Algorithms and Data Structures for Data Science

Graph Implementations 3

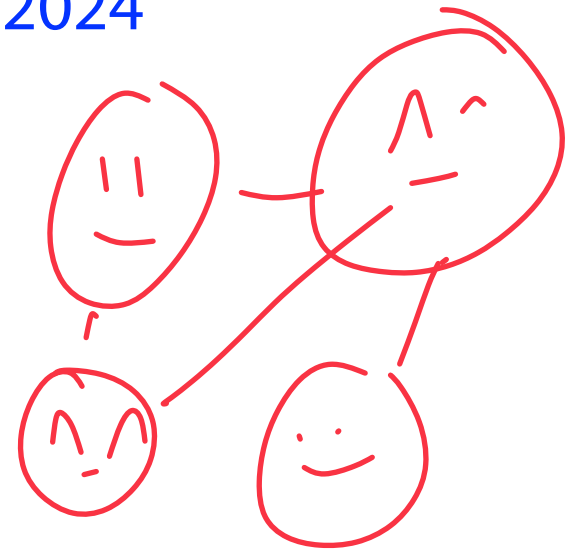
CS 277

April 1, 2024

Brad Solomon

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science



Learning Objectives

Practice implementing complex data structures (graphs)

Compare and contrast different implementations

Review Big O concepts in the context of graphs

Graph ADT

Find

getVertices() — return the list of vertices in a graph

getEdges(v) — return the list of edges that touch the vertex v

areAdjacent(u, v) — returns a bool based on if an edge from u to v exists

Insert

insertVertex(v) — adds a vertex to the graph

insertEdge(u, v) — adds an edge to the graph

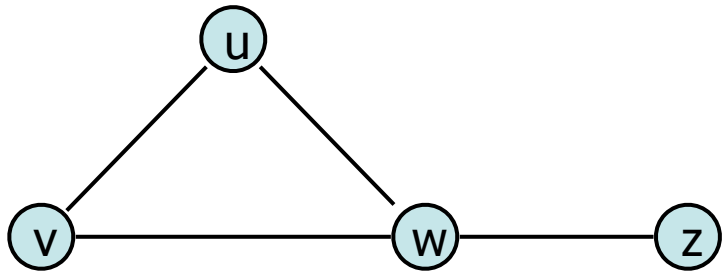
Remove

removeVertex(v) — removes a vertex from the graph

removeEdge(u, v) — removes an edge from the graph

Graph Implementation: Edge List

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



U
V
W
Z

U	V
U	W
V	W
W	Z

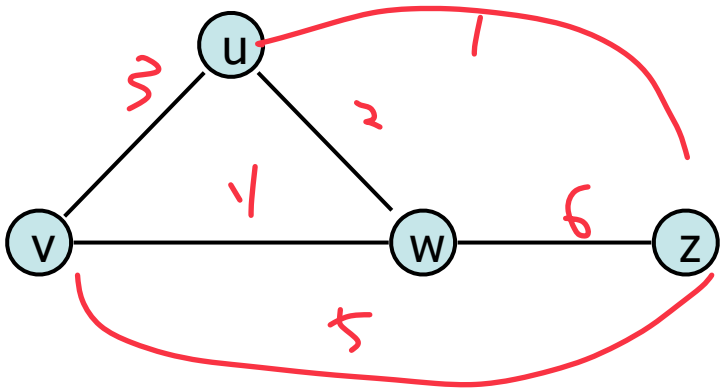
Very good at inserting new items

cost paid at lookup

Expressed as O(f)	Edge List
Space	n+m
insertVertex(v)	1*
removeVertex(v)	n+m
insertEdge(u, v)	1*
removeEdge(u, v)	m
getEdges(v)	m
areAdjacent(u, v)	m

Graph Implementation: Adjacency Matrix

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



U	0
V	1
W	2
Z	3

	u	v	w	z
u	0	1	1	0
v	1	0	1	0
w	1	1	0	1
z	0	0	1	0

O(1)

O(n)

n and m

if sparse graph
↳ $m < n^2$
↳ a lot of 0s

if dense graph
↳ $m \approx n^2$

edge changes

edge lookup

good at edge lookup

Expressed as O(f)	Adjacency Matrix
Space	n^2
insertVertex(v)	n*
removeVertex(v)	n*
insertEdge(u, v)	1
removeEdge(u, v)	1
getEdges(v)	n
areAdjacent(u, v)	1

Graph Implementations

O(m+n) ← O(m)*

I want a graph that uses the least amount of memory possible

↳ Edge List is how most graphs are stored!

I want a graph that has the fastest lookup for specific edges

↳ Adjacency matrix O(1)

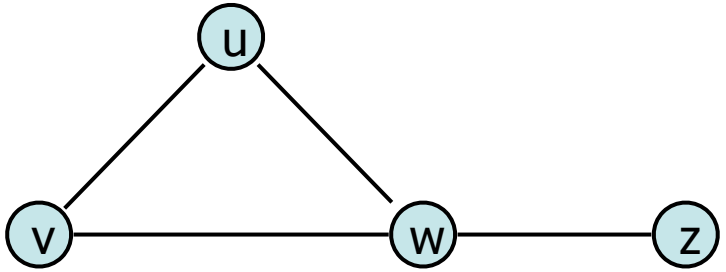
I want a graph that is efficient for a sparse dataset

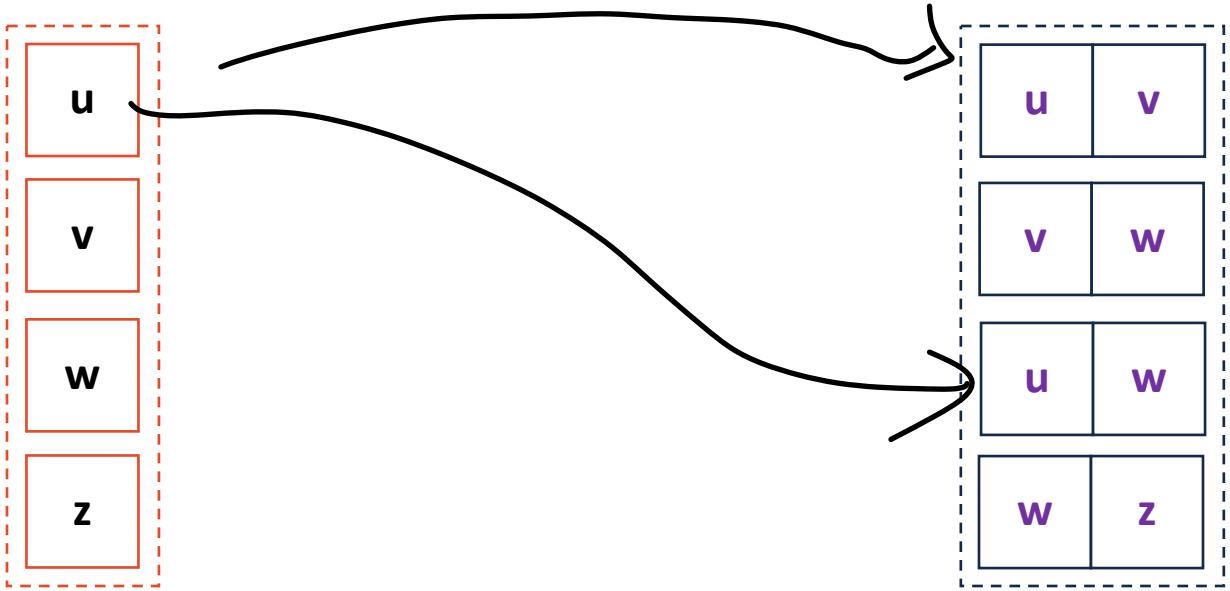
Graph Implementation: Edge List + ?

specific problem

|V|= n, |E|= m

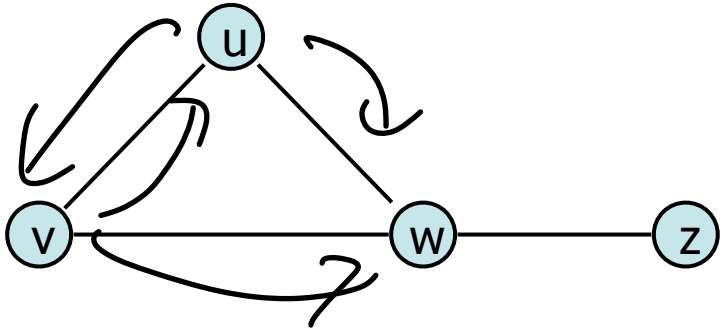
We want faster look up for all edges connecting to vertex u

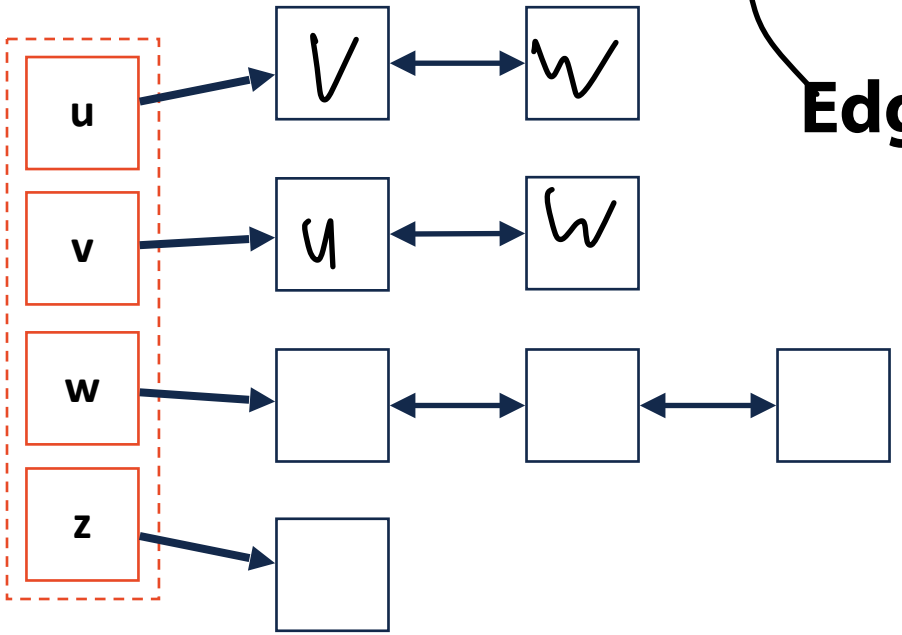




Adjacency List

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





Vertex Storage: Dictionary

↳ Keys are vertex labels

↳ Values are lists

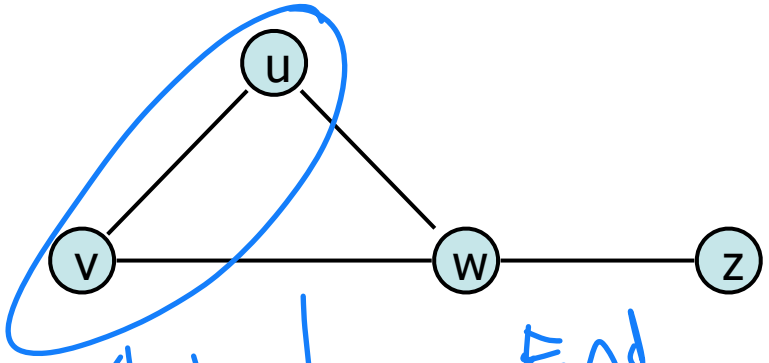
Bidirectional linked list

Edge Storage:

↳ are stored in the list

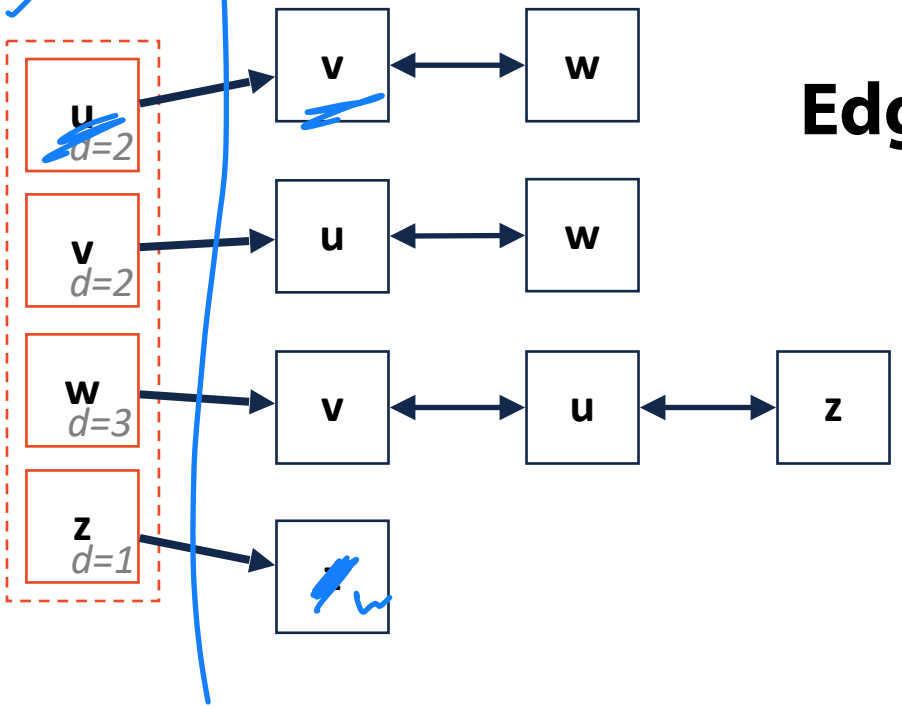
Adjacency List

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



Start

End



Dictionary [key = start vertex] = List of end vertices

Vertex Storage: Dictionary

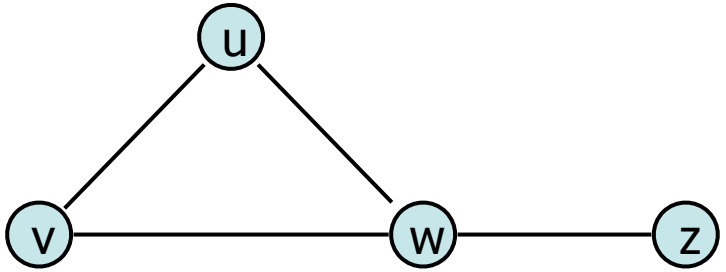
Edge Storage: are in the dictionary

↳ The lists are stored as

~~out going edges~~ end vertex

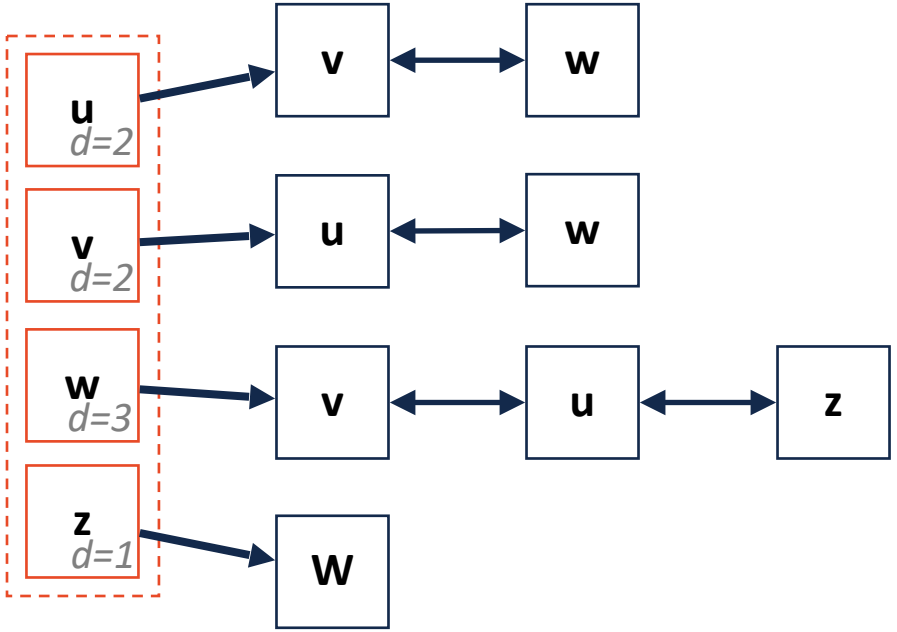
Adjacency List

|V|= n, |E|= m



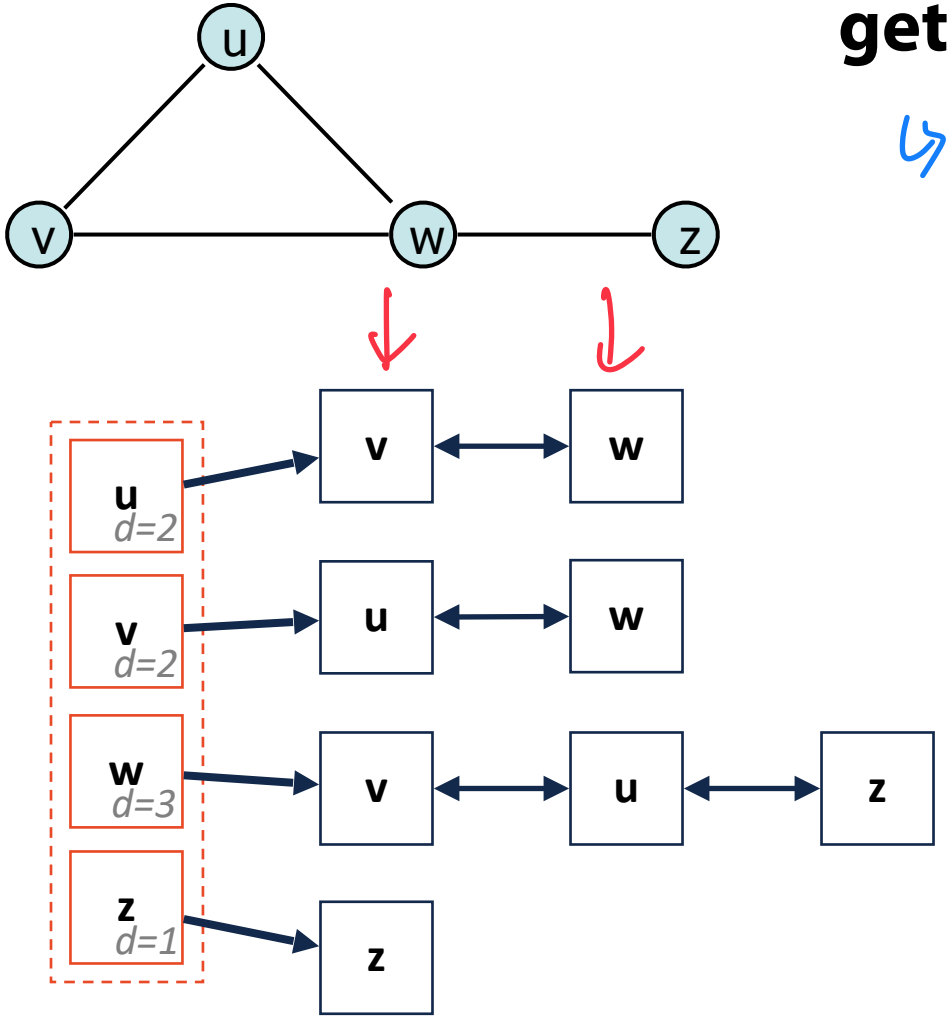
getVertices():

↳ Dictionary.keys()



Adjacency List

|V|= n, |E|= m



getEdges(v):

↳ return self.edges[v]

Dictionary

O(deg(v))

[(u,v), ...]

make (u,v) pair

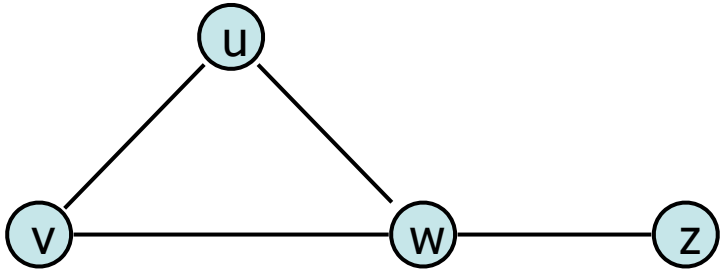
make (u,w) pair

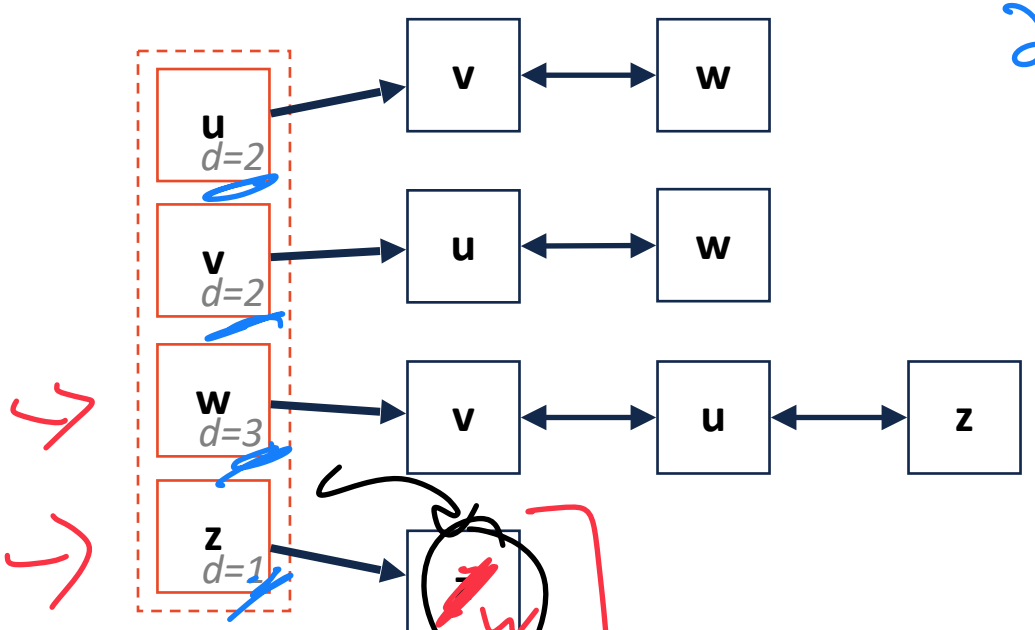
O(1)*

↳ if returning list is fine

Adjacency List

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





z has less edges than w

w, z

areAdjacent(u, v):

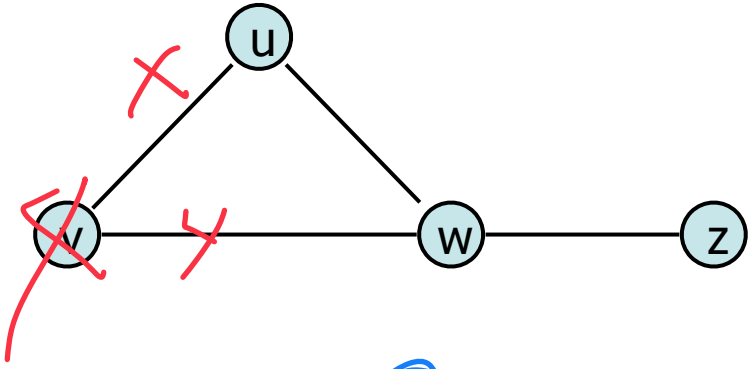
1) Do vertices exist?
   (Dictionary.keys())
2) Look up dictionary for key w/ smaller # of edges (call this v_1)
3) Find v_2 in dictionary[v_1]

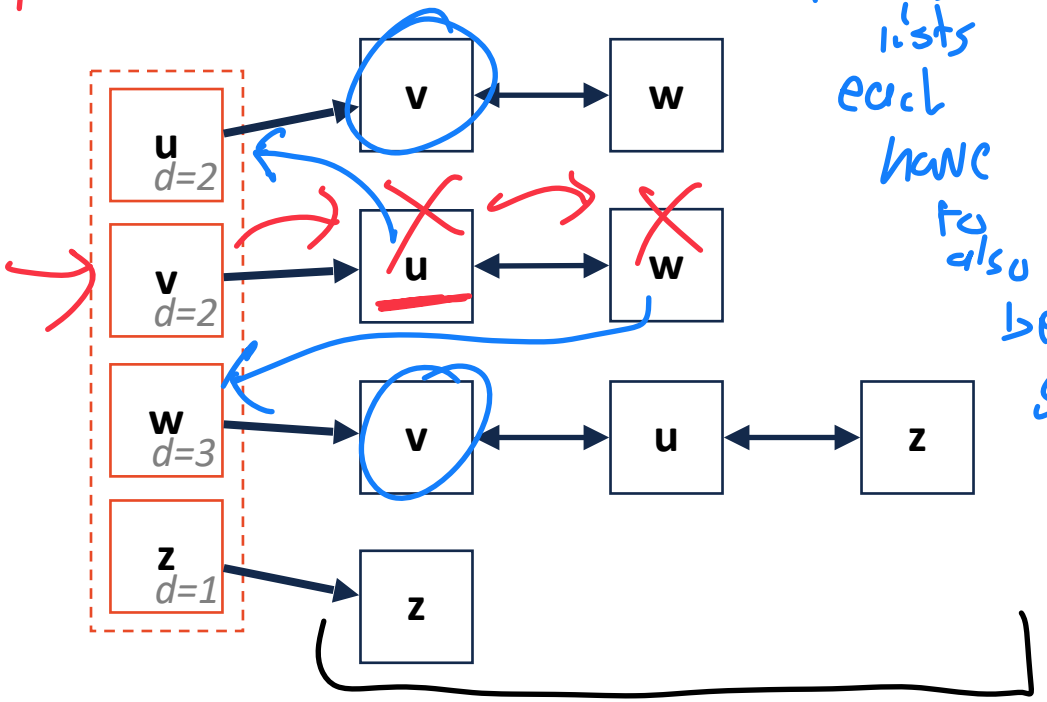
$O(n) \rightarrow O[\deg(v)]$

Dense graph — sparse

Adjacency List

|V|= n, |E|= m





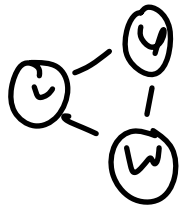
removeVertex(v):

1) From dictionary (v)

2) Find matching label in all edges

At most lists each have to also be searched

N lists

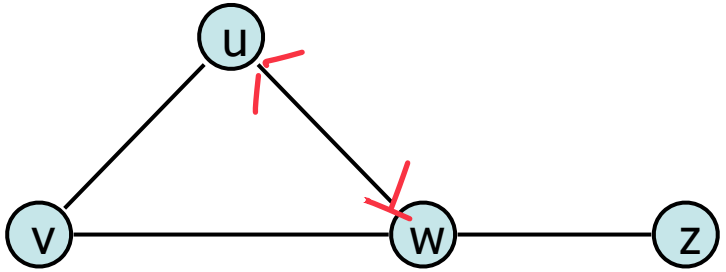


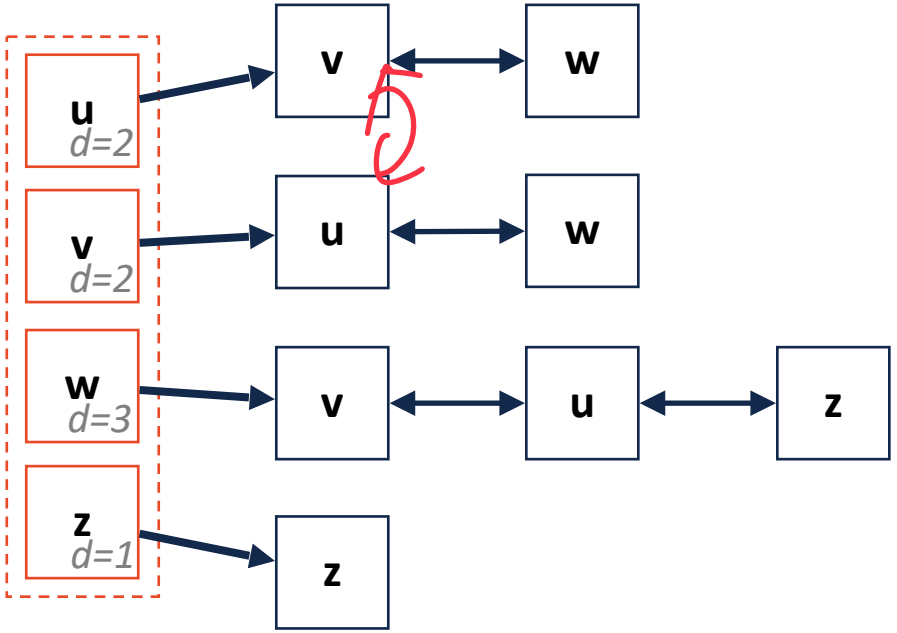
O(n²)

List of size n

Simple Adjacency List

|V|= n, |E|= m





What's wrong with our implementation?

↳ We store every edge twice

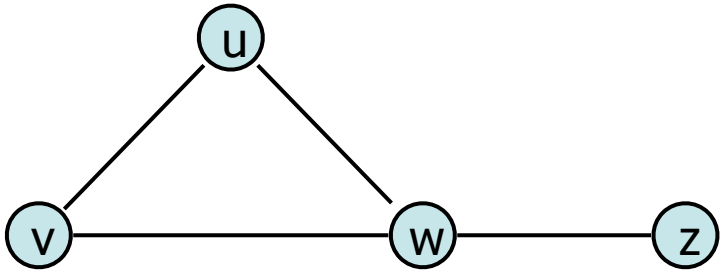
↳ But they aren't linked together!

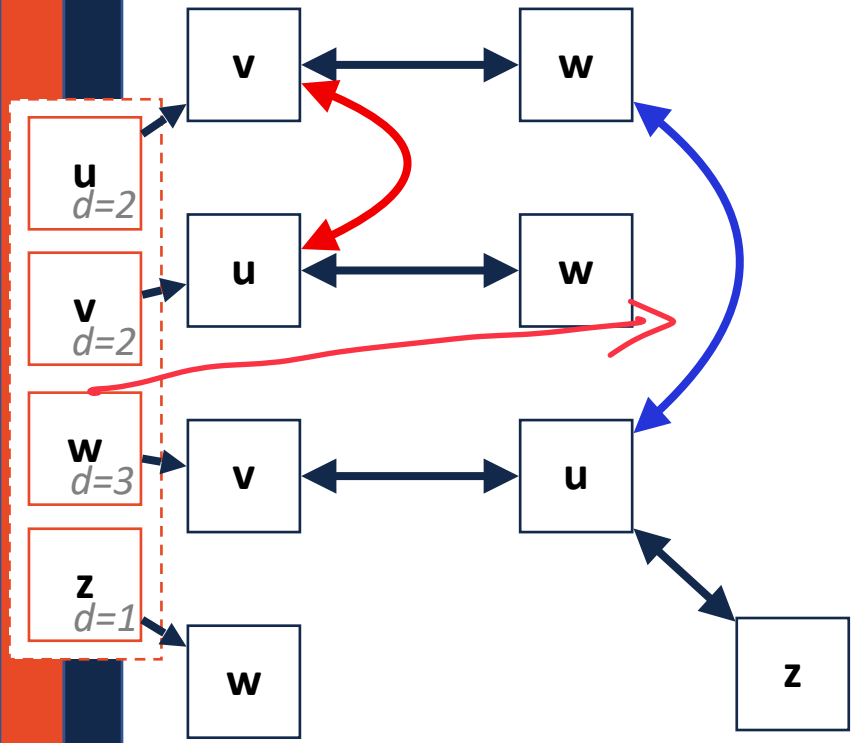
How can we fix it?

↳ Draw arrows connecting (u,v) to (v,u)

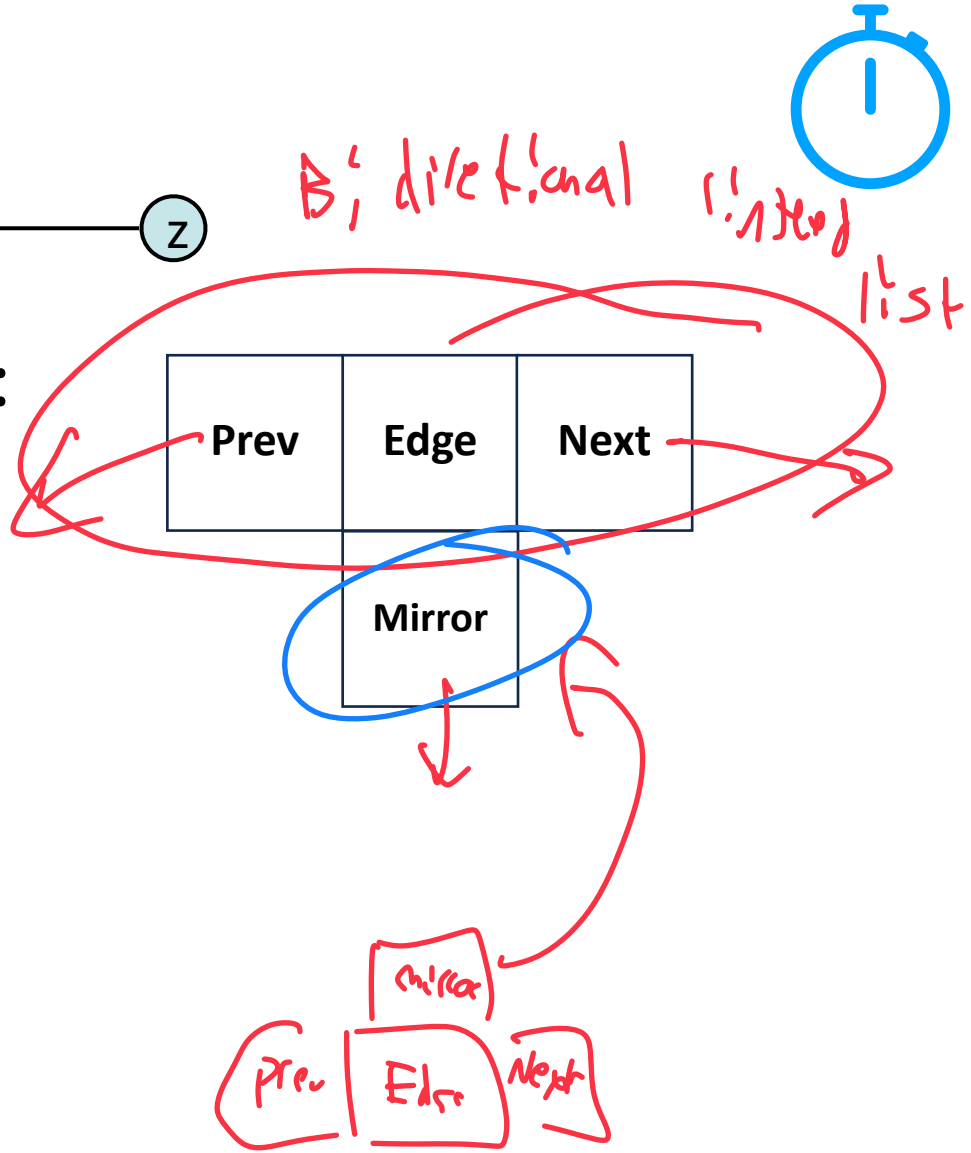
Adjacency List

|V|= n, |E|= m



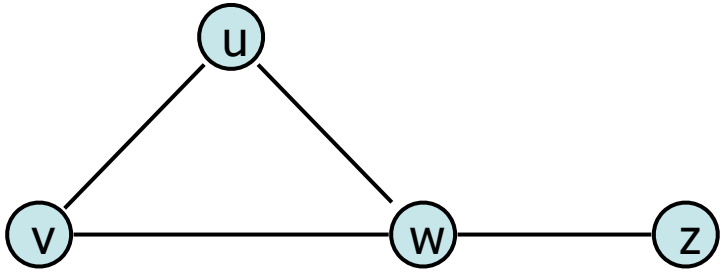


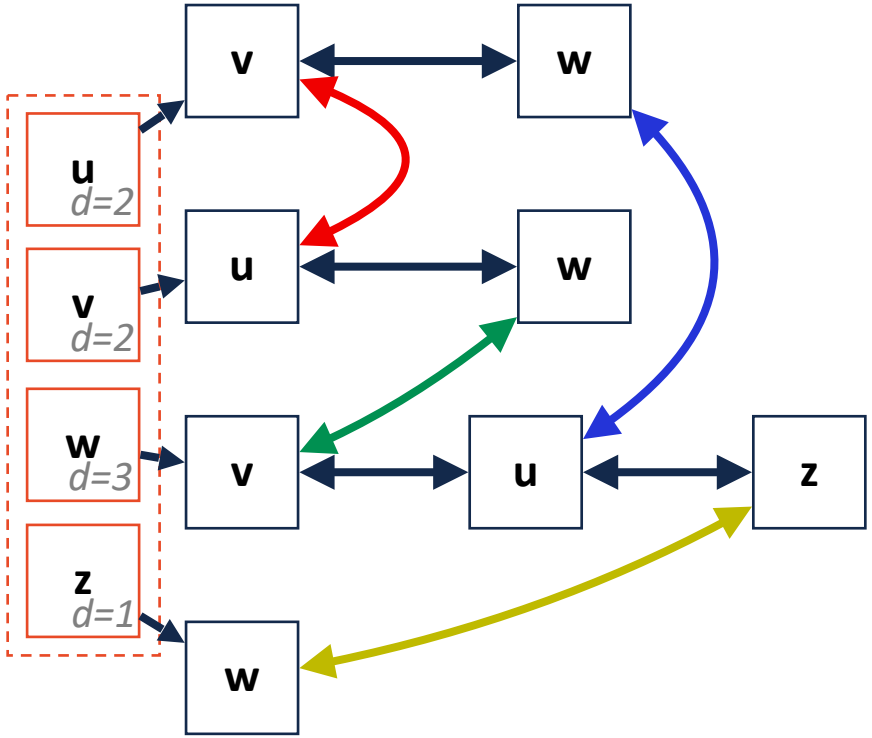
Adj List Node:



This is still a 'lie' but a more accurate one!

Adjacency List



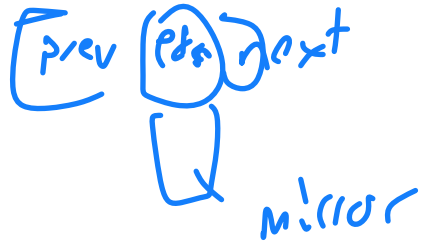


insertVertex(v):

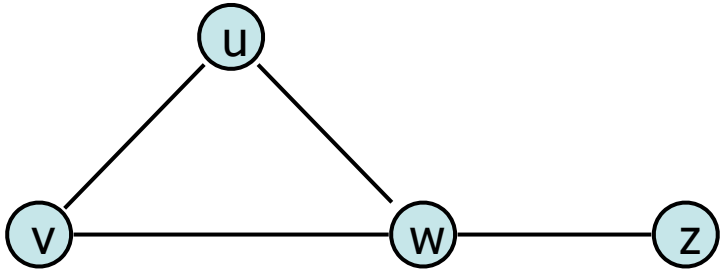
↳ self.edges [v] = L

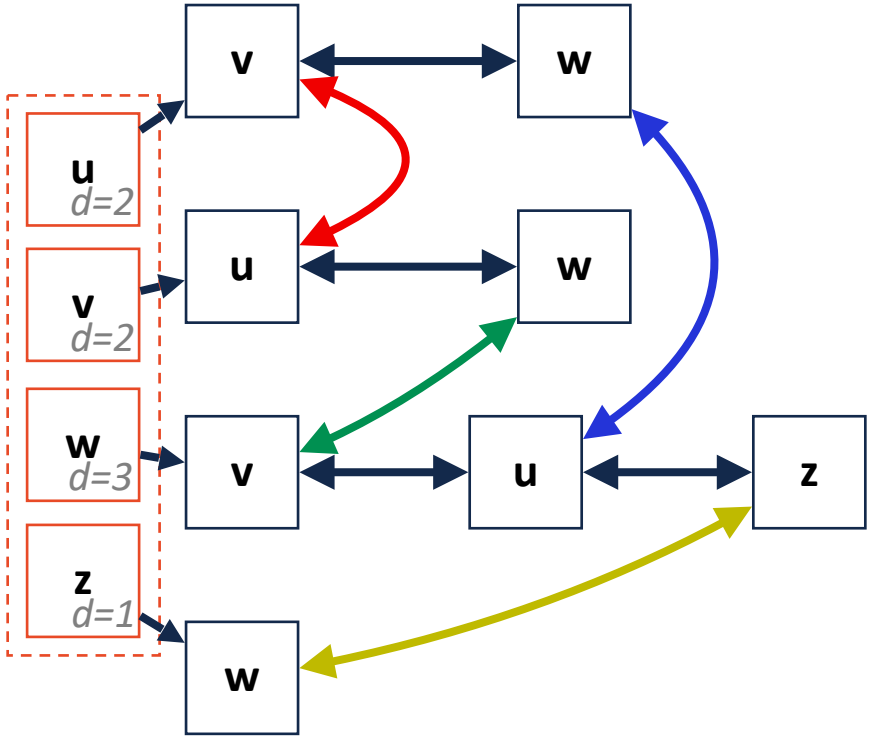
= LinkedList()

= FancyListNode()



Adjacency List





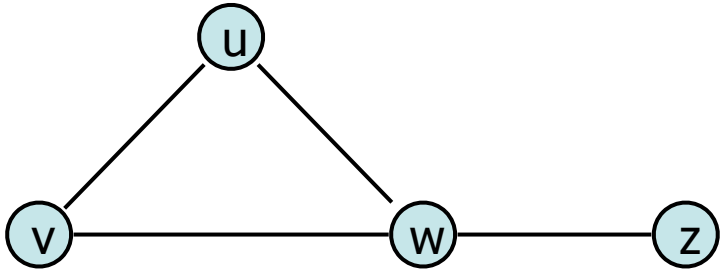
O(1)*

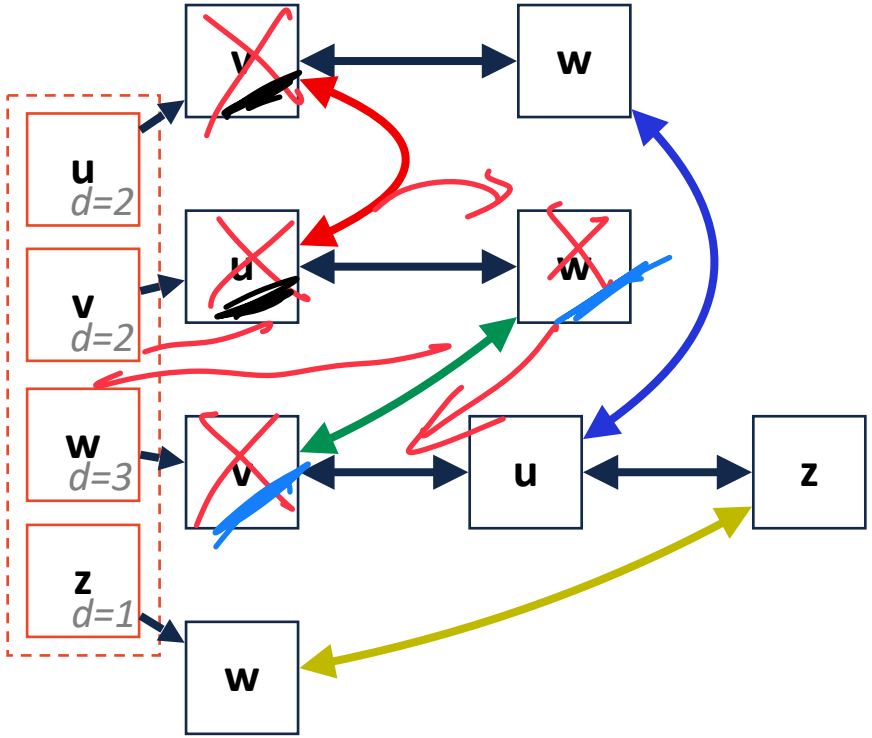
lookup is O(1)*

insertEdge(u, v):

1) If u or v is not in dictionary
   - ↳ Dictionary.keys() → list of keys
   - O(1)* ↳ add to dictionary (insertVertex)
2) Add else to dictionary twice
   - O(1)* Dictionary[u].append(v)
   - O(1)* [v].append(u)

Adjacency List





vertex

O(1)*

removeEdge(v, v):

1) Look up u or v (smaller degree)

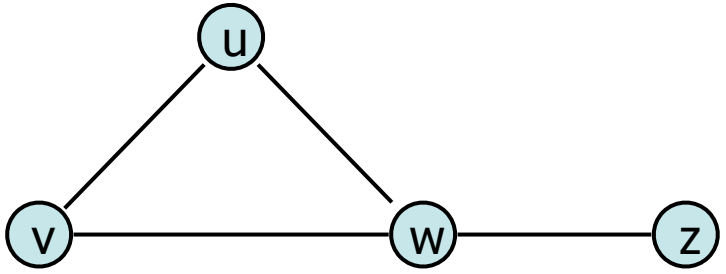
2) Walk through list deleting node & node's mirror!

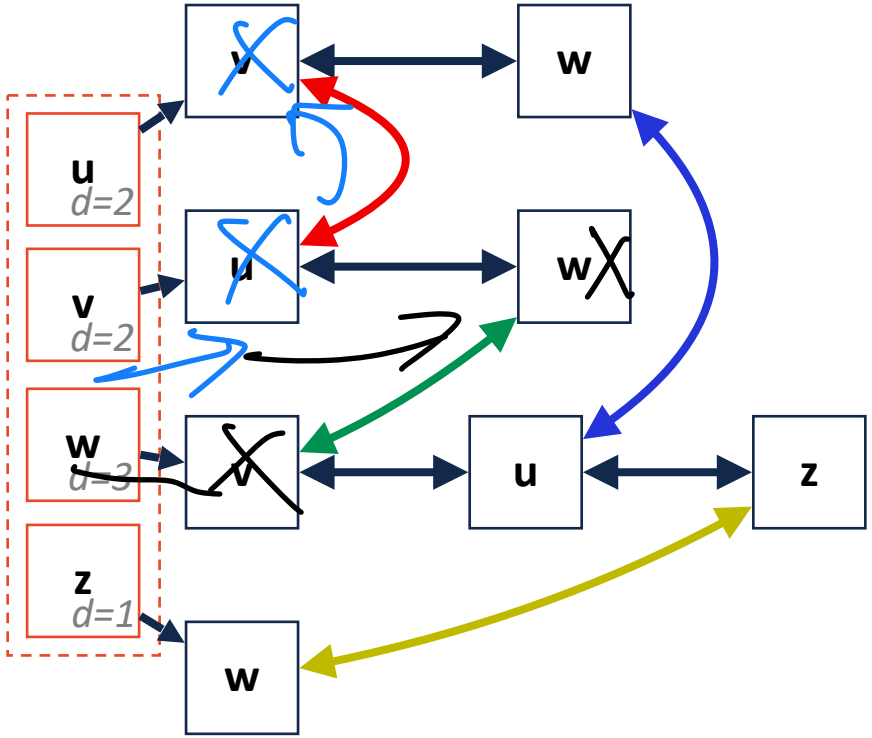
↳ O(n)

↳ every node deleted is technically 2 nodes

Overall: O(n)

Adjacency List





O(n)

removeEdge(u, v):

1) Look up ~~u~~ u and/or v

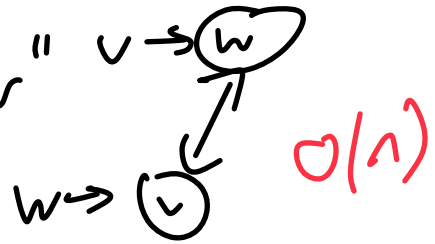
2) Remove (u,v) and (v,u)

↑ Two implementations

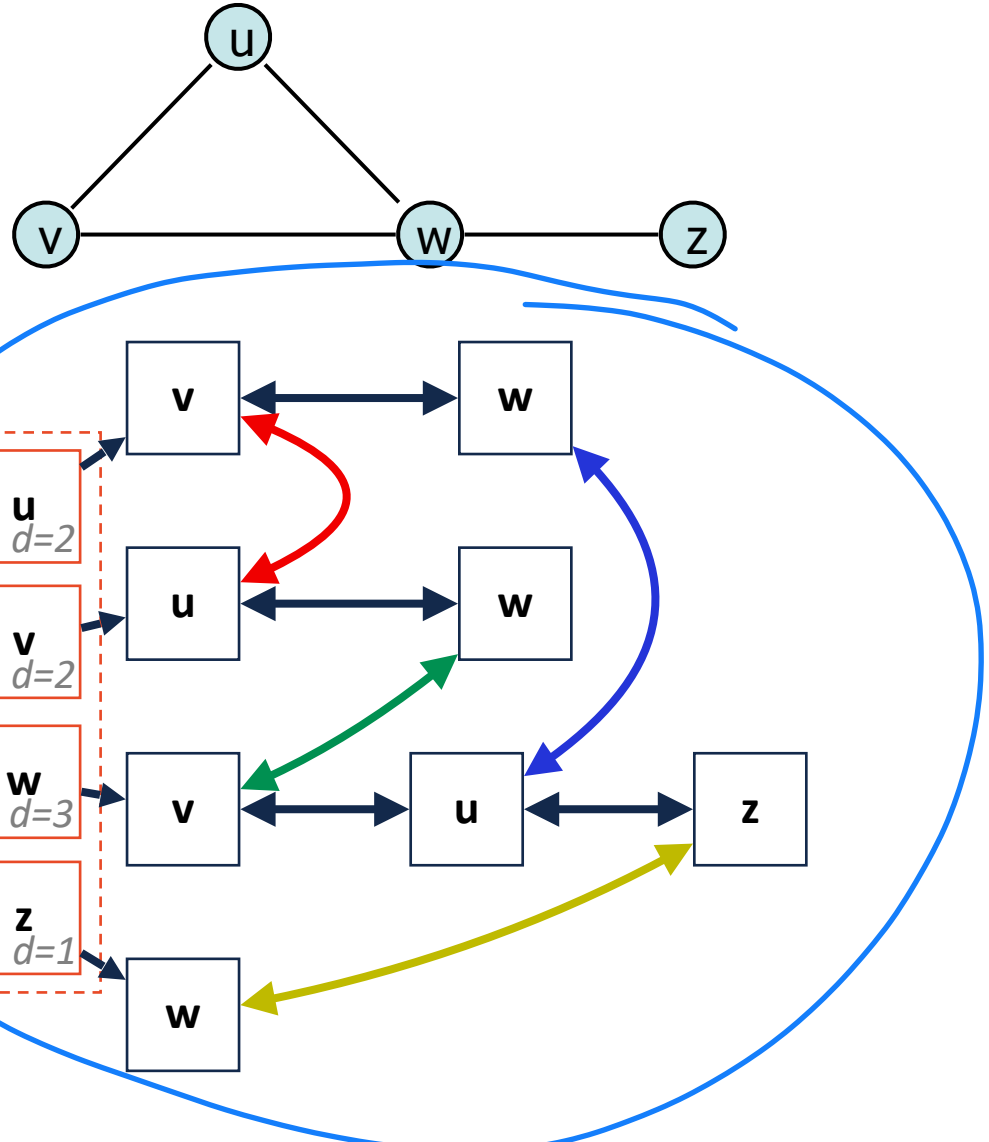
One Naive

One "Smarter"

v → w O(n)
w → v + O(n)



Adjacency List



Pros: Good at insert vertex or edge

- If sparse good at everything!

- Edge lookup – Not as good as adj matrix but... not bad

Cons:

↳ Removal of edges / vertices has to be done twice

↳ Coding it is a pain

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

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

Note: O(deg(v)) is O(n) at worst

Circle the best

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

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

Where do we go from here?

Consider: How does our implementation change for weights?
for directed edges?

Consider: How can we implement traversal on graphs?

Graph ADT

Consider: What are some common graph algorithms / uses?

Graphs in Python: NetworkX Package

NetworkX uses concepts from all three implementation methods

A graph can be built from an edge list, adjacency matrix, or adjacency list

A graph can be saved or output as any of the three formats

Many algorithms (and traversals) are built-in.

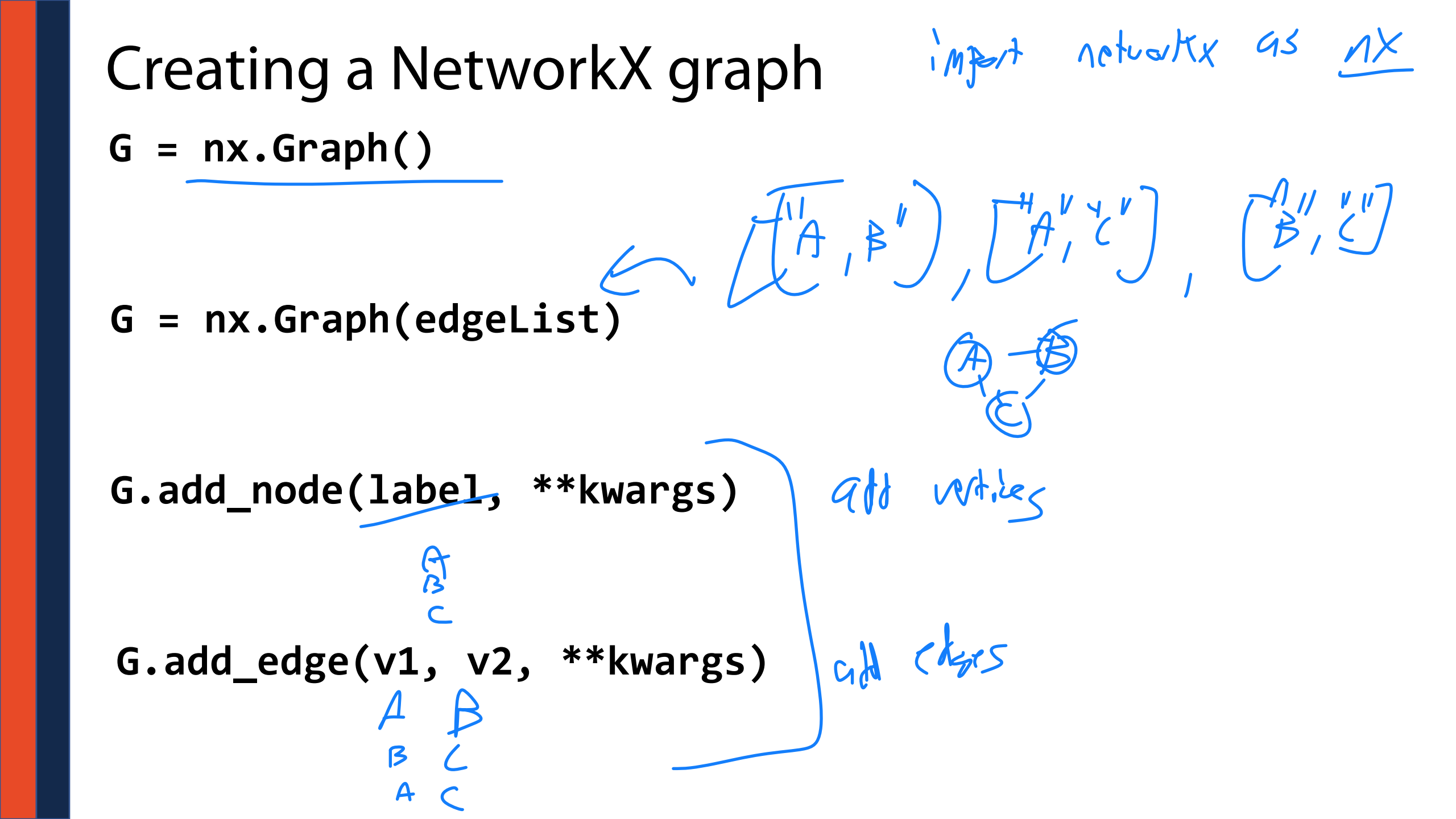
Creating a NetworkX graph

```
G = nx.Graph()
```

```
G = nx.Graph(edgeList)
```

```
G.add_node(label, **kwargs)
```

```
G.add_edge(v1, v2, **kwargs)
```



NetworkX Example

```
1  import networkx as nx
2
3  G = nx.Graph()
4
5  G.add_edge("A", "B")
6
7  G.add_edge("B", "C", weight=5)
8
9  G.add_edge("A", "C", anything="Bob", I="was", want="here")
10
11
12 print(G.nodes())
13
14
15 print(G.edges())
16
17
18 print(G.edges(data=True))
19
20
21
22
```

NetworkX Example

```
1  import networkx as nx
2
3  G = nx.Graph()
4
5  G.add_node("A")
6
7  G.add_node("B", name="Bob")
8
9  G.add_node("C", anything="Bob", I="was", want="here")
10
11 print(G.nodes())
12
13 print(G.nodes(data=True))
14
15
16
17
18
19
20
21
22
```

NetworkX Example

```
1  G=nx.random_regular_graph(3, 6)
2
3  nx.draw(G, edge_color='k', width=2, with_labels=True)
4
5  plt.show()
6
7  m = nx.adjacency_matrix(G)
8  print(m.todense())
9
10
11
12
13
14
15 adjL = nx.generate_adjlist(G)
16 for line in adjL:
17     print(line)
18
19
20
21
22
```

Graphs in Python: NetworkX Package

Networkx (and Python packages in general) can do a lot for you!

But they can sometimes make design decisions that don't work for you.

Ex: An adjacency list in NetworkX doesn't duplicate edges!