Data Structures

Graph Implementations 2

CS 225

Brad Solomon

April 1, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 4 on 4/6 — 4/8

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam out now on PL

Topics covered can be found on website

Register now!

https://courses.engr.illinois.edu/cs225/exams/

Lab this week is exam 4 review session

Like exam 2, many new questions will be released for lab

A great opportunity to go over practice exam too!

Form a study group to peer grade FRQs

Learning Objectives

On Exam

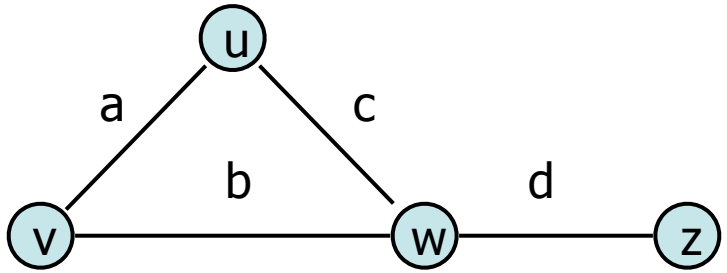
Discuss graph implementation and storage strategies

Introduce graph traversals ???

not on exam

Graph Implementation: Edge List

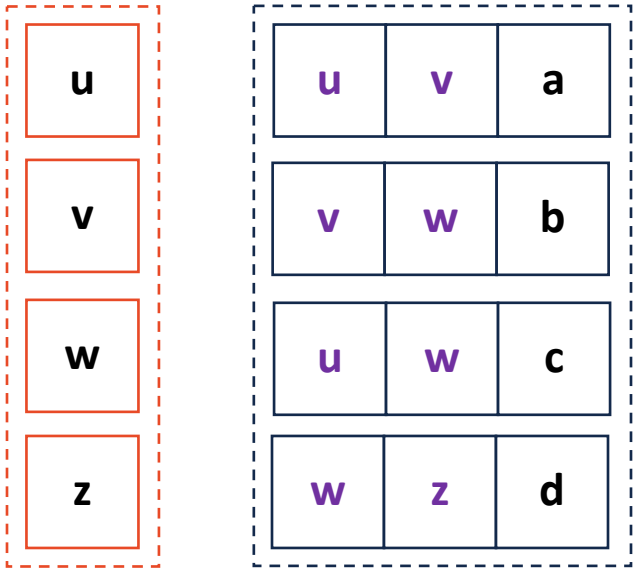
$|V| = n, |E| = m$



O(1)*

insertVertex(K key):

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



O(m)

removeVertex(Vertex v):

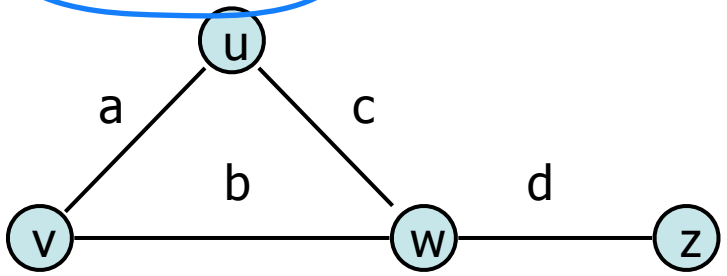
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

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



u	0
v	1
w	2
z	3

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

Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

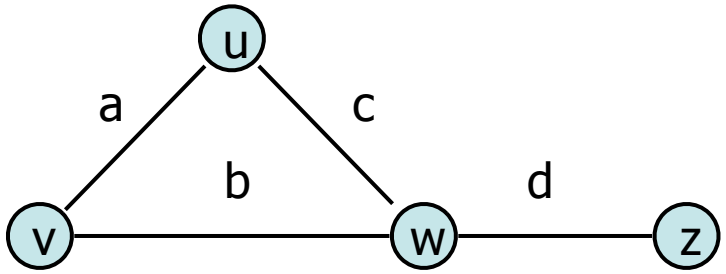
Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



removeVertex(Vertex v):

Remove v from vertex mapping

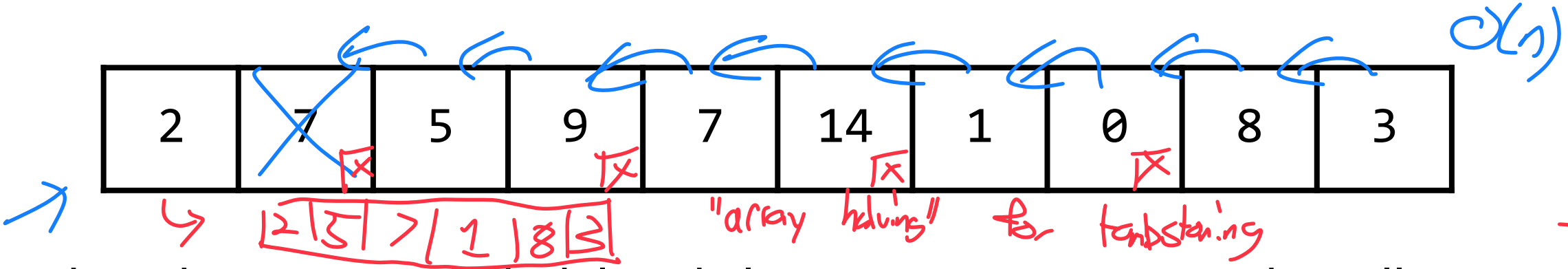
Remove an entire row / entire column

u	0
v	1
w	2
z	3

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

Amortized Removal with Tombstoning

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



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				-

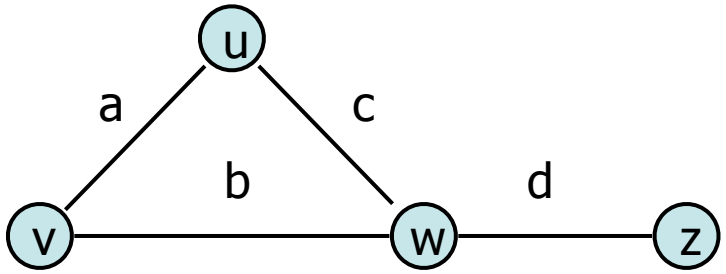
Adjacency Matrix:

removeVertex() by tombstoning |V| values

Resize when needed or by request

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



O(1)
insertEdge(Vertex v1, Vertex v2, K key):
removeEdge(Vertex v1, Vertex v2, K key):
areAdjacent(Vertex v1, Vertex v2):

O(n)
incidentEdges(Vertex v):

Get all edges

O(n)—O(n²)
insertVertex(K key):
removeVertex(Vertex v):

Sparse dense

$n-1 \leq m \leq n^2$

Also space is always $O(n^2)$

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

u
v
w
z

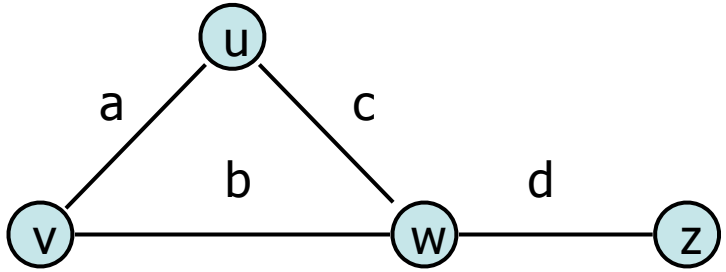
Graph Implementation Brainstorming

We want something…

Faster than an edge list

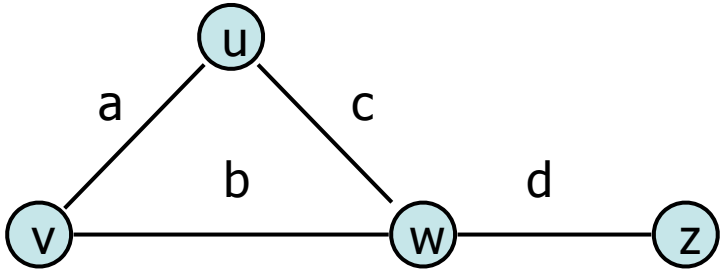
Less space than an adjacency matrix

Particularly good at **finding adjacent elements**

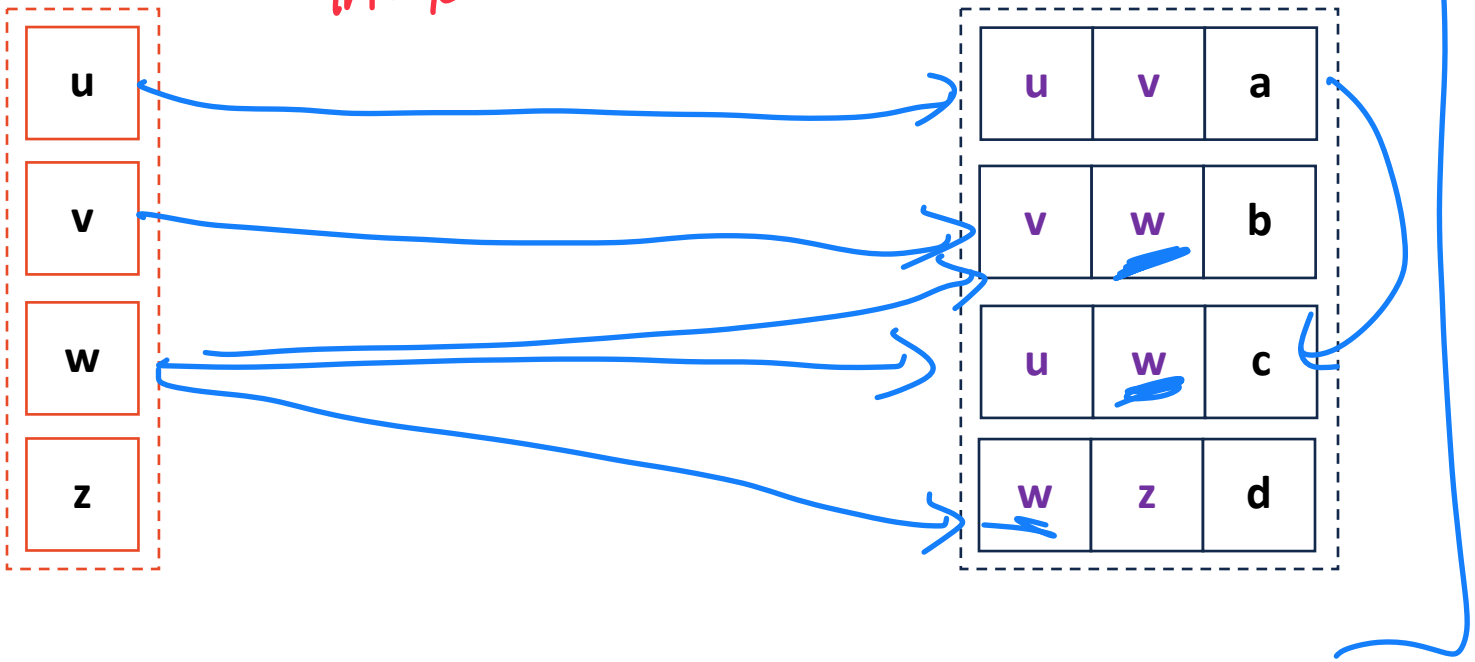


Graph Implementation: Edge List + ?

|V|= n, |E|= m



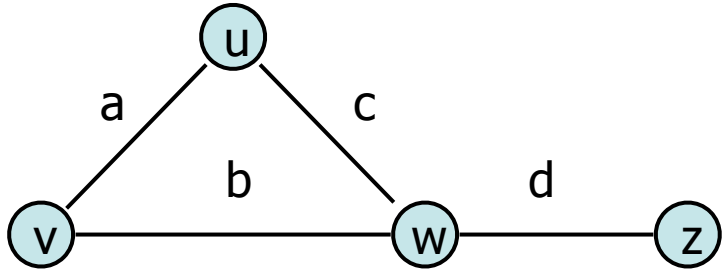
Maybe pointers can fix this?



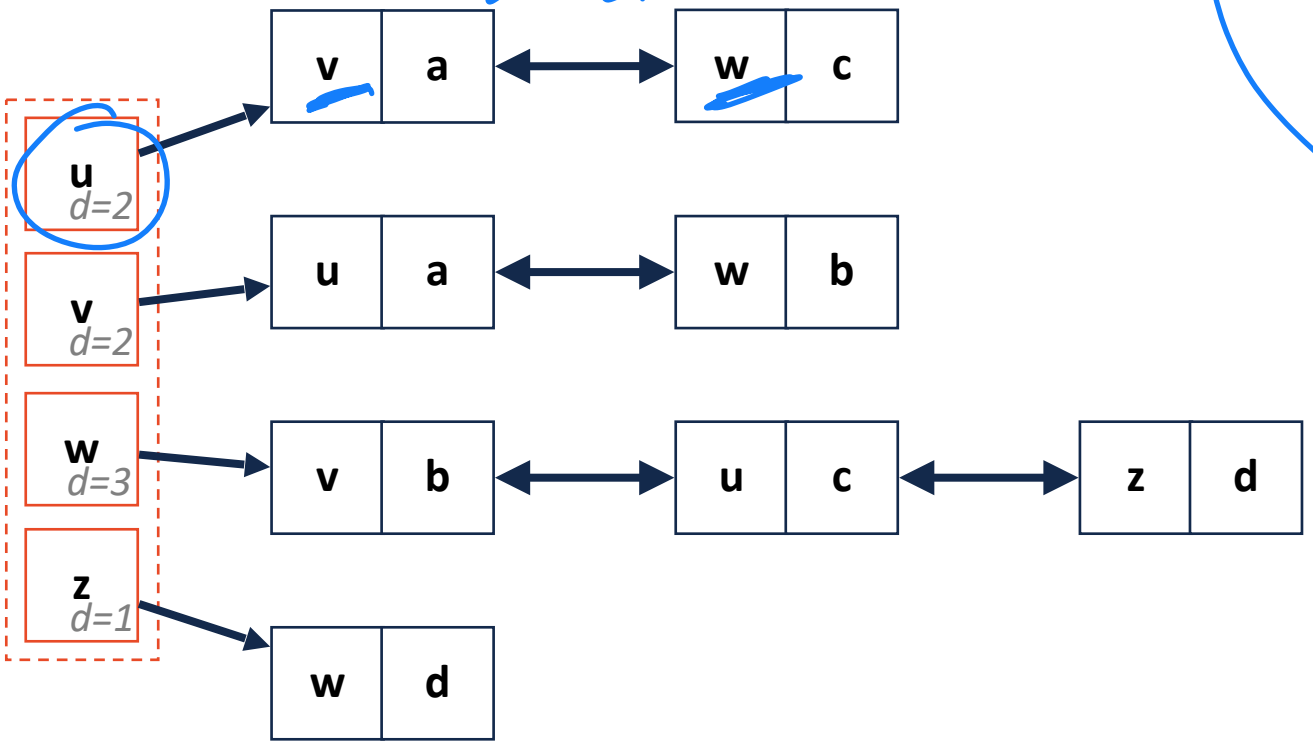
arrays are nice!

Naive Adjacency List

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



Edge storage



Vertex storage structure

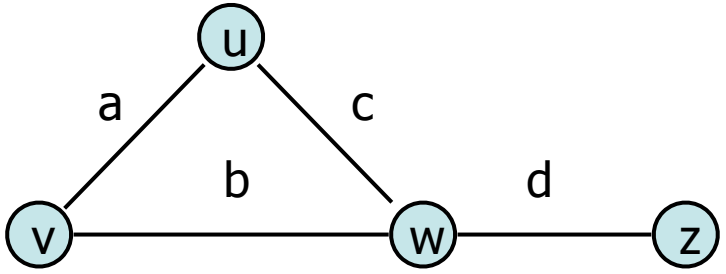
↳ Map keys as vertices values as a linked list

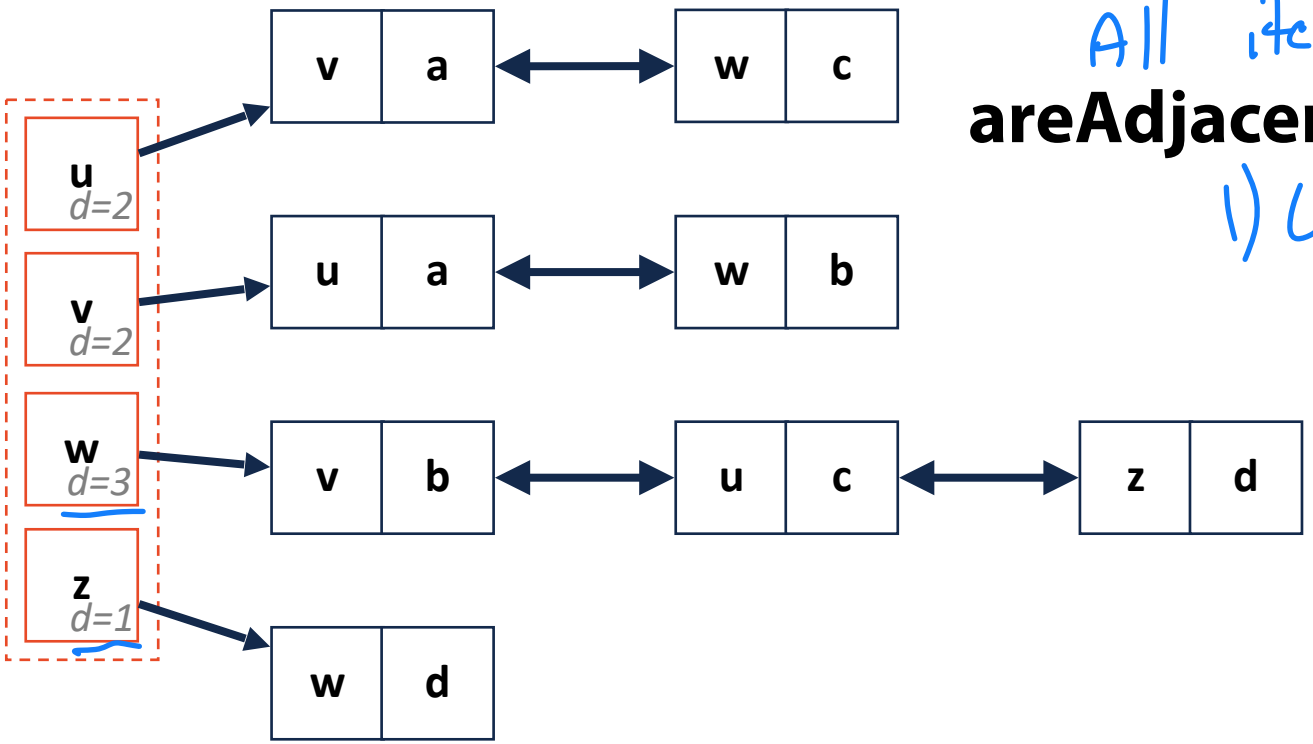
Edge stores all adjacent values as LL

↳ All V2 endpoints for V1 defined as key

Naive Adjacency List

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





incidentEdges(Vertex v):

1) Look up v in dictionary O(1)***

2) Walk along linked list

All items in list are incident

areAdjacent(Vertex v1, Vertex v2):

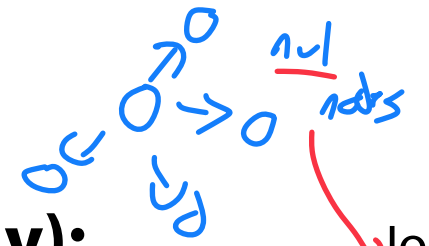
1) Look up either v_1 or v_2

↳ Pick LL w/ smaller size!

↳ The smaller degree is faster

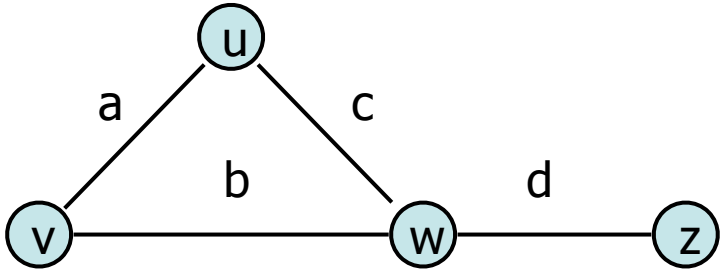
Naive Adjacency List

simple connected graph



Join Code: 225

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



incidentEdges(Vertex v):

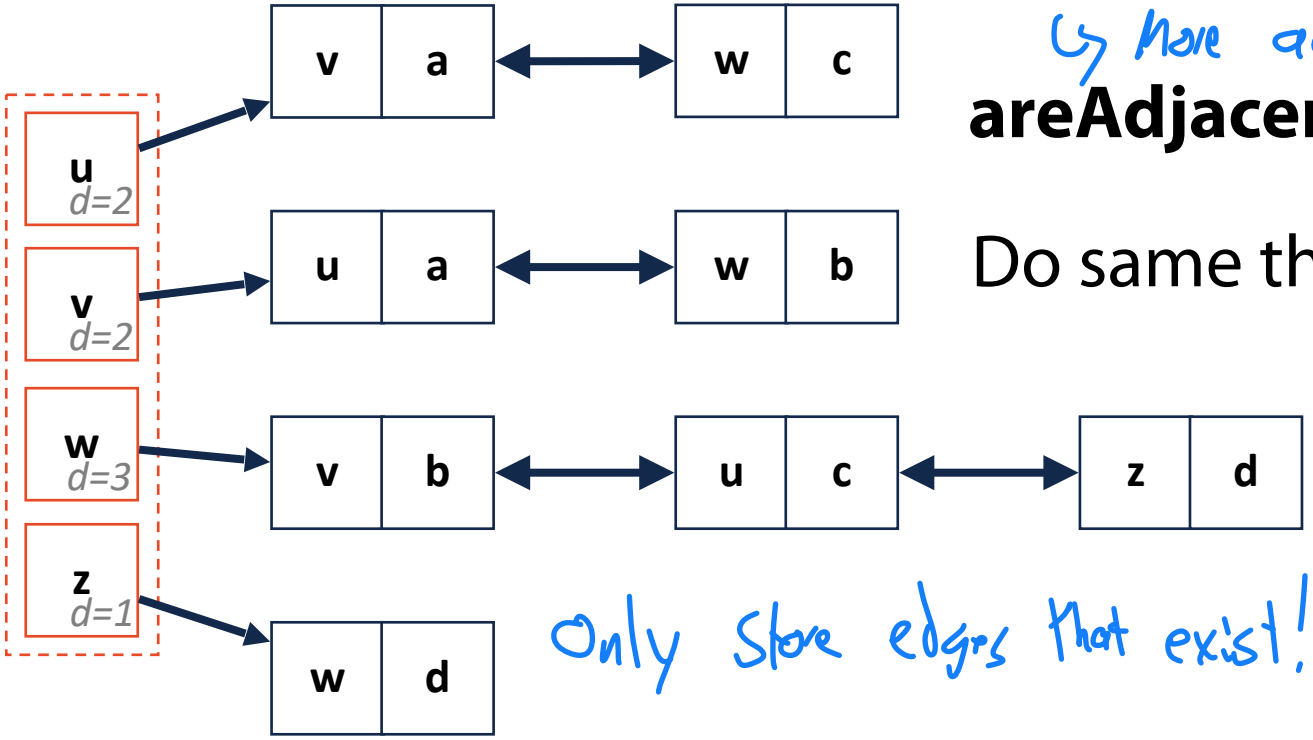
Lookup linked list for v

$O(\deg(v)) = O(n)$

Walk down list! Every edge is incident.

$O(n)$ $O(n)$ $O(n^2)$ $O(m^2)$

↳ More accurate Big O: $O(\deg(v))$



Only store edges that exist!

areAdjacent(Vertex v1, Vertex v2):

Do same thing but look for specific v2

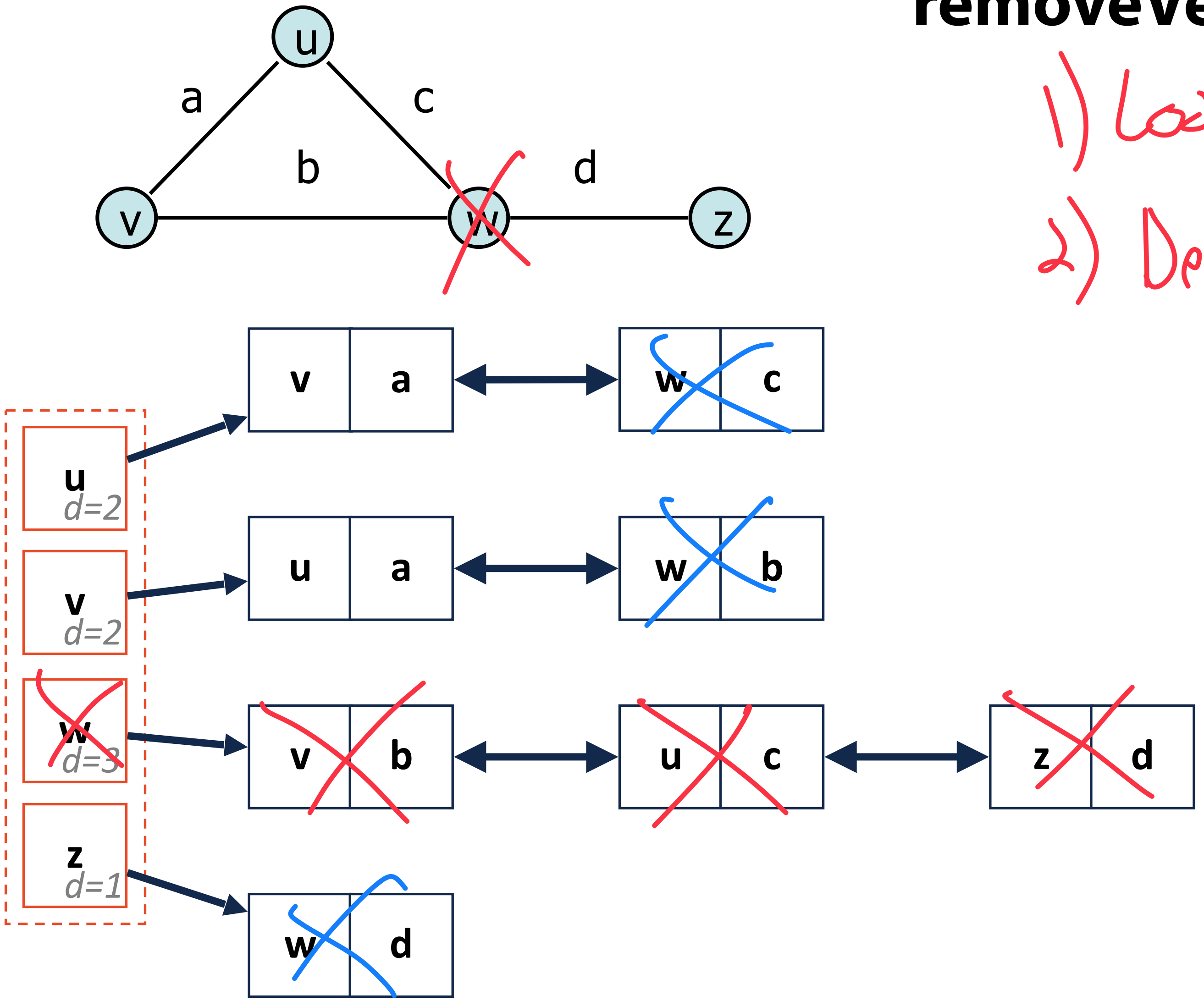
Both of the degrees

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

What are my Big Os?

Naive Adjacency List

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

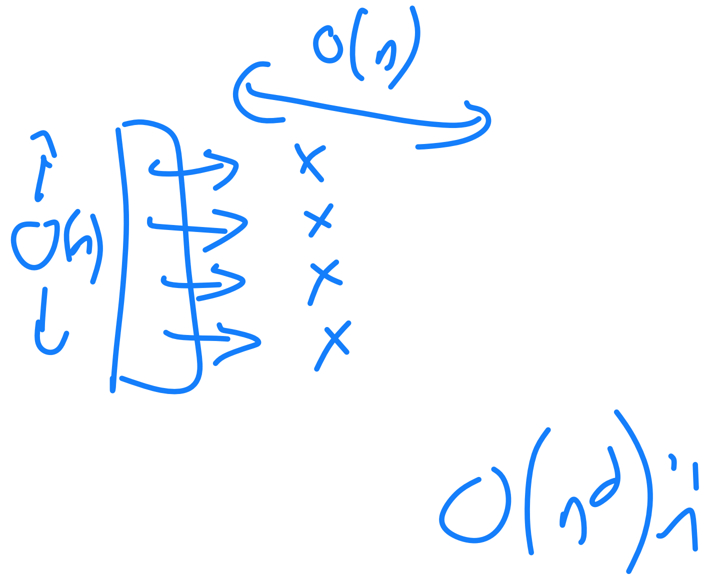


removeVertex(Vertex v):

1) Look up v in map

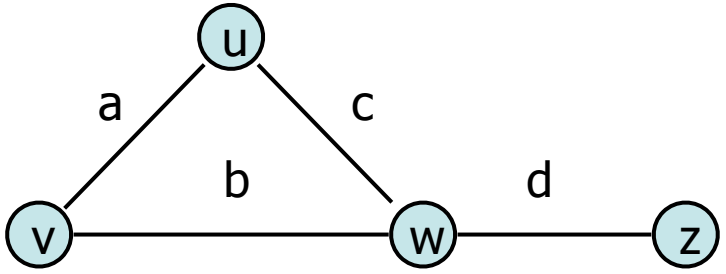
2) Delete LL value in map

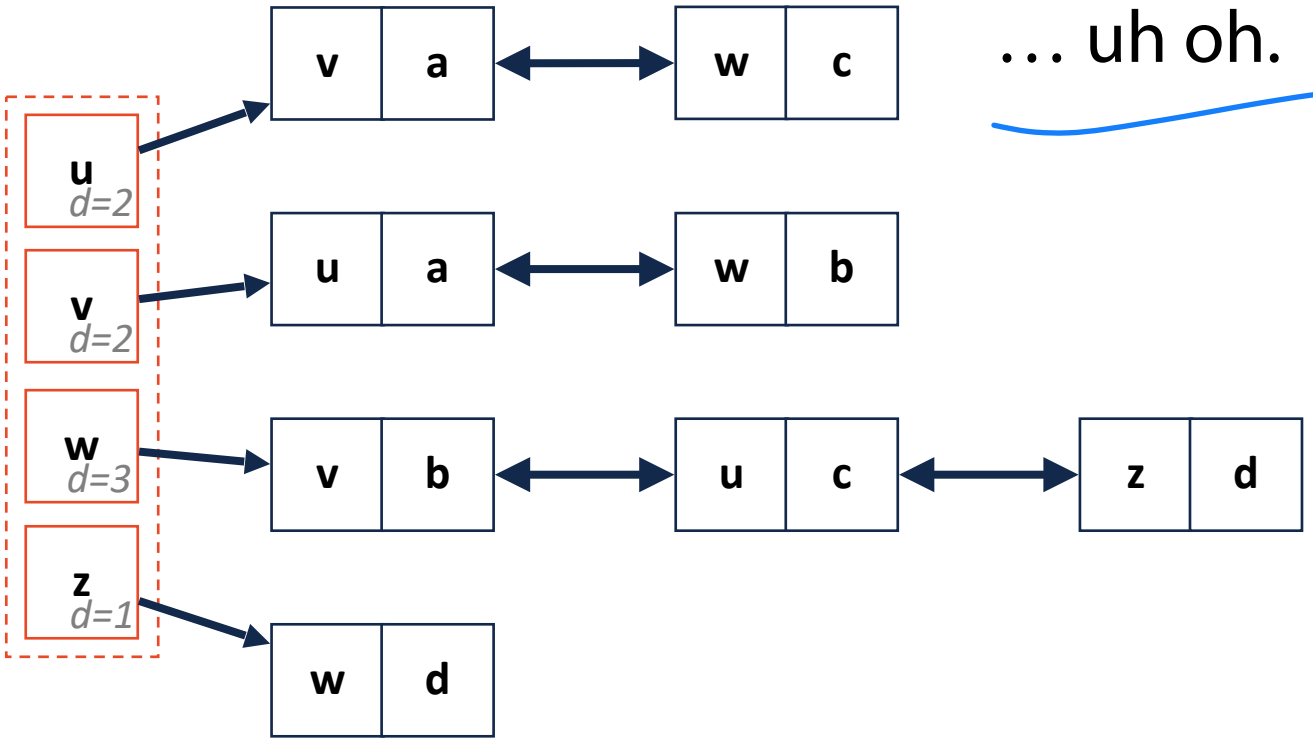
uh oh.....



Naive Adjacency List

|V|= n,|E|= m





removeVertex(Vertex v):

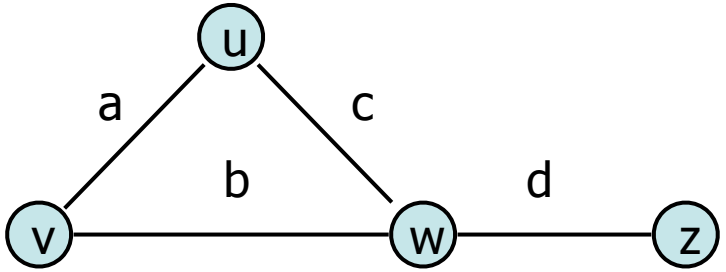
Look up V in our list and remove all entries.

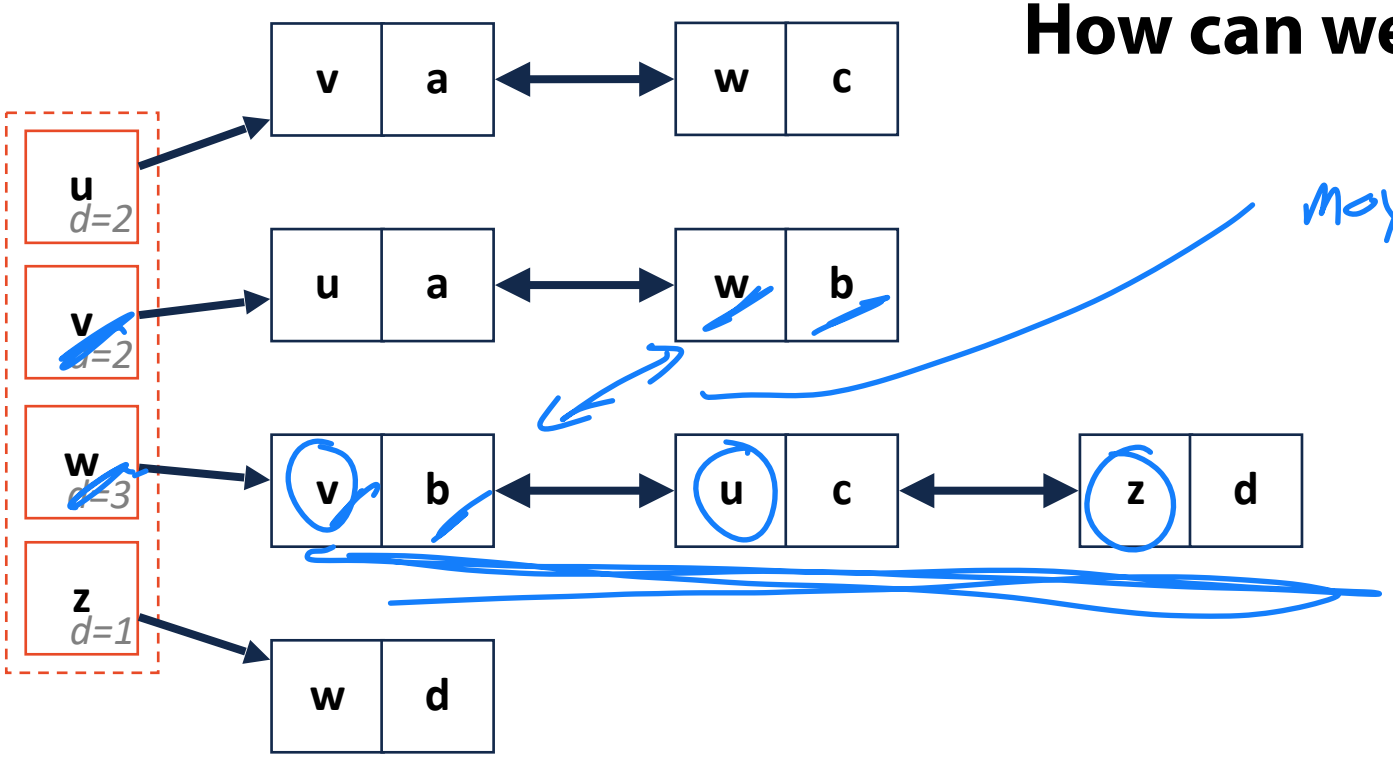
… And then look for V in every other list

… uh oh.

Naive Adjacency List

|V|= n, |E|= m





What's wrong with our implementation?

↳ An edge is stored twice

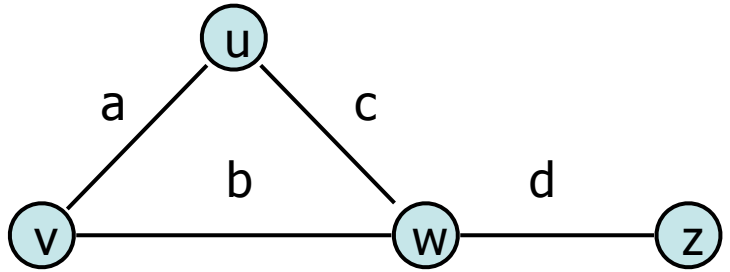
once for v_1 & once for v_2

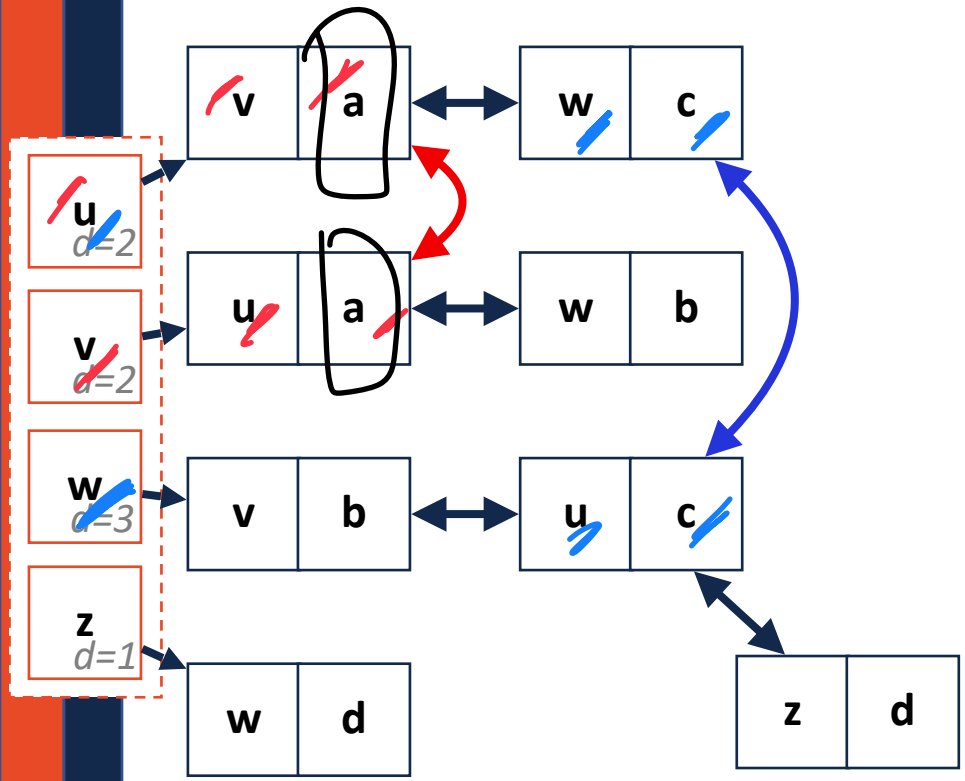
How can we fix it?

maybe more pointers?

Adjacency List

|V|= n, |E|= m



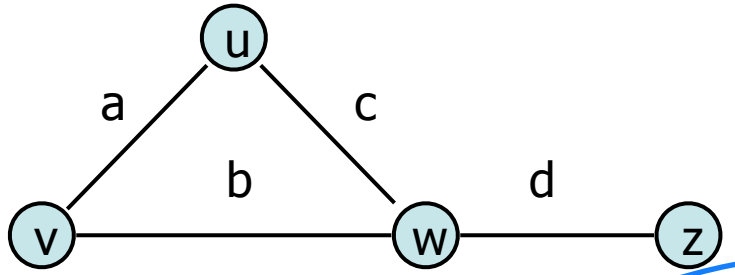


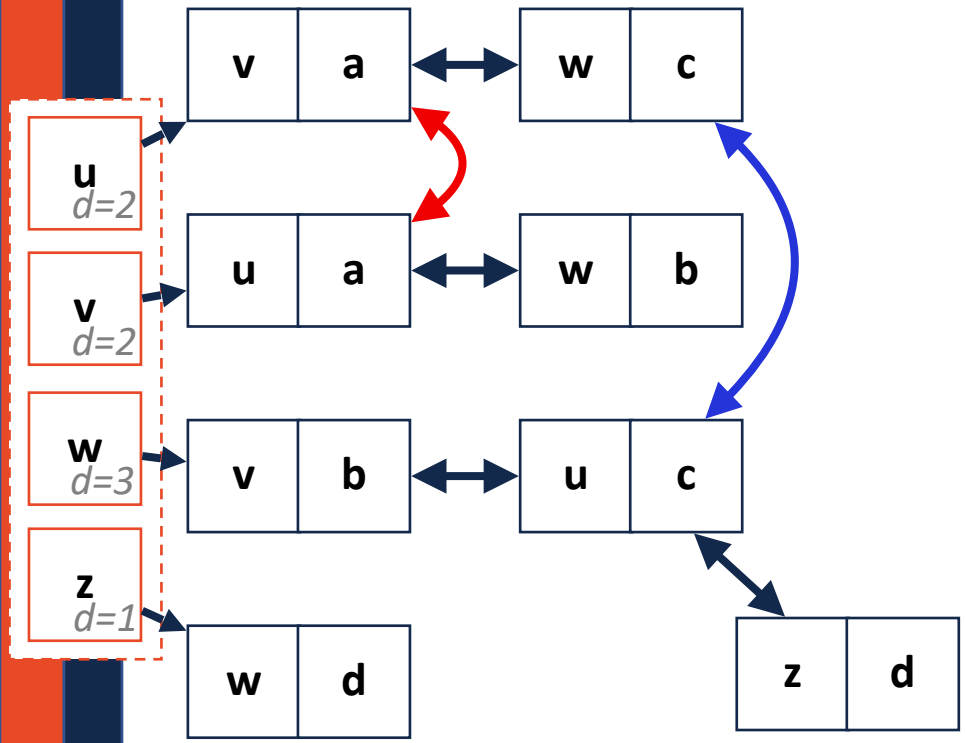
More pointers fixes life again! :)

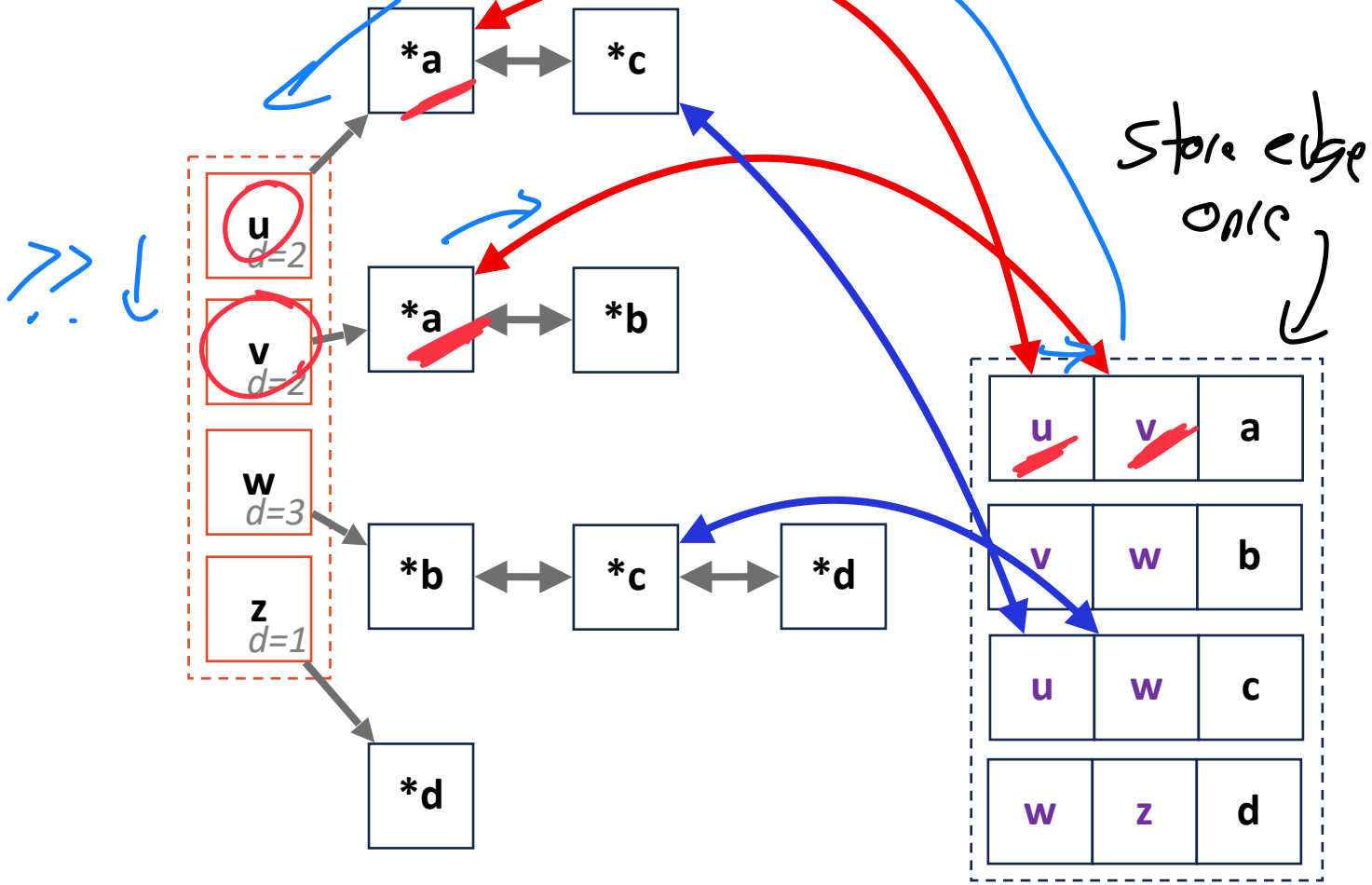
Still storing 'a' twice.

Adjacency List

|V|= n, |E|= m

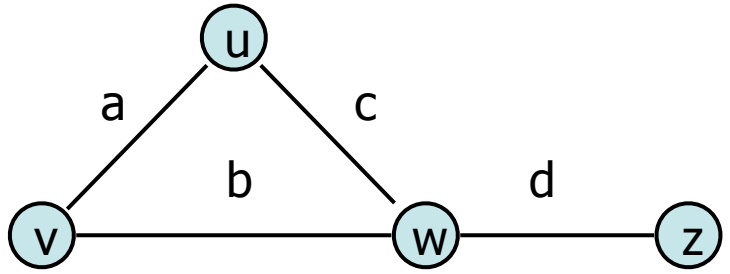






Adjacency List

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



Adj List Node:

*Prev	*Edge	*Next

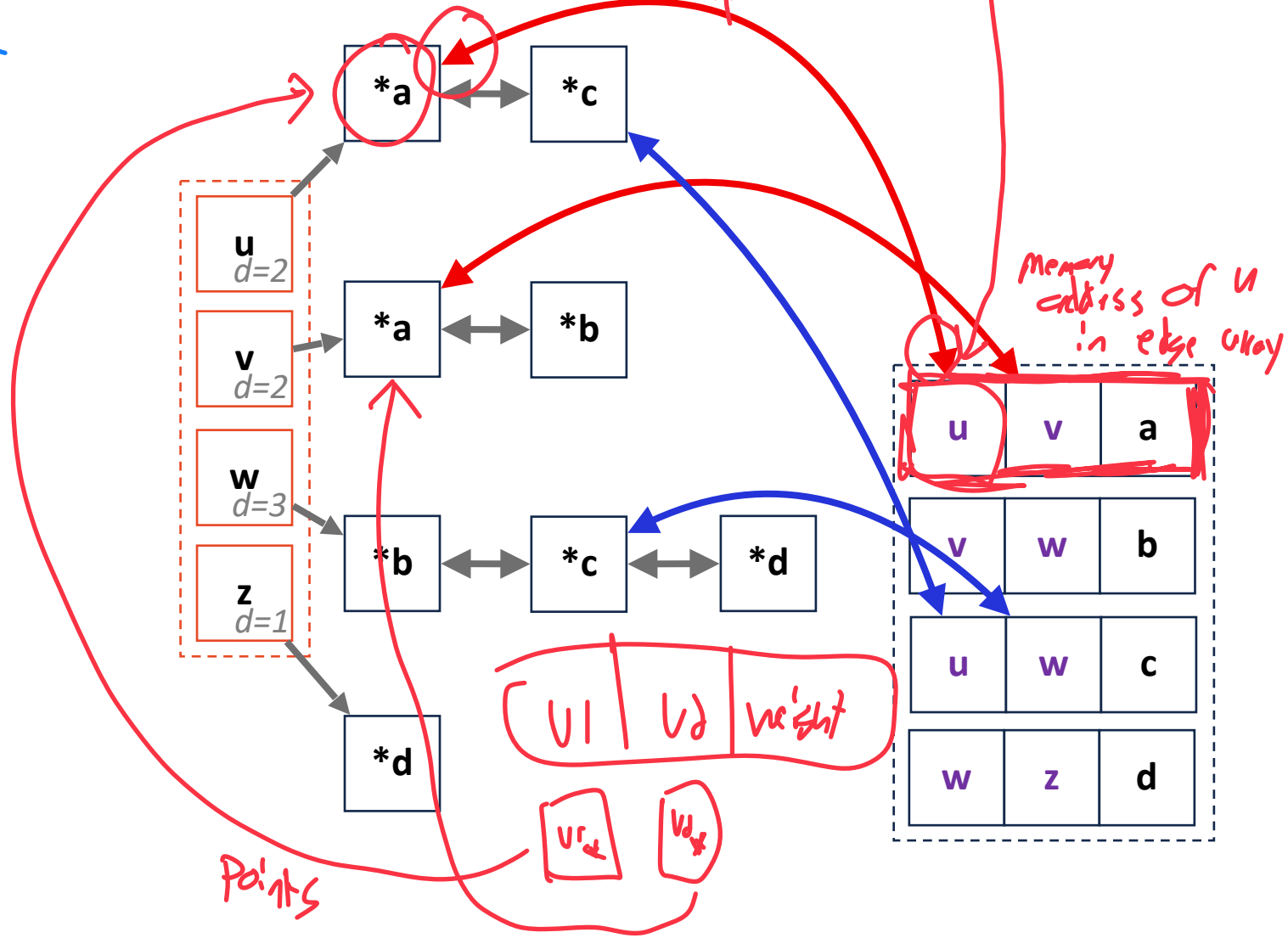
doubly linked

Edge List:

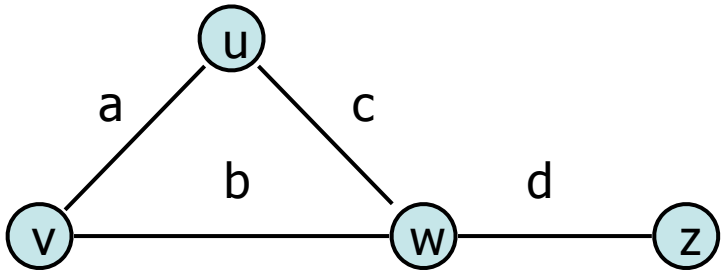
naive

V1	V2	Weight
*V1	*V2	

→ additional pointers



Adjacency List

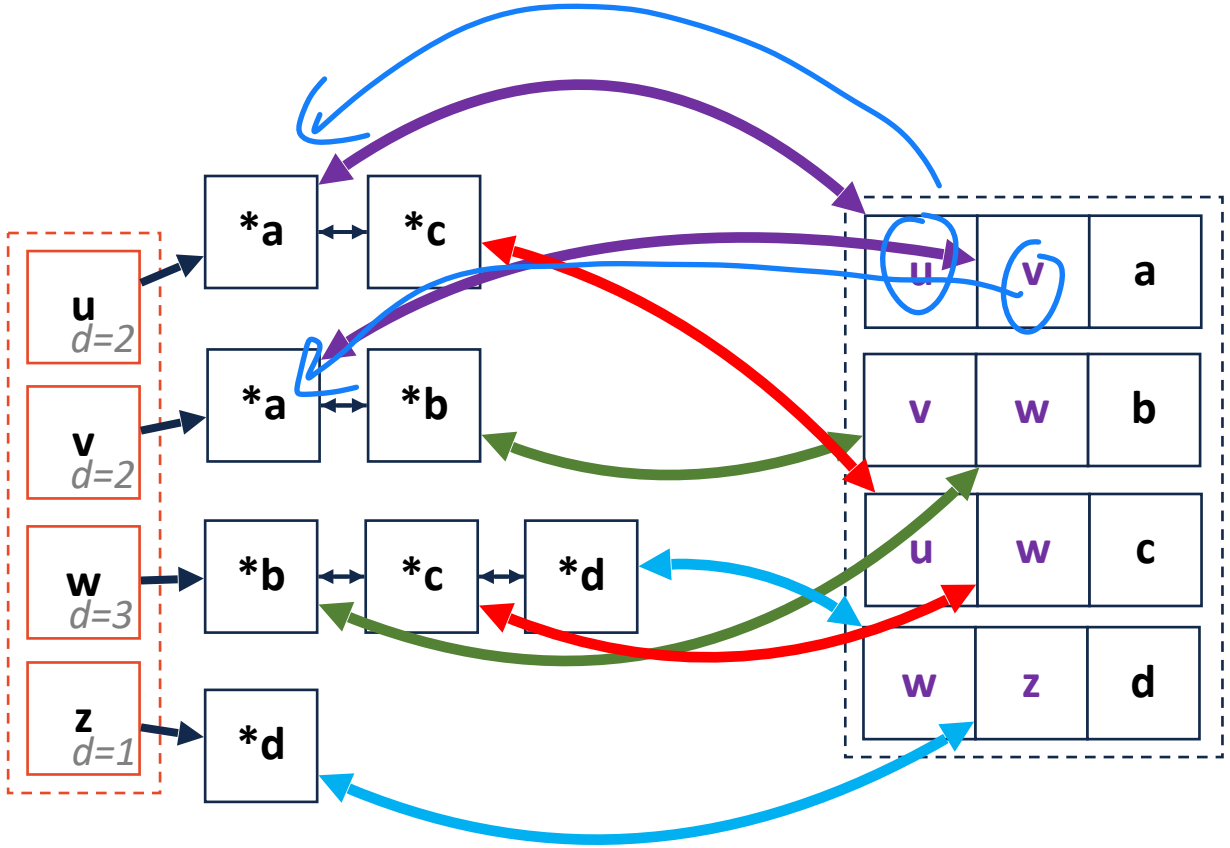


Vertex Storage:

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

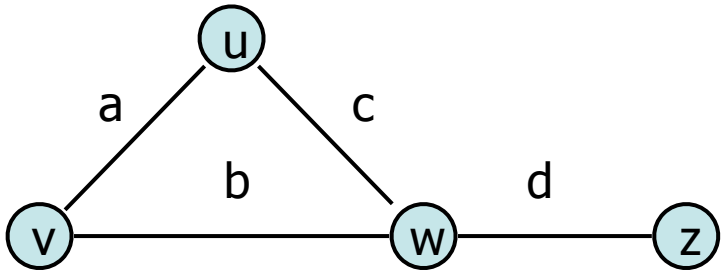


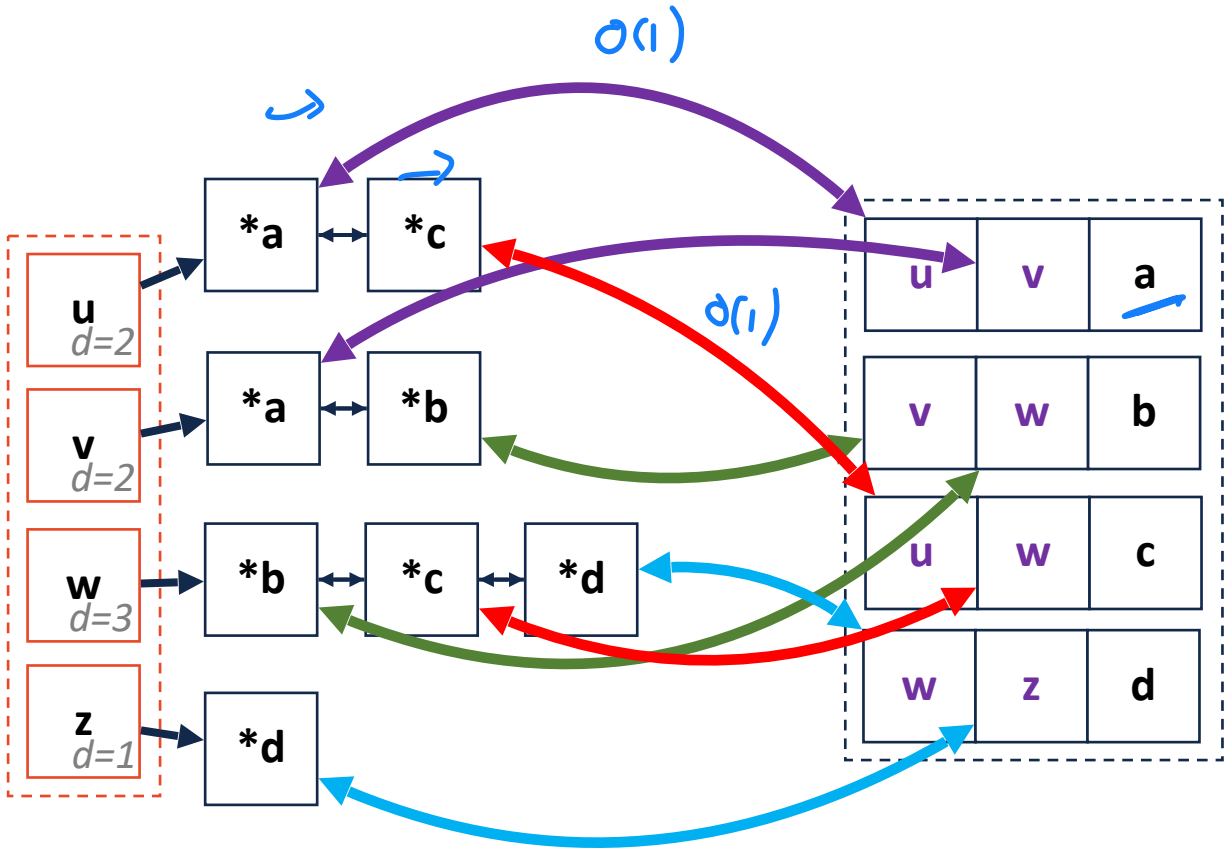
Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

Adjacency List





incidentEdges(Vertex v):

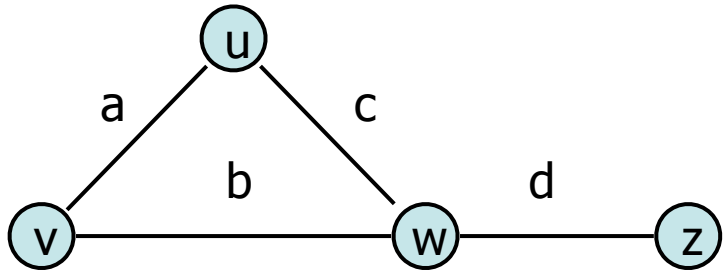
Look up vertex list (and walk across it)

[Each edge is a pointer lookup away]

O[deg(v)]

11

Adjacency List

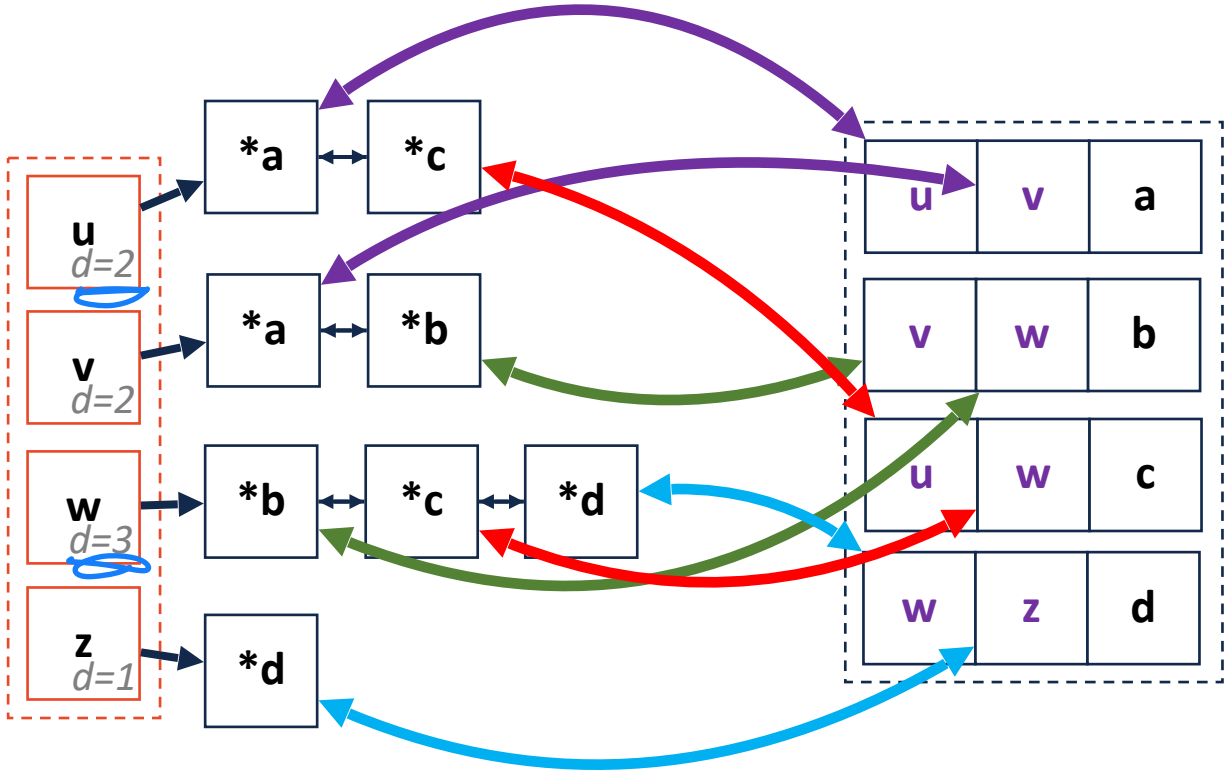


areAdjacent(Vertex v1, Vertex v2):

Look up min-degree vertex list

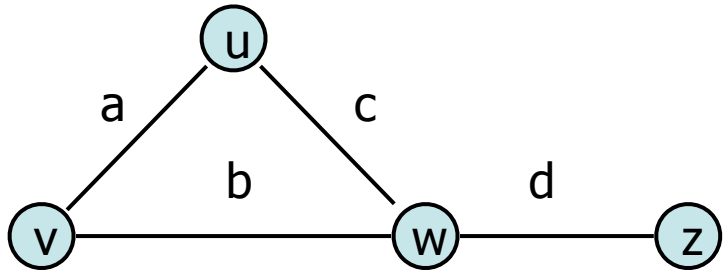
Search for other vertex across list

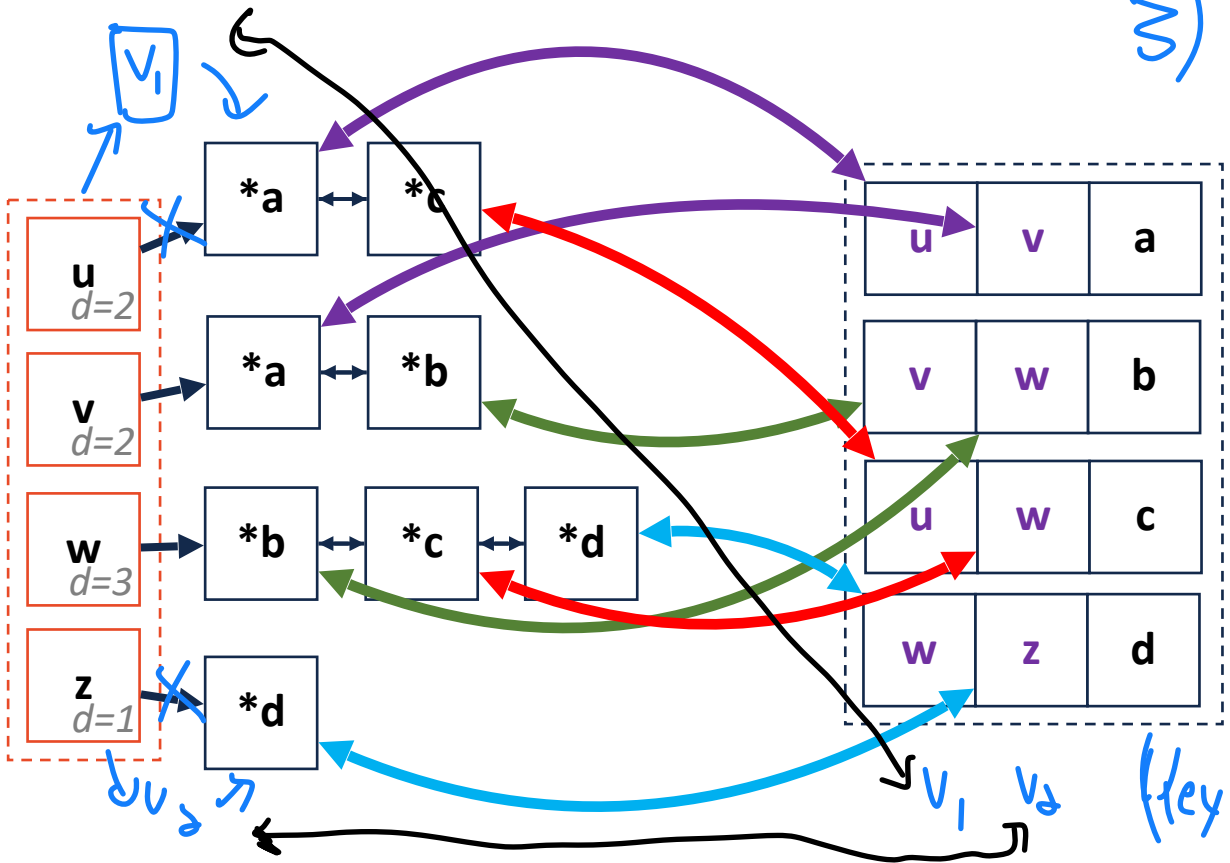
↳ I follow one extra pointer to reach edge list



$O\left[\min\left(\deg(v_1), \deg(v_2)\right)\right]$

Adjacency List





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

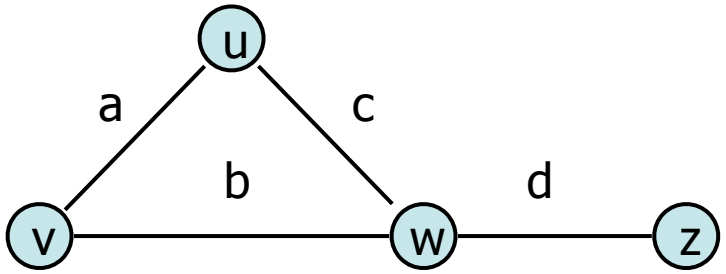
1) Add edge to edge array O(1)*

2) Add node to two linked lists
   ↳ Insert front? O(1)

3) All are connected by pointers O(1)

O(1)*

Adjacency List

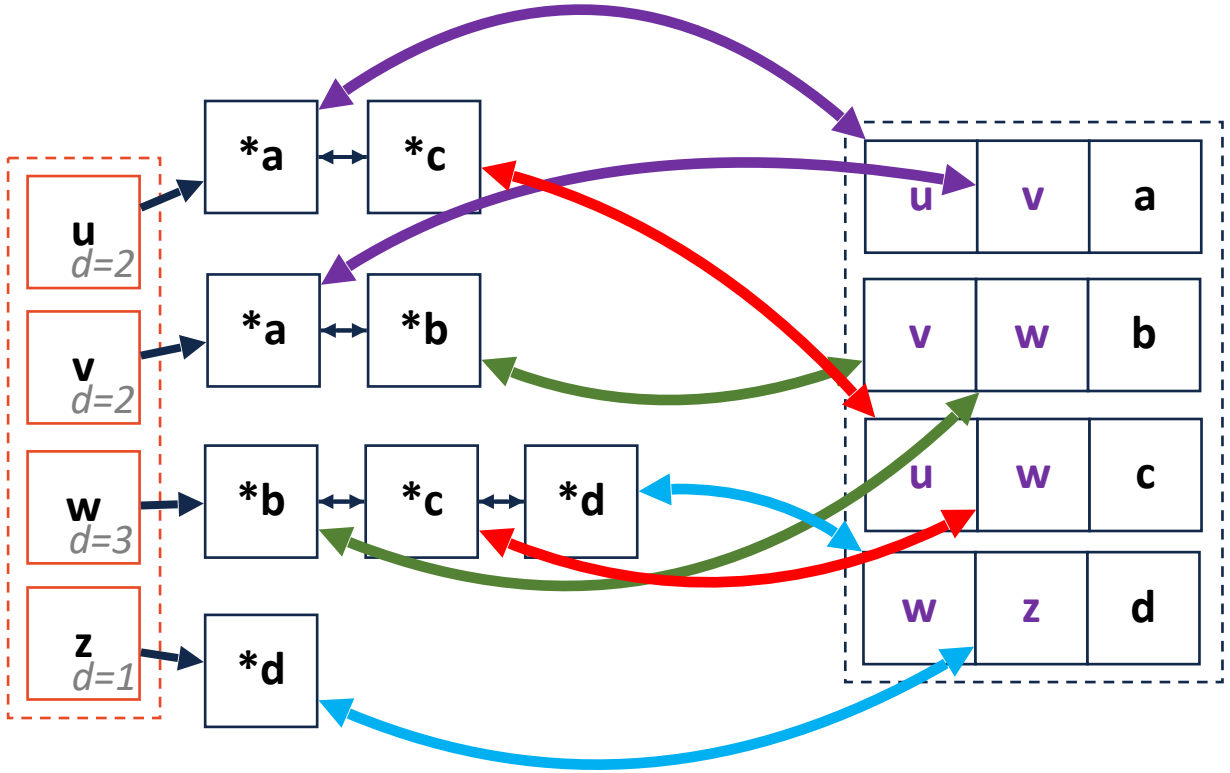


insertEdge(Vertex v1, Vertex v2, K key):

Add edge to edge list

Add node to each vertex list

Connect all three with pointers

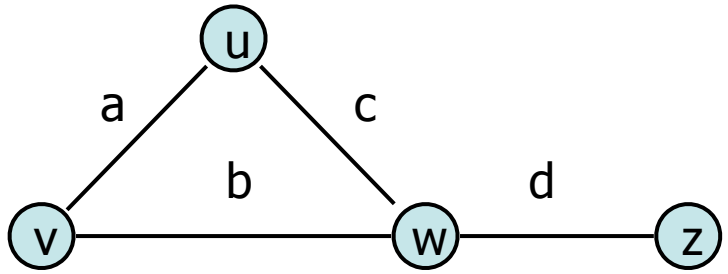


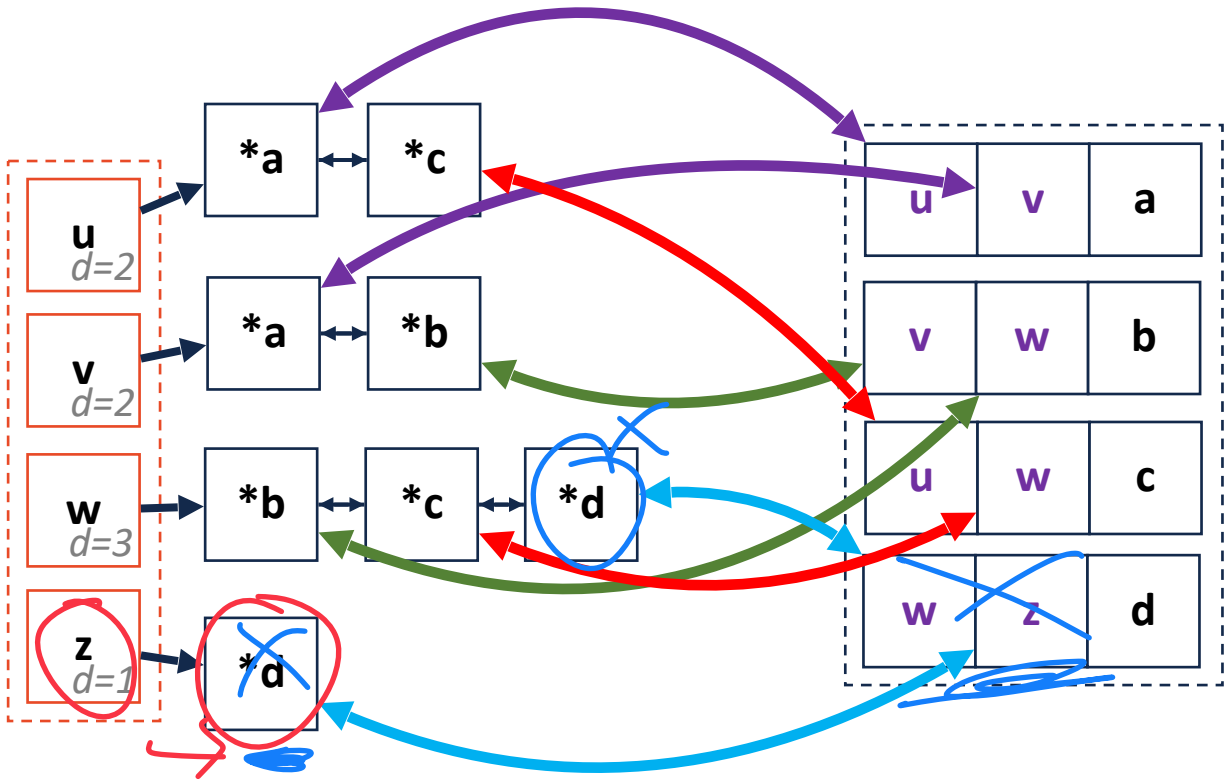
O(1)* 😊

Join Code: 225

What are my Big Os?

Adjacency List





removeEdge(Vertex v1, Vertex v2):

1) Find v_1 or v_2 in vertex map

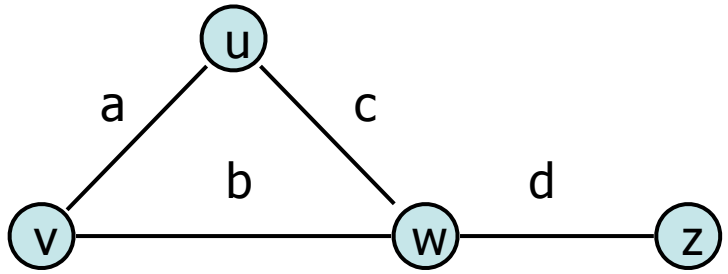
2) Walk down linked list

Remove three things:

1) entry in edge list
2) The linked list node in v_1
3) (Same) in v_2

↳ These are all connected!

Adjacency List



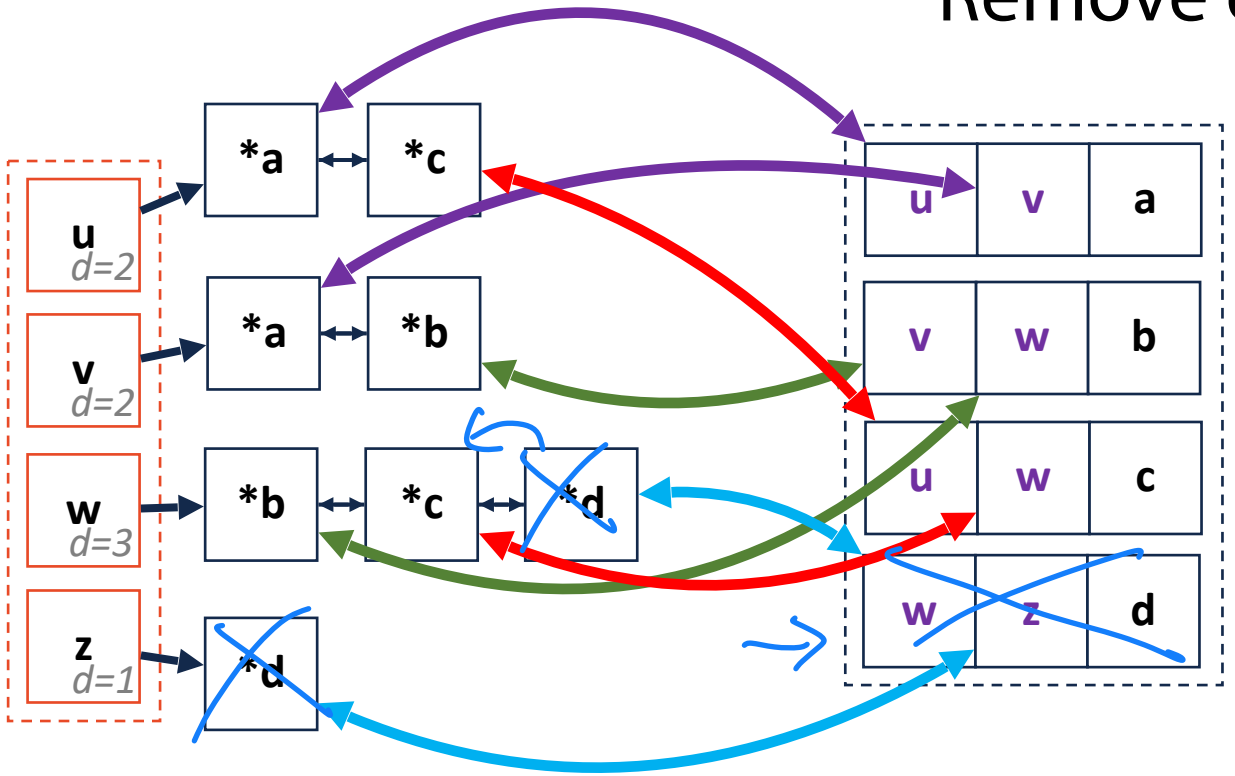
removeEdge(Vertex v1, Vertex v2, ~~Key~~):

Search min-degree vertex list

Remove mirrored entry using pointers

Remove edge from edge list

Remove entry from vertex list



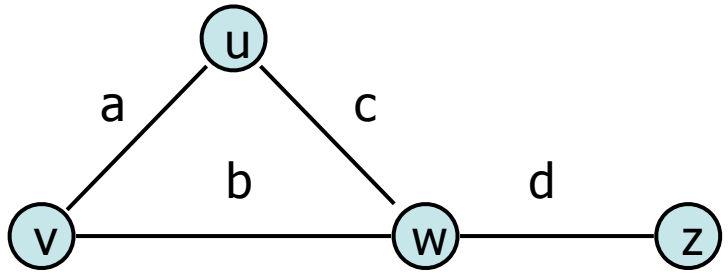
$\min [\deg(v_1), \deg(v_2)]$

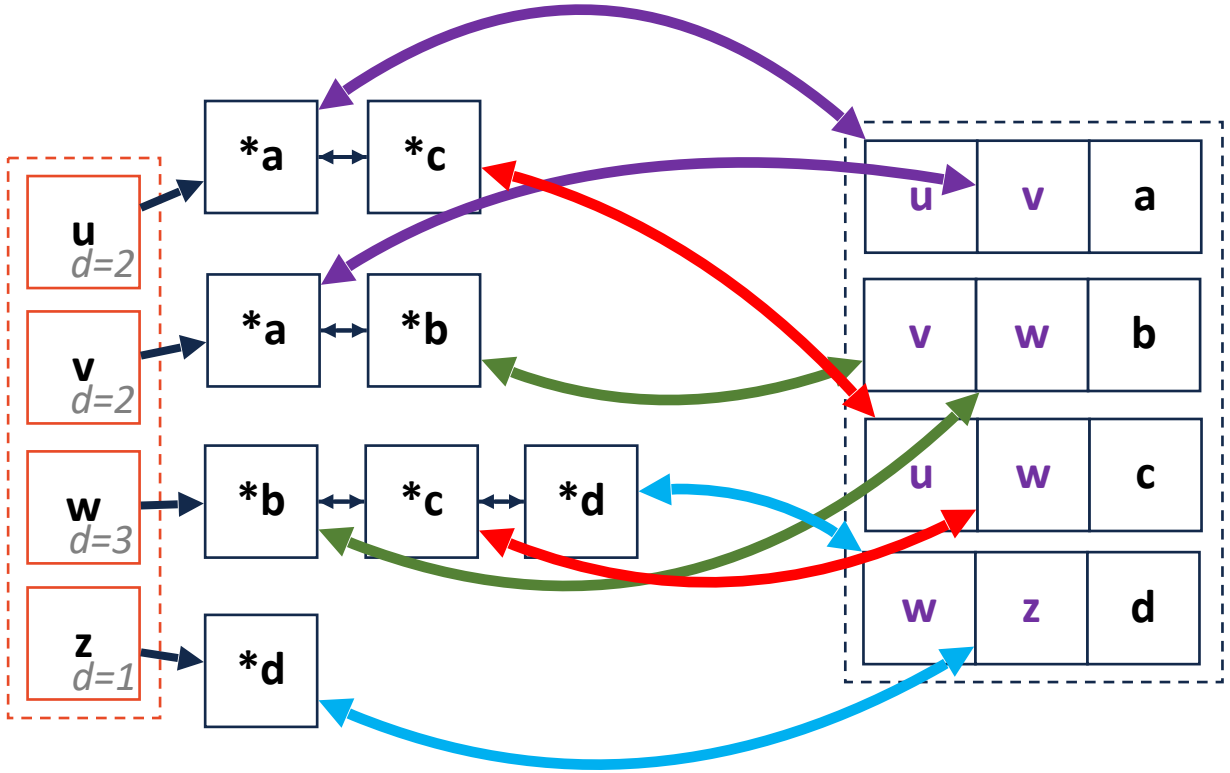
Join Code: 225

What are my Big Os?

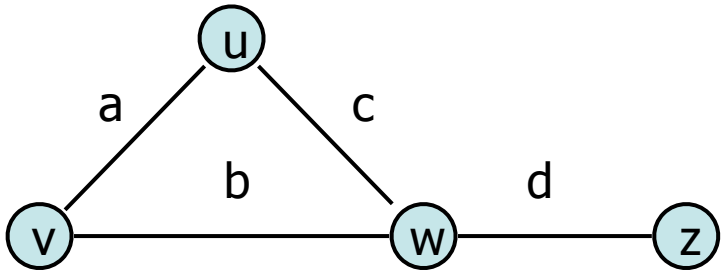
Adjacency List

insertVertex(K key):



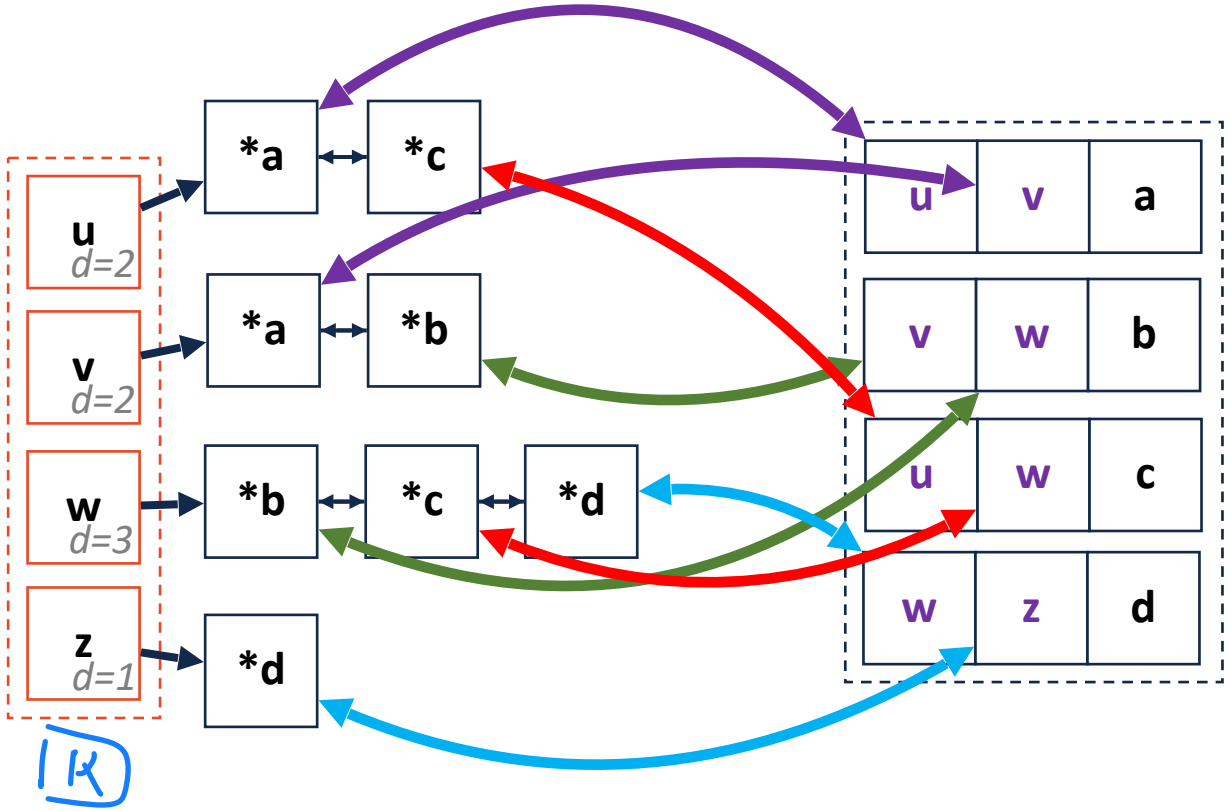


Adjacency List

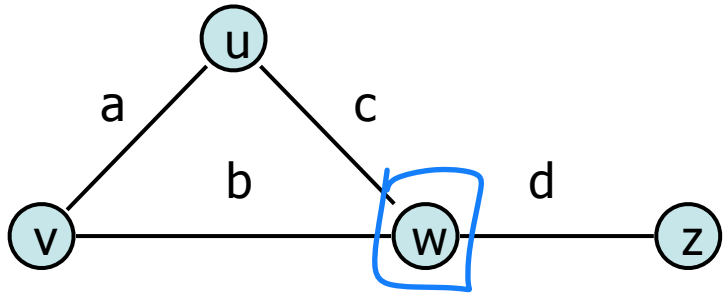


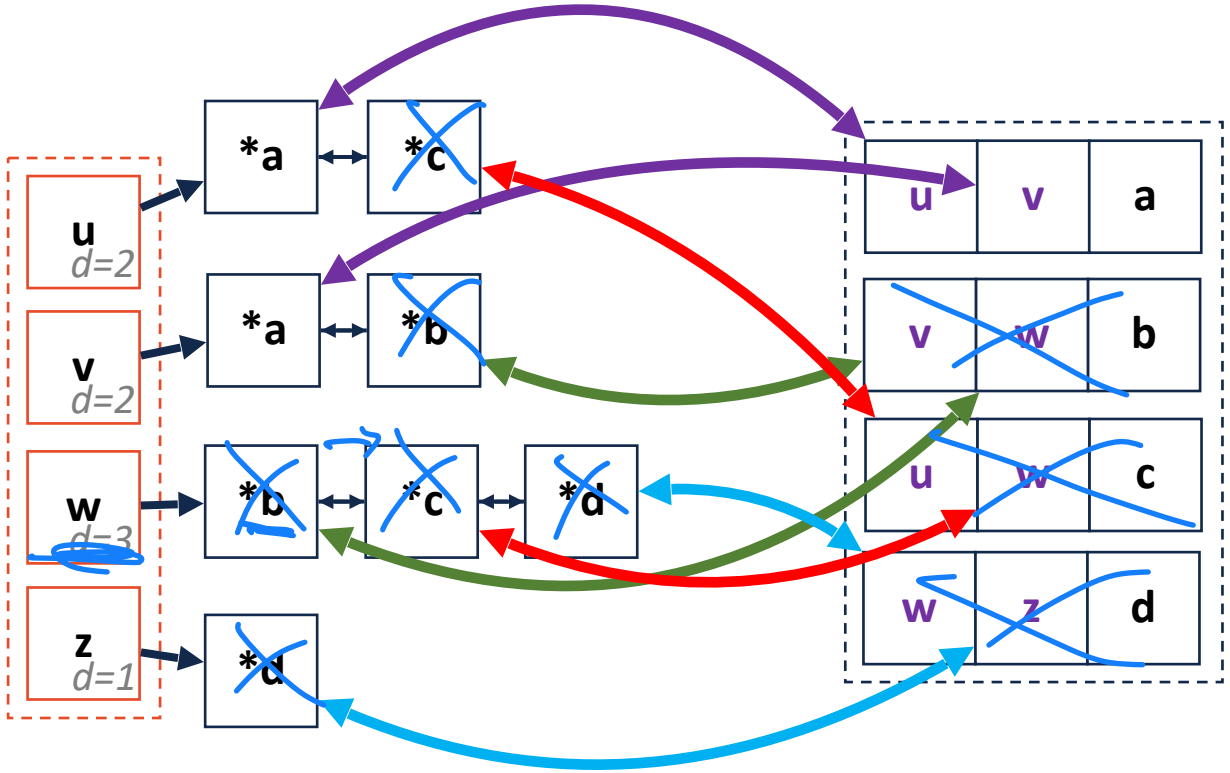
insertVertex(K key):

Add new empty list to vertex list.



Adjacency List





removeVertex(Vertex v): ≈ w

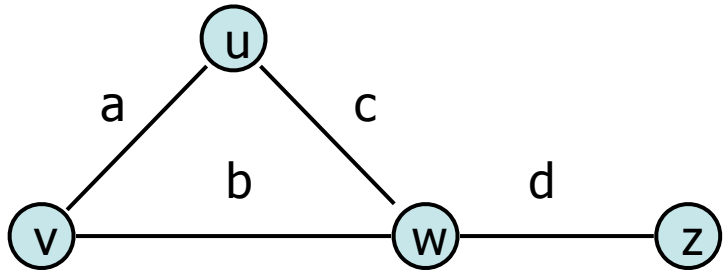
↳ Almost identical to removeEdge

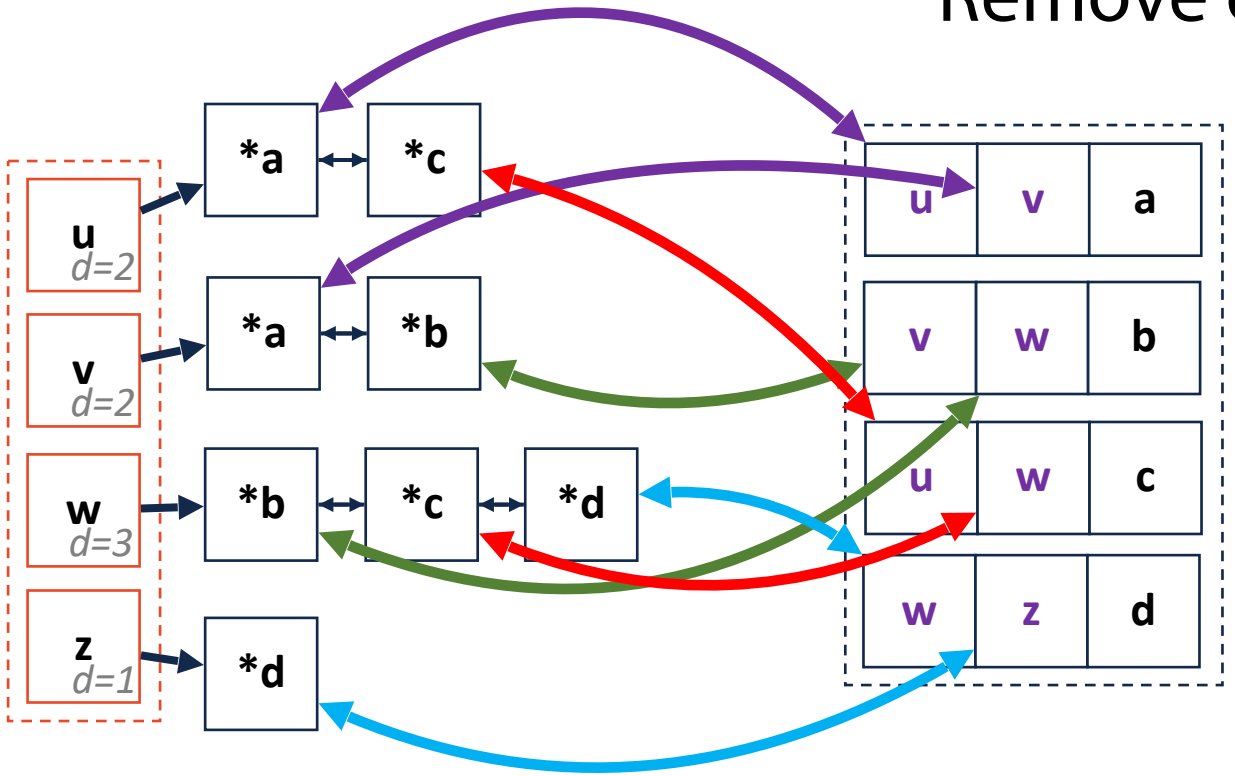
A single walk through V's linked list deleting all pointer links along the way

1. Delete opposite LL node O(1)
2. Delete entry in edge list O(1)
3. Delete LL node in V O(1)

O(1) for entry → degree(v) entries

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?

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

(compare when is best)

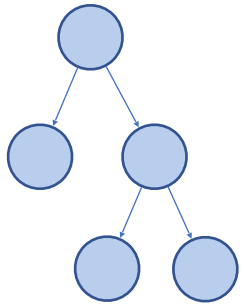
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

Graph Traversals

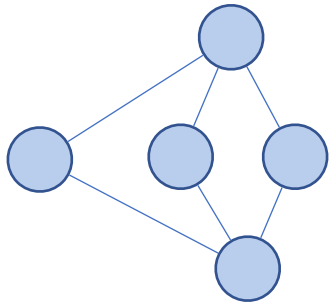
There is no clear order in a graph (even less than a tree!)

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

Tree traversals won't work — lets compare:

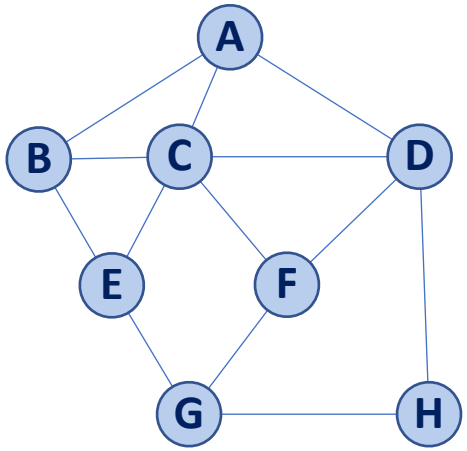


- Rooted
- Acyclic
-

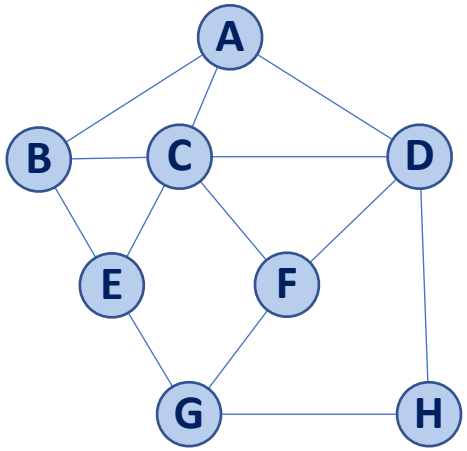


-
-
-

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