

# Data Structures

(1)

## Unit-1

### Abstract Data Types:-

Data Type :- It is used to declare/define a variable. It specifies the data store format for variable.

e.g:- ~~char a;~~  
~~int b;~~

4, a, 9.5  
int  $n = 4$   
if  $n = a$   
char

### Abstract Data type -

#### Data type

##### Operations

+  
-  
/

X

##### Properties (e.g.) Float

that must be no.s with decimal, +ve, -ve

$$\checkmark c = a + b;$$

how it is being done  
we don't know?

Process is always Hidden

These are the data types which ensures some properties as well as they also define some operations which are hidden.

(OR)

Definition - It is a collection of data and collection of operation on that data. It is implemented by specific data types. It specifies what the ADT operations do, not how to implement them.

# ADT is a mathematical model of DS.

# It describes a container which holds a finite no. of objects where the objects associated through a given binary relationship.

# ADT are entities that are definition of data and operations but don't have implementation.

# what operation are to be performed but not how these operations will be implemented. Process proving only the essentials & hiding the internal details.

# ADT is used to create data struct.

## Linear V/S Binary Search -

(2)

Linear - searches for an element by visiting all the elements sequentially until the element is found.

0	1	2	3	4	5	6	7
7	10	2	9	11	23	13	

can be sorted  
or unsorted.

Element found complexity :  $O(n)$   
Search for 2  
(n elements)

## Binary Search -

Searches for an element by breaking the search space into half. i.e in a sorted array.

0	1	2	3	4	5	6	7
8	9	11	18	22	31	88	

Time complexity  $O(\log n)$

The search continues towards either side of mid based on whether the element to be search is found on lesser or greater than mid.

## Linear

# Works on both sorted and unsorted arrays.

# Equality op's

#  $O(n)$

Binary  
# works on sorted arrays

# Equality op's

#  $O(\log n)$

Linear search is done through traversal

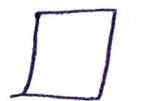
e.g.

0	1	2	3	4	5	6	7
14	8	10	12	15	16	2	8

 found if search for 2  
 e.g. for card  
 (Sorted or Unsorted)

## Binary Search -

Search for 238 Page  
 $\frac{250}{2}$  half  
238 1  $\leftarrow \frac{500}{2}$  Half  
converge



1000 Pages

Book

in Sorted Array must be sorted.  
not Sorted X

e.g.

0	1	2	3	4	5	6	7	8
2	8	14	32	66	100	104	120	130

if search for 8 Linear is better time 2yada  
200. Binary is better for sorted long array.

Low + High = greatest integer

Low	High	Mid	Search 200
0	8	4 $\Rightarrow$ 8/2 element found	
4	8	6 $\Rightarrow$ No 12/2 is not	
6	8	7 $\Rightarrow$ Yes {Binary}	

## Traversal -

(3)

```

for (i=0; i<n; i++)
{
    printf("%d[i]);  

}
O/P - 1, 2, 3, 4, 5

```

## Bubble Sort -

10	9	11	6	15	2
----	---	----	---	----	---

if  $10 < 9$  no

9 ⑩, 11, 6

9 10 ⑯ ⑪

Pass 1

9 10 6 11 ⑯ 2

9 10 6 11 2 ] 15

$\cancel{10}$

largest no is  
on the correct  
position

⑨ 10 6 11 ← 2

9 ⑩ 6 11 2

9 6 ⑩ 11 2

9 6 10 ⑪ 2

⑨ 6 10 2 ] ⑪ 15

6 ⑨ 10 2

6 9 ⑩ 2

6 9 2 ] 10

Total no of swapping  
time complexity  $\Theta(n^2)$

## Insertion Sort -

sort(int a[], int n) e.g. 8 6 5 10 9

```

    {
        int i, j, key;
        for(i=1; i<n; i++)
    }
  
```

key = a[i];

// j = i - 1;

while (j > 0 && a[j] > key)

{ a[j+1] = a[j];

j = j - 1;      i = 1

}

a[i+1] = key;      A[0] =   

/      {

Bubble Sort(int a[], n)

```

    {
        int swapped, i, j;      i
        for(i=0; i<n; i++)      j
    }
  
```

swapped = 0;

for(j=0; j < n-i-1; j++)

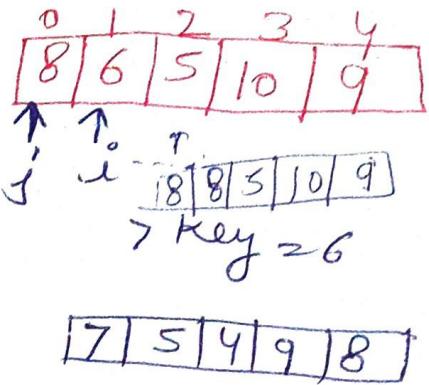
{

if (a[j] > a[j+1])

swap(a[j], a[j+1])

swapped = 1;

} if (swapped == 0) break;



```

#include <stdio.h> (1)
int linearSearch(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element)
            return i i;
    }
    else
        return -1;
}

```

## Operations on Arrays -

\* Traversal

0	1	2	3	4	5
7	8	9	10	15	1000000

\* Insertion - index no 2 → 5 <sup>99</sup>

\* Deletion case 1 - Maintain the order  
Best case insertion  $O(1)$

Worst Case -  $O(n)$   
outplace

case 2 - No need to maintain the order.

7	8	12	10	15	9	-
x						

Delete - 7 2 10 15 9 - - Size 21

Shifts worst case

7	9	2	10	15	9
← Best case					

No of element - upper bound - lower bound +1

Insertion at end -

Let A be an array of size Max. or N

LB	0	1	2	3	VB	UB
A[6]	0 - 5				A[5]	=

1- If  $VB = \text{Max}$  -

then print overflow & Exit.

2- Read data

3-  $VB = VB + 1$

$VB = 4$

4-  $A[VB] = \text{data}$

$VB = 5$

5- Stop or Exit

Insertion at Beginning -

Let A be an Array of size Max(OR) N

1- If  $VB \geq \text{Max}$  -

then print overflow & Exit

2- Read data

3-  $VB = K$

0	1	2	3	4	5	6
Data	3	2	6	8	5	

4- Repeat steps 5

while  $K \geq LB$

5-  $A[K+1] = A[K]$

$K = K + 1$

$$A[5] = A[4]$$

$$A[4] = A[3]$$

6-  $A[LB] = \text{data}$

7- Stop

## Insertion at a location

(5)  $A[2] = 10$

- Let  $A$  be an array of size  $N$
- 1- if  $VB = \text{Max} + 1$   
then print overflow & EXIT

2- Read data

3- Read loc

4-  $UB = K$

5- Repeat step 6

while  $K \geq \underline{\text{loc}}$

6-  $A[K+1] = A[K]$

7-  $A[\text{Loc}] = \text{Data}$   
stop

$A[0]$	1	2	3	4
	3	2	6	8

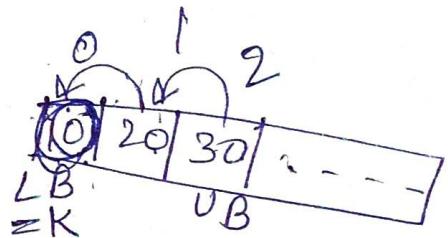
$$A[2] = 10$$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	2	10	6	8

$$A[3] = A[3]$$

## Deletion Algorithm - Array -

### Deletion from Beginning -



- 1- If  $VB = 0$  then print  
'Underflow'

2-  $K = LB$

3- Repeat step 4

while  $K < UB$

4-  $A[K] = A[K+1]$

5-  $K = K+1$

6-  $A[VB] = \text{NULL}$

$VB = VB + 1$

stop:

## Deletion of Particular element -

1- If  $VB = 0$ , print Underflow

2- Read data

3- ~~Reset~~  $K = LB$

4- Repeat step 5

    while  $A[K] \neq \text{data}$

5-  $R = K + 1$

6- Repeat step 7

    while  $K \leq VB$

7-  $A[K] = A[R]$

$R = R + 1$

8-  $A[VB] = \text{NULL}$

$VB = VB - 1$

    stop

5	10	15	X	20	25	VB
A[0]	ACW	ACW				

## Deletion from End -

1- If  $VB == \text{NULL}$  then

Underflow

2-  $A[VB] = \text{NULL}$

3-  $VB = VB - 1$

4- Stop

A[0]	1	2	3	--
3	5	7	9	--

LB=0                    VB=3

Insertion Sort - arrange  $N$  elements of array by inserting particular item in a particular place such the way that the item are in sorted order.

Stack ADT :- In order to create a

- 1 Stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of stack ADT are —

- 1- Push()  $\rightarrow$  Push an element into the stack.
- 2- POP()  $\rightarrow$  Remove the topmost element from the stack.
- 3- isEmpty / is full
- 4- Top()

Infix, Prefix & Postfix :-

Notation to write an expression

- 1- Infix -  $a+b$       (c)  $\frac{p}{q}$   
                (b)  $a-b$       (d)  $p \div q$

2- Prefix -  $5+5$   
 $S, S, +$

$\sqrt{S, S}$  (Pre)

$(operator) > (operand) > (operator) > (operand)$

$+ab$   
 $-xy$

$*pq$

```

isEmpty()
{
    if (Top == -1)
        return true
    else
        return false
}

POP()
{
    int Data
    if (!isEmpty())
    {
        Data = Stack[Top]
        Top = Top - 1
    }
    else
    {
        "Underflow"
    }
}

```

### Applications of Stack:-

- 1- Used in function calls
- 2- Infix to postfix conversion (and other similar conversion)
- 3- Parenthesis matching & more.

→ A stack is a **non-primitive** linear data structure. It is an ordered list in which addition of a new data item and deletion of already existing data item is done from only one end known as top of stack (TOS).

→ Most frequently accessible element in the stack is the top most element, whereas the least accessible element is the bottom of the stack.

## Insertion sort example

0	1	2	3	4	5
40	20	60	10	50	30

$j = 3$     $i = 4$

0	1	2	3	4	5
20	40	60	10	50	30

$j = 3$     $i = 4$

$$\text{key} = 20$$

$$j = 0$$

0	1	2	3	4	5
20	40	60	10	50	30

$j = 3$     $i = 4$

$$\text{key} = 60$$

$$j = 1$$

$40 > 60$  (False)

$$\text{key} = 10$$

$$j = 2$$

$$60 > 10$$

$$j = j - 1$$

$$j = 1$$

$$a[2] = 10$$

0	1	2	3	4	5
20	40	60	(60)	50	30

0	1	2	3	4	5
20	40	40	60	50	30

0	1	2	3	4	5
10	20	40	60	50	30

## Stack - (FILO (DR) LIFO)

### Operations-

int Max=5

int stack[Max]

int Top=-1

isfull()

if (Top == Max-1)

return true;

else

return false;

Push(Data)

{

if (!isfull())

{

Top=Top+1

stack[Top]=Data

}

else

"Overflow" ??

# Postfix notation is type of notation ⑧  
which is most suitable for a computer  
to calculate any expression. It is  
universally accepted notation for  
designing arithmetic & logical unit  
(ALU) of the CPU.

# Any expression entered into the  
computer is first converted into  
postfix notation, stored in stack  
& then calculated.

Cg       $A + B * (C + D) / F + D * E$  infix to  
prefix -

$\xrightarrow{A + B * (+ C D) / F + D * E} ()$

$\xrightarrow{A + * B + C D / F + D * E} / *$

$\xrightarrow{A + / * B + C D F + D * E} + -$

$A + / * B + C D F + * D E$

$\frac{+ A / * B + C D F + * D E}{+ + A / * B + C D F * D E}$

3-Postfix- Operand1 > Operand2 > Operator

e.g.: ab +  
XY -  
pq \*

Infix  
a \* b  
a - b

Prefix  
\* ab  
- ab

Postfix  
ab \*  
ab -

Infix:  $(A_1 * (B + C)) * D_2$  (conversion)  
Postfix: A  $\overset{1}{B} \overset{2}{C} + * \overset{3}{D} *$  (56)  $\downarrow$   
 A 8 \* D \*

16 D \*

Q1 -  $x - y * z$  convert this to prefix to  
Postfix -

Prefix  $\Rightarrow$  Step 1 - Parenthesize the expression

Polish notation  $(x - (y * z))$

$(x - [* y z])$

$- x [* y z]$

Q2 -  $p - q - r/a$   
 $(p - q) - (r/a)$

Prefix -

$(-pq) - (1/r)$

$- - pq / ra$

Postfix -  $\overline{\overline{x - (y * z))}}$   
 $x - [yz *]$   
 ~~$x y z * -$~~

Postfix -

Q2-  $a + b * c + (d * e + f) * g$

(9)

Convert this into Postfix expression.  
 $\Rightarrow abc * + de * f + g * +$

Q3-  $A + (B * C - (D / E \wedge F) * G) * H$

$$\begin{array}{c} A B C * D E F \wedge G * - H * + \\ \hline \end{array}$$

+ ↑  
\* ↑  
+ ↑

$\Rightarrow$  Evaluation of a Postfix expression using stack -

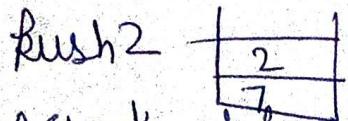
$$5, 6, 2, +, *, 12, 4, /, -,$$

Symbol Scanned	Stack	Opn
----------------	-------	-----

5	5	
6	5, 6	
2	5, 6, 2	
+	5, 8	
*	40	[6+2]
12	40, 12	5*8
4	40, 12, 4	
/	40, 3	12/4
-	37	40-3

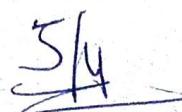
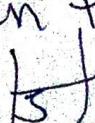
Q4 7, 2, -, 1, 3, +, / (Postfix)

Push 7 on the stack -



next is operator remove top 2

elements of stack & perform  $2 - 7 = 5$  place this result back on the stack



Postfix -  $A + B * \underline{C D + / F + D * E}$   $\downarrow$   
 $A + \underline{B C D + * / F + D * E}$   $\downarrow$   
 $A + \underline{B C D + * F /} + D * E$   
 $A + \underline{B C D + * F /} + P E *$   
 $\boxed{A B C D + * F /} + \boxed{D E *}$

C.Q.  $5 * (6+2) - (12/4)$

symbol scanned

Stack

Postfix expression

1-	5		5
2-	*	*	5
3-	(	*, (	5
4-	6	*, (	5, 6
5-	+	*, (, +	5, 6
6-	2	*, (, +	5, 6, 2
7-	)	*	5, 6, 2, +
8-	-	-	5, 6, 2, +, *
9-	(	-, (	5, 6, 2, +, *
10-	12	-, (	5, 6, 2, +, *, 12
11-	/	-, (, /	5, 6, 2, +, *, 12
12-	4	-, (, /	5, 6, 2, +, *, 12, 4
13-	)	-	5, 6, 2, +, *, 12, 4, /
14-			5, 6, 2, +, *, 12, 4, /, -

'OR) Postfix expression

$5 * \underline{6 2 +} - (12/4)$

$5 * \underline{6 2 +} - \underline{12 4 /}$

$5 6 2 + * 12 4 / -$

$\equiv$

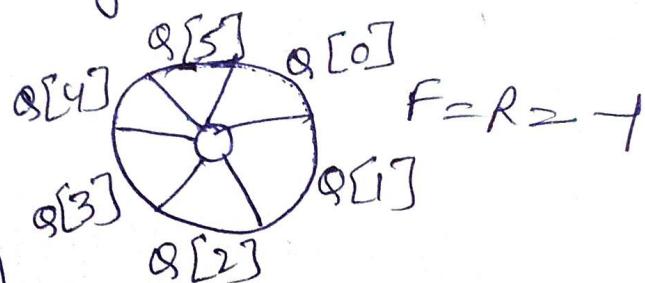
(P)

## Queue ADT:-

- 1) Enqueue(X)  $\rightarrow$  Insert X
- 2) Dequeue()  $\rightarrow$  Delete
- 3) Front()
- 4) IsEmpty()  $\rightarrow$  T/F
- 5) IsFull()

Appn - 1) Buffer  
 2) Operating system ~~P~~ in scheduling

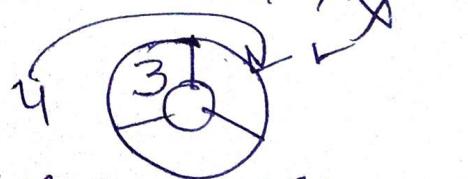
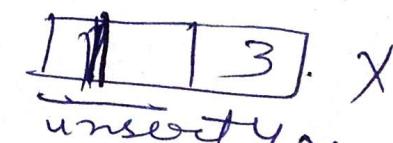
~~Q~~ Circular Queue - A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of queue is full.



# A circular queue overcome the problem of utilized space in linear queues implemented as arrays.

circular Queue has following conditions-

- 1) Front will always be pointing to the first element.
- 2) If front = Rear the queue will be empty.



3) Each time a new element is inserted into the queue the Rear is incremented by one.

$$\boxed{\text{Rear} = \text{Rear} + 1}$$

4) Each time an element is deleted from the queue the value of front is incremented by one.

$$\boxed{\text{Front} = \text{Front} + 1}$$

### Insertion in circular Queue -

QINSERT (Queue [MaxSize], item )

Step1 - If (Front == (Rear+1) % Maxsize)

    write queue is overflow & EXIT

Else : take the value

    if (Front == -1)

        Set Front = 0

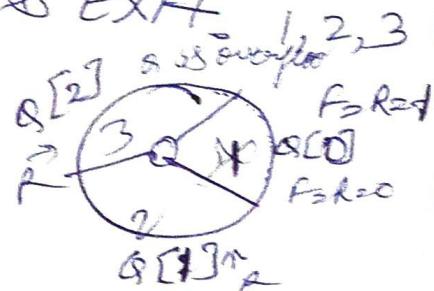
        Rear = 0

    else

        Rear = (Rear + 1) % Maxsize

[Assign Value]  $\rightarrow$  Queue[Rear] = Value

End if.]



$$f=1$$

$$F = (2+1) \% 3$$

(11)

$$Q \rightarrow A \times B - (C + D) \times (P/Q)$$

$$Q \rightarrow A + (B \times C - (D/E + F) \times H)$$

$$Q \rightarrow A/B + C - D$$

$$Q \rightarrow (A - B/C) * (D * E - F)$$

Queue ADT -

EnQueue :-

1- If, Rear == Max

Print overflow & Exit.

2- Read data

3- If front == 0, then

Front = Front + 1

Rear = Rear + 1

Q [ Rear ] = Data

4- else

Rear = Rear + 1

Q [ Rear ] = Data

5- Stop

DeQueue :-

1- If front == -1 then

Print underflow & Exit.

2- If front == Rear

Q [ Front ] = NULL

Front = -1

Rear = -1

else

Q [ front ] = NULL

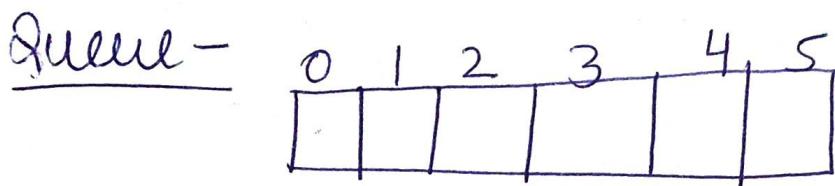
3- Exit front = front + 1

i.e. only one element is present in queue



Q - Convert  $A/B \setminus C - D$  into prefix form.

Q -  $(A - B / C) * (D * E - F)$  prefix  
 $* - A / B C - * D E F$



front = ↓  
 rear = ↓

f1 f0 {  
 L1 L0 }

i.e.) Empty

Insert A -

0	1	2	3	4	5
A					

F=R=0

Insert B, C, D -

0	1	2	3	4	5
A	B	C	D		

F=0 R=3

Deletion of element A -

0	1	2	3	4	5
	B	C	D		

F=1 R=3

Insertion of E -

0	1	2	3	4	5
	B	C	D	E	

F=1 R=4

Insert in Queue  
 En Queue  
 Delete  
 De Queue

\* Queue is a non-primitive linear data structure.

\* It is a homogeneous collection of elements in which new elements are added at one end called Rear end, and the existing elements are deleted from other end called the Front end.

\* The first added element will be the first to be removed from the queue.

## Delete operation in circular Queue - (12)

QDelete (Queue [Maxsize], item)

1- if (Front = -1)

write queue underflow & Exit

else,

item = Queue [Front]

if (Front == Rear)

Set Front = -1

Set Rear = -1

else

Front = ((Front + 1) % Maxsize)  
[endif]

2- Exit      item deleted

e.g. item = q [F]      }

item = q [0]

item = 10

if (F == R)

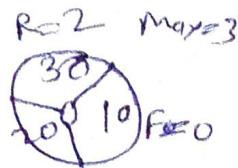
O = 2      false

F = (F + 1) % 3

F = (0 + 1) % 3

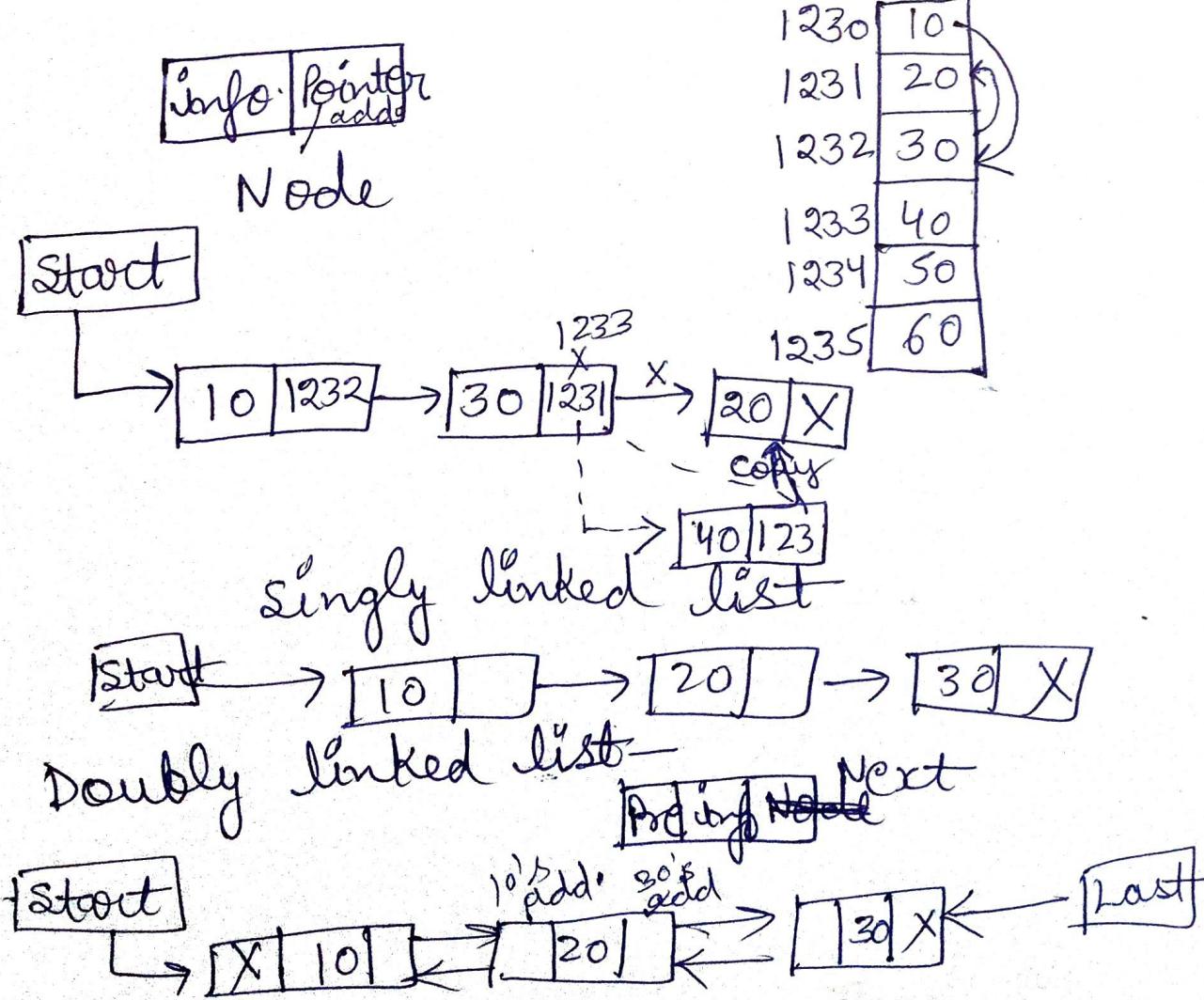
F = 1

10 is deleted.



## Linked List

- # A linked list is a linear data structure in which the elements are not stored at contiguous memory location.
- # A linked list is a dynamic data structure. The no. of nodes in a list is not fixed and can grow and shrink on demand.
- # Each element is called a node, which has two parts.
- # Information part which stores the info & pointer, which points to the next element.



## Deleting First Node from Linked list

Delete First (start)

Step1 - check for underflow

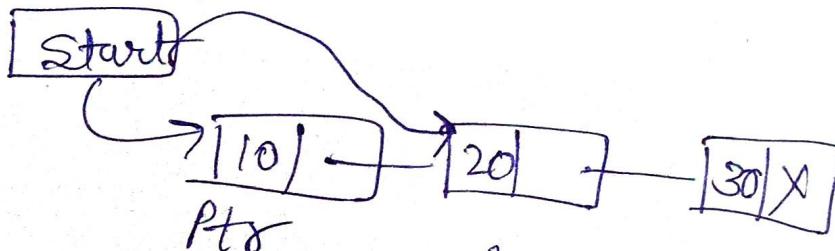
if  $\text{start} = \text{NULL}$  then  
point linked list empty.  
Exit

Step2 - set  $\text{ptr} = \text{start}$

Step3 set  $\text{start} = \text{start} \rightarrow \text{Next}$

Step4 - Point element deleted is  
 $\text{ptr} \rightarrow \text{info}$

Step5 - free( $\text{ptr}$ )



## Delete Last Node from Linked list

Delete (start)

Step1 Check for underflow

if  $\text{start} = \text{NULL}$  then

Point linked list is empty.

Step2 - If  $\text{start} \rightarrow \text{Next} = \text{NULL}$  then

Set  $\text{ptr} = \text{start}$

Set  $\text{start} = \text{NULL}$



Point element deleted is  $\text{ptr} \rightarrow \text{info}$

free( $\text{ptr}$ )

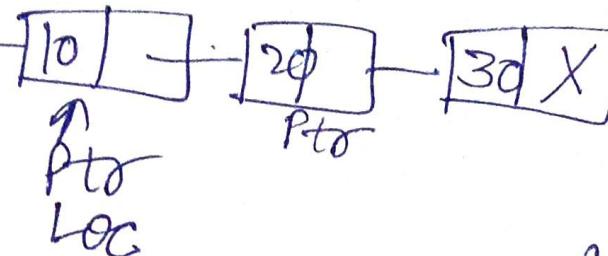
end if

Step3 - set  $\text{ptr} = \text{start}$

Step4 - Repeat Step5 & 6 until  
 $\text{ptr} \rightarrow \text{Next} != \text{NULL}$

- 3  
 5  
 8  
 9  
 13  
 19  
 22  
 23  
 25  
 27  
 28  
 29  
 30  
 33  
 34  
 35  
 37  
 40  
 46
- 
- Step5** - Set Loc = Pts  
**Step6** - Set Pts = Hs  $\rightarrow$  Next  
**Step7** - set Loc  $\rightarrow$  Next = NULL  
**Step8** - free(Pts)

C.g: Start



Difference between Singly and doubly linked list

### Singly linked list

# Singly linked list has nodes with data field and next link field (fwd link)

e.g: Data|Next

# It allows traversal only in one way.

# It requires one list pointer variable (start)

# It occupies less memory

# Complexity of insertion & deletion at known position is  $O(n)$

### Doubly linked list

# Doubly linked list has nodes with data field, & two pointer field (backward & fwd link)

e.g: Pre|Data|Next

# It allows a two way traversal

# It requires two list pointer variable (start & last)

# It occupies more memory.

# Complexity of insertion & deletion at known position is  $O(1)$ .

Insert a node at the end in singly linked list

Linked list -

Algorithm -

Insert\_Last(START, ITEM)

Step1 - check for overflow

IF  $Ptr = \text{NULL}$  then print overflow.  
Exit

else

$Ptr = (\text{Node}*) \text{malloc}(\text{sizeof}(\text{node}))$ ;

Step2 - Set  $Ptr \rightarrow \text{info} = \text{item}$

Step3 - Set  $Ptr \rightarrow \text{Next} = \text{NULL}$

Step4 - If  $\text{start} = \text{NULL}$  and then  
set  $\text{start} = Ptr$ ;

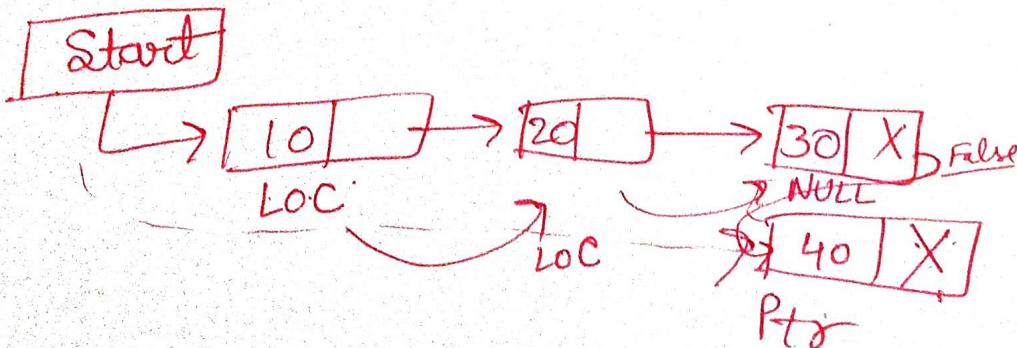
else

Step5 - Set  $Loc = \text{start}$

Step6 - Repeat step 7 until  $Loc \rightarrow \text{Next} \neq \text{NULL}$

Step7 - Set  $Loc = Loc \rightarrow \text{Next}$

Step8 - Set  $Loc \rightarrow \text{Next} = Ptr$ ;



# Insert node at beginning of linked list

Algorithm—

InsertFirst(Start, item)

Step 1— [Check for overflow]

If  $\text{Ptr} = \text{NULL}$  then  
point overflow  
exit

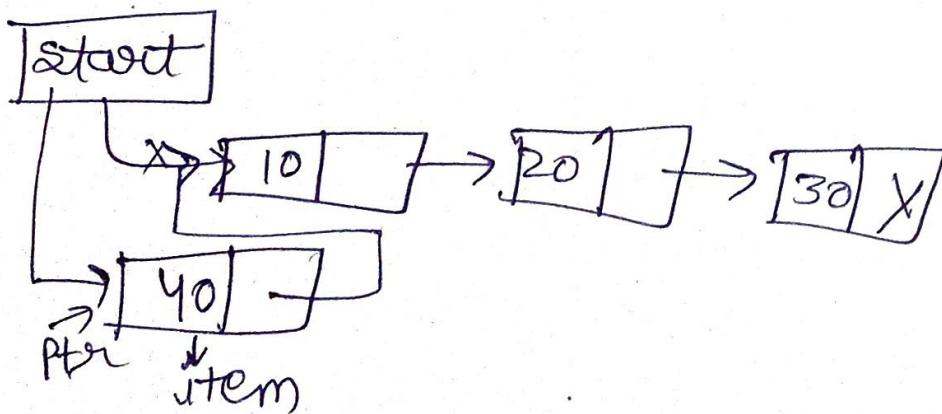
else

$\text{Ptr} = (\text{Node} *) \text{malloc}(\text{size of}(\text{Node}))$ .  
// creates new node from memory and  
assign its address to  $\text{ptr}$  //

Step 2— Set  $\text{ptr} \rightarrow \text{info} = \text{item}$

Step 3— Set  $\text{ptr} \rightarrow \text{next} = \text{start}$

Step 4— set  $\text{start} = \text{ptr}$



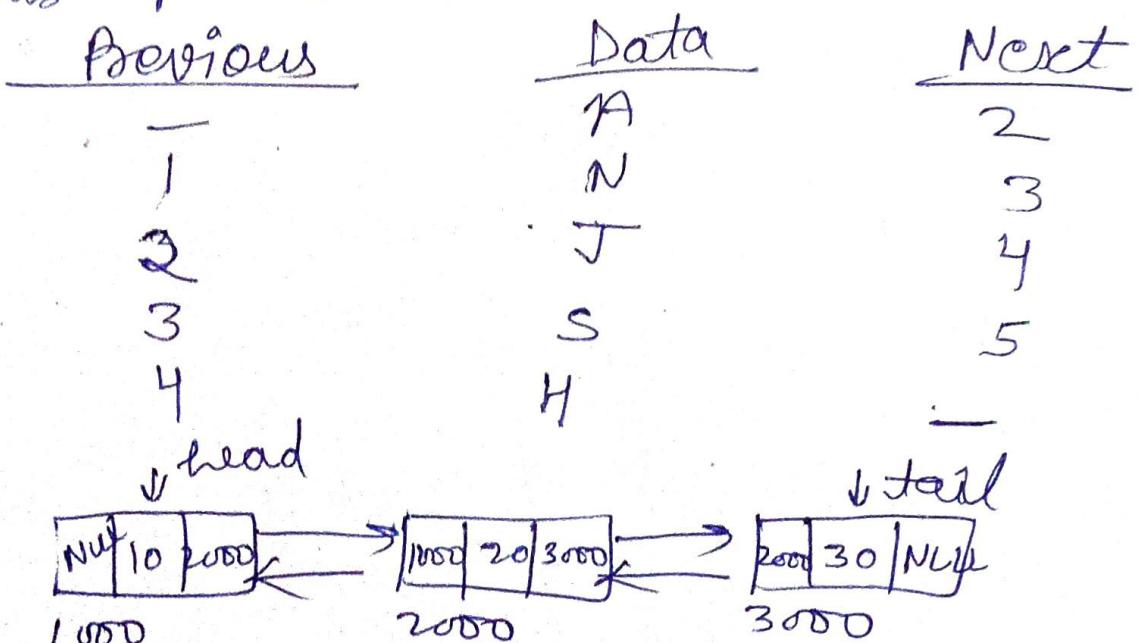
## Doubly linked list

(15)

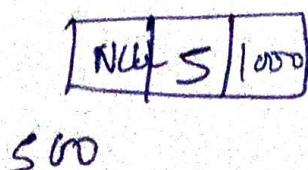
It is the linear collection of data item called node where each node has divided into three parts i.e. data part, previous part & next part.

Data part store the data item, previous part store the address of previous node and next part store the address of next node.

If starts with special pointer called first pointer & ending with last pointer.



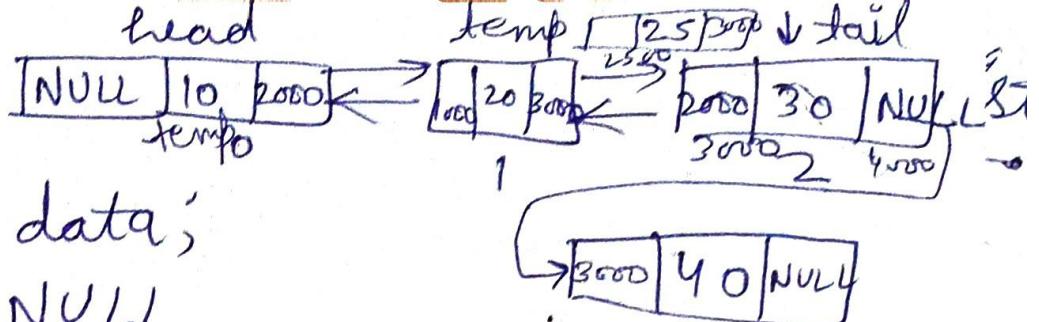
Insert at beginning,



~~ptr =~~       $\text{ptr} \rightarrow \text{info} = \text{data}$  }  
 $\text{ptr} \rightarrow \text{prev} = \text{NULL}$   
 $\text{ptr} \rightarrow \text{next} = \text{head}$   
 $\text{head} \rightarrow \text{prev} = \text{ptr}$   
 $\text{head} = \text{ptr}$

2	3
9	10
13	16
17	19
21	22
25	26
26	27
28	29
30	31
33	34
40	41
44	45
45	46
46	47
48	49
49	50
51	52
55	56
56	57
58	59
59	60
61	62
62	63
64	65
65	66
66	67
67	68
68	69
69	70
70	71
71	72
72	73
73	74
74	75
75	76
76	77
77	78
78	79
79	80
80	81
81	82
82	83
83	84
84	85
85	86
86	87
87	88
88	89
89	90
90	91
91	92
92	93
93	94
94	95
95	96
96	97
97	98
98	99
99	100

Ending :-



$\text{ptr} \rightarrow \text{info} = \text{data};$

$\text{ptr} \rightarrow \text{next} = \text{NULL}$

$\text{ptr} \rightarrow \text{prev} = \text{tail}$   
 $\text{tail} \rightarrow \text{next} = \text{ptr}$   
 $\text{tail} = \text{ptr}$

specific location -

B)  $\text{data} = 25$

$\text{loc} = 2$

1-  $\text{temp} = \text{head}$

Initialize counter I

Set  $I = 0$

repeat step 4 to 6 until  $I < \text{loc}$

Set  $\text{temp} = \text{temp} \rightarrow \text{next}$

$\text{ptr} \rightarrow \text{info} = \text{data}$

$\text{ptr} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$

~~$\text{temp} \rightarrow \text{next} = \text{ptr} = \text{ptr}$~~

$\text{ptr} \rightarrow \text{prev} = \text{temp}$

3  
2

3  
2

## Linked List Deleting Nodes

(B)

Deleting the Node from specific location in singly linked list. —

Delete\_location (Start, Loc)

Step 1 - check for underflow

if  $\text{ptr} = \text{NULL}$  then  
print underflow.

Exit

Step 2 - Initialize the counter I & pointers.

Set  $I = 0$ ;

Set  $\text{ptr} = \text{Start}$ ;

Step 3 - Repeat step 4 to 6 until  $I < \text{Loc}$

Step 4 - Set  $\text{temp} = \text{ptr}$

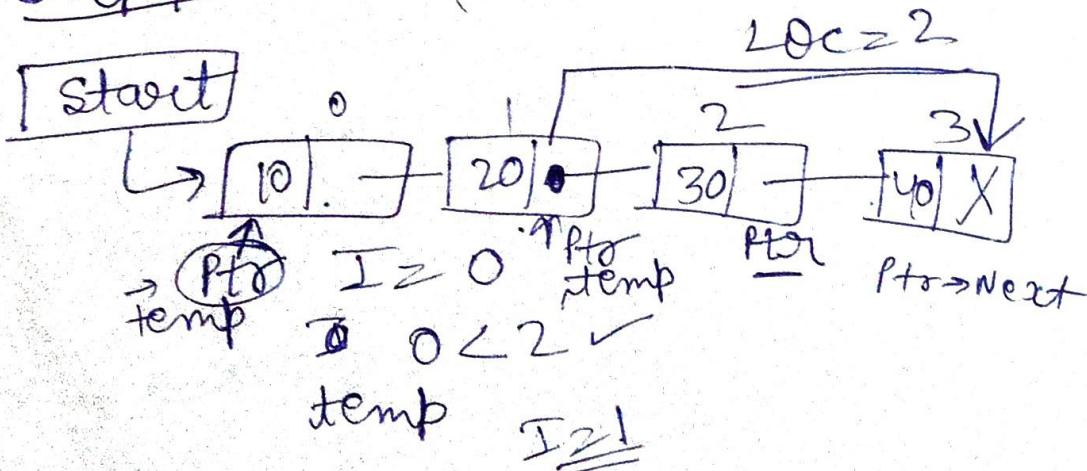
Step 5 - Set  $\text{ptr} = \text{ptr} \rightarrow \text{Next}$

Step 6 - Set  $I = I + 1$

Step 7 - Print element deleted is  $\text{ptr} \rightarrow \text{info}$

Step 8 - set  $\text{temp} \rightarrow \text{Next} = \text{ptr} \rightarrow \text{Next}$

Step 9 - Free( $\text{ptr}$ )



## Insert a node at Specific location-

Step1- Insert\_location (Start, item, Loc)

Step1- check for overflow

If  $\text{Ptr} == \text{NULL}$  then  
point overflow  
exit

else

$\text{ptr} = (\text{Node}*) \text{malloc}(\text{size of(Node)})$

Step2 - Set  $\text{ptr} \rightarrow \text{info} = \text{item}$

Step3- If  $\text{start} == \text{NULL}$  then  
Set  $\text{start} = \text{ptr}$

Set  $\text{ptr} \rightarrow \text{Next} = \text{NULL}$

Step4- Initialize the Counter I & pointer

Set  $I = 0$

Set  $\text{temp} = \text{start}$

Step5- Repeat steps 6 and 7 until  $I < \text{loc}$

Step6- Set  $\text{temp} = \text{temp} \rightarrow \text{Next}$

Step7- Set  $I = I + 1$

Step8- Set  $\text{ptr} \rightarrow \text{Next} = \text{temp} \rightarrow \text{Next}$

Step9- Set  $\text{temp} \rightarrow \text{Next} = \text{ptr}$

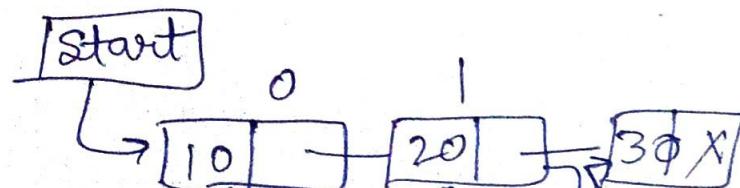
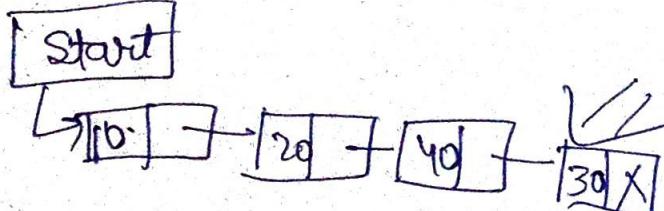


Diagram illustrating the insertion process:

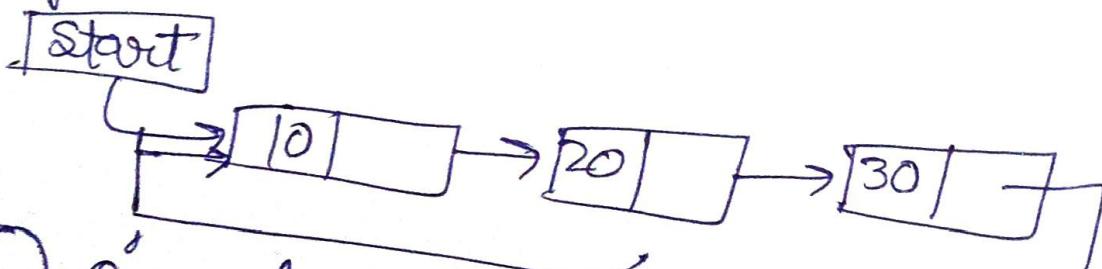
- Start pointer ( $\text{start}$ ) points to the head of the list.
- The list consists of four nodes with values 10, 20, 40, and 30X.
- The pointer  $\text{temp}$  is currently pointing to the node with value 10 (index 0).
- The pointer  $\text{ptr}$  (labeled as  $\text{step}$ ) contains the value 40.
- The variable  $I$  is set to 0.
- The condition  $0 < 1$  is true, so the loop continues.
- After one iteration,  $I = I + 1 = 0 + 1 = 1$ .
- The condition  $1 < 1$  is false, so the loop exits.

(17)

## Types of Linked Lists:-

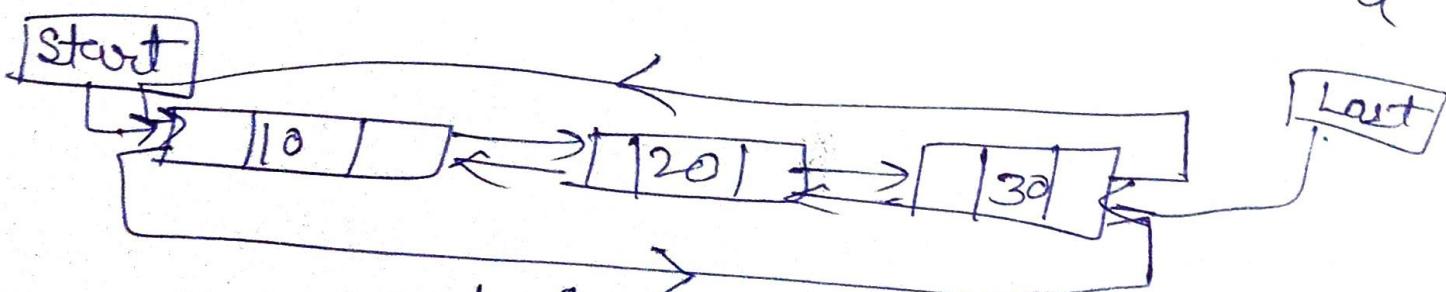
- singly linked list
- doubly linked list
- circular linked list
- ③ circular linked list

③ circular linked list :- It's one which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the link field of the last node.



## ④ circular Doubly linked list :-

It's one which has both the successor pointer and predecessor pointer in a circular manner.



## Create a node in circular linked list :-

ptr = (struct node\*)malloc(sizeof(node)); struct node

ptr → info = data

ptr → next = NULL

If head == NULL

head = ptr

```

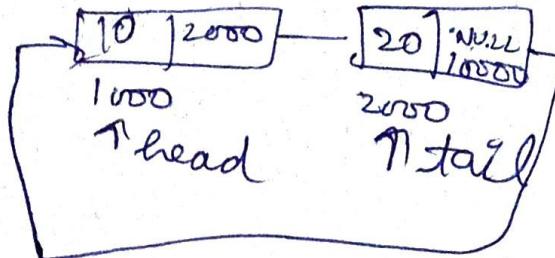
int data
struct node *next;
}
*ptr, *head, *tail;

```

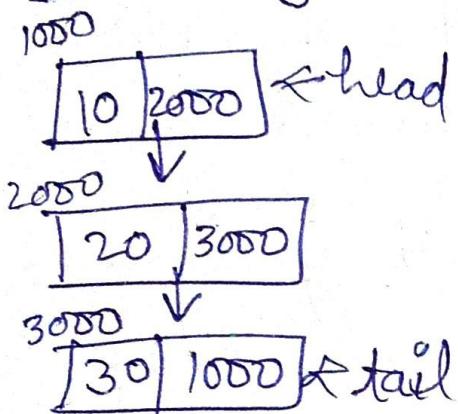
else

tail → next = pto  
tail = pto

tail → next = head;



Display -



10, 20, 30

✓  
tail  
temp = head;  
while (temp → next != head)  
{  
 printf("%d", temp → data);  
 temp = temp → next;  
}  
printf("%d", temp → data);

Inserting a node at the end of double circular linked list:-

