CLASSIFICATION of DATA STRUCTURE

Date: / / Page no: 1

Short note on "C"

⇒ C is a programming language which was developed at AT and T's Bell Laboratories of USA in 1972.
⇒ It was designed and written by 'Dennis Ritchie'

⇒ The c language is a high level language.
⇒ It replaced the more formitive languages, such as FORTRAN, COBOL, PL/I and PASCAL.

⇒ C is popular because it is reliable, simple and easy to use.
∴ It can be defined by following ways.

(i) Mother language
(ii) System programming language
(iii) Procedure-oriented programming language
(iv) Structured programming language.
(v) Mid-level programming language.

Features of C language

: It provides a lot of features. C is the widely used language.

(i) simple
(ii) Machine dependent or portable.
(iii) Mid level programming
(iv) structured programming
(v) Rich Library
(vi) Memory Management
(vii) Fast speed
(viii) Pointers
(ix) Recursion
(x) Extensible

Data Structure :

- Data may be organized in many different ways. The logical or Mathematical model of a particular organization of data is know as a Data structure.
- Some example of Data structure are : arrays, Linked list, stack, Queue etc.

Basic Terminology : Following terminology is used as data structures.

1. Data
2. Group items.
3. Record
4. File
5. Attribute and Entity.

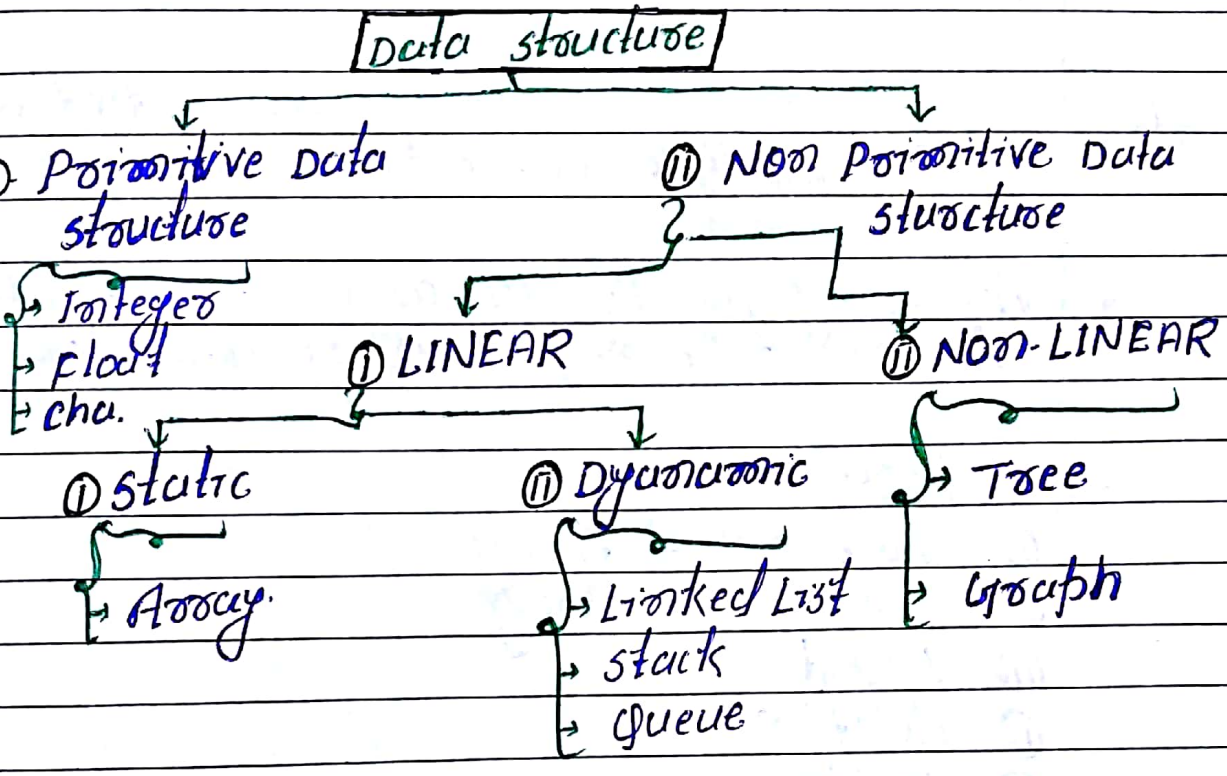
Need of Data structure : As applications are getting complexed and amount of data is day by day, there may arrise the following problems.

1. Processor speed
2. Data search
3. Multiple requests.

Advantages :

1. Efficiency
2. Reusability
3. Abstraction.

Data structure classifications: Data structure are normally divided into two broad categories.



Primitive Data structures: These are basic structure and are directly operated upon by the machine instruction.

- These have different representation on different computers. Integer, numbers, character, string etc Fall in this category.

Non-Primitive Data structures: The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.

① Arrays. ② List

Different non-primitive data structures / operations:

- Non-primitive data structures can be broadly categorized into three types. namely arrays, lists and Files.

(i) **Array:** These are data structure that can store multiple elements of same types under one name.

- An array is defined as a set of finite number of elements [Homogeneous] or data items.

operations:

(i) Traversal of Array
(ii) search an Array
(iii) Inserting
(iv) deletion
(v) sorting
(vi) Merging.

(ii) **Lists:** A list [linear linked list] can be defined as a collection of variable number of data items.

operations:

(i) Creation
(ii) Insertion
(iii) Deletion
(iv) Traversal
(v) Searching
(vi) concatenation
(vii) Display.

Date: / / Page no: (v)

(iii) **Stack** : A stack is a non-primitive data structure.

: operations :

(i) push
(ii) pop

(iv) **Queue** : A queue is a non-primitive data structure.

: operations :

(i) Insertion of an element in a queue.
(ii) Deletion of an element in a queue.

(v) **Tree** : A tree is non-linear data structure items are arranged in a sorted sequence.

: operations :

(i) Tree traversal
(ii) Insertion of nodes
(iii) Deletion of nodes
(iv) Searching for the node
(v) copying the tree.

(vi) **Graph** : A graph G consists of a set V of vertices and a set E of edges.

We write G(V.E) V is a finite and non empty set of vertices.

: operations :

(i) Searching on a graph
(ii) Inserting in a graph.

(iii) Deleting from a graph.
(iv) Traversing a graph.

(VII) Files: A file is a collection of logically related records.

÷ operation:
(i) creation of file
(ii) Reading of file
(iii) updation of file
(iv) Insertion in the file
(v) Deletion from the file.

Linear D.S	Non-linear D.S
⇒ elements form a sequentially.	• elements form a not sequentially.
⇒ Linear data structure include the following (a) Array (b) Linked list (c) stack and queues.	• Non-linear data structure include the following. (a) Tree (b) Graph

Uses of data structure ÷
(i) Identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using these entities and operations.

Abstract data types :

- Abstract data type or ADT is the abstract collection of data elements and their accessing functions.
- It is considered as a useful tool for specifying the logical properties of data type.
- The term abstract data type means the basic mathematical concept that define the data type.

: Implementation Aspects :

- Data structures are normally depended on the ability of a computer to fetch data and store data at any place in the memory.
- it is defined by pointer a bit string, representing a memory address which can be itself stored in memory and manipulated by the program.
- The output in a large amount of data to be stored. that is not acceptable.

Data structure operations : the following four operations play a major role in data processing on data structures.

① Traversing : Accessing each record exactly once so that certain items in the record may be processed.

- This accessing and processing is sometimes called visiting the record.

(ii) Searching :- Finding the location of the record with a given key value, or finding the locations of all records to the structure

(iii) Inserting :- Adding a new record to the structure.

(iv) Deleting :- Removing a record from the structure.

:- the following two operation, → which are used in special situation are.

① Sorting :- Arranging the records in some logical order.

② Merging :- combining the records in two different sorted file into a single sorted file.

Arrays :

- These are data structure that can store multiple elements of same data under one name.

country = {India, USA ---}

DS
- Linear [element form a seq.]
- Non-Linear [Rep. with either Arrays or Linked]

- An arrays elements is accessed by an index.
- Basic operation can be performed on Arrays.
  1. Traversal of Arrays
  2. Searching.
  3. Inserting
  4. deletion
  5. Sorting
  6. Merging.

Representation of Linear Arrays in Memory :

- The elements of an arrays are stored in consecutive memory locations.

$$LOC[LA[k]] = Base(LA) + w[k - lower\ bound]$$

(w: size of datatype)

- $LOC[LA[k]]$: address of k^{th} element
- $Base(LA)$: Base addr of Array
- k^{th} element no.

Date: / / Page no: X

Basic operations
- Traversal
- Inserting
- Deleting

Traversal :
→ LA . Lower bound = L.B. Upper bound = UB 5 → each element
↓ Process [Multiply each element by 2]

Algorithm :
1. Initialize counter I = L.B
2. Loop (while I ≤ UB)
3. Apply 5 to LA[I]
4. I = I+1
5. End loop
6. Exist

1	1 ← L.B
3	2
4	3
6	4 ← U.B

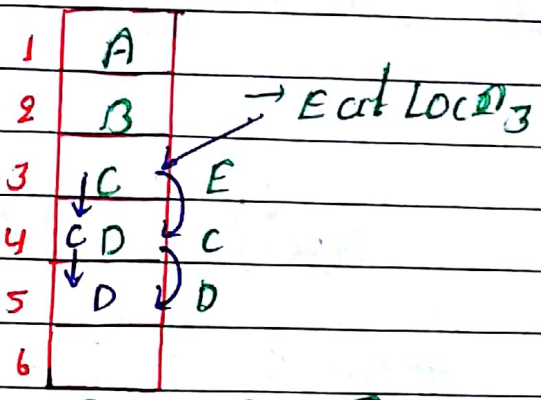
Solⁿ

I = 1	I = 3
I ≤ UB = 4	I ≤ UB = 4
5 to LA[1]	5 to LA[3]
⇒ 1 X 2 = 2	⇒ 3 X 2 = 6
I = 1+1 = 2	I = 3+1 = 4

Inserting : which is inserting an element in a linear Array of N element at Kth loc

Algorithm :
1. Initialize a counter I = N (N → No. of elements)
2. Loop [while I ≥ K]
3. LA[I+1] = LA[I]

(iv) I = I-1
(v) End loop
(vi) set LA[K] = item
(vii) N = N+1
(viii) Exit



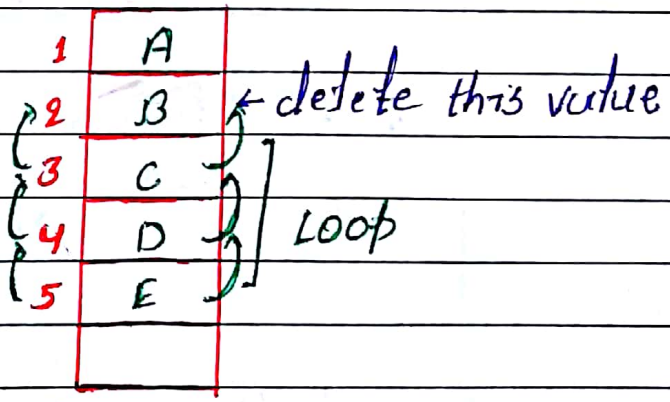
Sol:

(i)	(ii)	(iii)	(v)	(vi)	(vii)
I = N	K = 3	(i) 3	LA[5] = LA[4]	LA[3] = E	
I = 4	I ≥ K	(ii) 4	LA[4] = LA[3]	N = 4+1 = 5	

⇒ then create New array.

Deletion :

(i) set item = LA[K]
(ii) Loop for I = K to N-1
(iii) set LA[I] = LA[I+1]
(iv) End loop
(v) N = N+1
(vi) Exit.



Sol:

item = 2
I = 2 to 4
→ LA[2] = LA[3]
→ LA[3] = LA[4]
→ LA[4] = LA[5]

One Dimensional Array

- data type
- array name
- size

```
name
int a[5]
type of array  └ size of array
```

→ All elements of an array are stored in contiguous location.

a[0], a[1] ---- a[n]
└ No. of elements.

> Starting index of array = Lower Bound (L.B)
> Higher Index - 1 = Upper Bound (U.B)
> No. of elements = UB - LB + 1

∴ In 'C'

Starting location [0]

int a[10]
L.B = 0 then
U.B = 9 n = 9 - 0 + 1 = 10

Initializing 1-D Array :

→ Array are initialized as:

int a[] = {1, 2, 3, 4}

> Size of array [s] = Size of (a) / Size of variable type

Two Dimensional Array :

→ Like Matrix with rows and columns.

```
int a[5][2]
        └ 5 Rows and 2 columns.
```

	column 1	column 2
R1	a[0][0]	a[0][1]
R2	a[1][0]	a[1][1]
R3		
R4		
R5		

Initializing 2-D Arrays :

→ Arrays are initialized as

```
int a[2][2] = {{1,2},{3,4}}
```

eg :

→ Insertion in 2-D Array.

```
int a[3][4]
for(i=0 ; i<3 ; i++)
        └ no. of rows
{
for(i=0 ; j<4 ; j++)
        └ no. of column
{
scanf("%d", &a[i][j]);
}
}
```

```
a[0][0]   a[1][0]
a[0][1]   a[1][1]      } O/P
a[0][2]
   ⋮         ⋮
```

Transpose of a Matrix :

1	2
3	4
5	6

(3x2)

Transpose →

1	3	5
2	4	6

a[i][j] ------ → a[j][i]

```
for (i=0, i<3, i++)
{
  for (j=0; j<2; j++)
  {
    scanf ("%d", &a[i][j]);
  }
}
```

Memory Representation of 2-D Array :

① Row-Major order :

→ elements are stored Row wise. Row are listed on the basis of columns.

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \longrightarrow$$

Addr of elements = $\boxed{B + w[n(i-L_1) + (j-L_2)]}$

where,

B = Base addr.
W = size of element
n = no. of columns
L_1 = lower bound of Row
L_2 = lower bound of column.

Q → Arr[10:15, -2:2]

L.B of Row, U.B of Row, L.B of column, U.B of column

NR = UB - LB + 1
15 - 10 + 1
= 6

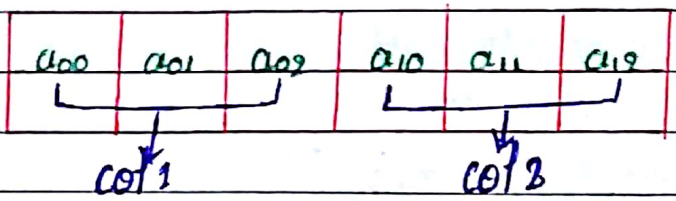
NC = UB - LB + 1
2 - (-2) + 1
= 5

→ Arr[6][5]

Date: / / Page no. VI

Column-Major order:

→ element are stored by column wise.



Add. of elements : $B + w[m(j-L_2)+(i-L_1)]$

$a[i][j]$

where,

B → Base element (add)
w → size of element
m → no. of row
L_1 → Lower bound of Row
L_2 → Lower bound of column.

eg: Given that

$B = 500$
$w = 1$ [one byte]
$L_2 = 10$
$L_1 = -20$
$add = [0][30] = ?$.

sol:

No of Row
$= L.B \text{ of } R - U.B + 1$
$20 - (-20) + 1$
$= 41$

we know that

→ $add[i][j] = B + w[m(j-L_2)+(i-L_1)]$

→ $add\ a[0][30] = 500 + 1[41(20)+20]$

= 500 + (820 + 20)

= 1340

Sparse Matrix :

→ it is 2-D Array that has min. no of non-zero elements.
→ it has majority of zero elements.
→ it Reduces scaling time.

	C_0	C_1	C_2	C_3
r_0	0	1	0	0
r_1	5	0	4	0
r_2	0	2	0	3
r_3	0	3	0	0

4×4

: No. of non-zero elements

: 3-column Rep : In this array of 3-column is used.

Total no. of rows/col/non zero elements.

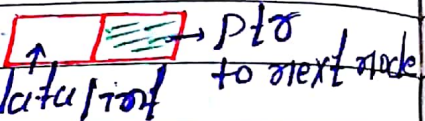
	Row	col	value
R_1	4	4	6
R_2	0	1	1
R_3	1	0	5
R_4	1	2	4
R_5	2	1	2
R_6	2	3	3
R_7	3	1	3

Linked List :-

→ Dynamic data structure → it means size of linked list can grow/shrink

→ elements of linked list are called **Node**, that contain two different

→ self Referential data types identify [points to itself in declaration]

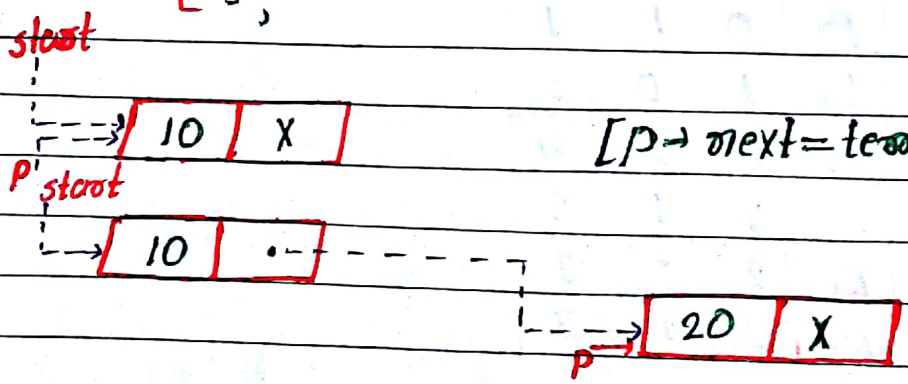


→ Insertion / Removal of nodes at any point is allowed
→ Random access of elements. it means u can insert / del. the node at → beg, end, specified Posn

:- Singly Linked List :-

→ one Link per node. which point to next node in list.

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



Advantages of Linked List :

(i) → dynamic data structure
(ii) → efficient memory utilization
(iii) → Easy insertion / deletion operation.

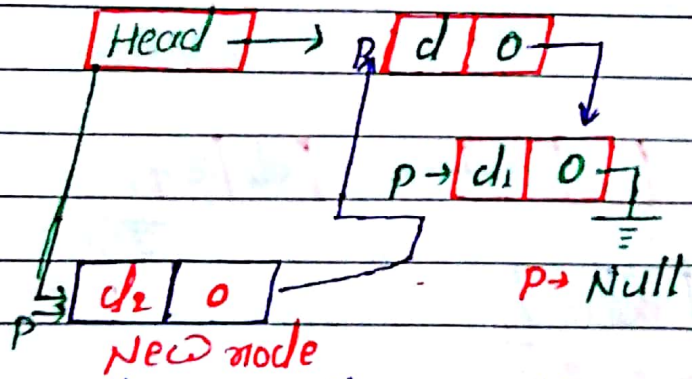
Disadvantages :

(i) → More memory
(ii) → Random access is time consuming.

Operations in Linked list :

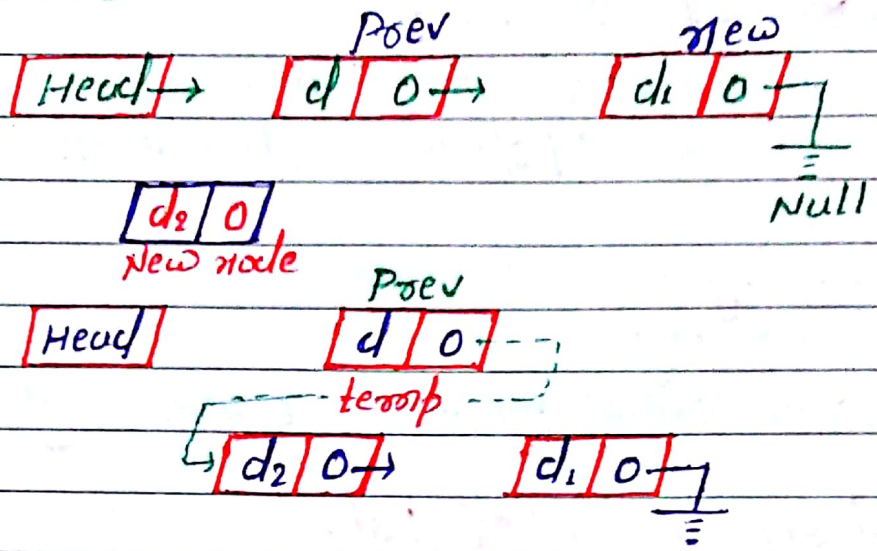
→ Insertion operation [single Linked list]

(i) Beginning :



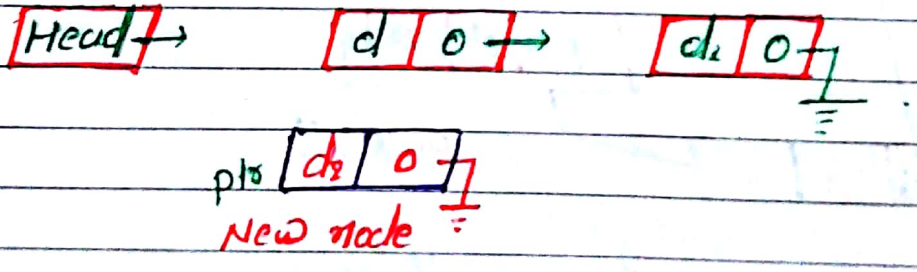
÷ if P = Null → overflow
÷ P → info = d_2
÷ P → next = Head
÷ Head = P

(ii) Middle :



: Prev → next = temp
: temp → next = new

(iii) END



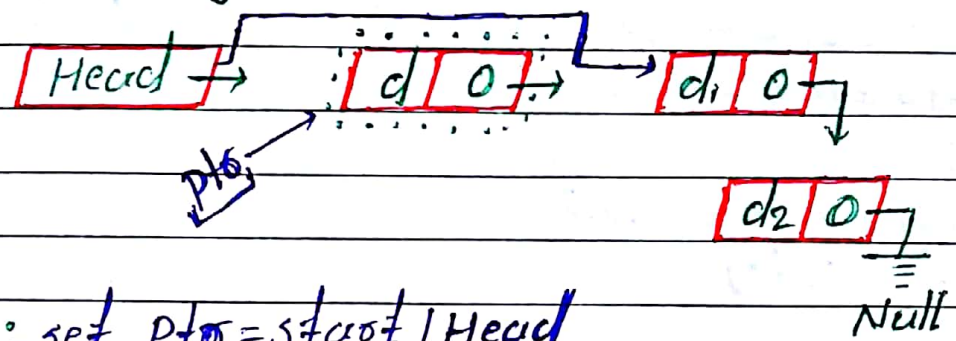
: ptr → info = d_2
: ptr → next = null] created a node

: LOC = Head
: Loop (while LOC → next != null)
: LOC = LOC → next
End loop
: LOC → next = ptr

Deletion operation :

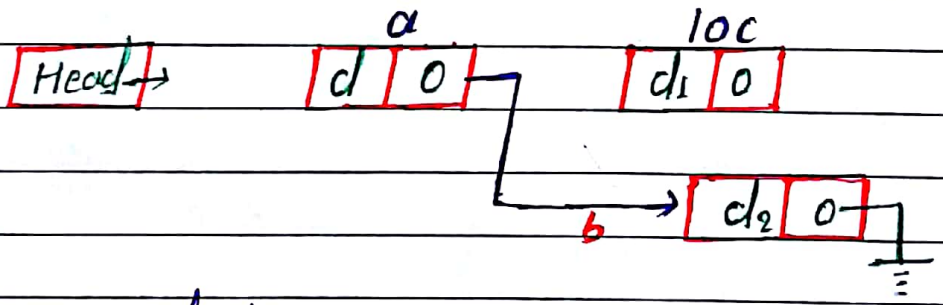
→ Insertion operation [Singly linked list]

(i) Beginning :



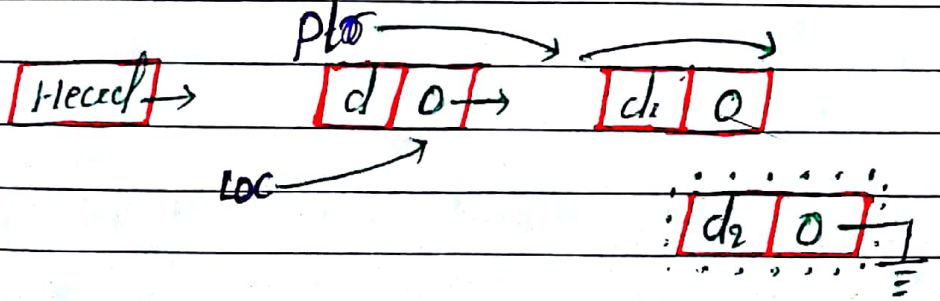
: set ptr = start / Head
: Head = Head → next
Print (ptr → info)
: free (ptr)

(ii) Middle :



: a → next = b
: free (loc)

(iii) END



÷ ptr = start / Head
÷ Loop = (Ptr → next = null)
÷ LOC = Ptr
↳ [it always one node behind ptr]
÷ Ptr = ptr → next
÷ End Loop
÷ LOC → next = null
÷ free (ptr)

Doubly linked list :

→ two way list
→ Traversal in both direction [Left to Right and R→L] is possible

prev info next

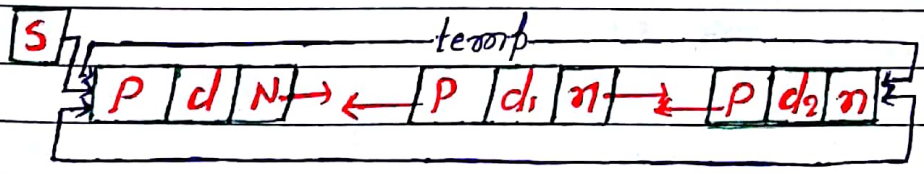
NODE → information part
→ Next pointer
→ PREV pointer

÷ Addition of first Node ÷

```
P→ info = d;
P→ prev = NULL;
P→ next = NULL
S= P;
```

Circular Doubly Linked List :

→ it is combination of circular and doubly linked list
→ it does not have any null pointer



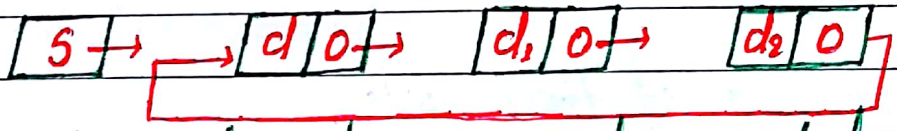
÷ The last node pointing to the first node and first node pointing to the last node.

÷ Addition example ÷

```
p→ INFO = d;      } → addition of first node
S = P;
temp→ prev = P;
P→ Prev = temp→ next;   } → addition of new node
temp→ next = P→ Prev;       "temp"
```

Circular Linked List:

→ all the nodes are linked in continuous circle without using null.

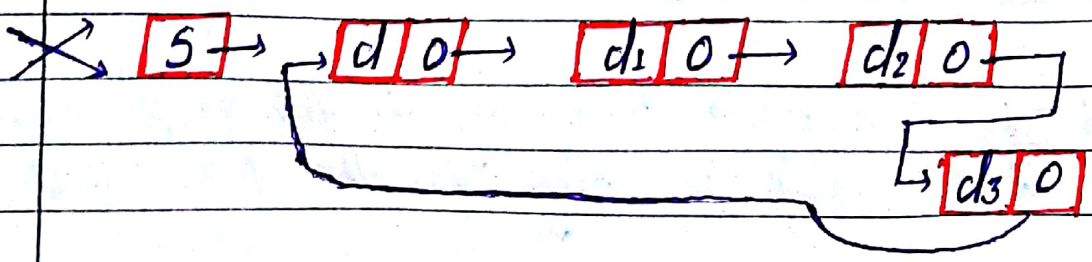


÷ Addition of node in circular Linked list ÷

```
P→ info=d;      } → initialization of linked list with
S = P;              node P.

temp→ info=d;       } → addition of new node
P → next=temp;          "temp"
temp→ next=S;

temp→ info=d3;
P→ next=temp;
temp→ next=S;
```



Linked List Vs Doubly Linked List :-

Linked List	Doubly Linked List
→ In linked list, each node has one link field.	→ In doubly linked list node has two link fields.
→ Accessibility of nodes in the forward direction.	→ Accessibility of nodes ~~in~~ can be bidirectional.
→ Insertion and deletions are simple compared to doubly linked list	→ Insertion and deletion are less simple as compared to linked list.
→ one pointer is used. So less memory is required	→ Two pointer are used. So more memory is required.

Linked List Vs Array :-

Linked List	Array
→ Linked list are dynamic data str.	→ Array are static data structure.
→ Efficient memory utilization	→ Not efficient memory utilization
→ complex application can be easily carried out	→ complex application cannot be easily carried out

Addition of two polynomials:

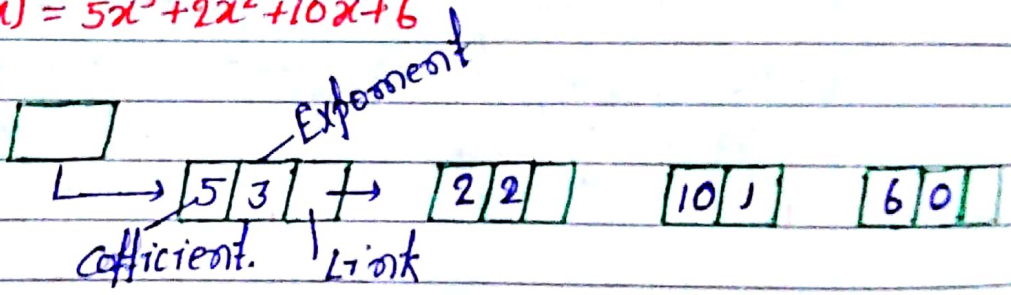
→ [Application of linked list]

$$P_1(x) = a_n x^e + a_{n-1} x^e{}_{n-1} + \cdots - a_1 x_1^e + a$$
$$P_2(x) = b_n x_n^e + b_{n-1} x^e{}_{n-1} + \cdots - b_1 x_1^e + b$$

$P_1(x) + P_2(x)$ = Linked lists

NODE → coefficient field
→ Exponent field
→ Link field.

eg: $P(x) = 5x^3 + 2x^2 + 10x + 6$




```
// struct poly
{ int coff;
  int expo;
  struct poly *ptr;
};
```

Algo.

(i) Read no. of terms in first polynomial 'p'
(ii) Read coff and exponent of first polynomial.
(iii) Read no. of term in second polynomial q
(iv) ---------- q

(v) set temp pointers 'p' and 'q' to traverse two polynomial resp.

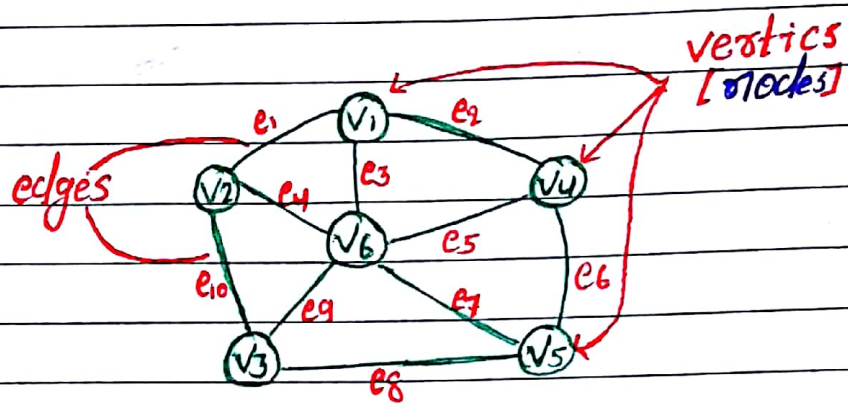
(vi) compare the exponents of two polynomials starting from first node.

÷ GRAPHS ÷

Date: / / Page no: 1

GRAPHS ÷

→ contains finite set of vertice and edges.
→ denoted as G(V.E)



÷ $V_G = \{V_1, V_2, V_3, V_4, V_5, V_6\}$
÷ $E_G = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$

(II) Terms ÷

(I) **Adjacent vertices** ÷ V_i and V_j are adjacent if there is an edges between V_i and V_j

(V_1, V_4) adj
(V_5, V_4) adj

(II) **Path** ÷ combination of edges more than one path between two vertices.

$[V_1 \rightarrow V_5]$
$e_2 \rightarrow e_6$
$e_3 \rightarrow e_7$

(III) **Cycle** ÷ path where first and last vertices are same.

$[V_1 \rightarrow V_6 - V_4]$

(iv) **Connected Graph** :- if there is exists a path b/w any two of its nodes.

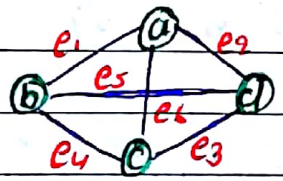
$[V_1 \xrightarrow{V_2} V_5]$

(v) **Complete Graph** :- if every node is connected to its adjacent node.

→ No. of edges in a complete graph with n node is.

$$\frac{n[n-1]}{2}$$

eg :-



Soln:-

→ Given that

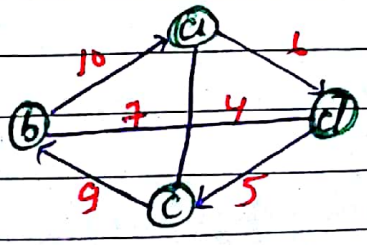
No. of node = 4

$n = 4$

$$\frac{4[4-1]}{2} = 6$$

Hence the no. of edges = 6.

(vi) **weighted Graph** :- if every edges in the graph is a assigned some weight or value.

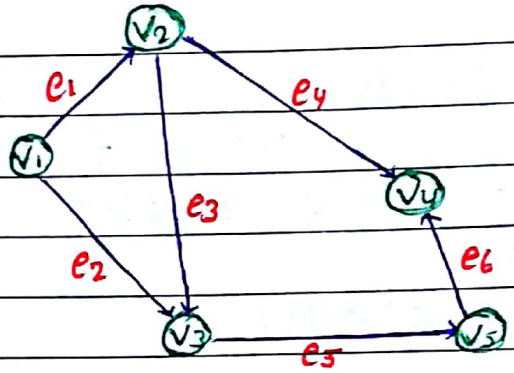


To finding :-

→ shortest path b/w two node

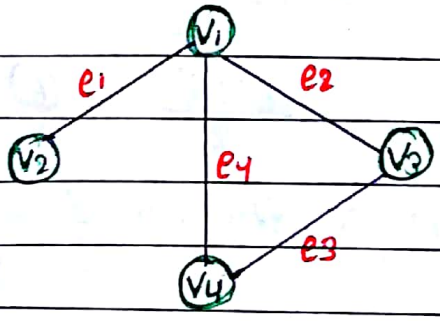
→ drawing MST

(vii) **Directed Graph** ÷ if each edge is assigned a direction.
→ it is also referred to as diagraph.



÷ vertex pairs (v_i, v_j) reads as $v_i - v_j$ and edge is directed from v_i to v_j

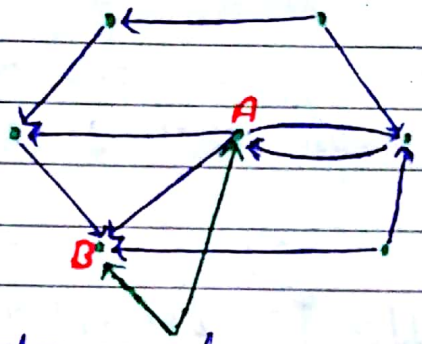
(viii) **Undirected Graph** ÷ A graph is called an undirected graph when the edges of graph are unordered pairs.



÷ set of vertices $V = \{v_1, v_2, v_3, v_4\}$
÷ set of edges $E = \{e_1, e_2, e_3, e_4\}$

Weakly connected graph :- A digraph can be weakly connected if you substitute each directed edge with an undirected edge.

→ A connected graph in which there is a path that connects each pair of vertices in both direction.



∴ A can reach B but B cannot reach A unless we transform the graph into a logically undirected graph.

Walks :-

Representation of Graph :

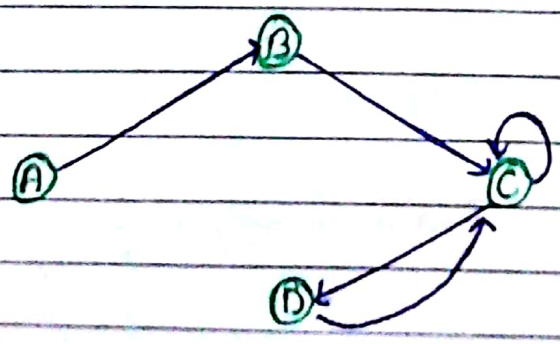
→ There are three major approaches to represent graph.

① **Adjacency Matrix Representation :** consider a graph G with set of vertices and set of edges G[V, E].

→ if there are N nodes in V(G) for N >= 1. the graph G may be represented by an adjacency matrix which is table with N rows and N columns.

$$A[i, j] = \begin{cases} 1. \text{ if and only if there is an edge } [V_i, V_j] \text{ in } E(G) \\ 0. \text{ otherwise.} \end{cases}$$

~~eg~~ :



Sol^n :

	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	0	0	1	1
d	0	0	1	0

→ Adjacency matrix of graph

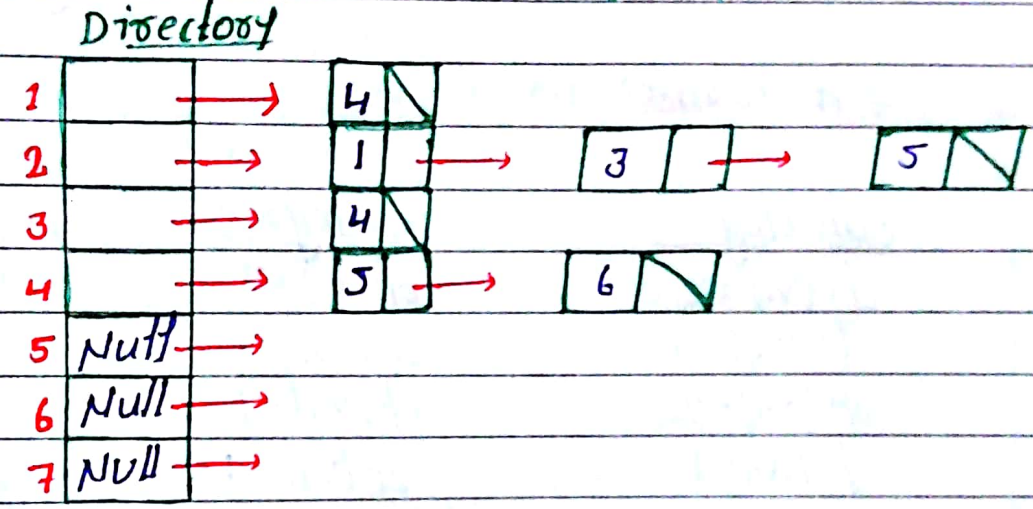
Date: / / Page no: (VI)

(iii) Adjacency List Representation :

→ In this representation a graph is stored as a linked structure.

→ it will represent graph using adjacency list.

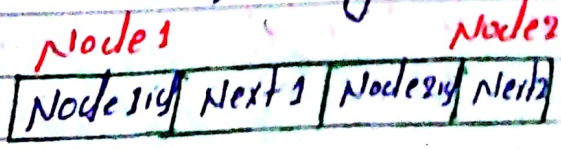
→ The adjacency list contains a directory and a set of linked list. This representation is also know as node directory representation



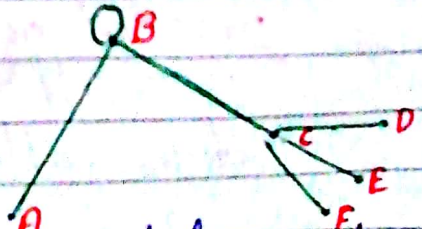
(iv) Multi-list Representation : A multi-list representation of graphs consists of two parts.

(i) A directory of node information and a set of linked lists of edge information.

(ii) The node directory contains one entry for each node of the graph.



(a) Sample Node for Edge [Vi, Vj]



(b) Undirected Graph.

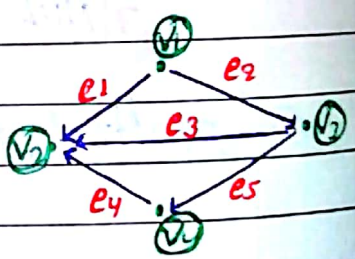
Out-degree of v_i ÷ The number of edges that are incident out of a vertex v_i are called the out-degree of v_i

÷ it written as $d^+[v_i]$

In-degree of v_i ÷ The number of edges incident into v_i are called the in-degree of v_i.

÷ it written as $d^-[v_i]$

out-degree	in-degree
$d^+(v_1) = 2$	$d^-(v_1) = 0$
$d^+(v_2) = 0$	$d^-(v_2) = 3$
$d^+(v_3) = 2$	$d^-(v_3) = 1$
$d^+(v_4) = 1$	$d^-(v_4) = 1$



÷ in directed graph. the sum of all in-degree is equal to the sum of all out degree.

$$\sum_{i=1}^{n} [d^+(v_i)] = \sum_{i=1}^{n} [d^-(v_i)]$$

Directed acyclic graph (DAG) ÷ A path from node to itself is know as cycle. if graph has cycle it is called cycle. else it called acyclic.

→ DAG is a short form of direct acyclic graph.

* ÷ The adjacency matrix of undirected graph is symmetric

* ÷ The adjacency matrix A is asymmetric for directed graphs and in the general case its eigenvalue are difficult.

÷ Three properties can be read off the difficult eigenvalue.—

(I) → acyclic graph
(II) → symmetric
(III) → Bipartite.

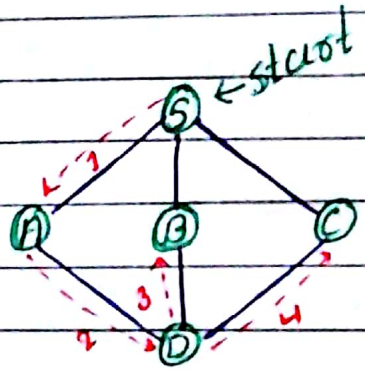
GRAPH TRAVERSAL :

→ A graph traversal means visiting of the all nodes of the graph.
→ It use preorder, postorder or inorder to examine the vertices.
→ It is two techniques for examination of the graph.

① Depth First Traversal :

→ Traverses a graph in depth.
→ stack is used for implementing.
→ Depth first traversal of an undirected graph is similar to preorder traversal of an ordered tree.

① Initialize stack



Algorithm :

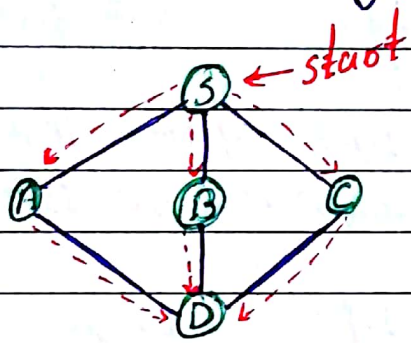
① → visited(v) = TRUE
② → visit(v)
③ → for each vertex w adj to v do
④ → if not visited(w)
⑤ → traverse(w);
⑥ → end;

B → Pop		C
D		D
A	→	A
S		S

↑ stack

(1) Breadth First Traversal :

→ Traverse a graph in breadthward motion
→ Queue is used
→ This traversal algorithem uses a queue to store the nodes of each level of the graph as and when they are visited.



① Initialize Queue

A → adj.
B | A
C | B | A

↳ deque

C | B
D | C | B

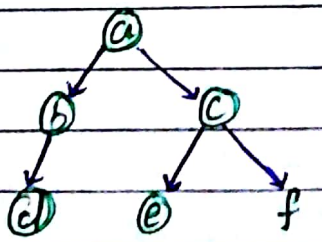
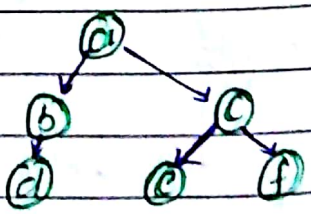
↳ deque

Algorithm :

(i) → create q (q);
(ii) → visited (v) = TRUE;
(iii) → while not empty (q) do
for all vertices w adj to v do
(iv) → if not visited then.

```
{
    insert (w, q)
    visited (w) = TRUE
}
delete (v, q);
```

BFS Vs DFS

÷BFS÷	÷DFS÷
→ BFS can be done with the help of queue.	→ DFS can be done with the help of stack
→ BFS is slower than DFS	→ DFS is more faster than BFS
→ BFS require more memory compare to DFS	→ DFS require less memory compare to BFS.
→ BFS is useful in finding shortest path.	→ DFS is not so useful in finding shortest path.
→ This algorithm works in single stages.	→ This algorithm work in two stages.
→ BFS is similar to level-order traversal	→ DFS is similar to pre-order traversal.
→ 	→ 

Minimum Spanning Tree [MST] :

→ A spanning tree in a graph G is a minimal subgraph connecting all the vertices of G

→ A spanning tree with the minimum weight in a weighted graph is called minimal spanning tree or shortest spanning tree or minimum cost spanning tree.

→ These methods for finding a minimal spanning tree in a graph.

(I) Kruskal Algorithm :

→ Builds minimum spanning tree of a graph by adding edges to the spanning tree one by one.

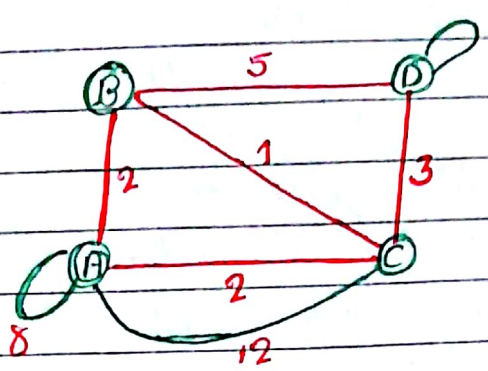
Pseudo code / Algorithm :

(i) → For each vertex v in G do
(ii) → create a set [actually tree] with element v
(iii) → Initialize a priority queue Q containing all edges in descreasing order of their weight.
(iv) → Define FOREST ← ϕ {contain all edges of MST}
(v) → while Tree has edges L= N-1 do, No. of vertices.

(vi) → select edge with minimum weight.
(vii) → if T[v] ≠ T[u] then
→ Add [u.v] to forest
→ Union [T[v]] and [Tu]

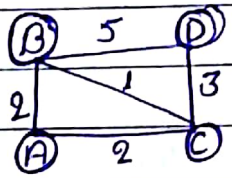
(viii) → Return T

Eg:- Graph with loops and parallel edges.



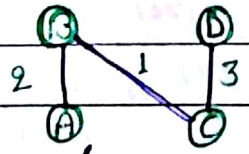
Sol:-

firstly Remove loops and parallel edge.



then,

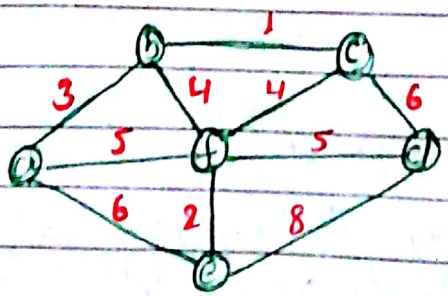
draw minimum edges.



→ sum of edges = 2+1+3
= 6

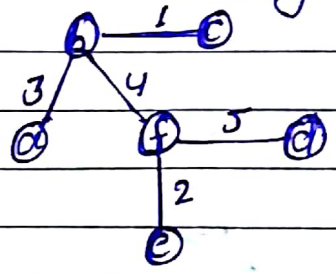
[Disjoint Tree/set]

eg:-



Sol:-

Draw the minm. edges.



→ sum of edges = 1+2+3+4+5

= (15)

(ii) PRIM'S Algorithm :- (MST) is defined for weighted graph

→ used to draw minimum spanning

↳ spanning tree of a graph consists of all vertices and some of the edges so that the graph not contain a cycle.

Algorithm:-

→ Let G(V, E) is a connected weighted undirected graph.

(i) → create two sets V and V' such that
- → V' is empty
- → V contain all vertices of graph
- → select minimum weighted edges (i, j) from G and insets i and j into the set V

(ii) → Repeat step 3 while V is not equal to V'

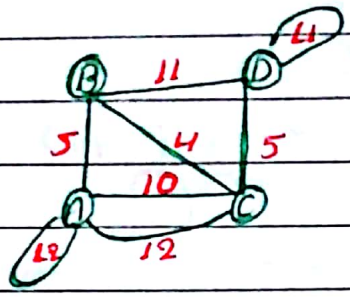
(iii) → Find all neighbors of all vertics which are in set V' such that one end point of neighbors

edge is in V' and another not in V'

(iv) → sum of all selected edges weight (s)

(v) → Exist

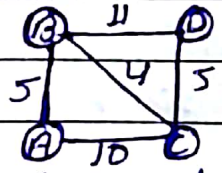
eg :- Loop parallel edges.



Solution

(i) firstly remove loops

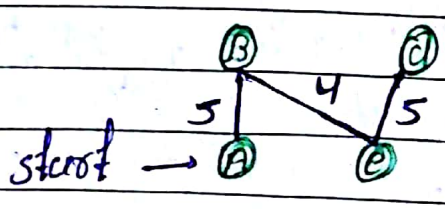
(ii) Remove parallel edge.



→ set of vertices V = {a, b, c, d}

then

draw Min edges.



→ sum of edges = 5+4+5

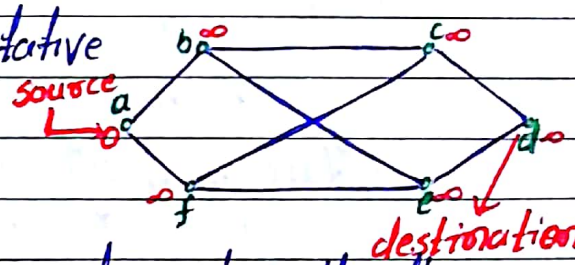
= (14)

Dijkstra's Algorithm :-

→ used to find shortest path b/w nodes in a graph.

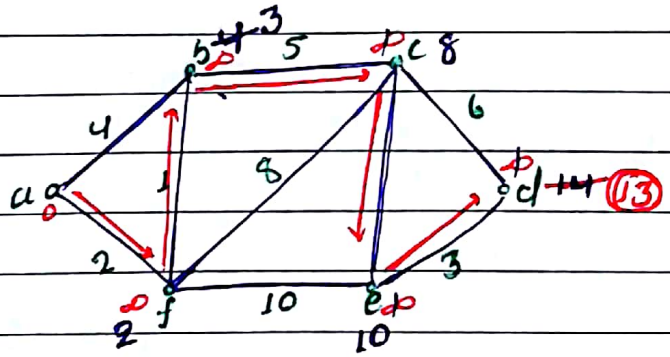
Steps :-

(i) → Assign every node a tentative distance.



(ii) → set initial node as current and make all other node as unvisited.

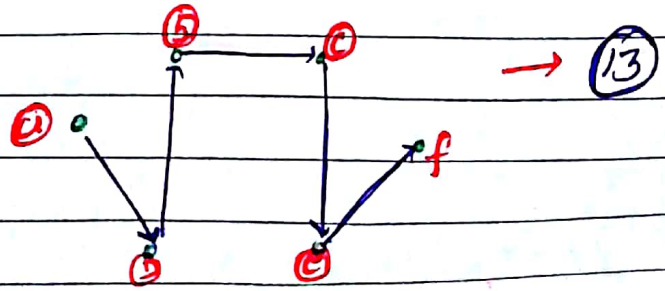
(iii) → For current node, consider all unvisited nodes and calculate tentative distance, compare current distance with calculated distance and asign the smaller value.



(iv) → when all the neighbors are considered of the current node mark it visited [visited node is never check again]

(v) → if the destination node is worked visited steps.

(vi) → END



Date: / / Page no:

Kruskal Vs Prim's :-

Kruskal	Prim's
→ Kruskal's algorithm is an algorithm in graph theory spanning tree for a connected weighted graph.	→ Prim's algorithms is the algorithm which searches a min. spanning tree for a connected weighted un-directed graph.

Date: __/__/____ Page no: _____

Application of Graph :

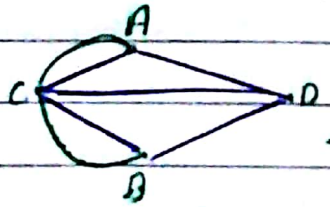
→ Graphs have found application in subjects like Geography, Chemistry, Engineering, social and biological sciences in numerous other area.

→ In computer science graphs are used in computer design etc.

→ A graph can also be used to represent physical * situation involving discrete objects and a relationship among them.

→ The following are the examples of such appln

① Konigsberg Bridge Problem :



→ Graph.

⑪ Utilities problem :

÷ TREE ÷

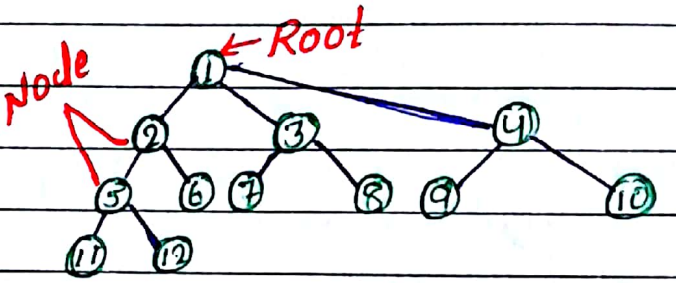
Date: / / Page no: 1

Tree ÷

→ Non-linear data structure.
→ A tree is defined as a finite set of one or more data items (nodes)

(*) Basic Terms ÷

(i) ROOT → First node
(ii) NODE → each data item
(iii) Degree of NODE → No. of subtree of a node.



÷ Degree of 1 = 3
÷ Degree of 2 = 2
÷ Degree of 3 = 2

(iv) Degree of a Tree → Maximum degree of node in a given tree.

÷ Degree of tree = 3 | Node → 3

(v) Terminal Node ÷ Node with zero degree.

÷ Node 6 = 0

(vi) Non-Terminal Node ÷ Any node except root node and whose degree is not zero

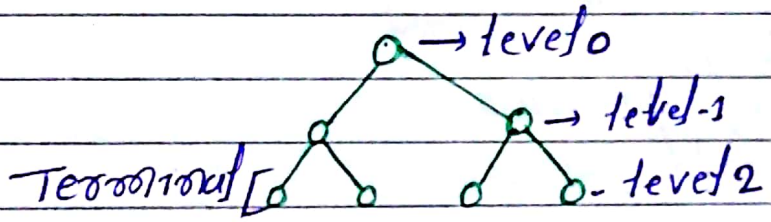
(vii) Edge → line that connects node.

(viii) Path → sequence of consequetive edge of source to destination.

(ix) Depth (Height) → Max. level of any node in a tree

OR

it is the no. of levels form root node to Terminal node.



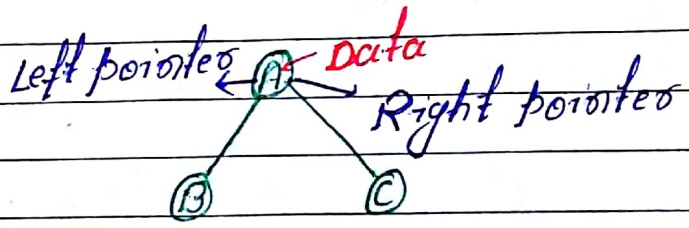
Depth = 2

(x) Order of tree : order of a tree defined as the minm. no. of keys in nonroot node

$\lceil m-1 \rceil / 2$

BINARY TREE :

→ Tree with finite no. of elements.

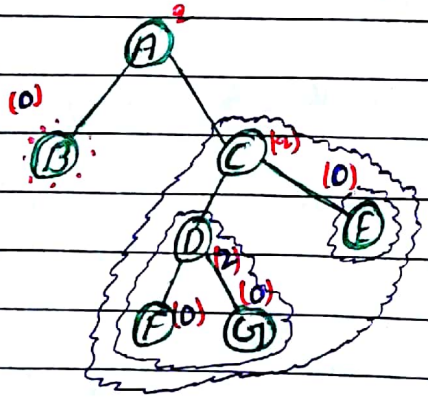


→ Node contain following infon
- ↳ Data
- ↳ Left pointer
- ↳ Right pointer

Strictly Binary Tree :

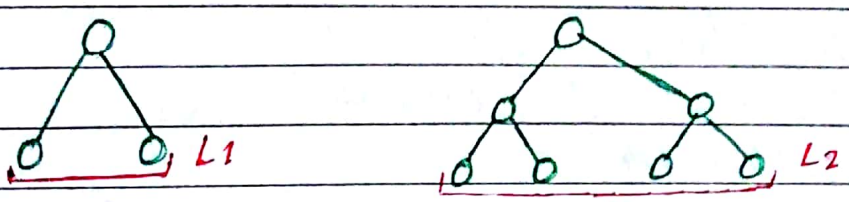
→ it is a binary tree with every node 'N' has either 0 to 2 tree.

→ Also know as extended 2-Tree or simply 2-Tree



Complete Binary Tree

→ All levels of the tree reside at the same level. ↳ terminal node.



∴ Formula for maximum of nodes for depth d

$$T = 2^0 + 2^1 + 2^2 + \dots 2^d$$

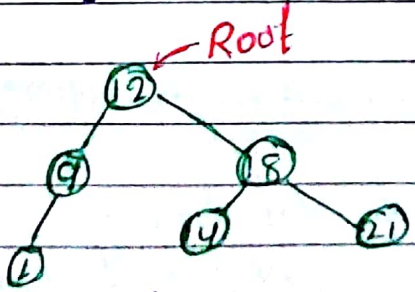
$$T = \frac{1 + 2(2^d - 1)}{2 - 1}$$

$2^n \rightarrow$ Level

$$\boxed{T = 2^{d+1} - 1}$$

Binary search Tree :

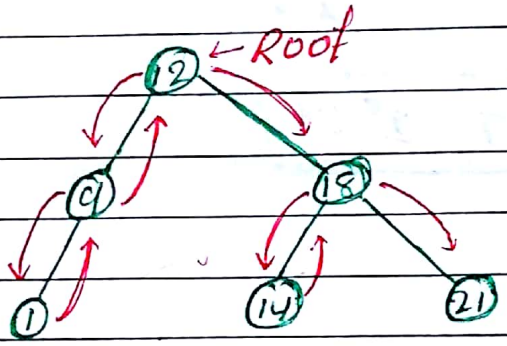
→ In Binary search tree, nodes left child must have value less than its parent value and nodes right child must be have value greater than it's parent value.



→ Binary search tree used mainly for searching purpose.

÷ Basic operations of BST

(i) Inserting a elements
(ii) Deletion of element
(iii) Searching
(iv) Sorting → it can be done by traversing a BST in Inorder
↳ Left Data Right

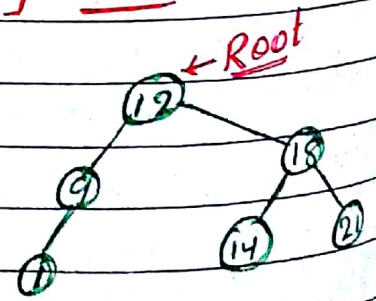


1	9	12	14	18	21	→ Inorder

Searching in BST ÷

Algorithm ÷

if root == search data return root
else
→ while data not found
→ if search data > node data] 14>12
→ goto Right subtree
else
→ goto left subtree
→ if data found
→ return node
end while
→ return data not found
end if



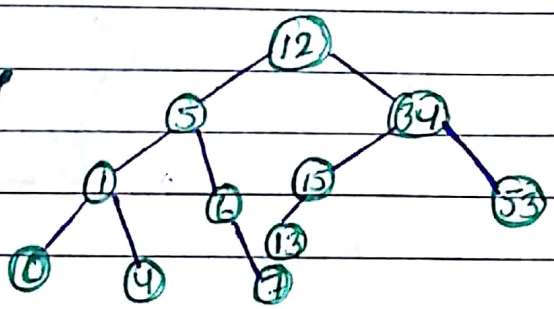
(II) Insertion in BST

Algorithm :

```
if root == NULL
→ create root node
→ return
→ if root exists then
→   if data > node
→   goto right subtree
  else
→   goto left subtree
  else if
→   insert data
  end if
```

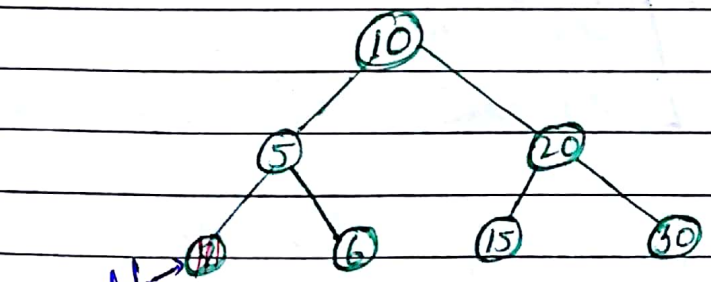
eg :- 12, 34, 5, 1, 15, 6, 7
0, 13, 4, 53

soln.



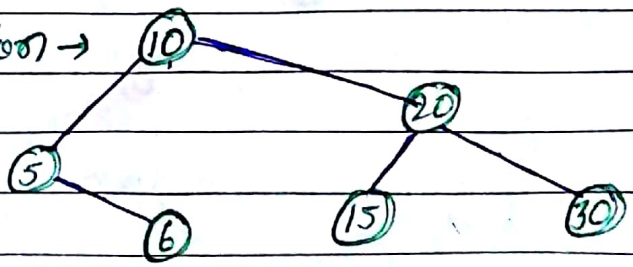
(III) Deletion in BST

(*) Case-1 :- if node to be deleted is a leaf node
[simply delete the node] ↳ node has no child



if is del. node not

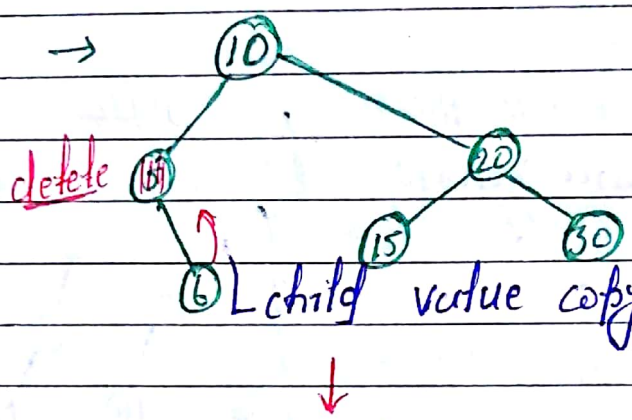
after deletion →



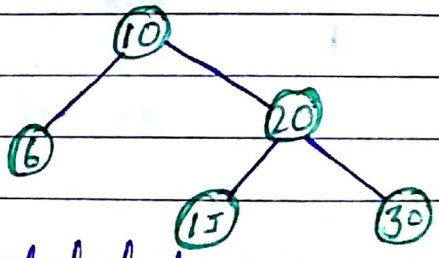
* Case-2 : if the node to be deleted has only one child.

[copy child to node and delete child]

→



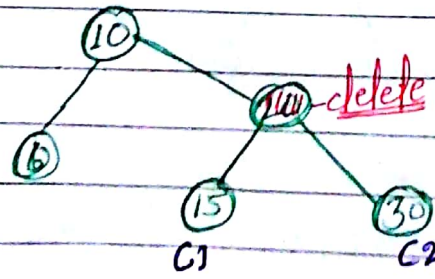
after deletion →



* Case-3 : if the node to be deleted has two children nodes.

[find inorder successor of node
then copy content of inorder successor
to node and delete inorder]

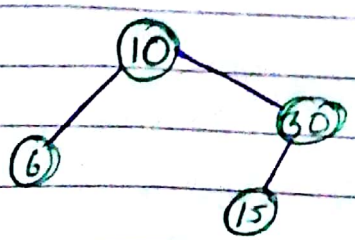
→



inorder case :

LDR → 6, 10, 15, 20, 30

20 is inorder successor of 30



→ after deletion

Binary Tree Traversal :

→ Traversal of a binary tree is to access every node of binary tree at most once.

Tree Traversal ways :

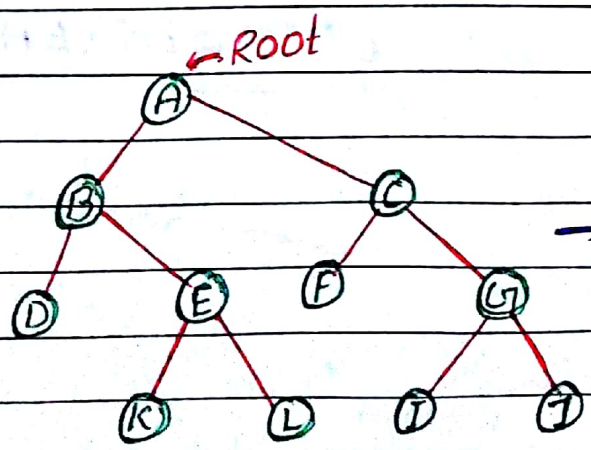
(i) Preorder → Data, Left, Right → DLR

(ii) Inorder → Left, Data, Right → LDR

(iii) Postorder → Left, Right, Data → LRD

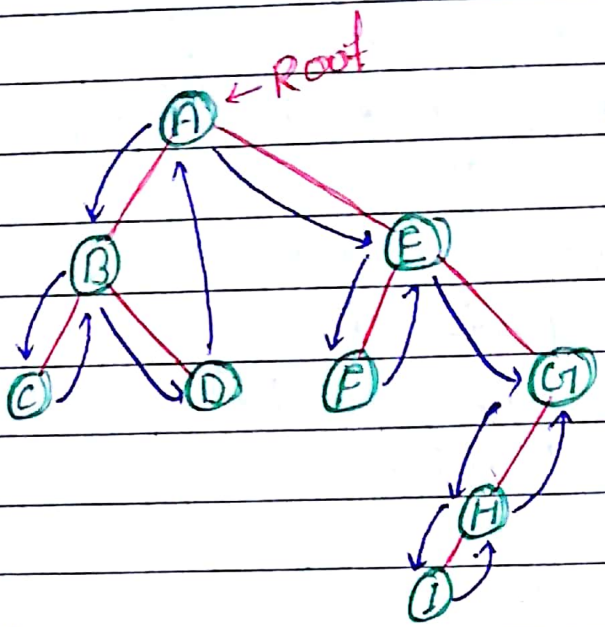
(iv) Level order

eg : preorder Traversal

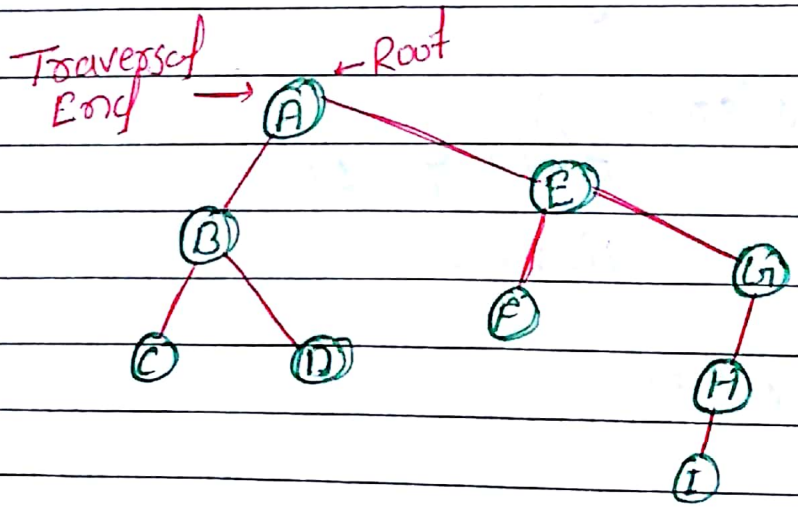


→ preorder → ABDEKLCFGIJ

eg :- Inorder, Postorder Traversal



Inorder → LDR → C B D A F E G I H G



Postorder → LRD → C D B F I H G E A

AVL TREE :

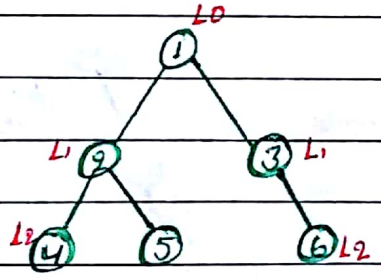
→ Example of Balanced Binary search tree
→ Invented by Adelson, Velski and Landis.

→ Def: AVL checks the height of left and Right subtree and assures that the difference is not more than 1

Balance Factor = Height[leftsubtree] - Height[R.S]

: difference can be either 0 or atmax 1

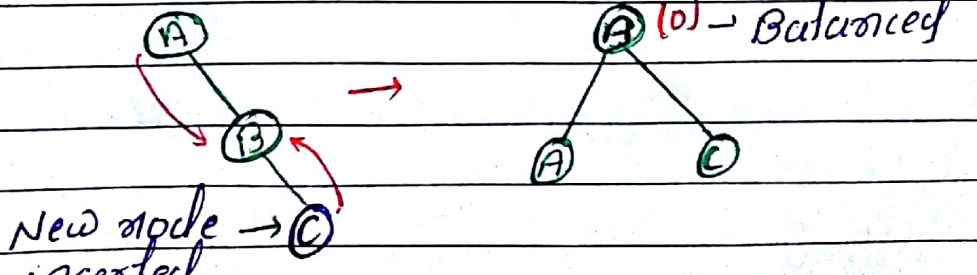
→ Time complexity in searching = $O[\log_2 n]$



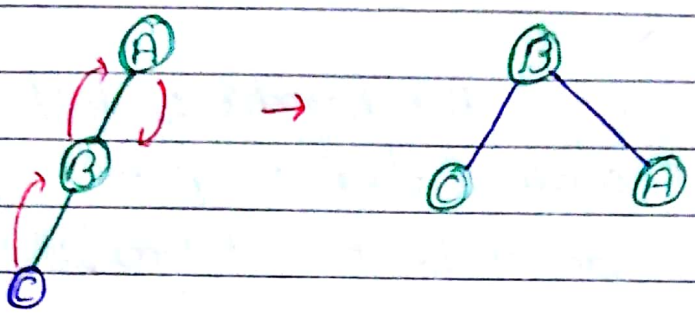
Difference 2-2=0

AVL Rotations [from AVL to AVL]

① Left Rotation : Used when node is inserted into right subtree of right subtree.
: After inserting new node tree becomes unbalanced.

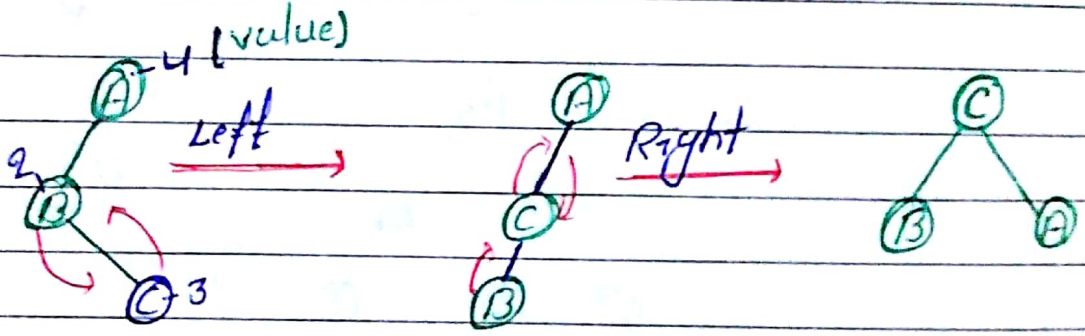


(ii) **Right Rotation** :- Node is inserted in the left of left subtree.

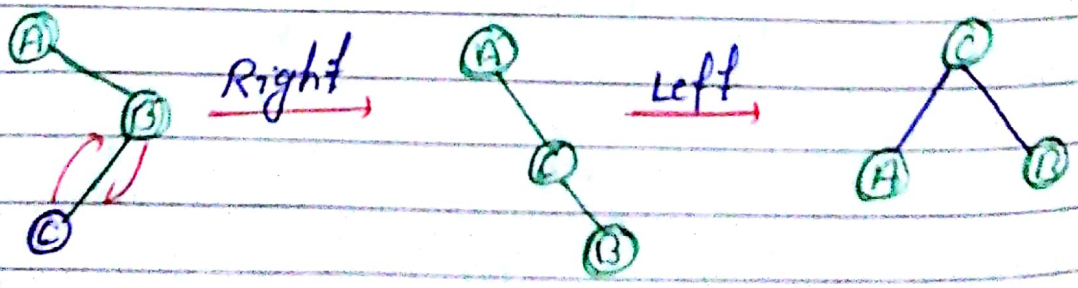


(iii) **Left-Right Rotation** :- [Double Rotation]

→ Node is inserted in the right of left-subtree and makes the tree unbalanced.



(iv) **Right-Left Rotation** :- Node is inserted into left of right subtree.



AVL tree insert Method :

→ A general algorithm for inserting a node into a height balanced tree is as follows :-

(i) → if this is the first insertion then allocate a node set its fields and exit.

(ii) → if the name is already in the tree then attach the new node to the existing node to the existing tree.

else.

(iii) search for an unbalanced node.

(iv) Adjust the balance factors. if there is no critical node exist.

(v) Rebalance the tree and exist

Deletion operation :

→ In the deletion of a node three cases arise

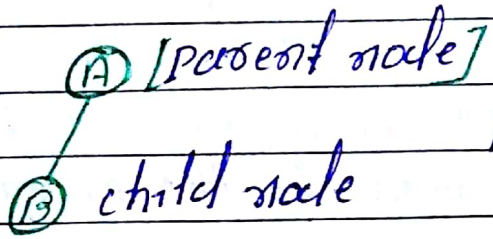
(i) Deletion of left node

(ii) Deletion of node that has one child

(iii) Deletion of node that has two children.

Heap Data Structure :

→ Special Balanced Binary tree data structure where root node is compared with its children and arranged accordingly.



Types of Heap :

(i) **Max Heap** : value of parent node is greater than that of child node.

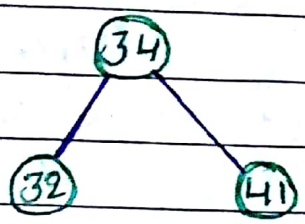
(ii) **Min Heap** : value of Root node is less than or equal to either of its children.

Construction of Max-Heap :

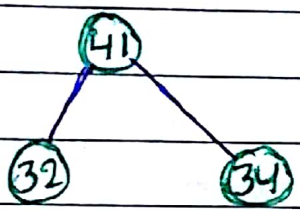
(i) create new node in the heap.
(ii) Assign value to the node.
(iii) compare value of child node with the parent node.
(iv) if parent < child then swap them.
(v) Repeat step 3, 2, 4 until Heap property holds.

eg: 34, 32, 41, 9, 13, 18, 26, 43, 25, 30

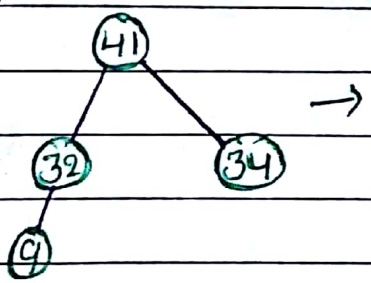
Soln



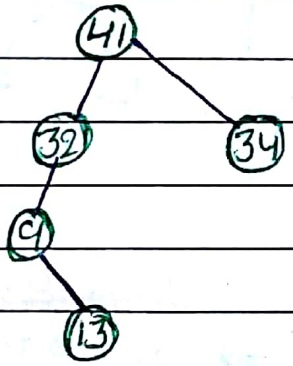
÷ if parent < child → swap



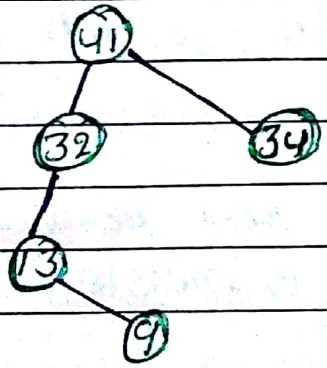
÷ insert 9



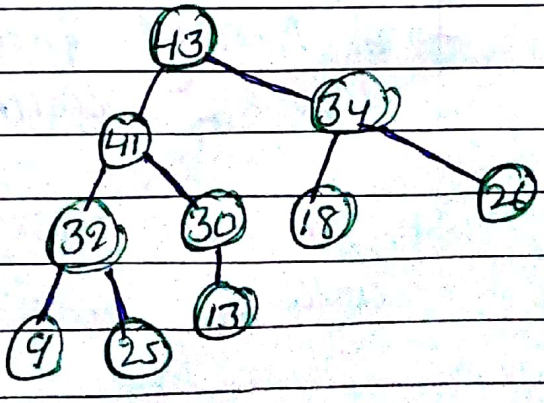
÷ insert 13



÷ if parent < child → swap

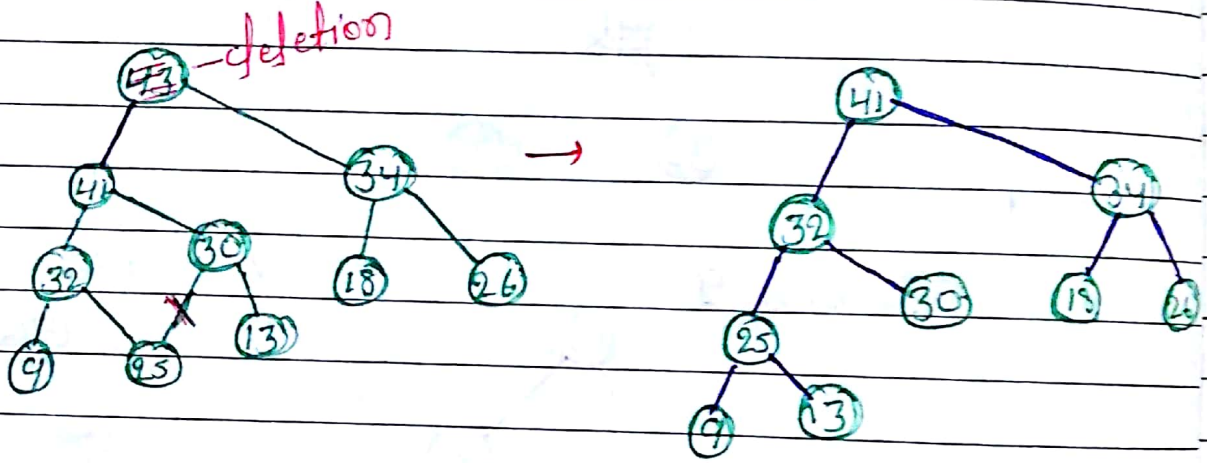


÷ insert 18



Deletion in Max-Heap : (Removal of node always occurs from root)

(i) → Delete Root node
(ii) → Move last element of last level to root
(iii) → compare value with child node
(iv) if parent < child
→ swap
(v) Repeat step 3 and 4



Applications of Trees :

→ A Binary search tree is an appropriate data structure for searching a value for application in which search time must be minimized.

→ A Binary tree may also be used for manipulation of arithmetic expression.

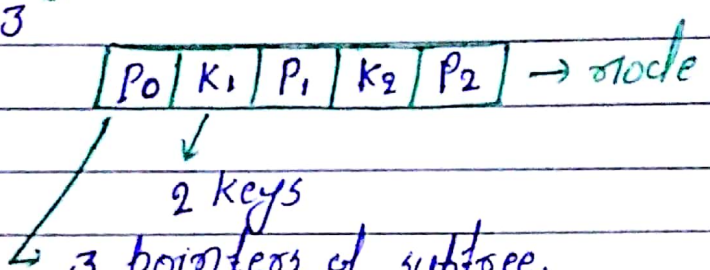
→ The tree also play an important role in the area of syntax analysis.

Multi Way :-

→ it is similar to Binary search tree
→ it has m no. of pointers = no. of subtree
→ it has (m-1) keys no. of in one node.

eg :- 3' way search tree.

m = 3

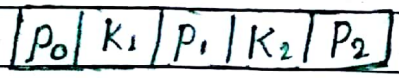


:- each key has 2 pointer one is left or one is Right.

:- left pointer :- it should always pt to values less than key / info.

:- Right pointer :- greater then key or info

Representation



$K_1 < K_2 < \dots < K_{m-1}$

:- each P_i is a pointer to a subtree and $0 \le i < m-1$

B+ Tree :

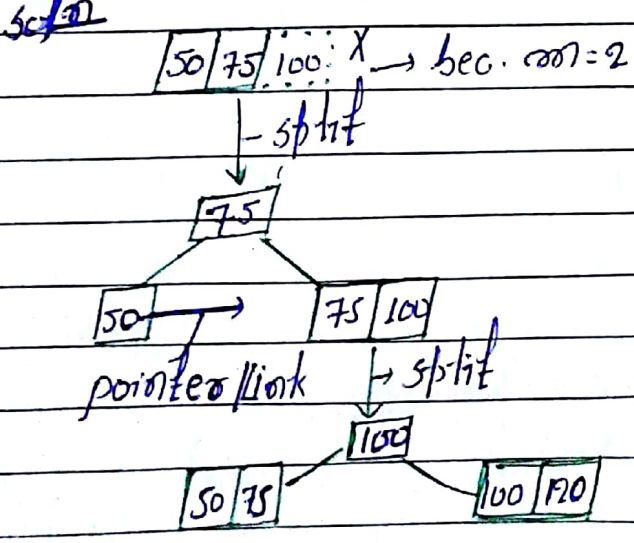
- It is used for database
- Best suited for Range queries.
- All the data is stored for leaf node.
- every leaf is at the same levels.
- all the leafs have pointers / links with each other.
- The B+ tree contains two parts.
  (i) index part
  (ii) sequence part
- The technique for implementing indexed sequential organization is use to the variation on the basic called B+ tree.

Threshold level(m) = max no. of elements at node.

: if m=2, that means
↳ 2 elements at a node

eg : 50, 75, 100, 120 - - - -

m=2



B-Tree.

- B tree is a balanced M-way tree.
- B-tree is also called the balanced sort tree.

(*) **Order** : order of a B-tree defined as the minimum no. of keys in a nonroot node [that is $\rightarrow \lfloor m-1/2 \rfloor$]

(*) **Degree** : Degree of a B-tree is the maximum no. of sons [i.e - m]

(*) **Characteristics** :

(i) All internal nodes exept the root node must have minimum $\lceil \frac{m}{2} \rceil$ non empty child nodes and max. [m] non empty child nodes.

(ii) Every external nodes should be at the same level.

(iii) Internal node contains one less than key than its no. of child nodes.

(iv) The root node must contain minimum two child nodes and maximum [m] child nodes

B-Tree Vs B+ Tree :

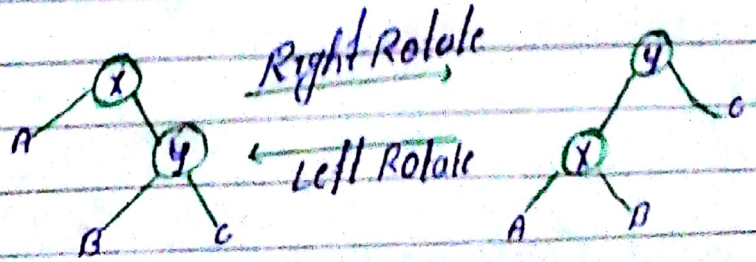
	B-Tree	B+ Tree
(i)	In B-tree all the levels have not been connected to form a linked list of the keys.	In B+ tree all the levels have been connected to form a linked list of the keys.
(ii)	The keys can only be accessed randomly.	B+ tree retains random property of the B-tree. It also allowing rapid sequential access.
(iii)	In the B-tree all keys are not maintained in leaves.	In the B+ tree all keys are maintained in leaves.

Red Black Tree :

A red black tree is balanced binary search tree with the following properties.

(i) Every node is colored either red or black
(ii) Every leaf is a NULL node, and is colored black
(iii) If a node is red, then both it children are black
(iv) Every simple path from a node to a descendant leaf contains the same no. of black nodes.
(v) A red black tree with n internal nodes height at most $2 \log (n+1)$

* Rotate.



STACKS AND QUEUES

Date: / / Page no:

STACKS :

- A data structure in which last item inserted is taken out first

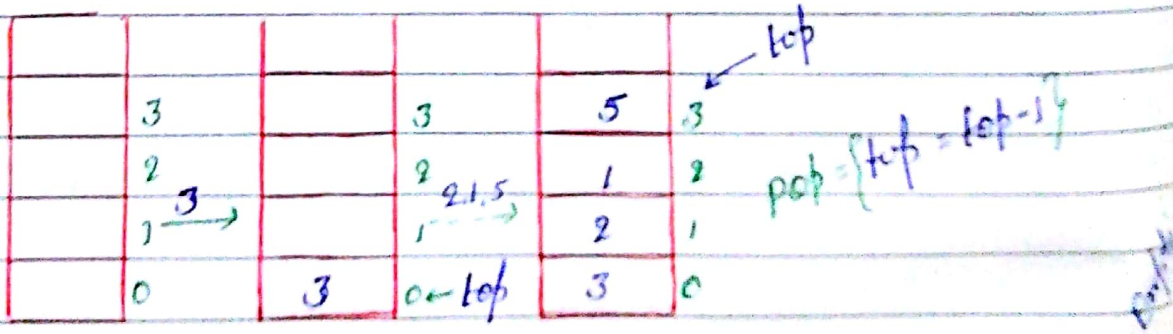
LIFO - [Last in first out]

- only one item inserted / PUSH at a time on top of stack
- only one item deleted / POP at a time from top of stack
- A stack is a non-~~linear~~ primitive linear data structure
- it is operate two operations

(i) PUSH
(ii) POP

→ The process of adding a new element to the top of stack is called PUSH

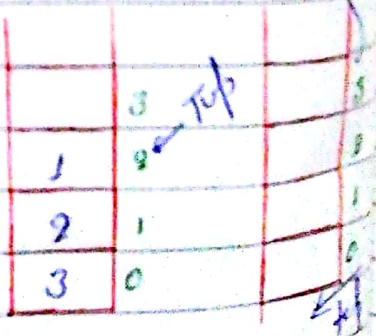
→ The process of deleting an element from the top of stack is called pop operation.



÷ when stack is empty

top = -1

PUSH = top+1



Date: / / Page no:

PUSH operation :

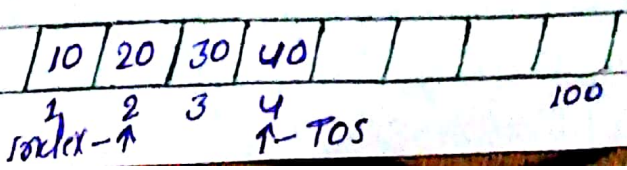
(i) Stack overflow?
    if top = Max stack, write overflow and exist.
(ii) Read item.
(iii) set Top = Top+1
(iv) set stack[TOP] = item
(v) Exist

POP operation

(i) Stack underflow?
    if Top = -1 then write underflow and exist.
(ii) Repeat steps 3 to 5 until top > 0
(iii) Set item = STack[top]
(iv) set top = TOP-1
(v) write deleted item
(vi) Exit

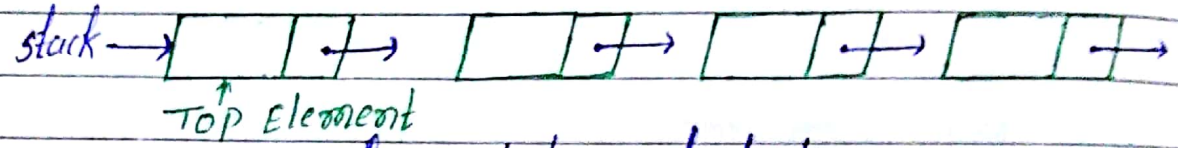
STACK Representation :

- Static implementation of stacks uses arrays to create stack
- The array implementation technique is very simple and easy to implement.
- Top of stack (TOS) is 0 (-1 for C/C++) for an empty stack
- We must store the index of an array containing the top element.
- The store the current position of the top element another field is used.



Date: / / Page no:

```
#define MAXSIZE 100
struct stack
{
    int top;
    int element[MAXSIZE];
};
struct stack s;
```



∴ Representation of stack ∴

- The pointer to the beginning of the list is considered to be the top of the stack
- The stack is initialized by setting its top value to NULL

CONVERSION of INFIX TO POSTFIX NOTATTION ∴

- The method of writing the operators of an expression either before their operands or after them the polish notation.

∴ There are three ways to write an expression

① Infix Notations ∴ when the operators exist between two operands then the expression is called infix notation

A + B

∴ the expression to add two numbers A and B is written in infix notation.

Date: / / Page no:

(II) **Prefix Notations** ÷ when the operators are written before their operands, then the expression is called the prefix Notation.

+AB

÷ The expression to add two no. A and B is written in prefix notations.

(III) **postfix Notations** ÷ when the operators are written after operands. it is called postfix or suffix notation. it is known as reverse polish notation.

AB+

÷ The expression to add two no. A and B written in postfix notation.

Algorithm (convert an infix to prefix form)

• suppose Q is an arithmetic expression infix expression to its ~~prefix~~ This algorithm finds the equivalent prefix expression P.

(I) Push ")" onto STACK and add "(" to the end of Q

(II) Scan Q from right to left and repeat steps (III) to (IV) of each element of Q.

(III) if an operand is encountered add it to P.

(IV) if a right parenthesis is encountered push it onto STACK

(V) if an operator (X) is encountered then.

(a)- Repeatedly pop from stack.

(VI) if left parenthesis is encountered

(a) Remove the Right parenthesis.

(VII) Exit

Date: / / Page no:

POSTFIX Expression using STACK :

- Once the expression is converted to postfix, we no longer require to remember the precedence rules.
- The expression may be evaluated by scanning from left to right.

Algorithm :

```
symb = next input character.
while not end of input
{
  if symb is an operand.
     push onto the stack
  else
  {
     pop two operands from the stack
     Result = op1 symbol op2
     push result onto the stack
  }
  symb = next input character.
}
return (pop stack)
```

Advantages of polish notation :

The advantages of polish notation are as follows—

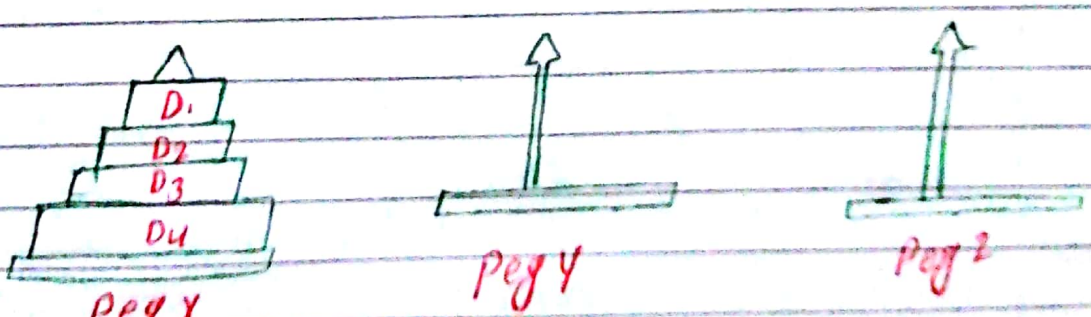
1. Polish notation allows us to write expression without need for parenthesis
   - a*(b+c)
   - be*a+bc
   - a*b+c is +*abc

Date: / / Page no:

(ii) polish notation is easier to evaluate for a computer.
(iii) Because polish notation requires no parenthesis so it saves memory and efforts.
(iv) We all are cozy working with prefix expressions
(v) Another advantage of polish notation over ordinary arithmetic notation

Towers of Hanoi problem :

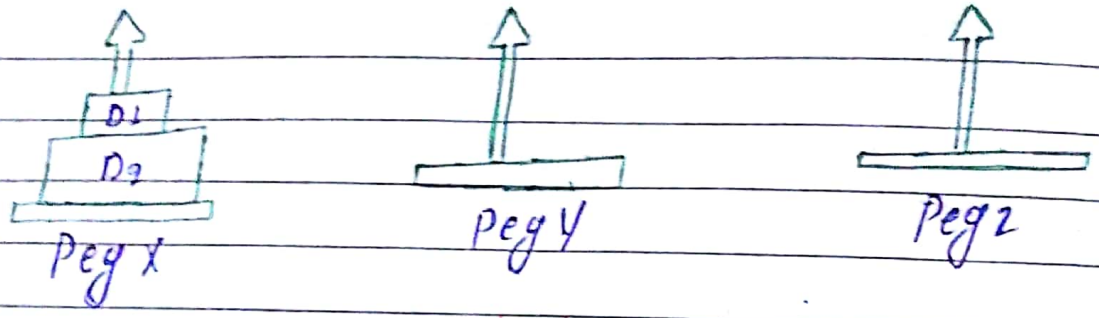
(i) The Towers of Hanoi is a famous recursive problem, which is based on three pegs and a set of disks of graduated sizes.
(ii) Following are the rules which must be followed before moving a disk
(iii) only one disk may be moved at a time
(iv) only the top disk on any peg may be moved to any other peg.
(v) A larger disk cannot be placed on a smaller one.



. The solution to this problem is very simple.
. if there are only two disks to move. First disk D_1 is moved to peg y and move the disk D_2 to peg z and finally the D_1 is moved from peg y to Peg z.

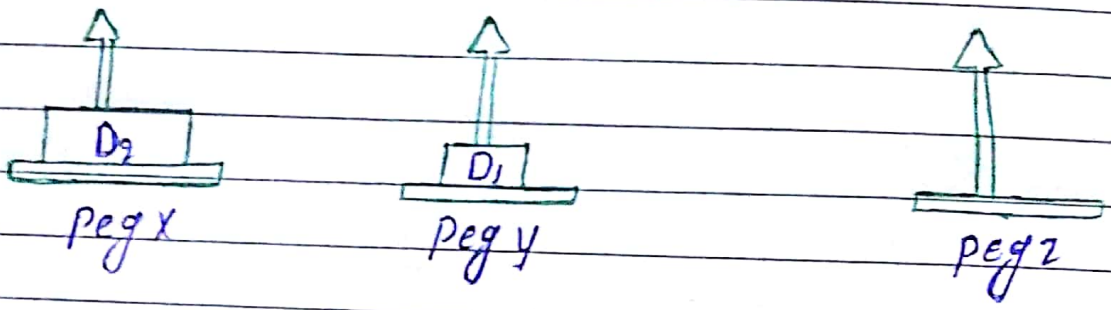
Date: / / Page no:

(i)



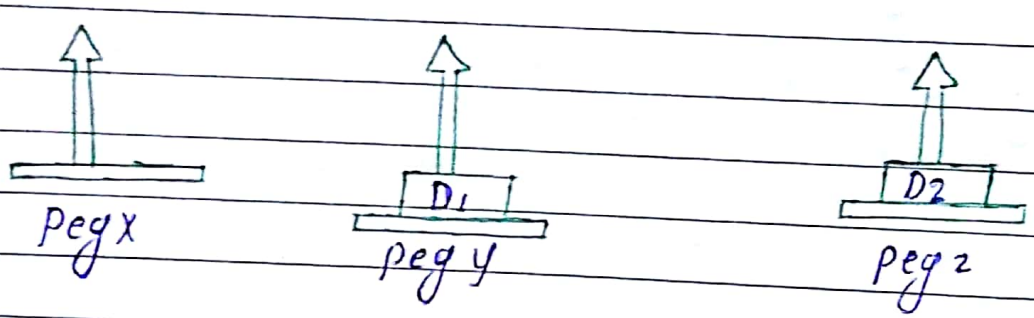
[First step]

(ii)



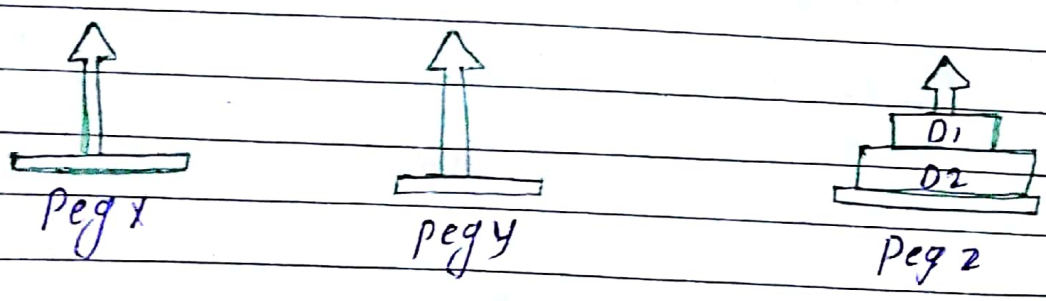
[Second step]

(iii)



[Third step]

(iv)



[Result]

Date: / / Page no:

Iteration Vs Recursion :

Iteration	Recursion
(i) It is a process of executing a statement or a set of statement until some specified condition is specified.	→ It is the technique of defining anything in terms of itself.
	→ There must be an exclusive if statement inside the recursive function specifying stopping condition.
(ii) Iteration involves four clear cut steps initialization condition execution and updation.	→ Not all problems have recursion solution.
(iii) Any recursive problem can be solved iteratively.	

Recursion Algorithm :

(i) The algorithm returns the value in the first parameter FIB

(i) if N=0 or N=1, then set FIB := N and Return.

(ii) call FIBONACCI [FIBA, N-2]

(iii) call FIBONACCI [FIBB, N-1]

(iv) set FIB := FIBA + FIBB

(v) Return.

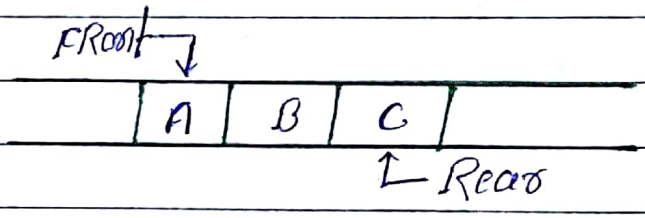
Date: / / Page no:

QUEUES :

- A queue is a non-primitive linear data structure.
- Linear data structure where insertion and deletion are performed at separate ends.

Insertion
⇓
Rear :- it is an ordered group of elements in which elements are added at one end called the rear end.

Deletion.
⇓
FRONT :- it is an ordered group of elements in which the existing elements are removed from the other end called the front end.



- Queue is a first-in-first-out we called FIFO.

✳ Operations

① Insertion of Element :

(i) Initialize F=0 R=-1
(ii) check overflow condition
if F=0 and R=MaxSize or F=R+1
(iii) if F=NULL
set F=0 and R=0
else if R=MaxSize
set R=0

(iv) - set R = R+1 → increment Rear counter
(v) → queue [R] = item
(vi) → Exit.

(ii) **Deletion of Element :**

(i) → check underflow condition
if F<0. write underflow. exit

(ii) → set item = queue [F] → suppose F=1 item q[1].

(iii) if F = R
then set F=R = NULL
else if F = MAXSIZE
then set F=0

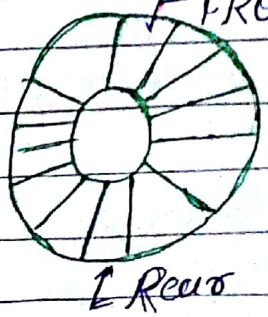
(iv) set F = F+1

(v) Exit

(*) **Circular queue**

- in this the rear pointer can point to the beginning of the queue when it reaches end of the queue.

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



: implement clockwise

Date: / / Page no:

Advantage : empty space can be filled again using rear pointer

Initialization : set code

```
front = 0;
Rear = 0;
```

Insertion :

```
Q[rear] = item;
rear = (rear+1) mod n;
```

Deletion :

```
if front] = item;
front = (front+1) mod n;
```

Disadvantage :

There is also problems associated with circular queue

DEQUE (Double Ended queue) :

- In this, insertion and deletion can be performed at both.
  i.e → front pointer can be used for insertion, and rear pointer can be used for deletion.
- There exits two variations of deque.

Date: / / Page no:

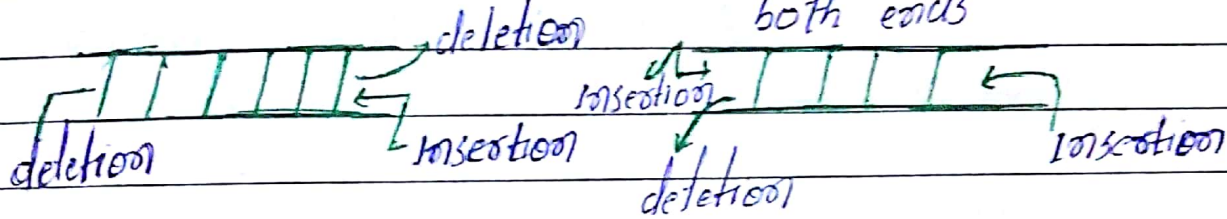
Variations

① Input Restricted queue

- deletion can be performed at both ends.
- Insertion at one end (rear)

② output Restricted queue

- deletion at one end (Front)
- Insertion can be performed at both ends



Priority Queue :

- Each element is inserted or deleted on the basics of their priority.
- Higher priority > Lower priority
- Same priority [FCFS basis] → First come - first serve

Types of priority queue :

① Ascending order priority queue : Lower priority no. high priority.

② Descending order priority queue : higher priority no. to higher priority

Date: / / Page no:

STACK Vs QUEUE :-

S!	
A stack is logically a LIFO type of list	A queue is logically a FIFO type of list.
In a stack insertions and deletion are possible only at one end.	In a queue insertion is done at one end and deletion is performed at other end.
only one item can be added at time	only one item can be added at a time.
only one item can be deleted at a time.	only one item be deleted at a time.
No element other than the top of stack element is visible	No element other than front and rear element are visible