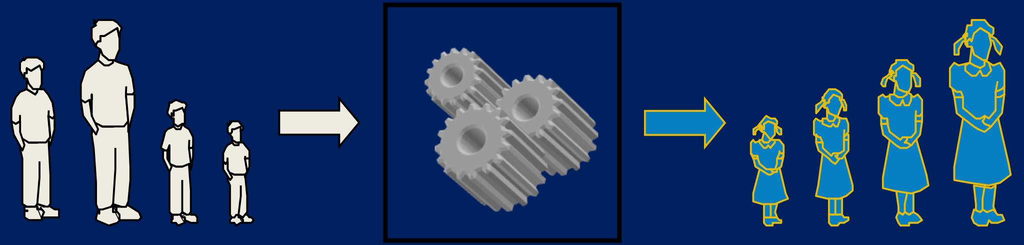
GLA UNIVERSITY MATHURA

Recognised by UGC Under Section 2(f)

Accredited with A Grade by NAAC

DESIGN & ANALYSIS OF ALGORITHM (BCSC0012)

Chapter 11: Graph Traversal



Input Algorithm Output

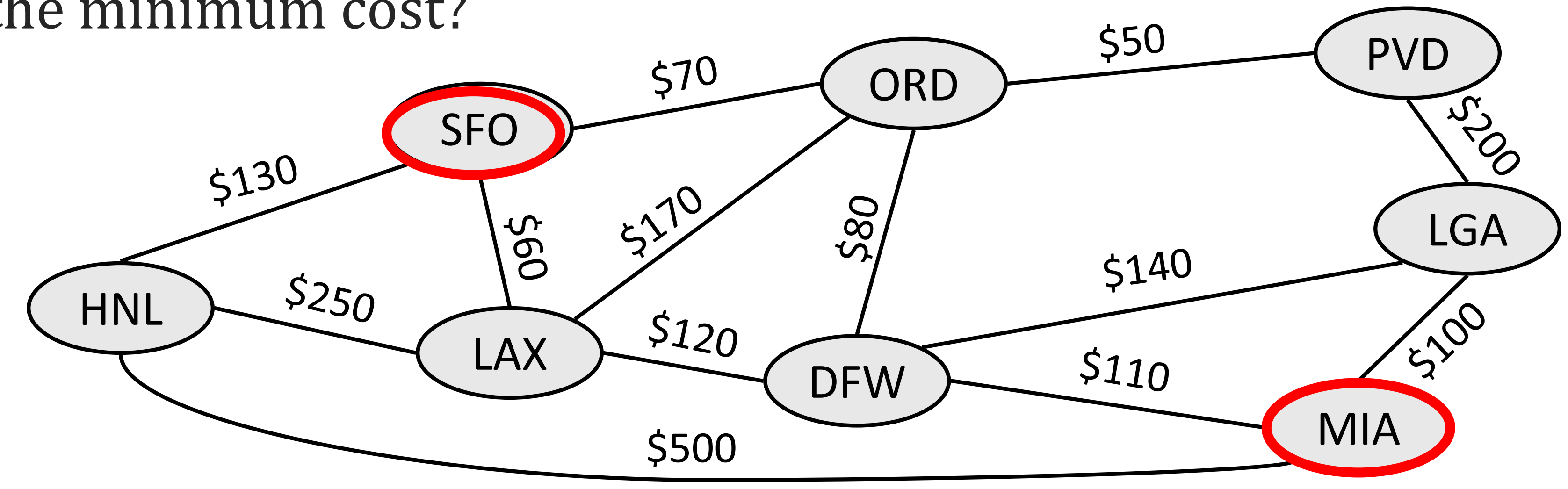
Prof. Anand Singh Jalal

Department of Computer Engineering & Applications

Searching for Paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).

- What is the shortest path from MIA to SFO?
  Which path has the minimum cost?



Graph Traversal

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.

The order in which the vertices are visited are important and may depend upon the algorithm.

There are two ways to traverse a Graph

1. BFS (Breadth First Search)
2. DFS (Depth First Search)

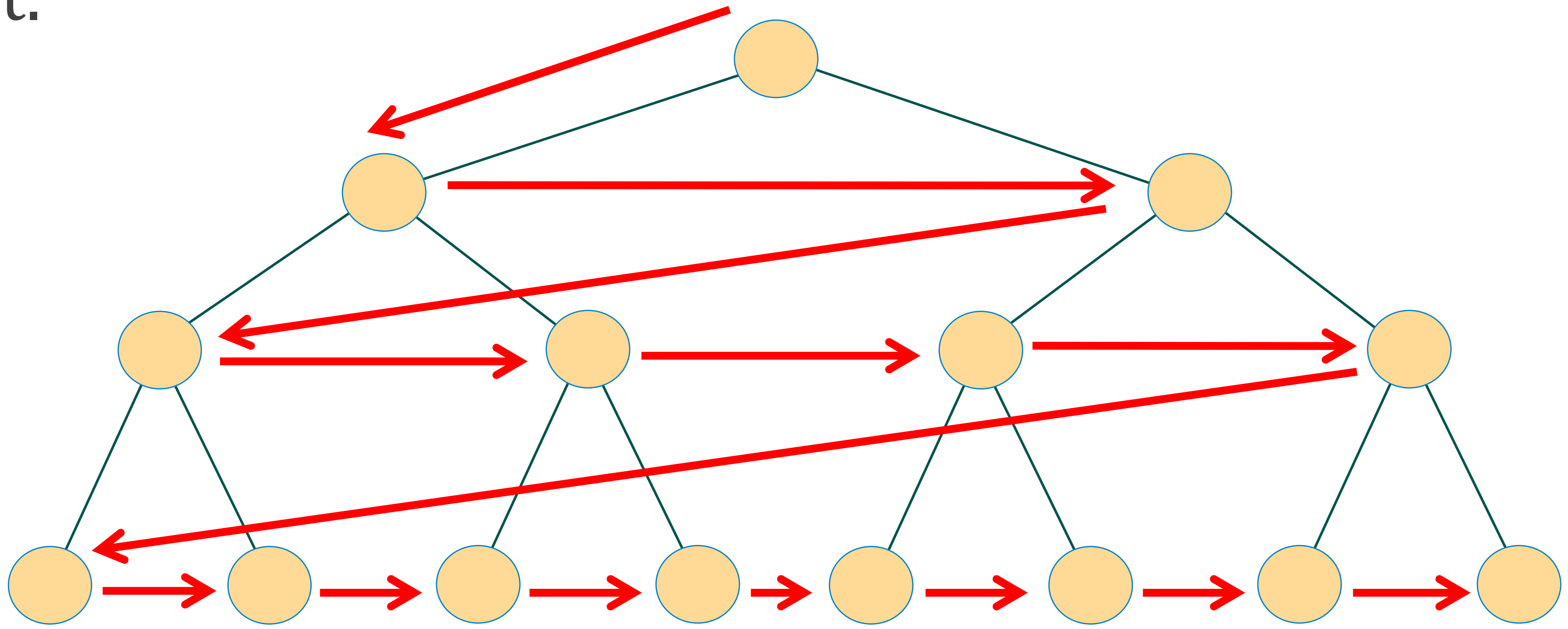
GLA UNIVERSITY MATHURA Recognised by UGC Under Section 2(f) Accredited with A Grade by NAAC

Breadth First Search(BFS)

Breadth-first search (BFS): Finds a path between two nodes by taking one step down all paths and then immediately backtracking.

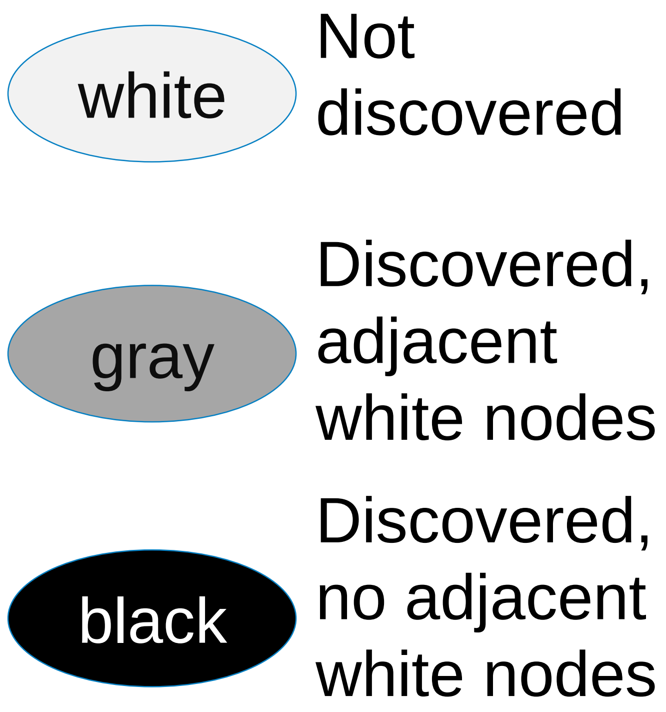
It is often implemented by maintaining a **queue** (FIFO Structure) of vertices to visit.

In BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.



Breadth First Search(BFS) ...

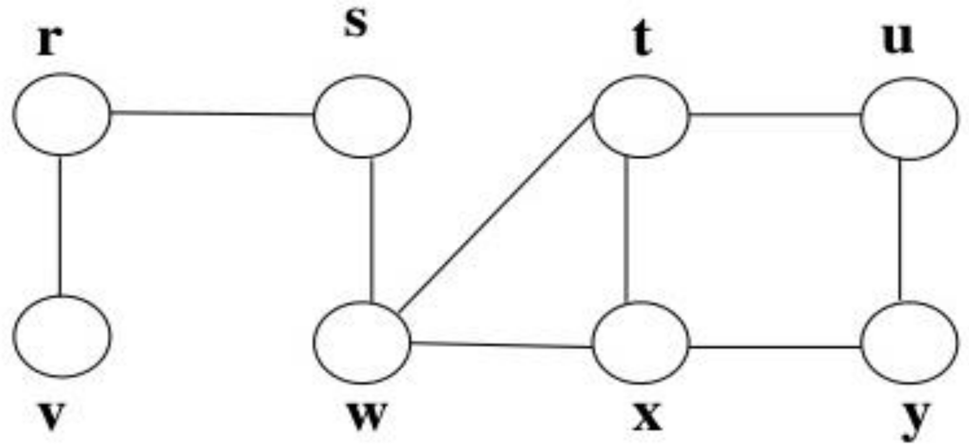
- Algorithm maintains a queue Q to manage set of vertices and start with s, source vertex
- Initially, all vertices except s are colored white, and s is gray.
- BFS algorithm maintains the following information for each vertex u:
- color[u] (white, gray and black): indicates status
  - white: not discovered yet
  - gray: discovered but not finished
  - black: finished
- d[u]: distance from s to u
- π(u): predecessor of u


white
Not discovered
gray
Discovered, adjacent white nodes
black
Discovered, no adjacent white nodes

Breadth First Search(BFS) …

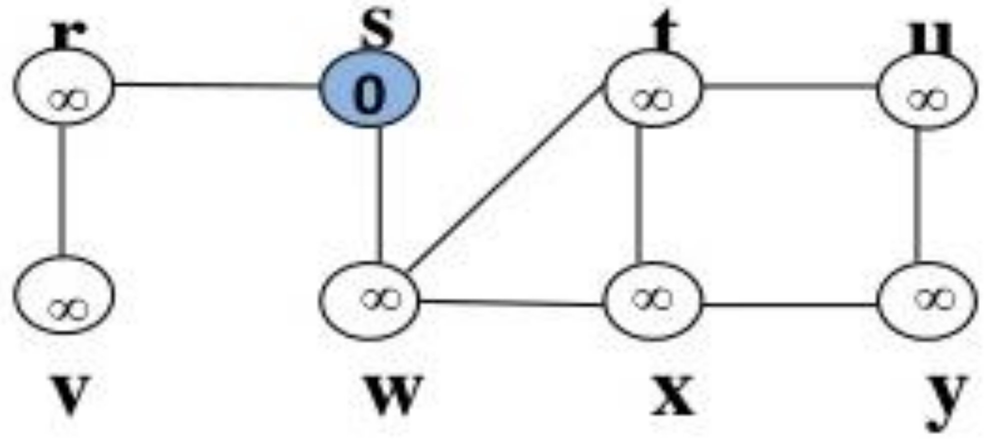
```
BFS(G, s)
1   for each vertex $u \in G.V - \{s\}$
2       $u.color$ = WHITE
3       $u.d = \infty$
4       $u.\pi$ = NIL
5   $s.color$ = GRAY
6   $s.d = 0$
7   $s.\pi$ = NIL
8   $Q = \emptyset$
9   ENQUEUE($Q, s$)
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE($Q$)
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)
18      $u.color$ = BLACK
```

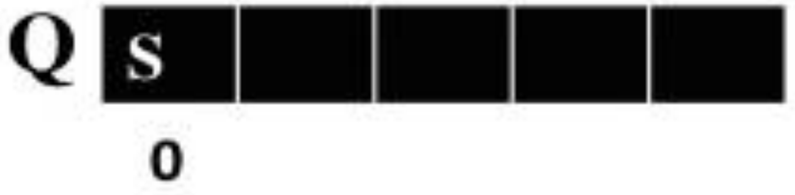
Graph G=(V, E)

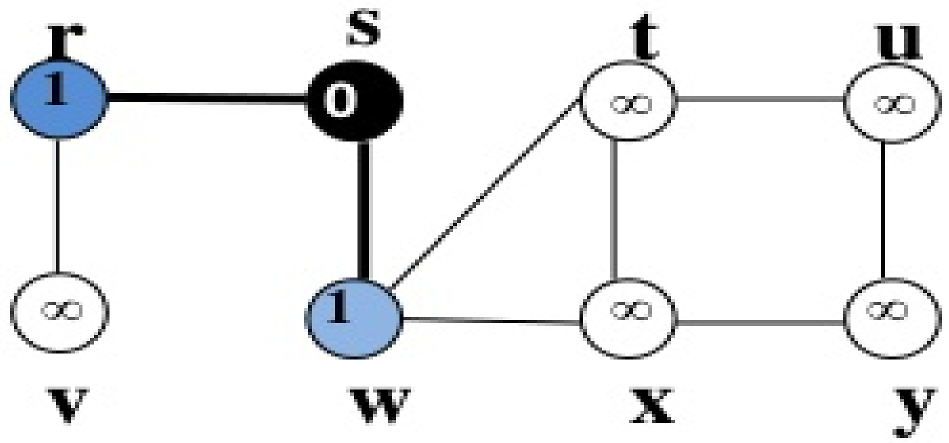


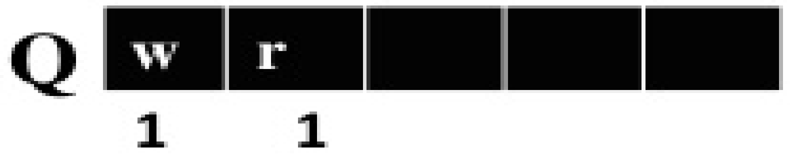
If all the edges in a graph are of the same weight, then BFS can also be used to find the minimum distance between the nodes in a graph.

Breadth First Search(BFS) …

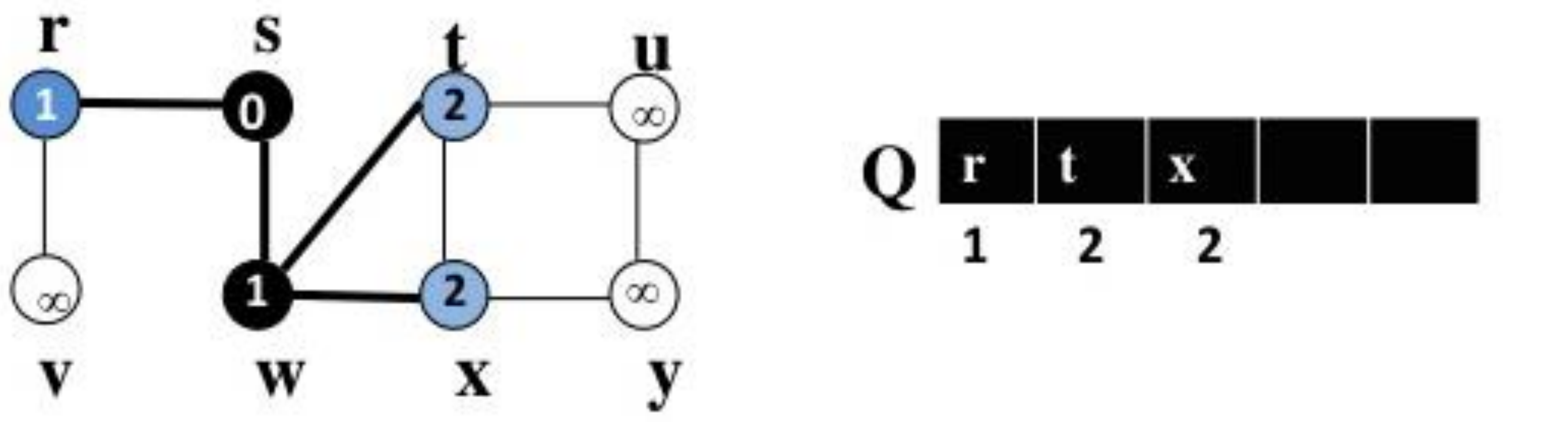


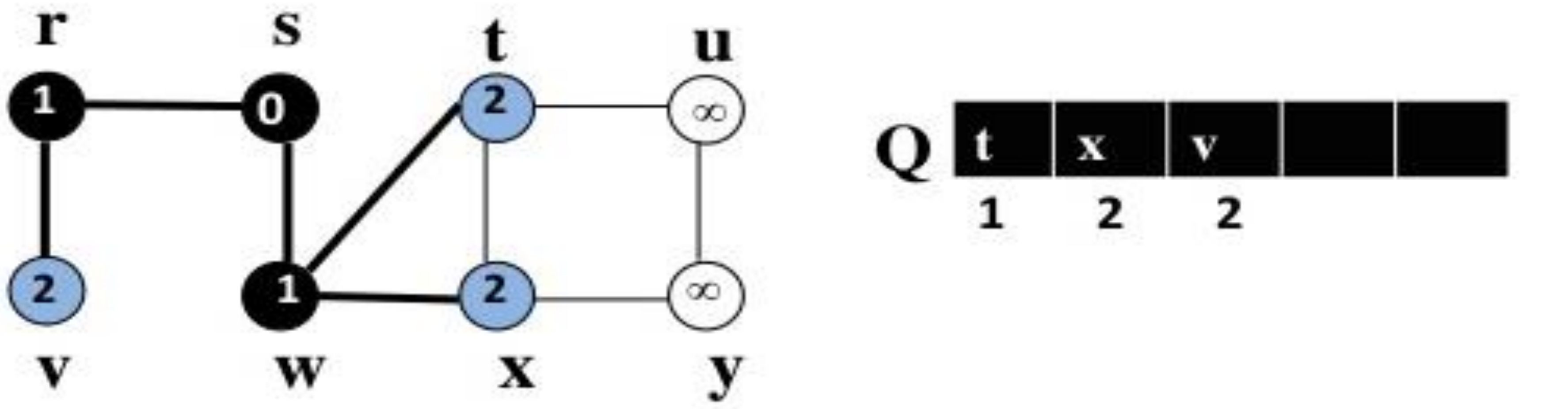




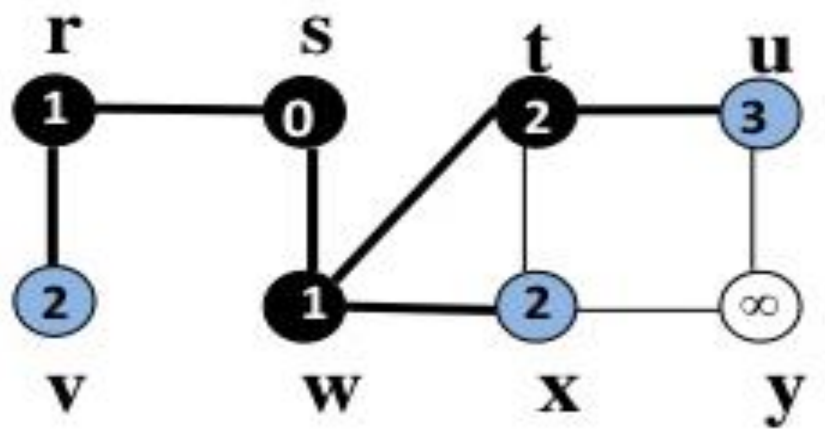


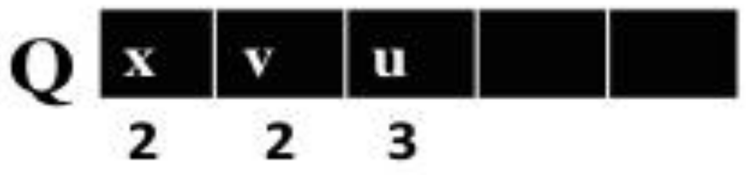
Breadth First Search(BFS) …

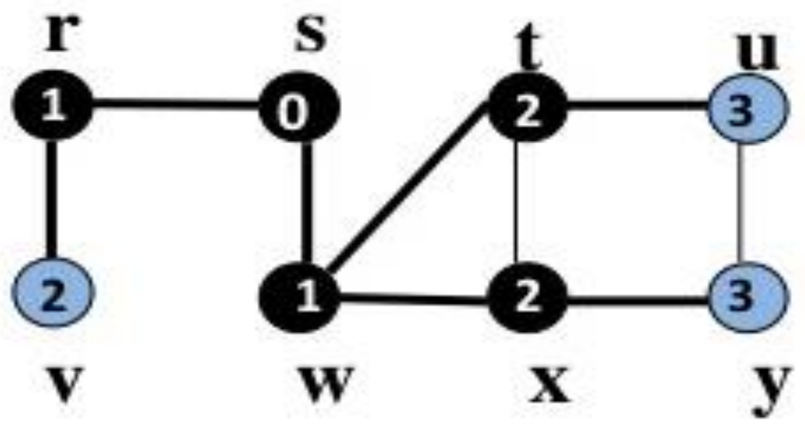


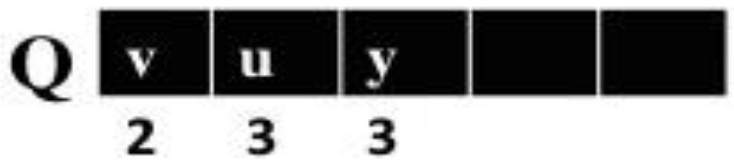


Breadth First Search(BFS) ...

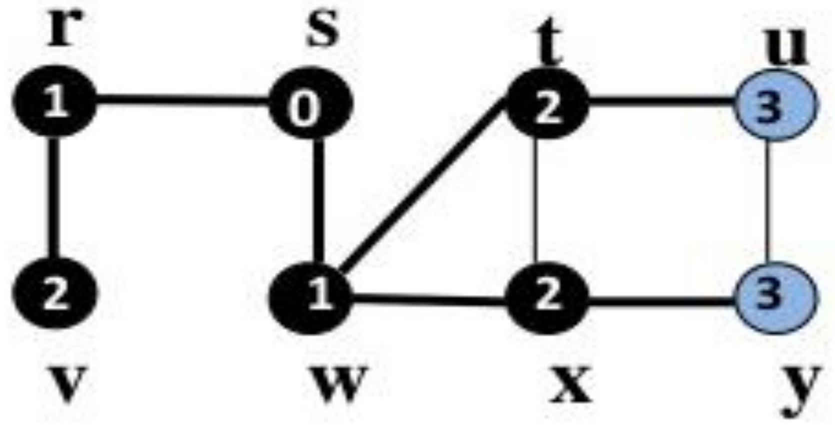


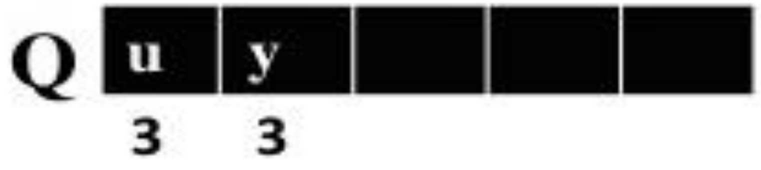


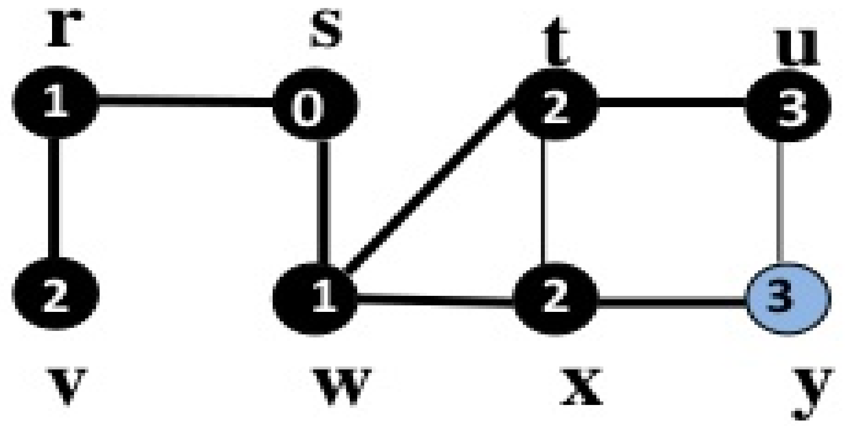


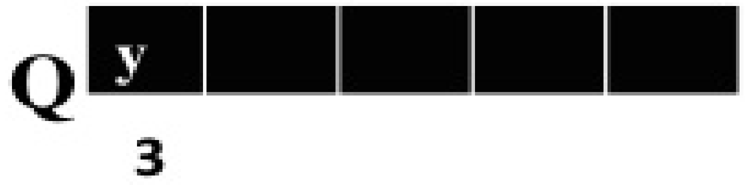


Breadth First Search(BFS) …

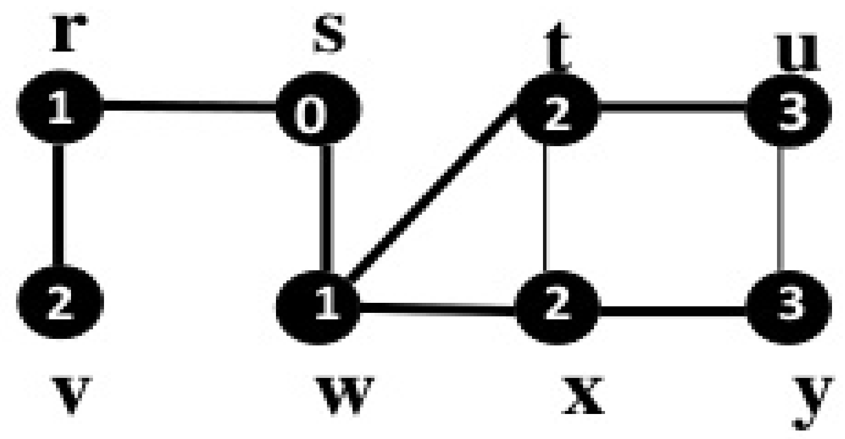


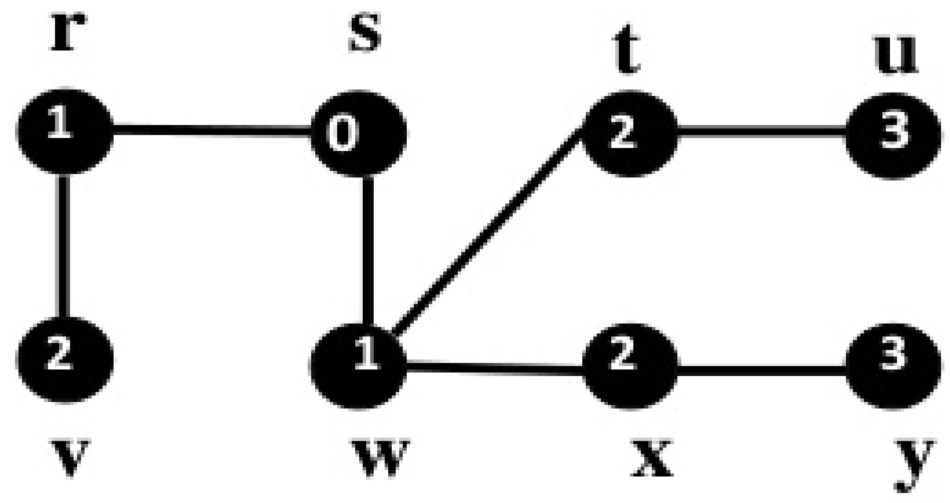






Breadth First Search(BFS) ...





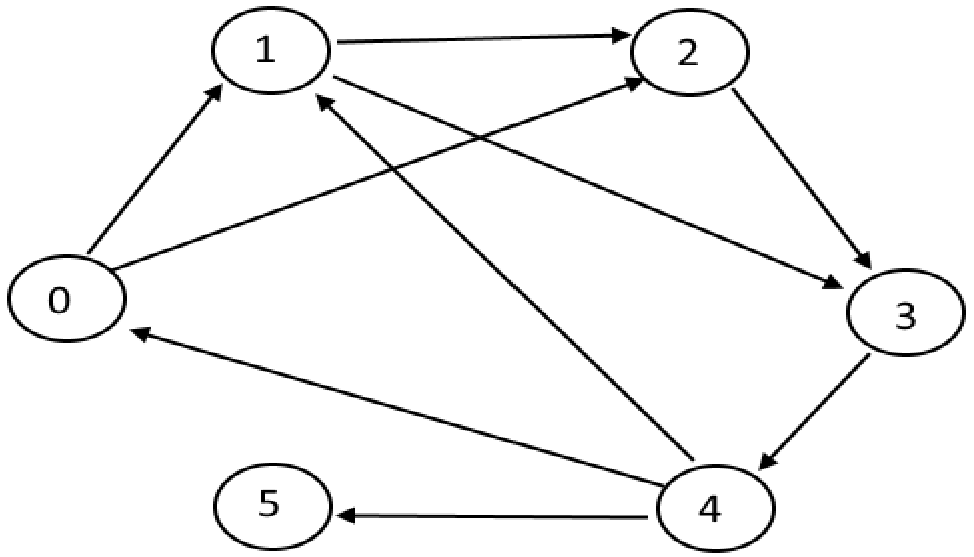
Spanning Tree

BFS traversal: s,w, r, t, x, v, u, y

Breadth First Search(BFS) ...

Find the BFS **traversal of following Graph**

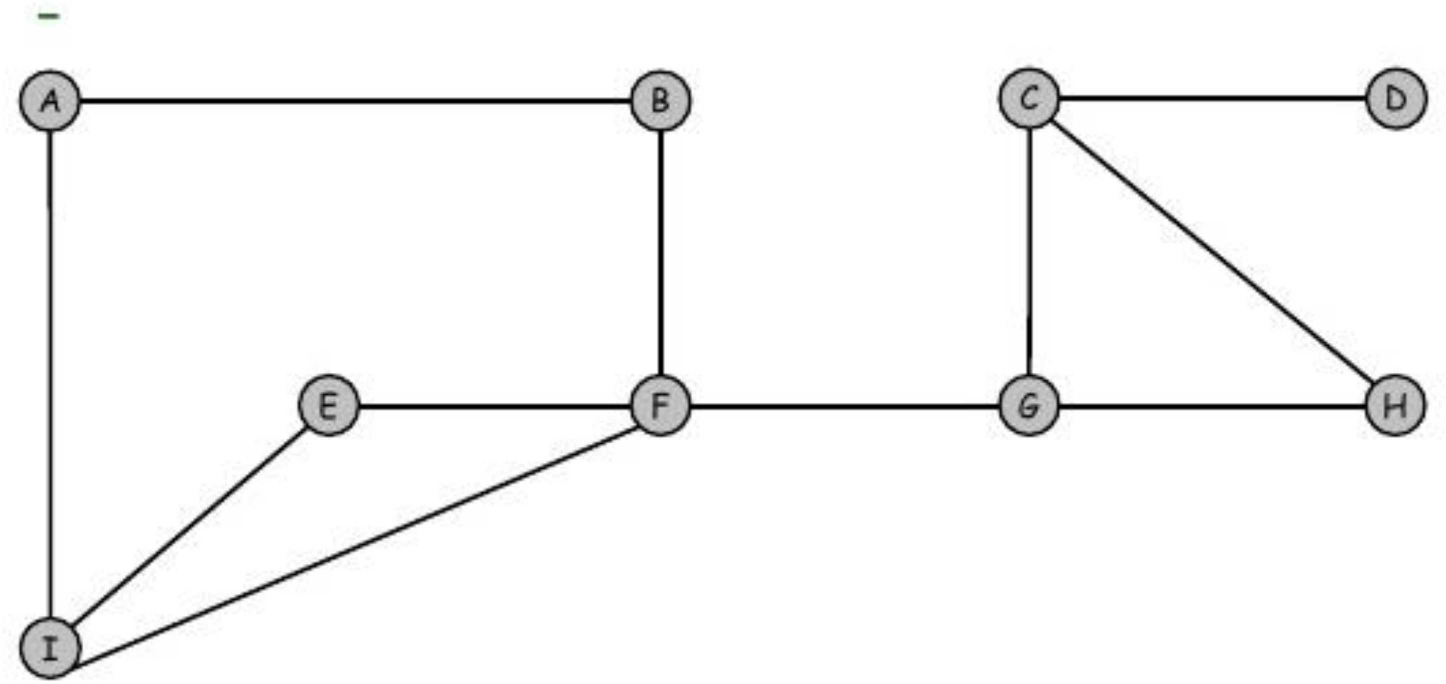
Graph



BFS starting from Node 0

Breadth First Search(BFS) …

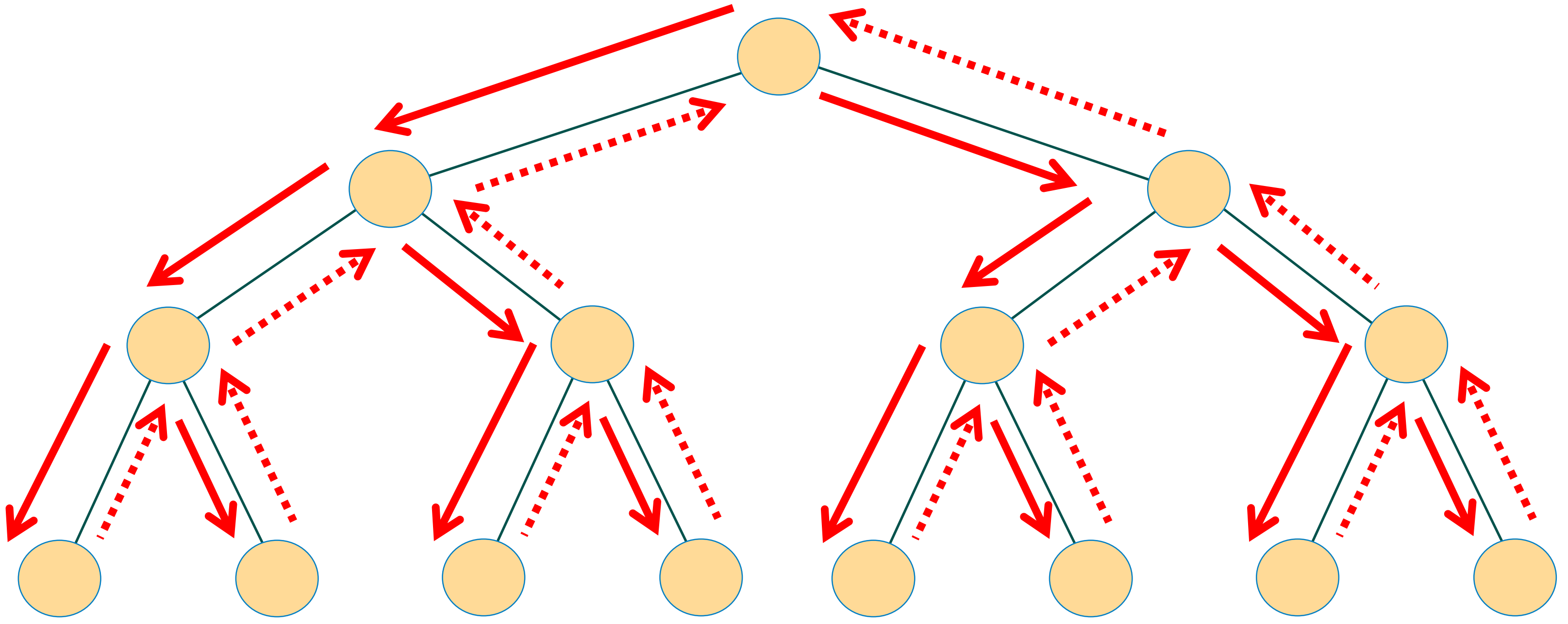
Find the BFS **traversal of following Graph**



Depth-First Search (DFS)

Depth-First Search (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.

- Often implemented recursively using **Stack (LIFO) structure**.
- Many graph algorithms involve visiting or marking vertices.



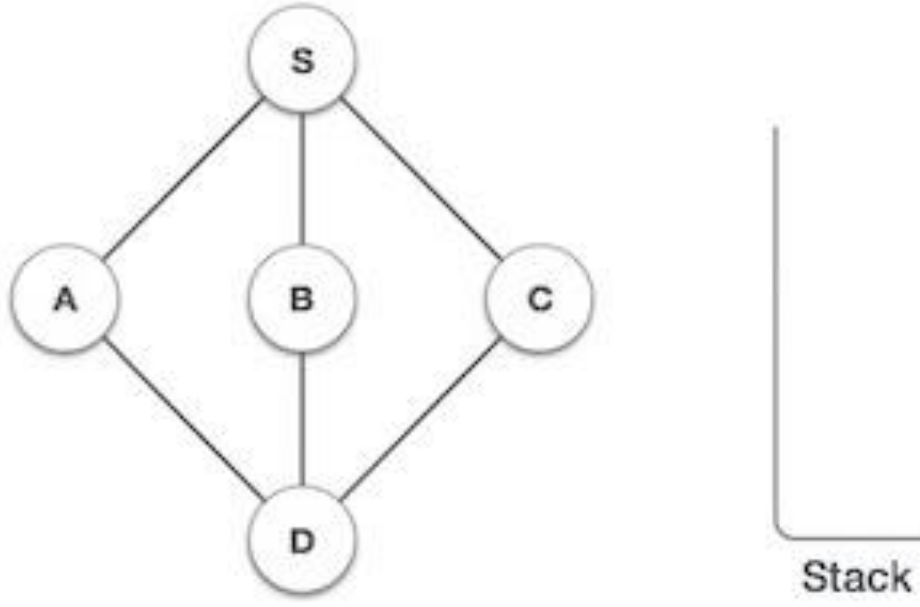
Depth-First Search (DFS)

The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty.

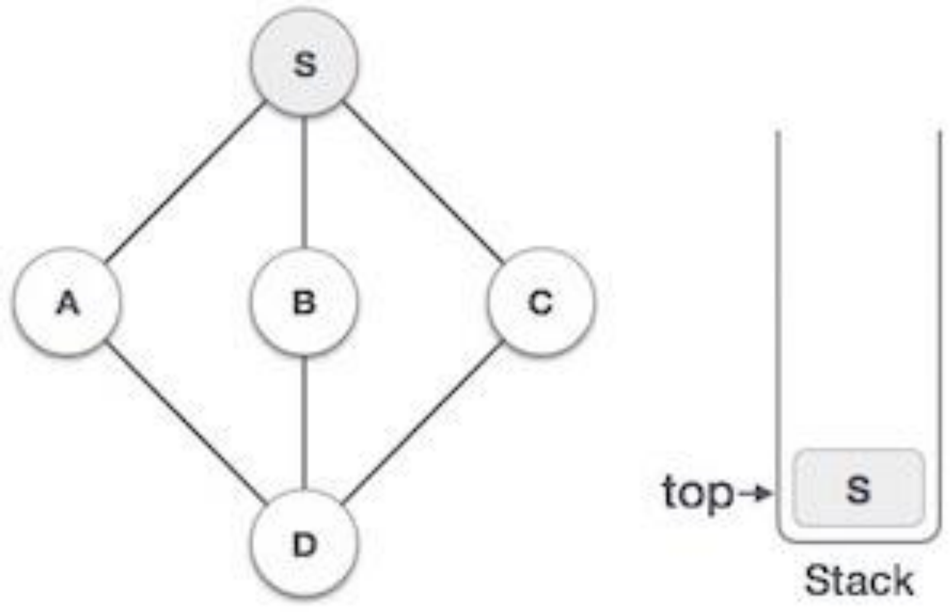
However, ensure that the nodes that are visited are marked. This will prevent from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Depth-First Search (DFS): Example



Initialize the stack.

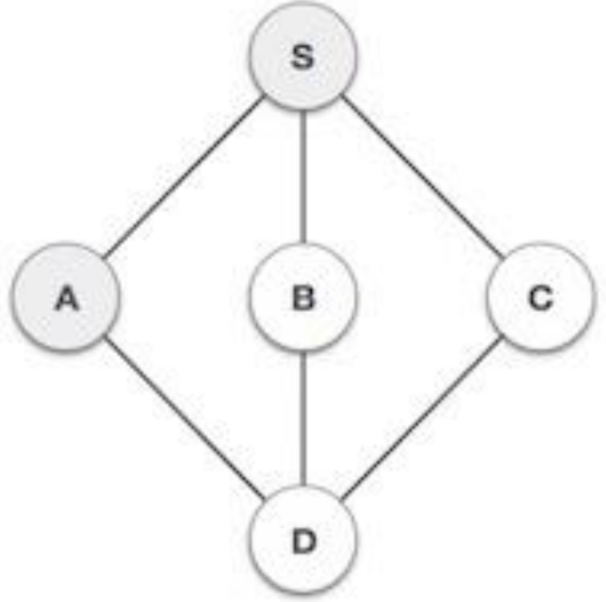
Depth-First Search (DFS): Example

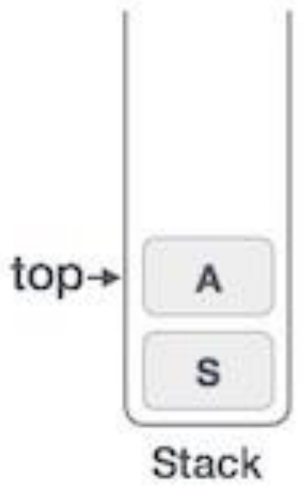


Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**.

We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

Depth-First Search (DFS): Example

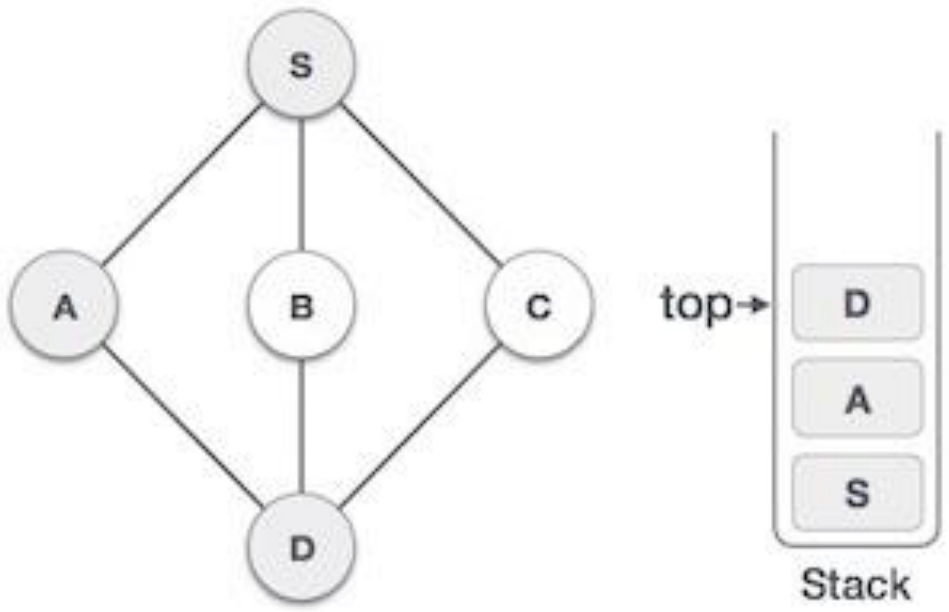




Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A.

Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

Depth-First Search (DFS): Example

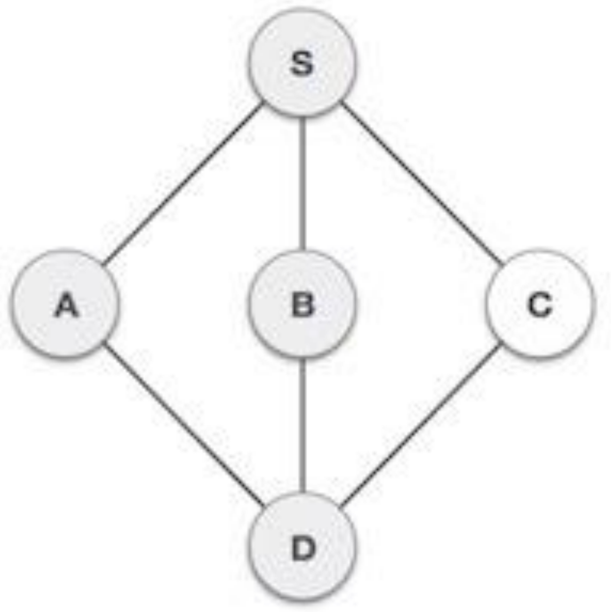


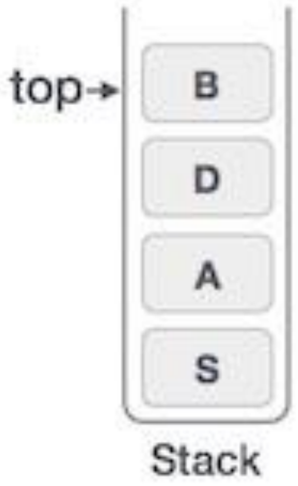
Visit **D** and mark it as visited and put onto the stack.

Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited.

However, we shall again choose in an alphabetical order.

Depth-First Search (DFS): Example

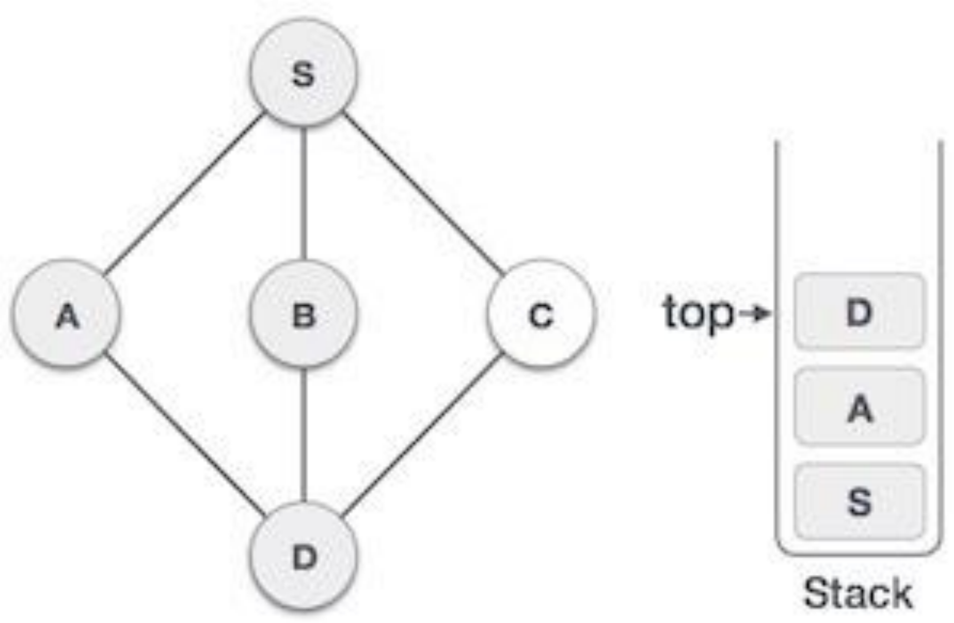




We choose **B**, mark it as visited and put onto the stack.

Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

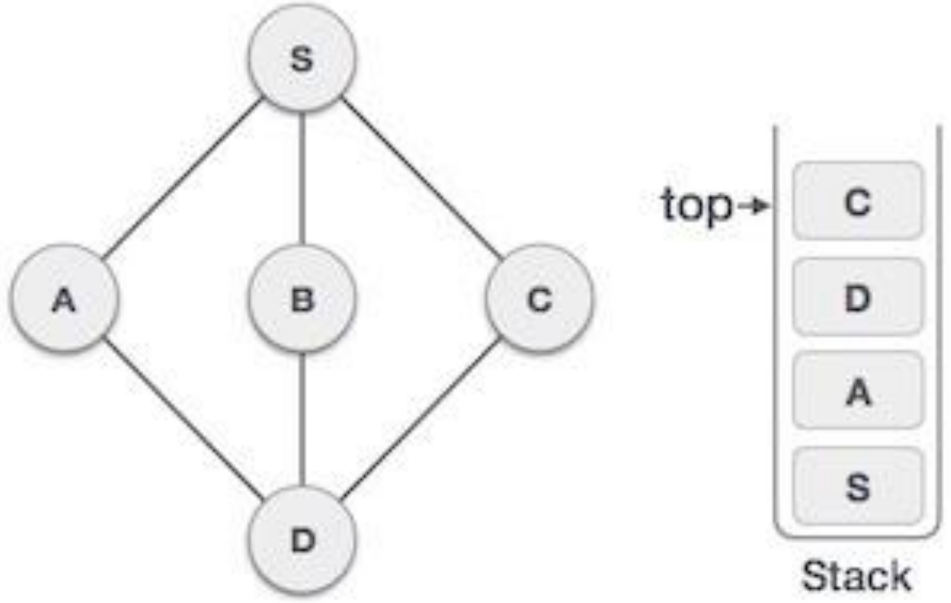
Depth-First Search (DFS): Example



We check the stack top for return to the previous node and check if it has any unvisited nodes.

Here, we find **D** to be on the top of the stack.

Depth-First Search (DFS): Example

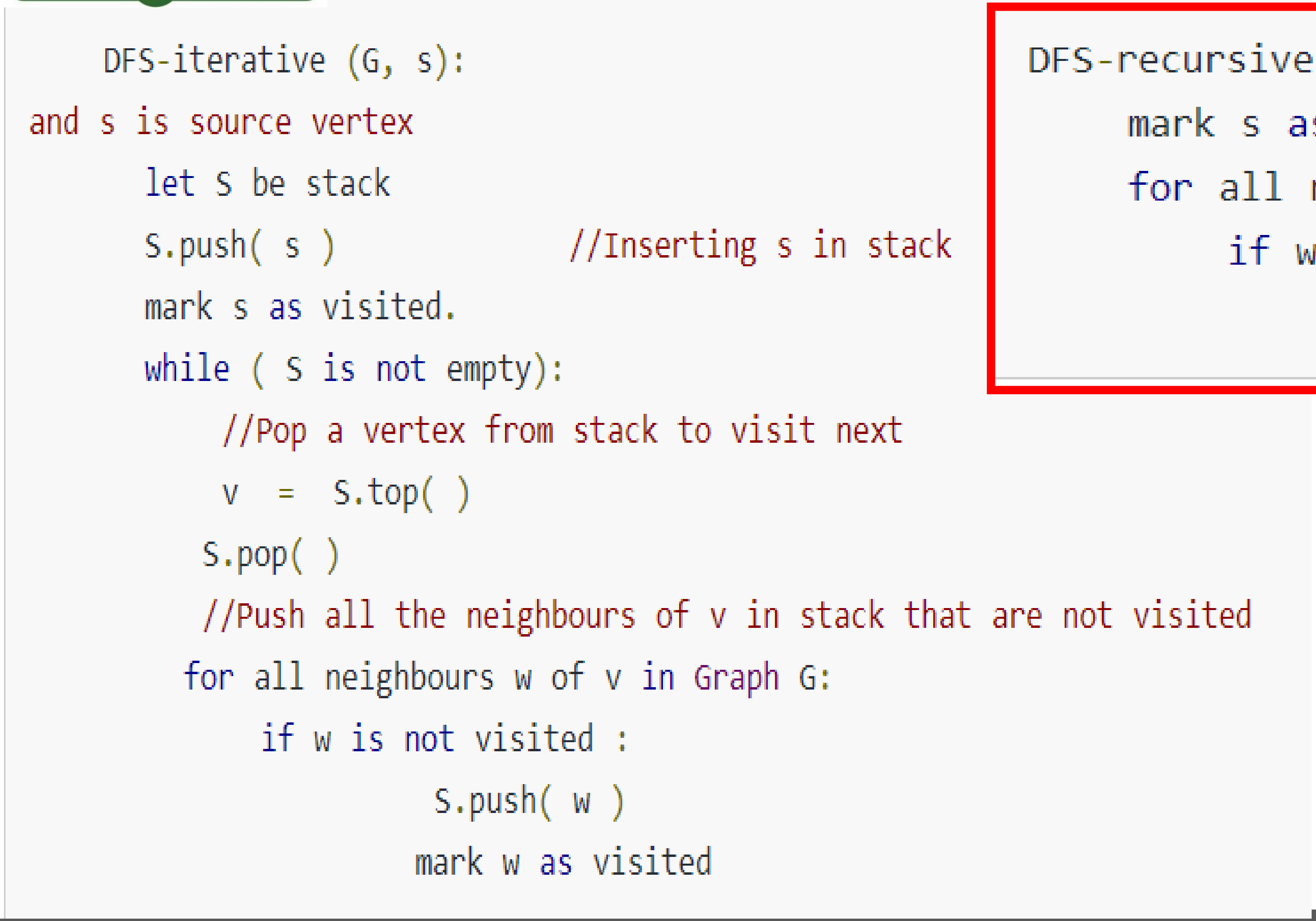


Only unvisited adjacent node is from **D** is **C** now.

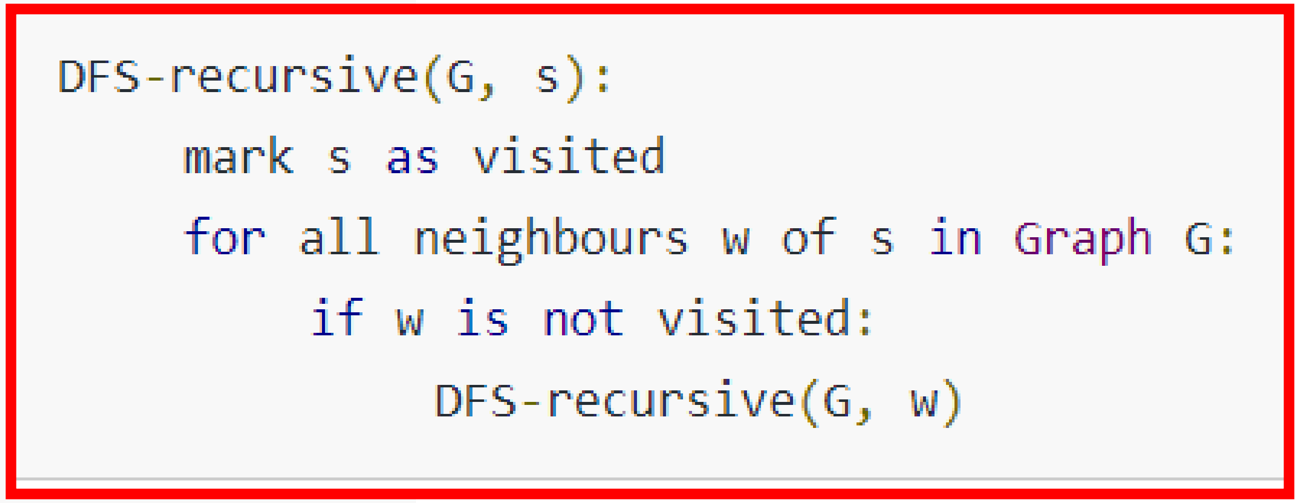
So we visit **C**, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

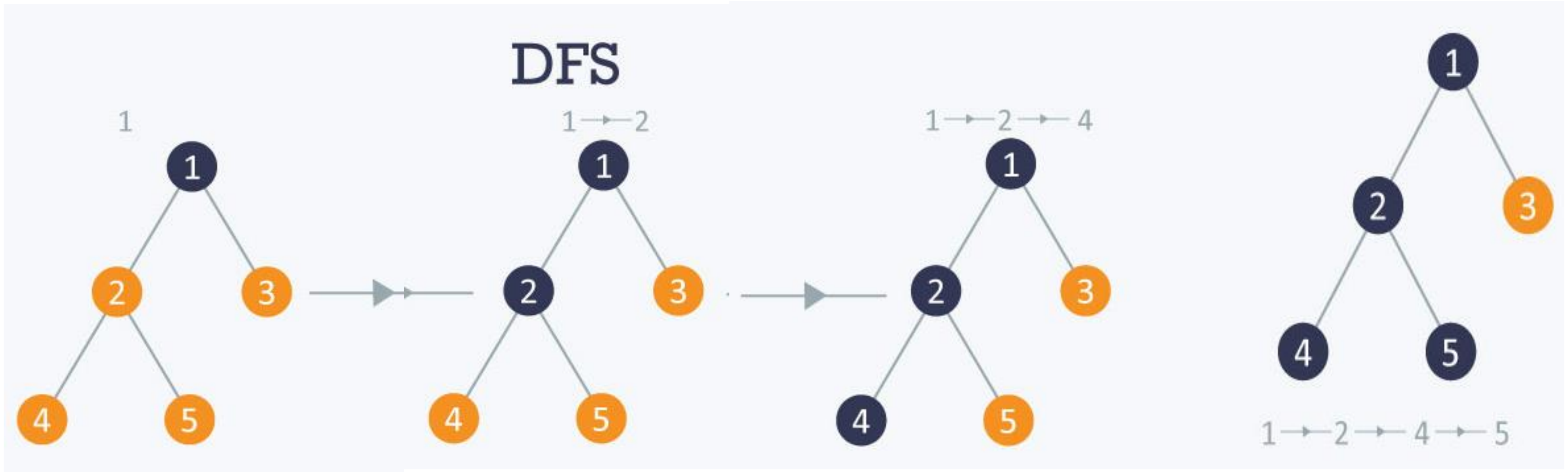
Depth-First Search (DFS) ...



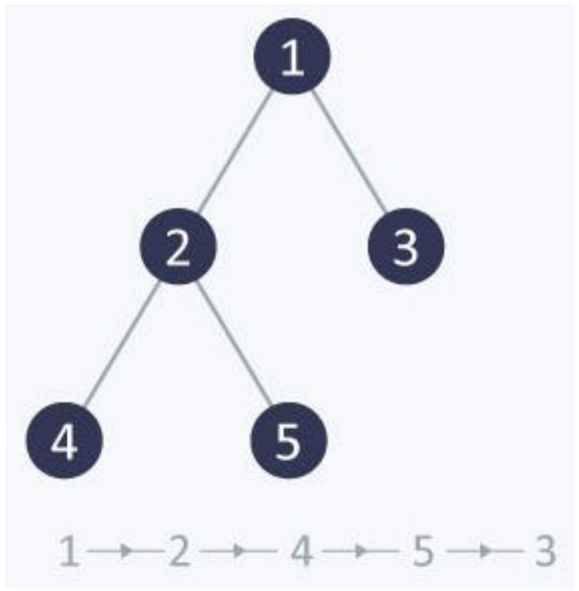
```
    DFS-iterative (G, s):
and s is source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited
```



```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```



DFS Example



DFS, BFS Runtime Complexity

- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?

- Answer: O(|V| + |E|)
  - where |V| = number of vertices, |E| = number of edges
  - Must potentially visit every node and/or examine every edge once.

  - when implemented using the adjacency list.

Any Questions ?

Dr. Anand Singh Jalal
Professor
Email: asjalal@gla.ac.in