(2)

```
int a=1, b=2;
main()
{
  int a = 20, b = 30;
  pf(a,b);
  C();
  pf(a,b);
  D();
}

D()
{
  pf(a,b);
  a=3, b=4;
  pf(a,b);
  E();
}

C()
{
  int a=50;
  pf(a,b);
  D();
  pf(a,b);
}

E()
{
  int ~~a=b;~~ b=6;
  pf(a,b);
  a=7, b=8;
}
```

	Static	Dynamic
①	20, 30	20, 30
②	50, 2	50, 30
③	1, 2	50, 30
④	3, 4	3, 4
⑤	3, 6	3, 6
⑥	50, 4	7, 4
⑦	20, 30	20, 4
⑧	7, 4	20, 4
⑨	3, 4	3, 4
⑩	3, 6	3, 6

(3)

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

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

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

E(int b, int a)
{ pf(a,b);
  a=b+a;
  a=b-a;
  a=7, b=8;
  pf(a,b);
}
```

	Static	Dynamic
①	3, 0	3, 0
②	0, 0	3, 0
③	4, 5	4, 5
④	6, 4	6, 4
⑤	7, 8	7, 8
⑥	4, 6	4, 6
⑦	4, 5	4, 5
⑧	3, 5	4, 5
⑨	6, 4	6, 4
⑩	7, 8	7, 8
⑪	4, 6	4, 6

Programming and Data Structures:

I. Programming:
→ Basics
(Operators, loops, functions, switch)
→ Storage classes, recursion
→ scope
→ pointers, strings
→ arrays
→ structures

II. Data Structures:
→ Dynamic Memory allocation
→ Linked list
→ Stack
→ Queue
→ Trees (BST, AVL)
→ Hashing

Textbooks:
1. 'c" language → Kernighan & Dennis Ritchie
2. Data Structures → "Sahani"
3. 'c' Test your aptitude → Venugopal & N. Chandrakanth

Faculty: Balakrishna Veerala Sir
Email ⇒ balakrishna.veerala@gmail.com

Facebook Page: Balakrishna Veerala CS

Basics of Programming:

Operators:

Operator	Precedence	Associativity:
( )	1. (high)	
*, /, %	2.	L → R
+ −	3.	L → R
=	4. (low)	R → L

*Associativity: If two or more operators are having the same precedence, then, the expression will be evaluated for using associativity.

E.g. ① $\overset{6}{2*3}+4$ = 10 | ② $\overset{2}{20/10}*3$ = 6 | ③ $2+\overset{5}{3}+\overset{20}{4*5}$ = 25 | ④ $15/7*2$ = 4

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

int, int → int
float, int → float
float, float → float

Expression	Assigned to int		Assigned to float	
5	5	5	5.0	5.0
5/2	2	2	2.0	2.0
5.0/2	2	2	2.5	2.5
5/2.0	2	2	2.5	2.5
5.0/2.0	2	2	2.5	2.5
2/5	0	0	0.0	0.0
2.0/5	0	0	0.4	0.4
2.0/5.0	0	0	0.4	0.4

Modulus Operator (%):

① 15 % 7 = 1
② −15 % 7 = −1
③ +15 % −7 = +1
④ −15 % −7 = −1
⑤ −15.5 % +7 = error
⑥ −3 % +5 = −3
⑦ +3 % −5 = +3

② $+7\overline{)-15}\,(-2$, -14, -1
③ $-7\overline{)+15}\,(-2$, $+14$, $+1$
④ $-7\overline{)-15}\,(2$, -14, -1

NOTE
↳ Modulus always gives numerator sign.
→ Modulus works only on integers. It does not work for float values.
→ If the numerator value is small without sign, it gives same value as output.

Relational and logical operators:

① All the relational and logical operators return 0 or 1.
② If the expression is true, it returns 1, and if the expression is false, it returns 0.
③ All non-zero is considered as true and "0" is considered as FALSE.

FALSE	TRUE
0	1
	-2
	2.5
	8
	-13.5

Eg ① a = 5 > 4 → a = 1 ①

② a = (5 > 4) + (5 > 4) → a = 2

③ a = 5 > 4 > 3 → a = 0 (1 > 3)

Eg. int i = 2 * 3/4 + 2.0/5 + 8/5;
printf("%d", i);

IF - ELSE:

output: Raj 8

```
Eg. main()
{ int a=5;
  if (a = 8)
  {
    printf("Raj");
  }
  else
  { printf("Keshore");
  }
  printf("%d", a);
}
```

a 8

**

NOTE
↳ Assignment operator assigns the value and returns assigned value.

printf();

Eg ① main()
{
int i;
i = printf("Hello");
printf("%d", i);
}

o/p:
Hello 5

Eg ② main()
{
int a;
a = printf("made, %d, easy", printf("Lucknow"));
printf("%d", a);
}

o/p: Lucknow made, 7, easy 11

NOTE
↳ printf() always returns, i.e., how many symbols displayed on the screen by that particular printf() only.
→ If we want, we can collect that value also.

scanf():

Q. WAP to read 2 integer values and display them on the screen.

NOTE
→ scanf() always returns integer, i.e., number of proper inputs given.

Age, yob	
*, $	invalid
2.5, 1-5	invalid
3.5, 1997	invalid
-3.5, 20	invalid
24, 1999	valid

```
main()
{
unsigned int a,b,sn;
   printf("Enter two integers only");
   BEGIN:
     sn= scanf("%d %d",&a,&b);
     if(sn!=2)
       { pf("wrong input given");
         pf("\n Re enter proper integers only");
          goto BEGIN;
       }
     pf("%d %d", a,b);
}
```

Ternary Operator:

condition ? if it is true : if it is false

NOTE
→ There should be an equal number of colon(:) & question marks and if they are not equal, it returns an errors.
→ Every colon(:) matches with the just before question mark.
→ Every question mark(?) followed by colon, not immediately but following.

04/10/2017

Eg ① : a = 5>4 ? 6 ? 10 : 20 : 30
a = 20

③ a = 6>7 ? 10 : 20
a = 20

Eg ② a = 3>4 ? 10 : 8>7 ? 20 : 30
a = 20

④ a = 1 + (2 ? 2+3 : 3) + 4;
a = 1 + 5 + 4 = 10

Q. a = 2>3 ? 5>4 ? 10 : 30 : 8>7 ? 30 : 40
a = 30

Q. a = 6>7 ? 2>3 ? 10 : 6>8 ? -2 : 3 : 0 ? -1 : 6>8 ? 10 : 3>4 ? 10 : 20
a = 20

Q. WAP to find the maximum of 3 no. using ternary operator.

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

Q. WAP to find minimum of 4 nos.

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

GATE 2008
Q. which combination of the integer variables x, y and z makes the variable 'a' get the value 4 in the following expression?
a = x>y ? x>z ? x : z : y>z ? y : z

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

Pre | post increment | decrement :

pre	post	pre	post
a = 10;	a = 10;	a = 10;	a = 10;
b = ++a;	b = a++;	b = --a;	b = a--;
a | b	a | b	a | b	a | b
11 | 11	11 | 10	9 | 9	9 | 10

Q.1. a = 1, 2, 3, 4
b = ++a * ++a * ++a;
(++a → 2, 3, 4)

a	b
4	64

② a = 1
b = ++a * a++ * ++a;
(++a → 2, 3; a++ → 3; ++a → 3)

a	b
4	27

③ a = 1
b = a++ * ++a * a++;
(a++ → 2; ++a → 2; a++ → 2)

a	b
4	8

④ a = 8
a = ++a * a++ * ++a;

a
Void

⑤ a = 50;
b = 50;
a = a++ + ++b;
b = b++ + ++a;

a	b
103	155

Loops :

① while
② do-while
③ for

Q. WAP to print 1 to 10 numbers using do-while & for loop.

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

```
main()
{int i; for (i=1; i<=10; i++)
   {printf ("%d\n", i);
   }
}
```

for (i=1; i≤10; printf("%d", i++));

①
```
main()
{
  int i = 5;
  while (i++ ≤ 7);
  printf("%d", i);
}
```
o/p: 9

②
```
main()
{
  int i = 1;
  do
  {
    while (i++ < 1);
  }while(i++ <= 2); ≥4
  printf("%d", i); 5
}
```
o/p: 5

Q. WAP to print following pattern of input:

Input 6 ↵

o/p: 1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6

Input 6 ↵

o/p:
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1

Q3. input 7 ↵

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

Switch-Case:

Syntax:

```
main()
{
    switch(Expression)
    {
        case Label1:
            body;
            break;
        case Label2:
            body;
            break;
        default:
            body;
            break;
    }
}
```

NOTE

→ Case and label should have space.
→ case label must end with a colon.
→ Case label must be constant.
→ case label should not be floating point numbers.
→ only 1 default will be allowed and it can be placed anywhere in the switch.

→ empty switch case is allowed.
→ default will be executed only when no case is matching.
→ After every case, break is required, otherwise, it will execute all the following cases until the occurence of break or till the end of the switch.

Q1.

```
main()
{
    int x;
    scanf("%d", &x);
    switch(x)
    {
        case 0: x = x+1; 1
        case 1: x = x+2; 3 3
        default: x = x+3; 6 6 5
    }
    printf("%d", x);
}
```

Input	Output
x = 0	6
x = 1	6
x = 2	5

Q2.

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

Input	Output
x = 0	1
x = 1	11
x = 2	102
x = 3	12

GATE. 2011

Q. main()
{
int i;
for (i=1; i<=25; i++)
{
switch (i)
{ case 0 : i += 5;
case 1 : i += 3;
case 2 : i += 4;
default : i += 5;
break;
}
printf("%d", i);
}
}

O/P: 13 19 25

GATE 2015

Q. main()
{
int i, j, k = 0; // k = -1
j = 2 * 3/4 + 2.0/5 + 8/5;
k -= --j; // k = -1
for(i = 0; i<5; i++)
{
switch (i+k)
{
case 1:
case 2: printf("%d", i+k); No
case 3: printf("%d", i+k); hack
default: printf("%d", i+k);
}
}}
The no. of times printf()
statement executed? 10

Q.
main()
{
int x = 5;
printf("%d", ++(x++)); error
printf("%d", (++x)++); error
printf("%d", (x+0)++); error
}

NOTE
↳ Pre increment and post increment, works only on the variables but not on constants and not on expressions.

#Functions:
→ The purpose of function is code reutilization.
→ The purpose of loop is code repetition.
→ The purpose of header file is function reutilization.
→ If you want to reuse the functions, then, we need to create the header file and include it in the required program.
→ Function has the following things:
① Prototype / Declaration
② Function Definition
③ Function Calling

Eg.
```
int add (int, int);          //prototype/declaration
main ()
{ int a;
  a = add (10,20);           //calling
  printf("%d", a);
}
int add (int x, int y)       //func" definition.
{ return (x+y);
}
```

#Short Circuit Evaluation:

Q1. a=1 b=1 c=1
a = ++b > 2 && ++c > 1;
pf(a,b,c);
O/P → 0, 2, 1

Q2 a=1 b=1 c=1 d=2
a = (++b > 1 || ++c > 1) && ++d > 1;
pf(a,b,c,d);
O/P → 1, 2, 1, 3

Q3. int i=-1, j=-1, k=-1, l=2, m;
m = (i++ && j++ && k++) || (l++);
pf(i, j, k, l, m);

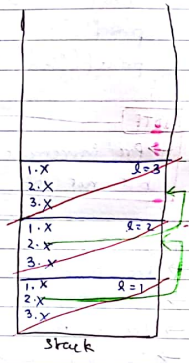
input	output
original	0 0 0 2 1
if k=0	0 0 1 3 1
if i=0	1 -1 +3 1

Recursion:

→ Function calling itself is known as recursion.
→ Recursion should definitely have a termination condition, otherwise it leads to infinite condition[loop].

Eg.
```
void mec(int l)
{
1. pf("\n%d", l);
2. if (l<=2)
     mec(l+1)
3. pf("\n %d", l);
}
void main()
{
  mec(1);
}
```

o/p →	
	1
	2
	3
	3
	2
	1



NOTE
↳ Recursive program internally executes with the help of stack.
→ Every time when the recursive call is made, activation record will be created in the stack.

Activation Record: The memory allocated in the stack frame for the recursive function call is called as activation record.

Bitwise Operator:

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
<< >>	Right shift
>> <<	Left shift

Eg ①

12 = 1100 = 00001100
& 25 = 11001 = 00011001
8 00001000

12 = 00001100
| 25 = 00011001
29 = 00011101

C language program:

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

o/p ⇒ 8 29

Right shift: [>>] Right shift operator shifts the bits towards the right side by certain number of specified bits.

212 = 11010100 = 00000000 11010100
212 >>2 = 00000000 00110101 = 53
212 >>7 = 00000000 00000001 = 1
212 >>8 = 00000000 00000000 = 0
212 >>0 = 00000000 11010100 = 212

#Left shift (<<):

→ Left shift operator shifts all the bits towards the left by certain number of specified bits.

```
212     = 00000000 11010100
212<<1  = 00000001 10101000 = 424
212<<0  = 00000000 11010100 = 212
212<<4  = 00001101 01000000 = 3392
```

Program:

```
main()
{
 int num=212, i;
 for(i=0; i<=2; i++)
 printf("\n%d%d", i, num>>i);
 for(i=0; i<=2; i++)
 printf("\n%d%d", i, num<<i);
}
```

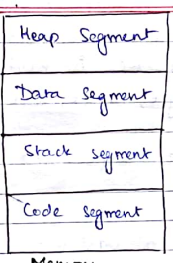
Output:

0	212
1	106
2	53
0	212
1	424
2	848

GATE 2017 2 marks Set-1 H.W.

05/12/2017

#Memory Segment:



Memory

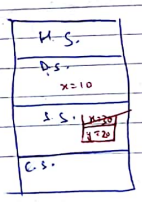
* Code Segment: In this segment, all the general instructions of the program which will not change throughout the execution will be stored.
→ Generally, this part is considered as read only and protected.

* Stack Segment: In the stack segment of the memory, all the recursive function calls, local variables and formal parameters will be stored.

* Data Segment: In the data segment of the memory, all the global data like global variables and static variables will be stored.

* Heap Segment: All the dynamic memory allocations like allocating memory by using malloc() calloc() and realloc() will be stored.

E.g.
```
int x=10;
main()
{ int y;
  y=20;
  { int x=30;
    pf("%d", x);
  }
  pf("%d%d", x, y);
}
```
o/p: 30 10 20



Storage Classes:

→ In the C language, every variable is having some characteristics like data type, scope, lifetime and default value.

→ The storage classes describe scope and lifetime of a variable.

* **Scope:** The region in which a variable is accessible and ~~avail~~ available is called as scope of a variable.

Type	Scope	Life time	Default value	storage	Memory segment
Auto	Body	Body	Garbage value	Memory	Stack segment
Register	Body	Body	Garbage value	Register	Register
Static	Body	Program	0	Memory	Data segment
Extern	Program	Program	0	Memory	Data segment

#Auto:

Syntax: (auto) datatype variablename; (auto: keyword)

E.g. int int x;

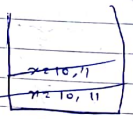
→ By default, all the local variables are auto variables.
→ If auto variable is not initialised, then, it contains garbage value.
→ For every function call, auto variable is re-created and reinitialised.
→ It has a local scope and local life time.

E.g.

```
main()
{ test();
  test();
}
```

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

o/p: 10 10



Stack seg.

#Register:

Syntax: register datatype variablename;

E.g. register int x;

→ The keyword register is not a command but it is just a request to the compiler to allocate the memory in registers instead of RAM.
→ If the free memory is available, then, memory will be allocated in register otherwise mem. will be allocated in the RAM only.
→ Register variables are faster compared to the auto variables and they will be generally used in loops and functions.
→ If it is not initialised, by default, it contains garbage value and it has local scope and local lifetime.

#Static:

Syntax: static datatype variablename;

E.g. static int x;

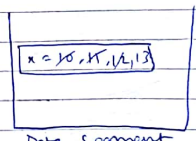
→ The variable persisting the previous static value, even though, after the destruction of several function calls is called as a static variable.
→ If static variable is not initialised, by default it contains 0.
→ The static variables will be created once and memory will be allocated once.
→ It has a local scope and lifetime will be ended after completion of the program.

E.g.

```
main()
{
  test();
  test();
  test();
}
```

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

o/p: 10, 11, 12



Data segment

Q1. int f (int n)
{
static int i = 1;
if (n >= 5)
return n;
n = n + i;
i++;
return f(n);
}
What is the return value of f(1)?
(a) 6 (b) 7 (c) 8 (d) 9

Q2. int incr(int i)
{
static int count = 0;
count = count + i;
return count;
}

main()
{
int i, j;
for (i = 0; i <= 4; i++)
j = incr(i);
}

What is the value of 'j' after the ~~execution~~ termination of program?

i	j	count
0	0	0
1	1	1
2	3	3
3	6	6
4	10	10

10

Q3
int f (int n)
{ static int r = 0;
if (n <= 0)
return 1;
if (n > 3)
{
r = n;
return f(n-2) + 2;
}
return f(n-1) + r;
}
What is the value of f(5)?
(a) 16 (b) 17 (c) 18 (d) 19

f(5)
r=5
f(3)+2 → 8
f(2)+5 → 16
f(1)+5 → 11
f(0)+5 → 6

Q4. main()
{
int i, x = 5, y = 10;
for (i=1; i<=2; i++)
{
y += f(x) + g(x); y = y + f(x) + g(x);
printf ("%d", y);
}
}
f(int x)
{ int y;
y = g(x);
return (x+y);
}
g(int x)
{ static int y = 5;
y = y + 7;
return (x+y);
}

i=1

main()	f()	g()
x = 5	f(5)	g(5) y = 5/12/19
y = 10	→ y = g(5) = 17	→ ret 17
y = 10 + 22 + g(5)	ret(22)	g(5)
10 + 22 + 24		→ 24 return
y = 56		

i=2

x = 5	f(5)	y = 19/26/33
y = 56	→ y = g(5) = 31	g(5)
y = 56 + f(5) + g(5)	ret(36)	→ return (31)
= 56 + 36 + 38		g(5)
y = 130		→ return (38)

186

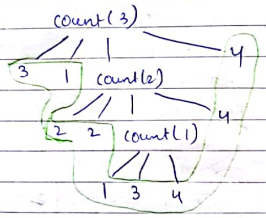
What is the sum of outputs printed by the above program.

GATE 2016 2 marks

Q5

```
void count(int n)
{
  static int d=1;
  printf("%d",n);
  printf("%d",d);
  d++;
  if(n>1)
  count(n-1);
  printf("%d",d);
}
void main()
{ count(3);
}
```

d = 1 2 3 4



What is the o/p printed

(a) 312213444 (b) 3121111222 (c) 3122134 (d) 3121112

Extern:

Syntax: extern datatype variablename;

→ These variables have scope and lifetime throughout the entire program.

→ They similarly behave like global variables and default value is 0.

→ Nomemory will be allocated for extern variables and they will take the value of external variable, i.e., global variable.

Eg①.
```
int a=8;
main()
{ extern int a;
  pf(a);
  a=20;
  pf(a);
}
```
o/p: 8 20

②
```
main()
{
  extern int a;
  printf("Hello");
}
```
o/p: Hello

③
```
main()
{
  extern int a;
  pf(a);
}
```
o/p: error

④
```
int a=5;
main()
{
  extern int a;
  extern int a;
  extern int a;
  pf(a);
}
```
o/p: 5

Recursion:

→ Function calling itself is called as recursion.

→ Recursion should definitely have a termination condition, otherwise, it will go into infinite loop.

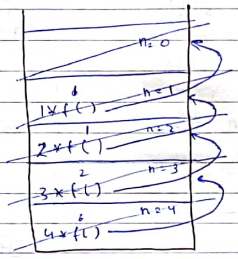
→ Four types of recursions available:

① Tail recursion
② Non-tail recursion
③ Indirect recursion
④ Nested recursion

Tail recursion:

→ In the program, very last statement is the recursive call and there is no other statement after that, then, it is called tail recursion.

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



24

o/p of fact(4)?

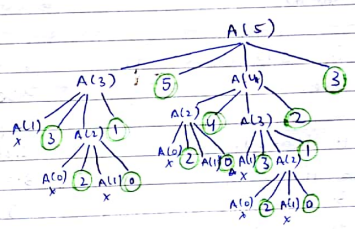
E.g ② void mec()
{
int l;
static int s;
l= ++s;
pf("\n%od %od", l, s);
if (l<=2)
mec();
}
void main()
{ mec();
mec();
}

o/p → 1 1 2 2 3 3 4 4

Non Tail Recursion:

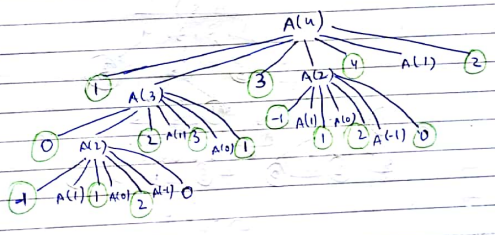
→ In the program, recursive call can be anywhere other than the last statement, then, it is called as non-tail recursion.

E.g ① A(n)
{
if (n≤1) return;
else
{
A(n-2);
pf(n);
A(n-1);
pf(n-2);
}
}



What is the output of A(5)? 3 2 0 1 5 2 0 4 3 2 0 1 2 3

② A(int n)
{
if (n≤1) return;
else
{ pf(n-3);
A(n-1);
pf(n-1);
A(n-2);
pf(n);
A(n-3);
pf(n-2);
}
}



Q What is the sum of o/ps when called for A(4)? 20.

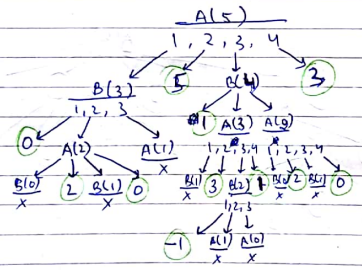
Indirect recursion:

→ Two or more functions calling each other is known as indirect recursion.

E.g.

```
A(int n)                    B(int n)
{                           {
  pf(n ≤ 1) return;           pf(n≤1) return;
  else                        else
  {                           {1. pf(n-3);
  1. B(n-2);                  2. A(n-1);
  2. pf(n);                   3. A(n-2);
  3. B(n-1);                    }
  4. pf(n-2);                 }
  }
}
```

What is the sum of all outputs printed when called for A(5)?



O/P: 0, 2, 0, 5, 1, 3, -1, 1, 2, 0, 3 ⇒ 16

Sum of the o/p for B(5)? ⇒ 12.

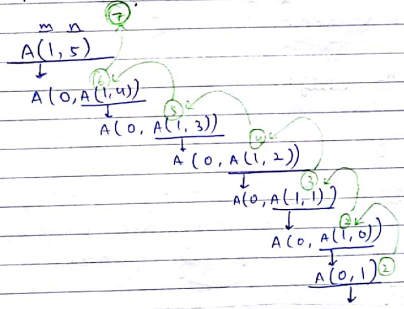
2, -1, 4, 0, 2, 0, 2, 3, -1, 1

Nested Recursion:

→ The recursive function which is passing itself as a parameter to a recursive call is called as nested recursion.

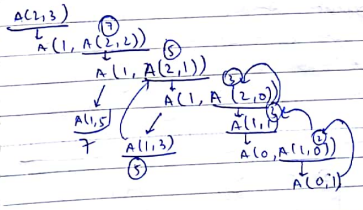
Q. $A(m,n) = \begin{cases} n+1 & , \text{if } m=0 \\ A(m-1,1) & , \text{if } n=0 \\ A(m-1, A(m,n-1)) & , \text{otherwise} \end{cases}$

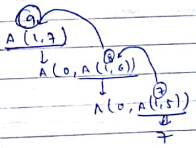
what is the o/p of A(1,5) = ?



output
A(1,5) = 7

what is the value of A(2,3)? = 9.





Q. what is o/p of ① A(3,2) ?
② A(3,3) ?

06/10/2017

Scope:

→ There are 2 types of scoping variable:

① Static scoping (compile time)

② Dynamic scoping (runtime)

NOTE

↳ Scoping rules are applicable only to the free variables.

* Free variable: The variable which is not declared in the function but used in the function is called as free variable.

Static Scoping:

→ In the static scoping, the free variables are resolved by using global variables.

Dynamic Scoping:

→ In the dynamic scoping, free variables are resolved by using from previous calling function.

NOTE

↳ In the worst case, dynamic scope behaves like static scoping, if no variables is present in the previous calling function, then, global variable will be use (printed).

→ C language by default follows static scoping.

→ The precedence of the local variable is higher than the global variable.

Eg: ① int a = 10; D & A

```
main()          A()
{ int a; D      { int b; D
  A();            b=20; A
}               }
```

② int a; D

```
main()          A()
{ a=10; A       { int b=20; D&A
  A();            a=30; A
}               }
```

Pseudo code

③ a=10; → D&A

```
main()          A()
{               {
  a=20; D&A       a=30; D&A
  A();            b=40; D&A
}               }
```

Declaration → D

Assignment → A

Eg-① int a=50, b=60;

```
main()                    swap()
{                         { int t;
  int a,b;                  t=a;
  a=10, b=20;               a=b;
  swap();                   b=t;
  pf(a,b);                  pf(a,b);
}                         }
```

Static Scoping	Dynamic Scoping
① 60, 50	① 20, 10
② 10, 20	② 20, 10

Eg-2 int a=5;

```
main()          B()          C()          D()
{ int a=10; ①   { int a=20; ② { int a=30; ③ { int a=40; ④
  B();            C();         D();          printf(a);
}               }            }             }
```

Static	Dynamic
5	① 30 ② 20 ③ 10 ④ 5

Worst case → Dynamic = Static

①

```
variable x : real
procedure show()
Begin
    printf(x);
end
procedure small()
Begin
    variable x:real;
        x=0.125;
    show();
end
begin
    x=0.25;
    show();
    small();
end
```

	Static	Dynamic
1)	0.25	① 0.25
②	0.25	② 0.125

②

```
var x,y : integer
procedure p(var n integer)
Begin
    x = (n+2)(n-3);
end
procedure Q()
Begin
    var x,y : integer
    x=3;
    y=4;
    p(y);
    pf(x);
end
Begin
    x=7 ; y=8;
    Q();
    pf(x);
end
```

	Static	Dynamic
	3, 6	6, 7

②

```
int a=1, b=2;
main()
{
  int a=20, b=30;
  pf(a,b);
  C();
  pf(a,b);
  D();
}

D()
{
  pf(a,b);
  a=3, b=4;
  pf(a,b);
  E();
}

C()
{
  int a=50;
  pf(a,b);
  D();
  pf(a,b);
}

E()
{
  int ~~a=10~~; b=6;
  pf(a,b);
  a=7, b=8;
}
```

Static	Dynamic
① 20, 30	① 20, 30
② 50, 2	② 50, 30
③ 1, 2	③ 50, 30
④ 3, 4	④ 3, 4
⑤ 3, 6	⑤ 3, 6
⑥ 50, 4	⑥ 7, 4
⑦ 20, 30	⑦ 20, 4
⑧ 7, 4	⑧ 20, 4
⑨ 3, 4	⑨ 3, 4
⑩ 3, 6	⑩ 3, 6

③

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

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

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

E(int b, int a)
{ pf(a,b);
  a=b+a;
  a=b-a;
  a=7, b=8;
  pf(a,b);
}
```

Static	Dynamic
① 3, 0	① 3, 0
② 0, 0	② 3, 0
③ 4, 5	③ 4, 5
④ 6, 4	④ 6, 4
⑤ 7, 8	⑤ 7, 8
⑥ 4, 6	⑥ 4, 6
⑦ 4, 5	⑦ 4, 5
⑧ 3, 5	⑧ 4, 5
⑨ 6, 4	⑨ 6, 4
⑩ 7, 8	⑩ 7, 8
⑪ 4, 6	⑪ 4, 6

POINTERS:

→ Pointer is a variable generally used to store the address.
→ The two different operative are mainly used in the pointers:
① (&) → Address of operator
② (*) → Value at location
→ Dereference operator
→ object at location / indirection operator.

E.g.: int *ptr; ⇒ ptr is a pointer variable of type integer where we can store address of integer variable.
(or) Integer pointer

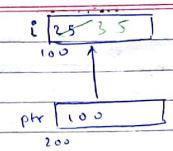
NOTE
↳ ① Address cannot be negative.
② Pointer variable takes 2 B of memory [It can be any type of pointer], but
③ It may change from system to system, depending on the O.S.
③ Pointer can dereference number of bytes depending on its data type.
④ Integer pointer can dereference or access 2 B, character pointer can access only 1B, float pointer can access 4B.

Contents.
→ Basics
→ Pointer to pointer
→ Pointer to arrays
→ Pointer to strings
→ Array of pointers to array
→ Array of pointers to strings
→ Pointers to structures
→ Pointers to function.
→ Multi dimensional arrays.

End

Basics:

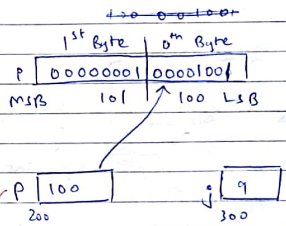
```
main()
{
  int i;
  int *ptr;
  i=25;
  ptr = &i;
  pf("%u %u", &i, ptr);   100, 100
  pf("%d %d", i, *ptr);   25, 25
  *ptr=35;
  pf("%d %d", &i, ptr);   100, 100
  pf("%d %d", i, *ptr);   35, 35
}
```



Pointer Conversion:

```
main()
{
1. int i=265, j;
2. char *p;
3. p=&i;
4. j=*p;
5. printf("%d", j);
}
```

o/p ⇒ 9



char pointer can dereference only 1B.

→ In the above program, we are storing the address of integer variable in the character pointer.
→ At the compilation time, we get the warning message saying "suspicious pointer conversion".

→ While accessing the value by using character pointer, it can dereference ~~only one~~ (access) only 1 Byte.

→ To overcome this drawback, we need to convert character pointer into integer pointer by applying typecasting.

→ In order to perform the type casting, we need to replace the statement (4) with:

```
j = *(int *)p;
```

```
j = *p
    (int *)
```

→ Now, if is temporarily converted as an integer pointer and it can dereference 2 Bytes, then, the o/p will be 265.

Parameter Passing Techniques:

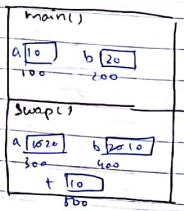
① Call by value
② Call by address (reference)

Call by value:

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

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

o/p → 10, 20



NOTE

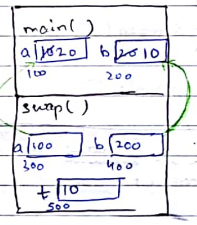
↳ The formal parameters which are changed in the swap function, will not reflect the actual parameters in the main function.

Call by address (reference):

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

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

o/p: 20, 10



NOTE

↳ In the question, if it is mentioned like they are using call by reference, then, implicitly we need to understand, the address is being passed and receivers are the pointer variables and actual parameters will get reflected [implicitly we need to assume *, & being used]

2 marks
GATE 2016

Q.

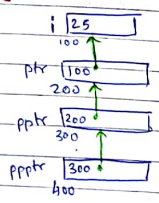
```
a = 3;
void n(x)
{
  x = x * a;
  printf(x);
}
void m(y)
{
  a = 1;
  a = y - a;
  n(a);
  printf(a);
}
void main()
{
  m(a);
}
```

What is the output of the following program, when parameters are passed by reference and dynamic scope is assumed?

(a) 6, 2
(b) 6, 6
(c) 4, 6
(d) 4, 4

Pointer to Pointer:

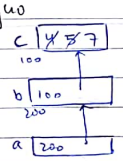
```
void main()
{
  int i;
  int *ptr; // it can dereference upto 1 level
  int **pptr; // It is called as double pointer/pointer to pointer, can access upto 2 levels
  int ***ppptr; // triple pointer/pointer to pointer to pointer
                // can dereference upto 3 levels
  i = 25;
  ptr = &i;
  pptr = &ptr;
  ppptr = &pptr;
}
```



print		o/p
① i	→	25
② &i	→	100
③ ptr	→	100
④ *ptr	→	25
⑤ &ptr	→	300
⑥ **pptr	→	25
⑦ *pptr	→	100
⑧ *ppptr	→	200
⑨ **ppptr	→	100
⑩ ***ppptr	→	25
⑪ &pptr	→	400
⑫ ****pppptr	→	error / garbage value
⑬ &pptr	→	300

W.B ngwo

Q37.



x = 4 7

**ppz = **ppz + 1
z = 5

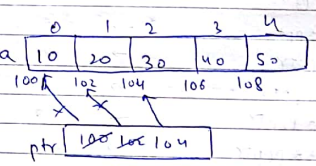
*py = *py + 2
y = 7

output = 7 + 5 + 7
= 19

Pointer to array:

→ Array name without subscript [ ] will give the base address of the array.
→ This base address will be constant and it cannot be changed, so, array name is called as constant pointer.

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



o/p: 30

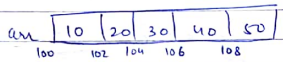
NOTE

→ In the above example, ptr, a both are pointers to array but 'ptr' is a pointer variable but 'a' is a pointer constant.

++ptr ✓	++a ✗
ptr++ ✓	a++ ✗

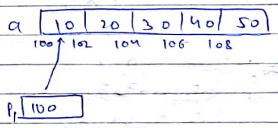
Pointer Arithmetic:

```
void main()
{
int arr[5] = {10,20,30,40,50};
pf("%d %d %d %d", arr[2], *(arr+2), *(2+arr), 2[arr]);
}
```



arr[2]	arr[2] = 2[arr]
=> *(arr+2) = *(2+arr)	
=> *(100+2) = *(2+100)	
=> *(104) = *(104)	
= 30 = 30	

Eg.
```
void main()
{
int a[5], = {10,20,30,40,50};
int *p1, *p2;
p1 = a;
p2 = a;
}
```



NOTE

→ We can add integer constant to the pointer.
Eg p_1+3 means skipping 3 elements in the forward direction.
→ We can subtract integer constant to the pointer.
E.g. p_1-3 means skipping 3 elements in the backward direction.
→ We can subtract 2 pointer variables.
E.g. p_1-p_2 means number of elements present from p_2 to p_1.
• This is allowed only when both the pointers are pointing to

same array and P_1 is greater than P_2.

Eg: $P_1 - P_2$

$P_1 = 106$ $P_2 = 100$

$P_1 - P_2 = \frac{106 - 100}{2} = 6 \Rightarrow$ (3) → no of elements.

(2) → size of each element

(4) → We cannot add, multiply, and divide between 2 pointers, because it has no meaning.

$P_1 + P_2$ X

$P_1 * P_2$ X

P_1 / P_2 X

(5) → We cannot add floating point number to a pointer variable because it has no meaning

$P_1 + 2.5$ X

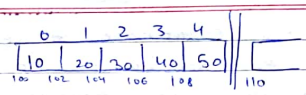
E.g.
```
void main()
{ int a[5] = {10, 20, 30, 40, 50};
  int *p1, *p2;
  p1 = &a[0];
  p2 = &a[4];
  printf("%d %d %d", p2-p1, p1+2, *p2);
}
```

O/p: $P_2 - P_1 = 108 - 100 = \frac{8}{2} = 4$

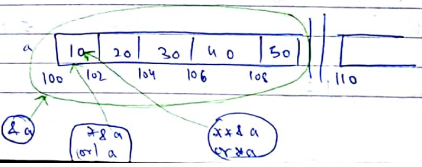
$P_1 + 2 = 100 + 4 = 104$

$*P_2 = 50$

E.g.
```
main()
{
  int a[5] = {10, 20, 30, 40, 50};
}
```



Expression	Print value	Type	Meaning
(1) &a	100	Pnt*[]	Base address of entire array.
(2) *&a	100	Int*	Base address of 0th element of the array
(3) a	100	int*	Base address of 0th element of the array.
(4) *a	10	Int	Value of the 0th element of the array.
(5) **&a	10	int	Value of the 0th element of the array.
(6) a+1	102	int*	Base address of the 1st element of the array.
(7) &a+1	110	int*[]	Base address of the next continuous array.
(8) **a	Garbage value or error	int or ?	Referring outside the array memory.
(9) *a+17	27	Int	Adding 17 to 0th element value.
(10) a+3	106	int*	Base address of 3rd elem. of the array



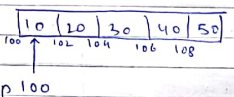
NOTE

→ In the C language, if we want to pass an entire array to the function, then it is not a good idea to pass all the array elements.

- Instead, by passing the base address of the array, we can access all the elements of the array.

Q.1

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



p[0]	p[1]	p[2]
= *(p+0)	= *(p+1)	= *(p+2)
= *(100)	= *(102)	= *(104)
= 10	= 20	= 30

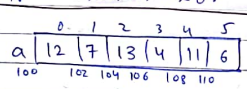
o/p = 10,20,30,40,50.

GATE 2010 2 marks

Q.

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

What is the o/p printed? 15



f(a,6)

12 + f(a[1],5) → 15

7 - f(a[2],4) → 3

13 - f(a[3],3) → 4

4 + f(a[4],2) → 9

11 - f(a[5],1) → 5

6 + f(a[6],0) → 6

0

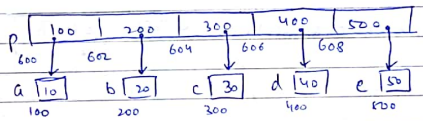
Array of pointers:

① int a[5]; → array of 5 integers

② int *a[5]; → a is an array of 5 integer pointers.

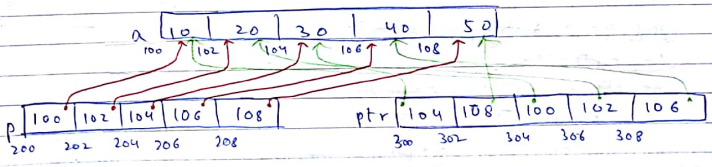
③ E.g.

```
main()
{
  int a=10, b=20, c=30, d=40, e=50;
  int *p[5] = {&a, &b, &c, &d, &e};
}
```



④

```
main()
{
  int a[5] = {10,20,30,40,50};
  int *p[5] = {a, a+1, a+2, a+3, a+4};
  int *ptr[5] = {a+2, a+4, a, a+1, a+3};
```



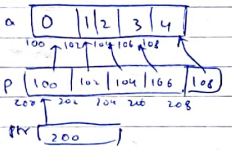
NOTE

→ In any operator's associativity is ~~any~~ right to left.

Q1.

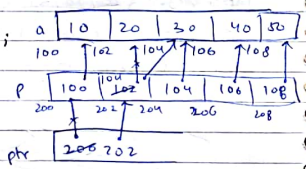
```
main()
{
int a[5]={0,1,2,3,4};
int *p[5]={a, a+1, a+2, a+3, a+4};
int **ptr=p;
ptr++;
pf(ptr-p, *ptr-a, **ptr);
*ptr++;
pf(ptr-p, *ptr-a, **ptr);
*++ptr;
pf(ptr-p, *ptr-a, **ptr);
++*ptr;
pf(ptr-p, *ptr-a, **ptr);
}
```

o/p
1, 1, 1
2, 2, 2
3, 3, 3
3, 4, 4



Q2.

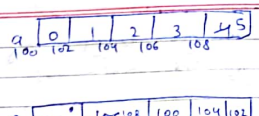
```
main()
{ int a[5]={10,20,30,40,50};
int *p[5]={a, a+1, a+2, a+3, a+4};
int **ptr=p;
**ptr++;
pf(ptr-p, *ptr-a, **ptr);
*++*ptr;
pf(ptr-p, *ptr-a, **ptr);
++**ptr;
pf(ptr-p, *ptr-a, **ptr);
}
```



o/p ① 1, 1, 20
② 1, 1, 30
③ 1, 1, 31

Q3.

```
main()
{
int a[5]={0,1,2,3,4};
int *p[5]={a+4, a+3, a, a+2, a+1};
int **ptr=p;
**ptr++;
pf(ptr-p, *ptr-a, **ptr);
*++*ptr;
pf(ptr-p, *ptr-a, **ptr);
++**ptr;
pf(ptr-p, *ptr-a, **ptr);
}
```



ptr 200 202

① 1, 3, 3
② 1, 4, 4
③ 1, 4, 5

2-Dimensional Arrays:

NOTE
↳ Array name without subscript always gives constant base address of the first element (0th element).

First Element

→ 1-D array: means base address of the normal element.
Eg. int a[5];
a ⇒ int *;

→ 2-D array: means the base address of first 1-D array.
Eg. int a[5][6];
a ⇒ int *[];

→ 3-D array: means the base address of first 2-D array.
Eg. int a[3][3][3];
a ⇒ int *[][];

#2-D Array:

→ The good example of 2-D array is pointer to pointer.
Eg. int a[i][j];

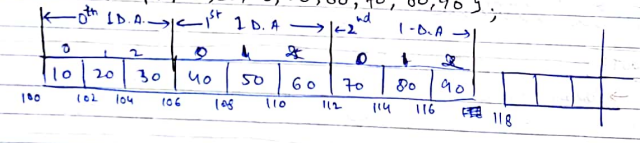
→ The meaning of the above statement is that it has i- rows and j- columns.

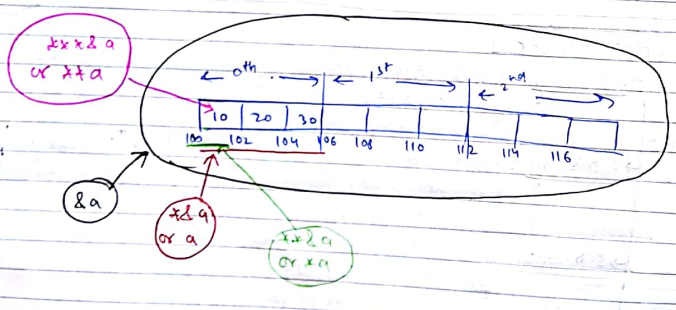
→ a[i] → represents constant base address of i^{th} row.

→ Eg a[i] = 2000;
↳ The above statement is not allowed and it gives error because we are trying to change the constant base address of i^{th} row.

→ The 2-D array will be stored in the memory in the form of multiple 1-D arrays.

int a[3][3] = {10, 20, 30, 40, 50, 60, 70, 80, 90};





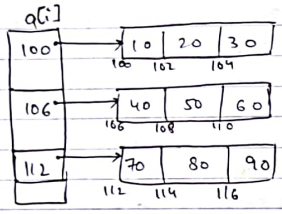
a[0] = *(a+0) = *a	a[i][j] ⇒ *(*(a+1)+2) ⇒ *(*(100+1)+2) ⇒ *(*(106)+2) ⇒ *(106+2)	⇒ *(110) ⇒ 60

	Expression	Print value	Type	Meaning
①	&a	100	int *[][]	Base address of the entire 2-D array
②	*&a	100	int *[]	Base address of 0th 1-D array.
③	a	100	int *[]	Base address of 0th 1-D array.
④	*a	100	int *	Base address of 0th element of 0th 1-D array.
⑤	a[0]	100	int *	Base address of 0th element of 0th 1-D array.
⑥	&a+1	118	int *[3][]	Base address of the next continuous 2-D array.
⑦	a+1	106	int *[]	Base address of the 1st 1-D array.
⑧	*a+1	102	int *	Base address of the 1st element of 0th 1-D array.
⑨	**a	10	int	Value of 0th element of 0th 1-D array.
⑩	*(a+2)+2	116	int *	Base address of the 2nd element of 2nd 1-D array.
⑪	*(*(a+1)+2)	60	int	Value of 2nd element of 1st 1-D array.
⑫	***a	Garbage value/err.	int or ?	Refering outside array memory.
⑬	***&a	10	int	Value of 0th element of 0th 1-D array.

H.W.- int a[4][5] = {0, 1, 2, ... 19}
print all (13) expressions.
B.A. = 100

a[2][1] → *(*(a+2)+1) → *(112+1) → *(114) = 80

int a[3][3] = {10, 20, ... 90};



GATE

```
main()
{
  int a[5][3]= {10, 20, 30, ..... 150};
  pf("%d", ((a==a[0]) && (a[0]== *a));
}
```

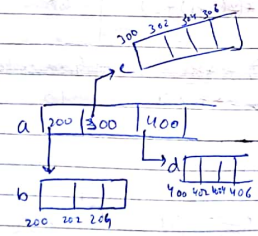
What is the o/p printed?

(a) 1
(b) 0
(c) Compilation error
(d) Runtime error

WB Pg 35.

Q9. int *A[10];
int B[10][10];

(I) A[2] = &x; ✓
(II) A[2][3]
(III) B[1] = 2002; ✗
(IV) B[2][3] = 10; ✓

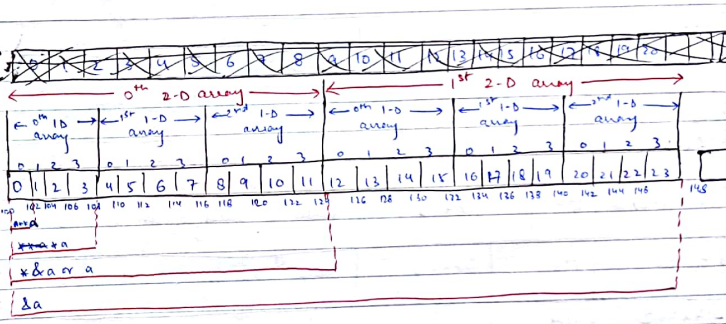


a[0]: b
a[1]: c
a[2]: d

#3-D Array:

→ In memory, 3-D array will be stored in the form of multiple 2-D arrays.
→ In the 3-D array, the array name without subscript will give the base address of 0th 2-D array.

int a[2][3][4] = {0, 1, 2, 3, ... 23}



Eg ① a[0][1][1]
→ *(*(*(a+0)+1)+1)
→ *(*((*a+1)+1))
→ *(*(*(a)+1)+1)
→ *(*(100+1)+1)
→ *(*(108)+1)
→ *(108+1)
→ *110
= 5

② a[1][2][3]
→ *(*(*(a+1)+2)+3)
→ *(*(124+2)+3)
→ *(*(140)+3)
→ *(140+3)
→ *(146)
→ 23

Expression	Printvalue	Type	Meaning
① &a	100	int *[][]	Base address of the entire 3-D array.
② a (or) *&a	100	int *[]	Base address of the 0th 2-D array.
③ *a	100	int *[]	Base address of 0th 1-D array of 0th 2-D array.
④ **a	100	int *	Base address of 0th element of 0th 1-D array of 0th 2-D array.
⑤ ***a	0	int	Value of 0th element of 0th 1-D array of 0th 2-D array.
⑥ ****a	G.V./error	int or ?	Refering outside array memory.
⑦ ***&a	100	int *	0th element of 0th 1-D array of 0th 2-D array → Base address of.
⑧ a+1	124	int *[][]	Base address of 1st 2-D array.
⑨ &a+1	148	int *[][]	Base address of next continuous 3-D array
⑩ *(a+1)+2	140	int *[]	Base address of the 2nd 1-D array of 1st 2-D array
⑪ *a+2	116	int *[]	Base address of the 2nd 1-D array of 0th 2-D array.
⑫ **a+2	104	int *	Base address of 2nd element of 0th 1-D array of 0th 2-D array.
⑬ *(*(a+1)+1)+2	136	int *	Base address of 2nd element of 1st 1-D array of 1st 2-D array.
⑭ **(a+1)+2	128	int *	Base address of 2nd element of 0th 1-D array of 1st 2-D array.

HW.

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

B.A := 100

Print all 14 expressions.

ASCII: a → 97, A → 65

Strings:

→ Character array is called as string.

→ String always ends with NULL character

→ NULL character representation is '\0' and ASCII value is 0.

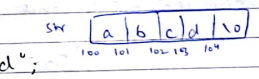
→ String name always gives base address and that will be constant.

→ Every character of a string is stored in the memory sequentially in contiguous locations in the form of their ASCII values.

Eg ① void main()
{
char str[20] = {'a', 'b', 'c', 'd'};
printf("%c %c %c %c", str[0], str[1], str[2], str[3]);
}

o/p ⇒ a b c d

② void main()
{
int str[20] = {"abcd"};
printf("%s", str);
}



NOTE

↳ ① %c will print character.

② %d will print ASCII value of the respective character.

*③ If you are reading str[0] in any expression while solving the problem, we need to take the respective character's ASCII value.

④ %s will take the base address and from that particular base address it will print all the characters until the NULL character.

INITIALISATION OF STRING:

1. char s[6] = "INDIA"; // Explicitly we need to allocate size for NULL char.
2. char s[] = "INDIA";
3. char s[6] = {'I', 'N', 'D', 'I', 'A', '\0'};
4. char *p = "INDIA";

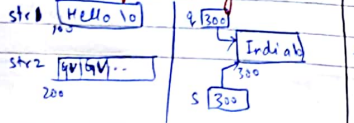
string constant

The meaning of above statement is, we are requesting the compiler to allocate the memory for the string INDIA and the base address of that will be stored in the pointer variable "p".

→ The above statement ④ is considered as a string constant.
→ Contents of the string [1,2,3] can be modified but the contents of string constant[4] cannot be modified.

Eg-① main()
{
char str1[] = "Hello";
char str2[20];
char *s = "India";
char *q;
str2 = str1; //error
q = s; //All-valid
}

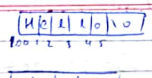
→ It will give error because we cannot modify the constant base address of the string.



Eg-② main()
{
char str1[10] = "Hello";
char str2[10] = "Hello";
if (str1 == str2)
pf("strings are equal");
else
pf("They are different");
}

o/p: They are different

Eg-③ main()
{
if (*"Hello" == *"Hai")
100 200
pf("They are same");
else
pf("They are different");
}



"Hello" gives base address of string
*(100) = H
*(200) = H

o/p: They are same
* In the above example it will just compare 1st character of both the strings, i.e., 'H'. Because they are same, the o/p is "They are same."
* It is not comparing the remaining characters of the string.

General functions in <string.h>

① strlen(str) = L
→ String length always gives the actual length of the string.
e.g. char *str = "India";
strlen(str) = 5

② strcpy(target, source);
③ strupr(str);
→ It converts all the letters to uppercase.
④ strlwr(str);
→ Converts all letters to lowercase.
⑤ str rev(str);
→ Reverses the given string.

Eg ① void main()

```
{
  char str[20] = "abcdef";
  pf("%s", str);      //100
  pf("%s", str+3);    //103
  str[3] = 97;
  pf("%s", str);
  str[3] = 0;
  pf("%s", str);
  str[3] = '0';
  pf("%s", str);
}
```

0	1	2	3	4	5	6
a	b	c	d	e	f	\0
100	101	102	103	104	105	106

① a b c d e f

② d e f

97	98	99	97	101	102	0

③ a b c a e f

97	98	99	0	101	102	0

④ a b c

97	98	99	48	101	102	0

⑤ a b c 0 e f

GATE2010

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

0	1	2	3	4	5	6
s	t	r	i	n	g	\0
100	1	2	3	4	5	6

length = 6

p	\0	g	n	i	r	t	s

s[length-i]

s[6]

No o/p

what is the o/p printed ?

(a) gnirts
(b) nirts
(c) string
(d) No output

GATE2011. char *p = "GATE2011";
printf("%s", p + p[3] - p[1]);

(a) GATE2011
(b) E 2011
(c) 2011
(d) 011

p	G	A	T	E	2	0	1	1
100	1	2	3	4	5	6	7	

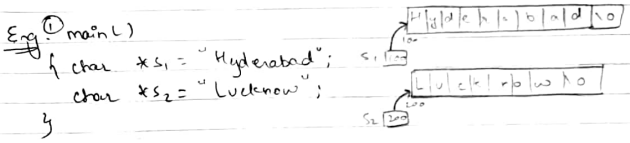
p 100

E A
69 - 65 = 4

Array of Pointers to Strings:

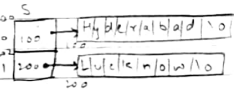
Eg ① main()

```
{ char *s1 = "Hyderabad";
  char *s2 = "Lucknow";
}
```



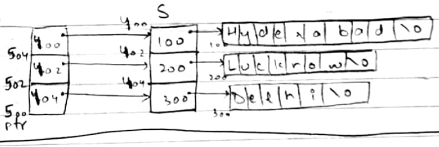
Eg ① main()

```
{
  char *s[2] = {"Hyderabad", "Lucknow"};
}
```



Eg. main()

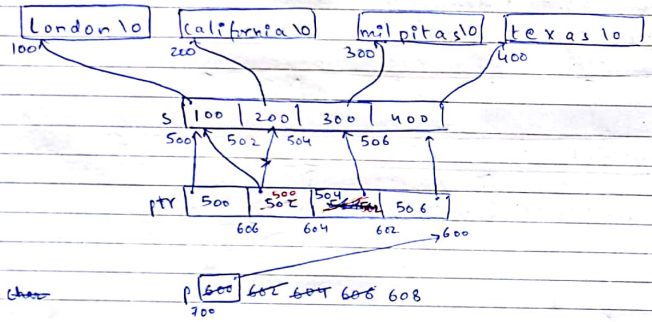
```
{ char *s[3] = {"Hyderabad", "Lucknow", "Delhi"};
  char **ptr[3] = {s+2, s+1, s};
}
```



Q1. main()

```
{ char *s[] = {"london", "california", "milpitas", "texas"};
  char **ptr[] = {s+3, s+2, s+1, s};
  char ***p = ptr;
  pf("%s", ** +1p);      //300, milpitas
  Pf("%s", *--*++p+3);   //100+3=103, don
  pf("%s", *(p[-2]+3));  //403, as
  pf("%s", p[-1][-1]+1); //201, alifornia
  Pf("%s", **p++);       //100, London
```

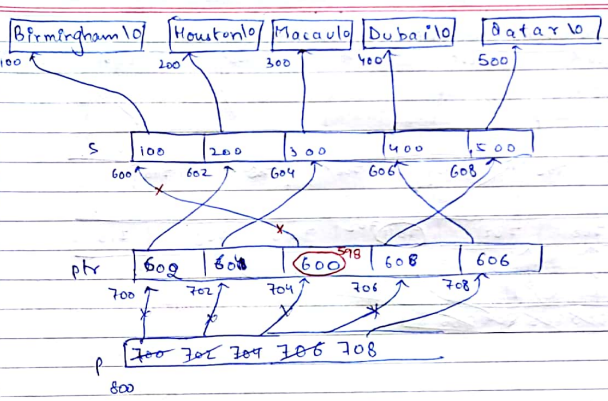
pf("%c", ***++p + 2);
}



① milpitas
② don don
③ as
④ alifornia
⑤ london **(604) = 100
⑥ ~~Garbage value~~ Garbage value

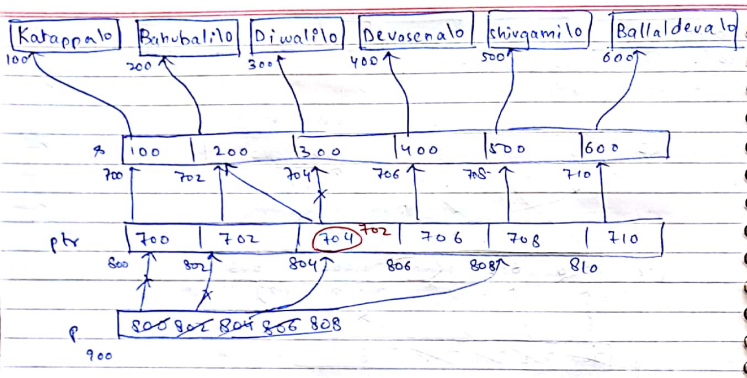
③ *(p[-2]) + 3
» *(*(p-2)) + 3
» *(604-2) + 3
» *600 + 3
» 400 + 3 = 403

main()
{ char *s[] = {"Birmingham", "Houston", "Macau", "Dubai", "Qatar"};
char **ptr[] = {s+1, s+2, s, s+4, s+3};
char ***p = ptr;
pf("%s", **++p); // 300, Macau
pf("%s", *--*++p+3); // q.v.
pf("%s", *(p[-2])+3); // 203, ston
pf("%s", p[-1][-1]+1); // 201, ouston
pf("%s", **p++); // q.v.
pf("%c", ***++p+2); // F
}



① Macau
② Garbage
③ (p[-2] + 3) = *(p-2) » ~~Garbage~~ / ston
④ p[-1][-1]+1 » *((*(p-1))-1) + 1 » ~~Garbage~~ / ouston
⑤ ~~Qatar~~ Garbage
⑥ F

main()
{ char *s[] = {"Katappa", "Bahubali", "Diwali", "Devasena", "Shivgami", "Balladeva"};
char **ptr[] = {s, s+1, s+2, s+3, s+4, s+5};
char ***p = ptr;
pf("%s", **++p);
pf("%s", *--*++p+3);
pf("%s", *(p[-2])+3);
pf("%s", p[-1][-1]+1);
pf("%s", **p++);
pf("%c", ***++p+2);
}



① Bahubali
② ubale
③ *(p[-2]+3) ⇒ *(*(p-2)+3) ⇒ ~~Devasena~~ appa
④ p[-1][-1]+1 ⇒ *(*(p-1)-1)+1 = atappa
⑤ Bahubali
⑥ U

Multidimensional Arrays on Strings:

Eg. main()
{
char mess[6][30] = {"Koi bhi lakshya",
"Bada nahi",
"jeesa wahi",
"jo dara nahi",
"Hara wahi",
"jo lada nahi"};
printf("%os", mess+3); // jo dara nahi
printf("%c", *(*(mess+2)+2)); // e
printf("%os", *(mess+4)); // Hara wahi
}

Q. WAP to implement string copy function.

```
void main()
{
char s2[20]="Made easy";
char s1[20];
strcopy(s1,s2);
printf("%os",s1);
}
void strcopy(*T,*s)
{
while(*s!='\0')
  { if(*T!='\0')
      { *T = *s;
        T++; s++;
      }
    else
      printf(" Error: Target smaller ");
  }
*T='\0';
}
```

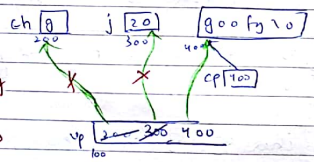
s2 [Made easy\0]

while(*T++ = *s++);

void pointers:

→ In the void pointer, we can store address of any type of variable
→ Void pointer does not know how many bytes it has to dereference (ix) access.
→ While accessing the data, we need to perform the respective typecasting and access,
→ Void pointers generally used in device driver programming.

```
E.g. main()
{
    void *vp; // declaring the void pointer.
    char ch = 'g';
    char *cp = "goofy";
    int j=20;
    vp = &ch;
    pf("%c", *(char*) vp); //g
    vp = &j;
    pf("%d", *(int *) vp); //20
    vp = cp;
    pf("%s", (char*) vp+3); //403, fy
}
```



Pointer to functions:

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

o/p ⇒ Hello

Meaning of ① ⇒ Declaring the function pointer.
→ p is a pointer to the function which does not take any arguments and return whose return type is void.
Meaning of ② ⇒ Assigning the base address of the test function to the pointer variable p.
Meaning of ③ ⇒ Calling the function using pointer variable p.

Terminology for various expressions:

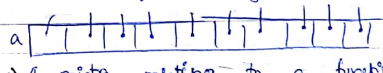
① void (*p)(); ⇒ 'p' is a pointer to a function which does not take any arguments and returns void.

② int (*p)(int,int); ⇒ 'p' is pointer to a function which takes 2 integer argument and returns integer.

GATE 2003
③ int (*p)(int*); ⇒ 'p' is a pointer to the function which takes integer pointer as argument and returns integer.

④ int a[10]; ⇒ 'a' is an array of 10 integers.

⑤ int *a[10] ⇒ 'a' is an array of pointers, where we can store addresses of 10 integers.



⑥ *(*)(*,*); ⇒ a pointer pointing to a function which takes 2 pointers as argument and returns pointer.

Arrays:

→ Array is a collection of similar type of elements, which is stored in contiguous memory locations.

→ Array supports random access.

1-D Arrays:

int a[0....9] = {10, 20, 30, ..., 100};

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100
100	102	104	106	108	110	112	114	116	118

① loc(a[7]) = 100 + (7 - 0)*2 (lower bound)
= 100 + 14 = 114

② loc(a[4]) = 100 + (4 - 0)*2
= 108

Q. a[24.....900], B.A. = 100, size of element = 5
loc(a[625]) = 100 + (625 - 24)*5
= 100 + 6015
= 3105

Q. a[-95.....+199], BA = 500, size of element = 8
loc[a[125]] = 500 + (25 + 95)*8
= 500 + 1760 = 2260

NOTE
↳ int a[lb.....ub], B.A., size of element = s.
loc(a[i]) = B.A. + [i - lb]*s

2-D Arrays:

→ 2-D arrays will be stored in the memory in the form of multiple 1-D arrays.

→ They will be stored in the 2 different ways:

① Row major order ② Column major order

NOTE
↳ C language by default follows row major order.

#Row Major order:

int a[1.....4][1.....5]
(lb1, ub1, lb2, ub2)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{bmatrix}$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
100	102	104	106	108	110	112	114	116	118	120	122	124	126	128	130	132	134	136	138

No. of rows = (4 - 1) + 1 = 4
No. of columns = (5 - 1) + 1 = 5
(ub1 - lb1 + 1, ub2 - lb2 + 1)

loc[a[4][3]] = 100 + ((4-1)*5 + (3-1))*2
= 100 + (15 + 2)*2
= 134

loc(a[3][5]) = 100 + ((3-1)*5 + (5-1))*2
= 100 + (10 + 4)*2
= 128

Q.1 a[75 ... 198] [-21 ... +155]

BA = 500 ; size of element = 6

loc(a[102][103]) = BA + [(102-75)* (155+22) + (103+22)]* 6

= 500 + (4779 + 124) x 6

= ~~29342~~ 29918

Q.2 a[-75 ..... +75] [-75 ..... +150]

BA = 100 size of element = 8

loc(a[-65][-10]) = 100 + [(-65 +75) x (150+76) + (-10+75)] * 8

~~= 100 + [10 x 66 + 65] x 6~~

~~= 100 + 0725 x 6~~

~~= 4450~~

= 100 [10 x 226 + 65] x 8

= 18700

NOTE

↳ a [lb1 ... ub1] [lb2 ..... ub2];

B.A. , size of element = s, R.M.O

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

Column Major order (C.M.O):

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

lb1 ub1 lb2 ub2

No. of rows = 4-1+1 = 4

No. of columns = 5-1+1 = 5

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

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}	a_{15}	a_{25}	a_{35}	a_{45}
100	102	104	106	108	110	112	114	116	118	120	122	124	126	128	130	132	134	136	138

loc(a[3][5]) = 100 + [(5 - 1) x 4 + (3 - 0)] x 2 (lb2, lb1)

= 100 + (16 + 2) x 2

= 136

loc(a[2][4]) = 100 + [(4-1) x 4 + (2-1)] x 2

= 100 + [12 + 1] x 2

= 126

Q① a[-55 ... +55] [-128, ... +128]

BA = 500 ; size of element = 6 C.M.O

loc(a[50][120]) = 500 + [(120+128) x 111 + (50 +55)] * 6

= 500 + [248 x 111 + 105] x 6

= 166298

② a[-76 ..... +52, -65 ... +37]

B.A. = 1000 , size of element = 8 C.M.O

loc(a[56][30]) = 1000 + [(30+65) * 129 + (50 +76)] * 8

= 1000 + [95 x 129 + 126] * 8

= 100048

NOTE

↳ a[lb1 ... ub1] [lb2 ... ub2]

B.A. =, size of element = s, C.M.O

loc(a[i][j]) = B.A. + [(j - lb2) * (ub1 - lb1 + 1) + (i - lb1)] * s

BA + [(j-lb2) * (ub1-lb1+1) + (i-lb1)] * s

LOWER TRIANGULAR MATRIX (LTM):

→ It should be a square matrix and only the elements which are $[i \geq j]$ will be stored and remaining elements will be 0.

Row major order

$a[1....4][1....4]$

lb_1 ub_1 lb_2 ub_2

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

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

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

$$= 100 + [6+2] \times 2$$

$$= 116$$

Q. $a[-25....+25][-25....+25]$ $\quad BA = 100$; size of element = 6.

$10 - -25 = (10--25)$

$$loc(a[10][8]) = 100 + \left[\frac{35 * 36}{2} + 8 + 25\right] \times 6$$

$$= 100 + [630 + 33] \times 6$$

$$= 4078$$

NOTE

$a[lb_1 \ldots ub_1][lb_2 \ldots ub_2]$

BA, size of element = s

$$loc[a[i][j]) = B.A. + \left[\frac{(i-lb_1)(i-lb_1+1)}{2} + (j-lb_2)\right] \times s$$

STRUCTURES

→ Structures are used to create the user defined datatypes.

→ In the structures, 2 operators are majorly used:

① Dot (.) ⇒ ~~Structure to member~~ Structure to member

② Arrow (⟶) ⇒ Pointer to member.

Important Points:

① Structure contains member variables.

② Size of the structure is sum of all member variables

③ We cannot define the structure without member variables.

④ We can define the structure inside the function.

⑤ We cannot define the function inside the structure.

⑥ We can create existing structure as member to other structure.

⑦ We can define the structure inside another structure. [It is called as container ship or nested structure]

⑧ We cannot create same structure variable as member to the structure because logically size is infinite.

⑨ We can create same structure pointer as member to the structure and this is called as self referential structure [This is used in the linked list].

E.g. `struct emp` (Keyword → struct, Name of structure → emp)

```
{
    int sal;
    char name[40];
    int id;
};
```

size = 44 B; member variables

```
typedef struct emp emp;
```

No need to write everytime

```
struct emp e1;
emp e1, e2;
```

structure variables.

#H.W Q. what is the difference between #define & typedef?

```
emp *ptr; ✓ //ptr is pointer variable of type struct emp
          //where we can store address of the structure
          // variable
e1.sal = 2000; ✓
e1.name = Ramesh ; ✗
e1.name = "Ramesh"; ✗ // we cannot assign the base address of
                      // the string in the normal variable.
strcpy(e1.name,"Ramesh");
ptr = &e2 ; ✓ // Assigning the base address of structure
              // variable e2 to ptr
e2.id = 1000; ✓
ptr -> sal = 2000; ✓ // member variable "sal" in the structure
                     //variable e2 will be updated with 2000.
e1 = e2; ✓ // this statement is allowed but not suggestable
           // to use in case if there are any pointer
           //variables in the structure.
e1/e2; ✗ // Not allowed.
e1*e2; ✗
e1+e2; ✗
e1-e2; ✗
```

Q.1.
```
void main()
{
  struct emp;
  {
    int sal;
    int id;
  };
  typedef struct emp EMP;
}
void me()
{
  EMP e1,e2,e3; // error
}
```

* Structure is defined inside the main and we are creating structure variable in outside function, so, it is not allowed.
* If we want to create structure variable in outside func", then we need to create the structure outside the main().

Q2.
```
struct emp
{
  int sal;
  int id;
  struct emp e1; ✗
};
```

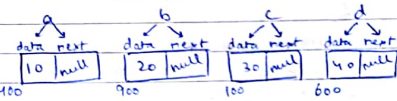
Invalid ⇒ refer important points (8).
we cannot create the same structure variable as member to the structure because logically size is infinite.

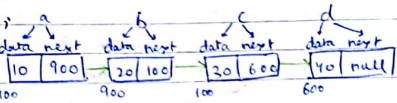
Q3.
```
struct node
{
  int data;
  struct node *next; ✓
};
```

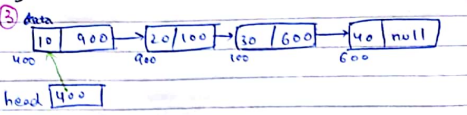
* We can create same structure pointer as member to that structure and this is called as self referential structure.

*next is a pointer variable of type structure node, where we can store address of any structure variable.

Eg.
```
struct node
{ int data;
  struct node *next;
};
typedef struct node node;
① node a = {10,null};
  node b = {20,null};
  node c = {30,null};
  node d = {40,null};
② a.next = &b;
  b.next = &c;
  c.next = &d;
③ node *head;
  head = &a;
```







UNIONS:

→ The size of the union is maximum size of member variable.
→ The same memory space will be shared by all the member variables.
→ As we know, all the members of structures can be accessed at any time but only one member of the union can be accessed at ~~one~~ a time and all other members contain garbage value.

Eg:
```
union data
{
  int i;
  float f;
  char str[20];
};

main()
{
  union data d;
  d.i = 10;
  d.f = 250.5;
  strcpy(d.str, "mode easy lucknow");
  pf("%d", d.i); //G.V
  pf("%f", d.f); //G.V.
  pf("%s", d.str); //mode easy lucknow
}
```

mode easy lucknow

Dynamic Memory Allocation (D.M.A.):

→ Array follows static memory allocation and we should know the requirement of the size prior to the declaration.
→ It is not possible to increase the size dynamically.
→ To overcome this problem, dynamic memory allocation will be used.
→ By using dynamic memory allocation, we can allocate the memory dynamically, how much we want and whenever we want.
→ The header file used to allocate dynamic memory allocation is `#include <alloc.h>`
→ Once the usage of dynamically allocated memory is finished, then, we need to deallocate the memory.

NOTE
↳ If the memory is allocated dynamically, and if it is not freed, then, it is called as memory leakage [memory wastage].

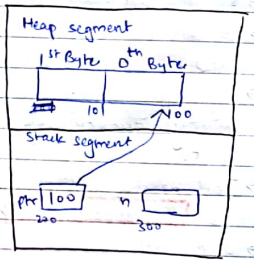
→ There are various functions used in the D.M.A:
① malloc(); ② calloc(); ③ realloc(); ④ free

* malloc():
① Dynamically allocates the memory in Bytes.
② It takes 1 argument, i.e., in Bytes and return type is void pointer.
Syntax:
`pointer_variable = (typecast *) malloc (size of (size));`
(typecast * : respective typecasting; size of : operator (not a function); size : in Bytes)

③ The memory allocated by using malloc() will be allocated in the heap segment of the memory.
④ The lifetime of this memory is entire program.

⑤ sizeof() operator estimates the size in Bytes depending on the O.S.

E.g: // int n;
int *ptr;
ptr = (int*) malloc (sizeof (int));



* malloc() dynamically allocates the memory in the heap segment and returns its base address (void pointer).
* Because it is returning base address, receiver should be pointer variable.
* malloc() doesn't know to ~~the~~ whom it is allocating the memory, it simply allocates the memory in the form of Bytes and returns void pointer.
* This is the reason, while accessing the memory, respective typecasting is required.
* Whatever the memory allocated in the stack segment can be accessed by using variable name and pointers but whatever the memory allocated in the heap segment, can be accessed only with the help of pointers.

*free():
① free() function deallocates the respective block of m/m.
② It takes 1 argument, i.e., the base address.
E.g: free(ptr);

DATA STRUCTURES

→ The logical and mathematical representation of data in the computer memory is called as data structure.
→ Data structure mainly ~~deals~~ deals with organising the data items based on their relationship.
→ Data structure mainly deals with the:
① How efficiently the data can be organised in the computer.
② How efficiently, data can be retrieved and manipulated (operations)

Program = Algorithm + Data Structure

Data Structure = Organised Data + Operations.

→ Data Structures are classified into 2 types:

① Linear
- → Array
- → Linked list
- → Stack
- → Queue

[Relationship between data items is linear (sequential)]

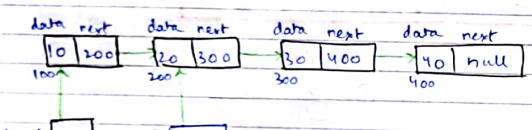
② Non-Linear
- → Trees
- → Graphs

[Relationship between data items is non-linear (hierarchical or parent child)]

Abstract Data Types (ADT):
→ In order to simplify the process of solving problems, data structures combined along with their operations is called as ADT.
→ ADT deals with:
① Declaration of data
② Declaration of operations

Linked List:

→ Linked list is a data structure used to store collection of data items.
→ Successive elements are connected by pointers and last element contains NULL.
→ We can increase the size of the linked list dynamically until the memory is full.
→ If the frequent insertions and deletions take place in our requirement, then, linked list is best suited.
→ Linked list supports only sequential access, it does not support random access.



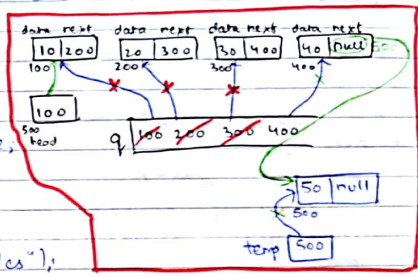

```
struct node
{ int ~~node~~ data;
  struct node *next;
};
typedef struct node node;
```

* Some useful statements:

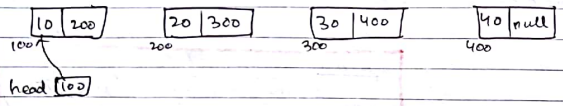
① If (head == null) ⇒ If it is TRUE, it means linked list is empty.
② If (head → next == null) ⇒ If it is TRUE, it means linked list is having only 1 element.
③ head → next; ⇒ 200
④ head = head → next; ⇒ Moving the head to the next node.
⑤ x = head → data; ⇒ x = 10
⑥ p = head → next; ⇒ p must be pointer variable of type "struct node". p is pointing to the second node of the linked list.

Q. WAP to create the linked list with "n" nodes.

```
#include <stdio.h>
#include <alloc.h>
void createnode();
struct node
{ int data;
  struct node *next;
};
typedef struct node node;
node *head = NULL;
main()
{ int n,i;
  printf("Enter no. of nodes");
  scanf("%d", &n);
  for(i=0; i<n; i++)
    createnode();
}
createnode()
{ node *temp,*q;
  temp = (node*) malloc(sizeof(node));
  temp->next = NULL;
  printf("Enter data item");
  scanf("%d", &temp->data);
  if(head == NULL)
  { head = temp;
  }
  else
  { q = head;
    while(q->next != NULL)
    { q = q->next;
    }
    q->next = temp;
  } //q = NULL;
  // temp = NULL;
}
```



Q. WAP to display all the data items of the linked list.




```
void display (node *head)
{ node *q;
  if (head == NULL)
    { pf ("List is empty");
    }
  else
    { q = head;
      while (q != NULL)
      {
        printf ("%d", q->data);
        q = q->next;
      }
    }
}
```

Q. WAP to count number of elements in the given linked list.

```
int listlength (node *head)
{
  node *q;
  int length = 0;
  q = head;
  while (q != NULL)
  { length++;
    q = q->next;
  }
  return length;
}
```

Q. WAP to insert new node in the beginning of the linked list.

```
void InsertAtBegin (node *head)
{ node *temp;
  temp = (node*) malloc (sizeof (node));
  pf ("Enter data item");
  temp->next = NULL;
  scanf ("%d", &temp->data);
  temp->next = NULL;
  if (head == NULL)
    { head = temp;
    }
  else
    { temp->next = head;
      head = temp;
    }
  //temp = NULL;
}
```

Q. WAP to insert new node at the end of the list.

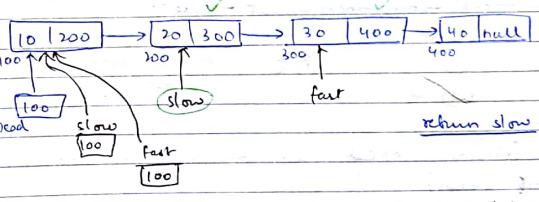
```
void InsertAtEnd (node *head)
{ node *temp, *q;
  temp = (node*) malloc (sizeof(node));
  temp->next = NULL;
  pf ("Enter data item");
  sf ("%d", &temp->data);
  if (head == NULL)
      head = temp;
  else
    { q = head;
      while (q->next != NULL)
        q = q->next;
      q->next = temp;
    } //q = NULL;
    //temp = NULL;
```

Q. WAP to insert a new node at a given position.

```
void insertAtPos(node *head, int pos)
{ node *q, *temp;
  int length, k=1;
  length = listlength(head);
  if(pos <0 || pos >length+1)
  { pf("Invalid position");
  }
  else if (pos==1)
  { insertAt begin (head);
  }
    else if (pos==length+1)
    { insertAtEnd (head);
    }
    else
    { temp=(node*) malloc(sizeof(node));
      temp->next= NULL;
      pf("Enter data item");
      scanf("%od", &temp->data);
      q= head;
      while(k<pos-1)
      {k++;
       q=q->next;
      }
      temp->next= q->next;
       q->next= temp;
    }
}
```

Q. WAP to find middle of the linked list.

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



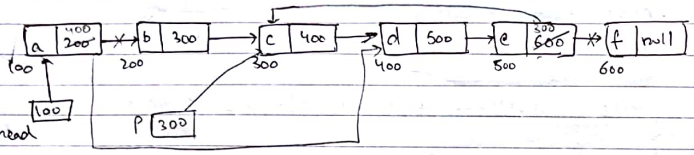
Q. WAP to find the middle of the linked list using list length.

```
node* middle(node *head)
{node *q;
 int length;
 length= listlength(head);
 if(head==NULL)
    return NULL;
 length=(length+1)/2;
 q=head;
 while(length!=1)
 { q=q->next;
   length--;
 }return q;
```

```
node* midle(node *head)
{node *p;
 int length, i=0;
 p=head;
 length= listlength(head);
 while(i<length/2)
 { p=p->next;
   i++;
 }
 return p;
}
```

GATE.

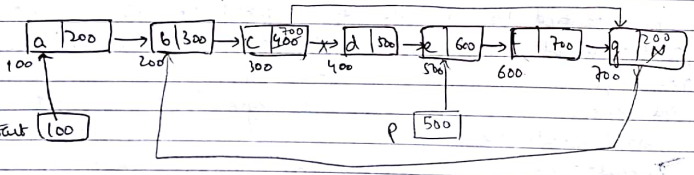
Q. Consider the following linked list with the given statements. What is the output printed?




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

(a) a (b) b (c) c (d) d (e) e

Q. Consider the following linked list with the given statements. What is the o/p printed?

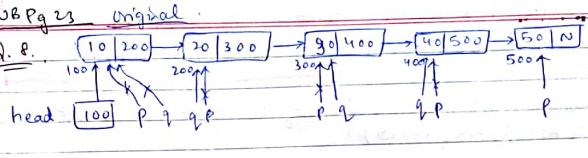



```
struct node *p;
p = start -> next -> next -> next -> next;
start -> next -> next -> next = p -> next -> next;
p -> next -> next -> next = start -> next
pf("%c", p -> next -> next -> next -> next -> next -> next -> next -> data);
```

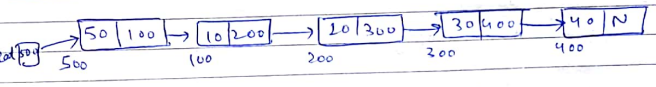
(a) a (b) b (c) c (d) d (e) e

WB Pg 23 original

Q. 8.



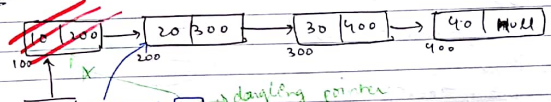
Modified:




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

(d)

Q WAP to delete the first node of the given linked list.




```
void deleteAtBegin (node *head)
{ node *q;
  if (head == NULL)
  { pf("List is Empty");
  }
  else
  { q = head;
    head = head -> next;
    free(q);
    q = NULL;
  }
}
```

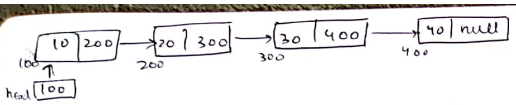
* Dangling Pointer : A pointer pointing to a location which does not exist in the memory.

Q. WAP to delete the last node of the given linked list.

```
void deleteAtEnd(node *head)
{ node *p, *q;
  if (head == NULL)
  { pf("List is empty");
  }
  else if (head->next == NULL)
       { free(head);
         head = NULL;
       }
  else
  { q = head;
    while (q->next != NULL)
    { p = q;
      q = q->next;
    }
    p->next = NULL;
    free(q);
    q = NULL;
  }
}
```

Q. WAP to delete the node at a given position.

```
void deleteAtPos(node *head, int pos)
{
  node *p, *q;
  int length = 0, k = 1;
  length = listlength(head);
  if (pos <= 0 || pos > length)
  { pf("Invalid position");
  }
  else if (pos == 1)
  { deleteAtBegin(head);
  }
  else if (pos == length)
  { deleteAtEnd(head);
  }
  else
  { p = head;
    while (k < pos)
    { q = p;
      p = p->next;
      k++;
    }
    q->next = p->next;
    free(p);
    p = NULL;
  }
}
```



Q. WAP to reverse the linked list.

```
node * reverse (node *head)
{
  node *p=null, *q=nlll , *r= null;
  r= head;
  while (r)
  { q = r → next;
    r → next = p;
    p= r;
    r=q;
  }
  return p;
}
```

Q. (GATE) Consider the function 'f' defined below:

```
struct item
{ int data;
  struct item *next;
};
int f ( struct item *p)
{
  return ((p==NULL) || (p→next==NULL) || ((p→data <= p→next→data) && f(p→next)));
}
```

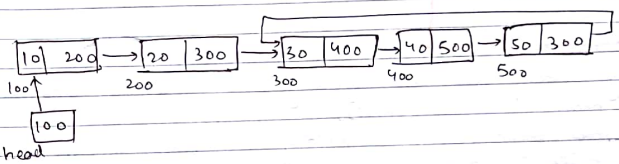
(Bracket has the highest priority)

for a given linked list, the function 'f' return '1' if and only if ?

(a) The list is empty or has exactly one element.
(b) The elements in the list are sorted in non decreasing order.
(c) The elements in the list are sorted in non increasing order.
(d) Not all elements in the list have same data value.

Increasing order	Non-decreasing
10, 20, 30, 40	10, 20, 20, 30, 30, 40

Q. WAP to find whether cycle exists or does not exists in the given linked list?




```
void findcycle (node *head)
{ node *slow, *fast;
    slow = head;
    fast = head;
  while (fast!=null && fast→next!=null && fast→next→next!=NULL)
  {
    slow = slow→next;
    fast = fast→next→next;
    if (slow==fast)
    { pf(" cycle exists");
      exit();
    }
  }
  printf (" No cycle exists");
}
```

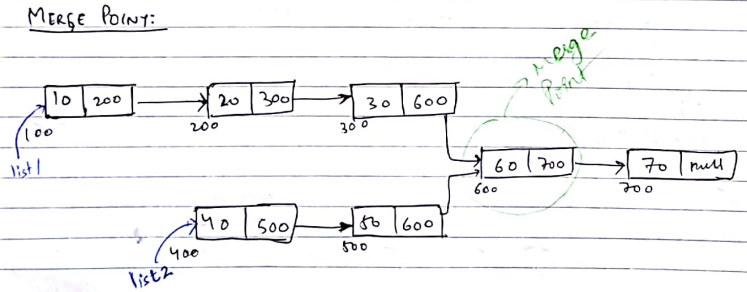
Solution 2:

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

* Initially, when you are creating the linked list, initialize, the boolean flag = 0 [FALSE];
* Every time, when we are visiting the node, check the flag value, if it is zero (0), make it 1, and move the pointer to the next node.
* If the flag value is already 1, it means that node is already visited and there exists a cycle.
* If there is no cycle, then, we will reach to the last node which contains NULL pointer.

Q. WAP to find the merge point in the given linked list.

MERGE POINT:




```
node* findmergepoint (node *list1, node *list2)
{
  int l1 = 0, l2 = 0, diff = 0, i;
  node *head1, *head2;
    head1 = list1;
    head2 = list2;
  l1 = listlength (head1);
  l2 = listlength (head2);
  if (l1 > l2)        head1 will always point to largest list
  { head1 = list1;
    head2 = list2;
    diff = l1 - l2;
  }
  else
  { head1 = list2;
    head2 = list1;
    diff = l2 - l1;
  }
  for (i = 0; i < diff; i++)
  head1 = head1 -> next;
  while (head1 != NULL && head2 != NULL)
  { if (head1 == head2)
      return head1;
    head1 = head1 -> next;
    head2 = head2 -> next;
  }
  return NULL;
}
```

DRAWBACKS OF SINGLE LINKED LIST:

① we cannot go back, because every node contains only 1 pointer to the next node.

② In the single linked list, last node's next pointer is always NULL and not utilised properly.

*To eliminate these drawbacks we will implement Circular single linked list.

Single linked list Time complexity:

	At begin	At End
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)

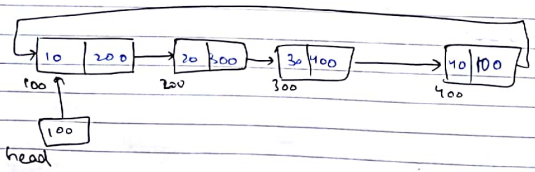
NOTE

↳ If the pointer "p" is given, pointing to some node, in the single linked list and deleting that node takes O(n) time.

07/10/2017

Circular Single Linked List (CSLL)

→ In the CSLL, the last node's next pointer will point to head node.



Q - WAP to find the length of CSLL.

```
int circularlistlength (node *head)
{ node *q;
  int count = 0;
  if (head == NULL).
      return count;
  q = head;
  do
  { count++;
    q = q->next;
  } while (q != head);
  return count;
}
```

H.W WAP to print all the contents of single circular linked list.

Q. WAP to insert new node in the beginning of Single circular linked list. Begin

```
void insertAtSCLL (node *head)
{
  node *temp, *q;
  temp = (node*) malloc (size of (node));
  temp -> next = NULL;
  printf ("Enter data");
  scf ("%d", &temp -> data);
  if (head == NULL)
  {  head = temp;
     head -> next = temp;
  }
  else
  { q = head;
    while (q -> next != head)
    { q = q -> next;
    }
    q -> next = temp;
    temp -> next = head;
    head = temp;
  }
  //temp = NULL;
  //q = NULL;
}
```

Q. WAP to delete first node of the SCLL.

```
void deleteAtBeginSCLL (node *head)
{ node *q;
  if (head == NULL)
  { pf ("List is Empty");
    return;
  }
  if (head -> next == head)
  {  free (head);
     head = NULL;
  }
  else
  { q = head;
    while (q -> next != head)
    { q = q -> next;
    }
    q -> next = head -> next;
    free (head);
    head = q -> next;
  }
  // q = NULL;
}
```

H.W.

① WAP to delete the last node in the SCLL.

② WAP to insertAtEnd of the SCLL.

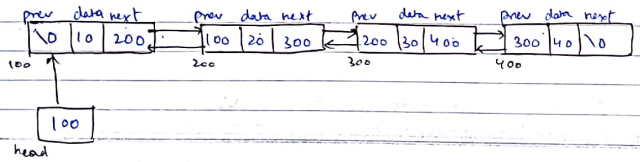
DOUBLE LINKED LIST (DLL) [Two way list]

→ The advantage of D.L.L. is we can navigate in both the directions.

→ A node in the single linked list cannot be deleted unless we have a ~~pointer~~ pointer to its previous node. But in D.L.L., we can delete a node even if we don't have previous node's address because each node has Left pointer and Right pointer pointing to their previous node and next node.

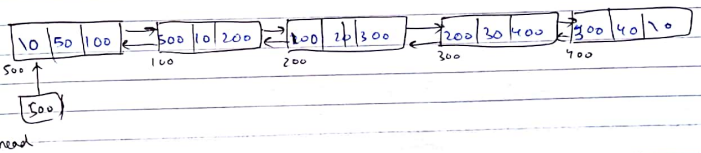
→ The disadvantage in the double linked list is that it requires an extra pointer and insertion and deletion will take more time because of more pointer updations.

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



Q. WAP to insert new node in the beginning of D.L.L.

```
void insertAtBeginDLL(node *head)
{ node *temp;
  temp= (node *) malloc( sizeof(node));
  pf("Enter data item");
  sf("%d" &temp→ data);
  temp→ next = NULL;
  temp→ prev = NULL;
  if (head == NULL)
  { head = temp;
  }
  else
   {  temp→ next = head;
      head→ prev = temp;
      head= temp;
   }
  //temp=NULL;
}
```



Q. WAP to delete the first node of the double linked list.

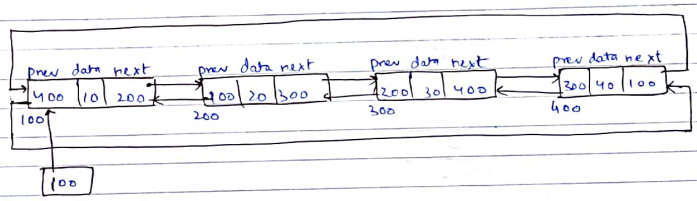
```
void deleteAtBeginDLL(node *head)
{  //node *q;
  if(head==NULL)
    {printf("List is empty");
     return;
    }
  if(head->next==NULL)
    {free(head);
     head=NULL;
    }
  else
    {head=head->next;  // q=head;  head=head->next;
     free(head->prev);  // free(q);
     head->prev=NULL;  // head->prev=NULL;  q=NULL;
    }
}
```

H.W: WAP to deleteAtEnd and insert the node new node at the end in D.L.L.

Circular Double Linked List (CDLL) or (DCLL):

→ In the CDLL, the first node's ~~previo~~ prev will point to the last node and last node's next will point to the first node.

→ The main advantage of CDLL is, if any pointer is lost, then, with the help of another pointer, we can reach the desired node.



H.W. WAP to insert → At Begin, At End

WAP to delete → At Begin, At End

NOTE

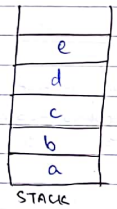
In the single linked list to search any node it takes O(n) time complexity.
To improve this to O(1), the concept of Skip list is used [Advanced Data Structures].

STACK

→ Stack is a data structure which follows Last in First out (LIFO) or First In Last out (FILO).
→ Elements will be inserted and deleted from the same end called Top.
→ The new element will be inserted at the top and the topmost element will be deleted first.

insertion:
a, b, c, d, e

deletion:
e, d, c, b, a



STACK

#Applications of Stack:
① Recursive function calls.
② Matching HTML and XML tags.
③ Undo and Redo operations (Restore).
④ Infix to Postfix conversion.
⑤ Postfix evaluation.
⑥ Fibonacci Series.
⑦ Towers of Hanoi.
⑧ Balancing paranthesis.
⑨ Page visited (Back Button).

Stack ADT:
* Declaration of Data: ① Space required to store stack elements.
② Variable "Top"

* Declaration of operations:
① push(x);
Data item
• Inserts element "x" into the stack using "Top".

② pop();
• deletes the element from the stack using "Top".

③ isFull();
• returns TRUE if the stack is full.

④ isEmpty();
• returns TRUE if the stack is empty.

Implementation of Stack:
① Using Array:

int s[5]; N=5

Initially,
top = -1

Stack

Q. WAP to perform push() operation into the stack.
① Initially check for the overflow condition.
② If it is not overflow, then increment the top and insert the element into the stack, with the help of top.

```
push (S, N, Top, x)
{
  if (Top == N-1)
    { pf ("overflow condition");
      exit();
    }
  Top++;
  S[Top] = x;
}
```

Q. WAP to perform pop() operation in a stack.

* Initially check for the underflow condition
* If it is not underflow, then, delete the element with the help of Top, then decrement the top.
* And return the deleted element.

```
int pop (S, N, Top)
{
  int item;
  if (Top == -1)
  { pf("Stack Underflow!");
    exit();
  }
  item = S[Top];
  Top--;
  return item;
}
```

```
if (Top == -1)
{pf("Underflow");
 exit();
}
return S[Top--];
```

Q. Find which of these operations are valid?

(a) getTop (pop (push (y, push (x, s)))) = y
(b) pop (pop (push (x, pop (push (x, s)))))
(c) isEmpty (push (c, push (b, pop (push (a, s))))) = FALSE
(d) push (a, push (b, push (c, pop ())));

Q. GATE2004

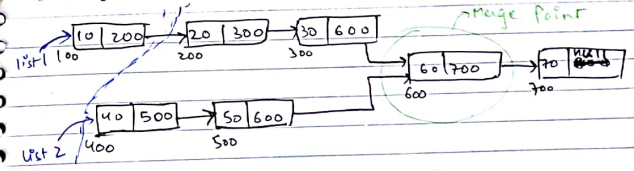
A program attempts to generate as many permutations as possible of the string "abcd" by pushing the characters a, b, c, d in the same order onto a stack, but it may pop off the top character at any time. Which one of the following strings cannot be generated by using this program.

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

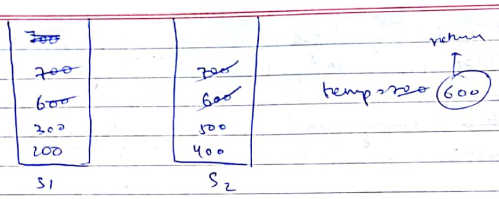
Q. Which of the following permutations can be obtained in the output [in the same order] using a stack assuming that i/p sequence is "1, 2, 3, 4, 5" in the same order.

(a) 3, 4, 5, 1, 2
(b) 1, 5, 2, 3, 4
(c) 5, 4, 3, 1, 2
(d) None of the above

Finding Merge Point in the linked list using stack:



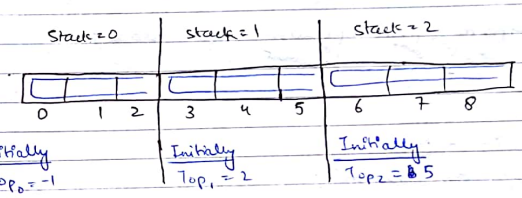
diff = $l_1 - l_2$



→ Find out the length of the list 1 and list 2 and take the difference using larger list length.
→ Traverse the difference until the length of both the lists ~~are~~ is equal.
→ Take the 2 stacks, push all the list1 addresses into stack 1 and push all the list 2 addresses into stack 2.
→ Now compare top addresses of both the stacks, if they are same, then, pop the elements from both the stacks ~~one~~ and keep the address in a temporary variable.
→ Continue this process until the top node addresses are different.
→ If they are different, return the temporary variable and that is the merge point.
→ If there is no merge point, then, initially itself the addresses will be different.

Implementing Multiple Stacks in Single array:

array size (N) = 9
No. of stacks (M) = 3 ; size of each stack $\left(\frac{N}{M}\right) = \frac{9}{3} = 3$



→ In the stack segment of the main memory, if we have 1 stack, we can execute only 1 recursive program.
→ If we want to execute multiple recursive programs, then, we require multiple stacks. It means stack area of the memory will be divided into multiple stacks [Then only we can execute multiple recursive programs].
→ This concept is similar to implementing multiple stacks in a single array.

Initial $Top_i = i * \frac{N}{M} - 1$

$Top_0 = 0 * \frac{9}{3} - 1 = -1$

$Top_1 = 1 * \frac{9}{3} - 1 = 2$

$Top_2 = 2 * \frac{9}{3} - 1 = 5$

NOTE

→ Overflow condition for stack i is, if top i is reaching to the initial top of (i+1) stack, then, it is overflow condition.

Q. WAP to perform push() operation into stack i.

```
void push (S, N, M, Ti, i, x)
{
  if (Ti == (i+1)*(N/M) - 1)
    {printf("Overflow condition");
     exit();
    }
  top++;  Ti++;        } => S[++Ti] = x;
  S[top] = x;  S[Ti] = x;
}
```

Q. WAP to perform pop() operation on the stack.

```
int pop (S, N, M, Ti, i)
{ int x;
  if (Ti == (i*N/M) - 1)
    {printf("Underflow condition");
     exit();
    }
  x = S[Ti--];        } => return (S[Ti--]);
  return x;
}
```

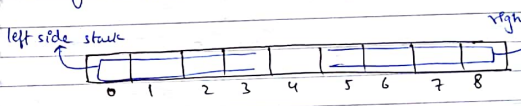
NOTE

→ In the previous implementation, if one stack is full and all the other remaining stacks are empty, then, it will show the overflow condition, even though, there are empty spaces available.

→ To avoid this problem, we will implement another approach

Implementing Multiple Stacks in a single array [Efficient way]

Array size (N) = 9 No. of stacks = 2



Initially: $T_L = -1$

Initially: $T_R = 9$

→ In the above implementation, we are using only 2 stacks in a single array, growing in opposite direction.

* Overflow condition for both the stacks:

$T_R - T_L = 1$
$T_R + 1 = T_R$
$T_R - 1 = T_L$

* Underflow condition:

for STACK$_L$:

$T_L = -1$

for STACK$_R$:

$T_R = N$

H.W.

WAP to implement push and pop operation in the ~~previ~~ above implementation.

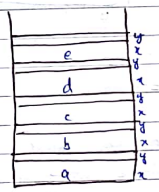
Finding avg lifetime of an element in stack:

→ The time taken for insertion or deletion of element is "x", and there is a delay of 'y' between every such operation.

→ The lifetime of element is considered as the time elapsed from end of push to start of pop operation that removes an element from the stack.

→ Then find average lifetime of an element.

element	Life time
e	y
d	y+x+y+x+y = 2x+3y
c	4x+5y
b	6x+7y
a	8x+9y



y+2x+3y+4x+5y+6x+7y+8x+9y

= 2x+4x+6x+8x + y+3y+5y+7y+9y

= 2x[1+2+3+4+...+n-1] + y[1+3+5+7+9+...+n]

= 2x·(n-1)n/2 + y[n²] (sum of n odd nos.)

⇒ n(n-1)x + yn² = (x(n-1)n + yn²)/n

⇒ x(n-1) + yn = nx - x + ny

⇒ n(x+y) - x

(c)

Infix, Prefix, Postfix:

* Infix: Binary operation between the operands is called as infix expression.

E.g. a+b

* Prefix: Binary operation before the operands is called as prefix expression.

E.g. +ab

* Postfix: Binary operation after the operands.

E.g. ab+

Infix to postfix, Infix to prefix Conversion:

Operator	Precedence	Associativity
( )	1	
↑	2	R→L
* /	3	L→R
+ −	4	L→R
=	5	R→L

Q.

E.g. A+B * (C+D)/F + D*E

Postfix:

⇒ A+B* CD+/F + D*E

⇒ A+ BCD+*/F + D*E

⇒ A+ BCD+*F/ + D*E

⇒ A+ BCD+*F/ + DE*

⇒ ABCD+*F/+DE*+

Prefix:

⇒ A+B * +CD/F + D*E

⇒ A+ *B+CD/F + D*E

⇒ A+ /*B+CDF + D*E

⇒ A+ /*B+CDF + *DE

⇒ +A/*B+CDF + *DE

⇒ ++A/*B+CDF*DE

Q1. Infix ⇒ a+b*c-d/e↑f*g+h

Postfix :	Prefix :
⇒ a+b*c-d/ef↑*g+h	⇒ a+b*c-d/↑ef*g+h
⇒ a+b*c-def↑/*g+h	⇒ a+b*c-/d↑ef*g+h
⇒ a+b*c-def↑/g*+h	⇒ a+b*c-*/d↑efg+h
⇒ a+bc*-def↑/g*+h	⇒ a+b*bc-*/d↑efg+h
⇒ abc*+-def↑/g*+h	⇒ +a*bc-*/d↑efg+h
⇒ abc*+def↑/g*-h+	⇒ -+a*bc*/d↑efgh
⇒ abc*+def↑/g*-h+	⇒ +-+a*bc*/d↑efgh

Q2. Infix ⇒ f*((d-b)↑(a/c))↑(e+g)*h

Postfix :

⇒ f*(db-↑ac/)↑eg+*h
⇒ f*(db-ac/↑)↑eg+*h
⇒ f*db-ac/↑↑eg+*h
⇒ f*db-ac/↑eg+↑↑*h
⇒ fdb-ac/↑eg+↑↑*h
⇒ fdb-ac/↑eg+↑*h*

Prefix :
⇒ f*(-db↑/ac)↑+eg*h
⇒ f*↑-db/ac↑+eg*h
⇒ f*↑↑-db/ac+eg*h
⇒ *f↑↑-db/ac+egh
⇒ **f↑↑-db/ac+egh

Prefix to Postfix Conversion :

→ In order to convert prefix to postfix, we need :
① Operator
② followed by 2 operands.

→ If we find them, then, convert them into postfix.

Eg ① +ab
⇒ ab+

② *+cd-ba
⇒ *cd+-ba
⇒ cd+*-ba
⇒ cd+*ba-
⇒ cd+ba-*

③ *↑/+cae*db↑fg
⇒ *↑/+cae*dbfg↑
⇒ *↑/+caedbfg↑*
⇒ *↑/caedbfg↑*+
⇒ *↑caedbfg↑/+
⇒ *cae↑/dbfg↑+

⇒ *↑/+cae*dbfg↑
⇒ *↑/+caedb*fg↑
⇒ *↑/ca+edb*fg↑

⇒ *↑/ca+e*db↑fg
⇒ *↑ca+e/*db↑fg
⇒ *↑ca+e/db*↑fg
⇒ *ca+e/db*↑↑fg
⇒ *ca+e/db*↑fg↑
⇒ *ca+e/db*↑fg↑*

Q. Match the following:

Prefix	Postfix:
① *+ab-cd	(a) ab*cd+-
② +-ab*cd	(b) ab*cd-+
③ +*ab-cd	(c) ab+cd-*
④ -*ab+cd	(d) ab-cd*+

① *+ab-cd
→ *ab+-cd
→ ab+cd-* (c)

② +-ab*cd
→ +ab-*cd
→ ab-cd*+ (d)

③ +*ab-cd
→ +ab*cd-
→ ab*cd-+ (b)

④ -*ab+cd (a)
→ -ab*cd+
→ ab*cd+-

NOTE

→ Procedure must be sequential for the computer in order to evaluate any expression.

→ If we are using the infix expression, then, we need to do multiple scans and jump from one point to other point according to the precedance.

→ To avoid this problem, the best solution is first converting infix expression to postfix expression and later evaluating the postfix expression in a single scan.

Infix to postfix Conversion using Operator Stack:

→ In order to convert infix to postfix operator stack will be used, it means, only operators will be push onto the stack.

E.g. ① ~~a+b*c↑d-e~~ a+b*c↑d-e

→ a+b*cd↑-e

→ a+bcd↑*-e

→ abcd↑*+e- → abcd↑*+e-

high → high above low.
low

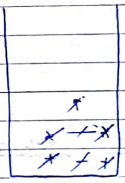
postfix: abcd↑*+e-

Size of stack = 3

E.g. ② a+b*c-d/e↑f*g+h

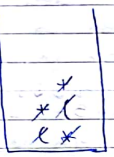
postfix: abc*+def↑/g*-h+

Stack size = 3



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

postfix: ab+cd+*



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

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

Stack size = 5

Q. When we will reach symbol "f" while scanning during conversion from infix to postfix what will be the contents of stack.

Infix: a+b-c/e↑f+g

abtce/f

(a) -, *
(b) /, *
(c) *, +
(d) *, -

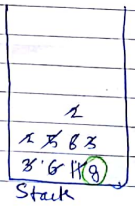
POSTFIX EVALUATION USING OPERAND STACK:

→ In order to evaluate postfix expression, operand stack will be used, it means only operands will be pushed on to the stack.

Eg. 32*5+62/-

op1	operand	op2	Result
3	*	2	6
6	+	5	11
6	/	2	3
11	-	3	8

Result = 8

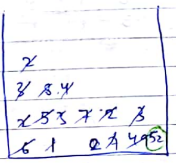


Stack

Eg. 623+-382/+*2↑3+

op1	operator	op2	Result
2	+	3	5
6	-	5	1
8	/	2	4
3	+	4	7
1	*	7	7
7	↑	2	49
49	+	3	52

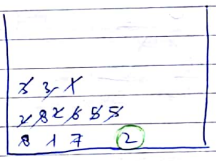
Size of stack = 4



Eg.. 823↑/23*+51*-

op1	operator	op2	Result
2	↑	3	8
8	/	8	1
2	*	3	6
1	+	6	7
5	*	1	5
7	-	5	2

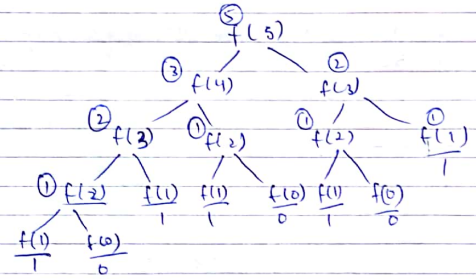
Size of stack = 3



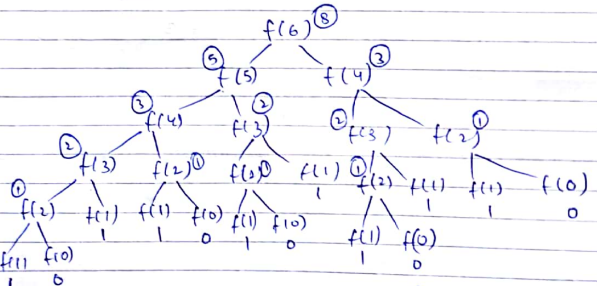
FIBONACCI SERIES:

n	0	1	2	3	4	5	6	7	8	9	10	11
Fib(n)	0	1	1	2	3	5	8	13	21	34	55	89

$$Fib(n) = \begin{cases} n & ; \text{if } n=0 \text{ or } n=1 \\ fib(n-1) + fib(n-2) & , \text{if } n>1 \end{cases}$$



No. of additions = 7
No of function calls = 15



No. of additions = 12
No. of func^{n} calls = 25

No. of additions:

f(0)	0
f(1)	0
f(2)	1
f(3)	2
f(4)	4
f(5)	7
f(6)	12
f(7)	20

No. of additions for Fib(n)
= no. of additions in fib(n-1)
+ no. of additions in fib(n-2)
+ 1

No. of function calls:

f(0)	1
f(1)	1
f(2)	3
f(3)	5
f(4)	9
f(5)	15
f(6)	25
f(7)	41

No. of function calls for fib(n)
= no. of function calls for fib(n-1)
+ no. of function calls for fib(n-2)
+ 1

Queue :

→ Queue is a data structure which follows the discipline of first in first out (FIFO) or ~~FI~~ last in last out (LILO).
→ Elements are inserted from one end and deleted from another end.
→ It requires 2 pointers :
① Rear → used to insert an element.
② Front → used to delete an element.

Applications of Queue :
① Ticket Reservations
② ~~Printer~~ Printer Spooler daemon
③ Job scheduling

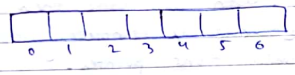
Queue ADT :
① Declaration of Data :
→ Space required to store queue elements.
→ Two pointers → Front
→ Rear

② Declaration of operations :
ⓐ Enqueue (x) ; //insert(x);
→ Enqueue() operation inserts the element "x" into the queue using rear pointer.

ⓑ Dequeue (); //delete(x);
→ Deletes the element from the queue using front pointer.

Implementation of Queue :
* Using Array :

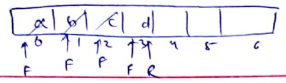
int Q[7]; N=7



initially
F = -1
R = -1

Q. WAP to perform enqueue operation on the queue.
• Initially check for the overflow condition.
• If it is the first insertion, then increment both front and rear pointers otherwise, increment only rear pointer and then, ~~insert~~ insert the element into the queue using rear pointer.

```
Enqueue (Q, N, F, R, x)
{
  if (R == N-1)
  { printf("Overflow condition");
    exit();
  }
  if (R == -1)
  { F++;
    R++;
  }
  else
    R++;
  Q[R] = x;
}
```



F = 0 1 2 3
R = 0 1 2 3

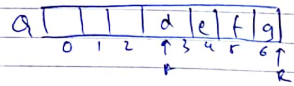
① Delete
② Check front = rear
③ If true f = -1 R = -1

Q. WAP to perform dequeue operation in the queue.

- Initially check for the underflow condition
- If it is not underflow, then, delete the element by using front pointer.
- If front and rear, both are same, then, reinitialise them back to -1. [Queue is completely empty].
- Otherwise increment only front pointer.

```
int Dequeue (Q, N, F, R)
{ int y;
  if (F == -1)
    { pf ("Underflow condition");
      exit();
    }  y = Q[F];
  if (F == R)
    { F = -1;
      R = -1;
    }                    last deletion
  else
    { F++;
    }
  return y;
}
```

NOTE

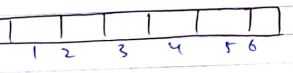


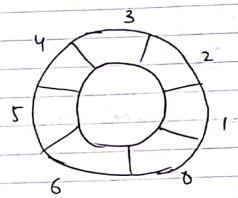
F = 0 1 2 3
R = 0 1 2 3 4 5 6

→ In the above implementation of linear queue, even though there are some empty spaces available, we cannot make use of them.

→ This situation arises when the rear pointer has reached extreme last and it will show the overflow condition. [as shown in the diagram].

Circular Queue:





Q. WAP to perform Enqueue operation in the circular queue.

- Initially check for the overflow condition.
- If it is not overflow and if it is the first insertion, then, increment both front and rear.
  Otherwise increment only ~~read~~ rear pointer by using mod.

Overflow Condition:

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

$R = (R+1) \bmod N$

```
Circular Enqueue (Q, N, F, R, x)
{
  if((R+1)%N==F)
   {pf("overflow");
    exit();
   }
  if (F==-1)
   {F++;
   }
  Q[(++R)%N] = x;
}
```

Q. WAP to perform Dequeue operation in the circular queue.

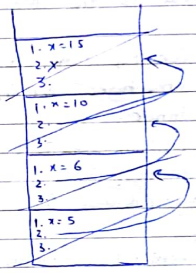
- Initially check for the underflow condition.
- If it is not underflow, then delete the element by using front pointer.
- If the front and rear, both are same, then, reinitialise, them back to −1, otherwise increment only front pointer using mod.

```
int CircularDequeue(Q, N, F, R)
{ if (F==-1)
   {pf("Underflow");
    exit();
   }
  int x;
   x = Q[F];
   if (F==R)
    { F=-1; R=-1; }
  else
    { F=(F+1)%N; }
  return x;
}
```

Q① Consider the following function which takes queue values as input and performs some functionality/operation [if the queue is empty, then NULL will be used].

Q = 5, 6, 10, 15, 15, 10, 6, 5

```
ABC (Queue Q)
{
  pf (!isEmpty(Q))
   {
     x = dequeue (Q);
     ABC(Q);
     enqueue (Q, x);
   }
}
```



What will be the values in Q after completion of the above program? 15, 10, 6, 5

Q② Consider the following program which uses 3 data structures.

Queue, (Q_1), Queue$_2$(Q_2) and stack (st)

The program takes Q_1 values as input and performs some operation.

Q_1 = 3, 7, 6, 2, 9, 0, 4, 5, 12, 8, 6, 0, 8, 3

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

① What is the sum of all the values of stack after completion of the above program? 58

② What is the max stack size used 12

Q. Consider the following program which uses 3 data structures Q_1, Q_2 and stack and takes Q_1 values as input.

Q_1 = 3, 7, 6, 2, 9, 0, 4, 5, 12, 8, 6, 0, 8, 3

```
xyz(Queue Q1)
{
  while(!isEmpty(Q1))
  {
    x = dequeue(Q1);
    if(x==0){
      pop(st);
      pop(st);
      while(!isEmpty(st))
      Enqueue(Q2, pop(st));
    }
    push(st, x);
  }
}
```

① what is the sum of all the values of Q_2 after completion of the above program. 37

② what is max stack size 6

Q. Consider the following program using 3 data structures Q_1, Q_2 and stack and taking Q_1 values as input.

Q_1 = 3, 7, 6, 2, 9, 0, 4, 5, 12, 8, 6, 0, 8, 3

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

① what is the sum of all the values of Q_2

② what is max stack size used.

Chapter 5

Q.8. ① i=1 $p = 1 \times \frac{x}{1} = x$

$s = 1 + x$;

② i=2 $p = x \times \frac{x}{2} = \frac{x^2}{2}$

$s = 1 + x + \frac{x^2}{2}$

③ i=3 $p = \frac{x^2}{2} \times \frac{x}{3} = \frac{x^3}{6}$

$s = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$

④ i=4 $p = \frac{x^3}{6} \times \frac{x}{4} = \frac{x^4}{24}$

$s = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^3}{24}$

$s = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$

As per, Taylor series → e^x

```
var n:int                                   n 10
procedure w(var x:int)
begin
    x = x+1;
    print x;
end
procedure D
begin
    var n:int                               o/p 4
    n=3;                                    n 84
    w(n);                                   100
end
begin
    n=10;
    D();
end
```

Implementing Queue Using Stacks:

Q. Let 'n' insert and ($m \leq n$) delete operations be performed in an arbitrary order on an empty Q (using only two stacks S_1 and S_2). Let x and y be number of push and pop operations performed respectively in the processes. Which one of the following is TRUE for all m and n?

(a) $n+m \leq x \leq 2n$ and $2m \leq y \leq n+m$
(b) $n+m \leq x < 2n$ and $2m \leq y \leq 2n$
(c) $2m \leq x < 2n$ and $2m \leq y \leq n+m$
(d) $2m \leq x < 2n$ and $2m \leq y \leq 2n$

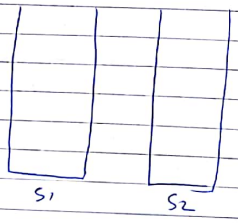
Insert (n): a, b, c, d, e
Delete (m): a, b, c,

Max = push = 5+5 = 2n
Min = push = 5+3 = n+m

Max = pop = 5+3 = n+m
Min = pop = 3+3 = 2m

Q WAP for enqueue and dequeue operations in the queue implemented using stacks.

Q




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

int Dequeue (S1, S2)
{
   if (S2 is not Empty)
   {  return pop(S2);
   }
   else if (S1 is Empty)
      {  pf ("Queue is Empty");
         exit();
      }
      else
      {  while (S1 is NOT Empty)
         {  x = pop(S1); //pop all elements from S1
            push (S2, x); // push into S2
         }
         return pop (S2);
      }
}
```

Implementing Stack using Queues:

Q_1 

Q_2 

Q. WAP to perform push and pop operation into the stack implemented using queues.

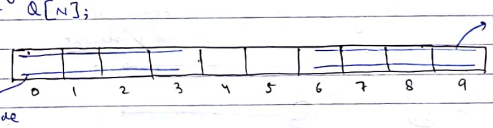
```
push (Q1, Q2, x)
{ if (Q1 is Empty)
    Enqueue (Q2, x);
  else
    Enqueue (Q1, x);
}

int pop (Q1, Q2)
{ if (Q1 is Empty)
  { if (Q2 is Empty)
    { pf("Stack Empty");
      exit();
    }
  else
    { while (Q2 not contain one element)
      { x = Dequeue (Q2);
        Enqueue (Q1, x);
      }
      return Dequeue (Q2);
    }
  }
  else
  { while (Q1 not contain one element)
    { x = Dequeue (Q1);
      Enqueue (Q2, x);
    }
    return Dequeue (Q1);
  }
}
```

Double Ended Queue: [2 Queues in single array]

array size(Q) = 10, N = 10

int Q[N];



Initially:
F_L =
R_L =

Initially:
F_R =
R_R =

→ Double ended queue is a concept of implementing 2 queues in a single array growing in opposite direction.

	Underflow condition	Overflow condition
Left Queue	$F_L == -1$	$R_R - R_L == 1$
Right Queue	$F_R == 10 (N)$	$R_R - R_L == 1$

H.W.

Q. WAP to perform Enqueue and Dequeue operations in the left side and right side queues.

TREES:

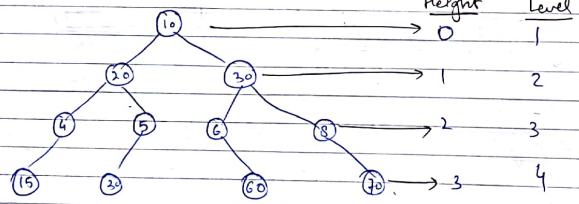
→ Tree is a non-linear ~~data~~ data structure, which follows hierarchical structure like parent-child relationship.

① Binary Trees
② n-ary Trees.

* Binary Trees:

→ In a binary tree, every node can contain atmost 2 child nodes.

E.g:



~~Height~~ Level NOTE

→ Height and level might change according to new definition.

Degree of a Tree: decides the no. of nodes each node can contain.

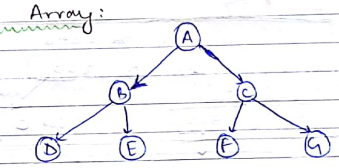
→ If the degree of a tree is "D", it means atmost every node can have "D" child nodes

Representation of Trees:

① Using array
② Using linked list

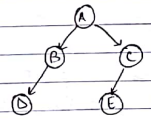
* Using Array:

Eg①



A	B	C	D	E	F	G
0	1	2	3	4	5	6

Eg②



A	B	C	D	-	E	-
0	1	2	3	4	5	6

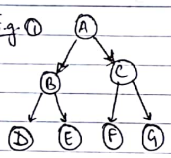
* Using Linked List:

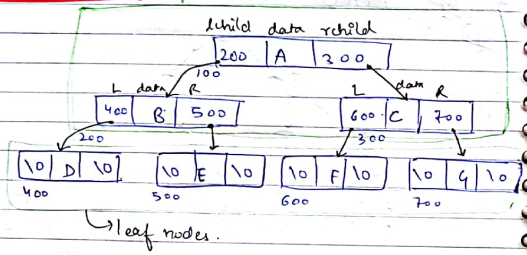
NOTE
→ The tree representation using linked list will be done by using double linked list.

```
struct node
{
  struct node *lchild;
  int data;
  struct node *rchild;
};
typedef struct node node;
```

non-leaf nodes (or) internal nodes

Eg: ①





↳ leaf nodes.

NOTE

→ While writing the programs in trees, simplest way is using recursion because every node is having same properties.

Q. WAP to to find the numbers of leaf nodes in the binary tree.

```
int NL(node *t)
{ if(t==NULL)
    return 0;
  if( t->lchild==NULL && t->rchild==NULL)
    return 1;
  return ( NL(t->lchild) + NL(t->rchild));
}
```

Q. WAP to find the number of non-leaf node, i.e., internal nodes in the binary tree.

```
int NI(node *t)
{
  if (t==NULL)
    return 0;
  if (t->lchild==NULL && t->rchild==NULL)
    return 0;
  return (1 + NI(t->lchild) + NI(t->rchild));
}
```

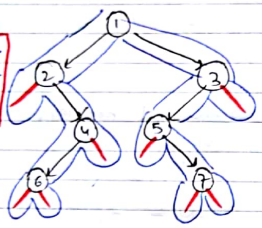
Q. WAP to find total number of nodes in a given binary tree.

```
int NN(node *t)
{ if(t==NULL)
    return 0;
  // if(t->lchild==NULL && t->rchild==NULL);
  // return 1;
  return (1 + NN(t->lchild) + NN(t->rchild));
}
```

Tree Traversals:

Preorder : Root, Left, Right
Inorder : Left, root, right
Post order : Left, Right, Root

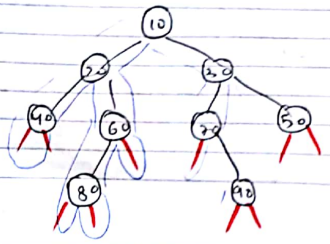
Preorder → 1st time visit
Inorder → 2nd time visit
Postorder → 3rd time visit



Preorder: 1 2 4 6 3 5 7
Inorder : 2 6 4 1 5 7 3
Post order : 6 4 2 7 5 3 1

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

NOTE
→ To apply this technique, every node should definitely have 2 child nodes.
→ If they are having original childs then no problem otherwise keep some dummy node. [/]

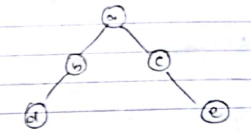


Pre → 10, 20, 40, 60, 80, 30, 70, 90, 50
In → 40, 20, 80, 60, 10, 70, 90, 30, 50
Post → 40, 80, 60, 20, 90, 70, 90, 50, 30, 10

Double order and Triple order traversals:

* Double order traversals:

```
Do (node *t)        // pointer to the root node
{ if (t)
  { pf (t->data);
    Do (t->left);
    pf (t->data);
    Do (t->right);
  }
}
```



What is the o/p for on above tree?
a, b, d, d, b, a, c, c, e, e

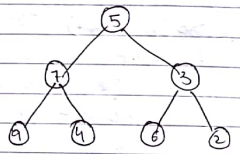
* Triple order traversals:

```
To (node *t)
{ pf(t)
  { pf (t->data);
    To (t->left);
    pf (t->data);
    To (t->right);
    pf (t->data);
  }
}
```

What is the output on above tree?
a b d d d b b a c c e e e c a

Indirect Recursion on Trees:

Q. Consider the following tree and functions defined:




```
A (node *t)
{
  if (t)
  {
1. pf (t→data);
2. B(t→left);
3. B(t→right);
  }
}
B (node *t)
{
  pf(t)
  {1. A(t→left);
   2. pf(t→data);
   3. A(t→right);
  }
}
```

what is the output of A(5)?

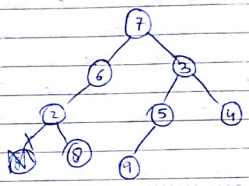
5, 9, 7, 4, 6, 3, 2

Construction of Unique Binary Tree:

→ The unique binary tree can be constructed only if any ~~the~~ two traversals are given and one of them should be inorder.

Post order: 8, 2, 6, 9, 5, 4, 3, 7

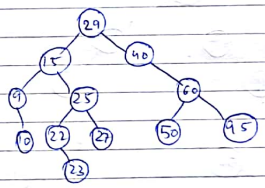
~~Inder~~ Inorder: 2, 8, 6, 7, 9, 5, 3, 4



GATE 2005

Q. post order: 10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

Inorder: 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95



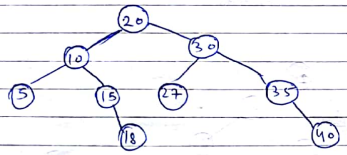
Preorder 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

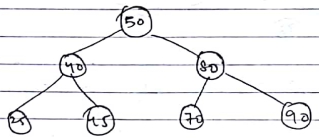
Binary Search Tree (BST):

→ BST is also a ~~tree~~ like a binary tree but with the restriction that all the elements in the left subtree should be smaller than the parent and all the elements in the right subtree should be greater than the parent.
→ All the element in the BST must be unique.
→ BSTs are used for searching an element.

Construction of BST:

20, 10, 15, 30, 35, 27, 5, 40, 18





Preorder : 50, 40, 25, 45, 80, 70, 90
Inorder : 25, 40, 45, 50, 70, 80, 90
Postorder : 25, 45, 40, 70, 90, 80, 50

NOTE
→ Inorder traversal of the BST always gives sorted order of the elements.
→ First element in the preorder and last element in the post order is same.

Q. Consider the numbers 1 to 100 in the BST. When searching for an element 43, which of the following sequences cannot be the sequences of nodes examined.
(a) ~~52,~~ 52, 14, 17, 40, 43
(b) 10, 65, 31, 48, 37, 43
(c) 81, 61, 52, 14, 41, 43
(d) ~~81, 61,~~ 3, 50, 40, 60, 43

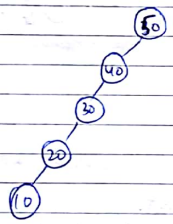
Q. When searching for the key=55, which of the following sequences cannot be examined.
(a) 10, 75, 64, 43, 60, 57, 55
(b) 90, 12, 68, 34, 62, 45, 55
(c) 9, 85, 47, 68, 43, 57, 55
(d) 79, 14, 72, 56, 16, 53, 55

Q. WAP to search an element in the BST.

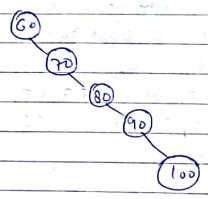
```
node * search (node *t, int x)
{
  if (t == NULL)
  { cout();
  }
  if (t->data == x)
    return t;
  if (x < t->data)
    search (t->left, x);
  else
    search (t->right, x);
}
```

NOTE

↳ The worst case time complexity to search an element in a BST is O(n).



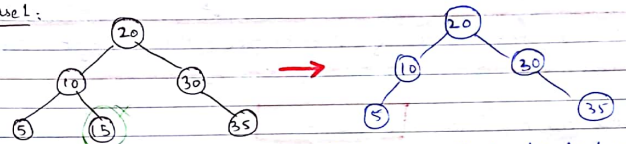
Left skewed BST



Right skewed BST

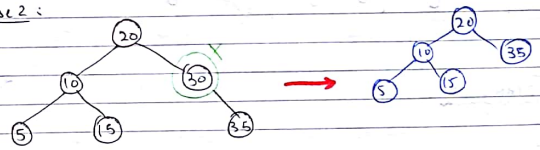
DELETIONS IN BST

Case 1:



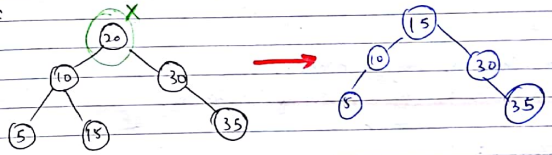
Delete 15: * Simply the node 15 because it is the leaf node and update its parent pointer with NULL.

Case 2:



Delete 30: * After deleting, the node 30, we need to update parent of node with the child of node 30, i.e., node 35 [address]

Case 3:



Delete 20:

Inorder: 5 10 15 20 30 35

Inorder predecessor
* Largest element in left subtree

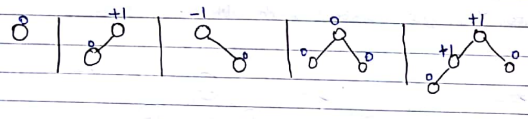
Inorder successor
* Smallest element in right subtree

11/6/2017

AVL Trees: [Height Balanced BST]
(Adelson, Valsley, Landis)

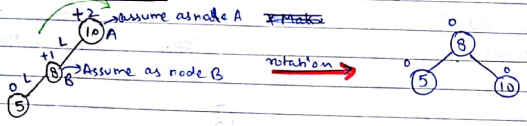
* Balance Factor = (Height of Left subtree) − (Height of right subtree)

* Accepted Balance Factor = (-1, 0, +1)



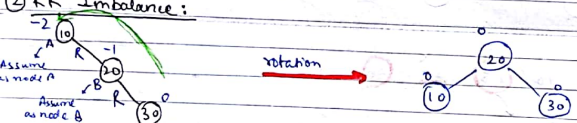
Rotation Techniques:

① LL Imbalance:



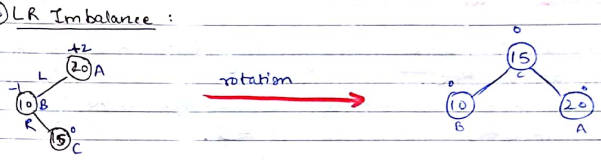
* Internally involves only 1 rotation.
* Make node A as the child of node B.

② RR Imbalance:

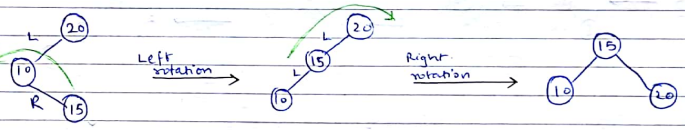


* Wherever you have imbalance, assume that node as node A.
* Make node A as child of node B.
* Internally, ~~now~~ involves only 1 rotation.

③ LR Imbalance:

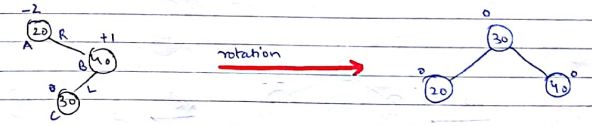


* Wherever you have imbalance, assume it as node A.
* Assume left, ~~node as node~~ child node of A as B.
* Assume right child node of B as node C.
* Make node A and B as child of node C.

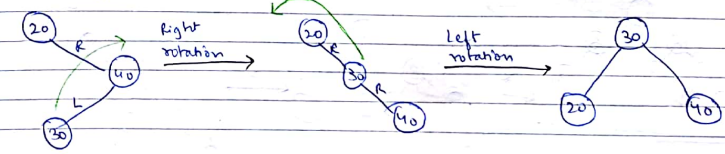


* Internally, it involves 2 rotations.

④ RL Imbalance:



* Make node A and B as child of node C.
* Internally, it involves 2 rotations.



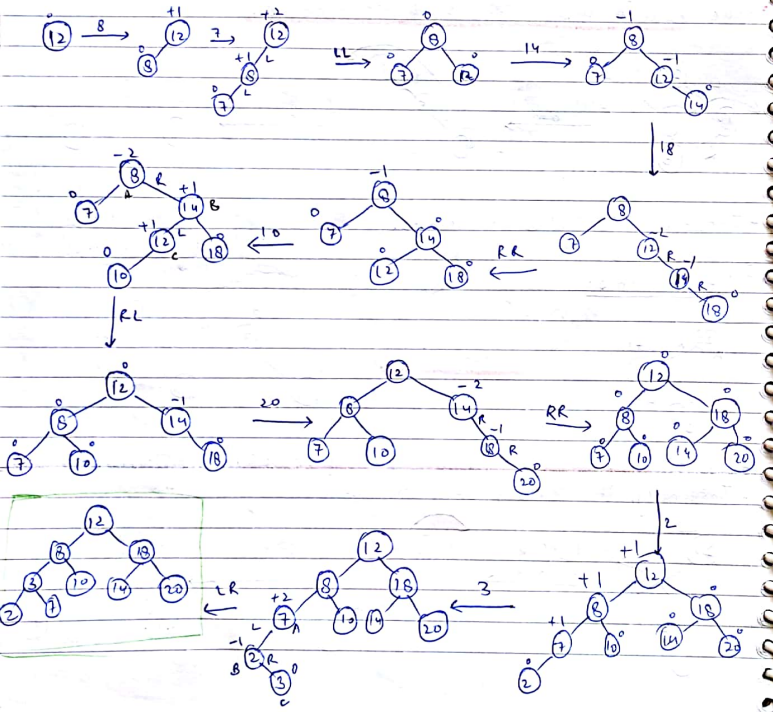
CONSTRUCTION OF AVL TREES:

→ The procedure is same as creating BSTs, inserting one element at a time.

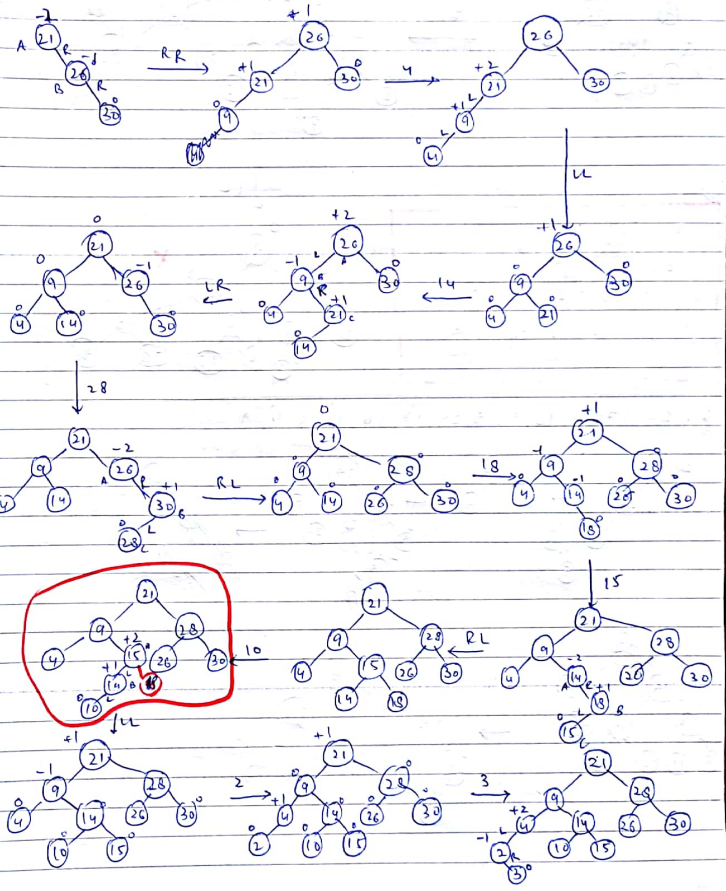
→ When a new node is inserted, the balance factor of all its ancestors will be updated.

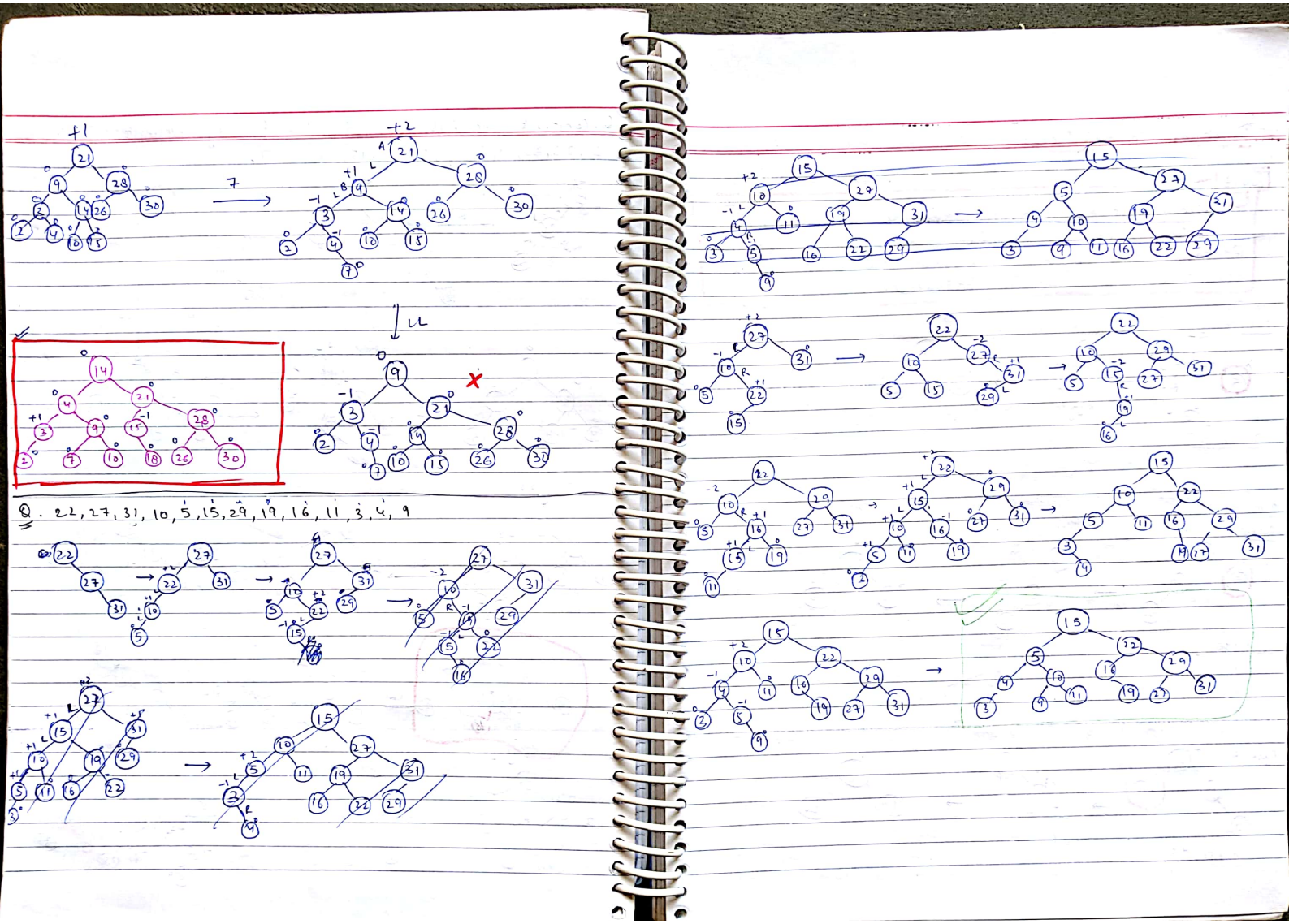
→ If any node is having imbalance, then, make the proper rotation technique and make it balanced.

E.g. 12, 8, 7, 14, 18, 10, 20, 2, 3



Q. Construct the AVL tree with the following elements:
21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7.



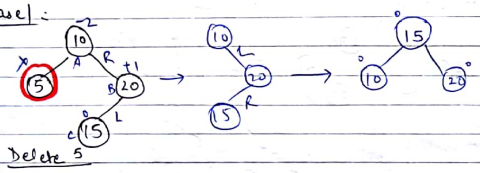


Deletions in AVL Trees:

NOTE

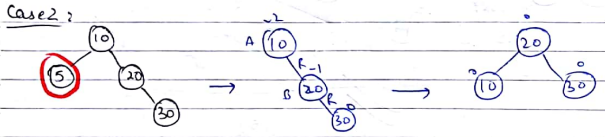
Deletion in AVL Tree is same as BST, but after deletion check the balance factor of every node and if there is imbalance, then, make the proper rotation technique and make it balanced.

Case1:



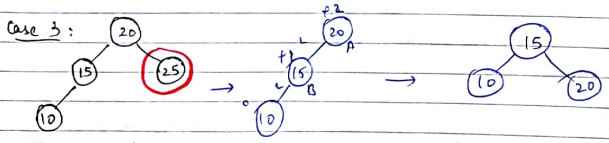
Delete 5

Case2:



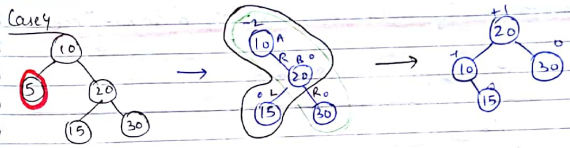
Delete 5

Case 3:



Delete 25:

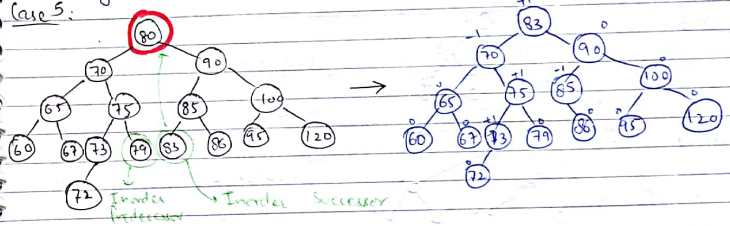
Case4



Delete 5:

- When there is a possibility of applying RR and RL rotation techniques, then, it is better to follow RR imbalance technique because it internally involves only one rotation. [But RL rotation internally takes 2 rotations]

Case 5:



Delete 80:

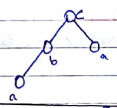
- We can replace the ~~root~~ node with the help of inorder predecessor or inorder successor, but after replacing, we need to check the balance factor of every node and if there is an imbalance, by applying proper rotation technique make it balanced.

Finding minimum no. of nodes with height 'h' in AVL Trees:

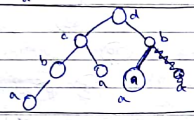
① h = 0 — nodes = 1

② h = 1 — nodes = 2

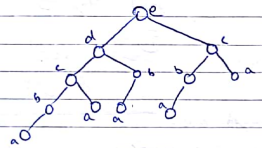
③ h = 2 — nodes = 4



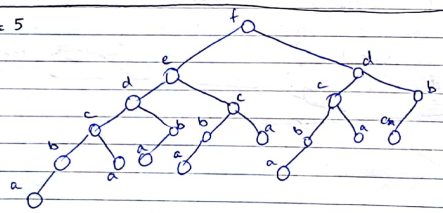
④ h = 3 — nodes = 7



⑤ h = 4 — nodes = 12



⑥ h = 5 — nodes = 20



Minimum no. of nodes in AVL tree = Min no of nodes with height (h-1) + Min no. of nodes with height (h-2) + 1

NOTE

→ Time complexity to search an element in AVL Tree:
Worst Case → O(logn)
Best Case → O(1)

HASHING

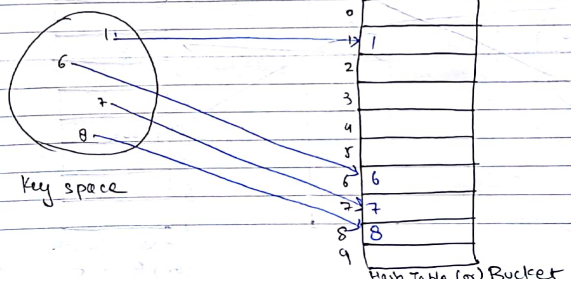
→ Hashing is used to search an element.
→ In order to search an element BSTs and AVL Trees are still taking more time.
→ To improve the search time complexity to O(1), the hashing technique will be used.
→ By using hashing, most of the time, we will achieve the search time complexity as O(1), but in some cases, the worst case time complexity is still O(n).

Definition: • Hashing is a mathematical computation over the key.
• The objective of hashing is to provide less collisions and improve the time complexity to O(1).

DIRECT ADDRESS TABLE:

→ Keys: 6, 8, 7, 1

Hash Table (M) = 10
M = (0-9)



→ In the direct address table, keys will be directly mapped on the hash table with the respective indexes w/o any computation.

→ Search time complexity:

B.C. / N.C → O(1)

→ This approach will fail, if we want to store, larger keys.

E.g. 80, 9999999999999, 2^{10103}

→ To store the larger keys, we need to maintain the large size hash table.

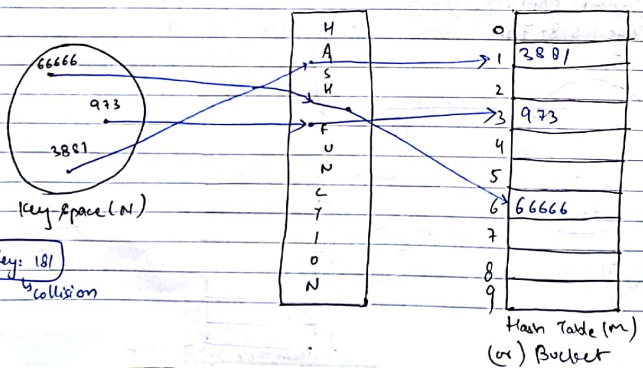
→ Even though, there are so many empty spaces available in the hash table and they are not efficiently used.

→ To avoid this problem, concept of hash function will be used.

Hash functions:

① Division module m:

m = 10 (0-9)



NOTE

↳ In the previous implementation, we can store the larger keys, but the problem comes when the collision occurs.

Eg. key = 181

*Collision: If 2 keys are mapped on to the same location on the hash table, then, it is called as collision.

→ To avoid the collision, various collision resolution techniques will be used.

Various Hash Functions:

① Division Modulo - M:

→ The value of M should be a prime number which is not closer to the powers of 2, then, it gives less collisions.

→ If we choose "M" values as exactly powers of 2, like, $M = 2^k$, then, Hash function of key =)

Hf(key) = LSB k bits always.

$M = 8 = 2^3 = 1000$ → LSB k bits.

$M = 7 \neq 2^k$
① key = 86 = 0 1 0 1 0 1 1 0 → 86 % 8 = 6 = 110 → 86 % 7 = 2
② key = 102 = 0 1 1 0 0 1 1 0 → 102 % 8 = 6 → 102 % 7 = 4
③ key = 118 = 0 1 1 1 0 1 1 0 → 118 % 8 = 6 → 118 % 7 = 6
④ key = 126 = 0 1 1 1 1 1 1 0 → 126 % 8 = 6 → 126 % 7 = 0

NOTE

↳ The better 'm' value is a prime no., which is not closer to power of 2, then, it gives less no. of collisions.

(2) Mid Square Method:

M = 1000 (0 - 999) → You can choose any digit.

Key = 5 2 3 6 9 1 2 6

1 2 3 4 5 6 7 8

$(22)^2 = 484$

Hash Table

484	52369126

(3) Folding Method:

(a) Fold Boundary Method:

M = 1000 (0 - 999) → You can choose any no. of digits.

Key = 5 2 3 6 7 6 1 2 3

523
+123
646

Hash Table

646	523676123

(b) Fold Shifting Method:

M = 1000 (0 - 999)

Key = 123 456 781

123
456
+ 781
1360

13
+ 60
73

Hash Table

73	123456 781

(4) Digit Extraction Method:

m = 1000 (0 - 999)

Key = 9 7 3 1 2 3 6 8 4

1 2 3 4 5 6 7 8 9

334

Hash Table

334	973123684

Collision Resolution Techniques:

- Chaining
- Open addressing
  1. Linear probing
  ~~2. Secondary probing~~
  3. Double hashing
  2. Quadratic Probing.

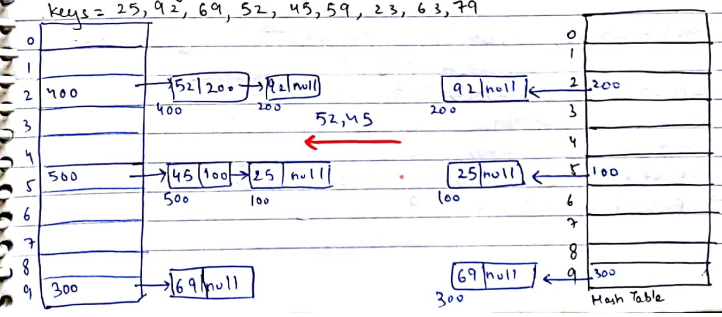
* Chaining:

→ Chaining is implemented with the help of linked list.

m = 10 (0 - - 9)

hf(k) = key mod M

keys = 25, 92, 69, 52, 45, 59, 23, 63, 79



→ The advantage of chaining is, we can handle any number of collisions until the memory is full.
→ Longest possible chain with 'n' keys is 'n'.
→ Search time complexity:
- Best case → O(1)
- Worst case → O(n)

→ In chaining, even though there are so many empty ~~spaces~~ space available in the hash table, we are using outside space in the form of linked list.
→ Insertion time complexity is always O(1).
→ Deletion time complexity:
Best Case → O(1)
Worst Case → O(n)

* Open Addressing:

① Linear Probing:

m = 10 (0 - 9)

hf(k) = key mod m;

LP(key, i) = (hf(key) + i) mod m;

i = 0, 1, ... 9 (m-1)

keys = 25, 38, 43, 68, 79, 46, 58, 65, 20

Initially i = 0, if collisions occurs make it 1.

① LP(25,0) = ((25%10)+0) = 5
② LP(38,0) = ((38%10)+0) = 8
③ LP(43,0) = ((43%10)+0) = 3
④ LP(68,0) = ((68%10)+0) = 8 • collision → ① collision
↳ LP(68,1) = ((68%10)+1) = 9

0	79
1	58
2	20
3	43
4	
5	25
6	46
7	65
8	38
9	68

⑤ LP(79,0) = ((79%10)+0) = 9 • → ①
↳ LP(79,1) = (79%10 + 1) = 10%10 = 0

⑥ LP(46,0) = ((46%10) +0) = 6

⑦ LP(58,0) = ((58%10)+0) = 8 •
↳ LP(58,1) = ((58%10)+1) = 9 •
↳ LP(58,2) = ((58%10)+2) = 10 •
↳ LP(58,3) = ((58%10)+3) = 11%10 = 1 → ③

⑧ LP(65,0) = (65%10+0) = 5 •
↳ LP(65,1) = (65%10+1) = 6 •
↳ LP(65,2) = (65%10+2) = 7 → ②

⑨ LP(20,0) = (20%10+0) = 0 •
↳ LP(20,1) = (20%10+1) = 1 •
↳ LP(20,2) = (20%10+2) = 2 → ②

Total collisions = 9

Pg25

Q.25. keys = 12, 18, 13, 2, 3, 23, 5, 15

$h(k) = k \bmod 10$.

(c)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

Q. keys = 53, 40, 91, 62, 70, 60

$h(k) = k \bmod 10$.

LP(70, 0) = 0+0 = 0
LP(70, 1) = 0+1 = 1
LP(70, 2) = 0+2 = 2
LP(70, 3) = 0+3 = 3
LP(70, 4) = 0+4 = 4
→ 4 collisions

LP(60, 0) = 0+0 = 0
LP(60, 1) = 0+1 = 1
LP(60, 2) = 0+2 = 2
LP(60, 3) = 0+3 = 3
LP(60, 4) = 0+4 = 4
LP(60, 5) = 0+5 = 5
→ 5 collisions

0	40
1	91
2	62
3	53
4	70
5	60
6	
7	
8	
9	

* Primary Clustering:
→ If the 2 keys are mapped onto the same starting location on the hash table, then, both will follow the same path unnecessarily in the linear manner and because of this search time complexity will increase.
→ This is called as primary clustering.

* Search Time Complexity:
Best case → O(1)
Worst case → O(m)

* Insertion time complexity:
Best case → O(1)
Worst case → O(m)

→ Deletions are little difficult and will create some problem to other elements, but, we can manage them, by storing some special symbol like "$" or "#" or anything.
→ In case, if there are more no. of deletions, then perform re-hashing.

Load factor (α):
→ The number of keys getting stored in 1 slot is known as load factor.

In 'm' slots → 'n' keys
In '1' slot → ?

$\frac{1}{m} \times n = \frac{n}{m} = \alpha$

$0 \le \alpha \le 1$

② Quadratic Probing:

$m = 10$ (0-9)

$hf(k) = k \bmod m$;

keys = 25, 98, 57, 75, 97, 18, 78, 46

$$Q.P.(key, i) = (hf(k) + c_1 i + c_2 i^2) \bmod m$$

Can take any constants $\leftarrow$ $c_1 = 1, c_2 = 1$, $i = 0, 1, 2, \ldots (m-1)$

① $QP(25,0) = ((25 \%_{10}) + 1\times0 + 1\times0^2) \%_{10}$
$= 5 \%_{10} = 5$

② $QP(98,0) = ((98 \%_{10}) + 1\times0 + 1\times0^2) \%_{10}$
$= 8 \%_{10} = 8$

③ $QP(57,0) = ((57 \%_{10}) + 1\times0 + 1\times0^2) \%_{10}$
$= 7 \%_{10} = 7$

④ $QP(75,0) = ((75 \%_{10}) + 1\times0 + 1\times0^2) \%_{10}$
$= 5$.

$QP(75,1) = ((7+5) \%_{10}) + 1\times1 + 1\times1^2) \%_{10}$
$= (5 + 1 + 1) \%_{10} = 7 \%_{10} = 7$.

$QP(75,2) = (5 + 1\times2 + 1\times2^2) \%_{10}$
$= 11 \%_{10} = 1$

⑤ $QP(97,0) = (7 + 0 + 0) \%_{10} = 7$.
$QP(97,1) = (7 + 1 + 1) \%_{10} = 9$

⑥ $QP(18,0) = (8 + 0 + 0) \%_{10} = 8$.
$QP(18,1) = (8 + 1 + 1) \%_{10} = 0$

0	18
1	75
2	
3	
4	78
5	25
6	46
7	57
8	98
9	97

Hash Table (m)

⑦ $QP(78,0) = (8 + 0 + 0) \%_{10} = 8$.
$QP(78,1) = (8 + 1 + 1) \%_{10} = 0$.
$QP(78,2) = (8 + 2 + 2) \%_{10} = 4$

⑧ $QP(46,0) = 6 + 0 + 0 = 6$

Total collisions = 6

Eg ②. keys = 53, 40, 91, 62, 70, 60

- $QP(53,0) = (3 + 0 + 0) \%_{10} = 3$
- $QP(40,0) = (0 + 0 + 0) \%_{10} = 0$
- $QP(91,0) = (1 + 0 + 0) \%_{10} = 1$
- $QP(62,0) = (2 + 0 + 0) \%_{10} = 2$
- $QP(70,0) = (0 + 0 + 0) \%_{10} = 0$.
- $QP(70,1) = (0 + 1 + 1) \%_{10} = 2$.
- $QP(70,2) = (0 + 2 + 4) \%_{10} = 6$
- $QP(60,0) = (0 + 0 + 0) \%_{10} = 0$
- $(60,1) = (0 + 1 + 1) \%_{10} = 2$
- $(60,2) = (0 + 2 + 4) \%_{10} = 6$
- $(60,3) = (0 + 3 + 9) \%_{10} = 2$
- $(60,4) = (0 + 4 + 16) \%_{10} = 0$
- $(60,5) = (0 + 5 + 25) \%_{10} = 0$
- $(60,6) = (0 + 6 + 36) \%_{10} = 2$
- $(60,7) = (0 + 7 + 49) \%_{10} = 6$
- $(60,8) = (0 + 8 + 64) \%_{10} = 2$
- $(60,9) = (0 + 9 + 81) \%_{10} = 0$
- $(60,10) = (0 + 10 + 100) \%_{10} = 0$

0	40
1	91
2	62
3	53
4	
5	
6	70
7	
8	
9	

Secondary Clustering

* Secondary Clustering:

→ If 2 keys are mapped onto the same starting location on the hash table, then they will unnecessarily follow the same path in a quadratic manner, because of this search time complexity will increase.

→ This is k/a secondary Clustering.

To remove this, we use double hashing.

* Search Time Complexity:
  Best Case → O(1)
  Worst Case → O(m)

* Insertion Time Complexity:
  Best Case → O(1)
  Worst Case → O(m)

* Deletion: Deletion is difficult to implement and create some problem but we can manage them by string any special symbole like "$".

③

Double Hashing:

m = 10 (0 - 9)
$hf_1(k) = k \bmod m$
$hf_2(k) = (k \bmod (m-2)) + 1$

keys = 25, 98, 57, 75, 97, 18, 78, 65, 46.

$DH(key, i) = (hf_1(k) + i * hf_2(k)) \bmod m$
$i = 0, 1, 2, \ldots 9(m-1)$

DH(25,0) = ((25%10) + 0 * hf2(k)) %10
= 5%10 = 5

DH(98,0) = ((98%10) + 0 * hf2(k)) %10
= 8 %10 = 8

DH(57,0) = ((57%10) + 0 * hf2(k)) %10
= 7 %10 = 7

DH(75,0) = ((75%10) + 0 * hf1(k)) %10
= 5%10 = 5 ✓

DH(75,1) = ((75%10) + 1 * (75%8)) %10
= (5 + 1) %10 = 9

DH(75,2) = ((75%10) + 2 * (75%8)) %10
= (5 + 6) %10 = 1

DH(97,0) = ((97%10) + 0 * hf2(k)) %10
= 7 ✓

DH(97,1) = ((97%10) + 1 * (97%8 + 1)) %10
= (7 + 2) %10 = 9 ✓

DH(97,2) = ((97%10) + 2 * (97%8 + 1)) %10
= (7 + 4) %10 = 1

DH(18,0) = ((18%10) + 0 * hf2(k)) %10
= 8 ✓

DH(18,1) = ((18%10) + 1 * (18%8 + 1)) %10
= (8 + 3) %10 = 1 ✓

DH(18,2) = (8 + 2 * (2 + 1)) %10
= 14%10 = 4

DH(78,0) ✓

DH(78,1) = ((78%10) + 0 * hf2(k)) %10
= 8 ✓

DH(78,2) = ((78%10) + 2 * (78%8 + 1)) %10
= 8 + 14 %10 = 2 ✓

~~DH(78,3) = 8 + 2~~

DH(65,0) = ((65%10) + 0 *) %10 = 5 ✓

DH(65,1) = (5 + 1 * 1) %10 = 7 ✓

DH(65,2) = (5 + 2 * 2) %10 = 9 ✓

DH(65,3) = (5 + 3 * 2) %10 = 1 ✓

DH(65,4) = (5 + 4 * 2) %10 = 3

DH(46,0) = 6

0	
1	~~75~~ 97
2	78
3	65
4	18
5	25
6	46
7	57
8	98
9	75

Total collisions = 11

Search Time Complexity:

Best Case → O(1)

Worst Case → O(m)

Insertion Time Complexity:

Best Case → O(1)

Worst Case → O(m)

→ Deletions may create some problem, but we can manage them by storing the special symbol. like "∅".

Deletion ~~Time~~ Time Complexity:

Best Case → O(1)

Worst Case → O(m)

CONCLUSION:

→ By using double hashing ~~or~~ with perfect hash fun" and if $n \leq m$, then, we can achieve worst case search time complexity as O(1).

NOTE

① The expected no. of probes in an unsuccessful search in open addressing techniques is: $\frac{1}{1-\alpha}$

② The expected no. of probes(attempts) in successful search in open addressing technique is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

"α" ⇒ load factor = $\frac{n}{m}$ $0 \leq \alpha \leq 1$