Data Structure

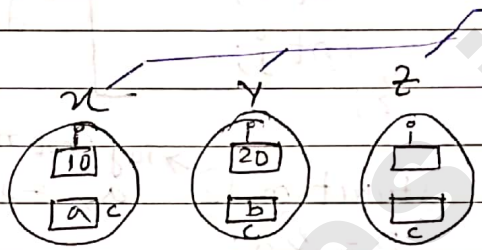
Gate Notes

Structures

- Introduction of Structure:

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

→ declaration



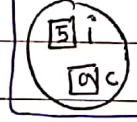
x.i = 10 | y.i = 20
x.c = 'a' | y.c = 'b'

→ member operaters

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

```
struct ex x, y, z;
```

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

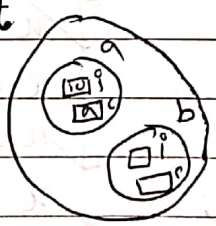



```
struct ex1
{
  struct ex a;
  struct ex b;
};
```

→ declaration.

```
struct ex1 t;
```

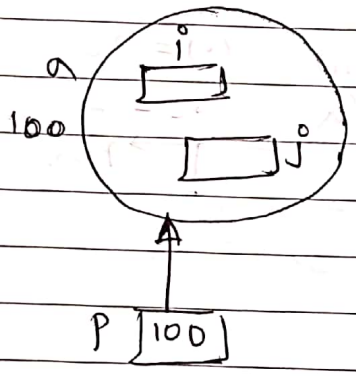
t.a.i = 10
a, a.c = 'a'
→ define.



- 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

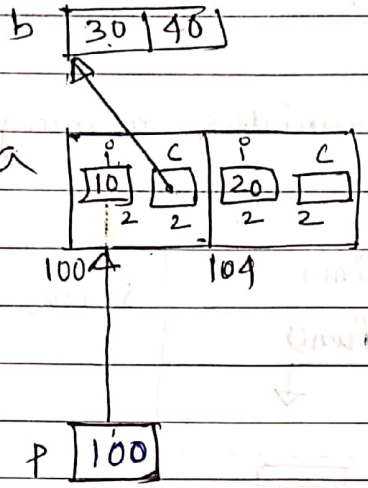
→ structures can be pass to a function as well as return by a function.

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



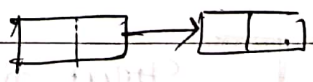
- $x = (++(p \to i))$ / $x = 11$
- $x = (++p) \to i$ / $x = 20$
- $x = (p++) \to i$ / means $p \to i$ means $p = p+1$ / $x = 10$
- $x = (*(p \to c))$ / $x = 30$
- $x = (*(p \to c))++$ / $x = 30$ (Post increment $x = 31$).
- $x = (*p \to c)++$ / $x = 30$.
- $x = (*(p++) \to c))$ / $x = 30$

- **Self referential structures:**

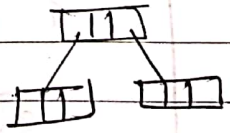
```
struct ex
{ int i;
  struct ex *link;
};
struct ex abc;
```

example of self referential structure =

(1) Link list.

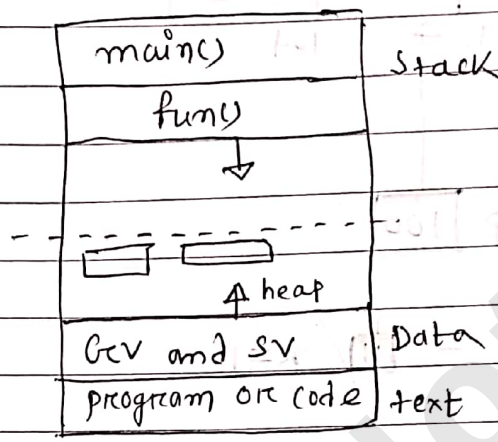


(2) tree



- **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 (sizeof(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))
```

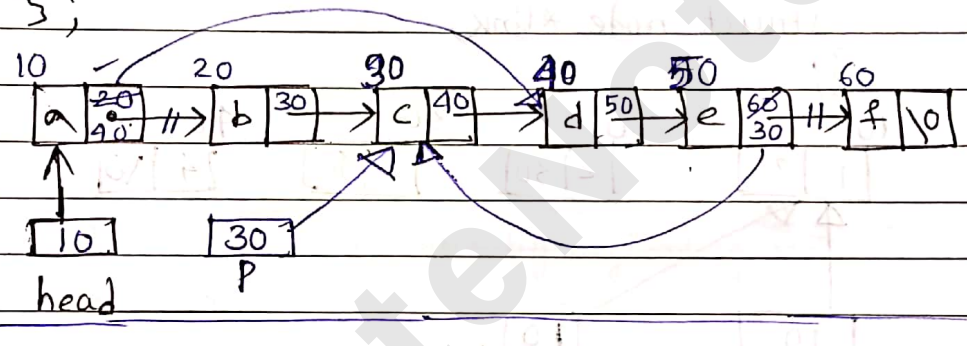
```
malloc(sizeof(struct node));
```

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);
```

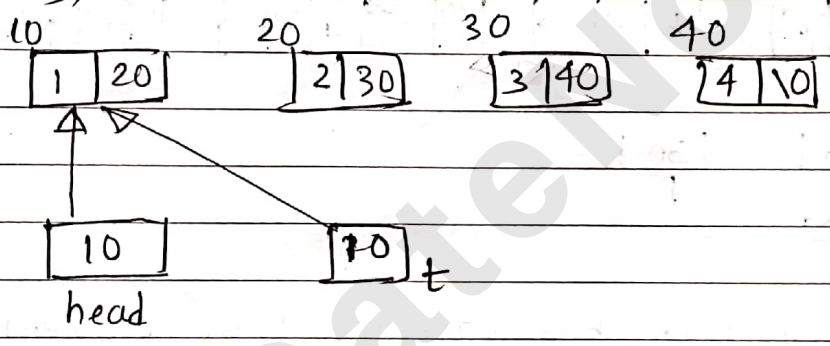
Output : "d"

Traversing of LL

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;
};
```

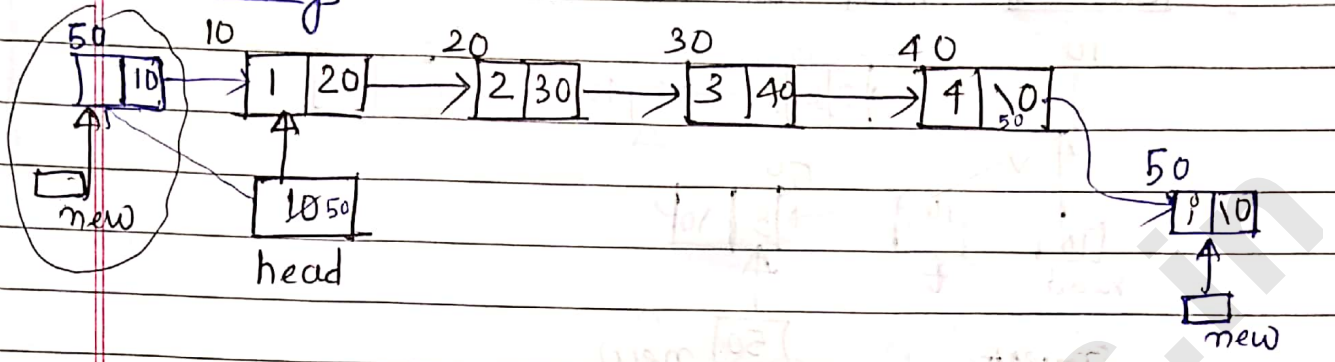


① Traversing :

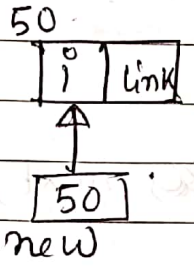
```
struct node *t;
t = head;

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))



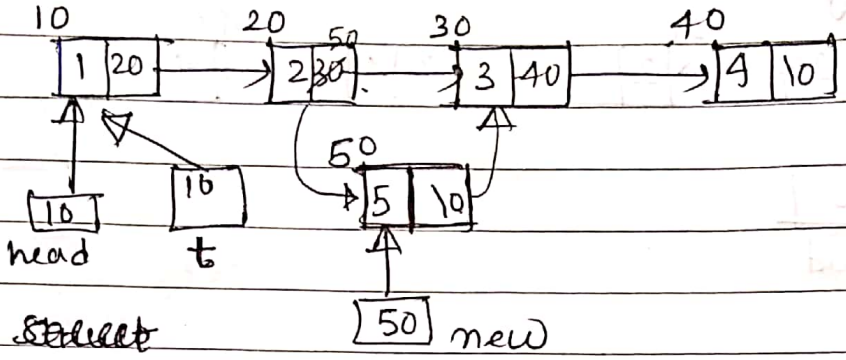
Case-1 Insert at the ~~bg~~ beginning =

```
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.



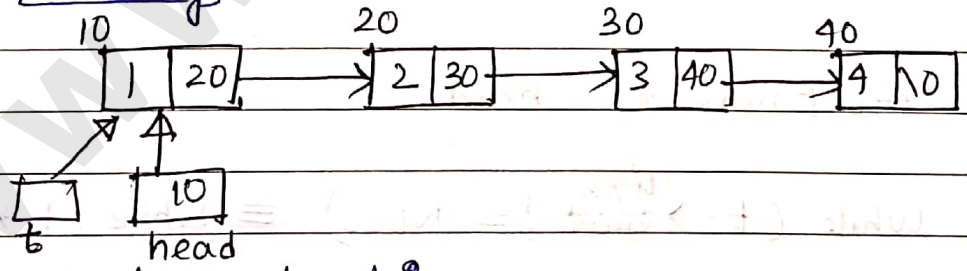
~~Struct~~

```
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;        (to delete ~~the~~ first one)
        link
free(t);
```

Recursive Condition Checking =

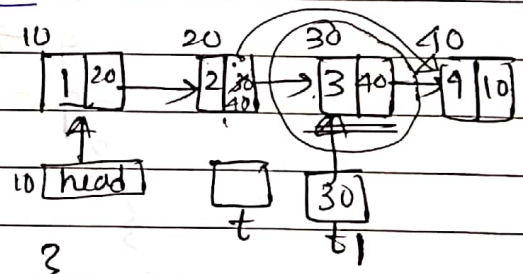
```
if (head == NULL)
    Return
if (head -> next == NULL)
    free (head)
```

before deletion check this. (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-2005-2008

Question - 1

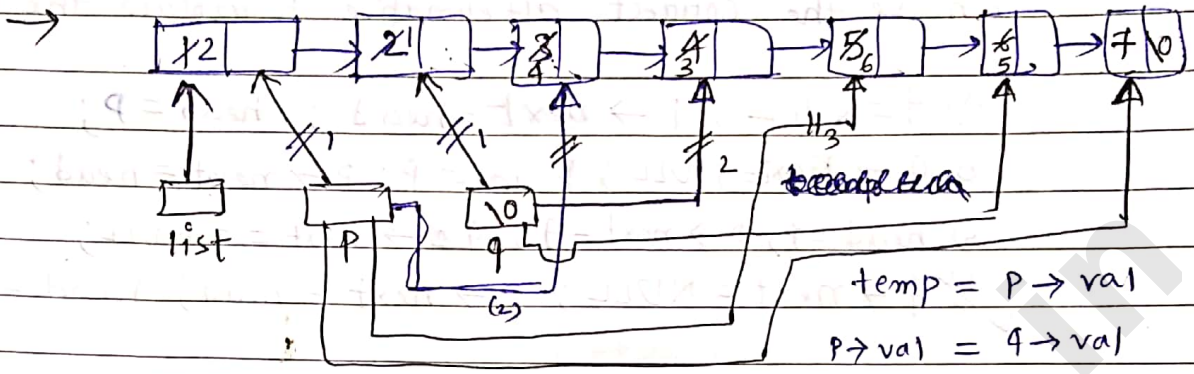
```
struct node
  {
    int val;
    struct node *next;
  };

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

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

(or) `if (!list || !(list->next)) return;` ≡ `if (list == NULL || !list)`

`while (q)` ≡ `while (q != NULL)`

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.

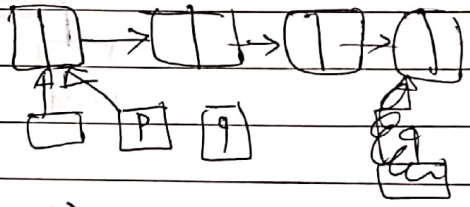


2 1 4 3 6 5 7

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 ;

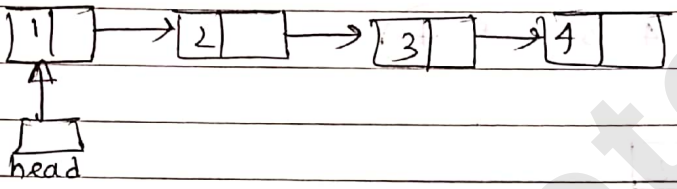
b) q → next = NULL ; head = p ; p → next = head ;

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

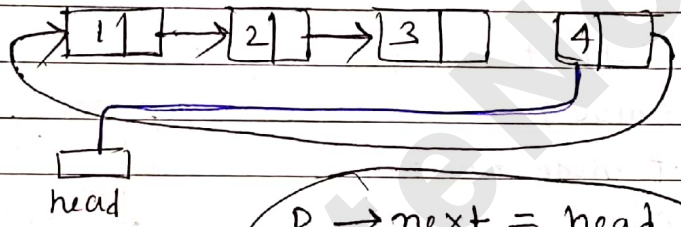
d) q → next = NULL ; p → next = head ; head = p ;

→

(I)



(II)



p → next = head

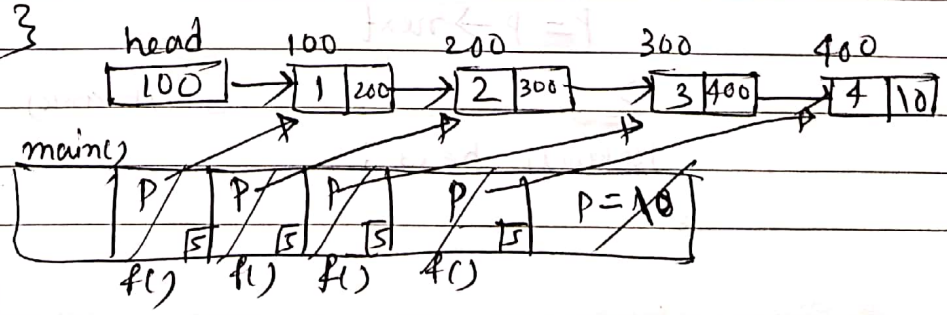
head = p

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)
      {
  }
```

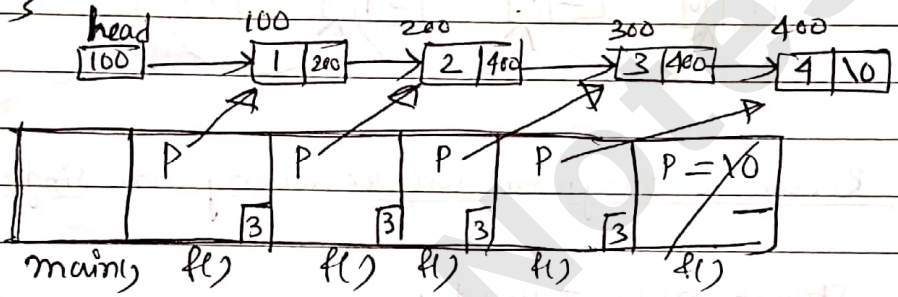
→



output: 1 2 3 4

(2)

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



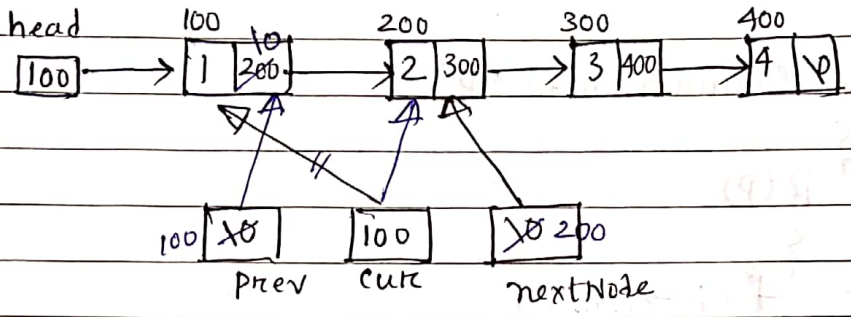
o/p: 4 3 2 1

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

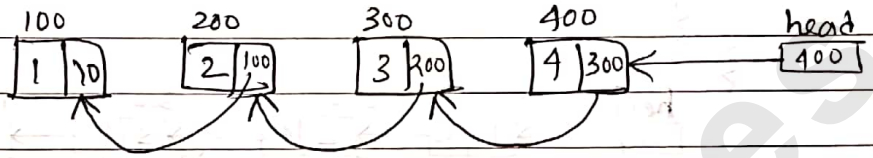
(1)

```
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);
}
```

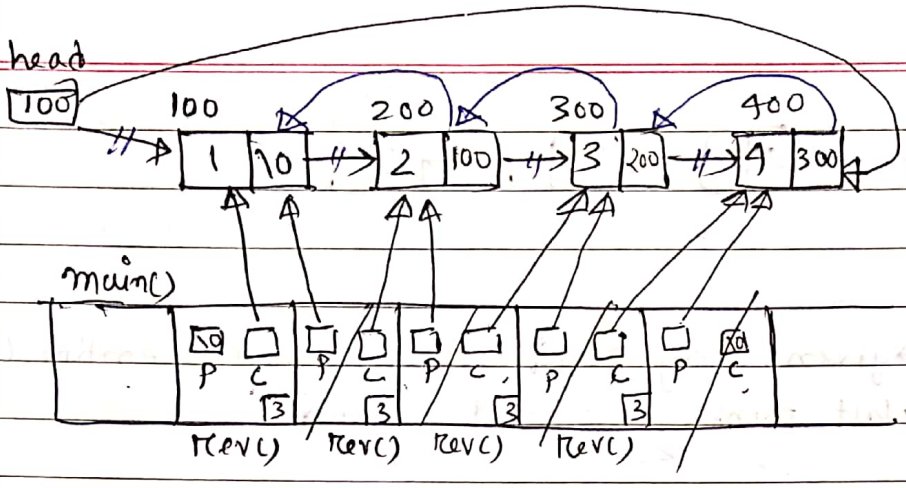


after operation



- 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 -

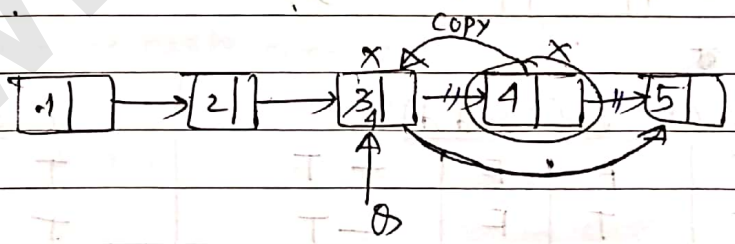
(A) $\log_2 n$ (B) $n/2$ (C) $\log_2 n - 1$ ~~(D)~~ n.

GATE-2004

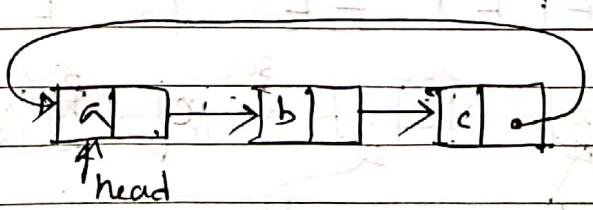
Question-4

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?

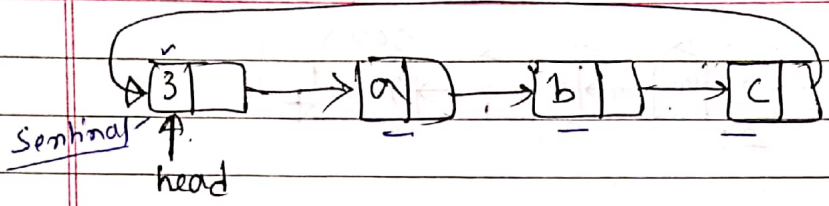
→ O(1).



- 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

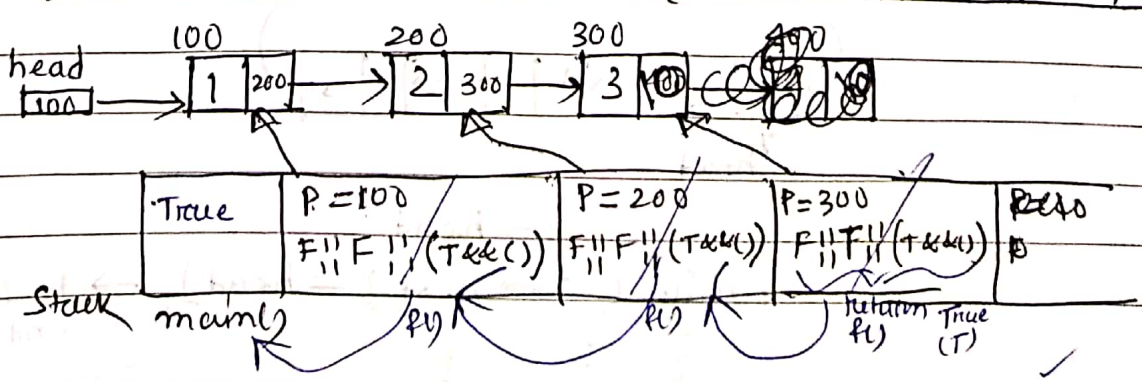
Question-5 (Checking if the list is in non decreasing)

```
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)));}
```

→ a (OR) b (OR) (c && d) (AND)

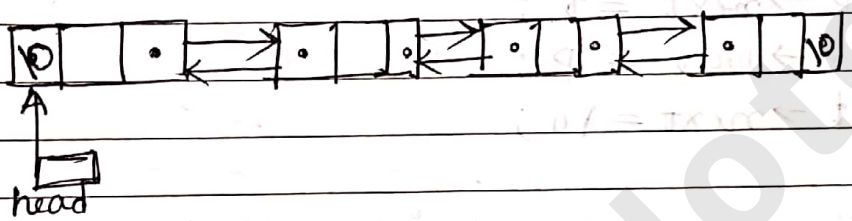
a	b	(c and d)	a or b or (c and d)
T	F	F	T
T	T	F	T
T	T	T	T
F	F	F	F

c	d	c and d
T	T	T
T	F	F
F	T	F
F	F	F

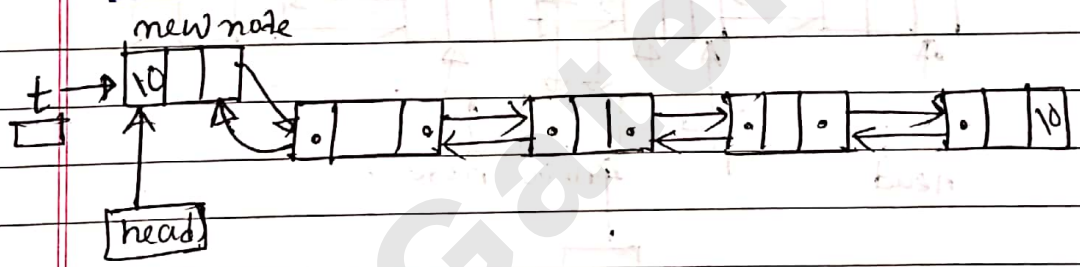


- **Insertion into doubly linked list :**

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

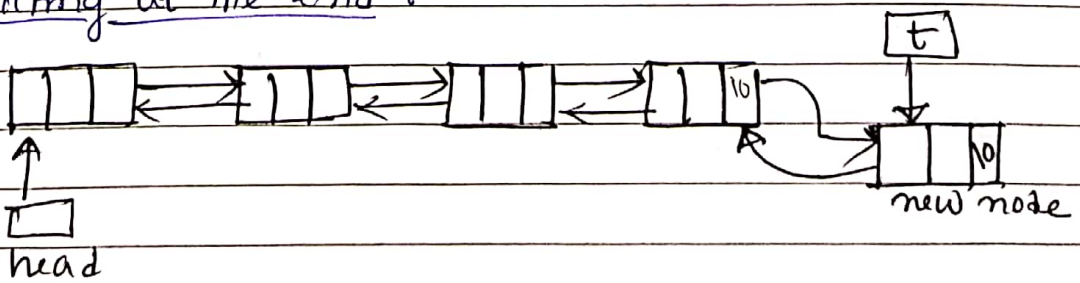


- **Insert a node at ~~front~~ beginning :-**




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

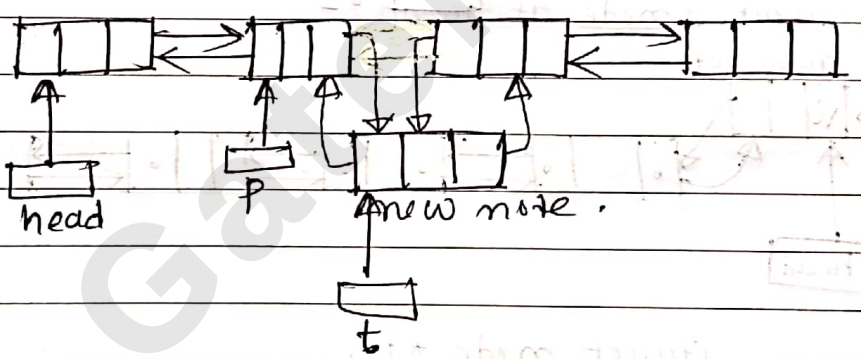
- **Inserting at the end :-**



~~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 :-



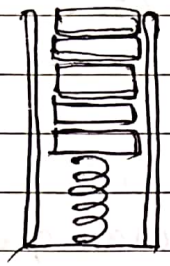

```
struct nod *t;
```

(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:**

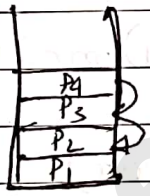


(insert) → push the plate from the top and take the plate from the top. ← (pop)

(stack) (FILO, LIFO)

- **Applications of stack:**

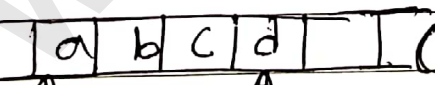
(i) Recursion.
(ii) IF → PF conversion.
(iii) Parsing
(iv) Browsers. →
(v) Editors. (use)
(vi) Tree traversals and graph traversal.



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

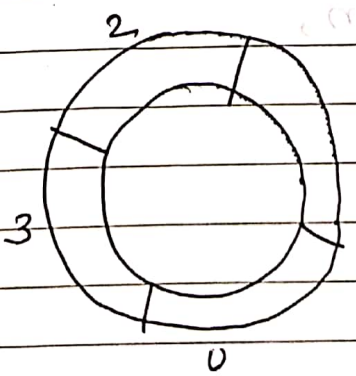
(enqueue) ← → (dequeue)

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



(Queue) (FIFO, LILO)

→ Insert from rear.
→ delete from front.

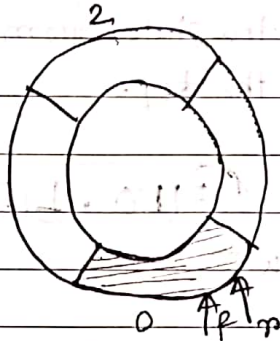


circular queue.

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 :-

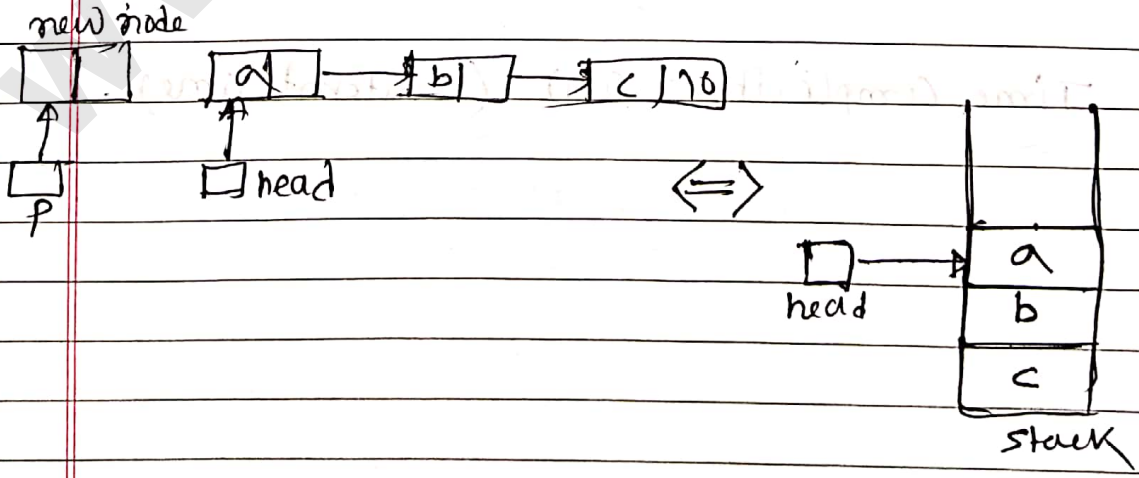
```
int Dequeue()
{
  if (front == rear)
    {
      Pf ("Q is empty");
      return -1;
    }
  else
    {
      front = (front+1) mod n;
      item = q[front];
      return item;
    }
}
```

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)

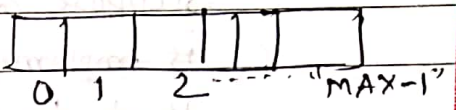
(2) deletion a item from stack =

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

  item = head → ~~di~~;
  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.

GR-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.

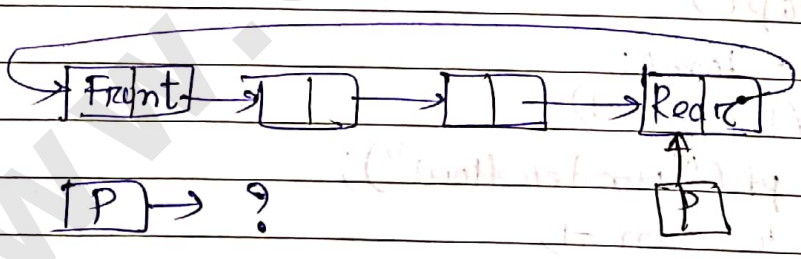
Queue Empty:

REAR = FRONT

GR-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.

→