CSCI 136:
Data Structures
and
Advanced Programming

Lecture 27

Graphs, part 2

Instructor: Dan Barowy

Williams

More graph definitions

Outline

- More graph defs
- Graph ADT operations
- Graph representations

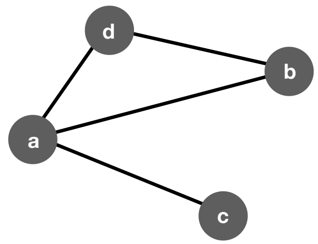
Reachability and Connectedness

"Siri, can I drive from Boston to Hong Kong?"

"Siri, can I drive from any point to any other point?"

Reachability

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

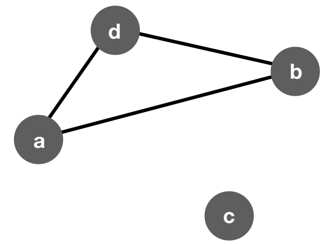


For an **undirected** graph **G**, **v** is **reachable** from vertex **u** iff **u** is **reachable** from vertex **v**.

Is **c reachable** from **d**? Yes.

Connectedness

An undirected graph **G** is **connected** if for every pair of vertices **u, v** in **G**, **v** is **reachable** from **u**.



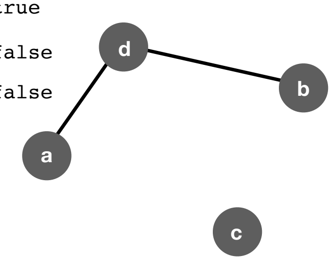
The set of all **vertices reachable from v**, along with all **edges** of **G** connecting any two of them, is called the **connected component of v**.

(note that the connected component is itself a graph)

Graph operations

Fundamental graph ADT operations

```
adjacent(a, d) = true
adjacent(a, b) = false
adjacent(a, c) = false
```



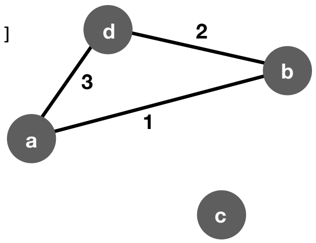

```
bool adjacent(Vertex u, Vextex v):
```

Given vertices **u** and **v**, are they **adjacent**?

(i.e., share an edge?)

Fundamental graph ADT operations

```
vertices(1) = [a, b]
vertices(2) = [d, b]
```

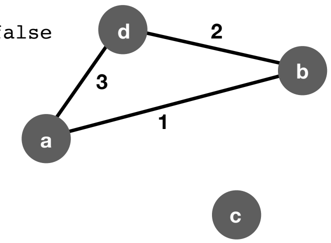



```
Vertex[] vertices(Edge e):
```

Given edge **e**, what are its **end points**?

Fundamental graph ADT operations

```
incident(a, 1) = true
incident(a, 2) = false
```



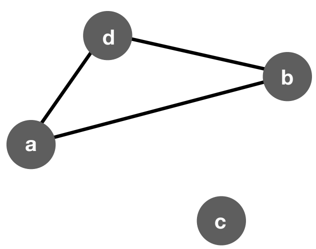

```
bool incident(Vertex v, Edge e):
```

Given vertex **v** and edge **e**, are they **incident**?

(i.e., is **v** an **endpoint** of edge **e**?)

Fundamental graph ADT operations

```
degree(a) = 2
degree(c) = 0
```

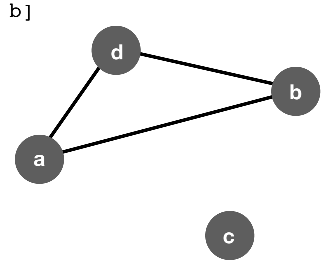



```
int degree(Vertex v):
```

Given vertex **v** how many vertices are **adjacent**?

Fundamental graph ADT operations

```
neighbors(a) = [d, b]
neighbors(c) = []
```



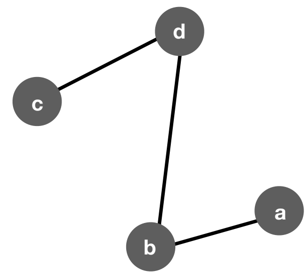

```
Vertex[] neighbors(Vertex v):
```

Given vertex **v** what other vertices are **adjacent**?

Graph representations

Adjacency matrix

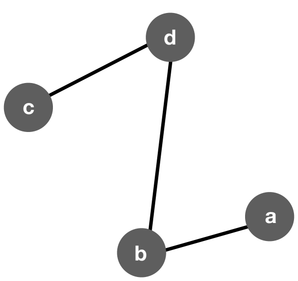
An **adjacency matrix** is a data structure for representing a finite graph. It consists of a **square matrix** (usually implemented as an array of arrays). In the simplest case, the **elements** of the matrix indicate **whether an edge is present**. Elements on the diagonal are **defined as zero**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	1
d	0	1	1	0

Adjacency matrix

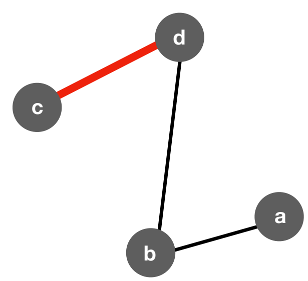
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	1
d	0	1	1	0

Adjacency matrix

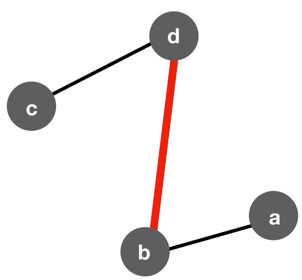
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	**1**
d	0	1	**1**	0

Adjacency matrix

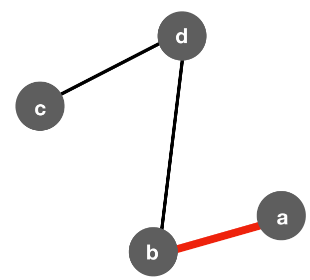
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	1
d	0	1	1	0

Adjacency matrix

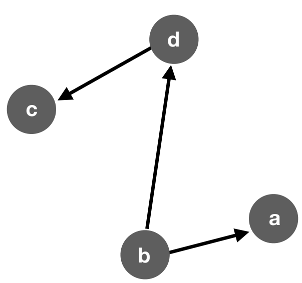
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	1
d	0	1	1	0

Adjacency matrix

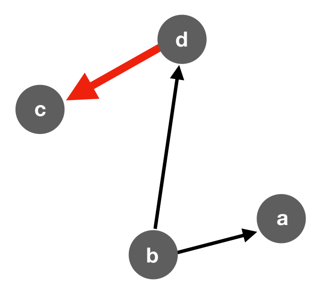
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form.



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	0	0	0
d	0	0	1	0

Adjacency matrix

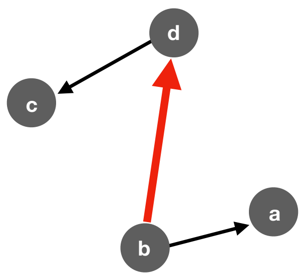
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form.



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	0	0	0
d	0	0	1	0

Adjacency matrix

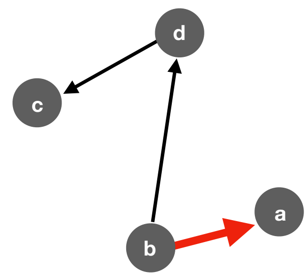
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form.



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	0	0	0
d	0	0	1	0

Adjacency matrix

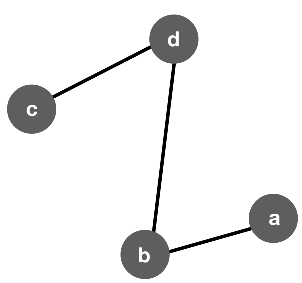
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form.



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	0	0	0
d	0	0	1	0

Adjacency list

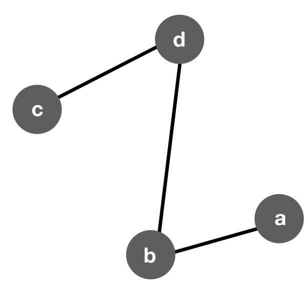
An **adjacency list** is a data structure for representing a finite graph. It consists of a **list of unordered lists**.




```
[[c,d],[d,b],[a,b]]
```

Adjacency list

There are many variants on adjacency lists. The most common is the **object-oriented adjacency list** that stores **a list of adjacent vertices** in each vertex object.

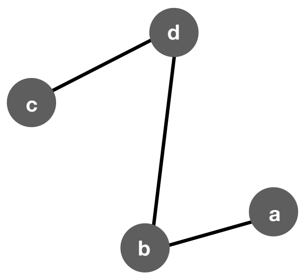


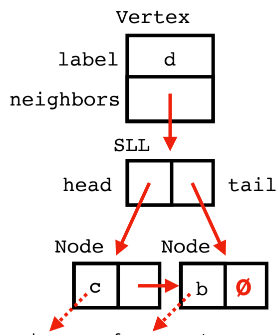

```
a: [b]
b: [a,d]
c: [d]
d: [b,c]
```

Adjacency list

Object-oriented adjacency list:

```
public class Vertex<T> {
    T label;
    List<Vertex<T>> neighbors = new SinglyLinkedList<>();
    …
}
```

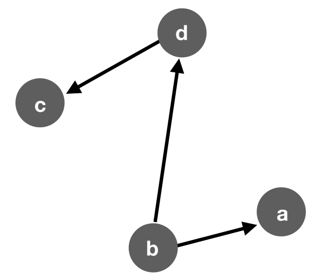




(strictly speaking, `c` and `d` are references to `Vertex` objects)

Adjacency list

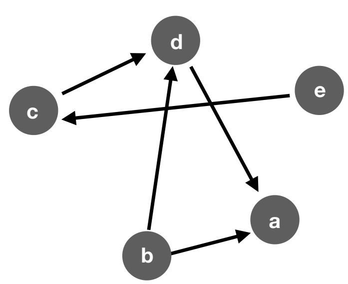
This latter version is **especially thrifty** for **directed graphs**.




```
a: []
b: [a,d]
c: []
d: [c]
```

Activity

Write down both **adjacency matrix** and **adjacency list** representations for this graph.



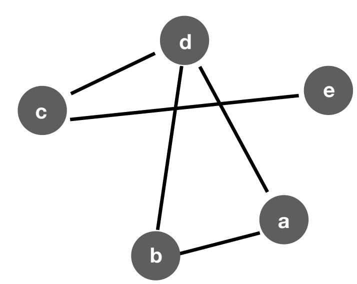
Which one is better for this graph? Why? (think Big-O)

Activity: connectedness

`boolean connected():`

How might I compute this using fundamental ops?

(`adjacent`, `vertices`, `incident`, `degree`, `neighbors`)



(note that graph is undirected)

Idea: breadth-first counting

Idea:

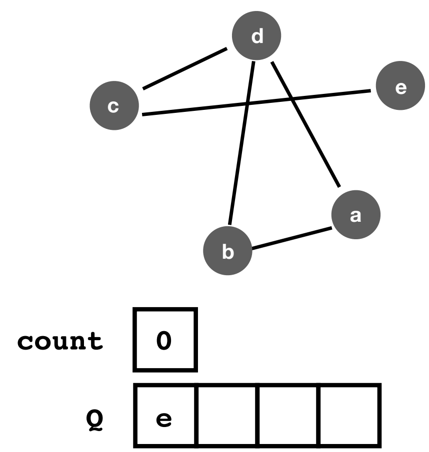
(suppose we know |G|)

boolean isConnected(Vertex start)

1. let count = 0
2. let Q be an empty queue
3. enqueue start
4. while Q not empty
    a. dequeue v
    b. count v
    c. mark v as visited
    d. put v's unmarked neighbors in Q
5. if count = # of vertices in graph, return true else false

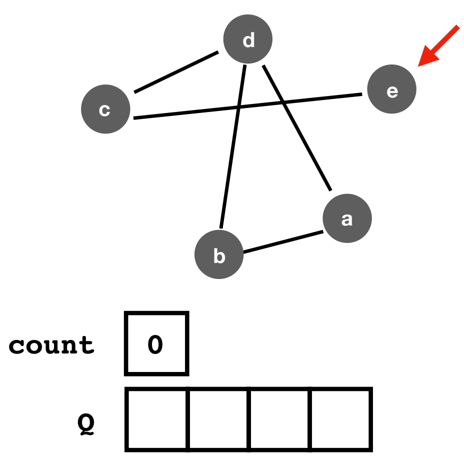
Algorithm: connectedness

initialize algorithm



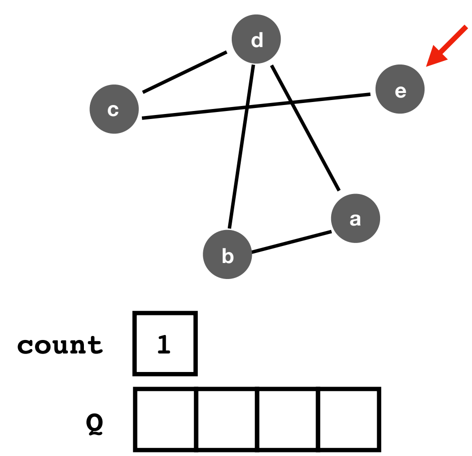
Algorithm: connectedness

dequeue v



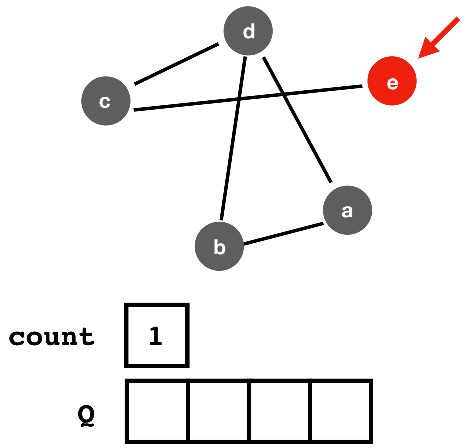
Algorithm: connectedness

count v



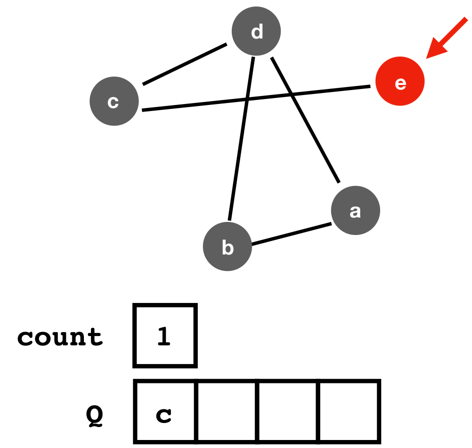
Algorithm: connectedness

mark v



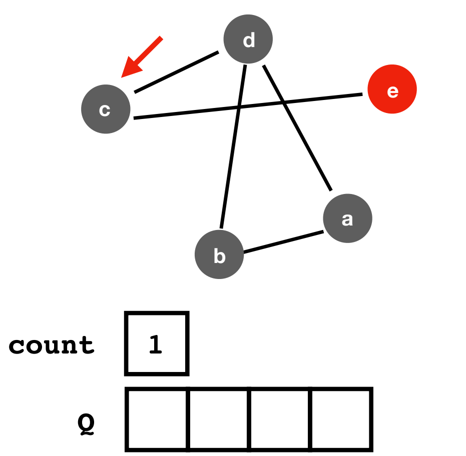
Algorithm: connectedness

enqueue unmarked neighbors



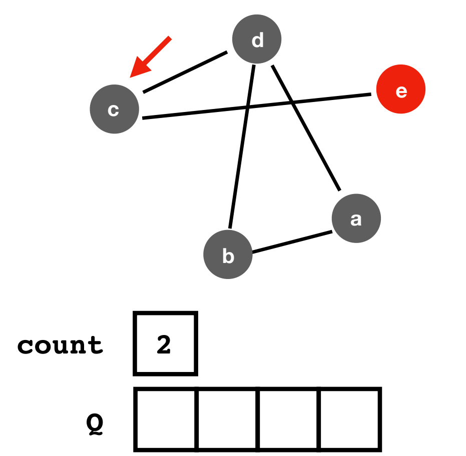
Algorithm: connectedness

dequeue v



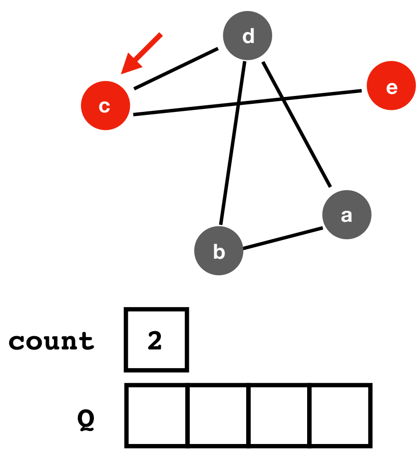
Algorithm: connectedness

count v



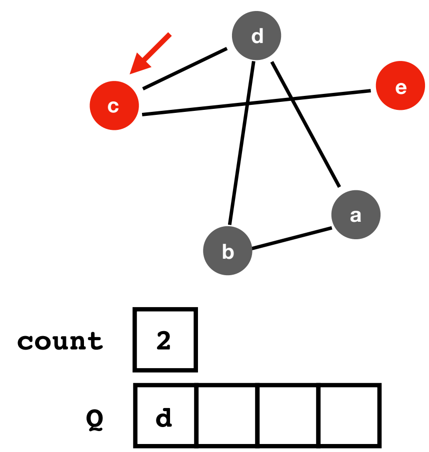
Algorithm: connectedness

mark v



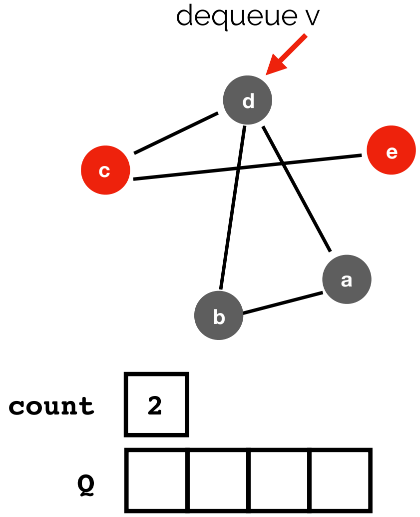
Algorithm: connectedness

enqueue unmarked neighbors



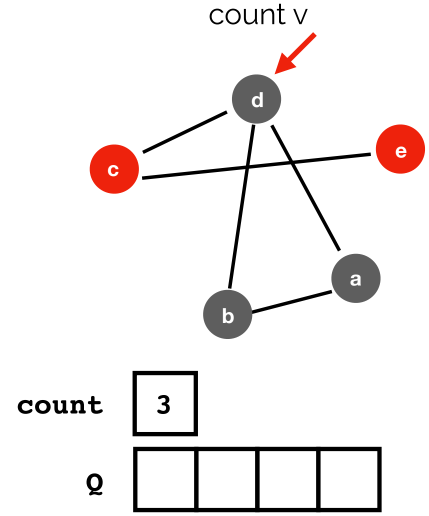
Algorithm: connectedness

dequeue v



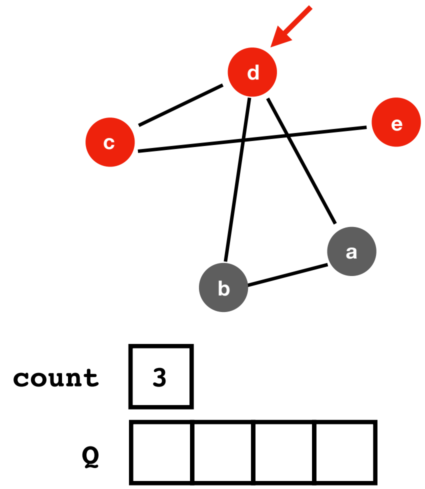
Algorithm: connectedness

count v



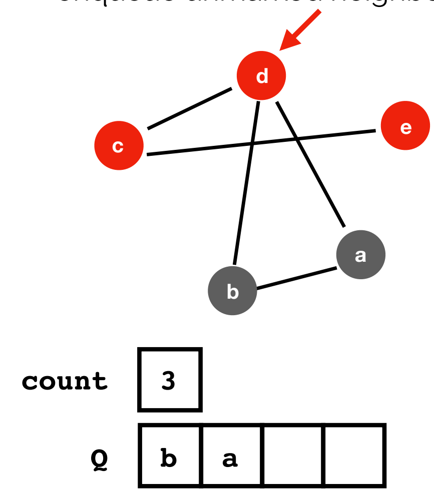
Algorithm: connectedness

mark v



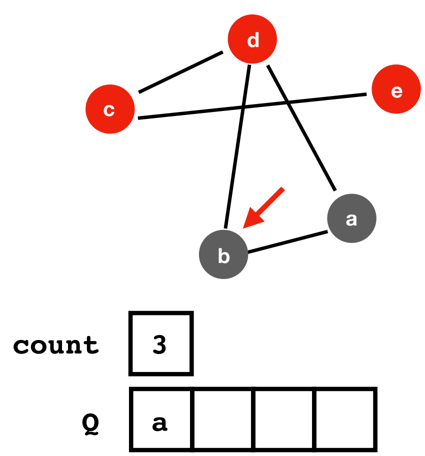
Algorithm: connectedness

enqueue unmarked neighbors



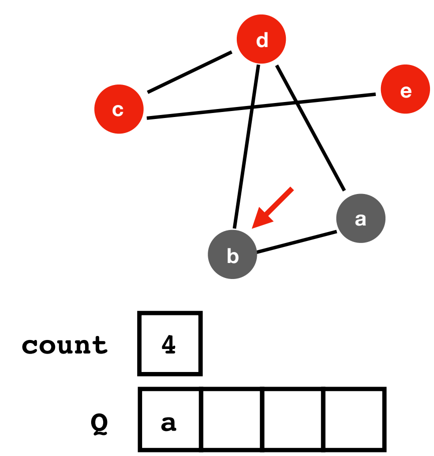
Algorithm: connectedness

dequeue v



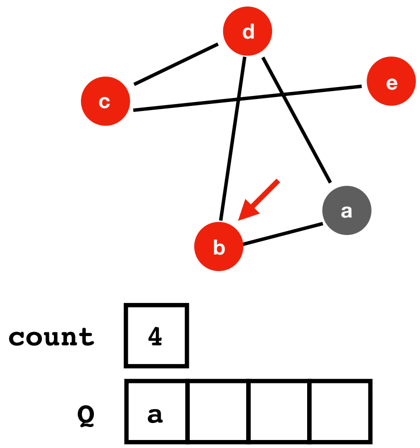
Algorithm: connectedness

count v



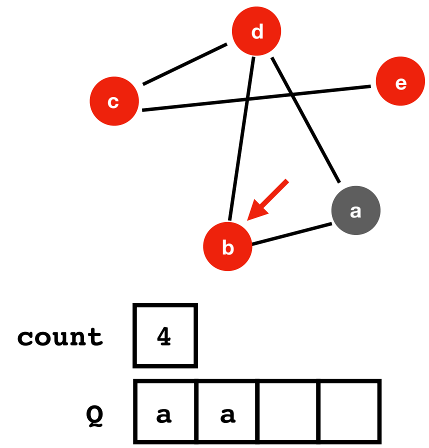
Algorithm: connectedness

mark v



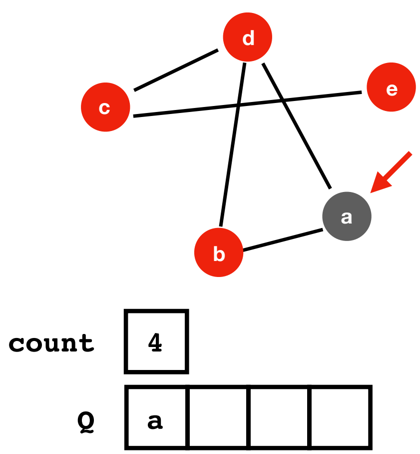
Algorithm: connectedness

enqueue unmarked neighbors



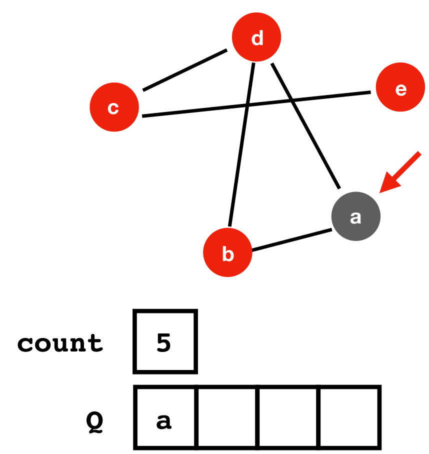
Algorithm: connectedness

dequeue v



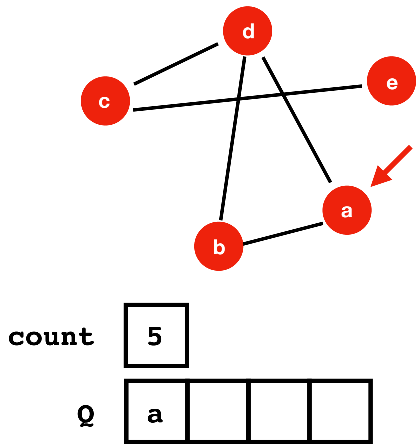
Algorithm: connectedness

count v



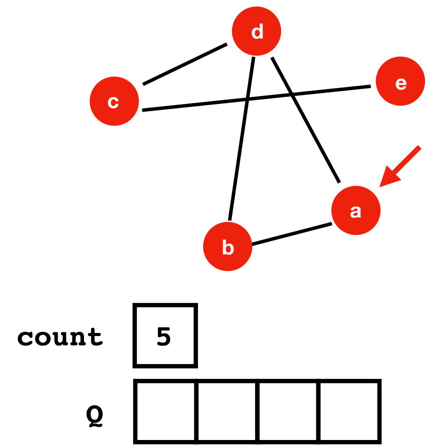
Algorithm: connectedness

mark v



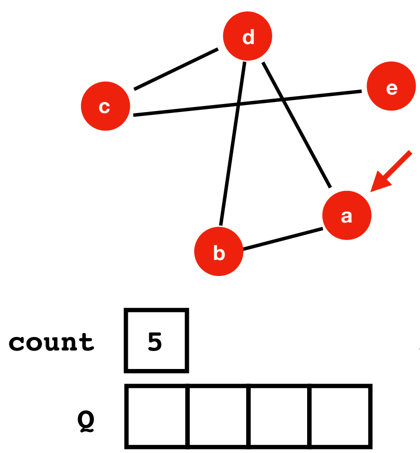
Algorithm: connectedness

dequeue v (but don't visit)



Algorithm: connectedness

compute |G| == count




```
return 5 == 5
   (true)
```

Recap & Next Class

Today we learned:

More graph definitions

Graph ADT operations

Graph representations

Next class:

Interesting graph problems