Data Structures

CSCI 2270-202: REC 09

Sanskar Katiyar

Logistics

Office Hours (Zoom ID on Course Calendar)

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Logistics

Due March 22 Midnight

Recitation 9, 10

Assignment 7

Submit on Moodle

Recitation Outline

1. BST: Parent Pointers
2. STL Vector: Review
3. Graph: Overview
4. Graph: Representation
5. Graph: Insertion, Deletion
6. Exercise

BST: Parent Pointers

sanskarkatiyar.github.io/CSCI2270/ > Recitation 9 > Code > **bst_parent**

STL Vector: Review

Vector

What is a Vector?

A vector stores a sequence *(of values)* whose size can change

Vector *(Class)* ≈ **Array** *(Data Member)* + *(Methods)*

Abstracts functionality like Array Doubling, Appending to Array

sanskarkatiyar.github.io/CSCI2270/ > Recitation 9 > Code > **vector**

Vector: Declaration

```
vector<string> student(100);
```

Data type of vector — *Name of vector* — *Initial size of vector*

Syntax	Description
`vector<int> nums(10);`	A vector of 10 integers, like size of array
`vector<double> nums;`	A vector of type double, size 0
`vector<int> nums = {10, 11, 21};`	Declaration with items

Vector: Methods

Syntax	Description
push_back(`elem`**);**	Append `elem` to the vector
pop_back();	Delete the last element
resize(`new_size, pad_value`**);**	Resize the vector to `new_size`
begin(); end();	*Pointers* to front and back element of vector
front(); back();	Element at front and back respectively
size(); capacity();	Number of elements and allotted size, resp.
at(`index`**);**	Element at index, same as `vector[index]`
insert(`pos_ptr, repeat, val, …`**);**	Inserting element at a location

Vector: Iterator

```
vector<string>::iterator it;
```

Data type of vector — *Iterator Name*

Syntax	Description
`it = nums.begin();`	Returns pointer to front element
`it = nums.end();`	Returns pointer to last element
`cout << *it;`	Dereference like a pointer

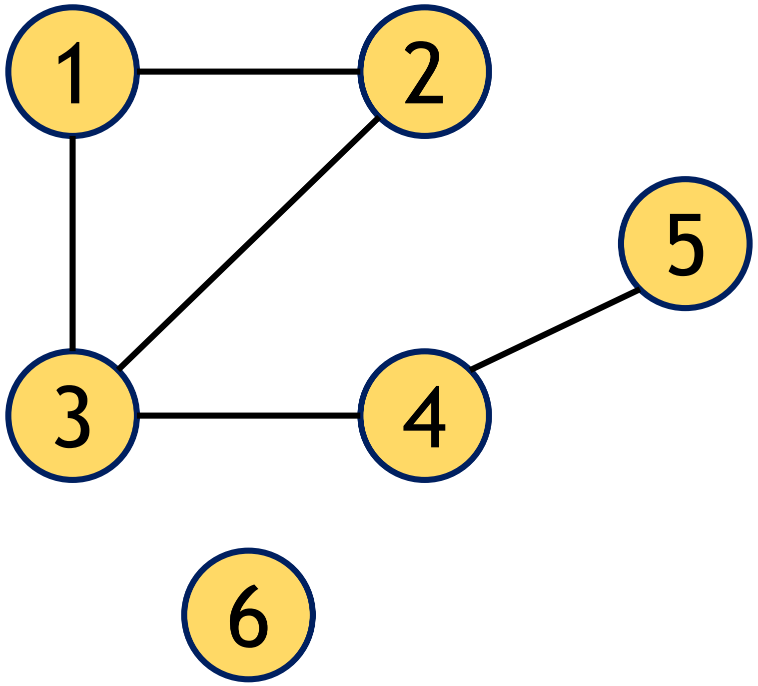
Graph: Overview

Graph ADT

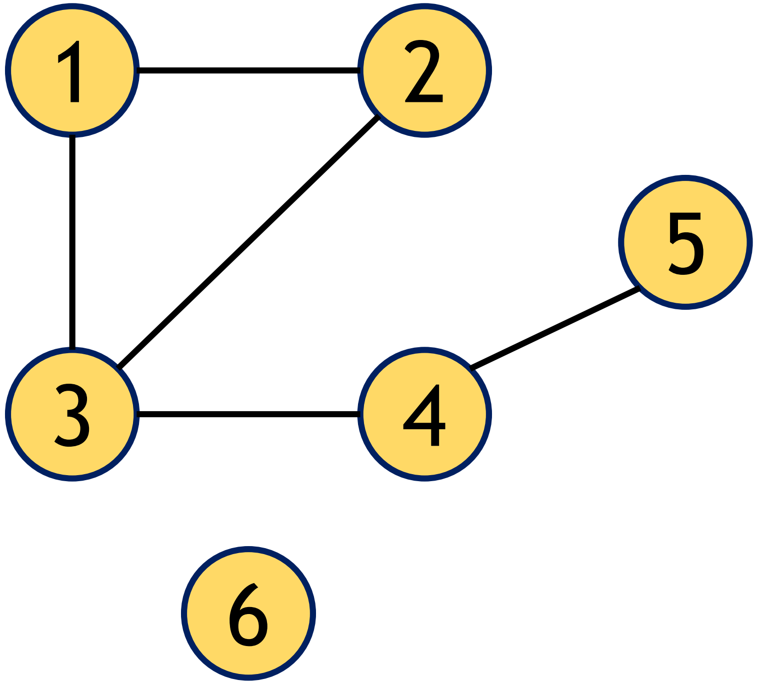
Composed of **Nodes (Vertices)**, **Edges**

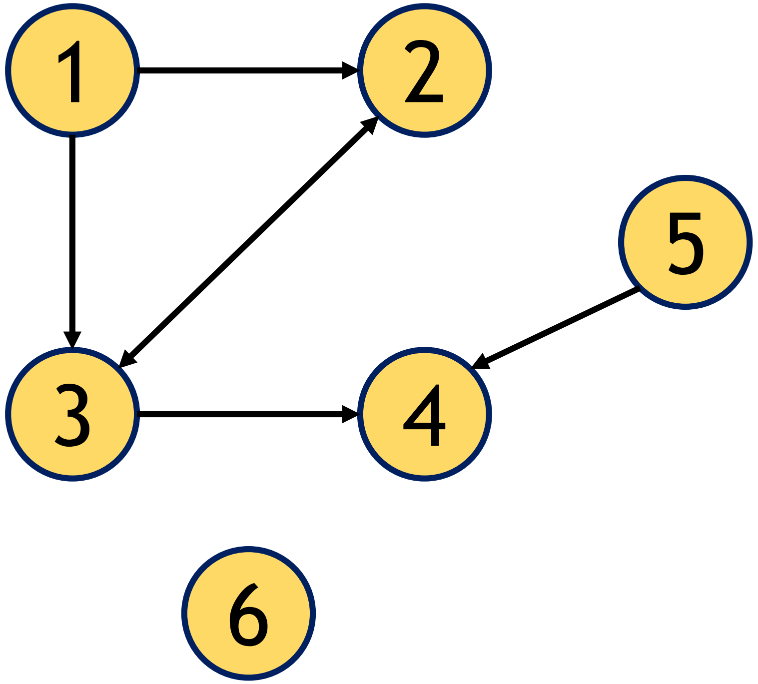
Non-linear, **Network**

Nodes are ***entities***, edges represent some ***relationship*** between these entities

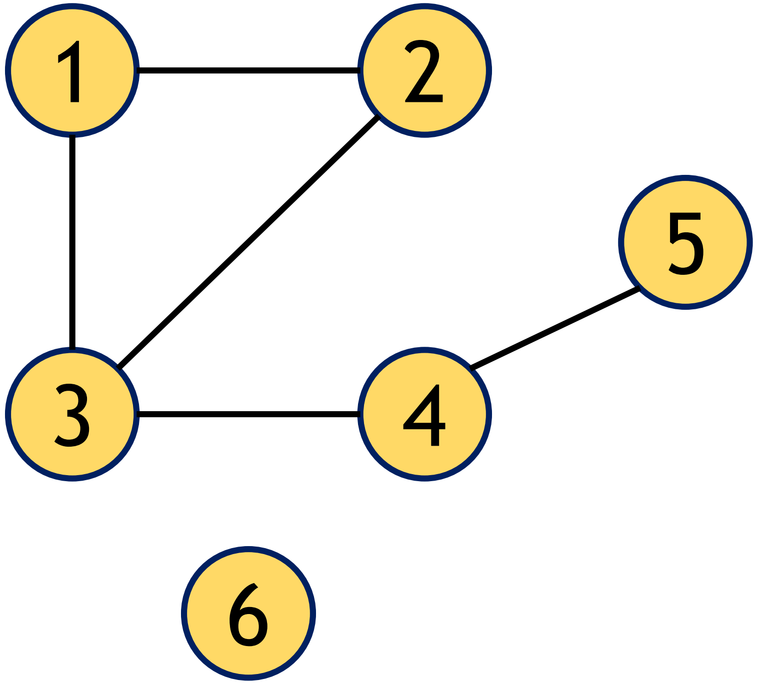


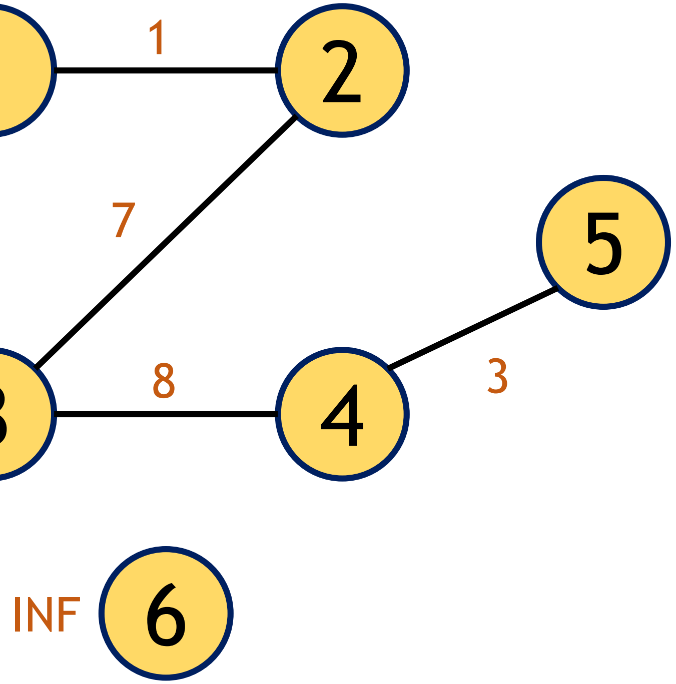
Graph: Undirected, Directed



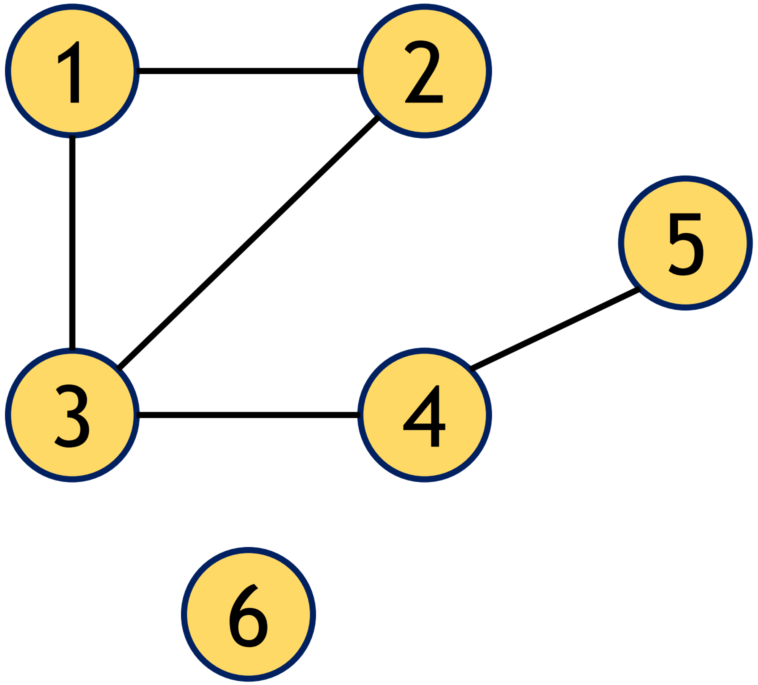


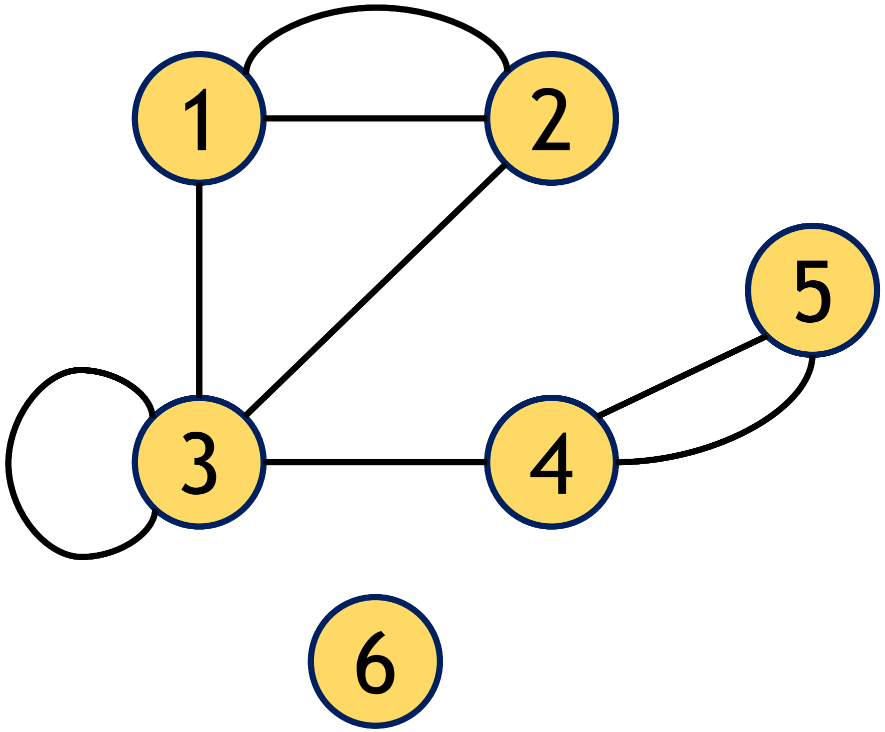
Graph: Unweighted, Weighted



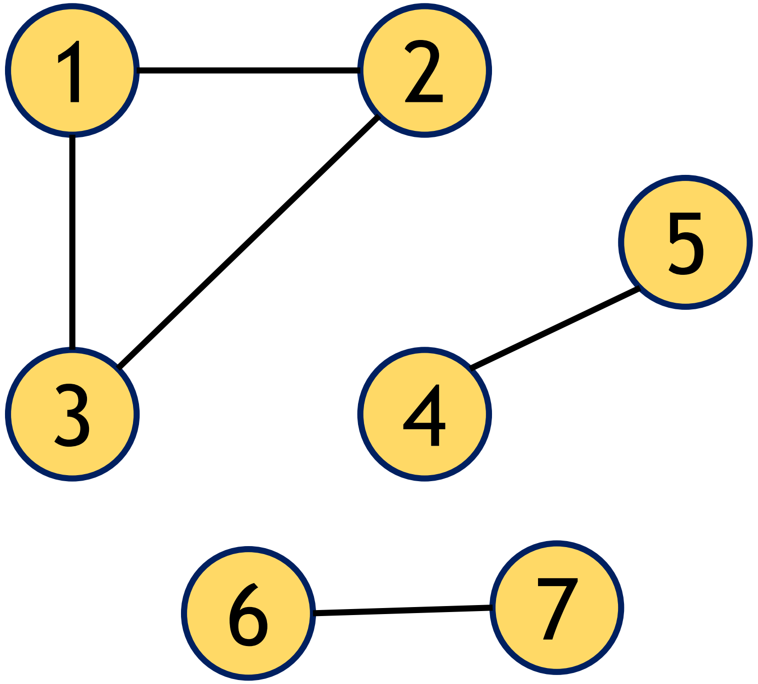


Graph: Single Edge, Multiple Edges





Graph: Connected Components



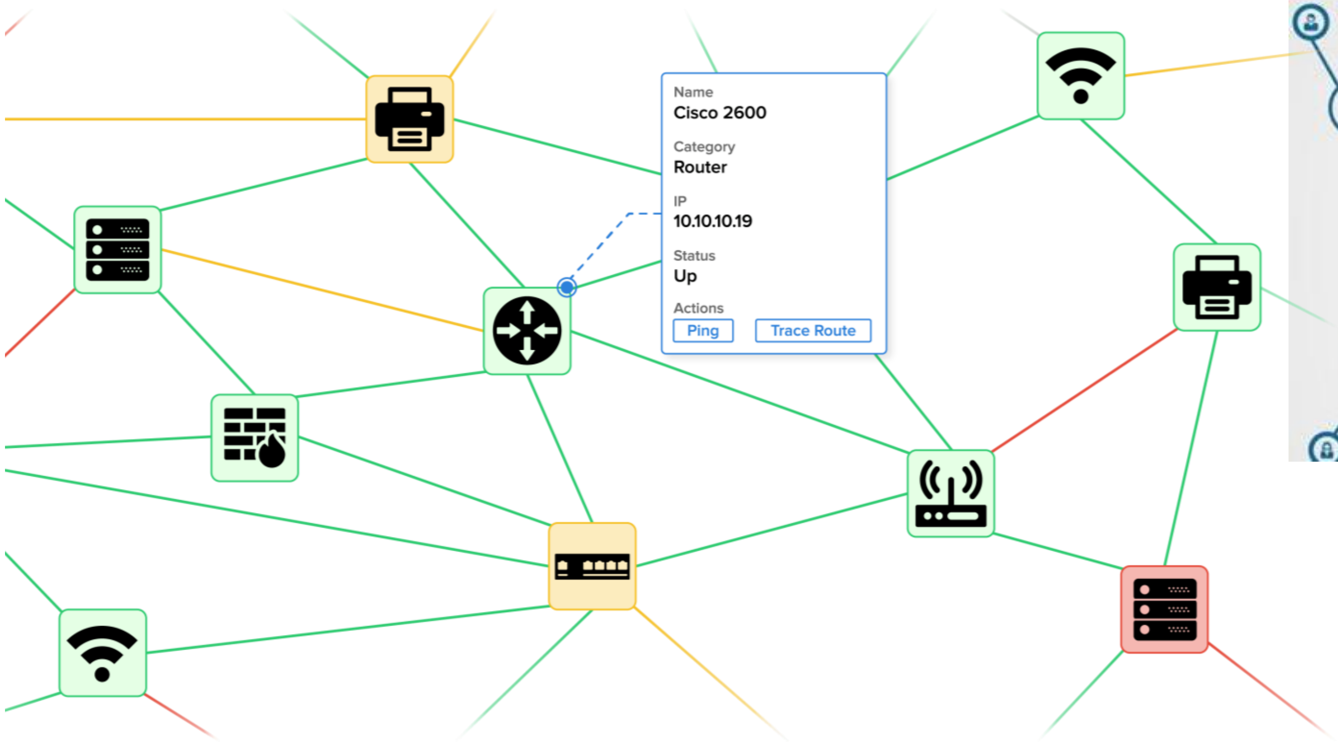
Shown: 3 components in a graph

They are referred all in the same graph

How many components does a tree have?

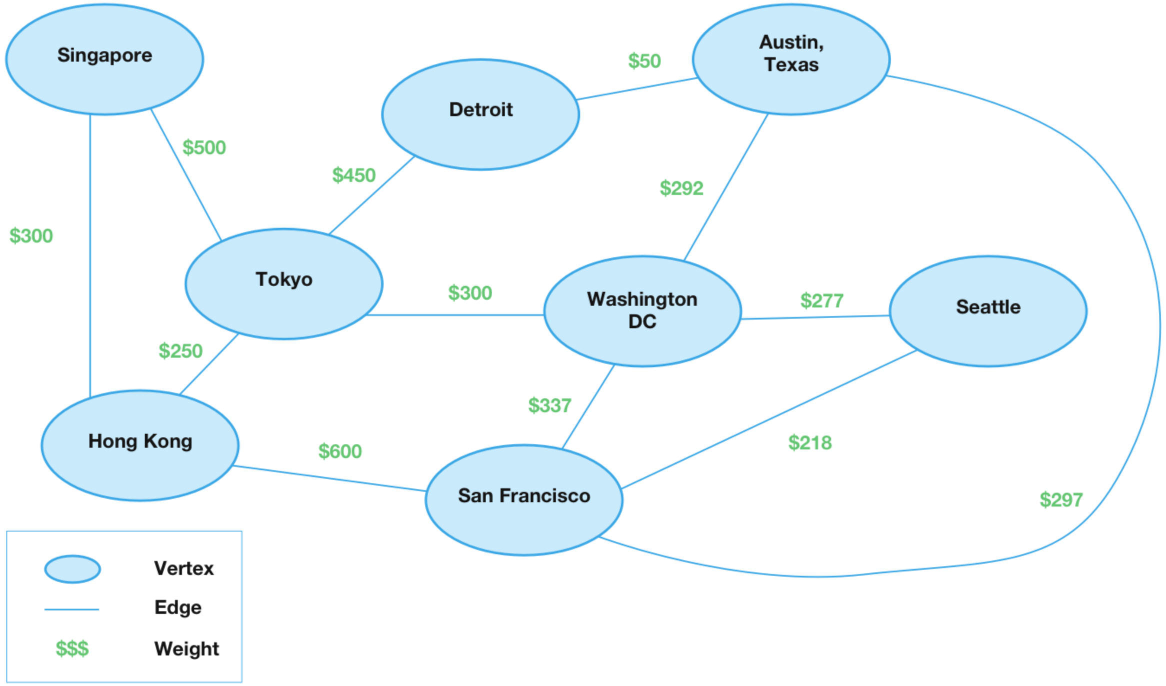
Graph: Applications

Maps, Networks



Graph: Applications

Maps, Networks

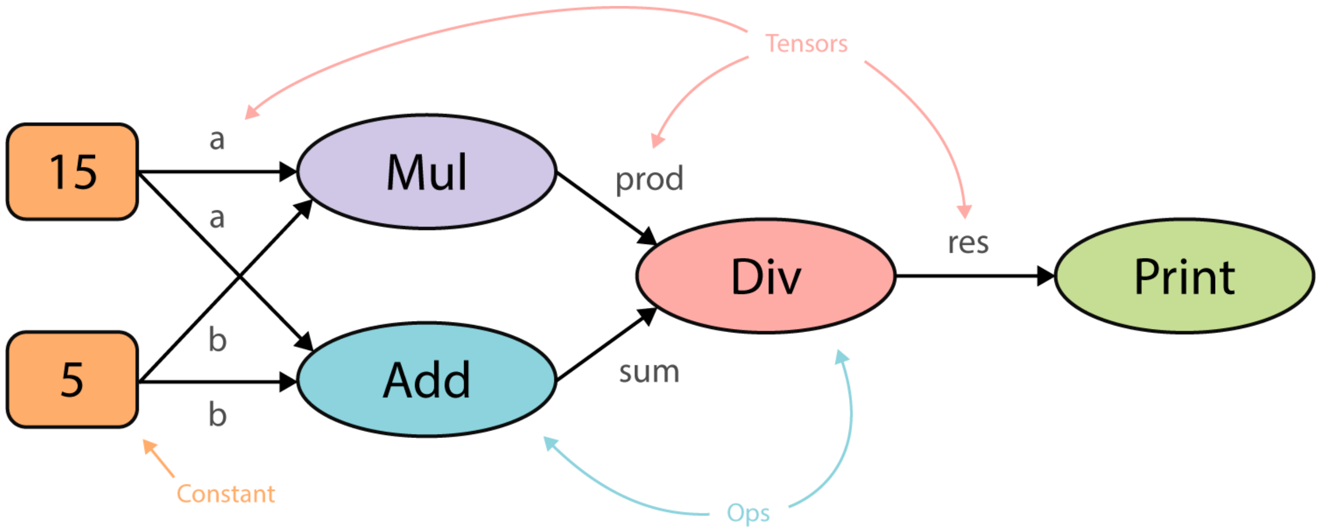


https://www.raywenderlich.com/773-swift-algorithm-club-graphs-with-adjacency-list

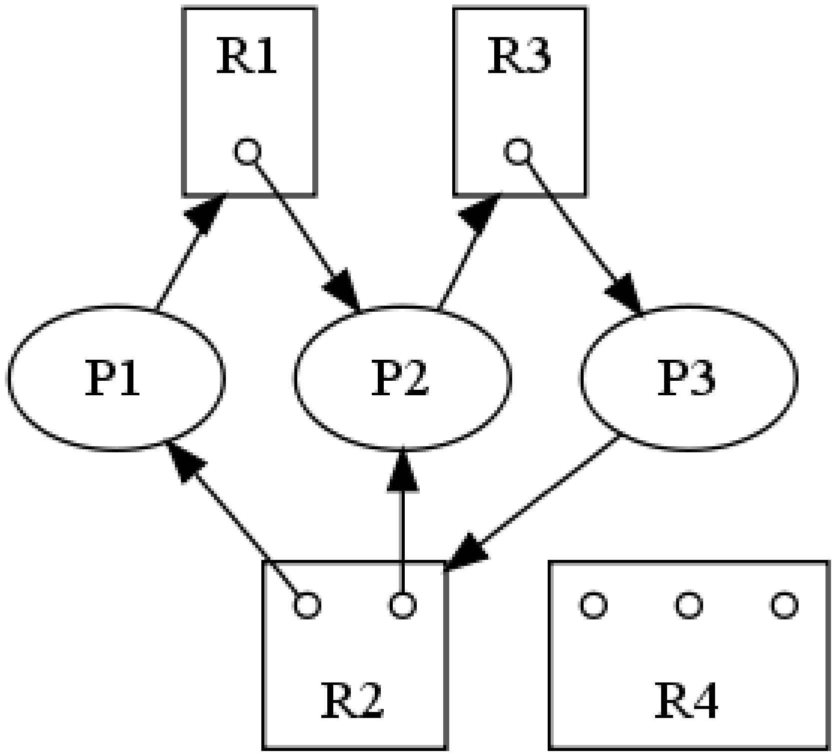
Ticket to Ride: London (Game models a Train Network)

Graph: Applications

Computational, Resource Allocation



https://medium.com/@d3lm/understand-tensorflow-by-mimicking-its-api-from-scratch-faa55787170d

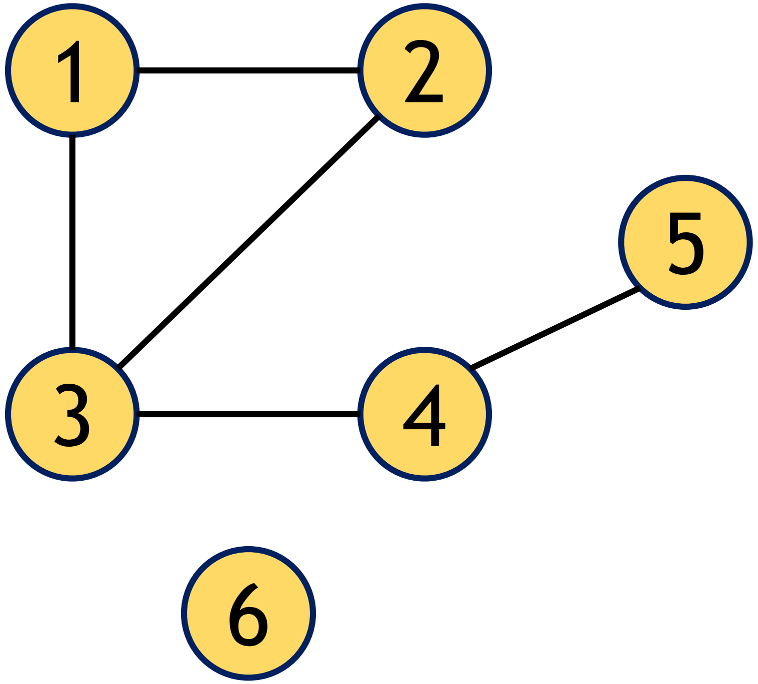


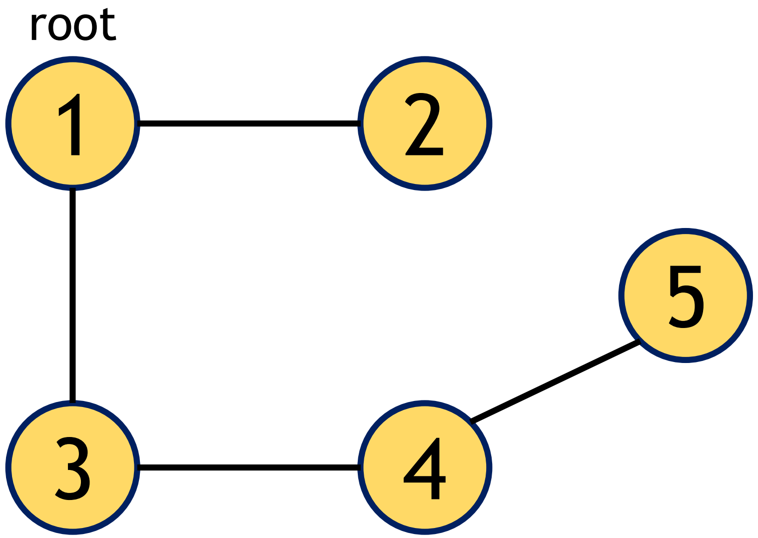
Resource Allocation Graph (with a Deadlock):
http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/deadlock.html

Graph ADT vs Tree ADT

BASIS	GRAPH	TREE
Model	Network	Hierarchical
Root	No such concept in a graph	Exactly one root node
Cycles, Loops	A graph can have self-loops, and cycles	Not permitted in a tree
Path	Multiple paths allowed between two nodes	Exactly one path between any two nodes
Connectivity	Singleton nodes are allowed	Tree is connected

Graph ADT vs Tree ADT





Graph: Representation

Graph: Representation

A variety of representations

Problem-dependent, Conventions

Popular:

- Adjacency Matrix
- Adjacency List
- Incidence Matrix

Graph: Adjacency Matrix

Matrix, Tensors*

Elaborate [Space ~ $O(N^2)$], Random access [Time ~ $O(1)$]

Great for dense graphs

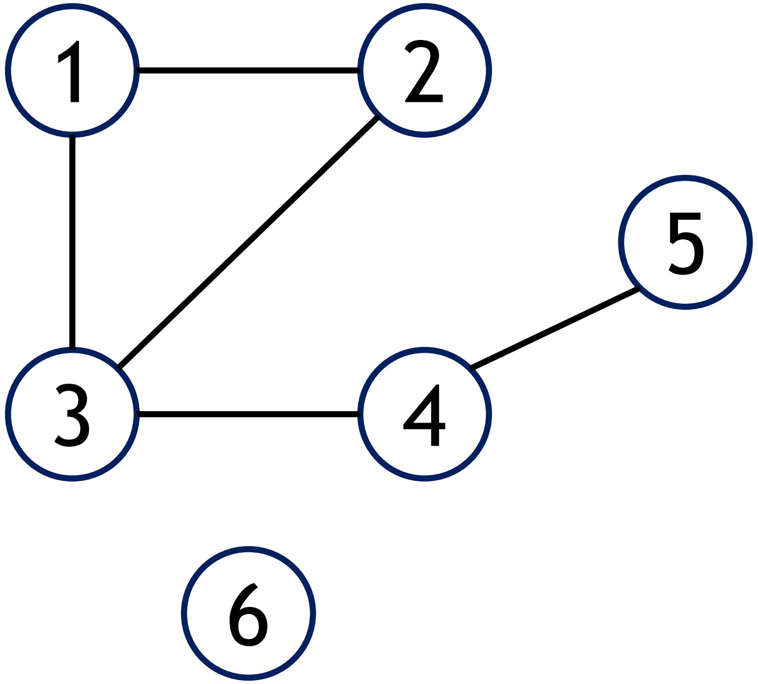
Graph has a lot of edges

Very few 0 entries in the matrix

Unsuitable where new nodes keep getting added

G[row_ix][col_ix]: Edge exists between node$_{row_ix}$ and node$_{col_ix}$

Graph: Adjacency Matrix



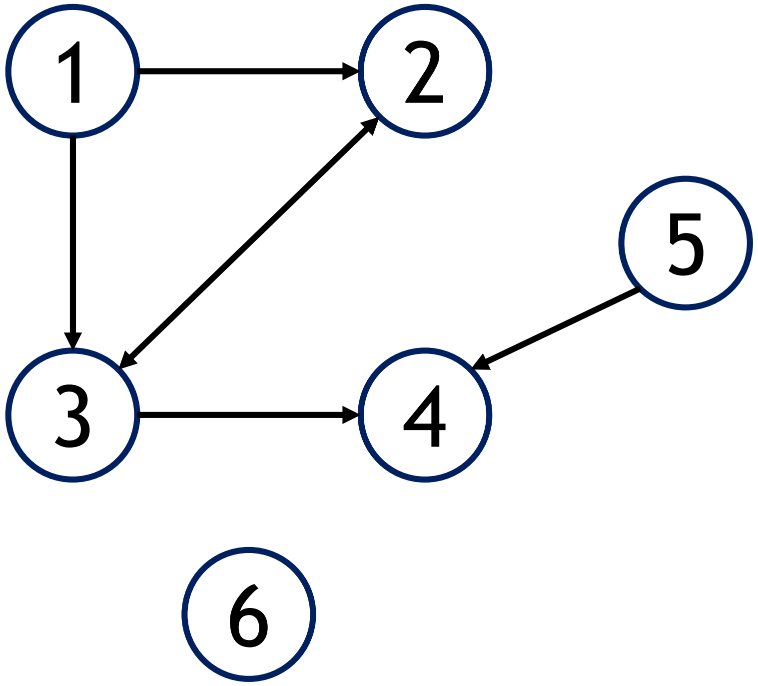
Undirected, Unweighted, No self-loops

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Symmetric Matrix

G[row_ix][col_ix]: Edge exists *from* node_{row_ix} and node_{col_ix}

Graph: Adjacency Matrix

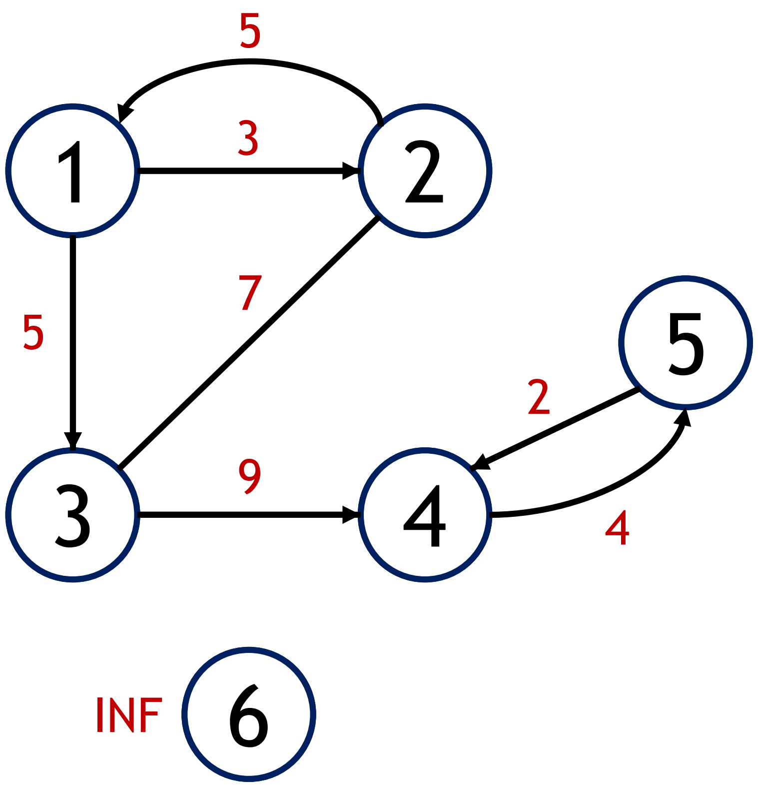


Directed, Unweighted, No self-loops

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

G[row_ix][col_ix]: Edge weight for edge *from* $node_{row\_ix}$ and $node_{col\_ix}$

Graph: Adjacency Matrix



Directed, Weighted, No self-loops

$$\begin{pmatrix} 0 & 3 & 5 & \infty & \infty & \infty \\ 5 & 0 & 7 & \infty & \infty & \infty \\ \infty & 7 & 0 & 9 & \infty & \infty \\ \infty & \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Graph: Adjacency List

Graphs are often sparse

Matrices for N nodes will have N^2 elements, most often 0

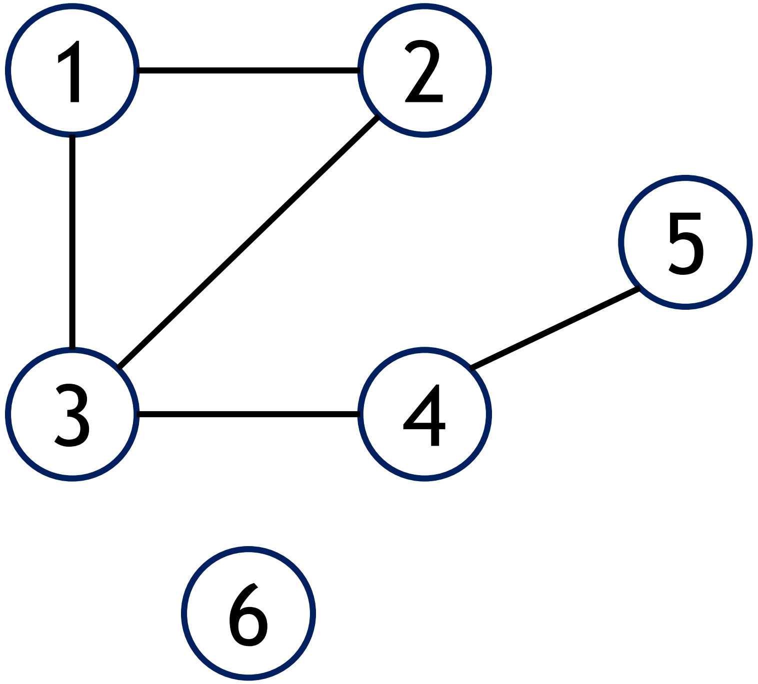
May waste a lot of storage, esp. undirected graphs

Linked List, Vectors can be used for this representation

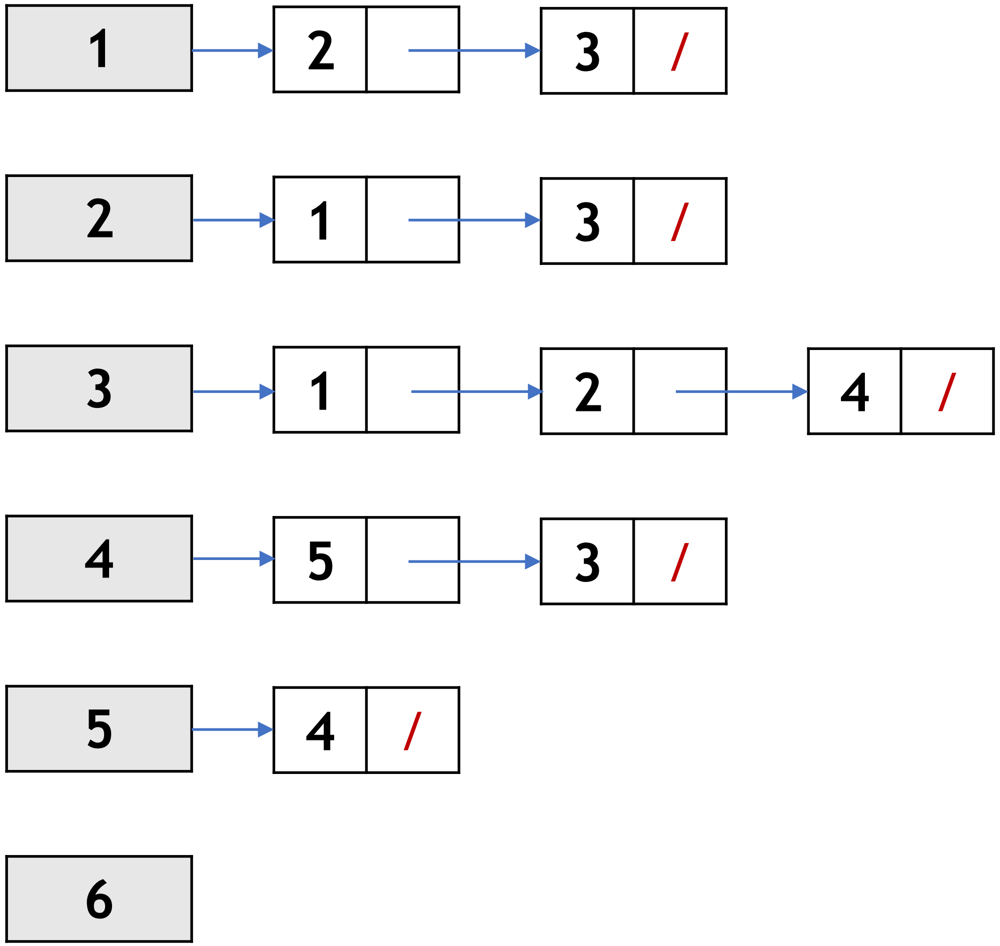
Sparse matrices are also another alternative

Easy addition of new nodes

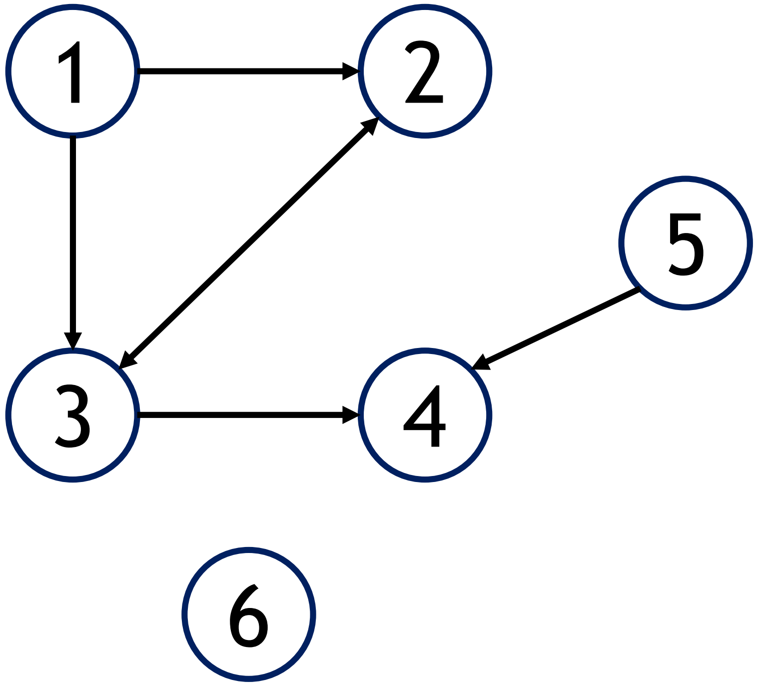
Graph: Adjacency List



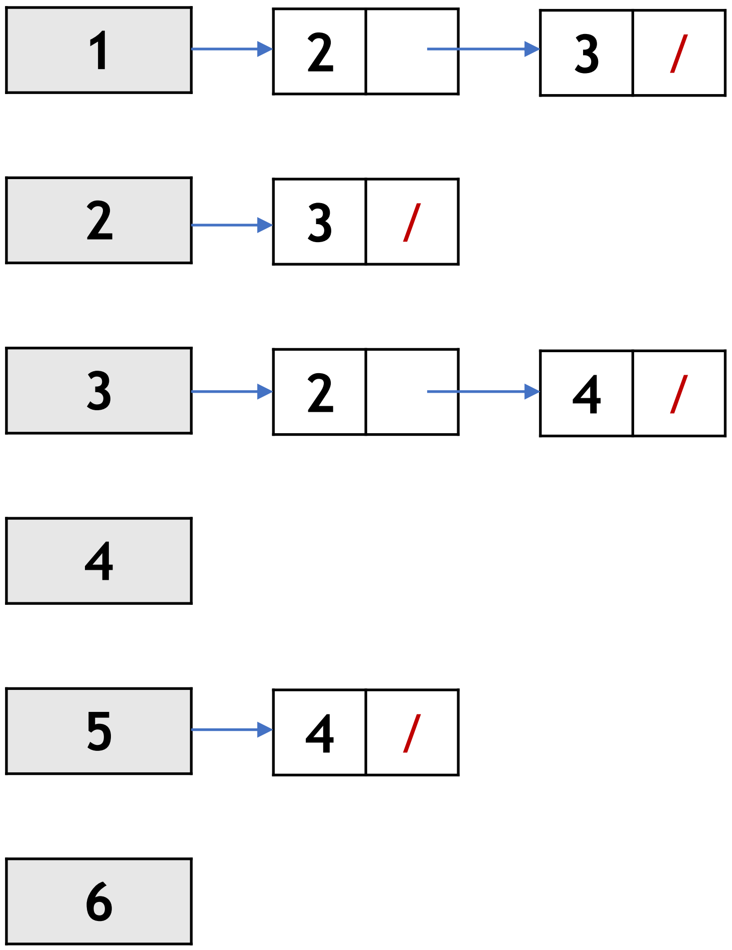
Undirected, Unweighted



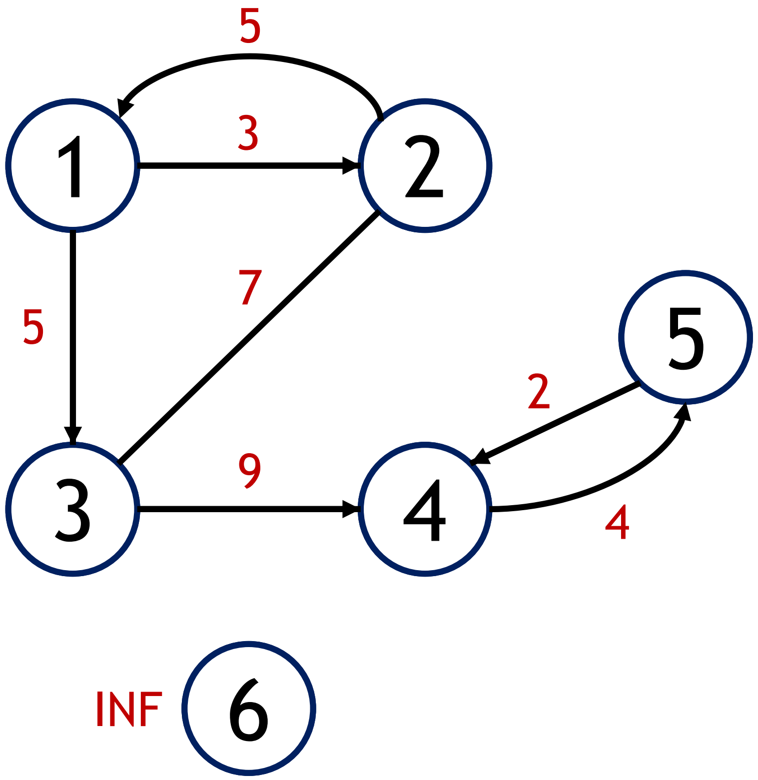
Graph: Adjacency List



Directed, Unweighted

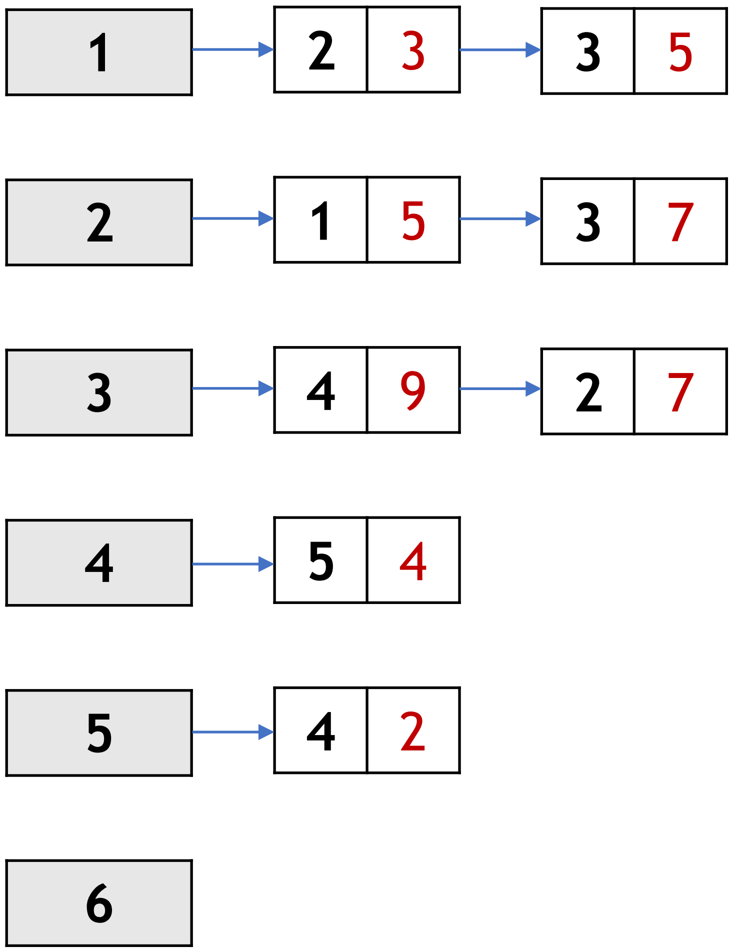


Graph: Adjacency List

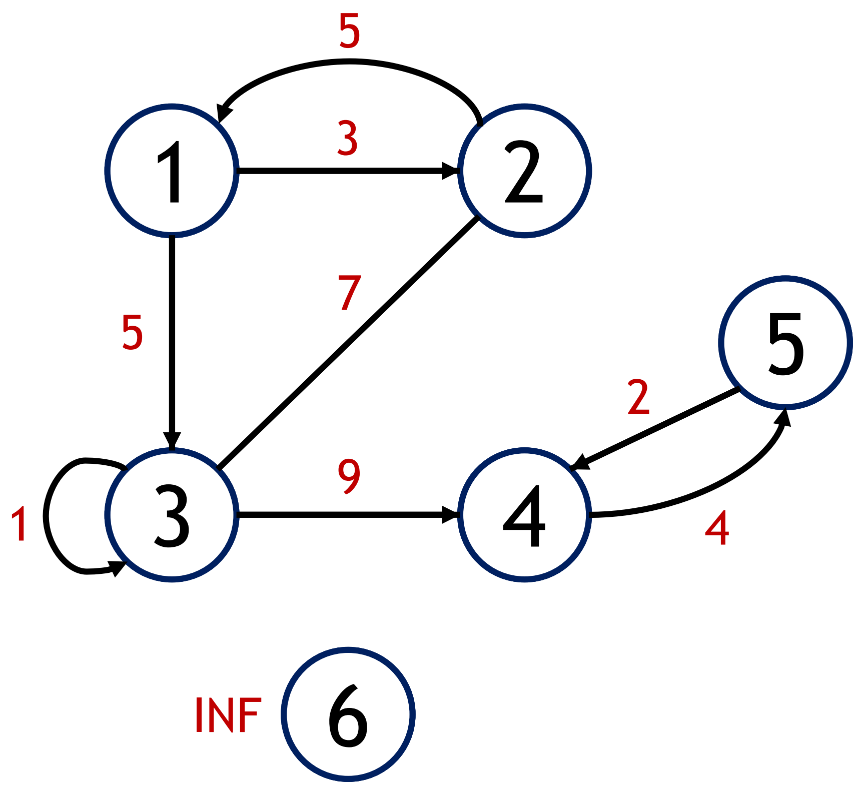


Weighted, Directed

next field is hidden

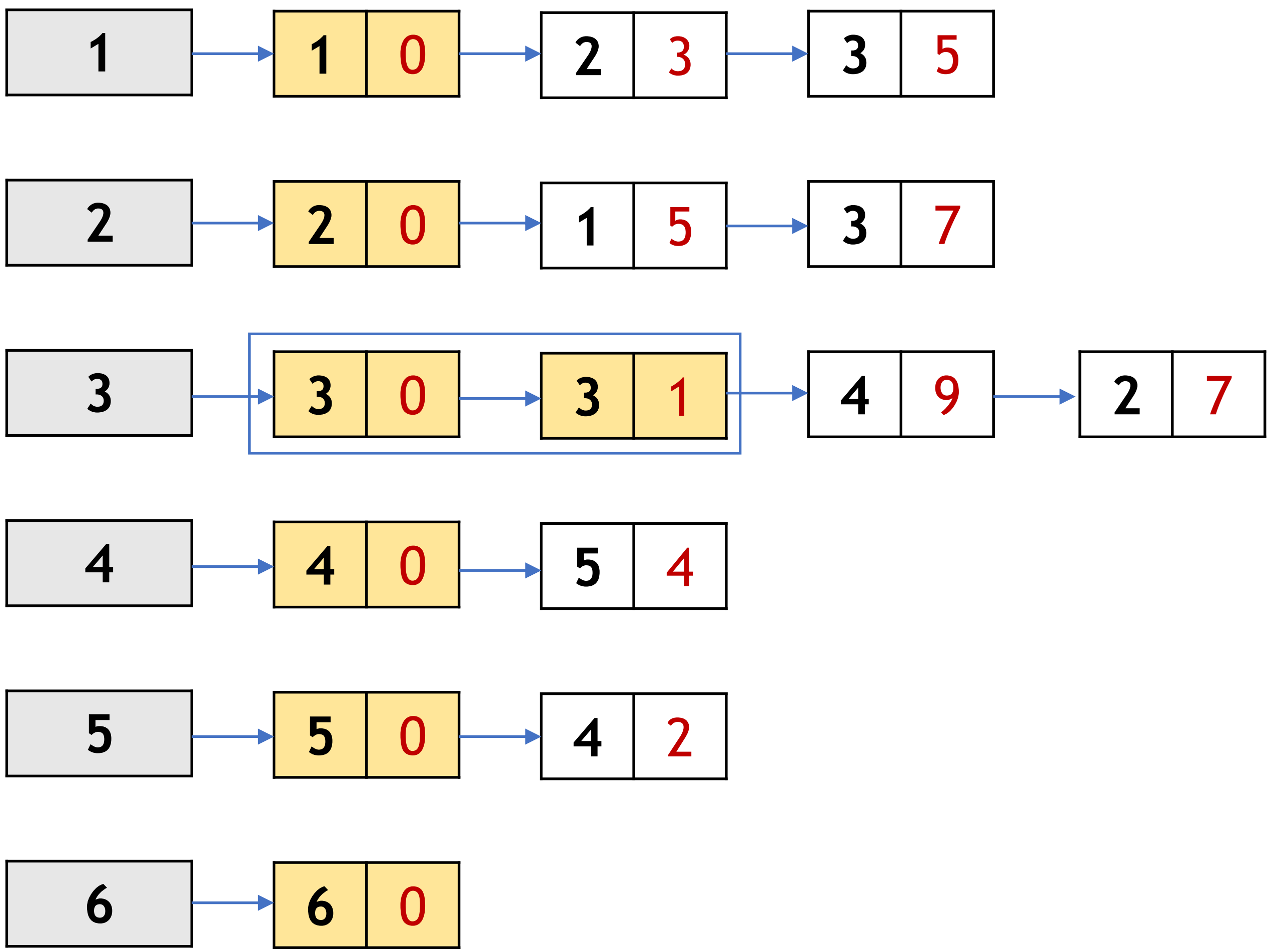


Graph: Adjacency List



Weighted, Directed,
Multiple edges, Self-loops

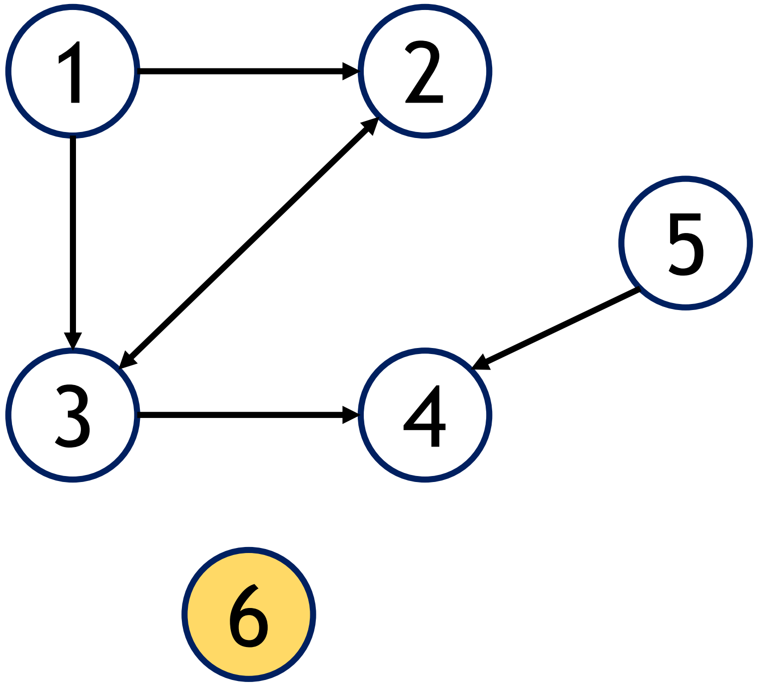
*next field is hidden



Graph: Adjacency Matrix

Insertion, Deletion

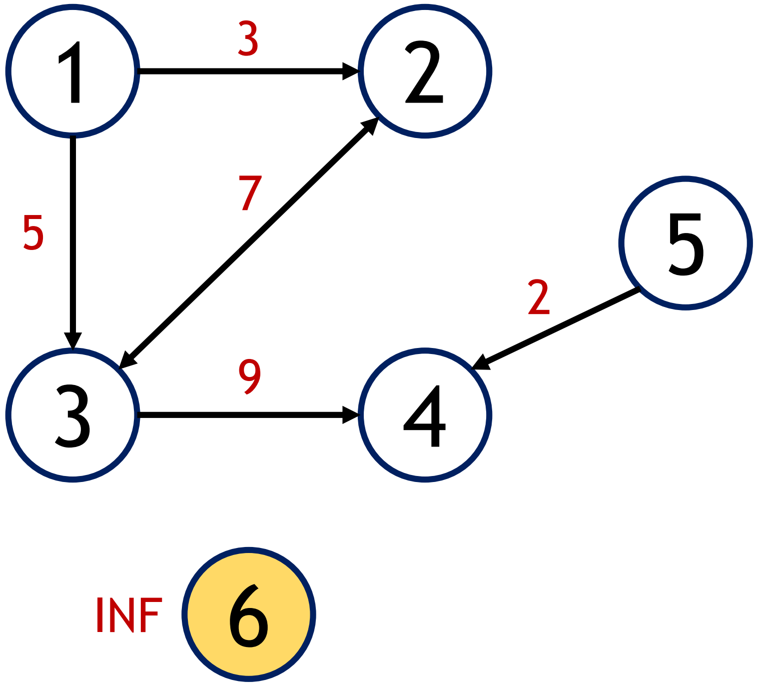
Adjacency Matrix: Inserting a Node



Directed, Unweighted

0	1	1	0	0	0
0	0	1	0	0	0
0	1	0	1	0	0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

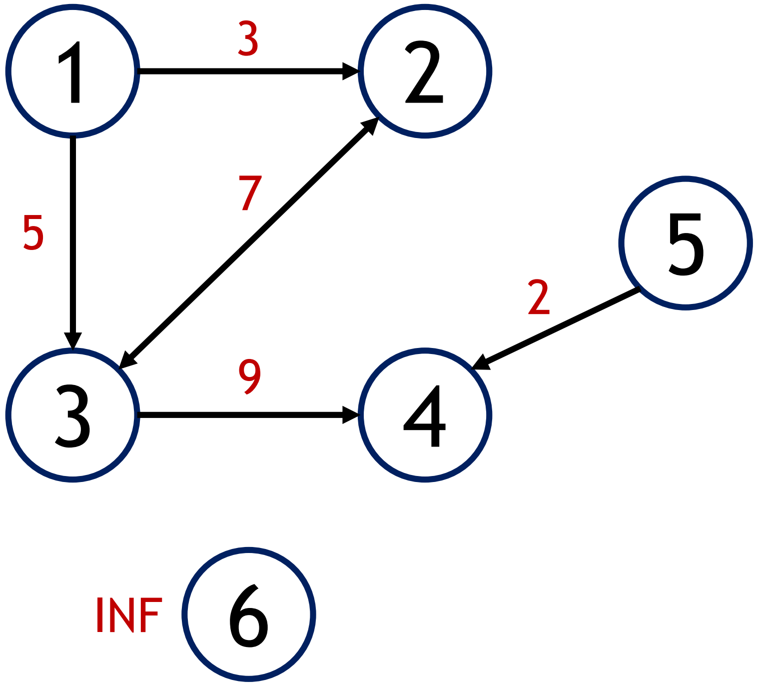
Adjacency Matrix: Inserting a Node



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

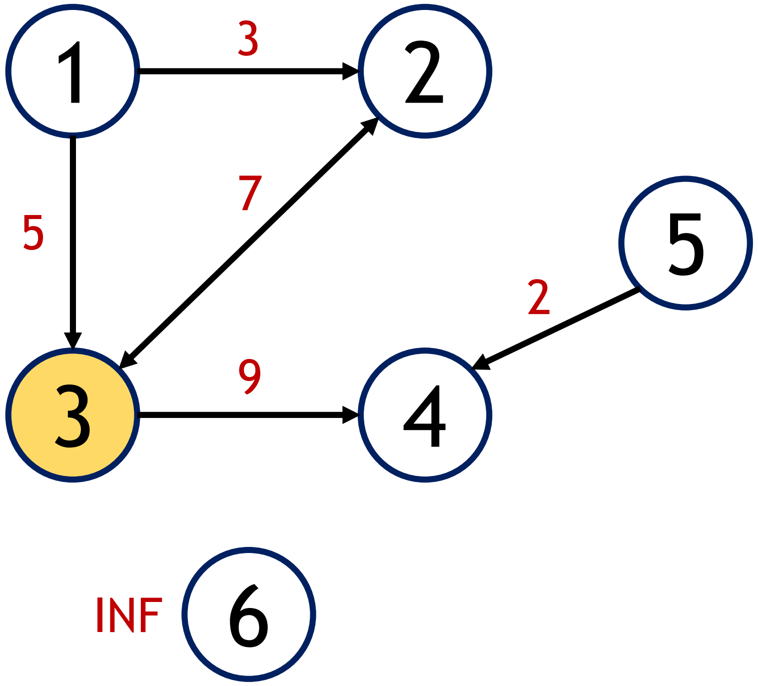
Adjacency Matrix: Deleting a Node



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

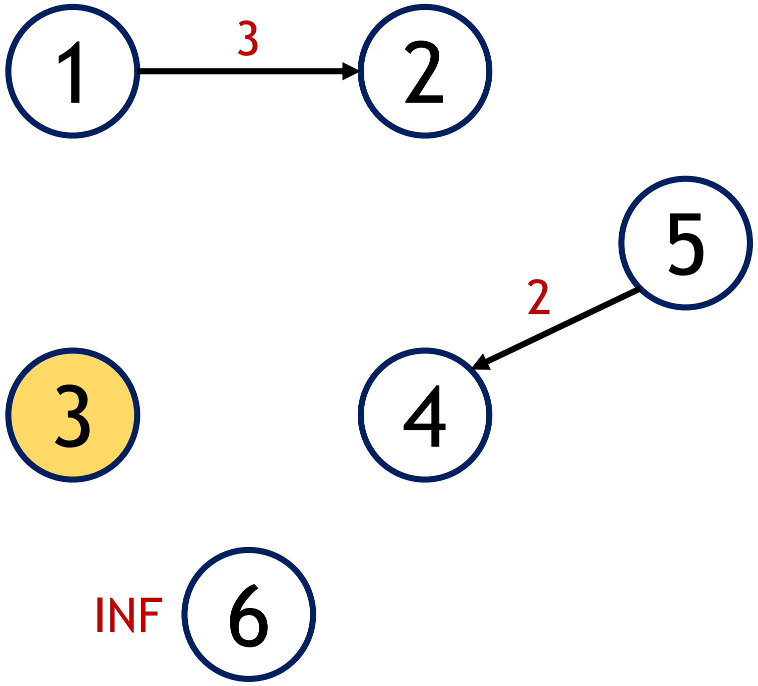
Adjacency Matrix: Deleting a Node



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

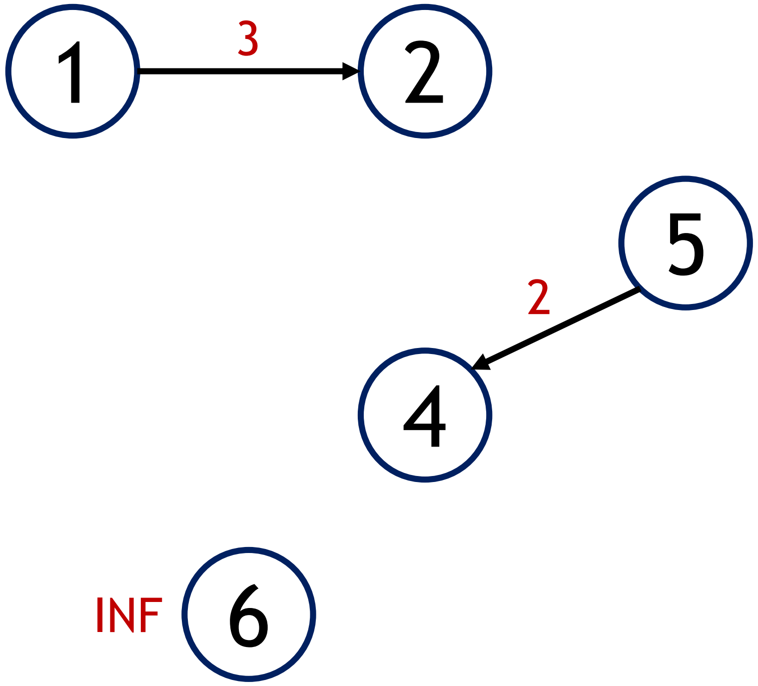
Adjacency Matrix: Deleting a Node



Directed, Weighted

0	3	∞	∞	∞	∞
∞	0	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

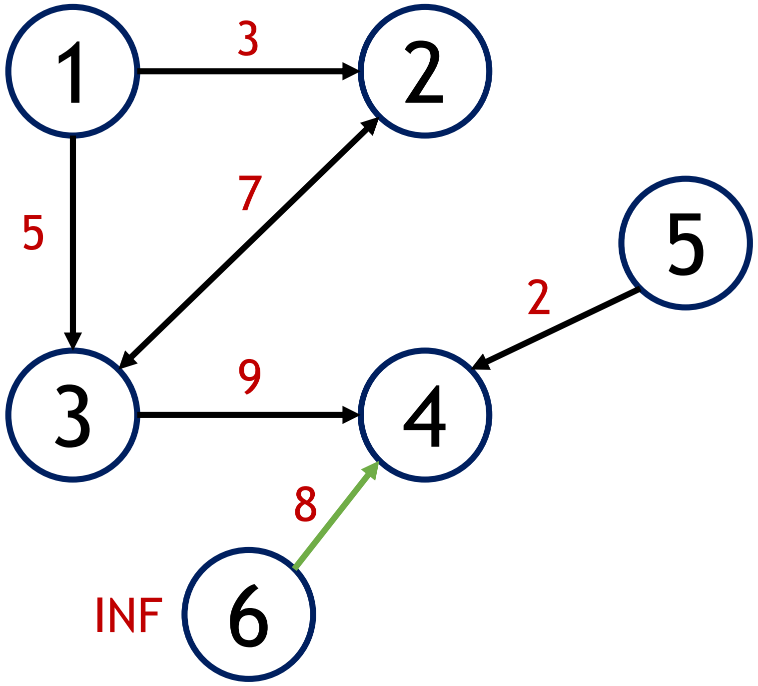
Adjacency Matrix: Deleting a Node



Directed, Weighted

0	3	0	0	∞
0	0	0	0	∞
0	0	0	0	∞
0	0	2	0	∞
∞	∞	∞	∞	0

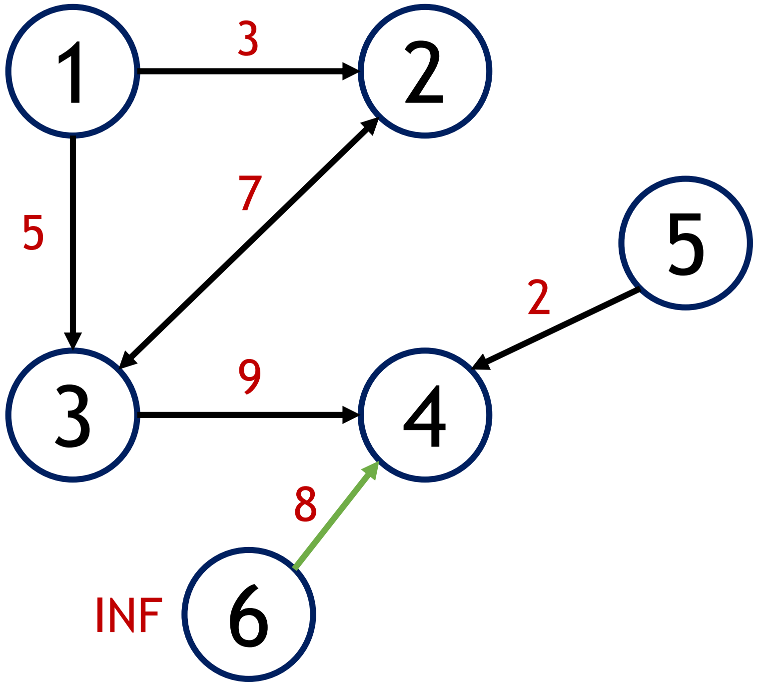
Adjacency Matrix: Inserting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

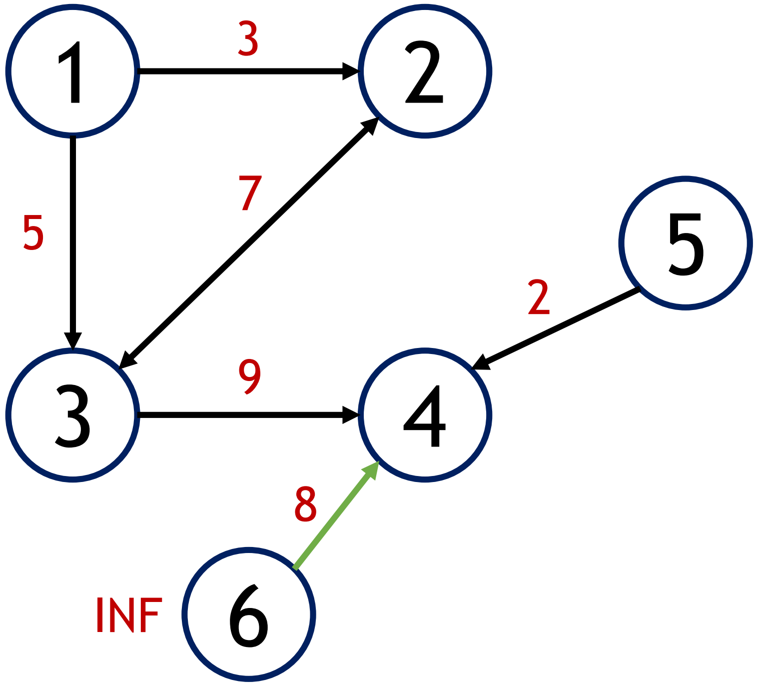
Adjacency Matrix: Inserting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	∞	∞	0

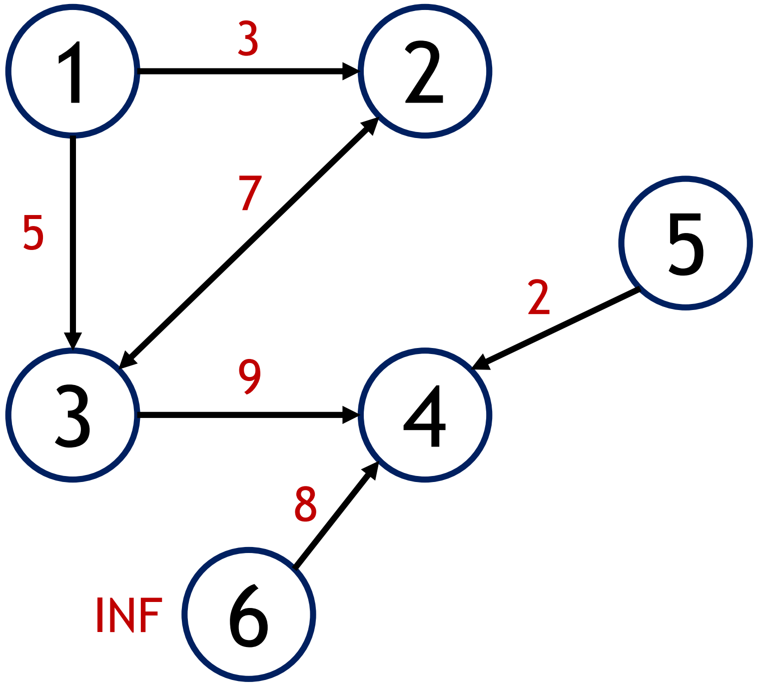
Adjacency Matrix: Inserting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	8	∞	0

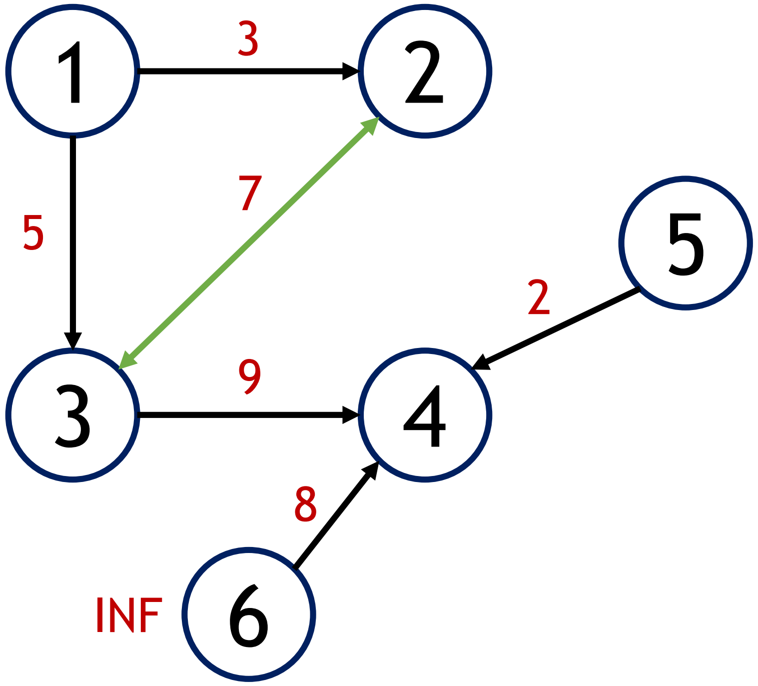
Adjacency Matrix: Deleting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	8	∞	0

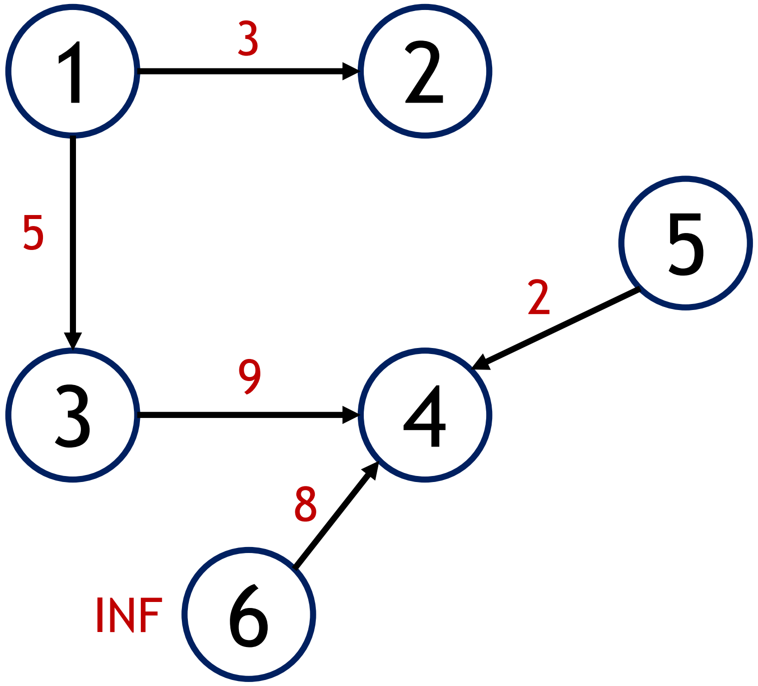
Adjacency Matrix: Deleting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	7	∞	∞	∞
∞	7	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	8	∞	0

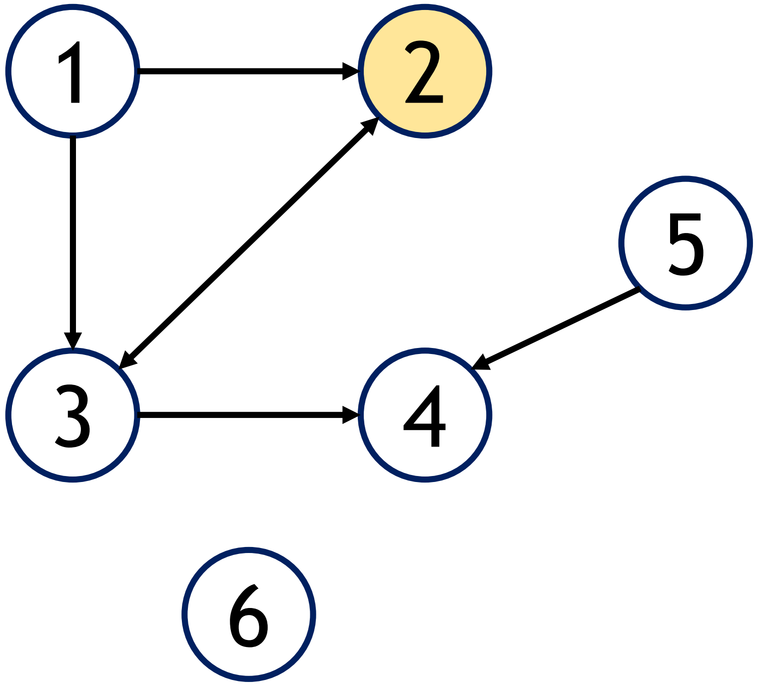
Adjacency Matrix: Deleting an Edge



Directed, Weighted

0	3	5	∞	∞	∞
∞	0	∞	∞	∞	∞
∞	∞	0	9	∞	∞
∞	∞	∞	0	∞	∞
∞	∞	∞	2	0	∞
∞	∞	∞	8	∞	0

Adjacency Matrix: Note



Directed, Unweighted

0	1	1	0	0	0
0	0	1	0	0	0
0	1	0	1	0	0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

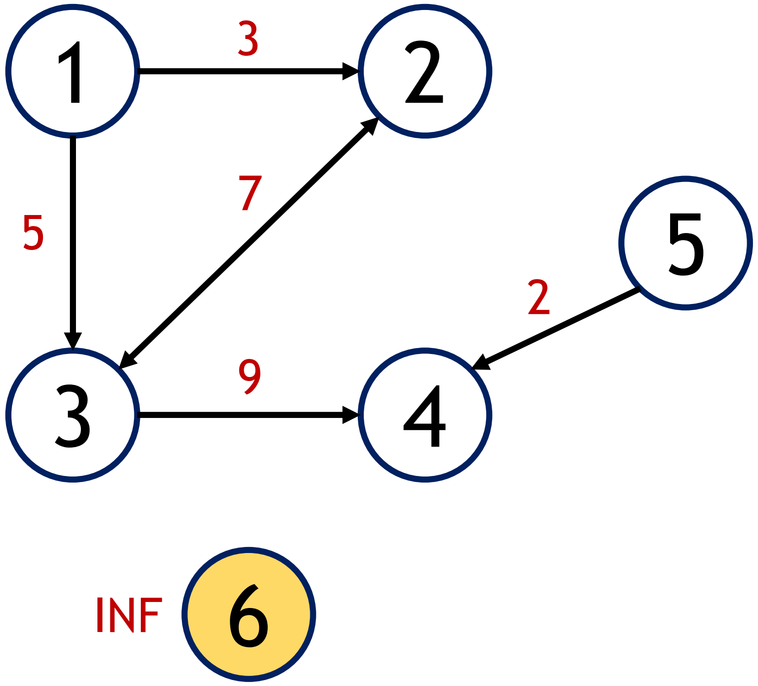
$\sum_j^n A_{i,j}$ = **# outgoing edges from node *i***

$\sum_j^n A_{j,i}$ = **# incoming edges at node *i***

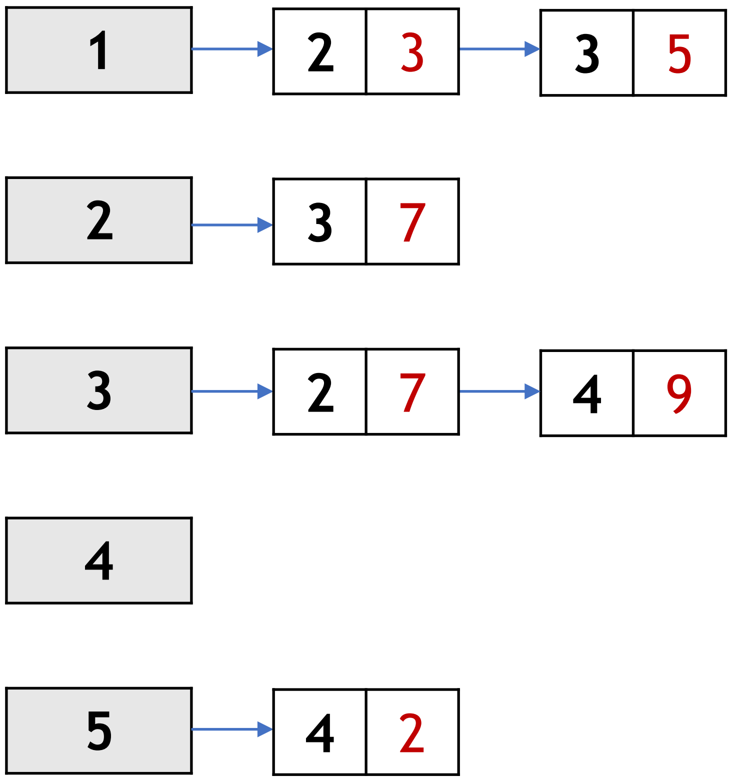
Graph: Adjacency List

Insertion, Deletion

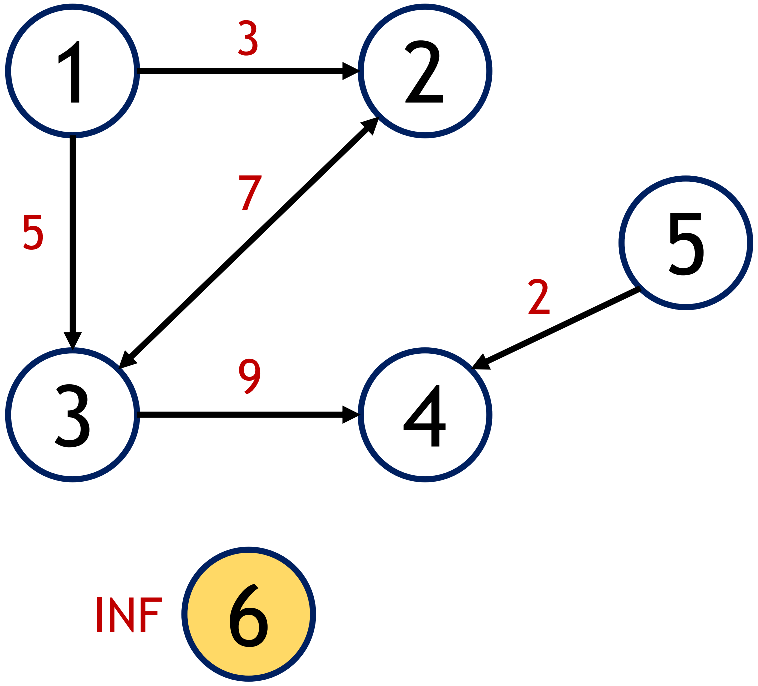
Adjacency List: Inserting a Node



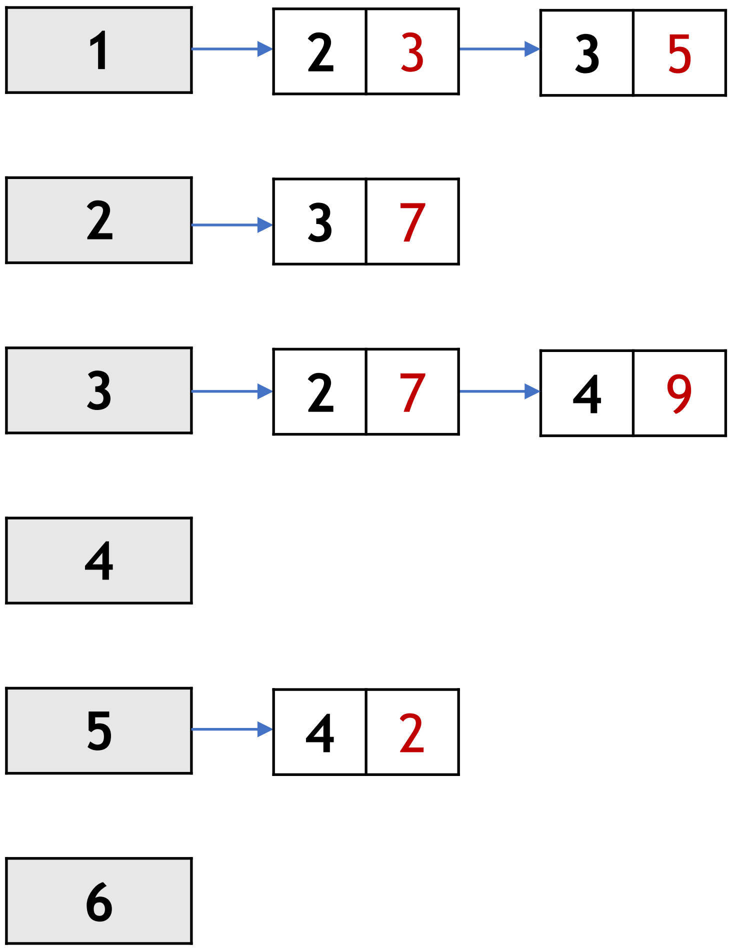
Directed, Weighted



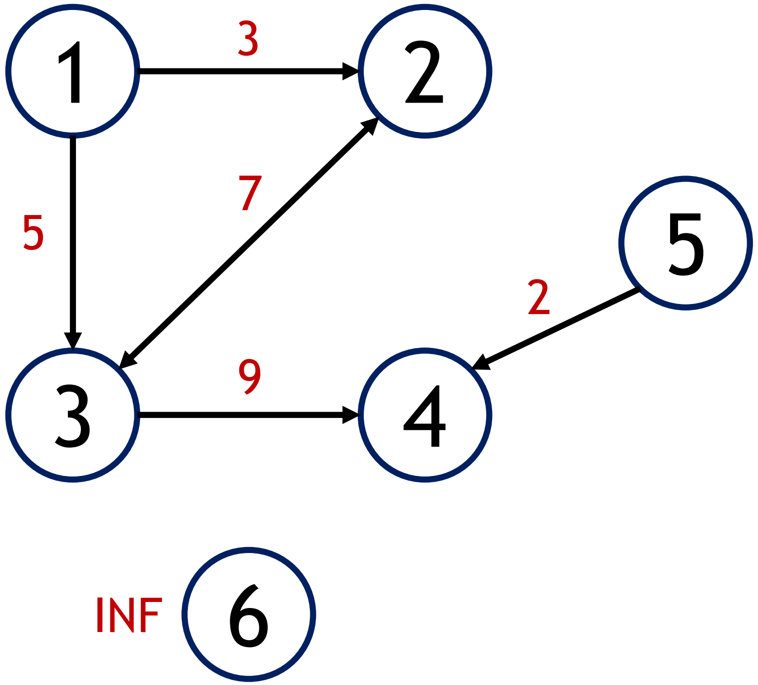
Adjacency List: Inserting a Node



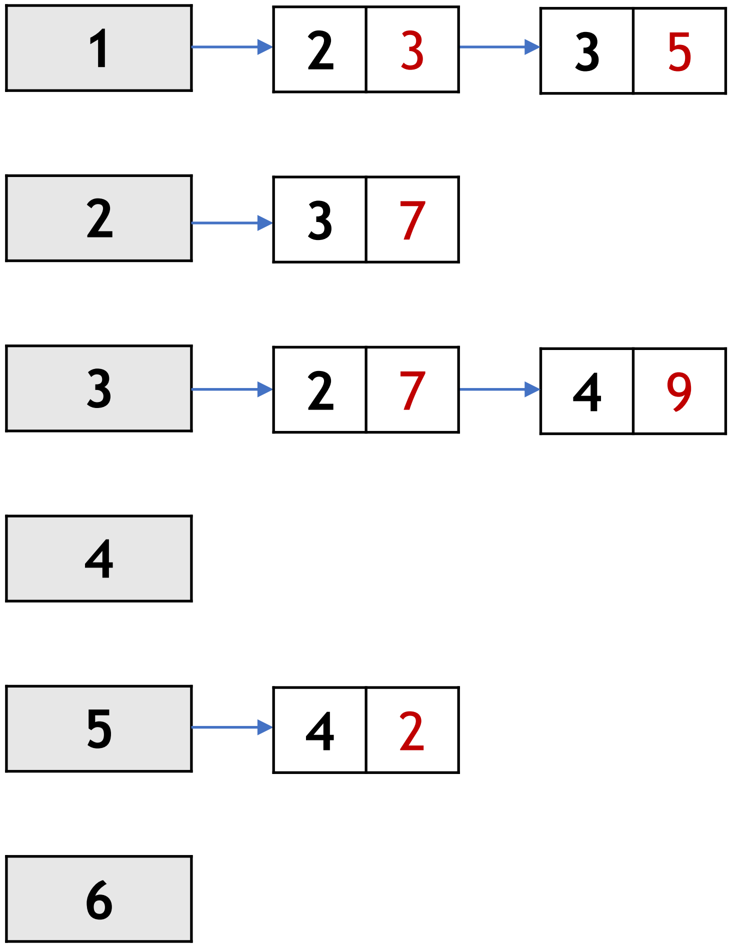
Directed, Weighted



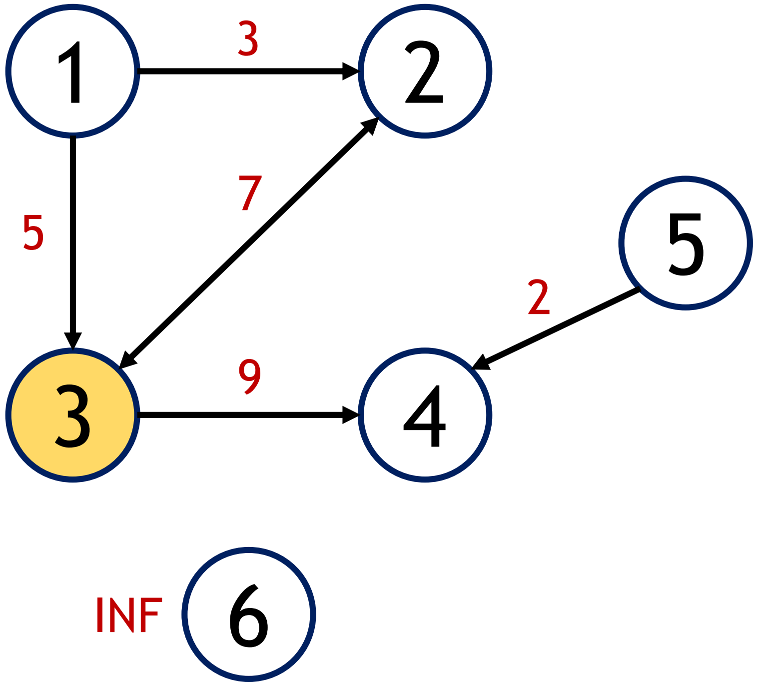
Adjacency List: Deleting a Node



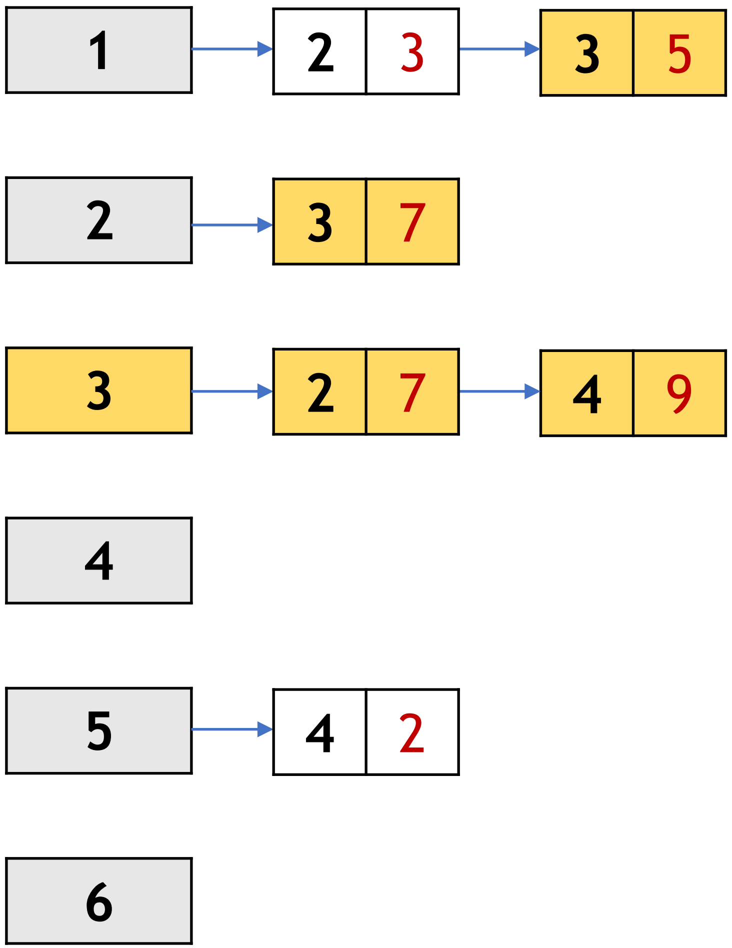
Directed, Weighted



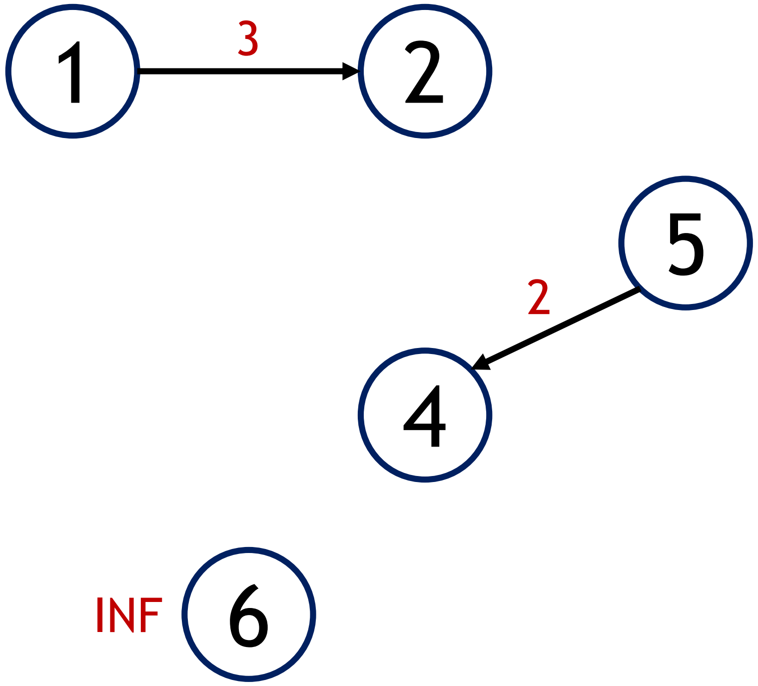
Adjacency List: Deleting a Node



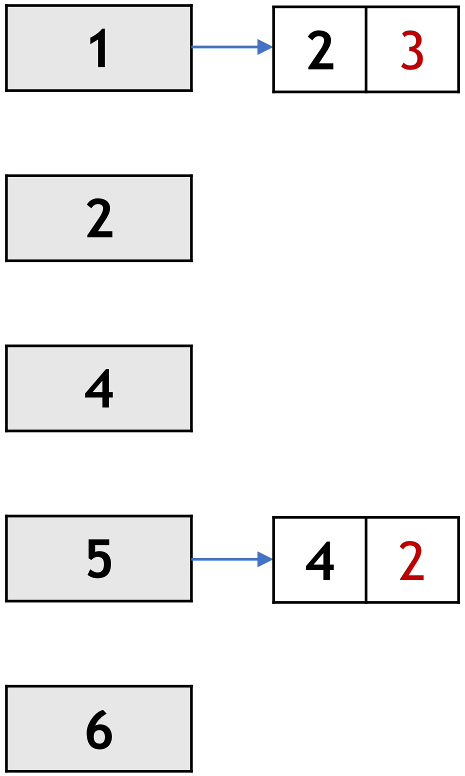
Directed, Weighted



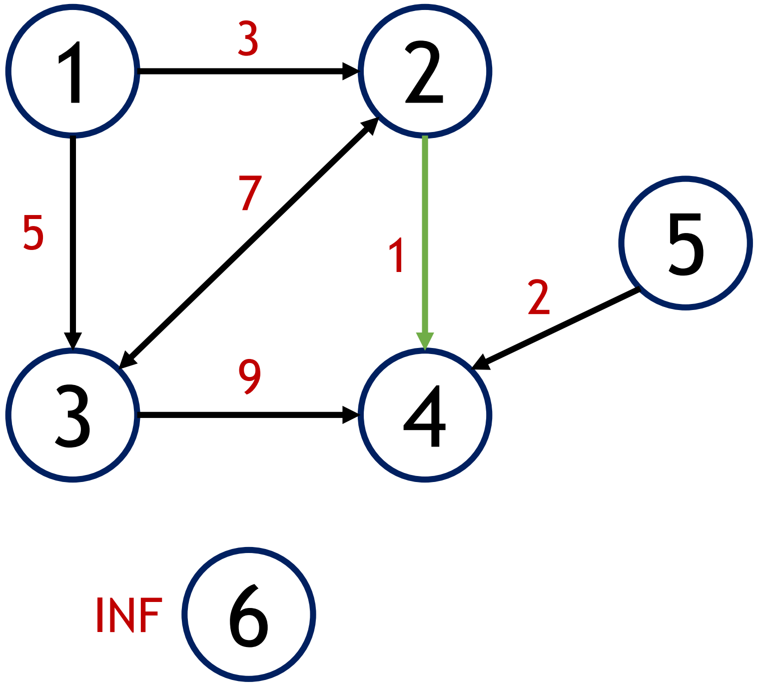
Adjacency List: Deleting a Node



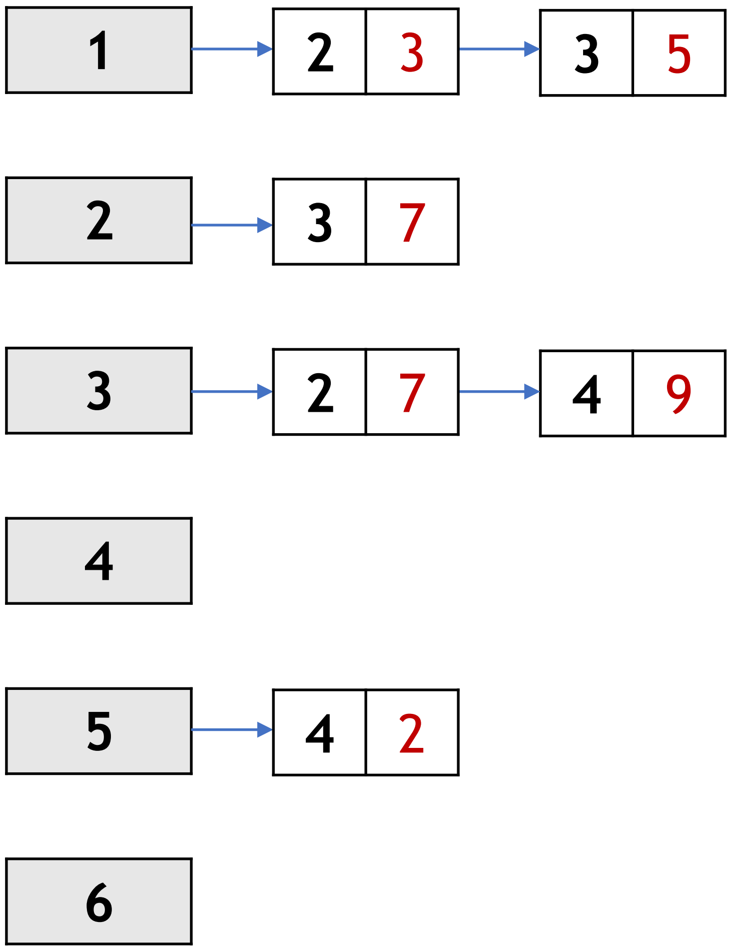
Directed, Weighted



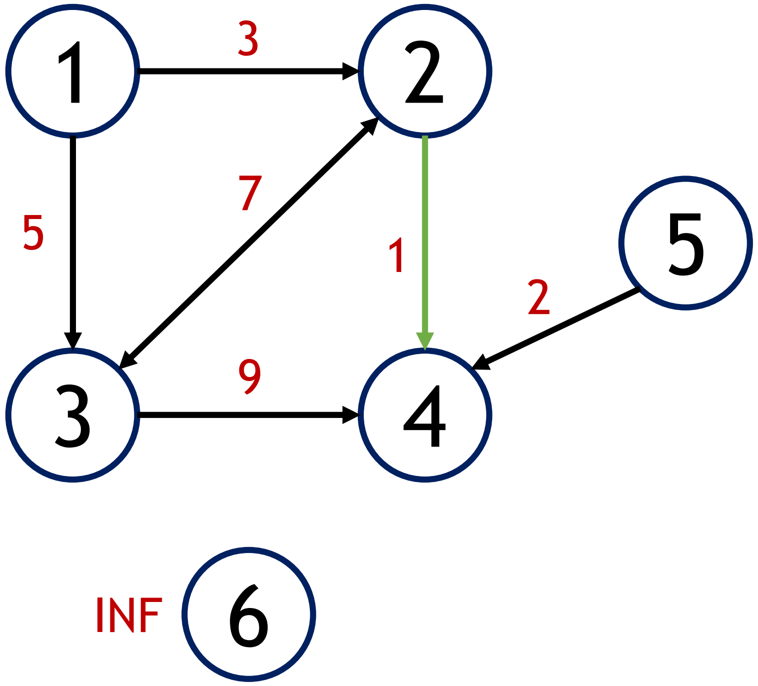
Adjacency List: Inserting an Edge



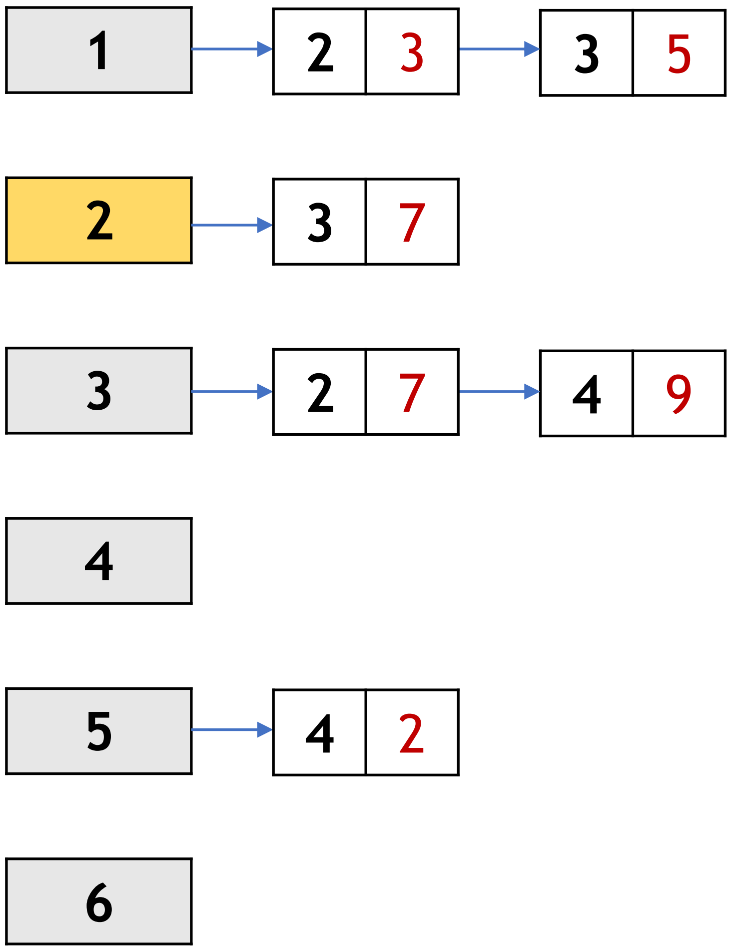
Directed, Weighted



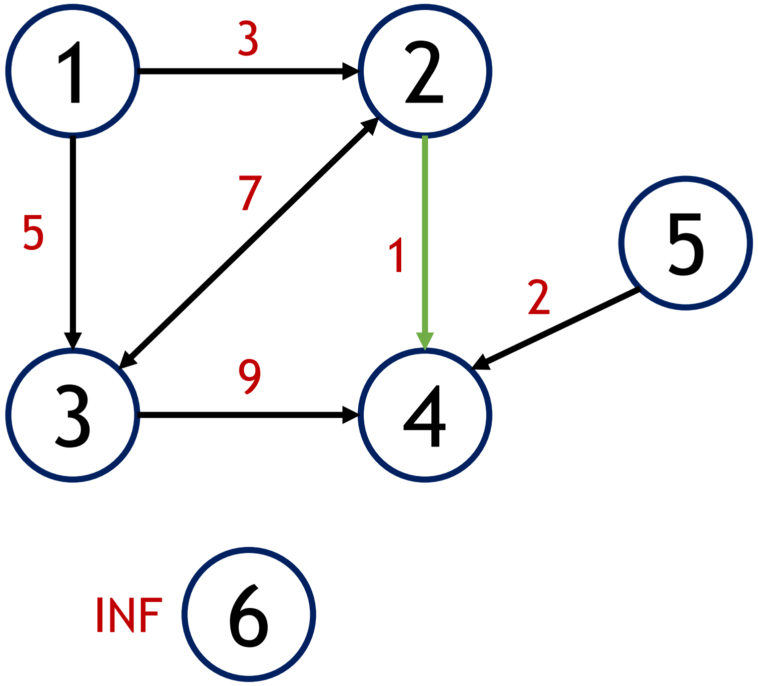
Adjacency List: Inserting an Edge



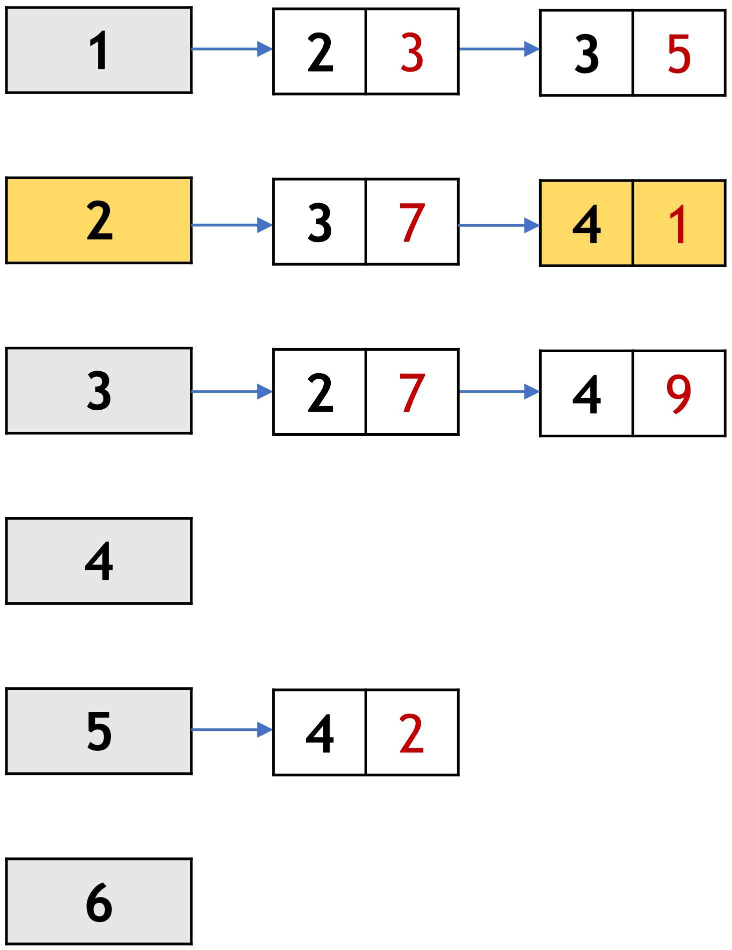
Directed, Weighted



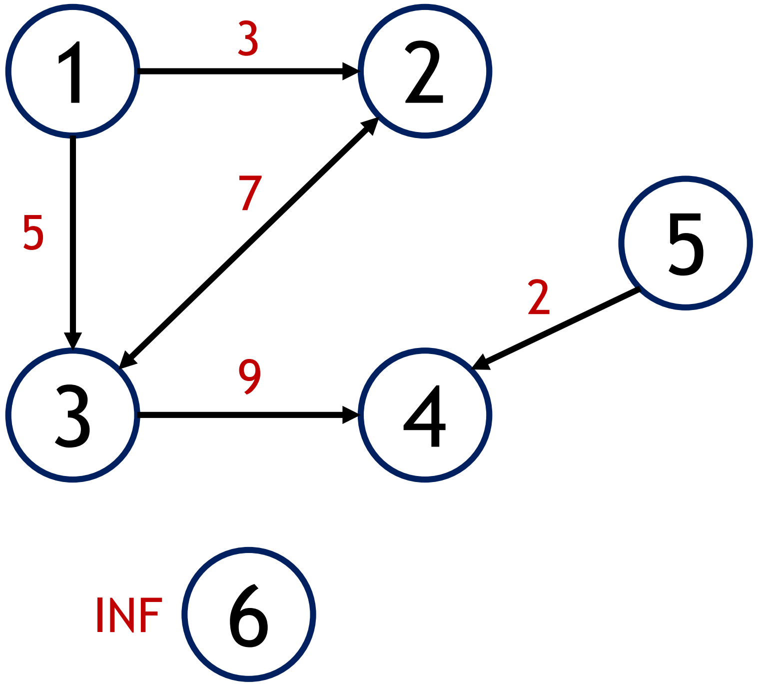
Adjacency List: Inserting an Edge



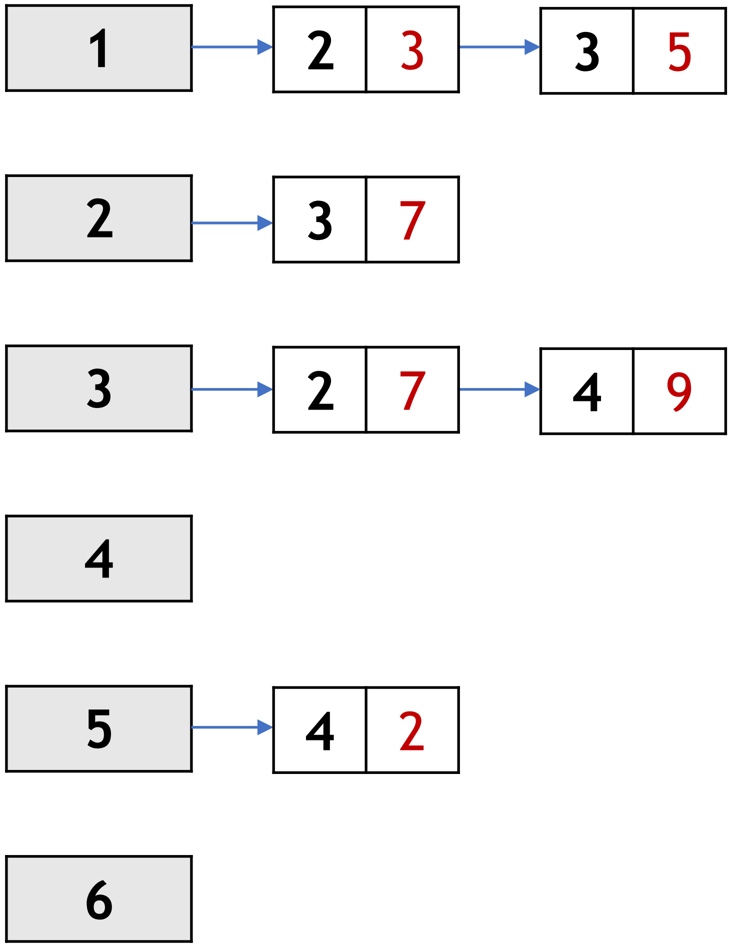
Directed, Weighted



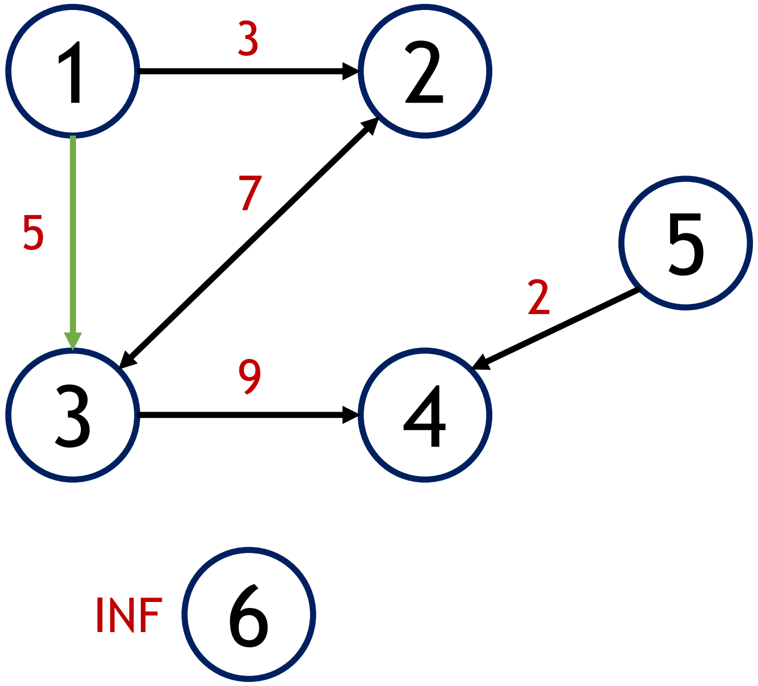
Adjacency List: Deleting an Edge



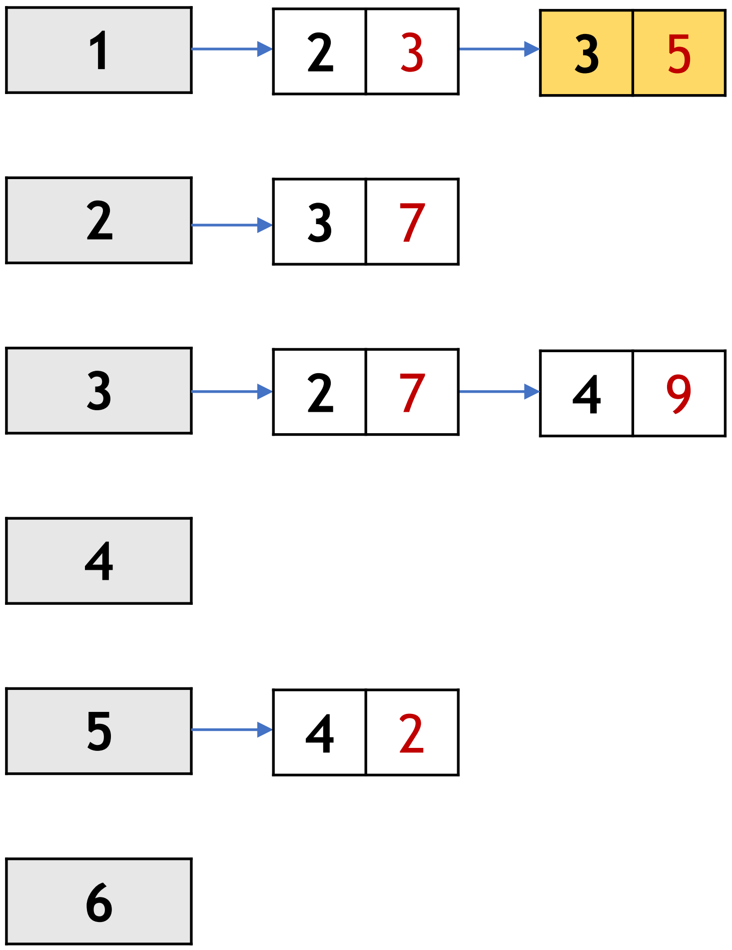
Directed, Weighted



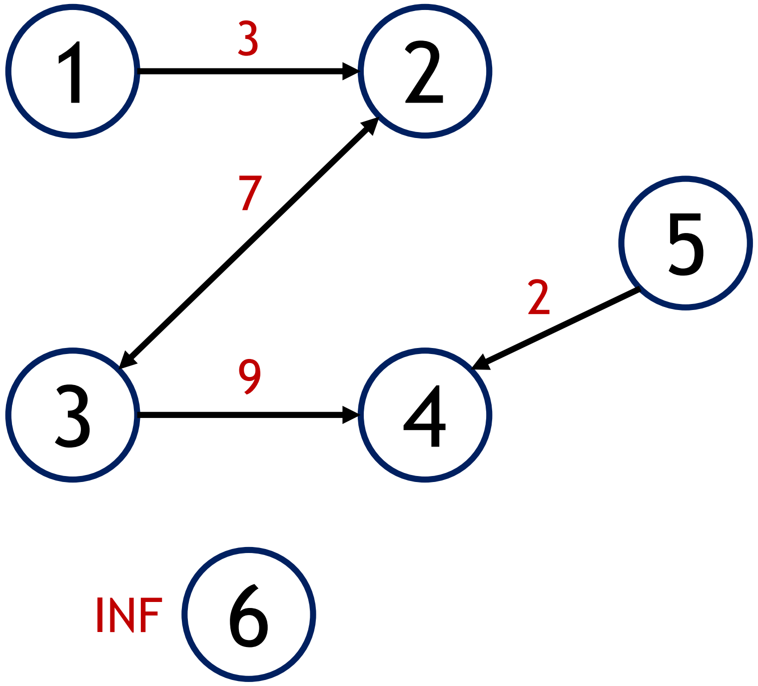
Adjacency List: Deleting an Edge



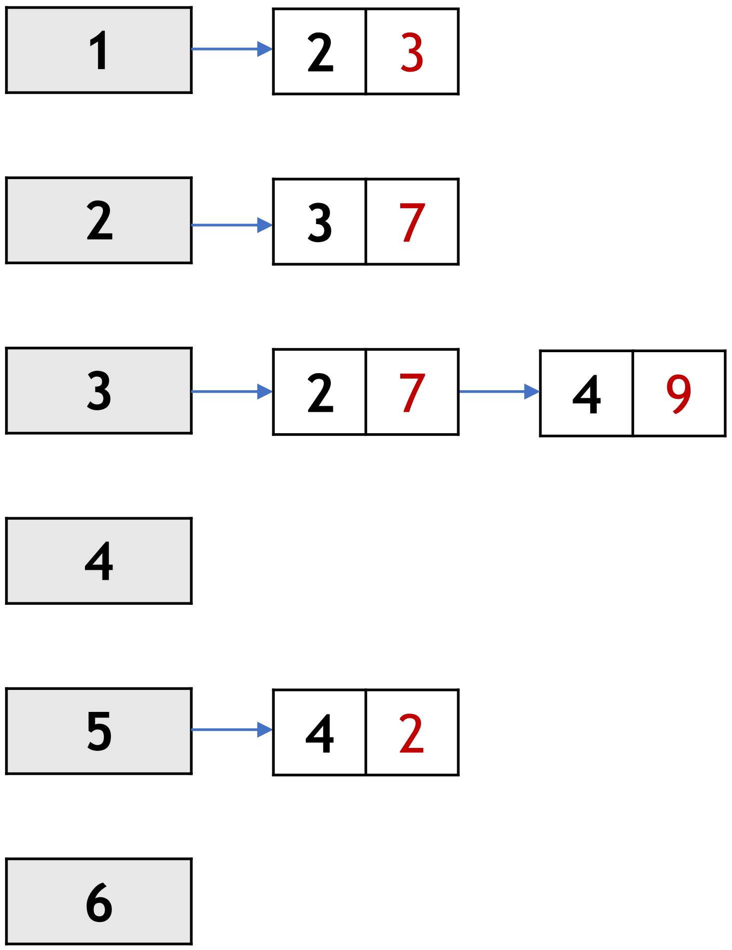
Directed, Weighted



Adjacency List: Deleting an Edge



Directed, Weighted



Exercise

Exercise: Note

```
struct vertex;

struct adjVertex{
  vertex *v;
};

struct vertex{
  std::string name;
  std::vector<adjVertex> adj;
};
```

Type of **`vertices`**: `vector<vertex*>`

Vertex at some index:

`vertices[i]` : type vertex*

Dereferencing a pointer (vertex*): `->`

Dereferencing a struct (adjVertex): `.`

Careful with dereferencing!

Exercise: Silver

Implement:

```
void Graph::printGraph()
```

Notes

1. You need to print each vertex and its adjacent vertices
2. Order of vertices does not matter
3. Vector methods