GRAPHS

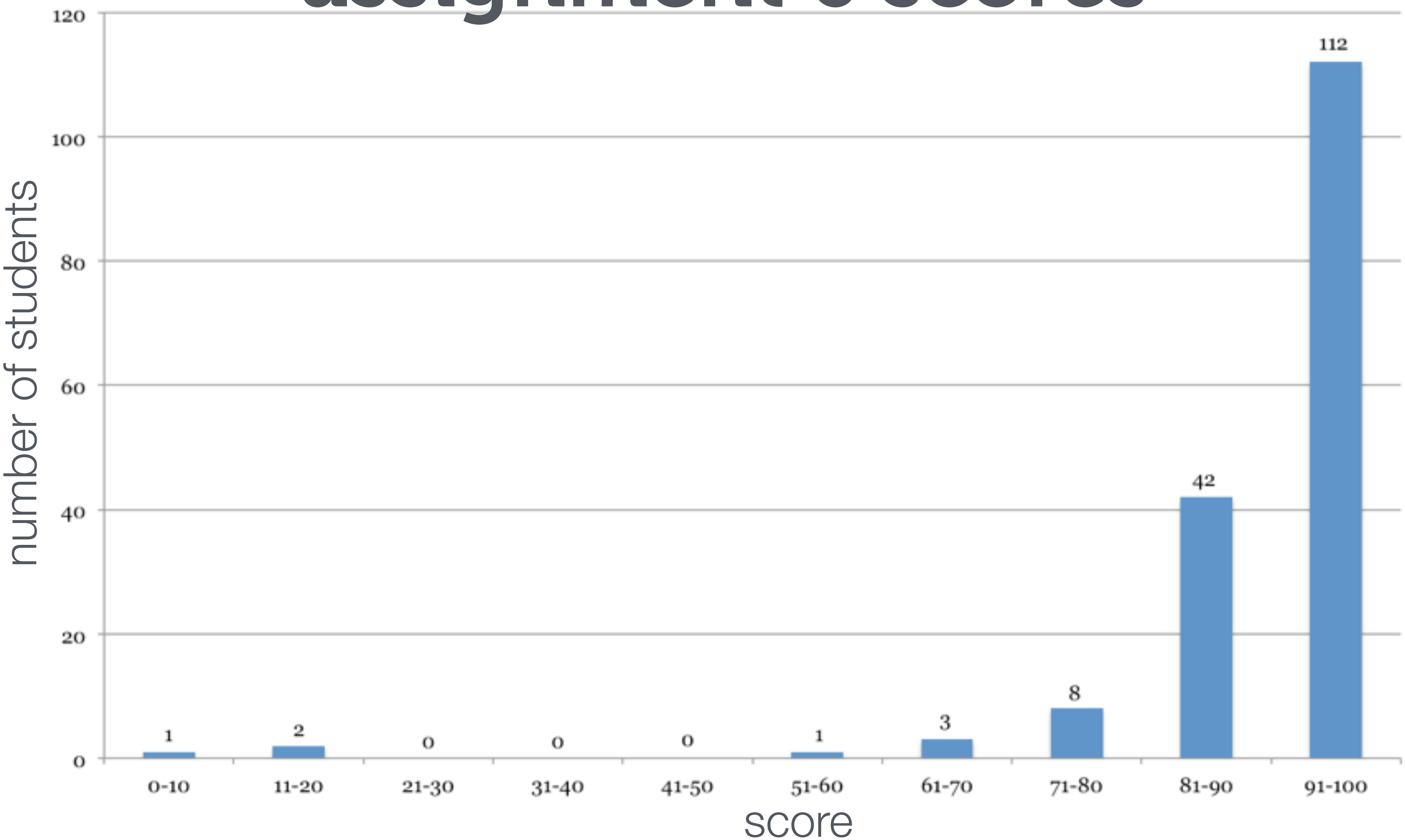
cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

administrivia…

-assignment 8 due Thursday

-assignment 9 out tomorrow… due in 2.5 weeks

assignment 6 scores



last time…

binary search trees (BSTs)

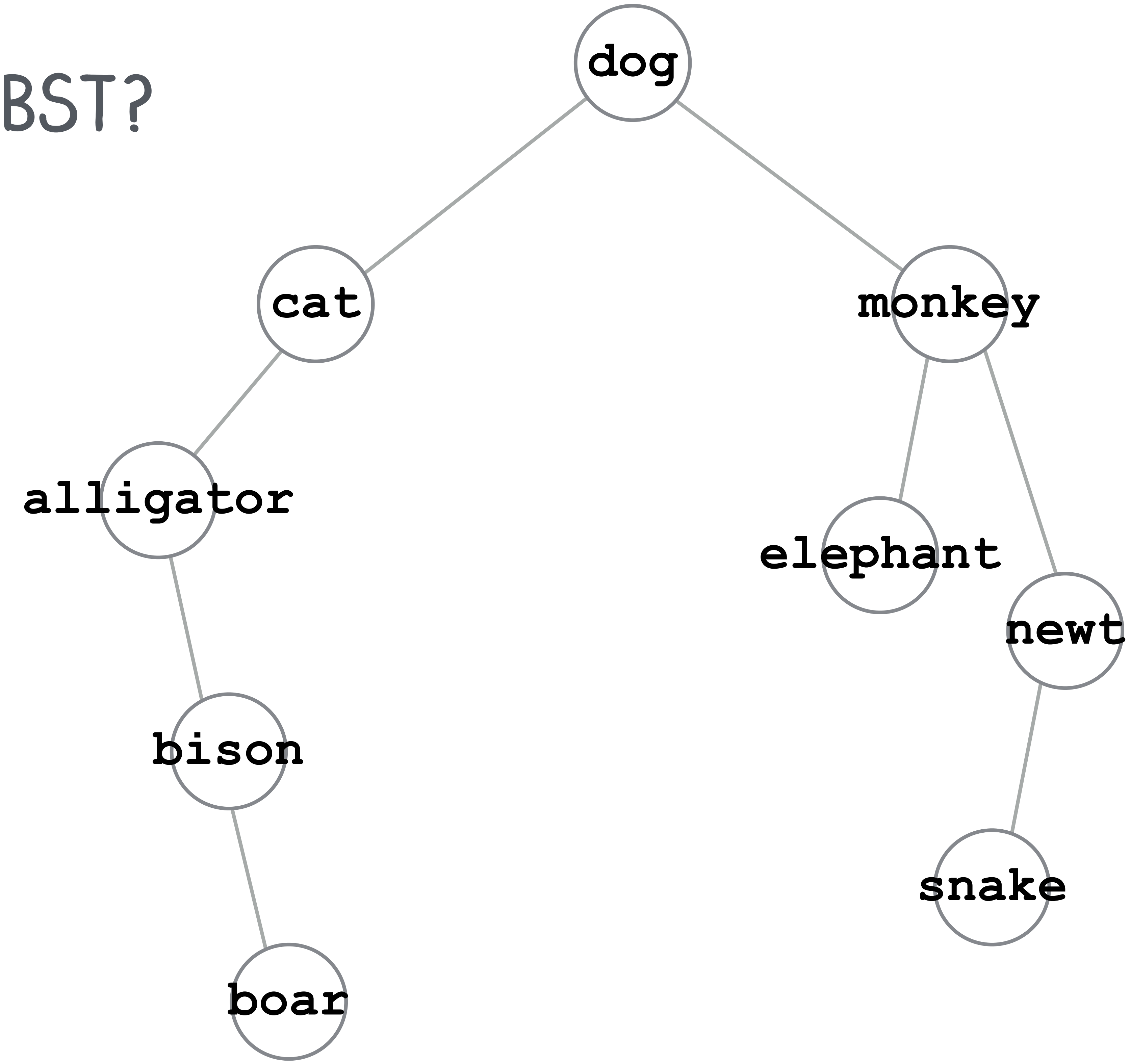
-a **binary search tree** is a binary tree with a restriction on the ordering of nodes

- -all items in the **left** subtree of a node are *less than* the item in the node
- -all items in the **right** subtree of a node are *greater than or equal to* the item in the node

-BSTs allow for fast searching of nodes

IS THIS A BST?

A) **yes**
B) **no**



insertion

insertion & searching

-average case: O(log N)

-inserted in random order

-worst case:O(N)

-inserted in ascending or descending order

-best case: O(log N)

-how does this compare to a sorted array?

deletion

-since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

-there are three different cases:

1. deleting a leaf node
2. deleting a node with one child subtree
3. deleting a node with two children subtrees

-first step of deletion is to find the node to delete

-just a regular BST search

-BUT, stop at the *parent* of the node to be deleted

deletion performance

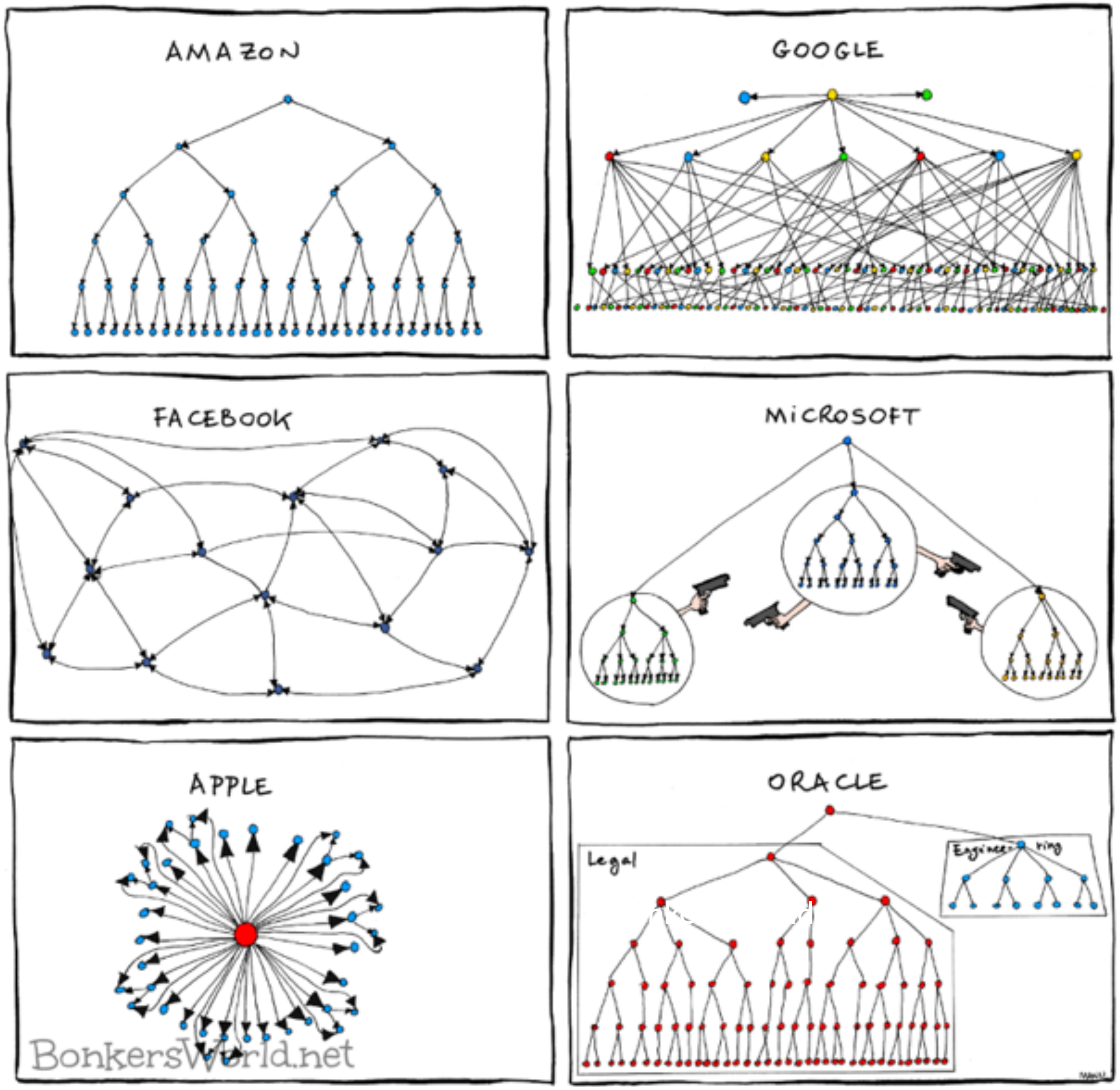
WHAT IS THE COST OF DELETING A NODE FROM A BST?

-first, find the node we want to delete: **O(log N)**

-cost of:

- -case 1 (delete leaf):
  - SET A SINGLE REFERENCE TO NULL: **O(1)**
- -case 2 (delete node with 1 child):
  - BYPASS A REFERENCE: **O(1)**
- -case 3 (delete node with 2 children):
  - FIND THE SUCCESSOR: **O(log N)**
  - DELETE THE DUPLICATE SUCCESSOR: **O(1)**

today…


AMAZON
GOOGLE
FACEBOOK
MICROSOFT
APPLE
ORACLE
Legal
Engineering
BonkersWorld.net


facebook


December 2010
Paul Butler

-graphs

-paths

-depth-first search

-breadth-first search

graphs

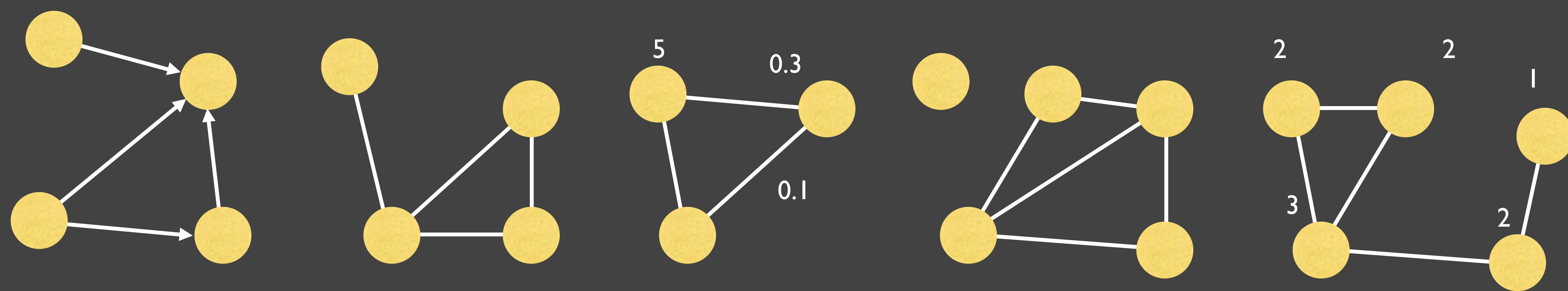
-trees are a *subset* of graphs

-a **graph** is a set of *nodes* connected by **edges**
- -an edge is just a link between two nodes
- -nodes don’t have a parent-child relationship
- -links can be bi-directional

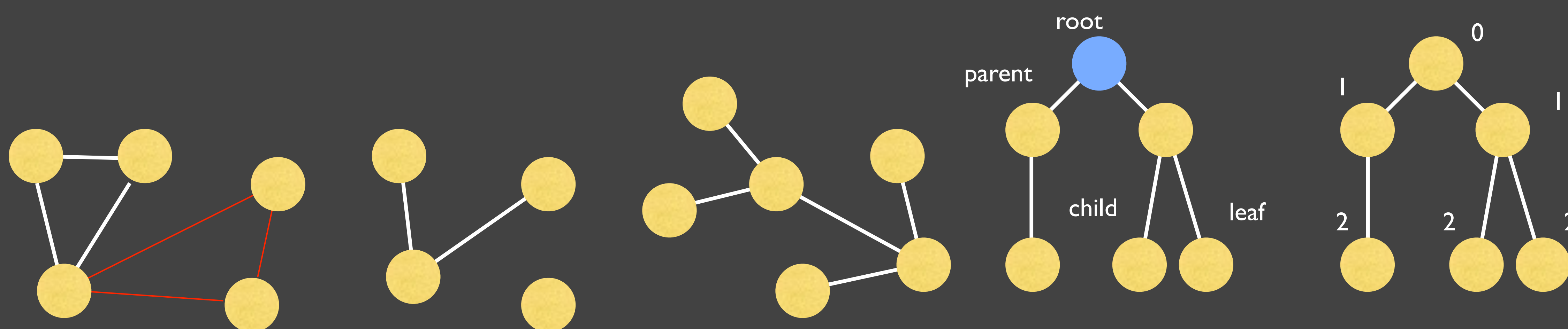
-graphs are used **EXTENSIVELY** throughout CS

NODES ARE CITIES, EDGES ARE FLIGHTS

some definitions

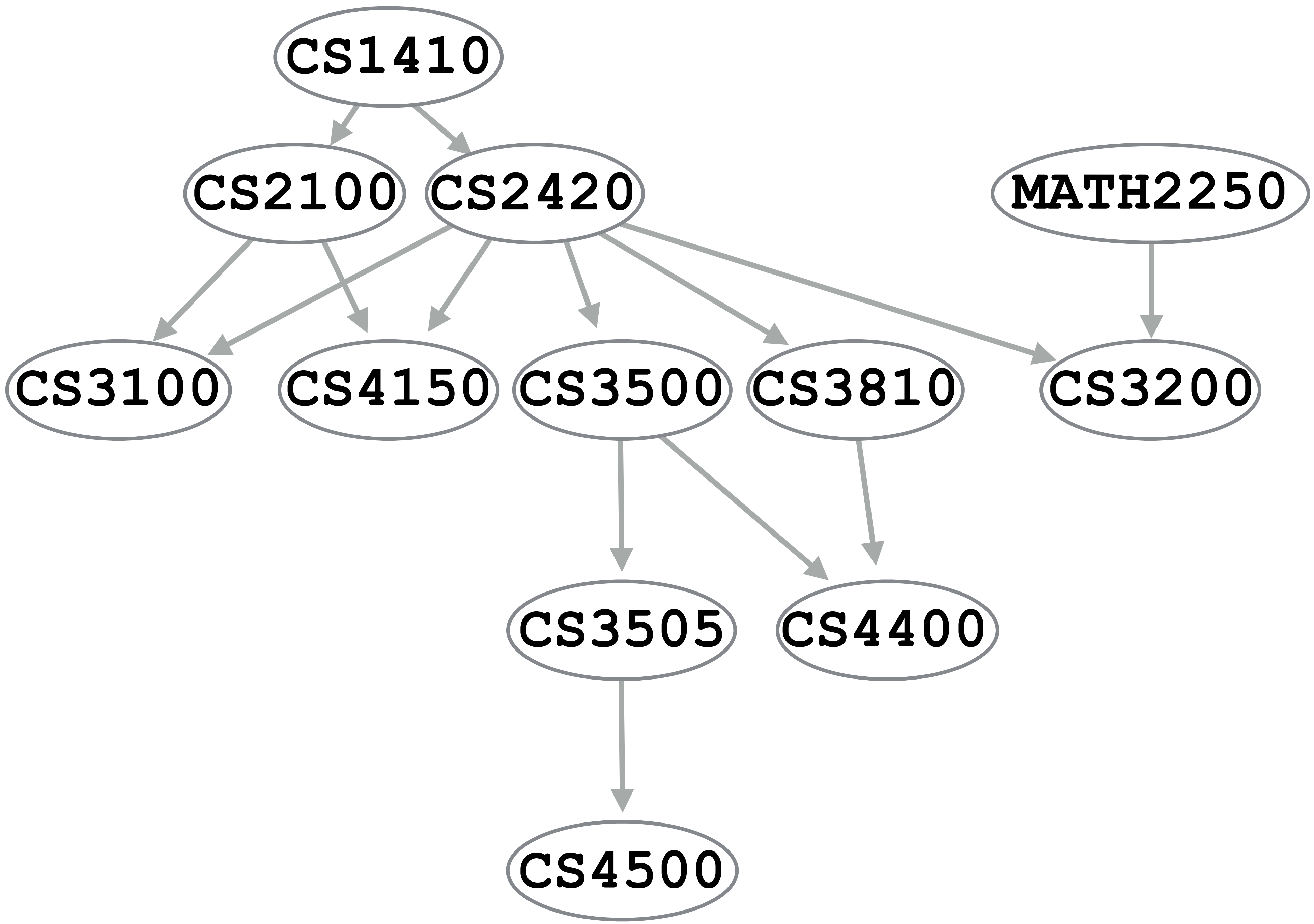


A directed graph — An undirected graph — Weighted — Unconnected — Node degrees



A cycle — An acyclic graph — A connected acyclic graph, a.k.a. a **tree** — A rooted tree or hierarchy — Node depths

WHAT MAKES THIS A GRAPH AND NOT A TREE?



-graphs have no root; must store all nodes

```
class Graph<E> {
  List<Node> nodes;
  …
}
```

-implementation is more general than a tree

```
class Node{
  E Data;
  List<Node> neighbors;
  …
}
```

-the order in which neighbors appear in the list is unspecified

-a different order still make the same graph!

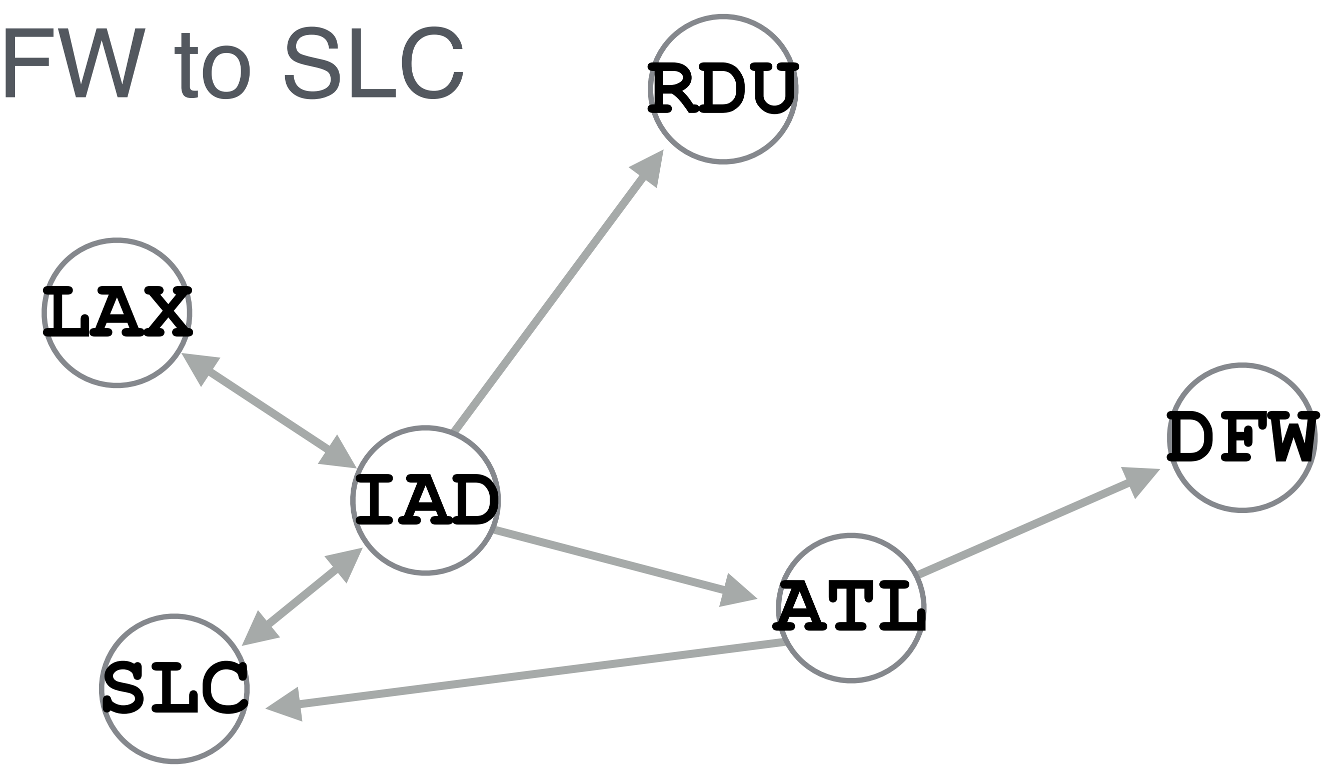
paths

-a **path** is a sequence of nodes with a start-point and an end-point such that the end-point can be reached through a series of nodes from the start-point

-in this example, there is a path from SLC to DFW

- -SLC — IAD — ATL — DFW

-there is *not* a path from DFW to SLC

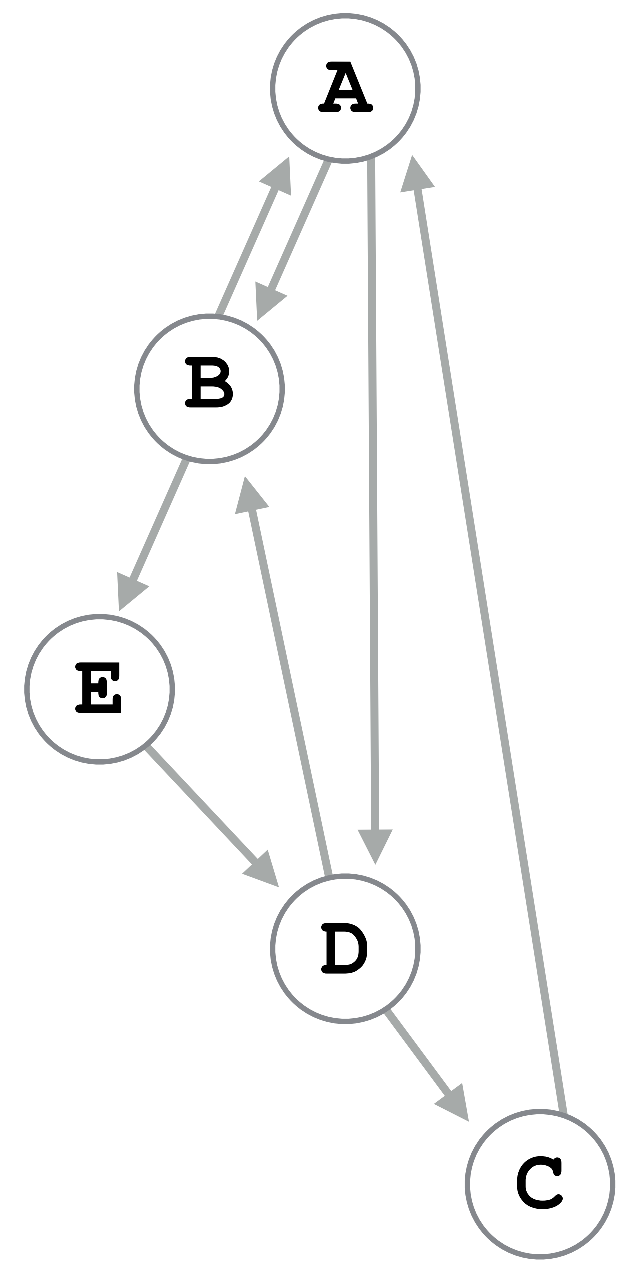


pathfinding

-there may be more than one path from one node to another

-we are often interested in the *path length*

-finding the shortest (or cheapest) path between two nodes is a common graph operation



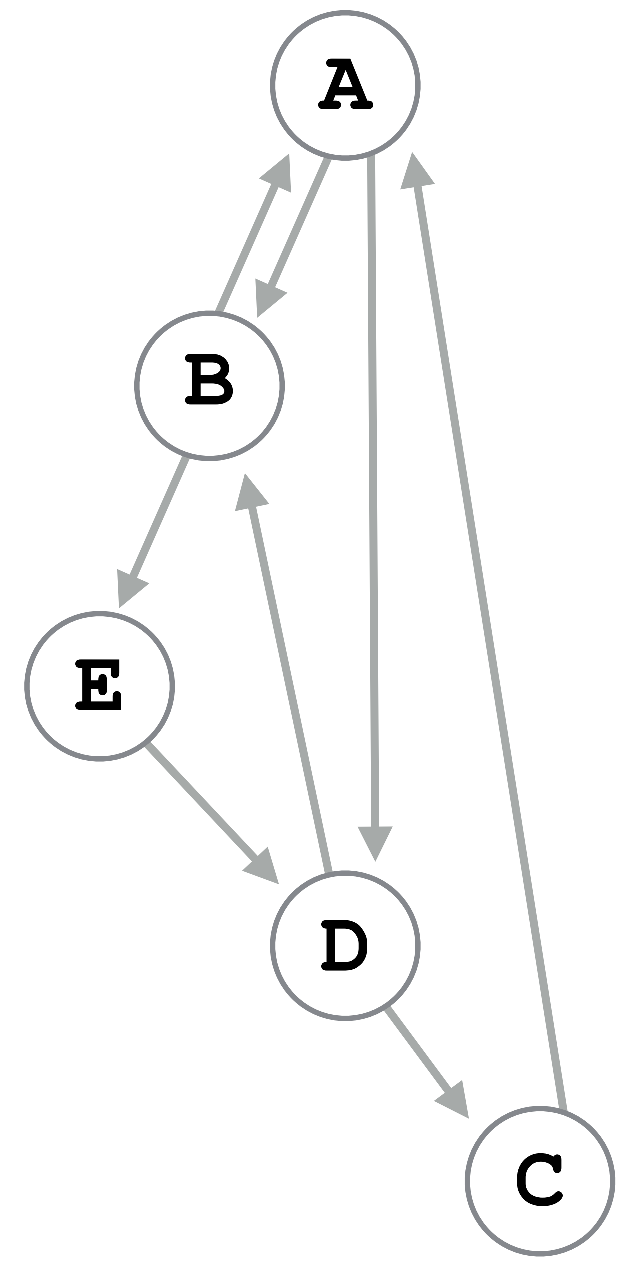
cycles

-a cycle in a graph is a path from a node back to itself

- -B — E — D — B

-while traversing a graph, special care must be taken to avoid cycles, otherwise what?

-can trees have cycles?



-any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph

-and solved with pathfinding!

example

-in games, moving a character around a space

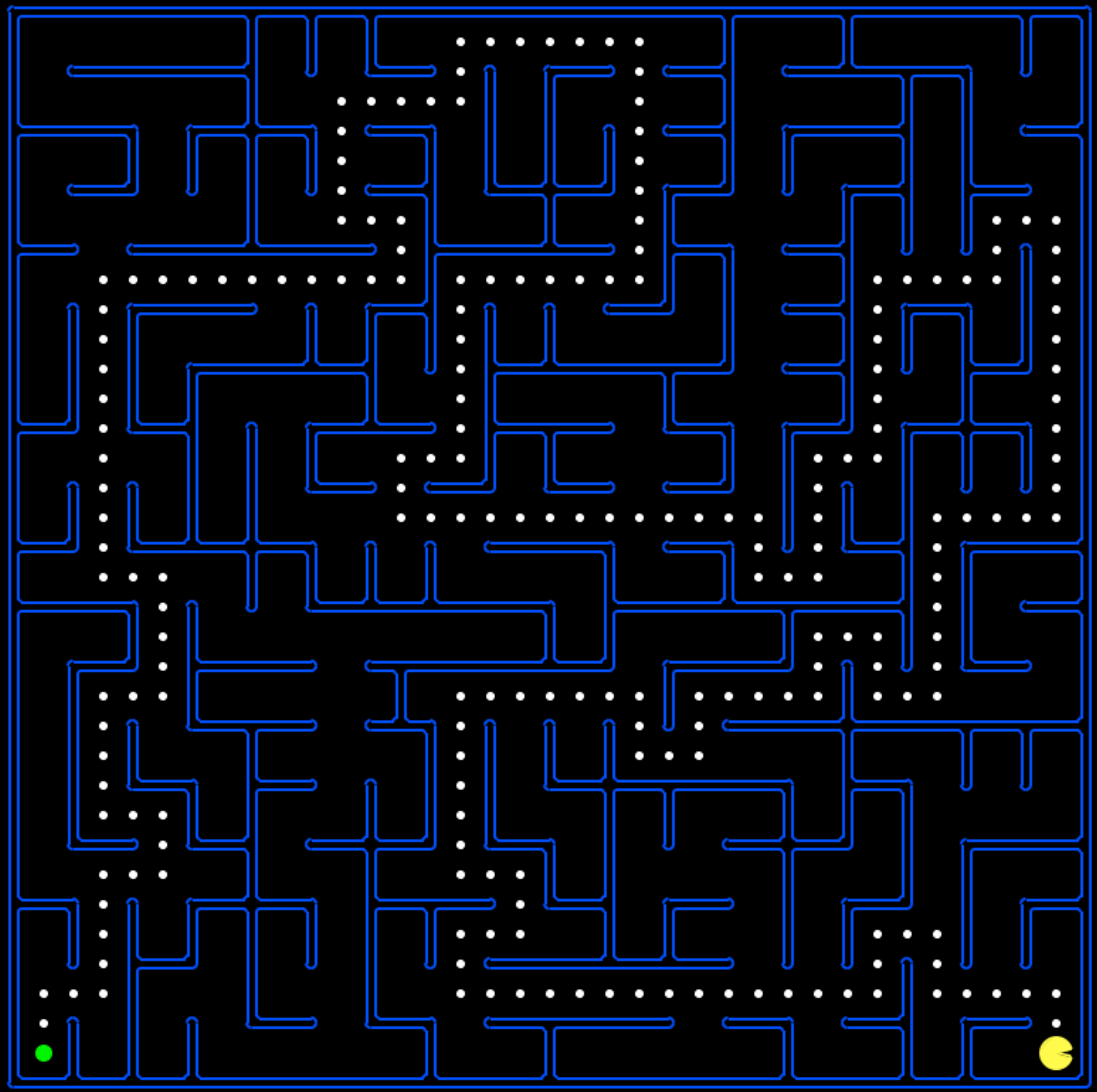
-character finds the shortest path from its current location to the destination

- -not always a straight line

-terrain is represented as a graph

- -every non-obstacle spot on the terrain is a node
- -nodes are connected to adjacent nodes

-navigating a maze…



-depth-first search (just like a tree) — DFS

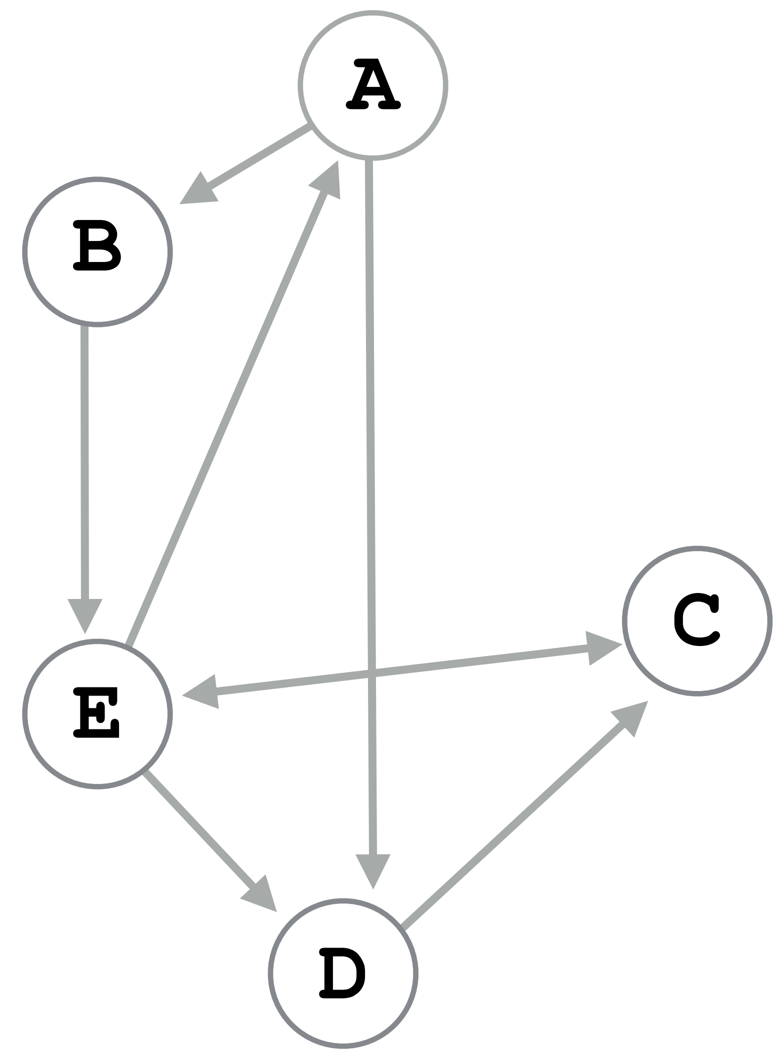
-breadth-first search — BFS

-if there exists a path from one node to another these algorithms will find it

   -the nodes on this path are the steps to take to get from point A to point B

-if multiple such paths exist, the algorithms may find different ones

WE WANT TO FIND A PATH FROM **A** TO **C**



depth-first search

-look at the first edge going out of the start node
-recursively search from the new node
-upon returning, take the next edge
-if no more edges, return

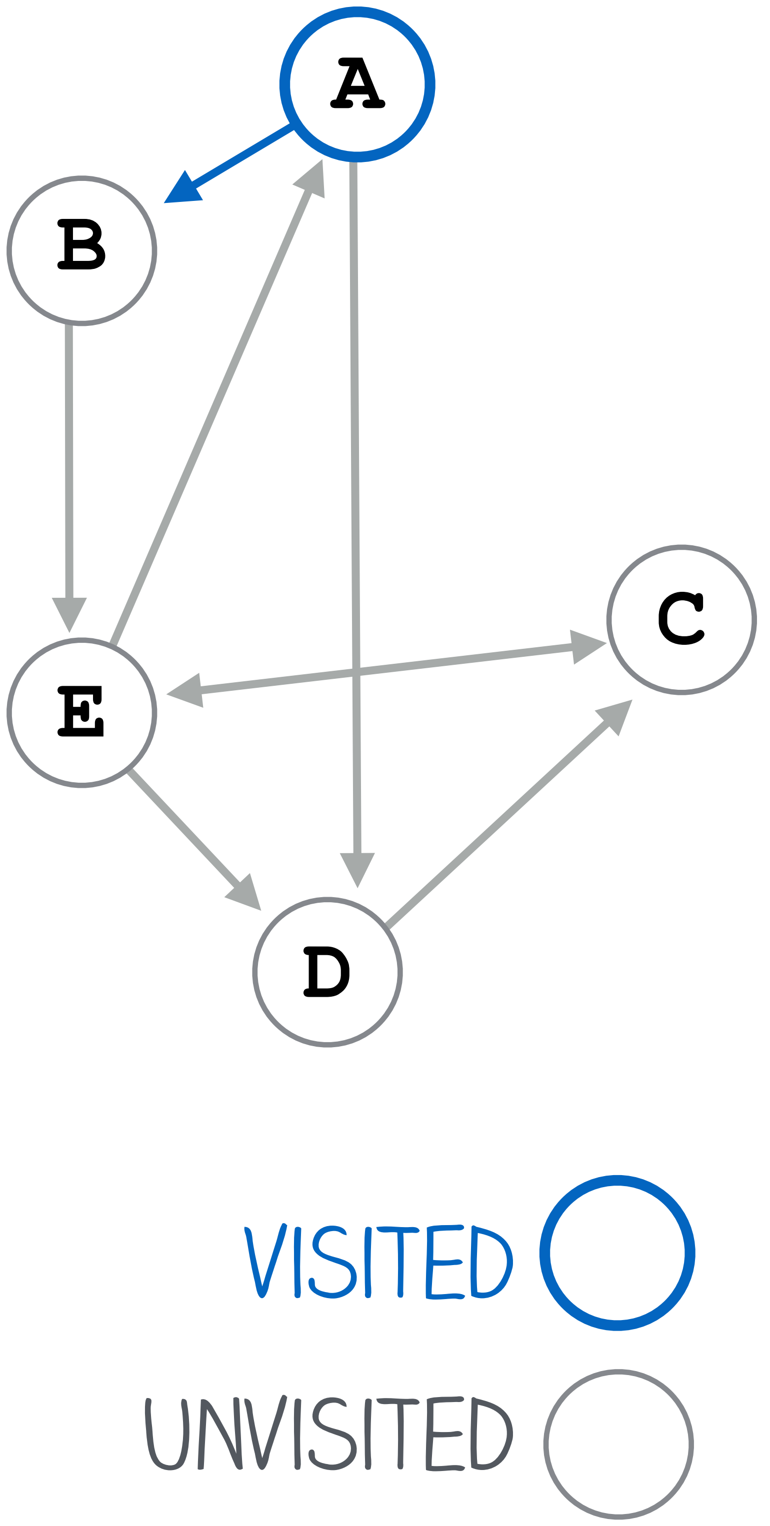
-when visiting a node, mark it as visited so we don't get stuck in a cycle
    -skip already visited nodes during traversal

-for each node visited, save a reference to the node where we came from to reconstruct the path

WE WANT TO FIND A PATH FROM **A** TO **C**

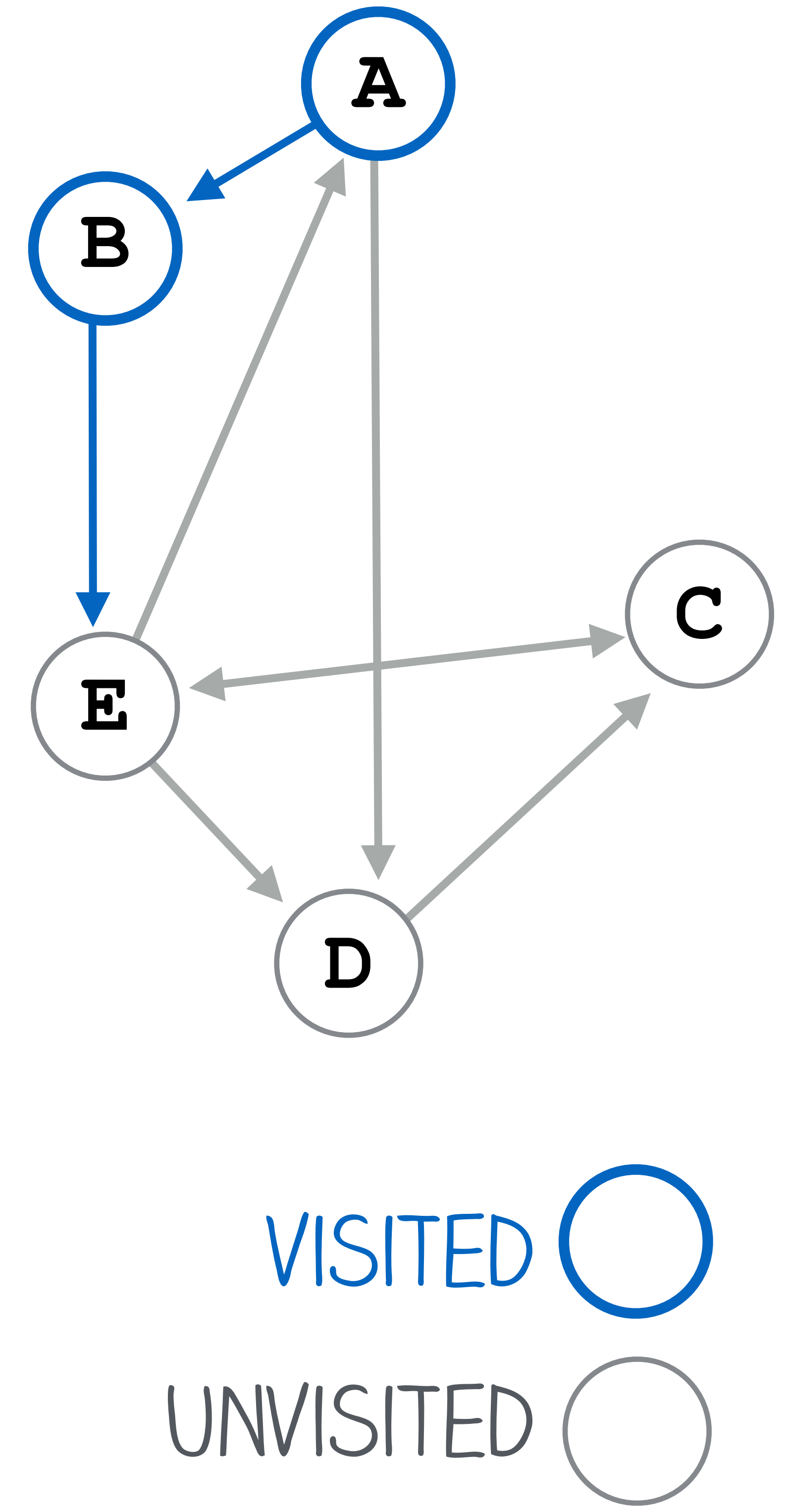
SO... START FROM **A**, TRAVERSE ITS FIRST EDGE, SAVE WHERE WE CAME FROM, AND RECURSE

```
A.visited = true
B.cameFrom = A
```



TRAVERSE THE FIRST UNVISITED NODE IN THE EDGE LIST RECURSIVELY, SAVE WHERE WE CAME FROM

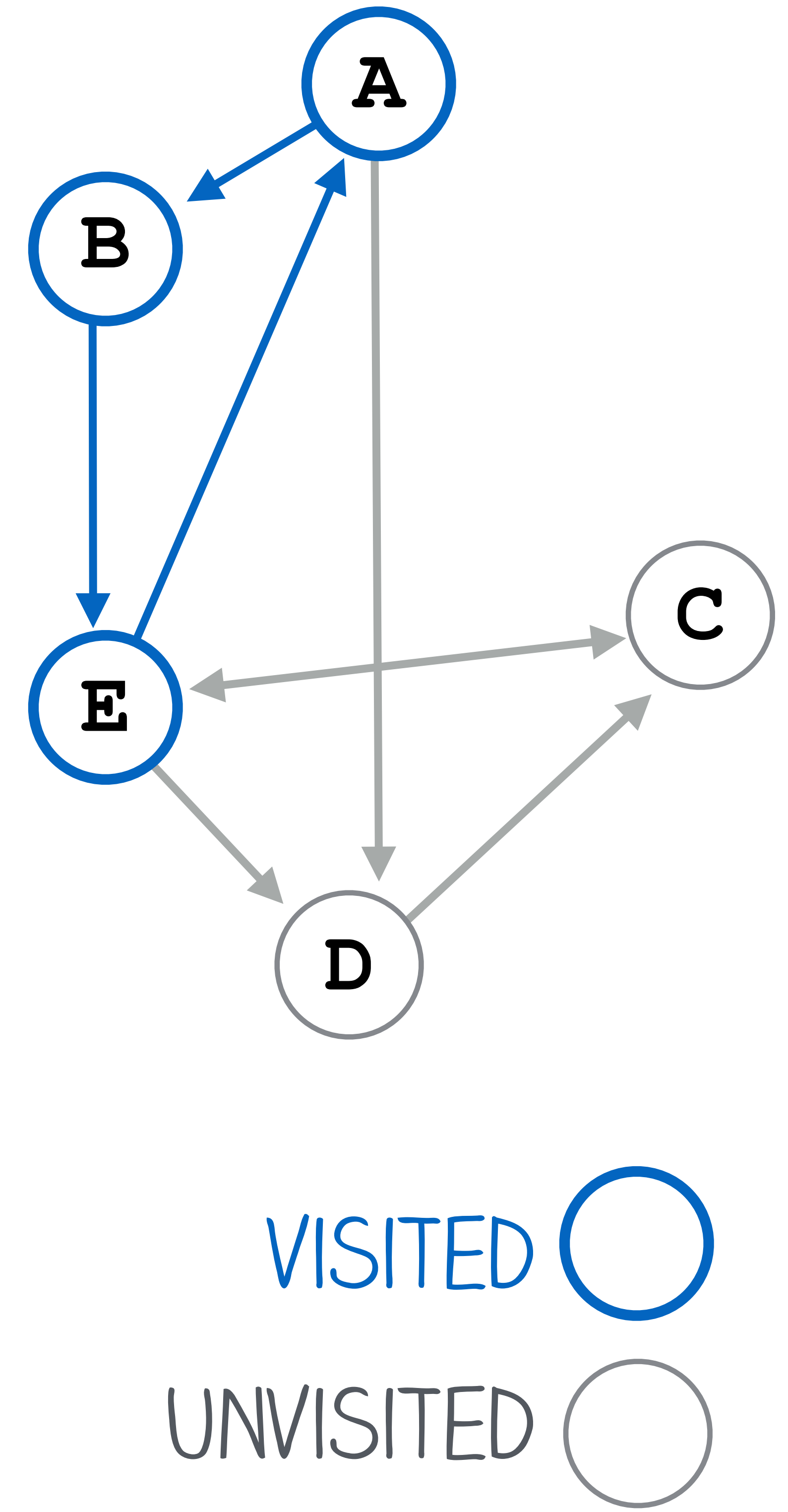
```
B.visited = true
E.cameFrom = B
```



TRAVERSE THE FIRST UNVISITED NODE IN THE EDGE LIST RECURSIVELY, SAVE WHERE WE CAME FROM

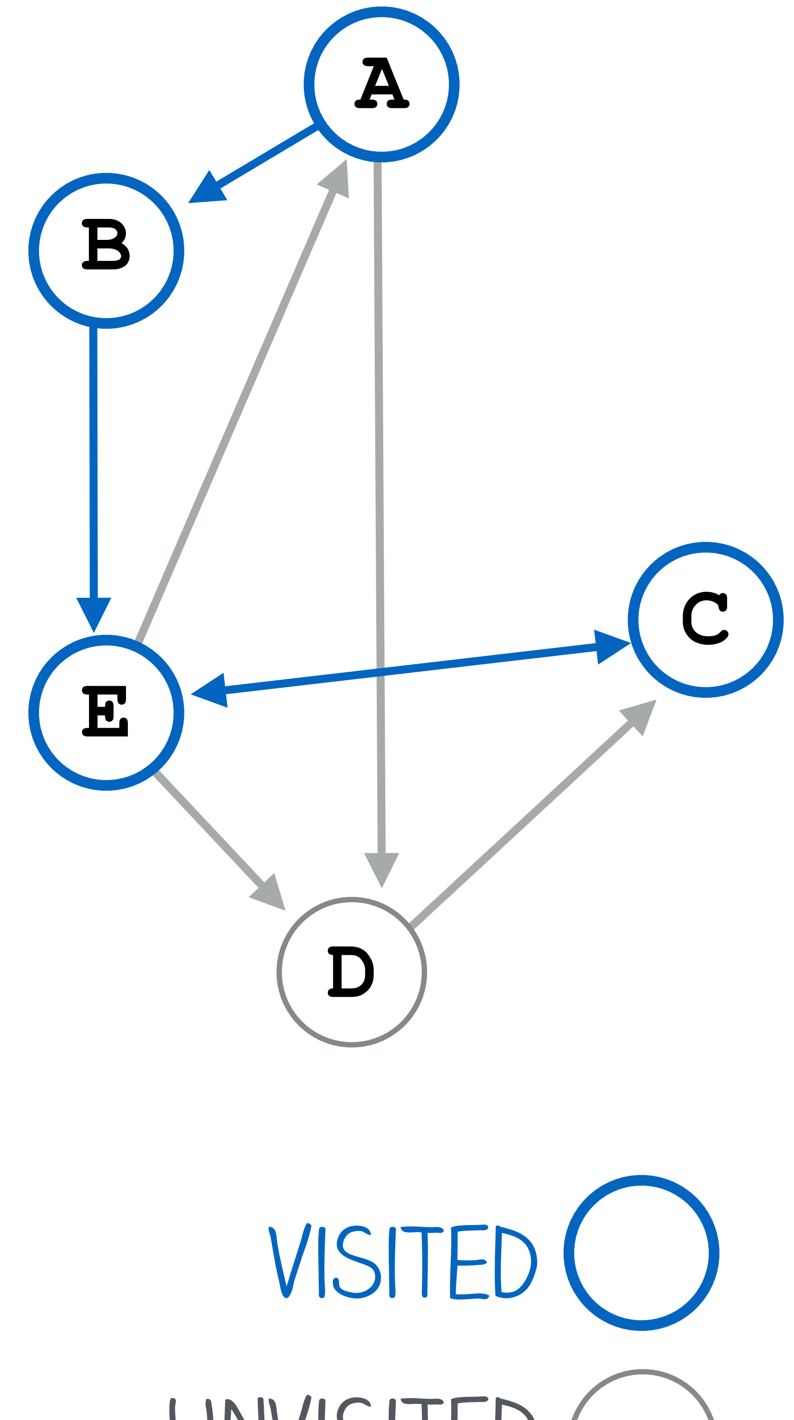
```
E.visited = true
```

LOOK AT THE FIRST EDGE; NODE **A** HAS ALREADY BEEN VISITED, SO SKIP



LOOK AT NEXT EDGE; **C** HAS NOT BEEN VISITED YET

```
C.cameFrom = E
```



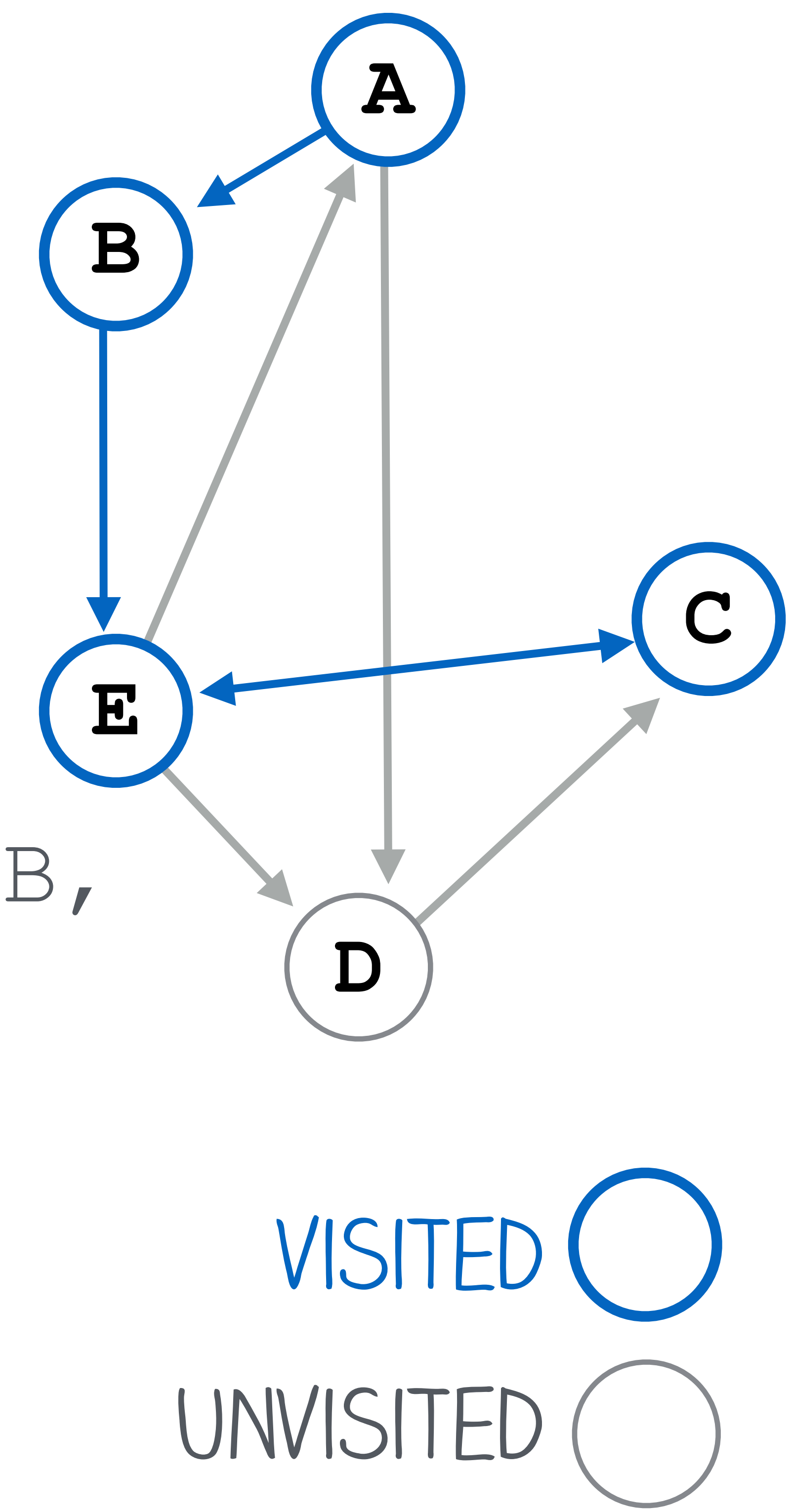
NODE **C** IS OUR GOAL. WE ARE DONE!

`C.visited = true`

FOLLOW EACH NODE'S **cameFrom** TO RECONSTRUCT THE PATH

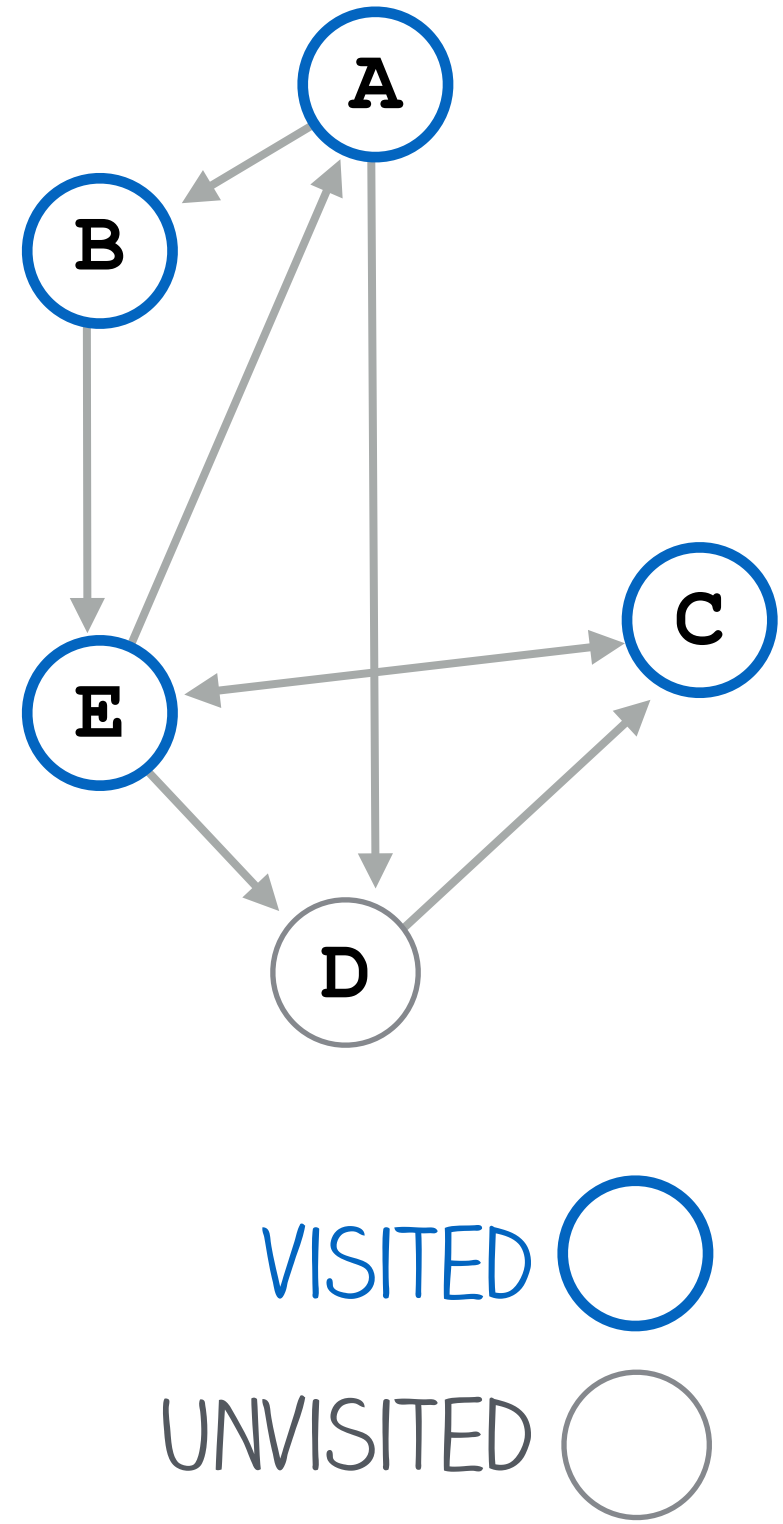
`C.cameFrom = E, E.cameFrom = B,`
`B.cameFrom = A`

PATH: **A — B — E — C**



IS THERE A BETTER (SHORTER) PATH FROM **A** TO **C**?

WHAT DETERMINES WHICH PATH DFS FINDS?



DFS IS NOT GUARANTEED TO FIND THE SHORTEST PATH, JUST A PATH.

```
DFS(Node curr, Node goal)
{
  curr.visited = true

  if(curr.equals(goal))
    return

  for(Node next : curr.neighbors)
    if(!next.visited)
    {
      next.cameFrom = curr
      DFS(next, goal)
    }
}
// path is now saved in nodes' .cameFrom
```

breadth-first search

- -instead of visiting deeper nodes first, visit shallower nodes first
  - -visit nodes closets to the start point first, gradually get further away

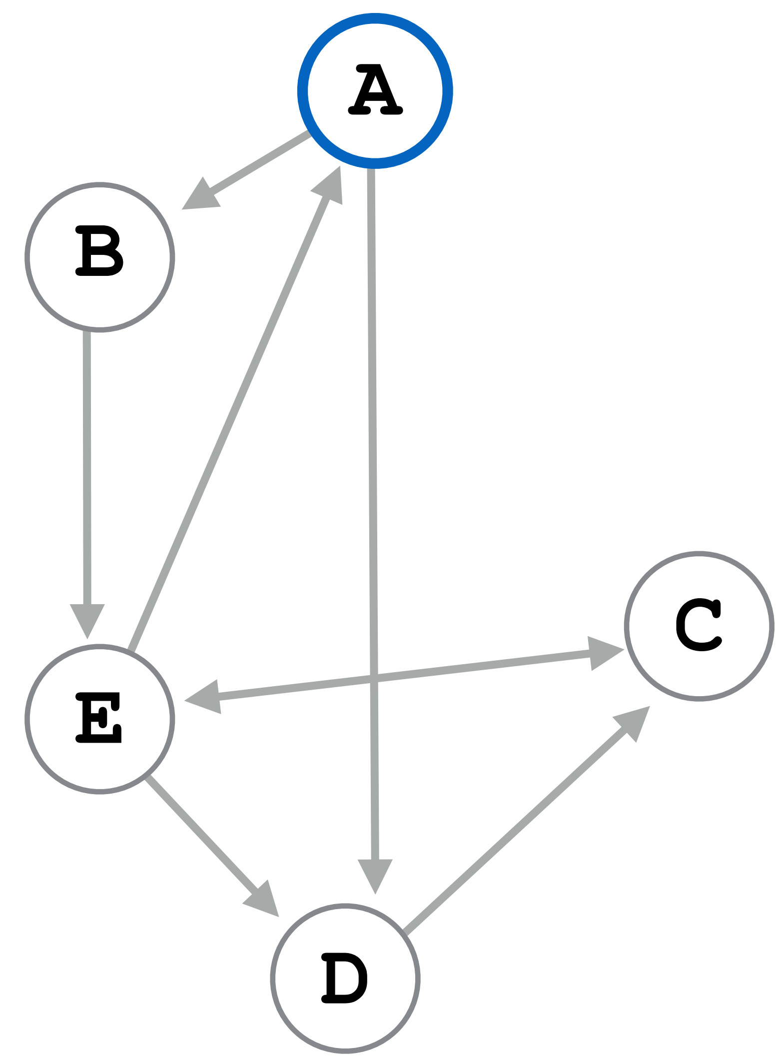
- -create an empty queue
- -put the starting node in the queue
- -while the queue is not empty
  - -dequeue the current node
  - -for each unvisited neighbor of the the current node
    - *-mark the neighbor as visited*
    - *-put the neighbor into the queue*

- -notice it is not recursive… it just runs until the queue is empty!

WE WANT TO FIND A PATH FROM **A** TO **C**

MARK AND ENQUEUE THE START NODE **A**

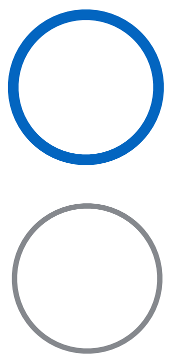
```
A.visited = true
```



queue: | A | | | | | |

VISITED

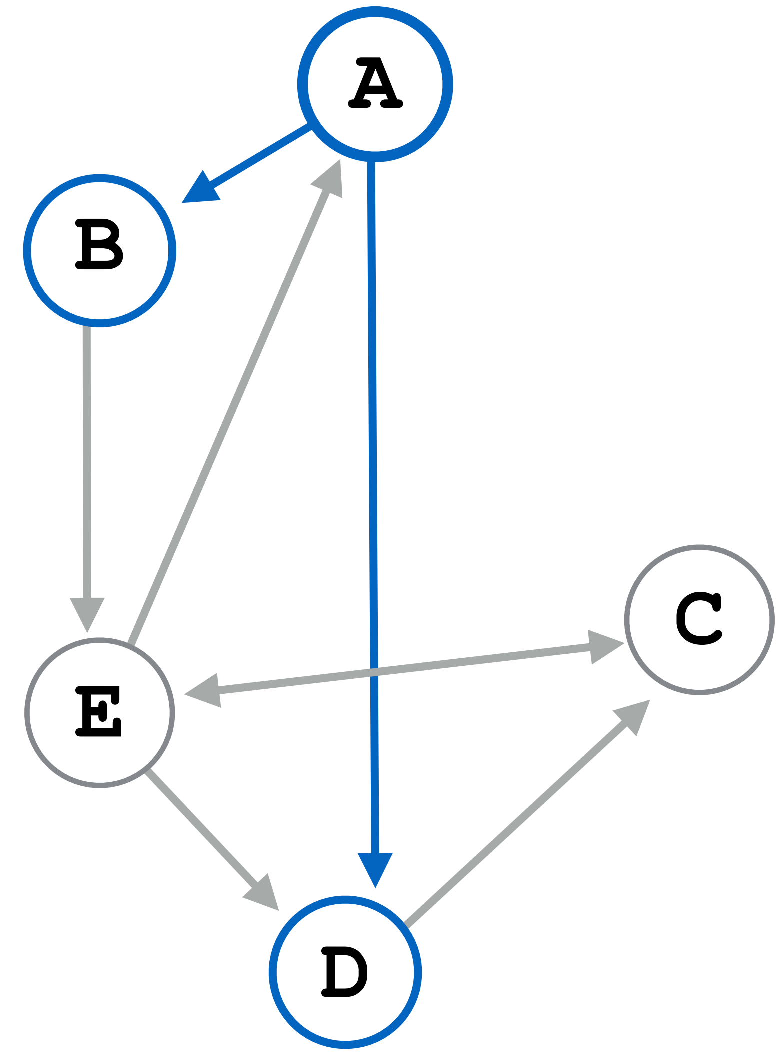
UNVISITED



DEQUEUE THE FIRST NODE IN THE QUEUE (**A**)

MARK AND ENQUEUE **A**'S UNVISITED NEIGHBORS

```
B.cameFrom = A
D.cameFrom = A
B.visited = true
D.visited = true
```



queue:

B	**D**				

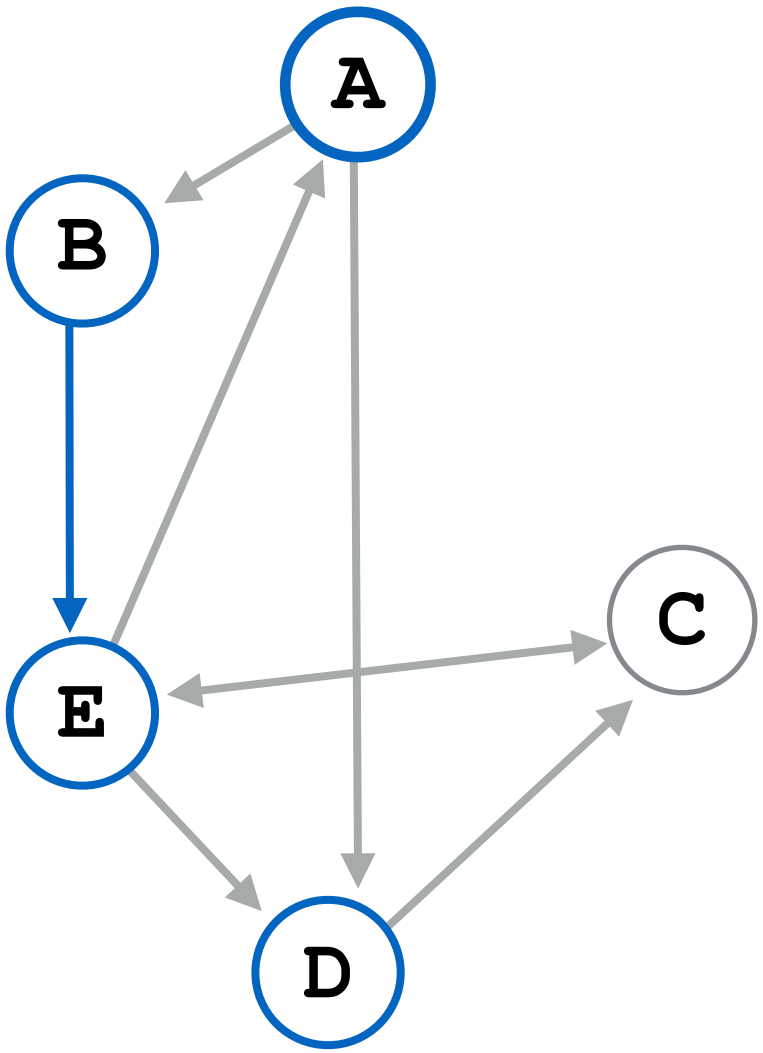
VISITED

UNVISITED

DEQUEUE THE FIRST NODE IN THE QUEUE (**B**)

MARK AND ENQUEUE **B**'S UNVISITED NEIGHBORS

```
E.cameFrom = B
E.visited = true
```



queue: | D | E | | | | |

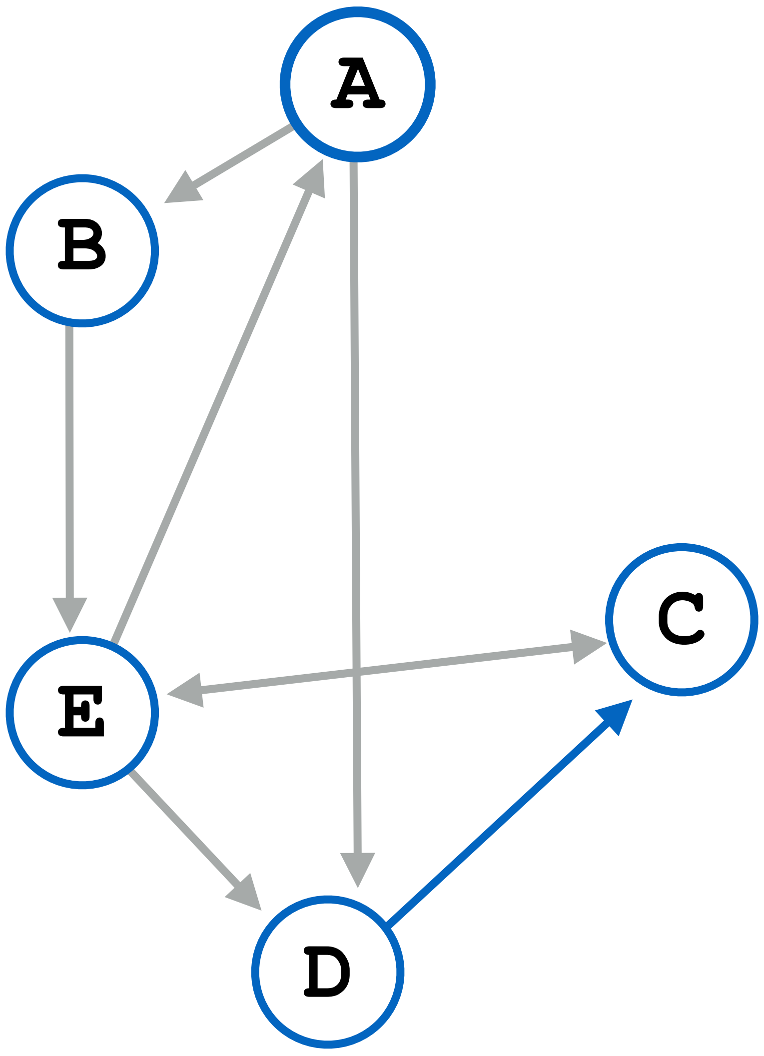
VISITED

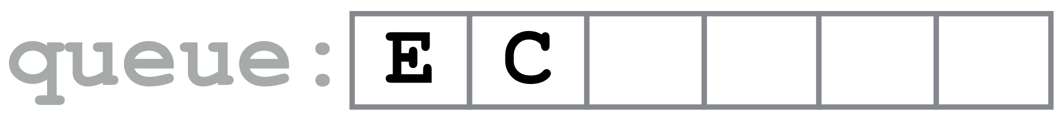
UNVISITED

DEQUEUE THE FIRST NODE IN THE QUEUE (**D**)

MARK AND ENQUEUE **D**'S UNVISITED NEIGHBORS

```
C.cameFrom = D
C.visited = true
```

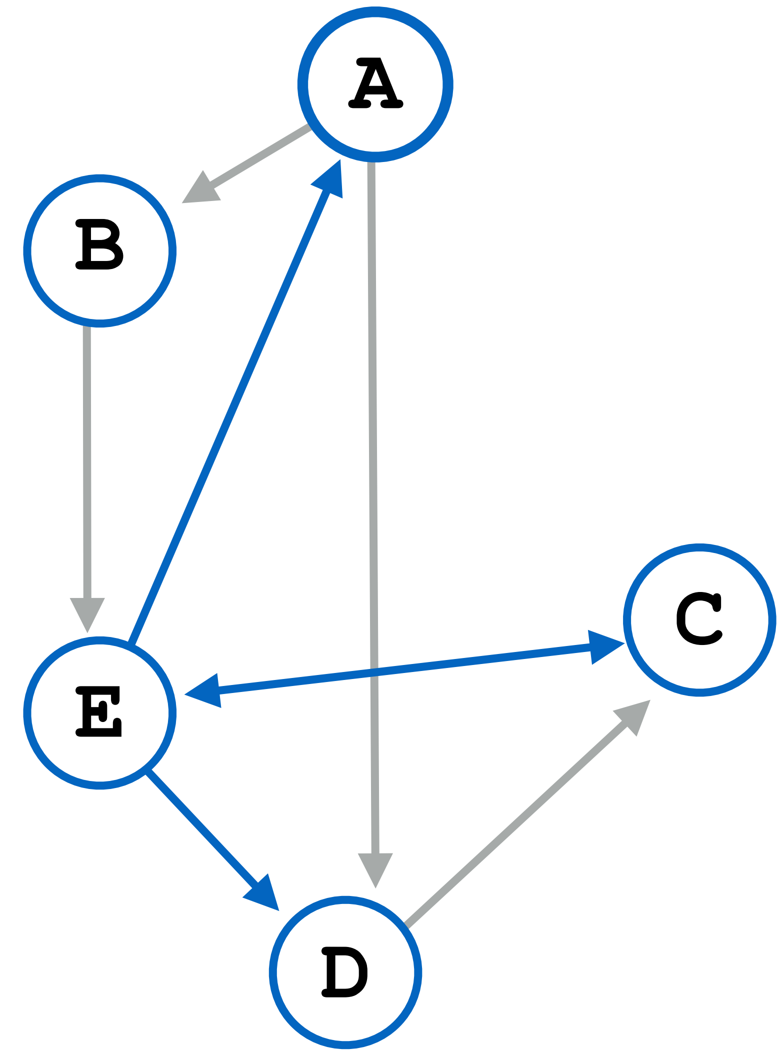


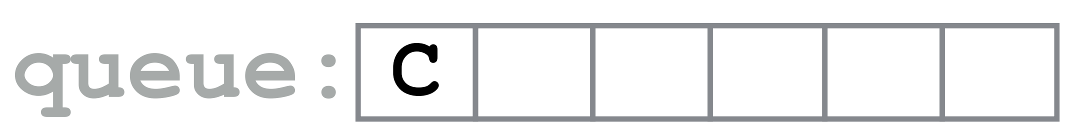


DEQUEUE THE FIRST NODE IN THE QUEUE (**E**)

MARK AND ENQUEUE **E**'S UNVISITED NEIGHBORS

(NO UNVISITED NEIGHBORS!)



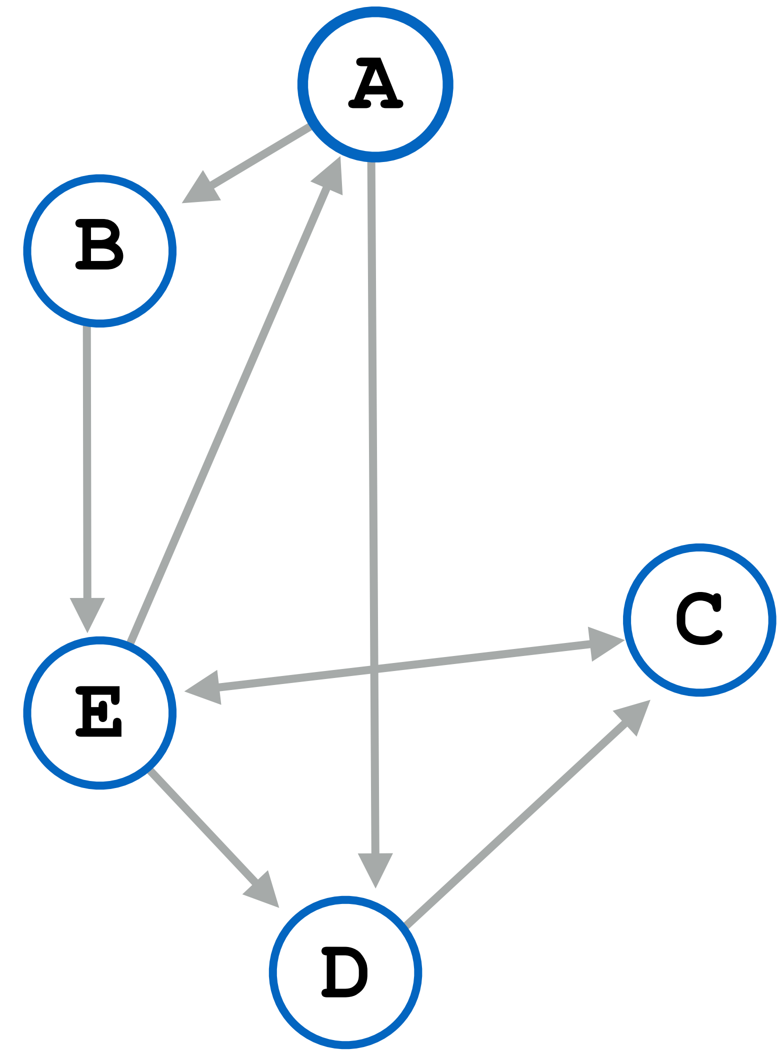


DEQUEUE THE FIRST NODE IN THE QUEUE (**C**)

C IS THE GOAL! RECONSTRUCT THE PATH WITH **`cameFrom`** REFERENCES

```
C.cameFrom = D,
D.cameFrom = A
```

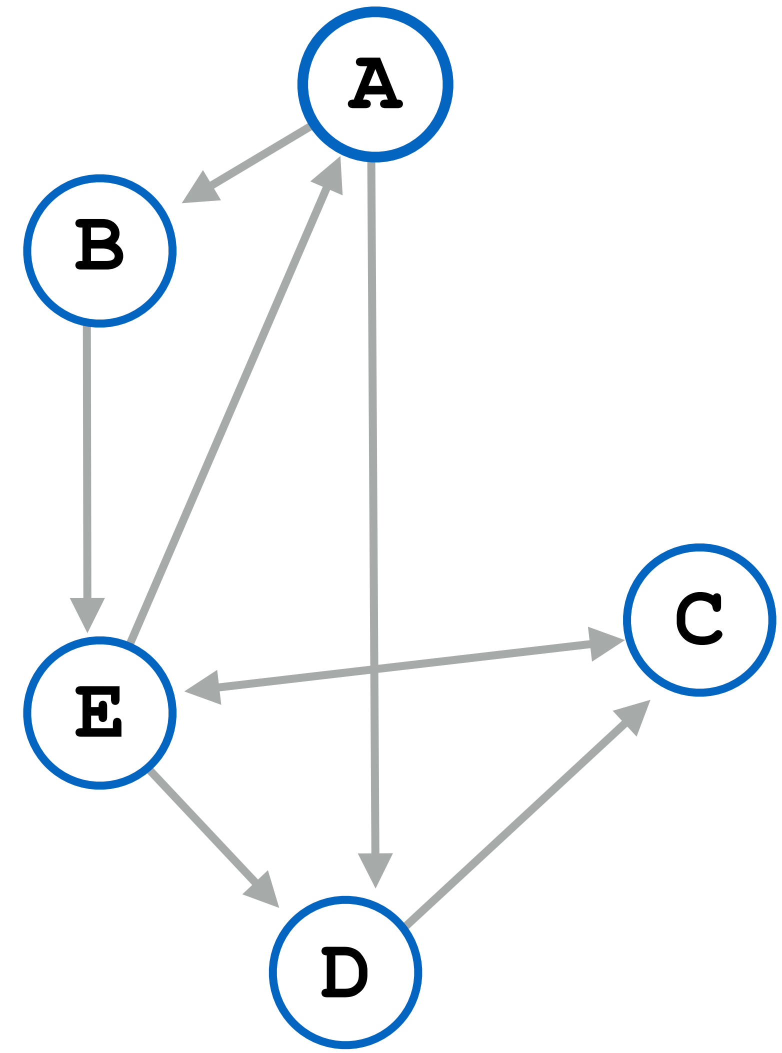
PATH: **A — D — C**



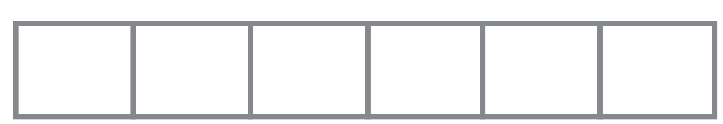
IS THIS THE SHORTEST PATH?
PATH: **A — D — C**

BFS VISITS NODES CLOSETS TO THE START-POINT FIRST

THEREFORE, THE FIRST PATH FOUND IS THE SHORTEST PATH (CLOSEST TO THE START NODE)



queue:



VISITED

UNVISITED

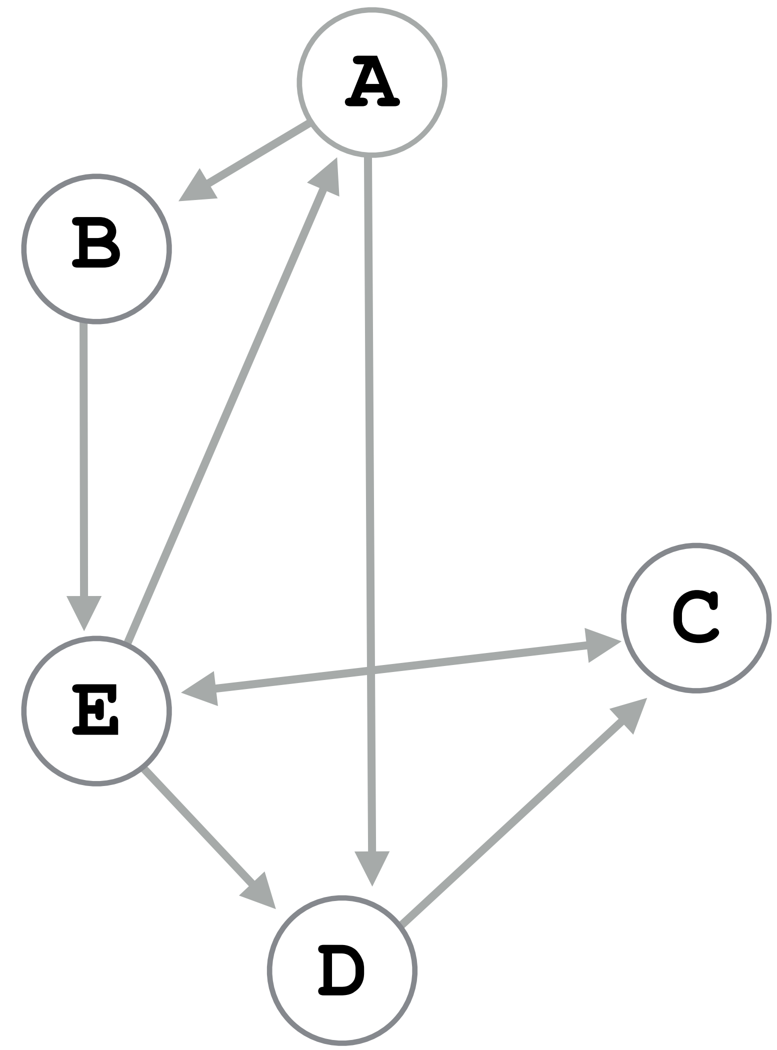
```
BFS(Node start, Node goal)
{
  start.visited = true
  Q.enqueue(start)

  while(!Q.empty())
  {
    Node curr = Q.dequeue()
    if(curr.equals(goal))
      return

    for(Node next : curr.neighbors)
      if(!next.visited)
      {
        next.visited = true
        next.cameFrom = curr
        Q.enqueue(next)
      }
  }
}
```

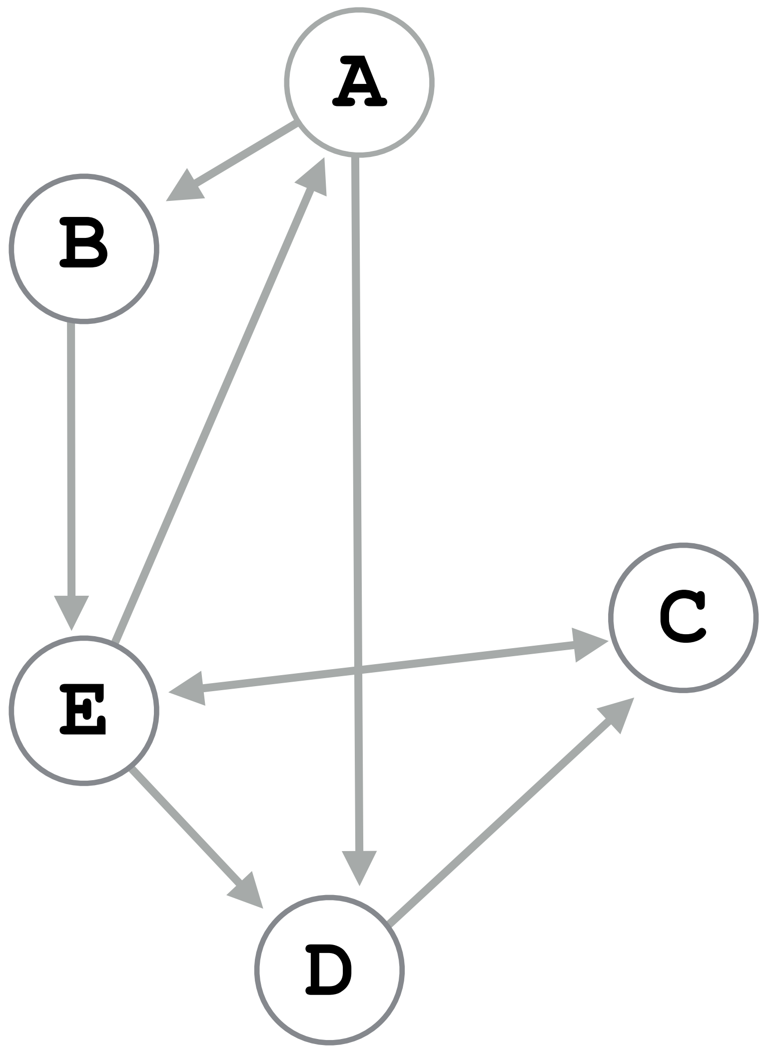
WHAT PATH WILL BFS FIND FROM **B** TO **C**?

A) **B E C**
B) **B E A D C**
C) **B E D C**
D) **none**



WHAT PATH WILL DFS FIND FROM **A** TO **D**?

A) **A B E D**
B) **A D**
C) **none**
D) **this is a trick question**



WHAT IS TRUE OF DFS, SEARCHING FROM A START NODE TO A GOAL NODE?

A) **if a path exists, it will find it**
B) **it is guaranteed to find the shortest path**
C) **it is guaranteed to not find the shortest path**
D) **it must be careful about cycles**
E) **a, b, and d**
F) **a, c, and d**
G) **a and d**

WHAT IS TRUE OF BFS, SEARCHING FROM A START NODE TO A GOAL NODE?

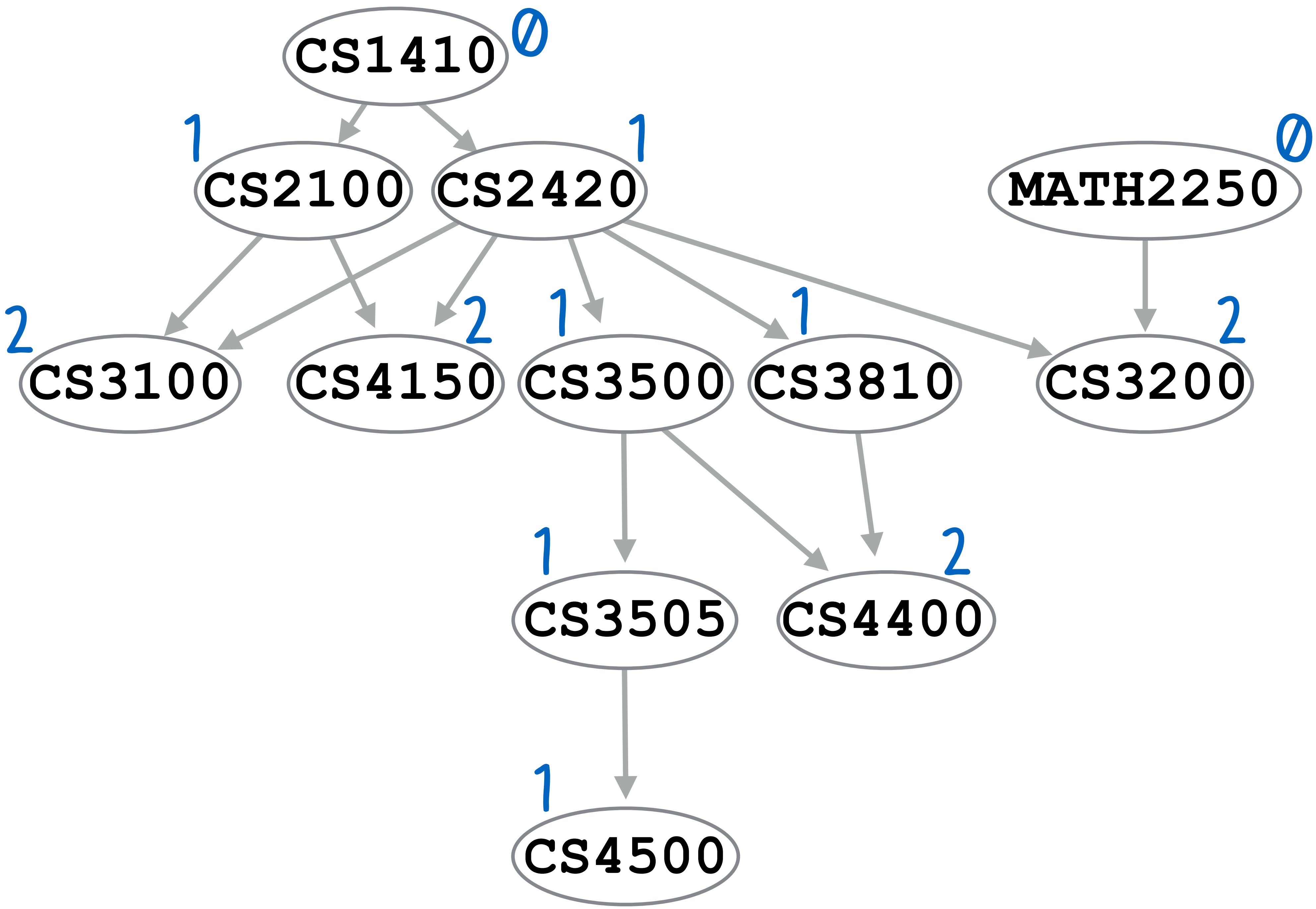
A) **if a path exists, it will find it**
B) **it is guaranteed to find the shortest path**
C) **it is guaranteed to not find the shortest path**
D) **it must be careful about cycles**
E) **a, b, and d**
F) **a, c, and d**
G) **a and d**

topological sort

-the **indegree** of a node is the number of edges it has incoming

-this can be saved as part of the `Node` class, and can be easily computed as the graph is constructed

-any time a node adds another node as a neighbor, increase the neighbor's indegree


CS1410
0
1
CS2100
CS2420
1
MATH2250
0
2
CS3100
CS4150
2
1
CS3500
1
CS3810
CS3200
2
1
CS3505
CS4400
2
1
CS4500

topological sort

-consider a graph with no cycles

-a topological sort orders nodes such that…

  -if there is a path from node A to node B, then A appears before B in the sorted order

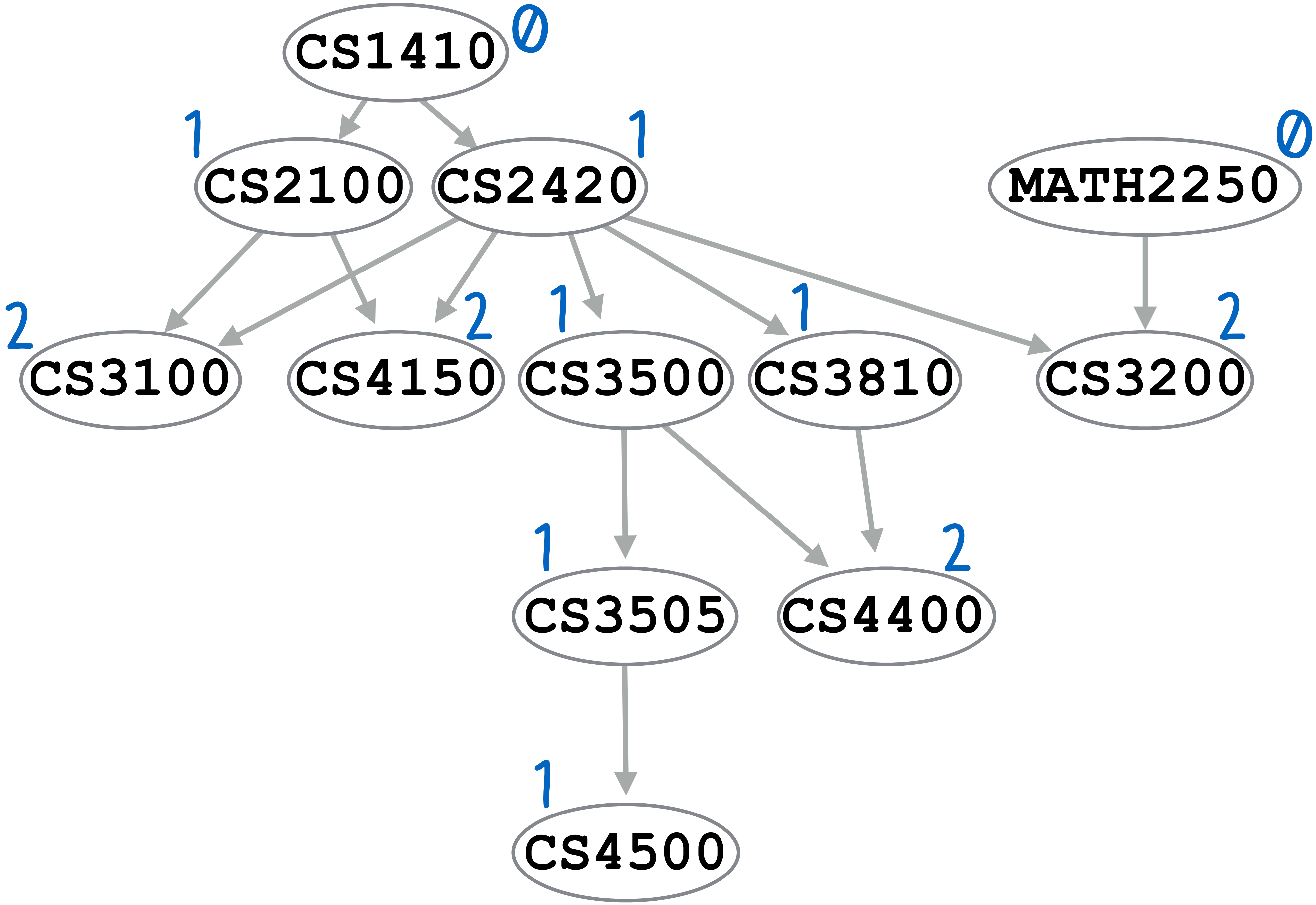
-example: scheduling tasks

  -represent the tasks in a graph

  -if task A must be completed before task B, then A has an edge to B

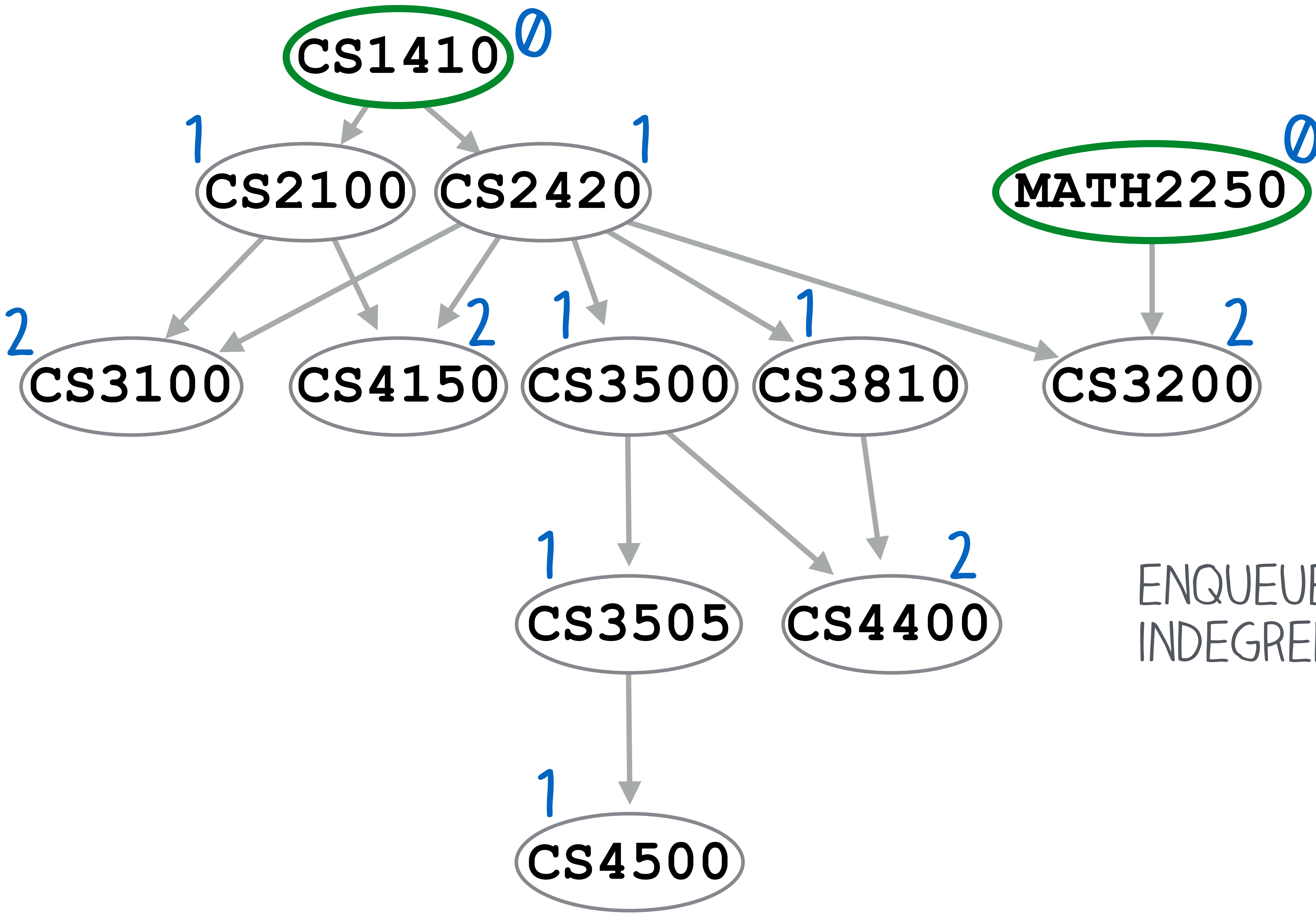
1. step through each node in the graph
   -if any node has indegree 0, add it to a queue

2. while the queue is not empty
   -dequeue the first node in the queue and add to the sorted list
   -visit that node's neighbors and decrease their indegree by 1
   -if a neighbor's new indegree is 0, add it to the queue



queue:

sorted list:

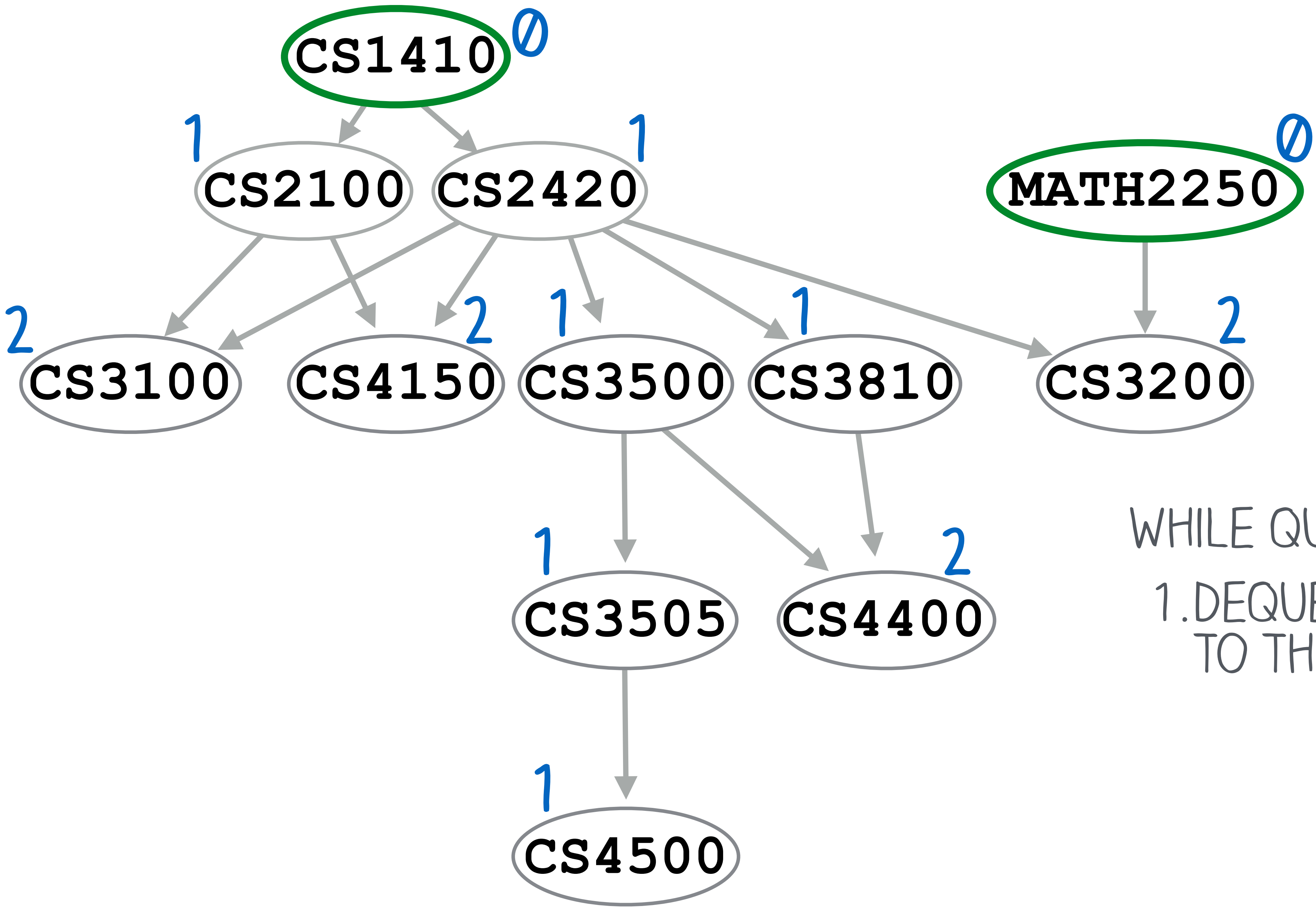


ENQUEUE ANY NODES WITH INDEGREE 0

queue:

CS1410	MATH2250

sorted list:

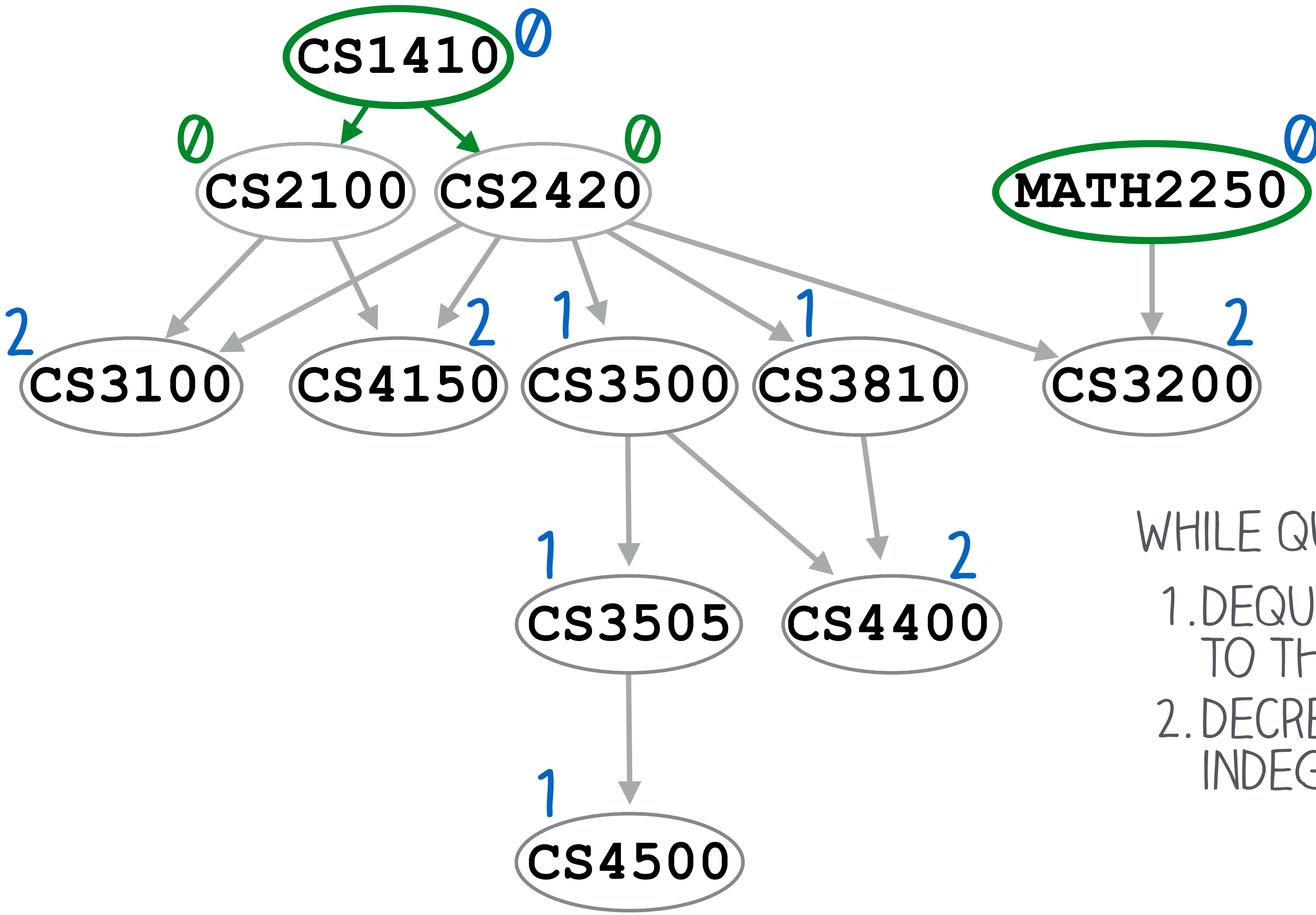


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST

queue: MATH2250

sorted list: CS1410

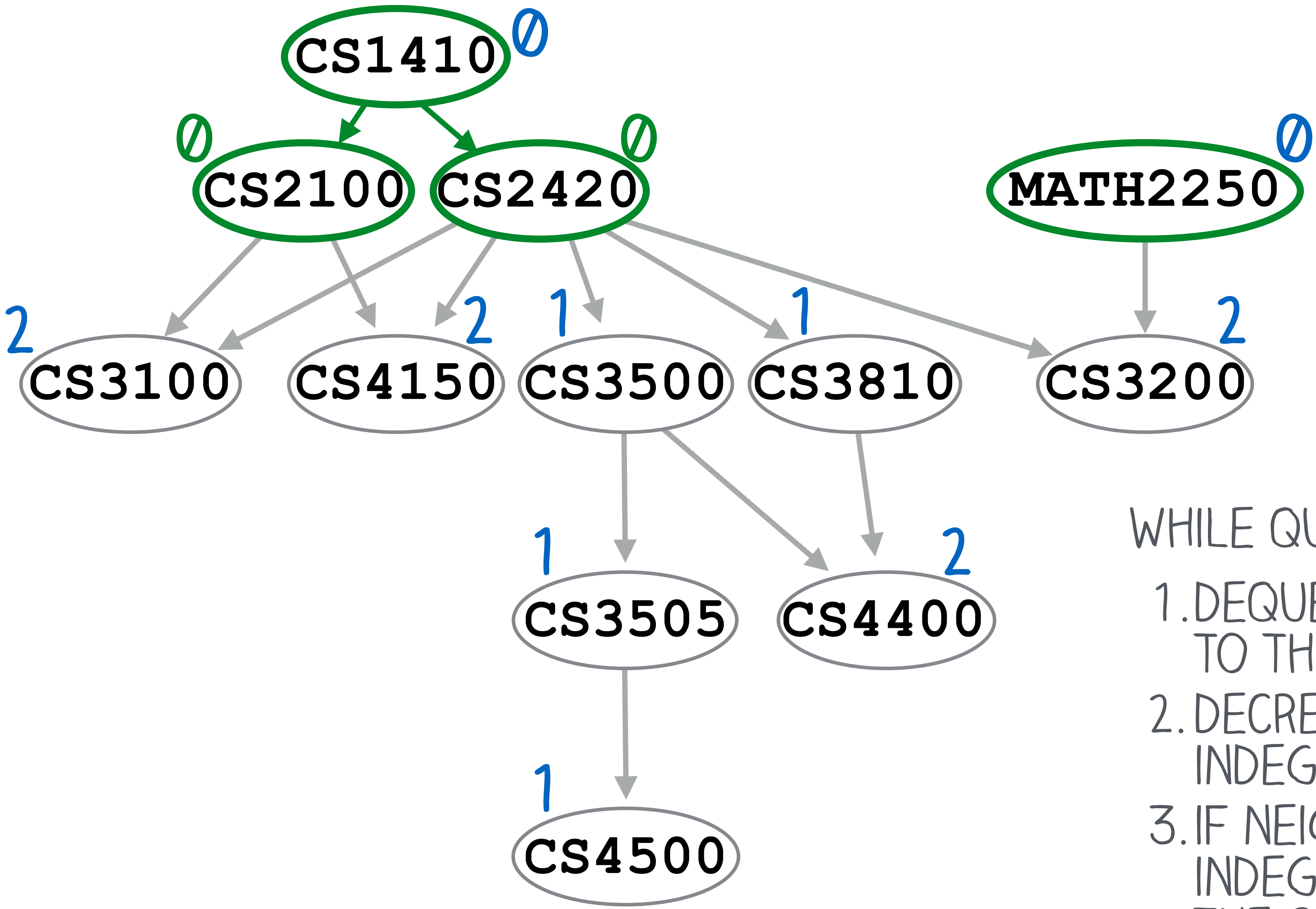


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE

queue: MATH2250

sorted list: CS1410



WHILE QUEUE NOT EMPTY:

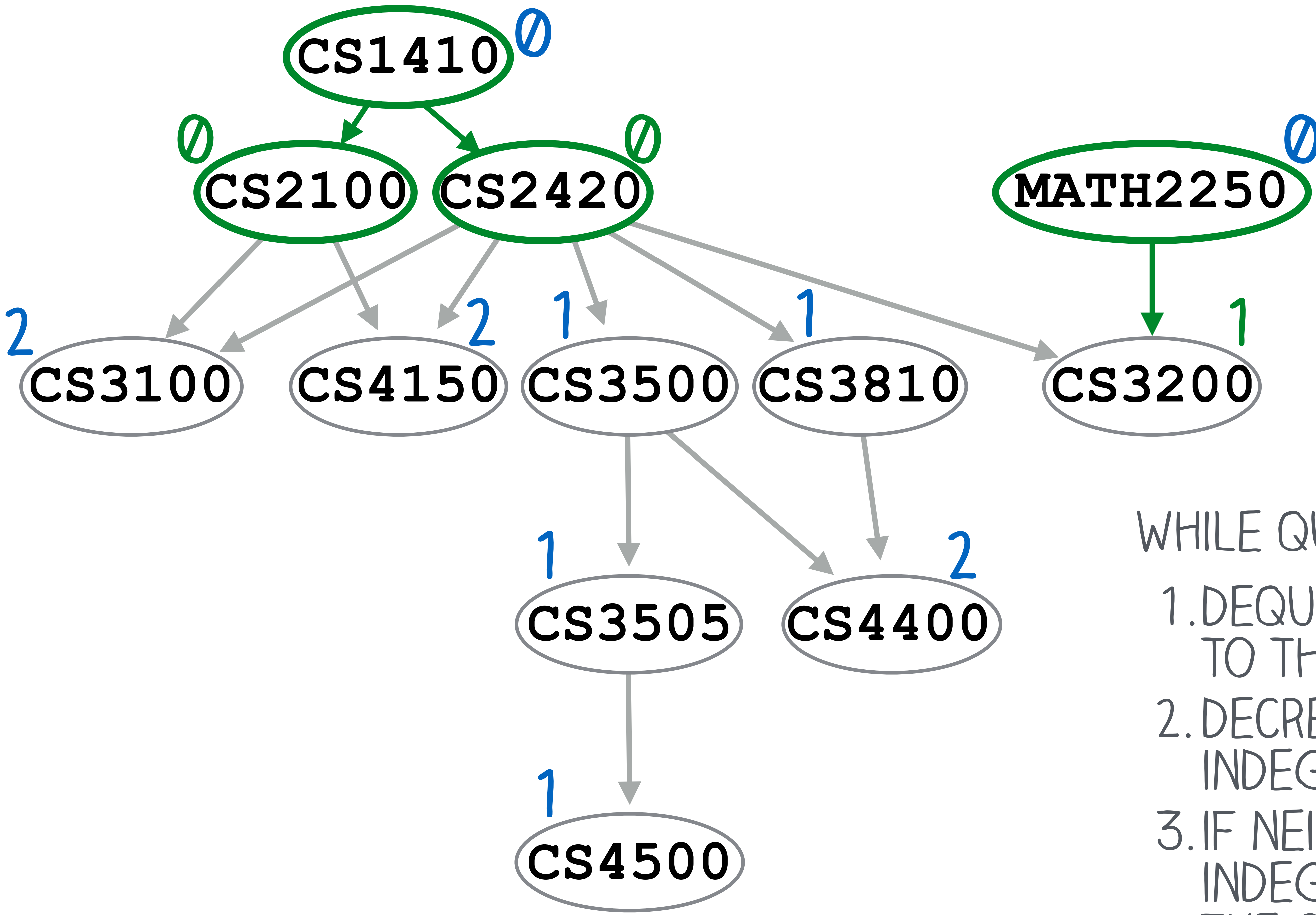
1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue:

MATH2250	CS2100	CS2420

sorted list:

CS1410

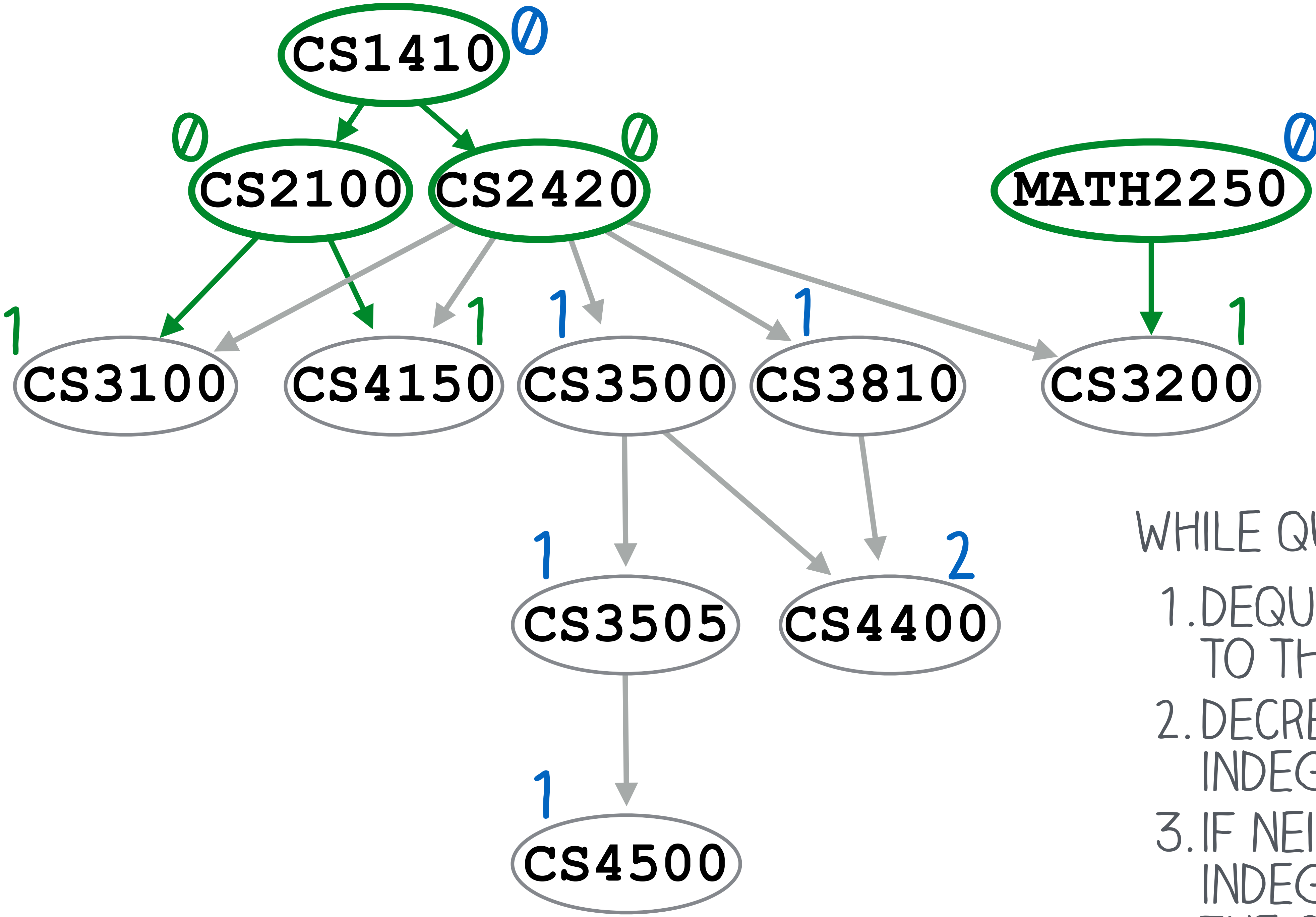


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS ∅, ADD IT TO THE QUEUE

queue: CS2100 | CS2420

sorted list: CS1410 | MATH2250

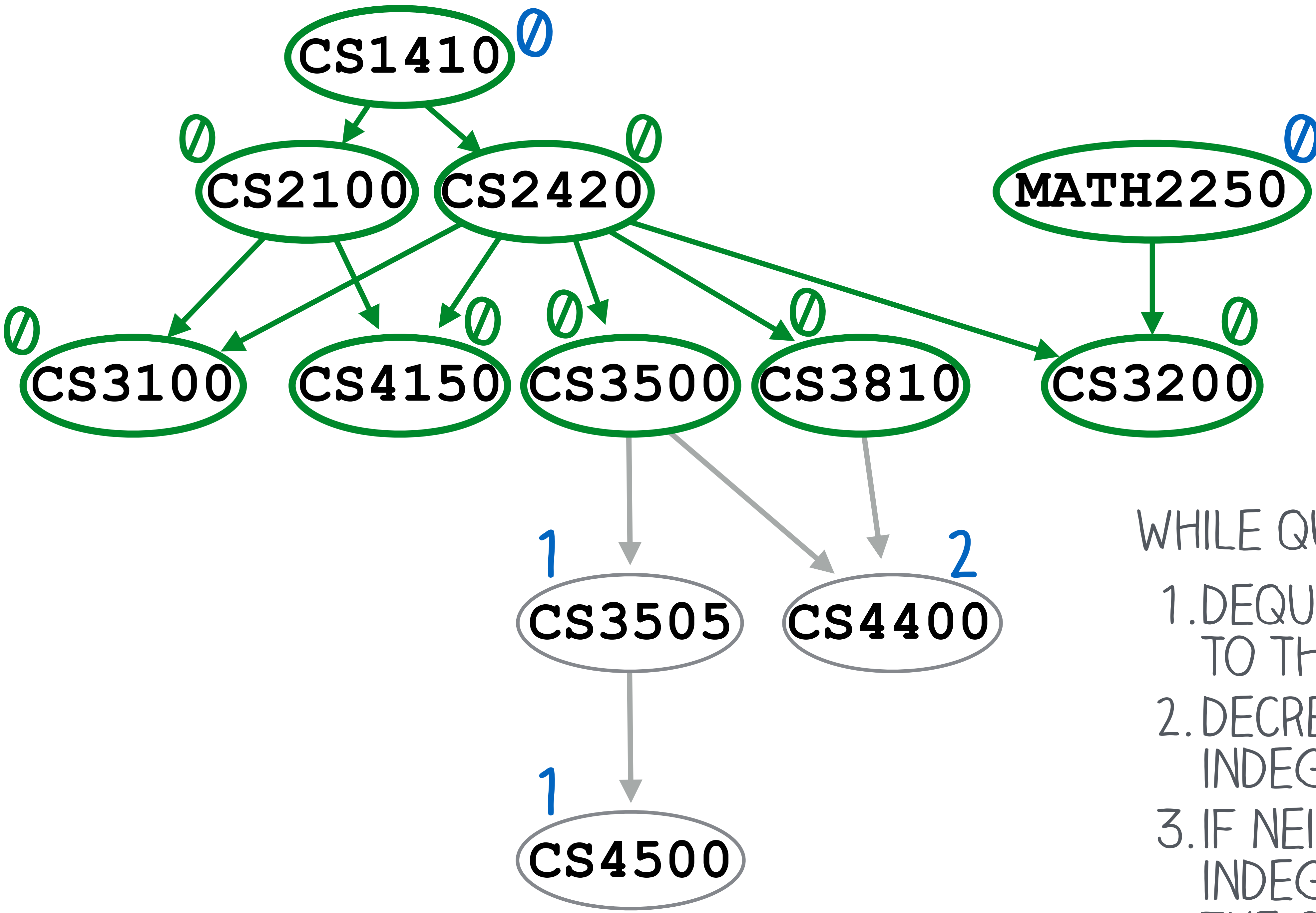


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS Ø, ADD IT TO THE QUEUE

queue: | CS2420 |

sorted list: | CS1410 | MATH2250 | CS2100 |

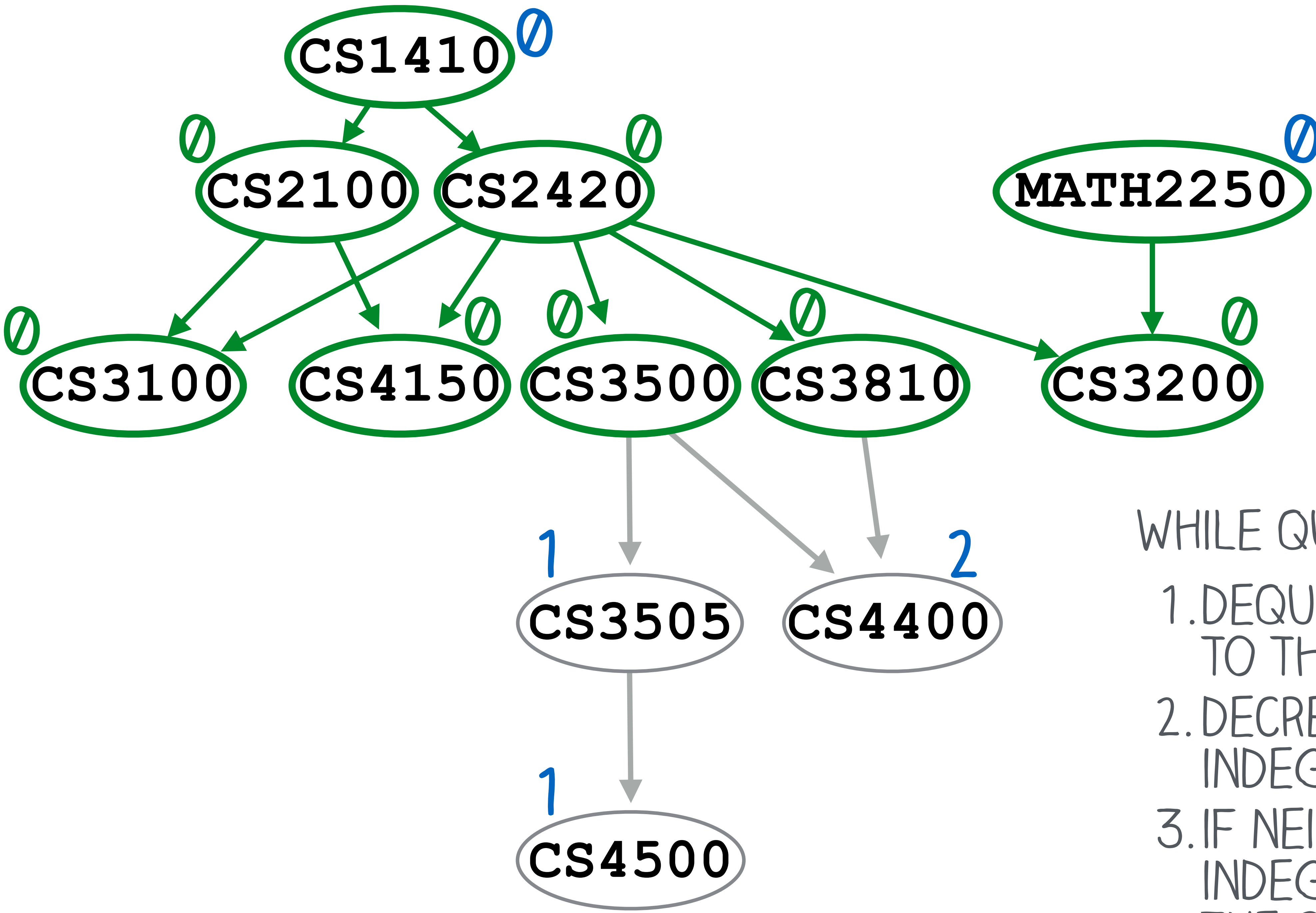


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS Ø, ADD IT TO THE QUEUE

queue:					
	CS3100	CS4150	CS3500	CS3810	CS3200

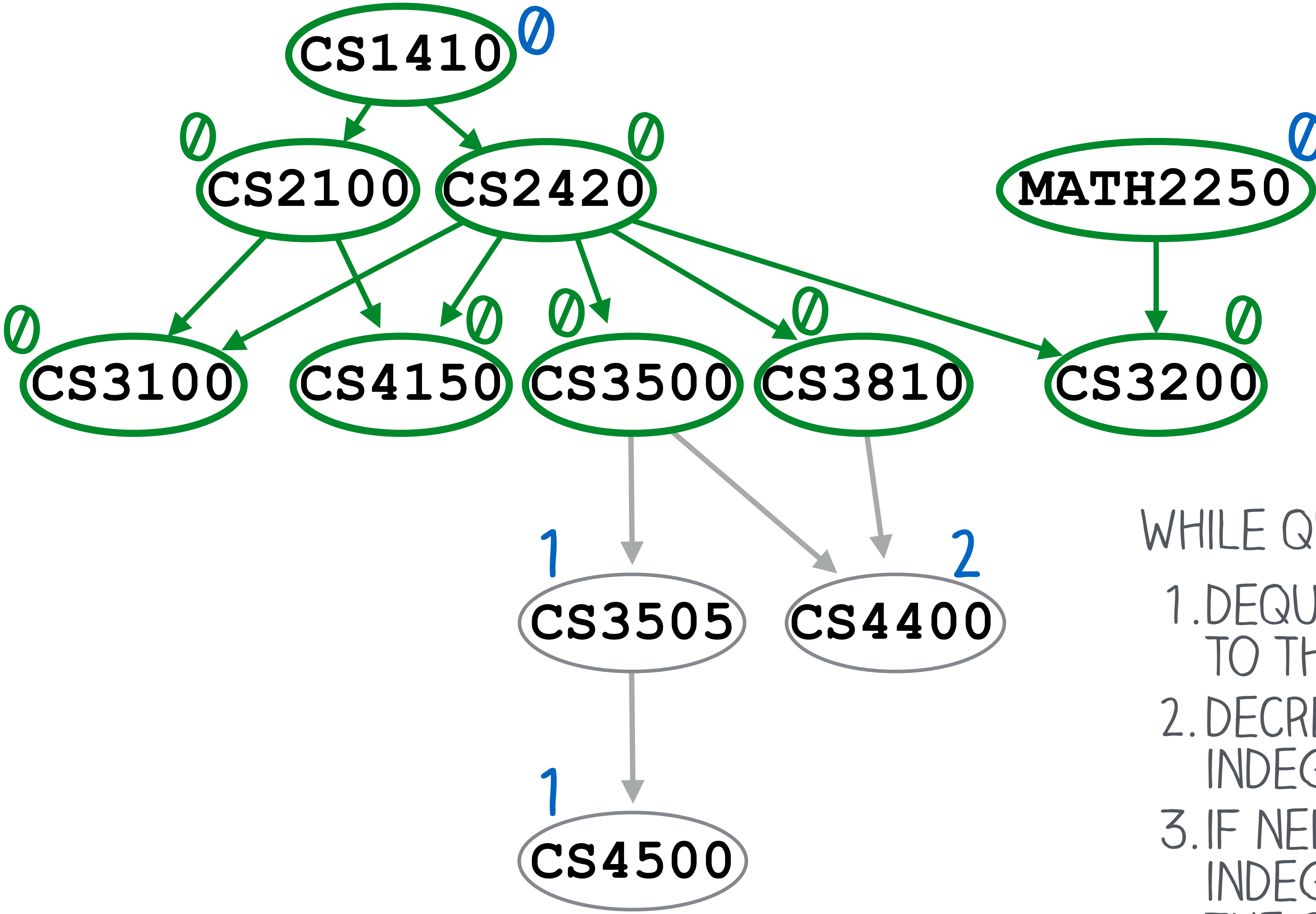
sorted list:				
	CS1410	MATH2250	CS2100	CS2420



WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue:	CS4150	CS3500	CS3810	CS3200	
sorted list:	CS1410	MATH2250	CS2100	CS2420	CS3100

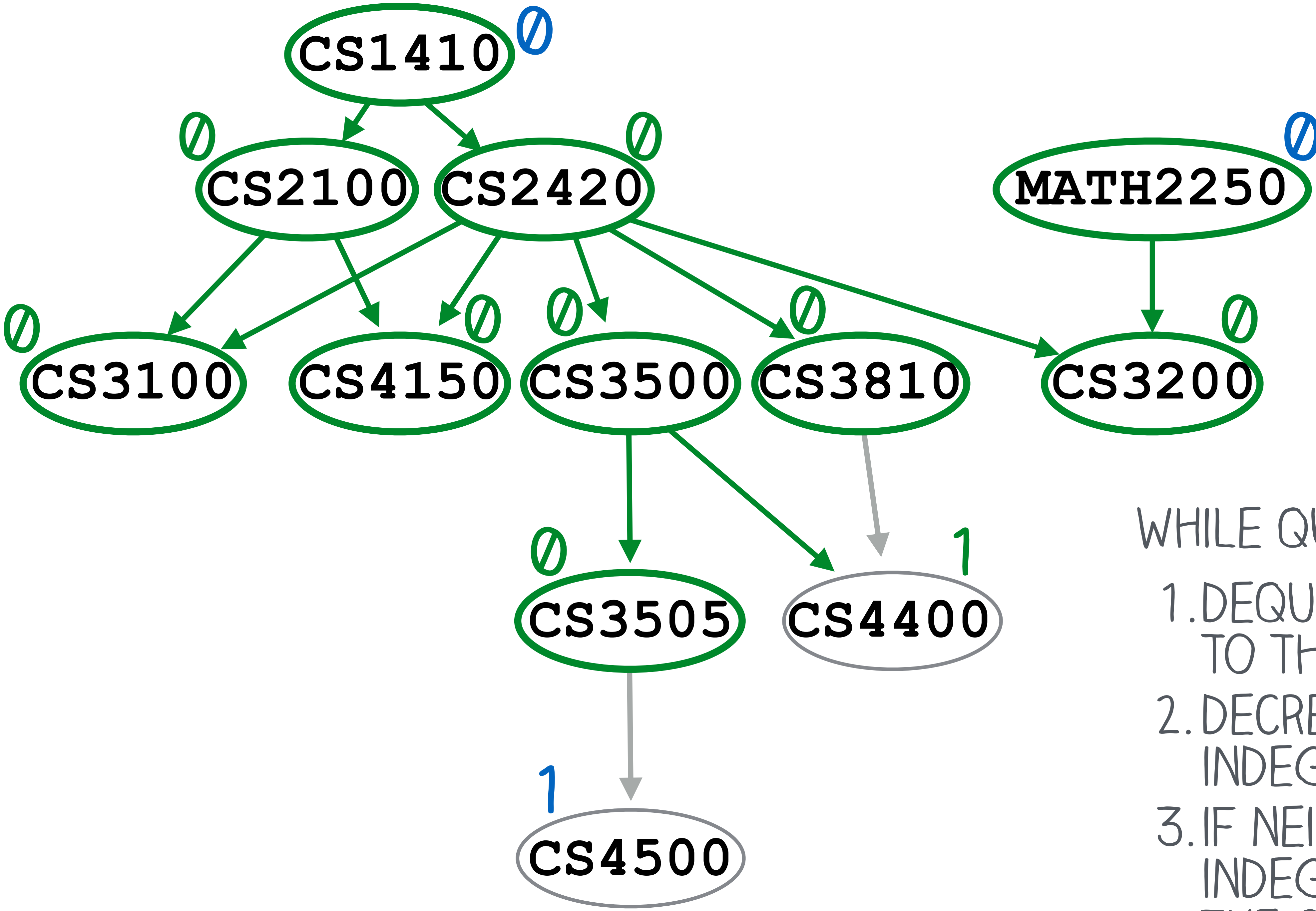


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue:			
	CS3500	CS3810	CS3200

sorted list:						
	CS1410	MATH2250	CS2100	CS2420	CS3100	CS4150

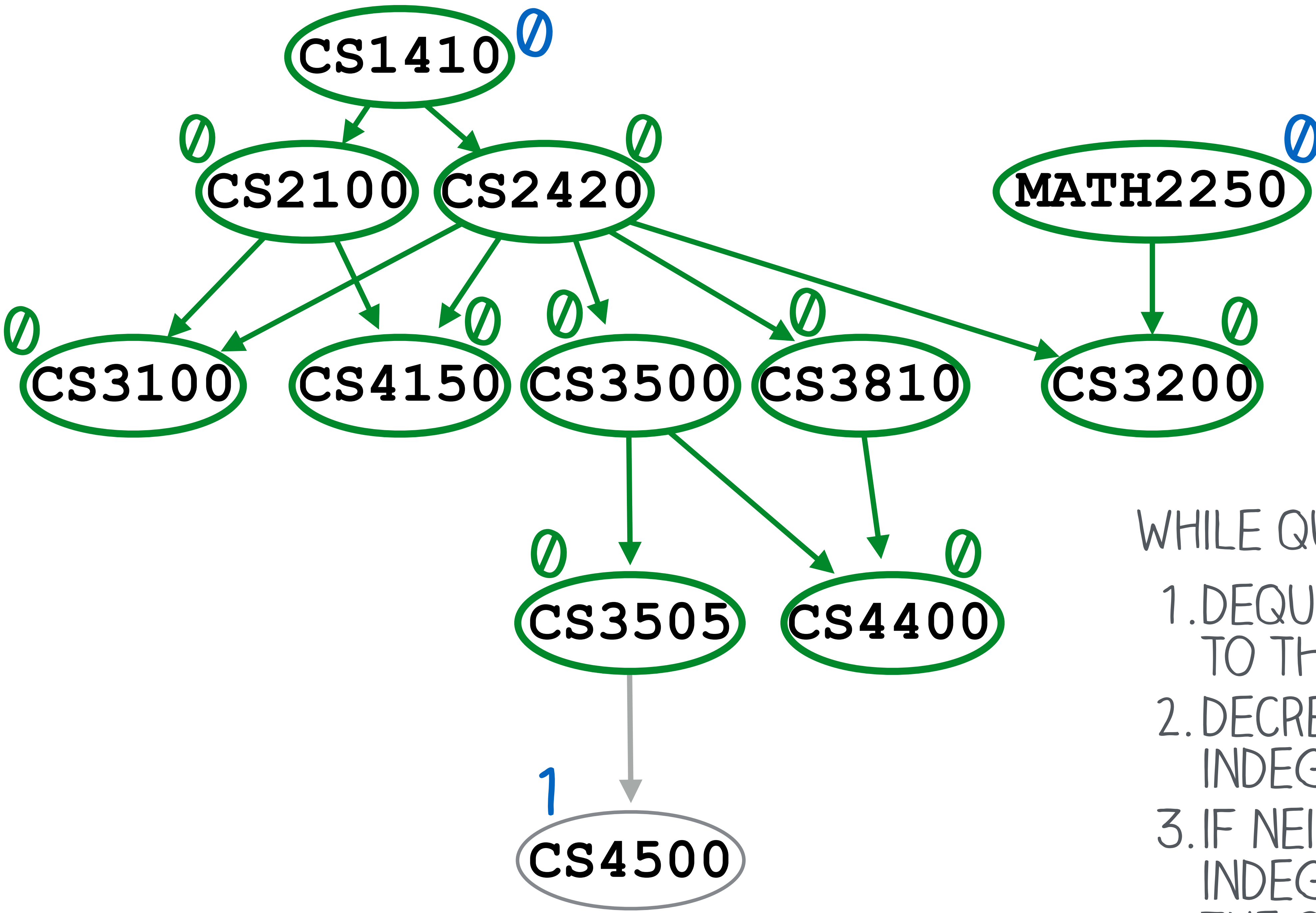


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS Ø, ADD IT TO THE QUEUE

queue: | CS3810 | CS3200 | CS3505 |

sorted list: | CS1410 | … | CS4150 | CS3500 |

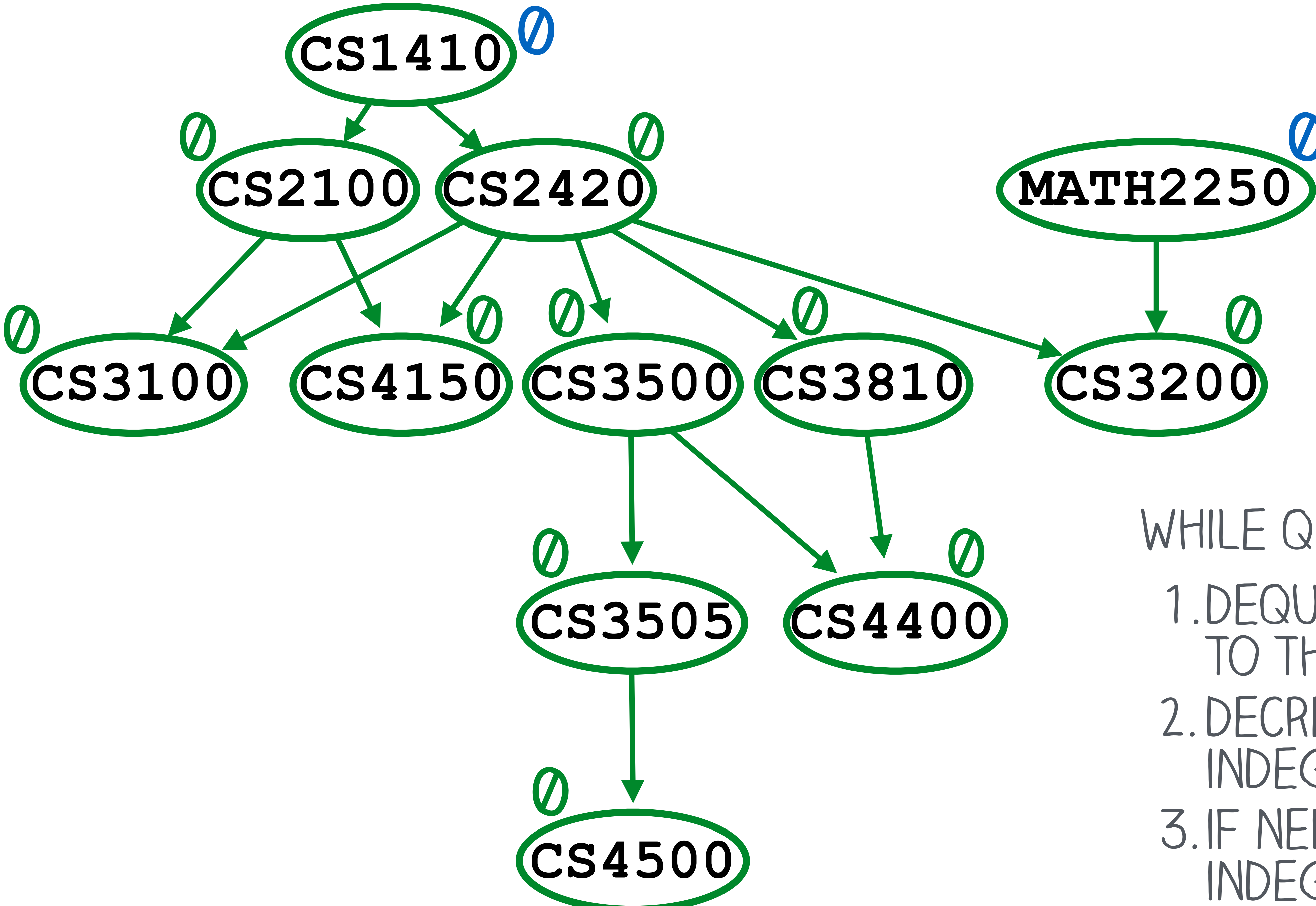


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue: CS3505 | CS4400

sorted list: CS1410 … CS4150 | CS3500 | CS3810 | CS3200



WHILE QUEUE NOT EMPTY:

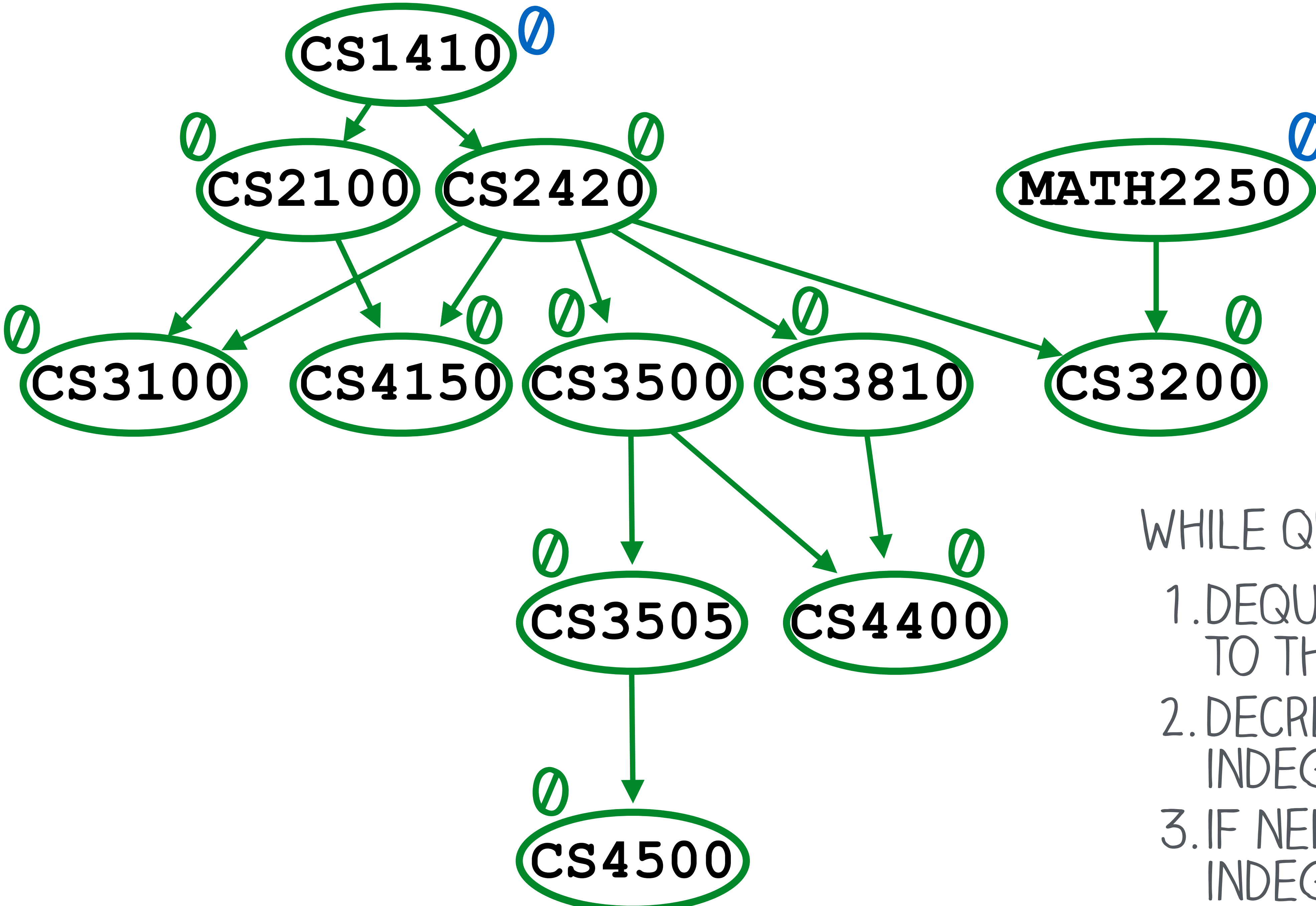
1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue:

CS4400	CS4500

sorted list:

CS1410	...	CS4150	CS3500	CS3810	CS3200	CS3505

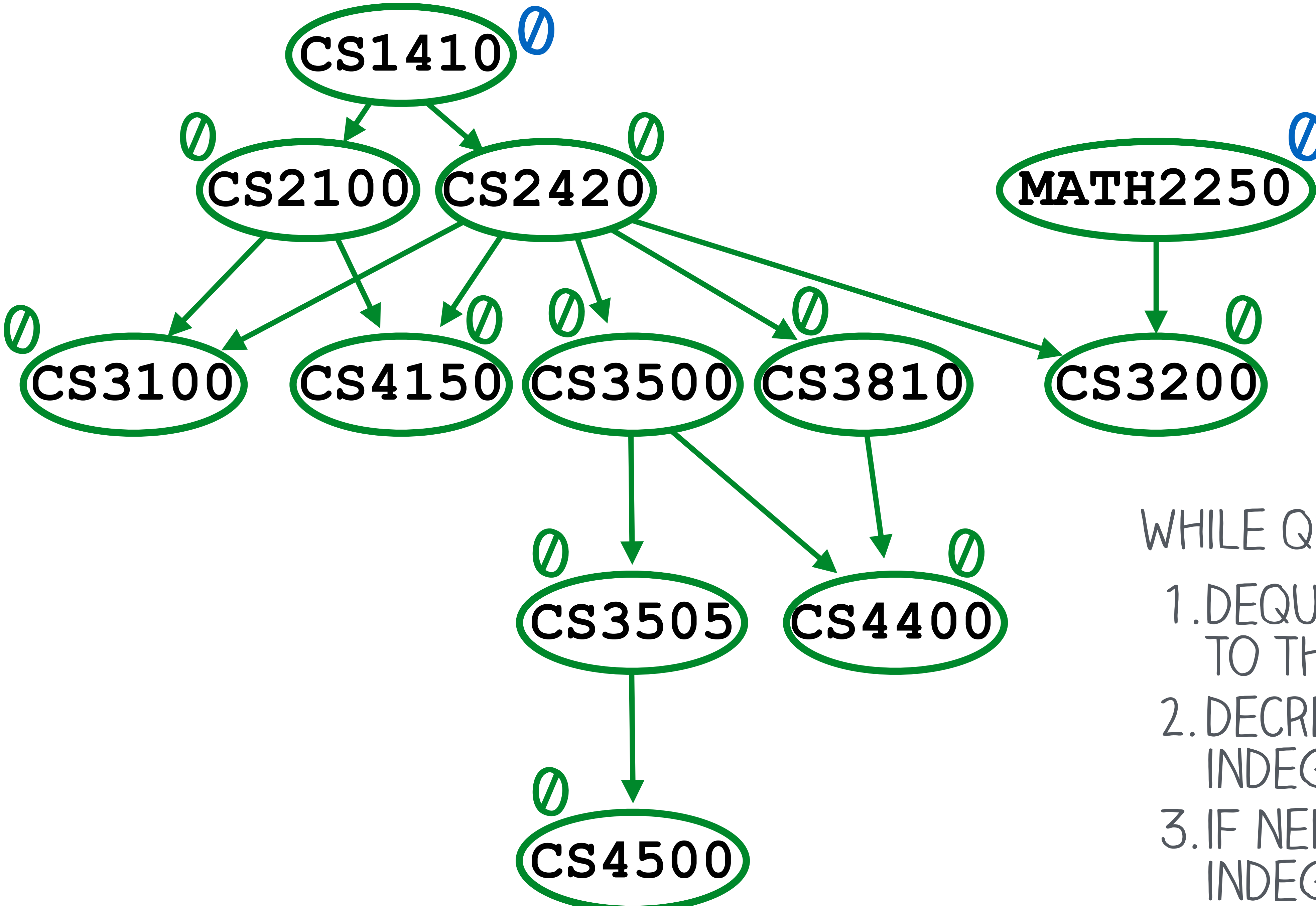


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue: CS4500

sorted list: CS1410 ... CS3810 CS3200 CS3505 CS4400

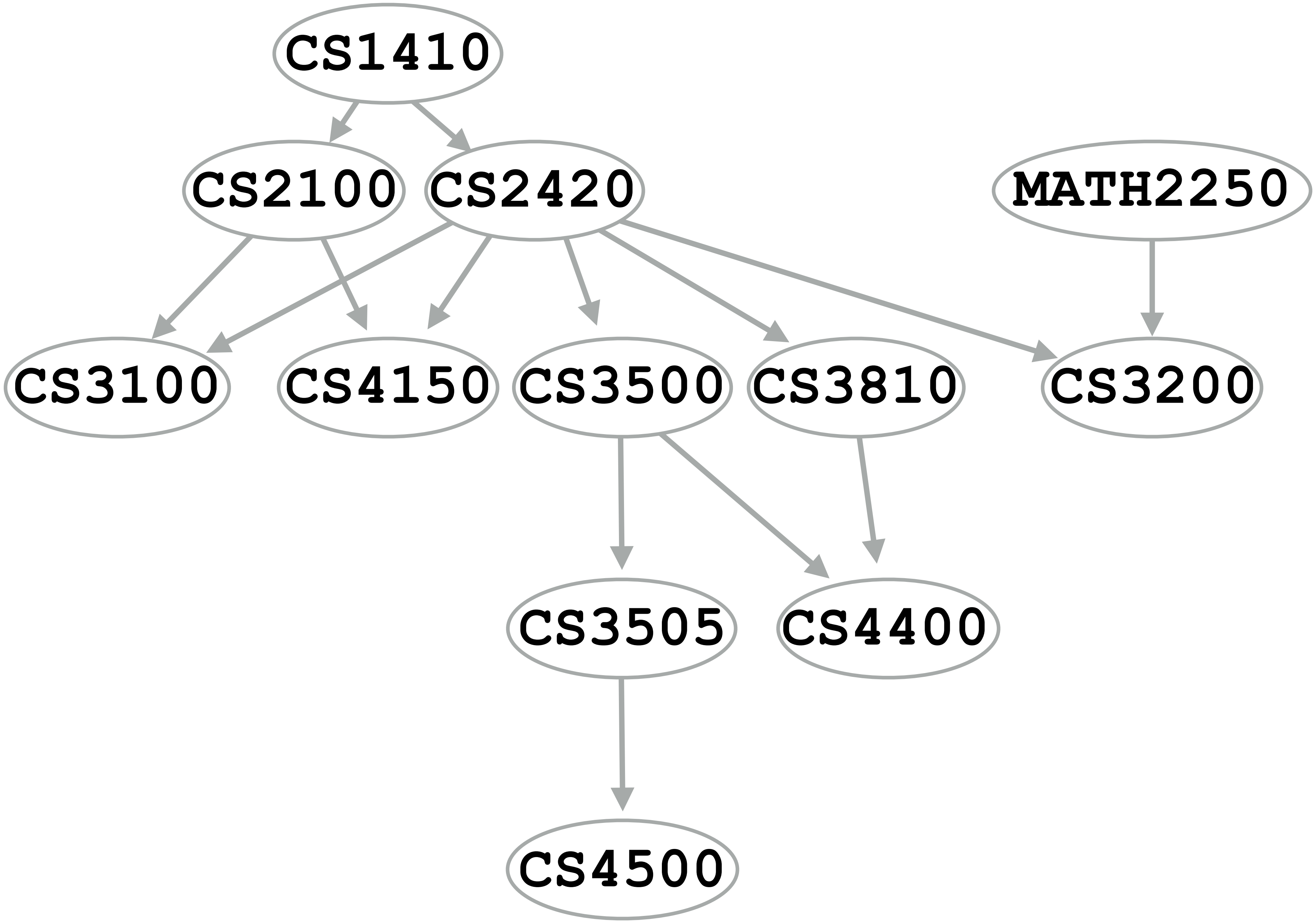


WHILE QUEUE NOT EMPTY:

1. DEQUEUE NODE, ADD IT TO THE SORTED LIST
2. DECREASE NEIGHBORS' INDEGREE
3. IF NEIGHBORS' NEW INDEGREE IS 0, ADD IT TO THE QUEUE

queue:

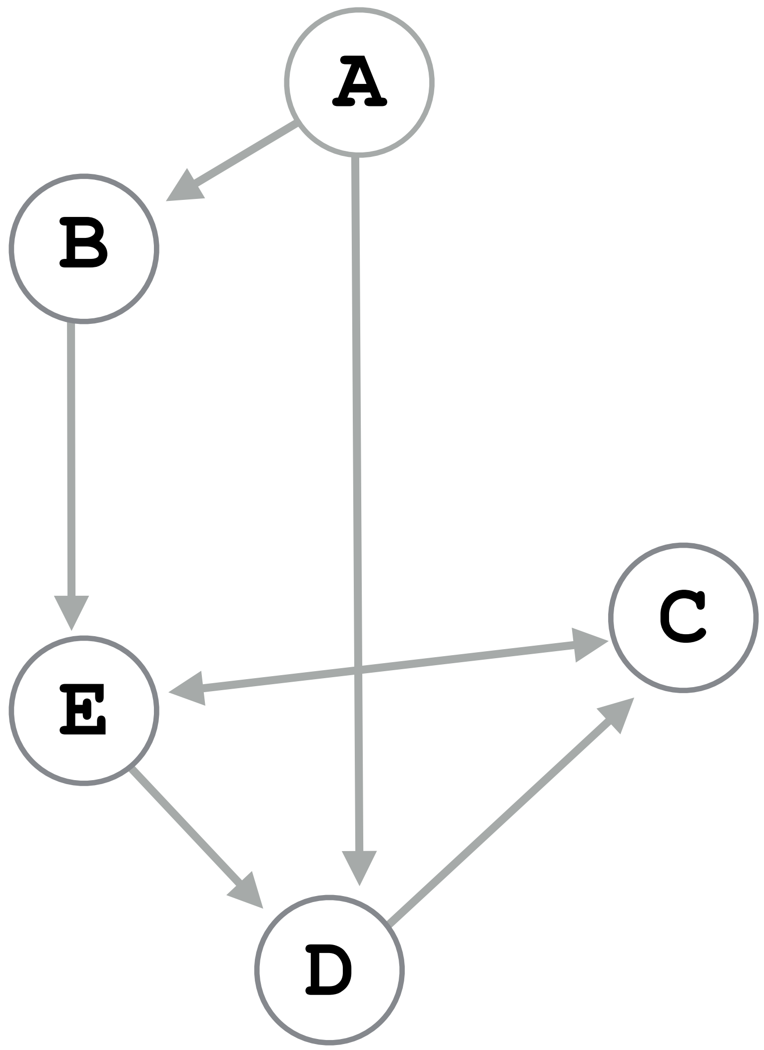
sorted list: | CS1410 | … | CS3810 | CS3200 | CS3505 | CS4400 | CS4500 |



sorted list: CS1410 | MATH2250 | CS2100 | CS2420 | CS3100 | CS4150 | CS3500 | CS3810 | CS3200 | CS3505 | CS4400 | CS4500

WHICH OF THE FOLLOWING IS A VALID TOPOLOGICAL ORDERING?

A) **A D C E B**
B) **C D E B A**
C) **A B E D C**
D) **A B C D E**



next time…

-reading

  -chapter 14 in book

-homework

  -assignment 8 due Thursday

  -assignment 9 out tomorrow