CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Video on (if possible)**
- **Microphone off (except questions, discussion)**
- **Office Hours TODAY! 4:30-6pm**
    - Join queue using Google Form (look on Piazza)
    - Wait for invite

- **Fill out TA office hour poll (Piazza)**

- **Practice:**
    - **Raising hands (click on “Participants”)**
    - **Writing on the white board (“Annotate”)**

MAR 17 OUTLINE

- **Review: Queues**
- **Application of stacks and queues:**
  - Shunting-yard algorithm
- **Begin: Graphs**

MAR 17 OUTLINE

- **Review: Queues**

- **Application of stacks and queues:**
  - Shunting-yard algorithm

- **Begin: Graphs**

THE QUEUE ADT

Insertions and deletions are First In First Out (FIFO)

-Insert at the back

-Delete from the front

Operations:

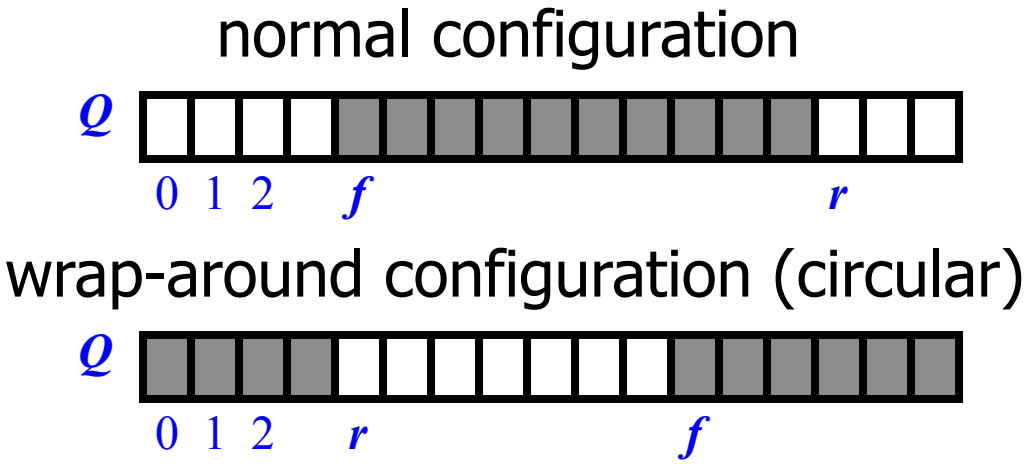
- `enqueue(Object)`
- `Object dequeue()`
- `Object first()`
- `int size()`
- `boolean isEmpty()`

ARRAY-BASED QUEUE IMPLEMENTATION

An array of size `n` in a circular fashion

Two `int`s to track front and size

- `f`: index of the front element
- `size`: number of stored elements



QUEUE WITH CIRCULAR ARRAY

3 instance variables

```
public class ArrayQueue<E> implements Queue<E> {

    public static final int CAPACITY = 1000;

    private int f;
    private int size;
    private E[] data;

    public ArrayQueue() {
        this(CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public ArrayQueue(int capacity) {
        f = 0;
        size = 0;
        data = (E[]) new Object[capacity];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public E first() throws EmptyQueueException {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return data[f];
    }
```

QUEUE WITH CIRCULAR ARRAY

3 instance variables

Use size for size() and isEmpty() methods

```
public class ArrayQueue<E> implements Queue<E> {

    public static final int CAPACITY = 1000;

    private int f;
    private int size;
    private E[] data;

    public ArrayQueue() {
        this(CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public ArrayQueue(int capacity) {
        f = 0;
        size = 0;
        data = (E[]) new Object[capacity];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public E first() throws EmptyQueueException {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return data[f];
    }
```

QUEUE WITH CIRCULAR ARRAY

3 instance variables

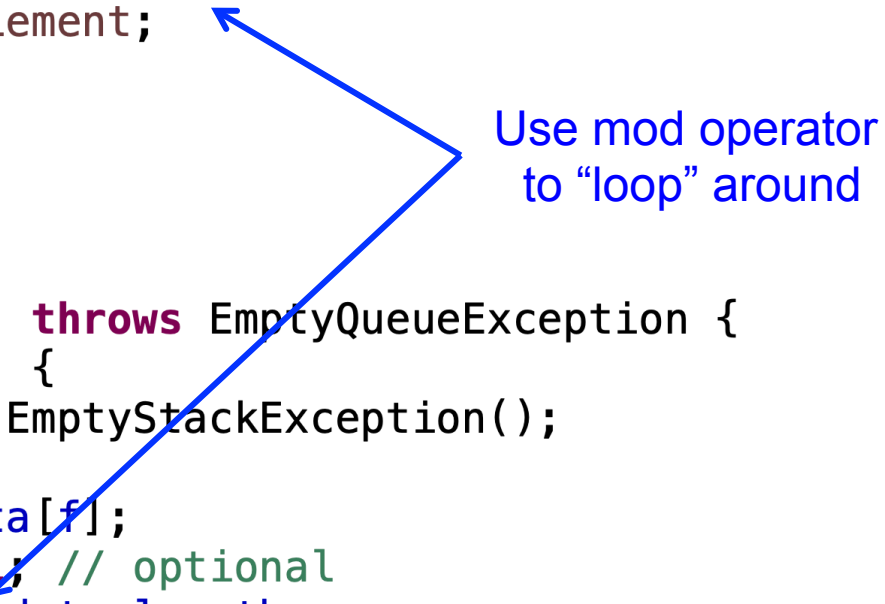
Use size for size() and isEmpty() methods

Return data at the front, but don't change queue

```
public class ArrayQueue<E> implements Queue<E> {

    public static final int CAPACITY = 1000;

    private int f;
    private int size;
    private E[] data;

    public ArrayQueue() {
        this(CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public ArrayQueue(int capacity) {
        f = 0;
        size = 0;
        data = (E[]) new Object[capacity];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public E first() throws EmptyQueueException {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return data[f];
    }
```

METHODS THAT CHANGE THE QUEUE

```
public void enqueue(E element) throws FullQueueException {
    if (size == data.length) {
        throw new FullQueueException();
    }

    int end = (f + size) % data.length;
    data[end] = element;

    size += 1;

}

public E dequeue() throws EmptyQueueException {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    E result = data[f];
    data[f] = null; // optional
    f = (f + 1) % data.length;

    size -= 1;
    return result;
}
```



METHODS THAT CHANGE THE QUEUE

```
public void enqueue(E element) throws FullQueueException {
    if (size == data.length) {
        throw new FullQueueException();
    }

    int end = (f + size) % data.length;
    data[end] = element;

    size += 1;

}

public E dequeue() throws EmptyQueueException {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    E result = data[f];
    data[f] = null; // optional
    f = (f + 1) % data.length;

    size -= 1;
    return result;
}
```

Dequeue returns data, but enqueue does not (adds data)

Implement in Lab 4!

Doubly-Ended Queue

THE DEQUE ADT

-Can add and remove from both the front and the back

-When using a circular array, still only need front (**f**), **size**, and **data** (the array)

Operations:

- `addFirst(Object)`
- `addLast(Object)`
- `Object removeFirst()`
- `Object removeLast()`
- `Object first()`
- `Object last()`
- `int size()`
- `boolean isEmpty()`

Implement in Lab 4!

Doubly-Ended Queue

THE DEQUE ADT

-Can add and remove from both the front and the back

-When using a circular array, still only need front (**f**), **size**, and **data** (the array)

Operations:

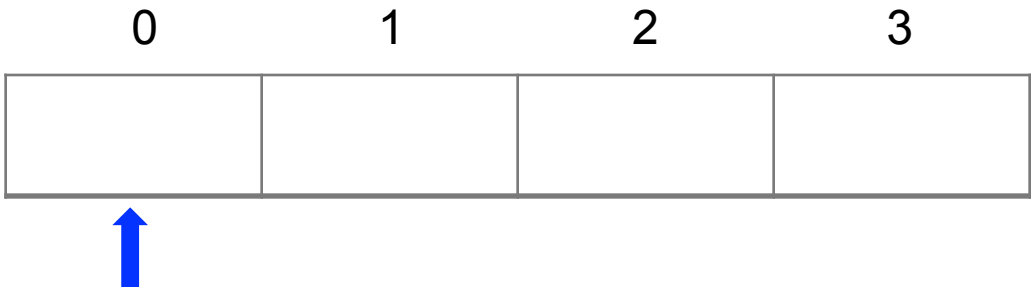
- `addFirst(Object)`
- **`addLast(Object)`**
- **`Object removeFirst()`**
- `Object removeLast()`
- **`Object first()`**
- `Object last()`
- **`int size()`**
- **`boolean isEmpty()`**

Orange: like regular Queue

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 0



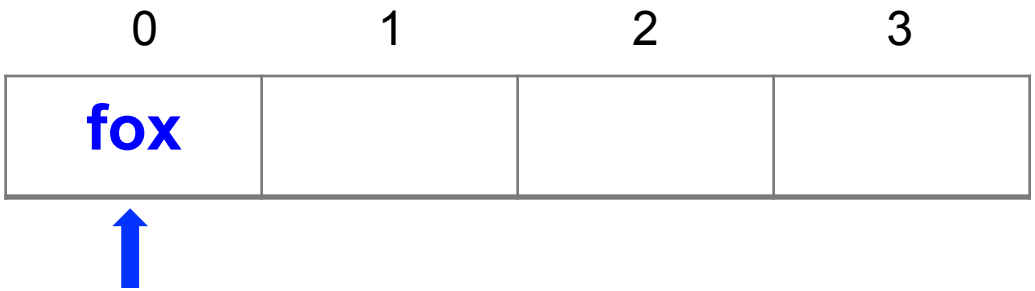
With a (random) partner, answer the following questions:

1) **Introduce yourselves** (names, where are you now, etc)
2) What does the **array** look like after this code is finished?
3) What does the **queue** look like read from first to last?

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

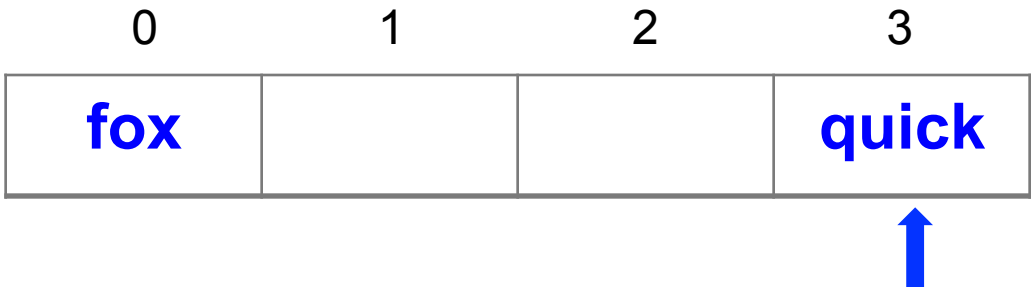
Size: 1



EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 2

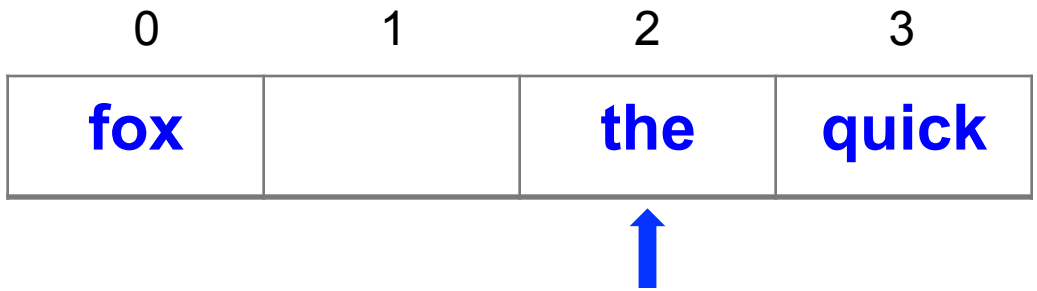


Arrow is front

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 3

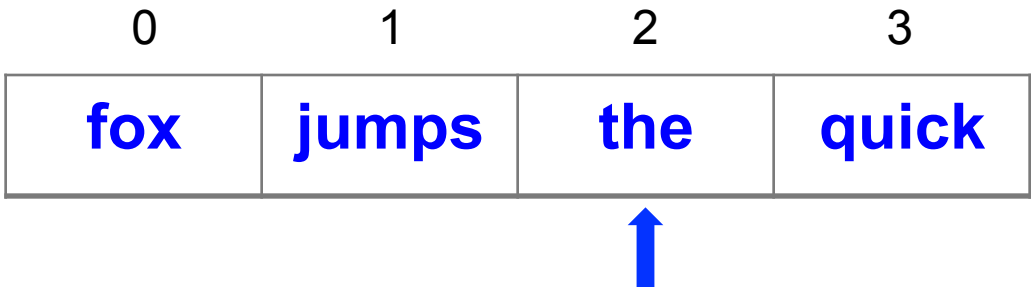


Arrow is front

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 4

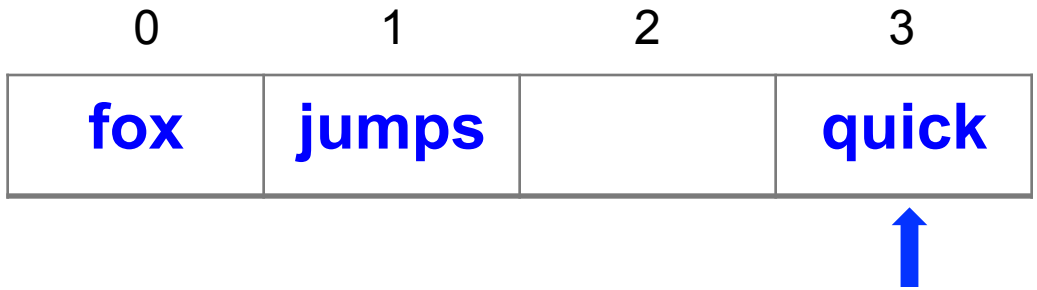


Arrow is front

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 3

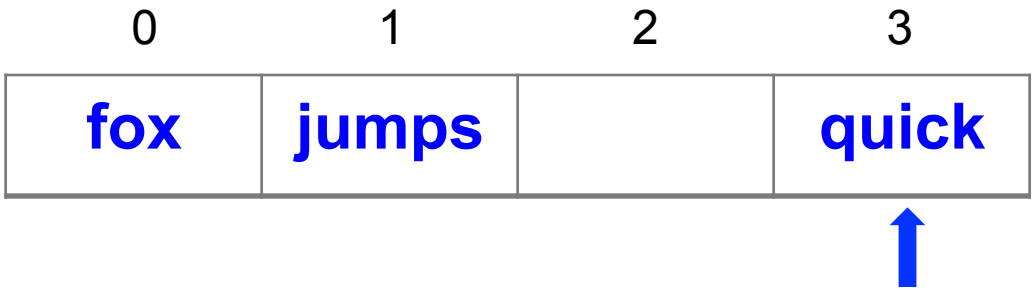


Arrow is front

EXAMPLE

```
ArrayDeque<String> dequeStr = new ArrayDeque<String>(4);
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 3



Arrow is front

2) What does the **array** look like after this code is finished?

[fox, jumps, *null*, quick]

3) What does the **queue** look like read from first to last?

(quick, fox, jumps)

EXAMPLE


```
Arra
dequeStr.addLast("fox");
dequeStr.addFirst("quick");
dequeStr.addFirst("the");
dequeStr.addLast("jumps");
dequeStr.removeFirst();
```

Size: 3

0	1	2	3
fox	jumps	the	quick

// add to front
f = 3
(f - 1) % length
(3 - 1) % 4 = 2

+ len

f = 2
size = 3

(f + size) % length
(2 + 3) % 4 = 1 // index of the new last

MAR 17 OUTLINE

- Review: Queues

- **Application of stacks and queues:**
  - Shunting-yard algorithm

- Begin: Graphs

SHUNTING-YARD ALGORITHM

Goal: convert infix ("regular") to post-fix

Then we can use our previous stack algorithm to compute result

Brief outline: Setup: need one **stack** and one **queue**

- Left parenthesis "(": push onto **stack**
- Number: add to **queue**
- Operator:
  - If current top operator has higher precedence, then pop that and add to **queue**
  - Push this operator onto **stack**
- Right parenthesis ")":
  - Pop all operators until "(" and add to **queue**
  - Pop "(" from **stack**

End: pop all remaining operators and add to **queue**

SHUNTING-YARD ALGORITHM

Example:

(4 x (1 + 3) − 8) / 7

Extra practice (after class):

A) $((2+1) \times 5) + (4 \times 3)\ \hat{}\ 2 \times 2$

B) $15 - 7 \times 2 / 4$

SHUNTING-YARD ALGORITHM

infix: “normal”

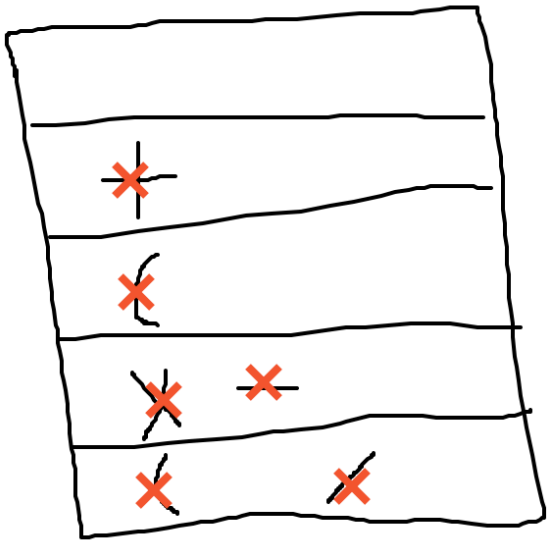
(4 x (1 + 3) - 8) / 7

operator is between the numbers

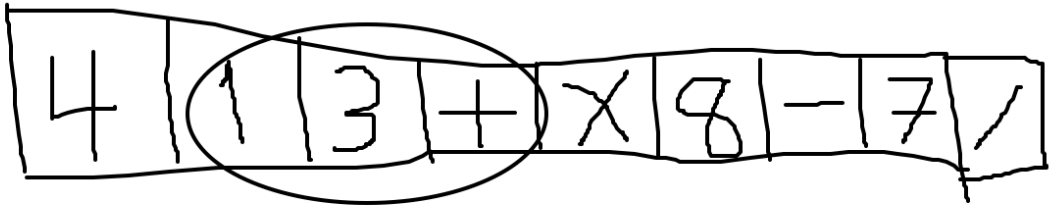
very difficult! (to compute)

goal: convert infix to postfix

Stack



Queue



operator is *after* the numbers

postfix!

very easy for the computer!

SHUNTING-YARD ALGORITHM

POSTFIX -> RESULT (REVIEW)

postfix -> result

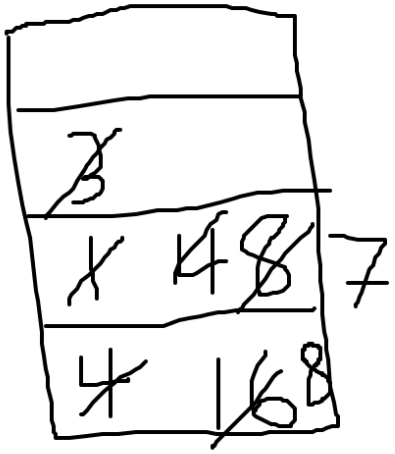
postfix: 4 1 3 + x 8 - 7 /

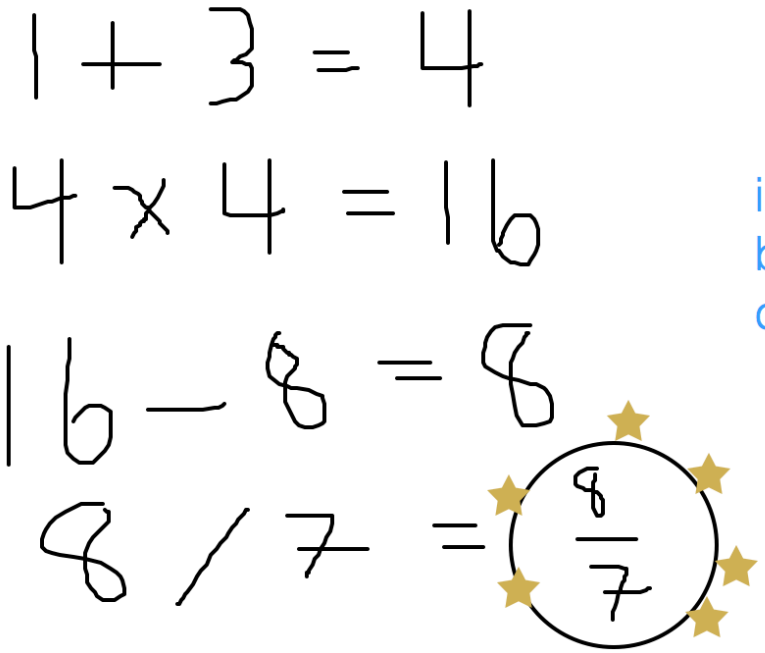
result of shunting-yard algorithm

Lab 4:

if operator.equals("+"):
num1 + num2;

stack





in java: answer would be 1 since integer division

better: use double!

MAR 17 OUTLINE

- Review: Queues

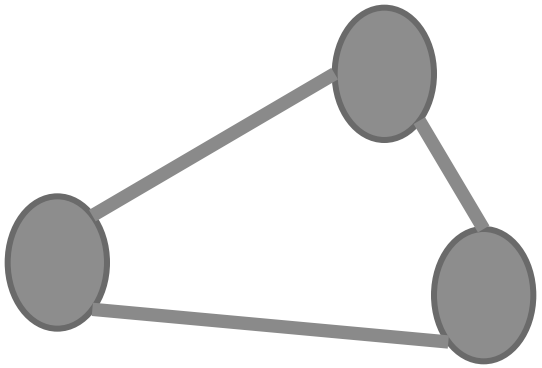
- Application of stacks and queues:
  - Shunting-yard algorithm

- **Begin: Graphs**

GRAPHS

Graphs (aka networks) represent relationships between pairs of objects.

Vertices (aka nodes) are the objects. (Singular: vertex)
Edges (aka links) are the relationships.



Note: these are graph theory graphs, not charts or plots.

EXAMPLE GRAPH: SOCIAL NETWORK

Vertex:

- person

Edge:

- interaction between two people

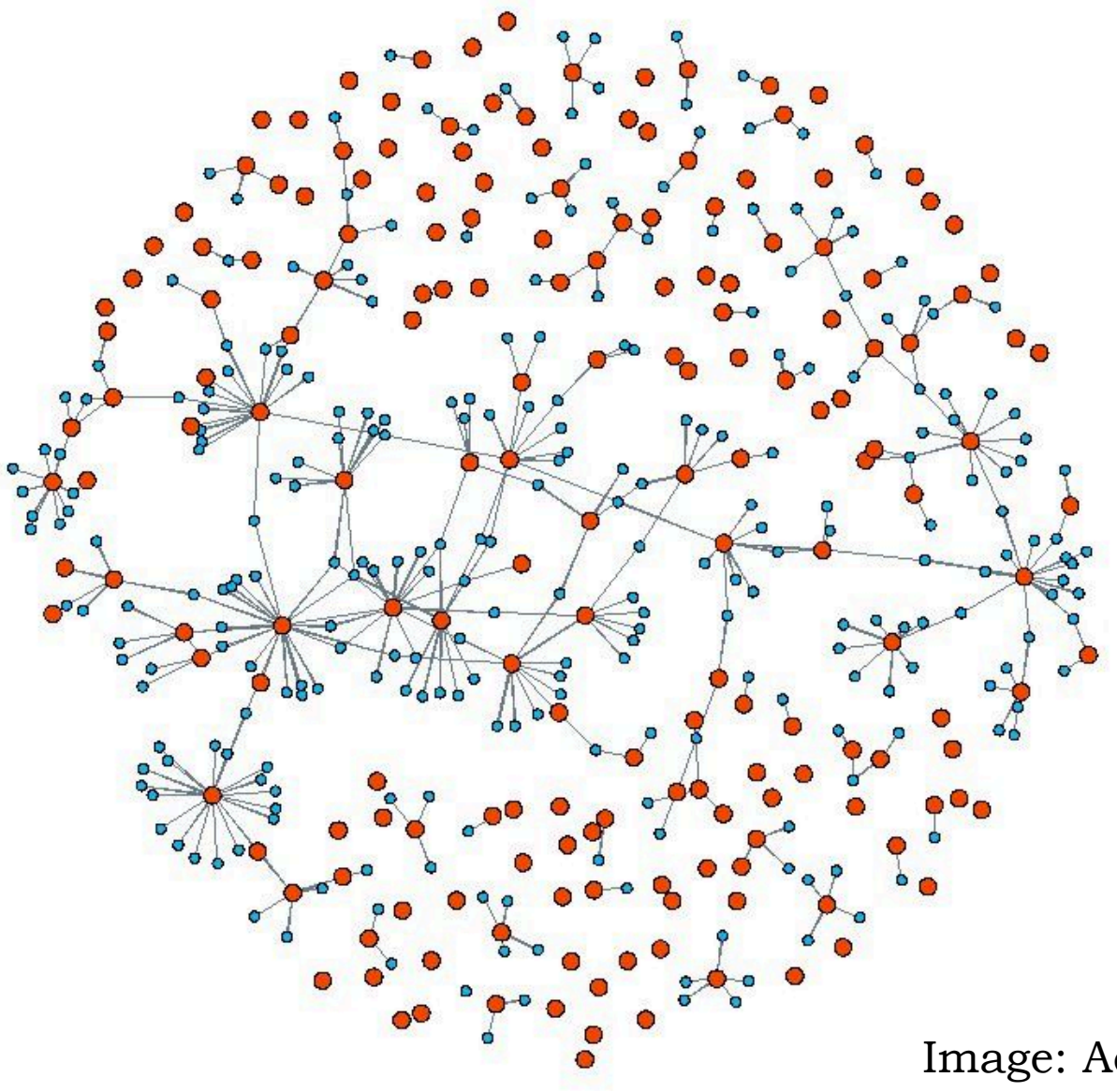


Image: Adobe

EXAMPLE GRAPH: TRANSPORTATION NETWORK

Vertices: train stops

Edges: train lines

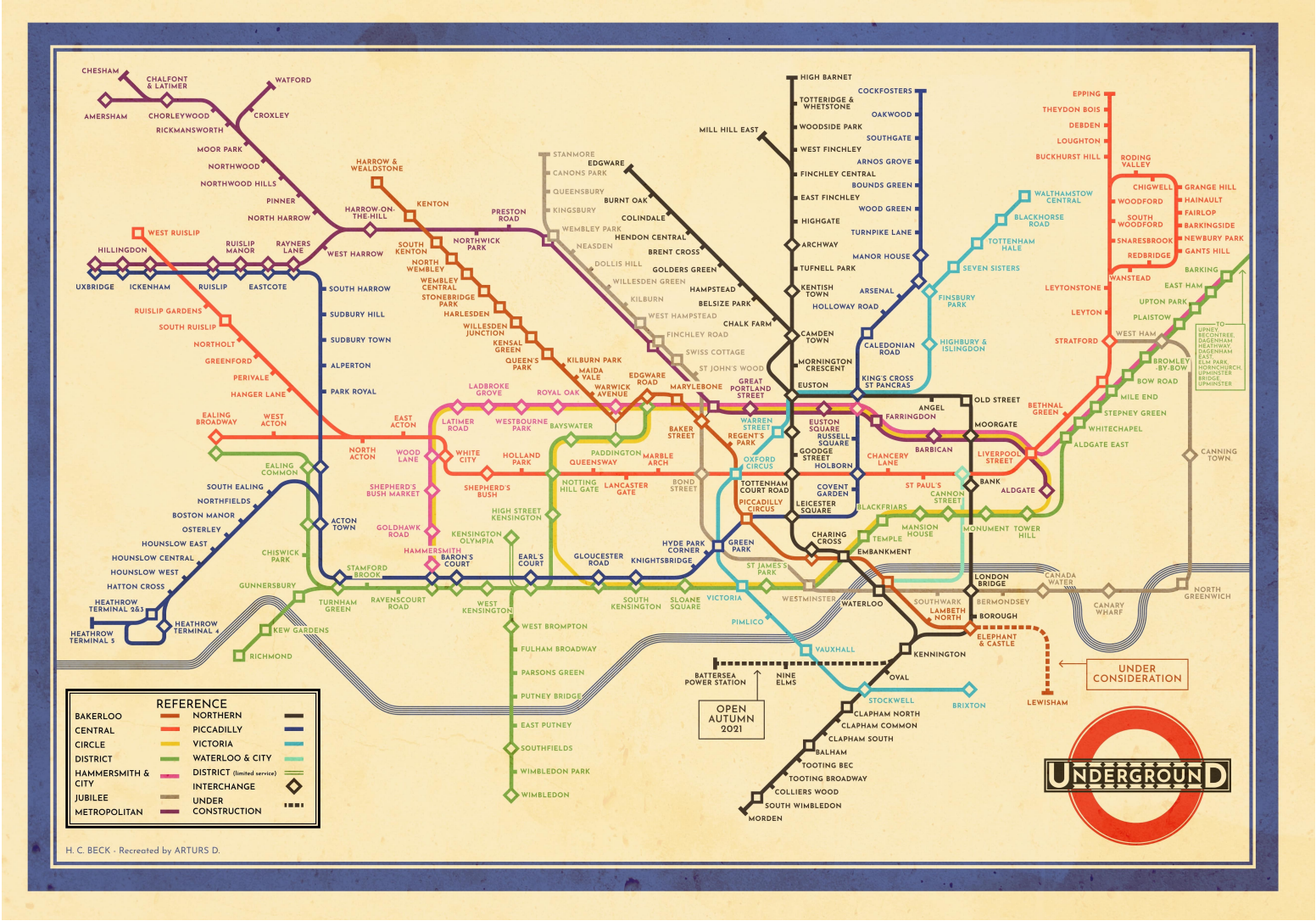


Image: Harry Beck

NOTATION

A graph *G* is a set of vertices *V* and a set of edges *E*:

$$G = (V, E)$$

Each edge is a pair of vertices:

$$(u, v) \in V$$

The total number of vertices in a graph is denoted *n*:

$$|V| = n$$

The total number of edges in a graph is denoted *m*:

$$|E| = m$$

DIRECTED VS. UNDIRECTED

Graphs can be *directed* or *undirected*.

Undirected graphs have edges where the vertices in the edges are *unordered.*

E.g., Facebook's friend network.

Directed graphs (aka digraphs) have edges where the vertices in the edge are *ordered*.

E.g., Twitter's follower network.

EDGE TERMINOLOGY

The two vertices joined by an edge are called *endpoints*.

If an edge is directed, the start vertex is the *origin* and the end vertex is the *destination*.

origin *destination*

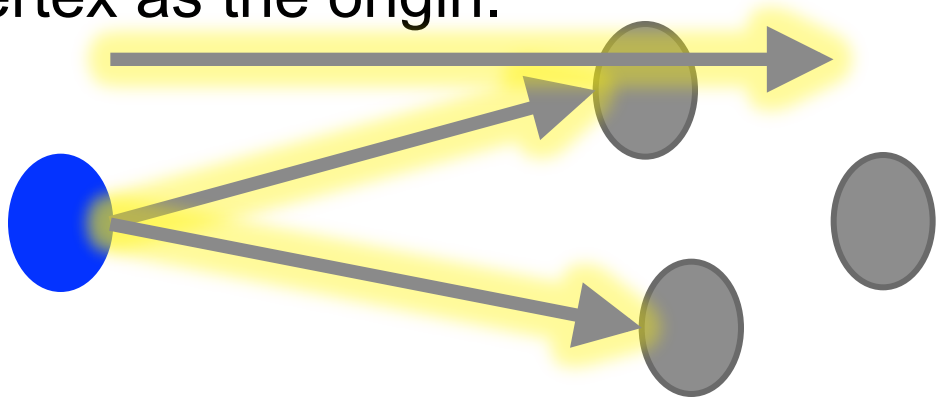
EDGE TERMINOLOGY

Two vertices are *adjacent* if they are joined by an edge.

Adjacent vertices are also known as *neighbors*.

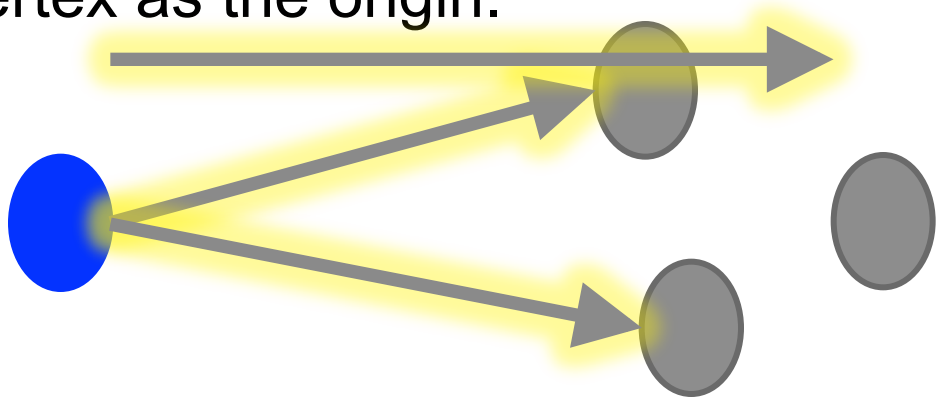
DIRECTED GRAPH TERMINOLOGY

The ***outgoing edges*** of a vertex in a directed graph are the edges with that vertex as the origin.

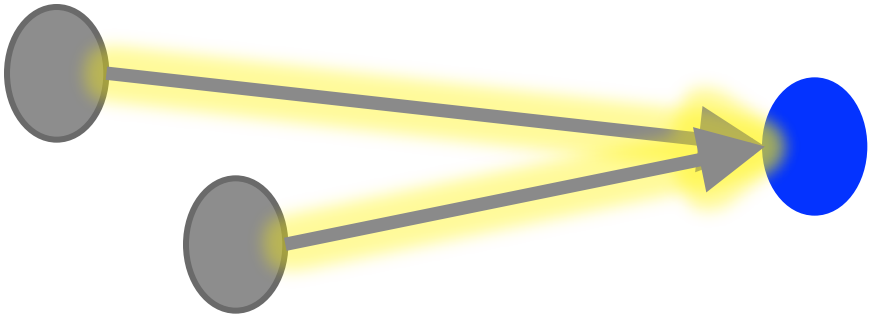


DIRECTED GRAPH TERMINOLOGY

The ***outgoing edges*** of a vertex in a directed graph are the edges with that vertex as the origin.

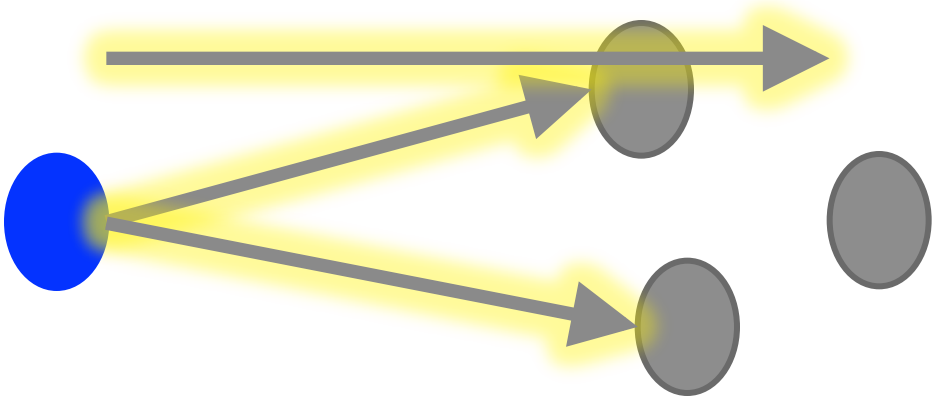


The ***incoming edges*** of a vertex in a directed graph are the edges with that vertex as the destination.

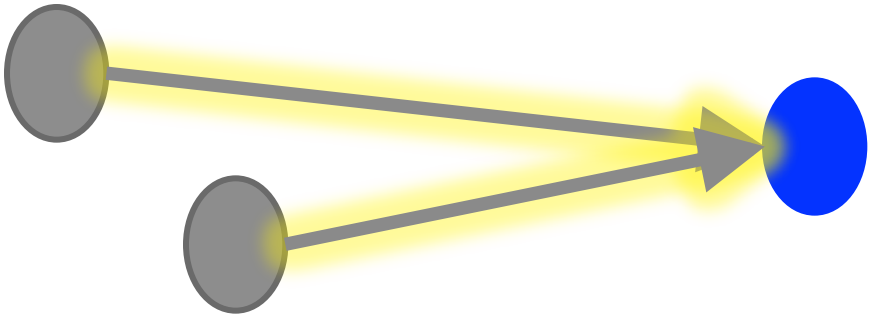


DIRECTED GRAPH TERMINOLOGY

The ***out-degree*** of a vertex is the number of outgoing edges.

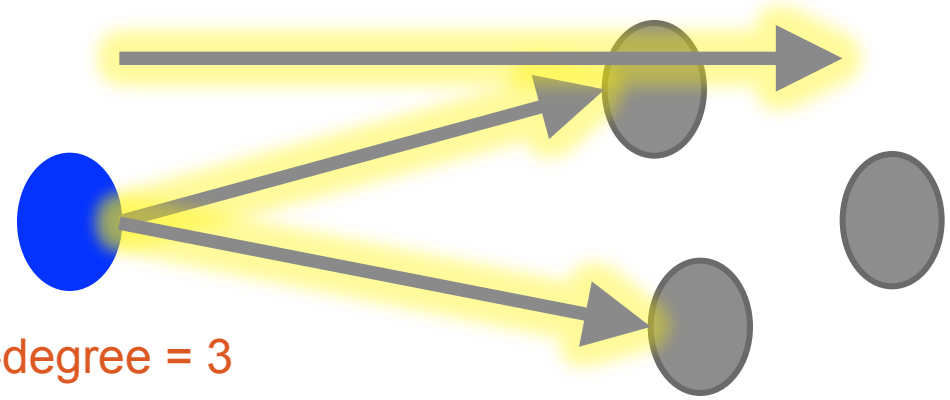


The ***in-degree*** of a vertex is the number of incoming edges to that vertex.

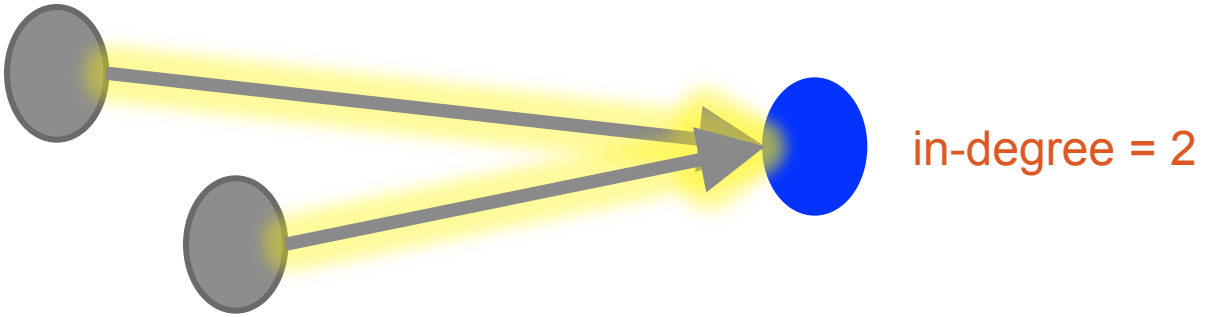


DIRECTED GRAPH TERMINOLOGY

The ***out-degree*** of a vertex is the number of outgoing edges.



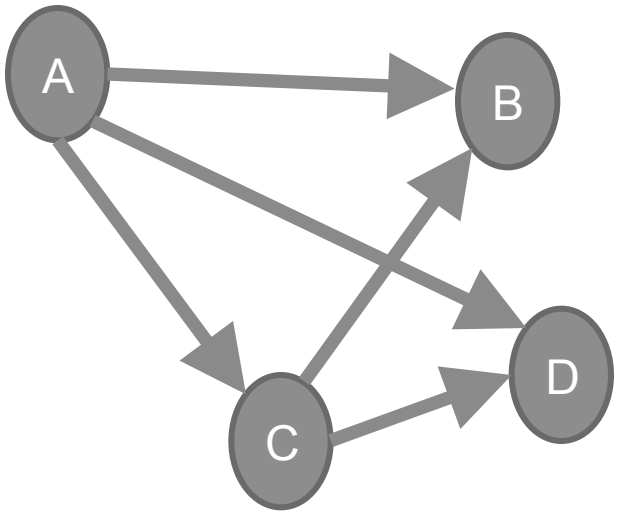
The ***in-degree*** of a vertex is the number of incoming edges to that vertex.



INDIVIDUAL EXERCISE

Identify the ***in-degree*** and ***out-degree*** for each of the nodes in the below graph:

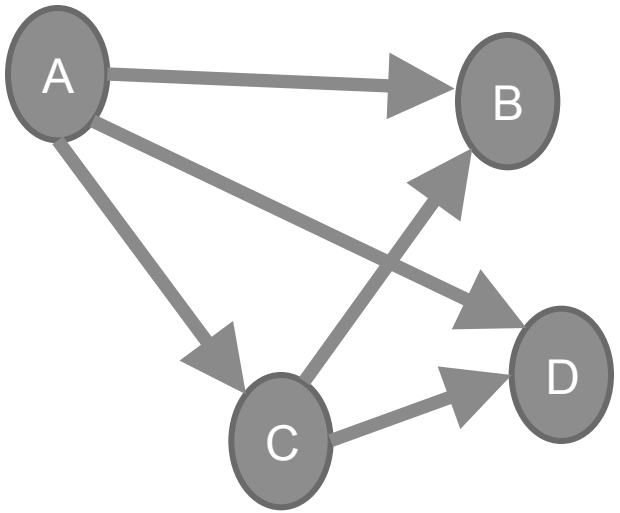
Node	in-degree	out-degree
A		
B		
C		
D		



INDIVIDUAL EXERCISE

Identify the ***in-degree*** and ***out-degree*** for each of the nodes in the below graph:

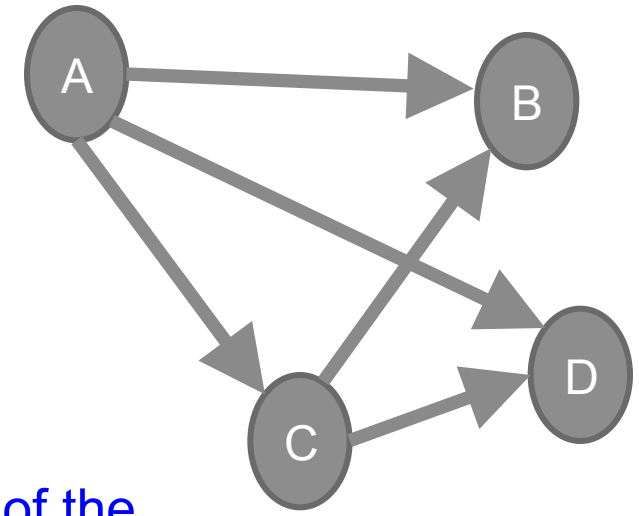
Node	in-degree	out-degree
A	0	3
B	2	0
C	1	2
D	2	0



INDIVIDUAL EXERCISE

Identify the ***in-degree*** and ***out-degree*** for each of the nodes in the below graph:

Node	in-degree	out-degree
A	0	3
B	2	0
C	1	2
D	2	0

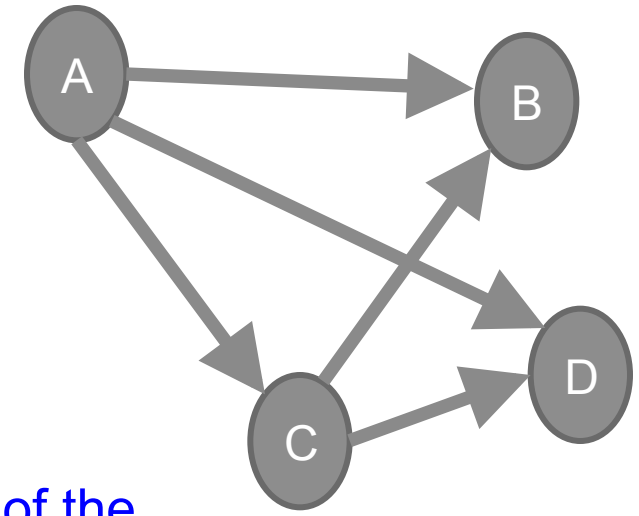


Q: What do you notice about the sums of the in-degrees and out-degrees?

INDIVIDUAL EXERCISE

Identify the ***in-degree*** and ***out-degree*** for each of the nodes in the below graph:

Node	in-degree	out-degree
A	0	3
B	2	0
C	1	2
D	2	0

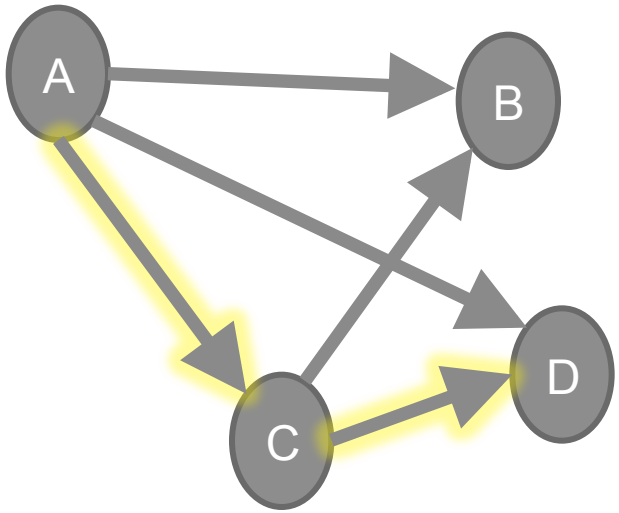


Q: What do you notice about the sums of the in-degrees and out-degrees?

A: They are the same and equal to the number of edges! Because each edge has one origin and one destination.

PATHS

A ***path*** is a sequence of alternating vertices and edges in the graph such that edges connect adjacent vertices in a continuous sequence. A ***directed path*** is the equivalent for a directed graph.



CYCLES

A ***cycle*** is a path that starts and ends at the same vertex and has at least one edge. A ***directed cycle*** is the same for a directed graph.

A directed graph is ***acyclic*** if it has no directed cycles.

(A ***tree*** is a special type of acyclic graph that we will discuss more next week.)

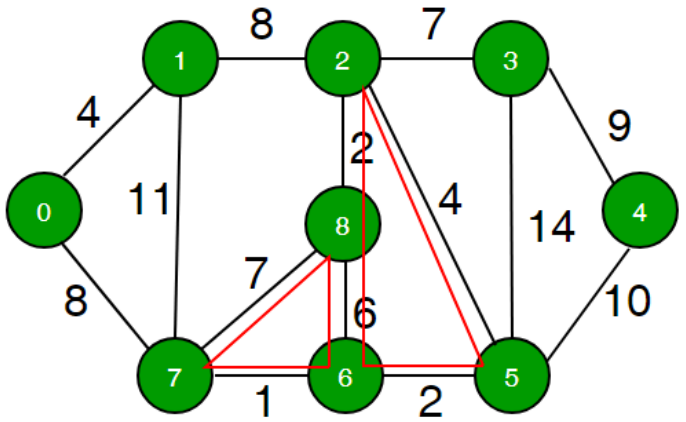


Image: GeeksforGeeks

CONNECTED COMPONENTS

A vertex *u* is ***reachable*** from a vertex *v* if there is a path from *u* to *v*.

A graph is ***connected*** if any vertex is reachable from any other vertex. A directed graph is ***strongly connected*** if there is a path between any vertices *v* and *u* as well as a path back from *u* to *v*.

THE GRAPH ADT

The designation of the graph as undirected or directed happens at construction time.

```
numVertices()
vertices()
numEdges()
edges()
getEdge(u,v)
endpoints(e)
removeVertex(v)
```

```
outDegree(v)
inDegree(v)
outgoingEdges(v)
incomingEdges(v)
insertVertex(elem)
insertEdge(u,v,elem)
removeEdge(e)
```

PAIR EXERCISE

What should each of these methods return for this specific graph (in cases of vertex and/or edge, do one example)?

```
numVertices()
vertices()
numEdges()
edges()
getEdge(u,v)
endpoints(e)
outDegree(v)
inDegree(v)
outgoingEdges(v)
incomingEdges(v)
```