

# **Algorithms & Data Structures**

## **Lesson 12: Topological Sort / Graph Traversals**

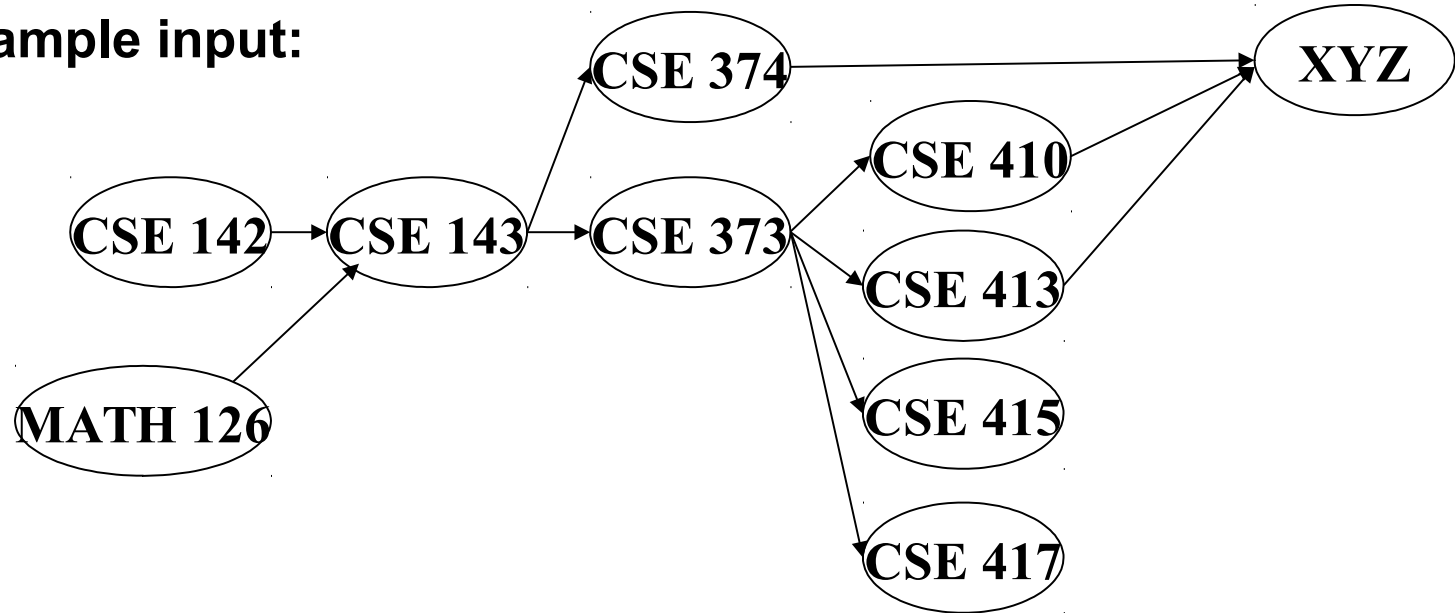
*Marc Gaetano*

Edition 2017-2018

# Topological Sort

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

**Example input:**

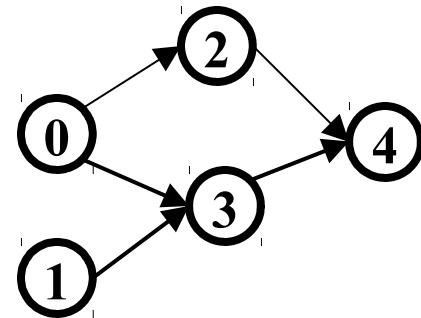


**One example output:**

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415

## 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
  - Graph with 5 topological orders:
- **Do some DAGs have exactly 1 answer?**
  - Yes, including all lists
- Terminology: A DAG represents a **partial order** and a topological sort produces a **total order** that is consistent with it



# Uses

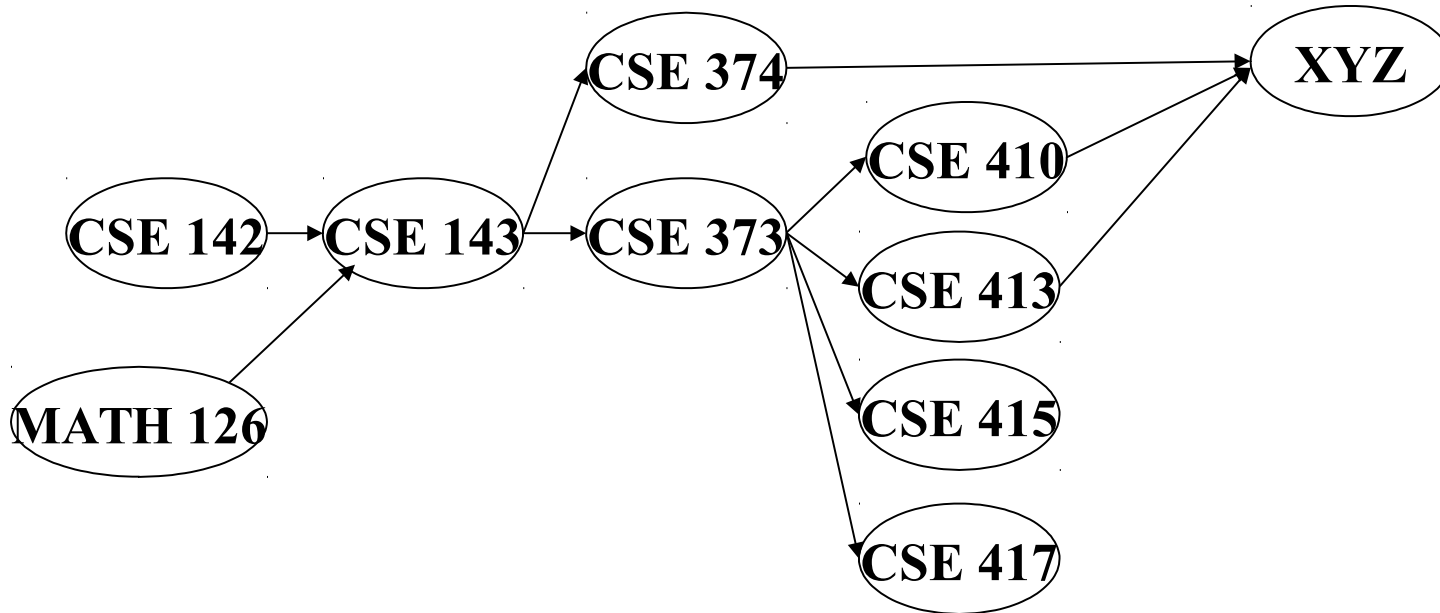
- Figuring out how to graduate
- Computing an order in which to recompute cells in a spreadsheet
- Determining an order to compile files using a Makefile
- In general, taking a dependency graph and finding an order of execution
- ...

# *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  $\mathbf{v}$  with labeled with in-degree of 0
  - b) Output  $\mathbf{v}$  and *conceptually* remove it from the graph
  - c) For each vertex  $\mathbf{u}$  adjacent to  $\mathbf{v}$  (i.e.  $\mathbf{u}$  such that  $(\mathbf{v}, \mathbf{u})$  in  $\mathbf{E}$ ),  
decrement the in-degree of  $\mathbf{u}$

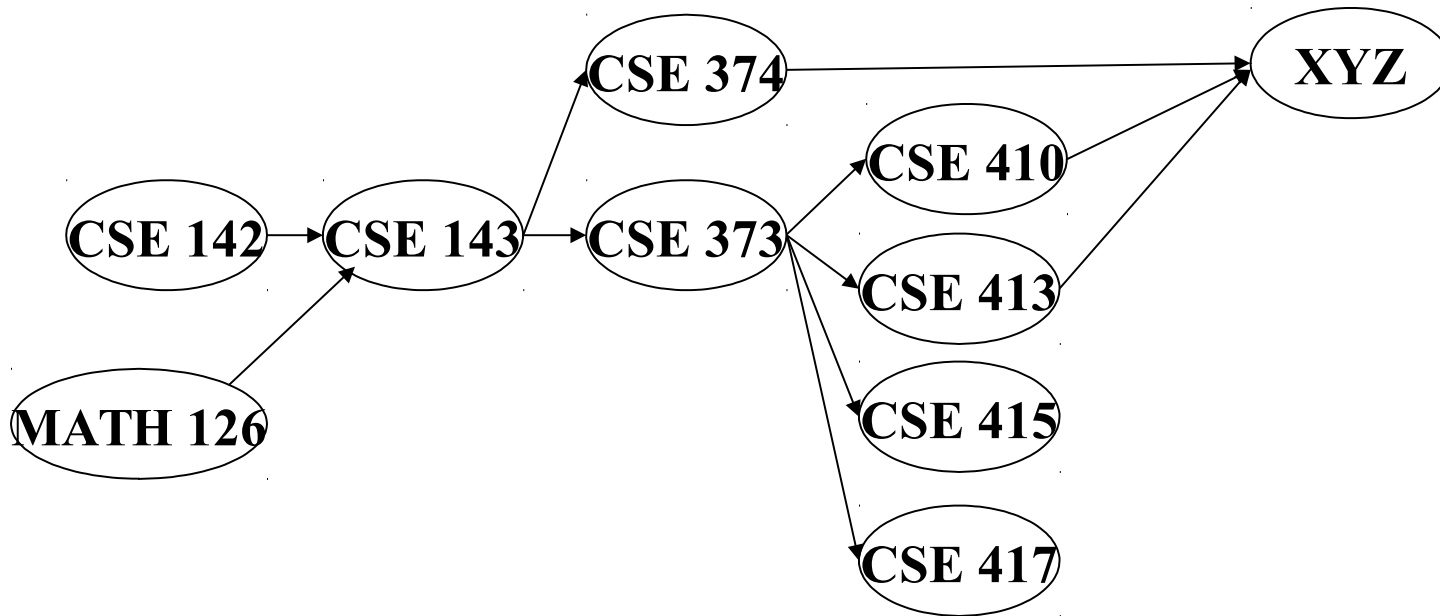
## Example

Output:



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?										
In-degree:	0	0	2	1	1	1	1	1	1	3

## Example

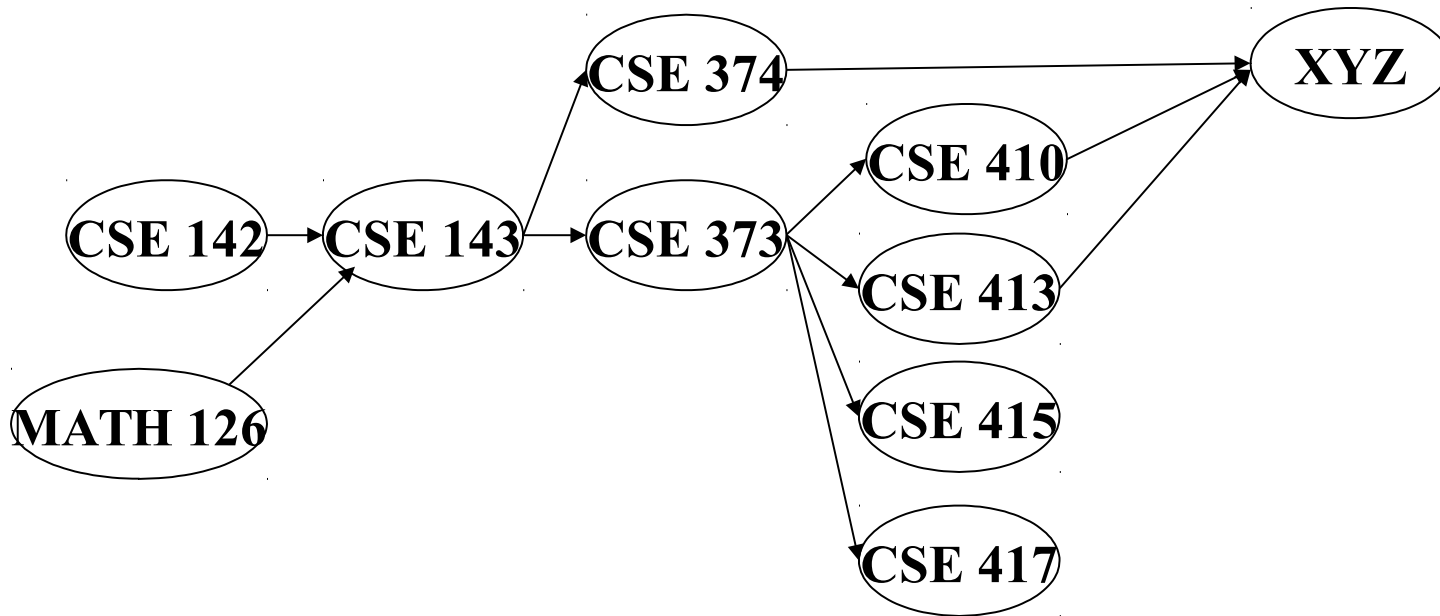


Output:

126

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?		x								
In-degree:	0	0	<del>2</del>	1	1	1	1	1	1	3
			1							

## Example



Output:

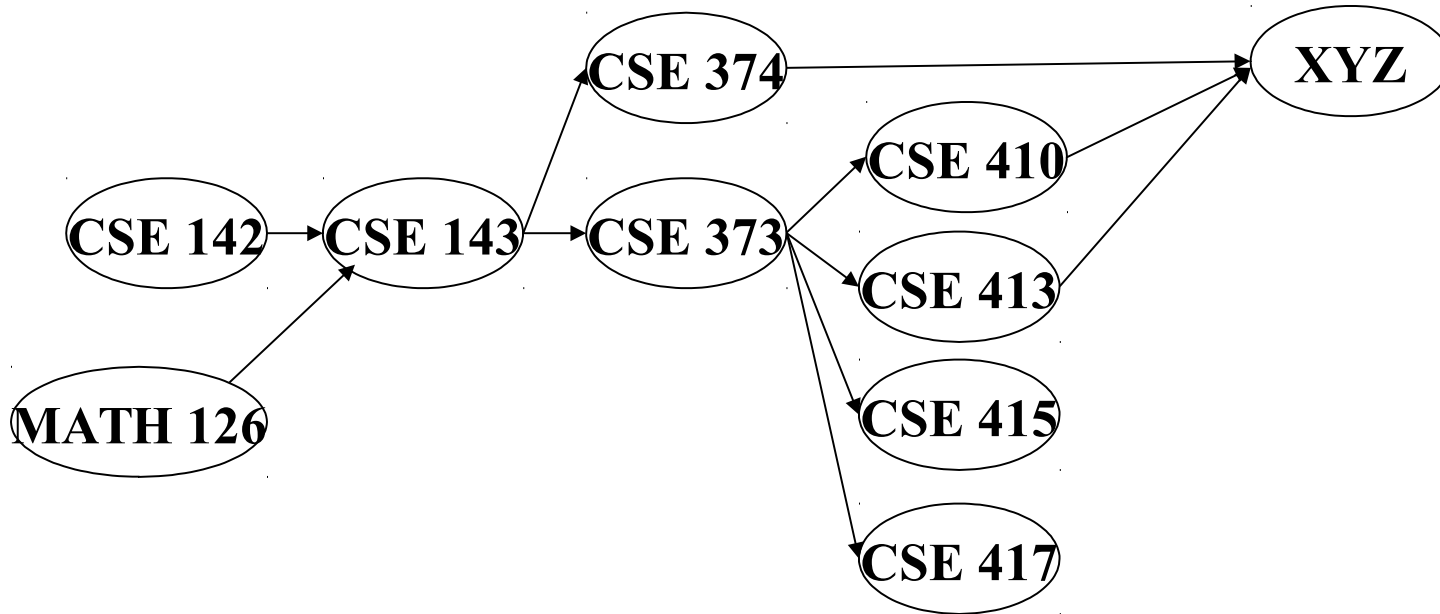
126

142

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x								
In-degree:	0	0	<del>2</del>	1	1	1	1	1	1	3
			<del>1</del>							
			0							



## Example



Output:

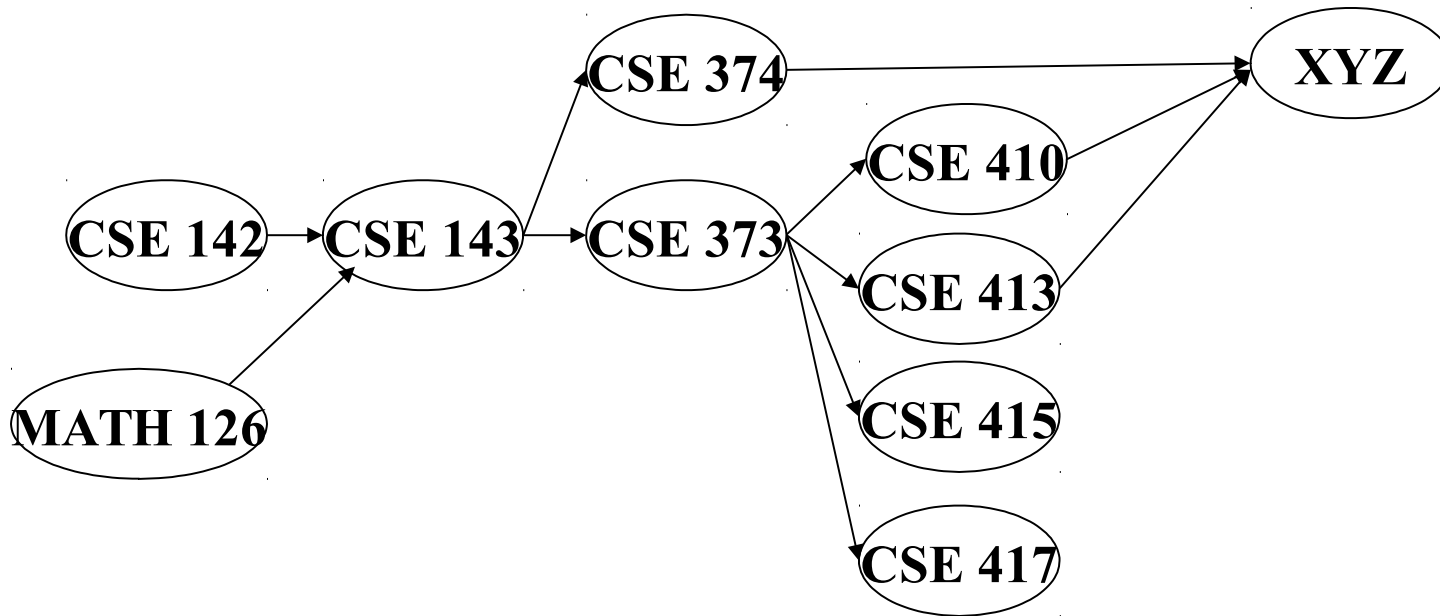
126

142

143

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x							
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0					
			0							

## Example



Output:

126

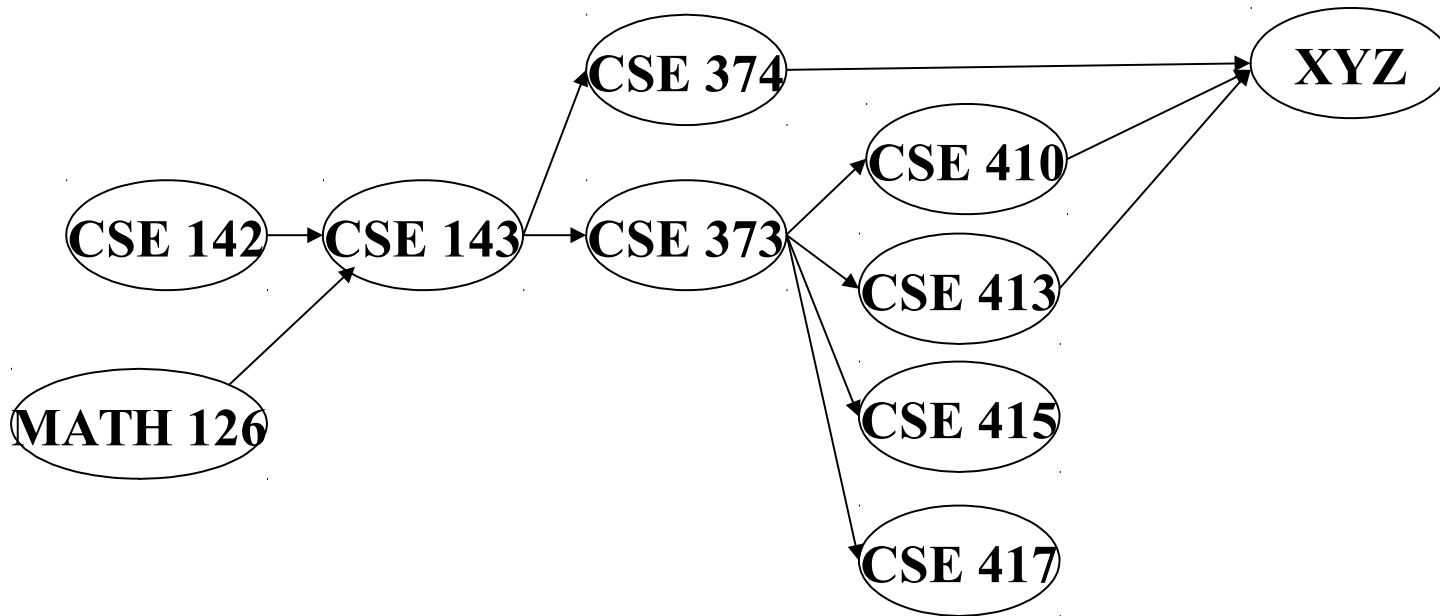
142

143

374

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x						
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	1	1	1	1	<del>3</del>
			<del>1</del>	0	0					2
			0							

## Example



Output:

126

142

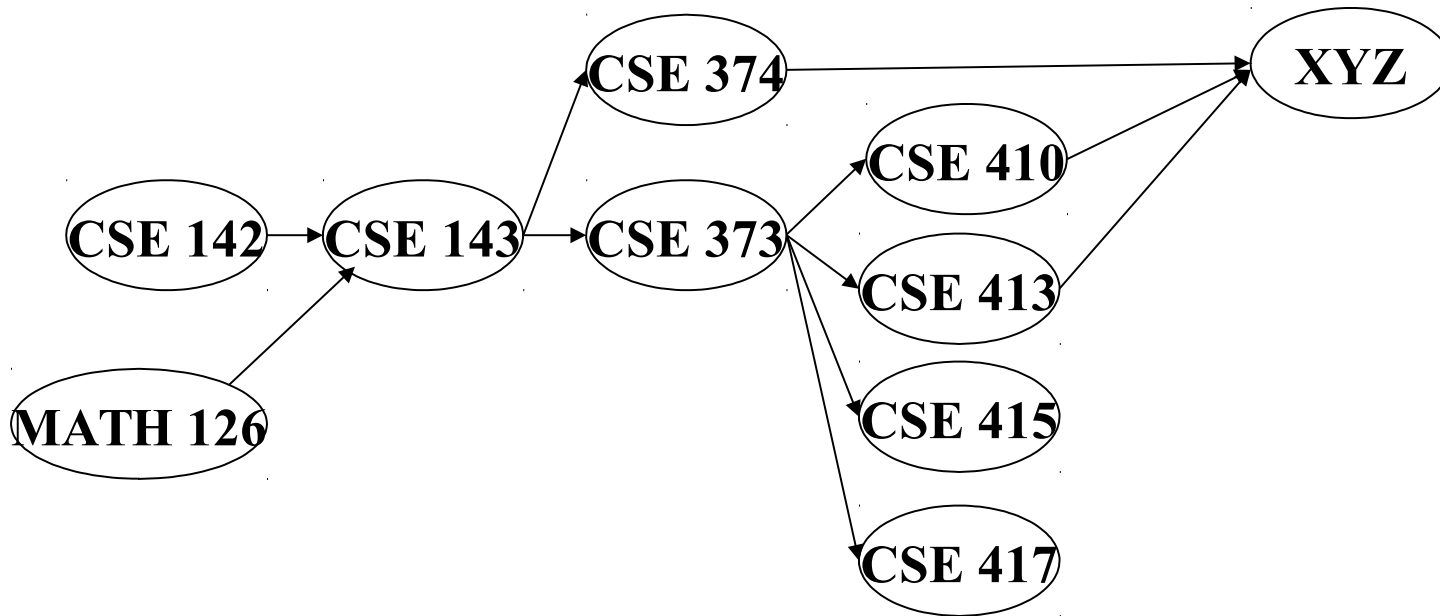
143

374

373

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x					
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	2
			0							

## Example



Output:

126

142

143

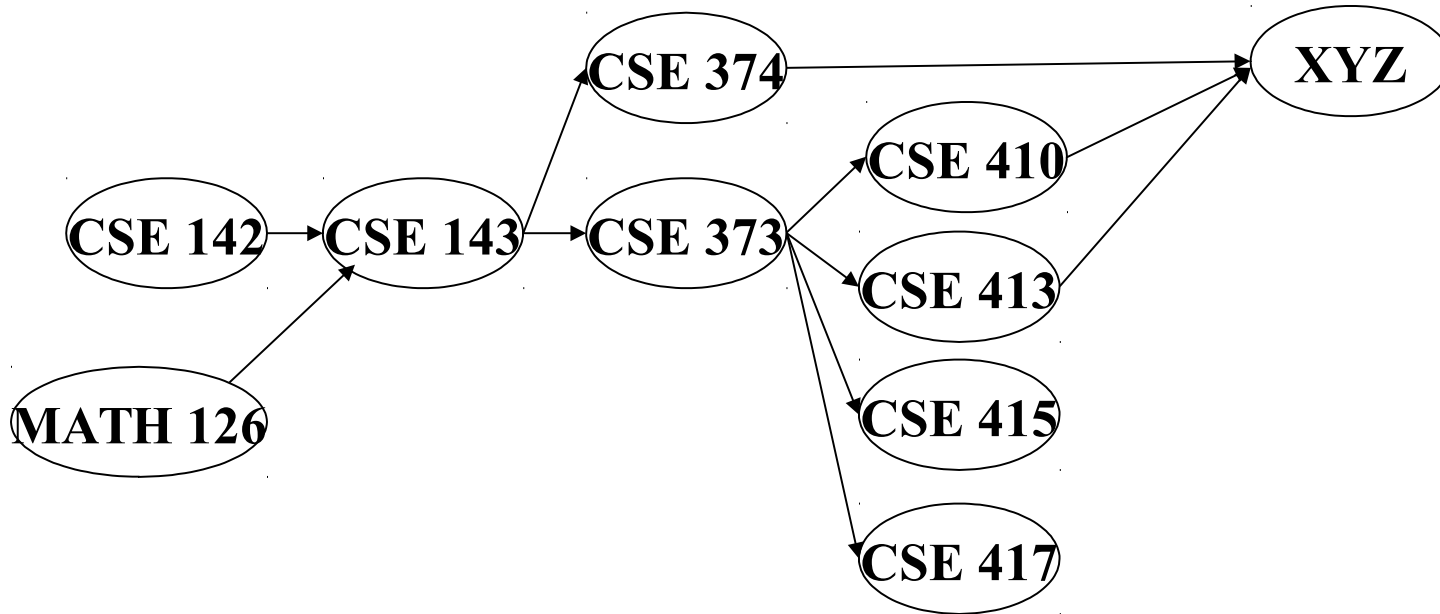
374

373

417

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x				x	
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	2
			0							

## Example

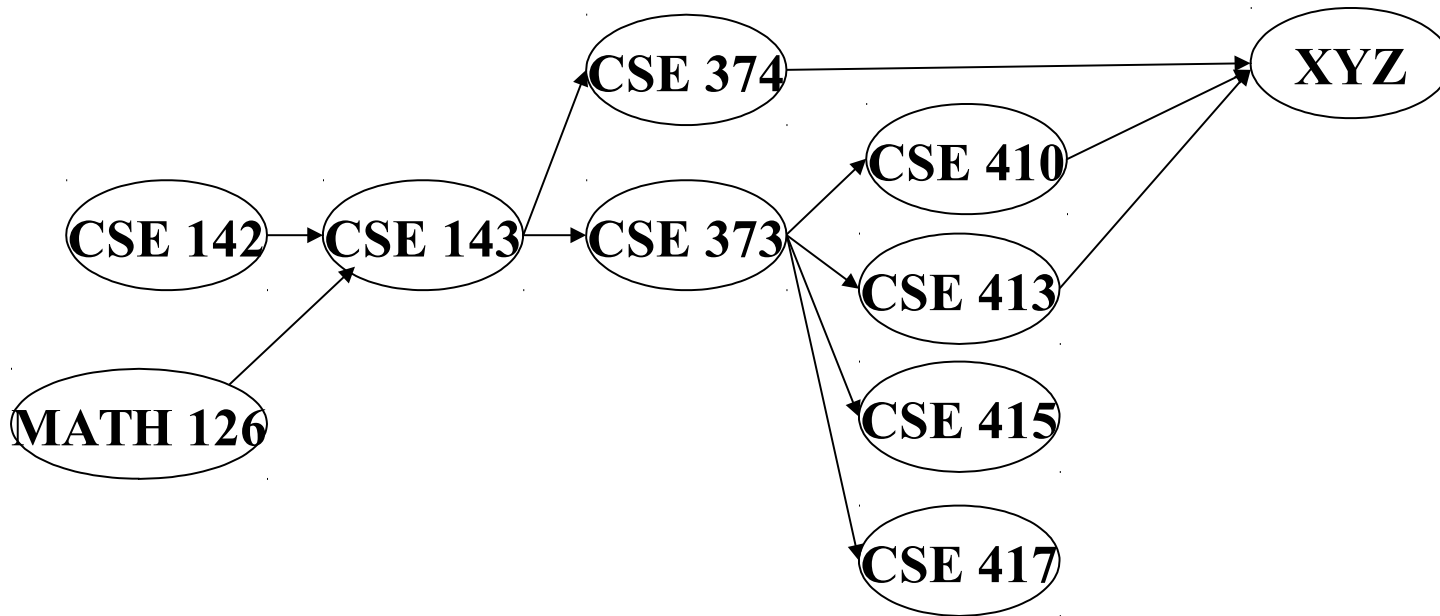


Output:

126  
142  
143  
374  
373  
417  
**410**

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	<b>x</b>			x	
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							1

## Example

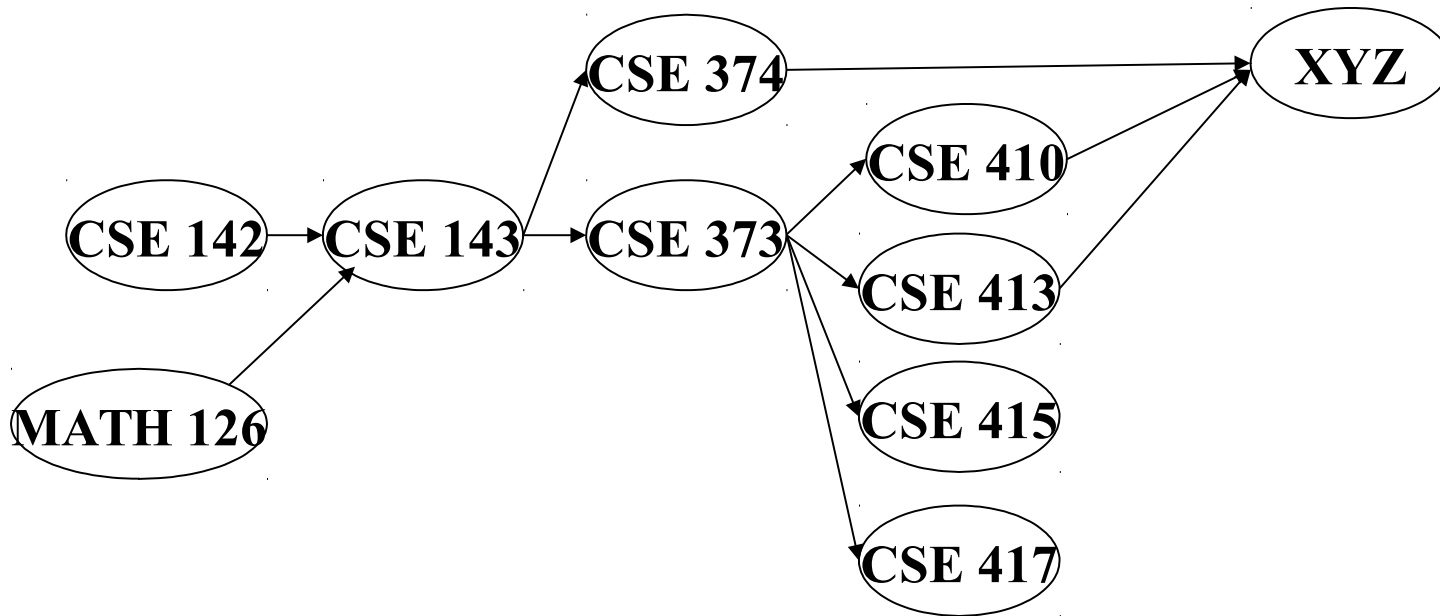


Output:

126  
142  
143  
374  
373  
417  
410  
**413**

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							<del>1</del>
										0

## Example



Output:

126

142

143

374

373

417

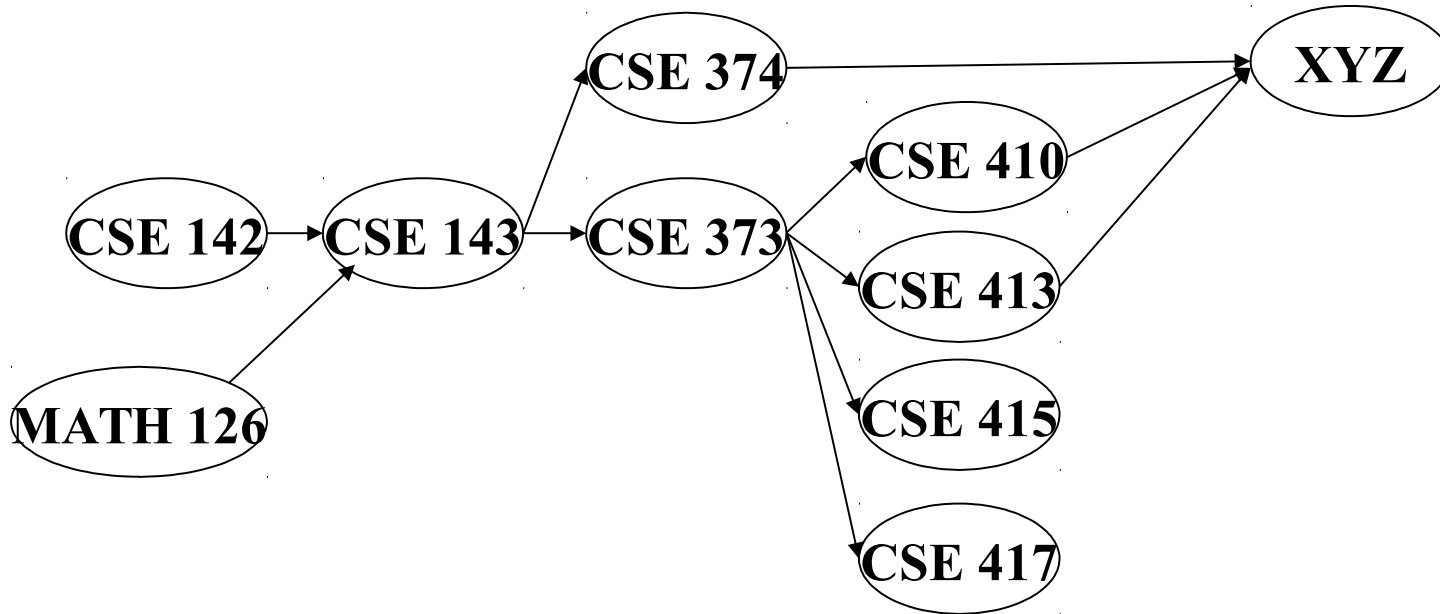
410

413

XYZ

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	x
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							<del>1</del>
										0

## Example



Output:

126

142

143

374

373

417

410

413

XYZ

415

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x	x	x	x
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							<del>1</del>
										0



## *Notice*

- Needed a vertex with in-degree 0 to start
  - Will always have at least 1 because no cycles
- Ties among vertices with in-degrees of 0 can be broken arbitrarily
  - Can be more than one correct answer, by definition, depending on the graph

## *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  $O(|V|^2)$  – 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, 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)  $\mathbf{v} = \text{dequeue}()$
  - b) Output  $\mathbf{v}$  and remove it from the graph
  - c) For each vertex  $\mathbf{u}$  adjacent to  $\mathbf{v}$  (i.e.  $\mathbf{u}$  such that  $(\mathbf{v}, \mathbf{u})$  in  $\mathbf{E}$ ), decrement the in-degree of  $\mathbf{u}$ , if new degree is 0, enqueue it

## *Running time?*

```
labelAllAndEnqueueZeros();  
while queue not empty {  
    v = dequeue();  
    put v next in output  
    for each w adjacent to v {  
        w.indegree--;  
        if (w.indegree==0)  
            enqueue(v);  
    }  
}
```

## Running time?

```
labelAllAndEnqueueZeros();  
while queue not empty {  
    v = dequeue();  
    put v next in output  
    for each w adjacent to v {  
        w.indegree--;  
        if (w.indegree==0)  
            enqueue(v);  
    }  
}
```

- 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!

# Graph Traversals

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

- Possibly “do something” for each node
  - Examples: print to output, set a field, etc.
- **Subsumed problem:** Is an undirected graph connected?
  - **Related but different problem:** Is a directed graph strongly connected?
    - Need cycles 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*

```
void 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)  
            }  
    }  
}
```



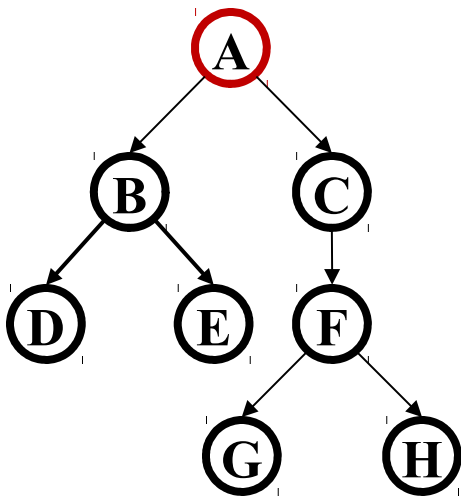
# *Running Time and Options*

- 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

Cool visualization: <http://visualgo.net/dfsbf.html>

## Example: trees

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

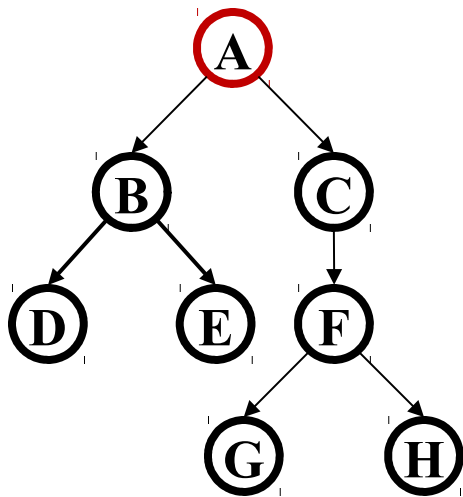


```
DFS(Node start) {  
    mark and process start  
    for each node u adjacent to start  
        if u is not marked  
            DFS(u)  
}
```

- 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

## Example: trees

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

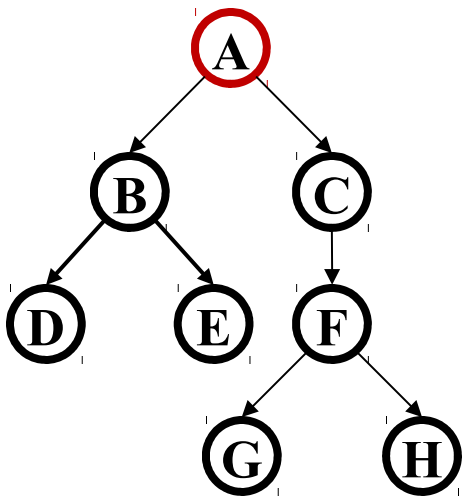


```
DFS2(Node start) {  
    initialize stack s to hold start  
    mark start as visited  
    while(s is not empty) {  
        next = s.pop() // and “process”  
        for each node u adjacent to next  
            if(u is not marked)  
                mark u and push onto s  
    }  
}
```

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

## Example: trees

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



```
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  
    }  
}
```

- 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  $\mathbf{x}$  to  $\mathbf{y}$ ”
- But depth-first can use less space in finding a path
  - If *longest path* in the graph is  $\mathbf{p}$  and highest out-degree is  $\mathbf{d}$  then DFS stack never has more than  $\mathbf{d} \cdot \mathbf{p}$  elements
  - But a queue for BFS may hold  $O(|V|)$  nodes
- A third approach:
  - *Iterative deepening (IDFS)*:
    - Try DFS but disallow recursion more than  $\mathbf{k}$  levels deep
    - If that fails, increment  $\mathbf{k}$  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?
  - Like getting driving directions rather than just knowing it’s possible to get there!
- How to do it:
  - 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)
  - If just wanted path *length*, could put the integer distance at each node instead

## Example using BFS

What is a path from Seattle to Tyler

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

