CS 106X, Lecture 21
Tries; Graphs

reading:

Programming Abstractions in C++, Chapter 18

Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

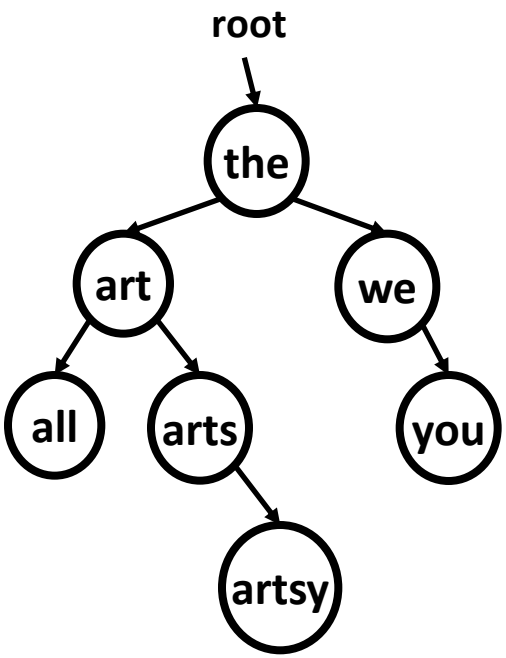
Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

The Lexicon

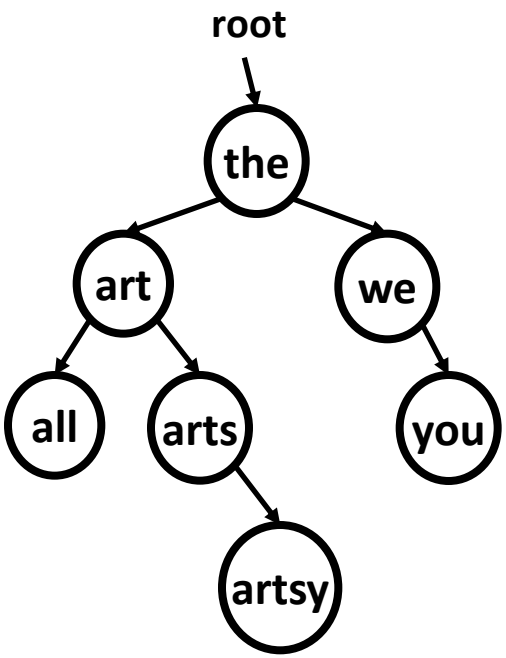
- Lexicons are good for storing words
  - `contains`
  - `containsPrefix`
  - `add`

The Lexicon



The Lexicon

```
contains?
containsPrefix?
add?
```



The Lexicon

S	T	A	R	T	L	I	N	G

S	T	A	R	T

The Lexicon

- We want to model a set of words as a tree of some kind
- The tree should be sorted in some way for efficient lookup
- The tree should take advantage of words containing each other to save space and time

单词里面包含单词

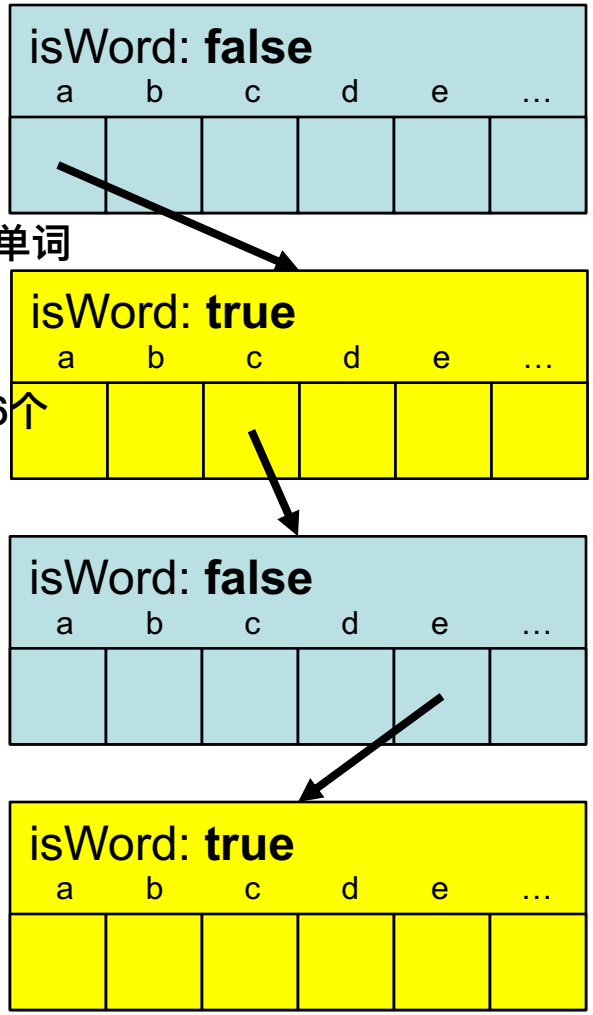
Tries

trie ("try"): A tree structure optimized for "prefix" searches

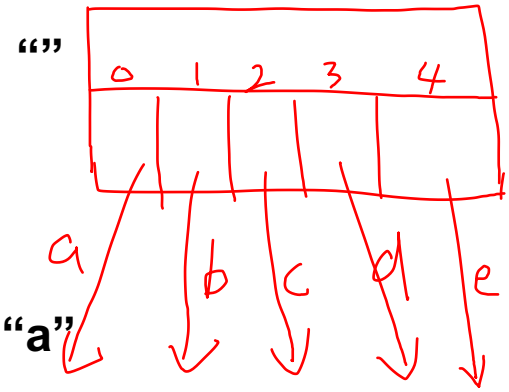
```
struct TrieNode {
    bool isWord;
    TrieNode* children[26];  // depends on the alphabet
};
```

Tries

isWord: 走到当前位置，是否构成了单词
children[26]的index代表字母
0->‘a’… index= letter – ‘a’
有点类似backtracking，下一步的26个选择



更好的理解作图：



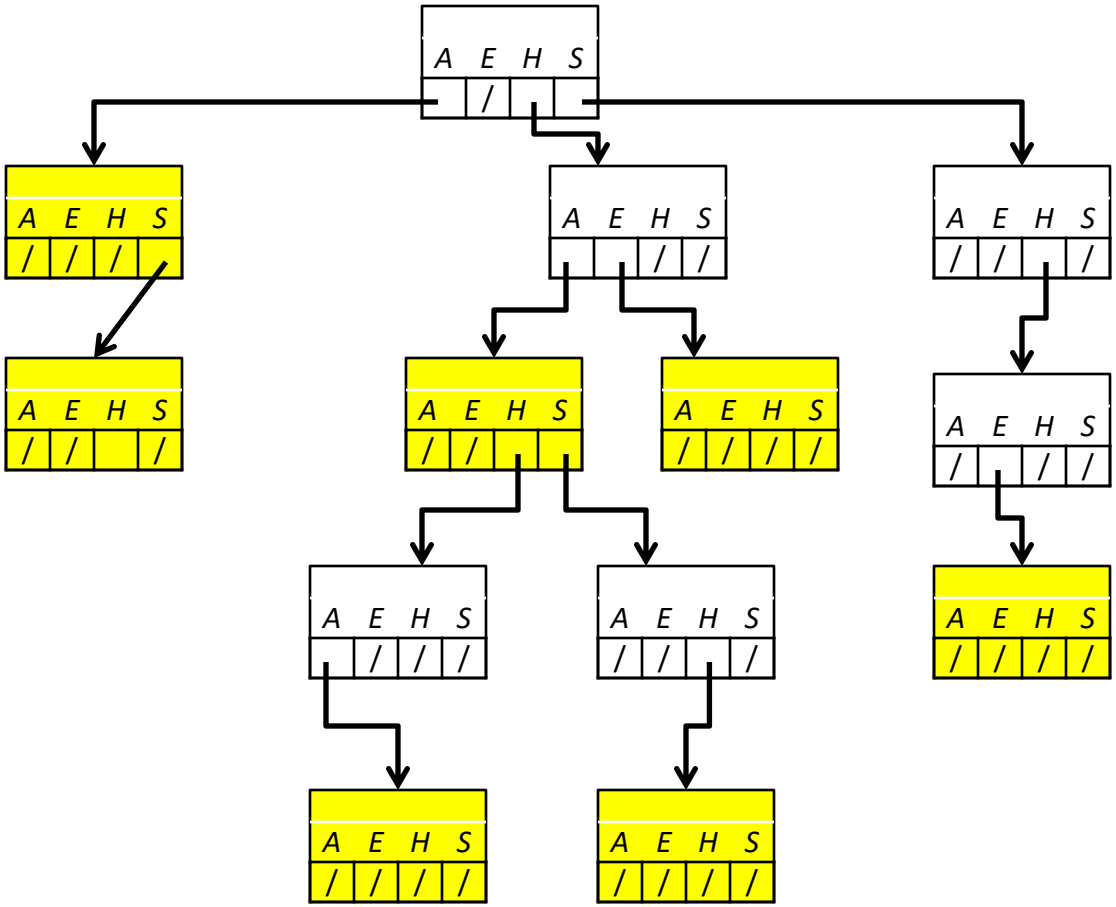
“a”

“ac”

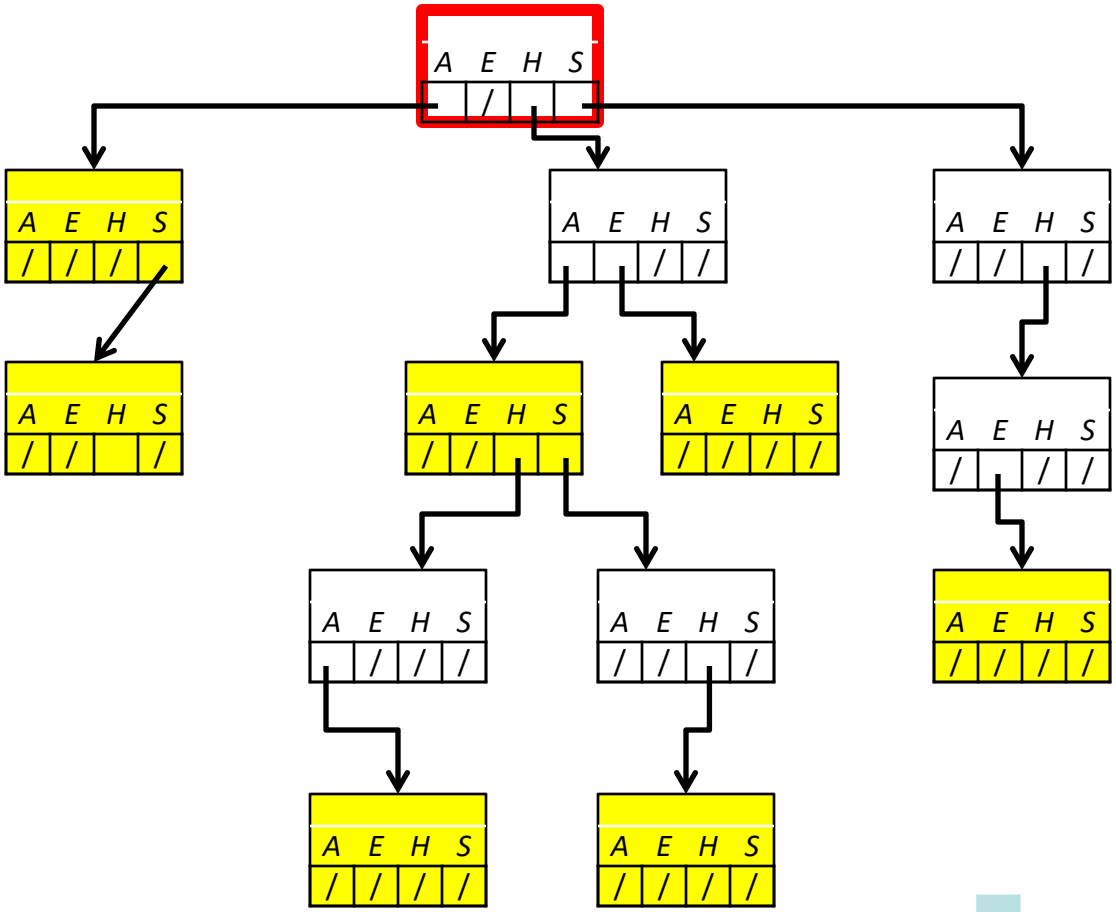
“ace”

Reading Words

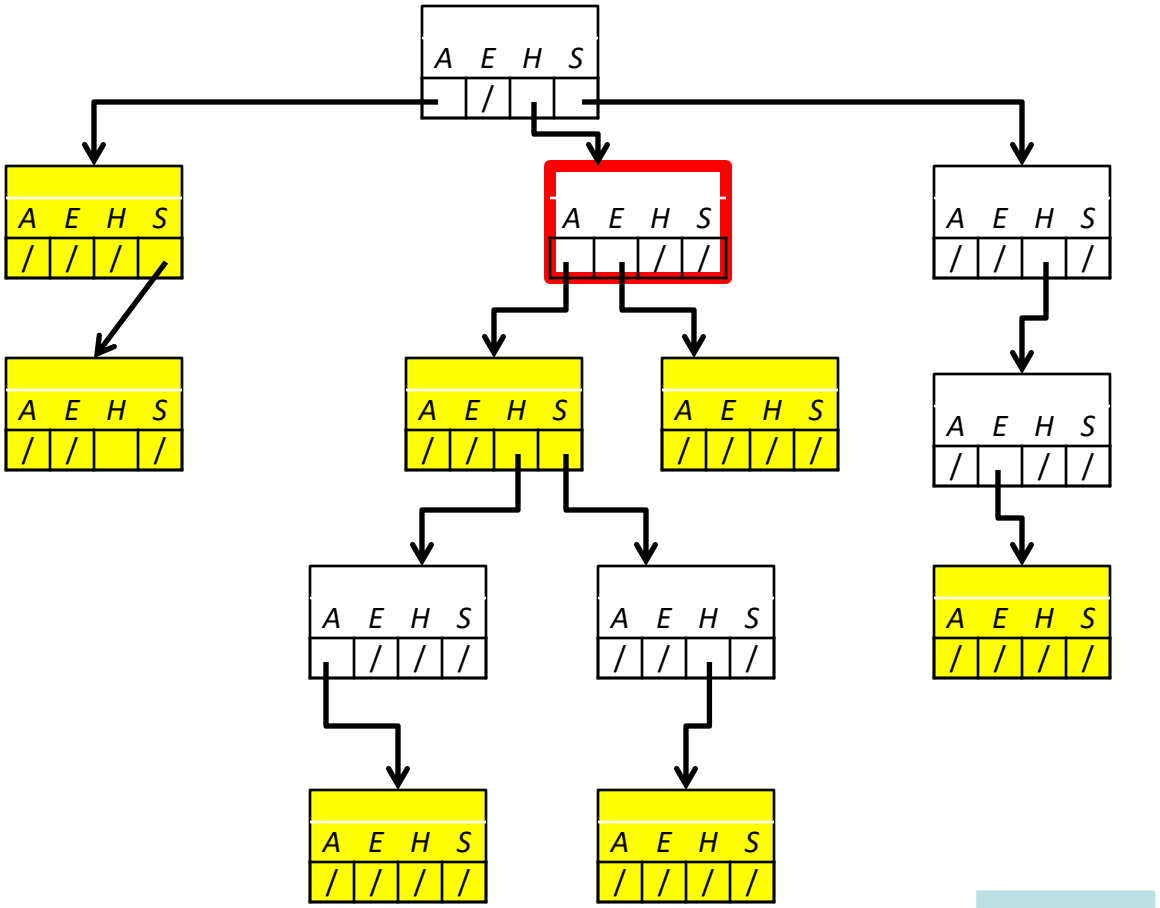
Yellow = word in the trie



Reading Words

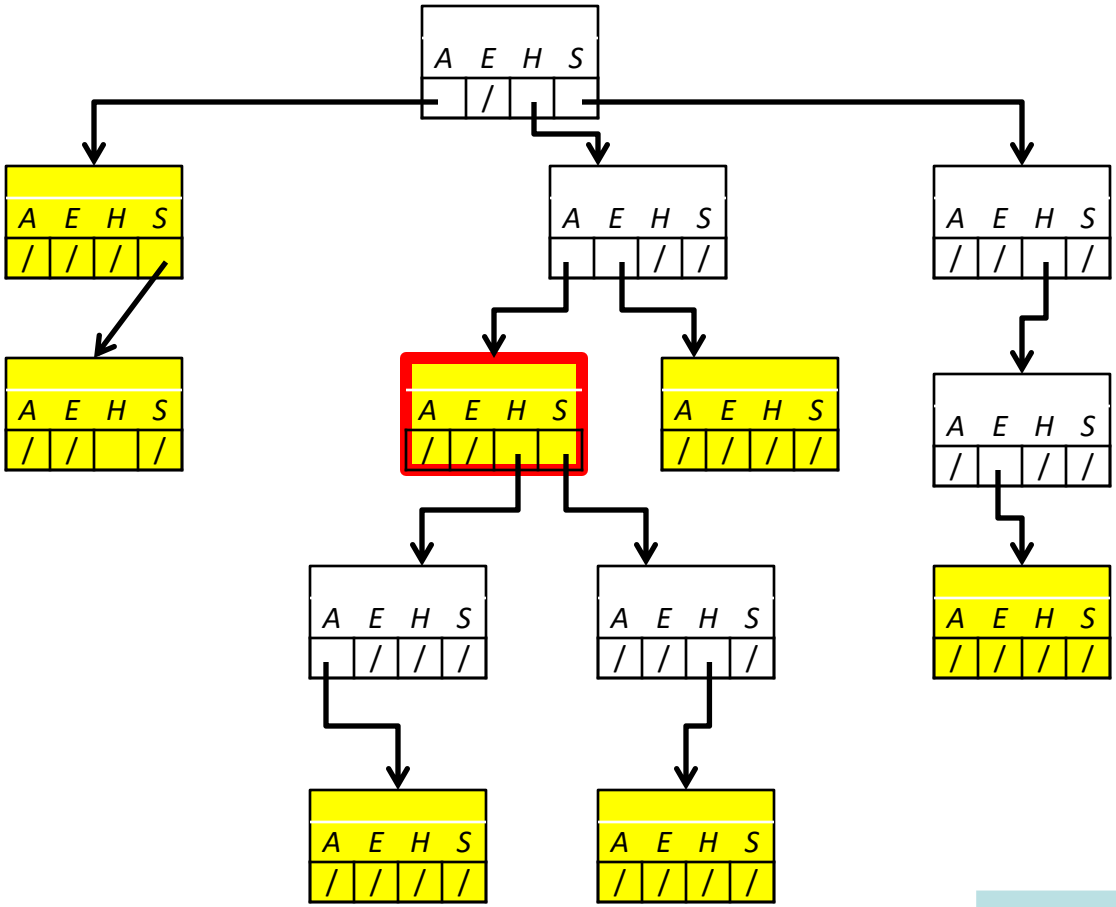


Reading Words



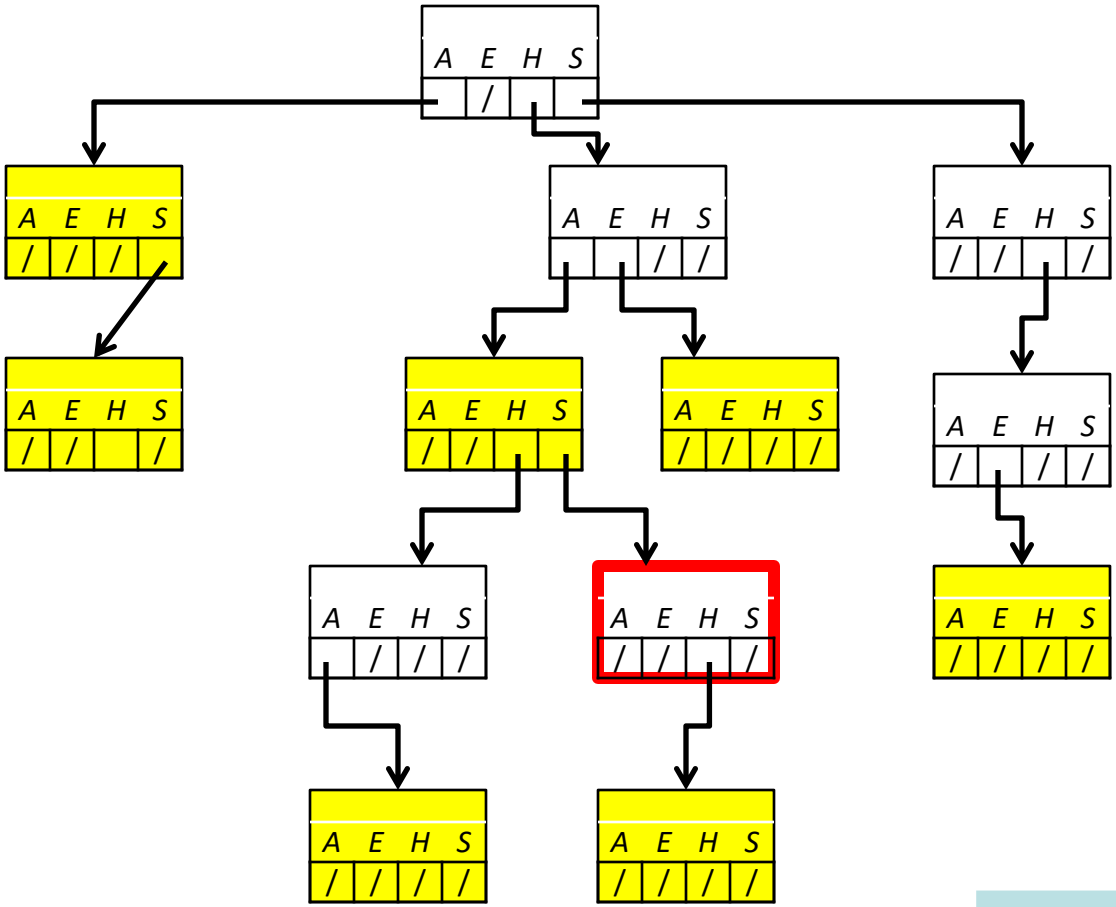
h

Reading Words



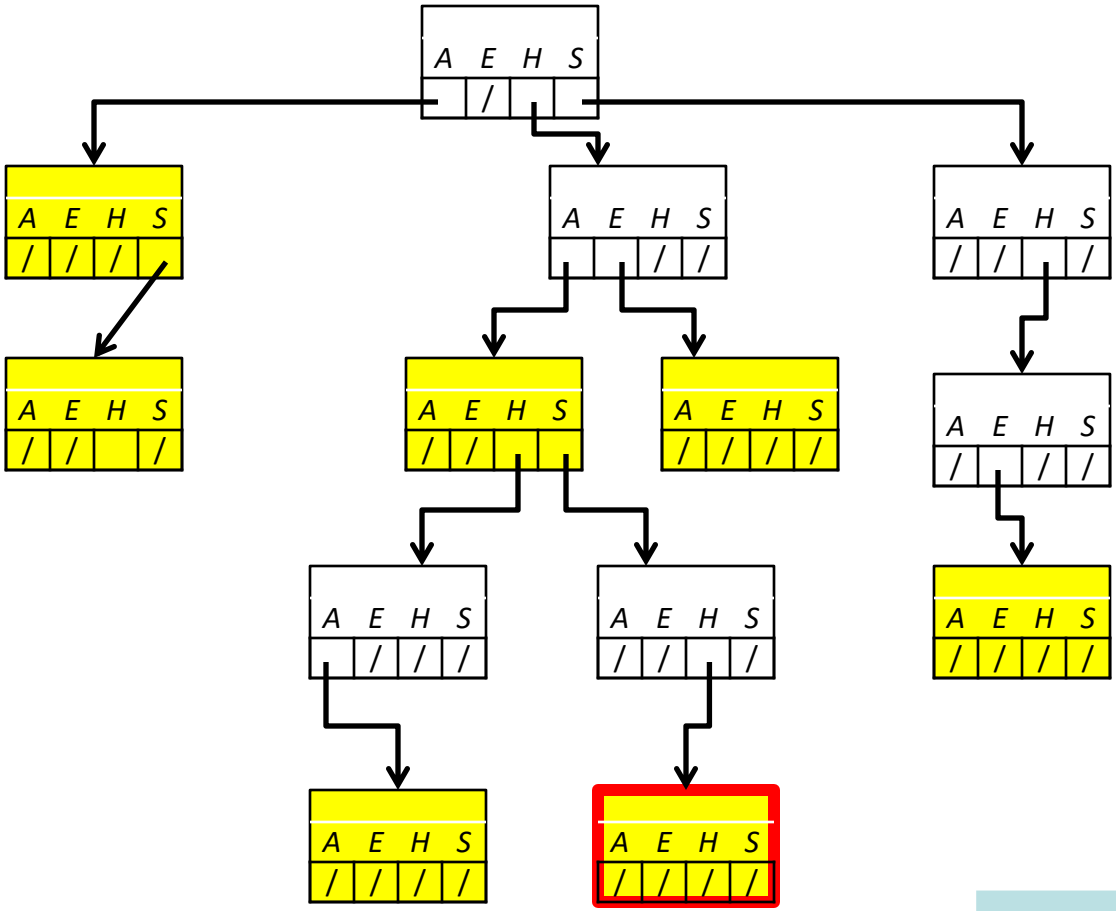
h a

Reading Words



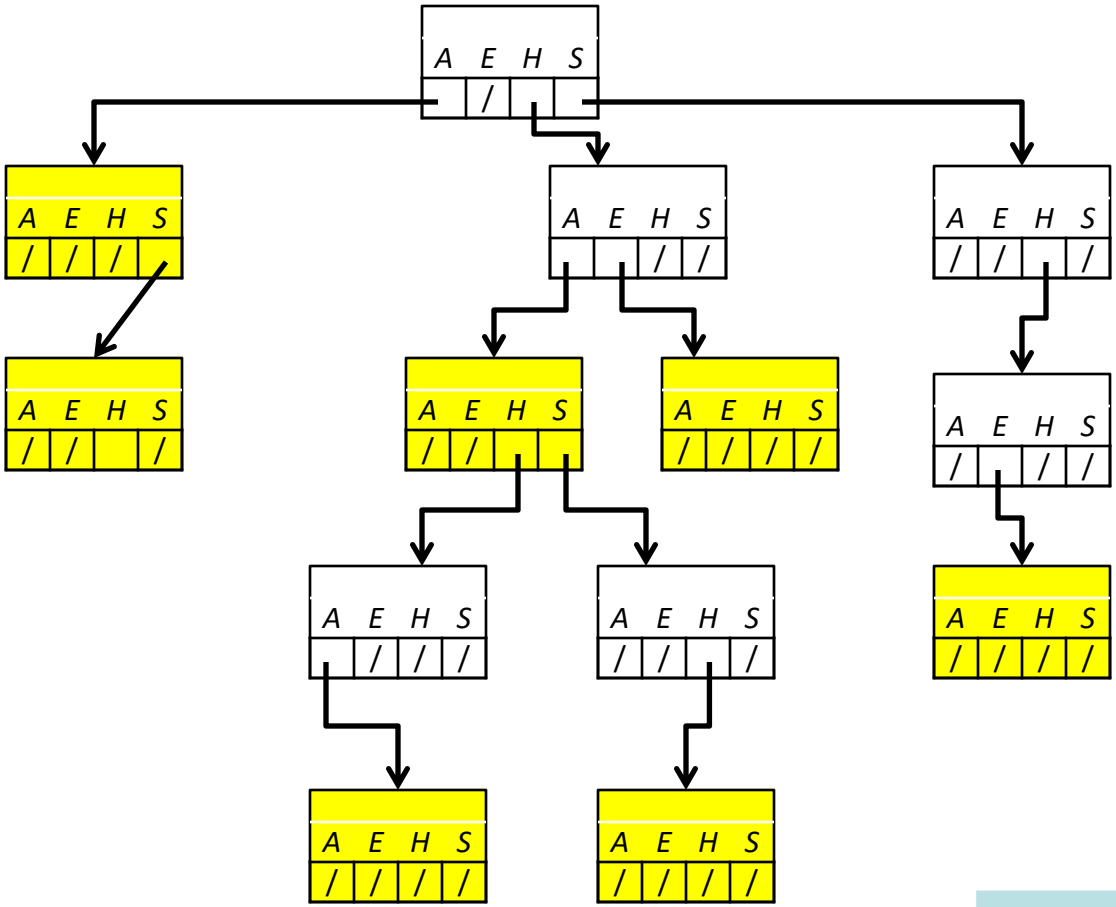
h	a	s

Reading Words



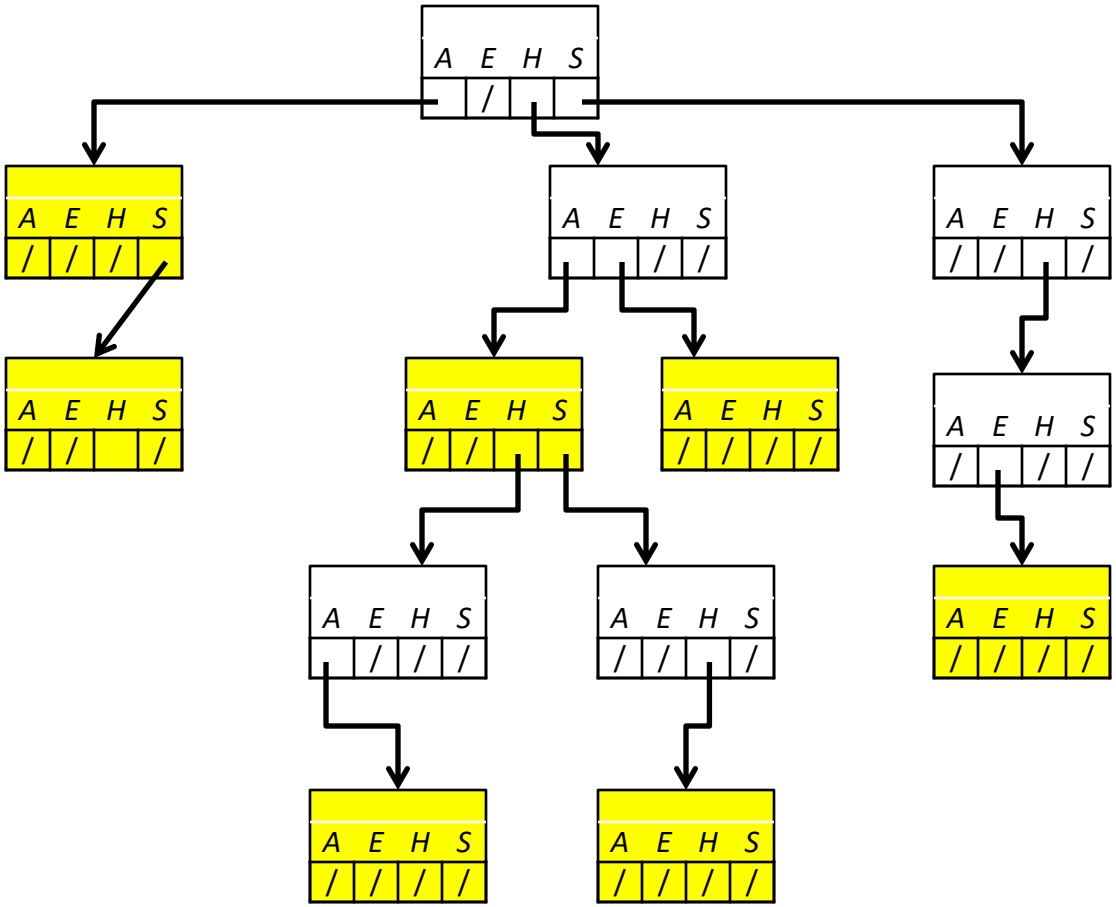
h	a	s	h

Reading Words



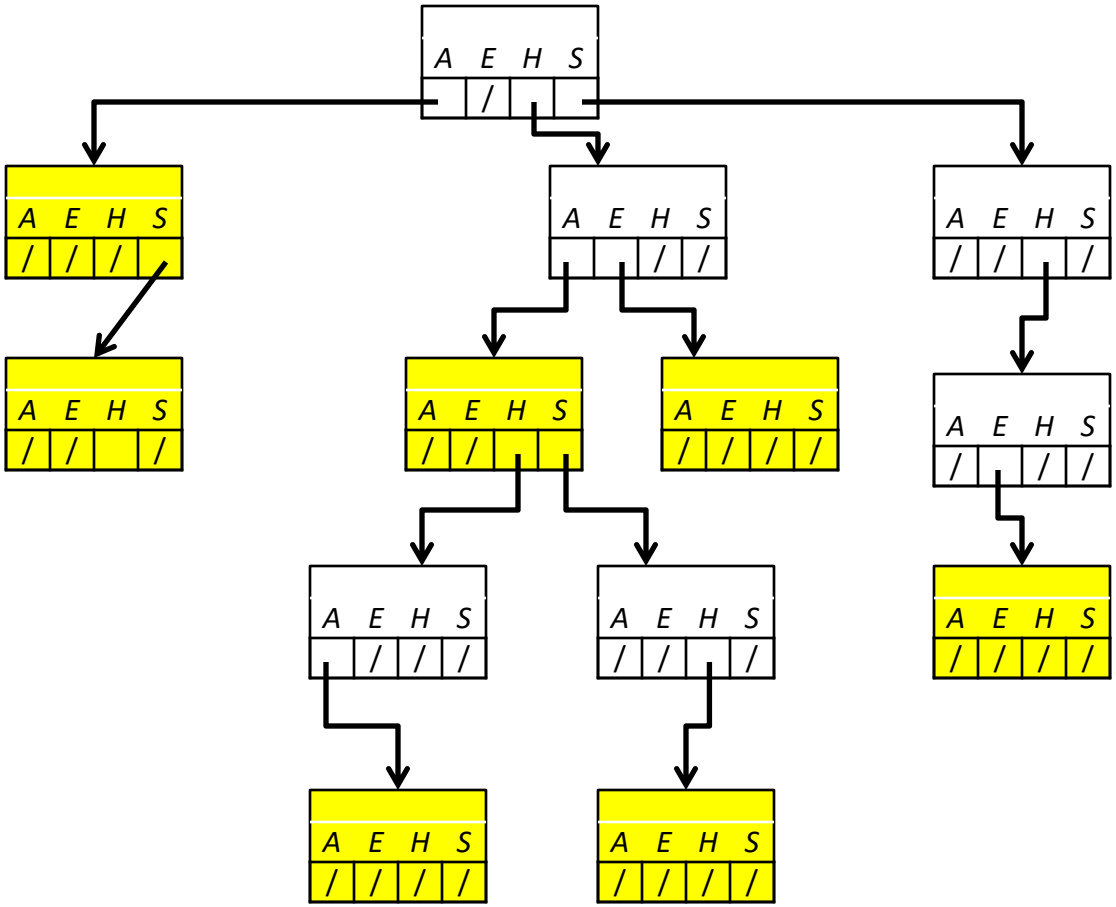
h	a	s	h

Reading Words



What are all the words represented by this trie?

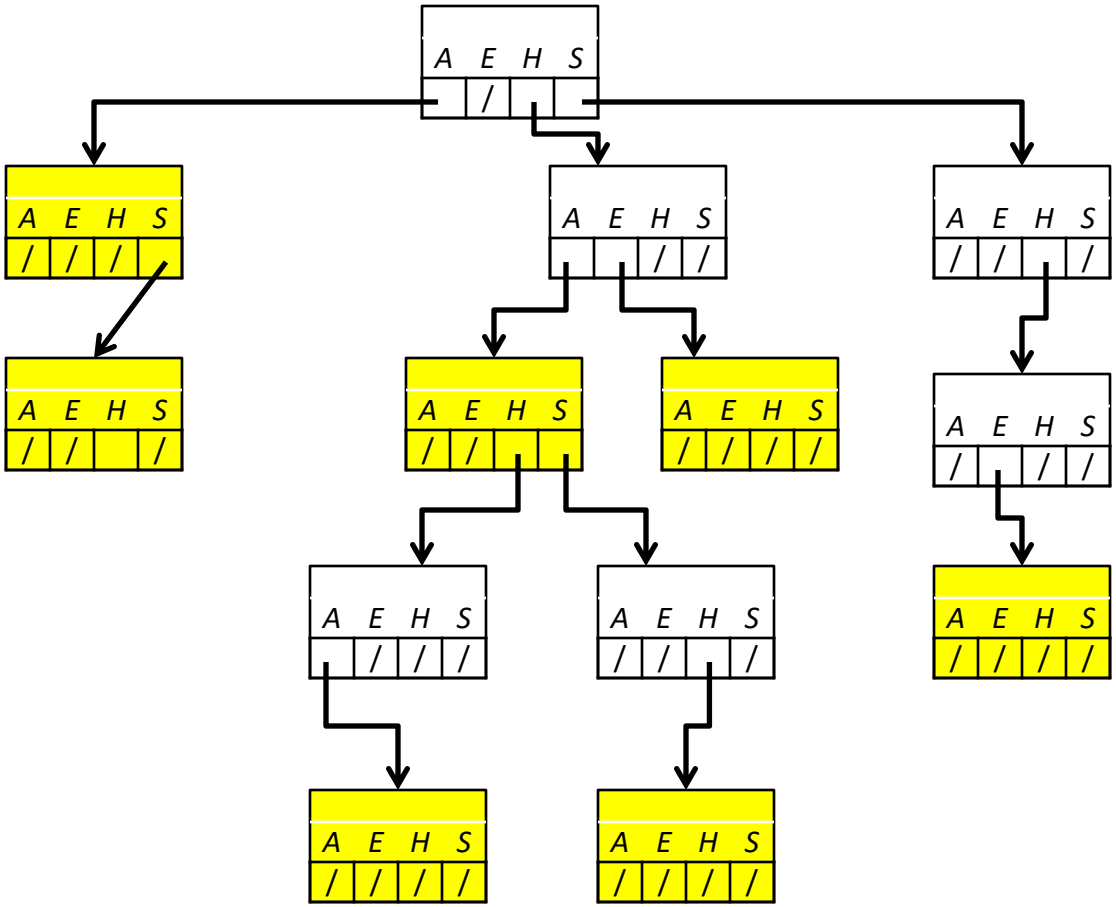
Reading Words



What are all the words represented by this trie?

a
as
ha
haha
he
she

Reading Words



What are all the words represented by this trie?

a
as
ha
haha
he
she

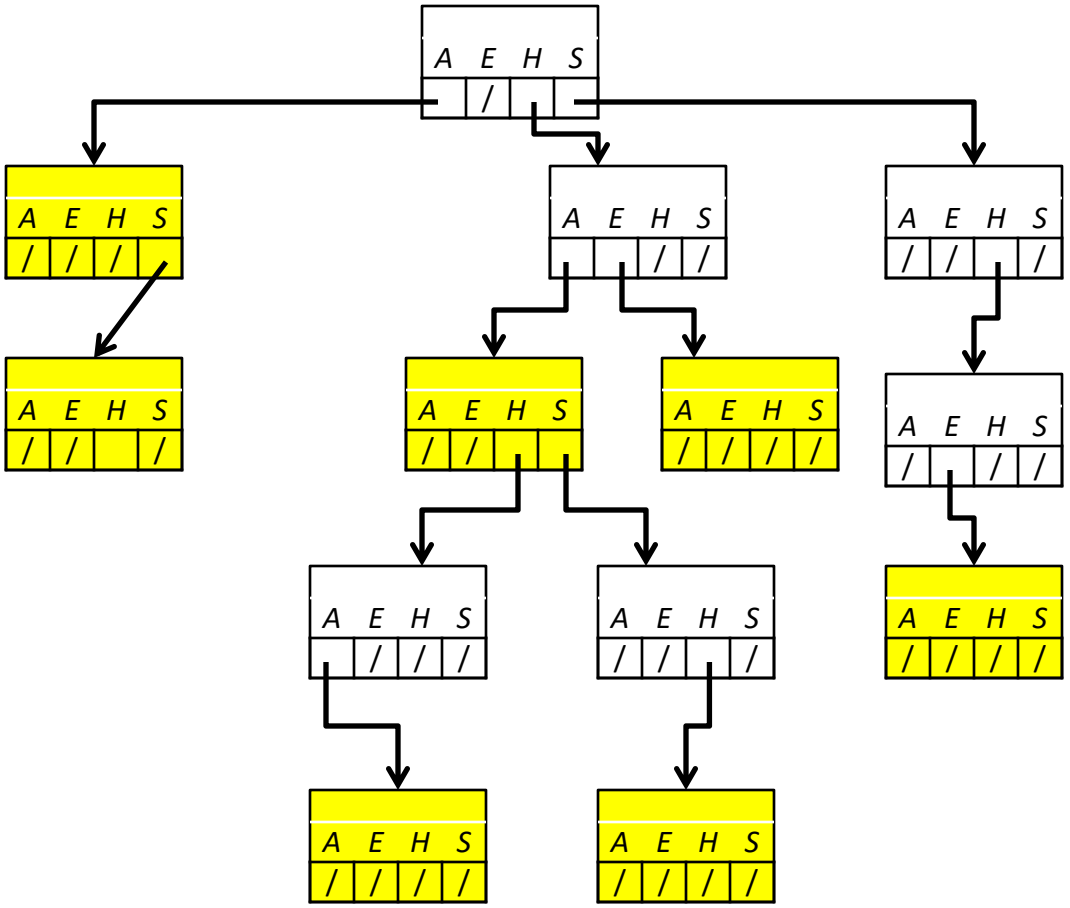
How can we write a function to print this?

PrintAllWords

```
void printAllWords(TrieNode* root) {
    printAllWordsHelper(root, "");
}

void printAllWordsHelper(TrieNode* root, string prefix) {
    if (root == nullptr) {
        return;
    }
    if (root->isWord) {
        cout << prefix << endl;
    }
    for (int i = 0; i < 26; i++) {
        printAllWordsHelper(root->children[i], prefix + char('a' + i));
    }
}
```

Prefixes



How can we write `containsPrefix`?

```
containsPrefix("a")   => true
containsPrefix("se") => false
```

containsPrefix

```
bool containsPrefix(TrieNode* node, string prefix) {
    if (node == nullptr) {
        return false;
    }
    if (prefix.length() == 0) {
        return true;
    }
    return containsPrefix(node->children[prefix[0] - 'a'],
                          prefix.substr(1));
}
```

Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

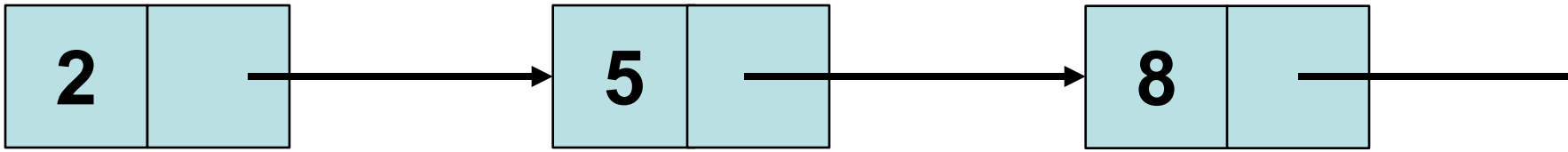
Announcements

- Brown Institute Data Visualization @ NY Times talk with Kevin Quealy Tues. 11/13 4:30PM in Packard 101

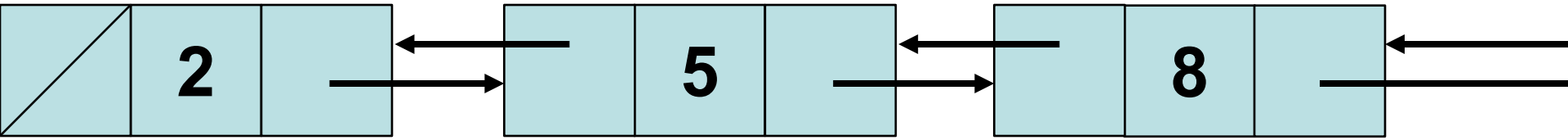
Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

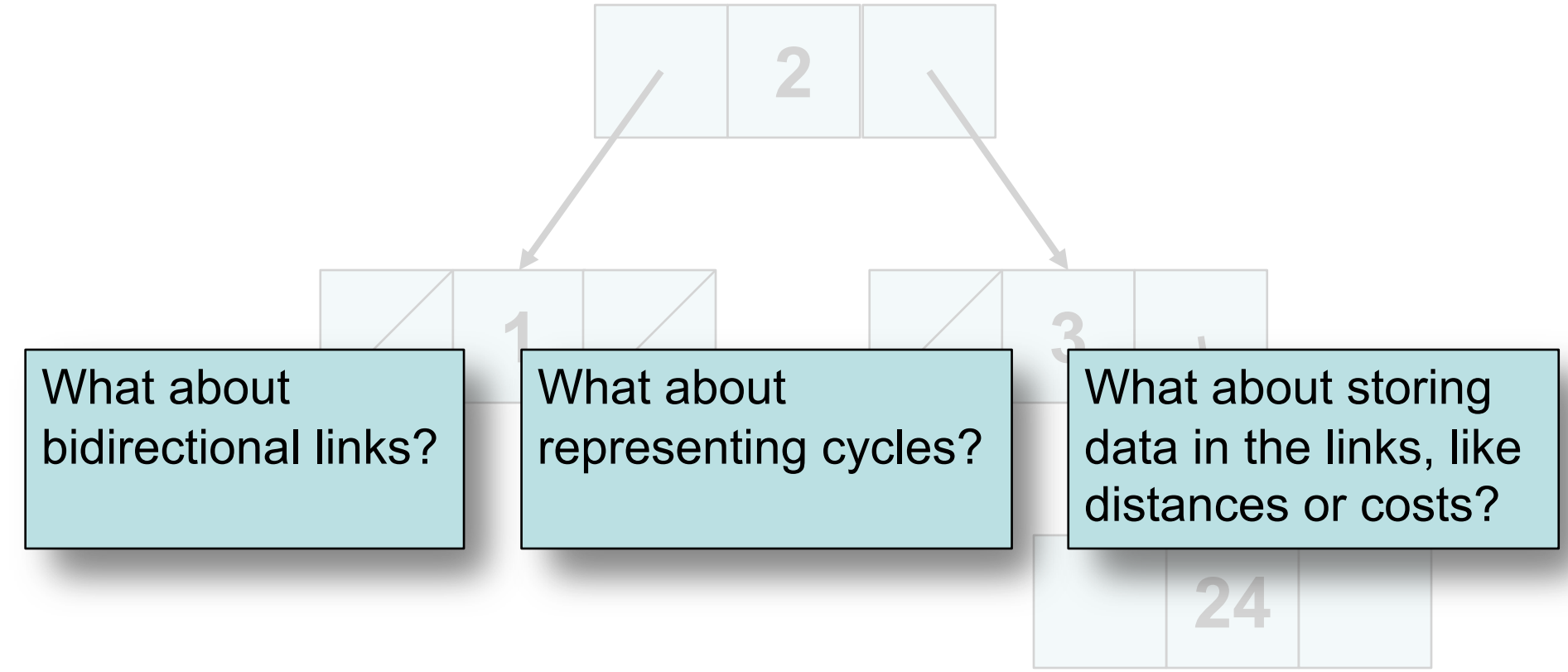
Linked Structures



Linked Structures



Linked Structures



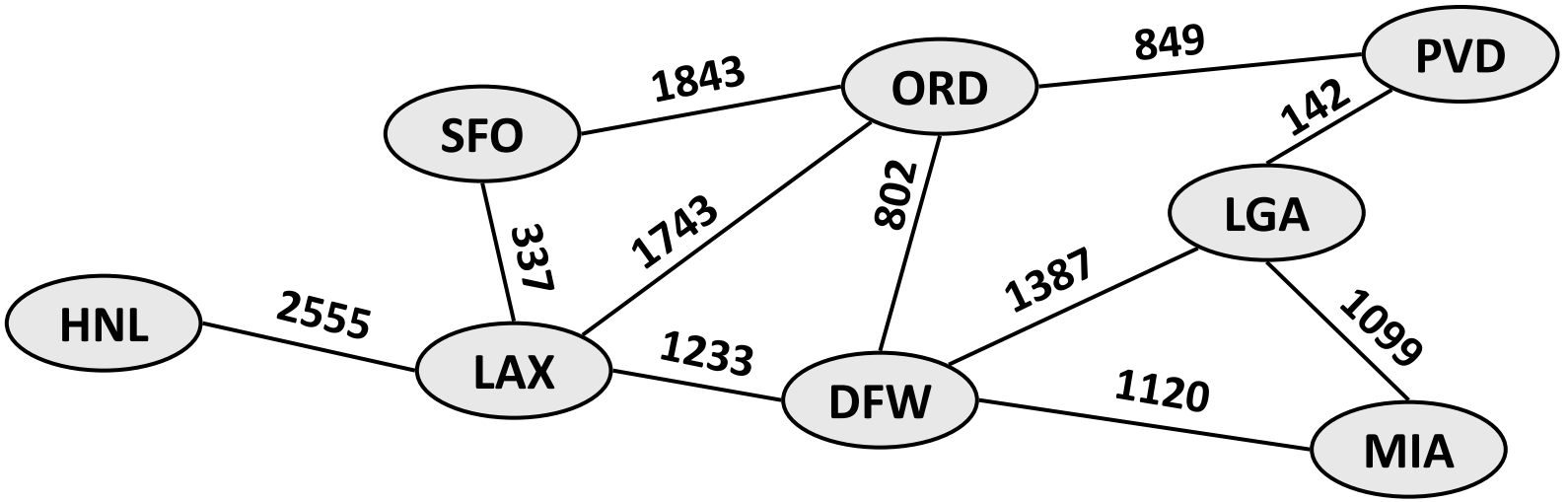
What about bidirectional links?

What about representing cycles?

What about storing data in the links, like distances or costs?

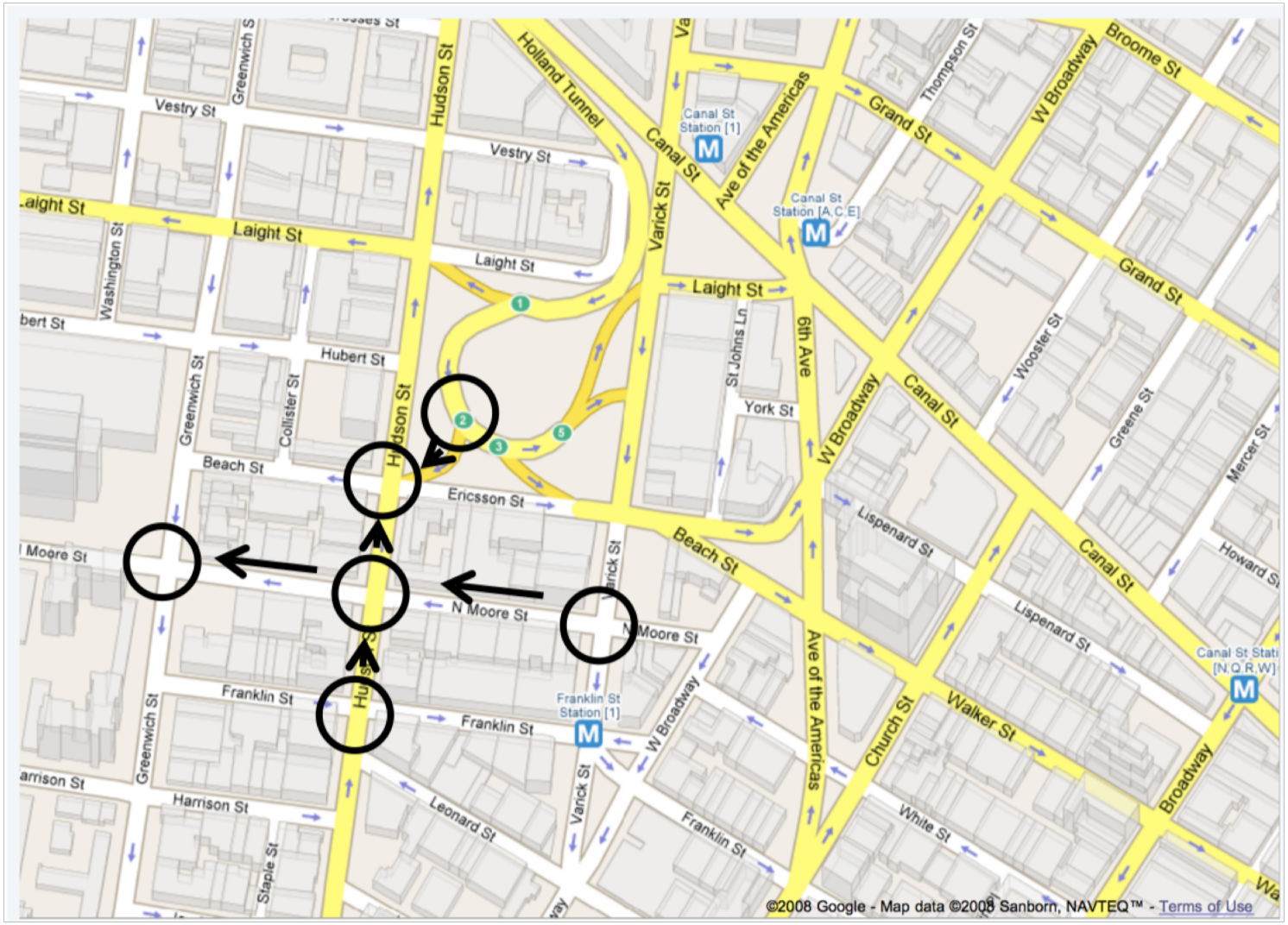
Linked Structures: Shortfalls

Representing airline flights and distances:

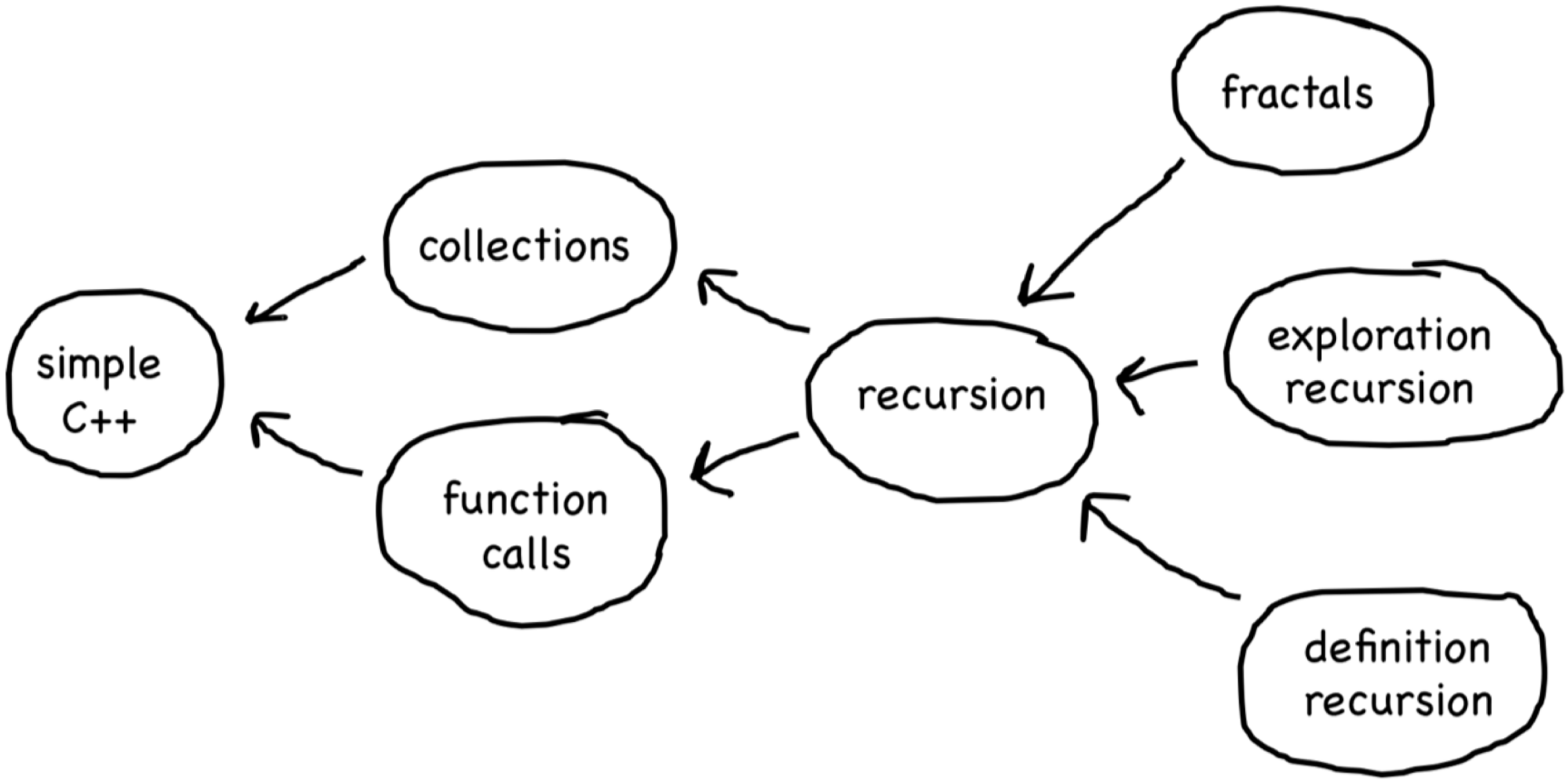


Linked Structures: Shortfalls

Linked Structures: Shortfalls

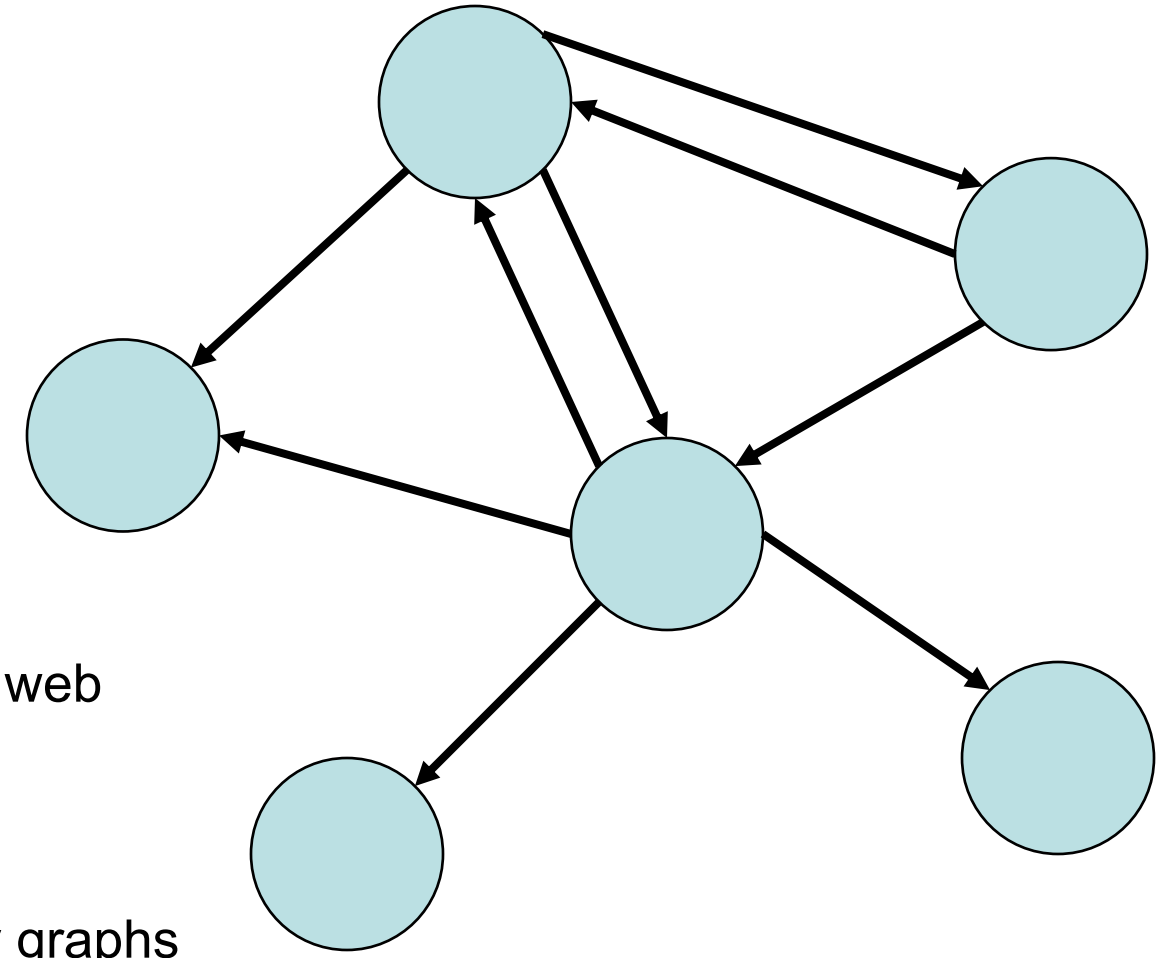


Linked Structures: Shortfalls



Graphs

A graph consists of a set of **nodes** connected by **edges**.

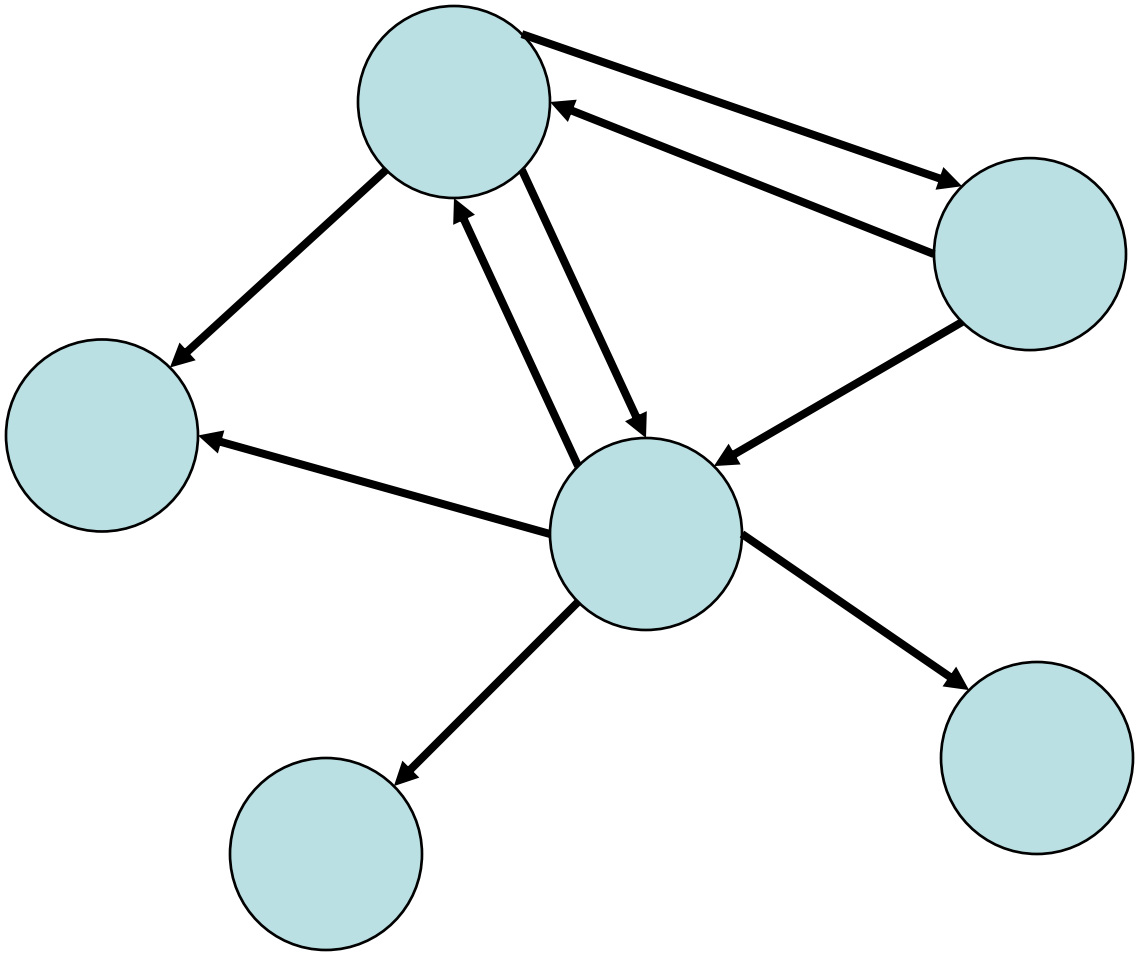


Graphs can model:

- Sites and links on the web
- Disease outbreaks
- Social networks
- Geographies
- Task and dependency graphs
- and more…

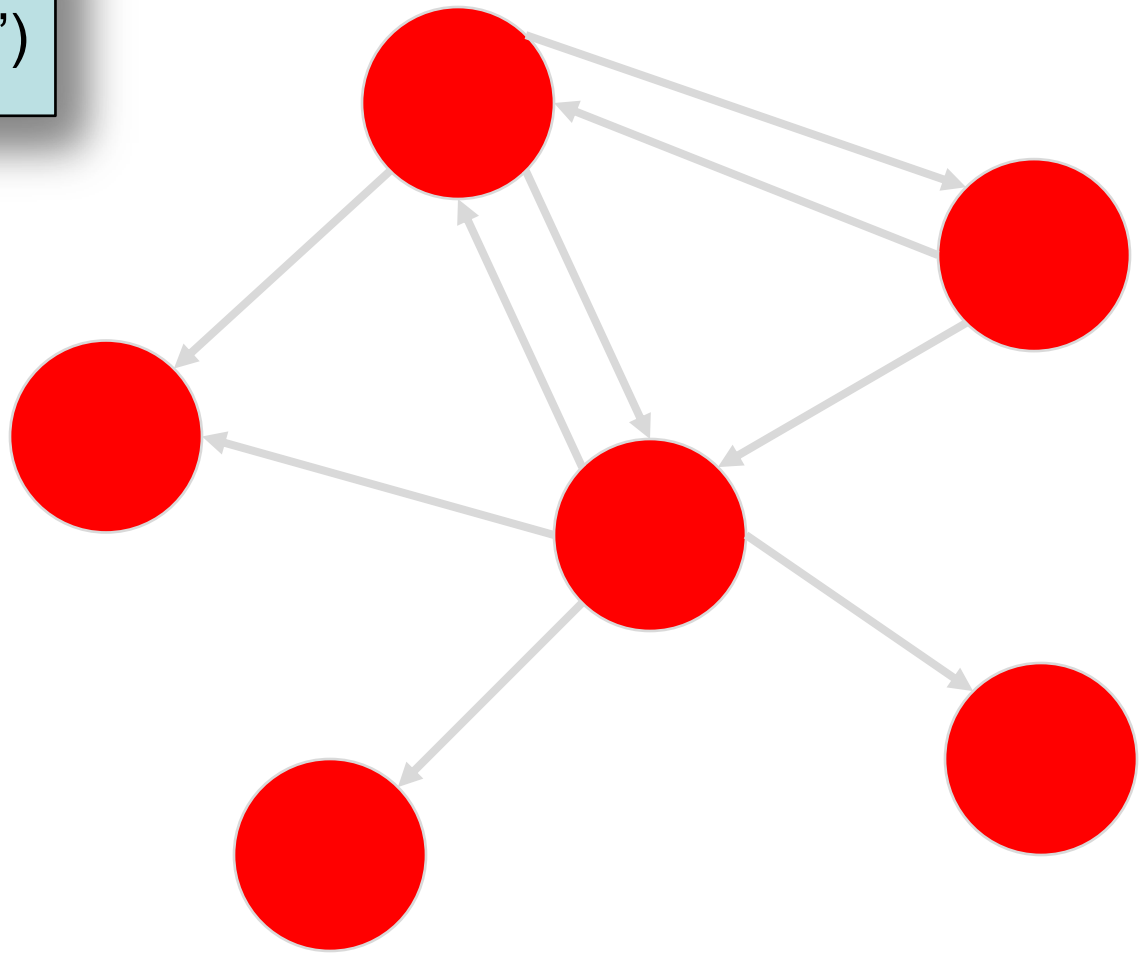
Graphs

A graph consists of a set of **nodes** connected by **edges**.



Nodes

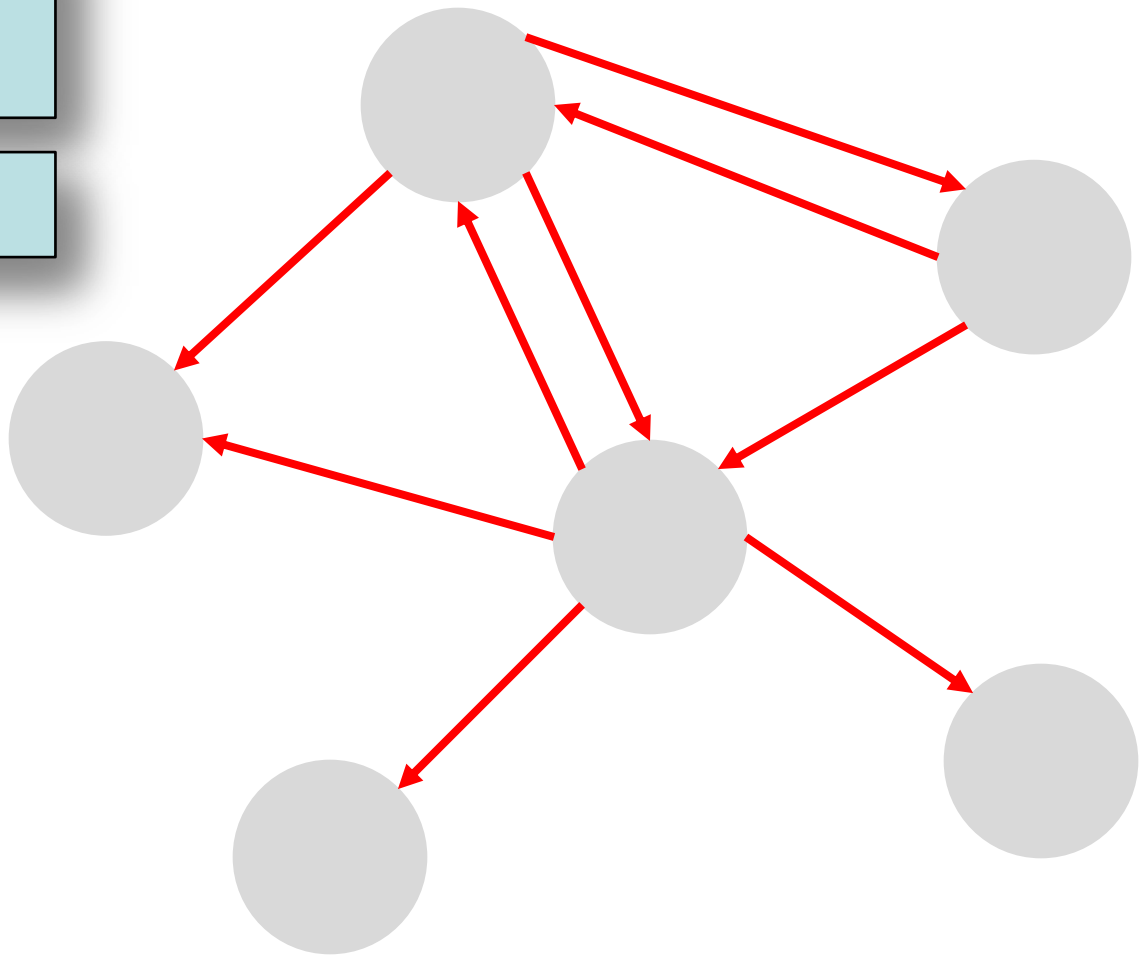
Nodes
("Vertices")



Edges

Edges
("Arcs")

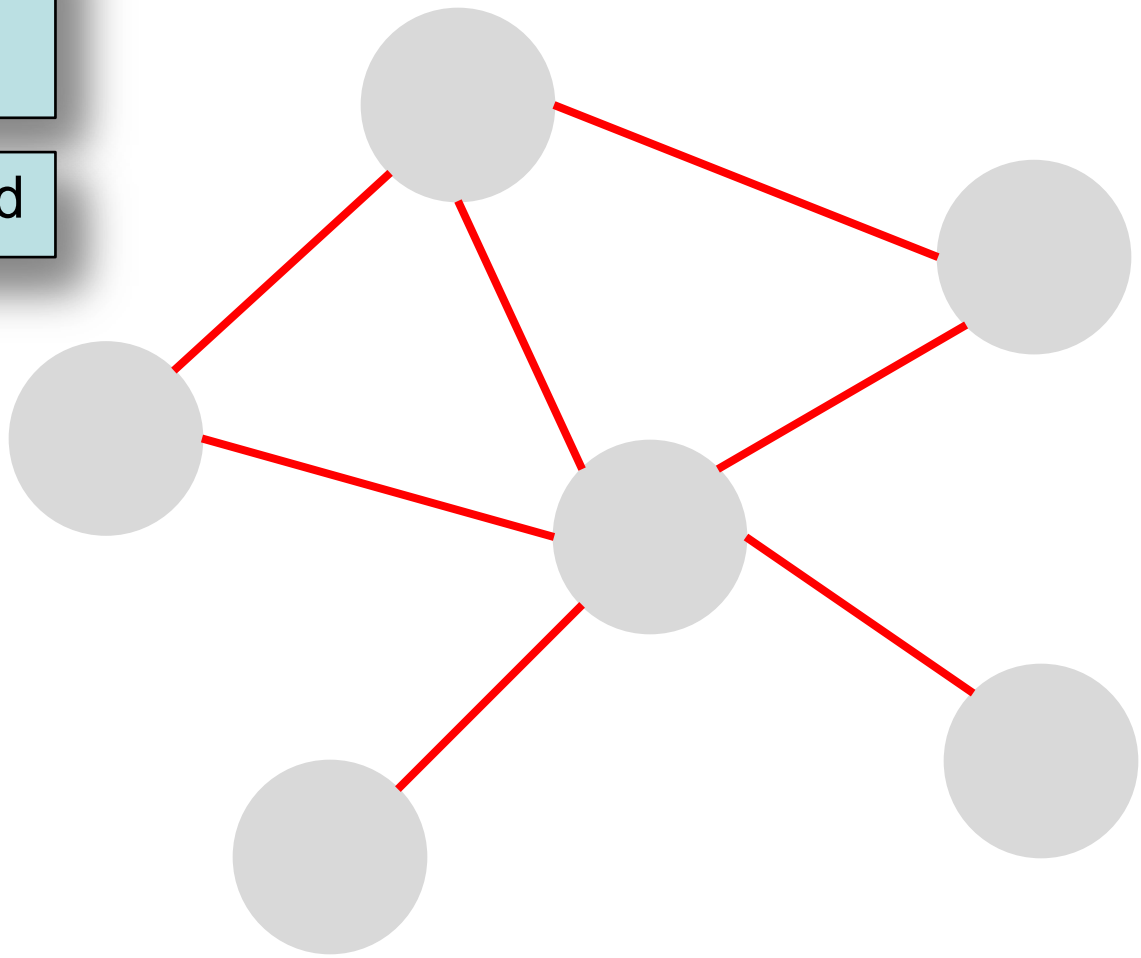
Directed



Edges

Edges
("Arcs")

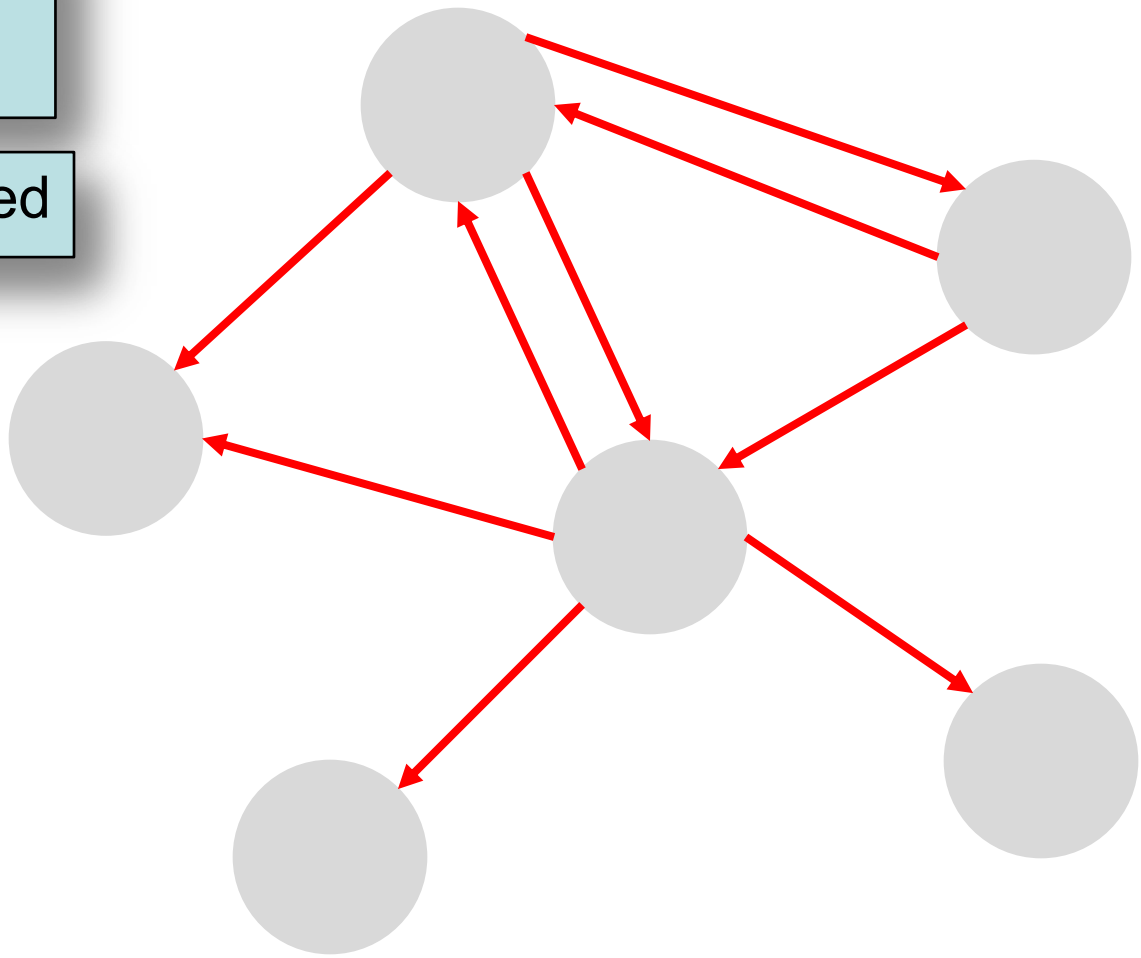
Undirected



Edges

Edges
("Arcs")

Unweighted

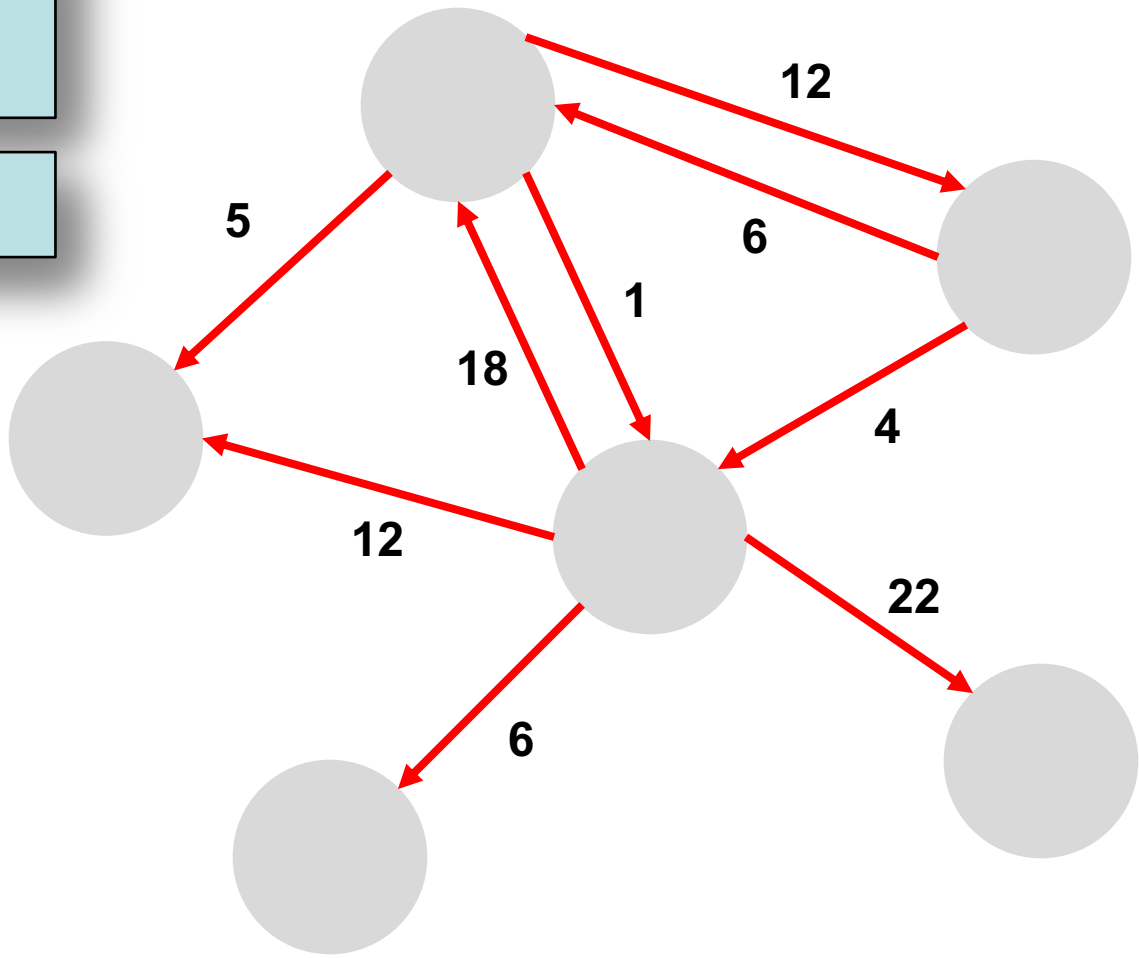


Weights

Edges ("Arcs")

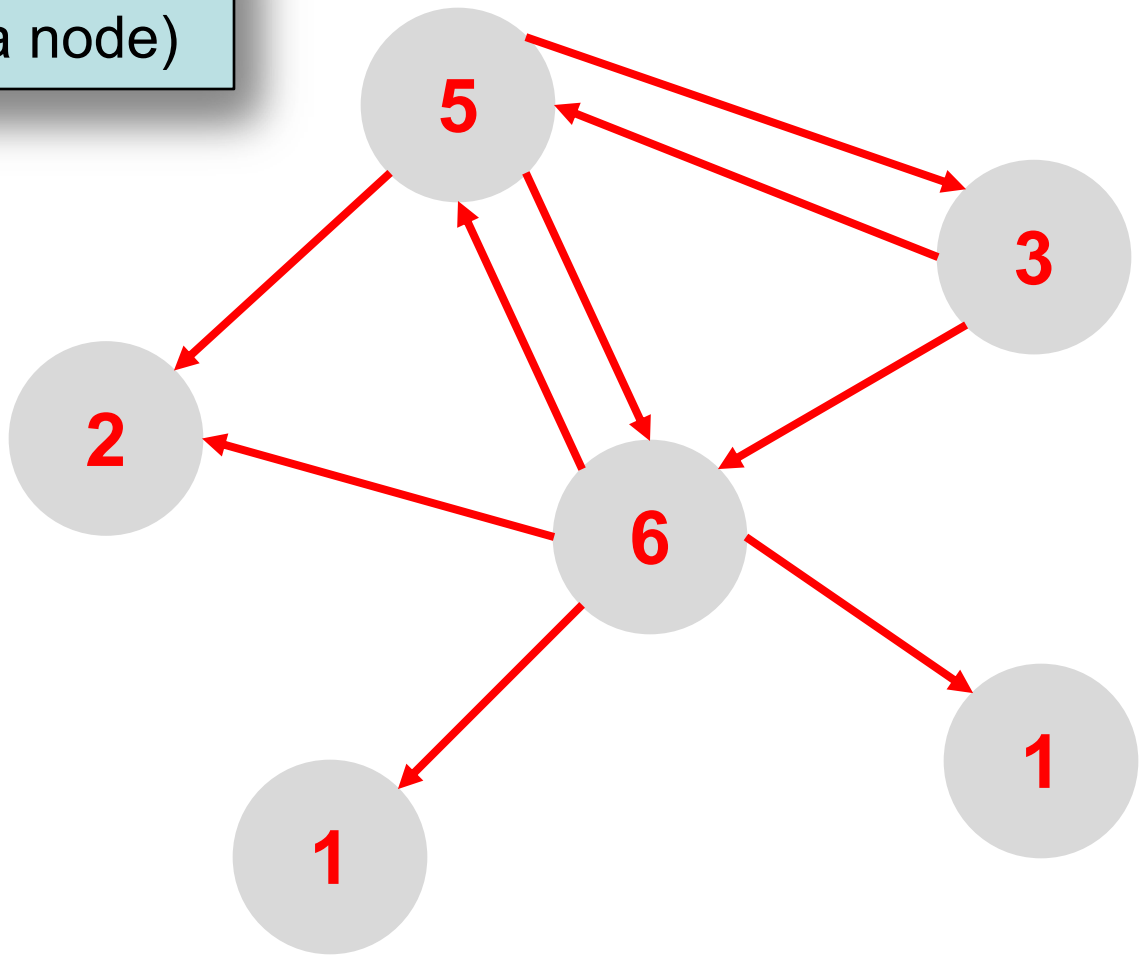
Weighted

A weight is a cost associated with a given edge.



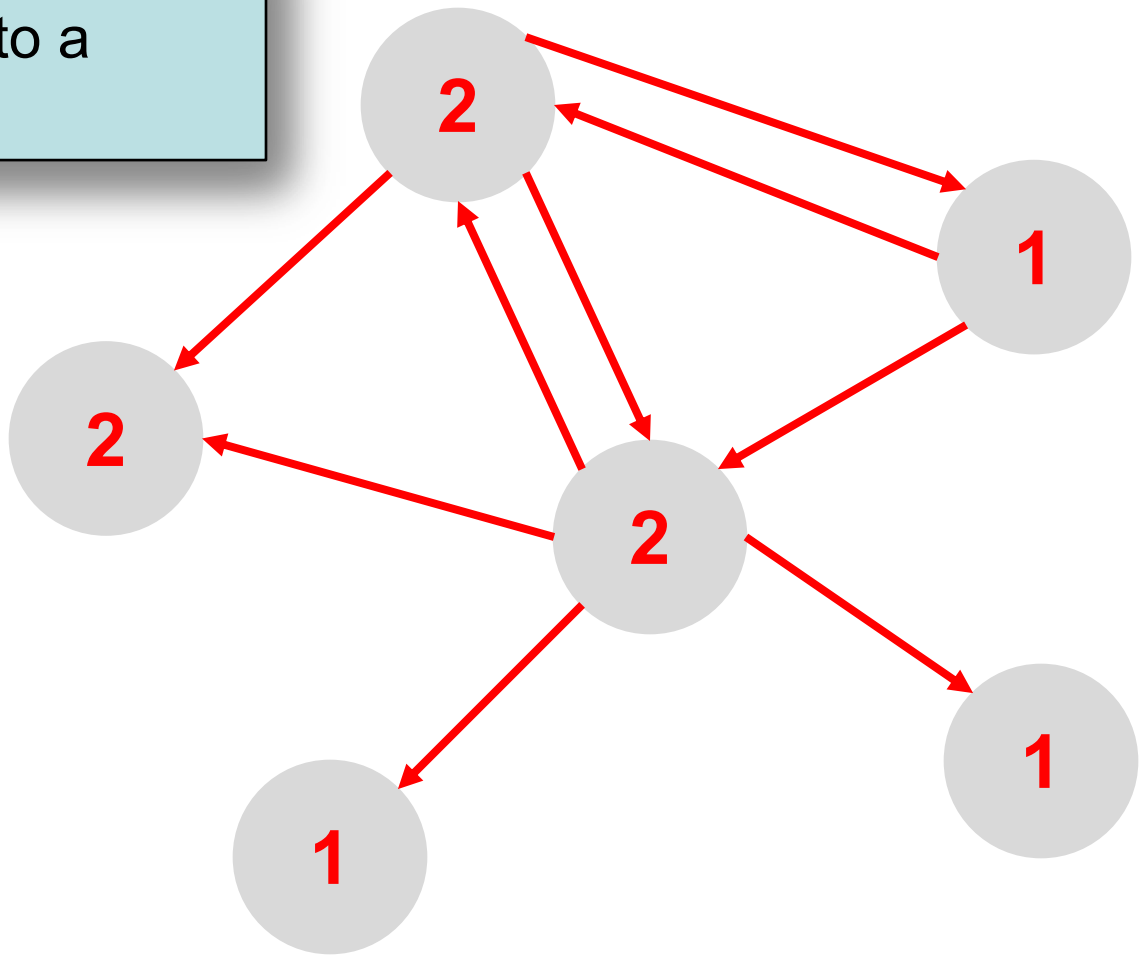
Degree

Degree (# edges touching a node)



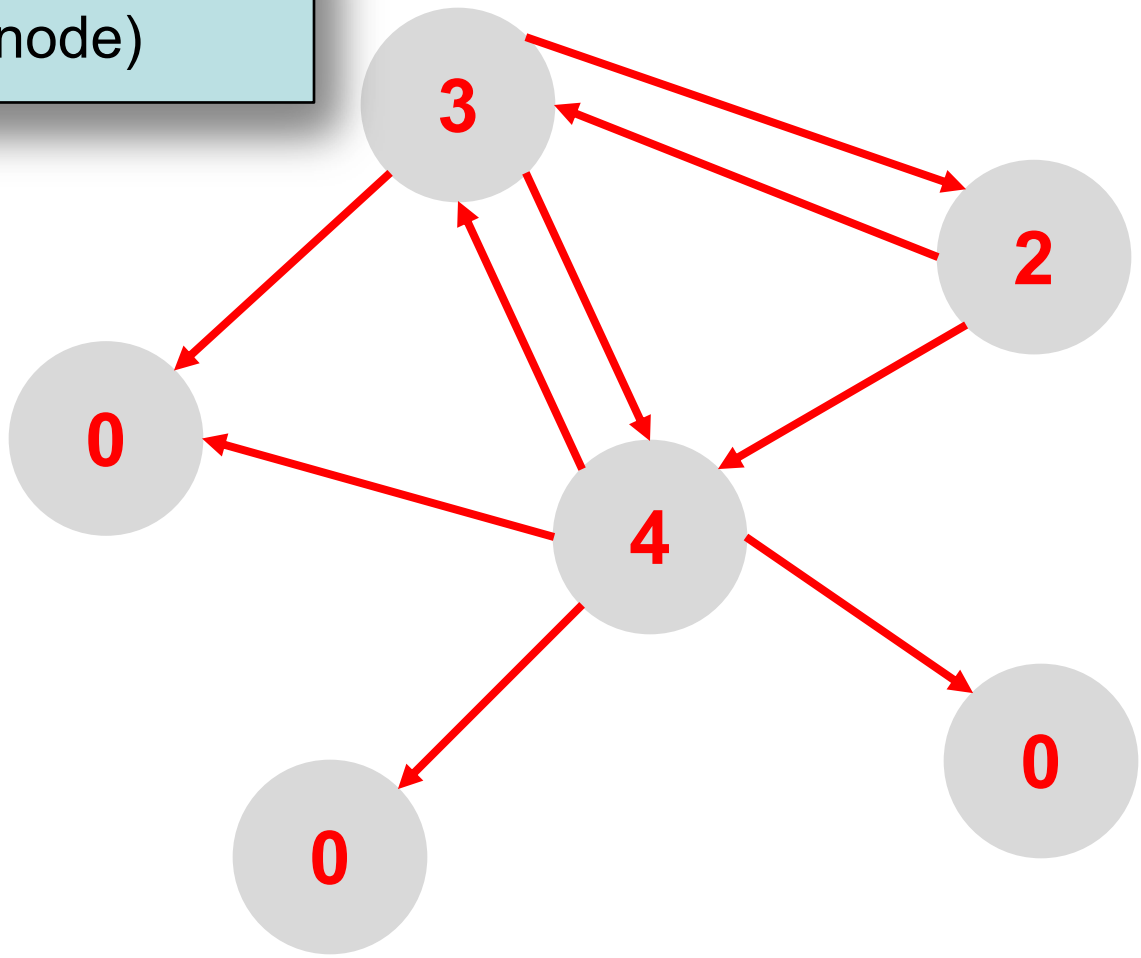
Degree

In-degree (# edges coming into a node)



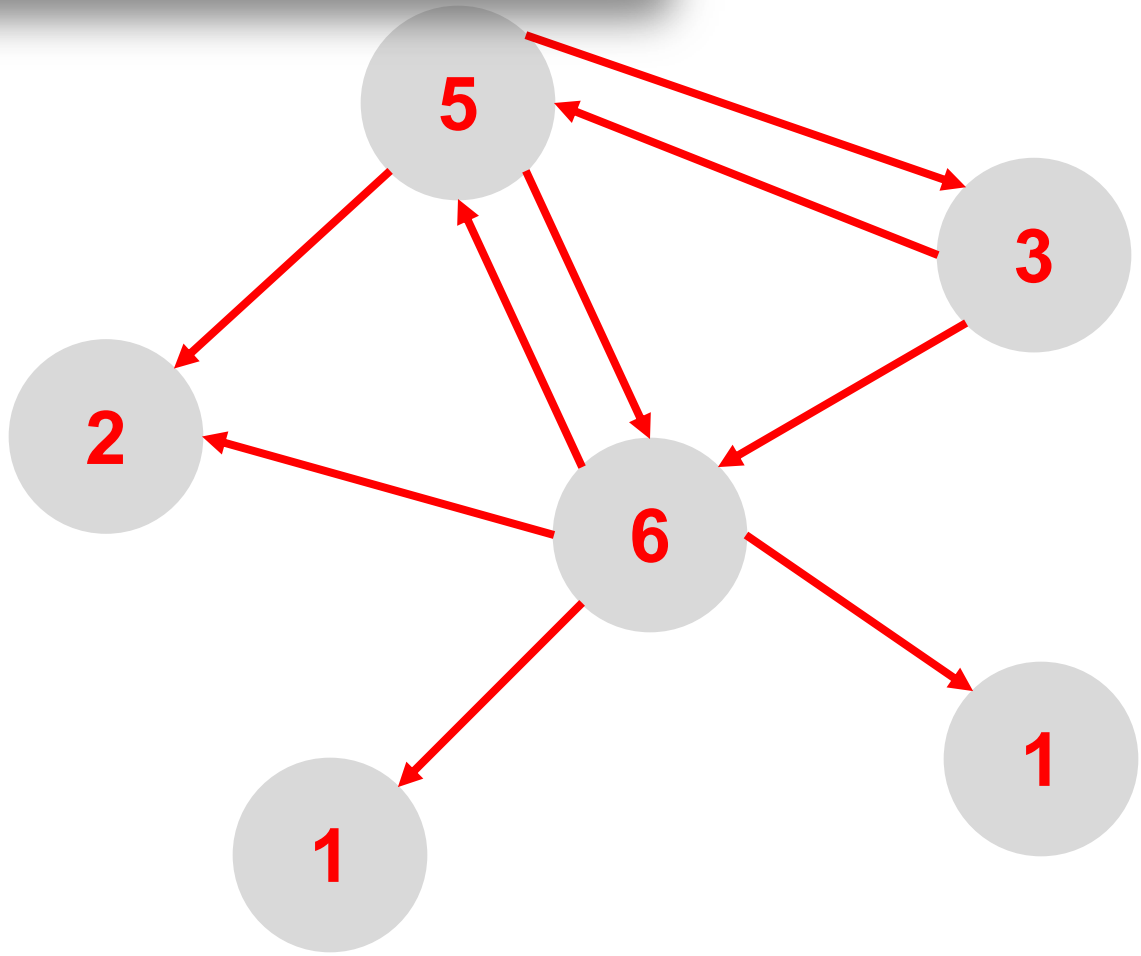
Degree

Out-degree (# edges leaving a node)



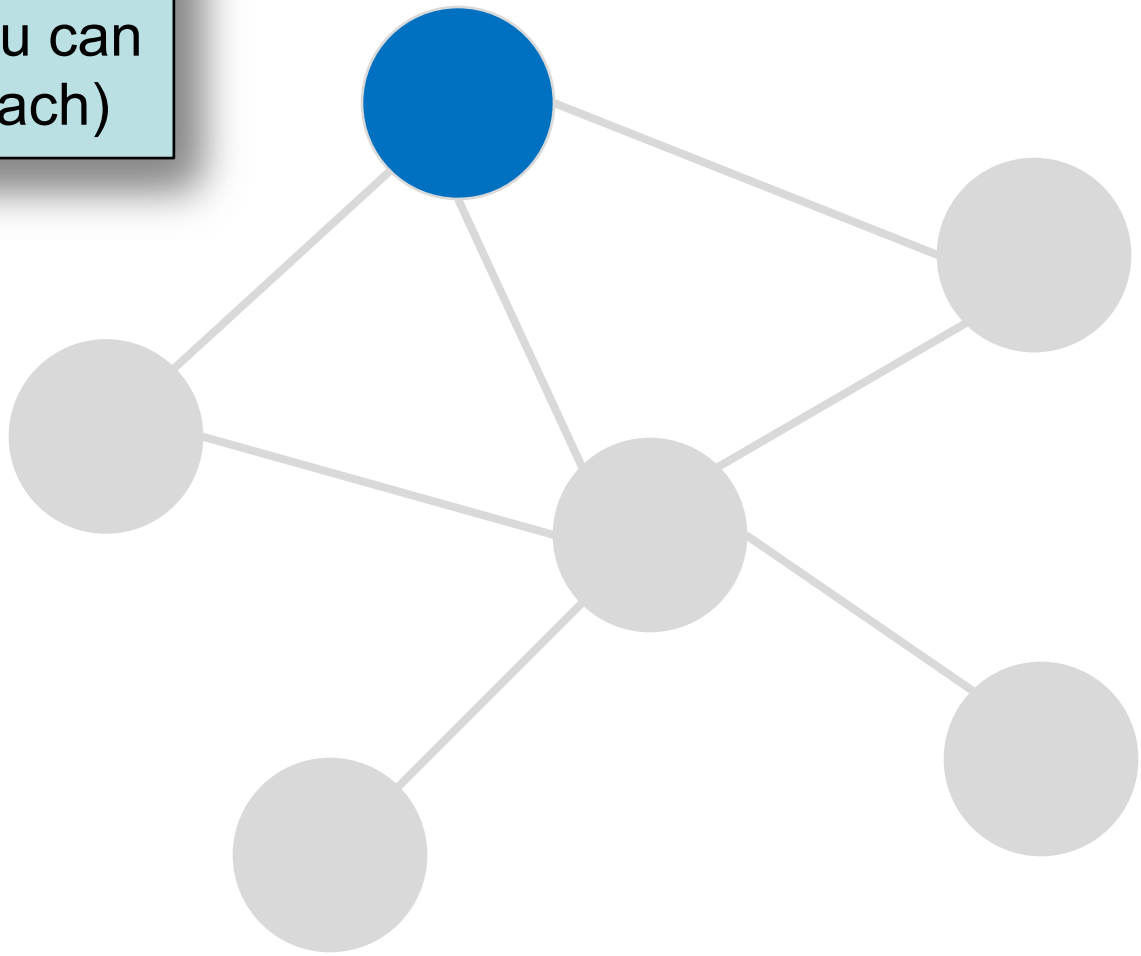
Degree

Degree = In-degree + Out-degree



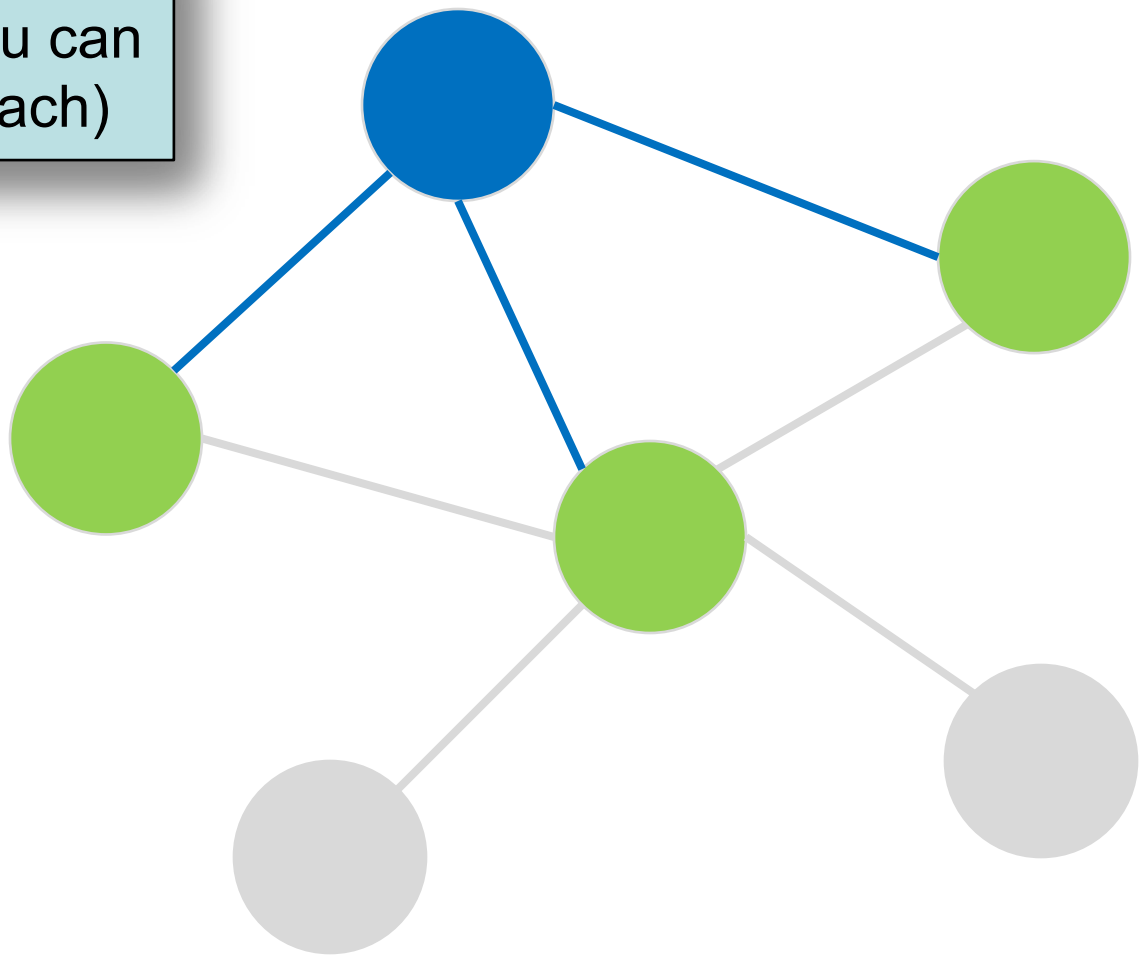
Neighbors

Neighbors (nodes you can directly reach)



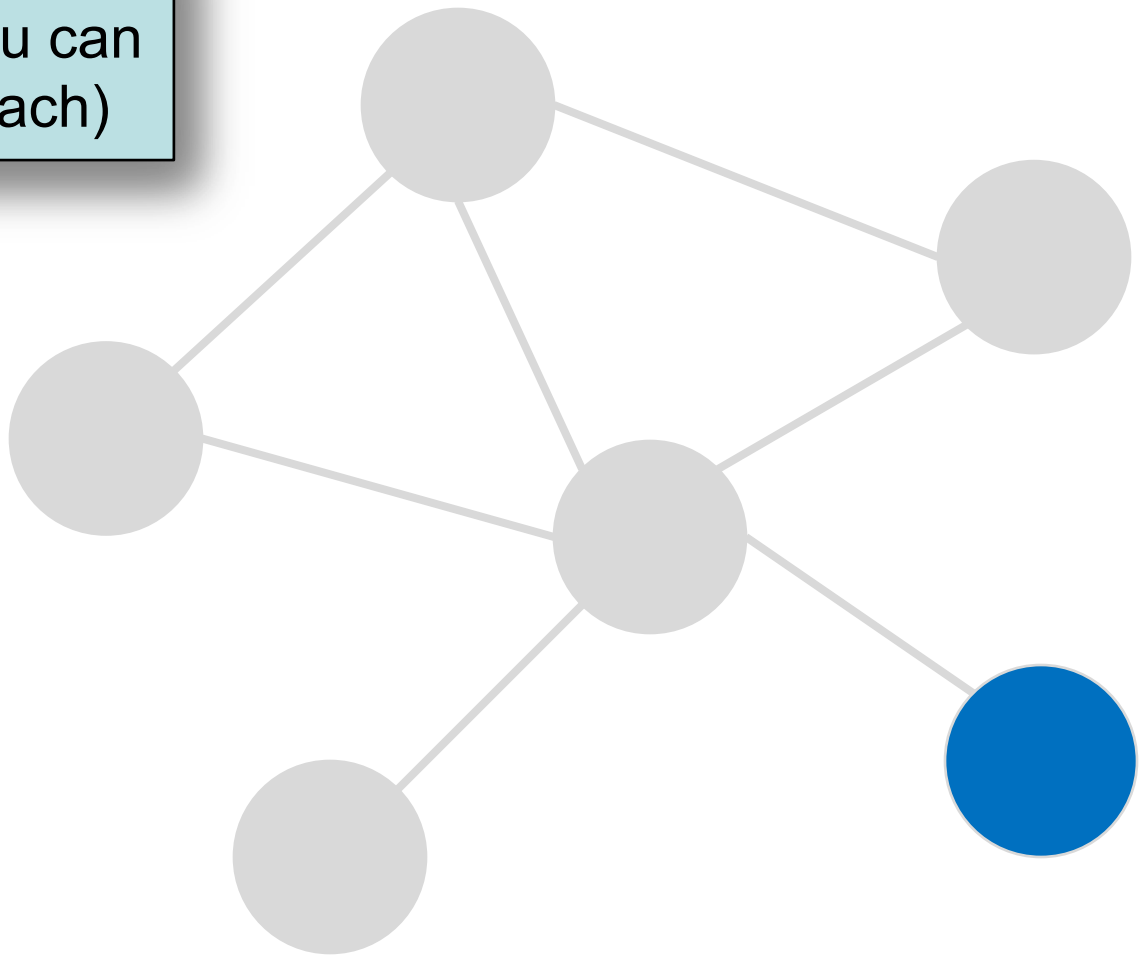
Neighbors

Neighbors (nodes you can directly reach)



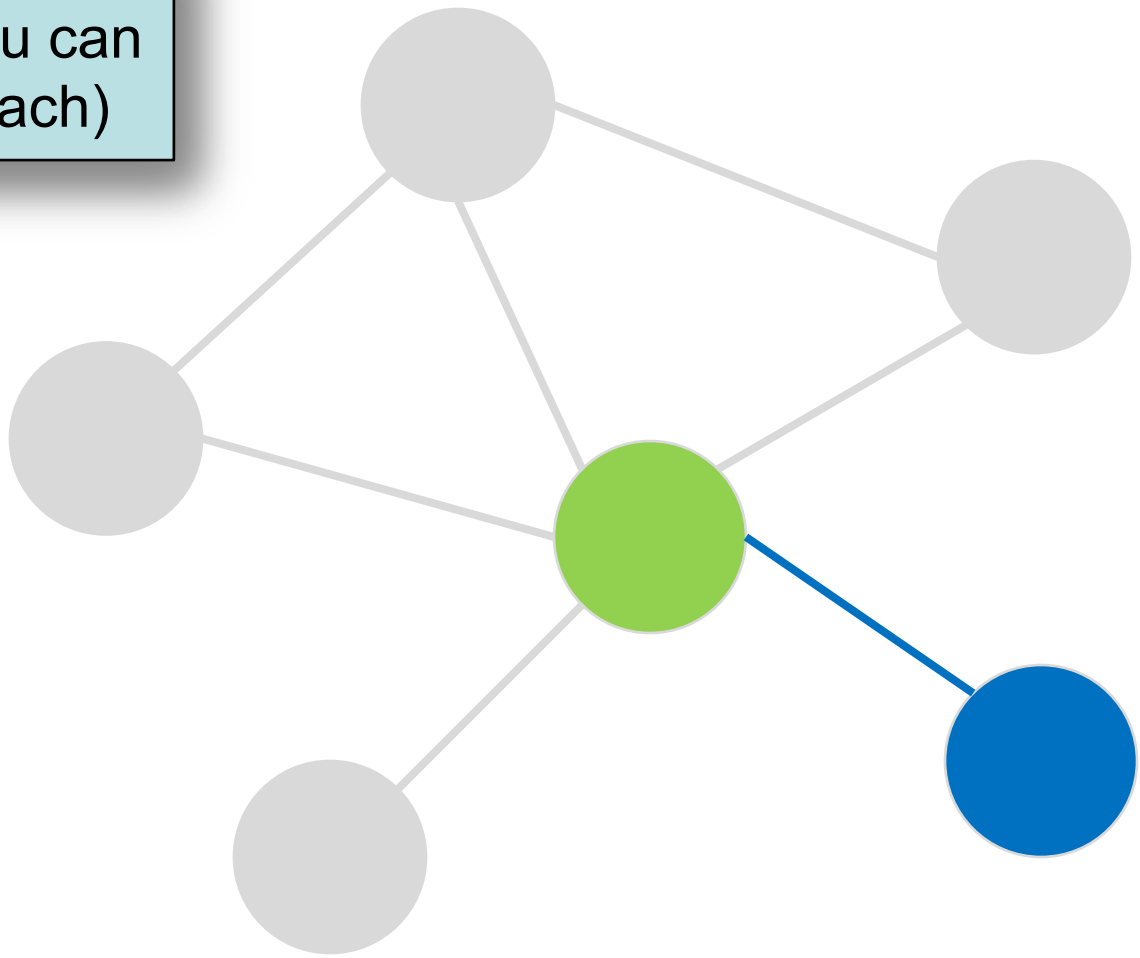
Neighbors

Neighbors (nodes you can directly reach)



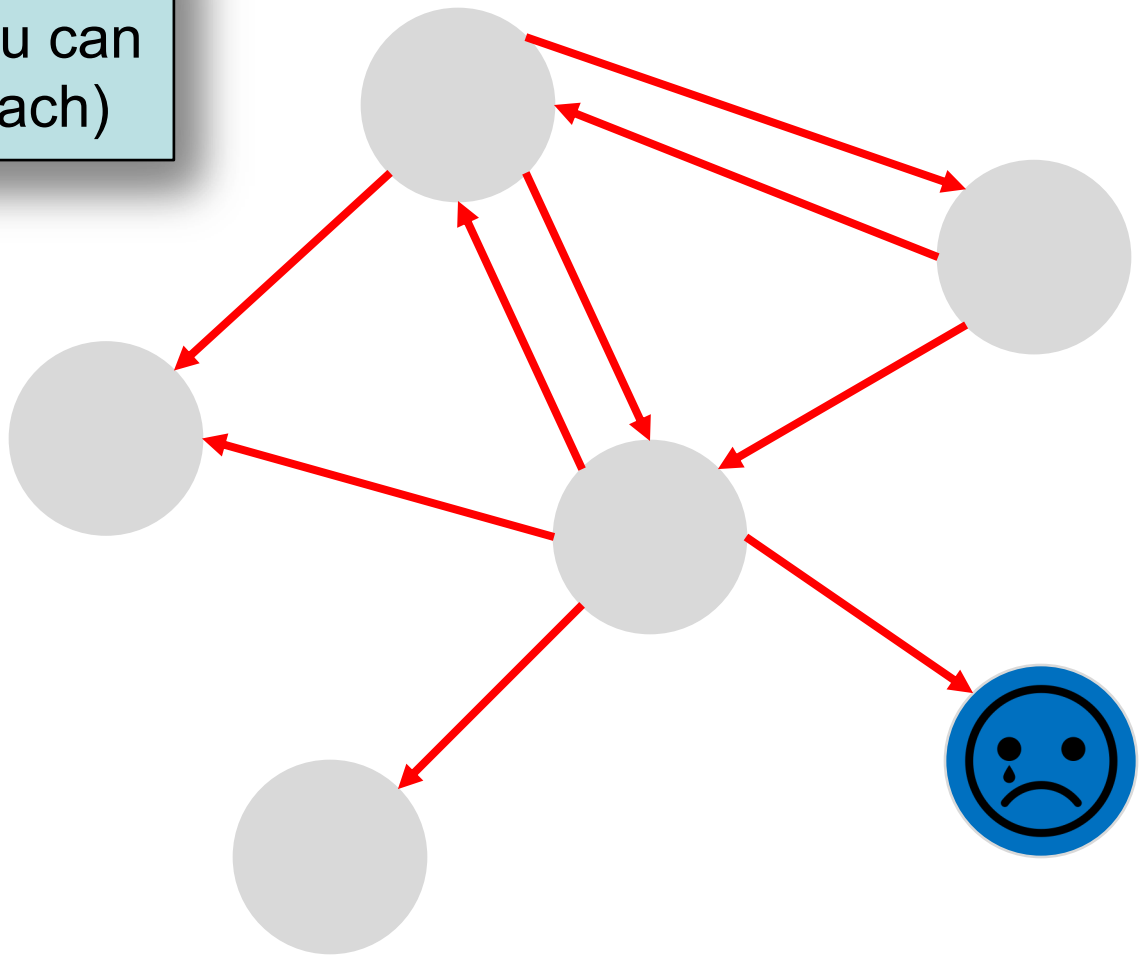
Neighbors

Neighbors (nodes you can directly reach)



Neighbors

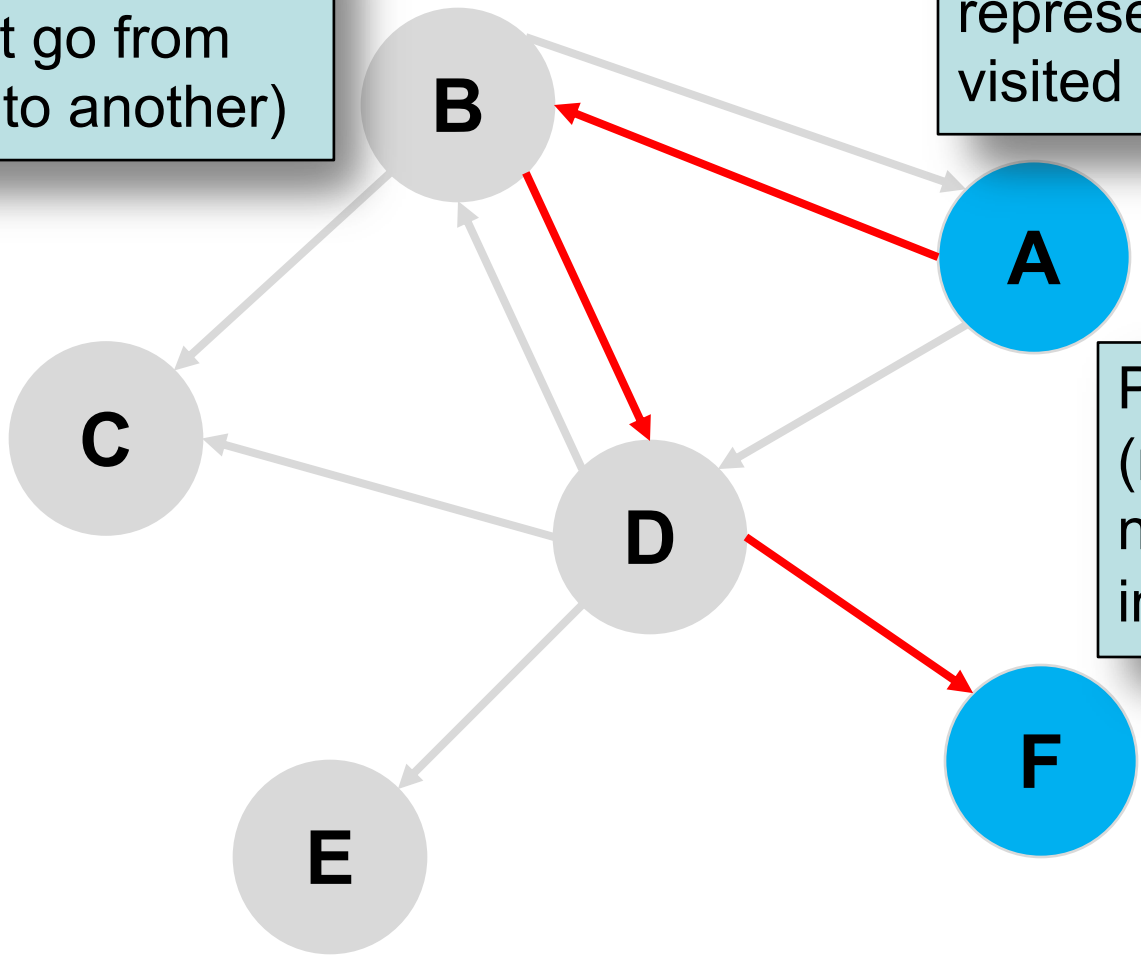
Neighbors (nodes you can directly reach)



Paths

Path (sequence of edges that go from one node to another)

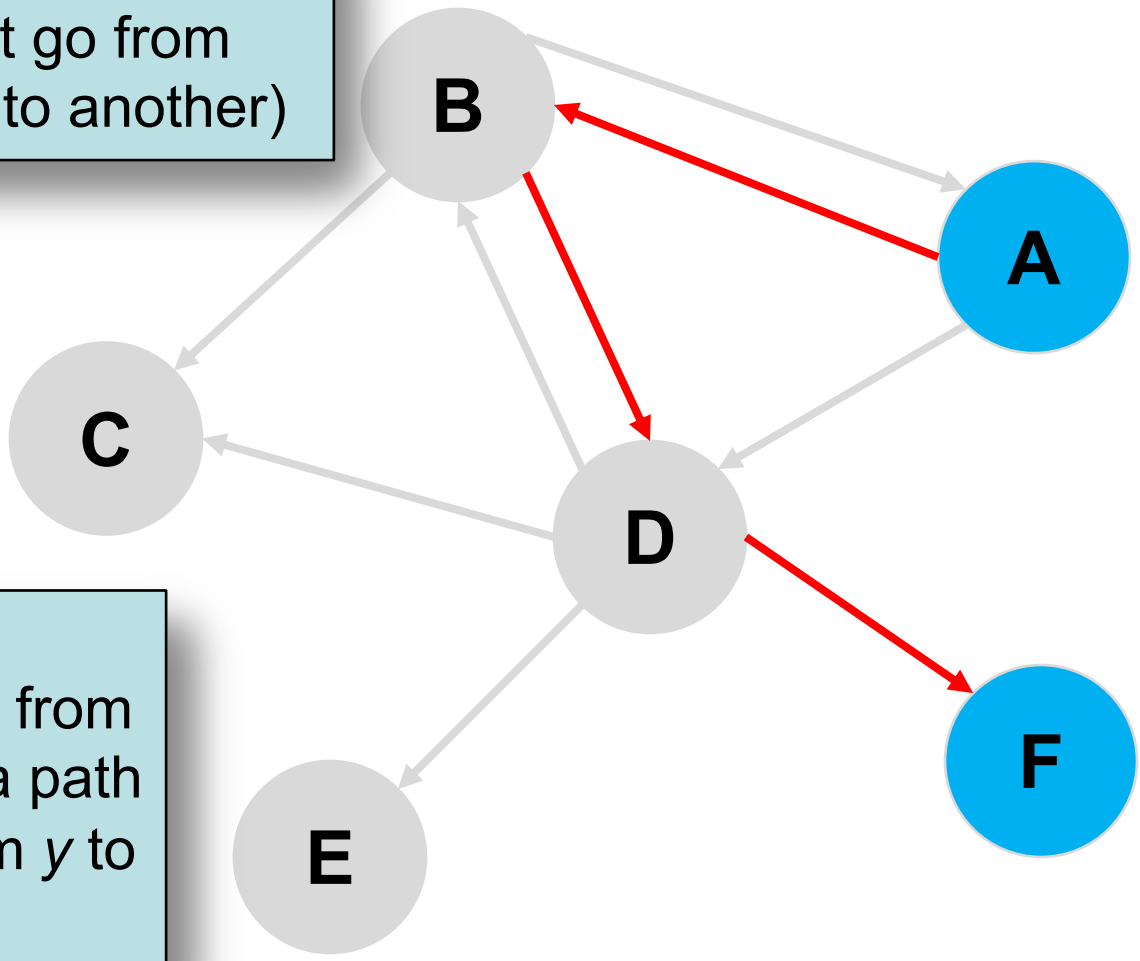
Paths are represented as nodes visited or edges taken



Path length = 3 (number of nodes or edges in the path)

Paths

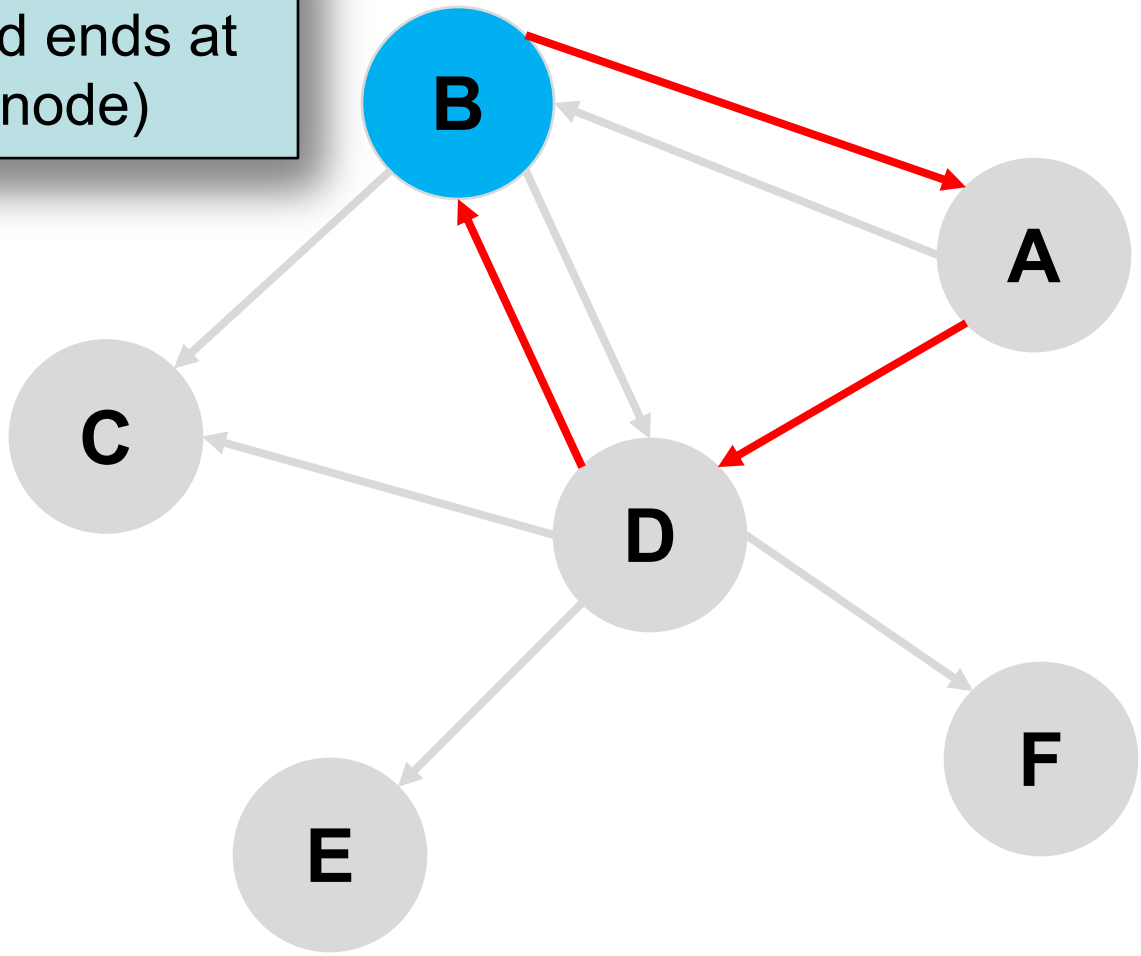
Path (sequence of edges that go from one node to another)



Node *x* is reachable from node *y* if a path exists from *y* to *x*.

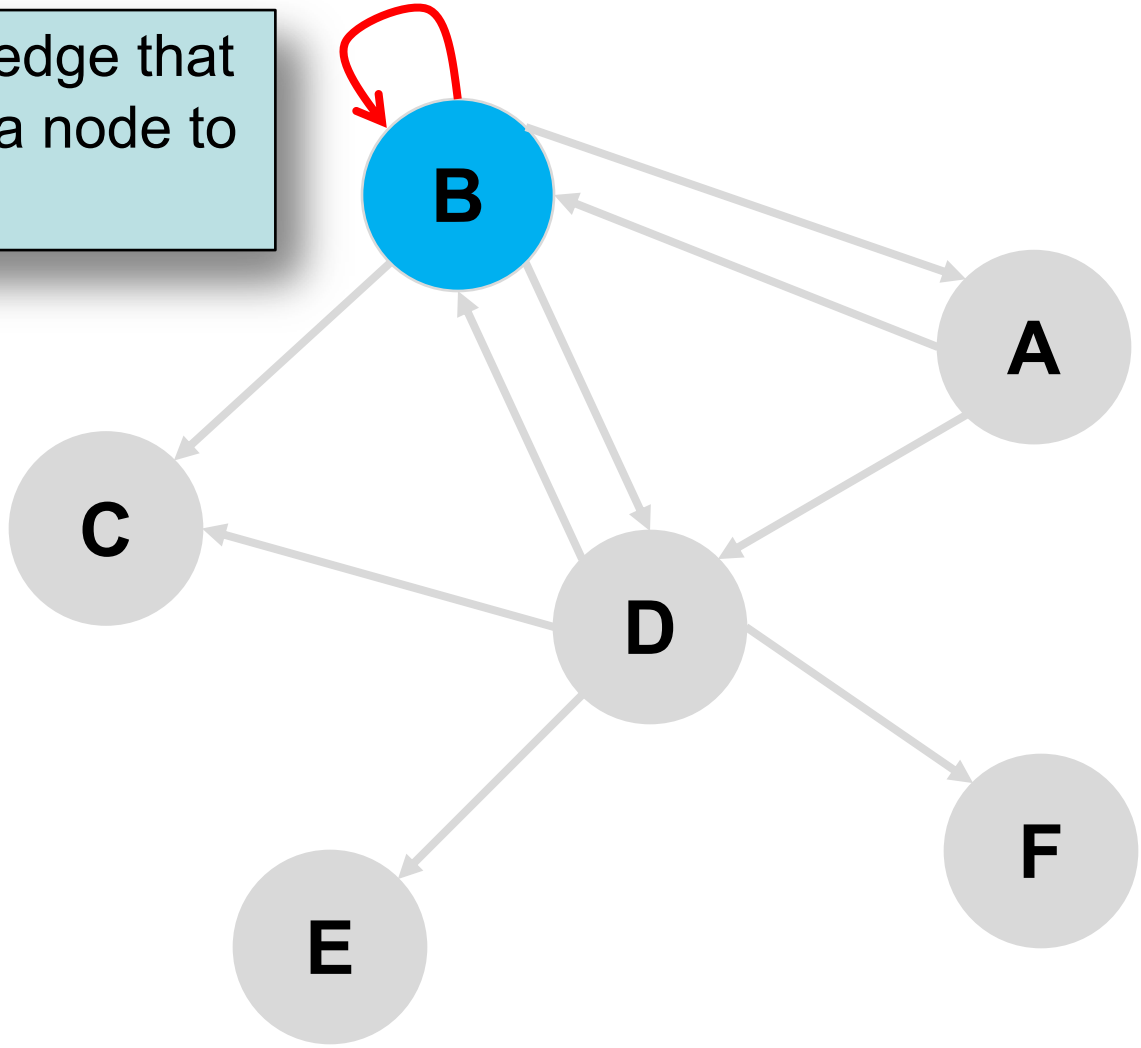
Cycles

Cycle (path that begins and ends at the same node)



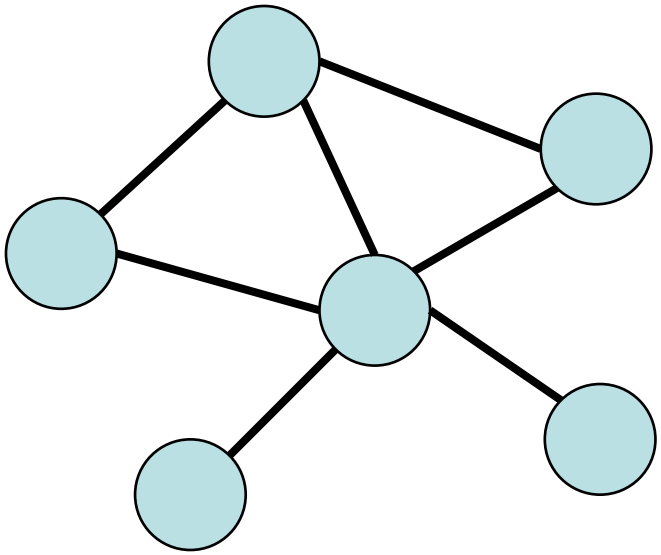
Loops

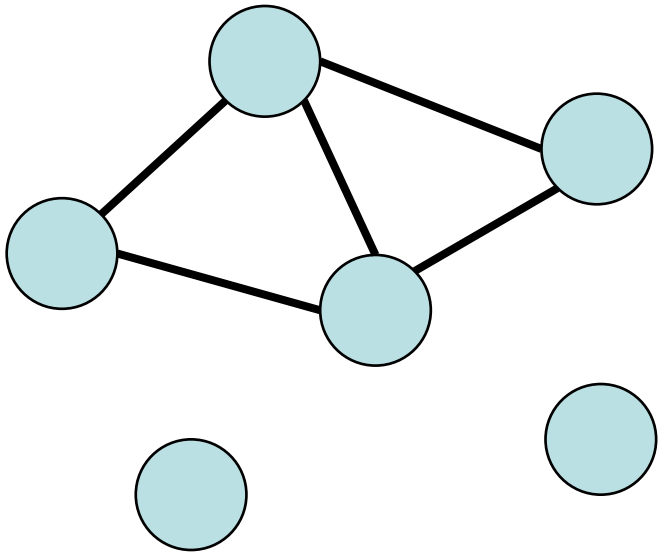
Loop (an edge that connects a node to itself)



Graph Properties

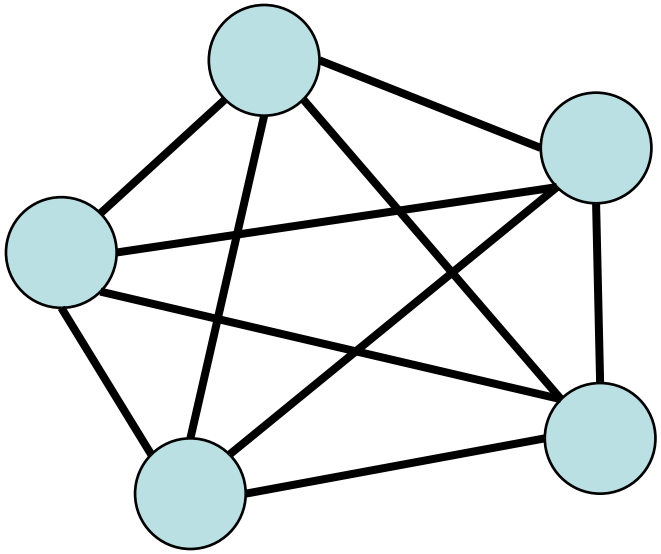
A graph is **connected** if every node is reachable from every other node.

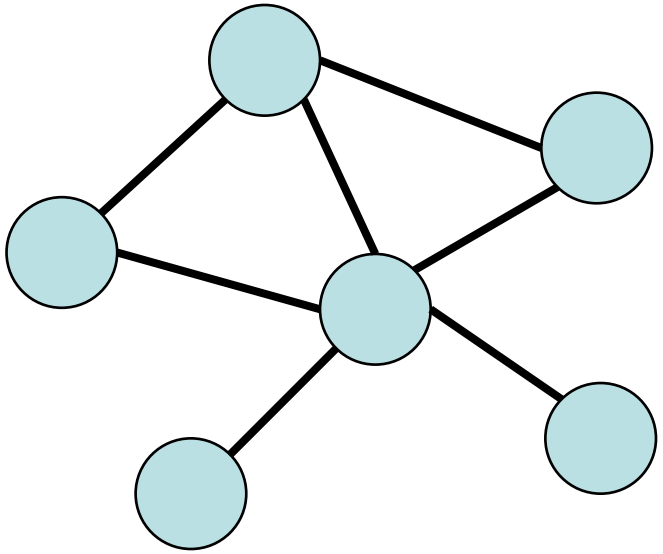




Graph Properties

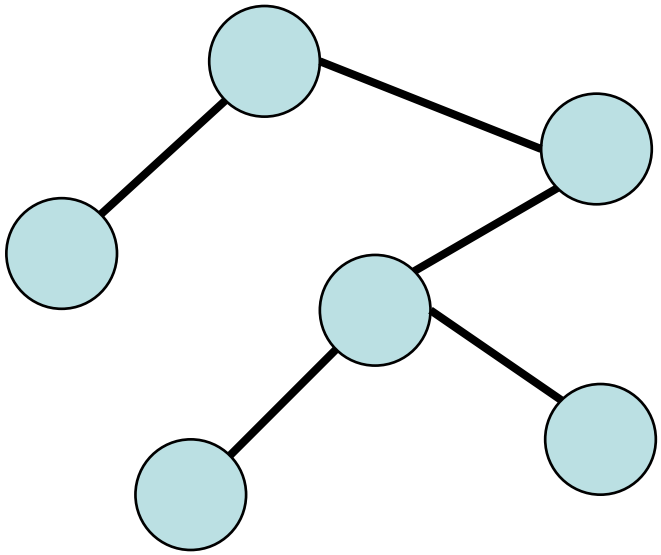
A graph is **complete** if every node has a direct edge to every other node.

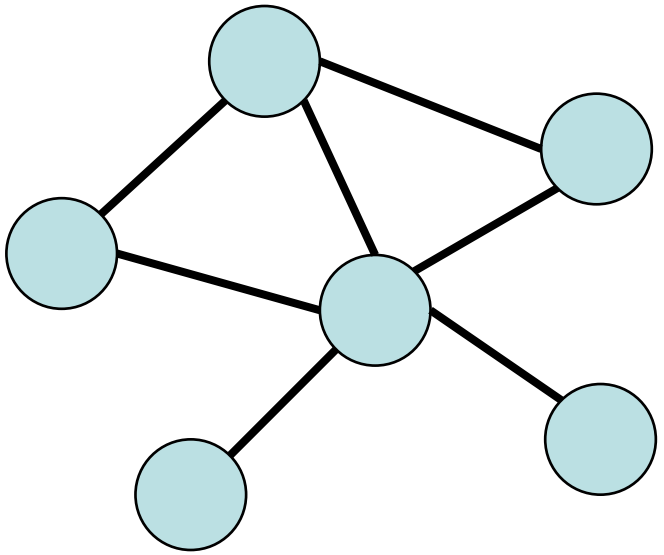




Graph Properties

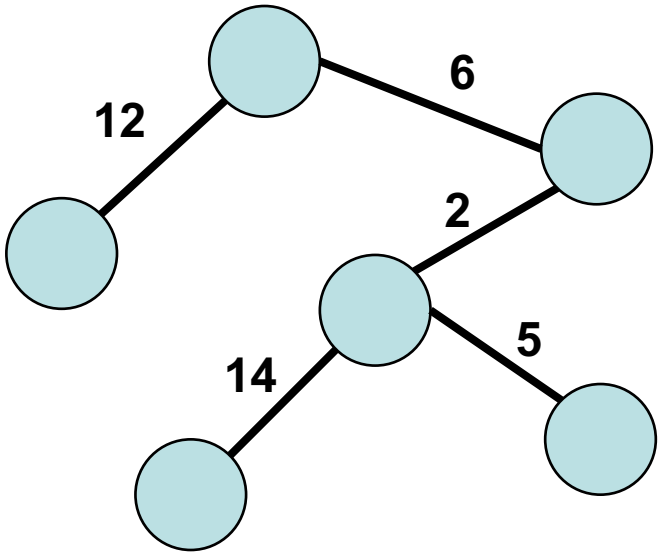
A graph is **acyclic** if it does not contain any cycles.



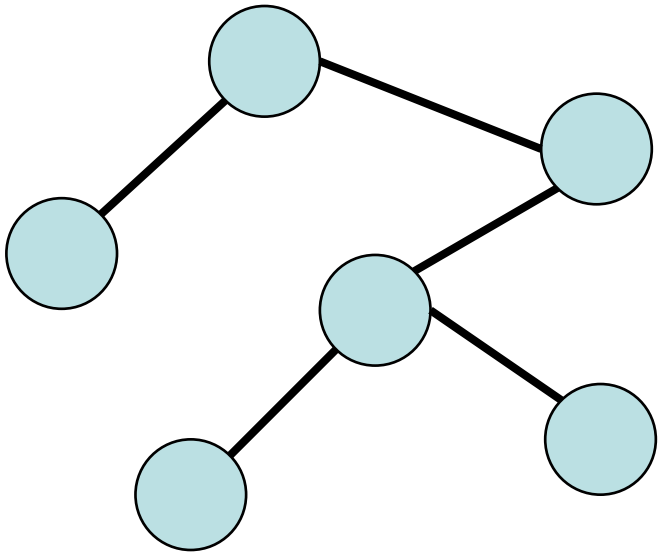


Graph Properties

A graph is **weighted** if its edges have weights, or **unweighted** if its edges do not have weights.



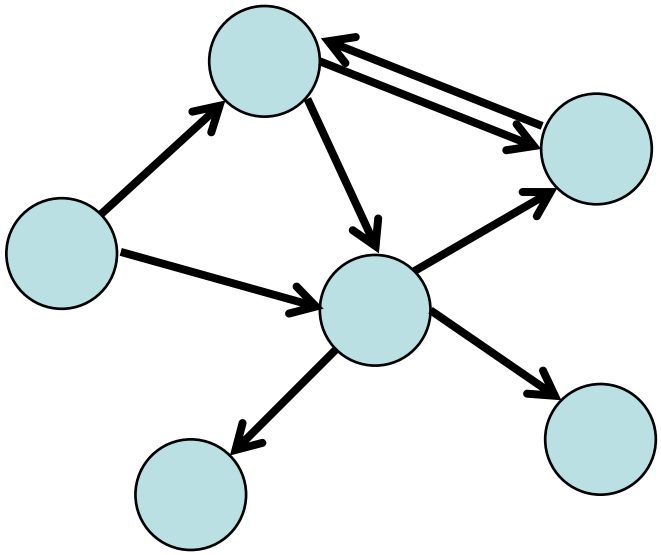
weighted



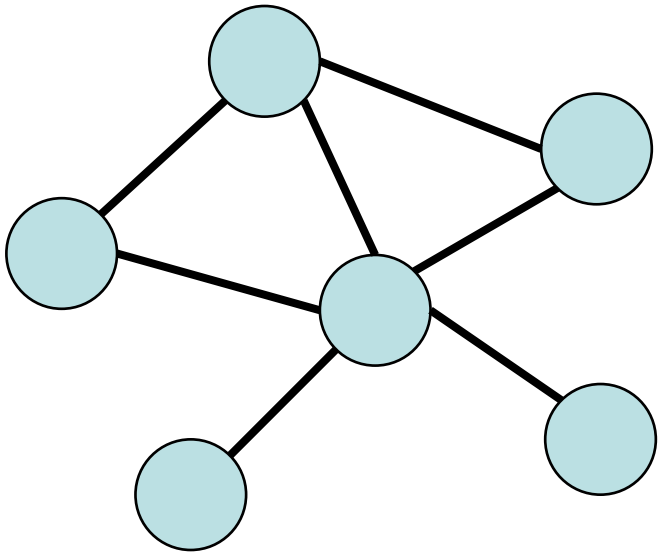
unweighted

Graph Properties

A graph is **directed** if its edges have direction, or **undirected** if its edges do not have direction (aka are bidirectional).



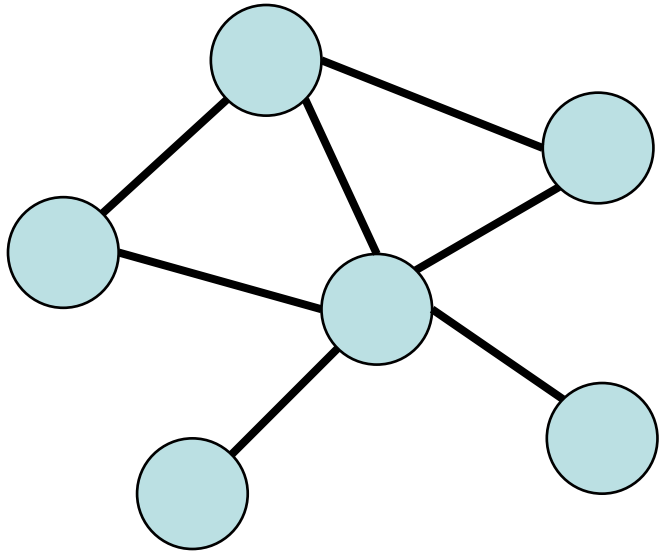
directed



undirected

Graph Properties

- Connected or unconnected
- Acyclic
- Directed or undirected
- Weighted or unweighted
- Complete

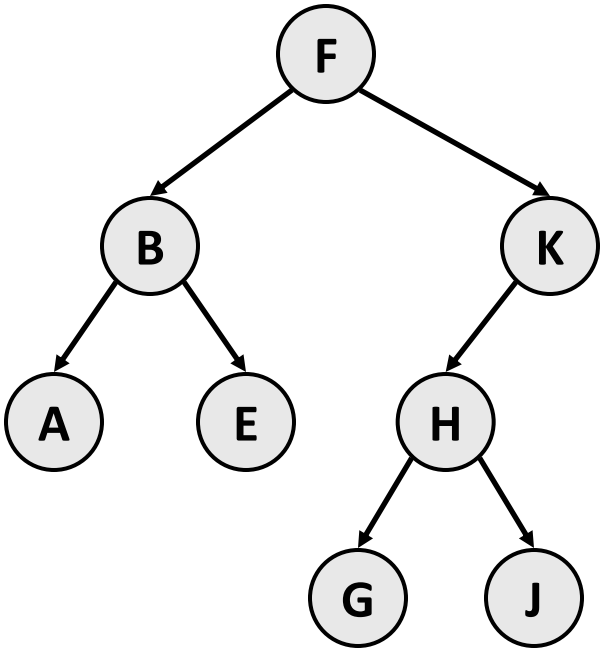


Which of these properties do **binary trees** have?

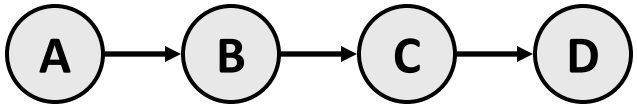
What about **linked lists?**

One Big, Happy Family

- A *binary tree* is a graph with some restrictions:
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.

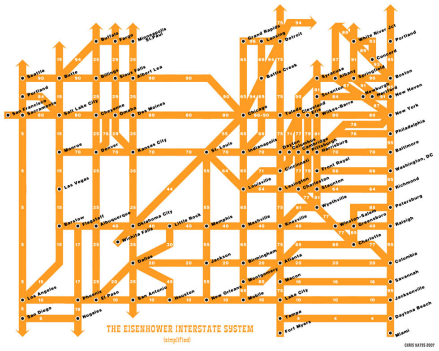


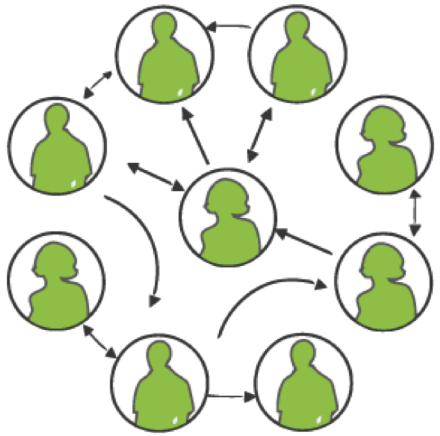
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.

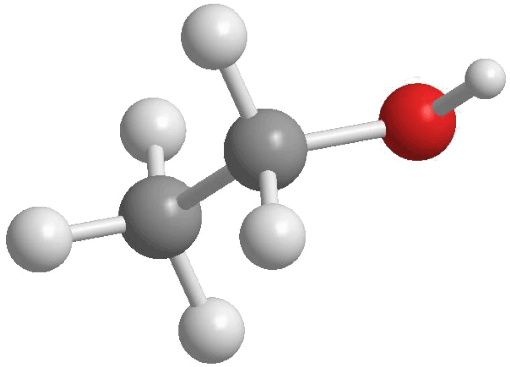


Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Functions in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze
  - Chemical bonds





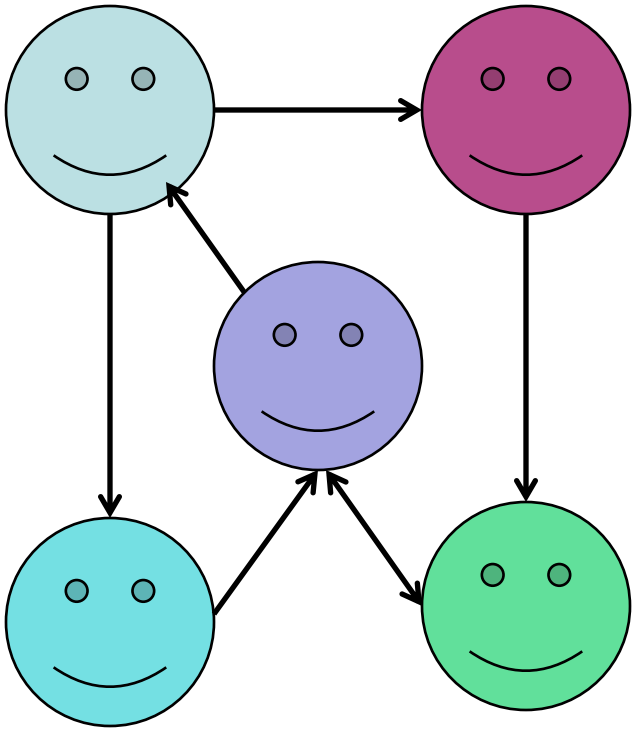


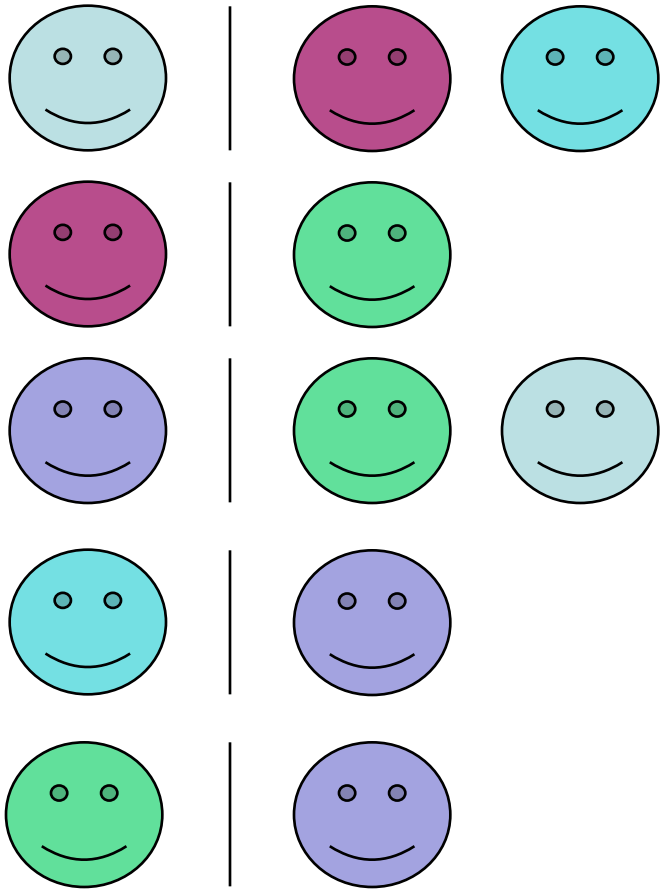
Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

Adjacency List

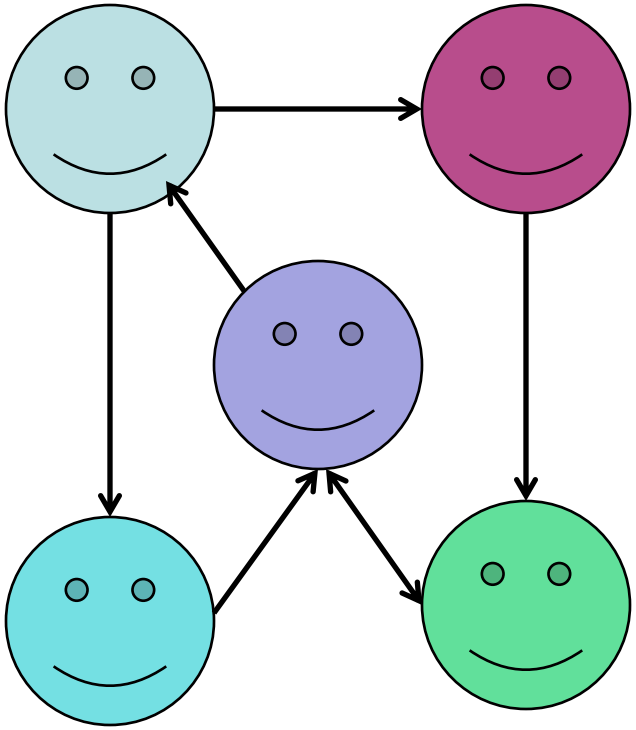
- Map<***Node***, Vector<***Node***>>
  - or Map<***Node***, Set<***Node***>>

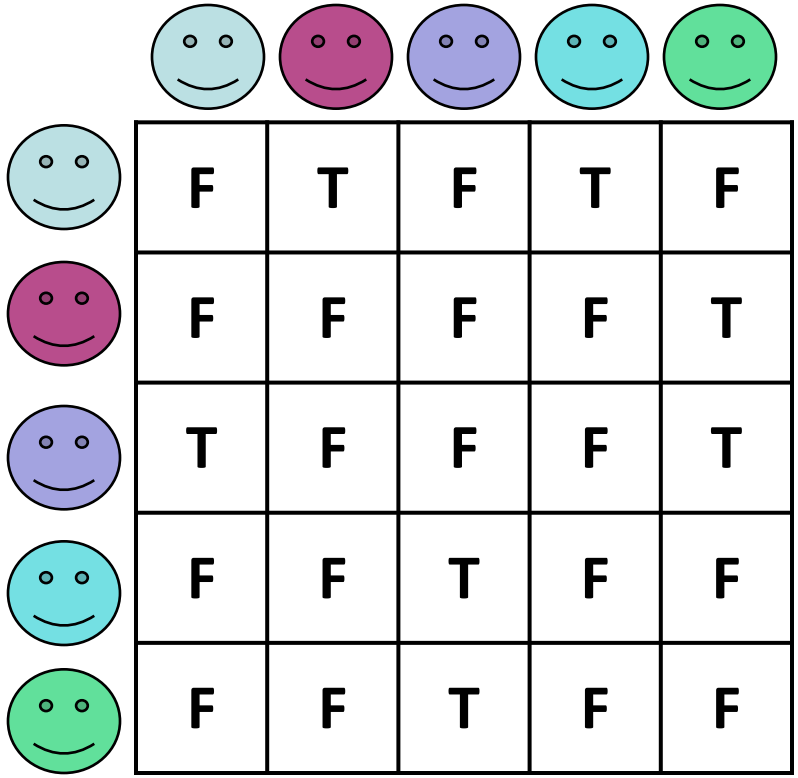




Adjacency Matrix

- Store a boolean grid, rows/columns correspond to nodes
  - Alternative to Adjacency List

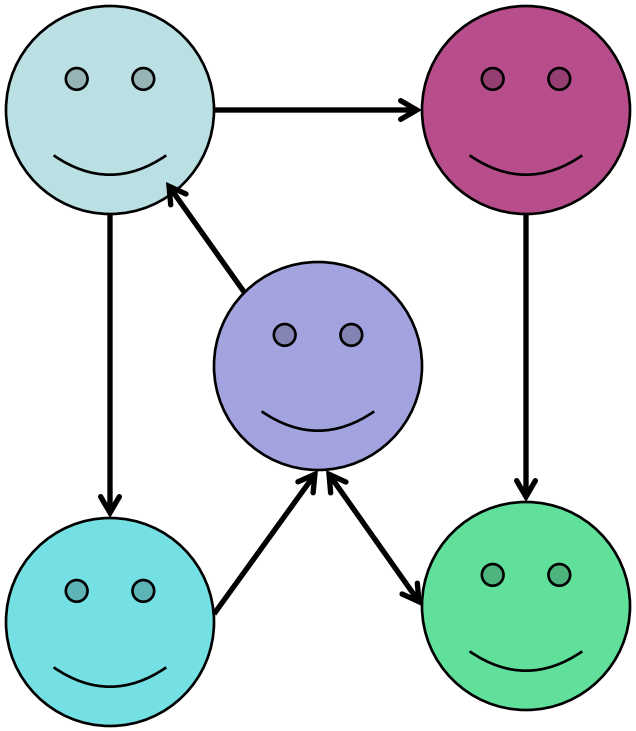




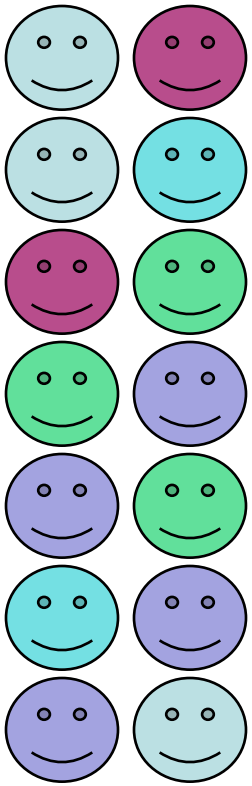
F	T	F	T	F
F	F	F	F	T
T	F	F	F	T
F	F	T	F	F
F	F	T	F	F

Edge List

- Store a `Vector<`***Edge***`>` (or `Set<`***Edge***`>`)
  - ***Edge*** struct would have the two nodes



`Vector<`***Edge***`>`

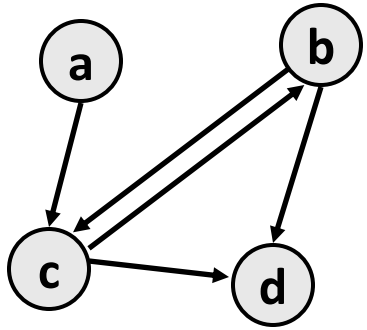


Stanford BasicGraph

- The Stanford C++ library includes a **`BasicGraph`** class representing a weighted, directed graph.
  - Based on an older library class named `Graph`

- You can construct a graph and add vertices/edges:

```
#include "basicgraph.h"
...
BasicGraph graph;
graph.addVertex("a");
graph.addVertex("b");
graph.addVertex("c");
graph.addVertex("d");
graph.addEdge("a", "c");
graph.addEdge("b", "c");
graph.addEdge("c", "b");
graph.addEdge("b", "d");
graph.addEdge("c", "d");
```



BasicGraph members

```
#include "basicgraph.h"    // a directed, weighted graph
```

`g.addEdge(v1, v2);`	adds an edge between two vertexes
`g.addVertex(name);`	adds a vertex to the graph
`g.clear();`	removes all vertexes/edges from the graph
`g.getEdgeSet()` `g.getEdgeSet(v)`	returns all edges, or all edges that start at ***v***, as a `Set` of pointers
`g.getNeighbors(v)`	returns a set of all vertices that ***v*** has an edge to
`g.getVertex(name)`	returns pointer to vertex with the given name
`g.getVertexSet()`	returns a set of all vertexes
`g.isNeighbor(v1, v2)`	returns `true` if there is an edge from vertex ***v1*** to ***v2***
`g.isEmpty()`	returns `true` if queue contains no vertexes/edges
`g.removeEdge(v1, v2);`	removes an edge from the graph
`g.removeVertex(name);`	removes a vertex from the graph
`g.size()`	returns the number of vertexes in the graph
`g.toString()`	returns a string such as `"{a, b, c, a -> b}"`

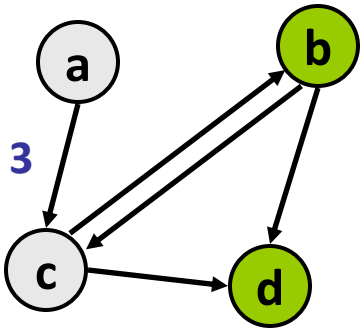
Using BasicGraph

- The graph stores a struct of information about each vertex/edge:

```
struct Vertex {
    string name;
    Set<Edge*> edges;

    ...
};
```

```
struct Edge {
    Vertex* start;
    Vertex* finish;
    double weight;
    ...
};
```



- These are returned as pointers by various functions:

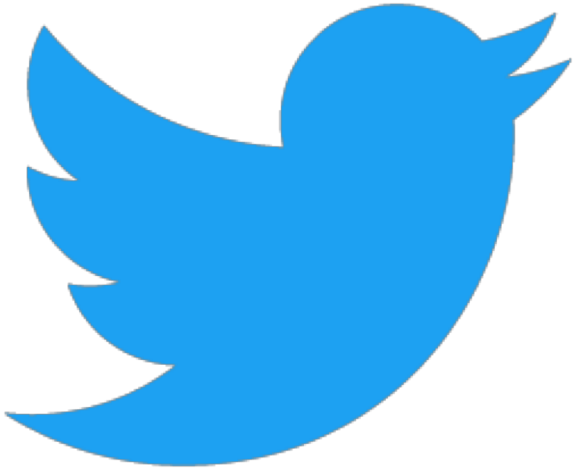
```
// example usage
Vertex* vc = graph.getVertex("c");
for (Vertex* neighbor : graph.getNeighbors(vc)) {
    cout << neighbor->edges << endl;
}

Edge* edgeAC = graph.getEdge("a", "c");
cout << edgeAC->start->name << endl;   // a
cout << edgeAC->end->name   << endl;   // c
```

Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

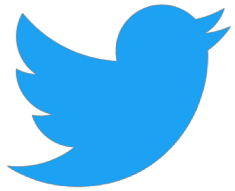
Twitter Influence



https://about.twitter.com/en_us/company/brand-resources.html

Twitter Influence

- Twitter lets a user follow another user to see their posts.
- Following is directional (e.g. I can follow you but you don't have to follow me back ☹)
- Let's define being *influential* as having a high number of *followers-of-followers*.
  - Reasoning: doesn't just matter how many people follow you, but whether the people who follow you reach a large audience.



- Write a function **`mostInfluential`** that reads a file of Twitter relationships and outputs the most influential user.

https://about.twitter.com/en_us/company/brand-resources.html

Recap

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs

Next time: Graphs and Graph Algorithms