CSE332: Data Abstractions

Lecture 13: Graph Traversal / Topological Sort

James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman, Richard Ladner, Steve Seitz

Midterm Question 1b

```
for(i = 1; i <= n; i = i * 2) {
   for(j = 0; j < i; j++) {
      sum++;
   }
}
```

For **n = 64**, outer loop will set **i** to values: 1, 2, 4, 8, 16, 32, 64

sum will have final value 1 + 2 + 4 + 8 + 16 + 32 + 64 = **2n - 1**

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code

- Indentation. Be consistent about tabs versus spaces.
  - Look at your code in a non-Eclipse editor and make sure it looks right (e.g., emacs, vim, notepad)

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code

- Remember your 142 / 143 style rules

  - Constants should be constant and capitalized

```
private static final int INITIAL_ARRAY_SIZE = 10;
```

  - Use proper Java naming conventions

```
camelCase
```

  - Give useful names to variables and methods

`a` is not an acceptable name for your inner array

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code

- Remember your 142 / 143 style rules

  - Comments! Write them!
    - They are not just for public methods
    - Many of you missing them for private methods, inner classes
    - This is **not** a helpful comment

```
// constructor
public ArrayStack() {
     ...
}
```

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code

- Remember your 142 / 143 style rules

  - Comments! Write them!
    - Useful to frame comments in terms of pre/post conditions
      - The expected input (valid ranges for each parameter)
      - Under what conditions exceptions will thrown
      - What will be returned

    - Also comment complex sections of code, as you will not remember exactly what you were doing 6 weeks later

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code
- Remember your 142 / 143 style rules
  - Boolean zen

```
if (size == 0) {          vs.      return size == 0;
   return true;
} else {
   return false;
}
```

Style Points

- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code

- Remember your 142 / 143 style rules

  - Boolean zen

```
if (size == 0) {          vs.     return size == 0;
   return true;
} else {
   return false;
}
```

Style Points

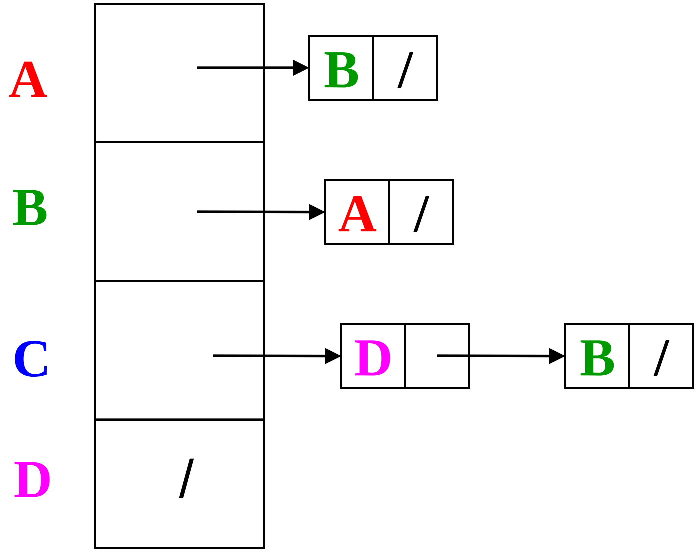
- There will be more opportunities to lose style points on Project 2
  - But here are some common issues in Project 1 code
- Do not use unnecessary fields that introduce more potential errors
  - No need for `size` in the ListStack if you only use it to check whether the list was empty (i.e., just check if `head` is `null`)
- Whitespace can be beautiful! Use it appropriately for readability
  `return size==0?true:false;` is bad zen and hard to read
- Do not delay the write up until 30 minutes before the project is due
  - It will be a worth a substantial chunk of your points
  - Your responses will not be up to par

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 List Properties



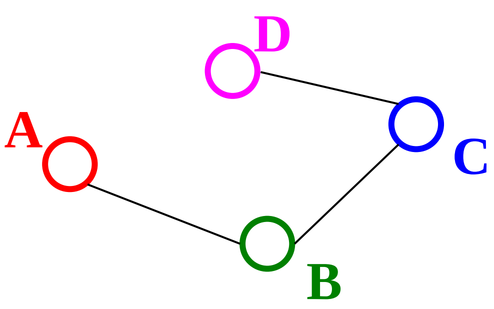
- Running time to:
  - Get all of a vertex's out-edges:
    O(*d*) where *d* is out-degree of vertex
  - Get all of a vertex's in-edges:
    O(|E|) (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    O(*d*) where *d* is out-degree of source
  - Insert an edge: *O*(1) (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 usually just stick with linked lists

Undirected Graphs

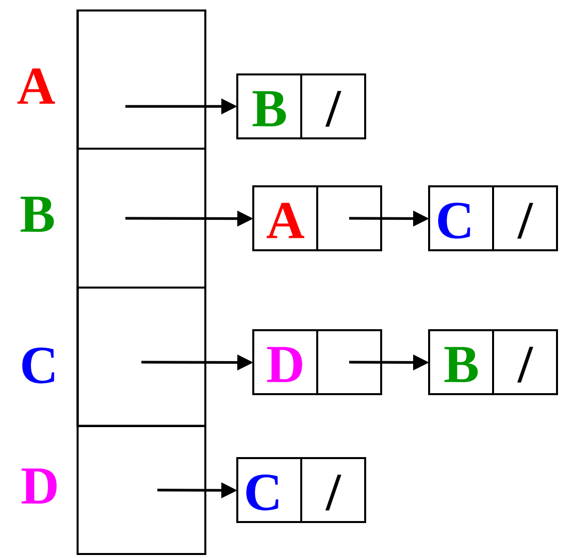
Adjacency matrices & adjacency lists both do fine for undirected graphs

- Matrix: Could save space by using only about half the array
  - How would you “get all neighbors”?
- Lists: Each edge in two lists to support efficient “get all neighbors”

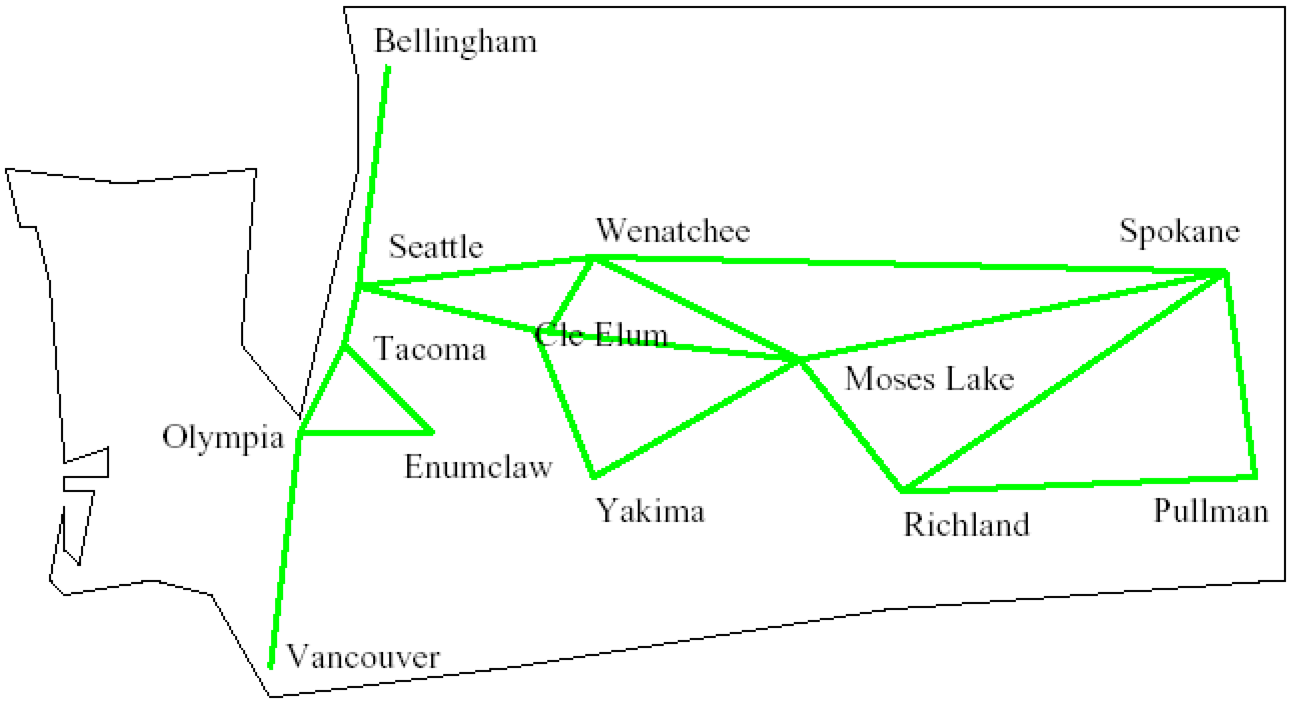
Example:



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

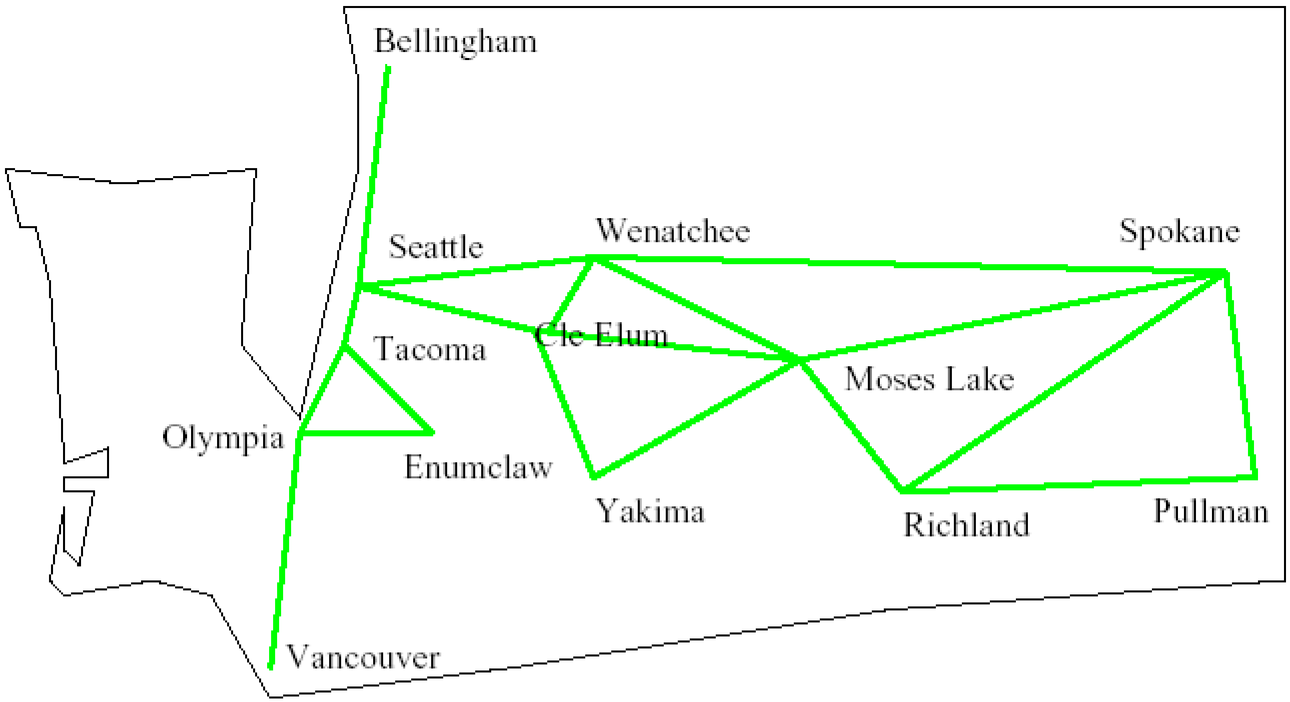


Some Applications: Moving Around Washington



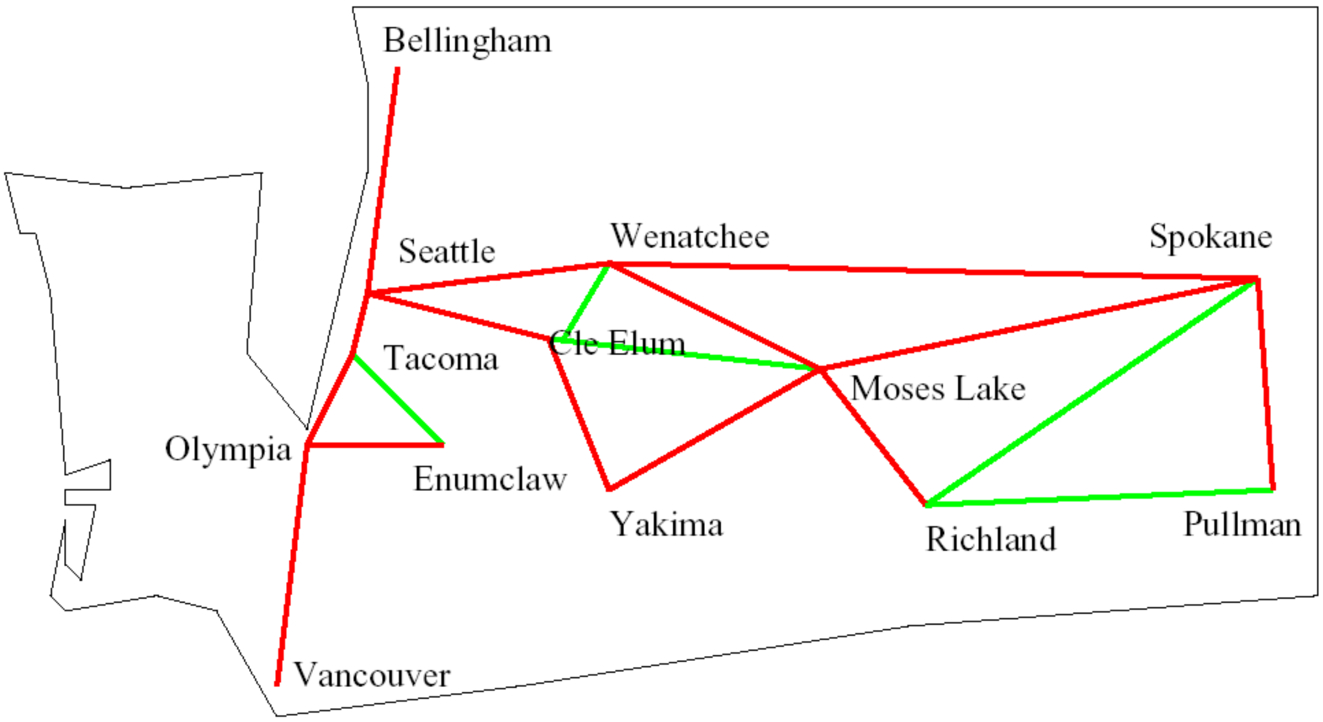
What's the *shortest way* to get from Seattle to Pullman?

Some Applications: Moving Around Washington



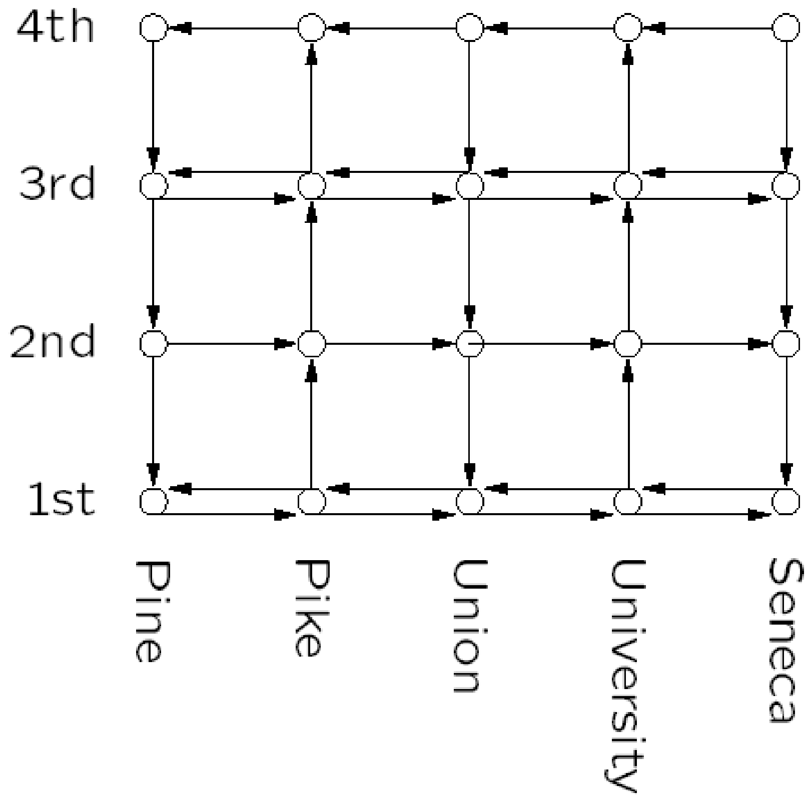
What's the *fastest way* to get from Seattle to Pullman?

Some Applications: Reliability of Communication



If Wenatchee's phone exchange *goes down*, can Seattle still talk to Pullman?

Some Applications: Bus Routes in Downtown Seattle



If we're at 3rd and Pine, how can we get to 1st and University using Metro?
How about 4th and Seneca?

Graph Traversals

For an arbitrary graph and a starting node **v**,
find all nodes *reachable* from **v** (i.e., there exists a path)

- Possibly “do something” for each node
- e.g., print to output, set some field, return from iterator, etc.

Related Problems:

- Is an undirected graph connected?
- Is a directed graph weakly / strongly connected?
  - For strongly, need a cycle back to starting node

Basic Idea:

- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once

Abstract Idea

```
traverseGraph(Node start) {
    Set pending = emptySet();
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
      next = pending.remove()
      for each node u adjacent to next
         if(u is not marked) {
           mark u
           pending.add(u)
         }
    }
}
```

Why do we need to **mark** nodes?

Running Time and Options

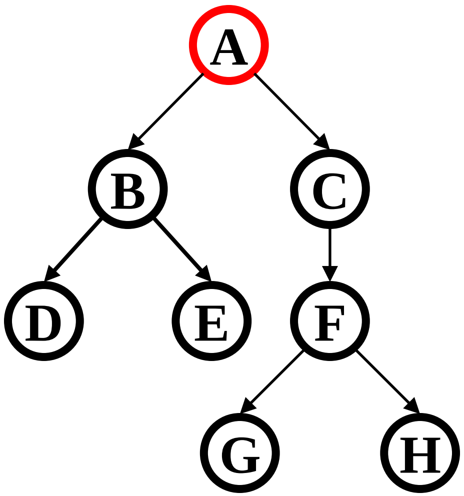
- Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$
  - Use an adjacency list representation

- The order we traverse depends entirely on add and remove
  - Popular choice: a stack “depth-first graph search” “DFS”
  - Popular choice: a queue “breadth-first graph search” “BFS”

- DFS and BFS are “big ideas” in computer science
  - Depth: recursively explore one part
    before going back to the other parts not yet explored
  - Breadth: Explore areas closer to the start node first

Recursive DFS, Example with Tree

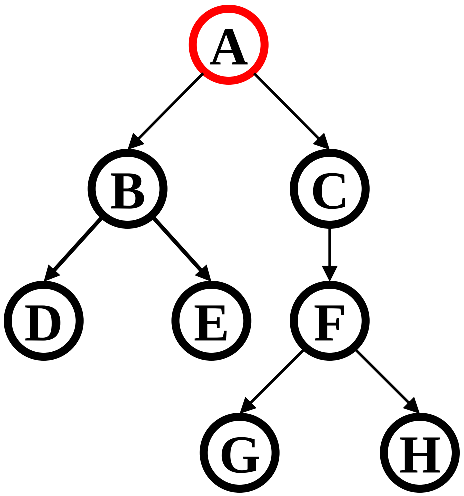
- A tree is a graph and DFS and BFS are particularly easy to "see"




```
DFS(Node start) {
  mark and process start
  for each node u adjacent to start
    if u is not marked
      DFS(u)
}
```

- Order processed: A, B, D, E, C, F, G, H
- Exactly what we called a "pre-order traversal" for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once

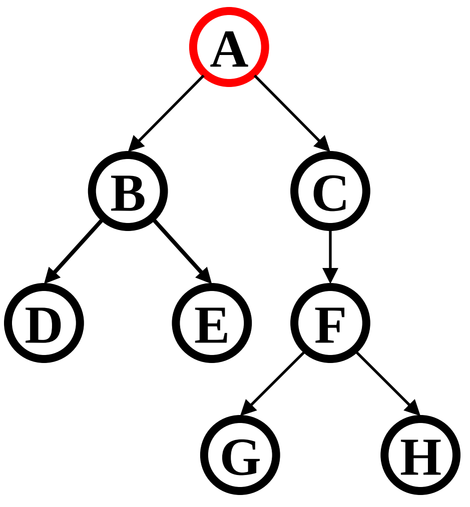
DFS with Stack, Example with Tree




```
DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and push onto s
  }
}
```

- Order processed: A, C, F, H, G, B, E, D
- A different but perfectly fine traversal

BFS with Queue, Example with Tree




```
BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and enqueue onto q
  }
}
```

- Order processed: A, B, C, D, E, F, G, H
- A "level-order" traversal

Comparison

- Breadth-first always finds shortest paths, i.e. “optimal solutions”
  - Better for “what is the shortest path from **`x`** to **`y`**”

- But depth-first can use less space in finding a path
  - If *longest path* in the graph is **`p`** and highest out-degree is **`d`** then DFS stack never has more than **`d*p`** elements
  - But a queue for BFS may hold $O(|V|)$ nodes

- A third approach:
  - *Iterative deepening (IDFS)*:
    - Try DFS up to recursion of **`K`** levels deep.
    - If that fails, increment **`K`** and start the entire search over
  - Like BFS, finds shortest paths. Like DFS, less space.

Saving the Path

- Our graph traversals can answer the reachability question:
  - "Is there a path from node x to node y?"

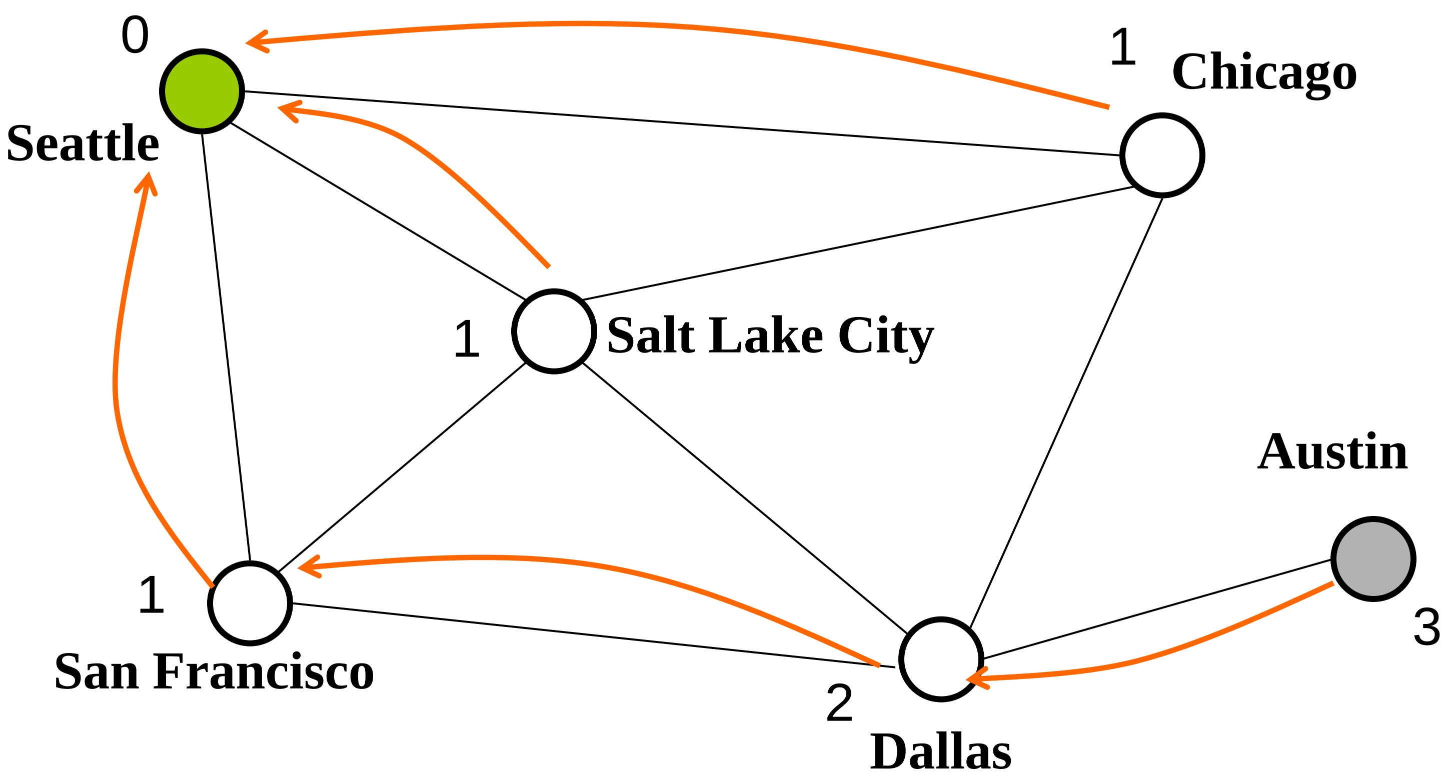
- But what if we want to actually output the path?

- Easy:
  - Instead of just "marking" a node, store the previous node along the path (when processing **`u`** causes us to add **`v`** to the search, set **`v.path`** field to be **`u`**)
  - When you reach the goal, follow **`path`** fields back to where you started (and then reverse the answer)

Example using BFS

What is a path from Seattle to Austin

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique

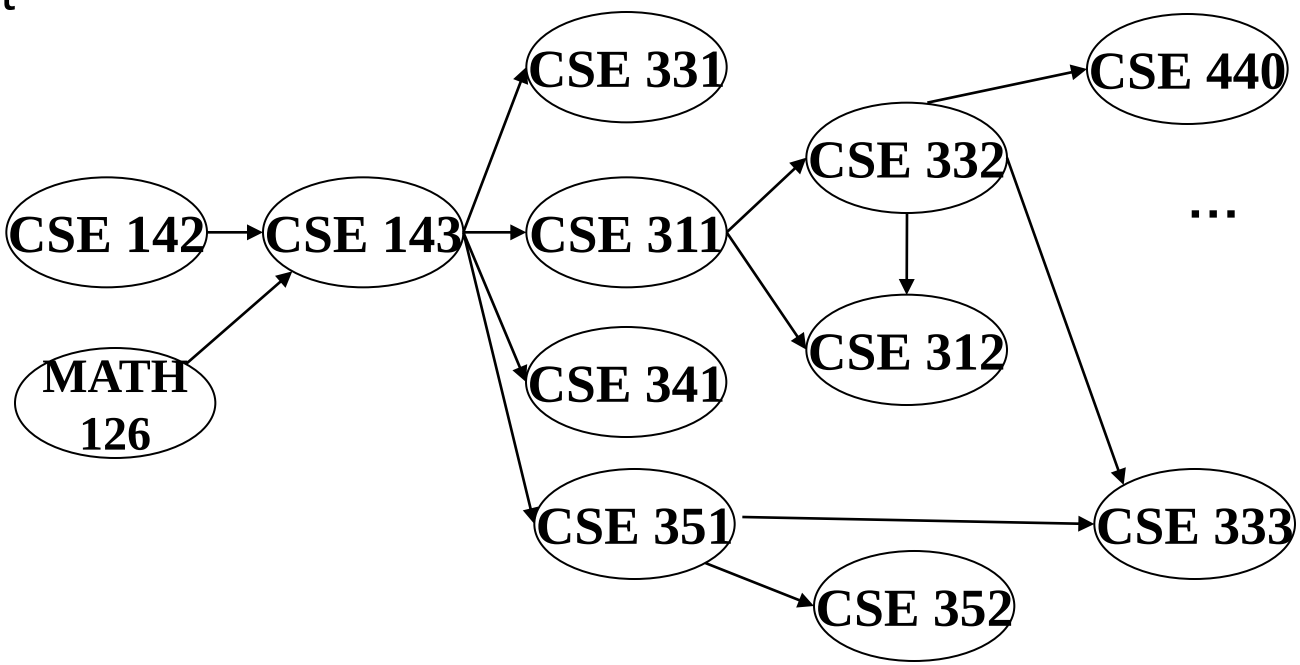


Disclaimer: Do not use for official advising purposes!
(Implies that CSE 332 is a pre-req for CSE 312 – not true)

Topological Sort

Problem: Given a DAG **`G=(V,E)`**, output all the vertices in order such that if 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

Questions and Comments

- Why do we perform topological sorts only on DAGs?
  - Because a cycle means there is no correct answer

- Is there always a unique answer?
  - No, there can be 1 or more answers; depends on the graph

- What DAGs have exactly 1 answer?
  - Lists

- Terminology: A DAG represents a partial order and
  a topological sort produces a total order that is consistent with it

Uses

- Figuring out how to finish your degree
- Computing order in which to recompute cells in a spreadsheet
- Determining the order to compile files with dependencies
- In general, using a dependency graph to find an order of execution

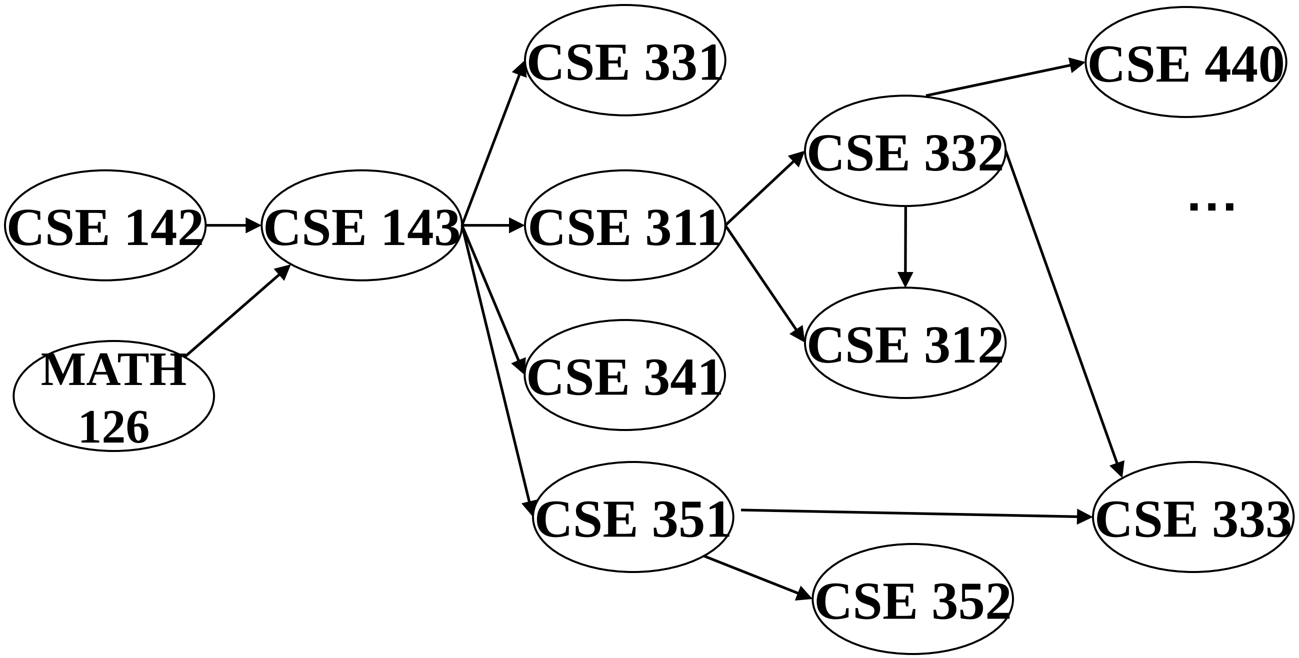
A First Algorithm for Topological Sort

1. Label each vertex with its in-degree
   – Think “write in a field in the vertex”
   – You could also do this with a data structure 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 **u** adjacent to **v**, decrement in-degree of **u**
      - (i.e., **u** such that (**v**,**u**) in **`E`**)

Example

Output:



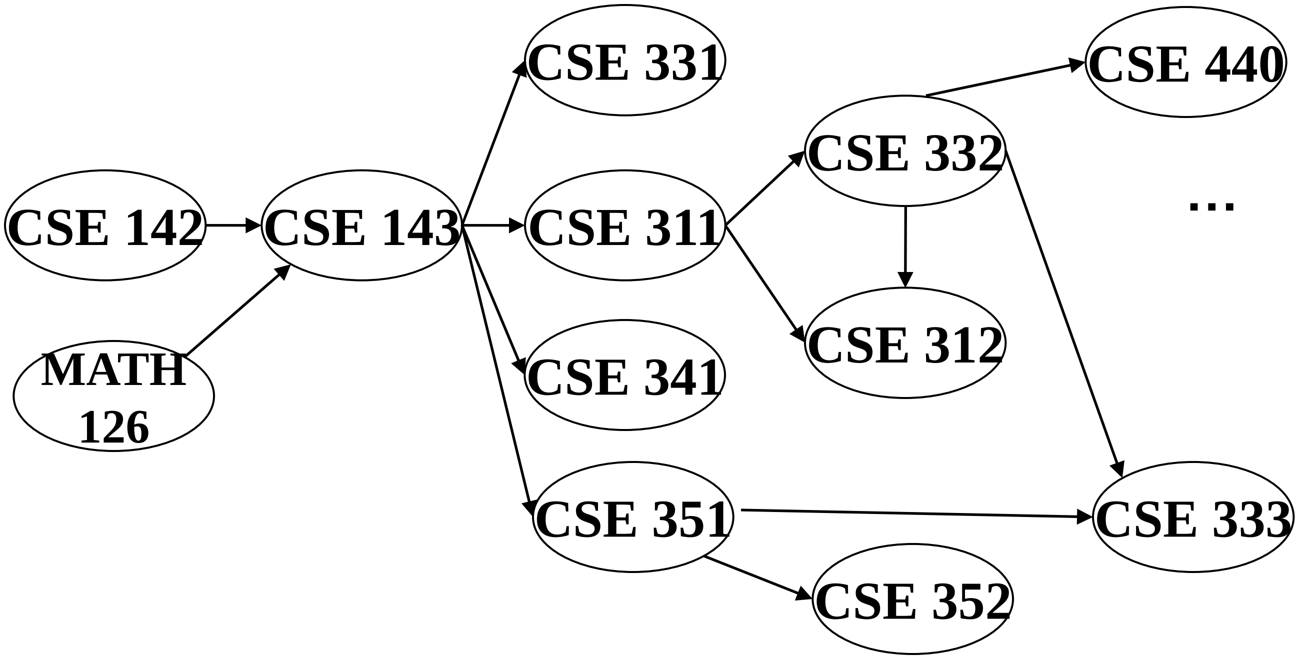
Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed?

In-degree:

Example

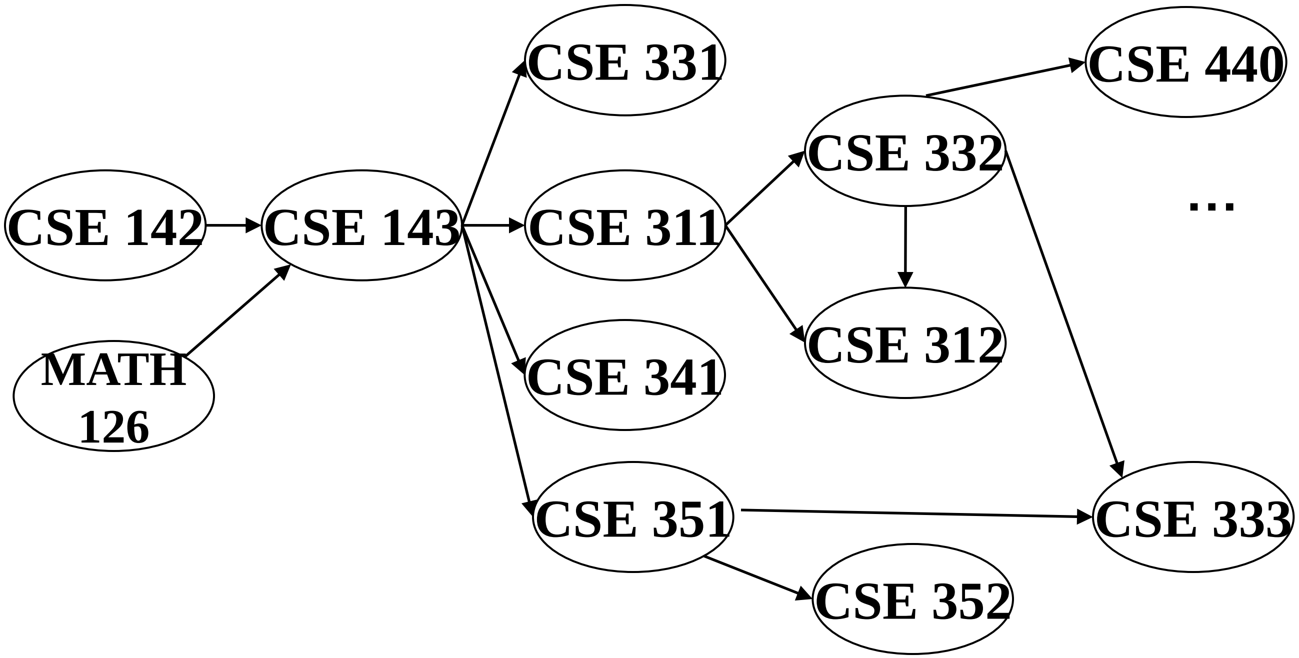
Output:



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

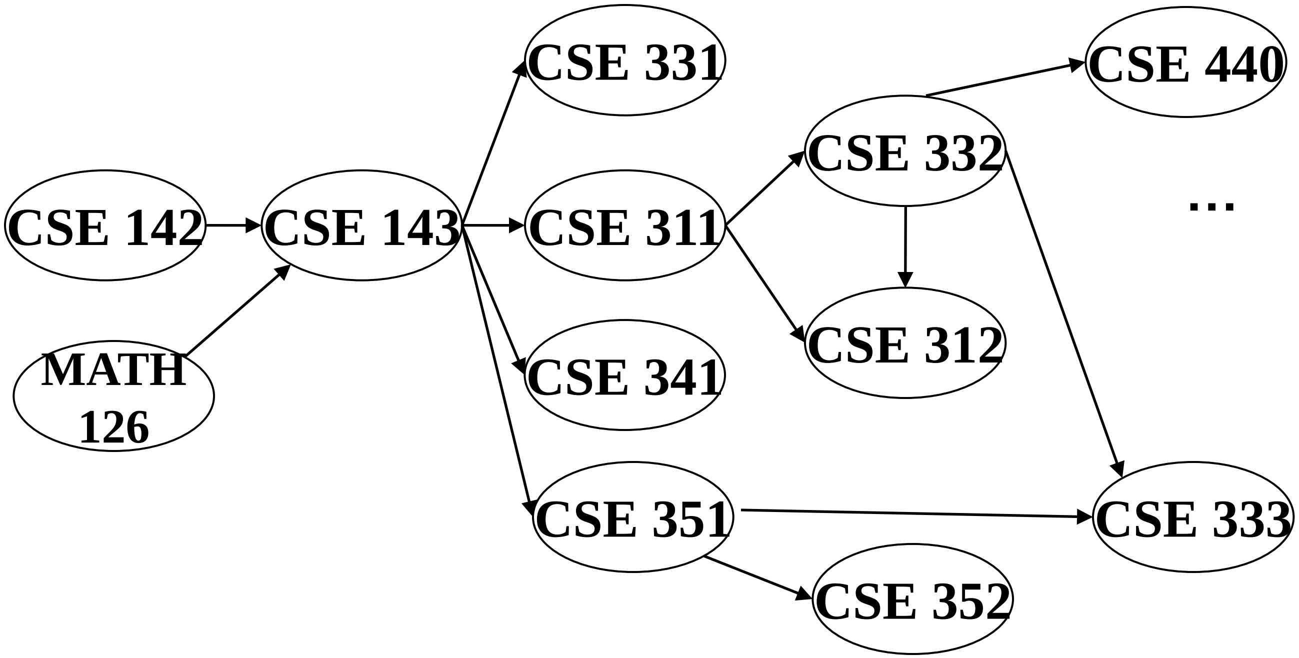
Output: 126



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

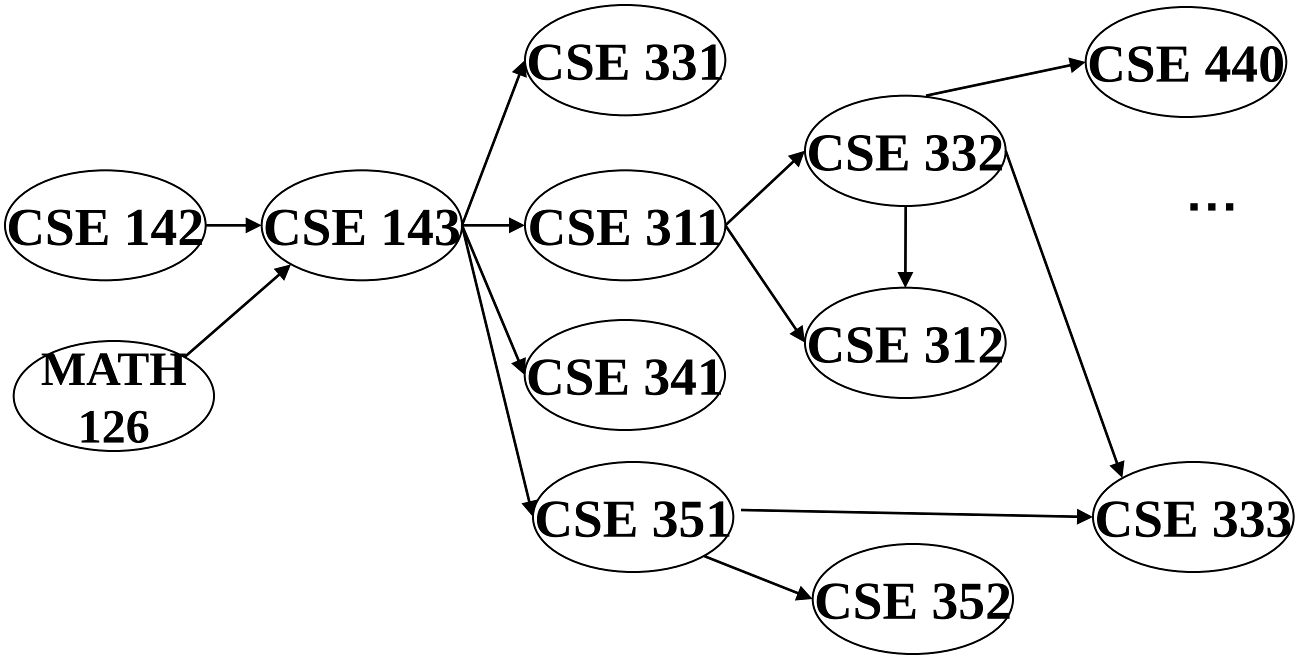
Output: 126
142



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

Example

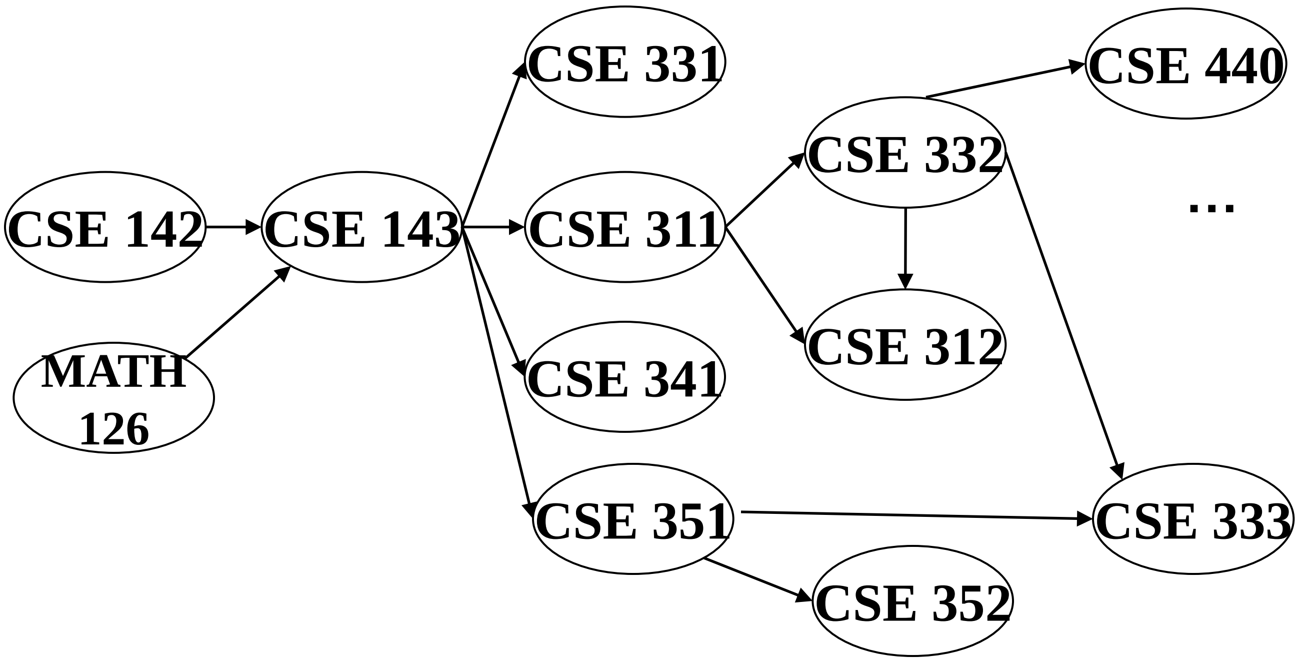
Output: 126
142
143



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

Example

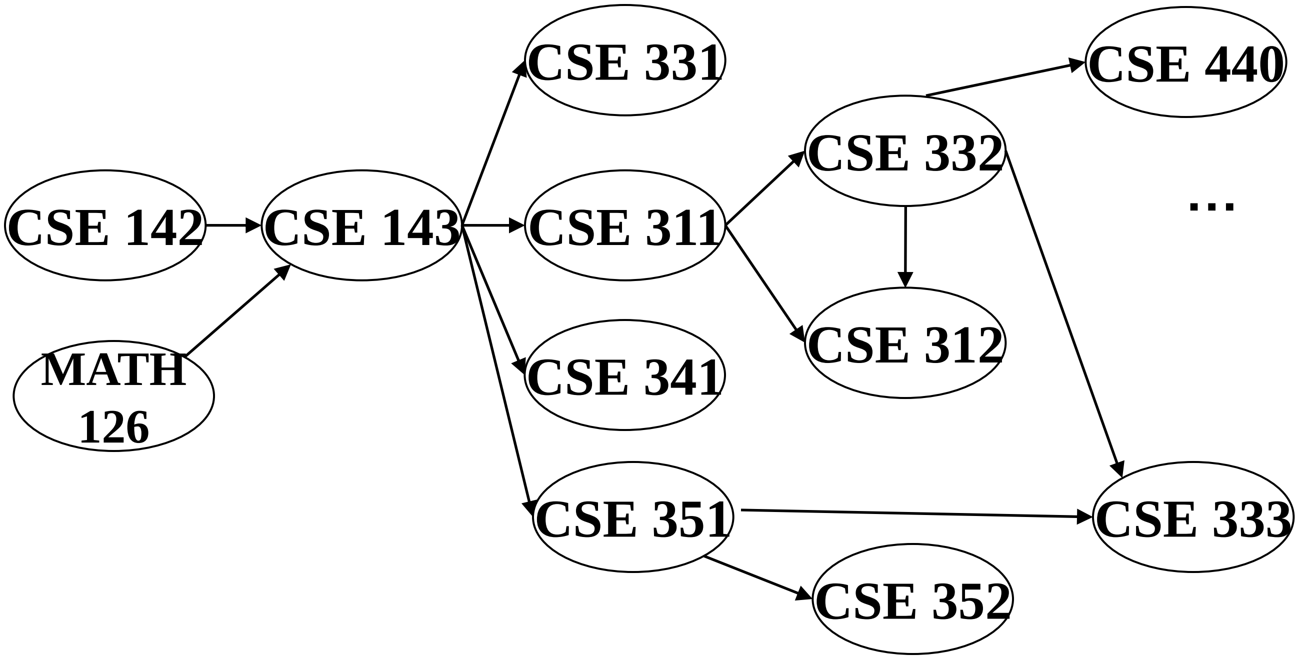
Output: 126
142
143
311
...



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

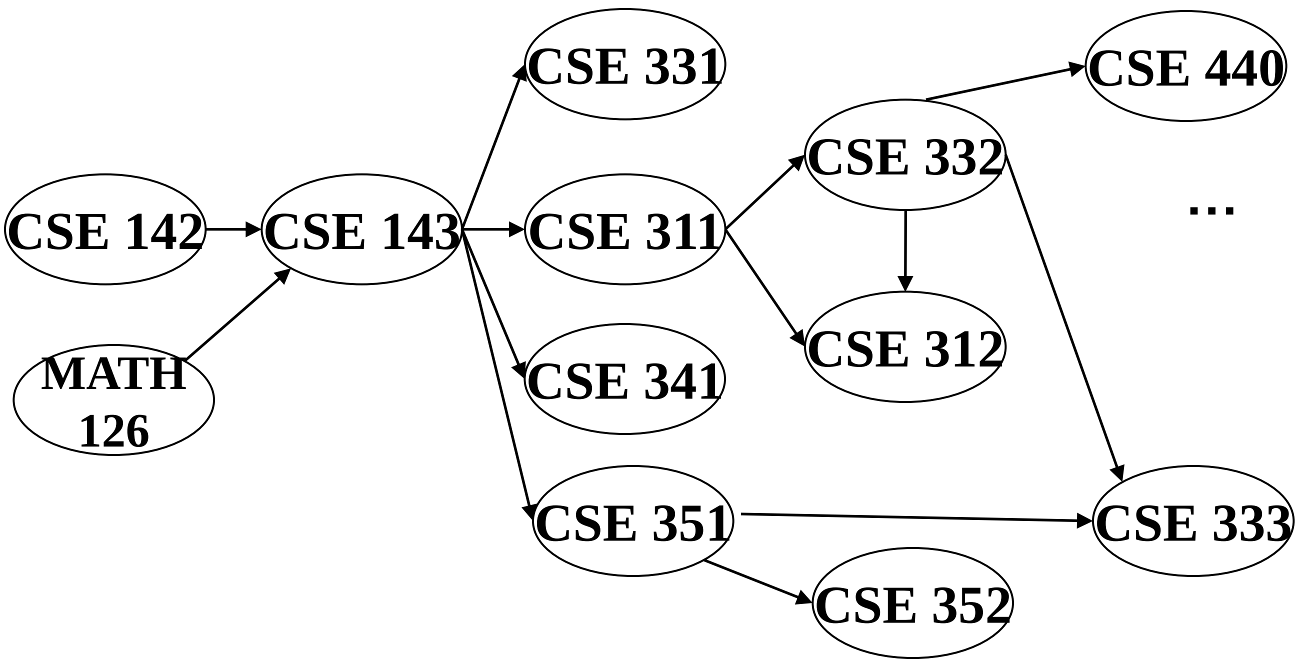
Example

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	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0		0	0		
			0									

Example

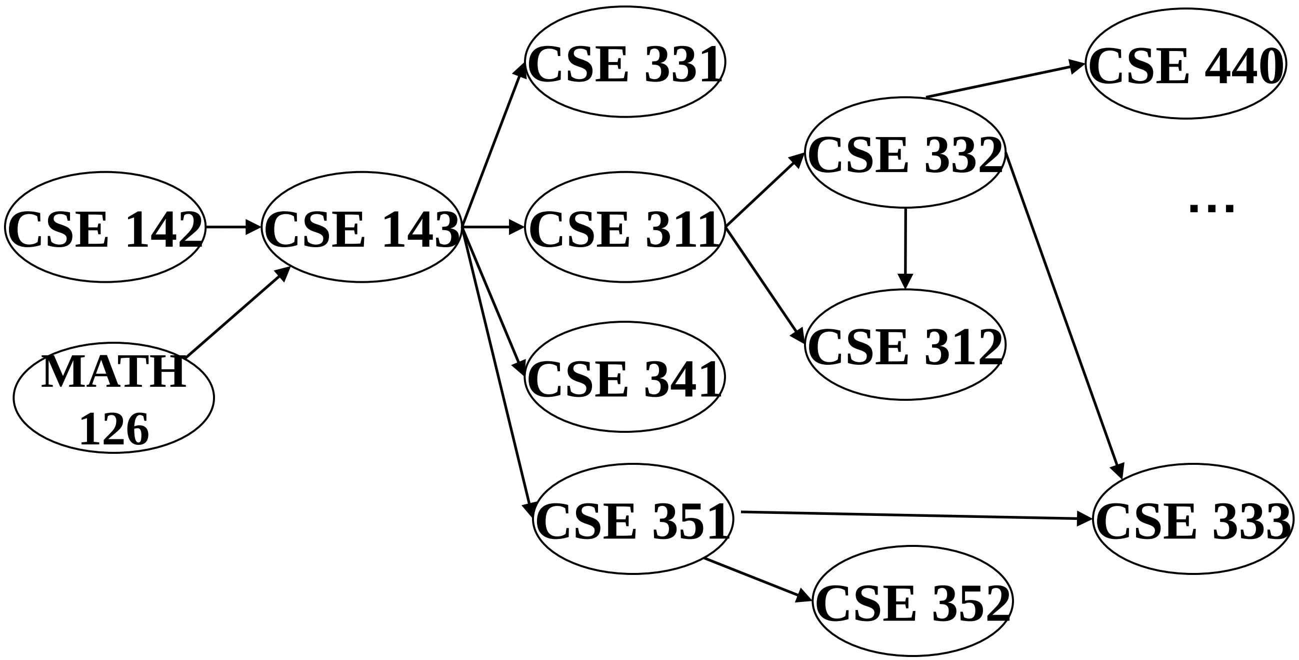


Output: 126
142
143
311
331
332

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

Example

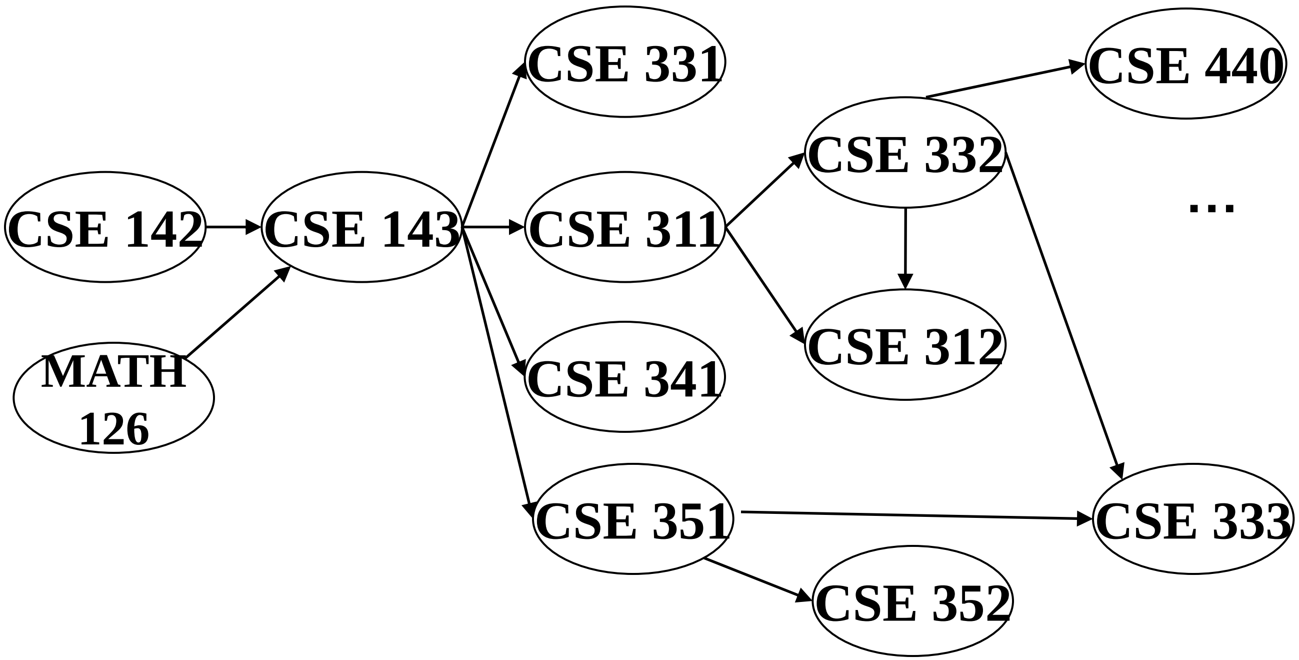
Output: 126
142
143
311
331
332
312



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	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

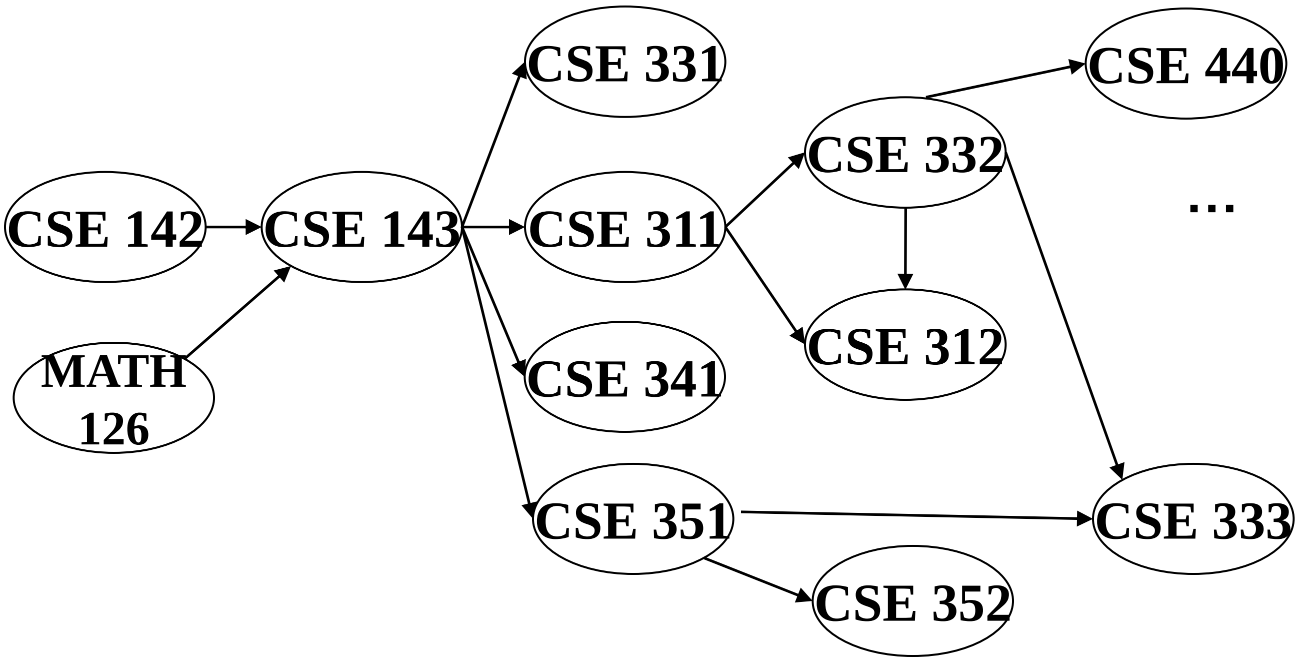
Example

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	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

Example

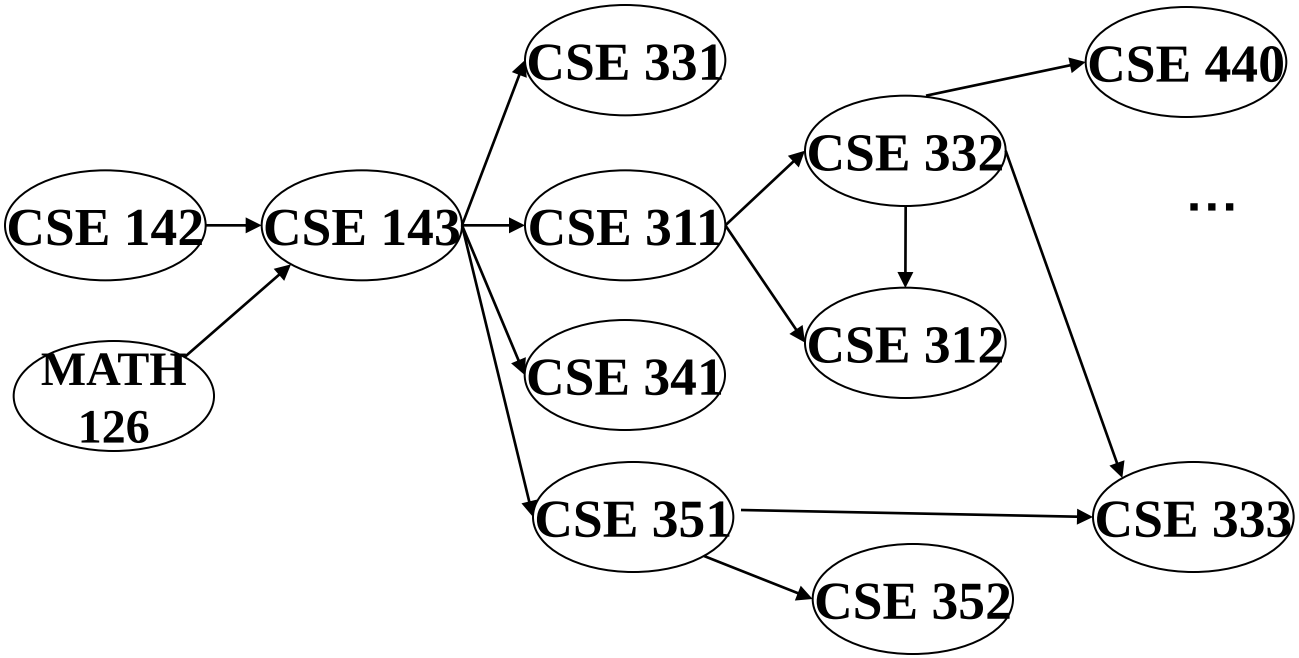


Output: 126
142
143
311
331
332
312
341
351

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	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

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

Example



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	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

Running Time?

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

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, bag, or something
- Order we process them affects the output but not correctness or efficiency, assuming 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 **u** adjacent to **v**,
      decrement the in-degree of **u**, if new degree is 0, enqueue it

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);
  }
}
```

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);
  }
}
```

- 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!