$$LOC(A[i][j]) = B.A + ((ub_2 - lb_2) - (ub_2 - j + 1)NNS + (i - j)) \times C$$

Q. $A[-5 \dots +5, -5 \dots +5]$

$BA = 1000, C = 10$

Lower Triangular Matrix
Column Major Order

$$\frac{66 - 21}{2} = 45$$

$$\bullet LOC(A[0][0]) = 1000 + (11^{NNS} - (5 - 0 + 1)NNS + 1)$$

$$= 1000 + \left(\frac{11 \times 12^6}{2} - \frac{3 \times 7}{2} + (0 - 0) \right) \times 10$$

$$\frac{7 \times 8 - 24}{2} = 24$$

$$= 1000 + 45 \times 10$$

$$= 1450$$

$$\bullet LOC(A[2][1]) = 1000 + (11NNS - (5 - 1 + 1)NNS + (2 + 1)) \times 10$$

$$= 1000 + (66 - 28 + 3) \times 10$$

$$= 1000 + (41) \times 10$$

$$= 1410$$

Tri-diagonal Matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{42} & a_{44} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{42} & a_{44} \end{bmatrix}$$

$$N-1 + N + N-1 = 3N-2 \text{ memory space.}$$

★ In Tridiagonal Matrix, apart from 1st & last row (which has 2 elements), rest all other rows have 3 elements.

a. Using Row Major :-

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}
1000	02	04	06	08	10	12	14	16	18	

$$\text{Loc}(a[3][4]) = 1000 + (3(2+3) + (4-2)) \times 2$$

$$= 1000 + (5+2) \times 2$$

$$= 1014$$

each row contains 3 elements, skipping desired no. of rows

but 1st row contains only 2 elements, subtracting 1 element

$$\boxed{\text{Loc}(a[i][j]) = B.A. + (3 \times (j - lb_1) - 1 + (i - j + 1)) \times C}$$

$$\text{Loc}(a[4][4]) = 1000 + (3 \times (4-1) - 1 + (4-4+1)) \times 2$$

$$= 1000 + 9 \times 2 = 1018$$

going to the desired column.

Using Column Major

$$\boxed{\text{LOC}(A[i][j]) = B.A. + (3 \times (j - lb_2) - 1 + (i - j + 1)) \times C}$$

$$\text{LOC}(A[3][4]) = 1000 + (3 \times (4-1) - 1 + (3-4+1)) \times 2$$

$$= 1000 + (8) \times 2$$

$$= 1016$$

Using Diagonal by Diagonal :- (Principal diagonal, Lower, Upper)

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

→

principal	Lower	Upper
a_{11}	a_{21}	a_{12}
a_{22}	a_{32}	a_{23}
a_{33}	a_{43}	a_{34}
a_{44}		

$\text{Loc}(A[i][j]) \rightarrow$ if $(i=j)$ (Principal diagonal)

$$= B.A. + (j - lb_1) \times C \quad [\text{if } i=j]$$

$$= B.A. + ((ub_2 - lb_2 + 1) + j - lb_2) \times C \quad [\text{if } i > j]$$

$$N(ub_2 - lb_2 + 1 = N)$$

$$= [B.A + (N + (N-1) + (i - lb_1)) \times C \quad (\text{if } i < j)]$$

Z-Matrix

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

Storing :-

- ① Row Major
- ② Column Major
- ③ Store 1st row, last row & remaining elements in diagonal.

Row-Maj

Using 3rd approach -

a_{11}	a_{12}	a_{13}	a_{14}	a_{41}	a_{42}	a_{43}	a_{44}	a_{23}	a_{32}
↓ N				↓ N				↓ N-2	

∴ total space = $3N-2$

$$Loc(A[i][j]) = \begin{cases} B.A + (j - lb_2) \times C & [\text{if } i = lb_1] \\ B.A + (N + (j - lb_2)) \times C & [\text{if } i = ub_1] \\ B.A + (2N + (i - lb_1 - 1)) \times C & [\text{if } j \neq lb_1 \text{ \& } j \neq ub_1 \text{ \& } (i+j) = N+1] \end{cases}$$

Toeplitz Matrix

if $A[i][j] = A[i-1][j-1] \quad \forall i, j$ where $i > 1 \text{ \& } j > 1$

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

Store only $(2N-1)$ elements

10	20	30	40
50	10	10	30
20	50	10	20
70	60	50	10

Store only these

Storing:-

- ① Row-Order
- ② Column-Order
- ③ Store 1st row, remaining element in 1st column

1000	a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{41}
	10	20	30	40	50	60	70
	00	02	04	06	08	10	12

$$\text{Loc}(a[i][j]) = \begin{cases} B.A + (j - \text{lb}_2) \times C & (\text{if } i = \text{lb}_1) \\ B.A + (N + (i - \text{lb}_1 - 1)) \times C & (\text{if } i > \text{lb}_1) \end{cases}$$

[Elements belongs to upper part.]
[Elements belongs to lower part.]

Three-Dimensional Matrix

int a[2][3][4] → 2 arrays of size 3x4

BA = 1000

Row Major Order

$$\begin{aligned} \text{LOC}(A[2][3][4]) &= 1000 + ((2-1) \times 12 + (3-1) \times 4 + (4-1)) \times 2 \\ &= 1000 + (12 + 8 + 3) \times 2 \\ &= 1000 + 46 = 1046 \end{aligned}$$

Loc & A

int a[-5...+5][-10...+10][-25...+25]

BA = 0

Column Major Order

C = 5

$$\begin{aligned} \text{Loc}(A[0][0][0]) &= 0 + (5 \times 61 \times 51 + (0+25) \times (61) + (0+10)) \times 5 \\ &= (3111 + 1525 + 10) \times 5 \\ &= 85450 \end{aligned}$$

$$\begin{array}{r} 3111 \\ \times 5 \\ \hline 15555 \\ 1535 \\ \hline 17090 \end{array} \quad \begin{array}{r} 61 \\ \times 25 \\ \hline 1525 \\ 3050 \\ \hline 3111 \end{array} \quad \begin{array}{r} 61 \\ \times 51 \\ \hline 161 \\ 3050 \\ \hline 3111 \end{array}$$

$$A[lb_1 \dots ub_1][lb_2 \dots ub_2][lb_3 \dots ub_3]$$

Row Major Order

$$LOC(a[i][j][k]) = B.A + ((i-lb_1) \times (ub_2-lb_2+1)(ub_3-lb_3+1) + (j-lb_2) \times (ub_3-lb_3+1) + (k-lb_3)) \times C$$

Column Major Order

$$Loc(a[i][j][k]) = B.A + ((i-lb_1) \times (ub_2-lb_2+1)(ub_3-lb_3+1) + (k-lb_3) \times (ub_2-lb_2+1) + (j-lb_2)) \times C$$

specifying this is optional

Q. `int a[5] = {10, 20, 30, 40, 50};`

`int a[][3] = {10, 20, 30, 40, 50, 60};`

`int a[][2][3]`

Optional mandatory

`int a[5];` → garbage values

`int a[5] = {10, 20, 30};`

4th & 5th elements are 0.

`int a[5] = {};` → all are 0.

`int a[5] = {10, 20, 30, 40, 50, 60};`

no compilation error, but `a[5]` is not accessible

Note:-

In multidimensional Array, 1st indices are mandatory.

Q. `main()`

`{ int a[3][3];`

`printf("%d", ((a = *a) + (*a = a[0])));`

`a = 1000`

`*a = 1000`

`*a = 1000`

`a[0] = *(a+0) = 1000`

`[0][0] = 1`

`a = 1000`

`a+0 = 1000 ⇒ *(a+0) ⇒ *1000 = 1000`

`*a+1 = 1006`

`a+2 = 1012`

`*(a+1) = *1006 = 1006`

`*(a+2) = *1012 = 1012`

row is not selected

row is selected. `*a+1 = a[1]`

`a = 1000`
`a+1 = 1006`

`*(a+1) = 1006`

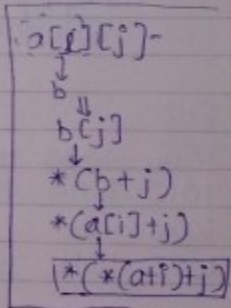
`*(a+1)+1 = 1008`

row is selected (we can't change the row now.)

row is not selected & hence we can change the row.

`(a+1) = 1006`

`(a+1)+1 = 1012`



00	01	02
10	20	30
40	50	60
70	80	90

- $a = 1000$
- $a+1 = 1006 \rightarrow$ row is not selected
- $*(a+1) = 1006 \rightarrow$ row is selected
- $*(a+1)+1 = 1008$
- $*(*(a+1)+1) = 50$
- $*(*(a+1)+1)+1 = 51$

Q. main()

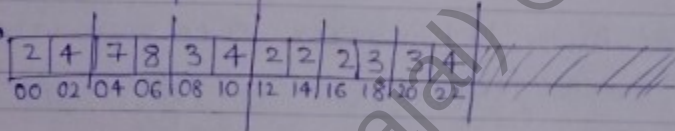
```

{ int a[2][3][2] = {2,4,7,8,3,4,2,2,2,3,3,4};
  printf("%u%u%u%u", a, *a, **a, ***a);
  printf("%u%u%u%u", a+1, *a+1, **a+1, ***a+1);
  printf("%u%u%u%u", *(a+2)+3, *(a[1]+2), *(a+2), *(a[1]+2));
  printf("%u%u%u%u", a+1, a[1]+1, a[1][2]+1, a[1][2][3]+1);
}

```

$a[2][3][2] =$

$a = 1000$



$*a \rightarrow$ array selected
 $**a \rightarrow$ array 2
 row selected
 $***a \rightarrow$ array, row
 & column
 selected

O/P:-

1000	1000	1000	2
1012	1004	1002	3
1036	2	1024	2
1012	1016	1022	garbage

3rd printf:-

$*(a[1]+2)$
 $\downarrow$
 $*(*(a+1)+2)$

$*a \rightarrow$ 0th array
 base address
 $**a \rightarrow$ 0th array
 0th row
 base
 address

★ $*(a+2)$

$\downarrow$
 1024
 array selected

★ $*(a[1]+1)$
 $\downarrow$
 1012
 array selected

$\downarrow$
 skip 1 row = $1012 + 4 = 1016$

row selected, column selected

row selected, array selected

$*(a+1)+2$

$\downarrow$
 $1012 + 2 \times 2$
 $= 1016$

$\downarrow$
 1016
 column selected

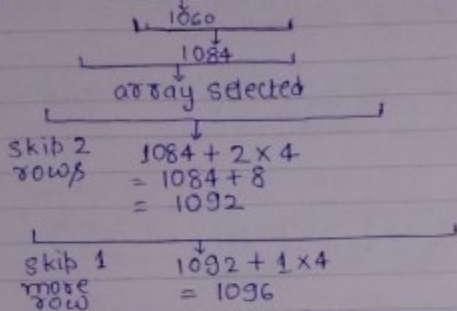
$*(a[2]+3)$

$\downarrow$
 1024
 array selected

$\downarrow$
 row $\rightarrow 1024 + 3 \times 2 = 1036$
 row selected

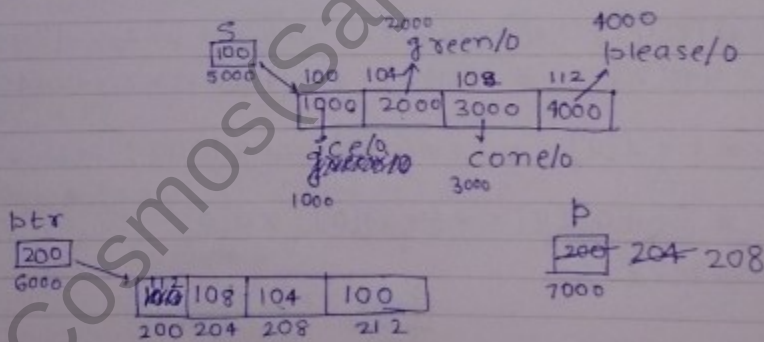
$$12 \times 7 = 84$$

$$Q. ((*(a+5)+2)+2)+1$$



```

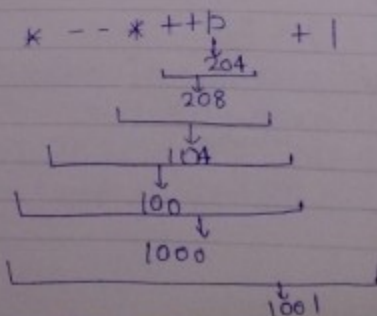
#include <stdio.h>
main()
{
    char *s[] = {"ice", "green", "cone", "please"};
    char **ptr[] = {s+3, s+2, s+1, s};
    char ***p = ptr;
    printf("%s", **++p);
    printf("%s", *--*++p+1);
}
  
```



O/p:-

```

b = 200
++p = 204
*++p = 3000
**++p = cone
  
```



∴ O/p:-
cone
ce

Q #include <stdio.h>

int a1[] = {6, 7, 8, 34, 67};

int a2[] = {25, 56, 28, 29};

int a3[] = {-12, 27, -31};

int *x[] = {a1, a2, a3};

main()

{ B(x);

}

B(int **a)

{ printf("%d", a[0][2]);

printf("%d", *a[2]);

printf("%d", **a++ + a[0]);

printf("%d", *(++a)[0]);

}

O/p :-

1st printf :- a[0][2] = *((a+0) + 2)

1st printf :- a[0][2] = *((a+0) + 2) = *(1000 + 2 * 2) = *(1004)

2nd printf :- *a[2] = **((a+2)) = **((4000 + 2 * 2)) = **((4004)) = *(3000)

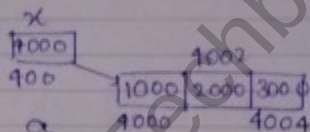
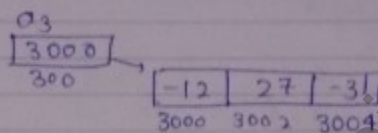
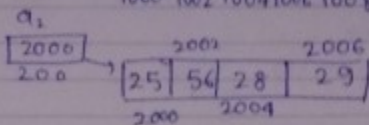
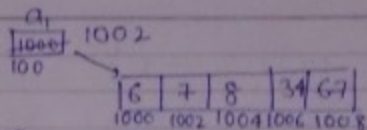
3rd printf :- **a++ + a[0] = **((a+0)) + a[0] = *(4000) + *(1000) = *(1002)

4th printf :- *(++a)[0] = *((++a)[0]) = *a[0] = *(1002) = 1002

even though
↑ x is 1-D pointer array.

x[1][2] :- **((x+1) + 2) = 28

★ using one dimensional array of pointer, we can get two-dimensional array.



O/p :-
8
-12
7
25

int a[5] → 1D

★ int a[5][5] → 2-D array, where every row contain uniform no. of columns.

★ int *x[] → 2-D array, but it will create variable columns in every row.

★ int a1[] = {6, 7, 8, 34, 67};
int a2[] = {25, 56, 28, 29};
int a3[] = {-12, 27, -31};
int a4[] = {10, 20};
int a5[] = {20};

int *x[] = {a1, a2, a3, a4, a5};

100	200	300	400	500
-----	-----	-----	-----	-----

Q. consider following C-declarations:-

(1) int A[10][10];

(2) int *B[10];

Which is true?

(a) A[5][6] = 10; T

(b) A[5] = 1000 F

(c) B[5][6] = 10; T

(d) B[5] = 1000 T

Q. main()

{ char string[] = "Hello World";
display(string);
}

void display(char *string)
{ printf("%s", string);
}

String

1000

H	e	l	l	o	W	o	r	l	d
01	03	05	07	09	11				
1000	02	04	06	08	10				

★ In the above program, we are using display fn. w/o declaring so compiler assumes display fn. by default integer value, but above program is returning void, so above program will give error of type mismatch.

Q. double foo(double); /* line 1 */
int main()

★ if we delete line 1 then conflicting datatype for fn. foo. type mismatch

```
{ double da, db;
  db = foo(da);
  return 0;
double foo(double a)
{ return a;
}
```

Q. void main()
{ int i=0;
for (int i=0; i<20; i++)

★ In switch statement, we can write default wherever we want, but it will be executed whenever there is no match.

{ switch(i)

```
{ case 0: i+=5; // we can
  case 1: i+=2; // but default
  case 5: i+=5; // anywhere
  default: i+=4; // we will have
  break; // same result.
```

```
{ printf("%d", i);
}
```

O/p:-

i=i+5

5

16

27 22 21

32

37

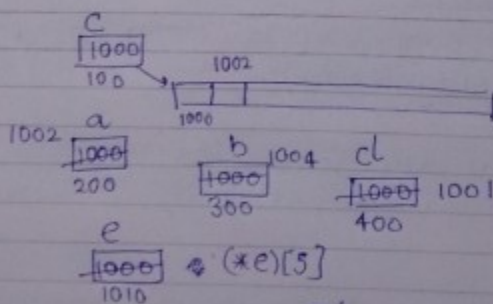
39

44

49

54

Q. main()
{ int c[100];
int *a = (int *)c;
float *b = (float *)c;
char *d = (char *)c;
int (*e)[5] = (int *)c;
for (i=0 to 5)



```
{ printf(c+i, a, b, d, e);
  a++;
  b++;
  d++;
  e++;
}
```

O/p:-

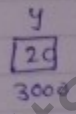
i	0	1	2	3	4	5
0	1000	1000	1000	1000	1000	1000
1	1002	1002	1004	1006	1008	1010
2	1004	1004	1008	1012	1016	1020
3	1006	1006	1012	1016	1020	1024
4	1008	1008	1016	1020	1024	1028
5	1010	1010	1020	1024	1028	1032

★ $\text{int} (*a)[5] \rightarrow$ a is a pointer to array which contains 5 elements,
 when we do $a++$, then a is incremented to 10B.

Function returning pointer :-

```
int fun();
main()
{
    int x=10;
    int *b;
    b = fun();
    printf("%d", x+(*b));
}
O/p:-
10+20=30
```

```
int *fun()
{
    int y=20;
    return (&y);
}
```

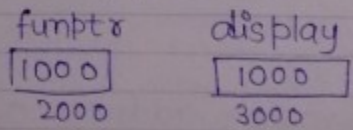


★ function will get the memory when the fn is called, & will be deallocated when the fn. returns.

★ fun is a function which returns pointer.

Functional Pointer

```
main()
{
    int display(); //creating memory for display
    int (*funptr)(); //fn whose address is allocated
    funptr = display;
    (*funptr)();
}
//value at 1000 (at 1000, it is the fn. display)
funptr is a pointer, which internally contain function's address.
```



★ function name indicates base address
 display indicates its base address.

O/p:-
 Hi

$\star \text{int } (*p)() \rightarrow p$ is pointer pointing to a function.
 $\text{int } *p[5] \rightarrow p$ is an array of pointers pointing to 5 addresses of int variables.
 $\text{int } *p[5]() \rightarrow p$ is an array of pointers pointing to 5 fn. which are returning integer.

Q. `main()`
`{ int (*ptr[3])();`
`ptr[0] = aaa;`
`ptr[1] = bbb;`
`ptr[2] = ccc;`
`ptr[2]();` or `(*ptr[2])()`
`}`

`1000 int aaa()`
`{ pf("aaa");`
`return 0;`
`}`

`2000 int bbb()`
`{ pf("bbb");`
`}`

`3000 int ccc()`
`{ pf("ccc");`
`}`

ptr
 4000 4002 4004
 1000 2000 3000
 0 1 2

$\star \text{ptr}[0] = \star \star (\text{ptr} + 0)$
 $= \star \star (1000)$
 $= \star (4000)$
 • value at address 4000

$(\star \text{ptr}[2])()$
 $\downarrow$
 $(\star \star (\text{ptr} + 2))()$
 $\downarrow$
 $(\star \star (4004))()$
 $\downarrow$
 $(\star (3000))()$
 at ccc fn.

Q. What does the following C-statement indicate
`int (*f)(int, int);`
 $\star f$ is a pointer contains address of a function which is taking two int variables as argument & returning an integer.

`int * (*f)(int *, int *);`
 f is a pointer containing address of a function which takes 2 integer pointers as arguments & returns integer pointer.

② `void (*abc)(int, void (*def)());`
 abc is a pointer containing address of a function which takes 2 arguments as i/p, out of which one is integer & the other is ② def is a pointer pointing to a fn. which returns void.
 & returns void.

③ `char * (*(*a[N])())()`

array of N pointers
 pointing to function(s)
 [doesn't take any input]

these function(s) are
 returning pointer(s) (pointer type
 is not specified)

these returned pointers
 are pointing to a function.

this function is returning
 character pointer.

* Array of N pointers pointing to functions which will return a pointer which is pointing to a function which is returning character pointer.

④ `(char * (*()) (*ptr[N])())`

(i) a pointer
 is pointing
 to a function

(ii) the function returns
 a character pointer.

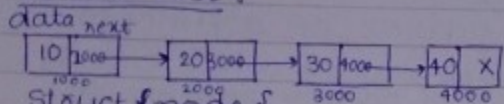
(iii)

(iii) character pointer
 is pointing to an
 array of pointers
 which are pointing
 to functions which
 are returning
 a pointer.

brackets are
 having associativity
 from left to right

12.08.12

Linked List :-

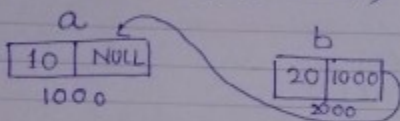


```
struct node {
    int data;
```

```
    struct node *next;
```

```
};
```

```
struct node a = {10, NULL};
```



```
struct node b = {20, &a};
```

```
struct node *ptr = &b;
```

Self referential data structure

- float type pointer is pointing to float.
- int type pointer is pointing to int.

ptr = 2000

*ptr $\rightarrow$ inside 2000 (i.e. selected the address 2000).

(*ptr).next = 1000

(*ptr).next :- selected the address 1000
 (*(*ptr).next), data = 10

(*(ptr $\rightarrow$ next)).data

(ptr $\rightarrow$ next) $\rightarrow$ data = 10

Q. Write a C program to print all data fields parts in every node of the linked list

Ans.

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

```
struct
```

```
main()
```

```
{ struct node *start = //starting address of linked list
```

```
    while
```

```
    ptr = start;
```

```
    while (ptr != NULL)
```

```
    { printf("%d", ptr  $\rightarrow$  data);
```

```
      ptr = (*ptr).next;
```

```
    }
```

```
}
```

```
int a;
int *ptr = &a;
ptr = (int*) a;
```

*if s is NULL, then $*s$ is not allowed (it gives segmentation error)

Q. Write a C program to return data of the last node of linked list.

Ans. `ptr = start;`
`while (ptr != NULL)` // Linked list contain atleast 1 node.
`{ if ((ptr->next) == NULL)`
`{ printf("%d", (*ptr).data);`
`{`
`ptr = (*ptr).next;`
`}`

★ struct node {

int data;

struct node *next;

};

int Display(struct node *s)

{ struct node *p = s;

if (p == NULL)

{ printf("Linked list is empty");

exit(0);

}

while (p->next != NULL)

{ p = p->next;

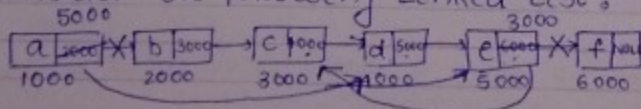
}

printf("%d", p->data);

}

// Linked list contains atleast 2 elements

Q. Consider the following Linked List :-



What would be the O/P after executing following sequence of statements :-

```

int a = 10;
int *b = &a;
printf ("%d", *b);

```

p
3000

first
1000

```

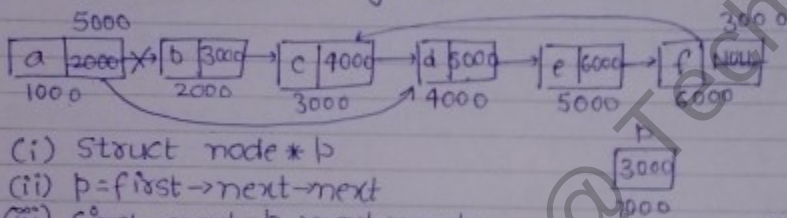
struct node *p;
p = first->next->next;
first->next = p->next->next;
p->next->next->next = p;
printf (first->next->next->next->data);

```

O/P:-

a

2. Consider the following linked list:-

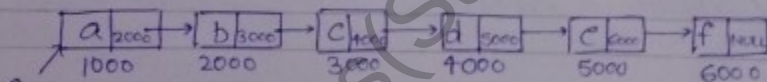


- Struct node *p
- p = first->next->next
- first->next = p->next->next;
- p->next->next->next->next = p;
- printf (first->next->next->next->data);

O/P:-

c

Adding Node in Linked List



(i) Starting of the linked list:- Start

Add at Start (struct node *s, int x)

```

{ struct node *ptr = (node *) malloc (sizeof (node));
  (*ptr).next = s;
  (*ptr).data = 'g';
  s = ptr;
}

```

(ii) Write a C program to add a node with data x at the end of the linked list.

```

Addatend (node * start)
{
    node * ptr = (node *) malloc (sizeof (node));
    (*ptr).data = x;
    node * temp = start;
    while (ptr->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = ptr;
    ptr->next = NULL;
}

```

(iii) Write a C program to add a node with data x after a node with data y.

```

→ Addinbtw (node * start)
{
    node * ptr = (node *) malloc (sizeof (node));
    (*ptr).data = x;
    node * temp = start;
    while (temp->data != y)
    {
        temp = temp->next;
        if (temp == NULL)
        {
            exit(0);
        }
    }
    ptr->next = temp->next;
    temp->next = ptr;
}

```

(iv) Write a C program to add a node with data x before a node with data y.

```

Addbefore (node * start)
{
    node * ptr = (node *) malloc (sizeof (node));
    (*ptr).data = x;
    node * temp = start;
    node * tempbef = start;
    while (temp->data != y)
    {
        tempbef = temp;
        temp = temp->next;
        if (temp == NULL) exit(0);
    }
    ptr->next = temp;
    tempbef->next = ptr;
}

```

```

if (tempbef == start)
{
    ptr->next = start;
    start = ptr; exit(0);
}
temp->next = ke
ptr->next = tempbef->next;
tempbef->next = ptr;
}

```

Or

```

Addbef (struct node *start)
{
    node *p = (node *) malloc (sizeof (node));
    p->data = x;
    node *q = start;
    if (q == NULL)
        return(1);
    if (q->data == 'y')
    {
        p->next = q;
        start = p;
    }
    while (q->next != NULL && (q->next)->data != 'y')
    {
        q = q->next;
    }
    if (q->next == NULL)
    {
        exit(1);
    }
    q->next = p->next;
    p->next = q;
}

```

Q. Which one of the following statement, adding an element x after current position

- current = Newnode(x, current)
- current = Newnode(x, current->next);
- current->next = Newnode(x, current);
- ☒ current->next = Newnode(x, current->next);

Q Consider the fn. f defined below:-

struct item

```
int data;
```

```
struct item *next;
```

 $\zeta;$

```
int f(struct item *b)
```

3

```
return((p==NULL)|| (p->next==NULL)|| ((p->data <= p->next->data)
&& f(p->next))));
```

3

for a given LL, the fn. f returns 1 iff

(a) if L^0 is empty or exactly one element

(b) elements in the LL are sorted in non-decreasing order.

(c) " " " " " " " -increasing

(d) none of the above.

Q Write a C program to delete a node at the beginning of the linked list.

~~del~~ del(node *start)

```

} node * ptr = Start;

```

```
Start = (*Start).next;
```

```
free(ptr); // the memory is deallocated pointed by ptr now
```

34

ptr = NULL; // any segmentation errors, we perform ptr = NULL

2

* if we write int a, then we can't deallocate the memory using free()

Q Write a C program to delete a node at the end of LL.
delatEnd (node *start)

$$\text{delatEnd}(\text{node} * \text{start})$$

```
if (node * ptr = start, prev = start;
```

if (ptx == NULL)

```
exit(1);
```

2

```

while (ptr->next != NULL)
{
    prev = ptr;
    ptr = ptr->next;
}
if (prev == start)
{
    free(prev);
    start = prev = ptr = NULL;
}
else
{
    prev->next = NULL;
    free(ptr);
    ptr = NULL;
}
}

```

Q. Write a C program to delete a node before which contains data x.

```

del (struct node * start)
{
    node * ptr = start, * prev = start;
    if (ptr == NULL)
    {
        exit(1);
    }
    while (ptr->data != 'x' && ptr->next != NULL)
    {
        prev = ptr;
        ptr = ptr->next;
    }
    if (ptr == NULL) { exit(1); }
    if (prev == ptr == start)
    {
        start = start->next;
        free(ptr);
        ptr = NULL;
    }
    else
    {
        prev->next = ptr->next;
        free(ptr);
        ptr = NULL;
    }
}

```

Q Write a C program to delete a node before x.
Using 3 pointers

```
del (node *start)
```

```
{ node *ptr = start, *prev = start, prevtwobef = start;
```

```
  if (ptr == NULL)
```

```
    {exit(1);
```

```
    }
```

```
  while(
```

```
    if (start->data == 'x')
```

```
      {exit(0);
```

```
      }
```

```
    if (start->next->data == 'x')
```

```
      {start = start->next;
```

```
        free(ptr);
```

```
        ptr = prev = prevtwobef = NULL;
```

```
    }
```

```
  while(
```

```
    ptr = (start->next)->next;
```

```
    prev = start->next;
```

```
    prevtwobef = start;
```

```
    while (ptr->data != 'x' && ptr->next != NULL)
```

```
      {prevtwobef = prev;
```

```
        prev = ptr;
```

```
        ptr = ptr->next;
```

```
      }
```

```
    if (ptr->next == NULL && ptr->data != 'x')
```

```
      {exit(1);
```

```
      }
```

```
    prevtwobef->next = prev->next;
```

```
    free(prev);
```

```
    prevtwobef = prev = ptr = NULL;
```

```
  }
```

Or

Using 2 pointers

del (node * start)

```
{ node * ptr = start, * prev = start;
```

```
  if (start == NULL)
```

```
    { exit(1);
```

```
    }
```

```
  if (start->data == 'x')
```

```
    { exit(1);
```

```
    }
```

```
  if (start->next->data == 'x')
```

```
    { start = start->next;
```

```
      prev = ptr = NULL;
```

```
      exit(0);
```

```
    }
```

```
  while (ptr->next
```

```
    ptr = start->next->next;
```

```
    prev = start->next;
```

```
    while ((ptr->next->data != 'x' && (ptr->next) != NULL)
```

```
      { prev = ptr;
```

```
        ptr = ptr->next;
```

```
      }
```

```
    if ((ptr->next) == NULL && ptr->data != 'x')
```

```
      { exit(1);
```

```
      }
```

```
    if (prev->next = ptr->next;
```

```
        free(ptr);
```

```
        ptr = prev = NULL;
```

```
  }
```

Using 1 pointer

del (node * start)

```
{ node * ptr = start;
```

```
  if (start == NULL)
```

```
    { exit(1);
```

```
    }
```

```
  if (start->data == 'x')
```

```
    { exit(0);
```

```
    }
```

Note:- In order to delete any node in the given linked list apart from last node two pointers (minimum 2 pointers) are required.

Q. Write a program to find middle of the linked list

```
find(node *start)
{
    node * ptr = start;
    int i = 0;
    while (ptr->next != NULL)
    {
        i++;
        ptr = ptr->next;
    }
    if (ptr == NULL)
    {
        exit(1);
    }
    if (ptr == start)
    {
        return (ptr);
    }
    if (i % 2 == 0)
    {
        i = i / 2;
    }
    else
    {
        i = i / 2 + 1;
    }
    ptr = start;
    for (int j = 1; j < i; j++)
    {
        ptr = ptr->next;
    }
}
```

Using 2 loops

$O(n)$

Using 1 loop

struct node*

find(node *start)

```
{
    node * ptr = start, * ptrmid = start;
```

```
    while (ptr->next != NULL && (ptr->next->next != NULL))
```

```
    {
        ptrmid = ptrmid->next;
```

```
        ptr = (ptr->next->next);
```

```
    }
    return (ptrmid);
}
```

$O(n)$

$\frac{n}{2}$ times

* Binary Search can be applied on Linked List but it is not efficient because it will take $O(n)$ times & $\log_2 n$ extra space.

Q. Write a C program to perform Binary Search on LL.

```
node * BS(node * start, int x)
{
    node * ptr = start;
    node * ptrmid = start; node * m;
    while (ptrmid->data
    if (start->next == NULL)
        if (start->data == x)
            return (start);
        else
            return (NULL);
    else
    {
        m = findmid(start);  $\rightarrow O(n)$ 
        if (m->data == x)
        {
            return (m);  $\rightarrow O(1)$ 
        }
        else if (m->data < x)  $\rightarrow O(1)$ 
        {
            m->next = NULL;
            BS(start);
        }
        else
        {
            start = m->next;
            BS(start);
        }
    }
}
```

Let $T(n)$ be the time complexity of Binary Search algo on a linked list which contains n elements, then

$T(n) = O(1)$, if $n=1$

$n + T\left(\frac{n}{2}\right)$, if $n > 1$

$\downarrow$
 $[O(n)]$

$\left[\begin{array}{l} \log_2 n \\ \text{stack space} \\ \text{is req} \end{array} \right]$

$n^{\log_2 2} = n^{\log_2 1} = n^0 = 1$

Q Write a C program to sort the given linked list.

BubbleSort (node *start)

```
{ node *ptr = start;  
  node *prev = start;
```

```
  for (int i
```

```
    int i = 1;
```

```
    while (ptr->next != NULL)
```

```
      { ptr = ptr->next;
```

```
        i++;
```

// length of linked list

```
      { ptr = start->next;
```

```
        for (int j = 0; j < i; j++)
```

```
          { while (ptr->next != NULL)
```

```
            { if (prev->data > ptr->data)
```

```
              { prev->temp = prev->data;
```

```
                prev->data = ptr->data;
```

```
                ptr->data = temp;
```

```
              }
```

```
            prev = ptr;
```

```
            ptr = ptr->next;
```

```
          } ptr = start;
```

```
        }
```

```
      }
```

```
    }
```

Sort (node *start)

```
{ node * ptr = start;
```

```
  int i = 1;
```

```
  while (ptr->next != NULL)
```

```
    { ptr = ptr->next;
```

```
      i++;
```

```
    }
```

```
    ptr = start;
```

```
    for (int j = 0; j < i; j++)
```

```
      {
```

```
while (ptr->next != NULL)
```

Or

```
Sort (struct node * S)
{ struct node * p, * q;
  for (p = S; p != NULL; p = p->next)
    { if (p->data
      for (q = p->next; q != NULL; q = q->next)
        { if (p->data > q->data)
          { swap(p->data, q->data);
            }
          }
    }
}
```

:- $O(n^2)$

Q. Write a C-program to reverse the given LL.

```
Reverse (node * start)
{ node * ptr = start, node * temp1, node * temp2;
  int i = 1; int temp1 = start;
  while (ptr != NULL) temp2 = start->next;
  while (ptr != NULL)
    { ptr = ptr->next;
      temp2 = (ptr->next)->next;
      ptr->next = temp2;
      (ptr->next)->next = temp1;
      temp1 = ptr;
      temp2 = temp2->next;
    }
}
```

~~node~~

struct node1

```
{ struct node * addr; struct node * next;
};
```

Reverse (struct node * start)

```
{ node * ptr = start; node1 * ptr1;
```

```
while (ptr != NULL)
```

```
{ ptr1 = (node1 *) malloc(sizeof(node1));
```

```
ptr1->addr = ptr;
```

```
ptr = ptr->next;
```

```
}
```

```
ptr = start;
```

Or

Reverse (struct node * s)

```
{ struct node * p, * q, * r;
```

```
r = s; q = p = NULL;
```

```
while (r != NULL)
```

```
{ p = q;
```

```
q = r;
```

```
r = r->next;
```

```
q->next = p;
```

```
{ s = q;
```

```
}
```

→ 3 pointers compulsory.

$O(n)$

- Q The following C func. takes a single LL as i/p argument. It modified the list by moving the last node to the front of the list & returns the list. Some part of the code is left blank.

```

typedef struct node {
    int value;
    struct node *next;
} node;

```

```

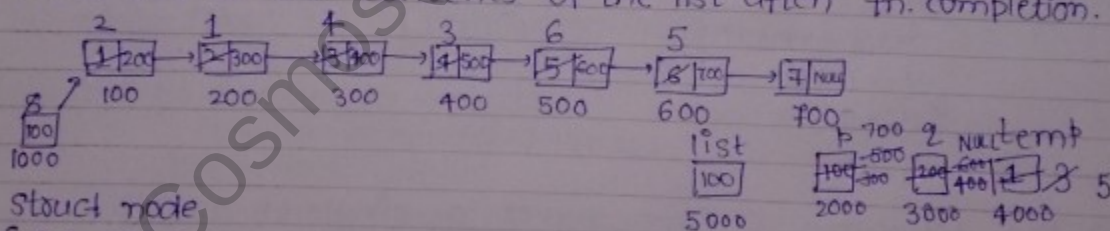
node* modifiedlist(node* head)
{
    node *p, *q;
    if ((head == NULL) || (head->next == NULL))
        return (head);
    q = NULL;
    p = head;
    while (p->next != NULL)
    {
        q = p;
        p = p->next;
    }
    p->next = head;
    q->next = NULL;
    head = p;
    return (head);
}

```

→ answer

Q. The following C fn. takes a singly linked list of integers as parameter & rearranges the elements of list. The fn. is called with the list containing the integers:-
1, 2, 3, 4, 5, 6, 7

What will be the contents of the list after fn. completion.



```

struct node
{
    int value;
    struct node *next;
}

```

```
void arrange(struct node *list)
```

```
{node *p, *q;
```

```
int temp;
```

```
if (!list || !list->next)
```

```
return;
```

```
p = list;
```

```
q = list->next;
```

```
while(q)
```

```
{temp = p->value; p->value = q->value; q->value = temp;
```

```
p = q->next;
```

```
q = p->next;
```

```
}
```

```
}
```

list now :- 2, 1, 4, 3, 6, 5, 7

Q. Write a C program to find a cycle (loop) in the given linked list.

```
struct node
```

```
{node * addr;
```

```
node * next;
```

```
};
```

```
findcycle (node * start)
```

```
{node1 * start1 = (node1 *) malloc (sizeof (node1));
```

```
start1->addr = start;
```

```
start1->next = NULL;
```

```
node1 * prev = start1;
```

```
node * ptr = start;
```

```
while (ptr != NULL)
```

```
{node1 * ptr1 = (node1 *) malloc (sizeof (node1));
```

```
prev->next = ptr1;
```

```
prev = ptr1;
```

```
ptr1->addr = ptr; ptr = ptr->next;
```

```
ptr1->next = ptr;
```

Or

Algorithm

- ① Take two pointers p_1 & p_2 .
- ② increment p_1 by 1 & p_2 by 2.
- ③ Max. n loops p_1 & p_2 are same comes out to be same. When p_1 & p_2 comes out to be equal, then there's a cycle.

Algorithm:-

Store every address in Array if that is not found in array continue until match is found.

- * If we allocate memory using Malloc, then elements inside struct, values have garbage value.
- * If we allocate memory using Calloc, then elements inside struct will be initialized to 0.

Q Write a C program to implement Stack using LL.

Q Write a C " " " " Queue " "

Q " " " " " " LL " arrays

Q " " " " " " perform following ops on LL:-

(i) Union of 2 LL.

(ii) Intersection of 2 LL.

(iii) Polynomial addition of 2 LL.

(iv) " multiplication of 2 LL.

Date

18.08.12

Stacks

```
struct stack { int data;  
                struct stack *next;  
}; struct stack *top = NULL;  
Push(struct stack *top, int x)  
{  
    struct stack *p = (struct stack *) malloc (sizeof (stack));  
    if (p == NULL)  
        { printf("Memory is full");  
          exit(1);  
        }  
    else  
        { p->data = x;  
          p->next = top;  
          top = p;  
        }  
}
```

→ [OCI]

```
Pop(struct stack *top)  
{ struct stack *temp = top;  
  if (top == NULL)  
      { printf("Stack is empty");  
        exit(1);  
      }  
  top = temp->next;  
  x = temp->data;  
  free(temp);  
  temp = NULL;  
}
```

→ [OCI]

Queue

* If To implement Queue property, we need atleast 2 pointers (Front & Rear).

Add

```
struct queue {  
    int data;  
    queue * next;  
};
```

```
struct queue *front = NULL, *rear = NULL;
```

Add(int x)

```
{ queue *temp = (queue *) malloc (size of (queue));
```

```
temp->data = x;
```

```
if (front == NULL && rear == NULL)
```

```
{ front = temp;
```

```
rear = front;
```

```
front->next = NULL;
```

→ [OCI]

```
}
```

```
else if (front != rear)
```

```
{ temp->next = front; rear->next = temp;
```

```
rear = temp;
```

```
}
```

```
}
```

Del(int x)

```
{ if queue *temp = front;
```

```
if (front == rear == NULL)
```

```
{ printf("Queue is empty");
```

```
exit(1);
```

```
// no element
```

```
}
```

```
if (front == rear)
```

→ [OCI]

```
{ front = front->next;
```

```
rear = NULL;
```

```
free(temp);
```

```
temp = NULL;
```

// only one-element

```
}
```

```
else
```

```
{ front = front->next;
```

```
free(temp);
```

```
temp = NULL;
```

```
}
```

```
}
```

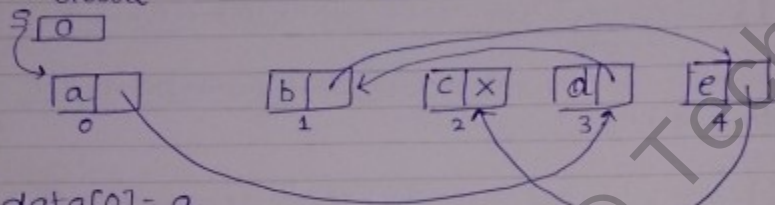
Implementing Linked List Using Arrays

0	a
1	b
2	c
3	d
4	e

data

0	3
1	4
2	-1
3	1
4	2

link



data[0] = a
link[0] = 3

Q. Write a C program to display data part of the linked list using array representation.

```
display(int *d, int *l, int *start)
{
    int *temp = start;
    while (*temp != -1)
    {
        printf("%d", data[temp]);
    }
}
```

```
display(data, link, s)
{
    int temp = s;
    while (temp != -1)
    {
        printf("%d", data[temp]);
        temp = link[temp];
    }
}
```

Q. Write a C program to delete a ^{last} node from linked list

del(data, link, s)

{ temp = s;

prev = s;

while (temp != 0 & link[temp] != -1)

{ prev = temp;

temp = link[temp];

}

link[prev] = -1;

}

or

deletelast(data, link, s)

{ temp = prev = s;

if (link[s] == -1)

{ s = -1;

exit(1);

}

else

{ while (link[temp] != -1)

{ prev = temp;

temp = link[temp];

}

link[prev] = -1;

}

}

Union of 2 Linked List:-

P → 10 → 20 → 30 → 40 → NULL

$O(n^2)$

Q → 15 → 20 → 25 → 30 → NULL

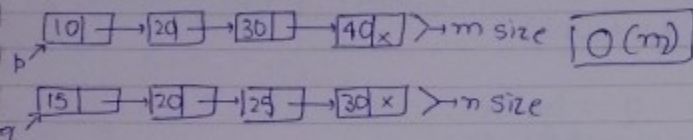
10 → 20 → 30 → 40 → 15 → 25 → NULL

10/8

Algo:-

1. Add first linked list to the result. $\rightarrow O(n)$
2. Take every element of 2nd LL & check if it is available in first linked list or not using Linear Search.
 - if it is available, don't add it in LL. $\rightarrow O(n^2)$
 - otherwise add it to LL.

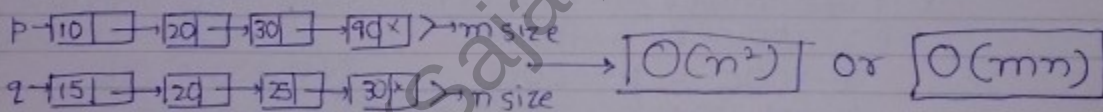
Concatenation of 2 LL



Algo:-

1. Go to the last node of the 1st LL. $\rightarrow O(n)$
2. & link that node with the starting node of the 2nd LL. $O(1)$.

Intersection of 2 LL:-



1. Take the 1st node of 1st Linked List & check whether the node's data (of 1st LL) matches with any node's data of 2nd Linked List.
 - if it matches with a then add it to result LL.
 - otherwise don't add.
2. Repeat Step 1 with all the other nodes of 1st LL.

Polynomial Addition

p: $10x^3 + 5x^2 + 2x + 5$

q: $20x^3 + 15x^2 + 5x + 20$

$30x^3 + 20x^2 + 7x + 25$

p: $10x^{15} + 20x^5 + 3x^2 + 16$

q: $5x^5 + 7(x + 10)$

$10x^{15} + 20x^5 + 25x^5 + 3x^2 + 7x + 26$

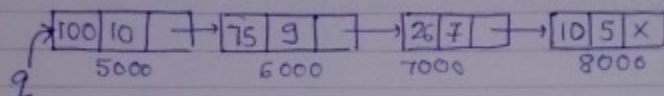
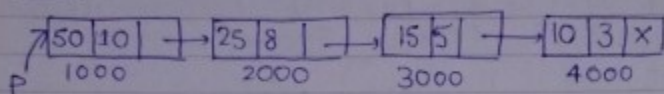
```

struct poly {
    int power;
    int coe;
    poly * next;
};

```

Polynomial addition (

Time Complexity:



while (p != NULL && q != NULL)

```

{
    if (p->power == q->power)
    {
        r->power = p->power;
        r->coe = p->coe + q->coe;
        p = p->next;
        q = q->next;
        r = r->next;
    }
}

```

else

```

{
    if (p->power > q->power)
    {
        r->power = p->power;
        r->coe = p->coe;
        r = r->next;
        p = p->next;
    }
}

```

else

```

{
    r->power = q->power;
    r->coe = q->coe;
    r = r->next;
    q = q->next;
}

```

if (p == NULL && q == NULL)
return (r);

else

```

{
    if (p == NULL)
        add q to r;
    else
        add p to r;
}

```

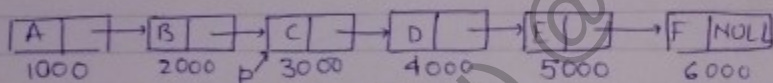
}

Polynomial Multiplication :-

```
while (p != NULL)
{
    while (q != NULL)
    {
        r->power = p->power + q->power;
        r->coe = p->coe * q->coe;
        r = r->next;
        q = q->next;
    }
    p = p->next;
    q = s; // starting of q
}
```

→ $O(n^2)$

Q. Consider the following LL:-



from the above linked list delete a node pointed by p if only p is given. (starting is not given.)

Ans.

```
del (node * p)
{
    node * temp = start;
    node * prev = start;
    while (temp != p)
    {
```

copy the < content

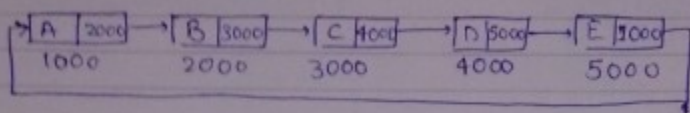
```
p->data = (p->next)->data; // but if p is last node,
p->next = (p->next)->next; // then this algo fails
free (p->next);
```

p

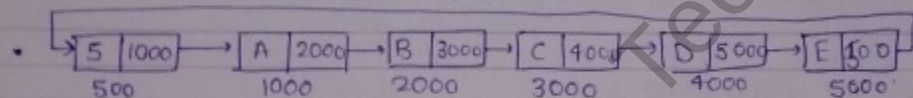
Drawbacks of Singly LL :-

- ① We can move only in one direction. (We can't go back)
- ② In singly LL last node's next part is always NULL w/o any use (we can keep starting node address, so that we can go back.)

Circular SLL :-



1. We can go back.
2. The drawback with circular SLL it may enter into the infinite loop.
3. In order to eliminate the infinite loop in circular SLL, we are going to head node.
4. Head node data part will contain valuable info. like total no. of nodes in circular SLL.

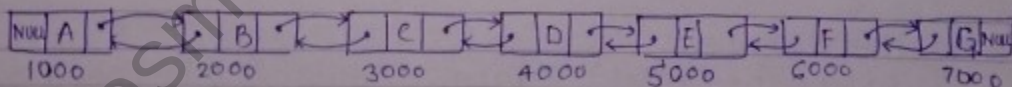


if

Doubly Linked List

struct DLL

```
{ struct DLL *left;  
  int data;  
  struct DLL *right;  
};
```



Q. Write a C program to delete a node which contain data as x in the given double linked list.

Ans. delC(int x)

```
{ struct DLL *ptr = start;  
  if (start->data ==  $x$ )  
  { start->right = start->right;  
    start->left = NULL;  
    free(ptr);  
    ptr = NULL;  
  }
```

```

* if (p != NULL)
{
    exit(1);
}

```

```

else
{
    while (ptr->data != x && ptr != NULL)
    {
        ptr = ptr->right;
        *if (ptr->right == NULL) { (ptr->left)->right = ptr->right;
        free(ptr); ptr = NULL; }
    }
    else { (ptr->left)->right = ptr->right;
    (ptr->right)->left = ptr->left;
    free(ptr);
    ptr = NULL;
}
}
}

```

Q Write a C program to insert a node with data x before a node with data y.

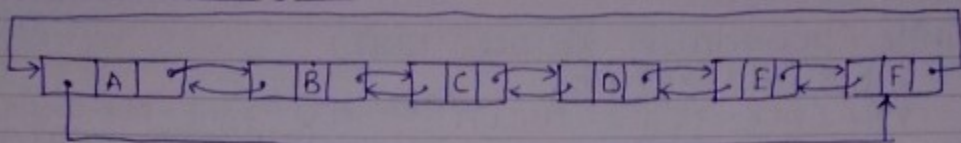
Ans. add (int y, dll * start)

```

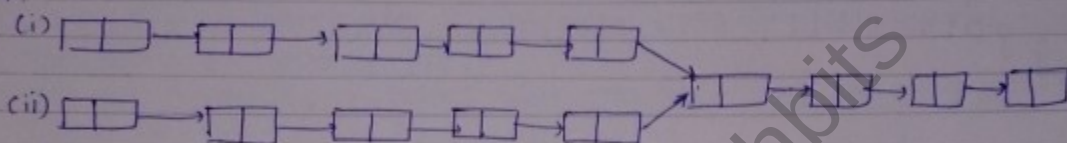
{
    dll * ptr = start;
    dll * temp = (dll *) malloc (sizeof (dll));
    while (ptr->data != y && ptr != NULL)
    {
        ptr = ptr->right;
        if (ptr == NULL) { exit(1); }
    }
    if (ptr == start)
    {
        temp->data = x;
        temp->left = NULL;
        start->left = temp;
        temp->right = temp->start;
        start = temp;
    }
    else if (start == NULL)
    {
        temp->data = x;
        exit(1);
    }
    else
    {
        temp->data = x;
        temp->left = ptr->left;
        temp->right = ptr;
        (ptr->left)->right = temp;
        ptr->left = temp;
    }
}
}

```

Circular Doubly LL



Q



mark all nodes with flag 0, & make the flag as 1 as the node is visited (do this for (i)st LL & then for (ii)nd & check if the flag is 0 or 1, if it is already 1, from that node onwards, count the no. of nodes till NULL.) $\rightarrow O(n)$
2n

★ Application of Doubly LL:- Binary Tree

Binary Tree

Q Write a C program to find no. of leaf nodes in the given binary tree.

Ans.

```
int totleafnodes (tree * p)
{
    if (p->left == NULL & p->right == NULL)
        return 1;
    else
    {
        totleafnodes (p->left);
        totleafnodes (p->right);
    }
}

void main()
{
    tree * start; // assume it is assigned start root
    int i = 0;
    if (start != NULL) // address
        totleafnodes (start);
}
```

```
struct tree
{
    tree * lc;
    int data;
    tree * rc;
};
```

or

NLN(struct tree *s)

```
{ if (s == NULL) return 0;
  else { if (s->lc == NULL && s->rc == NULL)
```

```
    return(1);
```

```
    else {
```

```
        xL = NLN(s->lc); → T(n/2)
```

```
        xR = NLN(s->rc); → T(n/2)
```

```
        return(xL + xR); → c
```

```
    }
}
```

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c$$

$$= 2T\left(\frac{n}{2}\right) + c$$

$$= \boxed{O(n)}$$

$$\text{or } T(n) = T(n) + T(n - (n+1)) + c$$

$$\text{if } n = 0$$

$$T(n) = \boxed{O(n^2)}$$

Q. Write a C program to find no. of non-leaf nodes in a given binary tree.

```
int i=0;
```

```
totnonleafnodes(struct tree *b)
```

```
{ * if (b->left != NULL || b->right != NULL)
```

```
    { i++;
```

```
    return;
```

```
    }
```

```
or else
```

```
    { totnonleafnodes(b->left);
```

```
      totnonleafnodes(b->right);
```

```
    }
```

```
}
```

```
void main()
```

```
{ if (root != NULL)
```

```
    { totnonleafnodes(root);
```

```
    }
```

```
}
```

or

```
NLN(struct tree *s)
```

```
{ if (s == NULL) return 0;
```

```
  else
```

```
    { if (s->lc == NULL && s->rc == NULL)
```

```
        return(0);
```

```

else
{
    xL = NLN(S->lc);
    xR = NLN(S->rc);
    return(1+xL+xR);
}
}
}

```

Q Write a C program to find total no. of nodes in the given binary tree.

```
int i=0;
```

```
totnodes(tree *p)
```

```
{
    if (p->left == NULL & p->right == NULL)
```

```
{
    if (p == NULL)
        return;
}
```

```
{
    i++;
```

```
    return;
}
```

```
else
```

```
{
    i++;
```

```
    totnodes(p->left);
```

```
    totnodes(p->right);
}
```

```
}
```

Or

```
NLN(tree *s)
```

```
{
    if (s == NULL) return 0;
}
```

```
else
```

```
{
    if (s->lc == NULL & s->rc == NULL)
```

```
        return (1);
}
```

```
else
```

```
{
    xL = NLN(s->lc);
```

```
    xR = NLN(s->rc);
```

```
    return(1+xL+xR);
}
```

```
}
```

```
}
```

```
}
```

Q Write a C program to find height of a given binary tree.

Ans height (tree * s)

```
{ if (s == NULL) return 0;
```

```
else
```

```
{ if (s->lc == NULL && s->rc == NULL)
    return 0;
```

```
else
```

```
{ xL = height(s->lc);
```

```
  xR = height(s->rc);
```

```
  return (1 + max(xL, xR));
```

```
}
```

```
}
```

or
height (tree * p)
{ n = no. of nodes

Q. Write a C program to check given binary tree is strict binary tree or not.

```
bool flag = true;
```

```
strictOrNot (tree * p)
```

```
{ if (p->left == NULL && p->right == NULL)
```

```
    { return;
```

```
  }
```

```
else if ((p->left == NULL && p->right != NULL) ||
```

```
        (p->left != NULL && p->right == NULL))
```

```
    { flag = false;
```

```
      return;
```

```
    }
```

```
else
```

```
    { strictOrNot(p->left);
```

```
      strictOrNot(p->right);
```

```
    }
```

```
}
```

Or

```
strictBT (tree * s)
```

```
{ if (s == NULL) return 1;
```

```
else
```

```
{ if (s->lc == NULL && s->rc == NULL)
```

```
    return 1;
```

```
else
```

```

if (s->lc != NULL && s->rc != NULL)
    return (strictBT(s->lc) && strictBT(s->rc));
else
    return 0;
}
}

```

Q Consider the following C program:-

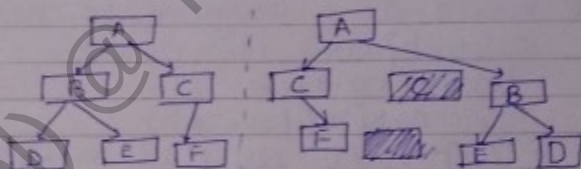
Tree (struct Bnode *s)

```

{ if (s != NULL)
    { t = (struct Bnode *) malloc (sizeof (struct Bnode));
      t->data = s->data;
      t->lc = s->rc; tree (s->rc);
      t->rc = s->lc; tree (s->lc);
      return (t);
    }
}

```

Mirror Image



Q. Write a C program to find min. element in the given binary search tree

Ans. int min = root->data;

findmin (tree *p)

```

{ if (p->left != NULL) if (p->right != NULL)

```

```

    return (p->data);

```

```

    if (p->data <

```

```

        findmin (p->left);
    }
}

```

or

find_min (tree *s)

```

{ if (s == NULL) return;

```

```

  else

```

```

    { while (s->lc != NULL)

```

```

        s = s->lc;

```

```

    } return (s->data);
}
}

```

Q. A data structure comprises of nodes, each of which has exactly 2 pointers to other nodes with no Null pointers, the following C program is to be used to count the no. of nodes in given binary tree. It uses a mark field assumed to be '0' initially for all nodes. Find the missing statement in the code :-

→ struct test

```
{ int info, mark;
```

```
  struct test *p, *q;
```

```
};
```

```
int nodecount(struct test *a)
```

```
{ if (a->mark)
```

```
  return (0);
```

```
  else
```

```
  { a->mark
```

```
    return (nodecount(a->p) + nodecount(a->q) + 1);
```

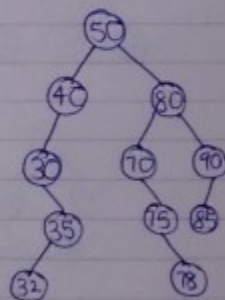
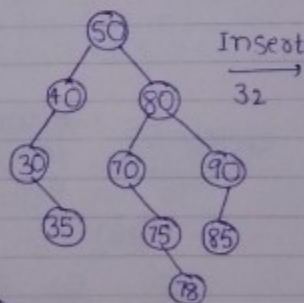
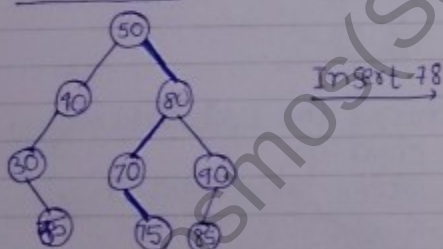
```
  }
```

```
  → a->mark = 1; (my answer)
```

Search

Binary Tree

Insertion



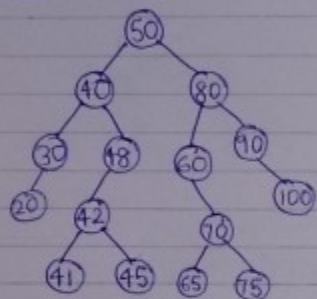
Finding the position to insert :-

- Best Case :- $O(1)$ (root itself)
- Worst Case :- $O(n)$
- Avg Case :- $O(\lg n)$

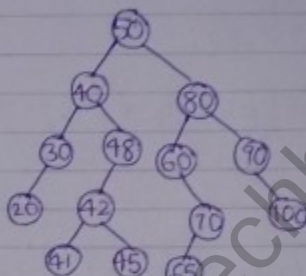
Algo

- ① Find the place of $x \rightarrow O(n)$
- ② Insert element $x \rightarrow \frac{O(1)}{O(n)}$

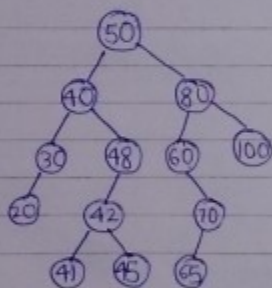
Deletion



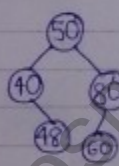
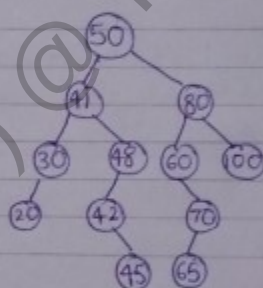
Delete 75
→



Delete 100
→

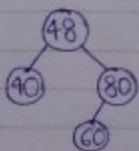


Delete
40
→

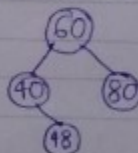


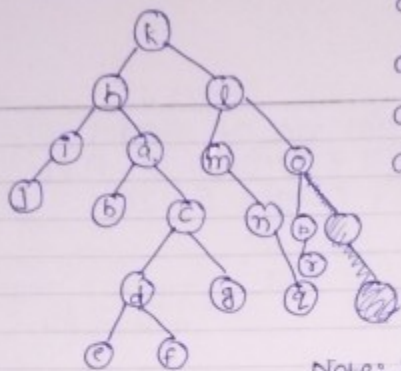
Delete 50
→

either choose the Left Subtree
largest child or
Right Subtree smallest child.



or





delete(a) = inorder predecessor or
inorder successor $\rightarrow b(x)$

$$\text{delete}(c) = f(0x)g$$
$$\text{delete}(d) = e \text{ (or) } f$$

delete(0) = connect l & p

delete(q) = no-need to do anything

$$n + O(1) = \boxed{O(n)}$$

finding place for adjustment

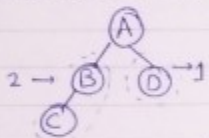
Note: If we delete any node from BST replacement will be done by inorder successor or inorder predecessor.

AVL Tree

(Adelson, Valski, Landies)

Or Balanced Binary Search tree [Size $\rightarrow O(\log n)$]

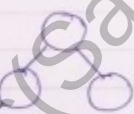
Balanced Factor = $H(LST) - H(RST)$ or $H(RST) - H(LST)$



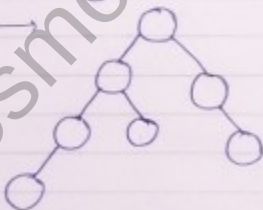
Balanced Factor (of A) = $2 - 1 = 1$

Note:- A BST in which every node's Balance factor is $-1, 0$ or (max. diff.) is called AVL tree.

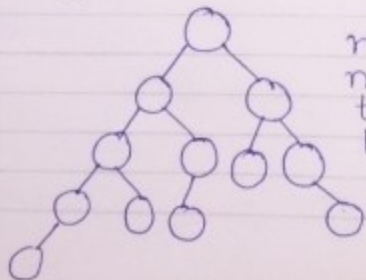
Height = 2 $\rightarrow$



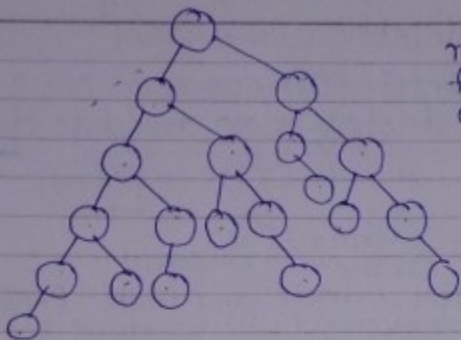
Height = 3 $\rightarrow$



Height = 4



height = 5



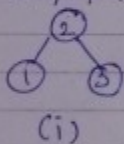
min. no. of nodes
for a tree to be AVL
of height 5 = 20

Min. no. of nodes in a height H AVL tree: $M(H)$

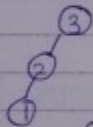
$$MNN(H) = MNN(H-1) + MNN(H-2) + 1$$

$$\frac{MNN(H-1) + MNN(H-2) + 1}{2}$$

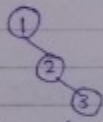
Q. n = 3(1, 2, 3) $\rightarrow$ BST



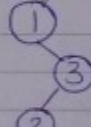
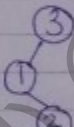
(c)



Cris



(iii)



②

AVL trees - (i)

BST: - (i), (ii), (iii), (iv), (v)

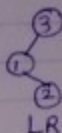
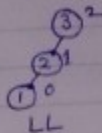
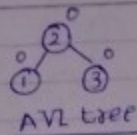
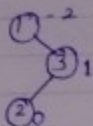
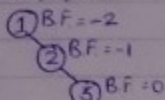
(ii), (iii), (iv), (v) are converted to (i) using rotations, & this rotation should take constant time.

Date

21.08.12

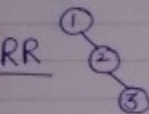
AVL Trees

$n=3 \Rightarrow (1,2,3)$

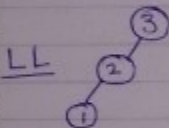
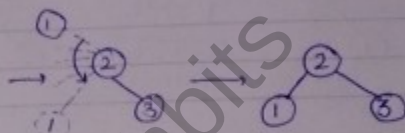


In this case ① has problem from R (because of ②) & ② has problem from its right, $\therefore$ RR problem.

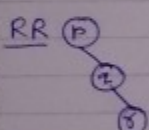
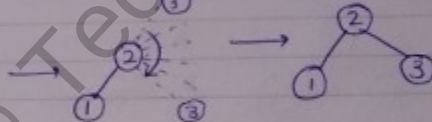
Rotations :-



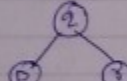
1. Keep the nodes as it is which has B.F. = 0, 1, -1 & rotate the node which has B.F. $\neq$ 0, 1, -1 to the anticlockwise (one left rotation)



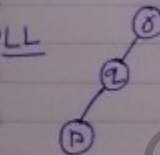
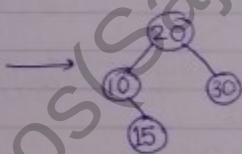
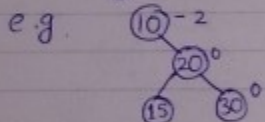
1. Keep the nodes as it is which has B.F. = 0, 1, -1 & rotate the node which has B.F. $\neq$ 0, 1, -1 clockwise (one right rotation)



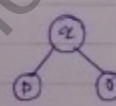
RR



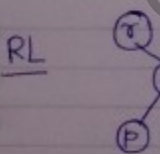
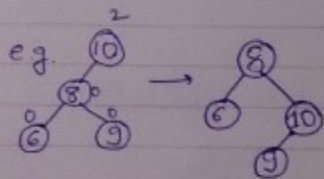
$q \rightarrow lc = p$
 $p \rightarrow rc = NULL$



LL

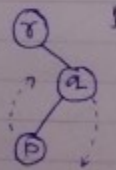


$q \rightarrow rc = r$
 $r \rightarrow lc = NULL$

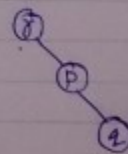


RL

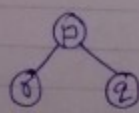
→



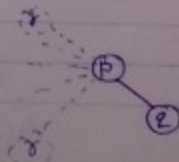
1. do one clockwise rotation



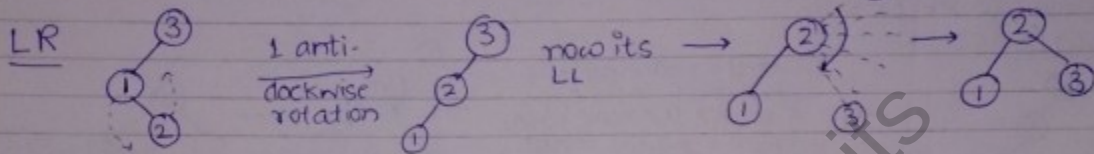
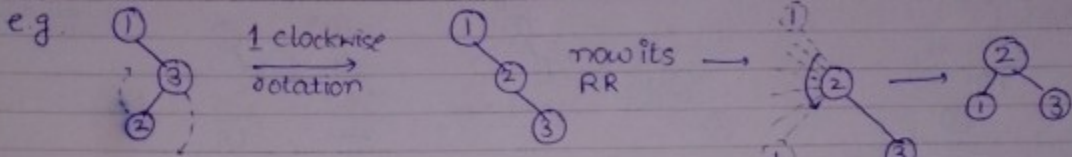
now this has RR problem.



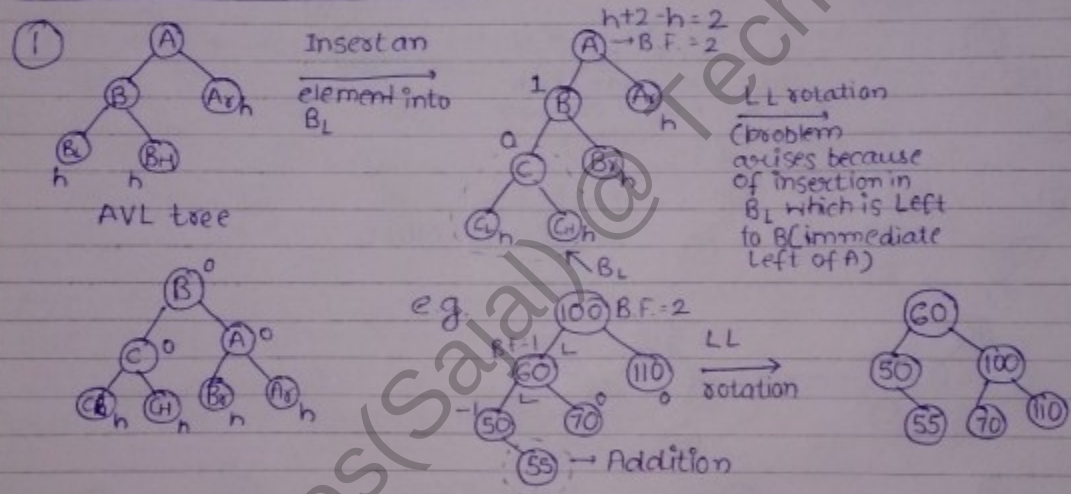
←



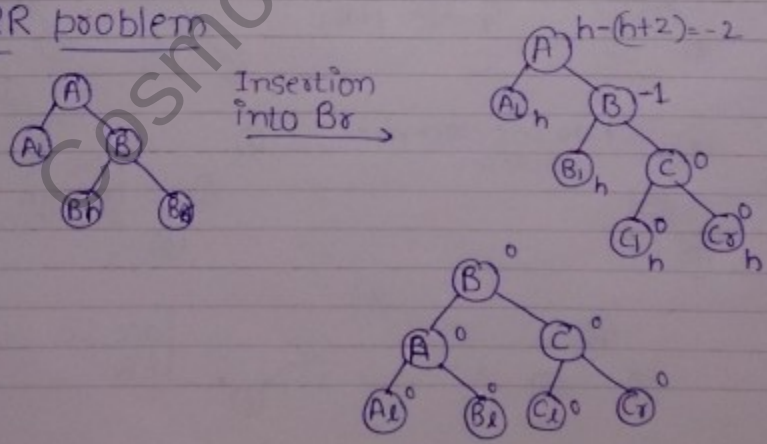
acc. to RR

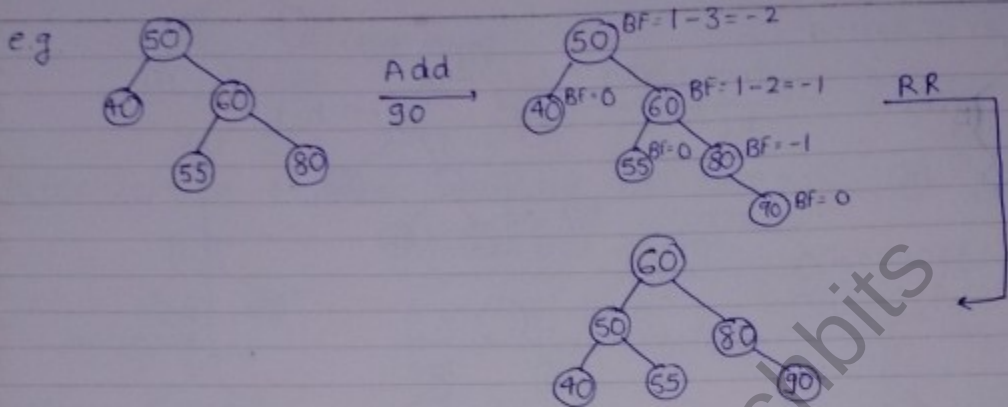


Insertion into AVL tree:-

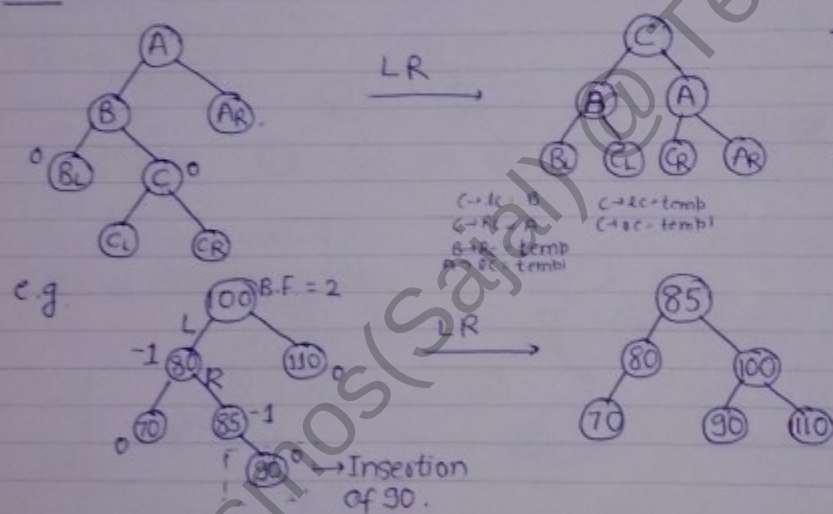


② RR problem



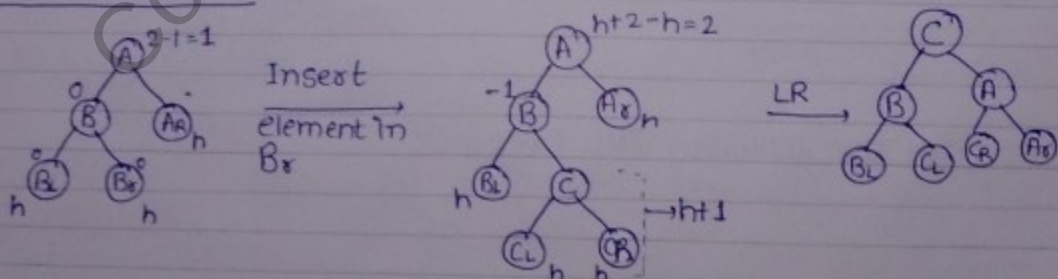


LR

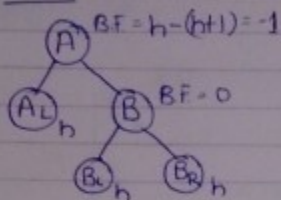


★ In LR problem 2 rotations are compulsory.
 ★ After doing one rotation we will get LL.
 ★ After getting LL problem, apply LL rotation.

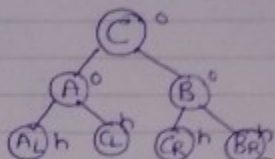
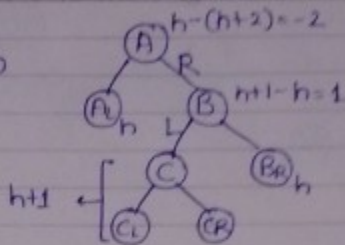
Sis's explanation:-



RL



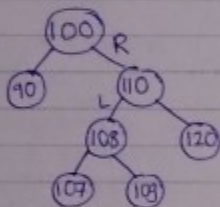
Insert into
 B_L



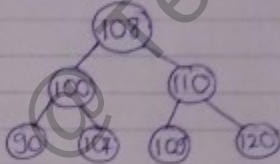
RL rotation

★ In RL rotation
2 reverse rotations
are compulsory
★ After 1 rotation
we will get RR
problem.
★ Apply 1 more
RR rotation to
get AVL tree.

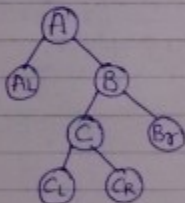
e.g.



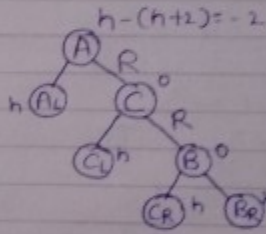
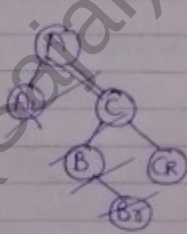
RL
problem



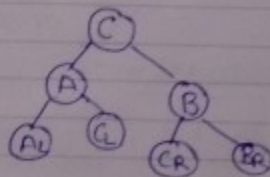
OR



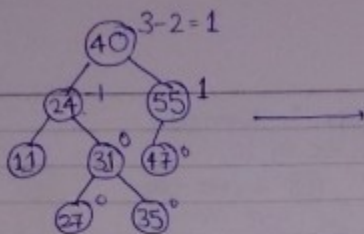
→



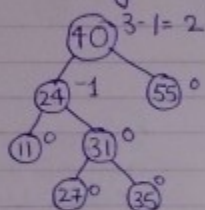
RR



Q. Consider the following AVL tree:-

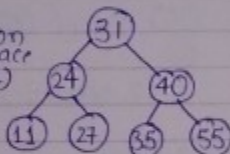


After deleting 47, show resultant AVL tree →



★ To delete an element,
search that element,
searching complexity :- $O(n)$ [BST] & not AVL
& then perform
the deletion algorithm. $O(\log n)$ [AVL & BST]

LR (Deletion takes place from RL)
Left rotation



Struct AVL

```
{ int data;  
  int height;  
  struct AVL *lc, *rc;  
};
```

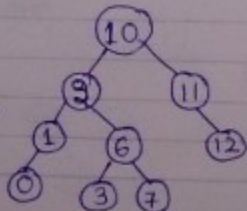
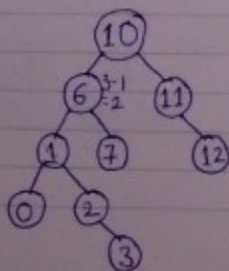
Complexity :-

- ① Finding place of deletion → $\log n$
- ② Deletion of that element → Constant
- ③ Traversing the complete path from site of deletion to the root to check the balancing factors = $\log n$

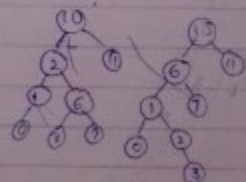
$$2 \log n = O(\log n)$$

Q. What will be the resulting AVL tree if we insert following elements into AVL tree:-
10, 6, 11, 12, 1, 7, 0, 2, 3

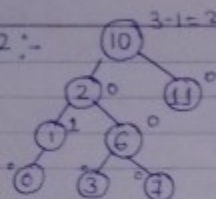
Addition → $O(\log n)$



AVL Tree



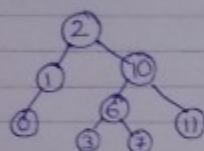
delete 12 :-



Now problem is because of Left subtree of 10, $\therefore$ L, now, we are at 2, we can 2 has B.F. = 0, $\therefore$ we can go both left & right, going left results in LL
" right " " LR

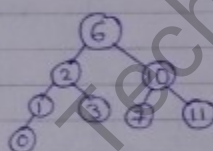
What will be the resulting AVL tree if we delete 12.

LL $\rightarrow$



1 rotation

LR $\rightarrow$

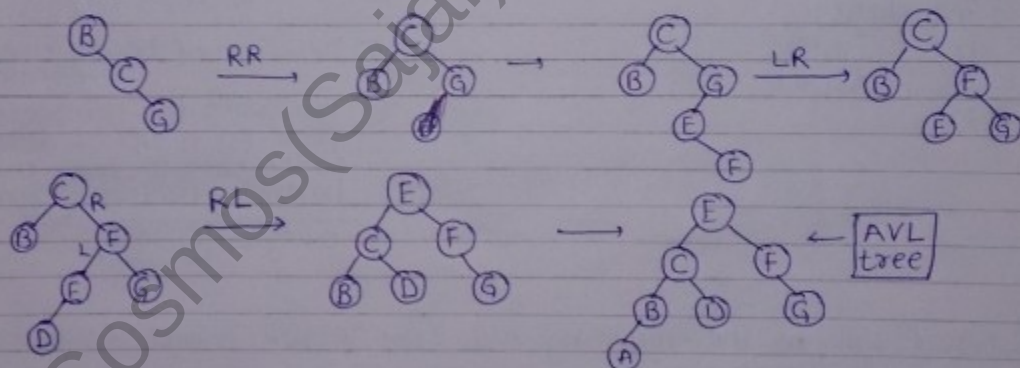


2 rotations

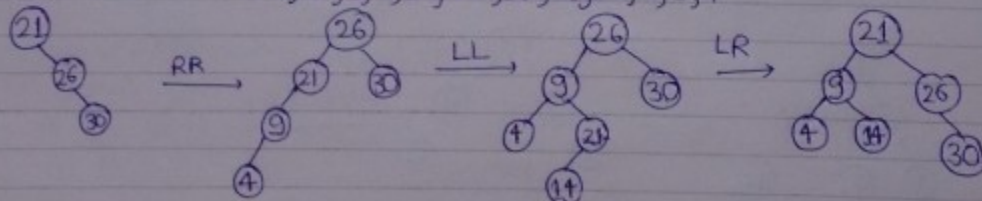
We could apply any of these. (either LL or LR.)

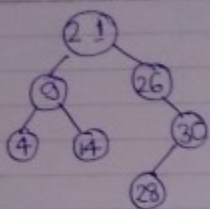
Q. Construct AVL trees from following keys.

B, C, G, E, F, D, A

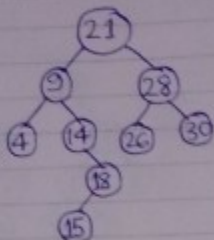


Q Create AVL:- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

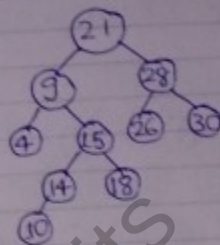




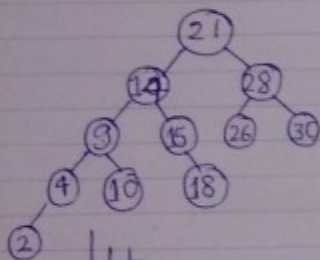
RL



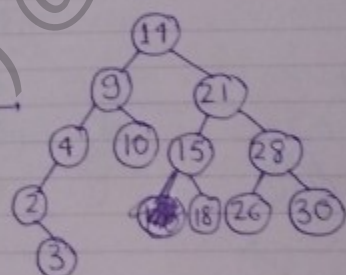
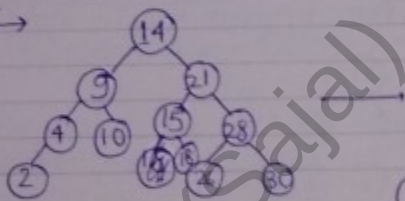
RL



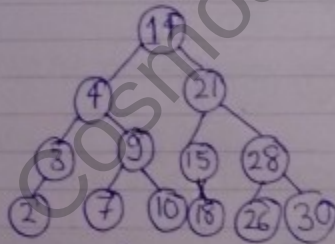
RL



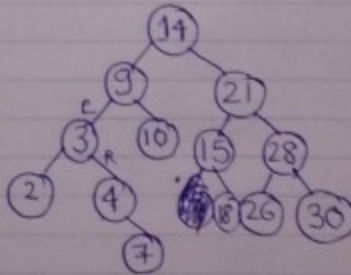
LL



LR

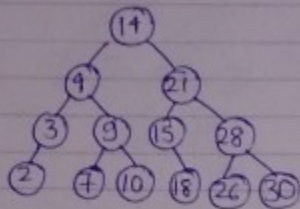


LR

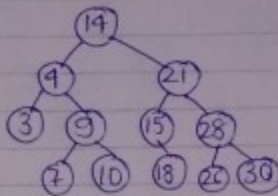


AVL Tree

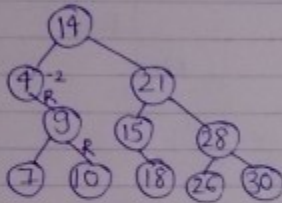
Q. Delete 2, 3, 10, 18, 4, 9, 14, 7, 15



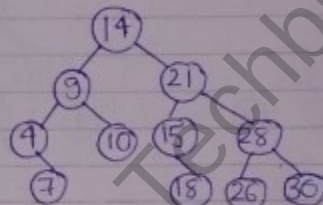
2



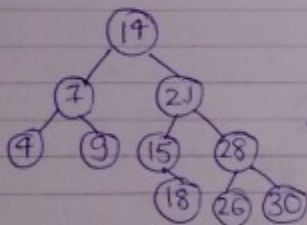
3



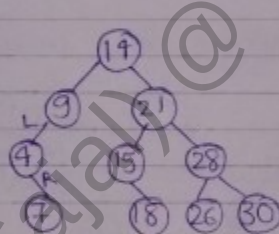
→



10

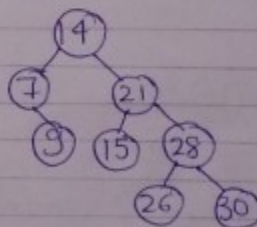
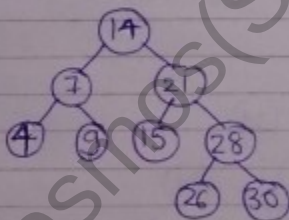


←

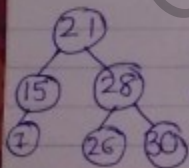


18

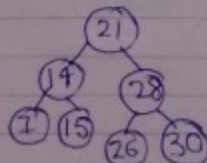
4



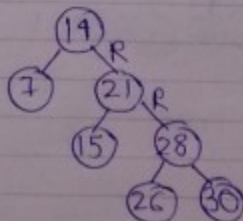
9

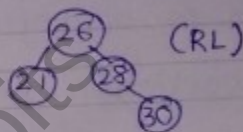
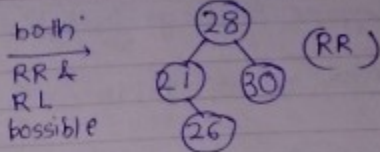
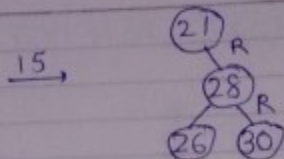
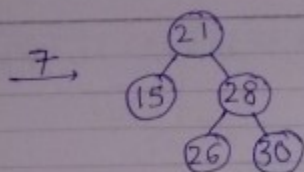


14



RR





Note:-

Algorithm for insertion

1. Find the place of the element → $O(\log n)$.
2. Insert that element → $O(1)$.
3. Traverse the same path to find B.F. → $O(\log n)$
4. If not balanced at some node, perform rotation → $O(1)$

Total Complexity: $O(\log n)$

Algorithm for deletion

1. Find the place of the element to be deleted → $O(\log n)$
2. Deletion of element → $O(\log n)$ [when there are 2 children of a node to be deleted.]
→ $O(1)$ [when children are 0 or 1.]
3. Check the balancing factors → $O(\log n)$ by traversing
4. If not balanced at some node, perform rotation → $O(1)$

Total Complexity: $O(\log n)$

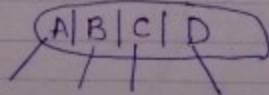
Multiway-Search Tree

1. Search Tree (Ordered Tree)
2. At every node minimum 2 children allowed (order of multiway search tree (m)).
3. Height of Multiway search tree is $O(\log_m n)$. [Balanced M-way search tree.]
4. Deletion & Insertion will take $O(\log_m n)$.

B-Tree:-

Order of B-Tree = m (assume) [means each node can have max. 'm' children.]

∴ maximum (m-1) keys
maximum m-children



min. $\lceil \frac{m}{2} \rceil$ children

min. $\lceil \frac{m}{2} \rceil - 1$ keys.

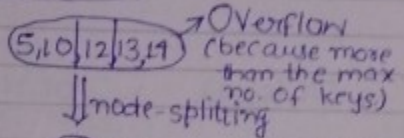
Degree = 5 | Keys: 5, 10, 12, 13, 14, 1, 2, 3, 4, 20, 18, 19, 17, 16, 15, 25, 23, 24, 22, 21, 30, 28, 29

max. no. of children = 5
 " " " keys = 4

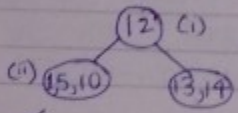
min no. of children = $\lceil \frac{5}{2} \rceil = 3$
 min. no. of keys = $\lceil \frac{5}{2} \rceil - 1 = 3 - 1 = 2$

Other than root node
 A min. condⁿ has to be satisfied by the nodes other than root

Insertion in B-Trees

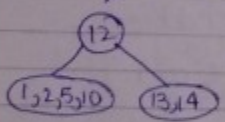


mode-splitting

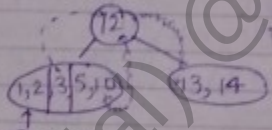


(i) $1 < 12$, therefore go left
 (ii) $1 < 13$, $\therefore$ left to 5.

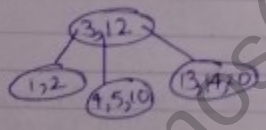
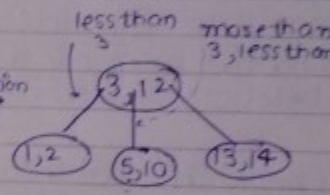
Insert 2



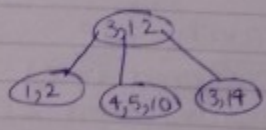
Insert 3



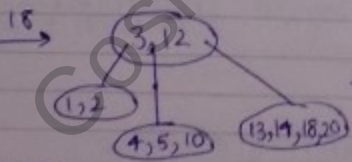
key redistribution



20

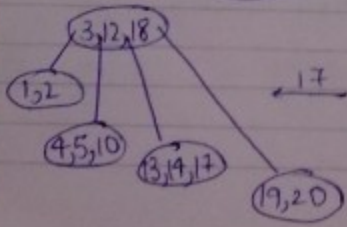
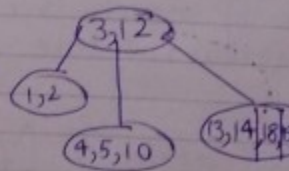


4

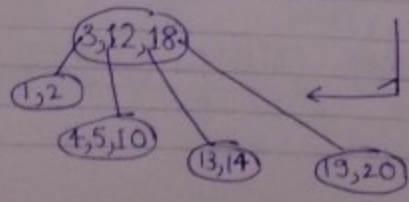


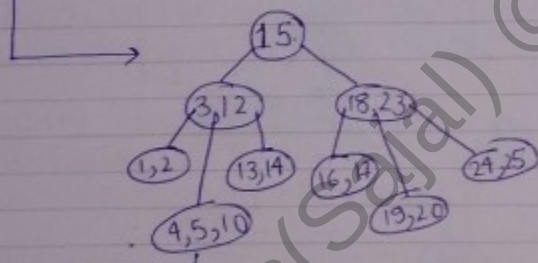
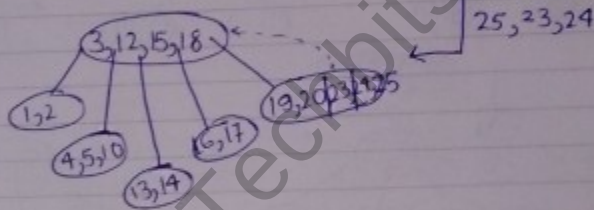
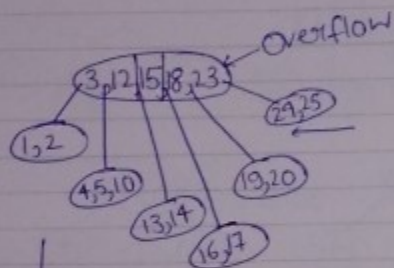
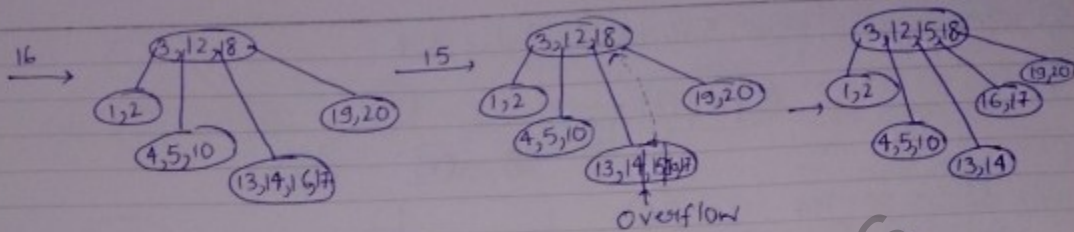
18

19



17



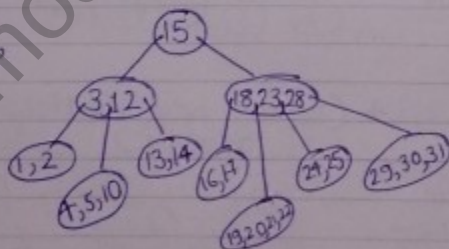


time complexity
for addition of
an element:-
search its correct
position:-

$$\boxed{m \times \log_{dm} n}$$

each
node can
have max.
(m-1) children

22, 21, 30, 31, 28,
→ 29

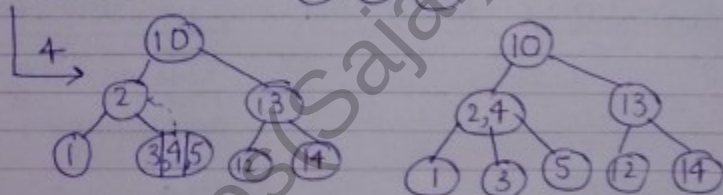
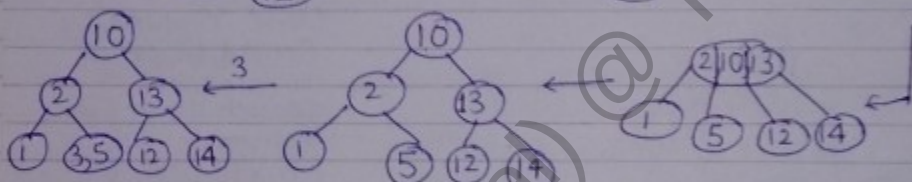
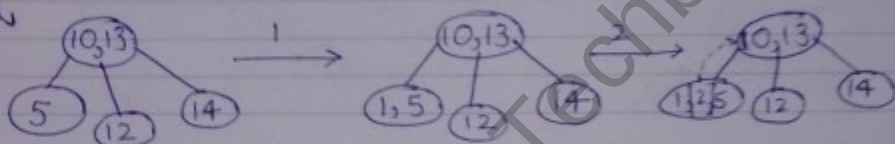
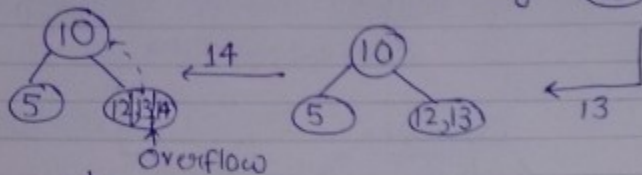
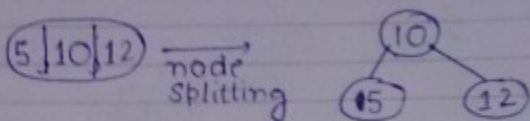


Note:-

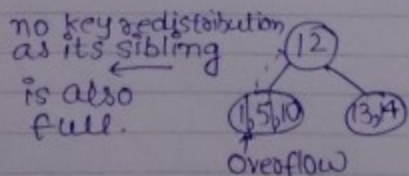
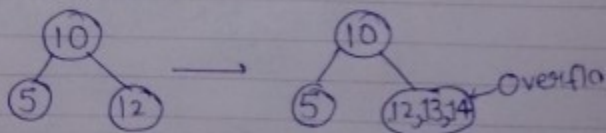
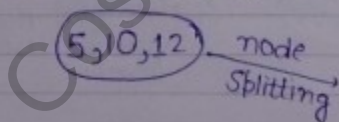
- (i) insertion is done
only in leaf
nodes
- (ii) all leaf nodes
are in the same
level.

★ ★ A / 2-3 tree (means max. 3 child)

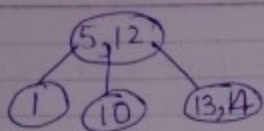
Q. degree = 3 | keys: 5, 10, 12, 13, 14, 1, 2, 3, 4



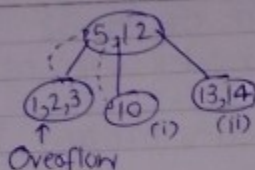
Q. Degree = 3
Keys: 5, 10, 12, 13, 14, 1, 2, 3, 4 (using key redistribution if possible)



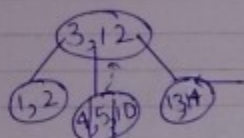
key redistribution (smallest one from that subtree which is under overflow condn.)



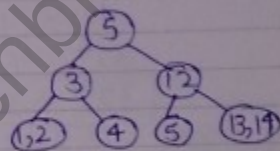
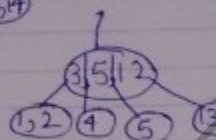
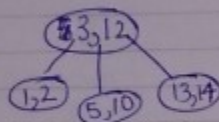
1, 2



if both the siblings (left & right) have space, carry on redistribution (left or right) for key redistribution, ask only the immediate siblings so ask for key redistribution with E_i only.



no key redistribution



Note:- While constructing B-Tree for the given tree keys, key-redistribution is always better than node splitting, because we are not wasting space & height of B-Tree also decreases, so time complexity also decreases

Q. Consider the following B-tree:-

2-3-4 Tree (A B-Tree with min. degree of 2, $\therefore$ max

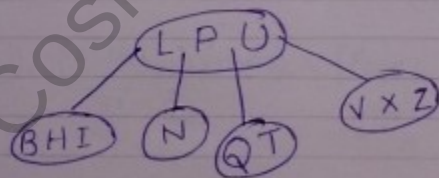
min. children = 2

min. keys = 1

max. children = 4

max. keys = 3)

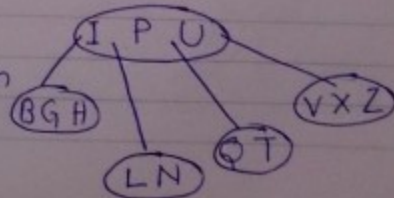
in which each data item is a letter



what is the result of inserting G in the above tree:-



key redistribution



Q. Consider a B-Tree with order = 32

(i) find max. no. of nodes present in a B-Tree with height

3.

(ii) find min. no. of nodes present in a B-Tree with height 3.

Ans. (i) height = $\log_{32} n$

$$3+1 = \log_{32} n$$

$$4 = \log_{32} n$$

$$32^4 = n$$

max. children = 32

max. keys = 31

min. children = 16

min. keys = 15

for max.

Level	nodes	keys	children
1	1	31	32
2	32	32×31	32×32
3	32×32	$32 \times 32 \times 31$	$32 \times 32 \times 32$
4	$32 \times 32 \times 32$	$32 \times 32 \times 32 \times 31$	$32 \times 32 \times 32 \times 32$

$$\text{Total nodes} = 32^3 + 32^2 + 32 + 32^0$$

$$= 32^4 \frac{32^4 - 1}{32 - 1}$$

$$\approx 32^8$$

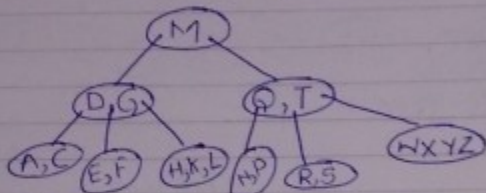
max. no. of nodes in a height h , order m B-Tree:-

$$= m^0 + m^1 + \dots + m^h$$

$$= \frac{1(m^{h+1} - 1)}{m - 1} = \frac{m^{h+1} - 1}{m - 1}$$

$$\text{max. keys} = (m-1)[m^0 + m^1 + \dots + m^h] = \frac{(m-1)(m^{h+1} - 1)}{(m-1)} = (m^{h+1} - 1)$$

Q. Consider the following B-Tree



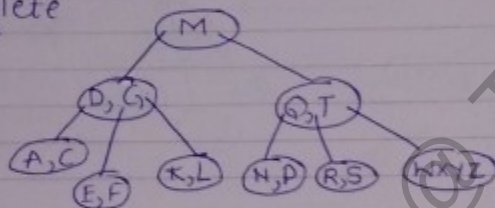
max. no. of keys = 4

order = 5

min. no. of children = $\lceil \frac{5}{2} \rceil = 3$

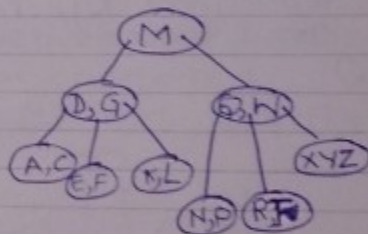
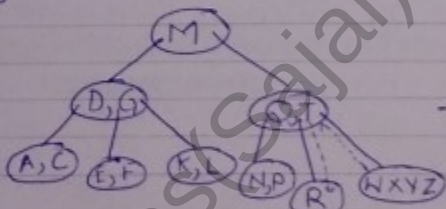
min. no. of keys = $3 - 1 = 2$

Delete
H



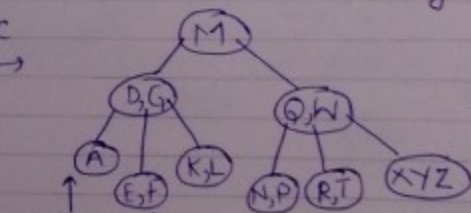
★ If the element H is present in leaf node, delete that element & check for the min. condⁿ. If it is satisfied, then do nothing, otherwise

Delete G

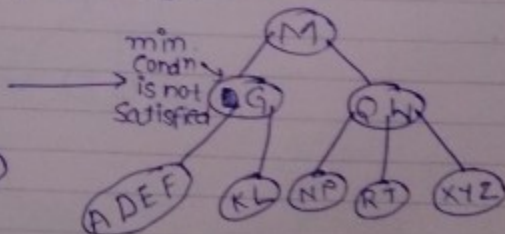


min. condⁿ not satisfied
ask left sibling → can't give
ask right → can "]
[if min. condⁿ is not satisfied,
distribute the keys from immediate
sibling (if possible)].

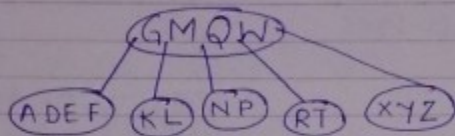
Delete C



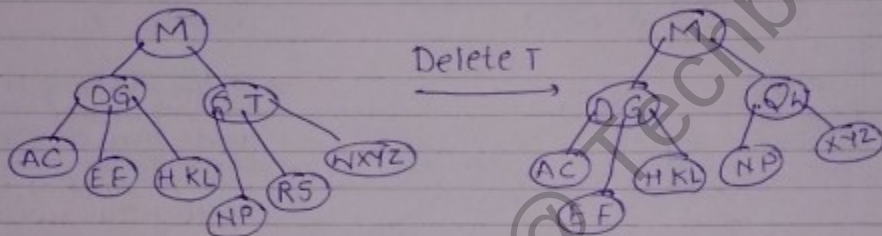
min. condⁿ is not satisfied
sibling can't help, so node merging



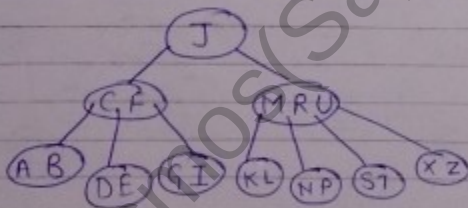
G asks the sibling, but it can't help, $\therefore$ node merging



Q. Consider the following B-Tree.

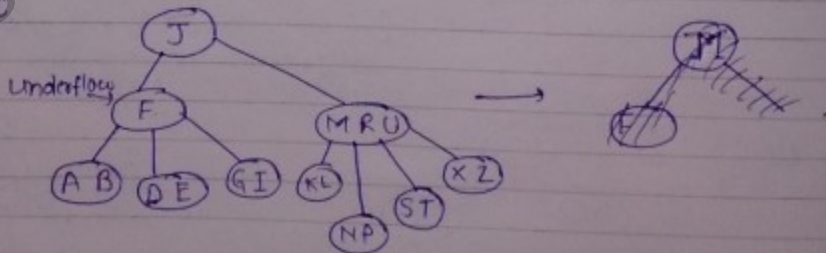


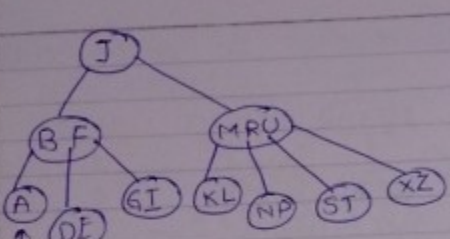
★ If we delete any element which is present in non-leaf node, replacement will be done by children, then if not by children, then it is done by sibling, still not satisfied merge with parent.



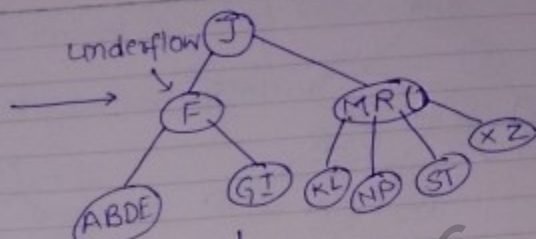
min. keys = 2
min. children = 3
max. keys = 4
max. children = 5

Delete C

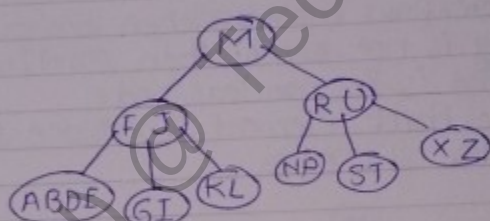




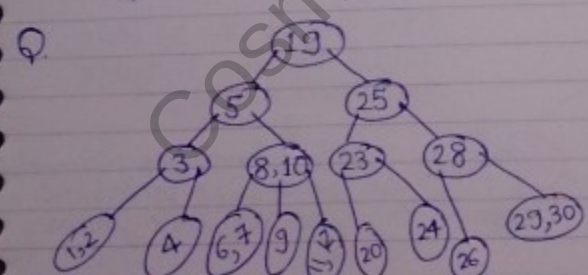
underflow
 (now A will ask its parent, if not satisfied by its sibling, then merge with parent)
 [in case of underflow, ask the parent, in case of deletion, replace blindly with children & then perform other opⁿ]



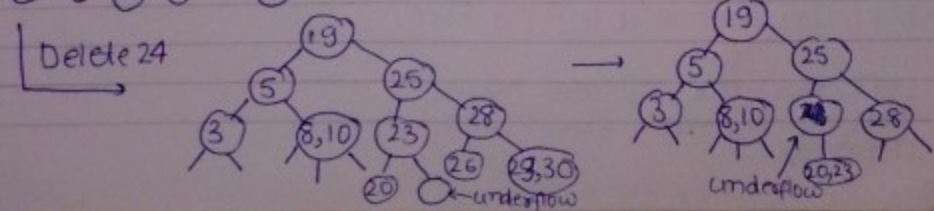
in case of underflow, ask the sibling 1st, (MRU) can give

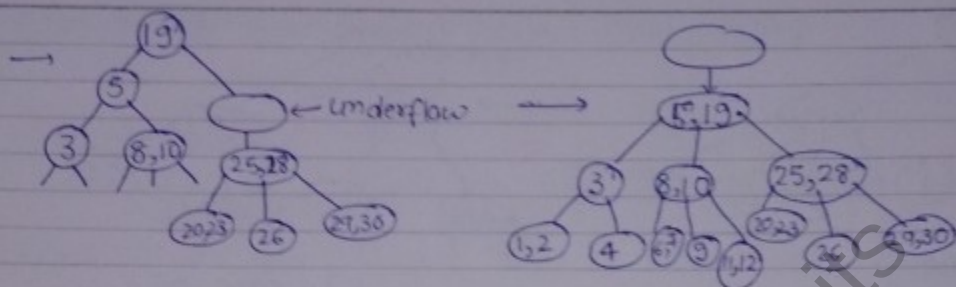


★★ In case of element deleted from non-leaf node, replace the element (or take the element) from children & then perform opⁿ of underflow (if there is some underflow in children).
 In case of underflow, ask help from siblings, if not then merge with parents.



min. - keys = 1
 min. - children = 2
 max. - children = 4
 max. - keys = 3

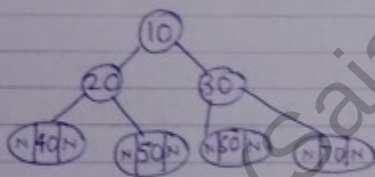




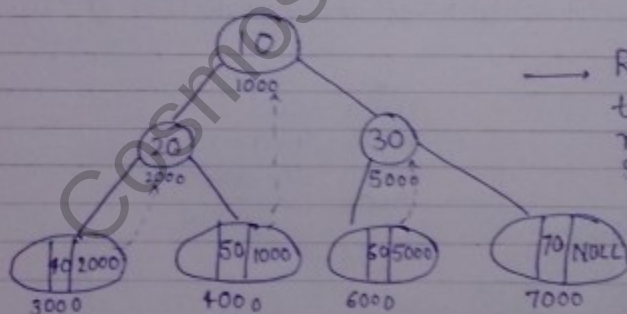
Deletion-Algorithm:-

1. Element is in leaf node
delete element & check for underflow
if underflow happened then borrow keys from siblings. If sibling is not ready then combine with the parent.
2. Element is in non-leaf node
delete that element & replace by inorder predecessor or inorder successor.

Q.



In Binary Tree, all leaf nodes left pointer & right pointer are null.



→ Right In threaded binary tree (right pt. of leaf node contain inorder successor address)

↓
To find inorder no. need of recursive procedure (no stack req.) using threaded BT.

→ Left-In Threaded Binary Tree

↓
Left pointer of leaf nodes contain inorder successor address.

Q In order 32 B-Tree with height 3, find min. keys & min. nodes.

level nodes keys children [min.] ✓

1	1	1	2
2	2	15*2	2*16
3	32(2*16)	32*15	2*16*16
4	2*16*16	2*16*16*15	0

Order m B-Tree with height h contain min. no. of nodes:-

$$1 + 2 \times \left(\frac{m}{2}\right)^0 + 2 \times \left(\frac{m}{2}\right)^1 + 2 \times \left(\frac{m}{2}\right)^2 + \dots + 2 \times \left(\frac{m}{2}\right)^{h-1}$$

$$= 1 + 2 \left\{ \frac{1 \left(\left(\frac{m}{2}\right)^h - 1 \right)}{\left(\frac{m}{2} - 1\right)} \right\} = 1 + 2 \left\{ \frac{\left(\frac{m}{2}\right)^h - 1}{\left(\frac{m}{2} - 1\right)} \right\}$$

Stack

One side is open another side is closed.

LIFO OR FILO :

Top is variable, which contain position of top most element.

ADT of Stack (what opⁿ perform on data structure)

1. push (stack, e) = stack (return)

2. pop (stack) = element of top.

3. IS empty (stack) = Boolean.

4. Full (stack) = Boolean.

Implementing Stack using array

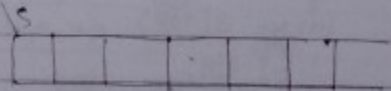
```
# int a[5] = {-----}
```

```
    a[10] = 0;
```

pf (a[10]) [a[10] is not allocated to you it give garbage]

Segmentation error = Mem is not available.

Transition (push)



push (g, N, TOP)

g: Name of stack or array

Top: of stack

N: Size of stack, array

push(S, N, TOP, X)

a	b	c	x	
0	1	2	3	
x	x	x	x	

deletion Top:
 --Top = 2
 --Top = 1
 --Top = 0
 --Top = -1

if (Top == N-1)

```

{
  printf("Stack is overflow");
  exit(1) ⇒ (abnormal Termination)
}

```

⇒ O(1)

else

```

{
  Top++;
  S[Top] = X;
}

```

S[Top++] = X [leads to data loss]

S[++Top] = X [✓]

pop(S, n, Top)

```

{
  if (Top == -1)

```

```

{
  printf("Stack is underflow"); ⇒ O(1)
  exit(1)
}

```

else

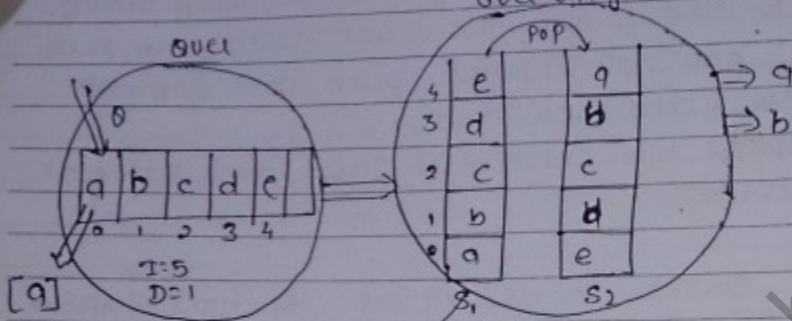
```

{
  int y = S[Top];
  printf("%d", y);
  Top--;
}

```

}

Q1) write a c program to implement queue using stack



1. $T = 5$ - push in S_1

2. $D = 5$ pop from S_1
 5 push into S_2
 1 pop from S_2

$\Rightarrow a$ (give '0' as deleted element)

1 pop from $S_2 \Rightarrow a$

2 pop from $S_2 \Rightarrow b$

3 pop from $S_2 \Rightarrow c$

4 pop from $S_2 \Rightarrow d$

5 pop from $S_2 \Rightarrow e$

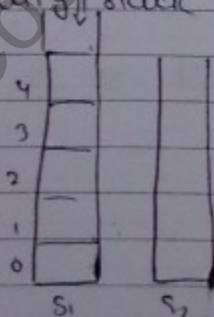
Q2) queue using stack

5 = T

3 = D

6 = T

3 = D



5 = push = S_1 ✓

5 = pop = S_1 ✓

5 = push = S_2 ✓

3 = pop = S_2 ✓

6 = push = S_1 ✓

2 = pop = S_2 ✓

6 = pop = S_1 ✓

6 = push = S_2 ✓

3 = pop = S_2 ✓

pop = $5 + 3 + 2 + 6 + 3$
 $= 19$ pop

22 push

17 pop

program

Dequeue()

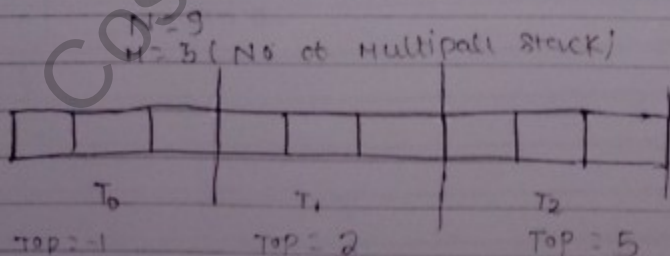
```

{
    if (s2 is empty)
    {
        if (s1 is empty)
            printf("underflow");
        else
        {
            while (s1 is not empty)
            {
                x = pop(s1);
                push(s2, x);
            }
        }
    }
}

```

#

Implementing multiple stack in signals



Find top: 1st Stack: (How many stack) $\frac{N}{M} - 1$

$$0 \times \frac{9}{3} - 1 = -1$$

$$1^{\text{st}} \text{ stack} = (1 \times \frac{9}{3}) - 1 = 3 - 1 = 2$$

$$2^{\text{nd}} \text{ stack} = 2(\frac{9}{3}) - 1 = 2 \times 3 - 1 = 5$$

$$\vdots$$

$$50^{\text{th}} \text{ stack} = 50(\frac{9}{3}) - 1 =$$

$$\vdots$$

$$i^{\text{th}} \text{ stack} =$$

$$i(\frac{M}{H}) - 1$$

Top

1 When we say the To is full.

Ans) When top of stack of To is equal to Top of 1st stack.

When 50th stack is full

$$\text{h)} \quad \text{Top}_{50^{\text{th}}} = 51(\frac{9}{3}) - 1 \quad \left(\text{next stack top} \right)$$

push opⁿ.

push (S, N, H, Ti, x)

(Ti = Top of th stack)

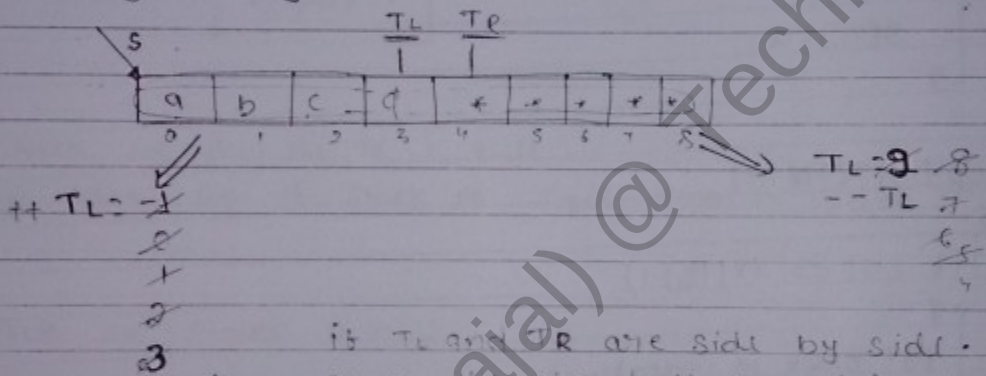
```
{
  if (Ti == (H * N) - 1)
  {
    p3( stack is full );
    exit(1);
  }
  else
  {
    Ti++;
    S[Ti] = x;
  }
}
```

pop (S, Ti) N, H)

```
{
  if (Ti == (H * N) - 1)
  {
    p3( stack is underflow );
    exit(1);
  }
  else
  {
    y = S[Ti];
    Ti--;
    return y;
  }
}
```

#1) using above c program we can implement multiple stack in single array but it is not efficient if -
 1) stack is full other remaining stack are empty it is giving stack is overflow.

#1) Efficient way of implementing multiple stack in single array.



if TL and TR are side by side then stack will give full condition but efficient use.

#1) $TL = TR - 1$

both should side by side
 $TL = 5$ $TR = 6$

$TL = TR - 1$

Application of stack

1) Recursion

a) total recursion

b)

1) Recursion

- i) Tail recursion
- ii) Non-tail recursion
- iii) Nested recursion
- iv) Indirect recursion

2) Prefix to postfix

3) postfix evaluation

4) Tower - Hanoi

5) Fibonacci Series

Q1 write a c program to find min^m element in the given array

Ans)

```
void min(int a, int n)
{
    if (n == 1)
        return a[0];
    else
        return min(a, n-1) < a[n-1] ? min(a, n-1) : a[n-1];
}
```

Recursive

#

min(a, i, j)

{ if (i == j)

return a[i];

else

{ if (a[i] > a[j])

min(a, i+1, j);

else

min(a, i, j-1)

}

Time complexity = $O(n)$ Space complexity = $O(n)$ Recursion

$$T = 1 + T(n-1)$$

$$\Rightarrow 1 + T(n-1) = \boxed{O(n)}$$

Q) Write a C-program to print the given array in the reverse order.

Ans

```
pr(a, i, j)
{
    if (i > j)
        return a[i];
    else
    {
        pr(a, i, j-1);
    }
}
```

$$T(n) = 1 + T(n-1)$$

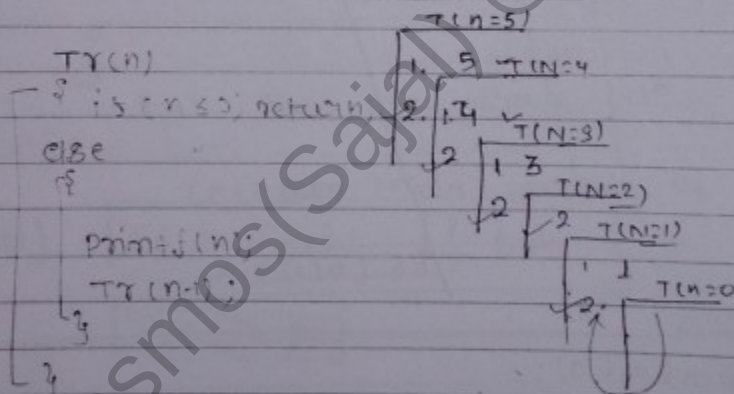
$$= O(n)$$

Space complexity: $O(n)$;

Time complexity: $O(n)$;

Tail recursion

##



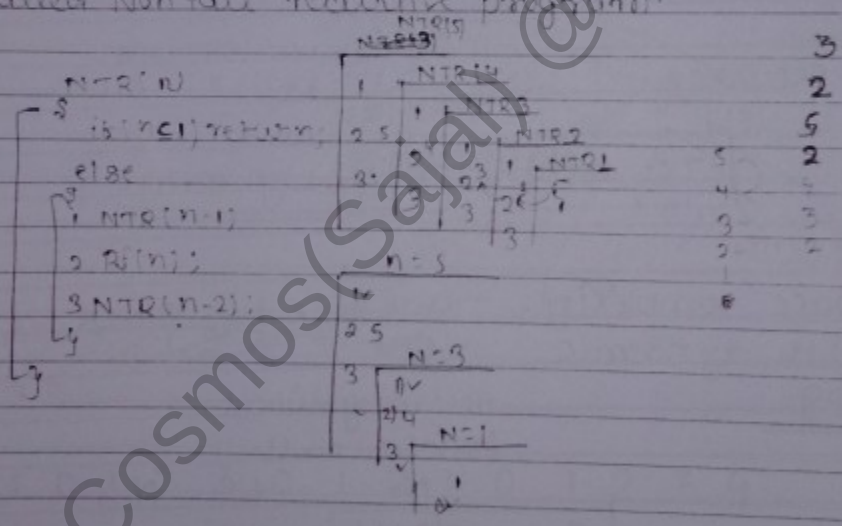
1. In the given recursive program, at the recursive function call is there then the function is called tail-recursive program.

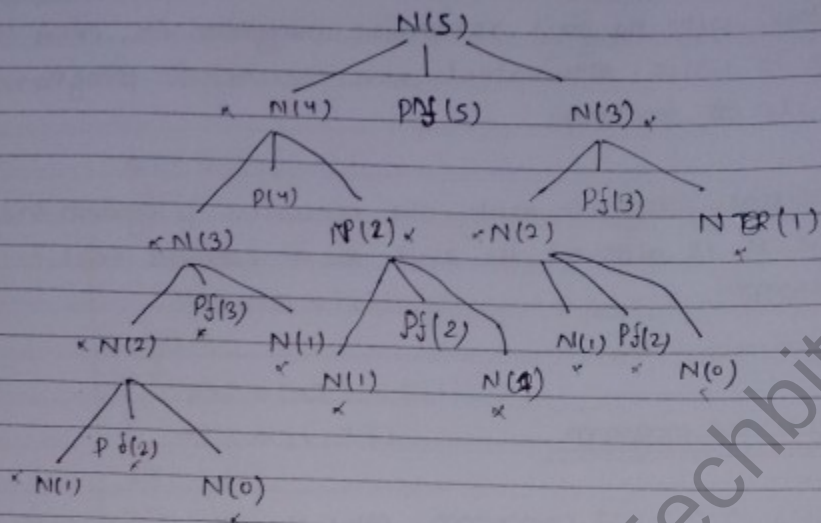
2. The draw back with tail recursive program is - we are waiting lot at stack step.

3. The advantage with the tail recursive program is, it is very easier to write, equivalent non recursive program with the help of for loop.
4. In given recursive program only one recursive program only one recursive is also at the end then called tail recursive program.

Non tail recursive program

In the given recursive program, After the recursive call there are something to do then recursive program called Non tail recursive program.





n level
binary
tree.

$$\downarrow$$

$$2^n$$

$$\downarrow$$

$$O(2^n)$$

2 3 4 2 5 2 3 Ans.

$T(n)$ Recursion: $T(n) = T(n-1) + T(n-2)$

$T(n) = O(2^n)$ [with dynamic]

Space complexity: Table size + Stack

with dynamic

$\downarrow$

Stack

$\downarrow$

prog

How many distinct
function call

N

+

N

$$S = O(2N)$$

without dynamic $T = O(2^n)$

Space: $O(n)$

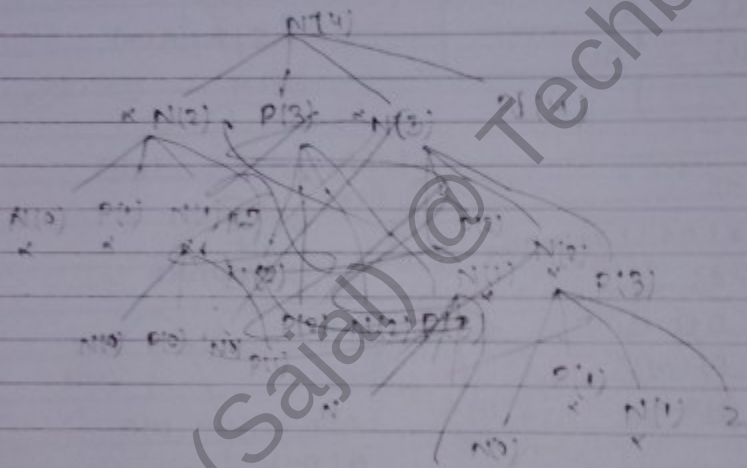
W/ dynamic program $O(n)$

Q1. Consider the recursive program

```

NTR(n)
{
    if (n <= 0) return
    else
    {
        NTR(N-2)
        pf(n-1)
        NTR(n-1)
        ps(n)
    }
}
    
```

NTR(4)



1 0 1 2 3 0 1 2 1 0 1 2 3 4

Time complexity $\Rightarrow O(n)$ } dynamic programming
 Space complexity $\Rightarrow O(n)$ }

Whenever function store in table then it's dynamic programming

NOTE! We are not waiting stack space necessarily

⇒ It is very difficult to find non-recursive program with the help of for loop

⇒

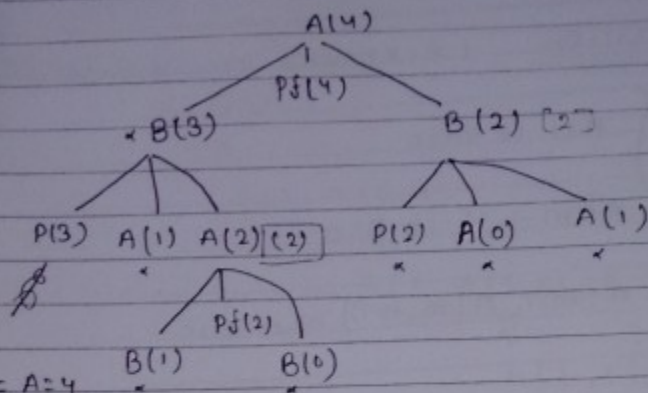
Indirect recursion

```
#1 A() B()
  { B()  { A()
    }    }
```

$T/P = A(u)$

```
A(n)
{ if (n ≤ 1) return
  else
  { B(n-1);
    pf(n);
    B(n-2);
  }
}
```

```
B(n)
{ if (n ≤ 1) return
  else
  { pf(n)
    A(n-2);
    A(n-1);
  }
}
```



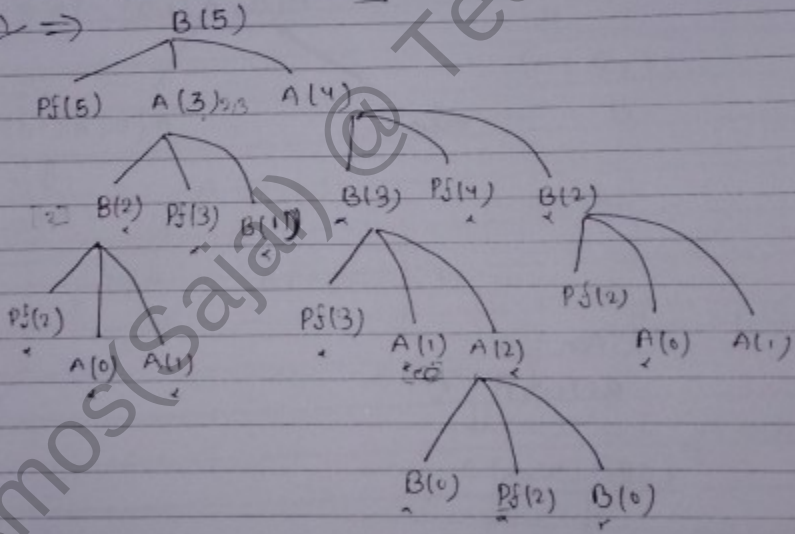
IP = A = 4

↓

3 2 4 2 Any.

$[IP = B = 5]$
↓

(not at home) ⇒



5 2 3 3 2 4 2

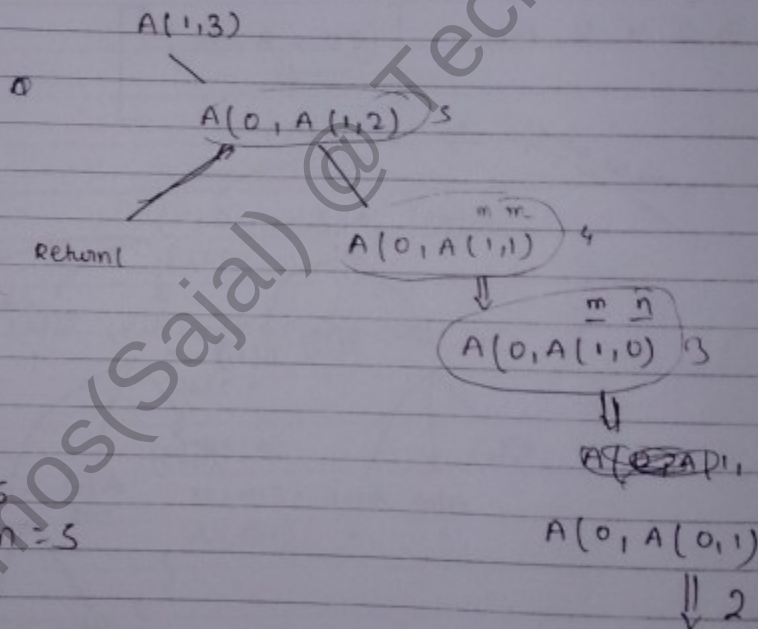
If T_n $A(n)$ then distance ~~function~~ function $= n^2$
distance

Nested recursion

(Ackermann's recursion relation)

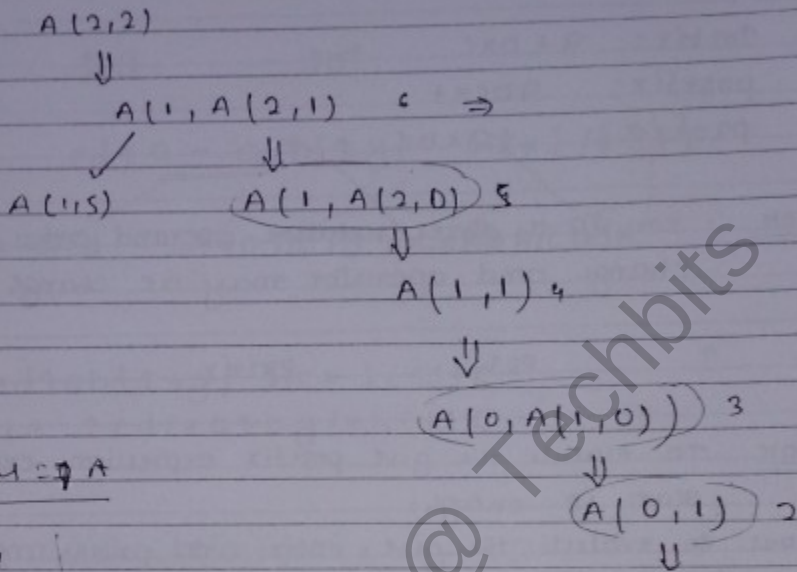
$$A(m, n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1, 1) & \text{if } n=0 \\ A(m-1, A(m, n-1)) & \text{other else} \end{cases}$$

$A(1, 3)$



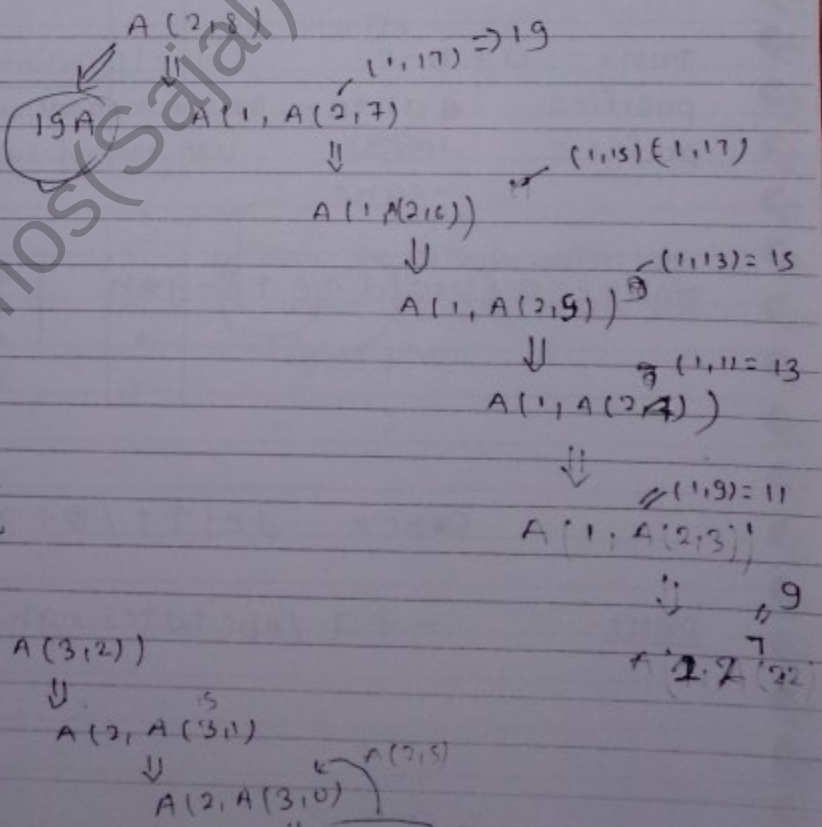
$A_m = 5$
Return = 5

$A(2,2)$



Ans: A

- (1,0)=2
- (1,1)=3
- (1,2)=4
- (1,3)=5
- (1,4)=5
- (1,5)=7
- (1,6)=8



$A(3,3)$

GI Ans

$A(3,3)$

$A(2, A(3,2))$

$\downarrow$

$A(2, A(3,1))$

$\downarrow$

$A(2, A(3,0))$

$A(2,1)$

Infix to postfix

Infix: $a + b \times c$

postfix: $abc \times +$

prefix: $+ a \times bc$

Note: In all the three notation operand order can't be change and operator may be change.

Infix Postfix Prefix

Note To evaluate the give postfix expression exactly one scan is enough.

but to evaluate the give infix and prefix many scan are required.

Infix: $a + b - c$

postfix: $abc - +$

prefix: $+ - abc$

(+, -) ((priority same))
when same who is in first with
but having high priority.

Infix: $a + b \times c / d + e \times f - g \times h$

[$\uparrow$ = higher priority
Associative property
for $\uparrow$ is right to left]

$a + b \times c \quad d e f \uparrow \uparrow / + g h \times A$

prefix

$- + a / \times b c \uparrow d \uparrow e f \times g h \quad \underline{A_1}$

Q1 infix = $e \uparrow d - a * b \uparrow f / g + h * c / i + j - k$

post fix $\uparrow e d$ $\uparrow b f$

$e \uparrow a - a * b \uparrow f / g$ $h * i / + j + k -$
 prefix = $- + - \uparrow e d$ $+ * a \uparrow b f / g$ $* h * i / + j + k -$

post fix = $e \uparrow a b f \uparrow + g / - h * i / + j + k -$

prefix = $- + + - \uparrow e d / * a \uparrow b g / * h c i j k$

Bracket Associativity is $L \rightarrow R$

$a^1 + b^2 * c^3$

Post fix

$a b c + + A$

+	3 pop
+	2

$a + b * c \uparrow d + e$

*
+
+

[when we occur with ^{low} lower priority than pop higher prion]

$a b c d \uparrow + + e -$

-

$a \uparrow b \uparrow c \uparrow d \uparrow e$
 $\emptyset$
 $\uparrow$
 $a b c d e \uparrow \uparrow \uparrow \uparrow \uparrow$
 $\uparrow$
 $\uparrow$
 $\uparrow$

Lower priority is coming then $\Rightarrow$ push.

High to low $\Rightarrow$ pop.

Left to right $\Rightarrow$ pop.

Right to left $\Rightarrow$ push.

Infix $e \uparrow d - a \times b \uparrow f / g + h \times c / i + j - k$

$e d \uparrow a b f \uparrow \times g - h c \uparrow i / \emptyset + j + \emptyset k -$

While converting infix to postfix we are using operator stack (only operators are push into stack)

$\uparrow$	//
$\times \times$	$\uparrow$
$i \times$	$\times \times -$

Max^m Stack space is = '3'.

$a + b \uparrow (c + (d \times e) \uparrow f) \uparrow g$

$a b c d e f \uparrow + g \uparrow \uparrow +$

$\times$
$\times \uparrow$
$+$
$\times \uparrow$
$\uparrow$
$+$

prefix to postfix

	prefix		postfix
1)	a	_____	a
2)	+ab	_____	ab+
3)	*a+bc	_____	abc+*

4) + * / - a b c d ↑ e / - e f g h

ab-c/d * ef-gh ↑ +

⇒

* a / b - c ↑ ↑ ↑ g f h + e * f, g, (Try at home)

g * b / c g f ↑ h ↑ e f g, * + ↑ ~~c b d g~~ - 1 *

Another way start from right.

* b

* a / b - c ↑ ↑ ↑ g f h + e * f, g,

Q1) $(a+b)(c-d) \times$

Find out infix (Postfix $\longrightarrow$ infix)

Ans $(a+b) \times (c-d) = a+b$

Q2) $(a \times b + c) \div d +$

~~Ans~~ $a \times b + c \div d$ Ans.

Postfix evaluation

Infix $= 2 + 3 \times 4 + 5 \uparrow 2 - 3 \times 6$

Postfix

$2\ 3\ 4\ \times\ 5\ 1\ \uparrow\ 2\ / \ 3\ 6\ \times\ -$

Max^m Stack Size = 2.

x	$\wedge / - \times$
+	

Total string length = 'N'.

$\lfloor \frac{N}{2} \rfloor$ = operand $\lfloor \frac{N}{2} \rfloor$ = operator.

Evaluation $\Rightarrow 2\ 3\ 4\ \times\ 5\ 1\ \uparrow\ 2\ / \ 3\ 6\ \times\ -$

		1 st Step	2 nd Step	3 rd Step
4	OP ₂	POP() = 3 rd = 4 $\times$ = 12	12 = 14	1 = 5' = 5
3	OP ₁	POP() = 2 nd = 3	2	5
2				

#			$14 + 2 = 16$
	6	* pop =	$OP_2 = 6 \quad OP_1 = 3 \quad = 18$
2	3 18	- pop =	$OP_2 = 18 \quad OP_1 = 14$
14	16		

$16 - 18 = -2$ A

```
#
int oic = 0;
n = postfix[i]
while (n != 10)
{
    if (n is operand)
    {
        push(n);
        i++;
    }
    else
    {
        OP2 = pop();
        OP1 = pop();
        r = OP2 * OP1;
        push(r);
    }
}
```

$\Theta(n)$ = Time complexity.

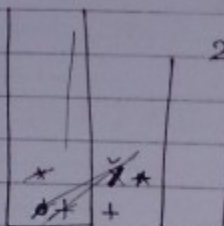
operand stack

$$\frac{\text{Max}^m \text{ Size}}{\text{Stack}} = \left\lceil \frac{n}{2} \right\rceil$$

NOTE While evaluating the given postfix expression we are using operand stack only operand are pushed into stack.

infix = $2 + 3 * 4 - 5 + 6 / 7 * 4 + 5$

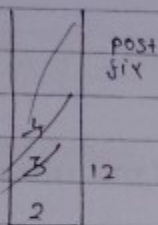
postfix



$234*-567/4*+5$

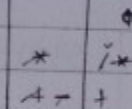
postfix = $234*+5-67/45+*+$

evaluation $\rightarrow$



post
fix

[Max^m Stack size = '2']



#1 Fibonacci Series

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

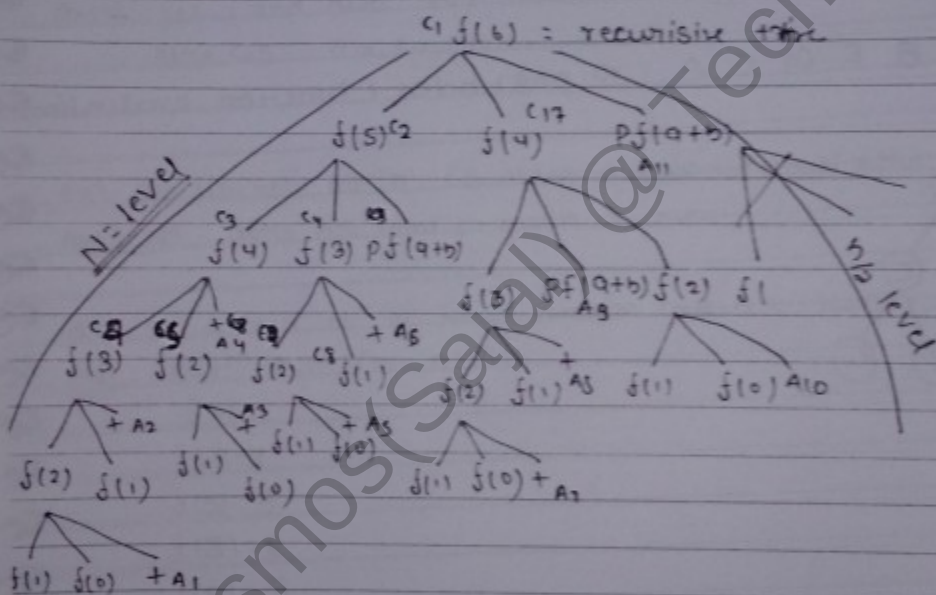
$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \end{cases} \quad \text{combinatorially: } n \text{ if } n=0 \text{ and } n=1$$

$$fib(n-1) + fib(n-2) \quad \text{otherwise}$$

```

# fib(n)
{
    if (n == 0 || n == 1)
        return n;
    else
    {
        a = fib(n-1);
        b = fib(n-2);
        return a+b;
    }
}

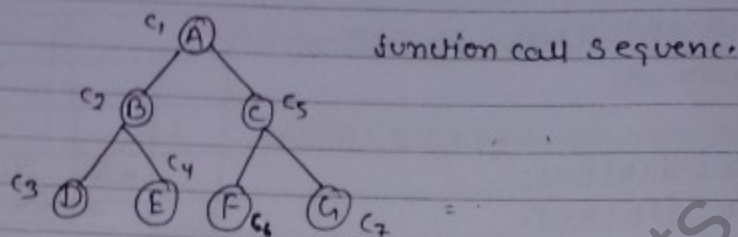
```



Make tree and give function nbre

Max^m Stack Size No of table

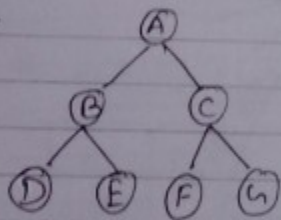
Function called in programming language = preorder.



A B D E C F G = function call sequence $\Rightarrow$ preorder

function Evaluation

D E B F G C A $\Rightarrow$ post order (function evaluation)



Q1) In fibbonaci of 'N' after how many function first addition will perform = $N+1$ function call.

Ans) Draw recursive tree it may be for any recursive program and give fun^c call order and count.

Q2) In fibbonaci of 'n' is there any addition after last function call.

Ans) yes (and also $N/2$ addition may be there)

After C25 = A10, A11, A12.

Q3) In fibbonaci of 'n' is there any function call after last addition.

Ans) No after A12 total program is over.

Q4) In fibbonaci of 'N' how many function call are there.

Ans) Max^m function $O(2^n)$

$$f(0) = 1$$

$$f(1) = 2$$

$$f(2) = 3$$

$$f(3) = 5$$

$$f(4) = 9$$

$$f(5) = 15$$

$$f(6) = 25$$

$$f(7) = 41$$

⋮

$$f(n) = f(n-1) + f(n-2) + 1$$

Given Fibonacci of 'n' how many addition will be there.

No. of addition

$$\Rightarrow f(0) = 0;$$

$$\Rightarrow f(1) = 0;$$

$$\Rightarrow f(2) = 1$$

$$f(3) = 2$$

$$f(4) = 4$$

$$f(5) = 7$$

$$f(6) = 12$$

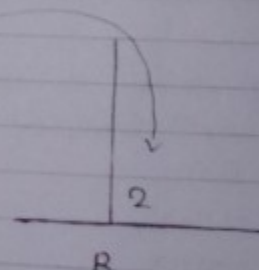
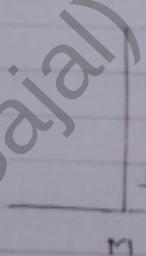
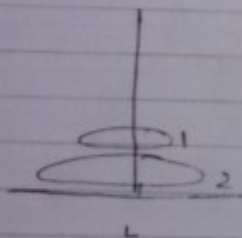
$$f(7) = 20$$

⋮

$$T(n) = 1 + T(n-1) + T(n-2)$$

$$f(n) = f(n-1) + f(n-2) + 1$$

Tower of Hanoi

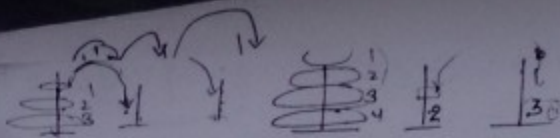


MOVES:

1) L → M

2) L → R

3) M → R

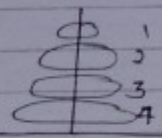


N=3

- L - R
- L - M
- R - M
- L - R
- M - L
- M - R
- L - R

N=4

- L - R
- L - M
- R - M



N=4 $\Rightarrow$

Top 3 have to send Middle = ie take 7 Step.

the move L - R (4 No block)

then 3 block at middle take 7 step Max total Step '7'.

- | | | |
|--------|---------|---------|
| 1) L-M | 6) R-M | 11) R-L |
| 2) L-R | 7) L-M | 12) M-R |
| 3) M-R | 8) L-R | 13) L-M |
| 4) L-M | 9) M-R | 14) L-R |
| 5) R-L | 10) M-L | 15) M-R |

N=5

- | | |
|--------------------------------|--------------|
| 1) Move Top 4 from L to M = 15 | } = 31 Step. |
| 2) Move Top 1 from L to R = 1 | |
| 3) Move Top 4 from M to R = 15 | |

$T_{OH}(n, \underset{\text{Source}}{L}, \overset{\text{Destination}}{R}, R)$

```

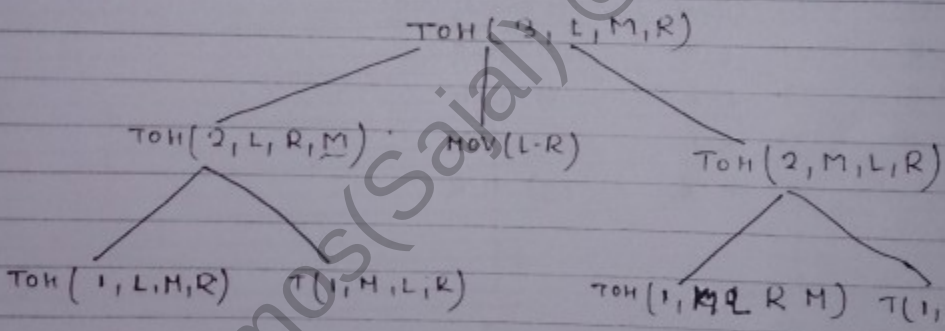
{
  if (n == 0) return
  else
  {
     $T_{OH}(n-1, L, R, M)$ 
     $T_{OH}(n-1, L, R, M)$ 
    Move (L → R)
     $T_{OH}(n-1, M, L, R)$ 
  }
}

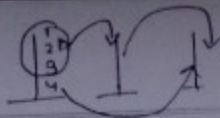
```

$$T(n) = T(n-1) + 1 + T(n-1)$$

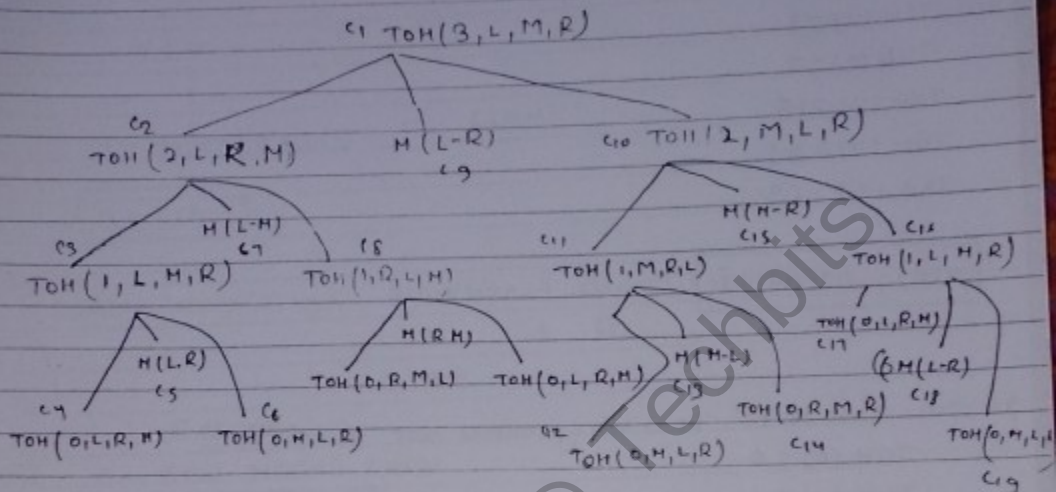
$$T(n) = 2T(n-1) + 1 \text{ (No. of moves)}$$

$$T(n) = 2T(n-1) + C = \text{Time complexity}$$





Recursive tree for TOH



Q1) In TOH(n) after how many function call move is take place

Ans) $N+1$ function call [After c4, there is H.]

Q2) TOH(n) is there any move after last function call

Ans) No move (last

Q3) In TOH is there any function call after last move

→ after H, there is [c15 (last function call)]

Q4) In TOH of '3' after how many function there is move for m

M → R

Ans) It is 6th move. (before it c12 function call)

Q5) In TOH of 'N' how many function call are there

No. of function call

$$T_{OH}(0) = 1$$

$$T_{OH}(1) = 3$$

$$T_{OH}(2) = 7$$

$$T_{OH}(3) = 15$$

$$T_{OH}(4) = 31$$

$$T_{OH}(5) = 63$$

$$T(n) = 2T(n-1) + 1$$

Q1 In T_{OH} of 'n' how many calls are there.

$$T_{OH}(0) = 0$$

$$T_{OH}(1) = 1$$

$$T_{OH}(2) = 3$$

$$T_{OH}(3) = 7$$

$$T_{OH}(4) = 15$$

$$T_{OH}(5) = 31$$

⋮

$$T(n) = (2T(n-1) + 1)$$

$$T(n) = 2^n - 1$$

Time $T(n) = O(2^n)$

Space: No. of level $O(n)$

Dynamic programming

$$T(n \dots) \\ \vdots \\ (nH) (3) (2) (1) = 6(n+1)$$

$$T = O(n)$$

Space complexity = Stacksize + auxiliary function

$$n + 6n \\ = 7n$$

Chapter

Q5) C - Row-Major Order

⁰⁼³³ ⁴⁰⁼⁹⁹
 char a[100][100]
 Base address = '0'
 $a[40][50] = 0 + (40)100 + (50-0)$
 $= 4000 + 50$
 $= 4050$

Q6) a 7) a 8) c 9) a 10) b 11) a 12) c 13) 14) 15) d

Q8)

		a/b			
a	a	b	c	d	
	e	f	g	h	
	i	j	k	l	
	m	n	o	p	

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

11) int a[2][10];
 int b[2][9]

a

1	5	9	13
0	0	10	14
0	0	11	15
0	0	0	16