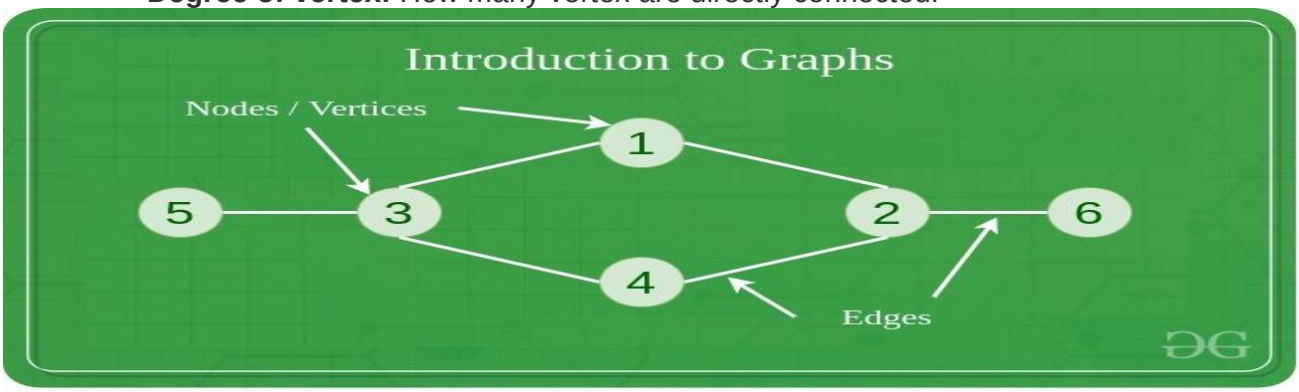
Graph is collection of set of vertices (denoted as **V**) and set of edges (denoted as **E**). *The graph is denoted by G (E, V).*

Components of a Graph:

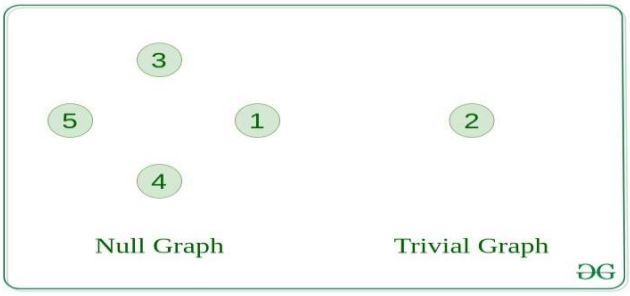
1. **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabeled.
2. **Edges:** Edges are the drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabeled.

   Adjacent Vertices: If we have direct edge from one node/vertex. to other node/vertex., will be adjacent vertices of that node/vertex.
   Ex: Node 1 ->Direct edge (2 and 3) so they are adjacent vertices.
   Degree of vertex: How many vertex are directly connected.



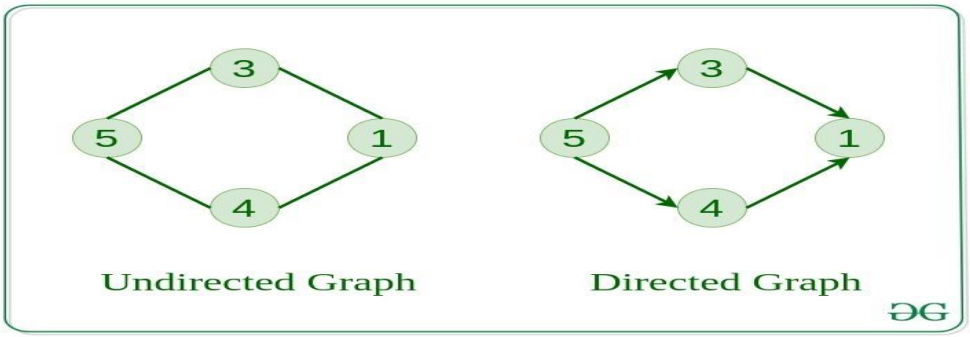
Types of Graph

Null Graph: A graph is known as null graph if there are no edges in the graph.



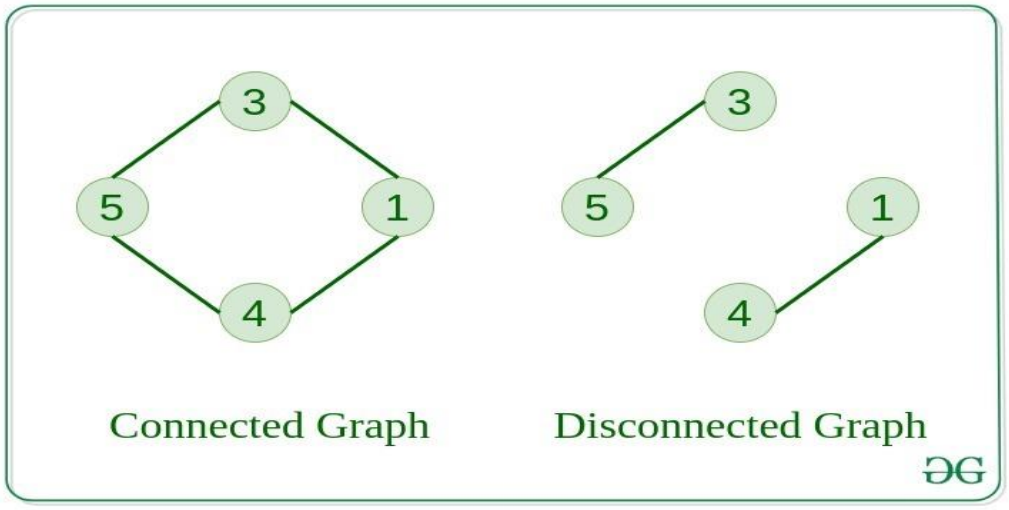
Trivial Graph: Graph having only a single vertex, it is the smallest graph possible.

Undirected Graph: A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.



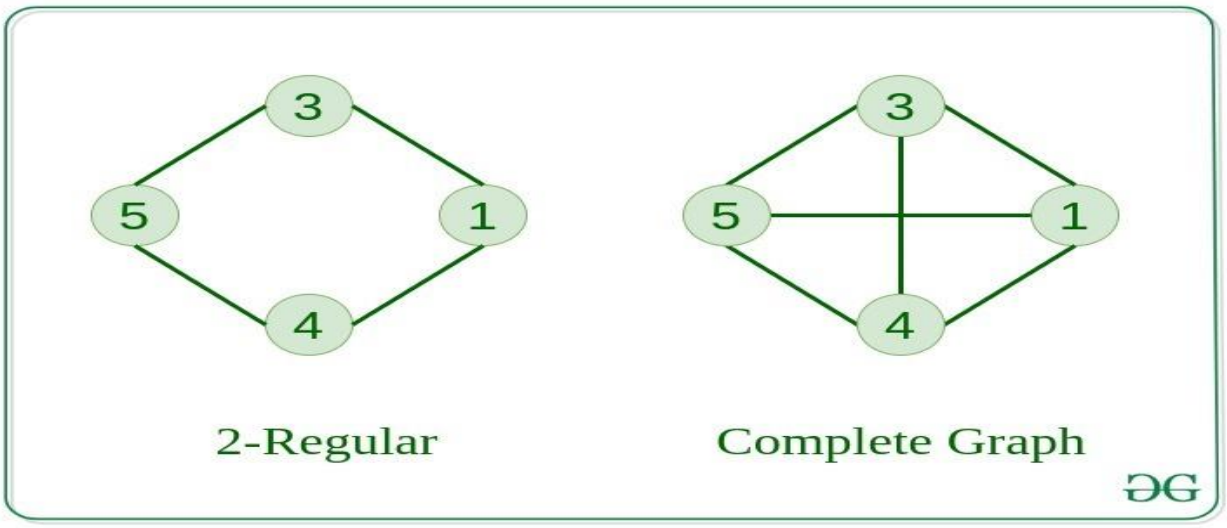
Directed Graph: A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

Connected Graph: The graph in which from one node we can visit any other node in the graph is known as a connected graph.



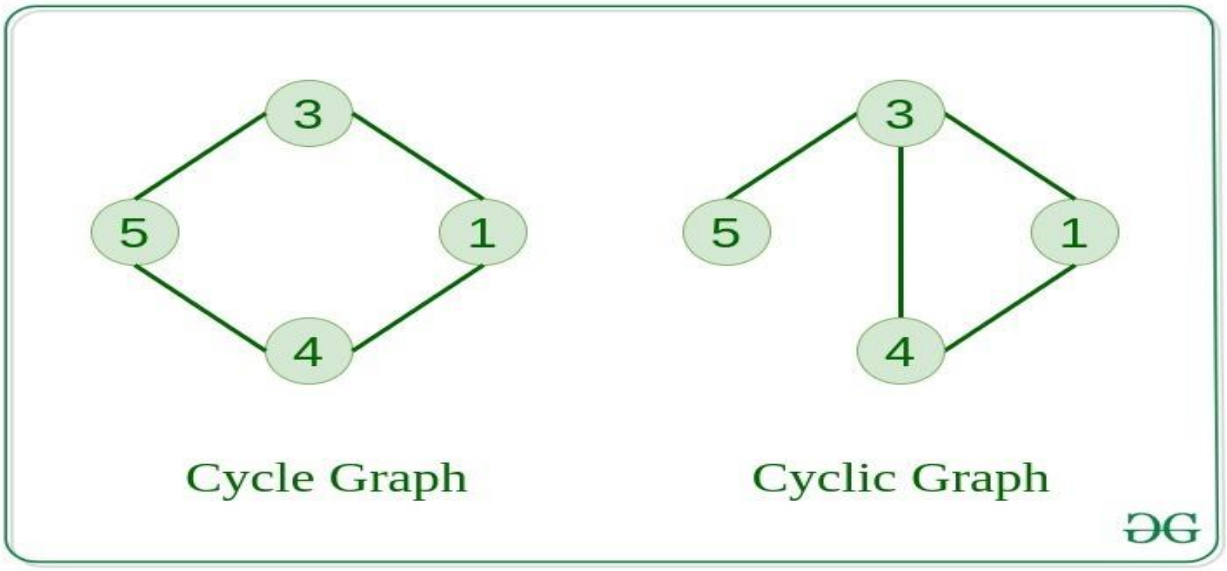
Disconnected Graph: The graph in which at least one node is not reachable from a node is known as a disconnected graph.

Regular Graph: The graph in which the degree of every vertex is equal to the other vertices of the graph. Let the degree of each vertex be **K** then the graph is called **K-regular**.



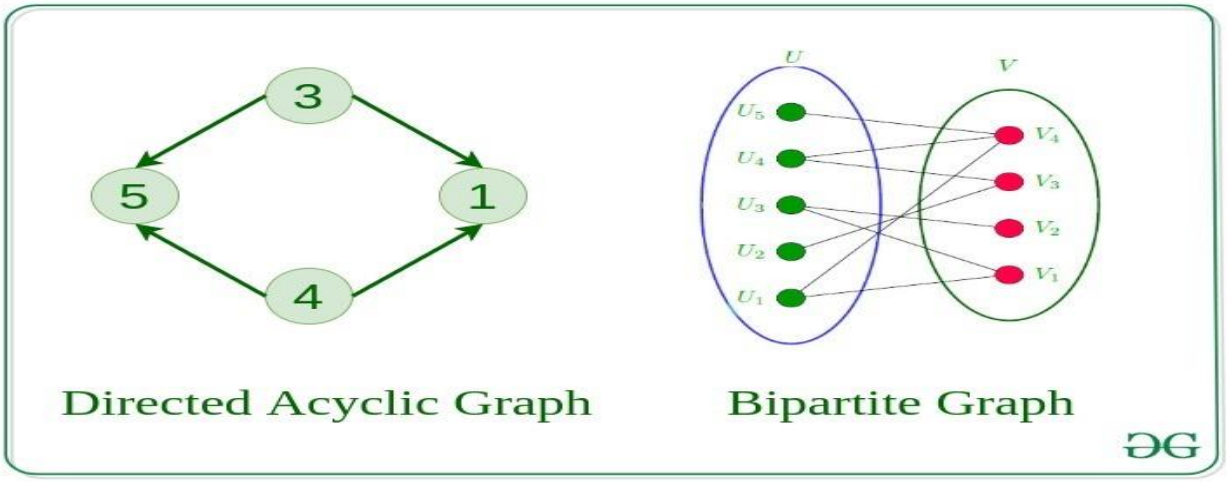
Complete Graph: The graph in which from each node there is an edge to each other node.

Cycle Graph: The graph in which the graph is a cycle, the degree of each vertex is
2



Cyclic Graph: A graph containing at least one cycle is known as a Cyclic graph.

Directed Acyclic Graph: A Directed Graph that does not contain any cycle.



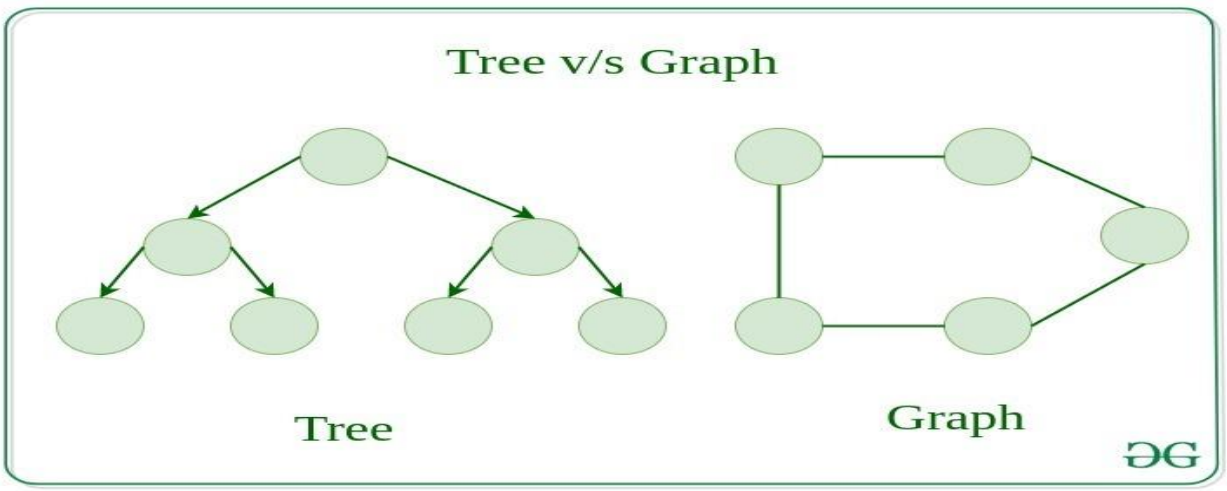
Bipartite Graph

1. A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.

Weighted Graph: A graph in which the edges are already specified with suitable weight is known as weighted graph. Weighted graph can be further classified as directed weighted graph and undirected weighted graph.

Tree v/s Graph

Trees are the restricted types of graphs, just with some more rules. Every tree will always be a graph but not all graphs will be trees. Linked List, Trees, and Heaps all are special cases of graphs.

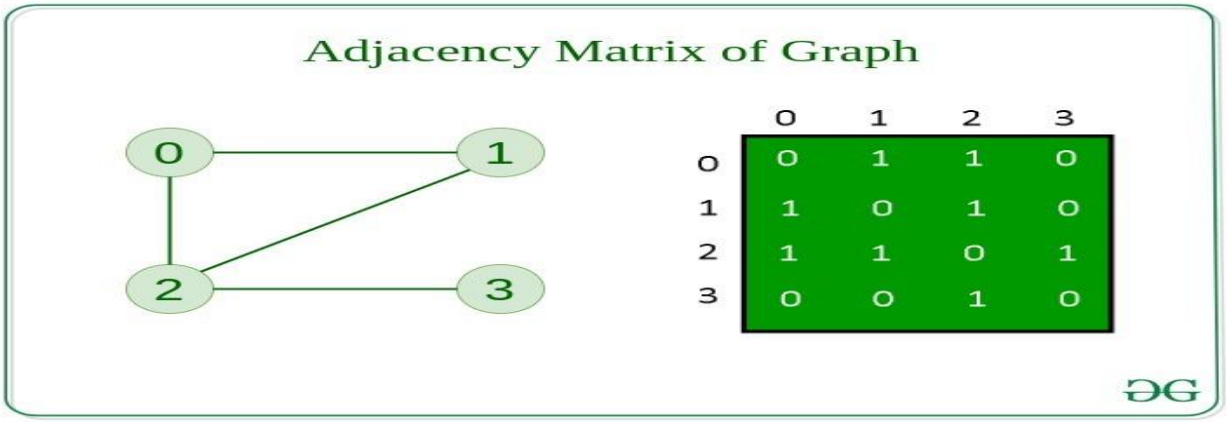


Representation of Graphs

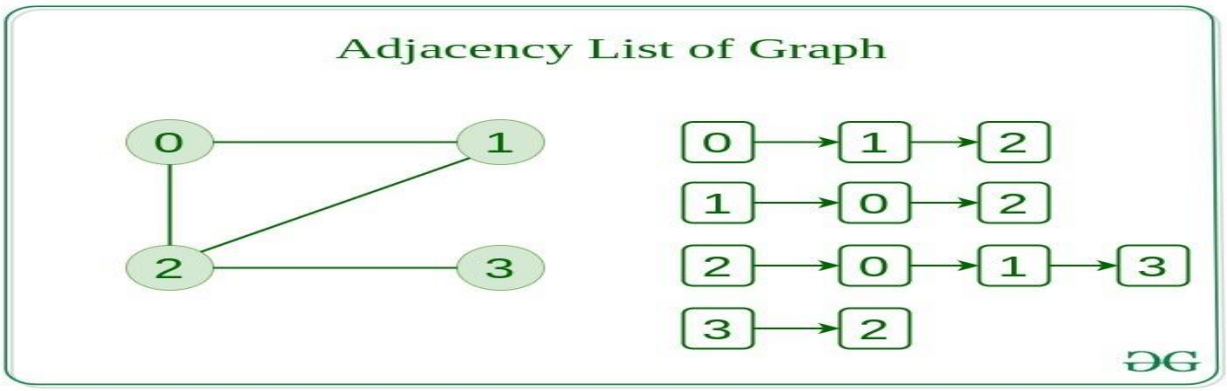
There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List
- Edge list

Adjacency Matrix: In this method, the graph is stored in the form of the 2D matrix where rows and column denote vertices. Each entry in the matrix represents the weight of the edge between vertices.



Adjacency List: This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Comparison between them

When the graph contains many edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing and edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Basic Operations on Graphs

Below are the basic operations on the graph:

- Insertion of Nodes/Edges in graph – Insert a node into the graph.
- Deletion of Nodes/Edges in graph – Delete a node from graph.
- Searching on Graphs – Search an entity in graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

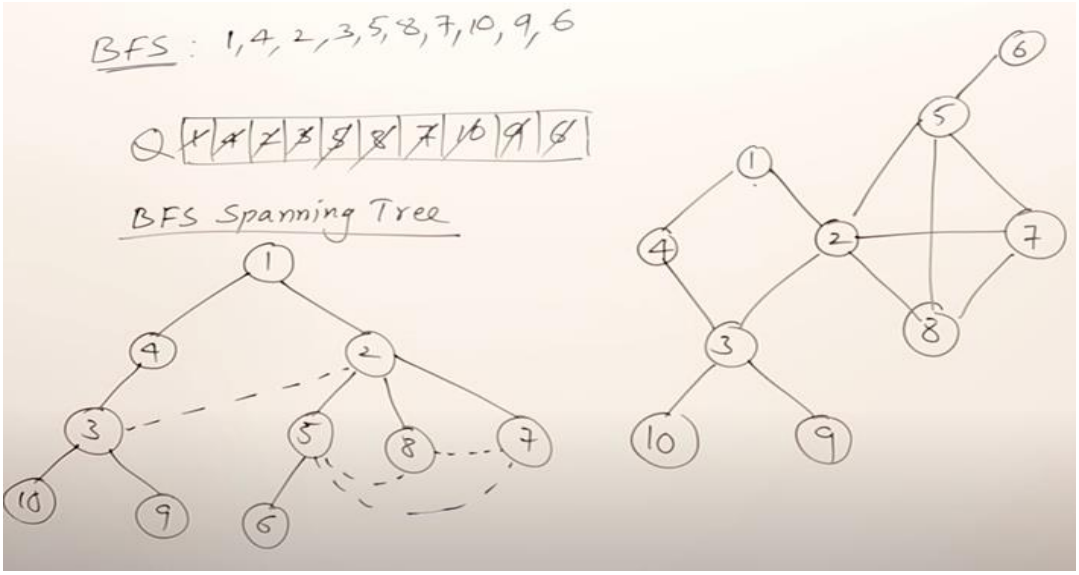
Usage of graphs

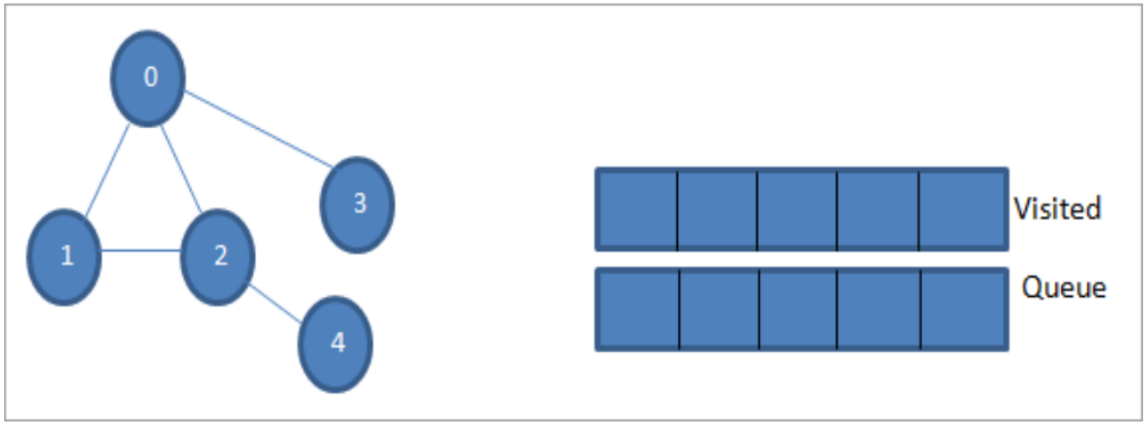
- Maps can be represented using graph and this then can be used by computers to provide various services like the shortest path between two cities.
- When various tasks depend on each other then this situation can be represented using a Directed Acyclic graph and we can find the order in which task can be performed using topological sort.
- State Transition Diagram represents what can be the legal moves from current states. In-game of tic tac toe this can be used.

Graph Traversal:

1) **BFS (Breadth First Search):**

- **Step 1:** Start with node S and enqueue it to the queue.
- **Step 2:** Repeat the following steps for all the nodes in the graph.
- **Step 3:** Dequeue S and process it.
- **Step 4:** Enqueue all the adjacent nodes of S and process them.
- [END OF LOOP]
- **Step 6:** EXIT






```
#include<iostream>
#include <list>
#include <queue>
using namespace std;

// a directed graph class
class Graph
{
   int V;    // No. of vertices

   // Pointer to an array containing adjacency lists
   list<int>* adjList;
          bool *visited;
public:
   Graph(int V);  // Constructor

   // add an edge from vertex v to w
   void addEdge(int v, int w);

   // BFS traversal sequence starting with s
   void BFS(int s);
};

Graph::Graph(int V)
{
   this->V = V;
   adjList = new list<int>[V];
          visited = new bool[V]{0};
}
 void Graph::addEdge(int u, int v)
{
   adjList[u].push_back(v); // Add v to u's list.
```

```
}

void Graph::BFS(int s)
{
   // queue to hold BFS traversal sequence
   queue<int> q;

   // Mark the current node as visited and enqueue it
   visited[s] = true;
   q.push(s);

   while(!q.empty())
   {
      // dequeue the vertex
      s = q.front();
      cout<<"node value:"<<s<<"\n";
      q.pop();

      // get all adjacent vertices of popped vertex and process each if not already visited
      for (auto i= adjList[s].begin(); i != adjList[s].end(); ++i)
      {
         if (!visited[*i])
         {
            visited[*i] = true;
            q.push(*i);
         }
      }
   }
}
// main program
int main()
{
   // create a graph
   Graph g(5);
   g.addEdge(0, 1);
   g.addEdge(0, 2);
   g.addEdge(0, 3);
   g.addEdge(1, 2);
   g.addEdge(2, 4);
   g.addEdge(3, 3);
   g.addEdge(4, 4);

   cout<<"Breadth First Traversal for given graph (with 0 as starting node): "<<endl;
   g.BFS(0);
```

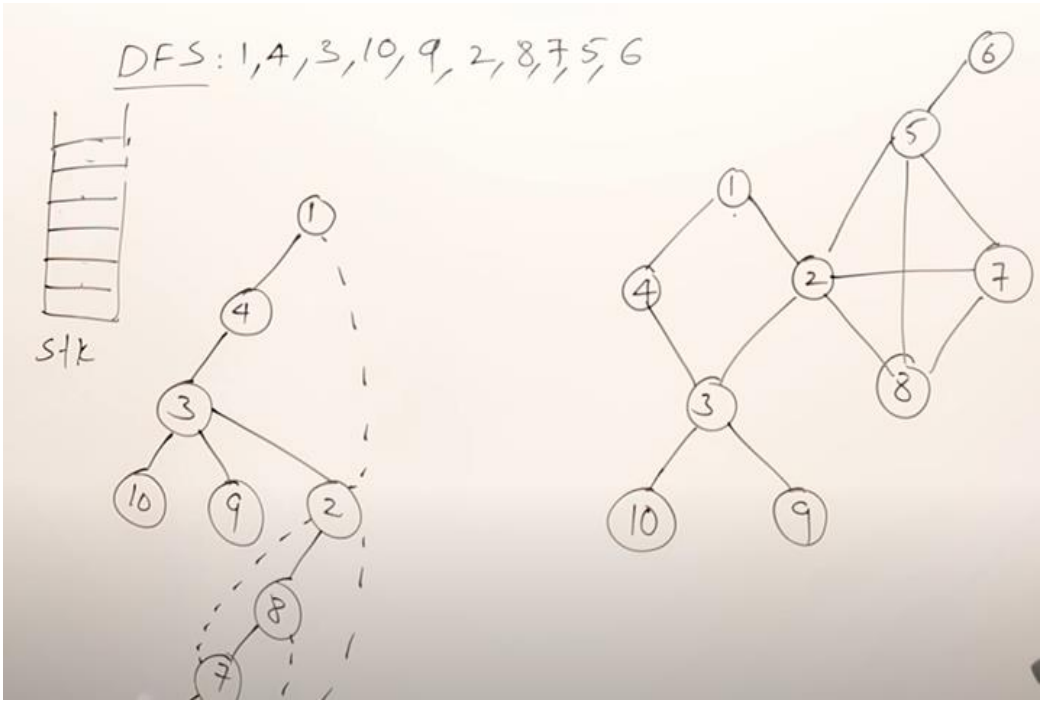
```
    return 0;
}
```

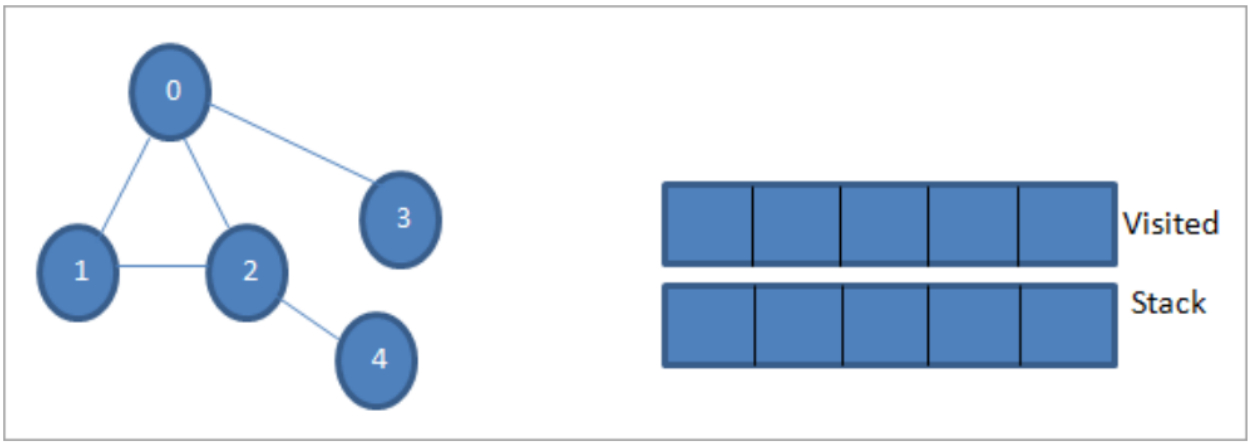
O/P: 0,1,2,3,4

---

2) DFS (Depth First Search):

- **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
- **Step 2:** Pop the top item from the stack and add it to the visited list.
- **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited to the stack.
- **Step 4**: Repeat steps 2 and 3 until the stack is empty.






```
#include <iostream>
#include <list>
using namespace std;
class Graph
{
        int numVertices;
        list<int> *adjLists;
        bool *visited;
 public:
        Graph(int V);
        void addEdge(int u, int v);
        void dfs(int vertex);
};
Graph::Graph(int vertices)
{
        numVertices = vertices;
        adjLists = new list<int>[vertices];
        visited = new bool[vertices]{0};
}
void Graph::addEdge(int u, int v)
```

```
{
	adjLists[u].push_front(v);
}
void Graph::dfs(int vertex)
{
	visited[vertex] = true;
	cout << vertex << " ";
	for (auto i = adjLists[vertex].begin(); i != adjLists[vertex].end(); ++i)
	{
		if (!visited[*i])
		{
			dfs(*i);
		}
	}
}


int main()
{
  // create a graph
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(2, 4);
    g.addEdge(3, 3);
    g.addEdge(4, 4);
  cout<<"Depth First Search from vertex 2 is :"<<endl;
```

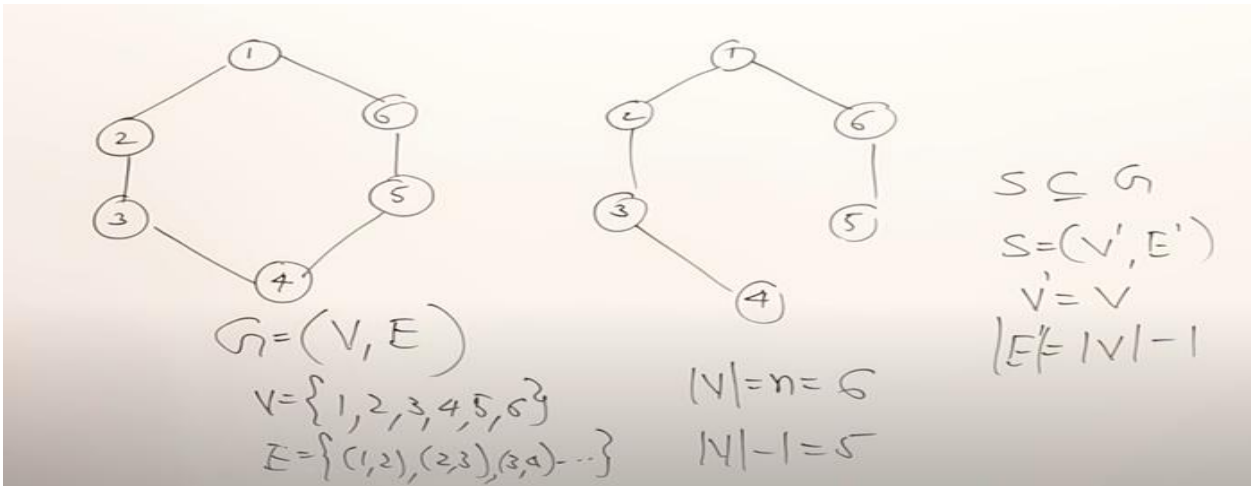
```
 g.dfs(0);


 return 0;

}
```

O/P: 0,3,2,4,1

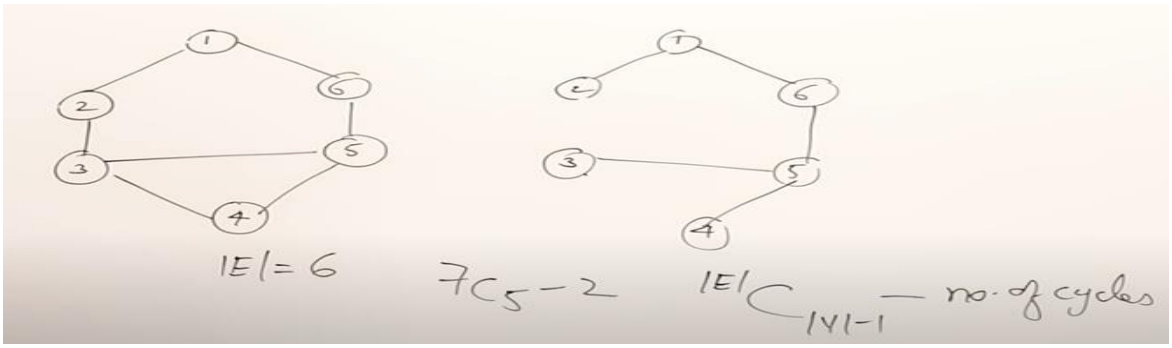
Spanning Tree: Spanning tree is sub graph of graph, which has all vertices, but edges will be no. of vertices-1(n-1). We can find minimum spanning tree only for connected graph.



How many spanning trees are possible from above graph??

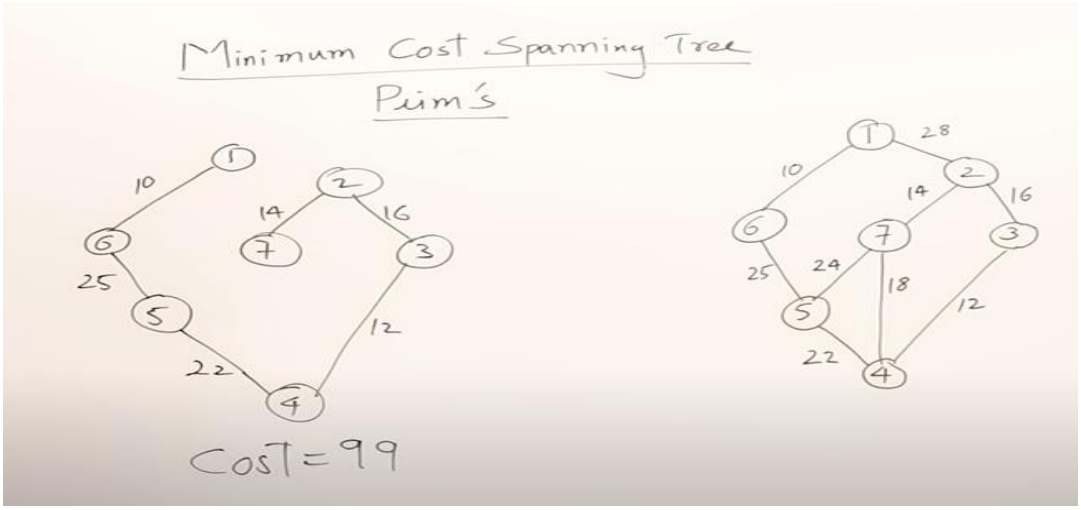
We can select **n-1** vertices, so **6C5** => **nCr = n! / r! * (n - r)! = 6**

Formula to calculate no. of spanning tree:



Minimum cost spanning tree: If weighted graph is given then what are the ways of finding minimum spanning tree:

1) **Prim's algo(Greedy approach):** Select the smallest edge and then select connected smallest edge.



3) Kruskal's algo (Greedy approach):

Select smallest edges but do not select edge if it's forming cycle.