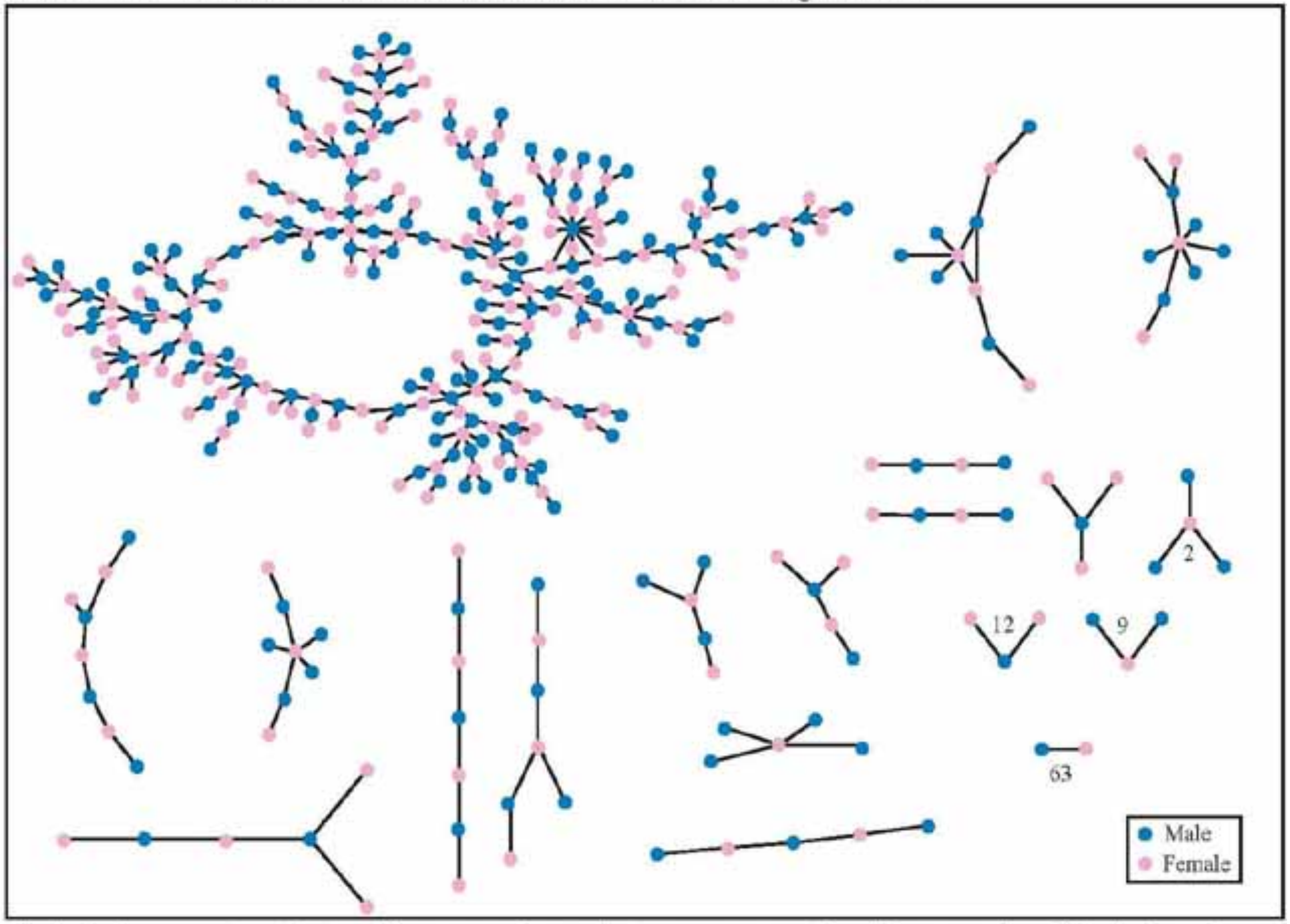
CS3000: Algorithms & Data
Jonathan Ullman

Lecture 10:
- Graphs
- Graph Traversals: DFS
- Topological Sort

Feb 19, 2020

What's Next

The Structure of Romantic and Sexual Relations at "Jefferson High School"



Each circle represents a student and lines connecting students represent romantic relations occuring within the 6 months preceding the interview. Numbers under the figure count the number of times that pattern was observed (i.e. we found 63 pairs unconnected to anyone else).

What’s Next

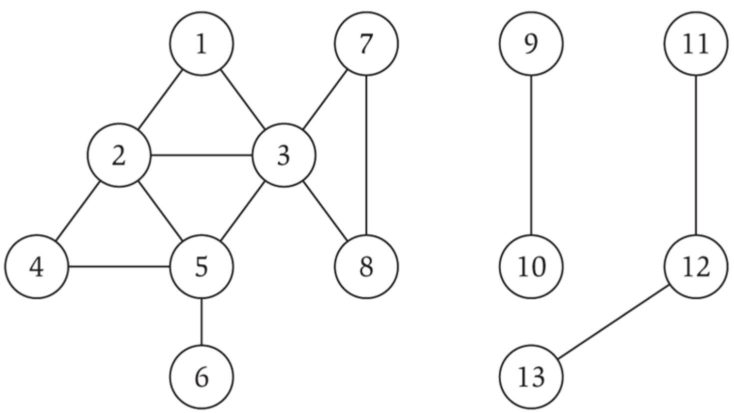
- **Graph Algorithms:**
  - **Graphs:** Key Definitions, Properties, Representations
  - **Exploring Graphs:** Breadth/Depth First Search
    - Applications: Connectivity, Bipartiteness, Topological Sorting
  - **Shortest Paths:**
    - Dijkstra
    - Bellman-Ford (Dynamic Programming)
  - **Minimum Spanning Trees:**
    - Borůvka, Prim, Kruskal
  - **Network Flow:**
    - Algorithms
    - Reductions to Network Flow

Graphs

Graphs: Key Definitions

- **Definition:** A directed graph $G = (V, E)$
  - V is the set of nodes/vertices
  - $E \subseteq V \times V$ is the set of edges
  - An edge is an ordered $e = (u, v)$ "from u to v"
- **Definition:** An undirected graph $G = (V, E)$
  - Edges are unordered $e = (u, v)$ "between u and v"

- **Simple Graph:**
  - No duplicate edges
  - No self-loops $e = (u, u)$

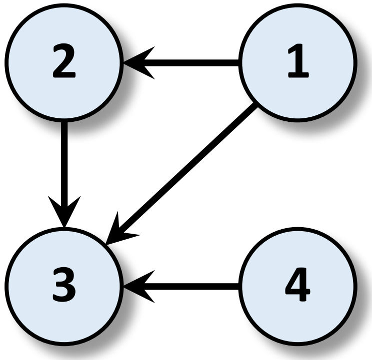


Adjacency Matrices

- The adjacency matrix of a graph $G = (V, E)$ with n nodes is the matrix $A[1{:}n, 1{:}n]$ where

$$A[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$

A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0



Cost

Space: $\Theta(V^2)$

Lookup: $\Theta(1)$ time

List Neighbors: $\Theta(V)$ time

Adjacency Lists (Undirected)

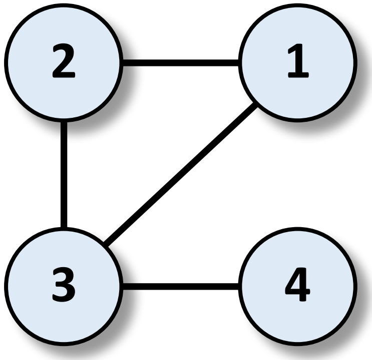
- The adjacency list of a vertex $v \in V$ is the list $A[v]$ of all u s.t. $(v, u) \in E$

$A[1] = \{2,3\}$

$A[2] = \{1,3\}$

$A[3] = \{1,2,4\}$

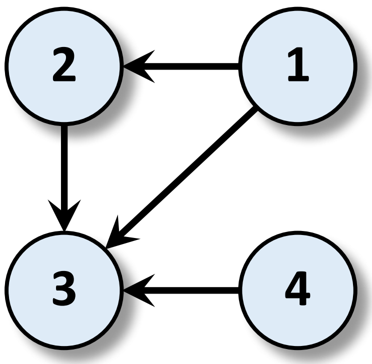
$A[4] = \{3\}$



Adjacency Lists (Directed)

- The adjacency list of a vertex $v \in V$ are the lists
  - $A_{out}[v]$ of all u s.t. $(v, u) \in E$
  - $A_{in}[v]$ of all u s.t. $(u, v) \in E$

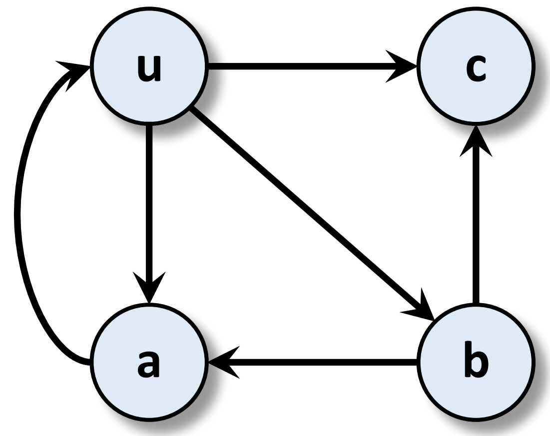
$A_{out}[1] = \{2,3\}$	$A_{in}[1] = \{\ \}$
$A_{out}[2] = \{3\}$	$A_{in}[2] = \{1\}$
$A_{out}[3] = \{\ \}$	$A_{in}[3] = \{1,2,4\}$
$A_{out}[4] = \{3\}$	$A_{in}[4] = \{\ \}$



Depth-First Search (DFS)

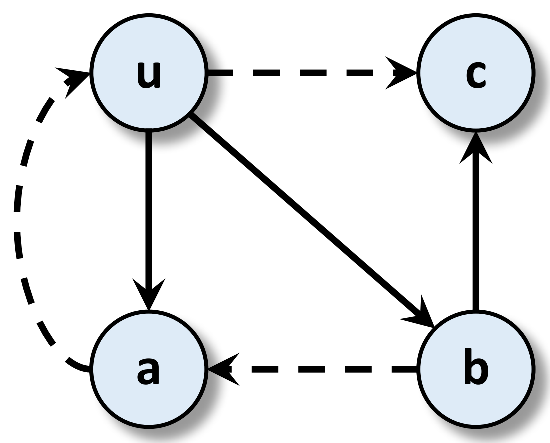
Depth-First Search

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
```

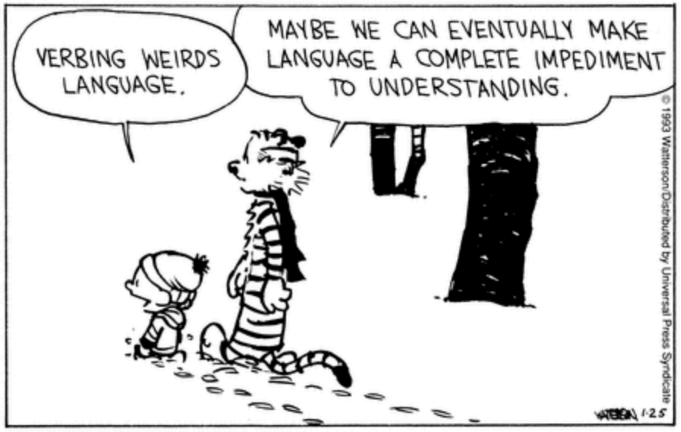


Depth-First Search

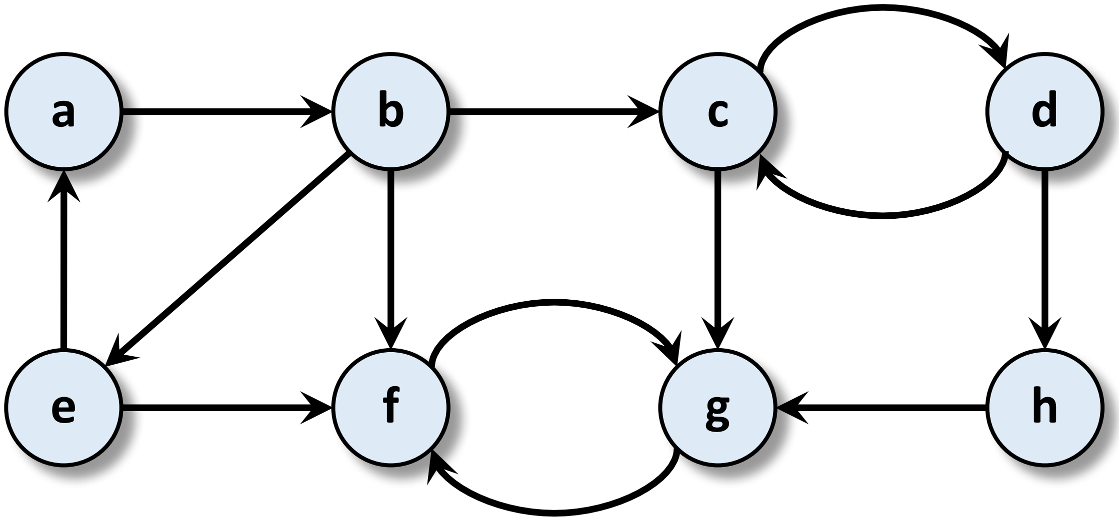
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - **Tree edges:** $(u, a), (u, b), (b, c)$
    - These are the edges that explore new nodes
  - **Forward edges:** (u, c)
    - Ancestor to descendant
  - **Backward edges:** (a, u)
    - Descendant to ancestor
    - Implies a directed cycle!
  - **Cross edges:** (b, a)
    - No ancestral relation



Ask the Audience



- DFS starting from node a
  - Search in alphabetical order
  - Label edges with {tree,forward,backward,cross}



Connected Components

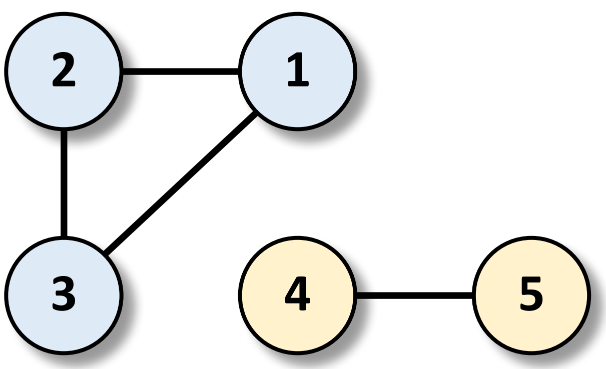
Paths/Connectivity

- A path is a sequence of consecutive edges in E
  - $P = u - w_1 - w_2 - w_3 - \cdots - w_{k-1} - v$
  - The length of the path is the # of edges

- An undirected graph is connected if for every two vertices $u, v \in V$, there is a path from u to v
- A directed graph is strongly connected if for every two vertices $u, v \in V$, there are paths from u to v and from v to u

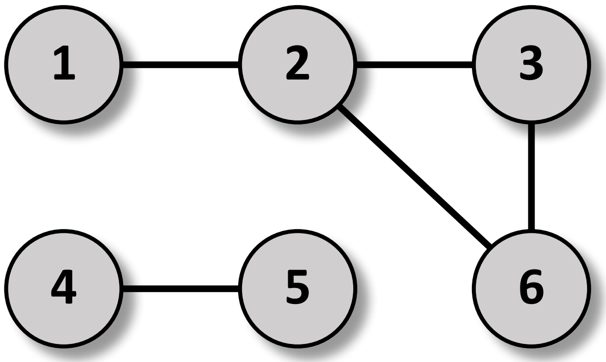
Connected Components (Undirected)

- **Problem:** Given an undirected graph G, split it into connected components
- **Input:** Undirected graph $G = (V, E)$
- **Output:** A labeling of the vertices by their connected component



Connected Components (Undirected)

- **Algorithm:**
  - Pick a node v
  - Use DFS to find all nodes reachable from v
  - Labels those as one connected component
  - Repeat until all nodes are in some component



Connected Components (Undirected)

```
CC(G = (V,E)):
  // Initialize an empty array and a counter
  let comp[1:n] ← ⊥, c ← 1

  // Iterate through nodes
  for (u = 1,…,n):
    // Ignore this node if it already has a comp.
    // Otherwise, explore it using DFS
    if (comp[u] != ⊥):
      run DFS(G,u)
      let comp[v] ← c for every v found by DFS
      let c ← c + 1

  output comp[1:n]
```

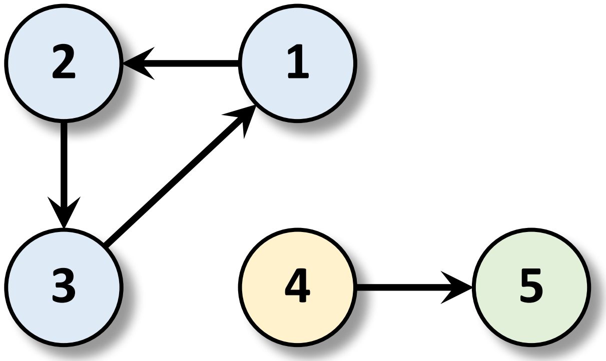
Running Time

Connected Components (Undirected)

- **Problem:** Given an undirected graph G, split it into connected components
- **Algorithm:** Can split a graph into conneted components in time $O(n + m)$ using DFS
- **Punchline:** Usually assume graphs are connected
  - Implicitly assume that we have already broken the graph into CCs in $O(n + m)$ time

Strong Components (Directed)

- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component



Strong Components (Directed)

- **Observation:** $\mathrm{SCC}(s)$ is all nodes $v \in V$ such that v is reachable from s and vice versa
  - Can find all nodes reachable from s using BFS
  - How do we find all nodes that can reach s?

Strong Components (Directed)

```
SCC(G = (V,E)):
  let Gᴿ be G with all edges "reversed"

  // Initialize an array and counter
  let comp[1:n] ← ⊥, c ← 1

  for (u = 1,…,n):
    // If u has not been explored
    if (comp[u] != ⊥):
      let S be the nodes found by DFS(G,u)
      let T be the nodes found by DFS(Gᴿ,u)
      // S ∩ T contains SCC(u)
      label S ∩ T with c
      let c ← c + 1

  return comp
```

Strong Components (Directed)

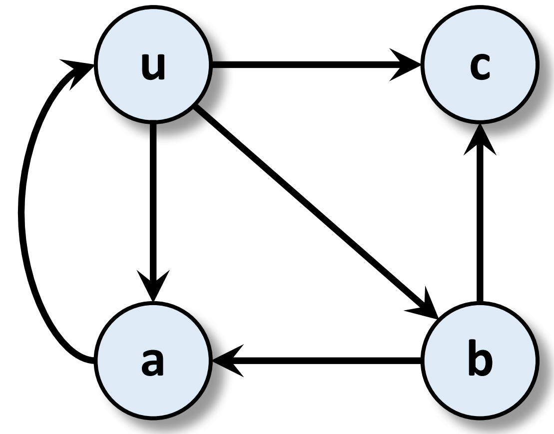
- **Problem:** Given a directed graph G, split it into strongly connected components
- **Input:** Directed graph $G = (V, E)$
- **Output:** A labeling of the vertices by their strongly connected component

- Find SCCs in $O(n^2 + nm)$ time using DFS
- **Can find SCCs in $O(n + m)$ time** using a more clever version of DFS

Post-Ordering

Post-Ordering

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)

  post-visit(u)
```

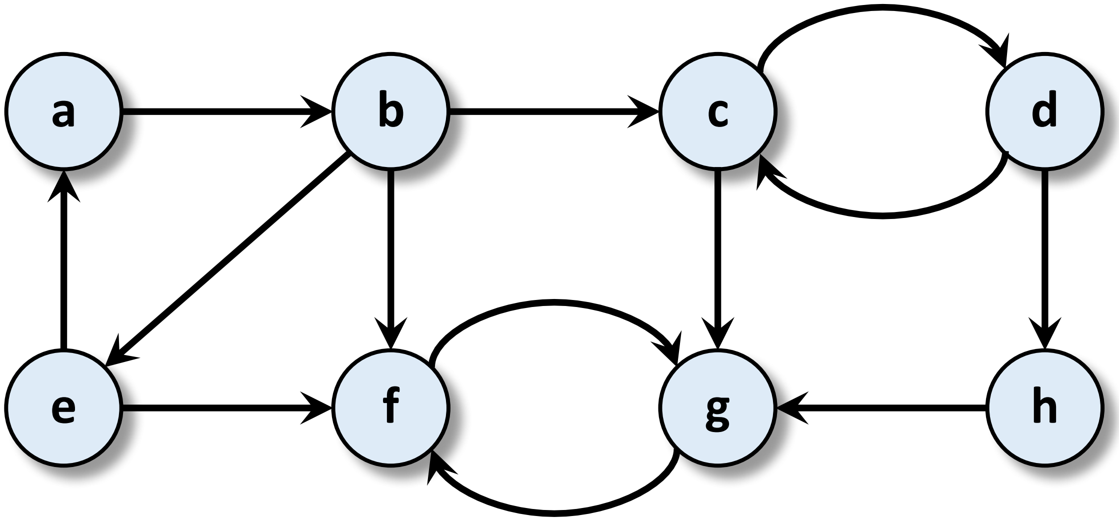


Vertex	Post-Order

- Maintain a counter **clock**, initially set **clock = 1**
- **post-visit(u):**
  set postorder[u]=clock, clock=clock+1

Example

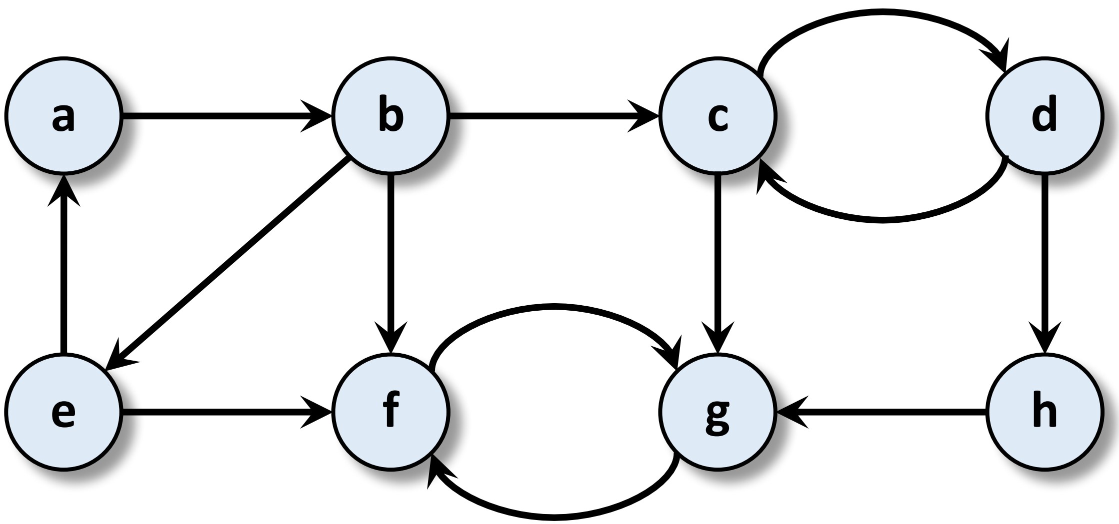
- Compute the **post-order** of this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Post-Order								

Example

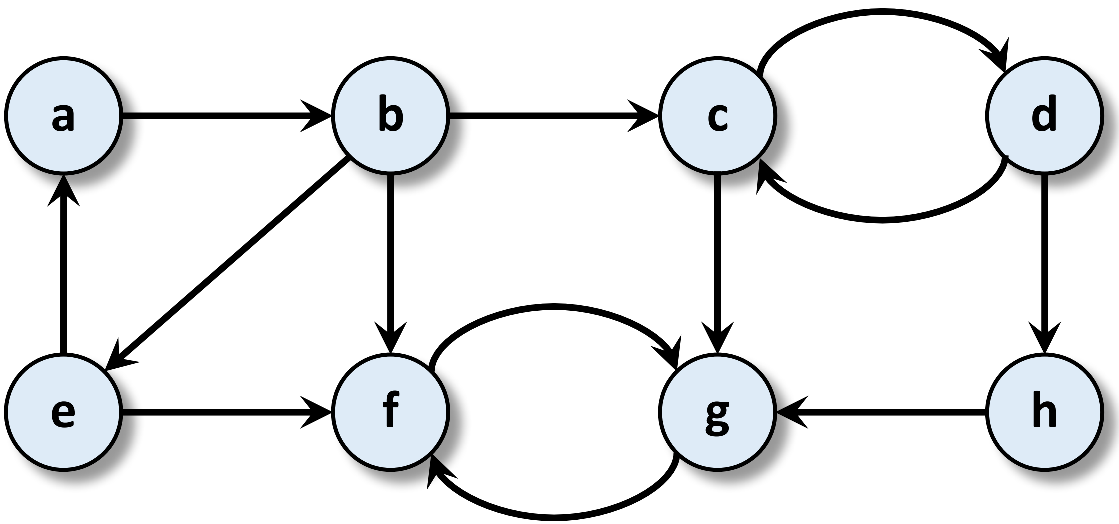
- Compute the **post-order** of this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

Obervation

- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

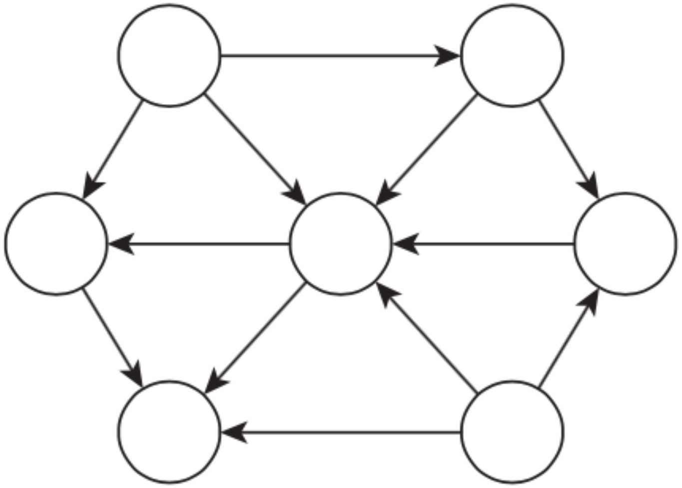
Observation

- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge
  - DFS(u) can’t finish until its children are finished
  - If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), thus DFS(v) is not called by DFS(u)
  - When we ran DFS(u), we must have had explored[v]=1
    - Thus, DFS(v) started before DFS(u)
  - DFS(v) started before DFS(u) but finished after
    - Can only happen for a backward edge

Topological Ordering

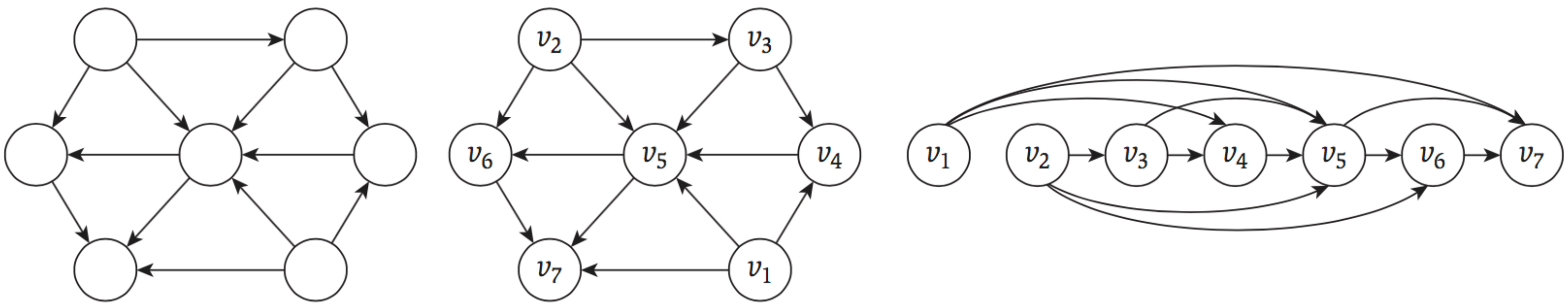
Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- Can be much more complex than a forest



Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \dots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG

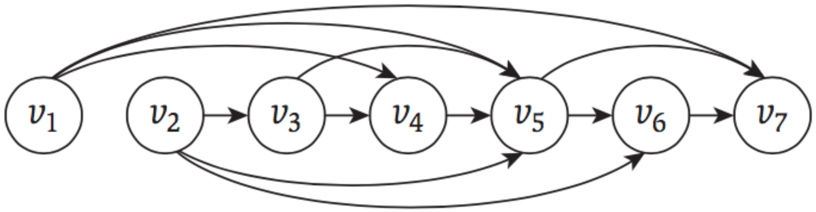
Directed Acyclic Graphs (DAGs)

- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **Thm:** G has a topological ordering $\Leftrightarrow$ G is a DAG
  - We will design one algorithm that either outputs a topological ordering or finds a directed cycle

Topological Ordering

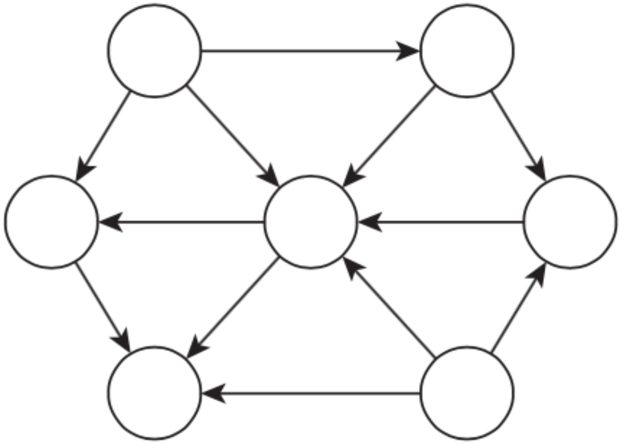
- **Observation:** the first node must have no in-edges



- **Observation:** In any DAG, there is always a node with no incoming edges

Topological Ordering

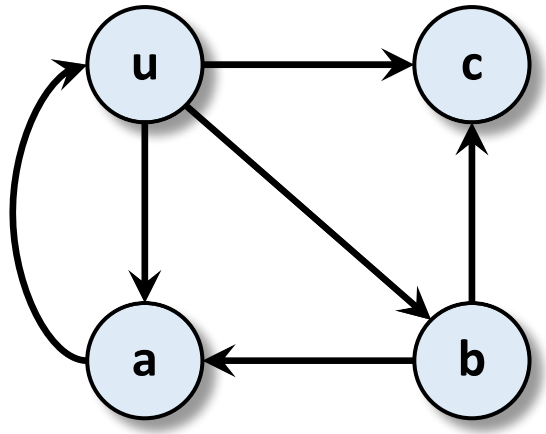
- **Fact:** In any DAG, there is a node with no incoming edges
- **Thm:** Every DAG has a topological ordering
- **Proof (Induction):**



Faster Topological Ordering

Post-Ordering

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)

  post-visit(u)
```

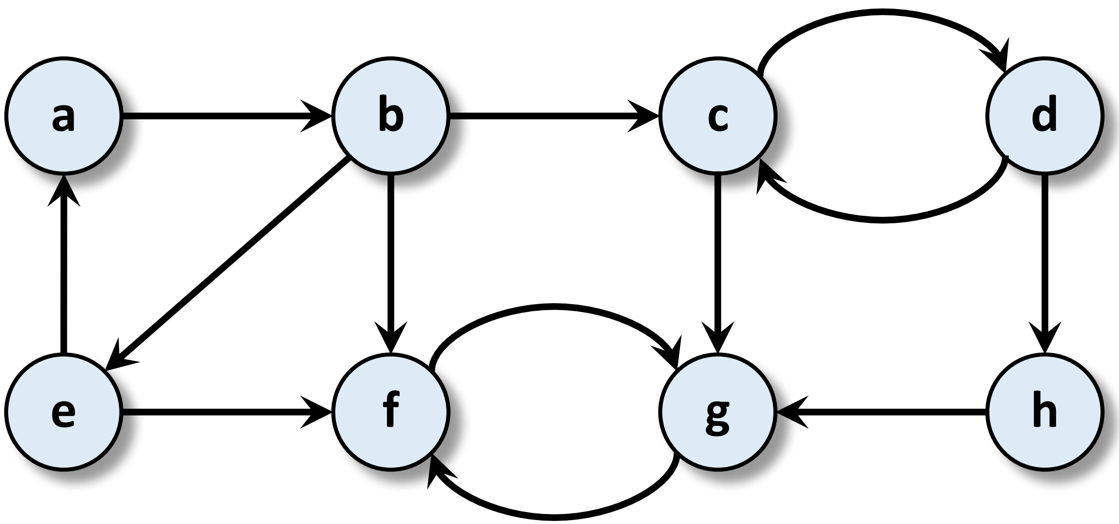


Vertex	Post-Order

- Maintain a counter **clock**, initially set **clock = 1**
- **post-visit(u):**
  set postorder[u]=clock, clock=clock+1

Example

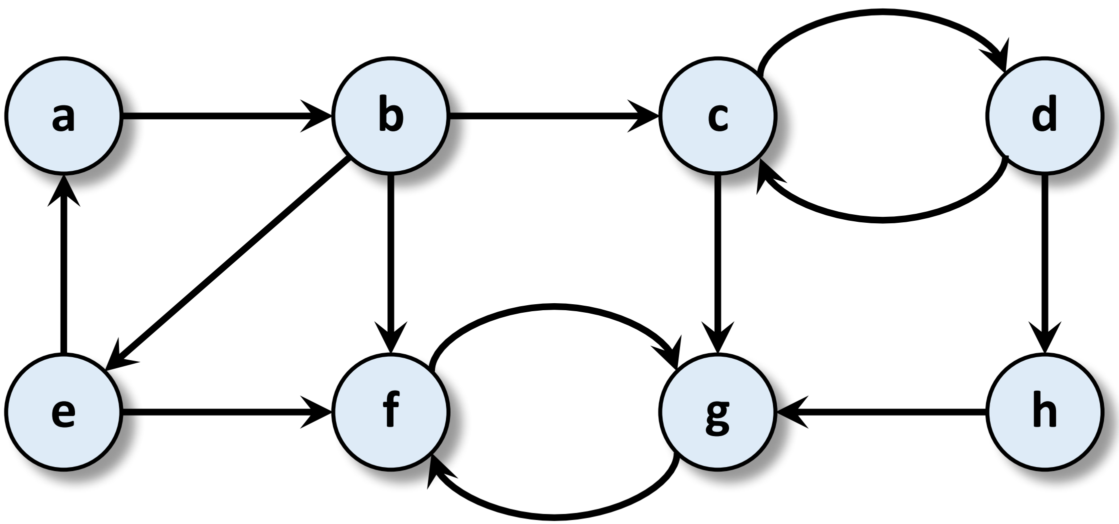
- Compute the **post-order** of this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Post-Order								

Example

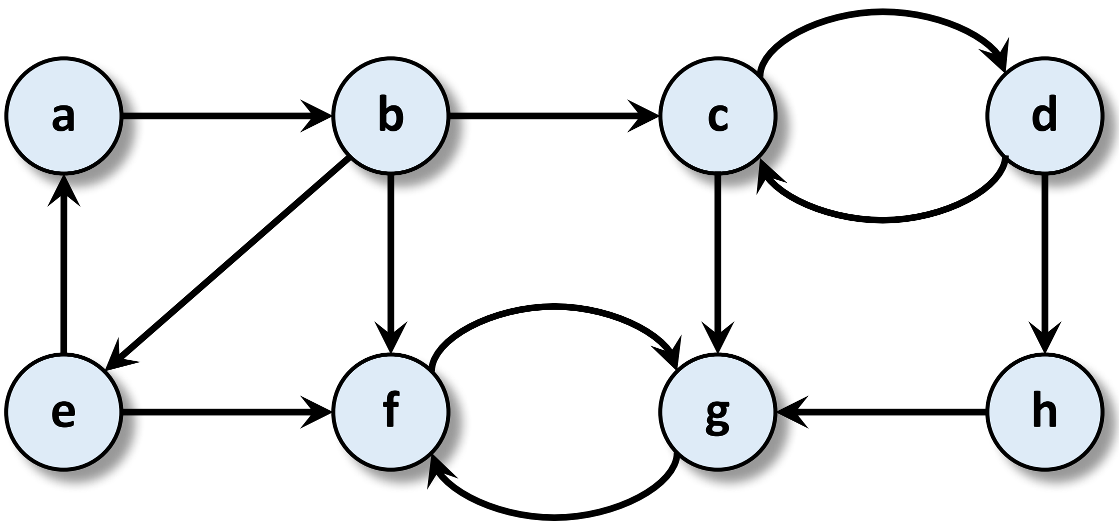
- Compute the **post-order** of this graph
  - DFS from a, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

Obervation

- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge



Vertex	a	b	c	d	e	f	g	h
Post-Order	8	7	5	4	6	1	2	3

Observation

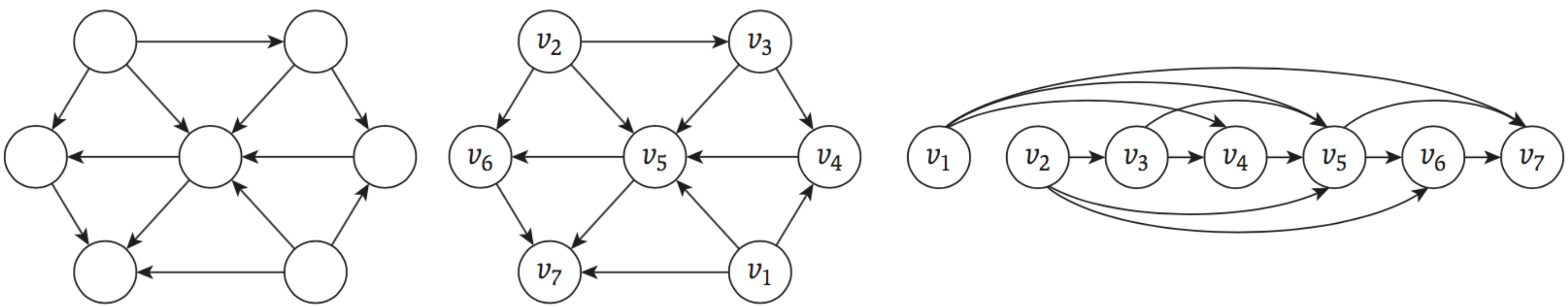
- **Observation:** if postorder[u] < postorder[v] then (u,v) is a backward edge
  - DFS(u) can’t finish until its children are finished
  - If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), thus DFS(v) is not called by DFS(u)
  - When we ran DFS(u), we must have had explored[v]=1
    - Thus, DFS(v) started before DFS(u)
  - DFS(v) started before DFS(u) but finished after
    - Can only happen for a backward edge

Fast Topological Ordering

- **Claim:** ordering nodes by decreasing postorder gives a topological ordering
- **Proof:**
  - A DAG has no backward edges
  - Suppose this is **not** a topological ordering
    - That means there exists an edge (u,v) such that postorder[u] < postorder[v]
    - We showed that any such (u,v) is a backward edge
    - But there are no backward edges, contradiction!

Topological Ordering (TO)

- **DAG:** A directed graph with no directed cycles
- Any DAG can be **toplogically ordered**
  - Label nodes $v_1, \ldots, v_n$ so that $(v_i, v_j) \in E \implies j > i$



- Can compute a TO in $O(n + m)$ time using DFS
  - Reverse of post-order is a topological order

Breadth-First Search

Exploring a Graph

- **Problem:** Is there a path from s to t?
- **Idea:** Explore all nodes reachable from s.

- Two different search techniques:
  - **Breadth-First Search:** explore nearby nodes before moving on to farther away nodes
  - **Depth-First Search:** follow a path until you get stuck, then go back

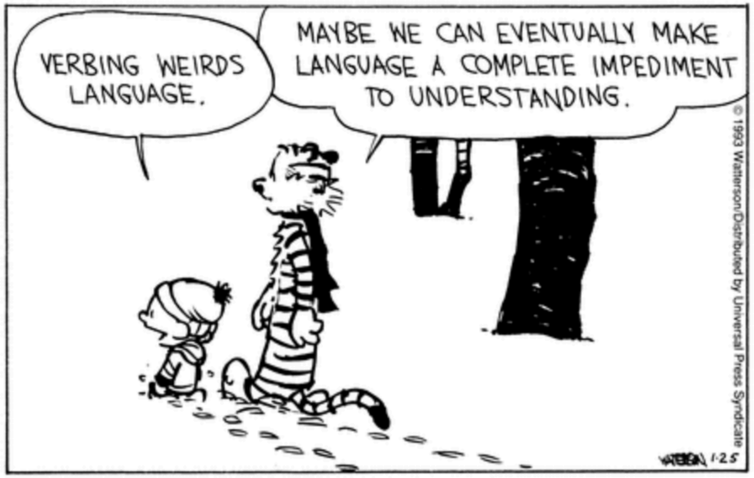
Breadth-First Search (BFS)

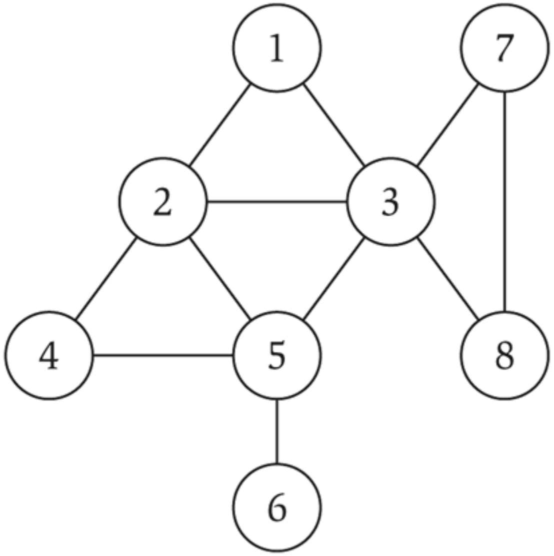
- **Informal Description:** start at s, find neighbors of s, find neighbors of neighbors of s, and so on…

- BFS Tree:
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1 that are not in L_0, L_1
  - $L_3 =$ all neighbors of L_2 that are not in L_0, L_1, L_2
  - …
  - $L_d =$ all neighbors of L_{d-1} that are not in $L_0, \ldots, L_{d-1}$
  - Stop when L_{d+1} is empty

Ask the Audience

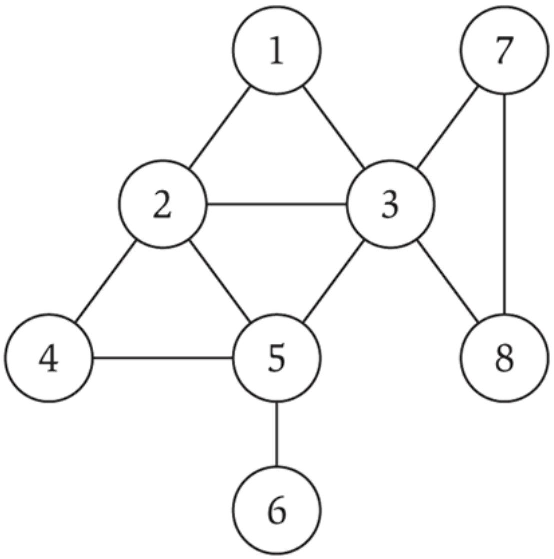
- BFS this graph from $s = 1$





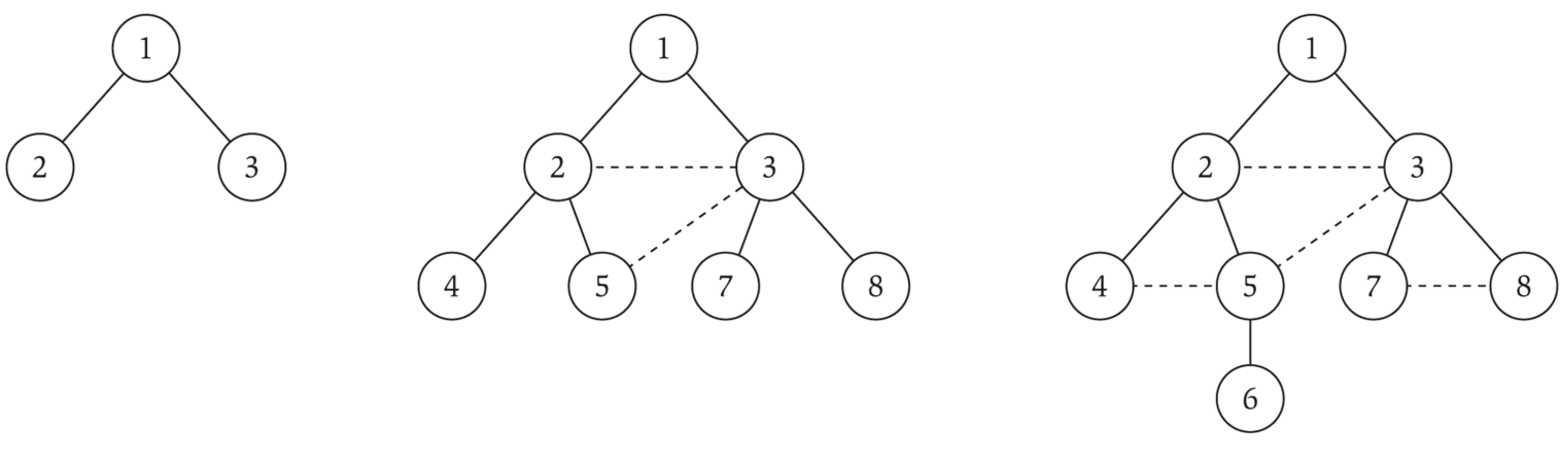
Ask the Audience

- BFS this graph from $\boldsymbol{s = 1}$



Breadth-First Search (BFS)

- **Definition:** the distance between s, t is the number of edges on the shortest path from s to t
- **Thm:** BFS finds distances from s to other nodes
  - L_i contains all nodes at distance i from s
  - Nodes not in any layer are not reachable from s



Breadth-First Search Implementation

```
BFS(G = (V,E), s):
  Let found[v] ← false ∀v
  Let found[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let i ← 0, L_0 = {s}, T ← Ø

  While (L_i is not empty):
    Initialize new layer L_{i+1}
    For (u in L_i):
      For ((u,v) in E):
        If (found[v] = false):
          found[v] ← true,
          layer[v] ← i+1
          Add (u,v) to T
          Add v to L_{i+1}
    i ← i+1
```

BFS Running Time (Adjacency List)

```
BFS(G = (V,E), s):
  Let found[v] ← false ∀v
  Let found[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let i ← 0, L0 = {s}, T ← ∅

  While (Li is not empty):
    Initialize new layer Li+1
    For (u in Li):
      For ((u,v) in E):
        If (found[v] = false):
          found[v] ← true,
          layer[v] ← i+1
          Add (u,v) to T
          Add v to Li+1
    i ← i+1
```

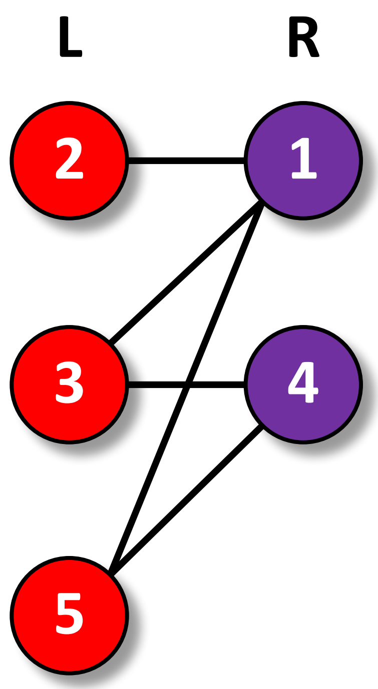
Bipartiteness / 2-Coloring

2-Coloring

- **Problem:** Tug-of-War Rematch
  - Need to form two teams $\boldsymbol{R}$, $\boldsymbol{P}$
  - Some students are still mad from last time
- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means u, v wont be on the same team
- **Output:** Split V into two sets $\boldsymbol{R}$, $\boldsymbol{P}$ so that no pair in either set is connected by an edge


2-Coloring (Bipartiteness)

- **Equivalent Problem:** Is the graph G bipartite?
  - A graph G is bipartite if I can split V into two sets L and R such that all edges $(u, v) \in E$ go between L and R



Designing the Algorithm

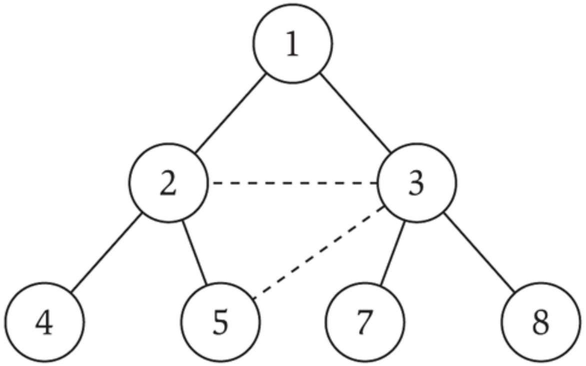
- **Key Fact:** If G contains a cycle of odd length, then G is not 2-colorable/bipartite

Designing the Algorithm

- **Idea for the algorithm:**
  - BFS the graph, coloring nodes as you find them
  - Color nodes in layer i **purple** if i even, **red** if i odd
  - See if you have succeeded or failed

Designing the Algorithm

- **Claim:** If BFS 2-colored the graph successfully, the graph has been 2-colored successfully
- **Key Question:** Suppose you have not 2-colored the graph successfully, maybe someone else can do it?



Designing the Algorithm

- **Claim:** If BFS fails, then G contains an odd cycle
  - If G contains an odd cycle then G can't be 2-colored!
  - Example of a phenomenon called **duality**