→ Graph Data Structure: A graph is a non-linear data structure which consists of 2 components:

1. A finite set of vertices called Nodes.
2. A finite set of ordered pair of the form (u,v) called as edges. The pair is ordered because (u,v) is not same as (v,u) in case of a directed graph. The pair of the form (u,v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

→ Representation:

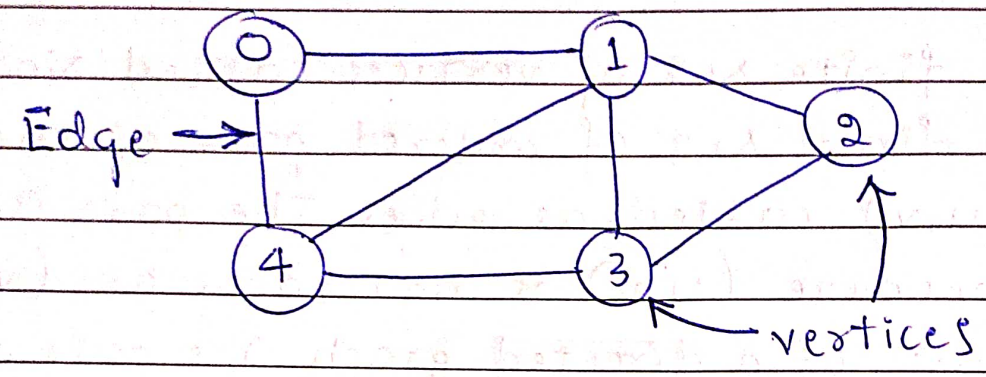
1. Adjacency Matrix
2. Adjacency List

*. Adjacency Matrix: is a 2D Array of size V x V, where, V is the number of vertices in a graph. Let the 2D array be A[][], a slot A[i][j] = 1, indicates that there is an edge from vertex i to j.

Adjacency matrix for Undirected graph is always a Symmetric matrix.

- A symmetric matrix is a square matrix that is equal to the transpose i.e., $A = A^T$

ex:



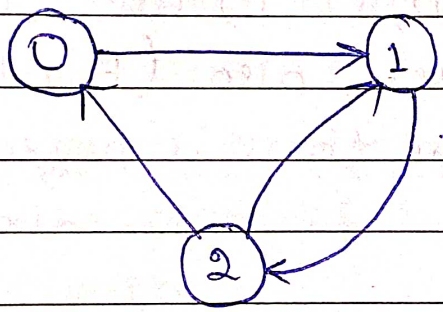
The Adjacency matrix for the above graph is,

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

X pros: Representation is easier to implement and follow. Removing an edge takes O[1] time (as we just have to make the element at A[i][j] to 0). Queries like checking whether there is an edge from vertex 'u' to vertex 'v' are efficient & can be done in O(1)

* Cons: It consumes more memory i.e, it takes $O(V^2)$, where V is the number of vertices. Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex v takes $O(V^2)$ time.

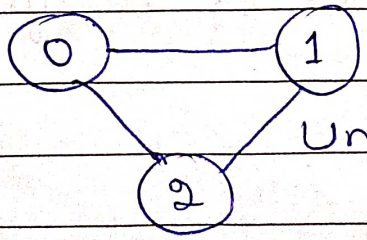
→ Directed Graph: A graph with directed edges is known as directed graph.



Directed edges.

Here, in the above graph the edge pair (2,0) is not same as (0,2).

→ Undirected Graph: A graph which has unordered pair of edges.

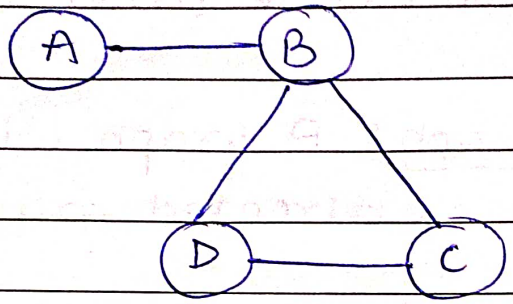


Undirected edges.

Here, in the above graph, the edge pair (2,0) is same as (0,2), we can traverse in both the

directions.

⇒ Adjacent Vertices: are the two vertices with a direct edge connecting them.



Here, (A,B) (B,D) (B,C) and (C,D) are the adjacent vertices, also (A,C) are not adjacent vertices because there is no direct edge from A to C or C To A.

⇒ Degree of a Vertex: There are 2 types of degrees.

1. Indegree: The Number of incoming edges to a node.

2. Outdegree: The Number of outgoing edges from a node.

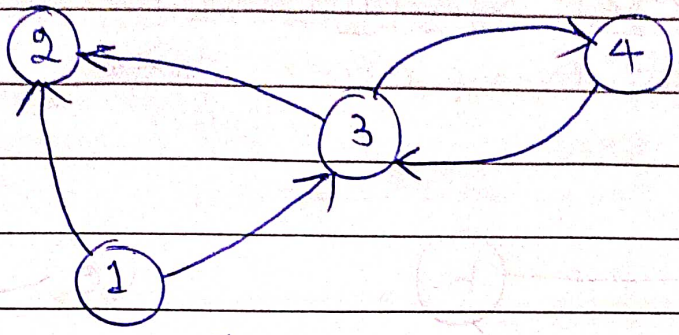


Fig: 1.1:

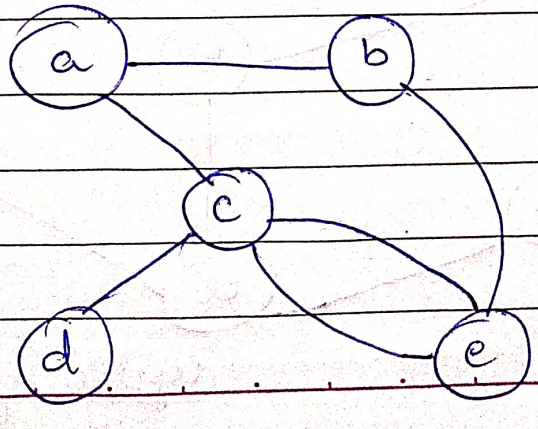
- The indegree of (1) is 0 as there is no incoming edges. similarly, the outdegree of (1) is 2 as there are 2 outgoing edges.

→ path between two vertices: All the vertices which comes in the path of two given vertices is called as path.

In the above graph (fig 1.1) the path from vertex (1) to (4) is 1 → 3 → 4.

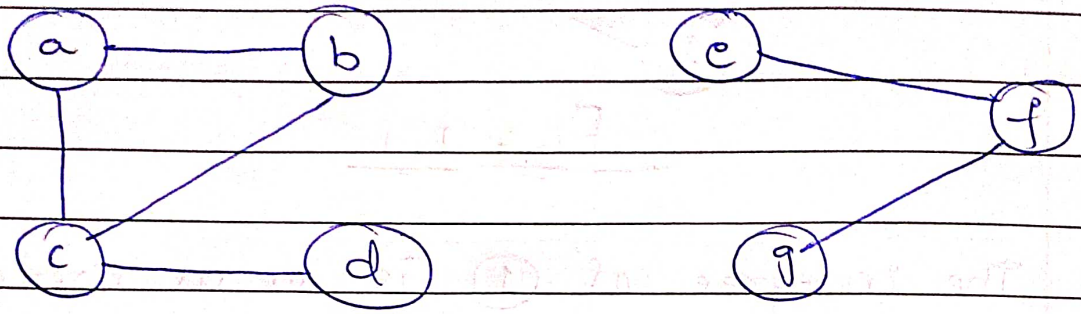
→ connected graph: Here, Each node has path from every other node.

Ex:



⇒ Disconnected Graph: Each node does not have path from every other node.

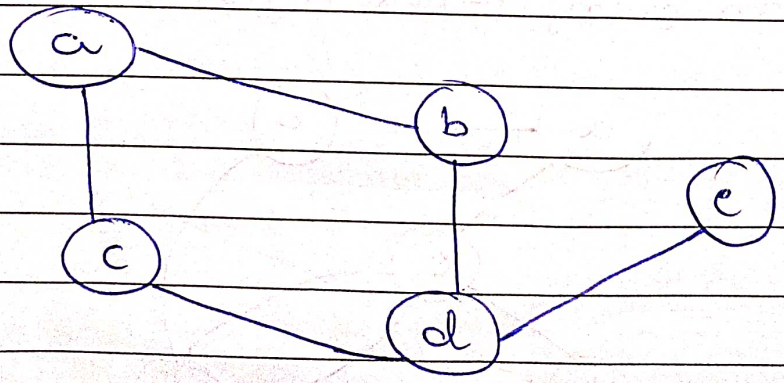
ex:



In the above graph, we cannot go from vertex a to f or vice-versa.

⇒ connected component: Each connected graph in a disconnected graph is called a connected component. In the above graph we have 2 connected components.

⇒ Cycle in a graph: path from a vertex to the same vertex in a graph is known as cycle.



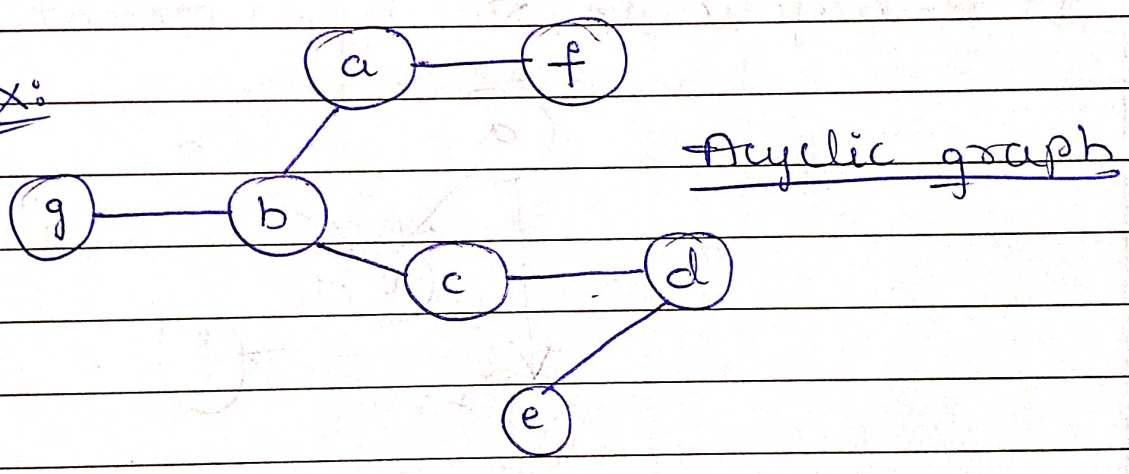
cyclic graph

In the above graph if starts from (a) go to b → d → c → (a) then again we are back to (a) so there is a cycle.

1. cyclic Graph: A graph which contains cycle is called cyclic graph.

2. Acyclic Graph: A graph which does not contain cycle is called acyclic graph.

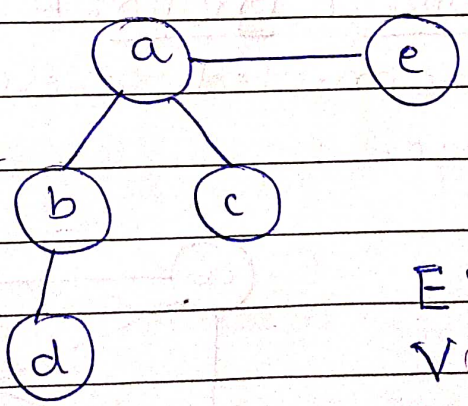
Ex:



Acyclic graph

→ Tree: is a connected acyclic graph.

* The total no. of edges is equal to the no. of vertices - 1.



E: edges
V: vertices

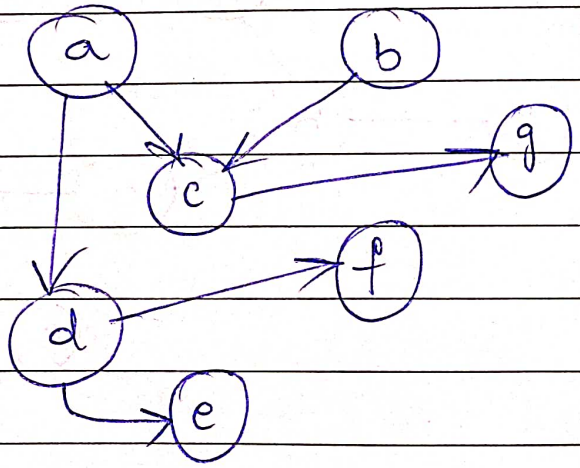
$E = V - 1$

properties of tree with n nodes:

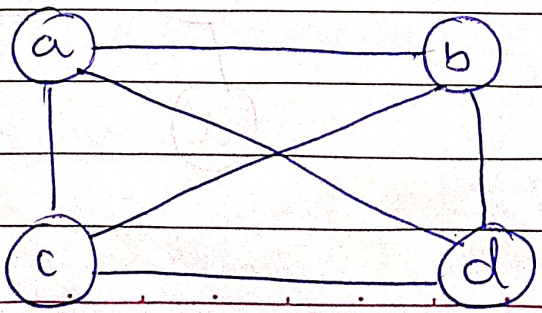
1. Number of edges = n-1
2. There are no cycles present in the graph.
3. Each node has a path from every other vertex.

Directed Acyclic Graph [DAG]: properties

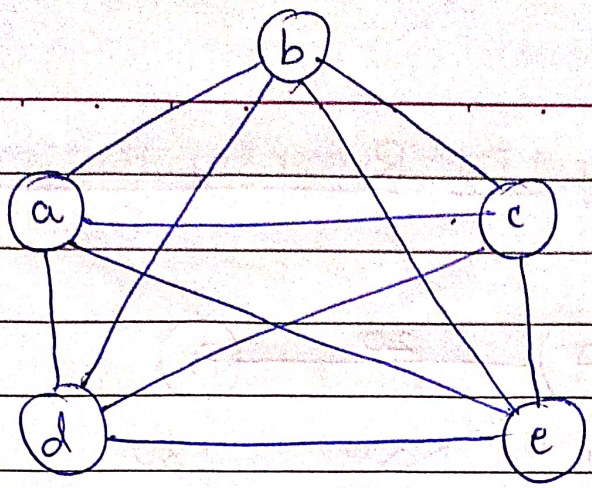
1. It has directed edges
2. It is acyclic i.e., No. cycles present.



complete Graph: A graph in which each vertex is connected to every other vertex by a direct edge.



eg: 1.1

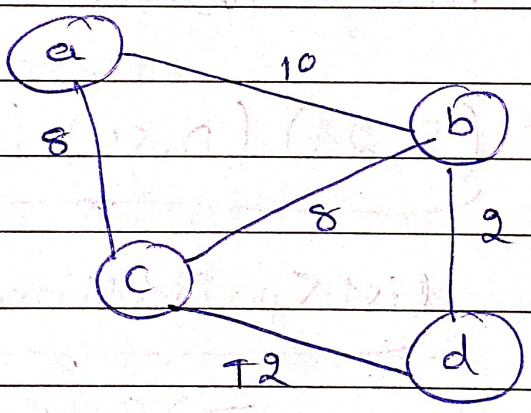


eg: 1.2

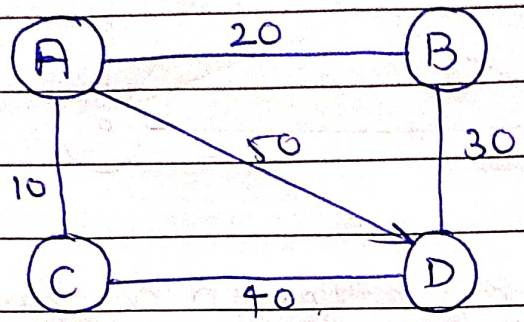
The number of edges in a compelete graph is
$^nC_2 = (n*(n-1))/2$

⇒ Weighted Graph: A graph with weighted edges is called a weighted graph.

ex:



⇒ Adjacency List Representation Using Hashmap



So, in the above graph the type of vertices is String. So, while creating or adding an edge from source (x) to destination (y) with weight w, and isbidirectional.

A → (B,20) (D,50) (C,10)

Key — list<pair<String, int>>

value.

hashmap.

The function could be,

```
g.addEdge("A", "B", 20, true);
```

x y w isbidirectional

7) Breadth First Search Traversal: on a Adjacency matrix: [2D Array]

1. Create a Boolean Array visited of size V, where V is the number of nodes (vertices).
2. Initialize the visited Array to false.
3. Take the source node
4. create a queue.
5. Add the source vertex (node) to the queue & mark it as visited.
6. Now, Add the vertices which are connected to source vertex to the queue and do the same for each vertex.

```
void BreadthFirstSearch(int src)
{
    bool visited[V];
    memset(visited, 0, sizeof(visited));
    queue<int> q;

    q.push(src);
    visited[src] = true;

    while(!q.empty())
    {
```

```
int node = q.front();
q.pop();

cout << node << ", ";

for (int i=1; i<=V; i++)
{
    if ( A[node][i] && !visited[i])
    {
        q.push(i);
        visited[i] = true;
    }
}
}
}
```

⇒ Breadth First Search traversal : [on a List]

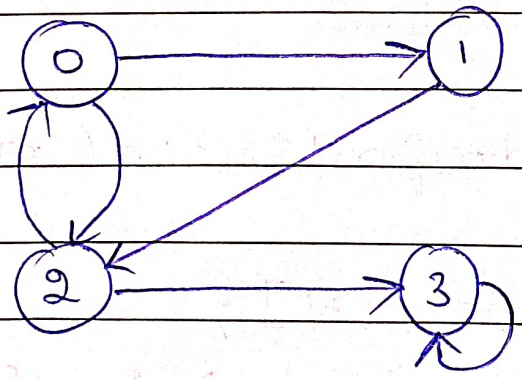
1. create a boolean Array visited of size V.
2. Take the source node
3. create queue
4. Add source node to the queue & mark as visited
5. while queue is not empty pop from front of the queue and

-> Adjacency List Representation:

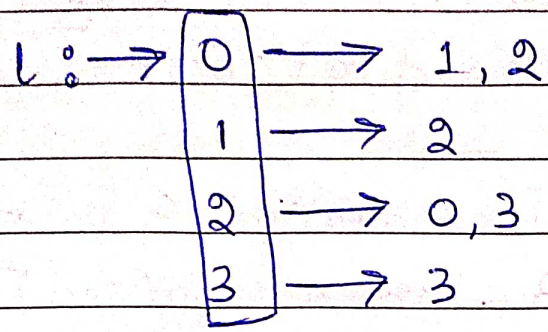
```
class Graph {

    int V;    // V: Number of vertices.
    map<int, list<int>> l;

    // Here, in the above map declaration, the key
    // is the node and the value is the list of
    // all the nodes which are directly connected.

    void addEdge (int x, int y)
    {
        l[x].push_back(y);
    // It means that the node x is directly
    // connected with y.
    }

    void breadthfirstSearch (int src)
    {
        bool visited[V];
        memset (visited, 0, sizeof(visited));
        queue<int> q;
        q.push(src);
        visited[src] = true;
```

```
while (!q.empty())
{
    int node = q.front();
    q.pop();
    cout << node << " ";

    for (auto nbr : l[node])
    {
        if (!visited[nbr])
        {
            q.push(nbr);
            visited[nbr] = true;
        }
    }
}
```

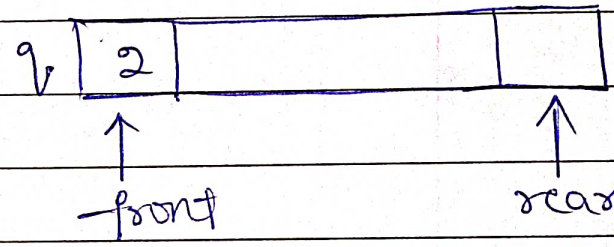
ex:



The Adjacency list for above graph is,



- similary, the BFS of above graph with (2) as src node is,



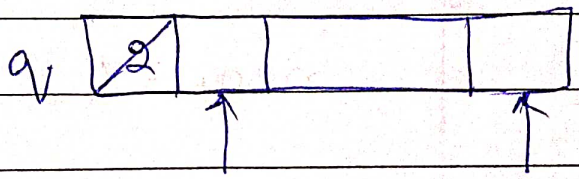
(1) src = 2

	0	1	2	3
visited	f	f	T	f

Now, queue is not empty, so it will enter into the while loop.

node = 2

Updated Queue,



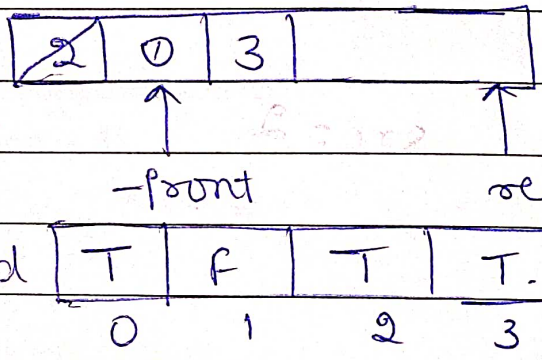
l[node] = 0, 3

nbr.

So, now visited of 0, is false. so will add 0 to the queue & visited[0] to true.

Now, nbr will get incremented & points to the next location i.e, 3, here, also visited[3] is false so, 3 will be added to queue and visited[3] = true

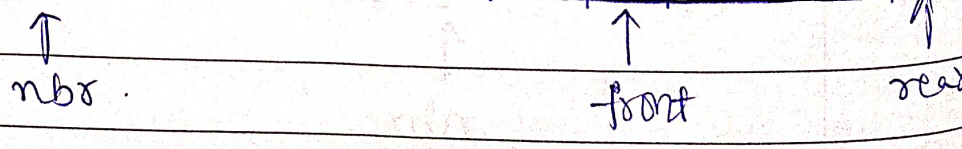
Updated Queue & visited Array contents.



Now, again nbr will be incremented & the contents of l[node] is over so it will come out of the for loop. node will get update with q.front() as,

node = 0

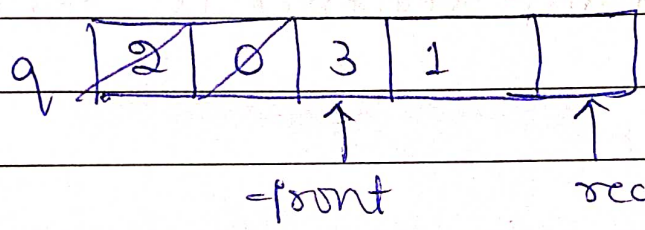
l[node] = 1, 2

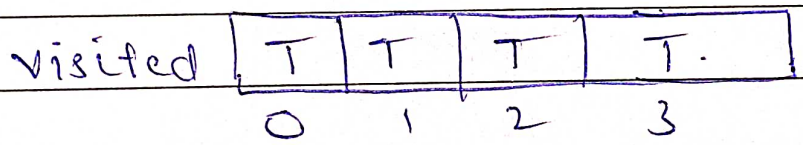


Updated, q

Visited of 1 is false, so 1 will be added to queue and visited[i] is marked true. nbr will now point to 2. as visited of 2 is true, nbr is again incremented, & it will come out of the for loop as l[0] is over

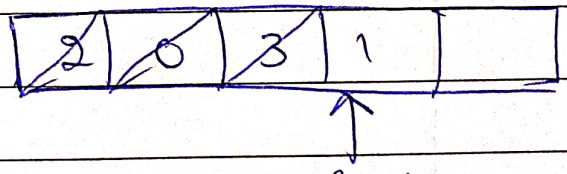
Update queue & visited Array contents are,





Now, node will get update with q.front() as,

node = 3



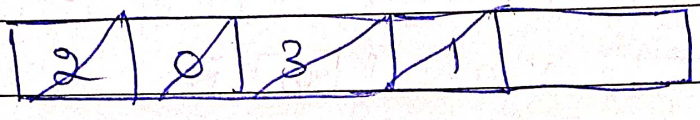
l[node] = l[3] = 3

nbr

At the visited[3] is true, nbr is incremented & it will come out of the for loop

Now, node = q.front() = 1

l[node] = l[1] = 2 & 2 is already visited.



NOw, queue is empty, so the BFS is complete & the traversal sequence is 2, 0, 3, 1.