Program Structure and Algorithms

Sid Nath

Lecture 6

Agenda

- Administrative
  - Quiz 4 next week on graphs

- Lecture

- Quiz

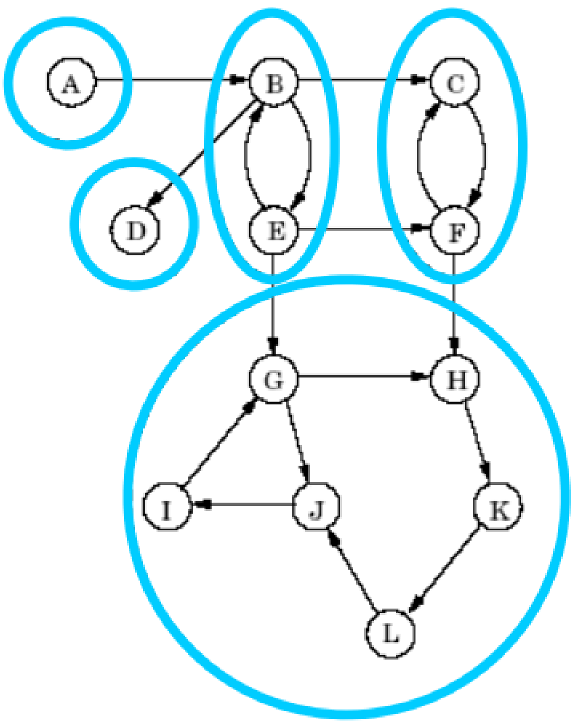
Strongly Connected Components

Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

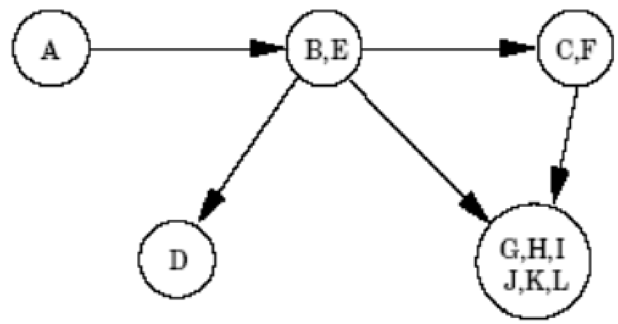
We partition V into strongly connected components

Metagraph

<u>Definition:</u> The <u>metagraph</u> of a directed graph G is a graph whose vertices are the SCCs of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.



SCCs in Graph



Metagraph

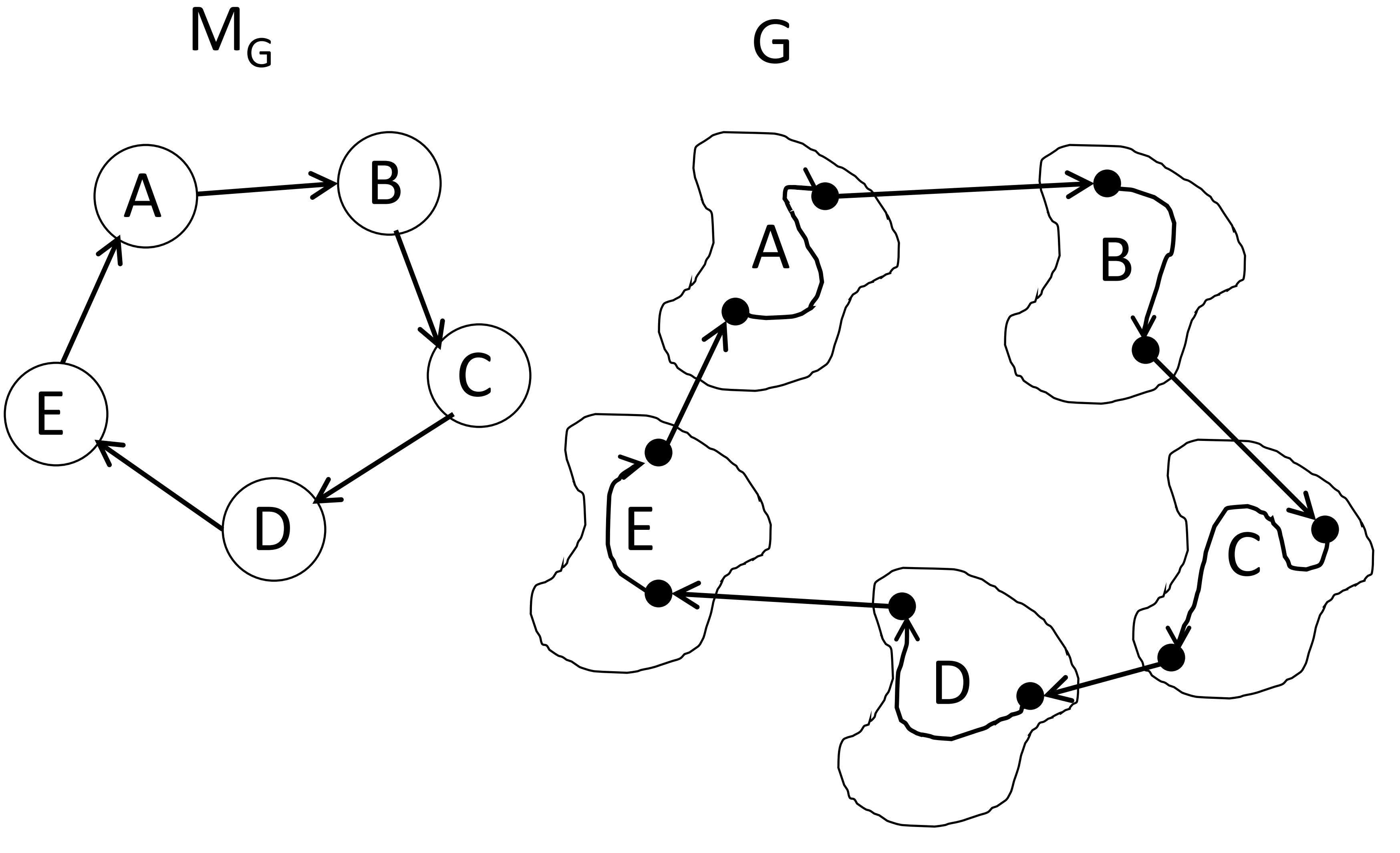
Result

Theorem: The metagraph of any directed graph is a DAG.

Proof (sketch):

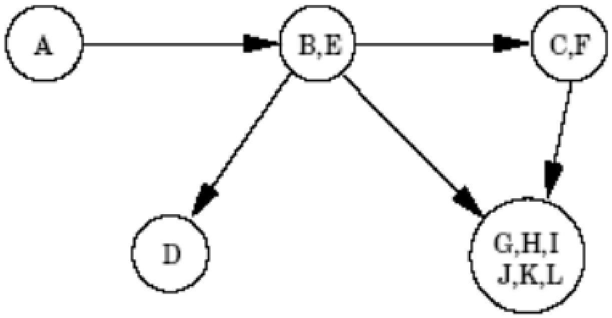
- Assume for sake of contradiction it is not.
- Then metagraph has a cycle.
- Use this to show that separated components should be connected.

Proof



Computing SCCs

Problem: Given a directed graph G compute the SCCs of G and its metagraph.

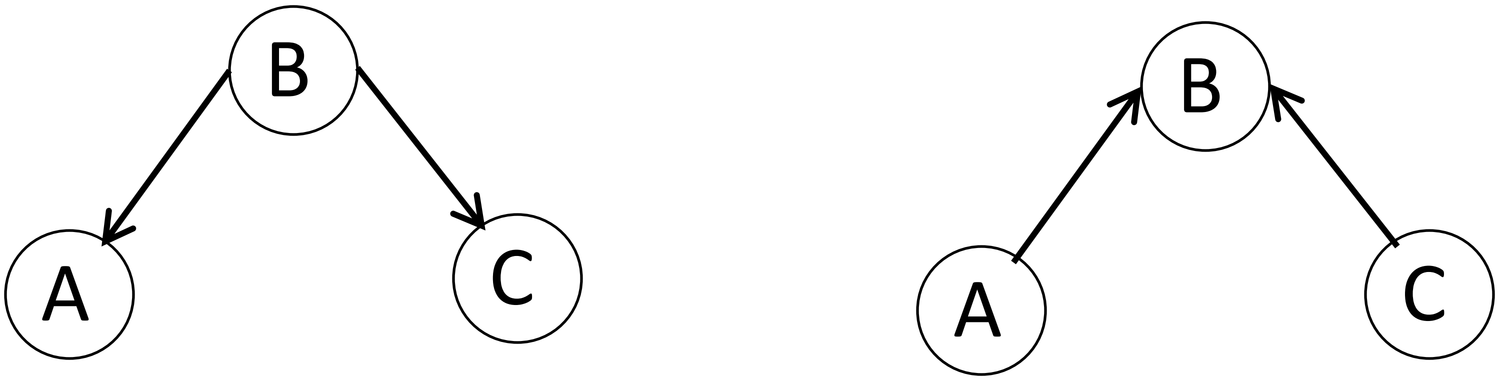


If we run explore() with a node in a sink SCC, then we will identify precisely that SCC

Idea: Find a node that is guaranteed to be in a sink SCC, and somehow continue

Reverse Graph

<u>Definition:</u> Given a directed graph G, the <u>reverse graph</u> of G (denoted G^R) is obtained by reversing the directions of all of the edges of G.



Other Properties of Reverse Graphs

Given a directed graph G and its reverse graph G^R:

- G and G^R have the *same* SCCs.
- The sink SCCs of G are the source SCCs of G^R.
- The source SCCs of G are the sink SCCs of G^R.

We had to find a node which is guaranteed to be a sink in G

So we can find a sink SCC of G, by finding a source SCC of G^R!

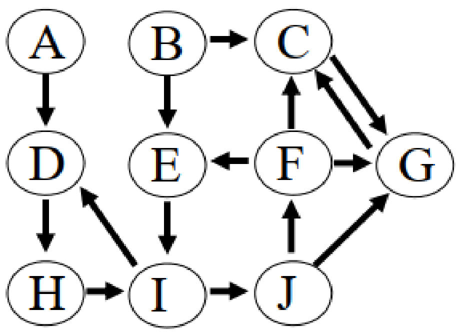
That is, the node with the highest post number in G^R is guaranteed to be a sink node in G

Algorithm

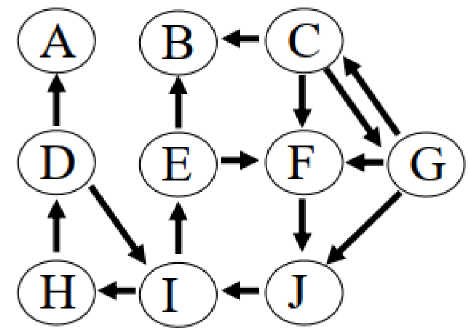
```
SCCs(G)
  Run DFS(G^R) record postorders
  Mark all vertices unvisited
  For v ∈ V in reverse postorder
  If not v.visited
     explore(v) mark component
```

Just 2 DFSs! Runtime $O(|V|+|E|)$.

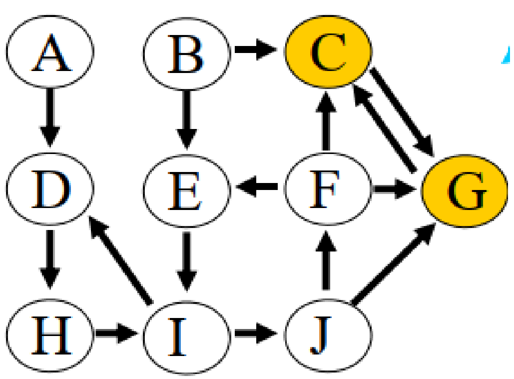
SCC Example



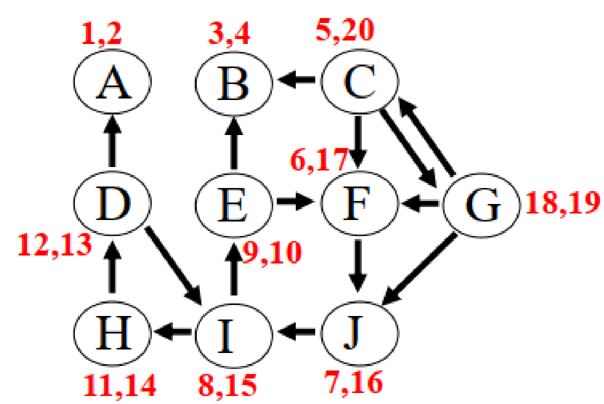
G



G^R

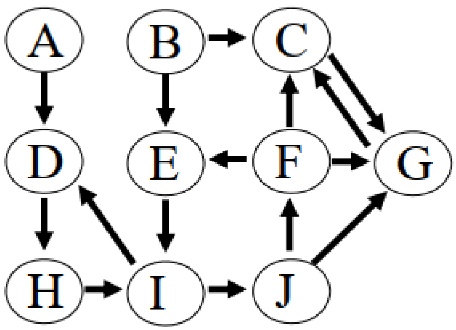


G

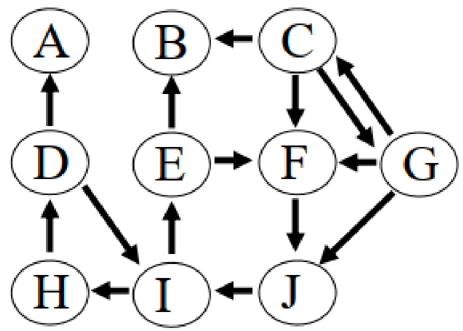


G^R

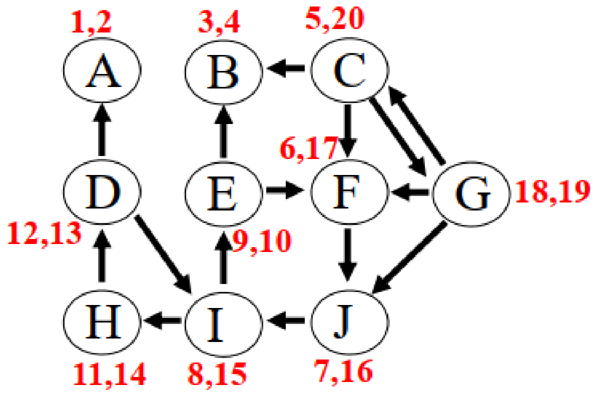
SCC Example



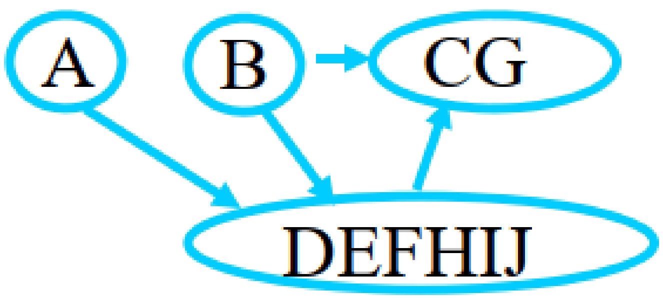
G

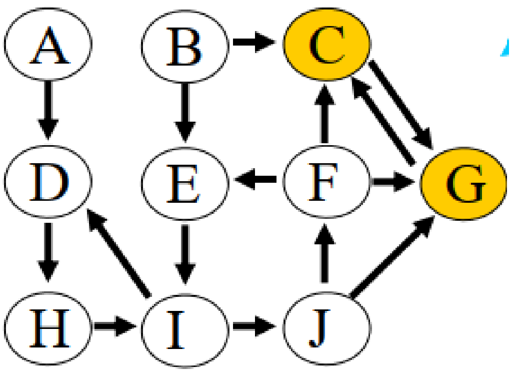


G^R



G^R

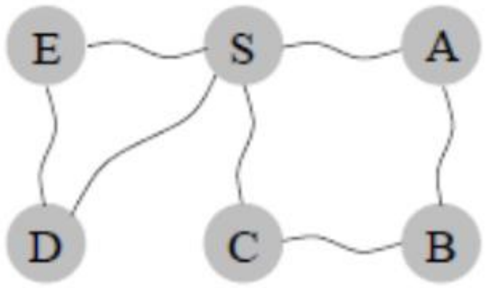


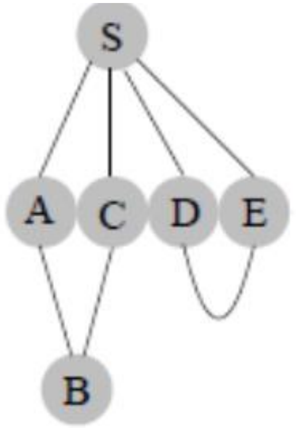


G

Distance in a Graph

- Distance between two nodes in the length of the shortest path between them

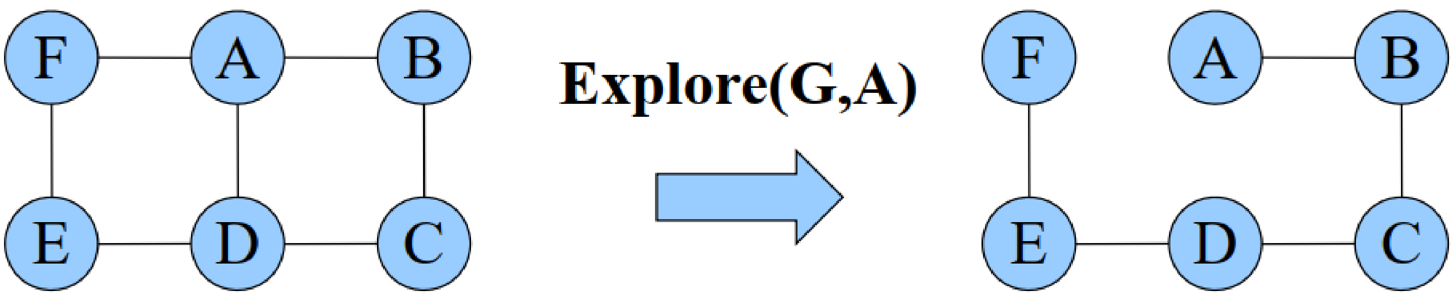




- If nothing specified, each edge length = 1
- d(S, B) = 2;
- d(S, D) = 1, …

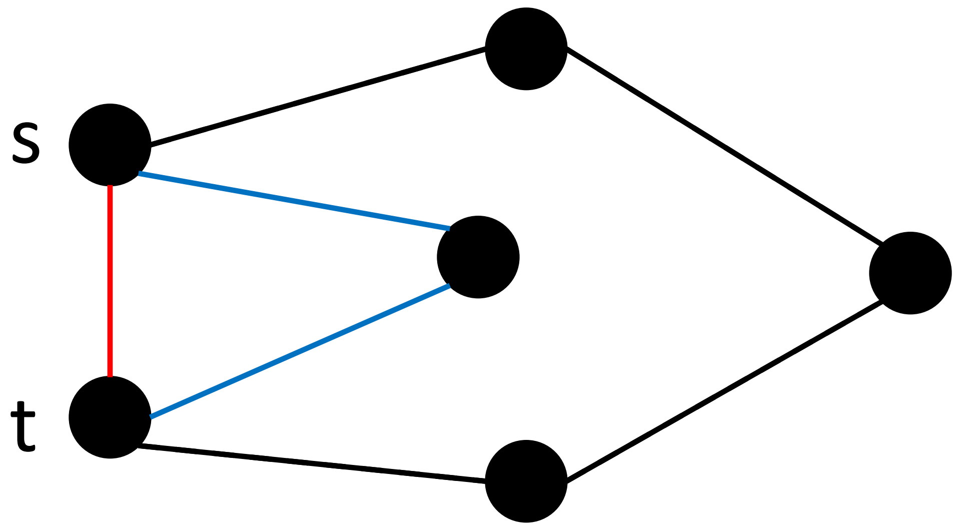
DFS Limitations

- DFS(G) finds all nodes reachable from a start node s
  - Explicit paths to these nodes == DFS tree
- So, DFS determines whether a path exists between nodes in a graph
- E.g., DFS finds a path of length 5 from A to F, but the shortest A-F path has length 1.



BFS Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.

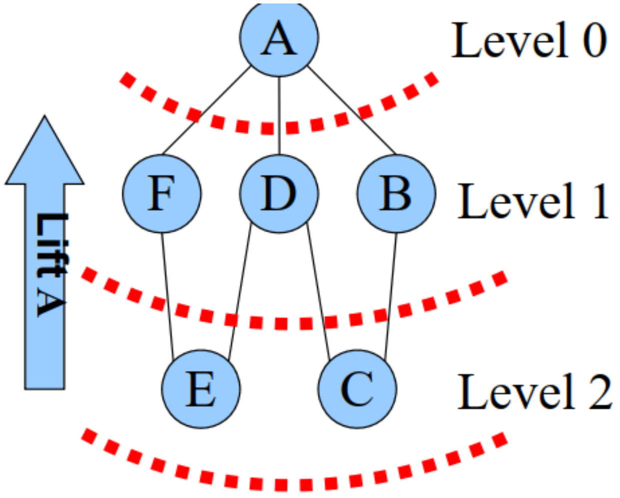


4 edges
2 edges
1 edge

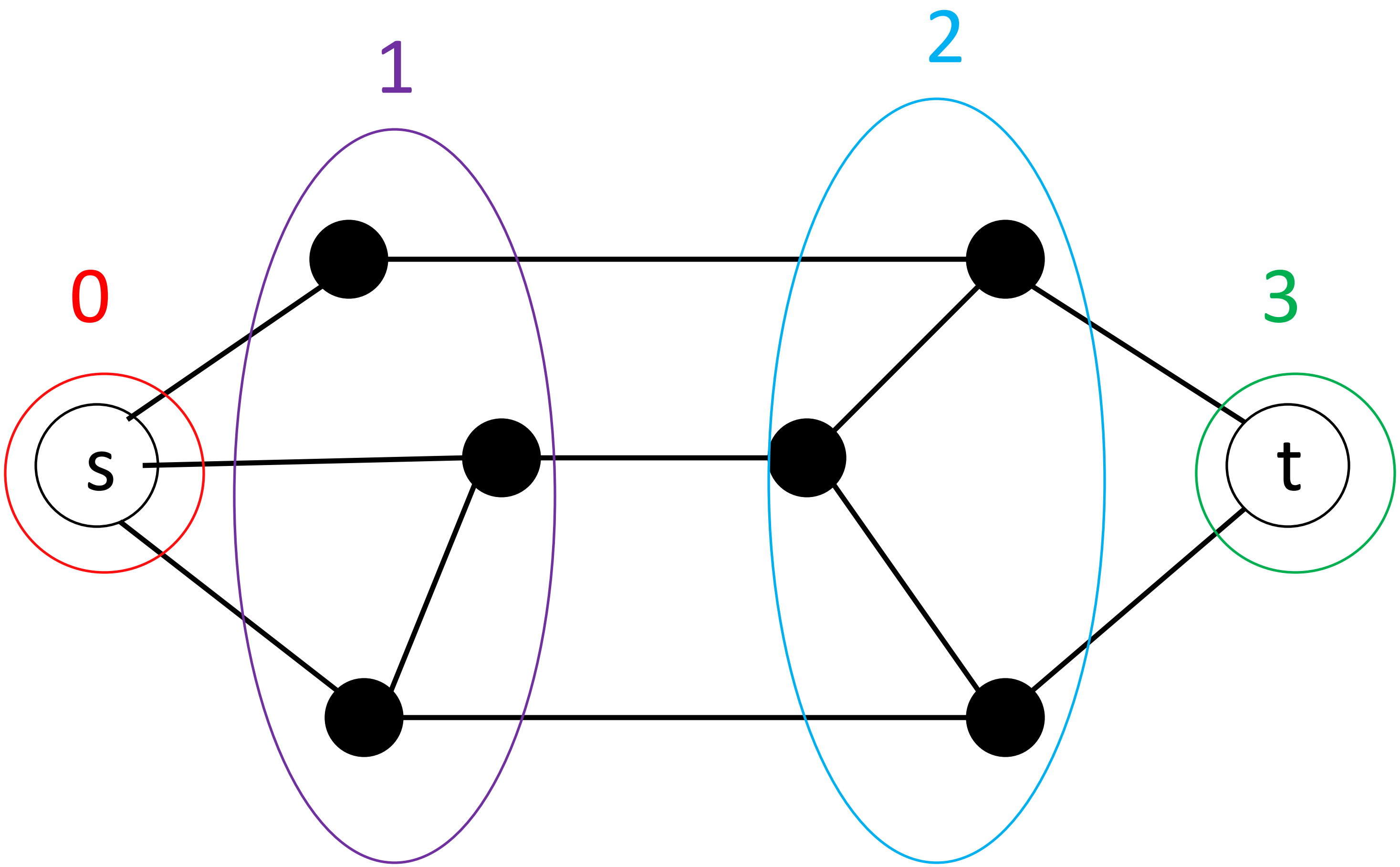
Best is 1 edge.

Breadth-First Search: The Idea

- Imagine vertices as balls, edges as strings tied to the balls
- To find the shortest paths: "lift the start vertex off the ground" → get a layered view of the graph
- Idea: Find vertices at distance 0, then 1, 2, etc.
- Suppose we've found all nodes at distance $\leq d$
- A node is at a distance $(d + 1)$ if:
  - It is adjacent to a node at distance d
  - It hasn't been seen yet



Example



Runtime

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← Q.dequeue()
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

O(|V|)

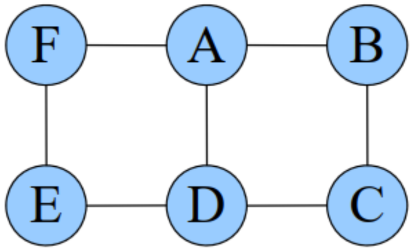
O(|V|) iterations

O(|E|) total iterations

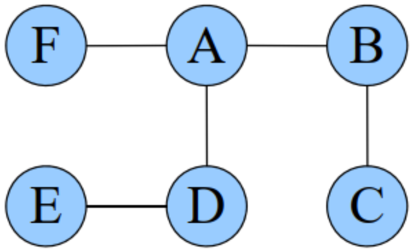
Total runtime:
O(|V|+|E|)

BFS Example

Queue	d(A)	d(B)	d(C)	d(D)	d(E)	d(F)
[**A**]	0	∞	∞	∞	∞	∞
*[**B** D F]	0	1	∞	1	∞	1
[**D** F C]	0	1	2	1	∞	1
[**F** C E]	0	1	2	1	2	1
*[**C** E]	0	1	2	1	2	1
[**E**]	0	1	2	1	2	1
*[ ]	0	1	2	1	2	1



BFS Tree



* = All nodes at level 0, 1, 2 (respectively, per each *) have been processed

DFS vs BFS

- Processed vertices (visited, dist $< \infty$)
- For each vertex, process all unprocessed neighbors
- Difference:
  - DFS uses a stack to store vertices waiting to be processed
  - BFS uses a queue
- Big effect
  - DFS goes depth first – very long path
  - BFS is breadth first – visits all side paths

Problem: Shortest Paths

Problem: Given a Graph G with vertices s and t and a length function ℓ, find the shortest path from s to t.

BFS finds the shortest path from s to t in a graph G in which all edge weights are 1.

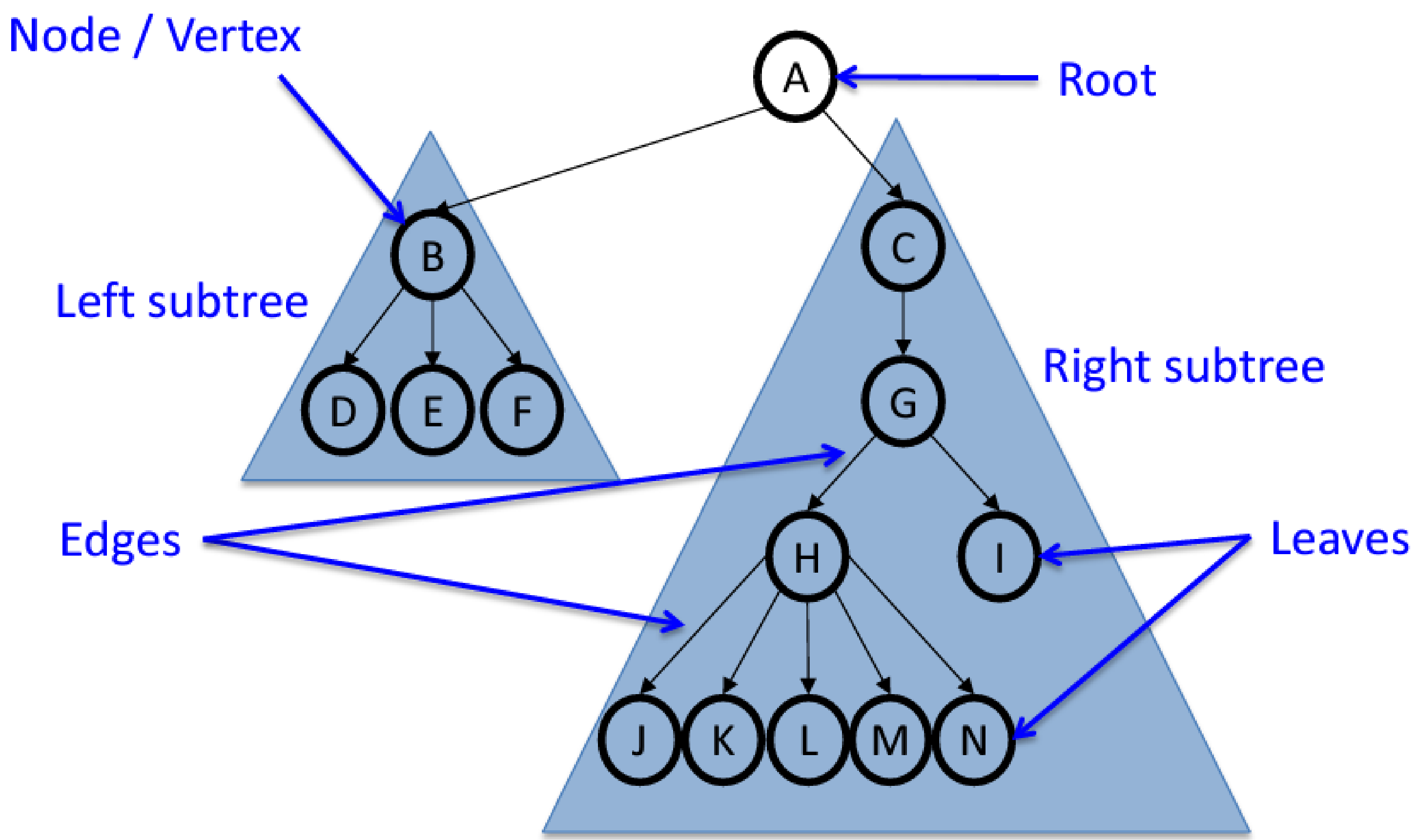
Tree Data Structure

- Nonlinear like graphs, $T = (V, E)$
- Can be empty
- Always contains a root node if non-empty
  - >= 1 substree(s) if $|V| \geq 2$ and $|E| \geq 1$; and at least one leaf/sink node
  - Else, zero subtrees
- Root is at level 0 (typically)
- Leaf: node with no children
- Non-leaf nodes are “internal” nodes

Trees

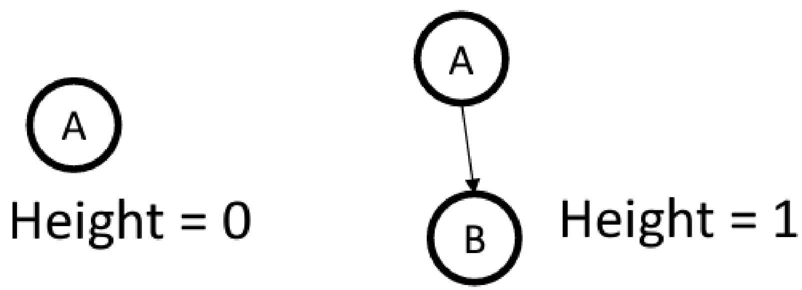
- A tree is connected and acyclic
- A tree has n nodes and (n-1) edges
- Any connected, undirected graph with |V|-1 edges is a tree
- An undirected graph is a tree if and only if (iff) there is a unique path between any pair of nodes
- Fact about trees: any two of the following properties imply the third:
  - Connected
  - Acyclic
  - |V|-1 edges

Tree Terminologies



Tree Calculations

- Height: Max # edges from the root to leaf
- Depth of a node: # edges from the root to the node



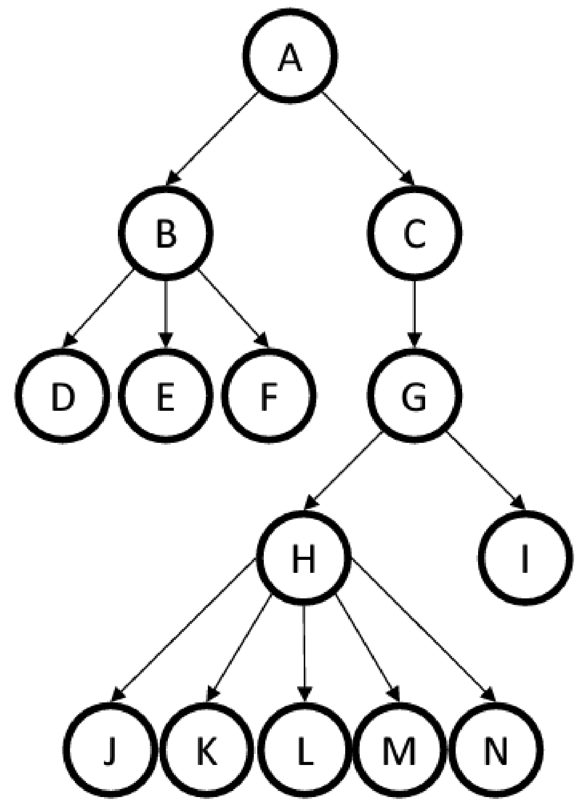
Tree Calculations

- Height: Max # edges from the root to leaf

- What is the height of this tree?
  - 4

- What is the depth of node G?
  - 2

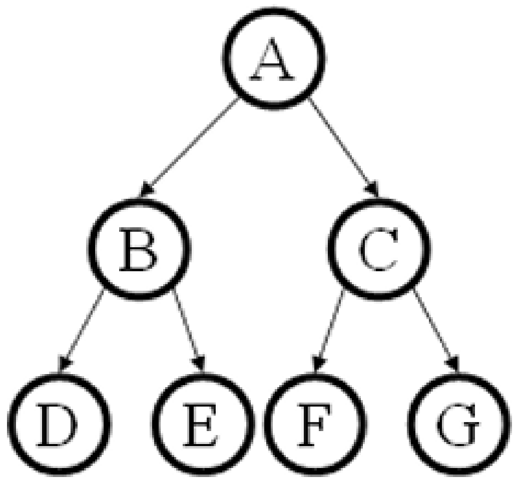
- What is the depth of node L?
  - 4

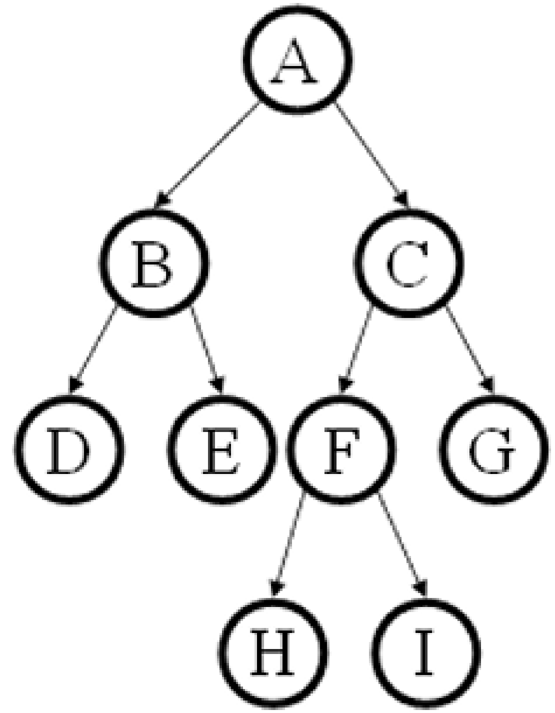


Binary Trees

- Any node has at most two children (i.e., a branching factor of 2)
- Consists of
  - A root (with data)
  - A left subtree (may be empty)
  - A right subtree (may be empty)

Binary Trees: Special Cases





Perfect/complete tree (2 nodes for root and internal nodes; leaves at the same level)

Full tree (0 or 2 children)

Create Root

```python
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
    def PrintTree(self):
        print(self.data)

root = Node(10)
root.PrintTree()
```

Insert

```python
    def insert(self, data):
# Compare the new value with the parent node
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                    if self.right is None:
                        self.right = Node(data)
                    else:
                        self.right.insert(data)
        else:
            self.data = data
```

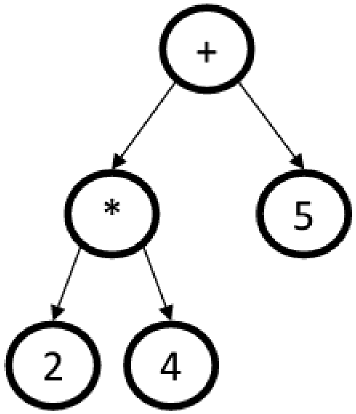
```python
# Print the tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()

# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

3 6 12 14

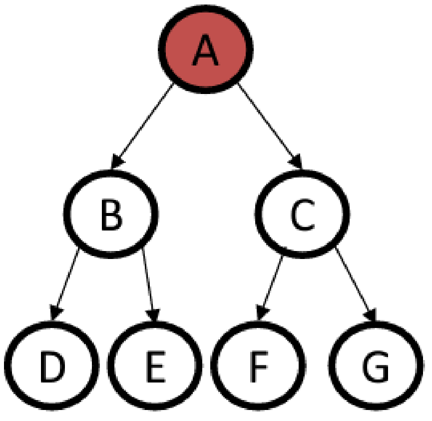
Binary Tree Traversals

- Pre-order: root, left subtree, right subtree
- In-order: left subtree, root, right subtree
- Post-order: left subtree, right subtree, root

- Example: expression tree
  - Pre-order: + * 2 4 5
  - In-order: 2 * 4 + 5
  - Post-order: 2 4 * 5 +



Traversals

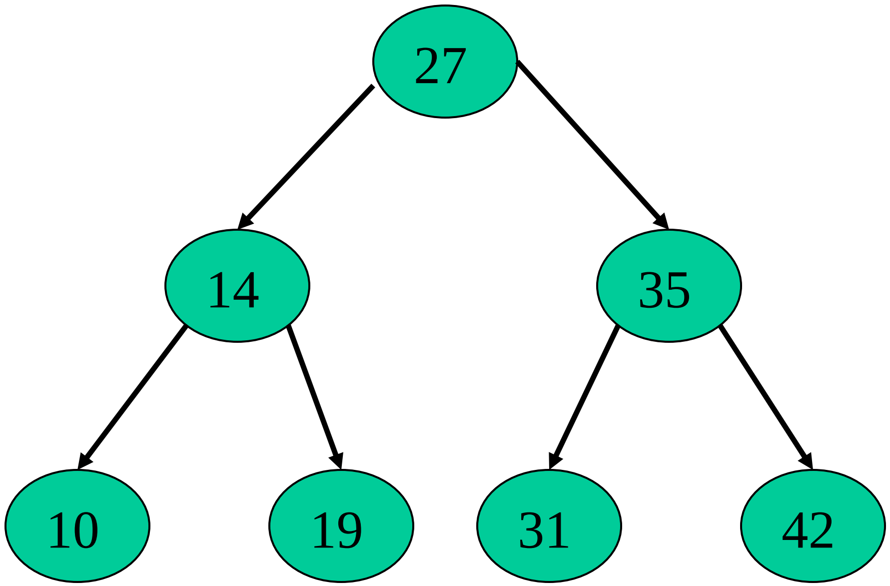


In-order: D B E A F C G

Pre-order: A B D E C F G

Post-order: D E B F G C A

Tree Example

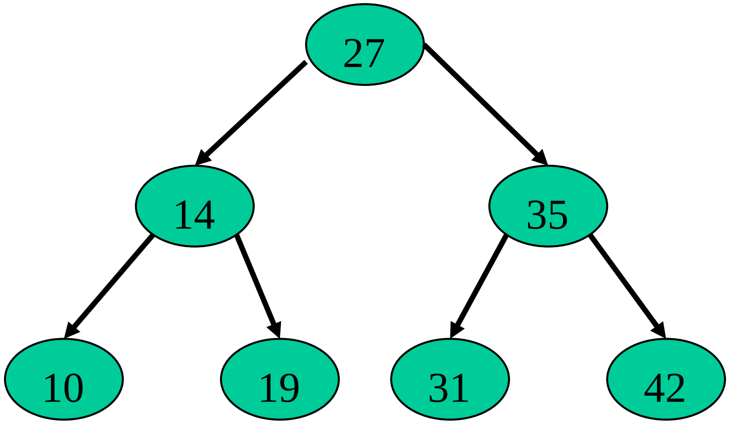



```
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
```

Pre-Order Traversal

```python
def PreorderTraversal(self, root):
    res = []
    if root:
        res.append(root.data)
        res = res + self.PreorderTraversal(root.left)
        res = res + self.PreorderTraversal(root.right)
    return res
```

```python
print(root.PreorderTraversal(root))
```

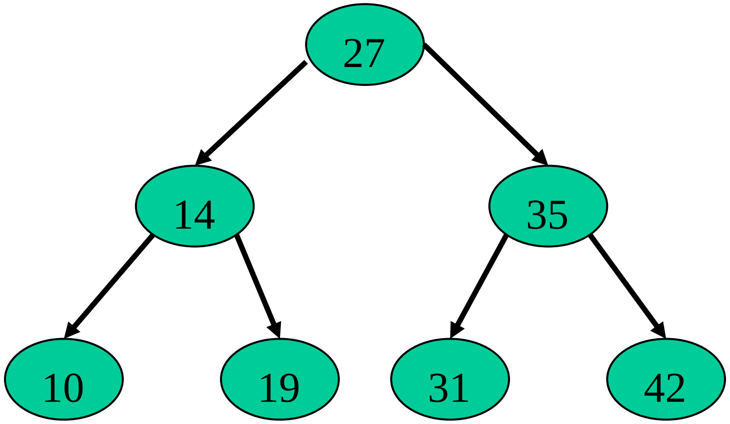


27, 14, 10, 19, 35, 31, 42

In-Order Traversal

```
def inorderTraversal(self, root):
    res = []
    if root:
        res = self.inorderTraversal(root.left)
        res.append(root.data)
        res = res + self.inorderTraversal(root.right)
    return res
```

```
print(root.inorderTraversal(root))
```

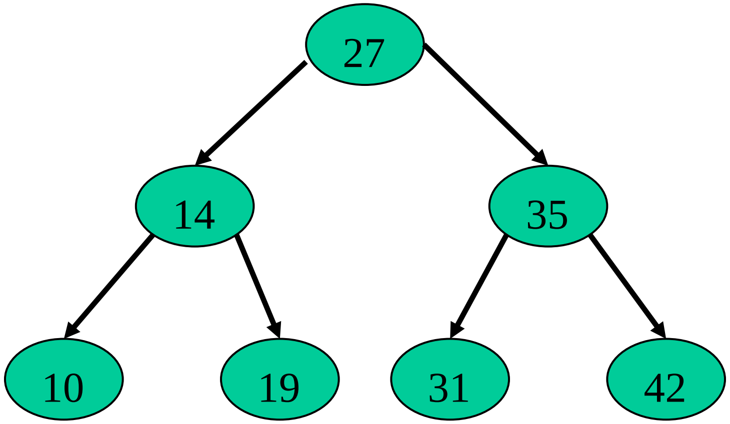


10, 14, 19, 27, 31, 35, 42

Post-Order Traversal

```
def PostorderTraversal(self, root):
res = []
if root:
res = self.PostorderTraversal(root.left)
res = res + self.PostorderTraversal(root.right)
res.append(root.data)
return res
```

```
print(root.PostorderTraversal(root))
```



10, 19, 14, 31, 42, 35, 27

Lecture 6 summary

- DAGs and Topo Sort
- BFS
- Trees and traversals