303105201 - Design of Data Structures

UNIT-8 GRAPHS

Definitions

Terminologies

Matrix and Adjacency List Representation of Graphs

Elementary Graph operations

Traversal methods: Breadth First Search and Depth First Search.

Graph

Graphs are fundamental data structures in computer science used to represent relationships between objects.

Definition:

A collection of nodes (or vertices) connected by edges.A graph can be defined as group of vertices and edges that are used to connect these vertices.

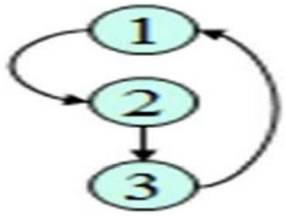
A Graph, G = (V, E), consists of two sets V and E

–V: finite non-empty set of vertices

–E: set of pairs of vertices, edges, e.g.(1,2) or <1,2>

Note: V(G): set of vertices of graph G

E(G): set of edges of graph G



$V(G) = \{1, 2, 3\}$
$E(G) = \{\langle 1,2\rangle \langle 2,3\rangle \langle 3,1\rangle\}$

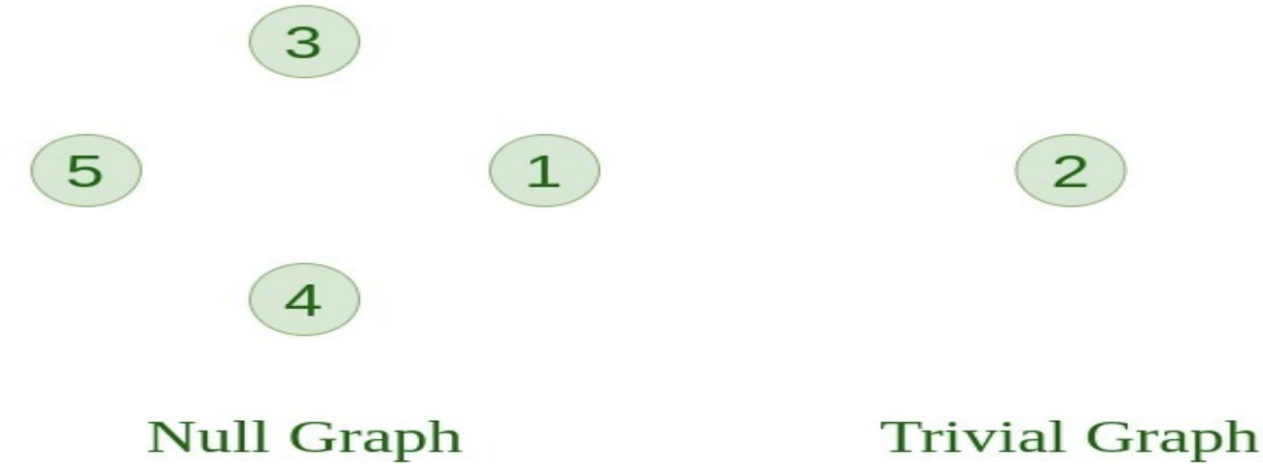
Types of Graphs:

Null Graph:

A graph is known as a null graph if there are no edges in the graph.

Trivial Graph:

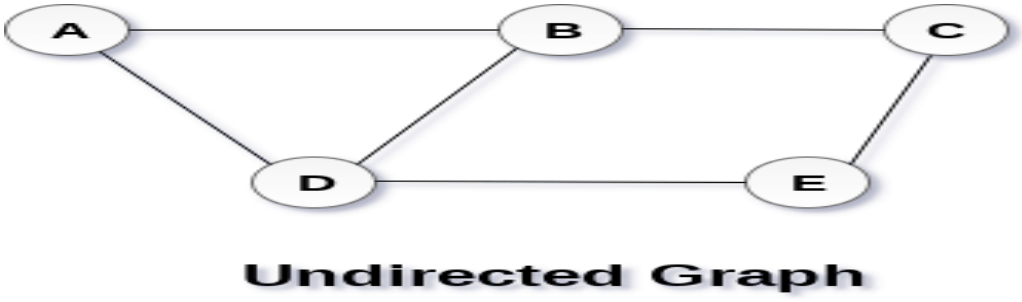
Graph having only a single vertex, it is also the smallest graph possible.



Undirected graph:

A graph where edges have no direction.

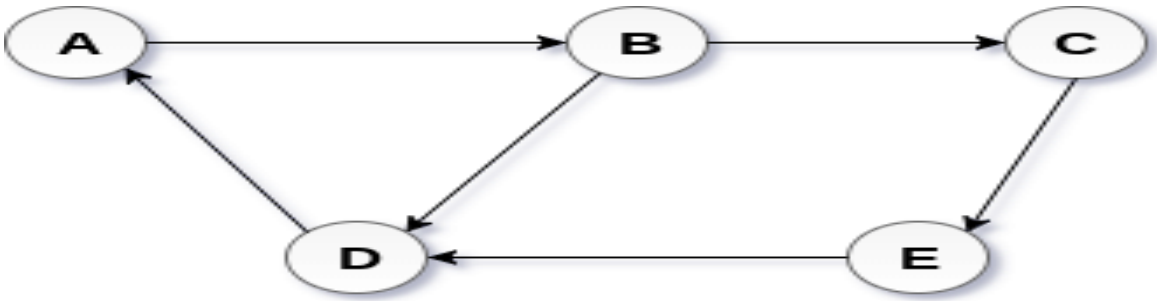
A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them.
A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.



Directed graph:

A graph where edges have a direction (from one vertex to another).

In a directed graph, edges form an ordered pair. A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



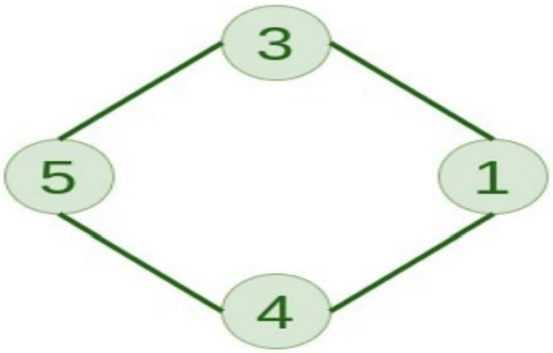
Directed Graph

Connected Graph:

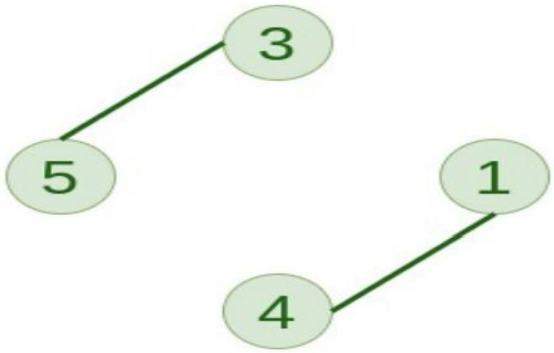
The graph in which from one node we can visit any other node in the graph is known as a connected graph.

Disconnected Graph:

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



Connected Graph



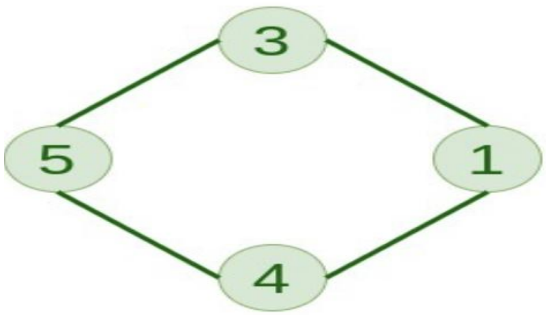
Disconnected Graph

Regular Graph:

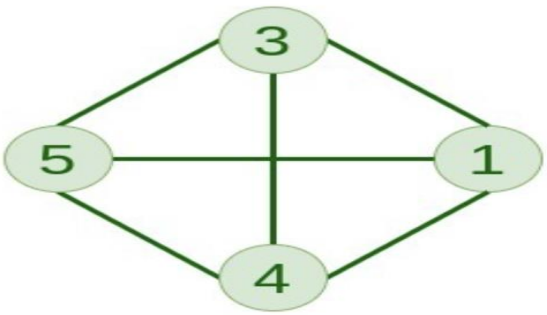
The graph in which the degree of every vertex is equal to K is called K regular graph.

Complete Graph:

The graph in which from each node there is an edge to each other node. A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.



2-Regular



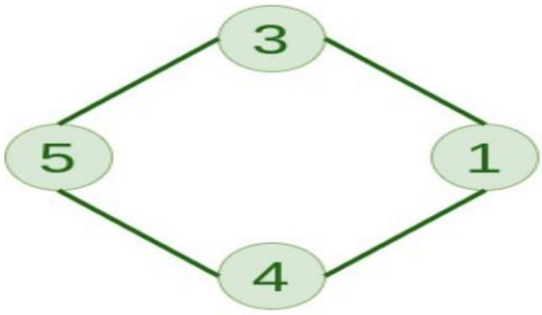
Complete Graph

Cycle Graph:

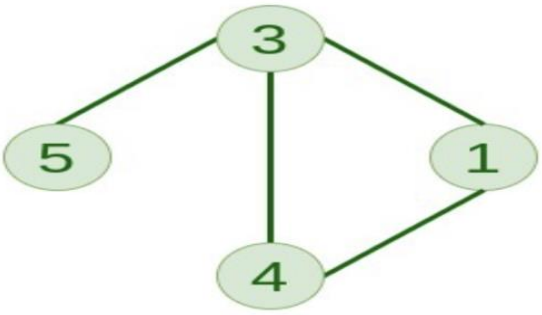
The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.

Cyclic Graph:

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



Cycle Graph



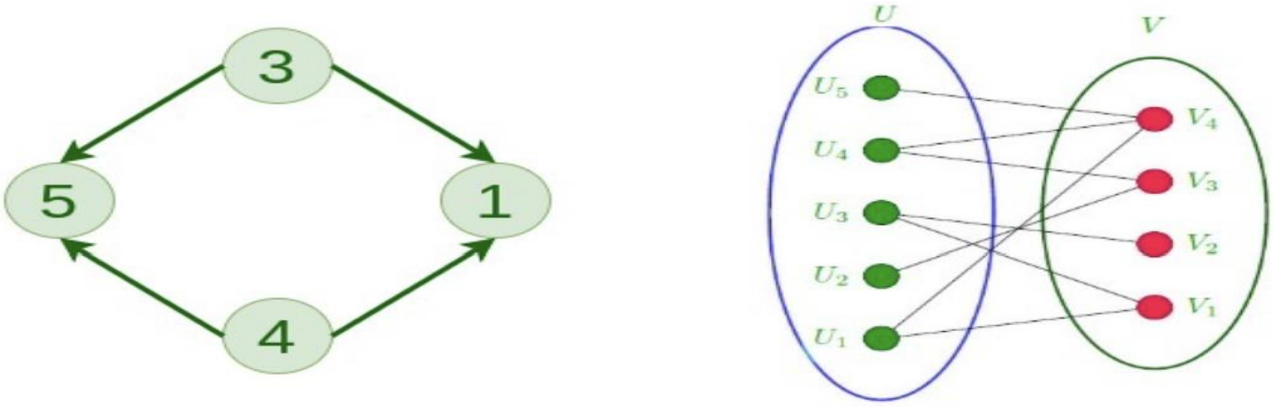
Cyclic Graph

Directed Acyclic Graph:

A Directed Graph that does not contain any cycle.

Bipartite Graph:

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



Directed Acyclic Graph Bipartite Graph

Terminologies:

Path:

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Edge:

A connection between two vertices.

Closed Path:

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Simple Path:

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.

Weighted Graph:

A graph where edges have weights (costs).

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph:

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop:

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes:

Two vertices are adjacent if they are connected by an edge.

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node:

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph representation

A graph is a data structure that consist a set of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- Sequential representation (or, Adjacency matrix representation)
- Linked list representation (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

Sequential representation:(Adjacency matrix)

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

// Example of an adjacency matrix for a graph with 4 vertices

int adjMatrix[4][4] = {

{0, 1, 0, 1},

{1, 0, 1, 0},

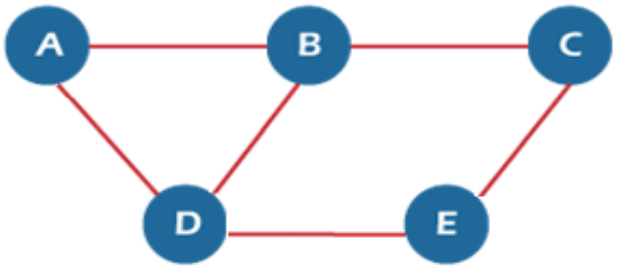
{0, 1, 0, 1},

{1, 0, 1, 0}

};

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.



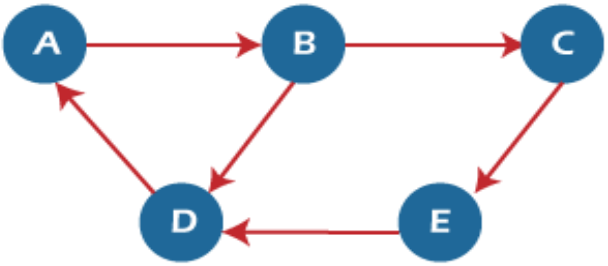
Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex.



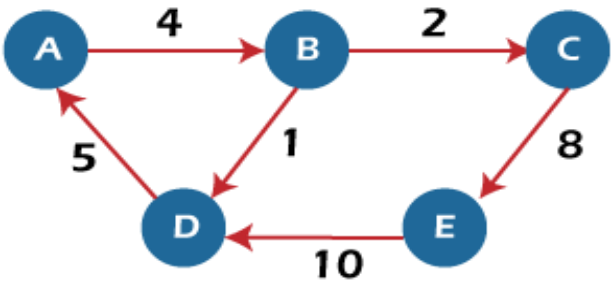
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path. The weights on the graph edges will be represented as the entries of the adjacency matrix.



weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

Linked list representation:(Adjacency List)

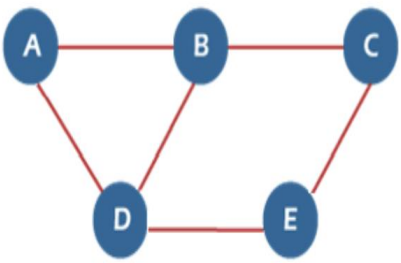
In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency

matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

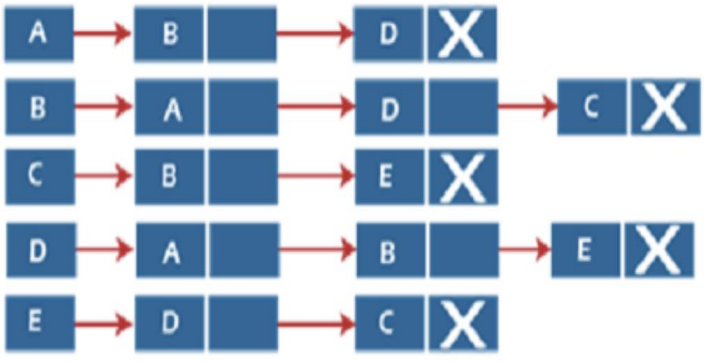
It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its

adjacencies. Example:

An adjacency list uses an array of lists to represent a graph. Each index in the array represents a vertex, and each element in the list at that index represents the vertices that are adjacent to it.

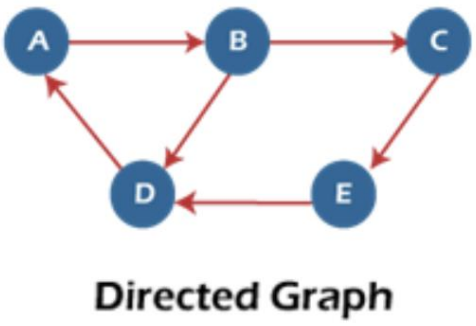


Undirected Graph

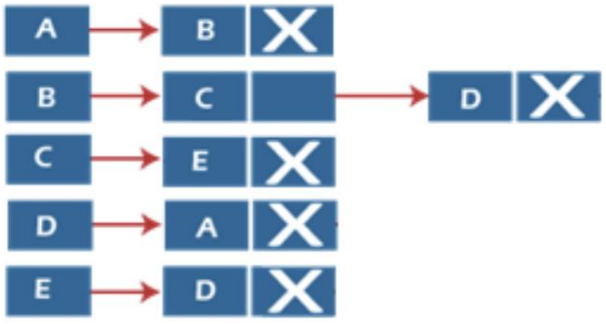


Adjacency List

```
// Example of an adjacency list for a graph with 4 vertices
struct Node {
int vertex;
struct Node* next;
};
struct Graph {
int numVertices;
struct Node** adjLists;
};
// Function to create a graph
struct Graph* createGraph(int vertices) {
struct Graph* graph = malloc(sizeof(struct Graph));
graph->numVertices = vertices;
graph->adjLists = malloc(vertices * sizeof(struct Node*));
for (int i = 0; i < vertices; i++)
graph->adjLists[i] = NULL;
return graph;
}
```



Directed Graph



Adjacency List

Elementary Graph operations

Elementary graph operations are fundamental actions that can be performed on graphs, which are essential structures in data science and computer science.

- Insertion: Adding vertices or edges.

- Deletion: Removing vertices or edges.

- Searching: Finding a vertex or edge.

- Traversal: Visiting all vertices or edges in a specific order.

Graph Representation

Adjacency Matrix: A 2D array where each cell (i, j) indicates the presence or absence of an edge between vertices i and j.

Adjacency List: A list where each vertex has a list of adjacent vertices, typically represented as linked lists or arrays.

Adding a Vertex

To add a vertex, simply extend the adjacency list or matrix to include a new row and column (in the case of the adjacency matrix).

Removing a Vertex

Involves deleting the vertex from the adjacency list and removing the corresponding row and column from the adjacency matrix.

Adding an Edge

Undirected Graph: Update the adjacency lists of both vertices. In the adjacency matrix, set the cells (i, j) and (j, i) to indicate an edge.

Directed Graph: Update the adjacency list for the source vertex and set the cell (i, j) in the adjacency matrix.

Removing an Edge

Undirected Graph: Remove the edge from both vertices' lists and set the corresponding cells in the adjacency matrix to 0.

Directed Graph: Remove the edge from the source vertex's list and set the appropriate cell in the adjacency matrix.

Traversal

Depth-First Search (DFS): Explores as far down a branch as possible before backtracking. Can be implemented using recursion or a stack.

Breadth-First Search (BFS): Explores all neighbors at the present depth before moving on to nodes at the next depth level, typically implemented using a queue.

Finding Paths

Shortest Path: Algorithms like Dijkstra's or Bellman-Ford can be used to find the shortest path from a source vertex to all other vertices.

All-Pairs Shortest Path: Floyd-Warshall algorithm or repeated application of Dijkstra's.

Checking Connectivity

Use DFS or BFS from a starting vertex to check if all vertices are reachable.

Finding Cycles

DFS can be modified to detect cycles in directed and undirected graphs.

Graph Properties

Degree of a Vertex: Count of edges incident to a vertex.

Connected Components: Subgraphs in which any two vertices are connected to each other by paths.

Traversal Methods:

Breadth-First Search (BFS)

- Uses a queue to explore the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

- *Algorithm:*

1. Start from a selected node, mark it as visited, and enqueue it.

2. Dequeue a node, process it, and enqueue all its unvisited neighbors.

3. Repeat until the queue is empty.

BFS traversal of a graph produces a Spanning Tree is a graph without loops.

We use Queue Data Structure with maximum size of total number of vertices in the graph to implement BFS traversal

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it onto the Queue.

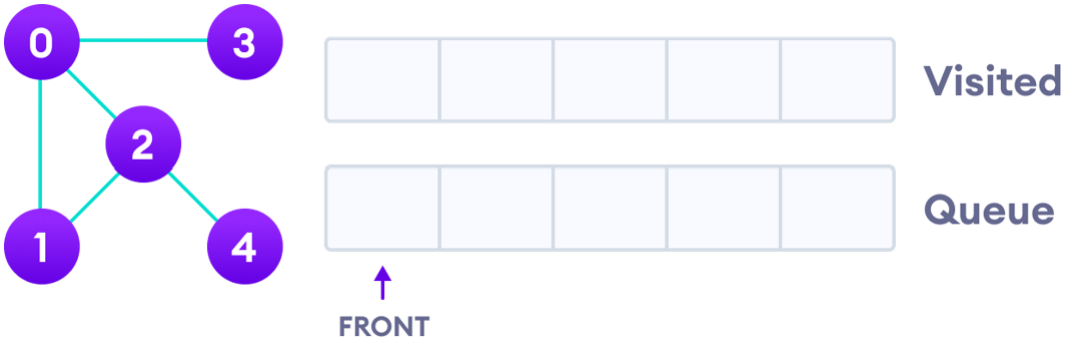
Step 3: Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4: When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

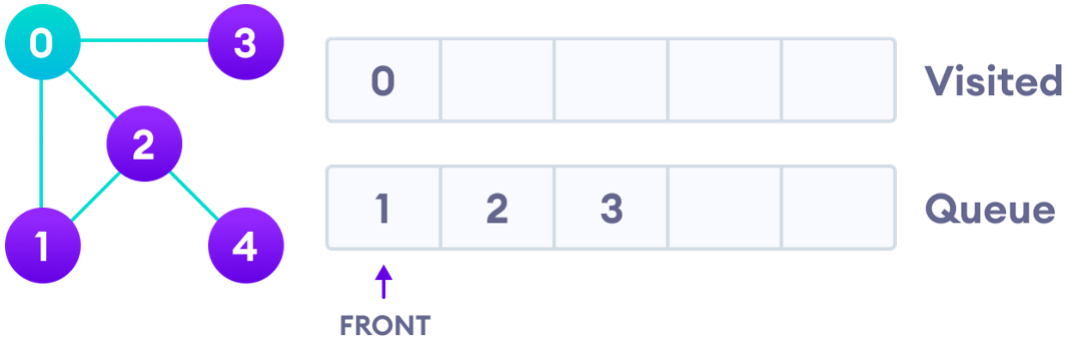
Step 5: Repeat Steps 3 and 4 until queue becomes empty

Step 6: When Queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

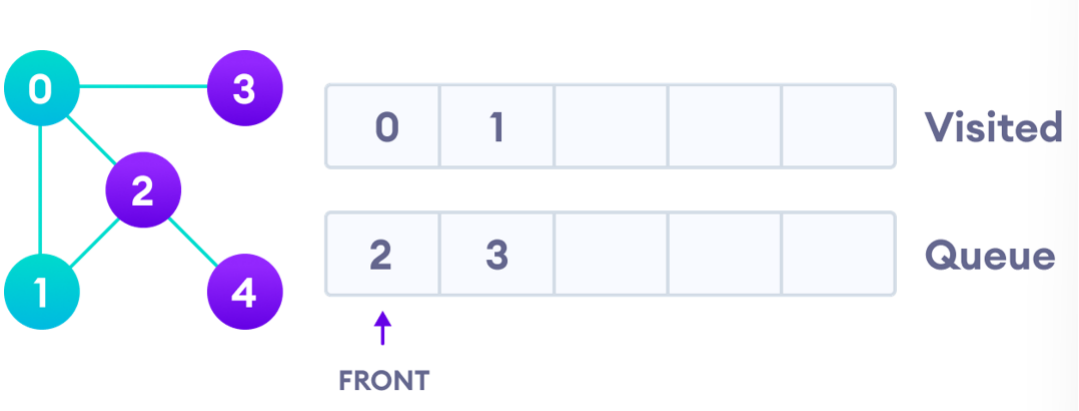
We use an undirected graph with 5 vertices.



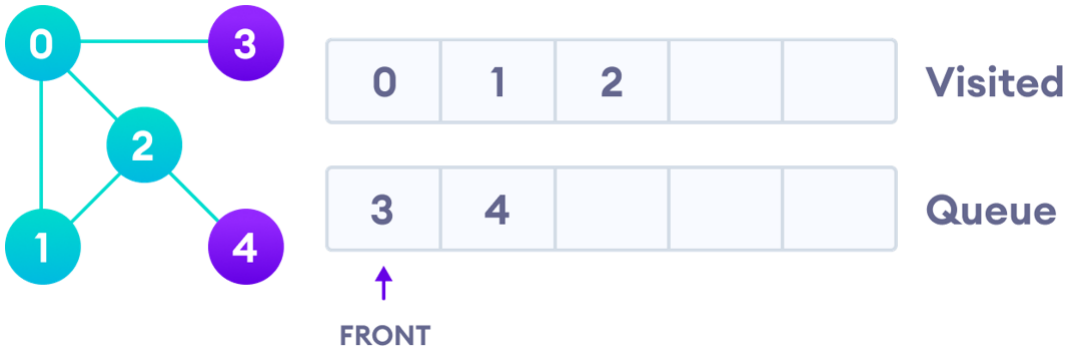
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



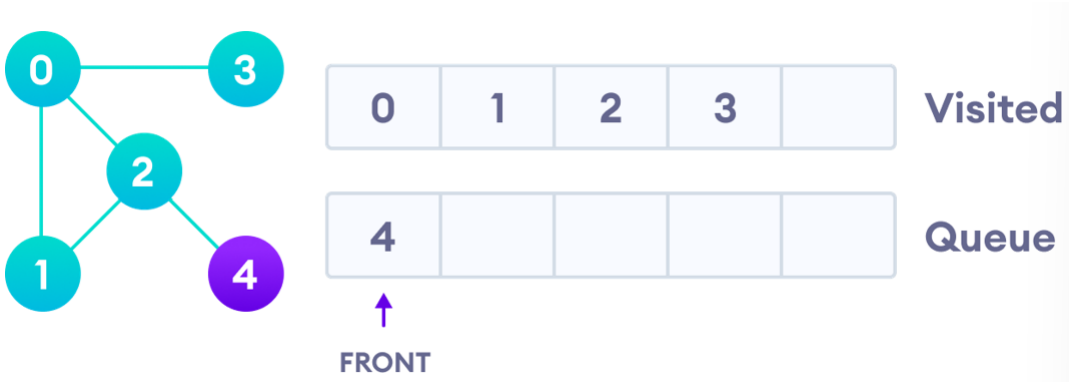
Next, we visit the element at the front of queue i.e 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



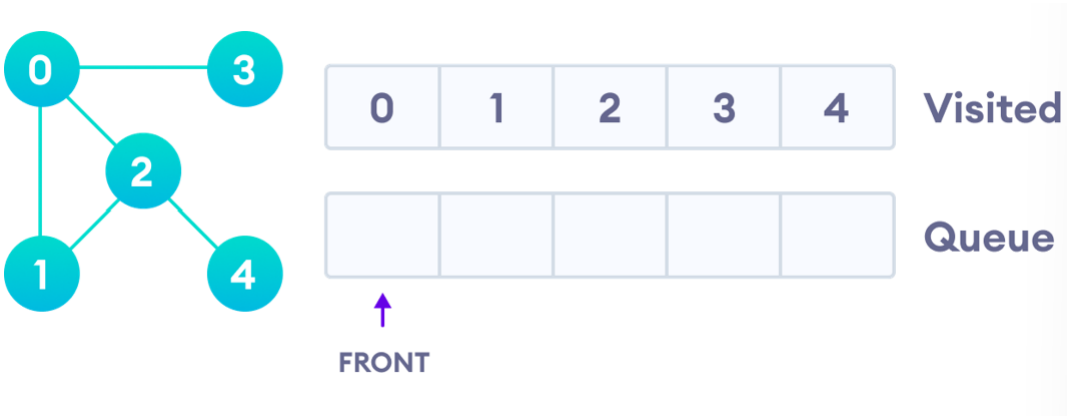
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the Queue and visit 3 , which is at the front of the queue.



Visit 2 which was added to queue earlier to add its neighbours



Only 4 remains in the queue since the only adjacent node of 3 i.e 0 is already visited. We visit it



Visit last remaining item in the queue to check if it has unvisited neighbours.

```
void BFS(struct Graph* graph, int startVertex) {

int visited[graph->numVertices];
```

```
for (int i = 0; i < graph->numVertices; i++) {
visited[i] = 0;
}
struct Node* queue = NULL; // Implement queue separately
// Enqueue startVertex and mark it as visited
// Continue the BFS process
}
```

Depth-First Search (DFS):

- Uses a stack (or recursion) to explore as far as possible along each branch before backtracking.

Algorithm:

1. Start from a selected node, mark it as visited.

2. Recursively visit all its unvisited neighbors.

-DFS traversal of a graph produces a spanning tree as final result. Spanning tree is a graph without loops.

-We use Stack Data Structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Step 1: Define a stack of size total number of vertices in the graph

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the stack.

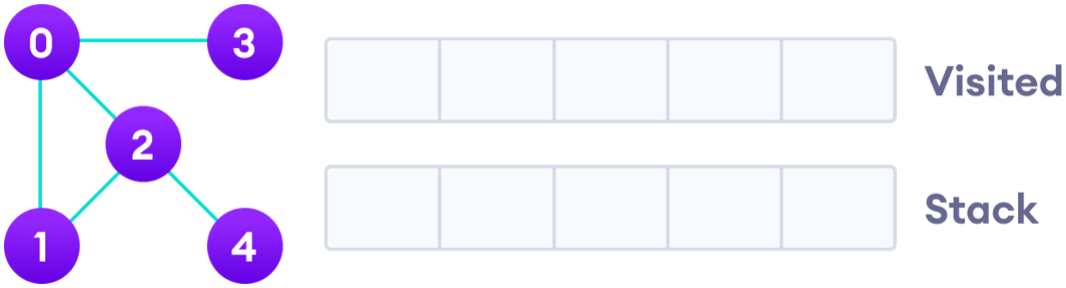
Step 3: Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4: Repeat Step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5: When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

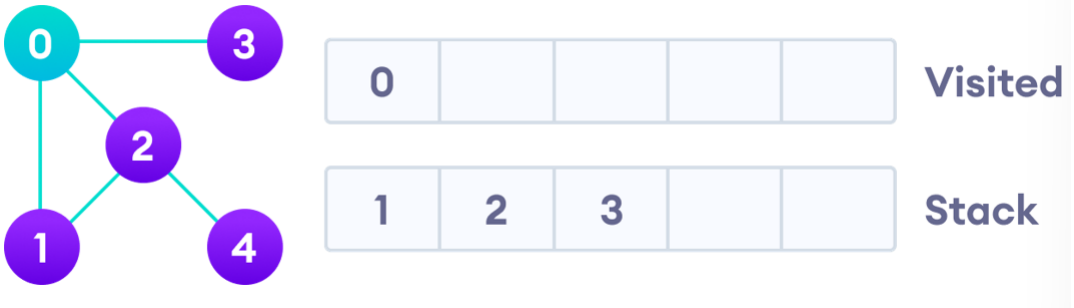
Step 6: Repeat Steps 3,4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.



Undirected graph with 5 vertices

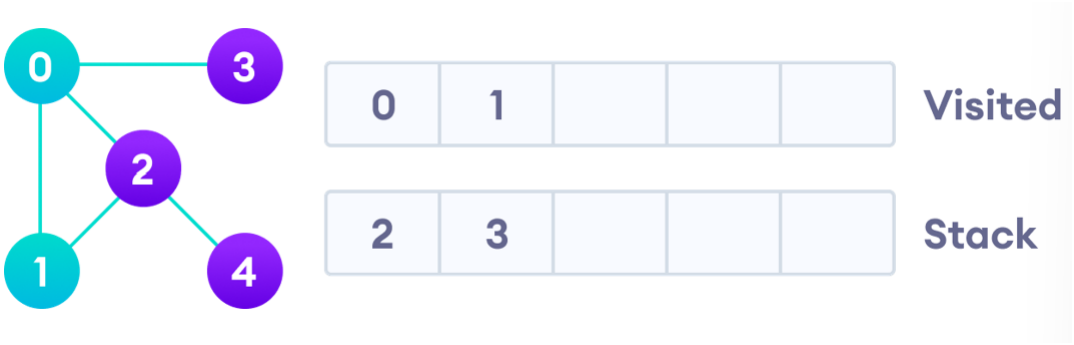
We start from vertex 0, the DFS algorithm starts by putting it in the VISITED list and putting all its adjacent vertices in the stack.



Visit the element and put it in the visited list

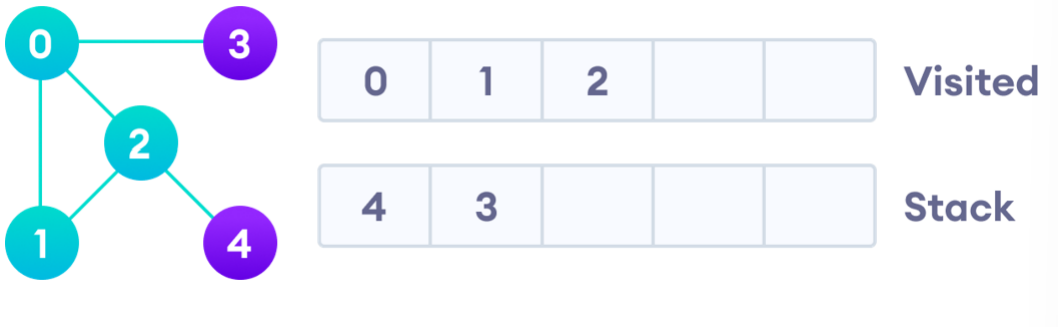
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes.

Since 0 has already been visited, we visit 2 instead.

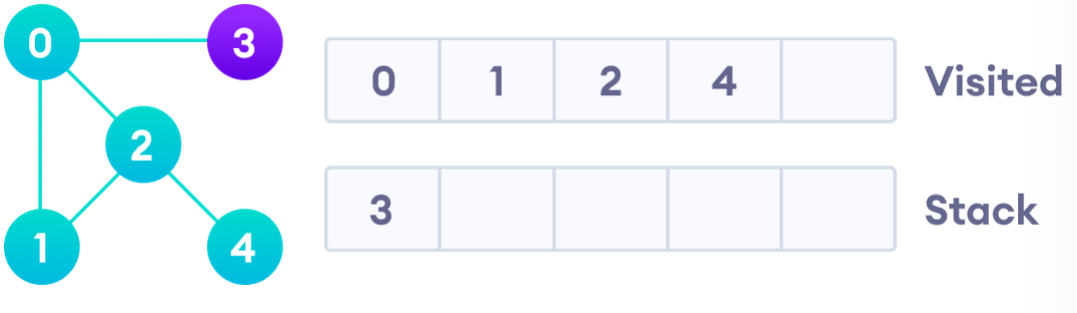


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

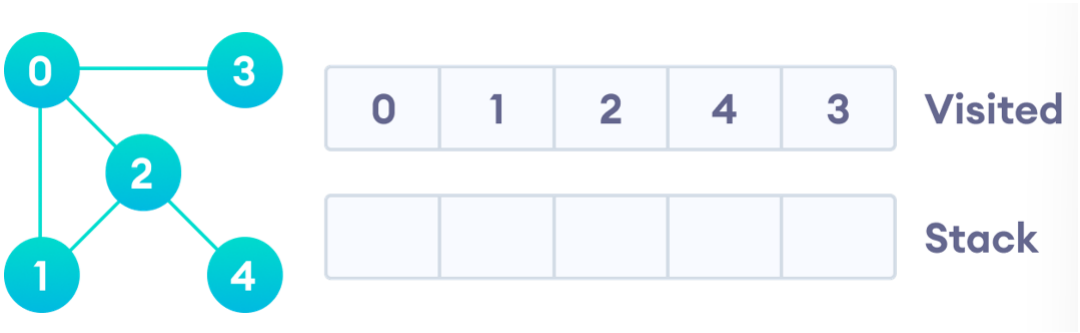


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph




```
void DFS(struct Graph* graph, int vertex, int visited[]) {
visited[vertex] = 1; // Mark as visited
// Process the vertex
struct Node* temp = graph->adjLists[vertex];
while (temp) {
int connectedVertex = temp->vertex;
if (!visited[connectedVertex]) {
DFS(graph, connectedVertex, visited);
}
temp = temp->next;
}
}
```

COMPLEXITY OF DEPTH FIRST SEARCH

The time complexity of the DFS algorithm is represented in the form of O (V + E)

Where V is the number of nodes and E is the number of edges The Space Complexity of the algorithm is O (V)