

---

# CSE 331

# Software Design & Implementation

Fall 2020

Section 6 – HW6, Path-Finding, and Parsing

---

# Administrivia

---

- HW5 due tonight
- HW6 due next Wednesday.
- Any questions?

# Agenda

---

- Overview of HW6
- Breadth-first search (BFS)
- Parsing a file in comma-separated-values (CSV) format
  - Very similar to tab-separated-values (TSV) format in HW6
- Test scripts and the new test driver

# 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 tab-separated-values (TSV) 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.tsv`) and TSV format
1. Complete `MarvelParser` class to read TSV-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.tsv`) and TSV format
1. Complete `MarvelParser` class to read TSV-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 take

---

```
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
```

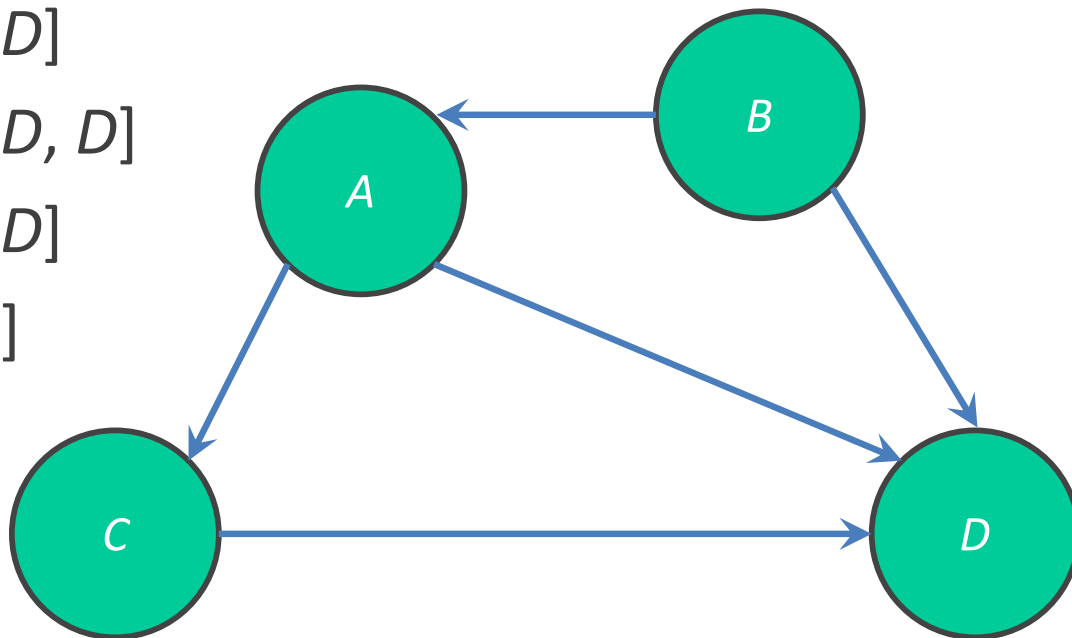
```
return false
```



# 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		

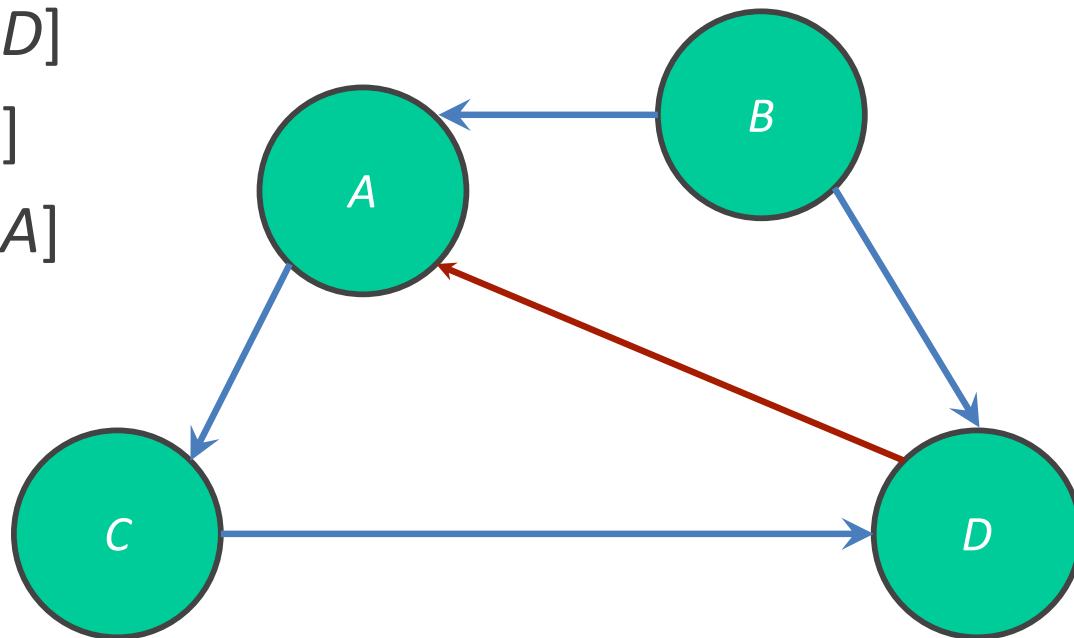


# BFS: example on a cyclic graph

---

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**

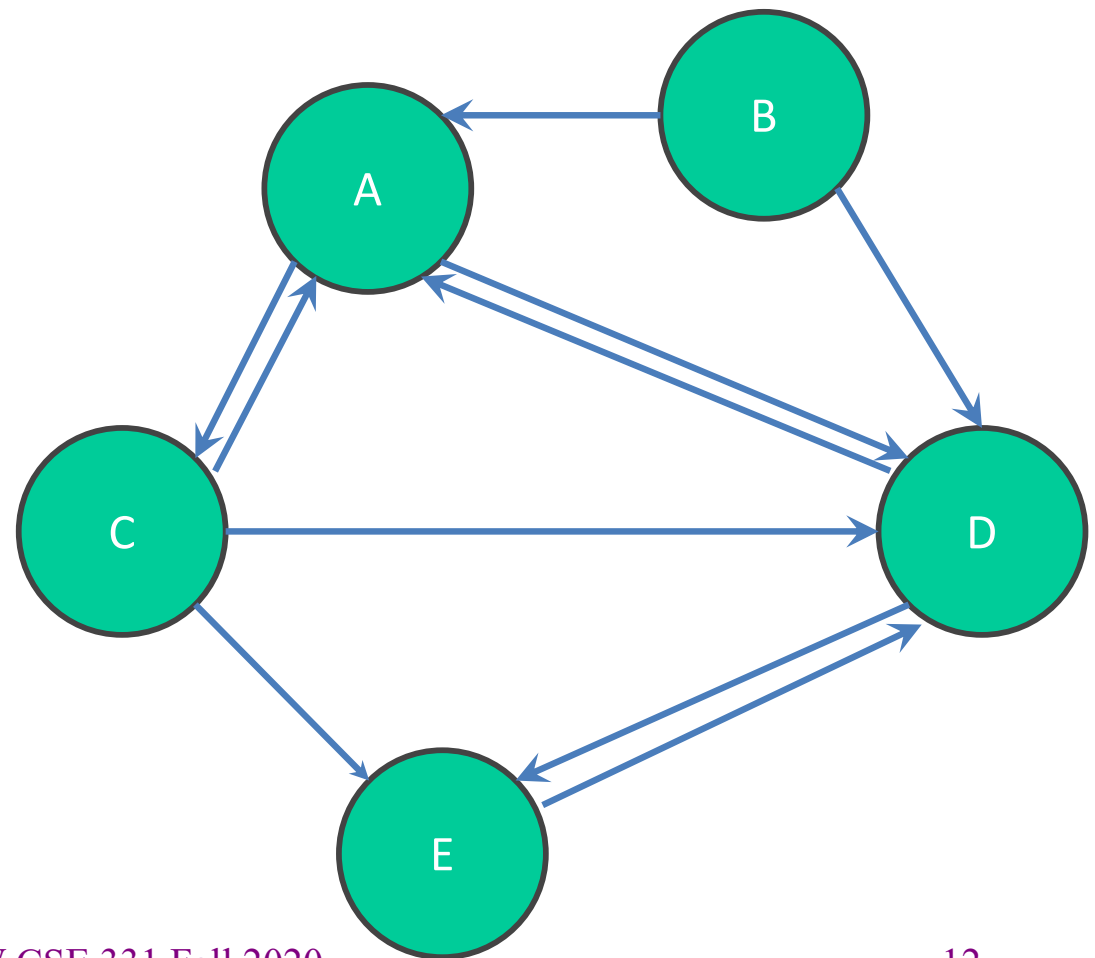
        push  $O$  onto queue

return false

# BFS: example on a cyclic graph

---

start = A  
goal = B



# BFS: example on a cyclic graph

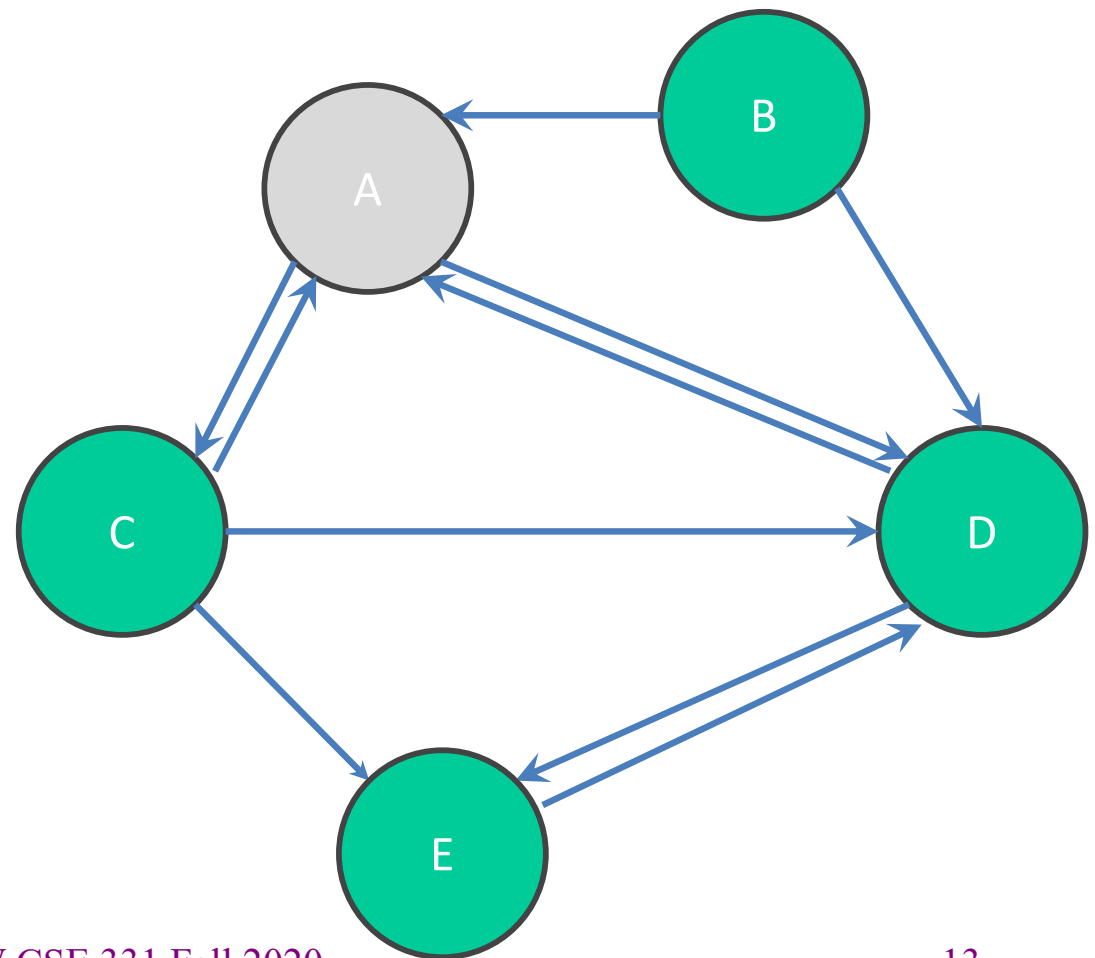
---

push start

$Q = [A]$

start = A

goal = B



# BFS: example on a cyclic graph

---

push start

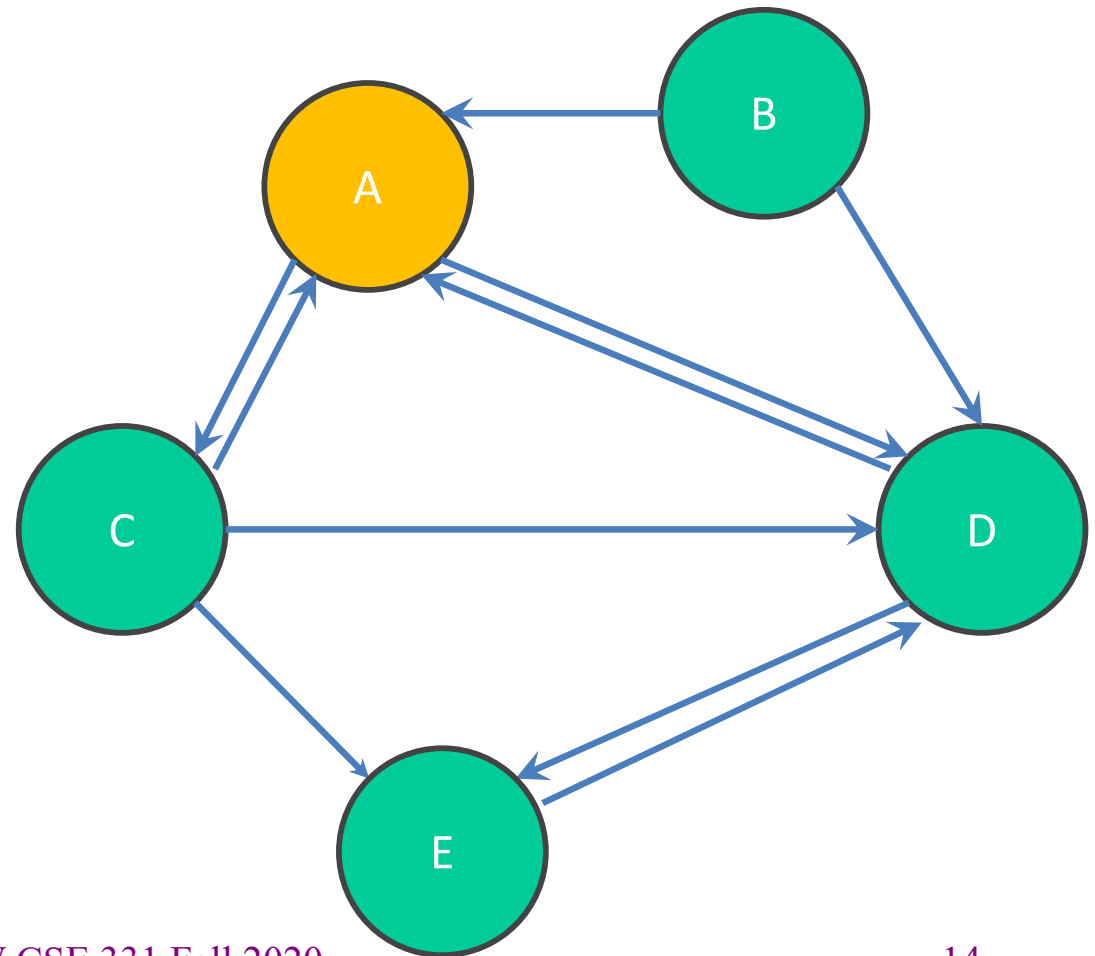
$Q = [A]$

start = A

pop A

$Q = []$

goal = B



# BFS: example on a cyclic graph

---

push start

$Q = [A]$

start = A

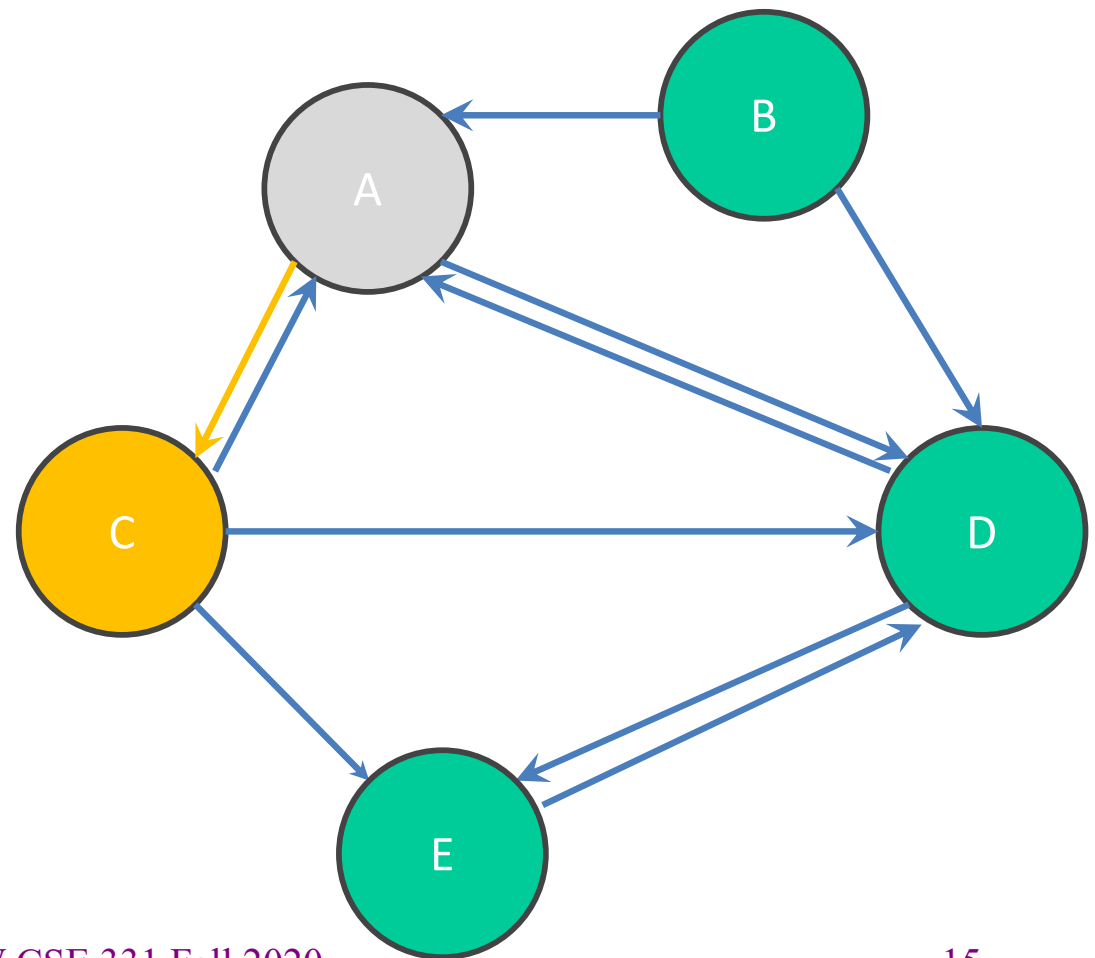
pop A

$Q = []$

goal = B

push C

$Q = [C]$

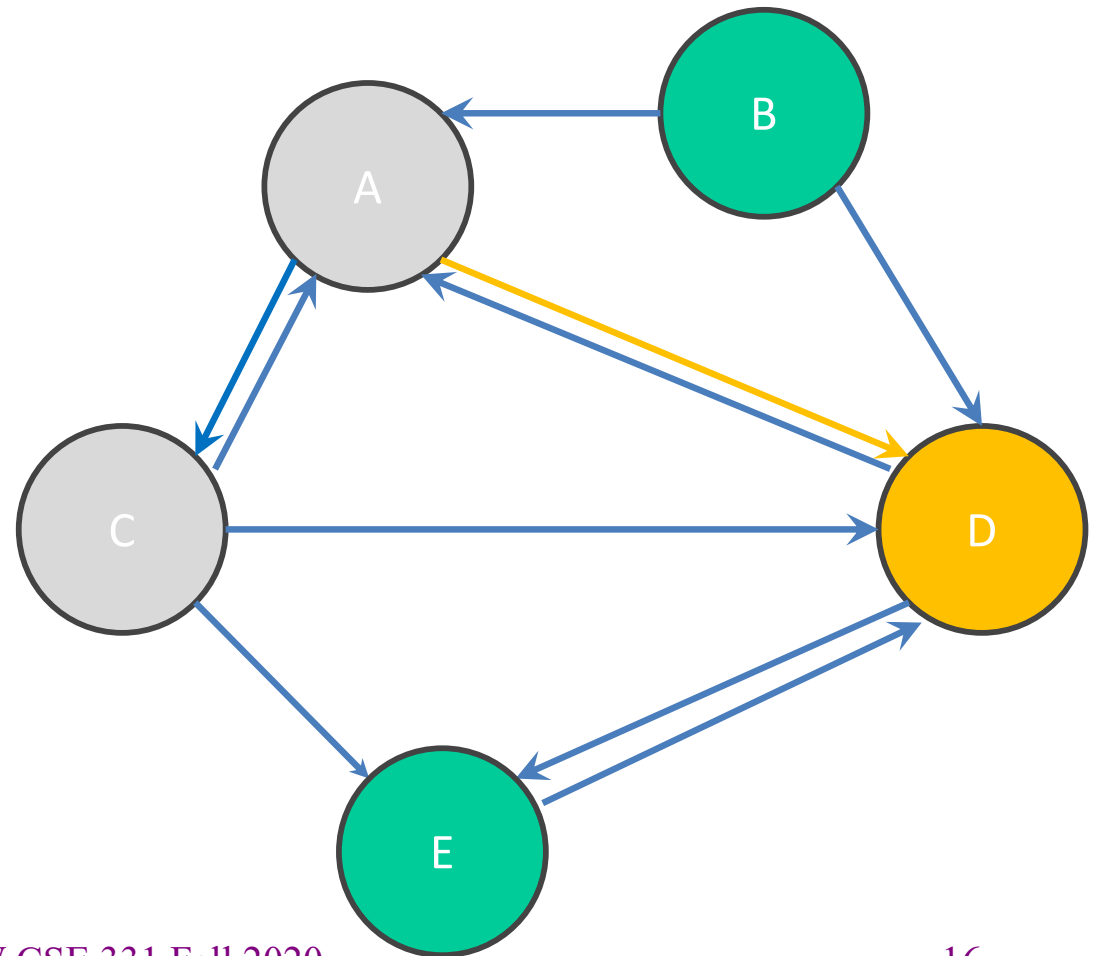


# BFS: example on a cyclic graph

---

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

$Q = [A]$

start = A

pop A

$Q = []$

goal = B

push C

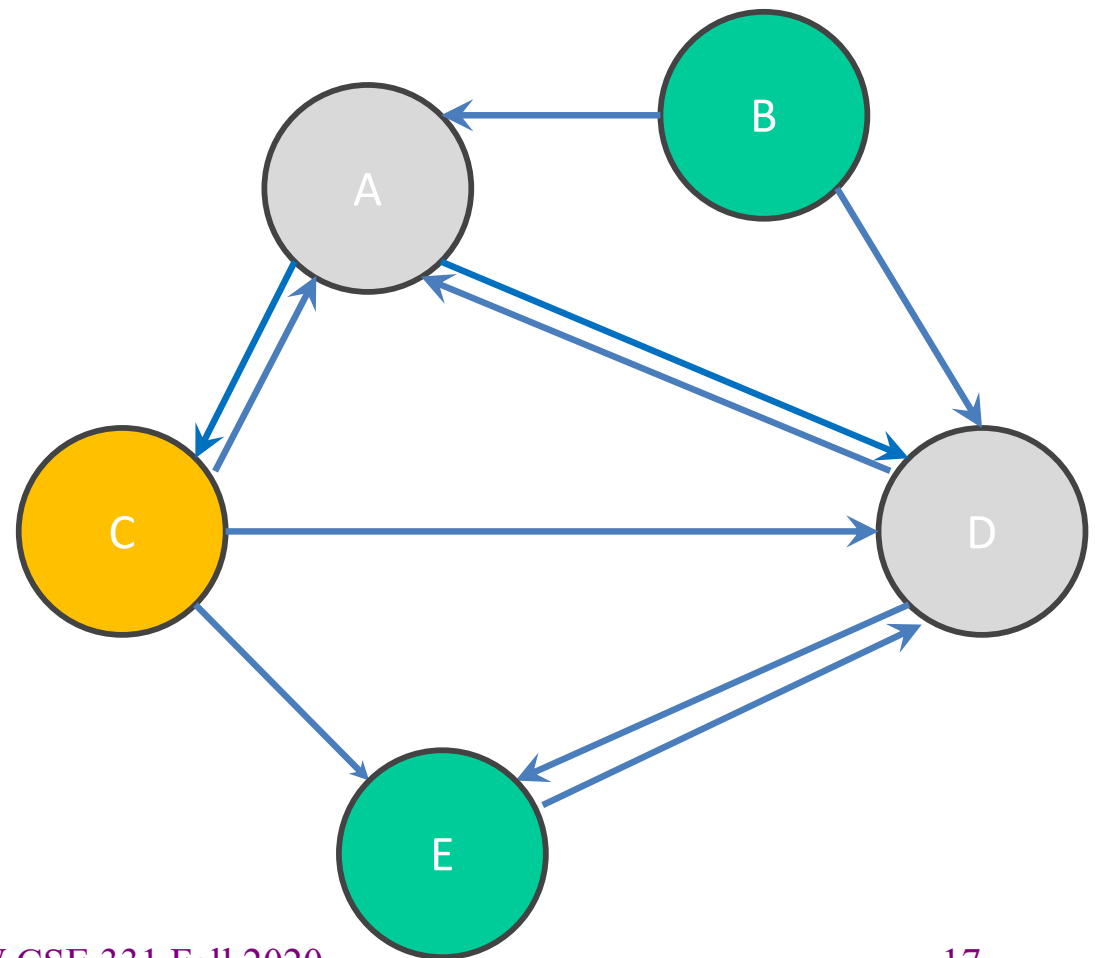
$Q = [C]$

push D

$Q = [D, C]$

pop C

$Q = [D]$

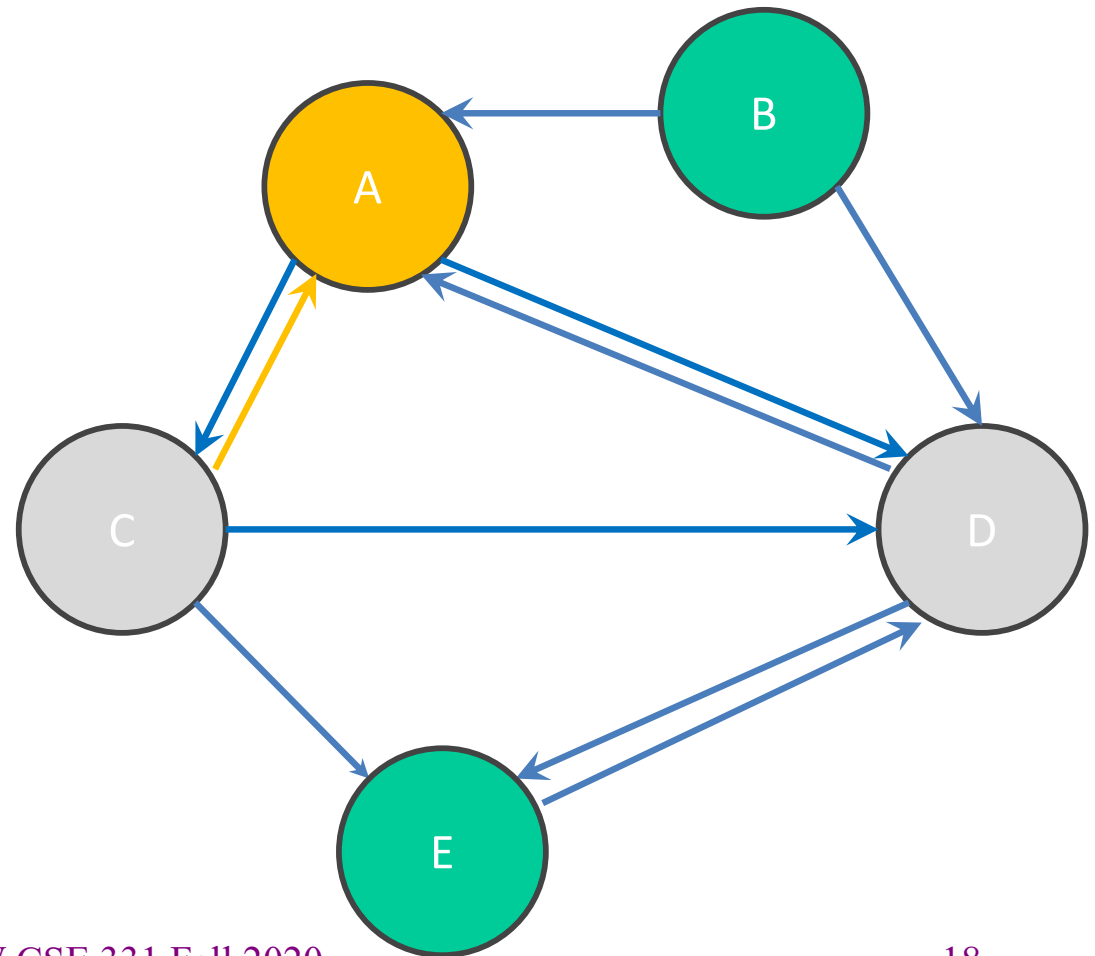


# BFS: example on a cyclic graph

---

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

start =  $A$   
goal =  $B$

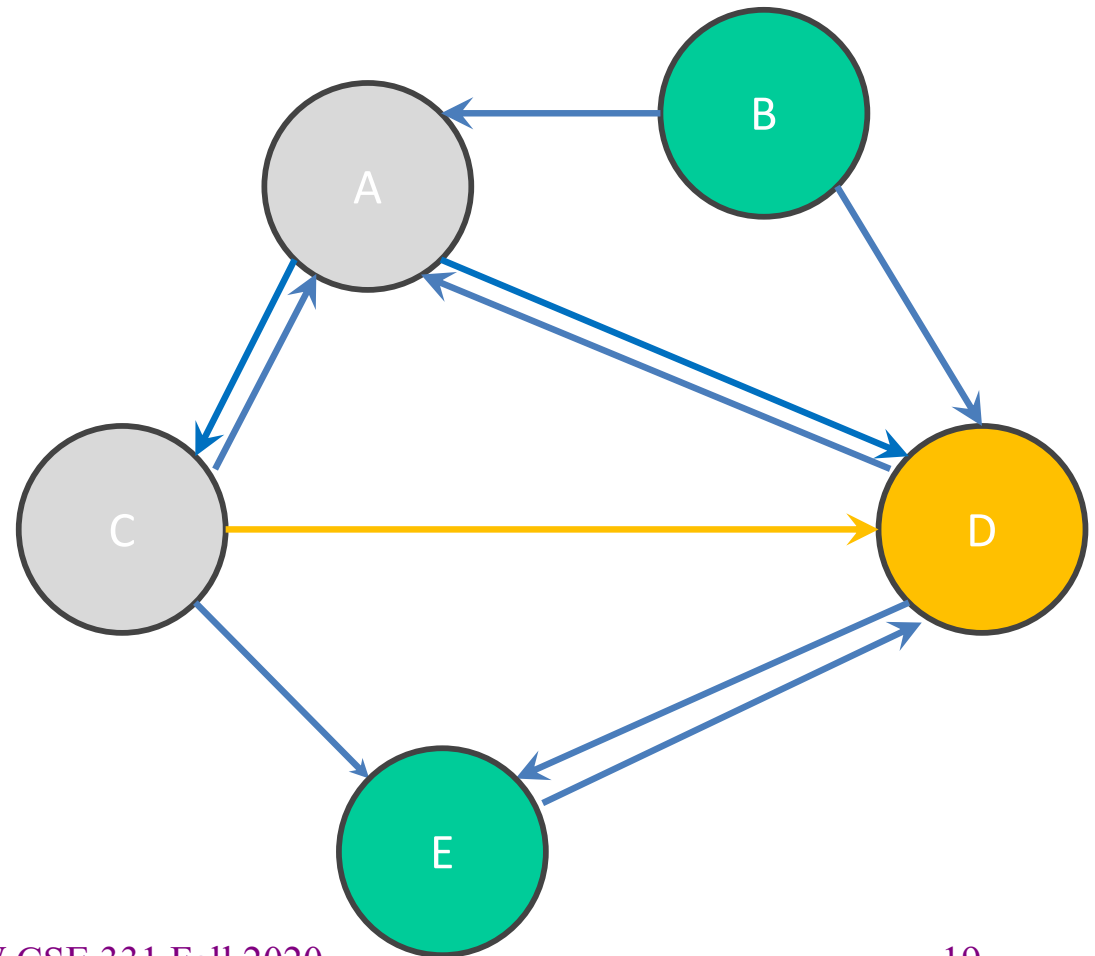


# BFS: example on a cyclic graph

---

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

start =  $A$   
goal =  $B$

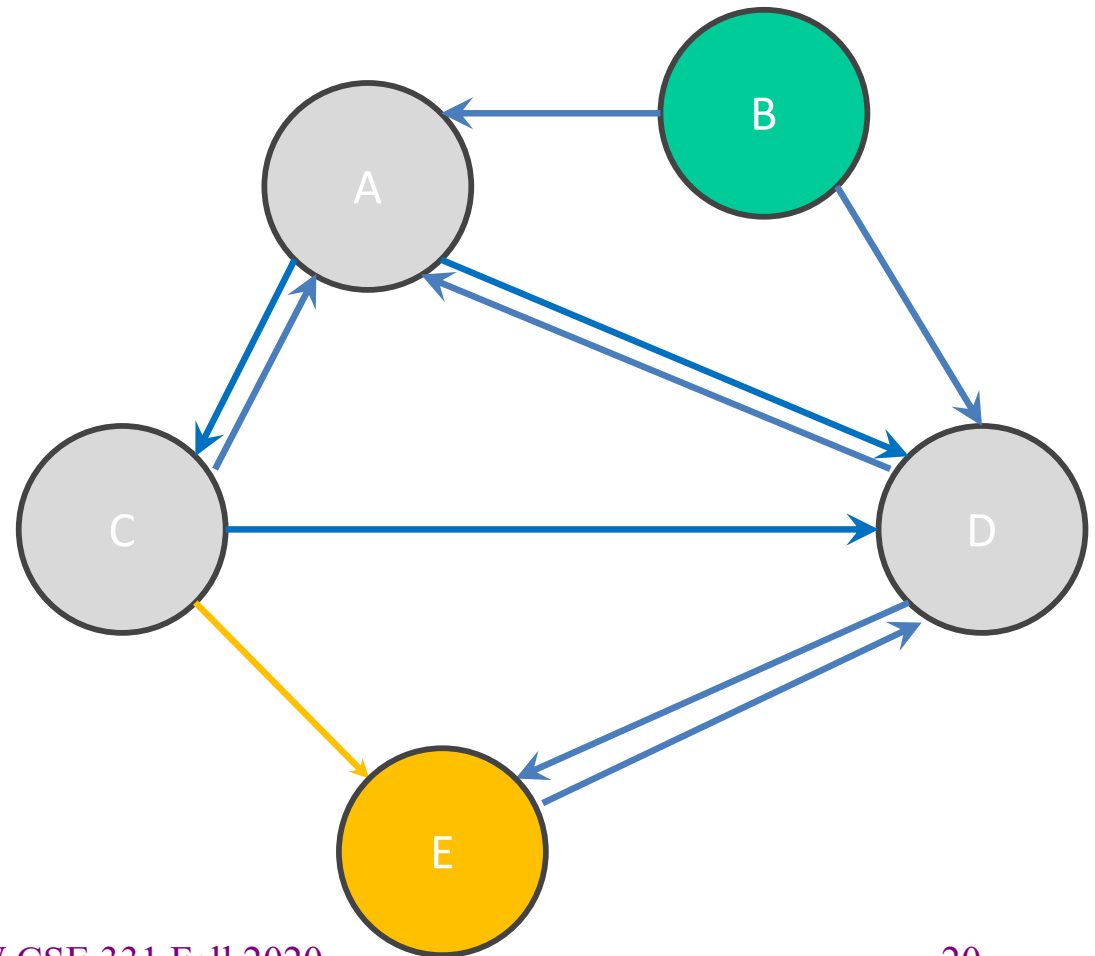


# BFS: example on a cyclic graph

---

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 =  $B$

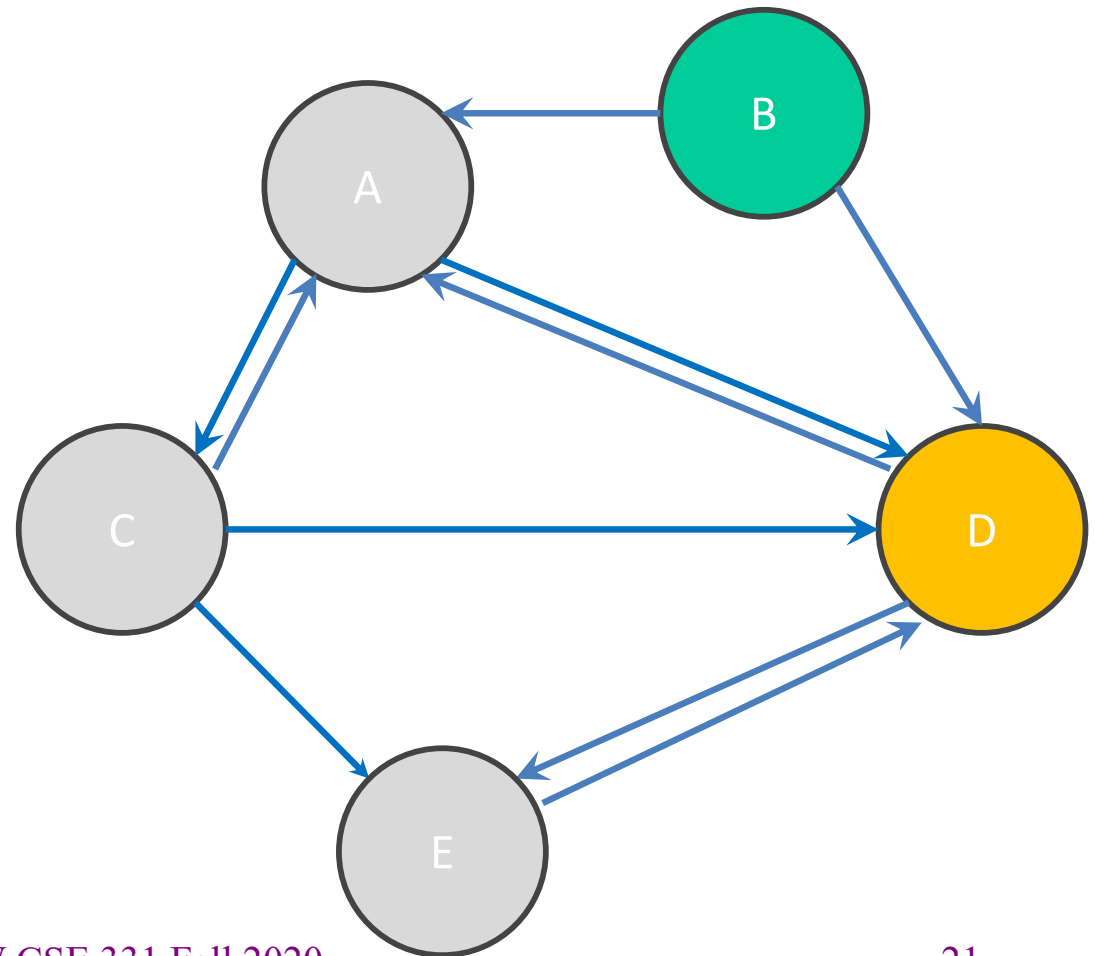


# BFS: example on a cyclic graph

---

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 =  $B$

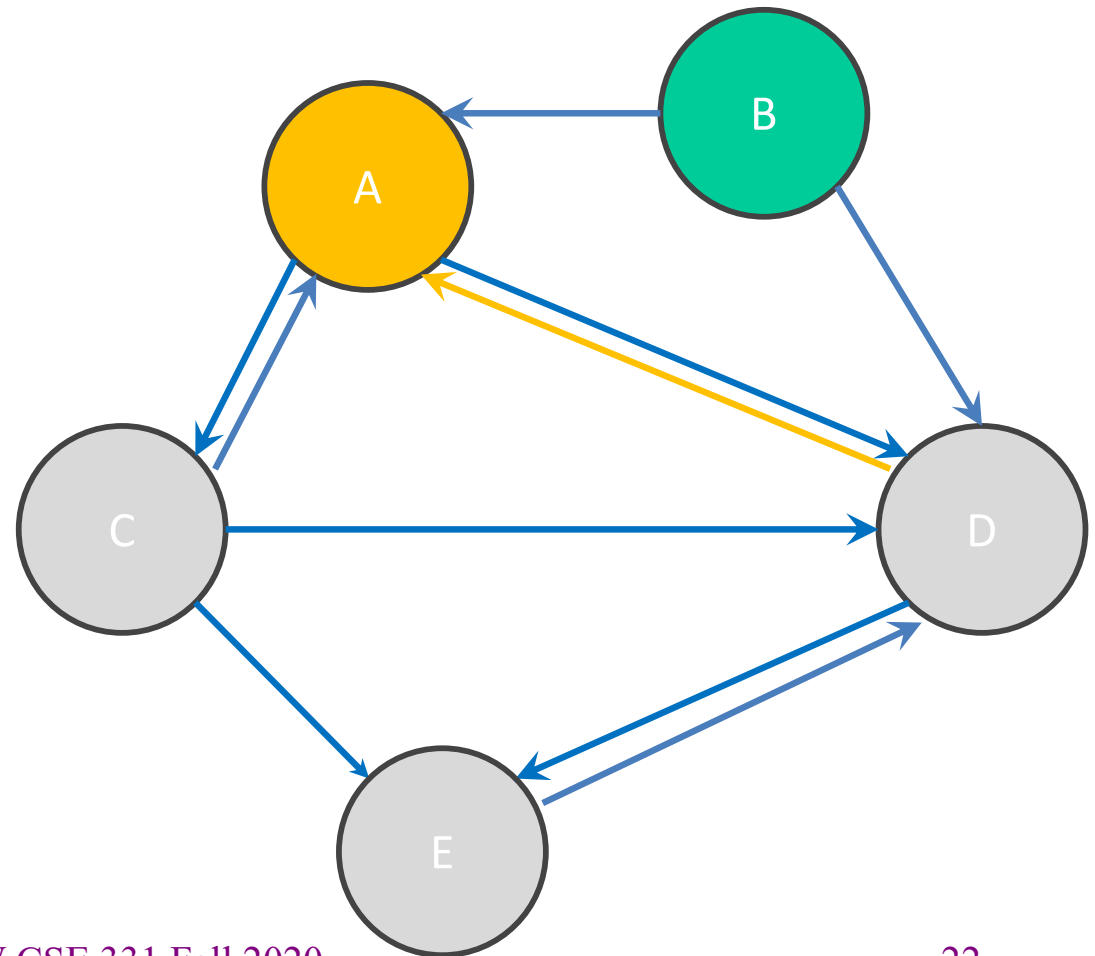


# BFS: example on a cyclic graph

---

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 =  $B$

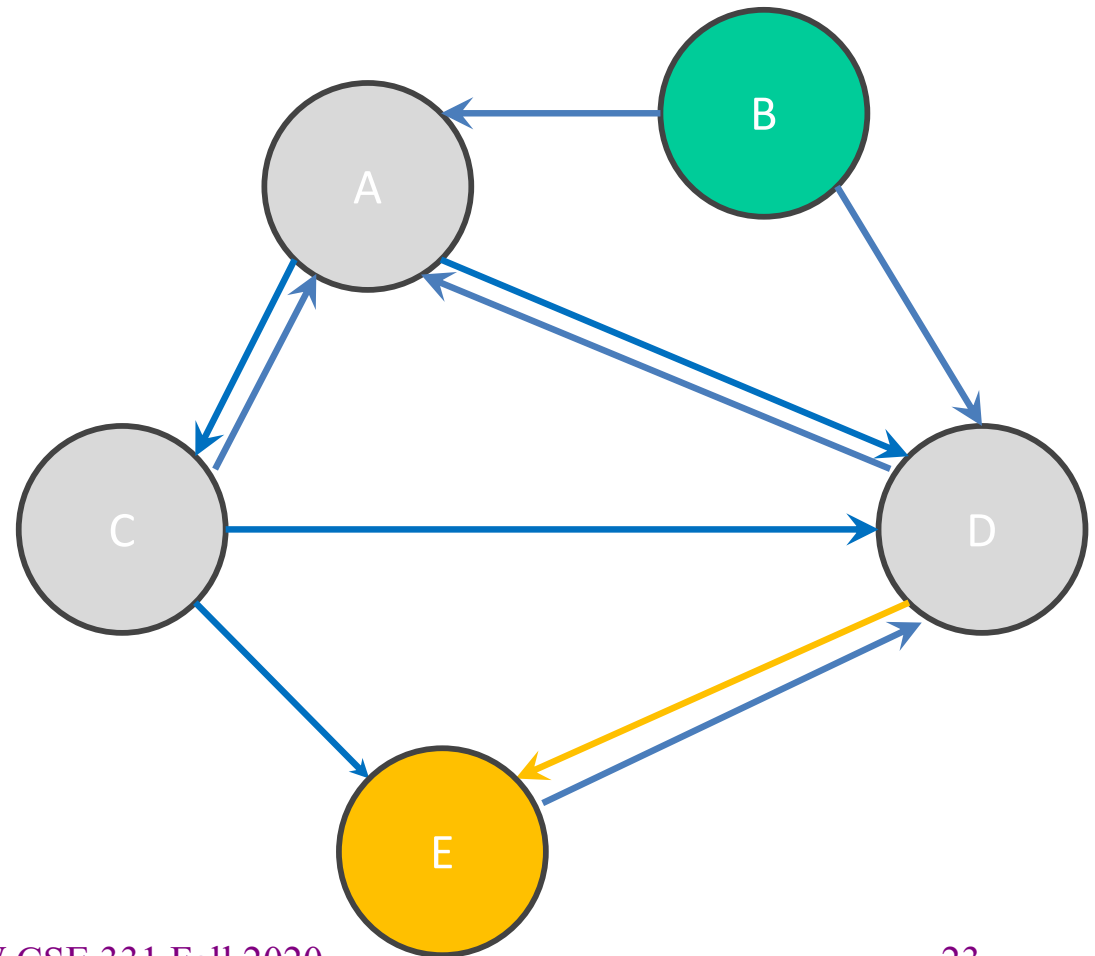


# BFS: example on a cyclic graph

---

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 =  $B$

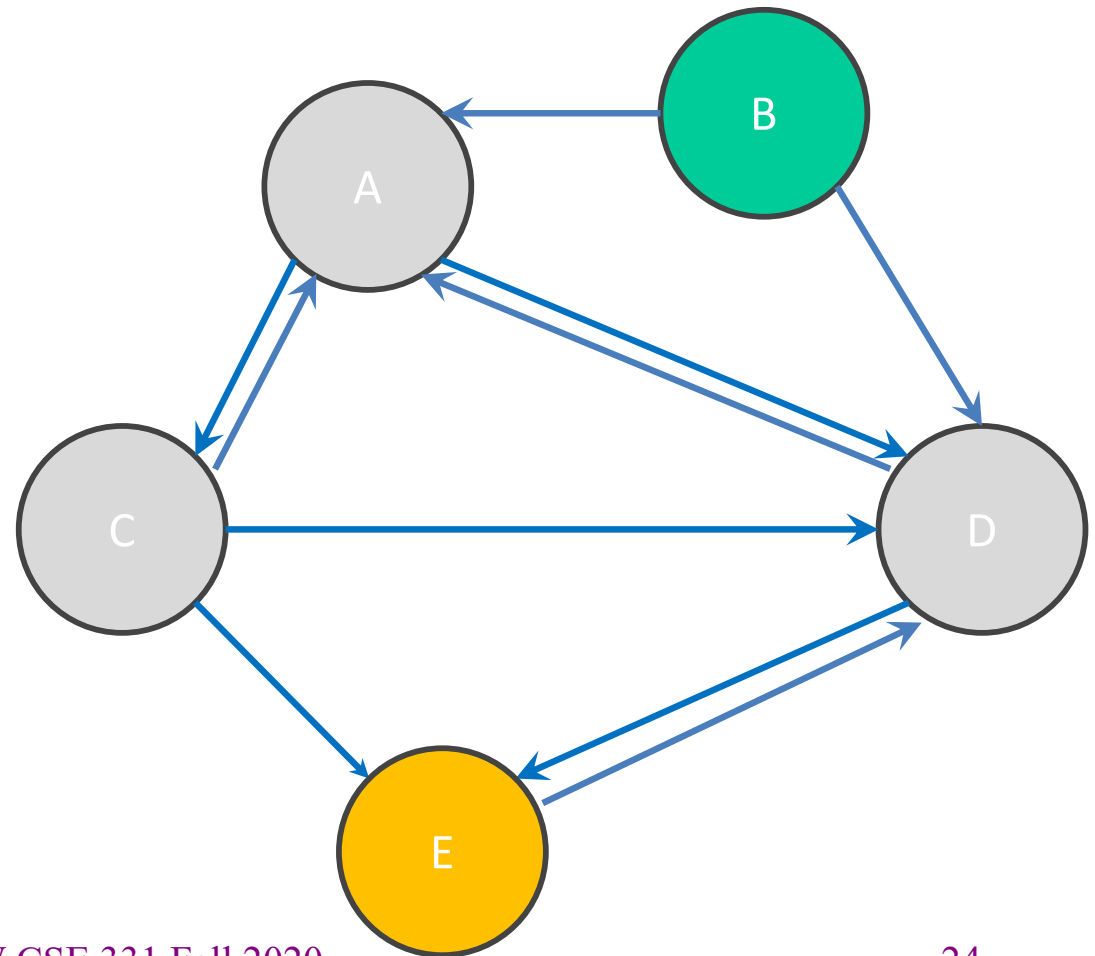


# BFS: example on a cyclic graph

---

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 =  $B$



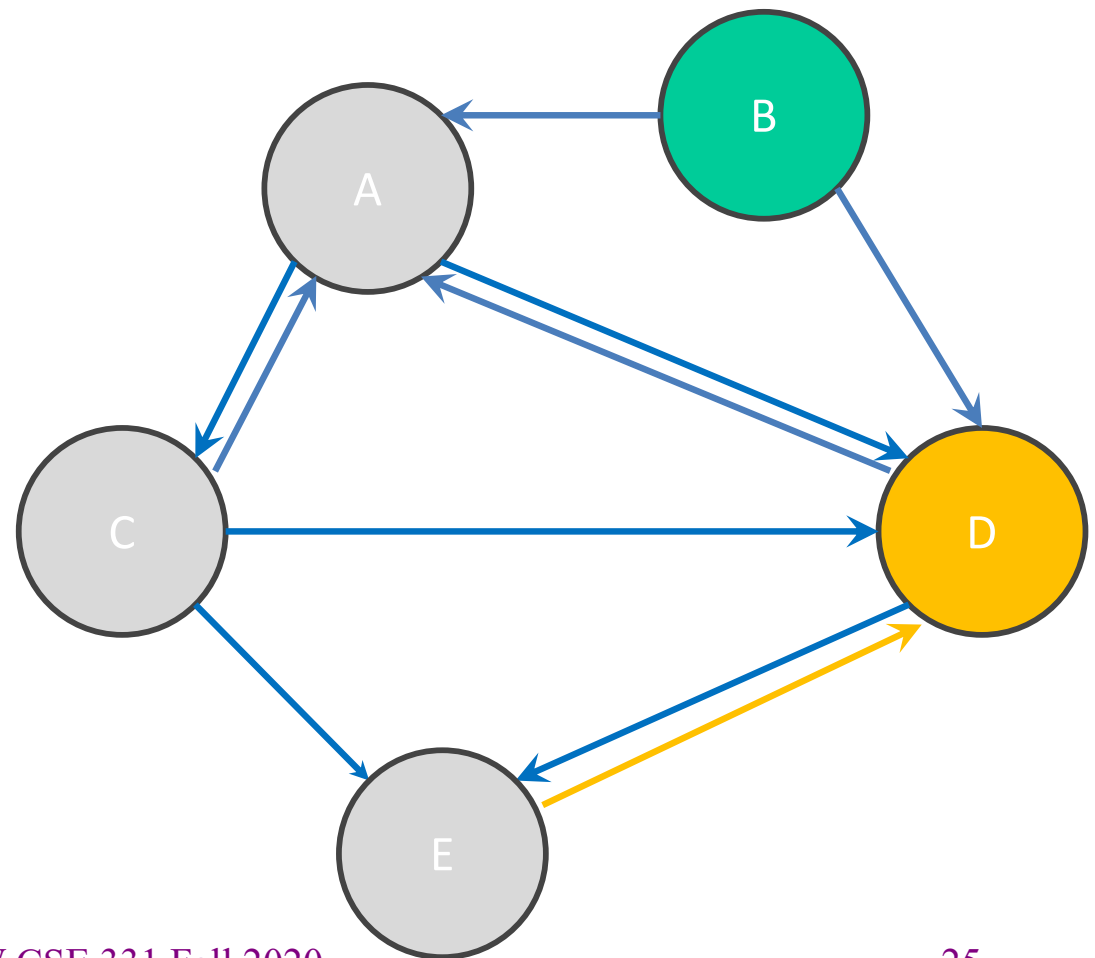


# BFS: example on a cyclic graph

---

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 =  $B$

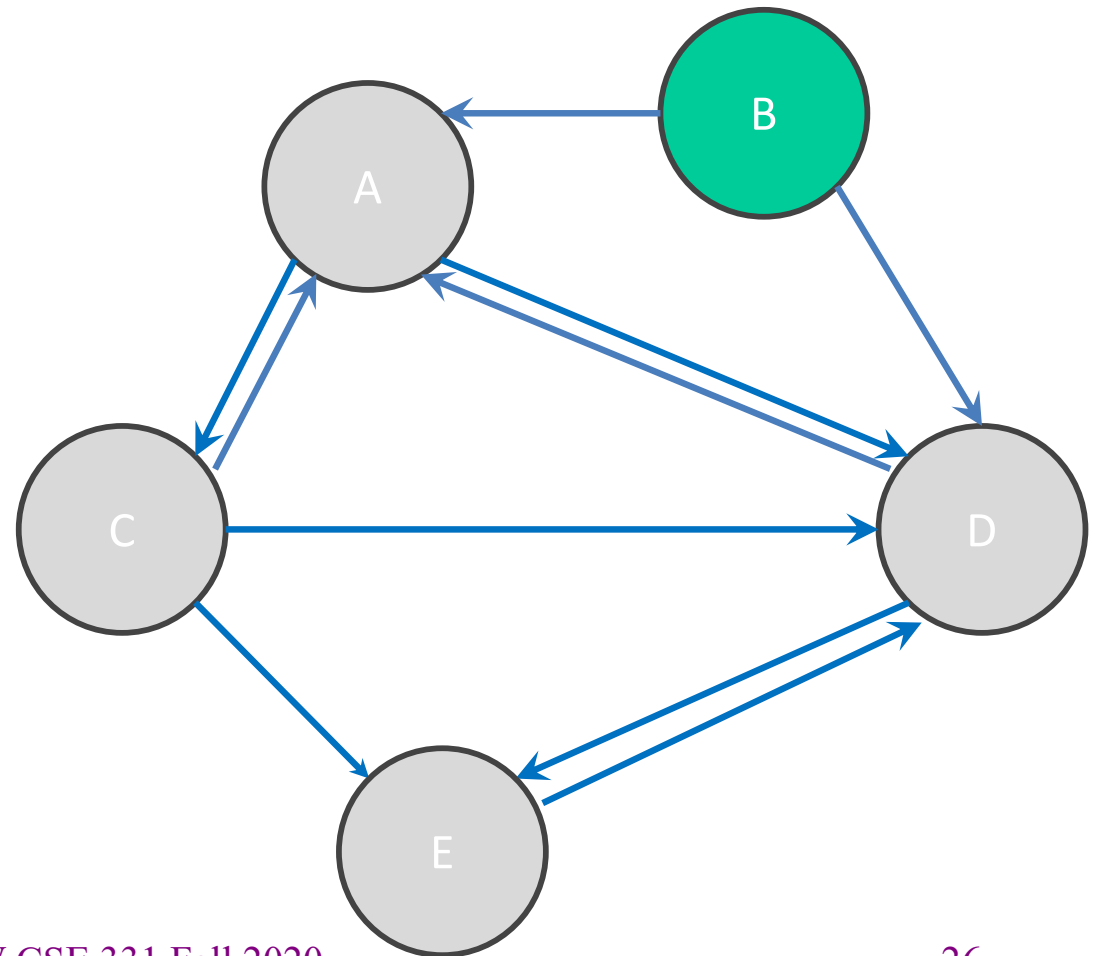


# BFS: example on a cyclic graph

---

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.tsv`) and TSV format
- 1. Complete `MarvelParser` class to read TSV-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 (',')
- For HW6, we will be using data formatted as TSV (Tab-Separated Values)
  - Columns are separated by tabs ('\t')

# 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.**
- Existing Java libraries already do this! No need to reinvent the wheel.
- For this class, we will be using the library OpenCSV as a parser.



# Dataset Parsers

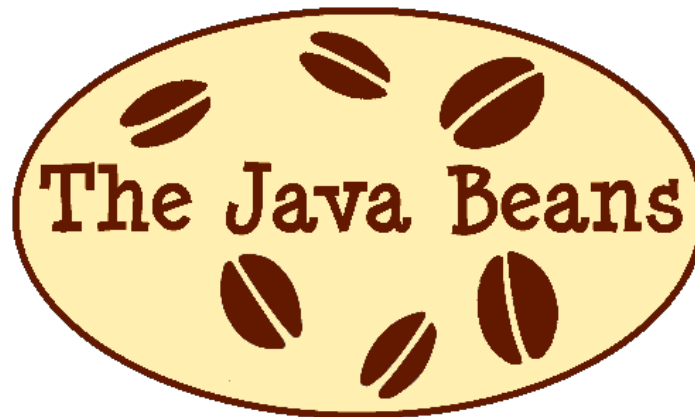
---

- OpenCSV needs to understand how your columns are structured to translate to Java code.
- Because rows have fixed columns, Java classes can be used to represent each row.
  - Each column is a field in the Java class.
- This class is known as a JavaBean!

# What is a JavaBean?

---

- A JavaBean is any class that...
  - has a public, zero-argument constructor
  - has several *properties*, *i.e.*, private fields each with getter and setter



# Example bean

---

```
public class UserModel {  
  
    private String name;  
  
    private String email;  
  
    public String getName() { return this.name; }  
    public void setName(String v) { this.name = v; }  
  
    public String getEmail() { return this.email; }  
    public void setEmail(String v) { this.email = v; }  
}
```

## **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
```

# Example bean (OpenCSV)

---

```
public class UserModel {  
    @CsvBindByName  
    private String name;
```

```
    @CsvBindByName
```

```
    private String email;
```

```
    public String getName() { return this.name; }  
    public void setName(String v) { this.name = v; }
```

```
    public String getEmail() { return this.email; }  
    public void setEmail(String v) { this.email = v; }
```

```
}
```

**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

# Example bean (OpenCSV)

---

```
public class UserModel {
```

```
    @CsvBindByName
```

```
    private String name;
```

```
    @CsvBindByName
```

```
    private String email;
```

```
    public String getName() { return this.name; }
```

```
    public void setName(String v) { this.name = v; }
```

```
    public String getEmail() { return this.email; }
```

```
    public void setEmail(String v) { this.email = v; }
```

```
}
```

Helps OpenCSV identify field names that match data column names

# From dataset to beans via OpenCSV

---

- OpenCSV converts each entry into an object of a chosen JavaBean class
- Returns an iterator to loop through each row of CSV!

```
// see hw spec for details on getting the BufferedReader
Reader reader = new BufferedReader(...);
```

```
Iterator<UserModel> csvUserIterator =
    new CsvToBeanBuilder<UserModel>(reader) // set input
        .withType(UserModel.class) // set entry type
        .withSeparator(',') // , for CSV
        .withIgnoreLeadingWhiteSpace(true)
        .build() // returns a CsvToBean<UserModel>
        .iterator();
```

# Demo

---

A quick walkthrough of the parser code for HW6.

# Outline of the assignment

---

0. Understand the dataset (`marvel.tsv`) and TSV format
1. Complete `MarvelParser` class to read TSV-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 TSV
  - 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> )
<b>LoadGraph</b> <i>name file.tsv</i>	<b>loaded graph</b> <i>name</i>
<b>FindPath</b> <i>graph node<sub>1</sub> node<sub>n</sub></i>	<b>path from</b> <i>node<sub>1</sub></i> <b>to</b> <i>node<sub>n</sub></i> : <i>node<sub>1</sub></i> <b>to</b> <i>node<sub>2</sub></i> <b>via</b> <i>edge<sub>1,2</sub></i> <i>node<sub>2</sub></i> <b>to</b> <i>node<sub>3</sub></i> <b>via</b> <i>edge<sub>2,3</sub></i> ... <i>node<sub>n-1</sub></i> <b>to</b> <i>node<sub>n</sub></i> <b>via</b> <i>edge<sub>n-1,n</sub></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 TestDriver 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!