
CSE 331

Software Design & Implementation

Winter 2022

Section 6 – HW6, Path-Finding with BFS, and Parsing

Administrivia

- HW5-part 2 due tonight!
 - We also will be reviewing the pieces from part 1, so make sure to fix things according to your feedback
- HW6 due next Thursday
- Make sure to tag your submissions correctly!
 - Commit and push everything first, then tag the final commit
- Any questions?

Agenda

- HW5 Reminders
- Overview of HW6
- Breadth-first search (BFS)
- Parsing a file in comma-separated-values (CSV) format
- Test scripts and the new test driver

HW5 Reminders

- Your graph should not sort anything!
 - This includes TreeMap/TreeSets
- Your graph should not be printing anything
 - Or returning String for the client to print (except toString())
- Your graph should not have any pathfinding
- Your graph should not have generic node/edge labels – just Strings for now
- Any questions?

HW6: The MarvelPaths program

- You were the implementor but now are the client of your graph ADT!
- MarvelPaths is a command-line program you write to find how two Marvel characters are connected through comic-book co-appearances
- Using a large dataset in comma-separated-values (CSV) format
 - Each entry is a particular appearance of a character in a comic book
- Dataset processed to initialize the social-network graph
- Main functionality is finding shortest path in this social network

Outline of the assignment

0. Understand the dataset (`marvel.csv`) and CSV format
1. Complete `MarvelParser` class to read CSV-formatted files
2. Implement graph initialization in `MarvelPaths` class
3. Implement path-finding via BFS in `MarvelPaths` class
4. Write suites of scripts tests and of implementation tests
 - Implement `MarvelTestDriver` for new test-script commands
5. Write `main` method in `MarvelPaths` for command-line usage

Outline of the assignment

0. Understand the dataset (`marvel.csv`) and CSV format
1. Complete `MarvelParser` class to read CSV-formatted files
2. Implement graph initialization in `MarvelPaths` class
3. Implement path-finding via BFS in `MarvelPaths` class
4. Write suites of script tests and of implementation tests
 - Implement `MarvelTestDriver` for new test-script commands
5. Write `main` method in `MarvelPaths` for command-line usage

Breadth-first search

- Breadth-first search (BFS) is an algorithm for path-finding
 - Works just as well on directed and undirected graphs
 - Often used to discover connectivity in a graph
- Finds a path with the least number of edges
 - Recall that a path is a chain of edges, like $\langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle$
 - Ignores edge labels, so not used for weighted graphs
- Often mentioned alongside depth-first search (DFS)
 - BFS looks “wide” whereas DFS looks “deep”
 - DFS can’t promise to find the shortest path

The BFS algorithm — first attempt (**incorrect**)

```
push start node onto a queue
```

```
while queue is not empty:
```

```
    pop node  $N$  off queue
```

```
    if  $N$  is goal node:
```

```
        return true
```

```
    else:
```

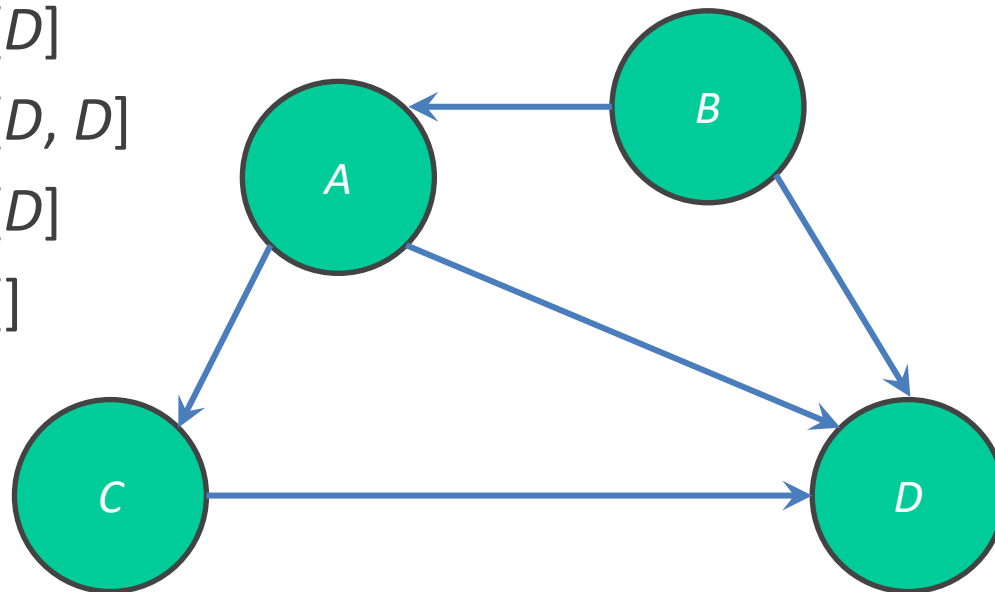
```
        for each node  $O$  in children of  $N$ :
```

```
            push  $O$  onto queue
```

```
return false
```

BFS: example on a simple graph

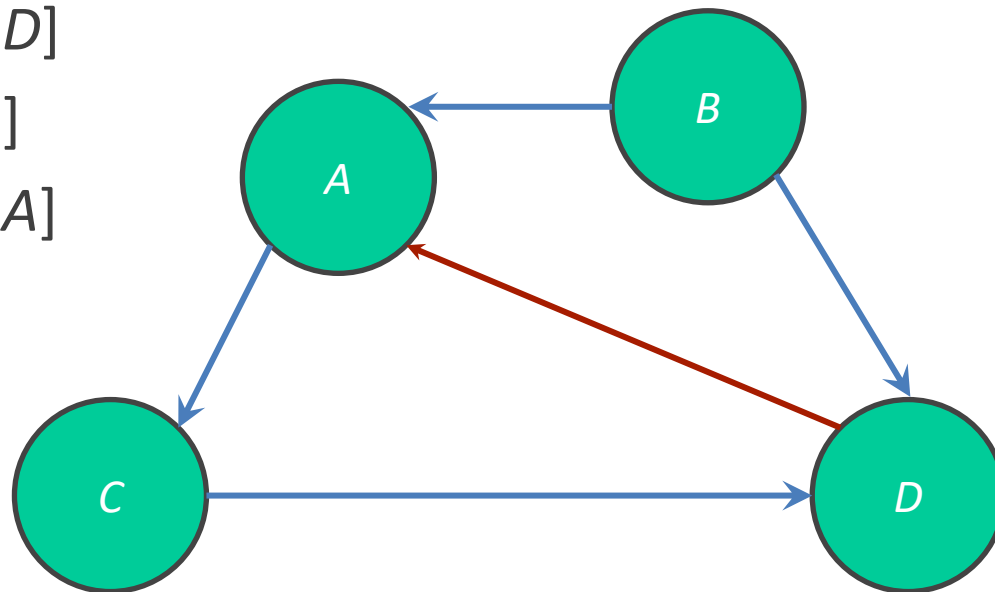
push start	$Q = [A]$	start = A
pop A	$Q = []$	goal = B
push C	$Q = [C]$	
push D	$Q = [D, C]$	
pop C	$Q = [D]$	
push D	$Q = [D, D]$	
pop D	$Q = [D]$	
pop D	$Q = []$	
return false		



BFS: example on a cyclic graph

push start	$Q = [A]$	start = A
pop A	$Q = []$	goal = B
push C	$Q = [C]$	
pop C	$Q = []$	
push D	$Q = [D]$	
pop D	$Q = []$	
push A	$Q = [A]$	

INFINITE LOOP!



The BFS algorithm

```
push start node onto a queue
```

```
mark start node as visited
```

```
while queue is not empty:
```

```
    pop node  $N$  off queue
```

```
    if  $N$  is goal:
```

```
        return true
```

```
    else:
```

```
        for each node  $O$  that is child of  $N$ :
```

```
            if  $O$  is not marked visited:
```

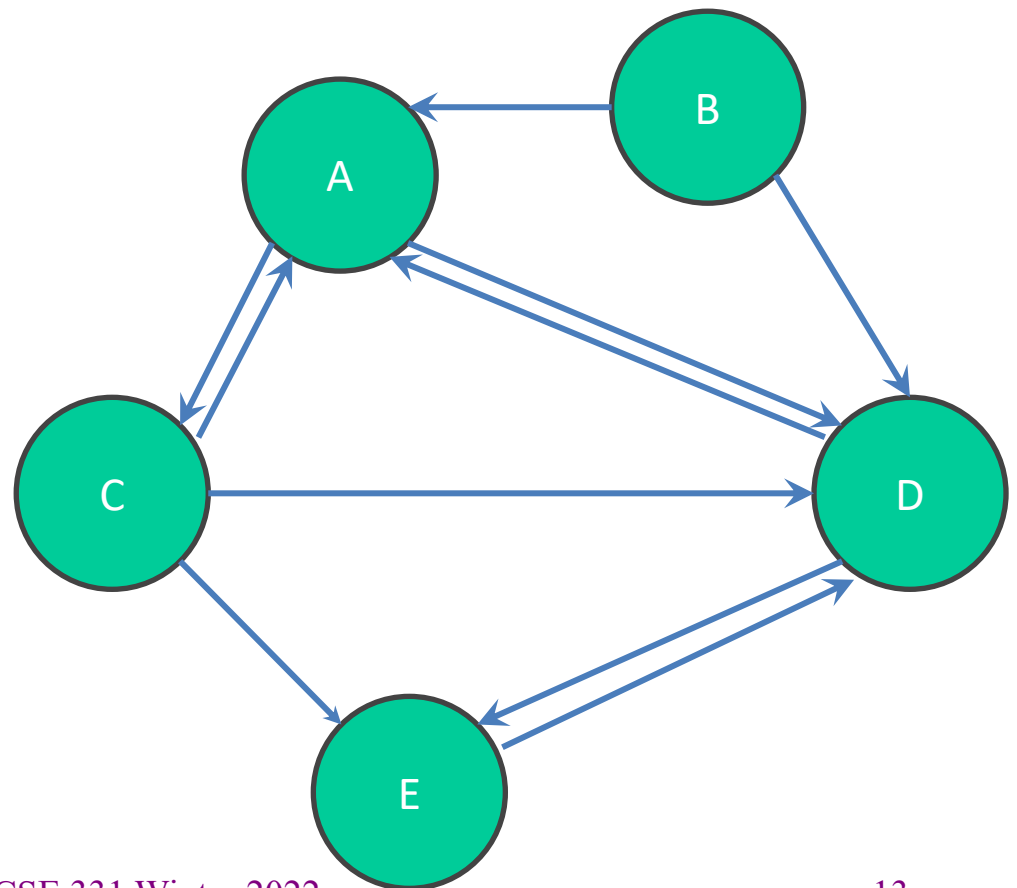
```
                mark node  $O$  as visited
```

```
                push  $O$  onto queue
```

```
return false
```

BFS: example on a cyclic graph

start = A
goal = B

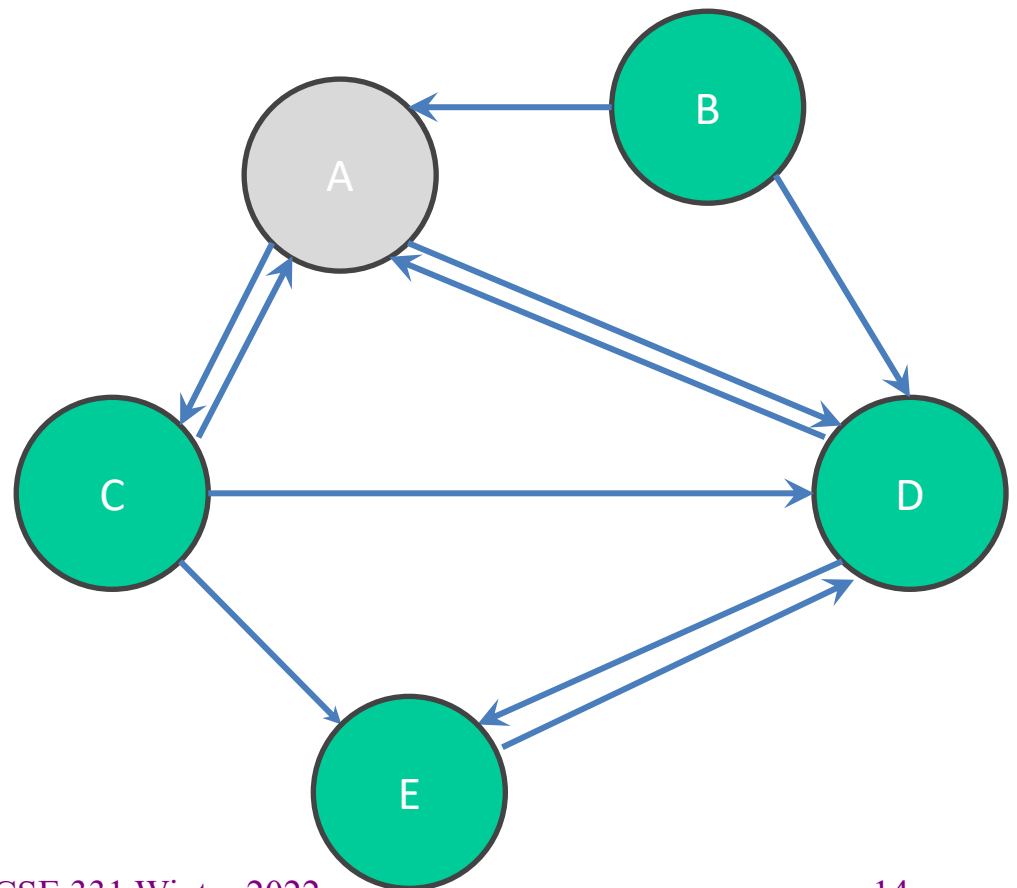


BFS: example on a cyclic graph

push start

$Q = [A]$

start = A
goal = B



BFS: example on a cyclic graph

push start

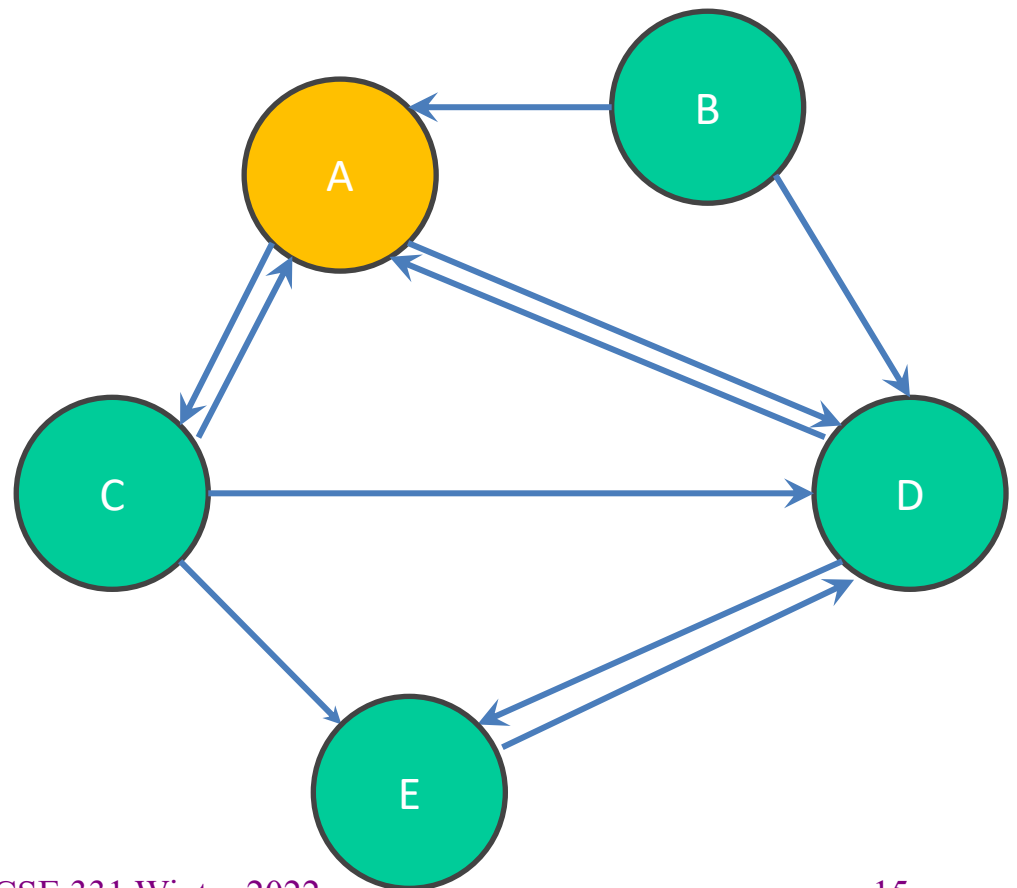
$Q = [A]$

start = A

pop A

$Q = []$

goal = B



BFS: example on a cyclic graph

push start

$Q = [A]$

pop A

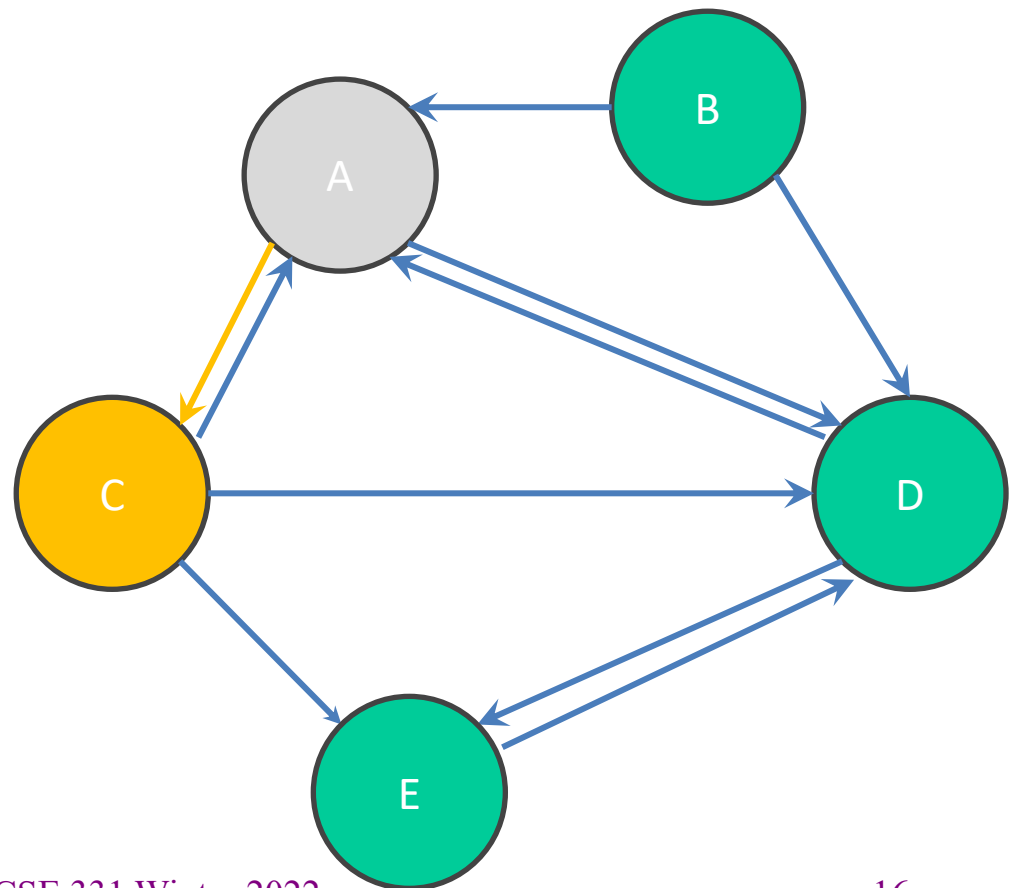
$Q = []$

push C

$Q = [C]$

start = A

goal = B



BFS: example on a cyclic graph

push start

$Q = [A]$

pop A

$Q = []$

push C

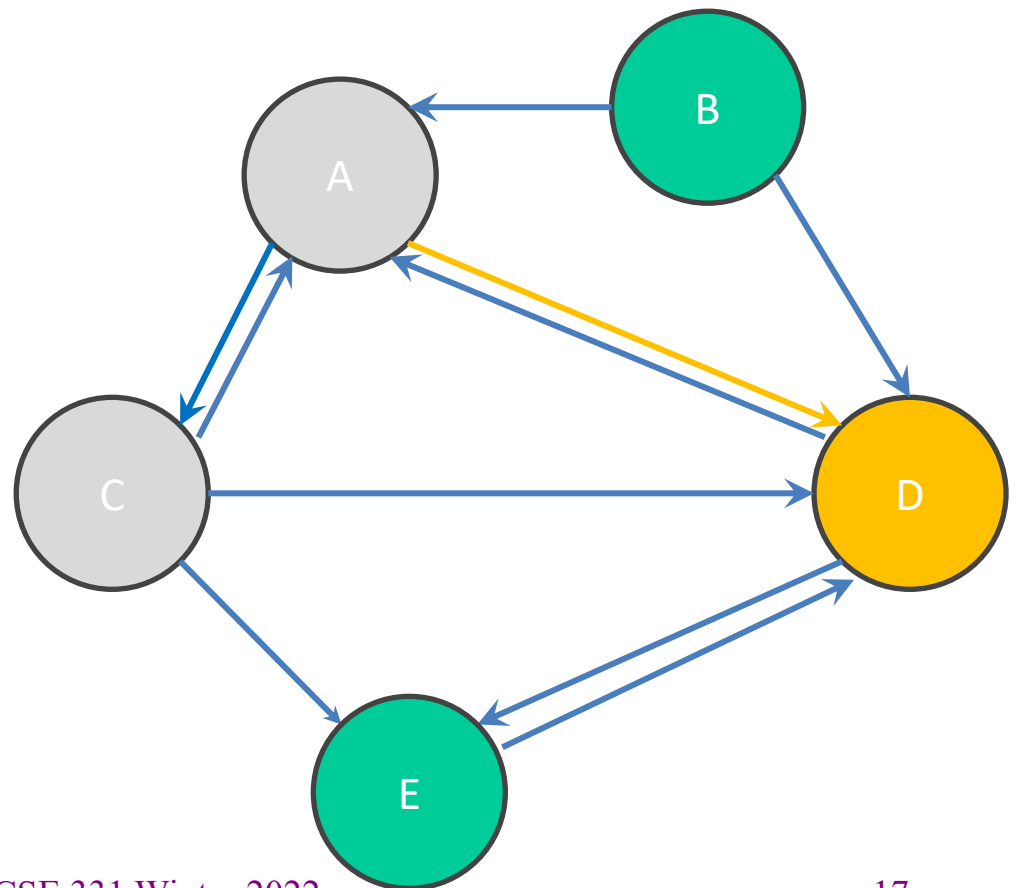
$Q = [C]$

push D

$Q = [D, C]$

start = A

goal = B



BFS: example on a cyclic graph

push start

$Q = [A]$

pop A

$Q = []$

push C

$Q = [C]$

push D

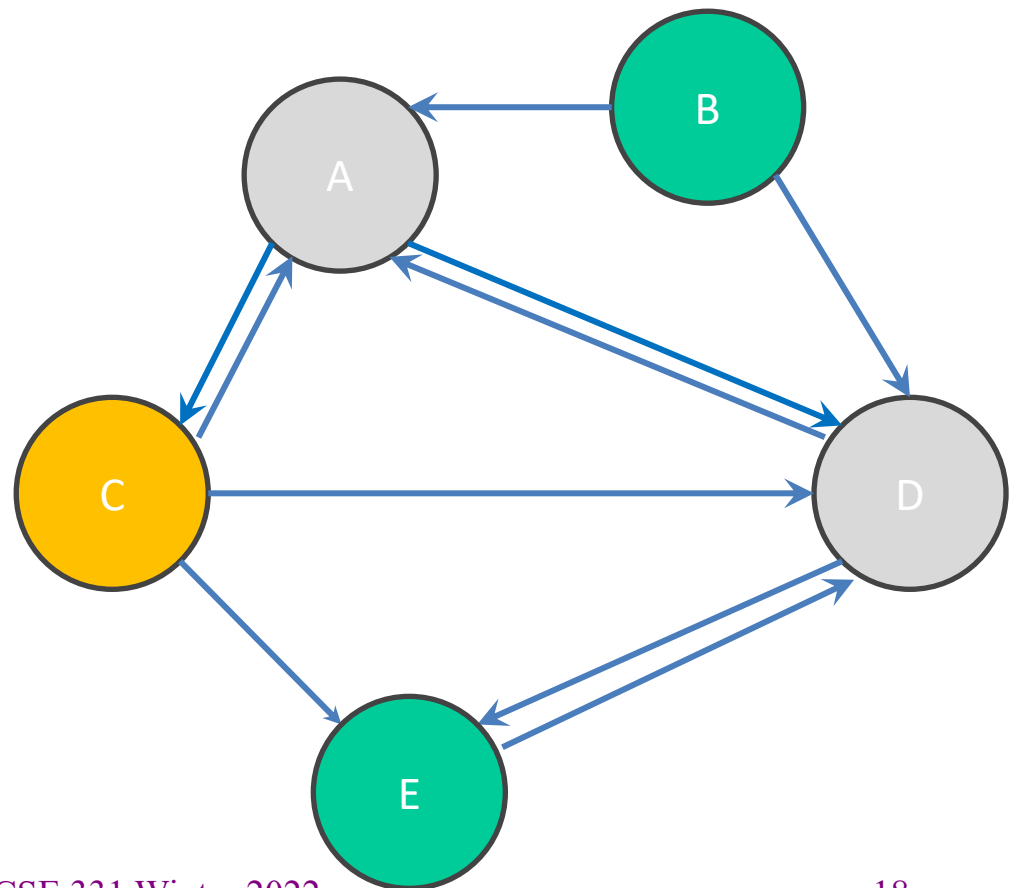
$Q = [D, C]$

pop C

$Q = [D]$

start = A

goal = B



BFS: example on a cyclic graph

push start

$Q = [A]$

pop A

$Q = []$

push C

$Q = [C]$

push D

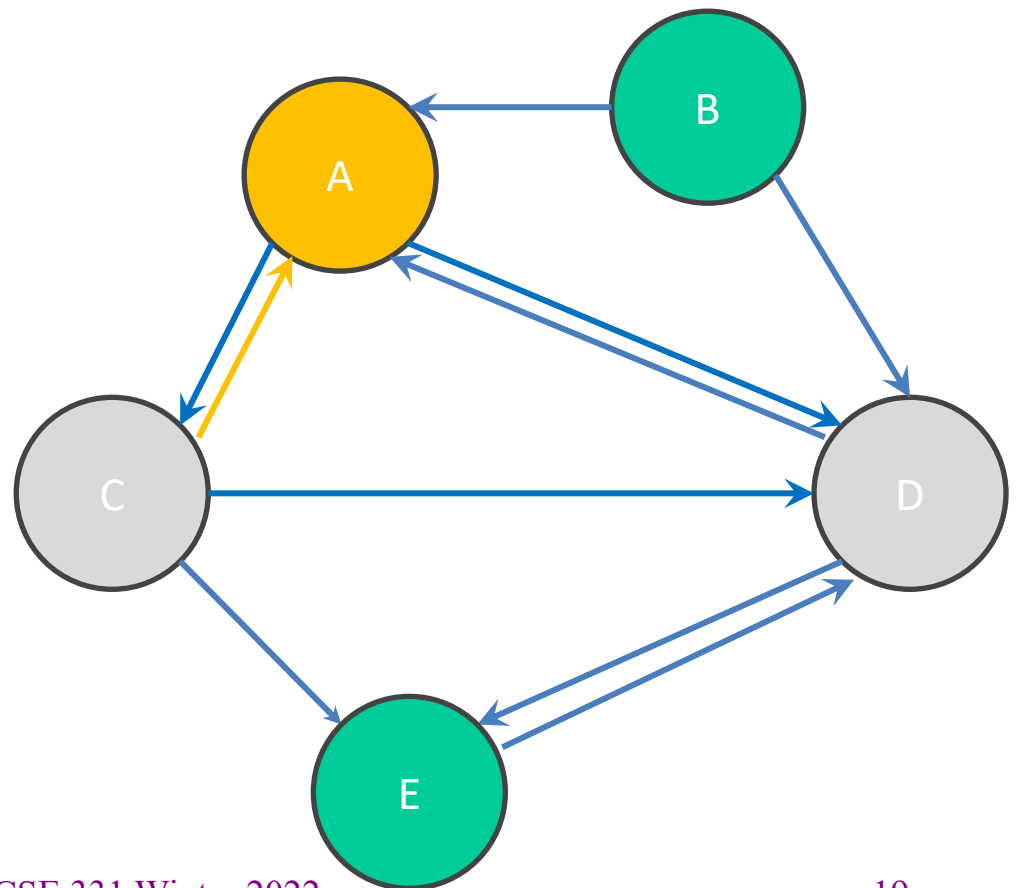
$Q = [D, C]$

pop C

$Q = [D]$

start = A

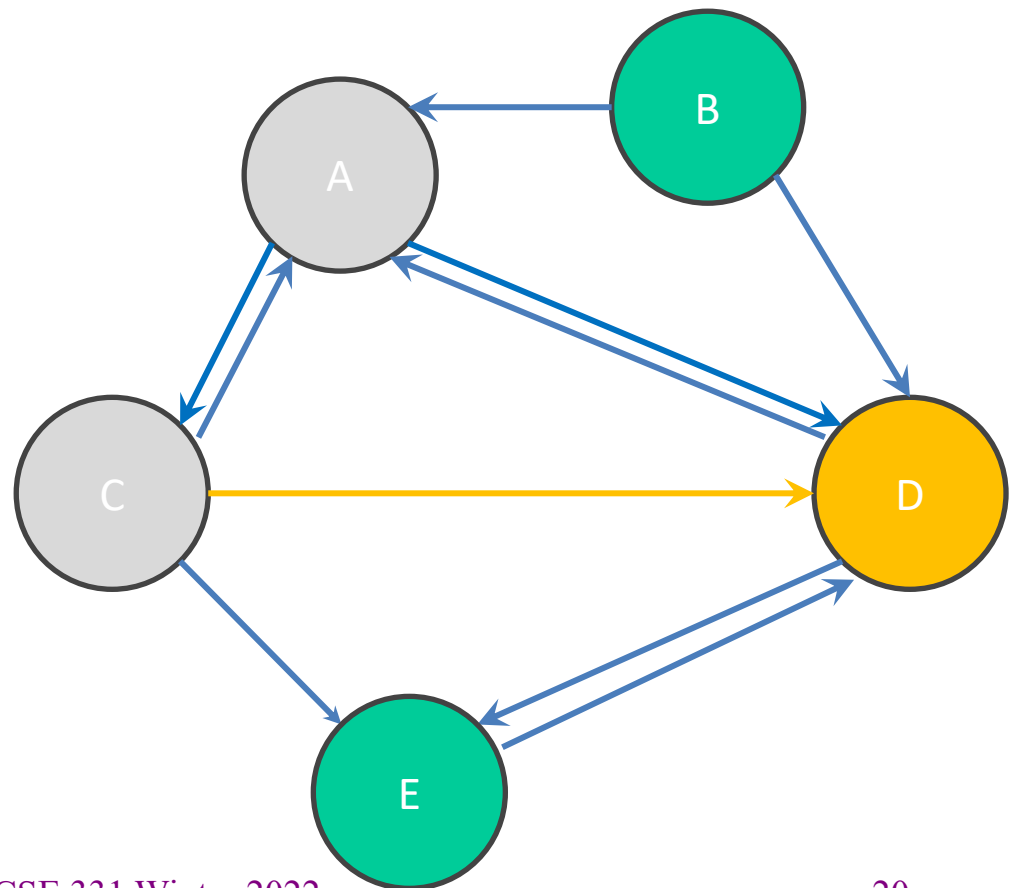
goal = B



BFS: example on a cyclic graph

push start $Q = [A]$
pop A $Q = []$
push C $Q = [C]$
push D $Q = [D, C]$
pop C $Q = [D]$

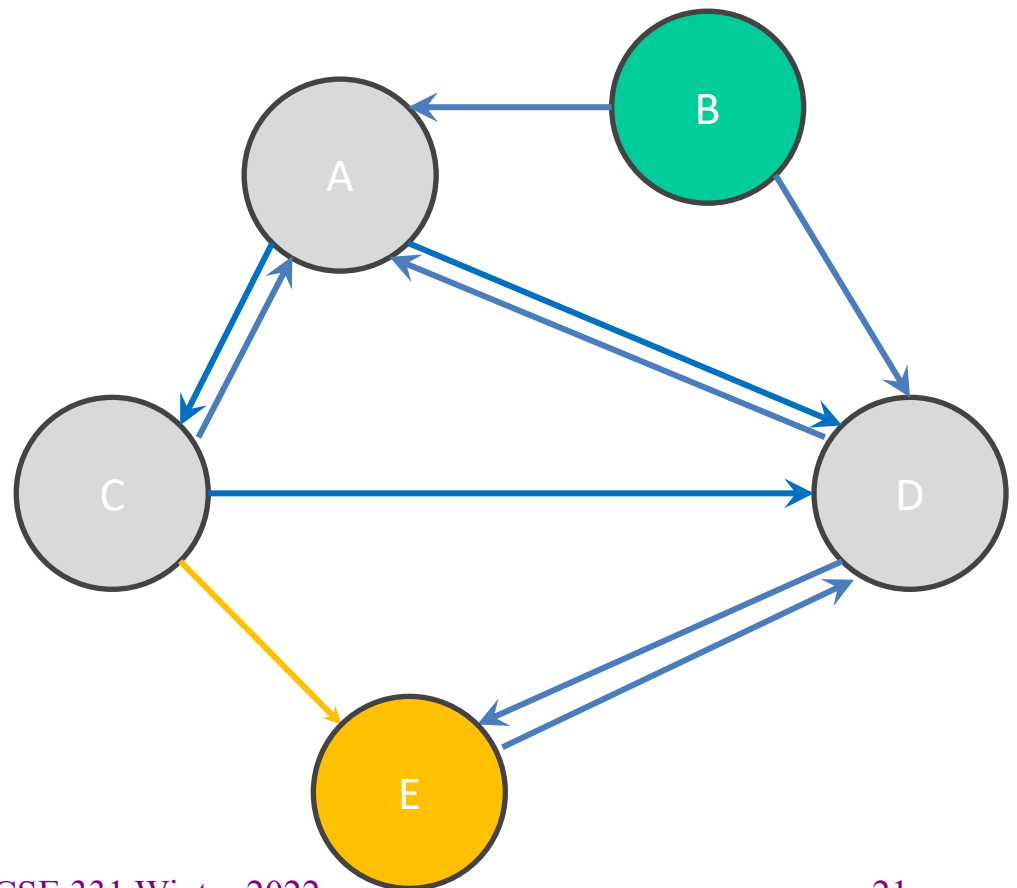
start = A
goal = B



BFS: example on a cyclic graph

push start $Q = [A]$
pop A $Q = []$
push C $Q = [C]$
push D $Q = [D, C]$
pop C $Q = [D]$
push E $Q = [E, D]$

start = A
goal = B



BFS: example on a cyclic graph

push start

$Q = [A]$

pop A

$Q = []$

push C

$Q = [C]$

push D

$Q = [D, C]$

pop C

$Q = [D]$

push E

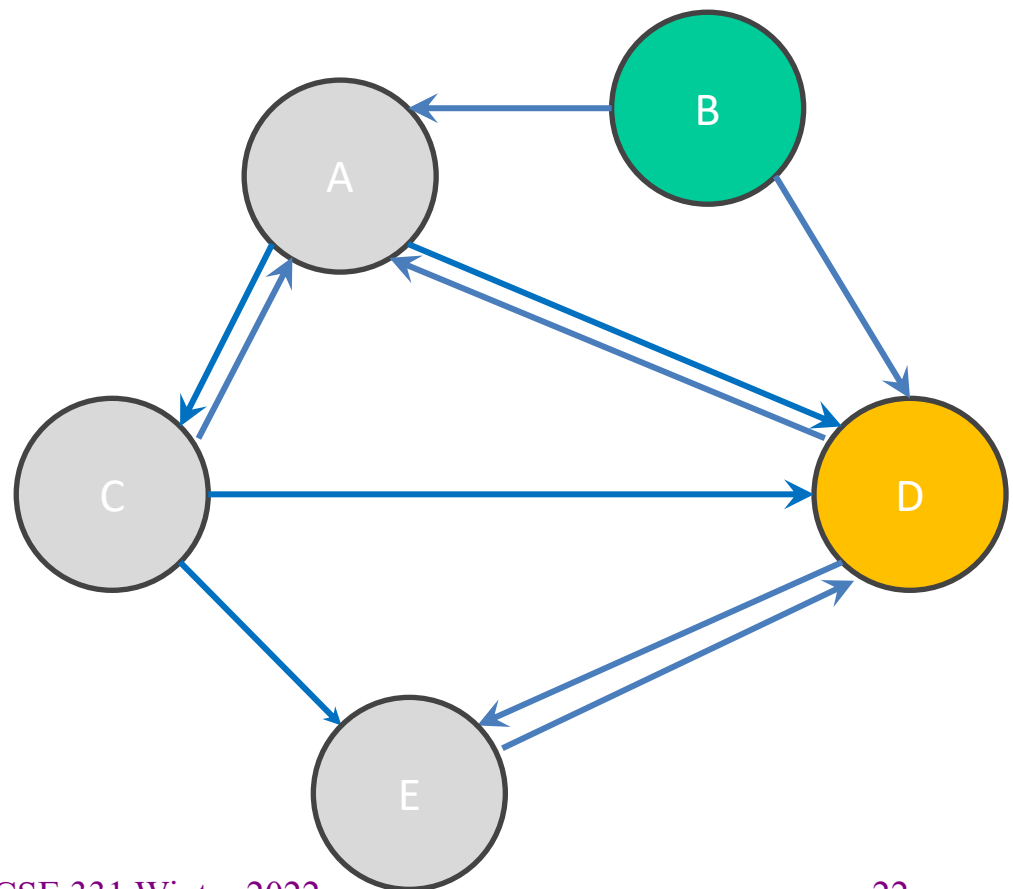
$Q = [E, D]$

pop D

$Q = [E]$

start = A

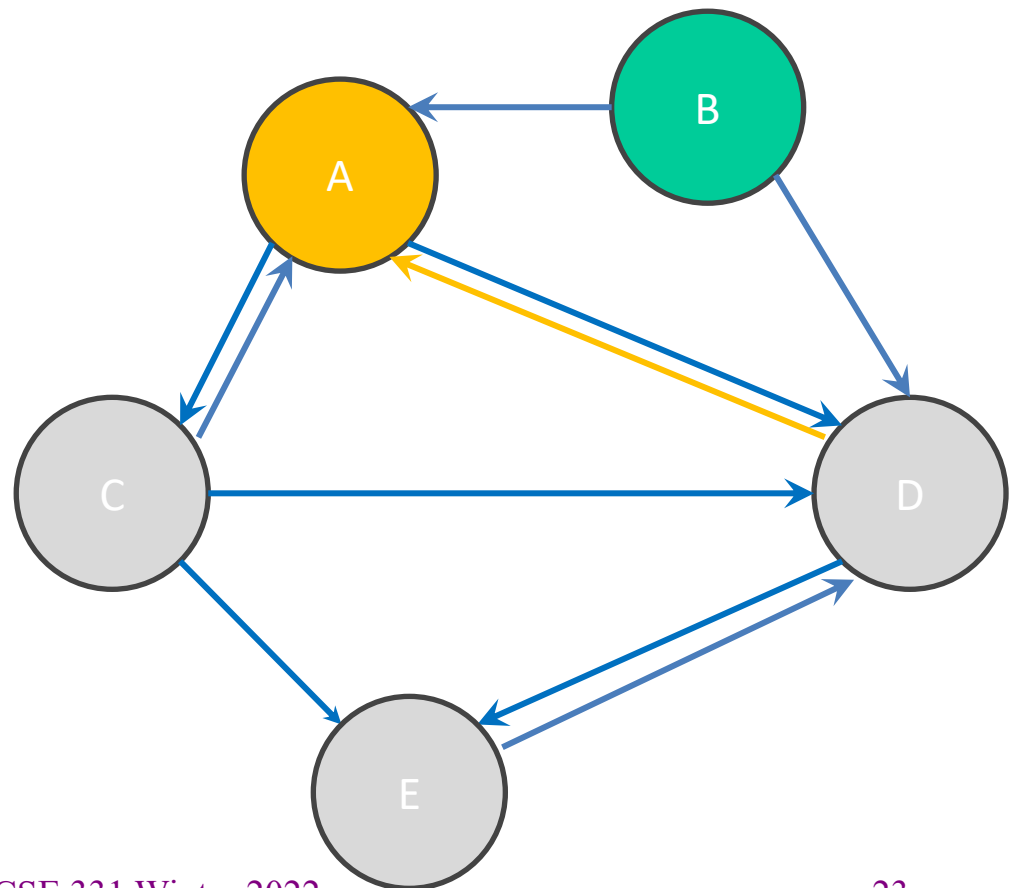
goal = B



BFS: example on a cyclic graph

push start $Q = [A]$
pop A $Q = []$
push C $Q = [C]$
push D $Q = [D, C]$
pop C $Q = [D]$
push E $Q = [E, D]$
pop D $Q = [E]$

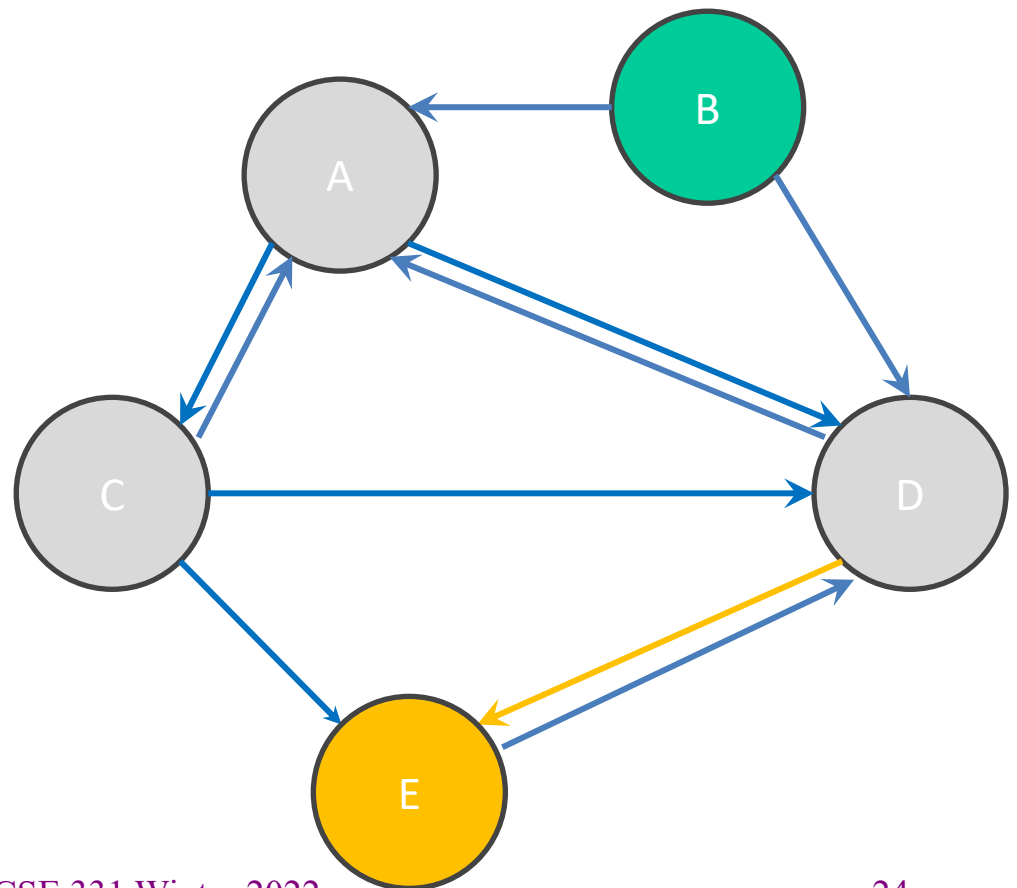
start = A
goal = B



BFS: example on a cyclic graph

push start $Q = [A]$
pop A $Q = []$
push C $Q = [C]$
push D $Q = [D, C]$
pop C $Q = [D]$
push E $Q = [E, D]$
pop D $Q = [E]$

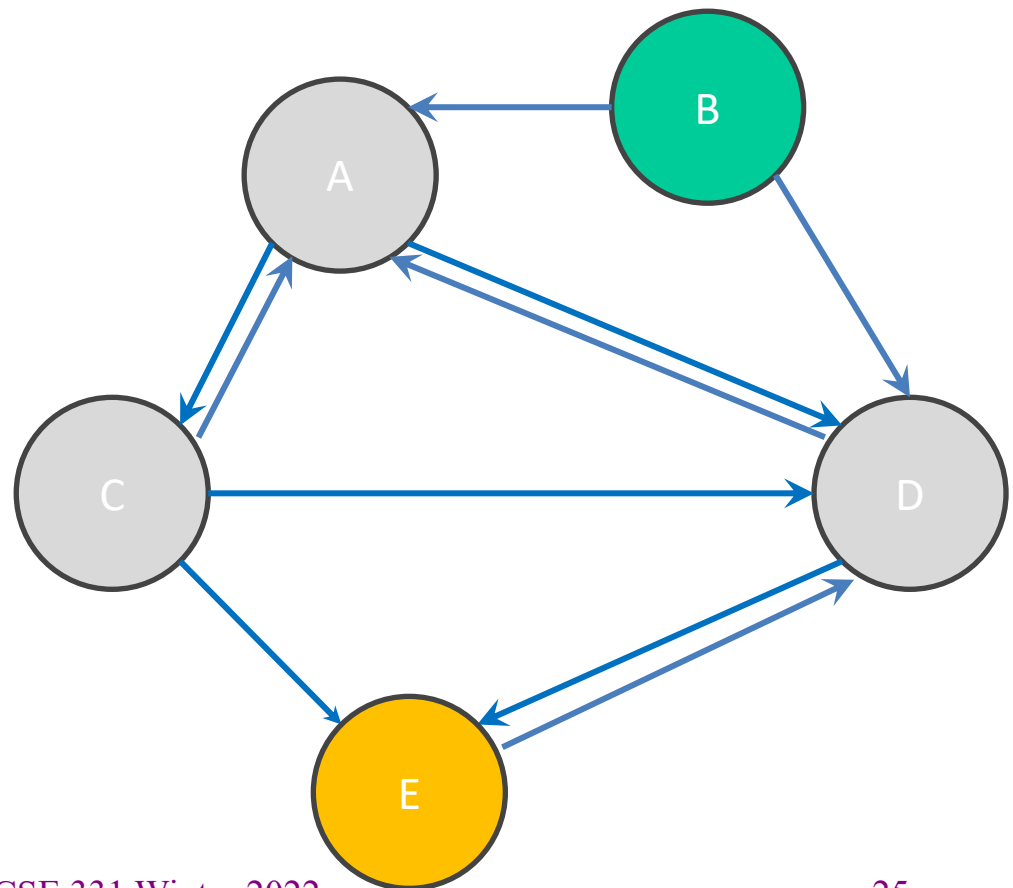
start = A
goal = B



BFS: example on a cyclic graph

push start	$Q = [A]$
pop A	$Q = []$
push C	$Q = [C]$
push D	$Q = [D, C]$
pop C	$Q = [D]$
push E	$Q = [E, D]$
pop D	$Q = [E]$
pop E	$Q = []$

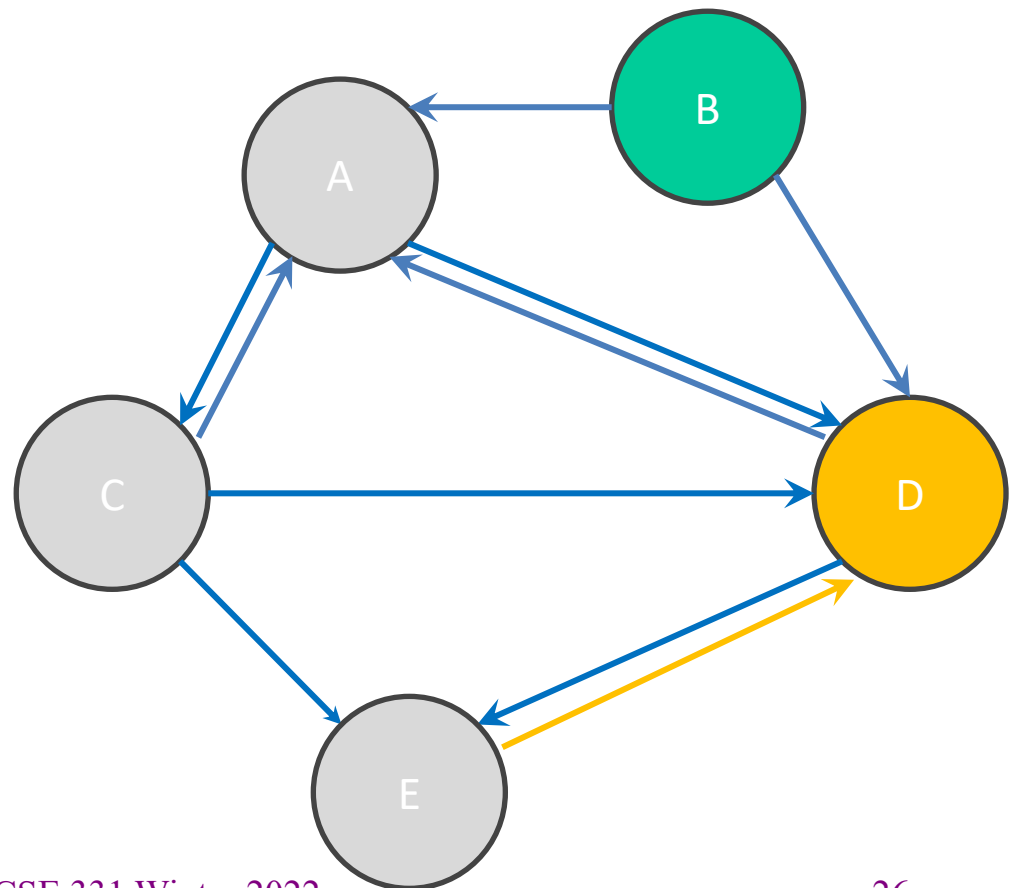
start = A
goal = B



BFS: example on a cyclic graph

push start	$Q = [A]$
pop A	$Q = []$
push C	$Q = [C]$
push D	$Q = [D, C]$
pop C	$Q = [D]$
push E	$Q = [E, D]$
pop D	$Q = [E]$
pop E	$Q = []$

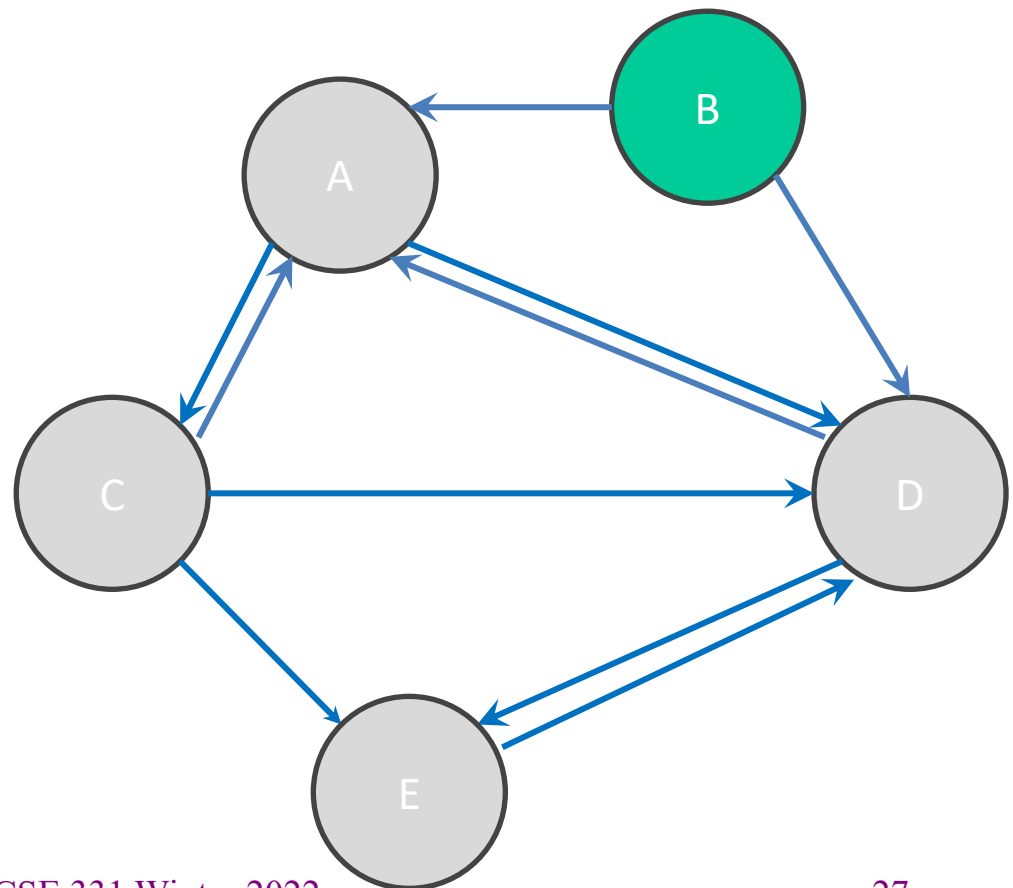
start = A
goal = B



BFS: example on a cyclic graph

push start $Q = [A]$
pop A $Q = []$
push C $Q = [C]$
push D $Q = [D, C]$
pop C $Q = [D]$
push E $Q = [E, D]$
pop D $Q = [E]$
pop E $Q = []$
return false

start = A
goal = B



Your turn!

Try running through the BFS algorithm on the worksheet.

BFS Reminders

- BFS is done on a graph, not inside the graph
 - This is why we have you create a **MarvelPaths** class!
- We will eventually want to allow other kinds of searches to be done on the graph, so BFS should not be hard-wired into the core Graph ADT
- Use the debug flag to turn off expensive **checkRep** for testing/grading

Outline of the assignment

0. Understand the dataset (`marvel.csv`) and CSV format
- 1. Complete `MarvelParser` class to read CSV-formatted files**
2. Implement graph initialization in `MarvelPaths` class
3. Implement path-finding via BFS in `MarvelPaths` class
4. Write suites of script tests and of implementation tests
 - Implement `MarvelTestDriver` for new test-script commands
5. Write `main` method in `MarvelPaths` for command-line usage

Reading in data

- Datasets are easily organized like a table or spreadsheet.
 - Each line is a row (*i.e.*, entry) in the dataset
 - Special characters usually separate the columns (*i.e.*, fields) of an entry
 - **Note:** fields can contain spaces
- One common data format: CSV (Comma-Separated Values)
 - Columns are separated by commas (',')

Structure of a CSV dataset

- First line of the CSV just names the fields of dataset entries.
- An example dataset in CSV format:

```
name,email
```

```
Kevin Zatloukal,kevinz@cs.uw.edu
```

```
Hal Perkins,perkins@cs.uw.edu
```

```
Mike Ernst,mernst@cs.uw.edu
```

```
Zachary Tatlock,ztatlock@cs.uw.edu
```

```
Dan Grossman,djg@cs.uw.edu
```


Parsing datasets

- Since datasets are structured, **we can interpret and parse the dataset programmatically.**
- For this assignment, we will do the file parsing for you (in a List of type String), but you need to store the results inside your graph.
- How should our graph be structured? What are the nodes? What are the edges?

Outline of the assignment

0. Understand the dataset (`marvel.csv`) and CSV format
1. Complete `MarvelParser` class to read CSV-formatted files
2. Implement graph initialization in `MarvelPaths` class
3. Implement path-finding via BFS in `MarvelPaths` class
4. **Write suites of script tests and of implementation tests**
 - **Implement `MarvelTestDriver` for new test-script commands**
5. Write `main` method in `MarvelPaths` for command-line usage

Script testing in HW6

- **Same test-script mechanism from HW5, but 2 new commands!**
 - New command **LoadGraph** to read and initialize graph from CSV file
 - New command **FindPath** to find shortest path in graph using BFS
- Must write the test driver (**MarvelTestDriver**) yourself
 - But you can copy/inherit most of it from **GraphTestDriver** in HW5

Command (in <i>foo.test</i>)	Output (in <i>foo.expected</i>)
LoadGraph <i>name file.csv</i>	loaded graph <i>name</i>
FindPath <i>graph node₁ node_n</i>	path from <i>node₁</i> to <i>node_n</i> : <i>node₁</i> to <i>node₂</i> via <i>edge_{1,2}</i> <i>node₂</i> to <i>node₃</i> via <i>edge_{2,3}</i> ... <i>node_{n-1}</i> to <i>node_n</i> via <i>edge_{n-1,n}</i>
...	...

LoadGraph and FindPath

- **LoadGraph** creates a *new* graph variable, much like **CreateGraph**
 - **LoadGraph** populates a graph with nodes and edges from dataset
 - **Note:** Other script commands (e.g., **AddNode**, **AddEdge**) can still mutate the graph once it has been loaded!
- **FindPath** breaks ties by lexicographic (alphabetic) order
 - Necessary when there are multiple shortest paths so the test output will be deterministic
 - **Sorting should not be implemented in your Graph ADT.** Lexicographic order should be done in BFS algorithm.
- **All this specified in detail on the homework's webpage**
 - You will need to read it to get things right :-)

Demo

A quick walkthrough of the starter code for HW6.

HW6 notes

- Read the assignment spec carefully!
 - Ensure that you are using the right file path in the right place to read the data file
 - Most common reason for failures during grading is incorrect file paths
- Helpful to test and debug using smaller datasets
 - Faster and easier to understand what's going on
- To run MarvelPaths or any program that does console I/O, use gradlew to run the desired gradle target *using the IntelliJ terminal window* (console I/O doesn't work right otherwise 🐛)
- When you are done, you will be able to find the shortest path from your command line!