CSE 332: Data Structures & Parallelism
Lecture 18: Graphs Intro

Arthur Liu

Summer 2022

Outline for Today

- Graph terminology
- Adjacency Matrix, Adjacency List

ADTs so far

- We’ve seen:
- Queues and Stacks
  - Our data points have some order we’re maintaining
- Priority Queues
  - Our data had some priority we needed to keep track of.
- Dictionaries
  - Our data points came as (key, value) pairs.

Graphs

Secure | https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk | Read Edit View history | Search Wikipedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

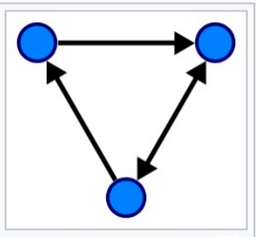
Graph (abstract data type)

From Wikipedia, the free encyclopedia

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).



A graph with three vertices and three edges.

Contents [hide]
1 Operations

Graphs are too versatile to think of them as only an ADT!

Graphs

Represent data points and the relationships between them.

That's vague.

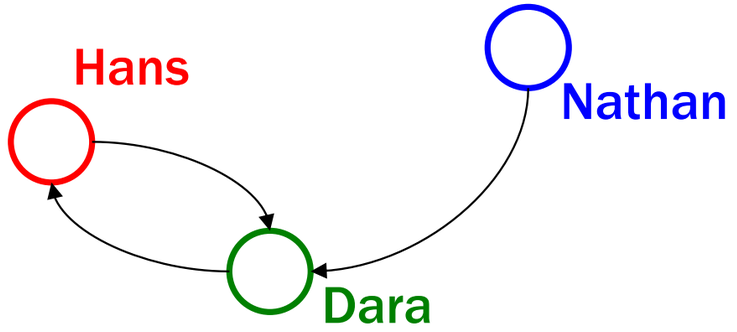
Formally:

A graph is a pair: G = (V,E)

V: set of **vertices** (aka **nodes**)

E: set of **edges**

Each edge is a pair of vertices.




```
V = {Hans,Dara,Nathan}
E = {(Nathan,Dara),
     (Hans,Dara),
     (Dara,Hans)}
```

Making Graphs

If your problem has **data** and **relationships**, you might want to represent it as a graph

How do you choose a representation?

Usually:

Think about what your “fundamental” objects are

- Those become your vertices.

Then think about how they’re related

- Those become your edges.

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Facebook friendships

Input data for the “6 degrees of Kevin Bacon” game

Course Prerequisites

Some examples

For each of the following think about what you should choose for vertices and edges.

The internet.

Vertices: webpages. Edges from a to b if a has a hyperlink to b.

Facebook friendships

Vertices: people. Edges: if two people are friends

Input data for the “6 Degrees of Kevin Bacon” game

Vertices: actors. Edges: if two people appeared in the same movie

Or: Vertices for actors and movies, edge from actors to movies they appeared in.

Course Prerequisites

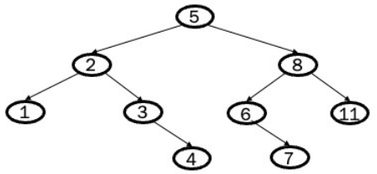
Vertices: courses. Edge: from a to b if a is a prereq for b.

More Graphs

- We've already used graphs to represent things in this course:

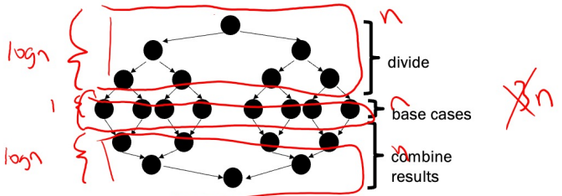
A LOT

EX3: An AVL tree?



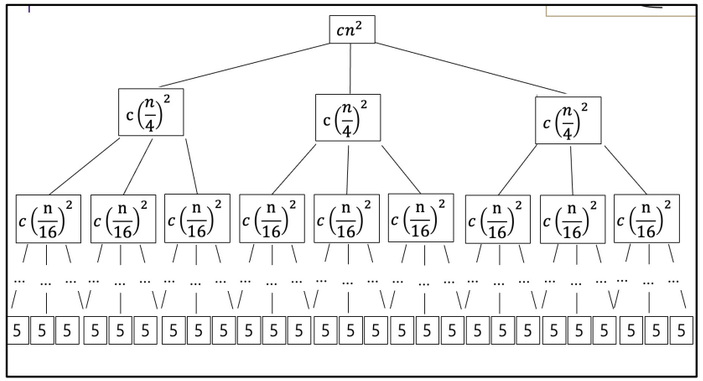
Our simple examples, in more detail

Our **`fork`** and **`join`** often look like this:



In this context, the span (T_∞) is:

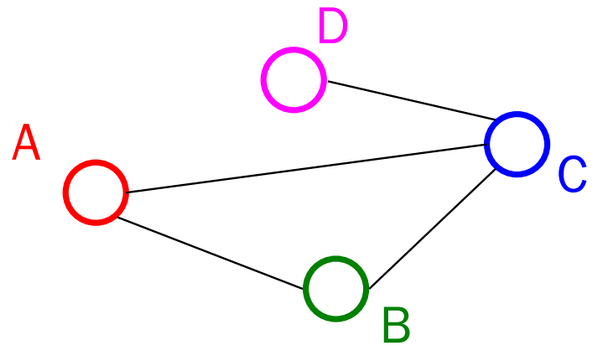
- The longest dependence-chain; longest 'branch' in parallel 'tree'
- Example: $O(\log n)$ for summing an array; we halve the data down to our cut-off, then add back together; $O(\log n)$ steps, $O(1)$ time for each
- Also called "critical path length" or "computational depth"



Graph Terms

Undirected Graphs

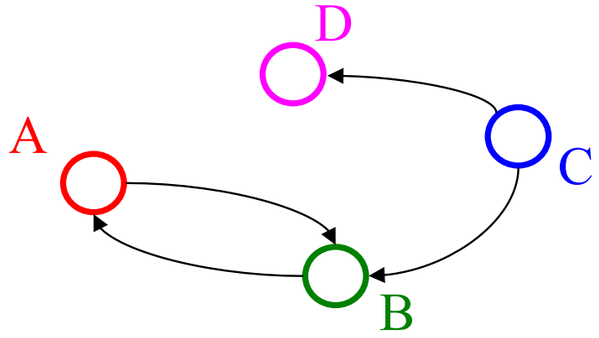
- In undirected graphs, edges have no specific direction
  - Edges are always “two-way”



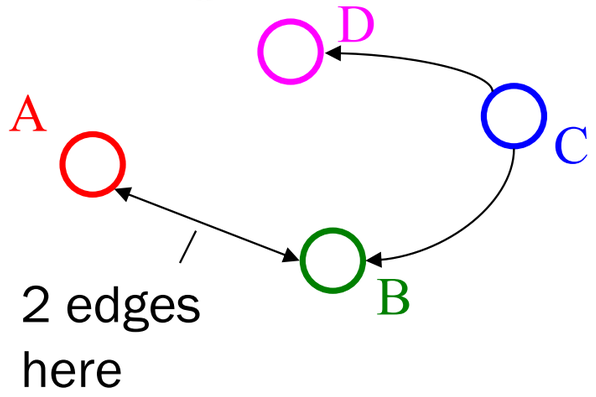
- **Thus, `(u,v) ∈ E` implies `(v,u) ∈ E`.**
  - **Only one of these edges needs to be in the set; the other is implicit**
- **Degree of a vertex: number of edges containing that vertex**
  - **Put another way: the number of adjacent vertices**

Directed Graphs

- In directed graphs (sometimes called digraphs), edges have a direction



or



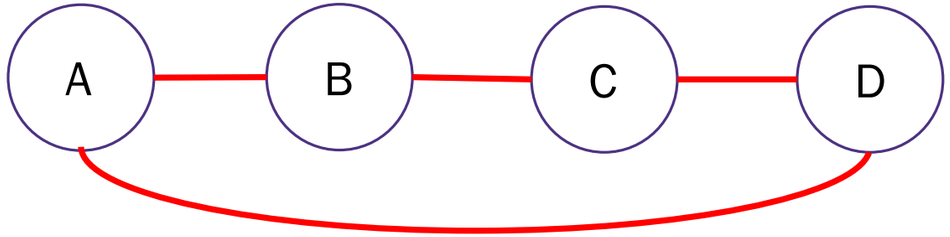
- **Thus, `(u,v) ∈ E` does *not* imply `(v,u) ∈ E`.**
  - **Let `(u,v) ∈ E` mean u → v**
  - **Call `u` the source and `v` the destination**
- **In-Degree of a vertex: number of in-bound edges,**

  i.e., edges where the vertex is the destination
- **Out-Degree of a vertex: number of out-bound edges**

  i.e., edges where the vertex is the source

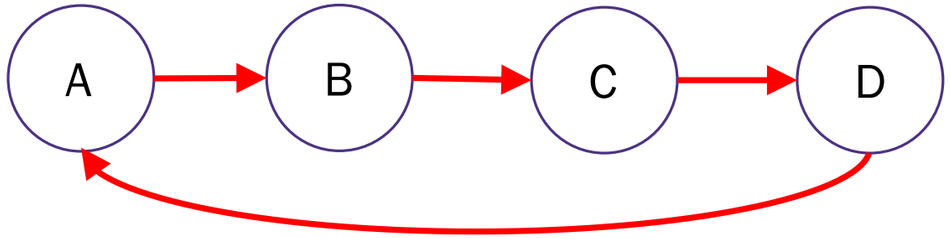
Graph Terms

Walk – A sequence of **adjacent** vertices. Each connected to next by an edge.



A,B,C,D is a walk.
So is A,B,A

(Directed) Walk–must follow the direction of the edges



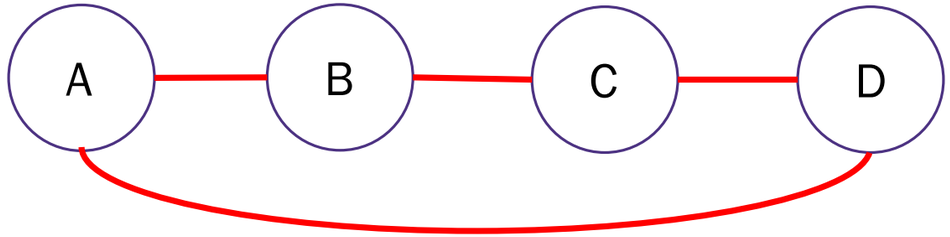
A,B,C,D,A is a directed walk.
A,B,A is not.

Length – The number of edges in a walk

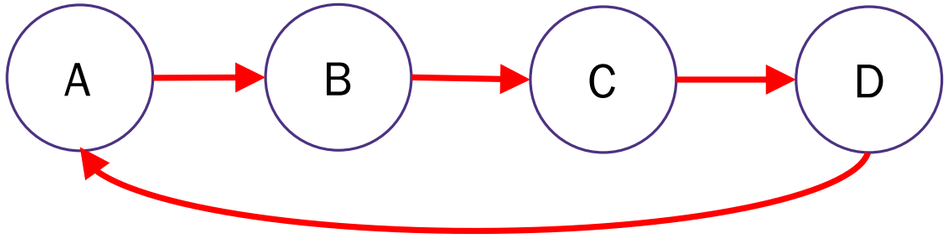
- (A,B,C,D) has length 3.

Graph Terms

Path – A walk that doesn't repeat a vertex. A,B,C,D is a path. A,B,A is not.



Cycle – path with an extra edge from last vertex back to first.

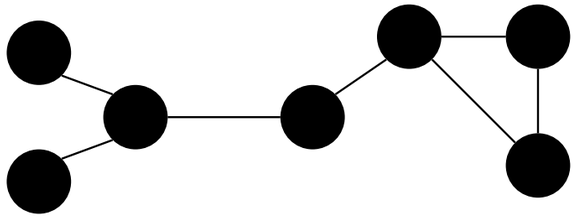


Be careful looking at other sources. Some people call our "walks" "paths" and our "paths" "simple paths"

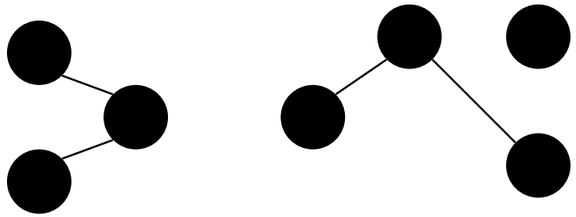
Use the definitions on these slides.

Undirected graph connectivity

- An undirected graph is connected if for all pairs of vertices **`u,v`**, there exists a *path* from **`u`** to **`v`**



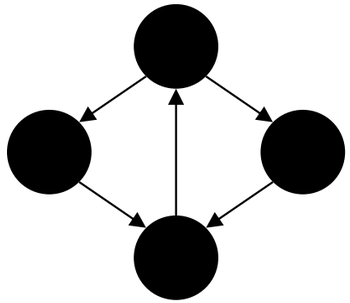
Connected graph



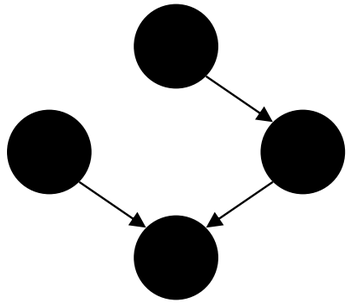
Disconnected graph

Directed graph connectivity

- A directed graph is strongly connected if there is a path from every vertex to every other vertex

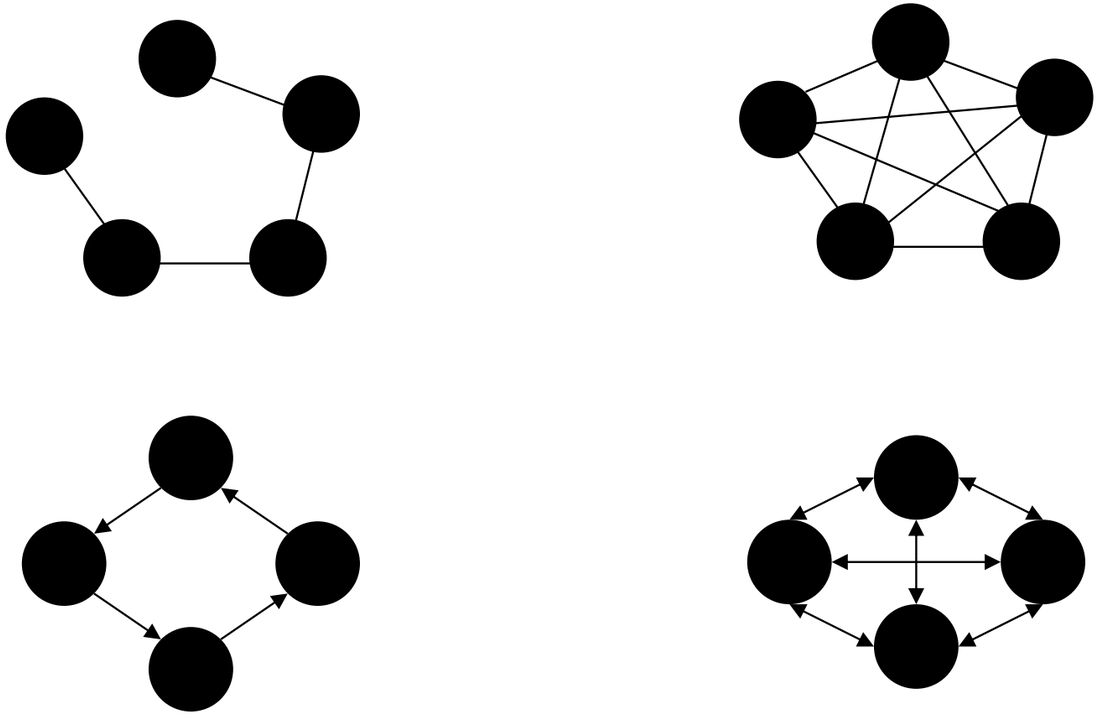


- A directed graph is weakly connected if there is a path from every vertex to every other vertex *ignoring direction of edges*

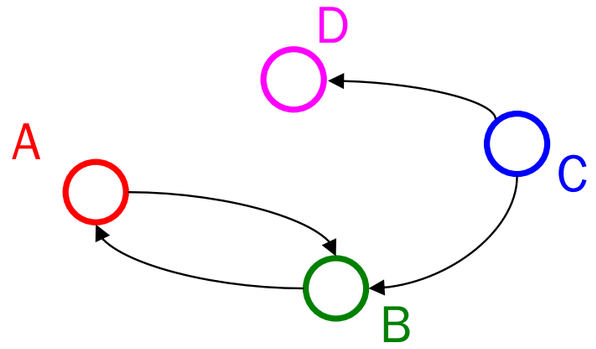


Complete Graph

A complete graph is a graph in which there is an ***edge*** between every pair of vertices.



Some math with edges



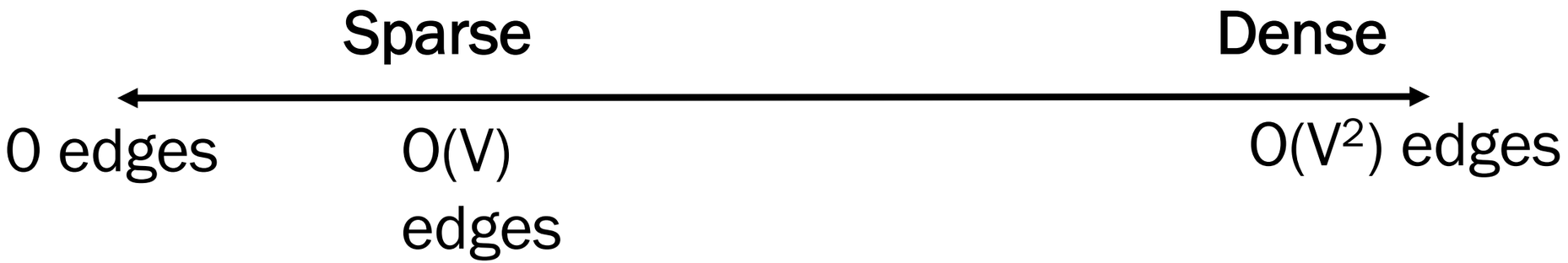

```
V = {A, B, C, D}
E = {(C, B),
     (A, B),
     (B, A)
     (C, D)}
```

For a graph **G = (V,E)**:

- **|V|** = N, is the number of vertices
- **|E|** = M, is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?

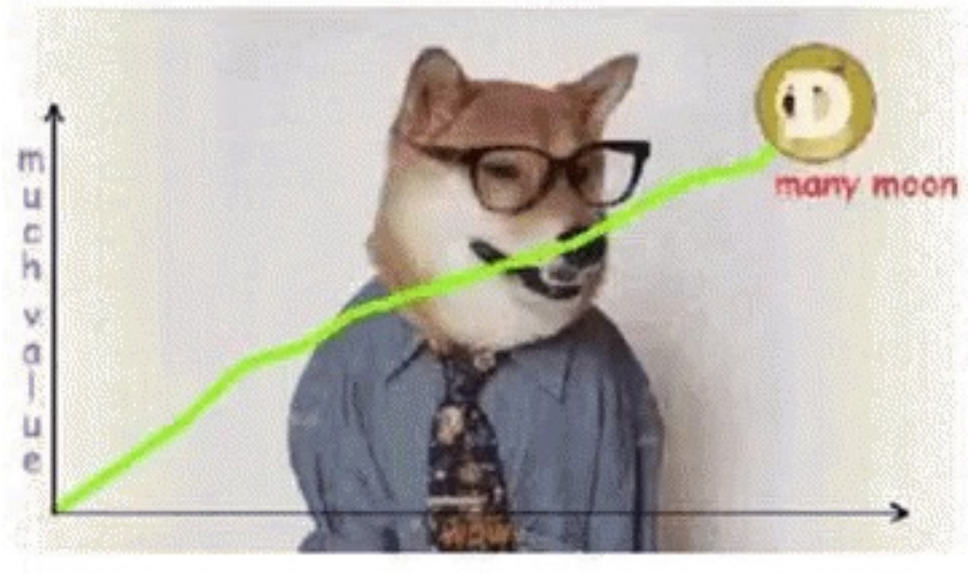
- If **(u,v) ∈ E**
  - Then **v** is a neighbor of **u**, i.e., **v** is adjacent to **u**
  - Order matters for directed edges
    - **u** is not adjacent to **v** unless **(v,u) ∈ E**

Density / Sparsity



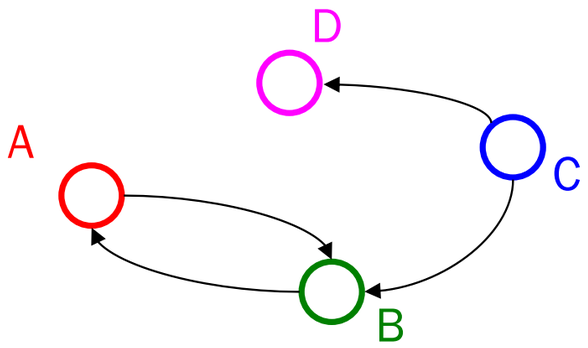
Undirected
connected graph
must have at least
V – 1 edges

Representing and Using Graphs



Adjacency Matrix

- Assign each node a number from **`0`** to **`|V|-1`**
- A |V| x |V| matrix (i.e., 2-D array) of Booleans (or 1 vs. 0)
  - If **`M`** is the matrix, then **`M[u][v] == true`** means there is an edge from **`u`** to **`v`**

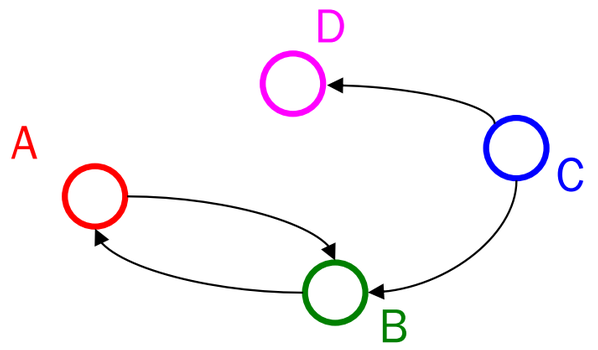


	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?

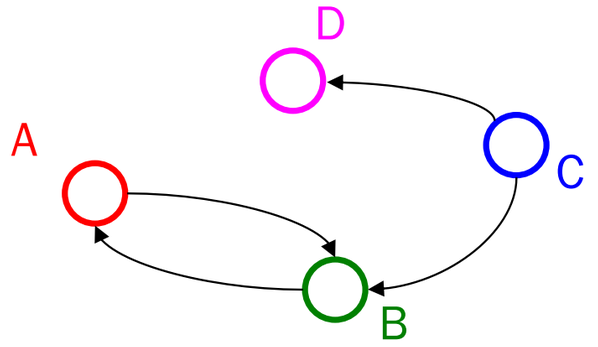
	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F



Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges: $O(|V|)$
  - Get a vertex's in-edges: $O(|V|)$
  - Decide if some edge exists: $O(1)$
  - Insert an edge: $O(1)$
  - Delete an edge: $O(1)$
- Space requirements:
  - $|V|^2$ bits
- Best for sparse or dense graphs?
  - Best for dense graphs

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F



Adjacency Matrix Properties

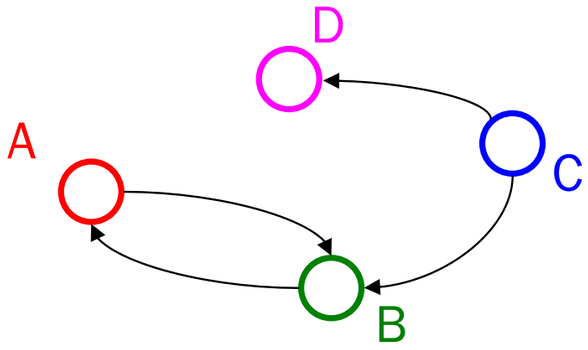
- How will the adjacency matrix vary for an *undirected graph*?
  - Undirected will be symmetric about diagonal axis

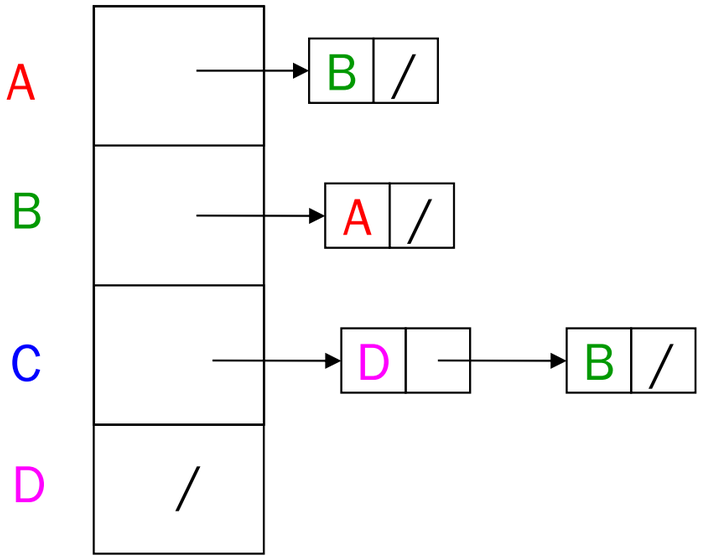
- How can we adapt the representation for *weighted graphs*?
  - Instead of a Boolean, store a number in each cell
  - Need some value to represent ‘not an edge’
    - In *some* situations, 0 or -1 works

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency List

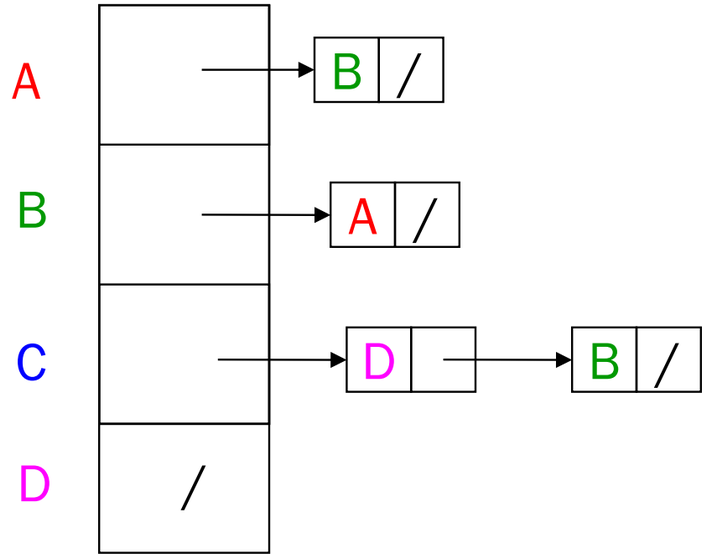
- Assign each node a number from **0** to **|V|−1**
- An array of length **|V|** in which each entry stores a list of all adjacent vertices (e.g., linked list)

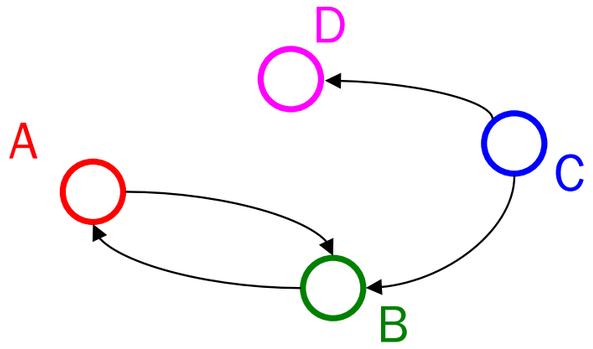




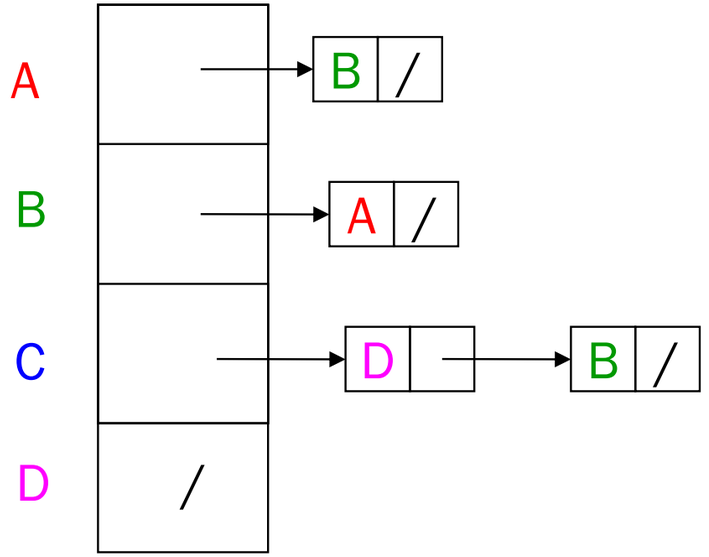
Adjacency List Properties

- Running time to:
  - Get all of a vertex's out-edges:
  - Get all of a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for dense or sparse graphs?

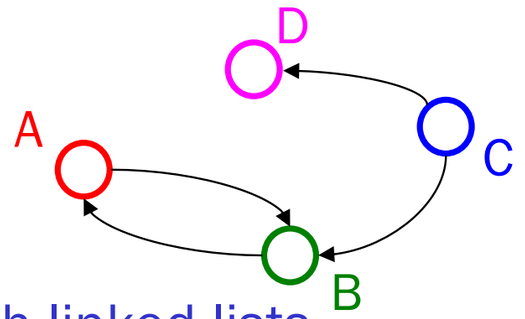




Adjacency List Properties



- Running time to:
  - Get all of a vertex's out-edges:
    $O(d)$ where d is out-degree of vertex
  - Get all of a vertex's in-edges:
    $O(|V| + |E|)$ (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    $O(d)$ where d is out-degree of source
  - Insert an edge: $O(1)$ (unless you need to check if it's there)
  - Delete an edge: $O(d)$ where d is out-degree of source
- Space requirements:
  - $O(|V|+|E|)$
- Best for dense or sparse graphs?
  - Best for sparse graphs, so can sometimes just stick with linked lists



Which is better?

Graphs are often sparse:

- Streets form grids
  - every corner is not connected to every other corner
- Airlines rarely fly to all possible cities
  - or if they do it is to/from a hub rather than directly to/from all small cities to other small cities

Adjacency lists should generally be your default choice

Some other special graphs we've seen

Trees as graphs

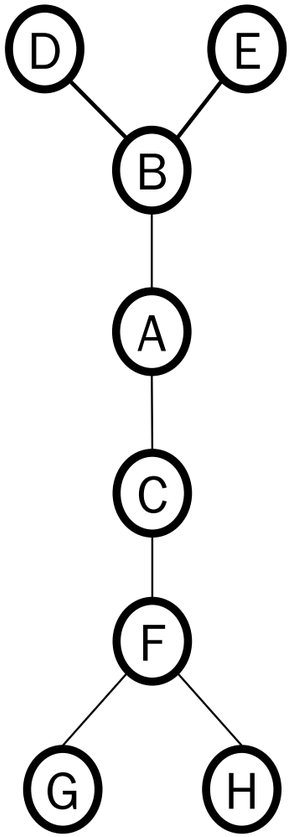
In graph theory,

we say a tree is a graph that is:

- undirected
- acyclic
- connected

How does this relate to the trees we know and love?...

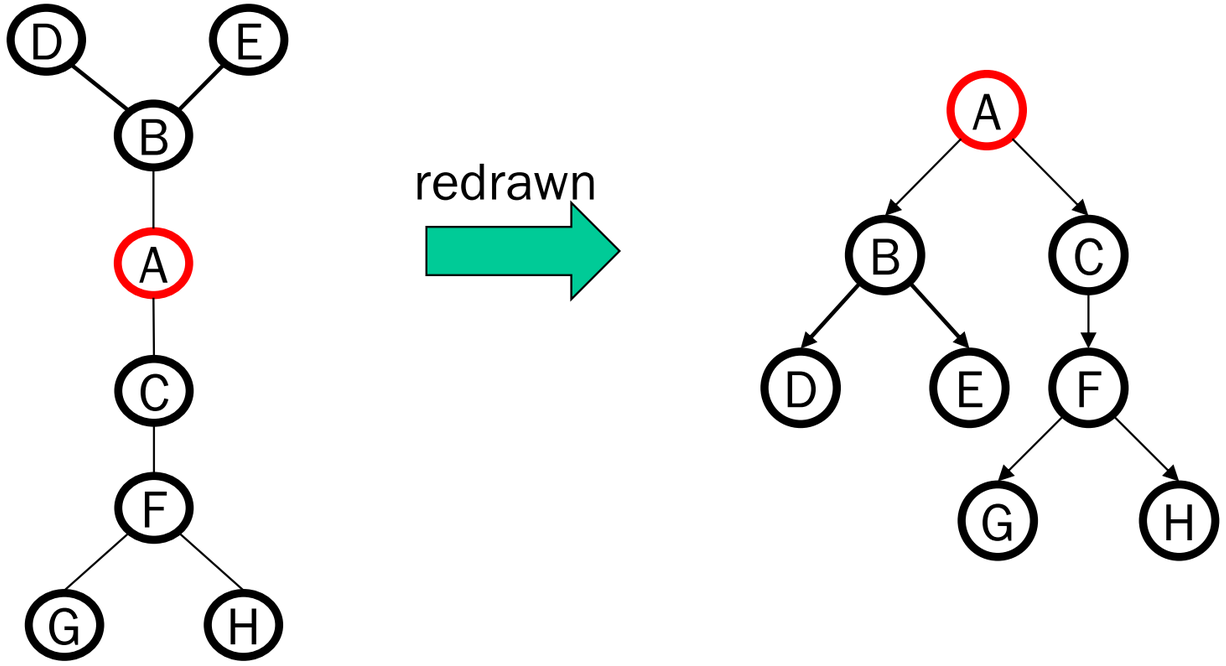
Example:



Rooted Trees

- We are more accustomed to rooted trees where:
  - We identify a unique (“special”) root
  - We think of edges as **directed**: parent to children

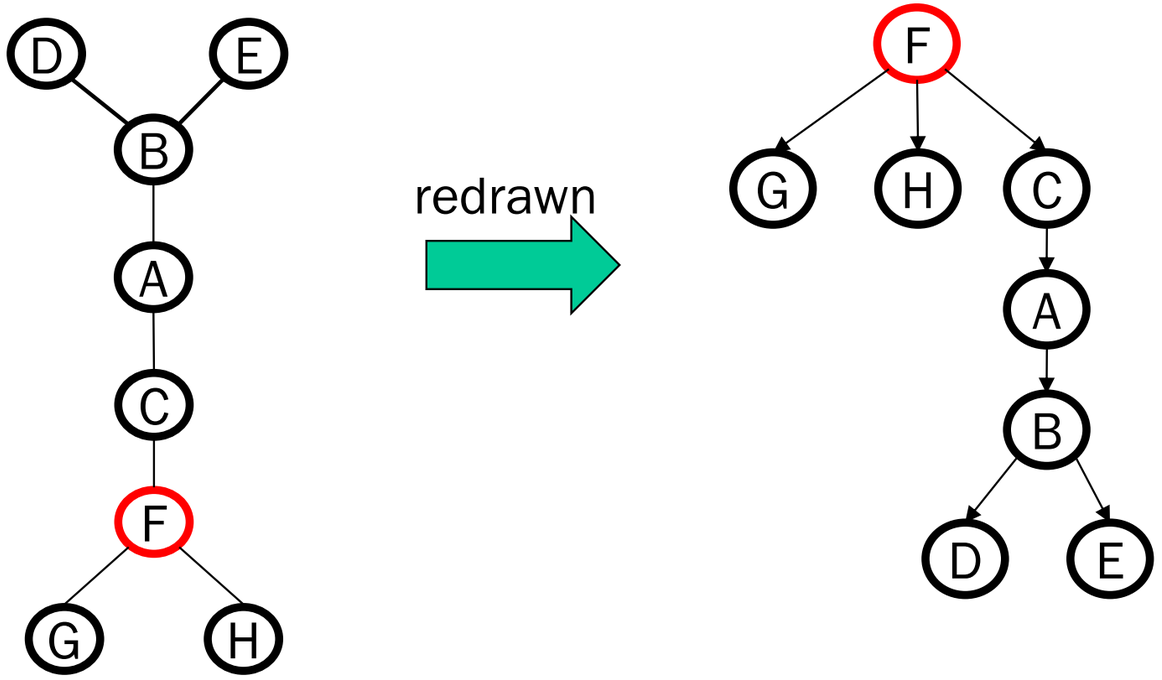
- Given a tree, once you pick a root, you have a unique rooted tree (just drawn differently and with undirected edges)



Rooted Trees (Another example)

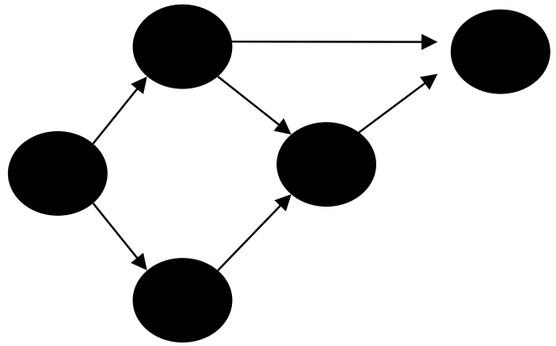
- We are more accustomed to rooted trees where:
  - We identify a unique (“special”) root
  - We think of edges as **directed**: parent to children

- Given a tree, once you pick a root, you have a unique rooted tree (just drawn differently and with undirected edges)

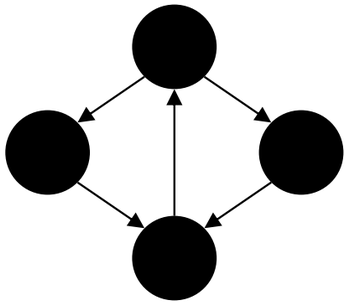


Directed acyclic graphs (DAGs)

- A DAG is a directed graph with no (directed) cycles
  - Every rooted directed tree is a DAG
    - But not every DAG is a rooted directed tree:



  - Every DAG is a directed graph
    - But not every directed graph is a DAG:



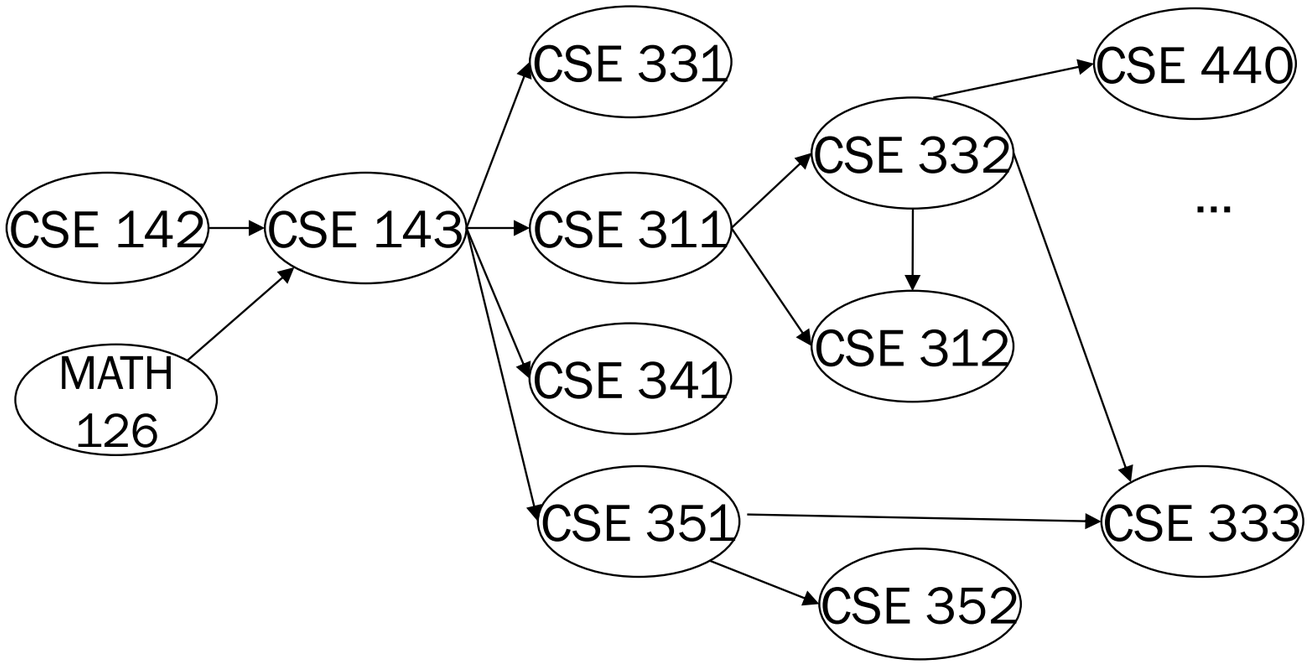
First graph algorithm!

Topological Sort

Disclaimer: Do not use for official advising purposes!
(Implies that CSE 332 is a pre-req for CSE 312 – not true)

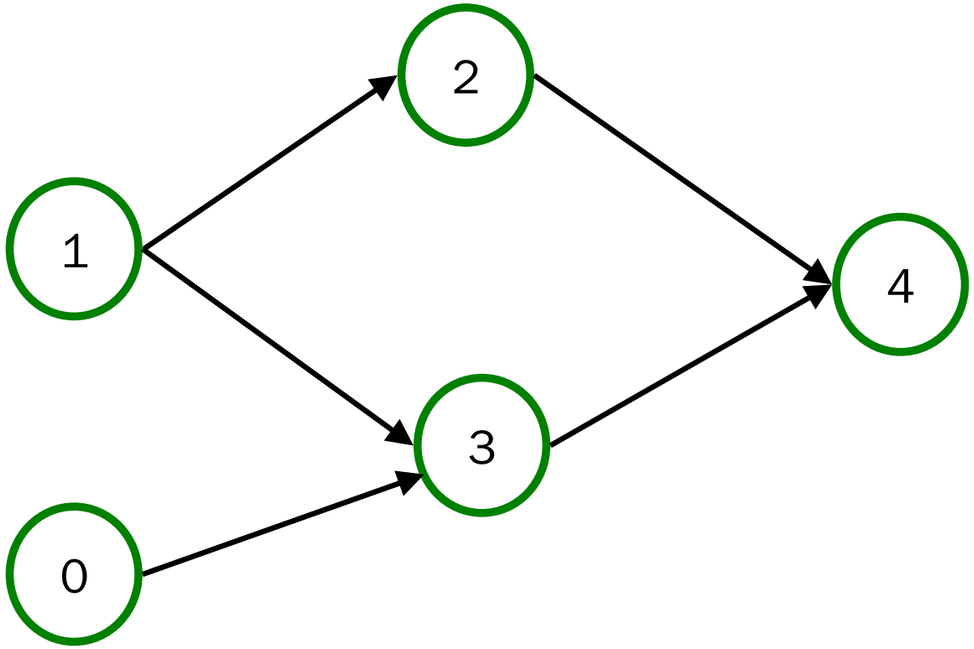
Problem: Given a DAG **`G=(V,E)`**, output all the vertices in order such that no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

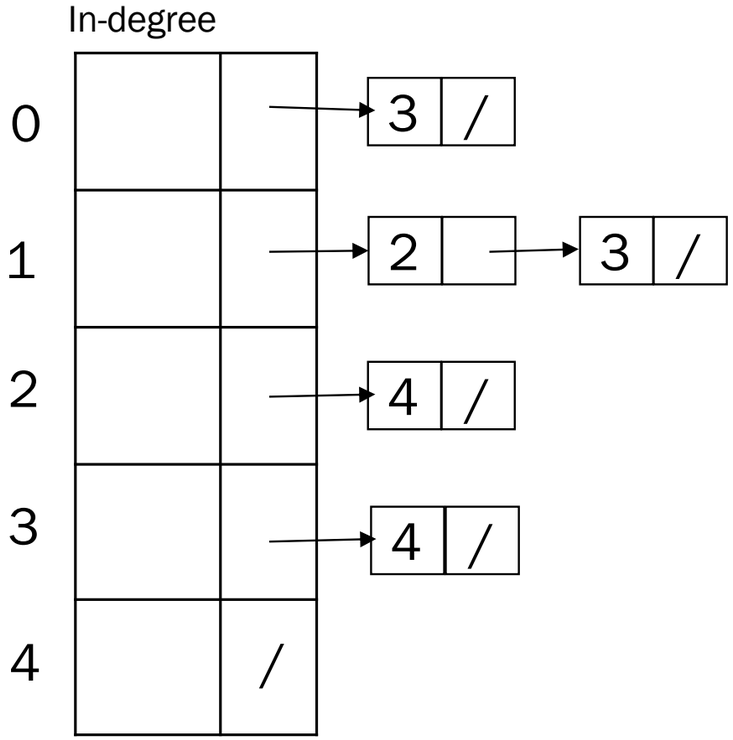
142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

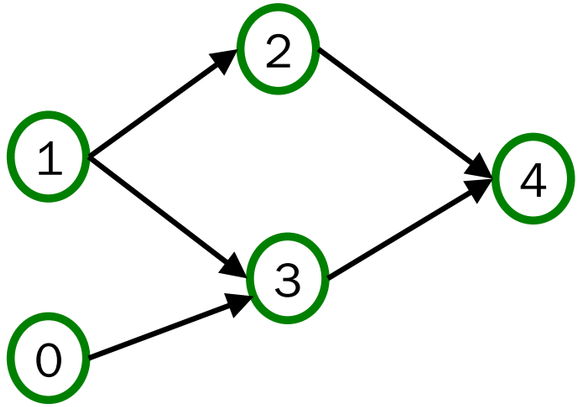


Valid Topological Sorts:

A First Algorithm for Topological Sort

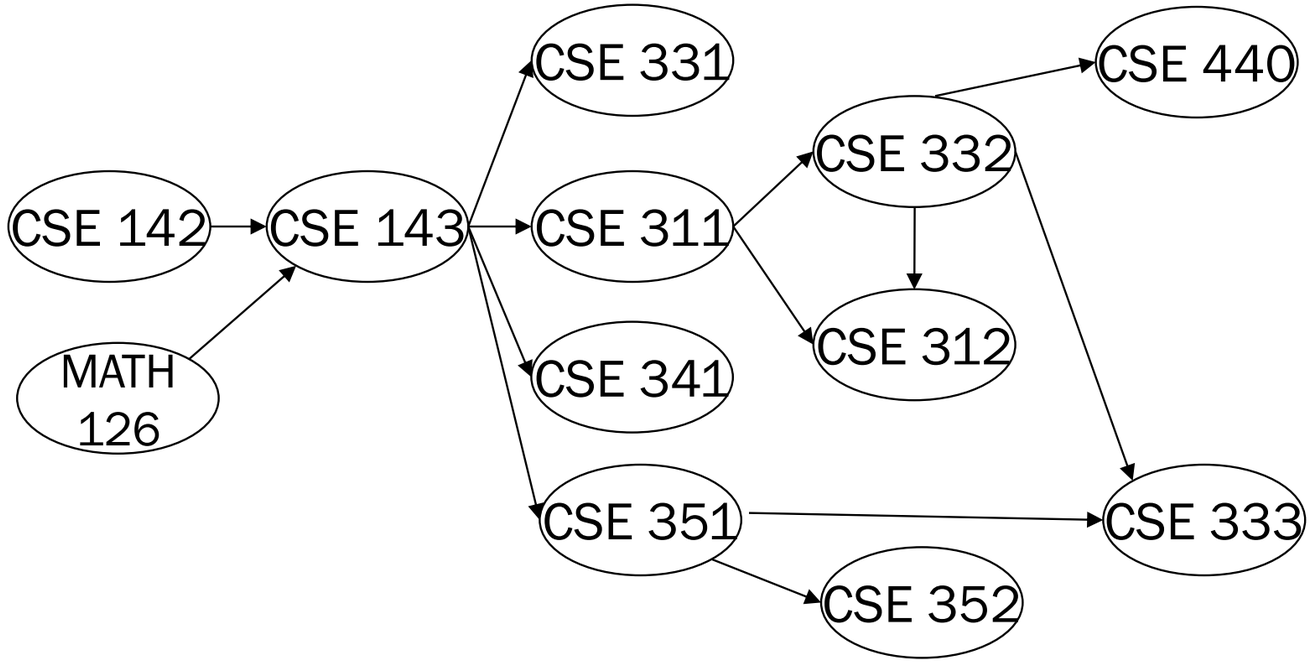
1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side
2. While there are vertices not yet output:
   a) Choose a vertex **v** labeled with in-degree of 0
   b) Output **v** and *conceptually* remove it from the graph
   c) For each vertex **w** adjacent to **v** (i.e. **w** such that (**v**,**w**) in **E**), decrement the in-degree of **w**





Example

Output:

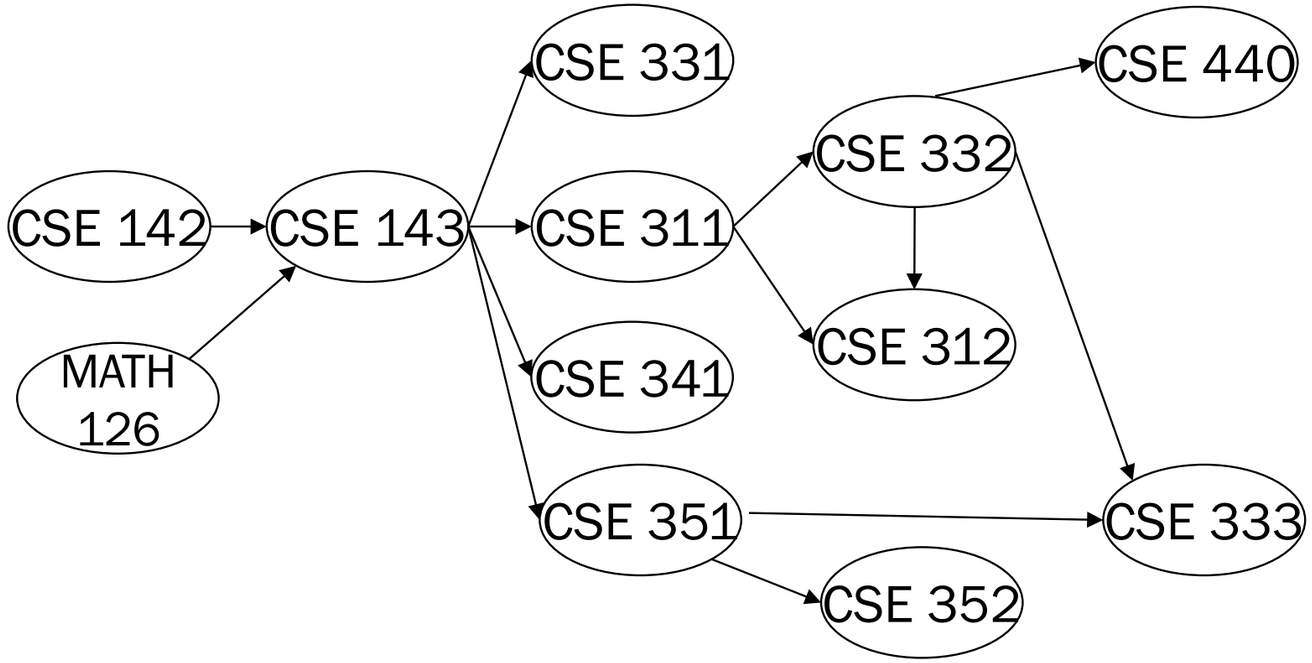


Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?												
In-degree	0	0	2	1	2	1	1	2	1	1	1	1

Example

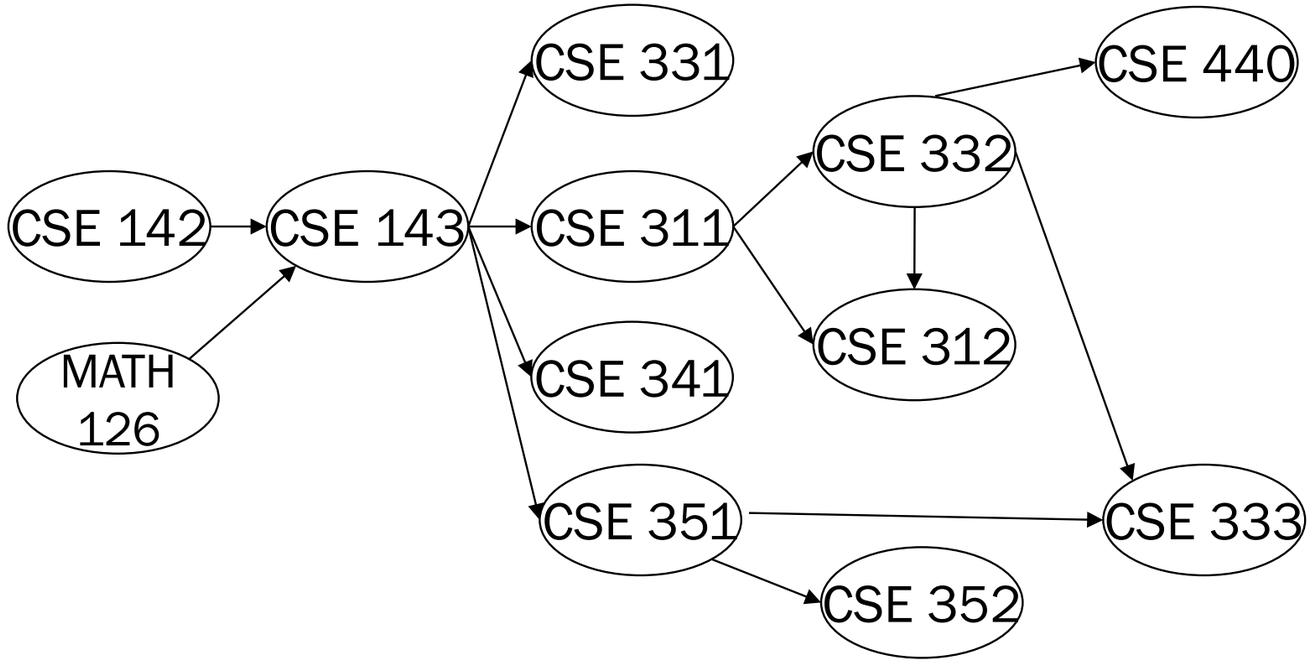
Output:

126



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x											
In-degree	0	0	~~2~~ 1	1	2	1	1	2	1	1	1	1

Example



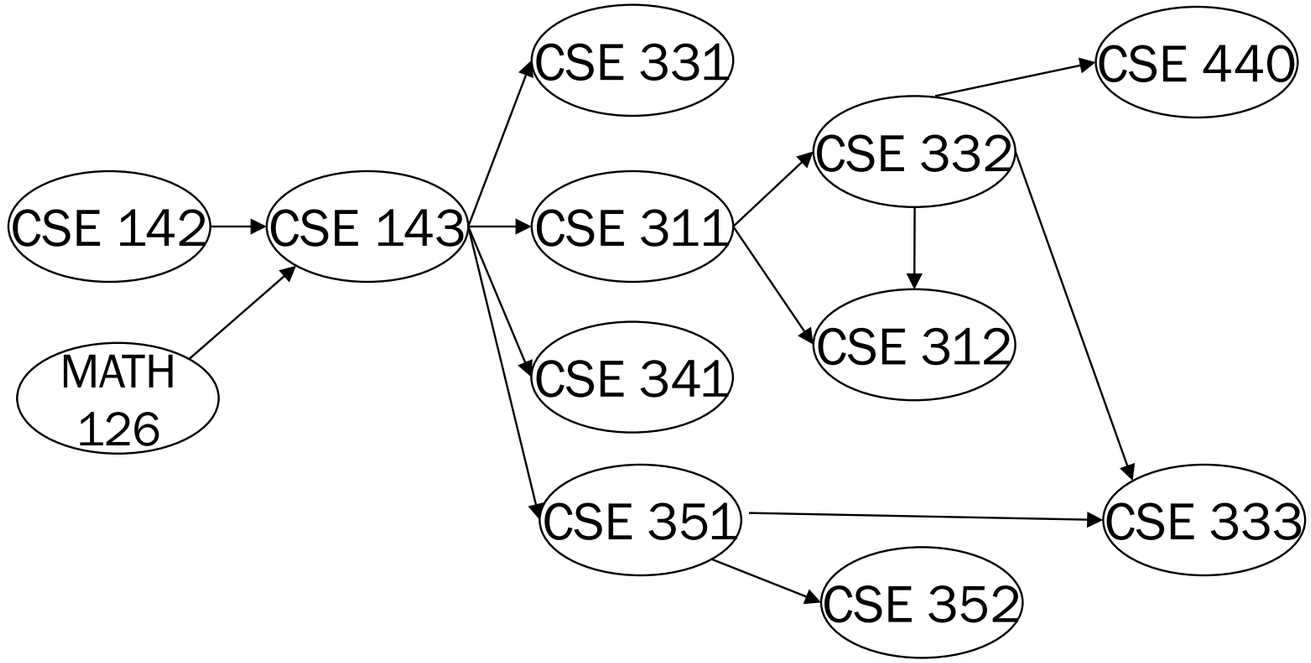
Output:

126

142

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x										
In-degree	0	0	~~1~~ 0	1	2	1	1	2	1	1	1	1

Example



Output:

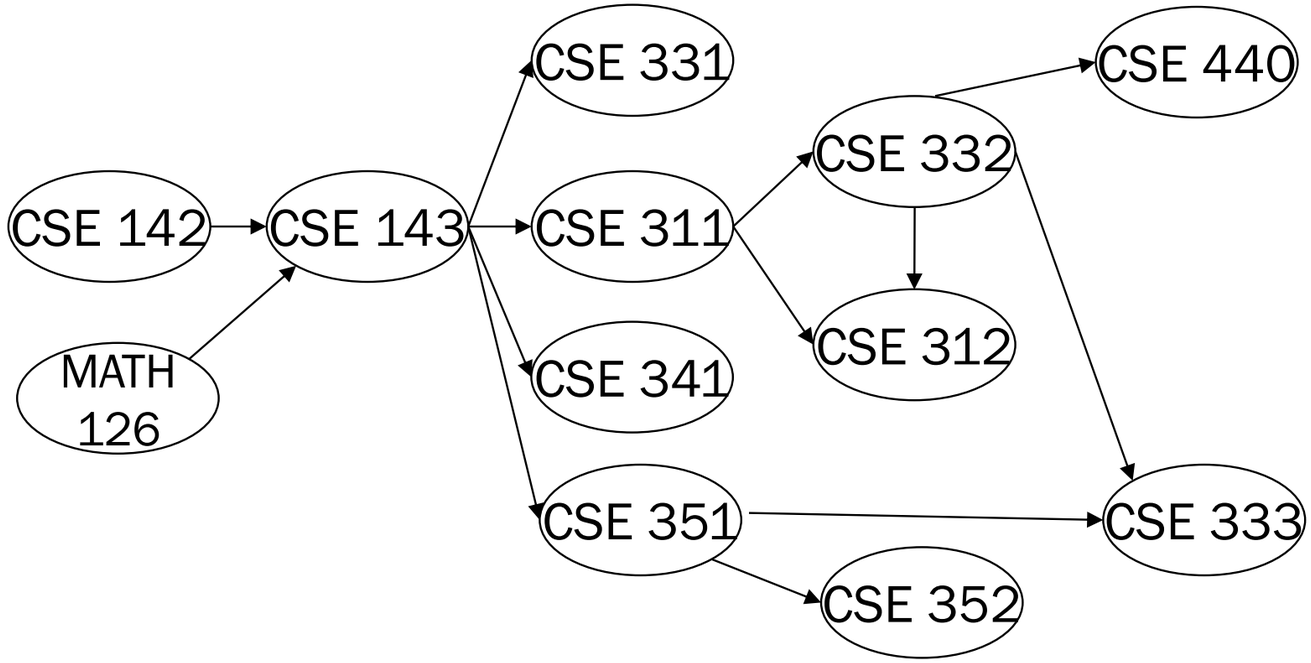
126

142

143

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x									
In-degree	0	0	0	~~1~~ 0	2	~~1~~ 0	1	2	~~1~~ 0	~~1~~ 0	1	1

Example



Output:

126

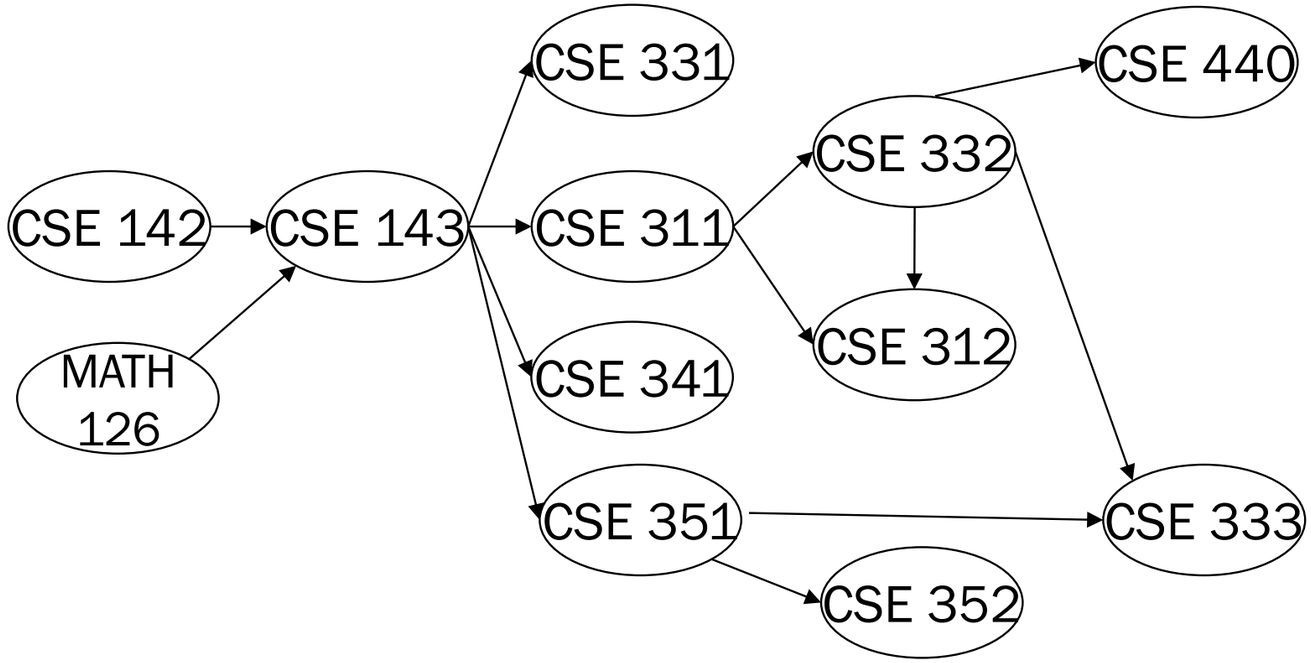
142

143

311

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x								
In-degree	0	0	0	0	~~2~~ 1	0	~~1~~ 0	2	0	0	1	1

Example



Output:

126

142

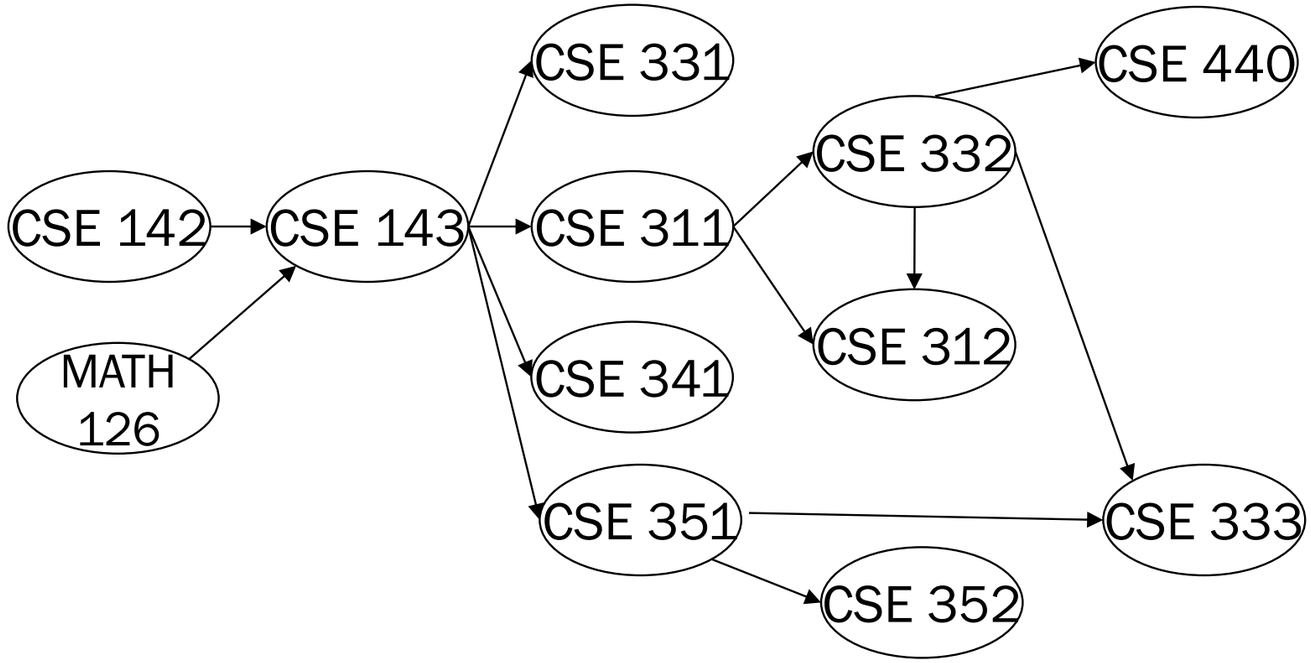
143

311

331

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x						
In-degree	0	0	0	0	1	0	0	2	0	0	1	1

Example



Output:

126

142

143

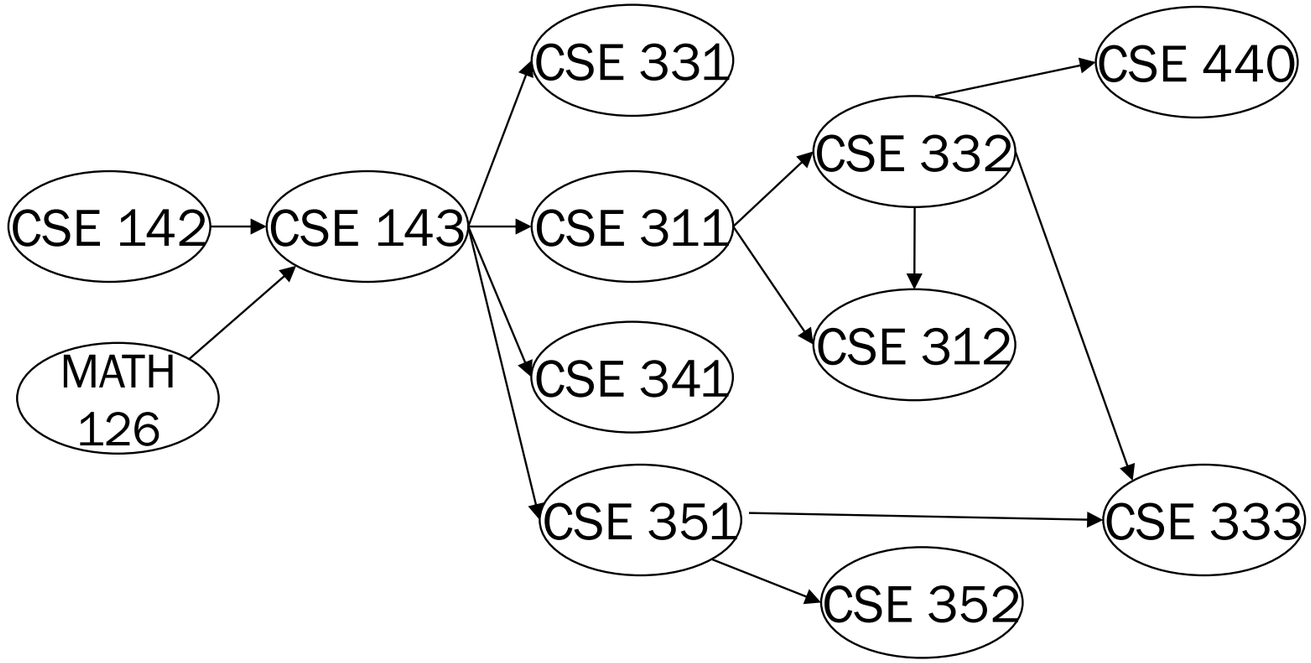
311

331

332

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x	x					
In-degree	0	0	0	0	~~1~~ 0	0	0	~~2~~ 1	0	0	1	~~1~~ 0

Example

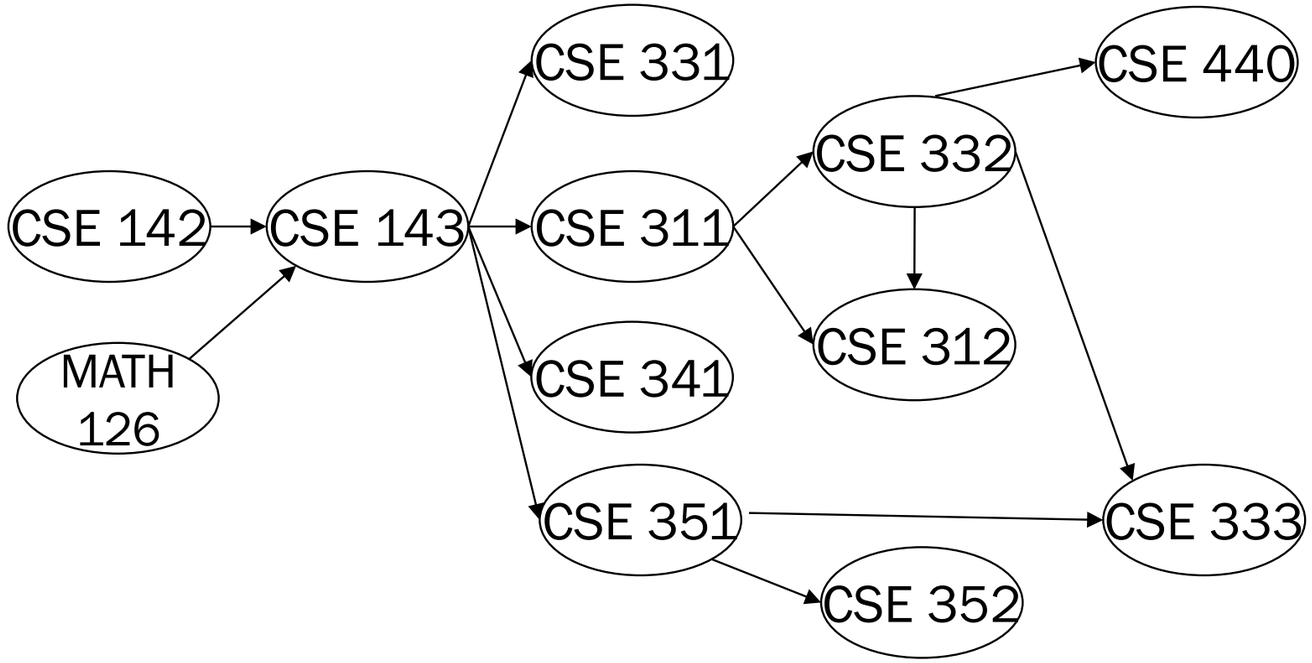


Output:

126
142
143
311
331
332
312

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x					
In-degree	0	0	0	0	0	0	0	1	0	0	1	0

Example

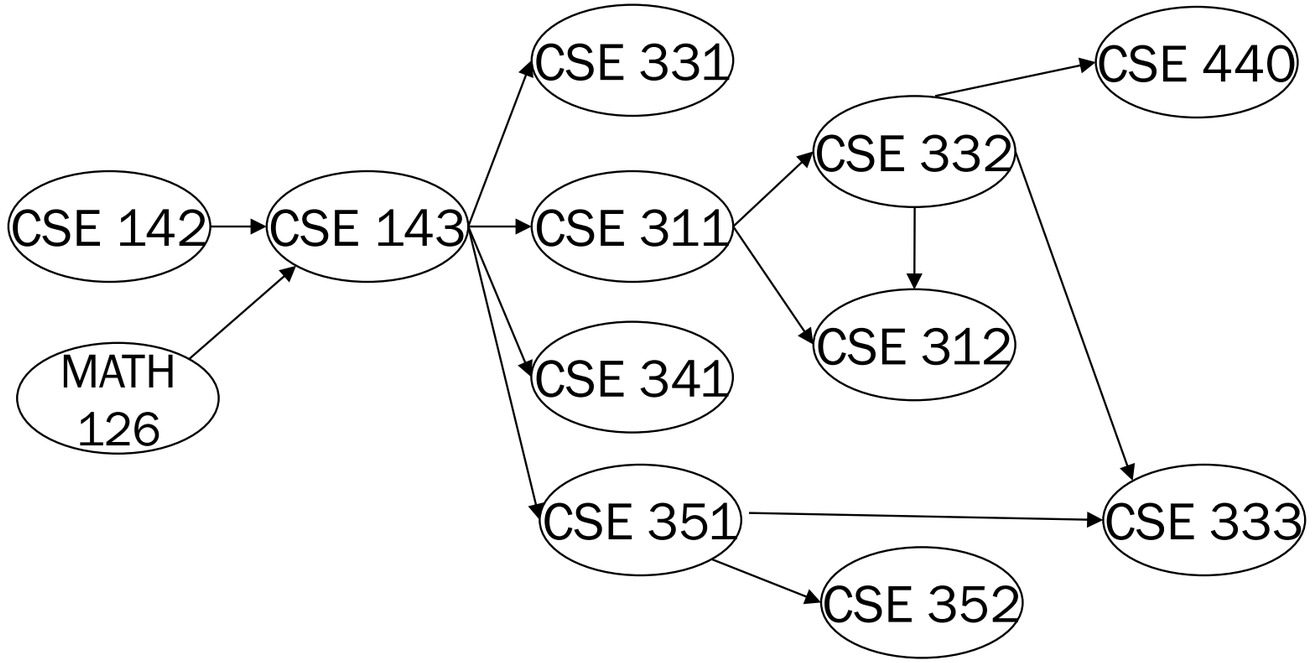


Output:

126
142
143
311
331
332
312
341

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x			
In-degree	0	0	0	0	0	0	0	1	0	0	1	0

Example

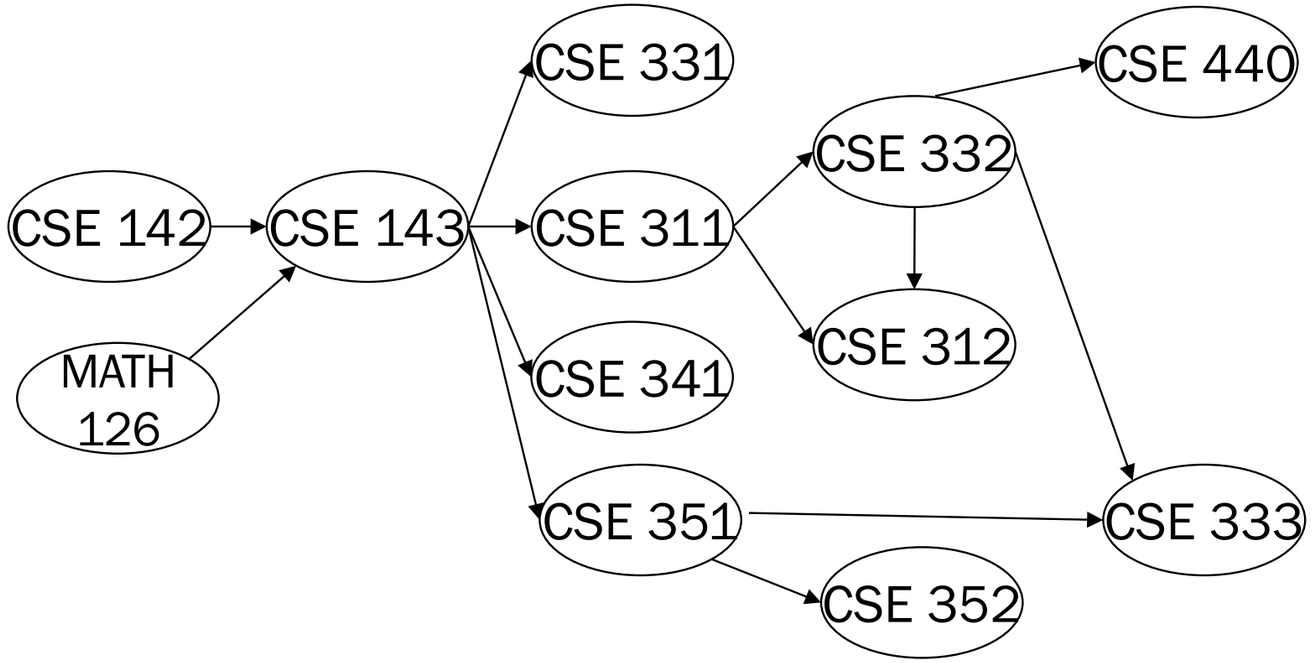


Output:

126
142
143
311
331
332
312
341
351

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x	x		
In-degree	0	0	0	0	0	0	0	~~1~~ 0	0	0	~~1~~ 0	0

Example

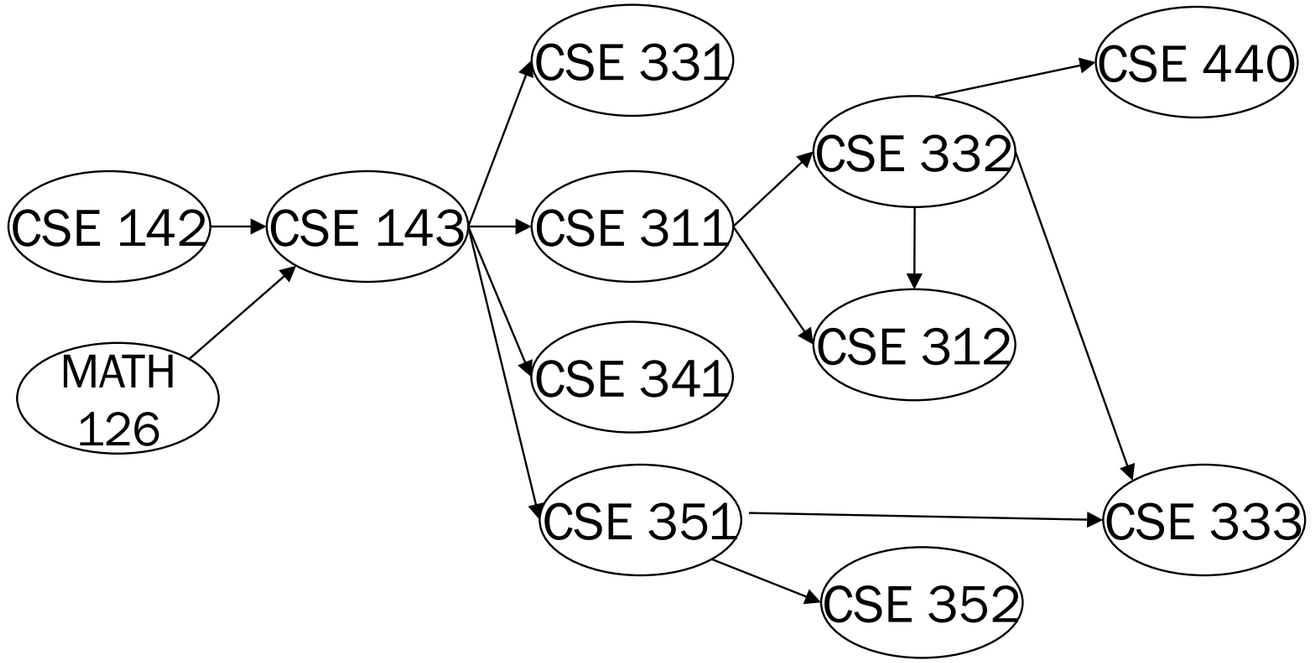


Output:

126
142
143
311
331
332
312
341
351
333

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x		
In-degree	0	0	0	0	0	0	0	0	0	0	0	0

Example

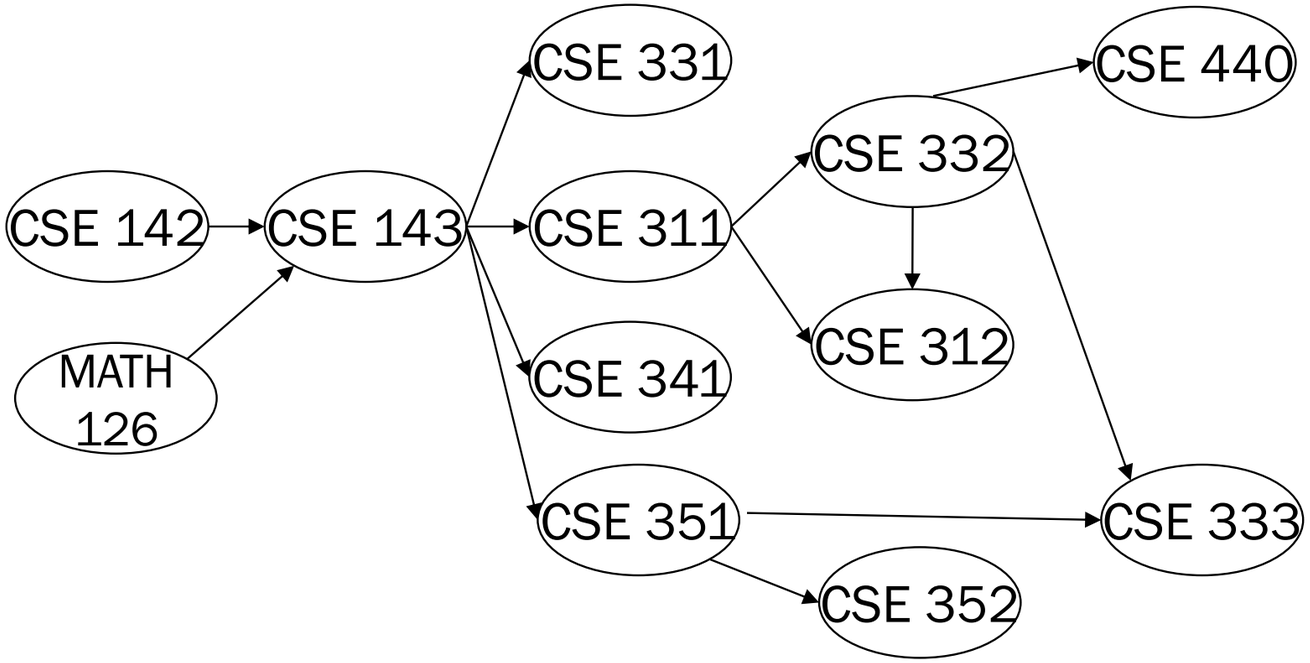


Output:

126
142
143
311
331
332
312
341
351
333
352

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	
In-degree	0	0	0	0	0	0	0	0	0	0	0	0

Example



Output:

126
142
143
311
331
332
312
341
351
333
352
440

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-degree	0	0	0	0	0	0	0	0	0	0	0	0

A couple of things to note

- Needed a vertex with in-degree of 0 to start
  - No cycles
- Ties between vertices with in-degrees of 0 can be broken arbitrarily
  - Potentially many different correct orders

- What DAGs have exactly 1 topological ordering?

Topological Sort: Running time?

```
labelEachVertexWithItsInDegree();

for(ctr=0; ctr < numVertices; ctr++){
  v = findNewVertexOfDegreeZero();
  put v next in output
   for each w adjacent to v
     w.indegree--;
}
```

Topological Sort: Running time?

```
labelEachVertexWithItsInDegree();

for(ctr=0; ctr < numVertices; ctr++){
  v = findNewVertexOfDegreeZero();
  put v next in output
   for each w adjacent to v
     w.indegree--;
}
```

- What is the worst-case running time?
  - Initialization $O(|V| + |E|)$ (assuming adjacency list)
  - Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
  - Sum of all decrements $O(|E|)$ (assuming adjacency list)
  - So total is $O(|V|^2 + |E|)$ – not good for a sparse graph!

Doing better

The trick is to avoid searching for a zero-degree node every time!

- Keep the "pending" zero-degree nodes in a list, stack, queue, box, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) v = dequeue()
   b) Output **v** and remove it from the graph
   c) For each vertex **w** adjacent to **v** (i.e. **w** such that (**v**,**w**) in **E**), decrement the in-degree of **w**, if new degree is 0, enqueue it

Topological Sort(optimized): Running time?

```
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++){
  v = dequeue();
  put v next in output
   for each w adjacent to v {
     w.indegree--;
     if(w.indegree==0)
       enqueue(w);
  }
}
```

Topological Sort(optimized): Running time?

```
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++){
  v = dequeue();
  put v next in output
   for each w adjacent to v {
     w.indegree--;
     if(w.indegree==0)
       enqueue(w);
  }
}
```

- What is the worst-case running time?
  - Initialization: $O(|V|+|E|)$ (assuming adjacency list)
  - Sum of all enqueues and dequeues: $O(|V|)$
  - Sum of all decrements: $O(|E|)$ (assuming adjacency list)
  - So total is $O(|E| + |V|)$ – much better for sparse graph!

Topological Sort Uses

- Figuring out how to finish your degree
- Determining the order to compile files using a Makefile
- Determining what assignment you should work on next

- In general, taking a dependency graph and coming up with an order of execution