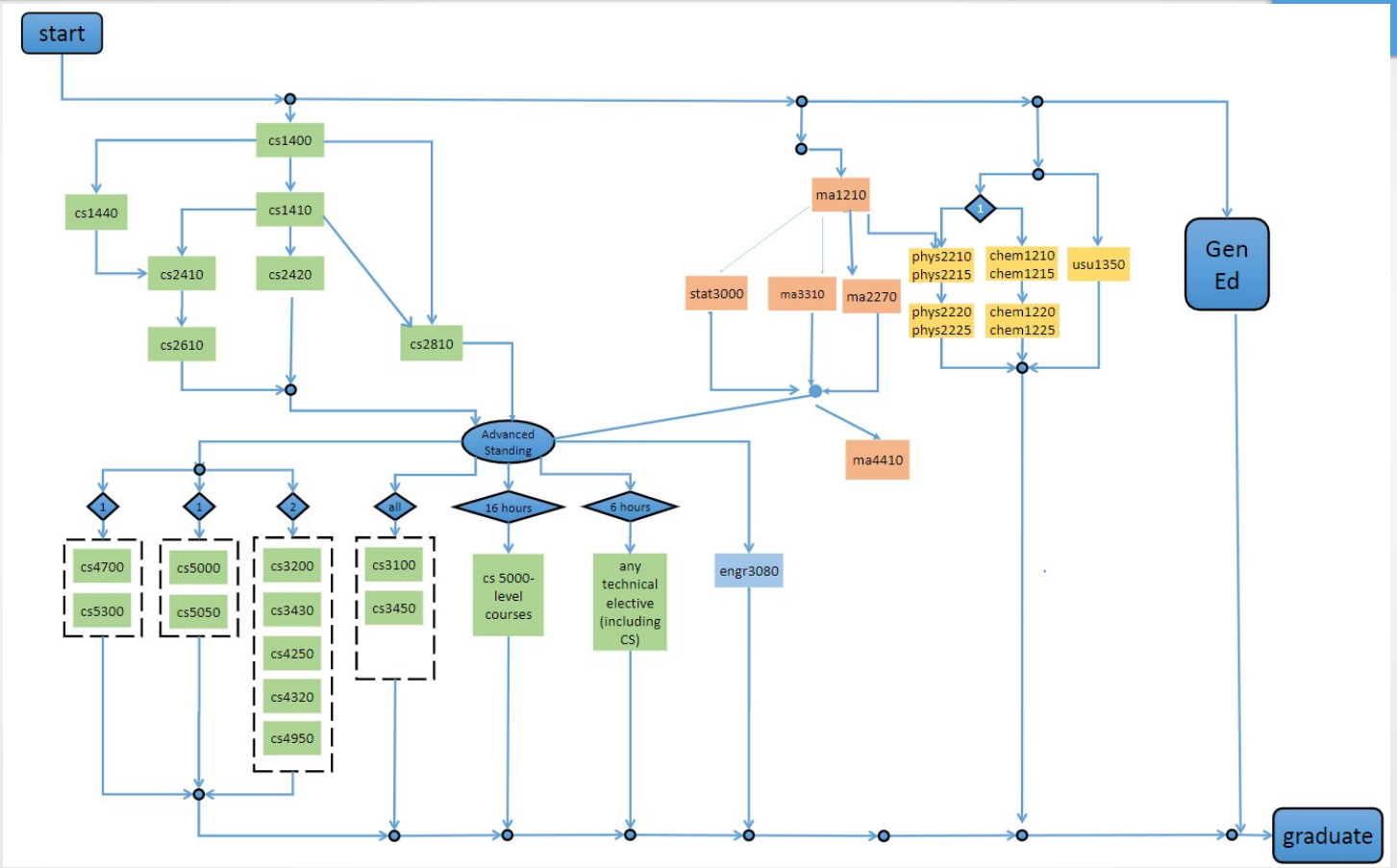
Graphs

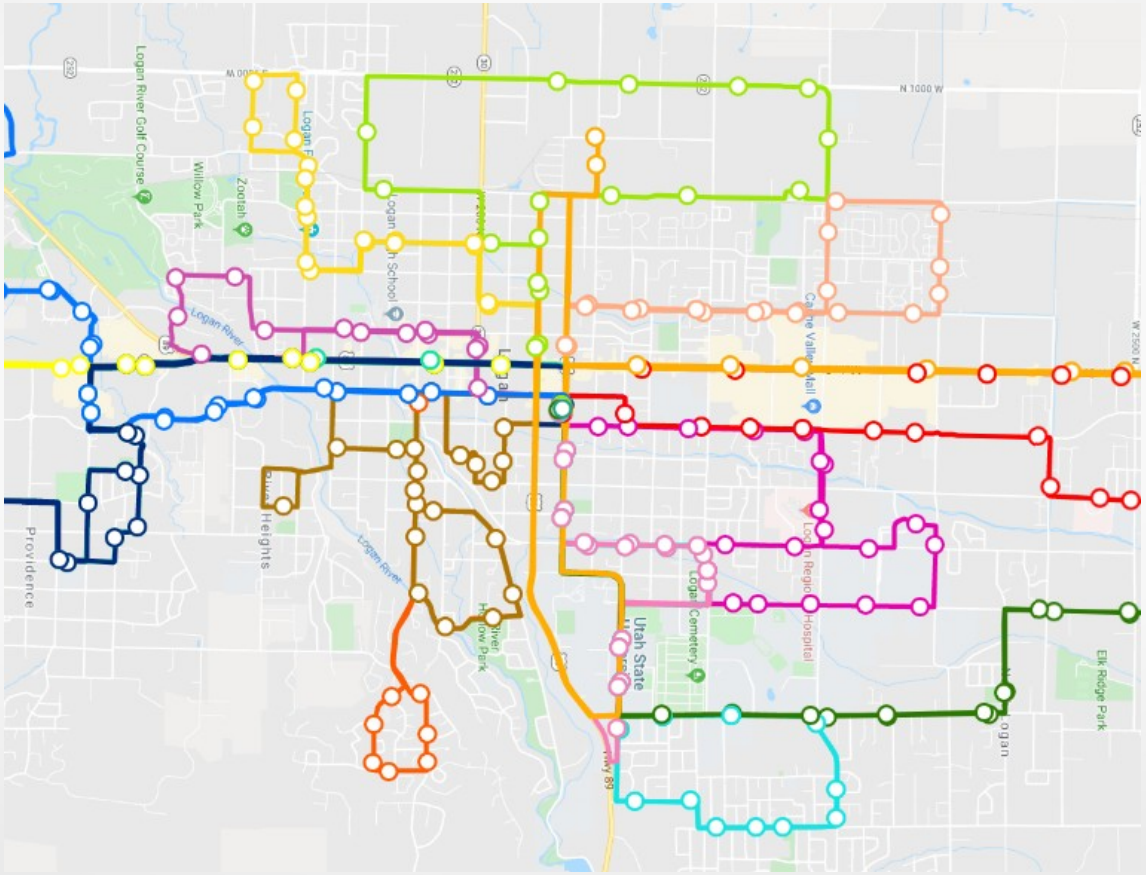
Graphs – A Definition

- A graph is a finite set of nodes with edges between nodes
- A graph G is a structure (V, E) consisting of:
  - a finite set V called the set of nodes (vertices)
  - a set E that is a subset of VxV. That is, E is a set of pairs of the form (x, y) where x and y are nodes in V

Graph – Example 1 : Degree Flow



Graph – Example 2 : Bus Routes



Graphs – Motivation

- What are some characteristics of binary trees?
  - They have a rigid structure where each node has a single node that points to it (or none, in the case of the root); not all things in life we want to represent are structured this way
- For example
  - I need to fly to Beijing
  - I want to find the cheapest way to fly there
  - How could we think about the data?
- For example
  - I have a set of candy bars in a bag
  - I want to distribute so everybody gets a kind they like
  - How could we represent the data?

Graphs – Motivation

- For example
  - I want to travel every bike path in Logan, but don’t want to ever travel the same path twice
  - Can I begin at my home, visit every path once, and then return home?
  - How would I represent the data, how would I perform the traversal?
- Human Graph
  - If you could pick on person who you would go to for help with an assignment, who would it be?
  - Use your arm as a link, point to that person
  - What if you could choose two people

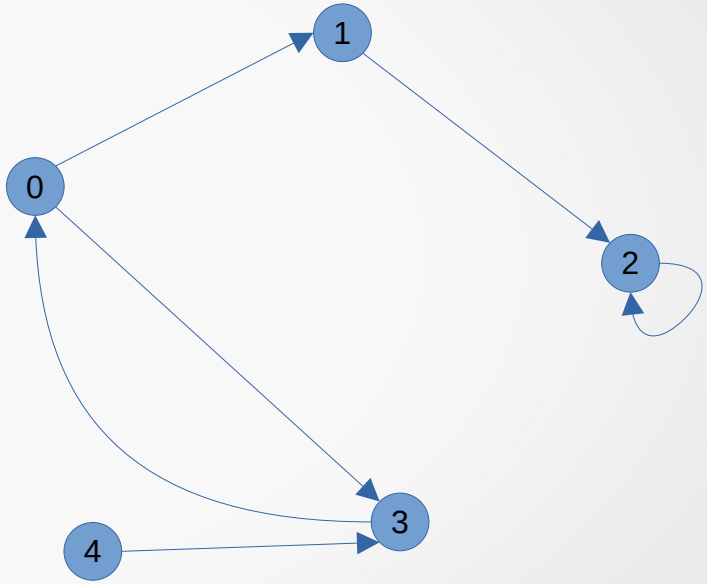
- In all of these cases, the data representation is a graph!

Graphs - Terminology

- ***Graph*** – set of *vertices/nodes* and set of *edges/arcs*
  - More formally, a graph is an ordered pair of finite sets V and E
    - V is the set of vertices
    - E is the set of edges
    - Sometimes the edges have weight; termed a *weighted graph*
    - Sometimes the edges have direction (can traverse from A to B, but not B to A)
      - A graph with directed edges is termed a *directed graph* or *digraph*
      - Otherwise it is called an *undirected graph*
- ***Successor*** – the *follows* relationship in a directed graph
- ***Predecessor*** – the *precedes* relationship in a directed graph
- ***Degree*** – the number of edges incident to or touching a vertex
  - *In-degree* – the number of edges coming into a vertex (digraph only)
  - *Out-degree* – the number of edges going out of a vertex (digraph only)
- ***Connected*** – A graph is connected if you can get from a node to every other node following edges

Graph – “Want to work with” Example

- V = { 0, 1, 2, 3, 4 }
- E = { (0, 1), (0, 3), (1, 2), (2, 2), (3, 0), (4, 3) }



- When (x, y) is an edge, we say
  - x is adjacent to y
  - y is adjacent from x
- From the example
  - 0 is adjacent to 1
  - 4 is not adjacent to 0
  - 2 is adjacent from 1

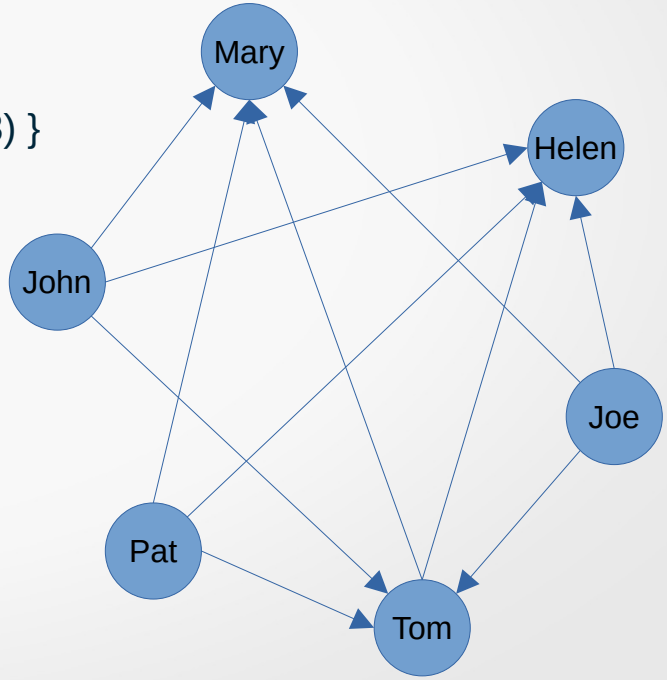
Graphs – “Younger Than” Example

- Draw a graph of five people around you
  - The people become the nodes
  - Let the edges be “is younger than you”
- Questions
  - Is it directed or undirected?
  - Is it connected?

Graphs – “Younger Than” Example

- Draw a graph of five people around you
  - The people become the nodes
  - Let the edges be “is younger than you”

- V = { Mary (15), John (12), Helen (15), Joe (12), Pat (12), Tom (13) }
- E = { (x, y) | if x is younger than y }



Graphs – Intuition

- Nodes represent entities (such as people, cities, computers, words, etc)
- Edges represent relationships between entities x and y, such as…
  - “x likes y” (nodes are people and food)
  - “x knows how to contact y” (nodes are people)
  - “x is a friend of y” (may not be reciprocal)
  - “x wants to work for y” (nodes are people and companies)
  - “x is a child of y” (nodes are people)
  - “there is a flight from x to y” (nodes are cities)
  - “clubs x and y share common members” (nodes are clubs)

Graph Representation

Graph Representation

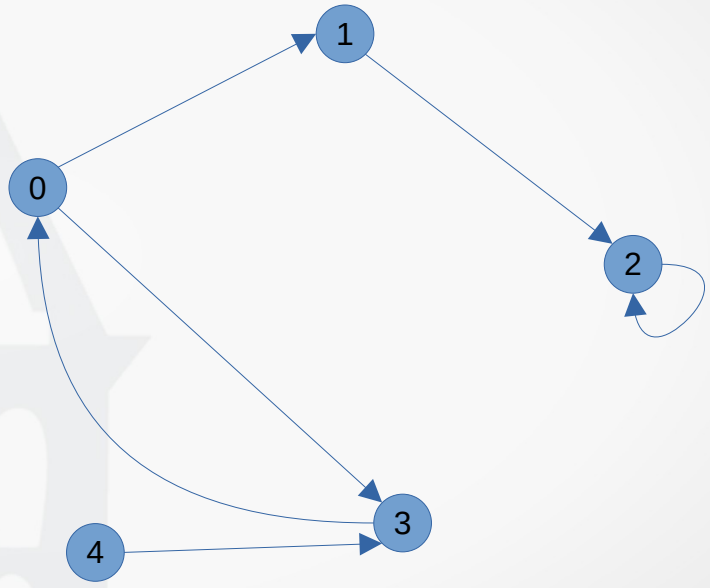
- For graphs to be computationally useful, they have to be conveniently represented in programs
- Two common representations (there are others)
  - Adjacency matrix representation
  - Adjacency list representation

Adjacency Matrix Representation

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re-)labeled 0, 1, 2, …, n - 1
  - Same as with union/find, integers are easier to work with
- A[i][j] = true; if (i, j) is an edge
- A[i][j] = false; if (i, j) is not an edge

Adjacency Matrix Representation – “Want to work with” Example

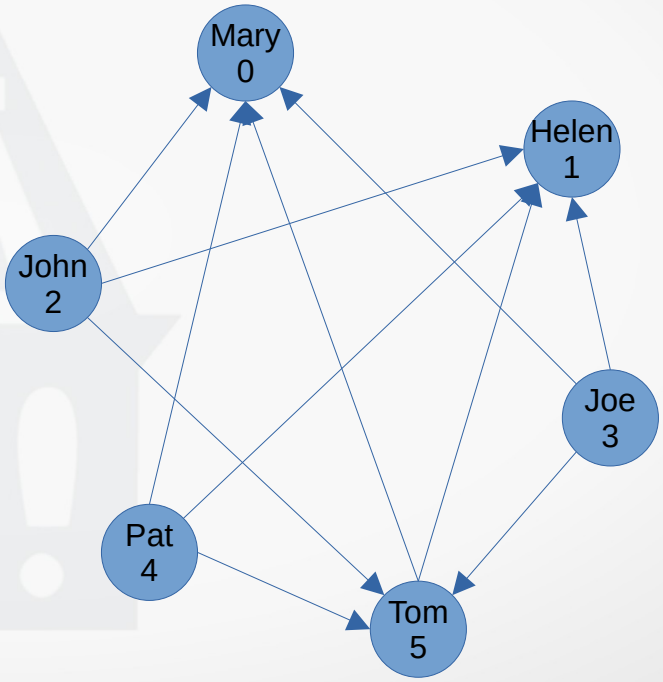
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



Adjacency Matrix Representation – “Younger Than” Example

- Re-label the nodes with numerical labels

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency Matrix – Pros/Cons

- Pros
  - Simple to implement
  - Easy and fast to tell if a pair (i, j) is an edge
    - simply check if A[i][j] is true or false

- Cons
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ memory

Adjacency List Representation

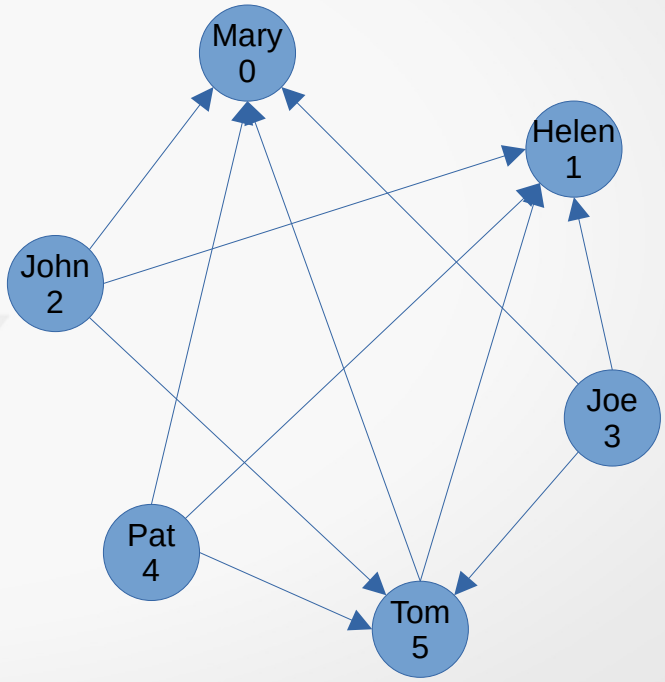
- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i
  - The nodes in list L[i] may or may not be in any particular order

Adjacency List Representation – “Younger Than” Example

- L[0]: empty
- L[1]: empty
- L[2]: (0) => (1) => (5)
- L[3]: (0) => (1) => (5)
- L[4]: (0) => (1) => (5)
- L[5]: (0) => (1)

Note: The linked list isn’t necessary in order

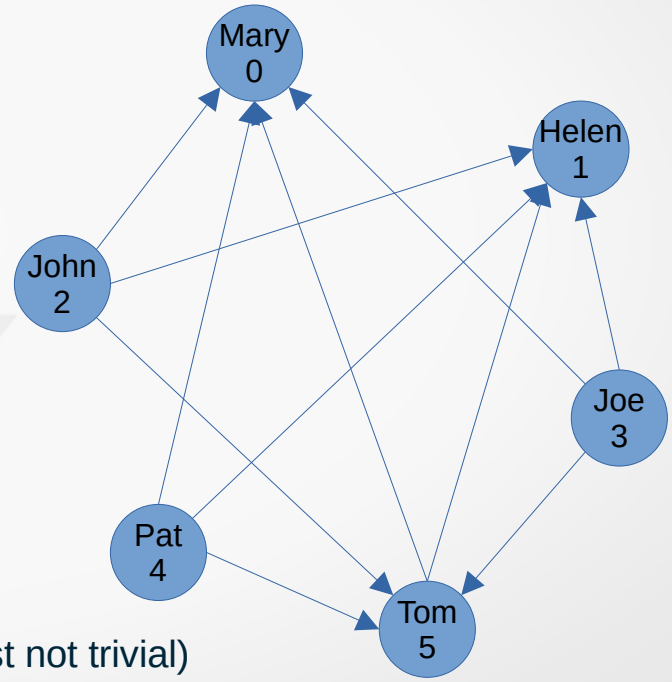
- What operations are easy?
- What operations are hard?



Adjacency List Representation – “Younger Than” Example

- L[0]: empty
- L[1]: empty
- L[2]: (0) => (1) => (5)
- L[3]: (0) => (1) => (5)
- L[4]: (0) => (1) => (5)
- L[5]: (0) => (1)

Note: The linked list isn’t necessary in order



- What operations are easy?
  - What are the nodes that follow John?
- What operations are hard?
  - Is there an edge from Joe to John? (it isn’t ***that*** hard, just not trivial)
  - Who is younger than Helen?

Adjacency List – Pros/Cons

- Pros
  - Saves space on memory: the representation only takes as many memory elements as there are nodes and edges
- Cons
  - It can take up to O(n) time to determine if a pair of nodes (i, j) share an edge
    - Would have to search the linked list L[i], which takes time proportional to the length of L[i]

Reminder – Directed vs Undirected Graphs

- If the directions of the edges matter, then we show the edge directions
  - Called a ***directed graph***, or ***digraph***
- If the relationships represented by the edges are symmetric (such as x is a sibling of y), then we don't show the directions of the edges
  - Called an ***undirected graph***

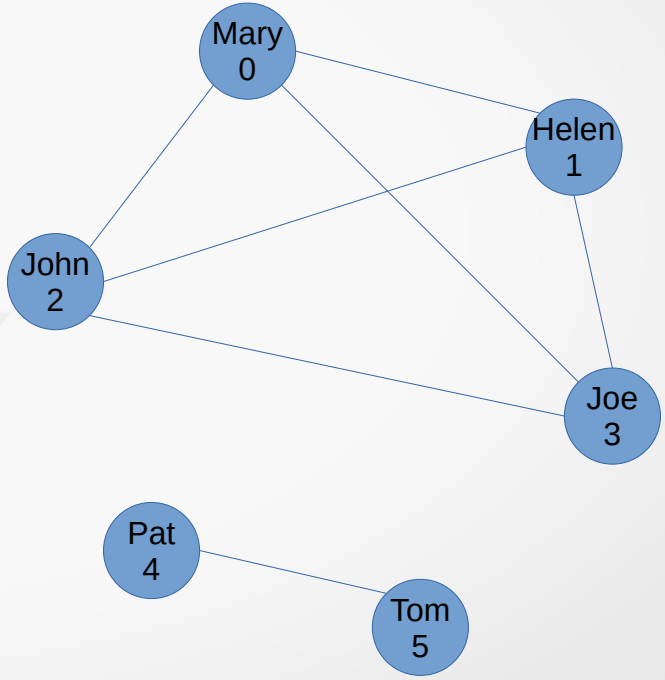
Example Representations - Undirected

Adjacency List

- L[0]: (1) => (3) => (2)
- L[1]: (3) => (2) => (0)
- L[2]: (0) => (1) => (3)
- L[3]: (2) => (0) => (1)
- L[4]: (5)
- L[5]: (4)

Adjacency Matrix

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



Example Representations - Undirected

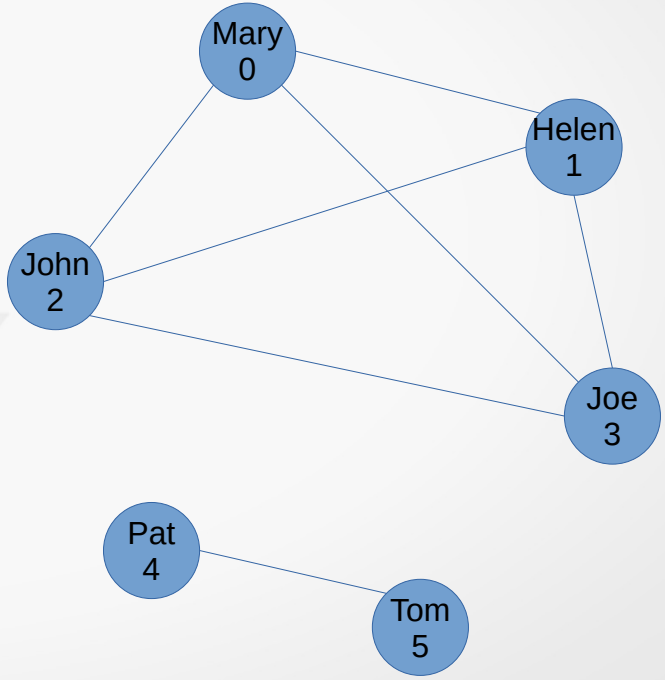
Notice: edges appear twice

Adjacency List

- L[0]: (1) => (3) => **(2)**
- L[1]: (3) => (2) => (0)
- L[2]: **(0)** => (1) => (3)
- L[3]: (2) => (0) => (1)
- L[4]: (5)
- L[5]: (4)

Adjacency Matrix

$$A = \begin{bmatrix} 0 & 1 & \mathbf{1} & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ \mathbf{1} & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



Adjacency List Representation – Example Code

```
public class Graph {
    private ArrayList<LinkedList<EdgeInfo>> adjacencyList;

    public Graph(int numberNodes) {
        adjacencyList = new ArrayList<LinkedList<EdgeInfo>>(numberNodes);
        for (int i = 0; i < numberNodes; i++) {
            adjacencyList.add(new LinkedList<EdgeInfo>());
        }
    }

    private boolean isValidNode(int index) {
        return index >= 0 && index < adjacencyList.size();
    }

    public void addEdge(int to, int from) {
        if (!isValidNode(to) || !isValidNode(from)) {
            throw new IndexOutOfBoundsException();
        }

        adjacencyList.get(from).addLast(new EdgeInfo(from, to));
    }
}
```

Adjacency List Representation – Example Code

```
private class EdgeInfo {
   public EdgeInfo(int from, int to) {
        this.from = from;
        this.to = to;
   }
   public String toString() {
       return from + "->" + to + " " ;
   }
   int from; // source of edge
   int to; // destination of edge
}
```

Adjacency List Representation – Example Code

```
private class EdgeInfo {
   public EdgeInfo(int from, int to) {
        this.from = from;
        this.to = to;
   }
   public String toString() {
       return from + "->" + to + " " ;
   }
   int from; // source of edge
   int to; // destination of edge
}
```

```
public String toString(String message) {
    StringBuilder sb = new StringBuilder();
    sb.append(message + "\n");
    int node = 0;
    for (LinkedList<EdgeInfo> a : adjacencyList) {
        sb.append((node++) + ": ");
        for (EdgeInfo e : a) {
            sb.append(e + " ");
        }
        sb.append("\n");
    }
    return sb.toString();
}
```

Practice

write the code to print out a node’s successors

Adjacency List Representation – Print Successors

```
public void printSuccessors(int n) {
    System.out.print(n + ": ");
    for (EdgeInfo e : adjacencyList.get(n)) {
        System.out.print( "->" + e.to);
    }
    System.out.println();
}
```

Practice

write the code to count all the edges in the graph

Adjacency List Representation – Count Edges

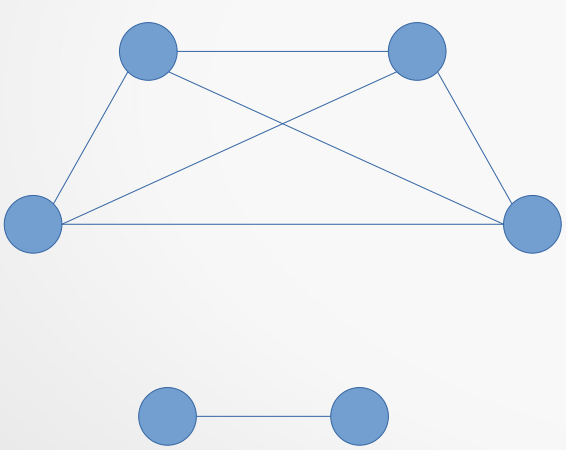
```
// Assumes no self edges in undirected graph

public int countEdges() {
    int count = 0;
    for (LinkedList<EdgeInfo> a : adjacencyList) {
        for (EdgeInfo e : a) {
            count++;
        }
    }
    // if undirected, counted all edges twice
    return count / 2;
}
```

Directed vs Undirected Graphs (again)

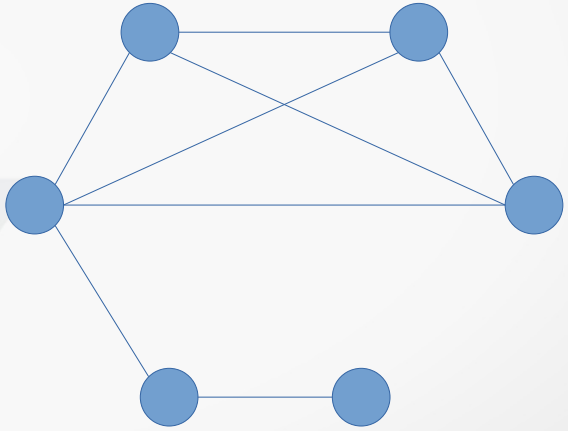
- Let G be a directed graph
  - The ***in-degree*** of a node x in G is the number of edges coming to x
  - The ***out-degree*** of a node x in G is the number of edges leaving x
- Let G be an undirected graph
  - The ***degree*** of a node x in G is the number of edges that have x as one of their end nodes
  - The ***neighbors*** of x in G are the nodes adjacent to x

Graph Connectivity – Undirected Graph

- An undirected graph is said to be ***connected*** if there is a path between every pair of nodes
  - Otherwise, the graph is disconnected



Disconnected



Connected

Graph Connectivity – Connected Components

- If an undirected graph is not connected, then each “piece” is called a ***connected component***
- If the graph is connected, then the whole graph is one single connected component

Graph Connectivity – Connected Components

- If an undirected graph is not connected, then each “piece” is called a ***connected component***
- If the graph is connected, then the whole graph is one single connected component

- Practice
  - Write the algorithm to determine if G is connected
  - Write the algorithm to find a graph’s connected components

Is Connected – Algorithm

- Mark all nodes as unvisited
- Select an unvisited node
  - Mark the node as visited
- Recursively visit all nodes reachable from this node
  - Mark each node as visited
- Are all nodes visited
  - Yes: connected
  - No: disconnected

Practice

write the code to determine if a graph is connected

Adjacency List Representation – Is Connected

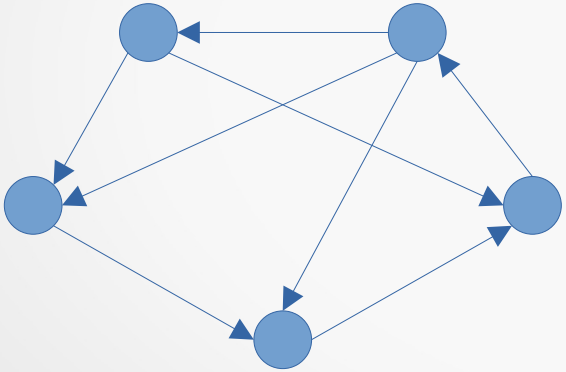
```
// Must add a 'visited' ArrayList to the Graph class
// Alternatively, add a 'visited' property to the nodes

public boolean isConnected() {
    clearVisitedFlags();
    traverse(0);
    for (int i = 0; i < vertexCount; i++) {
        if (!visited[i]) return false;
    }
    return true;
}

public void traverse(int n) {
    if (visited[n]) return;
    visited[n] = true;
    for (EdgeInfo e : adjacencyList.get(n)) {
        traverse(e.to);
    }
}
```

Find Connected Components – Algorithm

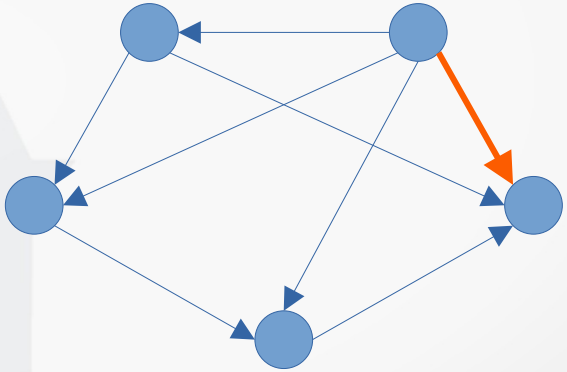
- Mark all nodes as unvisited
- Initialize component (piece) name to 0
- Repeat
  - Select an unvisited node
    - Mark the node as visited
    - Mark the node's component
  - Recursively visit all nodes reachable from this node
    - Mark each node as visited
    - Mark the node's component
  - Increment component name by 1

Strongly Connected Components – Directed Graph

- Every pair of vertices are reachable from each other
- Graph G is ***strongly connected*** if, for every (u, v) in V, there is some path from *u to v* and some path from *v to u*



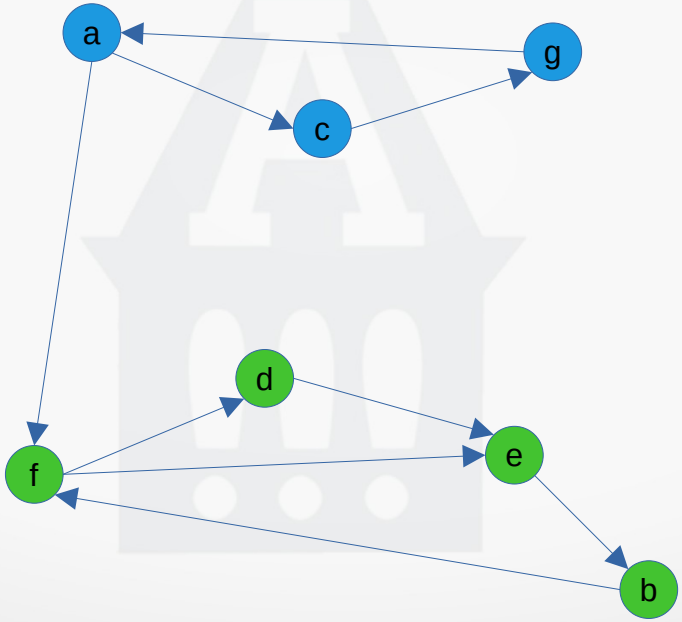
Strongly Connected



Not Strongly Connected

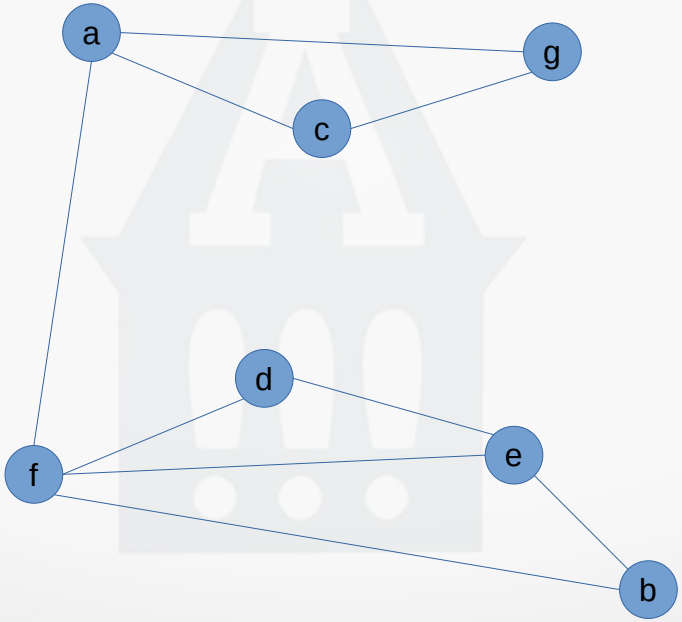
Strongly & Weakly Connected Components – Directed Graph

- Two strongly connected components
  - { a, c, g }
  - { f, d, e, b }



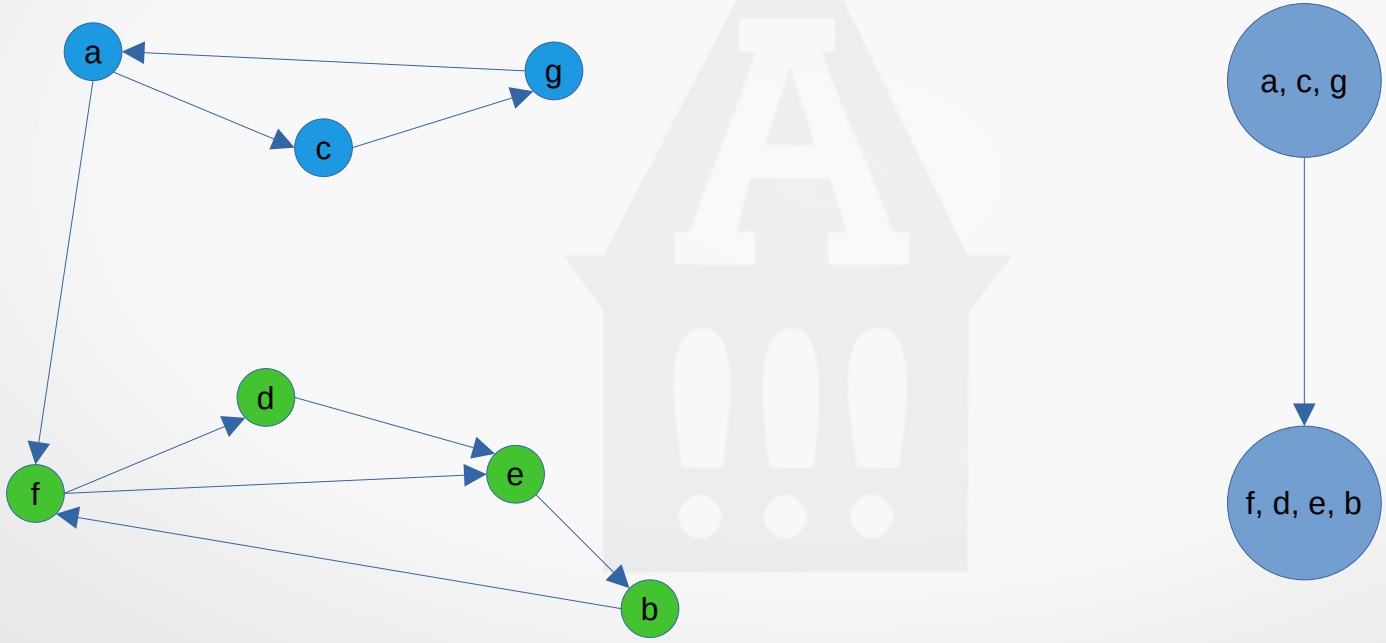
Strongly & Weakly Connected Components – Directed Graph

- A graph is ***weakly connected*** if the underlying undirected graph is connected



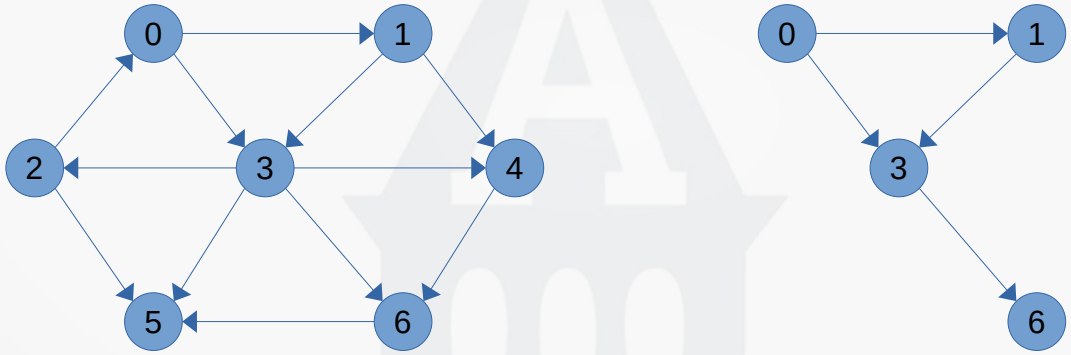
Strongly Connected Components

- Can collapse a Strongly Connected Component (SCC) into a single ***supernode***



Subgraph

- Suppose the graph on the left represents one way streets in a town and the nodes represent hotels
- Maybe I only care about hotels 0, 1, 3, 6. I can take those nodes and edges between them and talk about the ***subgraph***



- Subgraph – a graph that consists of a subset of vertices and all the edges between those vertices

Graphs – Additional Info

- By definition a ***simple graph*** does not contain
  - Multiple copies of the same edge
  - ***Self-edges*** – edges of the form (a, a); this is also called a ***loop***
    - Also known as a ***self-loop***; an edge that connects a vertex to itself
- If we allow multiple edges between the same pair of nodes, it is termed a ***multigraph***
  - e.g., multiple edges between hotels could represent two different roads between the hotels
- Paths
  - ***Path*** – a sequence of edges
  - ***Simple Path*** – there are no repeated vertices or edges
  - ***Length of a path*** – the sum of the lengths/weights of the edges on the path
- Cycle
  - A path that begins and ends at the same place
  - ***Simple Cycle*** – a cycle with no repeated vertices; except for the beginning and ending vertex