COMP_SCI 214: Data Structures and Algorithms

Graph ADTs

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 3 due next Tuesday
- Watch Dean Wes Burghardt’s academic integrity video
  - Chance for extra credit
  - Form closes tonight to be eligible
  - But also read academic integrity section in syllabus (very detailed on what’s allowed → no excuse to violate it)

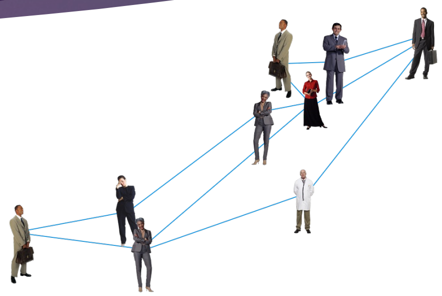
Announcements

- Exam 1 is Thursday
  - Read the instructions on the practice exam --- mostly the same instructions on the exam
  - I won't be taking clarifying questions during the exam for two big reasons:
    - It's super distracting for the students around you especially if I have to squeeze between rows and then talk
    - Students who get clarifications are at an advantage to those who didn't hear what I said
  - If you spot a genuine bug, I'll ask that you come down to talk to me so that us talking doesn't distract your neighbors.

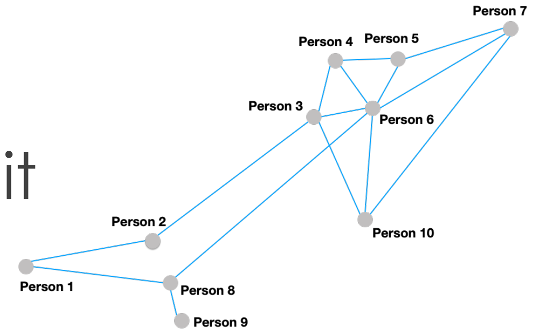
Exam tips

- Concepts are crucial but applying concepts to problems equally important
  - Your fundamentals will be strong if you're doing your best in and out of class
  - You will be able to apply your strong fundamentals to new problems
- Don't overthink but read problems **very** carefully (don't miss important details), write your thoughts/notes down, and take your time
- **Draw things out** for all problems where applicable (e.g., DSes, algorithm)
  - Draw things out to solidify your understanding of a problem and your solution
  - Detach appendix and scratch paper and use liberally
- Get a lot of rest!

Representing connections



- What if we need to represent complex connections?
  - Connections between users in a social network
  - Wikipedia link tracing (one page leads to many links)
  - The spread of a disease between users in locality



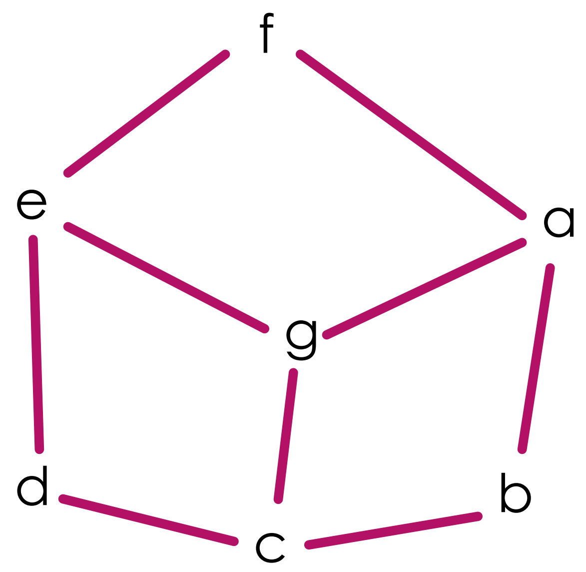
- Dictionaries, stacks, and queues aren't going to cut it

- Need another form of data structure – graphs!

Graphs review

Test your knowledge!

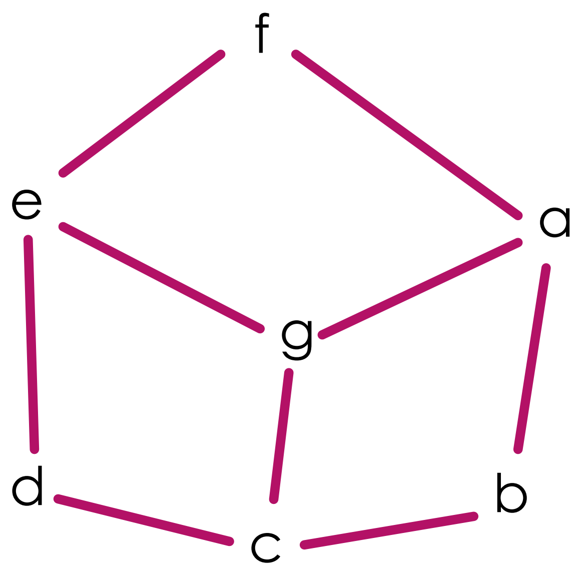
1. What is the degree of f?
   - 2



Test your knowledge!

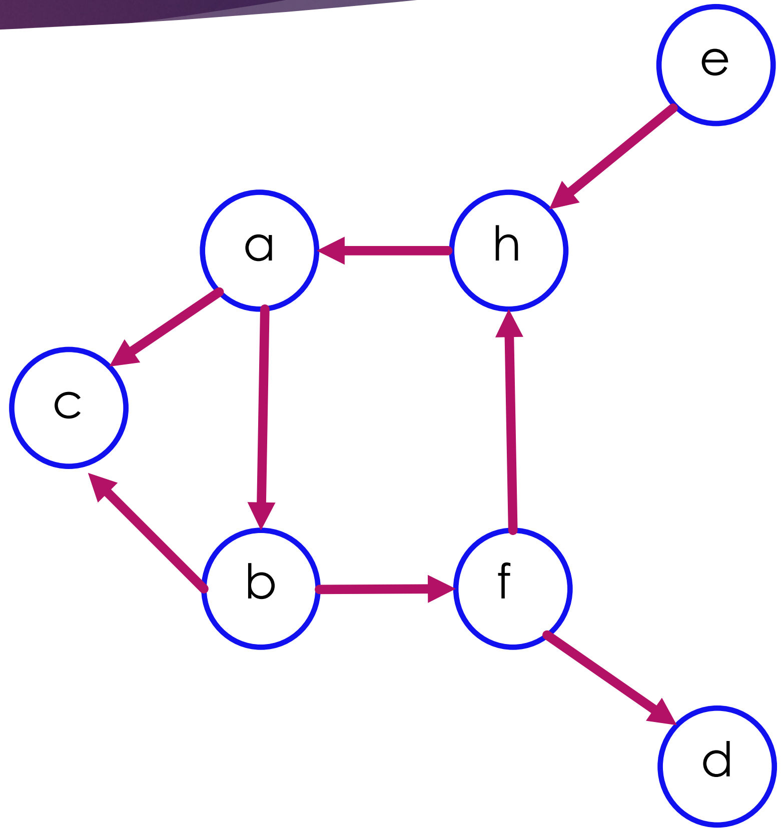
1. What is the degree of f?
   - 2

2. What is the degree of g?
   - 3



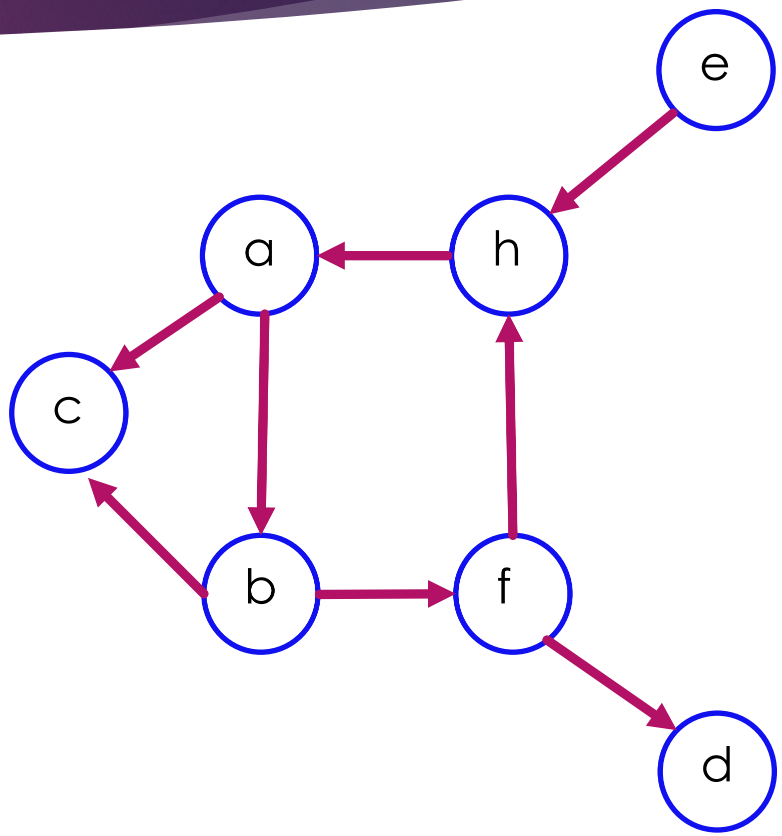
Test your knowledge!

1. Is E reachable from A?
   - No



Test your knowledge!

1. Is E reachable from A?
   - No
2. Are A and B strongly connected?
   - Yes



Open graphs Q&A

- Any questions?
  - Raise your hand
  - Ask on Piazza

Graph ADT

Different types of graphs

- Four different possible types of graphs
  1. Unweighted Undirected graph (UU)
  2. Unweighted Directed graph (UD)
  3. Weighted Undirected graph (WU) → **Homework 4**
  4. Unweighted Directed graph (WD)

- One ADT definition can't handle each of the above!
  - Each needs its own ADT with abstract values and operations

#1: Unweighted Undirected graph ADT

- Abstract values look like: (V = set of vertices, E = set of edges)
  - $V = \{0, 1, \dots, |V| - 1\}$
    - We will use natural numbers 0…n-1 as our vertices
  - One edge is a set of 2 vertices
  - $\cup E \subseteq V$

#1: UU graph ADT

- Abstract values look like: (V, E) (one edge is a set of 2 vertices)
- ADT in DSSL2

```
interface UUGRAPH:
  def add_edge(self, u: nat?, v: nat?) -> NoneC
  def has_edge?(self, u: nat?, v: nat?) -> bool?
  def get_vertices(self) -> SetC[nat?]
  def get_neighbors(self, v: nat?) -> SetC[nat?]
```

#1: UU ADT Laws

- $\{g = (V, E) \wedge n, m \in V\}$ $g.add\_edge(n, m) \Rightarrow None$ $\{g = (V, E \cup \{(n, m)\})\}$

- $\{g = (V, E) \wedge (n, m) \in E\}$ $g.has\_edge(n, m) \Rightarrow True$ $\{\}$
- $\{g = (V, E) \wedge (n, m) \notin E\}$ $g.has\_edge(n, m) \Rightarrow False$ $\{\}$

- $\{g = (V, E)\}$ $g.get\_vertices() \Rightarrow V$ $\{\}$

- $\{g = (V, E)\}$ $g.get\_neighbors(n) \Rightarrow \{m \in V : \{n, m\} \in E\}$ $\{\}$

#2: UD graph ADT

- Abstract values look like: (V, E)
- ADT in DSSL2

```
interface UDGRAPH:
  def add_edge(self, u: nat?, v: nat?) -> NoneC
  def has_edge?(self, u: nat?, v: nat?) -> bool?
  def get_vertices(self) -> SetC[nat?]
  def get_succs(self, v: nat?) -> SetC[nat?]
  def get_preds(self, v: nat?) -> SetC[nat?]
```

#2: UD ADT Laws

- $\{g = (V, E) \land n, m \in V\}$ $g.add\_edge(n, m) \Rightarrow None$ $\{g = (V, E \cup \{(n, m)\})\}$

- $\{g = (V, E) \land (n, m) \in E\}$ $g.has\_edge(n, m) \Rightarrow True$ $\{\}$
- $\{g = (V, E) \land (n, m) \notin E\}$ $g.has\_edge(n, m) \Rightarrow False$ $\{\}$

- $\{g = (V, E)\}$ $g.get\_vertices() \Rightarrow V$ $\{\}$

- $\{g = (V, E)\}$ $g.get\_succs(n) \Rightarrow \{m \in V : \{n, m\} \in E\}$ $\{\}$
- $\{g = (V, E)\}$ $g.get\_preds(n) \Rightarrow \{m \in V : \{m, n\} \in E\}$ $\{\}$

#4: WD graph ADT

- Abstract values look like: (V, E, w)
  - w maps edges to weights
- ADT in DSSL2

```
let weight? = OrC(num?, inf)

interface WDGRAPH:
  def set_edge(self, src: nat?, w: weight?,
               dst: nat?) -> NoneC
  def get_edge(self, src: nat?, dst: nat?) -> weight?
  def get_vertices(self) -> SetC[nat?]
  def get_succs(self, v: nat?) -> SetC[nat?]
  def get_preds(self, v: nat?) -> SetC[nat?]
```

#4: WD ADT Laws - Part 1

- $\{g = (V, E, w) \land n, m \in V \land a < \infty\}$
  $g.set\_edge(n, a, m) \Rightarrow None$
  $\{g = (V, E \cup \{(n, m)\}, w \cup \{(n, m) \rightarrow a\})\}$
- $\{g = (V, E, w) \land n, m \in V\}$
  $g.set\_edge(n, \infty, m) \Rightarrow None$
  $\{g = (V, E \setminus \{(n, m)\}, w \setminus \{(n, m) \rightarrow a\})\}$

- $\{g = (V, E, w) \land (n, m) \in E\}$ $g.get\_edge(n, m) \Rightarrow w(n, m)$ $\{\}$
- $\{g = (V, E, w) \land (n, m) \notin E\}$ $g.get\_edge(n, m) \Rightarrow \infty$ $\{\}$

#4: WD ADT Laws - Part 2

- $\{g = (V, E, w)\}\ \ g.get\_vertices() \Rightarrow V\ \ \{\,\}$

- $\{g = (V, E, w)\}\ \ g.get\_succs(n) \Rightarrow \{\, m \in V : \{n, m\} \in E\,\}\ \ \{\,\}$
- $\{g = (V, E, w)\}\ \ g.get\_preds(n) \Rightarrow \{\, m \in V : \{m, n\} \in E\,\}\ \ \{\,\}$

Concrete data structures for graphs

Cost parameters

- e = number of edges
- v = number of vertices

How can we implement a UU graph ADT?

- Using what we know so far?
  - A linked list of edges
  - Hash table of edges

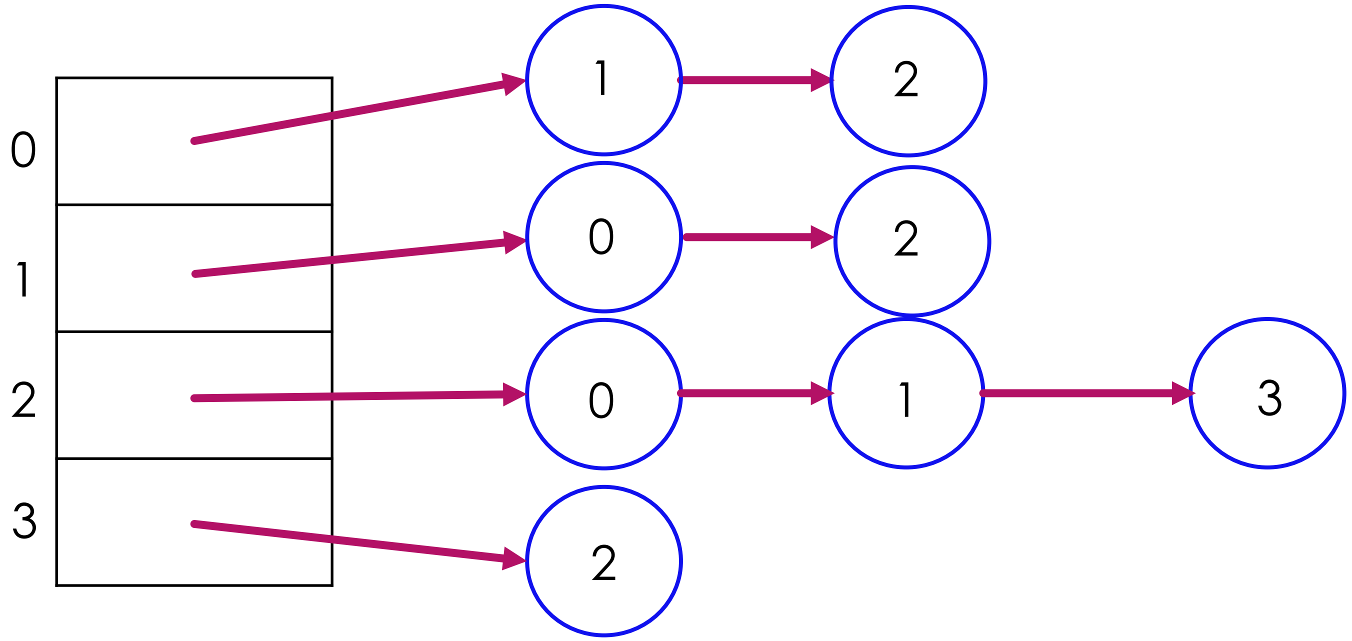
	Linked list of edges	Hash table of edges
has_edge	O(e)	O(1) avg + amortized
add_edge	O(1)	O(1) avg + amortized
get_neighbors	O(e)	O(e)

- Doesn’t allow vertex operations as these DSes only hold edge info
  - What if there are vertices without edges to/from them?
- Can we do better?

More data structures!

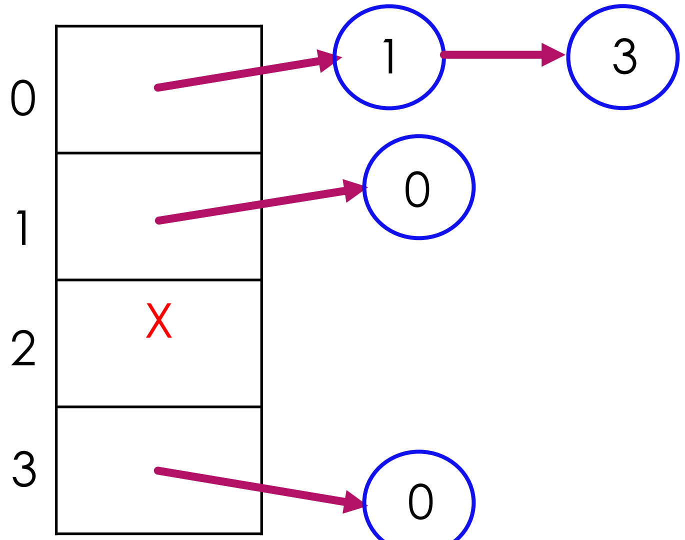
- Two common concrete data structures to represent graphs
  - Adjacency List (AL)
  - Adjacency Matrix (AM)

Adjacency list



Adjacency list

- Array of linked lists
- Each element in the array's index corresponds to a vertex #
- Each linked list for a vertex # contains that vertex's neighbors (or successors)
- Akin to dictionary mapping vertex # to list of neighbors or successors
  - Keys are 0… n-1 → direct addressing!



Adjacency matrix

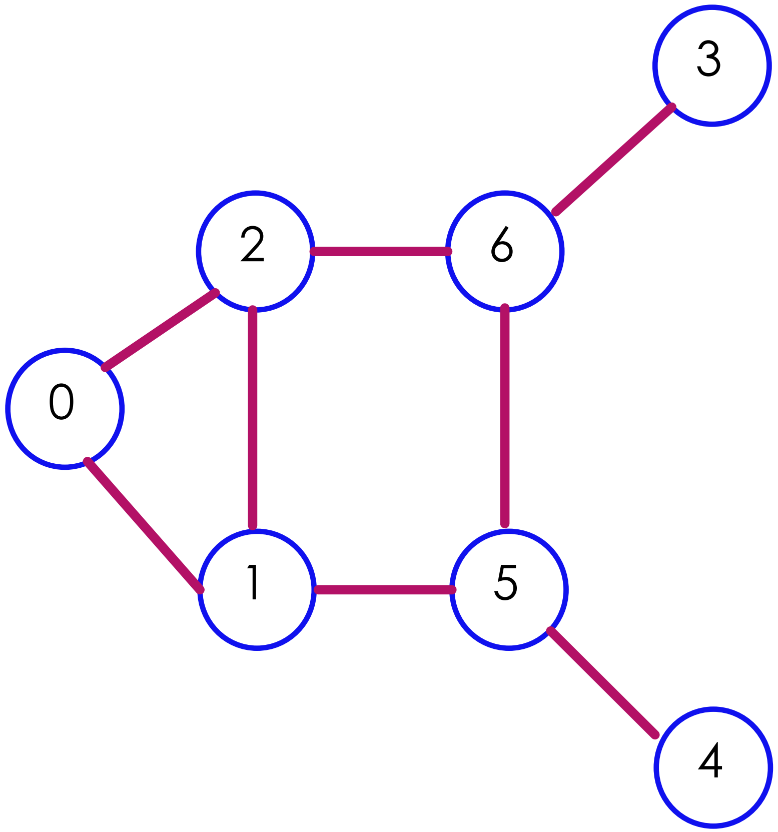
	0	1	2	3
0	F	F	F	F
1	F	F	T	T
2	F	T	F	F
3	F	T	F	F

Adjacency matrix

- v-by-v matrix (vector of vectors)
- Each element in the matrix contains a Boolean
  - True if there is an edge between two vertices
  - False if there is no edge between two vertices

	0	1	2	3
0	F	F	F	F
1	F	F	T	T
2	F	T	F	F
3	F	T	F	F

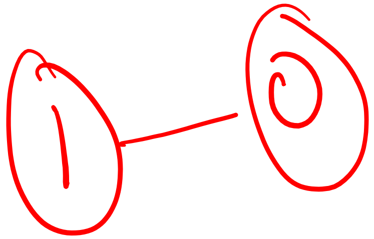
Undirected graph (unweighted)

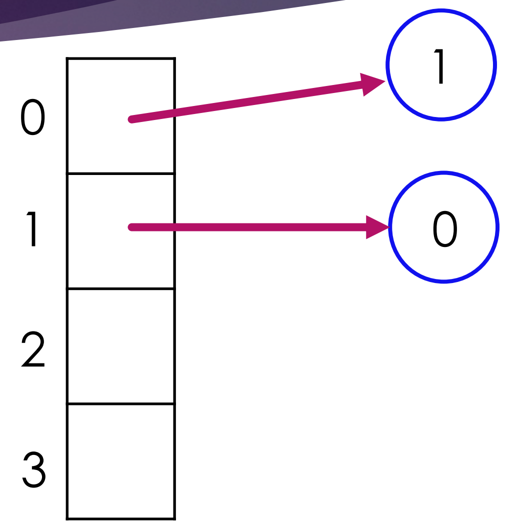


Undirected graph (unweighted)

- Representations are symmetric
- With an adjacency list
  - Every edge is represented in two lists
  - E.g., AL on the right is a graph with one edge: $0 - 1$
- With an adjacency matrix
  - Every edge is represented in two cells/elements
  - E.g., AM on the right is a graph with one edge: $0 - 1$

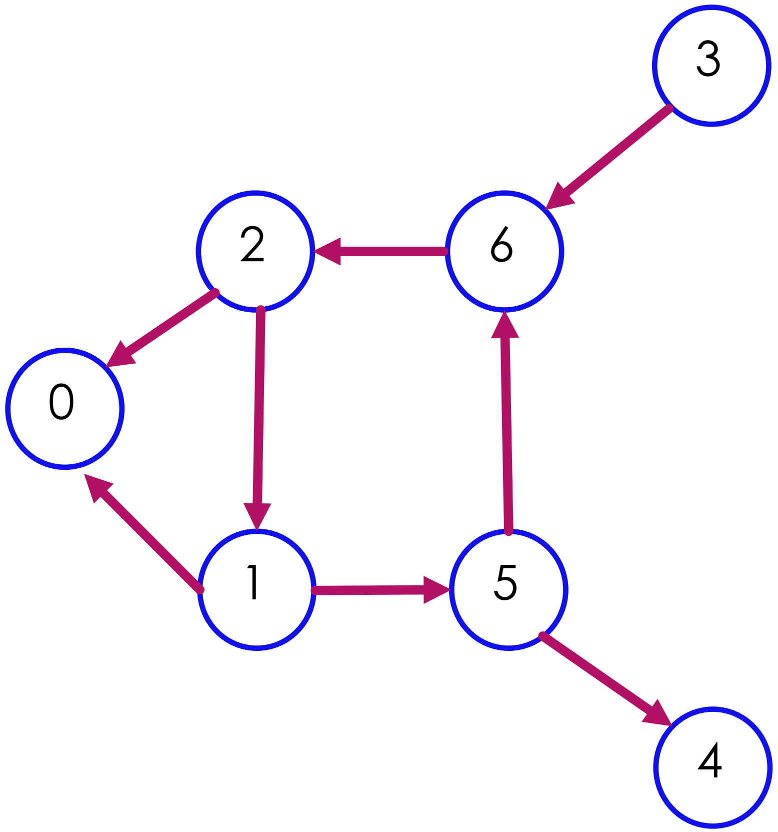
- $0 - 1$ is still only one edge





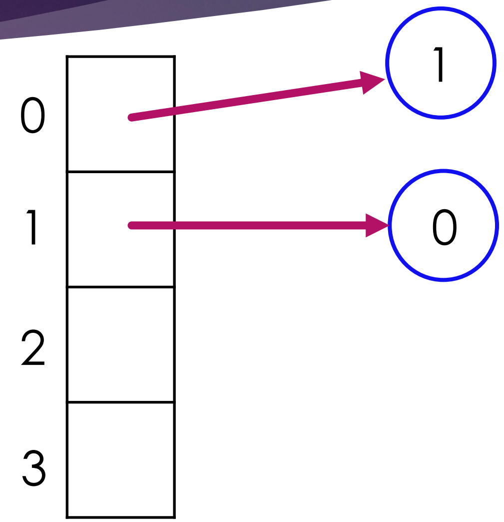
	0	1	2	3
0	F	T	F	F
1	T	F	F	F
2	F	F	F	F
3	F	F	F	F

Directed graph (unweighted)



Directed graphs (unweighted)

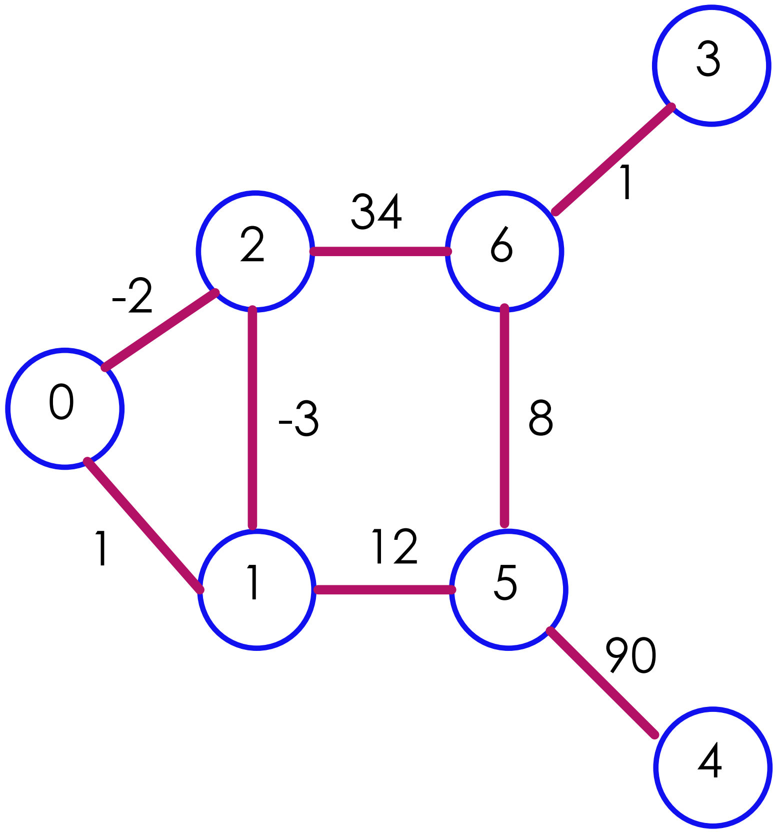
- Representations may not be symmetric
- With an adjacency list
  - Only the predecessor vertex # stores the edge
  - E.g., AL on the right is a graph with two edges: $0 \rightarrow 1$, $1 \rightarrow 0$



- With an adjacency matrix
  - Only the cell with the predecessor vertex # on the left stores the edge
  - E.g., AM on the right is a graph with one edge: $0 \rightarrow 1$

	0	1	2	3
0	F	T	F	F
1	F	F	F	F
2	F	F	F	F
3	F	F	F	F

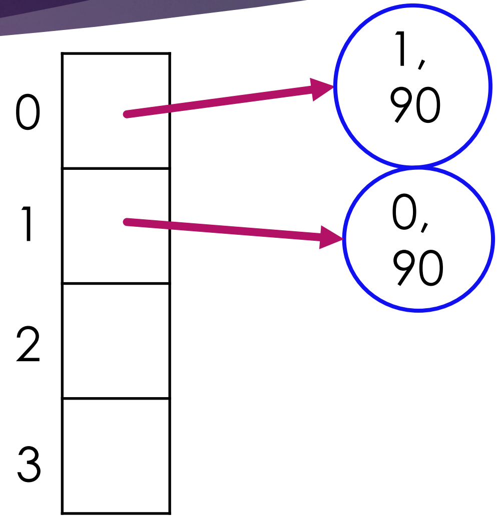
Undirected weighted graph



Undirected weighted graph

$0 \overset{90}{\text{——}} 1$

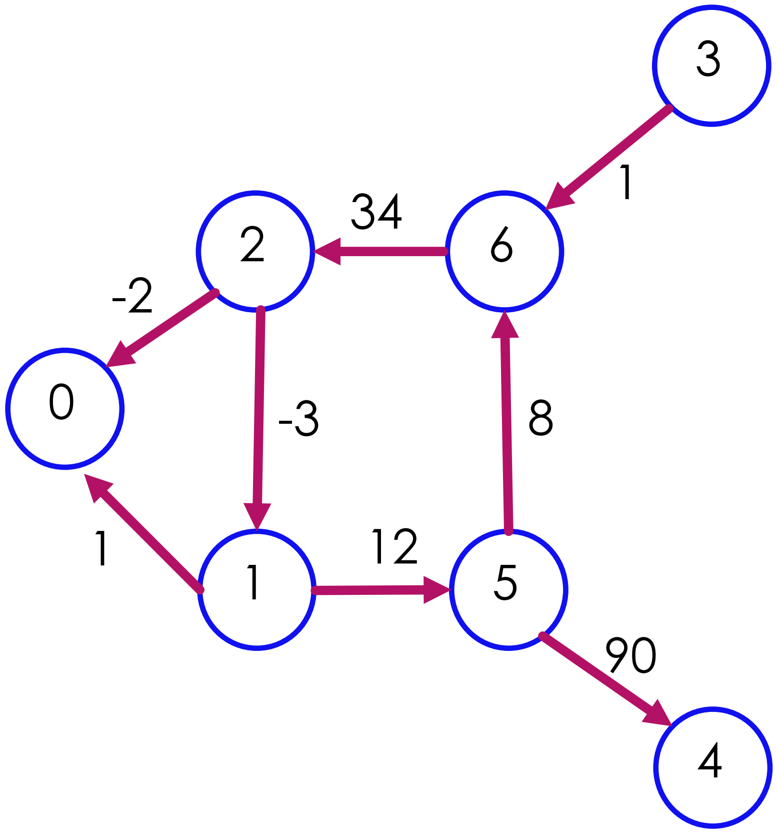
- With an adjacency list
  - Every item in the list can contain the other vertex # and the edge weight
  - E.g., AL on the right is a graph with one edge: $0 - 1$ with weight 90



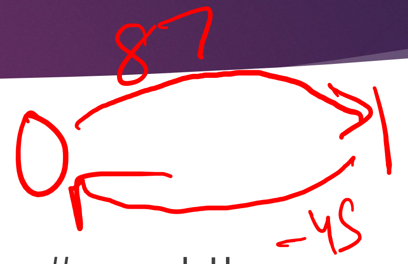
- With an adjacency matrix
  - All cells have a default value and cells for existing edges contain the edge weight
  - E.g., AM on the right is a graph with one edge: $0 - 1$ with weight 90

	0	1	2	3
0	∞	90	∞	∞
1	90	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞

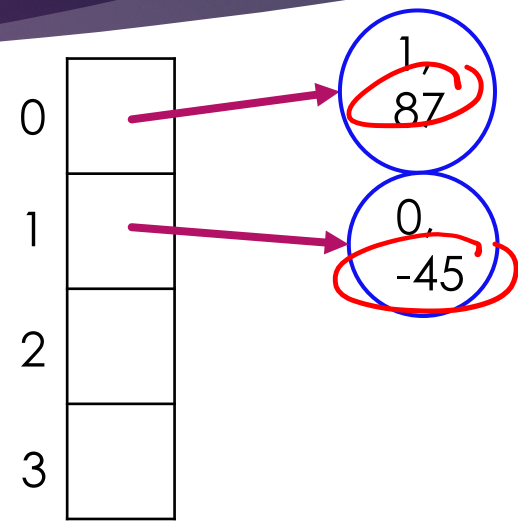
Directed weighted graph



Directed weighted graph



- With an adjacency list
  - Every item in the list can contain the successor vertex # and the edge weight
  - E.g., AL on the right is a graph with two edges: $0 \rightarrow 1$ (weight = 87), $1 \rightarrow 0$ (weight = -45)



- With an adjacency matrix
  - Cells for existing edges contain the edge weight; remaining cells contain a default value
  - E.g., AM on the right is a graph with one edge: $1 \rightarrow 0$ with weight 34

	0	1	2	3
0	∞	∞	∞	∞
1	34	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞

Complex data in graphs

- What if your vertices are not natural numbers?
  - Pretty much most useful graphs, e.g., social media users, cities)

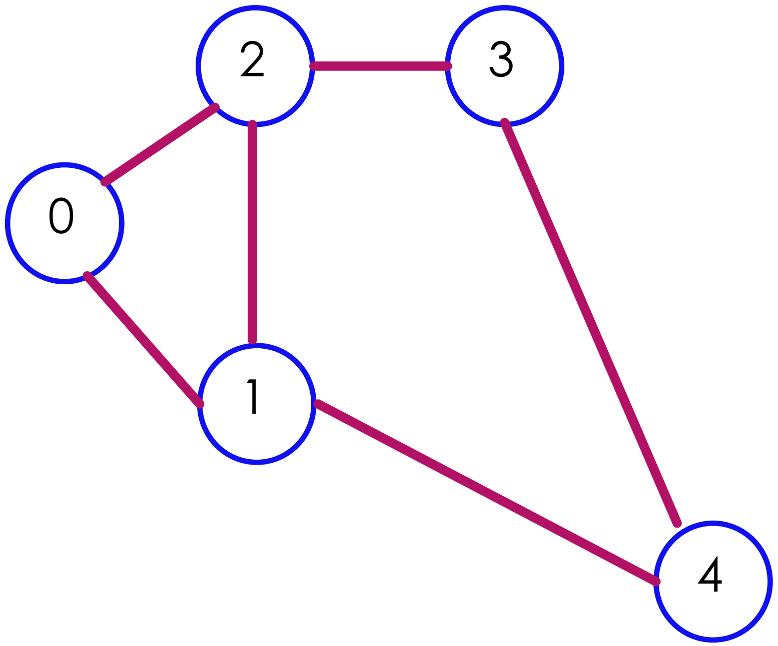
- For simplicity, still aim to use natural number vertices but…
- Just use another ADT to help: a dictionary!
  - Maintain a dictionary (your choice of impl.) of vertex ID #s to actual data
  - E.g., dictionary mapping natural numbers to city names

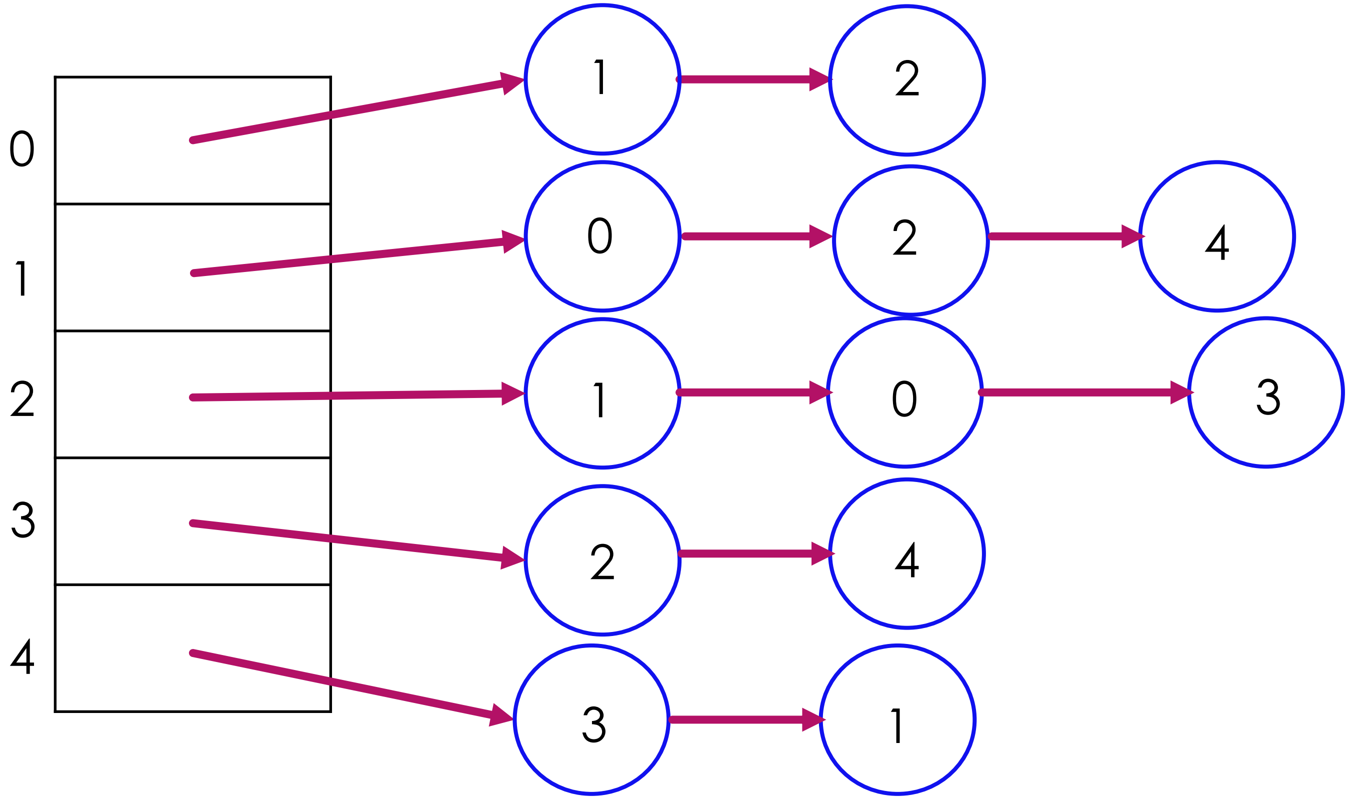
    {0: 'Chicago', 1: 'Milwaukee', 2: 'St. Louis'}
  - And maintain a dictionary in the other direction if needed

Practice with graph representations

UU Adjacency List

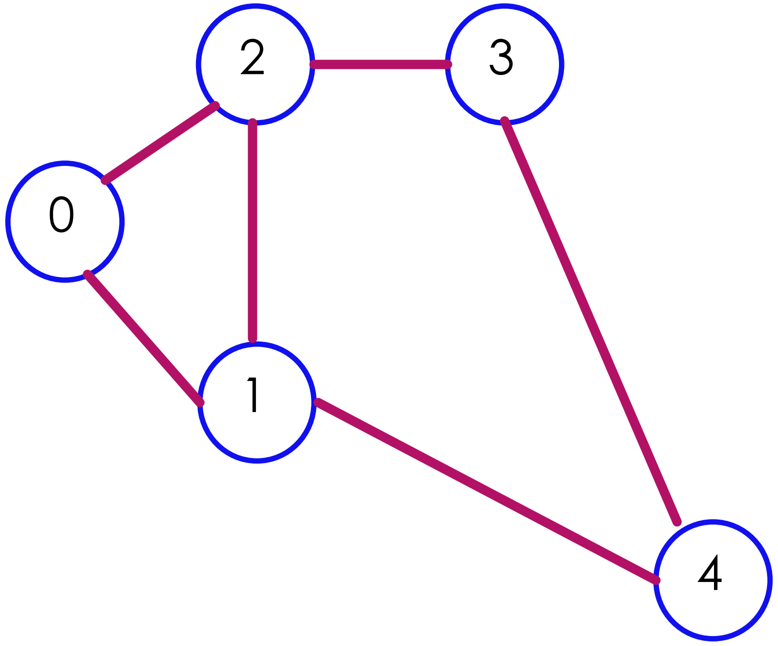
Symmetric: Bidirectional so one edge is represented twice!





UU Adjacency matrix

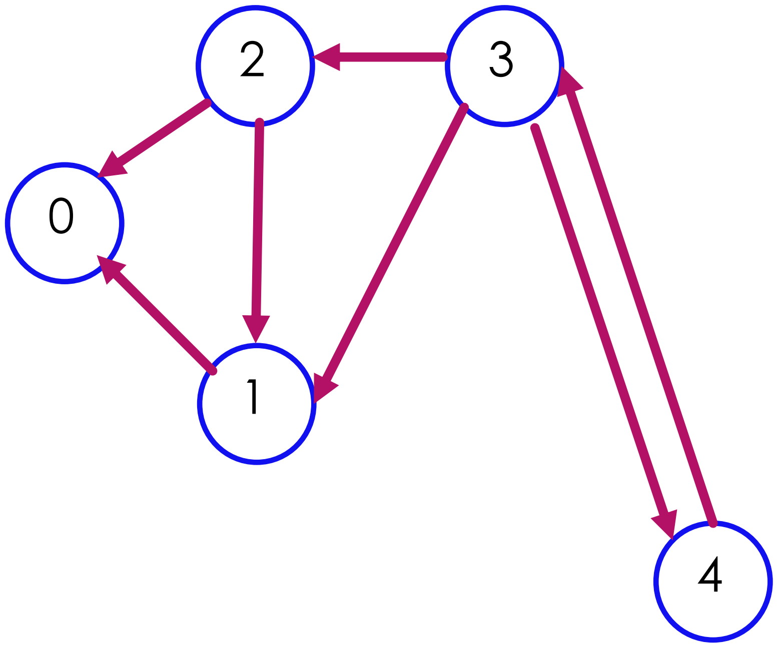
Symmetric: Bidirectional so one edge is represented twice!

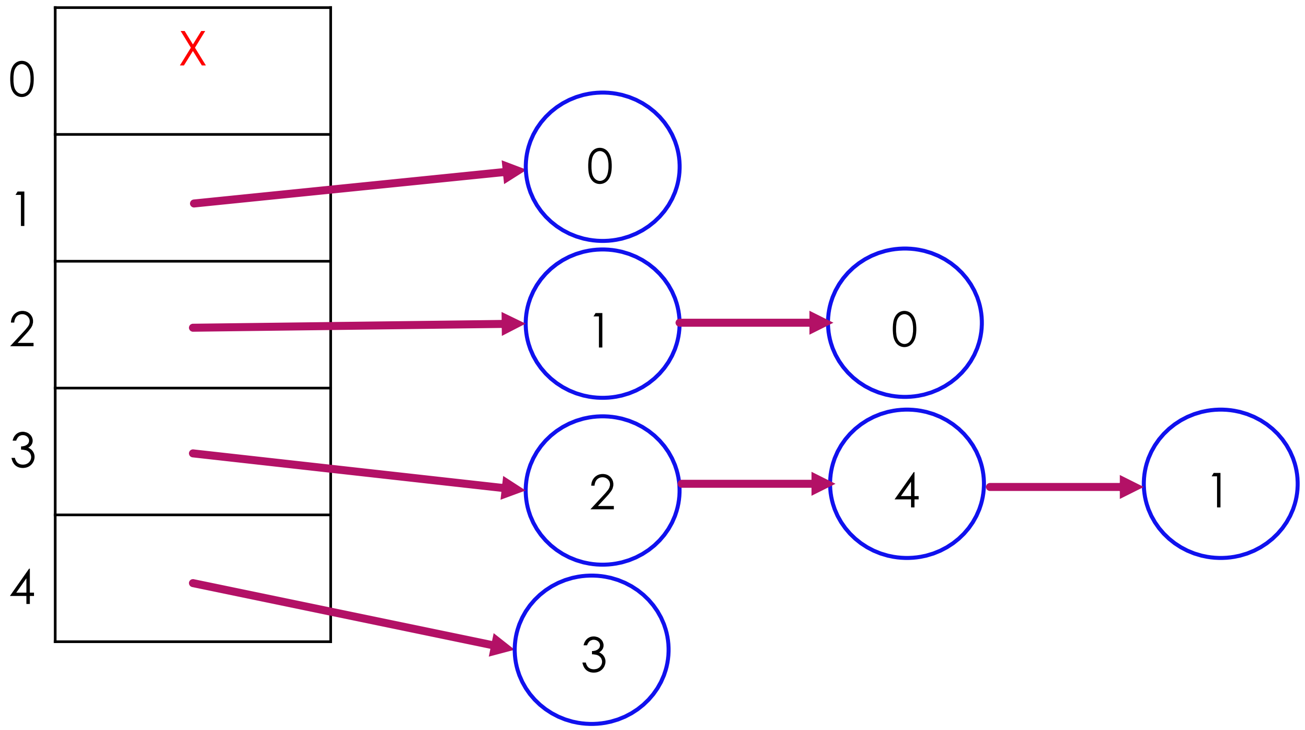


	0	1	2	3	4
0	F	T	T	F	F
1	T	F	T	F	T
2	T	T	F	T	F
3	F	F	T	F	T
4	F	T	F	T	F

UD Adjacency List

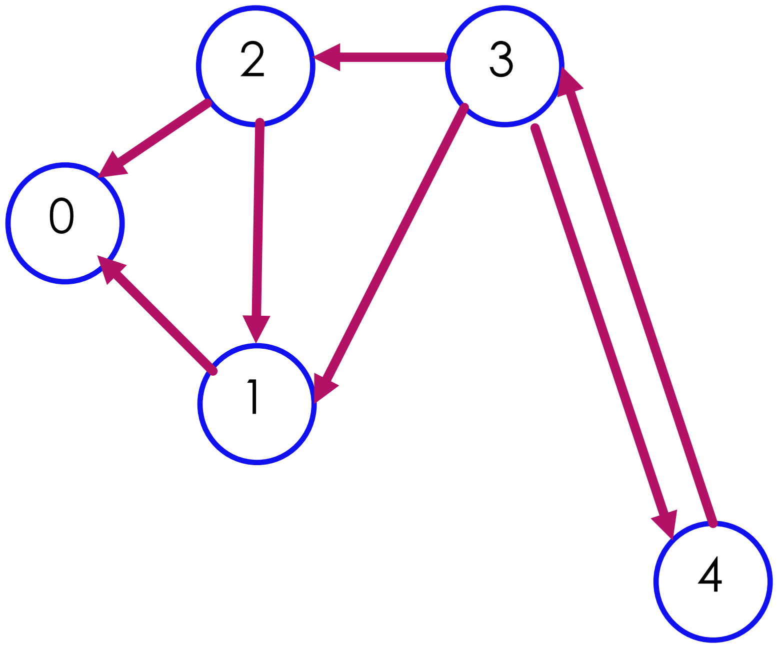
Asymmetric: Direction matters; one edge is represented once!





UD Adjacency matrix

Asymmetric: Direction matters; one edge is represented once!



	0	1	2	3	4
0	F	F	F	F	F
1	T	F	F	F	F
2	T	T	F	F	F
3	F	T	T	F	T
4	F	F	F	T	F

Same idea applies to weighted graphs

- **For adjacency list:** each vertex in a list also contains the weight

- **For adjacency matrix:** the cell contains the weight and non-existing edges have a default value

Graph implementation skeleton

```
class GraphImpl(UUGRAPH): #can implement one of the other graph ADTs too
    # Here goes data about the adjacency matrix or adjacency list
        # For Adjacency list, the data may look like:
            #  An array where each element can hold a linked list of edges
        # For Adjacency matrix, the data may look like:
            #  A vector of vectors where the outer vector is size v
            #  and each inner vector is also size v

    # Here goes functions that operate on the above data
```

Complexity of representations

Cost parameters

- e = number of edges
- v = number of vertices

- For simplicity, let's consider a UU graph
- Undirected graph → maximum value of $e = \frac{v(v-1)}{2}$

Space complexity: Let’s fill this in

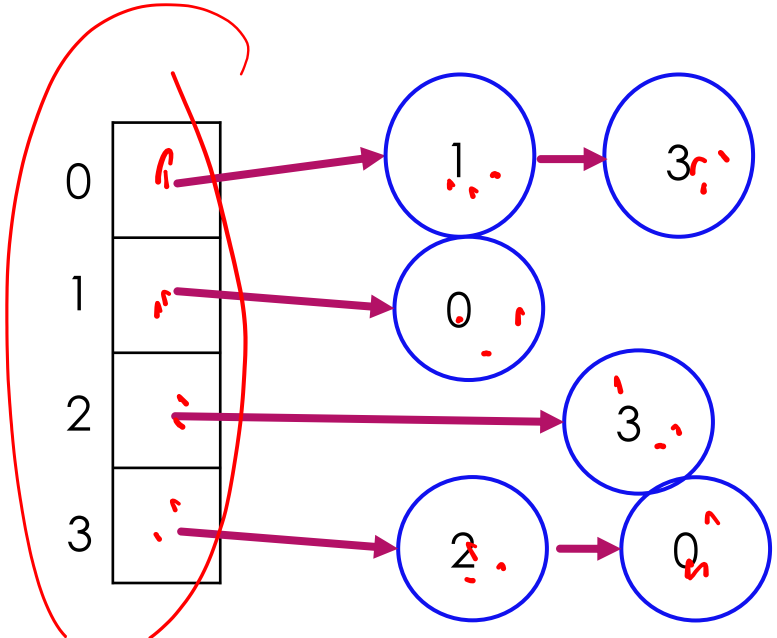
Adjacency list	Adjacency matrix

Space complexity

Adjacency list $O(v + \cancel{2}e)$

Array: V

Edges space: 2e



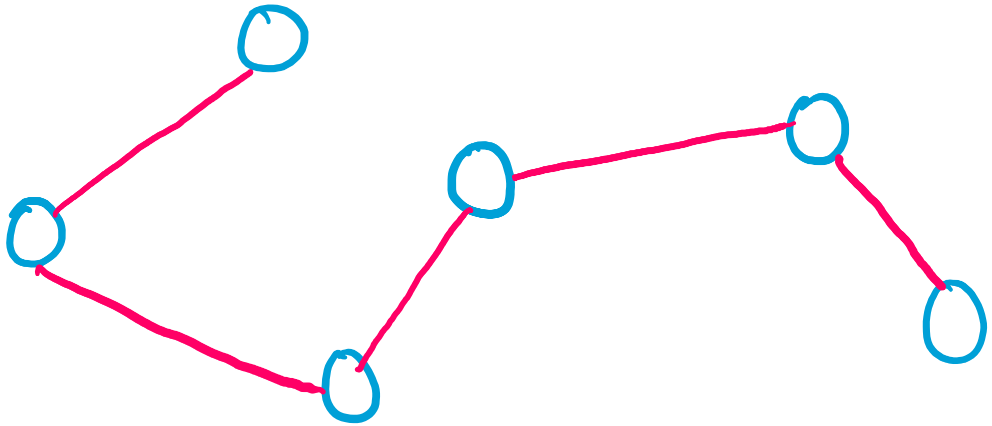
Adjacency matrix

$O(v^2)$

	0	1	2	3
0	F	T	F	F
1	T	F	F	F
2	F	F	F	F
3	F	F	F	F

In-class exercise (4 minutes)

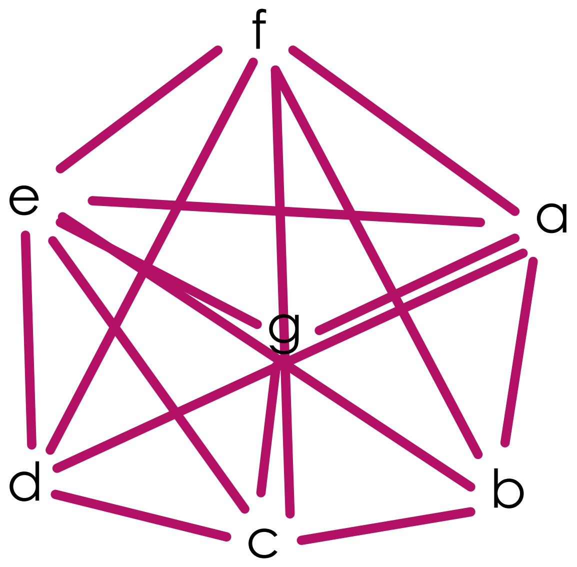
1. Assume you have a graph that looks like below. Which implementation would you choose out of **Adjacency List** and **Adjacency Matrix**?



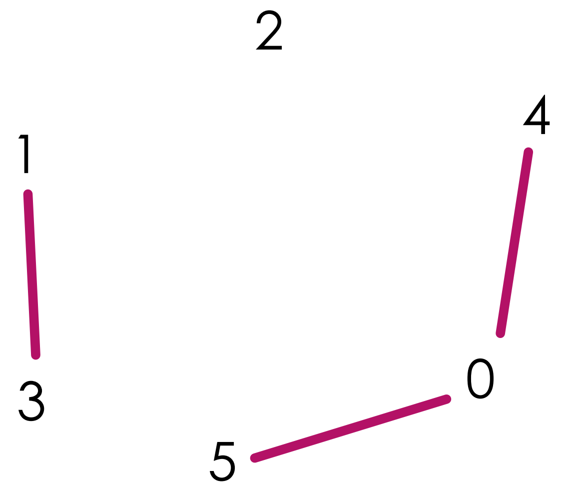
2. Explain why you picked your answer above in a few sentences. Be specific given **<u>this particular graph</u>**.

Dense vs. sparse graphs

- Dense graphs:
  - A lot of edges
  - Number of edges in $O(v^2)$



- Sparse graphs:
  - Fewer edges
  - Number of edges is in $O(v)$ or $O(v \log v)$



Space complexity

Adjacency list	Adjacency matrix
$O(v + e)$	$O(v^2)$

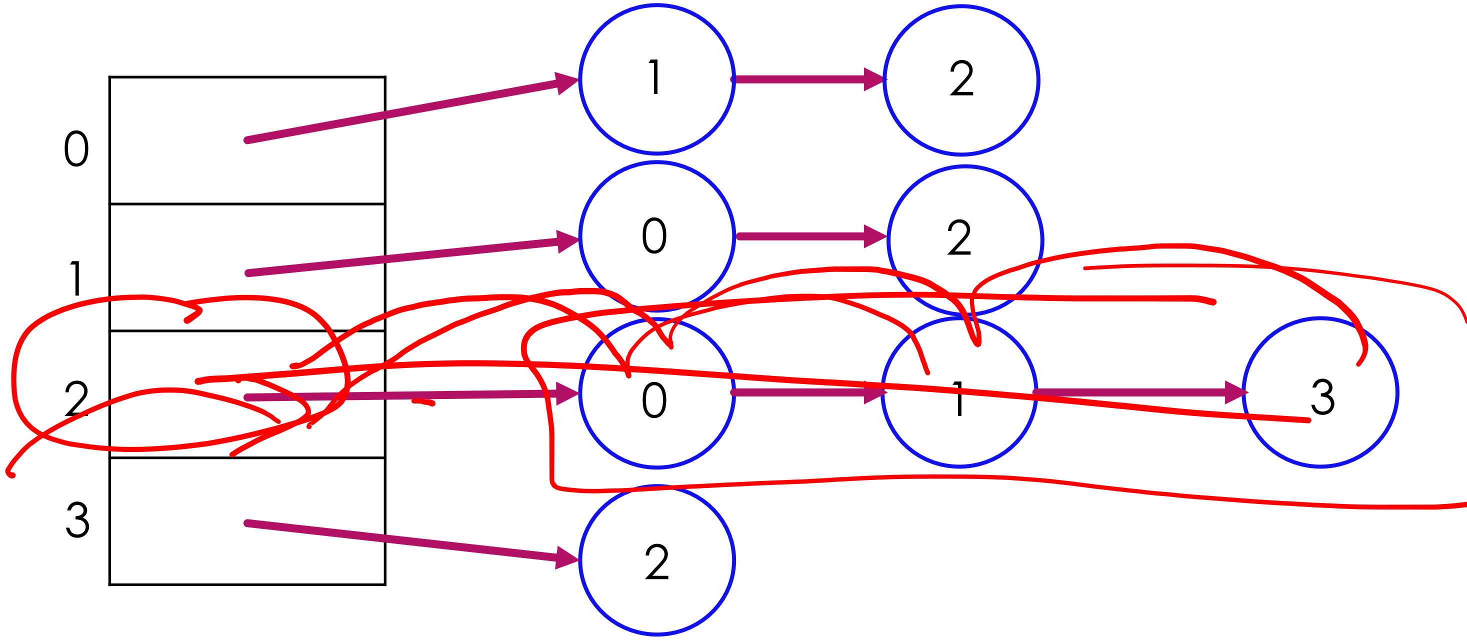
- Adjacency lists usually better for sparse graphs
- Adjacency list and matrix give same space complexity for dense graphs
  - What about time complexity though?
- Need to also consider operation complexity and importance of specific operations

Time complexity: Let’s fill this in

	Adjacency list	Adjacency matrix
add_edge		
has_edge		
get_vertices		
get_neighbors		

Adjacency list

	Adjacency list
add_edge	$O(e)$
has_edge	$O(e)$
get_vertices	$O(v)$
get_neighbors	$O(e)$



Adjacency matrix

	Adjacency matrix
add_edge	O(1)
has_edge	O(1)
get_vertices	O(v)
get_neighbors	O(v)

	0	1	2	3
0	F	F	F	F
1	F	F	T	T
2	F	T	F	F
3	F	T	F	F