# CS 361
# Data Structures & Algs
# Lecture 11

Prof. Tom Hayes
University of New Mexico
09-28-2010

# Last Time

- Priority Queues & Heaps
  - Heapify (up and down)
    - 1: Preserve shape of tree
    - 2: Swaps restore heap order property
  - Balanced Binary Tree using Array
- Quiz #2
- New Reading: secs 3.1 thru 3.4

# Quiz 2 grades

20, 20, 20, 20, 19, 17

16, 16, 15, 15, 15

14, 14, 14, 14, 13

12, 12, 12

11, 11, 11, 11

10, 9, 9, 9

7, 7, 3, 0

# Today

P.A. 2 due Monday, Oct 11

Graphs and Trees, terminology

Connectedness, Components

Traversal Algorithms

- Breadth First vs. Depth First

Testing Bipartiteness

# Graphs

A graph is a pair, (V,E), where:

- V is the set of vertices (also called “nodes”)
- E is a set of edges
- Each edge consists of a pair of vertices, called the endpoints of the edge.

Example: V = {1,2,3,4,5},
E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4} }.

(5 vertices, 6 edges).

# Kinds of Graphs

Vertices: can represent almost anything. Cities, people, computers, numbers.

Edges: represent some notion of "adjacency" or relationships like "knowing", "meeting", "liking", "being similar to." Anything that can involve (or not involve) a pair of vertices.

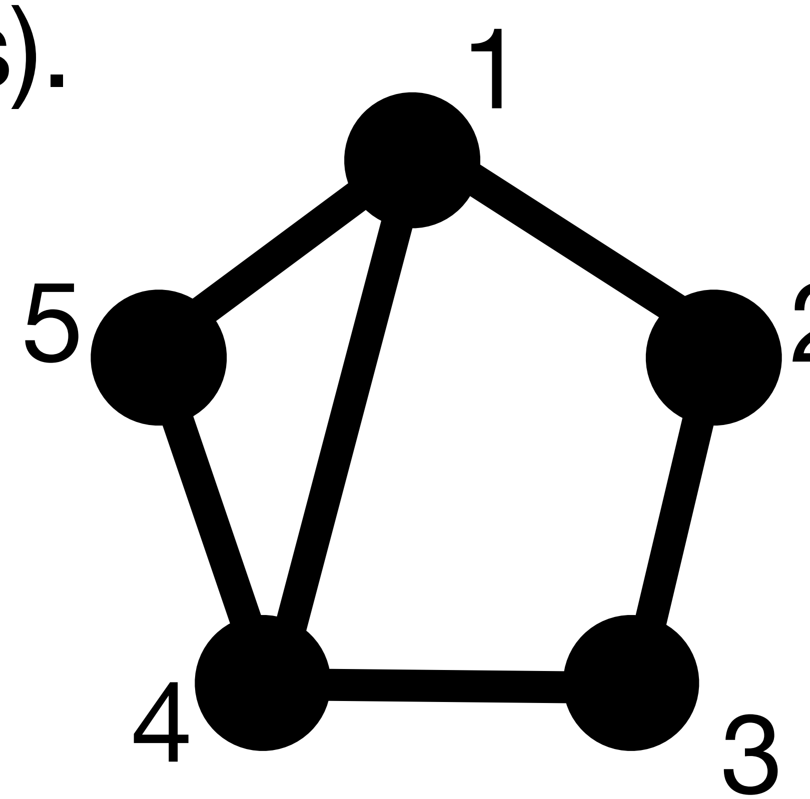
Sometimes: we also want to attach weights to the edges and/or the vertices. But not for today.

# Drawing a graph

A diagram of a graph is a picture, with a "dot" for each vertex, and a "segment" for each edge.

Example: V = {1,2,3,4,5},
E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4} }.

(5 vertices, 6 edges).



# Paths & Connectedness

A graph is connected if, for any two nodes v,w, there is a sequence of edges that joins v to w. A minimal such sequence of edges is called a “path” from v to w.

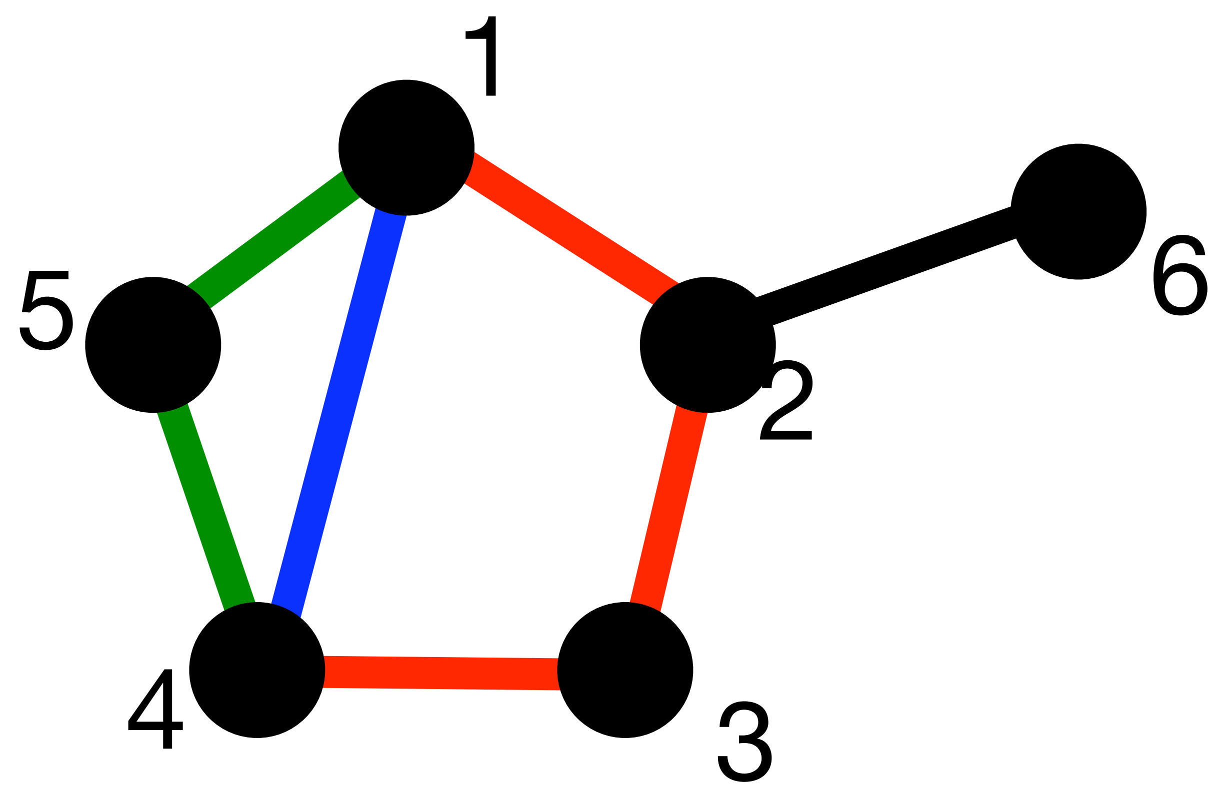
Example: one path from 1 to 4 is {1,2}, {2,3}, {3,4}.

Another is {1,4}.

A third is

{1,5},{5,4}.

{2,6} is on no path



# Components

Two nodes in a graph are said to be "in the same connected component" if there exists a path joining them.

Claim: this is an equivalence relation. Why?

Consequence: Every graph decomposes in a unique way into its connected components.

Obs: G is connected iff G has only one connected component.

# Equivalence Relations

Let ~ be a binary relation on a set S. (For elements a, b in S, "a ~ b" is a proposition which can be true or false.)

We say "~ is an equivalence relation" if 3 axioms hold:

1) reflexive. "a ~ a" is always true.

2) symmetric. "a ~ b" is equivalent to "b ~ a"

3) transitive. If a~b and b~c, then a~c.

# Equivalence Classes

Suppose ~ is an equivalence relation on S.

Then S can be decomposed into subsets $S_1$, $S_2$, $S_3$, etc. called "equivalence classes," meaning:

For all a,b in same equiv. class $S_i$, we have a~b.

For all a in $S_i$, b in $S_j$, $i \neq j$, we have NOT(a~b).

Equivalence classes are an alternative way of defining an equivalence relation.

# Components

a, b : vertices in graph G.

"a ~ b":  "there is a path from a to b"

Equivalence relation.

Equivalence class containing a: All vertices that can be reached by a path from a.  "Connected component containing a."

# of connected components?

# Components

a, b : vertices in graph G.

"a ~ b": "there is a path from a to b"

Equivalence relation.

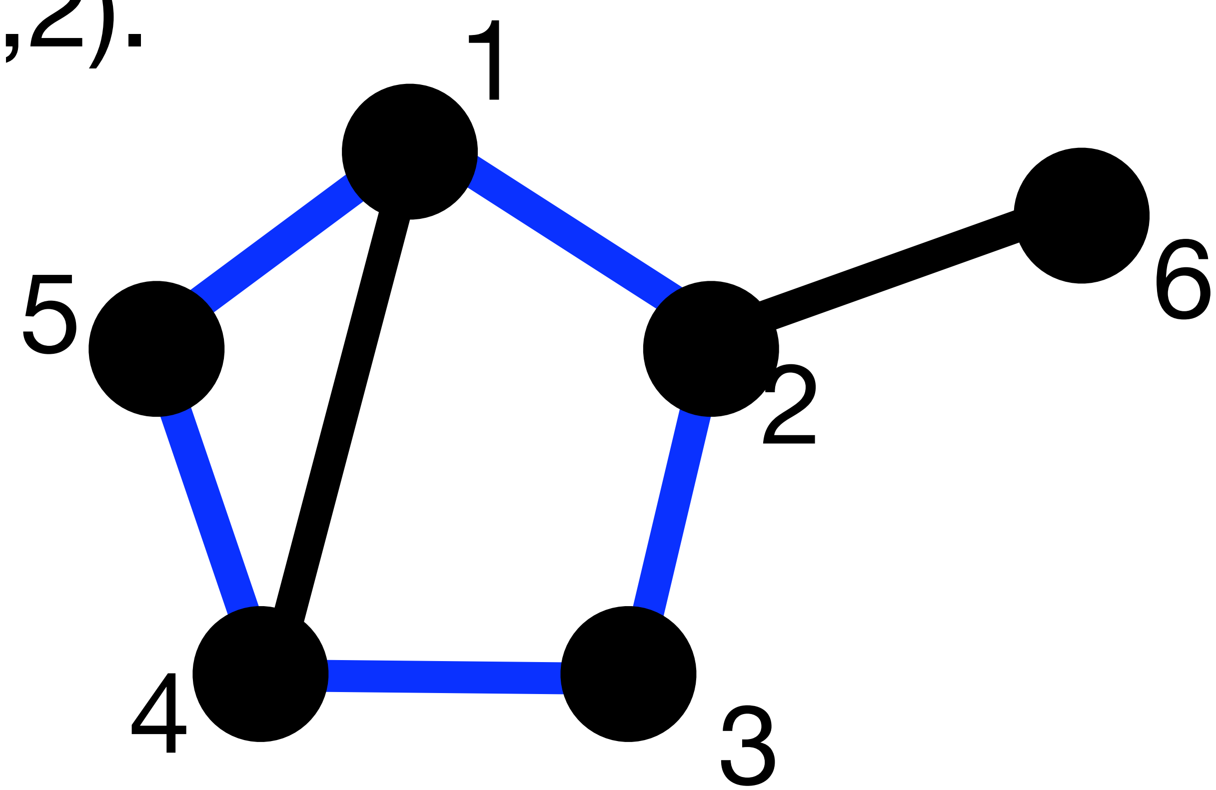
Equivalence class containing a: All vertices that can be reached by a path from a. "Connected component containing a."

# of connected components? Between 1 and n, where n=#vertices in G.

# Cycles

Def: A cycle in a graph is a closed loop with no repeated edges or nodes (except the start and end).

Example: In this graph, (1,2,3,4,5) is a cycle. So is (1,4,5). So is (1,4,3,2).



# Trees & Forests

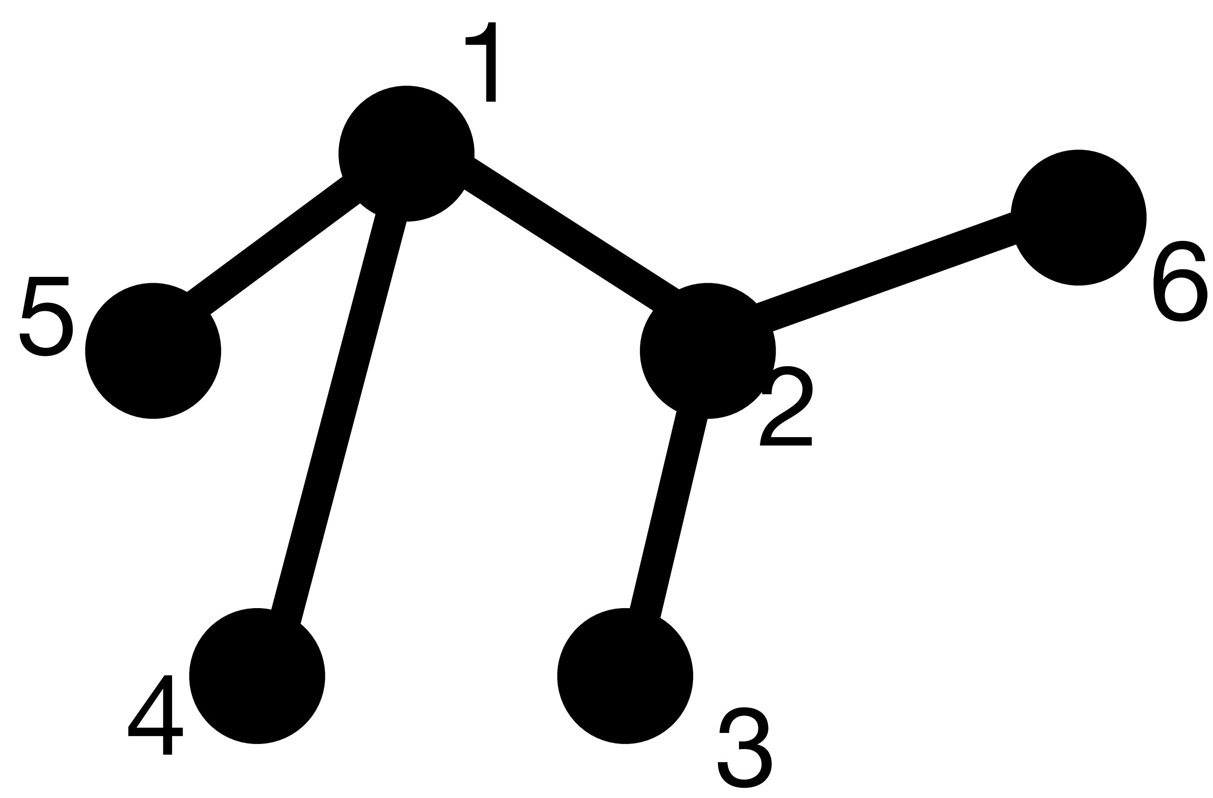
Def: A forest is a graph with no cycles.

Def: A tree is a connected graph with no cycles.

Remark: Every forest is a union of trees.

A tree is a special case of a forest.

Example: a tree.



# Trees & Forests

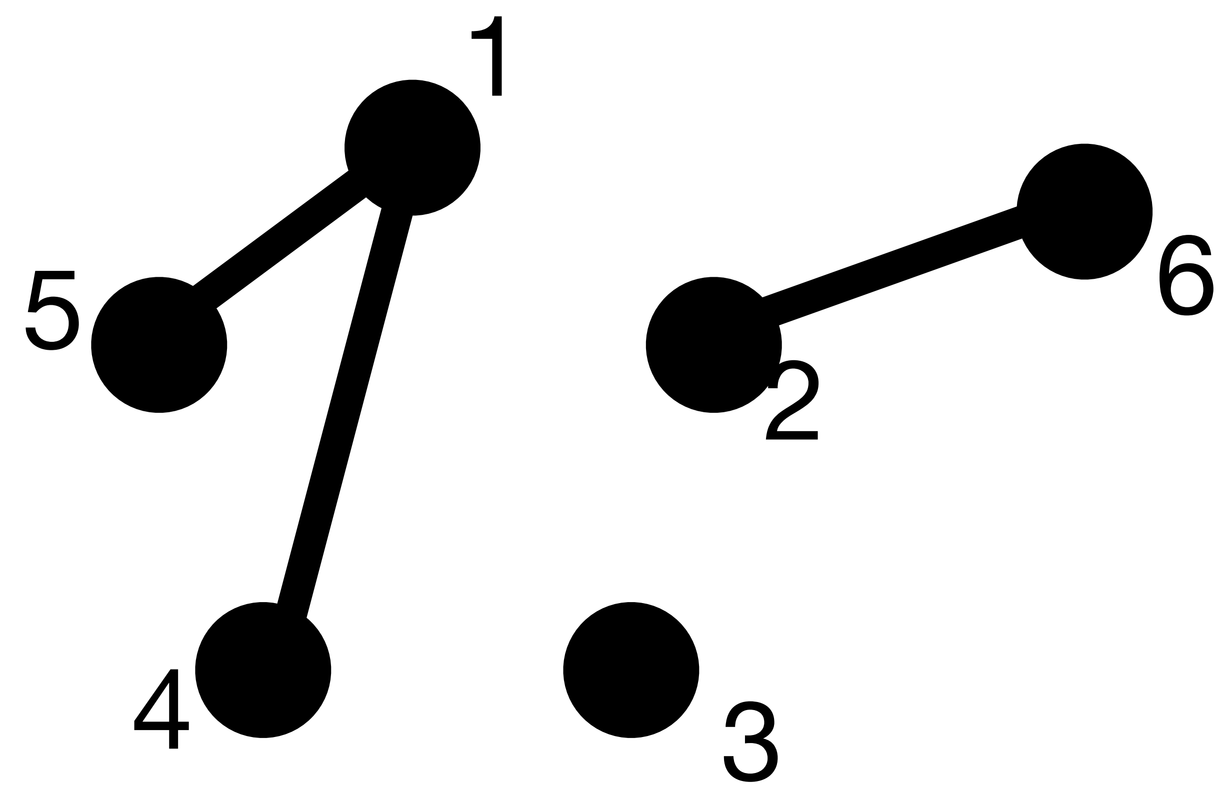
Def: A forest is a graph with no cycles.

Def: A tree is a connected graph with no cycles.

Remark: Every forest is a union of trees.

A tree is a special case of a forest.

Example: a forest.

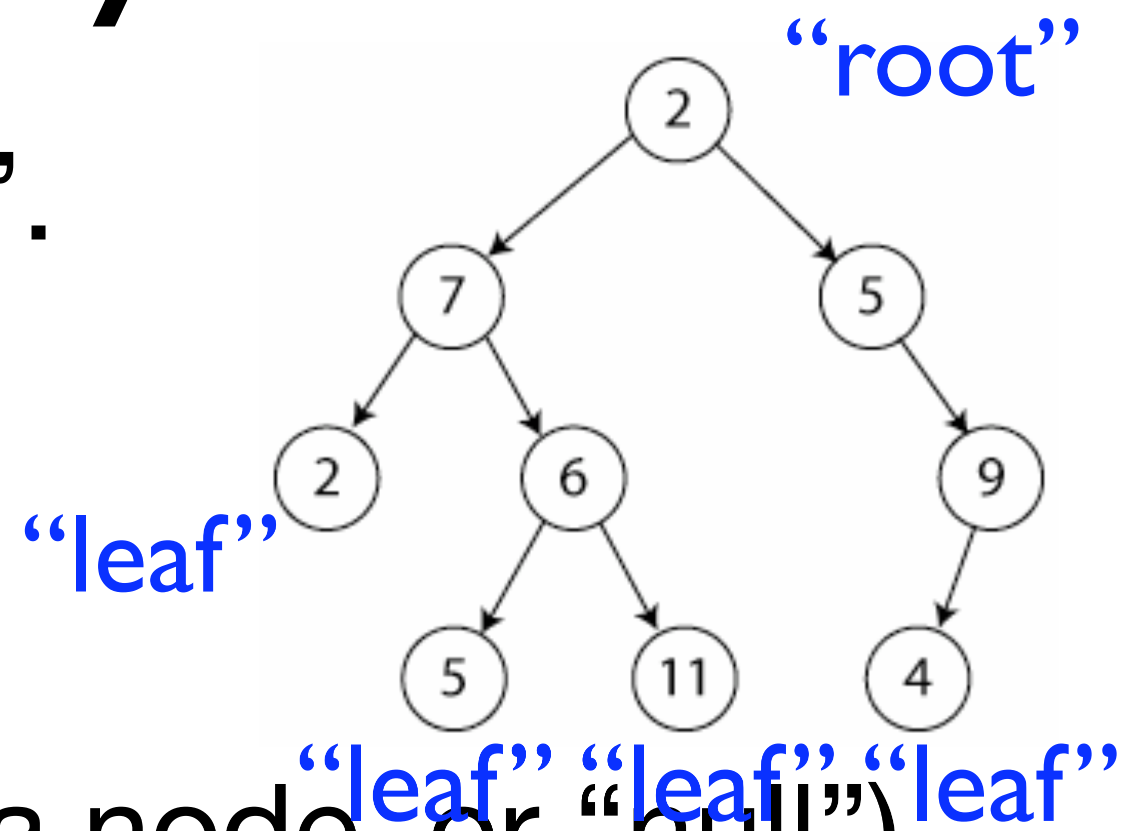


# Recall: Binary Trees

Data is stored in “nodes”.

Each node has 4 fields:

- data
- parent (either a ref to a node, or “null”)
- left_child (reference to a node or null)
- right_child (reference to a node or null)




The graph for a binary tree is a tree.
(Connected, no cycles)

# Rooted Trees

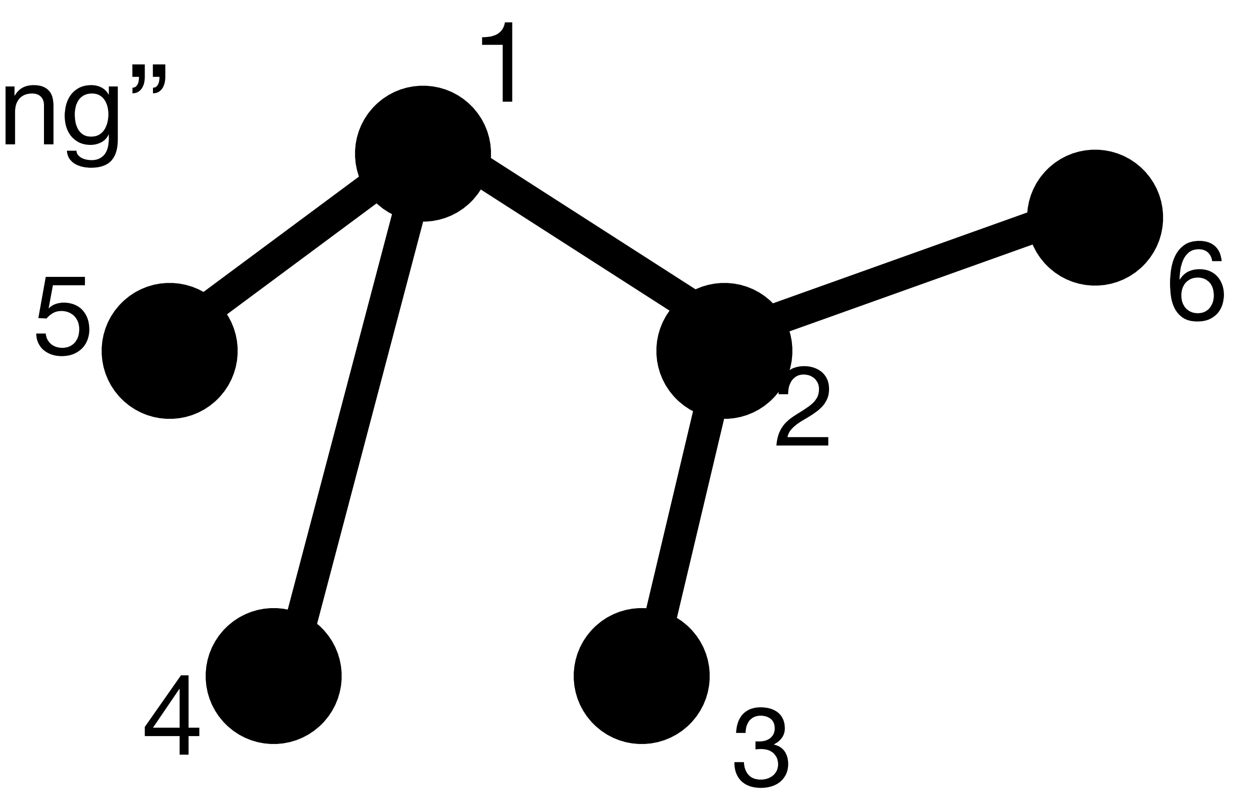
A binary tree has a special node called the root.

Every node, v, has a unique path to the root.

parent(v) is the first node along this path.

In a general tree, any node can be made the root.

This is called “rooting”



# Rooted Trees

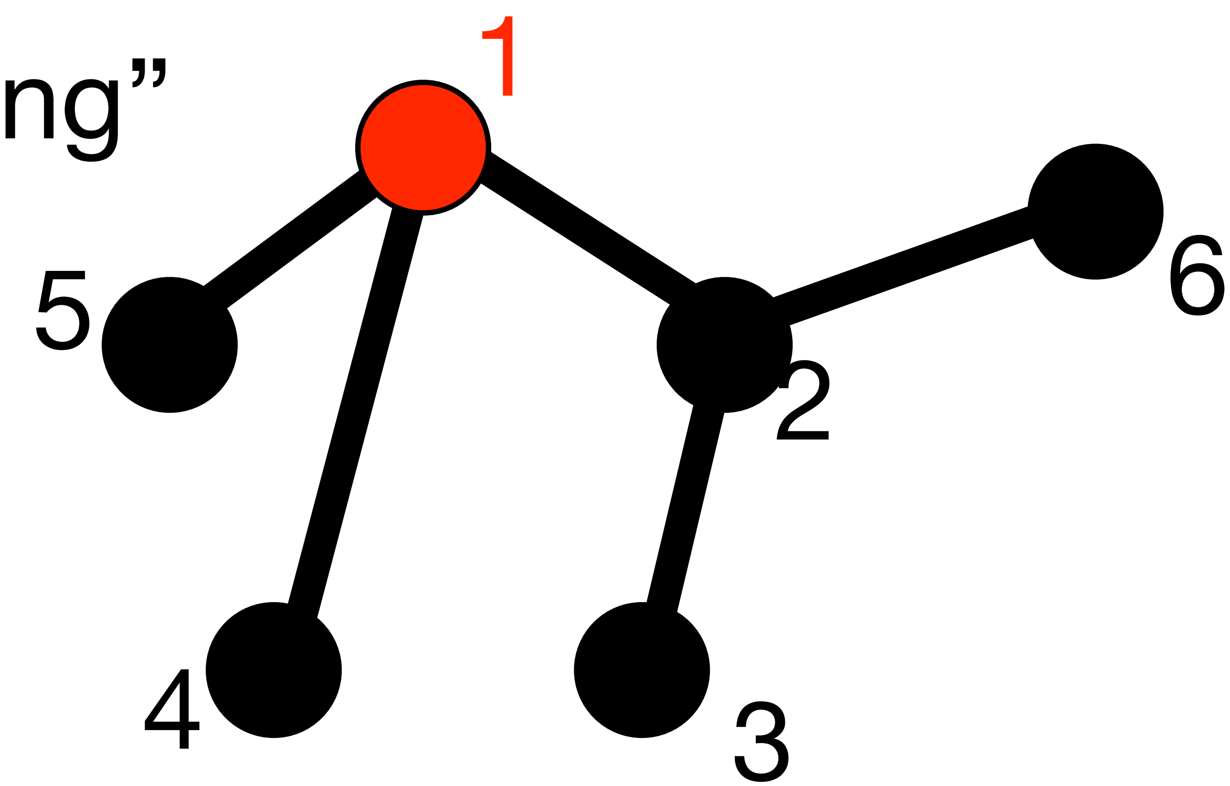
A binary tree has a special node called the root.

Every node, v, has a unique path to the root.

parent(v) is the first node along this path.

In a general tree, any node can be made the root.

This is called "rooting"



# Rooted Trees

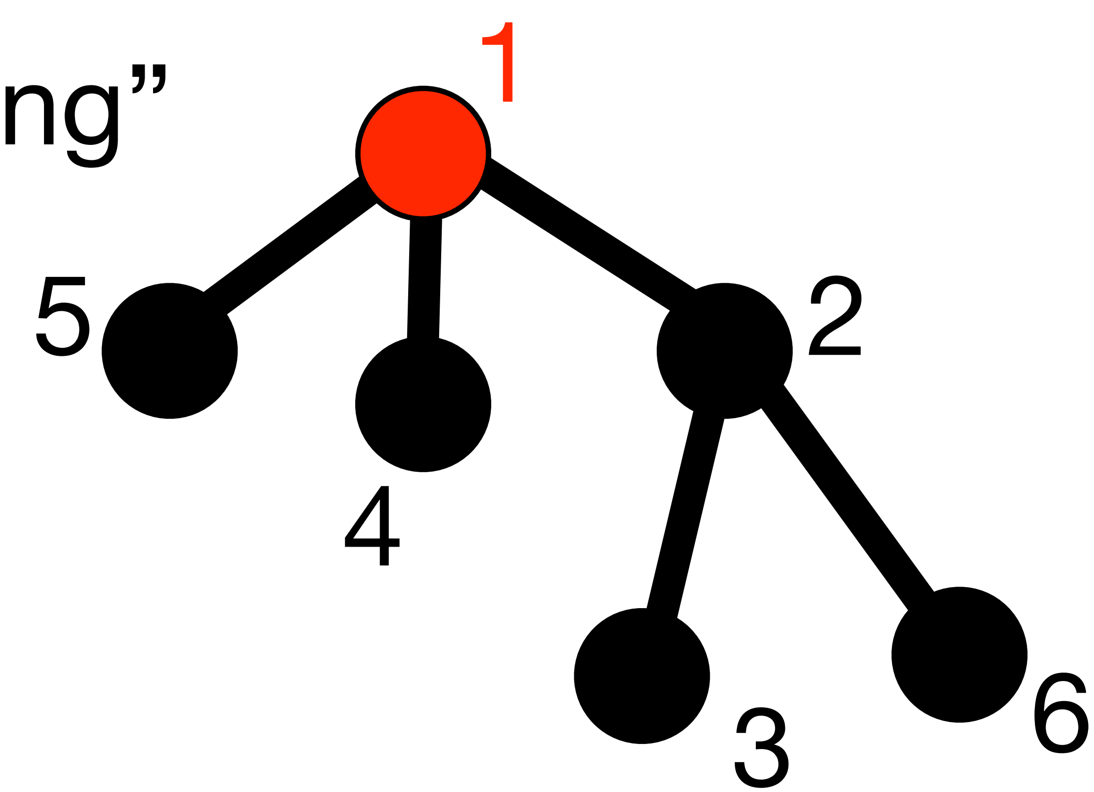
A binary tree has a special node called the root.

Every node, v, has a unique path to the root.

parent(v) is the first node along this path.

In a general tree, any node can be made the root.

This is called “rooting”



# Rooted Trees

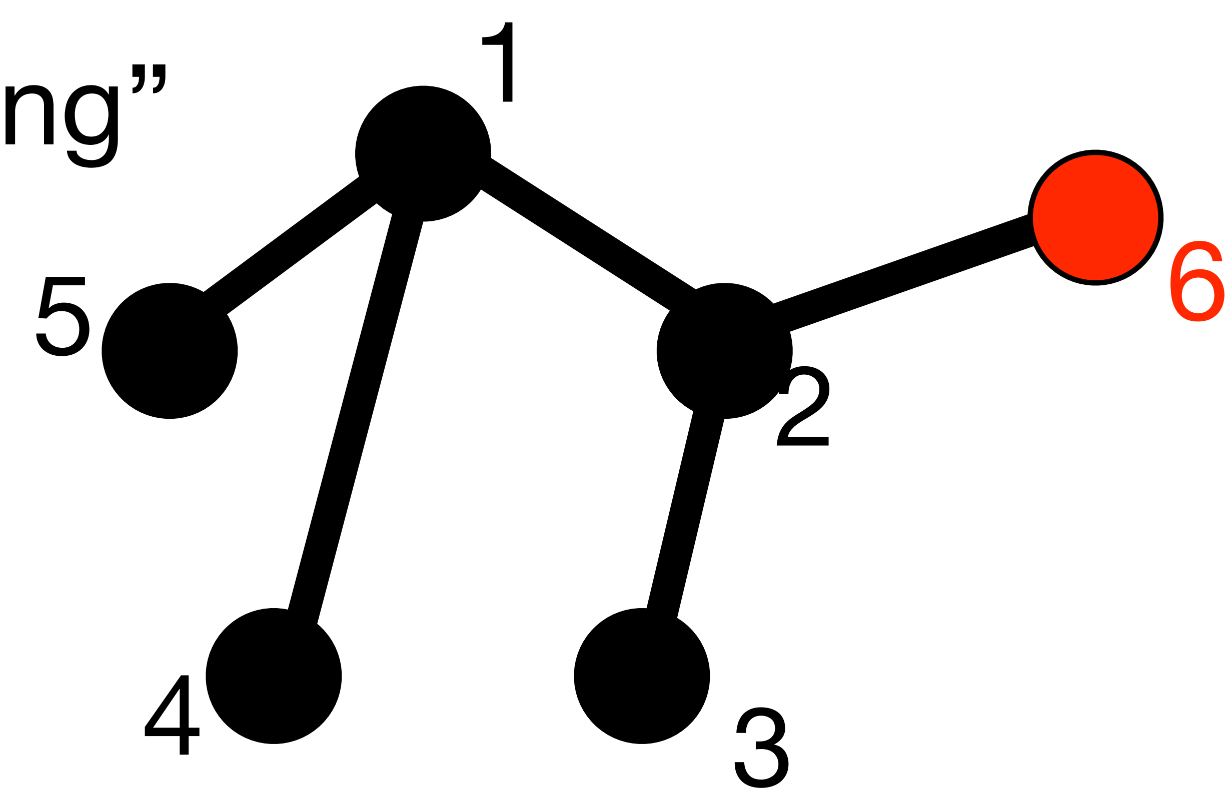
A binary tree has a special node called the root.

Every node, v, has a unique path to the root.

parent(v) is the first node along this path.

In a general tree, any node can be made the root.

This is called "rooting"



# Rooted Trees

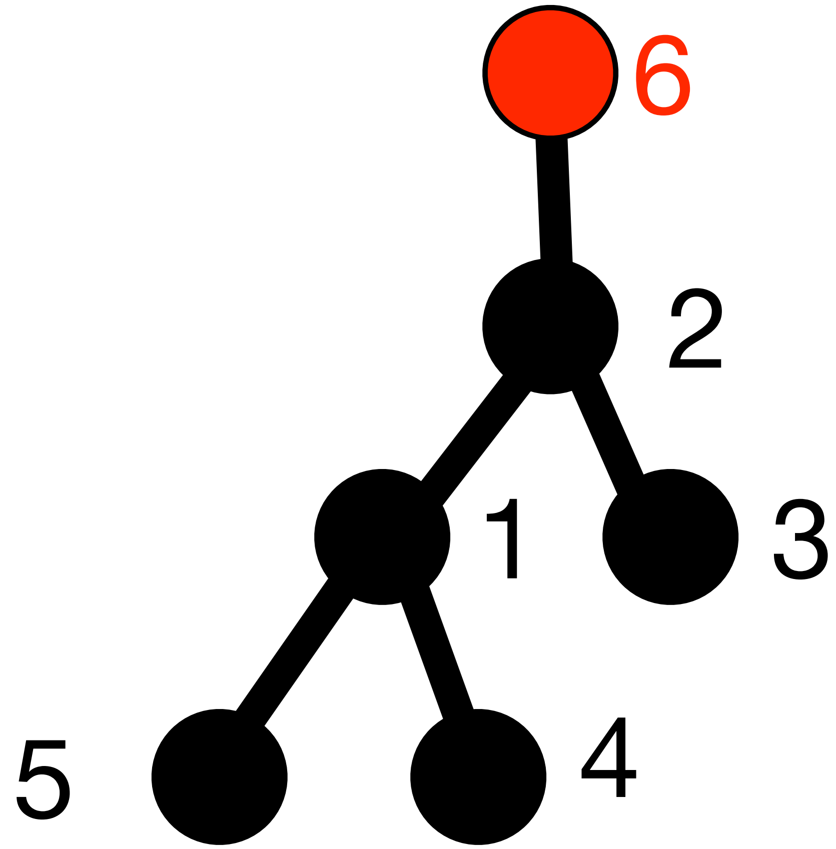
A binary tree has a special node called the root.

Every node, v, has a unique path to the root.

parent(v) is the first node along this path.

In a general tree, any node can be made the root.

This is called “rooting”



# Testing Connectedness

Input: A graph G, and vertices s,t.

Output: A path from s to t, if one exists, and

otherwise output “Disconnected”

How do we proceed?

- First issue: How do we store a graph in the computer?

# Storing a Graph

2 main approaches:

(a) Adjacency list representation. (Better)

(b) Adjacency matrix. (Worse)

# Adjacency List repn

Graph:

int N = how many vertices there are.

Adj[ v ] = A List of the neighbors of v.

So: we have an Array of Linked Lists.

Example: V = {1,2,3,4,5},
E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4}, {1,3} }.

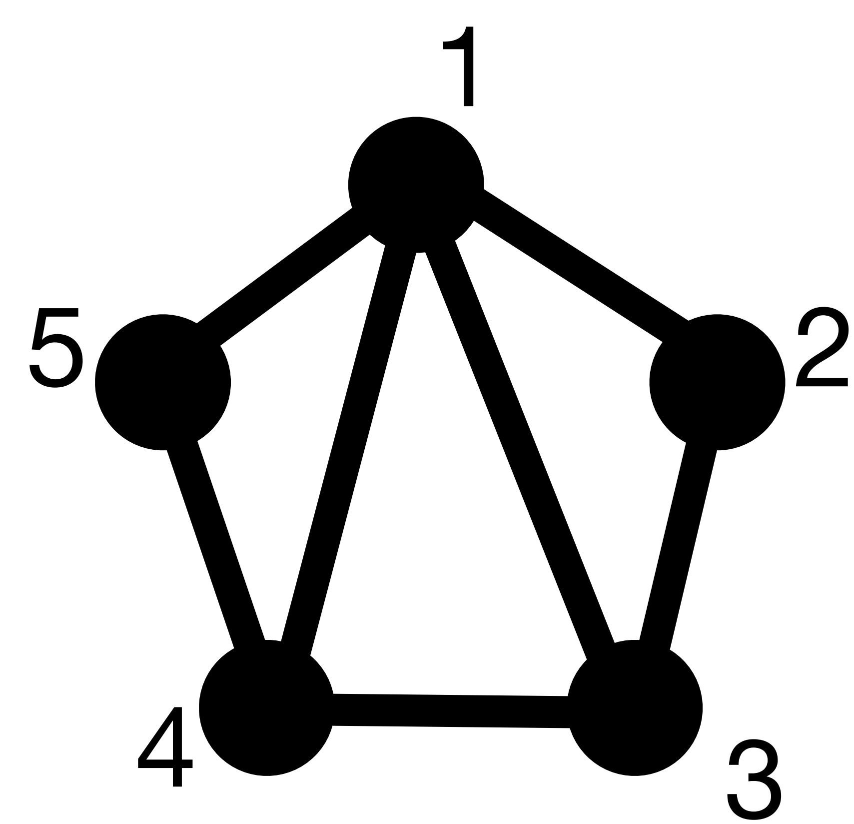
(5 vertices, 7 edges).

# Adjacency List repn

Example: V = {1,2,3,4,5},
E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4}, {1,3} }.

(5 vertices, 7 edges).

Adj[1] = {2, 5, 4, 3}
Adj[2] = {1, 3}
Adj[3] = {2, 4, 1}
Adj[4] = {3, 5, 1}
Adj[5] = {4, 1}



# Adjacency Matrix repn

Graph:

int N = how many vertices there are.

A = n x n matrix of 0's and 1's

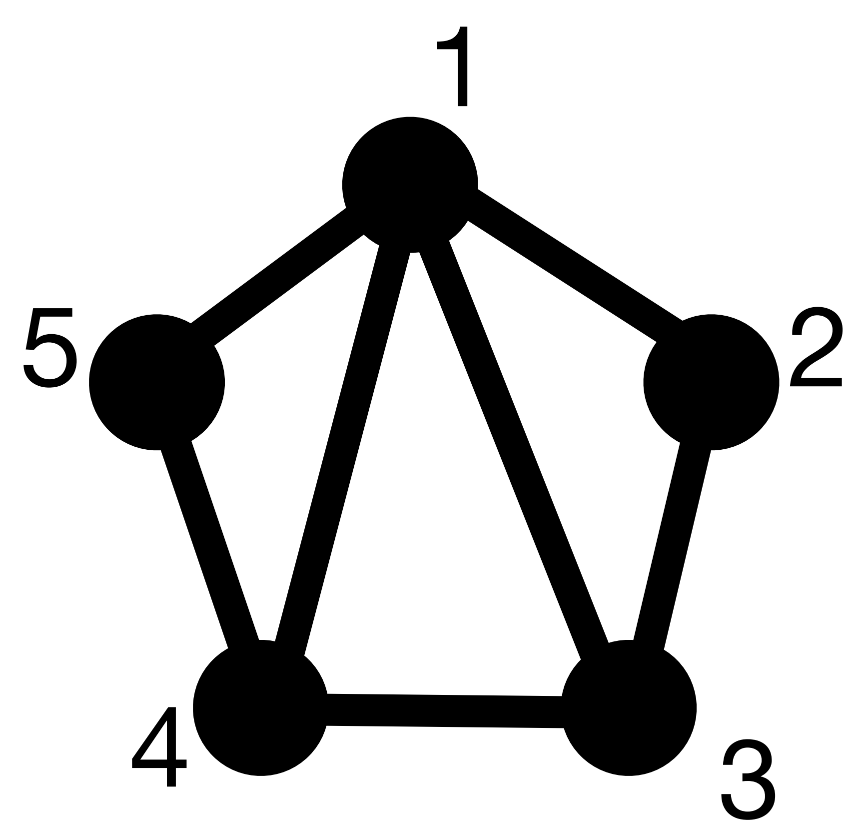
A[i,j] = 1 means the edge {i, j} is included.

# Adjacency List repn

Example: V = {1,2,3,4,5},
E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4}, {1,3} }.

(5 vertices, 7 edges).

A = 0 1 1 1 1
1 0 1 0 0
1 1 0 1 0
1 0 1 0 1
1 0 0 1 0



# Adjacency List repn

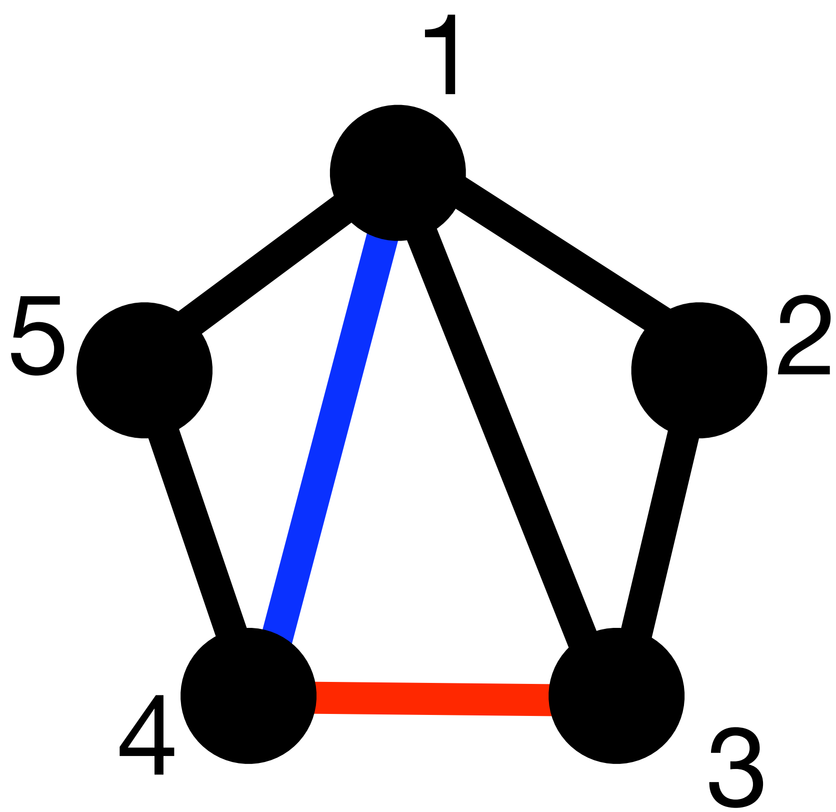
Example: V = {1,2,3,4,5},

E = { {1,2}, {2,3}, {3,4}, {4,5}, {5,1}, {1,4}, {1,3} }.

(5 vertices, 7 edges).

A =

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |



# Adjacency Matrix repn

Graph:

- int N = how many vertices there are.
- A = n x n matrix of 0's and 1's
- A[i,j] = 1 means the edge {i, j} is included.

Why is this worse than Adjacency List (in general)?

# Adjacency Matrix repn

Graph:

int n = how many vertices there are.

A = n x n matrix of 0's and 1's

A[i,j] = 1 means the edge {i, j} is included.

Why is this worse than Adjacency List (in general)?

Always requires n^2 space (and n^2 time to read/write it). So what? How many edges can there be?

# Adjacency Matrix repn

Graph:

int n = how many vertices there are.

A = n x n matrix of 0's and 1's

A[i,j] = 1 means the edge {i, j} is included.

Why is this worse than Adjacency List (in general)?

Always requires n^2 space (and n^2 time to read/write it). So what? How many edges can there be? There can be

$$\binom{n}{2} = \Theta(n^2)$$

# Adjacency Matrix repn

Graph:

int n = how many vertices there are.

A = n x n matrix of 0's and 1's

A[i,j] = 1 means the edge {i, j} is included.

Why is this worse than Adjacency List (in general)?

Always requires n^2 space (and n^2 time to read/ write it). So what? How many edges can there be? There can be $\binom{n}{2} = \Theta(n^2)$

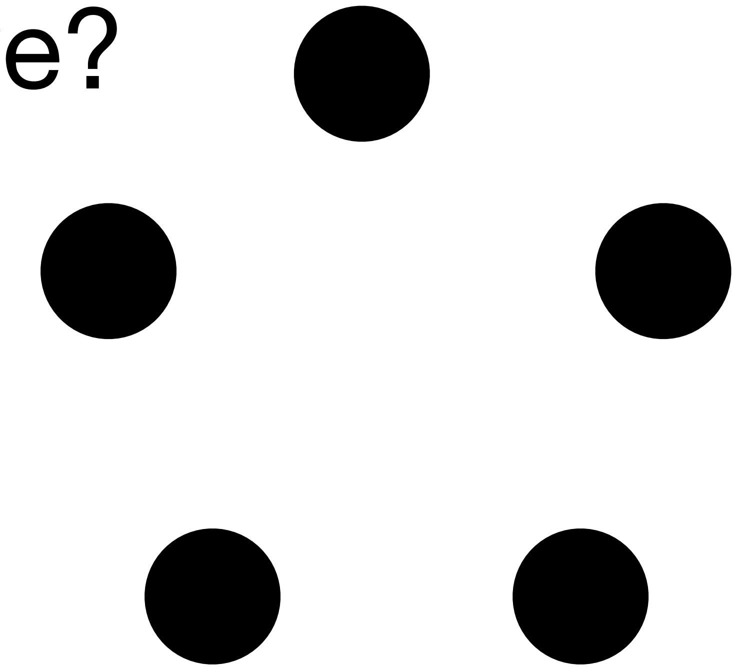
# How many edges?

Suppose G is a graph, with N vertices. What is the fewest edges G can have?

# How many edges?

Suppose G is a graph, with N vertices.  What is the fewest edges G can have?

Zero.



# How many edges?

Suppose G is a graph, with N vertices.  What is the fewest edges G can have?

Zero.

Suppose G is a connected graph with N vertices. What is the fewest edges G can have?

# How many edges?

Suppose G is a graph, with N vertices. What is the fewest edges G can have?

Zero.

Suppose G is a connected graph with N vertices. What is the fewest edges G can have?

N-1. Proof?

# How many edges?

Suppose G is a graph, with N vertices. What is the fewest edges G can have?

Zero.

Suppose G is a connected graph with N vertices. What is the fewest edges G can have?

N-1. Proof? Induction: Start with empty graph. Then there are N connected components. Each edge we add can reduce the number of components by 0 or by 1. So it takes at least N-1 edges to make G connected.

# How many edges?

Suppose G is a graph, with N vertices. What is the fewest edges G can have?

Zero.

Suppose G is a connected graph with N vertices. What is the fewest edges G can have?

N-1. In this case, G is always a tree!

# How many edges?

Suppose G is a graph, with N vertices. What is the fewest edges G can have?

Zero.

Suppose G is a connected graph with N vertices. What is the fewest edges G can have?

N-1. In this case, G is always a tree!

What is the most edges G can have?

$$\binom{N}{2} = \frac{N(N-1)}{2} = \Theta(N^2)$$

# Testing Connectivity

Input: A graph G, and vertices s,t.

Output: A path from s to t, if one exists, and otherwise output “Disconnected”

How do we proceed?

- Start at s, and “search outward”

# Testing Connectivity

Input: A graph G, and vertices s,t.

Output: A path from s to t, if one exists, and

otherwise output "Disconnected"

How do we proceed?

- Start at s, and "search outward"
- Build up a tree, rooted at s, as we go.
- Eventually, we will find all nodes in the component of s. If t is there, the path from t to s is t, parent(t), parent(parent(t)), ..., s