

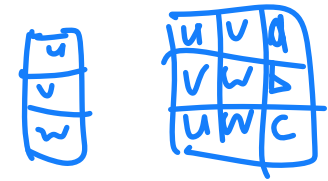
Data Structures

Graph Implementations

CS 225

Brad Solomon

October 26, 2026

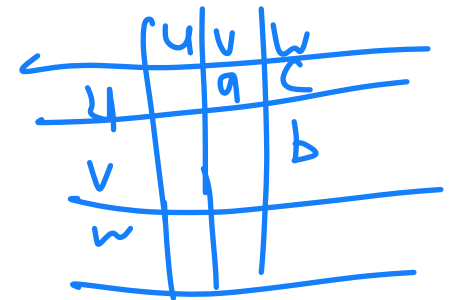


u s



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



KICKOFF CEREMONY: November 6th, 6PM
@ Digital Computer Lab 1320



DSDG
DATA SCIENCE
DEVELOPMENT
GROUP UIUC

INTRODUCING HACKATHON

NOV 2026

6-8th

\$5 Registration
Fee

OPEN TO ALL MAJORS

WINNER CASH PRIZE

SPEAKERS FROM MICROSOFT
COPILOT + MORE



Interest Form!!

Questions? Reach out to
dsdg.illinois@gmail.com

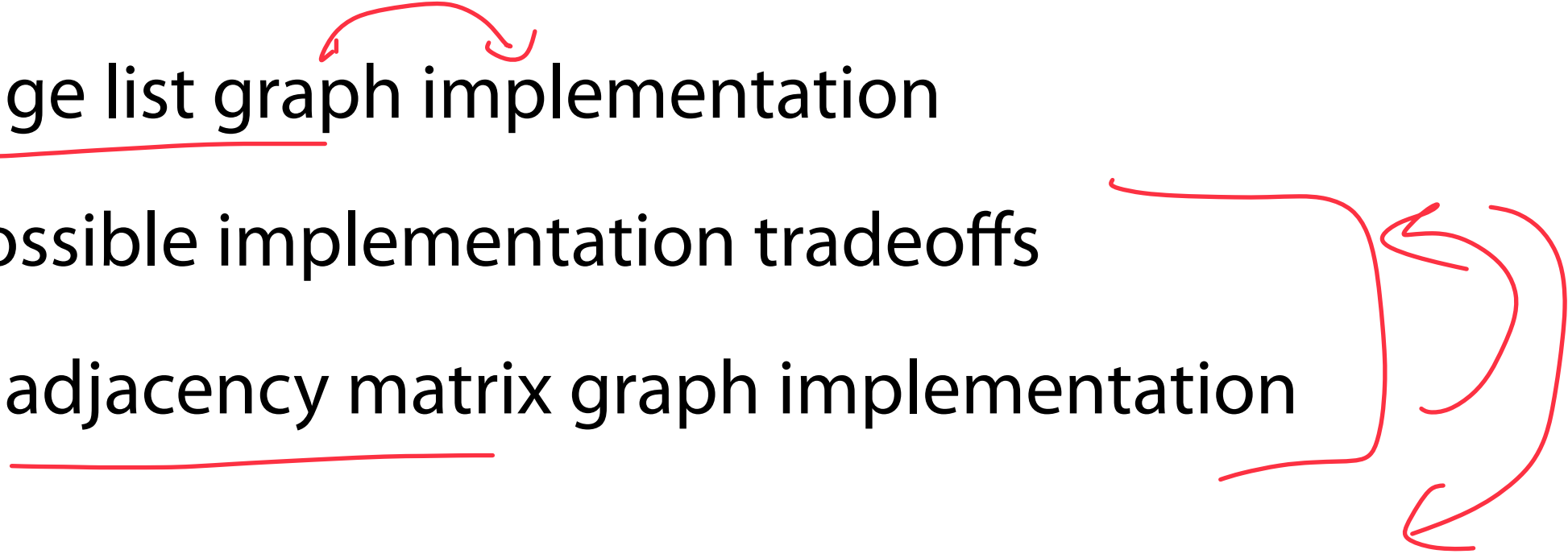
Learning Objectives

Review edge list graph implementation

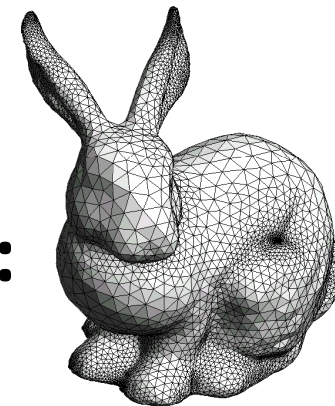
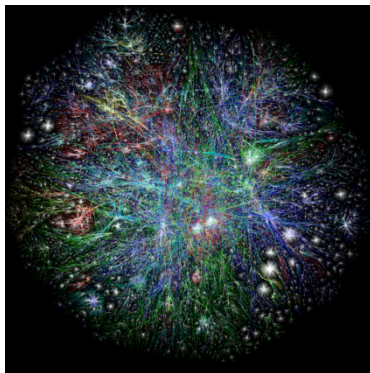
Explore possible implementation tradeoffs

Introduce adjacency matrix graph implementation

adjacency list

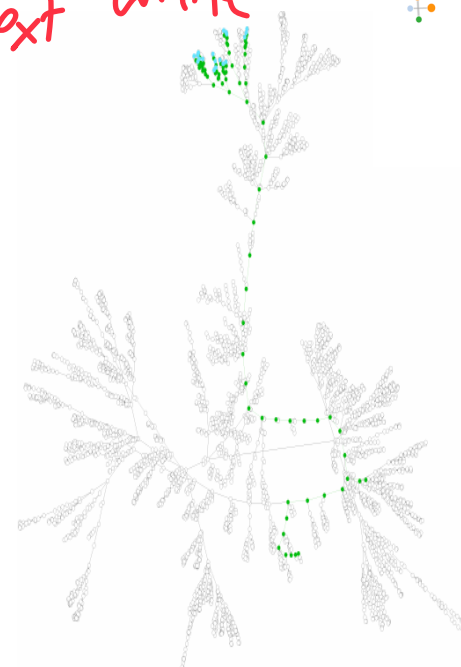
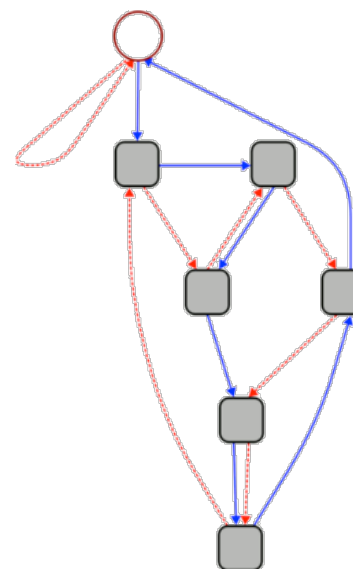
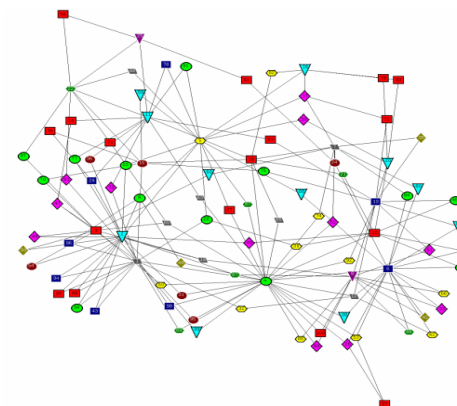
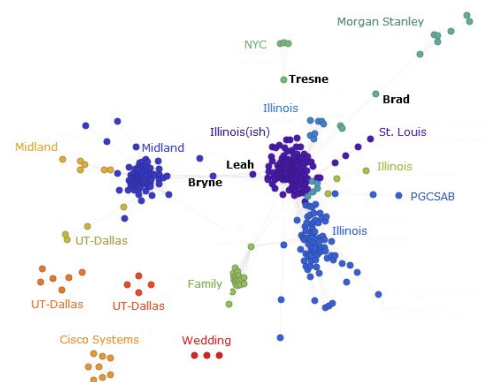
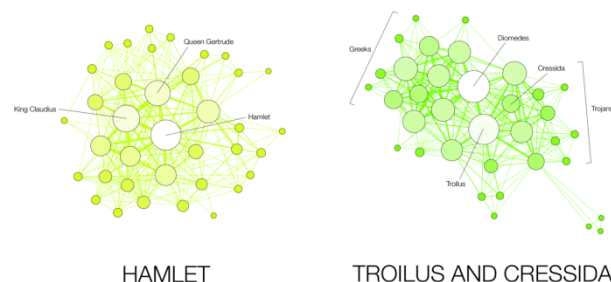
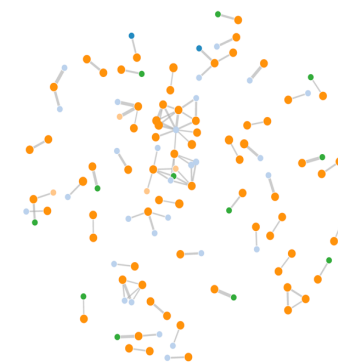
Hand-drawn red arrows and a bracket. A bracket on the right side groups the 'edge list' and 'adjacency matrix' text. Two curved arrows point from the 'edge list' text to the 'adjacency matrix' text, suggesting a comparison or relationship between the two graph implementations.

Graphs



To study all of these structures:

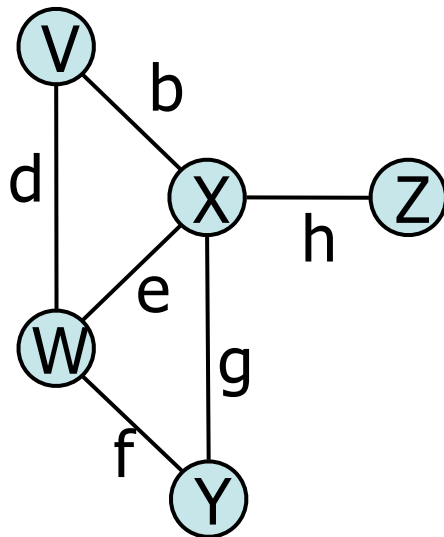
1. A common vocabulary *← Friday*
2. Graph implementations *← Here we are*
3. Graph traversals *← Friday!*
4. Graph algorithms *← Next week*



Graph ADT

Data:

- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.

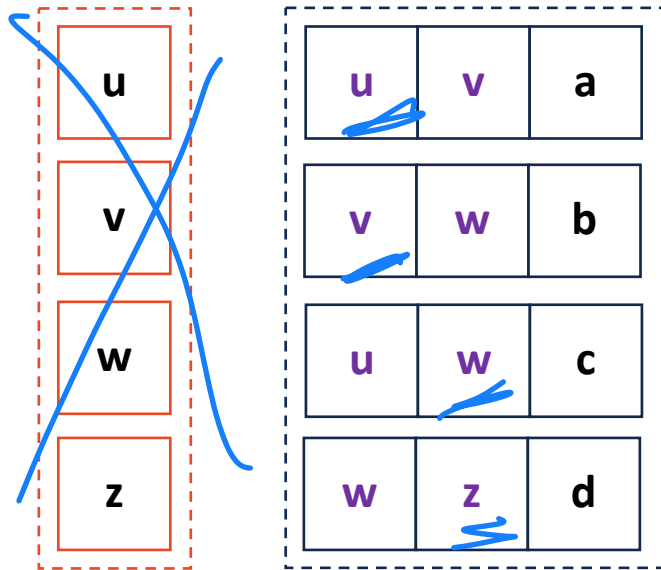
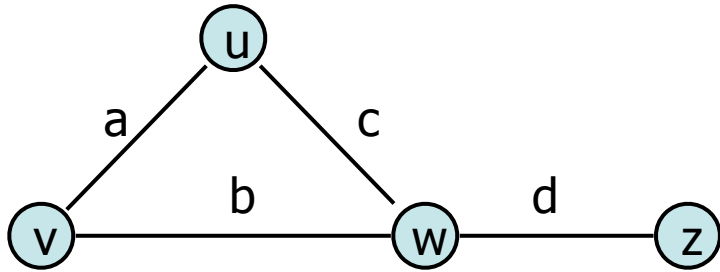


Functions:

- insertVertex(K key);
 - insertEdge(Vertex v1, Vertex v2, K key);
 - removeVertex(Vertex v);
 - removeEdge(Vertex v1, Vertex v2);
 - getEdges(Vertex v);
 - areAdjacent(Vertex v1, Vertex v2);
 - origin(Edge e);
 - destination(Edge e);
- find*

Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



Vertex Storage:

An optional list of vertices

↳ If connected graph

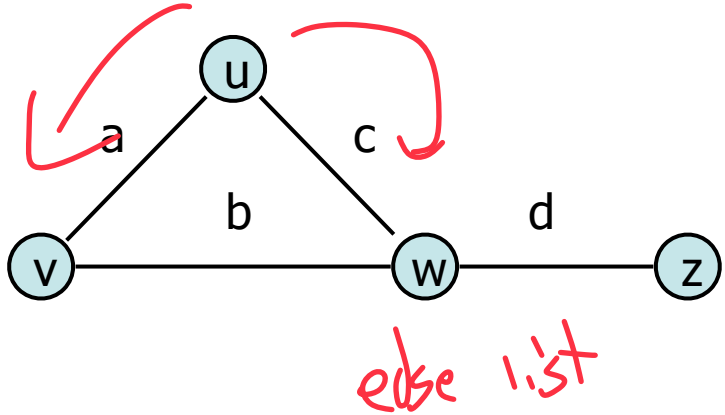
Edge Storage:

A list storing edges as (V1, V2, Weight)

Connected graphs can be stored as just an edge list!

Graph Implementation: Edge List $|V| = n, |E| = m$

$u \rightarrow v, w$



u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

$$n-1 \leq m \leq O(n^2) \approx \frac{n(n-1)}{2}$$

getEdges(Vertex v)

↳ loop through edge list
 ↳ check both v_1 & v_2

$O(1)$ 5%

$O(n)$ 25%

→ $O(n)$ 40%

$O(n+m)$ 20%

* → $O(n^2)$ 10%

areAdjacent(Vertex v1, Vertex v2)

$O(m^2)$

↳ loop through edge list
 ↳ check both v_1 & v_2

$v_2 \neq v_1$

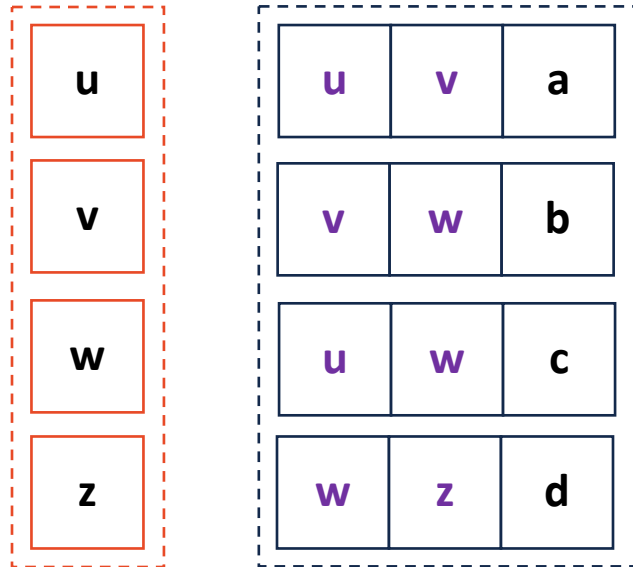
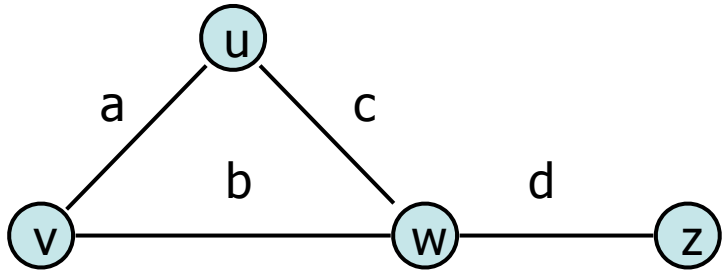
What is the Big O?

$O(m)$



Join Code: 225

Graph Implementation: Edge List $|V| = n, |E| = m$



getEdges(Vertex v)

Search edge storage for 'v'

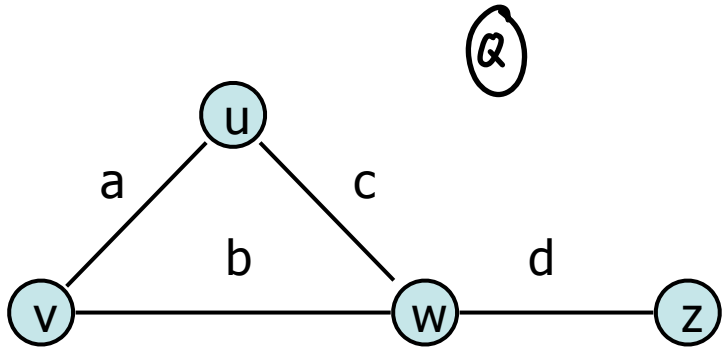
As unsorted array: $O(m)$

areAdjacent(Vertex v1, Vertex v2)

Search edge storage for 'v1' AND 'v2'

As unsorted array: $O(m)$

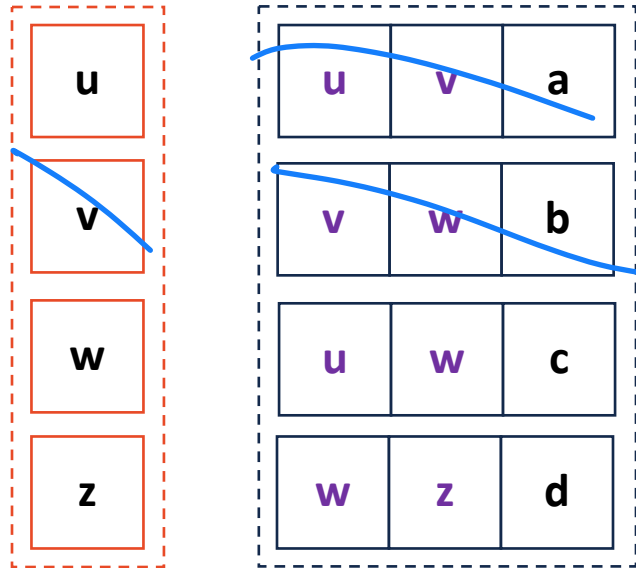
Graph Implementation: Edge List $|V| = n, |E| = m$



insertVertex(K key)

↳ Add to end of vertex list

↳ $O(1)^*$



Q

removeVertex(Vertex v)

↳ Search through vertex list, remove $\checkmark$

& Search edge list & remove all sources of v

either endpoint

What is the Big O?

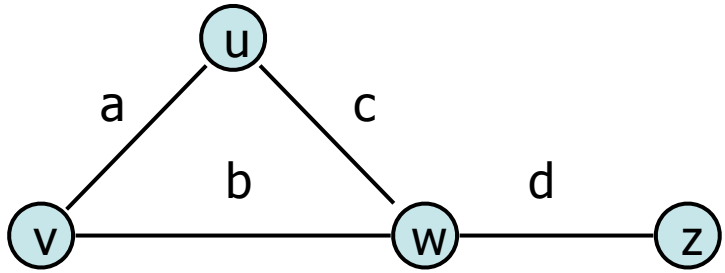
$O(n+m)$

$\sum O(n+m)$
 $10\% O(m)$
 $5\% O(n^2)$



Join Code: 225

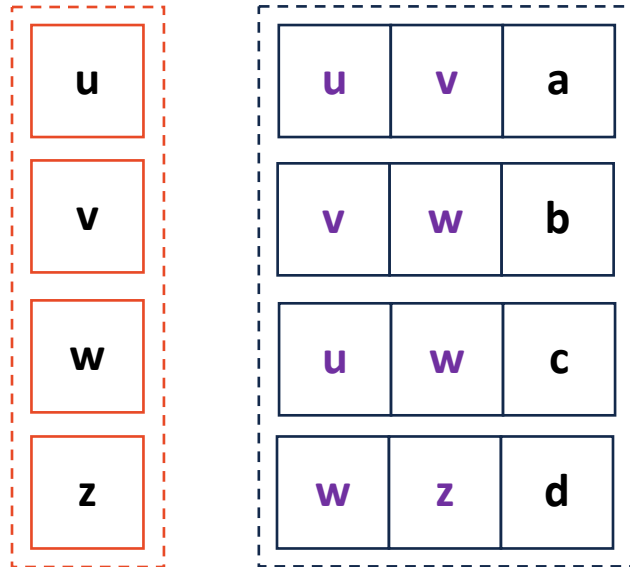
Graph Implementation: Edge List $|V| = n, |E| = m$



insertVertex(K key)

Add 'key' to vertex array

As unsorted array: $O(1)^*$



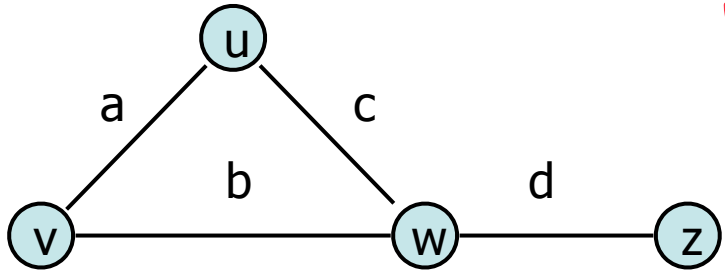
removeVertex(Vertex v)

Remove vertex from vertex array

Remove vertex from edge array

As unsorted array: $O(n+m)$

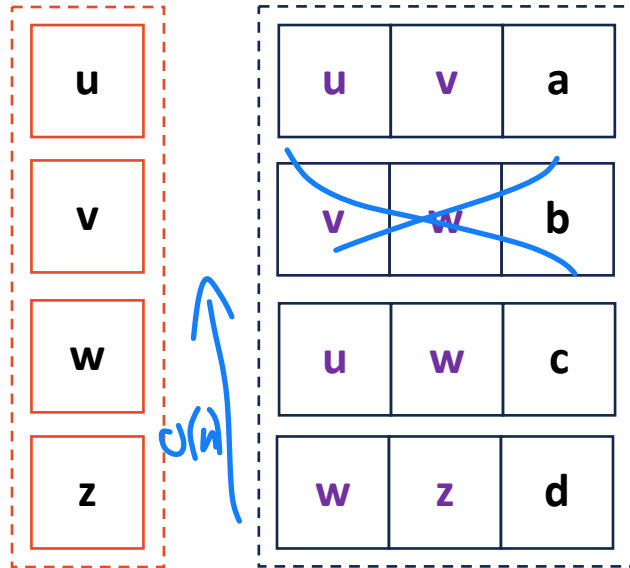
Graph Implementation: Edge List $|V| = n, |E| = m$



insertEdge(Vertex v1, Vertex v2, K key)

→ Add to back of edge list $O(1)^*$

unsorted way



$v1, v2, K$

removeEdge(Vertex v1, Vertex v2)

→ Search through edge list

→ Find specific item being removed

$O(m)$

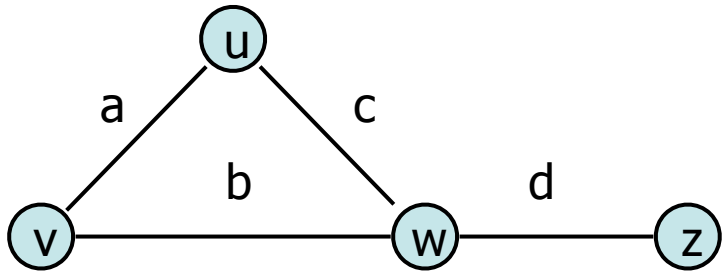
$O(m)$

What is the Big O?



Join Code: 225

Graph Implementation: Edge List $|V| = n, |E| = m$



u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

→ **insertEdge(Vertex v1, Vertex v2, K key)**

Tricky! Can we assume edge doesn't exist?

If yes — insertBack in $O(1)^*$

If no — search costs $O(m)$

removeEdge(Vertex v1, Vertex v2)

Search edge array for 'v1' and 'v2'

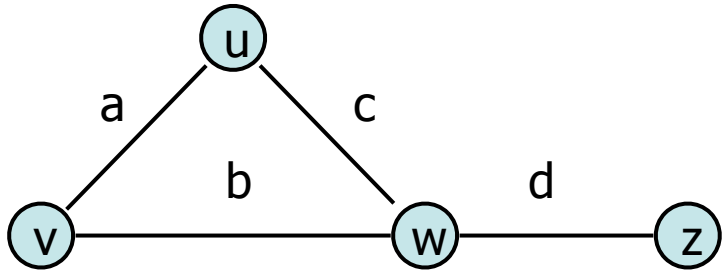
Remove it if it exists

Since unsorted array: $O(m)$

↓ (5 2 2 5)
Simple
one edge per pair

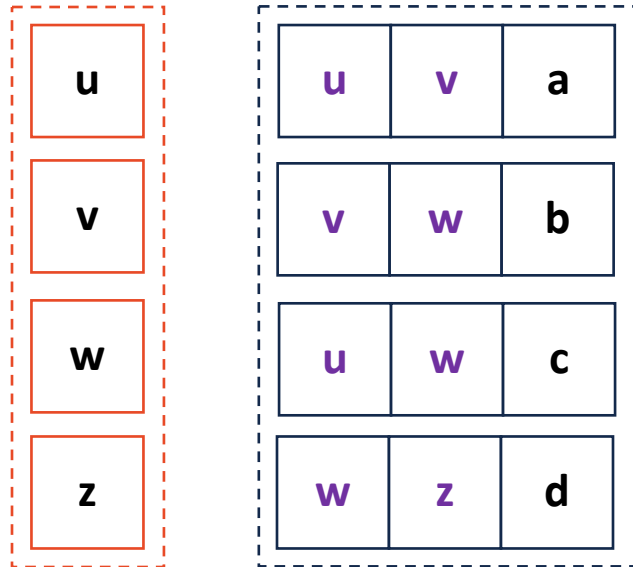
Graph Implementation: Edge List $|V| = n, |E| = m$

$O(1)^*$



insertVertex(K key)

insertEdge(Vertex v1, Vertex v2, K key)



$O(m)$

getEdges(Vertex v)

areAdjacent(Vertex v1, Vertex v2)

removeVertex(Vertex v)

removeEdge(Vertex v1, Vertex v2)

Graph Implementation: Edge List



Pros:

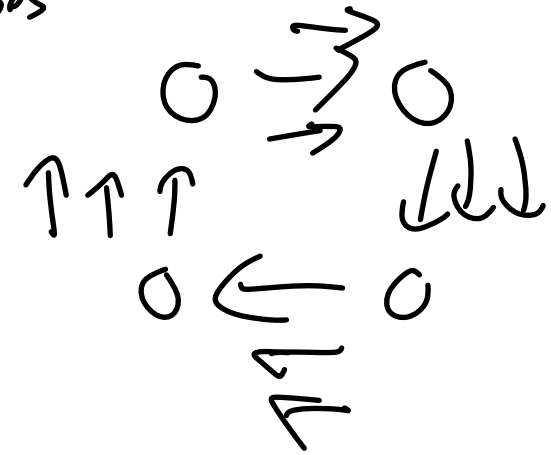
- Insert vertex or edge in $O(1)^*$ time
 - Minimal storage cost
- Good data for storage

→ Good at filtering — single for loop over edges
↳ list is ordered

Cons:

↳ All our access functions (finds)

↓
modify functions



Graph Implementation: Brainstorming better

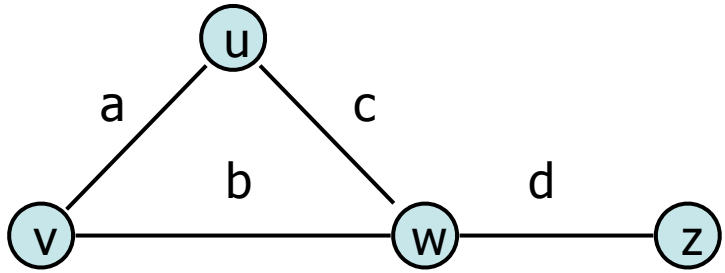
What operations might I want to do very quickly?

is Adjacent (v_1, v_2) $\rightarrow O(1)$ time

What modifications might allow me to do these things faster?

$\hookrightarrow$ A 2D Matrix

Graph Implementation: Adjacency Matrix



$O(1)$ adjacency

edge list

u		
v		
w		
z		

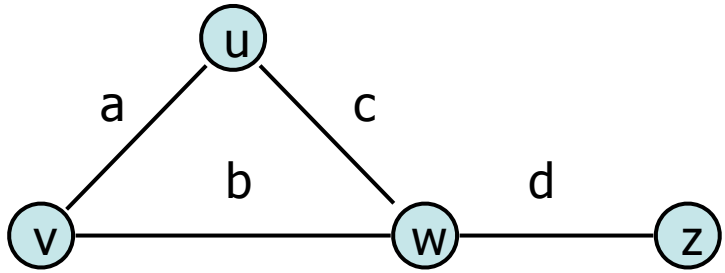
u	v	a
v	w	b
u	w	c
w	z	d

Add an Adj Matrix

	u	v	w	z
u	F	T	T	F
v	T	F	T	F
w	T	T	F	T
z	F	F	T	F

Bools

Graph Implementation: Adjacency Matrix

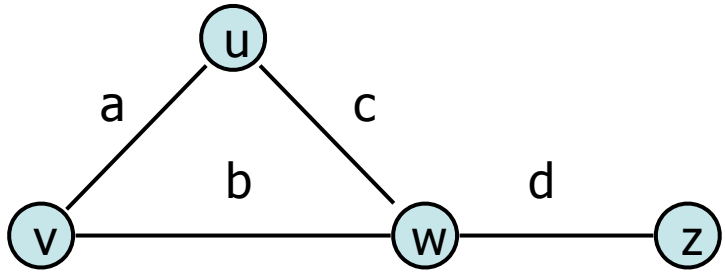


u
v
w
z

	u	v	w	z
u	—	a	c	—
v	a	—	b	—
w	c	b	—	d
z	—	—	d	—

If undirected graph:
↳ Store only upper
diagonal

Graph Implementation: Adjacency Matrix



Many ways to connect vertex w) index in matrix:

1) Map $\rightarrow O(1)$ ***
2) List of tuples $\rightarrow O(n)$

hash table
y/n

u	0
v	1
w	2
z	3

$O(1)$
Black Box

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

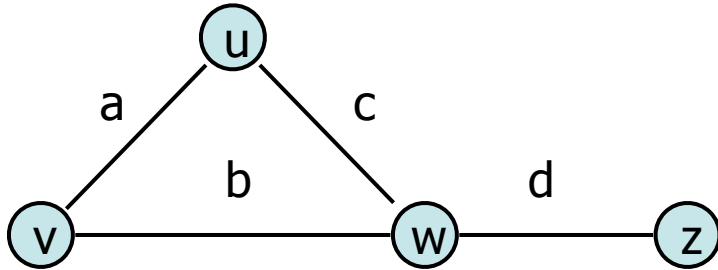
find()
↓
index return

4) Search tree $O(\log n)$

$O(n)$

Graph Implementation: Adjacency Matrix

$$|V| = n, |E| = m$$



Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

*O(1)** hash table
(Trust me :))*



Edge Storage:

A $|V| \times |V|$ matrix of edges

Weight is stored at position (u, v)

u	0
v	1
w	2
z	3

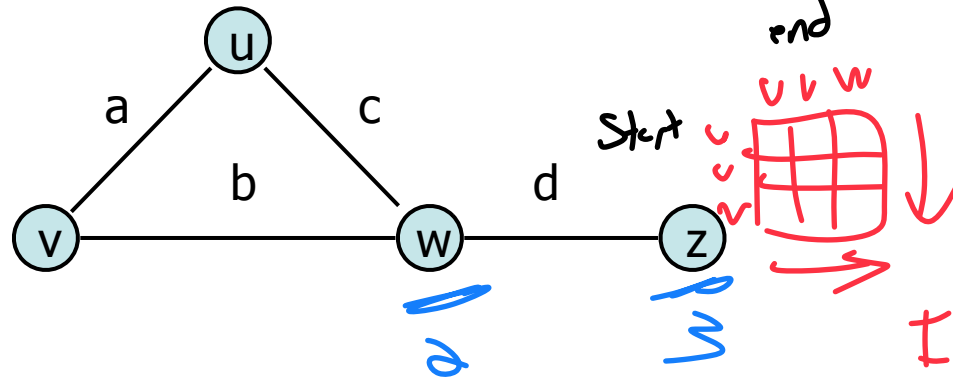
	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$

only upper diagonal

getEdges(Vertex v):



↳ If undirected
 ↳ look up row/col index
 ↳ Look at every entry in (i, c)
 (r, i)

If directed: look up row or column incident Edges
Outgoing Edges

areAdjacent(Vertex v1, Vertex v2):

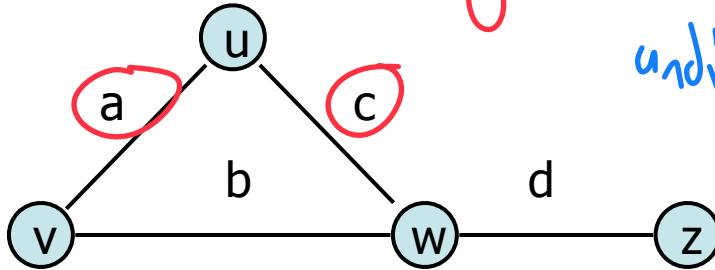
↳ $M[v1][v2]$

make sure $v1 < v2$

$M[3][2] \rightarrow M[2][3]$

	0	1	2	3
u	0			
v	1	-	a	c
w	2		-	b
z	3			d

Why row & col?
 $\begin{bmatrix} [a, 1, 0] \leftarrow 0 \\ [c, 0] \leftarrow 1 \\ [d] \leftarrow 2 \end{bmatrix}$

$$|V| = n, |E| = m$$


undirected 

getEdges(Vertex v):

$$O(n)$$

Look up row (or column) in matrix

↳ At most $n-1$ edges to look up

→ we must filter & return non-zero entries!

Directed also $O(n)$ [return filter on full $\frac{1}{2} \log n$]

areAdjacent(Vertex v1, Vertex v2):

- Look up row (and column) in matrix

u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

$$\hookrightarrow O(1)$$

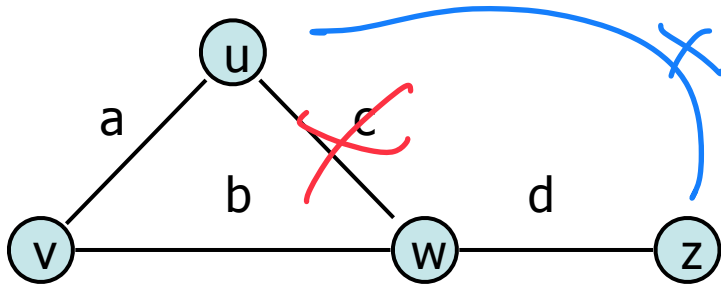
$$n = 1 \text{ billion}$$
[illegible]

* B'g Weakness of Math, 18

What are my Big Os?

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



insertEdge(Vertex v1, Vertex v2, K key):

↳ Look up $[V1][V2]$ & modify value!

removeEdge(Vertex v1, Vertex v2, K key):

↳ Look up $[V1][V2]$ & modify value

u	0
v	1
w	2
z	3

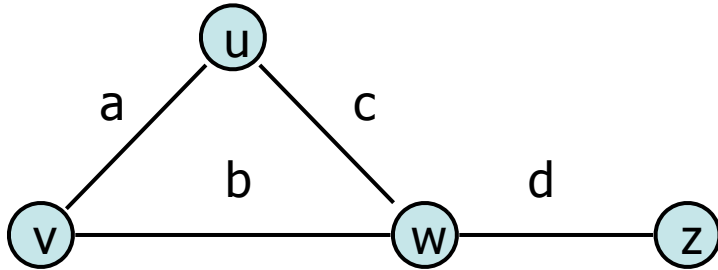
	0	1	2	3
0	-	a	~~c~~	~~0~~
1		-	b	0
2			-	d
3				-

$O(1)$ ☺

Graph Implementation: Adjacency Matrix



$|V| = n, |E| = m$



insertEdge(Vertex v1, Vertex v2, K key):

Look up row and column

Change value

removeEdge(Vertex v1, Vertex v2, K key):

Look up row and column

Change value

u	0
v	1
w	2
z	3

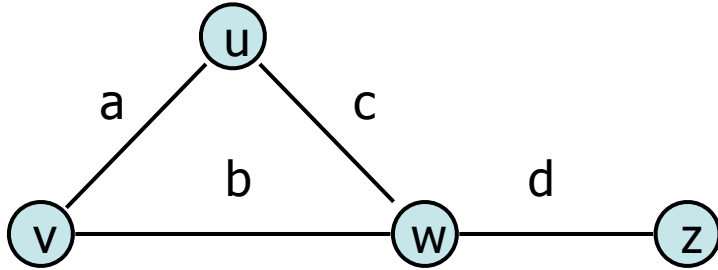
	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

What are my Big Os?

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$

insertVertex(K key):

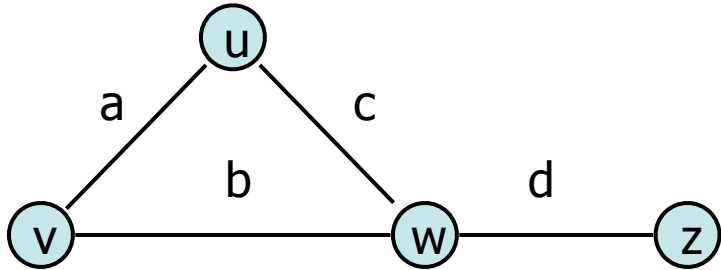


u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



insertVertex(K key):

Insert new key to vertex mapping

Add a new row or column to matrix

Add a new item to each old row / column

u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-



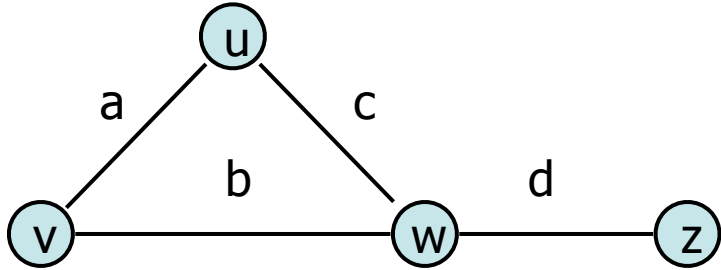
Join Code: 225

What are my Big Os?

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$

removeVertex(Vertex v):

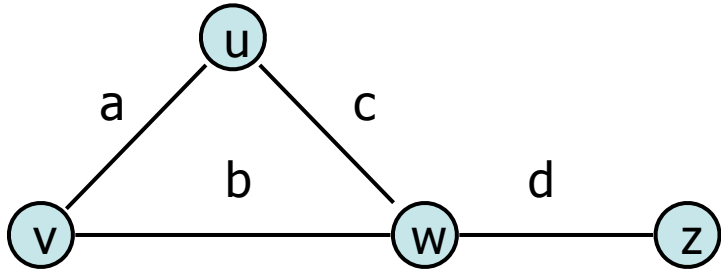


u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



removeVertex(Vertex v):

Join Code: 225

Remove v from vertex mapping

Remove an entire row / entire column

u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

What are my Big Os?

Amortized Removal with Tombstoning

Remove an item by replacing its value or flipping a flag indicating 'deletion'

2	7	5	9	7	14	1	0	8	3
---	---	---	---	---	----	---	---	---	---

Turns an $O(n^2)$ cost into an $O(1)$ cost... by not actually deleting anything.

Amortized Removal with Tombstoning

Remove an item by replacing its value or flipping a flag indicating 'deletion'

2	7	5	9	7	14	1	0	8	3
---	---	---	---	---	----	---	---	---	---

When there are enough deleted elements to merit resize, do it all at once!

u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Adjacency Matrix:

removeVertex() by tombstoning $|V|$ values

Resize when needed or by request

Graph Implementation: Adjacency Matrix



Pros:

Cons:

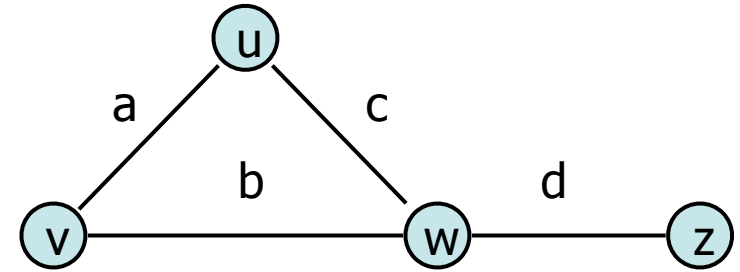
Graph Implementation Brainstorming

We want something...

Faster than an edge list

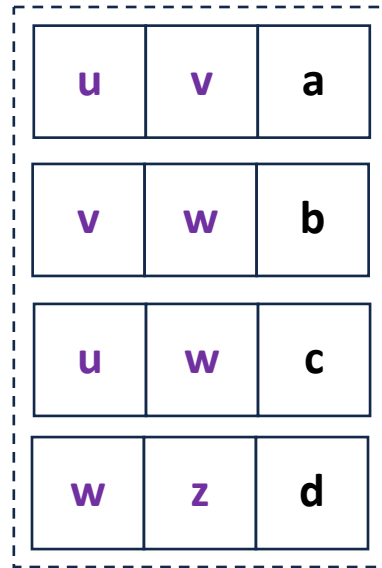
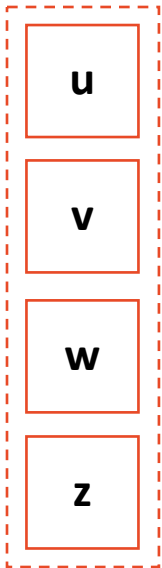
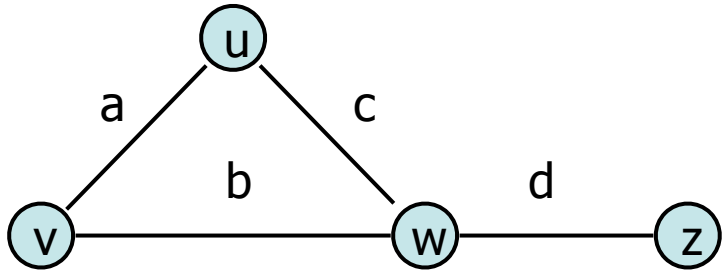
Less space than an adjacency matrix

Particularly good at **finding all adjacent elements (neighbors)**



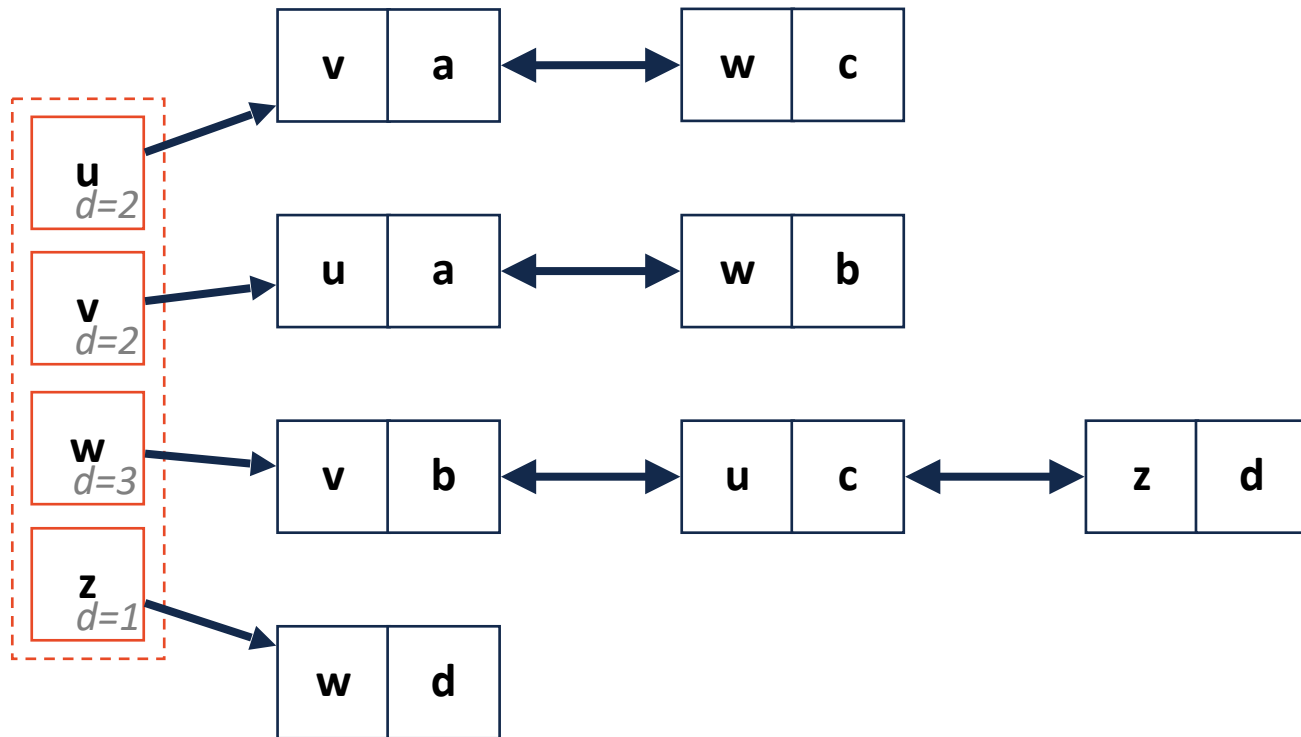
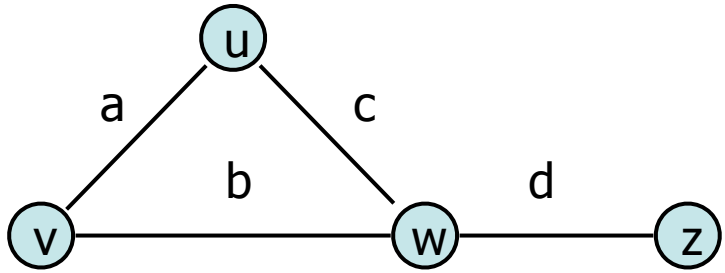
Graph Implementation: Edge List + ?

$$|V| = n, |E| = m$$



Graph Implementation: Adjacency List

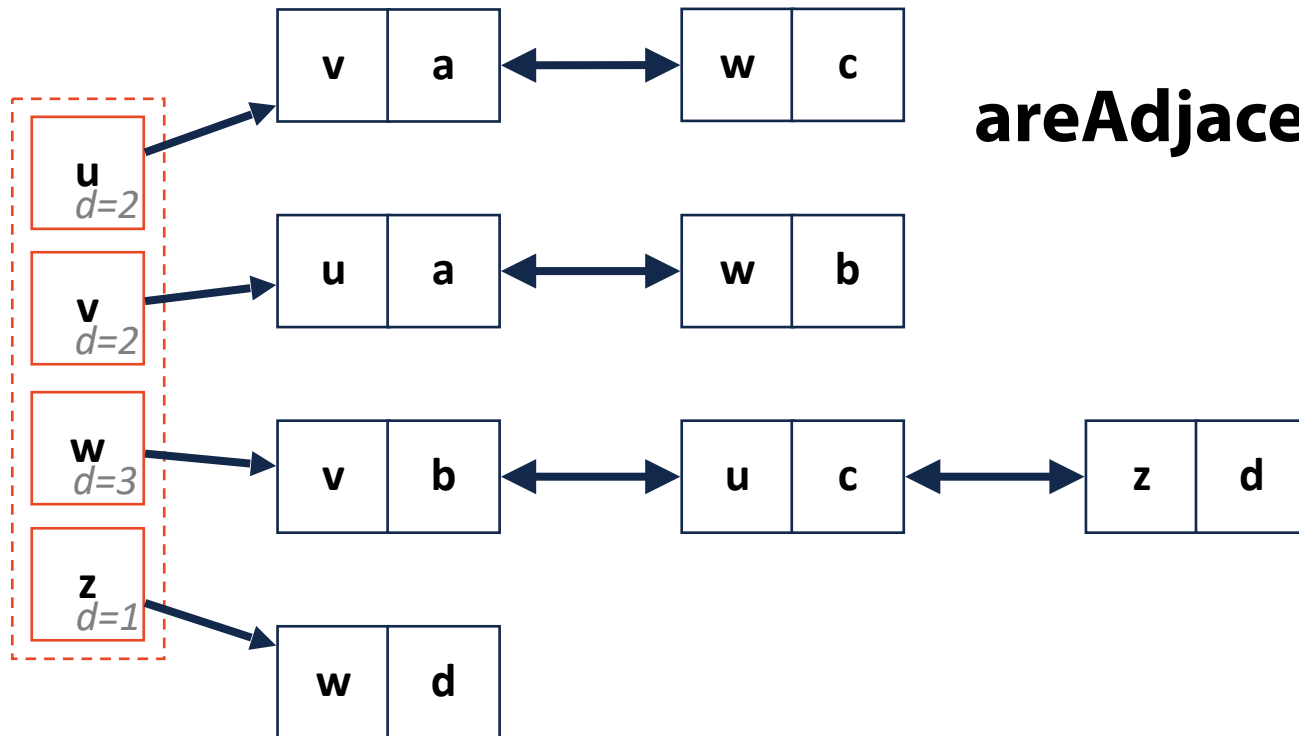
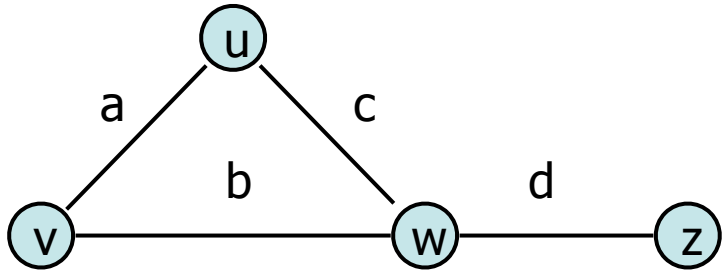
$$|V| = n, |E| = m$$



Graph Implementation: Adjacency List

$|V| = n, |E| = m$

getEdges(Vertex v):

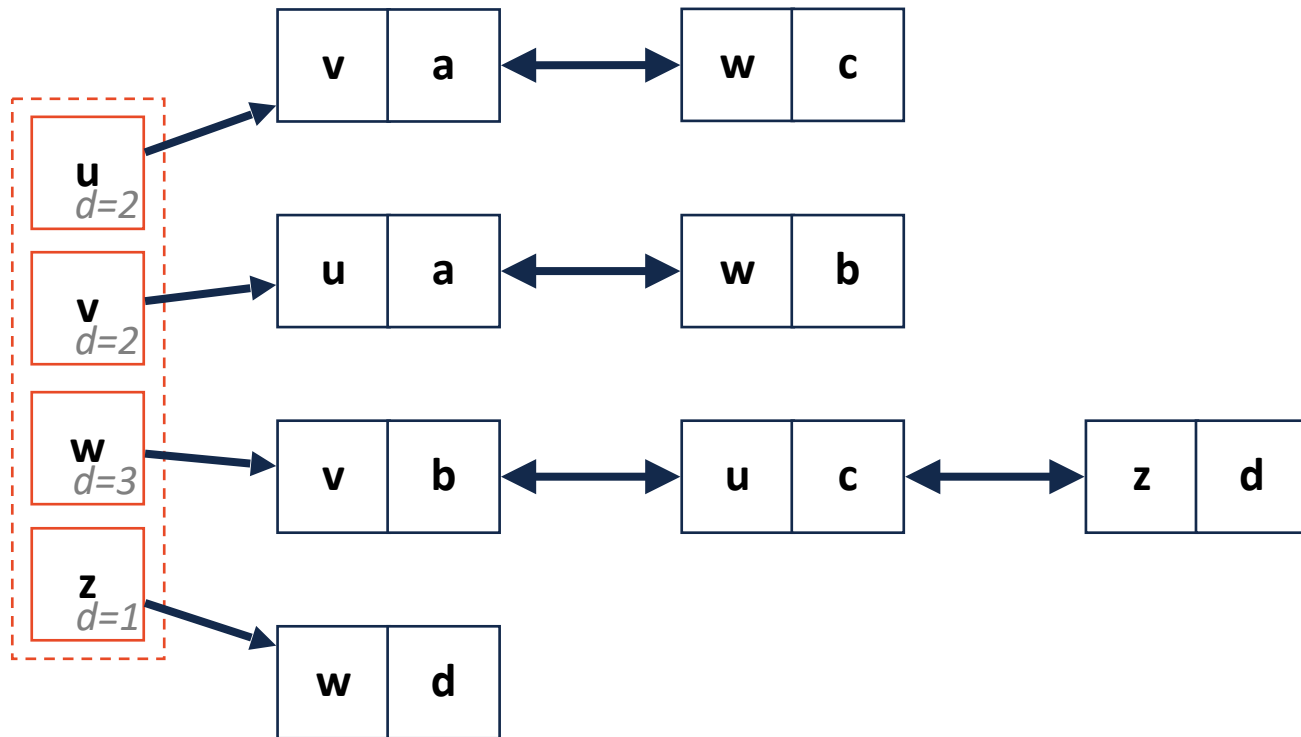
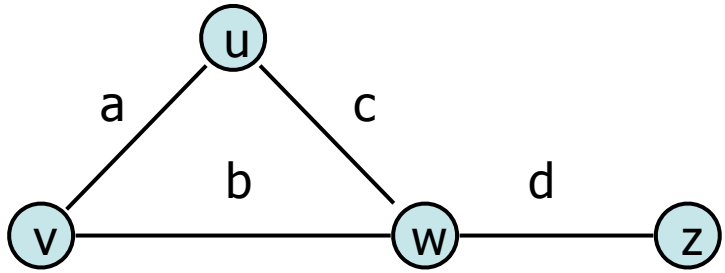


areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency List

$|V| = n, |E| = m$

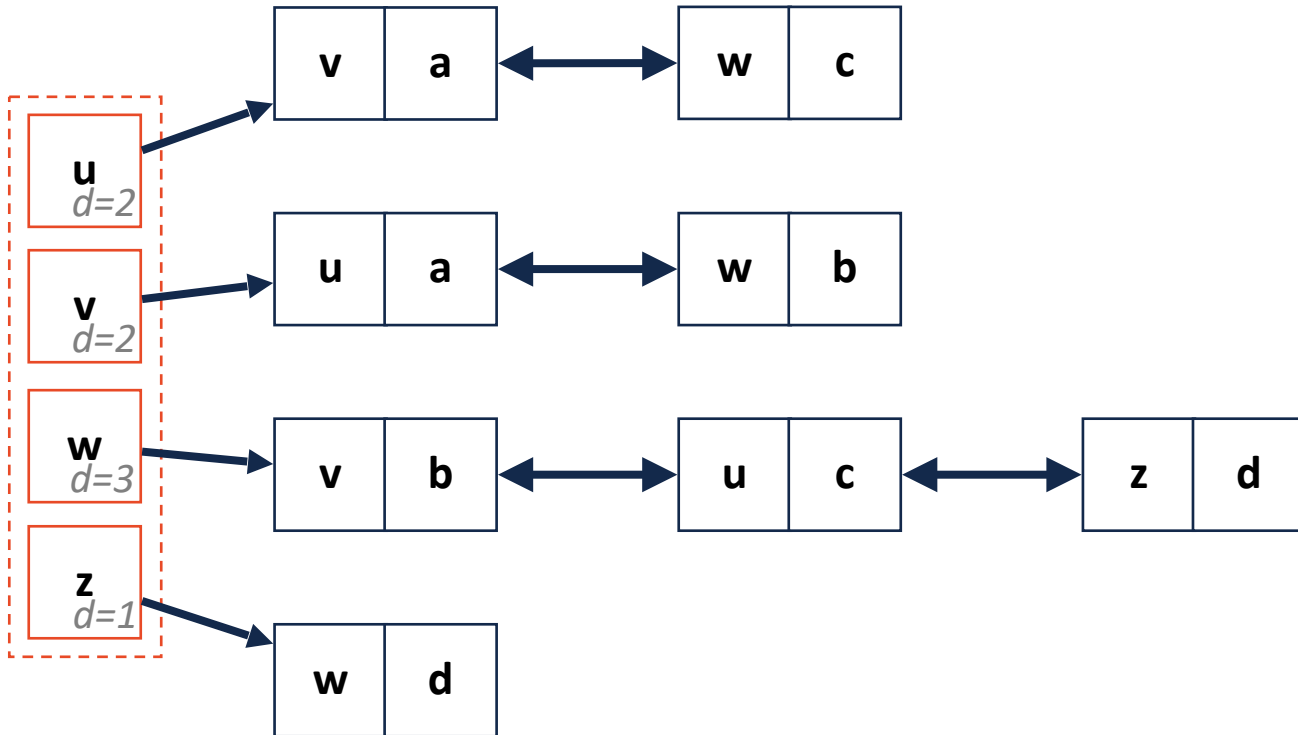
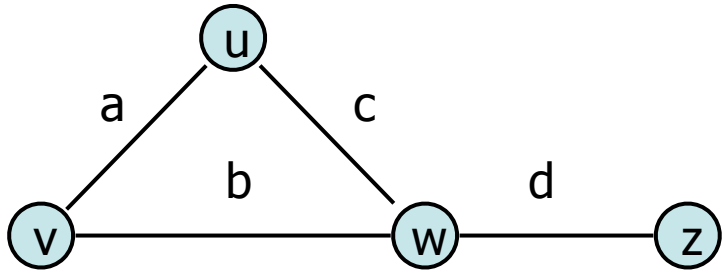
removeEdge(Vertex v1, Vertex v2, K key):



Graph Implementation: Adjacency List

$$|V| = n, |E| = m$$

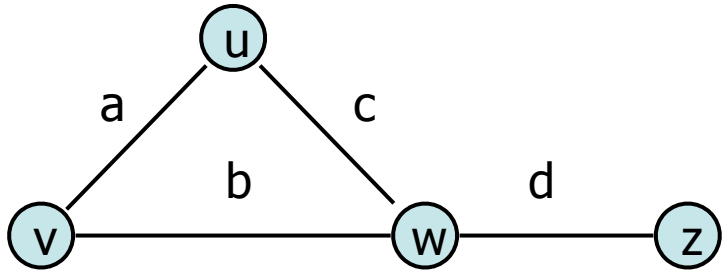
removeVertex(Vertex v):



Graph Implementation: Adjacency List



$$|V| = n, |E| = m$$



What's wrong with our implementation?

How can we fix it?

