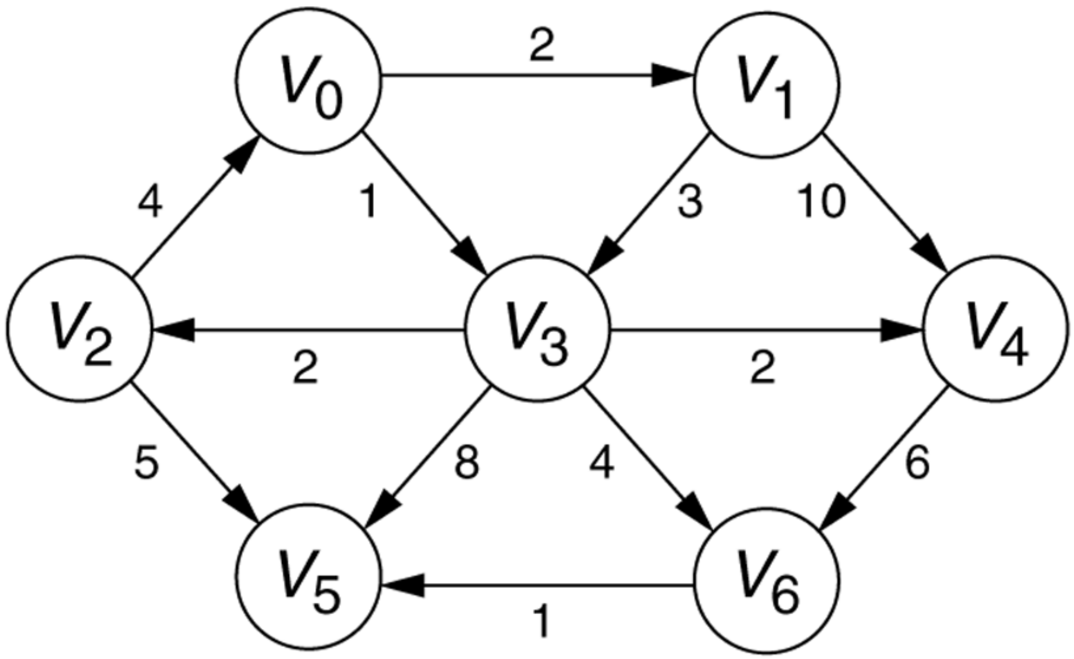
CSE 373: Data Structures and Algorithms

Lecture 18: Graphs II

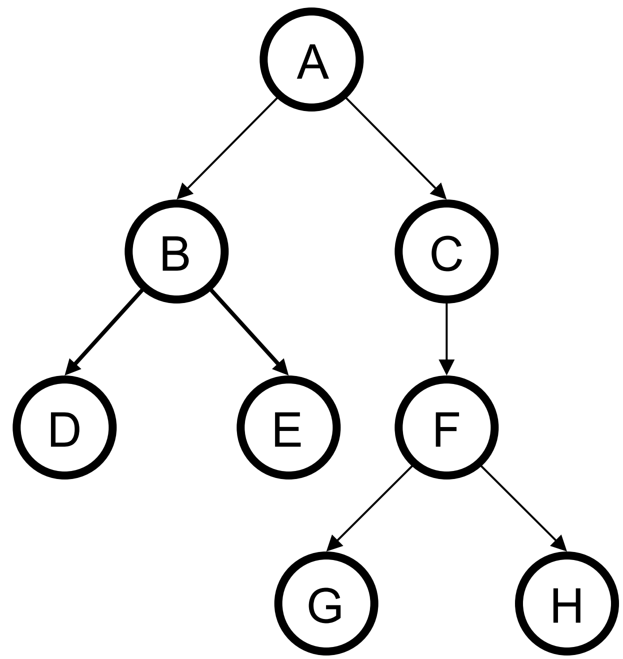
Directed graphs

- **directed graph (digraph)**: edges are one-way connections between vertices



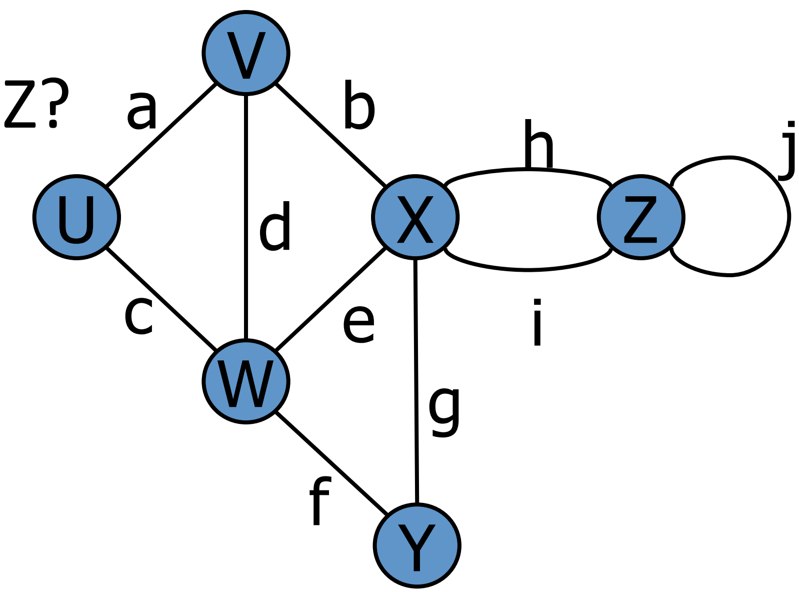
Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is directed
  - the tree is acyclic
  - there is exactly one directed path from the root to every node



More terminology

- **degree**: number of edges touching a vertex
  - example: W has degree 4
  - what is the degree of X? of Z?



- **adjacent** vertices: connected directly by an edge

  - If graph is directed, a vertex has a separate *in/out degree*

Graph questions

- Are the following graphs directed or not directed?
  - Buddy graphs of instant messaging programs?
    (vertices = users, edges = user being on another's buddy list)
  - bus line graph depicting all of Seattle's bus stations and routes
  - graph of movies in which actors have appeared together

- Are these graphs potentially cyclic? Why or why not?

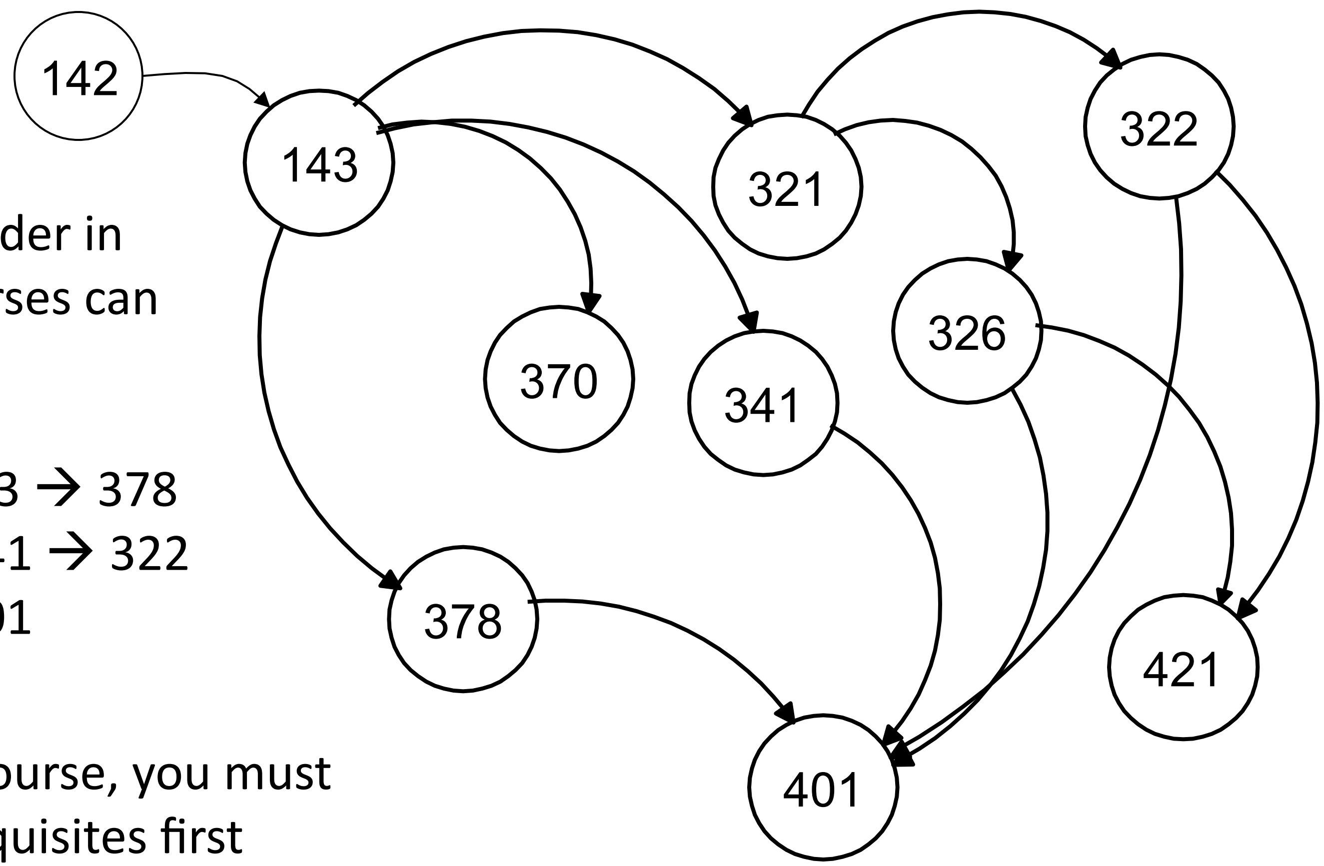
Graph exercise

- Consider a graph of instant messenger buddies.
  - What do the vertices represent? What does an edge represent?
  - Is this graph directed or undirected? Weighted or unweighted?
  - What does a vertex's degree mean? In degree? Out degree?
  - Can the graph contain loops? cycles?

- Consider this graph data:
  - Jessica's buddy list: Meghan, Alan, Martin.
  - Meghan's buddy list: Alan, Lori.
  - Toni's buddy list: Lori, Meghan.
  - Martin's buddy list: Lori, Meghan.
  - Alan's buddy list: Martin, Jessica.
  - Lori's buddy list: Meghan.

  - Compute the in/out degree of each vertex. Is the graph connected?
  - Who is the most popular? Least? Who is the most antisocial?
  - If we're having a party and want to distribute the message the most quickly, who should we tell first?

Topological Sort



Problem: Find an order in which all these courses can be taken.

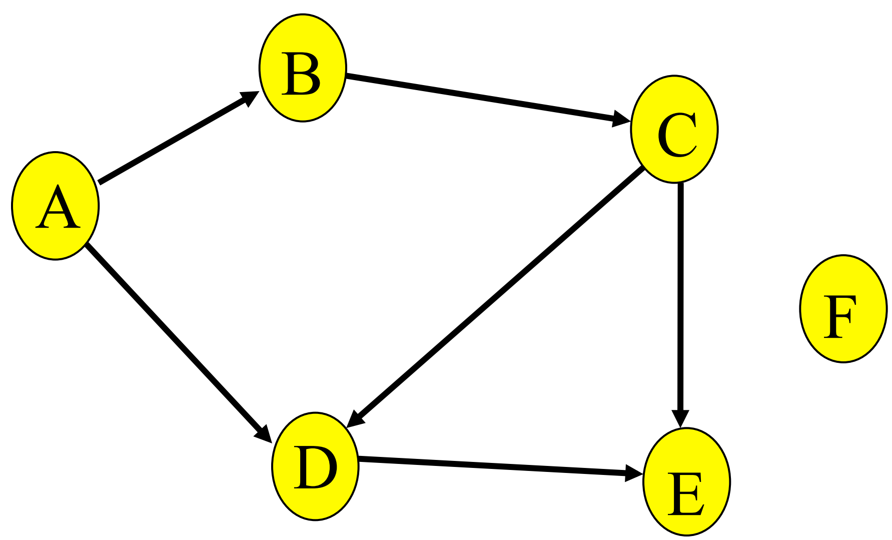
Example: 142 → 143 → 378 → 370 → 321 → 341 → 322 → 326 → 421 → 401

In order to take a course, you must take all of its prerequisites first

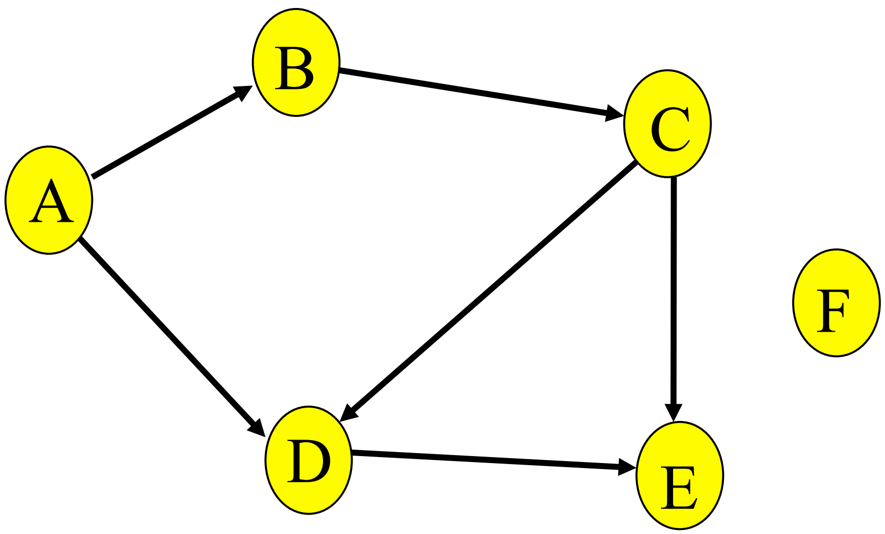
Topological Sort

Given a digraph $G = (V, E)$, find a total ordering of its vertices such that:

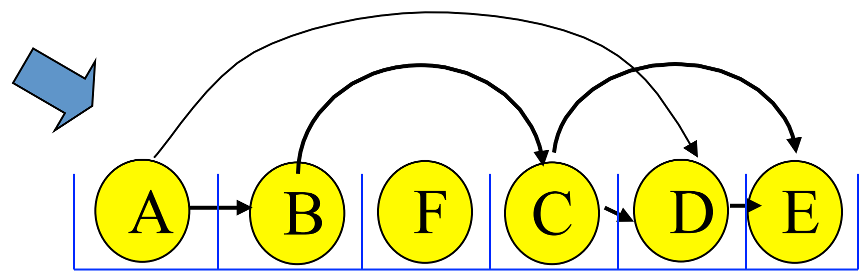
for any edge (v, w) in E, v precedes w in the ordering



Topo sort - good example

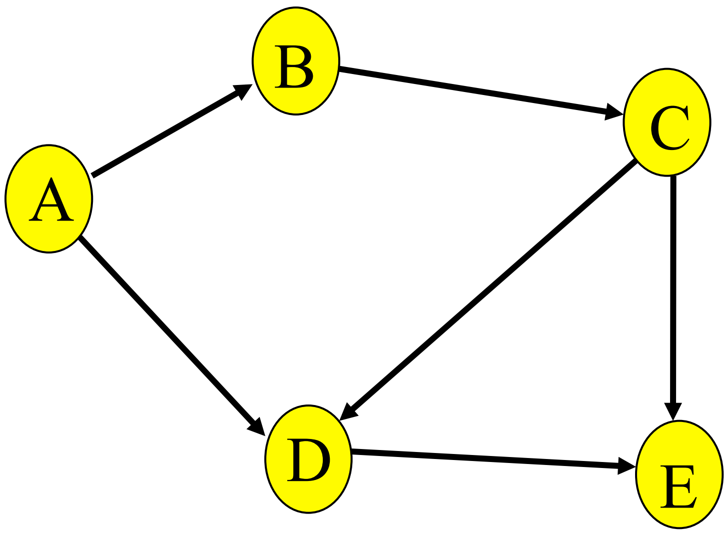


Any total ordering in which all the arrows go to the right is a valid solution

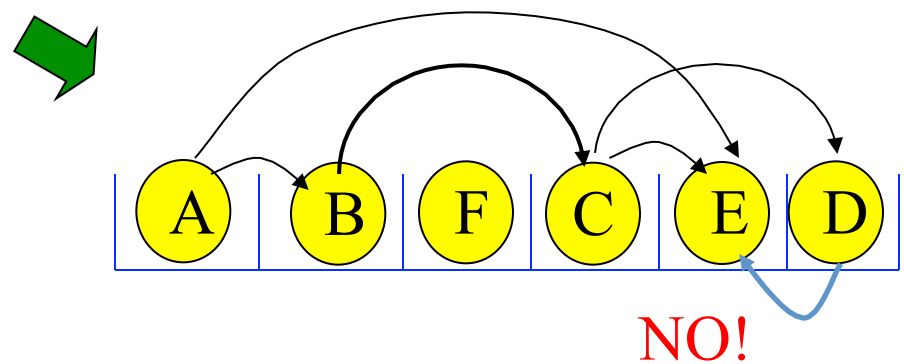


Note that F can go anywhere in this list because it is not connected.
Also the solution is not unique.

Topo sort - bad example

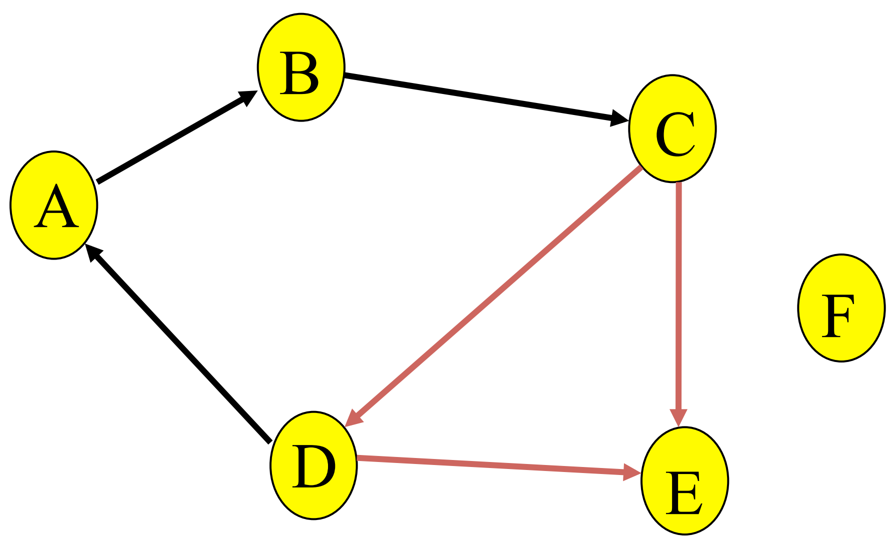


Any ordering in which an arrow goes to the left is not a valid solution



Only acyclic graphs can be topologically sorted

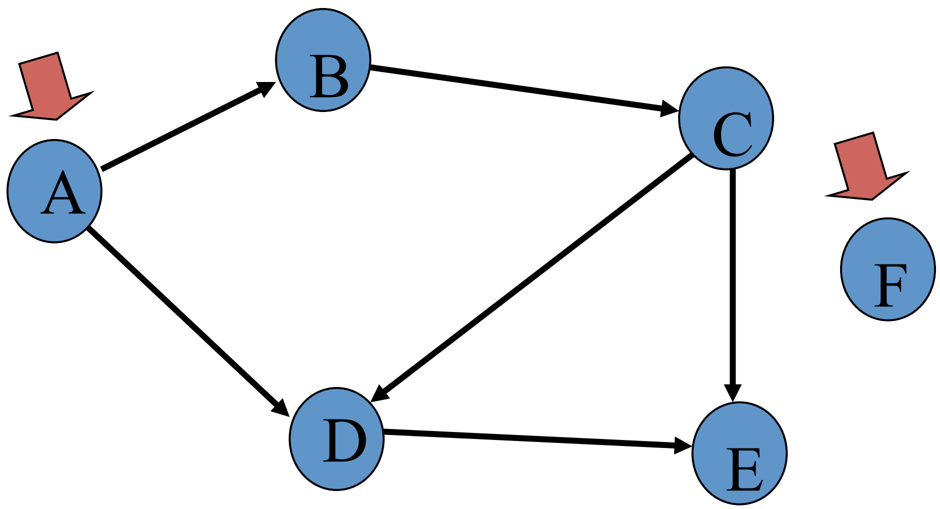
- A directed graph with a cycle cannot be topologically sorted.



Topological sort algorithm: 1

<u>Step 1</u>: Identify vertices that have no incoming edges

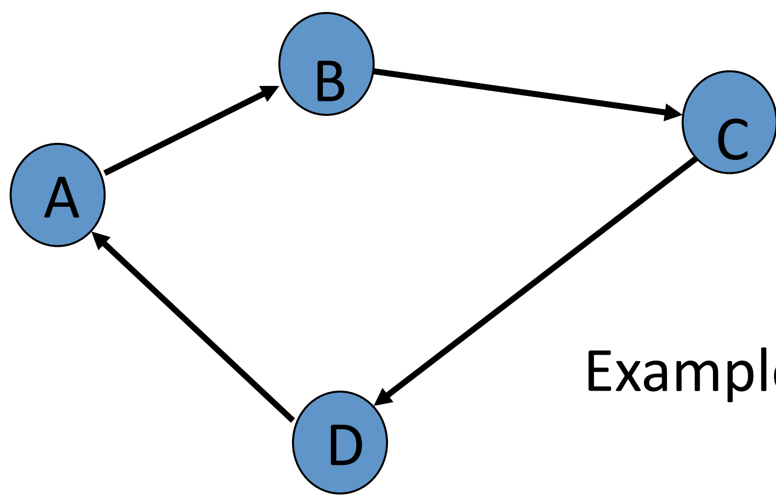
- The "in-degree" of these vertices is zero



Topo sort algorithm: 1a

Step 1: Identify vertices that have no incoming edges

- If *no such vertices*, graph has only cycle(s) (cyclic graph)
- Topological sort not possible – Halt.

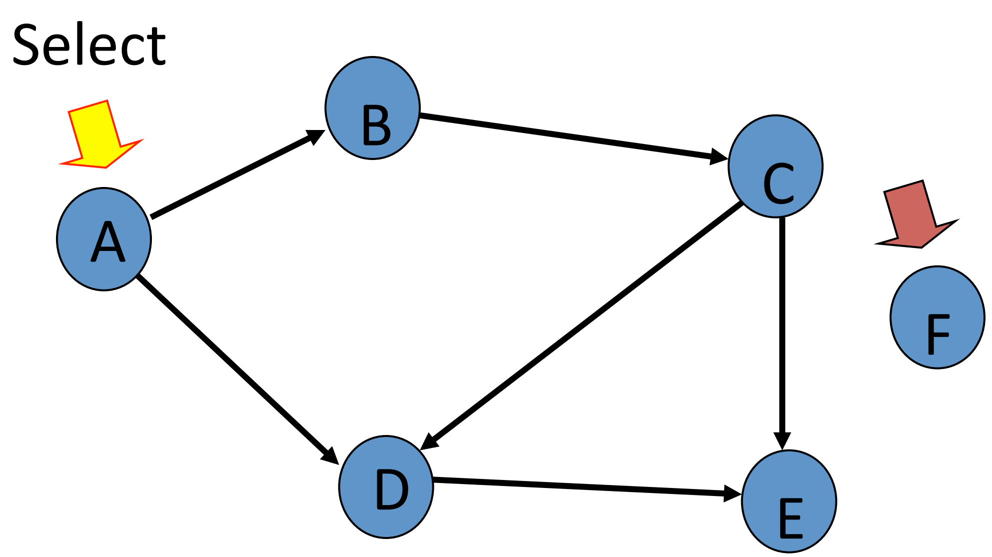


Example of a cyclic graph

Topo sort algorithm:1b

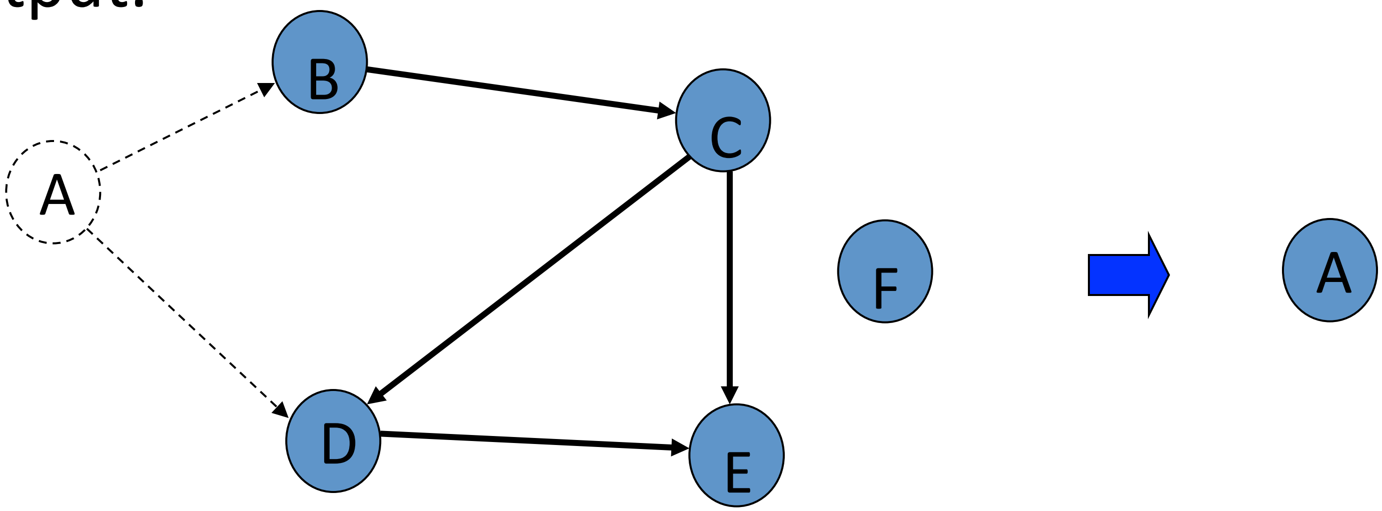
<u>Step 1</u>: Identify vertices that have no incoming edges

- Select one such vertex



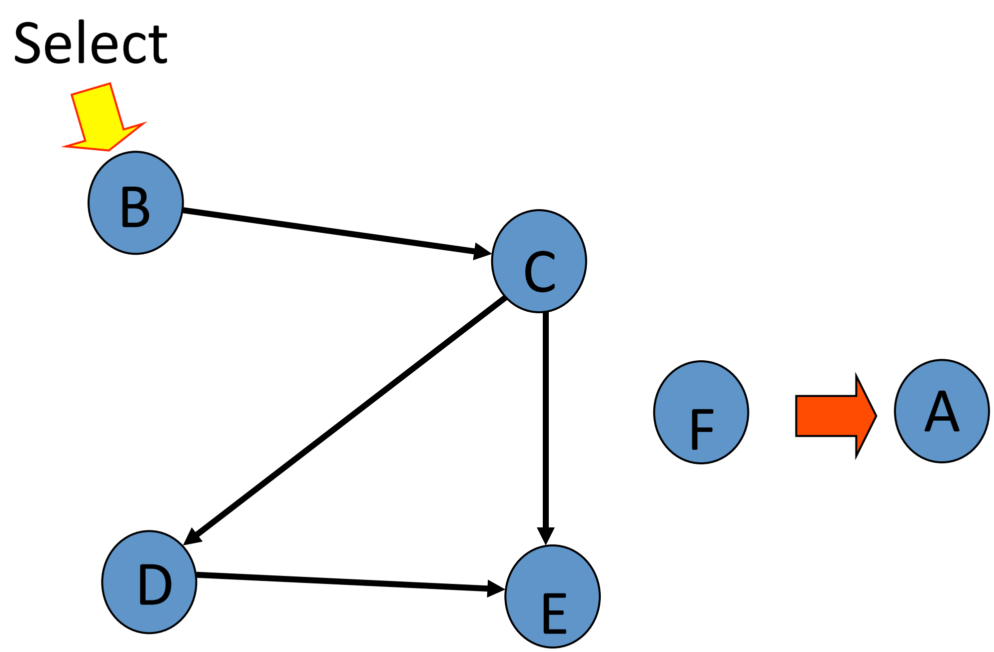
Topo sort algorithm: 2

Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



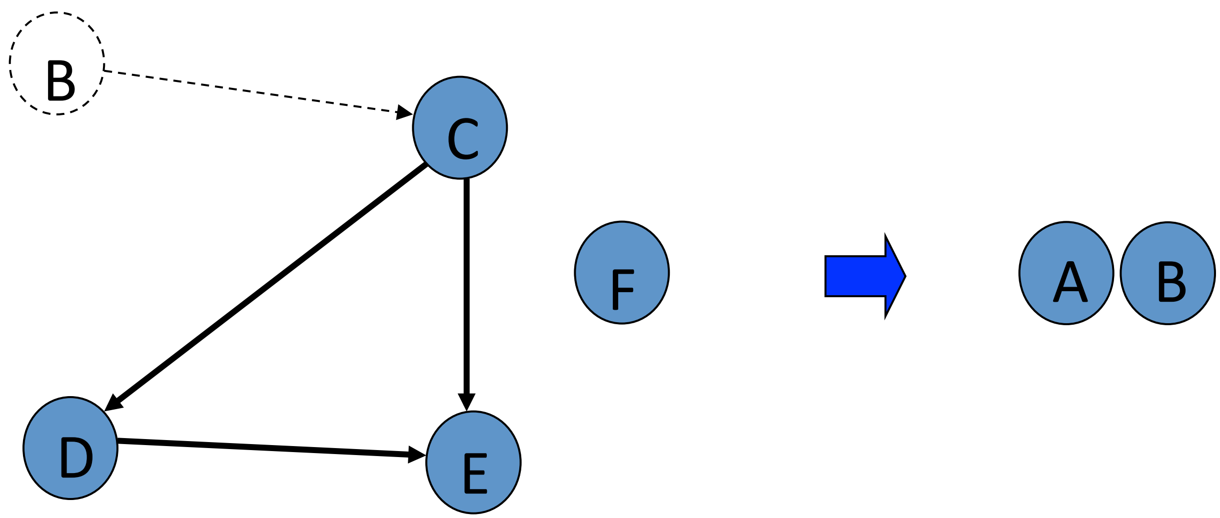
Continue until done

Repeat <u>Step 1</u> and <u>Step 2</u> until graph is empty



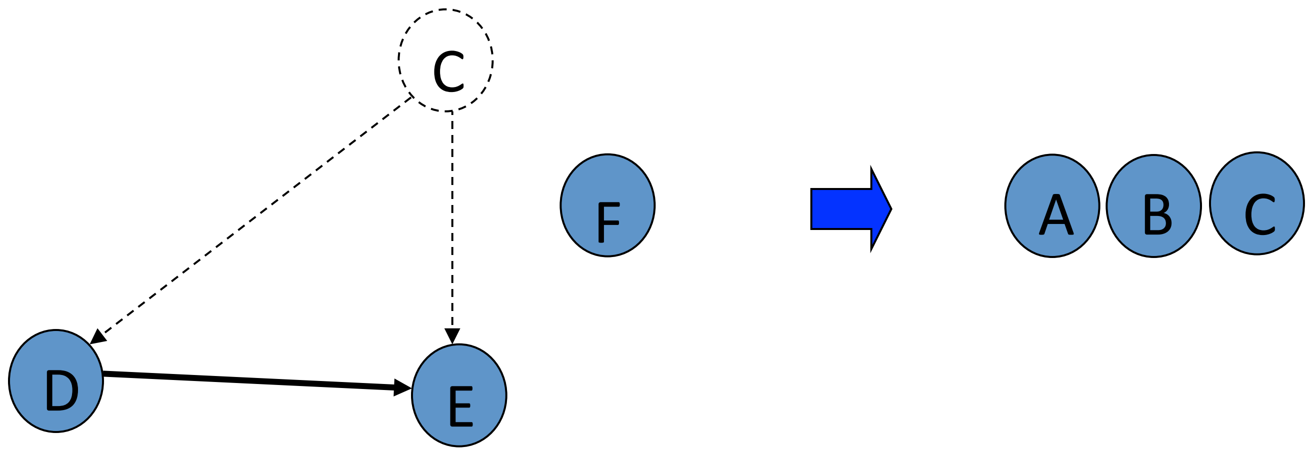
B

Select B. Copy to sorted list. Delete B and its edges.



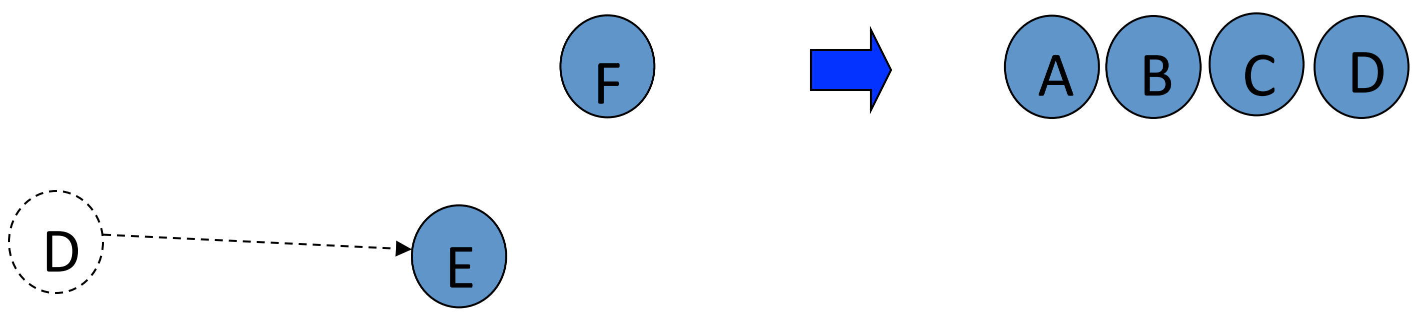
C

Select C. Copy to sorted list. Delete C and its edges.



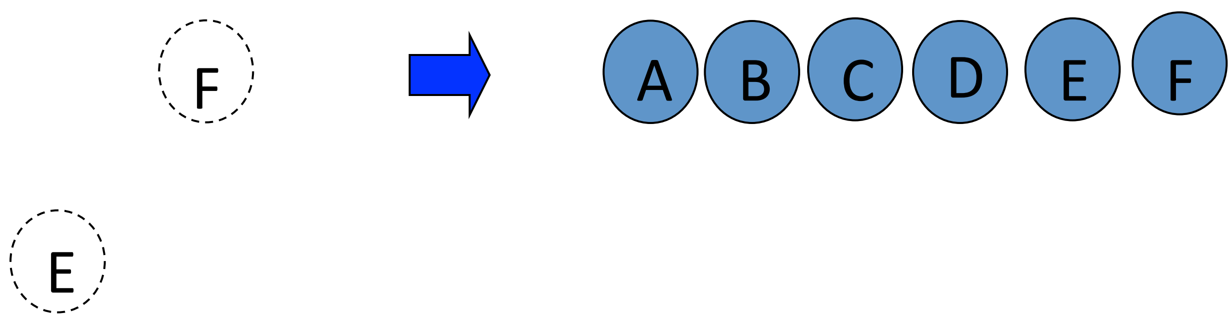
D

Select D. Copy to sorted list. Delete D and its edges.

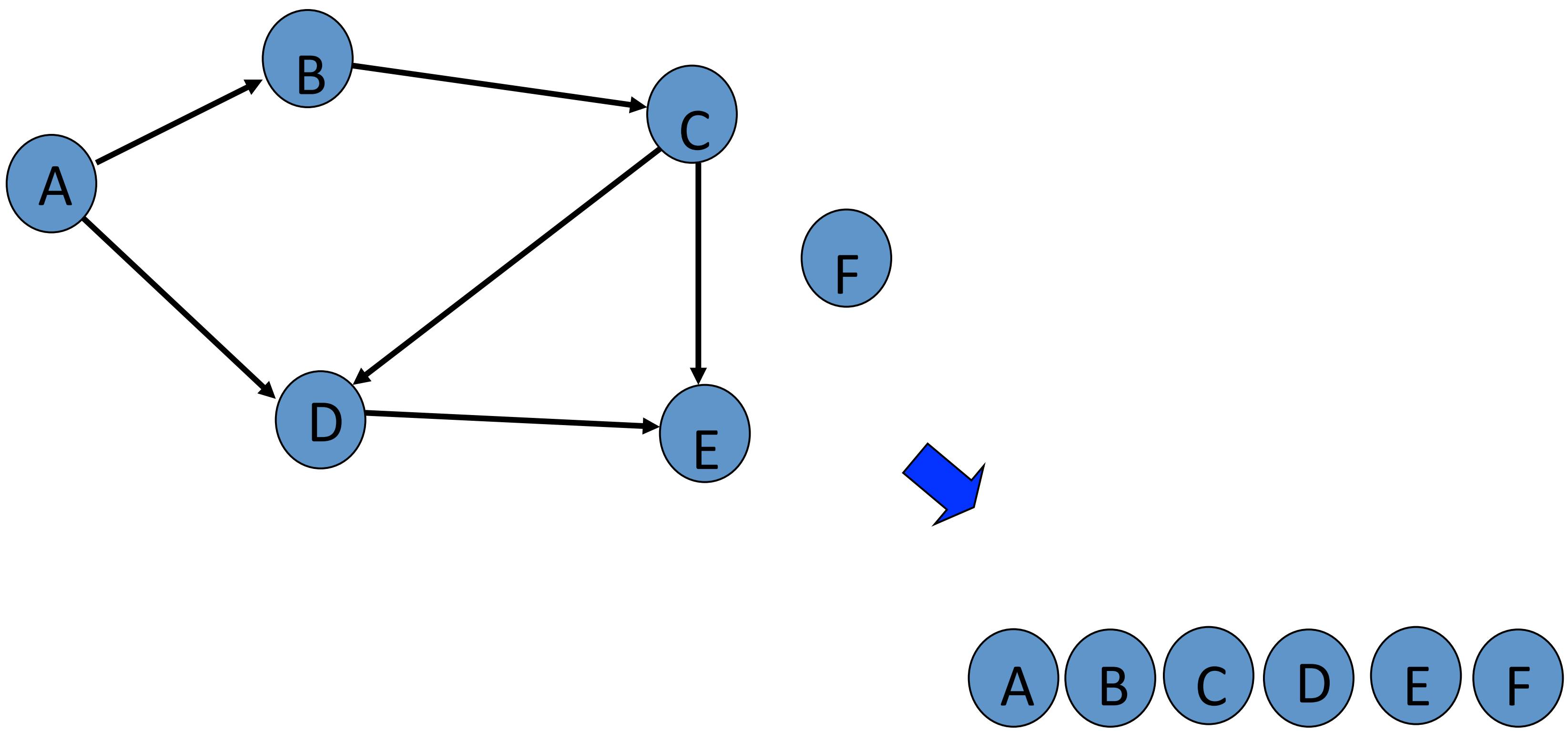


E, F

Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.



Done



Topological Sort Algorithm

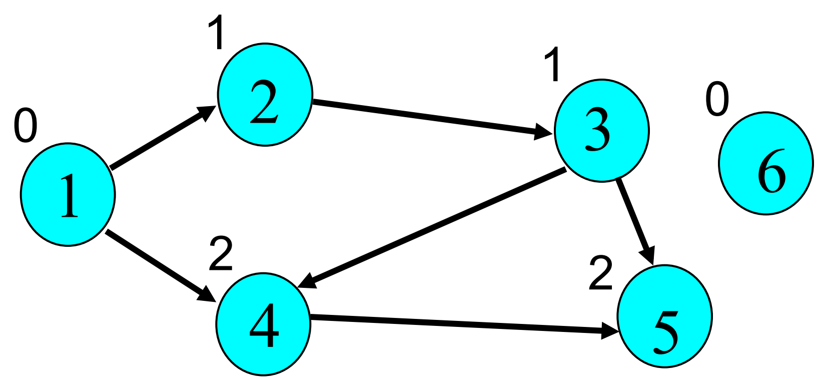
1. Store each vertex's In-Degree in an array D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
   (a) Dequeue and output a vertex
   (b) Reduce In-Degree of all vertices adjacent to it by 1
   (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.

Pseudocode

```
Queue Q := [Vertices with in-degree 0]
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x]; // y gets a linked list of vertices
  while y ≠ null do
    D[y.value] := D[y.value] - 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);
    y := y.next;
  endwhile
endwhile
```

Topo Sort w/ queue

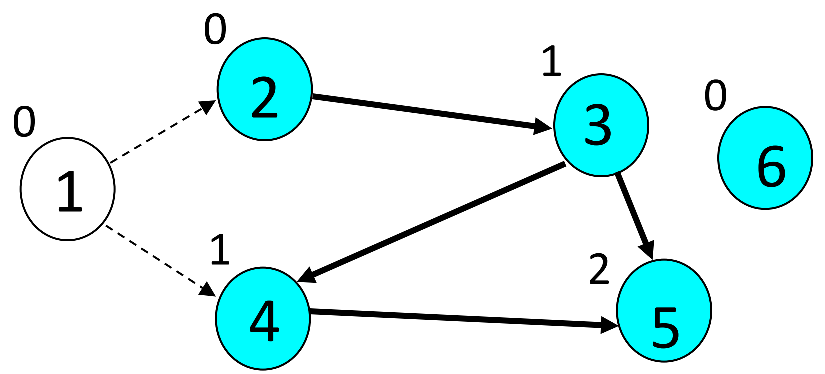
Queue (before):
Queue (after): 1, 6



Answer:

Topo Sort w/ queue

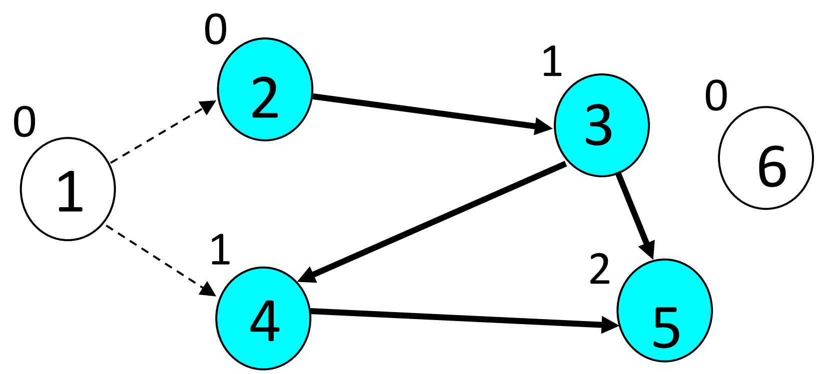
Queue (before): 1, 6
Queue (after): 6, 2



Answer: 1

Topo Sort w/ queue

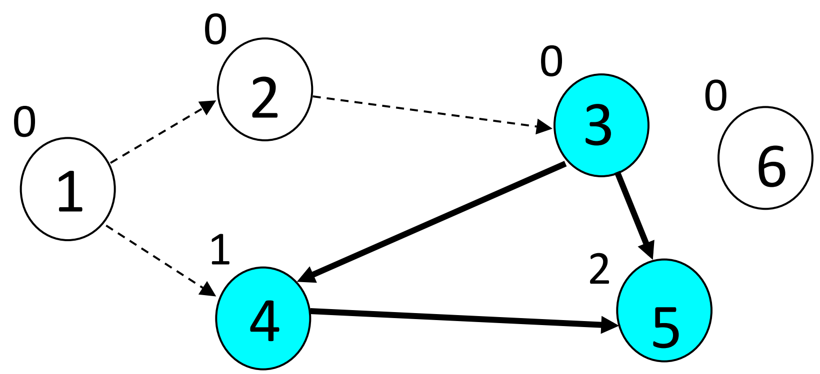
Queue (before): 6, 2
Queue (after): 2



Answer: 1, 6

Topo Sort w/ queue

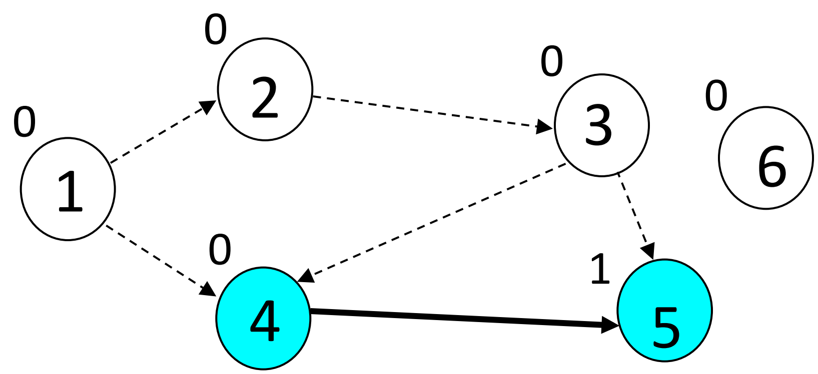
Queue (before): 2
Queue (after): 3



Answer: 1, 6, 2

Topo Sort w/ queue

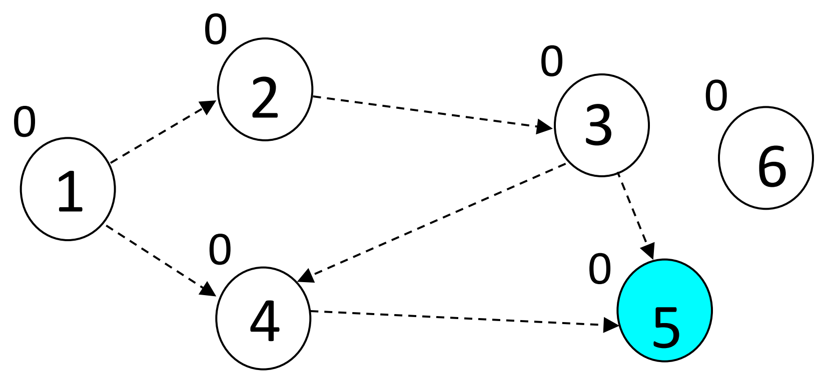
Queue (before): 3
Queue (after): 4



Answer: 1, 6, 2, 3

Topo Sort w/ queue

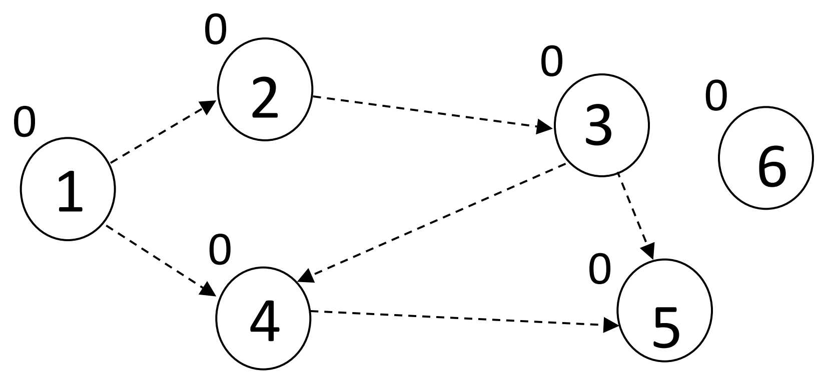
Queue (before): 4
Queue (after): 5



Answer: 1, 6, 2, 3, 4

Topo Sort w/ queue

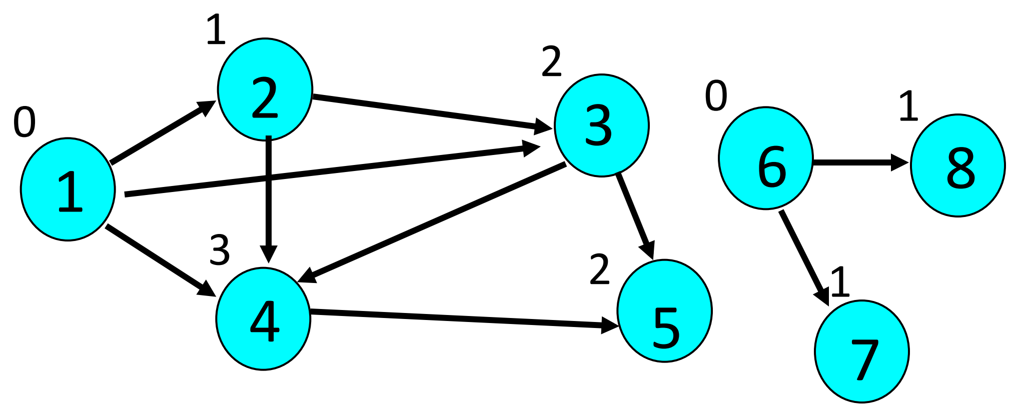
Queue (before): 5
Queue (after):



Answer: 1, 6, 2, 3, 4, 5

Topo Sort w/ stack

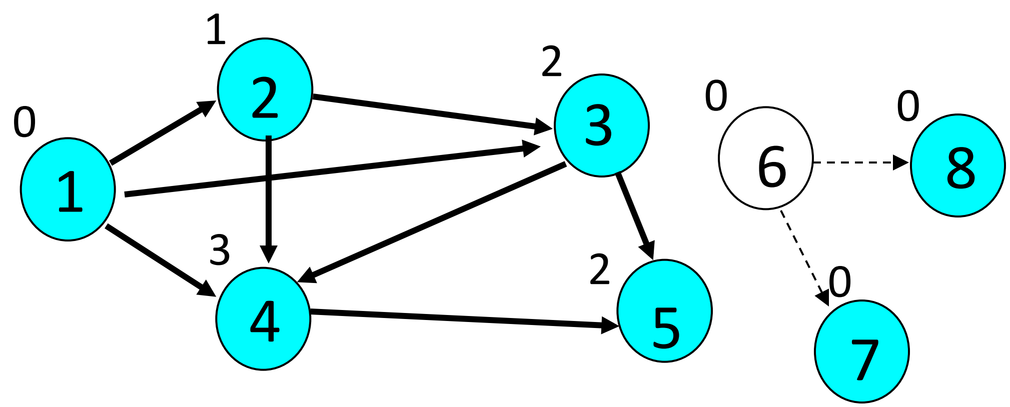
Stack (before):
Stack (after): 1, 6



Answer:

Topo Sort w/ stack

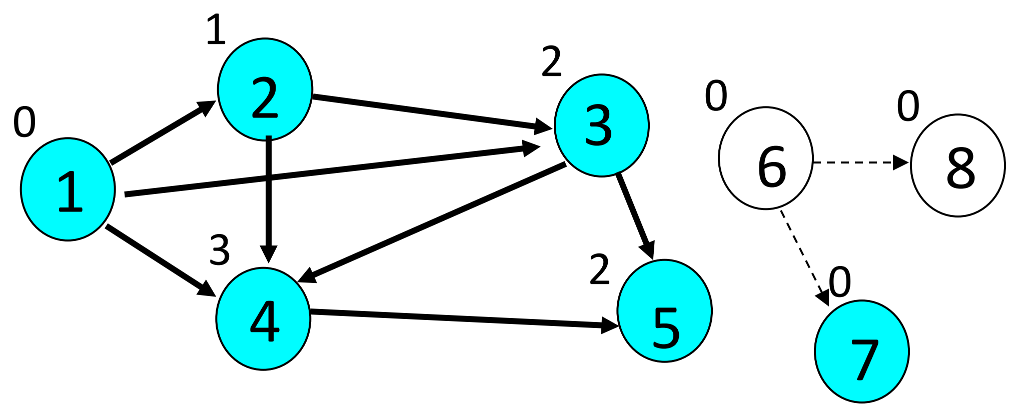
Stack (before): 1, 6
Stack (after): 1, 7, 8



Answer: 6

Topo Sort w/ stack

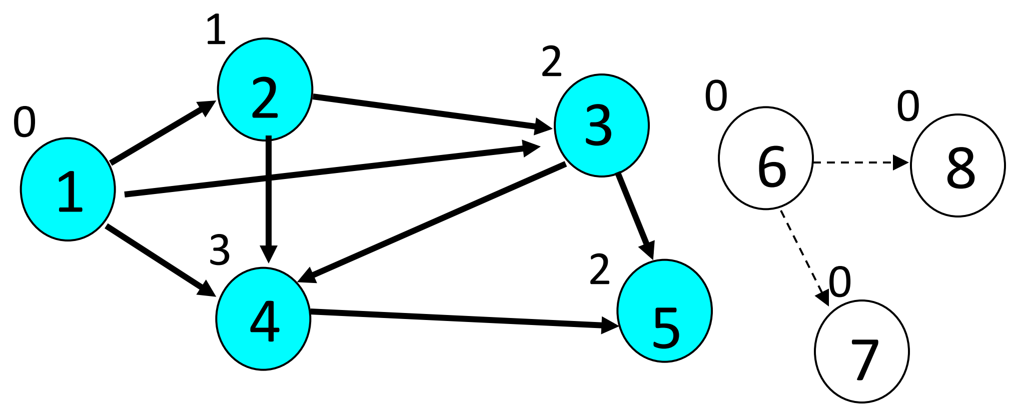
Stack (before): 1, 7, 8
Stack (after): 1, 7



Answer: 6, 8

Topo Sort w/ stack

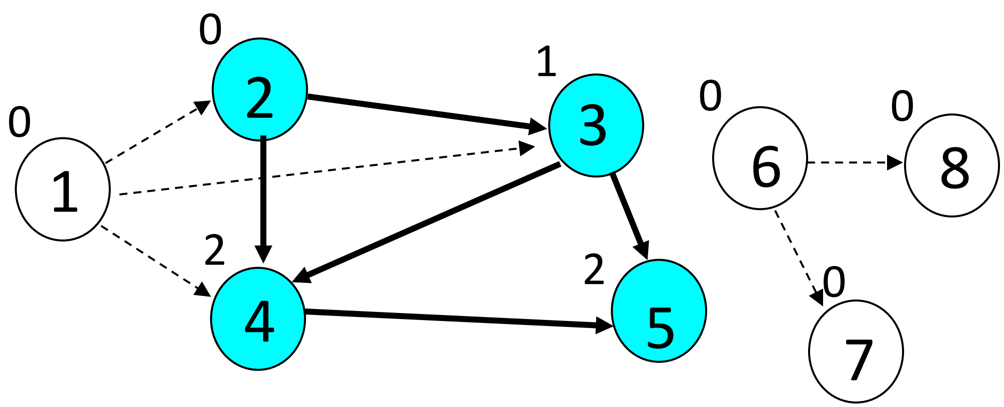
Stack (before): 1, 7
Stack (after): 1



Answer: 6, 8, 7

Topo Sort w/ stack

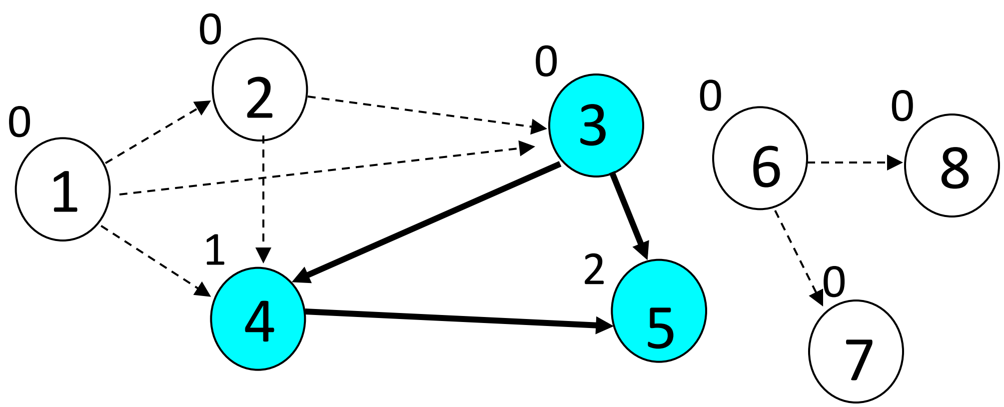
Stack (before): 1
Stack (after): 2



Answer: 6, 8, 7, 1

Topo Sort w/ stack

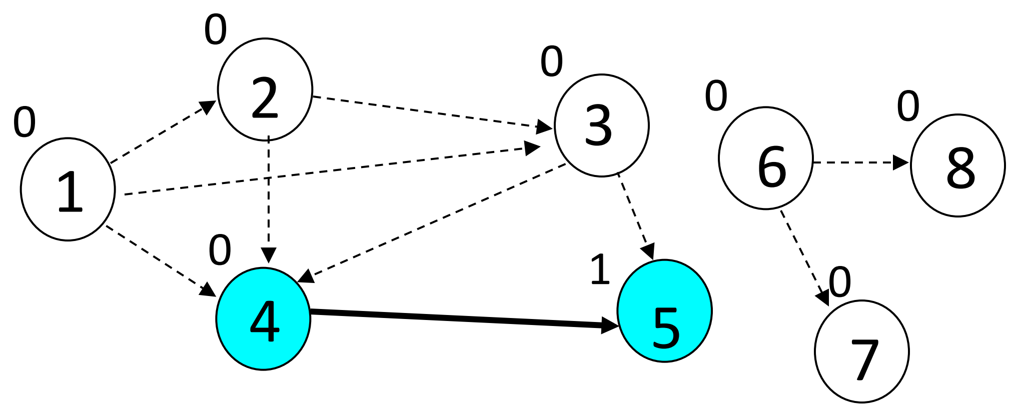
Stack (before): 2
Stack (after): 3



Answer: 6, 8, 7, 1, 2

Topo Sort w/ stack

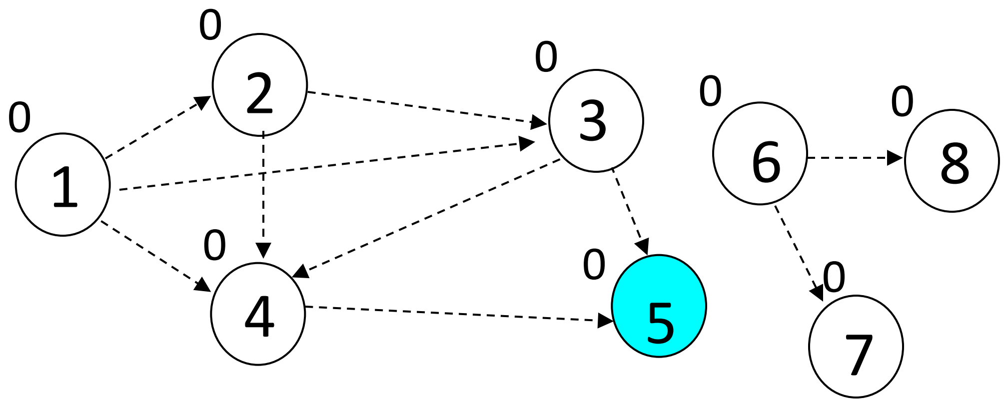
Stack (before): 3
Stack (after): 4



Answer: 6, 8, 7, 1, 2, 3

Topo Sort w/ stack

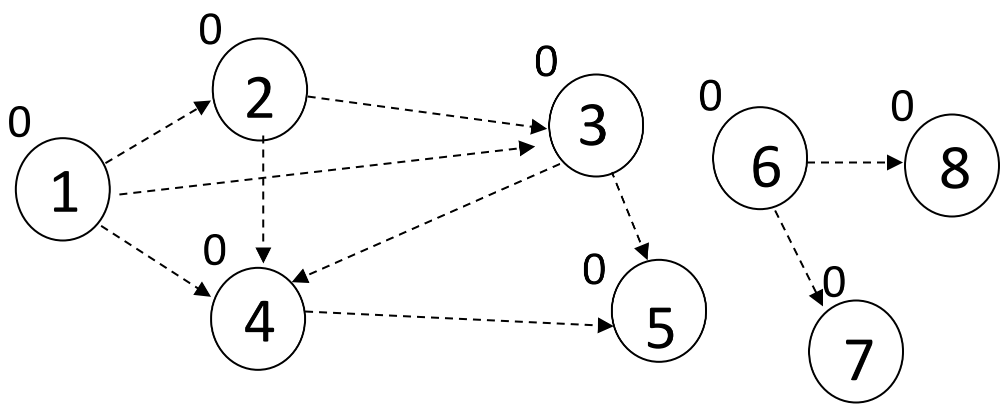
Stack (before): 4
Stack (after): 5



Answer: 6, 8, 7, 1, 2, 3, 4

Topo Sort w/ stack

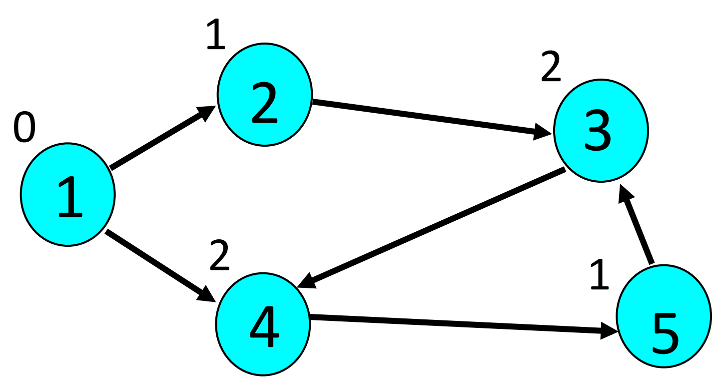
Stack (before): 5
Stack (after):



Answer: 6, 8, 7, 1, 2, 3, 4, 5

TopoSort Fails (cycle)

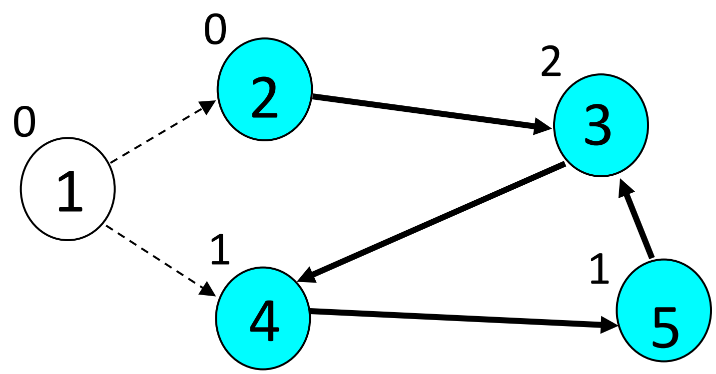
Queue (before):
Queue (after): 1



Answer:

TopoSort Fails (cycle)

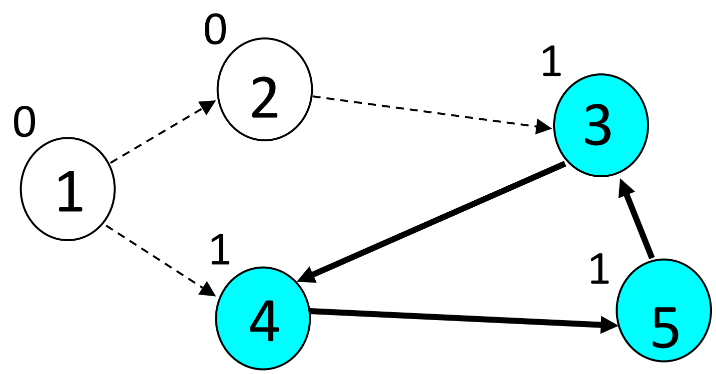
Queue (before): 1
Queue (after): 2



Answer: 1

TopoSort Fails (cycle)

Queue (before): 2
Queue (after):



Answer: 1, 2

What is the run-time???

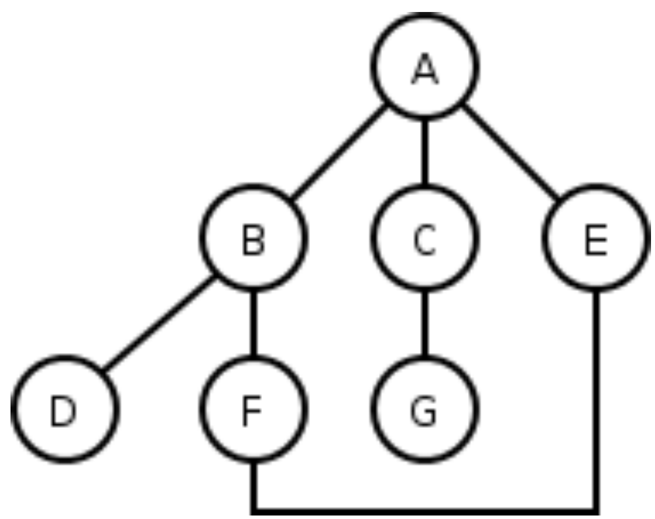
```
Initialize D   // Mapping of vertex to its in-degree
Queue Q := [Vertices with in-degree 0]
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x]; // y gets a linked list of vertices
  while y ≠ null do
    D[y.value] := D[y.value] - 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);
    y := y.next;
  endwhile
endwhile
```

Topological Sort Analysis

- Initialize In-Degree array: O(|V| + |E|)
- Initialize Queue with In-Degree 0 vertices: O(|V|)
- Dequeue and output vertex:
  – |V| vertices, each takes only O(1) to dequeue and output: O(|V|)
- Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices:
  – O(|E|)
- For input graph G=(V,E) run time = O(|V| + |E|)
  – Linear time!

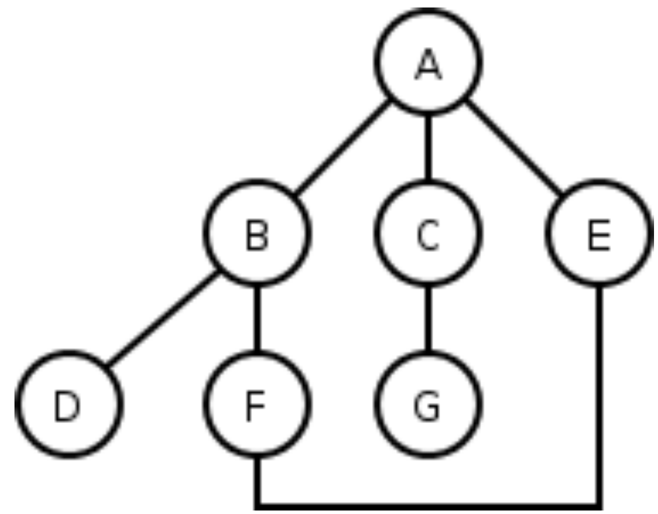
Depth-first search

- **depth-first search (DFS)**: finds a path between two vertices by exploring each possible path as many steps as possible before backtracking
  - often implemented recursively



DFS example

- All DFS paths from A to others (assumes ABC edge order)
  - A
  - A -> B
  - A -> B -> D
  - A -> B -> F
  - A -> B -> F -> E
  - A -> C
  - A -> C -> G



- What are the paths that DFS did not find?

DFS pseudocode

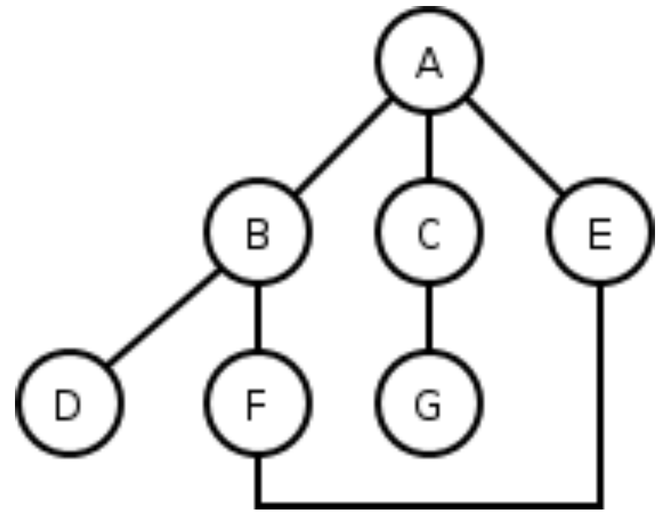
- Pseudo-code for depth-first search:

dfs(v1, v2):
    dfs(v1, v2, {})

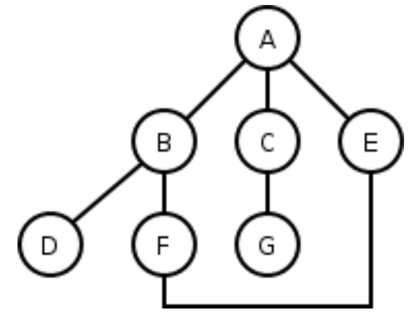
dfs(v1, v2, path):
    path += v1.
    mark v1 as visited.
    if v1 is v2:
        path is found.

    for each unvisited neighbor v_i of v1
    where there is an edge from v1 to v_i:
        if dfs(v_i, v2, path) finds a path, path is found.

    path -= v1.   path is not found.



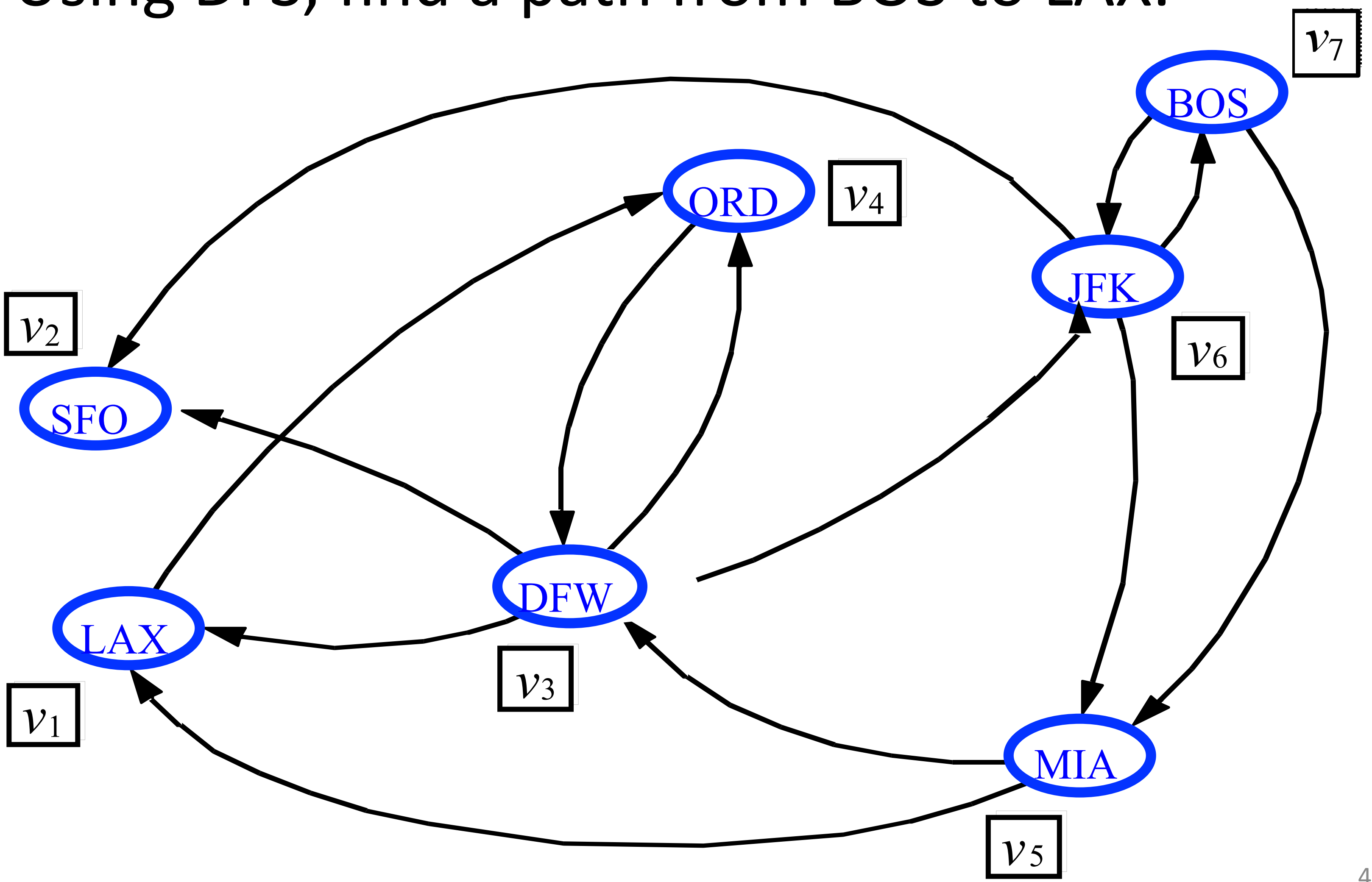
DFS observations



- guaranteed to find a path if one exists
- easy to retrieve exactly what the path is (to remember the sequence of edges taken) if we find it
- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: DFS(A, E) may return A -> B -> F -> E

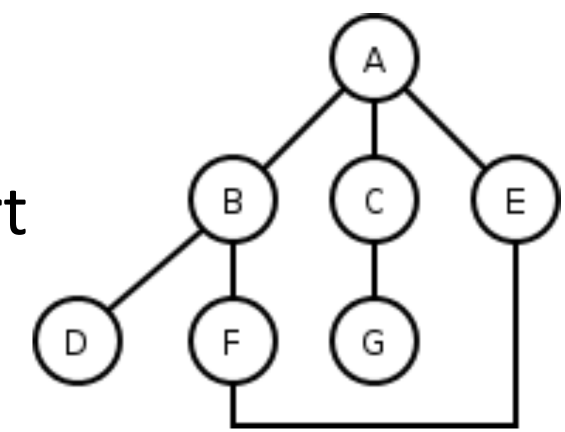
Another DFS example

- Using DFS, find a path from BOS to LAX.



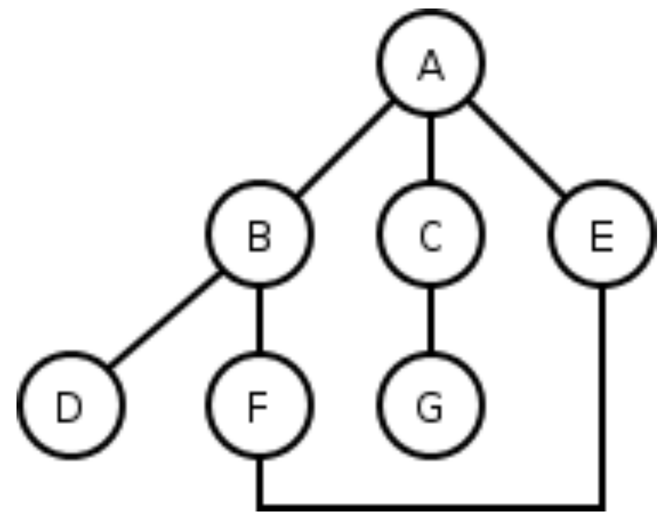
Breadth-first search

- **breadth-first search (BFS)**: finds a path between two nodes by taking one step down all paths and then immediately backtracking
  - often implemented by maintaining a list or queue of vertices to visit
  - BFS always returns the path with the fewest edges between the start and the goal vertices



BFS example

- All BFS paths from A to others (assumes ABC edge order)
  - A
  - A -> B
  - A -> C
  - A -> E
  - A -> B -> D
  - A -> B -> F
  - A -> C -> G



- What are the paths that BFS did not find?

BFS pseudocode

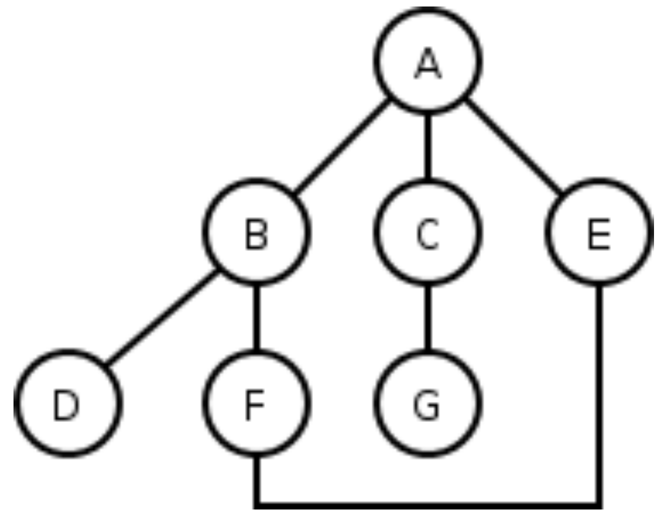
- Pseudo-code for breadth-first search:

```
bfs(v1, v2):
  List := {v1}.
  mark v1 as visited.

  while List not empty:
    v := List.removeFirst().
    if v is v2:
      path is found.

    for each unvisited neighbor vi of v
    where there is an edge from v to vi:
      List.addLast(vi).

  path is not found.
```

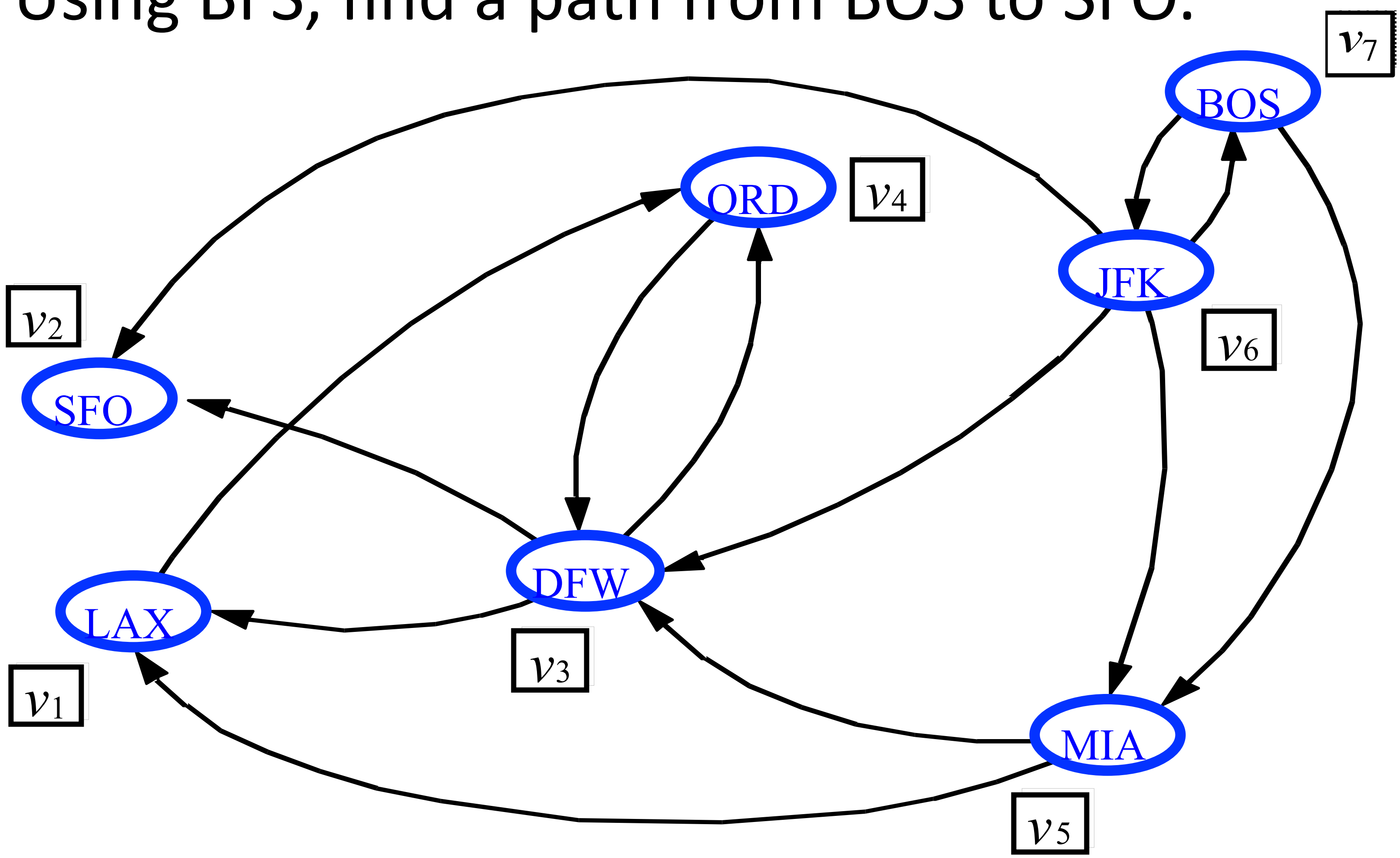


BFS observations

- *optimality*:
  - in unweighted graphs, optimal. (fewest edges = best)
  - In weighted graphs, not optimal.
    (path with fewest edges might not have the lowest weight)
- *disadvantage*: harder to reconstruct what the actual path is once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a Path array/list in progress
- observation: any particular vertex is only part of one partial path at a time
  - We can keep track of the path by storing *predecessors* for each vertex (references to the previous vertex in that path)

Another BFS example

- Using BFS, find a path from BOS to SFO.



DFS, BFS runtime

- What is the expected runtime of DFS, in terms of the number of vertices V and the number of edges E ?
- What is the expected runtime of BFS, in terms of the number of vertices V and the number of edges E ?
- Answer: $O(|V| + |E|)$
  - each algorithm must potentially visit every node and/or examine every edge once.
  - why not $O(|V| * |E|)$?
- What is the space complexity of each algorithm?