Unit-1

Data: value or set of values.

Exp: Student (Entity)

Name	Roll No.	Age	Marks%	DOB
X	1	20	75	2/9/2000
Y	2	22	63	7/5/1999
Z	3	18	48	6/3/200
⋮	⋮	⋮	⋮	⋮

Attributes: Properties of Entity.

Exp (Name, Roll No, Age, Marks, DOB etc)

Record: Each Row is called record.

Field: Each column is called Field.

Information: Meaningfull or Processed data.

Data Structure: - DS is a format for storing & organizing data.

- The logical and mathmatical model of particular organization to handle data is called DS.

- DS is a way of organizing all data items that consider not only the stored element but also their relationship to each other.

Data Structure specifies following 4 things:
(i) Organization of data (ii) Accessing methods
(iii) Degree of associativity (iv) Processing alternatives for Information

Classification of data structure:

① According to nature of size
(a) Static DS (b) Dynamic DS

Static DS: A data structure is said to static if we can store data upto a fix number. Memory allocate at compile time.
Exp: array.

Dynamic DS: A Dynamic DS is that data structure which allows the programmer to change its size during program Execution.
Exp: Linked list [memory allocate Run time]

② According to occurrence:
(1) Linear DS (2) Non Linear DS

Linear DS: In linear DS data is stored in consecutive memory location or data is stored in sequential form i.e. every element in the structure has a unique predecessor and unique successor.
Exp: Stack, Queue, Array, Linked list.

Non linear DS: Not sequential form not consecutive memory location.
Exp: tree, Graph.

Algo:-

- An algo is well defined list of steps for solving a particular problem.
- An algo is a set of rules for carrying out calculation either by hand or on a machine.

Every algo must satisfy the following criteria -

① Input: There are zero or more quantities which are externally supplied.

② Output: At least one quantity is produced.

③ Definiteness: Each instruction must be clear & unambiguous.

④ Finiteness: Algo will terminate after a finite number of steps.

⑤ Effectiveness: It is not enough that each operation be definite but it must also be feasible.

Desirable attributes for algorithms:

① Generality: An algo should solve a class of problem, not just one problem.

Eg:- an algo to calculate the average of four values is not as generally useful as that

That calculate the average of an arbitrary number of values.

2) Good Structure:-

- Explain
- Understand
- Test
- Modify it

3) Efficiency:- An algo's speed of operation is often an important property, as is its size or compactness.

4) Ease of use:- This property describes the convenience & ease with which users can apply the algo. to their data. Some times what makes an algo easy to understand for the user also make it difficult to design for the designer (and vice-versa)

5) Robustness:- can handle abnormal condition. (Invalid data)

6) Economy:-

Algo Design Technique:-

1) Greedy Approach: This algo works in steps. On each each step it selects the best available option until all options finish. Eg: shortest path algo

(2) Divide and conquer Approach:

- Divide the original problem into a set of sub problems. (similar type)
- Solve every subproblem individually, recursively.
- combine the sol^{n} of the sub-problems into a sol^{n} of the whole original problem.

(3) Modular Programming:

We devide the big problem into smaller ones which are totally different from eachother. Then we combine the sol^{n} of all smaller problems and we get the sol^{n} of big problem. Here we can use different algo design for all small problems. This approach gives different modules for different problems.

(4) Backtracking algo:- Backtracking algo try each possibility until they find the right one. It is a depth first search of the set of possible sol^{n}. During the search if an alternative does not work, the search backtracks to the choice point and tries

the next alternatives. When the alternatives are exhausted, the search returns the previous choice point & try the next alternative there. If there are no more choice points, the search fails.

(5) Top down approach :-

Prog^r should be devided into a main module & its related module. Each module ~~should~~ else should be divided into sub modules according to sw Engg & programming style. This division of modules processes until the module can't be further subdevided.

In C the idea of top down is done using function. coding prog^r stub

(6) Bottom up approach :-

It is opposite of top down. drivers

In this starting the design with specific modules and builds them into more complex structure ending at the top. The bottom up approach is widely used for testing, because each of the lowest level functions is written & tested first. Once lowest level functions have been tested & verified to be correct.

the next level of functions may be tested. Since the lowest level functions already have been tested, any detected errors are probably due to higher level of function. This process continues, moving up the levels, untill finally the main function is tested.

(1) Greedy Approach: This approach works in steps. In each steps it selects the best available option untill all options finish.

Eg: Shortest path algo

Overview of various Data Structure:

(1) Array
(2) Linked list
(3) Queue
(4) Stack
(5) Graph
(6) Tree

Different Areas where D.S. are used:

1. Compiler Design
2. Operating System
3. DBMS
4. Artificial Intelligence
5. Graphics

Data Structure operations:

1. Create
2. Destroy
3. Insertion
4. Deletion
5. Updation
6. Searching
7. Sorting
8. Traversing
9. ~~Destroying~~ Merging

Dynamic memory allocation:

```
int *p;
p = (int *) malloc (n * sizeof (int));
or
p = (int *) calloc (n, sizeof (int));
free (p);
```

Efficiency of algo:

The efficiency of an algo can be checked by -

- correctness
- implementation of an algo
- simplicity of an algo
- Execution time & memory requirement of an algo

Read step by step instructions of algo for logical correctness & testing it on some data using mathematical technique to prove it correct.
Analyse the simplicity of an algo & Design the algo in a simple way so that it becomes easier to be implemented.

Time & space complexity:

Space complexity :- space complexity of an algo is the amount of memory it needs to run to complete.
The space needed by a program consists of following components -

- Instruction space
- Data Space
- Environment stack space : This space is needed to store the information to resume the suspended functions.

• Recursion stack space: space needed by recursive functions.

Space complexity S(n): An algo A for a problem P is said to have space complexity of S(n) if the no. of bits required to complete its run for an input of size n is always less than equal to S(n).

Time complexity:

The time complexity of an algo is the amount of time it needs to run to complete. To measure the time complexity accurately, we have to count all sorts of operations performed in an algo. If we know the time for each one of the primitive operation performed in a given computer, we can easily compute the time taken by an algo to complete its execution. Time complexity also depends on CPU speed & HW.

Time complexity T(n): An algo A for a problem P is said to have time complexity of T(n) if the no. of steps required to complete its run for an input of size n is always less than equal to T(n).

Time-space trade-off:

Time-space or time-memory trade-off is a way of solving a problem in less time by using more storage space or vice-versa. we implement different data structure, some data structure takes more space but improves the run time and some takes less space but effects run time of algo.

we can take a simple case of hash table & linked list. Hash table takes more space but searching is fast, Linked list takes less space than hash table but searches sequentially and slow.

If you want to store uncompressed data then it will take less time but more space. or if you store compressed data then it will take less space but more time due to time taken by compressed algo.

There are three cases of complexity of an algo:

(a) Worst case: The worst case of an algo is the function which defines the max. number of steps taken for a data of size n.

(b) Best case:- The best case of an algo is the function which defines the ~~average~~ min. number of steps taken for a data of size n.

(c) Average case:- The average case of an algo is the function which defines the average number of steps taken for a data of size n.

Exp:- for linear search.

Worst case : item at last
Best case : ——— first
Average cases ——— in middle

Abstract Data type:- (ADT)

- ADT is a set of operation which is used with the component of ~~the element~~ of that abstract data type.

The data type where we are not concerned about how the accessing will be implemented is refered as ADT. Also we are not concered ~~about~~ with space & time efficiency.

Exp:-(1) List ADT.

component-
Item
operation-
Insertion
Deletion
Search
Display

Here Item is component of ADT-List can contain number of items.
operation on these items can be insertion, deletion, search, display.

Exp-2:
·STACK ADT-
component-

Item
operation
Push (insert item into stack from top)
Pop (Delete item from top)

We not concern how will stack implemented using array or linked-list.
Definition of ADT have Two Parts
(1) Description of the way in which components are related to each other.
(2) Statements of operation that can be performed on the data type.

Growth of function: How the complexity varies for different input size & for different patterns in input.
For standardizing these measures Asymptotic analysis is performed & after performing asymptotic analysis complexity is expressed in term of Asymptotic Notation.

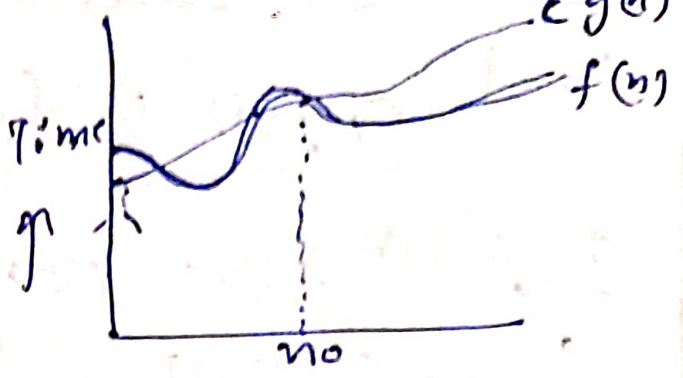
Growth:- How the computation time of problem grows as the input grows gradually.
It can be linear growth. $y = ax + b$ [Power atmost 1]
It can be polynomial growth. $y = ax^n + bx^{n-1} + \dots$ n is non negative integer
It can be exponential growth. a^x where $a > 1$ constant

Asymptotic Analysis: performed on asymptotic curves.
Asymptotic means a line that tends to converge to a curve, for a function $f(n)$ as the value of n grows.

Asymptotic Notation: Tells about the worst case and best case i.e. upper bound and lower bound of the algorithm.

Big-oh Notation :- This notation gives upper bound for a function ~~with this constant factor.~~

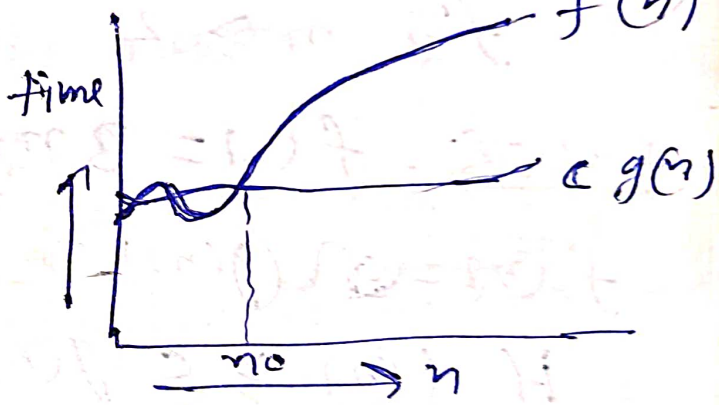
~~$f(n) = O(g(n))$~~ For a given function $g(n)$, we denote it by $O(g(n))$.



$O(g(n)) = \{ f(n) :$ There exist positive constants c & n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0, c > 0 \}$

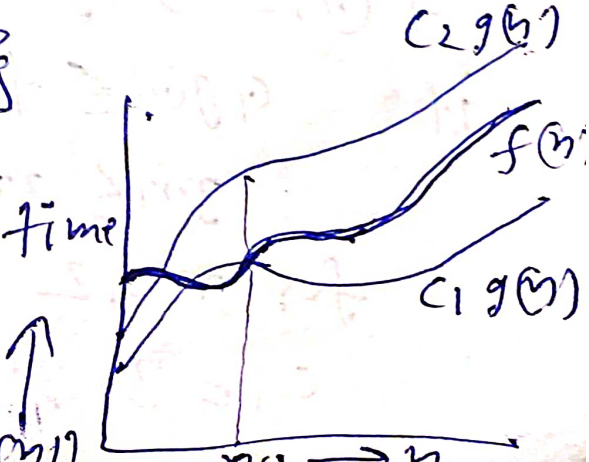
Ω- Notation :-

This notation gives lower bound of function. For a given function $g(n)$, we denote it by $\Omega(g(n))$.



$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c & n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0, c > 0 \}$

Θ- notation :- It provides the lower and upper bound of the function. For a given function $g(n)$, we denote it by $\Theta(g(n))$.



$\Theta(g(n)) = \{ f(n)$: There exist positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Exp:- ① $f(n) = (3n+2)$

$f(n) = O(g(n))$.

if $f(n) \leq c\, g(n)$

$3n+2 \leq 4n$

$\leq 3n+n$ $\quad c = 4$

for $n \geq 2$

Exp:- ② $f(n) = 3n+2$

$f(n) = \Omega(g(n))$

if $f(n) \geq c\, g(n)$

$3n+2 \geq 3n$ $\quad c \geq 3$

for $n \geq 1$

Exp:- ③ $f(n) = 3n+2$

$f(n) = \Theta(g(n))$

if $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$3n \leq 3n+2 \leq 4n$

for $n \geq 2$

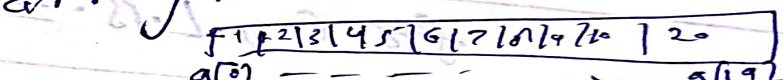
$c_1 = 3, \quad c_2 = 4$

Array

Array: an array is finite, orderd and collection of homogeneous data elements.
Array is finite because it contains only limited no. of elements and orderd because the elements are stored in contiguous memory location in a linear order

One-Dimensional array:-

Array will be declare as
a[20], where a is the name of array & 20 is size of array.



No of element in array = upper bound - lower bound + 1

$n = UB - LB + 1$

$n = 19 - 0 + 1$
$= 20$

The elements of array are referenced respectively by an index or subscript.

Exp: a[0], a[1], a[2], — — — a[n]

where a is name of array & 0, 1, 2, — [n] are index or subscript.

a[n] is called subscriptial variable.

Representation of single dimensional array in memory :-

The address of the first element of array is called the base address & denoted by Base [A].

Using the Base address, we can find out the address of the k^{th} element with the help of following formula —

$$Loc(A[k]) = Base(A) + w * (k - L.B.)$$

for C, $L.B. = 0$

$$Loc(A[k]) = Base(A) + w * k$$

(w = word size)

Exp:- Consider an array of A is declared as array of int with size 50 & first element is stored at memory address 101. So find the address of 5^{th} element. Given $w = 4$.

Sol^n:- $Loc(A[k]) = Base(A) + w * k$

$Loc(A[5]) = 101 + 4 * 5$

$= 121$

Q2:- Conside the linear array
A(5:50), B(-5:10) and C(18)
(i) Find the number of elements in each array.
(ii) Suppose Base(A) = 300 & w = 4. find the address of A[15] & A[40], a[55].

Sol:- (i)

length of A = UB - LB + 1
= 50 - 5 + 1
L(A) = 46

L(B) = 10 - (-5) + 1
L(B) = 16

L(C) = 18 - 1 + 1
= 18

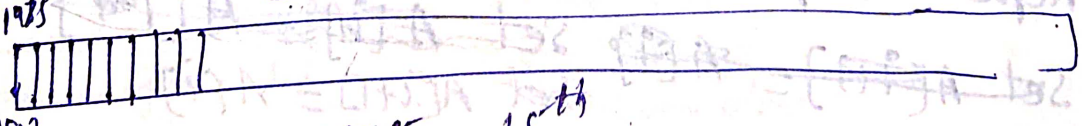
LB = 1

A(15) = 300 + 4(15-5)
= 340

A(40) = 300 + 4(40-5)
= 440

A(55) = 300 + 4(55-5)
= 500

Q3:- Consider an array A, which records the no of T.V. sold each year from 1975 to 2000. Suppose the base address 101 and w = 4 then find the log A[1990]. Base(A) = 101



k = 1990 - 1975 = 15th

log A[15] = Base(A) + w * k
= 101 + 4 × 15
= 161
Ans

Algo:-

① Traversing algo

Let A be an array of n elements & k is used as subscript

Traverse (A, n)

1. Set k = 0
2. Repeat steps 3 & 4 while k <= n-1
3. Display A[k]
4. k = k+1
5. Exit.

② Insertion algo :-

Let A be an array of size N which contains n elements & k is location where we have to insert an element ITEM

INSERT (A, N, n, k, ITEM)

1. if (n = N-1) then Display "overflow" & exit.
2. set i = n+1
3. Repeat steps 4 & 5 while i >= k
4. Set A[i+1] = A[i]
5. set i = i-1
6. Set A[k] = item
7. Reset n = n+1
8. Exit

Deletion from array:

DELETE (A, N, n, k, ITEM)

(i) if (n = 0) then Display "underflow" and exit.

(1) Set ITEM = A[k]

(2) Repeat set i = k

(3) Repeat steps 4 & 5 while i < n-1

(4) set A[i] = A[i+1]

(5) i = i+1

(6) Reset n = n-1

(7) Display new array

(8) Exit

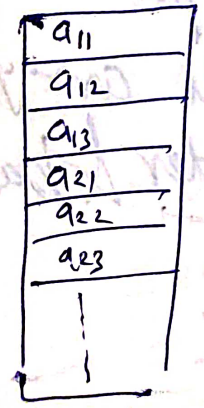
Representation of two-Dimensional array in memory:-

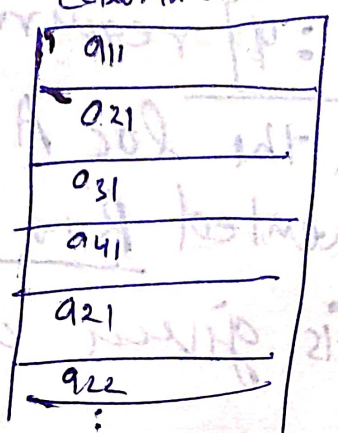
R \ C	1	2	3
1	200 a_{11}	202 a_{12}	204 a_{13}
2	206 a_{21}	208 a_{22}	210 a_{23}
3	212 a_{31}	214 a_{32}	216 a_{33}
4	218 a_{41}	220 a_{42}	222 a_{43}

Row major order

R \ C	1	2	3
1	200 a_{11}	208 a_{12}	216 a_{13}
2	202 a_{21}	210 a_{22}	218 a_{23}
3	204 a_{31}	212 a_{32}	220 a_{33}
4	206 a_{41}	214 a_{42}	222 a_{43}

Column order major

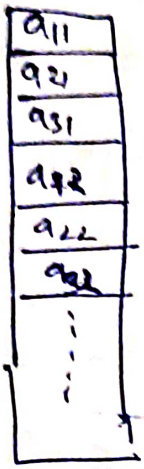




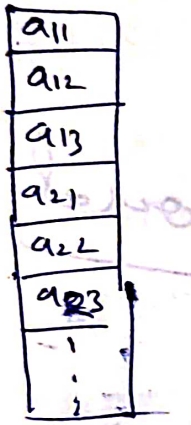
Column major order :- order of array = M×N

Sol:

$$Loc(A[J][K]) = \text{Base Address} + w\left[(J-L_1) + M(K-L_2)\right]$$



Row Major order :- order of array = M×N



$$Loc(A[J][K]) = \text{Base Address} + w\left[N(J-L_1) + (K-L_2)\right]$$

w = word size
L_1 = Lower Bound of Row
L_2 = ——————— column

Ex:- A two dimensional array is defined as A [3:7, -1:4] requires 4 words per memory cell. Find the loc A [5,2] if the array is implemented Row major order & base Address is given as 200.

Sol:- $L_1 = 3$, $L_2 = -1$

$M = 4$, $N = 6$ & $W = 4$, $I = 5$, $K = 2$

Base Address = 200

Row-major order

$$Loc(A[5,2]) = Base\ Add + W(N(I - L_1) + (K - L_2))$$

$$= 200 + 4[6(5-3) + (2 - (-1))]$$

$$= 200 + 4[12 + 3]$$

$$= 200 + 60$$

$$= 260$$

Exp:-2:- $A[20][5]$, BA = 200 & $W = 4$

Find out the address of $A[12,3]$ using column major order in C language.

Ans:- $M = 20$, $N = 5$

$$Loc(A[I,K]) = Base\ Add + W[(I - L_1) + M(K - L_2)]$$

$L_1 = L_2 = 0$ becaus C.

$$Loc(A[12,3]) = 200 + 4[12 + 20(3 - 0)]$$

$$= 200 + 4 \times 72$$

$$= 200 + 288$$

$$= 488$$ Ans

Exp:-3:- Array A[20][50] required 4 bytes of storage for each element. BA = 2000. Find loc of A[10][10] when the array is stored

(i) Row-major
(ii) Column-major

(i) Row major

$$A[10][10] = \text{Base Add} + w[N(J-1) + (K-1)]$$
$$= 2000 + 4[50 \times (10-1) + (10-0)]$$
$$= 2000 + 4[500 + 10]$$
$$= 2000 + 2040$$
$$= 4040$$

(ii) Column order

$$A[10,10] = \text{Base Add} + w[M(J-1) + (K-1)]$$
$$= 2000 + 4[10 + 20(10-0)]$$
$$= 2000 + 840$$
$$= 2840$$

Ans

Three Dimensional Array:

Effective Index E_i for given subscript k_i can be calculated by formula

$$E_i = k_i - \text{lower bound}$$

Row-major Order:-

$$Loc(A[i][j][k]) = BaseAdd + w[(E_1 L_2 + E_2) L_3 + E_3]$$

Column-major order:

$$Loc(A[i][j][k]) = BaseAdd + w[(E_3 L_2 + E_2) L_1 + E_1]$$

Where L_1, L_2 & L_3 are lengths of the three dimensions of array & E_1, E_2, E_3 are Effective index of respective subscript.

Exp:- Suppose A is three dimensional array given as $A(1:9, -4:1, 5:10)$. Suppose the array A stores in memory in row-major order & base address of $A = 400$ & $w = 2$. Then find out add of $A[5, -1, 8]$.

Ans:- $L_1 = 9-1+1 = 9$, $L_2 = 1-(-4)+1 = 6$, $L_3 = 10-5+1 = 6$

$E_1 = 5-1 = 4$, $E_2 = -1-(-4) = 3$, $E_3 = 8-5 = 3$

$Loc(A[5, -1, 8]) = 400 + 2[(4 \times 6 + 3)6 + 3]$

$= 400 + 2[162 + 3]$

$= 830$

Application of Array:

1. Array Addition
2. ——— Subtraction
3. ——— Multiplication
4. Transpose of an array
5. Addition of rows & columns
6. To check whether the given matrix is sparse or not.
7. To maintain polynomial in memory.

Sparse Matrix:-

many elements are zero.

Exp:-

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 4 \\ 1 & 0 & 0 \end{bmatrix}$$

Representation of sparse matrix in memory:

(a) Array Representation:

In the array representation of a sparse matrix only the non-zero elements are stored so that space can be reduced. Each non zero element in the sparse matrix is represented as (Row, Column, value).

for this, a Two-Dimensional array containing 3 columns can be used. The first column for row number, second column for column no. & third column for value of non-zero elements.

Exp:-

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 4 & 3 & 0 \\ 0 & 0 & 6 & 0 \end{bmatrix} \Rightarrow$$

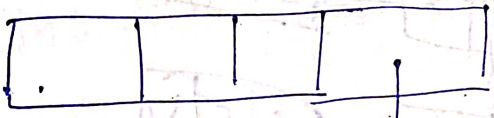
IN-C

Row	colum	value
0	0	2
1	1	1
2	1	4
2	2	3
3	2	6

Linked Representation of a Sparse matrix:-

For linked list representation we use three type of node

(1) Information node: which contains four thing (i) no of rows (ii) no. of columns



(iii) no of non zero elements (iv) Address of first Head node

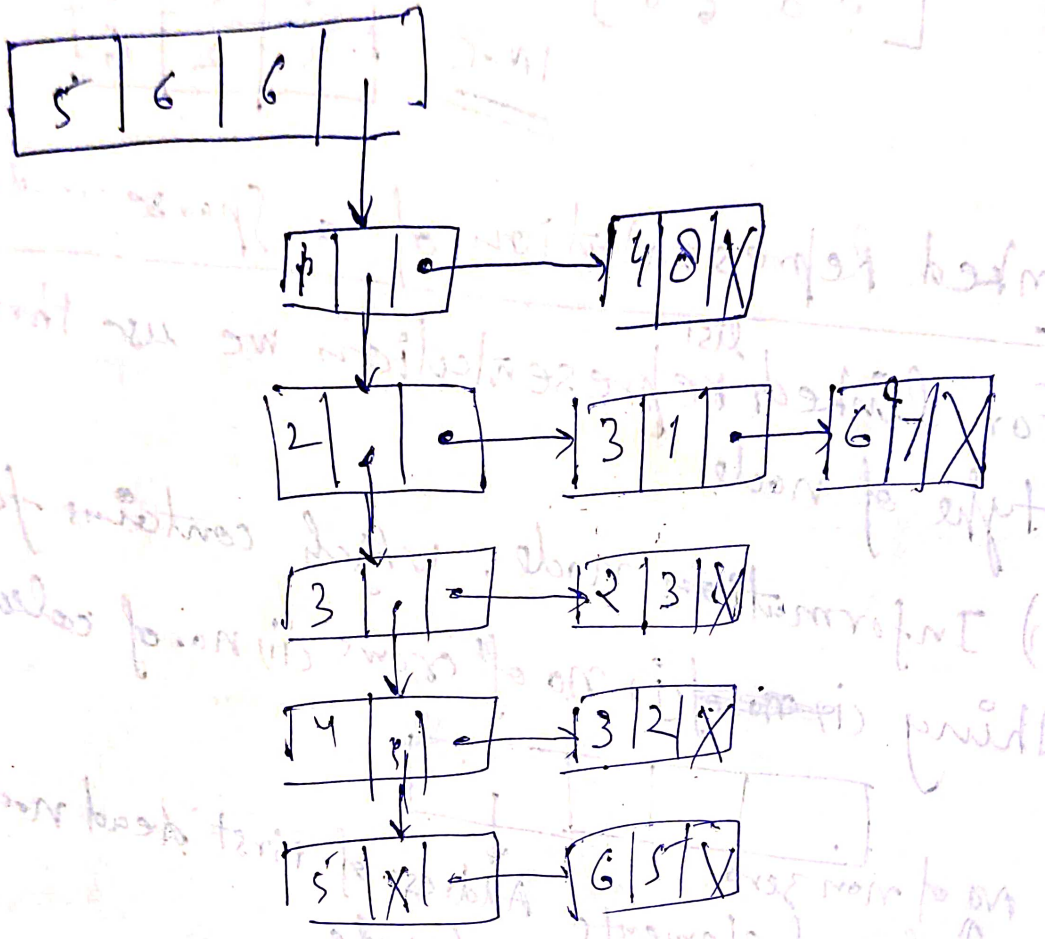
(iii) ~~value of~~ elements
(iv) Address of first Head node

(2) Head node: which contains Three field (i) Row number (ii) Pointed to next Head node (iii) Pointer to first non zero element of that row

(3) Node:- which contains Three field (i) contains column no (ii) contains value (iii) Pointer to next non zero element of that row.

Ex:-

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 7 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix}_{5 \times 6}$$



Traversing Two Dimensional Array:-

Colum major order Algo:-

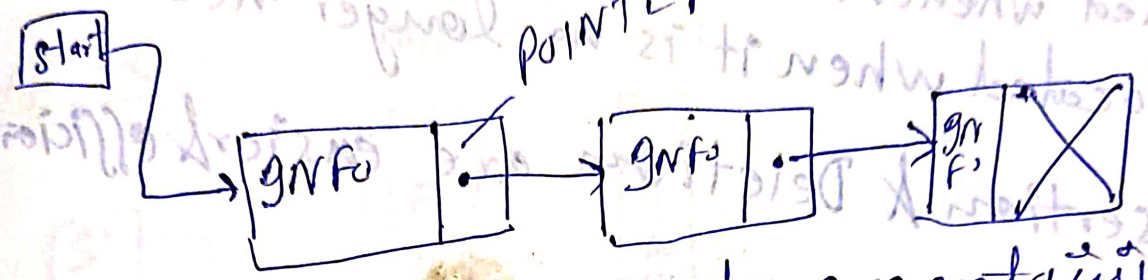
1. For I=0 to column-1
2. For I=0 to Row-1
3. Display A[I][J]
4. Stop

Row major order Alg:-

1. for I=0 to Row-1
2. for J=0 to column-1
3. Display A[I][J]
4. Stop

Linked List:-

- Linked lists are special lists of some data elements linked to one another.
- Each element is called a node which has two parts. (i) INFO part which stores the information and (ii) POINTER which points to the next elements.



These nodes are structure containing two fields
(i) Data field (ii) Link field

* The link field of last node contains NULL.

Linkedlist is linear or Non-linear:

- According to accessing the element linked list is linear.
- According to storage linked list is nonlinear.

Application of Linked list:

- Arithmetic operation can be easily performed.
- Manipulation & evaluation of polynomial is easy
- Useful in symbol table construction.

Advantages:

(i) Linked lists are dynamic data structure: They can grow or shrink during the execution of a program.

(ii) Efficient memory utilization: Here memory ~~are~~ is not pre-allocated. Memory is allocated whenever it is required & it is deallocated when it is no longer needed.

(iii) Insertion & Deletion are easier & efficient

Disadvantages:

(1) More memory: no. of field more so more memory required

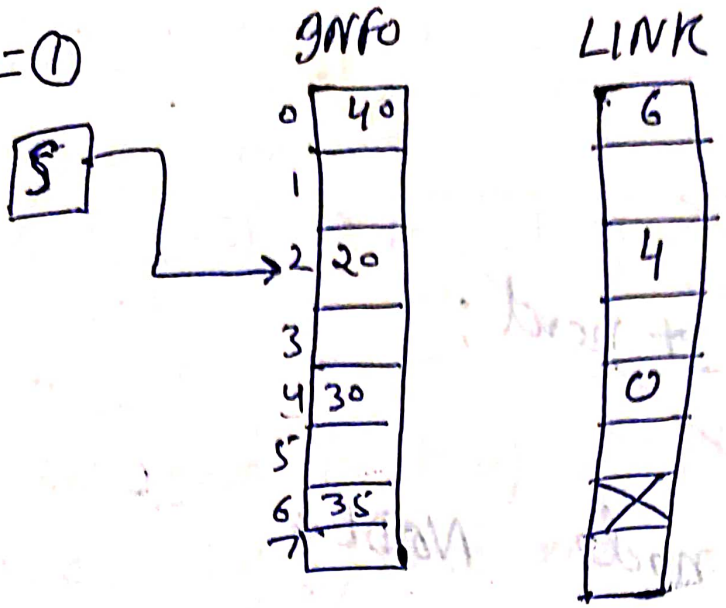
(2) Access to an item is time-consuming

Representation of linked list in memory: (using Array)

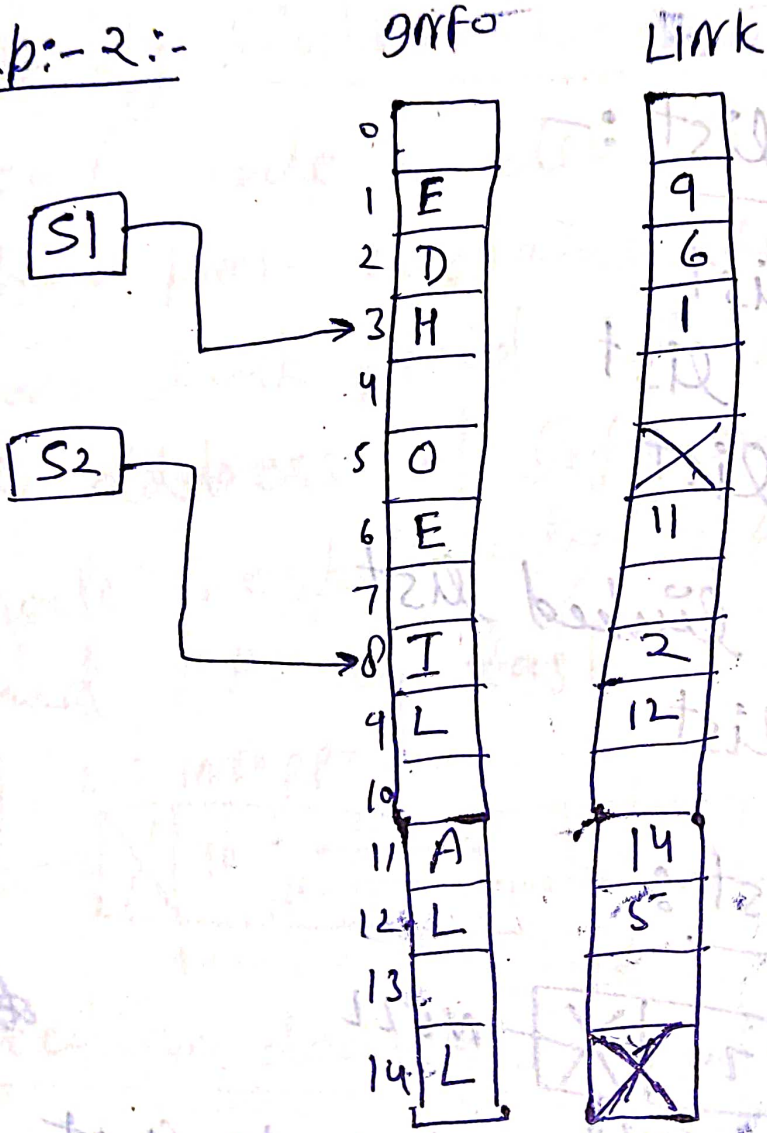
For the representation of linked list in memory, it requires two linear array called INFO and LINK which contains the information part and address of the next node in the linked list respectively. The linked list also have a pointer S which contains the address of the first node in the list.

Exp:

Exp:- ①


INFO
LINK
S
0
1
2
3
4
5
6
40
20
30
35
6
4
0

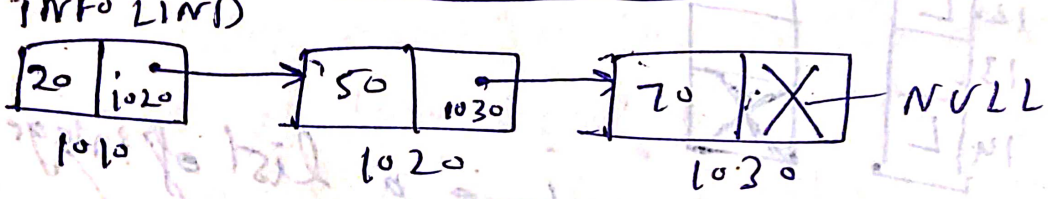
Exp:- 2:-


INFO
LINK
S1
S2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
E
D
H
O
E
I
L
A
L
L
9
6
1
11
2
12
14
5

Types of linked list :-

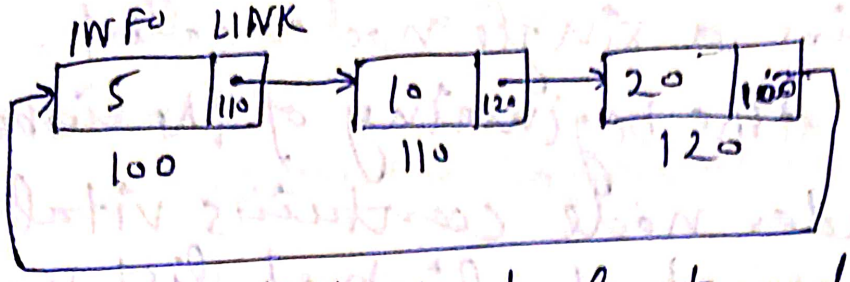
1. Linear linked list
2. circular linked list
3. Doubly linked list
4. Circular doubly linked list
5. Header linked list

Linear linked list :



Each node is devided in two parts, first parts contains the infor^n of element and the second parts contains the address of next node in list. Null Pointer represents that there are no more nodes in the list.

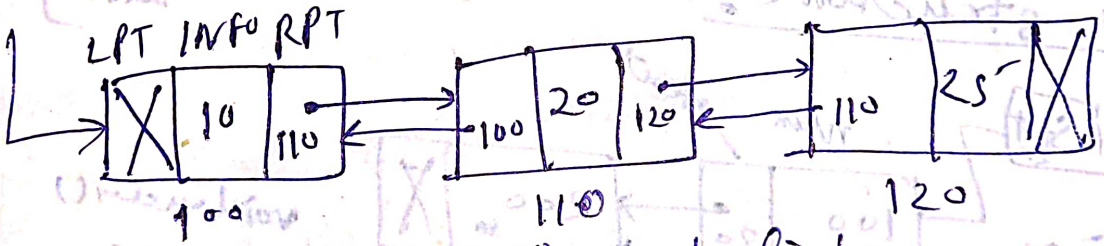
Circular linked list :-



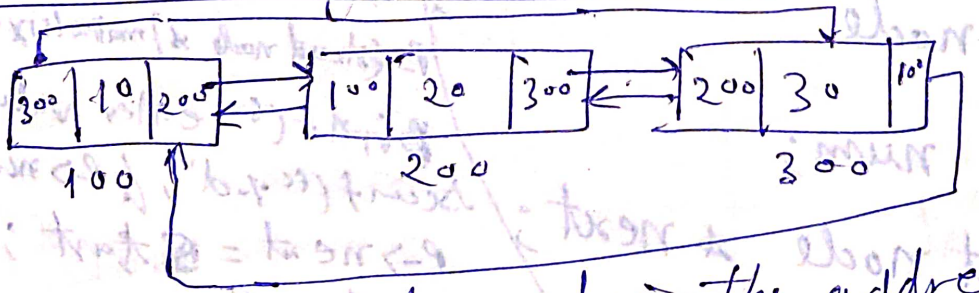
The second part of last node contains the address of first node.

Doubly linked list (Two-way list) :-

Each node is devided in three parts, Info parts contains the information and two link field LPT & RPT the contains the address of left node and Right node respectively. LPT of First node and RPT of last node is NULL.

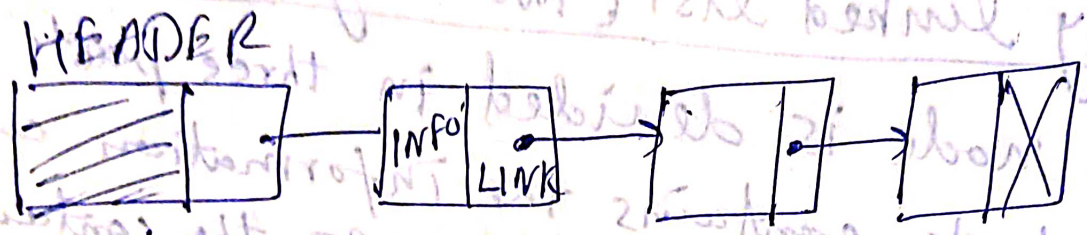


Circular doubly linked list:



LPT of first node contains the address of last node
RPT of last node contains the address of first node not Null.

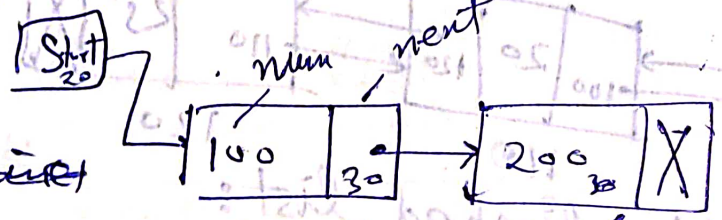
Header linked list: In header linked list always contains a single node, called header node, at the beginning of the linked list. The Header node contains vital information about the linked list such as number of nodes in the list, whether the list is sorted or not.



No data in this node

~~Operation in Single Linked list:-~~ Linear

~~Creat~~ Representation of linked list in memory using structure:- or [Create linear linked list]



~~Void main()~~

```
{
struct node
{ int num;
  struct node * next;
};
struct node *start=NULL;
```

```
void main()
{
struct node *p;
p=(struct node *)malloc(sizeof(struct node));
printf("enter value of node");
scanf("%d",&p->num);
p->next=start;
start=p;
}
```

Create linear linked list:-

```
void main()
{struct node *p,*q;
  clrscr();
p =(struct node*)malloc (sizeof(struct node));
q =(struct node *) malloc(sizeof(struct node));
printf("enter the value of node p");
scanf("%d", &p->num);
printf("enter the value of node q");
scanf("%d",&q->num);

start = p;
p->next = q;
q->next = NULL;
getch();
}
```

Operation in linear linked list:

1. Insertion
2. Deletion
3. Traversing

Traversing in linear linked list:

Algo:-

TRAVERSE (L, START, PTR, INFO, NEXT)

0. If START = NULL then list is empty & exit.
1. Set PTR = START
2. Repeat steps 3 & 4 while PTR ≠ NULL
3. Display PTR → INFO
4. Set PTR = PTR → NEXT
5. Exit

* Same for Doubly linked list

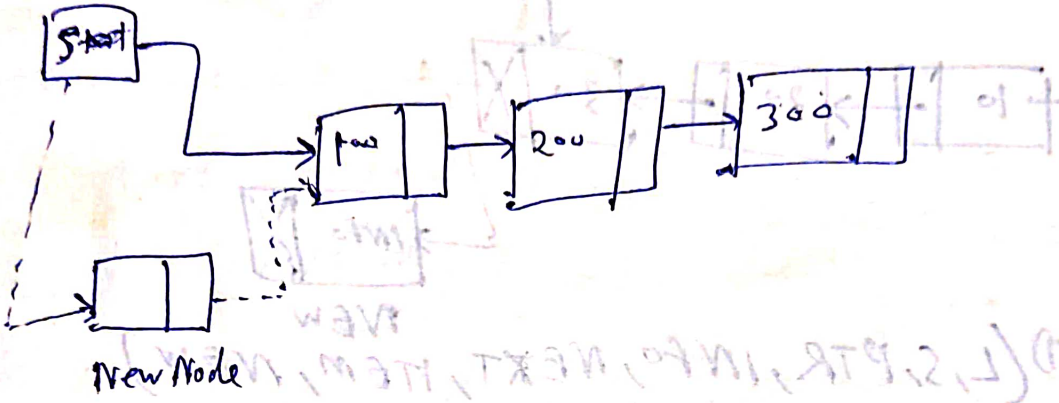
C- code:-

```
Void Traverse()
{
  struct node *Ptr;
  Ptr = start;
  if(start==NULL) printf("list is empty");
  while (Ptr != NULL)
  {
    printf("%d \t", Ptr->info);
    Ptr = Ptr->next;
  }
}
```

Insertion in linear linked list:

(i) Insertion at beginning
(ii) Insertion at last
(iii) Insertion at specific position

(i) Insertion at beginning:-



Algorithm:

INSERTBEG (L, S, INFO, NEXT, ITEM, NEW)

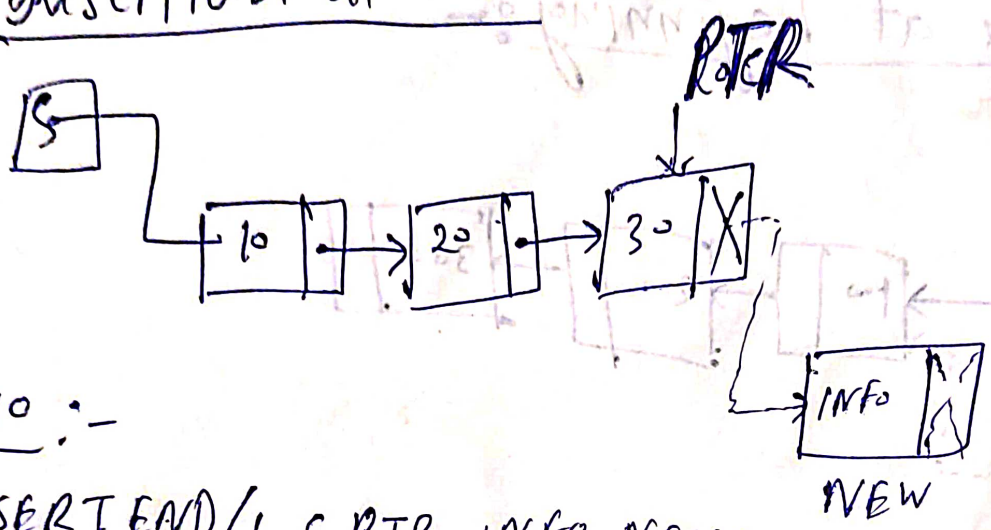
(1) Create a new node and name it NEW.
(2) If NEW = NULL then write Overflow & exit
(3) Set NEW -> INFO = ITEM
(4) Set NEW -> NEXT = S
(5) Set S = NEW
(6) Exit

C- coding-

```
void addbeg()
{
  struct node *NEW;
NEW=(struct node *) malloc(sizeof(struct node));
  printf("enter value of node");
```

```
scanf("%d", &NEW->info);
NEW->next = start;
start = NEW;
}
```

(ii) Insertion at last:



Algo :-

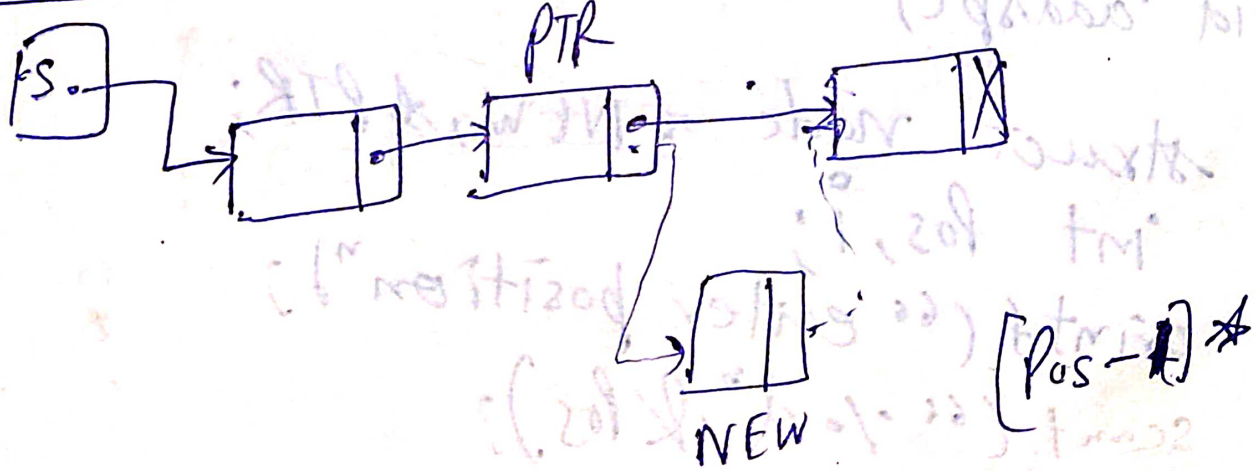
INSERTEND(L, S, PTR, INFO, NEXT, ITEM, NEW)

1. Create a new node & name it NEW
2. If NEW = NULL then write Overflow & exit.
3. Set NEW -> INFO = ITEM
4. Set NEW -> NEXT = NULL
5. If S = NULL then set S = NEW and Exit
6. SET PTR = S
7. Repeat steps 8 while PTR -> NEXT != NULL
8. PTR = PTR -> NEXT
9. PTR -> NEXT = NEW
10. Exit

c-coding :-

```
void addlast()
{ struct node *NEW, *PTR;
NEW=(struct node *) malloc (sizeof(struct node));
printf("enter value of node");
scanf("%d", &NEW->INFO);
NEW->NEXT = NULL;
if (start == NULL)
  start = NEW;
else
  { PTR = start;
    while (PTR->NEXT != NULL)
    { PTR = PTR->NEXT;
    }
  PTR->NEXT = NEW;
}
}
```

(iii) Insert at specific position :-



Algo:-

INSERT SPC (L, S, PTR, INFO, NEXT, ITME; NEW, POS)

(1) Create a new node and name it NEW.
(2) if NEW = NULL then write Overflow & exit.
(3) set NEW -> INFO = ITEM
(4) if POS = 1 then
    NEW -> NEXT = S
    S = NEW and exit
(5) set I = 1 and PTR = S
(6) Repeat step 7 & 8 untill I < (POS-1)
(7) set PTR = PTR -> NEXT & if PTR = NULL. Display list is small & exit
(8) set I = I + 1
(9) set NEW -> NEXT = PTR -> NEXT.
(10) PTR -> NEXT = NEW
(11) Exit.

C - coding:-

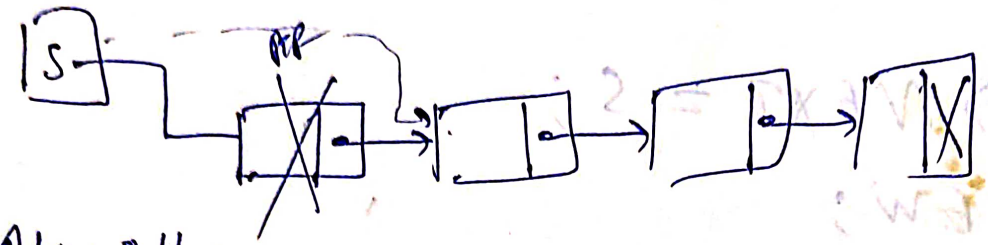
```
void addsp()
{
    struct node *NEW, *PTR;
    int Pos, i;
    printf("enter position");
    scanf("%d", &Pos);
```

```
NEW=(struct node *)malloc(sizeof(struct node));
printf("enter value of node");
scanf("%d", &NEW->INFO);
if(Pos==1)
{
   NEW->NEXT = S;
   S = NEW;
}
else
{ PTR = S;
   for(i=1; i<(Pos-1); i++)
   { PTR = PTR->NEXT;
     if(PTR==NULL)
     { printf("List is small");
       exit();
     }
   }
   NEW->NEXT = PTR->NEXT;
   PTR->NEXT = NEW;
}
```

Deletion in linear linked list:

(i) Deleleting from the beginning
(ii) ——————— end
(iii) ——————— specific position

(i) Deleting from the beginning:



Algorithm :-

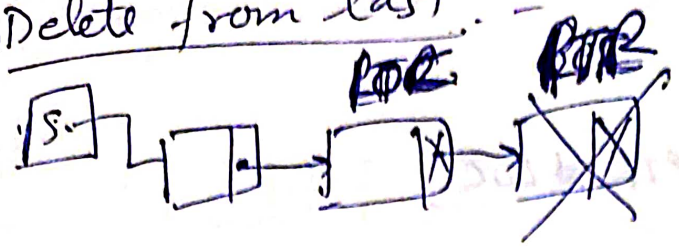
(1) DELETEBEG (L, S, NEXT, PTR)

(1) If S = NULL Then write ~~Overflow~~ underflow and exit.
(2) ~~Set S = S -> next~~ PTR = S;
(3) S = S -> next;
(4) ~~Display~~ (PTR) -> Info
(5) free (PTR);
(6) Exit

C- coding :-

```
void deletebeg()
{   struct node *PTR;
    If (S == NULL)
    Printf("Underflow");
    else{ PTR = S;
    { printf("%d is deleted", S->info));
      S = S -> next;
    } free(PTR);
}
```

iii) Delete from last :-



Algo:-

DELETE (L, S, NEXT, PTR,)

(1) If S = NULL then write under flow & exit.

(2) PTR = S and LOC = PTR -> NEXT

(3) If LOC = NULL then S = NULL and free(PTR); exit

(4)

DELETE (L, S, PTR, LOC, INFO, NEXT)

(1) If S = NULL then write under flow and exit

(2) If S -> Next = NULL then

set PTR = S

set S = NULL

print PTR -> INFO and free(PTR) and Exit

(3) PTR = Start;

(4) Repeat steps 5 & 6 till PTR -> NEXT != NUL

(5) set LOC = PTR

(6) set PTR = PTR -> NEXT

(7) set LOC -> NEXT = NULL

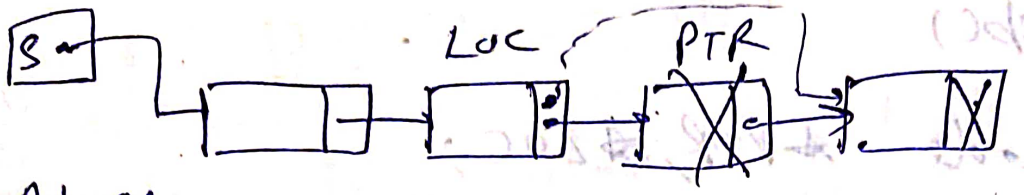
(8) Display PTR -> INFO

(9) free(PTR (10) Exit

c-coding :-

```
void deletelast()
{
    struct node *PTR, *LOC;
    if (S == NULL)
    {printf ("List is empty");
    return;
    }
    else if (S->NEXT == NULL)
    {
        PTR = S;
        S = NULL;
        printf ("deleted value is %d", PTR->INFO);
        free (PTR);
    }
    else
    {  PTR = S;
       while (PTR -> NEXT != NULL)
       {
          LOC = PTR;
          PTR = PTR -> NEXT;
       }
    printf ("%d is deleted", PTR->INFO);
    LOC -> NEXT = NULL;
    free (PTR)
    }
}
```

(iii) Deletion from Specific position:



Algo:-

DELETE (L, S, INFO, NEXT, POS, LOC, PTR, I)

(1) If S = NULL then write overflow & exit

(2) If POS = 1 then

Set PTR = S

set S = S->NEXT

Display PTR -> INFO, free (PTR) and exit

(3) PTR = S and I = 1

(4) Repeat step 5, 6 & 7 till I < POS

(5) set LOC = PTR

(6) set PTR = PTR -> NEXT if PTR = NULL write list is small & exit.

(7) I = I + 1

(8) Set LOC -> NEXT = PTR -> NEXT

(9) Display PTR -> INFO

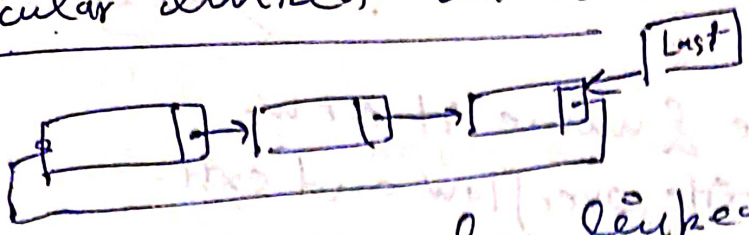
(10) Free (PTR)

(11) Exit

c-coding:-

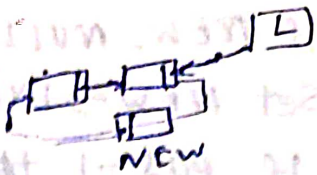
```
void deletesp()
{
  struct node *PTR, *LOC;
  int i, pos;
  printf("enter position");
  scanf("%d", &pos);
  if(S==NULL)
  printf("List is empty");
  else if (pos==1)
  {
    PTR=S;
    S=S->NEXT;
  printf("%d is deleted", PTR->DATA);
  free(PTR);
  }
else
  { for(i=1; i<pos; i++)
    {
      LOC=PTR;
      PTR=PTR->NEXT;
    }
    IF (PTR == NULL) printf("List is small"); return;
  LOC->NEXT = PTR->NEXT;
  printf("%d is deleted", PTR->DATA);
  }free(PTR);
```

Circular linked list:



Insertion in circular linked list:

(1) Insertion at beggining or at end:



① Creat a new node & name it NEW.
② If NEW = NULL then write overflow and exit.
③ NEW -> INFO = ITEM.
④ If Last = NULL then

Last = NEW
NEW -> NEXT = Last and exit

⑤ NEW -> NEXT = Last -> NEXT
⑥ Last -> NEXT = NEW
⑦ Last = NEW * Insertion at first This line will not be written. [Skip this line]
⑧ Exit

C-coding:

```
void addbeg()
{
  struct node *NEW;
  NEW=(struct node *) malloc (size of (struct node));
  printf("enter value of node");
  scanf("%d", &NEW->INFO);
  if (Last == NULL)
  {
    Last = NEW;
    NEW -> NEXT = Last;
  }
  else { NEW -> NEXT = Last -> NEXT;
         Last -> NEXT = NEW;
  } } Last = NEW; ——— // Skip this line for Insertion at start.
```

Insertion at Specific Position:

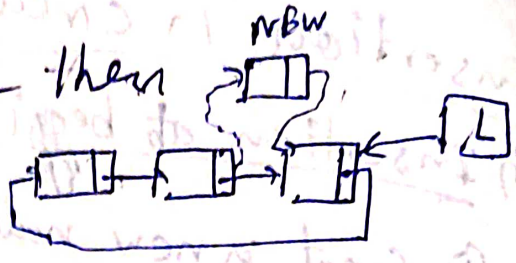
INSERT SPC

① Create a new node & name it NEW.

② If NEW = NULL then write overflow and exit.

③ Set NEW → INFO = ITEM.

④ If POS = 1 and Last = NULL then



~~If Last = NULL then~~

Last = NEW ;

NEW → NEXT = Last and exit

⑤ If Pos = 1 and Last != NULL then

NEW → NEXT = Last → NEXT

Last → NEXT = NEW

~~L = NEW~~ and exit

⑥ PTR = Last → NEXT and I = 1

⑦ Repeat steps ⑧ & ⑨ ~~until~~ till I < (Pos - 1)

⑧ PTR = PTR → NEXT if PTR = Last → ~~NEXT~~ then write list is small and exit.

⑨ I = I + 1

⑩ IF PTR = Last then.

NEW → NEXT = ~~PTR~~ → NEXT

~~PTR~~ → NEXT = NEW

Last = NEW and exit.

⑪ NEW → NEXT = PTR → NEXT

PTR → NEXT = NEW

⑫ Exit

C-coding:-

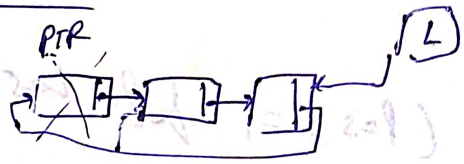
```
void addsp()
{ struct node *NEW, *PTR;
  int pos, i;
NEW = (struct node *) malloc (sizeof(struct node));
printf ("enter value of node");
scanf ("%d", &NEW->INFO);
printf ("enter position");
scanf ("%d", &Pos);
if (Pos == 1 && Last == NULL)
{
    Last = NEW;
    NEW->NEXT = Last;
}
else if (Pos == 1 && Last != NULL)
{ NEW->NEXT = Last->NEXT;
  Last->NEXT = NEW;
  Last = NEW;
}
else { PTR = Last->NEXT;
       for (i = 1; i < (Pos-1); i++)
       { PTR = PTR->NEXT;
         if (PTR = Last->NEXT)
       { printf ("list is small");
         return();
       }
     }
}
```

```
if (PTR==Last)
{ NEW->NEXT = PTR->NEXT;
  PTR->NEXT=NEW;
  Last=NEW;
}
else
  {
    NEW-> NEXT=PTR->NEXT;
    PTR->NEXT=NEW;
  }
}
```

Deletion from circular linked list:

① Delete from start:—

DELETE BEG



① If Last = NULL then write underflow and exit.
② PTR = Last->NEXT.
③ If Last = Last->NEXT then
   Display PTR->INFO
   free(PTR) and exit.
   Last = NULL and exit.
④ Last->NEXT = PTR->NEXT;

① If Last = NULL Then write underflow & exit.
② PRT = Last->Next;
③ Display RTR->INFO is deleted.
④ If Last = Last->NEXT Then
   Last = NULL, free(PTR) and exit.

(5) Last -> NEXT = PTR -> NEXT ;

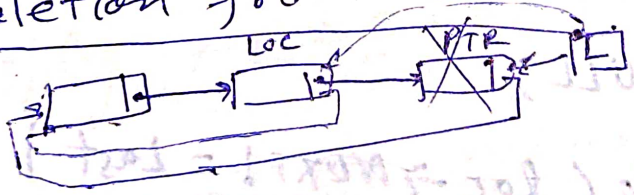
(6) free (PTR).

(7) Exit.

c- coding :-

```
void deletebeg ()
{ struct node *PTR;
 if (Last == NULL)
 printf (" Underflow ");
 else {
      PTR = Last -> NEXT;
      printf(" %d is deleted ", PTR -> INFO);
       if (Last == Last -> NEXT)
        Last = NULL;
        free(PTR);
       else Last -> NEXT = PTR -> NEXT;
      free (PTR);
    }
}
```

(2) Deletion from end :-



DELETE LAST :-

(1) If Last == NULL then write Underflow & exit.

(2) PTR = Last and Loc = Last -> Next ;

(3) Display PTR -> INFO is deleted.

4) if Last = Last -> NEXT then

Last = NULL, free (PTR) and exit..

5) Repeat steps 6) till LOC -> NEXT != Last.

6) Loc = Loc -> NEXT;

7) Loc -> NEXT = Last -> NEXT.

8) Last = Loc.

9) free (PTR).

10) Exit.

C-coding:-

```
void deletelast()
{ struct node *PTR, *LOC;
  if (Last == NULL)
   printf ("Underflow");
  else
   {
     PTR = Last;
     LOC = Last -> NEXT;
    printf("%d is deleted", PTR -> INFO);
     if (Last == Last -> NEXT)
      Last = NULL;
     else {
          while (loc -> NEXT != Last)
              Loc = Loc -> NEXT;
          LOC -> NEXT = LAST -> NEXT;
          Last = LOC;
        }
     free (PTR);
   }
}
```

Deletion from specific positions:- Loc

① If Last = NULL then write underflow and exit.

② PTR = Last → NEXT.

③ if Last = Last → NEXT & Pos = 1 then

④ Last = NULL, Display (PTR → INFO), free(PTR) & exit.

⑤ If Pos = 1 and (Last != Last → NEXT) then

Last → NEXT = PTR → NEXT.

Display (PTR → INFO) and free(PTR) and exit.

⑥ I = 1.

⑦ Repeat steps 7 & 8 till I < P

LOC = PTR and PTR = PTR → NEXT

if (PTR = Last → NEXT then write List is small & exit

⑧ I = I + 1

⑨ If PTR = Last then

Loc → NEXT = Last → NEXT

Last = LOC

Display (PTR → INFO) and free(PTR) & exit.

⑩ LOC → NEXT = PTR → NEXT

⑪ Display PTR → INFO)

⑫ free (PTR)

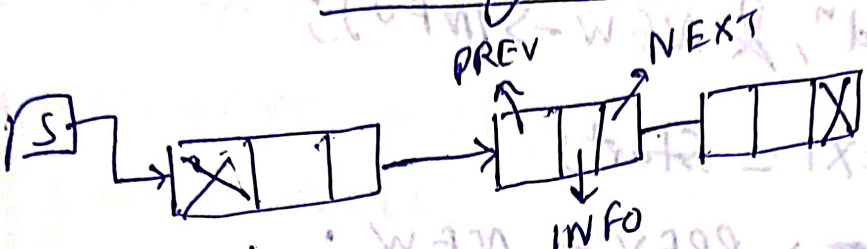
⑬ Exit.

```
void deleteatp()
{
    struct node *PTR, *LOC;
    int i, pos;
    if (Last == NULL)
    printf("Underflow");
    else
    { PTR = Last -> NEXT;
      if (Last == Last -> NEXT && pos == 1)
        { printf("%d is deleted", PTR -> INFO);
          Last = NULLL;
          free (PTR);
          return;
        }
      else if (Pos == 1 && Last != Last -> NEXT)
        {
          Last -> NEXT = PTR -> NEXT;
          printf("%d is deleted", PTR -> INFO);
          free (PTR);
          return ;
        }
      else { i=0;
          for (i=1; i<Pos-1 ; i++)
          { Loc = PTR;
            PTR = PTR -> NEXT;
            if (PTR == Last -> NEXT)
            {printf("list is small");
             return ;
          } }
          if (PTR = last)
          {
            Loc -> NEXT = PTR -> NEXT;
            printf("%d is deleted", PTR -> INFO);
            free (PTR);
            return ;
          else Loc -> NEXT = PTR -> NEXT.
```

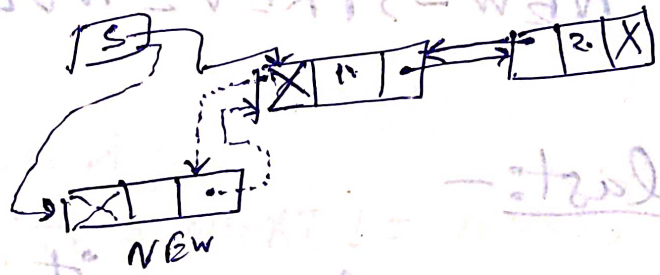
~~printf("%d is deleted", PTR->INFO);~~
~~free(PTR);~~

```
if (PTR == last)
    last = loc;
free(PTR);
```

Doubly linked list:



Insertion at ~~beginning~~ start:-




```
struct node
{ int INFO;
  ~~struct node *NEXT, *PTR;~~
  struct node *NEXT, *PREV;
};
```

INSERT BEG

1. Create a new node and name it NEW.
2. If NEW = NULL then write overflow and exit.
3. Set NEW -> INFO = ITEM
4. Set NEW -> NEXT = S
5. Set Start -> ~~NEXT~~ PREV = NEW
6. Start = NEW
7. NEW -> PREV = NULL
8. Exit

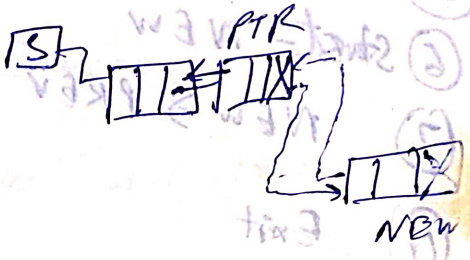
C-coding:

```
Void addlast();
{
  struct node *NEW;
NEW = (struct node *) malloc (sizeof (struct node));
  printf ("enter value of node");
  scanf ("%d", &NEW->INFO);

    NEW->NEXT = Start;

    Start ~~->NEXT~~ -> PREV = NEW;

    Start = NEW;

    ~~Start~~ NEW -> PREV = NULL;

}
```

Insert at last:-

INSERTLAST

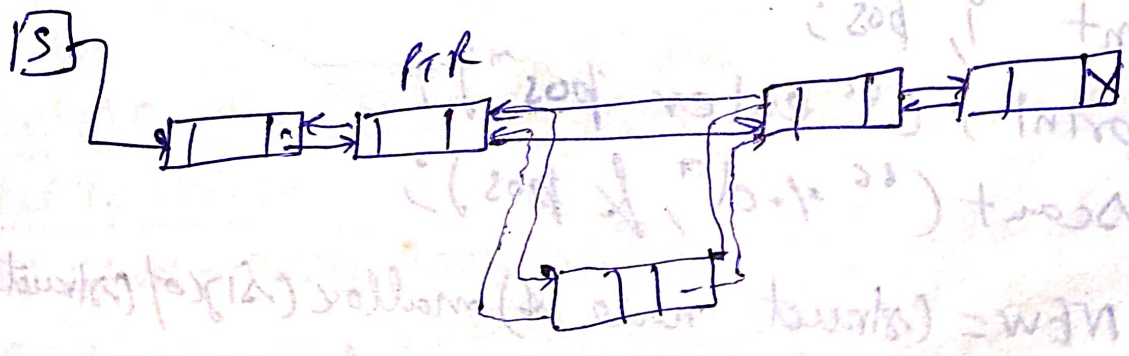
1. Create a new node & name it NEW.
2. If NEW = NULL then write overflow & exit.
3. Set NEW -> INFO = ITEM.
4. Set NEW -> NEXT = NULL.
5. IF S = NULL then S = NEW and NEW -> PREV = NULL and exit.
6. Set PTR = S;
7. Repeat step 8 while PTR -> NEXT != NULL.
8. PTR = PTR -> NEXT.
9. PTR -> NEXT = NEW;
10. NEW -> PREV = PTR.
11. Exit.



c-coding:-

```
void addlast()
{ struct node *NEW, *PTR;
NEW = (struct node *) sizeof (struct node));
printf("enter value of node");
scanf("%d", &NEW->INFO);
NEW->NEXT = NULL;
if (start == NULL)
{ start = NEW;
  NEW->PREV = NULL;
}
else
{ PTR = start;
  while (PTR->NEXT != NULL)
  PTR = PTR->NEXT;
  PTR->NEXT = NEW;
  NEW->PREV = PTR;
}
}
```

Insertion at specific position:



INSERT SP

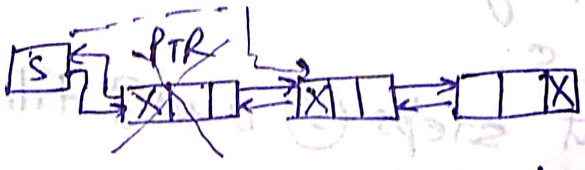
(1) Create a new node & name it NEW
(2) if NEW = NULL then write overflow & exit.
(3) set NEW -> INFO = ITEM
(4) if POS = 1 and START = NULL then
    Set NEW -> NEXT = START
    START -> PREV = NEW
    START = NEW
    NEW -> PREV = NULL and exit.
(5) set I = 1 and PTR = START
(6) Repeat step 7 & 8 till I < POS - 1
(8) set PTR = PTR -> NEXT and if (PTR = NULL) then write list is small and exit.
(9) I = I + 1.
(10) NEW -> NEXT = PTR -> NEXT.
(11) NEW -> PREV = PTR
(12) PTR -> NEXT -> PREV = NEW
(13) PTR -> NEXT = NEW
(14) Exit

C- coding :-

```
void addsp()
{
    struct node *NEW, *PTR;
    int i, pos;
    printf("enter pos");
    scanf("%d", &pos);
    NEW = (struct node *) malloc (sizeof(struct node));
    printf("enter value of node");
```

```
scanf("%d", &NEW->INFO);
if (POS==1)
{
  NEW->NEXT = start;
  Start->PREV = NEW;
  Start = NEW;
  NEW->PREV = NULL; could exit.
}
else
{
  PTR = Start;
  for(i=0; i<(pos-2); i++)
  {
    PTR = PTR->NEXT;
    if(PTR == NULL)
    {printf("list is small");
     return;
    }
  }
  NEW->NEXT = PTR->NEXT;
  NEW->PREV = PTR;
  PTR->NEXT => NEW PREV = NEW;
  PTR->NEXT = NEW
}
}
```

Delete at start:



DELETEBEG:-

① If Start = NULL then write underflow and exit.

② PTR = Start; ③ ~~If S->NEXT = NULL then S = NULL~~ ~~PTR->INFO, free(PTR)~~

④ Start = Start->NEXT

④ Display PTR->INFO.

⑤ Start -> PREV = NULL.

⑥ free(PTR).

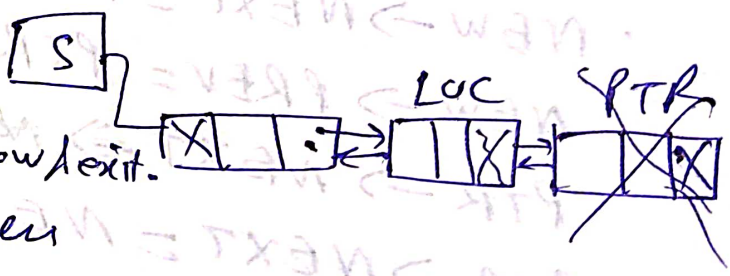
⑦ Exit.

c- coding:

```
void deleteStart()
{
  struct node *PTR;
  if(start == NULL)
  printf("Under flow");
  else
    {
      PTR = start;
      printf("%d is deleted", PTR->INFO);
      start = start->NEXT;
      start->PREV = NULL;
      free(PTR);
    }
}
```

Delete from last:-

DELETELAST



① If S = NULL then write underflow & exit.

② If S->NEXT = NULL Then

Set PTR = S;

S = NULL;

Distblay PTR->INFO, free(PTR) and exit.

③ PTR = S.

④ Repeat steps ⑤ and ⑥ till PTR->NEXT != NULL

⑤ set LOC = PTR.

⑥ PTR = PTR->NEXT.

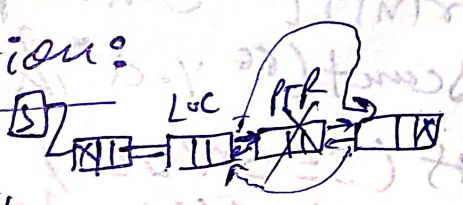
⑦ Set LOC->NEXT = NULL.

⑧ Display PTR -> INFO.
⑨ free (PTR).
⑪ Exit.

c- coding :-

```
void deletelast()
{ struct node *PTR, *LOC;
  if (S==NULL)
  printf("underflow");
  else if (S->NEXT==NULL)
      {  PTR=S;
         S=NULL;
       printf("%d is deleted", PTR->INFO);
        free (PTR);
      }
  else {
        PTR=S;
        while(PTR->NEXT != NULL)
        {  LOC = PTR;
           PTR=PTR->NEXT;
        }
        LOC->NEXT=NULL;
       printf("%d is deleted", PTR->INFO);
       free(PTR);
      }
}
```

Delete from specific position:



DELETESP

① If S=NULL write underflow & exit

② If POS = 1 then

Set PTR = S

set S = S->NEXT

S->PREV = NULL

Display PTR->INFO, free(PTR) & exit.

③ PTR = S and I = 0

④ Repeat steps ⑤, ⑥ and ⑦ till I < POS-1

⑤ set LOC = PTR

⑥ PTR = PTR->NEXT IF (PTR == NULL) then write list is small and exit.

⑦ I = I + 1

⑧ Display PTR->INFO

⑨ LOC->NEXT = PTR->NEXT.

⑩ PTR->NEXT->PREV = PTR->PREV.

⑪ free(PTR).

⑫ Exit.

c-coding :-

```
void deletesp()
{
    struct node *PTR, *LOC;
    int i, pos;
    if(S==NULL)
    printf("underflow");
    else
    printf
    printf("enter position");
    scanf("%d", &pos);
    if(S==NULL)
    printf("Underflow");
```

```
else if (pos==1)
  {PTR=S;
   S=S->NEXT;
   S->PREV=NULL;
  printf("%d is deleted", PTR->INFO);
  free(PTR);
  }
else {
      PTR=S;
      for(i=0;i<pos;i++)
        {
          LOC=PTR;
          PTR=PTR->NEXT;
         if(PTR==NULL)
         {printf("List is small");
          return;
         }
        }
    LOC->NEXT=PTR->NEXT;
    PTR->NEXT->PREV=PTR->PREV;
    printf("%d is deleted", PTR->INFO);
    free(PTR);
    }
}
```

Traversing (Displaying) the Doubly linked list:

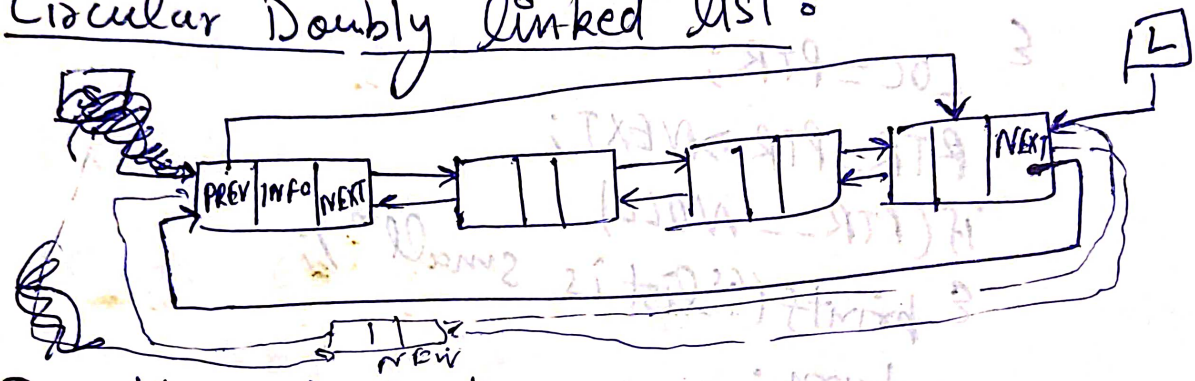
TRAVERSE

1. Set PTR = Start;
2. Repeat step 3 & 4 while PTR != NULL.
3. Display PTR->INFO
4. PTR = PTR->NEXT.
5. Exit.

c-coding :-

```
void traverse()
{
  struct node *PTR;
  PTR = Start;
  while (PTR != NULL)
  { printf("%d \t", PTR->INFO);
    PTR = PTR->NEXT;
  }
}
```

Circular Doubly linked list:



Insertion at start:-

(1) Create a new node and name it NEW.
(2) If NEW = NULL then write overflow and exit.
(3) ~~if (S == NULL) then~~ NEW->INFO = ITEM.
(4) If L = NULL then

```
L = NEW;
NEW->NEXT = NEW;
NEW->PREV = NEW and Exit.
```

(5) NEW->NEXT = L->NEXT.
(6) NEW->PREV = L.
(7) L->NEXT->PREV = NEW
(8) L->NEXT = NEW (9) Exit.

e-coding:-

```
void addstart()
{ struct node *NEW;
NEW=(struct node *)malloc(sizeof(struct node));
printf("enter value of node");
scanf("%d",&NEW->INFO);
if(Last==NULL)
{ Last=NEW;
  NEW->NEXT=NEW;
  NEW->PREV=NEW;
}
else
{
  NEW->NEXT=Last->NEXT;
  NEW->PREV=Last;
  Last->NEXT->PREV=NEW;
  Last->NEXT=NEW;
}
}
```

~~Void~~ Insert at last:

① Create a new node & name it NEW.
② If NEW=NULL then write overflow and exit.
③ NEW->INFO=ITEM
④ if Last=NULL then

Last=NEW.
NEW->NEXT=NEW.
NEW->PREV=NEW and exit.

⑤ NEW->NEXT=Last->NEXT.
⑥ NEW->PREV=Last.

(7) ~~Lst->NEXT~~ Last->NEXT->PREV = NEW

(8) L->NEXT = NEW

(9) Last = NEW

(10) Exit.

C-Coding :-

```
void addlast()
{
  struct node *NEW;
NEW=(struct node *)malloc(sizeof(struct node));
printf("enter value of node");
scanf("%d",&NEW->INFO);
 if(Last==NULL)
   { Last=NEW;
     NEW->NEXT=NEW;
     NEW->PREV=NEW;
   }
 else
     { NEW->NEXT=Last->NEXT;
       NEW->PREV=Last;
       Last->NEXT->PREV=NEW
       Last->NEXT=NEW;
       Last=NEW;
     }
}
```

Insert at specific position :-

(1) Create a new node & name it NEW.

(2) If NEW = NULL then write overflow and exit.

(3) Set NEW->INFO = ITEM.

④ If Pos = 1 and Last = NULL then
Last = NEW;
NEW -> NEXT = NEW;
NEW -> PREV. and exit.

⑤ If Pos = 1 and Last ≠ NULL then
NEW -> NEXT = Last -> NEXT.
NEW -> PREV = Last.
Last -> NEXT -> PREV = NEW.
Last -> NEXT = NEW. and exit.

⑥ PTR = Last -> NEXT. and I = 0

⑦ Repeat step ⑧ & ⑨ till i < (Pos-2)

⑧ PTR = PTR -> NEXT if PTR = Last -> NEXT then write list is small and exit.

⑨ I = I+1.

⑩ ~~If PTR = Last then~~
NEW -> NEXT = ~~PTR~~ -> NEXT.
NEW -> PREV = Last.
Last -> ~~NEXT~~ -> PREV = NEW.
Last -> NEXT = NEW
Last = NEW and exit.

⑪ NEW -> NEXT = PTR -> NEXT.

⑫ ~~PTR~~ -> NEXT -> PREV = NEW

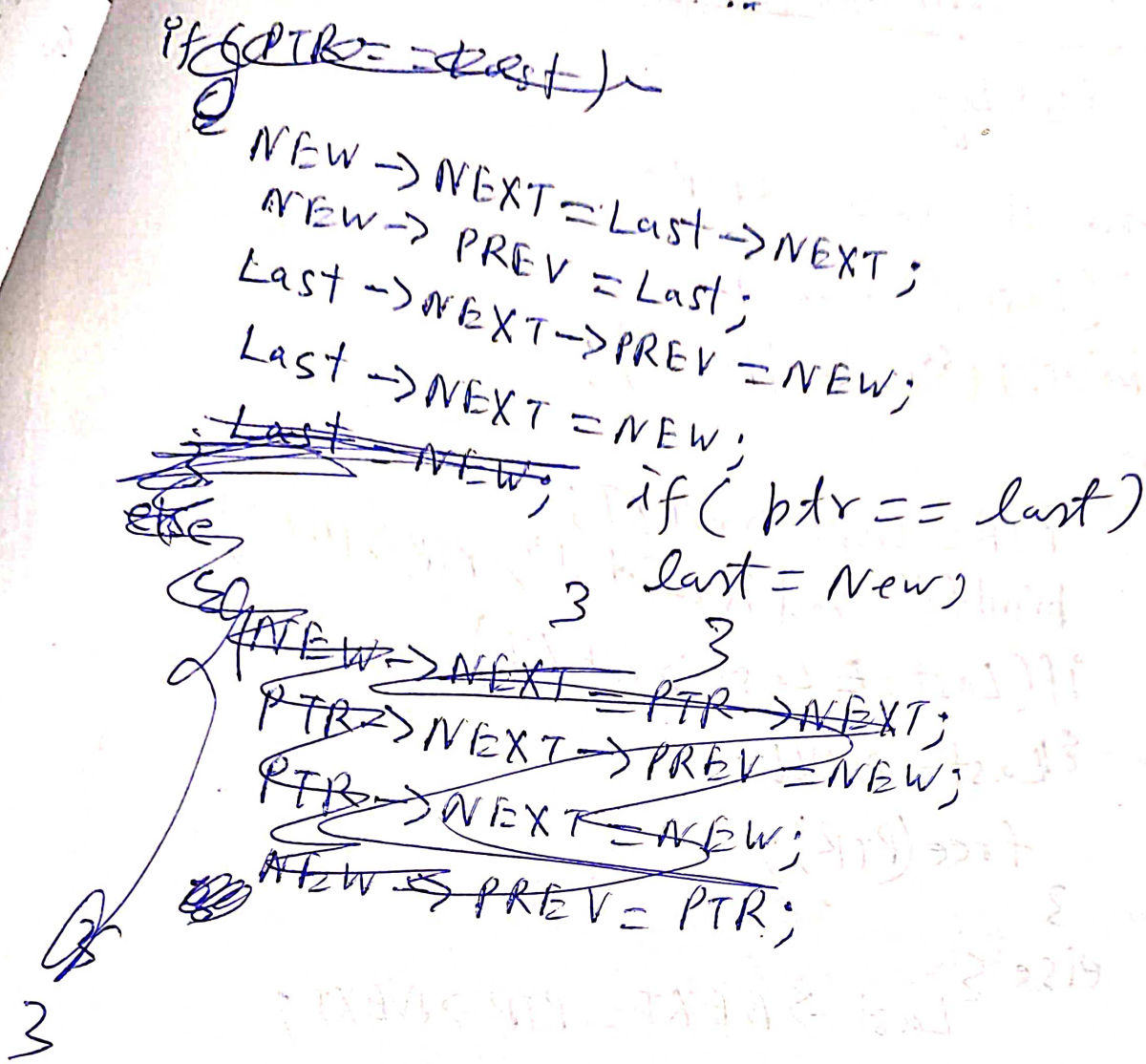
⑬ PTR -> NEXT = NEW

⑭ NEW -> PREV = PTR

⑮ ~~exit~~ If PTR = last then last = NEW.
Exit

C - coding:-

```
void addsp()
{
  struct node *PTR, *NEW;
  int i, pos;
  printf("enter position");
  scanf("%d", &pos);
  NEW=(struct node *)malloc(sizeof(struct node));
  printf("enter value of node");
  scanf("%d", &NEW->INFO);
 if (pos==1 && Last==NULL)
  {
    Last=NEW;
    NEW->NEXT=NEW;
    NEW->PREV=NEW;
  }
 else if (pos==1 && Last!=NULL)
  {
    NEW->NEXT=Last->NEXT;
    NEW->PREV=Last;
    Last->NEXT->PREV=NEW;
    Last->NEXT=NEW;
  }
 else {
       PTR=Last->NEXT;
       for(i=0; i<(pos-2); i++)
         {
          PTR=PTR->NEXT;
          if (PTR==NULL)
           {
            printf("list is small");
            return;
           }
         }
```



Deletion from start:

1. IF Last = NULL then write underflow and exit.
2. PTR = Last -> NEXT.
3. Display (PTR -> INFO) is deleted.
4. If Last = Last -> NEXT then
   Last = NULL, free (PTR) and exit.
5. Last -> NEXT = PTR -> NEXT;
6. Last -> NEXT -> PREV = Last;
7. free (PTR).
8. Exit

```
void deletebeg()
{
  struct node *PTR;
  if (Last == NULL)
    printf("underflow");
  else{
      PTR = Last -> NEXT;
      printf("%d is deleted", PTR->INFO);
      if(Last == Last -> NEXT)
      {Last = NULL;
        free(PTR);
      }
      else {
          Last -> NEXT = PTR -> NEXT;
          PTR -> NEXT -> PREV = Last;
          free (PTR);
        }
    }
}
```

Deletion from last: -

① If Last = NULL then write underflow and exit.
② PTR = Last and LOC = Last -> NEXT.
③ Display PTR -> INFO is deleted.
④ If Last = Last -> NEXT then
Last = NULL, free (PTR) and exit.
⑤ Repeat step 6 till LOC -> NEXT != Last.
⑥ LOC = LOC -> NEXT

7. LOC->NEXT = Last->NEXT.
8. Last->NEXT->PREV = LOC.
9. Last = LOC.
10. free (PTR).
11. Exit.

c-coding :-

```
void deletelast()
{ struct node *PTR, *LOC;
  if(Last == NULL);
  printf("Underflow");
  else
    { PTR = Last;
      LOC = Last->NEXT;
      printf("%d is deleted", PTR->INFO);
  if(Last == Last->NEXT)
    { Last = NULL;
      free(PTR);
    }
  else
    { while(LOC->NEXT != Last)
        LOC = LOC->NEXT;
      LOC->NEXT = Last->NEXT;
      Last->NEXT->PREV = LOC;
      Last = LOC;
      free(PTR);
    }
  }
}
```

Deletion from specific position:

① If Last = NULL then write underflow and exit.

② PTR = Last -> NEXT.

③ If Pos=1 and Last = Last -> NEXT then
Last = NULL, Display (PTR->INFO), free(PTR)
and exit.

④ If Pos=1 and Last != Last -> NEXT then
Last -> NEXT = PTR -> NEXT.
PTR -> NEXT -> PREV = Last. Display (PTR->INFO)
free (PTR) and exit.

⑤ I = 0

⑥ Repeat step ⑦ & ⑧ till I < Pos-1.

⑦ LOC = PTR.

⑧ PTR = PTR -> NEXT if (PTR = Last -> NEXT) then write list is small and exit.

⑨ I = I+1

⑩ If PTR = Last then
LOC -> NEXT = Last -> NEXT.
Last -> NEXT -> PREV = LOC. → Display (PTR->INFO)
Last = LOC., free(PTR). and exit.

⑪ LOC -> NEXT = PTR -> NEXT.

⑫ PTR -> NEXT -> PREV = LOC

⑬ Display (PTR->INFO)

⑭ free (PTR)

⑮ exit.

C-Coding :-

```
void deletespc()
{ struct node *PTR, *LOC;
  int i,pos;
  printf("enter position\n");
  scanf("%d",&pos);
  if (Last == NULL)
  printf("Underflow\n");
  else
  { PTR = Last->NEXT;
    if ( pos==1 && Last = Last->NEXT)
    { Last = NULL;
      printf("%d is deleted\n", PTR->INFO);
      free(PTR);
    }
    else if (pos==1 && Last != Last->NEXT)
    { Last->NEXT = PTR->NEXT;
      PTR->NEXT->PREV = Last;
      printf("%d is deleted\n", PTR->INFO);
      free(PTR);
    }
    else { for (i=0; i<(pos-1); i++)
           { LOC = PTR;
             PTR = PTR->NEXT;
             if (PTR == Last->NEXT)
             { printf("list is small");
               return;
             }
           }
           LOC->NEXT = PTR->NEXT;
           PTR->NEXT->PREV = LOC;
           printf("%d is deleted\n", PTR->INFO);
           if (PTR == Last)
             Last = LOC;
           free(PTR);
    }
}
```

Traversing of circular Doubly linked list:-

① Set ~~PTR = Last -> NEXT~~. If Last = NULL then list is empty & exit.

② Set PTR = Last -> NEXT.

③ ~~If LastPR = LAST -> NEXT~~

④ ~~If Last = Last -> NEXT then~~

③ Display (PTR -> INFO) ~~and Exit~~.

④ PTR = PTR -> NEXT.

⑤ Repeat Step ⑥ & ⑦ while PTR != last -> NEXT.

⑥ ⑤ Display PTR -> INFO

⑦ Set PTR = PTR -> NEXT

⑧ Exit.

* same for circular linked list

C-coding:-

```
void traverse()
{ struct node *PTR;
  if (Last == NULL)
  printf("list is empty");
  else
  { PTR = Last -> NEXT;
    printf("%d", PTR->INFO);
    ~~if (Last == Last->NEXT)~~
    ~~{ printf("%d\t", PTR->INFO);~~
    ~~else~~ PTR = PTR->NEXT;
    while (PTR != last->NEXT)
    { printf("%d\t", PTR->INFO);
      PTR = PTR->NEXT;
    }
  }
}
```

Polynomial Representation:-

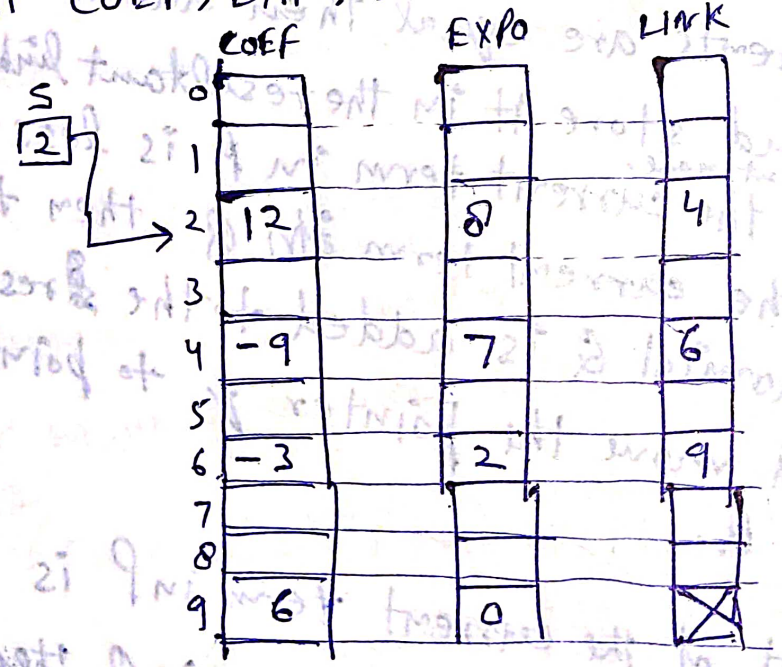
Linked list is used to represent polynomial expression. In linked list representation of polynomial each node contains three fields:

(i) Coefficient field : holds the value of coefficient of term
(ii) Exponent field : contains the exponent value of that term.
(iii) Link field : contains the address of next term.

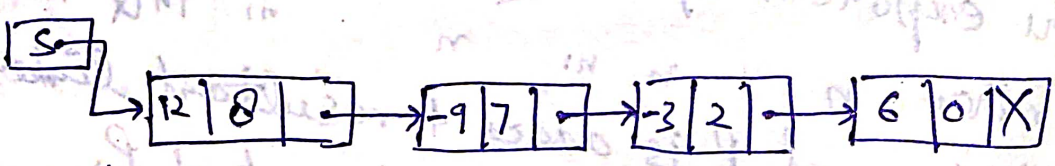
Exp:- Represent the polynomial

$$P(x) = 12x^8 - 9x^7 - 3x^2 + 6$$ (i) using Array (ii) linked list

ans:- (i) It will requires three linear arrays which will call COEF, EXPO, LINK and is represented as:



(ii)




```
struct node
{ int coef;
  int expo;
  struct node * next;
};
```

Addition of two polynomial:

The steps involved in addition of two polynomial are given as follows:

(i) Read the number of terms in the first polynomial P.

(ii) Read the coefficients and exponents of the first polynomial

(iii) Read the number of terms in the second polynomial Q.

(iv) Read the coefficients and exponents of the second polynomial.

(v) Set the temporary pointers p and q in start to traverse the two polynomials P and Q respectively.

(vi) compare the exponents of two polynomials [starting from the first node] pointed by p & q.

(a) If both exponents are equal then add the coefficients and store it in the resultant linked list. and move p & q to next node.

(b) If the exponent of the current term in P is less than the exponent of the current term in Q then term of second polynomial Q is added to the resultant linked list. And move the pointer q to point to the next node of Q.

(c) If the exponent of the current term in P is greater than n n n1 n n1 in Q then current term of P is added to resultant linked list and move the pointer p to next node of P.

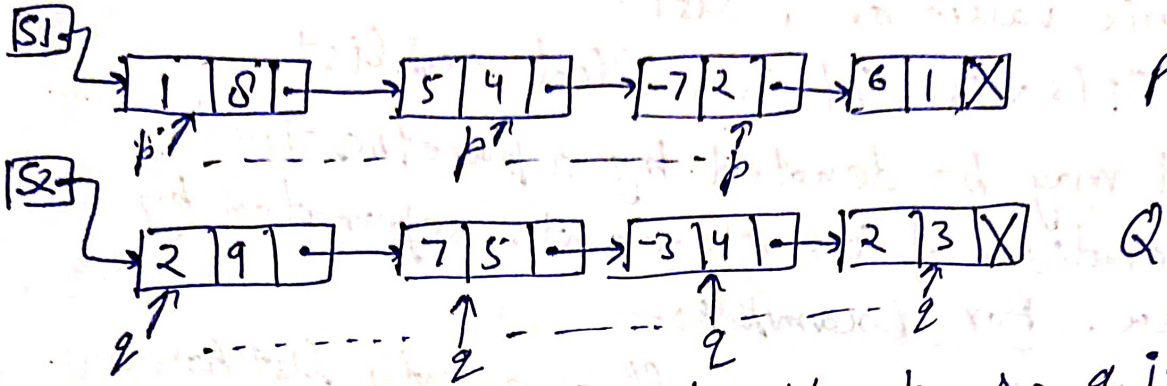
(vii) Repeat step 6 until one list is empty.

(viii) Append the remaining nodes of either of the polynomials to the resultant linked list.

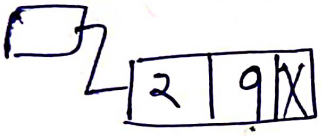
Ex/p:-

$P(1) = x^8 + 5x^4 - 7x^2 + 6x$

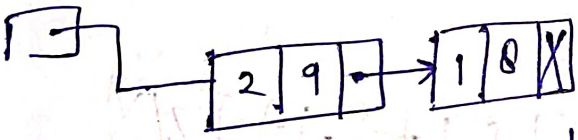
$P(2) = 2x^9 + 7x^5 - 3x^4 + 2x^3$



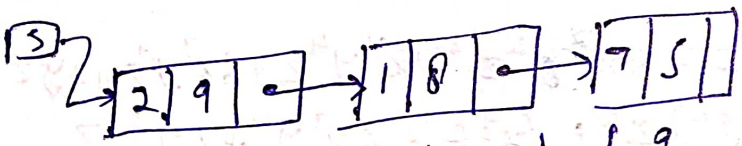
Comare exponent of q is greater than p so q is added to resultant linked list and move q to next node.



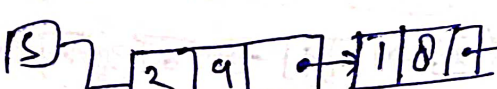
exponent of p is greater than q so p is added to R and move p to next node.



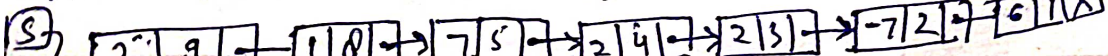
exponent of q is greater than p so q is added to R and move q to next node.



exponent of p = exponent of q so coefficients of both are added and store it resultant list & move p & q to next



exponent is q is greater than p so q is added to R and move q to next. q is null so p is added to R. nodes of P are added to R.



Generalised Linked list :-

A generalised link is a sequence of $n \geq 0$ elements.

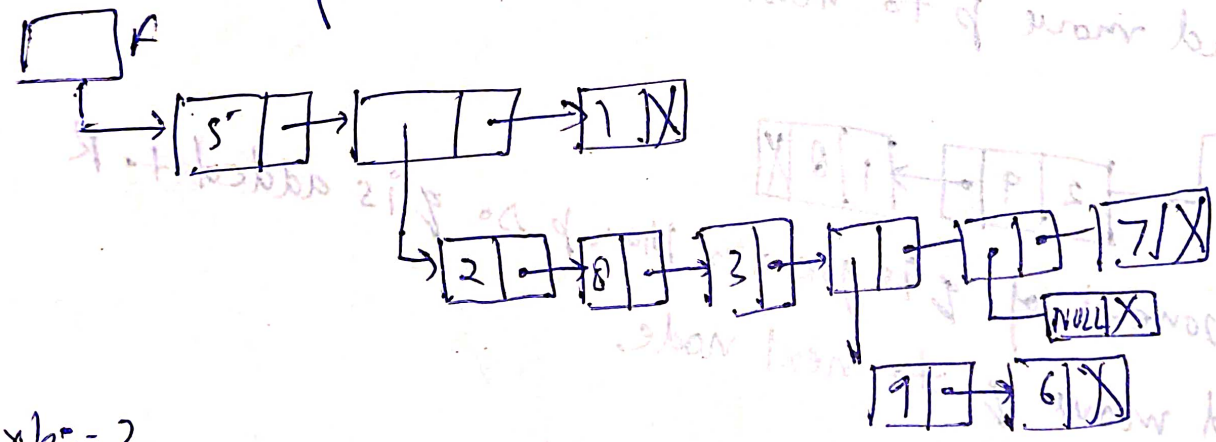
(i) $A = (a_0, a_1, a_2 \text{---} a_{n-1})$ where a_i is either atomic value or a list.

when a_i is list, it is called sublist.

A list may be denoted by a parethesised enumeration of its elements separated by comma. For Example:-

1. $A = ()$ represents null or empty list have 0 length.
2. $A = (2, (3,4), 8)$ list having length 3, there are two atomic ~~value~~ node and a list having Two elements.

Exp1:- $A = (5, (2, 8, 3, (9, 6), (), 7), 1)$

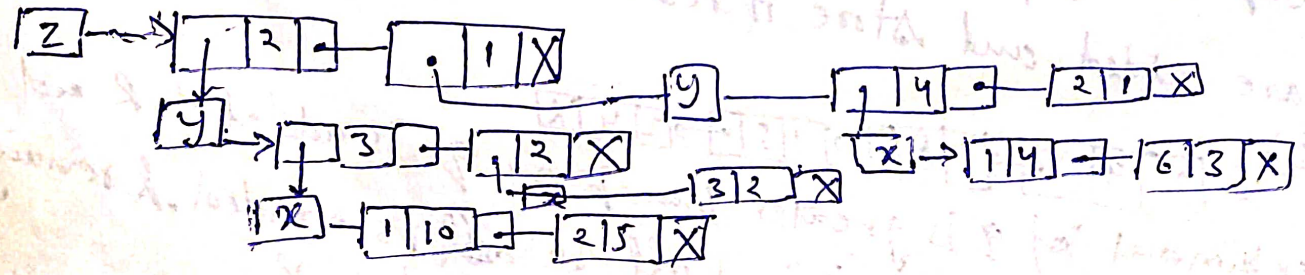


Exp:- 2

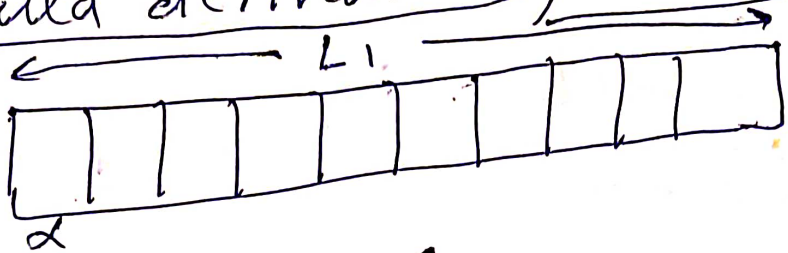
$P(x,y,z) = x^{10}y^3z^2 + 2x^5y^3z^2 + 3x^2y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

$P(x,y,z) = (x^{10}y^3 + 2x^5y^3 + 3x^2y^2)z^2 + (x^4y^4 + 6x^3y^4 + 2y)z$

$P(x,y,z) = [(x^{10} + 2x^5)y^3 + 3x^2y^2]z^2 + [(x^4 + 6x^3)y^4 + 2y]z$



Formula derivation for 1-D array:



Let Base = α, $w = 1$

Add (A[1]) = α

Add (A[2]) = $\alpha + 1$

Add (A[3]) = $\alpha + 2$

Add (A[4]) = $\alpha + 3$

⋮

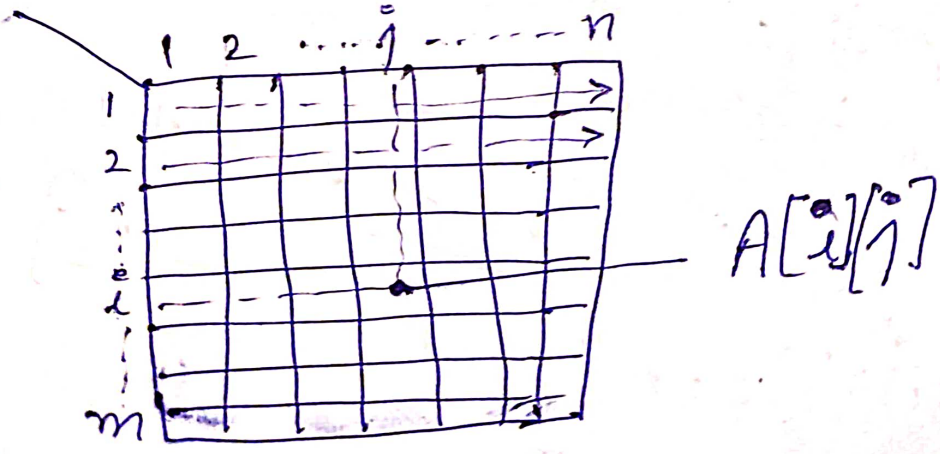
Add (A[k]) = $\alpha + (k-1)$

for word size w.

Add (A[k]) = $\alpha + w(k-1)$

Add (A[k]) = BA + w(E1)

Formula Derivation of 2-D Array :— (Row-major order)



Let Base Add = α, $w = 1$

Add of $(A[1][1]) = \alpha$

$(A[1][2]) = \alpha + 1$

$(A[1][3]) = \alpha + 2$

$A[1][n] = \alpha + n - 1$

$A[2][1] = \alpha + n - 1 + 1 = \alpha + n$

$A[2][2] = \alpha + n + 1$

$A[2][3] = \alpha + n + 2$

$A[2][n] = \alpha + n + n - 1 = \alpha + 2n - 1$

$A[3][1] = \alpha + 2n$

$A[3][2] = \alpha + 2n + 1$

$A[3][n] = \alpha + 2n + n - 1 = \alpha + 3n - 1$

$A[4][1] = \alpha + 3n$

~~$A[i][1] = \alpha + (n-1)n$~~

~~$A[i][2] = \alpha + i * n + 1$~~

$A[i][1] = \alpha + (i-1) * n$

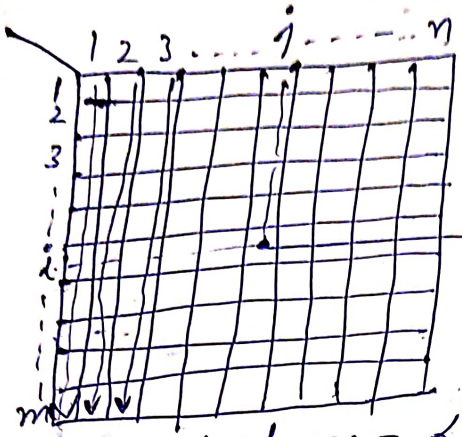
$A[i][2] = \alpha + (i-1) * n + 1$

$A[i][3] = \alpha + (i-1) * n + 2$

$A[i][j] = \alpha + (i-1) * n + (j-1)$

Add of $(A[i][j]) = BA + w[E_1 * L_2 + E_2]$

Formula derivation of 2-D Arrays :-
(Column major order)



Let Base Address = α, $w = 1$

Address of $A[1][1] = \alpha$

$A[2][1] = \alpha + 1$

$A[3][1] = \alpha + 2$

$\vdots$

$A[m][1] = \alpha + m - 1$

$A[1][2] = \alpha + m$

$A[2][2] = \alpha + m + 1$

$A[3][2] = \alpha + m + 2$

$A[m][2] = \alpha + m + m - 1 = \alpha + 2m - 1$

$A[1][3] = \alpha + 2m$

$\vdots$

$A[1][j] = \alpha + (j-1) * m$

$A[2][j] = \alpha + (j-1) * m + 1$

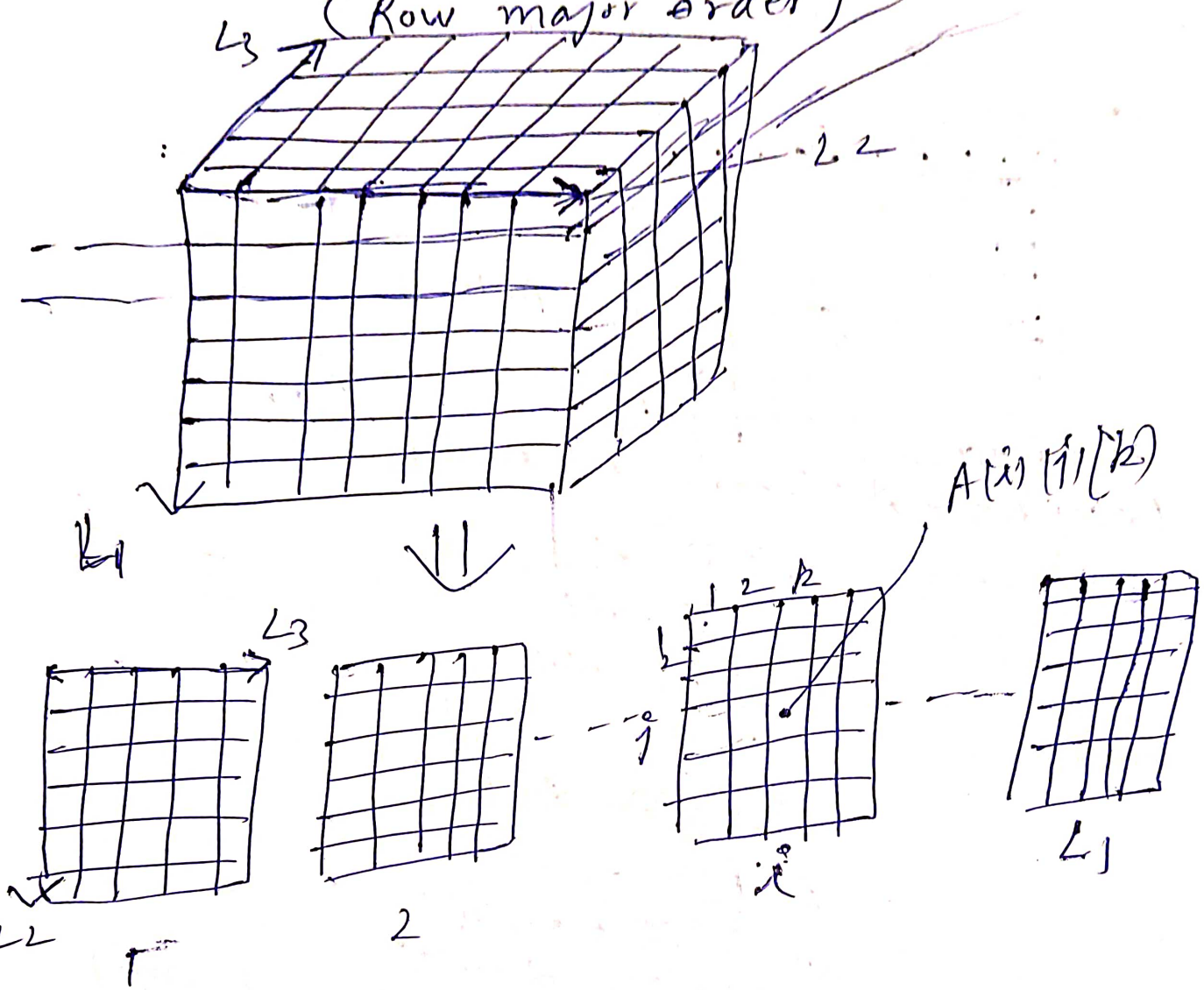
$A[3][j] = \alpha + (j-1) * m + 2$

$A[i][j] = \alpha + (j-1) * m + (i-1)$

Add of $A[i][j] = BA + w[E_2 l_1 + E_1]$

formula derivation for 3-D Array :

(Row major order)



$A[1][1][1] = \alpha$

$A[2][1][1] = \alpha + L_2 L_3$

$A[3][1][1] = \alpha + 2L_3 + L_2 L_3$

$A[3][1][1] = \alpha + 2 L_2 L_3$

$A[i][1][1] = \alpha + (i-1) L_2 L_3$

$A[i][1][2] = \alpha + (i-1) L_2 L_3 + 1$

$A[i][1][L_3] = \alpha + (i-1) L_2 L_3 + L_3 - 1$

$A[i][2][1] = \alpha + (i-1) L_2 L_3 + L_3$

$A[i][2][2] = \alpha + (i-1) L_2 L_3 + L_3 + 1$

$A[i][2][3] = \alpha + (i-1) L_2 L_3 + L_3 + 2$

$A[i][2][L_3] = \alpha + (i-1)L_2L_3 + L_3 + L_3 - 1$

$A[i][3][1] = \alpha + (i-1)L_2L_3 + 2L_3$

$A[i][j][1] = \alpha + (i-1)L_2L_3 + (j-1)L_3$

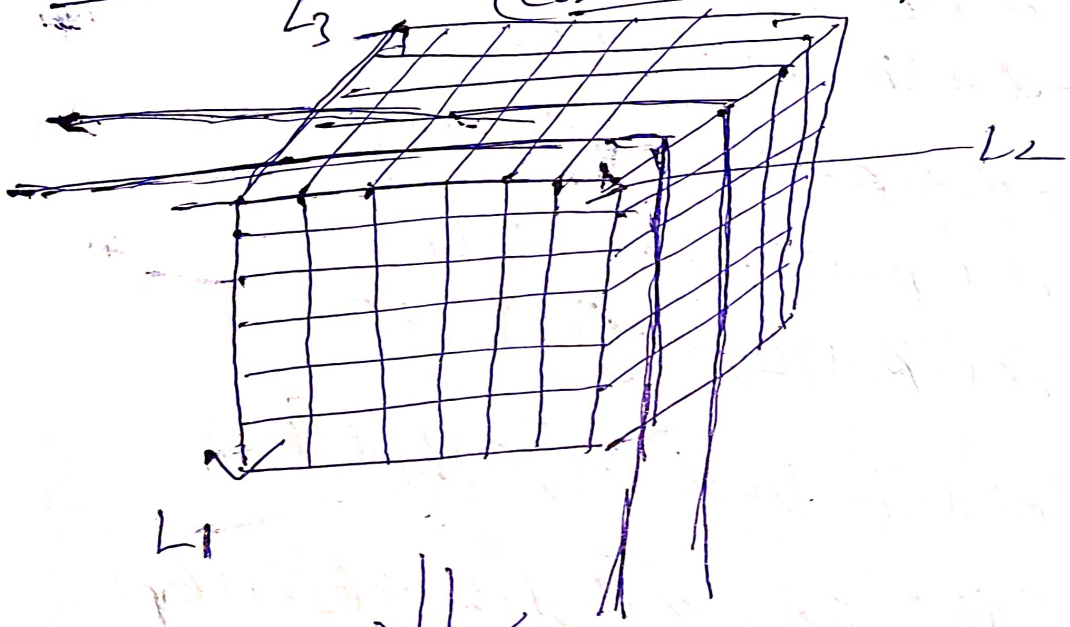
$A[i][j][2] = \alpha + (i-1)L_2L_3 + (j-1)L_3 + 1$

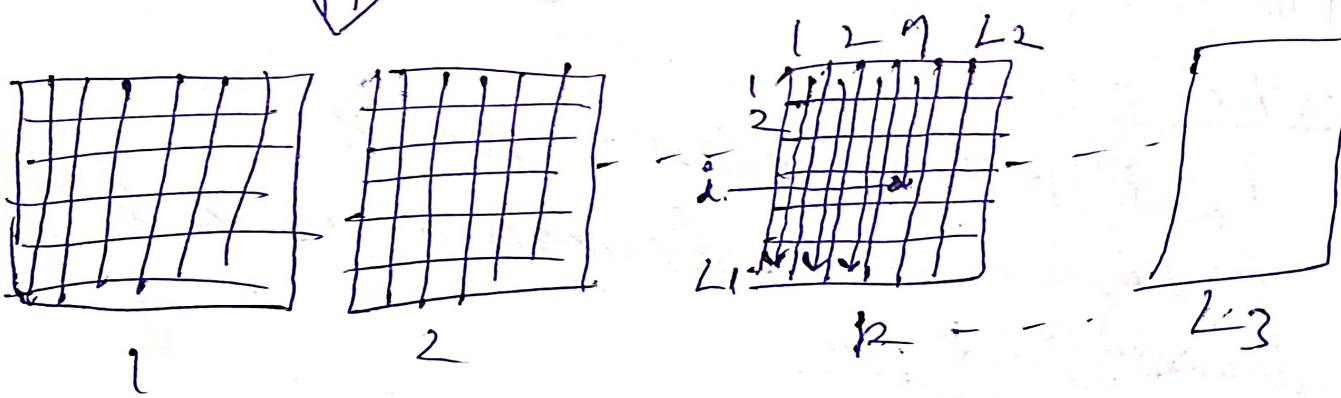
$A[i][j][3] = \alpha + (i-1)L_2L_3 + (j-1)L_3 + 2$

$A[i][j][k] = \alpha + (i-1)L_2L_3 + (j-1)L_3 + (k-1)$

$Add of (A[i][j][k]) = BA + W[E_1L_2L_3 + E_2L_3 + E_3]$

Formula Derivation of 3D-Array: (Column major order)





$A[1][1][1] = \alpha$

$A[1][1][2] = \alpha + L_1 L_2$

$A[1][1][3] = \alpha + L_1 L_2 + L_1 L_2 = \alpha + 2 L_1 L_2$

$\vdots$

$A[1][1][k] = \alpha + (k-1) L_1 L_2$

$A[2][1][k] = \alpha + (k-1) L_1 L_2 + 1$

$A[3][1][k] = \alpha + (k-1) L_1 L_2 + 2$

$\vdots$

$A[L_1][1][k] = \alpha + (k-1) L_1 L_2 + L_1 - 1$

$A[1][2][k] = \alpha + (k-1) L_1 L_2 + L_1$

$A[2][2][k] = \alpha + (k-1) L_1 L_2 + L_1 + 1$

$A[3][2][k] = \alpha + (k-1) L_1 L_2 + L_1 + 2$

$\vdots$

$A[L_1][2][k] = \alpha + (k-1) L_1 L_2 + L_1 + L_1 - 1$

$A[1][3][k] = \alpha + (k-1) L_1 L_2 + 2 L_1$

$\vdots$

$A[i][j][k] = \alpha + (k-1) L_1 L_2 + (i-1) L_1$

$A[2][j][k] = \alpha + (k-1) L_1 L_2 + (j-1) L_1 + 1$

$A[3][j][k] = \alpha + (k-1) L_1 L_2 + (j-1) L_1 + 2$

$\vdots$

$A[i][j][k] = \alpha + (k-1) L_1 L_2 + (j-1) L_1 + (i-1)$

Add of $A[i][j][k] = BA + W[E_3 L_1 L_2 + E_2 L_1 + E_1]$

where :-

E_k Effective index $= k - LB$

L_1, L_2, L_3 are length of 1, 2 and 3 dimension array of

Q. A two dimensional array is defined as A[3:7, -1:4] requires 4 words per memory cell. Find the loc of A[5,2] if array is implemented Row major order and base address is given as 200.

Ans:

$$Add[A[i][j]] = BA + w[E_1 l_2 + E_2]$$

$$Add[A[i][j]] = BA + w[(i - lb1) l_2 + (j - lb2)]$$

where $lb1$ = lower bound 1 = 3

$lb2$ = lower bound 2 = -1

$l_1 = 7 - 3 + 1 = 5$

$l_2 = 4 - (-1) + 1 = 6$

So. $Add(A[5,2]) = BA + w[(i - lb_1) l_2 + (j - lb_2)]$

$= 200 + 4[(5-3) * 6 + 2 - (-1)]$

$= 200 + 4[12 + 3]$

$= 260$

Q. A[20][5], BA = 200, w = 4. Find out the address of A[12,3] using column major order in C language.

Ans:

$$Add[A[i][j]] = BA + w[E_2 l_1 + E_1]$$

$Add(A[i][j]) = BA + w[(j - lb_2) l_1 + (i - lb_1)]$

where

$BA = 200, w = 4, lb_1 = lb_2 = 0, l_1 = 20$

$Add\ A[12,3] = 200 + 4[(3-0) * 20 + (12-0)]$

$= 200 + 4[60 + 12]$

$= 200 + 288$

$= 488$

Q3 Array A[20][50] required 4 bytes of storage for each element. Base Address is 2000. Find loc of A[10][10] when the array is stored in (i) Row major order in C language (ii) and column major order.

Ans: $Add[A[i][j]] = BA + w[E_1 l_2 + E_2]$ [Row major]

$Add[A[i][j]] = BA + w[(i - lb_1) l_2 + (j - lb_2)]$

$BA = 2000, w = 4, i = 10, j = 10, lb_1 = lb_2 = 0, l_2 = 50$

$Add(A[10][10]) = 2000 + 4[(10-0) * 50 + (10-0)]$

$= 2000 + 2040$

$= 4040$

$Add[A[i][j]] = BA + w[E_2 l_1 + E_1]$ column major

$= 2000 + 4[(j - lb_2) * l_1 + (i - lb_1)]$

$= 2000 + 4[(10-0) * 20 + 10]$

$= 2000 + 840$

$= 2840$