

Unit-1

Data: value or set of values.

Ex: Student (Entity)

| Name | Roll No. | Age | Marks | DOB |
|------|----------|-----|-------|-----------|
| X | 1 | 20 | 75 | 29/2/2000 |
| Y | 2 | 22 | 63 | 7/5/1999 |
| Z | 3 | 10 | 40 | 2/3/2000 |
| : | : | : | : | : |
| : | : | : | : | : |
| : | : | : | : | : |

Attributes: properties of Entity.

Ex: (Name, Roll No, Age, Marks, DOB etc)

Record: Each Row is called record.

Field: Each column is called field.

Information: Meaningful or Processed data.

Data Structure: - DS is a formal for storing & organizing data.
- The logical and mathematical 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 structures:

- 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.
Ex: array.

Dynamic DS: A Dynamic DS is that data structure which allows the programmer to change its size during program execution.
Ex: linked list [Memory allocated Run time]

- ② According to occurrences:

- (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.

Ex: Stack, Queue, Array, linked list.

Non linear DS: Not sequential form not consecutive memory location.

Ex: 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 calculations 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 algorithm:

- ① Generality: An algo should solve a class of problem, not just one problem.
Ex.: an algo to calculate the average of few 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 Techniques:-

① Greedy Approach: This algo works in steps. In each each step it selects the best available option until all options finish. 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 soln of the sub-problems into a soln of the whole original problem.

③ Modular programming:

We divide the big problem into smaller ones which are totally different from each other. Then we combine the soln of all smaller problems and we get the soln of big problem. Here we can use different algo design for all small problems. This approach gives different module 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 soln. During the search if an alternative does not work, the search back tracks 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 divided into a main module & its related module. Each module ~~also~~ ^{else} should be divided into sub modules according to SW Bugg^g & programming style. This division of modules processes until the module can't be further sub divided.

On C the idea of top down is done using function. coding Prog^r ~~as~~ ^{stubs}.

⑥ Bottom up approach:

It is opposite of top down.

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. 5
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, until finally the main function is tested.

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

Ex: 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 ~~Destorying~~ Merging

Dynamic memory allocation:

```
int *p;
```

```
p = (int *) malloc (n * sizeof (int));
```

```
p = (int *) or 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 operations 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 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 affects 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 uncompress 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 .

Expt:- for linear search.

worst case : item at last
Best case : first
Average case : in middle

Abstract Data type:- (ADT)

ADT is a set of operations 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 referred as ADT. Also we are not concerned ~~about~~ with space & time efficiency.

Expt:- ① List - ADT

Component -

Item

Operations -

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.

Ex-2:

• STACK ADT -

Component -

Item

Operations

Push (insert item into stack from top)

Pop (Delete item from top)

We not concern how will stack implemented using array or linked-lists.

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

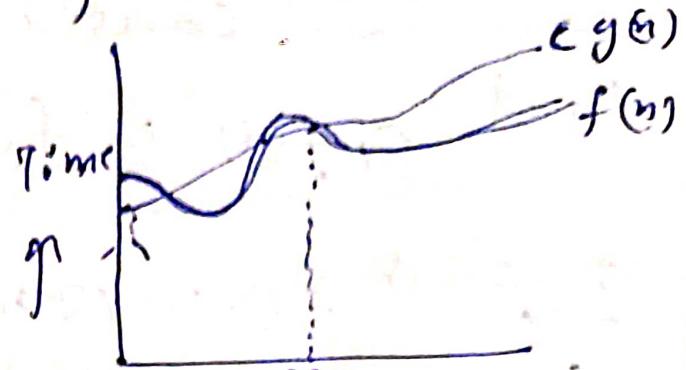
Asymptotic Analysis: performed on asymptotic curves.

Asymptotic means a line that tends to converge to a curve, for a function $f(n)$ as the value of n grows.

Asymptotic Notations 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 constant factor.

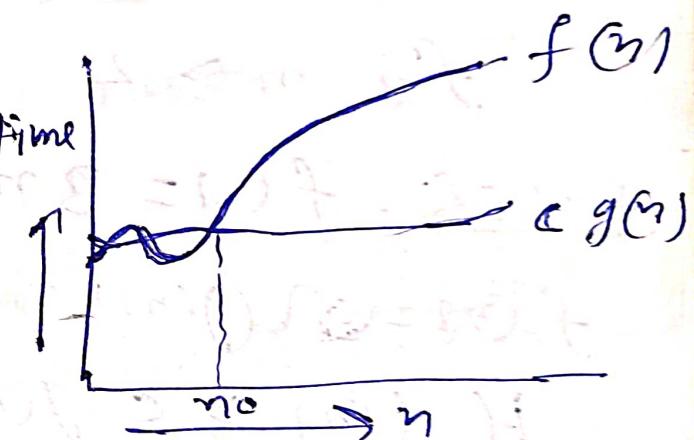
$f(n) = O(g(n))$ For a given function $g(n)$, we denote it by $O(g(n))$.



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

Ω -Notation :-

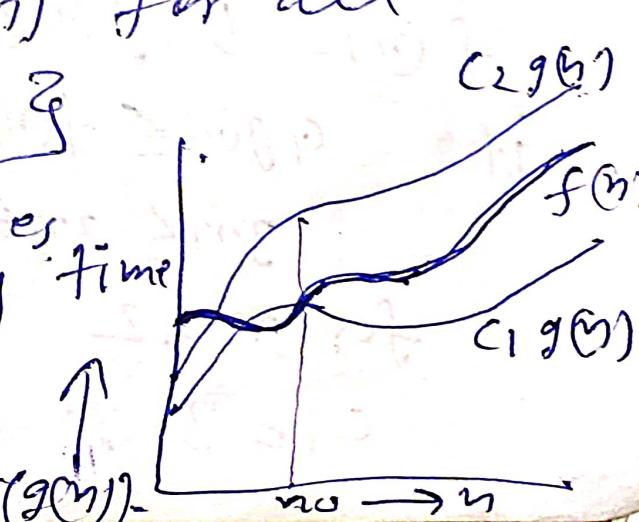
This notation gives lower bound of function. For a given function $g(n)$, we denote it by $\Omega(g(n))$.



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

Θ -notation :- It provides the lower and upper bound of the function.

For a given function $g(n)$, we denote it by $\Theta(g(n))$.



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

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

Expt:- ① $f(n) = 3n+2$

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

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

$$\begin{aligned} 3n+2 &\leq 4n \\ &\leq 3n+n \end{aligned}$$

for $n \geq 2$

Expt:- ② $f(n) = 3n+2$

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

If $f(n) \geq c g(n)$

$$3n+2 \geq 3n$$

for $n \geq 1$ $c \geq 3$

Expt:- 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 $n \geq 2$

$$c_1 = 3, c_2 = 4$$

Array

L10

Array: an array is finite, ordered and collection of homogeneous data elements.

Array is finite because it contains only limited no. of elements and ordered because the elements are stored in contiguous memory location in a linear order.

One-Dimensional array:

Array will be declared as

$a[20]$, where a is the name of array

& 20 is size of array.

$a[0] \dots a[19]$

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

$$n = UB - LB + 1 \quad a_n = 19 - 0 + 1 \\ = 20$$

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

Ex: $a[0], a[1], a[2], \dots, a[n]$

where a is name of array & $0, 1, 2, \dots, n$ 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 $\text{Base}(A)$.

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

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

for C, $L.B. = 0$

$$\text{Loc}(A[k]) = \text{Base}(A) + w * k$$

word size

Expt:- consider an array of A is declared as array of int with size 50 & first element is stored at memory address 100. So find the address of 5th element given $w=4$.

$$\text{Soln:- } \text{Loc}(A[5]) = \text{Base}(A) + w * k$$

$$\text{Loc}(A[5]) = 100 + 4 * 5 \\ = 120$$

Q2:- Consider the linear array

$A(S:50)$, $B(-5:10)$ and $C(10)$

11

(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)

$$\text{length of } A = LB - LB + 1 \\ = 50 - 5 + 1$$

$$L(A) = 46$$

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

$$L(C) = 16$$

$$L(C) = 10 - 1 + 1 \\ = 10$$

$$\begin{aligned} A(15) &= 300 + 4(15-5) \\ &= \cancel{300} \cancel{+ 4(15-5)} \\ A(40) &= 300 + 4(40-5) \\ &= 440 \\ A(55) &= 300 + 4(55-5) \\ &= \cancel{300} \cancel{+ 4(55-5)} \end{aligned}$$

Q3:- Consider an array A , which records the no. of TVs sold each year from 1975 to 2000.

Suppose the base address 101 and $w=4$ then

find the $\log A[1990]$.

$$\text{Base}(A) = 101$$



$$R = 1990 - 1975 = 15^{\text{th}}$$

$$\begin{aligned} \log A[15] &= \text{Base}(A) + w * R \\ &= 101 + 4 * 15 \\ &= 161 \end{aligned}$$

Algo:-

① Traversing algo
Let A be an array of n elements & k is used as subscript
Traverse (A, n)

- ① Set k = 0, while $k \leq n - 1$
- ② Repeat steps 3 & 4
- 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~~ i = n
- ③ Repeat steps 4 & 5 while $i \geq k$
- ④ Set ~~A[i+1] = A[i]~~ Set ~~A[i+1] = A[i]~~
Set $A[i+1] = A[i]$
- ⑤ Set $i = i - 1$
- ⑥ Set $A[k] = \text{item}$
- ⑦ Reset $n = n + 1$
- ⑧ Exit

Deletion from array:

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

(ii) 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 :-

array in memory :-

| R | C 1 | 2 | 3 |
|---|-----------------|-----------------|-----------------|
| 1 | a ₁₁ | a ₁₂ | a ₁₃ |
| 2 | a ₂₁ | a ₂₂ | a ₂₃ |
| 3 | a ₃₁ | a ₃₂ | a ₃₃ |
| 4 | a ₄₁ | a ₄₂ | a ₄₃ |

Row major order

| 1 | a ₁₁ | a ₁₂ | a ₁₃ |
|---|-----------------|-----------------|-----------------|
| 2 | a ₂₁ | a ₂₂ | a ₂₃ |
| 3 | a ₃₁ | a ₃₂ | a ₃₃ |
| 4 | a ₄₁ | a ₄₂ | a ₄₃ |

Column order major

| |
|-----------------|
| a ₁₁ |
| a ₁₂ |
| a ₁₃ |
| a ₂₁ |
| a ₂₂ |
| a ₂₃ |

| |
|-----------------|
| a ₁₁ |
| a ₂₁ |
| a ₃₁ |
| a ₄₁ |
| a ₂₂ |

Column major order :- Order of array = $M \times N$

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

SOL

| |
|-----------------|
| a ₁₁ |
| a ₂₁ |
| a ₃₁ |
| a ₄₁ |
| a ₅₁ |
| a ₆₁ |
| ⋮ |

Row major order :- Order of array = $M \times N$

| |
|-----------------|
| a ₁₁ |
| a ₁₂ |
| a ₁₃ |
| a ₂₁ |
| a ₂₂ |
| a ₂₃ |
| ⋮ |

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

w = word size

L₁ = Lower Bound of Row

L₂ = _____ 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 2000.

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

$M = 5, N = 6 \quad \& \quad w = 4, J = 5, K = 2$

Base Address = 200

Row-major order

$$\text{Loc}(A[5, 2]) = \text{Base Add} + w(N(J-L_1) + (K-L_2))$$

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

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

$$= 200 + 60$$

$$= \underline{\underline{260}}$$

Expt-2 :- $A[20][5], BA = 200 \quad \& \quad w = 4$

Find out the address of $A[12, 3]$ using

column-major order in C language.

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

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

$L_1 = L_2 = 0$ because C .

$$\text{Loc}(A[12, 3]) = 200 + 4[12 + 20 \times (3-0)]$$

$$= 200 + 4 \times 72$$

$$= 200 + 288$$

$$= \underline{\underline{488}}$$

Expt:-3 :- Array $A[2^0][5^0]$ required 4 bytes of storage for each element. $BA = 2000$. Find loc of $A[1^0][1^0]$ when the array is stored.

(i) Row-major

(ii) Column-major

(iii) Row-major

$$A[1^0][1^0] = \text{Base Add} + w[N(J-1) + (K-1)]$$

$$= 2000 + 4[5^0 \times (1^0 - 1) + (1^0 - 0)]$$

$$= 2000 + 4[5^0 + 1^0] + 005 =$$

$$= 2000 + 2040$$

$$\underline{\underline{= 2040}}$$

(ii) Column order

$$A[1^0, 1^0] = \text{Base Add} + w[(J-1) + M(K-1)]$$

$$= 2000 + 4[1^0 + 2^0(1^0 - 0)]$$

$$= 2000 + 840$$

$$\underline{\underline{= 2040}}$$

Dig

Three Dimensional Array:

14

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

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

Row-major Order:

$$\text{loc}(A[i][j][k]) = \text{BaseAdd} + w[(E_1 L_2 + E_2) L_3 + E_3]$$

Column-major order:

$$\text{loc}(A[i][j][k]) = \text{BaseAdd} + w[(E_3 L_2 + E_2) L_1 + E_1]$$

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.

Ex:- 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 $A(5, -1, 8)$.

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

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

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

$$= 400 + 2[142 + 3] \\ = \underline{\underline{830}}$$

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 :-

Ex:-

| 1 | 2 | 3 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 4 |
| 1 | 0 | 0 |

Many elements are zero.

Representation of Sparse matrix in memory:

① 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.

Expt:-

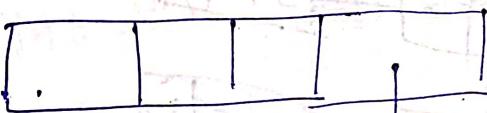
| Row | Column | Value |
|-----|--------|-------|
| 0 | 0 | 2 |
| 1 | 1 | 1 |
| 2 | 1 | 4 |
| 2 | 2 | 3 |
| 3 | 2 | 6 |

IN-C

Linked Representation of a Sparse matrix:-

For linked representation we use three type of node

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



No of non zero ↓
 Address of first dead node

iii) Value of elements
 (iv) Address of first Head node

(i) Address of first Head node

(ii) Head node: which contains Three field

(i) Row number (ii) pointed to next Node (iii)

Pointer to first non zero element of that row

(iii) Node: - which contains Three field

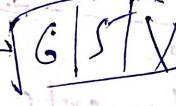
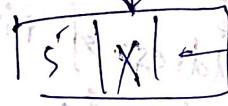
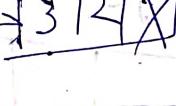
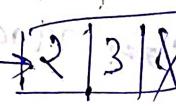
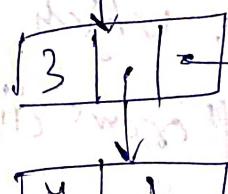
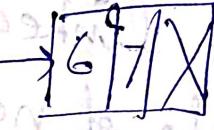
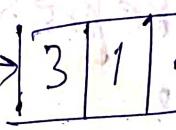
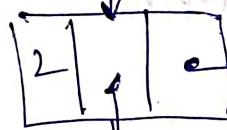
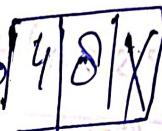
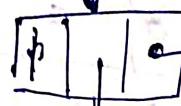
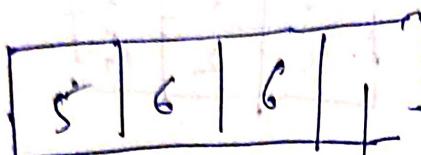
(i) contains column no. (ii) contains value

(iii), pointer to next non zero element of that row

Expt:-

$$\begin{bmatrix} 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 7 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

5×6



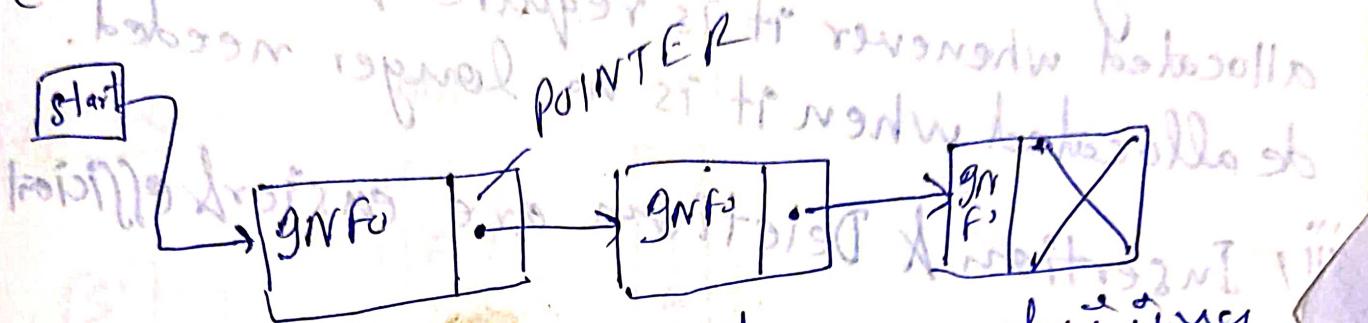
Traversing Two Dimensional Array

Column major order Algo:- Row major order Algo:-

- | | |
|----------------------------------|----------------------------------|
| ① For $J=0$ to $\text{column}-1$ | ① For $I=0$ to $\text{Row}-1$. |
| ② For $I=0$ to $\text{Row}-1$ | ② For $J=0$ to $\text{column}-1$ |
| ③ Display $A[I][J]$ | ③ Display $A[I][J]$ |
| ④ Stop | 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

(i) Data field (ii) Link field

* The link field of last node contains **NULL**.

Linked list is linear or Non-linear.

Linked list is 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 (using array)

Representation of linked list in memory:

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 \$ which contains the address of the first node in the list.

Ex:-

Exp:- 1

S1

gnfo

| | |
|---|----|
| 0 | 40 |
| 1 | |
| 2 | 20 |
| 3 | |
| 4 | 30 |
| 5 | |
| 6 | 35 |
| 7 | |

LINK

| |
|---|
| 6 |
| 4 |
| 0 |
| X |

L17

short bursts

full file transfer

long bursts

medium bursts

short file

medium file

long file

medium file

long file

medium file

long file

medium file

Exp:- 2 :-

S1

gnfo

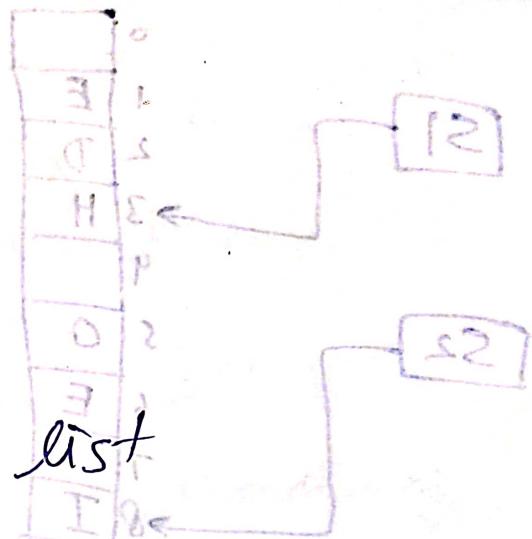
| | |
|----|---|
| 0 | |
| 1 | E |
| 2 | D |
| 3 | H |
| 4 | |
| 5 | O |
| 6 | E |
| 7 | |
| 8 | I |
| 9 | L |
| 10 | |
| 11 | A |
| 12 | L |
| 13 | |
| 14 | L |

LINK

| |
|----|
| 9 |
| 6 |
| 1 |
| X |
| 11 |
| 2 |
| 12 |
| 14 |
| 5 |

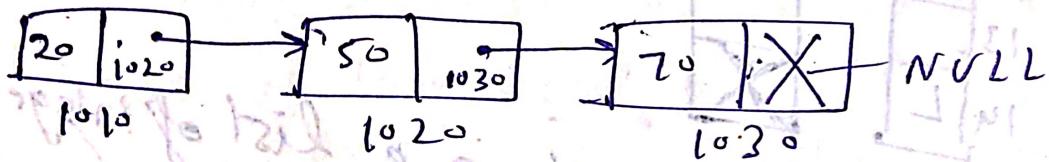
Types of linked list :-

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



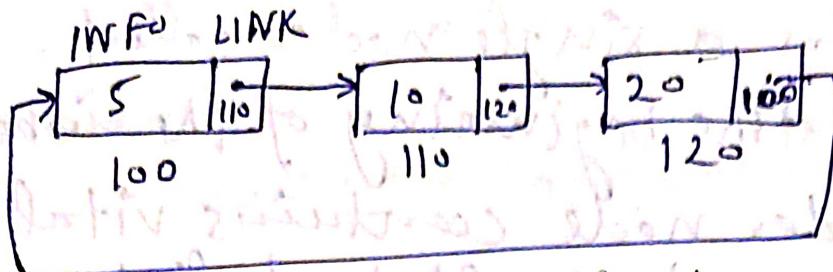
Linear linked list :-

INFO LIND



Each node is divided in two parts, first part contains the "info" of element and the second part contains the address of next node in list. Null pointer represents that there are no more nodes in the list.

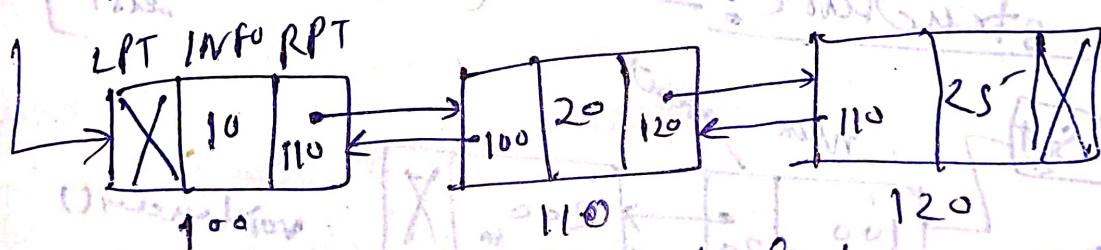
circular linked list :-



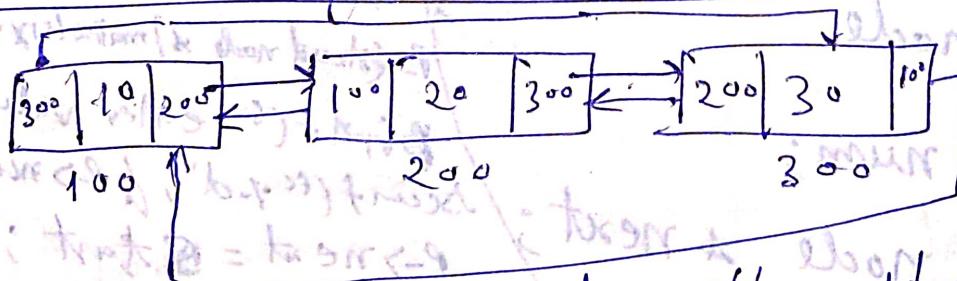
The second part of last node contains the address of first node.

Doubly linked list (two-way list) :-

Each node is divided in three parts, Info parts contains the information and 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.



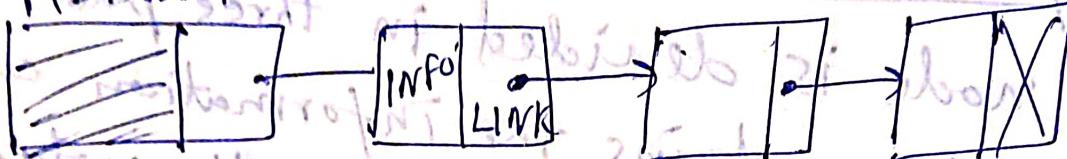
circular doubly linked list:



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.

HEADER

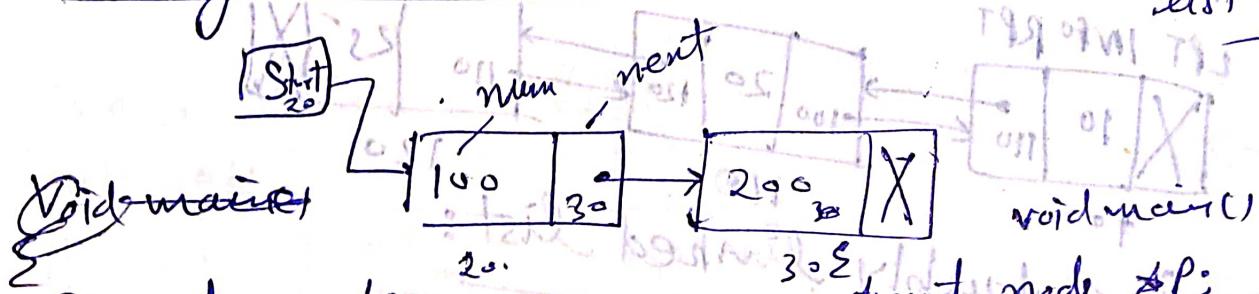


No data in
this node

Linear

Operations in Single Linked list

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



struct node

{ int num;

struct node * next;

}; struct node * start;

Struct node * Start=NULL;

Create linear linked list:-

struct node * p;

p=(struct node *)malloc(sizeof(struct node));

printf("enter value of node");

scanf("%d", &p->num);

p->next = start;

start = p;

~~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();

}

Operations in linear linked list

① Insertion

② Deletion

③ Traversing

Traversing in linear linked list

Algo:-

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

① If start=NULL then list is empty & exit.

② Set PTR=START

③ Repeat steps ③ & ④ while PTR ≠ NULL

③ Display PTR → INFO

④ Set PTR = PTR → NEXT

⑤ Exit

C-code :-

* Same for Doubly
linked list

Void Traverse()

{

struct node *ptr;

ptr = start;

if (start == NULL) prints ("list is empty");

while (ptr != NULL)

empty");

{

printf ("%d \t", ptr->info);

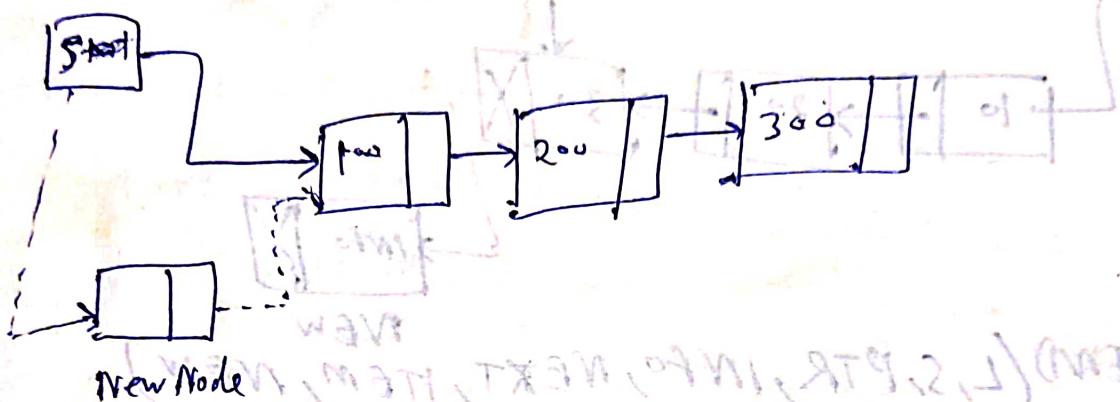
ptr = ptr → next;

}

Insertion in linear linked list:

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

(i) Insertion at beginning:



Algorithm:

INSERT BEG (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");
    NEW->info=ITEM;
    NEW->next=S;
    S=NEW;
}
```

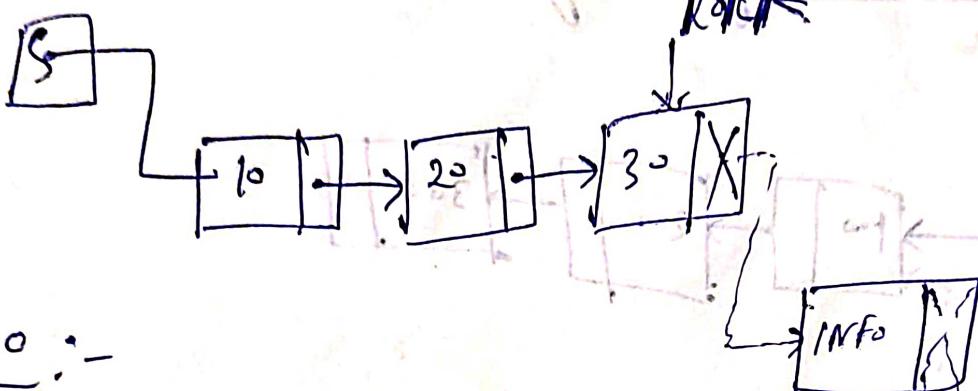
scanf("%d", & NEW->info);

NEW->next = start;

start = NEW;

3

(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* or *ITEM* is a string
- 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;
```

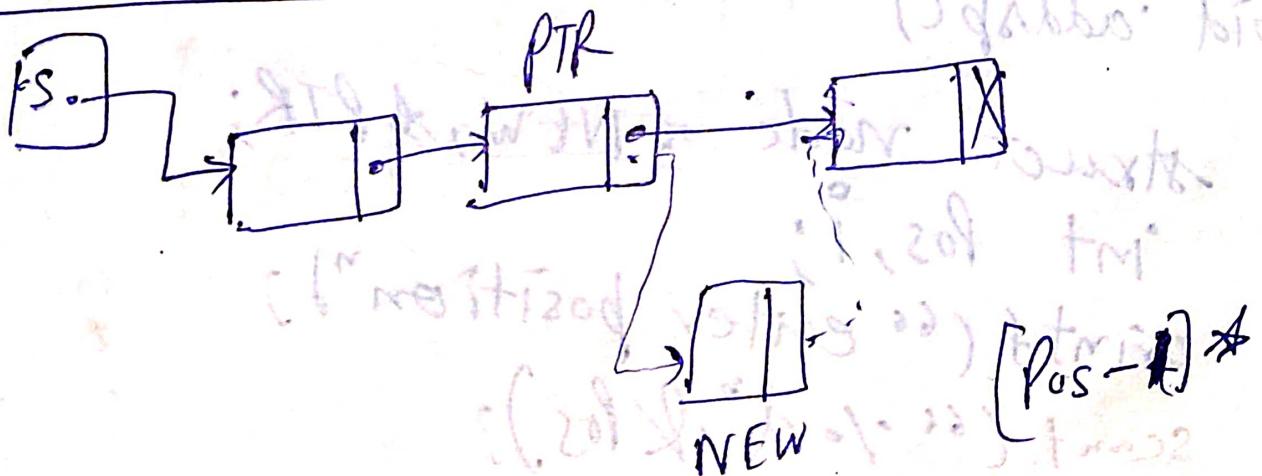
```
}
```

```
PTR->NEXT = NEW;
```

```
3
```

```
3
```

(iii) Insert at specific position:-



Algo:-

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

create a new node and name it NEW.

① Create a new node and name it NEW.

② If NEW=NULL then write Overflow & exit.

③ Set NEW \rightarrow INFO = ITEM

④ If POS=1 Then

NEW \rightarrow NEXT = ~~S~~

S = NEW "and exit"

⑤ Set I=1 and PTR=S

⑥ Repeat step 7 & 8 until I \leq (POS-1)

⑦ Set PTR = PTR \rightarrow NEXT & if PTR=NULL, display list is small & exit

⑧ Set I = I + 1

⑨ Set NEW \rightarrow NEXT = PTR \rightarrow NEXT

⑩ PTR \rightarrow NEXT = NEW

⑪ Exit.

C-Coding:-

Void addspc()

{

struct node *NEW, *PTR;

int pos, i;

printf("enter position"),

scanf("%d", &pos),

NB W = C strud node \rightarrow malloc (size of C strud node));
printf(" enter value of node"),
scanf("%d", & NEW \rightarrow INFO);
if (Pos == 1)

{

 NEW \rightarrow NEXT = S;

 S = NEW;

3

else

{ PTR = S;

 for (i=1; i<(Pos-1); i++)

 PTR = PTR \rightarrow NEXT;

 if (PTR == NULL)

 printf(" List is small");

 return exit();

3

3

 NEW \rightarrow NEXT = PTR \rightarrow NEXT;

 PTR \rightarrow NEXT = NEW;

3

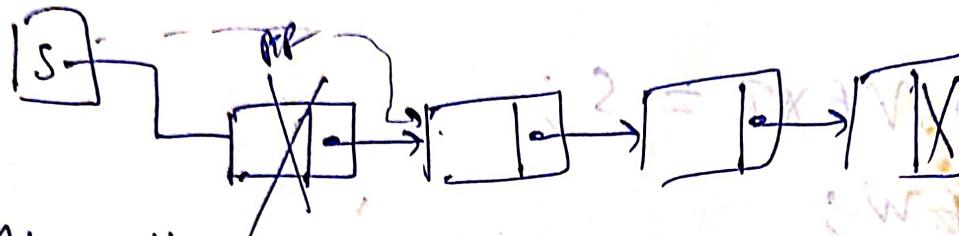
Deletion in linear linked list

(i) Deleting from the beginning

(ii) ~~Deleting from the end~~

(iii) ~~Deleting from specific position~~

(i) Deleting from the beginning:



Algorithm :-

① DELETEBEG (L, S, NEXT; PTR)

① If $S = \text{NULL}$ Then write ~~Overflow and underflow~~.

② Set $S = S \rightarrow \text{next}$ ③ $\text{PTR} = S$;

C-coding:- ④ $S = S \rightarrow \text{next}$;

⑤ $\text{free}(\text{PTR}) \Rightarrow \text{INFO}$

void deletebeg()

{ struct node *PTR; } ⑥ exit

If ($S == \text{NULL}$)

printf("Underflow");

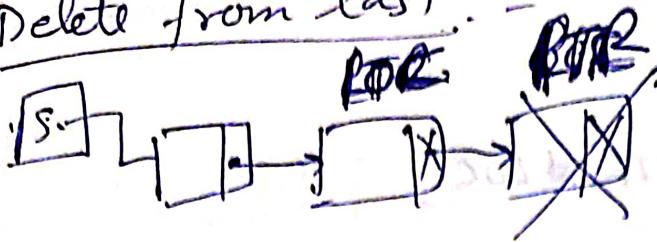
else {

② $\text{PTR} = S$; } printf("%d is deleted", S->info); }

$S = S \rightarrow \text{next}$;

③ $\text{free}(\text{PTR})$;

iii) Delete from last:-



Algo:-

DELETE (L, S, NEXT, PTR,)

(1) If $S = \text{NULL}$ then write under flow & exit.

(2) If $PTR = S$ and $\text{Loc} = \text{PTR} \rightarrow \text{NEXT}$

(3) If $\text{Loc} \neq \text{NULL}$ then $S = \text{NULL}$ and $\text{free}(\text{PTR})$, exit

(4) DELETE (L, S, PTR, Loc, INFO, NEXT)

If $S = \text{NULL}$ then write under flow and exit.

(1) If $S = \text{NULL}$ then write under flow and exit.

(2) If $S \rightarrow \text{Next} = \text{NULL}$ then

set $\text{PTR} = S = 1$ $\text{INFO} \leftarrow \text{RTS}$ $\text{NEXT} \leftarrow \text{RTS}$

Set $S = \text{NULL}$

print $\text{PTR} \rightarrow \text{INFO}$ and $\text{free}(\text{PTR})$ and Exit

(3) If $\text{PTR} \neq \text{S}$ then:

(4) Repeat steps 5 & 6 till $\text{PTR} \rightarrow \text{NEXT} = \text{NULL}$

(5) Set $\text{Loc} = \text{PTR}$

(6) Set $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$

(7) Set $\text{Loc} \rightarrow \text{NEXT} = \text{NULL}$

(8) Display $\text{PTR} \rightarrow \text{INFO}$

(9) $\text{free}(\text{PTR})$ (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;

JUN = 3

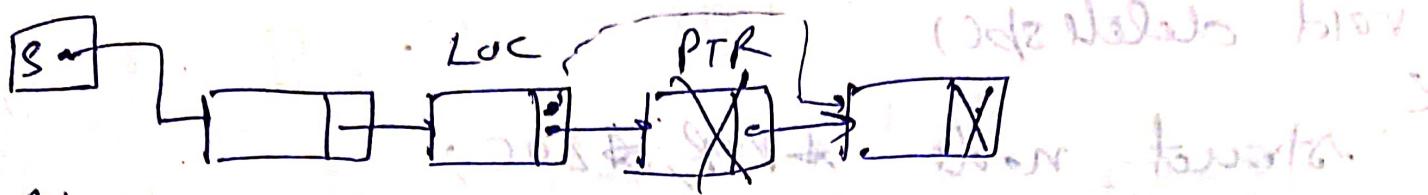
printf ("%d is deleted", PTR->INFO);

LOC->NEXT = NULL;

free (PTR)

3

(iii) Deletion from Specific position:



Algo:-

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

① If $S = \text{NULL}$ then write overflow & exit.

② If $POS = 1$ then

Set $PTR = S$

Set $S = S \rightarrow \text{NEXT}$

Display $PTR \rightarrow \text{INFO}$ and exit

③ $PTR = S$ and $I \neq 1$ $I = 1$

④ Repeat steps 5, 6, & 7 till $I < POS$

⑤ Set $LOC = PTR$

⑥ Set $PTR = PTR \rightarrow \text{NEXT}$ if $PTR = \text{NULL}$ write list is small & exit.

⑦ $I = I + 1$

⑧ Set $LOC \rightarrow \text{NEXT} = PTR \rightarrow \text{NEXT}$

⑨ Display $PTR \rightarrow \text{INFO}$

⑩ free (PTR)

⑪ EXIT

C-coding:-

```
void delete(spc)
```

```
{ 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, LOC = PTR->NEXT;
```

```
S = S->NEXT;
```

```
printf("%d is deleted", PTR->data);
```

```
free(PTR);
```

```
else
```

```
{ for (i = 1; i < pos; i++)
```

```
{ LOC = PTR;
```

```
PTR = PTR->NEXT;
```

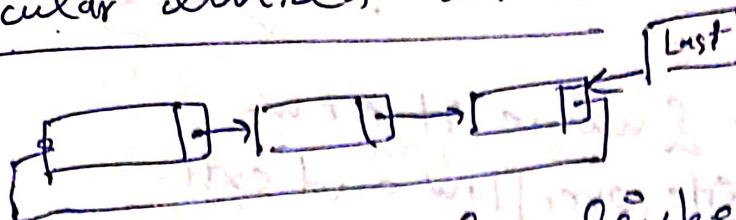
```
if (PTR == NULL) printf("List is small"); return;
```

```
LOC->NEXT = PTR->NEXT;
```

```
printf("%d is deleted", PTR->data);
```

```
free(PTR);
```

Circular linked lists



Insertion in circular linked list:

(i) Insertion at beginning or at end:

- ① Create a new node & name it NEW.
- ② If $NEW = \text{NULL}$ then write overflow and exit.
- ③ $NEW \rightarrow \text{INFO} = \text{ITEM}$.
- ④ If $\text{Last} = \text{NULL}$ then

$\text{Last} = \text{NEW}$

$\text{NEW} \rightarrow \text{NEXT} = \text{Last}$ and exit

$\text{TXAV} \leftarrow \text{tail} = \text{TXAV} \leftarrow \text{NEW}$

$\text{WAV} = \text{TXAV} \leftarrow \text{tail}$

- ⑤ $\text{NEW} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT}$

$\text{Last} \rightarrow \text{NEXT} = \text{NEW}$

$\text{TXAV} \leftarrow \text{tail} = \text{TXAV} \leftarrow \text{tail}$

$\text{WAV} = \text{TXAV} \leftarrow \text{tail}$

$\text{Last} = \text{NEW}$ * Insertion at first This line will not be written [skip this line]

$\text{Last} \rightarrow \text{NEXT} = \text{tail} = \text{tail} = \text{tail}$

C-coding:

Void addbeg()

{ 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=Last;

3

else { NEW->NEXT = Last->NEXT;

Last->NEXT = NEW;

3 Last=NEW; } // skip this line for insertion at start.

Insertion at Specific Position



INSERT SPEC

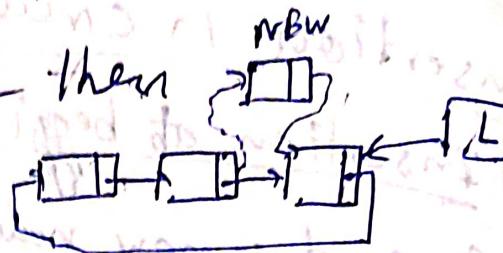
- Create a new node & name it NEW.

If NEW=NULL then write overflow and exit.

Set NEW->INFO = ITEM.

If pos=1 then Last=NULL then

If ~~Last=NULL~~ then



Last=NEW;

NEW->NEXT=Last and exit

If Pos=1 and Last!=NULL then

NEW->NEXT=Last->NEXT

Last->NEXT=NEW

~~Last=NEW~~ and exit

PTR=Last->NEXT and I=1

Repeat steps ⑧ & ⑨ until I<(Pos-1)

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

I=I+1

IF PTR=Last then

NEW->NEXT=PTR->NEXT

PTR->NEXT=NEW

Last=NEW and exit.

NEW->NEXT=PTR->NEXT

PTR->NEXT=NEW

Exit

C-Coding :-

void addSPC()

{ 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 = NEW; }

last->NEXT = NEW;

~~last = NEW;~~

else

{ PTR = last->NEXT;

~~for~~,

info(i=1; i < (pos-1); i++)

~~info~~,

~~for~~,

~~for~~,</

```

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 Deleting from start :-



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

- 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 $\text{Last} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$

⑥ $\text{free}(\text{PTR})$

⑦ Exit.

c-coding:-

```
void deletebeg()
```

{ struct node * PTR;

if (Last == NULL)

printf("Underflow");

else {

PTR = Last \rightarrow NEXT;

printf("%d is deleted", PTR \rightarrow INFO);

if (Last == Last \rightarrow NEXT)

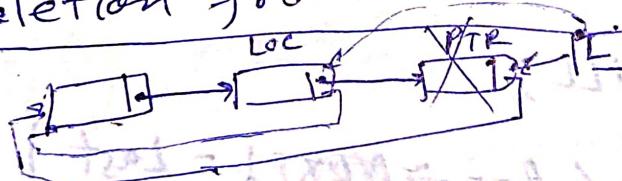
Last = NULL;

~~free(PTR);~~

else Last \rightarrow NEXT = PTR \rightarrow NEXT;

free(PTR);

② Deletion from end:



DELETE LAST :-

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

② PTR = Last and LOC = Last \rightarrow Next

③ Display PTR \rightarrow INFO is deleted.

4) If $\text{Last} = \text{Last} \rightarrow \text{NEXT}$ then
 $\text{Last} = \text{NULL}$, $\text{free}(\text{PTR})$ and exit..

(5) Repeat steps(6) until $\text{Loc} \rightarrow \text{NEXT} \neq \text{Last}$.

(6) $\text{Loc} = \text{Loc} \rightarrow \text{NEXT};$

(7) $\text{Loc} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT},$

(8) $\text{Last} = \text{Loc}.$

(9) $\text{free}(\text{PTR}).$

(10) Exit.

C-Coding:-

void deletedlast()

{ struct node *PTR, *LOC; }

if ($\text{Last} == \text{NULL}$)

printf("Underflow");

else

PTR = Last;

LOC = Last \rightarrow NEXT;

printf("-d is deleted ", PTR \rightarrow info);

if ($\text{Last} == \text{Last} \rightarrow \text{NEXT}$)

Last = NULL;

else

.while ($\text{Loc} \rightarrow \text{NEXT} \neq \text{Last}$)

$\text{Loc} = \text{Loc} \rightarrow \text{NEXT};$

$\text{Loc} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT};$

Last = LOC;

3. free(PTR);

- Deletion from specific positions :- Loc
-
- ① If $\text{Last} = \text{NULL}$ then write underflow and exit.
- ② $\text{PTR} = \text{Last} \rightarrow \text{NEXT}$.
- ③ if $\text{Last} = \text{Last} \rightarrow \text{NEXT}$ and $\text{Pos} = 1$ then
- ④ $\text{Last} = \text{NULL}$, $\text{Display}(\text{PTR} \rightarrow \text{INFO})$, $\text{free}(\text{PTR})$ & exit.
- ⑤ If $\text{Pos} = 1$ and $(\text{Last} \rightarrow \text{Last} \rightarrow \text{NEXT}) \neq \text{NULL}$
- $\text{Last} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$.
 Display ($\text{PTR} \rightarrow \text{INFO}$) and $\text{free}(\text{PTR})$ and exit.
- ⑥ $I = 1$ till $I \leq \text{Pos}$
- ⑦ Repeat steps 7 & 8 till $\text{PTR} = \text{Last} \rightarrow \text{NEXT}$
- ⑧ $\text{LOC} = \text{PTR}$ and if $(\text{PTR} = \text{Last} \rightarrow \text{NEXT})$ then
 write "List is small & zero".
- ⑨ $I = I + 1$
- ⑩ If $\text{PTR} = \text{Last}$ then
- $\text{Loc} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT}$
 $\text{Last} = \text{LOC}$
- ⑪ $\text{Display}(\text{PTR} \rightarrow \text{INFO})$ and $\text{free}(\text{PTR})$ & exit.
- ⑫ If $\text{Loc} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$
 Display $(\text{PTR} \rightarrow \text{INFO})$ and $\text{free}(\text{PTR})$ and exit.
- ⑬ Exit.

```
void deleteSPC()
{
    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 = NULL;
            free(PTR);
            return;
        }
        else if (Pos == 1 && Last != Last->NEXT)
        {
            Last->NEXT = PTR->NEXT;
            printf("%d is deleted", PTR->INFO);
            free(PTR);
            return;
        }
        else
        {
            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);
            }
        }
    }
}
```

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

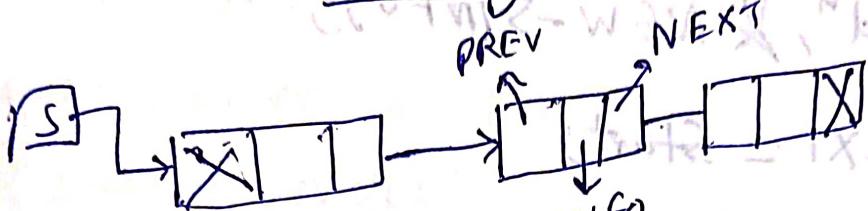
~~free(PTR); if (PTR == start)~~

~~start = loc;~~

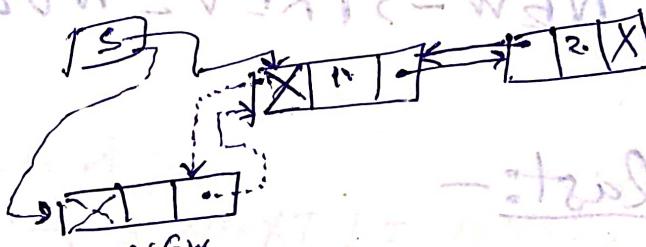
~~free(PTR);~~

~~3 3 3 (char,loc) (char,loc) (char,loc) = void~~

Doubly linked list :-



Insertion at beginning start :-



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=start
- ⑤ Set Start->PREV=NEW
- ⑥ start=NEW
- ⑦ NEW->PREV=NULL
- ⑧ Exit

C-coding:

Void oddfunc():

```

{ 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;

Now Start = NEW; *pointed to insertion*

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

Insert at last:-

INSERT LAST

- 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.
- 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.

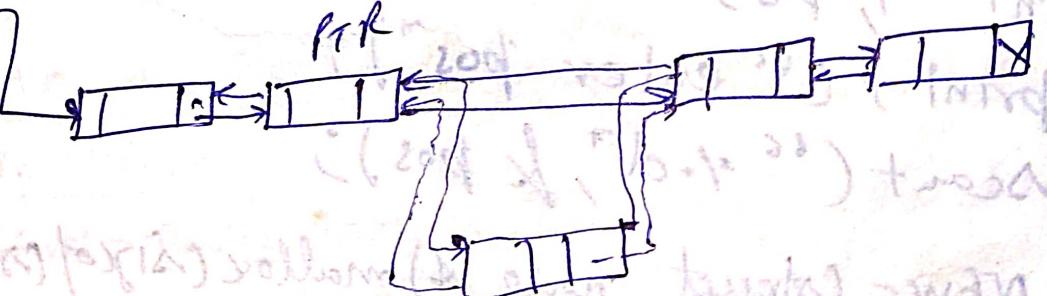
C-coding :-

* void addlast()

{
 Extract node ~~new~~ *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;
 NEW->PREV = NULL;
 }
 else
 {
 PTR = start;
 while (PTR->NEXT != NULL)
 PTR = PTR->NEXT;
 PTR->NEXT = NEW;
 NEW->PREV = PTR;
 }
}

Insertion at specific position

IS



NEW

INSERT SP

- Scm
if
- ① Create a new node & name of NEW
 - ② If $NEW = \text{NULL}$ then write overflow & exit.
 - ③ Set $NEW \rightarrow INFO = ITEM$ then show "Success".
 - ④ If $POS = 1$ and $Start = \text{NULL}$ then
 - Set $NEW \rightarrow NEXT = Start$
 - $Start \rightarrow PREV = NEW$
 - $Start = NEW$
 - $NEW \rightarrow PREV = \text{NULL}$ and exit.
 - ⑤ Set $I = 1$ and $PTR = Start$
 - ⑥ Repeat step 7 & 8 till $I < POS - 1$ (if $INFO == \text{NULL}$ then list is small and exit).
 - ⑦ Set $PTR = PTR \rightarrow NEXT$ and If ($PTR = \text{NULL}$) then exit.

$$⑨ I = I + 1.$$

$$⑩ NEW \rightarrow NEXT = PTR \rightarrow NEXT.$$

$$⑪ NEW \rightarrow PREV = PTR$$

$$⑫ PTR \rightarrow NEXT \rightarrow PREV = NEW$$

$$⑬ PTR \rightarrow NEXT = NEW$$

⑭ Exit

C-coding:-

void addspc()

E

Struct node *NEW, *PTR;

int i, pos;

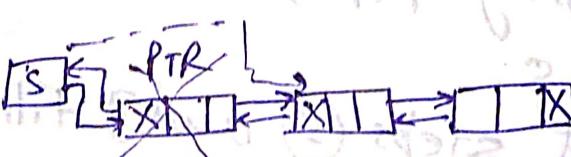
printf("enter pos");

scanf("%d", &pos);

NEW = (struct node *) malloc (size of struct node));

printf("enter value of node");

Scanf ("< .c > ", & NEW->INFO);
 if (pos == 1)
 {
 NEW->NEXT = start;
 Start->PREV = NEW;
 Start = NEW;
 NEW->PREV = NULL; ~~and exit~~
 3
 } else
 {
 PTR = start;
 for (i = 0; i < (pos - 2); i++)
 {
 PTR = PTR->NEXT;
 if (PTR == NULL)
 printf ("list is small");
 return;
 }
 3
 NEW->NEXT = PTR->NEXT;
 NEW->PREV = PTR;
 PTR->NEXT = NEW; ~~PTR->PREV = NEW;~~
 PTR->NEXT = NEW; ~~if MN = TX3N < 2~~ ①
 PTR->NEXT = NEW; ~~if MN = TX3N < 2~~ ②
 3
 }

Delete at start: 

- DELETEBEGIN:-
 ① If start = NULL then write underflow and exit.
 ② PTR = Start; ~~if PTR->NEXT = NULL then scan NULL~~
 ③ ~~Display PTR->INFO, free (PTR), and exit.~~
 ④ Start = Start->NEXT

- ④ Display $\text{PTR} \rightarrow \text{INFO}$.
 ⑤ $\text{start} \rightarrow \text{PREV} = \text{NULL}$.
 ⑥ $\text{start} \rightarrow \text{free}(\text{PTR})$.
 ⑦ Exit.

c-coding:

void deletedStart()

{ struct node *PTR;

if($\text{start} == \text{NULL}$)

printf("Under flow");

else

 PTR = Start;

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

~~Start = Start->NEXT;~~

~~Start->PREV = NULL;~~

~~free(PTR);~~

3

Delete from last: -

S

Loc

PTR

- ① If $S = \text{NULL}$ then write underflow/exit.
- ② If $S \rightarrow \text{NEXT} = \text{NULL}$ Then

Set $\text{PTR} = S$;

$S = \text{NULL}$,

Display $\text{PTR} \rightarrow \text{INFO}$, $\text{free}(\text{PTR})$ and exit.

- ③ $\text{PTR} = S$;
- ④ Repeat steps ⑤ and ⑥ till $\text{PTR} \rightarrow \text{NEXT} != \text{NULL}$
- ⑤ Set $\text{LOC} = \text{PTR}$.
- ⑥ $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$.
- ⑦ Set $\text{LOC} \rightarrow \text{NEXT} = \text{NULL}$.

- ⑧ Display PTR \rightarrow INFO.
- ⑨ free(PTR).
- ⑩ Exit.

C-coding:-

void deletelast()

{ struct L_node *PTR, *LOC; S = I; loc = 2; }

if (S == NULL)

printf("underflow");

else if (S \rightarrow NEXT == NULL)

PTR = S;

S = NULL;

printf("%d is deleted", PTR \rightarrow INFO);

free(PTR);

3 V389 \leftarrow AT9 = V389 \leftarrow TX3N \leftarrow AT9

else {

PTR = S;

LOC = PTR;

while (PTR \rightarrow NEXT != NULL)

LOC = PTR;

PTR = PTR \rightarrow NEXT;

3 : 301 \leftarrow AT9; over loops

LOC \rightarrow NEXT = NULL; over loops

printf("%d is deleted", PTR \rightarrow INFO);

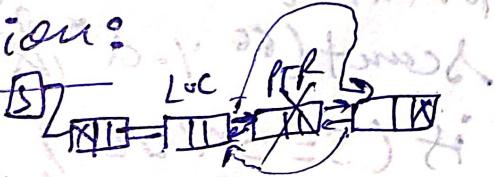
free(PTR);

3 (position of writing) HTR

Delete from specific Position:

DELETE SP

① If S = NULL write underflow & exit



② If $\text{pos} = 1$ then

Set $\text{PTR} = S$

Set $S = S \rightarrow \text{NEXT}$

$S \rightarrow \text{PREV} = \text{NULL}$

Display $\text{PTR} \rightarrow \text{INFO}$, $\text{free}(\text{PTR})$ & exit.

③ $\text{PTR} = S$ and $I = 0$

④ Repeat steps ⑤, ⑥ and ⑦ till $I < \text{pos} - 1$

⑤ Set $\text{LOC} = \text{PTR}$

⑥ $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$ (If $(\text{PTR} == \text{NULL})$ then write list is small and exit.

⑦ $I = I + 1$

⑧ Display $\text{PTR} \rightarrow \text{INFO}$

⑨ $\text{LOC} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$

⑩ $\text{PTR} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{PTR} \rightarrow \text{PREV}$

⑪ $\text{free}(\text{PTR})$.

⑫ Exit.

C-coding :-

void deleteS() {

{

struct node *PTR, *LOC;

int i, pos;

if (~~if (S == NULL)~~)

~~printf("Underflow")~~

else

~~print~~

printf(" enter position");

scanf("%d", &pos);

if ($S == \text{NULL}$)

printf(" Underflow");

else if ($\text{pos} == 1$)

$\text{PTR} = \text{S}$;

$\text{S} = \text{S} \rightarrow \text{NEXT}$;

$\text{S} \rightarrow \text{PREV} = \text{NULL}$;

 printf("-%d is deleted", PTR->INFO); ; trail - 371
 free(PTR); ; (JUN = 1 AT 7) + 610

 3

else {

 PTR = S;

 for (P = 1; i < pos; i++)

 LOC = PTR;

 PTR = PTR->NEXT;

 if (PTR == NULL)

 E printf("list is small");

 return;

 3 \rightarrow traverse to next node
 to link 3. If every time from word to word (1)

 LOC->NEXT = PTR->NEXT;

 PTR->NEXT->PREV = PTR->PREV;

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

 free(PTR);

 3

3 Traversing (Displaying) the Doubly linked list:

TRAVERSE

(1) Set $\text{PTR} = \text{Start}$;

(2) Repeat step (3) & (4) while $\text{PTR} \neq \text{NULL}$:

(3) Display $\text{PTR} \rightarrow \text{INFO}$

(4) $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$.

(5) Exit.

C-coding :-

void traverse()

{ struct node *PTR;

PTR = Start;

while (PTR != NULL)

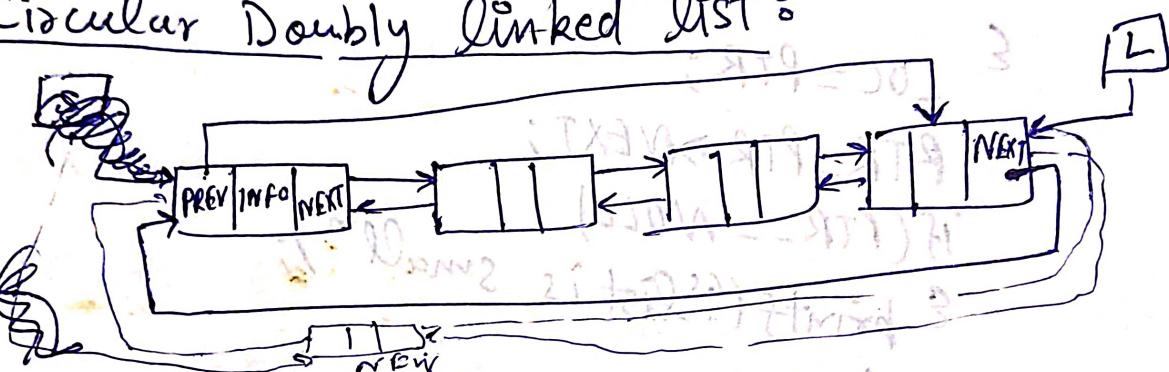
{ printf("%d \t", PTR->INFO); }

PTR = PTR->NEXT;

}

3

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;

~~S = 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.

c-coding :-

```

void add_start()
{
    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 If $LAST = NULL$ then

$$LAST = NEW$$

$$NEW->NEXT = NEW$$

$$NEW->PREV = NEW \text{ and exit}$$

$$(5) \quad NEW->NEXT = LAST->NEXT$$

$$(6) \quad NEW->PREV = LAST$$

- ⑦ ~~Last->NEXT~~ Last->NEXT \rightarrow PREV = NEW
- ⑧ L \rightarrow 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 \rightarrow PREV = NEW

Last->NEXT = NEW;

Last = NEW;

}

3

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.

- Date _____
Page No. _____
- ④ If $\text{Pos} = 1$ and $\text{Last} = \text{NULL}$ then
 $\text{Last} = \text{NEW};$
 $\text{NEW} \rightarrow \text{NEXT} = \text{NEW};$
 $\text{NEW} \rightarrow \text{PREV} = \text{NULL}$ and exit.
- ⑤ If $\text{Pos} = 1$ and $\text{Last} \neq \text{NULL}$ then
 $\text{NEW} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT};$
 $\text{NEW} \rightarrow \text{PREV} = \text{Last};$
 $\text{Last} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{NEW}.$
 $\text{Last} \rightarrow \text{NEXT} = \text{NEW}$ and exit.
- ⑥ $\text{PTR} = \text{Last} \rightarrow \text{NEXT}$ and $I = 0$
- ⑦ Repeat step ⑧ & ⑨ till $I < (\text{Pos} - 1)$
- ⑧ $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$ if $\text{PTR} = \text{NULL}$ then write list
is small and exit.
- ⑨ $I = I + 1.$
- ⑩ If ~~$\text{PTR} = \text{Last}$~~ then
 ~~$\text{NEW} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$~~
 ~~$\text{NEW} \rightarrow \text{PREV} = \text{Last}$~~
 ~~$\text{Last} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{NEW}$~~
 ~~$\text{Last} \rightarrow \text{NEXT} = \text{NEW}$~~
 ~~$\text{Last} = \text{NEW}$ and exit.~~
- ⑪ $\text{NEW} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}.$
- ⑫ ~~$\text{PTR} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{NEW}$~~
- ⑬ ~~$\text{PTR} \rightarrow \text{NEXT} = \text{NEW}$~~
- ⑭ ~~$\text{NEW} \rightarrow \text{PREV} = \text{PTR}$~~
- ⑮ ~~If $\text{PTR} = \text{last}$ then $\text{Last} = \text{NEW}$,
exit~~

C - Coding :-

```
void addspc()
```

```
    {  
        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=NULL;  
        }  
        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;  
        }  
    }
```

If (PTR == last) {

 NEW \rightarrow NEXT = Last \rightarrow NEXT;
 NEW \rightarrow PREV = Last;
 Last \rightarrow NEXT \rightarrow PREV = NEW;
 Last \rightarrow NEXT = NEW;
~~Last \rightarrow NEW;~~
 if (bdr == last)
 last = NEW;
 else
 if (PTR == last)
 NEXT \rightarrow NEXT = PTR \rightarrow NEXT;
 PTR \rightarrow NEXT \rightarrow PREV = NEW;
 PTR \rightarrow NEXT = NEW;
 PTR \rightarrow PREV = PTR;

3

Deletion from Start:

- ① IF Last = NULL then write underflow and exit.
- ② PTR = Last \rightarrow NEXT.
- ③ Display (PTR \rightarrow INFO) is deleted.
- ④ If Last = Last \rightarrow NEXT then
 Last = NULL, free (PTR) and exit.
- ⑤ Last \rightarrow NEXT = PTR \rightarrow NEXT
- ⑥ Last \rightarrow NEXT \rightarrow 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 = 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

- 137
- ⑦ $\text{LOC} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT}$.
 - ⑧ $\text{Last} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{LOC}$.
 - ⑨ $\text{Last} = \text{LOC}$.
 - 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);
```

```
3 3 3
```

Deletion from specific position:

- C -
void
()
- ① If $\text{Last} = \text{NULL}$ then write underflow and exit.
 - ② $\text{PTR} = \text{Last} \rightarrow \text{NEXT}$.
 - ③ If $\text{Pos} = 1$ and $\text{Last} = \text{Last} \rightarrow \text{NEXT}$ then
 $\text{Last} = \text{NULL}$, $\text{Display}(\text{PTR} \rightarrow \text{INFO})$, $\text{free}(\text{PTR})$
 and exit.
 - ④ If $\text{Pos} = 1$ and $\text{Last} \neq \text{Last} \rightarrow \text{NEXT}$ then
 $\text{Last} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$.
 $\text{PTR} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{Last} \cdot \text{Display}(\text{PTR} \rightarrow \text{INFO})$
 $\text{free}(\text{PTR})$ and exit.
 - ⑤ $I = 0$
 - ⑥ Repeat steps ⑦ & ⑧ till $I < \text{Pos} - 1$.
 - ⑦ $\text{LOC} = \text{PTR}$.
 - ⑧ $\text{PTR} = \text{PTR} \rightarrow \text{NEXT}$ is ($\text{PTR} = \text{Last} \rightarrow \text{NEXT}$) then
 write list is small and exit.
 - ⑨ $I = I + 1$
 - ⑩ If $\text{PTR} = \text{Last}$ then
 $\text{LOC} \rightarrow \text{NEXT} = \text{Last} \rightarrow \text{NEXT}$.
 $\text{Last} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{LOC}$. $\text{Display}(\text{PTR} \rightarrow \text{INFO})$
 $\text{Last} = \text{LOC}$, $\text{free}(\text{PTR})$ and exit.
 - ⑪ $\text{LOC} \rightarrow \text{NEXT} = \text{PTR} \rightarrow \text{NEXT}$.
 - ⑫ $\text{PTR} \rightarrow \text{NEXT} \rightarrow \text{PREV} = \text{LOC}$
 - ⑬ $\text{Display}(\text{PTR} \rightarrow \text{INFO})$
 - ⑭ $\text{free}(\text{PTR})$
 - ⑮ exit.

C-Coding :-

void deleteSP()

{ struct node *PTR, *LOC;
 int i, pos;
 if (Last == NULL) printf("Underflow");
 else

{ PTR = Last -> NEXT;
 if (pos == 1 && Last == Last -> NEXT)

{ Last = NULL;
 printf("y-d is deleted", PTR->info);
 free(PTR);

{ else if (pos == 1 && Last != Last -> NEXT)

{ Last -> NEXT = PTR -> NEXT;
 PTR -> NEXT -> PREV = Last;
 printf("y-d is deleted", 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;

{ if (PTR == Last)

{ LOC -> NEXT = PostR -> NEXT;

PostR -> NEXT -> PREV = LOC;

Last = LOC;

printf("y-d is deleted", PTR->info);

free(PTR); if (PTR == Last)
 Last = LOC;

else

{ LOC -> NEXT = PTR -> NEXT;

PTR -> NEXT -> PREV = LOC;

printf("y-d is deleted", PTR->info);

free(PTR);

3 3 3

3 3

3 3

Traversing of circular Doubly linked lists-

- ① Set PTR = Last \rightarrow NEXT; If Last = NULL then list is empty & exit.
- ② Set PTR = Last \rightarrow NEXT.
- ③ If Last \neq PTR = Last \rightarrow NEXT
- ④ If last = Last \rightarrow NEXT. Then
- ⑤ Display (PTR \rightarrow INFO) and Exit.
- ⑥ PTR = PTR \rightarrow NEXT.
- ⑦ Repeat Steps ⑥ & ⑦ while PTR \neq last \rightarrow NEXT.
- ⑧ Display PTR \rightarrow INFO
- ⑨ Set PTR = PTR \rightarrow NEXT
- ⑩ Exit.

C-coding:-

```
void traverse()
```

* Same for circular
linked list

```
struct node *PTR;  
if (Last == NULL)  
    prints ("list is empty");  
else  
    {  
        PTR = Last  $\rightarrow$  NEXT;  
        prints ("%d\t", PTR  $\rightarrow$  INFO);  
        if (Last == Last  $\rightarrow$  NEXT)  
            {  
                prints ("%d\t", PTR  $\rightarrow$  INFO);  
            }  
        else  
            {  
                PTR = PTR  $\rightarrow$  NEXT;  
                while (PTR  $\neq$  last  $\rightarrow$  NEXT)  
                    {  
                        prints ("%d\t", PTR  $\rightarrow$  INFO);  
                        PTR = PTR  $\rightarrow$  NEXT;  
                    }  
            }  
    }
```

Polynomial Representation :-

Linked list is used to represent polynomial expressions. 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.

| coeff | Expo | Link |
|-------|------|------|
|-------|------|------|

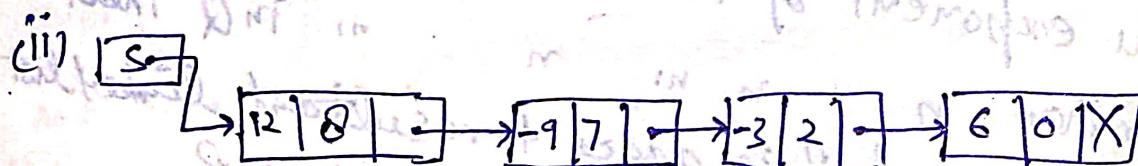
Exp:- Represent the polynomial

$$P(x) = 12x^8 - 9x^7 - 3x^2 + 6 \quad (\text{using Array (ii) linked list})$$

ans:- (ii) will requires three linear arrays which will

call COEF, EXPO, LINK and is represented as:

| COEF | EXPO | LINK |
|------|------|------|
| 0 | | |
| 2 | | |
| 12 | 8 | 4 |
| -9 | 7 | 6 |
| -3 | 2 | 9 |
| 6 | 0 | X |



struct node

```

< int coeff;
  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 P.
- (iii), Read the number of terms in the second polynomial Q.
- (iv) Read the coefficients and exponents of the second polynomial Q.

⑤ Set the temporary pointers p and q^{in start} to traverse the two polynomials P and Q respectively.

⑥ compare the exponents of two polynomials [starting from the first node] pointed by p & q.

⑦ if both exponents are equal then add the

coefficients and store it in the resultant linked list.
and move p & q to next node.

⑧ 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.

⑨ if the exponent of the current term in P is

greater than n₁ n₂ n₃ n₄ n₅ n₆ n₇ n₈ n₉ in Q then

current term of P is added to resultant linked list

and move the pointer p to next node of P.

⑩ repeat step 6 until one list is empty.

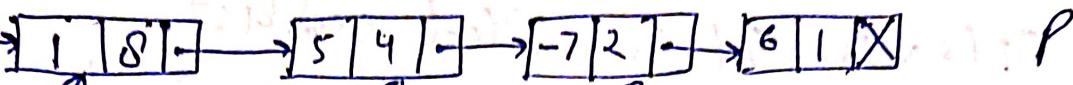
⑪ Append the remaining nodes of either of the polynomials to the resultant linked list.

Exps-

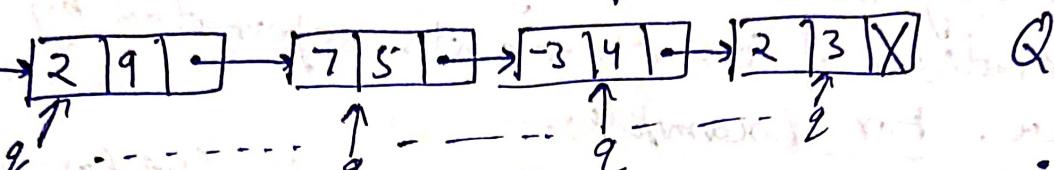
$$P(x) = x^8 + 5x^4 - 7x^2 + 6x$$

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

(S1)

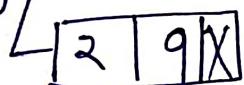


(S2)



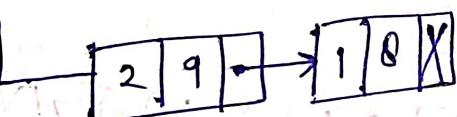
Comps exponent of q is greater than p so q is added to resultant linked list and move q to next node.

(R)



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

(R)



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

(S)



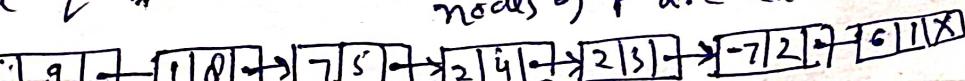
exponent of p = exponent of q so coefficients of both are added and stored in resultant list & move p & q to next.

(S)



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

(S)



n Generalised Linked list :-

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

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

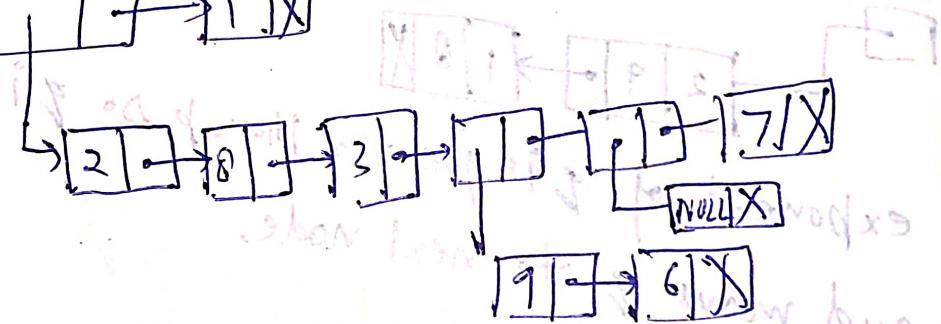
When a_i is a list, it is called sublist.

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

1. $A = ()$ represents null or empty list having 0 length.

2. $A = (2, (3, 4), 8)$ list having length 3, there are two atomic value node and a list having two elements.

Expt 2 $A = (5; (2, 8, 3, (9, 6)), (1, 7), 1)$

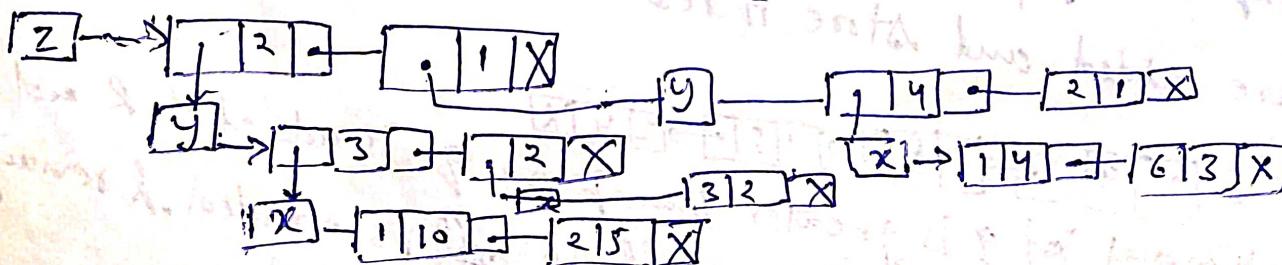


Expt 2

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

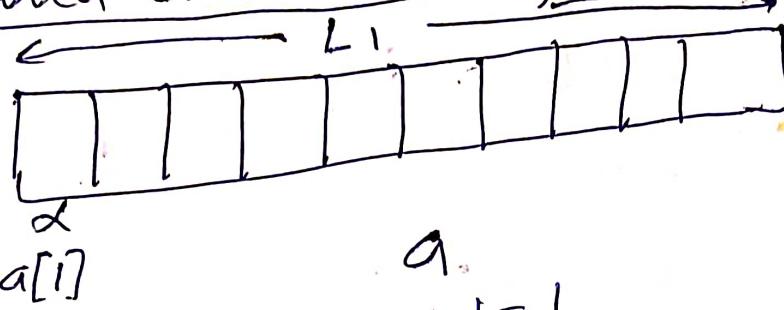
$$P(x, y, z) = (x^{10}y^3 + 2x^5y^3 + 3x^2y^2)z^2 + (x^4y^4 + 6x^3y^4 + 2y)z$$

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



(41)

Formula derivation for 1-D array:



Let Base = α , $w = 1$

$$\text{Add}(A[1]) = \alpha$$

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

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

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

$$\vdots$$

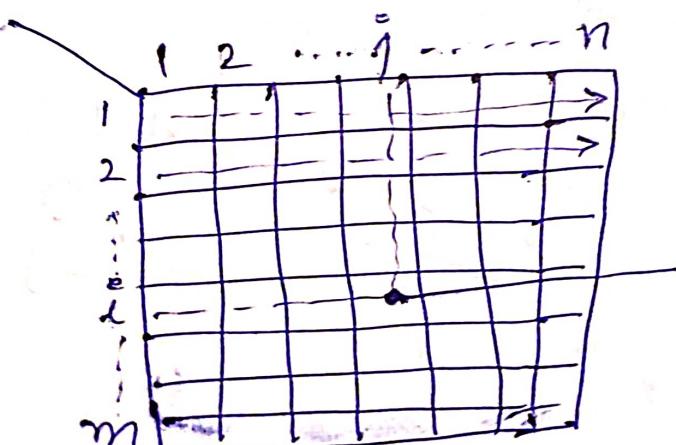
$$\text{Add}(A[k]) = \alpha + (k-1)$$

for word size w .

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

$$\text{Add}(A[k]) = BA + w(E_1)$$

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



$A[i][j]$

Let Base Add $\hat{=} \alpha$, $w=1$

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

$$\text{Add of } (A[1][2]) = \alpha + 1$$

$$\text{Add of } (A[1][3]) = \alpha + 2$$

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

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

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

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

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

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

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

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

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

$$\cancel{A[i][1]} = \alpha + \cancel{(i-1)n}$$

$$\cancel{A[i][2]} = \alpha + \cancel{i * n + 1}$$

$$A[i][1] = \alpha + \cancel{(i-1)n}$$

$$A[i][2] = \alpha + \cancel{(i-1)n + 1}$$

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

$$A[i][j] = \alpha + \cancel{(i-1)n} + \cancel{(j-1)}$$

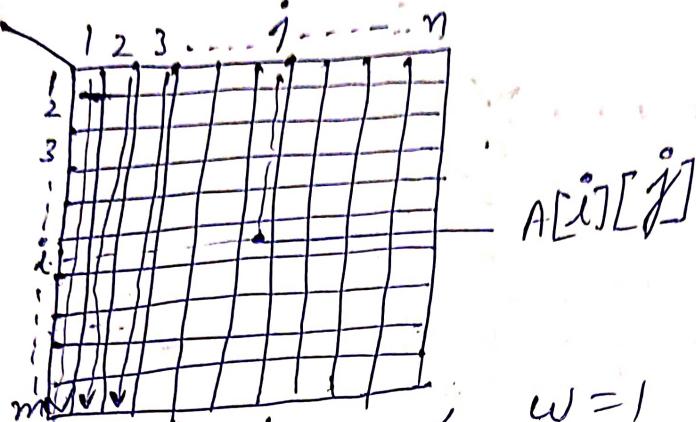
$$A[i][j] = \alpha + \cancel{(i-1)n} + (j-1)$$

$$\text{Add of } (A[i][j]) = \beta A + w [E_1 * L_2 + E_2]$$

Formula derivation of 2-D Arrays :-

(Column major order)

(42)



Let Base Address = α , $w = 1$

$$\text{Address of } A[0][1] = \alpha$$

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

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

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

$$A[0][2] = \alpha + m$$

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

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

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

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

$$A[0][3] = \alpha + 2m$$

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

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

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

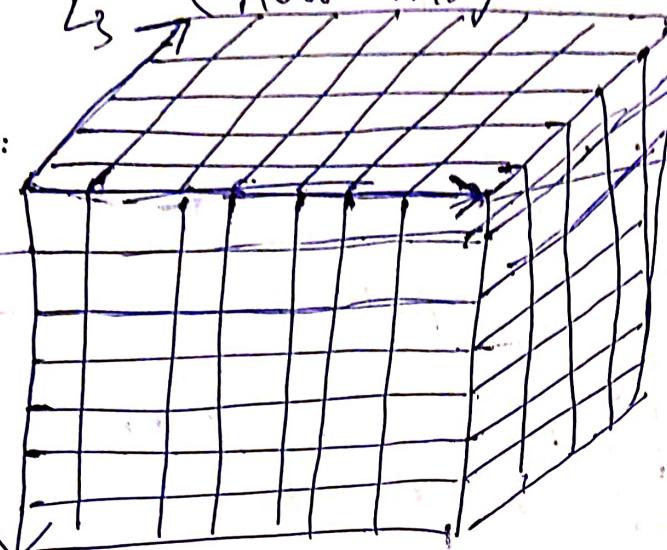
$$A[4][3] = \alpha + (j-1) * m + 3$$

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

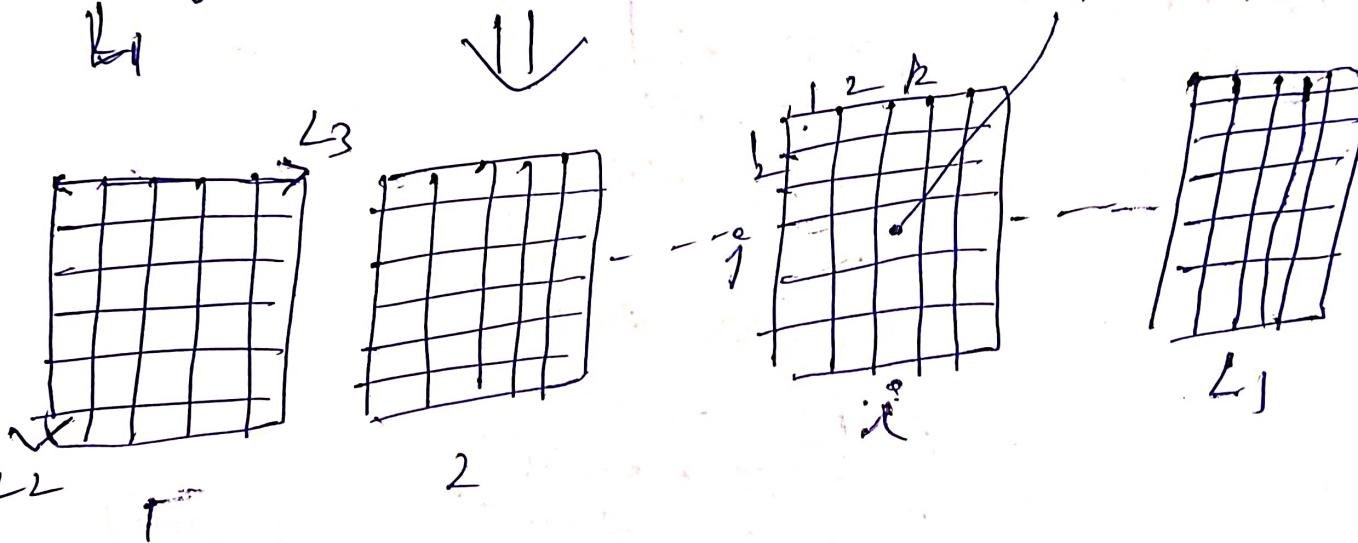
$$\text{Add of } A[i][j] = \beta A + w [E_2 l_1 + E_1]$$

formula derivation for 3-D Array :

(Row major order)



$A[i][j][k]$



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

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

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

$$A[4][1][1] = \alpha + 3L_2 L_3$$

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

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

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

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

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

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

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

(43)

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

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

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

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

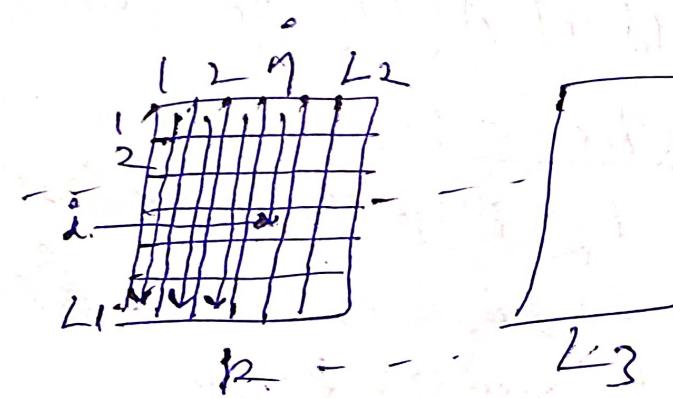
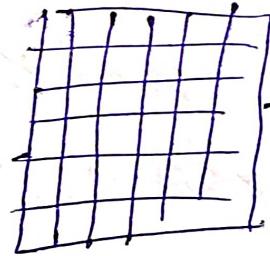
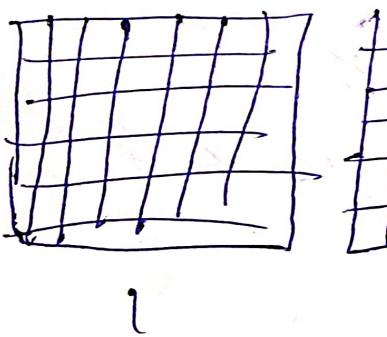
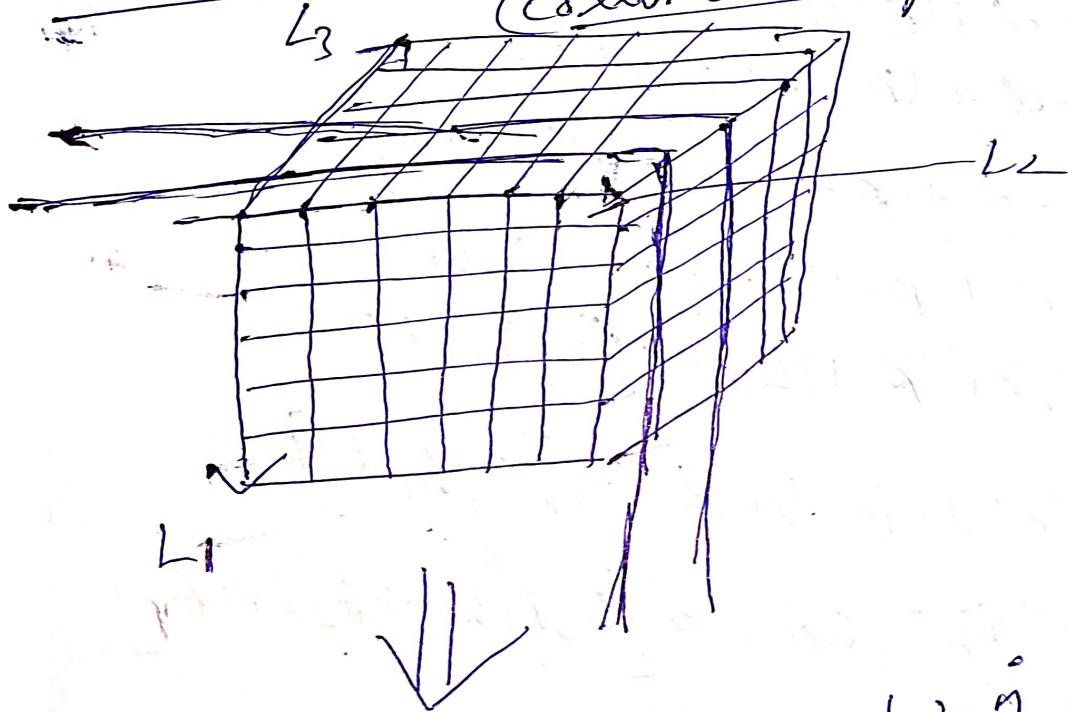
$$A(i)(j)(3) = \alpha + (i-1)L_2L_3 + (j-1)L_3 + 2$$

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

$$A(i)(j)(k) = \alpha + (i-1)L_2L_3 + (j-1)L_3 + E_1L_2L_3 + E_2L_3 + E_3$$

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\text{Add of } A[i][j][k] = BA + w [E_3 L_1 L_2 + E_2 L_1 + E_1]$$

where :-

$$E_R \text{ Effective index} = k - LB$$

L_1, L_2, L_3 are lengths of 1, 2 and 3 dimensions or say of array.

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:

$$\text{Add}[A[i][j]] = BA + w[E_1 l_2 + E_2]$$

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

where lb_1 = lower bound 1 = 3

lb_2 = lower bound 2 = -1

$$l_1 = 7-3+1 = 5$$

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

$$\begin{aligned} \text{So, Add}(A[5,2]) &= BA + w[(i-lb_1)l_2 + (j-lb_2)] \\ &= 200 + 4[(5-3)*6 + 2 - (-1)] \\ &= 200 + 4[12 + 3] \\ &= \underline{\underline{260}} \end{aligned}$$

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

Ans:

$$\text{Add}[A[i][j]] = BA + w[E_2 l_1 + E_1]$$

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

where

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

$$\begin{aligned}\text{Add } A[12,3] &= 200 + 4[(3-0)*20 + (12-0)] \\ &= 200 + 4[60 + 12] \\ &= 200 + 288 \\ &= \underline{\underline{488}}\end{aligned}$$

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) column major order.

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

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

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

$$\begin{aligned}\text{Add}(A[10][10]) &= 2000 + 4[(10-0)*50 + (10-0)] \\ &= 2000 + 2040 \\ &= \underline{\underline{4040}}\end{aligned}$$

$$\begin{aligned}\text{Add}[A(i)[j)]) &= \overline{BA + w[E_2 l_1 + E_1]} \quad \text{column major} \\ &= 2000 + 4[(j-lb_2)l_1 + (i-lb_1)] \\ &= 2000 + 4[(10-0)*20 + 10] \\ &= 2000 + 840 \\ &= \underline{\underline{2840}}\end{aligned}$$