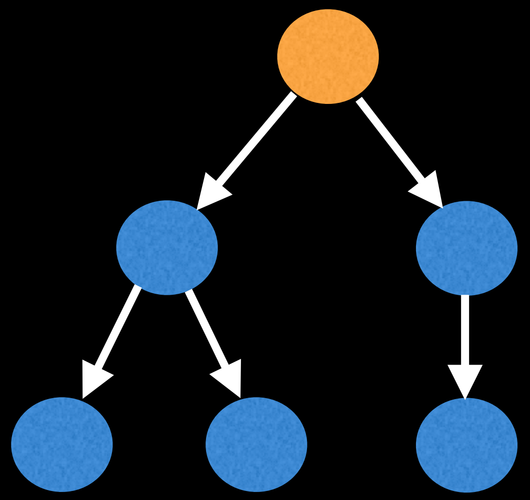
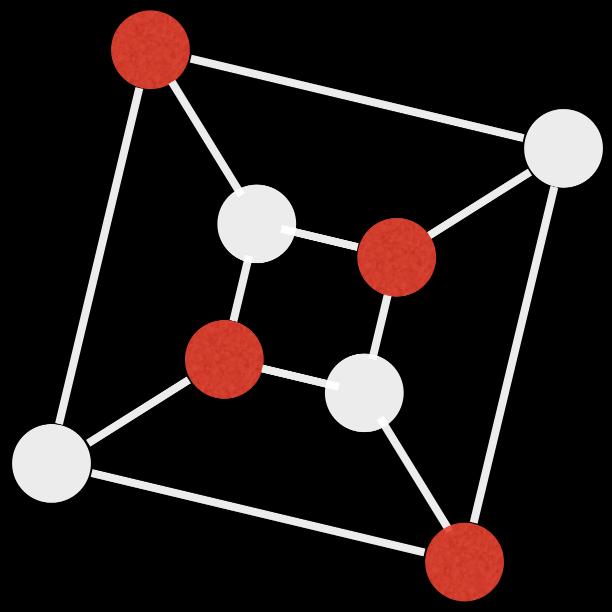


Graph Theory Intro & Overview



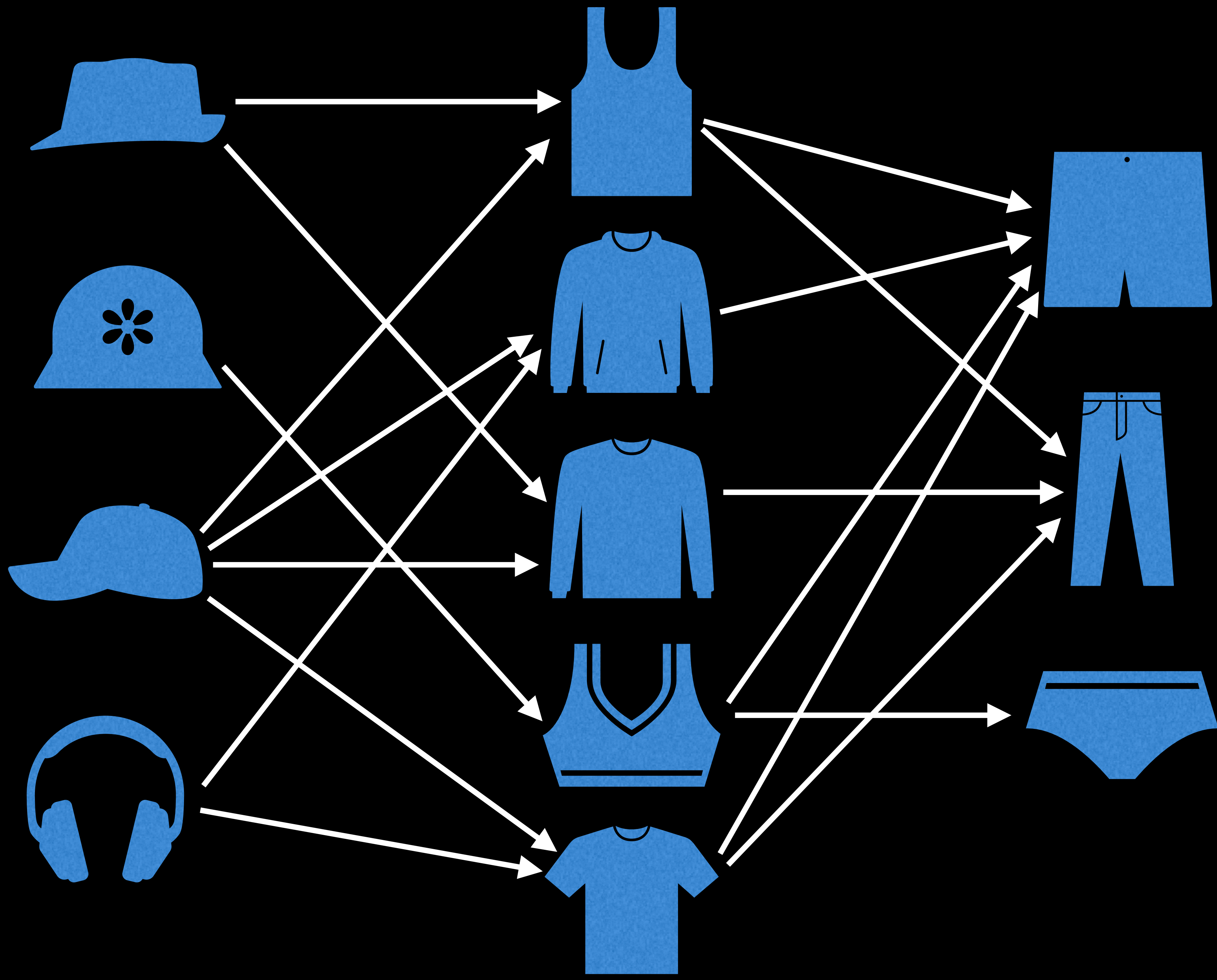
William Fiset

Brief introduction

Graph theory is the mathematical theory of the properties and applications of graphs (networks).

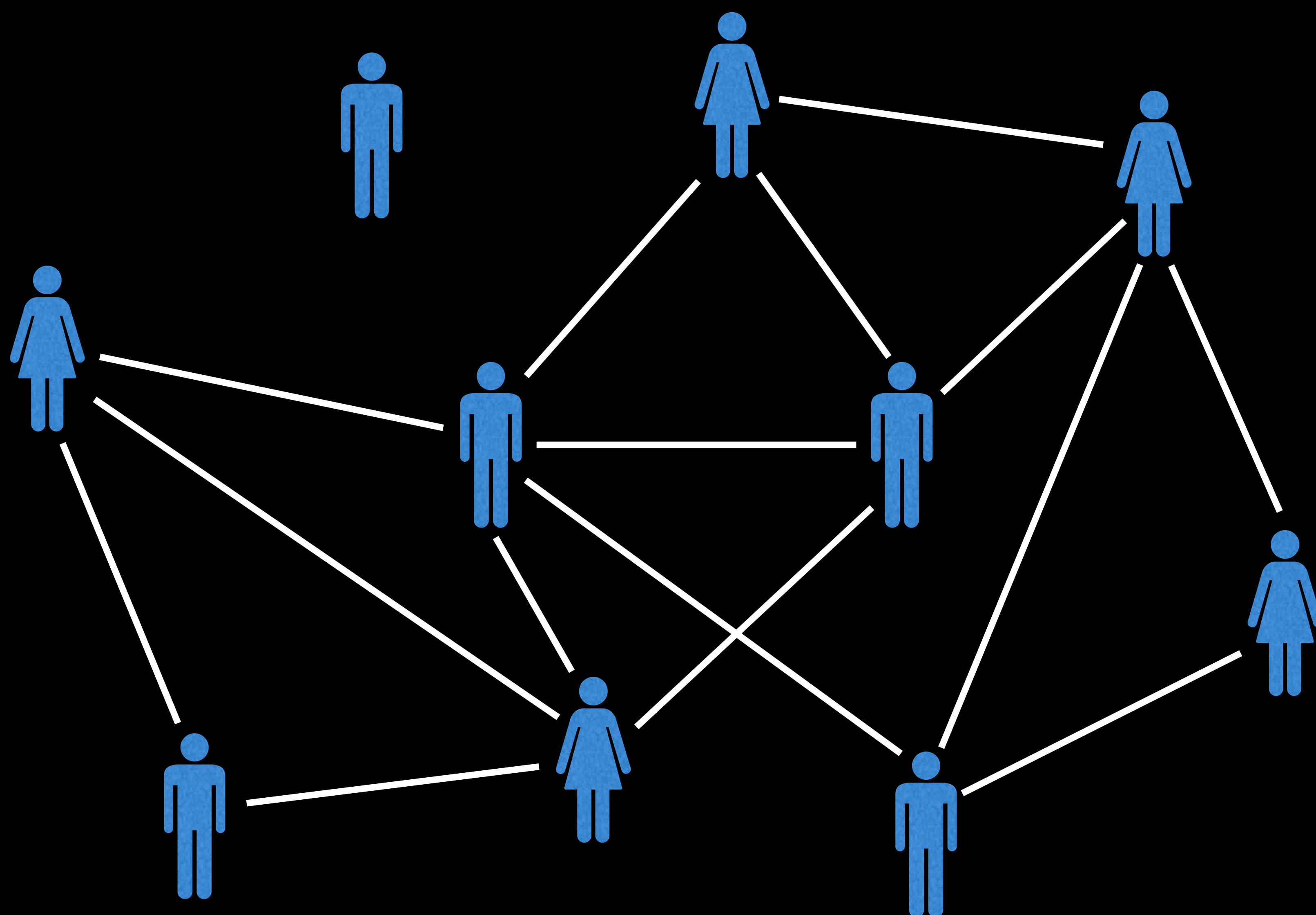
The goal of this series is to gain an understanding of how to apply graph theory to real world applications.

Brief introduction



A graph theory problem might be:
Given the constraints above, how many different sets of clothing can I make by choosing an article from each category?

Brief introduction



The canonical graph theory example is a social network of friends.

This enables interesting questions such as: how many friends does person X have? Or how many degrees of separation are there between person X and person Y?

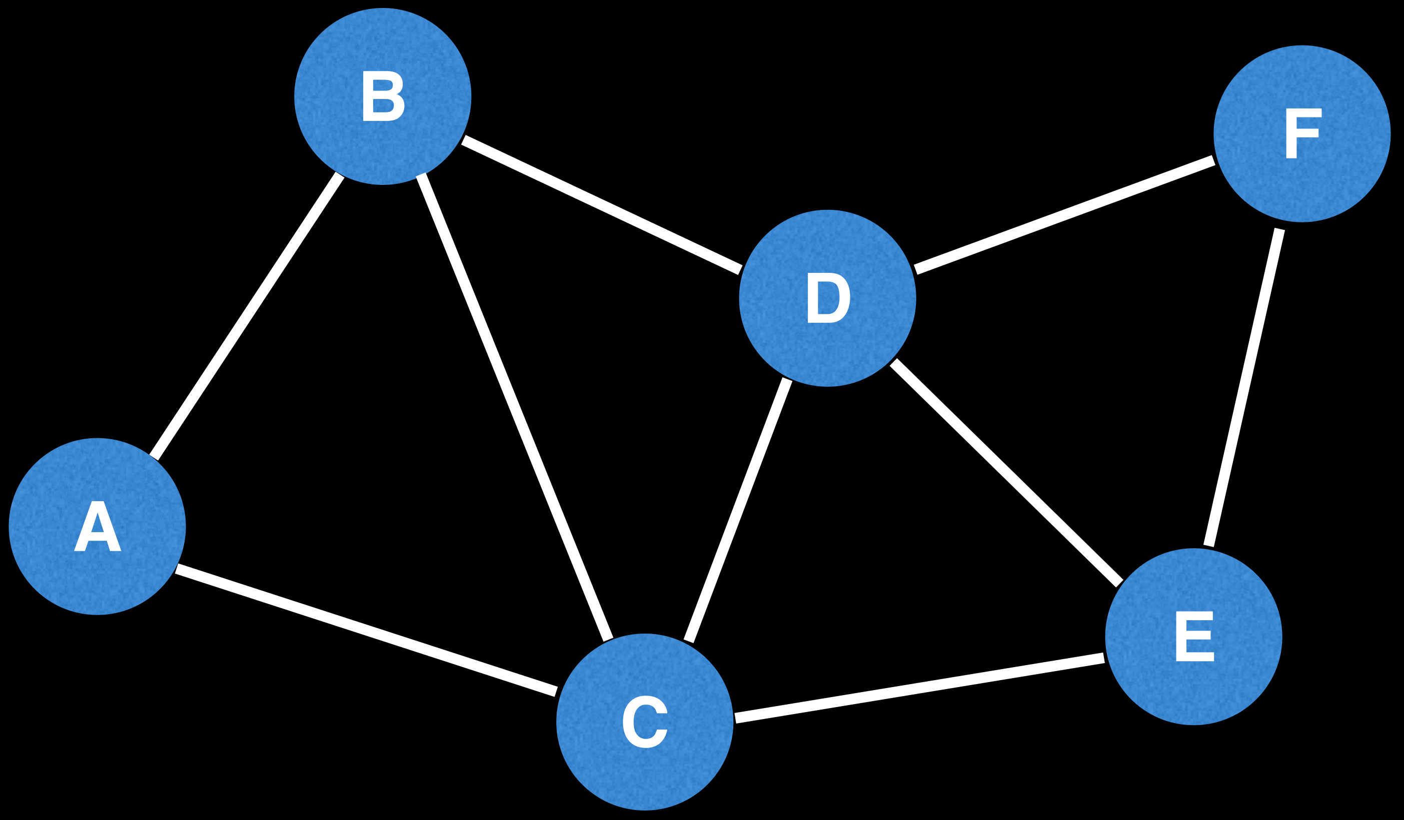
Types of Graphs

Undirected Graph

An **undirected graph** is a graph in which edges have no orientation. The edge (u, v) is identical to the edge (v, u). – Wiki

Undirected Graph

An **undirected graph** is a graph in which edges have no orientation. The edge (*u*, *v*) is identical to the edge (*v*, *u*). – Wiki



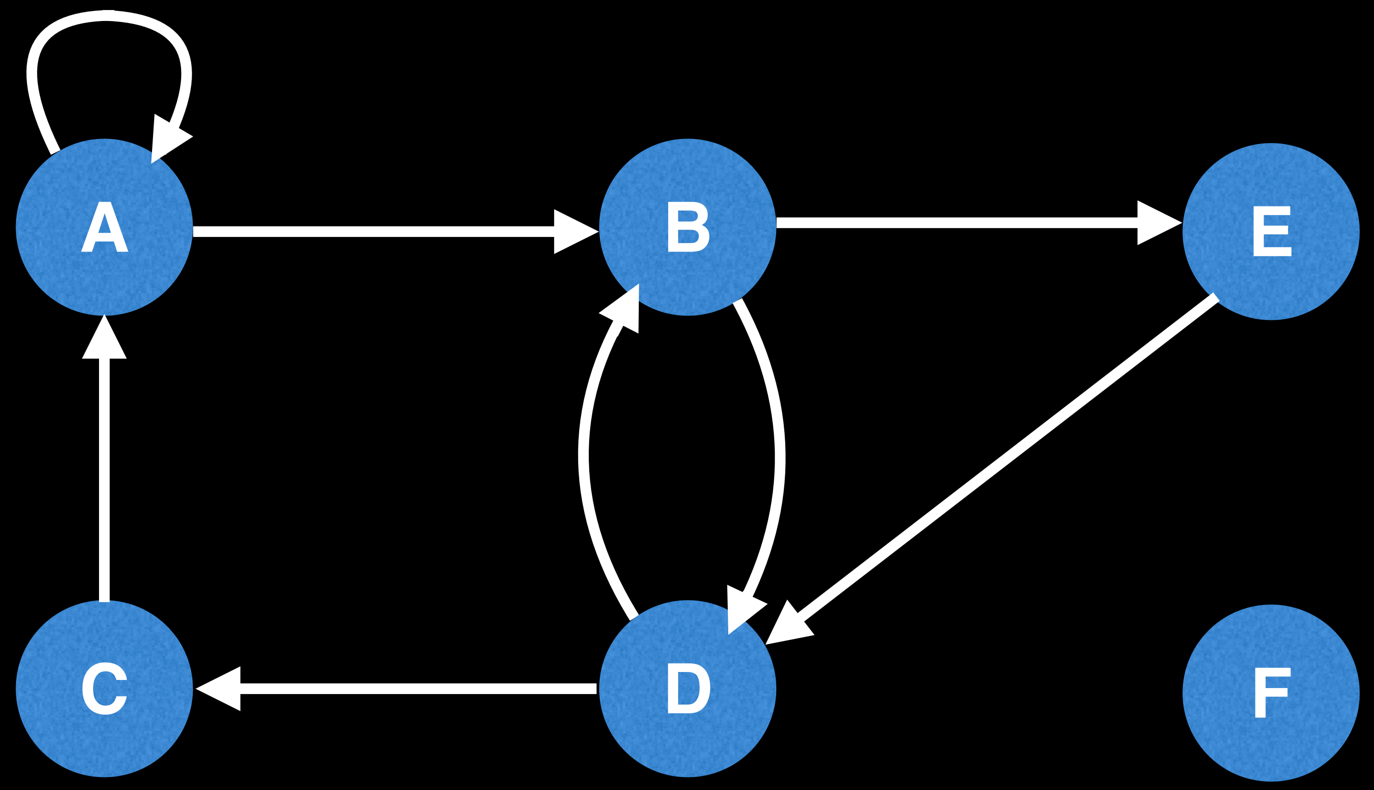
In the graph above, the nodes could represent cities and an edge could represent a bidirectional road.

Directed Graph (Digraph)

A ***directed graph*** or ***digraph*** is a graph in which edges have orientations. For example, the edge (u, v) is the edge *from* node u *to* node v.

Directed Graph (Digraph)

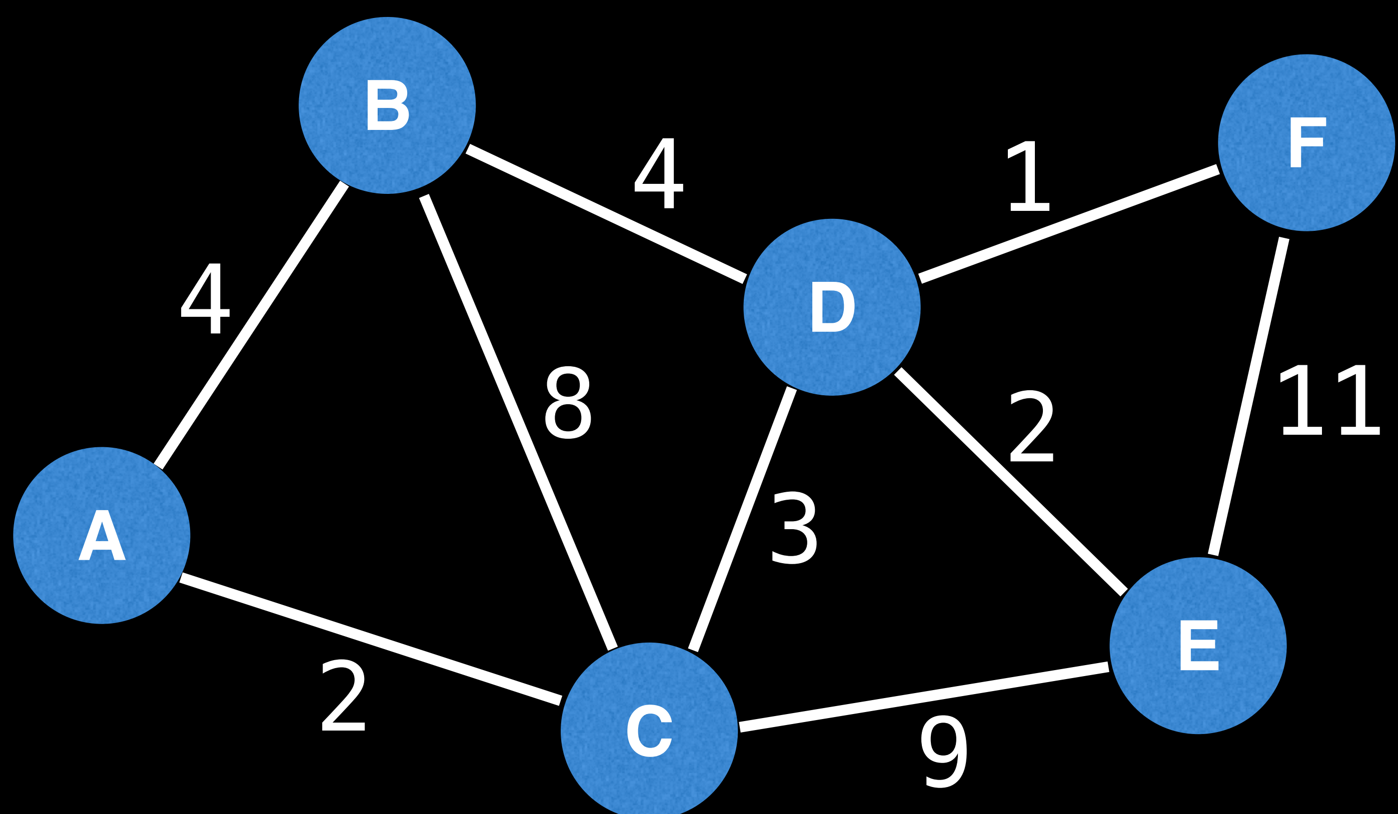
A ***directed graph*** or ***digraph*** is a graph in which edges have orientations. For example, the edge (u, v) is the edge *from* node u *to* node v.



In the graph above, the nodes could represent people and an edge (u, v) could represent that person u bought person v a gift.

Weighted Graphs

Many graphs can have edges that contain a certain weight to represent an arbitrary value such as cost, distance, quantity, etc…

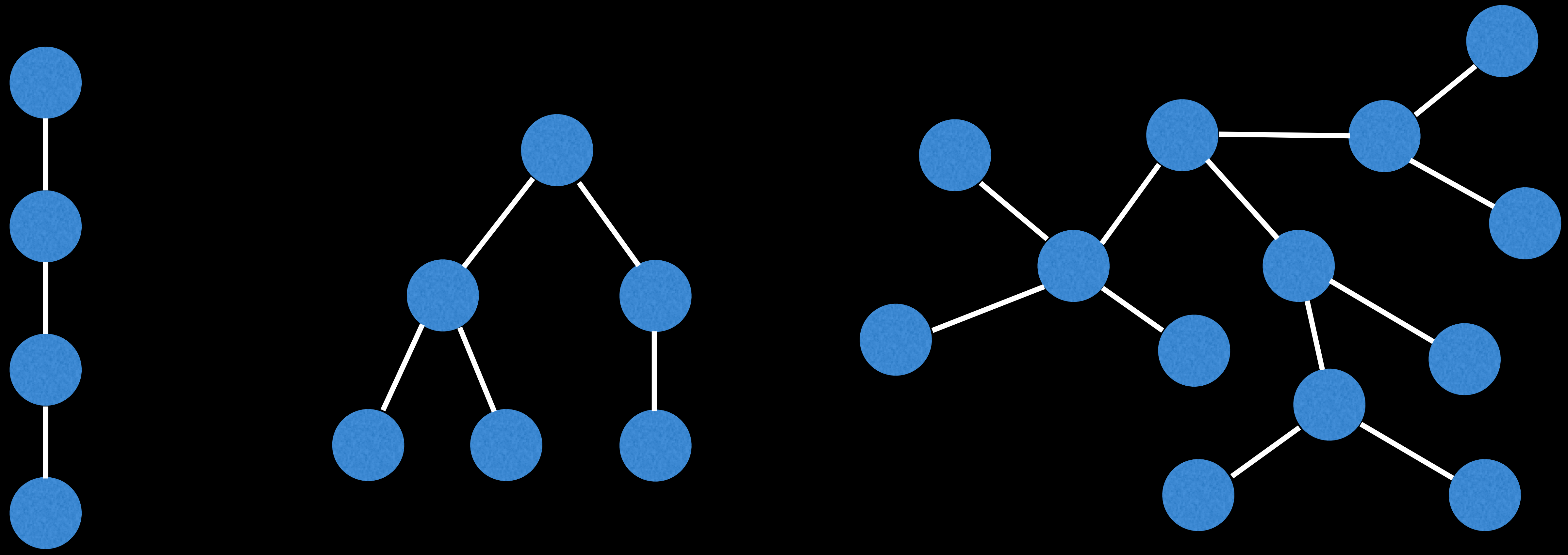


NOTE: I will usually denote an edge of such a graph as a triplet (u, v, w) and specify whether the graph is directed or undirected.

Special Graphs

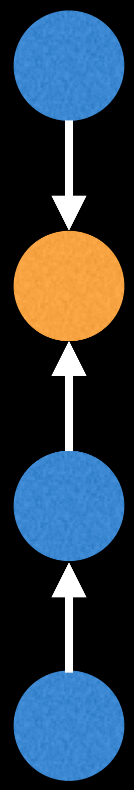
Trees!

A **tree** is an **undirected connected graph with no cycles**. Equivalently, it is a connected graph with N nodes and N−1 edges.

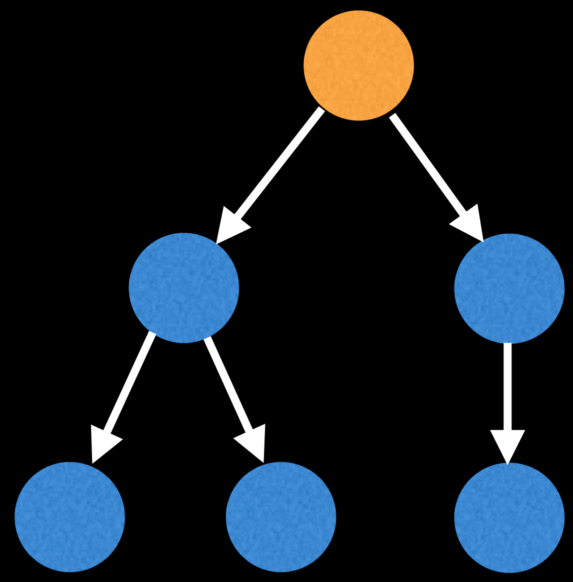


Rooted Trees!

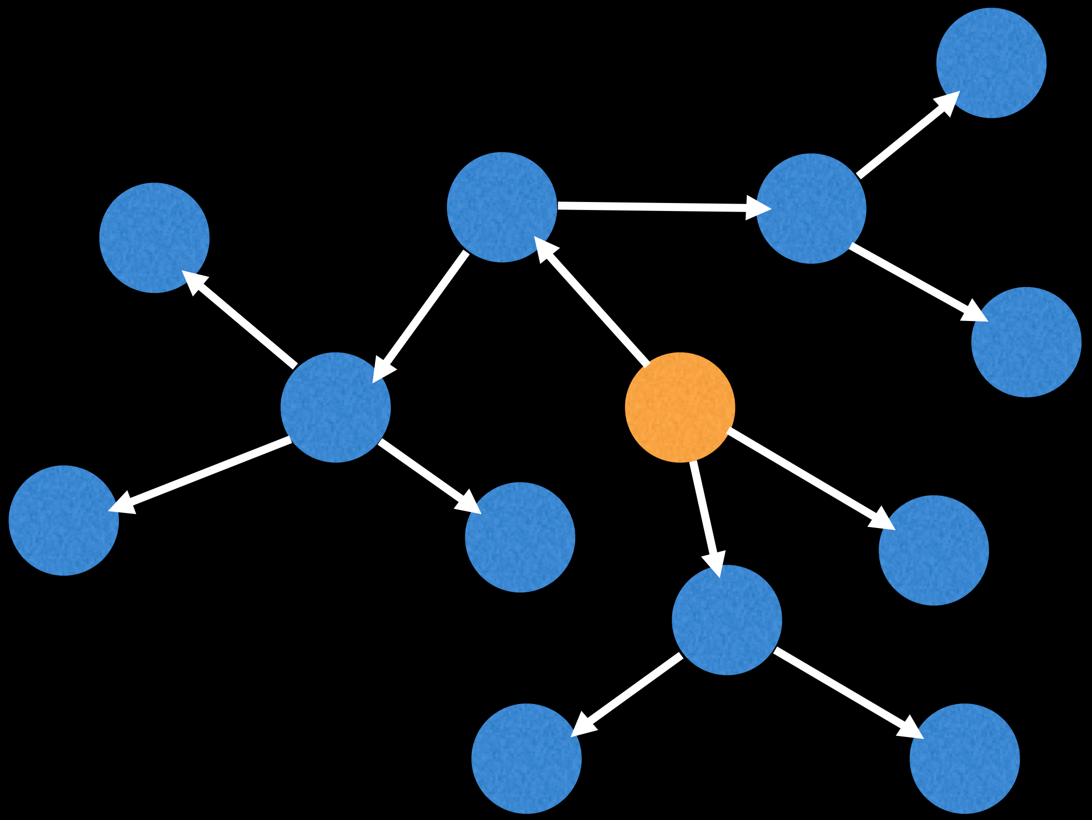
A **rooted tree** is a tree with **a designated root node** where every edge either points away from or towards the root node. When edges point away from the root the graph is called an **arborescence (out-tree)** and anti-arborescence (in-tree) otherwise.



In-tree



Out-tree

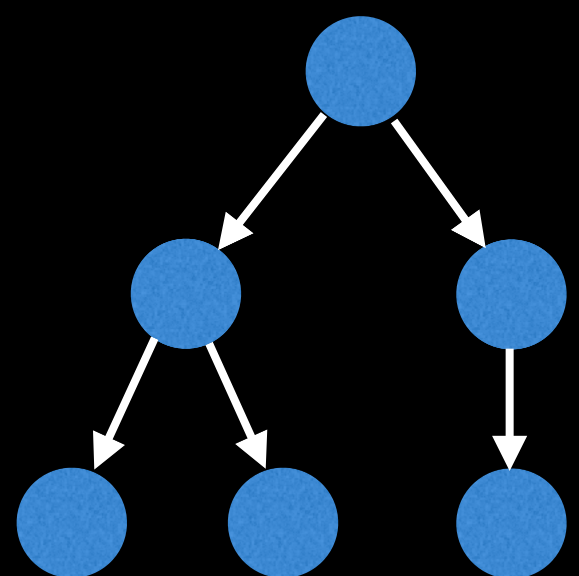


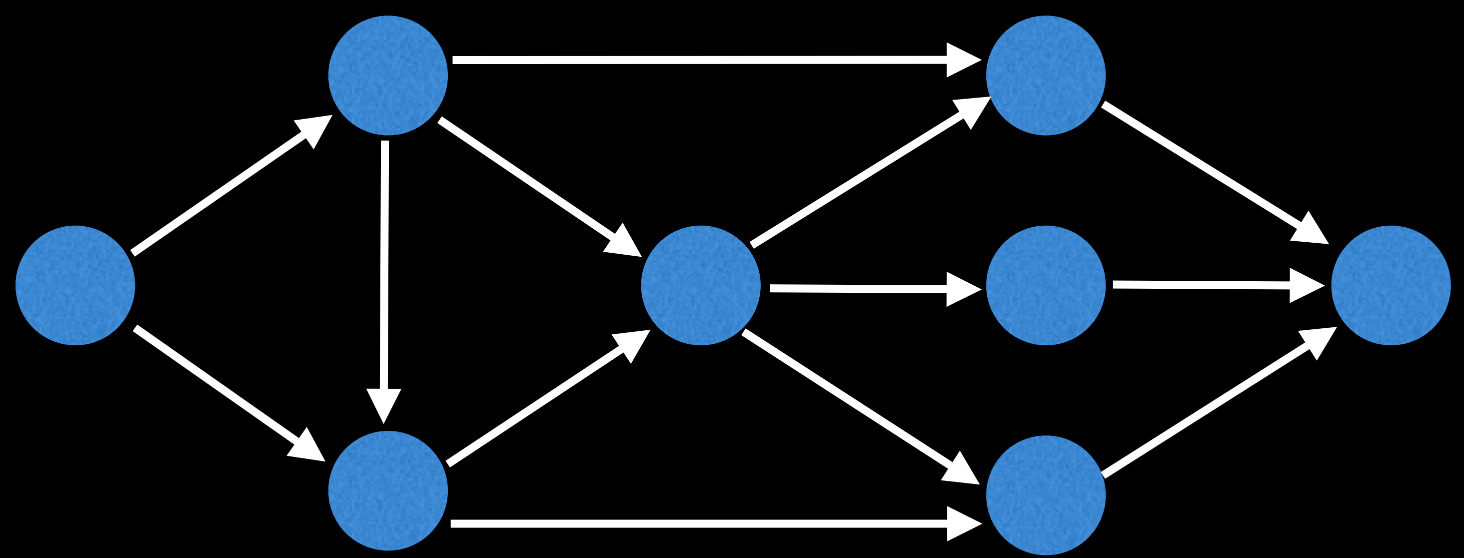
Out-tree

Directed Acyclic Graphs (DAGs)

DAGs are **directed graphs with no cycles**. These graphs play an important role in representing structures with dependencies. Several efficient algorithms exist to operates on DAGs.

Cool fact: **All out-trees are DAGs** but **not all DAGs are out-trees**.

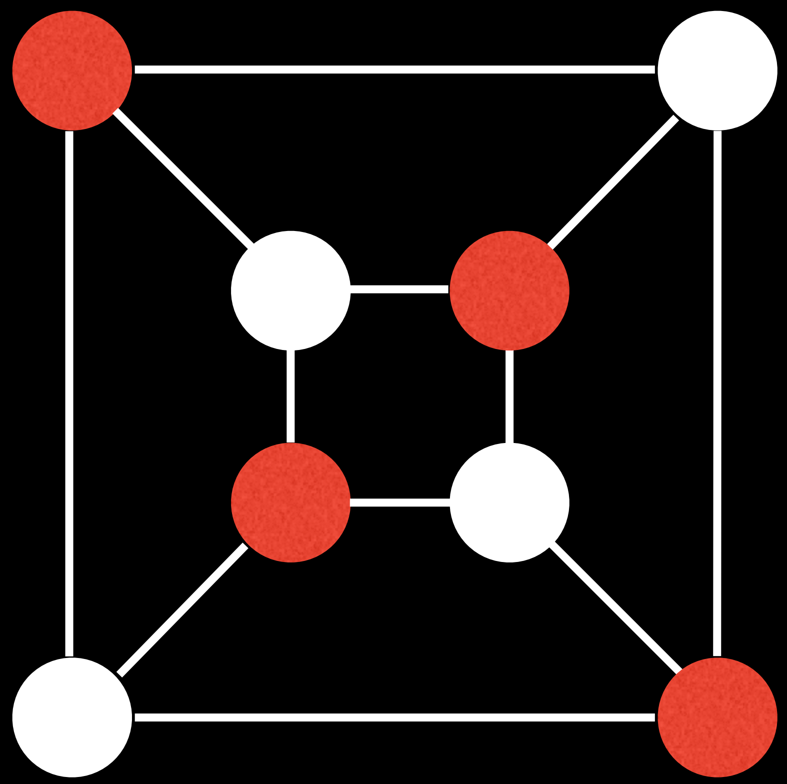


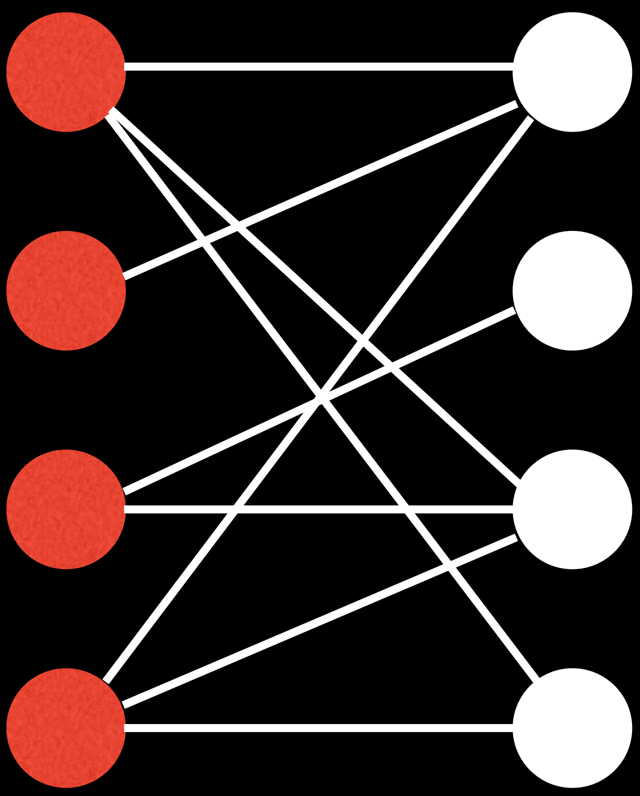


Bipartite Graph

A **bipartite graph** is one whose *vertices* can be split into two independent groups U, V such that every edge connects betweens U and V.

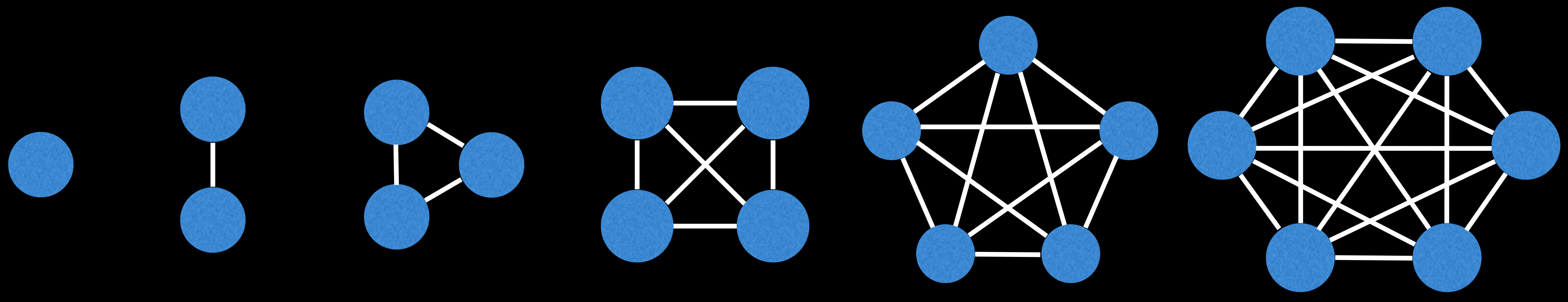
Other definitions exist such as: The graph is two colourable or there is no odd length cycle.





Complete Graphs

A **complete graph** is one where there is a unique edge between every pair of nodes. A complete graph with n vertices is denoted as the graph K_n.

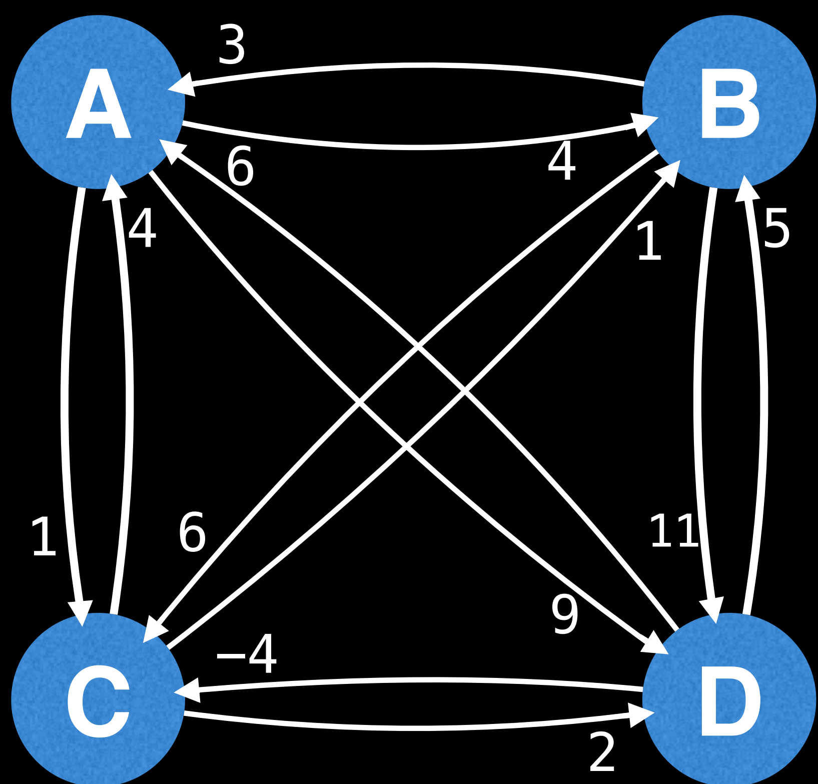


K_1 K_2 K_3 K_4 K_5 K_6

Representing Graphs

Adjacency Matrix

A **adjacency matrix** *m* is a very simple way to represent a graph. The idea is that the cell m[i][j] represents the edge weight of going from node i to node j.



$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 4 & 1 & 9 \\ 3 & 0 & 6 & 11 \\ 4 & 1 & 0 & 2 \\ 6 & 5 & -4 & 0 \end{bmatrix}
\end{array}
$$

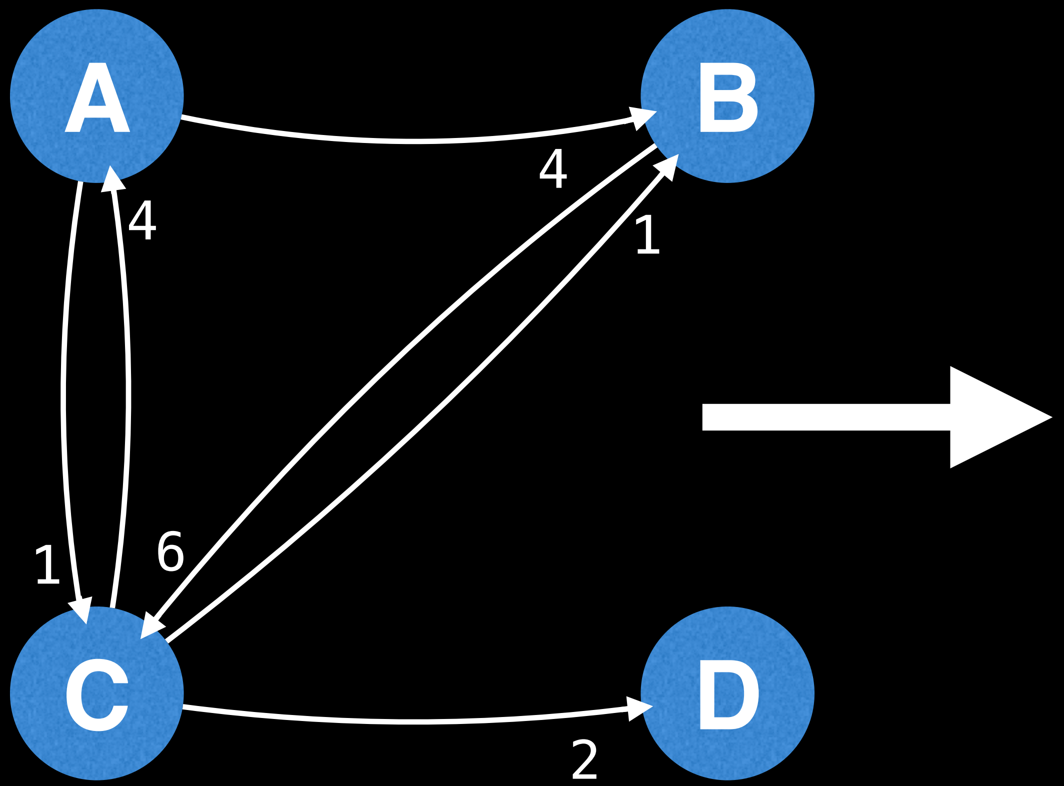
NOTE: It is often assumed that the edge of going from a node to itself has a cost of zero.

Adjacency Matrix

Pros	Cons
Space efficient for representing dense graphs	Requires **$O(V^2)$** space
Edge weight lookup is **$O(1)$**	Iterating over all edges takes **$O(V^2)$** time
Simplest graph representation	

Adjacency List

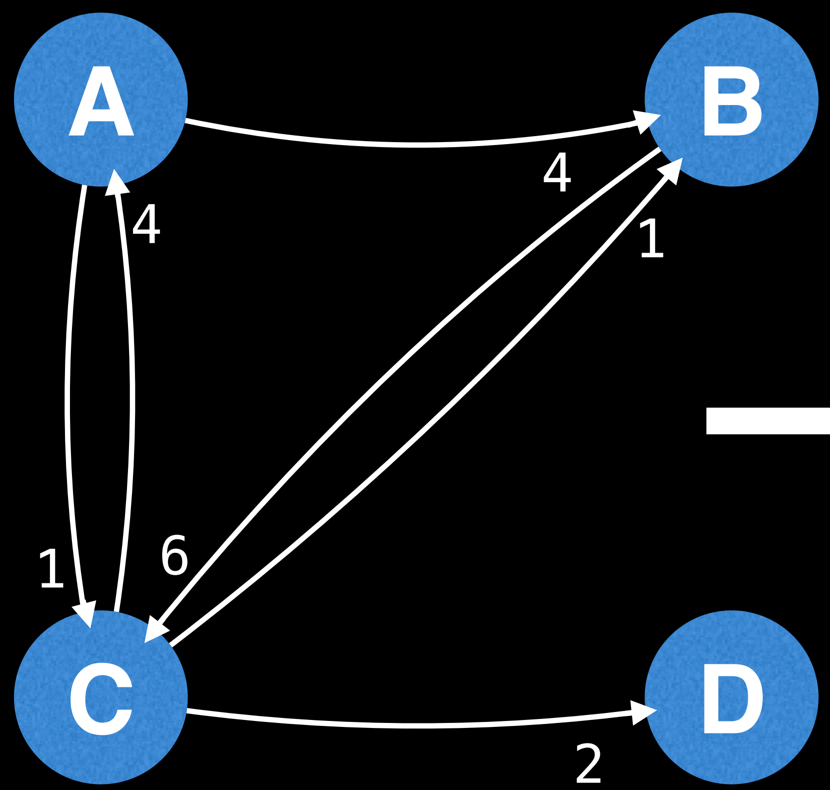
An **adjacency list** is a way to represent a graph as a map from nodes to lists of edges.




```
A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []
```

Adjacency List

An **adjacency list** is a way to represent a graph as a map from nodes to lists of edges.




```
A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []
```

Node C can reach

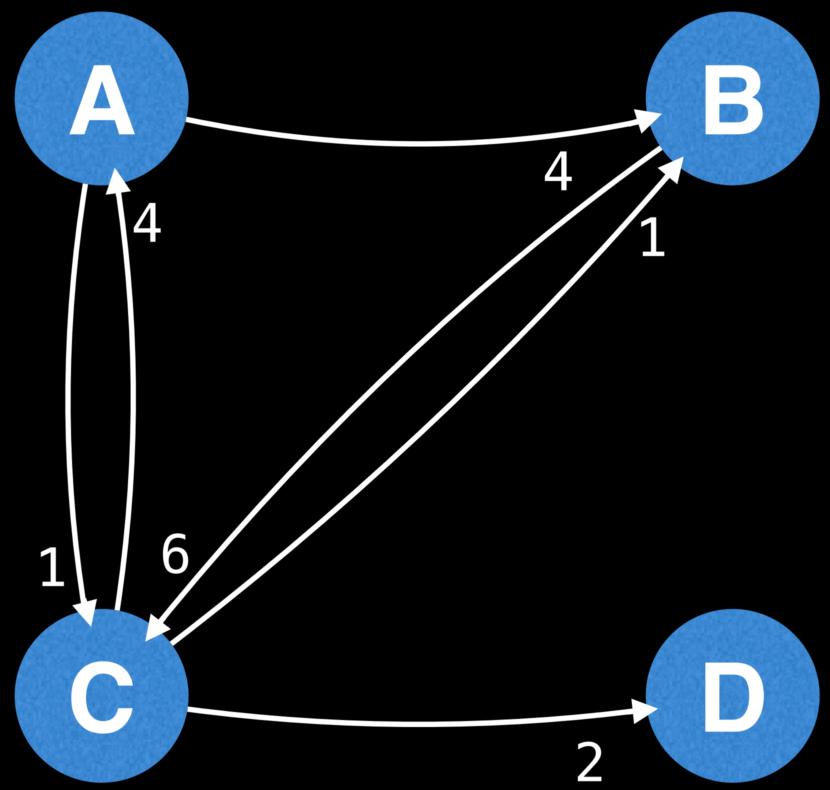
- Node A with cost 4
- Node B with cost 1
- Node D with cost 2

Adjacency List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
	Slightly more complex graph representation

Edge List

An **edge list** is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u,v,w) means: “the cost from node u to node v is w”



[(C,A,4), (A,C,1),
(B,C,6), (A,B,4),
(C,B,1), (C,D,2)]

This representation is seldomly used because of its lack of structure. However, it is conceptually simple and practical in a handful of algorithms.

Edge List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
Very simple structure	

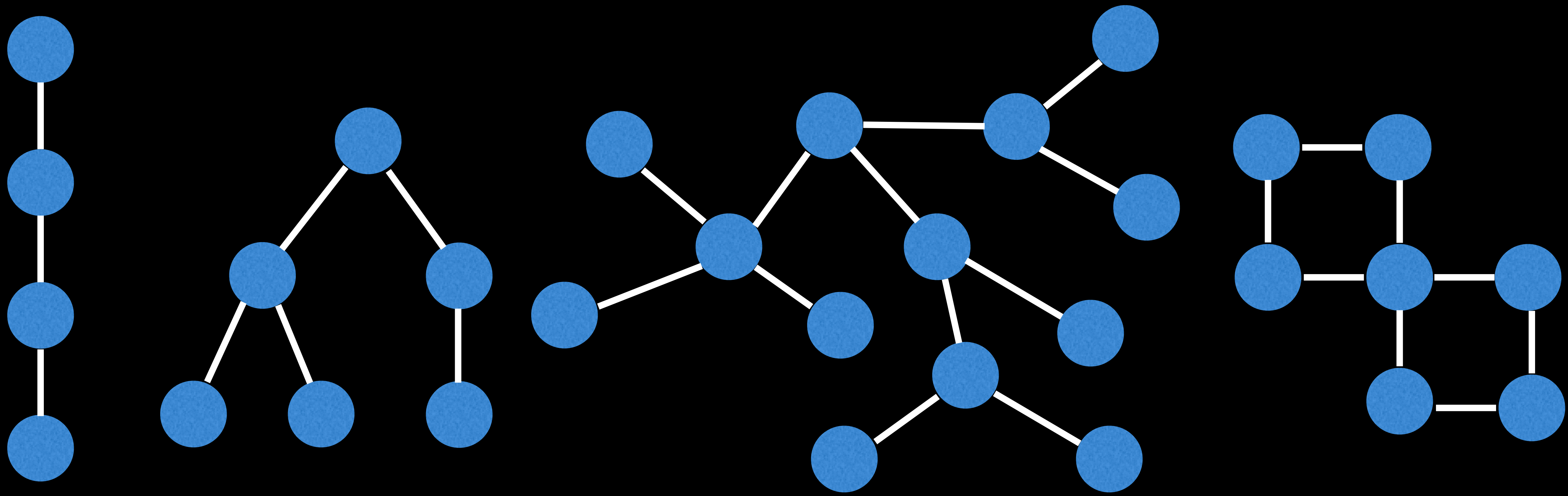
Storage and representation of trees

Definitions and storage representation

William Fiset

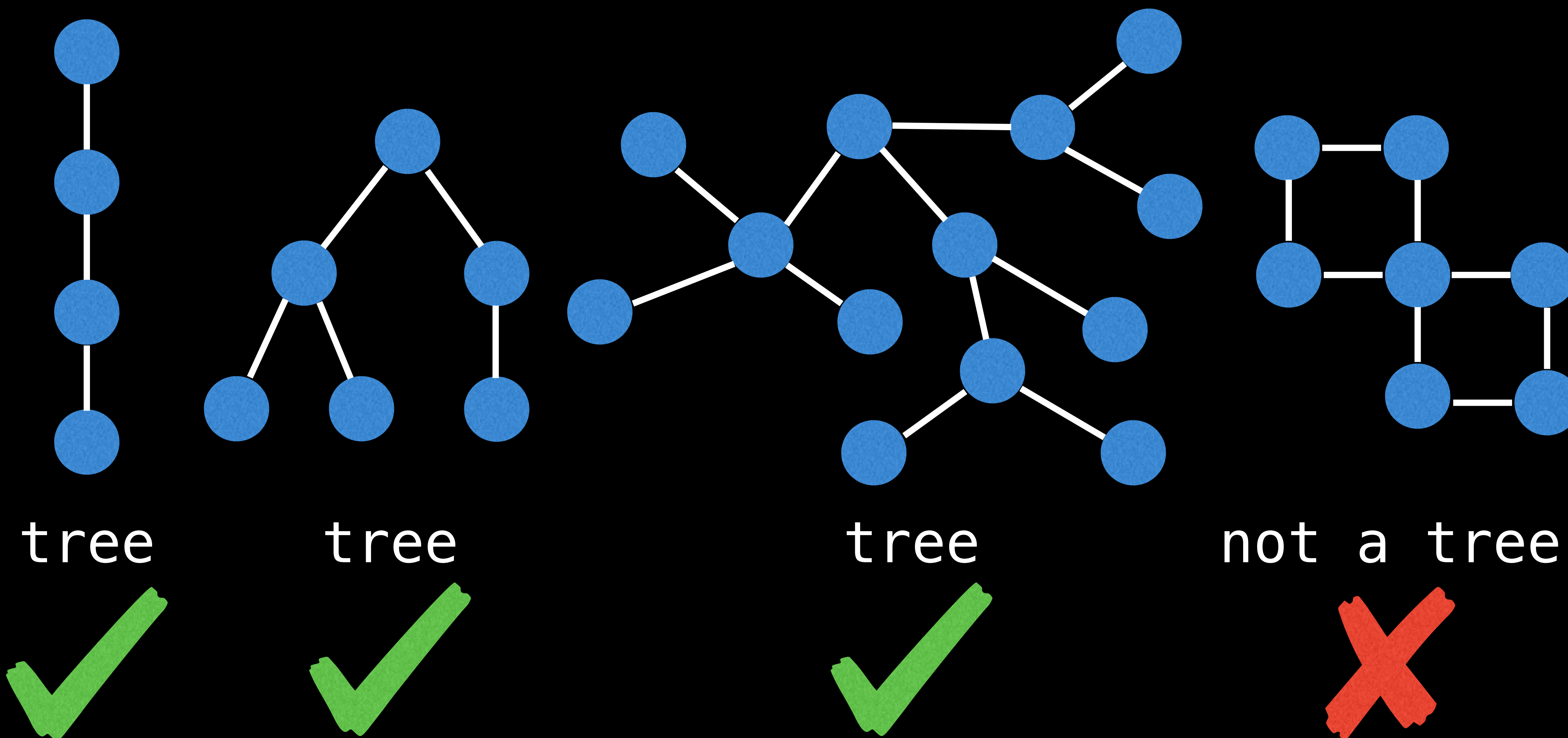
Trees!

What ***is*** a tree?



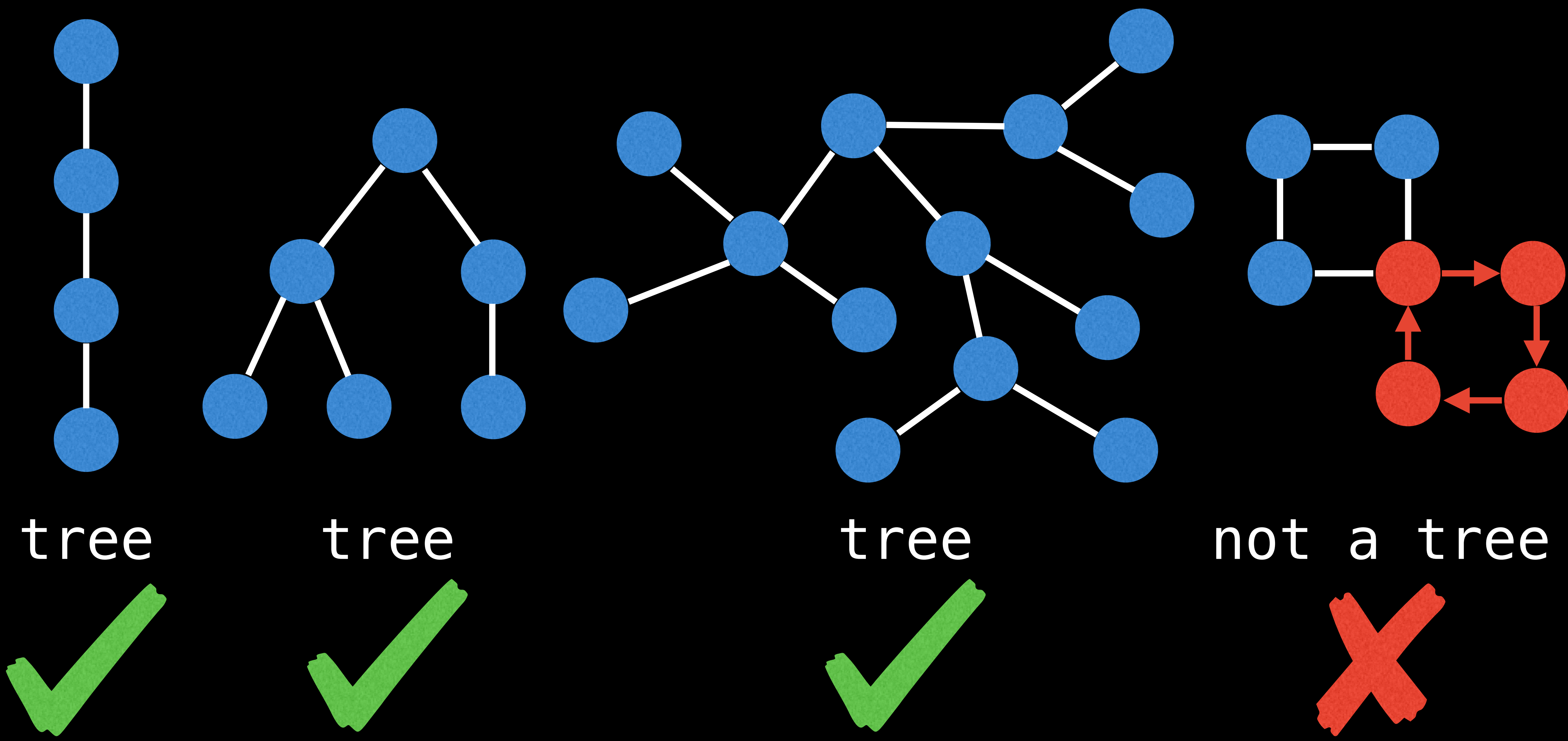
Trees!

What ***is*** a tree?



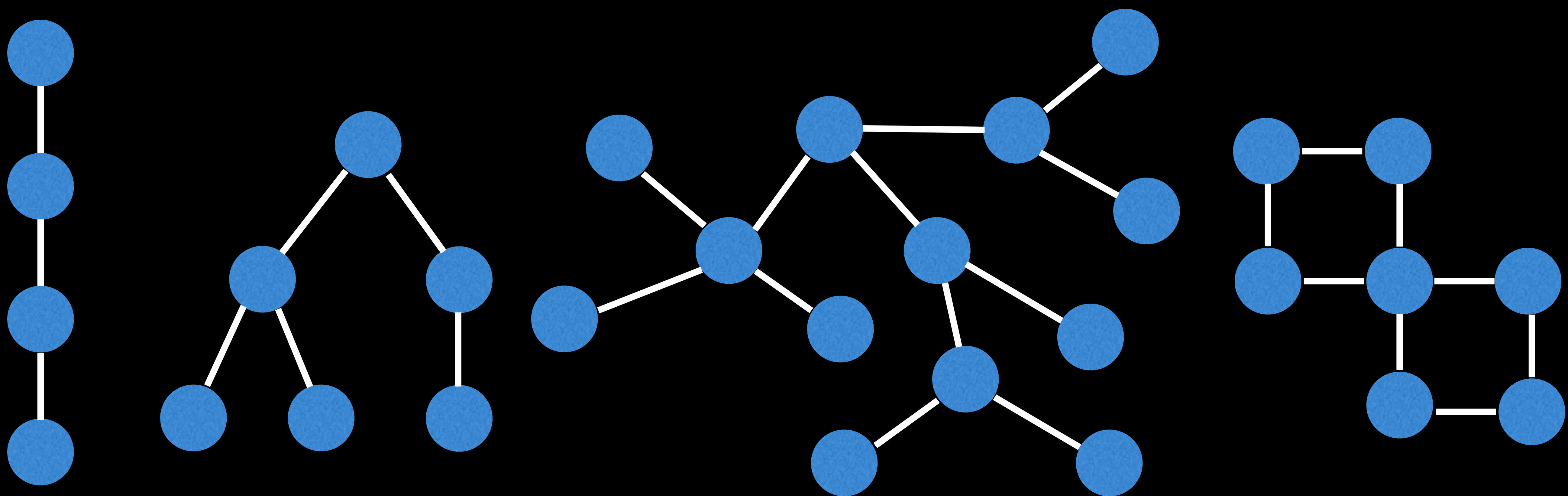
Trees!

A **tree** is a connected, undirected graph with **no cycles**.



Trees!

Equivalently, a **tree** it is a connected undirected graph with N nodes and N–1 edges.



4 nodes
3 edges

6 nodes
5 edges

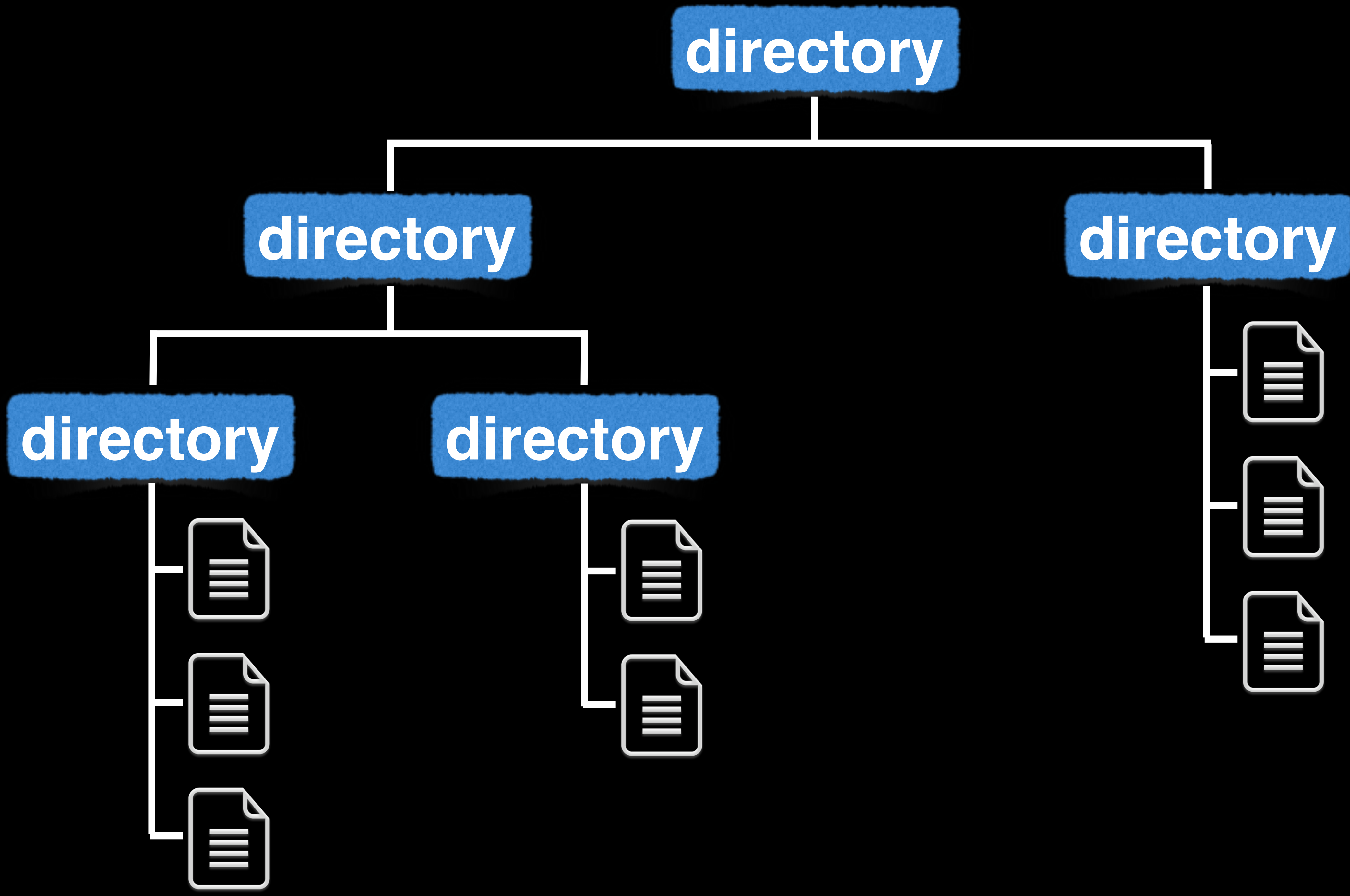
13 nodes
12 edges

7 nodes
8 edges

Trees out in the wild

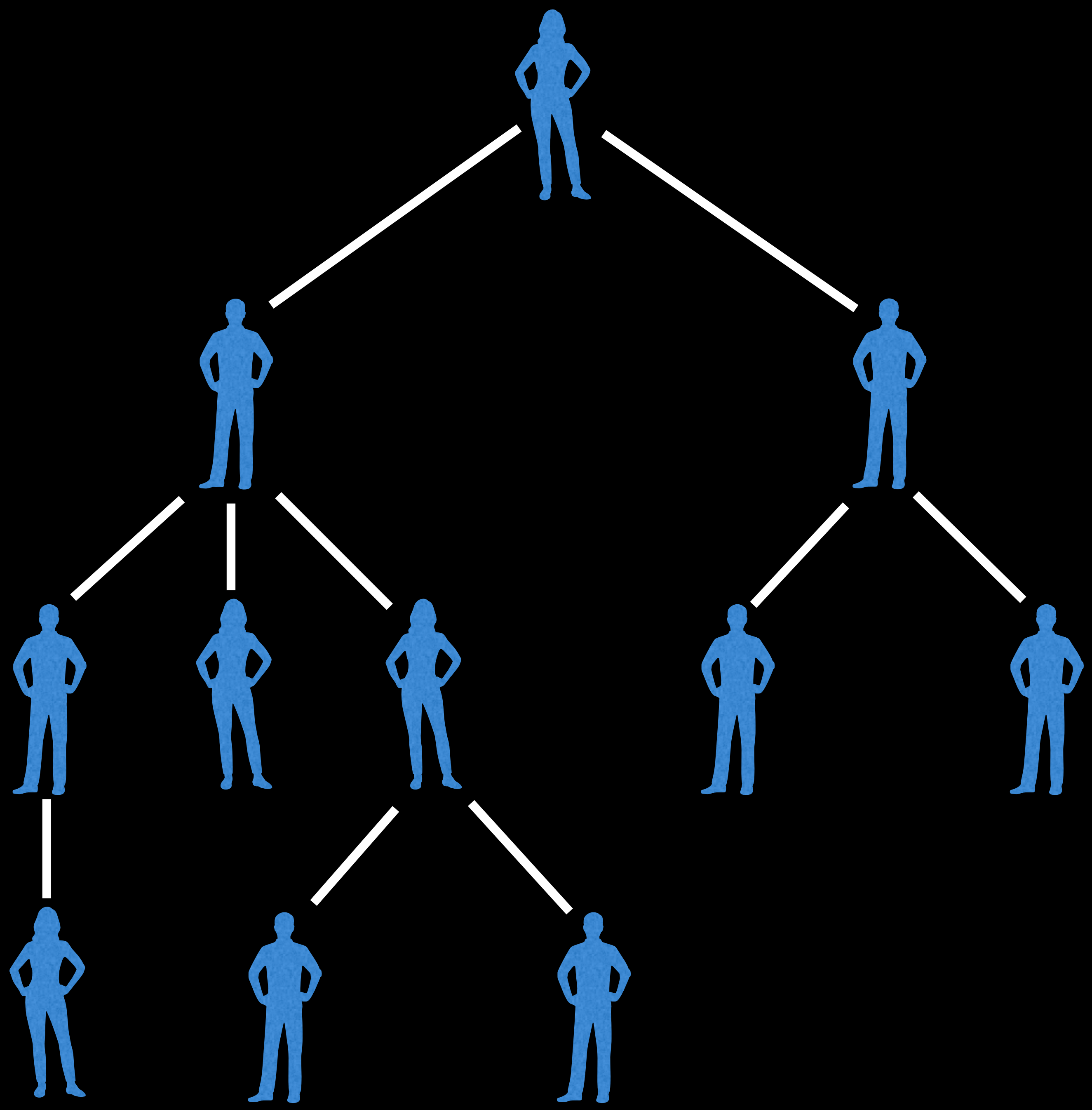
Trees out in the wild

Filesystem structures are inherently trees



Trees out in the wild

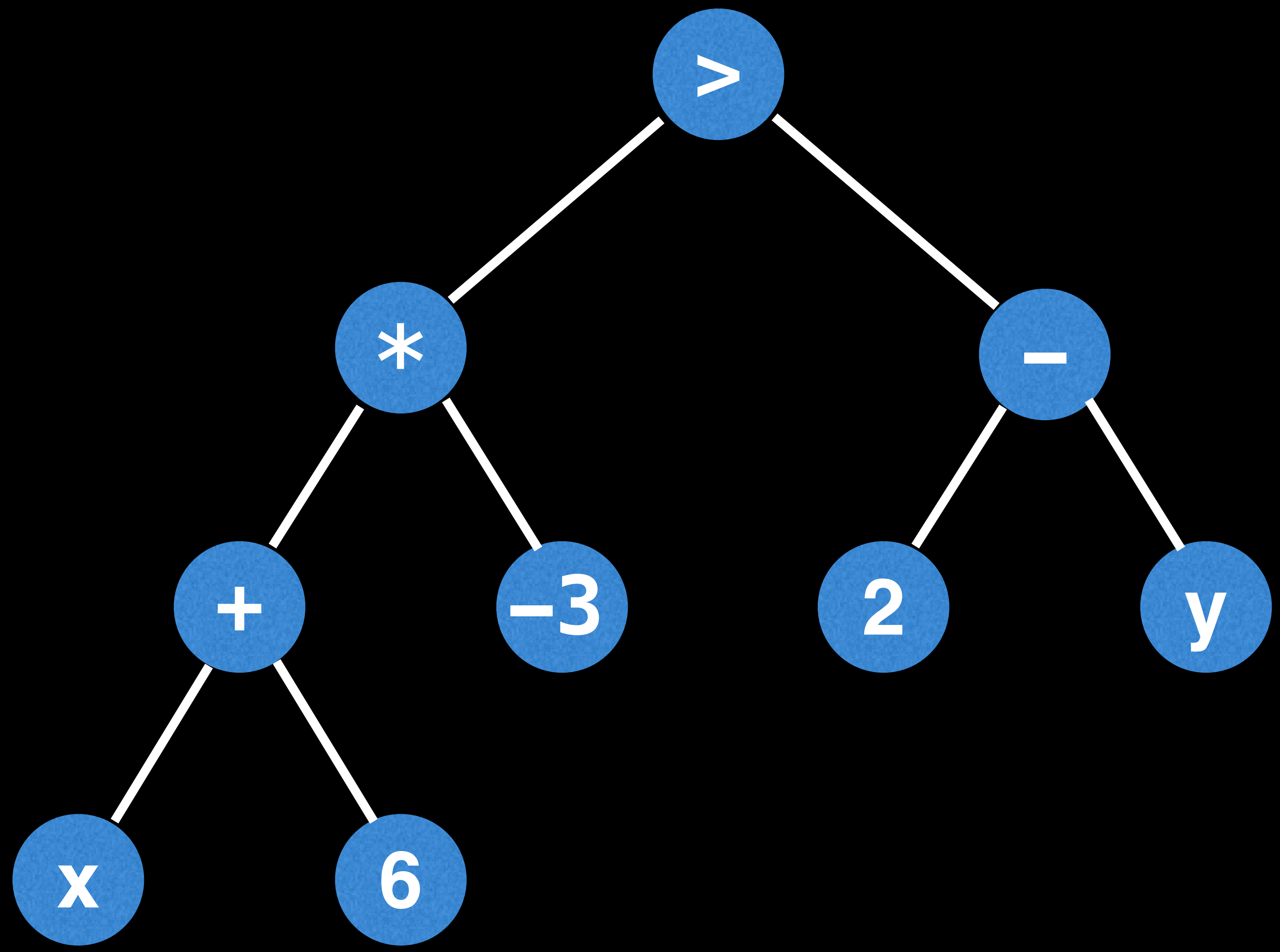
Social hierarchies



Trees out in the wild

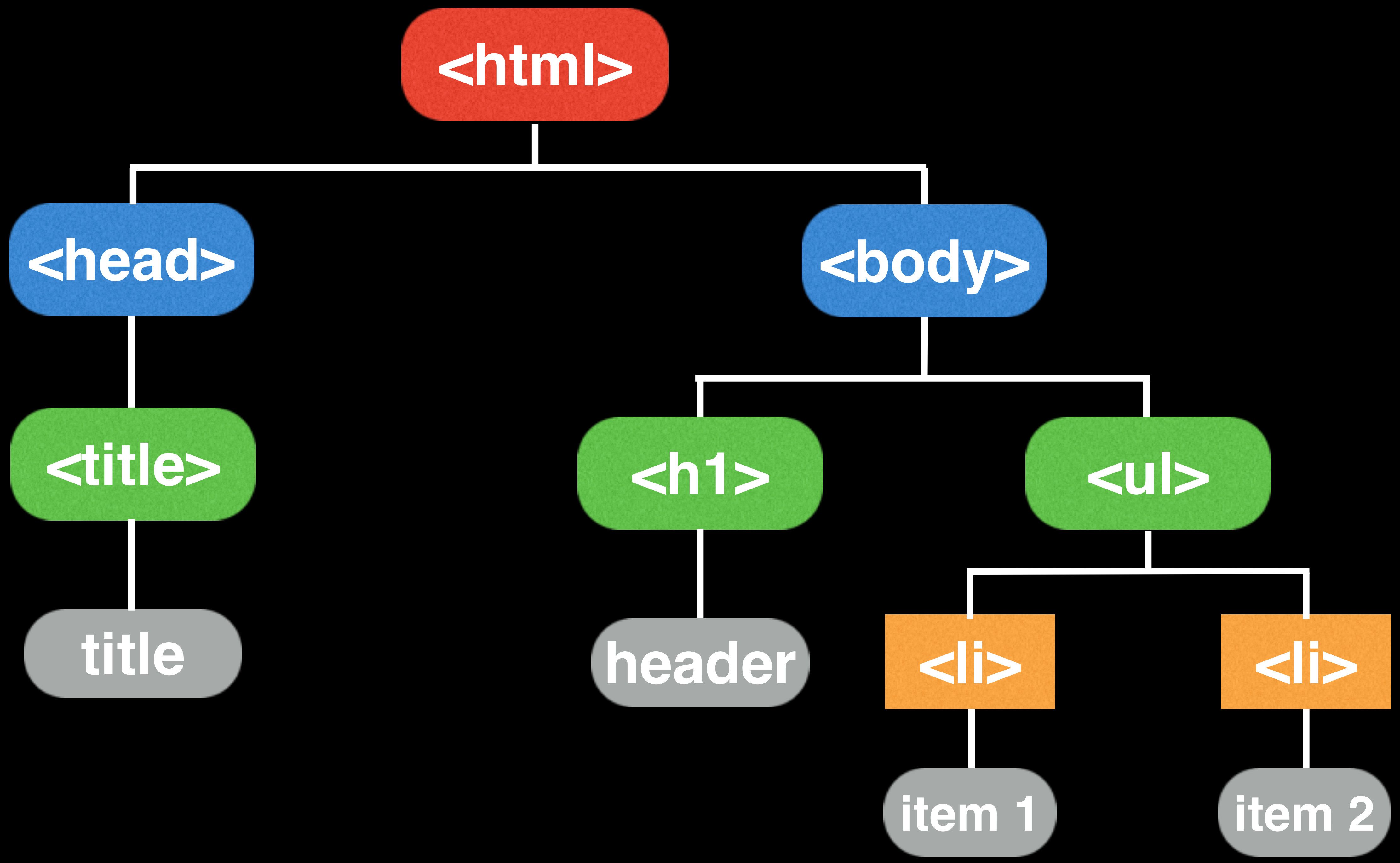
Abstract syntax trees to decompose source code and mathematical expressions for easy evaluation.

$$((x + 6) * -3) > (2 - y)$$



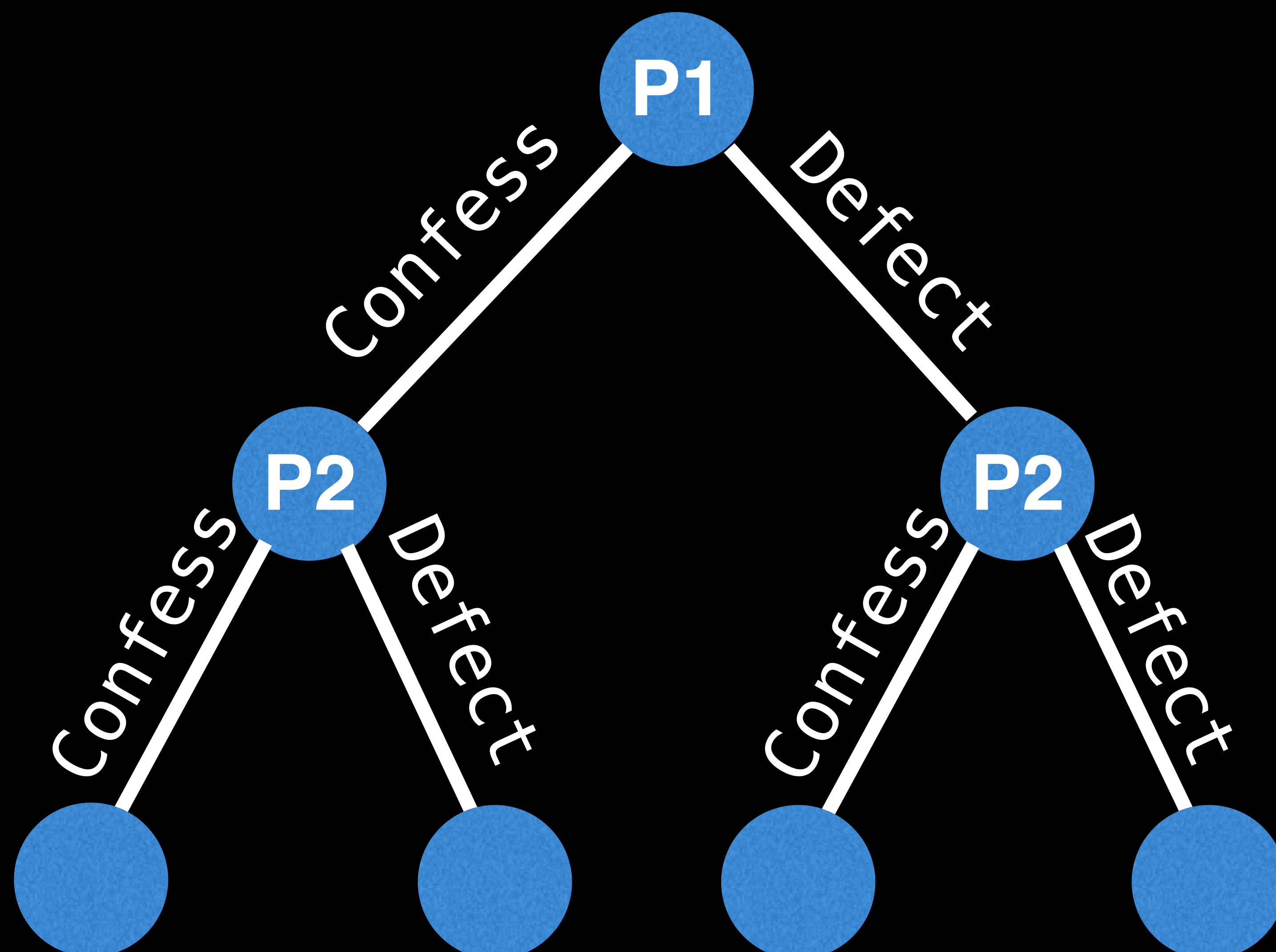
Trees out in the wild

Every webpage is a tree as an HTML DOM structure



Trees out in the wild

The decision outcomes in game theory are often modeled as trees for ease of representation.



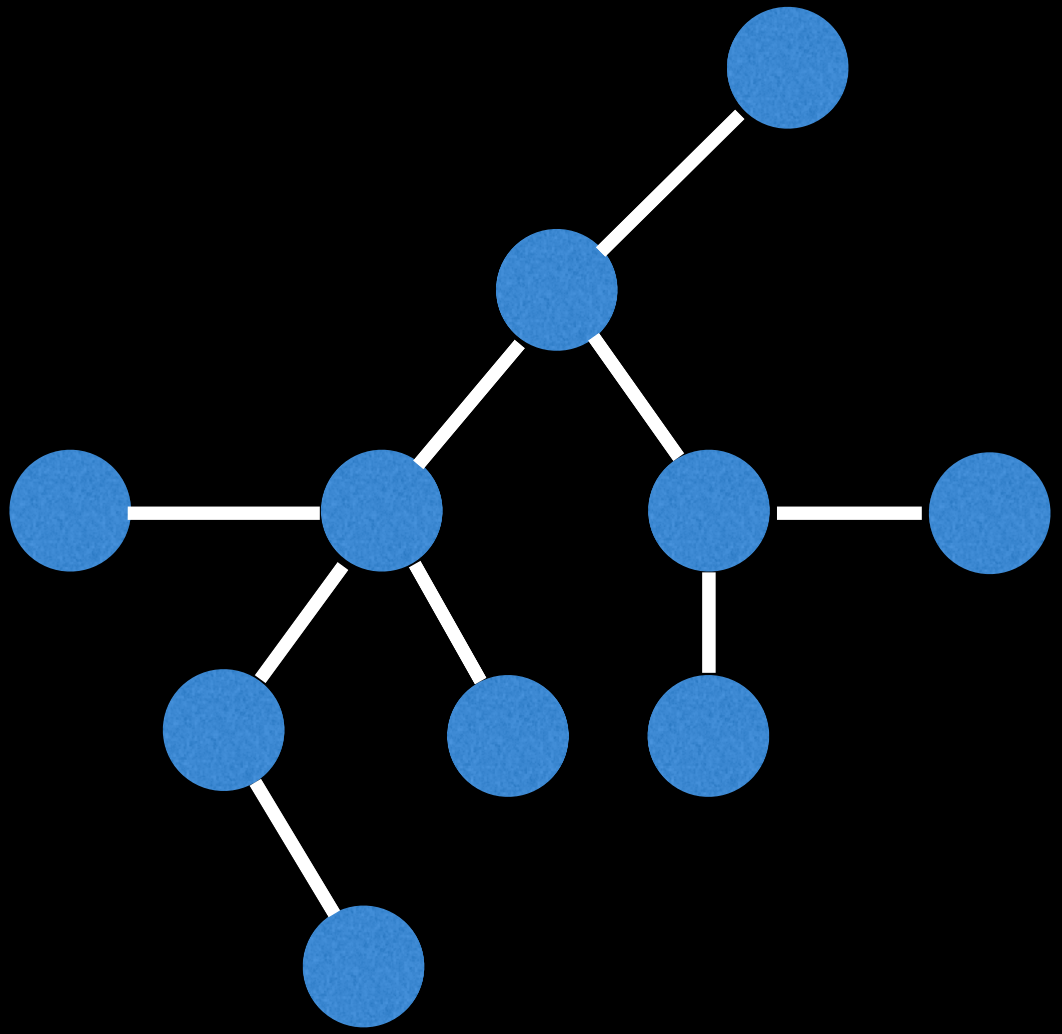
Tree of the prisoner's dilemma

Trees out in the wild

There are many many more applications…

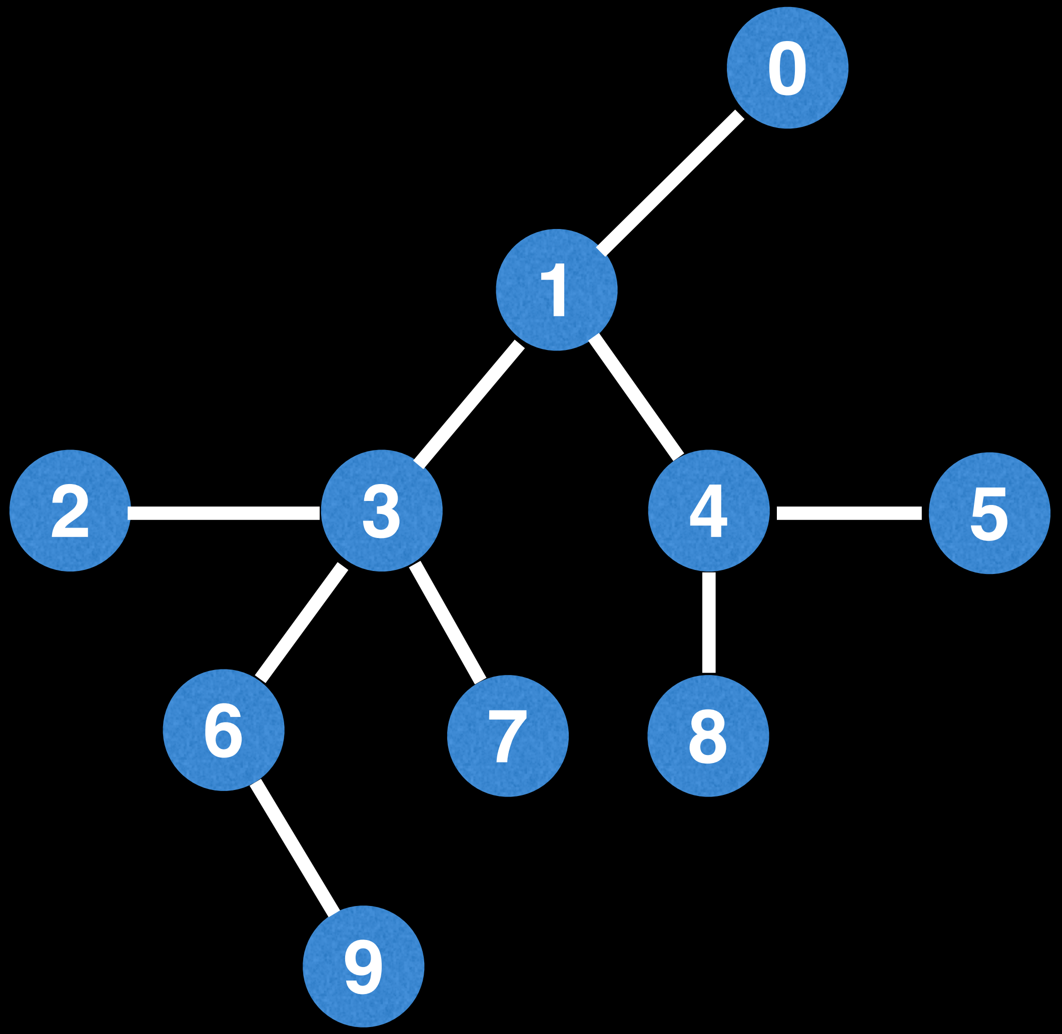
- Family trees
- File parsing/HTML/JSON/Syntax trees
- Many data structures use/are trees:
  - AVL trees, B-tree, red-black trees, segment trees, fenwick trees, treaps, suffix trees, tree maps/sets, etc…
- Game theory decision trees
- Organizational structures
- Probabilty trees
- Taxonomies
- etc…

Storing undirected trees

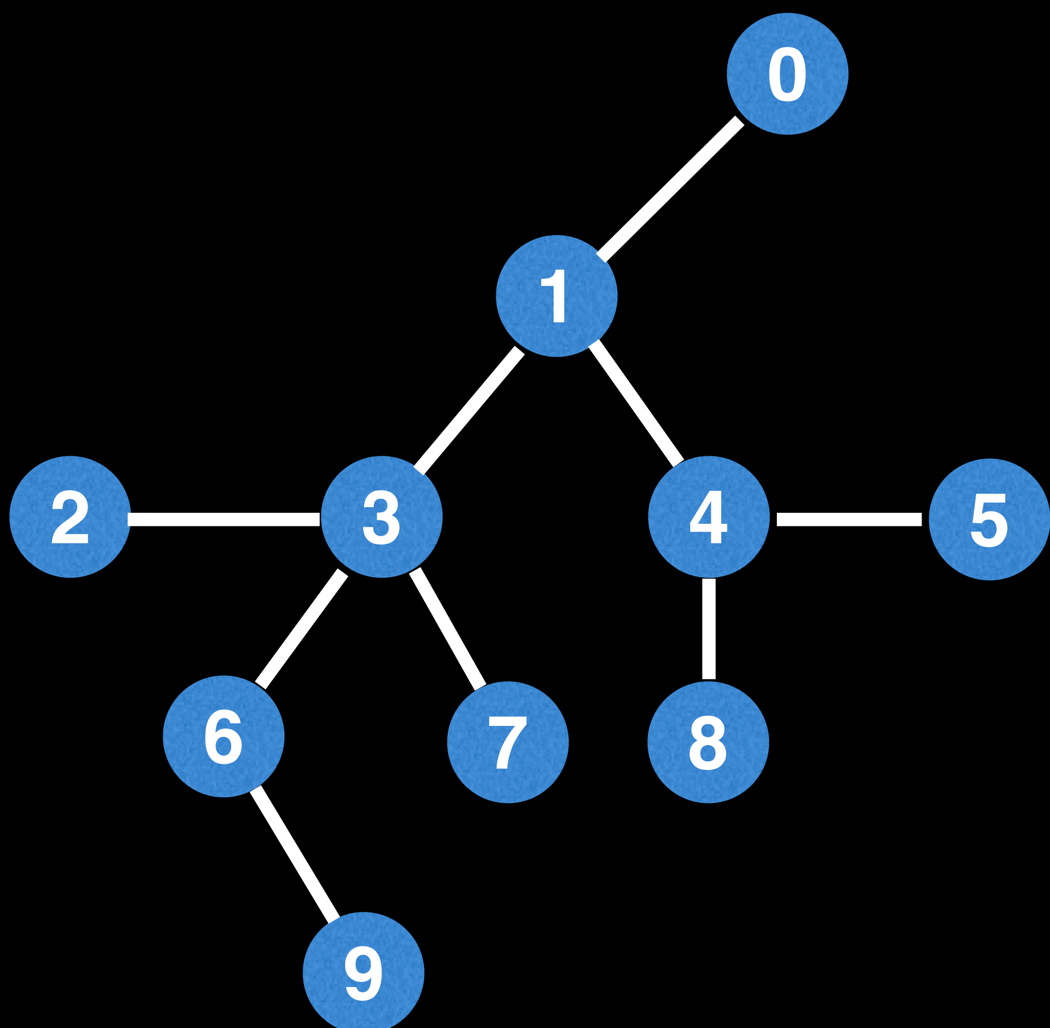


Storing undirected trees

Start by labelling the tree nodes from [0, n)



Storing undirected trees

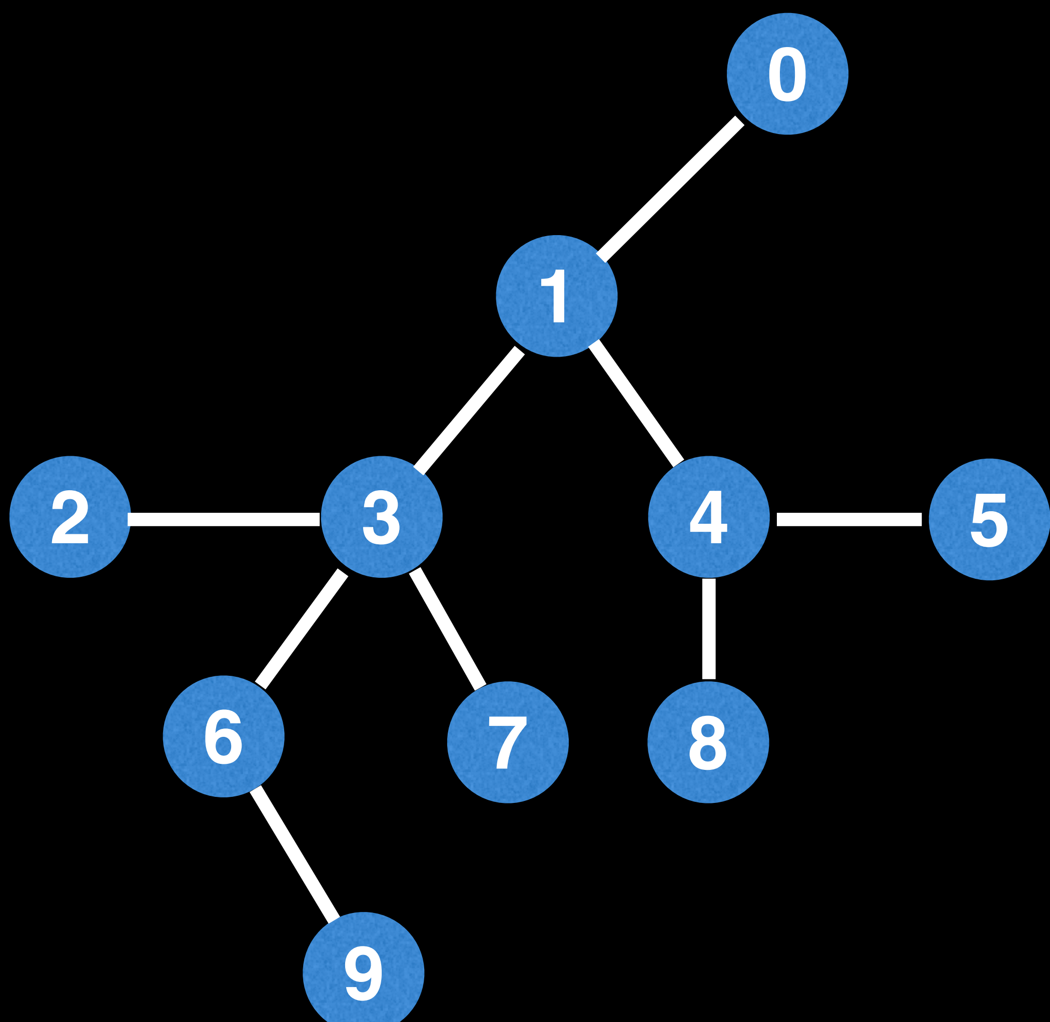


edge list storage representation:

```
[(0, 1),
 (1, 4),
 (4, 5),
 (4, 8),
 (1, 3),
 (3, 7),
 (3, 6),
 (2, 3),
 (6, 9)]
```

pro: simple and easy to iterate over.

Storing undirected trees

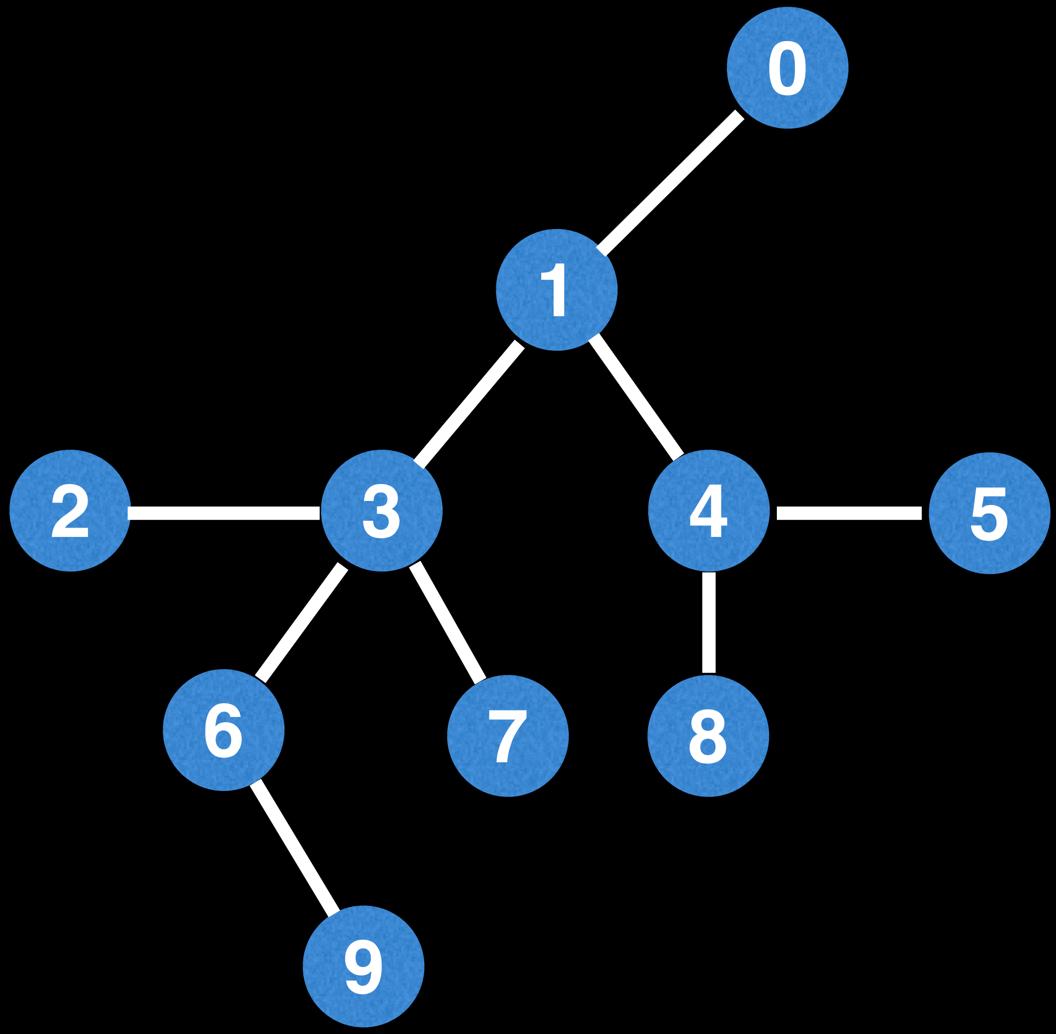


edge list storage representation:

```
[(0, 1),
 (1, 4),
 (4, 5),
 (4, 8),
 (1, 3),
 (3, 7),
 (3, 6),
 (2, 3),
 (6, 9)]
```

con: storing a tree as a list lacks the structure to efficiently query all the neighbors of a node.

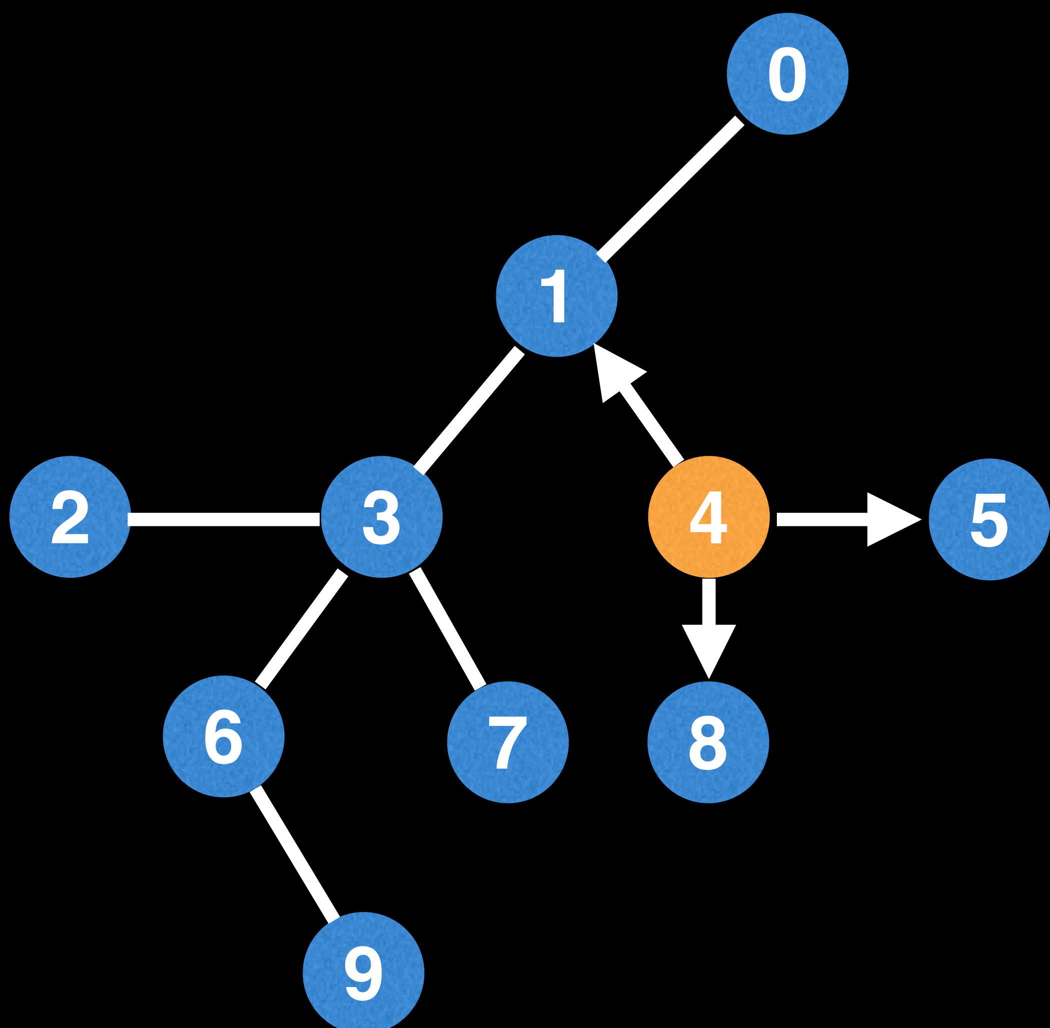
Storing undirected trees



adjacency list representation

```
0 -> [1]
1 -> [0,3,4]
2 -> [3]
3 -> [1,2,6,7]
4 -> [1,5,8]
5 -> [4]
6 -> [3,9]
7 -> [3]
8 -> [4]
9 -> [6]
```

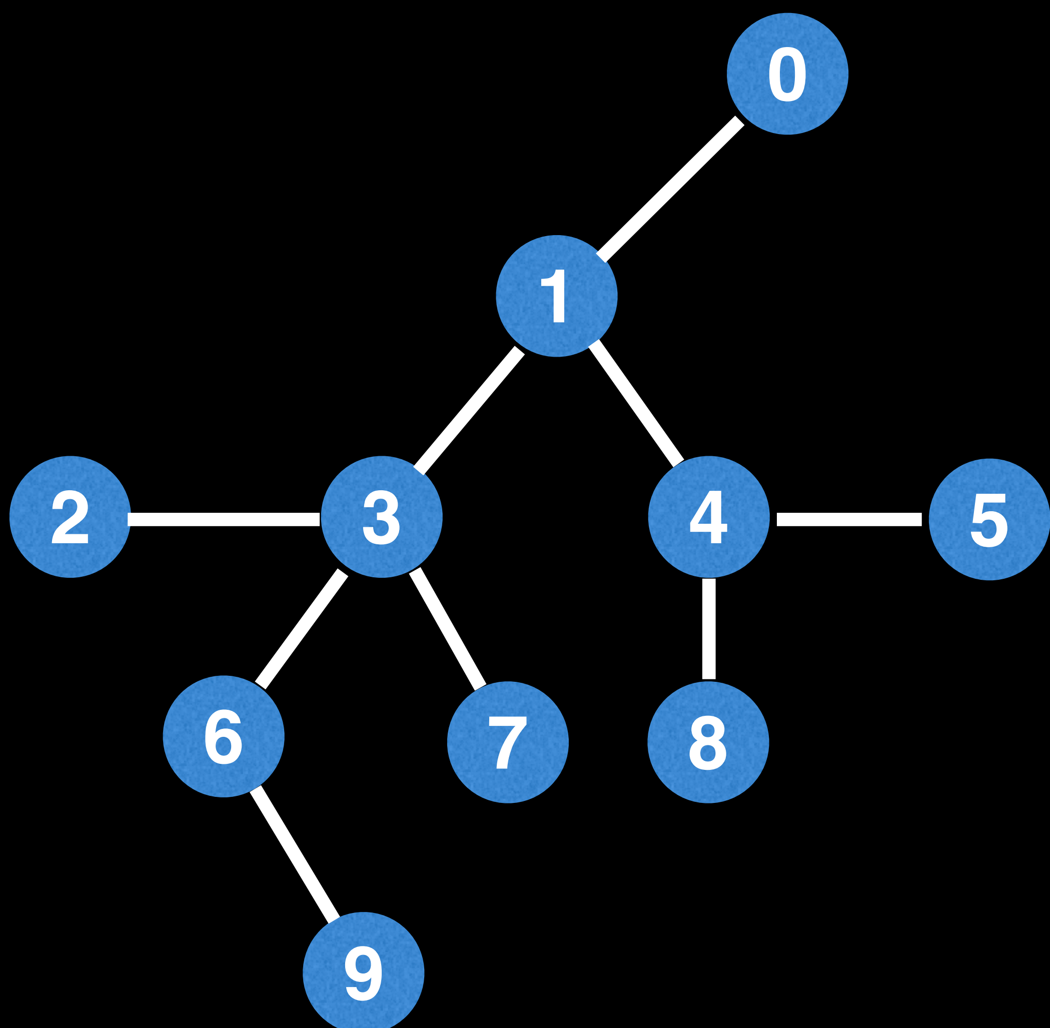
Storing undirected trees



adjacency list representation

```
0 -> [1]
1 -> [0,3,4]
2 -> [3]
3 -> [1,2,6,7]
4 -> [1,5,8]
5 -> [4]
6 -> [3,9]
7 -> [3]
8 -> [4]
9 -> [6]
```

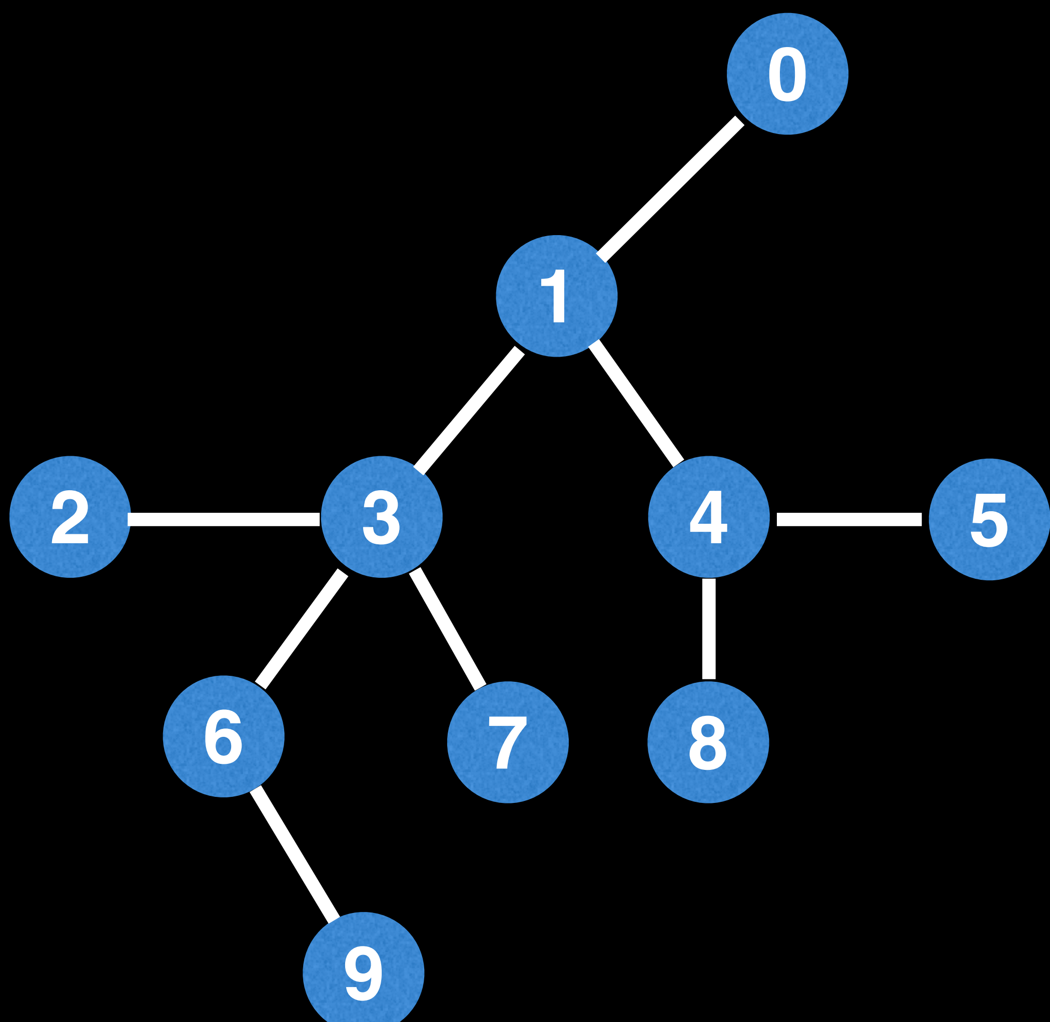
Storing undirected trees



adjacency matrix representation

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	1	1	0	0	0	1	1	0	0
4	0	1	0	0	0	1	0	0	1	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	1
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0	0

Storing undirected trees



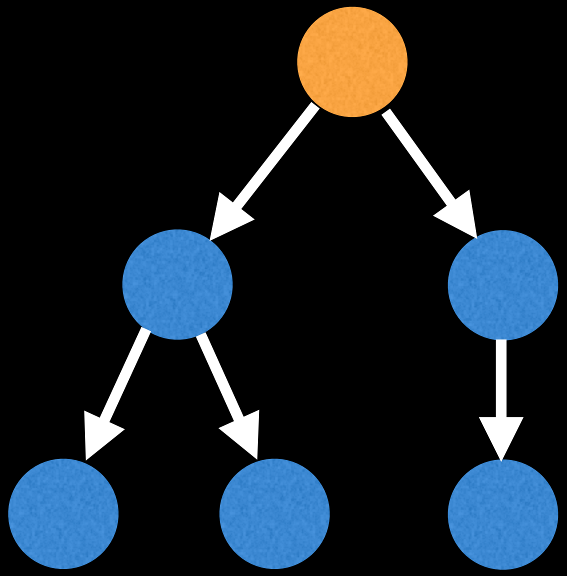
adjacency matrix representation

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	1	1	0	0	0	1	1	0	0
4	0	1	0	0	0	1	0	0	1	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	1
7	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0	0

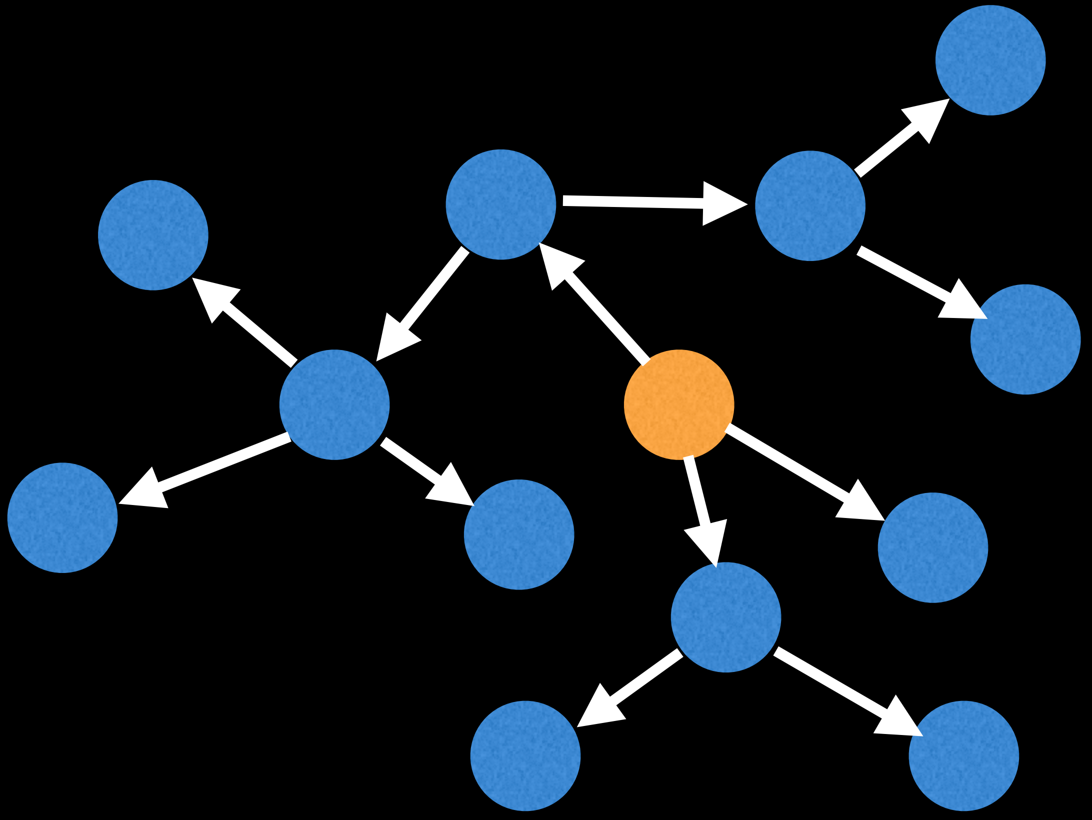
In practice, avoid storing a tree as an adjacency matrix! It's a huge **waste of space** to use n^2 memory and only use $2(n-1)$ of the matrix cells.

Rooted Trees!

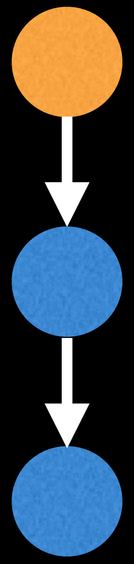
One of the more interesting types of trees is a **rooted tree** which is a tree with a designated **root node**.



Rooted tree



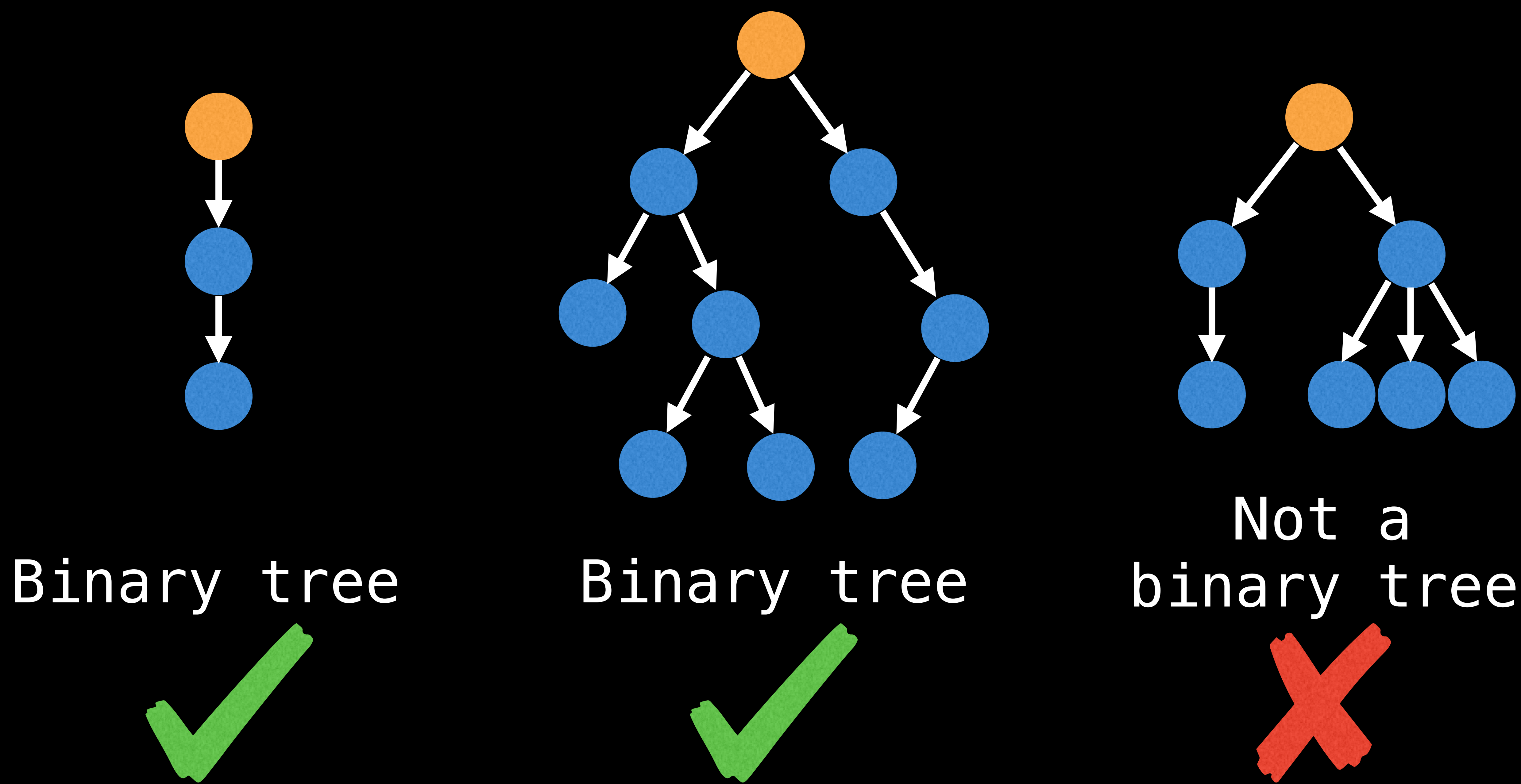
Rooted tree



Rooted tree

Binary Tree (BT)

Related to rooted trees are **binary trees** which are trees for which every node has **at most two child nodes**.



Binary Search Trees (BST)

Related to binary trees are **binary search trees** which are trees which satisfy the BST invariant which states that for every node x:

x.left.value ≤ x.value ≤ x.right.value