Queue :-

- Queue is a linear data structure where the first element is inserted from one end called REAR & Deleted from the other end called as FRONT

⇒ Front points to the beginning of the queue and Rear points to the end of the queue

⇒ Queue follows the FIFO (first in first out) structure

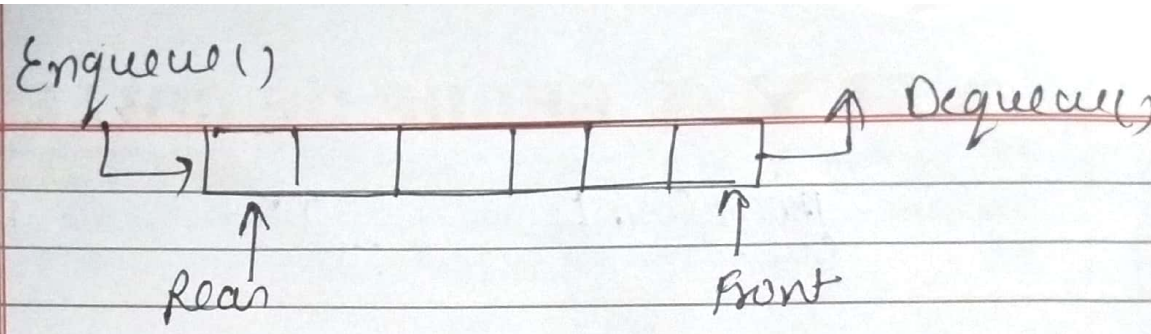
⇒ According to its FIFO structure, element inserted first will also be removed first

⇒ This contrasts with stacks, which are last in first out) (LIFO)

⇒ The process to add an element into queue is called Enqueue &

The process of removal of an element from the queue is Dequeue

Name of Lecturer:



A real world Example of queue can be Single-Lane one-way road, where the vehicle enters first- Exits first

Representation of queue :-

Queue may be represented in the Computer by means of one-way lists or linear array.

→ Unless otherwise stated or implied each of our queue will be maintained by a linear array QUEUE & two pointer variables

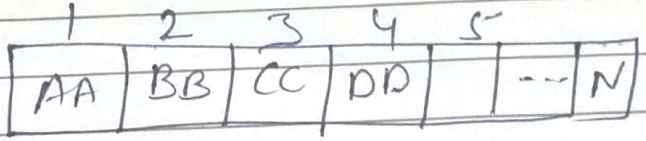
FRONT :- Containing the location of the front element of the queue

REAR :- Containing the location of the rear element of the queue.

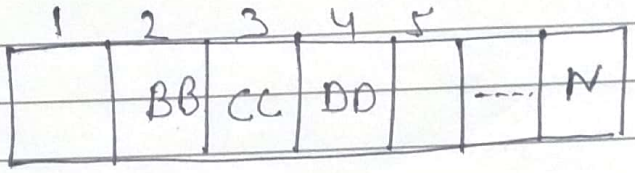
Name of Lecturer: .....

The Condition FRONT = NULL, indicates that the queue is empty

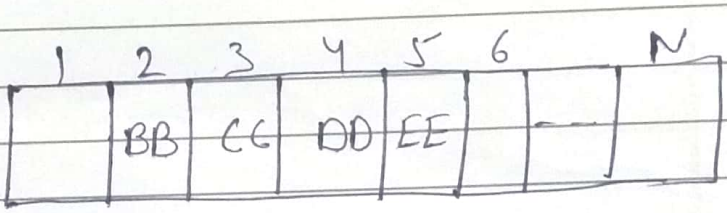
FRONT = 1
REAR = 4



FRONT = 2
REAR = 4



FRONT = 2
REAR = 6



The above figure shows the way the array QUEUE will be stored in memory with N elements.

⇒ Observe that whenever an element is deleted from the queue, the value of Front is increased by 1 i.e.

$$FRONT = FRONT + 1$$

⇒ Whenever an element is added to the queue, the value of REAR is increased by 1

i.e. Rear = Rear + 1

Name of Lecturer: .....

This means that after N insertions, the rear element of the queue will occupy QUEUE[N]. This occurs even through the queue itself may not contain many elements.

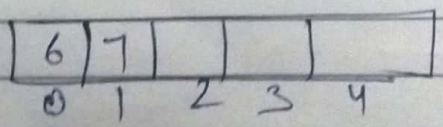
Name of Lecturer: .....

Queue:

Insertion :-

```
int queue[5];
int Rear = -1
int Front = 0

void insert (int n)
{
  if (Rear == 4)
  {
    printf ("queue overflow");
    exit(0);
  }
  else
  {
    rear = rear+1;
    queue[rear+1] = n;
  }
}
```



r=-1
F

r=0 value of 6
r=1 value of 7.

Name of Lecturer: .....

Algorithm:

Insert (queue, Rear, max, Item)

Here :-

queue is queue
Rear is insert position
max is size
Item is value

Step1 If check the queue is overflow.

```
if (Rear = max) then
{
    print the "queue is overflow";
    Exit.
}
```

Step2 Set Rear = Rear + 1

Step3 Store the value rear position.

Set queue [rear] = Item;

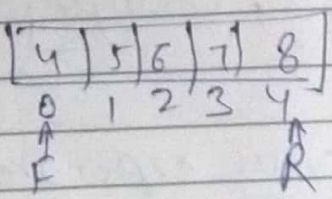
Step4 Exit

Name of Lecturer: .....

Deletion

```
int queue[5];
int Rear = -1
int front = 0

Void delete()
{
    if (rear < front)
    { printf (" queue is underflow");
      exit(0);
    }
else
      i = queue[front];
      front = front + 1;
      printf ("%d", i);
}
```



Algorithm:

delete (que, front, Rear, item)

step1 firstly we check the queue is underflow

If (Rear < front) then

print the queue is underflow

Exit();

End if

step2 Remove the value from front.

set item = que[front];

step3 Increment the front by 1

set front = front+1;

step4 Exit.

Note:- Suppose we want to insert an element item into a queue at the time queue does occupy last part of the array. i.e Rear=N

=> One way to do this is to simply move the ~~array~~ ~~changing front & Rear~~ entire queue to the beginning of the array, changing FRONT & Rear according.

=> The above procedure is very expensive thus we adopt the concepts of Circular queue.

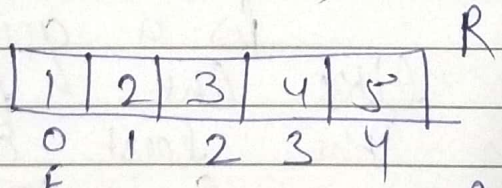
Name of Lecturer: .....

Q- What is the problem with simple, linear or static queue & how we can solve it?

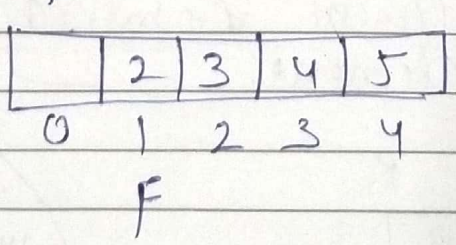
When we work with simple queue some time we face a problem on the time of insertion the problem is we says queue is full if the rear or max but this not always be true.

for example :- In case first rear is max position & we say queue is full, Actually full but in case second rear is also on Maxm position we say queue is full & but not actually full.

⇒



⇒



So if we have space from the front we can not utilize.

Solution:- There are various type solution when we delete the value front- shift a head but to ~~utilis~~ utilize if shift all the value front & rear position one back.

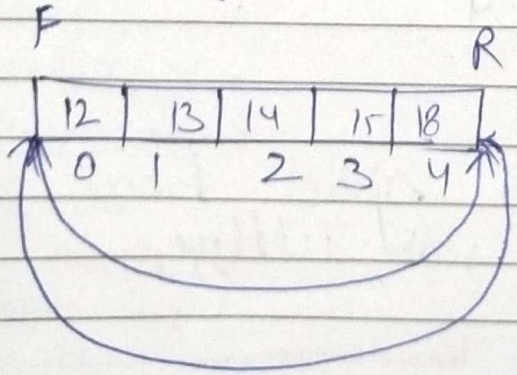
11	12	13	14	18
0	1	2	3	4

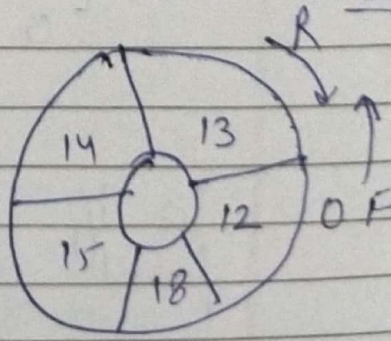
F

but this method is not a good method because in this method we required to shift all the element & rear front position one back only to one.
So to solve this problem we require other buffer alternate which is circular queue.

Circular queue :-

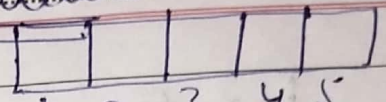
is a queue in which we assume like the first element come after the last element & the last element coming before the first element. like a Circular



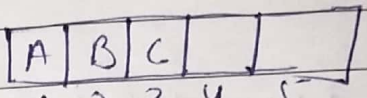


Name of Lecturer: .....

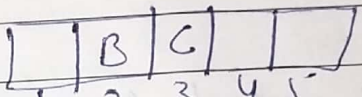
a) Initially Empty F=0 R=0



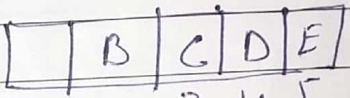
b) A, B & C inserted F=1 R=3



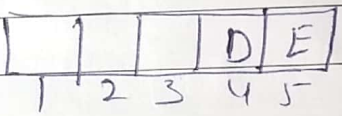
c) A deleted F=2 R=3



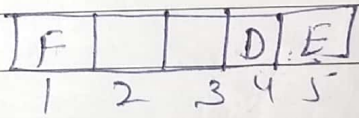
d) D & E inserted F=2 R=5



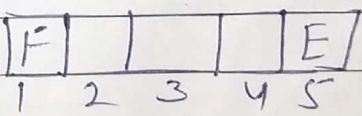
e) B & C deleted F=4 R=5



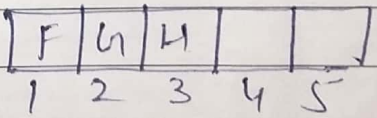
f) F inserted F=4 R=1



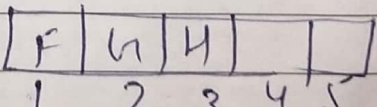
g) D deleted F=5 R=1



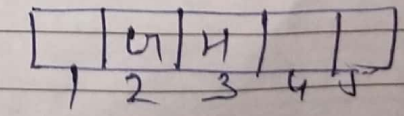
h) G & H inserted F=5 R=3



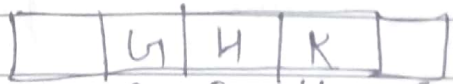
i) E deleted F=5 R=3



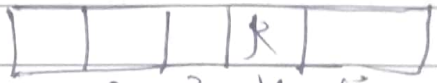
j) F deleted F=2 R=3



k) K inserted F = 2, R = 4



l) G & H deleted F = 4, R = 4



m) K deleted F = 0, R = 0

Queue is empty.

Circular queue :

Insertion :-

```
int cqu[05];

int Rear = -1
int Front = -1

void insert (int n)
{
  if ( Front == 0 && Rear == 4) || ( Front == Rear+1)
  { printf (" queue is overflow");
    exit (0);
  }
  else if ( Front == -1 && Rear == -1)
  {
    Front = 0;
    Rear = 0;
  }
  else if (Rear == 4)
    rear = 0;
  else
    rear = rear+1;
    cqu[rear] = n;
}
```

Algorithm :-

insert (Cque, Rear, Front, Item, Max)

Here :-

Cque is circular queue
Rear is insert position
Item is insert value
Max is size.

Step1 We check the overflow queue.

```
if (Front==0 && Rear==MAX) || (Front==Rear+1)
    printf("overflow");
    Exit(0);
EndIf
```

Step2 if (Front==-1 && Rear==-1) (queue is empty)

```
{ set front = 0
  set Rear = 0
```

End if

Step3

```
if (Rear==max) then
    set Rear = 0
otherwise
    set Rear = Rear+1
End if
```

Name of Lecturer: .....

Step 4 Store the value on rear position
set Cque[Rear] = item

Step 5 Exit

Program :- DQueue (deletion in queue)

```
void dll()
{
  if ( Front == -1 && Rear == -1)
  {
    printf (" queue is underflow");
    Exit(0);
  }
  item = Cque[Front];
  if (Rear == Front)
  {
    Front = -1;
    Rear = -1;
  }
  elseif (Front == 4)
    Front = 0;
  else
    Front = Front + 1;
  printf(" deleted value is = %d", item);
```

Name of Lecturer: .....
...ure: .....

Algorithm :
Deletion (cque, Rear, Front, item, max)

Step1 We check the underflow condition
if (Rear == -1 && front == -1) then
printf (" underflow");
Exit(0);
Endif

Step2 Delete the value from queue
Set item = cque[front]

Step3 if (front == Rear) (Means this is last element)
Set of Front = -1
Set of Rear = -1
Endif

Step4 if (front == Max) then
Set Front = 0;
otherwise
Set Front = Front + 1;
End if

Step5 Exit.

Name of Lecturer: .....

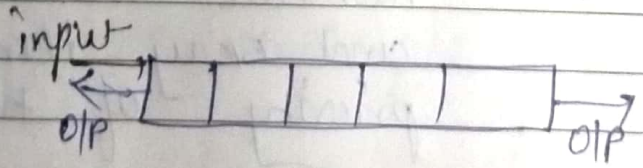
DE-queue :

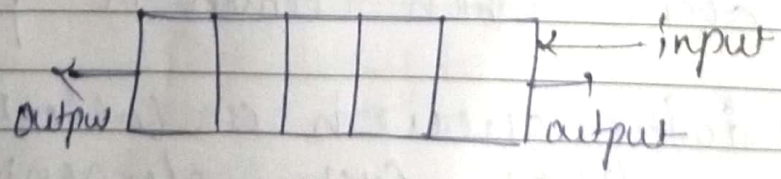
This queue means double ended queue in which insertion & deletion is perform from both end.

We mainly categoried into two categories.

① Input Restricted queue
② Output Restricted queue.

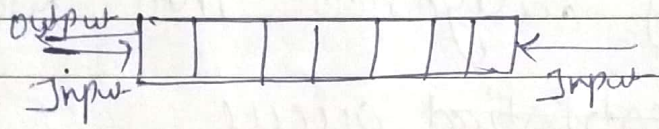
⇒ Input Restricted dequeue are those in which output can be perform both end but Input is perform only one end.

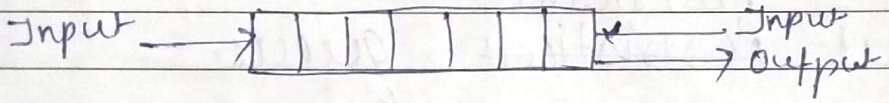




Output Restricted

In which output can be perform only from one end but input can be perform from both end.





Priority Queue :-

This queue is a queue in which we assign the numeric value to each and every item called priority of the item

~~which decide which element is processed when~~

⇒ A priority queue is a collection of elements such that each element has been assigned a priority.

The order in which elements are deleted and processed comes from the following rules.

① An element of higher priority is processed before any element of lower priority

② Two elements with the same priority are processed according to the order in which they were added to the queue.

⇒ A prototype of a priority queue is a time-sharing system.

⇒ There are various ways of maintaining a priority queue in H/o.

- one-way list
- multiple queue.

One-way list representation of a priority Queue :-

One way to maintain a priority queue in H/o is by means of a one-way list, as follows.

a) Each node in the list will contain three items of information :-

* An information field INFO
* A priority no. PRN.
* A Link no. LINK.

by A node x precedes a node y in the list
(i) when x has higher priority than y
(ii) when both have the same priority but x was added to list before y

Note:- Thus the order in the one-way list corresponds to the order of the priority queue.

priority no. will operate the usual way the lower the priority no. the higher the priority

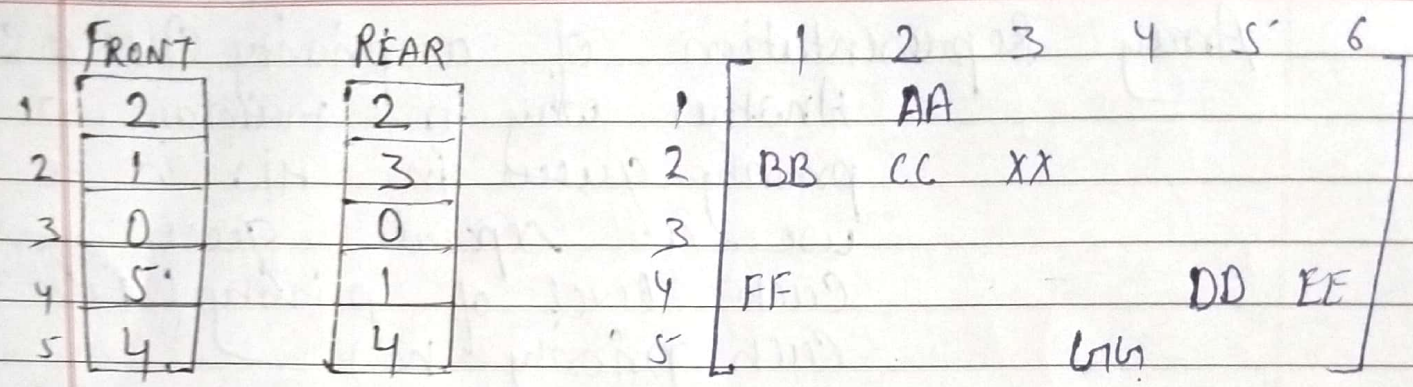
Array Representation of a priority Queue :-

Another way to maintain a priority queue in H/s is to use a seperate queue for each level of priority (for each priority no.)

⇒ Each such queue will appear in its own circular array & must have its own pair of pointers, Front & Rear.

⇒ In fact, each queue is allocated the same amount of space, a 2-D array QUEUE can be used instead of the linear array.

The following figure indicate this representation for one priority queue observe that FRONT[K] and REAR[K] contain, ~~represn~~ respectively the Front & Rear elements of row K of QUEUE, the row that maintains the queue of elements with priority no. K


FRONT
REAR
1 2 3 4 5 6
1 2 2
2 1 3
3 0 0
4 5 1
5 4 4
1
2 AA
3 BB CC XX
4
5 FF DD EE

Application of Queues:-

Round Robin algo

A popular use of queue data structure is the scheduling problem in the operating system.

⇒ Round-Robin is one of the simplest scheduling algo for process is an operating system, which assigns time slices to each process in equal portion and in order, handling all process without priority.

⇒ This scheduling designed for time-sharing system in which CPU is shifted to the next process after fixed interval time, which is called time quantum/ time slice.

⇒ Completion Time :- Time at which process completes its execution.

Burst Time :- Burst Time refers to the time required in mili seconds by a process for its execution.

Turn Around Time :- Time diffrunce b/w completion time & arrival time.

$$TAT = CT - AT$$

Waiting Time (WT) :- Time diffrunce b/w turn around time & burst time.

$$WT = TAT - BT$$

for exa:-

Process	Burst time	Arrival Time	CT	TAT	WT
P_1	~~5~~ x 0	0	7	7	4
P_2	~~4~~ x 0	0	9	9	5
P_3	~~3~~ x 0	0	10	10	7

given Time quantum = 2

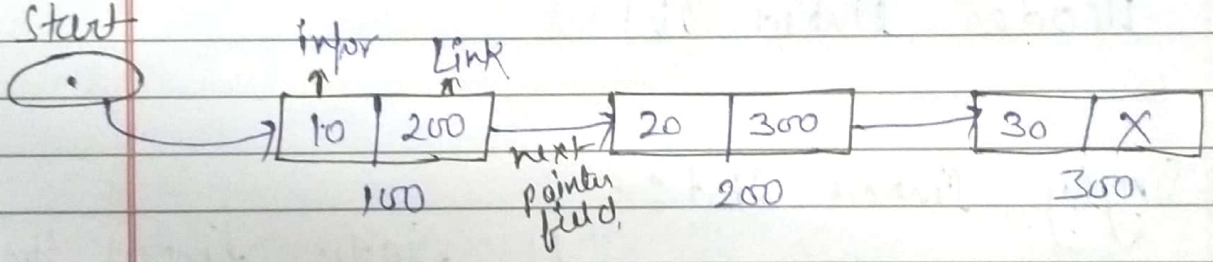
P_1	P_2	P_3	P_1	P_2	P_3
0	2	4	6	7	9	10

Average waiting Time :- $\frac{4+5+7}{3} = \frac{16}{3}$ 5.33

Linked Lists :-

A linked list or one-way list is a linear collection data elements called nodes, where the linear order is given by means of pointers.

⇒ Each node is divided into two parts.
① Information of the element
② Link field or nextpointer, contains the address of the next node in the list



⇒ The linked list contain a list pointer variable "Start" which contain the address of the first node in the list.

⇒ The pointer to the last node contains a special value called the null value / pointer.

Name of Lecturer: .....

Syntax

```
struct node
{
    int data;
    struct node *next;
};
struct node *start = 0, *newnode;
```

Types of linked list :-

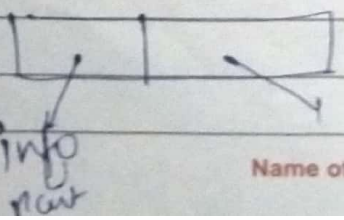
1. Singly linked list.
2. Doubly linked list
3. Circular linked list
4. Header linked list

Singly linked list :-

A Singly linked list is a special type of linked list in which each node has only one link that points to the next node in the linked list.

⇒ Traversal of items can be done in the forward direction only due to the linking of every node to its next node.

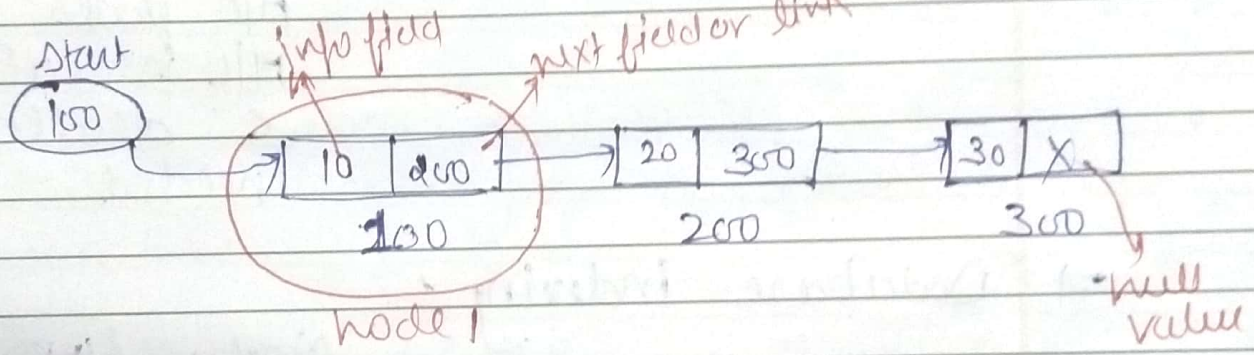
Each node have two parts :-



Name of Lecturer: .....

→ The **Start** Contains the Starting address of the linked list i.e. It contains the address of the first node of the linked List.

→ Next part of the last node contains the null value which indicates that it is the end of linked.



Characteristics of a Singly linked list:-

⇒ 1. Each node holds a single value & a reference to the next node in the list.

⇒ The list has a head, which is a reference to the first node in the list and a tail, which is a reference to the last node in the list.

⇒ The nodes are not stored in a contiguous block of memory, but instead, each node holds the address of the next node in the List.

Name of Lecturer: .....

⇒ Accessing elements in a singly linked list required traversing the list from the head to the desired node, as there is no direct access to a specific node in m/o.

Application of Singly linked list :-

⇒ Memory Management :-

Singly linked lists can be used to implement m/o pools, in which m/o is allocated & deallocated as needed.

⇒ Database indexing :-

Singly linked lists can be used to implement linked lists in database allowing for fast insertion and deletion operations.

⇒ Representing polynomials and sparse matrics :-

Singly linked list can be used to efficiently represent polynomials and sparse matrics, where most elements are zero.

Name of Lecturer: .....

7 Operating Systems :- Singly linked lists are used in operating system for tasks such as scheduling processes and managing system resources.

Advantages & Disadvantages of Single Linked List :-

Advantages :-

① It is a dynamic data structure that is, it can grow & shrink during run time.

② Insertion & deletion operations are easier.

③ Efficient memory utilization (no need to pre allocate memory)

④ Linear data structure such as Stack & Queue can be easily implemented using linked list.

Name of Lecturer: .....

Disadvantage :-

Reverse traversing is very difficult in case of single linked list.

⇒ No random access in Linked list, we have to access each node sequentially.

Representation of linked list in memory :-

⇒ Let list be a linked list Then LIST will be maintained in memory as follows :→

⇒ first of all LIST requires two linear array "INFO" and "LINK", such that INFO[K] and LINK[K] contain respectively the information part & next pointer field of a node LIST.

⇒ LIST also require a variable name such as START which contains the location of the begining of the list.

Note :- We will chose '0' to indicate NULL value

The following example of linked list indicate that the nodes of a list need not occupy adjacent element in the array INFO and LINK

Name of Lecturer: .....

→ More than one list may be maintain in the same linear arrays INFO & LINK

→ However, each list must have its own pointer variable giving location of its first node.

START = 9

	INFO	LINK
1		
2		
3	O	6
4	T	0
5		
6	□	11
7	H	10
8		
9	N	3
10	I	4
11	E	7
12		

START = 9
So INFO[9] = N is the first character
LINK[9] = 3 INFO[3] = O
LINK[3] = 6 INFO[6] = □ (blank)
LINK[6] = 11 INFO[11] = E
LINK[11] = 7 INFO[7] = X
LINK[7] = 10 INFO[10] = I
LINK[10] = 4 INFO[4] = T
LINK[4] = 0 The list ended

Algo for create a node in single linked list:-

step1 Include alloc.h
We will be allocating m/o using dynamic memory allocation in this library all functions are included

step2 Define node structure
```
struct node
{
    int data;
    struct node * next;
} *start = null;
```

step3 We create a node using DMA method:-
=> we don't have prior knowledge about no. of nodes. So we are call "malloc function to create node at run time

newnode = (struct node *) malloc (size of (struct

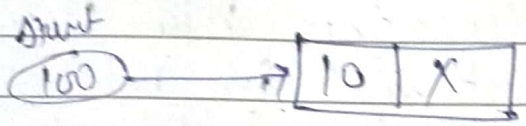
step4 Fill the information in newly created node 6-
=> whenever we create a new node, make its next field Null.
newnode → next = null;

Name of Lecturer: .....

Step5 Create very first node:-

If node created in the above step is very first node then

We need to assign it as starting node.



newnode = 100

Set start = temp = newnode

Step6 Create second or n^{th} node :-

→ Let assume, we have 1 node already created i.e. we have first node.

⇒ Now we have called Create() function again

```
else
{
    temp → next = newnode;
    temp = newnode;
}
```

in which we create link between temp & newnode and move temp pointer to the newnode pointer.

Step7 Stop.

Name of Lecturer: .....