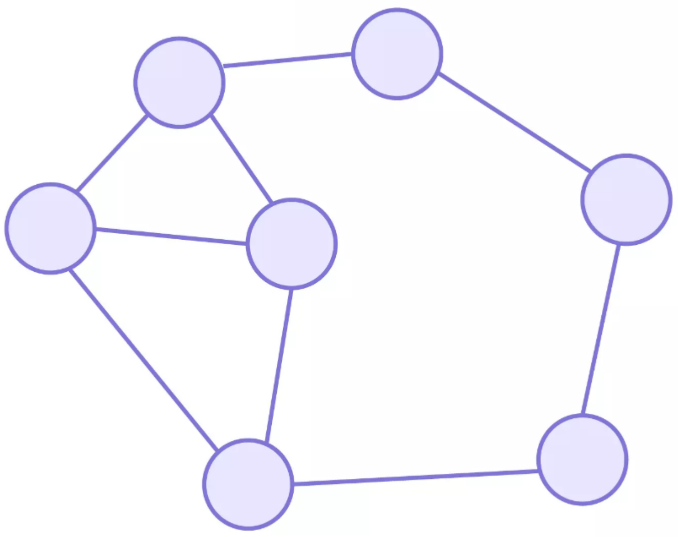
Practical–8
Implementation of Graph

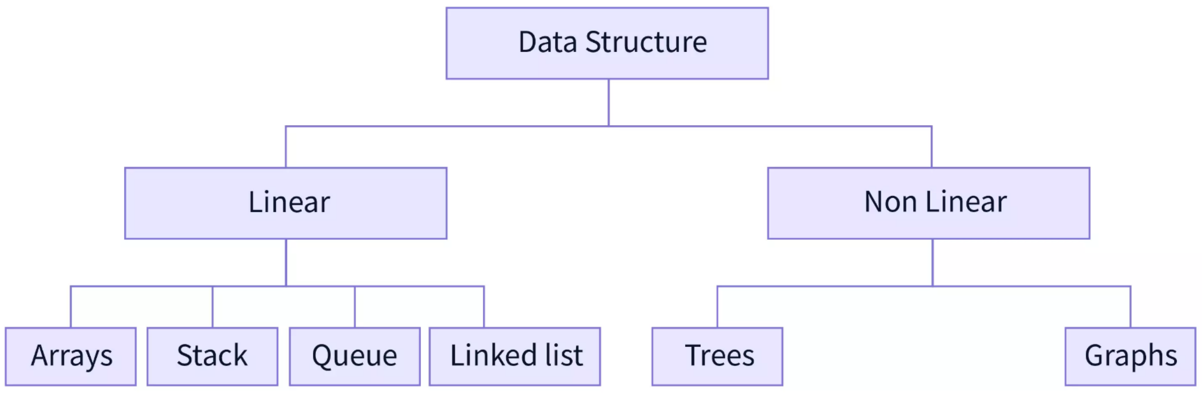
"A Graph is a non-linear data structure that consists of nodes and edges which connects them".

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



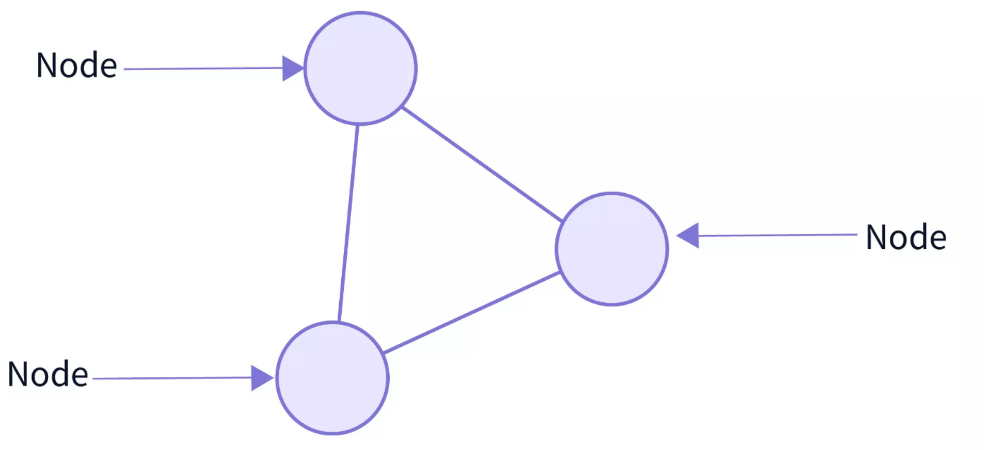
Non-linear Data Structure

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:

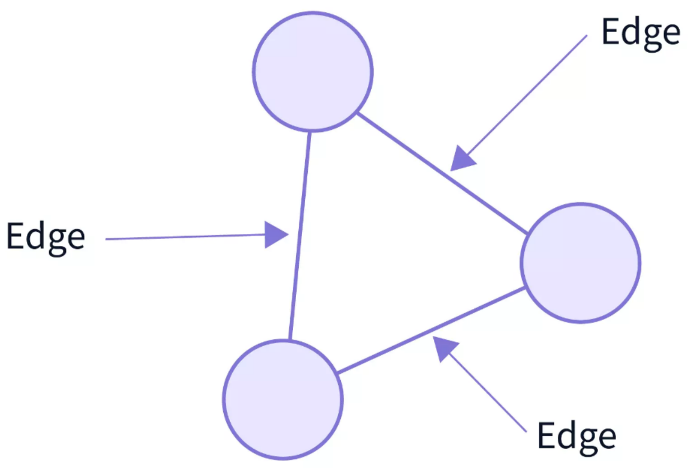
- Nodes create complete networks in any graph.
- They are one of the building blocks of a graph data structure.
- They connect the edges and create the main network of a graph.
- They are also called vertices.
- A node can represent anything such as any location, port, houses, buildings, landmarks, etc.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



This graph has 3 nodes

Edges:

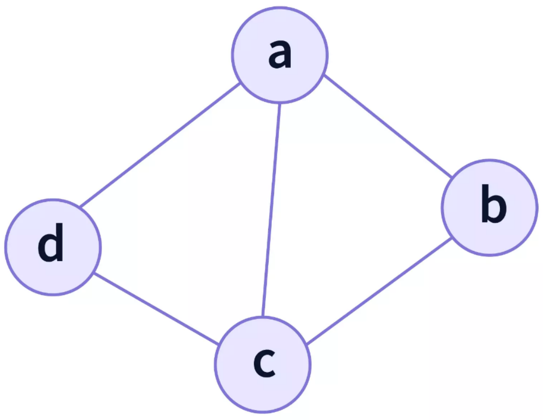
Edges basically connect the nodes in a graph data structure. They represent the relationships between various nodes in a graph. Edges are also called the path in a graph.



This graph has 3 edges

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:

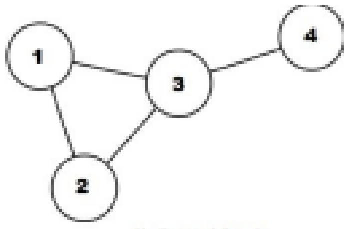
```
V = {a, b, c, d}

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).

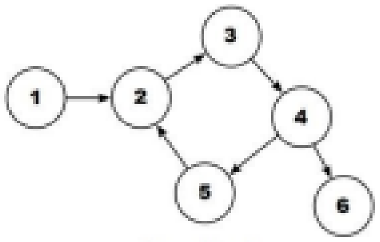
Types of Graphs in Data Structure

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.



Undirected Graph



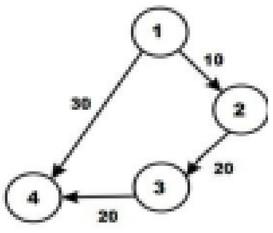
Directed Graph

2. Directed: A graph in which all the edges are uni-directional. 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



Weighted Graph

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

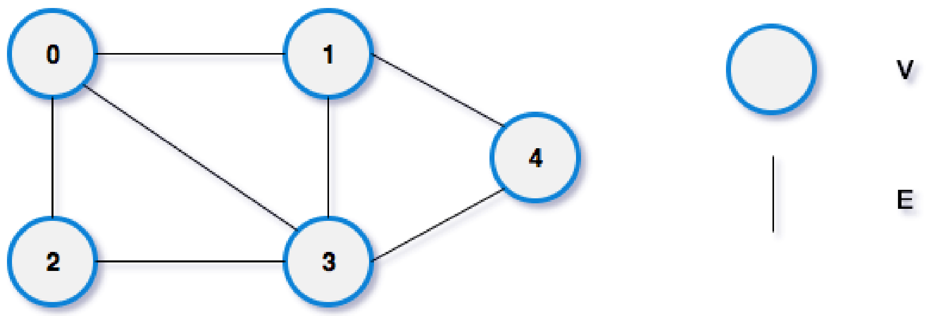
An Adjacency Matrix is the simplest way to represent a graph. It is a 2D array of V x V vertices with each row and column representing a vertex. The matrix is consists of "0" or "1". 0 depicts that there is no path while 1 represents that there is a path.

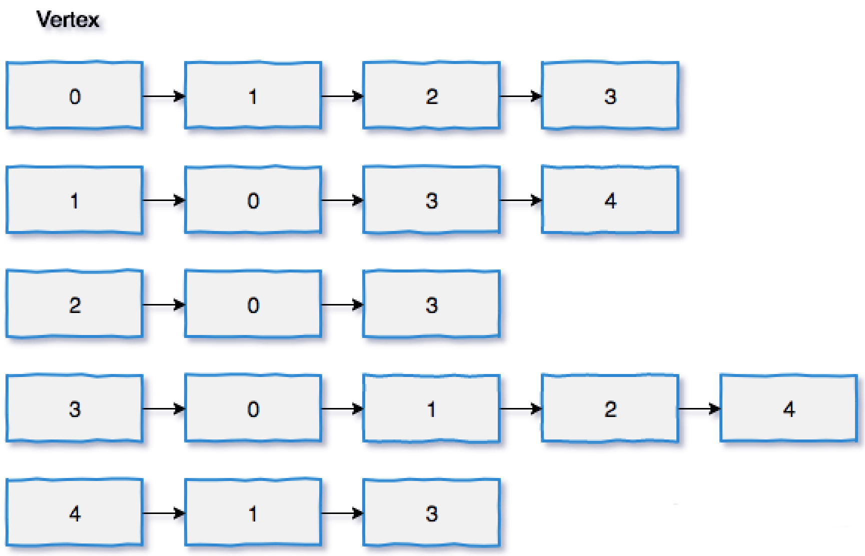
Operations on Graph in Data Structure

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 List





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
```

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;
}
```

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;
}
```

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 graph using DFS (using both matrix and list)
   - ➔ implement traversal of graph using BFS. (using both matrix and list)

2. 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 graph using DFS (using both matrix and list)
   - ➔ implement traversal of graph using BFS. (using both matrix and list)

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