Lecture 12
Graphs

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

Inter-data Relationships

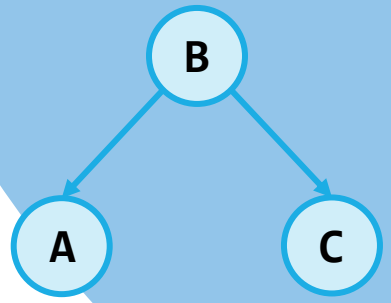
Arrays

- Elements only store pure data, no connection info
- Only relationship between data is order

0	1	2
A	B	C

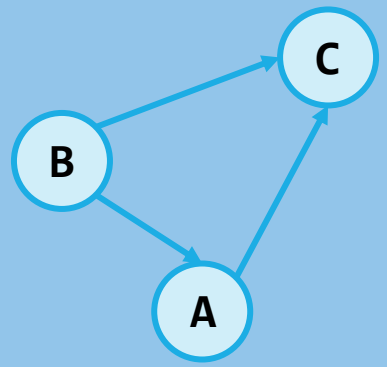
Trees

- Elements store data and connection info
- Directional relationships between nodes; limited connections



Graphs

- Elements AND connections can store data
- Relationships dictate structure; huge freedom with connections



Applications

Physical Maps

- Airline maps
  - nodes are airports, edges are flight paths
- Traffic
  - nodes are addresses, edges are streets

Relationships

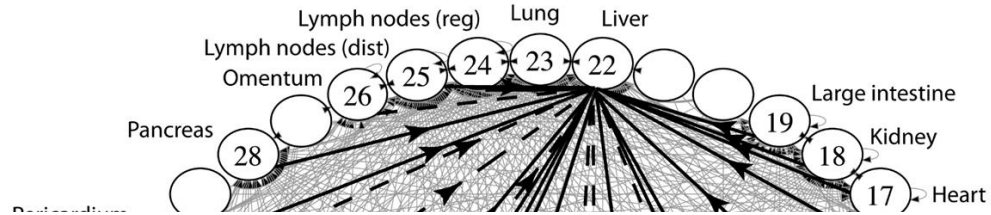
- Social media graphs
  - nodes are accounts, edges are follower relationships
- Traffic
  - nodes are classes, edges are usage

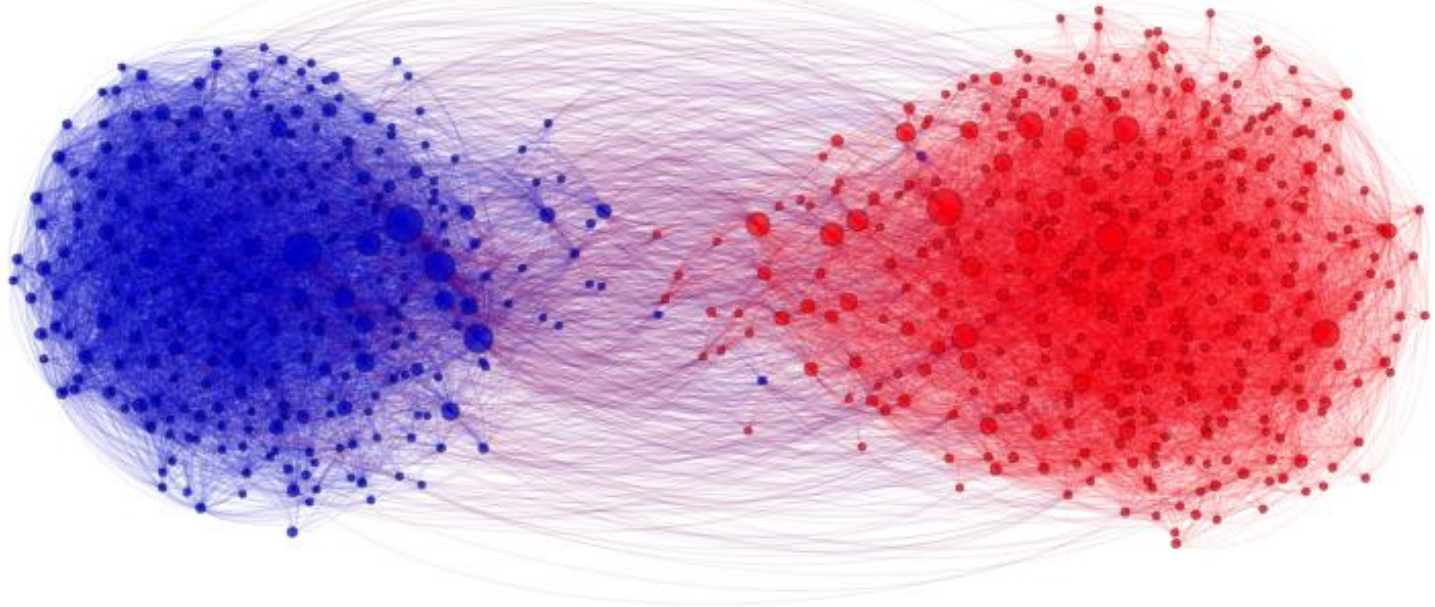
Influence

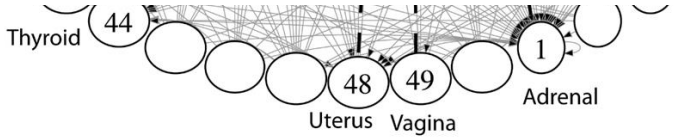
- Biology
  - nodes are cancer cell desinations, edges are migration paths

Related topics

- Web Page Ranking
  - nodes are web pages, edges are hyperlinks
- Wikipedia
  - nodes are articles, edges are links







And so many more!!

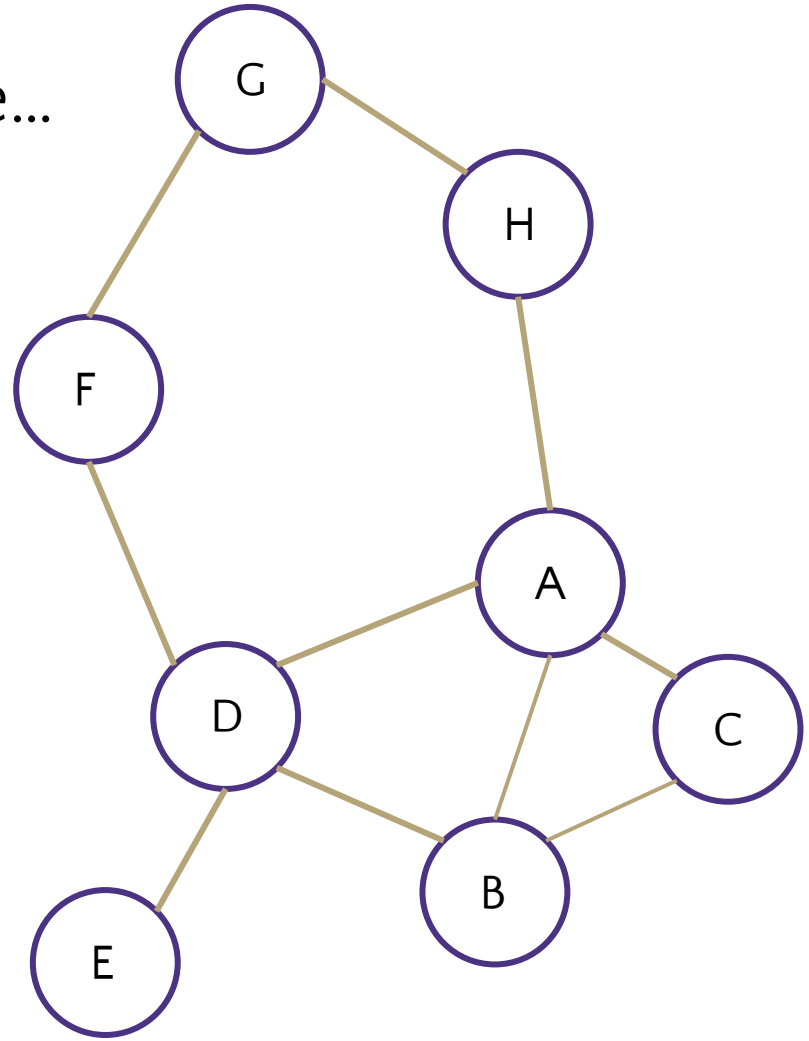
www.allthingsgraphed.com

Graph: Formal Definition

A **graph** is defined by a pair of sets G = (V, E) where...

- V is a set of **nodes**
  - A node or "node" is a data entity
  - V = { A, B, C, D, E, F, G, H }

- E is a set of **edges**
  - An edge is a connection between two nodes
  - E = { (A, B), (A, C), (A, D), (A, H),
    (C, B), (B, D), (D, E), (D, F),
    (F, G), (G, H)}



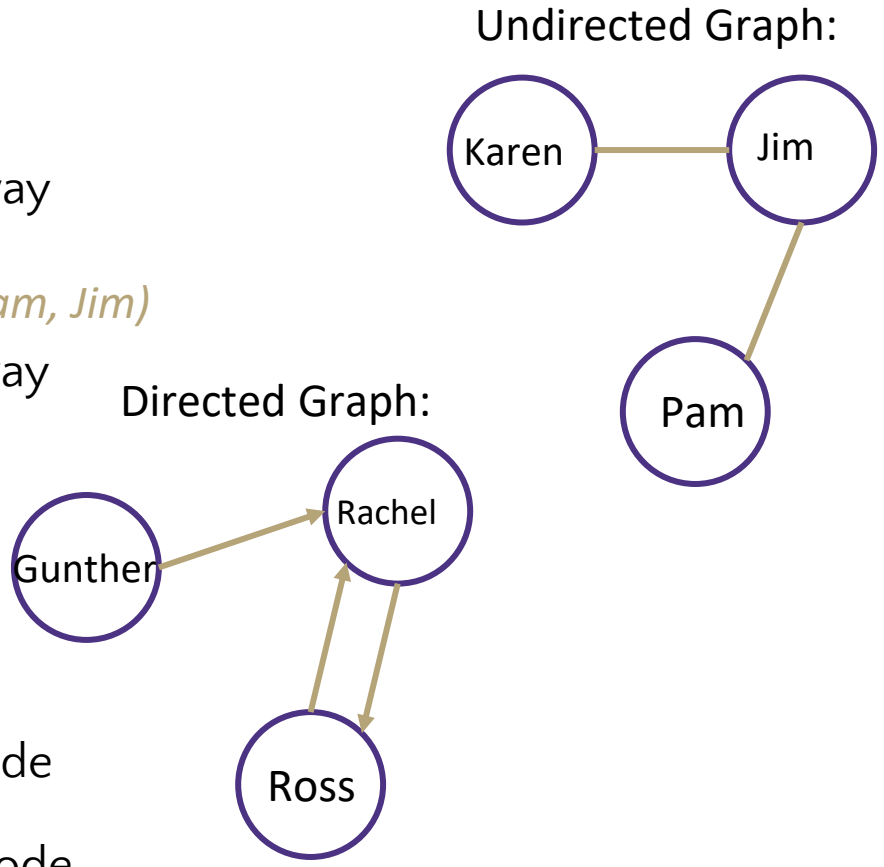
Graph Terminology

Graph Direction

- **Undirected graph** – edges have no direction and are two-way
  - V = { Karen, Jim, Pam }
  - E = { (Jim, Pam), (Jim, Karen) } *inferred (Karen, Jim) and (Pam, Jim)*
- **Directed graphs** – edges have direction and are thus one-way
  - V = { Gunther, Rachel, Ross }
  - E = { (Gunther, Rachel), (Rachel, Ross), (Ross, Rachel) }

Degree of a Node

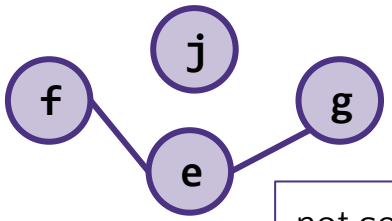
- **Degree** – the number of edges connected to that node
  - Karen : 1, Jim : 2, Pam : 1
- **In-degree** – the number of directed edges that point to a node
  - Gunther : 0, Rachel : 2, Ross : 1
- **Out-degree** – the number of directed edges that start at a node
  - Gunther : 1, Rachel : 1, Ross : 1



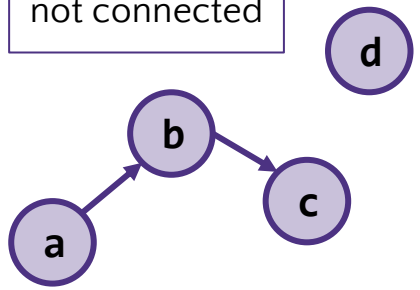
More Graph Terminology

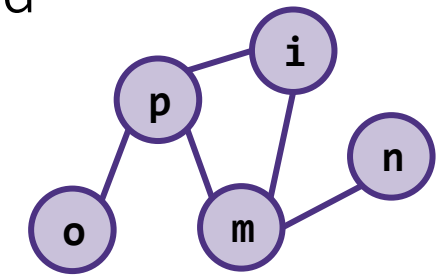
Two nodes are **connected** if there is a path between them

- If all the nodes are connected, we say the graph is **connected**
  - A directed graph is **weakly connected** if replacing every directed edge with an undirected edge results in a connected graph
  - A directed graph is **strongly connected** if a directed path exists between every pair of nodes
- The number of edges leaving a node is its **degree**

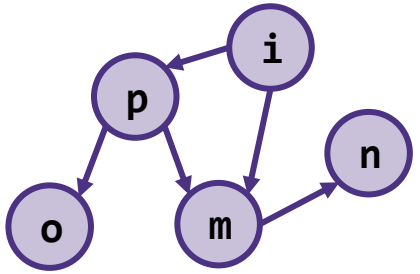


not connected





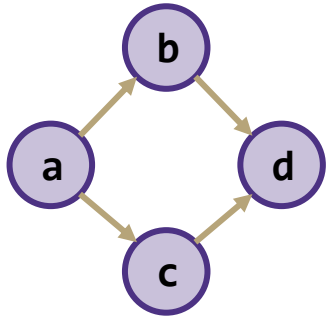
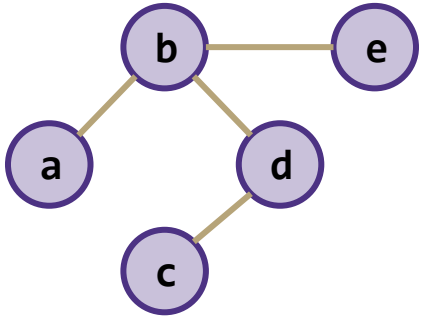
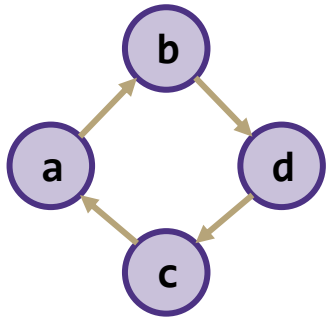
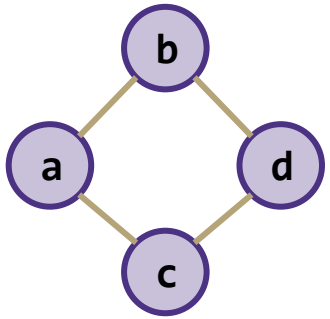
connected



A **path** is a sequence of nodes connected by edges

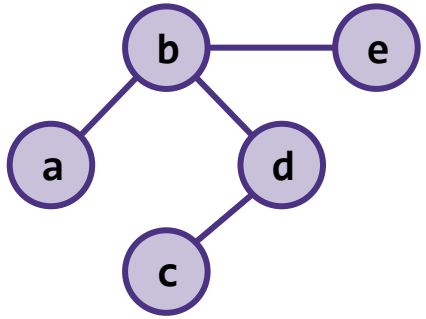
- A **simple path** is a path without repeated nodes
- A **cycle** is a path whose first and last nodes are the same
  - A graph with a cycle is **cyclic**

Directed vs Undirected; Acyclic vs Cyclic

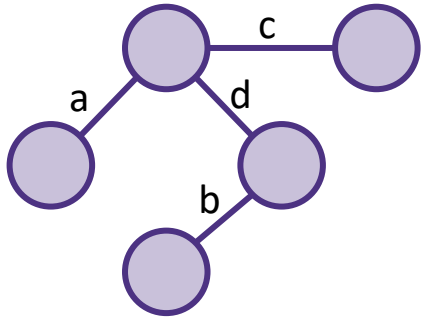
	Directed:	Undirected:
Acyclic:		
Cyclic:		

Labeled and Weighted Graphs

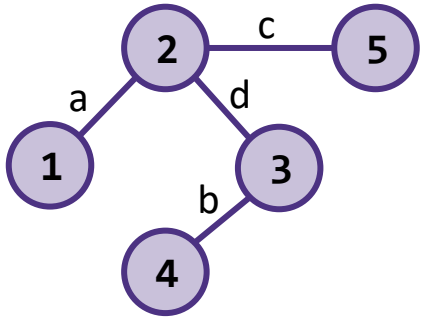
Node Labels



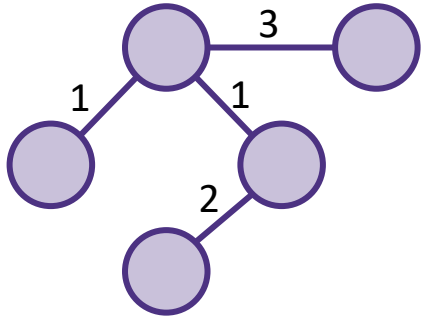
Edge Labels



Node & Edge Labels



Numeric Edge Labels
(**Edge Weights**)



Multi-Variable Analysis

- So far, we thought of everything as being in terms of some single argument “n”
- With graphs, we need to consider:
  - n (or |V|): total number of nodes (sometimes written as V)
  - m (or |E|): total number of edges (sometimes written as E)
  - deg(u): degree of node u (how many outgoing edges it has)

Adjacency Matrix

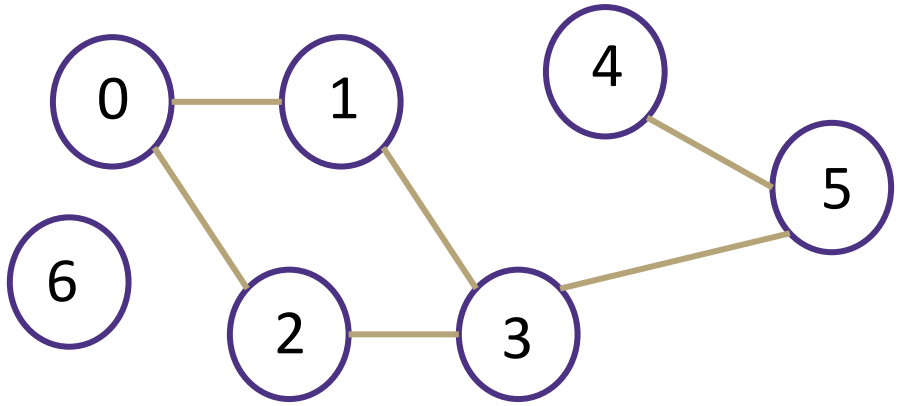
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get out-neighbors of u: O(n)
- Get in-neighbors of u: O(n)

Space Complexity: $O(n^2)$

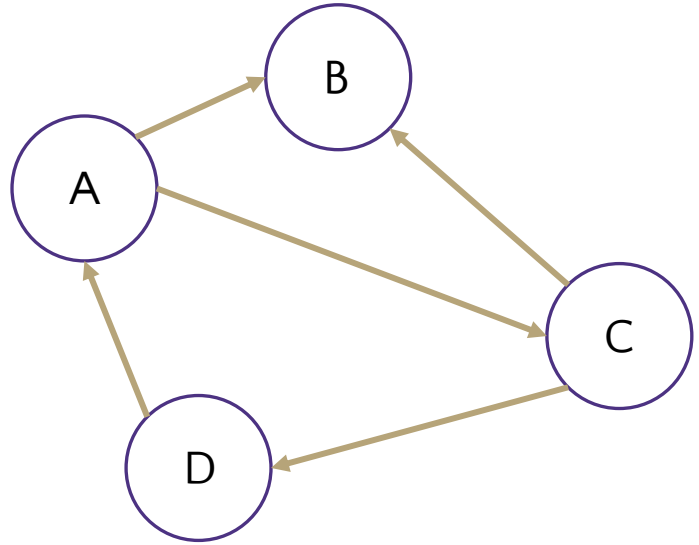
More suitable for dense graphs



	0	1	2	3	4	5	6
0	0	**1**	**1**	0	0	0	0
1	**1**	0	0	**1**	0	0	0
2	**1**	0	0	**1**	0	0	0
3	0	**1**	**1**	0	0	**1**	0
4	0	0	0	0	0	**1**	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

For an undirected graph, adjacency matrix is symmetric w.r.t diagonal line. A node u's out-neighbors are the same as its in-neighbors

Adjacency List



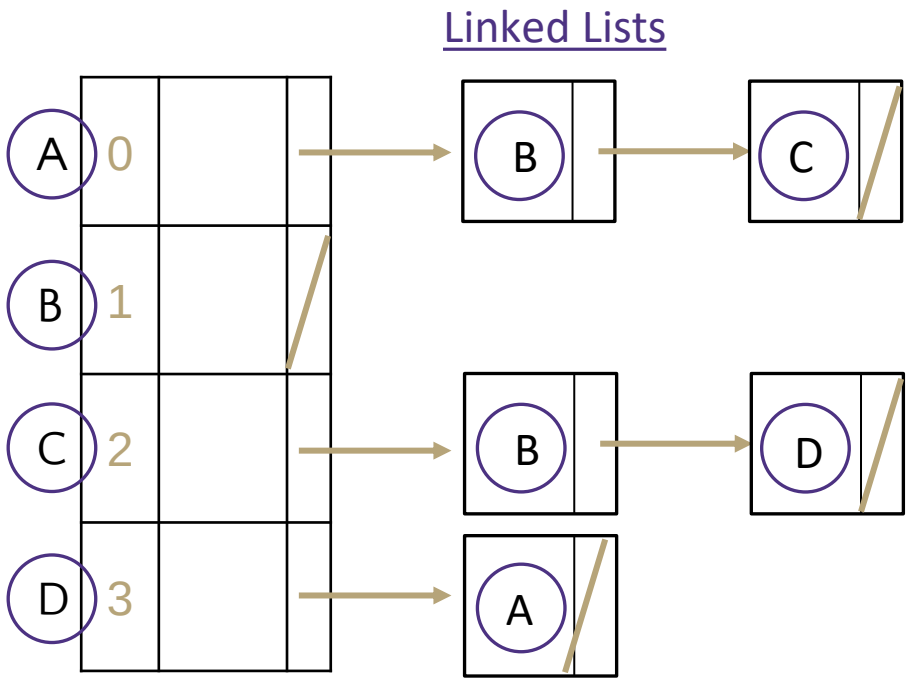
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(deg(u))
- Check edge exists from (u,v): O(deg (u))
- Get outneighbors of u: O(deg(u))
- Get inneighbors of u: O(n + m)

Space Complexity: O(n + m)

More suitable for sparse graphs



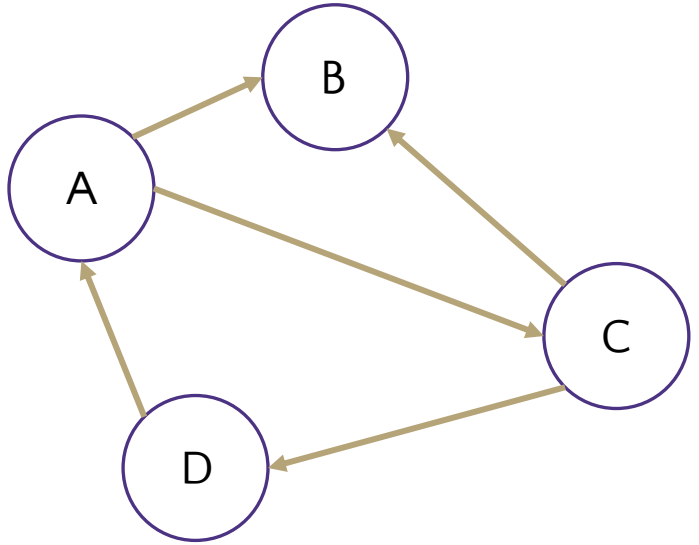
Adjacency List

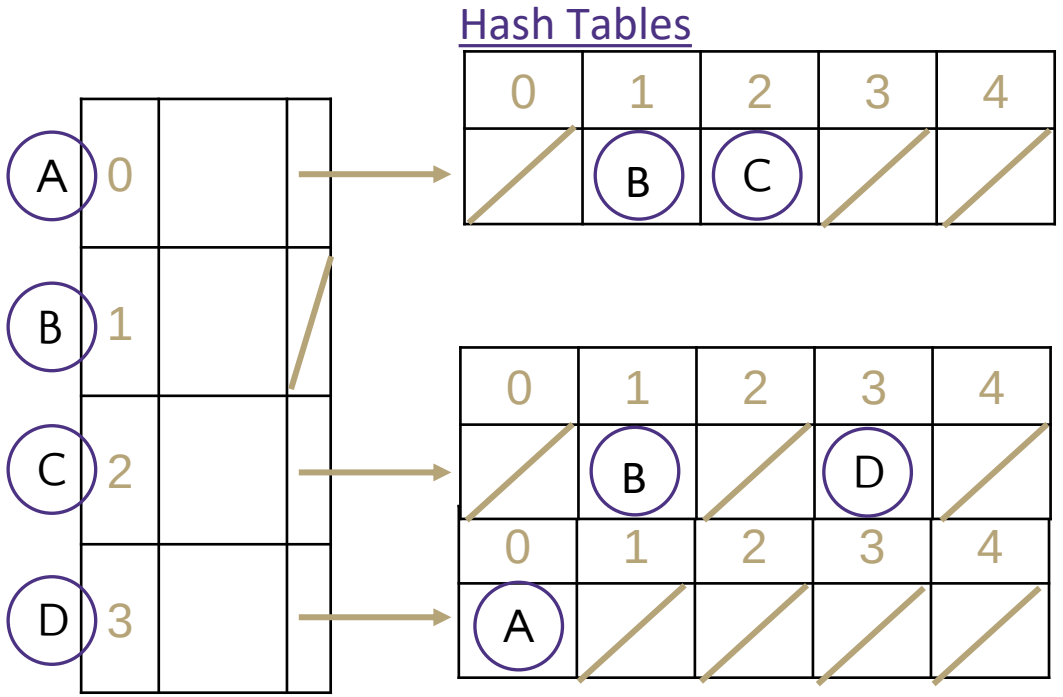
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (assuming a good hash function so all hash table operations are O(1)) (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get outneighbors of u: O(deg(u))
- Get inneighbors of u: O(n)

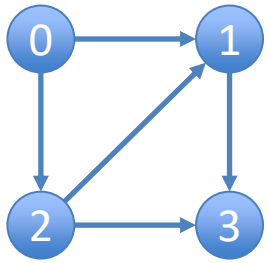
Space Complexity: O(n + m)





2-Hop Neighbors (through Matrix Multiplication)

- Matrix multiplication for finding two-hop neighbors



A graph and its adjacency matrix representation

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	0	0	0

The adjacency matrix representation

	0	1	2	3
0	0	1	0	2
1	0	0	0	0
2	0	0	0	1
3	0	0	0	0

Node 3 is a two-hop neighbor of node 0 along two different paths

Node 3 is a two-hop neighbor of node 2 along 1 path

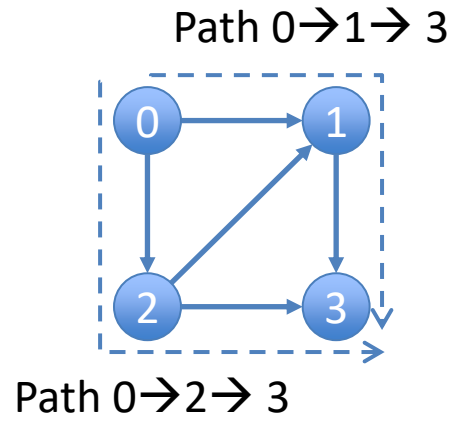
The adjacency matrix representation for two-hop neighbors, obtained by matrix-matrix product of the adjacency matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

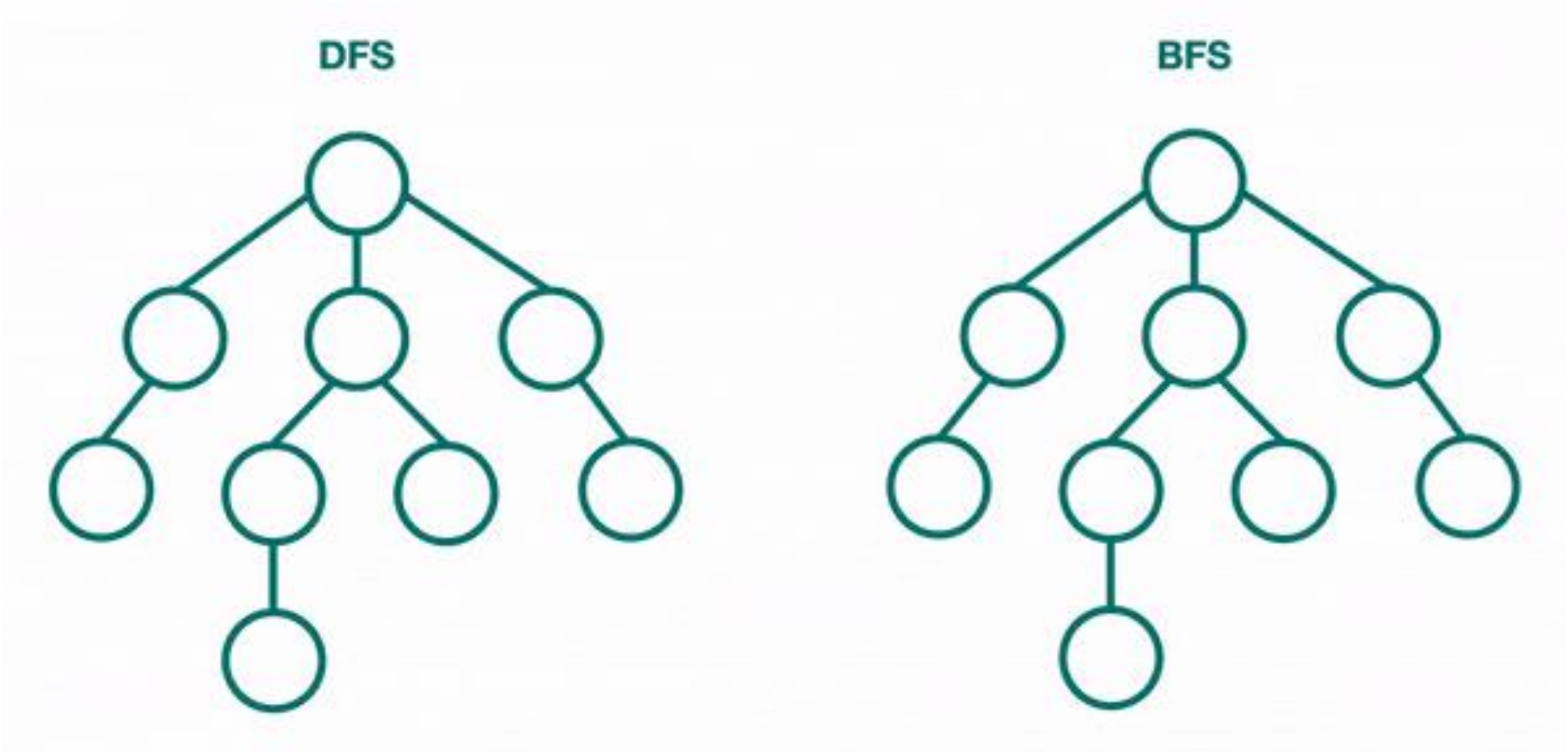
2-Hop Neighbors (through Matrix Multiplication)

- Consider the multiplication of the first row of the left matrix wit the last column of the right matrix:
  - 0*0 + 1*1 + 1*1 + 0*0 = 2.
- This means that there are two 2-hop paths from 1 to 3:
  - Path 0→1→ 3 consisting of two edges 0→1 & 1→ 3, corresponding to the first term of 1*1
  - Path 0→2→ 3 consisting of two edges 0→2 &2→ 3, corresponding to the second term of 1*1

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

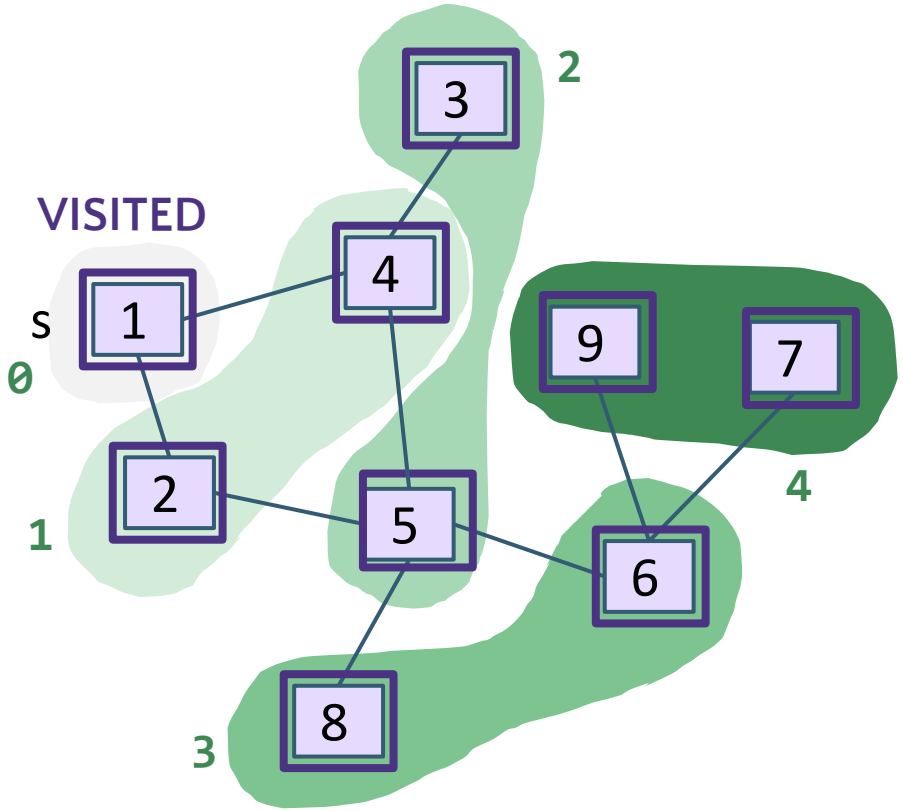


BFS & DFS for Trees



Breadth-First Search (BFS)

BFS: Explore nodes “layer by layer”; like level-order traversal of a tree, now generalized to any graph; visit 1-hop neighbors, then 2-hop neighbors, …, until all nodes have been visited



This is our goal, but how do we translate into code?

- Use a data structure to ”queue up” children…

```
for (Node n : s.neighbors) {
```

BFS Implementation

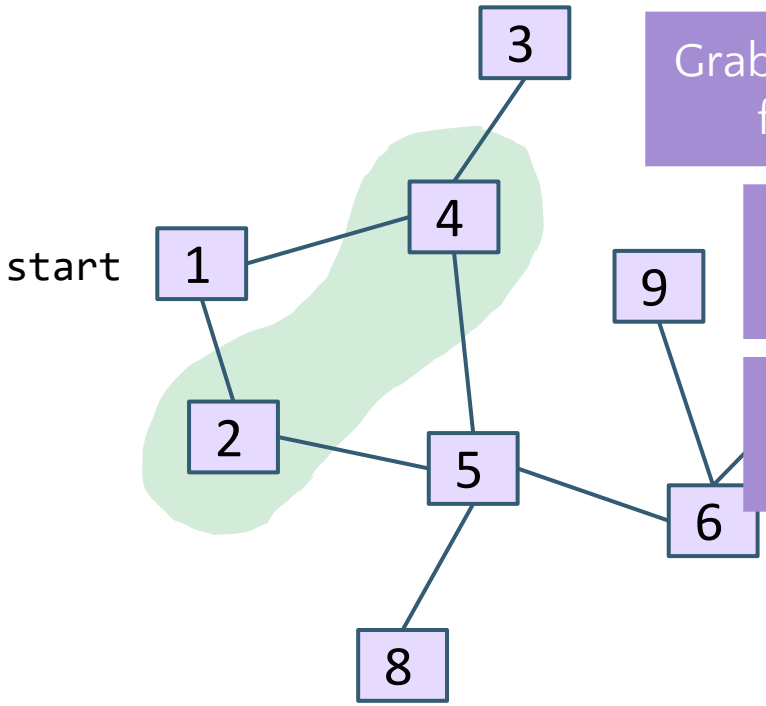
A queue keeps track of "**outer edge**" of nodes still to explore

Kick off the algorithm by adding start to perimeter

Grab one element at a time from the perimeter

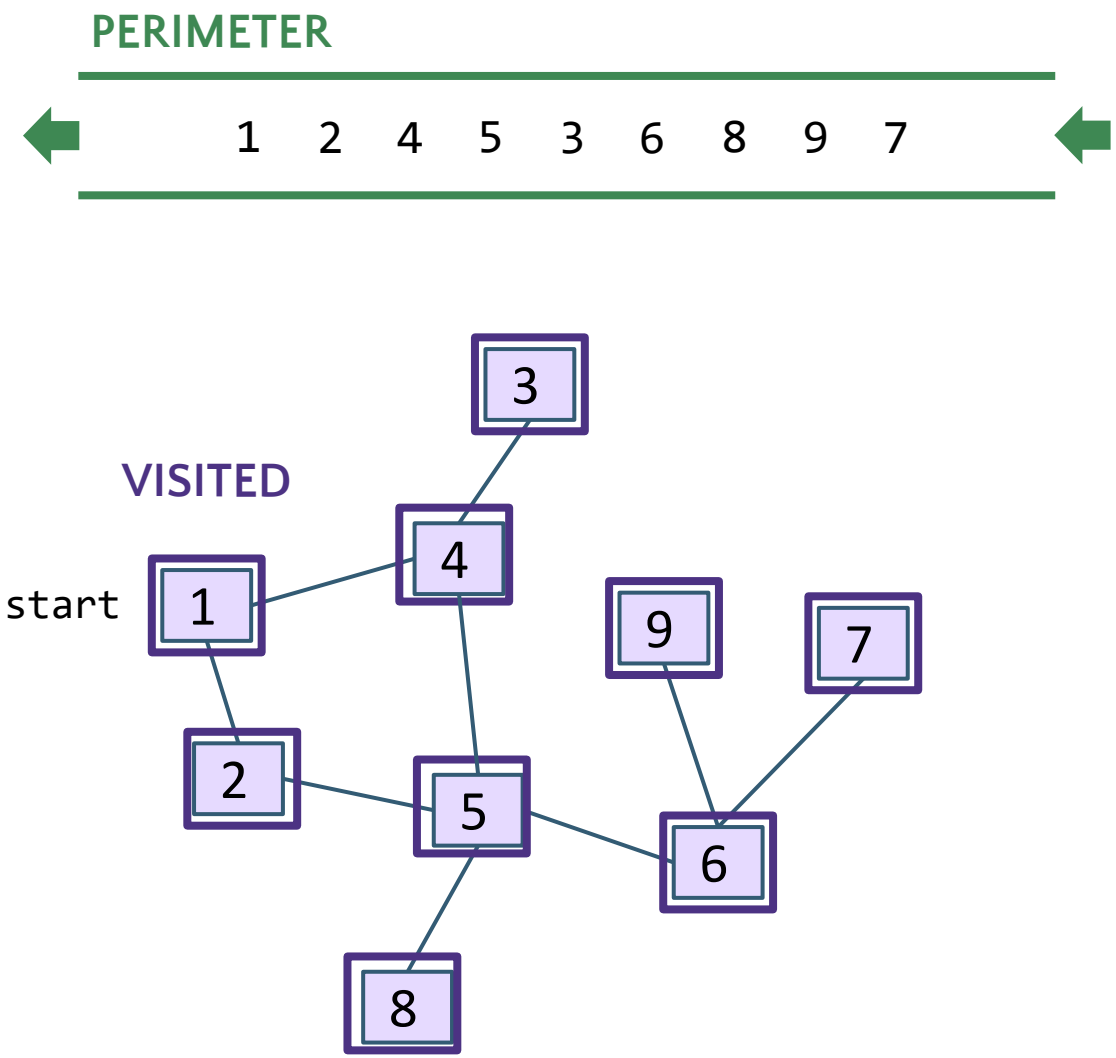
Look at all that element's unvisited children

Add new ones to perimeter!



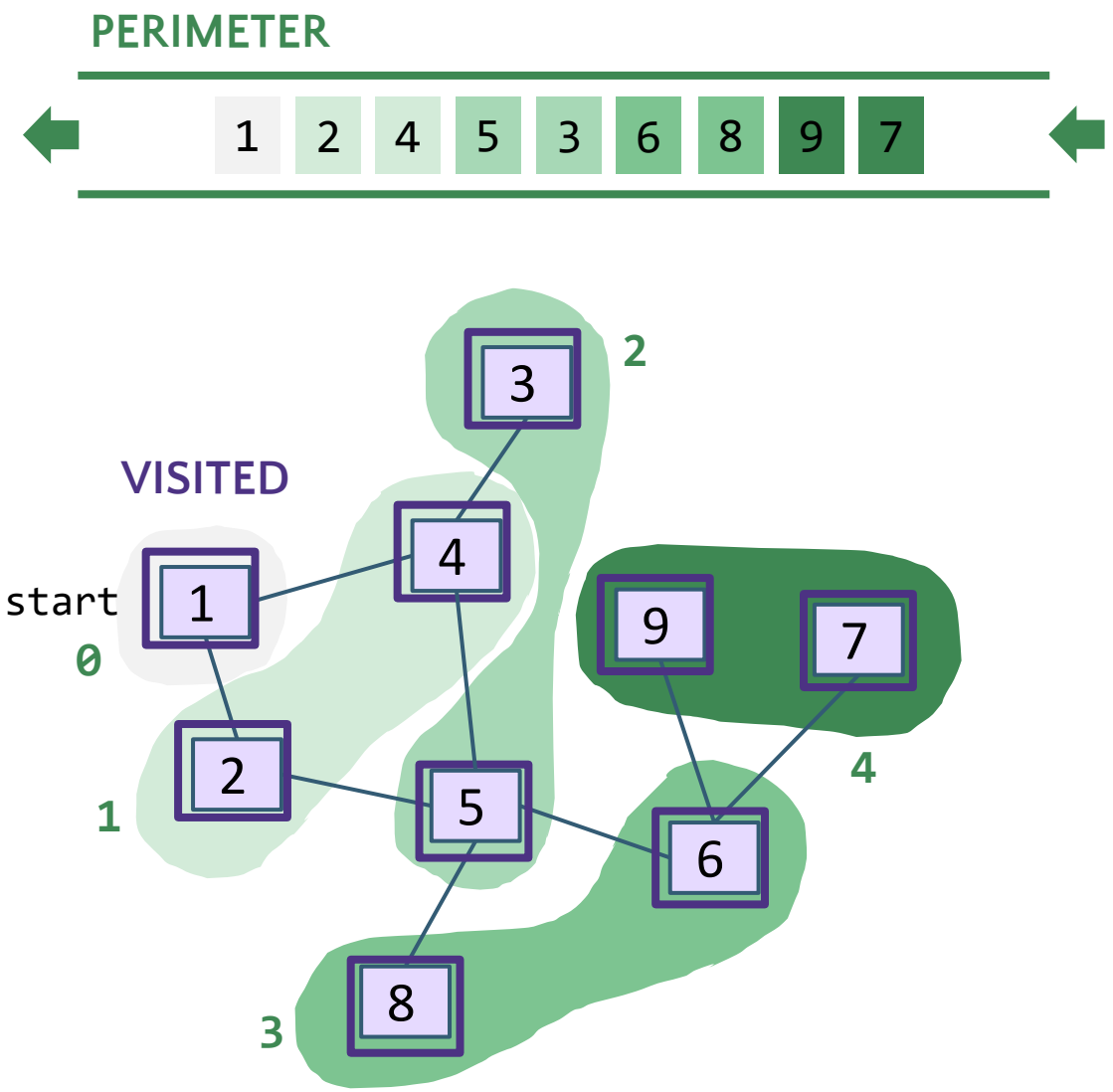

```
bfs(Graph graph, Node start) {
   Queue<Node> perimeter = new Queue<>();
   Set<Node> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
         Node to = edge.to();
         if (!visited.contains(to)) {
           perimeter.add(to);
           visited.add(to);
         }
       }
     }
   }
```

BFS Implementation: In Action




```
bfs(Graph graph, Node start) {
  Queue<Node> perimeter = new Queue<>();
  Set<Node> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Node to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

BFS Intuition: Why Does it Work?

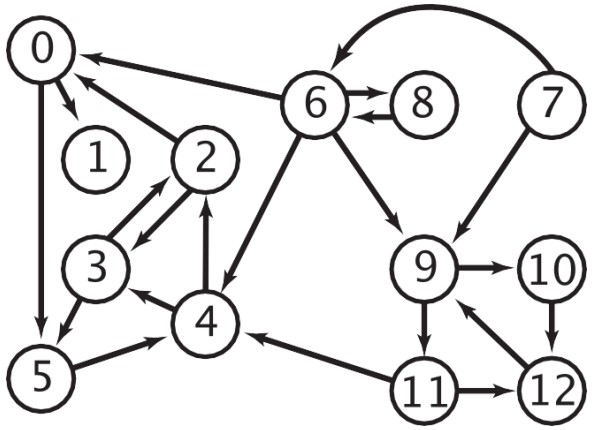


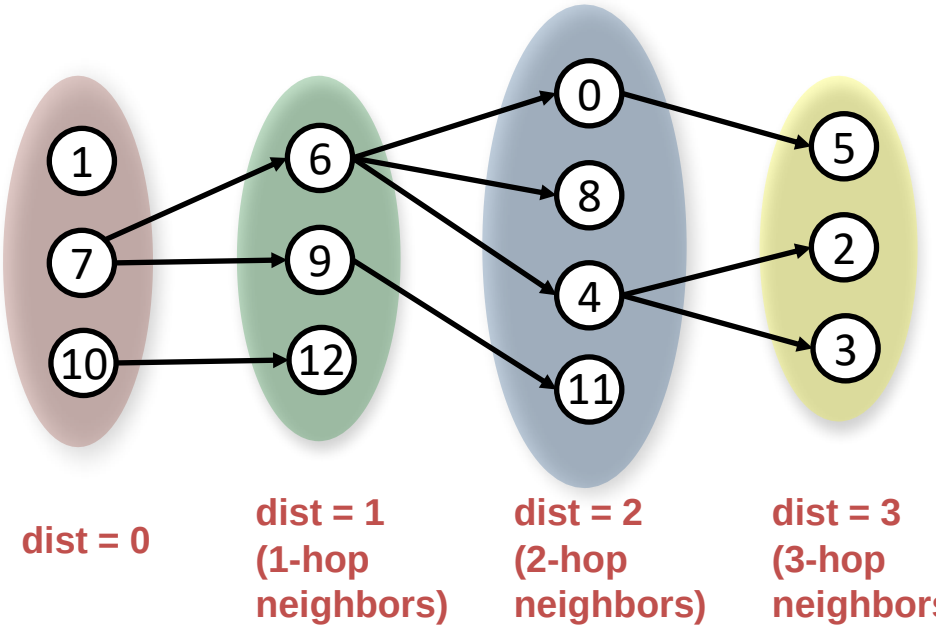

```
bfs(Graph graph, Node start) {
  Queue<Node> perimeter = new Queue<>();
  Set<Node> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Node to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
```

- Using FIFO queue means we explore an entire layer before moving on to the next layer
- Keep going until `perimeter` is empty

BFS Application: Multiple-Source Shortest Paths Problem

- Given a digraph and a set of source nodes, find shortest path from any node in the set to every other node, assuming all edges have weight 1.
  - e.g., S = { 1, 7, 10 }.
  - Shortest path to 4 is 7 → 6 → 4.
  - Shortest path to 5 is 7 → 6 → 0 → 5.
  - Shortest path to 12 is 10 → 12.
- Can be done with BFS, and initialize by enqueuing all source nodes

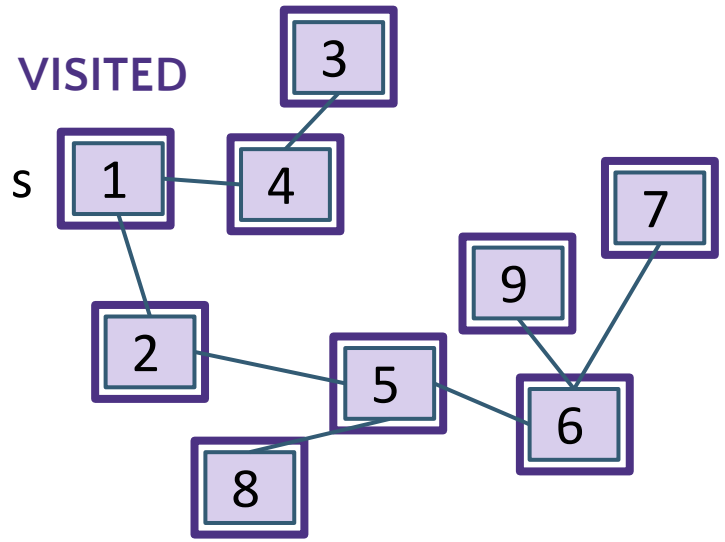




Depth-First Search (DFS) (Recursive Algorithm)

- DFS explores one branch "all the way down" before coming back to try other branches
- Depending on the order of visiting branches, many possible orderings: e.g., {1, 2, 5, 6, 9, 7, 8, 4, 3}, {1, 4, 3, 2, 5, 8, 6, 7, 9}, etc.

```
Set<Node> visited; // assume global
connected(Node s, Node t) {
  if (s == t) {
    return true;
  } else {
    visited.add(s);
    for (Node n : s.neighbors) {
      if (!visited.contains(n)) {
        if (connected(n, t)) {
          return true;
        }
      }
    }
    return false;
  }
}
```



DFS w/ Stack vs. BFS w/ Queue

```
dfs(Graph graph, Node start) {
  Stack<Node> perimeter = new Stack<>();
  Set<Node> visited = new Set<>();

  perimeter.add(start);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
        if (!visted.contains(from)) {
      for (Edge edge:graph.edgesFrom(from)) {
        Node to = edge.to();
                perimeter.add(to)
      }

      visited.add(from);
    }
  }
}
```

```
bfs(Graph graph, Node start) {
  Queue<Node> perimeter = new Queue<>();
  Set<Node> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Node from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Node to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

Recap: Graph Traversals

DFS

(Iterative)

- Follow a "choice" all the way to the end, then come back to revisit other choices
- Uses a **stack**!

BFS

(Iterative)

- Explore layer-by-layer: examine every node at a certain distance from start, then examine nodes that are one level farther
- Uses a **queue**!

DFS

(Recursive)

← On huge graphs, might overflow the call stack

Topological Sort

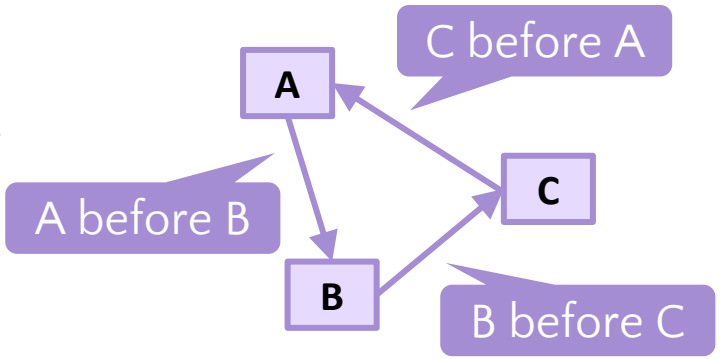
A Directed Acyclic Graph (DAG) : A directed graph (digraph) without any cycles

A DAG encodes a "dependency graph"

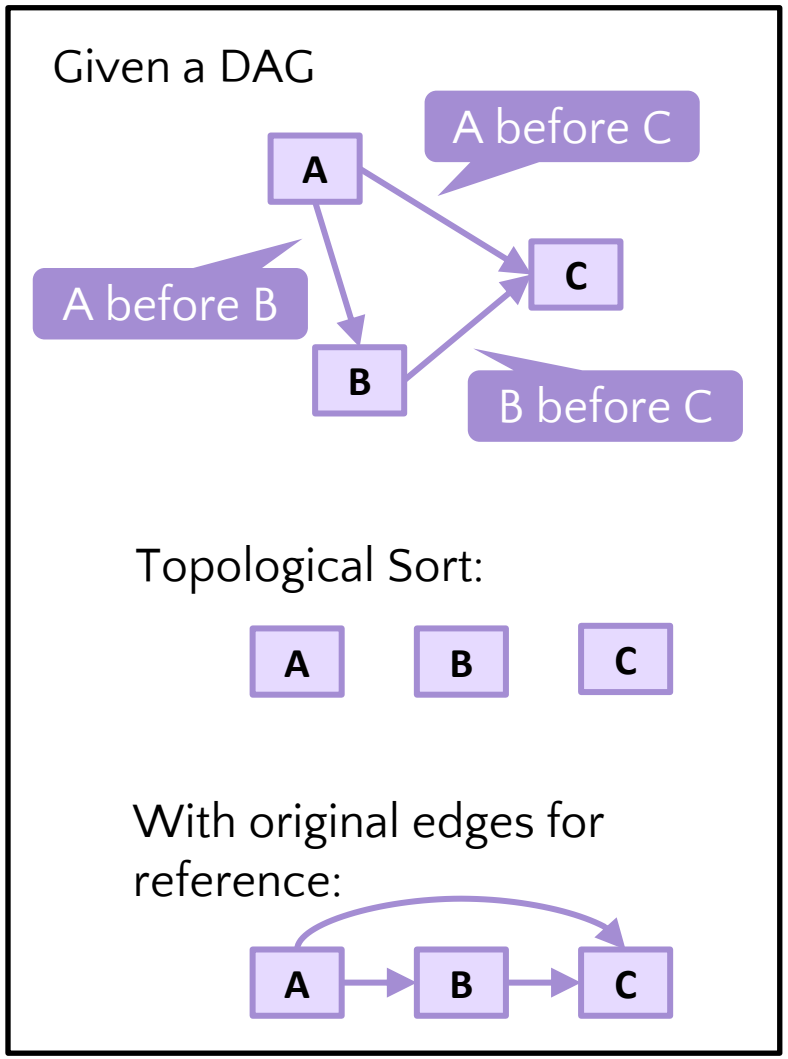
- An edge (u, v) means u must precede v
- A topological sort or topological ordering of a DAG gives a total node ordering that **respects dependencies**

Applications:

- Compiling multiple Java files
- Multi-job Workflows

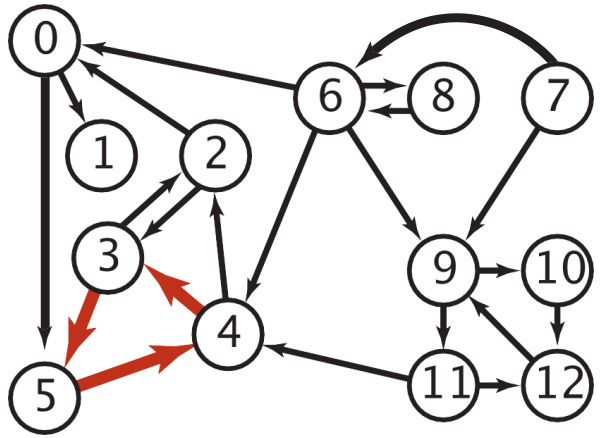


Not a DAG. No possible topological sort



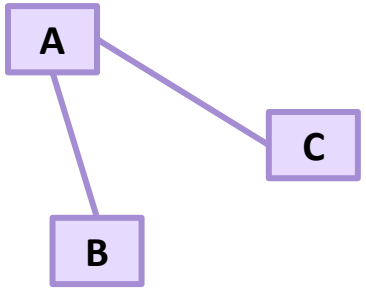
Cycles and Undirected Edges

- Why is topological sort not possible for digraphs containing cycles?
  - Imagine a graph with 3 nodes and edges = {1 to 2 , 2 to 3, 3 to 1} forming a cycle. Now if we try to topologically sort this graph starting from any node, it will always create a contradiction to our definition. All the nodes in a cycle are indirectly dependent on each other hence topological sortfails

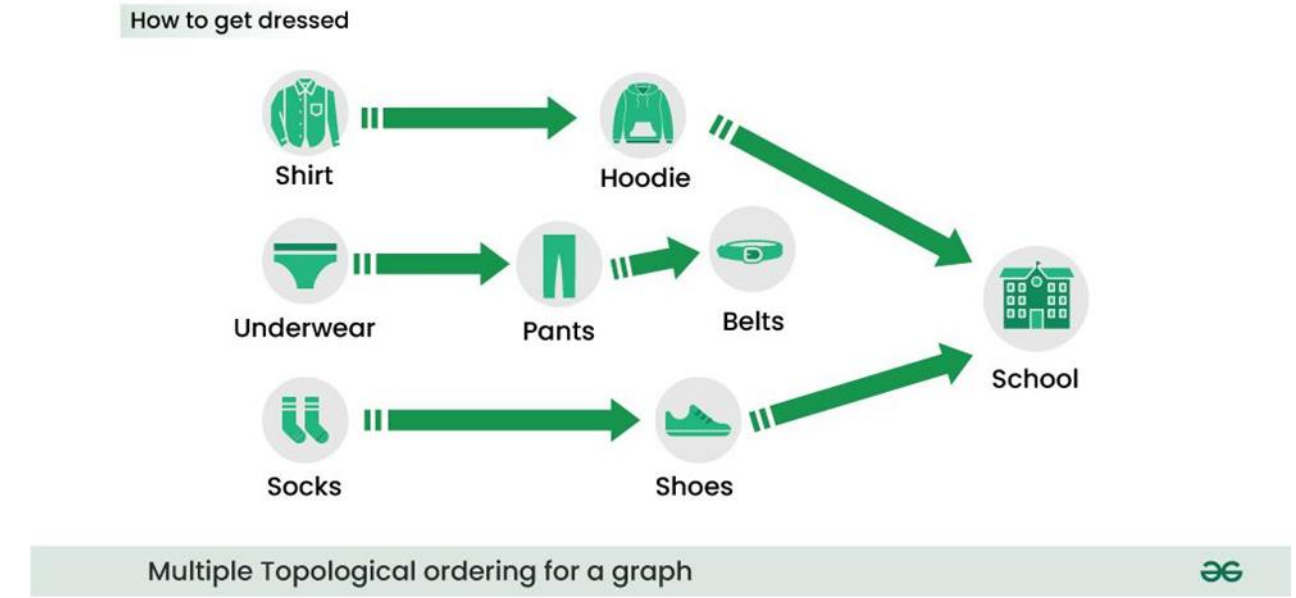


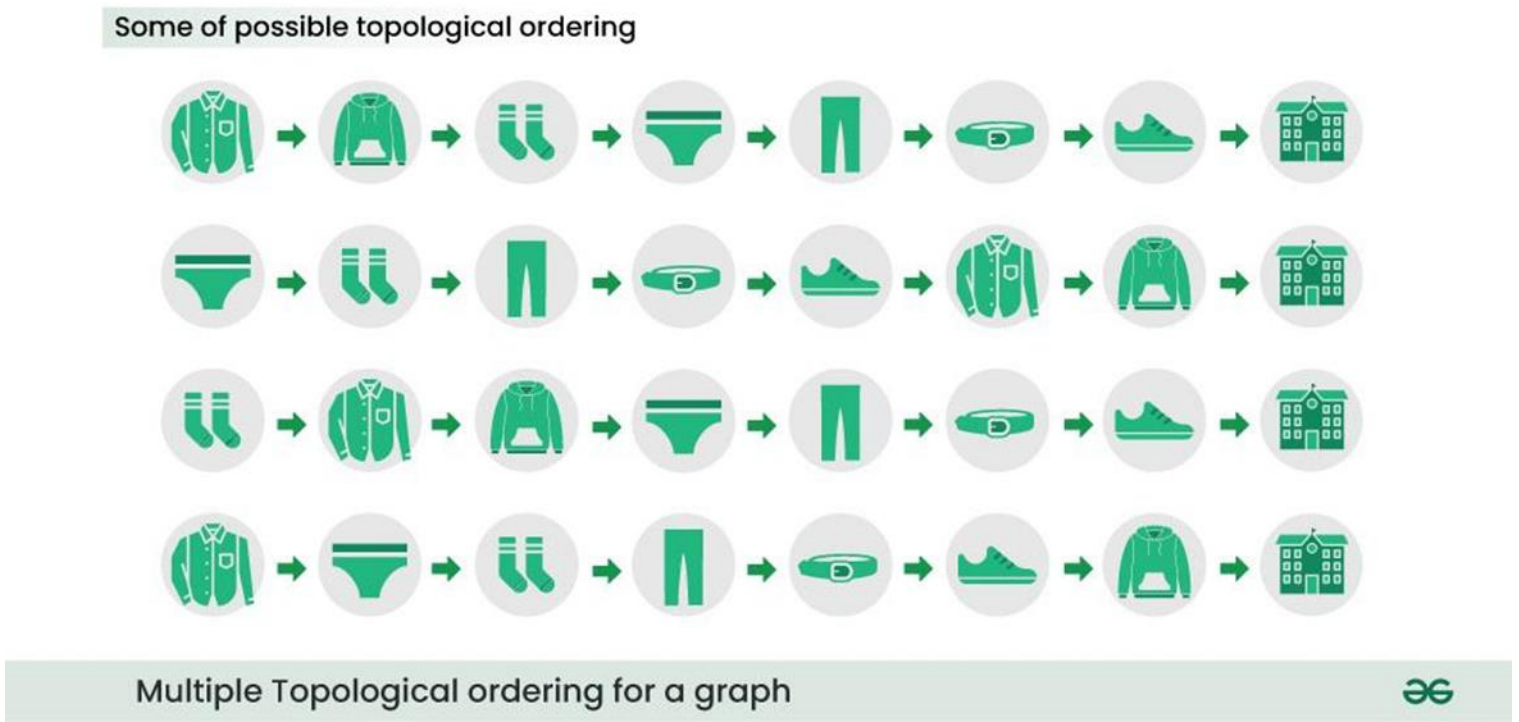
a digraph with a directed cycle

- Why is topological sort not possible for graphs with undirected edges?
  - Special case of a cycle. An undirected edge between two nodes u and v means, there is an edge from u to v as well as from v to u. Because of this both the nodes u and v depend upon each other and none of them can appear before the other in the topological sort without creating a contradiction



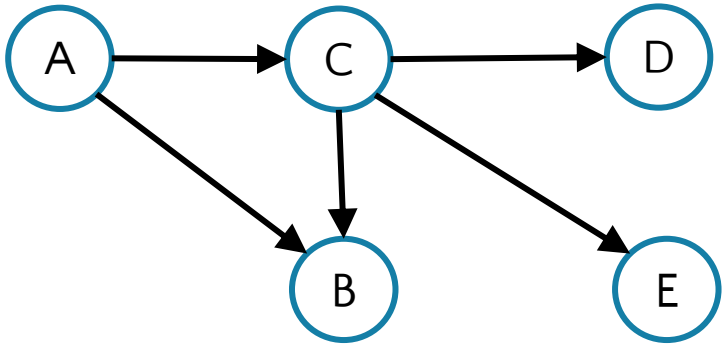
Topological Sort Example I





Topological Sort Example II

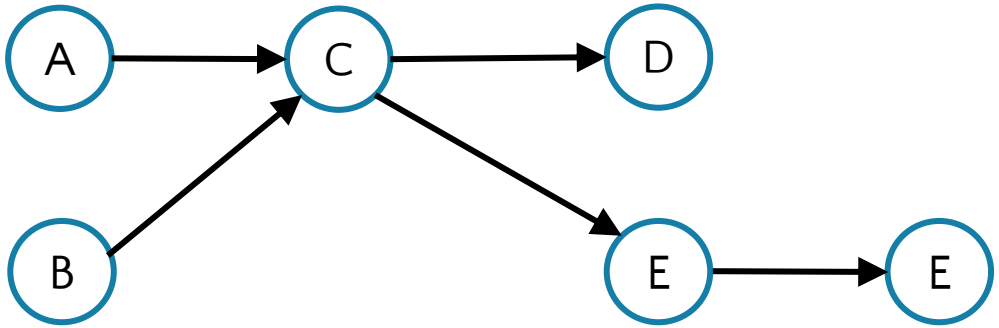
Give all possible topological orderings of this DAG



By observation, we can get (A, C, B, D, E), (A, C, B, E, D)

Topological Sort Example III

Give all possible topological orderings of this DAG



By observation, we can get: all possible interleavings with (AB) or (BA) before C, (DEF) or (DFE) or (EFD) after C. e.g., ABCDEF, BACEDF, etc.

DFS Traversals and Topological Sort

- DFS pre-order traversal
  - Visit a node during DFS forward traversal before visiting all its unvisited neighbors
  - .Pre-order traversal is obtained in the order that nodes are pushed onto the stack, or when the recursive function call goes forward in the call stack
  - Upon finishing traversal starting from one node, restart from another unvisited node
- DFS post-order traversal
  - Visit a node during DFS backtracking after visiting all its unvisited neighbors, i.e., after reaching a deadend.
  - Post-order traversal is obtained in the order that the nodes are popped off the stack, or when the recursive function call returns from the call stack
  - Upon finishing traversal starting from one node, restart from another unvisited node
  - Any post-order traversal of a graph must end with a node with incoming edges (no predecessors), hence any topological sort must start with a node with no predecessors
    - Tree traversal is a special case: any post-order traversal of a tree must end with the root
- Topological sort:
  - Perform DFS post-order traversal starting from any node (often, but not necessarily, a node with no predecessors, to reduce the number of restarts) to get an ordered list of nodes, then reverse the node order to get a topological sort (one of multiple possible)
  - Intuition: DFS post-order traversal outputs nodes from the deepest (farthest away from the starting node) to the starting mode, hence the reverse order is a topological sort from the starting node

DFS Traversal of Graphs: Pre-order, Post-order

```
function preOrderTraversal(node) {
  if (node !== null) {
    visitNode(node);
    preOrderTraversal(node.left);
    preOrderTraversal(node.right);
  }
}
```

```
function postOrderTraversal(node) {
  if (node !== null) {
    postOrderTraversal(node.left);
    postOrderTraversal(node.right);
    visitNode(node);
  }
}
```

Recall: Binary Tree traversal with DFS: pre-order, post-order

```
function preOrderTraversal(node) {
  if (node !== null) {
    visitNode(node);
    foreach(c ∈ node.UnvisitedNeighbors) {
        preOrderTraversal(c);}
    }
}
```

```
function postOrderTraversal(node) {
  if (node !== null) {
        foreach(c ∈ node.UnvisitedNeighbors) {
            postOrderTraversal(c);}
        visitNode(node);
  }
}
```

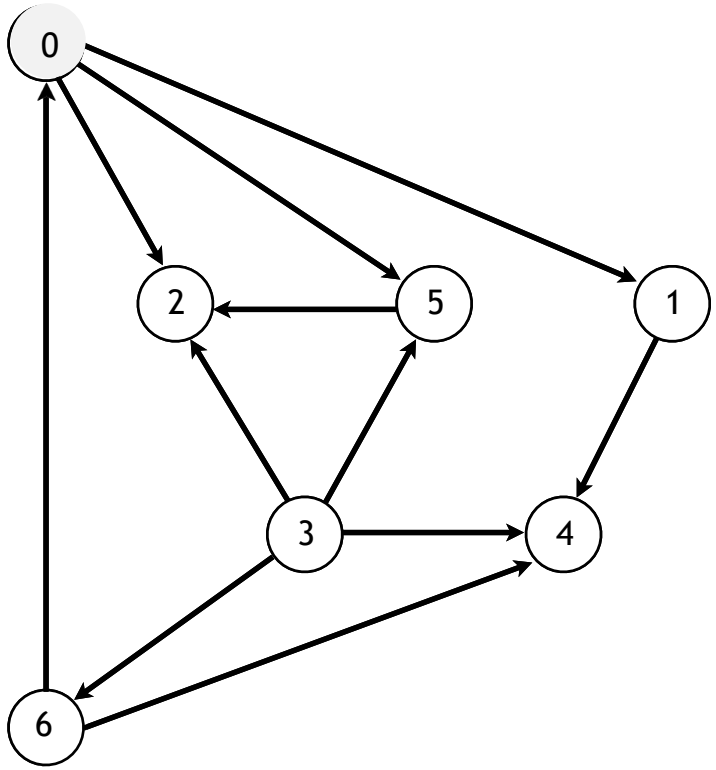
The traversal algorithms works for both undirected and directed graphs. The only difference is how to get neighbors of node v, as each undirected edge is treated as two directed edges in both directions.

Resolving Ambiguities

- As There are typically multiple possible traversals of the same graph. In the lecture and exams, we often use the following rule to resolve any ambiguities:
- “When there are multiple possible orders of visiting the next node, select the next node in alphabetical or numerical order.”

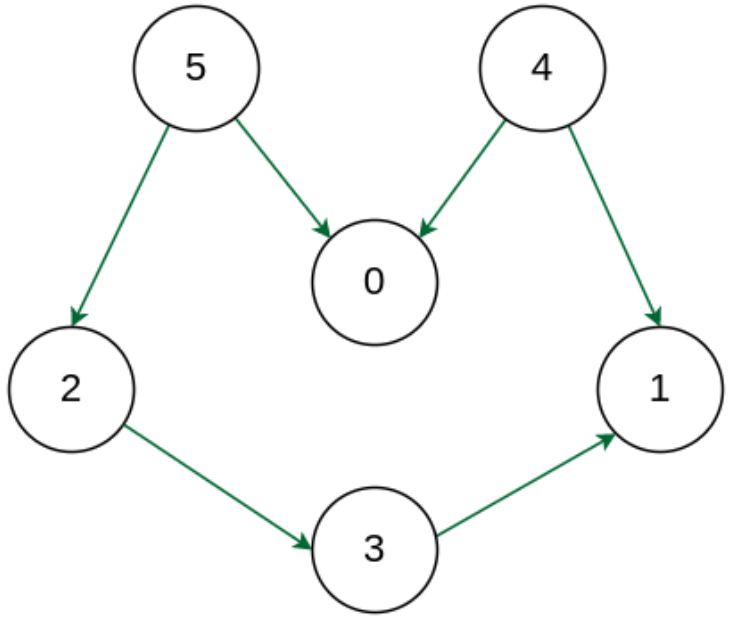
DFS Traversal Example I

- Start from node 0
- Visit nodes 0->1->4. Since 4 has no successors, backtrack to 0.
  - Pre-order: (0, 1, 4); post-order: (4, 1)
- Visit node 2. Since 2 has no successors, backtrack to 0
  - Pre-order: (0, 1, 4, 2); post-order: (4, 1, 2)
- Visit node 5. Since 5's successor 2 has been visited, backtrack to 0. Since all of node 0's successors have been visited, we visit it in post-order
  - Pre-order: (0, 1, 4, 2, 5); post-order: (4, 1, 2, 5, 0)
- Restart from node 3, visit its successor 6, then backtrack. Since all of node 3's successors have been visited, we visit it in post-order
  - Pre-order: (0, 1, 4, 2, 5, 3, 6); post-order: (4, 1, 2, 5, 0, 6, 3)
- All nodes and their successors have been visited, so the algorithm terminates. A topological sort corresponding to this post-order is (3, 6, 0, 5, 2, 1, 4)
- A BFS traversal: (0, 2, 5, 1, 4, 3, 6)
- This is one of many possible traversals, e.g., if we start from node 3, then we have the traversals: pre-order: (3, 2, 4, 5, 6, 0, 1); post-order: (2, 4, 5, 1, 0, 6, 3); Topological sort: (3, 6, 0, 1, 5, 4, 2); BFS: (3, 2, 4, 5, 6, 0, 1)



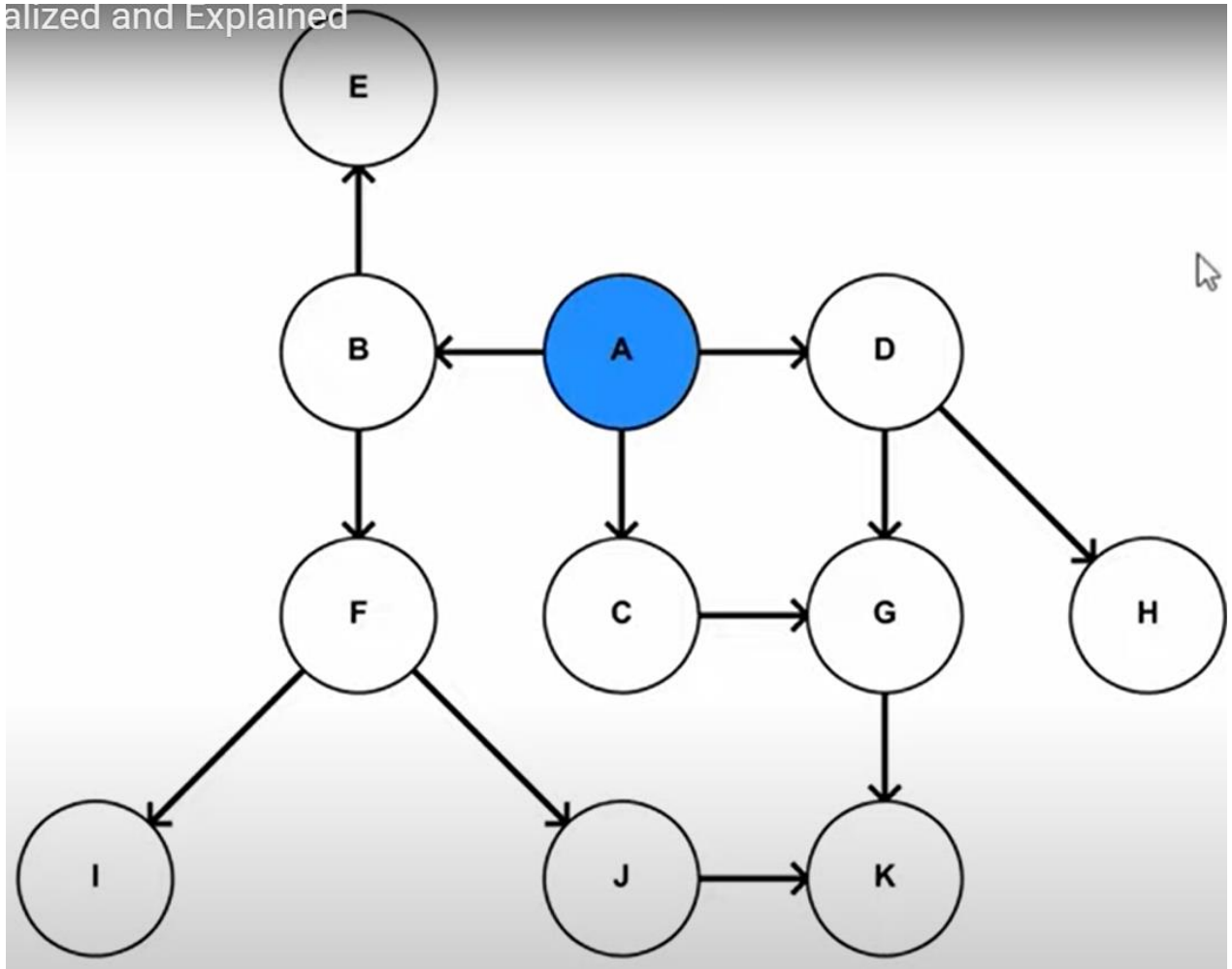
DFS Traversal Example II

- Starting from node 5:
  - Pre-order traversal: (5, 2, 3, 1, 0, 4)
  - Post-order traversal: (1, 3, 2, 0, 5, 4)
  - Topological sort: (4, 5, 0, 2, 3, 1)
  - BFS traversal: (5, 0, 2, 3, 1, 4)
- Starting from node 4:
  - Pre-order traversal: (4, 0, 1, 5, 2, 3)
  - Post-order traversal: (0, 1, 4, 3, 2, 5)
  - Topological sort: (5, 2, 3, 4, 1, 0)
  - BFS traversal: (4, 0, 1, 5, 2, 3)
- Starting from node 0:
  - Pre-order traversal: (0, 5, 2, 3, 1, 4)
  - Post-order traversal: (0, 1, 3, 2, 5, 4)
  - Topological sort: (4, 5, 2, 3, 1, 0)
  - BFS traversal: (0, 5, 2, 3, 1, 4)
- You may try starting from any other node.



DFS Traversal Example III

- Starting from node A:
- Pre-order traversal: (A, B, F, I, J, K, E, C, G, D, H)
- Post-order traversal: (I, K, J, F, E, B, G, C, H, D, A)
- Topological sort: (A, D, H, C, G, B, E, F, J, K, I)
- BFS traversal: (A, B, C, D, E, F, G, H, I, J, K)
- Starting from a different node will give a different topological sort, but all of them must start with A, since it is the only node without any predecessors (i.e., it must precede all the other nodes based on the DAG)
- Any post-order traversal must visit A last, since all of A's sucessors must be visited before visiting A

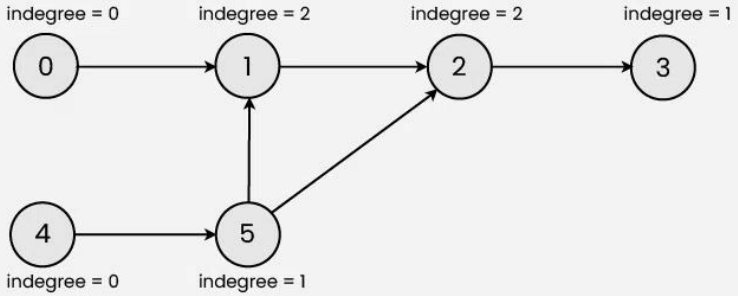


Topological Sort Visualized and Explained
https://www.youtube.com/watch?v=7J3GadLzydI

Kahn's algorithm for Topological Sort

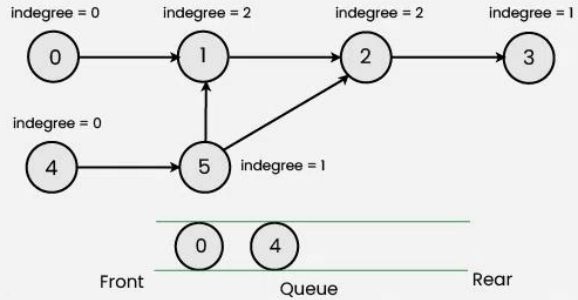
- The algorithm works by repeatedly finding nodes with no incoming edges, removing them from the graph, and updating the incoming edges of the remaining nodes. This process continues until all nodes have been ordered.
  - Add all nodes with in-degree 0 to a queue.
  - While the queue is not empty:
    - Remove a node from the queue.
    - For each outgoing edge from the removed node, decrement the in-degree of the destination node by 1.
    - If the in-degree of a destination node becomes 0, add it to the queue.
  - If the queue is empty and there are still nodes in the graph, the graph contains a cycle and cannot be topologically sorted.
  - The nodes in the queue represent the topological sort of the graph.
- Time Complexity: O(V+E).
  - The outer for loop will be executed V number of times and the inner for loop will be executed E number of times.

01 Step | Calculate the indegree of all the nodes



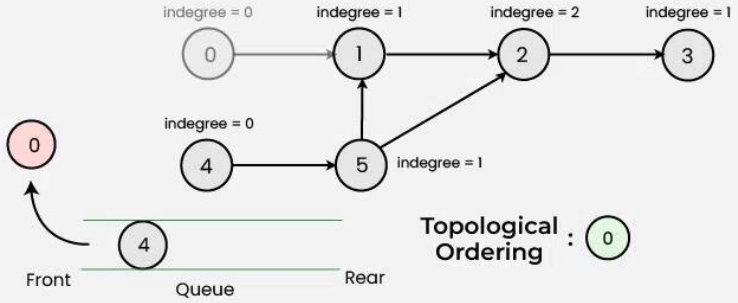
Kahn's Algorithm for Topological Sorting

02 Step | Enqueue nodes having indegree = 0 (node 0 & 4)



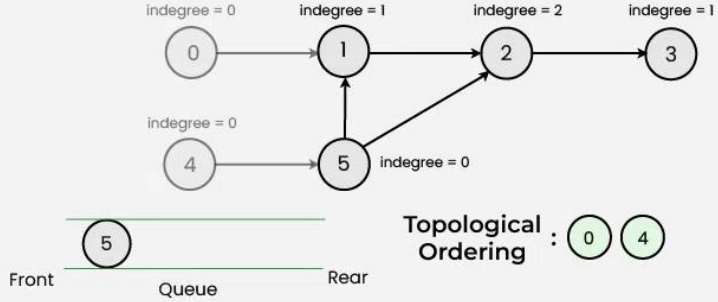
Kahn's Algorithm for Topological Sorting

03 Step | Dequeue element at front (node 0) & decrement indegree of neighboring nodes



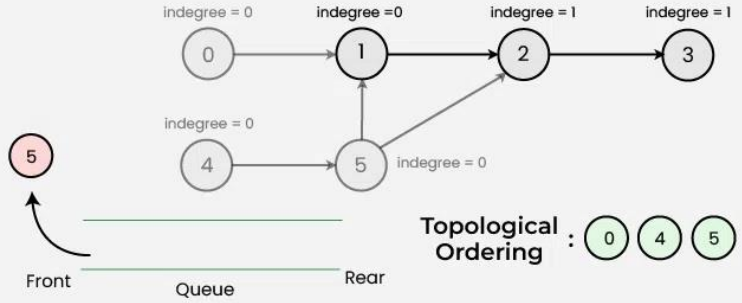
Kahn's Algorithm for Topological Sorting

05 Step | Enqueue neighboring nodes having indegree = 0 (node 5)



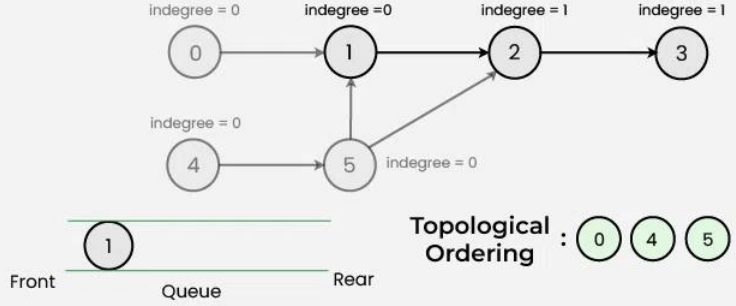
Kahn's Algorithm for Topological Sorting

06 Step | Dequeue element at front (node 5) & decrement indegree of neighboring nodes

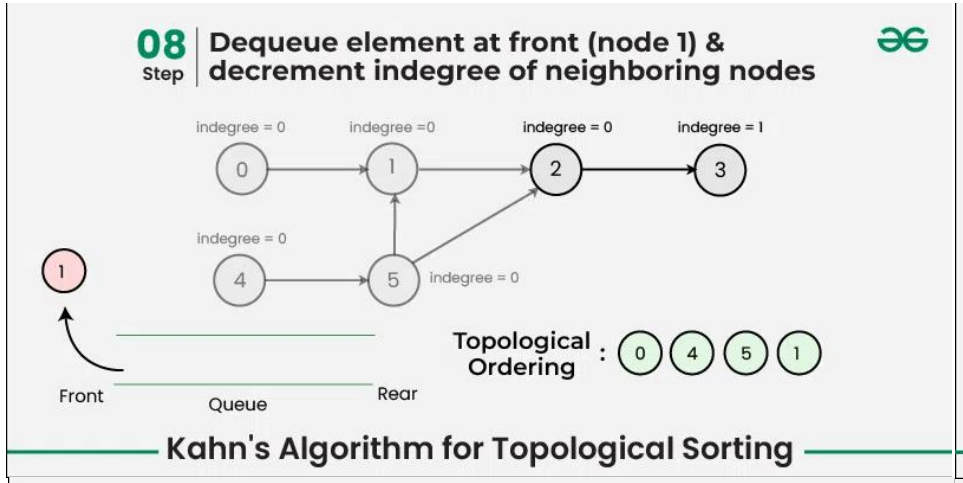


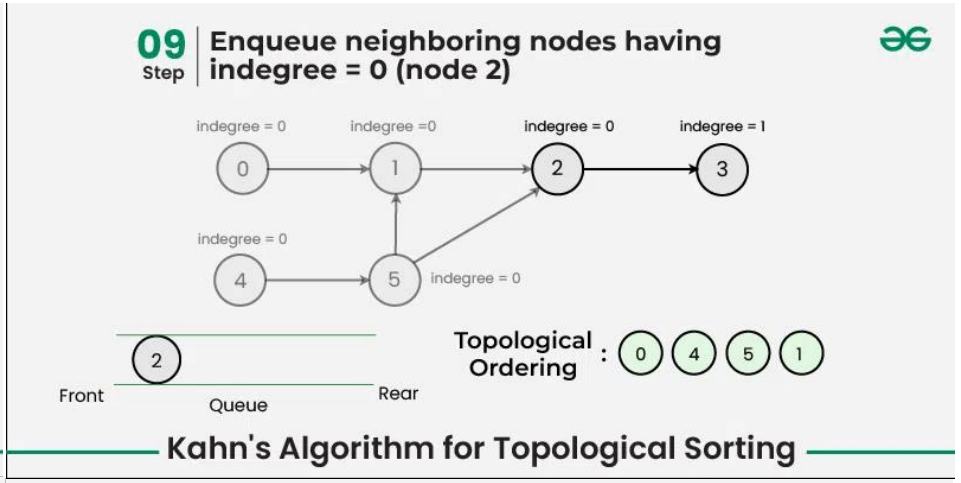
Kahn's Algorithm for Topological Sorting

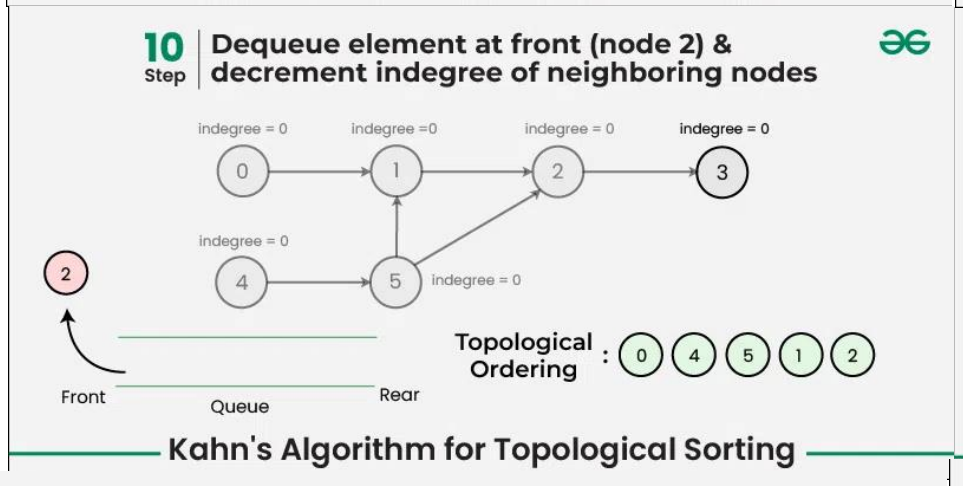
07 Step | Enqueue neighboring nodes having indegree = 0 (node 1)

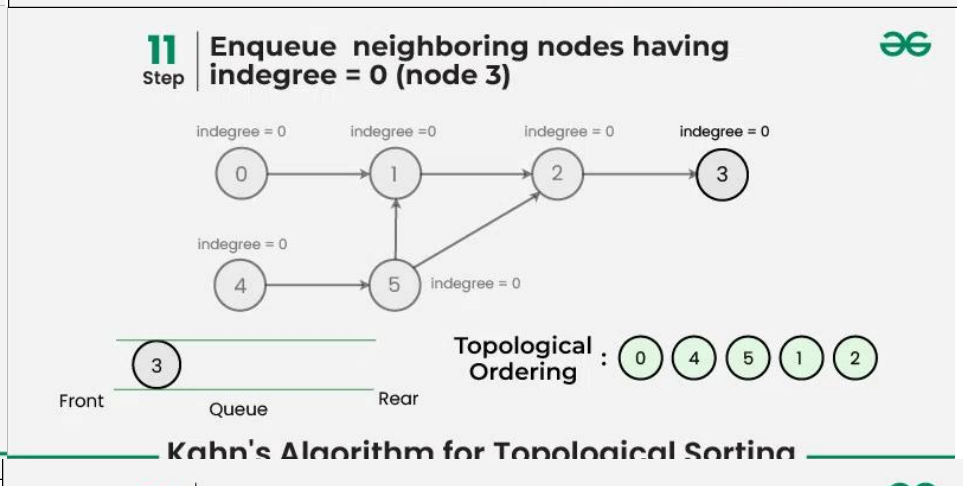


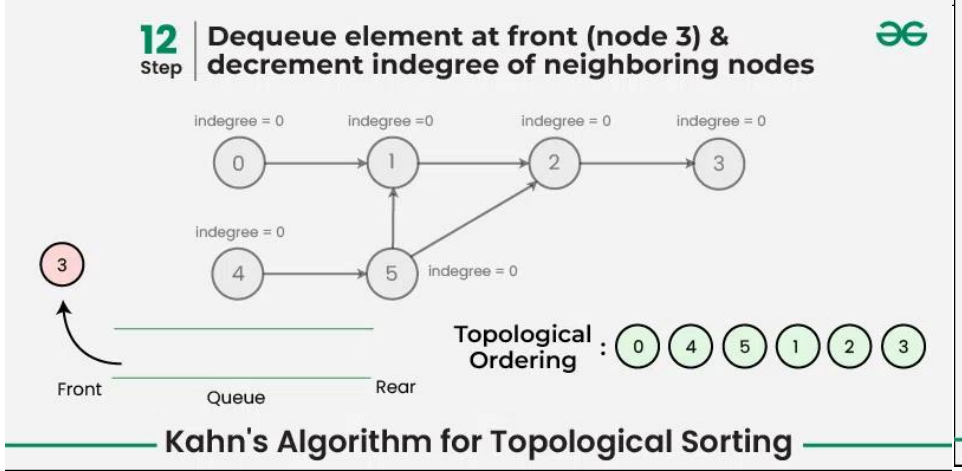
Kahn's Algorithm for Topological Sorting

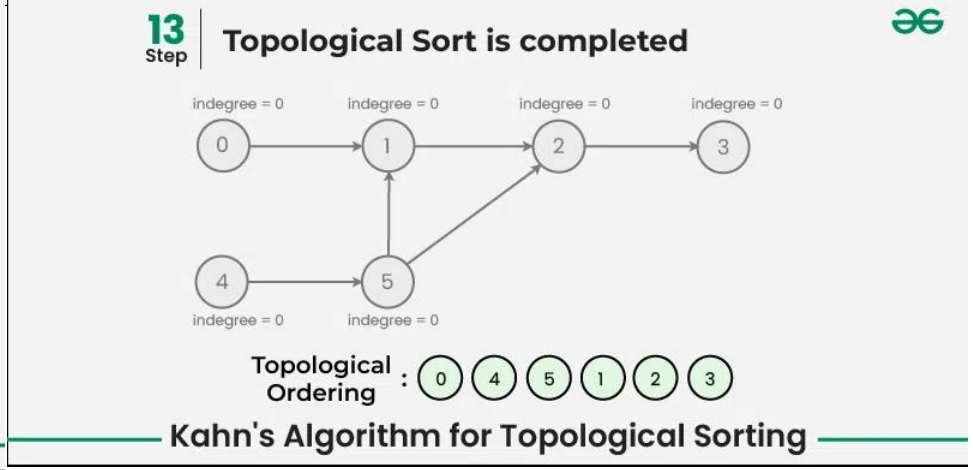

08 Step
Dequeue element at front (node 1) & decrement indegree of neighboring nodes
indegree = 0
indegree =0
indegree = 0
indegree = 1
indegree = 0
indegree = 0
1
Front
Queue
Rear
Topological Ordering : 0 4 5 1
Kahn's Algorithm for Topological Sorting


09 Step
Enqueue neighboring nodes having indegree = 0 (node 2)
indegree = 0
indegree =0
indegree = 0
indegree = 1
indegree = 0
indegree = 0
2
Front
Queue
Rear
Topological Ordering : 0 4 5 1
Kahn's Algorithm for Topological Sorting


10 Step
Dequeue element at front (node 2) & decrement indegree of neighboring nodes
indegree = 0
indegree =0
indegree = 0
indegree = 0
indegree = 0
indegree = 0
2
Front
Queue
Rear
Topological Ordering : 0 4 5 1 2
Kahn's Algorithm for Topological Sorting

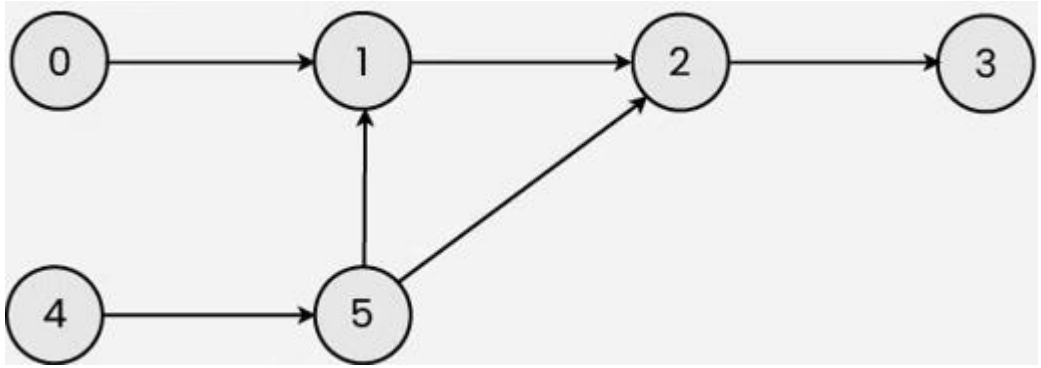

11 Step
Enqueue neighboring nodes having indegree = 0 (node 3)
indegree = 0
indegree =0
indegree = 0
indegree = 0
indegree = 0
indegree = 0
3
Front
Queue
Rear
Topological Ordering : 0 4 5 1 2
Kahn's Algorithm for Topological Sorting


12 Step
Dequeue element at front (node 3) & decrement indegree of neighboring nodes
indegree = 0
indegree =0
indegree = 0
indegree = 0
indegree = 0
indegree = 0
3
Front
Queue
Rear
Topological Ordering : 0 4 5 1 2 3
Kahn's Algorithm for Topological Sorting


13 Step
Topological Sort is completed
indegree = 0
indegree = 0
indegree = 0
indegree = 0
indegree = 0
indegree = 0
Topological Ordering : 0 4 5 1 2 3
Kahn's Algorithm for Topological Sorting

Graph Traversals

- Give one DFS pre-order, in-order and post-order traversal of the following graph, and a topological sort.
- Starting from node 4:
  - Pre-order traversal: (4, 5, 1, 2, 3, 0)
  - Post-order traversal: (3, 2, 1, 5, 4, 0)
  - Topological Sort: (0, 4, 5, 1, 2, 3)
  - BFS: (4, 5, 1, 2, 3, 0)
- Starting from node 0:
  - Pre-order traversal: (0, 1, 2, 3, 4, 5)
  - Post-order traversal: (3, 2, 1, 0, 5, 4)
  - Topological Sort: (4, 5, 0, 1, 2, 3)
  - BFS: (0, 1, 2, 3, 4, 5)



References

- Breadth-first search in 4 minutes (for a tree)
  - https://www.youtube.com/watch?v=HZ5YTanv5QE
- Depth-first search in 4 minutes (for a tree)
  - https://www.youtube.com/watch?v=Urx87-NMm6c
- Graph Traversals - Breadth First and Depth First (for an undirected graph)
  - https://www.youtube.com/watch?v=bIA8HEEUxZI
- Breadth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=N6wicLpEmHY&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=5
- Depth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=5GcSvYDgiSo&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=6
- Topological Sort Visualized and Explained
  - https://www.youtube.com/watch?v=7J3GadLzydI&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=7
- Graph Algorithms, Programming and Math Tutorials (Playlist)
  - https://www.youtube.com/playlist?list=PLj8W7XIvO93oxLOZTi8JFghuRcKieIZU-

Full-Length Lectures

- [CSE 373 WI24] Lecture 14: Graph Traversals
  - https://www.youtube.com/watch?v=1IJUv3ljqyU&list=PLEcoVsAaONjd5n69K84sSmAuvTrTQT_Nl&index=13