Data Structures

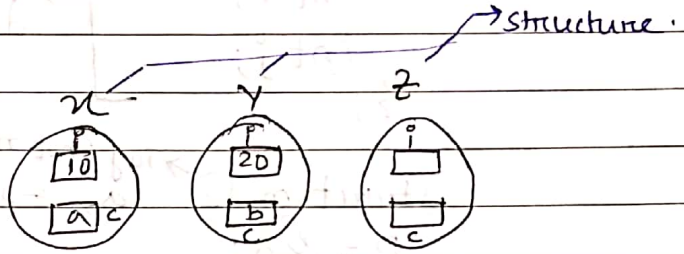
GATE CSE NOTES

Structures

- Introduction of Structure:

```
struct
{ int i;
  char c;
} x, y, z;
```

→ declaration



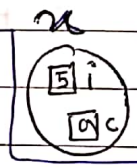
x.i = 10 | y.i = 20
x.c = 'a' | y.c = 'b'

→ member operaters

```
struct ex → Tag
{ int i;
  char c;
};
```

struct ex x,y,z;

struct ex x = {5, 'a'};

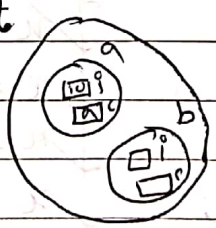



```
struct ex1 *
{
  struct ex a;
  struct ex b;
};
```

→ declaration.

struct ex1 t;

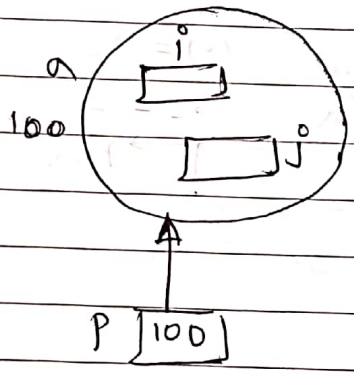
t.a.i = 10
a, a.c = 'a'
→ define.



- Example on Structures, arrays and pointers =

```
struct node
{ int i;
  int j;
};
```
→ declaration (no memory allocated)

```
struct node a, *p;
p = &a;
```
node → Tag (not necessary)



(*p).i → access of member of structure using pointer.

a.i → access by name of structure.

p→i same as (*p).i

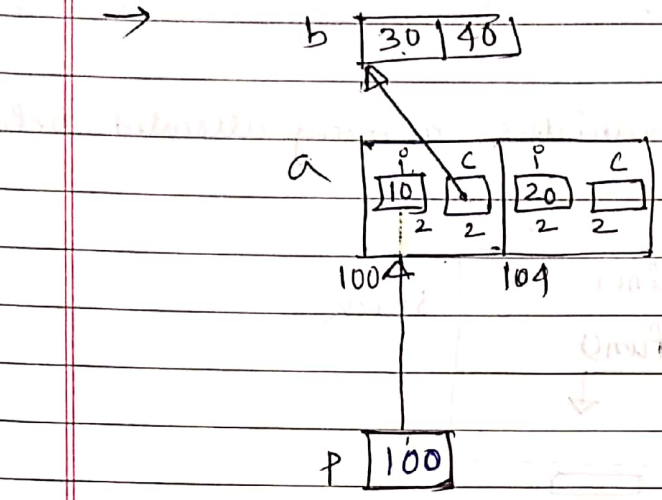
→ structures can be pass to a function as well as return by a function.

struct node fun (struct node n1, struct node n2);

Example:

```
struct node
{ int i;
  int *c;
};
struct node a[2], *p;
int b[2] = {30,40};
p = &a[0];
a[0].i = 10; a[1].i = 20;
a[0].c = b;
```

- ++p → i
- x = (++p) → i
- x = (p++) → i
- x = *p → c
- x = *p → c++
- x = (*p → c)++
- x = *p++ → c.



$x = (++(p \to i))$ / $x = 11$

$x = (++p) \to i$ / $x = 20$

$x = (p++) \to i$ / means $p \to i$ means $p = p+1$ / $x = 10$

$x = (*(p \to c))$ / $x = 30$

$x = (*(p \to c))++$ / $x = 30$ (For post increment $x = 31$).

$x = (*p \to c)++$ / $x = 30$.

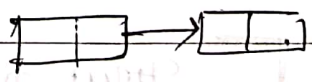
$x = (*(p++) \to c))$ / $x = 30$

- **Self referential structures:**

```
struct ex
{ int i;
  struct ex *link;
};
struct ex abc;
```

example of self referential structure =

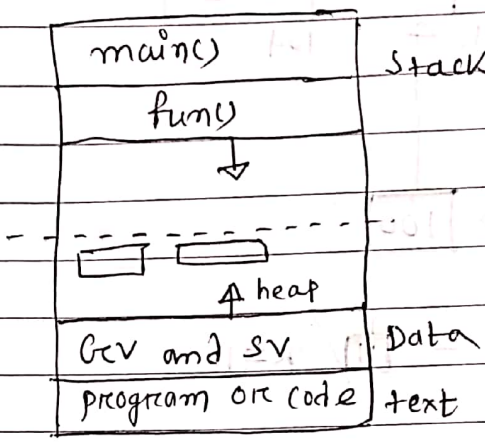
(1) link list.



(2) tree

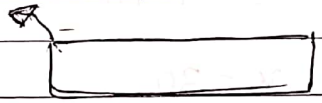
- **Malloc** =

→ static and global variables memory allocated before run the program.



10

```
void * malloc (int);
```



malloc function call make a space ~~in~~ Dinamicly in the heap of process and then return the starting address of this ~~space~~ location.

```
int *p = (int*) malloc (2);
```

```
void * malloc (sizeof(int));
```
— use this one

```
int*p=(int*) malloc (size of (2))
```
↳ type casting

syntax (which we to make struct and get pointer)

```
struct node
{ int i;
  struct node *l;
};
```

```
struct node* p = (struct node*) malloc (sizeof (struct node))
```

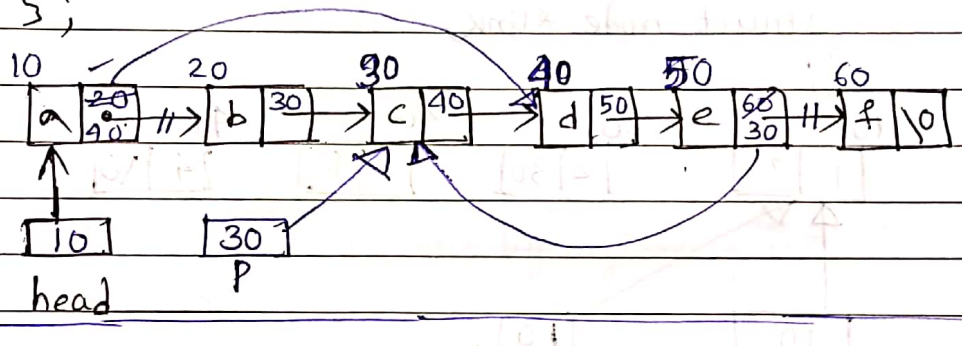
```
malloc (sizeof(struct node));
```

Linked-List

- Introduction of Single linked list :

→ single linked list contain nodes, and this node generaly structured ~~tt~~ and which are self ~~referrancial~~ referrencial.

```
struct node
{
  char data;
  struct node *link;
};
```



→ Every node has only one link (pointer) so, that this is called single linklist.

struct node *head;

→ linklist are sequential access.

→ going to any particulare number structere (link list) take O(n) time.

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

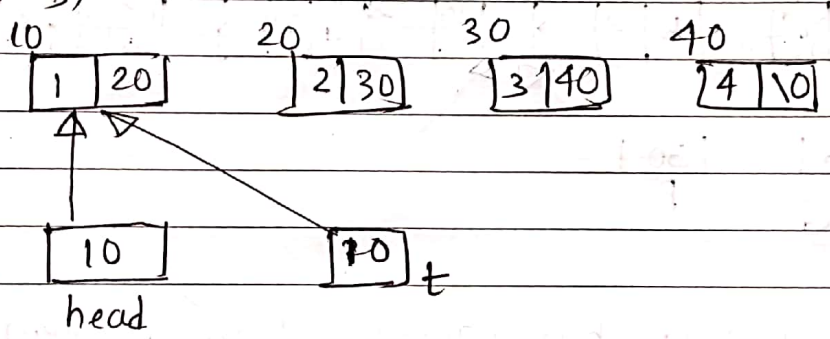
Output : "d"

~~Traversing of~~

In link list generally perform three operations —

- Traversing. (traverse the Entire list).
- inserting. (creat new node and insert it).
- delete. (delete a node).

```
struct node
{
  int i;
  struct node *link;
};
```

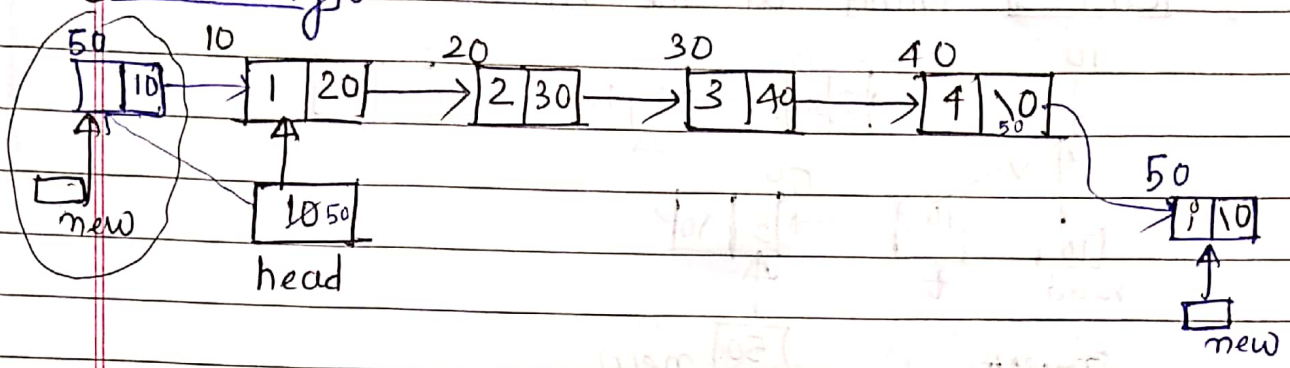


① Traversing :

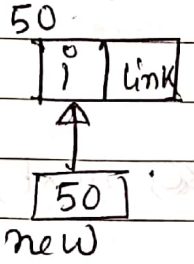
```
struct node *t;
  t = head;

while (t != NULL)   ≡ while (t)
{printf ("%d", t->i)

  t = t->link. }
```

output : 1 2 3 4

② Inserting :



struct node *new = (struct node *) malloc (sizeof (struct node))



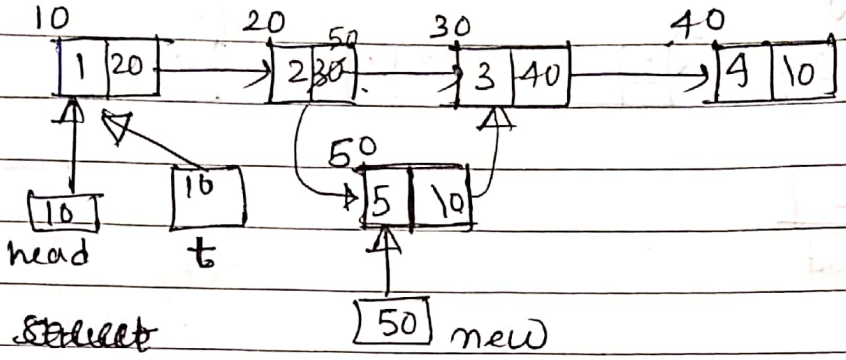
Case-1 Insert at the ~~bg~~ beginning =

```
new → link = head
head = new
```

Case-2 insert at the end =

```
struct node *t = head;

while (t → link != NULL)  ≡  while (t → link)
{ t = t → link; }

t → link = new;
New → link = NULL;
```

Case-3 Insert at the middle.



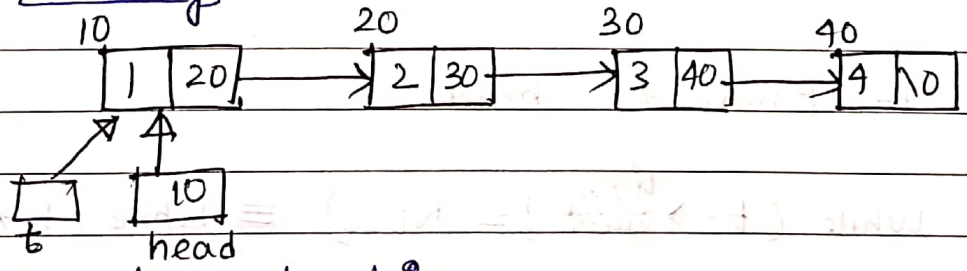
~~Struct~~

```
Struct node *t; = head;
while (t -> i != 2)
{ t = t -> next; }
        (link)
```

~~t -> next = new;~~
~~(link)~~

```
new -> link = t -> link
t -> link = new;
```

③ Deleting :



Delete from head :-

```
Struct node *t = head;
head = head -> next;
          link
free(t);
```

(to delete ~~the~~ first one)

Recursive Condition Checking =

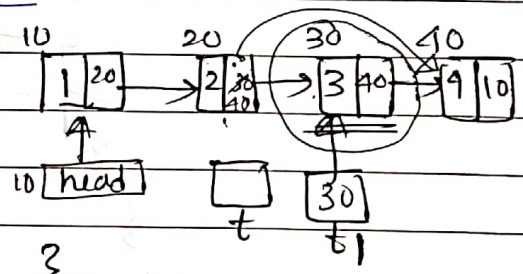
```
if (head == NULL)
    Return
if (head -> next == NULL)
    free (head)
```

before deletion check this ~~one~~. (head are null or not)
(head → link is null or not)

- delete from tail :

```
struct node *t = head;
While (t -> link -> link != NULL)
{ t = t -> list; }

free (t -> list)
t -> list = null
```

- delete node which contain i = 3 :




```
struct node *t = head;

While (t -> list -> i != 3)
{ t = t -> list; }

struct node *t1 = t -> list;
t -> list = t -> list -> list;
Free (t1);
```

GATE-2005-2018

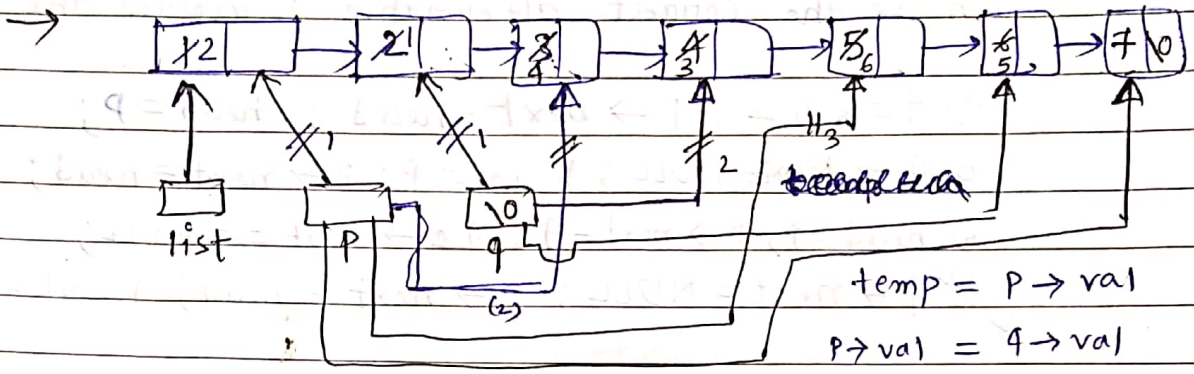
Question - 1

```
struct node
  {
    int val;
    struct node *next;
  };

void rearrange (struct node *list)
  {
    struct node *p, *q;
    int temp;
                    (or)
    if (!list || !(list -> next)) return;
    p = list; q = list -> next;

    while (q)   ≡ (while (q != NULL))
    {
      temp = p -> val; p -> val = q -> val;
      q -> val = temp; p = q -> next;
      q = p ? p -> next : 0;
    }
  }
```

(list == NULL) ≡ (!list)

1 2 3 4 5 6 7

(A) 1,2,3,4,5,6,7.
(B) 2,1,4,3,6,5,7.
(C) 1,3,2,5,4,7,6.
(D) 2,3,4,5,6,7,1.

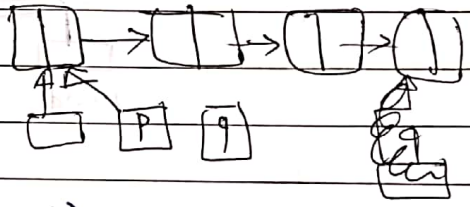


2 1 4 3 6 5 7

Gate-2010

Question-2

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




```
Node * move-to-front (Node *head)
{
  Node *p, *q;
  if((head == NULL) || (head → next == NULL))
    return head;

  q = NULL;
  p = head;
  while (p → next != NULL)
  {
    q = p;
    p = p → next;
  }
  (blank)
  return head;
}
```

Choose the correct alternative to replace the blank

a) q = NULL ; p → next = head ; head = p ;

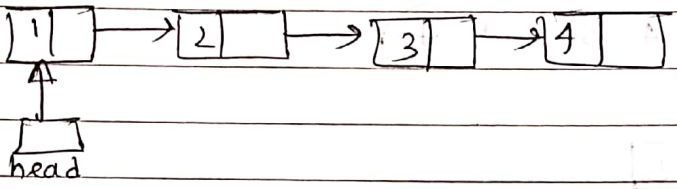
b) q → next = NULL ; head = p ; p → next = head ;

c) head = p ; p → next = q ; q → next = = NULL ;

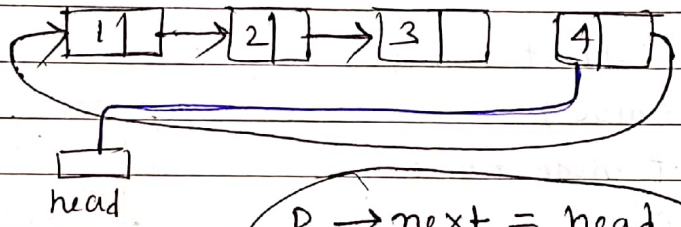
d) q → next = NULL ; p → next = head ; head = p ;

→

(I)



(II)



p → next = head

head = p

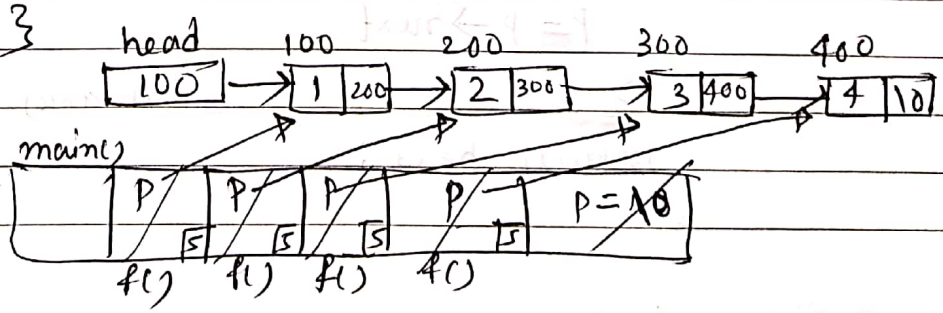
q → next = NULL

- **printing the elements of single linked list using recursion**

①

```
void f(struct node *p)
{
  if (p)
  { printf("%d", p -> data );
    f(p -> link)
    {
}
```

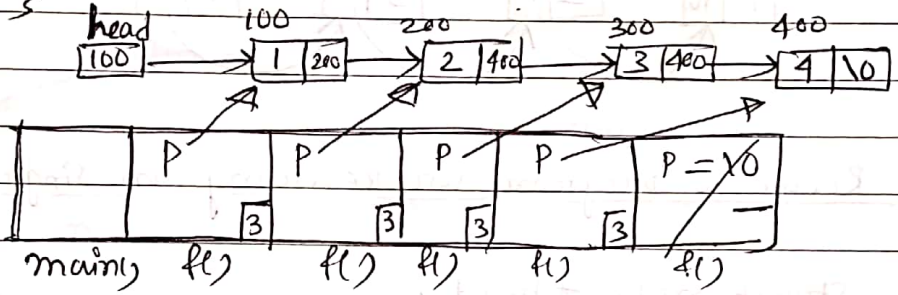
→



output: 1 2 3 4

(2)

```
void f(struct node *p)
{
1   if (p)
    {
2       f(p->link);
3       printf("%d", p->data);
    }
}
```



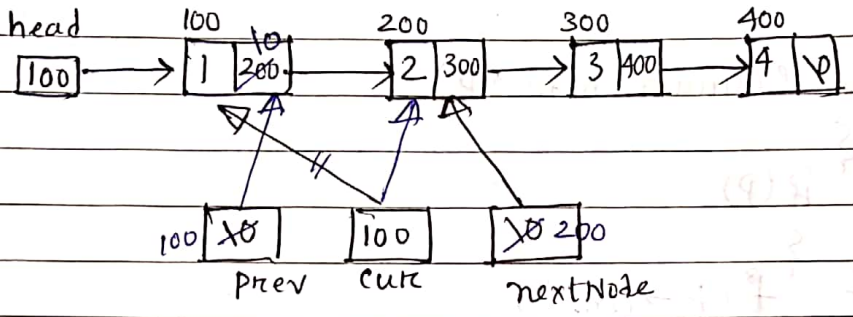
o/p: 4 3 2 1

- **Reversing an single linkedlist using iteration program**

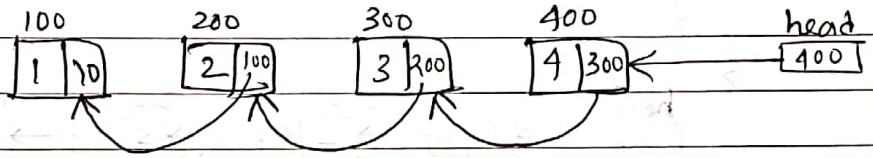
(1)

```
Struct node
{
    int i;
    struct node *next;
};

struct node *reverse(struct node *cur)
{
    struct node *prev = NULL, *nextnode = NULL;

While (cur) {
    nextNode = cur->Next;
    Cur->next = prev;
    prev -> cur;
  cur -> nextNode;
}
    return prev; }
```

```
main(){
head = reverse(head);
}
```

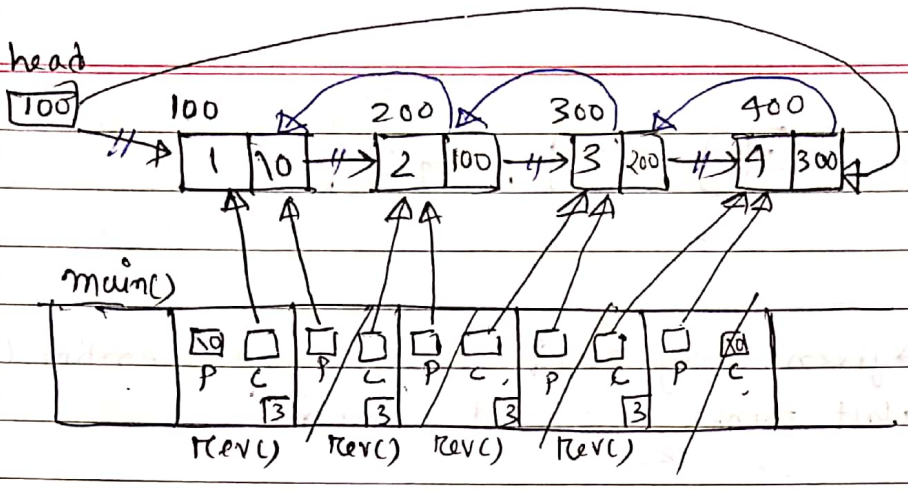


after operation



- Recursive program for reversing an single linked list =

```
struct node *head;

void reverse (struct node * prev, struct node *cur)
  {
  ① if (cur)
     {
  ②   reverse (cur, cur → link);
  ③   cur → link = prev;
     }
   else
      head = prev;
  } }

void main()
  {
     :
     reverse (NULL, head);
  }
```



GATE-2003

Question-3

In the worst case, the number of comparisons needed to search a single linked list of length 'n' for a given element is -

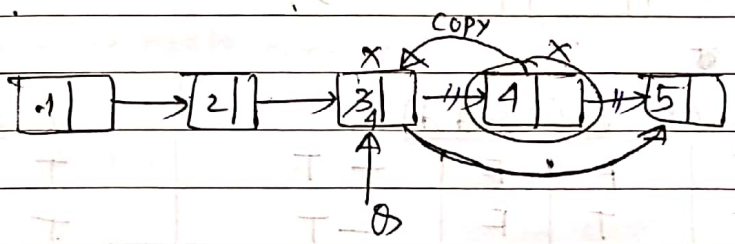
(A) $\log_2 n$ (B) $n/2$ (C) $\log_2 n - 1$ ~~(D)~~ n.

GATE-2004

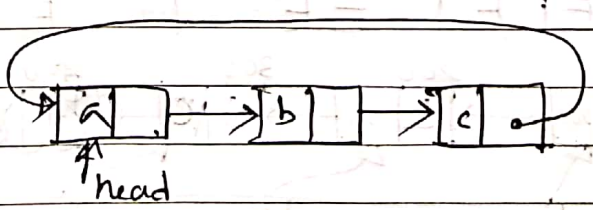
Question-4

Let 'p' be a single linked list, let 'q' be a pointer to an intermediate node 'x' in the list. What is the worst case time complexity of the best known algorithm to delete the node 'x' from the list?

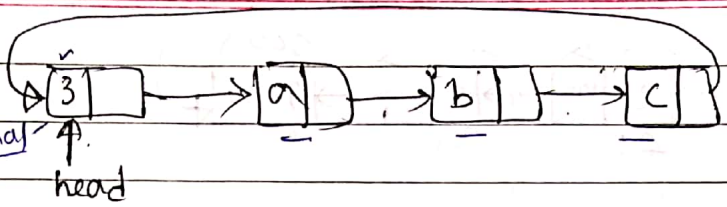
→ O(1).



- Circular linked list =




```
p = head;
while (p->next != head)   → to find the end of the list.
{ : }
```



Adv:→ given any point we can search entire list.
→ last pointer are not wasted.

Q-2003

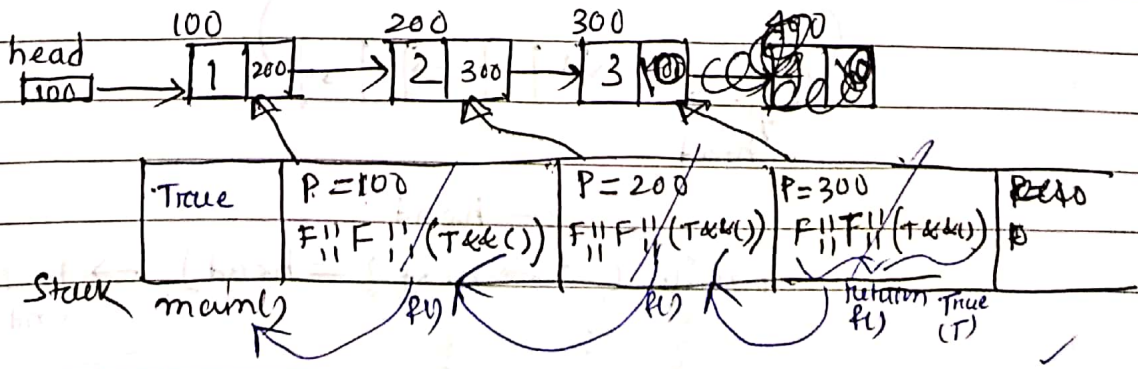
Question-5 (Checking if the list is in non decreasing)

```
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)));}
```

→ (OR) (OR) (AND)

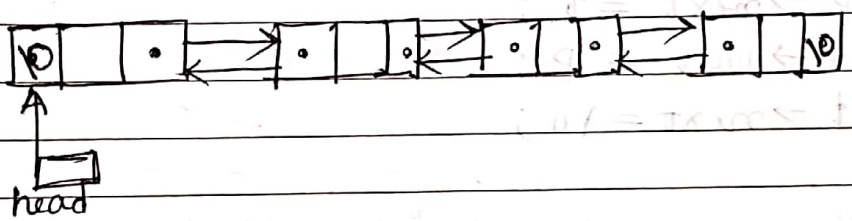
a || b || (c && d)

a	b	c	d	a or b or (c and d)	c and d
T	F	F	F	T	T
T	T	F	F	T	F
T	T	T	F	T	F
F	F	F	F	F	F

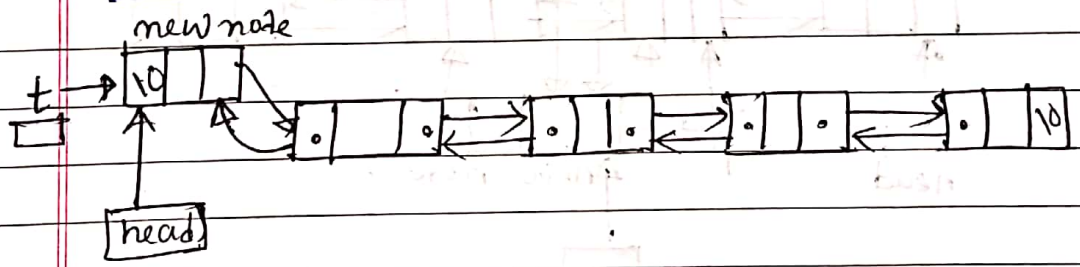


- Insertion into doubly linked list :

```
Struct node
{
  int i;
  struct node *prev;
  struct node *next;
};
```

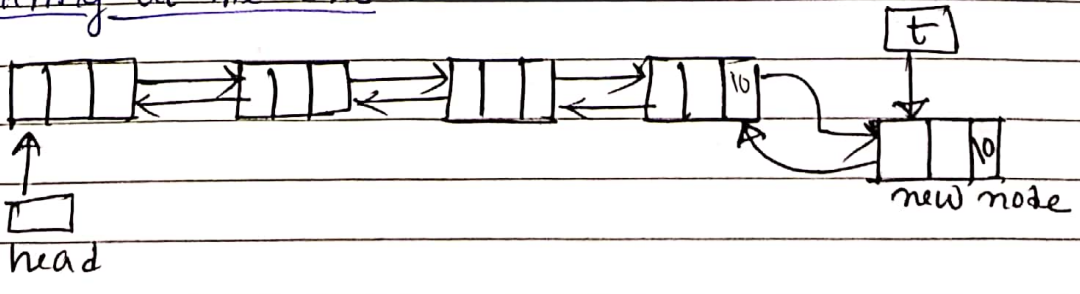


- Insert a node at ~~front~~ beginning :-




```
struct node *t;
t -> next = head;
head = t;
t -> prev = NULL;
head -> next -> prev = ~~t~~ head;
```

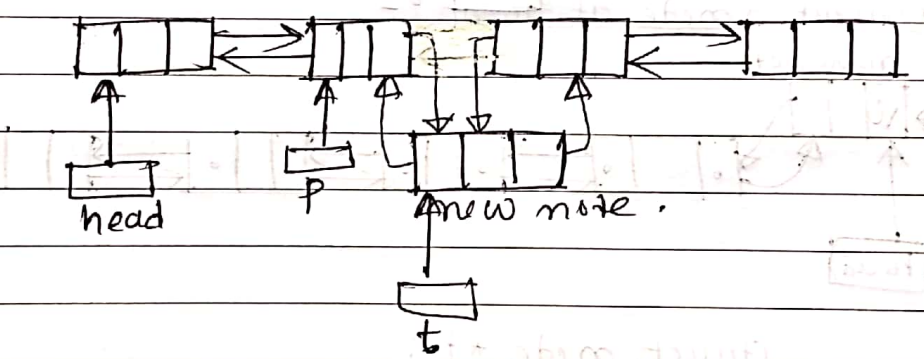
- Inserting at the end :-



~~while (~~

```
struct node *p;
p = head;

while (p->next)
  { p = p->next; }

p->next = t;
t->prev = p;
t->next = 10;
```

- Inserting the node at intermediate :-




```
struct nod *t;
```

(hold the link and then modify)

```
t->prev = p;
t->next = p->next;
p->next = t;
p->next->prev = t;
```

Stacks and Queues

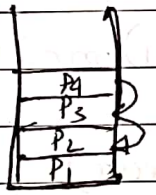
- **Introduction to stacks:**

(insert) → push the plate from the top and take the plate from the top. ← (pop)

(stack) (FILO, LIFO)

- **Applications of stack:**

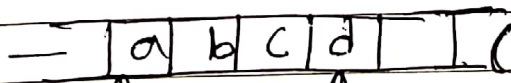
(i) Recursion.
(ii) IF → PF conversion.
(iii) Parsing
(iv) Browsers. →
(v) Editors. (use)
(vi) Tree traversals and graph traversal.



- **Implementation of a Queue using circular array =**

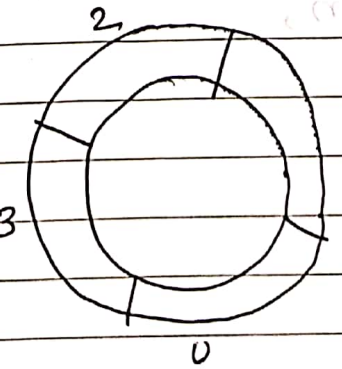
(enqueue) ← → (dequeue)

→ Insert the element from one side and delete the element from other side.



(Queue) (FIFO, LILO)

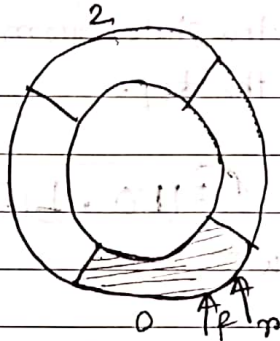
→ Insert from rear.
→ delete from front.



circular queue.

If → In circular queue, if the size of an array 'n' then we use '(n-1)' amount of size.

⇒ ~~→ one less~~



circular queue.

→ should a blank space.

① Inserting an item in queue =

```
enqueue(item)
{
  rear = (rear+1) mod n;

  if(front == rear)
  { pf("Q is full");

    if(rear == 0)  rear = n-1;
    else
Else rear = rear-1;
    return;
  }

  else {
      q[rear] = item;
      return;
      }
}
```

② delete an item from queue :-

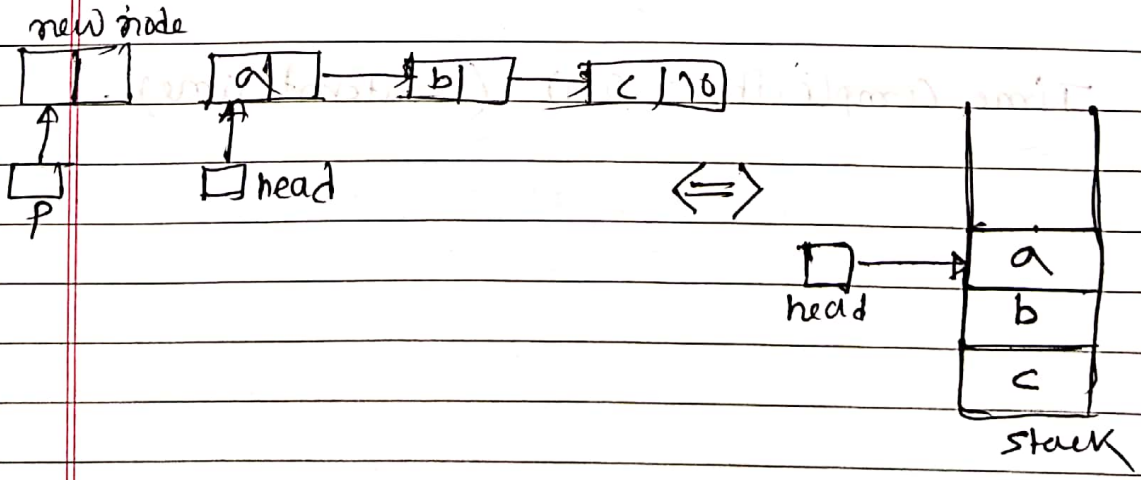
```
int Dequeue()
{
  if (front == rear)
    {
      Pf ("Q is empty");
      return -1;
    }
  else
    {
      front = (front+1) mod n;
      item = q[front];
      return item;
    }
}
```

Over flow :
(rear+1) mod n == front

Under flow
front == rear

- Linked List Implementation of stack :

```
struct node
{
  int i;
  struct node *link;
};
```



(1) Inserting a new item =

```
push (int item)
{
  struct node *p = (struct node*) malloc (sizeof (struct node));
  if (p == NULL) { pf("error of malloc"); return; }

  p → data = item;
  p → link = ~~head~~ NULL;
  p → link = head;
  head = p;
}
```

Time complexity = O(1) (constant time)

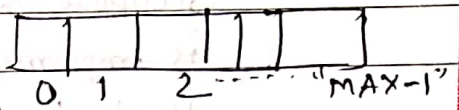
(2) deletion a item from stack =

```
int pop()
{ struct node *p;
  int item;
  if (head == NULL) { pf ("Under flow"); return -1; }

  item = head → i;
  p = head;
  head = head → next;
  free (p);
```

Time complexity = O(1) (constant time).

- Implementation of stack using array =

```
int stack [MAX];
int top = -1; //(stack Empty)
```



- puring a item =

```
Void push (int item)
{
   if (top == MAX-1)
      pf("overflow");

   else {
      top = top+1;
      stack[top] = item;
   }
   return;
}
```

stack [++top] = item;

- poping a item =

```
int pop()
{ int temp;
  if (top == -1)
  { pf("underflow");
    return -1;
  }
  else
  {
     temp = stack[top];
     top = top-1;
  }
  return temp;
}
```

temp = stack [top--];

→ Time complexity (push, pop) = O(1) Time.

GATE-2012

Question-1

Suppose a circular queue of capacity (n-1) elements is implemented with an array of 'n' elements. Assume that the insertion and deletion operations are carried out using 'REAR' and 'FRONT' as array index variables respectively, Initially, REAR=FRONT=0. The conditions to detect queue ~~with~~ full and queue empty are —

→

Queue Full:

(REAR+1) mod n == FRONT.

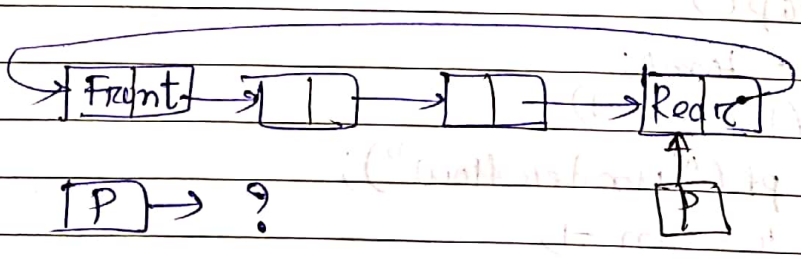
Queue Empty:

REAR = FRONT

GATE-2004

Question-2

A circularly linked list is used to represent a Queue. A single variable 'p' is used to access the Queue. To which node should 'p' point to such that both the operations enQueue and deQueue can be performed in constant time?



a) Read node
b) FRONT node
c) not possible.
d) node next to Front.

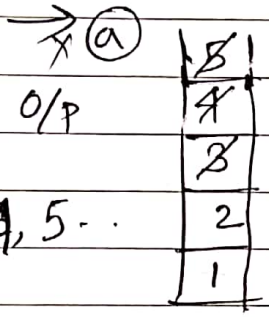
→

Question-3

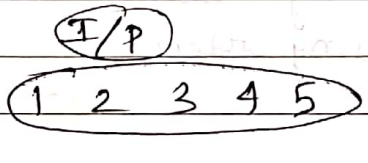
Which of the following permutations can be obtained in the o/p (in the same order) using a stack assuming that the i/p is the sequence 1, 2, 3, 4, 5 in the order?

✗ a) 3,4,5,1,2. ✓ b) 3,4,5,2,1.
✗ c) 1,5,2,3,4. ✗ d) 5,4,3,2,1,2.

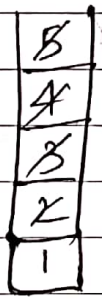
✗ (a)



o/p
3,4,5...

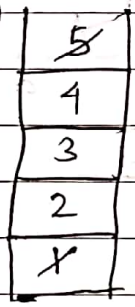


✓ (b)



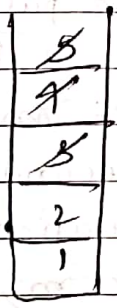
o/p
3,4,5,2,1

✗ (c)



1,5,...

✗ (d)



5,4,3,

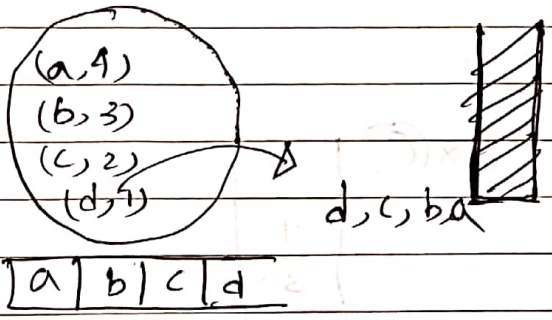
G-1997

Question - 4

A priority queue 'Q' is used to implement a stack 's' that stores characters PUSH(c) is implemented as ~~Insert~~ Insert (Q, c, K) where K is an appropriate integer key chosen by the implementation. POP is implemented as DELETE MIN(Q). For a sequence of PUSH operations, the keys chosen are in.

a) Non - decreasing order.
b) Non - increasing order.
c) strictly increasing order.
d) strictly decreasing order.

→

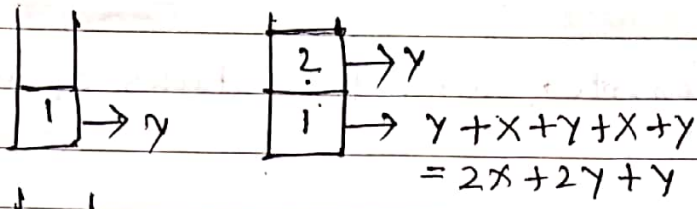


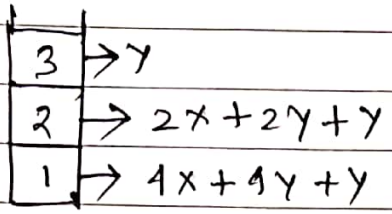
G-2003

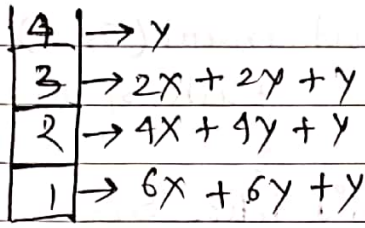
Question - 5

Let 's' be a stack of size $n \geq 1$. Starting with the ~~ent~~ empty stack, suppose we push the first 'n' natural numbers in sequence and then perform 'n' pop operations. Assume that push and pop operations take 'x' seconds each, and 'y' seconds elapse between the end of one such stack operation and the start of the next operation. For $m \geq 1$, define the stack-life of 'm' as the elapsed from end of push(m) to the start of the pop operation that removes 'm' from s. The average stack-life of an element of this stack is:

→







→ $2(n-1)x + 2(n-1)y + ny$

for n elements,

average life time,

$$\frac{2*\left(\sum_{i=1}^{n-1} i\right)x + 2\left(\sum_{i=1}^{n-1}\right)y + ny}{n}$$

$$\Rightarrow \frac{2*\frac{(n-1)(n-1+1)}{2}*x + 2*\frac{(n-1)(n-1+1)}{2}*y + ny}{n}$$

$$\Rightarrow \frac{n(n-1)x + n(n-1)y + ny}{n}$$

$$\Rightarrow (n-1)x + (n-1)y + y$$

$$\Rightarrow (n-1)(x+y) + y$$

$$\Rightarrow n(x+y) - x - y + y$$

$$\Rightarrow \boxed{n(x+y) - x}$$

06-2006

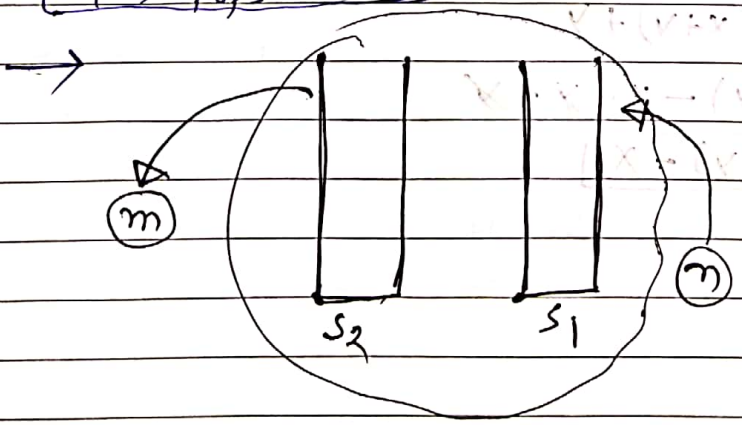
Question-6

Queue is implemented using two stacks S_1 and S_2 as given below:

```
void insert(Q, x) {
    push(S1, x);
    }
void delete(Q) {
    if (stack-empty (S2)) then
    if (stack-empty (S1)) then {
      printf ("Q is empty");
      return;
    }
  else while (!(stack-empty (S1)) {
        x = pop(S1);
        push(S2, x);
      }
  x = pop(S2);
  }
```

n → Inserts
m (≤n) → deletes
x → Pushes
y → Pops

What is relation b/w all this?



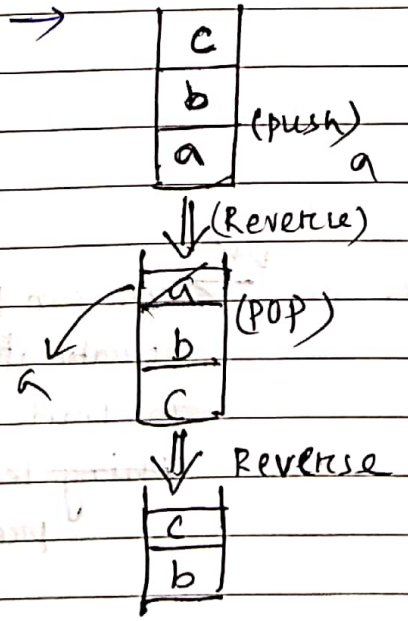
In Best case :	Worst case :
PUSH : $2m + (n-m)$ $= m+n$	PUSH : $2n$
POP : $2m$	pop : $2(n+m)$

$$m+n \leq x \leq 2n$$
$$2m \leq y \leq n+m$$

gate-2000 **Question - 7**

Suppose a stack implementation supports, in addition to PUSH, and pop, an operation "REVERSE" the order of the elements on the stack.

→ Implement a queue using the above stack implementation, show how to implement "ENQUEUE" using a single operation and "DEQUEUE" using a sequence of 3 operations.



Enqueue → push

Dequeue → Reverse → pop → Reverse

Infix → postfix

- **Infix to postfix Conversion algorithm :-** (Operator stack)

→ postfix related anything stack is required.

example :

① $a+b$ → Infix expression :
$ab+$ → postfix expression.

② $a+b*c$ → Infix exp
$= a+bc*$
$= abc*+$ → postfix expression.

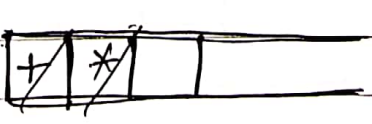
③ $a+b-c$
$= ab+-c$
$= ab+c-$

+ <. (
(<. +

④ $a+(b-c)$ − Infix
$= a+bc-$
$= abc-+$ — postfix.

⑤ $a+b*c$ − infix
$= a+bc*$
$= abc*+$ — postfix

⑥ $a + b * c$ − Infix.



$= abc*+$ − Postfix.

→ you can push an operator, * only when the inside one is having lesser precidence.

⑦ a * b + c — Infix expression.

Stack: [*] [+]

a b * c + — postfix expression.

⑧ a * b + (c − d) — Infix

[*] [+] [(] [−]

a b * c d − + — postfix expression.

Algorithm

a) Creat a stack.
b) For each character 't' in the i/p.

{
if ('t' is an operand)
output 't';

else if ('t' is a right parenthesis)
{
Pop and output tokens until a left parenthe-sis is popped (but don't o/p)

else // t is an operator or left parenthesis
{
pop and o/p tokens until one of lower priority than 't' is encounted or a left parenthesis is encounted or the stack is empty.
push t
}

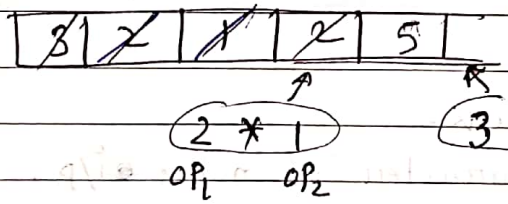
c) pop and o/p all the tokens untill the stack is empty.

Space or Time complexity — O(n)
Data structure used — stack
on PUSH only — operators.

Postfix → Infix

- Postfix evaluation algorithm = Operand stack

3+2*1 — Infix
3+21*
321*+ — postfix



push — only operand.

Algorithm for postfix evaluation =

1. Scan the postfix string from left to right.
2. Initialize an empty stack.
3. Repeat steps 4 and 5 till all the characters are scanned.
4. If the scanned character is an operand, push it onto the stack.
5. If the scanned character is an operator, and if the operator is unary, then pop an element

from the stack. If the operator is binary, then pop two elements from the stack. After popping the elements ~~from the stack~~ apply the operator to those popped elements. push the result on to the stack.

6) After all the elements are scanned, the result will be in the stack.

TREES

- Introduction to tree traversals:

- Traversing means → We want to see What is information present in Every node.

- Searching means → seaching for a particular node.

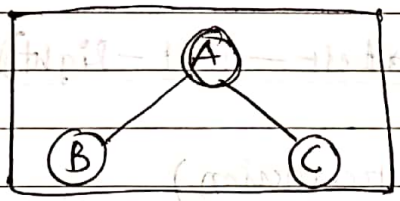
Methods of traversing

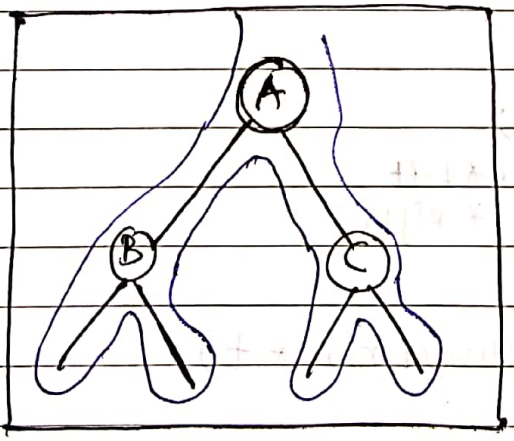
visit

2nd time → Inorder traversing. (Left-Root-Right) - BAC

1st time → preorder traversing (Root-Left-Right) - ABC

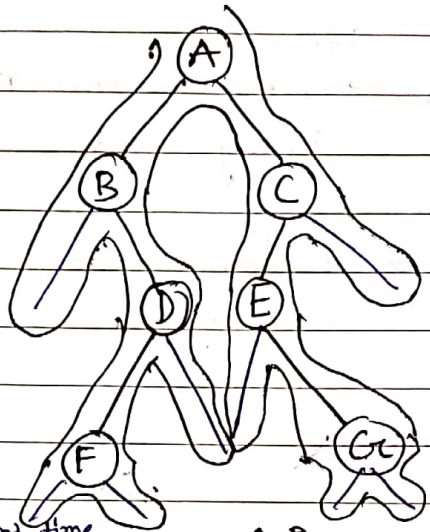
3rd time → postorder traversing. (Left-Right-Root) - BCA





1st time - A B C - (pre)
2nd time - B A C - (Inorder)
3rd time - B C A - (post)

example :



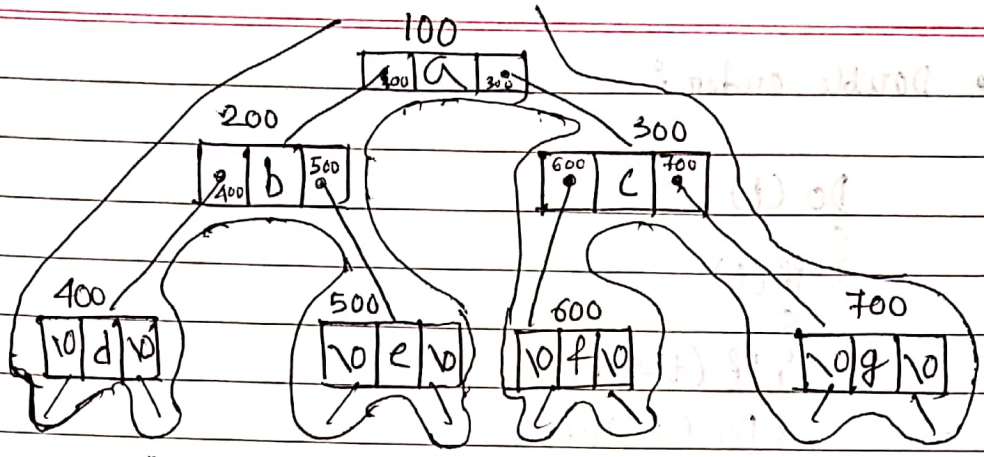
- Inorder (2nd time order) → B F D A E G C
- ~~post~~ preorder (1st time) → A B D F G E C
- postorder (3rd time) → F D B G E C A

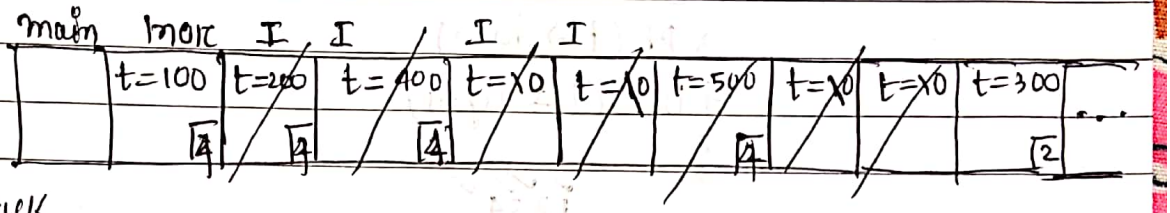
- Implementation of traversals and time and space analysis -

Inorder traversing → (left — root — Right).

Program : (Using recursion)

```
struct node
{  char data;
   struct node *left
               *Right;
}

void Inorder (struct node *t)
{
   if (t!=NULL)
   {
   1. Inorder (t->left);
   2. printf ("%c", t->data);
   3. Inorder (t->Right);
   }
}
```





Stack

O/p: dbeafcg

In/pre/post order traversal all take)

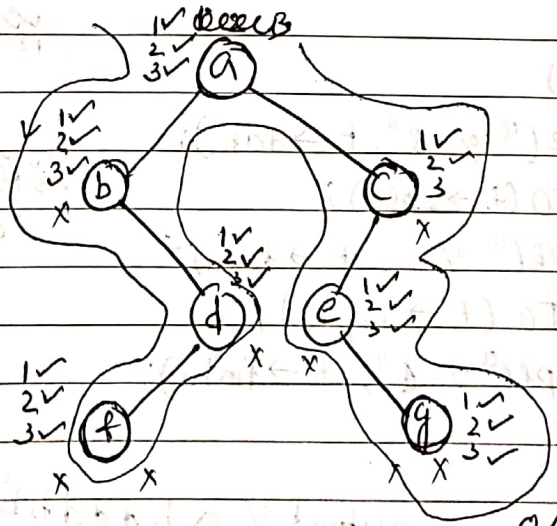
Time and space complexity = O(n).

{ Inorder → 1, 2, 3 (lines)
pre order → 2, 1, 3
post order → 1, 3, 2 }

- Double order traversal:

(one other method)

example: (Inorder) (root as input)

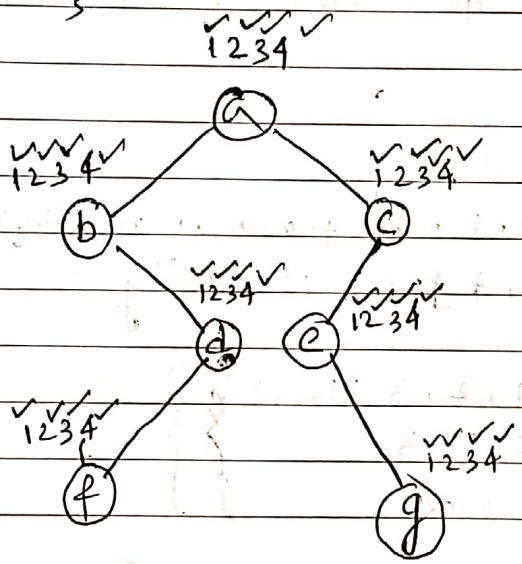



```
INORDER(t)
{
1. INORDER(t->Left)
2. pf
3. INORDER(t->Right)
}
```

O/p: b f d a e g c

- Double order :

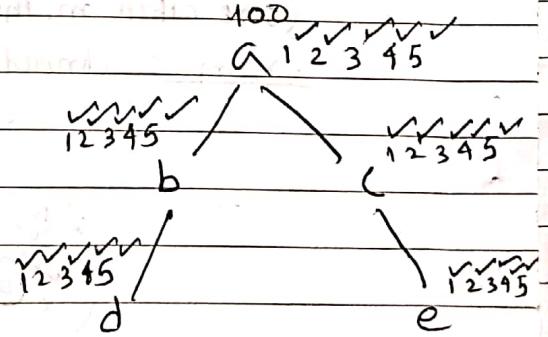
```
DO (t)
{ PF(t)
  {
  1 PF (t→data)
  2 DO (t→left)
  3 PF (t→data)
  4 DO (t→right)
  }
```



Output :
a b b d f f d a c e e g g c a

- Triple order traversal :

```
(100)
void TO (struct node *t)
{
  if (t)
  1. { pf("%d", t→data);
  2. TO (t→Left);
  3. PF("%c", t→data);
  4. TO (t→Right);
  5. pf("%d", t→data);
  }
}
```

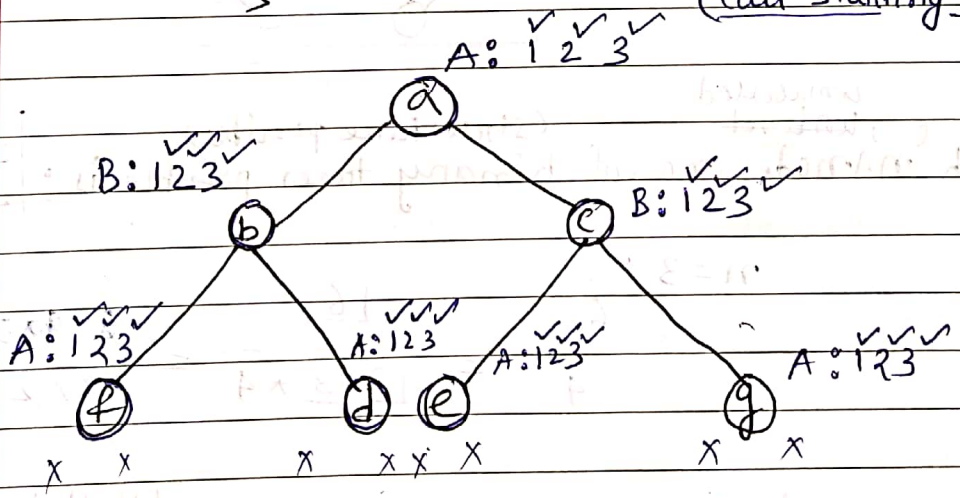


output : a b d d d b b a c c e e e c c a

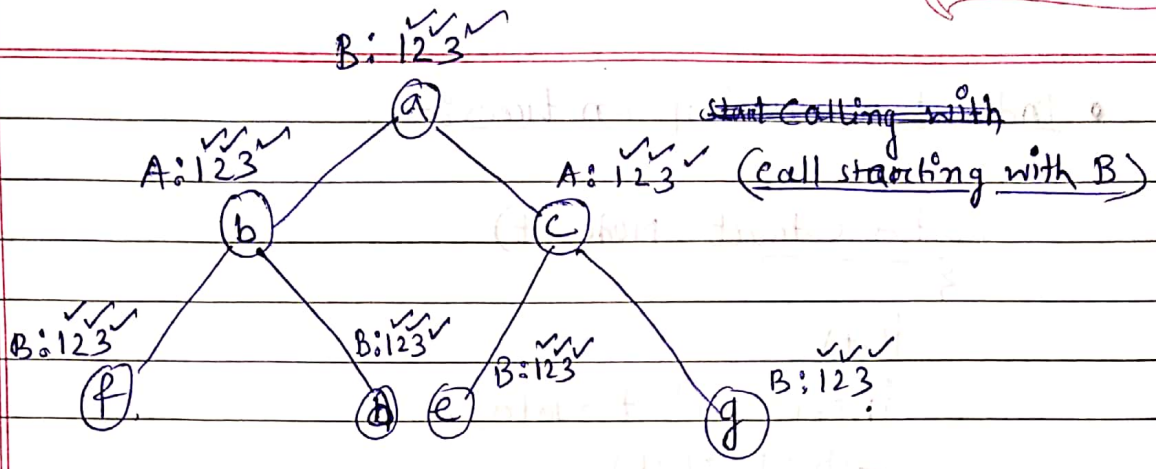
- **Indirect recursion on trees:-**

```
void A (struct Node *t)
{
  if(t)
  {1. pf("%d", t->data);
   2. B (t->left);
   3. B (t->right);
  }
}

void B (struct node *t)
{
  if(t)
  {
   1. A (t->left);
   2. pf("%d", t->data);
   3. A (t->Right);
  }
}
```

(call starting with A)



afbdecg — output.

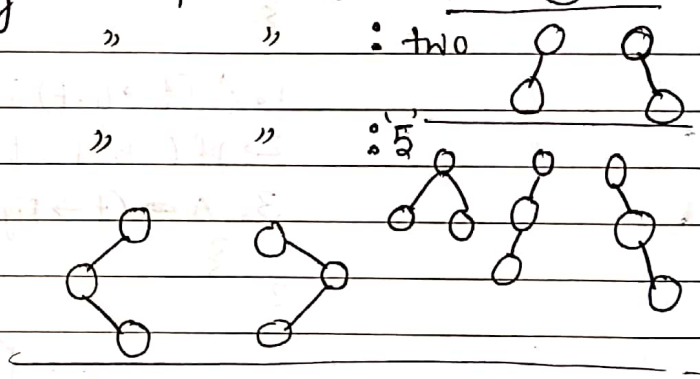


~~start calling with~~ (call starting with B)

bfdaceg → output.

(structures)

- **Number of Binary trees possible:**
  - Number of Binary tree possible with ~~unlevel~~ unlabelled node =
    1 node no. of binary trees possible is: one
    2 " " " " : two
    3 " " " " : 5



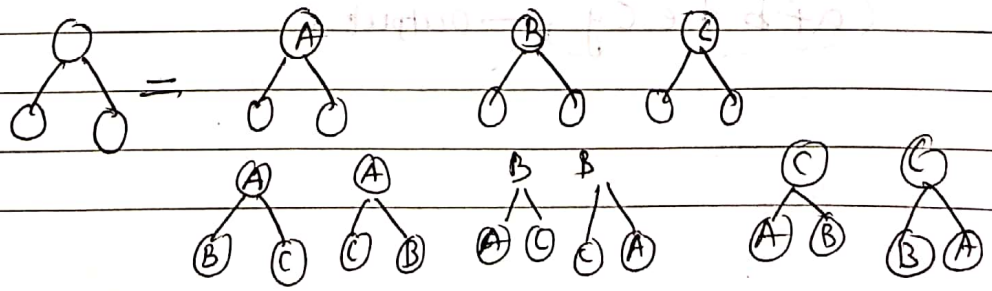
with 'n' unlabelled ~~unlevel~~ node no. of Binary trees possible is: (structure possible)

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

$n = 3$

$$\frac{6C_3}{4} = \frac{\lfloor 6}{\lfloor 3 \lfloor 3 * 4} = \frac{6 \times 5 \times 4}{3 \times 2 \times 4} = 5$$

- **Number of Binary tree possible with the ~~velevel~~ Labelled node :=**



With 'n' labelled nodes no. of binary tree possible are =

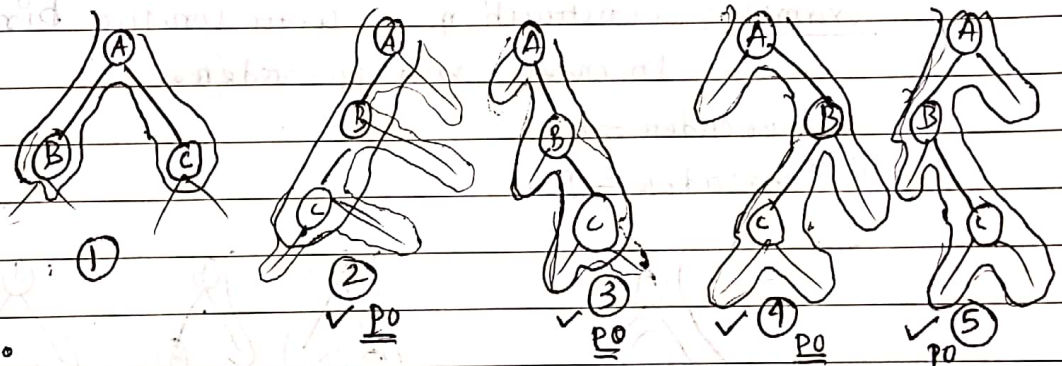
$$= \frac{{}^{2n}C_n * n!}{(n+1)}$$

Question type =
How many node possible?

Question:
All the Binary Trees with 3 nodes A, B and C which have preorder ABC (1st time) How many binary tree possible?

→ With 3 unlabelled node trees possible : 5
" 3 labelled " " : 5 × 12 = 30.

→ fig:



When:
preorder "ABC" && PostOrder is "CBA" —
(3rd time)

fig: ② ③ ④ ⑤ . (4-trees possible)

When: pre "ABC" (1) && post "CBA" (3) && Inorder "BCA" (2nd time visit) =

fig: ~~④~~ ⑤ (only - one tree possible)

(here one tree that satisfy all the condition)

→ ~~no. of tree possible with node~~

given { 'n' nodes
Preorder -

no. of trees possible is = $\frac{^{2n}C_n}{(n+1)}$

→ with given preorder and postorder you may get more than '1' binary tree.

*→ with given pre ~~and~~, post, and Inorder we always get only 'one' binary tree.

(~~unic~~) unique

example: construction of ~~bina~~ unique binary tree using Inorder and preorder.

(1st) preorder = ABC

(2nd) Inorder = BAC

→

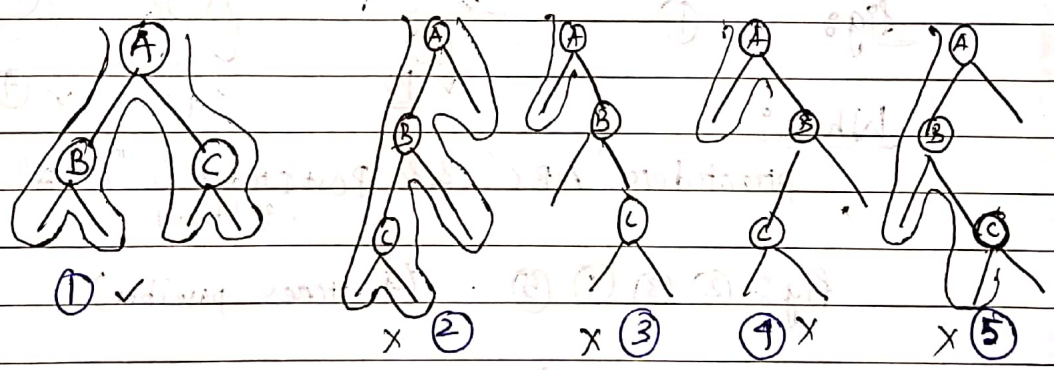


fig:

→ 5 binary tree possible with preorder ① ② ③ ④ ⑤.

→ ~~2~~ 1 BTS possible with (In and pre both order) fig - ①.

IN + PRE
IN + POST } → possible

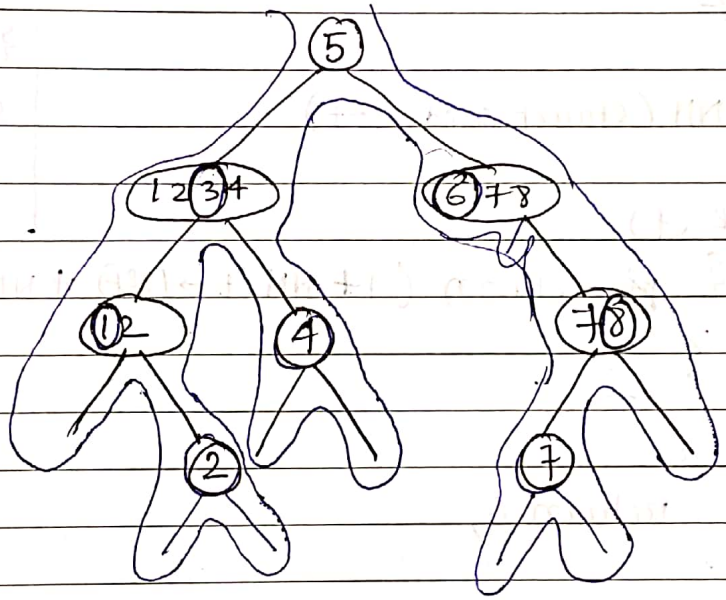
POST + PRE → not possible to find IN.

Question:

INORDER: 1, 2, 3, 4, 5, 6, 7, 8.
PREORDER: 5, 3, 1, 2, 4, 6, 8, 7.
POSTORDER: ?

→

IN – L (Root) R
Pre – (Root) L R

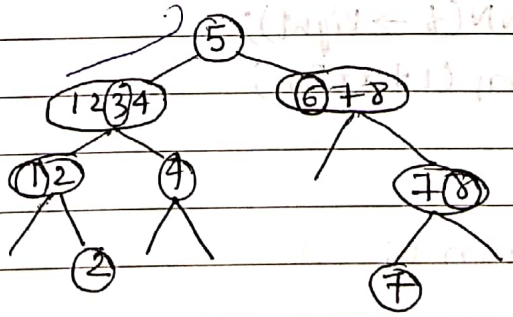


POSTORDER: 2 1 4 3 7 8 6 5

Question:

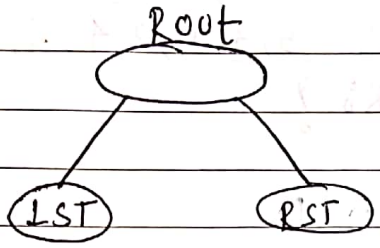
INORDER: 1 2 3 4 5 6 7 8 (L Root R)
POST: 2 1 4 3 7 8 6 5 (L R Root)
PRE: ?

→



PRE: 5 3 1 2 4 6 8 7.

• Recursive program to count the number of nodes :-



NN(T) = 1 + NN(LST) + NN(RST)
= 0; if T is NULL.

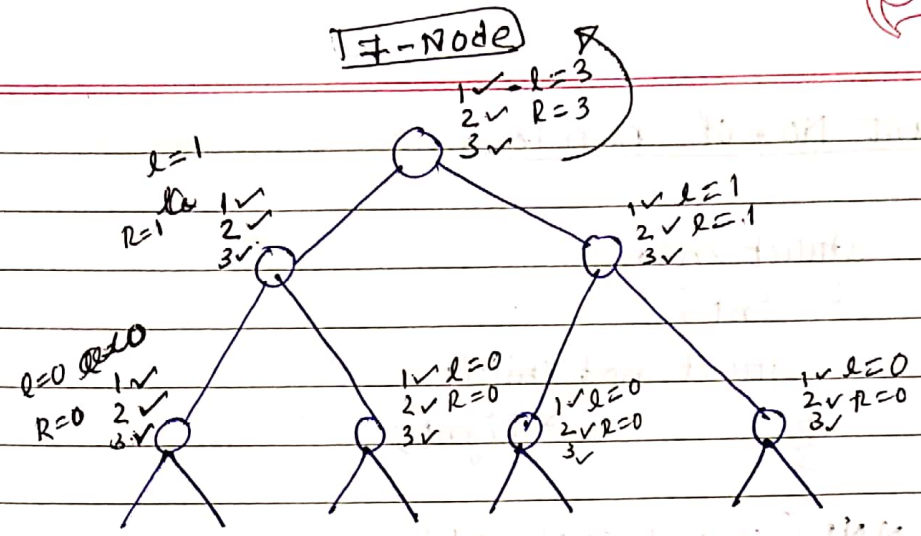
struct node
{int i;
struct node *
*left
*Right;
}

program :-

```
int NN(struct node *t)
{
  if(t)
  { return (1+NN(t->left)+NN(t->Right));
  }
  else
    return 0;
}
```

or

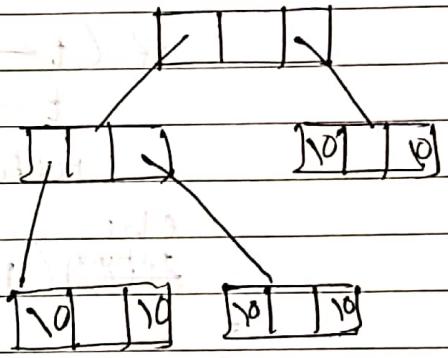
```
int NN(struct node *t)
{
  if(t)
  int l,R;
1. l = NN(t->left);
2. R = NN(t->Right);
3. return (1+l+R);
  }
else
  return 0;
}
```



- **Recursive program to count the no. of leaves and non leaves =**

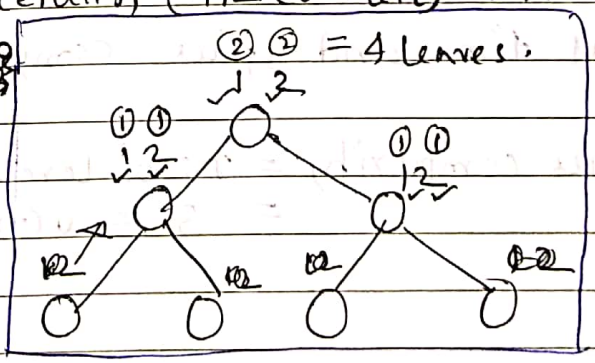
- **Count NO. of leaves :**

```
struct node {
    int i;
    struct node *left
                *right;
}
```



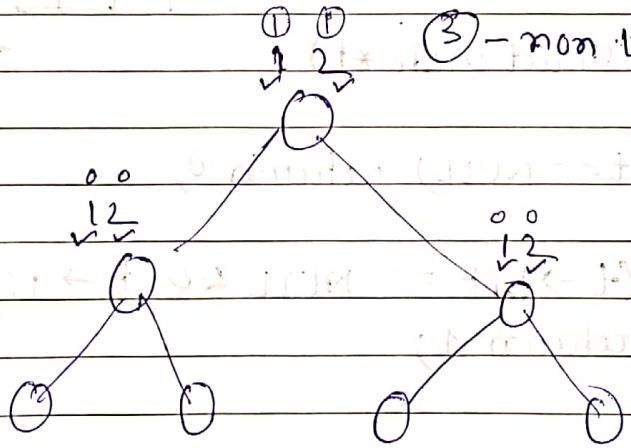
→ 3 leaves

```
int NL(struct node *t)
{
    if (t==NULL) return 0;

    if (t->left == NULL && t->Right == NULL)
        return 1;

    else
        return (NL(t->left) + NL(t->Right));
}
```



• Count No. of non leaves :

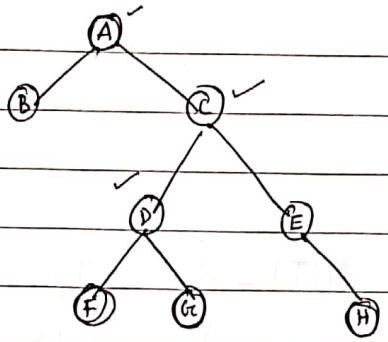
```
struct node
{ int i;
  struct node *left,
              *Right;
};

Int NNL(struct node *t)
{
  if (t == NULL) return 0;

  if (t->left == NULL &&
      t->Right == NULL)
    return 0;
  else return (1 + NNL(t->left) + NNL(t->Right);
}
```



⇝ Total time and space compuxity = O(n).

→ Space compuxity = no. of levels of tree.
= 3 = O(n).

- Recursive program to find the full nodes =



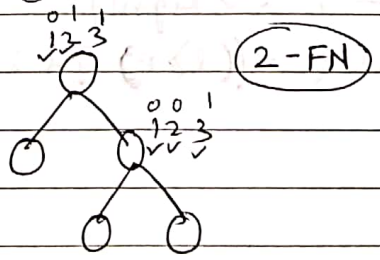
Total Full node : 3 (A,C,D)
Total node : 4 (A, C, D, E).

FN(T) = 0 ; T = NULL. (return)
= 0 ; T is a leaf.
= FN(T→LST) + FN(T→RST) ; if T has only one child.
= FN(T→LST) + FN(T→RST) + 1 ; if T is a full node.

Program :

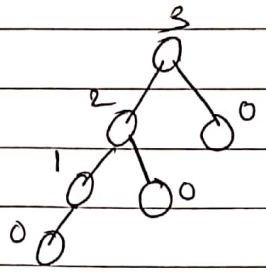
```
int FN (struct node *t)
{
   if (!t) return 0;
   if (!t->left && !t->right)
      return 0;

   return (FN(t->left) + FN(t->right) + (t->left && t->right) ? 1 : 0);
}
```

① ② ③



→ Total Time && space complexity is = O(n).

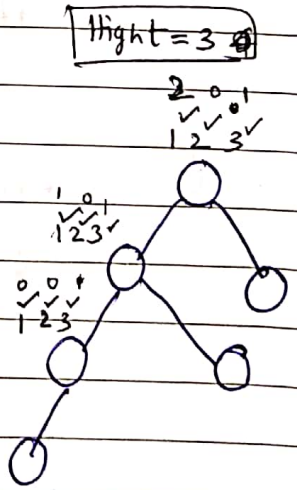
• Recursive program to find the height of a tree :-



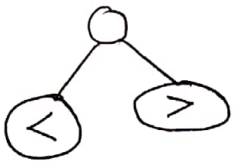
Recursive equation, $H(T) = \begin{cases} 0, & T \text{ is empty.} \\ 0, & T \text{ is leaf.} \\ 1 + \max(H(LST), H(RST)); & \text{otherwise.} \end{cases}$

Program:

```
struct node
{ int i;
  struct node * Left
              * Right;
}

int struct node *t;
int H(struct node *t) {
    int l, r;
    if (!t) return 0;

    if ( (!(t->left))&&(!(t->Right)))
      return 0;

    Else
      1. l = H(t->left)
      2. r = H(t->Right)
    3. return (1+((l>r) ? l : r));
}
```



→ Time and space complexity is = O(n).



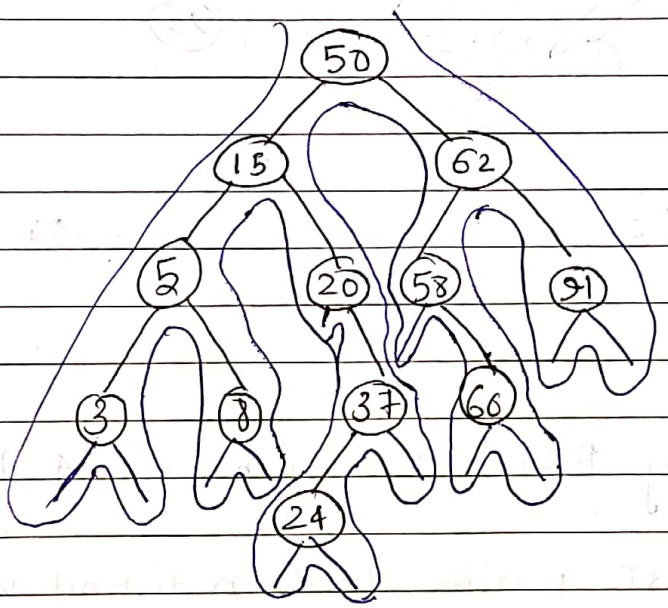
classmate
Date
Page

- **Introduction of to BST (Binary search Tree) =**

→ Binary search tree are a special kind of Binary tree. → Inorder is sorted order. (adv).

g-1996 **Question** – ①

50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24



→ The Inorder traversal of a BST is going to give sorted order of element.

Inorder: (2nd time) 3, 5, 8, 15, 20, 24, 37, 50, 58, 60, 62, 91. (assending order)

g-2005 **Question** – ②

Post Order: 10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29.

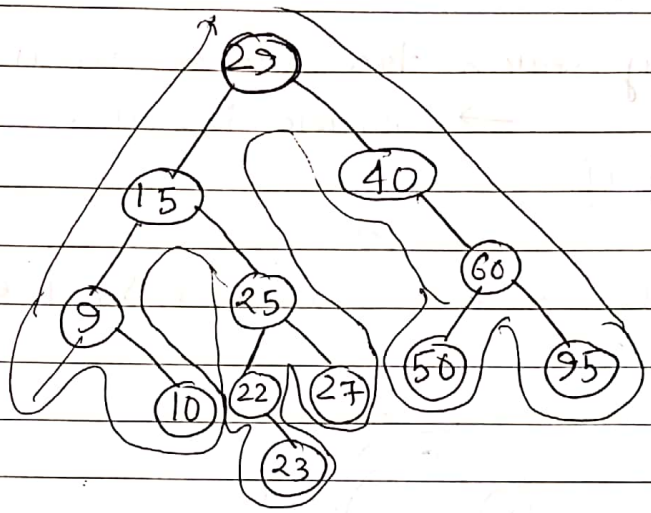
Inorder: ?

→ Inorder: 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95. (assending Order)

then find ~~post~~ preorder: ?

9 10 (15) 22 23 25 27 (29) (40) 50 60 95

Root



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

g-2005 Question-(3)

How many BSTs are possible with 4 distinct keys. value

→ No. of BST possible with n distinct keys are,

→ (same as Binary Unlabelled tree).

$$\frac{2n C_n}{n+1}$$

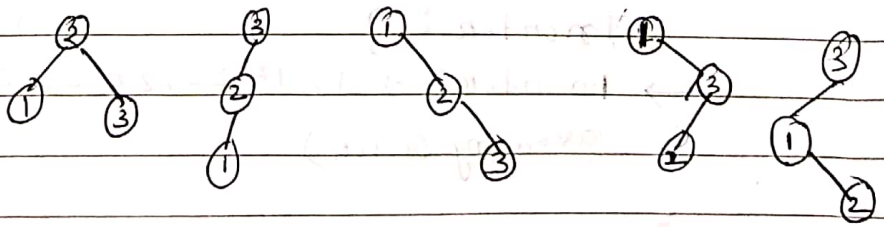
$$\frac{8C_4}{4+1} = \frac{\lfloor 8}{\lfloor 4 . \lfloor 4 . 5}$$

$$= \frac{8*7*6*5}{4*3*2*5}$$

= 14 (BST) possible.

∅ BSTs 3 distinct keys = 1 2 3

∅ ∅



g-2006

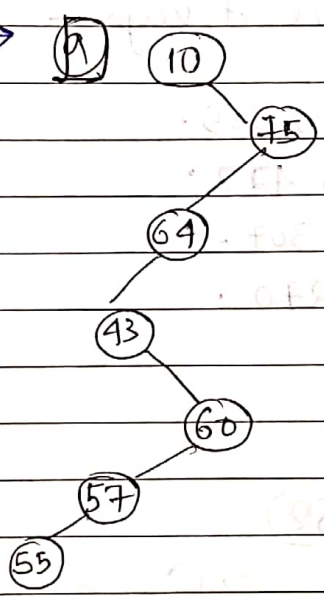
Question - (4)

(1 – 100) in BST search for 55 which of the following sequences cannot be the sequence of nodes examined ?

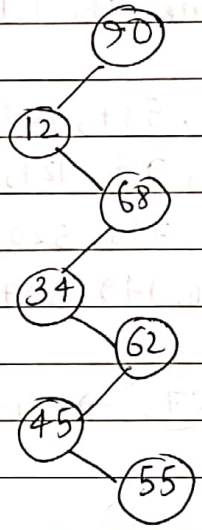
a) {10, 75, 64, 43, 60, 57, 55}
b) {90, 12, 68, 34, 62, 45, 55}
x c) {9, 85, 47, 68, 43, 57, 55}
d) {79, 14, 72, 56, 16, 53, 55}

(Inorder traversal ascending order)

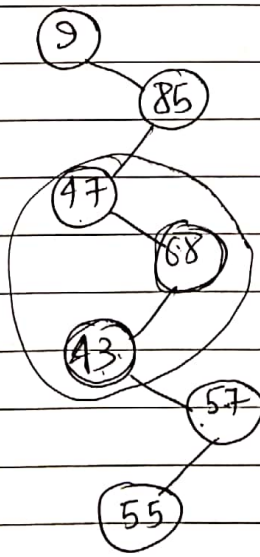
→ a)



b)



x c)



d)