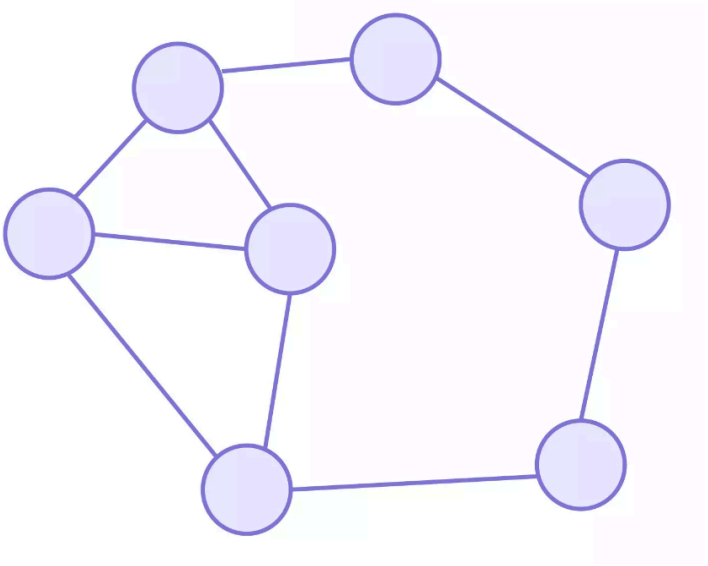
Practical–8
Implementation of Graph

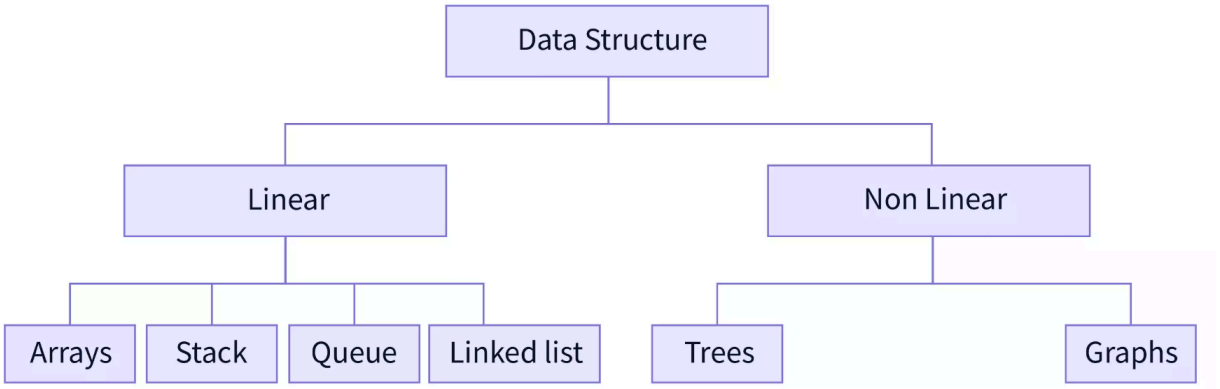
"A Graph is a non-linear data structure that consists of nodes and edges which connects them".

A graph data structure is a collection of nodes that consists of data and are connected to other nodes of the graph.



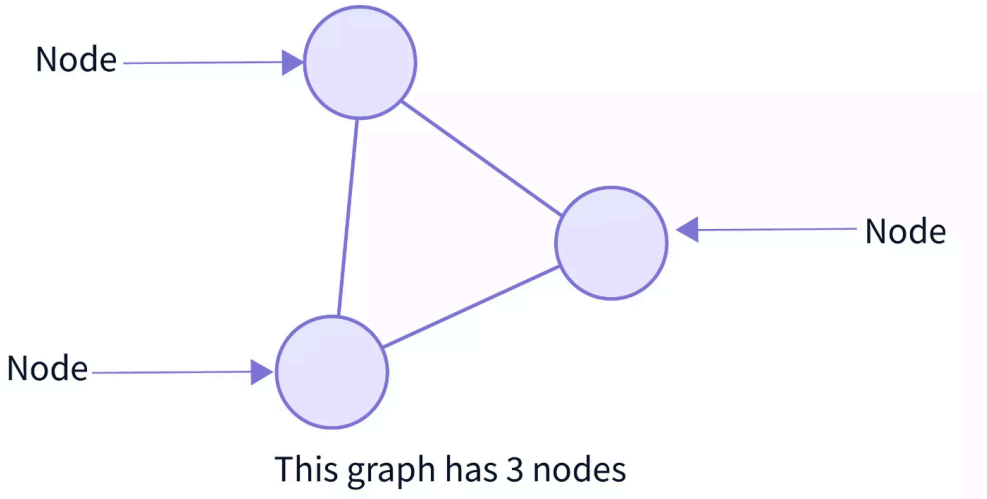
Non-linear Data Structure

In a non-linear data structure, elements are not arranged linearly or sequentially. Because the non-linear data structure does not involve a single level, an user cannot traverse all of its elements at once.



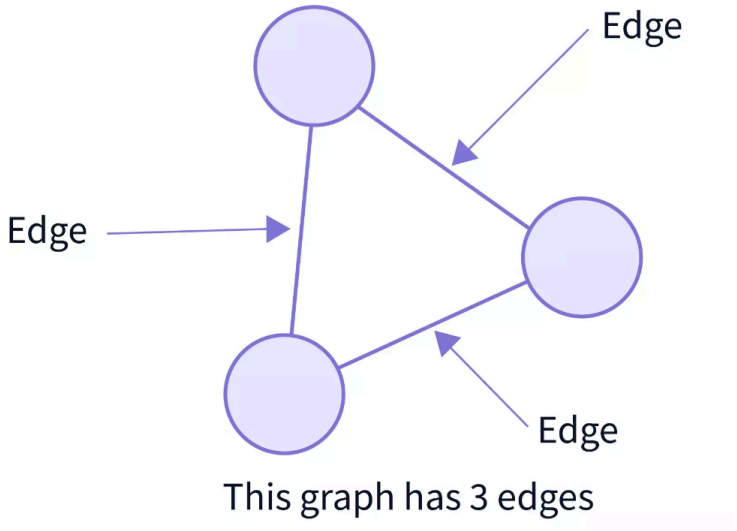
Nodes:

- Nodes create complete networks in any graph.
- They are one of the building blocks of a graph data structure.
- They connect the edges and create the main network of a graph.
- They are also called vertices.
- A node can represent anything such as any location, port, houses, buildings, landmarks, etc.
- They basically are anything that you can represent to be connected to other similar things, and you can establish a relation between them.



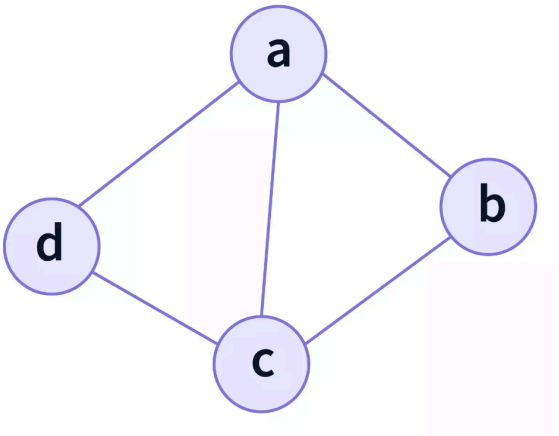
Edges:

Edges basically connect the nodes in a graph data structure. They represent the relationships between various nodes in a graph. Edges are also called the path in a graph.



A graph data structure (V,E) consists of:

- A collection of vertices (V) or nodes.
- A collection of edges (E) or path



A graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. In the above graph:

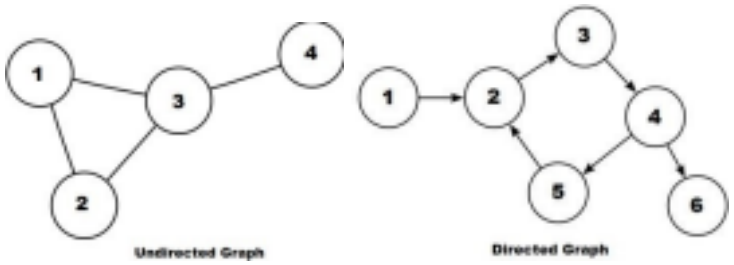
```
V = {a, b, c, d}

E = {ab, ac, ad, bc, cd}
```

In the above graph,
|V| = 4 because there are four nodes (vertices) and,
|E| = 5 because there are five edges (lines).

Types of Graphs in Data Structure

The most common types of graphs in data structure are mentioned below:

1. Undirected: A graph in which all the edges are bi-directional.
The edges do not point in a specific direction.

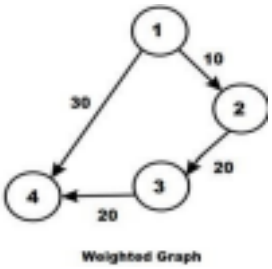


2. Directed: A graph in which all the edges are unidirectional. The edges point in a single direction.

3. Weighted Graph: A graph that has a value associated with every edge. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. Weighted graphs are typically used in modeling computer networks.

An edge in a weighted graph is represented as (u, v, w), where:

- u is the source vertex
- v is the destination vertex
- w represents the weight associated to go from u to v



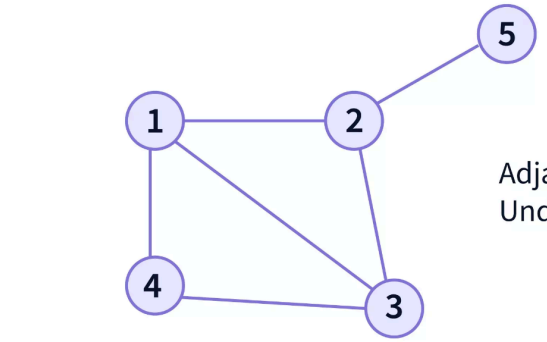
4. Unweighted Graph: A graph in which there is no value or weight associated with the edge. All the graphs are unweighted by default unless there is a value associated.

An edge of an unweighted graph is represented as (u, v), where:

- u represents the source vertex
- v is the destination vertex Graph Representation in Data Structure

Adjacency Matrix

An Adjacency Matrix is the simplest way to represent a graph. It is a 2D array of V x V vertices with each row and column representing a vertex. The matrix consists of "0" or "1". 0 depicts that there is no path while 1 represents that there is a path.



Adjacency Matrix for Undirected Graph

Nodes →

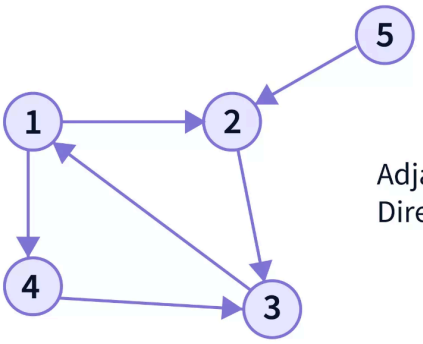
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	1
3	1	1	0	1	0
4	1	0	1	0	0
5	0	1	0	0	0

Nodes ↑

Operations on Graph in Data Structure

Following are the basic graph operations in data structure:

- Add/Remove Vertex – Add or remove a vertex in a graph.
- Add/Remove Edge – Add or remove an edge between two vertices. • Check if the graph contains a given value.
- Find the path from one vertex to another vertex.



Adjacency Matrix for Directed Graph

Nodes →

Nodes ↑	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	1	0	0
5	0	1	0	0	0

Pros of Adjacency Matrix

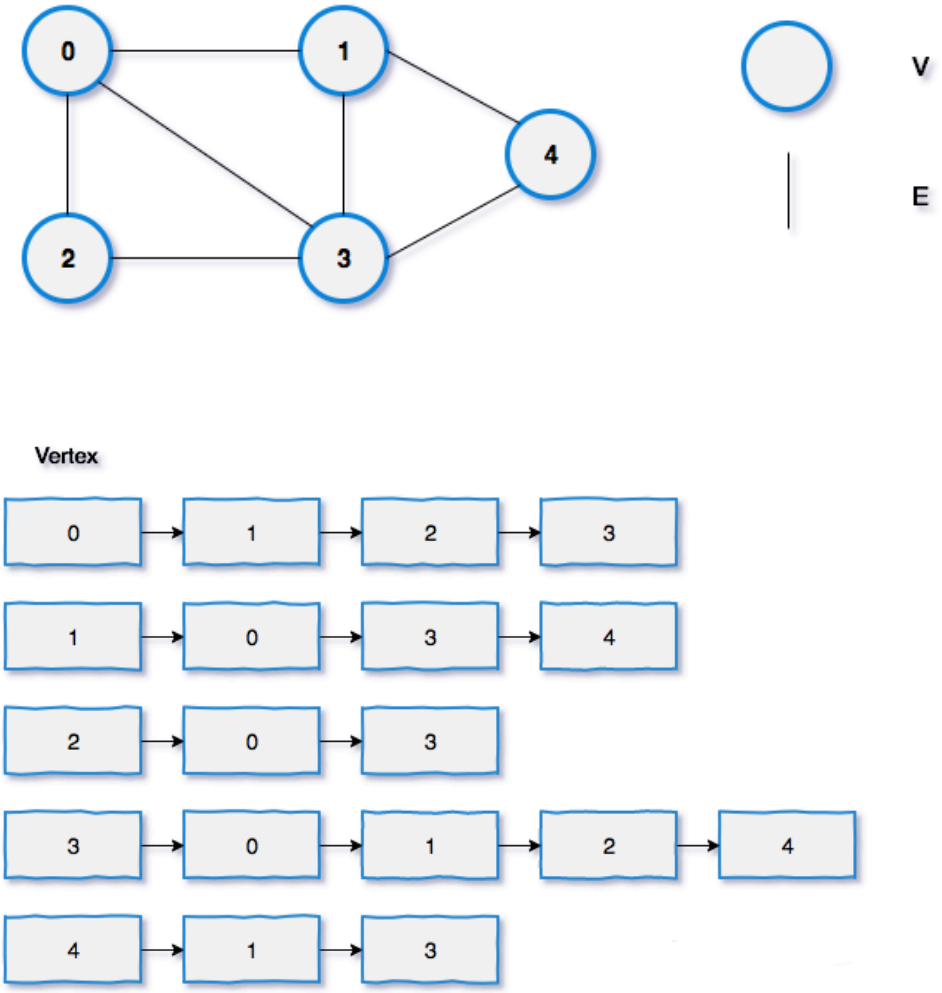
- Adjacency Matrices performs the basic operations like adding an edge, removing an edge, and checking whether there is an edge from node i to node j very efficiently, and in constant time usually.
- For dense graphs, where the number of edges are very large, adjacency matrices are the best choice. Because, in big-O terms they don't take up more space, and operations are much faster. Maximum of the cells of the matrix are filled because of more number of edges, hence it is very space efficient. If the graph is sparse, then most of the cells are vacant, hence wasting more space.

Cons of Adjacency Matrix

- The most notable disadvantage that comes with Adjacency Matrix is the usage of VxV space, where V is the number of vertices. In real life, we do not have such dense connections ,where so many edges will be used. So, the space mostly gets wasted, and hence for sparse graphs with less number of edges Adjacency Lists are always preferred.
- Operations like inEdges(to check whether there is an edge directing towards this node) and outEdges(to check whether there is an edge directing out from this node) are expensive in Adjacency Matrix (will study ahead).

Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a node.
- Each element in the linked list represents the nodes that are connected to that node by an edge.



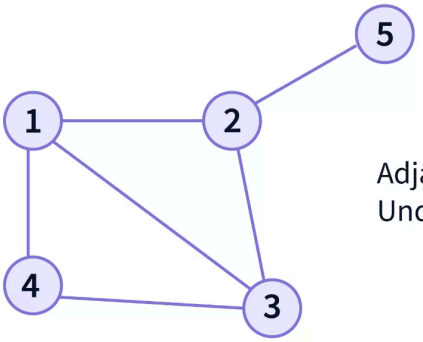
Pros of Adjacency list

- Since, we only store the value for the edges in the linked lists, the adjacency lists are efficient in terms of storage(for sparse graphs). So, if for some graph we have 1000 edges or 1 edge, we can represent them very efficiently without wasting any space, because we will use the list for storage.
- Since the adjacency lists are storage efficient, they are useful for storing sparse graphs. Sparse graphs are the graphs, which have the edges much lesser than the number of edges expected.

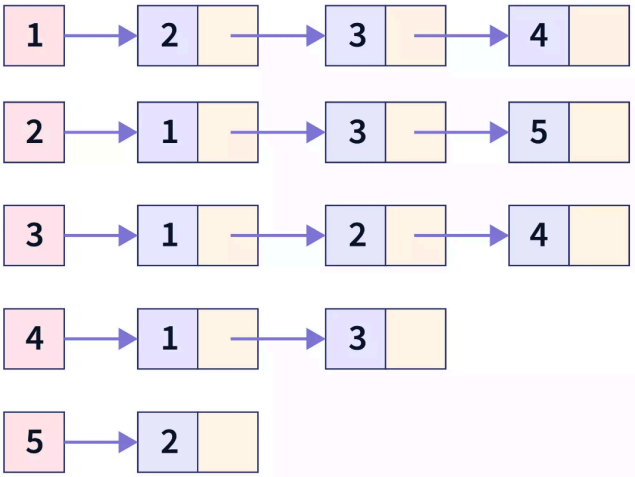
- Adjacency list helps to find all the nodes next to any node easily. Because, a node, points to all the other nodes which are connected to it, hence it becomes very simple to find out all the adjacent nodes.

Cons of Adjacency list

- To find any particular node in an adjacency list, we need to explore all the connected nodes. So, usually it is slower as compared to finding a node in an adjacency matrix.
- It is not a good option for dense graphs because we will unnecessarily be storing so many edges in the linked list and hence it will neither be memory efficient, nor time efficient in terms of certain operations(like search or peek).



Adjacency List for
Undirected Graph



Graph Traversal in Data Structure

The process of finding all vertices/nodes in a graph is known as graph traversal. The order of the vertices visited during the search process is also determined by the graph traversal. We have a few major standard techniques to traverse on non-linear tree and graph data structure they are majorly classified into the,

- Breadth first traversal
- Depth first traversal

Graph traversal is the process of visiting or updating each vertex in a graph. The order in which they visit the vertices is used to classify the traversals. There are two ways to implement a graph traversal:

1. Breadth-First Search (BFS) – It is a traversal operation that horizontally traverses the graph. It traverses all the nodes at a single level before moving to the next level. It begins at the root of the graph and traverses all the nodes at a single depth level before moving on to the next depth level.

```
Step 1:   SET STATUS=1 (ready state)
          for each node in G
Step 2:   Enqueue the starting node A
          and set its STATUS=2
          (waiting state)
Step 3:   Repeat Steps 4 and 5 until
          QUEUE is empty
Step 4:   Dequeue a node N. Process it
          and set its STATUS=3
          (processed state).
Step 5:   Enqueue all the neighbours of
          N that are in the ready state
          (whose STATUS=1) and set
          their STATUS=2
          (waiting state)
          [END OF LOOP]
Step 6: EXIT
```

2. Depth-First Search (DFS): This is another traversal operation that traverses the graph vertically. It starts with the root node of the graph and investigates each branch as far as feasible before backtracking.

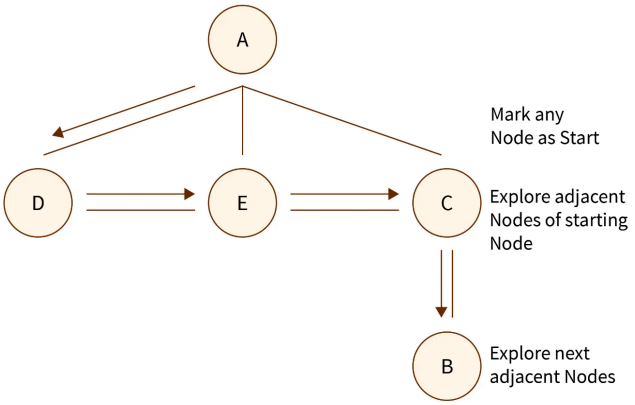
```
Step 1:   SET STATUS=1 (ready state) for each node in G
Step 2:   Push the starting node A on the stack and set
          its STATUS=2 (waiting state)
Step 3:   Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS=3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS=1) and
          set their STATUS=2 (waiting state)
          [END OF LOOP]
Step 6:   EXIT
```

Breadth-First Search

Breadth-first search (BFS) is a fundamental graph traversal algorithm used to explore nodes in a graph. It systematically visits all nodes at a certain depth before moving on to the next level. By employing a queue data structure, BFS ensures that nodes are visited in a breadthward motion, making it particularly useful for finding the shortest path in unweighted graphs.

Algorithm of BFS

- Initiate the process by appending any vertex from the graph to the rear of the queue.
- Extract the front item from the queue and include it in the list of visited nodes.
- Generate nodes for the adjacent vertices of the current node and insert those that have not yet been visited.
- Repeat steps two and three until the queue becomes empty.



Create a Queue (say Q)
Mark the initial Vertex V => Visited.
Add V to the Queue-Q

WHILE Queue-Q is not empty:
    Remove an element from the front of Queue-Q
  Store that element in Vertex (say U)
    Loop over all the Unvisited Neighbors of Vertex-U
        Mark the neighbor as Visited
        Add the Neighbor to the Q.

Applications of BFS

- Shortest Path Finding: Breadth-First Search (BFS) in graph algorithms is utilized to find the shortest path between two vertices in an unweighted graph.
- Network Analysis: BFS aids in traversing and analyzing networks, such as social networks or the internet, to understand connectivity patterns.
- Minimum Spanning Tree (MST): BFS is utilized to construct a Minimum Spanning Tree, ensuring the most cost-effective connection of all vertices in a graph, commonly applied in network design and optimization.

Depth-First Search (DFS):

The traversal or searching begins with the arbitrary starting node, we can choose any node to be the starting node as per our requirement all traversals would be a valid DFS traversal. After deciding on the starting node, explore any of the adjacent nodes of the starting node, and before traversing the other remaining adjacent node of the starting node the algorithm explores all possible adjacent nodes in the single path till its most possible depth. When there is nothing left to explore or can say all nodes are already visited, the algorithm reverts back to the starting node and then follows the same strategy to discover other remaining nodes which haven't been traversed yet.

```
depthFirstSearch(graph, startingVertex)
    make startingVertex as visited

     // Loop on each adjacent vertices of startingVertex
    for each adjacentVertex of startingVertex
        if nextStartingVertex is not visited
            // Call the DFS
            depthFirstSearch(graph, adjacentVertex)

main()
{
    // Initialization
    for each vertex in Graph
        make vertex as not visited


     // Run DFS for each vertex to cover all disconnected
Components
    for each vertex in Graph
        if vertex is not visited
            depthFirstSearch(Graph, vertex)
}
```

Depth-First Search (DFS):

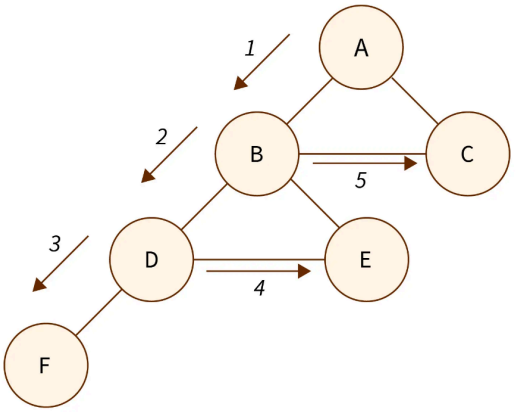
```
#include <stdio.h>
#define MAX 8
void depth_first_search(int adj[][MAX],int visited[],int
start)
{
   int stack[MAX];
   int top=-1, i;
   printf("%d-",start);
```

```
   visited[start] = 1;
   stack[++top] = start;
   while(top != -1)
   {
      start = stack[top];
      for(i = 0; i < MAX; i++)
      {
            if(adj[start][i] && visited[i] == 0)
            {
            stack[++top] = i;
            printf("%d", i);
            visited[i] = 1;
            break;
            }
      }
      if(i == MAX)
      top--;
   }
}
int main()
{
   int visited[MAX] = {0}, i, j;
   int adj[MAX][MAX];
   printf("\n Enter the adjacency matrix: ");
   for(i = 0; i < MAX; i++)
      for(j = 0; j < MAX; j++)
      scanf("%d", &adj[i][j]);

   printf("DFS Traversal: ");
   depth_first_search(adj,visited,0);
   printf("\n");
   return 0;
}
```

Depth-first search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It employs a stack data structure to maintain a path and explores as deeply as possible before backtracking. DFS is often used in topological sorting, solving maze problems, and detecting cycles in graphs due to its exhaustive nature.

Algorithm of DFS

- Begin by placing a vertex from the graph at the top of the stack.
- Extract the top item from the stack and mark it as visited.
- Generate a list of adjacent nodes for the current vertex and add them to the top of the stack if they haven't been visited yet.
- Keep iterating over steps 2 and 3 until the stack has been fully emptied.



Traversal = A → B → D → F → E → C

Create a stack (say S)
Mark the initial Vertex V => Visited.
Push V onto the stack-S

WHILE stack-S is not empty:
    Pop an element from the front of stack-S
  Store that element in Vertex (say U)
    Loop over all the Unvisited Neighbors of Vertex-U
        Mark the neighbor as Visited
        Push the Neighbor to the S.

Applications of DFS

- Locating any route between the graph's vertices u and v.
- Determining how many connected elements are present in an undirected graph.
- Sorting a given directed graph topologically.
- Finding components with strong connections in the directed graph.
- In the given graph to find cycles or to determine whether the provided graph is bipartite.

Breadth-First Search (BFS)

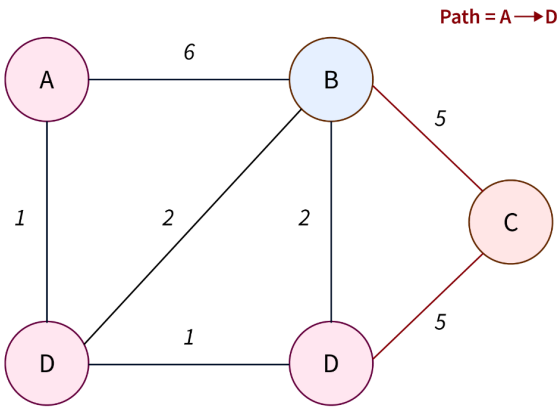
```
#include <stdio.h>
#define MAX 8
void breadth_first_search(int adj[][MAX],int visited[],int
start)
{
   int queue[MAX],rear = -1,front =-1, i;
   queue[++rear] = start;
   visited[start] = 1;
   while(rear != front)
   {
      start = queue[++front];
      if(start == 4)
      printf("5\t");
      else
      printf("%d \t",start);
      for(i = 0; i < MAX; i++)
      {
            if(adj[start][i] == 1 && visited[i] == 0)
            {
                  queue[++rear] = i;
                  visited[i] = 1;
            }
      }
   }
}
int main()
{
   int visited[MAX] = {0};
   int adj[MAX][MAX], i, j;
   printf("\n Enter the adjacency matrix: ");

   for(i = 0; i < MAX; i++)
      for(j = 0; j < MAX; j++)
            scanf("%d", &adj[i][j]);

   breadth_first_search(adj,visited,0);
   return 0;
}
```

Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm is a graph search algorithm used to find the shortest path between nodes in a weighted graph. It operates by iteratively selecting the node with the shortest distance from a source node and updating the distances to its neighboring nodes accordingly. Dijkstra's algorithm is widely used in various applications such as routing protocols in computer networks and navigation systems in transportation.

Algorithm of Dijkstra's

- Initialize all vertices to infinity, except the source vertex.
- Push the source vertex into a priority queue in the format (distance, vertex), setting its distance to 0.
- Extract the vertex with the minimum distance from the priority queue.
- Update distances after extracting the minimum distant vertex by considering the condition (current vertex distance + edge weight < next vertex distance).
- If a visited vertex is extracted, proceed to the next iteration without utilizing it.
- Continue steps 3 to 5 until the priority queue becomes empty.



Vertex	Shortest Distance from A	Previous Vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Pseudocode

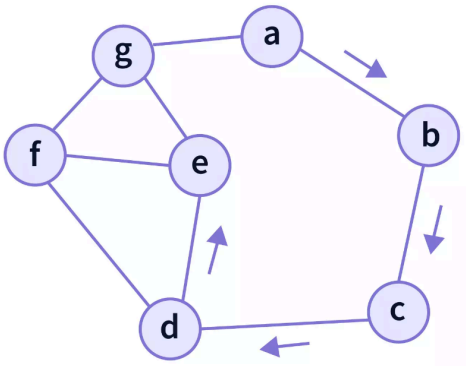
```
function Dijkstra(Graph, source):
  for each vertex v in Graph:
    distance[v] = infinity

 distance[source] = 0
 G = the set of all nodes of the Graph

 while G is non-empty:
    Q = node in G with the least dist[ ]
    mark Q visited
    for each neighbor N of Q:
       alt_dist = distance[Q] + dist_between(Q, N)
       if alt-dist < distance[N]
          distance[N] := alt_dist

 return distance[ ]
```

Applications of Dijkstra's Algorithm

- Navigation Apps: Dijkstra's algorithm powers mapping services like Google Maps, offering optimal travel routes between locations.
- Networking: Essential for determining minimum-delay paths in computer networks and telecommunications.
- Game Strategies: Utilized in game-playing algorithms to make strategic moves and reach specific objectives efficiently.
- Route Optimization: Key in logistics for optimizing delivery routes, public transport systems, and supply chain management.

Graph Terminology
Let us now see various terminologies associated with a graph data structure.

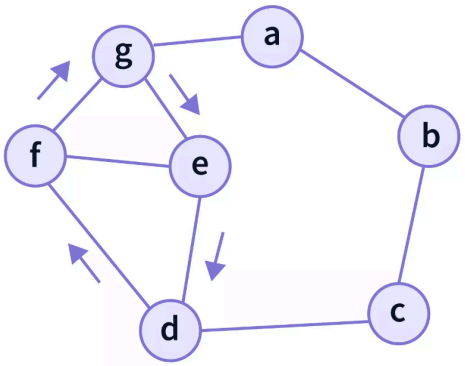
Path
A path in a graph is a finite or infinite set of edges which joins a set of vertices. It can connect to 2 or more nodes. Also, if the path connects all the nodes of a graph in Data Structure, then it is a connected graph, otherwise it is called a disconnected graph.
There may or may not be a path to each and every node of the graph. In case, there is no path to any node, then that node becomes an isolated node.



Path in a graph

In the above graph, the path from 'a' to 'e' is = {a,b,c,d,e}

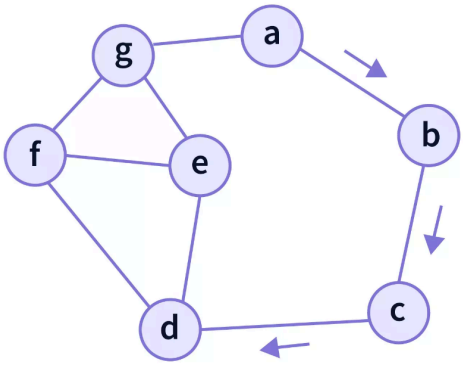
Closed Path
A path is called a closed path if the initial node is the same as the terminal(end) node. A path will be a closed path if : $V_0 = V_n$, where V_0 is the starting node of the graph and V_n is the last node. So, the starting and the terminal nodes are the same in a closed graph.



Closed Path in a graph

The above graph have a closed path, where the initial node = {e} is the same as the final node = {e}. So, the path becomes = {e,d,f,g,e}.

Simple Path
A path that does not repeat any nodes(vertices) is called a simple path. A simple path in a graph exists if all the nodes of the graph are distinct, expect for the first and the last vertex, i.e. $V_0 = V_n$, where V_0 is the starting node of the graph and V_n is the last node.



Simple Path in a graph

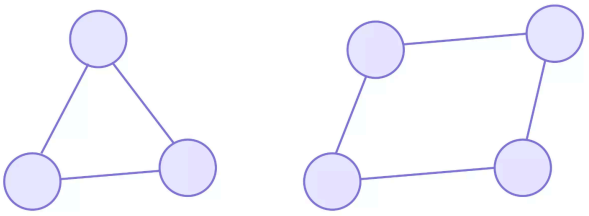
In this example, a,b,c,d is a simple path. Because, this graph does not have any loop or cycle and none of the paths point to themselves.

Degree of a Node

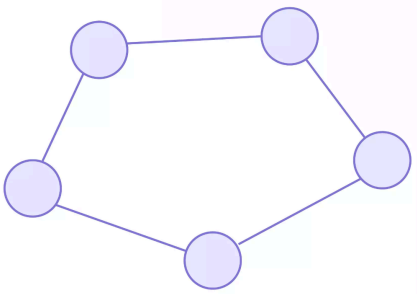
Degree of a node is the number of edges connecting the node in the graph. A simple example would be, suppose in facebook, if you have 100 friends then the node that represents you has a degree of 100.

Cycle Graph

A simple graph of 'n' nodes(vertices) (n>=3) and n edges forming a cycle of length 'n' is called a cycle graph. In a cycle graph, all the vertices are of degree 2.



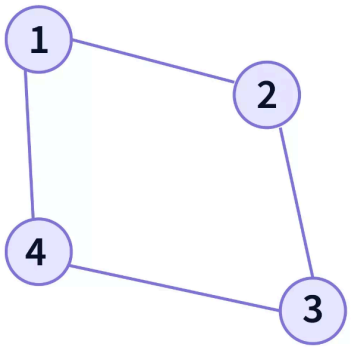
Cycle Graphs



In the above graphs,
Each vertex has degree 2. Therefore, they are cycle graphs.

Connected Graph

Connected graph is a graph in which there is an edge or path joining each pair of vertices. So, in a connected graph, it is possible to get from one vertex to any other vertex in the graph through a series of edges.



Connected Graph

In this graph,

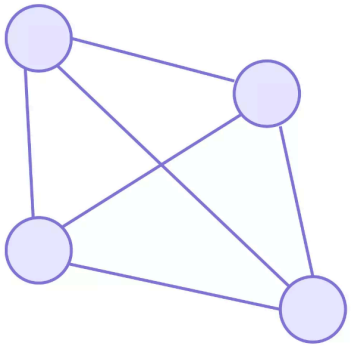
- we can visit from any one vertex to any other vertex.
- There exists at least one path between every pair of vertices.
- There is not a single vertex in a connected graph, which is unreachable(or isolated).
- Therefore, it is a connected graph.

Complete Graph

In a complete graph, there is an edge between every single pair of nodes in the graph. Here, every vertex has an edge to all other vertices. It is also known as a full graph.

A graph in which exactly one edge is present between every pair of vertices is called a complete graph.

- A complete graph of 'n' vertices contains exactly nC_2 edges.
- A complete graph of 'n' vertices is represented as K_n



Complete Graph

In the above graph,

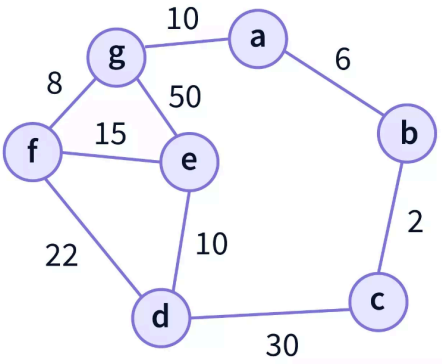
- All the pairs of nodes are connected by each other through an edge.
- Every complete graph is a connected graph, however, vice versa is not necessary.

Note:

In a Complete graph, the degree of every node is n-1, where, n = number of nodes.

Weighted Graph

In weighted graphs, each edge has a value associated with them (called weight). It refers to a simple graph that has weighted edges. The weights are usually used to compute the shortest path in the graph.



Weighted graph

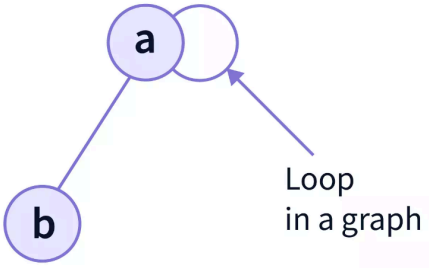
The above graph is a weighted graph, where each edge is associated with a weight. The weights may represent for example, any distance, or time, or the number of connections shared between two users in a social network.

It is not mandatory in a weighted graph that all nodes have distinct weight, i.e. some edges may have same weights.

Loop

A loop (also called a self-loop) is an edge that connects a vertex to itself. It is commonly defined as an edge with both ends as the same vertex.

Although all loops are cycles, not all cycles are loops. Because, cycles do not repeat edges or vertices except for the starting and ending vertex.

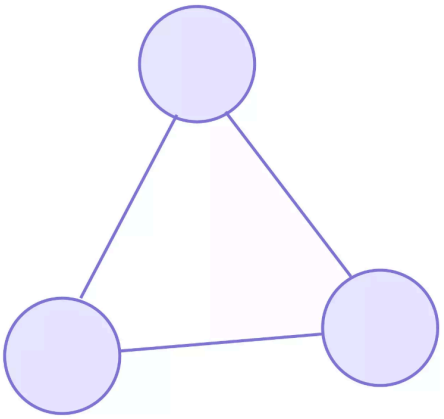


In the above graph,

- The node 'a' has a loop in itself.
- The starting and ending point of the edge in node 'a' is the same.

Hence, there is a loop in the graph.

Simple Graph
A graph having no self loops and no parallel edges in it is called a simple graph.



Here,

- This graph consists of three vertices and three edges.
- There are neither self loops nor parallel edges.
- Therefore, it is a simple graph.

.

Practical Applications of Graph

- GPS systems and Google Maps use graphs to find the shortest path from one destination to another.
- The Google Search algorithm uses graphs to determine the relevance of search results.
- The World Wide Web is the biggest graph. All the links and hyperlinks are the nodes and their interconnection is the edges. This is why we can open one webpage from the other.
- Social Networks like facebook, twitter, etc. use graphs to represent connections between users.
- The nodes we represent in our graphs can be considered as the buildings, people, groups, landmarks or anything in general , whereas the edges are the paths connecting them.

Exercise

1. Write a program to implement an undirected graph with the following.

    - ➔ Create an adjacency matrix.
    - ➔ Create an adjacency List.
    - ➔ Print the information of the graph such as number of edges, edges list, degree of each vertex. (using both matrix and list)
    - ➔ implement traversal of graphs using DFS (using both matrix and list)
    - ➔ implement traversal of graphs using BFS. (using both matrix and list)

Note: Include output for 2 graphs (having more than 7 vertices) in each program. Also show a visual graph in your submission.

---

a. Write a program to implement a directed graph with the following.

- ➔ Create an adjacency matrix.
- ➔ Create an adjacency List.
- ➔ Print the information of the graph such as number of edges, edges list, degree of each vertex. (using both matrix and list)
- ➔ implement traversal of graphs using DFS (using both matrix and list)
- ➔ implement traversal of graphs using BFS. (using both matrix and list)

Note: Include output for 2 graphs (having more than 7 vertices) in each program. Also show a visual graph in your submission.