Graphs & Graph Algorithms

CMSC 132

Department of Computer Science

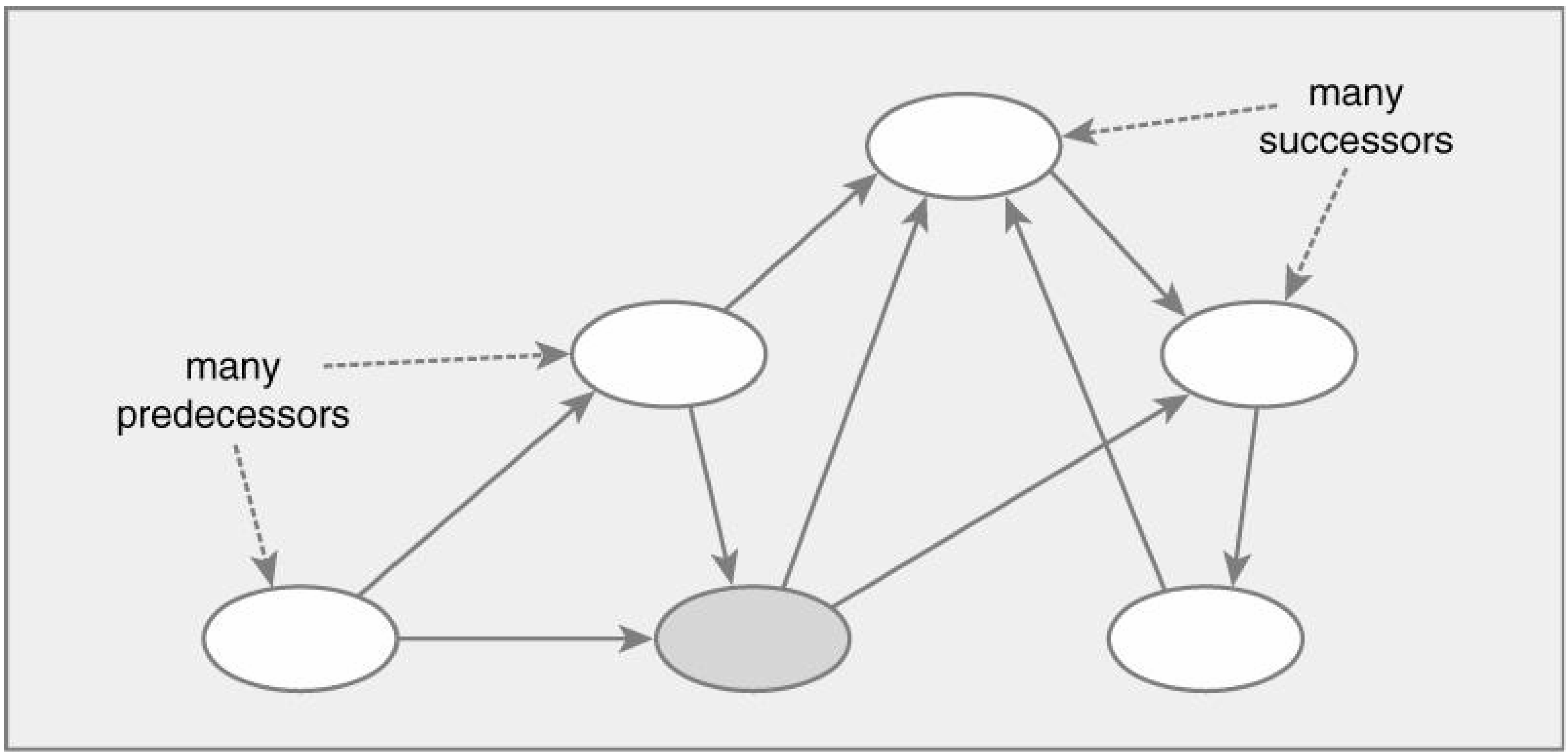
University of Maryland, College Park

Overview

- Graph Data Structures
- Graph Definitions
- Graph Operations
- Breadth-first Search (BFS)
- Traversals Orders
- Depth-first Search (DFS)
- Traversal Algorithms
- BFS vs. DFS Traversal

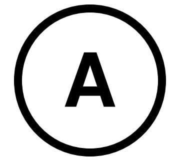
Graph Data Structures

- **Many-to-many relationship between elements**
  - **Each element has multiple predecessors**
  - **Each element has multiple successors**

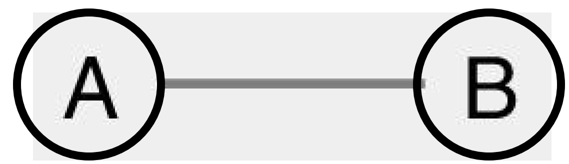


Graph Definitions

- Node
  - Element of graph
  - State
    - List of adjacent nodes

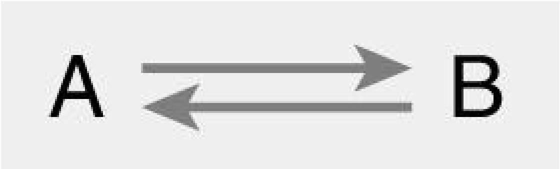


- Edge
  - Connection between two nodes
  - State
    - Endpoints of edge

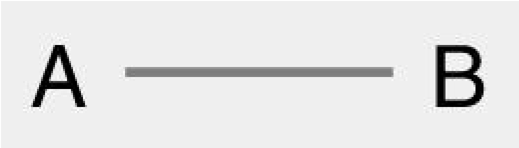


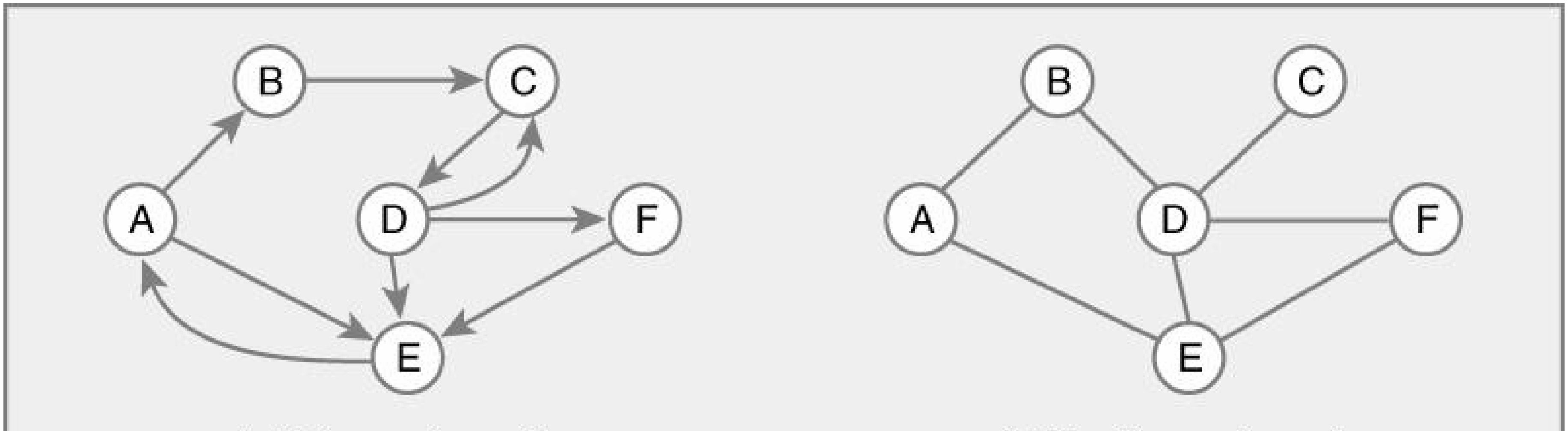
Graph Definitions

- Directed graph
  - Directed edges



- Undirected graph
  - Undirected edges

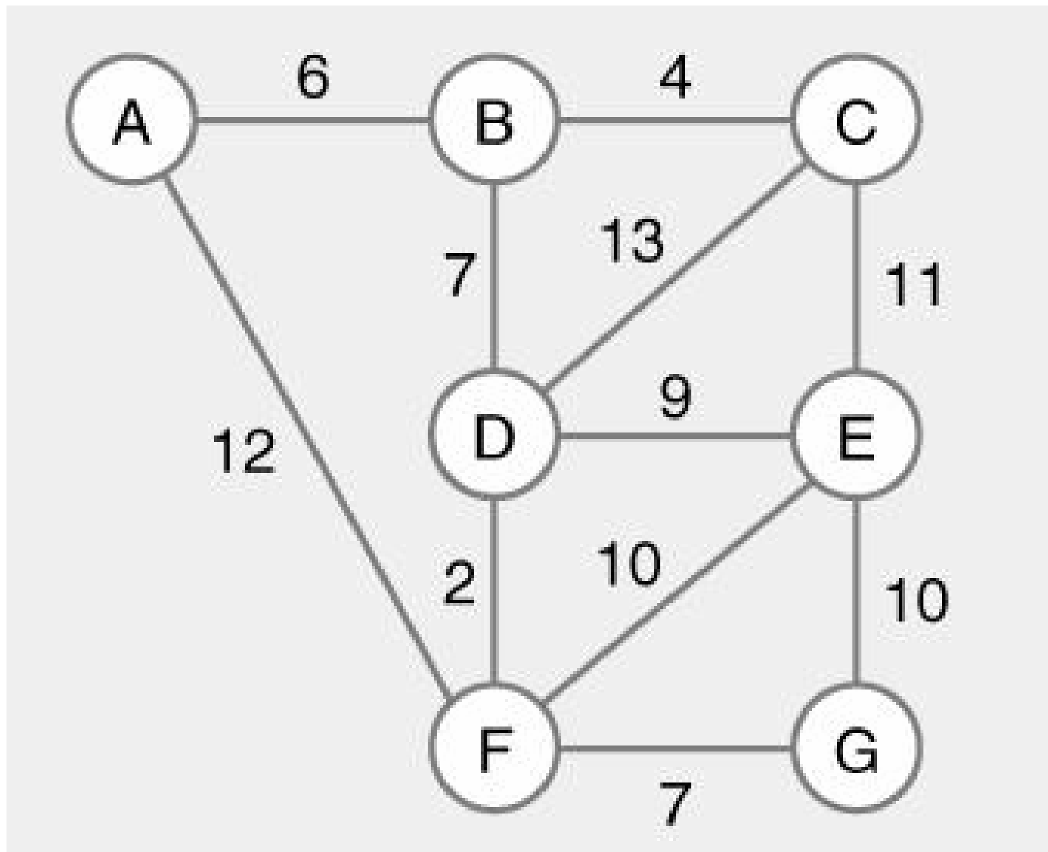




(a) Directed graph (b) Undirected graph

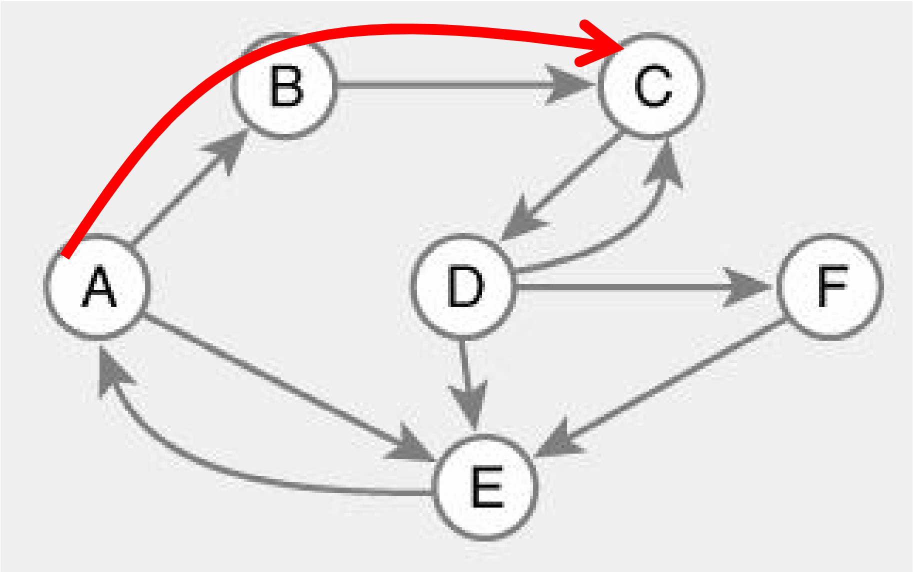
Graph Definitions

- **Weighted graph**
  - **Weight (cost) associated with each edge**



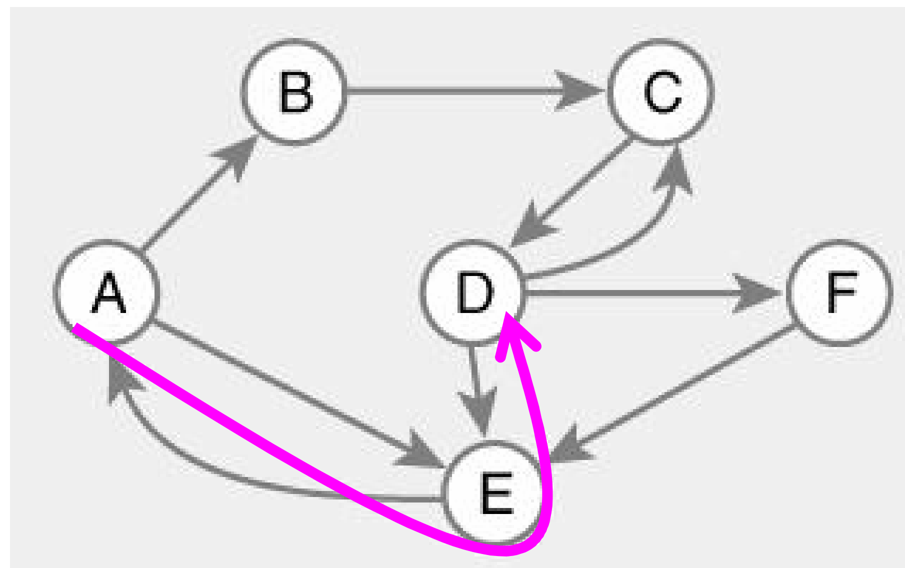
Graph Definitions

- **Path**
  - **Sequence of nodes n_1, n_2, … n_k**
  - **Edge exists between each pair of nodes n_i , n_{i+1}**
  - **Example**
    - **A, B, C is a path**



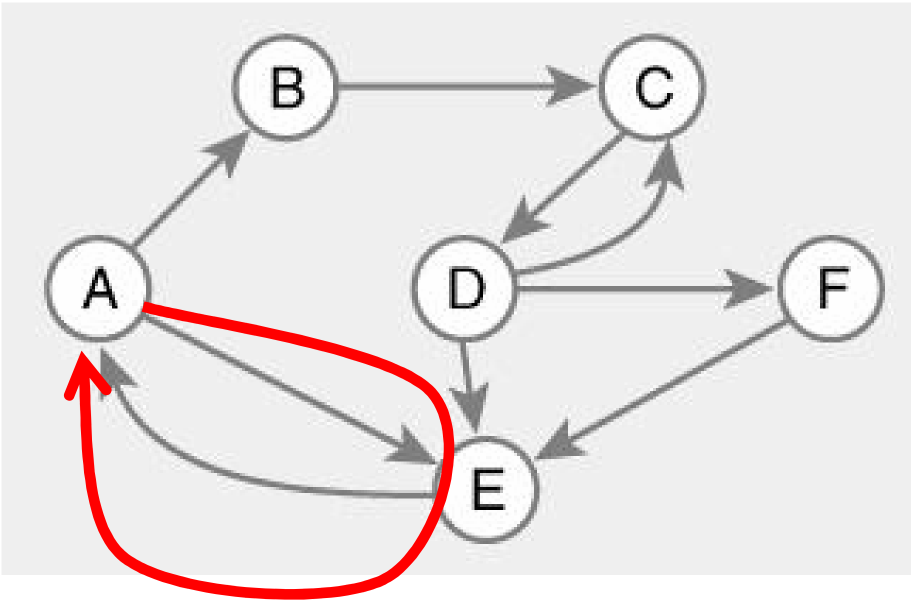
Graph Definitions

- **Path**
  - **Sequence of nodes n_1, n_2, … n_k**
  - **Edge exists between each pair of nodes n_i , n_{i+1}**
  - **Example**
    - **A, B, C is a path**
    - **A, E, D is not a path**



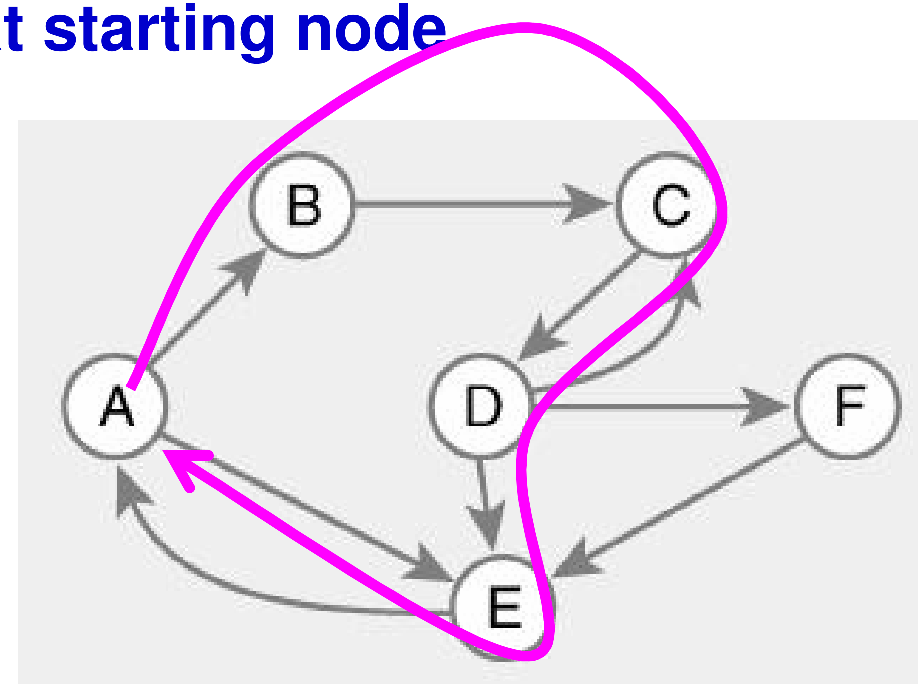
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A



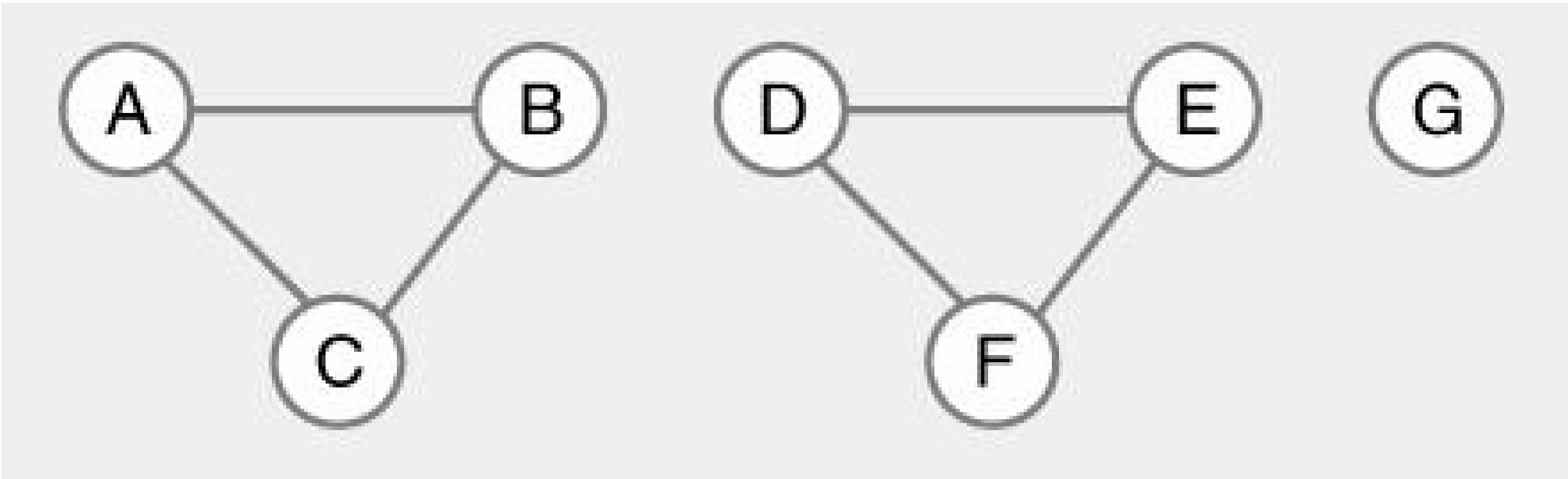
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A
    - A, B, C, D, E, A
- Simple path
  - No cycles in path
- Acyclic graph
  - No cycles in graph



Graph Definitions

- Reachable
  - Path exists between nodes
- Connected graph
  - Every node is reachable from some node in graph



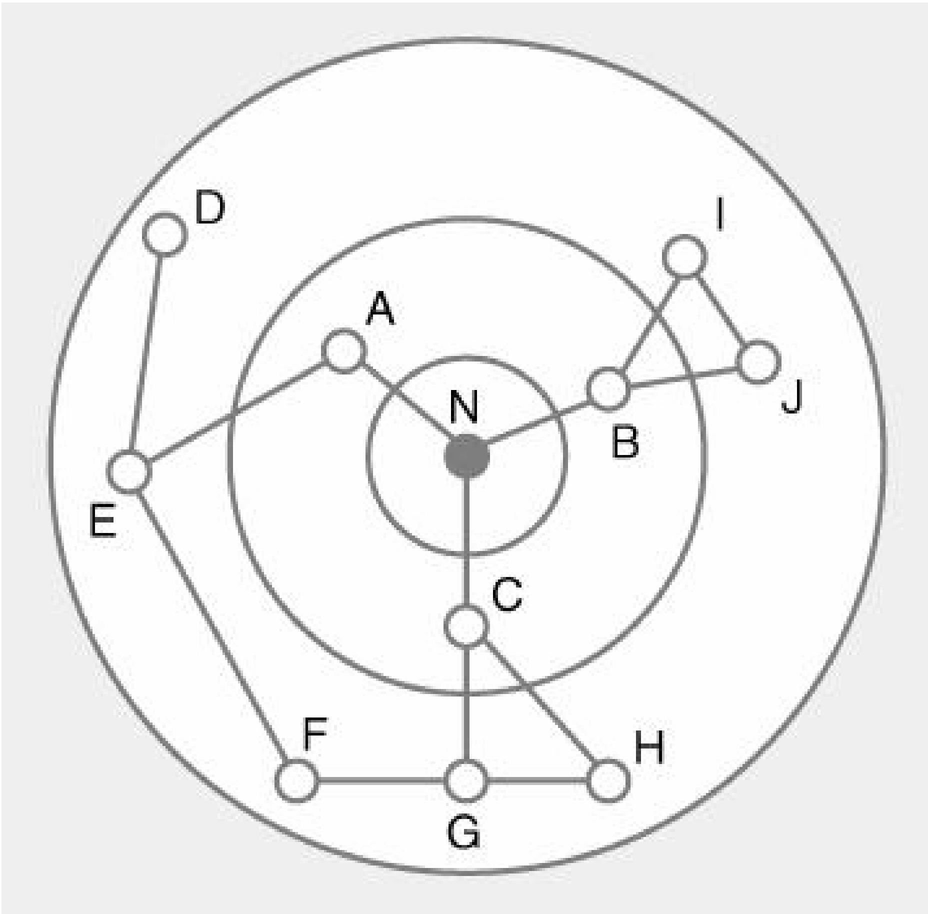
Unconnected graphs

Graph Operations

- Traversal (search)
  - Visit each node in graph exactly once
  - Usually perform computation at each node
  - Two approaches
    - Breadth first search (BFS)
    - Depth first search (DFS)

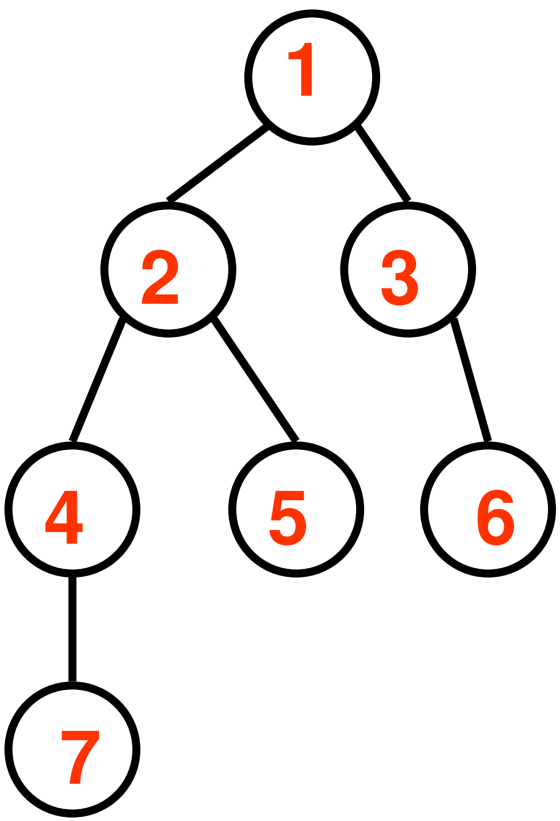
Breadth-first Search (BFS)

- **Approach**
  - **Visit all neighbors of node first**
  - **View as series of expanding circles**
  - **Keep list of nodes to visit in queue**
- **Example traversal**
  - **n**
  - **a, c, b**
  - **e, g, h, i, j**
  - **d, f**

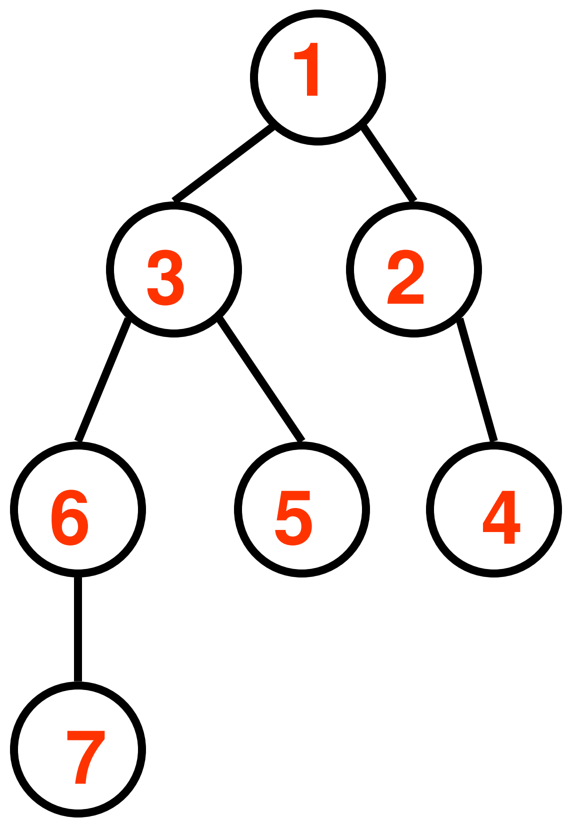


Breadth-first Search (BFS)

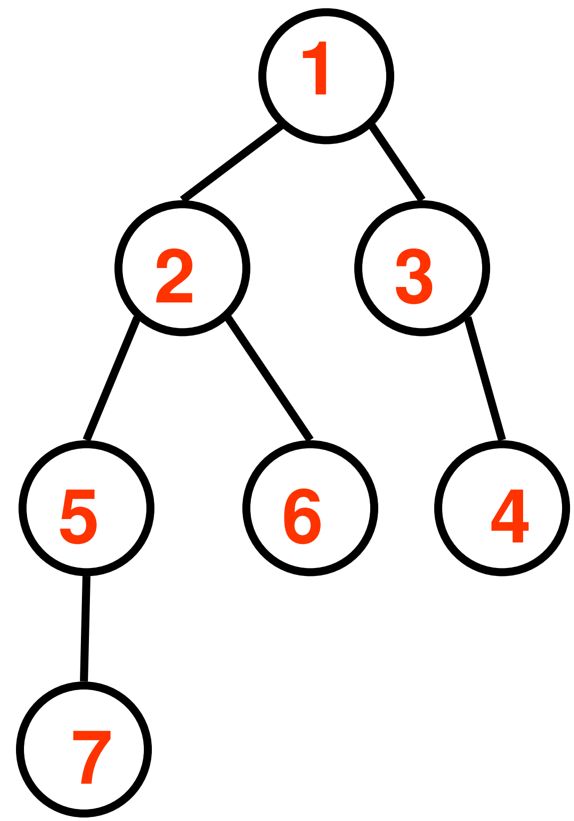
- **Example traversals starting from 1**



Left to right



Right to left



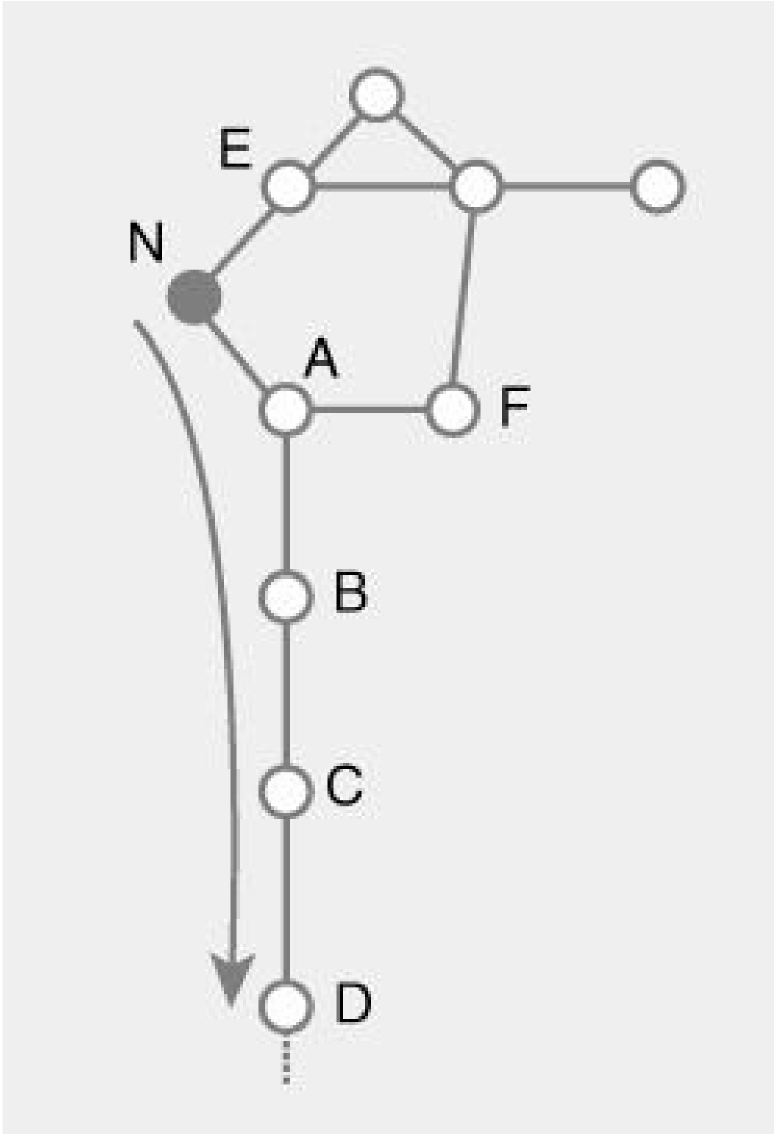
Random

Traversals Orders

- Left to right or right to left doesn't make a lot of sense for graphs
  - Each node just has a set of successors and predecessors; no order among edges
- For breadth first search
  - visit all nodes at distance k from starting point before visiting any nodes at (minimum) distance k+1 from starting point

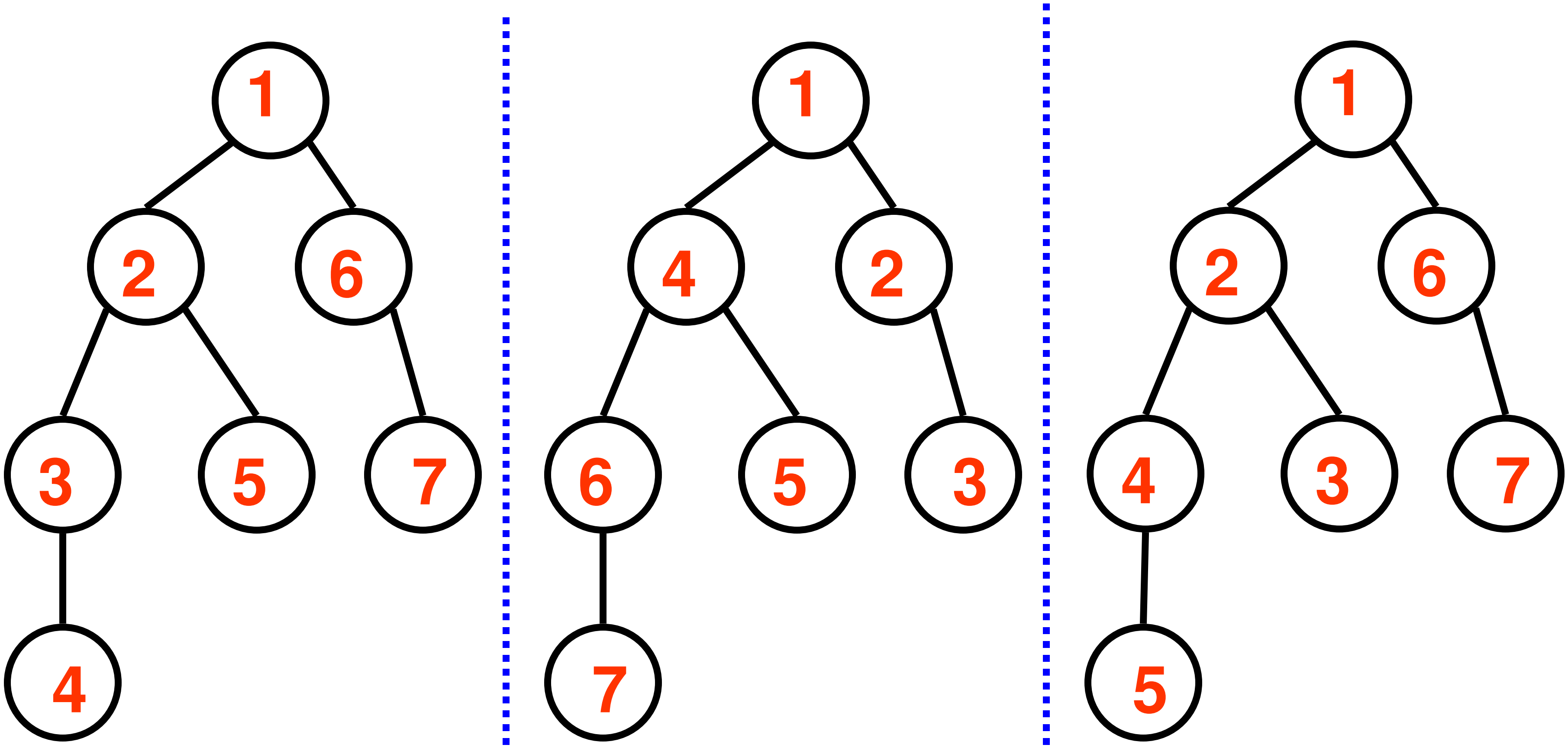
Depth-first Search (DFS)

- **Approach**
  - **Visit all nodes on path first**
  - **Backtrack when path ends**
  - **Keep list of nodes to visit in a stack**
- **Example traversal**
  - **n, a, b, c, d, …**
  - **f …**



Depth-first Search (DFS)

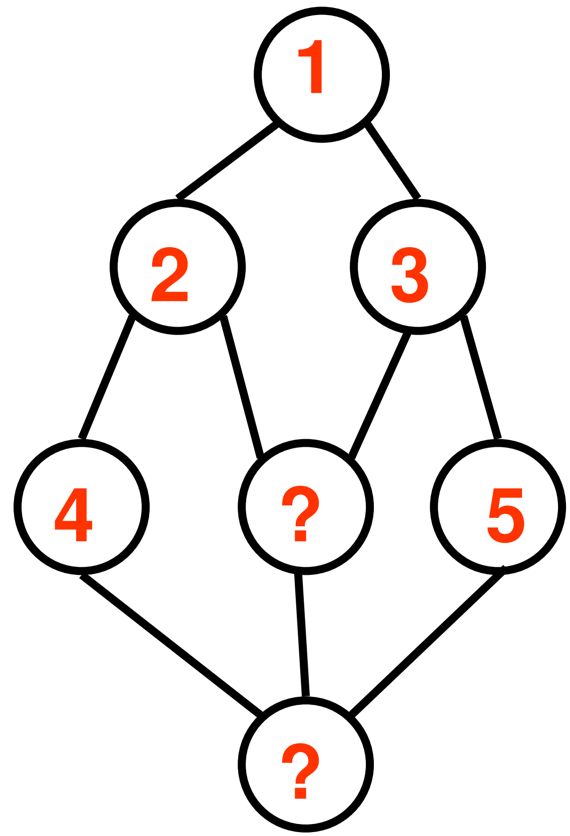
- **Example traversals from 1**



Left to right | **Right to left** | **Random**

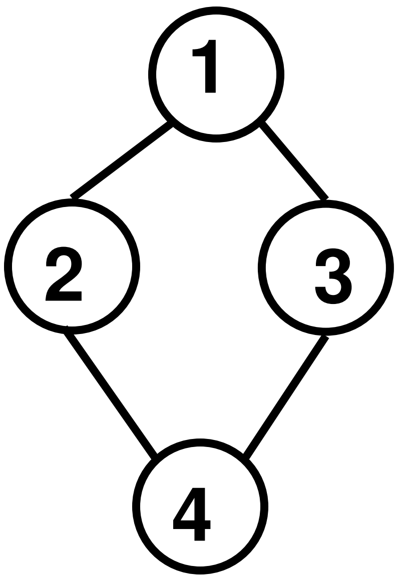
Traversal Algorithms

- **Issue**
  - **How to avoid revisiting nodes**
  - **Infinite loop if cycles present**
- **Approaches**
  - **Record set of visited nodes**
  - **Mark nodes as visited**

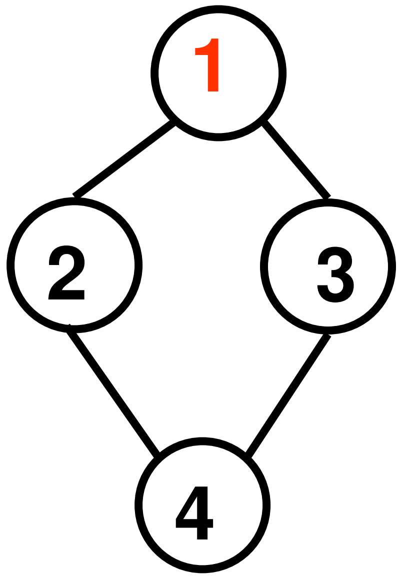


Traversal – Avoid Revisiting Nodes

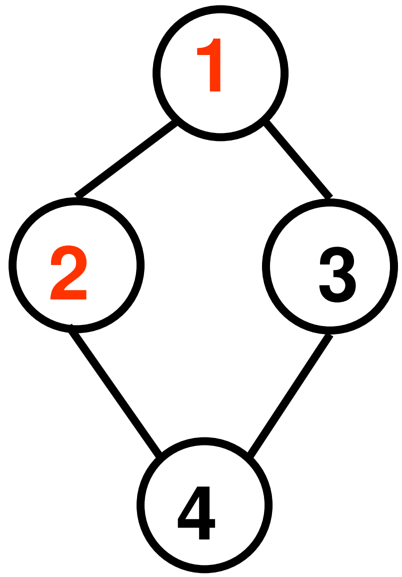
- Record set of visited nodes
  - Initialize { Visited } to empty set
  - Add to { Visited } as nodes is visited
  - Skip nodes already in { Visited }



V = ∅



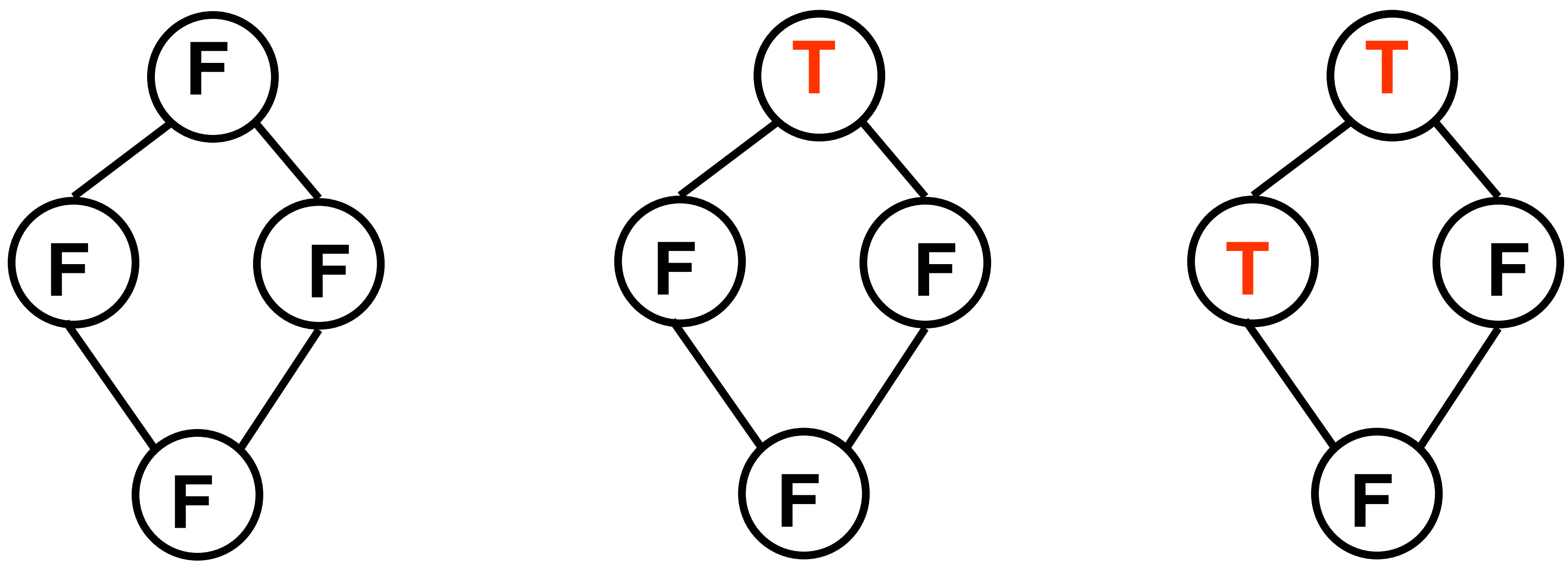
V = { 1 }



V = { 1, 2 }

Traversal – Avoid Revisiting Nodes

- Mark nodes as visited
  - Initialize tag on all nodes (to False)
  - Set tag (to True) as node is visited
  - Skip nodes with tag = True



Traversal Algorithm Using Sets

{ Visited } = ∅

{ Discovered } = { 1st node }

while ({ Discovered } ≠ ∅)

    take node X out of { Discovered }

    if X not in { Visited }

        add X to { Visited }

        for each successor Y of X

            if (Y is not in { Visited })

                add Y to { Discovered }

Traversal Algorithm Using Tags

```
for all nodes X
     set X.tag = False
{ Discovered } = { 1st node }
while ( { Discovered } ≠ ∅ )
    take node X out of { Discovered }
    if (X.tag = False)
        set X.tag = True
        for each successor Y of X
            if (Y.tag = False)
                add Y to { Discovered }
```

BFS vs. DFS Traversal

- Order nodes taken out of { Discovered } key
- Implement { Discovered } as Queue
  - First in, first out
  - Traverse nodes breadth first
- Implement { Discovered } as Stack
  - First in, last out
  - Traverse nodes depth first

BFS Traversal Algorithm

```
for all nodes X
    X.tag = False
put 1st node in Queue
while ( Queue not empty )
    take node X out of Queue
    if (X.tag = False)
        set X.tag = True
        for each successor Y of X
            if (Y.tag = False)
                put Y in Queue
```

DFS Traversal Algorithm

```
for all nodes X
      X.tag = False
put 1st node in Stack
while (Stack not empty )
     pop X off Stack
     if (X.tag = False)
          set X.tag = True
          for each successor Y of X
               if (Y.tag = False)
                    push Y onto Stack
```

Recursive DFS Algorithm

```
Traverse( )
    for all nodes X
        set X.tag = False
    Visit ( 1st node )
Visit ( X )
    set X.tag = True
    for each successor Y of X
        if (Y.tag = False)
            Visit ( Y )
```