Unit 1 :
Introduction to Data Structure

* The data structure is the term used to describe the way data is stored and the term algorithm is used to describe the way the data is processed.
- Algorithm + Data Structure = Program

* Data type and Data Object :-

Data :- Data refers to the value or value set. Data is divided into atomic or non-atomic data.

* Atomic Data :-
It is a single or non-decomposable entity. For ex integer.
* Non-Atomic Data (Composite Data) :-
A data that can be broken into sub-field that have meaning is called composite data. For ex date i.e. dd-mm-yyyy.

* Data Types :-

Data types in C

- User defined → structure, union, enum
- Built-in → int, char, float, double, void, etc.
- Derived → Array, Function, Pointer

* User - defined Data types :-

1) Structure :-

Structure is a tool for packing together logically related data items of different types. It is heterogeneous.

For ex,
```
struct employee
{
char emp-name[20];
int emp-id;
float basic;
};
```

2) Union :-

Union are very much similar to structures where both are used to group no. of different element only the difference is the member of union share same storage area even though the individual element may different in types.

For ex,
```
union Id
{
char color[12];
int size;
};
```

3) Enum (Enumeration) :-

Enumeration type consist of integral constant. For ex,
enum demo { const 1, const 2, ..., const n }

* Derived Data types :-

1) Array :-
Assign a single name to whole group of elements.

2) Pointer :-
Pointer is a memory location that holds memory address.

** * Data Object :-
Data Object represent container for data values (a place where data values maybe stored and later retrieved).
Data Object is a set of attribute.
For ex., data object alphabet is defined as
$D = \{ A, B, C, \ldots, Z \}$

** * Abstract Data type [ADT] :-
An abstract data type (ADT) is a data declaration package together with the operation that are meaningful for the data type.
ADT include :-
i) Domain :- Collection of data
ii) Functions :- Set of operations on the data
iii) Axioms :- A set of rules of behaviour governing the interaction of operations (protocols).

- Advantages of ADT :-
1) Encapsulation
2) Information hiding

3) Implementation Details Hiding.
4)
ADT is reusable data structure.
5) Encapsulation ensures that data cannot be corrupt.

* Data Structure :-

It is a set of Domains D, a design domain $\in D$, a set of functions F and set of axioms A. The triple (D, F, A) denotes the data structure d and it will usually return as d where domain (D) denotes data object. Functions (F) denotes set of operations that can be carried out on the data object and axioms (A) denotes the properties and rules of the operation.

* Operations on Data Structure :-

1) Insertion
2) Deletion
3) Searching
4) Traversal (Visiting each & every node).
5) Sorting
6) Merging

* Need of Data Structure :-

Data Structure is used to study how data is stored in a computer so that operations can be implemented efficiently.
Data Structure is especially important when you have large amount of information.

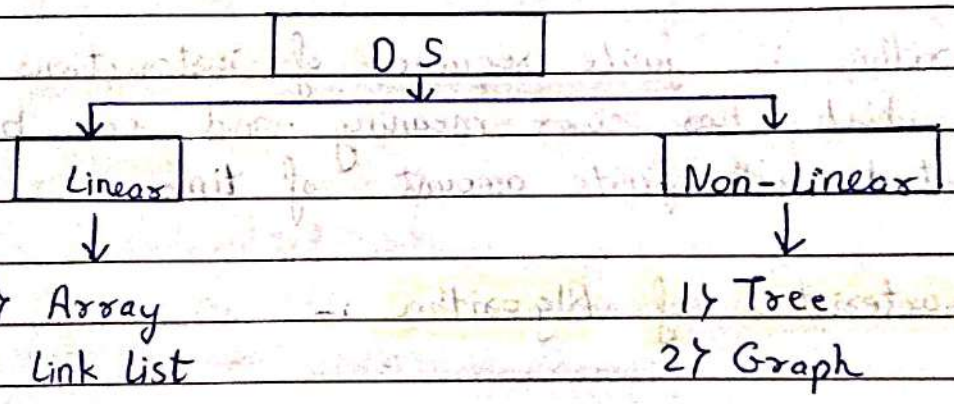
It is also used to organise data for efficient storage and manipulation.
DS helps us to understand the relationship of one element with another.

** * **Advantages of DS :-**

1) It allows information to store on disk very efficiently.
2) We can access data anytime and anywhere.
3) It allows processing of data on software system.
4) It is secure way of storing data.

** * **Types of Data Structure :-**

Data Structure can be divided in 2 types :-
1) Linear DS
2) Non-linear DS



* Linear DS :-

1) In Linear DS, data elements are arranged in linear or sequential order.
2) In this DS, single level is involved.

3) Data items / elements can be traversed in single ~~run~~.

4) It is easy to implement but memory is not utilised in efficient way.

* Non - Linear DS :-

1) A DS is said to be non-linear if element are attached in hierarchal manner and multiple ~~tabels~~ levels are involved in this type of DS, element do not form a sequence.

2) Data items cannot be traversed in single run.

3) These are difficult to implement but memory is utilised in efficient way.

* Algorithm Analysis :-

Algorithm is finite sequence of instructions each of which has clear meaning and can be executed with finite amount of time.

** * Characteristics of Algorithm :-

1) Input
Algorithm should accept 0 or more inputs.

2) Output
Algorithm must produce atleast 1 result or O/P.

3) Definiteness
Each step of algo. must be clear and meaningful.

4) Finiteness
Algo. must halt. It must terminate after finite steps.

5) Efficientness :-
Instruction can be execute in finite time.

** * Performance Analysis :- [For 2 Mks]
It is an algorithm, in which process of calculating space required by that algo. & time req. by that algo. Performance analysis uses following majors :
1) Space req. to complete the task of that algo. (Space Complexity)
2) Time req. to complete the task of that algo. (Time Complexity).

1) Space Complexity :-
Total amt. of computer memory req. by an algo. to complete it's execution is called as space complexity of that algo.
There are 3 reasons :-
i) Instruction Space
It is amt of memory used to store compile version of instruction.
ii) Environmental Stack
It is the amt. of memory used to store info. of partially executed functions at a time of function call.
iii) Data Space
It is amt. of memory used to store all the variables & constants.

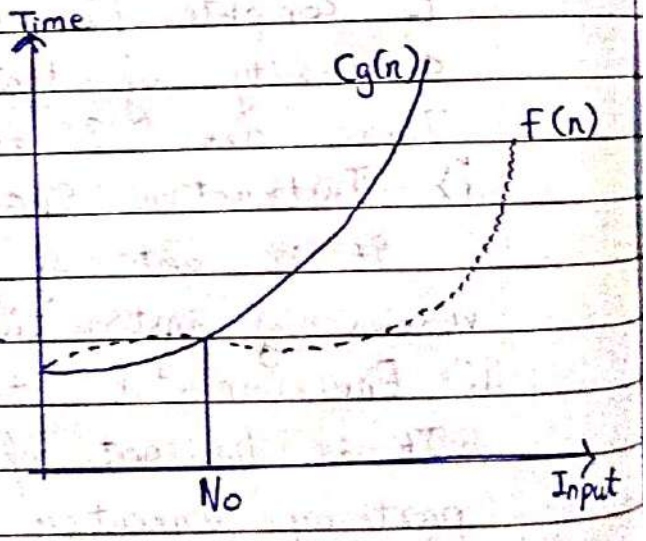
2) Time Complexity :-
The time Complexity of an algo. is the total amt. of time req. by an algo. to complete it's execution.

** * **Frequency Count** :-
It tells how many times the statement is executed in the program.

Time req. by each statement depends on :-
i) Time req. for executing it once
ii) No. of times the statement is executed.

** * **Asymptotic Notation** :-
Asymptotic Notation of an algo. is a mathematical representation of it's complexity.
Types of Asymptotic Notation :-

** i) **Big-oh (O) Notation** :-



Big-oh Notation is used to define the upper bound of an algo. in terms of time complexity. This notation indicates maximum time req. by an algo. for all input values that means Big-oh notation describe the worst case of algo. time complexity.
It is defined as follows :-

$$f(n) <= Cg(n)$$

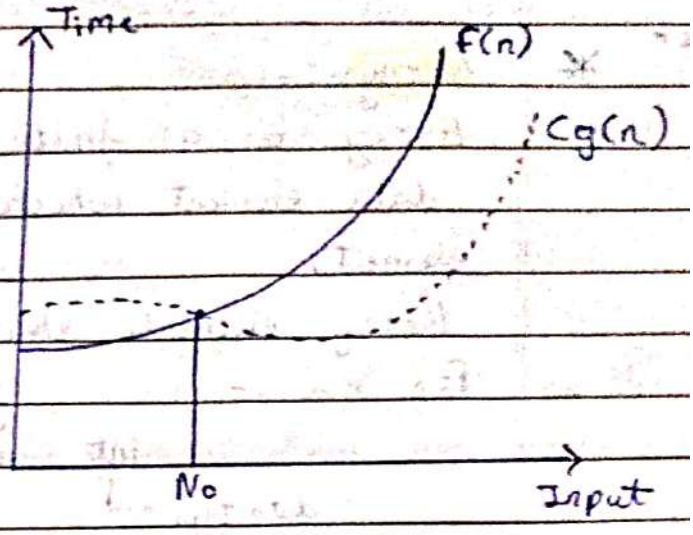
for all $n >= n_0$ &

$C > 0$,

$n_0 >= 1$

ii) Big - Omega (Ω) Notation :-

It is used to define the lower bound of an algo. in terms of time complexity. It always indicates the minimum time req. by an algo. for all input values that means Big - Omega notation describe best case of algo. time complexity.



It is defined as follows :-

$$f(n) >= cg(n)$$

for all $n >= n_0$ &

$c > 0$

$n_0 >= 1$

Unit II :-

Array

** * **Array** :-

Array is a finite ordered collection of homogeneous data element which provides random access to the element.

Array elements share a common name.

For ex :-

array-name

int a[5]

data type — array size

a	10	20	30	40	50
	0	1	2	3	4

* Application of Array :-

1) Array is used to store list of values.
2) Array is used to perform matrix operation.
3) Array is used to implement search algo.
4) Array is used to implement sorting algo.
5) Array are used to implement Data Structure.
6) It is used to implement CPU scheduling algo.

* Types of Array :-

Array are classified into 2 types :-

Array

- One-Dimensional
- Multi-dimensional
  - Two dimensional
  - n dimensional

* One-Dimensional Array :-
The simplex form of an array is one dimensional array. The array is given a name and it's element are referred by their subscript or indices.

** * Calculate the address of any element of an array by using formula.
Address (i^{th}) element = Base address + (offset of the ith element from base address)

where,
offset of ith element = (no. of element before i^{th} element) * (size of each element)

* Example :-
If int a[5] is an array of 5 elements, each integer is of 2 byte and base address is 100 then find the address of 3rd and 5th element.

⇒ Using above formula,
Add. (3rd element (2)) = 100 + (2×2)
= 100 + 4
∴ a[2] = 104
Add. (5th element (4)) = 100 + (4×2)
= 100 + 8
∴ a[4] = 108

* 2-Dimensional Array :-

It is a collection of element place in m rows and n columns. In 2-D array, elements are stored either column by column i.e., column major representation or row by row i.e., row major representation.

** 1) **Row major representation** :-

To calculate address in 2-D array using formula.

arr[i][j] = base address + [(i × no. of columns + j) × size of element (data type)]

Example :-

- i) To calc. address of num[2][3] of int. array of size 3×4 with base address 1000.

⇒

0th				1st				2nd			
1	2	3	4	5	6	7	8	9	10	11	12
1000	1002	04	06	08	10	12	14	16	18	20	22

Using formula,

arr[2][3] = 1000 + [(2 × 4 + 3) × 2]

= 1000 + [11 × 2]

= 1000 + 22

arr[2][3] = 1022

- ii) To calculate add. of A(1,2) of int. array of size 2×3 with base address 100.

⇒ A(1,2) = 100 + [(1 × 3 + 2) × 2]

= 100 + [5 × 2]

= 100 + 10 = 110

** * **Column major representation :-**

Base address of element in 2-D array with column major is calculated using formula.

arr[i][j] = base address + [(j X no. of rows + i) X size of element (data type)]

Example :-

— int A[2][3] = {1, 2, 3, 4, 5, 6} assuming A is stored in column major order with first element of A is at address 1000 and each int. occupying 2 bytes. What would be the address of element A[1][2] ?

0th	1st	2nd			
0	1	2	3	4	5
1	2	3	4	5	6
1000	02	04	06	08	10

⇒ Using formula,

A[1][2] = 1000 + (2X2 + 1) X 2

= 1000 + 5 X 2

= 1000 + 10

= <u>1010</u>

* Sparse matrix :-

It is 2-D data object with m rows & n columns. Therefore having total mXn element in which most of the element have value 0. Sparse matrix contains lesser non-zero element than 0. So, instead of storing zeros with non-zero element, we only store non-zero element. This means storing non-zero element with triples (row, column & value) where.

Row - index non-zero element

Column - index non-zero element

Value - value of non-zero element present at index

(row & column).

For example :-

$$\begin{array}{c|cccc} 0 & 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 0 & 4 \\ 2 & 3 & 0 & 0 & 6 \\ 3 & 0 & 0 & 0 & 0 \\ & 0 & 1 & 2 & 3 \end{array}$$

Row	0	1	2	2
Column	1	3	0	3
Value	7	4	3	6

* Use of Sparse Matrix :-

1) Storage
There are lesser non-zero element than zeros and because of this lesser memory can be used to store only those element.

2) Computing time
It can be saved by logically design data structure traversing only non-zero element.

** Sorting Concept :-
Sorting is a technique to rearrange the element in ascending or descending order which can be numerical, graphical or user-defined order.
Sorting Algo. can be divided :-

1) Internal Sort
This method used only the primary memory during sorting process. All data items are held in main memory & no secondary memory is used for sorting process.
Ex :- Bubble sort, insertion sort and quick sort.

2) External sort :-
Sorting large amt. of data requires external or secondary memory. This process uses external memory such as hard disk, floppy disk and magnetic tape.
For ex :- Merge Sort.

i) Bubble sort :-
In this sorting method, the list is divided into two sub-lists :- sorted & unsorted. The smallest element is bubbled from unsorted sublist after moving the smallest element the imaginary wall moves one element ahead.

** Algorithm :- Arranging element in ascending order

Step 1 : Read N (no. of element)
Step 2 : Read Array A[0], A[1] ..... A[n-1]
Step 3 : for i=0 to n-1 do
for j = 0 to n-i-1 do
if (A[j] > A[j+1]) then
temp = A[j]
A[j] = A[j+1]
A[j+1] = temp
Step 4 : Stop

* Example :- [5 M]
Consider the array A containing following element to be sorted using bubble sort.

1) A = { 13, 11, 14, 15, 19, 9 }

Pass 1 : 13 11 14 15 19 9

11 13 14 15 19 9

11 13 14 15 19 9

11 13 14 15 19 9

11 13 14 15 19 9

11 13 14 15 9 [19]

Pass 2 :

11 13 14 15 9 [19]

11 13 14 9 [15] [19]

Pass 3 :

11 13 9 [14] [15] [19]

Pass 4 :

11 9 13 14 15 19

Pass 5 :

9 11 13 14 15 19

2) Sort following members using bubble sort method.

{108, 3, 97, 65, 11, 23, 57, 93, 100}

Pass 1 : 108 3 97 65 11 23 57 93 100

3 108 97 65 11 23 57 93 100

3 97 108 65 11 23 57 93 100

3 97 65 108 11 23 57 93 100

3 97 65 11 108 23 57 93 100

3 97 65 11 23 108 57 93 100

3 97 65 11 23 57 108 93 100

3 97 65 11 23 57 93 108 100

3 97 65 11 23 57 93 100 $\boxed{108}$

Pass 2 :

3 97 65 11 23 57 93 100 $\boxed{108}$

3 65 97 11 23 57 93 100 $\boxed{108}$

3 65 11 97 23 57 93 100 $\boxed{108}$

3 65 11 23 97 57 93 100 $\boxed{108}$

3 65 11 23 57 97 93 100 $\boxed{108}$

3 65 11 23 57 93 97 $\boxed{100}$ $\boxed{108}$

Pass 3 :

3 65 11 23 57 93 97 $\boxed{100}$ $\boxed{108}$

3 11 65 23 57 93 97 $\boxed{100}$ $\boxed{108}$

3 11 23 65 57 93 97 $\boxed{100}$ $\boxed{108}$

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

Pass 4 :

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

Pass 5 :

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

Pass 6 :

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

Pass 7 :

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

Pass 8 :

3 11 23 57 65 93 97 100 108

3 11 23 57 65 93 97 100 108

* Time Complexity for bubbled sort :-
The no. of passes req. may be between 1 & n-1 i.e., n-1 comparison in first iteration (n-2) in 2nd iteration & one comparison last iteration. The total no. of comparison $= \frac{n(n-1)}{2}$ so the time complexity is order of n^2.

* Advantages & Disadvantages :-

i) Simple & easy to implement
ii) It's slowest method because complexity is order of n^2.
iii) Inefficient for large array size.

☆ Bubbled sort Function

```
void Bubble_sort ( int A[max] , int n)
{
  int i, j, temp ;
  for( j=0 ; j<n-i-1 ; j++)  {
   if ( A[j] > A[j+1] )  {
    temp = A[j] ;
    A[j] = A[j+1] ;
    A[j+1] = temp ;
   }
  }
}
```

☆ Insertion sort :-

In insertion sort the element is inserted at appropriate place. The array is divided into two part sorted and unsorted sub array. In each pass first element of unsorted sub array is pickup & move into sorted sub array by inserting in suitable position. :-

☆ Algorithm :-

Step 1 : Set $i=1$;
Step 2 : key = $A[i]$;
Step 3 : Set $j = i-1$;
Step 4 : if key < $A[j]$ then
$A[j+1] = A[j]$;
$j = j-1$;
repeat Step 2 until $j >= 0$
Step 5 : $A[j+1]$ = key ;
$i = i+1$;
Step 6 : Repeat step from 2 to 5 till $i<N$
Step 7 : Stop ;

☆ Example : Sort element using insertion sort.
{ 22, 14, 25, 10, 5 }

⇒ 22 | 14 25 10 5
sorted | unsorted array
key = 14 — 14 < 22 = true

14 22 | 25 10 5
sorted | unsorted
key = 25 — 25 < 22 = false

14 22 25	10 5
sorted array	unsorted

key = 10 — 10 < 25 = True
10 < 22 = True
10 < 14 = True

10 14 22 25	5
sorted array	unsorted

Key = 5 — 5 < 25 = True
5 < 22 = True
5 < 14 = True
5 < 10 = True

∴ 5 10 14 22 25

★ Complexity of insertion sort :-
Total no. of comparisons are $(n-1) + (n-2) + (n-3) \ldots$
$\ldots + 1$ which is,

$$(n-1) + (n-2) + (n-3) + \ldots + 1$$

$$= \frac{n(n-1)}{2}$$

$$= \frac{n^2 - n}{2}$$

$$= O(n^2)$$

Worst case complexity is $O(n^2)$
Avg. case complexity is $O(n^2)$
Best case complexity is $O(n)$

☆ Advantages & Disadvantages :-

1) Relatively simple & easy to implement.
2) Inefficient for large array. as time complexity is $O(n^2)$.
3) Insertion sort is highly efficient if the array is already in sorted order.

☆ Quick Sort :-

Quick sort is fastest sorting method. It follows divide & conquer method i.e. no. are divided & again sub-divided. This division goes on until it is not possible to divide further. The procedure is applied recursively to the two part of array on the either side of pivot element. It is also called partition exchange sort. Pivot element can be any element from the array. It can be 1st element, last element or any random element.

* Example :-

1) 35, 50, 15, 25, 80, 20, 90, 45

⇒ P→ [35] 50 15 25 80 20 90 45
L h ← - - - - h

[35] 20 15 25 80 50 90 45
h L

25 20 15 [35] 80 50 90 45

[25] 20 15 +∞ 35 [80] 50 90 45 +∞
L h

L greater than equal to
H Less than equal to
Condition true then stop
Condition false then i++ or i-- for L and H

15 20 25 35 $\boxed{80}$ 50 90 45 +∞
L H

15 20 25 35 $\boxed{80}$ 50 45 90 +∞
H L

15 20 25 35 45 50 80 90 +∞

24 30 27 32 11 21 19

⇒ P → $\boxed{24}$ 30 27 32 11 21 19
L H

$\boxed{24}$ 19 27 32 11 21 30
L L H H

$\boxed{24}$ 19 21 32 11 27 30
L L H H

$\boxed{24}$ 19 21 11 32 27 30
H L

11 19 21 $\boxed{24}$ 32 27 30
L H

11 19 21 24 $\boxed{32}$ 27 30
L H

11 19 21 24 $\boxed{32}$ 27 30 +∞L
H

11 19 21 24 $\boxed{30}$ 27 32 +∞
H L

11 19 21 24 27 30 32 +∞

* Time Complexity for best & average case is $O(n \log n)$ and for worst case $O(n^2)$.

* Advantages :-

i) It is faster method among all sorting algo. methods.
ii) It's efficiency is relatively good.
iii) Requires small amt. of memory.

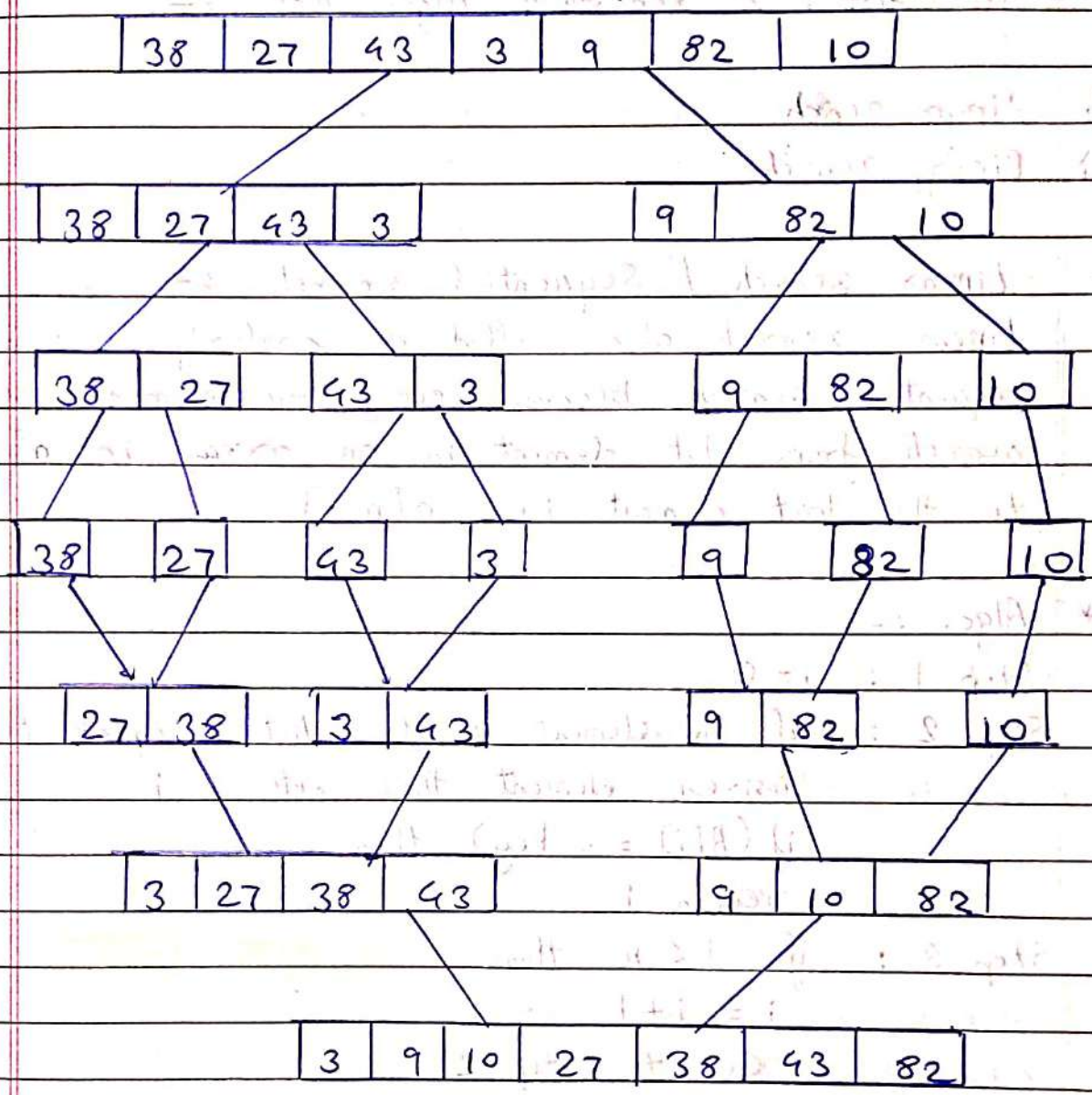
* Disadvantage :-

i) It is complex method of sorting.
ii) It is little hard to implement than other sorting algo.

* Merge Sort :-
The basic concept of merge sort is divide the list into 2 smaller sublist of approximately equal size. Recursively repeat this procedure till only ~~the~~ One element is left.

Algorithm :-
1. Divide step
If a given array A has 0 or 1 element simply written it is already sorted, otherwise split into 2 subarrays each combining half of the element.
2. Conquer step
Conquer by recursively sorting 2 sub arrays.
3. Combine step
Combine the element by merging 2 sorted sub array and into a sorted sequence.

* Example :-
38, 27, 43, 3, 9, 82, 10



* Time complexity for merge sort for best, avg. & worst case is $O(n \log n)$

** Searching Technique :-

Searching algo. are used to find the elements in the list. 2 searching algo. are :-

1) Linear search
2) Binary search

1) Linear search / Sequential search :-

Linear search also called as orderly search or sequential search because every key element is search from 1st element in an array i.e., a[0] to the last element i.e., a[n-1]

* Algo. :-

Step 1 : i = 0
Step 2 : if the element in the list is equal to the desired element then return i
if (A[i] = = key) then
return i
Step 3 : if i < n then
i = i + 1
Go to step 2
Step 4 : if i = n then return -1
Step 5 : Stop

* Complexity of linear search is O(n)

* Advantages :-

i) Linear search is simple & easy to understand & implement.
ii) It does not require the data in the array to be sorted in any particular order.
iii) If key element match the 1st element in the array then linear search algo. is best case in terms of executing time.

* Disadvantages :-

i) If the array size is very large then linear search is not efficient.
ii) When key element match the last element in the array or key element doesn't match any element in the array then this search algo. is in worst case.

** ☆ Binary search :-
Binary search is quicker than linear search. It cannot be applied on unsorted data structure. Binary search is based on divide & conquer approach.

* Algo :-
Step 1 : Initialize $lb = 0$, $ub = n-1$
Step 2 : $lb <= ub$
Step 3 : $mid = \left[\frac{lb + ub}{2} \right]$
Step 4 : if $a[mid] == key$
Step 5 : Set $pos == mid$

Step 6 : Break & jump to step 10
Step 7 : Else if key < a[mid]
Step 8 : ub = mid - 1
Step 9 : Else lb = mid + 1
Step 10 : If pos < 0
Step 11 : printf "Element is not found"
Step 12 : Else print pos

* Example :-

Consider array 5, 7, 11, 19, 30, 35, 42 & key = 11.

⇒

5	7	11	19	30	35	42
0	1	2	3	4	5	6

$$mid = \frac{lb + ub}{2} \qquad [lb = 0, ub = 6]$$

$$= \frac{0 + 6}{2} = 3$$

i.e. a[3] > key

$$ub = mid - 1 \qquad [ub = 2, lb = 0]$$

$$= 3 - 1 = 2$$

$$mid = \frac{lb + ub}{2} = \frac{0+2}{2} = 1$$

i.e. a[1] < key

$$lb = mid + 1 \qquad [lb = 2, ub = 2]$$

$$= 1 + 1 = 2$$

$$mid = \frac{lb + ub}{2}$$

$$= \frac{2+2}{2}$$

$$= 2$$

i.e. a[2] = key

* Advantages :-

i) It is very time efficient as time complexity is $O(\log n)$.

* Disadvantages :-

i) Array should be sorted before applying this algo on it.

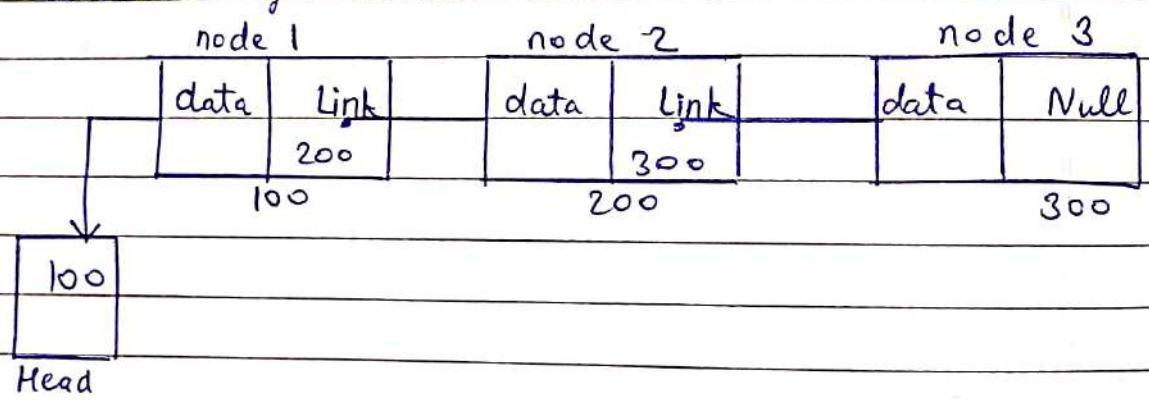
Unit III

LINKED LISTS

** * **Linked list** :-

Linked list or one way list is a linear collection of data element called nodes. Each node contains 2 parts :- 1st part is data and 2nd part is link / address.

* Structure of Linked List :-



* Operations on Linked List :-

Major operation performed on Linked List are as follows :

i) Create
ii) Traverse
iii) Insert a node
iv) Delete a node

Few more operations are :

i) Searching a node
ii) Updating node

iii) Counting length
✓ iv) Reverse the Linked List
v) Sort the Linked List
✓ vi) Concatenate 2 Linked Lists
vii) Merge 2 sorted Linked Lists.

* Representation of Linked List in memory :-
Linked list can be implemented as :
i) Static implementation method
ii) Dynamic implementation method

* Static Representation of Linked List :-
A static Data Structure is organized or collection of data in memory i.e. fixed in size. This results in the maximum size needs to be known in advance as memory cannot be reallocate at the runtime.

* Drawbacks of static representation :-

i) A fixed no. of node have to be represent at the start of the program execution.
ii) To predict no. of nodes in prior is not always possible.
iii) Memory is wasted due to static allocation. The node must be allocated to a program only when it is required which is possible in dynamic memory allocation.

☆ Dynamic Memory Allocation :-

A Linked List is linear data structure in which element are not sorted at continuous memory allocation. Each data element or node holds a pointer (link) to the next node in the list.

☆ Node Structure :-

```
struct node
{
int data ;
struct node * next ;
} ;
```

☆ Type of Linked List :-

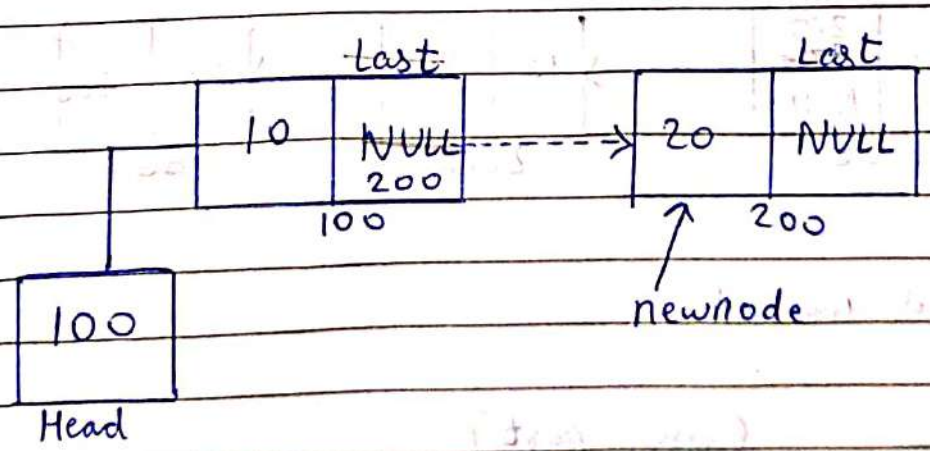
- Linked List
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List
    - Singly circular linked list
    - Doubly circular linked list

☆ Singly Linked List :-

A Linked List in which every node has one field linked to provide information about where the next node of list is called as singly linked list. In singly linked list we can traverse only one direction.

* Append Function into singly linked List :-

```
struct node *append (struct node *head)
{
    struct node *newnode = NULL, *temp = head;
    newnode = (struct node *) malloc (sizeof(struct node));
newnode -> next = NULL;
scanf ("%d", & newnode -> data);
if (head == NULL)
    head = last = newnode;
else {
    last -> next = newnode;
    last = newnode;
}
return head;
}
```

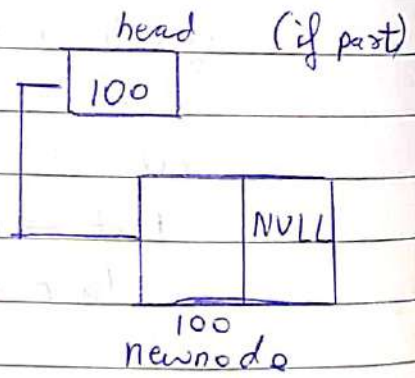


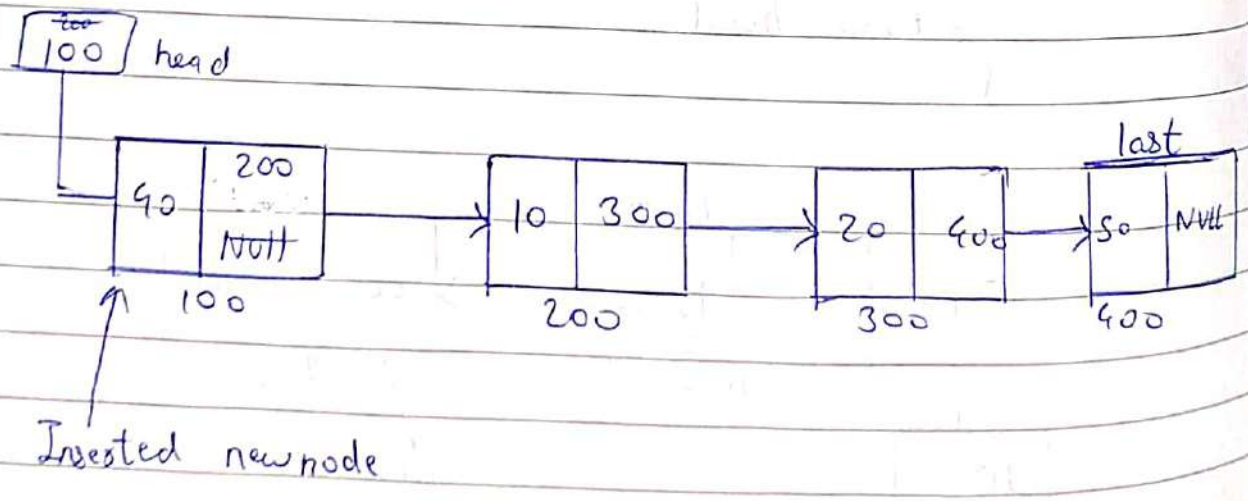
☆ Insertion of a node :-

There are various position where node can be inserted ;

i) Insert at the beginning of the node (1st position) :-

```
struct node *SLL-insert-front (struct node *head)
{
 struct node *newnode = NULL ;
 newnode = (struct node *) malloc (sizeof (struct node));
 newnode -> next = NULL ;
 scanf ("%d", & newnode -> data) ;
 if ( head == NULL)
      head = newnode ;
 else
  {
 newnode -> next = head ;
   head = newnode ;
 }
   return head ;
 }
```

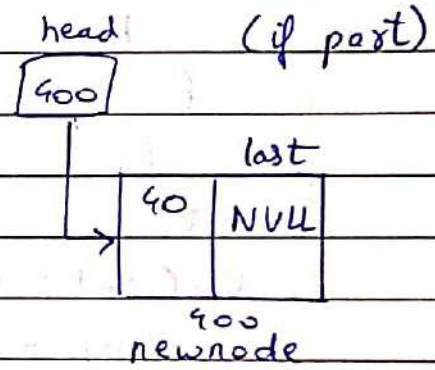


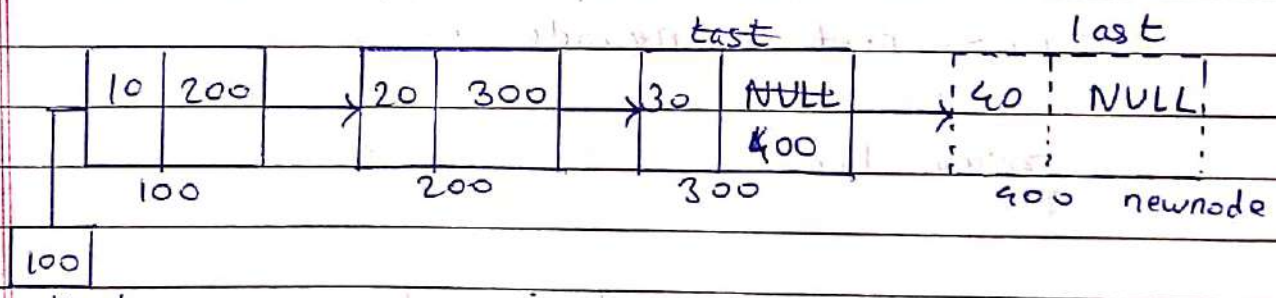


(else part)

ii) Inserting at end :-

```
struct node *SLL_insert_end (struct node *head)
{
struct node *newnode = NULL , *temp = NULL;
newnode = (struct node *) malloc (sizeof (struct node));
newnode -> next = NULL;
scanf ("%d", & newnode -> data);
 if ( head == NULL)
    head = last = newnode ;
 else
 {
   last -> next = newnode ;
   last = newnode ;
 }
   return head ;
}
```

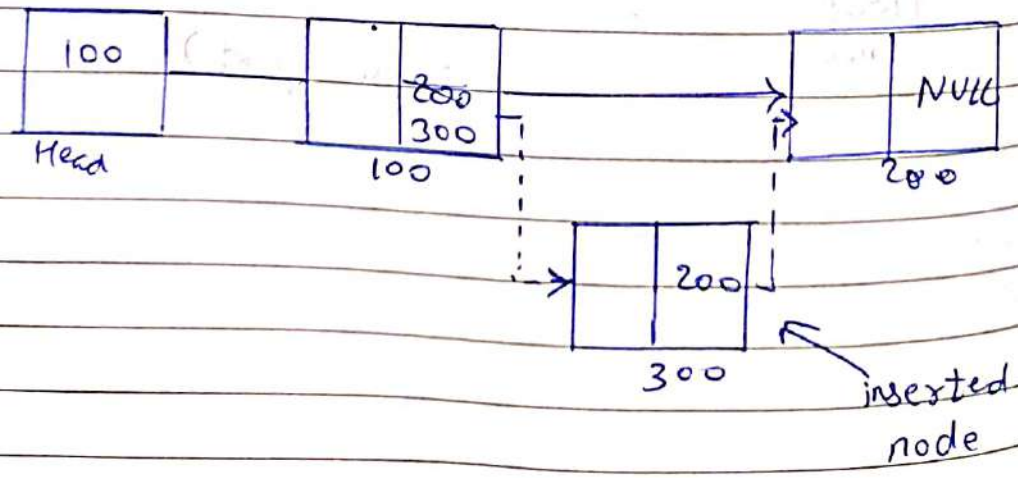




(else part)

iii) Insert at given position (middle of the linked list):-

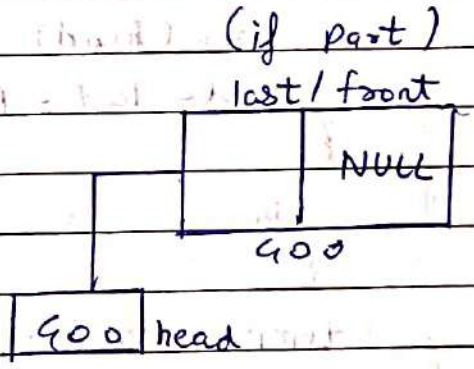
```
struct node *SLL_insert_middle(struct node *head, int pos)
 {
   int i;
  struct node *newnode = NULL, *temp = head;
  newnode = (struct node*) malloc (size of (struct node));
  newnode -> next = NULL;
    if (pos == 1)
   {
    newnode -> next = head;
      head = newnode;
   }
    else
   {
  for(i=1; i< pos-1; i++)   {
     temp = temp -> next;
  newnode -> next = temp -> next;
  temp -> next = newnode;
   }  }
   return head;
  }
```

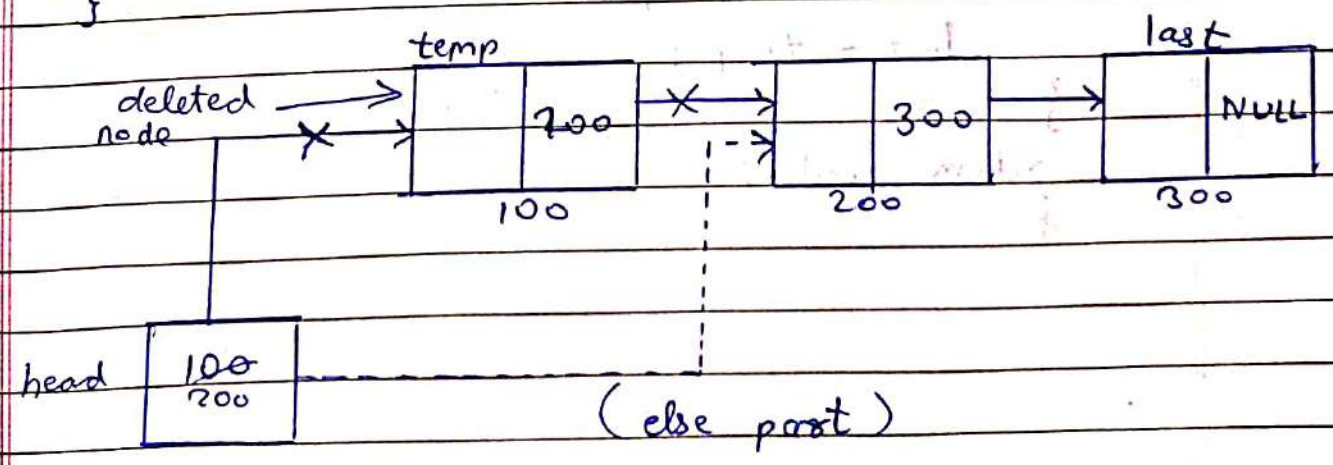


☆ Deleting a node from linked list :-
Deletion can also have various cases like insertion.

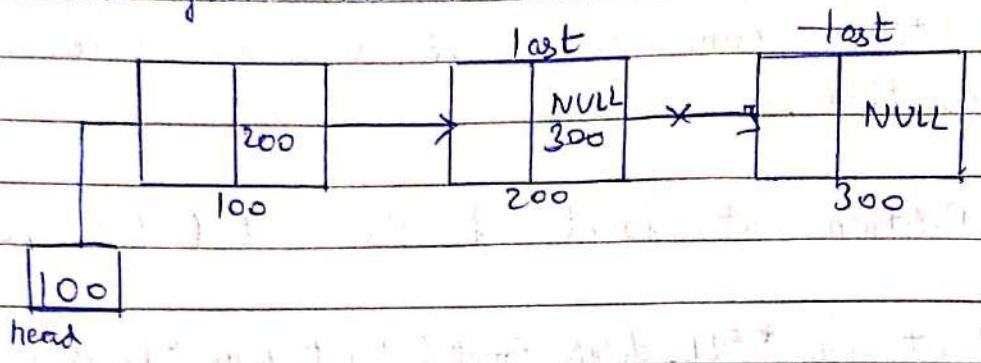
i> Deletion at front of the list (1st position) :-

```
struct node * SLL_delete_front (struct node *head )
  {
    struct node *temp = NULL ;
printf (" Deleted element is %d \n", head -> data) ;
     if ( head -> next == NULL )
     {
        free ( head) ;
        head = last = NULL ;
     }
     else
     {
     temp = head ;
     head = head -> next ;
     temp -> next = NULL;
      free (temp) ;
     }
     return head ;
  }
```





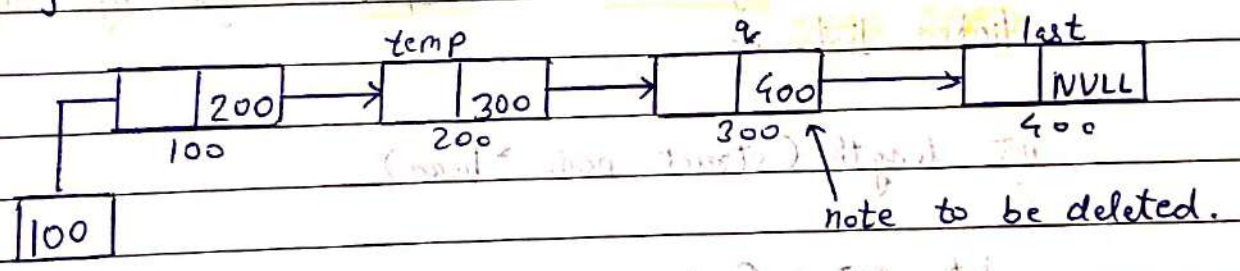
☆ Deletion from end :-




```
struct node *SLL_delete_end (struct node *head)
 {
   struct node * temp = NULL;
  printf(" Deleted element is %d\n", head -> data);
   if (head -> next == NULL)
    {
      free (head);
      head= last = NULL;
    }
     else
     {
    temp = head;
    while (temp -> next != last)
          temp = temp -> next;
      temp -> next = NULL;
           free (last);
          last = temp;
    }
     return head;
    }
```

☆ Deletion at any position of the list (middle of the list) :-

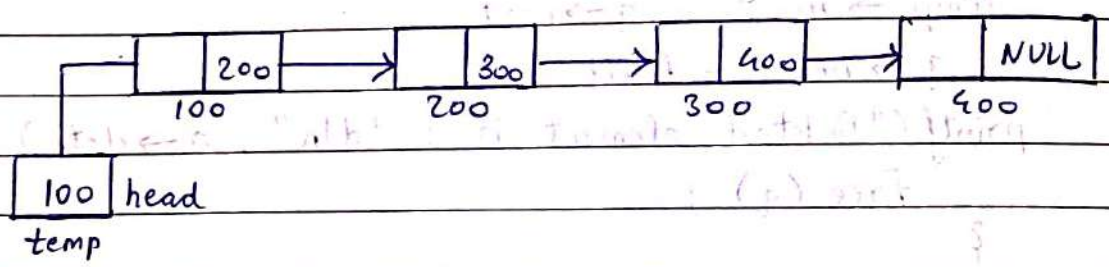
```
struct node *SLL_delete_middle (struct node *head, int pos)
{
  int i;
struct node *q = NULL, *temp = head;
for (i=1 ; i<pos-1 ; i++)  {
 temp = temp -> next ;
  q = temp -> next ;
temp -> next = q -> next ;
  q -> next = NULL ;
printf ("Deleted element is : %d\n", q -> data) ;
    free (q) ;
 }
  return head ;
}
```



☆ Traversing or display a linked list :-

Traversing means visit all element sequentially in the list. To traverse a link list, start from 1st node using next field, go to second node & so on.

```
void display (struct node *head)
{
   struct node * temp = head ;
printf (" In Linked List contents are : In") ;
    while (temp != NULL)
    {
    printf ("%d ", temp -> data) ;
     temp = temp -> next ;
      }
    }
```



Write a C function to compute length of singly linked list :-

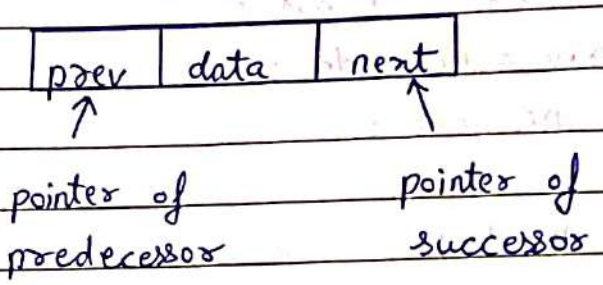
```
int length (struct node *head)
{
  int cnt = 0 ;
 while ( head != NULL )
 {
   cnt ++ ;
   head = head -> next / link ;   [ any 1 next or link]
 }
   return cnt ;
}
```

☆ Doubly Linked List :-

Doubly Linked list is a linear collection of data element called nodes where each node is having 3 fields, i) A pointer field which contain address of Predecessor node in the list.

ii) The data field which contain actual data like integer value.

iii) Pointer field which contain address of successor node in the list.



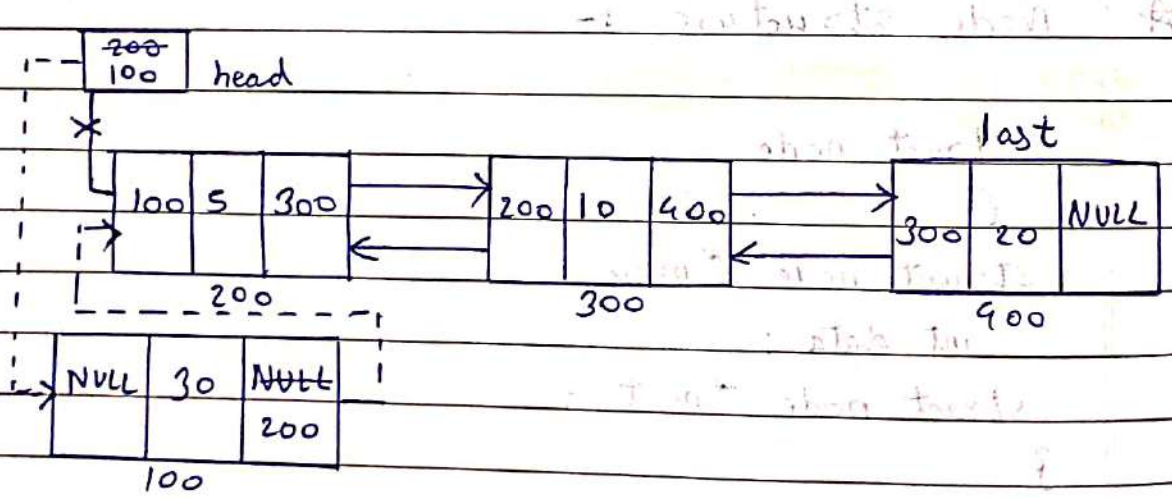
☆ Node structure :-

```
struct node
{
struct node *prev ;
  int data ;
struct node *next ;
}
```

☆ Insertion of node in doubly linked list :-

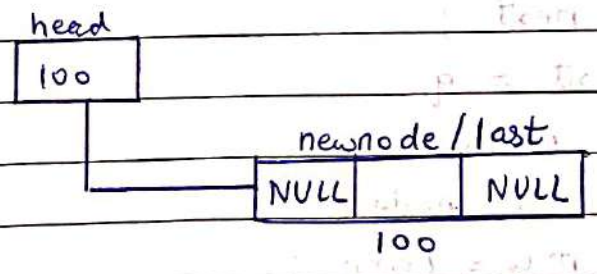
1) Insert at beginning of the list (1st position) :-

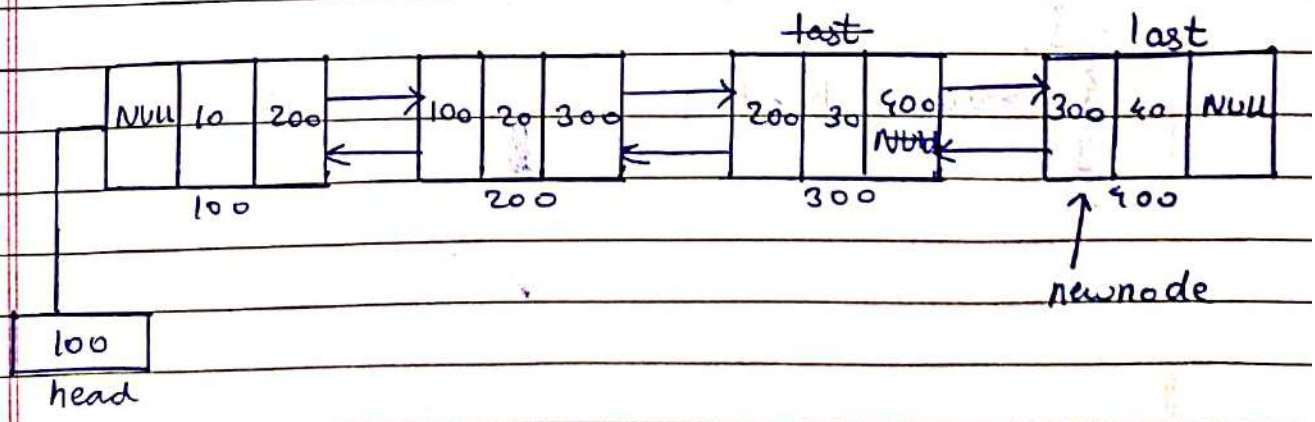
```
struct node *DLL_insert_front (struct node *head)
{
 struct node * newnode = NULL;
newnode = (struct node *) malloc (size of (struct node));
newnode -> prev = newnode -> next = NULL;
 scanf ("%d", newnode -> data);
   if (head == NULL)
      head = last = newnode;
   else {
    newnode -> next = head;
     head -> prev = newnode;
       head = newnode;
    }
      return head;
    }
```



2) Insert at the end (last position) :-

```
struct node *DLL_insert_end (struct node *head)
{
struct node *newnode = NULL, *temp = head;
newnode = (struct node *) malloc (sizeof (struct node));
newnode -> prev = newnode -> next = NULL;
scanf ("%d", & newnode -> data) ;
    if ( head == NULL)
  {
    head = last = newnode;
  } else
  {
    last -> next = newnode;
    newnode -> prev = last ;
        last = newnode ;
  }
      return head ;
  }
```





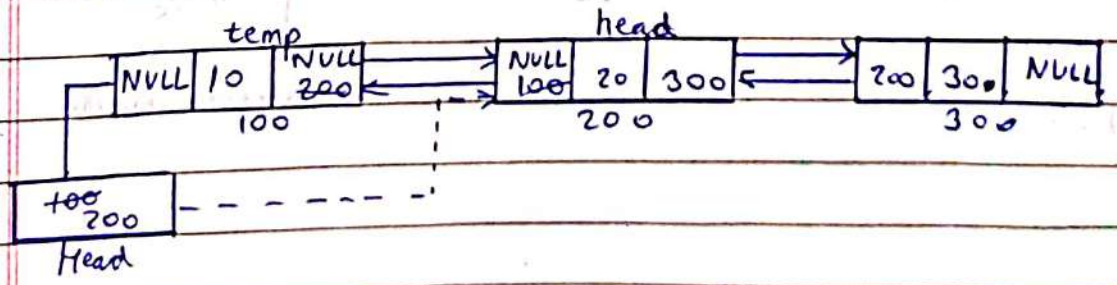
3) Insertion of node at given position (middle) :-

```
struct node *DLL_insert_middle (struct node *head , int pos)
 {
   int i;
struct node *newnode = NULL; *temp = NULL, *q = NULL;
newnode = (struct node*) malloc ( sizeof (struct node));
 newnode -> prev = newnode -> next = NULL;
 scanf ("%d", &newnode -> data);
    if (pos == 1)
 {
    newnode -> next = head ;
        head -> prev = newnode ;
              head = newnode ;
 } else
  {
     temp = head ;
     for ( i=1 ; i<pos-1 ; i++)  {
    temp = temp -> next ;
    q = temp -> next ;
    newnode -> next = q ;
     q -> prev = newnode ;
     temp -> next = newnode ;
     newnode -> prev = temp ;
  }
 } return head ;
 }
```

☆ Delete a node from Doubly Linked List :-

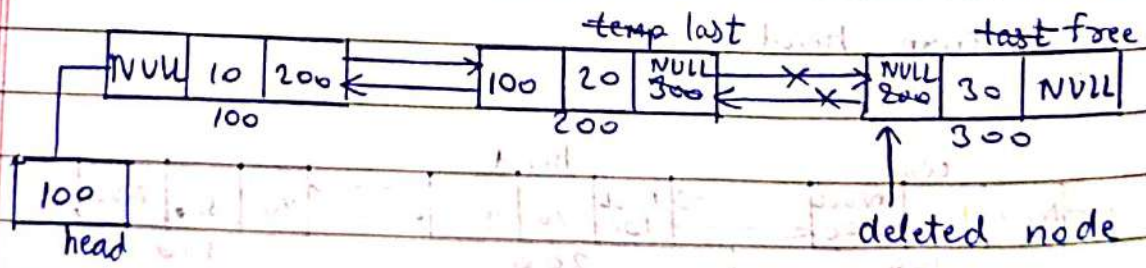
1) Delete at the front of the list / Delete at the beginning :

```
struct node *DLL_delete_Front (struct node *head)
 {
  struct node *temp = NULL ;
 printf (" Deleted element is %d\n", head -> data) ;
  if ( head -> next == NULL)
   {
     free( head) ;
    head = last = NULL ;
  }  else
   {
   temp = head ;
    head = head -> next ;
   head -> prev = NULL ;
   temp -> next = NULL;
    free ( temp) ;
  }
     return head ;
  }
```



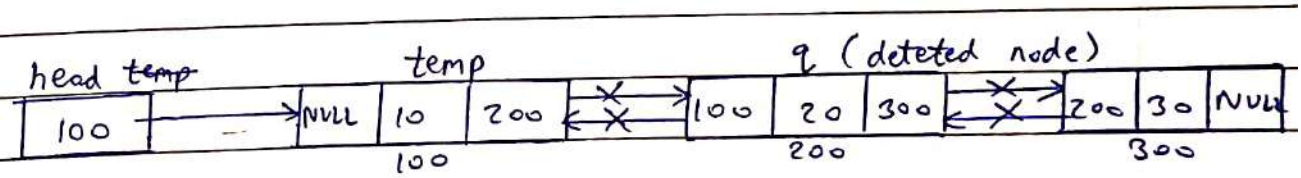
2) Deletion at end :

```
struct node * DLL- delete_end (struct node *head)
{
  struct node * temp = NULL ;
printf (" Deleted element is %d \n", head -> data) ;
  if ( head -> next == NULL)
   {
     free (head) ;
   head = last = NULL ;
  }
   else
   {
   temp = last -> prev ;
   temp -> next = NULL ;
    free (last) ;
    last = temp ;
   }
return head ;
}
```



3) Delete from specific Position / middle :-

```
struct node *DLL_delete_middle (struct node *head, int pos)
 {
  int i;
struct node *temp = NULL , *q = NULL ;
 temp = head ;
for ( i=1 ; i<pos-1 ; i++)
  temp = temp -> next ;
   q = temp -> next ;
  temp -> next = q -> next ;
                q -> next -> prev = temp ;
     q -> next = q -> prev = NULL;
printf (" Deleted element is : %d\n", q -> data) ;
   free (q) ;
  }
  return head;
 }
```



☆ Traversing a link :- Traversing a node in doubly linked list is ~~said~~ same as in singly linked list only the advantage of DLL is list can traverse in both direction (forward & backward) using next & previous pointer.

**

```
void display (struct node *head)
 {
   struct node *temp = head ;
printf ("In Linked list contents are : In");
 while ( temp != NULL)
 {
printf ("%d \t", temp -> data) ;
  temp = temp -> next ;
  }
}
```

** ☆ Difference between singly & doubly Linked List :-

	Singly L.L.	Doubly L.L.
1)	Singly L.L. allows us to go one way direction.	Doubly L.L. has 2 way direction.
2)	Singly L.L. ~~gives~~ use less memory per node (because only 1 pointer).	Doubly L.L. uses more memory per node as compared to S.L.L.
3)	In S.L.L., node contains 2 parts : one is for data & other for link part.	In D.L.L., each node contains 3 parts : Previous, data & next/link part.
4)	S.L.L. is one directional.	D.L.L. is bi-directional.
5)	If we need to save memory to update nodes value frequently	If we need faster performance in searching & memory is not

S.L.L.	D.L.L.
and searching is not required, we can use S.L.L.	a limitation, then we use D.L.L.

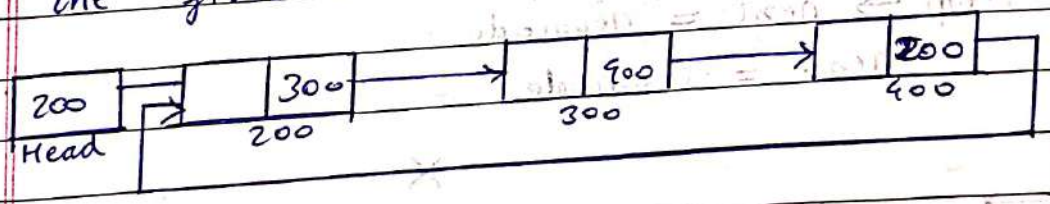
** ☆ **Circular Linked List (C.L.L.) :-**

A C.L.L. is a list in which last node points to the head node. A circular list have neither a beginning nor end. In C.L.L., it is possible to reach any other node.

** * **Types of C.L.L. :-**

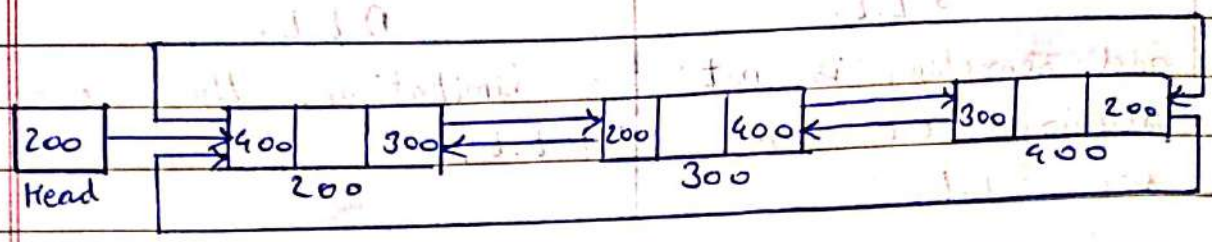
1) Circular Singly Linked List (C.S.L.L.) :-

In S.L.L., every node point to it's next node in the sequence & the last node points NULL. But in C.S.L.L., every node point to it's next node in the sequence but last node point to the first node in the list.



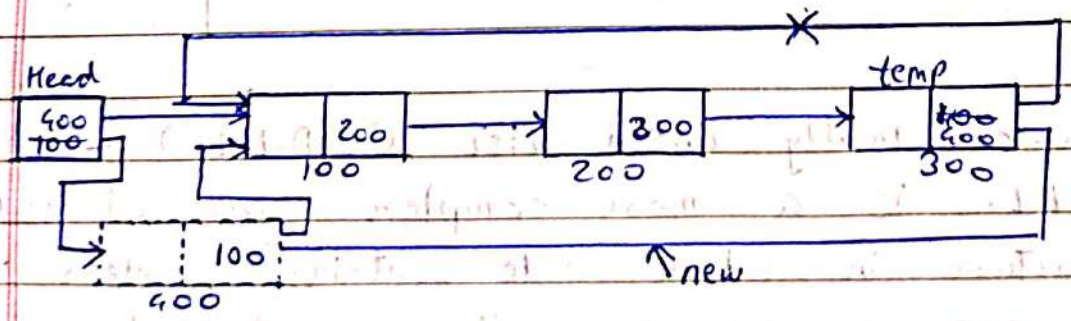
2) Circular Doubly Linked List (C.D.L.L.) :-

C.D.L.L. is a more complex type of Data Structure in which node contains pointers to it's previous node as well as next node. C.D.L.L. doesn't contain NULL in any node. The last node of the list contains address of the first node and first node of the list also contains address of the last node.



★ Insertion at the beginning in C.S.L.L. (Insert at front) :-

Step 1 : Create a newnode
Step 2 : Check whether list is empty (head == NULL)
Step 3 : If it is empty then
set head = newnode
newnode → next = head
Step 4 : If it is not empty then,
set temp = head
Step 5 : Traverse the temp to it's next node until it reaches to last until (temp → next == head)
Step 6 : set newnode → next = head
temp → next = newnode
head = newnode



☆ Insertion at the end in C.S.L.L. (at last) :-

Step 1 : Create a newnode
Step 2 : Check whether list is empty (head == NULL)
Step 3 : If it is empty then,
set head = newnode ;
newnode → next = head
Step 4 : If it is not empty,
set temp = head
Step 5 : Traverse the temp to it's next node until it reaches to last node until (temp → next == head)
Step 6 : set temp → next = newnode
newnode → next = head

☆ Insertion at given position in C.S.L.L :-

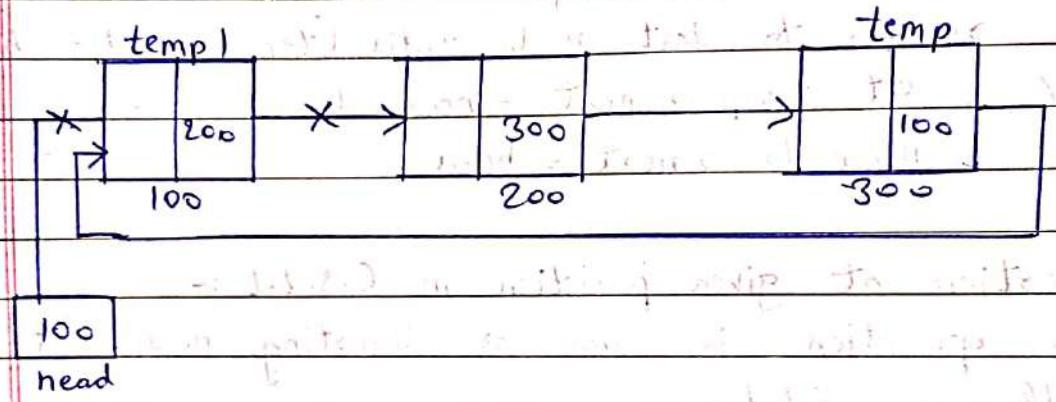
This operation is same as inserting node at middle of S.L.L.

☆ Deletion in C.S.L.L :-

1) Delete from beginning of C.S.L.L. (Delete from front) :-

```
struct node *CSLL_delete_front (struct node *head)
{
  struct node *temp = NULL , *temp1 = NULL ;
  if (head → next == head)
  {
     free (head) ;
     head = NULL ;
  }
  else {
```

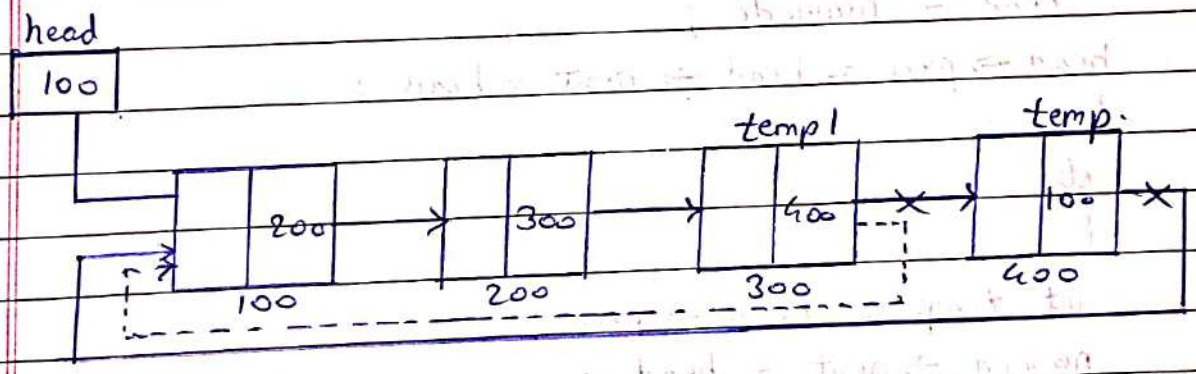
```
temp = temp1 = head ;
while ( temp -> next != head )
    temp = temp -> next ;
    head = head -> next ;
    temp -> next = head ;
    temp1 -> next = NULL ;
    free (temp1) ;
    }
    return head ;
}
```



2) Delete from end of C.S.L.L :-

```
struct node * CSLL_delete_end ( struct node * head)
{
struct node * temp = head , * temp1 = NULL ;
if ( head -> next == head)
    {
        free (head) ;
        head = NULL ;
    }
else    {
        temp1 = temp ;
        temp = temp -> next ;
```

```
while ( temp -> next != head)
  {
    temp = temp -> next ;
    temp1 = temp1 -> next ;
  }
  temp1 -> next = head ;
  temp -> next = NULL ;
   free (temp) ;
}
  return head ;
}
```



3) Delete node at given position in C.S.L.L :-
It is same as deleting node at middle of S.L.L.

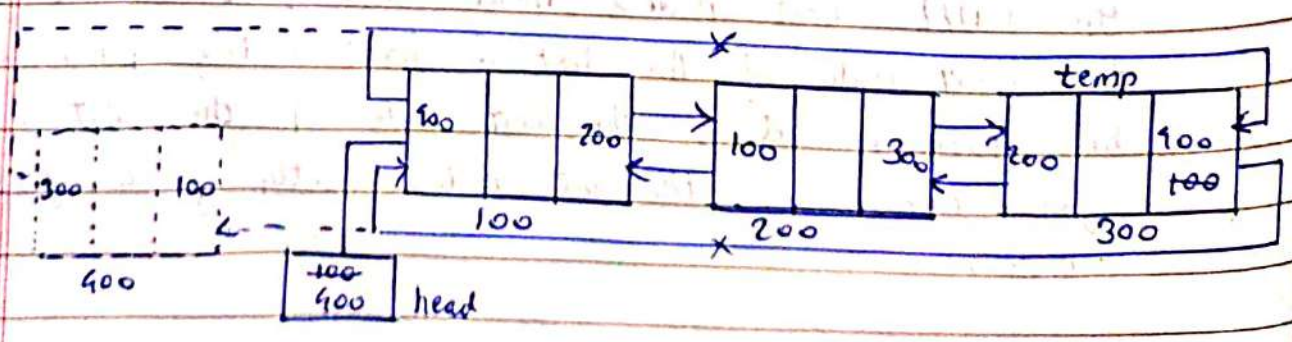
☆ Circular Doubly Linked List (C.D.L.L) :-

In CDLL Last node's next link part is set to the first node of the list & first node's previous link part is set to the last node of the list. These gives access to the last node directly from first node.

☆ Insertion in C.D.L.L. :-

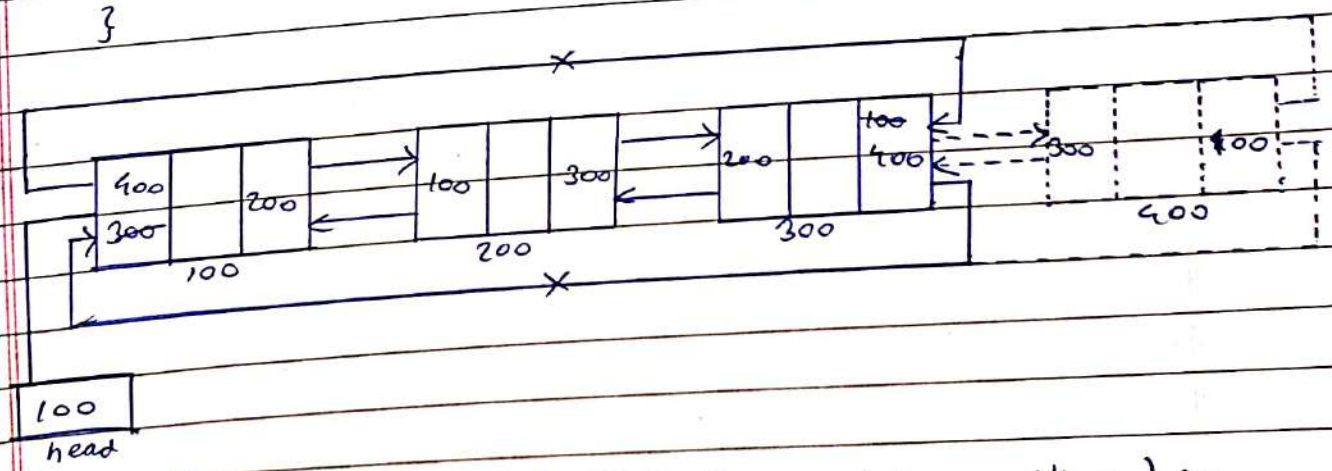
1) Insertion in beginning of CDLL (insert at front) :-

```
struct node *CDLL-insert-front (struct node *head)
{
   struct node * newnode = NULL ;
newnode = (struct node *) malloc (sizeof (struct node));
  newnode -> prev = newnode -> next = NULL;
   scanf ("%d", & newnode -> data );
  if (head == NULL)
   {
     head = newnode ;
   head -> prev = head -> next = head ;
   }
   else
   {
   int temp = head -> prev ;
   newnode -> next = head ;
   head -> prev = newnode ;
   temp -> next = newnode ;
   newnode -> prev = temp ;
       head = newnode ;
   }
   return head ;
   }
```



2) Insert at the end of CDLL (insert at last) :-

```
struct node *CDLL_insert_end (struct node *head)
{
  struct node *newnode = NULL;
newnode = (struct node *) malloc (sizeof (struct node));
scanf ("%d", & newnode -> data);
  if (head == NULL) {
    head = newnode;
  newnode -> next = newnode -> prev = head;
  }
  else {
  temp = head -> prev;
  newnode -> next = head;
  head -> prev = newnode;
  temp -> next = newnode;
  newnode -> prev = temp;
  }
  return head;
  }
```

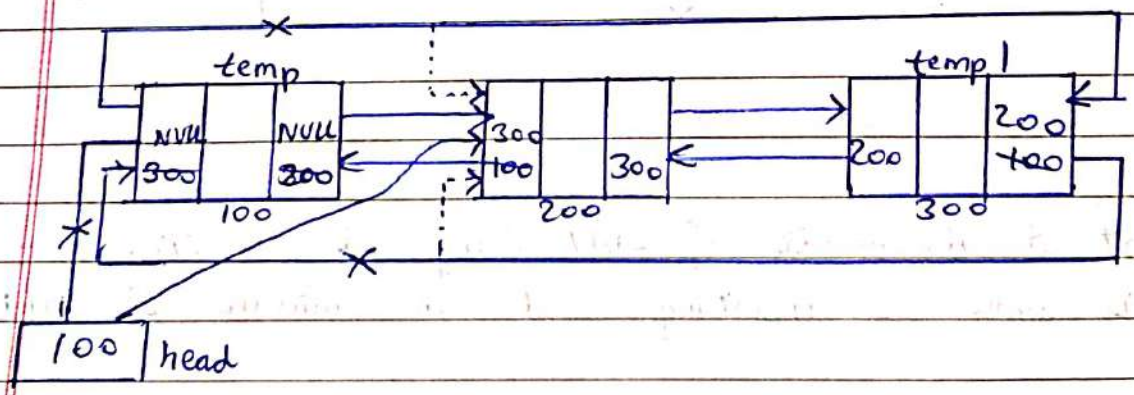


3) Insert at the middle of CDLL (any given position) :-
It is same as inserting node in middle of DLL.

☆ Deletion in C.D.L.L :-

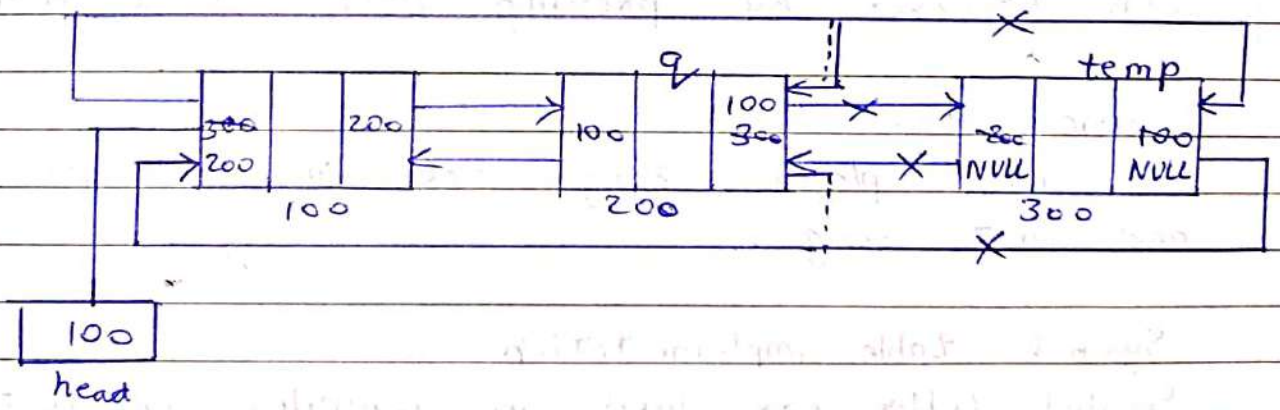
1) Deletion at beginning in CDLL (at front) :-

```
struct node *CDLL_delete_front (struct node * head)
  {
  struct node * temp = NULL , *temp1 = NULL ;
  if ( head -> next == head)
   {
      free (head) ;
      head = last = NULL ;
   }
    else
   {
      temp = head ;
     temp 1 = head -> prev ;
       head = head -> next ;
     head -> prev = temp 1 ;
     temp1 -> next = head ;
   temp -> prev = temp -> next = NULL;
          free (temp) ;
   }
    return head ;
   }
```



27 Delete from end in CDLL (at last) :-

```
struct node * CDLL_delete_end (struct node *head)
{
struct node *temp = head->prev , *q = temp->prev;
 if (head->next == head)
   {
     free (head);
     head = last = NULL;
 }
  else
  {
    temp = head->prev;
      q = temp->prev;
  q->next = head;
head->prev = q;
 temp->next = temp->prev = NULL;
     free (temp);
  }
  return head;
}
```



3) Delete a specific node in CDLL (from middle) :-
Deleting a node at given position of CDLL, this operation is same as deleting node at middle of DLL.

** ☆ Real World Application of Linked List :-

1) Garbage collection
In operating system uses free space management for allocation of block. In order to reuse the memory after deallocation of block, OS provides Linked List to point free block.

2) Searching
When we search particular number we use traversing in Linked List. When given number is match with some number in list, then searching is successful.

3) Previous & next page in Web Browser
We can access previous & next URL search in web browser by pressing back & next button.

4) Music Player
In music player, songs are linked to previous and next song.

5) Symbol table implementation
Symbol tables are used in compiler construction & related fields. They can be used as a directory

for the values. Linked list are best suited for implementing this table.

6) To perform arithmetic operations on large numbers. and Polynomial addition and multiplication.

7) Radix Sort

Unit 4 :-

Stack And Queue

** ☆ Stack :-
Stack is a order to collection of homogenous data element where the insertion & deletion operation takes place at only one end called as Top of the Stack.

Stack is a set of element in a Last in Front Out (LIFO) technique.

☆ Stack related terms :-

1) Stack :-
It is memory position which is used for storing the element in LIFO technique.

2) TOP :-
Top pointer points to the top of the stack element from where elements are inserted or deleted.

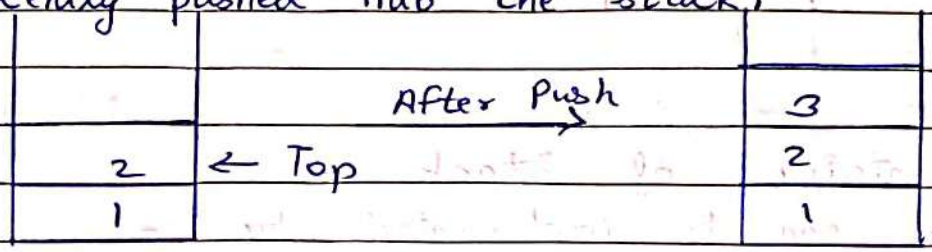
3) Stack Underflow :-
When there is no element in the stack to be delete then stack is underflow. That is stack is empty.

4) Stack Overflow :-
When the Stack contains number of element is equal to the capacity & no more element can be add or can be added then the stack is overflow i.e., stack is full.

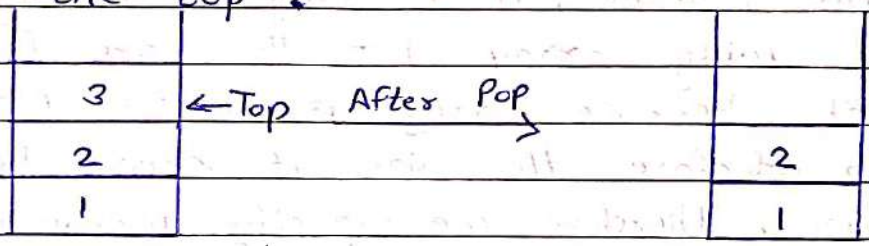
✱ Primitive Operations on Stack :-

1) init (s)
To initialize stack 's'

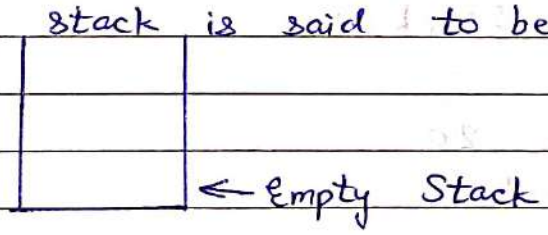
2) push
push operation add an element on the top of the stack. Top always points to the element which is recently pushed into the stack.



3) Pop
Pop removes an element which is at the top of the stack. After removing topmost element, size of the stack reduce by 1 & element below it become the top.



4) IsEmpty
It returns true if stack is empty, else returns false when there is no element present on the stack then stack is said to be empty.



5) Is Full
When top of the stack reach to the stack capacity then stack is said to be full. This function returns true if stack is full else returns false.

5	← Top (max capacity of stack)
4	
3	
2	
1	

☆ Implementation of stack :-
Stack can be implemented by :-
1) Static Implementation of Stack (using array)
2) Dynamic Implementation of Stack (using Linked List)

** 1) Static implementation using array :-
An array is used to store ordered list of elements. Hence, it is easy to manage stack when represented using array but there are limitations on the stack because array is static Data Structure once we declare the size it cannot be modified at runtime. Therefore we specify maximum size of array which leads to poor memory utilization.

☆ Operations on Stack with respect to array implementation :-

☆ Structure of Stack :-

```
# define max 20
struct stack
```

```
{
  int data [max] ;
  int top ;
}
```

1) init ()

```
  S1. top = -1
```

In this function, top is initialized to -1 which indicates stack is empty.

2) Is Empty ()

```
if ( S1. top == -1)
   return 1 ;
 else
 return 0;
```

3) Is Full ()

```
if ( S1. top == size -1 )
   return 1 ;
   else
   return 0;
```

4) Push

Push element

```
if (S1. top == size -1 )
printf (" Stack is overflow ");
else {
    S1. top ++ ;
  S1. data [S1. top] = element ;
}
```

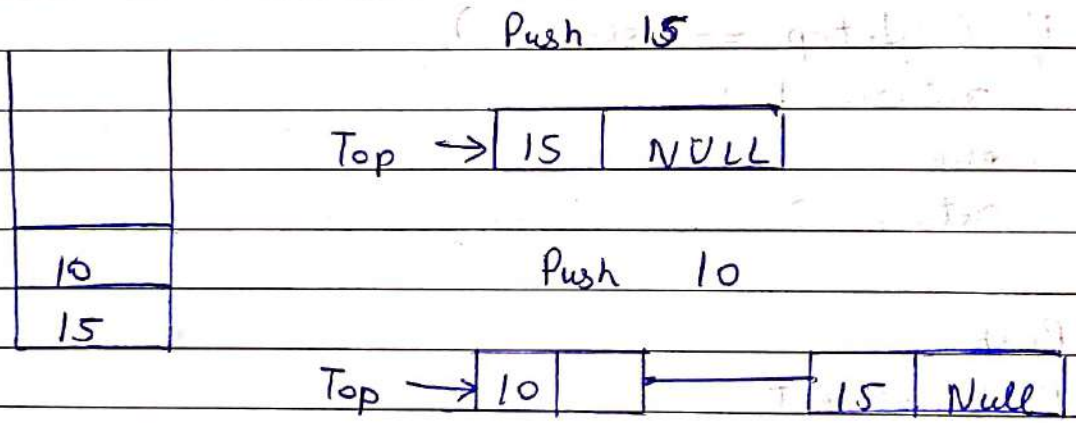
5) Pop

```
if ( S1.top != -1)
return (S1. data [S1 . top - - ] ;
else
printf (" Stack is underflow ") ;
```

☆ Dynamic implementation of Stack using Linked List :-
In Linked list representation, we can add or remove element from any position of the list. In link representation, we cannot need prior knowledge of the element of the list. When we remove element, memory is freed.

☆ Operations on Stack w.r.t. Linked List :-

Link Representation :-



1) Push ()

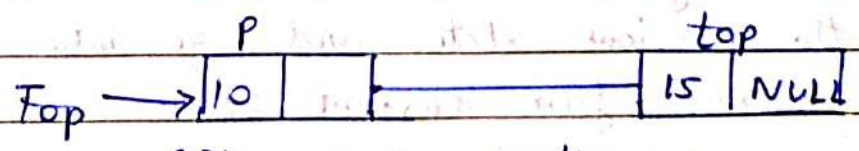
```
void push (struct node *top, int element)
  {
    struct node *p = NULL ;
p= (struct node*) malloc ( sizeof (struct node)) ;
```

```
if ( p == NULL)
printf (" Stack is overflow ");
else
 {
   p -> data = element ;
   p -> next = top ;
        top = p ;
 }
}
```

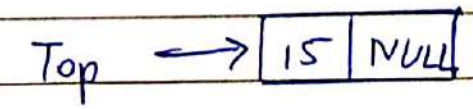
2) Pop ()

```
int pop (struct node * top)
 {
  struct node *p = NULL ;
   int x ;
  if (top == NULL)
 {
 printf (" Stack is underflow ");
    return -1 ;
 }
  else      p = top ;
 {          x = top -> data ;
        top = top -> next ;
        free(p) ;
   }
```

$\boxed{x = 10}$



After pop operation



☆ Applications of Stack :-

1) Expression Conversion
An expression can be represented in prefix, postfix or Infix notation. Stack can be used to convert one form of expression to another.

2) Expression Evaluation
Stack can be used to evaluate prefix, postfix & infix expression.

3) Syntax Parsing
Many compilers use stack for parsing the syntax of expressions, programs or blocks.

4) String Reversal
Stack is used to reverse a string. We can push characters of a string one by one into stack and then pop characters from stack.

5) Parenthesis Checking
Stack is used to check proper opening and closing parenthesis.

6) Back-tracking
Backtracking is used when we want to comeback to the previous state and go into some other path to get back from current state we need to store previous state. For this purpose, we need stack.

4) Function Call

Stack is used to keep information about the active function or sub-routines.

* Recursion

If the procedure is defined by itself is called recursive process. In the recursion procedure is called itself directly or indirectly. Directly means procedure call itself repeatedly. Indirectly means procedure calls another procedure and then procedure or functions are executed repeatedly.

Fact (1) = 1					
2 * fact (1)	1	2 * fact (1)	2		
3 * fact (2)		3 * fact (2)		3 * fact (2)	6
4 * fact (3)		4 * fact (3)		4 * fact (3)	4 * 6 = 24

It is same as,

(4 * (3 * (2 * (fact 1)

1

2

6

24

* Rule of recursion is there must be a terminating condition to stop recursion.

*** ☆ Expression Conversion and Evaluation :-

1) Infix
If the operator symbol are placed between the operands, then the expression is in infix notation. For eg., A + B.

2) Prefix
In this notation operator is prefix to operands i.e., operator is written ahead of operands then this expression is called Prefix Expression. It is also called Polish Notation. For eg., +ab.

3) Postfix
If the operator symbol are placed after it's operands then the expression is in postfix notation. This is also called reverse Polish Notation. For eg., ab+.

☆ Operator Precedence and Associativity Rule :-

Precedence	Operator	Associativity
Lowest	+, −	left to right
Middle	*, /	left to right
Highest	^ or ↑ or $	right to left.

☆ Algorithm to convert infix to postfix expression :-

Step 1 : Read infix expression from left to right & character by character.

Step 2 : For each character 'ch' in the input string if 'ch' is an operand, then output ch.

Step 3 : if 'ch' is an operator, then following action should be taken,

1) Check whether there is any operator already present in stack or not.
2) If stack is empty, then push operator into stack.
3) If present then check priority of 'ch' is greater than priority of topmost stack operator.
4) If priority of 'ch' is greater than push 'ch' into stack.
5) Else pop & print operator until one of the lowest priority operator is encountered or stack becomes empty.

Step 4 : If 'ch' is left parenthesis, push 'ch' into stack.

Step 5 : If 'ch' is right parenthesis, pop an output token until left parenthesis is pop.

☆ Example :-

Consider the following example P/Q − R * S + T.
Convert this infix expression into postfix expression.

⇒

Input	Stack	Output
P		P
/	/	P
Q	/	PQ
-	-	PQ/
R	-	PQ/R
*	-*	PQ/R
S	-*	PQ/RS
+	+	PQ/RS*
		PQ/RS*-
T	+	PQ/RS*-T
Final expression		PQ/RS*-T+

Postfix Expression :-
(PQ/RS*-T+)

☆ Example 2 :-

(A+(B*C-(D/E↑F)*G*H)

⇒

Input	Stack	Output
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D

Input	Stack	Output
E	(+(-(/	ABC * DE
↑	(+(-(/↑	ABC * DE
F	(+(-(/↑	ABC * DEF
)	(+(-	ABC * DEF ↑/
*	(+(-*	ABC * DEF ↑/
G	(+(-*	ABC * DEF ↑/G
)	(+	ABC * DEF ↑/ G * -
*	(+*	ABC * DEF ↑/ G * -
H	(+*	ABC * DEF ↑/ G * - H
)		ABC * DEF ↑/ G * - H * +

☆ Algorithm to convert postfix expression into infix expression :-

Step 1 : Read the expression from left to right, one character at a time.

Step 2 : If character is operand, push it to stack.

Step 3 : If a character is a operator,

i) Pop operand from the stack, say it as S1

ii) Pop operand from the stack, say it as S2

iii) Perform S2 operator S1 and push it to stack.

Step 4: Once the expression iteration is completed, initialize the result string and pop out from stack and add it to result.

Step 5 : Return the result.

* Example :-

ABC / - AK / L - *

Input	Action	Stack
A	push A	[A]
B	push B	[A, B]
C	push C	[A, B, C]
/	pop C	[A, B]
	pop B	[A]
	push [B/C]	[A, B/C]
-	pop [B/C]	[A]
	pop A	[ ]
	push [A-[B/C]]	[A-[B/C]]
A	push A	[A-[B/C], A]
K	push k	[A-[B/C], A, k]
/	pop k	[A-[B/C], A]
	pop A	[A-[B/C]]
	push [A/K]	[A-[B/C], [A/K]]
L	push L	[A-[B/C], [A/K], L]
-	pop L	[A-[B/C], [A/K]]
	pop [A/k]	[A-[B/C]]
	push [[A/K] - L]	[A-[B/C], [A/k]-L]
*	pop [[A/k] - L]	[A-[B/C]]
	pop [A-[B/C]	[ ]
	push [A-[B/C]] * [[A/k] - L]	[A-[B/C] * [A/K] - L]

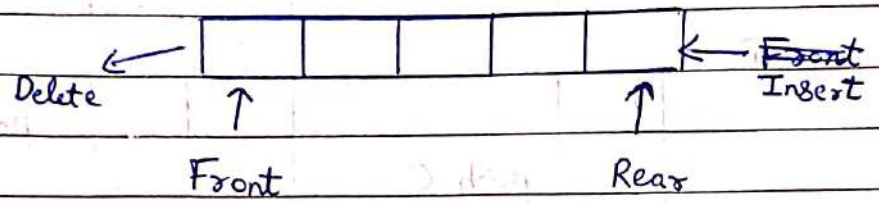
** Evaluate the postfix expression :

1) AB + C − [let A=3, B=4 & C=2]

A — push A	A+B = 3+4 = 7	7−2 = 5
	7 — push 7	5 — push 5
B, A — push B		
	C, 7 — push C	pop 5
A — pop B		
	7 — pop C	
pop A		
	pop 7	

** ☆ Queue :-

Queue is a linear list of element in which data can only be inserted at one end called rear and deleted from other end called the front. 1st element in a queue will be removed first & last element will be removed last so queue is called first in first out (FIFO).



** ☆ Operations on Queue :-

Queue is an ADT. Operation on Queue are as follows :

1) Create
It creates an empty Queue 'Q'.

2) Insert
It adds element to the rear end of the queue 'Q'

3) Delete
It deletes the element from the front end of the queue Q

4) isFull
It returns true, if 'Q' is full otherwise false.

5) is Empty
It returns true if queue is empty otherwise returns false.

6) Peek
It is used to return the value of first element without deleting it.

```
struct queue {
    int a[max];
    int front, rear;
}
```


☆ Representation of Queue :-

There are 2 ways to represent a queue in memory :

i) Static implementation of Queue (using array)
ii) Dynamic implementation of Queue (using L.L.)

1) Static implementation of Queue :-

Various operations on Queue :-

i) Create :-

It creates an empty queue in which front and rear are initialize to -1.

```
void initq(struct queue *q)
{
    q->front = q->rear = -1;
}
```

ii) isEmpty :-

This operation check whether queue is empty or not.

```
int isEmpty(struct queue *q)
{
    if (q->front == q->rear)
        return 1;
    else
        return 0;
}
```

iii) is Full :-

When rear is pointing to last location in the array then there is no space for new element in the queue and we call it as queue is full.

```
int isFull (struct queue *q)
{
  if ( q -> rear == max -1)
      return 1;
    else
    return 0;
}
```

iv) Insert :-

We insert an element in the queue if queue is not full.

```
void insert ( struct queue *q, int x)
{
  q -> data [ ++ (q -> rear) ] = x ;
}
```

v) Delete :-

```
int delete (struct queue *q)
{
return q -> data [++ (q -> front) ] ;
}
```

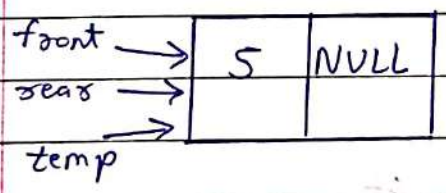
In delete operation, elements delete from the front of queue and returns the same and also set front to

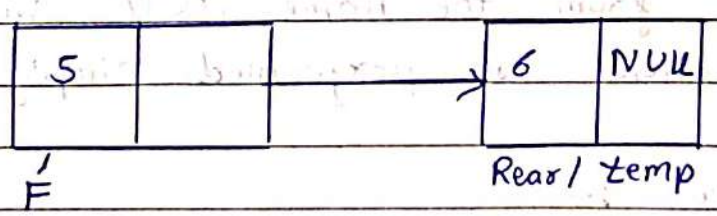
next element. We should increement front first and then delete the element and return that element.

☆ Dynamic implementation of Queue (Using Linked List):-

1> Insert element in Queue :-

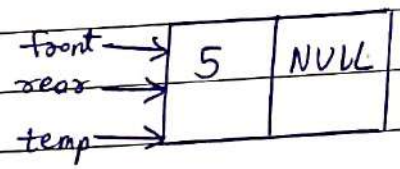
```
void insert Q (int n)
{
 struct queue * temp ;
 temp = (struct queue *) malloc (sizeof (struct queue )) ;
 temp -> data = n ;
 temp -> next = NULL ;
  if ( front == NULL )
    rear = front = temp ;
  else
  {
   rear -> next = temp ;
      rear = temp ;
   }
 }
```

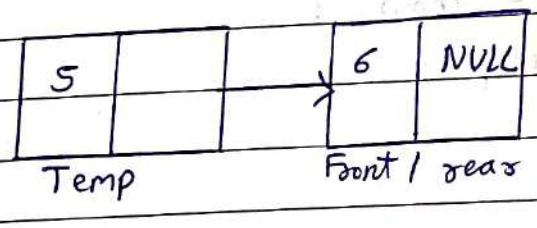




2) Delete element in Queue :-

```
int delete Q ()
{
  int x ;
  struct queue *temp = front ;
  x = front -> data ;
   if ( front == rear)
     front = rear = NULL ;
  else
     front = front -> next ;
  free (temp) ;
  return x ;
}
```





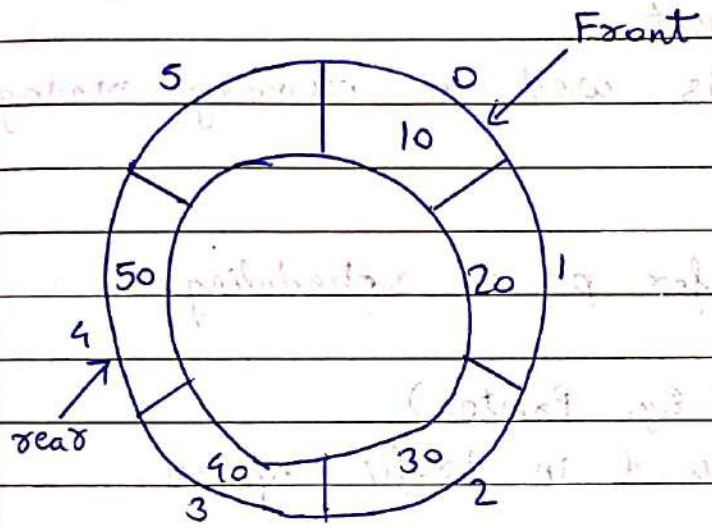
★ Types of Queue :-

1) Simple Queue / Linear Queue :-
As it clear from the name itself, in simple queue, operations are performed simply

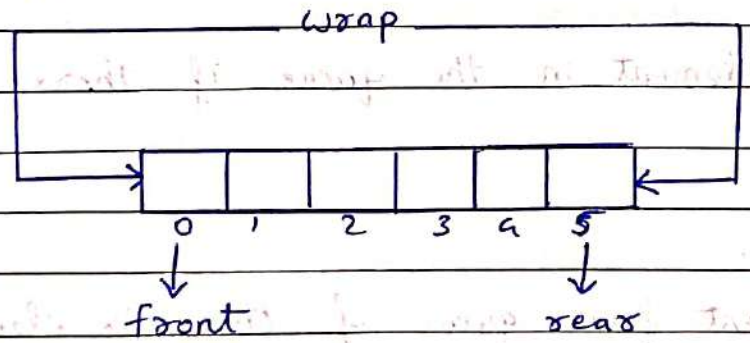
2) Circular Queue :-
The technique which essentially allows the queue to

wrap around upon reaching the end of the queue. eliminate this drawback such technique which allows queue to wrap around from end to start is called circular Queue.

☆ Logical View :-



☆ Physical View :-



☆ When adding element in linear queue / simple queue, when we reach the location n-1, we cannot add more element in the queue but in circular queue when rear = n-1 and we want to add new element, rear should be set to 0th index i.e.,

if (rear = n-1) ⟹ rear = (rear + 1) % n
rear = 0 ;
rear = rear + 1

** ☆ Application of Circular Queue :-

1) Memory Management
Circular queue is used in memory management.

2) Process Scheduling
CPU use queue for process scheduling.

3) Traffic System (E.g. Printer)
Queue are also used in traffic system.

☆ Operations on Circular Queue :-

1) Enqueue :-
Adding an element in the queue if there is space available.

2) Dequeue :-
Removing element from queue if there is element present in the queue.

3) Front :-
Get the 1st element from the queue.

4) Rear :-
Get the last element from the queue.

5) isEmpty or isFull :-
It checks if the queue is empty or full.

★ Advantages of Circular Queue :-

1) Circular Queue consume less memory in compare to linear queue because if we delete one element from circular queue, we can reuse that space.

2) It avoids shifting of Queue elements.

3) Circular Queue is used in many aplications like : Memory management, process scheduling and Traffic system.

★ Priority Queue :-
Priority Queue is a collection in which element can be added at any time but the only element that can be removed is the one with the highest priority.
There are 2 rules to maintain priority queue :-
1) Element with highest priority is processed before any element of lower priority.
2) If there are elements with the same priority, then element are processed according to the order in which they are added to the queue i.e. first come, first serve (FCFS) or FIFO basis.

☆ Operations on Priority Queue :-

1) isEmpty :-
To check priority queue is empty or not.

2) isFull :-
To check priority queue is full or not.

3) insert :-
If Queue is not full, insert the element.

4) delete :-
If not empty, delete the highest priority element.

** ☆ Application of Queue :-

1) Queue is used for Job scheduling in operating system. For e.g., Printer, Process Queue, etc.

2) Queue is used in simulation. E.g., In airport, to keep the track of problems like no. of planes processed.

3) Handling of Interruptions in Real time system.

4) To process the job in multi-user system, queue is used.

5) In batch programming, to process job sequentially.

☆ Difference between Stack & Queue :-

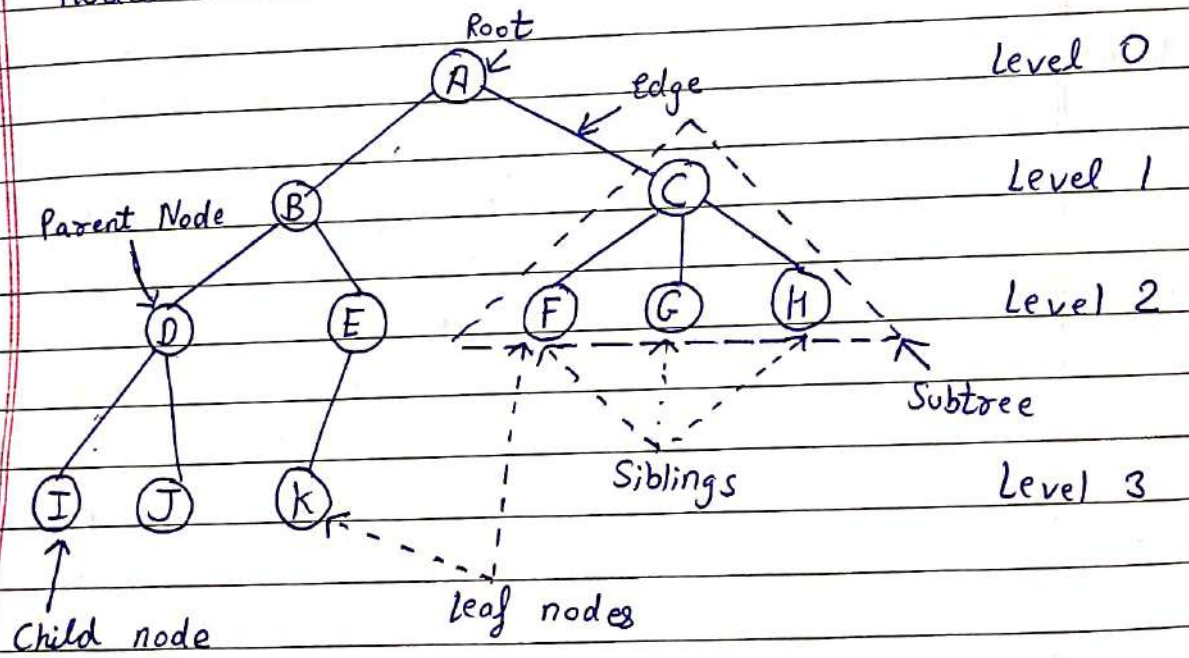
1) Definitions
2) LIFO & FIFO
3) Structure
4) Stack has 1 ends & queue has 2 ends.

Unit 5 :-

TREE

** ☆ Tree :-

Tree data structure is a non-linear data structure that is used to represent and organize data in the form of parent child relationship. The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes.



☆ Terminologies used in Tree :-

1. Parent Node :

The node which is an immediate predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.

2. Child Node :
The node which is the immediate successor of a node is called the child node of that node. {D, E} are the child nodes of {B}.

3. Root Node :
The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

4. Leaf Node / External Node :
The node which do not have any child are called Leaf Nodes. {I, J, K, F, G, H} are the leaf nodes in tree.

5. Sibling :
Children of the same parent node are called siblings. {D, E} are siblings of parent {B} and {F, G, H} are siblings of parent {C}.

6. Level of a Node :
The count of edges on the path from the root node to that node. The root node has level 0.

7. Subtree :-
Any node of the tree along with its descendant.

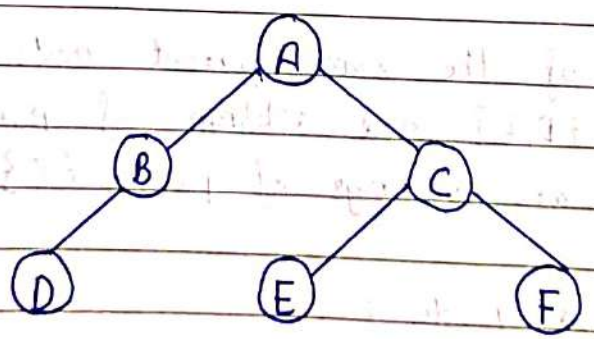
8. Degree of Node :-
In a tree, the degree of a node is defined as the number of children that node has.

9. Height of the tree :-
The length of the longest path from the root of the tree to a leaf node of the tree. The maximum depth or height of tree is the no. of edges in the tree from the root to the deepest node.

** ☆ Binary Trees & its types :-

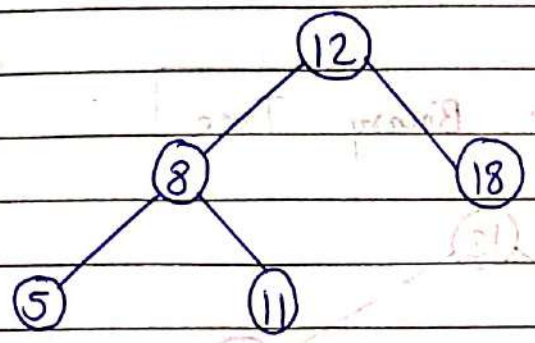
1) ☆ Binary Trees :-
It is a non-linear and hierarchial data structure where each node has at most 2 children referred to as the leaf child & the right child. The topmost node in a binary tree is called root & the bottom-most nodes are called leaves.

E.g.,



2) Full Binary Tree :-
A binary tree is a full binary tree if every node has 0 or 2 children. A Full binary tree is a binary tree in which all nodes except leaf nodes have 2 children.

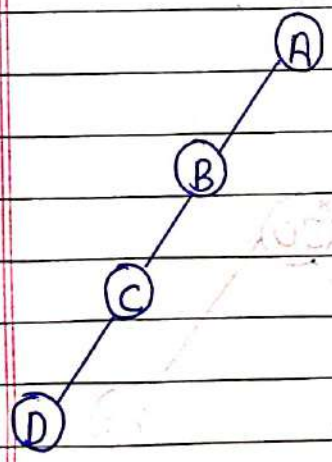
E.g.,



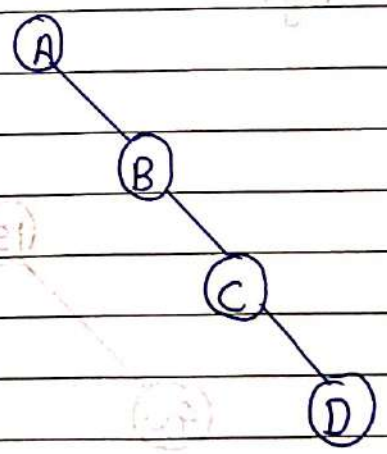
3) Skewed Binary Tree (Degenerate Tree) :-
A skewed binary tree is a degenerate tree in which the tree is either left nodes or the right nodes. Thus, there are 2 types of skewed binary tree :- left-skewed and right-skewed binary tree.

E.g.:-

Left-skewed



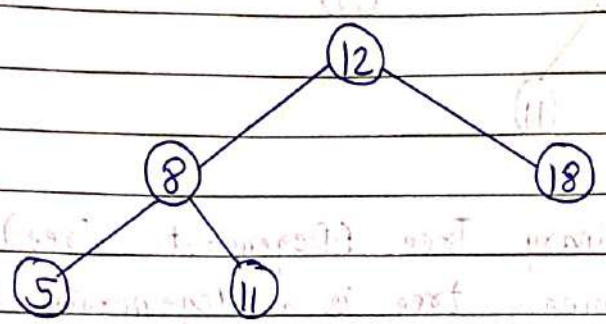
Right-skewed



4) Complete Binary Tree :-
A Binary tree is a complete binary tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible. At the lowest level the nodes have 0 children and the level above that nodes can have 0, 1 or 2 children.

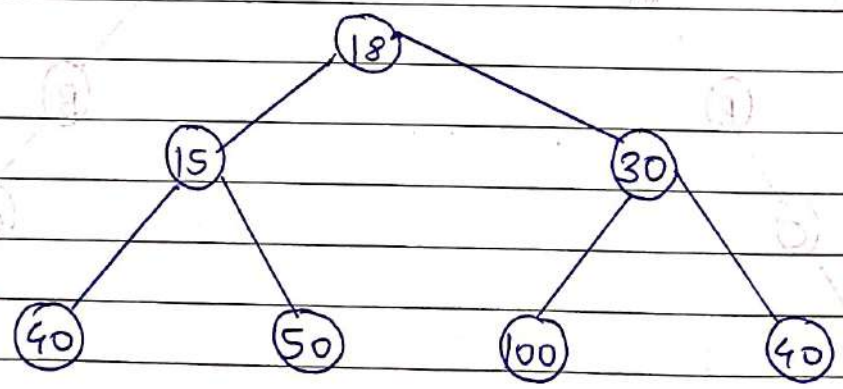
E.g :-

Complete Binary Tree



5) Perfect Binary Tree :-
A binary tree is a Perfect Binary Tree in which all the internal nodes have 2 children and all leaf nodes are at the same level.

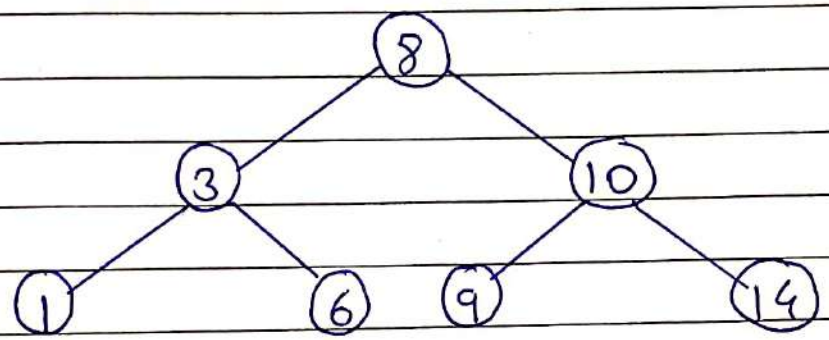
E.g. Perfect Binary Tree



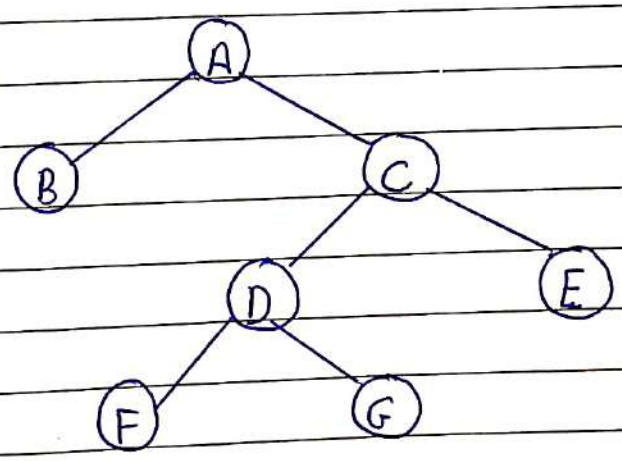
6) Binary Search Tree :-
It is a node-based binary tree data structure,
- The left subtree of a node contains only nodes with keys lesser than its parent node.
- The right subtree of a node contains only nodes with keys greater than its parent node.
- The left and right subtree each must also be a

binary search tree.

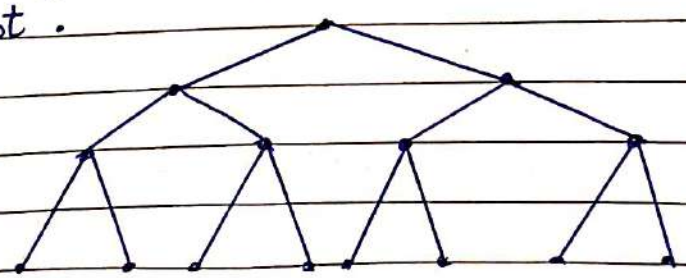
Binary Search Tree



7) Strictly Binary Tree :-
A strictly binary Tree is a binary tree where all non leaf nodes have 2 branches.



8) Rooted tree :-
A rooted tree is a tree with a designated node (top most node) called the root. In rooted tree every edge directly or indirectly originates from the root.



★ Representation of Binary Tree :-
Binary tree can be represented in 2 ways :-

1) Sequential Representation using (Array) / Static
2) Link Representation (Dynamic)

1) Sequential representation of Binary tree (using Array):-
In sequential representation, numbering of the nodes starting with nodes on level 0, then level 1 and so on.
In Complete Binary Tree we calculate height by using formula $(2^{h+1}-1)$. The nodes can be stored in 1-D array. Tree, with the node numbered at the location TREE (i). The array of size can be calculated using formula $(2^{h+1}-1)$ or 2^d-1 (where d = no. of levels).

★ The root node is stored in the first memory location as the first element in the array.
Following rules can be used to decide the location of any i^{th} node of a tree:
For any node with index i, $1 \leq i \leq n$;

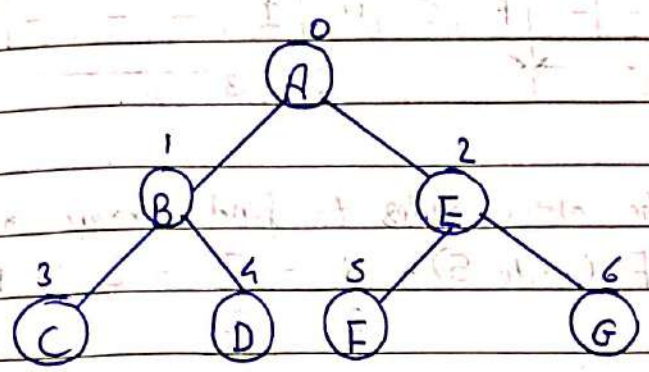
a) PARENT (i) = $\left[\frac{i}{2}\right]$ if $i \neq 1$

If i=1 then it is root which has no parent.

b) LCHILD (i) = 2 * i , if $2i \leq n$.
If $2i > n$, then i has no left child

c) RCHILD (i) = 2i + 1, if $2i+1 \leq n$
If $(2i+1) > n$, then i has no right child.

★ Example 1:



No. of Level = 3 (0 to 2) and height = 2

∴ We need the array of size $2^{2+1}-1$ is $2^3-1 = 7$.

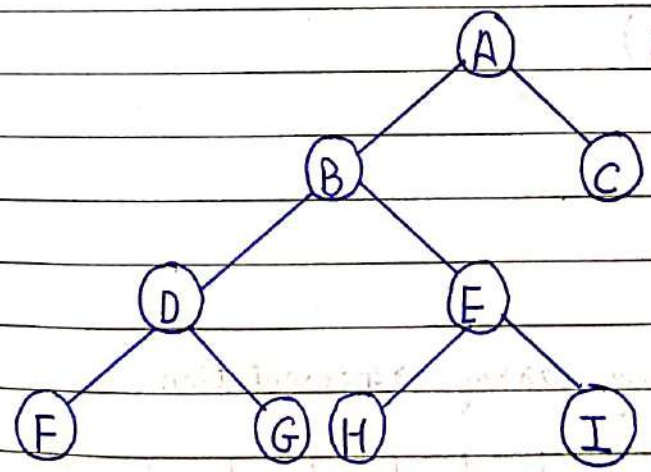
The representation of above <u>perfect binary tree</u> using array is as followed:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

We will number each node of tree starting from the root. Nodes on same level will be numbered left to right.

★ Example 2 :-

<u>Complete binary tree</u>



Here, depth = 4 (level), ∴ we need the array of size $2^4-1 = 15$. The representation of the above binary tree using array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	F	G	H	I	-	-	-	-

Level

0 1 2 3

We can apply the above rules to find array representation.

1. Parent of node E (node 5) $= \frac{i}{2} = \frac{5}{2} = 2$ i.e. B.

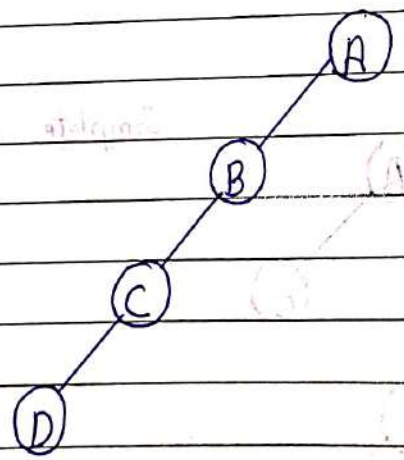
Hence, node B is at position 2 in the array.

2. Left Child $(i) = 2i$
For e.g., left child of $E = 2 \times 5 = 10$ i.e., H.
Since, E is the 5th node of tree.

3. Right child $(i) = 2i + 1$
For e.g., Right child of $D = 2 \times 4 + 1 = 9$ i.e. G.
Since, D is the 4th node of tree & G is the 9th element of array.

☆ Example 3 :-

Skewed binary tree



The tree has following array representation,

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B		C				D							

The array size is $2^4 - 1 = 15$

Thus, only 4 positions are occupied in the array and 11 are wasted.

- From the above e.g., it is clear that binary tree using array seems to be easiest, but the method has certain drawbacks. In most of the representation there will be a lot of unused space. For complete binary trees the representation is ideal as no space is wasted.
- But for the skewed binary tree, less than half of the array is used and the more is left unused. In the worst case a skewed binary tree of depth k will required $2^{k+1}-1$ locations of array and occupying only few of them.