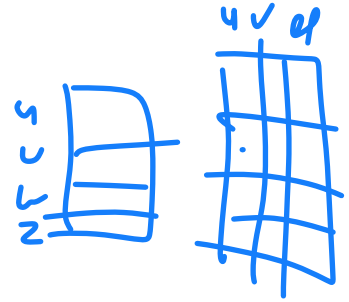
Data Structures

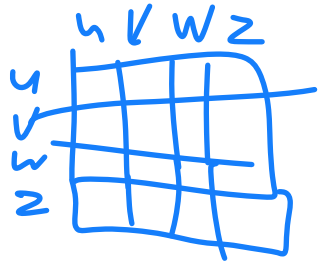
Graph Implementations 2

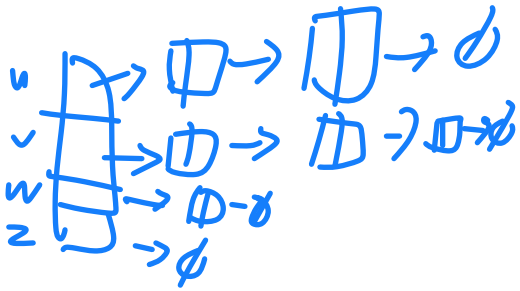
CS 225

Brad Solomon

October 28, 2026







UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

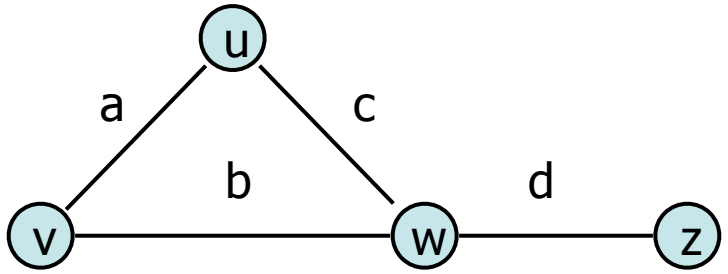
Review adjacency matrix graph implementation

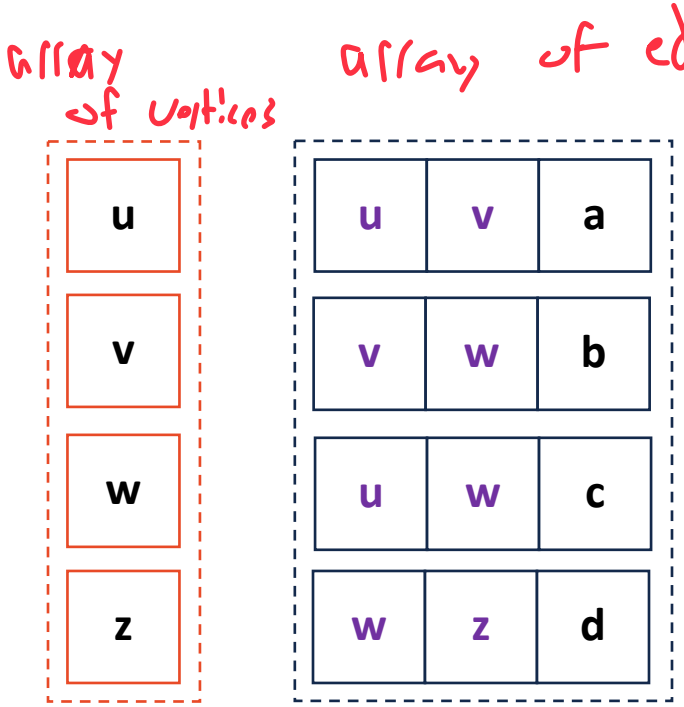
Explore possible implementation tradeoffs

Introduce adjacency list graph implementation

Discuss graph traversal algorithms

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





O(1)* unsorted list

insertVertex(K key):

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

O(m) unsorted list

removeVertex(Vertex v):

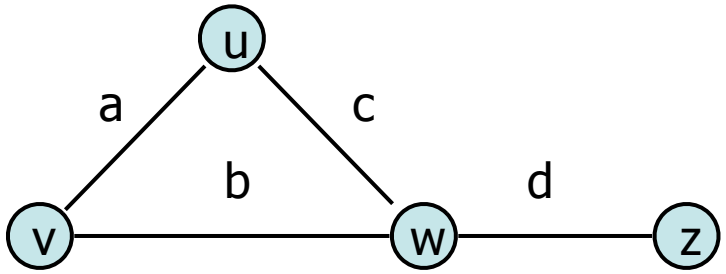
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

u	0
v	1
w	2
z	3

Edge Storage:

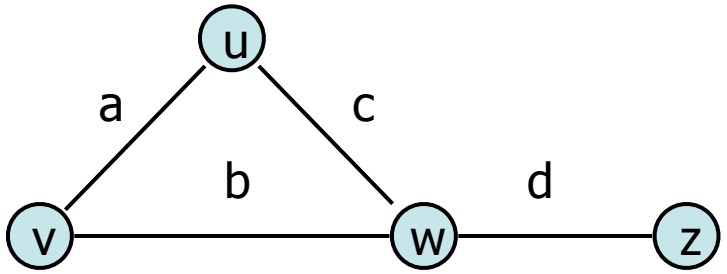
A |V| x |V| matrix of edges

Weight is stored at position (u, v)

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

Graph Implementation: Adjacency Matrix

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



u	0
v	1
w	2
z	3

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

getEdges(Vertex v):

Look up row (or column) in matrix

O(n) for both directed / undirected

areAdjacent(Vertex v1, Vertex v2):

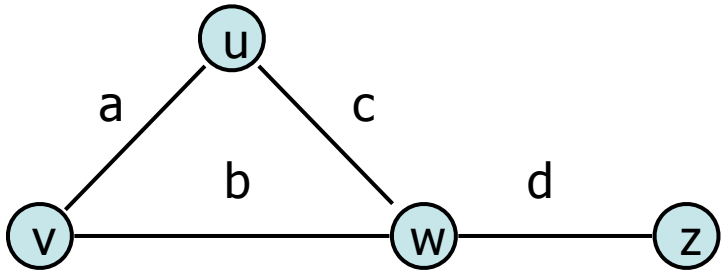
Look up row (and column) in matrix

O(1)

Graph Implementation: Adjacency Matrix



|V|= n, |E|= m



u	0
v	1
w	2
z	3

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

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

Look up row and column

Change value

O(1)

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

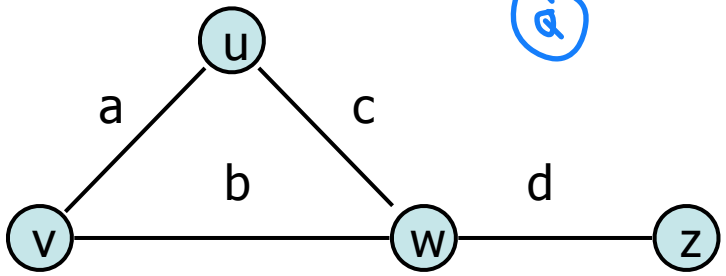
Look up row and column

Change value

O(1)

Graph Implementation: Adjacency Matrix

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



insertVertex(K key):

↳ Add vertex to vertex storage $(O(1)^{***})$

↳ Add row / column

u	0
v	1
w	2
z	3

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

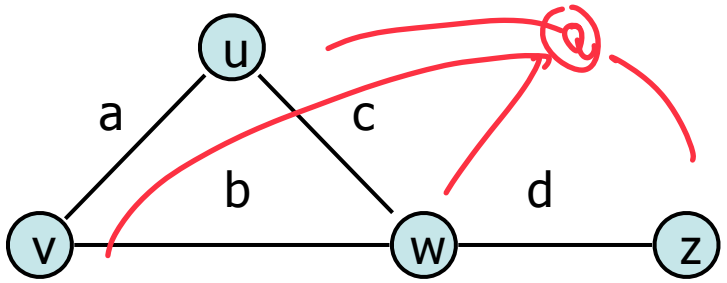
4

4

Q | 4

Graph Implementation: Adjacency Matrix

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



insertVertex(K key):

Join Code: 225

Insert new key to vertex mapping → O(1)*

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				-

4

reallocate entire matrix → O(n²) 50%

adding n items → O(n)* 30%

total inserts (insertBack) O(1)* per array → O(n) 10%

O(n+m) 10%

* = amortized

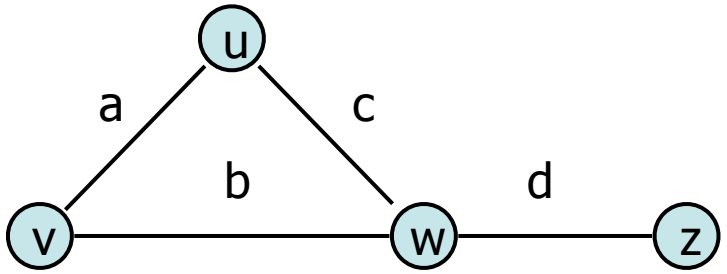
If directed O(n)

If undirected first O(1)

What are my Big Os?

Graph Implementation: Adjacency Matrix

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



u	0
v	1
w	2
z	3

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

removeVertex(Vertex v):

Remove v from vertex mapping $O(1)^{**}$

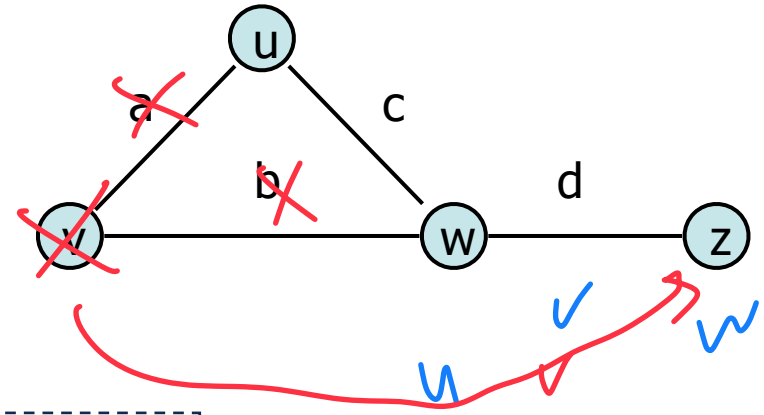
Remove an entire row / entire column

Naive: Removing $n-1$ items from $n-1$ arrays
↳ $O(n^2)$

Better: Swap removed item to back $O(n)^*$
↳ remove back $n-1$ times

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



u	0
v	1
w	2
z	3

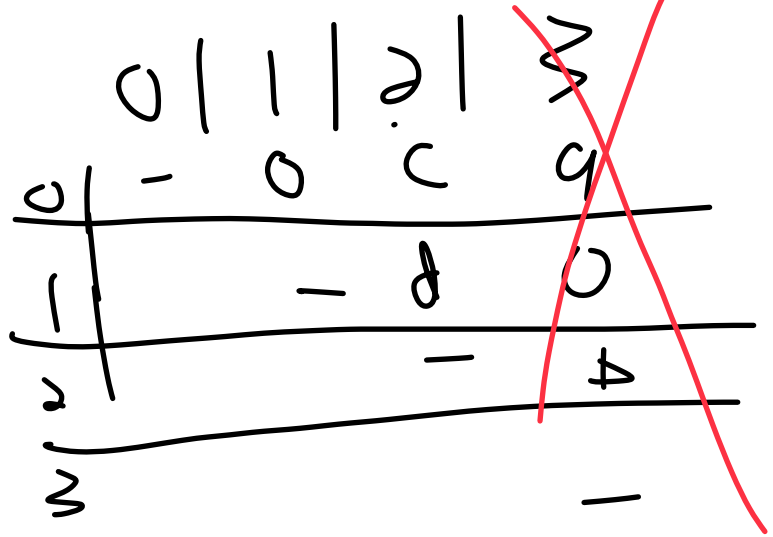
	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

removeVertex(Vertex v):

Swap target vertex w/ last vertex

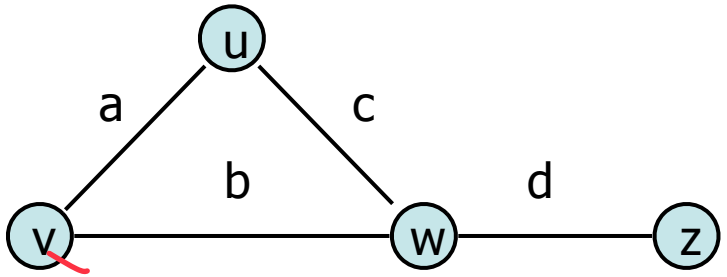
Swap matrix values w/ last vertex

Now mutual removeBack()



Graph Implementation: Adjacency Matrix

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



removeVertex(Vertex v):

Swap target vertex w/ last vertex

Swap matrix values w/ last vertex

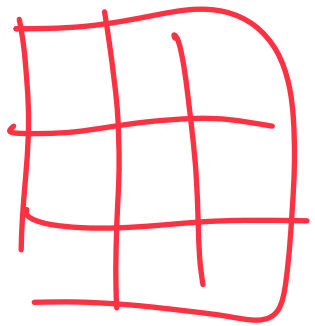
Now mutual removeBack()

This is the only tombstone

	u	0
	v	1
	w	2
	z	3

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

Visually what's getting removed



reallocate

$O(n^2)$

$O(n)$

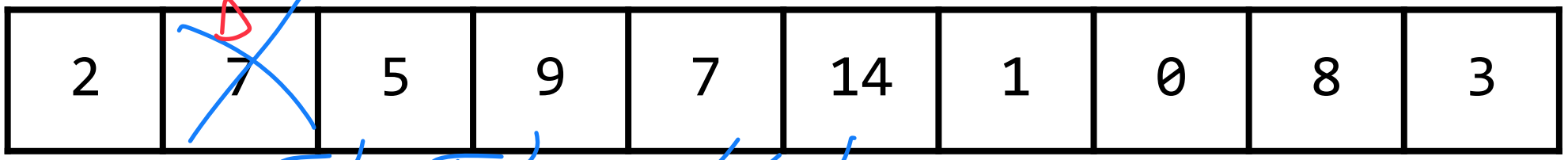
How many items do we want to do this?

Amortized Removal with Tombstoning

Heuristic

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

Tombstone: 1 bit flag (Deleted)

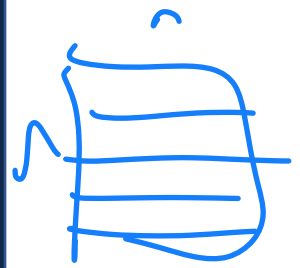


Turns an O(n^2) cost into an O(1) cost… by not actually deleting anything.

(It's actually more like O(n) cost with proper amortization)*

When enough tombstones are placed, process all removal at once!

↳ "enough" == half array? ↳ Now I can downsize to half size



→ n/2 → O(n) to make new array / O(1) removal calls

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:

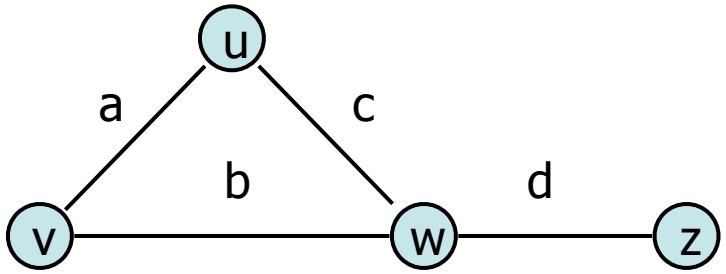
removeVertex() by tombstoning |V| values

Resize when needed or by request

Graph Implementation: Adjacency Matrix



|V|= n, |E|= m



	u	v	w	z
u	-	a	c	0
v		-	b	0
w			-	d
z				-

u
v
w
z

O(1)

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

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

areAdjacent(Vertex v1, Vertex v2):

O(n)

getEdges(Vertex v):

O(n)* — $O(n^2)$

insertVertex(K key): → O(n)*

removeVertex(Vertex v): multiple possible O(n*)

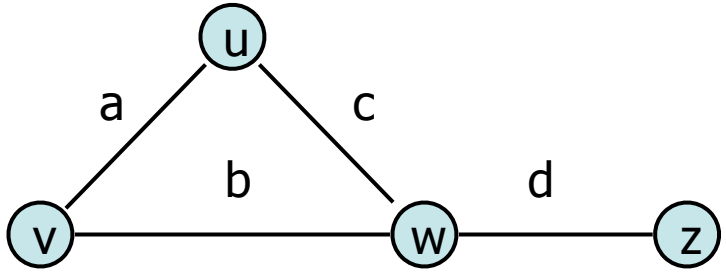
Graph Implementation Brainstorming

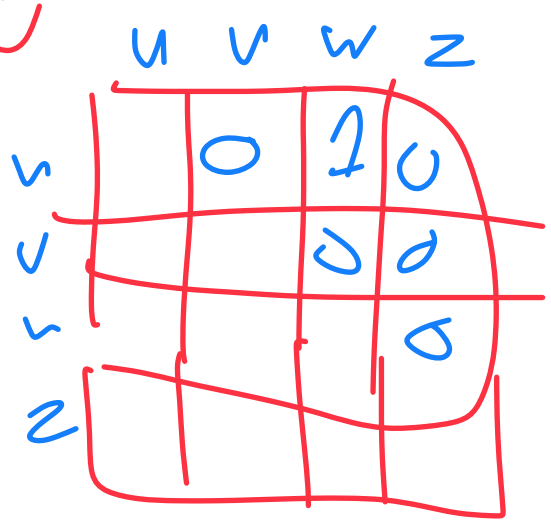
We want something…

Faster than an edge list

Less space than an adjacency matrix

Particularly good at **finding adjacent elements**





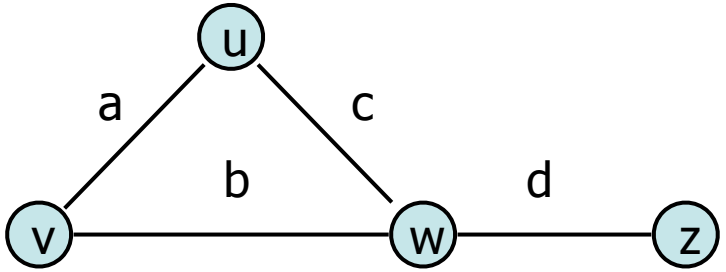
Always stores $O(n^2)$ values

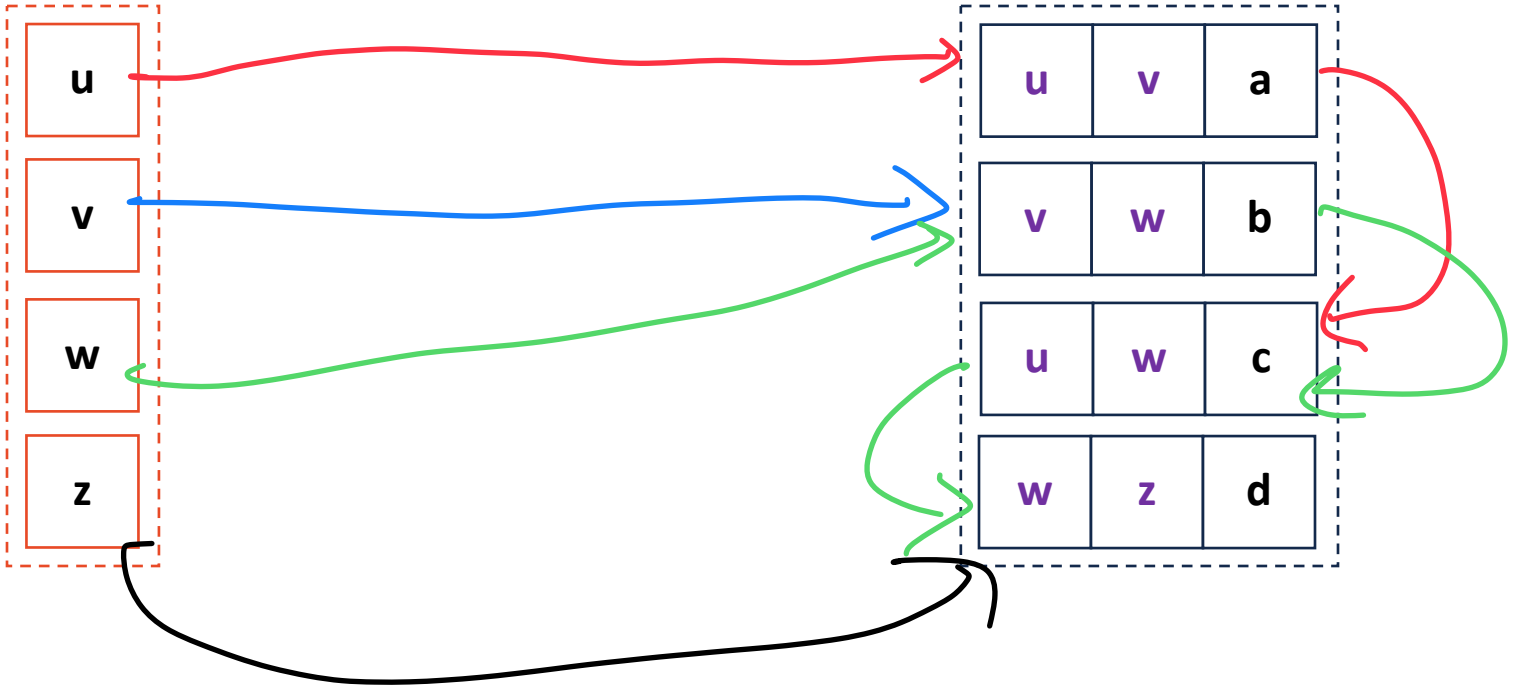
$n-1 \leq m \leq O(n^2)$

Graph Implementation: Edge List + ?

|V|= n, |E|= m

Stuck one more list on edge list



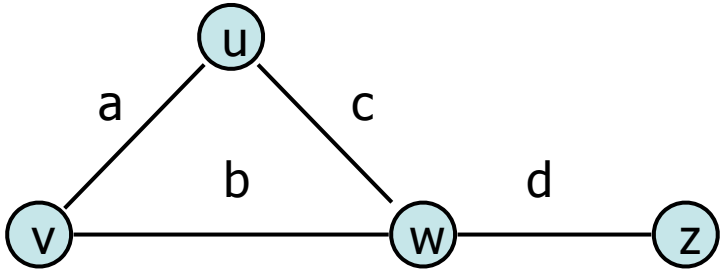


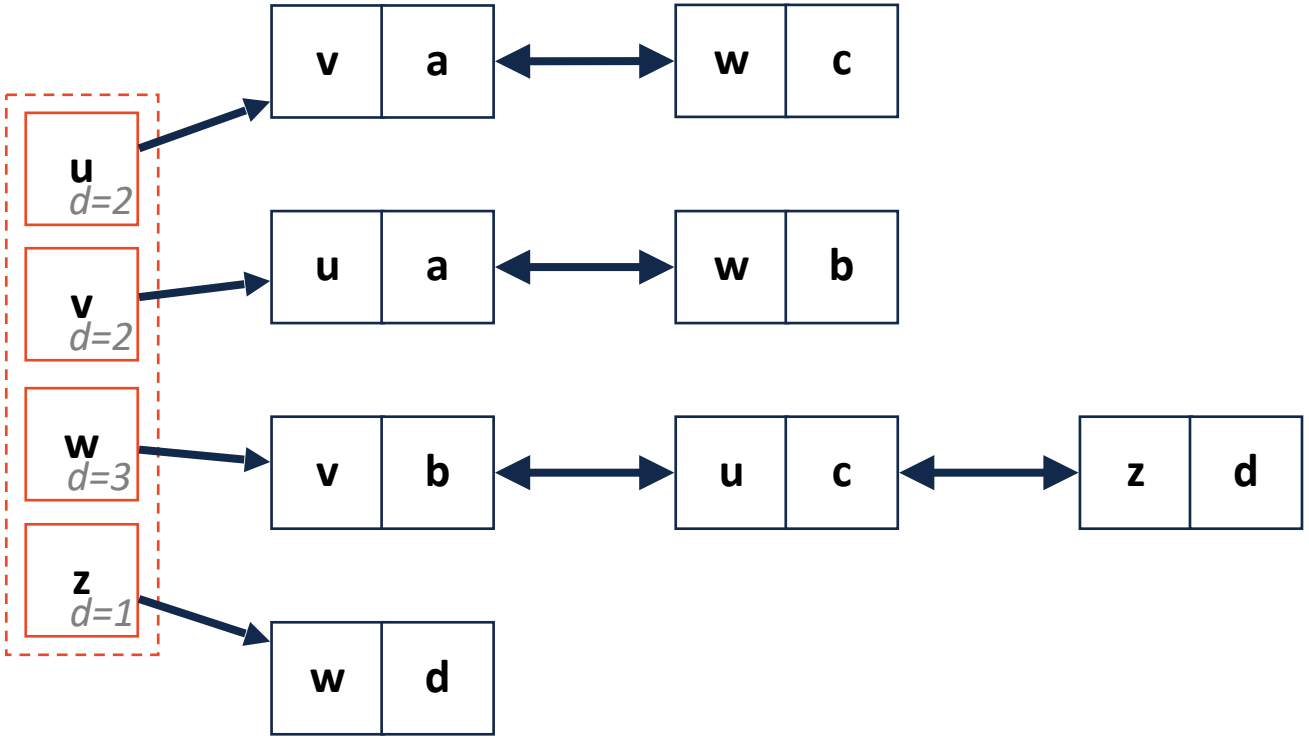
Naive Adjacency List

|V|= n, |E|= m

One linked list for vertex!

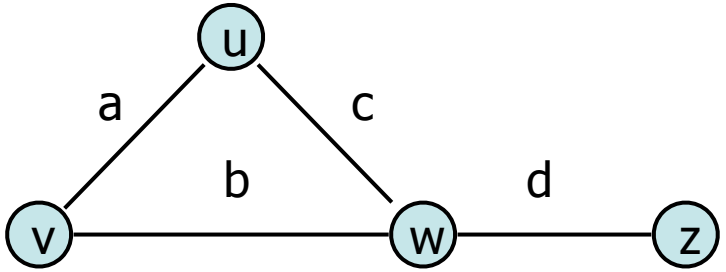
Store only adjacent edges

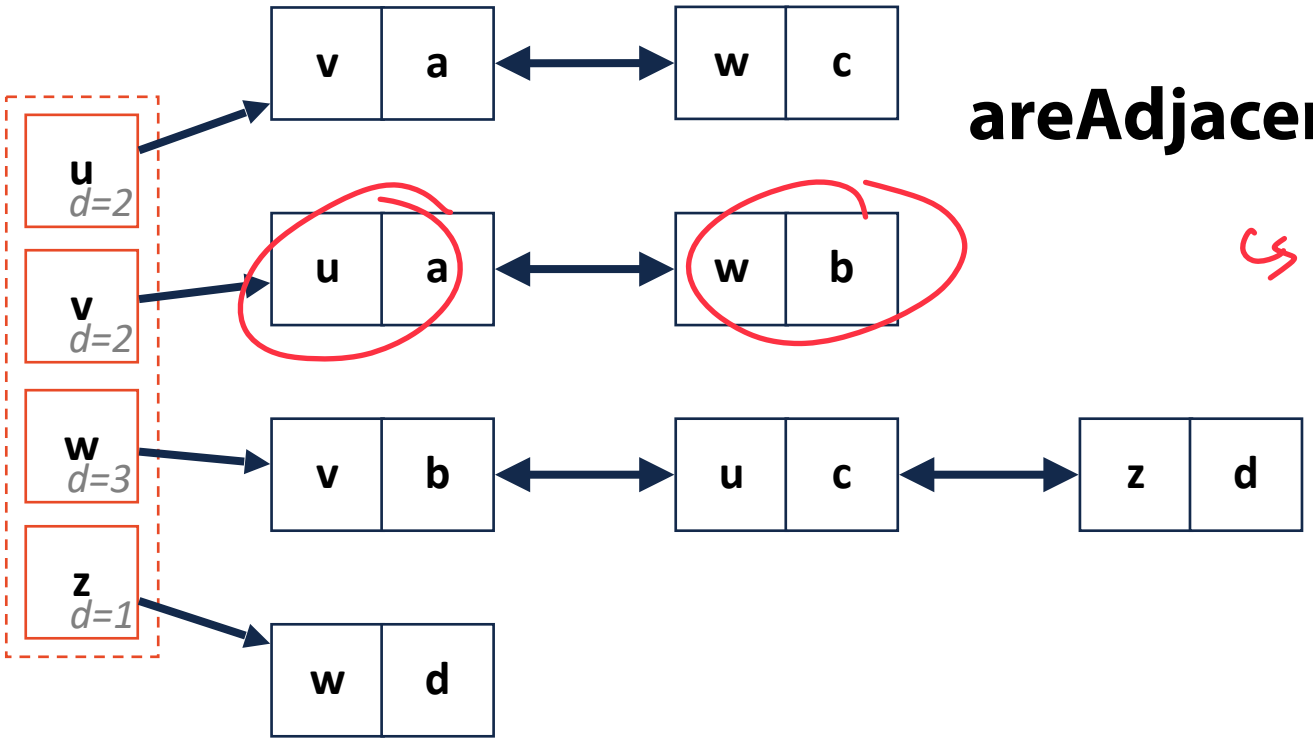




Naive Adjacency List

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





getEdges(Vertex v):

↳ walk along linked list for v

↳ return ALL values

areAdjacent(Vertex v1, Vertex v2):

↳ walk along LL for v_1 or v_2

↳ find v_2 or v_1

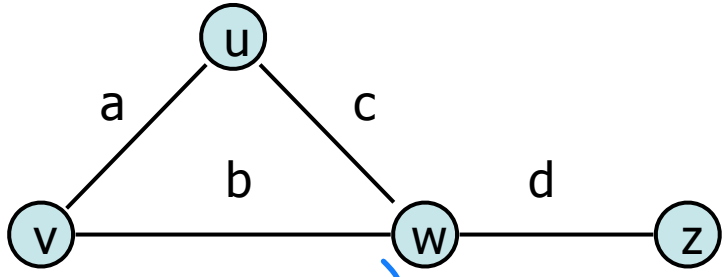
B.S O here?

Naive Adjacency List

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

$\text{degree}(v) \leq n-1$

Join Code: 225



getEdges(Vertex v):

Lookup linked list for v

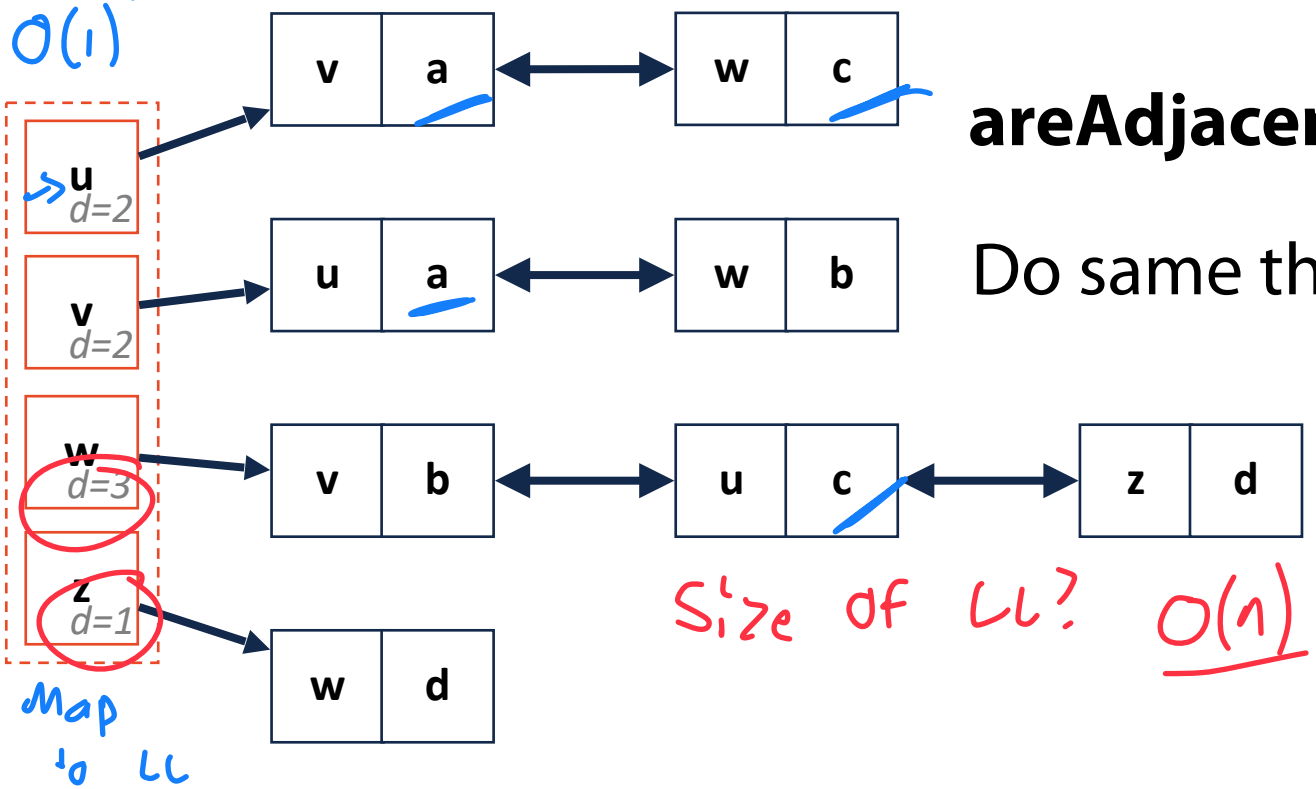
Walk down list! Every edge is incident.

degree(v)

Start

O(1) ***

end



areAdjacent(Vertex v1, Vertex v2):

Do same thing but look for specific v2

$\min[\deg(v_1), \deg(v_2)]$

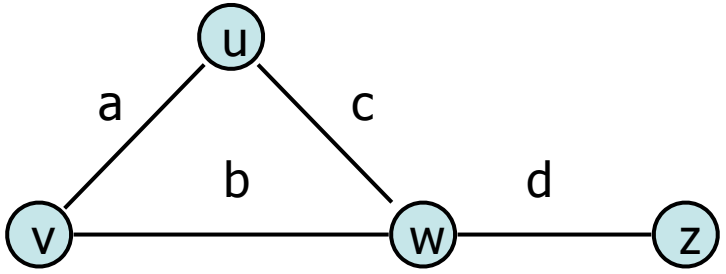
O(n) ← more right

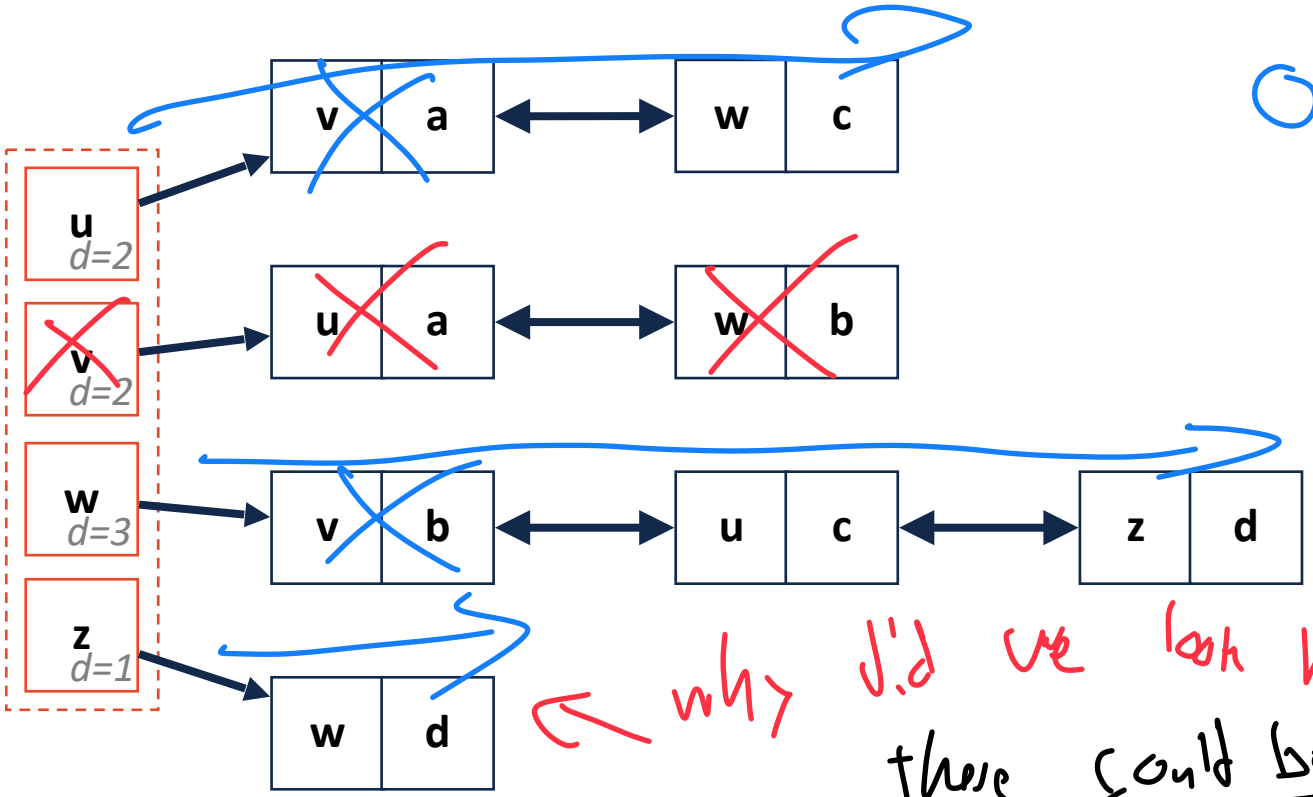
O(m)

What are my Big Os?

Naive Adjacency List

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





removeVertex(Vertex v):

↳ remove all items in LL ✓

↳ remove v from map

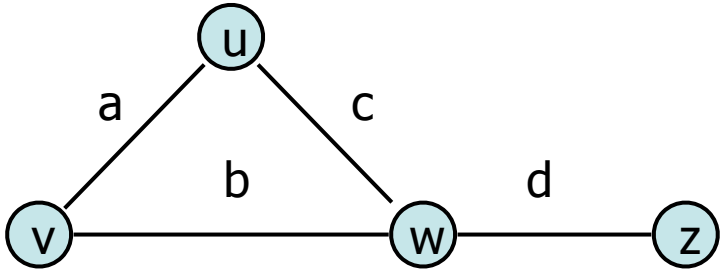
Oh no.... b/c undirected graph every edge exists twice!

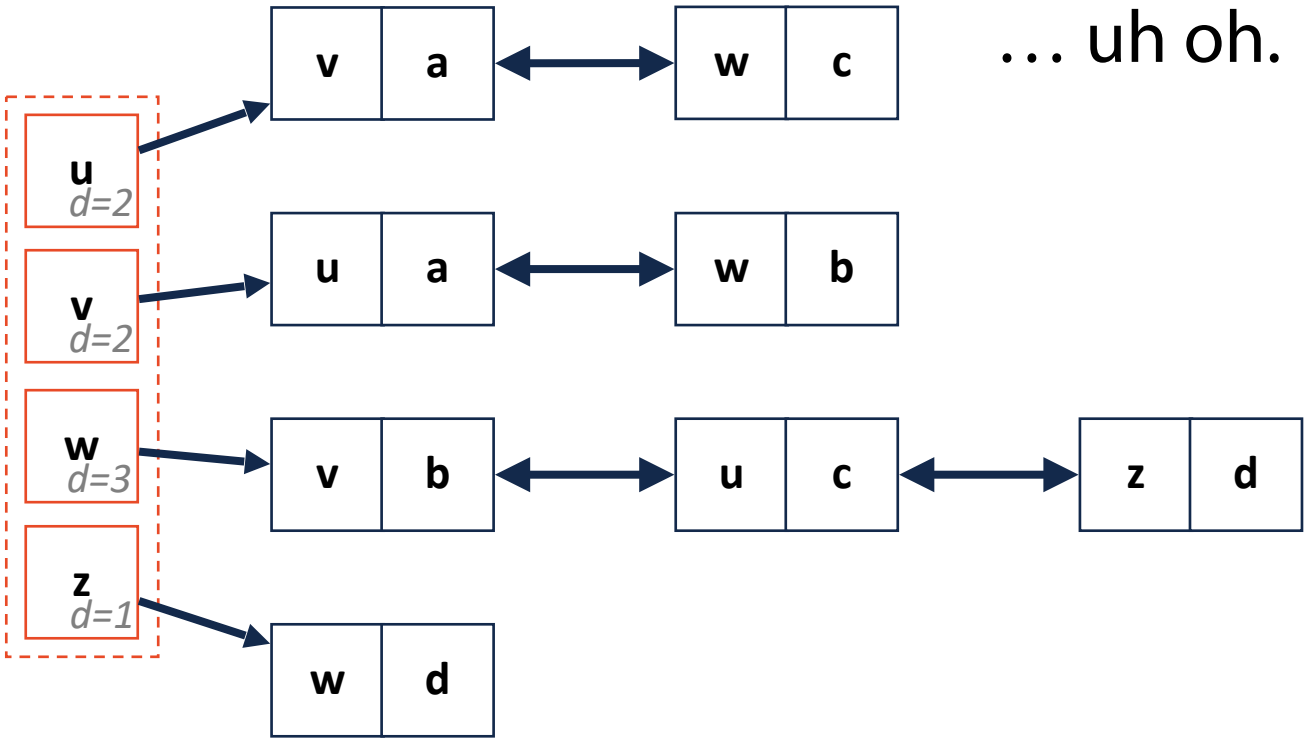
$O(n^2)$ look at every edge!!

why did we look here? these could be a v in this list.

Naive Adjacency List

|V|= n, |E|= m





removeVertex(Vertex v):

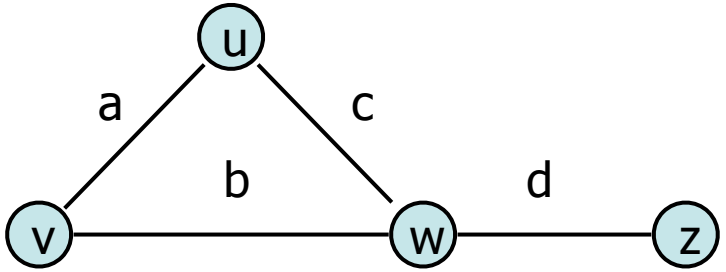
Look up V in our list and remove all entries.

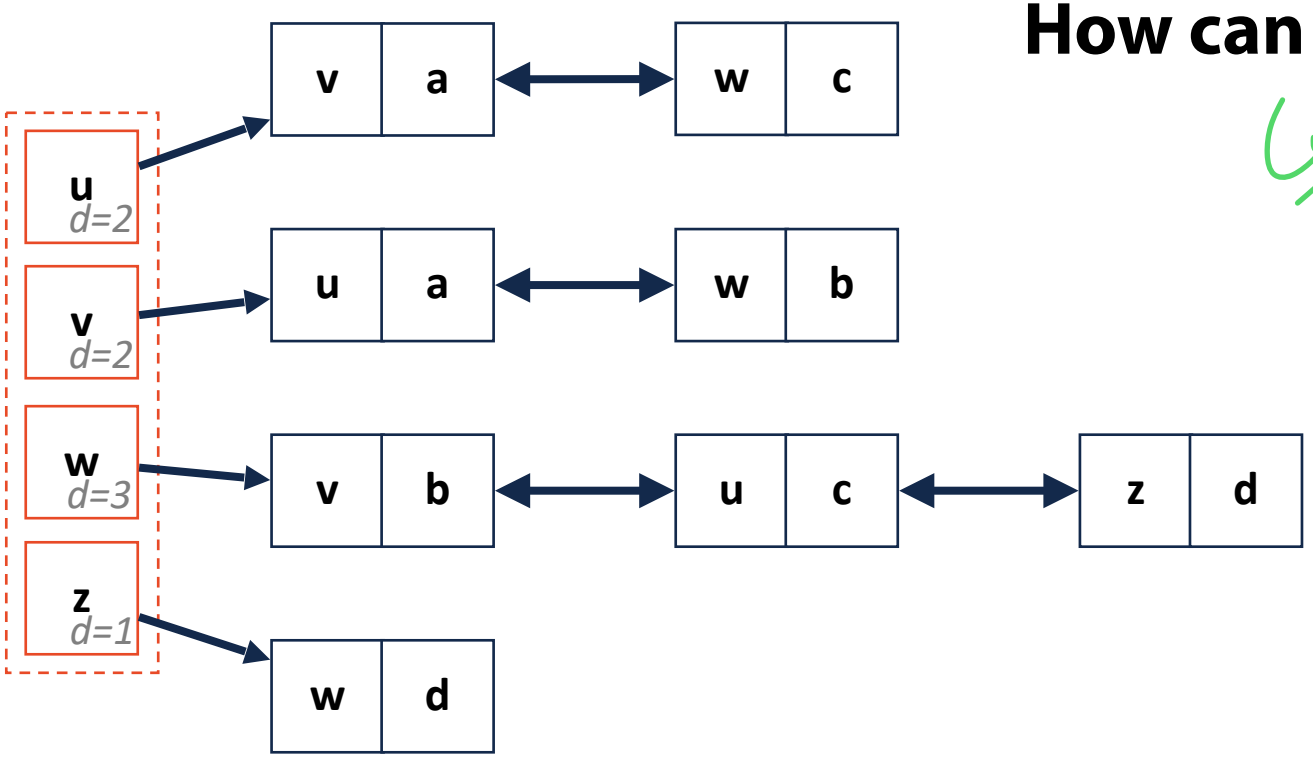
… And then look for V in every other list

… uh oh.

Naive Adjacency List

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





What's wrong with our implementation?

↳ Edges are often stored twice

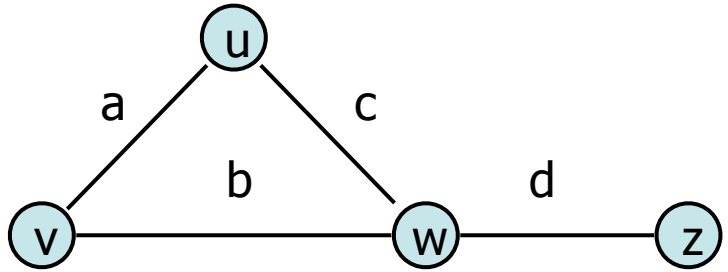
↳ Removing an edge or vertex requires looking at many LLs

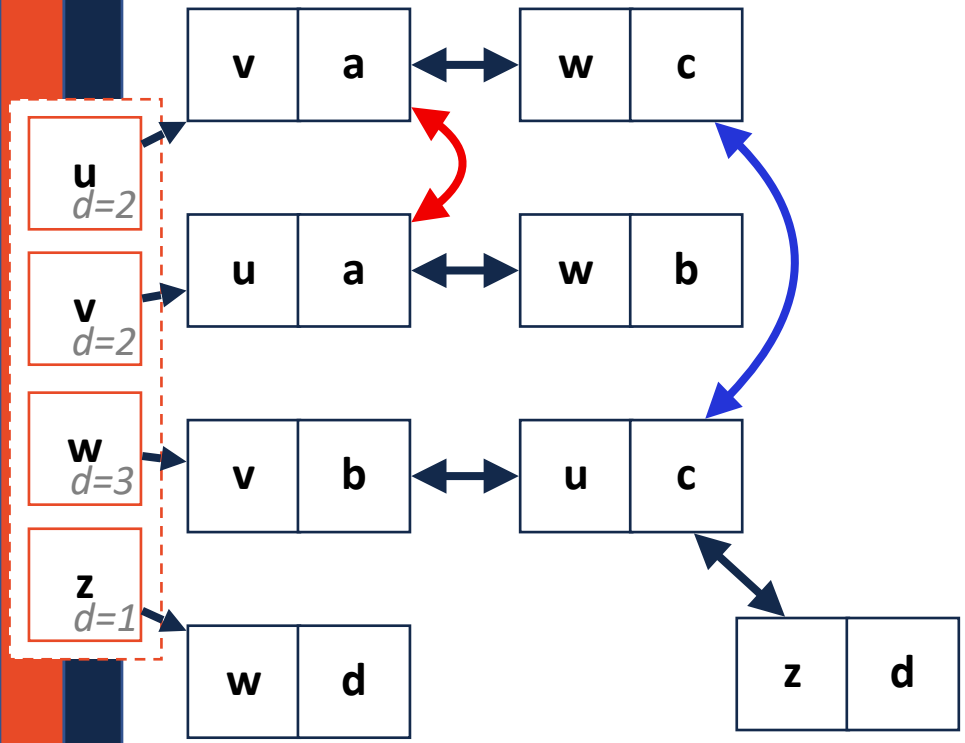
How can we fix it?

↳ Add more linked lists :)

Adjacency List

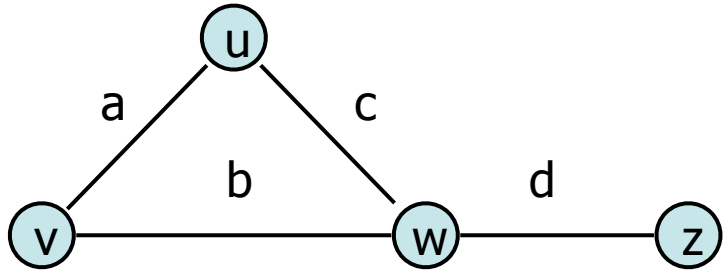
|V|= n, |E|= m

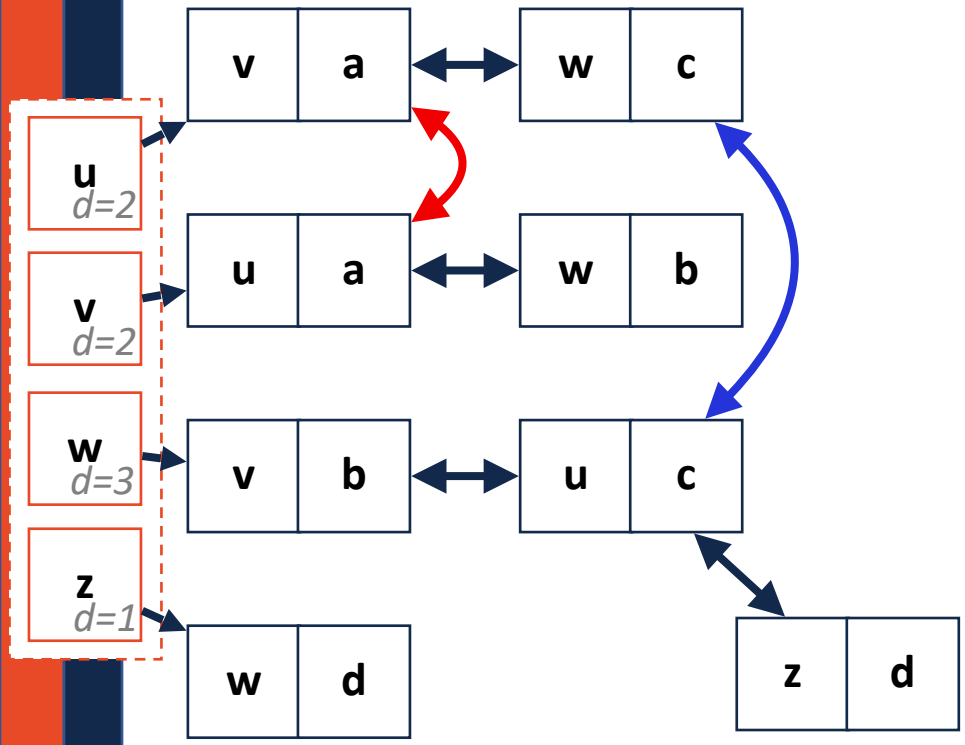


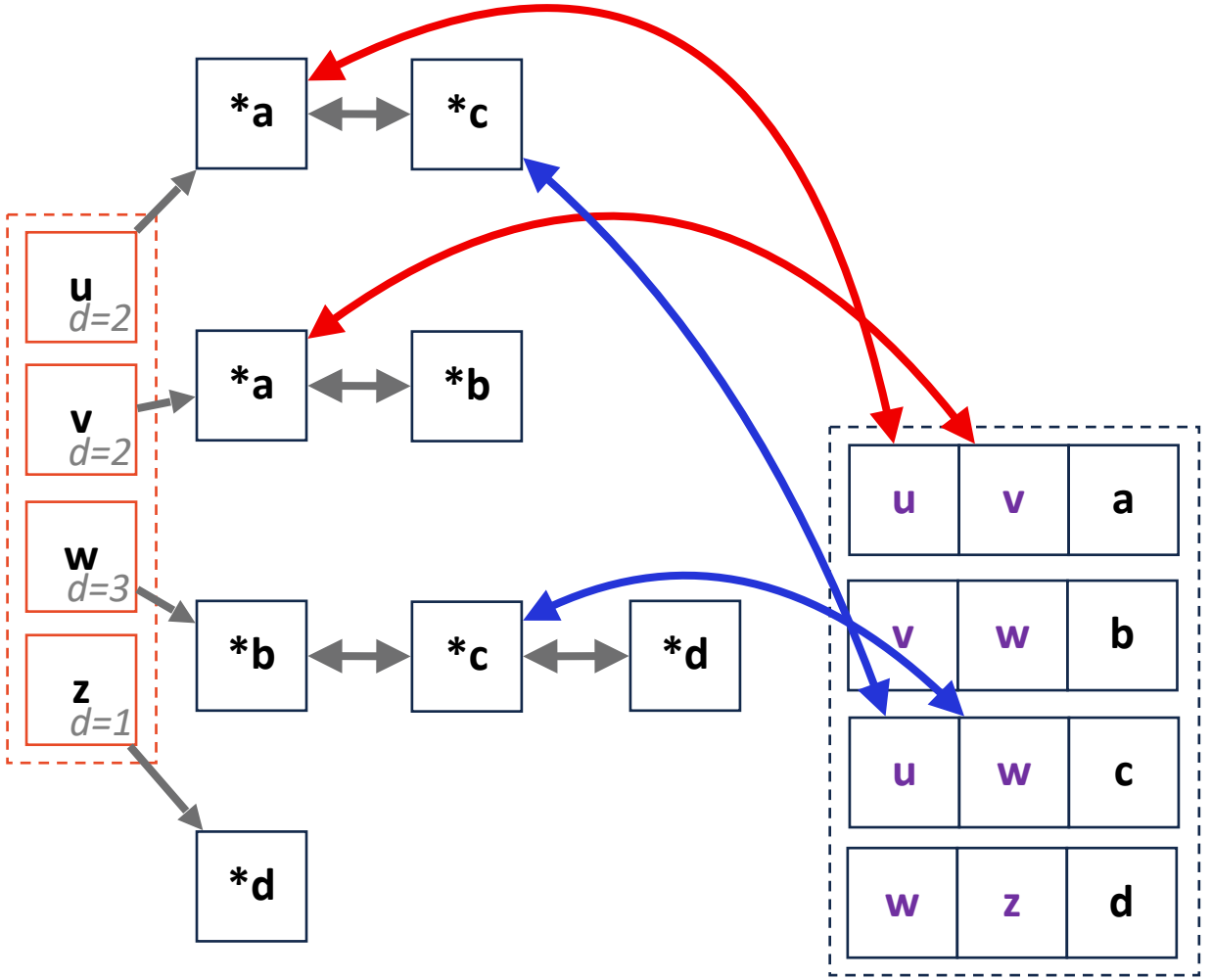


Adjacency List

|V|= n, |E|= m







Adjacency List

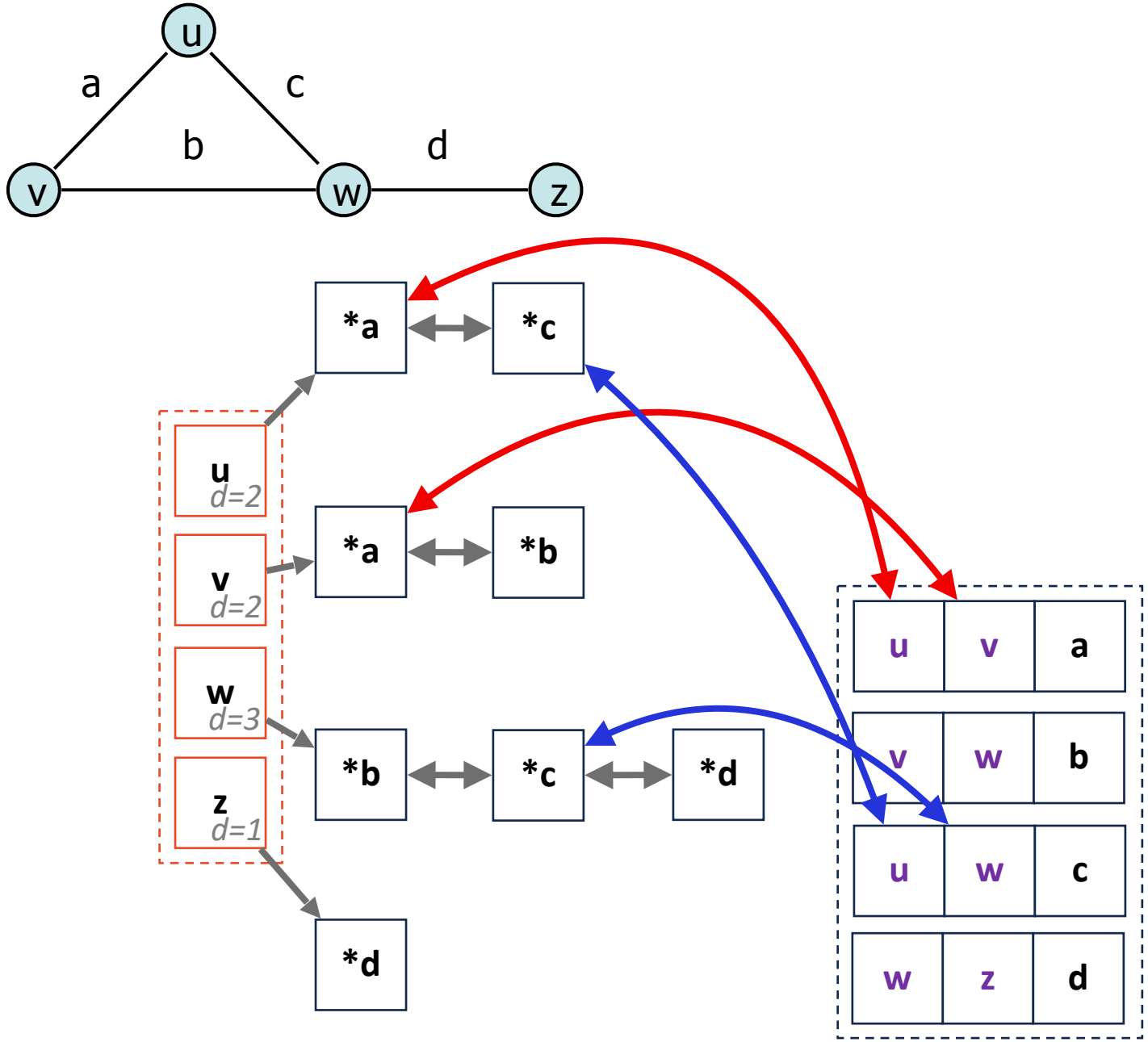
|V|= n, |E|= m

Adj List Node:

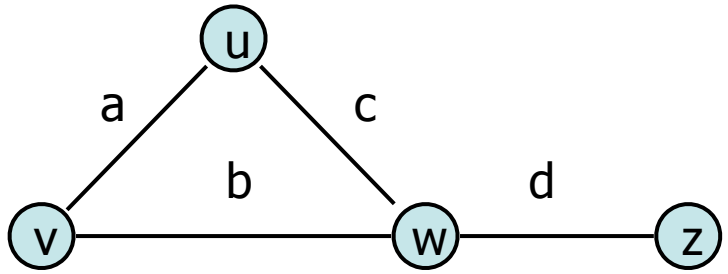
*Prev	*Edge	*Next

Edge List:

V1	V2	Weight
*V1	*V2	



Adjacency List

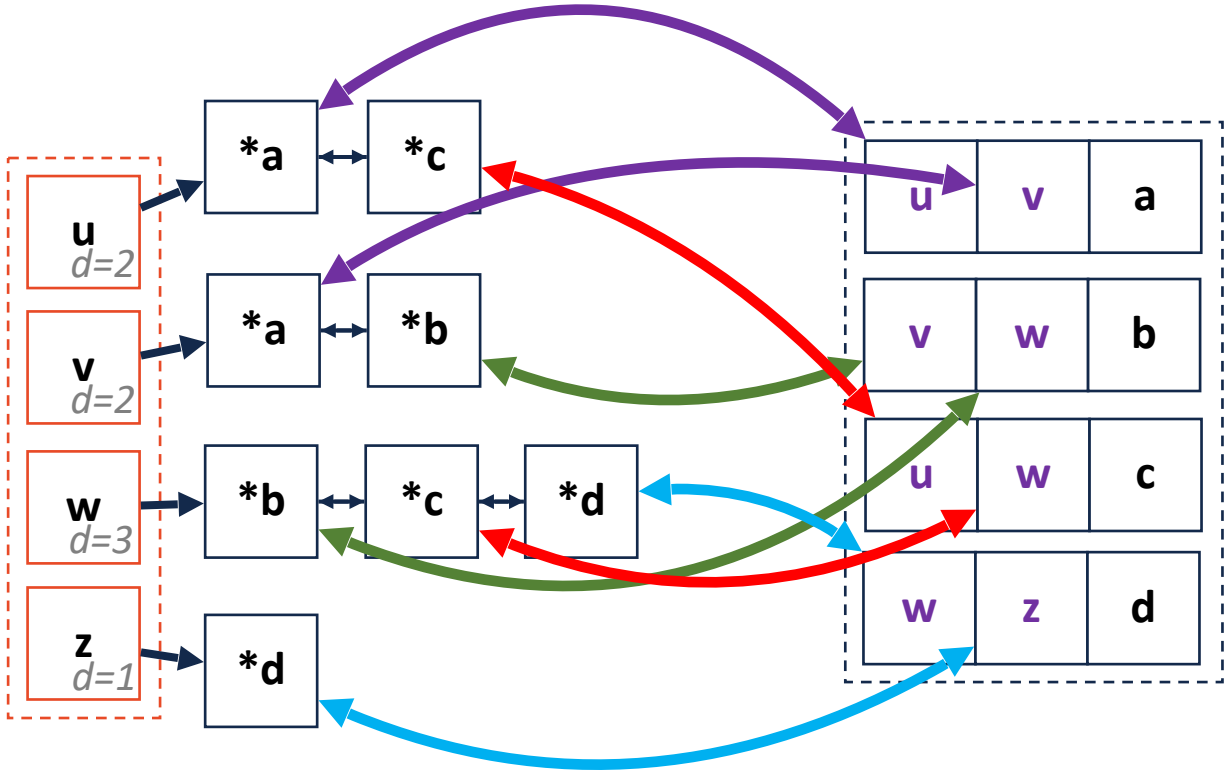


Vertex Storage:

Vertices stored in map w/ linked list values

A bidirectional linked list with size variable

Each node is a pointer to edge in edge list

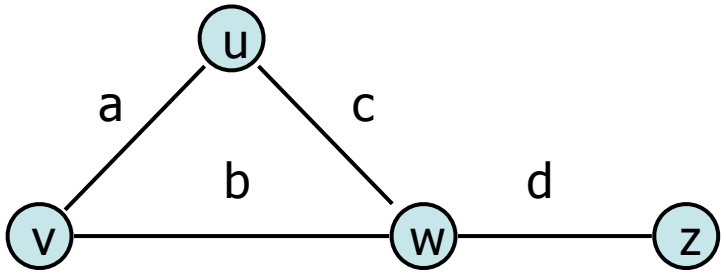


Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

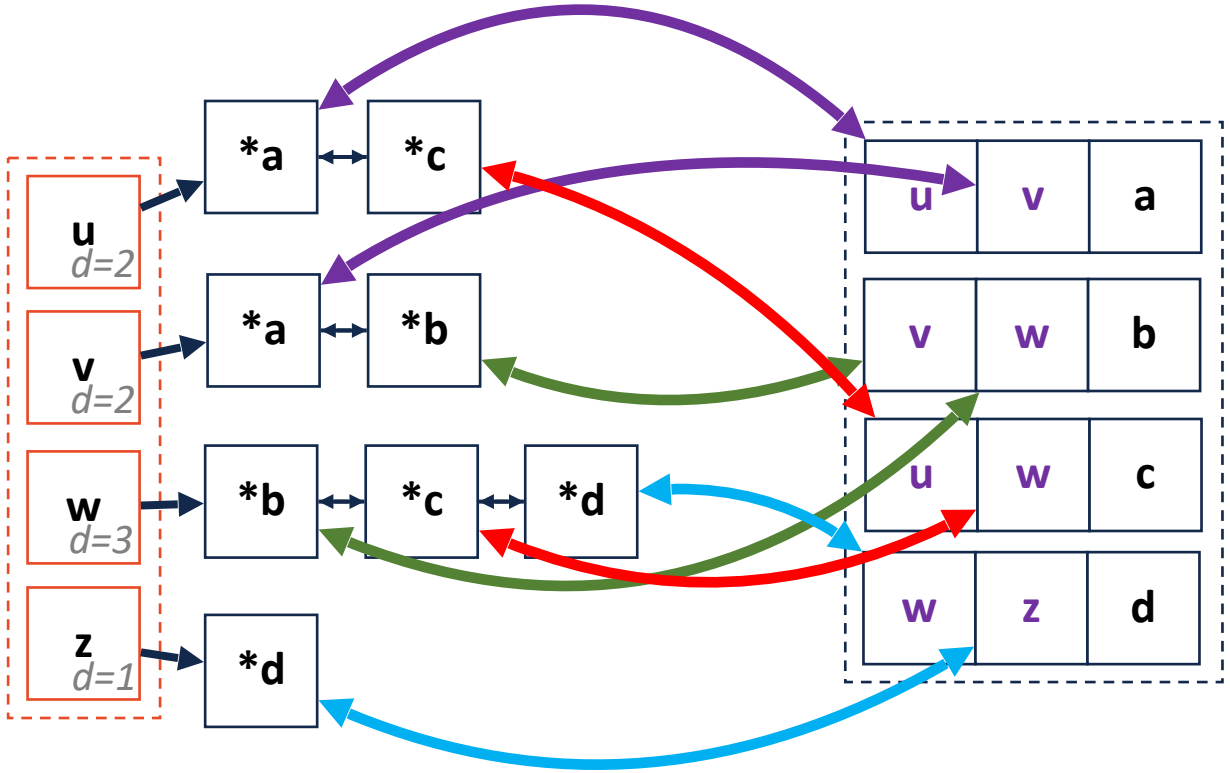
Adjacency List



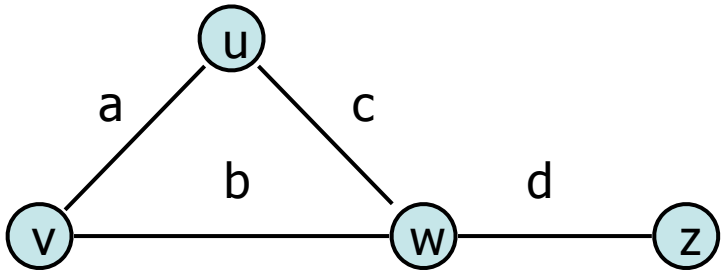
getEdges(Vertex v):

Look up vertex list (and walk across it)

[Each edge is a pointer lookup away]



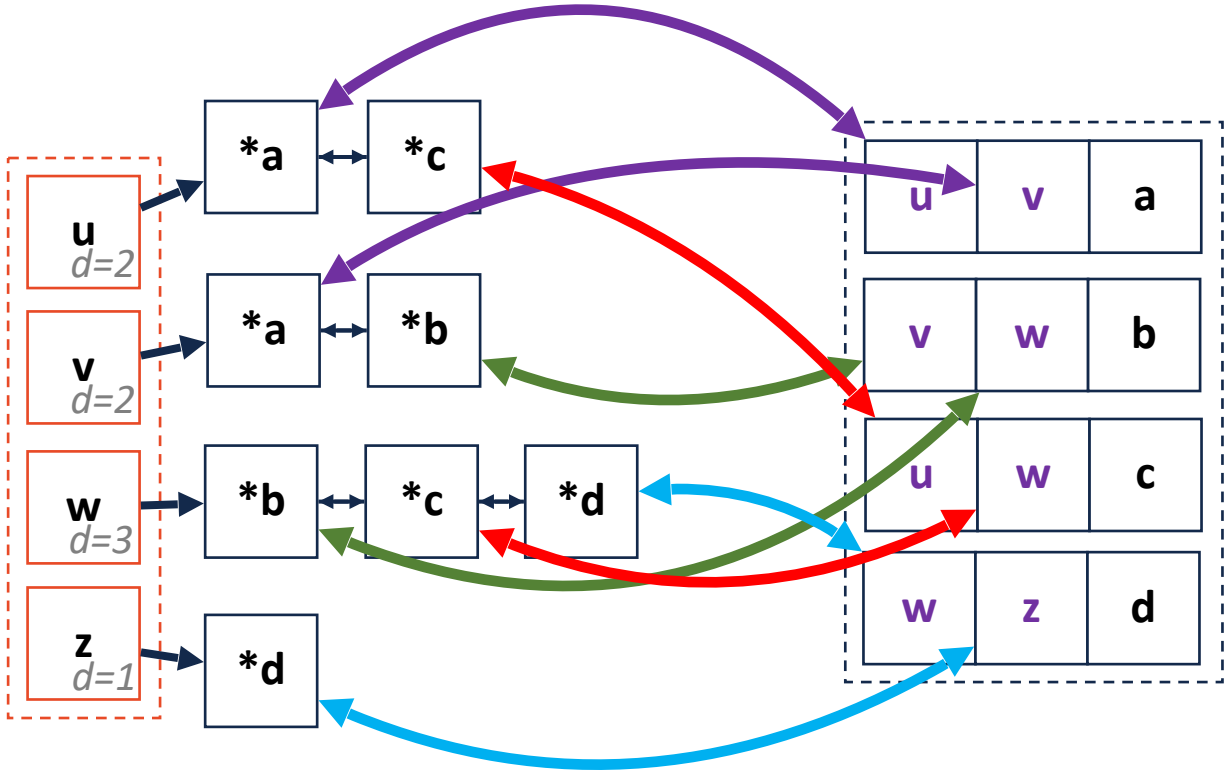
Adjacency List



areAdjacent(Vertex v1, Vertex v2):

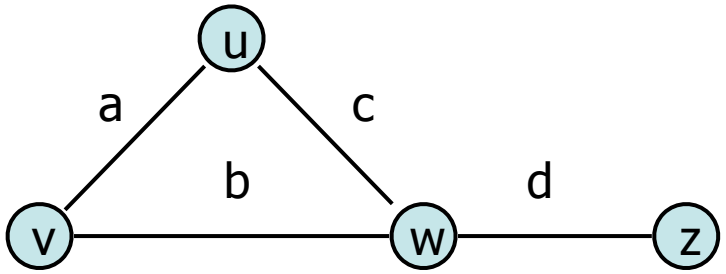
Look up min-degree vertex list

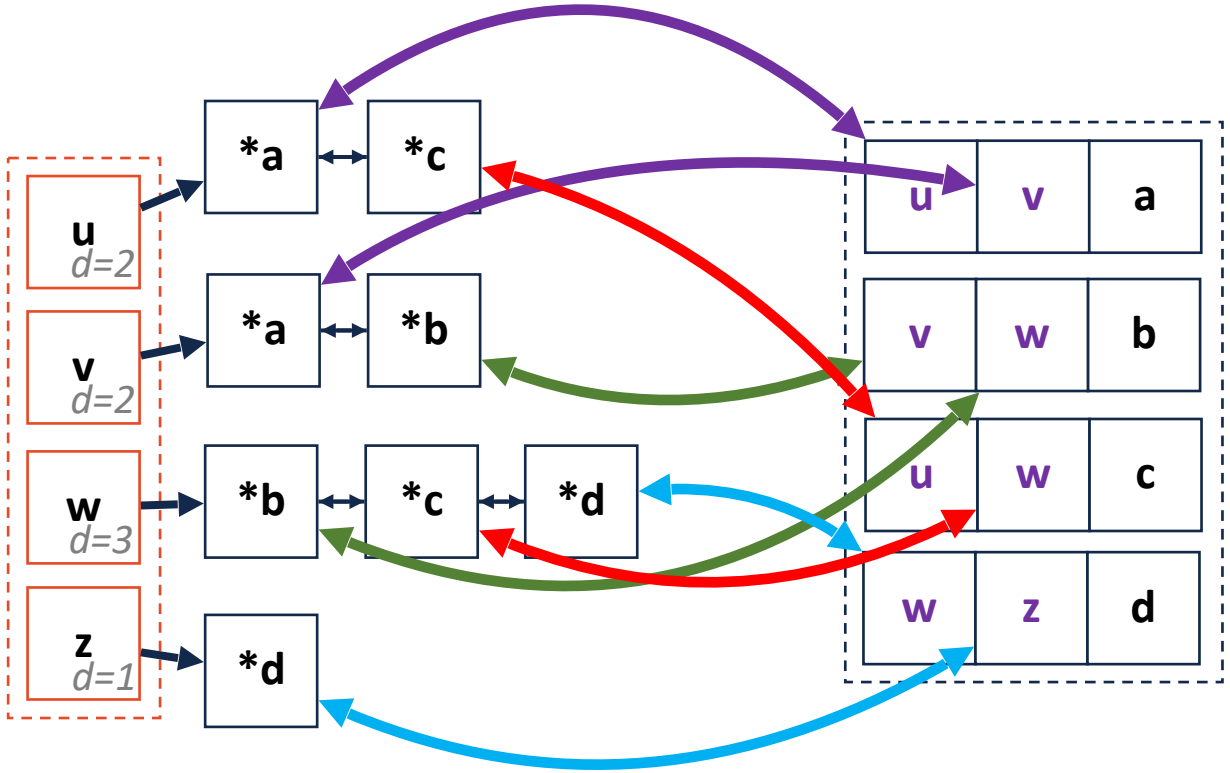
Search for other vertex across list



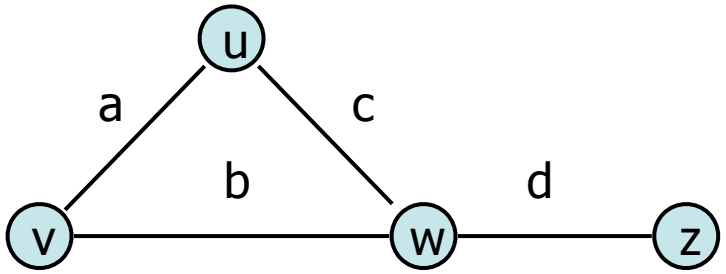
Adjacency List

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





Adjacency List

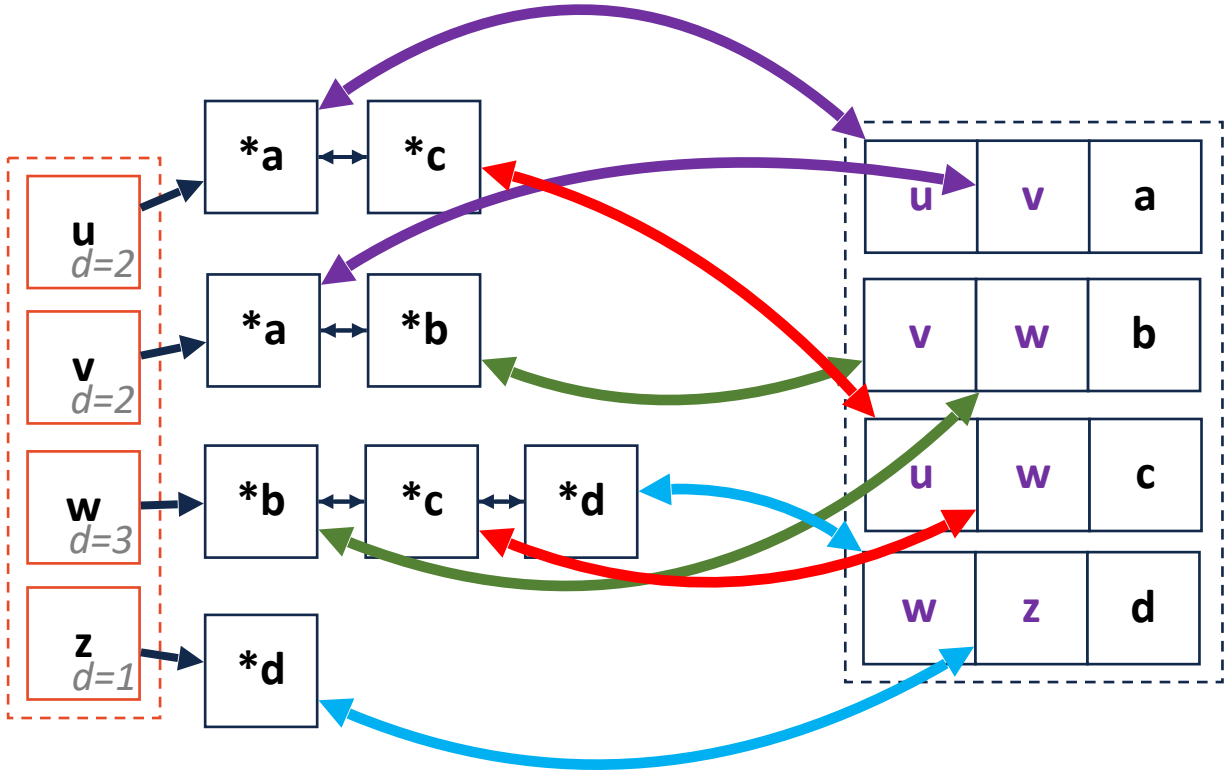


insertEdge(Vertex v1, Vertex v2, K key):

Add edge to edge list

Add node to each vertex list

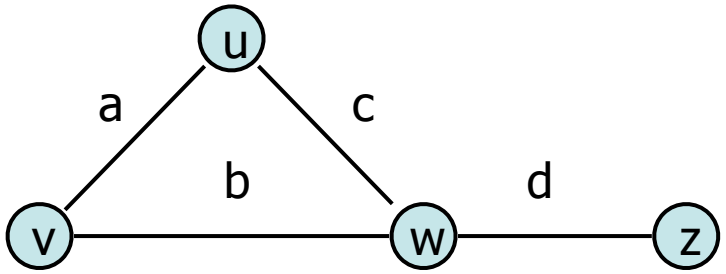
Connect all three with pointers

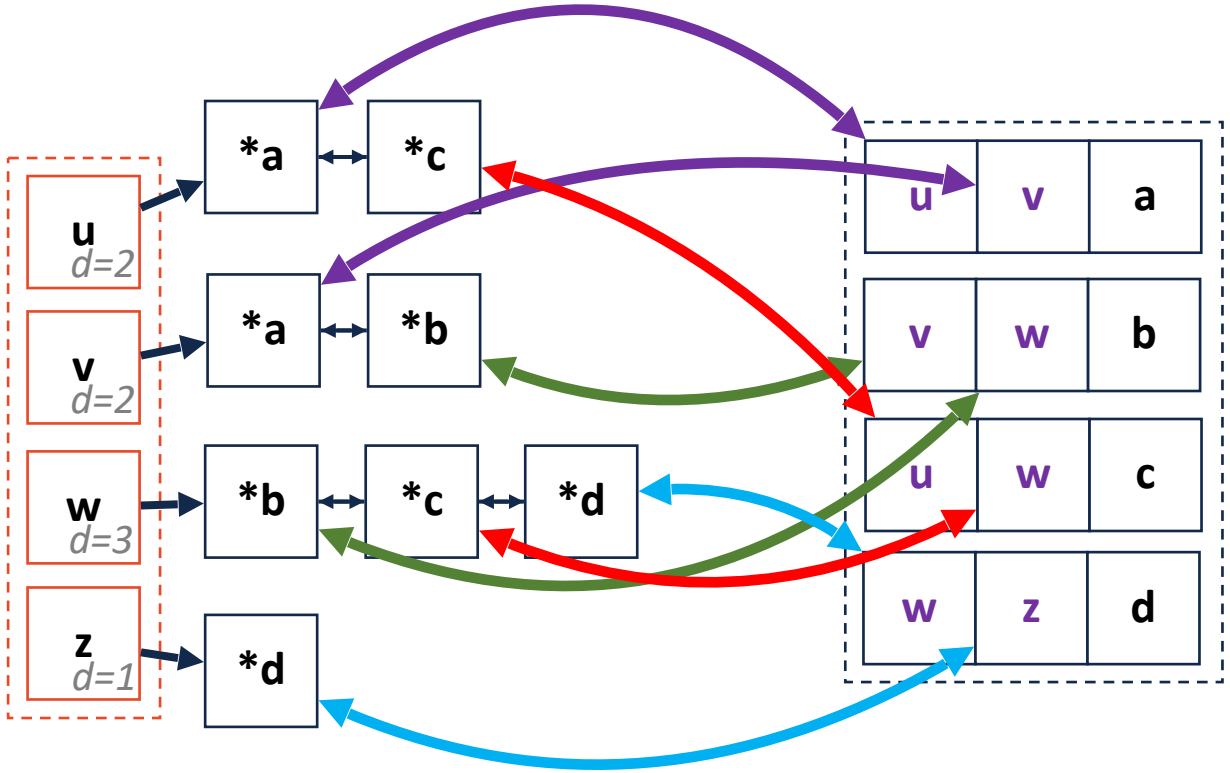


Join Code: 225

What are my Big Os?

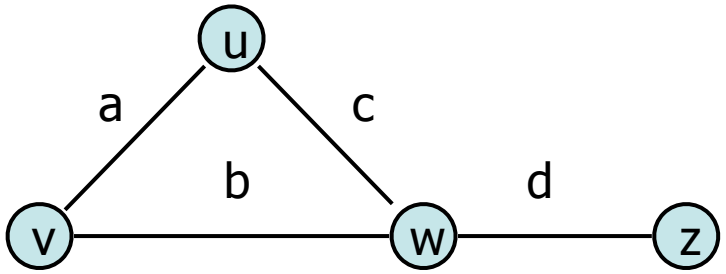
Adjacency List





removeEdge(Vertex v1, Vertex v2):

Adjacency List



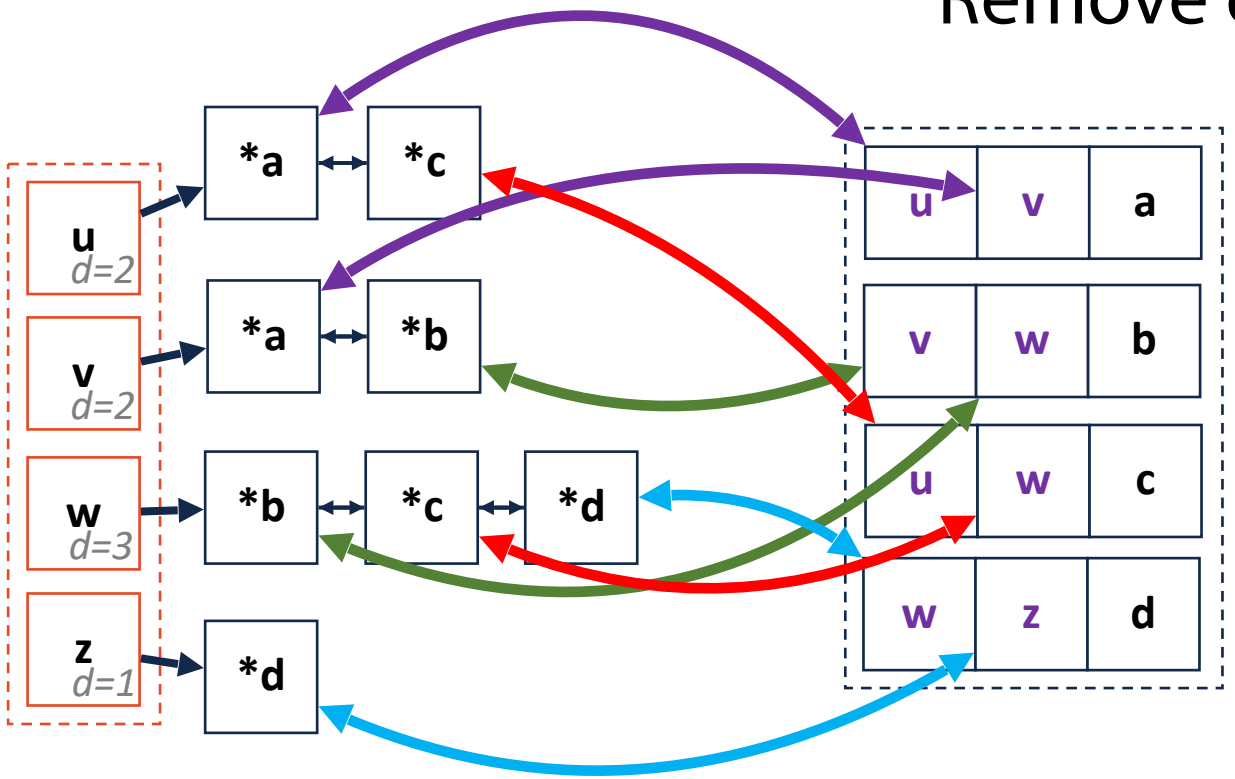
removeEdge(Vertex v1, Vertex v2):

Search min-degree vertex list

Remove mirrored entry using pointers

Remove edge from edge list

Remove entry from vertex list

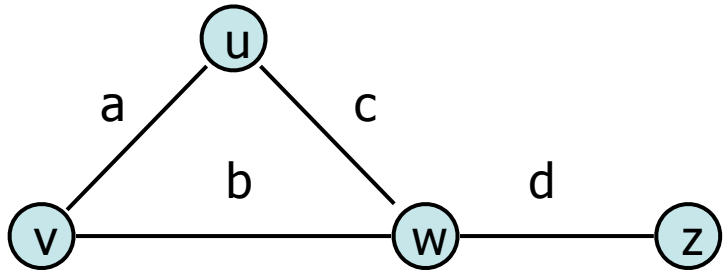


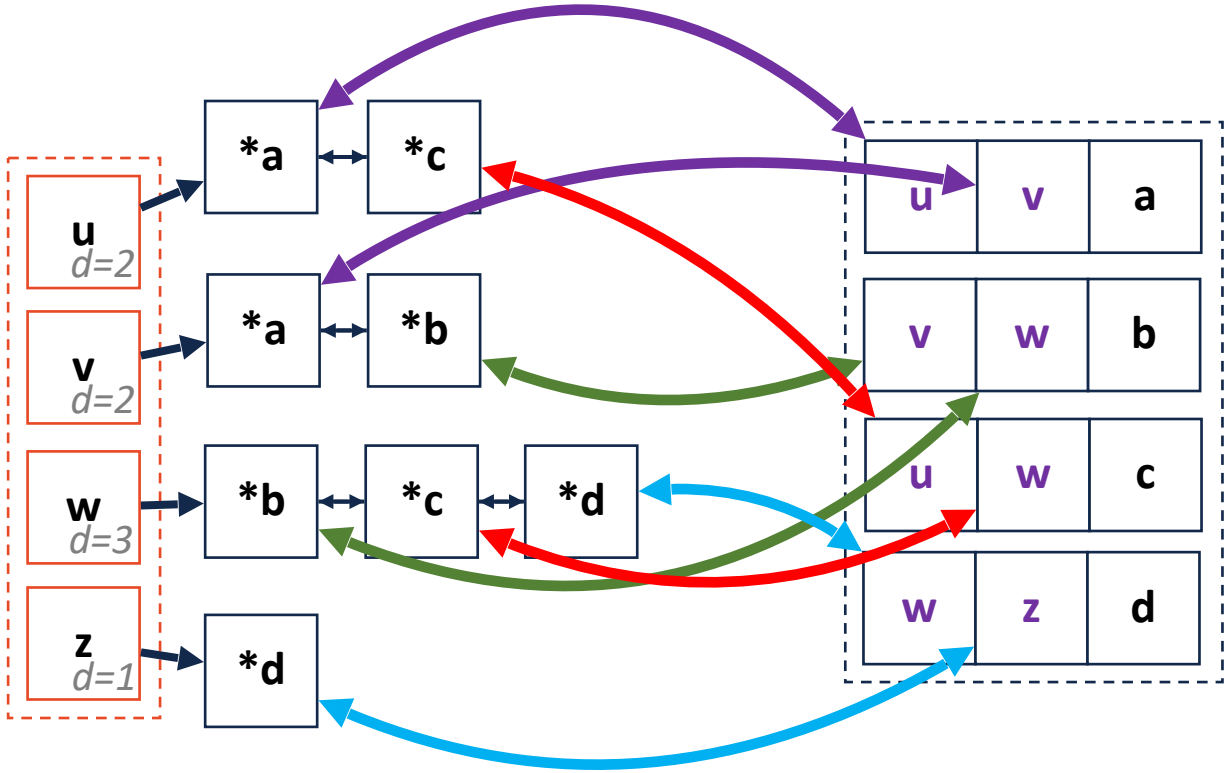
Join Code: 225

What are my Big Os?

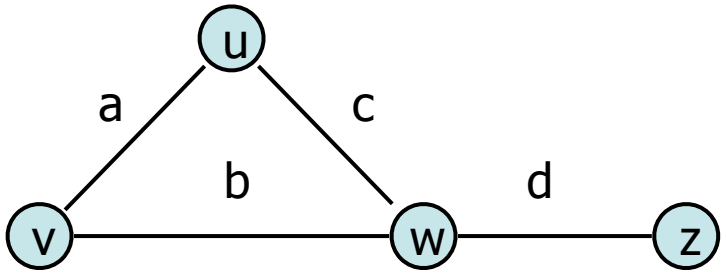
Adjacency List

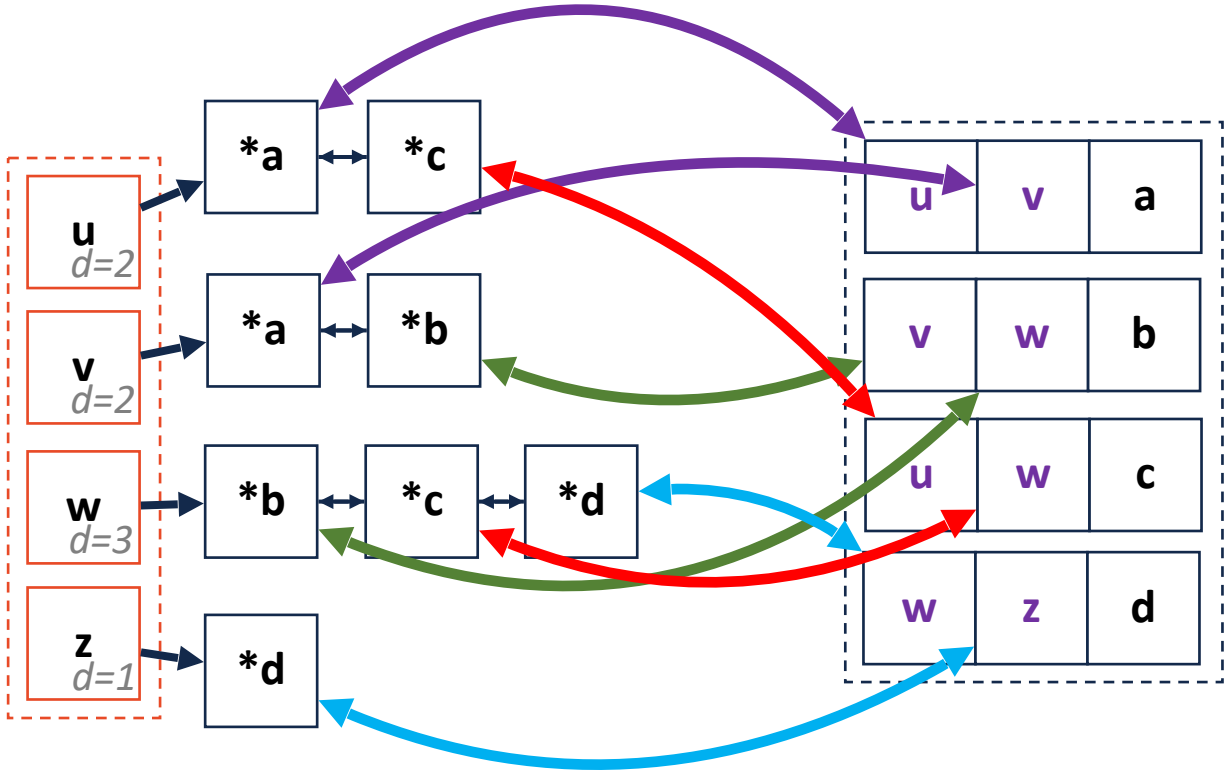
insertVertex(K key):





Adjacency List

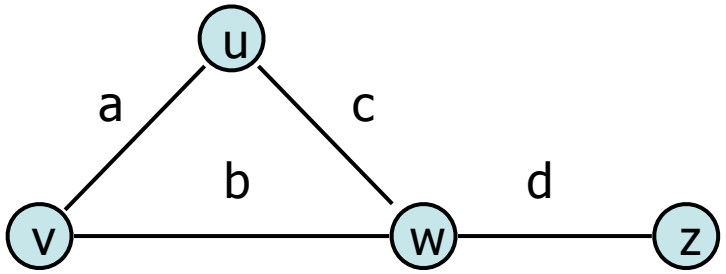


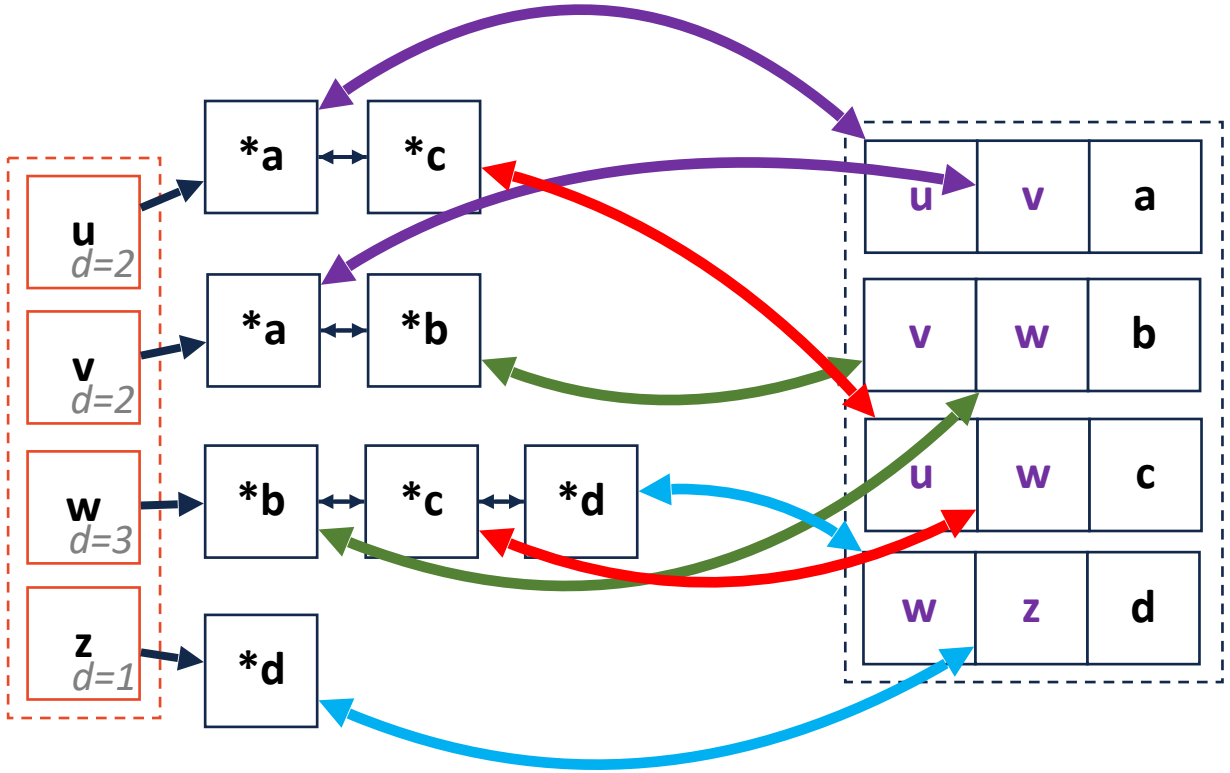


insertVertex(K key):

Add new empty list to vertex list.

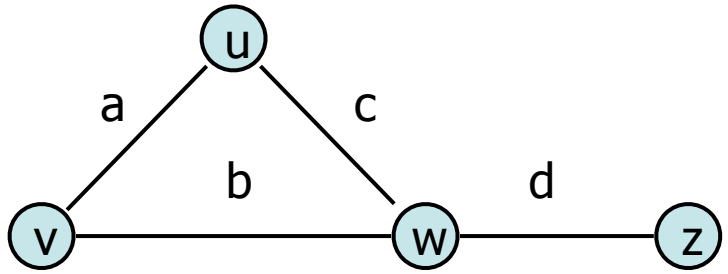
Adjacency List

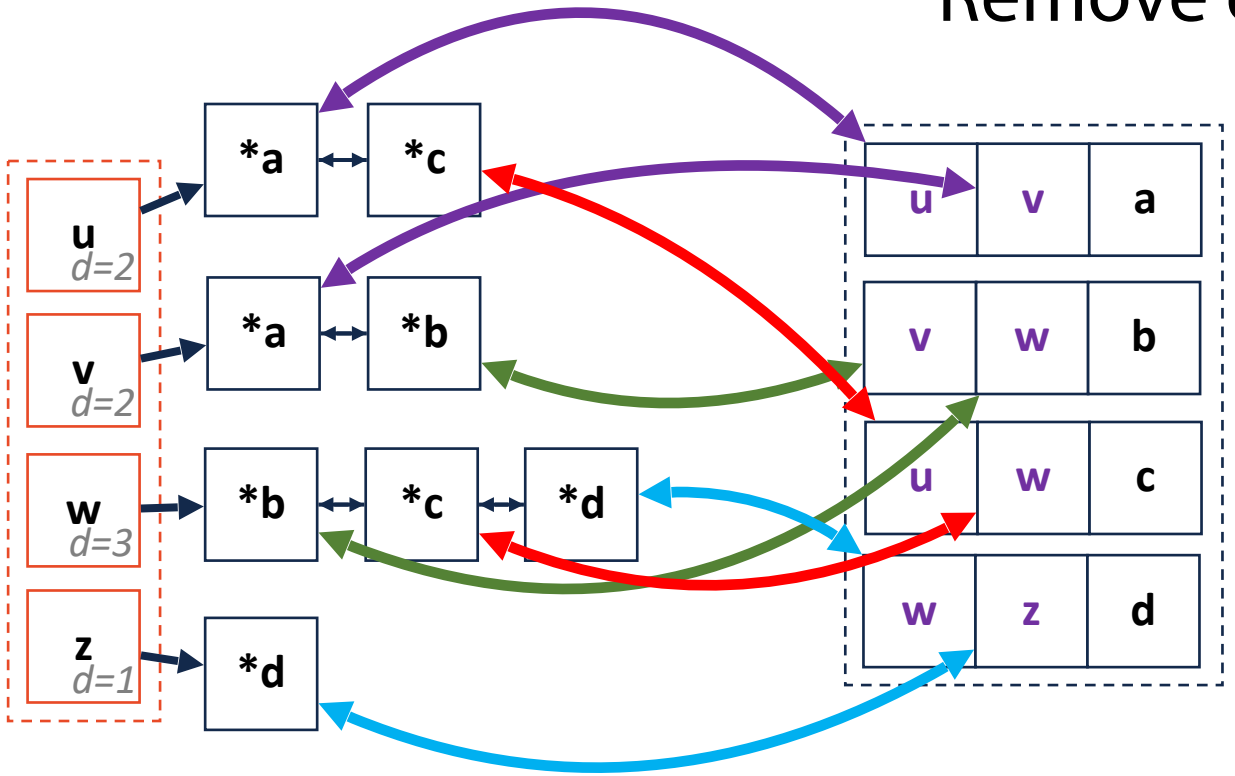




removeVertex(Vertex v):

Adjacency List





removeVertex(Vertex v):

Look up vertex in vertex list

Walk across list

Remove mirrored entry using pointers

Remove edge from edge list

Remove entry from vertex list

Join Code: 225

What are my Big Os?

|V|= n, |E|= m



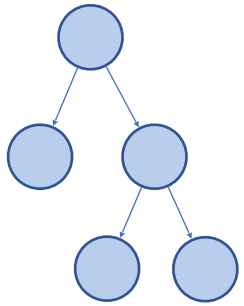
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n* / n^2	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

Graph Traversals

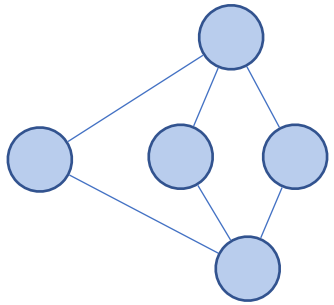
There is no clear order in a graph (even less than a tree!)

How can we systematically go through a complex graph in the fewest steps?

Tree traversals won't work — lets compare:

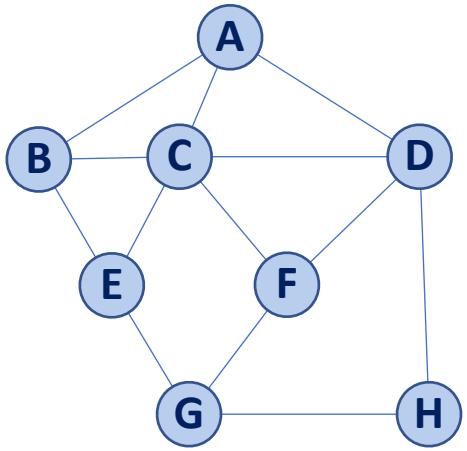


- Rooted
- Acyclic
-

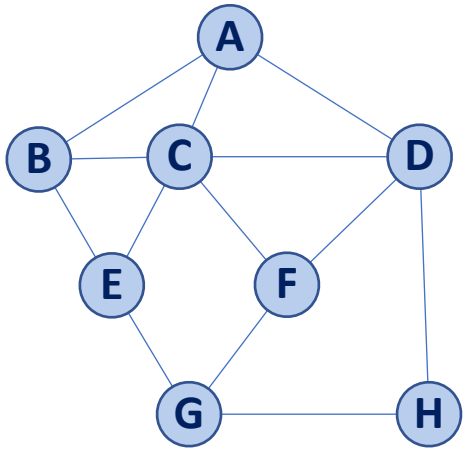


-
-
-

Traversal: BFS



Traversal: BFS



v	d	P	Adjacent Edges
A			B C D
B			A C E
C			A B D E F
D			A C F H
E			B C G
F			C D G
G			E F H
H			D G