Middlebury

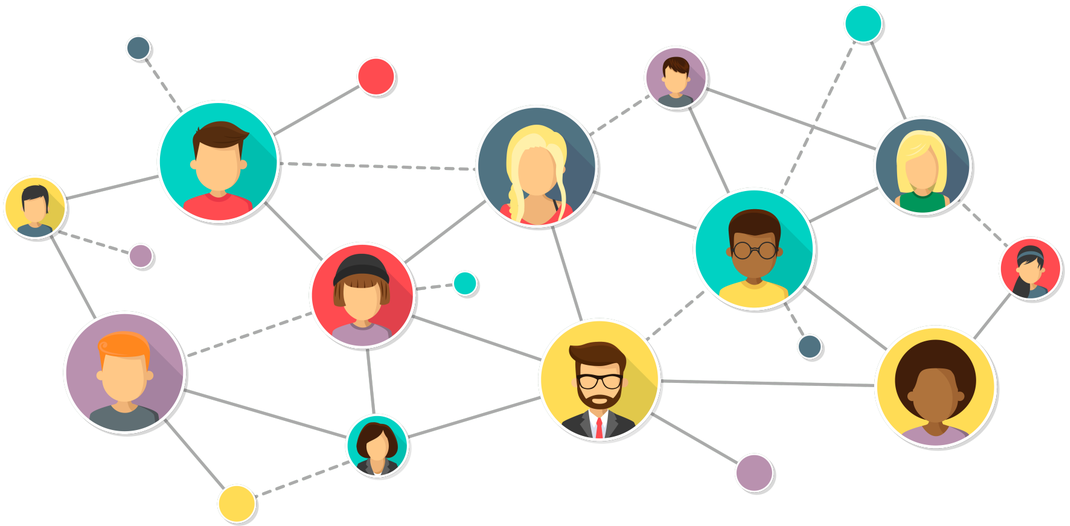
CSCI 201: Data Structures

Spring 2025

Lecture 11M: Graphs

Goals for today:

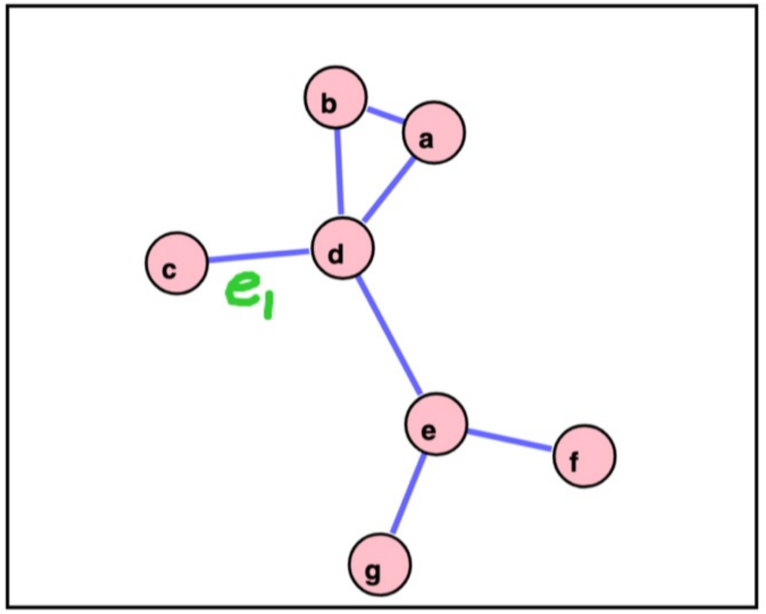
- Learn what is a **graph**, **edge**, **vertex**/**node**, **path**, **directed edge**, **cycle**; learn what **weights** represent; what is the **degree** of a node.
- Different types of graphs: **directed**, **complete**, **connected**.
- Represent a graph using an **adjacency matrix**.
- Represent a graph using **adjacency lists**.
- Represent graphs in Java.



Terminology: a graph consists of *two sets*: nodes (a.k.a. vertices) and edges

edge e1 is incident to nodes c and d

Nodes c and d are adjacent



degree (d) = 4

degree (f) = 1

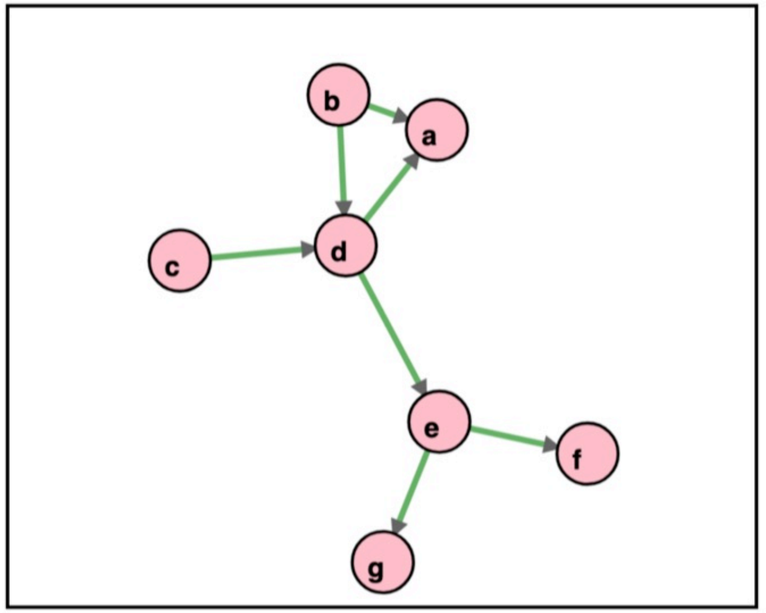
b - d - e - g is a path

- Two vertices are **adjacent** if there is an edge between them. The edge is **incident** to both vertices.
- The **degree** of a node is the number of edges incident to it.
- A **path** is a sequence of nodes $p_1,\ p_2, \ldots, p_k$ where there is an edge $(p_i,\ p_{i+1})$ in the set of edges. No edge is repeated. A *simple path* has no repeated vertices.
- A **cycle** is a path where the first and last node are the same.

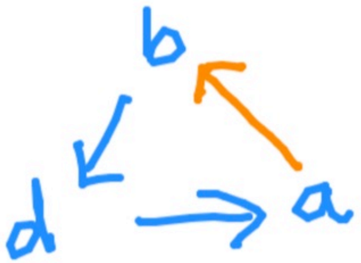
a - b - d - a is a cycle

Graph G = (V, E)
vertex set V, edge set E

Edges can also have a direction

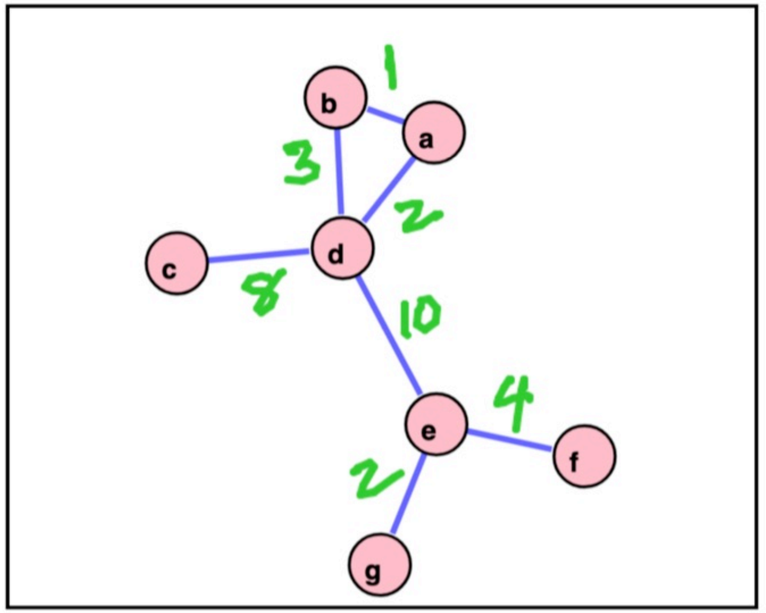


Is there a cycle in this graph? No



This has a cycle

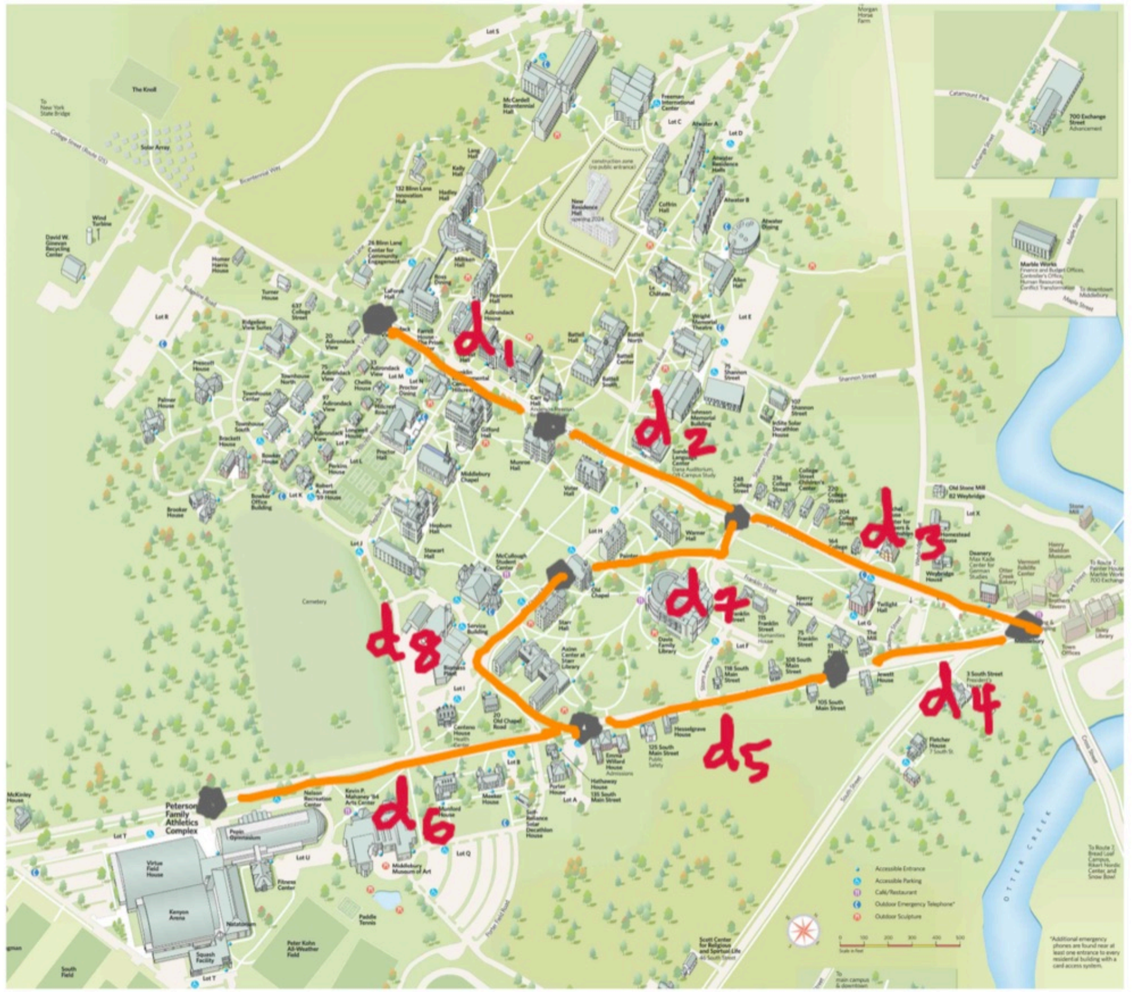
Edges can also have weights



What can weights be used for?

distances between nodes
costs
flow between nodes

We might want to calculate the *shortest path* between two nodes

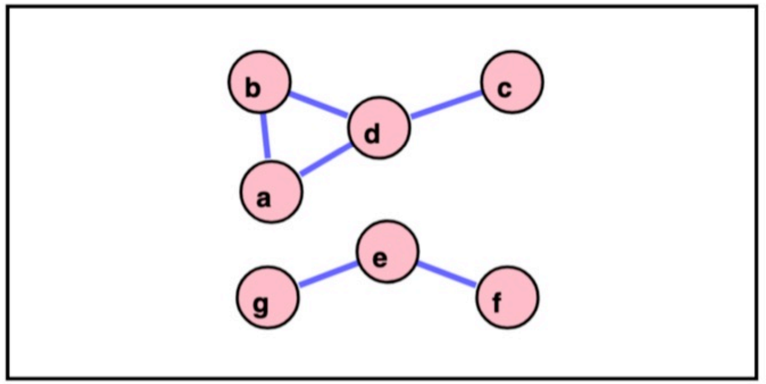


weights = distances (d_i's) between nodes

A graph is *connected* if there is a path between every pair of nodes

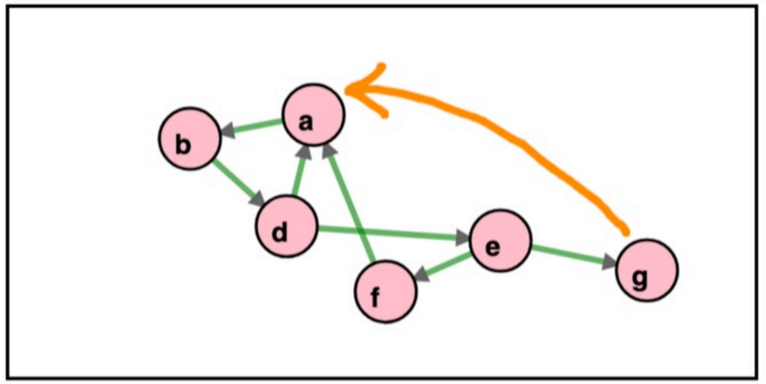
graph

has → 2 connected components



not connected

Strongly connected (for directed graphs): every pair of nodes is reachable by a path



Now strongly connected

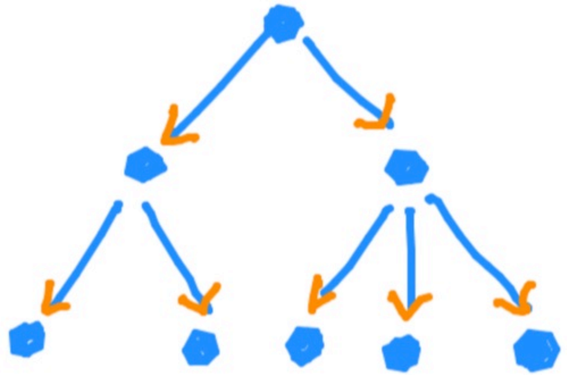
Which edge can we add to make this graph strongly connected?

We have seen graphs before! A tree is a special type of graph

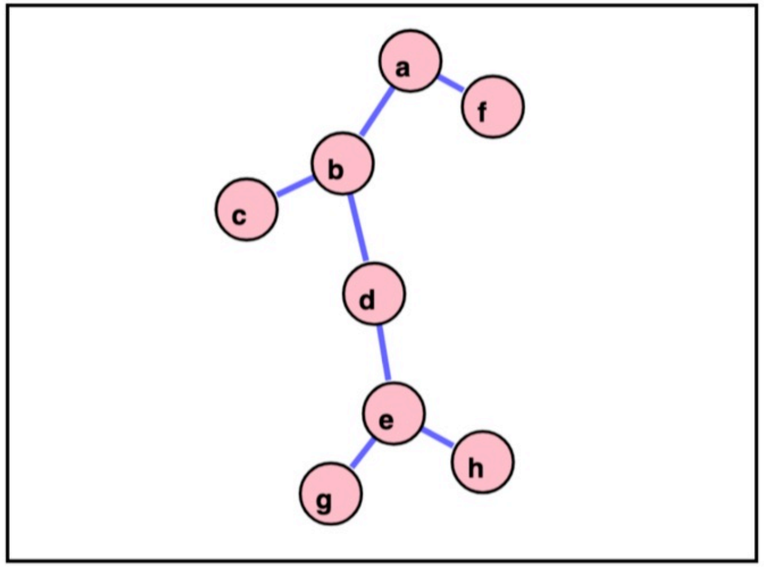
A tree is a **connected**, **undirected** graph without any cycles.

← no cycles ≡ acyclic

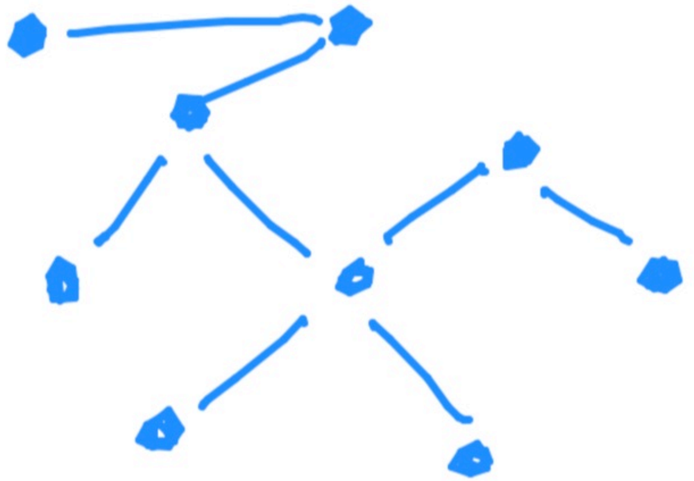
rooted tree



directed edges are implied



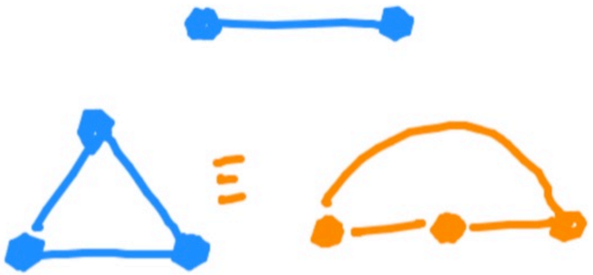
free/unrooted tree

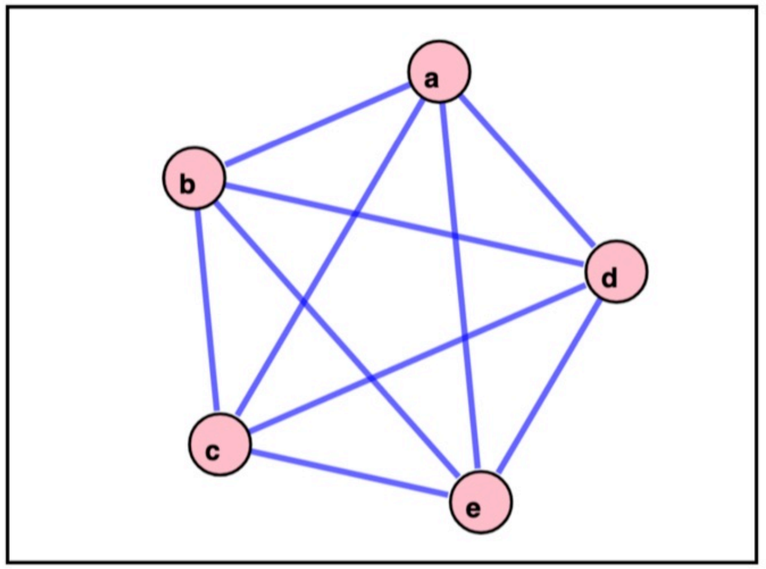


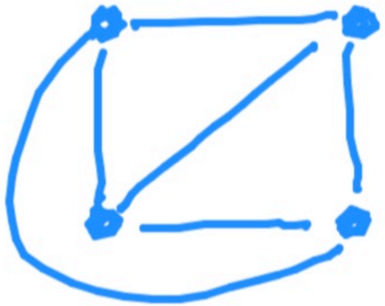
Trees can be rooted or free.

"DAG": directed acyclic graph

Complete graph: every pair of nodes is connected by an edge







planar graph: can be drawn in 2D with no edge crossings

Representing graphs: adjacency matrix and adjacency lists

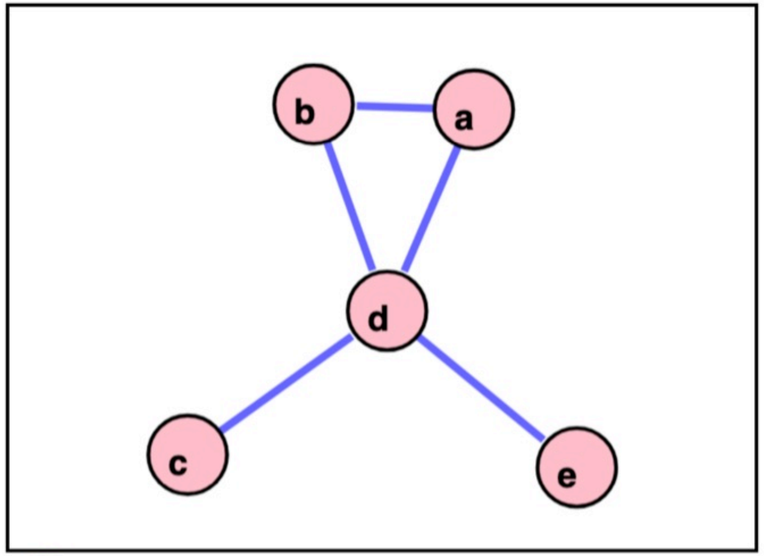
2D array

good for dense graphs

good for sparse graphs

adjacency matrix

each entry (u,v) is
- 1 if edge u to v
- 0 otherwise



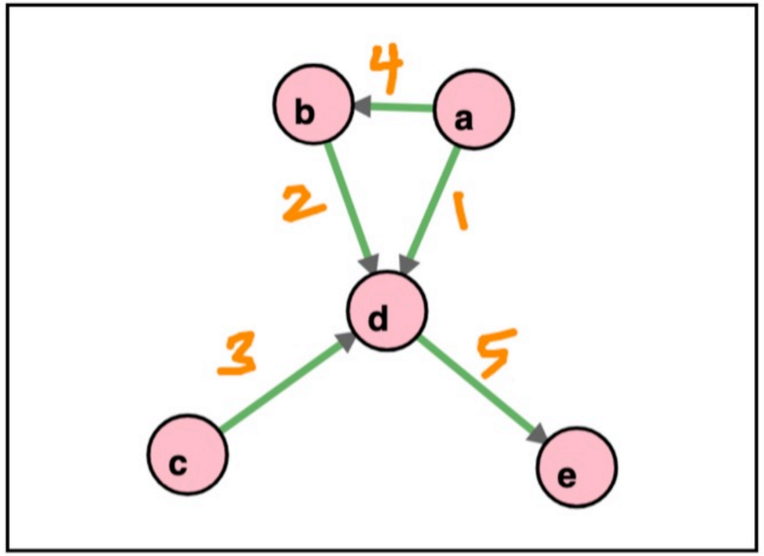
	a	b	c	d	e
a	0	1	0	1	0
b	1	0	0	1	0
c	0	0	0	1	0
d	1	1	1	0	1
e	0	0	0	1	0

← n^2 size for n vertices

adjacency lists

node	list
a	[b,d]
b	[a,d]
c	[d]
d	[a,b,c,e]
e	[d]

Representing directed graphs (also with edge weights)



adjacency matrix

	a	b	c	d	e
a	0	4	0	1	0
b	0	0	0	2	0
c	0	0	0	3	0
d	0	0	0	0	5
e	0	0	0	0	0

adjacency lists

(node, weight)

node	adj. lists
a	[(b,4), (d,1)]
b	[(d,2)]
c	[(d,3)]
d	[(e,5)]
e	[]

Implementing this in Java using adjacency lists (or adjacency sets)

```java
public class Graph<Node> {

  // adjacency lists: node -> set of adjacent nodes
  private HashMap<Node, HashSet<Node>> adj;

  public Graph() {
    adj = new HashMap<>();
  }

  public void addEdge(Node a, Node b) {
    if (!adj.containsKey(a)) {
      adj.put(a, new HashSet<>());
    }
    if (!adj.containsKey(b)) {
      adj.put(b, new HashSet<>());
    }
    adj.get(a).add(b);
    adj.get(b).add(a);
  }

  public boolean hasEdge(Node a, Node b) {
    // or adj.get(b).contains(a) since undirected
    return adj.get(a).contains(b);
  }

  Set<Node> getNodes() {
    return adj.keySet();
  }
}
```

```java
public static void main(String[] args) {
  Graph<Character> graph = new Graph<>();

  graph.addEdge('a', 'b');
  graph.addEdge('a', 'd');
  graph.addEdge('b', 'd');
  graph.addEdge('c', 'd');
  graph.addEdge('d', 'e');
  graph.addEdge('e', 'f');
  graph.addEdge('e', 'g');

  System.out.println(graph.hasEdge('b', 'a'));

  System.out.println(graph.getNodes());
}
```

Brainstorm: how can we design a `getEdges()` method?

{a,b} undirected edge

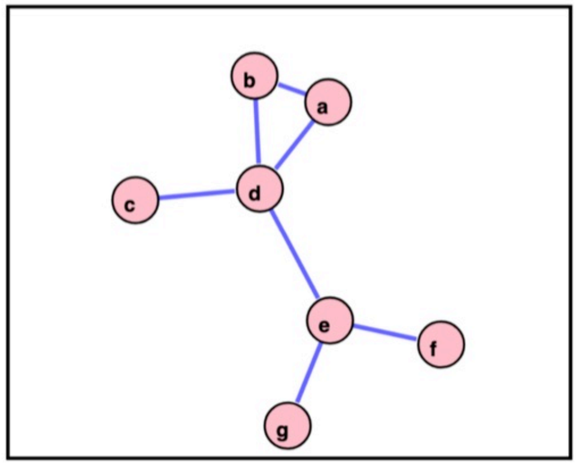
(a,b) directed edge

{ {a,b}, {a,d}, {b,d}, {c,d}, {d,e}, {e,f}, {e,g} }

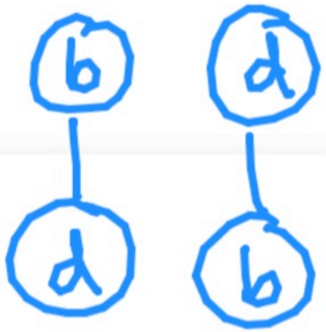
Retrieving the set of edges

Maybe use a helper Edge class?

```java
public class Graph<Node> {

  private HashMap<Node, HashSet<Node>> adj;

  HashSet<Edge> getEdges() {
    HashSet<Edge> edges = new HashSet<>();
    for (Node u : adj.keySet()) {
      HashSet<Node> list = adj.get(u);
      for (Node v : list) {
        edges.add(new Edge(u, v));
      }
    }
    return edges;
  }
}
```




```java
class Edge {
  public Node u;
  public Node v;

  public Edge(Node u, Node v) {
    this.u = u;
    this.v = v;
  }
}
```



are the same

define hashCode() and equals()

```java
class Edge {
  public Node u;
  public Node v;

  public Edge(Node u, Node v) {
    this.u = u;
    this.v = v;
  }

  @Override
  public int hashCode() {
    return 31 * Math.min(u.hashCode(), v.hashCode())
              + Math.max(u.hashCode(), v.hashCode());
  }

  @Override
  @SuppressWarnings("unchecked")
  public boolean equals(Object otherObj) {
    Edge otherEdge = (Edge) otherObj;
    return (u.equals(otherEdge.u) && v.equals(otherEdge.v)) ||
           (u.equals(otherEdge.v) && v.equals(otherEdge.u));
  }

  public String toString() {
    return "(" + u.toString() + ") -- (" + v.toString() + ")";
  }
}
```

Additional notes:

- **Next class** Wednesday 4/30: attend talk by Zale Young '20, 12:45 in MBH 104 -- pizza at 12:35
- **Homework 9** due Thursday 5/1: implement a hash table with linear probing to handle collisions
- **Lab 9** Friday 5/2: Graph Algorithms