Colorado State University

CS 320
Algorithms: Theory and Practice
Wk 4: Graphs

Sanjay Rajopadhye

Colorado State University

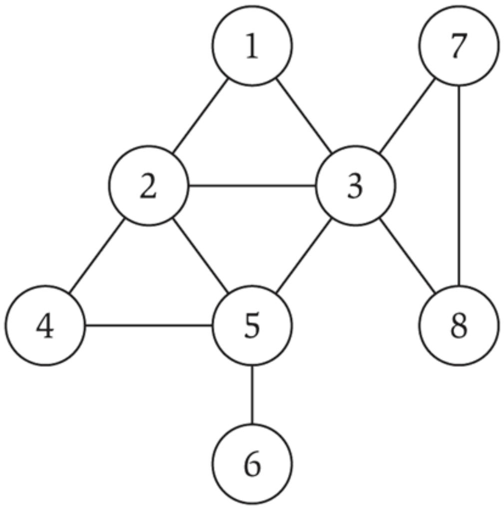
Sept 2023

Topics (CLRS Ch 22, pp 589-623)

- Representation
- Breadth First Search/Depth First Search
- Connected components
- Cycles
- Bipartite graphs (testing)
- (Strongly) connected components
- Topological Sort

Undirected Graphs G = (V, E)

- V = set of nodes.
- E = set of edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

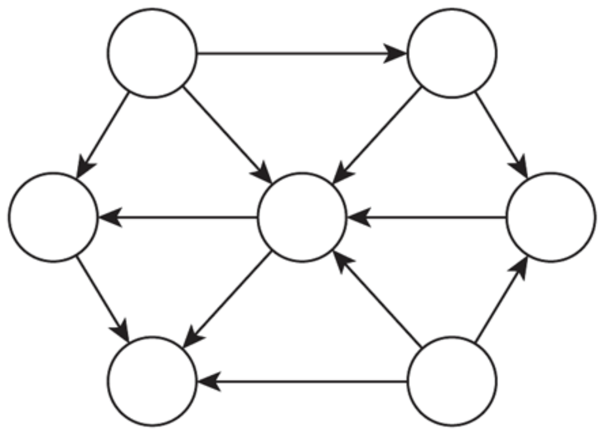
n = 8

m = 11

- What is the maximum possible value for $|E|$?

Directed Graphs

- Directed graph. G = (V, E)
  - Edge (u, v) goes from node u to node v.
  - Maximum number?



- Example. Web graph - hyperlink points from one web page to another.
  - Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph definitions

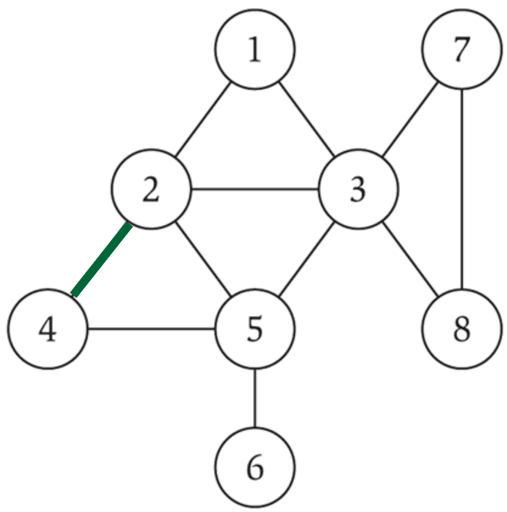
- Graph *$G = (V, E)$*, V: set of *nodes* or *vertices*,
- E: set of *edges* (pairs of nodes).
- In an *undirected* graph, edges are unordered pairs (sets) of nodes. In a *directed* graph edges are ordered pairs (tuples) of nodes.
- *Path*: sequence of nodes $(v_0..v_n)$ s.t. $\forall i: (v_i, v_{i+1})$ is an edge. *Path length*: number of edges in the path, or sum of weights. *Simple path:* all nodes distinct.
- *Cycle*: path with first and last node equal. *Acyclic graph*: graph without cycles. *DAG*: directed acyclic graph.
- Two nodes are *adjacent* if there is an edge between them. In a *complete graph* all nodes in the graph are adjacent.

more definitions

- An undirected graph is *connected* if for all nodes v_i and v_j there is a path from v_i to v_j . An undirected graph can be partitioned in *connected components*: maximal connected sub-graphs.
- A directed graph can be partitioned in *strongly connected components*: maximal sub-graphs C where for every u and v in C there is a path from u to v and there is a path from v to u.
- G'(V', E') is a *sub-graph* of G(V,E) if $V' \subseteq V$ and $E' \subseteq E$
- The sub-graph of G *induced* by V' has all the edges
- $(u,v) \in E$ such that $u \in V'$ and $v \in V'$.
- In a *weighted graph* the edges have a weight (cost, length,..) associated with them.

Graph representation: adjacency matrix

- Adjacency matrix. n-by-n matrix with A_{uv} = 1 if (u, v) is an edge, or weight_{uv} in a weighted graph.
  - For undirected graphs, each edge is represented *twice*.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all outgoing edges from a node takes $\theta(n)$

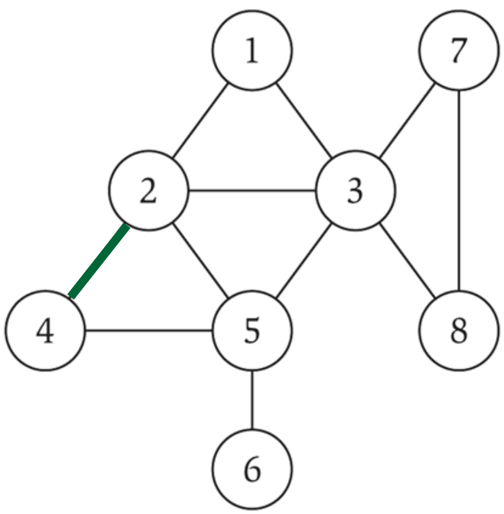


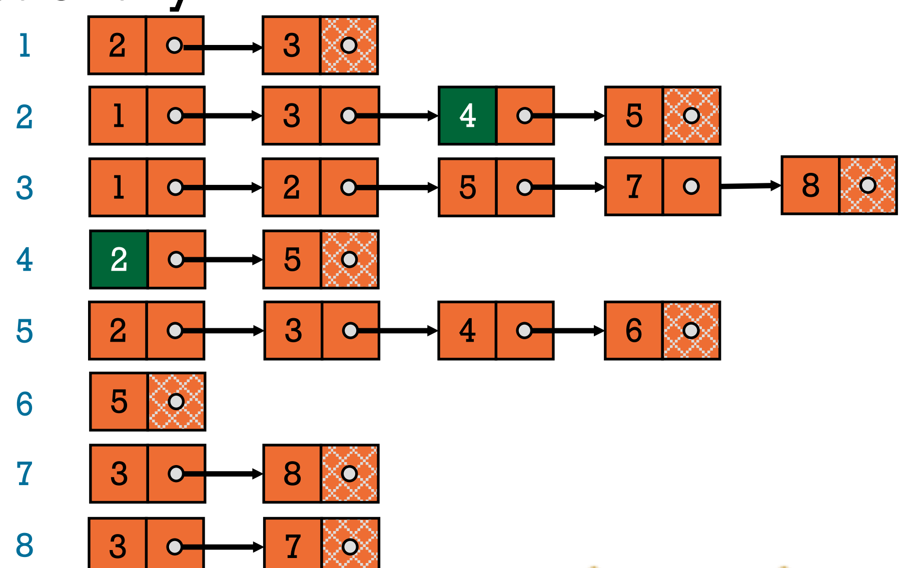
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph representation: adjacency list

Adjacency list. Node indexed array of lists.

- For undirected graphs, each edge is again represented twice.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(degree(u)) time.
- Identifying all outgoing edges from a node takes O(degree(u)) time
- Identifying all edges takes Θ(m + n) time.
- Cool python representation: dictionary





Which Implementation

- Which implementation best supports common graph operations:
  - Is there an edge between vertex i and vertex j?
  - Find all vertices adjacent to vertex j

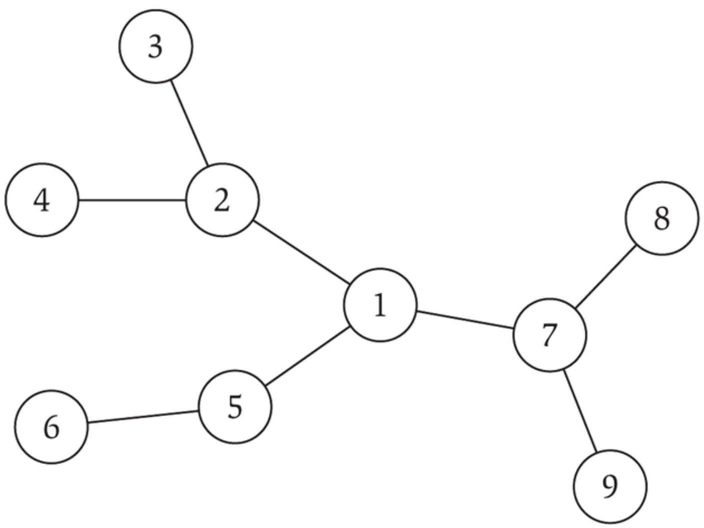
- Which best uses space?

Trees

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

How many edges does a tree have?

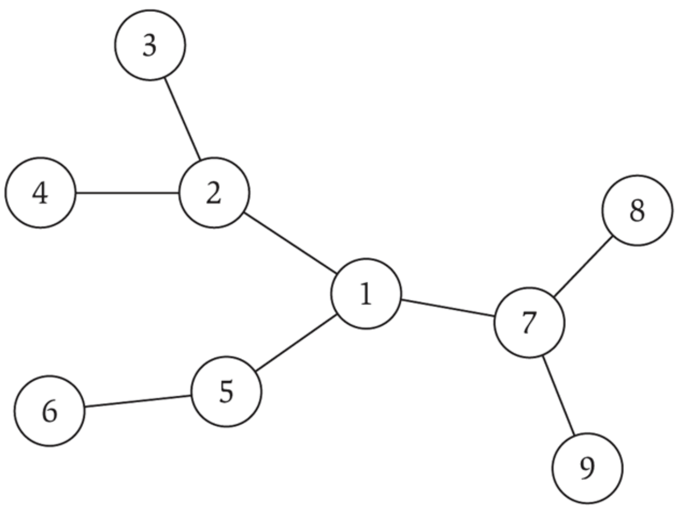
- Given a set of nodes, build a tree step wise
  - every time you add an edge, you must add a new node to the growing tree. WHY?
  - how many edges to connect n nodes?



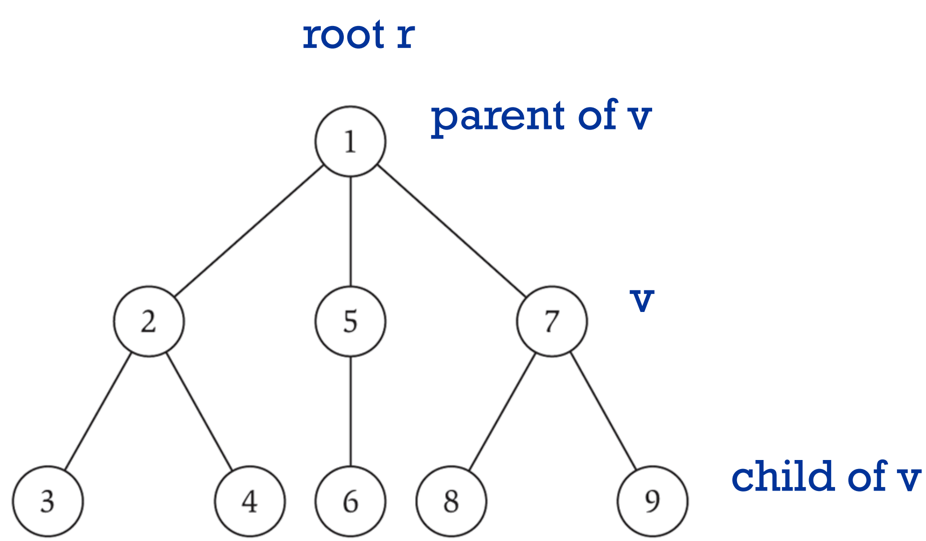
Colorado State University

Rooted Trees

- Rooted tree. Given a tree T, choose a root node r and orient each edge below r; do same for sub-trees.
- Models hierarchical structure. By rooting the tree it is easy to see that it has n-1 edges.



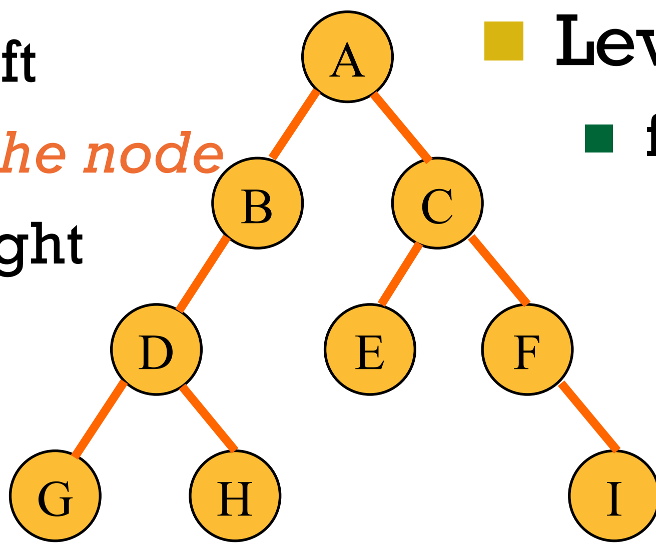
a tree



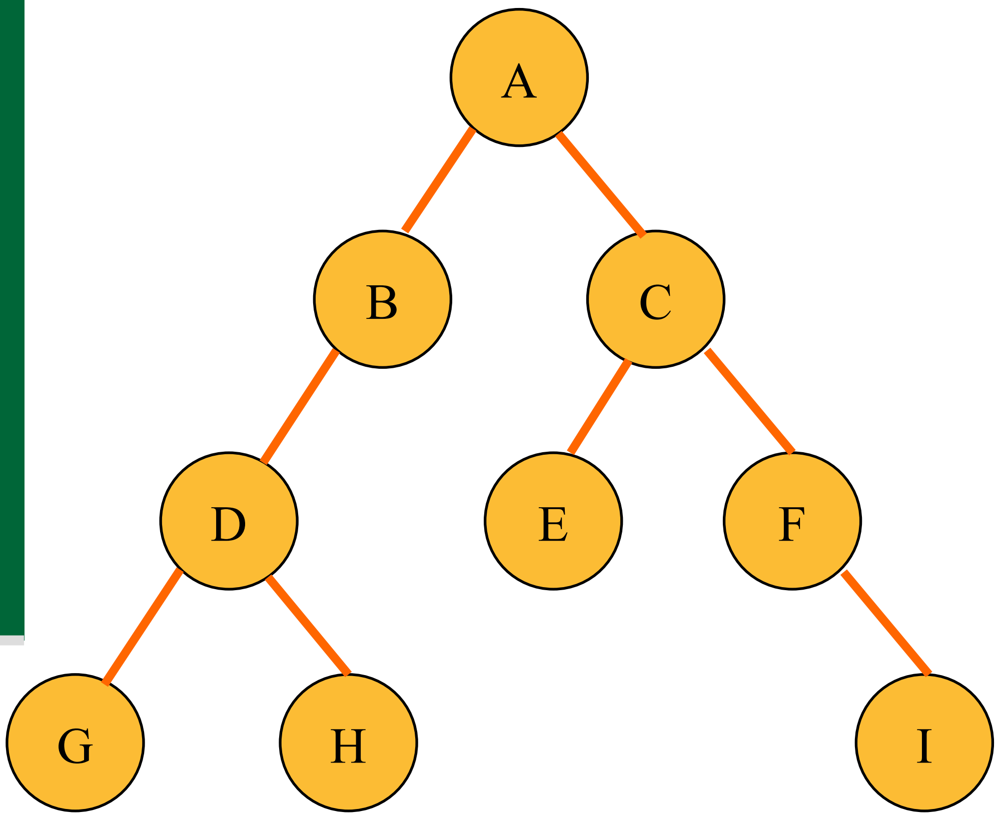
the same tree, rooted at 1

Traversing a Binary Tree

- Pre order
  - *visit the node*
  - go left
  - go right
- In order
  - go left
  - *visit the node*
  - go right
- Post order
  - go left
  - go right
  - *visit the node*
- Level order / breadth first
  - for d = 0 to height
    - visit nodes at level d



Traversal Examples



Pre order

A B D G H C E F I

In order

G D H B A E C F I

Post order

G H D B E I F C A

Level order

A B C D E F G H I

IMPLEMENTATION of these traversals??

Tree traversal Implementation

- recursive implementation of preorder
  - The steps:
    - visit node
    - preorder(left child)
    - preorder(right child)
  - What changes need to be made for in-order, post-order?
- How would you implement level order?

Tree traversal implementation

- Recursive implementation of preorder. The basic steps:
  - visit node
  - preorder (left child)
  - preorder (right child)
- What changes need to be made for in-order, post-order?
- How would you implement level order?

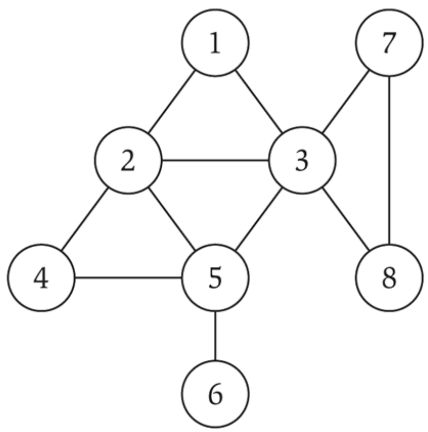
Tree traversal implementation

- Recursive implementation of preorder. The basic steps:
  - visit node
  - preorder (left child)
  - preorder (right child)
- What changes need to be made for in-order, post-order?
- How would you implement level order?

Connectivity

- s-t connectivity problem. Given two node s and t, is there a path between s and t?

- s-t shortest path problem. Given two nodes s and t, what is the length of the shortest path between s and t? Length: either in terms of number of edges, or sum of weights of the edges in the path



Colorado State University

Graph traversal

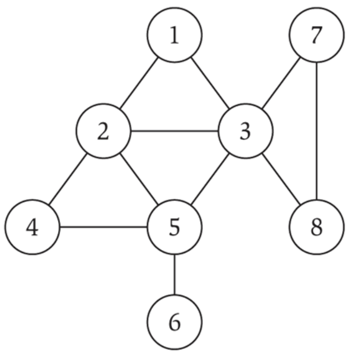
What makes it different from tree traversal

- You can visit the same node more than once
- You can get in a cycle
- What to do about it:
  - *Mark* the nodes

    -White: unvisited

    -Grey: (still being considered) on the frontier: not all adjacent nodes have been visited yet

    -Black: off the frontier: all adjacent nodes visited (not considered anymore)

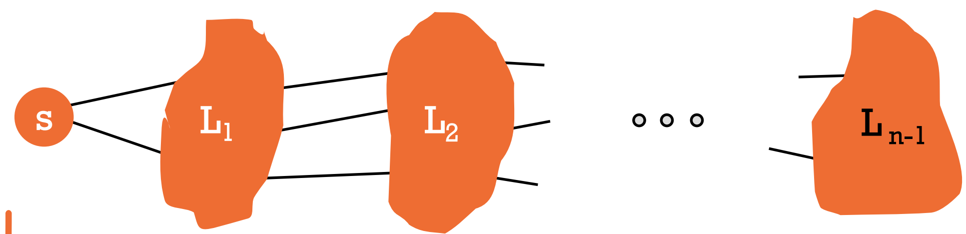


Breadth First Search (BFS)

- Like *level traversal* in trees BFS(G, s) explores the edges of G, and locates every node reachable from s in a *level order*, using a queue
- BFS also computes the *distance*: number of edges from s to all these nodes, and the *shortest path* (minimal #edges) from s to v.
- BFS expands a *frontier* of *discovered* but not yet visited nodes. Nodes are colored white, grey or black. They start out undiscovered or white.

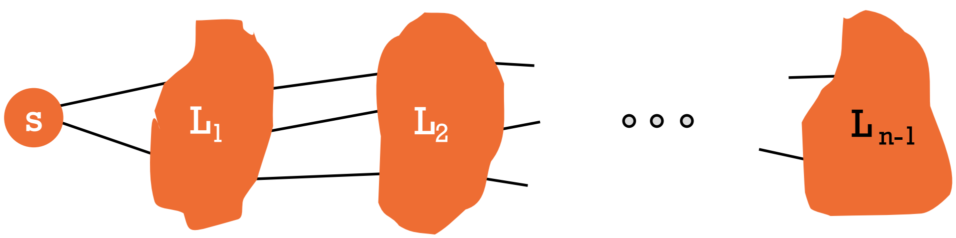
BFS intuition

- BFS intuition. Explore outward from s, adding nodes one "layer" at a time.
- BFS algorithm.
  - $L_0 = \{s\}$.
  - L_1 = all neighbors of L_0.
  - L_2 = all nodes not in 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.
- For each i, L_i consists of all nodes at distance exactly i from s. There is a path between s and t iff t appears in some layer.

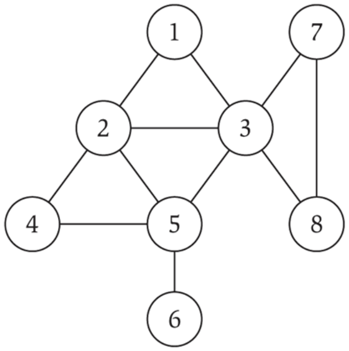


Breadth First tree

- BFS produces a *Breadth First Tree* rooted at s: when a node v in L_{i+1} is discovered as a neighbor of node u in L_i we add edge (u,v) to the BF tree

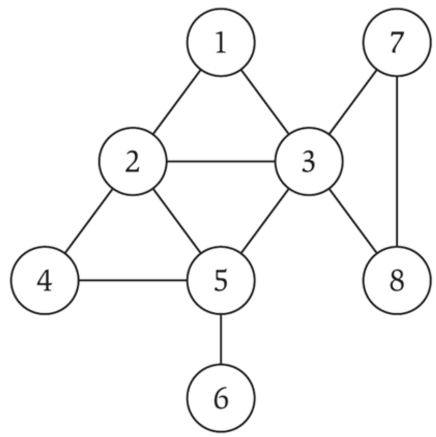


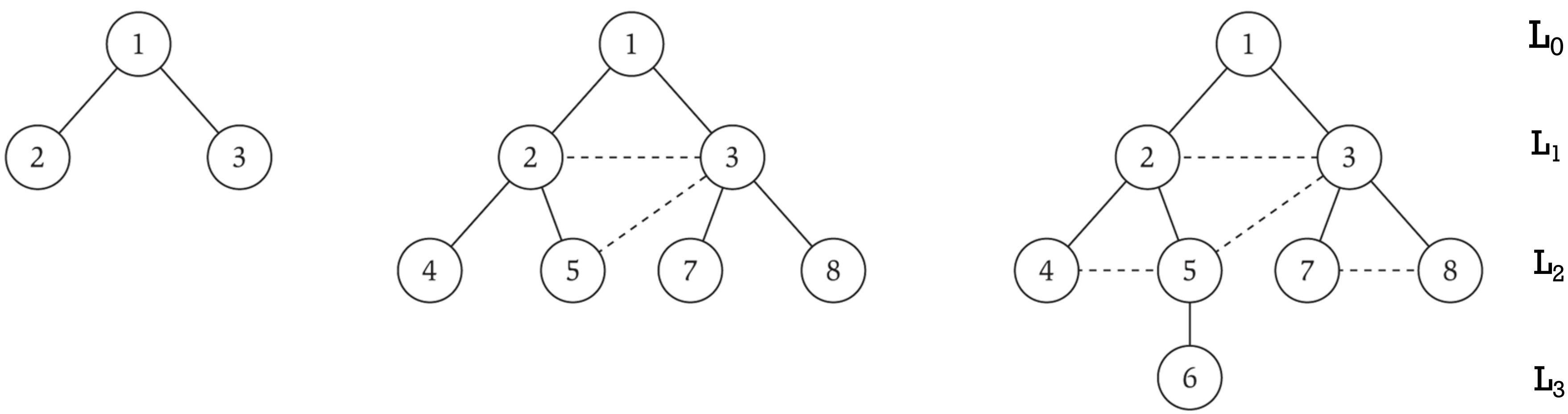
- Property. Let T be a BFS tree of G, and let (x,y) be an edge of G. Then the level of x and y differ by at most 1. *WHY?*
- *Either in the same layer (2,3) for root 1, or in two adjacent layers (2,4) for root 1.*



Colorado State University

Breadth First Search





(a) (b) (c)

Breadth First Search (BFS)

```
BFS(G,s)
  #d: distance,  c: color,  p: parent in BFS tree
  forall v in V-s {c[v]=white; d[v]=□,p[v]=nil}
  c[s]=grey; d[s]=0; p[s]=nil;
  Q=empty;
  enque(Q,s);
  while (Q != empty)
     u = deque(Q);
      forall  v in adj(u)
         if (c[v]==white)
            c[v]=grey; d[v]=d[u]+1; p[v]=u;
            enque(Q,v)
     c[u]=black;
   # don't really need grey here, why?
```

We don't use grey; we just test for unvisited (white) so we can paint v black (visited) immediately.

BFS complexity

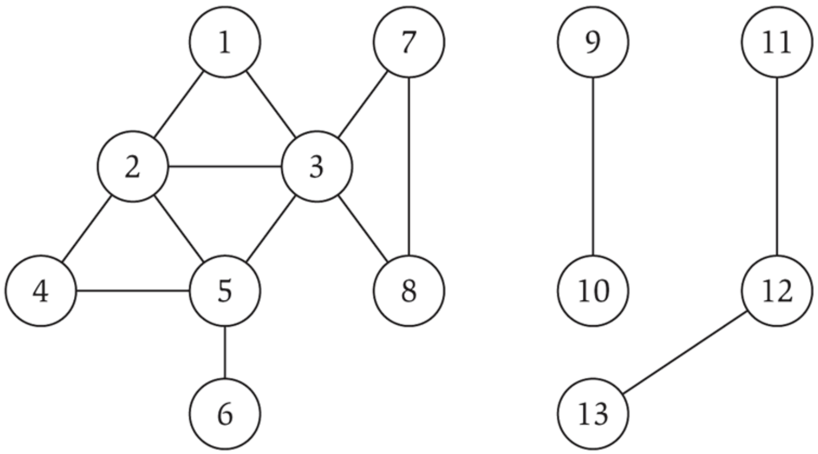
- Each node is painted *white* once, and is *enqueued* and *dequeued* at most once.
- *Why? Once a node is not white, we don't enqueue/ dequeue it anymore.*
- Enque and deque take constant time. The adjacency list of each node is scanned only once, when it is dequeued.

- Therefore time complexity for BFS is

$$O(|V| + |E|) \text{ or } O(n + m)$$

Connected components

- A graph is *connected* if there is a path between any two nodes
- The *connected component* of a node s is the set of all nodes reachable from s



- Connected component containing the node 1 is

$\{1, 2, 3, 4, 5, 6, 7, 8\}$

One graph with three connected components.

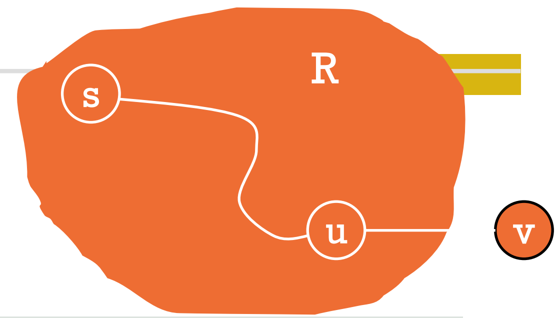
Connected components

- Given two nodes s and t, their connected components are either identical or disjoint

Proof: two cases: either there is a path between s and t or there isn't.

- If there is a path: take a node u in the connected component of s, and construct a path from t to u as follows: from t to s, and then from s to u, so $CC_s = CC_t$
- If there is no path: assume that the intersection contains a node u. Use it to construct a path between s and t as follows: from s to u, then u to t: this is a *contradiction*.

Connected components



- Generic algorithm for finding connected components

```
R = {s}  # connected component of s is initially s.
while there is an edge (u,v) where u is in R and v is not in R:
    add v to R
```

- Upon termination, R is the connected component containing s. Many variants, based on
  - BFS: explore in order of distance from s.
  - DFS: explores edges *from the most recently discovered node*; backtracks when reaching a dead-end.