29-07-2025

Tuesday

UNIT-02

HEAPS AND GRAPHS

* Priority Queue:

- In a priority queue we always delete the element with highest priority.
- Priority queue are implemented by heap trees
- Heap trees are mainly two types:

(1) Min Heap (2) Max Heap

* Min Heap:

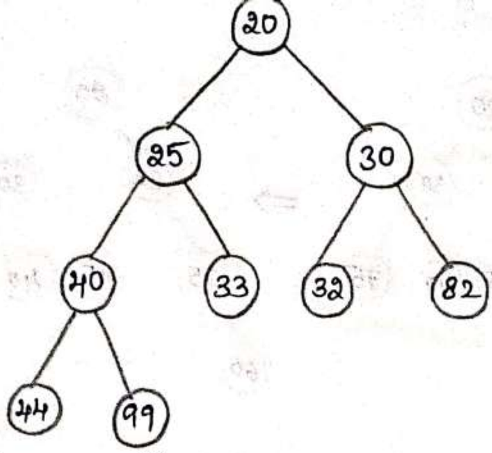
A min heap tree is a binary tree which satisfies the following two properties:

1. Structure property:

All levels are completely filled except the last level In last level the node will inserted from left to right

2. Order property: (Min Heap property)

For every node the value in the node is less than or equal to the value of its children.



* Operations on Heap:-

1) Insertion or construction

2) Deletion

* Technical terms or function:-

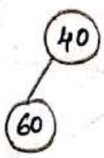
1. Build heap
2. Re-heapify
3. Heap up
4. Heap down
5. Delete

* Construct min Heap tree for following data?

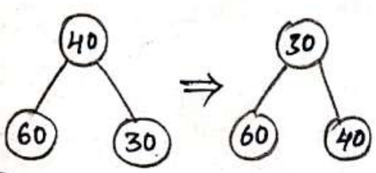
40, 60, 30, 35, 45, 20, 75, 22, 38, 15, 24

1. Insert 40

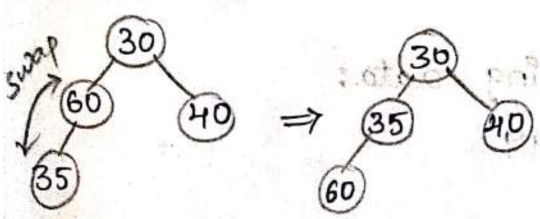
2. Insert 60



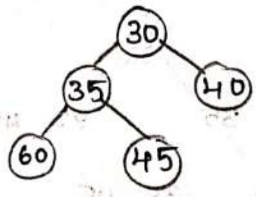
3. Insert 30



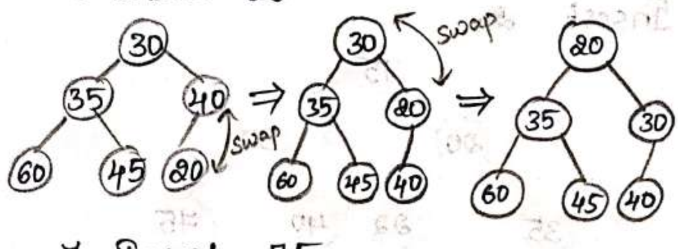
4. Insert 35



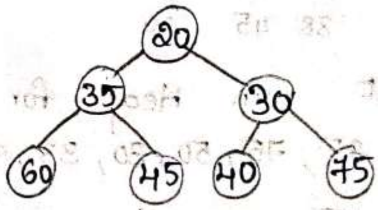
5. Insert 45



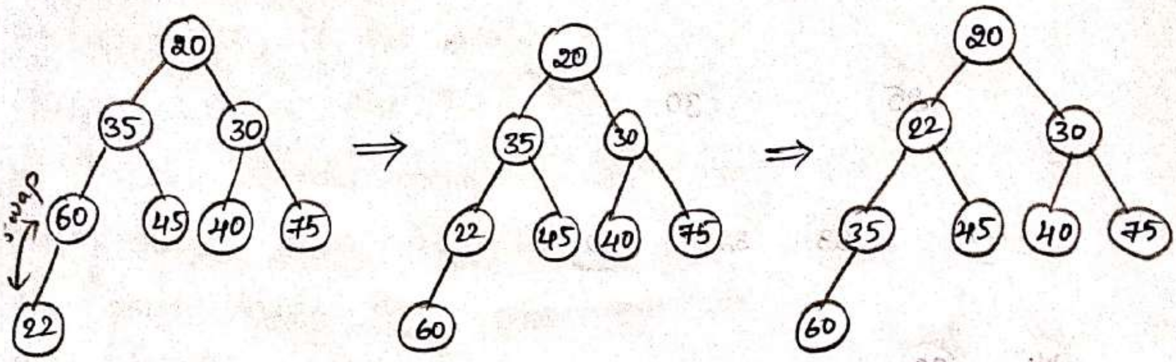
6. Insert 20



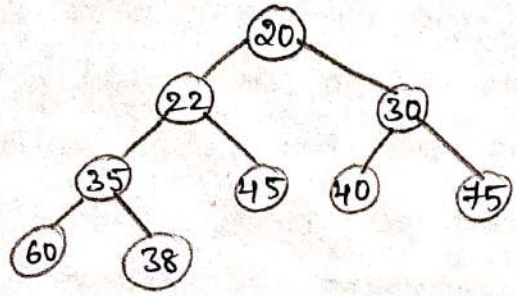
7. Insert 75



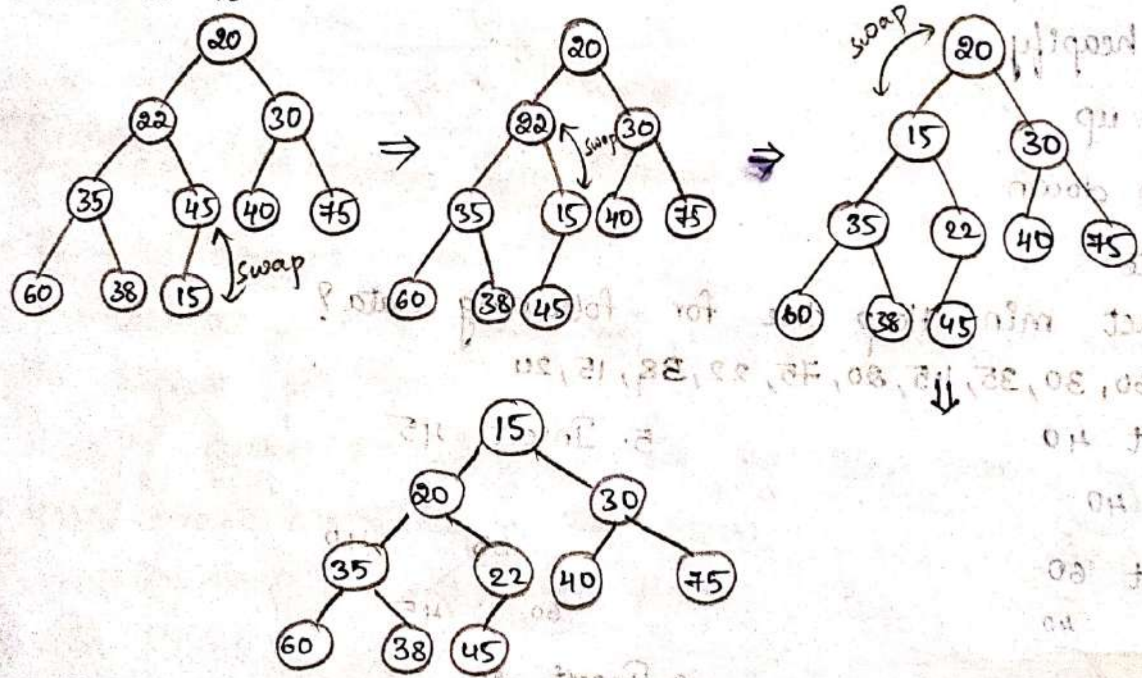
8. Insert 22



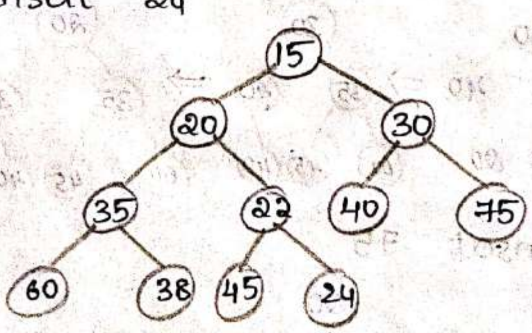
9. Insert 38



10. Insert 15



11. Insert 24

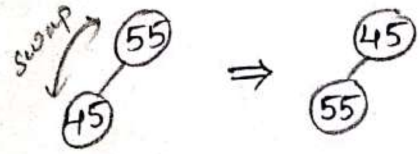


* Construct min Heap for following data:

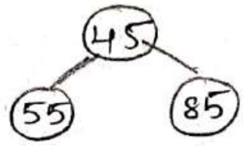
55, 45, 85, 75, 50, 30, 33, 99, 66, 49

1. Insert 55

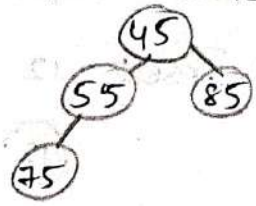
2. Insert 45



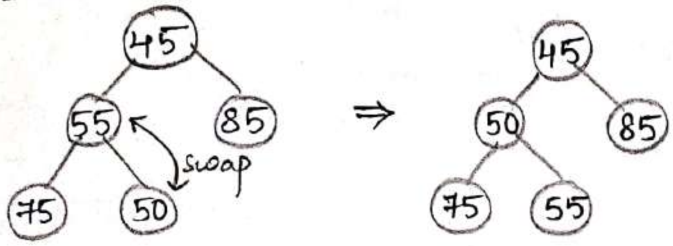
3. Insert 85



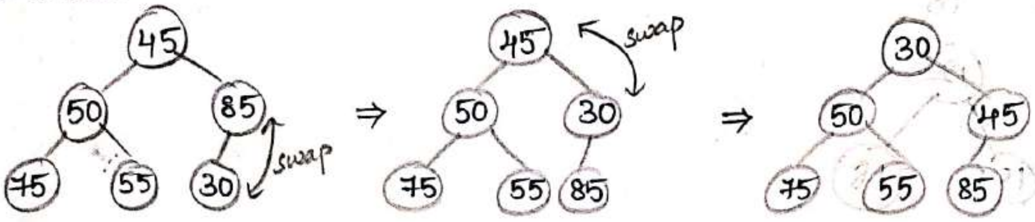
4. Insert 75



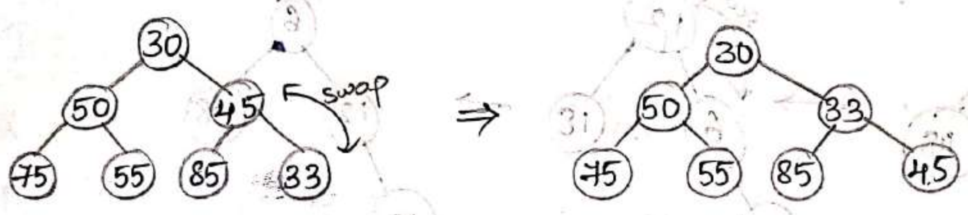
5. Insert 50



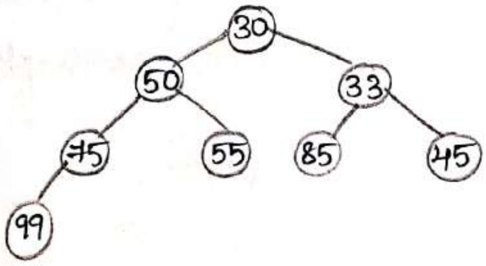
6. Insert 30



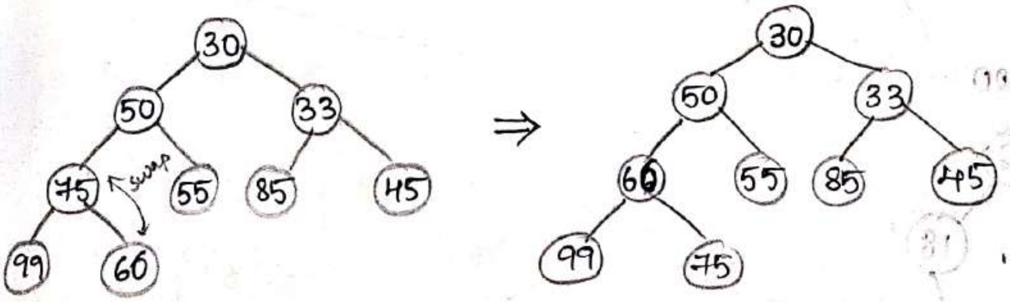
7. Insert 33



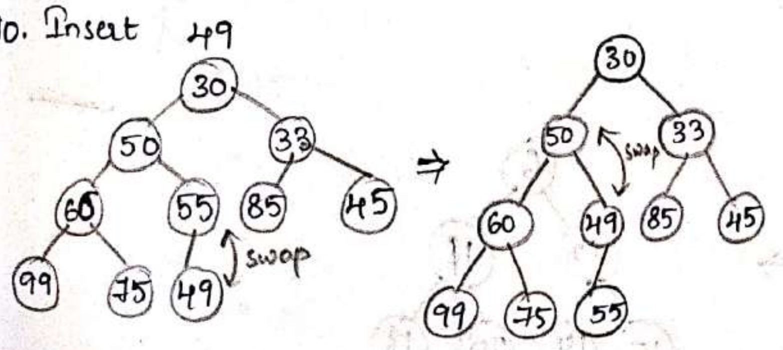
8. Insert 99



9. Insert 66



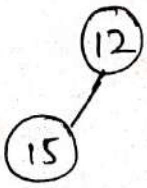
10. Insert 49



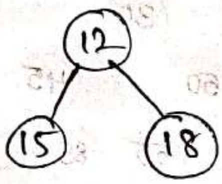
* Data: 12, 15, 18, 6, 14, 20, 11, 22, 16

1) Insert 12

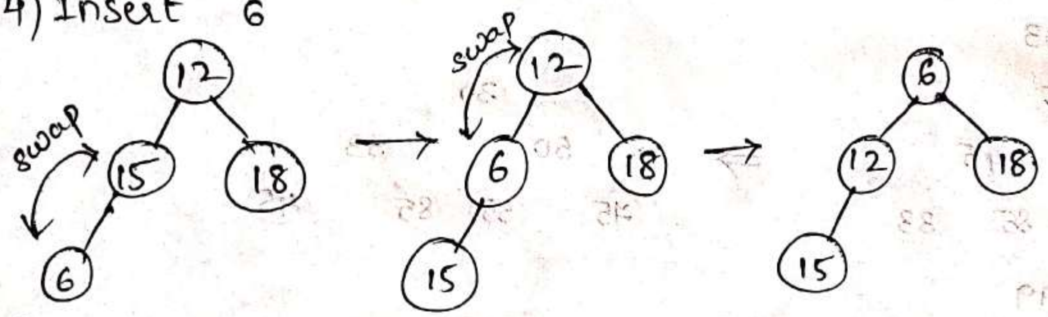
2) Insert 15



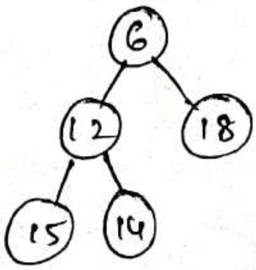
3) Insert 18



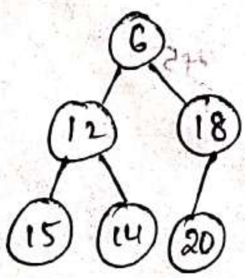
4) Insert 6



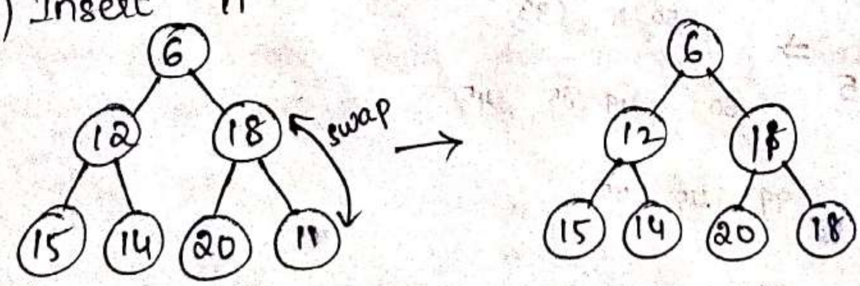
5) Insert 14



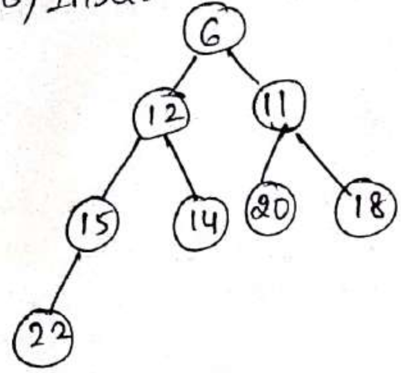
6) Insert 20



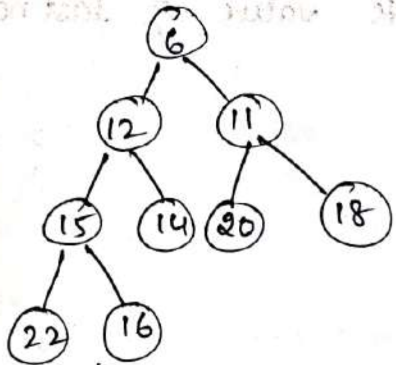
7) Insert 11



8) Insert 22



9) Insert 16

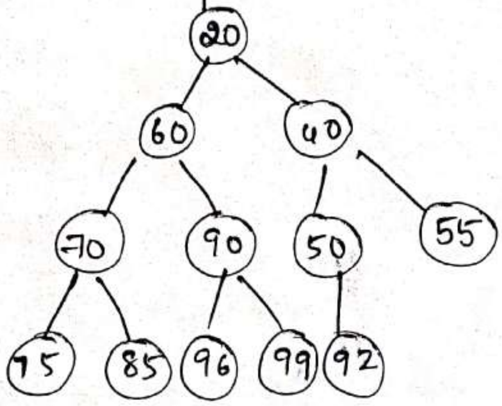


* **Insert / Heap up:-**

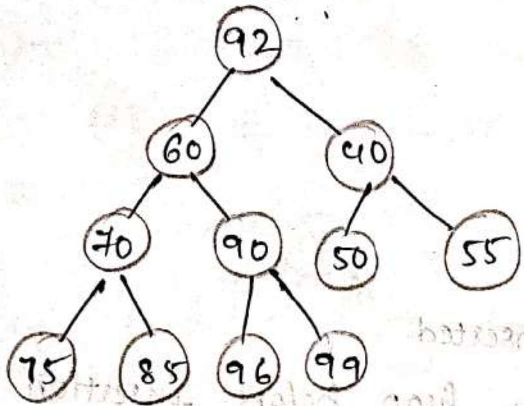
//x: element to be inserted
//n: no. of elements in heap before insertion
//a: array of elements

```
Algorithm Insert (a, n, x)
{
    n := n+1;
    pos := n;
    a[pos] := x;
    while (pos > 1)
    {
        par := pos/2;
        if (a[pos] < a[par])
        {
            swap (a[par], a[pos]);
            pos := par;
        }
        else
            break;
    }
}
```

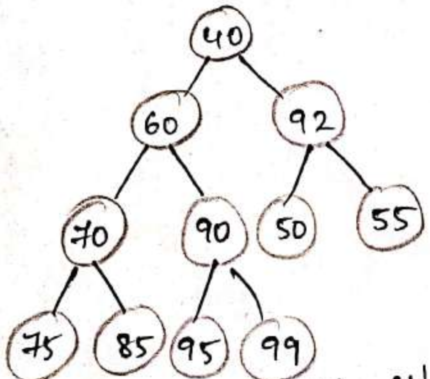
*Explain deletion process in min heaps?



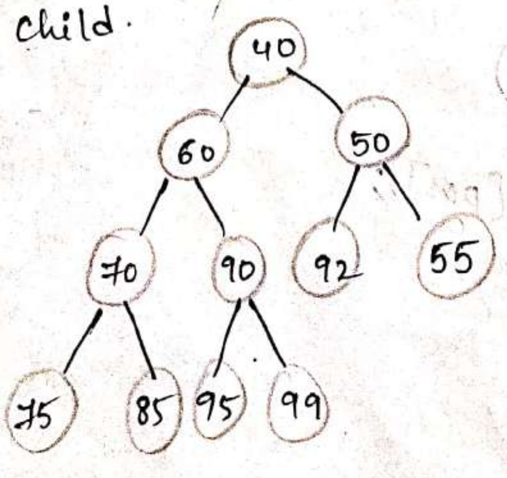
Delete the root element & write value of last node in last level in root node.



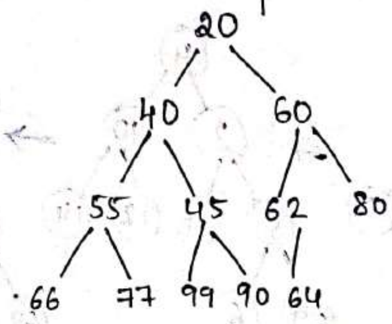
Compare root value with value two of it's children & swap with smallest child.

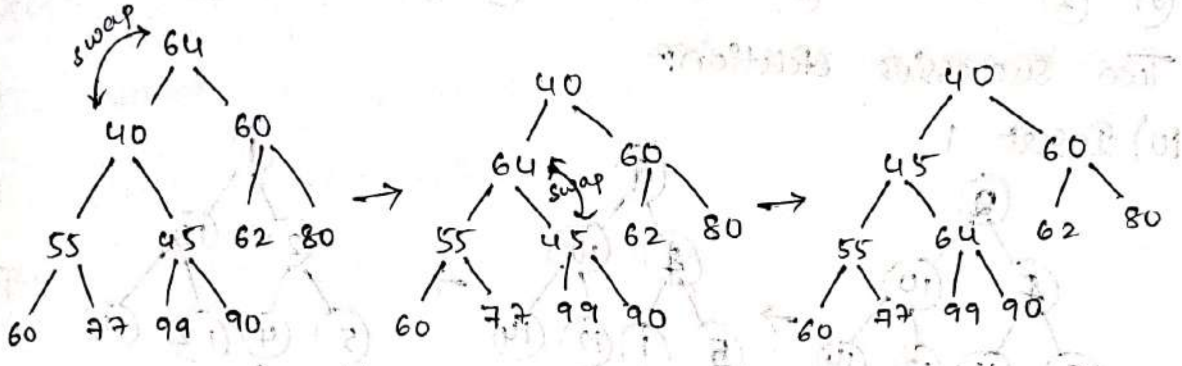


Again compare 92 with it's children & swap with smallest child.



Data: Explain Deletion process





<u>Note:</u> Delete = ReHeapify ⇒ Heapify(1) = HeapDown(1)

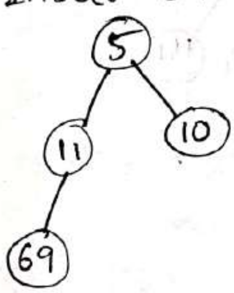
Eg: 11, 10, 5, 69, 6, 12, 14, 4, 2

1) Insert 11

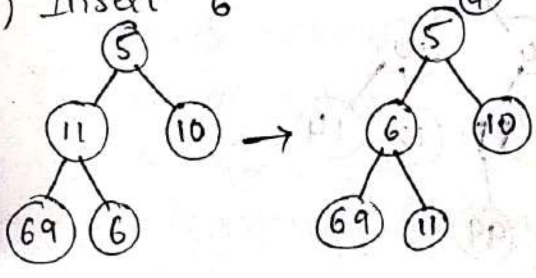
2) Insert 10

3) Insert 5

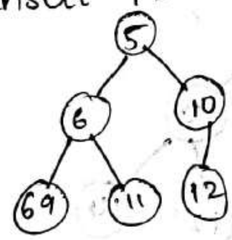
4) Insert 69



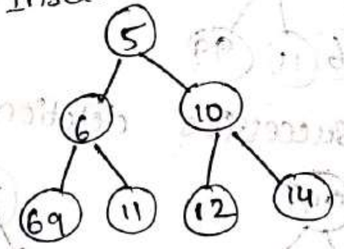
5) Insert 6



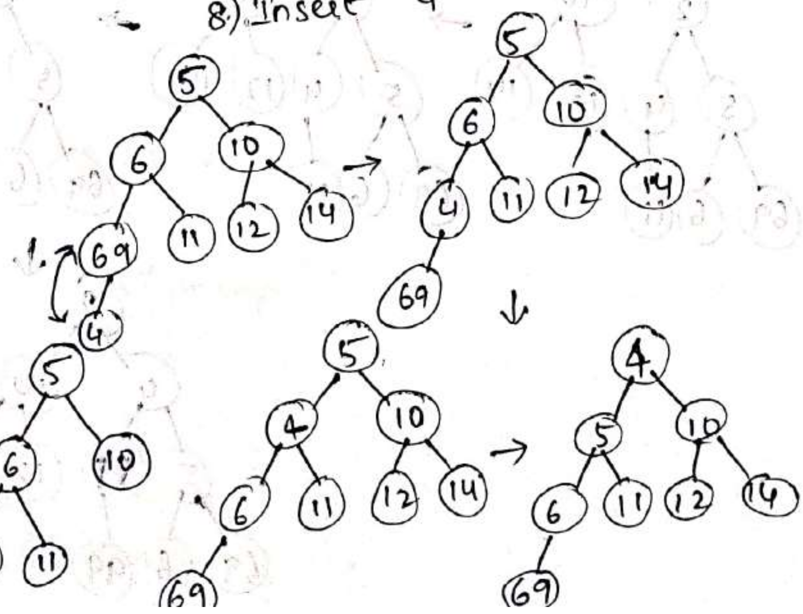
6) Insert 12



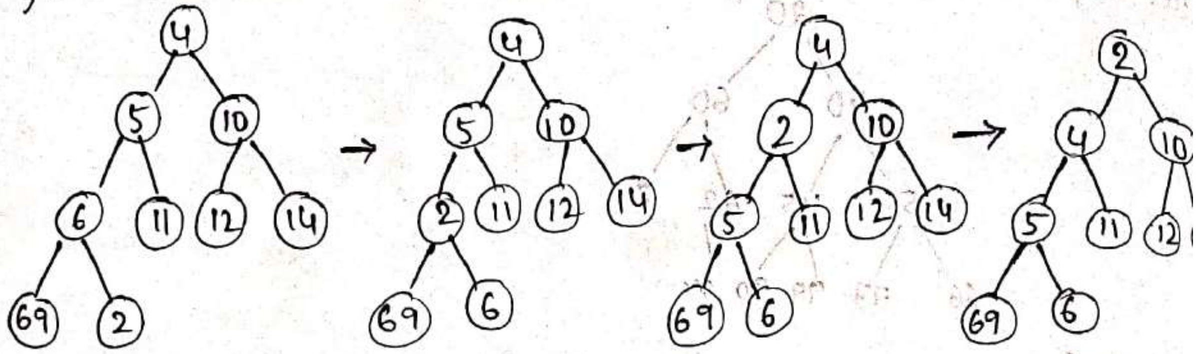
7) Insert 14



8) Insert 4

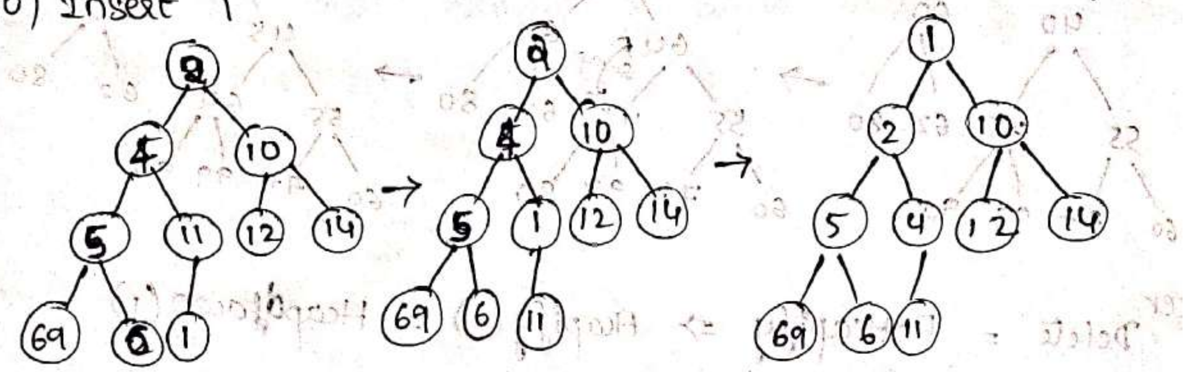


9) Insert 2

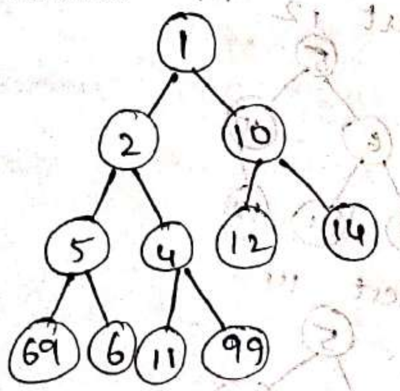


~~Two successive deletion~~

10) Insert 1

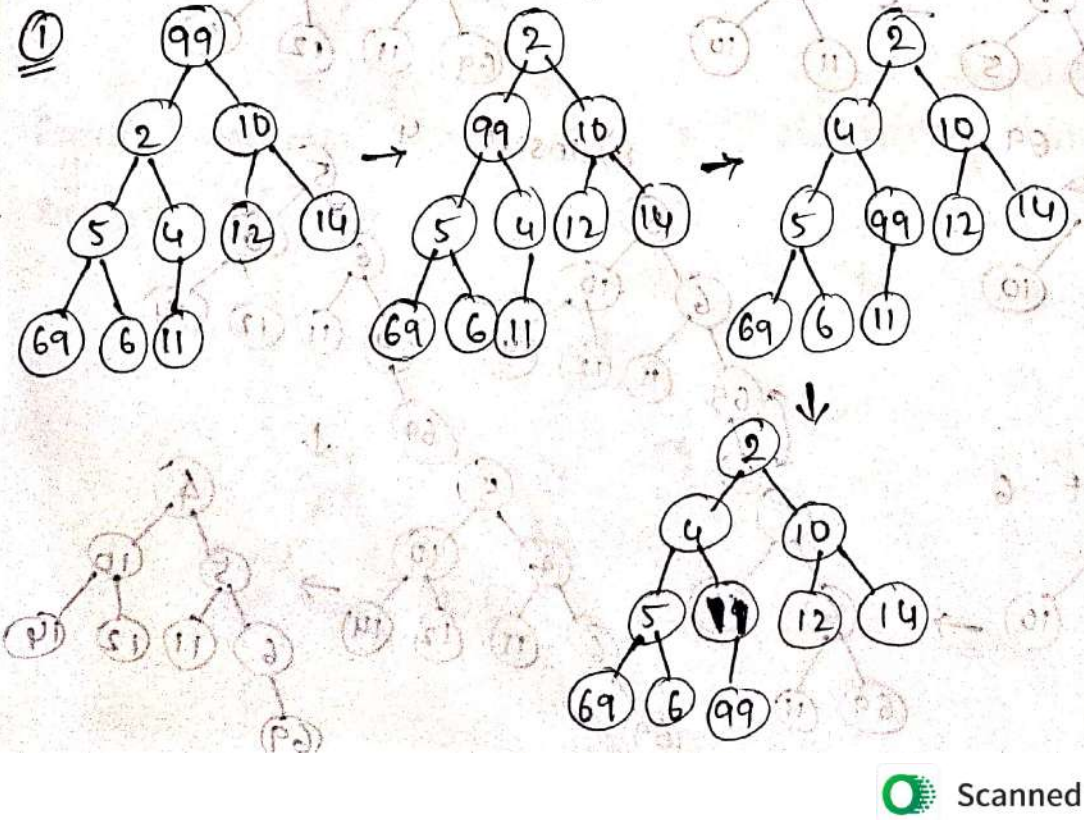


11) Insert 99

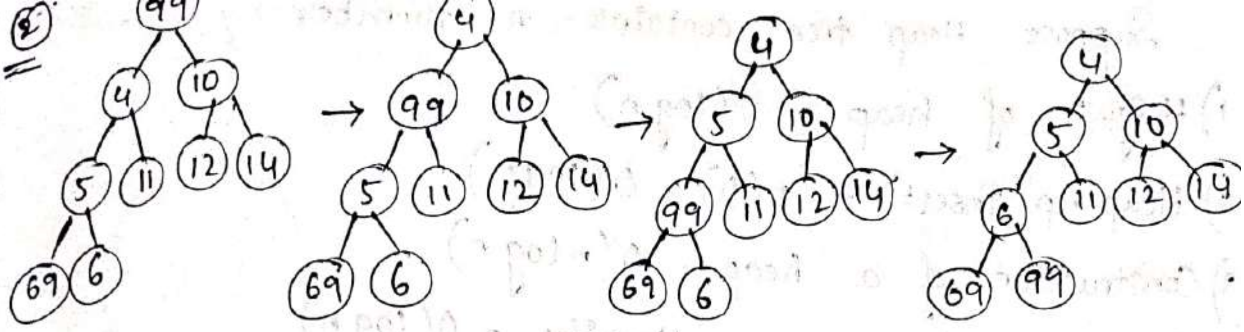


Two successive deletion process:-

①



②



* Deletion in min Heap : Pseudo code

```
// n: Number of elements in heap before deletion
// a : Array of elements
Algorithm Delete (a, n)
{
    x = a[1];
    a[1] = a[n];
    n = n-1;
    Min-Heapify(a,1,n);
    return(x);
}
```

* Heapify in min-Heap:-

```
// n : no.of elements in heap
// a : Array of elements
// i : The position , where we start heapify.
Algorithm MinHeapify (a, i, n) :
{
    l = 2*i ; r = 2*i+1 ;
    smallest = i;
    if (l ≤ n and a[l] < a[smallest])
            smallest = l;
    if (r ≤ n and a[r] < a[smallest])
            smallest = r;
    if (smallest ≠ i){
        swap(a[i], a[smallest]);
        MinHeapify (a, smallest, n);
} }
```

Suppose Heap tree contains n number

1) Height of heap $= O(\log n)$
2) Heap up (Insert) $= O(h) = O(\log_2 n)$
3) Construction of a heap $= O(n \log n)$
4) Delete or Heapdown or Heapify $= O(\log n)$
5) Build Heap $= O(n)$.

Note:-

Heap & Binary Heap are same

*Applications of Binary Heap:-

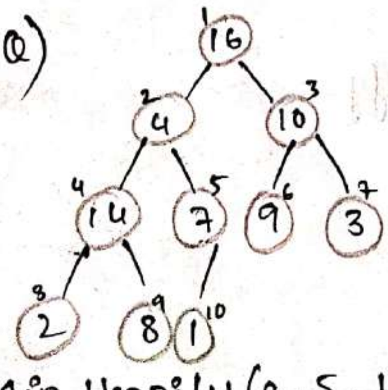
1) Implementation of heap sort
2) Implementation of priority queue.

⇒ Algorithm Build Min Heap(a)

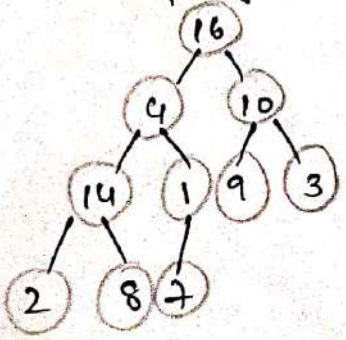
```
{
    n = length(a);
    for i = [n/2] down to 1 do
            minHeapify(a, i, n)
}
```

T.C $\geq O(n)$

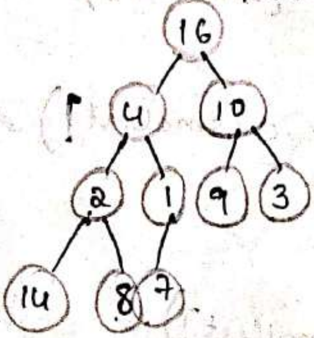
Q)



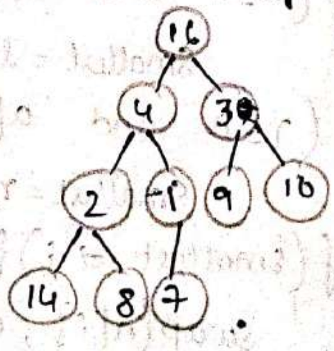
Min Heapify(a, 5, 10)



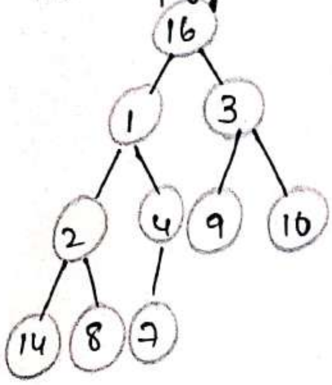
MinHeapify(a,4,10)



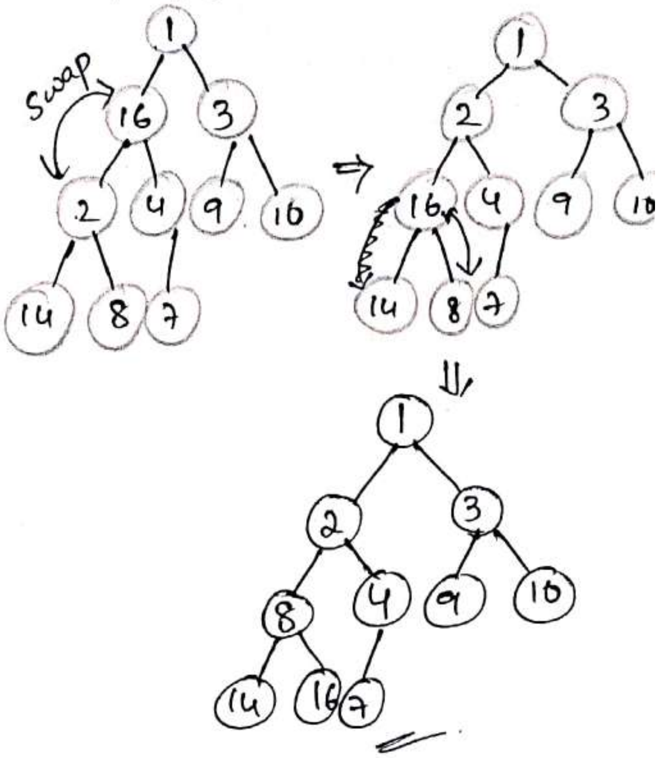
MinHeapify(a,3,10)



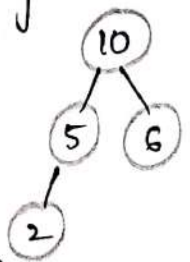
MinHeapify (a, 2, 10)



MinHeapify (a, 1, 10)

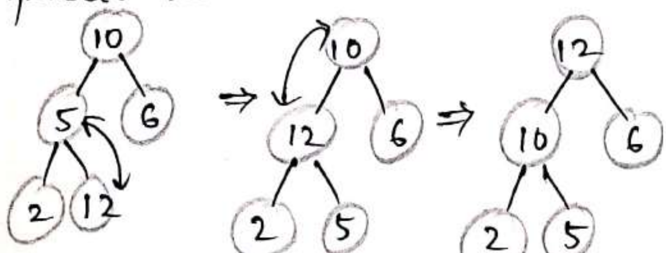


Q) Arjun — max heap

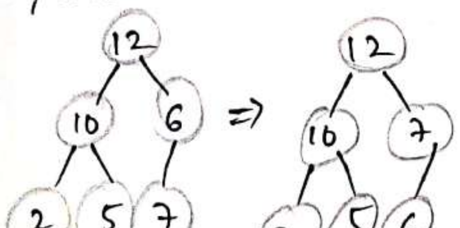


let us apply construction one by one

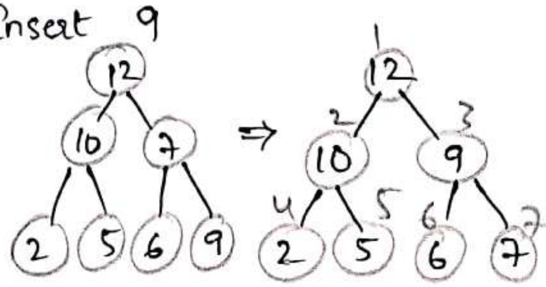
1) Insert 12



2) Insert 7

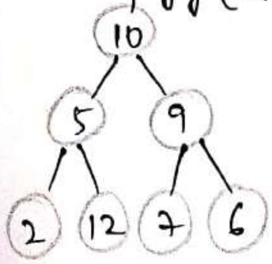


3) Insert 9

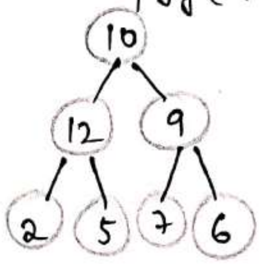


let us apply build heap

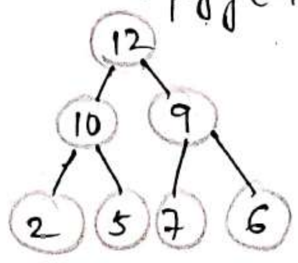
MaxHeapify (a, 3, 7)



MaxHeapify (a, 2, 7)



MaxHeapify (a, 1, 7)



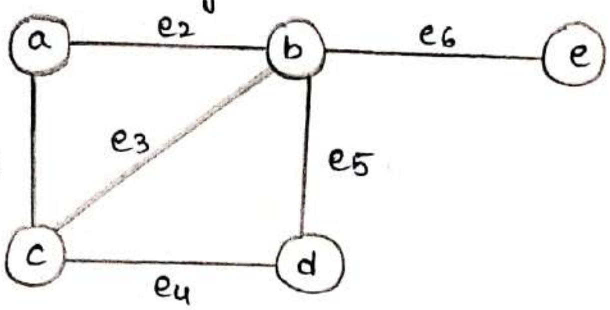
05-08-2025
Tuesday

* GRAPHS *

* Graph:

- A graph is a pair of (V, E) of sets.
- The elements of V are called vertices and the elements of E are called edges.

Eg:-



$V = \{a, b, c, d, e\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$E = \{\{a,c\}, \{a,b\}, \{b,c\}, \{c,d\}, \{b,d\}, \{b,e\}\}$

* Types of graphs:-

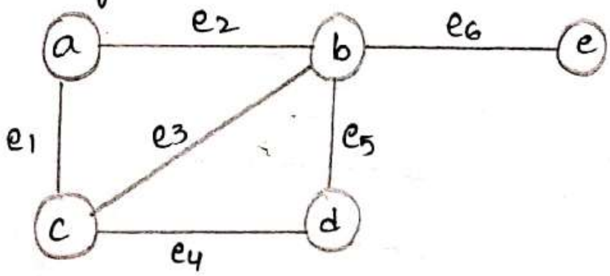
There are mainly two types of graphs:

1. Undirected Graph
2. Directed / Di Graph

* Undirected graph:-

- In an undirected graph, edges do not have any direction associated with them.
- Each edge e_K associated with unordered pair $\{u, v\}$ of vertices

Eg:-



Edge e_5 joins b & d.

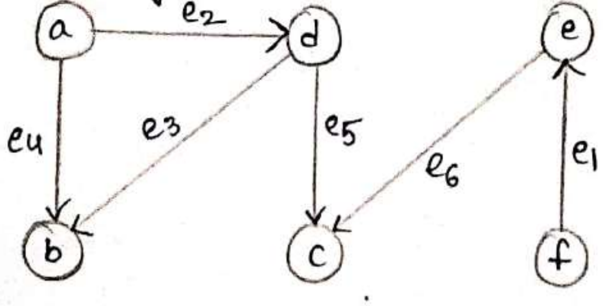
Edge e_5 joins d & b.

b and d are end points of e_5.

* Directed graph:-

- In a directed graph, edges have direction.
- Each edge e_K associated with ordered pair (u, v) of vertices

Eg:-



$V = \{a, b, c, d, e, f\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$E = \{(f,e), (a,d), (d,b), (a,b), (d,c), (e,c)\}$

Edge e_5 is from d to c.

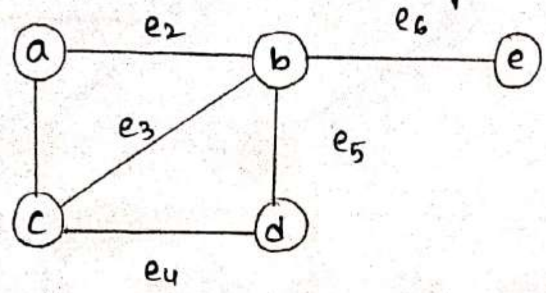
d is source / initial node of edge e_5

c is destination / terminal node of edge

* Adjacent vertices:-

- In an undirected graph, two vertices u and v are said to adjacent if there is an edge between those two vertices u and v.

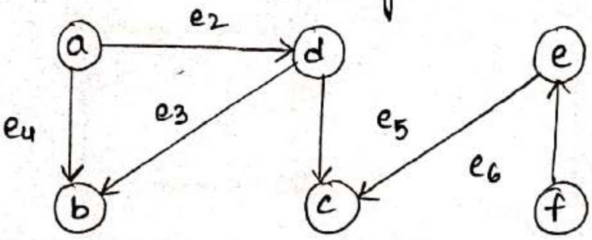
Egr



a, c are adjacent
b, d are adjacent
a, d are not adjacent
d, e are not adjacent

- In an directed graph, vertex v is said to be adjacent to vertex u if there is an edge from u to v.

Egr

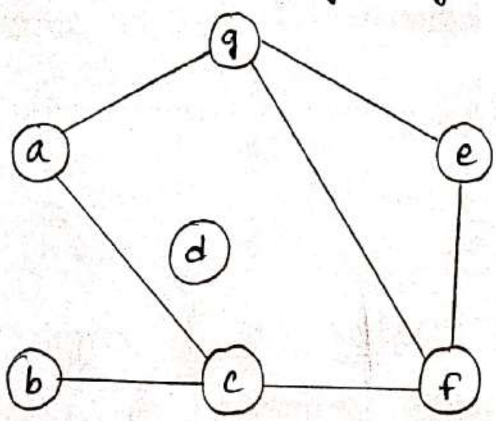


d is adjacent to a.
c is adjacent to e.
a is not adjacent to b.
d is not adjacent to b.

* Degree of a vertex:-

- Number of edges drawn to a vertex is called degree of vertex v and is denoted by deg(v).

Egr

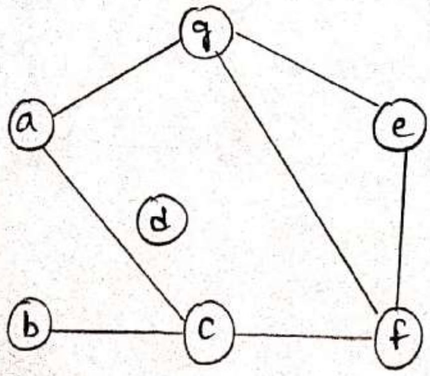


deg(a) = 2
deg(b) = 1
deg(c) = 3
deg(d) = 0
deg(e) = 2
deg(f) = 3
deg(g) = 3

* Isolated vertex:-

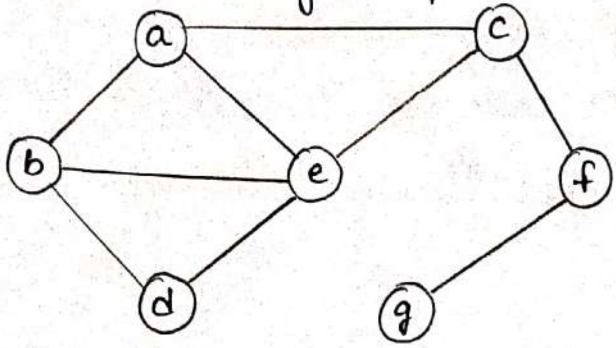
- A vertex of degree 0 is called isolated vertex.

Egr



d is isolated vertex

* Find the degree of vertices in the following graph:-



$deg(a) = 3$

$deg(b) = 3$

$deg(c) = 3$

$deg(d) = 2$

$deg(e) = 4$

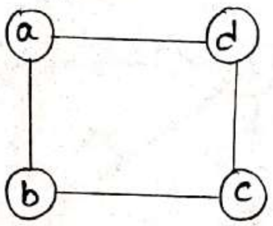
$deg(f) = 2$

$deg(g) = 1$

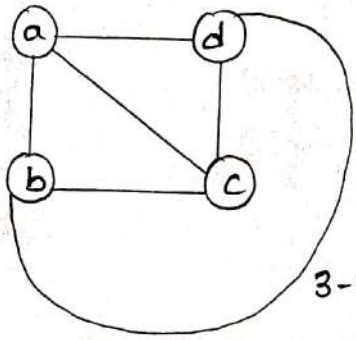
* Regular graph:-

- Graph in which all the vertices are of same degree is called regular graph.

Egr



2-regular graph

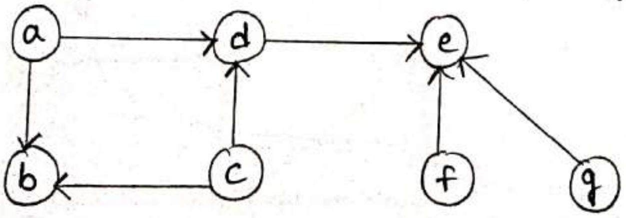


3-regular graph

* In degree of a vertex:-

- Number of incoming edges drawn to a vertex is v called indegre of vertex v and is denoted by $deg^+(v)$.

Egr



$deg^+(a) = 0$

$deg^+(b) = 2$

$deg^+(c) = 0$

$deg^+(d) = 2$

$deg^+(e) = 3$

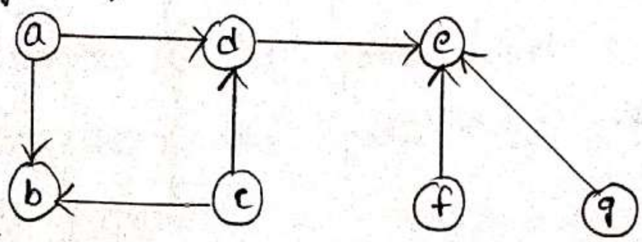
$deg^+(f) = 0$

$deg^+(g) = 0$

* Out degree of a vertex:-

- Number of outgoing edges drawn to a vertex v is called ~~outd~~. outdegree of vertex v and is denoted by $deg^-(v)$.

Egr



$deg^-(a) = 2$

$deg^-(b) = 0$

$deg^-(c) = 2$

$deg^-(d) = 1$

$deg^-(e) = 0$

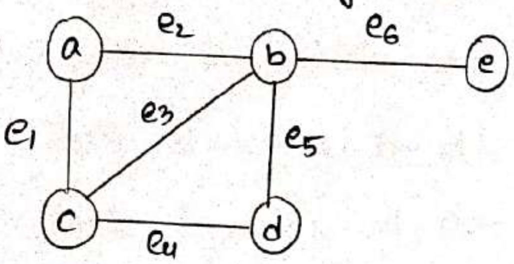
$deg^-(f) = 1$

$deg^-(g) = 1$

* Incident vertices:-

- In an undirected graph, if e is an edge between two vertices u and v, then we say that the edge e is incident with u & v.

Egr



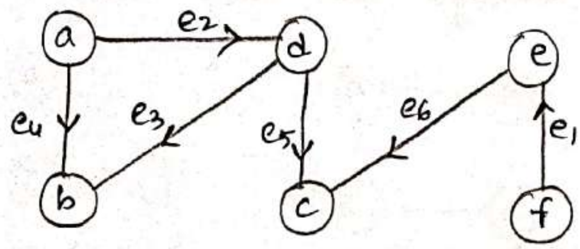
Edge e_1 is incident with a & c

Edge e_5 is incident with b & d.

06-08-2025
Wednesday

- In a directed graph, if e is an edge from vertex u to v, then we say that the edge e is incident form u and incident to v.

Egr



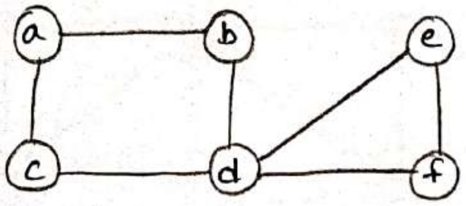
Edge e_1 is incident from f.

Edge e_1 is incident to e

* Path:-

A path P from a vertex u to vertex v is a sequence $P = \{v_0, v_1, v_2, v_3, \dots v_n\}$ of n+1 vertices such that $u = v_0$, $v = v_n$ and v_i is adjacent to v_{i-1} for all $i = 1, 2, \dots, n$.

Egr



b-d-e-f is a path from b to f

a-d-c-d-f is a path from a to f

a-c-d-f-e is a path from a to e

* Length of path:-

- The number of edges in a path P is called length of path P.

Egr Length of the path b-d-e-f is 3

Length of the path a-d-c-d-f is 4

Length of the path a-c-d-f-e is 4

* Simple path:-

- A path with no repeated vertices except at ends.

Egr Path b-d-e-f is a simple path

Path d-c-a-b-d is a simple path

Path a-d-e-f-d-c is not a simple path

Path a-d-a-c-d-f is not a simple path.

* Closed path:-

- A path in which end vertices are same

Eg:- Path d-c-a-b-d is closed path

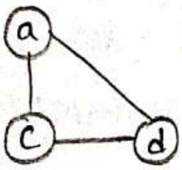
Path b-a-c-d-a-b is closed path

Path a-d-a-c-d-f is not closed path

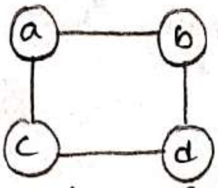
* Cycle:-

- A closed path no repeated vertices except at ends.

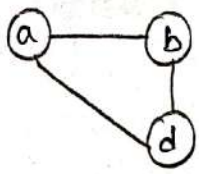
Egr



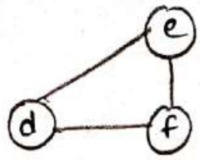
d-c-a-d



a-b-d-c-a



a-b-d-a

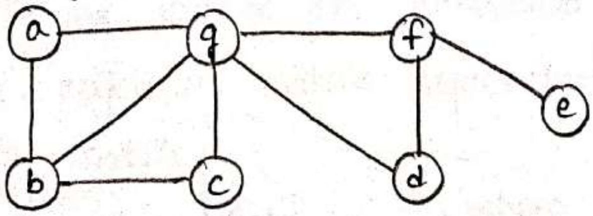


e-f-d-e

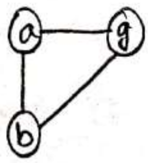
* Cyclic graph:-

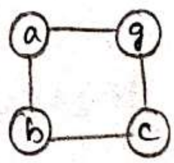
- A graph is called cyclic graph if there exists a cycle in the graph.

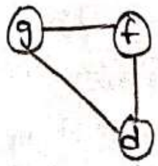
Egr

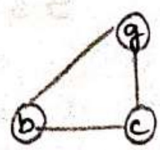


Cycles:





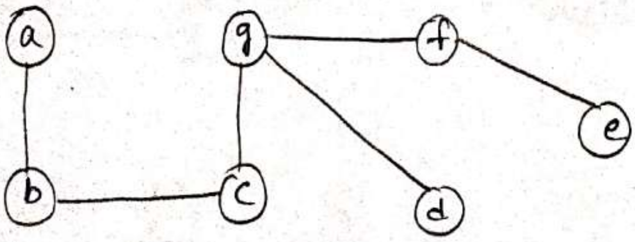




* ACyclic Graph:-

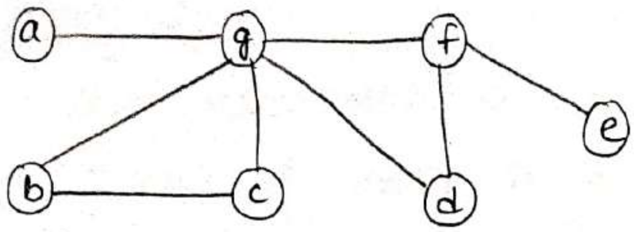
- A graph without any cycles is called Acyclic graph.

Egr



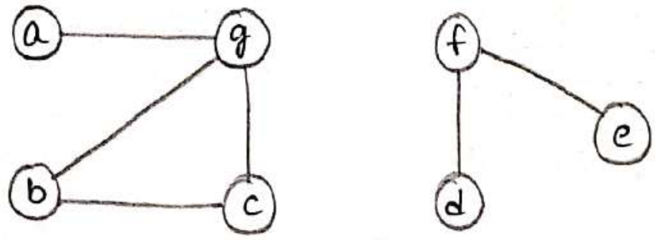
* Connected Graph:-

- An undirected graph is ~~called~~ said to be connected if there is a path between every pair of vertices.

Egr 

* Disconnected graph:-

- An undirected graph which is not connected.

Egr 

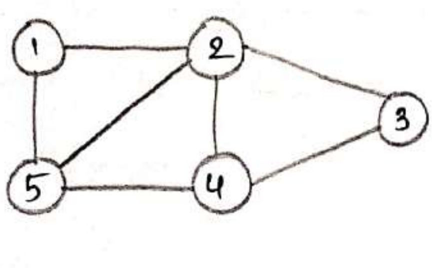
* How to represent a graph?

- There are 3 standard ways to represent a graph $G = (V, E)$
  1. Adjacency matrix
  2. Adjacency list
  3. Incidence matrix

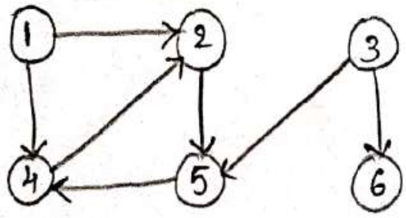
* Adjacency matrix:-

- Let $G = (V, E)$ be a graph with $|V| = n$.
- Let the vertices are numbered $1, 2, \ldots, n$
- The adjacency matrix represented of a graph G consists of a $n \times n$ matrix.

$$A = [a_{ij}]_{n \times n} \quad , \text{where} \quad a_{ij} = \begin{cases} 1 & \text{if } [i, j] \in E \\ 0 & \text{otherwise} \end{cases}$$

Egr 

$$\text{Adjacency matrix } A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

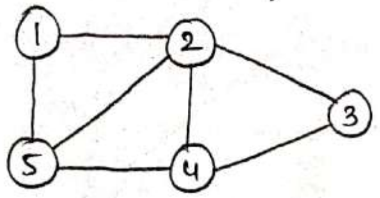


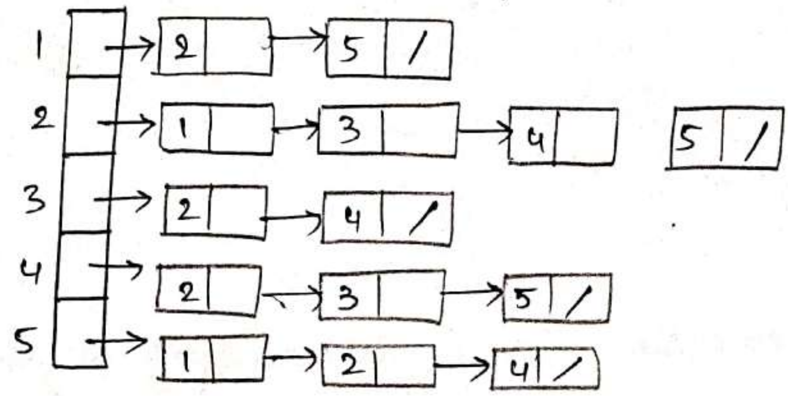
$$\text{Adjacency matrix } A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

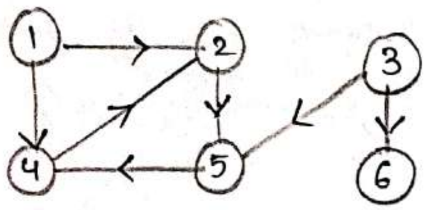
* Adjacency List:-

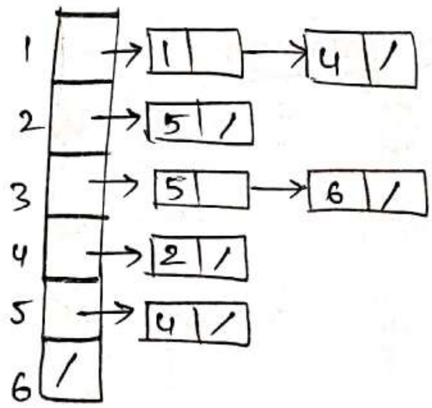
- In this representation we create an array of size n where each element of array is a linked list of adjacency vertices

Eg:-







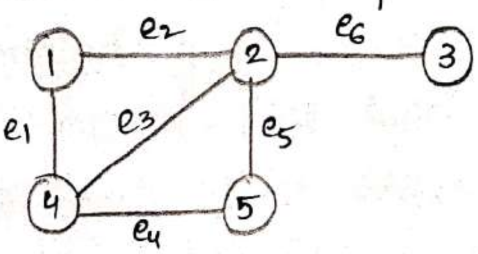


* ~~Indi~~

* Incidence matrix:

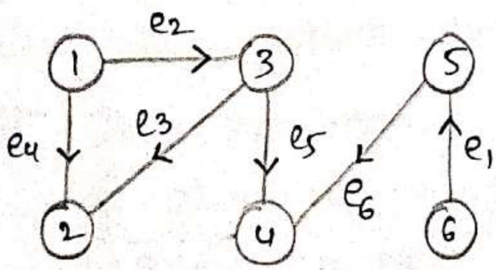
- An incidence matrix is a matrix where rows correspond to vertices and columns correspond to edges.
- Each element in the matrix indicates whether a particular vertex is incident to a particular edge.

Eg:-



Incidence matrix

$$I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

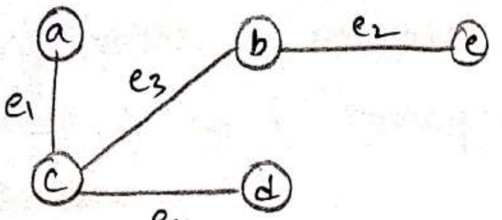


$$I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & +1 & +1 & 0 & 0 \\ 0 & +1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & +1 & +1 \\ +1 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \end{array}$$

* Tree:-

- A connected graph without any cycles is called tree
- A tree with n vertices will have n-1 edges.

Eg:-

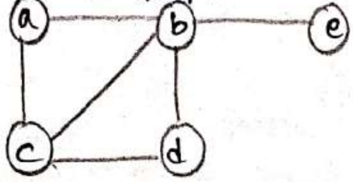


* Spanning tree:-

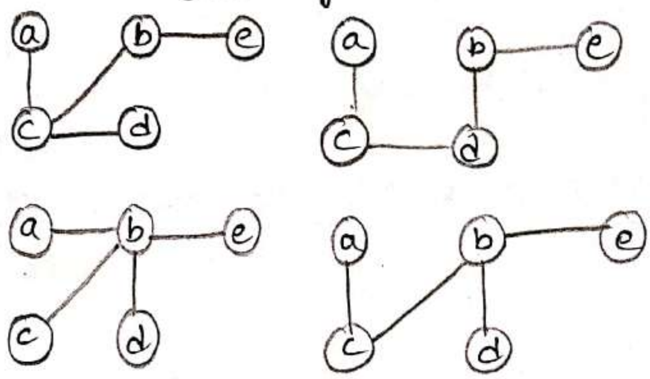
- A tree (i.e, connected, acyclic graph) which contains all the vertices of the graph.

Eg:-

Graph



Spanning tree



* Graph traversal algorithms:-

- Traversing a graph:

  Systematically follow the edges of a graph to visit the vertices of graph.
- Standard graph - traversal algorithms:
  1. Breadth - First Search (BFS)
  2. Depth - first Search (DFS)
- The difference between the two algorithms is in the order in which they explore the unvisited edges of the graph.

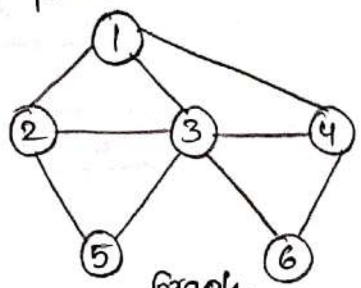
Note:- There are mainly two algorithms to find a spanning tree of given ~~tree~~ graph.

1. BFS 2. DFS

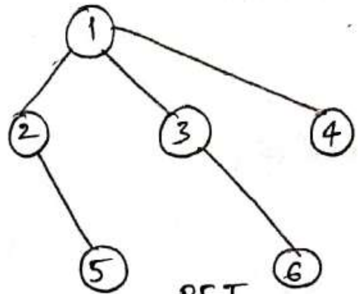
* Breadth First Tree (BFT):

- BFS produces a Breadth-First tree with root s that contains all reachable vertices.
- For any vertex v reachable from s, the simple path in the BFT from s to v corresponds to a shortest path from s to v in G.

Shortest path: Path containing the smallest number of edges.



Graph



BFT

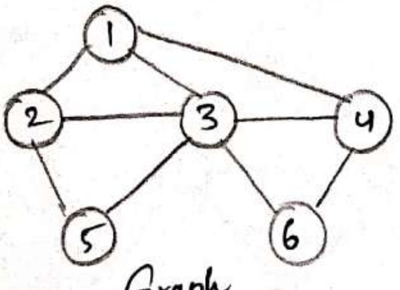
* Breadth First Search (BFS):-

- BFS algorithm discovers all vertices at distance K from S before discovering any vertices at distance K+1.
- Main idea:-

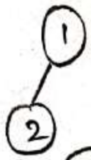
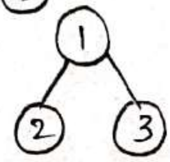
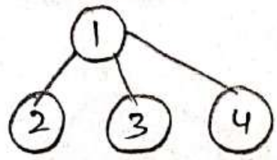
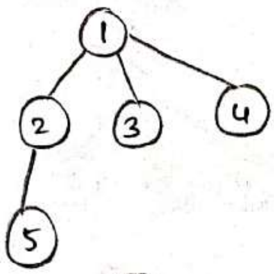
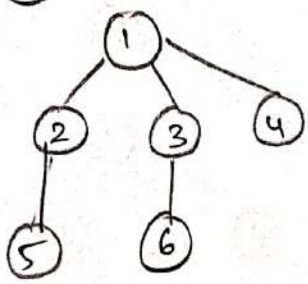
  Mark 's' as visited. Add s to the queue.
  Repeat the following till the queue is empty.
    Remove a vertex (say ω) from the queue.
    For each neighbour x of ω that is not yest visited
        mark x as visited
        Add x to the queue
      mark ω as processed.

* Apply BFS to the following graph (or)

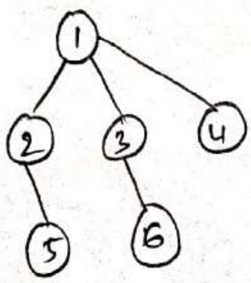
Find the spanning tree of the following graph using BFS?



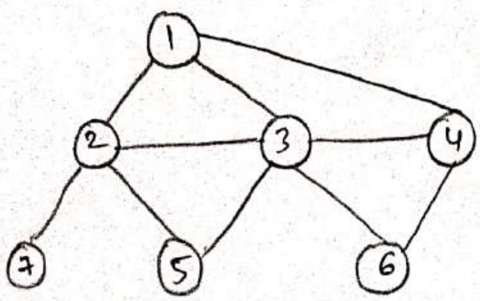
Graph

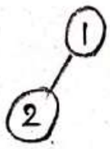
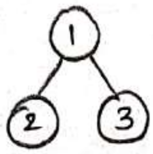
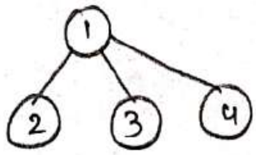
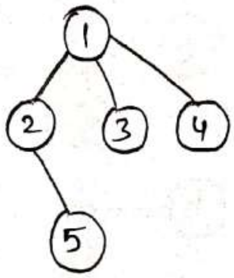
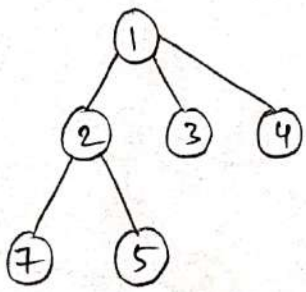
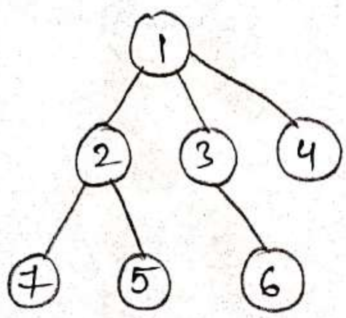
	Queue	Spanning Tree	BFS Traversal
1)	1	–	–
2)	~~1~~ 2 3 4		1
3)	~~2~~ 3 4 5		1,2
4)	~~3~~ 4 5 6		1,2,3
5)	~~4~~ 5 6		1, 2, 3,4
6)	~~5~~ 6		1,2,3,4,5
7)	~~6~~		1,2,3,4,5,6

∴ The queue is empty and hence the BFS traversal is 1,2,3,4,5,6 and spanning tree is

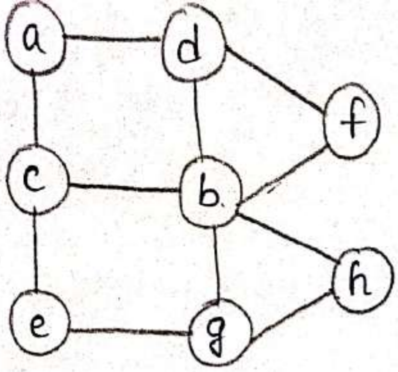


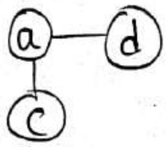
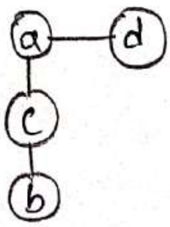
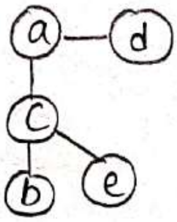
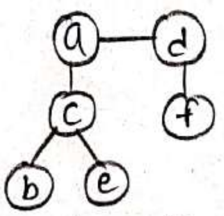
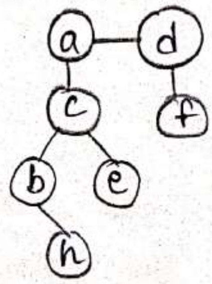
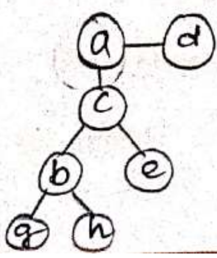
* Eg-2



	Queue	Spanning tree	BFS traversal
1)	1	-	-
2)	~~1~~ 2 3 4		1
3)	~~2~~ 3 4 5 7		1, 2
4)	~~3~~ 4 5 7 6		1, 2, 3
5)	~~4~~ 5 7 6		1, 2, 3, 4
6)	~~5~~ 7 6		1, 2, 3, 4, 5
7)	~~7~~ 6		1, 2, 3, 4, 5, 7
8)	~~6~~		1, 2, 3, 4, 5, 7, 6

Eg:- 3



Queue	S.T	BFS
1) ~~a~~	—	—
2) ~~a~~ c d		a
3) ~~c~~ d b e		a, c
4) ~~d~~ b e f		a, c, d
5) ~~b~~ e f h g		a, c, d, b
6) ~~e~~ f h g		a, c, d, b, e
7) ~~f~~ h g		a, c, d, b, e, f
8) ~~h~~ g		a, c, d, b, e, f, h
9) g		a, c, d, b, e, f, h, g

* Breadth First Search (BFS): Pseudo code

// Let G is the graph & is the source node

```
Algorithm BFS(G, S)
{
    Let Q be queue.
    Q.Enqueue(s)
    Mark s as visited.
    while (Q is not empty)
    {
        V = Q.Dequeue( )
        ~~for each~~
        if (w is not visited)
        {
            Q.Enqueue(w)
            Mark w as visited.
        }
        print(v);
    }
}
```

* Analysis:-

Graph with n vertices, m edges	Adjacency matrix	Adjacency List
Space required	$O(n^2)$	$O(n+m)$
Time required	$O(n^2)$	$O(n+m)$

* Depth First Search (DFS):-

The Strategy followed by Depth-First Search is to search deeper in the graph whenever possible.

Main idea:

Mark s as visited. Push s on to the stack.

Repeat the following till the stack is empty:

Pop a vertex (say w) from the stack.

For each neighbour x of w that is not yet visited

Mark x as visited.

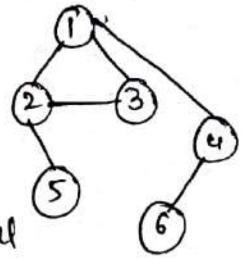
Add x to the stack.

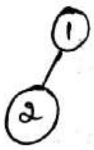
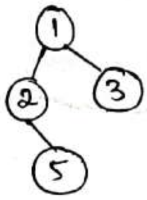
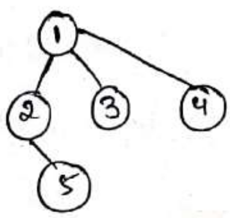
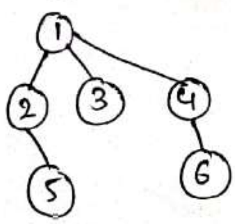
Mark w as processed.

* Apply DFS to the following graph:

or

Find the ST of DFS for following:



	Stack	S.T	DFS Traversal
1)	1	–	–
2)	~~1~~ → 2, 3, 4	1	1
3)	~~2~~, 3, 4 → 5, 3, 4		1, 2
4)	~~5~~, 3, 4.		1, 2, 5
5)	~~3~~, 4		1, 2, 5, 3
6)	~~4~~ → 6		1, 2, 5, 3, 4
7)	~~6~~		1, 2, 5, 3, 4, 6

The strategy followed by depth-first search is to search deeper in the graph whenever possible.

* Recursive DFS pseudo code:-

```
// Let G is graph & s is source vertex
DFS(G,S)
{
    Mark S as visited
    for each neighbour w of s in graph G
        if (w is not visited)
            DFS (G,V)
}
```

* Depth first Search (DFS): pseudo code

```
// Let G is graph & s is source x1, x2
Algorithm DFS(G,S)
{
    Let S be stack
        S. push(s)      // Adding s to stack
    Mark s as visited
    while (S is not empty)
      {
        V = S.pop()
        //push all the neighbors of v in stack that are not visited
        for each neighbor w of v in graph G
            if (w is not visited)
            {
                S.push(w)
                Mark w as visited
            }
            print(v)   // Vertex V processed
      }
}
```

* Analysis

Graph with n vertices, m edges	Adjacency matrix	Adjacency List
Space required	$O(n^2)$	$O(n+m)$
Time required	$O(n^2)$	$O(n+m)$

* Disjoint sets:-

- A disjoint set S is a collection of sets $S_1 \ldots S_n$ where $\forall i \neq j \; S_i \cap S_j = \emptyset$
- Each set has a representative which is a member of set (usually the min if the elements are comparable)

* Operations on disjoint sets:-

1) **Make - Set(x):-**

Creates a new set where x is it's only element (& it is the representative of set)

2) **Union(x, y):-**

Unites the dynamic sets that contain x & y say S_x & S_y into a new set which is the union of 2 sets.

3) **Find(x):-**

Return the representative of the set containing x.

Eg: Maintain a set of pairwise disjoint sets.

$\rightarrow \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$

- Each set has a unique name, one of its members.

$\{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$

- Union(x, y) - take the union of 2 sets named x & y

$\rightarrow \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}$

$\rightarrow$ Union(5, 1)

$\rightarrow \{3,5,7,1,6\}, \{4,2,8\}, \{9\}$

• Find (x) — return the name of set containing x

$\{3, 5, 7, 1, 6\}$, $\{4, 2, 8\}$, $\{9\}$

find(1) = 5

find(4) = 8

find(9) = 9

• Applications:-

→ Finding no. of connected components in a graph.

→ Finding minimum spanning tree.

* How to represent disjoint sets?

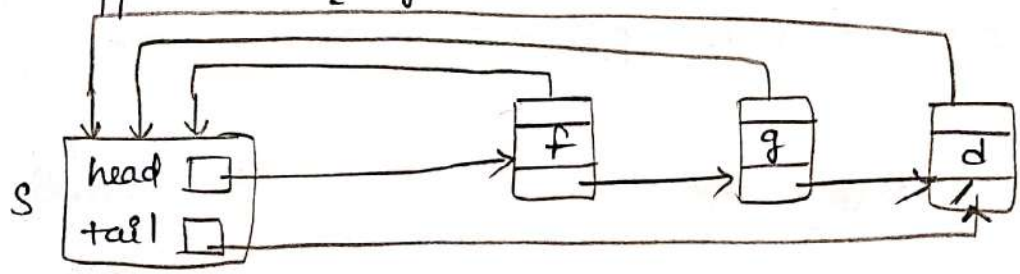
There are mainly 2 representations for disjoint sets

① Linked List representation ② Rooted tree representation

1) Linked List representation:-

The representative is set member in the 1st object in the list.

Eg: Suppose $S = \{f, g, d\}$



2) Rooted tree representation:-

The root of each tree contains the representation and is it's own parent.

Eg: Suppose $S = \{f, g, d\}$

Representation of sets
f as representative

//x: The element for which set/tree to be created

Algorithm MakeSet(x)

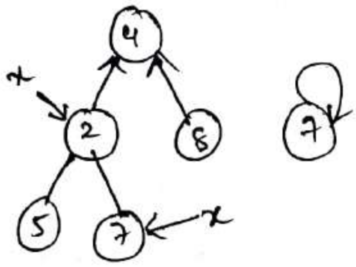
{

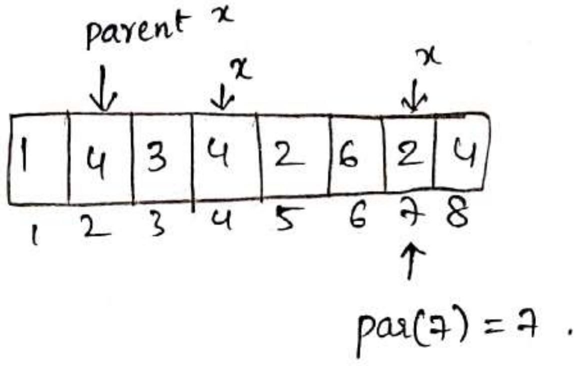
parent[x] = x;

}

Scanned with OKEN Scanner

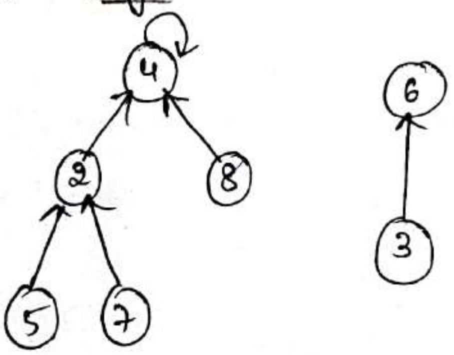
```
// Returns root of the tree which containes x
Algorithm Find(x)
{
    while (parent[x]!=x)
        x= parent [x];
    return(x);
}
```



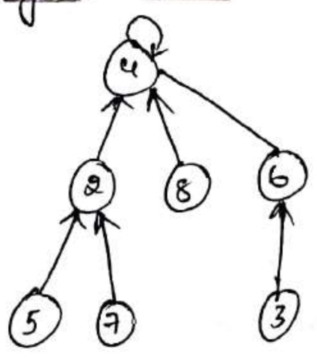



```
// Returns root of the tree which contains x
Algorithm Union(x, y)
{
    u= Find(x); // finding root of tree which contains x
    v= Find(y); // finding root of tree which contains y
    if(u!=v)
        parent[v]=u; // Attaching tree v to tree . u .
}
```

eg: Before union



after union



unit-2