**Topic 1 : Introduction to Graphs**

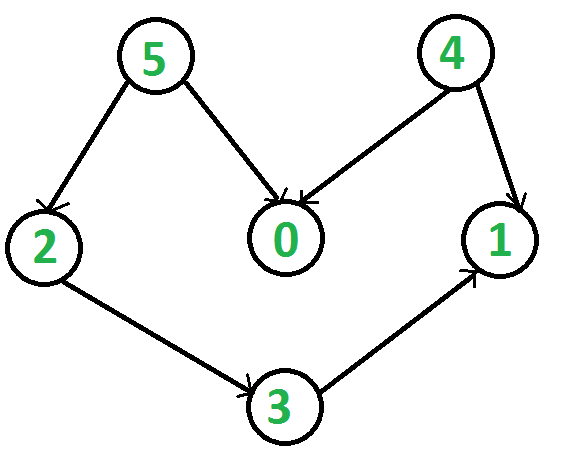
A ***Graph*** is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pairs of the form (u, v) called an edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). 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.

**Graphs are used to represent many real-life applications**:

* Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
* Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

**Directed and Undirected Graphs**

* **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.  
    
  For Example, the below graph is a directed graph:  
  
* **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**.  
    
  Following is an example of an undirected graph with 5 vertices:  
  

**Representing Graphs**

Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

* **Adjacency Matrix:** The 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 adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.  
    
  The adjacency matrix for the above example undirected graph is:  
  Adjacency Matrix Representation  
    
  ***Pros****:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).  
    
  ***Cons****:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

* **Adjacency List:** Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.  
    
  Adjacency List Representation of Graph  
    
  *Below is the implementation of the adjacency list representation of Graphs*:  
    
  **Note**: In below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of a linked list. The vector implementation has advantages of cache friendliness.

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A simple representation of graph using STL

#include<bits/stdc++.h>

using namespace std;

// A utility function to add an edge in an

// undirected graph.

void addEdge(vector<int> adj[], int u, int v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

// A utility function to print the adjacency list

// representation of graph

void printGraph(vector<int> adj[], int V)

{

for (int v = 0; v < V; ++v)

{

cout << "\n Adjacency list of vertex "

<< v << "\n head ";

for (auto x : adj[v])

cout << "-> " << x;

printf("\n");

}

}

// Driver code

int main()

{

int V = 5;

Run

Java



**Output**:

Adjacency list of vertex 0  
 head -> 1-> 4  
  
 Adjacency list of vertex 1  
 head -> 0-> 2-> 3-> 4  
  
 Adjacency list of vertex 2  
 head -> 1-> 3  
  
 Adjacency list of vertex 3  
 head -> 1-> 2-> 4  
  
 Adjacency list of vertex 4  
 head -> 0-> 1-> 3

***Pros****:* Saves space O(|V|+|E|). In the worst case, there can be a C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.  
  
***Cons****:* Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

**Topic 2 : Breadth First Traversal of a Graph**

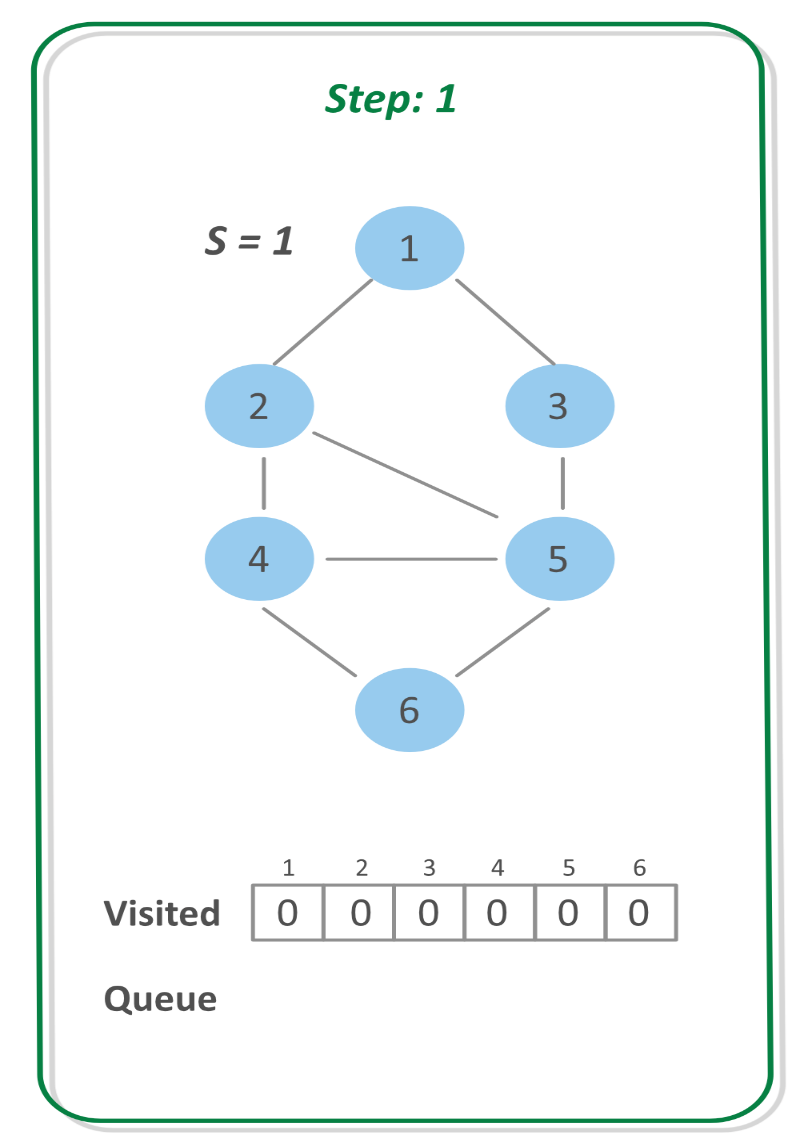
The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.  
  
The BFS traversal of Graphs also traverses the graph in levels. It starts the traversal with a given vertex, visits all of the vertices adjacent to the initially given vertex and pushes them all to a queue in order of visiting. Then it pops an element from the front of the queue, visits all of its neighbours and pushes the neighbours which are not already visited into the queue and repeats the process until the queue is empty or all of the vertices are visited.  
  
The BFS traversal uses an auxiliary boolean array, say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.  
  
**Complete Algorithm**:

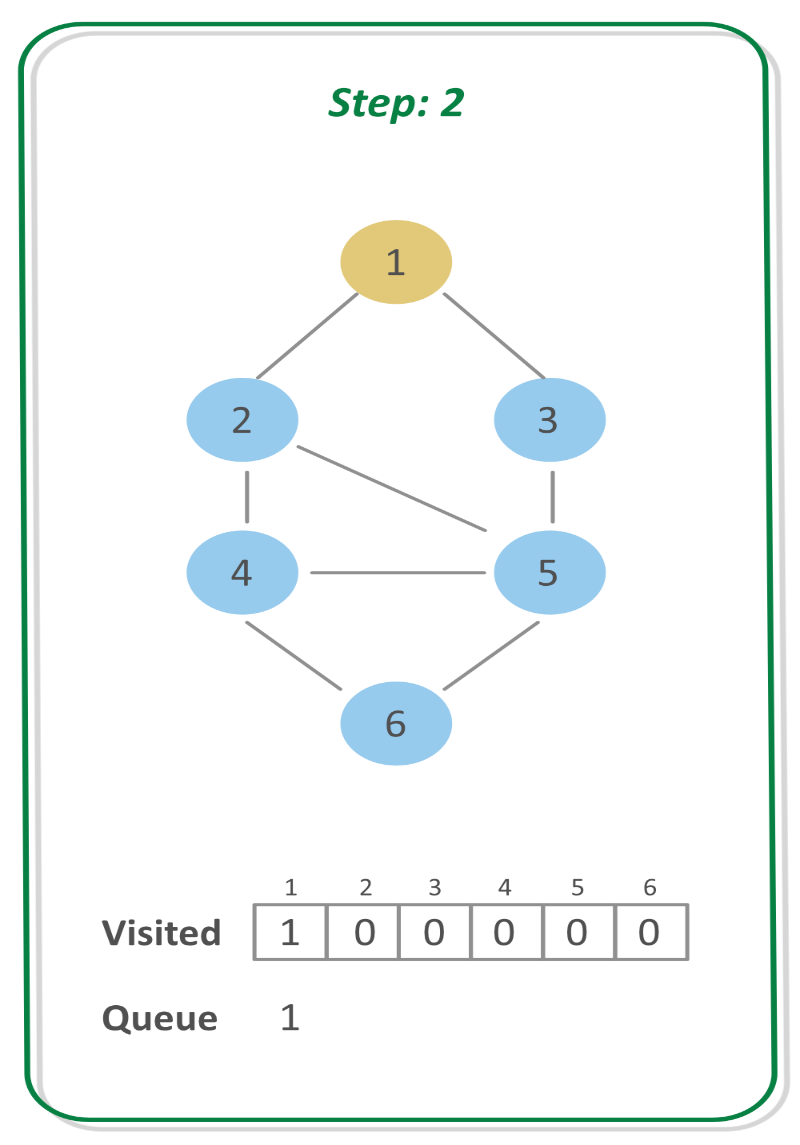
1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
3. Until the Queue is non-empty, repeat the below steps:  
   * Pop an element from the queue and print the popped element.
   * Traverse all of the vertices adjacent to the vertex popped from the queue.
   * If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

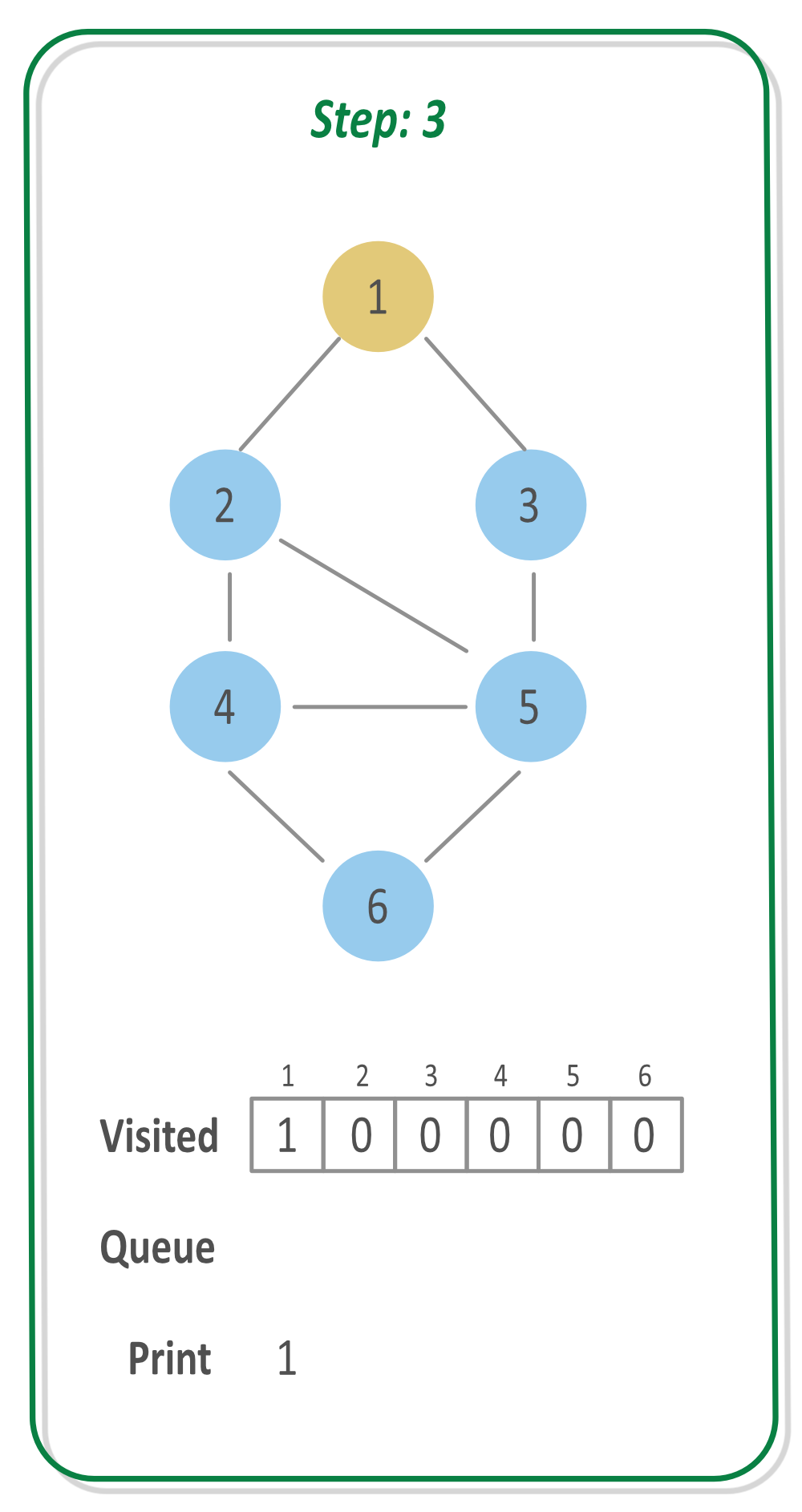
**Illustration**:

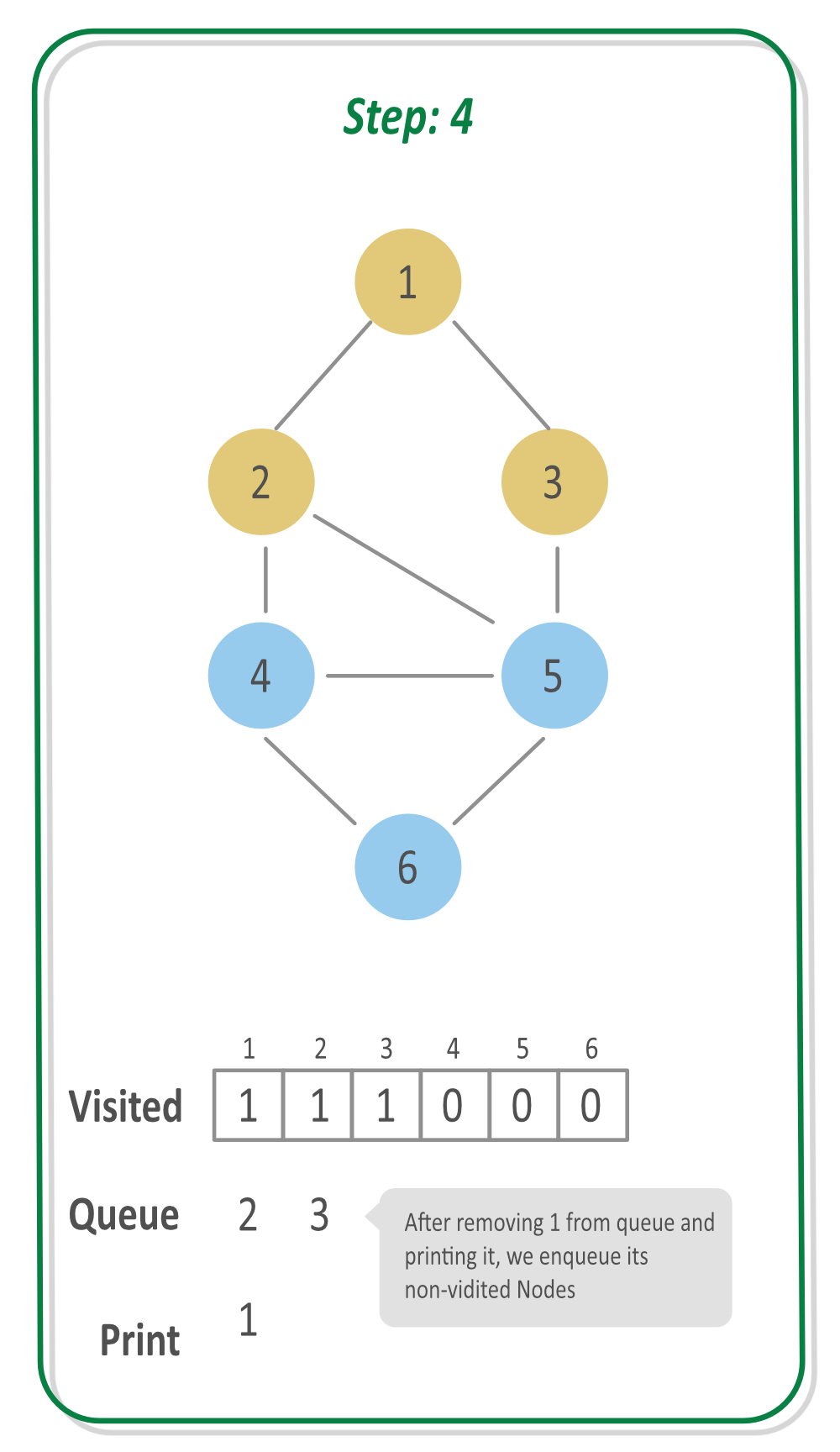
Consider the graph shown in the below Image. The vertices marked **blue** are *not-visited* vertices and the vertices marked **yellow** are *visited*. The vertex numbered **1** is the source vertex, i.e. the BFS traversal will start from the vertex 1.  
  
Following the BFS algorithm:

* Mark the vertex 1 visited in the visited[] array and push it to the queue.



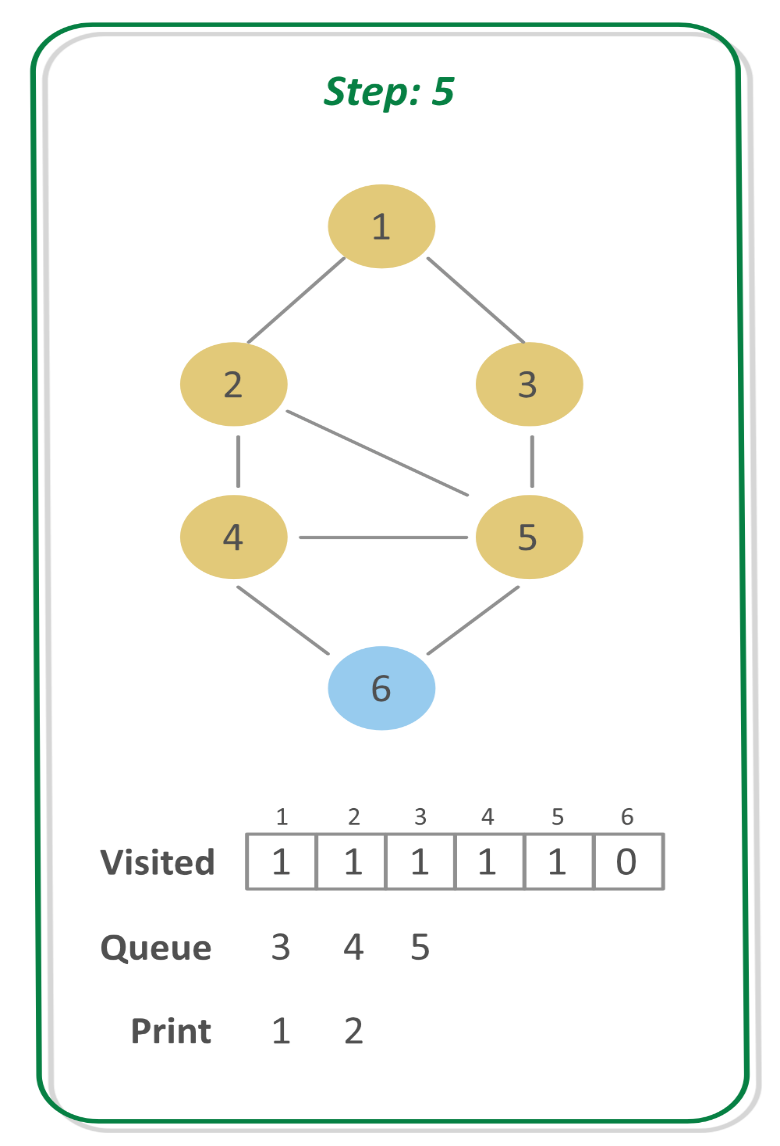


**Step 3**: POP the vertex at the front of the queue that is 1, and print it.  


**Step 4**: Check if adjacent vertices of the vertex 1 are not already visited. If not, mark them visited and push them back to the queue.  
  
 

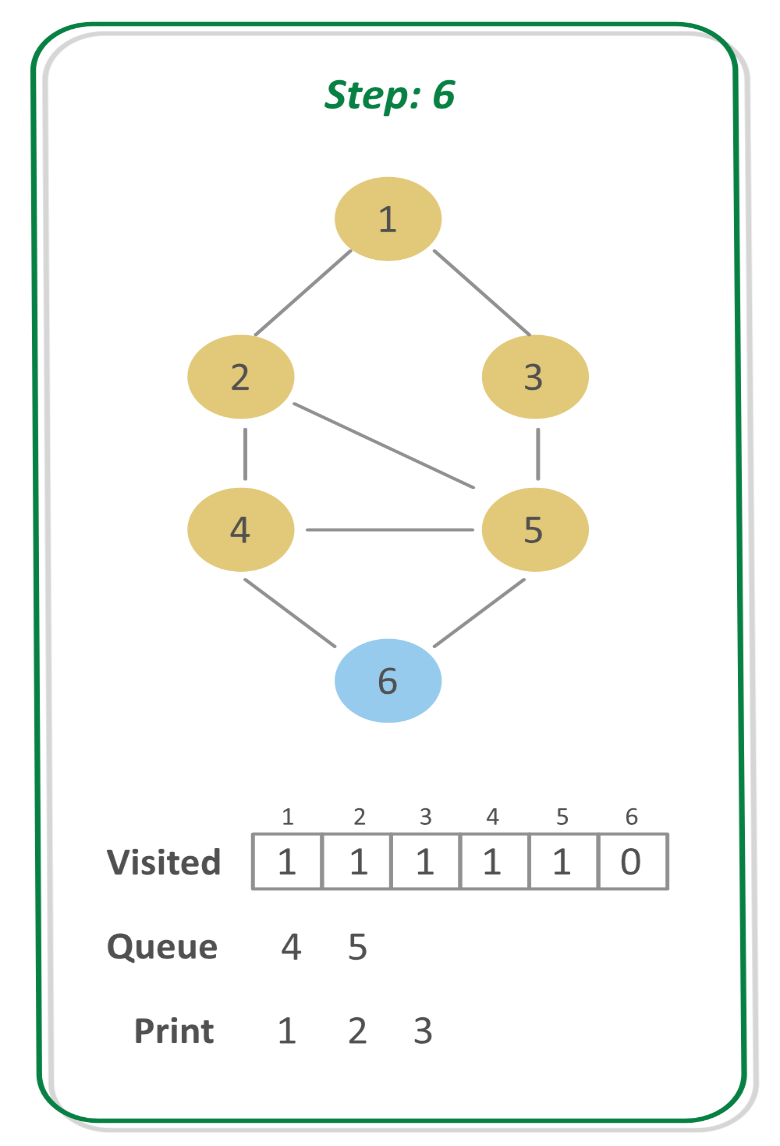
**Step 5:**

* POP the vertex at front that is 2 and print it.
* Check if the adjacent vertices of 2 are not already visited. If not, mark them visited and push them to queue. So, push 4 and 5.



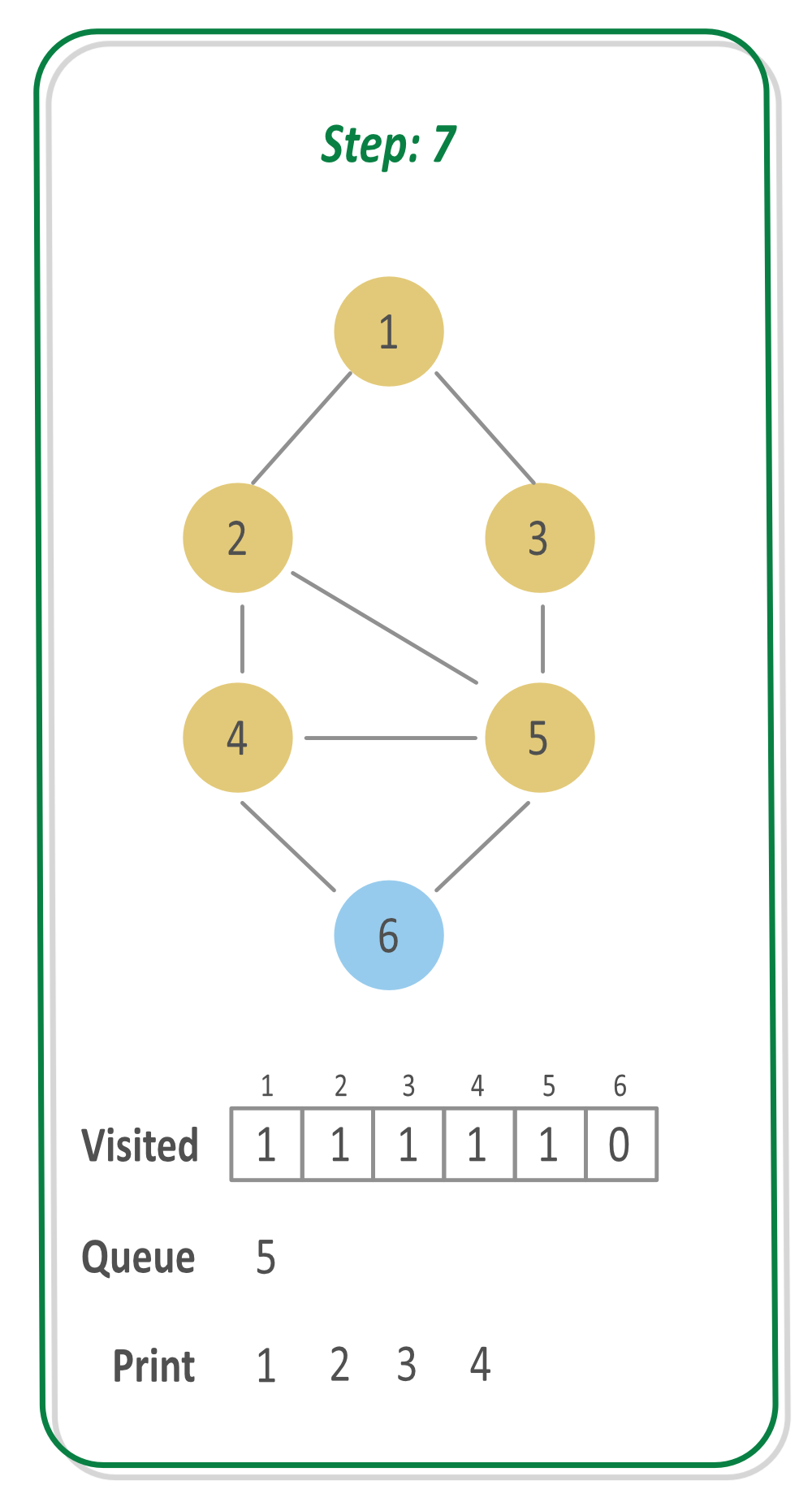
**Step 6:**

* POP the vertex at front that is 3 and print it.
* Check if the adjacent vertices of 3 are not already visited. If not, mark them visited and push them to queue. So, don't push anything.



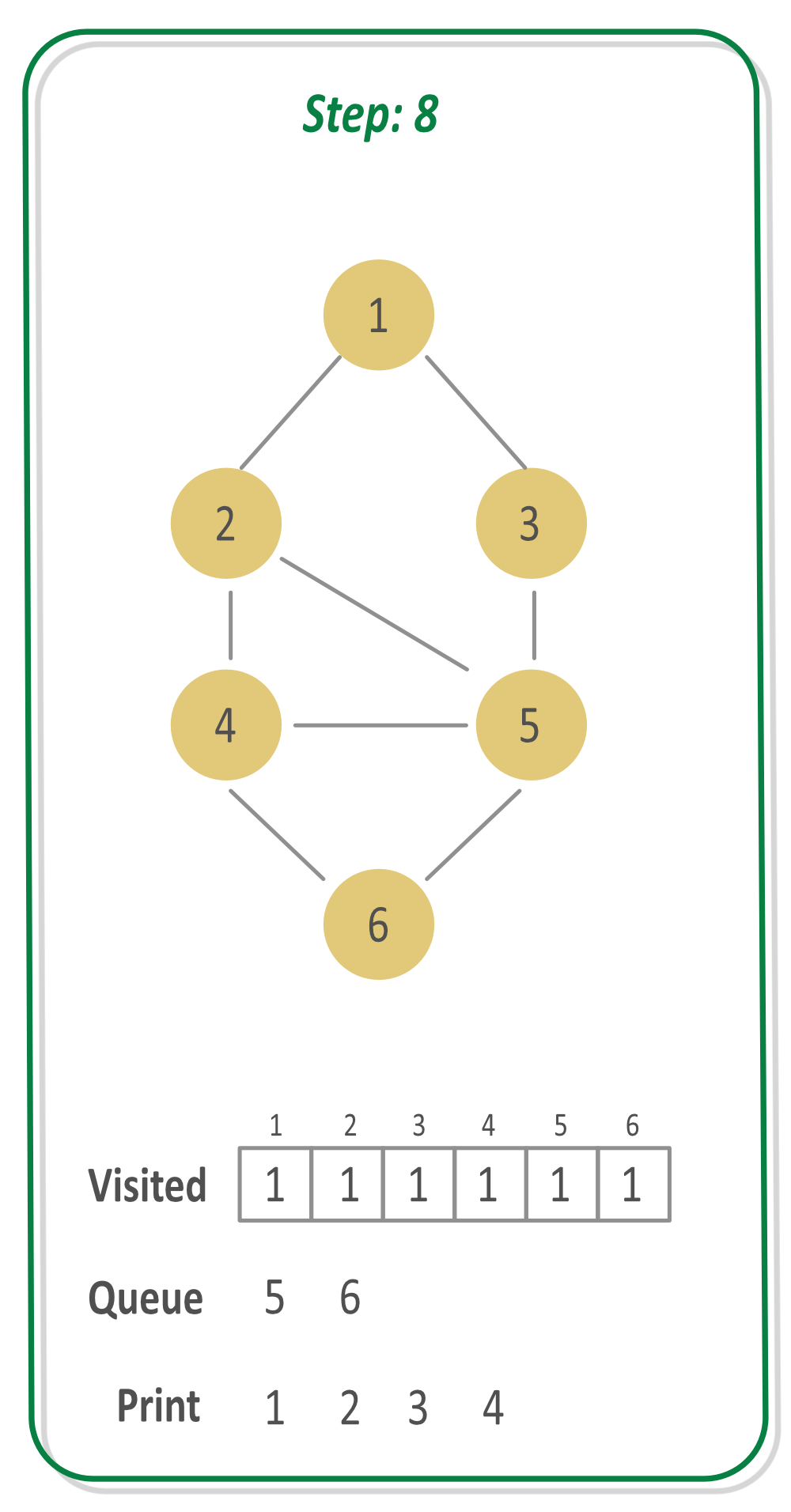
**Step 7:**

* POP the vertex at front that is 4 and print it.



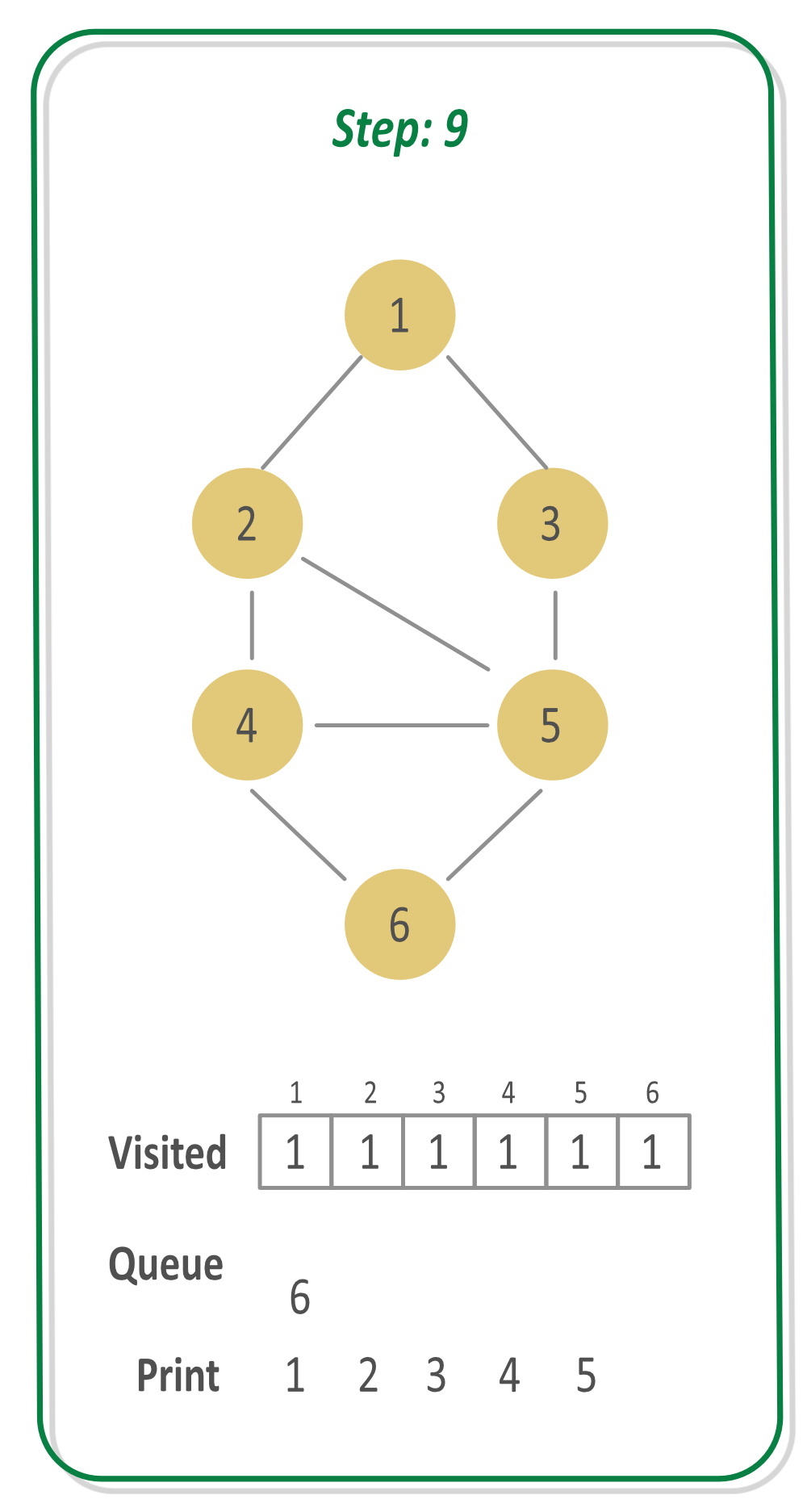
**Step 8:**

* Check if the adjacent vertices of 4 are not already visited. If not, mark them visited and push them to queue. So, push 6 to the queue.



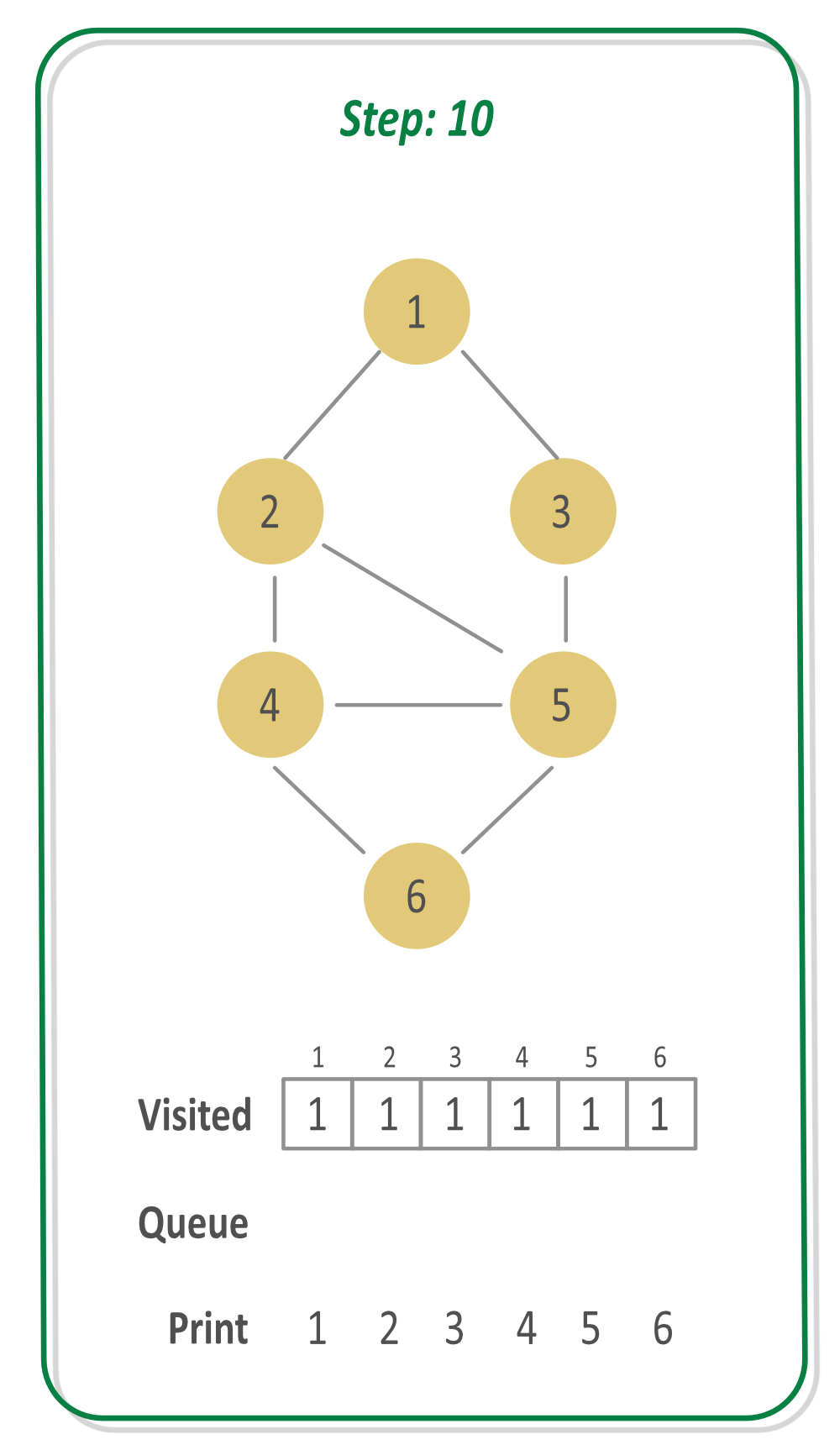
**Step 9:**

* POP the vertex at front, that is 5 and print it.
* Since, all of its adjacent vertices are already visited, don't push anything.



**Step 10:**

* POP the vertex at front, that is 6 and print it.
* Since, all of its adjacent vertices are already visited, don't push anything.

***Since the Queue is empty now, it means that the complete graph is traversed.***

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to implement BFS traversal

// of a Graph

#include <bits/stdc++.h>

using namespace std;

// A utility function to add an edge in an

// undirected graph.

void addEdge(vector<int> adj[], int u, int v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

// Function to perform BFS traversal of the given Graph

void BFS(vector<int> adj[], int V)

{

// Initialize a boolean array

// to keep track of visited vertices

bool visited[V + 1];

// Mark all vertices not-visited initially

for (int i = 1; i <= V; i++)

visited[i] = false;

// Create a Queue to perform BFS

queue<int> q;

// Our source vertex is vertex

// numbered 1

Run

Java

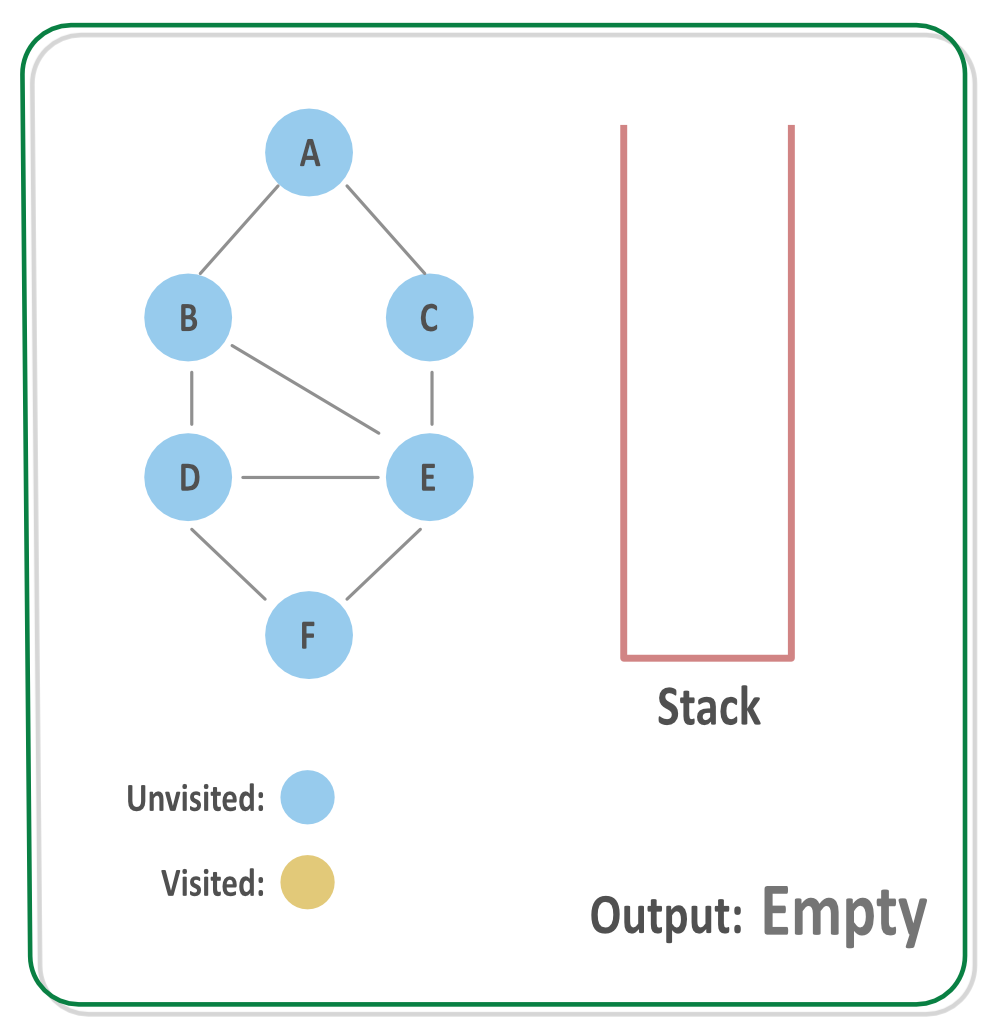


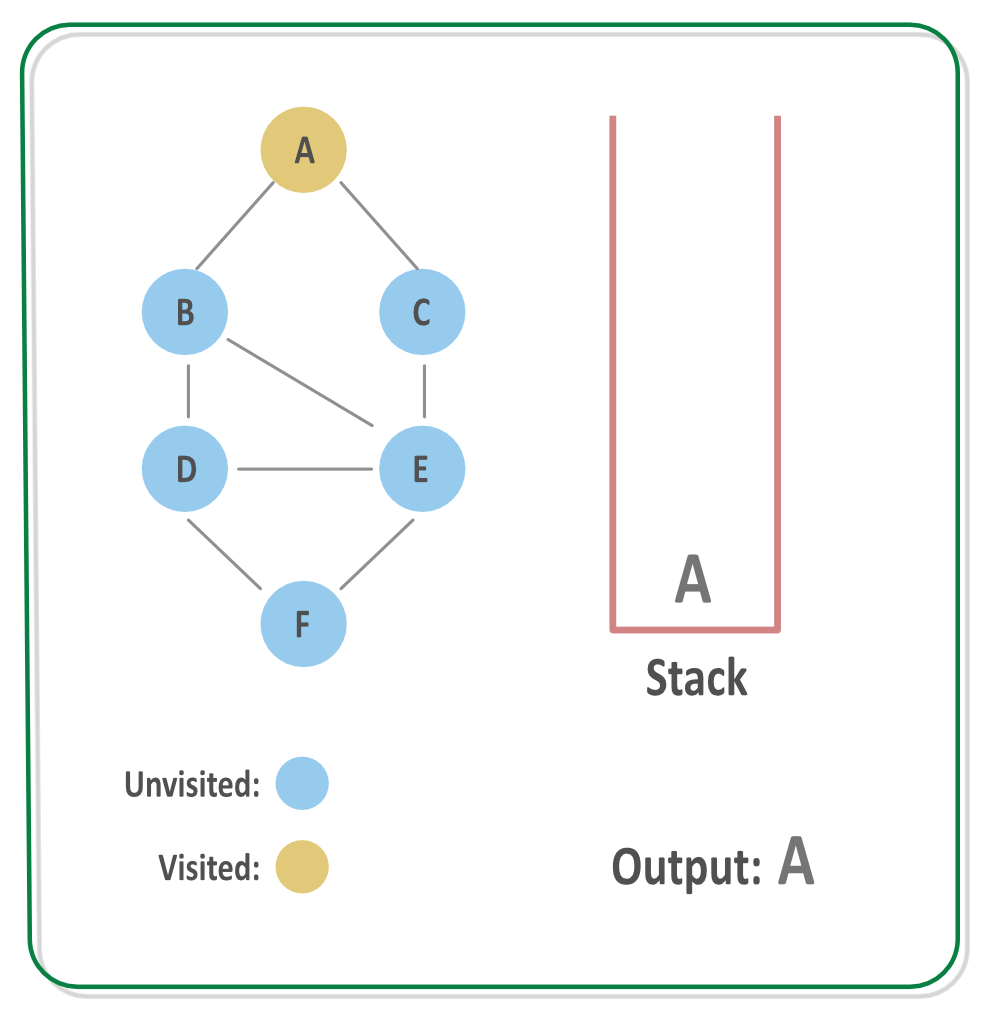
**Output:**

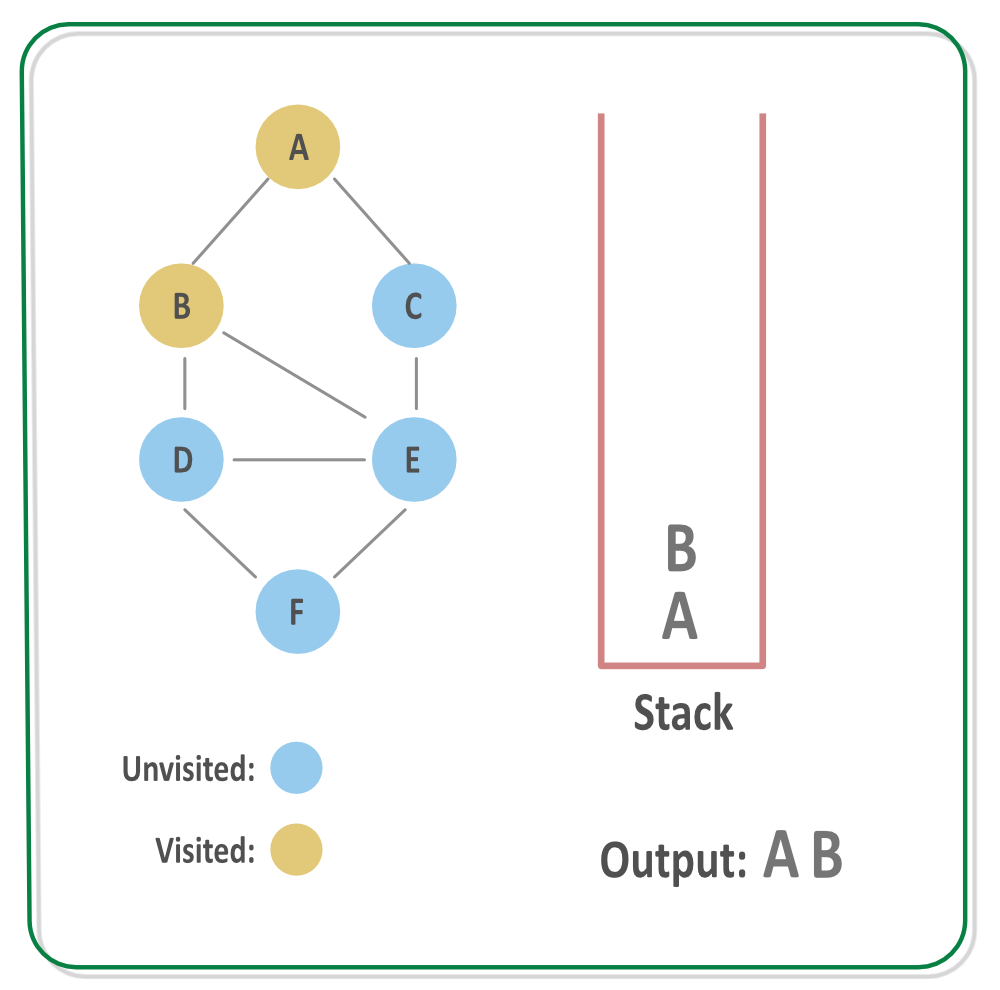
1 2 3 4 5 6

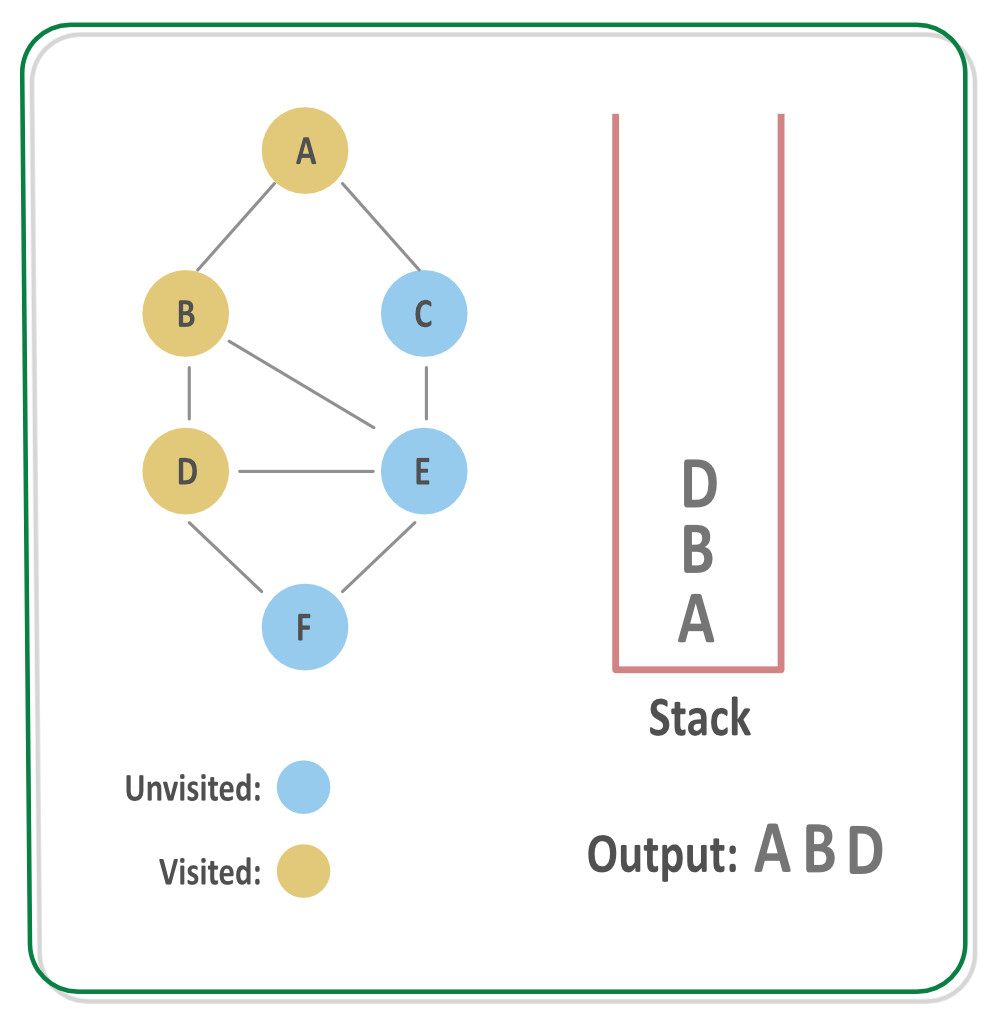
**Topic 3 : Depth First Traversal of a Graph**

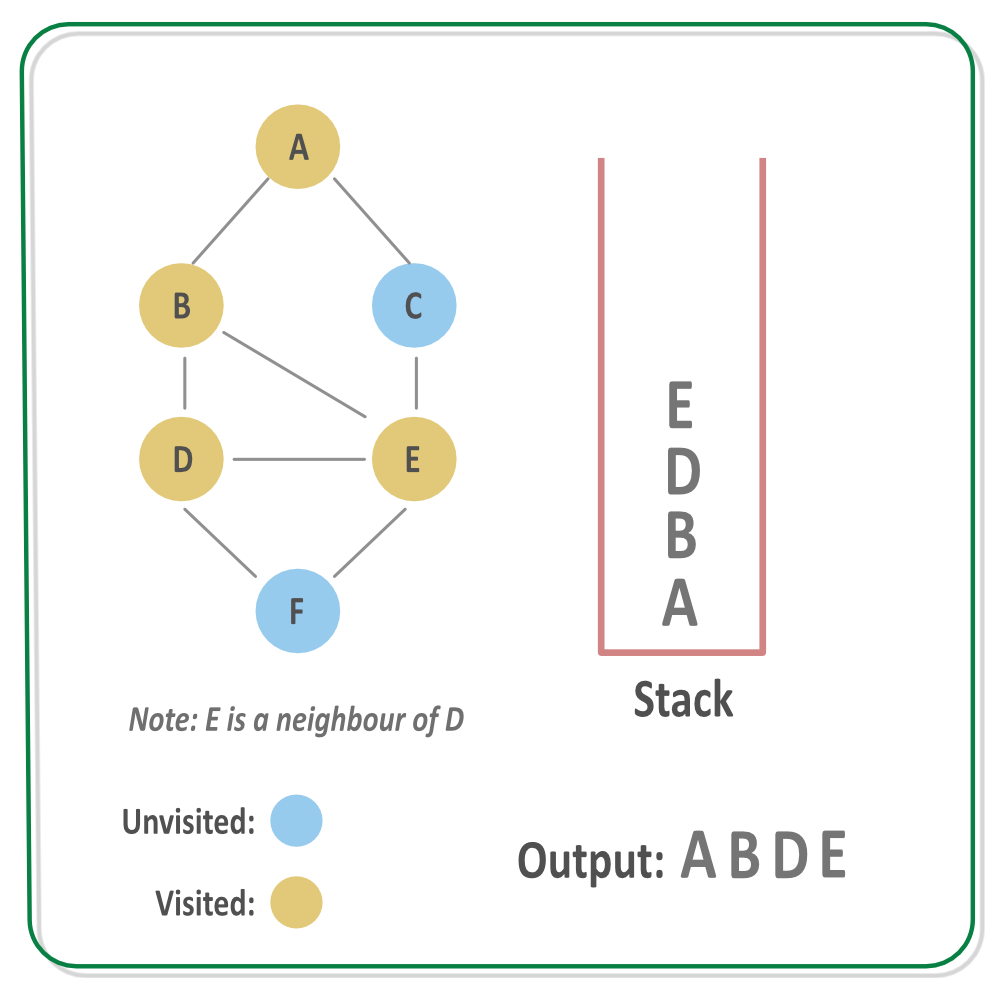
The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.  
  
We can implement the DFS traversal algorithm using a recursive approach. While performing the DFS traversal the graph may contain a cycle and the same node can be visited again, so in order to avoid this we can keep track of visited arrays using an auxiliary array. On each step of the recursion mark, the current vertex visits and calls the recursive function again for all the adjacent vertices.  
  
**Illustration**:

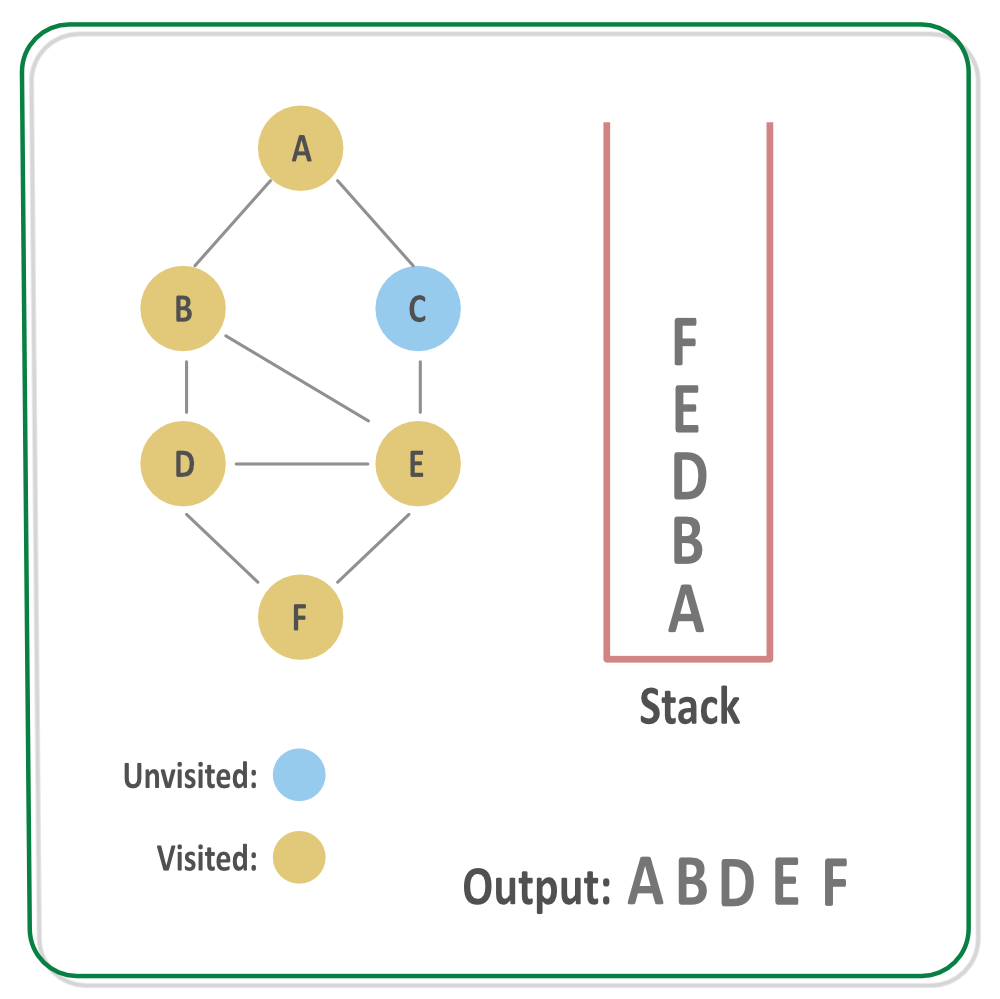
**Step 1:**Consider the below graph and apply the DFS algorithm recursively for every node using an auxiliary stack for recursive calls and an array to keep track of visited vertices.  


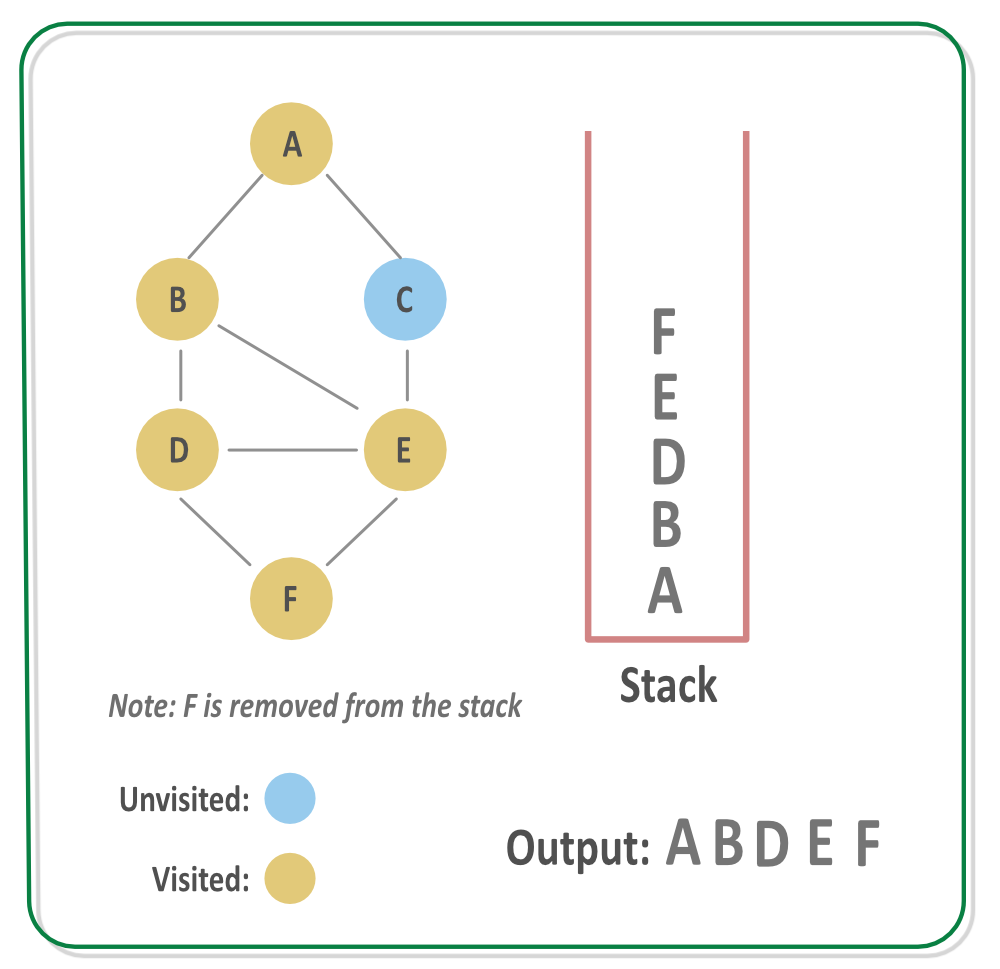
**Step 2:** Process the vertex A, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is B. The vertex A is put on the auxiliary stack for now.  


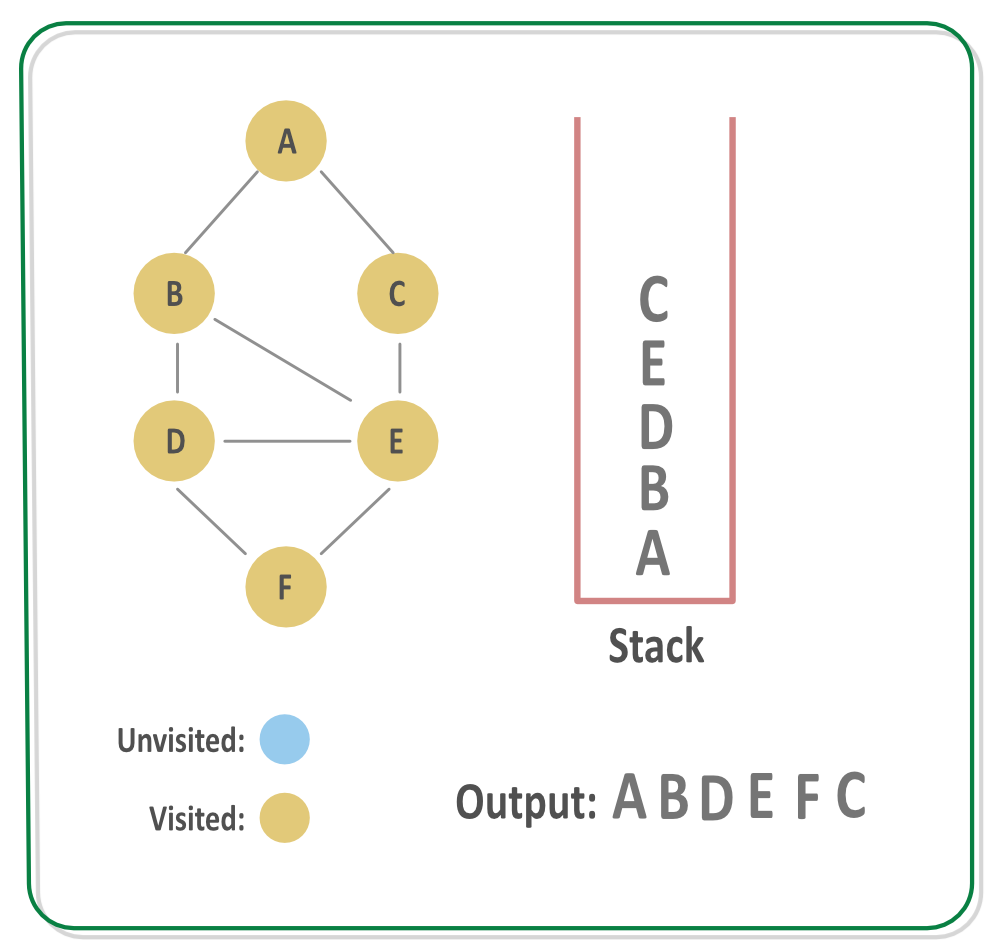
**Step 3:**Process the vertex B, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is D. The vertex B is put on the auxiliary stack for now.  


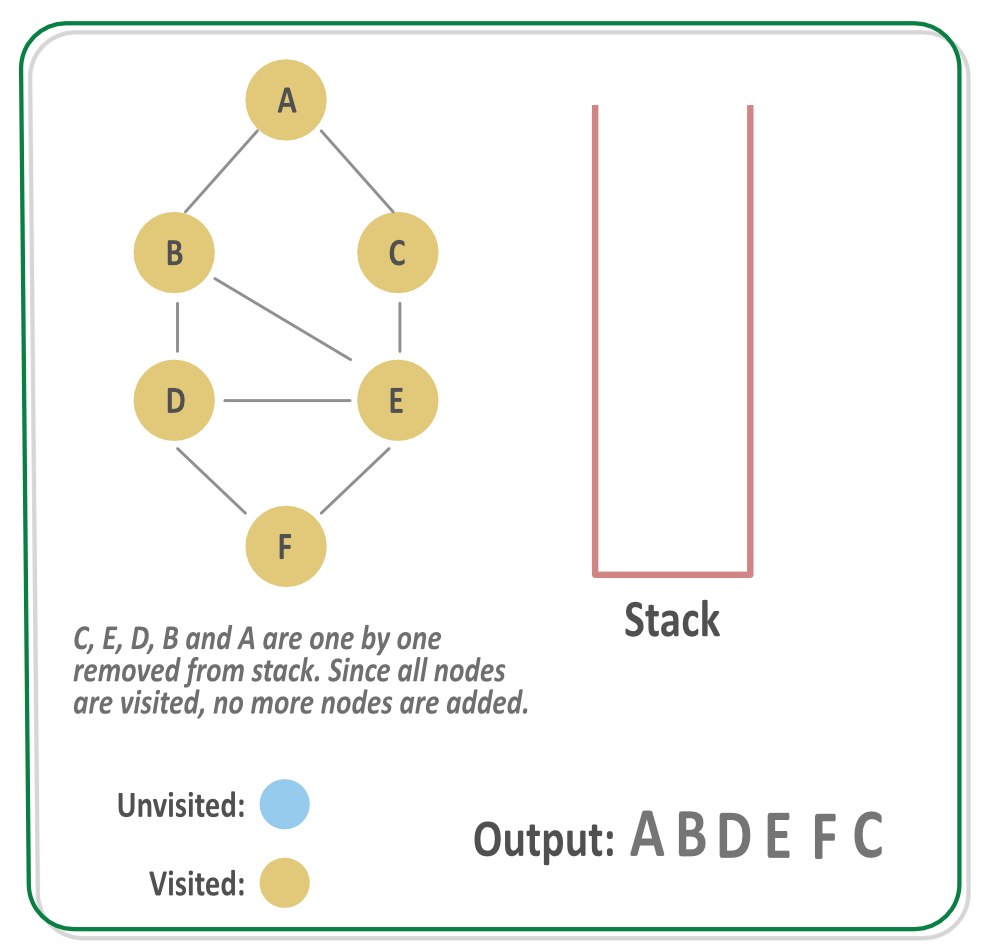
**Step 4:** Process the vertex D, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is E. The vertex D is put on the auxiliary stack for now.  


**Step 5:**Process the vertex E, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is F. The vertex E is put on the auxiliary stack  
for now.  


**Step 6:** Process the vertex F, mark it visited and call DFS for its adjacent vertices. There are no adjacent vertices of the vertex F, so backtrack to find a new path. The vertex F is put on the auxiliary stack for now.  


**Step 7:**Since the vertex F has no adjacent vertices left unvisited, backtrack and go to previous call, that is process any other adjacent vertex of node E, that is C.  


**Step 8:** Process the vertex C, mark it visited and call DFS for its adjacent vertices. The vertex C is put on the auxiliary stack for now.  


**Step 9**: Since there are no adjacent vertices of C, backtrack to find a new path and keep removing nodes from stack until a new path is found. Since all of the nodes are processed so the stack will get empty.  


**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include<iostream>

#include<list>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing

// adjacency lists

list<int> \*adj;

// A recursive function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

Graph::Graph(int V)

Run

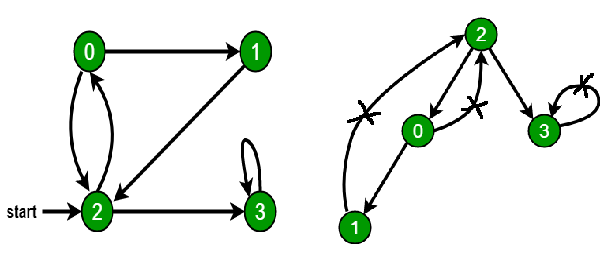
Java



**Output:**

Following is Depth First Traversal (starting from vertex 2)  
2 0 1 3

**Topic 4 : Detecting Cycle in a Graph**

**Problem**: Given a graph(directed or undirected), check whether the graph contains a cycle or not.  
  
**Solution**: Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.  
  
  
To detect a back edge, we can keep track of vertices currently in the recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We can use an auxiliary array, say, *recStack[]* to keep track of vertices in the recursion stack.  
  
Therefore, for every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not a parent of v, then there is a cycle in the graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.  
  
**Note**: The above method can be used to detect a cycle in both *directed* and *undirected graphs*.  
  
Below is the implementation of the above approach:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ Program to detect cycle in a graph

#include<iostream>

#include <list>

#include <limits.h>

using namespace std;

class Graph

{

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

bool isCyclicUtil(int v, bool visited[], bool \*rs); // used by isCyclic()

public:

Graph(int V); // Constructor

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

bool isCyclic(); // returns true if there is a cycle in this graph

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

// Utility function to detect cycle in a Graph

Run

Java



**Output**:

Graph contains cycle

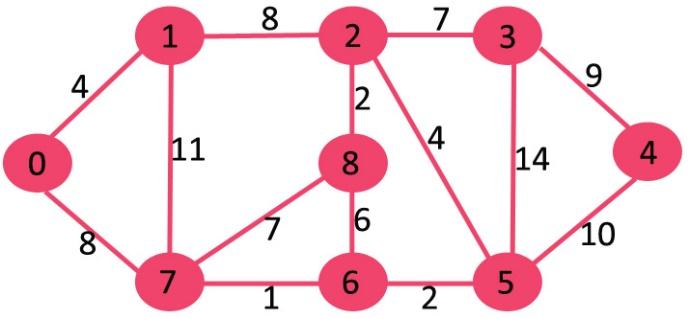
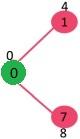
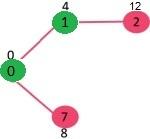
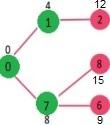
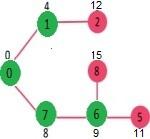
The ***Time Complexity*** of this method is the same as the time complexity of *DFS traversal* which is O(V+E), where V is the number of vertices and E is the number of edges.

**Topic 5 : Dikstra’s Algorithm for Shortest Path in a Weighted Graph**

*Given a graph and a source vertex in the graph, find the shortest paths from source to all vertices in the given graph.*

Dijkstra's algorithm is a variation of the BFS algorithm. In Dijkstra's Algorithm, a SPT*(shortest path tree)* is generated with a given source as root. Each node at this SPT stores the value of the shortest path from the source vertex to the current vertex. We maintain two sets, one set contains vertices included in the shortest path tree, the other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.  
  
Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given weighted graph.  
  
**Algorithm**:

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* doesn't include all vertices:  
   * Pick a vertex u which is not there in *sptSet* and has minimum distance value.
   * Include u to *sptSet*.
   * Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand the above algorithm with the help of an example. Consider the below given graph:  
  
  
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, including it in *sptSet*. So *sptSet*becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.  
  
  
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.  
  
  
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
  
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.  
  
  
We repeat the above steps until *sptSet*doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).  
  
**Implementation**:  
Since at every step we need to find the vertex with minimum distance from the source vertex from the set of vertices currently not added to the SPT, so we can use a min heap for easier and efficient implementation. Below is the complete algorithm using priority\_queue(min heap) to implement Dijkstra's Algorithm:

1) Initialize distances of all vertices as infinite.  
  
2) Create an empty priority\_queue pq. Every item  
 of pq is a pair (weight, vertex). Weight (or   
 distance) is used as the first item of pair  
 as the first item is by default used to compare  
 two pairs  
  
3) Insert source vertex into pq and make its  
 distance as 0.  
  
4) While either pq doesn't become empty  
 a) Extract minimum distance vertex from pq.   
 Let the extracted vertex be u.  
 b) Loop through all adjacent of u and do   
 following for every vertex v.  
  
 // If there is a shorter path to v  
 // through u.   
 If dist[v] > dist[u] + weight(u, v)  
  
 (i) Update distance of v, i.e., do  
 dist[v] = dist[u] + weight(u, v)  
 (ii) Insert v into the pq (Even if v is  
 already there)  
   
5) Print distance array dist[] to print all shortest  
 paths.

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Program to find Dijkstra's shortest path using

// min heap in STL

#include<bits/stdc++.h>

using namespace std;

# define INF 0x3f3f3f3f

// iPair ==> Integer Pair

typedef pair<int, int> iPair;

// To add an edge

void addEdge(vector <pair<int, int> > adj[], int u,

int v, int wt)

{

adj[u].push\_back(make\_pair(v, wt));

adj[v].push\_back(make\_pair(u, wt));

}

// Prints distance of shortest paths from the source

// vertex to all other vertices

void shortestPath(vector<pair<int,int> > adj[], int V, int src)

{

// Create a priority queue to store vertices that

// are being preprocessed. This is weird syntax in C++.

// Refer below link for details of this syntax

// http://geeksquiz.com/implement-min-heap-using-stl/

priority\_queue< iPair, vector <iPair> , greater<iPair> > pq;

// Create a vector for distances and initialize all

Run

Java



**Output**:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Time Complexity**: The time complexity of the Dijkstra's Algorithm when implemented using a min heap is O(E \* logV), where E is the number of Edges and V is the number of vertices.  
  
**Note**: The Dijkstra's Algorithm **doesn't work** in the case when the Graph has negative edge weight.

**Topic 6 : Bellman-Ford Algorithm for Shortest Path**

**Problem**: Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.  
  
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suits well for distributed systems. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*  
  
**Algorithm**: Following are the detailed steps.

* *Input:* Graph and a source vertex *src*.
* *Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1. This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is the source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in a given graph.  
   Do following for each edge u-v:  
   * If dist[v] > dist[u] + weight of edge uv, then update dist[v] as:*dist[v] = dist[u] + weight of edge uv*.
3. This step reports if there is a negative weight cycle in the graph. Do following for each edge u-v. If dist[v] > dist[u] + weight of edge **uv**, then "Graph contains negative weight cycle".

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.  
  
***How does this work?*** Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most **i**edges are calculated. There can be maximum **|V| - 1** edge in any simple path, that is why the outer loop runs |v| - 1 time. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most (i+1) edge.  
  
**Example**:  
Let us understand the algorithm with the following example graph.  
  
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*  
Example Graph  
Let all edges be processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get the following distances when all edges are processed for the first time. The first row shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.  
  
  
The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed a second time (The last row shows final values).  
  
  
The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.  
  
**Implementation:**

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph

struct Edge

{

int src, dest, weight;

};

// a structure to represent a connected, directed and

// weighted graph

struct Graph

{

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

Run

Java



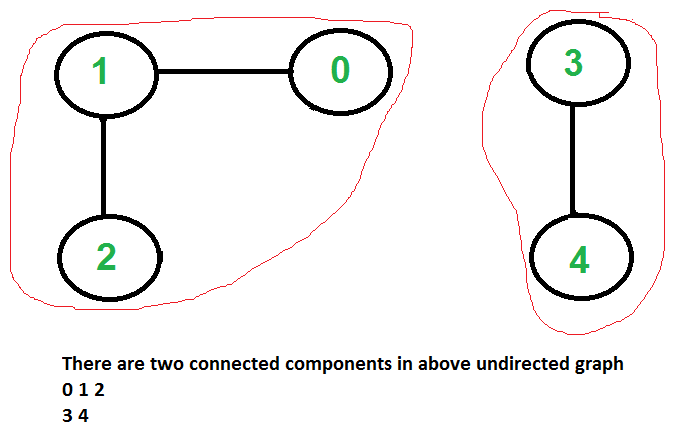
**Output:**

Vertex Distance from Source  
0 0  
1 -1  
2 2  
3 -2  
4 1

**Important Notes**:

1. Negative weights are found in various applications of graphs. For example, instead of paying the cost for a path, we may get some advantage if we follow the path.
2. Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijksra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

**Topic 7 : Number of Strongly Connected Components in an Undirected Graph**

**Problem**: Given an Undirected Graph. The task is to find the count of the number of *strongly connected components* in the given Graph. A **Strongly Connected Component** is defined as a subgraph of this graph in which every pair of vertices has a path in between.  
  
  
  
Finding the connected components for an undirected graph is an easier task. The idea is to traverse all of the unvisited vertices, and for each unvisited vertex print, it's DFS or BFS traversal.  
  
Below is the algorithm following the DFS traversal to find all connected components in an undirected graph:

1) Initialize all vertices as not visited.  
2) Do following for every vertex 'v'.  
 (a) If 'v' is not visited before, call DFSUtil(v)  
 (b) Print new line character  
  
// This Function performs DFS traversal  
// of vertex v.  
DFSUtil(v)  
1) Mark 'v' as visited.  
2) Print 'v'  
3) Do following for every adjacent 'u' of 'v'.  
 If 'u' is not visited, then recursively call DFSUtil(u)

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to print connected components in

// an undirected graph

#include <iostream>

#include <list>

using namespace std;

// Graph class represents a undirected graph

// using adjacency list representation

class Graph {

int V; // No. of vertices

// Pointer to an array containing adjacency lists

list<int>\* adj;

// A function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

void addEdge(int v, int w);

void connectedComponents();

};

// Method to print connected components in an

// undirected graph

void Graph::connectedComponents()

{

// Mark all the vertices as not visited

bool\* visited = new bool[V];

for (int v = 0; v < V; v++)

Run

Java



**Output:**

Following are connected components   
0 1 2   
3 4

**Topic 8 : Sample Problems on Graph**

**Problem #1 : Find the number of islands**

**Description -** Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands  
**Example:**

Input : mat[][] = {{1, 1, 0, 0, 0},  
 {0, 1, 0, 0, 1},  
 {1, 0, 0, 1, 1},  
 {0, 0, 0, 0, 0},  
 {1, 0, 1, 0, 1}   
Output : 5

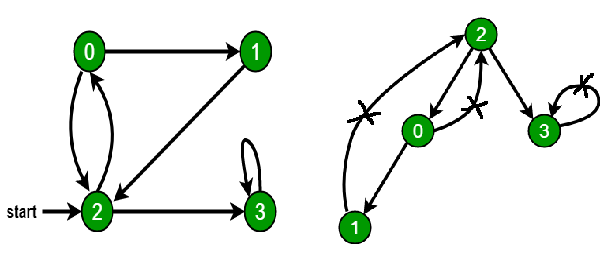
**Solution -** The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.  
  
**What is an island ?** A group of connected 1s forms an island. For example, the below matrix contains 5 islands

{**1**,  **1**, 0, 0, 0},  
 {0, **1**, 0, 0, **1**},  
 {**1**, 0, 0, **1**, **1**},  
 {0, 0, 0, 0, 0},  
 {**1**, 0, **1**, 0, **1**}

**Pseudo Code**

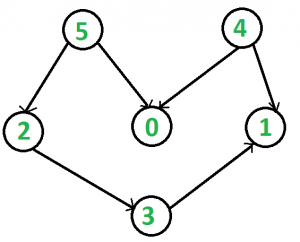
// A utility function to do DFS for a   
// 2D boolean matrix. It only considers   
// the 8 neighbours as adjacent vertices   
void DFS(int M[][COL], int row, int col,   
 bool visited[][COL])   
{   
 // These arrays are used to get   
 // row and column numbers of 8   
 // neighbours of a given cell   
 rowNbr[] = { -1, -1, -1, 0, 0, 1, 1, 1 }  
 colNbr[] = { -1, 0, 1, -1, 1, -1, 0, 1 }  
   
 // Mark this cell as visited   
 visited[row][col] = true  
   
 // Recur for all connected neighbours   
 for (int k = 0; k < 8; ++k)   
 if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited))   
 DFS(M, row + rowNbr[k], col + colNbr[k], visited)  
}   
   
// The main function that returns   
// count of islands in a given boolean   
// 2D matrix   
int countIslands(int M[][COL])   
{   
 // Make a bool array to mark visited cells.   
 // Initially all cells are unvisited   
 bool visited[ROW][COL]  
 memset(visited, 0, sizeof(visited))  
   
 // Initialize count as 0 and   
 // travese through the all cells of   
 // given matrix   
 int count = 0  
 for (int i = 0; i < ROW; ++i)   
 for (int j = 0; j < COL; ++j)   
   
 // If a cell with value 1 is not   
 if (M[i][j] && !visited[i][j]) {   
 // visited yet, then new island found   
 // Visit all cells in this island.   
 DFS(M, i, j, visited)   
   
 // and increment island count   
 ++count  
 }   
   
 return count  
}

**Problem #2 : Detect Cycle in a Directed Graph**

**Description -**Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false.  
**Solution -** Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.  
  
  
For a disconnected graph, we get the DFS forest as output. To detect cycles, we can check for a cycle in individual trees by checking back edges.  
  
To detect a back edge, we can keep track of vertices currently in the recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We have used the recStack[] array to keep track of vertices in the recursion stack.  
**Pseudo Code**

// Utility function to check back edge in a directed graph  
bool isCyclicUtil(int v, bool visited[], bool \*recStack)   
{   
 if(visited[v] == false)   
 {   
 // Mark the current node as visited and part of recursion stack   
 visited[v] = true  
 recStack[v] = true  
   
 // Recur for all the vertices adjacent to this vertex   
 list < int > :: iterator i  
 for(i = adj[v].begin(); i != adj[v].end(); ++i)   
 {   
 if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )   
 return true  
 else if (recStack[\*i])   
 return true  
 }   
   
 }   
 recStack[v] = false; // remove the vertex from recursion stack   
 return false;   
}   
   
// Returns true if the graph contains a cycle, else false.   
bool isCyclic()   
{   
 // Mark all the vertices as not visited and not part of recursion   
 // stack   
 bool \*visited = new bool[V]  
 bool \*recStack = new bool[V]  
 for(int i = 0; i < V; i++)   
 {  
 visited[i] = false  
 recStack[i] = false  
 }   
   
 // Call the recursive helper function to detect cycle in different   
 // DFS trees   
 for(int i = 0; i < V; i++)   
 if (isCyclicUtil(i, visited, recStack))   
 return true  
   
 return false  
}

**Problem #3 : Topological Sorting**

**Description -** Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.  
For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).  
  
**Solution -**We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.  
**Pseudo Code**

// A recursive function used by topological sort   
void topologicalSortUtil(int v, bool visited[],   
 stack &Stack)   
{   
 // Mark the current node as visited.   
 visited[v] = true  
   
 // Recur for all the vertices adjacent to this vertex   
 list < int > :: iterator i  
 for (i = adj[v].begin(); i != adj[v].end(); ++i)   
 if (!visited[\*i])   
 topologicalSortUtil(\*i, visited, Stack);   
   
 // Push current vertex to stack which stores result   
 Stack.push(v)  
}   
   
// The function to do Topological Sort. It uses recursive   
// topologicalSortUtil()   
void topologicalSort()   
{   
 stack < int > Stack  
   
 // Mark all the vertices as not visited   
 bool \*visited = new bool[V]  
 for (int i = 0; i < V; i++)   
 visited[i] = false  
   
 // Call the recursive helper function to store Topological   
 // Sort starting from all vertices one by one   
 for (int i = 0; i < V; i++)   
 if (visited[i] == false)   
 topologicalSortUtil(i, visited, Stack)  
   
 // Print contents of stack   
 // Topological Order  
 while (Stack.empty() == false)   
 {   
 print(Stack.top())   
 Stack.pop()  
 }   
}

**Problem #4 : Minimum time required to rot all oranges**

**Description -** Given a matrix of dimension m\*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

0: Empty cell  
1: Cells have fresh oranges  
2: Cells have rotten oranges

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [ i,j ] can rot other fresh orange at indexes [ i-1, j ], [ i+1, j ], [ i, j-1 ], [ i, j+1 ] (up, down, left and right). If it is impossible to rot every orange then simply return -1.  
  
**Examples**

Input: arr[][C] = { {2, 1, 0, 2, 1},  
 {1, 0, 1, 2, 1},  
 {1, 0, 0, 2, 1}};  
Output:  
All oranges can become rotten in 2 time frames.  
  
  
Input: arr[][C] = { {2, 1, 0, 2, 1},  
 {0, 0, 1, 2, 1},  
 {1, 0, 0, 2, 1}};  
Output:  
All oranges cannot be rotten.

**Solution -** The idea is to use Breadth First Search. Below is the algorithm.

1) Create an empty Q.  
2) Find all rotten oranges and enqueue them to Q. Also enqueue a delimiter to indicate the beginning of the next time frame.  
3) While Q is not empty do following  
....3.a) Do following while delimiter in Q is not reached  
........ (i) Dequeue an orange from the queue, rot all adjacent oranges. While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.  
....3.b) Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.

**Pseudo Code**

// This function finds if it is possible to rot all oranges or not.  
// If possible, then it returns minimum time required to rot all,  
// otherwise returns -1  
int rotOranges(int arr[][C])  
{  
 // Create a queue of cells  
 queue < cell > Q  
 cell temp  
 ans = 0  
 // Store all the cells having rotten orange in first time frame  
 for (int i=0; i < R; i++)  
 {  
 for (int j=0; j < C; j++)  
 {  
 if (arr[i][j] == 2)  
 {  
 temp.x = i  
 temp.y = j  
 Q.push(temp)  
 }  
 }  
 }  
 // Separate these rotten oranges from the oranges which will rotten  
 // due the oranges in first time frame using delimiter which is (-1, -1)  
 temp.x = -1  
 temp.y = -1  
 Q.push(temp)  
 // Process the grid while there are rotten oranges in the Queue  
 while (!Q.empty())  
 {  
 // This flag is used to determine whether even a single fresh  
 // orange gets rotten due to rotten oranges in current time  
 // frame so we can increase the count of the required time.  
 bool flag = false  
 // Process all the rotten oranges in current time frame.  
 while (!isdelim(Q.front()))  
 {  
 temp = Q.front()  
 // Check right adjacent cell that if it can be rotten  
 if (isvalid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)  
 {  
 // if this is the first orange to get rotten, increase  
 // count and set the flag.  
 if (!flag) ans++, flag = true  
  
 // Make the orange rotten  
 arr[temp.x+1][temp.y] = 2  
  
 // push the adjacent orange to Queue  
 temp.x++  
 Q.push(temp)  
  
 temp.x-- // Move back to current cell  
 }  
 // Check left adjacent cell that if it can be rotten  
 if (isvalid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {  
 if (!flag) ans++, flag = true  
 arr[temp.x-1][temp.y] = 2  
 temp.x--  
 Q.push(temp) // push this cell to Queue  
 temp.x++  
 }  
 // Check top adjacent cell that if it can be rotten  
 if (isvalid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {  
 if (!flag) ans++, flag = true  
 arr[temp.x][temp.y+1] = 2  
 temp.y++  
 Q.push(temp) // Push this cell to Queue  
 temp.y--  
 }  
 // Check bottom adjacent cell if it can be rotten  
 if (isvalid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {  
 if (!flag) ans++, flag = true  
 arr[temp.x][temp.y-1] = 2  
 temp.y--  
 Q.push(temp) // push this cell to Queue  
 }  
 Q.pop()  
 }  
 // Pop the delimiter  
 Q.pop()  
 // If oranges were rotten in current frame than separate the  
 // rotten oranges using delimiter for the next frame for processing.  
 if (!Q.empty()) {  
 temp.x = -1  
 temp.y = -1  
 Q.push(temp)  
 }  
 // If Queue was empty than no rotten oranges left to process so exit  
 }  
 // Return -1 if all arranges could not rot, otherwise -1.  
 return (checkall(arr))? -1: ans  
}

**Topic 9 : Prim’s Minimum Spanning Tree Algorithm**

**What is the Minimum Spanning Tree?**

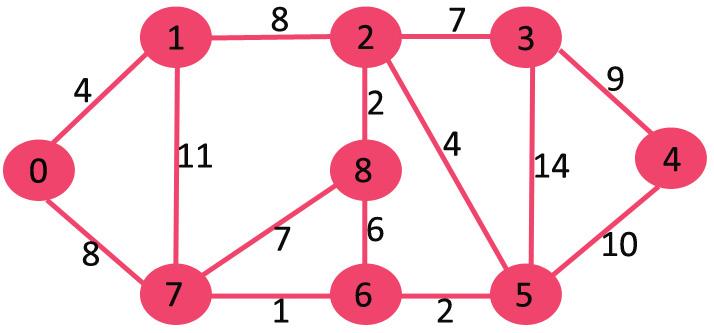
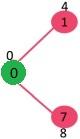
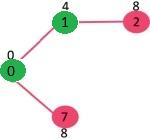
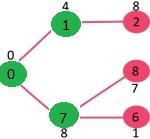
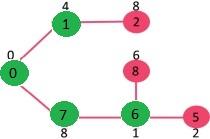
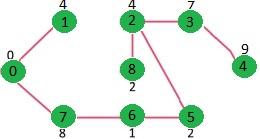
Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree (MST)** or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.  
  
**Number of edges in a minimum spanning tree:** A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

**Prim's Algorithm**

Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.  
  
A group of edges that connects two sets of vertices in a graph is called cut in graph theory. *So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*  
***How does Prim's Algorithm Work?*** The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning*Tree. And they must be connected with the minimum weight edge to make it a *Minimum*Spanning Tree.  
  
***Algorithm***:

1. Create a set *mstSet* that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

1. While mstSet doesn't include all vertices:  
   * Pick a vertex *u* which is not there in *mstSet*and has minimum key value.
   * Include *u*to mstSet.
   * Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.  
  
Let us understand this with the help of following example:  
  
The set *mstSet*is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, including it in *mstSet*. So *mstSet*becomes {0}. After adding to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.  
  
  
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.  
  
  
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 be picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).  
  
Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key values of vertex 5 and 8 are updated.  
  
  
We repeat the above steps until *mstSet*includes all vertices of the given graph. Finally, we get the following graph.  
  


***How to implement the above algorithm?***

We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(int key[], bool mstSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the

// constructed MST stored in parent[]

int printMST(int parent[], int graph[V][V])

{

cout<<"Edge \tWeight\n";

for (int i = 1; i < V; i++)

Run

Java

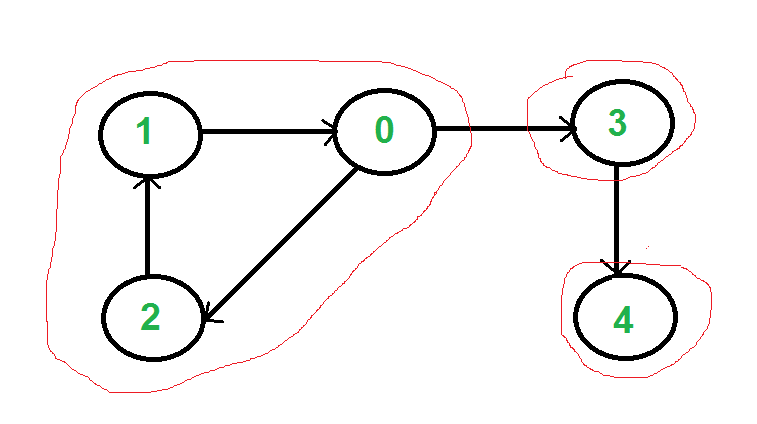


**Output**:

Edge Weight  
0 - 1 2  
1 - 2 3  
0 - 3 6  
1 - 4 5

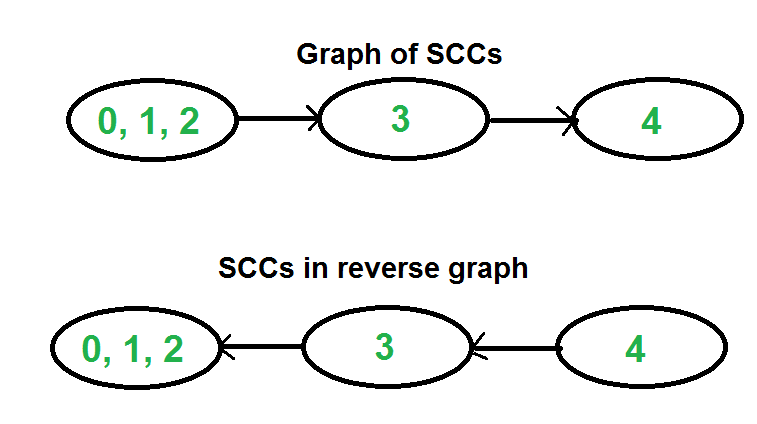
**Time Complexity** of the above program is O(V^2). If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.

**Topic 10 : Kosaraju’s Algorithm | Strongly Connected Components**

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.  
  
  
We can find all strongly connected components in O(V+E) time using Kosaraju’s algorithm. Following is Kosaraju's algorithm.

1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in the stack as 1, 2, 4, 3, 0.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as a source and do DFS . The DFS starting from v prints strongly connected components of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

**How does this work?**

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise, DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start at 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sunk in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way of getting this sequence. However, if we do a DFS of graphs and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC . For example, in the DFS of above example graph, the finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And the finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example, finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of the complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appears after both 3 and 4.  
  
In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in the stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using a sequence of vertices in the stack, we process vertices from the sink to the source (in the reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.  
  
  
Below is the implementation of Kosaraju's algorithm:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ Implementation of Kosaraju's algorithm to print all SCCs

#include <iostream>

#include <list>

#include <stack>

using namespace std;

class Graph

{

int V; // No. of vertices

list<int> \*adj; // An array of adjacency lists

// Fills Stack with vertices (in increasing order of finishing

// times). The top element of stack has the maximum finishing

// time

void fillOrder(int v, bool visited[], stack<int> &Stack);

// A recursive function to print DFS starting from v

void DFSUtil(int v, bool visited[]);

public:

Graph(int V);

void addEdge(int v, int w);

// The main function that finds and prints strongly connected

// components

void printSCCs();

// Function that returns reverse (or transpose) of this graph

Graph getTranspose();

};

Run

Java



**Output**:

Following are strongly connected components in given graph

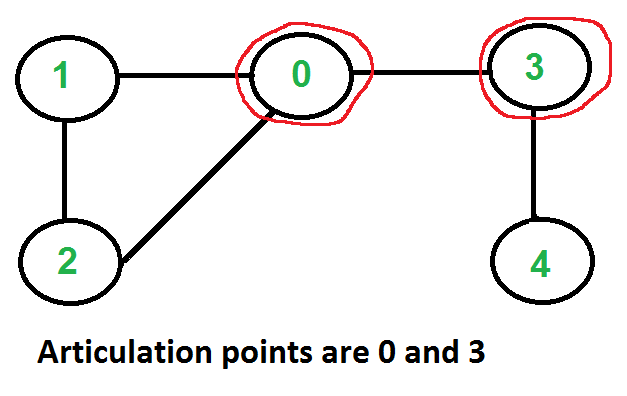
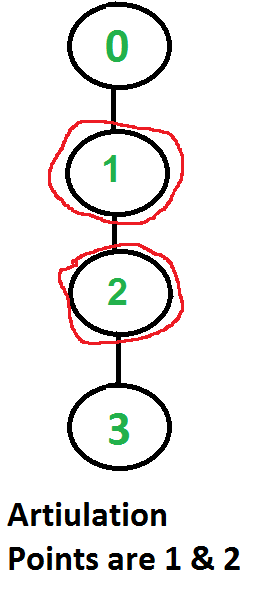
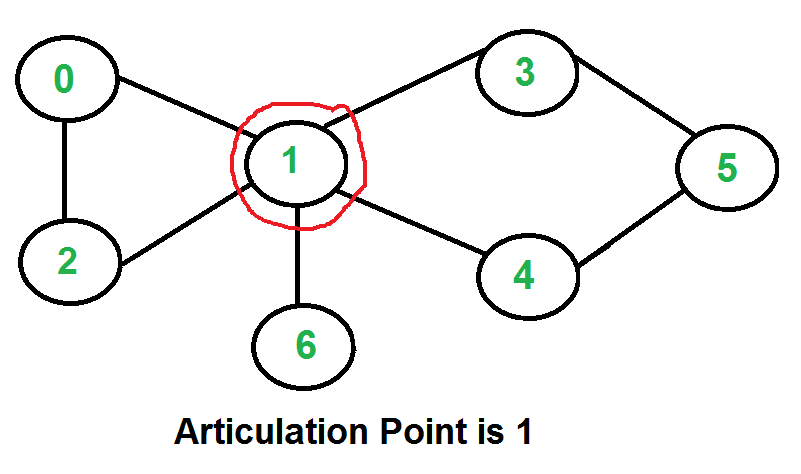
0 1 2

3

4

**Time Complexity:**The above algorithm calls DFS, find reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simply traverse all adjacency lists.  
  
**Applications:** SCC algorithms can be used as the first step in many graph algorithms that work only on a strongly connected graph. In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked a commonly liked page or played a game.

**Topic 11 : Articulation Points (or Cut Vehicles ) in a Graph**

A vertex in an undirected connected graph is an articulation point (or cut vertex) if and only if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.  
  
For a disconnected undirected graph, an articulation point is a vertex removing which increases the number of connected components.  
  
Following are some example graphs with articulation points encircled with red color.  
  
  


**How to find all articulation points in a given graph?**

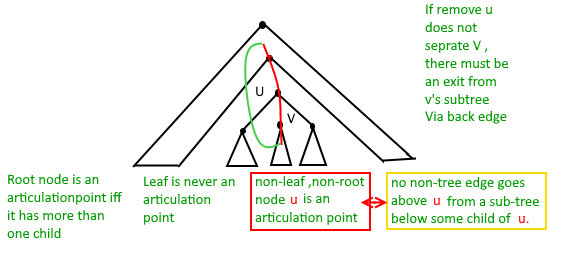
A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graphs. Following are steps of a simple approach for the connected graph.

1. For every vertex v, do following:  
   * Remove v from graph
   * See if the graph remains connected (We can either use BFS or DFS)
   * Add v back to the graph

The **time complexity** of the above method is O(V\*(V+E)) for a graph represented using adjacency list. Can we do better?  
  
**A O(V+E) algorithm to find all Articulation Points (APs)**. The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS trees. In a DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is adjacent of u in the graph). In a DFS tree, a vertex u is an articulation point if one of the following two conditions is true.

1. u is the root of the DFS tree and it has at least two children.

1. u is not the root of DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

The following figure shows the same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.  
  
  
  
We do DFS traversal of the given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.  
  
How to handle the second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

**low[u] = min(disc[u], disc[w])**   
where w is an ancestor of u and there is a back edge from   
some descendant of u to w.

Following are C++ and Java implementation of Tarjan's algorithm for finding articulation points:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ program to find articulation points

// in an undirected graph

#include<iostream>

#include <list>

#define NIL -1

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

void APUtil(int v, bool visited[], int disc[], int low[],

int parent[], bool ap[]);

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // function to add an edge to graph

void AP(); // prints articulation points

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

Run

Java

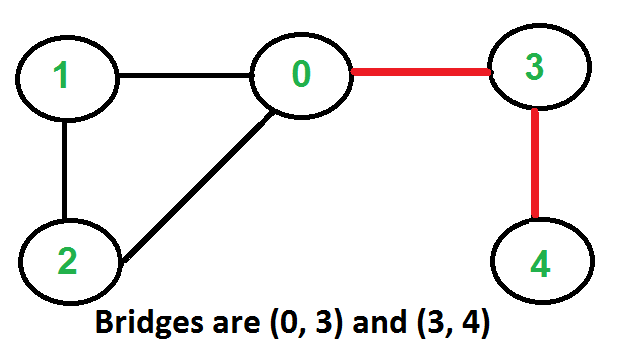
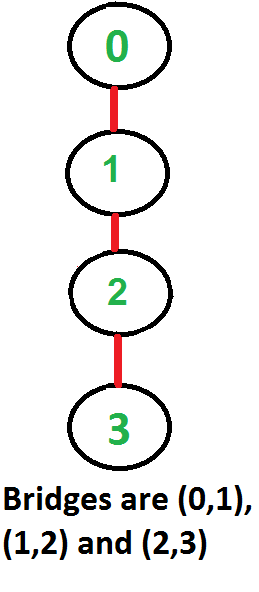
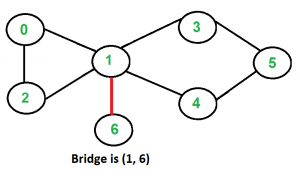


**Output**:

Articulation points in first graph  
0 3  
Articulation points in second graph  
1 2  
Articulation points in third graph  
1

**Time Complexity:** The above function is simple DFS with additional arrays. So time complexity is the same as DFS which is O(V+E) for adjacency list representation of the graph.

**Topic 12 : Bridges in a Graph**

An edge in an undirected connected graph is a bridge *if and only if* removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removing which increases the number of disconnected components.  
  
Like ***Articulation Points***, bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.  
  
Following are some example graphs with bridges highlighted with red colour:  
  
  


**How to find all bridges in a given graph?**

A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of a simple approach for a connected graph.

1. For every edge (u, v), do following  
   * Remove (u, v) from graph.
   * See if the graph remains connected (We can either use BFS or DFS)
   * Add (u, v) back to the graph.

The**time complexity**of the above method is O(E\*(V+E)) for a graph represented using adjacency list. Can we do better?  
  
**A O(V+E) algorithm to find all Bridges** is similar to that of O(V+E) algorithm for Articulation Points. We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the previous post, the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, "low[v] > disc[u]"*.  
  
Following are C++ and Java implementations of the above approach:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ program to find bridges in a given undirected graph

#include<iostream>

#include <list>

#define NIL -1

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

void bridgeUtil(int v, bool visited[], int disc[], int low[],

int parent[]);

public:

Graph(int V); // Constructor

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

void bridge(); // prints all bridges

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

Run

Java



**Output**:

Bridges in first graph

3 4

0 3

Bridges in second graph

2 3

1 2

0 1

Bridges in third graph

1 6

**Time Complexity:** The above function is simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.