# Data Structures and Algorithms

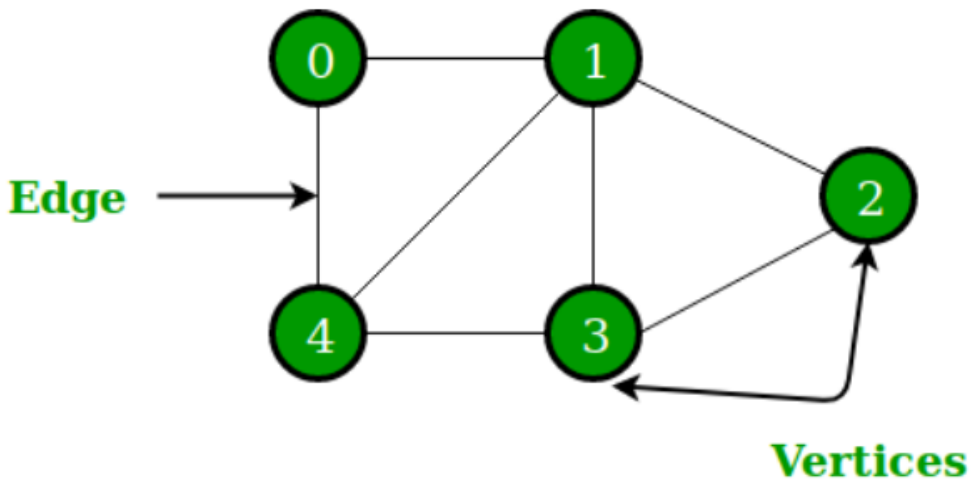
Trong-Hop Do

University of Information Technology, HCM city

# Graph

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

> *A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.*




set of vertices V = {0,1,2,3,4}
set of edges E = {01, 12, 23, 34, 04, 14, 13}.

# Graph representations

The following two are the most commonly used representations of a graph.

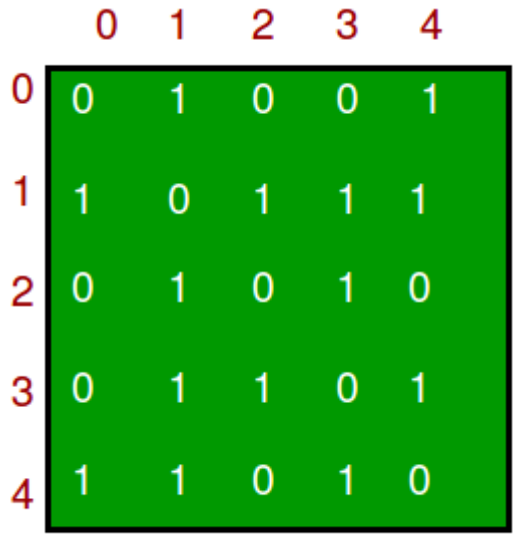
1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

# Adjacency Matrix

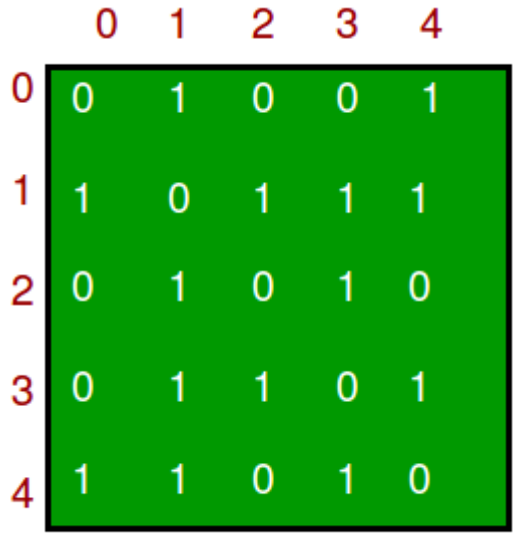
- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

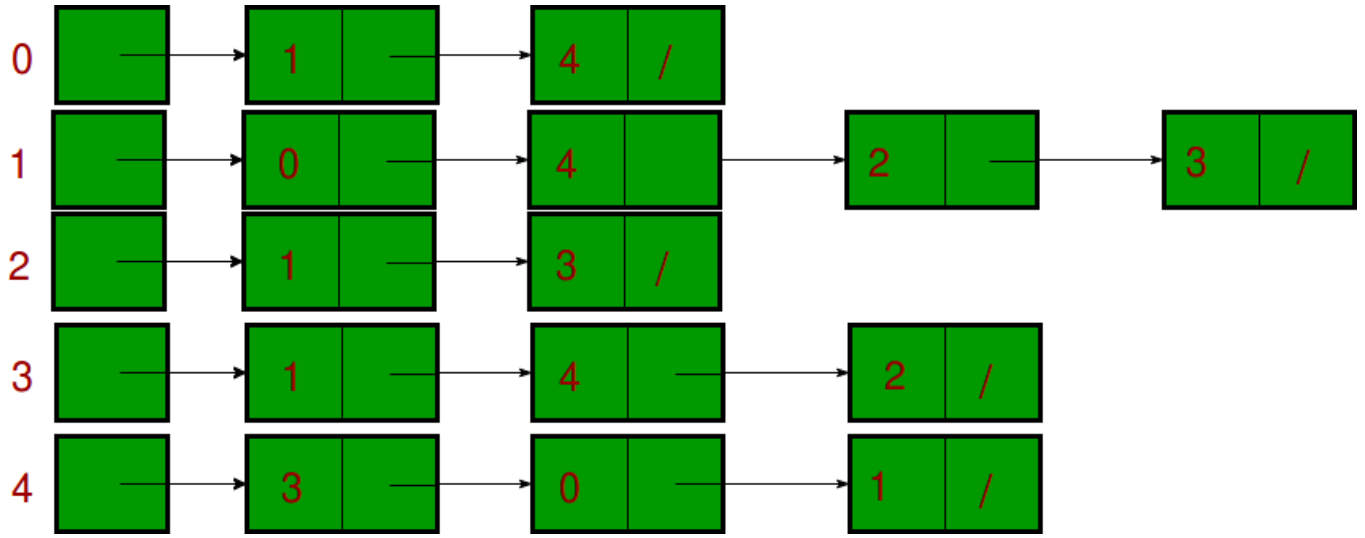
# Adjacency Matrix

- 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.



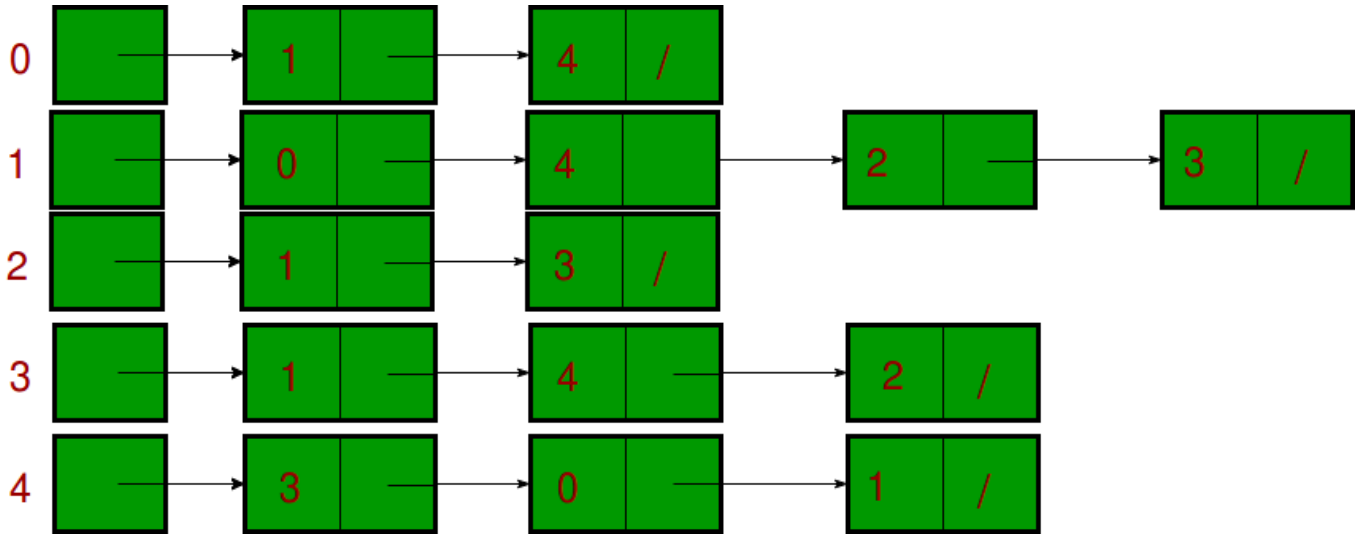
# Adjacency List:

- An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.



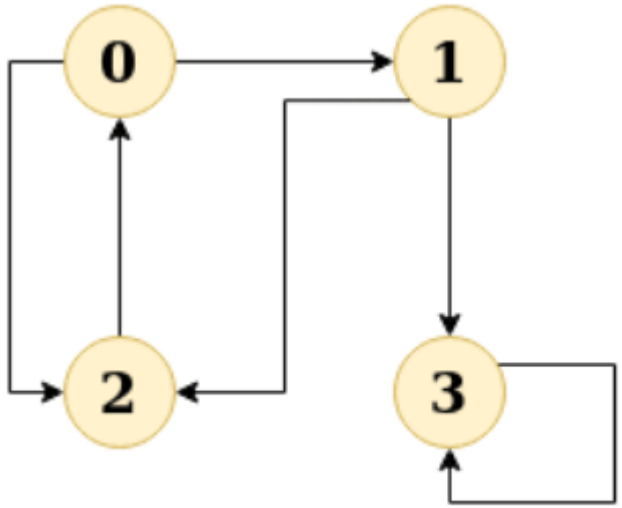
# Adjacency List:

- Pros: Saves space O(|V|+|E|) . In the worst case, there can be 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).



# Adjacency List:

Edge list: 2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3



# Depth First Search

**Approach**:

- Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

**Algorithm**:

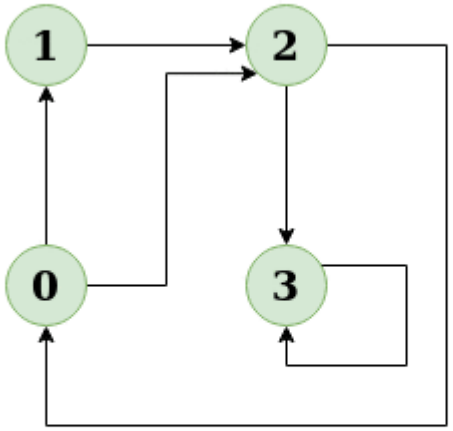
- Create a recursive function that takes the index of node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

# Depth First Search

***Input:*** *n = 4, e = 6*

*0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3*

***Output:*** *DFS from vertex 1 : 1 2 0 3*

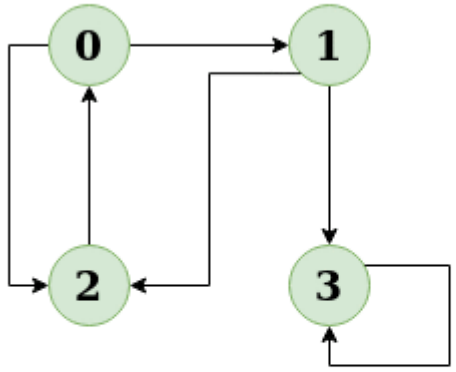



Green is unvisited node.
Red is current node.
Orange is the nodes in the recursion stack.

***Input:*** *n = 4, e = 6*

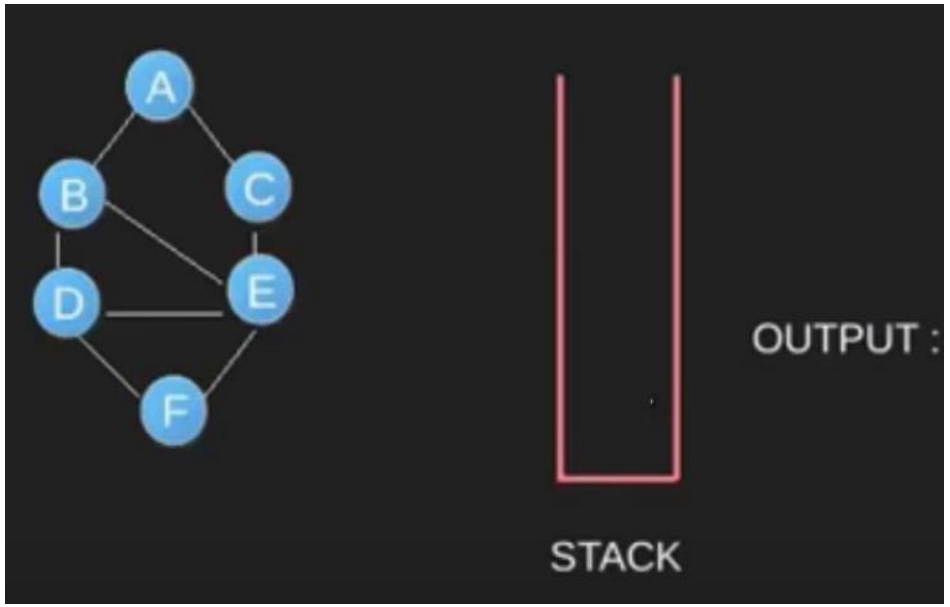
*2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3*

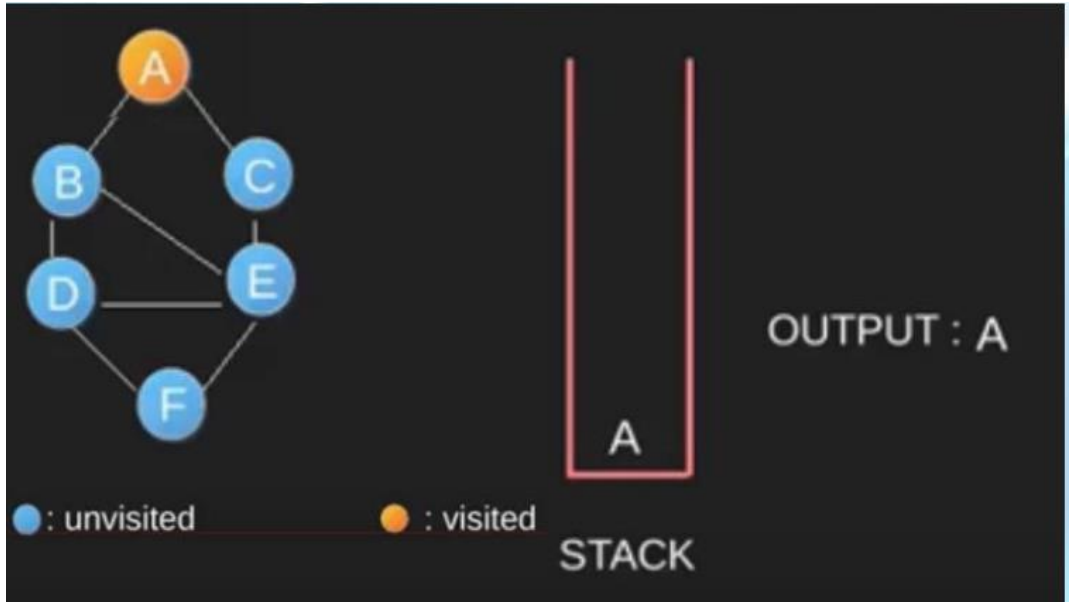
***Output:*** *DFS from vertex 2 : 2 0 1 3*



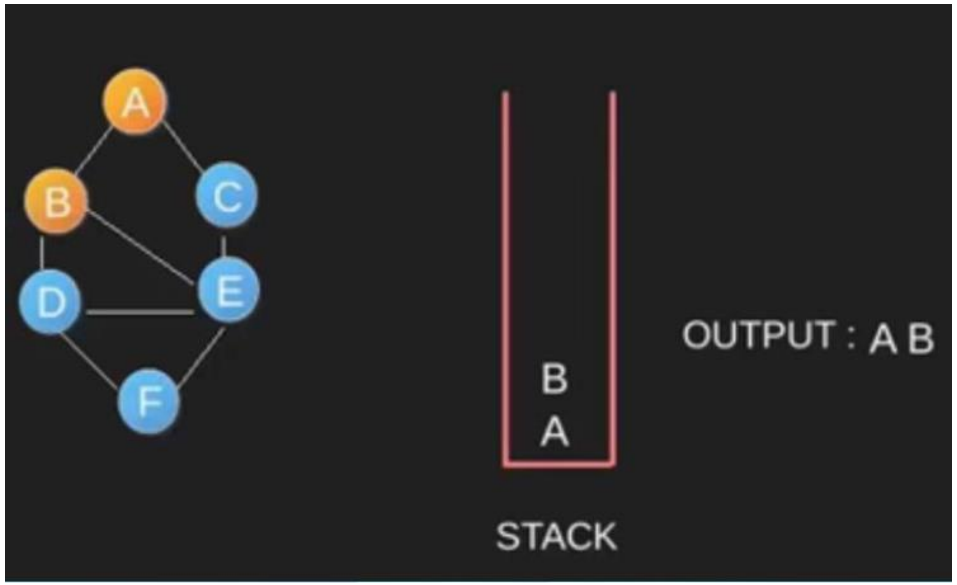

Green is unvisited node.
Red is current node.
Orange is the nodes in the recursion stack.

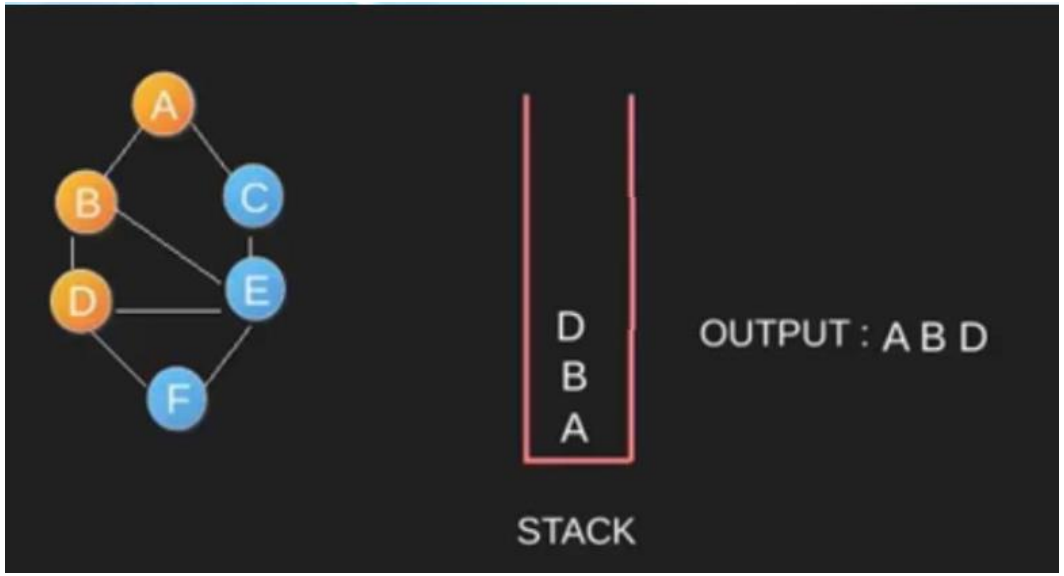
# Depth First Search



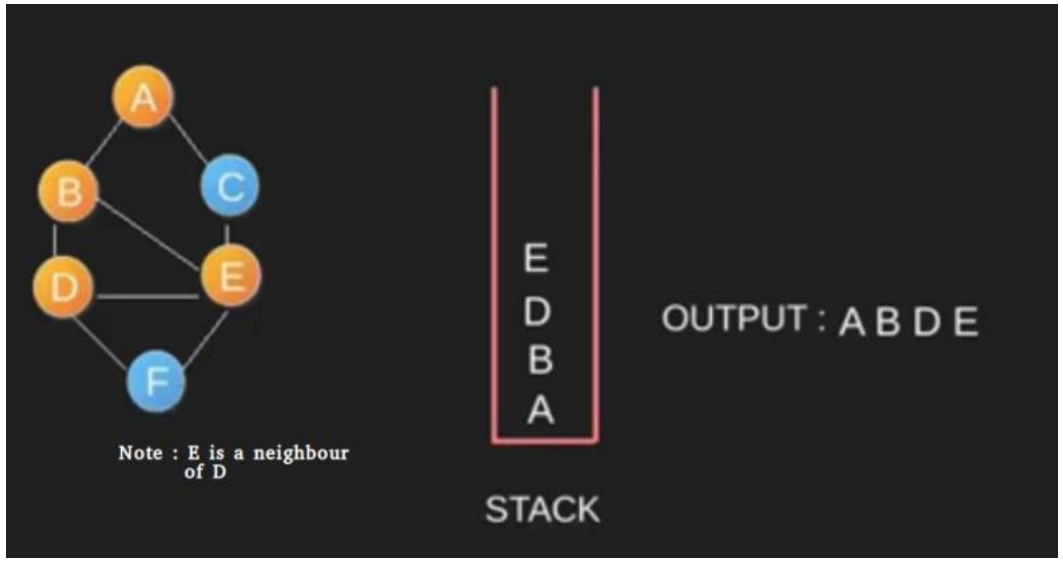



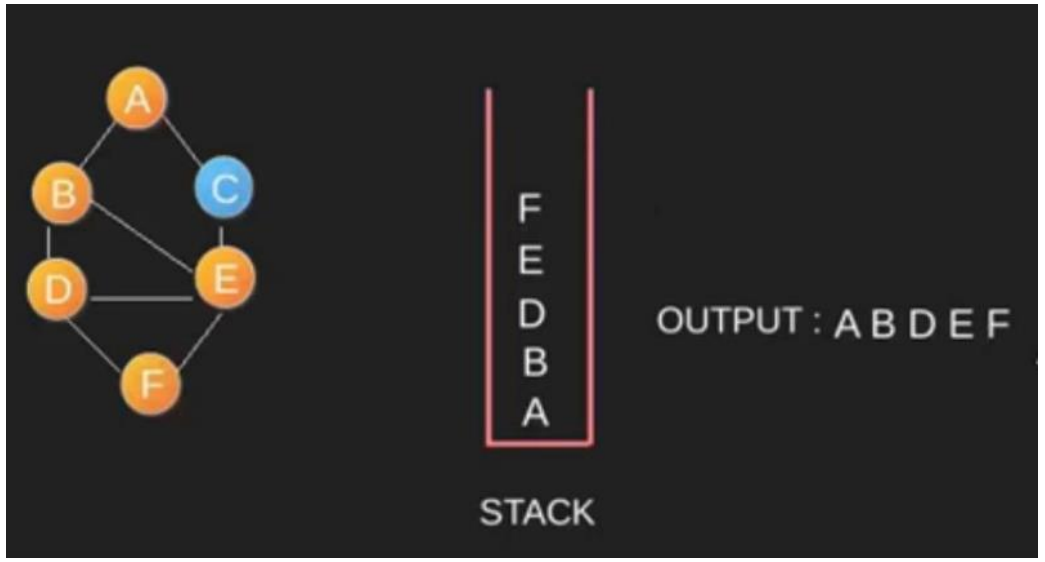
# Depth First Search



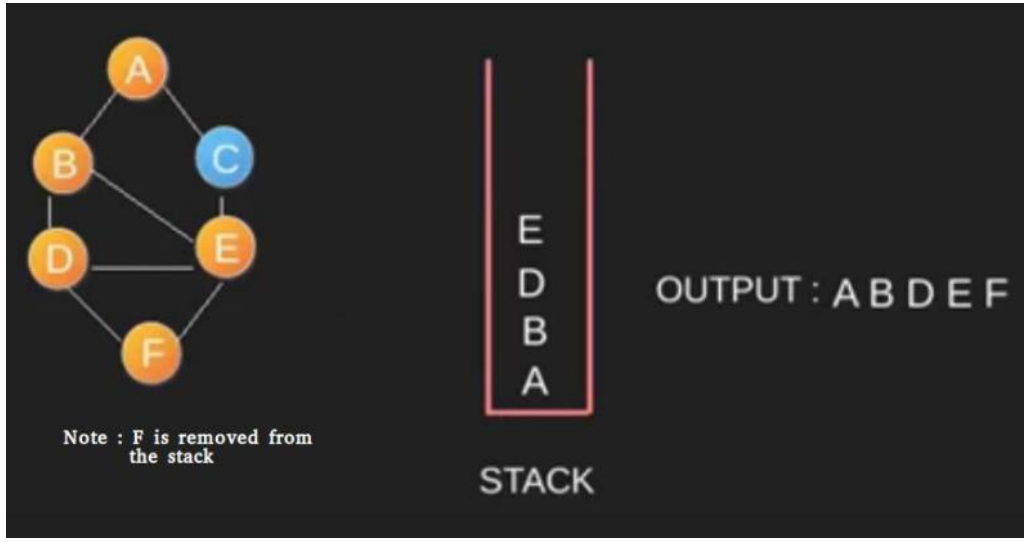



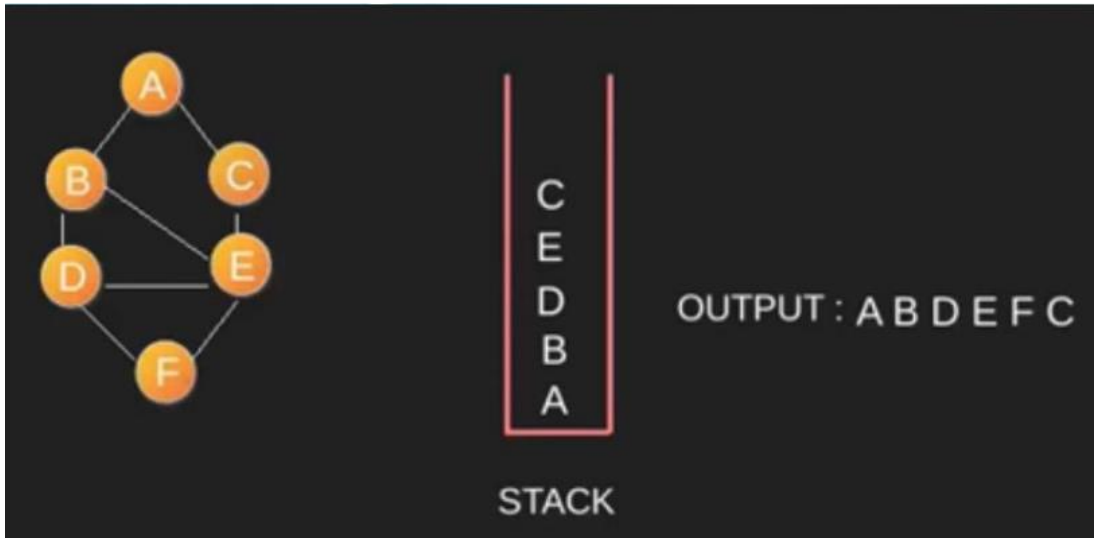
# Depth First Search



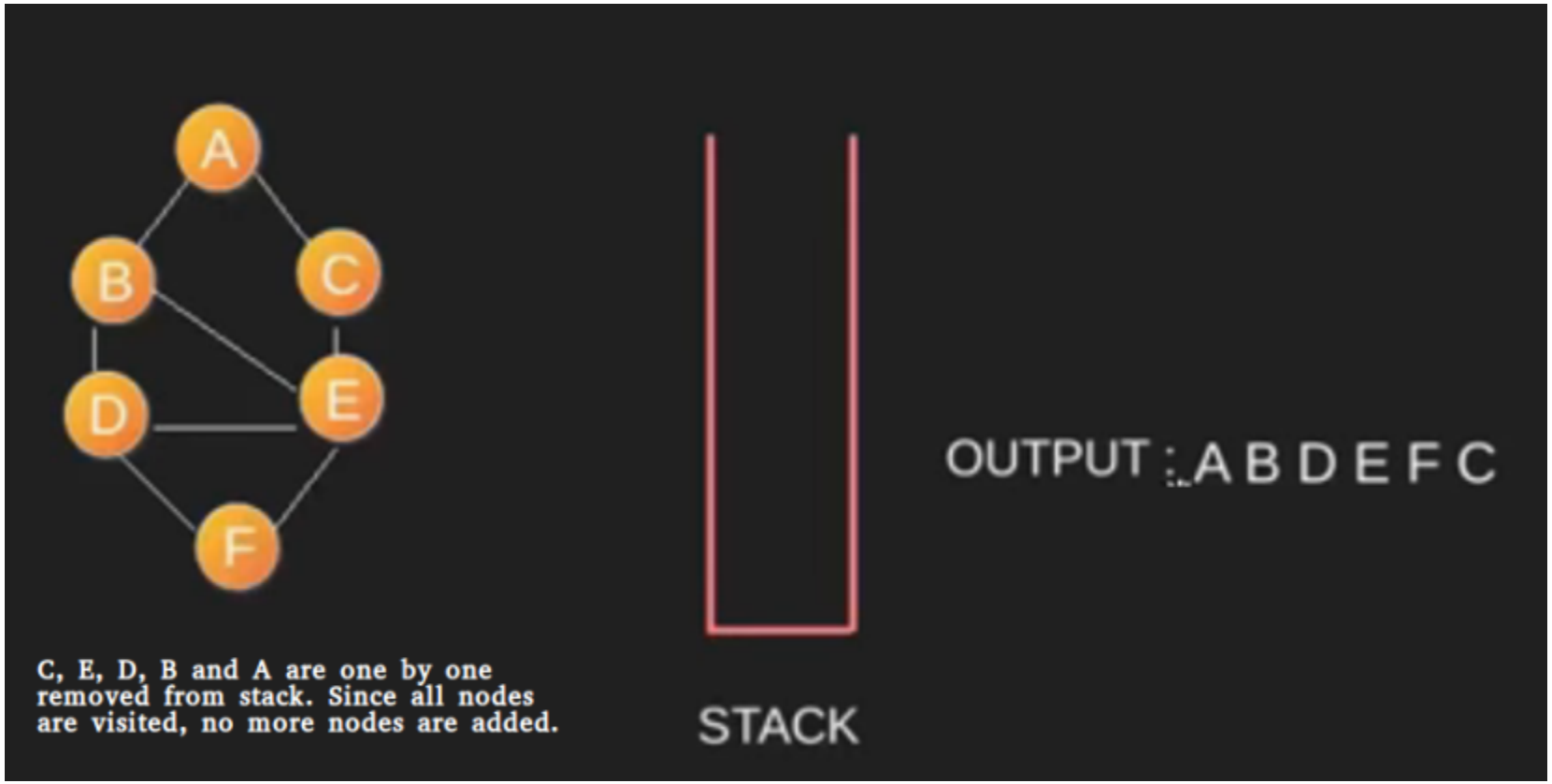



# Depth First Search



# Depth First Search



# Depth First Search

- Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- Space Complexity: O(V).
- Since, an extra visited array is needed of size V.

# Depth First Search (Handling Disconnected Graph )

**Solution**:

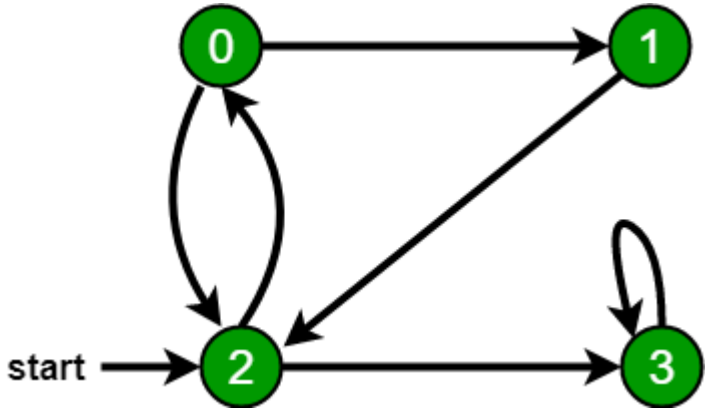
- The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex as in the case of a Disconnected graph. To do complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS.
- The recursive function remains the same.

**Algorithm**:

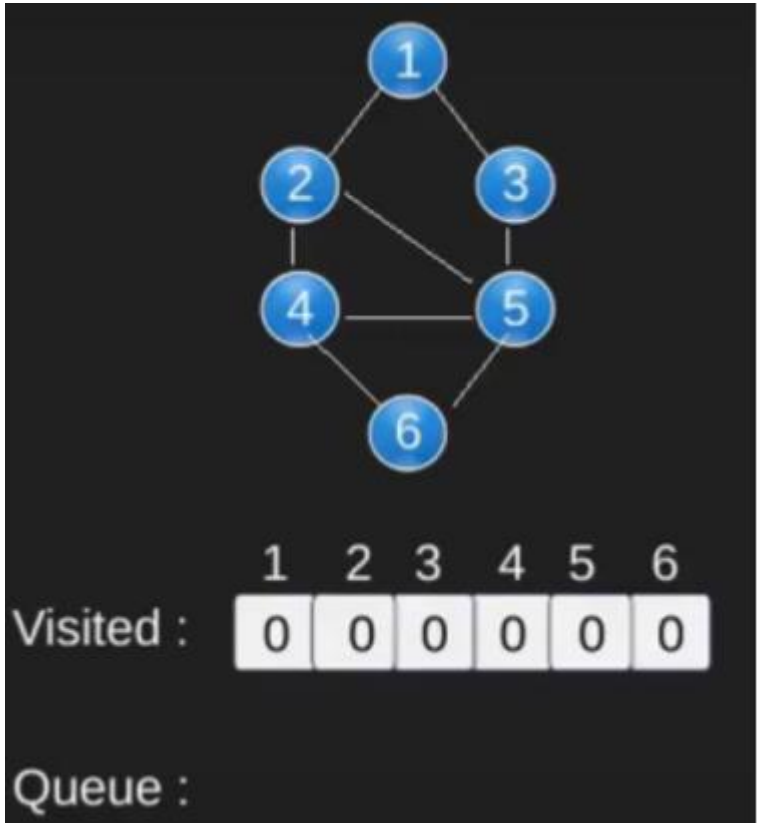
- Create a recursive function that takes the index of node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.
- Run a loop from 0 to number of vertices and check if the node is unvisited in previous DFS then call the recursive function with current node.
- Implementation:

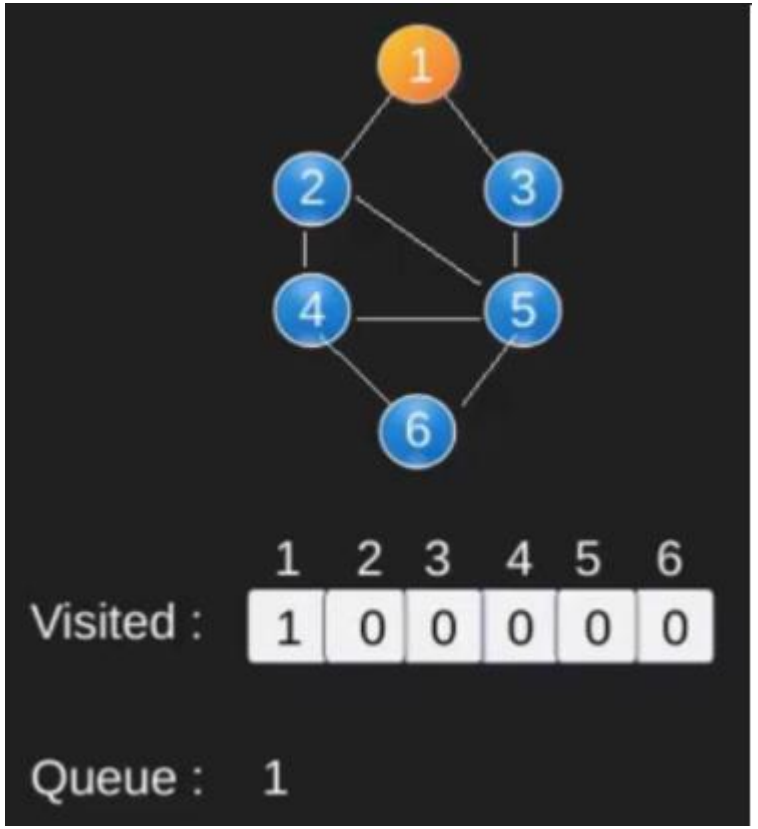
# Breadth First Search

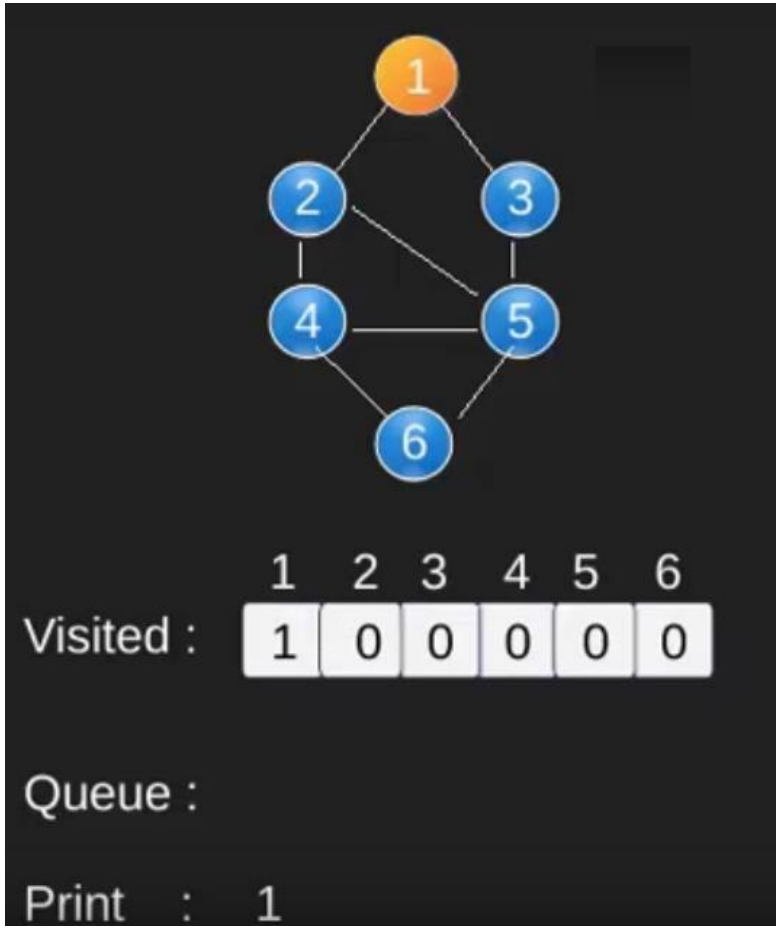
- Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree . The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
- A Breadth First Traversal of the following graph is 2, 0, 3, 1.



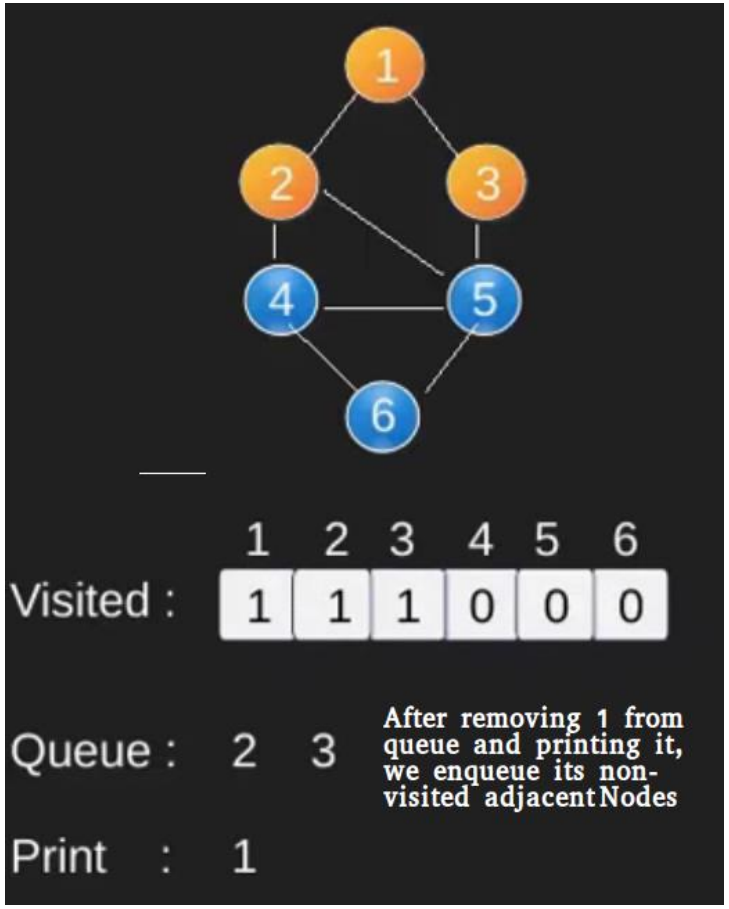
# Breadth First Search

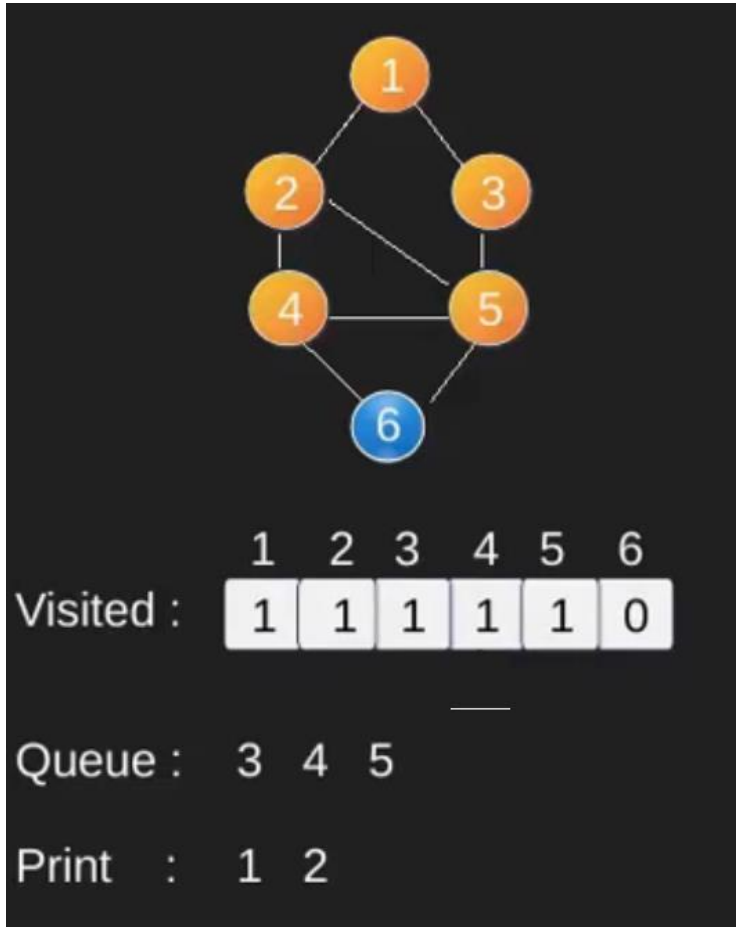


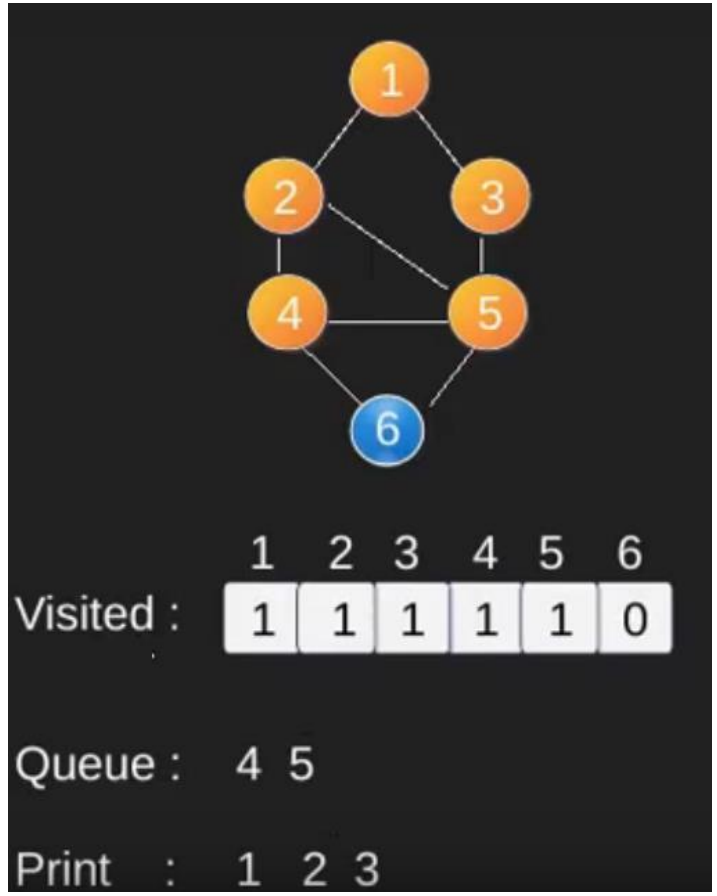



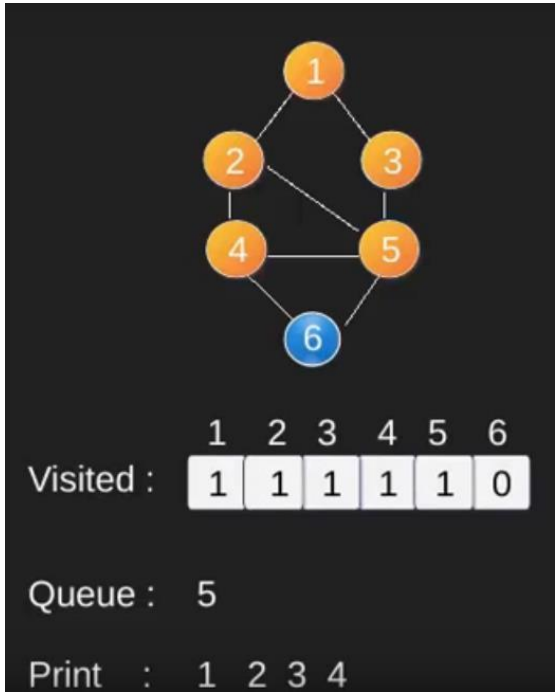
# Breadth First Search

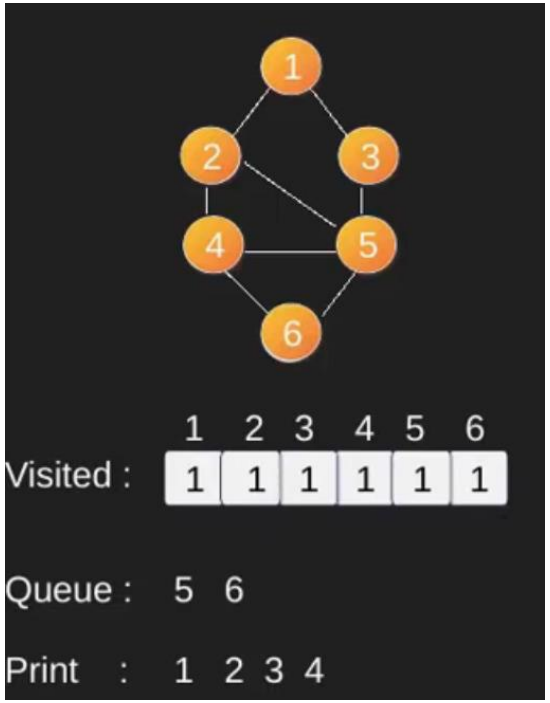


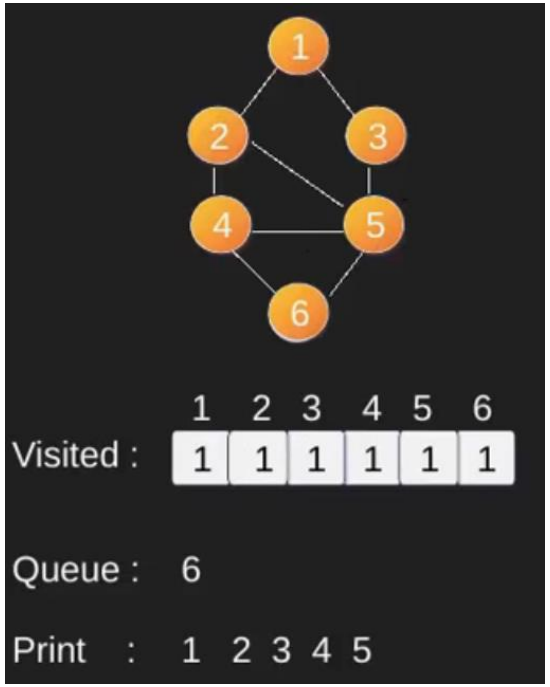



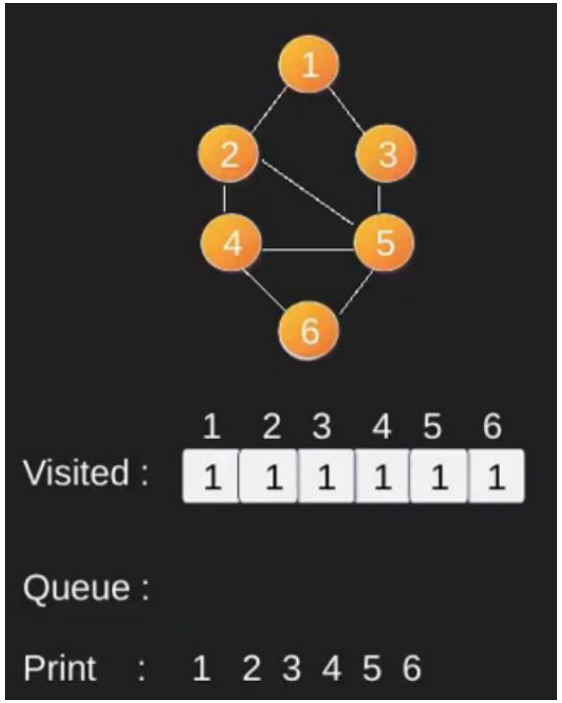



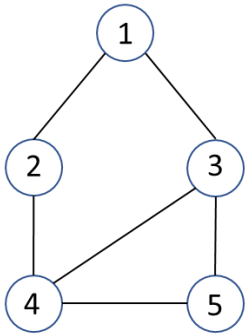
# Breadth First Search






Step 1: enqueue 1

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 0 | 0 | 0 | 0 |

Queue: 1

Print:

Step 2: dequeue 1, print 1, enqueue 2 and 3

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 1 | 1 | 0 | 0 |

Queue: 2, 3

Print: 1

Step 3: dequeue 2, print 2, enqueue 4

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 1 | 1 | 1 | 0 |

Queue: 3, 4

Print: 1, 2

Step 4: dequeue 3, print 3, enqueue 5

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 1 | 1 | 1 | 1 |

Queue: 4, 5

Print: 1, 2, 3

Step 5: dequeue 4, print 4, enqueue nothing

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 1 | 1 | 1 | 1 |

Queue: 5

Print: 1, 2, 3, 4

Step 6: dequeue 5, print 5, enqueue nothing

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Visited: | 1 | 1 | 1 | 1 | 1 |

Queue:

Print: 1, 2, 3, 4, 5

# Dijkstra’s shortest path algorithm

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 the source is calculated and finalized. Initially, this set is empty.

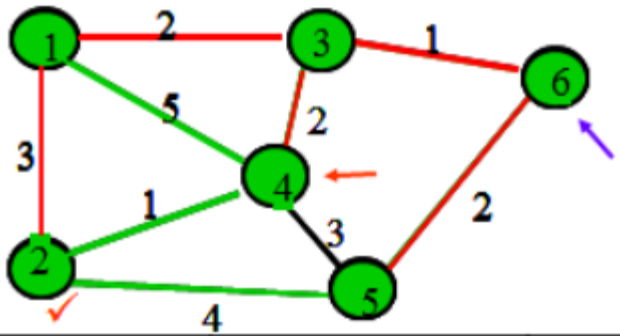
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

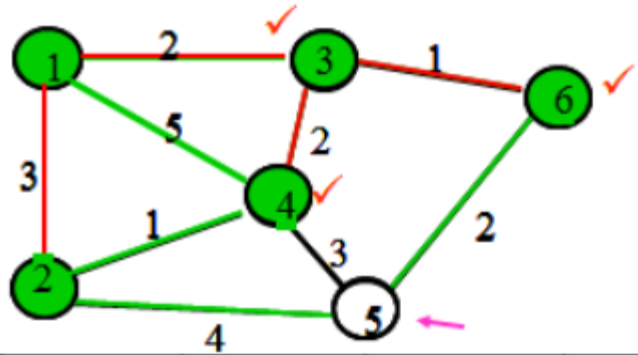
**3)** While *sptSet* doesn’t include all vertices

....**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.

....**b)** Include u to *sptSet*.

....**c)** 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.

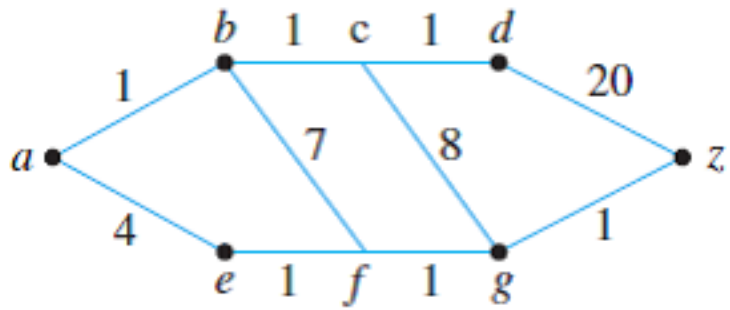



| Iteration | N | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ |
|---|---|---|---|---|---|---|
| Initial | {1} | 3 | 2 | 5 | ∞ | ∞ |
| 1 | {1,3} | 3 | 2 | 4 | ∞ | 3 |
| 2 | {1,2,3} | 3 | 2 | 4 | 7 | 3 |
| 3 | {1,2,3,6} | 3 | 2 | 4 | 5 | 3 |
| 4 | {1,2,3,4,6} | 3 | 2 | 4 | 5 | 3 |
| 5 | {1,2,3,4,5,6} | 3 | 2 | 4 | 5 | 3 |

# Dijkstra’s shortest path algorithm

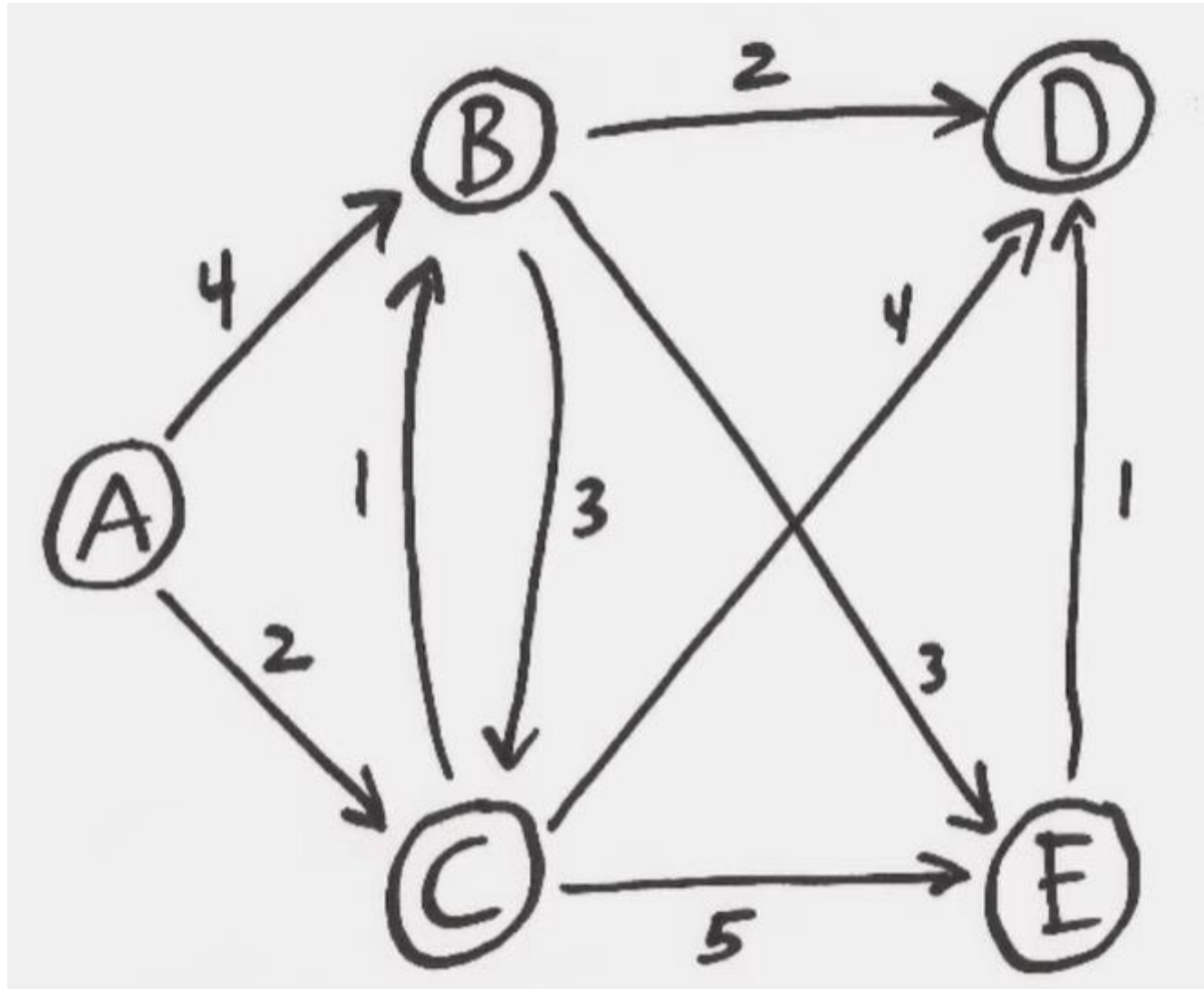
```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
 9      dist[source] ← 0
10
11      while Q is not empty:
12          u ← vertex in Q with min dist[u]
13
14          remove u from Q
15
16          for each neighbor v of u still in Q:
17              alt ← dist[u] + length(u, v)
18              if alt < dist[v]:
19                  dist[v] ← alt
20                  prev[v] ← u
21
22      return dist[], prev[]
```




Reference:

**Table**

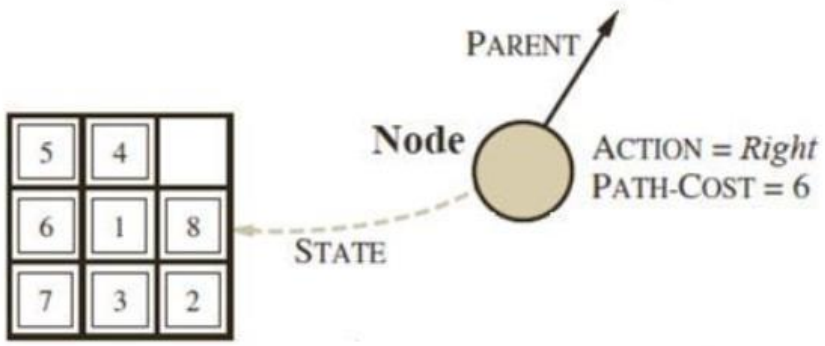
| Step | $V(T)$ | $E(T)$ | $F$ | $L(a)$ |
|---|---|---|---|---|
| 0 | $\{a\}$ | $\emptyset$ | $\{a\}$ | 0 |
| 1 | $\{a\}$ | $\emptyset$ | $\{b,\ c\}$ | 0 |
| 2 | $\{a, b\}$ | $\{\{a, b\}\}$ | $\{c, d, e\}$ | 0 |
| 3 | $\{a, b, c\}$ | $\{\{a, b\},\ \{a, c\}\}$ | $\{d, e\}$ | 0 |
| 4 | $\{a, b, c, e\}$ | $\{\{a, b\},\ \{a, c\},\ \{c, e\}\}$ | $\{d, z\}$ | 0 |
| 5 | $\{a, b, c, e, d\}$ | $\{\{a, b\},\ \{a, c\},\ \{c, e\},\ \{e, d\}\}$ | $\{z\}$ | 0 |
| 6 | $\{a, b, c, e, d, z\}$ | $\{\{a, b\},\ \{a, c\},\ \{c, e\},\ \{e, d\},\ \{e, z\}\}$ | | |


A
B
C
D
E
4
2
1
3
2
4
3
1
5

# Uniform cost search

## Infrastructure for Search Algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- Each node n of the tree contains four components:
  n.STATE: the state in the state space to which the node corresponds;
  n.PARENT: the node in the search tree that generated this node;
  n.ACTION: the action that was applied to the parent to generate the node;
  n.PATH-COST: the cost of the path from the initial state to the node,



```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

```
procedure uniform_cost_search(Graph, start, goal) is
    node ← start
    cost ← 0
    frontier ← priority queue containing node only
    explored ← empty set
    do
        if frontier is empty then
            return failure
        node ← frontier.pop()
        if node is goal then
            return solution
        explored.add(node)
        for each of node's neighbors n do
            if n is not in explored then
                frontier.add(n)
```

# Uniform-Cost Search for Travelling in Romania