Data Structures

Unit-1

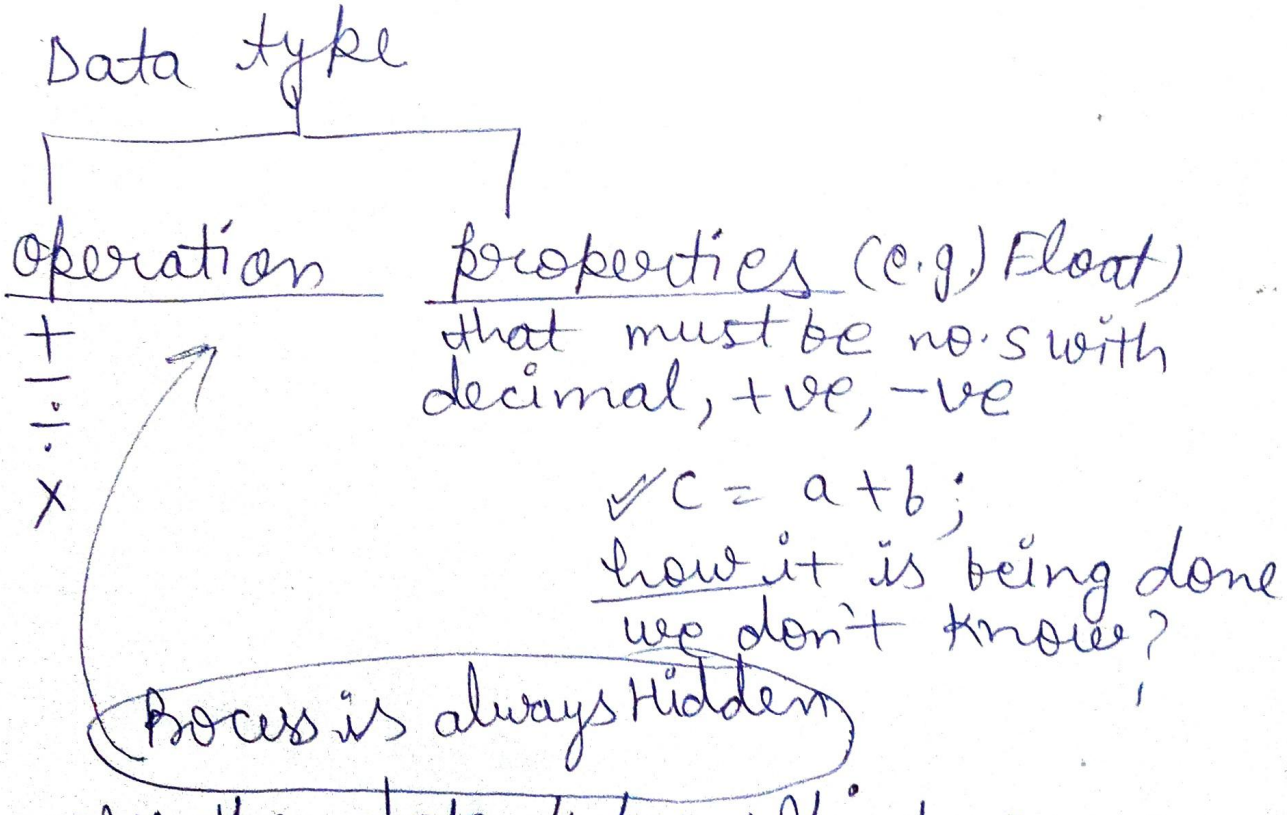
Abstract Data Types :-

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 x = 4
if x = a
char

Abstract Data type -



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.

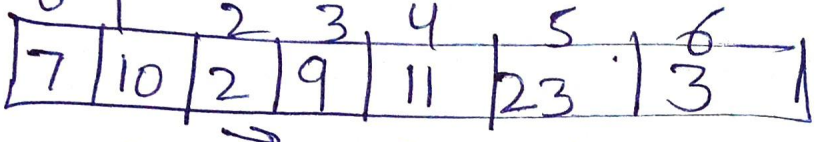
ADT are entities that are definitions of data and operations but don't have implementation.

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 structure

Linear V/S Binary Search—

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



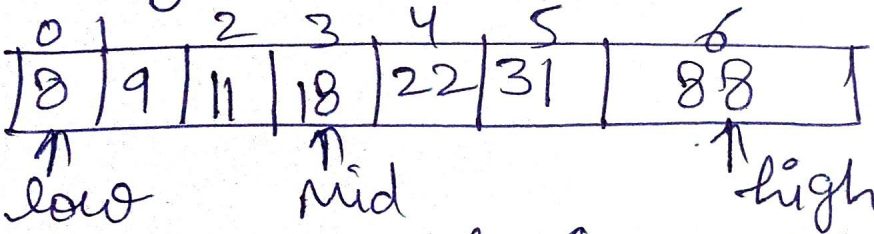
⇒ can be sorted or unsorted.

∴ Element found complexity : $O(n)$ (n elements)

search for 2

Binary Search—

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



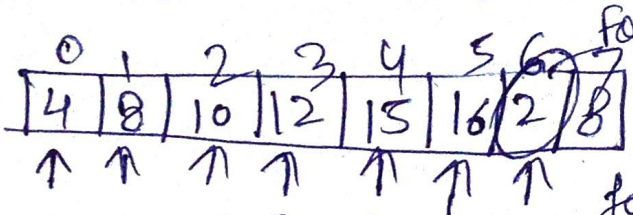
Time complexity $O(\log n)$

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

Linear	Binary
# Works on both sorted and unsorted arrays.	# works on sorted arrays
# Equality op^n's	# Equality op^n
# $O(n)$	# $O(\log n)$

Linear search is done through traversal

e.g.



if search for 2

e.g. For card

(Sorted or Unsorted) for

Binary search -

Search for (238) Page

250 Half ← 1 Half

1 500

(238) converge

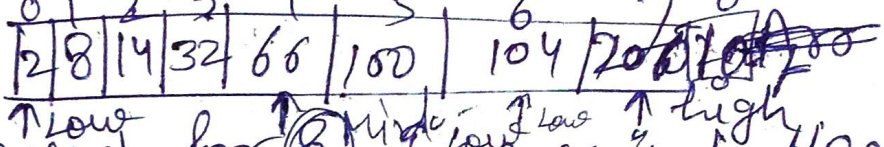
1000 Pages

Book

in sorted Array must be sorted.

not sorted X

e.g.



if search for (8) Linear is better time zyada

(200). Binary is better for sorted long array.

$\frac{Low + High}{2}$ = greatest integer

Search 200

Low	high	Mid	
0	8	4 ⇒ 8/2 No.	element found or not
4	8	6 ⇒ No	12/2
6	8	7 ⇒ yes	Binary

Traversal –

```
for (i=0; i<n; i++)
{
printf(arr[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 ~~6~~ (11)

Pass 1

9 10 6 11 (15) 2

9 10 6 11 2] 15 — largest no is on the correct position

n+1

(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

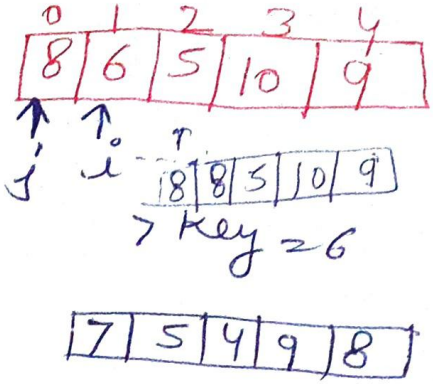
Total no of swapping

time complexity

Insertion Sort—

```
Sort(int a[], int n)
{
  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;
    }
    a[j+1] = key;
  }
}
```

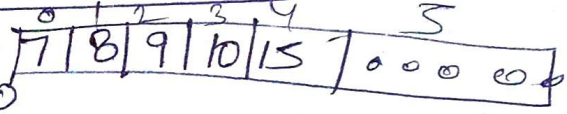
e.g. 8 6 5 10 9




```
Bubble Sort(int a[], n)
{
  int swapped i, j;
  for(i=0; i<n; i++)
  {
    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>
int linearSearch(int arr[], int size, int
                 element)
{
for(int i=0; i<size; i++)
  {
   if(arr[i] == element)
     {
      return i;
     }
  }
else
return -1;
```

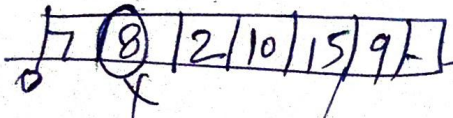
Operations on Arrays —



* Traversal
* Insertion — index no 2 → 5
* Deletion
* Searching

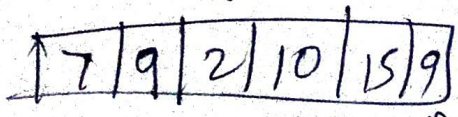
Case 1 — Maintain the order
Best case insertion - O(1)
Worst case - O(n)
[1st place]

Case 2 — No need to maintain the order



Delete - 7 2 10 15 9 — Size-1

Shifts worst case

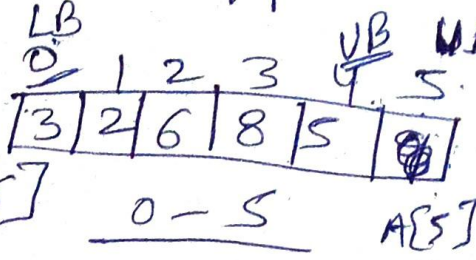


← Best case

No of element = Upper bound − Lower bound + 1

Insertion at end −

Let A be an array of size Max. or N

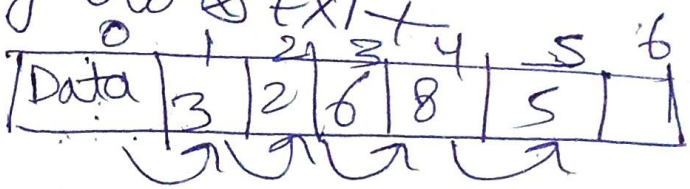


1- If UB == Max →
then print overflow & Exit.
2- Read data
3- UB = UB + 1 UB = 4
4- A[UB] = data UB = 5
5- Stop or Exit

Insertion at Beginning −

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

1- If UB == Max →
then print overflow & Exit
2- Read data
3- UB = K
4- Repeat steps 5
while K ≥ LB
5- A[K+1] = A[K]
K = K − 1;
6- A[LB] = data
7- Stop.



A[5] = A[4]
A[4] = A[3]

Insertion at a location

Let A be an array of size N

1- if UB = = Max →
then print overflow & EXIT

2- Read data

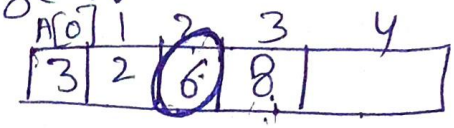
3- Read loc

4- UB = K

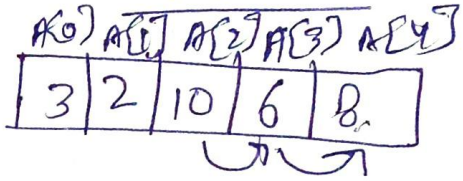
5- Repeat step 6
while K ≥ loc

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

7- A[LOC] = Data
stop



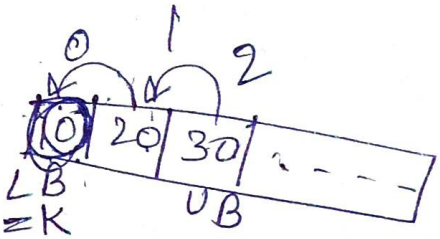
A[2] = 10



A[4] = A[3]

Deletion Algorithm – Array –

Deletion from Beginning –



1- If UB = 0 then print
'Underflow'

2- K = LB

3- Repeat step 4
while K < UB

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

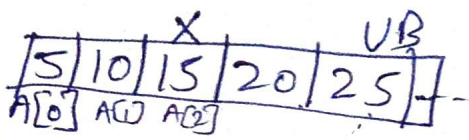
5- K = K+1

6- A[UB] = NULL
UB = UB-1

Stop.

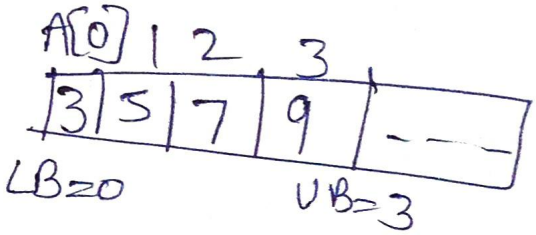
Deletion of Particular element -

1- If UB=0, print 'Underflow'
2- Read data
3- ~~Read~~ K=LB
4- Repeat step 5
   while A[K] ≠ Data
5- K = K+1
6- Repeat step 7
   while K ≤ UB
7- A[K] = A[K+1]
   K = K+1
8- A[UB] = NULL
   UB = UB-1
   stop



Deletion from End -

1- If UB == NULL then
   ~~Or~~ Underflow
2- A[UB] = NULL
3- UB = UB-1
4- Stop



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 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() → Push an element into the stack.
2- POP() → Remove the topmost element from the stack.
3- isempty / is full
4- Top()

Infix, Prefix & Postfix :-

Notation to write an expression —

1- **Infix** — (a) a+b (c) p/q
(b) a-b (d) b÷y

2- **Prefix** — (Operand <Operator> Operand) x-y

5+5
5, 5, +
+ 5, 5 (Pre)

<Operator> <operand> <Operand>
+ab
-xy
*pq

(Post)

```
isempty()
{
 if(Top == -1)
   return True
   else
    return false
   }
PoP()
{
 int Data
 if(! isempty)
   {
    Data = stack[TOP]
    Top = Top-1
    }
    else
    {
     "underflow"
          }
```

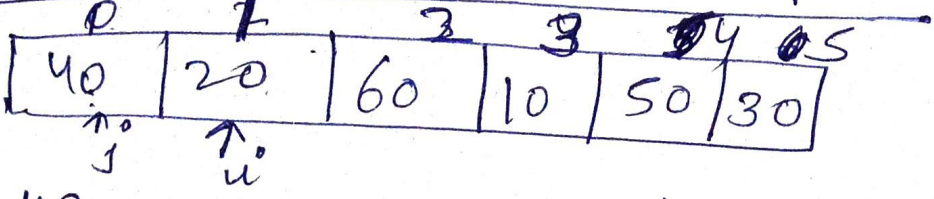
⇒ A stack is a non-primitive linear data structure it is an ordered list in which addⁿ of a new data item and deletion of already existing data item is done from only one end known as top of stack (TOS)

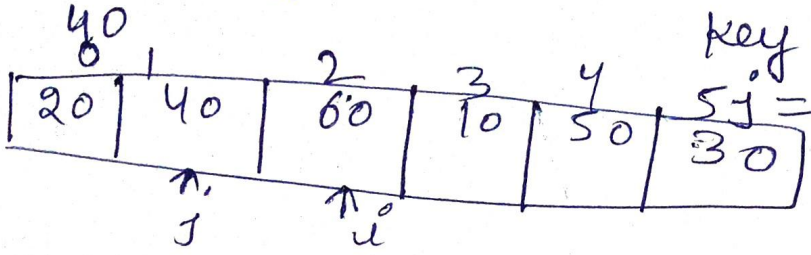
⇒ Most frequently accessible element in the stack is the top most element, whereas the least accessible element is the bottom of the stack.

Applications of stack:—

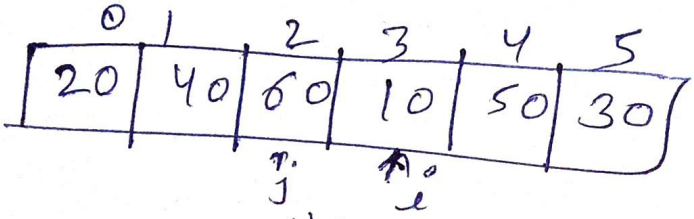
1- Used in function calls
2- Infix to postfix conversion (and other similar conversion)
3- Parenthesis matching & more.

Insertion Sort example-

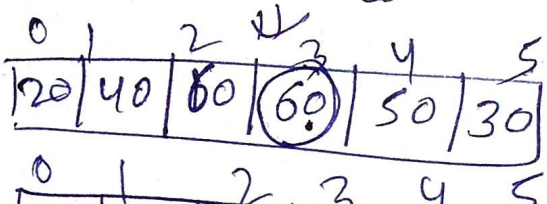


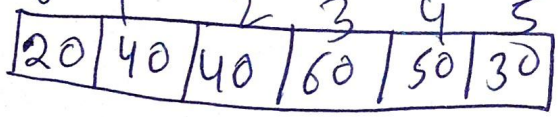


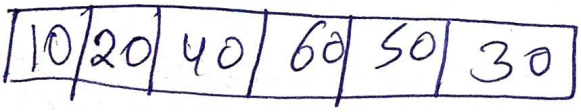
Key = 20
j = 0



Key = 60
j = 1
40 > 60 (false)







Key = 10
j = 2
60 > 10
j = j - 1
j = 1
a[2] = 10

Stack :- (FILO (OR) 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 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.

e.g. A+B*(C+D)/F+D*E infix to prefix -

A+B*(+CD)/F+D*E

A+*B+CD/F+D*E

A+/*B+CDF+D*E

A+/*B+CDF+*DE

+A/*B+CDF+*DE

++A/*B+CDF*DE

( )
/ *
+ −

3- Postfix - <operand1> <operand2> <operator>

e.g. ab+

XY-

pq*

Infix	Prefix	Postfix
a*b	*ab	ab*
a-b	-ab	ab-

Infix: ((A* (B+C))*D) (conversion ⇓)

Left → Right

Postfix - A B C + * D *

(56)

A 8 * D *

16 D *

Q1- X-Y*Z convert this to prefix to postfix -

Prefix → Step 1 - Parenthesize the expression

Polish notation (X-(Y*Z))

(X-[*YZ])

-X[*YZ]

Postfix - (X-(Y*Z))

X-[YZ*]

XYZ*-

Q2- p-q-r/a

(p-q)-(r/a)

Prefix -

(-pq)-(/ra)

--pq/ra

Postfix -

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

convert this into postfix expression.

⇒ abc*+de*f+g*+

Q3- A+(B*C-(D/E↑F)*G)*H

⇓

ABC*DEF↑/G*-H*+

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

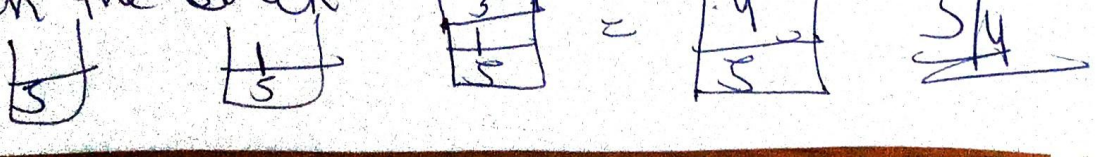
Q4

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

Push 7 on the stack -

Push 2

next is operator remove top 2 elements of stack & perform 7-2=5 place this result back on the stack



Postfix - A+B*CD+/F+D*E

A+BCD+*/F+D*E

A+BCD+*F/+D*E

A+BCD+*F/+DE*

ABCD+*F/+DE*+

ABCD+*F/+DE*+

( )
/ * ↓
+ -

e.g: 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* 62+ - (12/4)

5* 62+ - 124/

562+* - 124/ ⇒

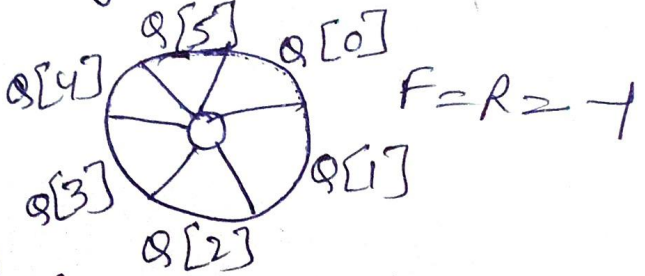
562+*124/-

Queue ADT:-

1) Enqueue(X) → Insert X
2) Dequeue() → Delete
3) Front()
4) Isempty() → T/F
5) Is Full()

Appⁿ - 1) Printer
2) Operating system in scheduling

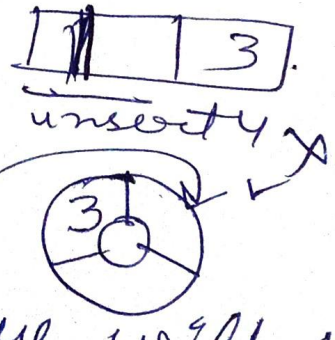
Circular Queue - A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of queue is full.



A circular queue overcome the problem of unutilized 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.

$$Rear = Rear + 1$$

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

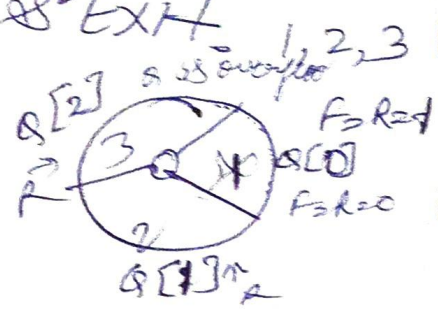
$$Front = Front + 1$$

Insertion in circular Queue –

CINSERT (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] → Queue[Rear] = value
[End if]



f = 1
f == (2+1) % 3

Q⇒ A X B − (C + D) x (P / Q)

Q− A + (B X C − (D / E ↑ F) X H)

Q − A / B ↑ C − D

Q − (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
   front = front + 1
3- Exit

i.e. only one element is present in queue.

Q- Convert A/B↑C-D into prefix form.

⇓

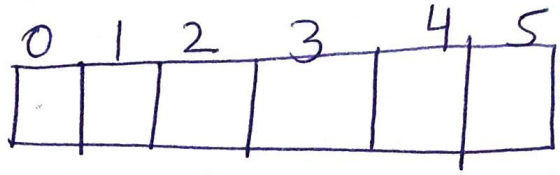
-/A↑BCD

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

⇓

* -A/BC - *DEF

Queue-



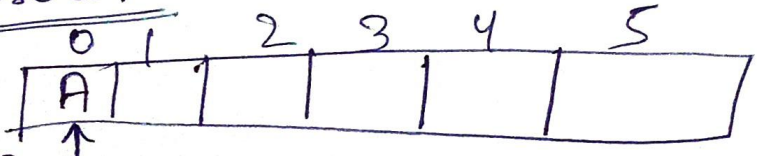
front = -1
Rear = -1

i.e. Empty

FIFO
OR
LILO

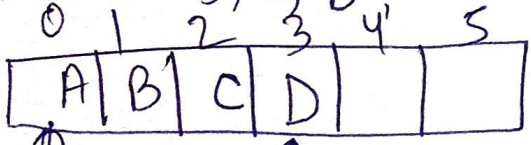
Insert in Queue
En Queue
Delete
De Queue

Insert A-



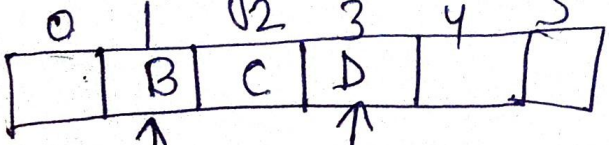
F=R=0

Insert B, C, D -



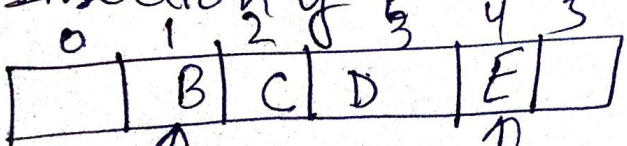
F=0 R=3

Deletion of element A -



F=1 R=3

Insertion of E -



F=1 R=4

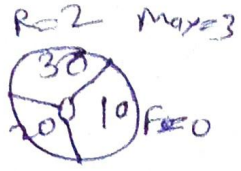
* Queue is a non-primitive linear data structure.

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

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

Delete operation in circular Queue — 12

```
Delete (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]
         item deleted
2- Exit
```



e.g. item = q[F]
item = q[0]
item = 10

if (F = = R)
0 = = 2 false
F = (F+1) %. 3
F = (0+1) %. 3
F = 1

10 is deleted.

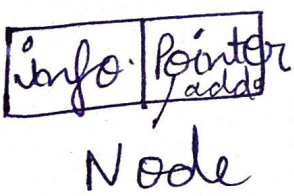
Linked list

A linked list is a linear data structure in which the elements are not stored at contiguous memory location.

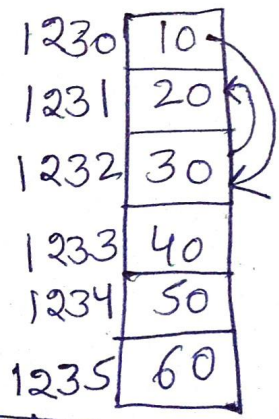
A linked list is a dynamic data structure. The no. of nodes in a list is not fixed and can grow and shrink on demand.

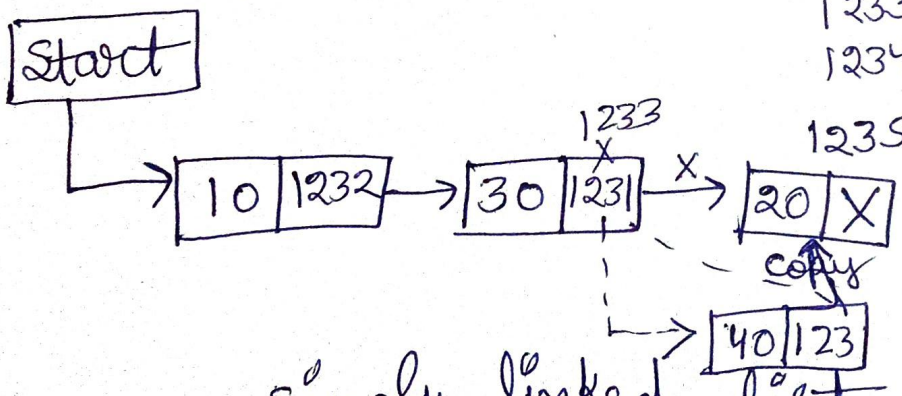
Each element is called a node, which has two parts.

Information part which stores the info & pointer, which points to the next element.

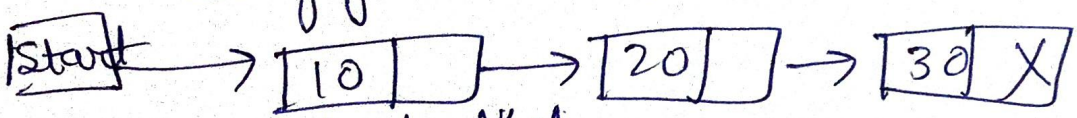


Node

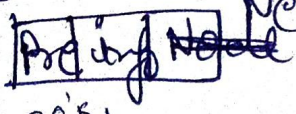


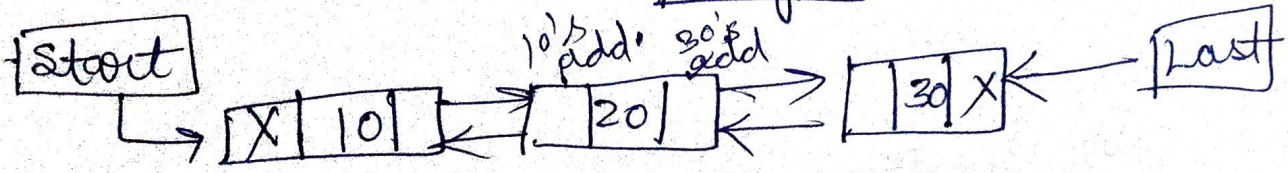


Singly linked list



Doubly linked list —





Deleting First Node from Linked List

Delete First(start)

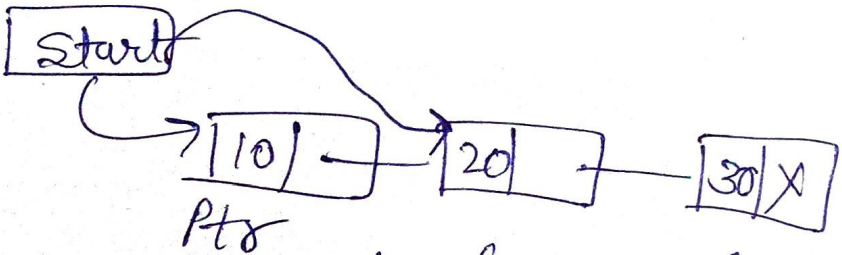
Step1 - check for underflow
if start = NULL then
print linked list empty.
Exit

Step2 - set Ptr = start

Step3 set start = start → Next

Step4 - Print element deleted is
ptr → info

Step5 - free(ptr)

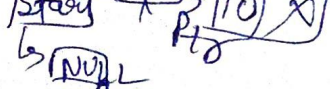


Delete Last Node from Linked list :-

Delete(start)

Step1 check for underflow
if start = NULL then
print Linked list is empty.

Step2 - if start → Next = NULL then
Set Ptr = start
Set start = NULL



print element deleted is = Ptr → info
free(Ptr)
end if

Step3 - set Ptr = start

Step4 - Repeat Step5 & 6 untill
Ptr → Next != NULL

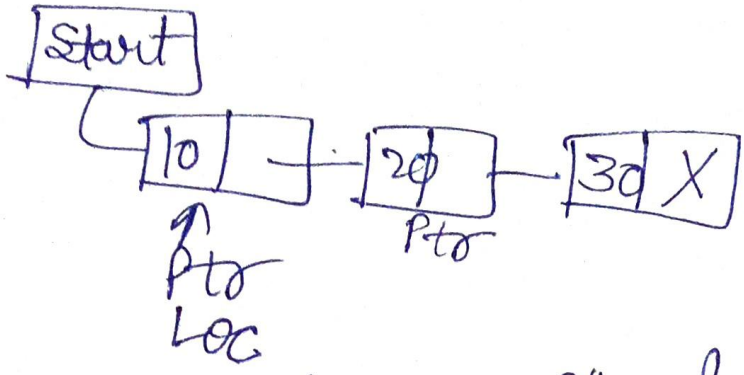
Step5- Set Loc = Ptr

Step6- Set Ptr = Ptr → Next

Step7- Set Loc → Next = NULL

Step8- free(ptr)

e.g:



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 insertions & deletion at known position is O(n)

Doubly linked list

Doubly linked list has nodes with data field & two pointer field (Bckward & 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 -

Algorithm -

Insert_Last(START, ITEM)

Step1 - check for overflow
If Ptr = NULL then print overflow.
Exit
else
Ptr = (Node*) malloc(sizeof(node));

step2 - set ptr→info = item

step3 - set ptr→Next = NULL

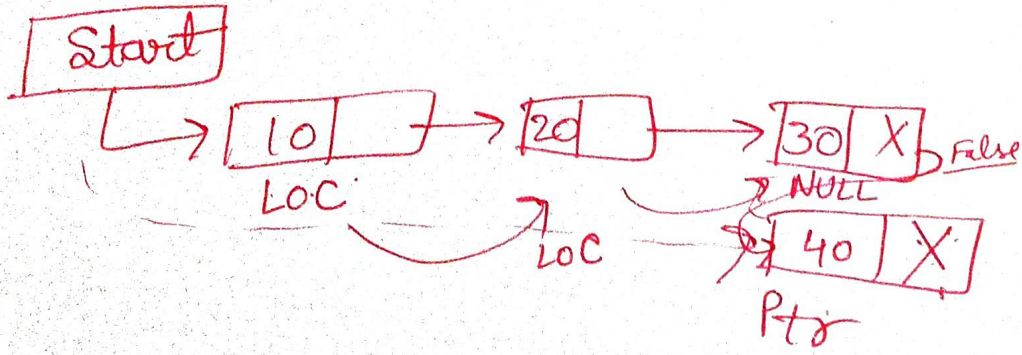
step4 - If start = NULL and then
set start = ptr;
else

step5 - set Loc = Start

step6 - Repeat step 7 until Loc→Next != NULL

step7 - set Loc = Loc→Next

step8 - set Loc→Next = Ptr;



Insert node at beginning of linked list

Algorithm—

Insert_first(start, item)

Step 1- [Check for overflow]

If Ptr = NULL then

print overflow

exit

else

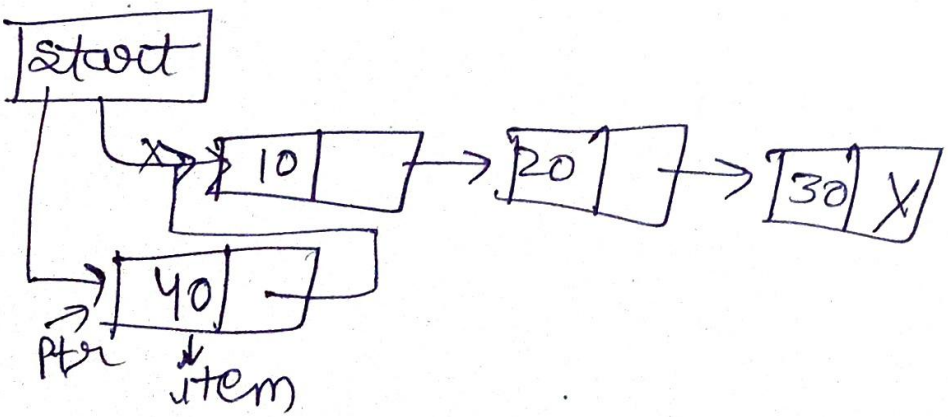
Ptr = (Node *) malloc (size of(Node)).

// create new node from memory and assign its address to ptr //

Step 2- Set Ptr → info = item

Step 3- Set Ptr → ~~info = item~~ next = start

step 4 set start = Ptr



Doubly linked list

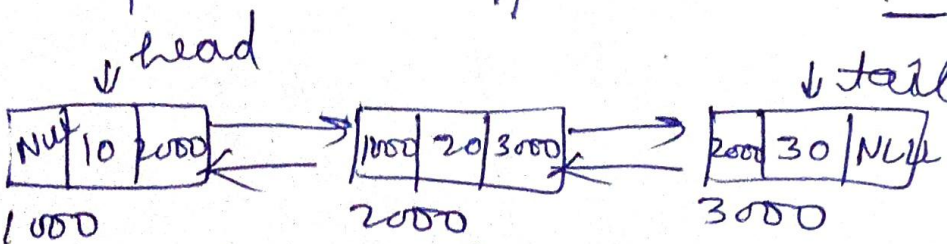
(15)

It is the linear collection of data item called node where each node has devided 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.

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

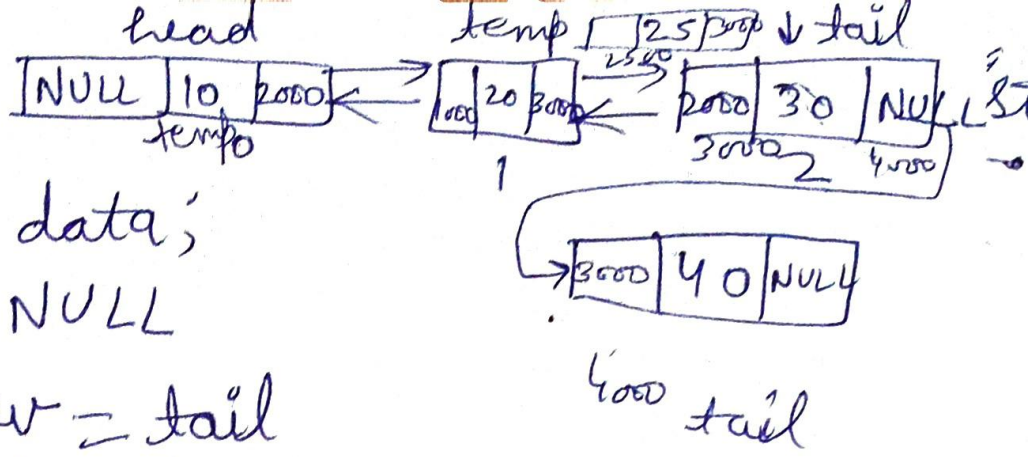
Previous	Data	Next
—	A	2
1	N	3
2	J	4
3	S	5
4	H	—



Insert at beginning,

~~next~~ ptr → info = data
ptr → prev = NULL
ptr → next = head
head → prev = ptr
head = ptr

Ending :-




```
ptr -> info = data;
ptr -> next = NULL
ptr -> prev = tail
tail -> next = ptr
tail = ptr
```

specific location-

data = 25
loc = (2)

1- temp = head
Initialize counter I
Set I = 0
repeat step 4 to 6 untill I < loc
Set temp = temp -> next
ptr -> info = data
ptr -> next = temp -> next
temp -> next = ptr
ptr -> prev = temp

Linked List Deleting Nodes

Deleting the Node from specific location in Singly linked list:-

Delete_location (start, loc)

Step1 - check for underflow
if ~~Ptr~~ start = NULL then
Print underflow.
Exit

Step2 - Initialize the counter I & pointers.
Set I = 0;
Set ptr = start;

Step3 - Repeat step 4 to 6 until I < loc

Step4 - Set temp = Ptr

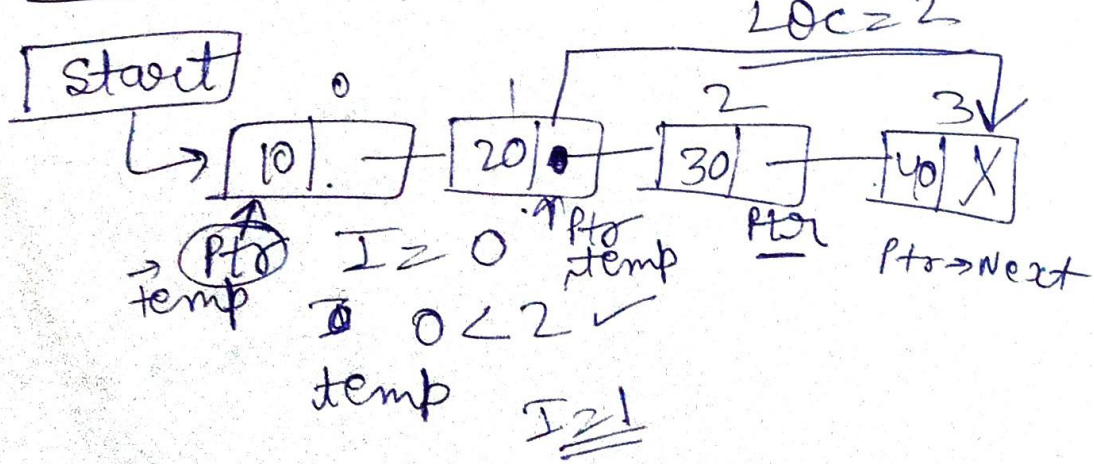
Step5 - Set Ptr = Ptr → Next

Step6 - Set I = I + 1

Step7 - Print element deleted is ptr → info

Step8 - Set temp → Next = Ptr → Next

Step9 - Free (Ptr)



Insert a node at specific location-

Step1 - Insert_location(start, item, Loc)

Step1 = check for overflow
If Ptr == NULL then
print overflow
exit
else
Ptr = (Node*) malloc(sizeof(Node))

Step2 - Set ptr → info = item

Step3 - If start = NULL then
Set start = Ptr
Set Ptr → Next = NULL

Step4 - Initialize the counter I & pointer
Set I = 0
Set temp = start

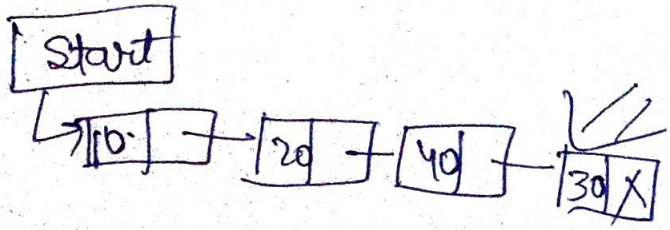
Step5 - Repeat steps 6 and 7 untill I < loc

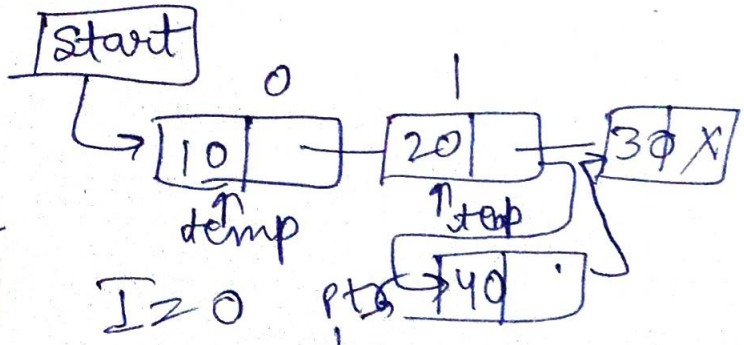
Step6 - Set temp = temp → Next

Step7 - Set I = I + 1

Step8 - Set Ptr → Next = temp → Next

Step9 - Set temp → Next = ptr





I = 0

0 < 1

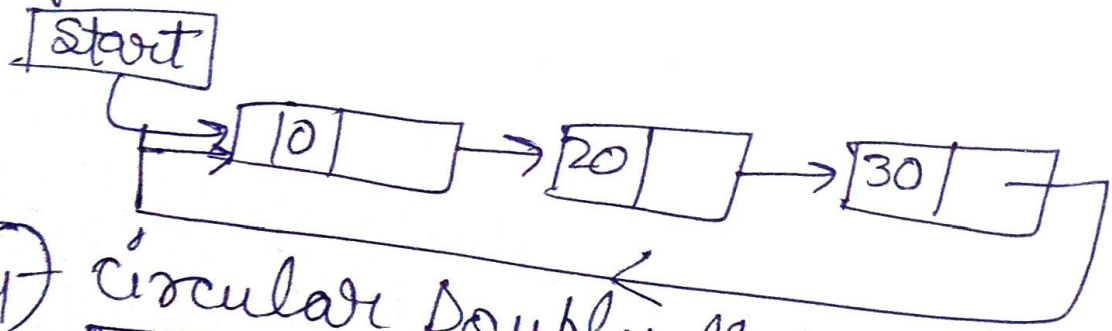
I = I + 1 = 0 + 1

1 < 1 false

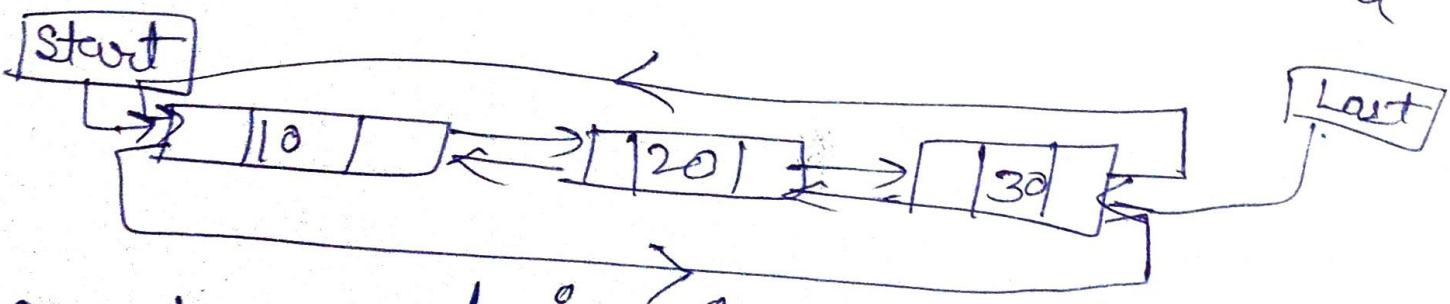
Types of Linked Lists:-

- singly linked list
- Doubly Linked list
- Circular linked list
- Circular doubly 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.

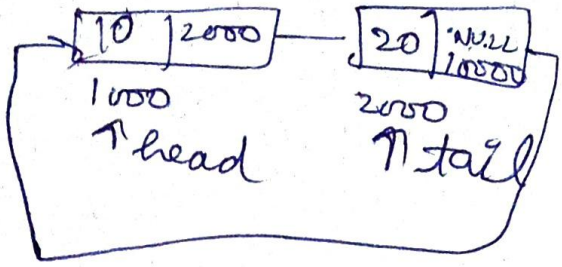


Create a node in circular linked list:-

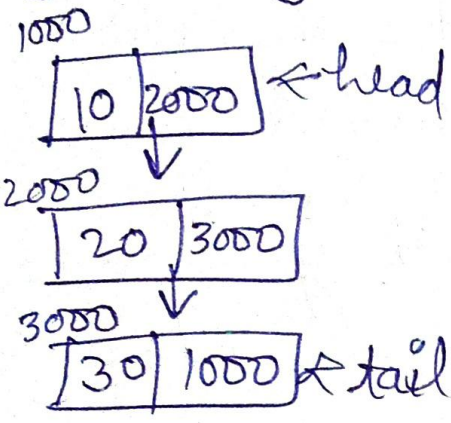
```
ptr = (struct node *) malloc(sizeof(Node));
ptr -> info = data
ptr -> next = NULL
if head == NULL
    head = ptr
    tail = ptr
```

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

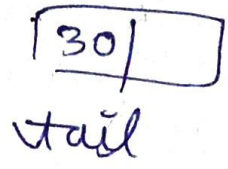
```
else
    tail->next = ptr
    tail = ptr
    tail->next = head;
```



Display -



10, 20, 30




```
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:-