Data Structure

Data ⇒ Anything to give information is called data.

Ex ⇒ StudentName, Student Rollno.

Structure ⇒ Representation of data is called structure.

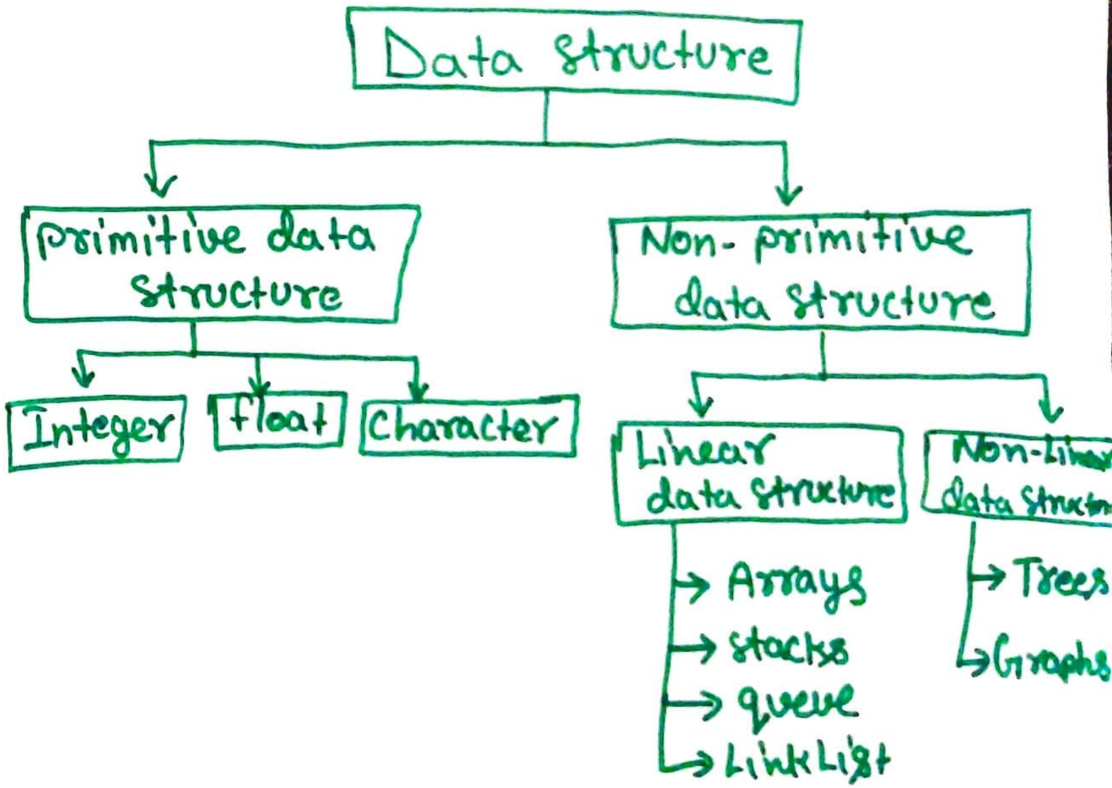
Ex ⇒ graph, Arrays, List.

Data Structure ⇒

- Data Structure = Data + Structure
- Data Structure is a way to store and organize data so that it can be used efficiently (better way)
- Data structure is a way of organizing all data items and relationship to Each other.

Types of data Structure ⇒

There are mainly two types of data Structure.



Primitive data structure ⇒ These are basic structure and are directly operated by machine instruction.

Ex ⇒ integer, float, Character.

Non-primitive data structure ⇒ These are derived from the primitive data structure. it's a Collection of same type or different type primitive data structure.

Ex ⇒ Arrays, Stack, trees.

Data structure operation ⇒

The data which is stored in our data structure are processed by some set of operatio

i) Insertion ⇒ Add a new data in the data structu

ii) Deleting ⇒ Remove a data from the data structure

iii) Sorting ⇒ Arrange data in increasing or decreasing order.

iv) Searching ⇒ find the Location of data in data structure.

v) merging ⇒ Combining the data of two different sorted files into a single sorted file.

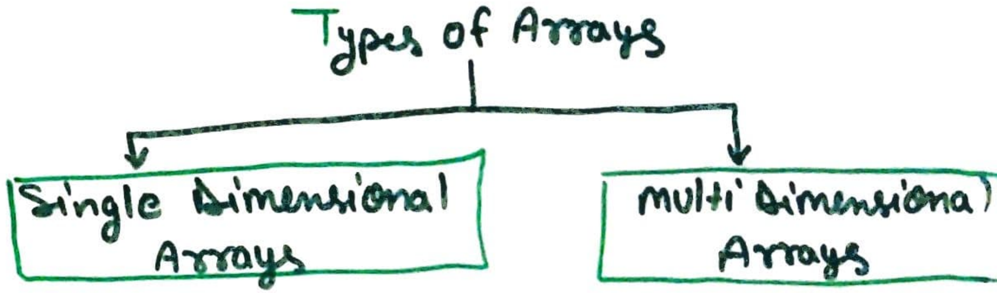
vi) Traversing ⇒ Accessing Each data Exactly one in the data Structure so that Each data item is traversed or Visited.

Arrays

- An Array Can be defined as an infinite Collection of homogeneous (Similar type) Elements.

- Array are always stored in Consecutive (specific) memory Location.

- Array Can be Store multiple values which Can be referenced by a single name.



1) Single Dimensional Arrays ⇒ • It's also known as One Dimensional (1D) Array.

- It's use only one Subscript to define the Elements of Arrays.

[Row] [Col]

Declaration ⇒

data-type var_name [Expression];
Size

Ex ⇒

```
int num [10];
Char C [5];
```

Initializing one-Dimensional Array ⇒

Data-type var-name [Expression] = { Values};

Ex ⇒

```
int num [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
Char a [5] = { 'A', 'B', 'C', 'D', 'E'};
```

2) **Multi-Dimensional Arrays ⇒** multidimensio Arrays use more then one Subscript to describe the Arrays Elements. [ ][ ][ ] —

Two Dimensional Arrays ⇒ It's use two Subscript, one Subscript to represent row value and second Subscript to represent Column value.

[row] [colum]

It mainly use for matrix Representation.

Declaration two-Dimensional Arrays ⇒

data-type var-name [rows] [colums];

Ex ⇒

```
int num [3] [2];
```

Initialization 2-D Arrays ⇒

data-type var-name [rows] [colums] = { values};

Ex ⇒

```
int num [3] [2] = { 1, 2, 3, 4, 5, 6};
or
int num [■] [ ] = { 1, 2, 3, 4, 5, 6};
```

$$\underset{3\times2}{\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}}$$

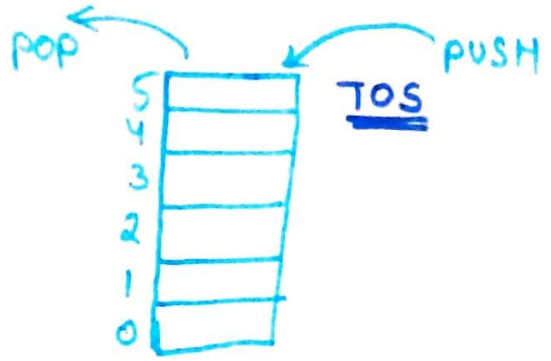
```
num[0,0] = 1
nu[01] = 2
num[10] = 3
num[11] = 4
num[20] = 5
nu [21] = 6
```

write a program to read & write one Dimensional Array.

```
# include <Stdio.h> — Standard input out
                      printf Scanf
# include < Conio.h> → Console input out
void main()                Clrscr(), getch()
{
  ~~int a[10], i~~  int a[10], i;
   Clrscr();
printf (" Enter the Array Elements");
 for (i=0; i<=9; i++)
 {
   Scanf ("%d", &a[i]);
 }
printf (" the Entered Array is");
 for (i=0; i<=9; i++)
 {
   printf ("%d\n", a[i]);
 }
   getch();
 }
```

Stacks (Data Structure)

- Stack is a Non-primitive Linear data Structure.
- It is an ordered List in which addition of new data item and deletion of already Existing data item is done from only one End known as Top of stack (TOS)



- The Last added Element will be the first tobe Removed from the Stack. This is the reason stack is Called Last-in-first out (LIFO) type of List.

Operations on stack.

There are two operation of stack.

1> PUSH operation ⇒ • The process of adding a new Element to the top of stack is Called PUSH operation.

- Every new Element is adding to stack top is incremented by one.
- In Case the array is full and no new Elemnt Can be added it's called Stack full or Stack overflow Condition

2> POP operation ⇒ • The process of deleting an Element from the top of Stack is called POP operation

- After Every pop operation the Stack (TOP) is decremented by one.
- If there is no Element on the stack and the pop is performed then this will result into Stack underflow Condition.

Stack operation & Algorithm

➧ Stack has two operation.

1) PUSH operation ➧

2) pop operation ⇒

1) PUSH operation ⇒ • The process of Adding a new Element of the top of stack is Called PUSH operation

- Every PUSH operation Top is incremented by one.

$$\boxed{TOP = TOP + 1}$$

- In Case the Array is full no new Element is added. this Condition is Called Stack full or Stack overflow Condition.

Algorithm for inserting an item into the stack (PUSH operation).

PUSH (Stack [maxSize], item)

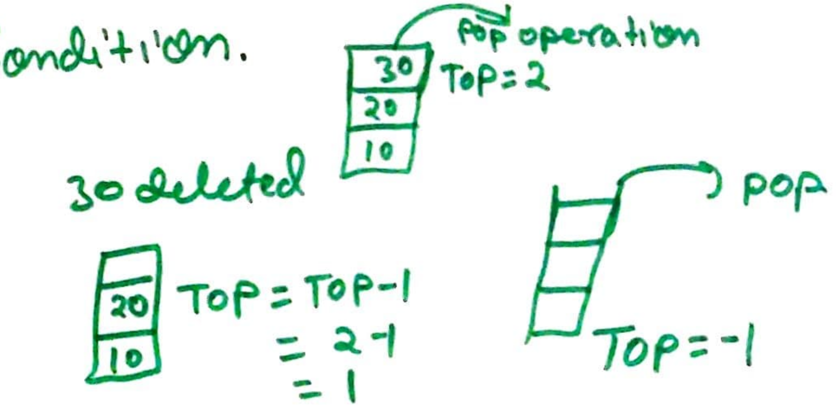
```
Step 1: initialize
            Set top = -1
Step2: Repeat Steps 3 to 5 until Top < maxSize-1
Step3: Read Item
Step4: Set top = top+1
Step5: Set Stack [TOP] = item
Step 6: print "Stack overflow"
```

2 → pop Operation ⇒

- The process of Deleting an Element from the top of Stack is Called POP operation.
- After Every pop operation the Stack Top is decremented by one.

$$Top = Top - 1$$

- If there is no Element on the Stack and the pop operation is performed then this will result into STACK UNDERFLOW Condition.



Algorithm for deleting an item from the Stack (POP)

```
pop (Stack [maxSize], item)

Step1: Repeat Steps 2 to 4 until $Top \geq 0$

Step2:  Set item = Stack [TOP]

Step3:  Set top = top - 1

Step4:  print, No. deleted is, Item

Step5:  Print stack under flows.
```

Stacks (prefix & postfix)

Stack Notation ⇒ There are three stack Notation.

1) Infix Notation ⇒ where the operator is written in-between the operands.

Ex ⇒ A + B + operator
A, B operands

2) Prefix Notation ⇒ In this operator is written before the operands. It is also known as polish Notation.

Ex ⇒ + A B

3) Posfix Notation ⇒ In this operator is written After the operands. It is also known as Suffix Notation.

Ex ⇒ AB+

Q ⇒ Convert the following Infix to prefix and postfix for $(A+B) * C/D + E^F/G$

prefix ⇒ $(A+B) * C/D + E^F/G$

$+AB * C/D + E^F/G$

Let $\underline{+AB = R_1}$

$R_1 * C/D + E \wedge F/G$

$R_1 * C/D + \wedge EF/G$

Let ⇒ $\underline{\wedge EF = R_2}$

$R_1 * C/D + R_2/G$

$R_1 * /CD + R_2/G$

Let ⇒ $\underline{/CD = R_3}$

$R_1 * R_3 + R_2/G$

$R_1 * R_3 + /R_2G$

Let ⇒ $\underline{/R_2G = R_4}$

$R_1 * R_3 + R_4$

$* R_1 R_3 + R_4$

Let $* \underline{R_1 R_3 = R_5}$

$R_5 + R_4$

$\underline{+ R_5 R_4}$

Now Enter the value of R_5, R_4, R_3, R_2, R_1

$+ * R_1 R_3 / R_2 G$

$+ * + AB/CD/\wedge EFG$ ✓

postfix ⇒ $(A+B) * C/D + E \wedge F/G$

$\boxed{AB+} * C/D + E \wedge F/G$

$\underline{\text{Let } AB+ = R_1}$

$R_1 \underline{*} C \underline{/} D \underline{+} E \underline{\wedge} F/G$

$R_1 * C/D + \boxed{EF\wedge}/G$

$\underline{\text{Let } EF\wedge = R_2}$

$R_1 \underline{*} C \underline{/} D \underline{+} R_2 \underline{/} G$

$R_1 * \boxed{CD/} + R_2/G$

$\underline{\text{Let } CD/ = R_3}$

$R_1 * R_3 + R_2/G$

$R_1 * R_3 + \boxed{R_2G/}$

$\underline{\text{Let } R_2G/ = R_4}$

$R_1 \underline{*} R_3 \underline{+} R_4$

$\boxed{R_1 R_3 *} + R_4$

Let $\underline{R_1 R_3 * = R_5}$

$R_5 + R_4$

$\underline{R_5 R_4 +}$

Now Enter the value of $R_5, R_4, R_3, R_2 R_1$

$R_5 R_4 +$

$R_1 R_3 * R_4 +$

$AB+CD/ * R_2 G/ +$

$AB+CD/ * \boxed{EF\wedge} G/+$

postfix Expression

prefix and postfix using tabular form

Ex ⇒ Convert (A+B*C) into prefix and postfix using tabular form

to convert in prefix following operation perform

1) Reverse the input string
2) perform tabular method and find postfix expression.
3) Reverse this postfix Expression string to find the prefix.

Priority
^ → highest
*, / → 2 highest
+ - → Lowest priority

Ex ⇒ A + B*C

first to Add bracates

(A + B*C)

Reverse string

(C*B + A)

Tabular form.

Symbol Scanned	stack	postfix Expression
(	(	—
C	(	C
*	(*	C
B	(*	CB
+	(+	CB*
A	(+	CB*A
)	—	CB*A+

So the postfix Expression CB*A+. Now reverse this Expression to get the prefix so prefix is

+A*BC prefix

to Convert postfix ⇒ Direct perform tabular form (A+B*C)

Symbol Scanned	Stack	postfix Expression
		—
(	(	A
A	(	A
+	(+	AB
B	(+	AB
*	(+*	ABC
C	(+*	ABC*+
)		

postfix Expression = ABC*+

Queues

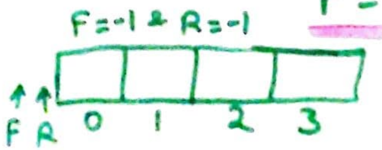
- Queue is a Non-primitive Linear data structure.
- It is an homogeneous Collection of Elements in which new Elements are added at one End Called the Rear End, and the Existing Element are deleted from other End called the front End.
- The first added Element will be the first to be remove from the queue. that is the reason queue is Called (FIFO) first-in first out type List.
- In queue Every insert operation Rear is incremented by one

$$R = R+1$$

and Every deleted operation front is incremented by one

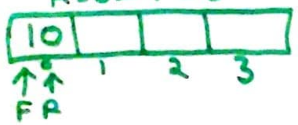
$$F = F+1$$

Ex ⇒

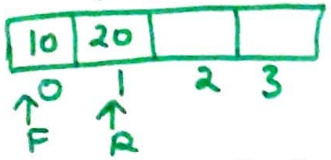


Empty queue

insert 10
R=0 & F=0

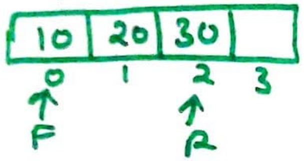


insert 20 R=1 & F=0



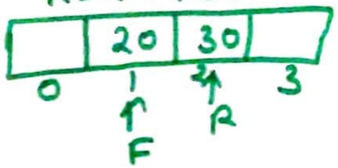
insert 30

R=2 & F=0



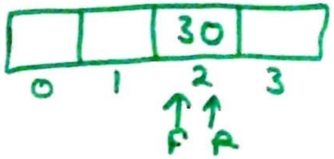
deleted Element. first delete 10

R=2 & F=1



deleted second element.

R=2 & F=2



Operation on Queue

1) To insert an Element in a Queue ⇒

Algo ⇒

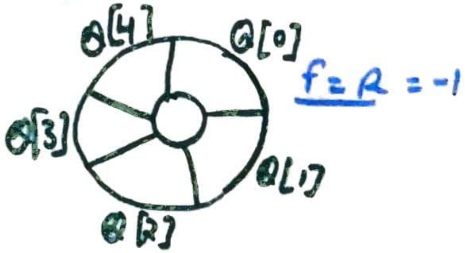
```
                QINSERT [QUEUE[maxSize], ITEM]
Step1 :  Initialization
                 Set front = -1
                 set Rear = -1
Step 2:  Repeat Steps 3 to 5 until
                       Rear < maxSize -1
Step 3:  Read item
Step4 :    if front = = -1 then
              front = 0
              Rear = 0
         else
              Rear = Rear+ 1
Step5 :  Set QUEUE[Rear] = item
Step6 :  print, Queue is overflow
```

2) To Delete an Element from the queue >

```
QDELETE (Queue[maxsize], item)
Step 1: Repeat step 2 to 4 until front >= 0
Step 2: Set item = Queue[front]
Step 3: If front == Rear
            Set front = -1
            Set Rear = -1
        Else
            front = front + 1
Step 4: print, No. Deleted is, item
Step 5: Print "Queue is Empty or
                  underflow".
```

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 Condition:

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

Insert an Element in Circular queue ⇒

```
Algo ⇒   QINSERT (QUEUE[MAXSIZE], Item)

   Step 1 ⇒  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]
   Step 2 ⇒ Exit
```

Queue (Data Structure)

operation on Queue

10, 20, 30, 40

Ex ⇒ maxSize = 3

1) front = -1 ✓ Empty queue
   Rear = -1

2) 3 to 5 Step Repeat
   R < maxSize - 1
   -1 < 3 - 1
   -1 < 2 true
   3 4 5

3) Read item
   Read 10

4) f = = -1
   -1 = = -1 true
   f = 0
   R = 0

5) set q[0] = item
   q[0] = 10

10		
q[0]	q[1]	q[2]

queue

10		
q[0]	q[1]	q[2]

f=0 R=0

Rear < maxSize-1
0 < 3-1
0 < 2 true

Read 20

4) if f==-1
0==-1 false

Else
R=R+1
R=0+1.
R=1

5) q[1] = 20

10	20	
q[0]	q[1]	q[2]

f=0 R=1

2) Rear < maxSize-1
1 < 3-1
1 < 2 true

Read 30
if f==-1
0 == -1 false

Else

R = R+1
R = 1+1 = 2

5) set q[2] = 30

10	20	30
q[0]	q[1]	q[2]
f=0		R=2

Last Rear < maxSize-1
2 < 3-1
2 < 2 false

6) queue is overflow

DELETE an Element in Circular queue ⇒

Algo → QDELETE(Queue[maxSize], Item)

```
1) if (front = -1)
       write queue underflow and Exit.
   Else:  item = Queue[front]
       if (front == Rear)
       Set front = -1
       Set Rear = -1
   Else:  front = ((front+1) %. maxSize)
       [ End if Statement ]
       → item deleted.
2. Exit.
```

QUEUE (Data Structure)

Delete operation on queue

Ex ⇒ maxsize = 3

10	20	30
q[0]	q[1]	q[2]

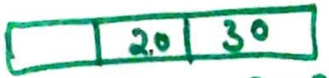
F=0 R=2

Case 1 ⇒

```
1)  f >= 0
    0 >= 0 true
2)  set item = q[0]
        item = 10
3)  f == R
    0 == 2 false
  Else
    f = f+1
    f = 0+1 = 1
4)  item is deleted
    10 is deleted
```

	20	30
q[0]	q[1]	q[2]

F=1 R=2



F = 1   R = 2

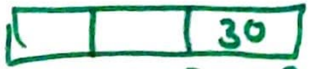
Case 2. 1) f >= 0
1 >= 0 true

2) item = q[1]
item = 20

3) if f == R
1 == 2 false

Else
f = f + 1
f = 1+1 = 2

4) item is deleted
20 is deeleted



F = 2 R = 2

Case 3 1) f >= 0
2 >= 0 true

2) item = q[2]
item = 30

3) if f == R
2 == 2 true
set f = -1
R = -1

4) item is deleted

f = -1
R = -1

Case 4. f >= 0
-1 >= 0 false

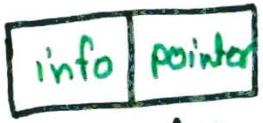
Step 5: queue is Empty
queue is <u>Underflow</u>.

Linked Lists

- A Linked List is a Linear data structure, in which the Elements are not stored at Contiguous memory Location.

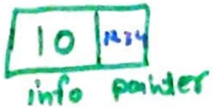
- A Linked List is a dynamic data structure. The No. of nodes in a List is not fixed and can grow and shrink on demand.

- Each Element is called a node, which has two parts. **info** part which stores the information and **Pointer** which point to the next Element.

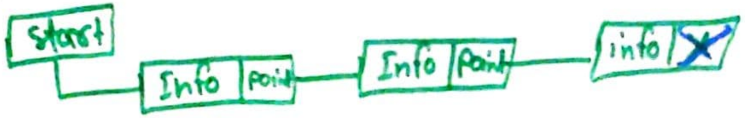


Node

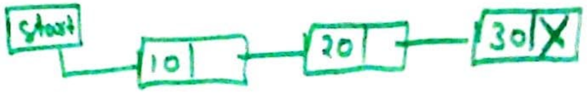
Ex ⇒



Ex ⇒



Ex ⇒



Advantages of Linked Lists

1) Linked Lists are dynamic data structure : That i's, they can grow and shrink during the Execution of a program.

2) Efficient memory Utilization : Here, memory i's not pre-alloCated. memory is allocated whenever it's required. And it's deallocated (Removed) when it's notonger needed.

3) Insertion and deletions are Easier and efficient : It provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

4) many Complex Applications Can be easily Carried out with linked Lists.

Operation ON Linked List:

The Basic operation to be performed on the Linked Lists are :-

1) Creation :- This operation are used to Create a Linked List. In this node is Created and Linked to the Another node.

2) Insertion :- this operation is used to insert a new node in the Linked List. A new node may be inserted

→ At the beginning of a Linked List.

→ At the End of a Linked List.

→ At the Specified position in a Linked List.

3) Deletion :- This operation is used to delete an item (a node) from the Linked List. A node may be deleted from.

→ Beginning of a Linked List

→ End of a Linked List

→ Specified position in the List.

4) Traversing ÷ It's a process of going through all the nodes of a Linked List from one End to the other End.

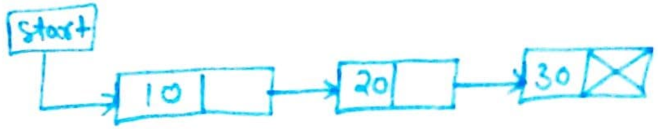
5) Concatenation ÷ It's the process of joining the second List to the End of the first list.

6) Display ÷ This operation is used to print Each and Every node's information.

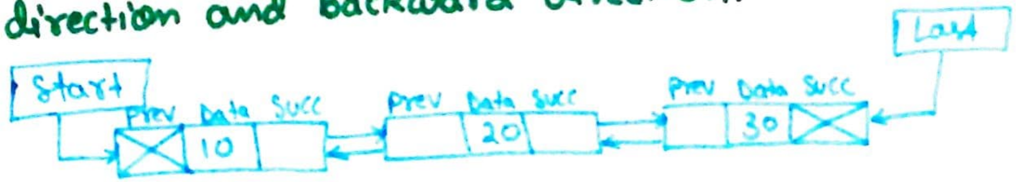
Types of Linked List

- Basically, there are four types of Linked List.

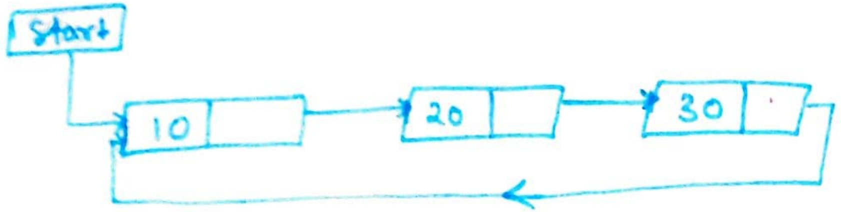
1 ⇒ Singly-Linked List ⇒ It's one in which all nodes are Linked together in Some sequential manner. It's also Called Linear Linked List.



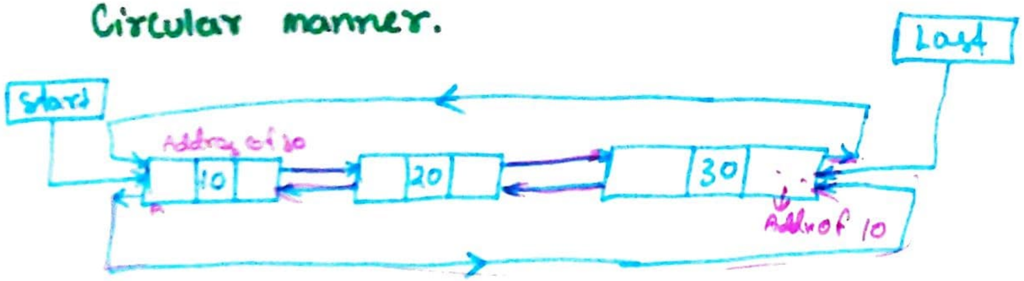
2 ⇒ doubly-Linked List ⇒ it's one in which all nodes are linked together by multiple Links which help in Accessing both the successor node (Next node) and predecessor node (previous node) within the List. This helps to traverse the List in the forward direction and backward direction.



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 Sorting 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.



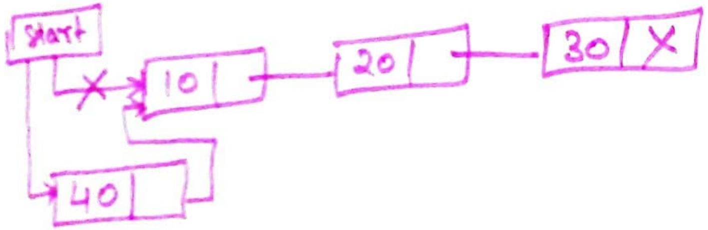
Inserting Nodes in Linked List

1) Inserting at the beginning of the list.

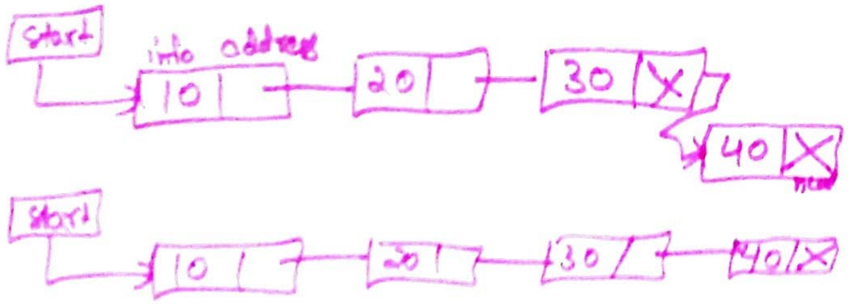
2) Inserting at the End of the List

3) Inserting at the Specified position Within the List.

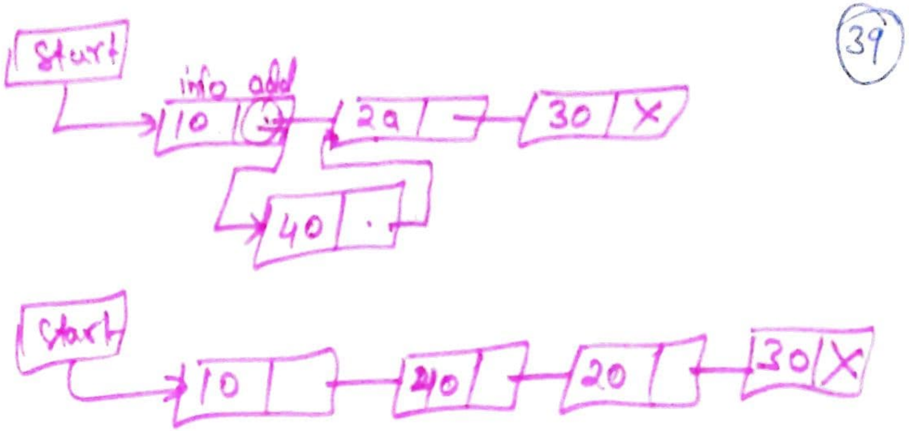
1)



2)



3)

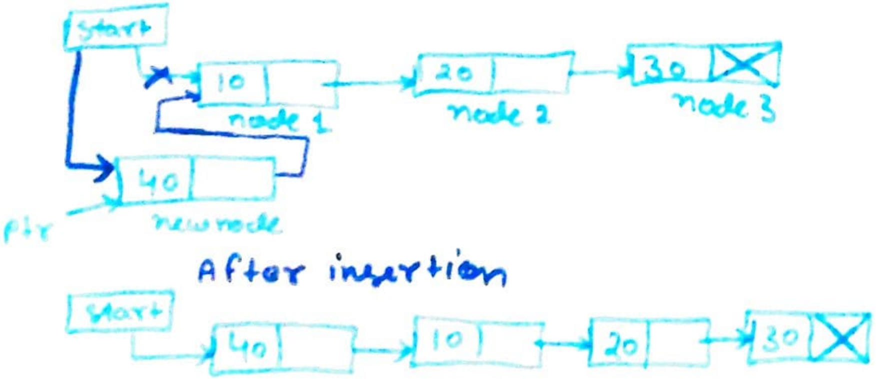


LINKED LIST

Inserting A Node AT the Beginning in Linked List

Algorithm ⇒

```
INSERT_FIRST(START, ITEM)

Step1: [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.

Step2:  Set PTR → INFO = Item

Step3:  Set PTR → Next = START

Step4:  Set START = PTR
```

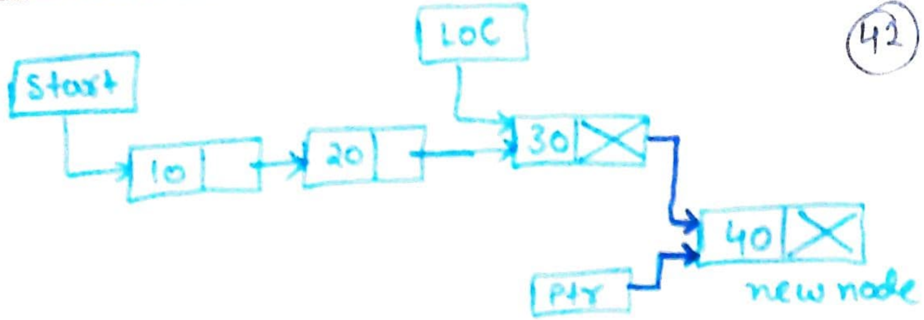


LINKED LIST

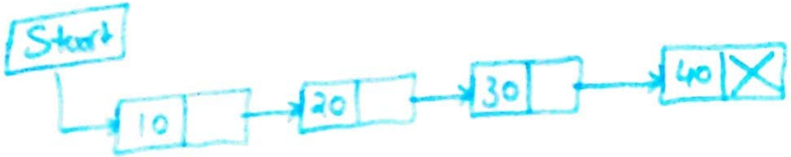
Insert A Node AT The End in Singly Linked List

Algorithm ⇒

```
Insert_Last(START, ITEM)

Step1:  Check for overflow
          IF ptr = NULL then
            print overflow
          Exit
        Else
          PTR = (Node*) malloc (size of (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 != NU
Step7: Set Loc = Loc → Next ,
Step8: Set Loc → Next = ptr ,
```



After Insertion



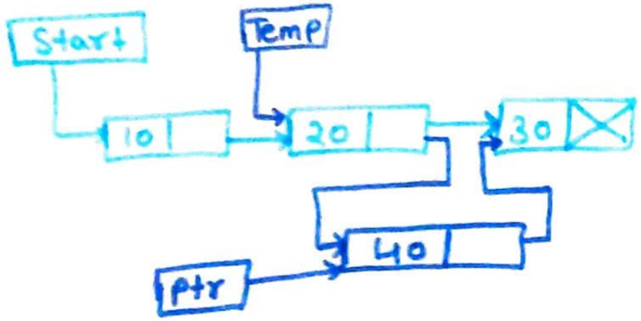
LINKED LIST

Inserting a Node at the Specified Position in Singly Linked List.

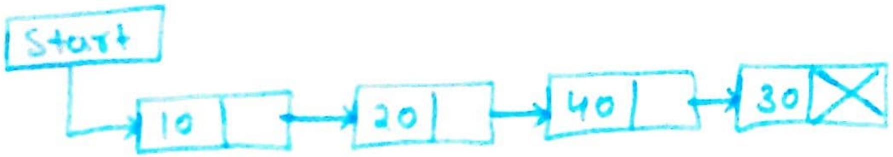
Algorithm ⇒

```
Insert_Location (START, Item, LOC)

Step1:  Check for overflow
          If ptr == NULL then
             print overflow
          Exit
        Else
        ptr = (Node *) malloc (size of(Node))

Step2:  set ptr → INfo = item

Step3:  IF start = NULL then
            set Start = ptr
          set ptr → Next = NULL

Step4:  Initialize the Counter I and Pointers
        set I = 0
        Set temp = Start
```

```
Step5: Repeat Steps 6 and 7 until I< Loc
Step6:   set temp = temp → Next
Step7:   set I = I+1
Step8:   Set ptr → Next = temp → Next
Step9:   set temp → Next = ptr.
```



After Insertion



Deleting Node in Linked List

- Deleting a node from the Linked List has three instances.

1 → Deleting the first node of the Linked List.

2 ⇛ Deleting the Last node of the Linked List.

3 ⇛ Deleting the node from Specified position of the Linked List.