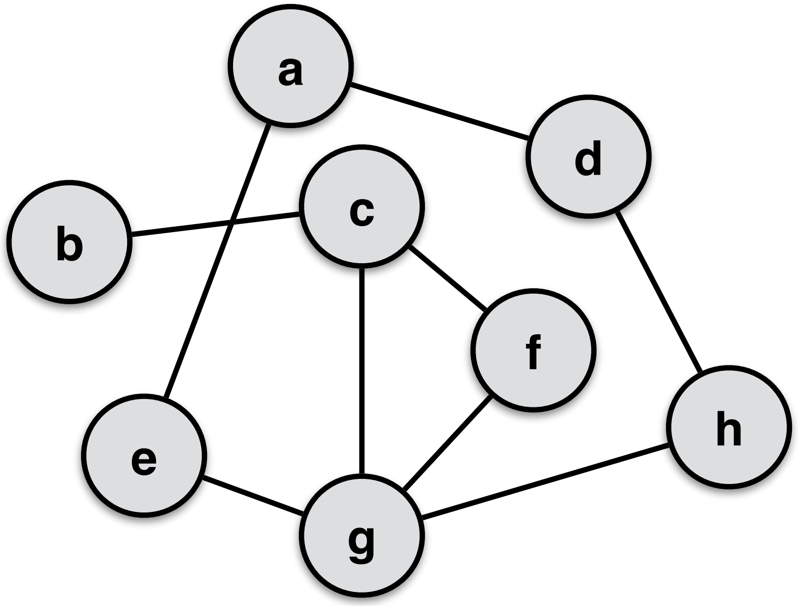
Relationships — Graphs

Introduction: Graphs

- A **Graph** represents a set of relationships among “things.”
- **Vertices** represent the “things”
- **Edges** represent connections/relationships between “things.”



Introduction: Graphs

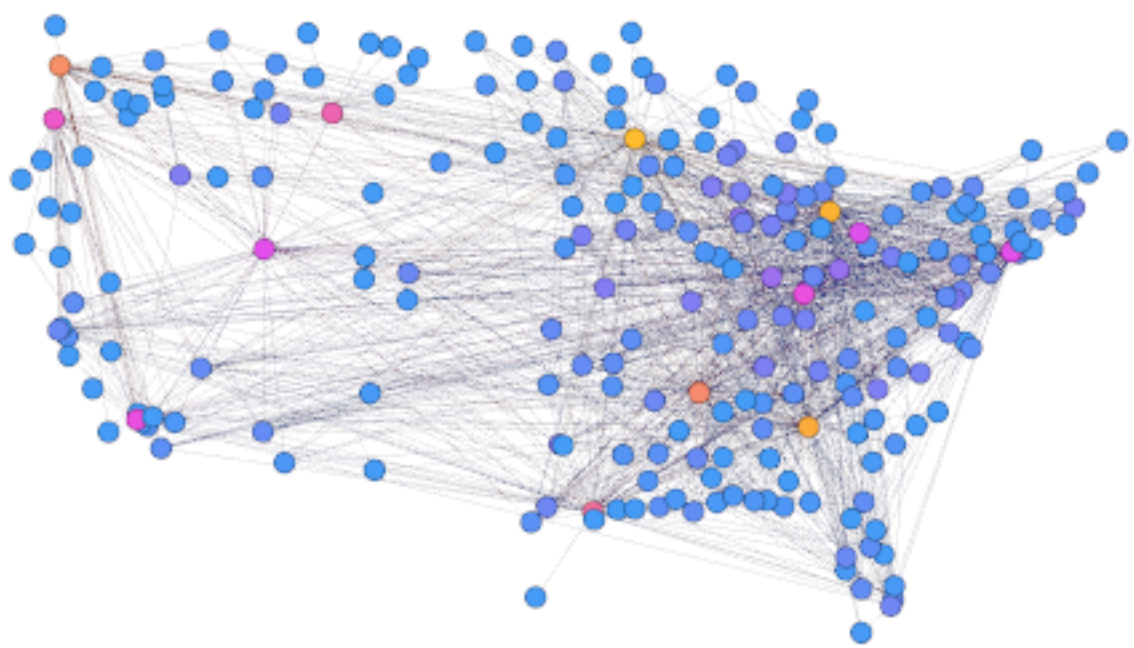
- A **Graph** represents a set of relationships among “things.”
- **Vertices** represent the “things”
- **Edges** represent connections/relationships between “things.”
- The Graph ADT is generally applicable for *many* applications!

ex. road maps (intersections/roads)

Introduction: Graphs

- A **Graph** represents a set of relationships among “things.”
- **Vertices** represent the “things”
- **Edges** represent connections/relationships between “things.”
- The Graph ADT is generally applicable for *many* applications!

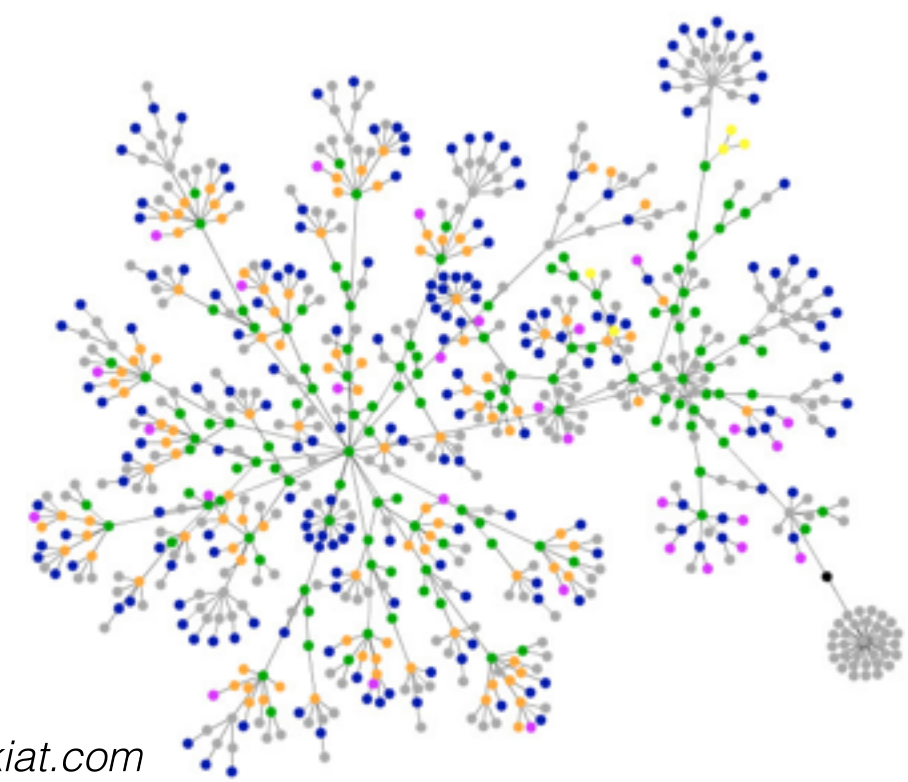
ex. airline routes (airports/flights between them)



Introduction: Graphs

- A **Graph** represents a set of relationships among “things.”
- **Vertices** represent the “things”
- **Edges** represent connections/relationships between “things.”
- The Graph ADT is generally applicable for *many* applications!

ex. the web (websites/hyperlinks)



hongkiat.com

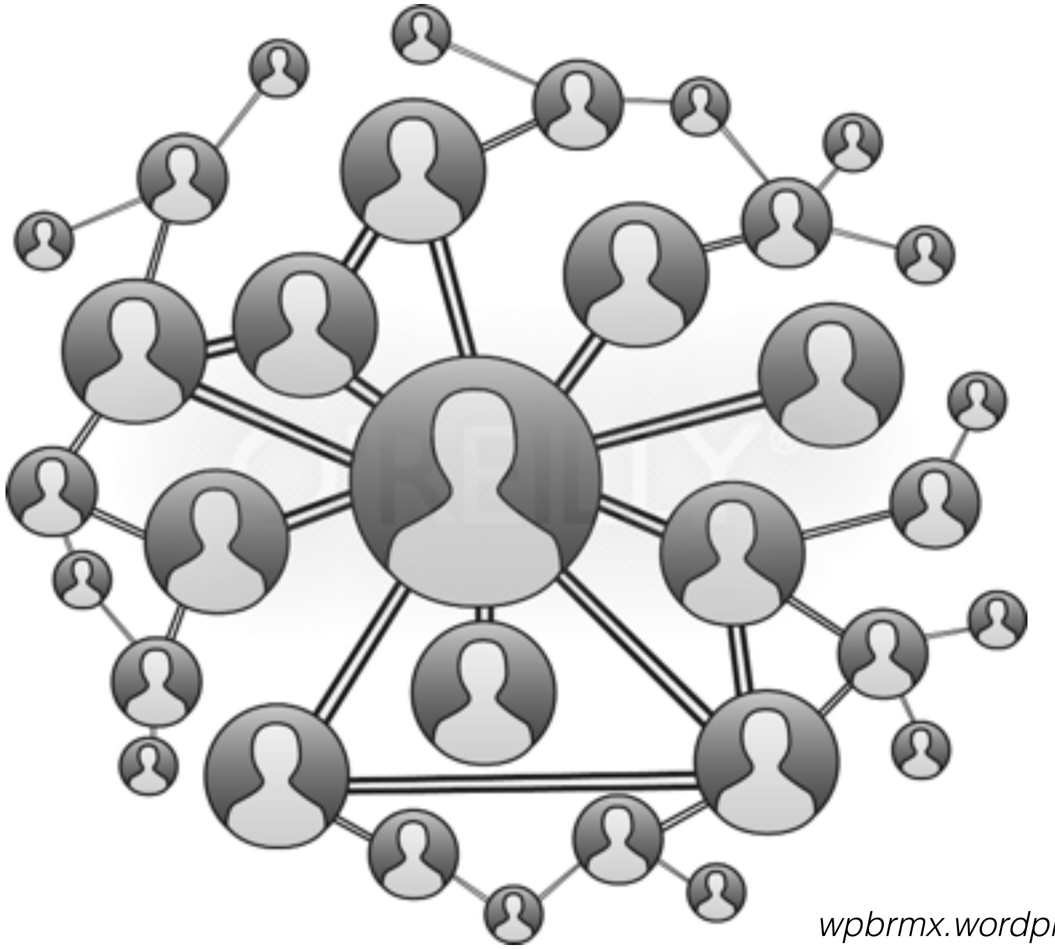
ex. networks (computers/connections)

myignou.in

Introduction: Graphs

- A **Graph** represents a set of relationships among “things.”
- **Vertices** represent the “things”
- **Edges** represent connections/relationships between “things.”
- The Graph ADT is generally applicable for *many* applications!

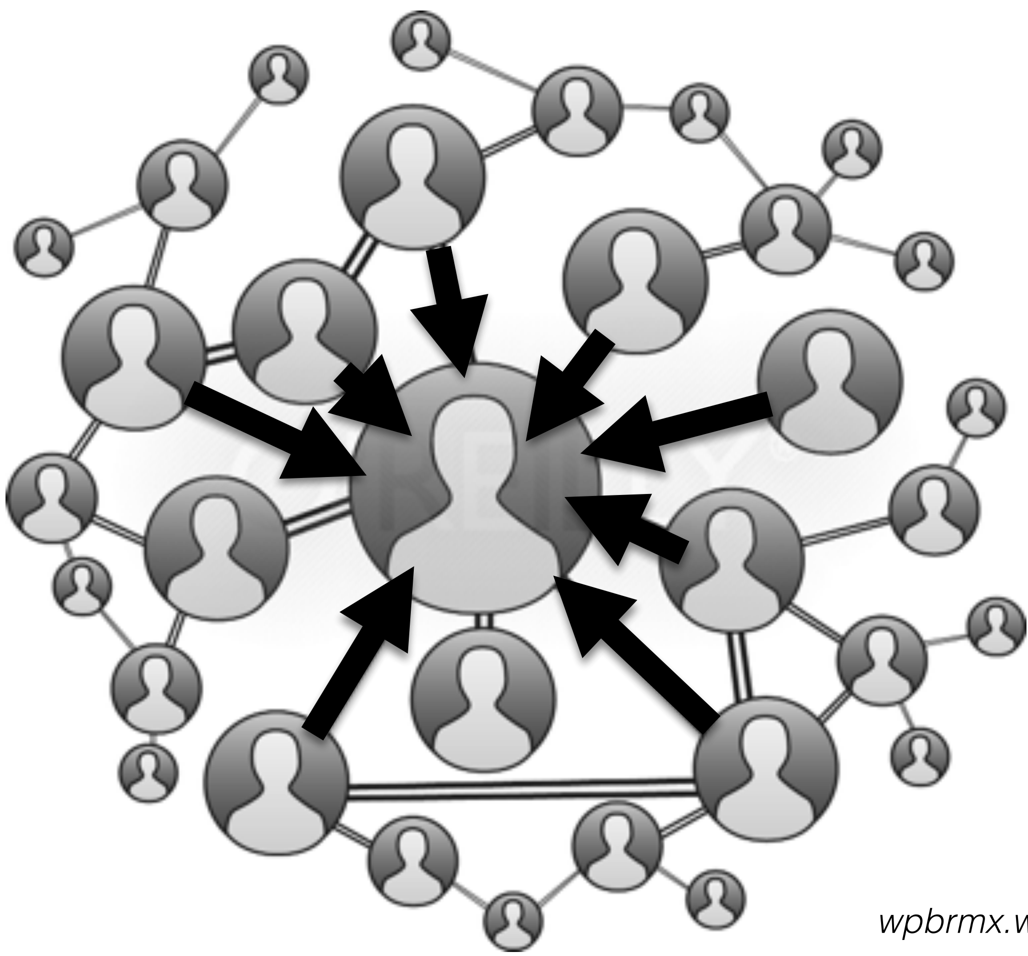
ex. social networks — Facebook(people/“friends”)



wpbrmx.wordpress.com

Introduction: Graphs

- A **Graph** represents a set of relationships among "things."
- **Vertices** represent the "things"
- **Edges** represent connections/relationships between "things."
- The Graph ADT is generally applicable for *many* applications!



wpbrmx.wordpress.com

ex. social networks: Twitter — (people/"followers")*

"followers" brings up the notion of a ***directed*** *edge (connection) since person A may "follow" person B, but person B may not follow person A.*

More Terminology

adjacent — describes 2 vertices connected by an edge

degree — # of incident edges

in-degree — # of incoming edges

out-degree — # of outgoing edges

Introduction: Graphs

- A note about graph types: **symmetric** and **asymmetric**.
- Symmetric
  - Facebook — If A is B's friend, B is A's friend too.
- Asymmetric
  - Twitter — If A follows B, it does not imply that B follows A.
  - Unrequited Love — another unfortunate example of asymmetric relationships… :(
- Symmetric graphs —> **undirected graphs**
  - "An edge is between A and B."
- Asymmetric graphs —> **directed graphs** (a.k.a. ***digraphs***)
  - "An edge is *from* A *to* B."
- There are also **mixed graphs** — i.e., a graph with directed and undirected edges.
  - ex. Streets. Some roads are one-way, while others are not.
- We will primarily talk about undirected graphs today. We will get into directed graphs in SA9 and next class.

Graph Code: Getting Setup…

- The online notes link to the net-datastructures JAR that you need.
- The online notes also discuss how you configure the JAR and how to use it (very similar to how we had to setup the JavaCV JARs).
- NOTE: this configuration needs to be done to *each* project that uses the graph code.
- AdjacencyListGraphMap.java is the only source code *not* included in the JAR.
- Graph.java | Position.java | AdjacencyListGraph.java — all included in the JAR (just **import net.datastructures.*;** to use). They've been included from the lecture notes so that you can see the files we need.

Graph Code: Graph.java

Some Notes:

- The book (pg. 618) defines the ADT seen in Graph.java, though there are some differences since the book's version addresses *directed graphs*.
- **Graph<V, E>** is *generic* in terms of the objects that serve as *Vertices* & *Edges*.
- **Vertex<V>** and **Edge<E>** are interfaces (defined in the JAR) that package up these generic types — these go in the graph.
- *[Run through Graph.java interface in class]*

- (!) Graph ADT is missing… retrieving a vertex that has a particular element value!
  - We can iterate through a collection of all vertices and test them (we show sample code for this in code that we will see soon).

Graph Code: Graph.java

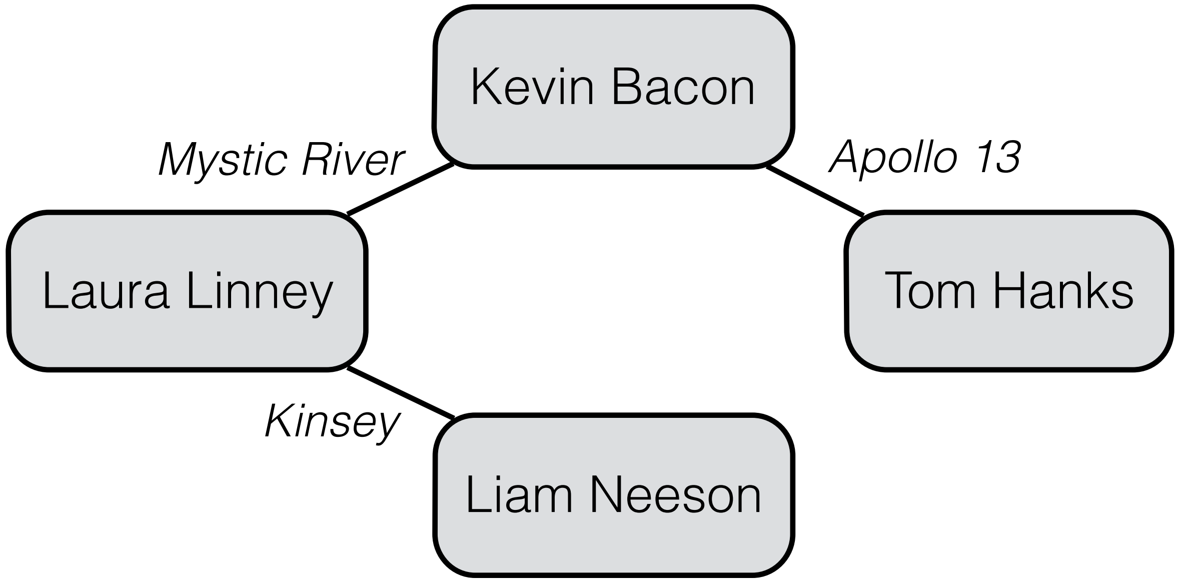
Some Notes:

- Graphs are cool because we can do stuff with them!
- Stats we could compute…
    - Find most popular (most edges).
    - Find mutual acquaintances (“cliques” where all vertices have edges to each other).
    - etc.
- We will look at some of these in upcoming lectures and lab 5!

Graph Code: AdjacencyListGraphMap.java

Some Notes:

- Our undirected graph implementation: **AdjacencyListGraphMap<V, E>**
- In this code we build and play with a “move-costar graph”
  - vertices == actors,
  - edges == movies

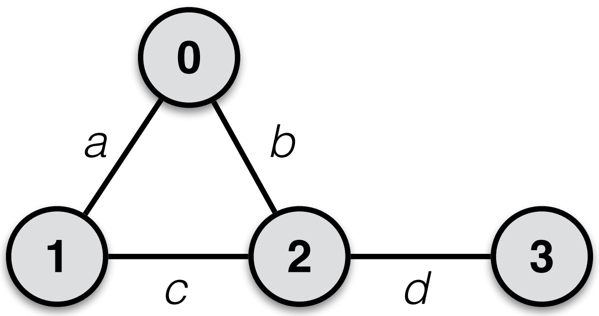


- *[Run through AdjacencyListGraphMap.java in class]*
  - *Use main() to guide discussion since there is A LOT going on in this code…*
  - *Talk about details later — just demo the basic interface.*

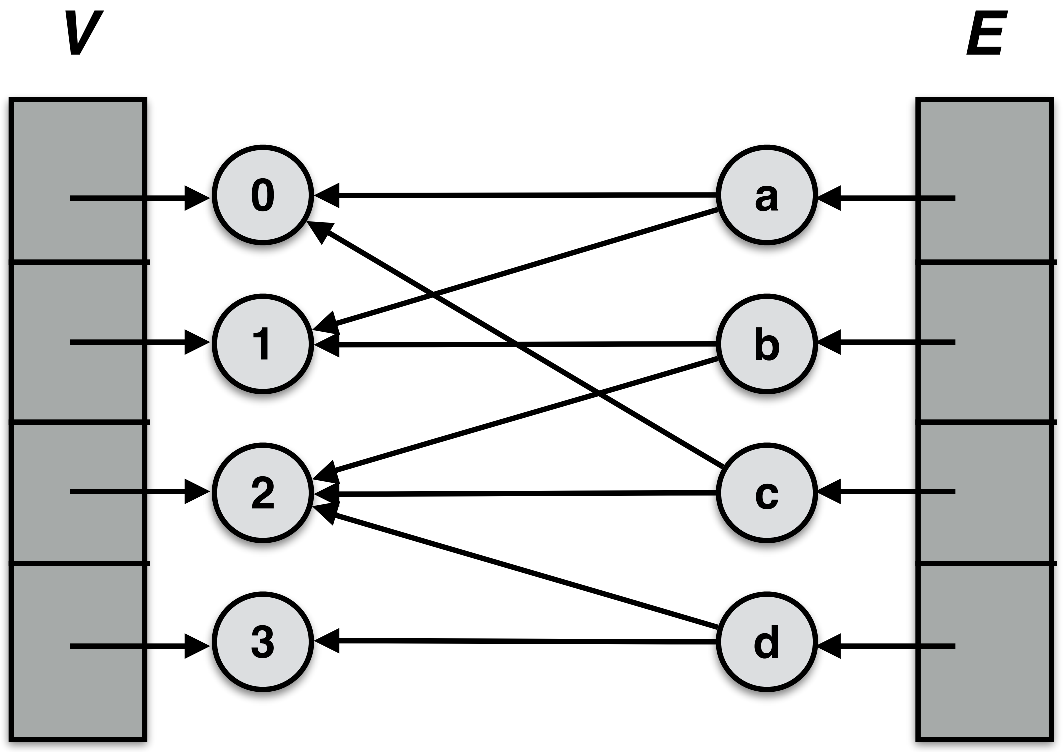
Representation

Q: What data structures can we use to represent a graph?!

Representation: Edge List

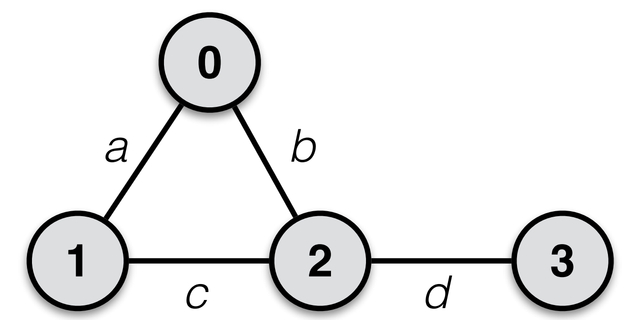


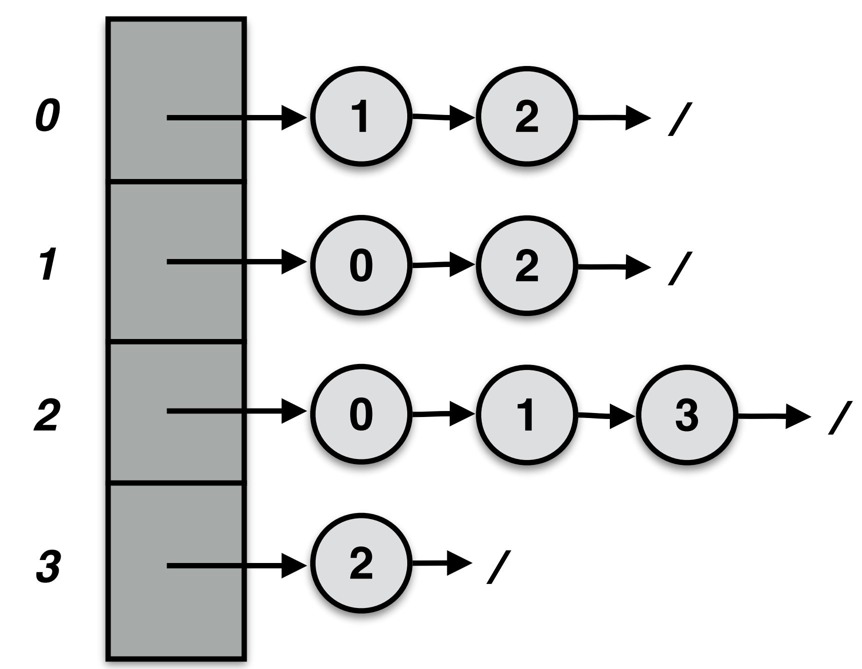
- **Simplest:** Edge List
- List of Vertex objects (V)
- List of Edge objects (E)
- Edge objects keep track of its Vertex end points; Vertex objects don't refer to incident edges.
- Typically, these lists are linked lists (for easy insertion/deletion).
- Operations (*n* vertices, *m* edges)
  - insertVertex(), insertEdge(), and removeEdge() — O(1)
  - replace() — O(1)
  - vertices() and edges() — O(n) and O(m)
  - endVertices() and opposite() — O(1)
  - incidentEdges() and areAdjacent() — O(m)
  - removeVertex — O(m)
  - Also note storage: O(n + m) space.
- Some operations are slow because of our edge list structure…



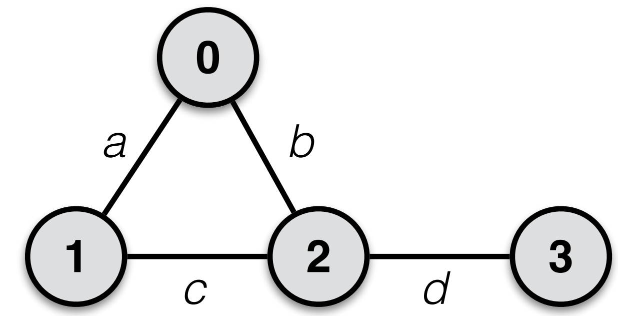
Representation: Adjacency List

- **Improvement:** enable quicker access *from* vertex *to* incident edges (vertices)!
- Traditionally, incident edges stored in a linked list (though any "collection" could work).
- Aptly named **Adjacency List** — list of adjacent edges (vertices).
- (!) Sometimes you'll see edges, sometimes you'll see vertices… vertices make more sense since a vertex in an adjacency list *by definition* has an edge between it and current vertex. Silly computer scientists…
- Operations (*n* vertices, *m* edges)
  - incidentEdges()/incidentVertices() — O(1)
  - traversing A.L. for *v* takes O(*degree(v)*).
  - removeVertex — O(*degree(v)*).
  - areAdjacent(v,u) — O(min(*degree(v), degree(u)*)).
- Same…
  - insertVertex(), insertEdge(), and removeEdge() — O(1)
  - replace() — O(1)
  - vertices() and edges() — O(n) and O(m)
  - endVertices() and opposite() — O(1)
  - Also note storage: *still* O(n + m) space.
    - note: 2m (undirected) vs. m (directed)





Representation: Adjacency Matrix



- The only method that takes longer than needed is areAdjacent(u,v). To speed this up we make an **Adjacency Matrix** (combined with the edge list structure).
- Vertices numbered 0 to n-1. Create *n x n* matrix. A[i][j] = ref. to edge between vertex i and j.
- Undirected graphs are symmetric: A[i][j] == A[j][i]; not necessarily true for digraphs.
- <u>Operations</u> (*n* vertices, *m* edges)
  - areAdjacent(v,u) — Θ(1).
  - incidentVertices() — Θ(n) since we have to look through entire row or column.
  - insert/delete — Θ(n^2) since we have to rebuild the matrix.
- The more usual definition of the A.M. has *only* the matrix. The vertices are numbered and the entries are boolean simply to indicate if an edge exists or not.
- If you want edge values, you could store references (or null) in matrix entries.
- Enumerating all edges takes Θ(n^2) — have to look at *all* entries.
- Also, take Θ(n^2) space even for ***sparse*** graphs (i.e., few edges).

	0	***1***	***2***	***3***
0	*0*	*1*	*1*	*0*
1	*1*	*0*	*1*	*0*
2	*1*	*1*	*0*	*1*
3	*0*	*0*	*1*	*0*

Implementation

- The book's implementation is a bit different than what we are used to…
- For LL we've used:
    - Element object as an inner class (not accessible from outside the class).
    - get/set are possible given an index into the list.
    - can use iterator to iterate over list.
    - No one ever needs to know about the Element class…
- In GTG, they use a **Position** object. The interface specifies a single method, **element()**, which simply returns the data stored in the object.
- Their node classes for Lists, Trees, etc., implement the Position interface.

```
public interface Position<E> {
    E element();
}
```

- In the context of Graphs:
    - **Vertex** and **Edge** are both **Position**s, storing vertex and edge data as their elements, *and* maintain info. about where they are in the graph data structure.

Implementation: AdjacencyListGraph.java

- The book provides an adjacency list implementation in **AdjacencyListGraph.java**.
- It uses LLs to hold edges and vertices.
- It also provides some other methods, including degree(v) and toString() for the entire graph.

[Walk through some code]

- Inner classes **MyVertex** and **MyEdge** store the data and implement the extended **Position** interface.
- Ignore details of **MyPosition** for now; we'll return to it next class.
- The class itself just stores lists of edges and vertices.
- Most methods go through this extra step of casting a position to a vertex or edge as appropriate, and then calling the appropriate method on that thing.
- Note linear search (on smaller degree vertex) for **areAdjacent()**.
- Note maintenance of corresponding structure when remove edge or vertex.

Implementation: Other Notes

- Values in vertices are often names/identifiers (String)
- It is helpful to be able to find a vertex, given its identifier.
    - We will use this in Lab 5 — the Kevin Bacon Game!
- Therefore, we have provided a modified version of **AdjacencyListGraph.java** called **AdjacencyListGraphMap.java**
    - Stores vertices in Map rather than LL.
    - The “key” is the value stored in the element.
    - *Only works if vertices are unique!*
        - Provides getVertex(id) — returns Vertex<V> corresponding to that id
    - Had to update instance variables, constructor, insertVertex(), removeVertex(), and replace() — all operations on Vertex objects.
    - Also overloaded methods that take Vertex<V> params to take V params — this requires a simple one-liner and makes code elsewhere a bit nicer.
    - main() does the Kevin Bacon tests illustrated earlier.