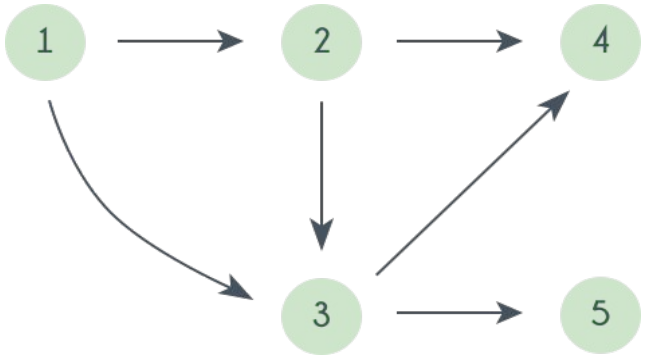
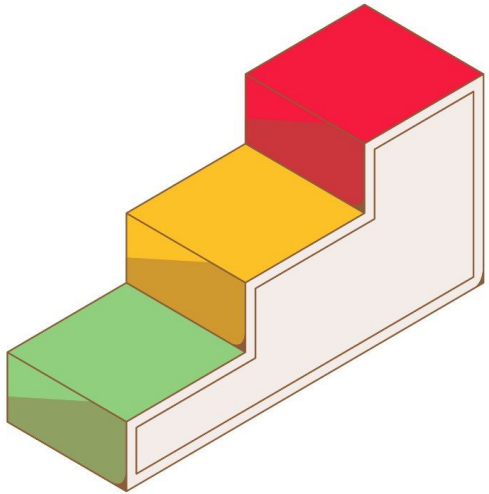
# Topological Sort



# Lecture Flow

1) Pre-requisites
2) Understanding the Logic
3) Different Implementations
4) Recognizing in Questions
5) Applications of Topological sort
6) Common Pitfalls
7) Quote of the day

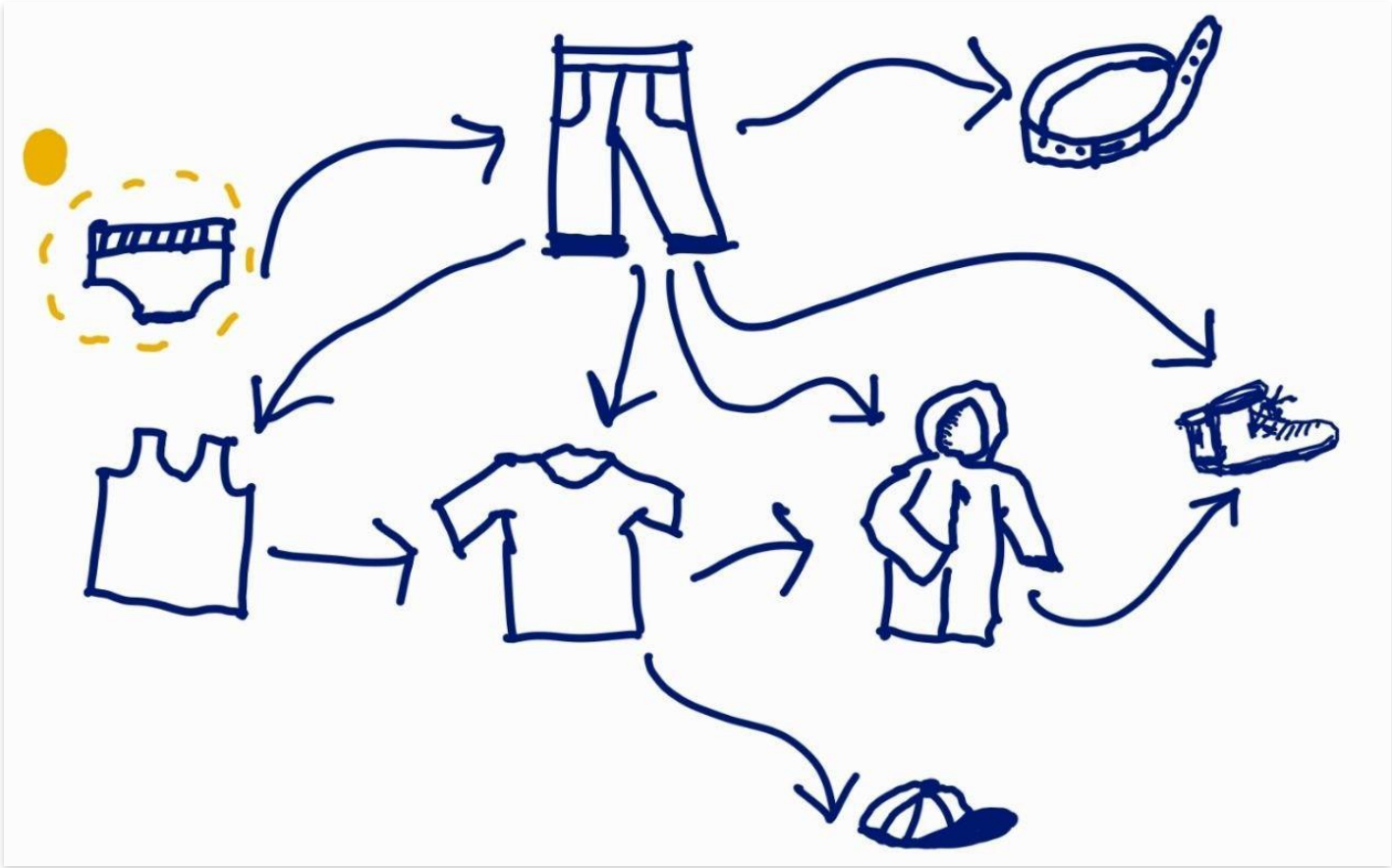


# Pre-requisites

- Graph theory
- Cycles in graphs
- Graph traversal (BFS, DFS)
- Backtracking

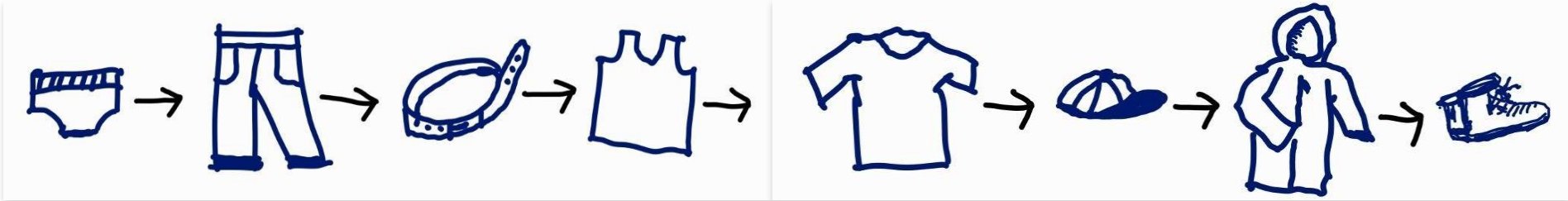
# Understanding the Logic

# Real Life Example



# Order of Clothes

You wear your clothes **one by one**



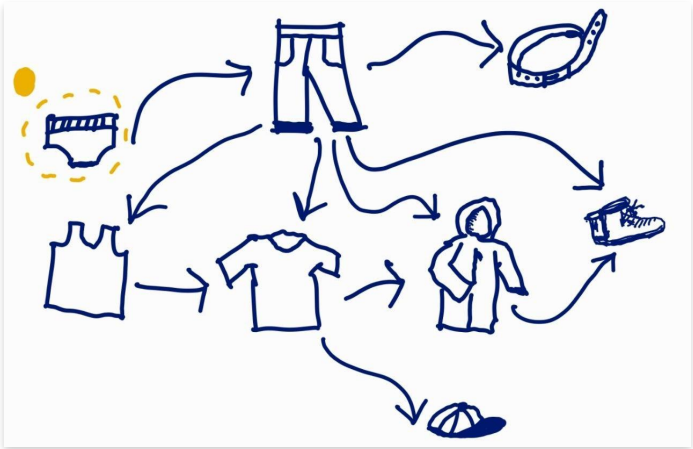
Boxer > Trouser > **Belt** > Undershirt > T-shirt > Cap > Coat > Shoes

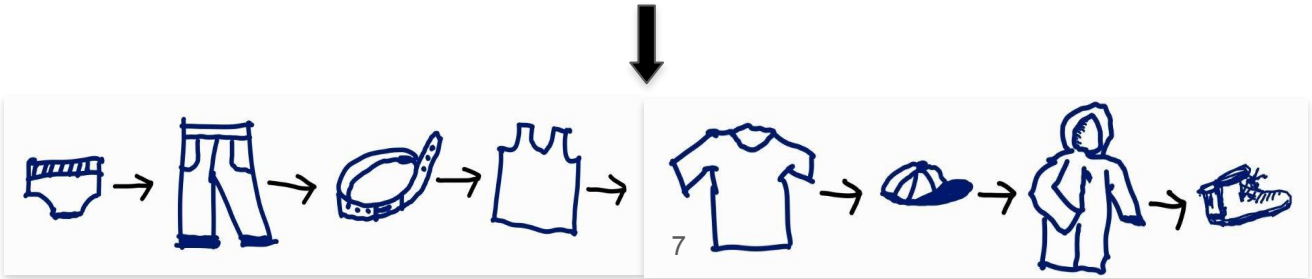
Boxer > Trouser > Undershirt > T-shirt > Cap > **Belt** > Coat > Shoes

Boxer > Trouser > Undershirt > T-shirt > Cap > Coat > Shoes > **Belt**

# Topological Sort

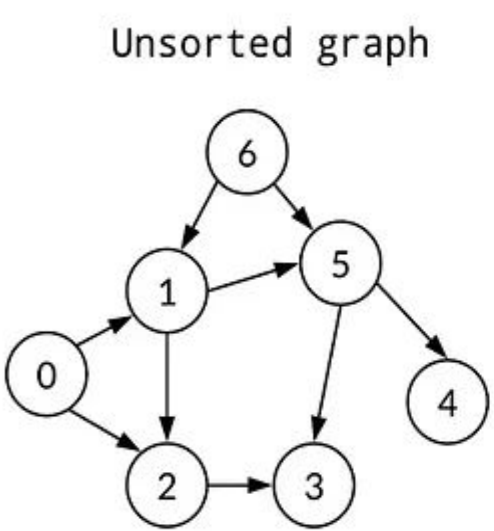
**Topological Sort** is a way to arrange a collection of tasks or events in such a sequence that each task comes before the tasks that depend on it.

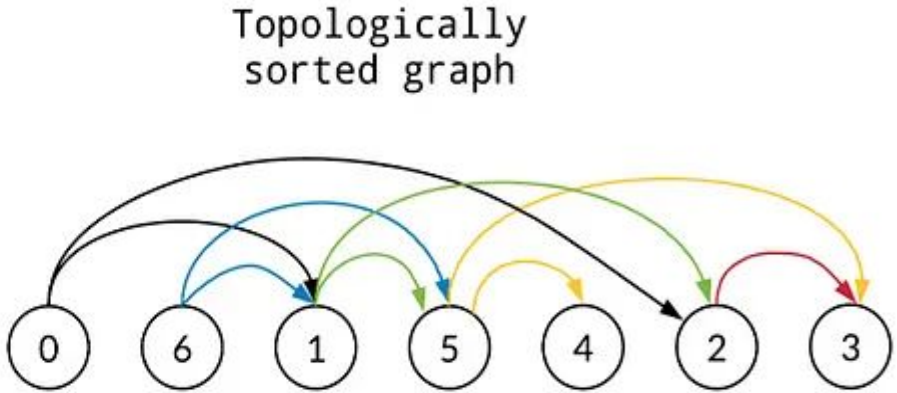




# Topological Sorting

A topological sort is a **linear ordering of nodes** in a **directed acyclic graph (DAG)**



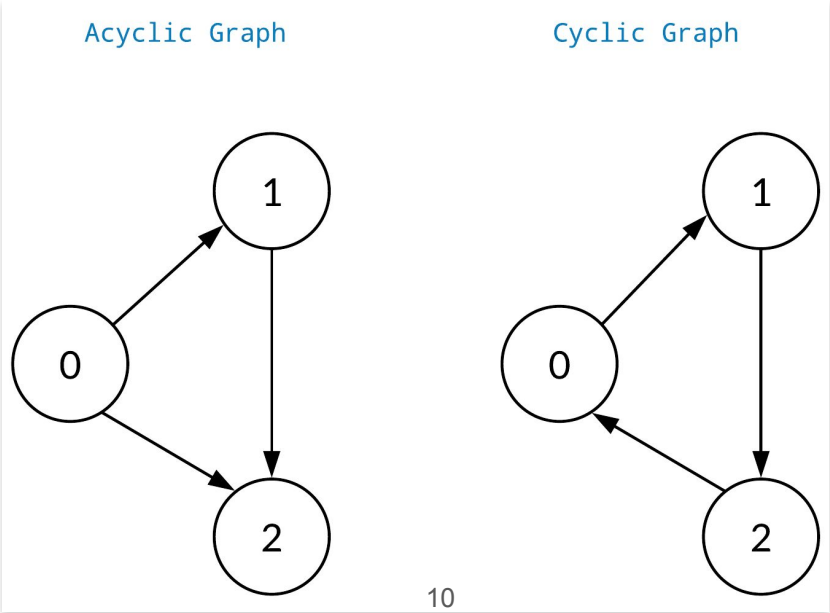
# What is a Directed **Acyclic** Graph (DAG) ?

# Directed Acyclic Graph (DAG)

A **directed** graph with **no cycles**

Topological Sorting for a graph **is not possible** if the **graph is not a DAG**

**Circular dependencies** cannot be resolved in real life



# Motivation Question

# Course Schedule II

## 210. Course Schedule II

Medium 10533 338 Add to List Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return *the ordering of courses you should take to finish all courses.* If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

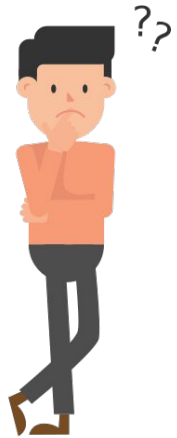
**Example 1:**

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].
```

**Example 2:**

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.
So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].
```

# How is topological sorting done ?



# Different Implementation

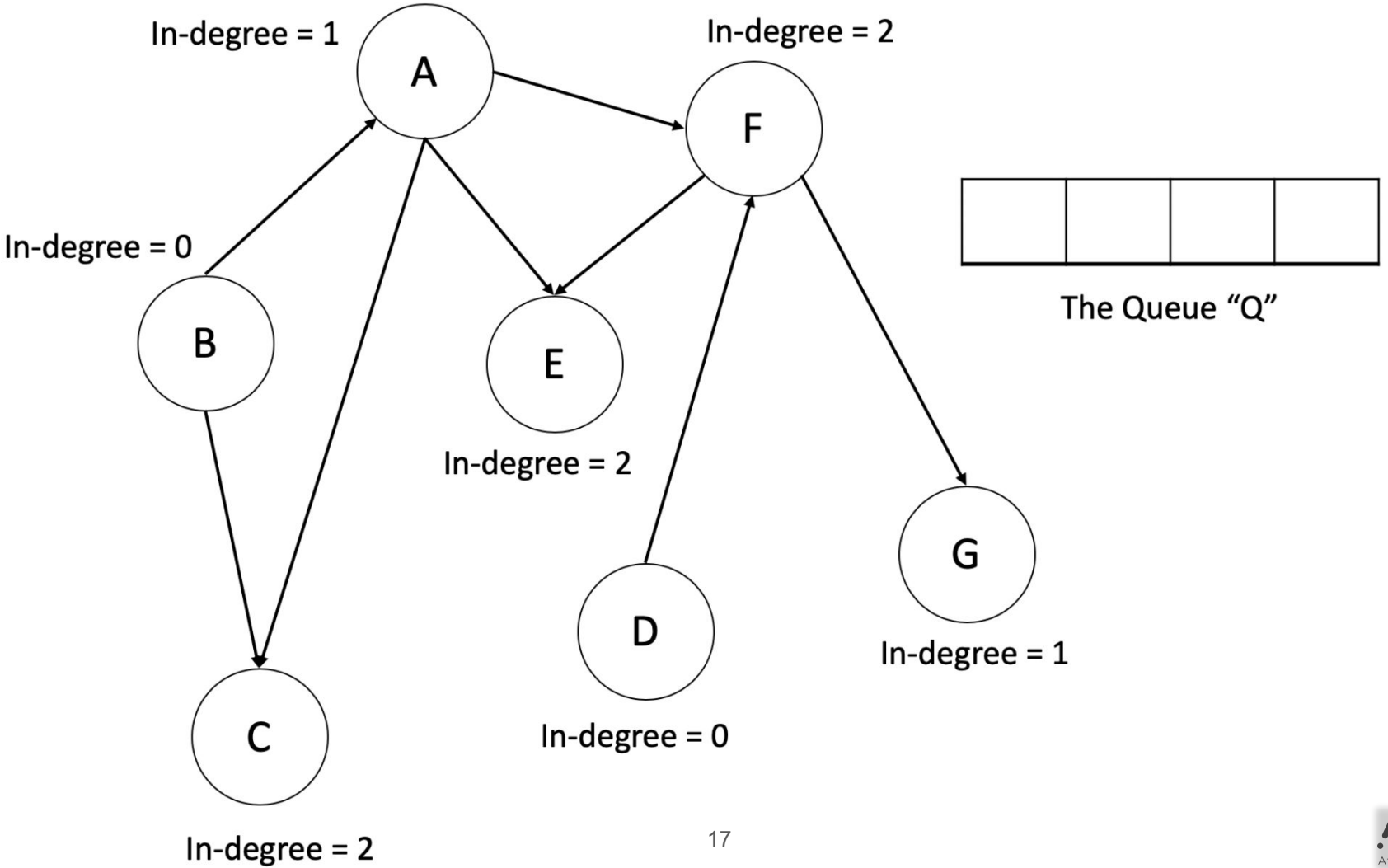
To find the **topologically sorted** order of **DAG**, we have 2 algorithms that can be used

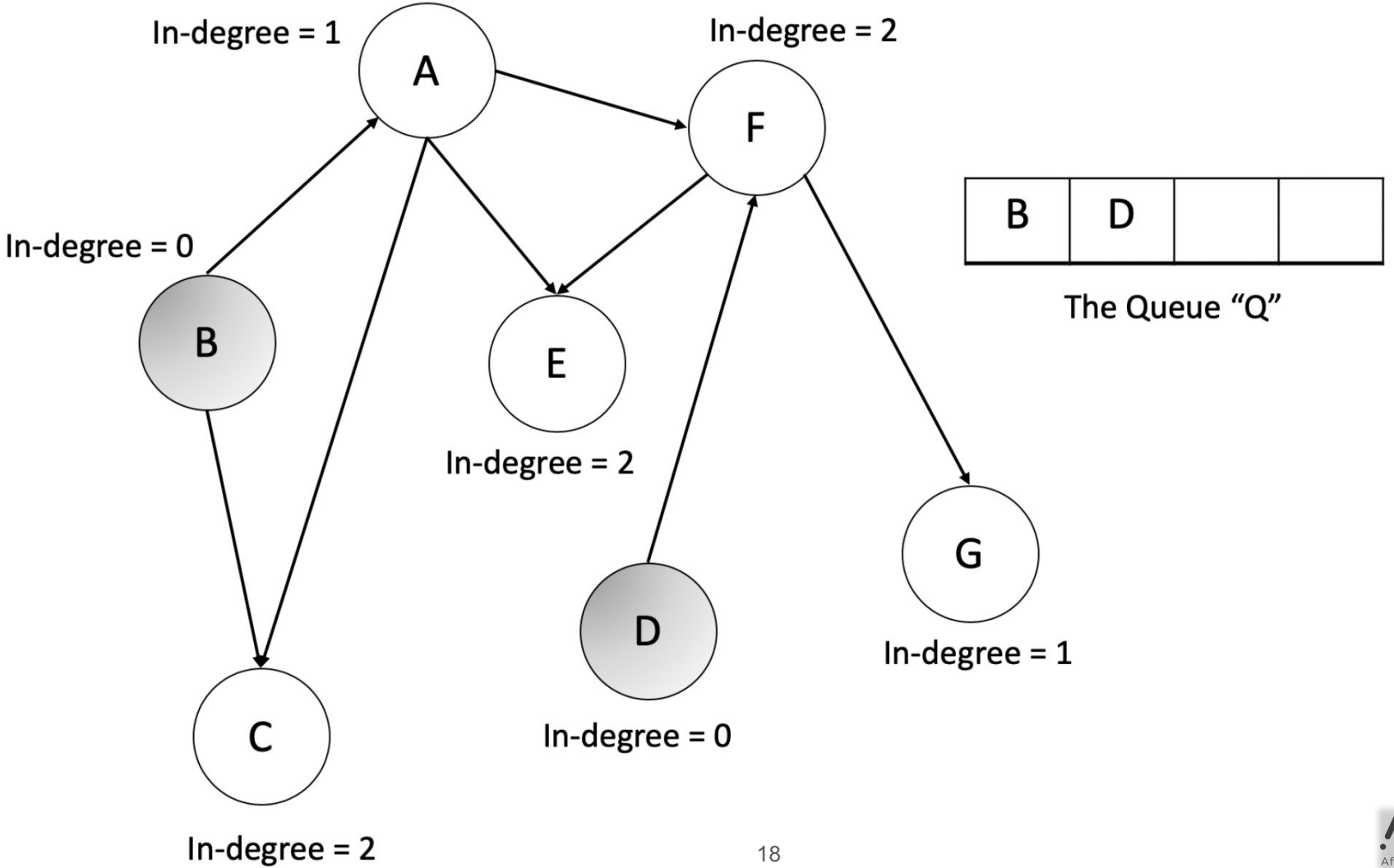
- Kahn's Algorithm
- Modified Depth-first search

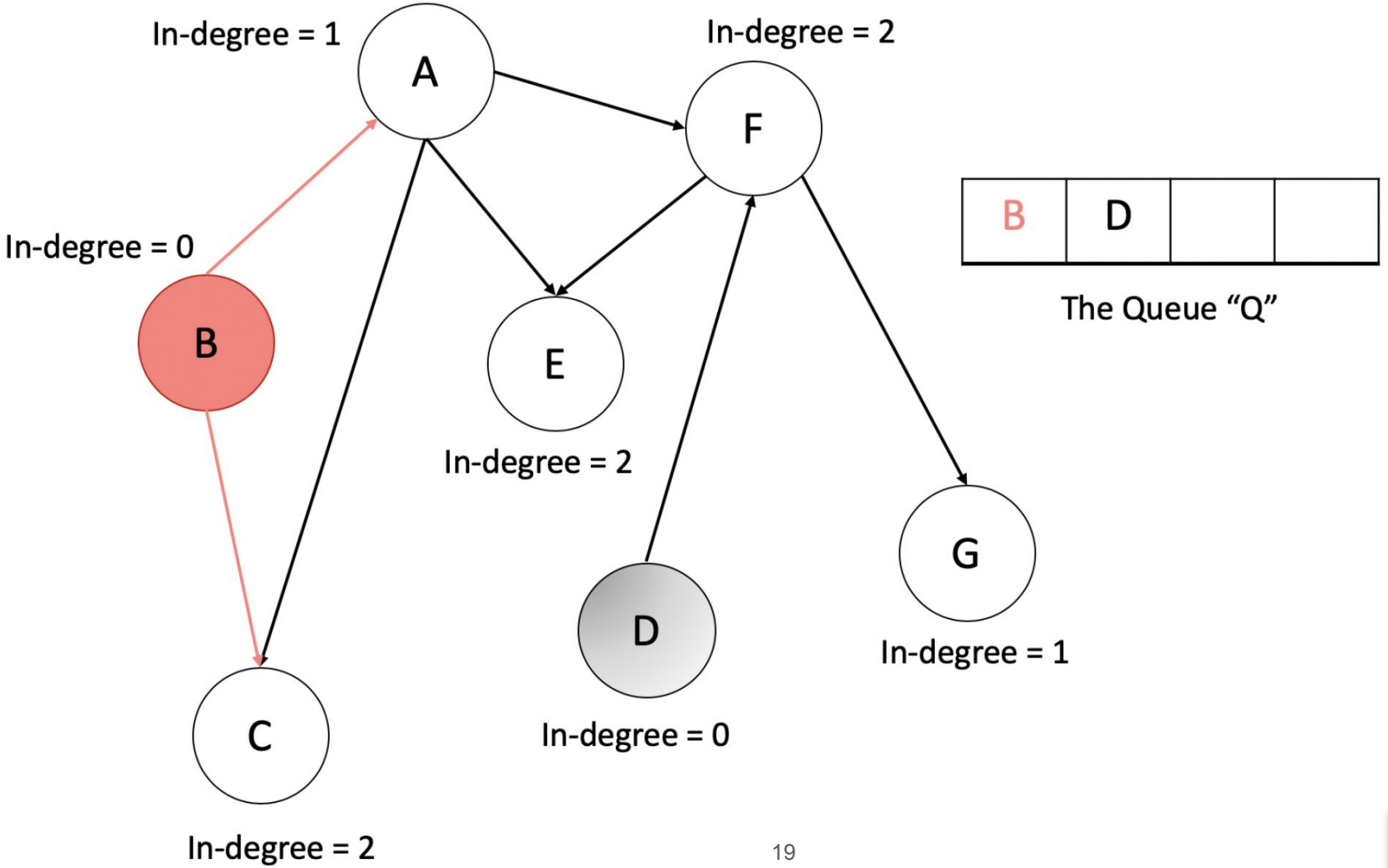
# BFS Implementation - Kahn's Algorithm

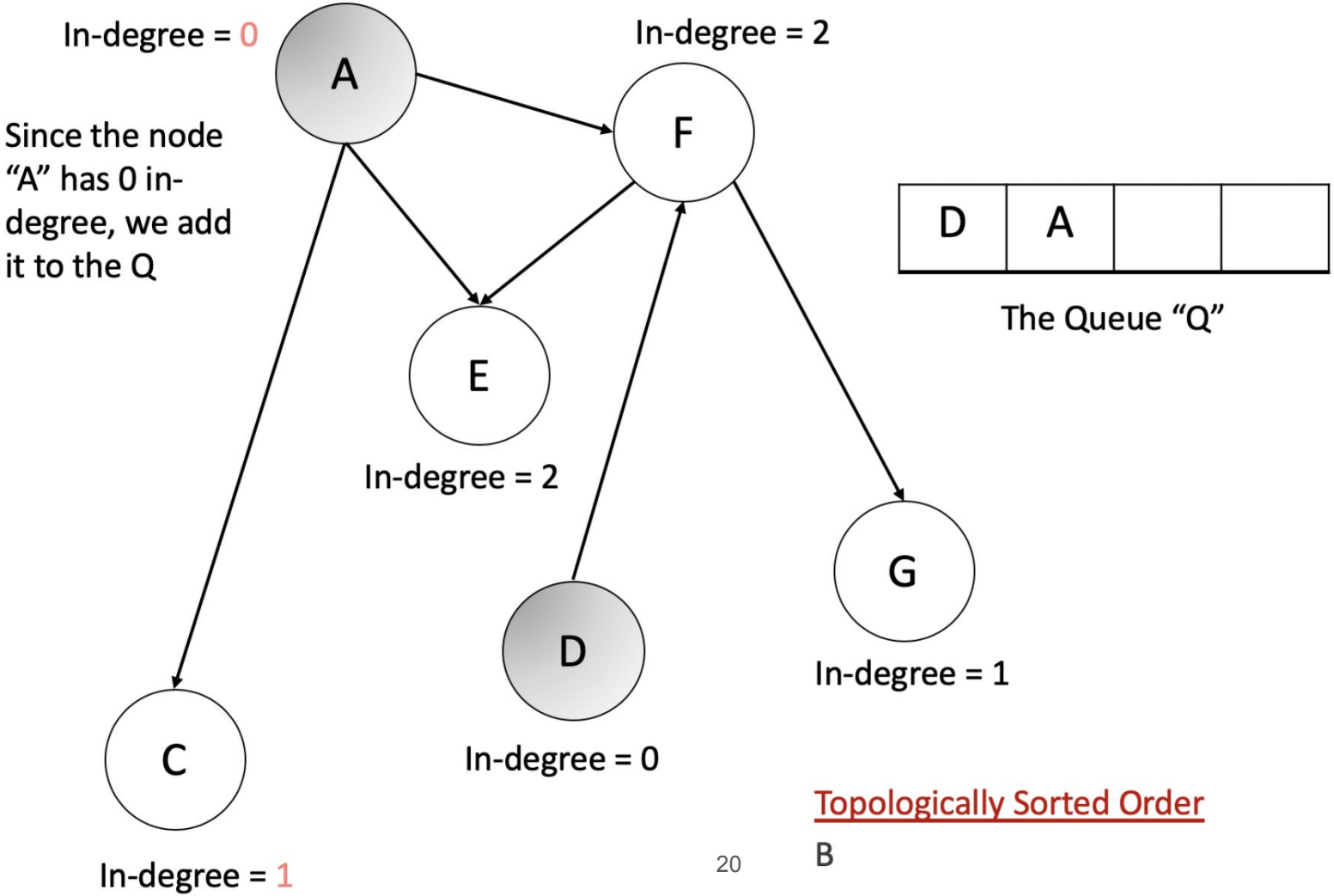
# Kahn's Algorithm

- It starts by finding all the nodes in the graph that have no incoming edges - these are the **source nodes**.
- The algorithm adds these source nodes to the queue, since they **don't depend on any other nodes**.
- The algorithm then **removes the source nodes** from the graph, along with their outgoing edges.
- It then **finds new source nodes** in the reduced graph and the process is repeated.



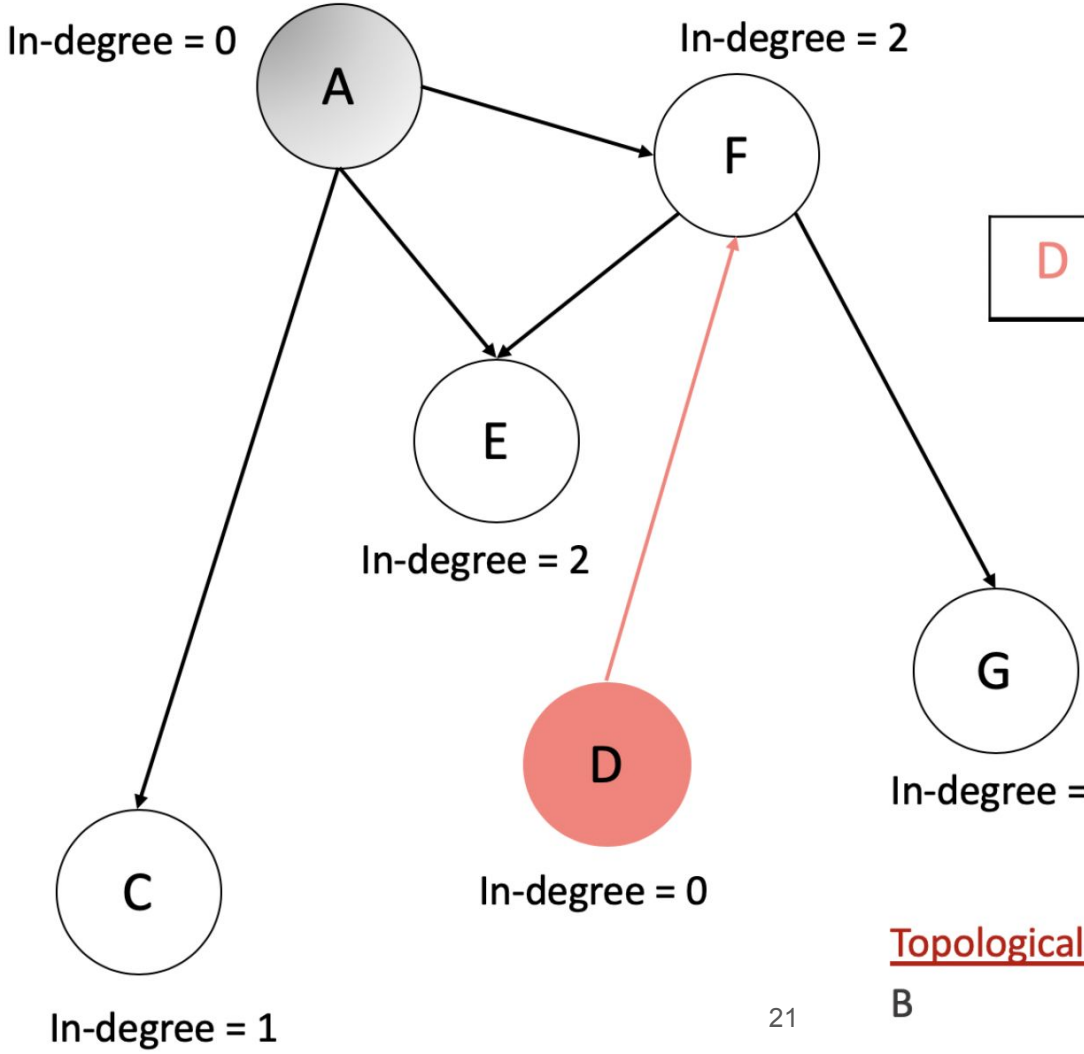







Topologically Sorted Order

B

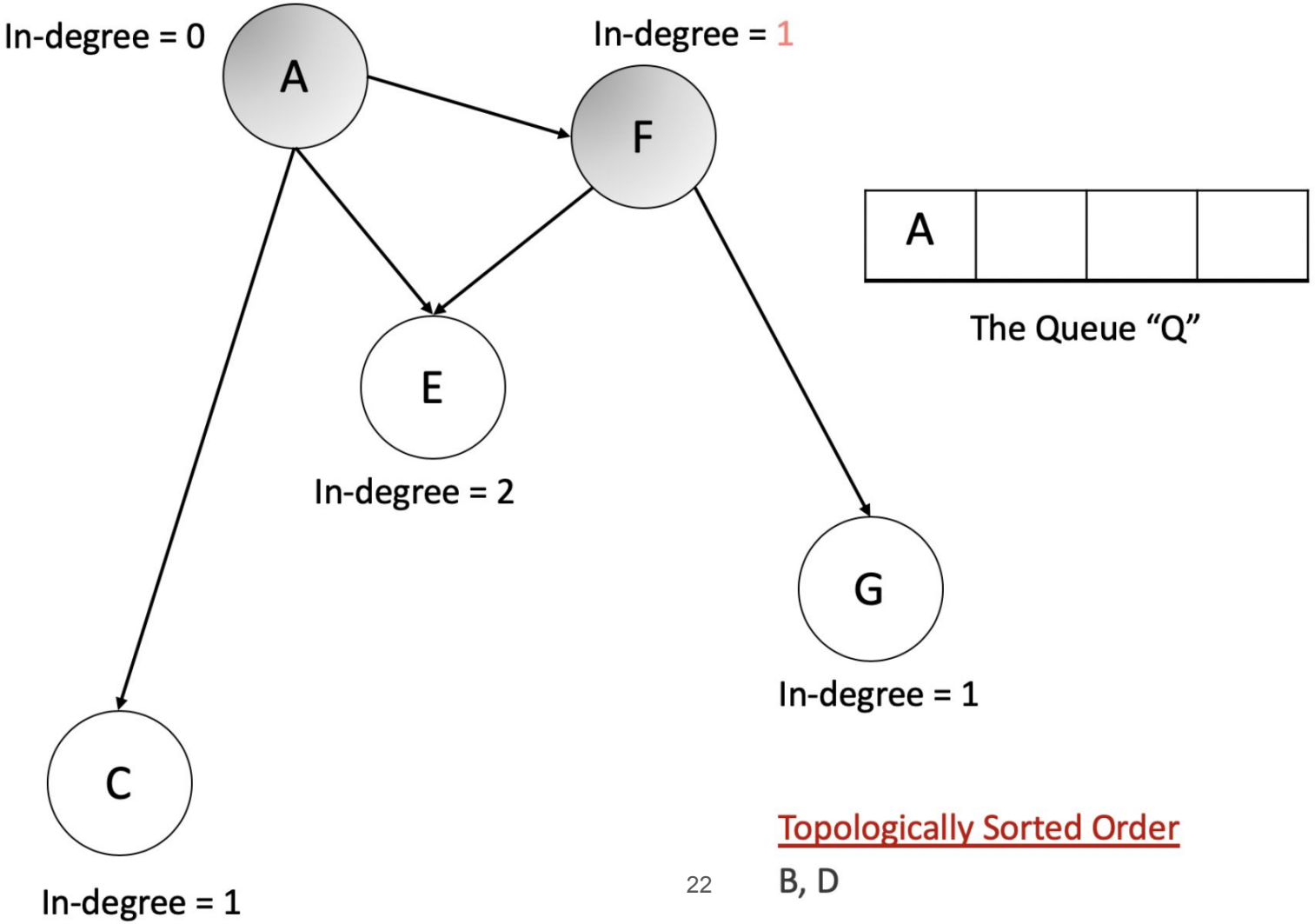




The Queue "Q"

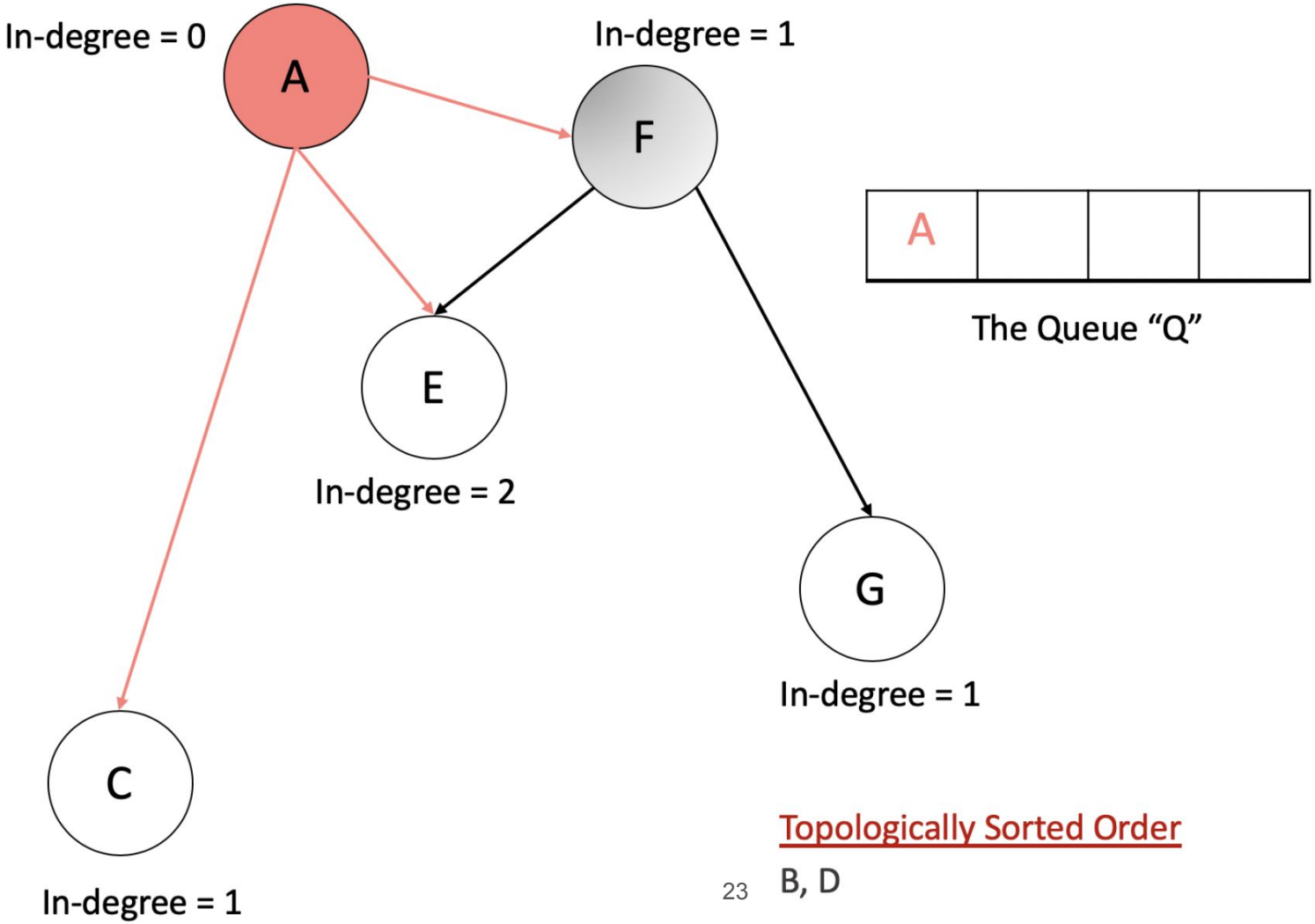
Topologically Sorted Order

B



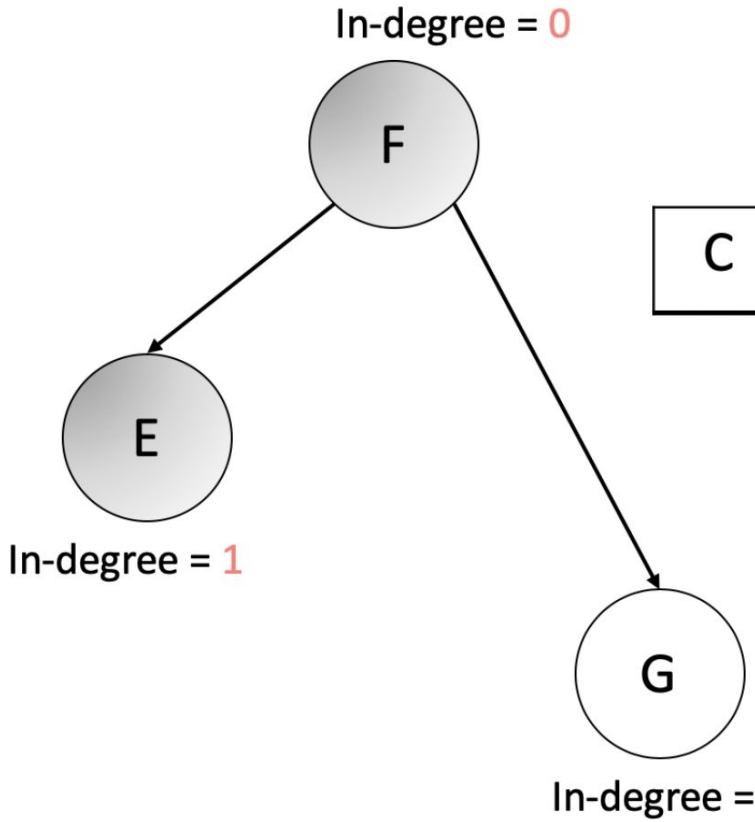

Topologically Sorted Order

B, D



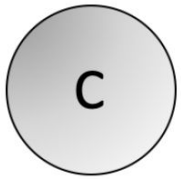

Topologically Sorted Order

B, D



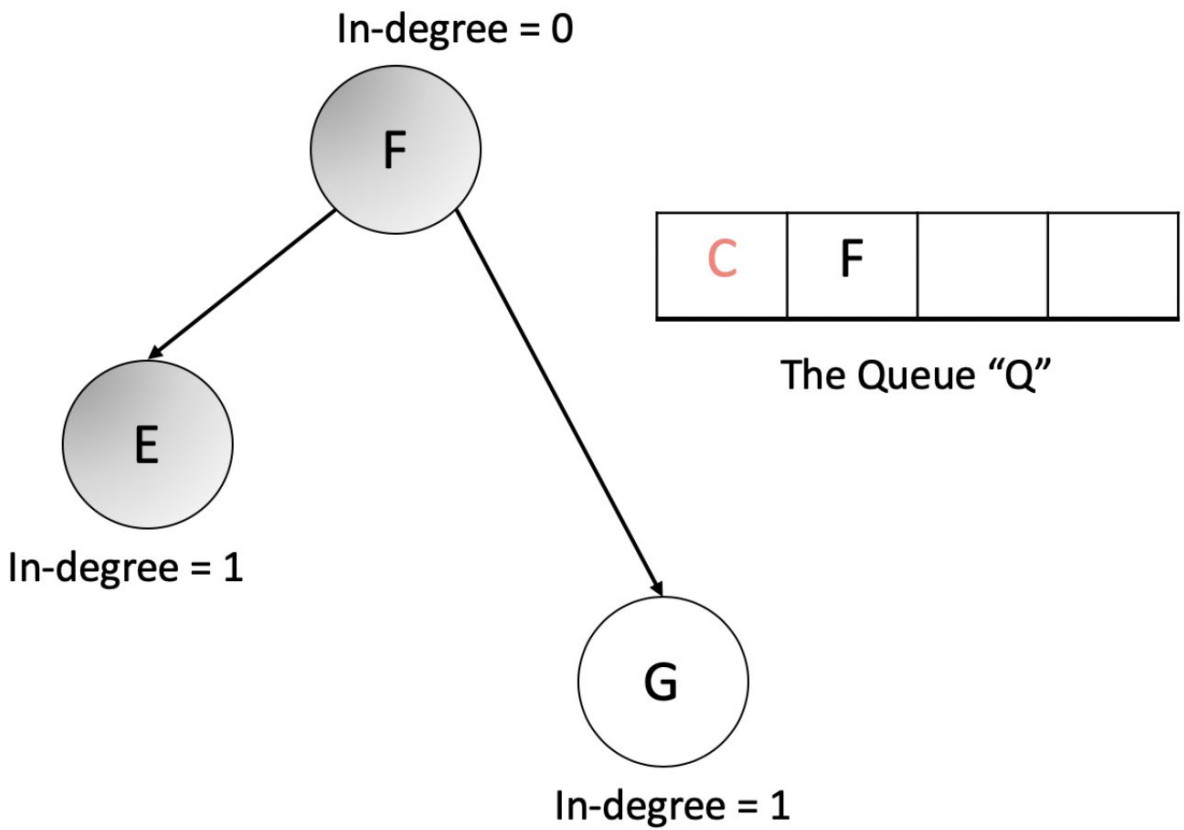

| C | F | | |
|---|---|---|---|

The Queue "Q"




Topologically Sorted Order

B, D, A

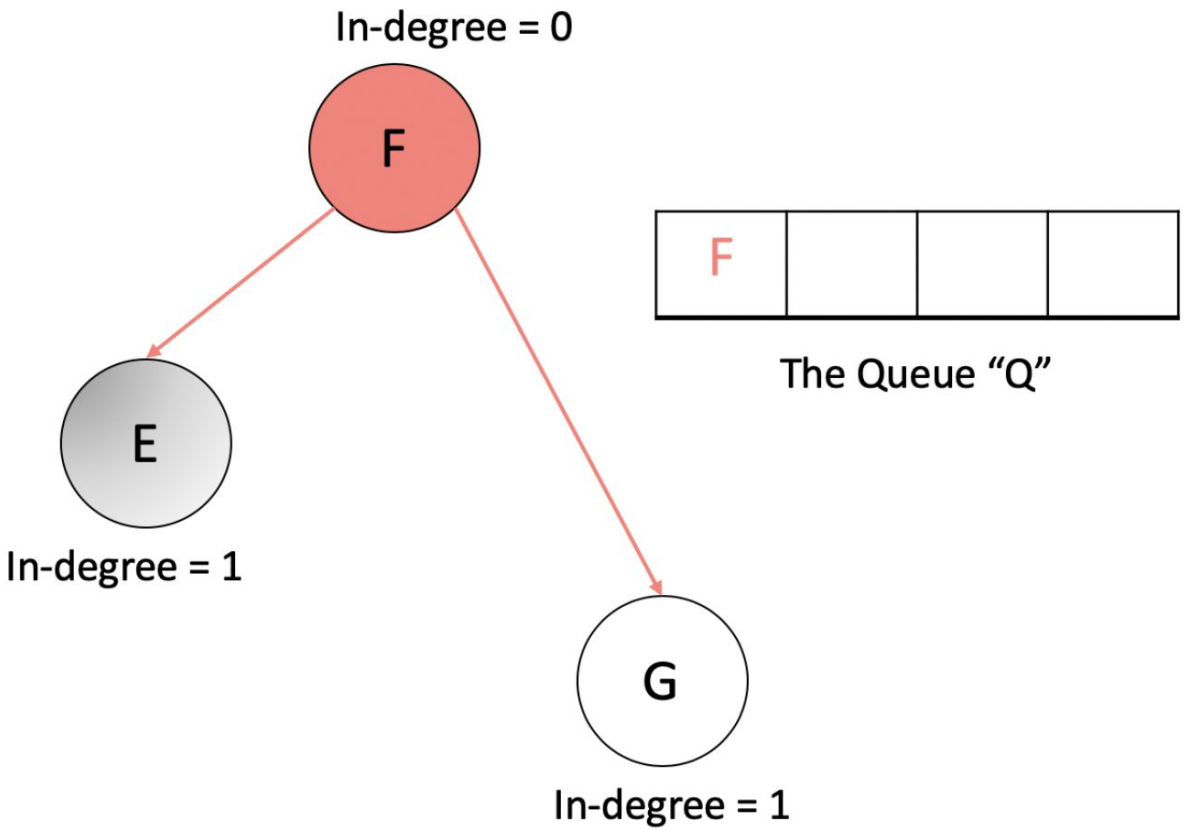



C

In-degree = 0

Topologically Sorted Order

B, D, A




Topologically Sorted Order

B, D, A, C

| E | G | | |
|---|---|---|---|

The Queue “Q”

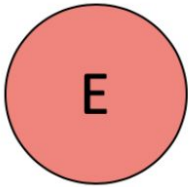

In-degree = 0


In-degree = 0

Topologically Sorted Order

B, D, A, C, F


The Queue "Q"



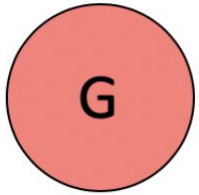

In-degree = 0


In-degree = 0

Topologically Sorted Order

B, D, A, C, F


The Queue “Q”




In-degree = 0

Topologically Sorted Order

B, D, A, C, F, E

The Queue “Q”

Topologically Sorted Order

B, D, A, C, F, E, G

# BFS Implementation

```
class Solution:
    def findOrder(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]
        incoming = [0 for _ in range(numCourses)]
        queue = deque()
        order = []

        for course, pre in prerequisites:
            graph[pre].append(course)
            # Count incoming edges for each node
            incoming[course] += 1

        # Enqueue all nodes with no incoming edges
        for course in range(numCourses):
            if incoming[course] == 0:
                queue.append(course)

        while queue:
            course = queue.popleft()
            order.append(course)

            for neighbor in graph[course]:
                # Current node is removed, so neighbor
                # has one less incoming edge.
                incoming[neighbor] -= 1
                # If neighbor has no remaining incoming
                # edges, it can now be processed.
                if incoming[neighbor] == 0:
                    queue.append(neighbor)

        # Why?
        if len(order) != num Courses:
            return []

        return order
```

## Time Complexity

- Let **V** be **number of Nodes (Courses)** and **E** be **number of edges** .
- We are traversing through the entire graph and **we will only visit a node once**. Which would take a time complexity of **O(V + E).**

## Space Complexity

- Let **V** be **number of Nodes (Courses)** and **E** be **number of edges** .
- We will store the nodes, and also the edges.

  Space Complexity = **O (V + E)**
- What would the space complexity be if we ignore the complexity of building the graph?

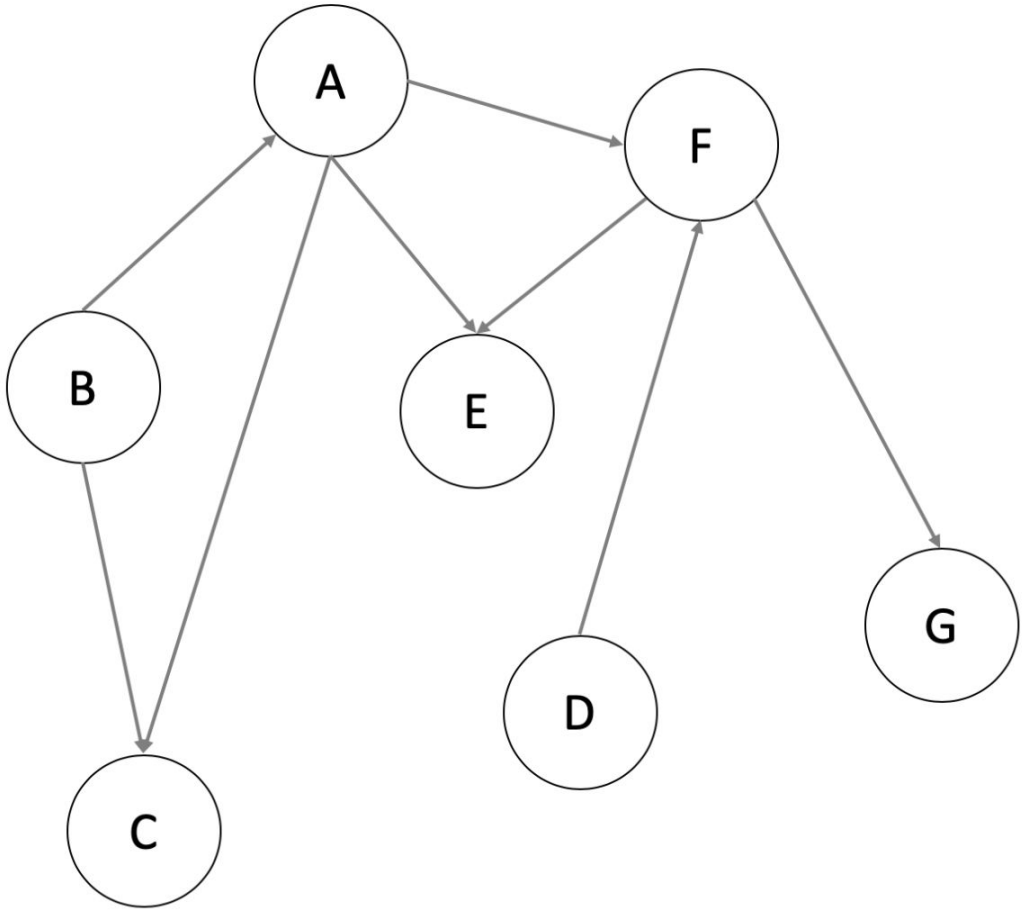
# DFS Implementation

# DFS Algorithm

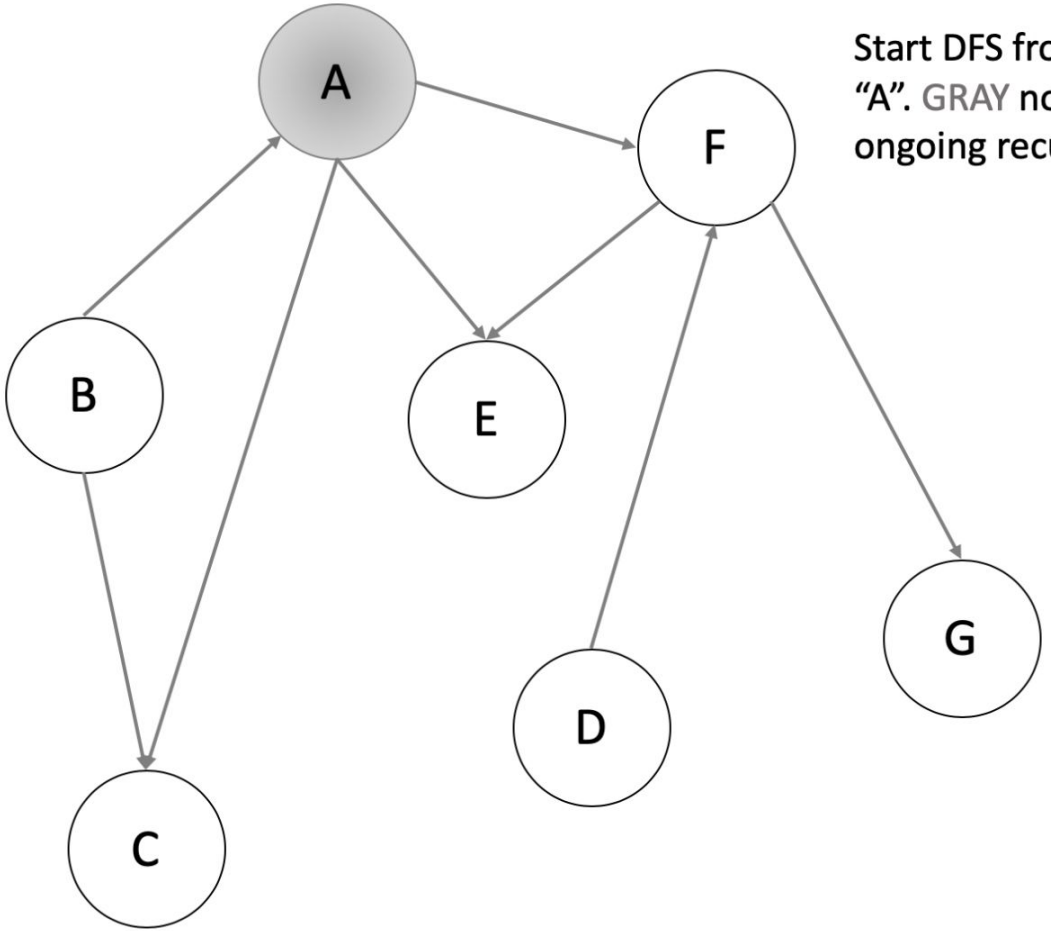
- It creates the topological ordering in **reverse order**.
- It starts from **any node**, and traverses depth-first until it reaches a node with no outgoing edges.
- As it backtracks, it puts the nodes in to a **stack**. This stack represents the reverse topological ordering.
- Maintains three colors to keep track of node state:
    - **White** - Unvisited node
    - **Grey** - Node visited in current path
    - **Black** - Node visited in a different path

# DFS Algorithm

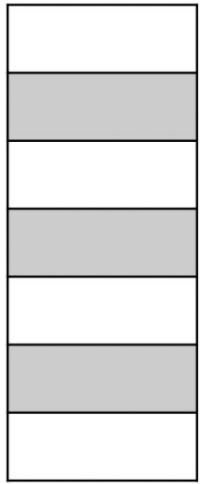
- During traversal:
  - Only traverse to **white** nodes.
  - If a **black** node is found, skip it - it has been processed.
  - If a **grey** node is found, this means there is a cycle. **Why?**



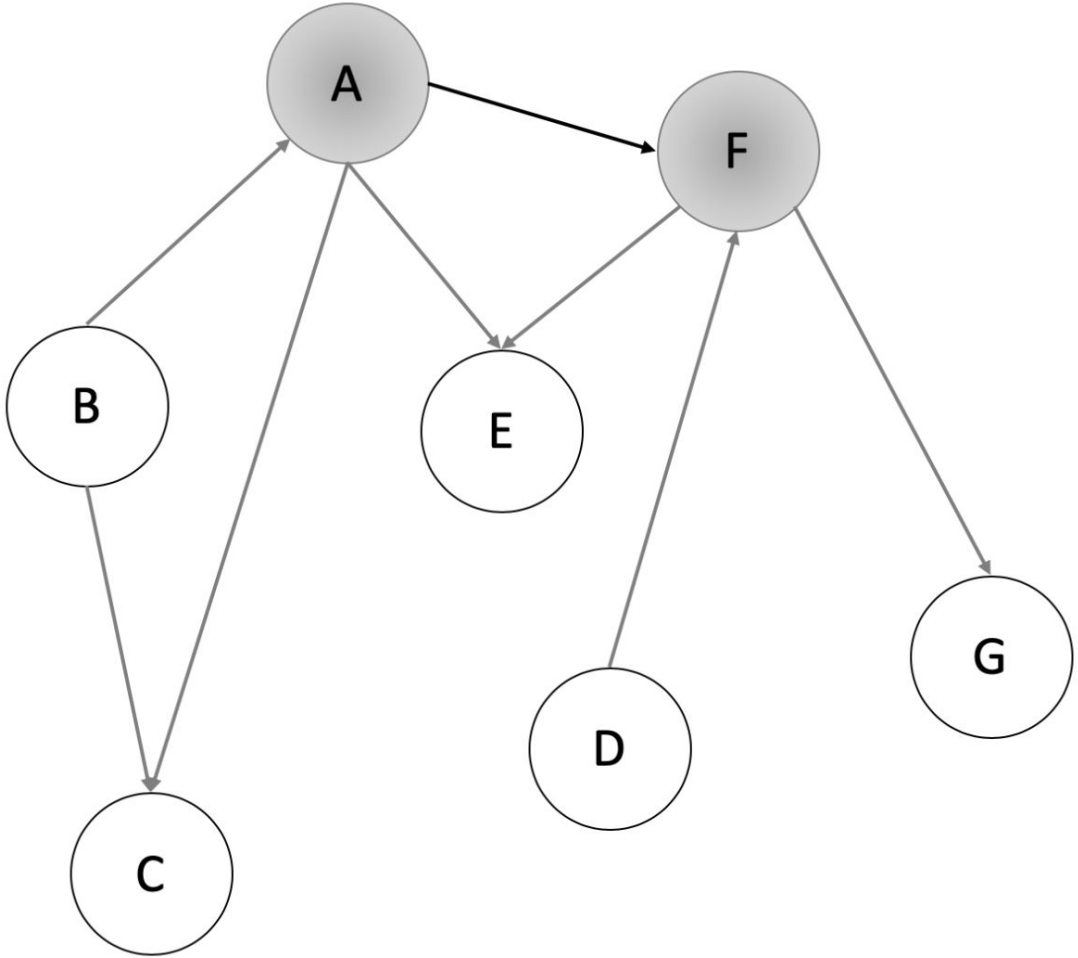

Stack S



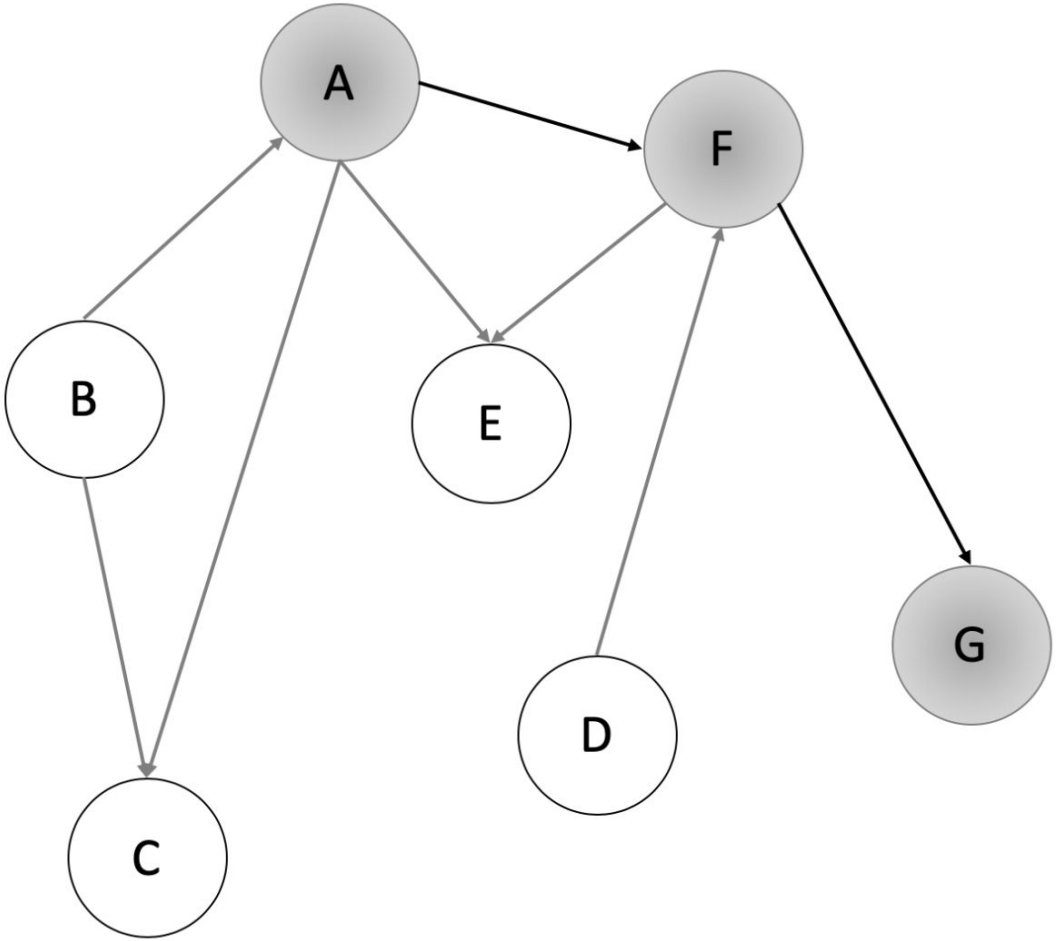

Start DFS from the node "A". GRAY nodes depict ongoing recursion.



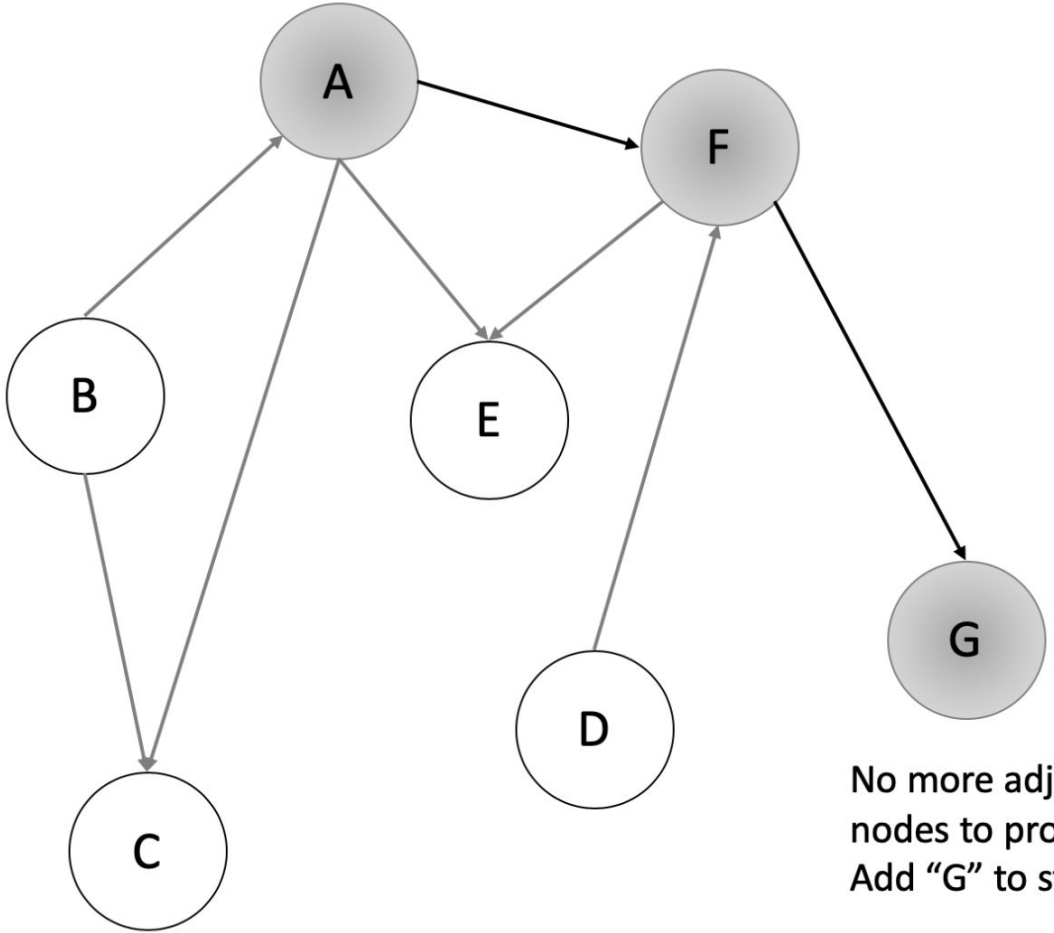
Stack S




Stack S

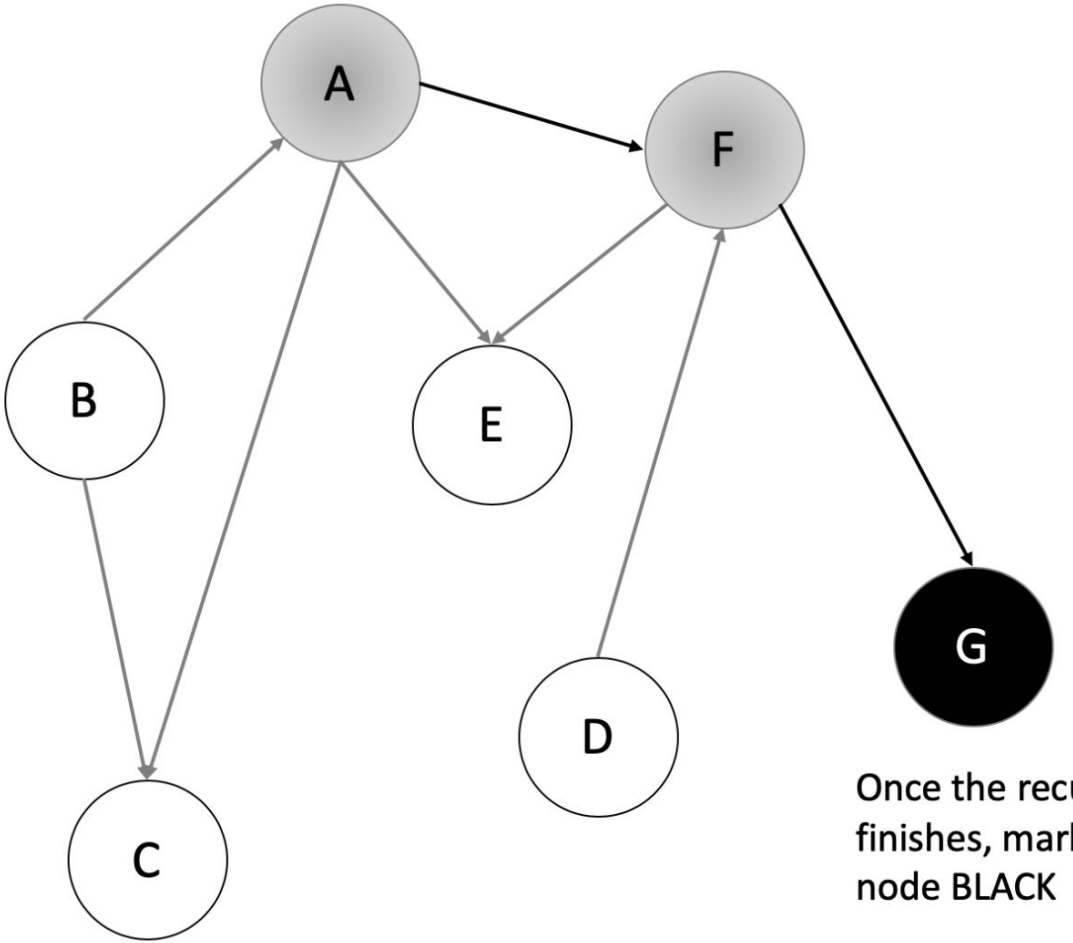



Stack S



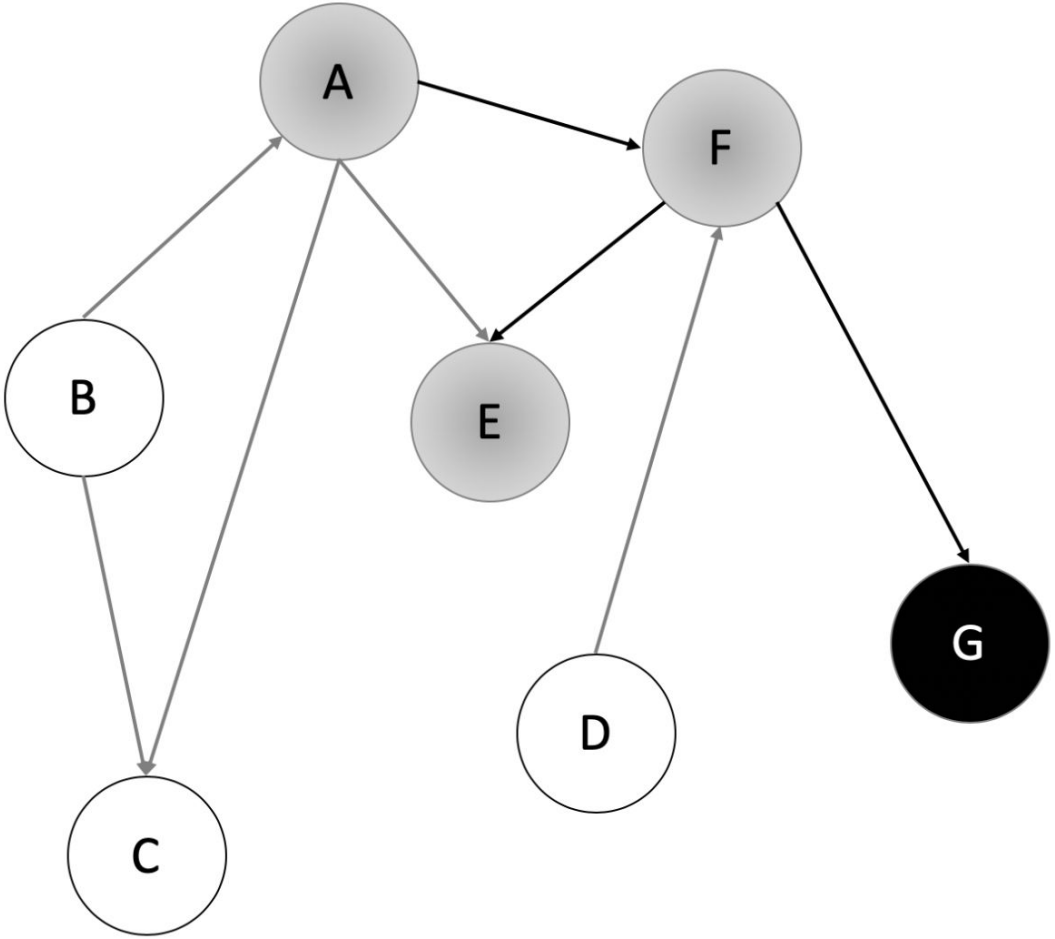

No more adjacent nodes to process. Add “G” to stack.


Stack S



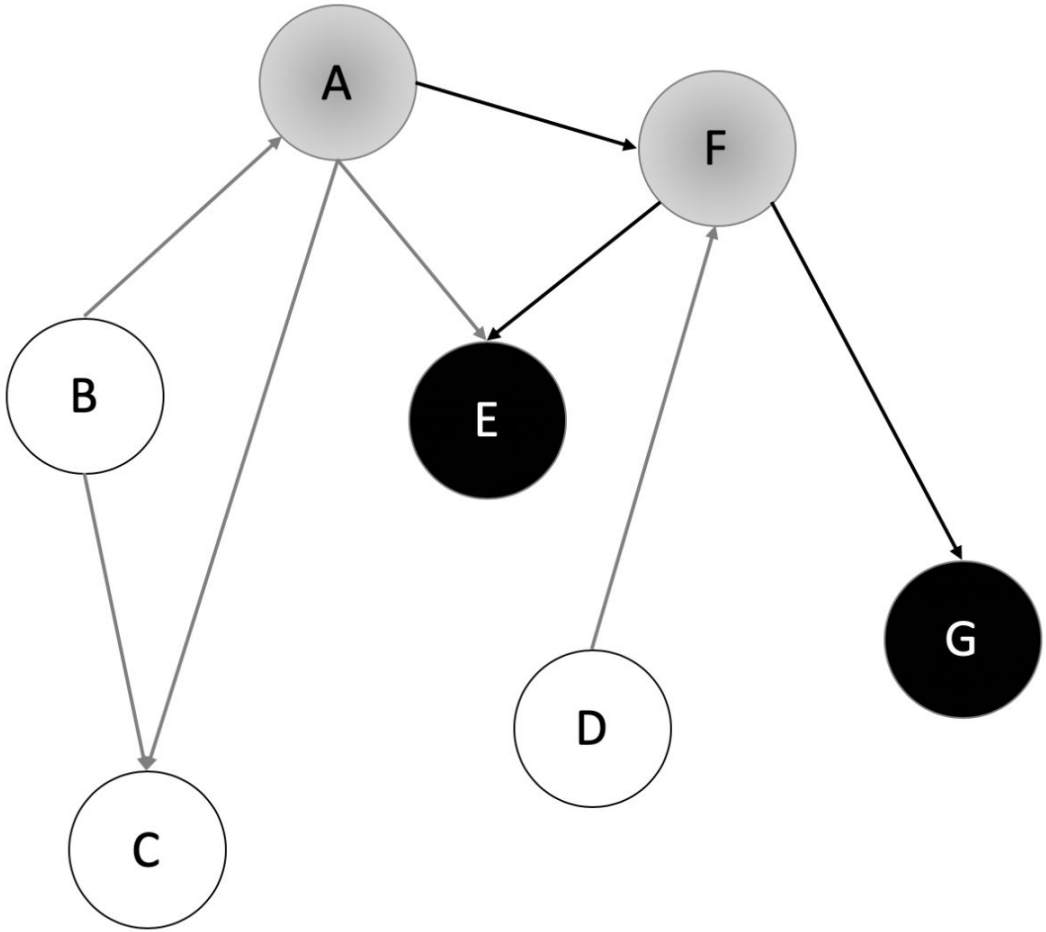

Once the recursion finishes, mark the node BLACK

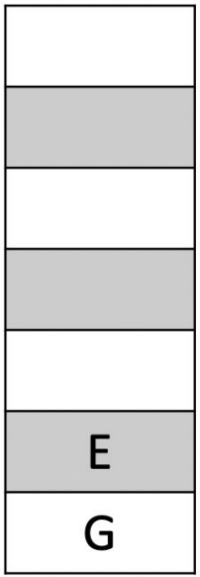

Stack S



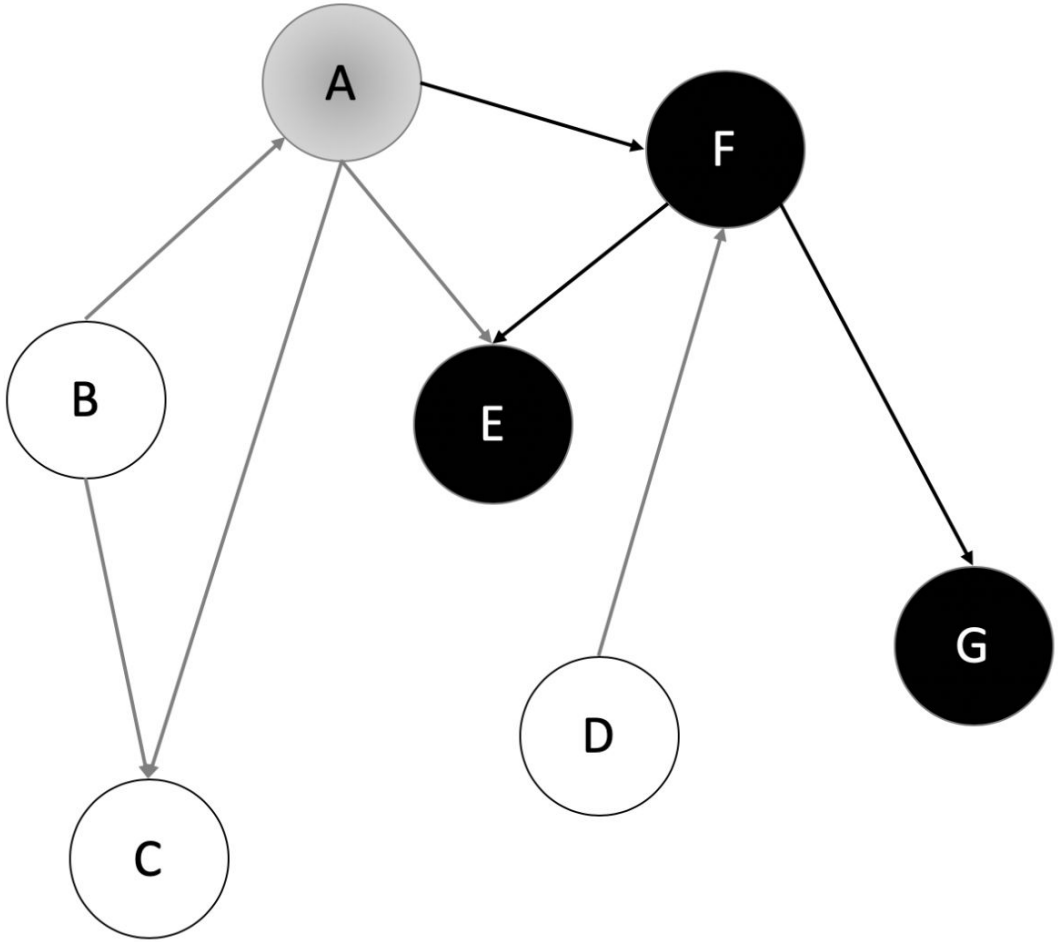


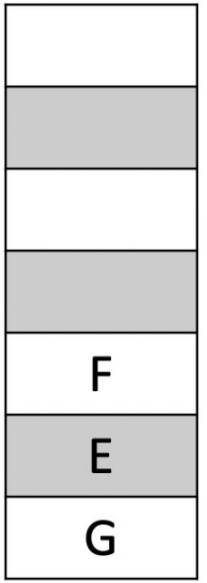
Stack S



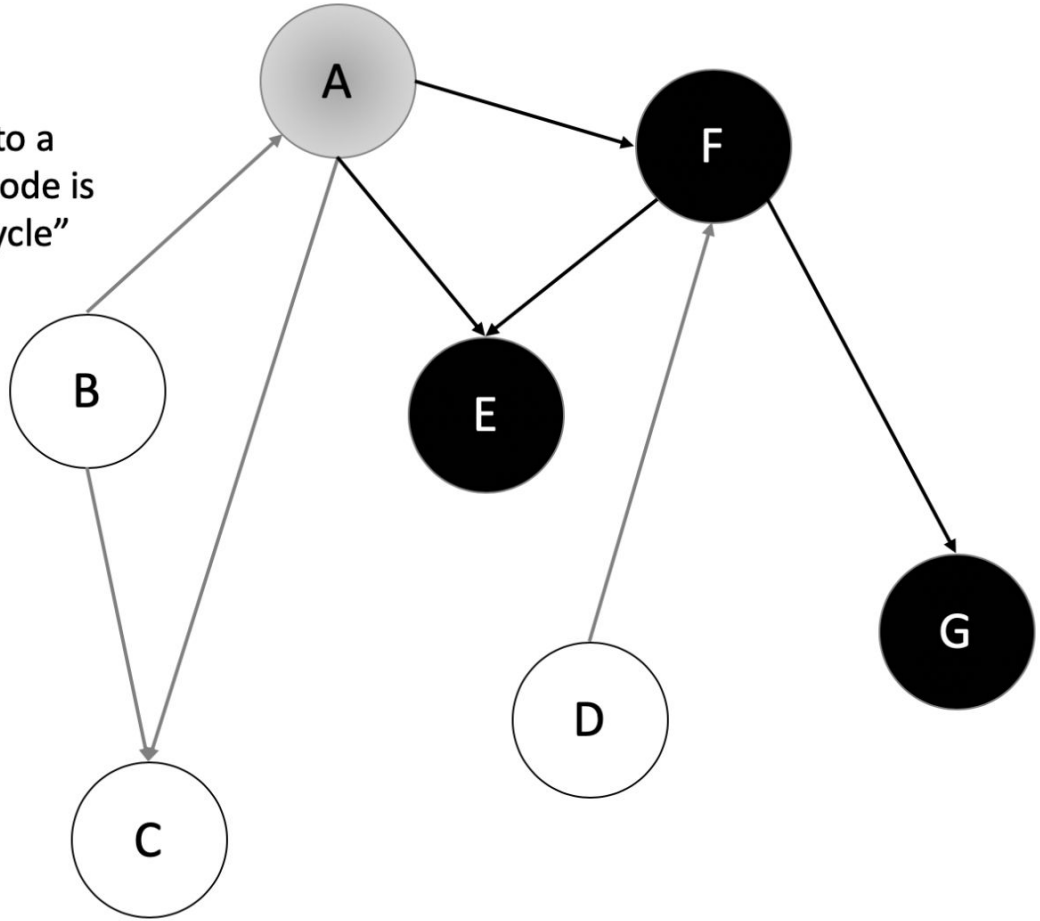

Stack S

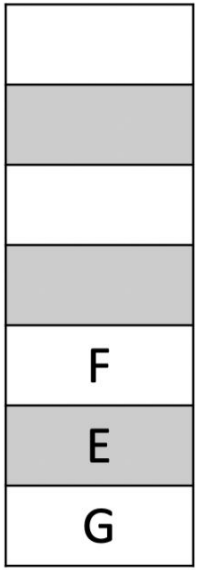


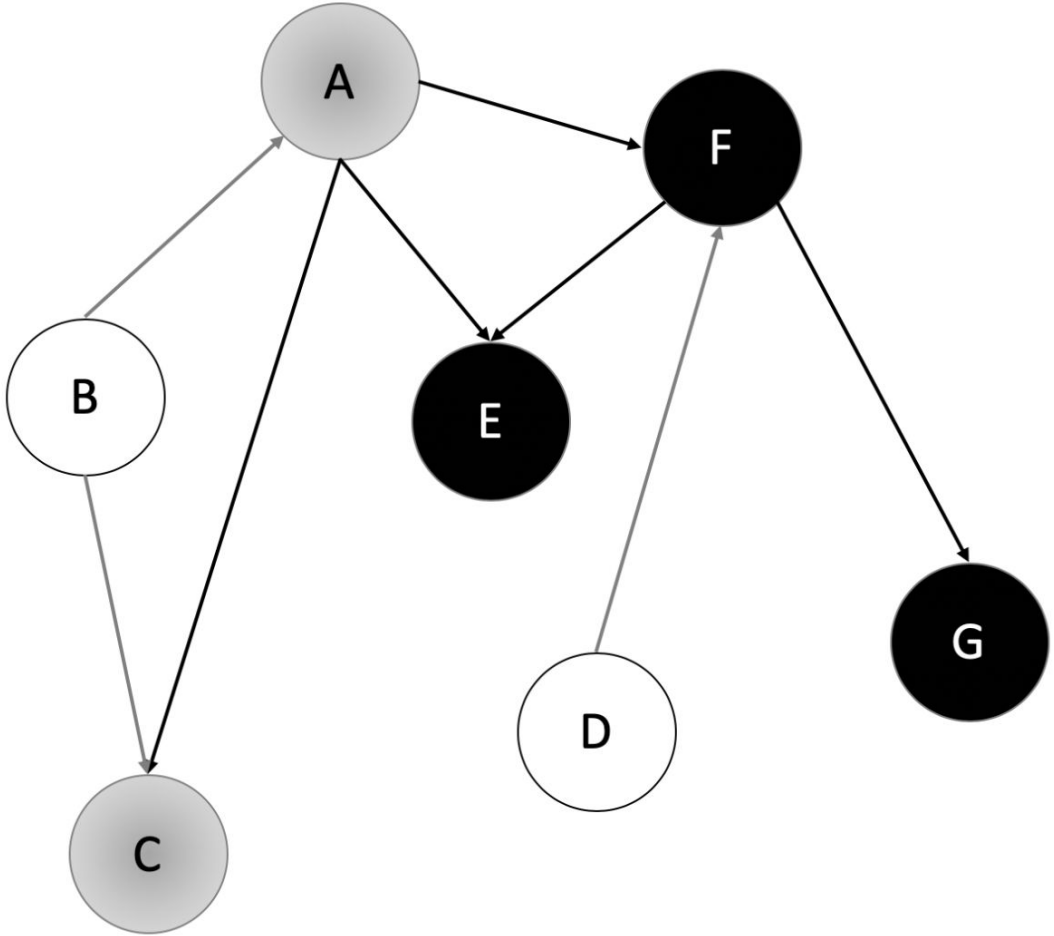

Stack S

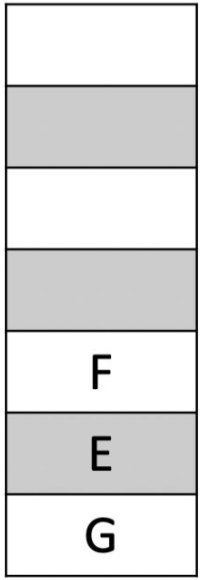
An edge leading to a BLACK node is not a “cycle”



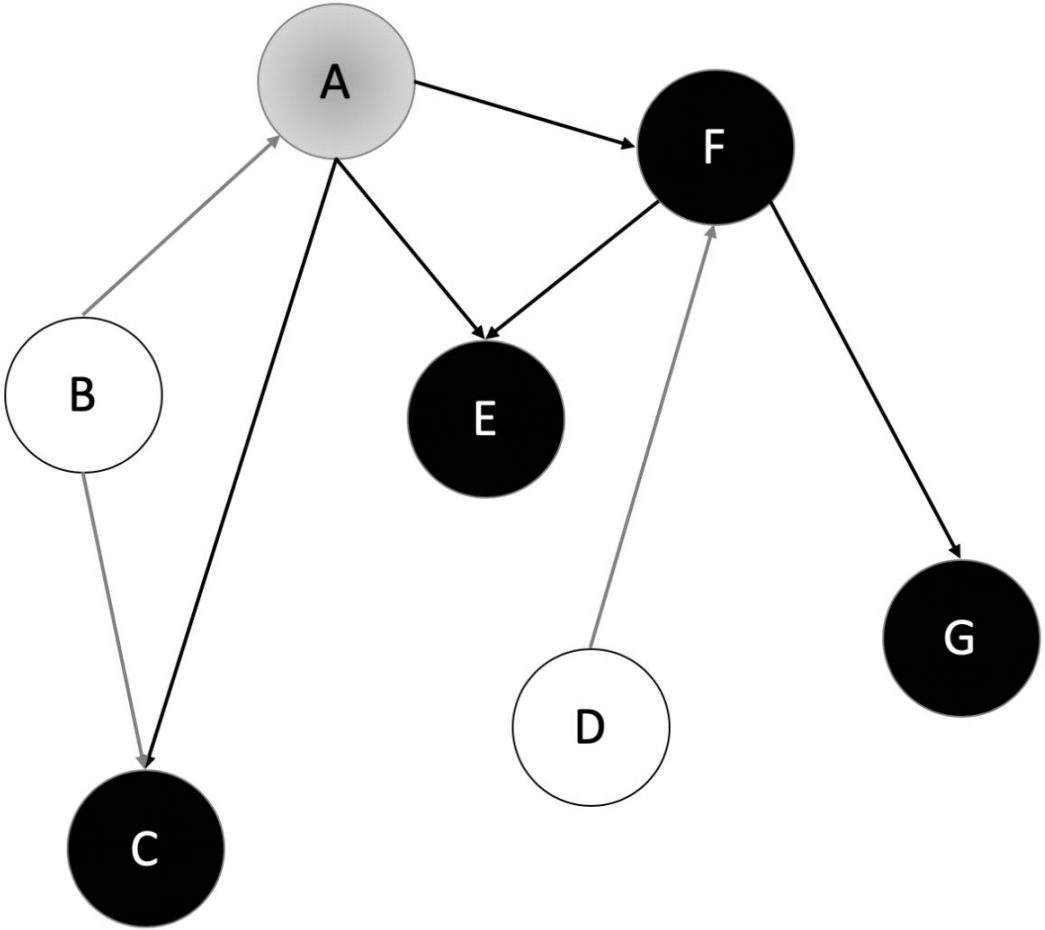



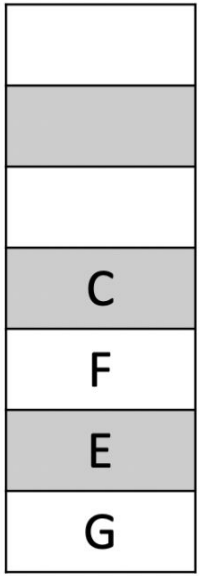

Stack S



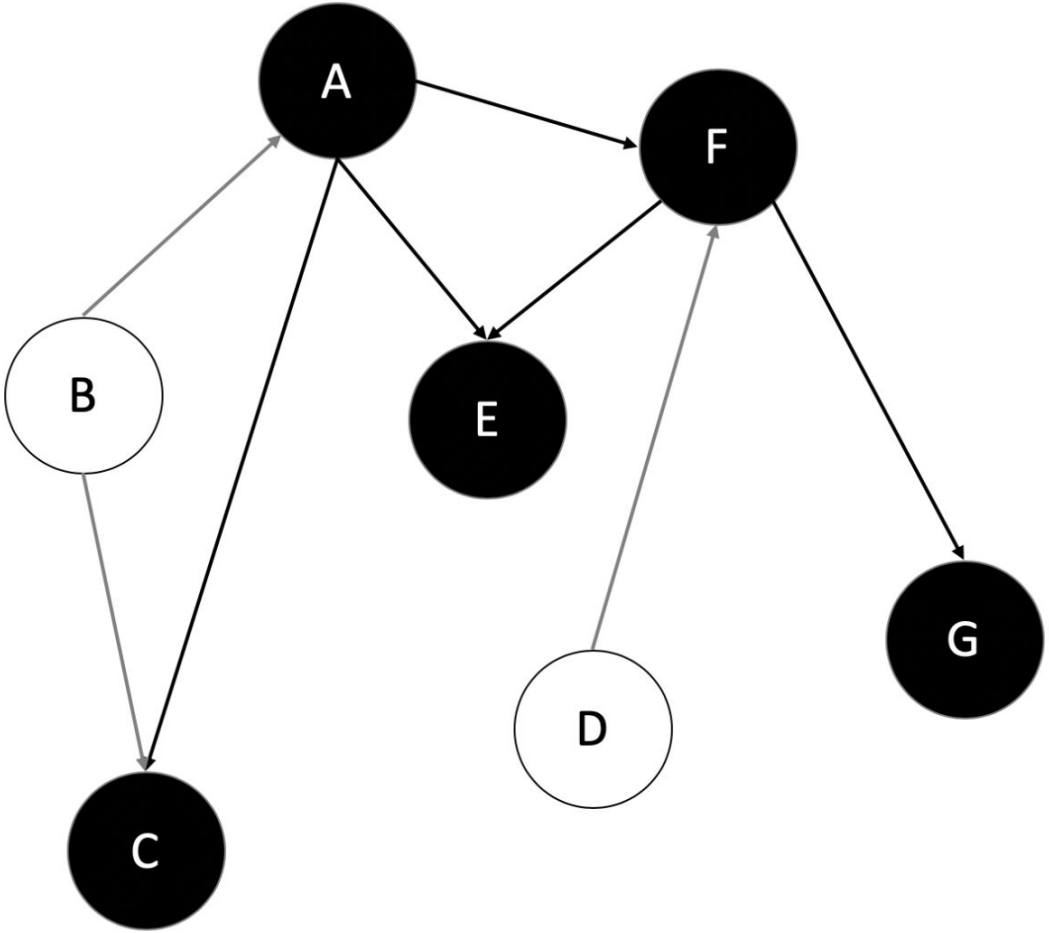



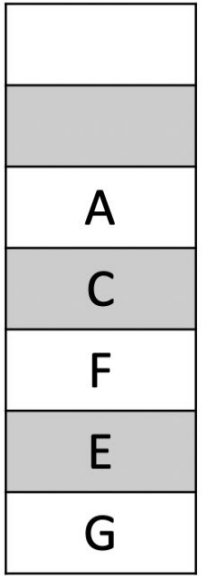

Stack S



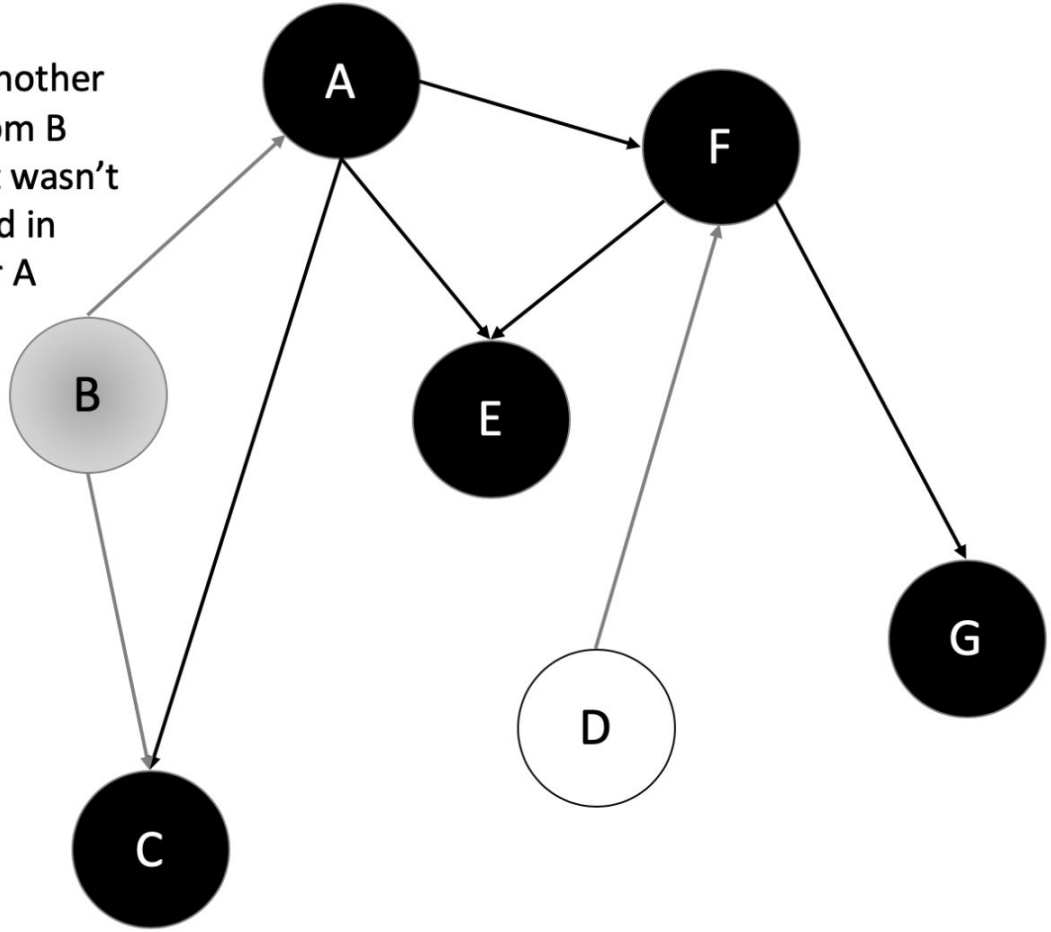

Stack S

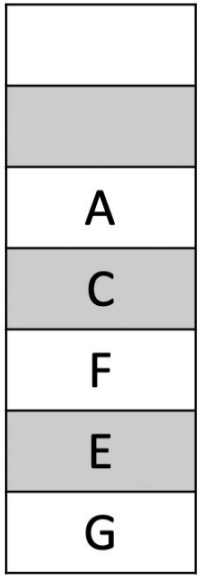


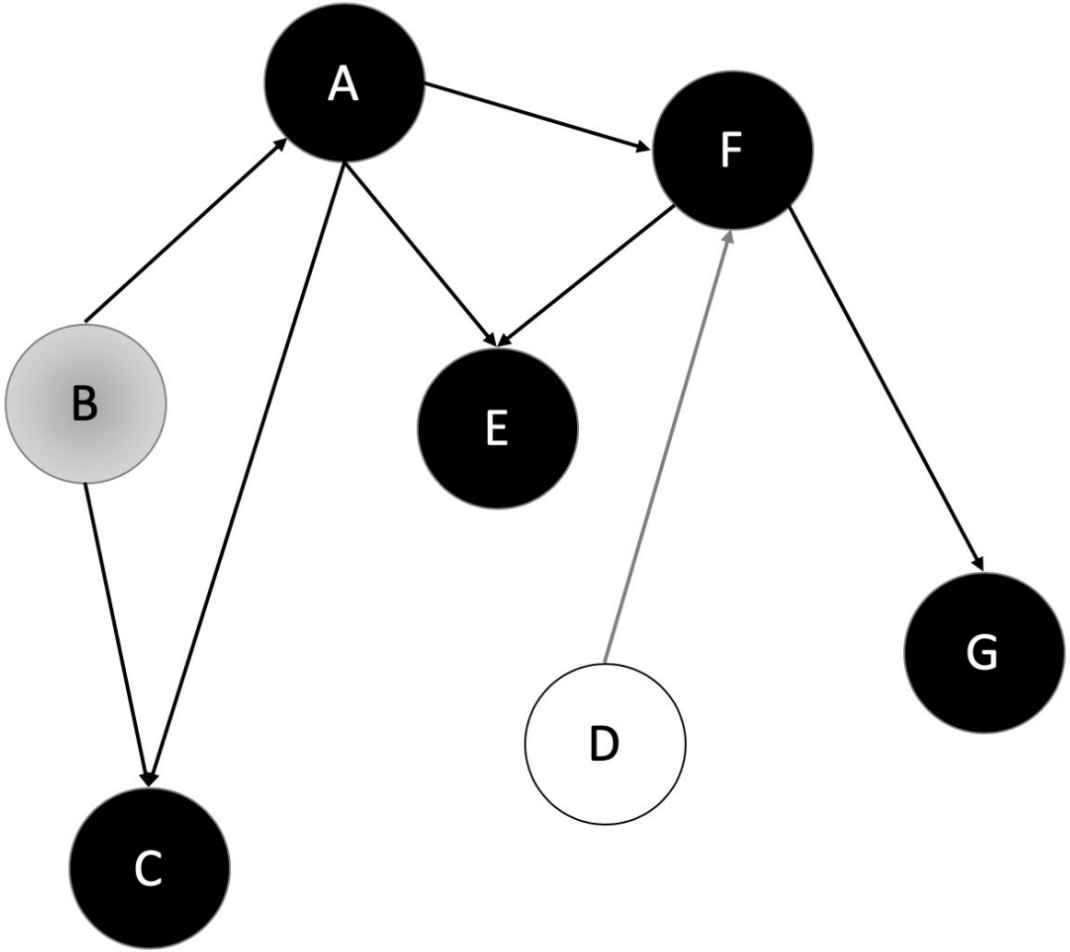

Stack S

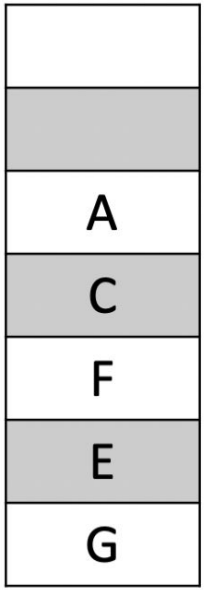
Start another DFS from B since it wasn't covered in DFS for A



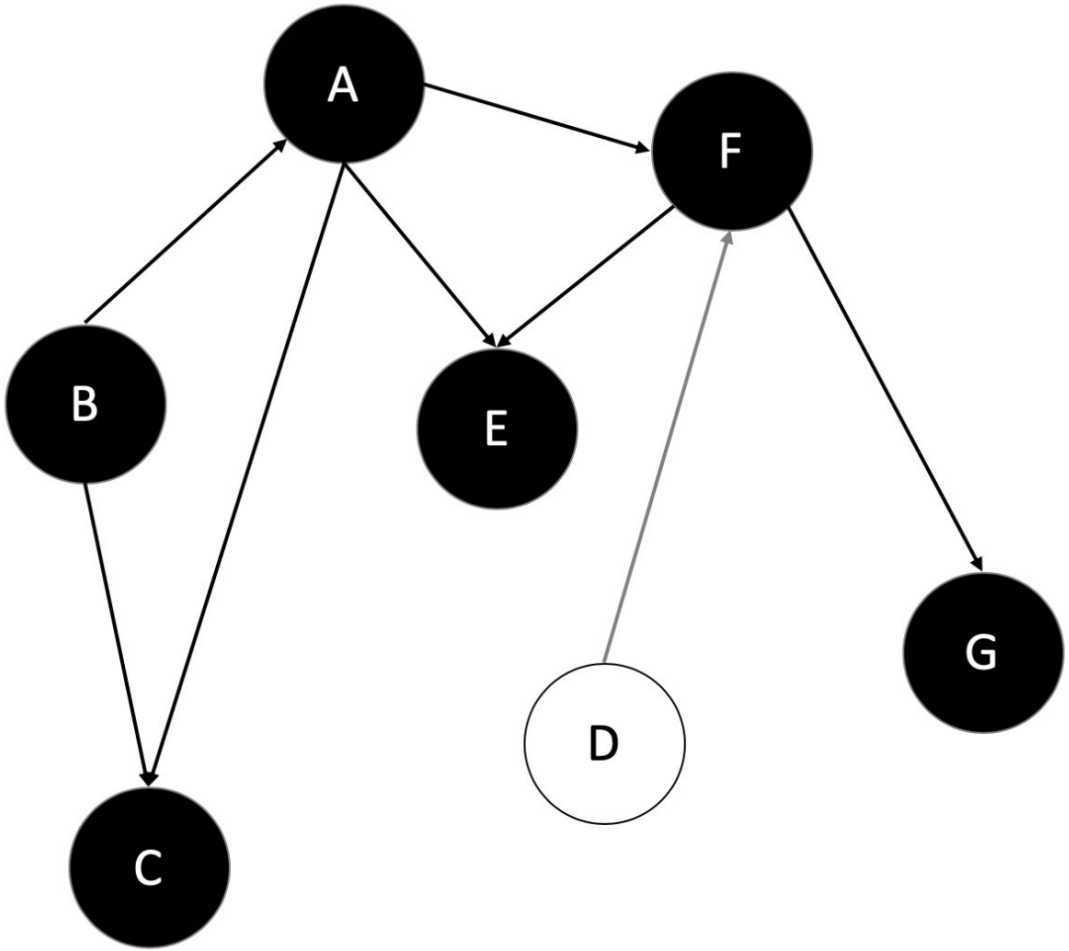



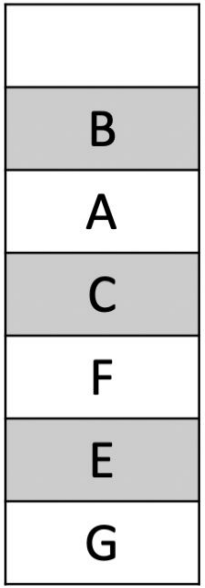

Stack S



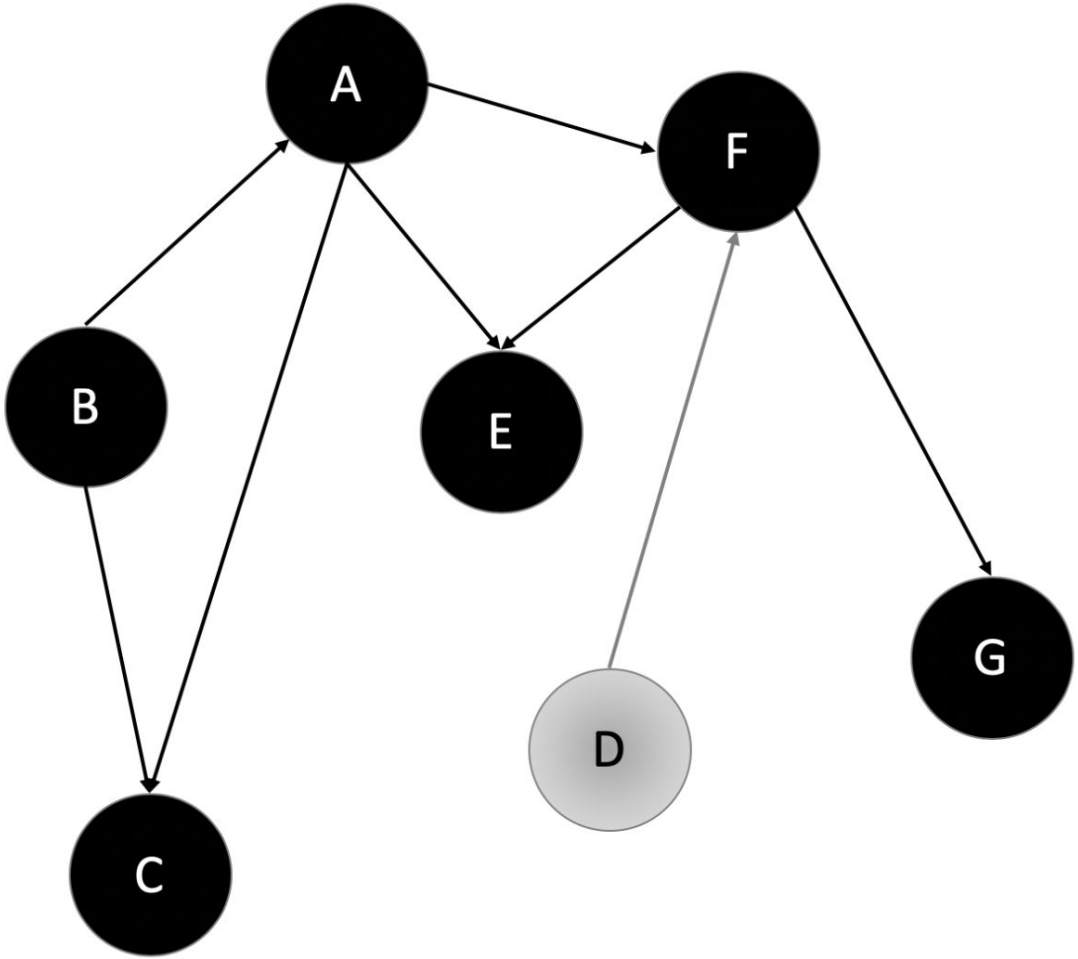



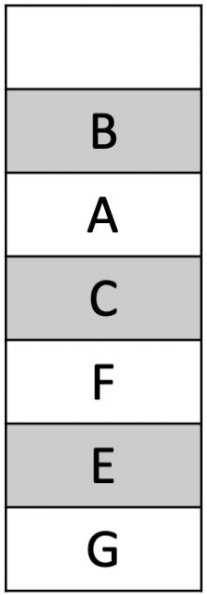

Stack S



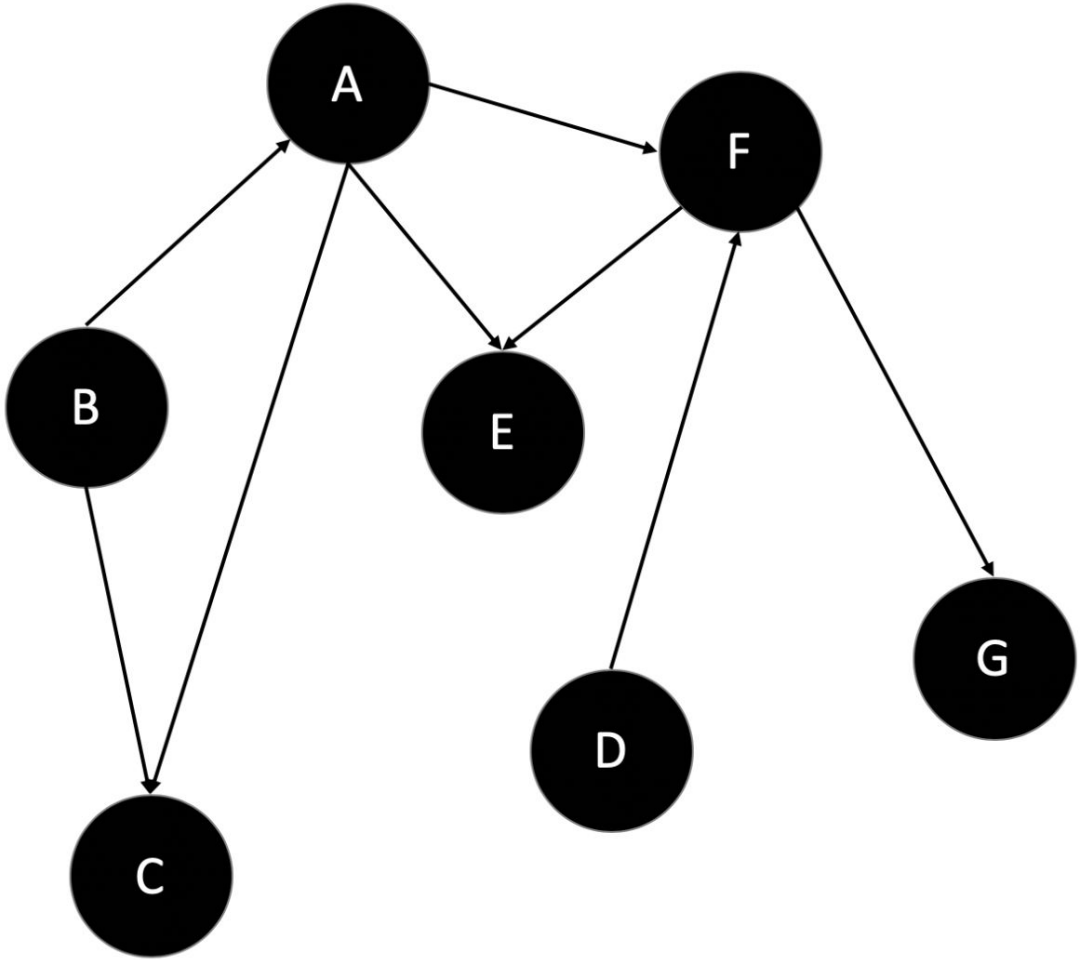




Stack S



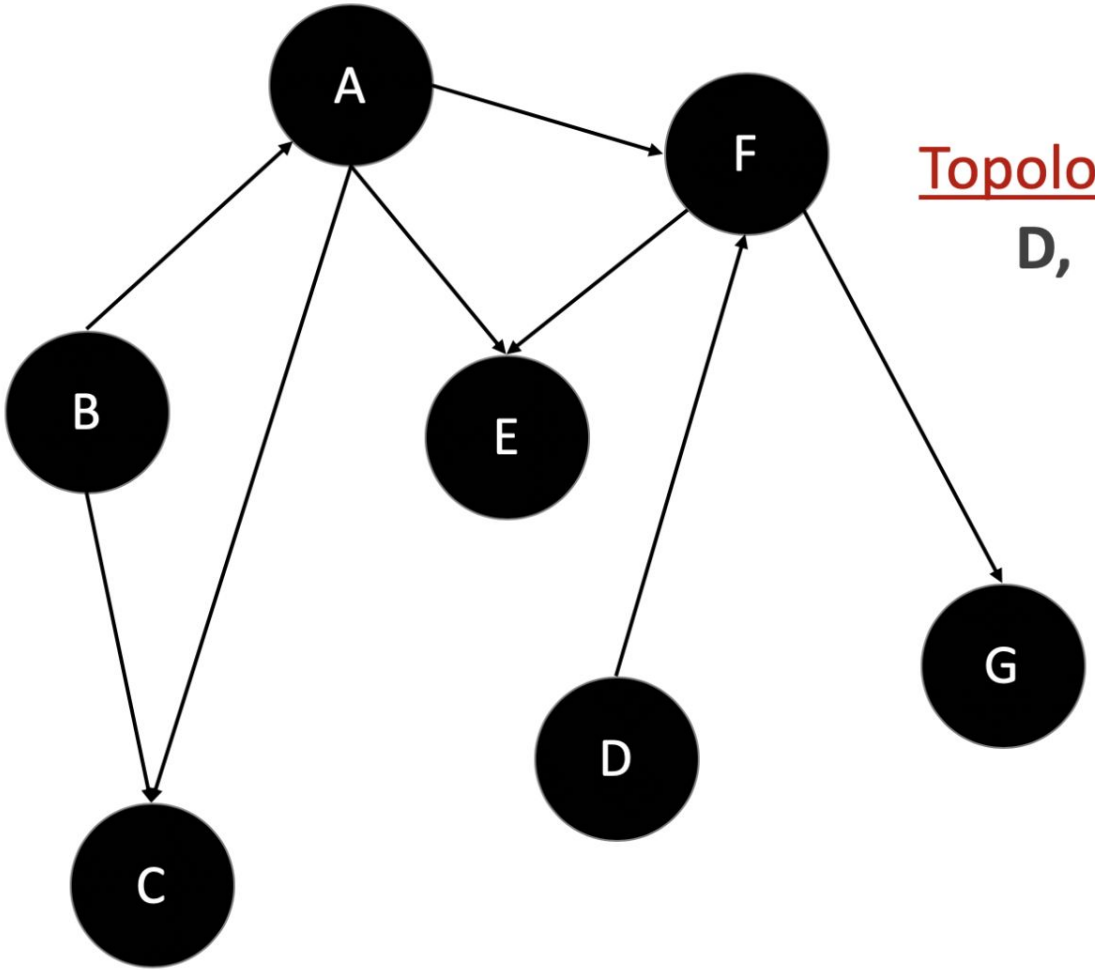




Stack S




| D |
|---|
| B |
| A |
| C |
| F |
| E |
| G |

Stack S




## Topologically Sorted Order

**D, B, A, C, F, E, G**

| D |
|---|
| B |
| A |
| C |
| F |
| E |
| G |

Stack S

# DFS Implementation

```
class Solution:
    def findOrder(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]
        # 0 -> White, 1 -> Gray, 2 -> Black
        colors = [0 for _ in range(numCourses)]
        order = []

        for course, pre in prerequisites:
            graph[pre].append(course)

        for node in range(numCourses):
            if colors[node] != 0:
                continue
            if not self.topSort(node, colors, graph,
order):
                return []

        return reversed(order)
```

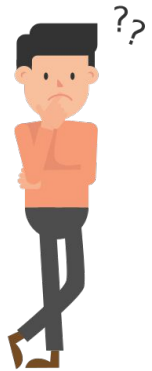
```
# This function creates a topological ordering.
# It returns False if it finds a cycle.
def topSort(self, node, colors, graph, order):
    # Cycle found - Why?
    if colors[node] == 1:
        return False

    colors[node] = 1
    for neighbor in graph[node]:
        if colors[neighbor] == 2:
            continue
        if not self.topSort(neighbor, colors, graph, order):
            return False

    # Color nodes black as we backtrack
    colors[node] = 2
    order.append(node)
    return True
```

# DFS Implementation

Can you think of another implementation where we don't have reverse the final answer?



## Time Complexity

- Let **V** be **number of Nodes (Courses)** and **E** be **number of edges** .
- We are traversing through the entire graph and **we will only visit a node once**. Which would take a time complexity of **O(V + E).**

## Space Complexity

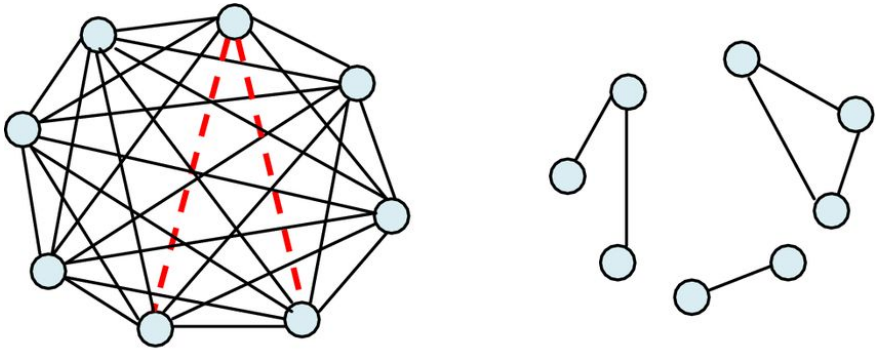
- Let **V** be **number of Nodes (Courses)** and **E** be **number of edges** .
- We will store the nodes, and also the edges.

  Space Complexity = **O (V + E)**
- What would the space complexity be if we ignore the complexity of building the graph?

# Practice Problem

# When do we use which implementation

What implementation would be better for a given DAG which is **very sparse**, with n vertices and m edges, and you need to find a topological ordering of the vertices?



# When do we use which implementation

- One possible algorithm to use is **Kahn's algorithm (BFS)**, which uses a queue to keep track of vertices with no incoming edges.
- However, suppose that the DAG is very sparse, with only a few edges relative to the number of vertices. In this case, using Kahn's algorithm may not be the most efficient choice, as it requires maintaining a queue with a large number of empty slots.

# When do we use which implementation

- A better algorithm in this case might be **a variation of depth-first search (DFS)** that uses a stack to store the vertices in reverse topological order.
- This algorithm has a space complexity of **O(V)**, which is much more efficient than Kahn's algorithm when the graph is sparse.

# Real Life Applications

Here are some notable applications of topological sort

- Task Scheduling
    - Topological sort ensures tasks are sequenced correctly by analyzing task dependencies in a DAG.
- Software Dependency Resolution
    - Topological sort resolves dependencies between software components, ensuring correct order during compilation or deployment.

# Real Life Applications Continued

- Compiler Optimizations
  - Topological sort optimizes code generation in compilers, ensuring correct variable allocation and efficient loop compilation techniques.
- DeadLock detection
- Course Scheduling

# Recognizing in Questions

# Directed Acyclic Graph (DAG)

## 2192. All Ancestors of a Node in a Directed Acyclic Graph

Difficulty Hidden 625 8 Add to List Share

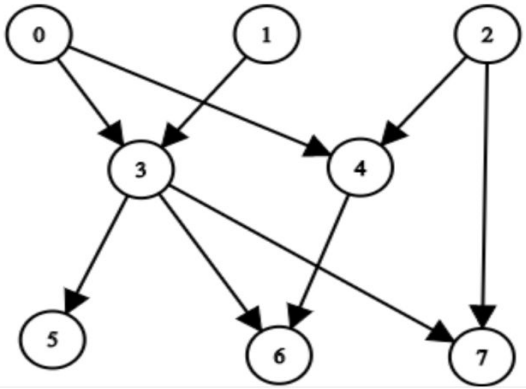
You are given a positive integer `n` representing the number of nodes of a **Directed Acyclic Graph** (DAG). The nodes are numbered from `0` to `n - 1` (**inclusive**).

You are also given a 2D integer array `edges`, where `edges[i] = [from_i, to_i]` denotes that there is a **unidirectional** edge from `from_i` to `to_i` in the graph.

Return *a list* `answer`, *where* `answer[i]` *is the* ***list of ancestors*** *of the* `i^th` *node, sorted in* ***ascending order***.

A node `u` is an **ancestor** of another node `v` if `u` can reach `v` via a set of edges.

**Example 1:**



# Directed Acyclic Graph (DAG)

Competitions › Topological sort

Problems Leaderboard

## Avengers

Time limit **2** seconds

Memory limit **128** MiB

NurlashKo, Nurbakyt, and Zhora are members of the last ninja clan fighting against even emperor Ren'swild reign. After devastating defeat in an open battle, they decided to split their army into three camps and wage a guerrilla war.

One of Emperor Ren's ridiculous reforms allows to pass roads between cities only in one direction. Also, he chose the allowed directions of the roads in such way, so that it's impossible to start and return to the same city after passing several roads.

Right now, the clan is deciding where to place their camps. Emperor Ren's army makes regular raids inspecting some path. If Army crushes all three of the camps during their raid, clan wouldn't be able to regroup and would lose the war. Help the clan to choose three cities, so that there is no path that passes through all three of these cities.

# Courses / Prerequisites

## 1462. Course Schedule IV

Difficulty Hidden 1019 53 Add to List Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [a_i, b_i]` indicates that you **must** take course `a_i` first if you want to take course `b_i`.

- For example, the pair `[0, 1]` indicates that you have to take course `0` before you can take course `1`.

Prerequisites can also be **indirect**. If course `a` is a prerequisite of course `b`, and course `b` is a prerequisite of course `c`, then course `a` is a prerequisite of course `c`.

You are also given an array `queries` where `queries[j] = [u_j, v_j]`. For the `j^th` query, you should answer whether course `u_j` is a prerequisite of course `v_j` or not.

Return *a boolean array* `answer`, *where* `answer[j]` *is the answer to the* `j^th` *query.*

**Example 1:**


```
Input: numCourses = 2, prerequisites = [[1,0]], queries = [[0,1],[1,0]]
Output: [false,true]
Explanation: The pair [1, 0] indicates that you have to take course 1 before you can take course 0.
Course 0 is not a prerequisite of course 1, but the opposite is true.
```

# Dependency

## 1203. Sort Items by Groups Respecting Dependencies

Difficulty Hidden 708 95 Add to List Share

There are `n` items each belonging to zero or one of `m` groups where `group[i]` is the group that the `i`-th item belongs to and it's equal to `-1` if the `i`-th item belongs to no group. The items and the groups are zero indexed. A group can have no item belonging to it.

Return a sorted list of the items such that:

- The items that belong to the same group are next to each other in the sorted list.
- There are some relations between these items where `beforeItems[i]` is a list containing all the items that should come before the `i`-th item in the sorted array (to the left of the `i`-th item).

Return any solution if there is more than one solution and return an **empty list** if there is no solution.

**Example 1:**

| Item | Group | Before |
|---|---|---|
| 0 | -1 | |
| 1 | -1 | 6 |
| 2 | 1 | 5 |
| 3 | 0 | 6 |
| 4 | 0 | 3, 6 |
| 5 | 1 | |
| 6 | 0 | |
| 7 | -1 | |

# Scheduling

## Tasks Scheduling

There are 'N' tasks, labeled from '0' to 'N-1'. Each task can have some prerequisite tasks which need to be completed before it can be scheduled. Given the number of tasks and a list of prerequisite pairs, find out if it is possible to schedule all the tasks.

**Example 1:**

```
Input: Tasks=3, Prerequisites=[0, 1], [1, 2]
Output: true
Explanation: To execute task '1', task '0' needs to finish first. Similarly, task '
before '2' can be scheduled. A possible scheduling of tasks is: [0, 1, 2]
```

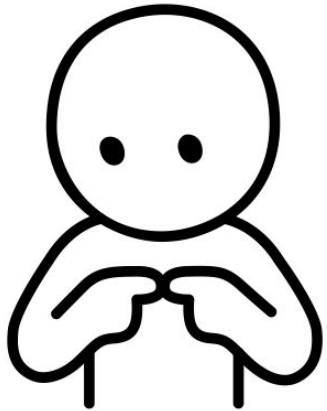
**Example 2:**

```
Input: Tasks=3, Prerequisites=[0, 1], [1, 2], [2, 0]
Output: false
Explanation: The tasks have cyclic dependency, therefore they cannot be scheduled.
```

As you can see, we can represent the task and its dependency as an edge in a directed graph. There's a little change in the above code that need to be changed to solve this problem so I let this as an exercise for you. Instead of returning a list, you just need to return a boolean.

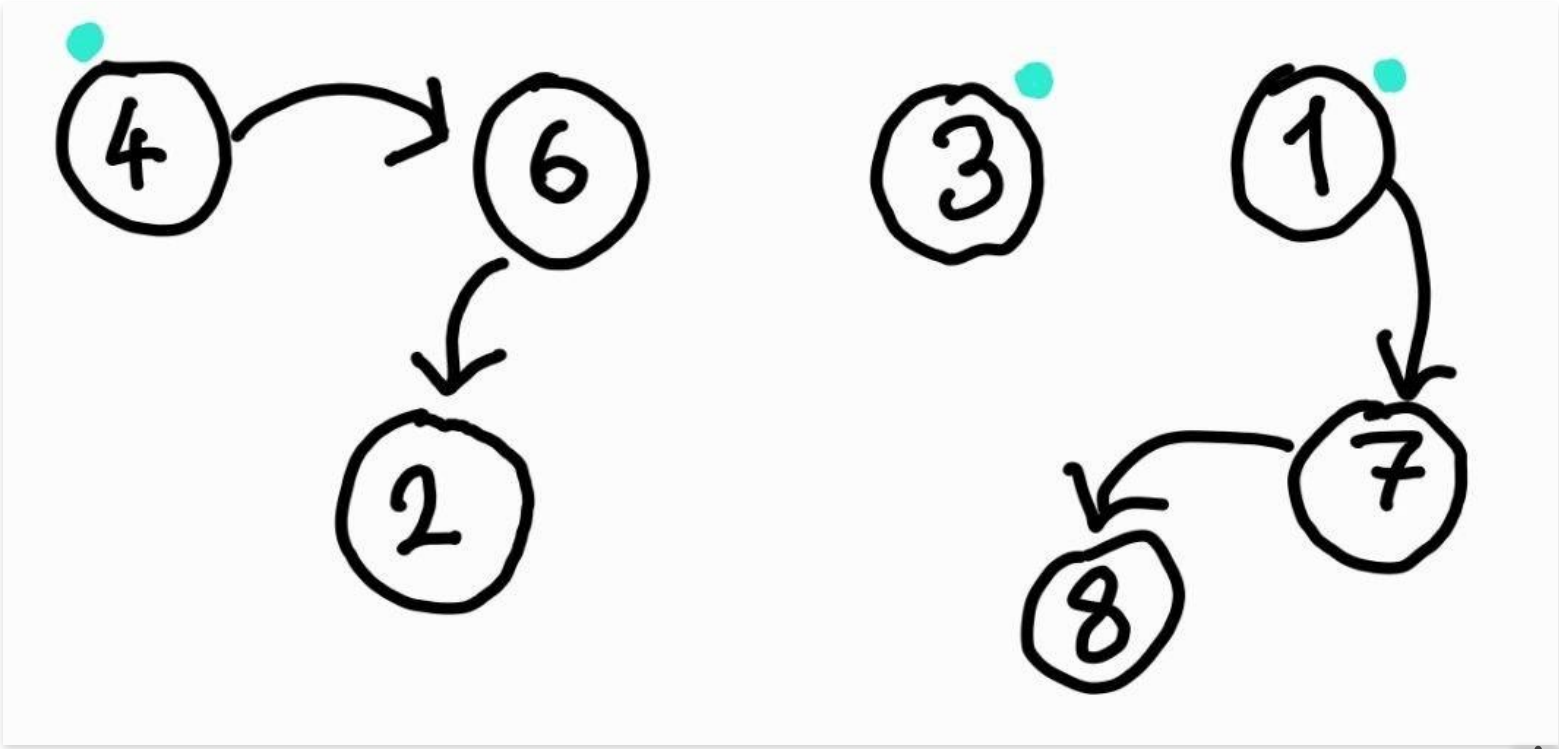
Another application of Topological Sort is to check if a directed graph contains a cycle.

# Common Pitfalls



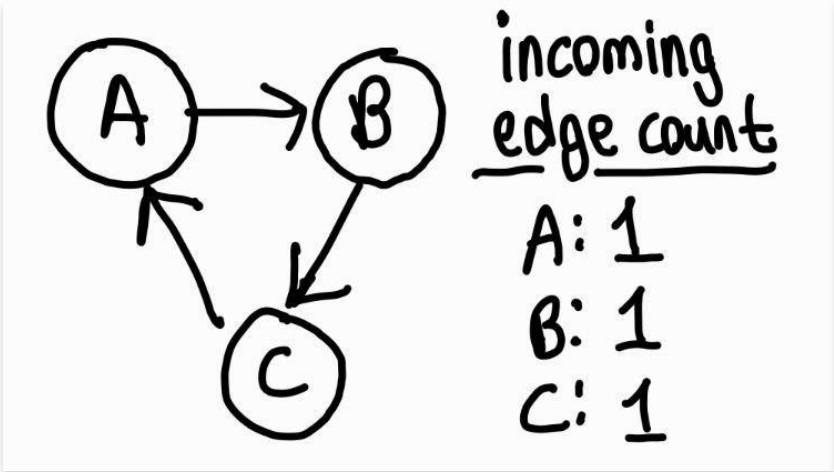
# Multiple Components

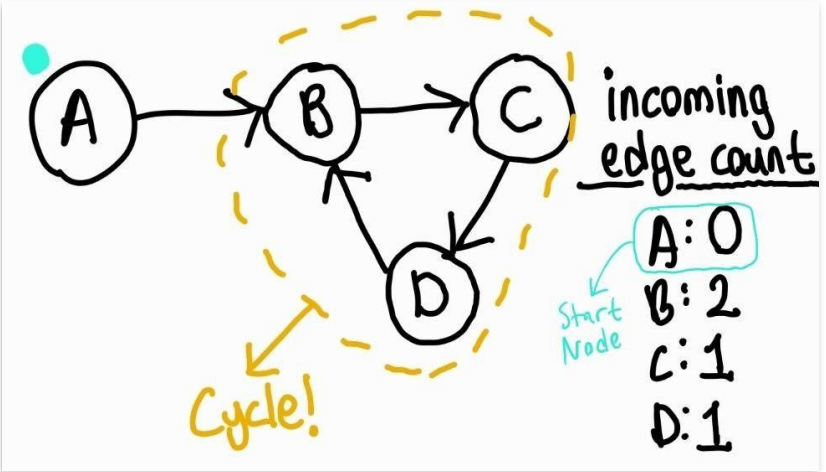
Graph may contain **multiple components**, don't forget to **traverse all the components**



# Cycle Detection Common Pitfalls

Don't try to detect cycle only by using incoming edge counts

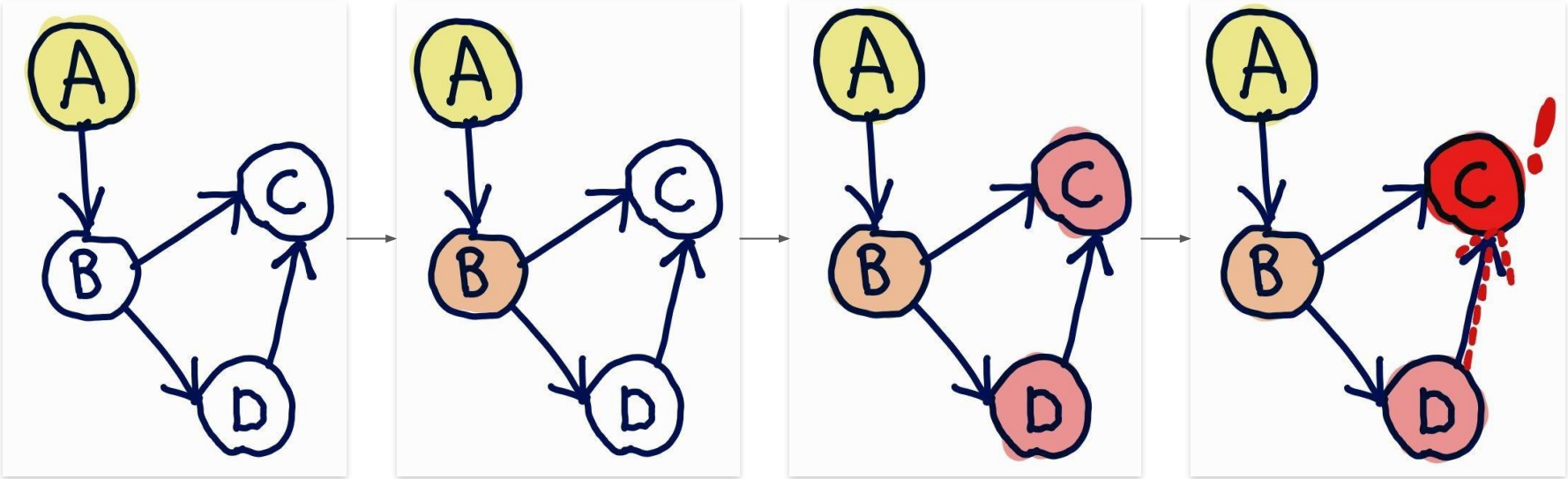


# Cycle Detection Common Pitfalls - 2

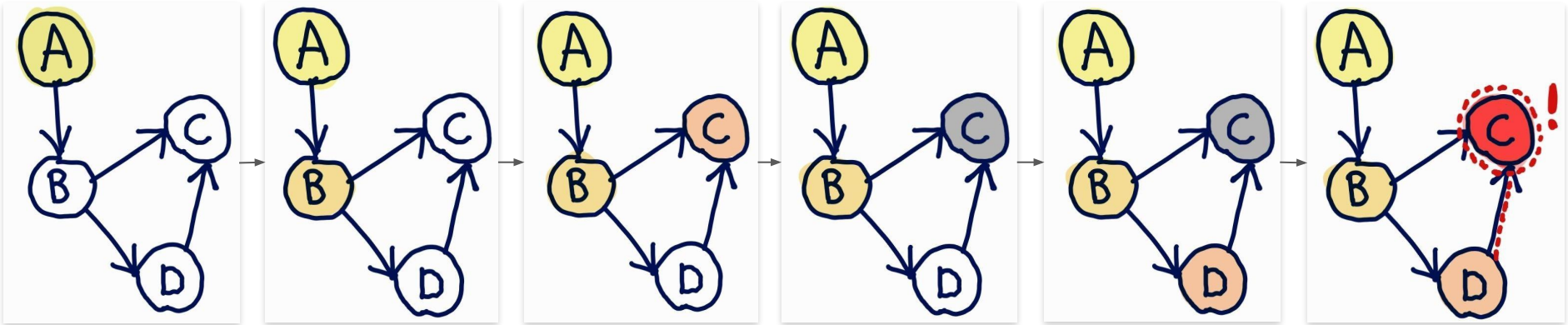
Don't try to use vanilla BFS or DFS to detect cycles

Why Vanilla BFS with Storing Visited Nodes won't Work?



# Cycle Detection Common Pitfalls - 2

Why DFS with Storing Visited Nodes won't Work?



# Resources

- Topological Sort - TopCoder
- Kahn's Algorithm - TakeUForward
- Topological Sort using DFS - TakeUForward
- Topological Sort Visualization

# Practice Problems

Course Schedule

All Ancestors of a Node in a Directed Acyclic Graph

Alien Dictionary

Loud and Rich

Course Schedule IV

Minimum Height Trees

Parallel Courses III

Sort Items by Groups Respecting Dependencies

Build a Matrix With Conditions

Longest Cycle in a Graph

Gardener and Tree

# For the curious: Real-world implementation of Topological Sorting

(Not a comprehensive list)

- MakeFiles in Software Building Procedure
- Apache Airflow usually in MLOps
- Version Control Systems such as Git
- Compilers such as GCC

# Quote of the day

"Order is the shape upon which beauty depends."

By Pearl S. Buck