Algorithm Design and Analysis

CSE 565

LECTURE 3

Data Structures

Graphs

- Traversals
- Strongly connected components

Sofya Raskhodnikova

Measuring Running Time

- Focus on scalability: parameterize the running time by some measure of "size"
  - (e.g. n = number of men and women)
- Kinds of analysis
  - Worst-case
  - Average-case (requires knowing the distribution)
  - Best-case (how meaningful?)
- Exact times depend on computer; instead measure asymptotic growth

Computational Model

Unless explicitly stated otherwise

- All numbers and pointers fit into a single word (block) of memory
- Constant-time operations
  - Operations on words: arithmetic op's, shifts, comparisons, etc
  - Following a pointer
  - Array lookup

Ignore cache, virtual memory, pretend everything fits in RAM

We will sometimes drop these assumptions

- E.g.: for numerical problems, we might count bit operations

Data structures

Data Structures vs Abstract Data Types

- Data structure: concrete representation of data
  - Array
  - Linked list implemented with pointers
  - Binary heap in array
  - Adjacency list representation
- Abstract Data Type (ADT) : set of operations and their semantics (meaning/behavior)
  - Priority queue
  - Stack, queue
  - Graph
  - Dictionary

Basic Data Structures

- Lists
  - O(1) time: Insert/delete anywhere we have a pointer
- Array
  - O(1) time: append, lookup

Good for

- Stack ADT: Last in, First out (LIFO)
  - O(1) time: Push, pop
- Queue ADT: First in, First out (FIFO)
  - O(1) time: enqueue, dequeue

Dictionary ADT

- Dictionary: Set of (key,value) pairs.
- Operations on dictionary S
  - S.Insert(key, value)
  - S.Find(key)
  - S.delete(key)

  (Definitions of how to handle repeated keys vary.)

Dictionary Data Structures

Data Struct.	Find	Insert	Delete (after Find)
Unsorted array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted array	$\Theta(\log(\mathrm{n}))$	$\Theta(\mathrm{n})$	$\Theta(\mathrm{n})$
Binary search tree	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$
Balanced binary search tree	$\Theta(\log(\mathrm{n}))$	$\Theta(\log(\mathrm{n}))$	$\Theta(\log(\mathrm{n}))$
Hash table (**expected** time over the choice of hash function; **worst case** over data)	1	1	1

Here n = # of items currently in dictionary.
Table entries are worst-case asymptotic running times.

Priority Queue ADT

- Set of (key, value) pairs
  - *Values* are unique, keys are not
- Operations
  - Q.Insert(k,v)
  - Q.Changekey(v, k_{new})
  - Q.Extract-min()
- Often implemented as a binary heap
  - KT Chapter 2.4

Exercise

- How can you simulate an array with two unbounded stacks and a small amount of memory?
  - (Hint: think of a tape machine with two reels)

- What if you only have one stack and constant memory? Can you still simulate arbitrary access to an array?
  - (Hint: think about pushdown automata.)

Graphs

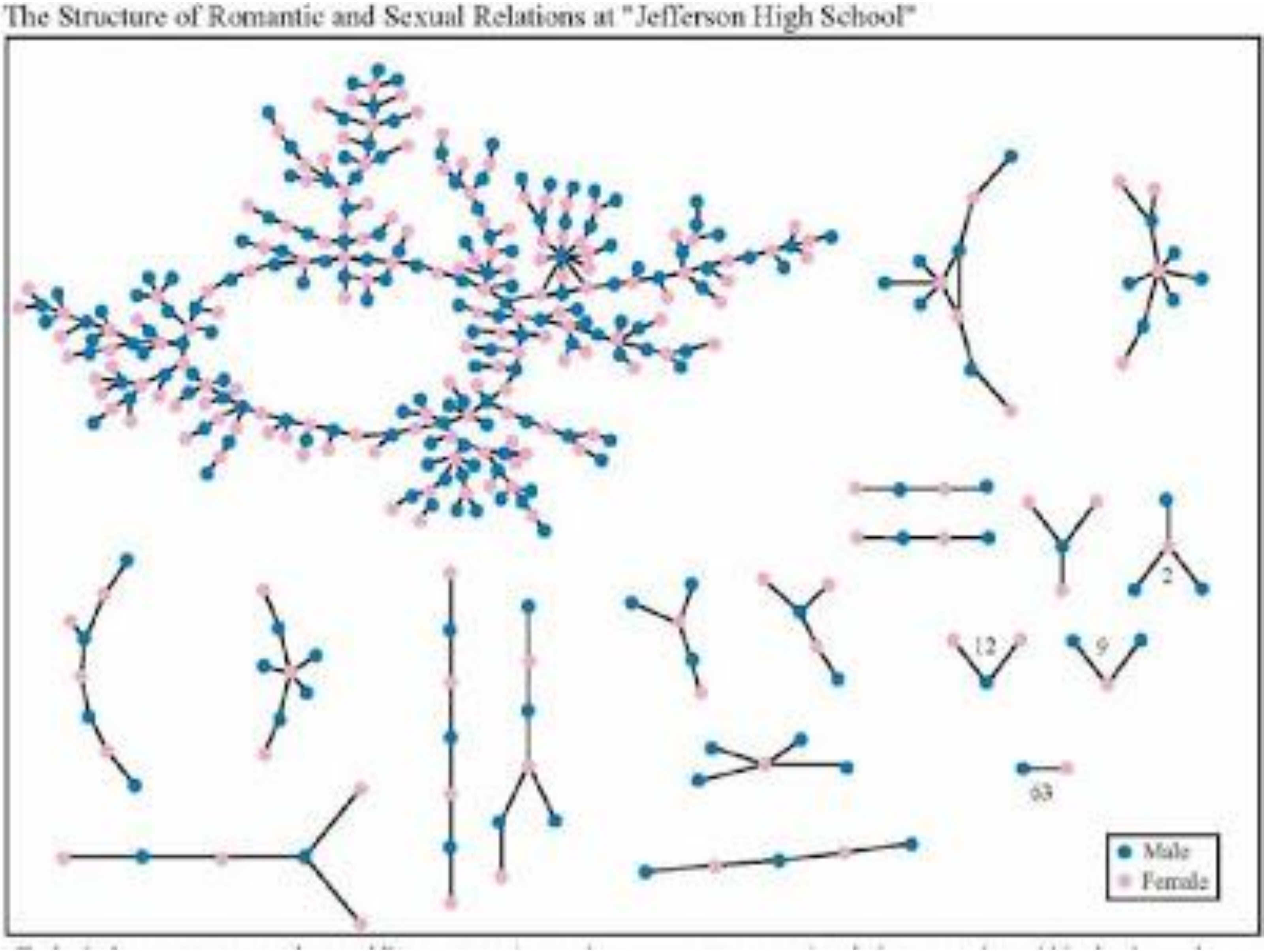
Graphs (KT Chapter 3)

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (synonym: ***nodes***),
- a set $E \subseteq V \times V$ of ***edges***
- An edge $e=(u,v)$ goes "from u to v" (may or may not allow $u=v$)

- In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices
  - Sometimes write $e=\{u,v\}$
- How many edges can a graph have?
  - In either case, $|E| = O(|V|^2)$.

Graphs are everywhere

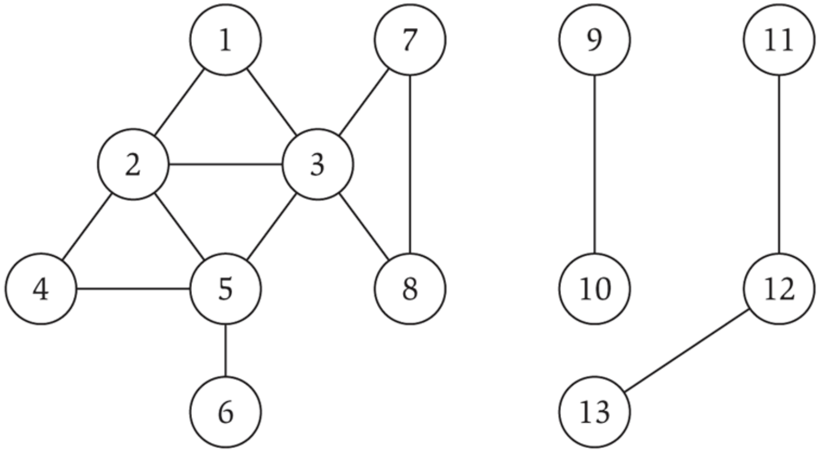


Graphs are everywhere

Example	Nodes	Edges
Transportation network: airline routes	airports	nonstop flights
Communication networks	computers, hubs, routers	physical wires
Information network: web	pages	hyperlinks
Information network: scientific papers	articles	references
Social networks	people	"u is v's friend", "u sends email to v", "u's MySpace page links to v"

Paths and Connectivity

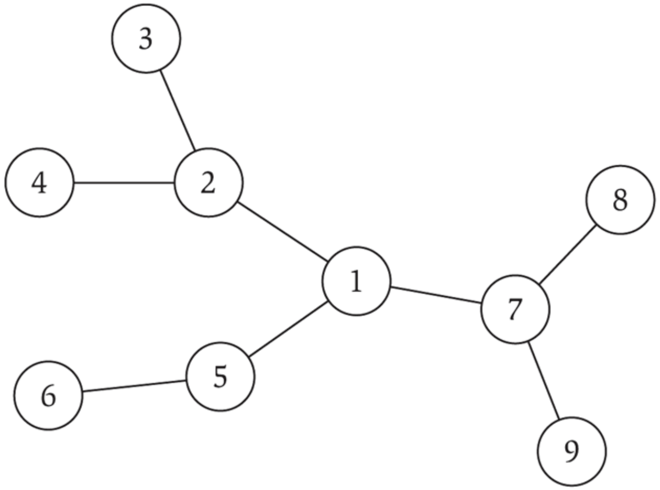
- **Path** = sequence of consecutive edges in E
  - $(u,w_1), (w_1,w_2), (w_2,w_3), \ldots, (w_{k-1}, v)$
  - Write $u \leftrightsquigarrow v$ or $u \rightsquigarrow v$
  - (Note: in a directed graph, direction matters)
- Undirected graph G is **connected** if for every two vertices u,v, there is a path from u to v in G



Trees

- **Def.** An undirected graph is a tree if it is connected and does not contain a cycle.

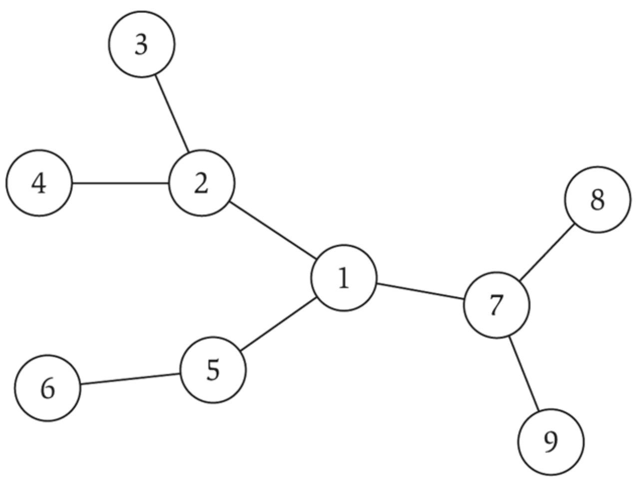
- **Theorem**. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.



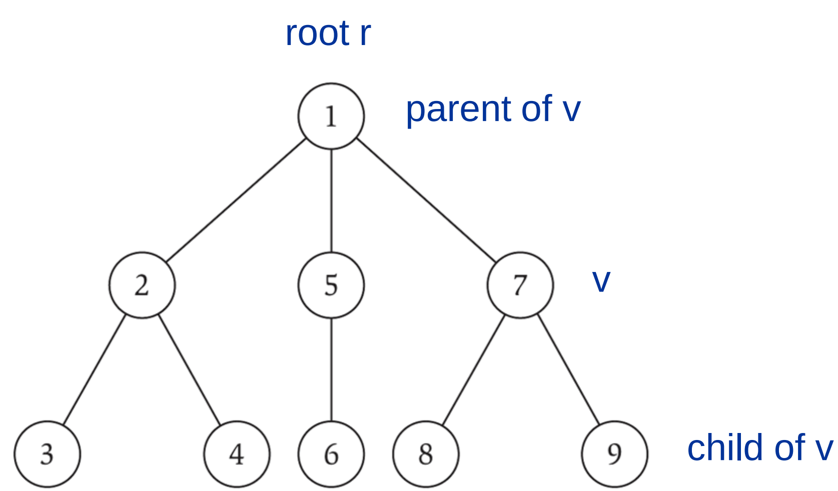
Rooted Trees

- Rooted tree: Given a tree T, choose a root node ***r*** and orient each edge away from ***r***.

- Models hierarchical structure.



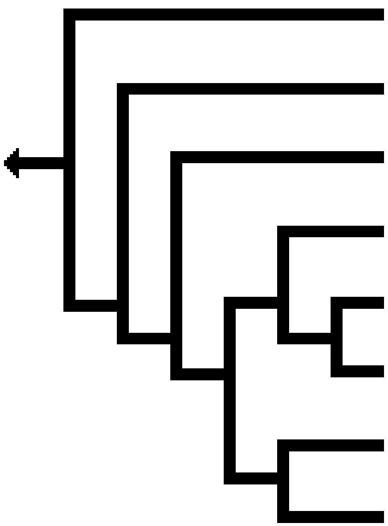
a tree



the same tree, rooted at 1

Phylogeny Trees

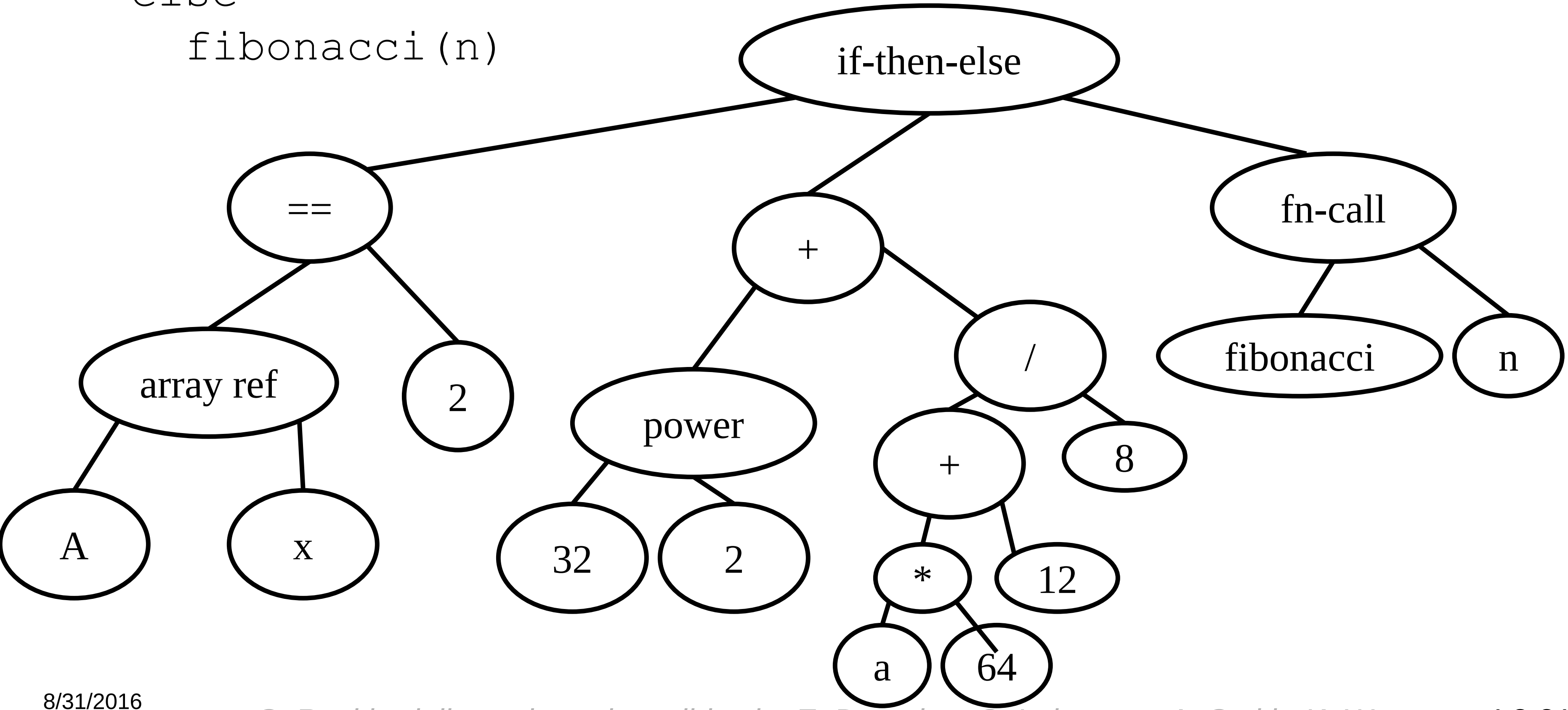
- Phylogeny trees. Describe evolutionary history of species.



Parse Trees

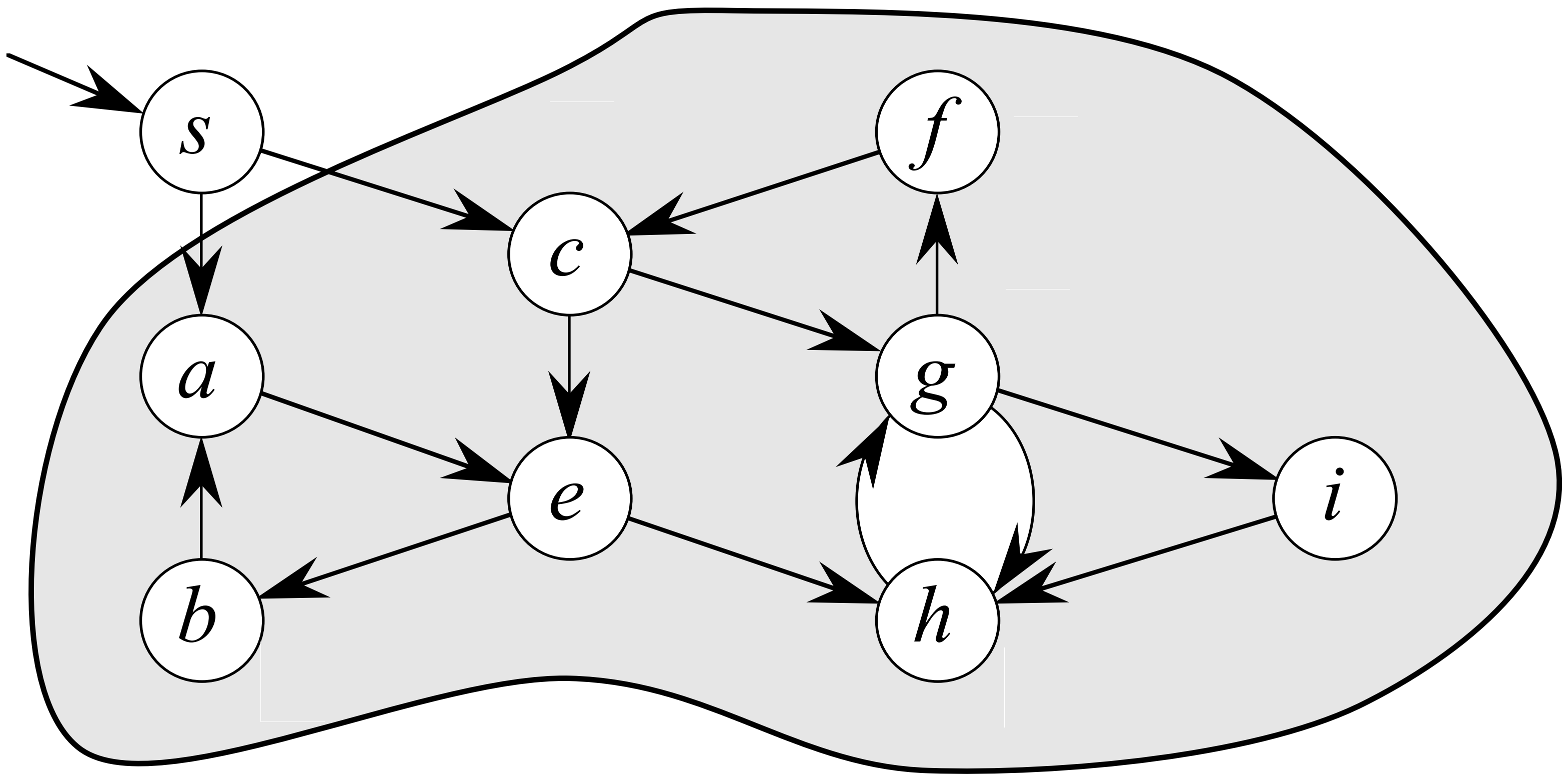
- Internal representation used by compiler, e.g.:

```
if (A[x]==2) then
  (32² + (a*64 +12)/8)
else
  fibonacci(n)
```



Paths and Connectivity

- Directed graph?
  - **Strongly connected** if for every pair, $u \rightsquigarrow v$ and $v \rightsquigarrow u$



Exploring a graph

Classic problem: Given vertices $s,t \in V$, is there a path from s to t?

Idea: explore all vertices reachable from s

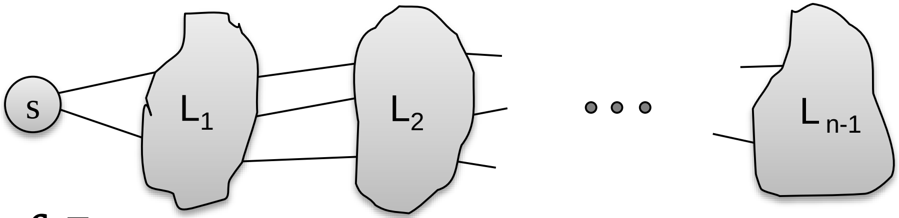
Two basic techniques:

How to convert these descriptions to precise algorithms?

- Breadth-first search (BFS)
  - Explore children in order of **distance** to start node
- Depth-first search (DFS)
  - Recursively explore vertex's children before exploring siblings

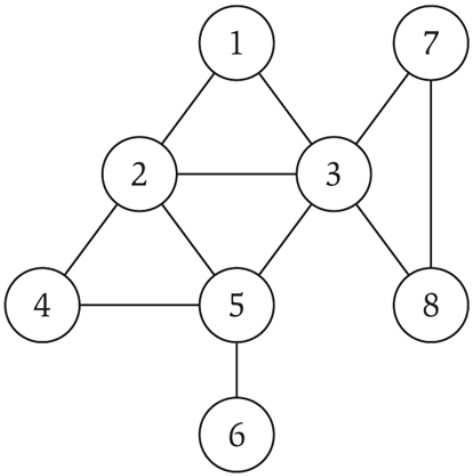
Breadth First Search

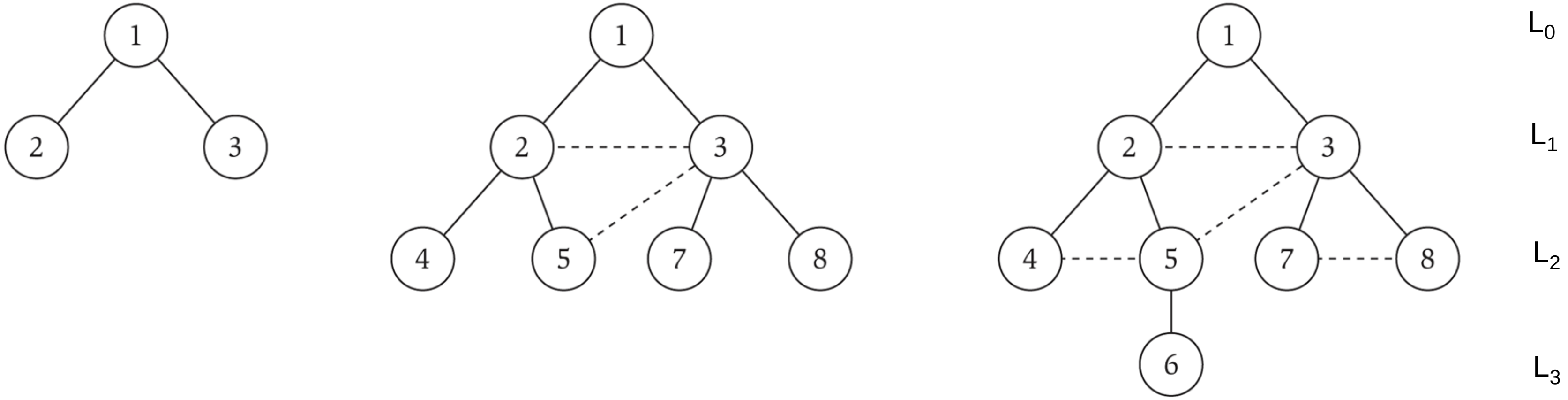
- BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.
- BFS algorithm.



  - $L_0 = \{ s \}$.
  - L_1 = all neighbors of L_0.
  - L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
  - L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Breadth First Search

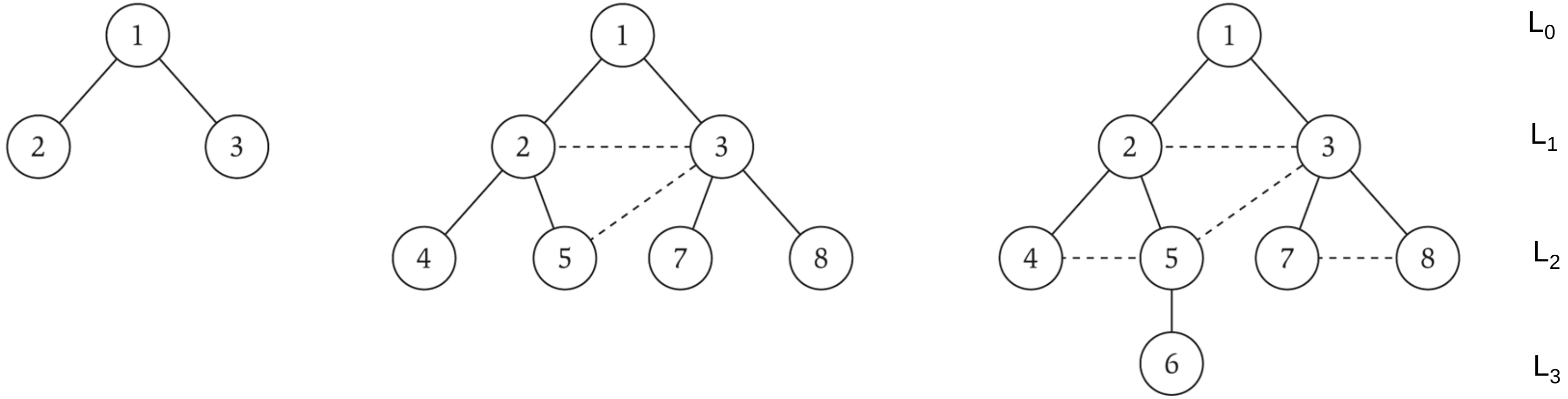




(a) (b) (c)

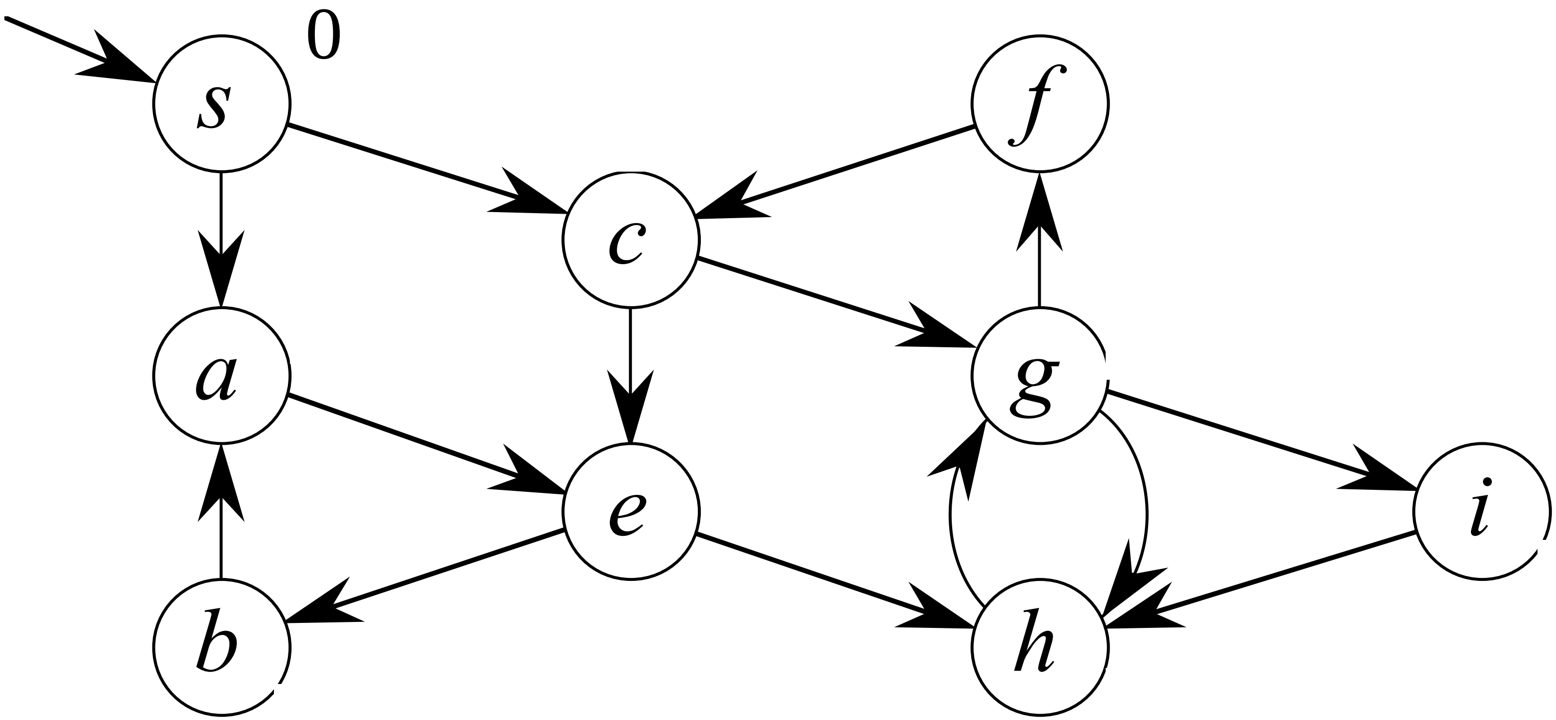
Breadth First Search

- Distance(u, v): number of edges on shortest path from u to v
- Properties. Let T be a BFS tree of G = (V, E).
  - Nodes in layer ***i*** have distance ***i*** from root ***s***
  - Let (x, y) be an edge of G. Then the levels of x and y differ by at most 1.



(a) (b) (c)

BFS example (directed)



Implementing Traversals

Generic traversal algorithm

1. $R = \{s\}$
2. **While** there is an edge (u,v) where $u \in R$ and $v \notin R$,
   - Add v to R

To implement this, need to choose…

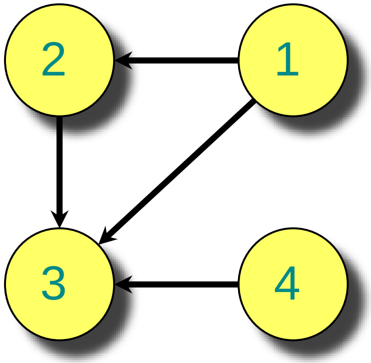
- Graph representation
- Data structures to track…
  - Vertices already explored
  - Edge to be followed next

These choices affect the order of traversal

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



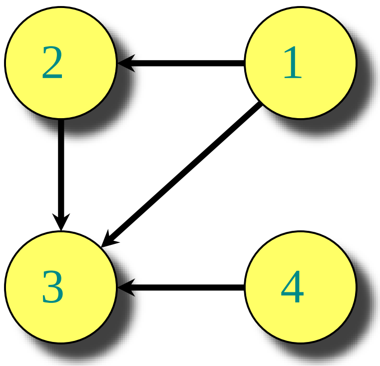
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Storage: $\Theta(V^2)$
Good for dense graphs.

- Lookup: $O(1)$ time
- List all neighbors: $O(|V|)$

Adjacency list representation

- An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

Typical notation:
$n = |V|$ = # vertices
$m = |E|$ = # edges

Storage: $\Theta(V+E)$
Good for sparse graphs.

For undirected graphs, $|\ Adj[v]\ | = degree(v)$.
For digraphs, $|\ Adj[v]\ | = out\text{-}degree(v)$.

How many entries in lists? $2|E|$
Total $\Theta(V + E)$ storage.

- List all neighbors: O(degree) time
- Lookup(u,v): O(min(degree(u),degree(v)) time

Other representations?

- Can we get
  - $O(1)$ lookup /insertion/deletion
  - $O(degree(v))$ list all neighbors of v
  - $O(V + E)$ storage?

- (Hint: hash tables)

BFS with adjacency lists

- d[1 .. n]: array of integers
  - initialized to infinity
  - use to track distance from root (infinity = vertex not yet explored)
- Queue Q
  - initialized to empty
- Tree T
  - initialized to empty

BFS pseudocode

BFS(s):

1. Set d[s]=0
2. Add s to Q

O(1) time, run once overall.

3. While (Q not empty)
   a) Dequeue (u)

O(1) time, run once per vertex

   b) For each edge (u,v) adjacent to u
      a) If d[v] == ∞ then
         a) Set d[v] =d[u]+1
         b) Add edge (u,v) to tree T
         c) Enqueue v onto Q

O(1) time per execution, run at most twice per edge

Total: O(m+n) time (linear in input size)

Notes

- If s is the root of BFS tree,
- For every vertex u,
  - path in BFS tree from s to u is a shortest path in G
  - depth in BFS tree = distance from u to s
- Proof of BFS correctness: see KT, Chapter 3.

BFS Review

- Recall: Digraph G is s**trongly connected** if for every pair of vertices, $s \leadsto t$ and $t \leadsto s$
- **Question**: Give an algorithm for determining if a graph is strongly connected. What is the running time?