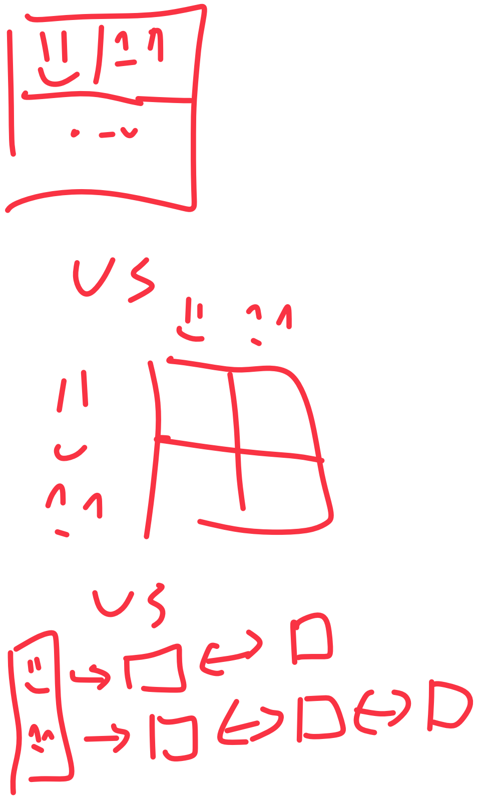
Data Structures

Graph Implementations 2

CS 225
Brad Solomon

October 25, 2024



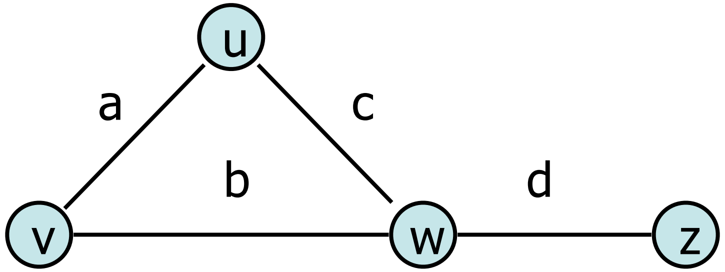
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

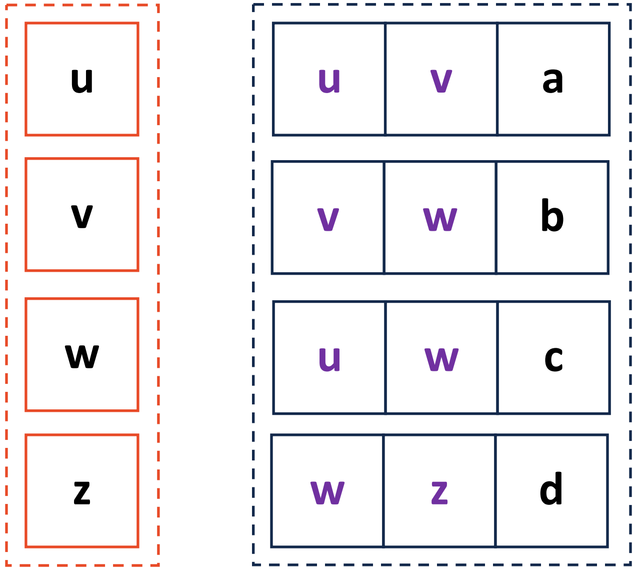
Learning Objectives

Discuss graph implementation and storage strategies

Introduce graph traversals

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





O(1)*

insertVertex(K key):

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

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

removeVertex(Vertex v):

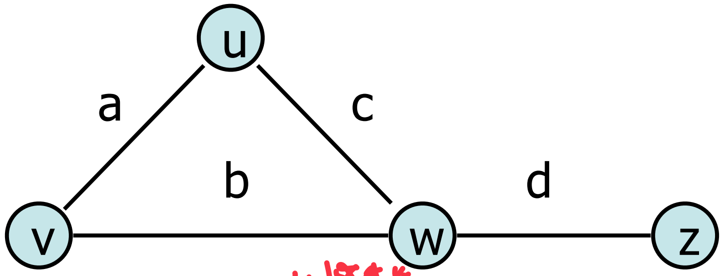
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

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



O(1)***

hash table

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

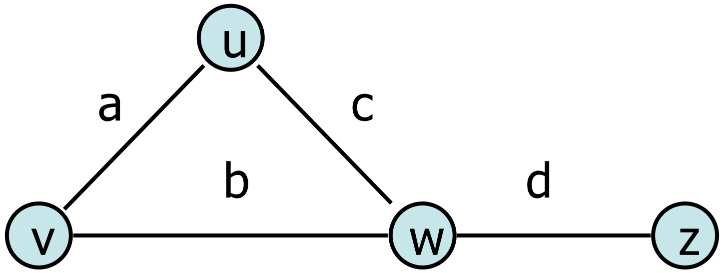
Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



u	0
v	1
w	2
z	3

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

upper diagonal storage

removeVertex(Vertex v):

1) Look up both row & col

2) Replace w/ tombstone value

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

7 | 9 | 1 | 0 | 8

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

Adjacency Matrix:

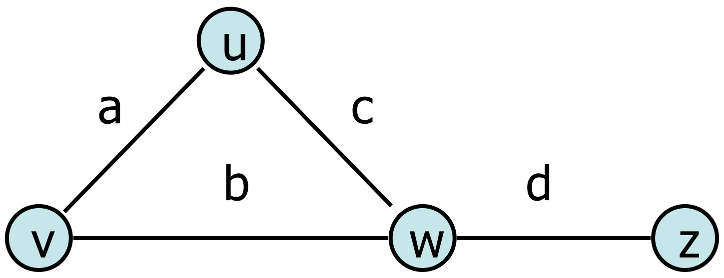
removeVertex() by tombstoning |V| values

Resize when needed or by request

O(1)

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



- u
- v
- w
- z

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

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):

$O(n)$—$O(n^2)$ ← Implementation dependent

insertVertex(K key):

removeVertex(Vertex v):

And $O(n^2)$ to store!

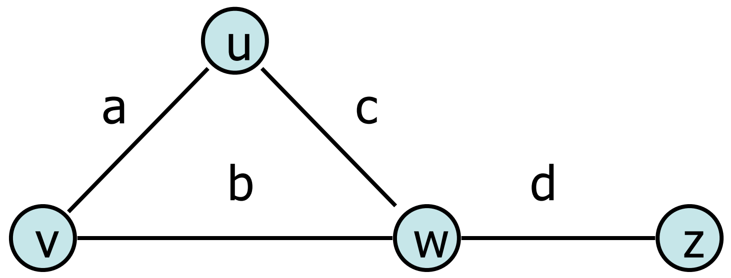
Graph Implementation Brainstorming

We want something…

Faster than an edge list

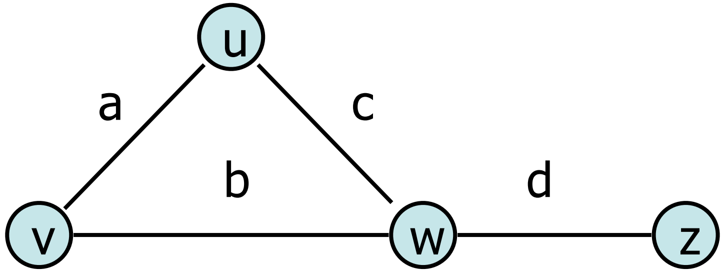
Less space than an adjacency matrix

Particularly good at **finding adjacent elements**

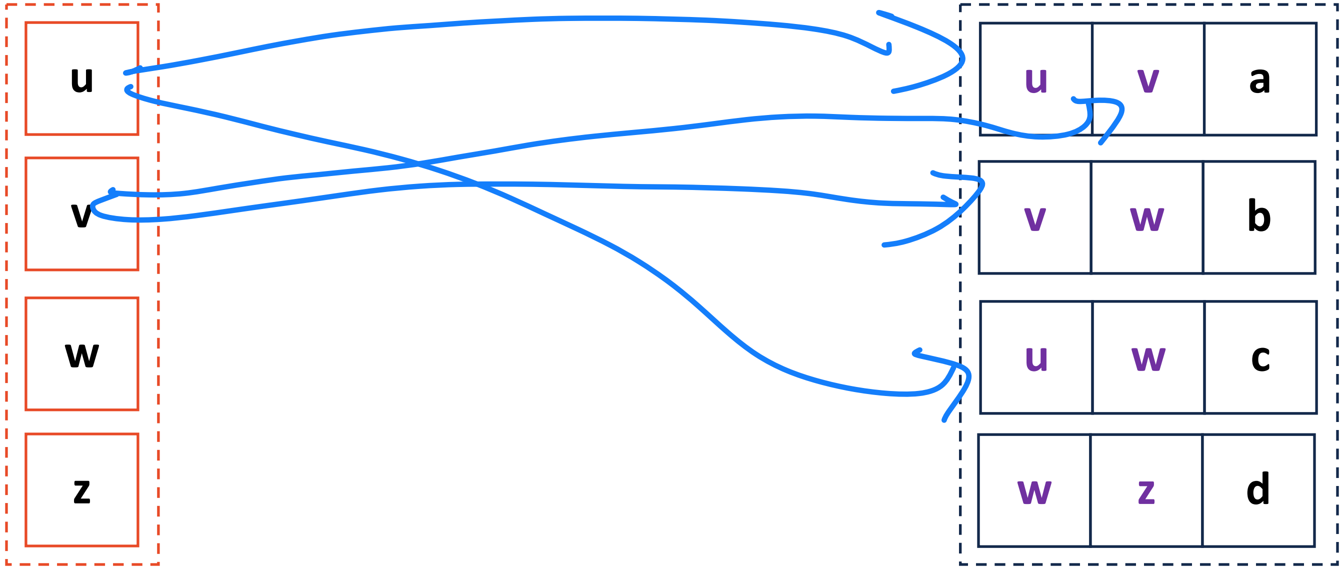


Graph Implementation: Edge List + ?

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



Pointers!

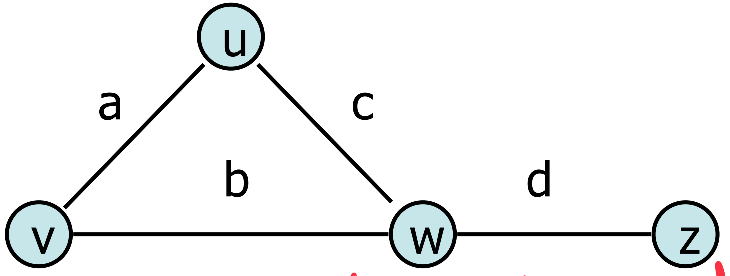


Look through entire list

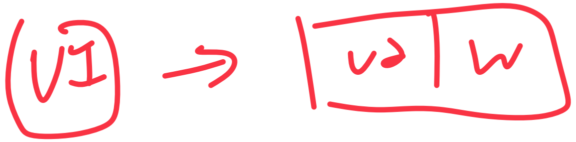
Naive Adjacency List

Vertex list of linked lists

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

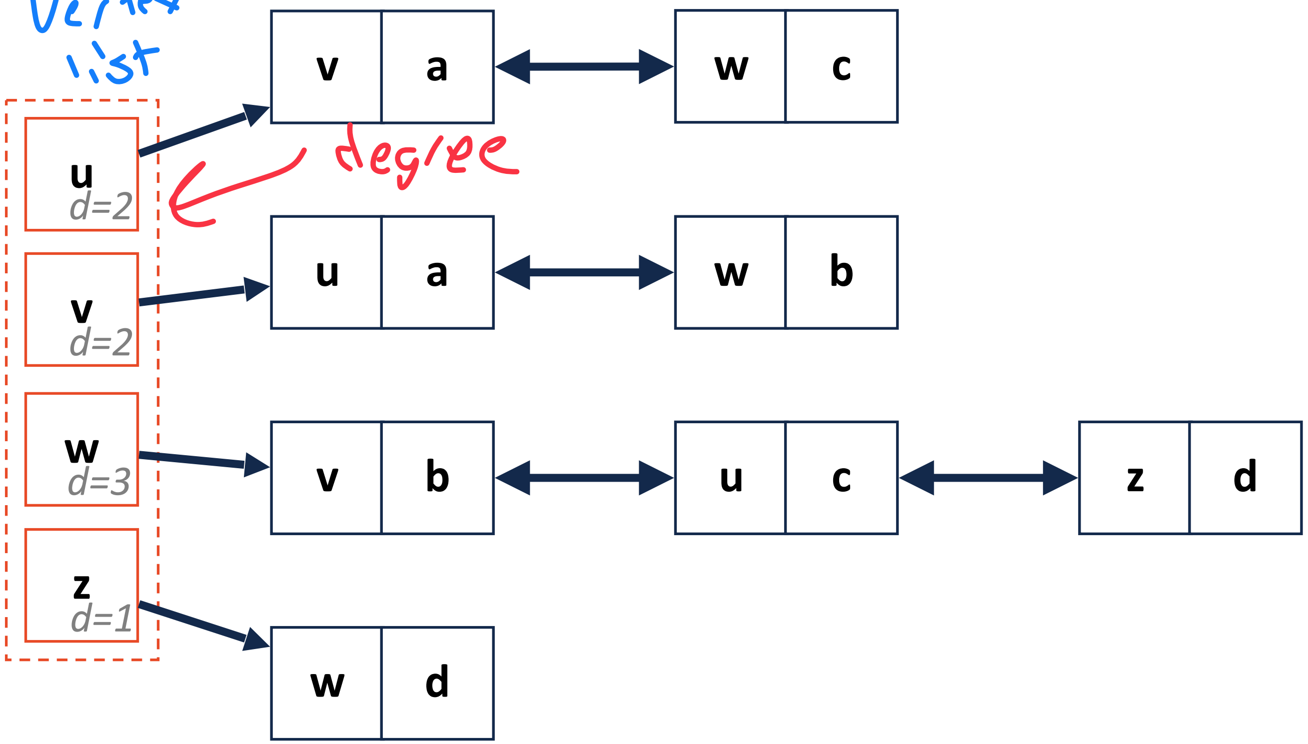


Bidirectional edge list w/ size



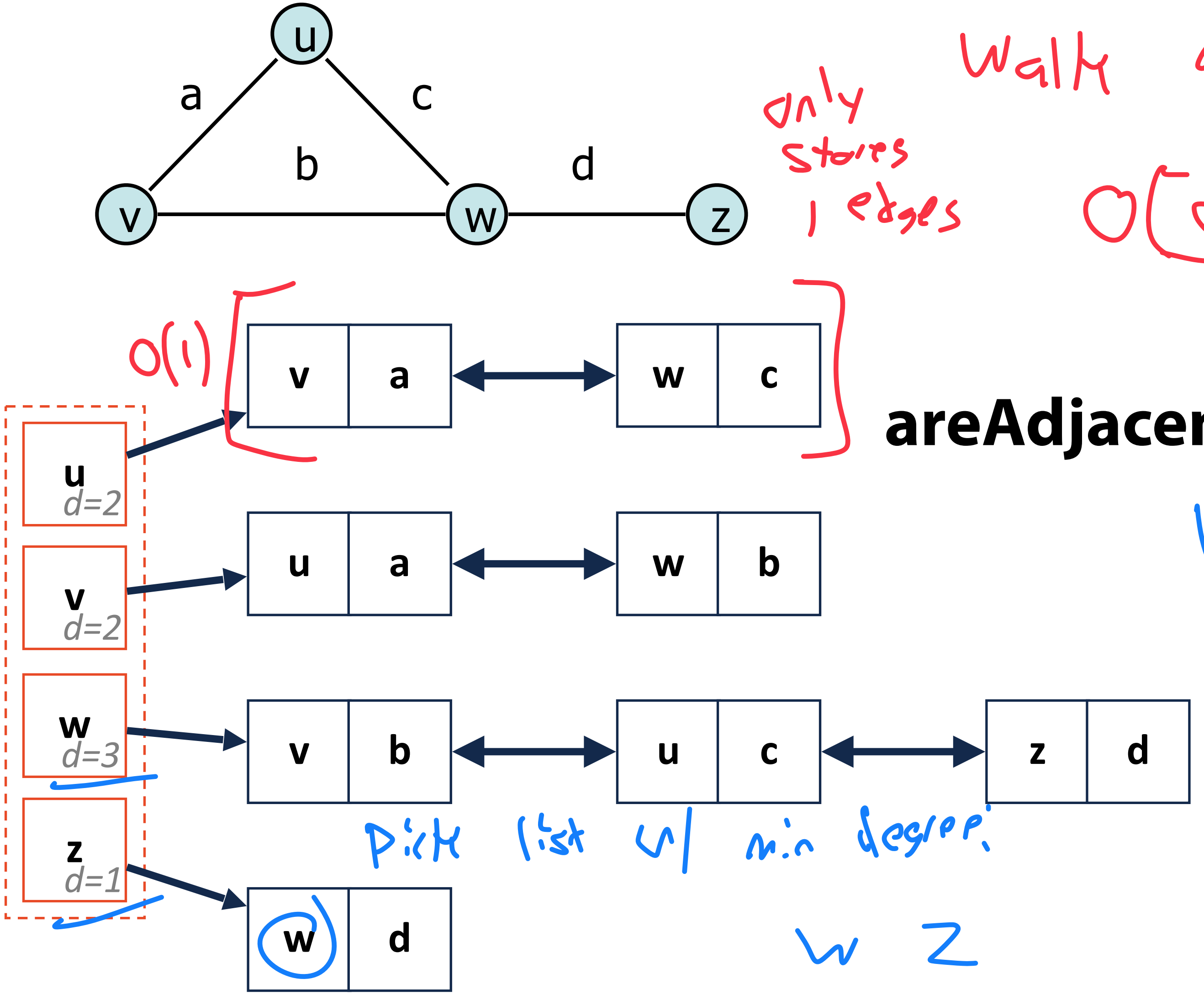
Vertex list

degree



Naive Adjacency List

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



incidentEdges(Vertex v):

Walk across edge list

$O[\deg(v)]$ ← optimal for deg(v) neighbors

areAdjacent(Vertex v1, Vertex v2):

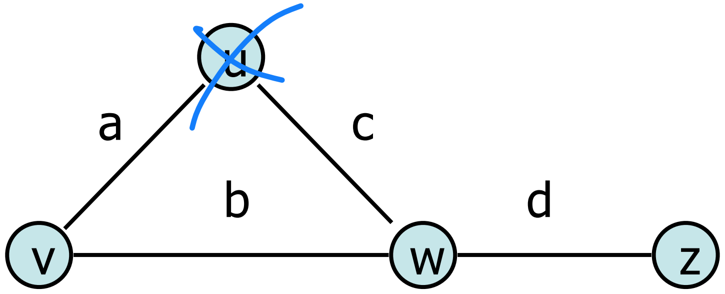
Walk across edge list

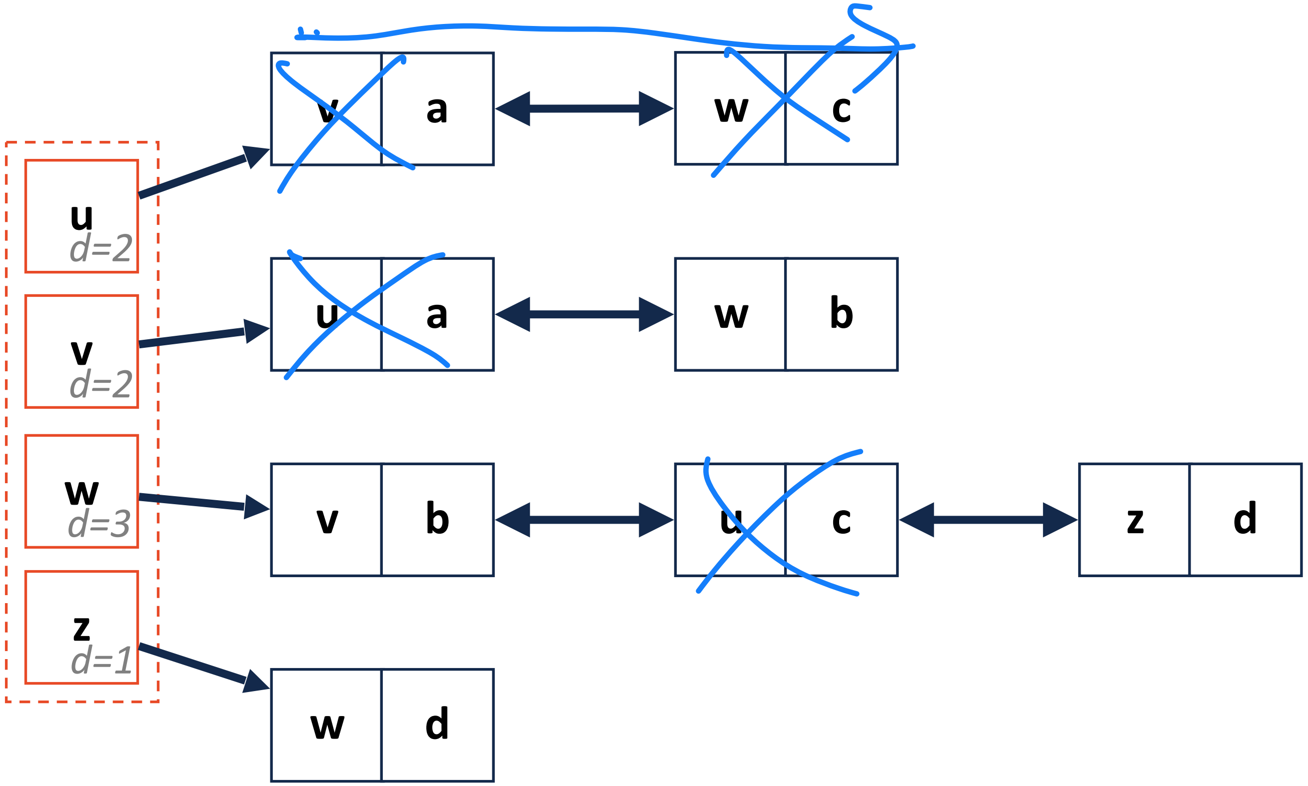
↳ search for v2

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

Naive Adjacency List

|V|= n, |E|= m



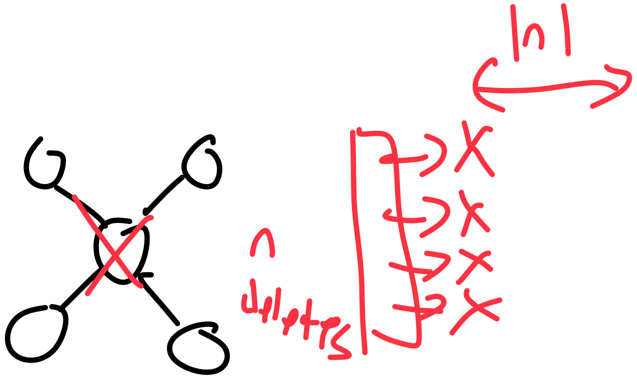


removeVertex(Vertex v):

Walk along vertex's edge list

↳ Have to delete in v list

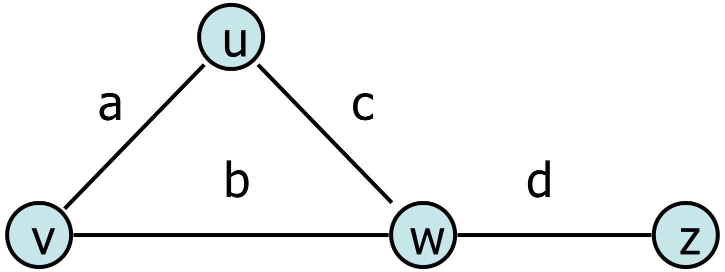
but also in every neighbor

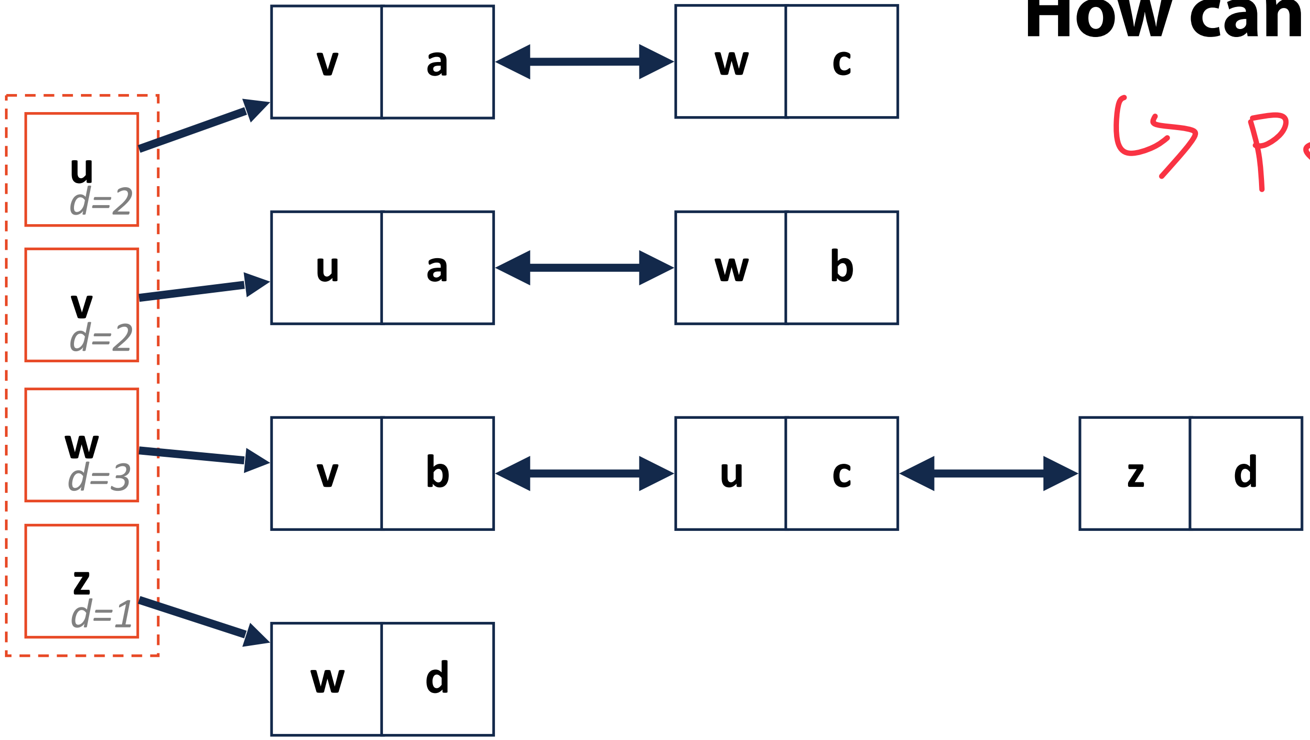


$O(n^2)$

Naive Adjacency List

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





What's wrong with our implementation?

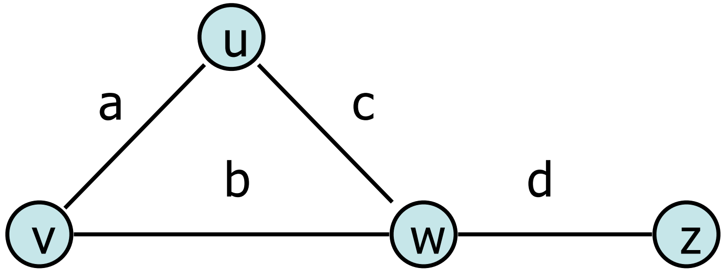
↳ Have to delete in up to every vertex

How can we fix it?

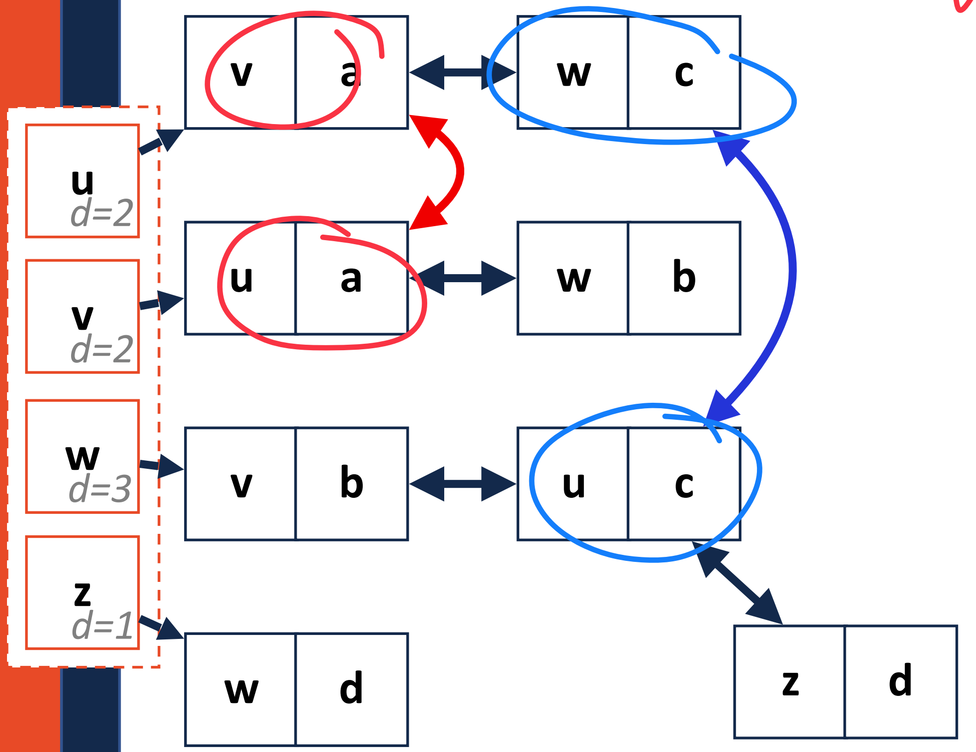
↳ Pointers!

Adjacency List

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



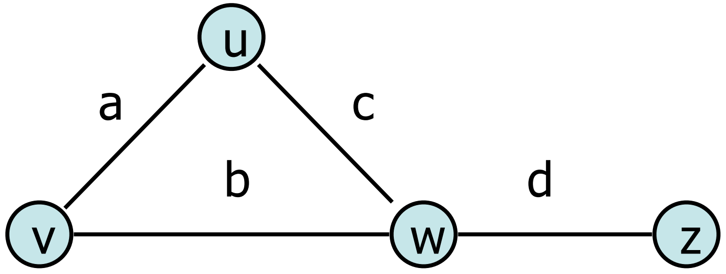
We want connections between listNodes

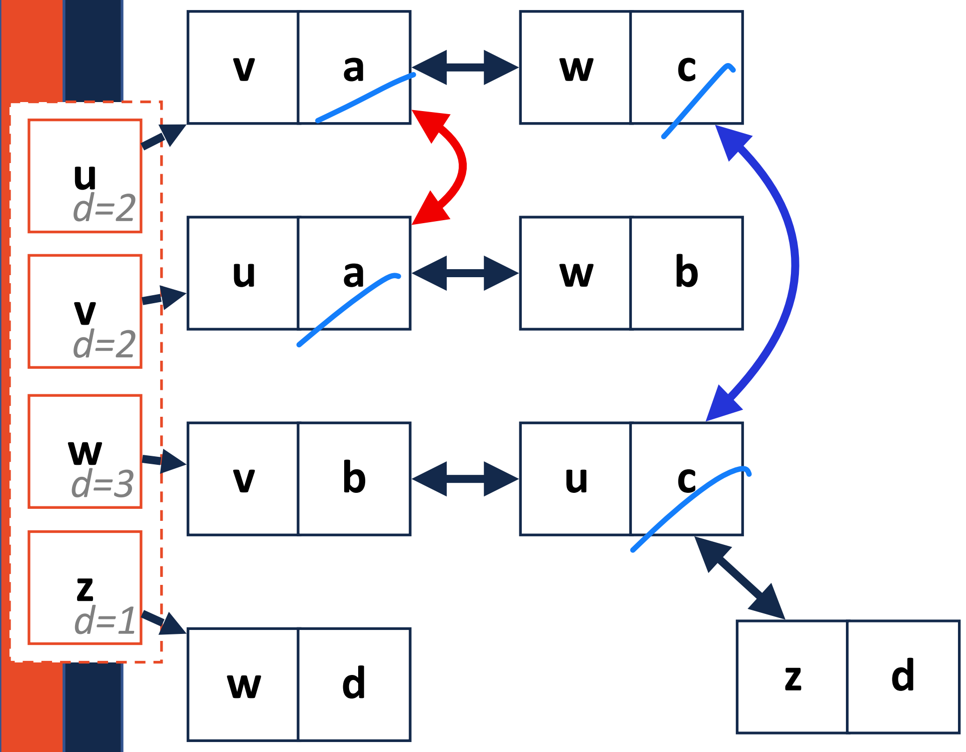


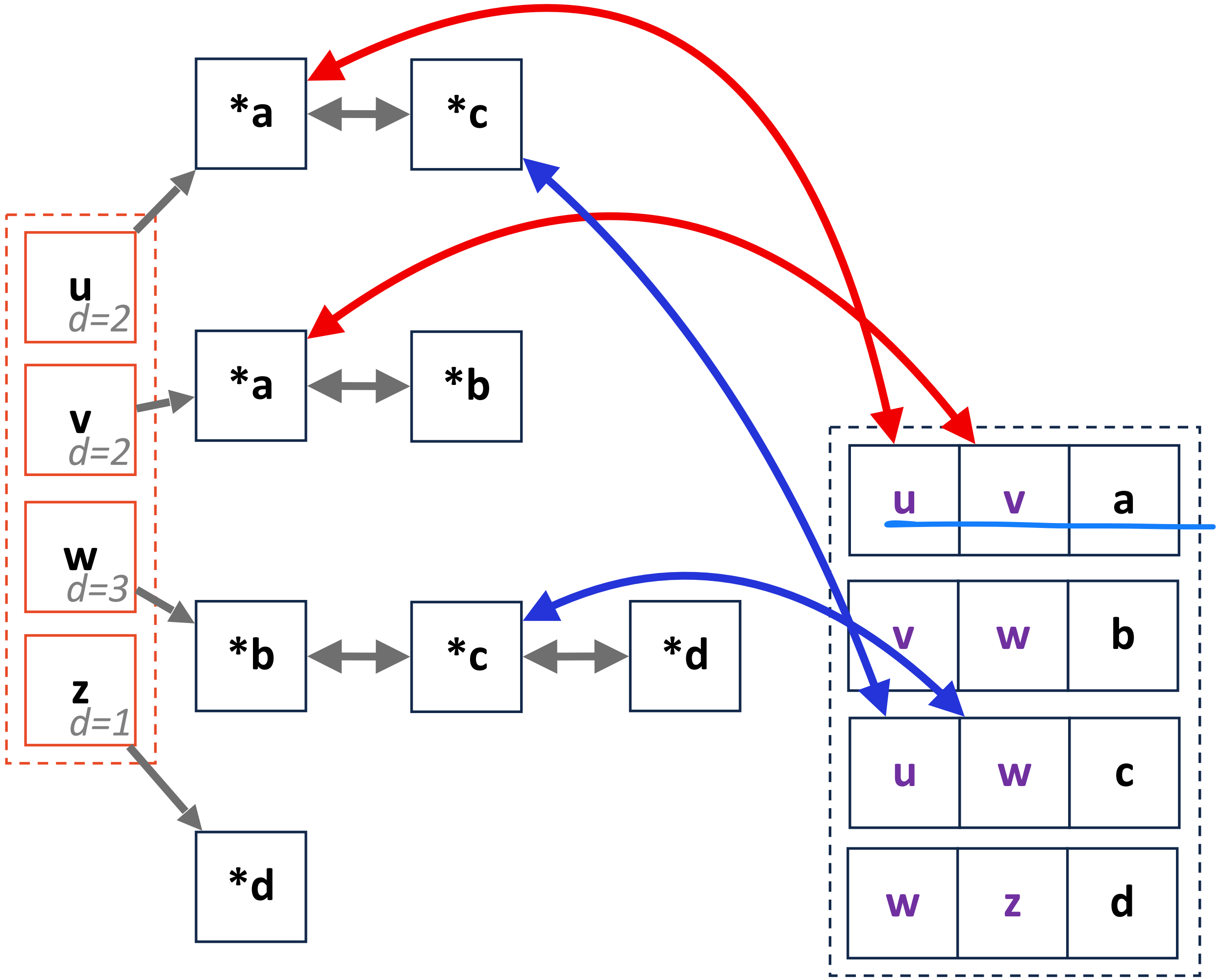
Adjacency List

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

1) No more double storing







Adjacency List

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

Adj List Node:

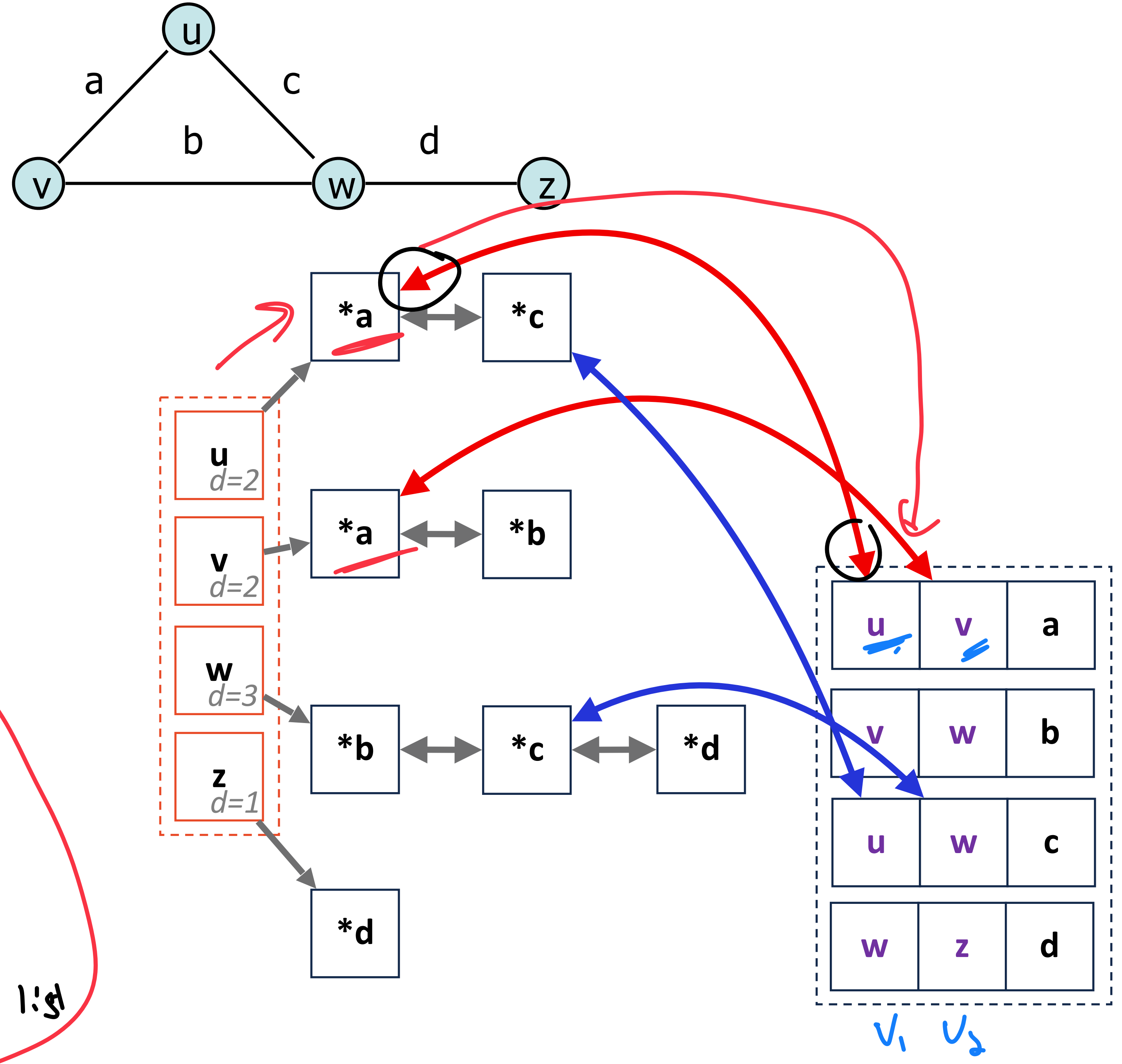
*Prev	*Edge	*Next

Edge List:

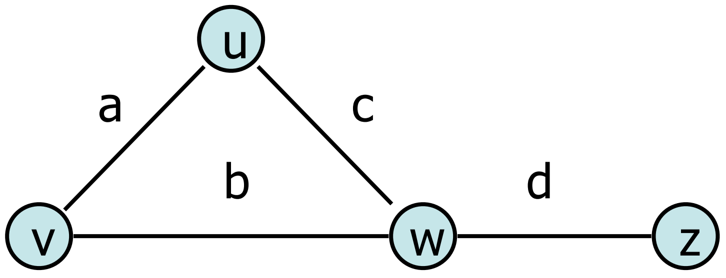
normal

V1	V2	Weight
*V1	*V2	

plus pointers back to adj list



Adjacency List



Vertex Storage:

A bidirectional linked list with size variable

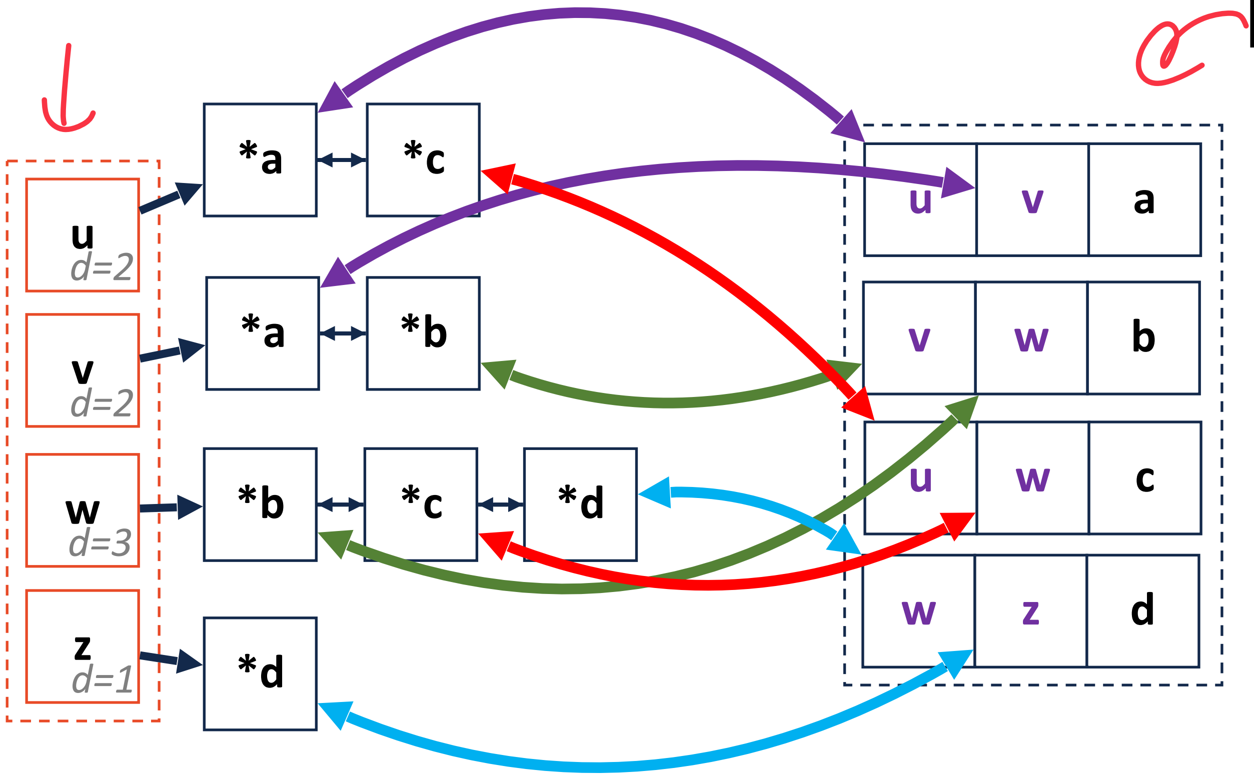
Each node is a pointer to edge in edge list

Hash table of linked lists

Edge Storage: edge list

A list of (v1, v2, weight) edges

Also store pointers back to nodes



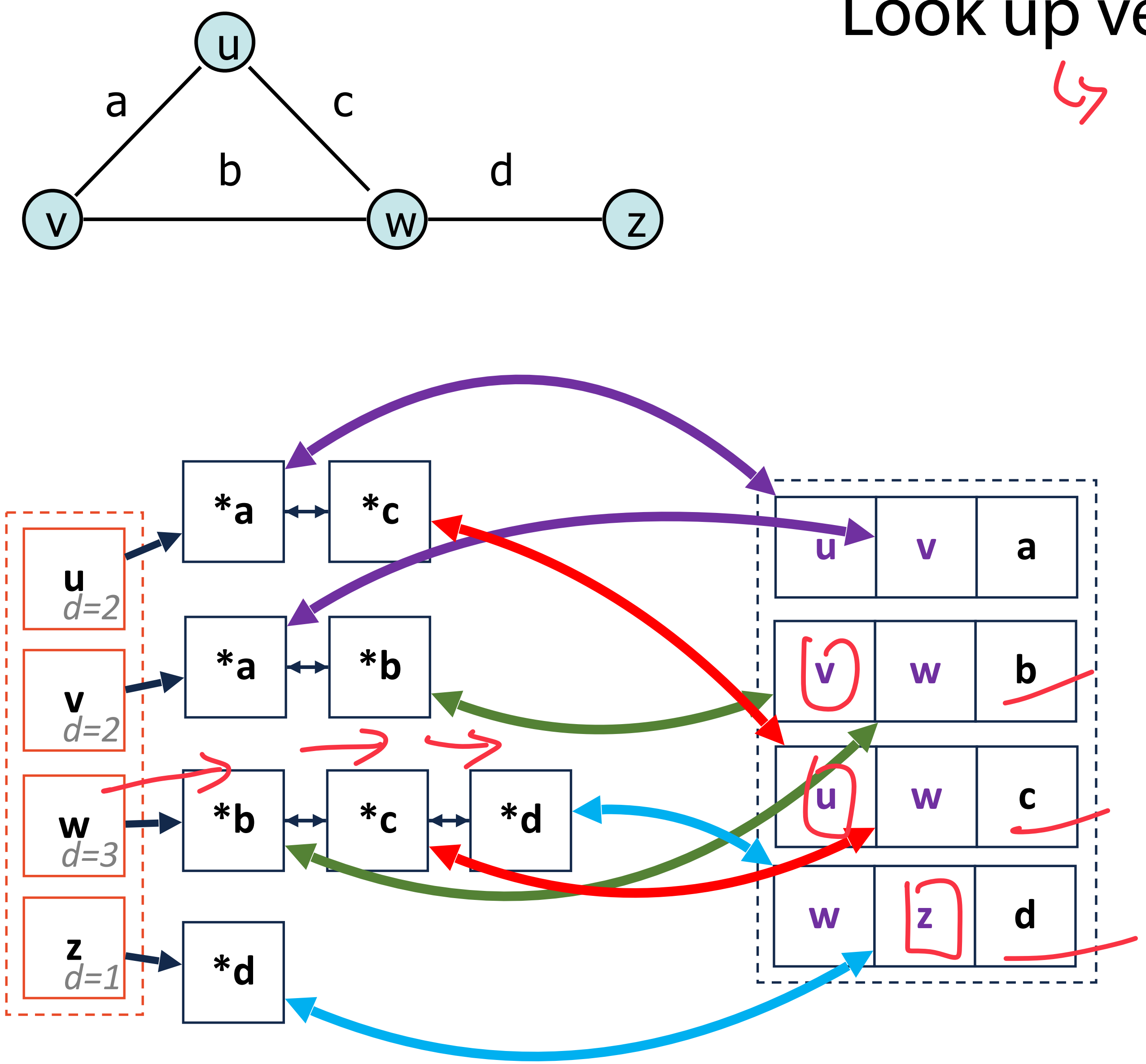
Adjacency List

!! **incidentEdges(Vertex v):** w

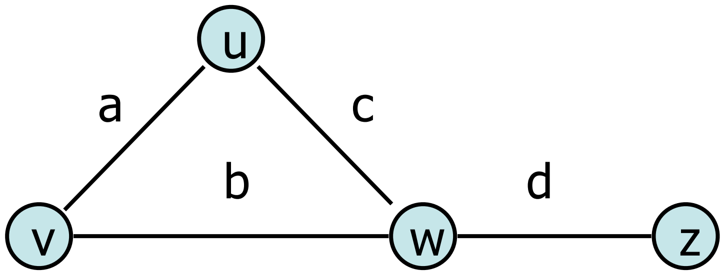
Look up vertex list (and walk across it)

↳ And look up edge by pointer

$O(\deg(v))$



Adjacency List

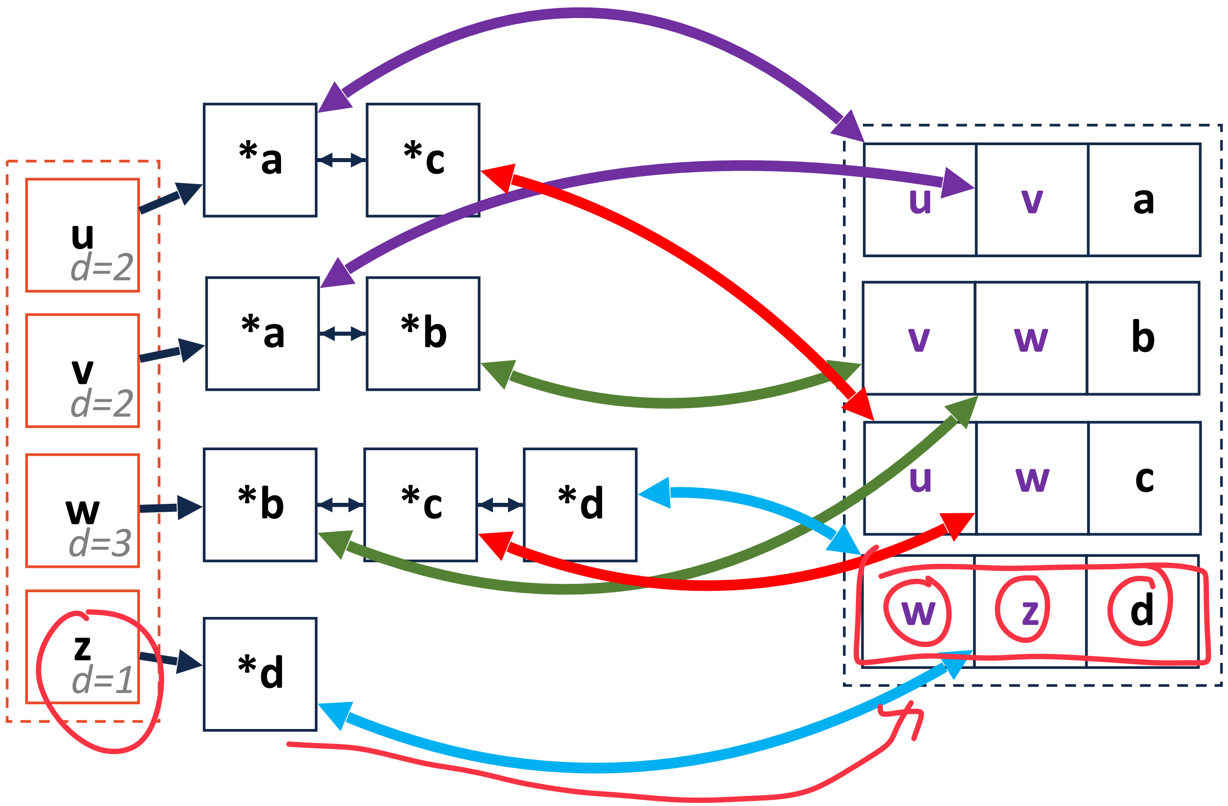


areAdjacent(Vertex v1, Vertex v2):

w z

Look up min-degree vertex list

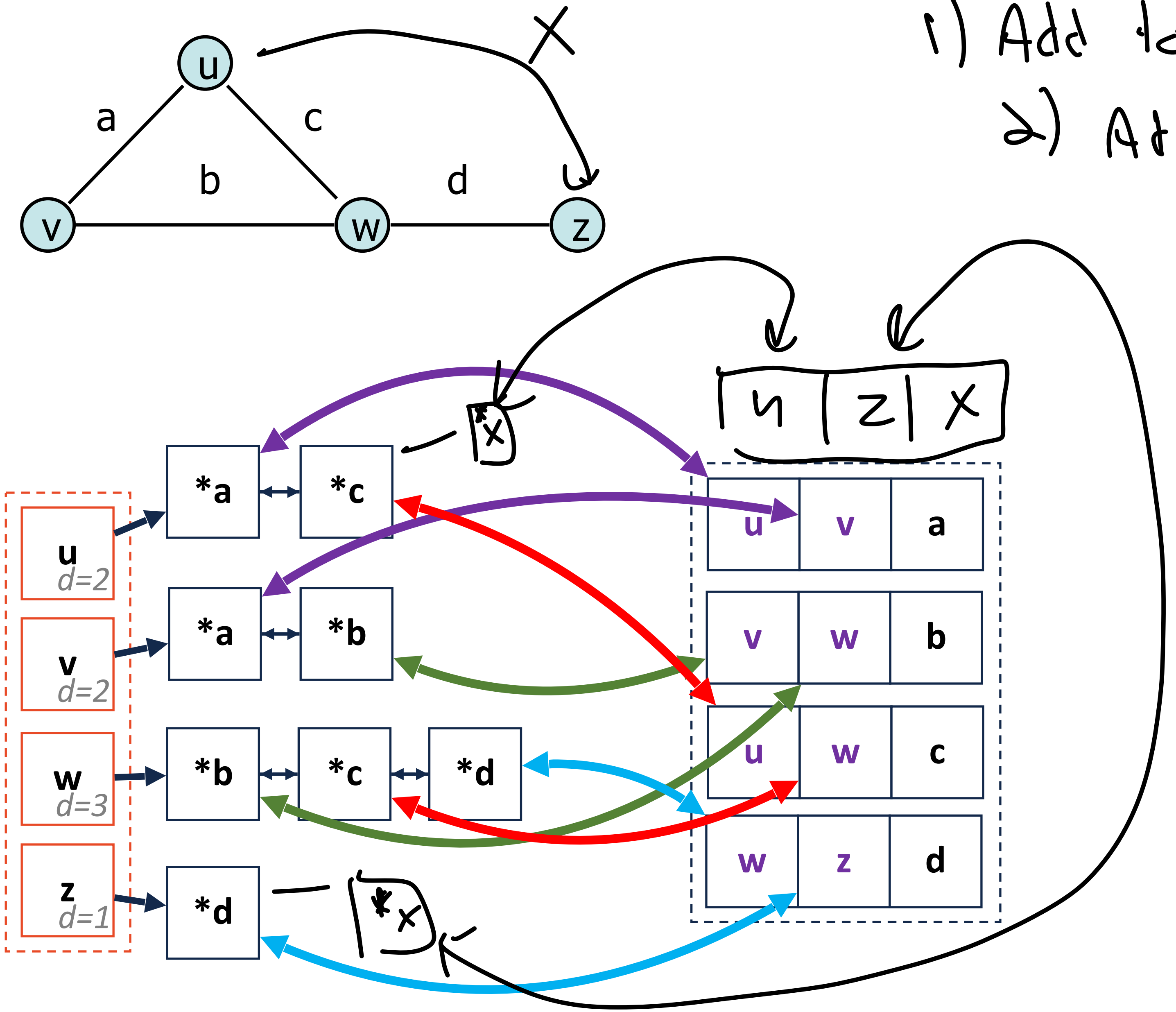
Search for other vertex across list



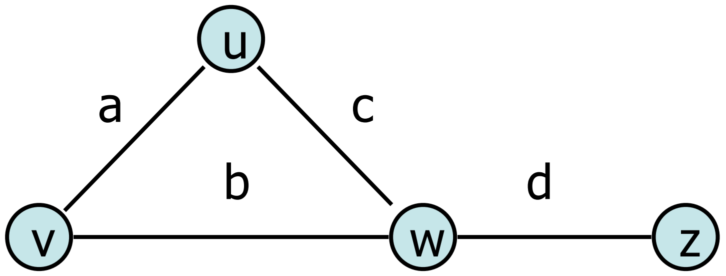
Adjacency List

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

1) Add to edge list
2) Add to both adj list vertices



Adjacency List



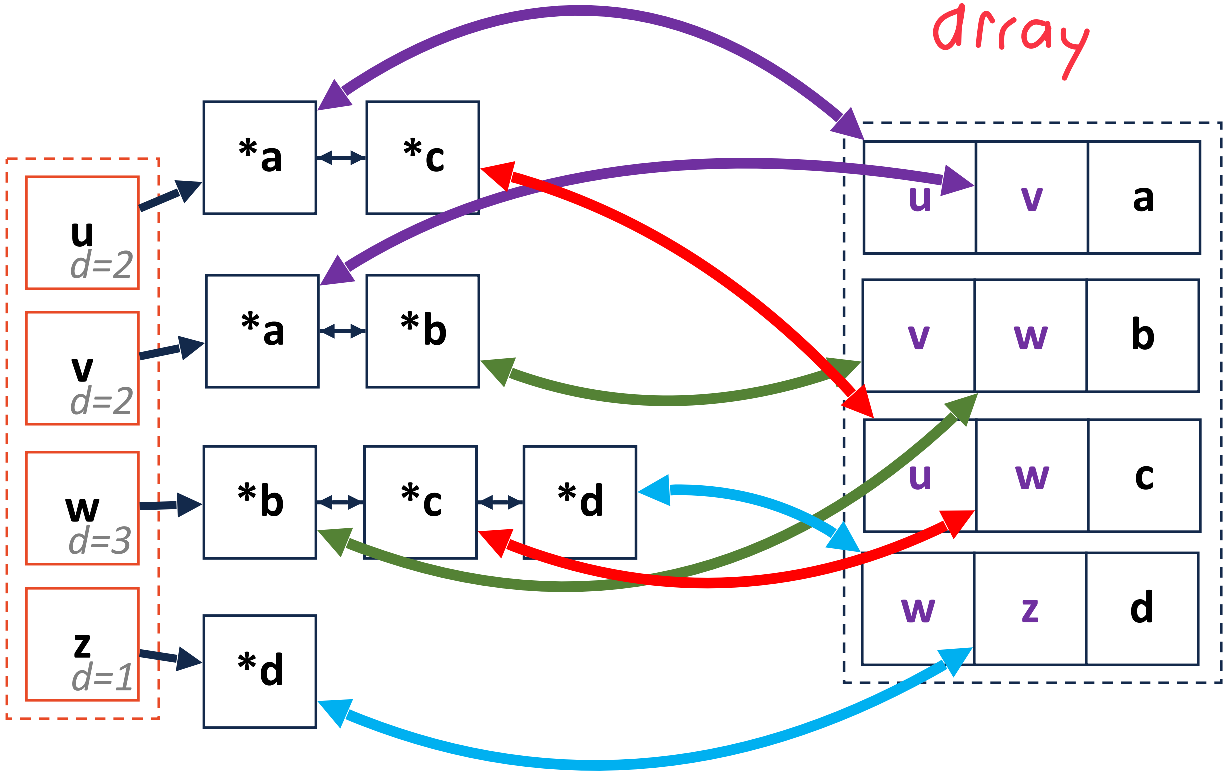
insertEdge(Vertex v1, Vertex v2, K key):

Add edge to edge list $O(1)^*$

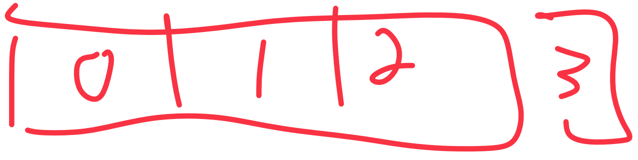
Add node to each vertex list $O(1)$

Connect all three with pointers $O(1)$

array



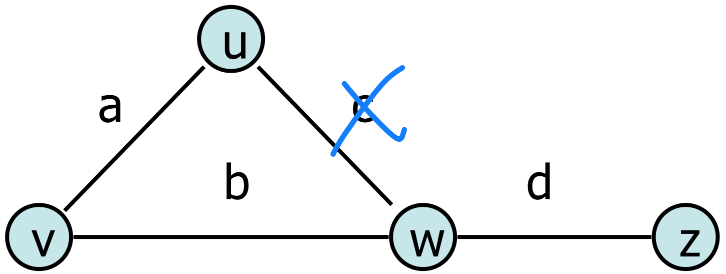
$O(1)^*$

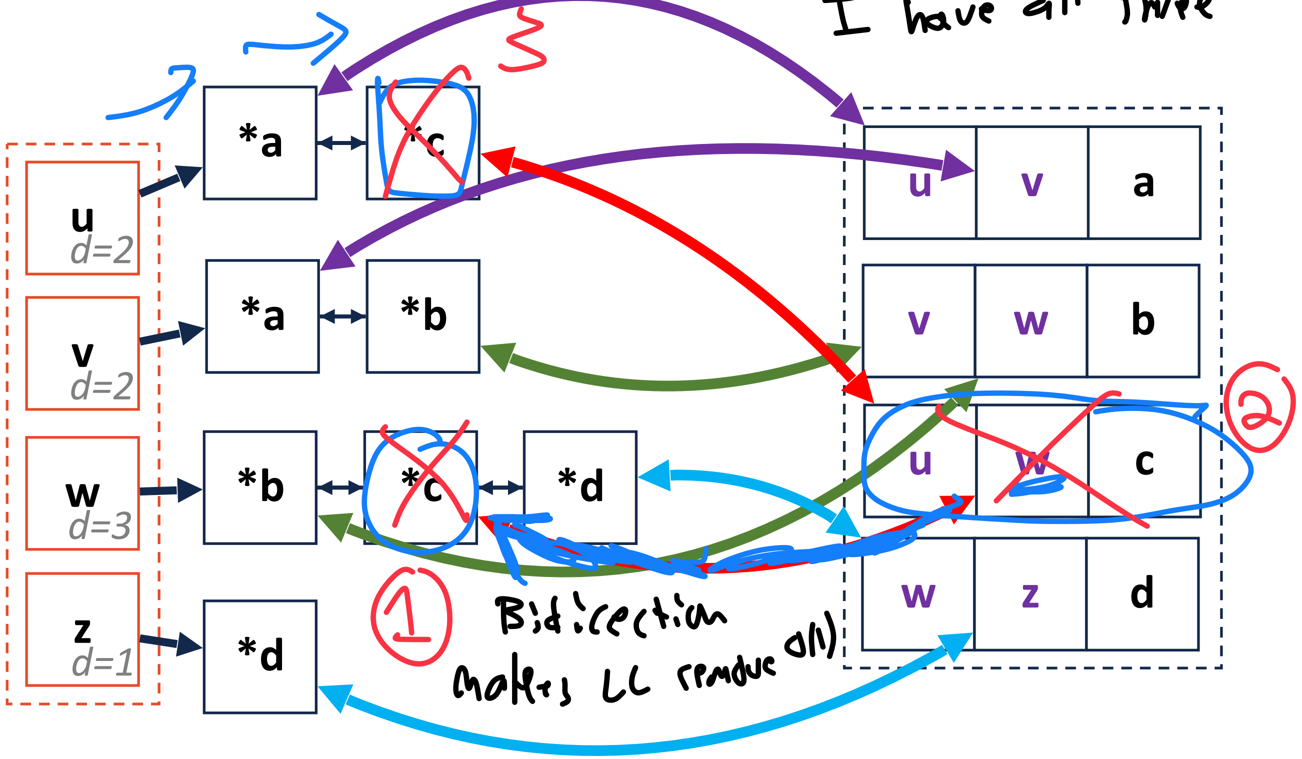


$O(1)$ most time

$O(n)$ every n

Adjacency List





Bidirectional arrows means if I have one I have all three

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

u w c

1) Search min list for vertex $O(\lceil deg(v) \rceil)$

2) Find all three locations

v1 ← min vertex degree

v2 $O(1)$

edge $O(1)$

3) Delete in order

1) v2 $O(1)$

2) edge $O(1)$

3) v1 $O(1)$

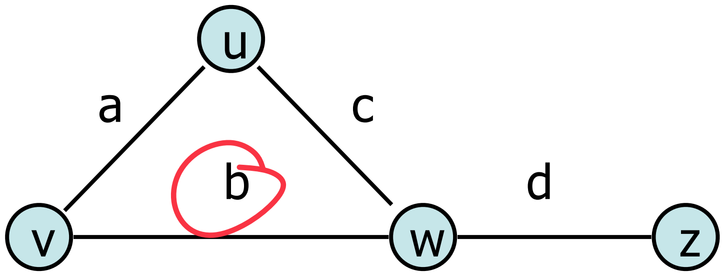
v1 → edge → v2

$O(\lceil deg(v) \rceil)$!!

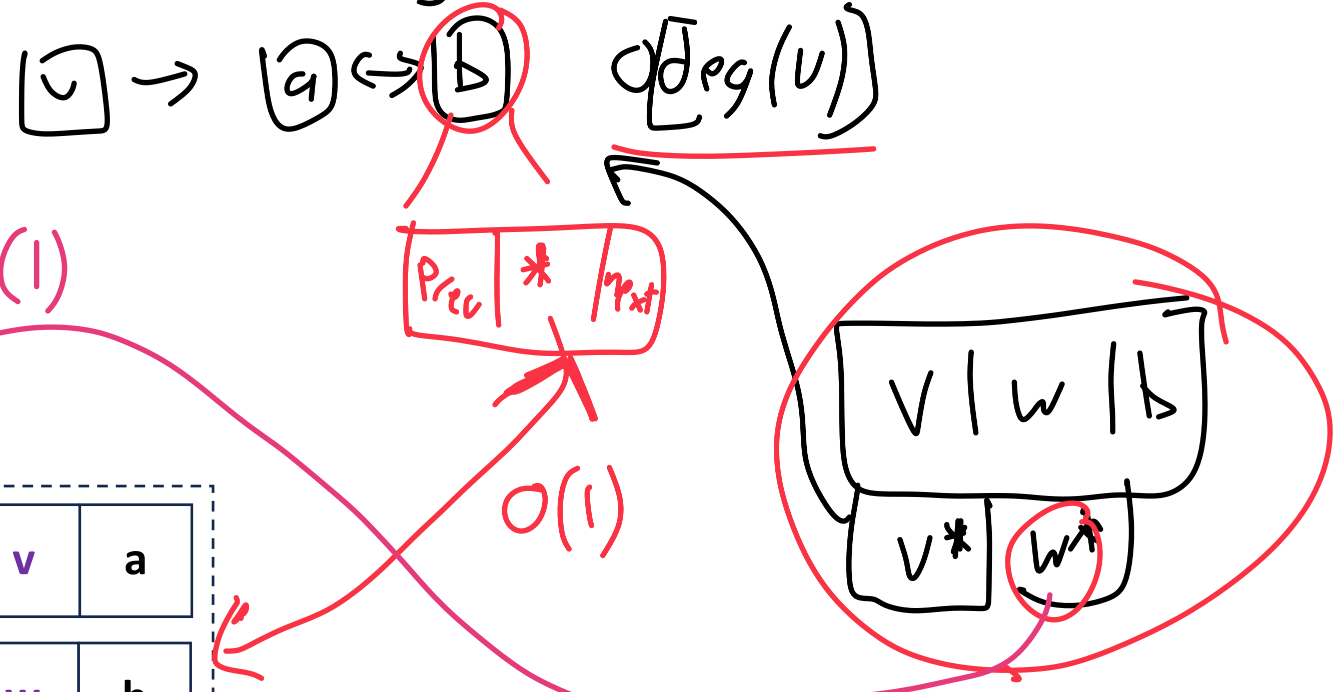
Adjacency List

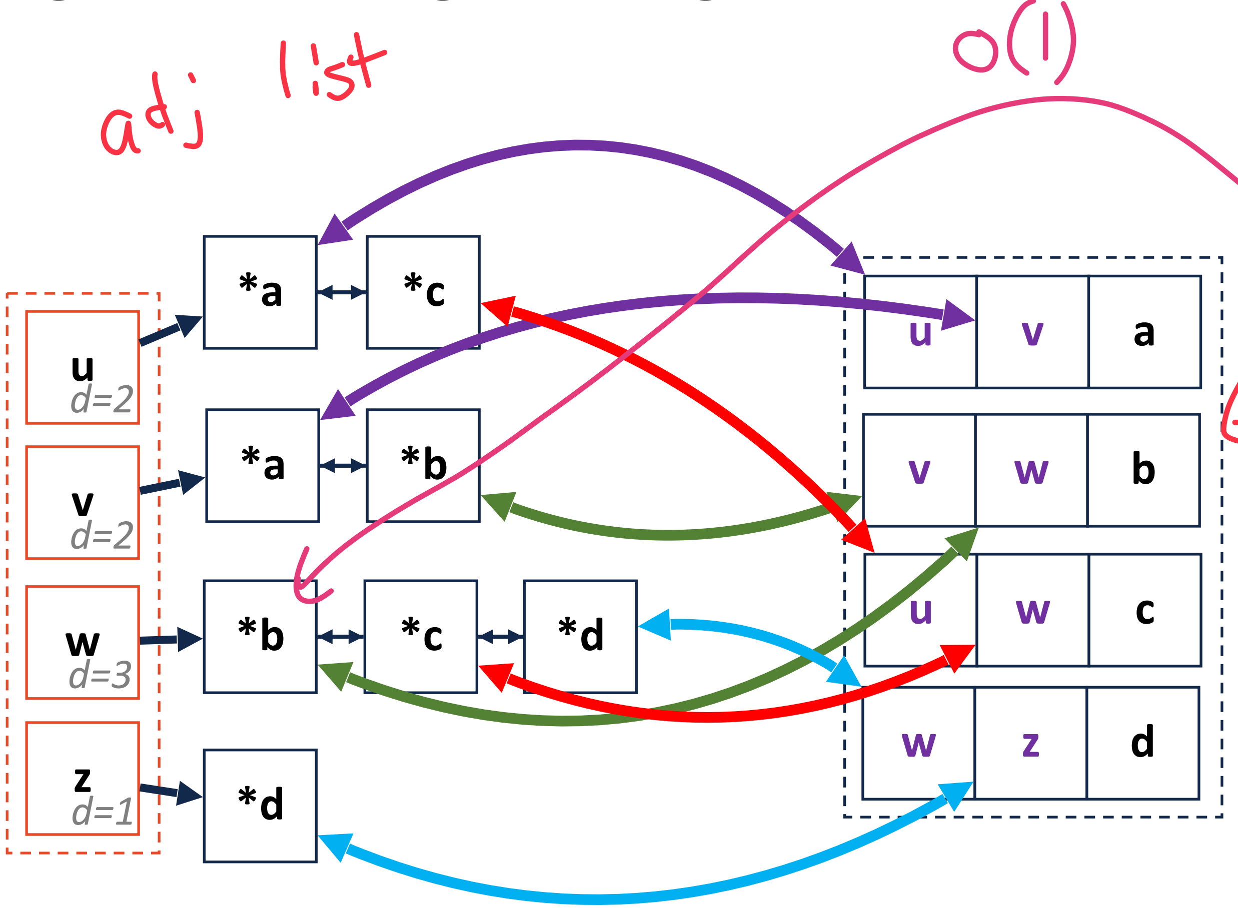
removeEdge(Vertex v1, Vertex v2, K key):

v w



Search min-degree vertex list



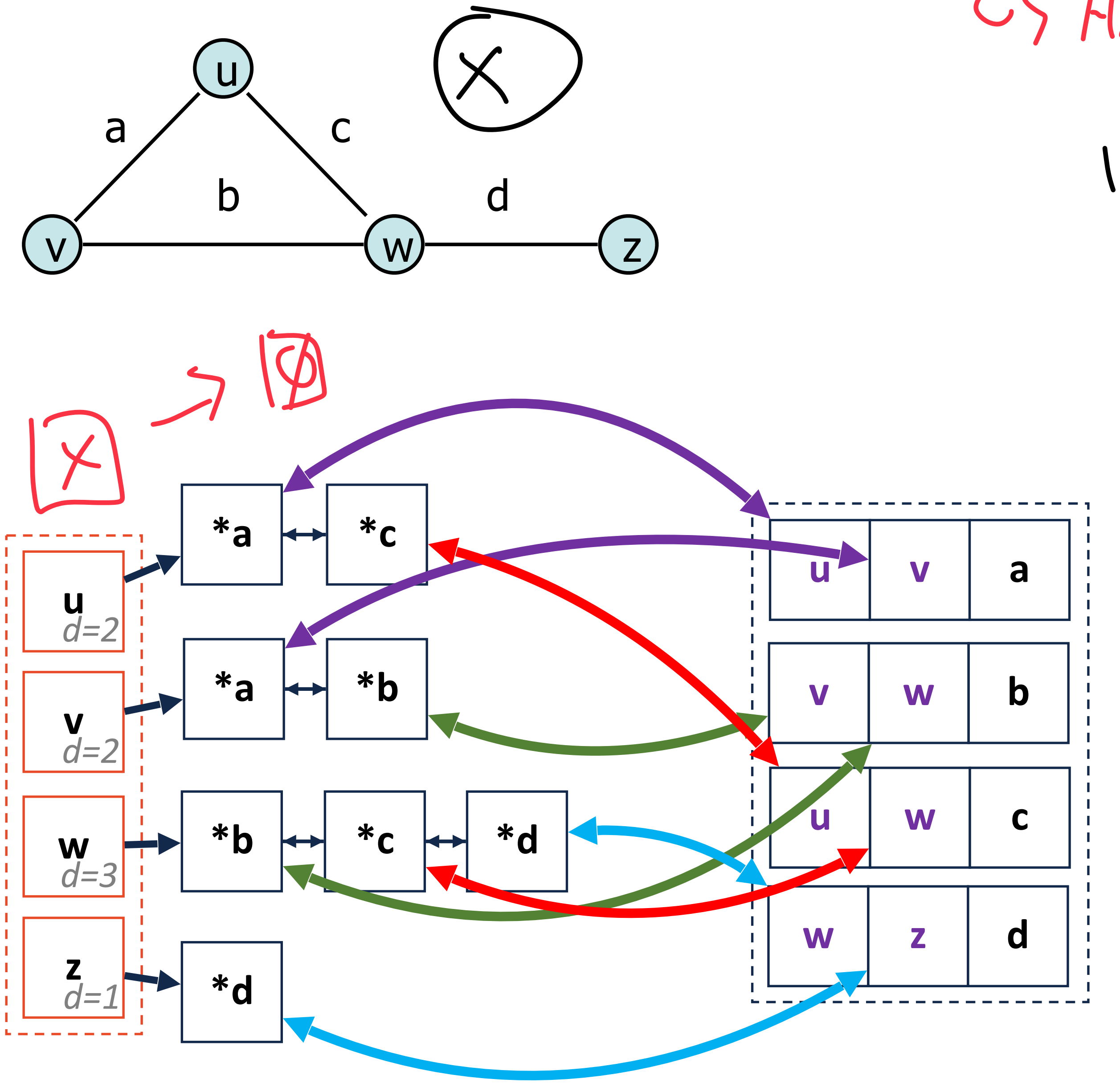


Remove mirrored entry using pointers

Remove edge from edge list O(1) delete

Remove entry from vertex list

Adjacency List



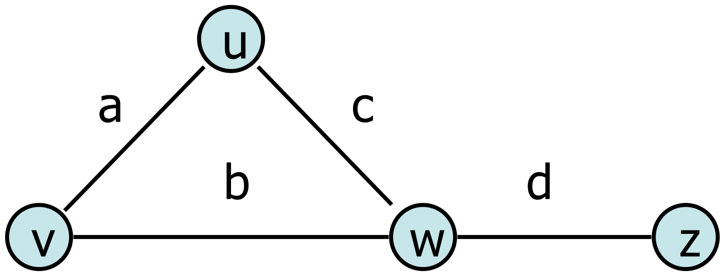
insertVertex(K key):

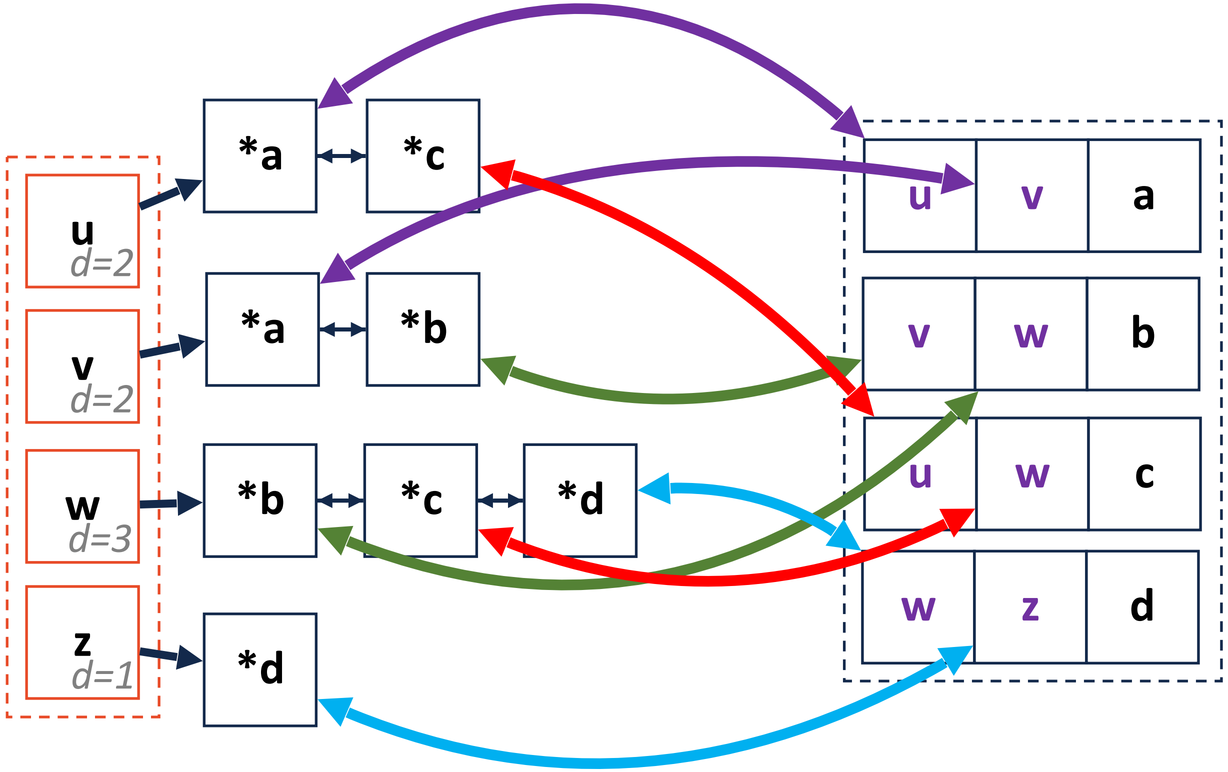
↳ Hash table

1) Add new entry to hash table w/ empty linked list

$O(1)$

Adjacency List

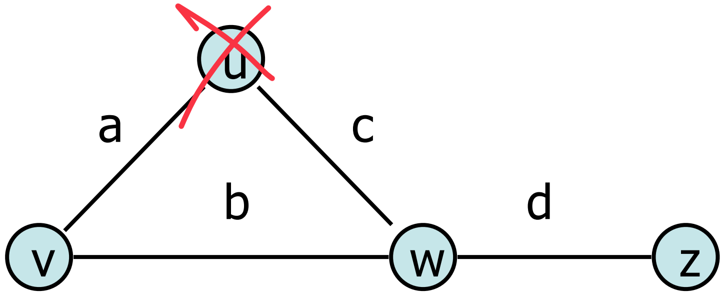


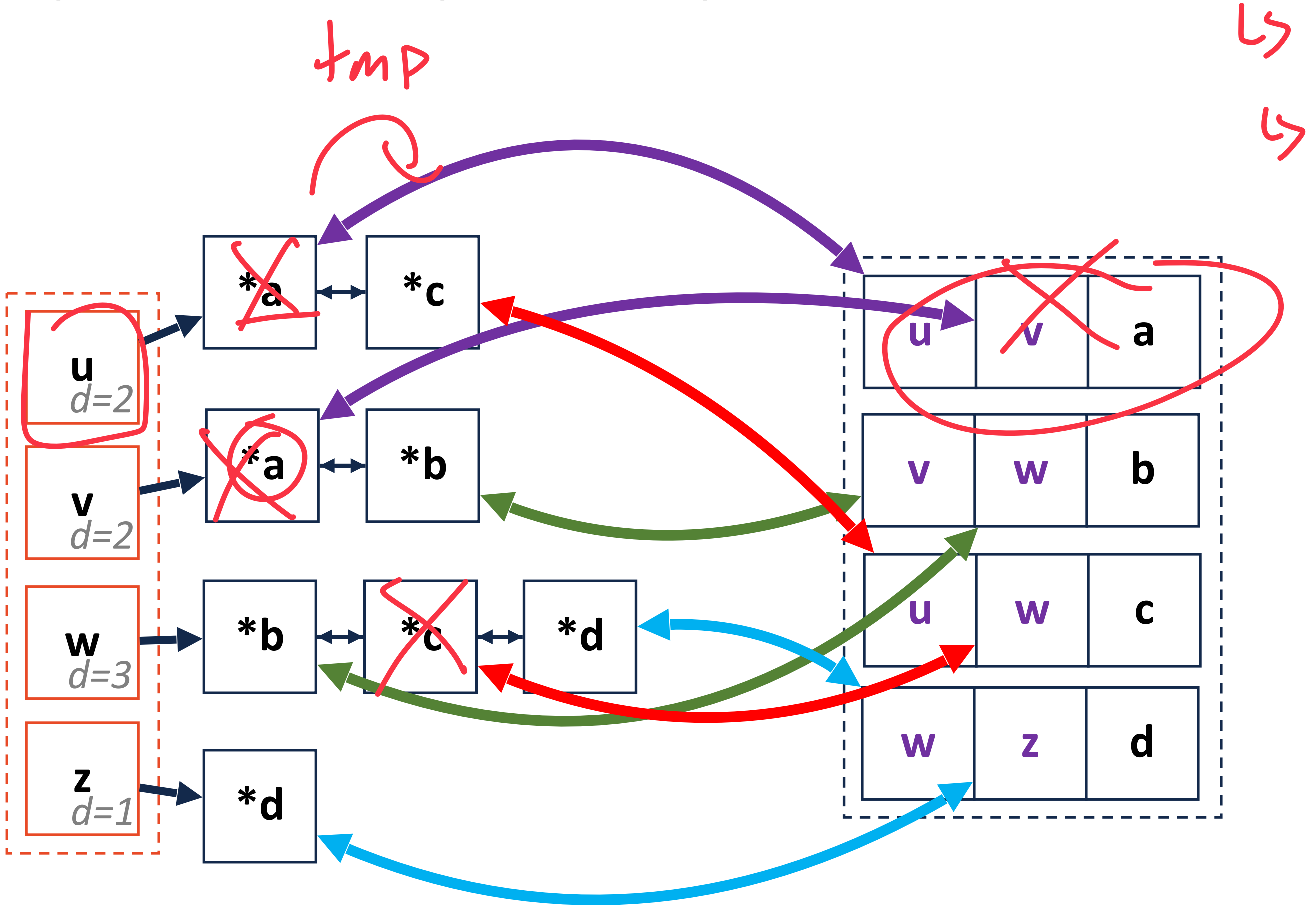


insertVertex(K key):

Add new empty list to vertex ~~list~~. hash table

Adjacency List





removeVertex(Vertex v):

1) Find v O(1)

2) walk across & delete every part of edge

only work

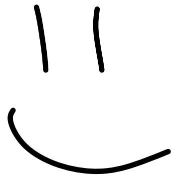
↳ v1 pointer O(1)

↳ Edge O(1)

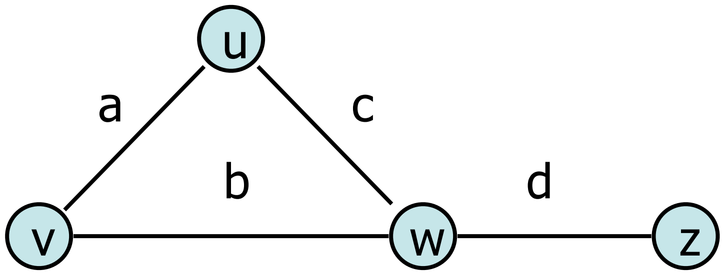
↳ v2 pointer O(1)

O(1)

O[deg (v)]



Adjacency List



removeVertex(Vertex v):

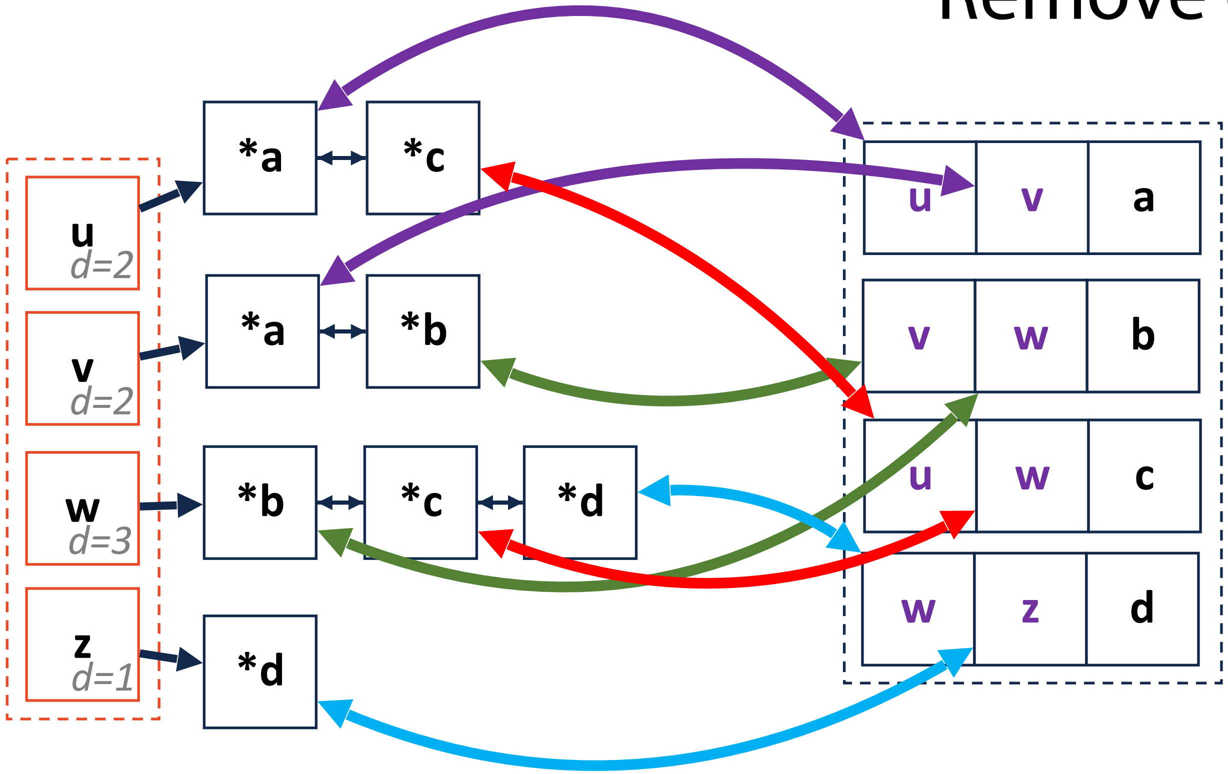
Look up vertex in vertex list

Walk across list

Remove mirrored entry using pointers

Remove edge from edge list

Remove entry from vertex list



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

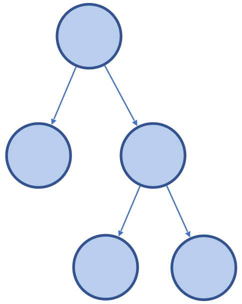
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

Graph Traversals

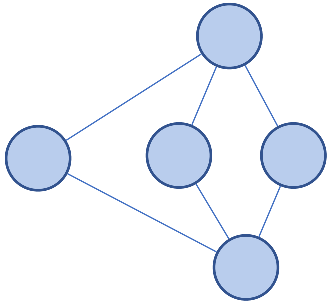
There is no clear order in a graph (even less than a tree!)

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

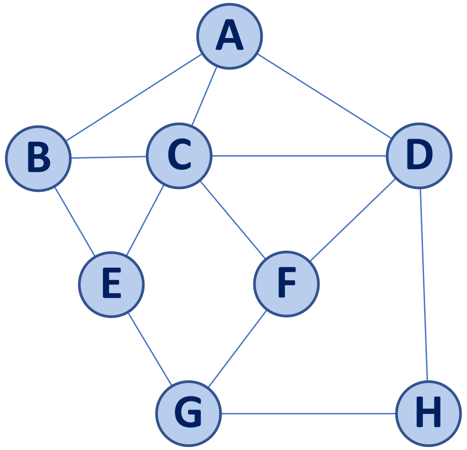
Tree traversals won’t work — lets compare:



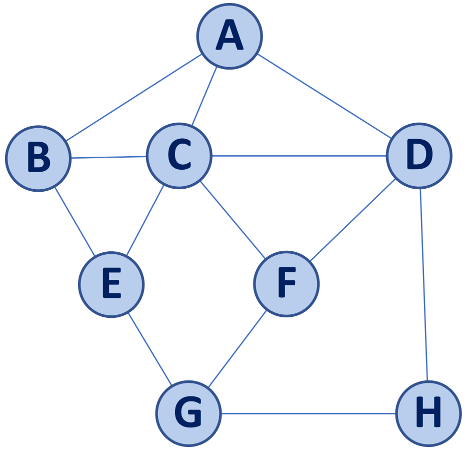
- Rooted
- Acyclic



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