# CS112: Data Structures

## Lecture 11

### Graphs

# Upcoming Schedule

- **Wed, July 20:**
  - 6-8:30 work on Proj. 3 with Binh & me here to help (bring laptops!)
- **Mon, July 25:**
  - 6-7 recitation
  - 7:10 - 8:30 review
- **Wed, July 27:**
  - 6 - 7:20 Midterm 2 (info to be posted)

# Review: Priority Queues

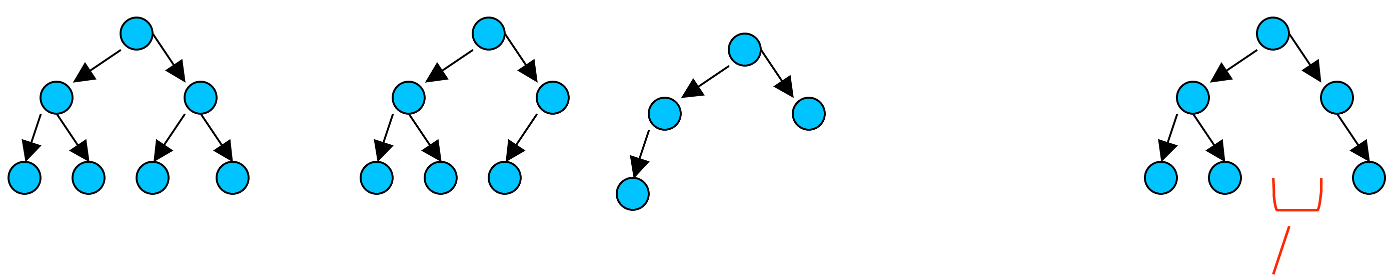
- **Each data item has a priority**
- **Add items to queue in any order**
- **Highest Priority First Out**
  - **add A:5, B:3, C:6**
  - **remove C**
  - **add D:8**
  - **remove D, remove A**

# Implement as an array

- Unsorted or sorted:  insert or delete is O(n)
- Can we find a data structure that gives O(insert + delete) bettern than O(n)?

# Heap

- **A heap is a way to implement a priority queue with O(log n) complexity**
- **A heap is a complete binary tree**
  - **all levels except maybe the last are full**
  - **last level filled from left to right**

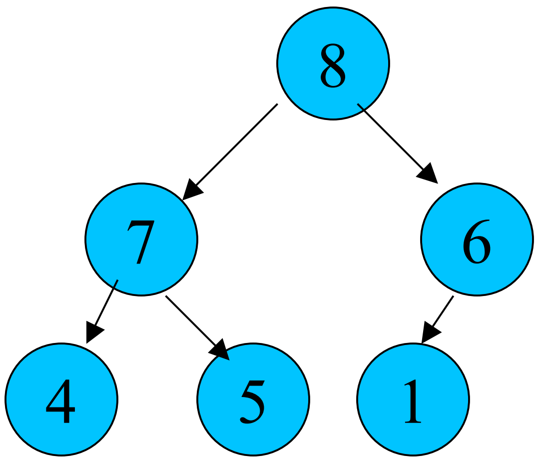


good

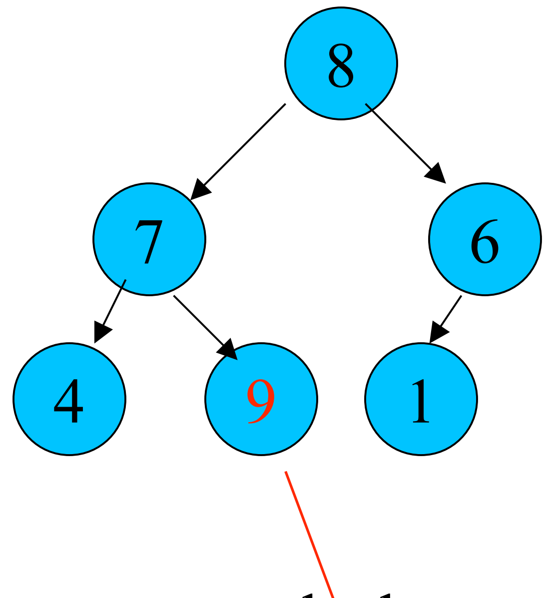
bad

# Heap

- **The number at a node is greater than the number at any descendant**



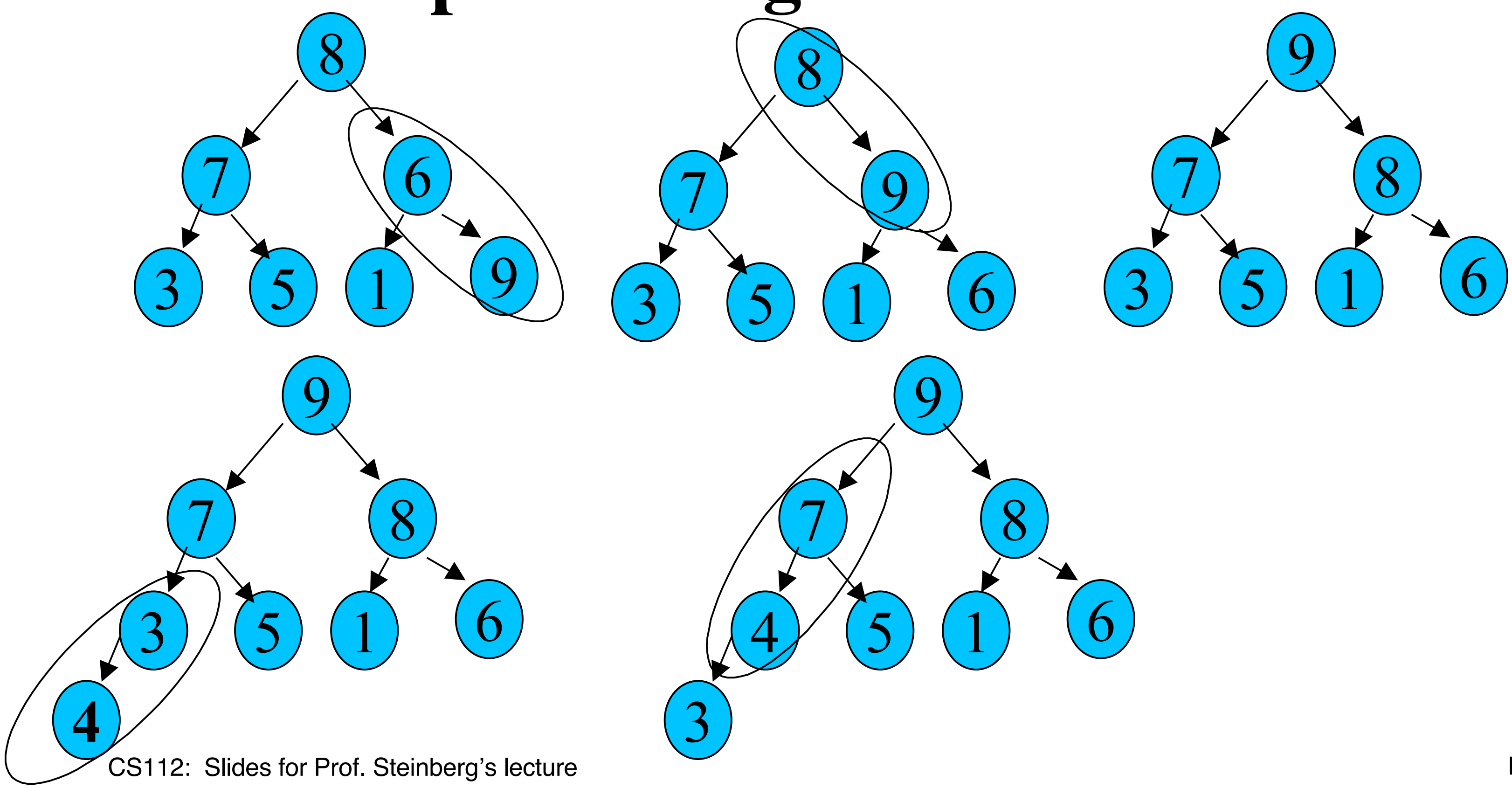

good




bad

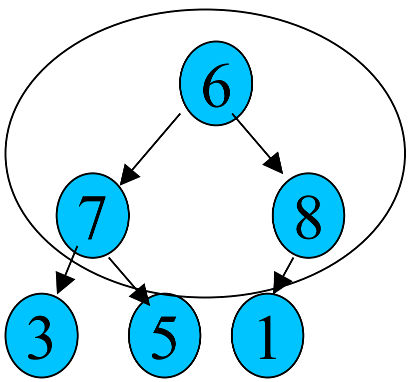
# Heap Insert

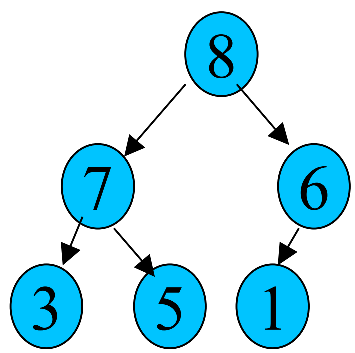
- **Add node at end of last level**
- **Move up restoring order**



# Heap Deletion

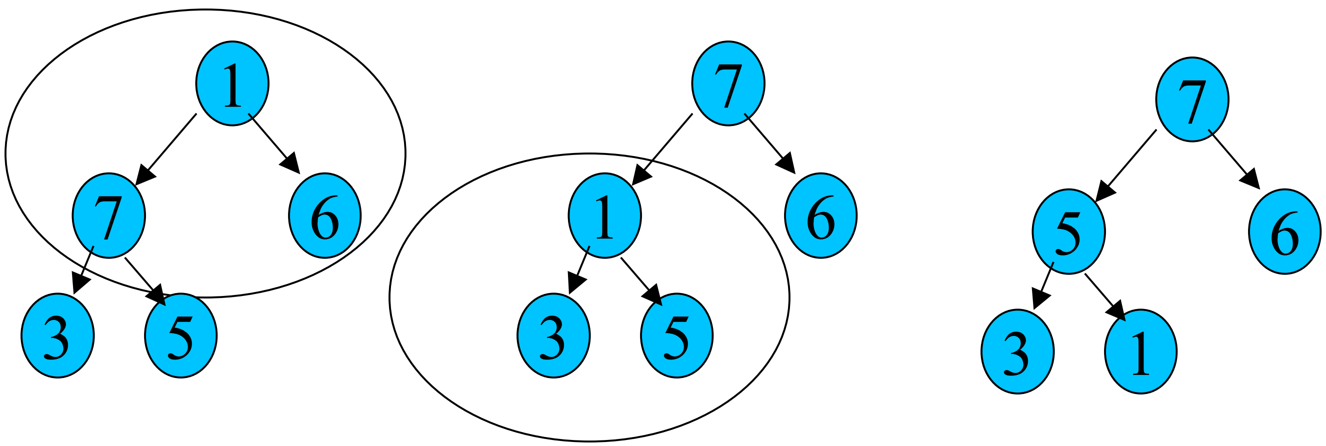
- Copy out data at root
- Delete last node on last row & put data in root
- Move down restoring order



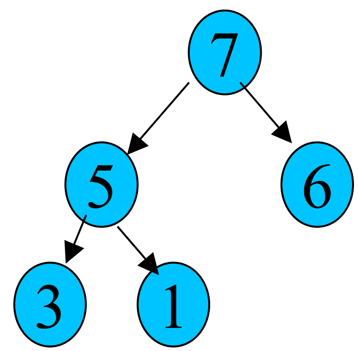
# Heap Deletion

- Compare current node and two children
  - if current node largest, stop
  - if left node largest swap current and left
  - ditto if right largest



# Heap Representation

- Store heap in an array
  - For node at index j, children are at 2j+1 and 2j+2
  - Root at index 0

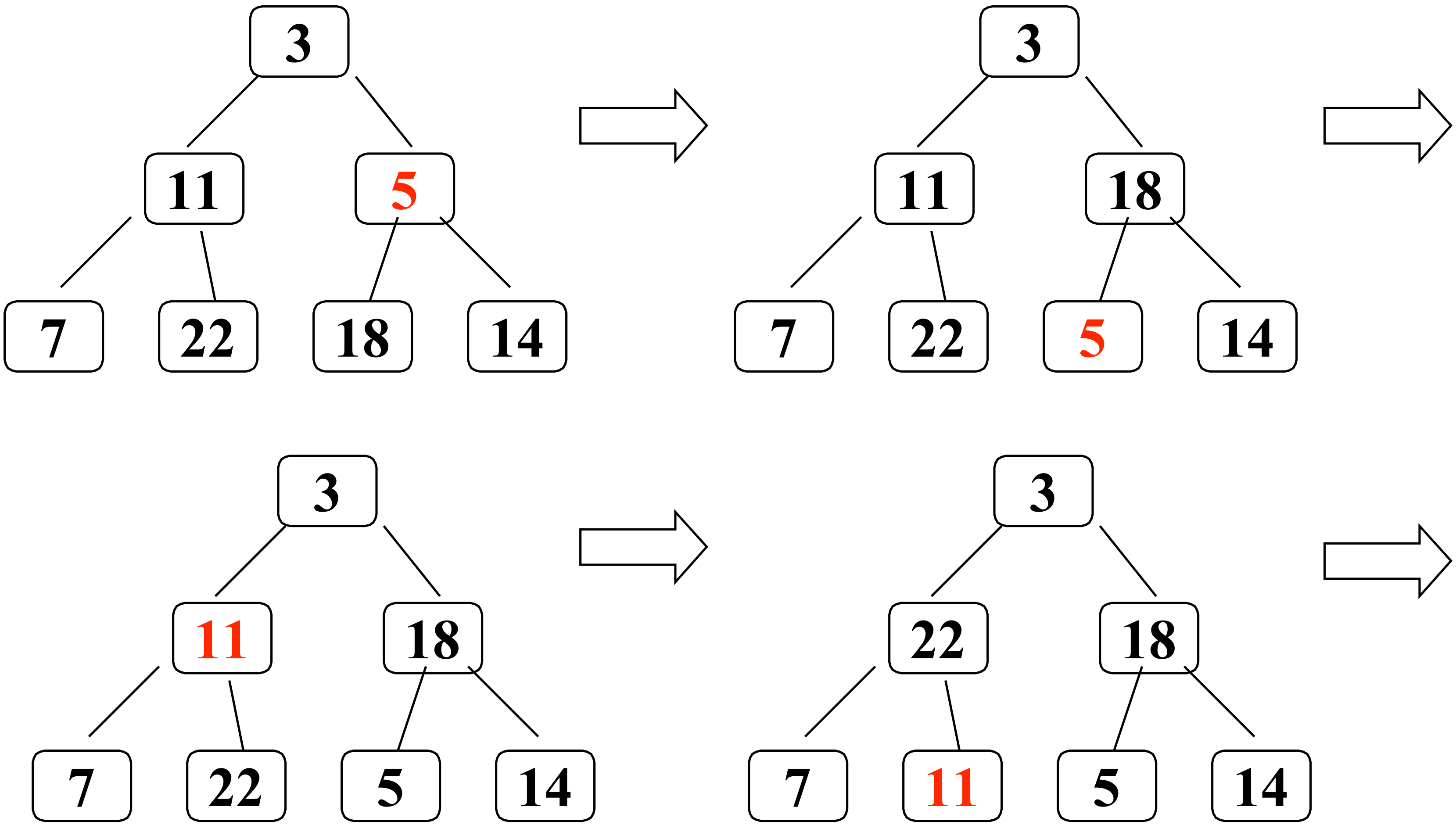



| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 7 | 5 | 6 | 3 | 1 |

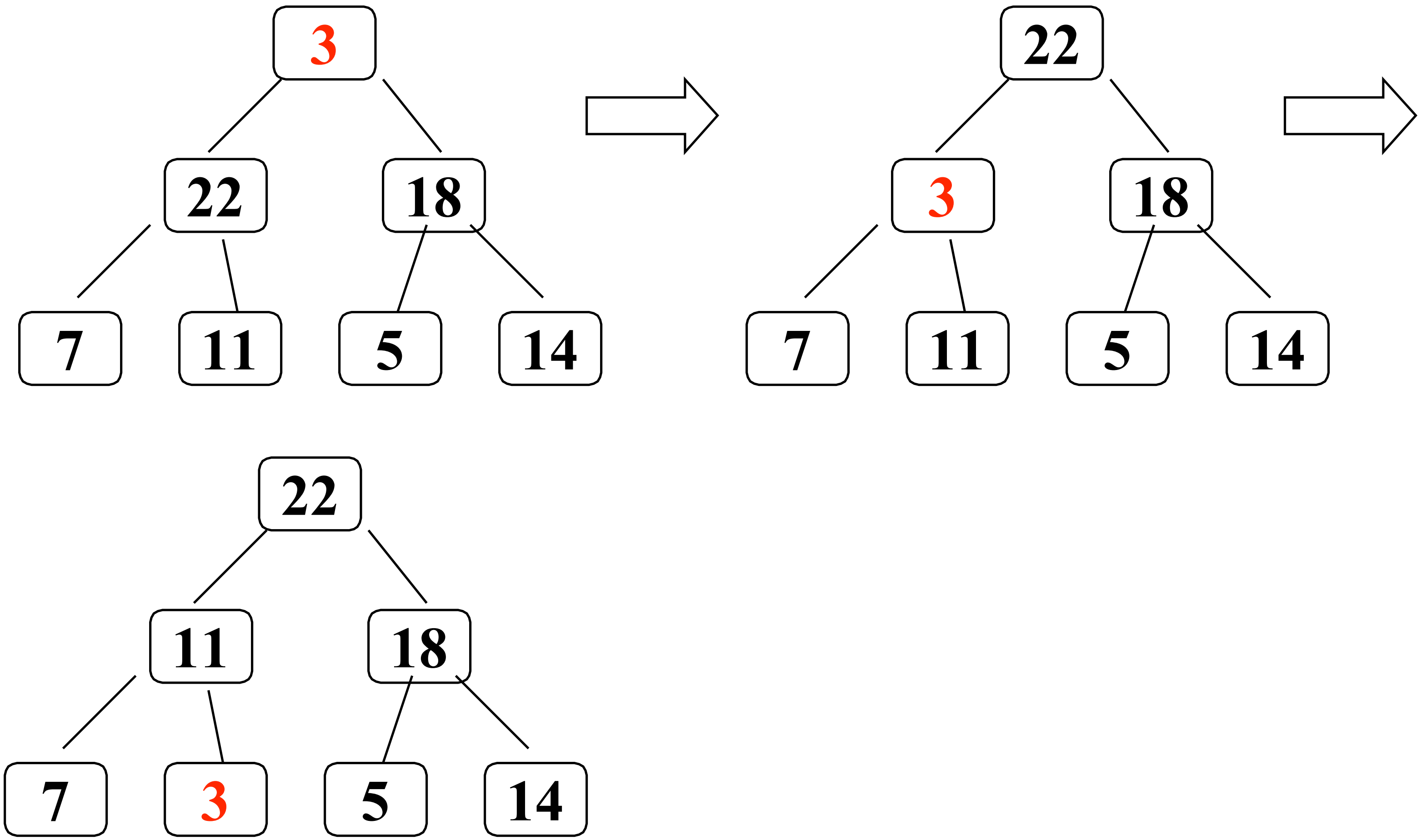
# Building a Heap from an Array

- **Go from last non-leaf to index 0**
- **At each node, do filter-down**
- **Work at a node is O(height of node)**
- **In a complete binary tree, majority of nodes close to bottom, so adds up to O(n)**

# Building a Heap from an Array



# Building a Heap from an Array



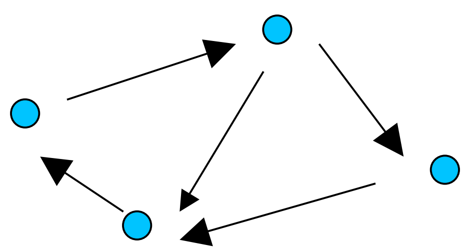
# Sorting with trees

- **Sort with heap or AVL tree: O(n log n)**
  - **insert all the data**
  - **read in order**
    - **AVL tree: do inorder traversal**
    - **heap: remove all nodes**

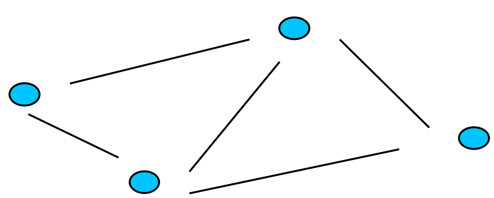
# Graphs

**Generalization of trees**

- **Digraph (Directed Graph)**
  - **Like a tree but any vertex can point to any other**



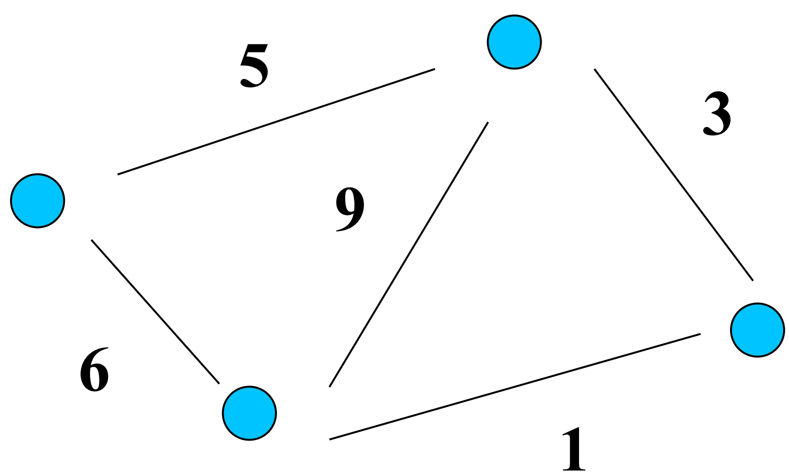
- **Graph**
  - **like digraph but arcs have no direction**



# Graphs

**Generalization of trees**

- **Weighted Graph**
  - **Positive integer weights on each edge**



# Applications

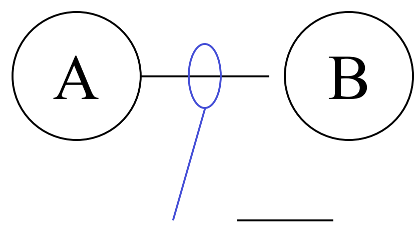
- Paths
  - On streets (eg mapquest)
- Electrical networks
  - On circuit boards
  - Power lines
- Constraints
  - Ordering constraints on building steps
  - Sudoku

# Applications

- Relationships
  - Web page references
  - Friendships (online and real world)
- Etc, etc, etc

# Notation

- **Arcs are named by the vertices they connect**




**(B,A) or $\overline{\text{BA}}$**

# Representing Graphs

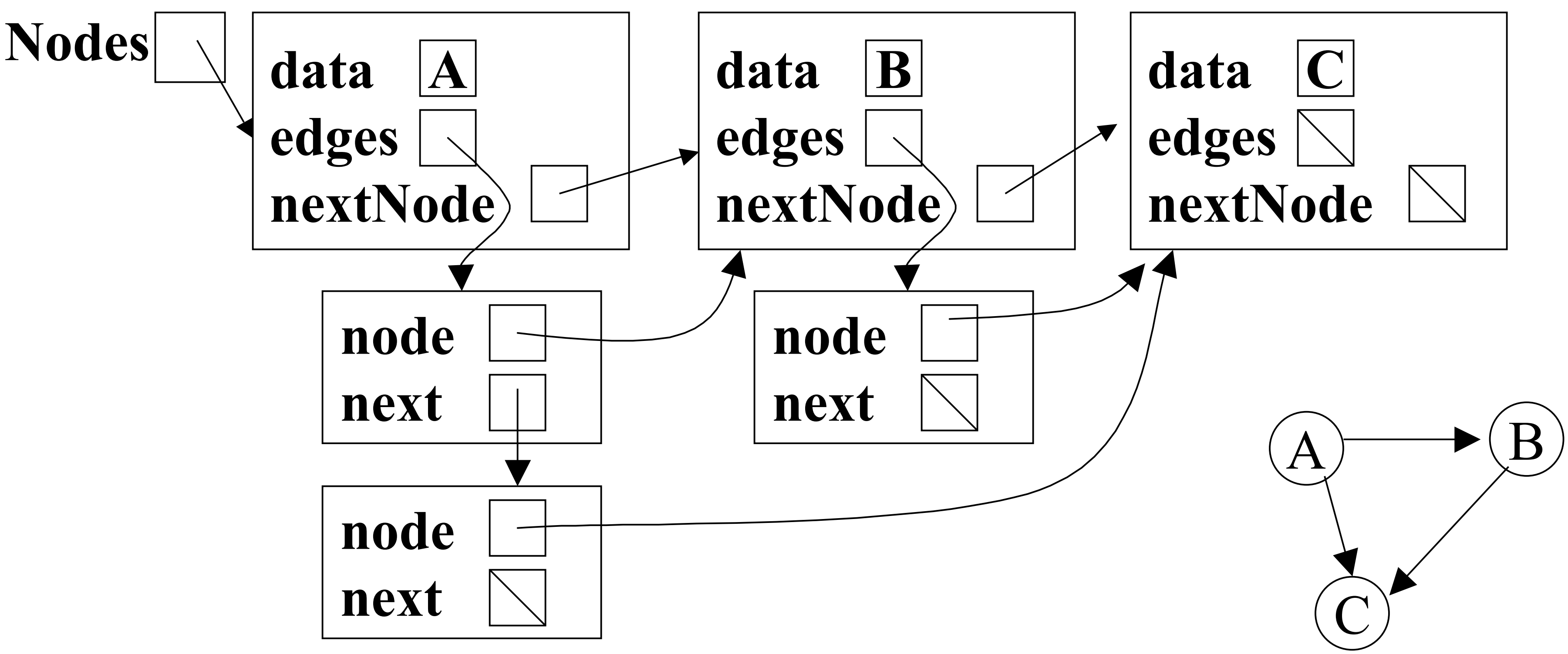
- **Adjacency list**
  - for each node, linked list of edges

```
public Gnode{
  String data;
  Edge edges;
  Gnode nextNode;
  …
}
```

```
public Edge{
  Gnode node;
  Edge next;
  …
}
```

# Representing Graphs

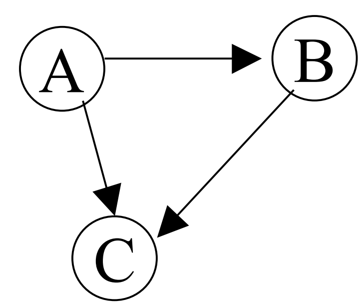
- **Adjacency list**
  - for each node, linked list of edges



# Representing Graphs

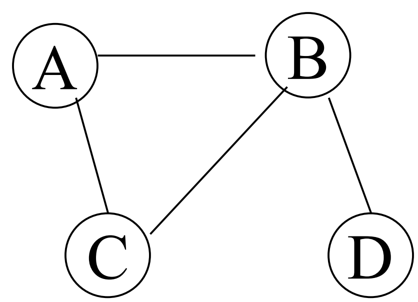
- **Adjacency matrix**
  - n x n boolean matrix: is there an arc?

| From \To | A | B | C |
|---|---|---|---|
| A | F | T | T |
| B | F | F | T |
| C | F | F | F |



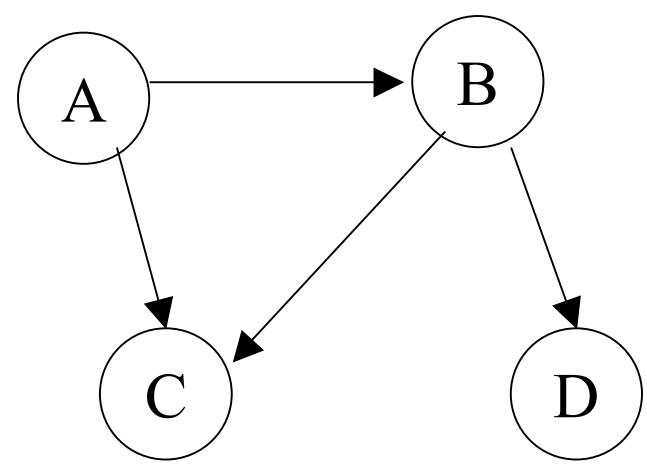
# Graph Concepts

- **Neighbors of a vertex: vertices that it shares an arc with**
  - **Neighbors of A are B and C**
- **Degree of a vertex: number of neighbors**
  - **Degree of A is 2, degree of B is 3**



# Graph Concepts

- **In degree (in a digraph): number of vertices that have arcs to this vertex**
  - **In degree of B is 1**
- **Out degree (in a digraph): number of vertices that have arcs from this vertex**
  - **Out degree of B is 2**



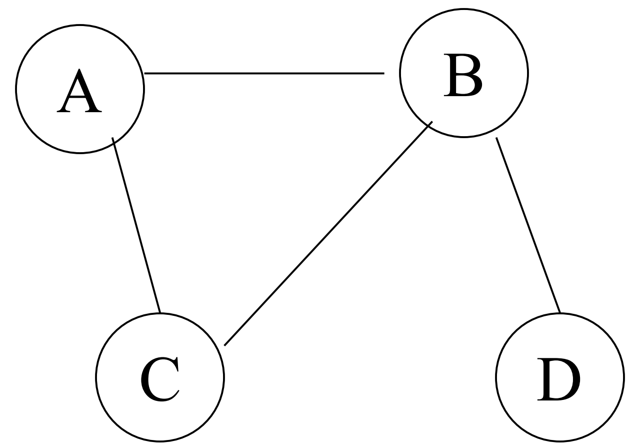
# Graph Concepts

- **(Simple) Path**
  - **Sequence of arcs**

    **(A,B),(B,C)**
  - **May not revisit a vertex**

    **(B,A),(A,C),(C,B),(B,D)**
  - **Except last vertex may = first**

    **(B,A),(A,C),(C,B)**
- **Vertex A is** reachable **from B if there is a path from B to A**

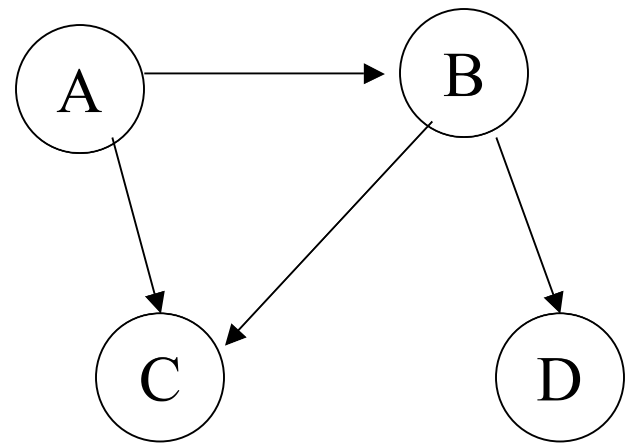


# Graph Concepts

- **(Simple) Path**
  - **On digraph must follow arc directions**

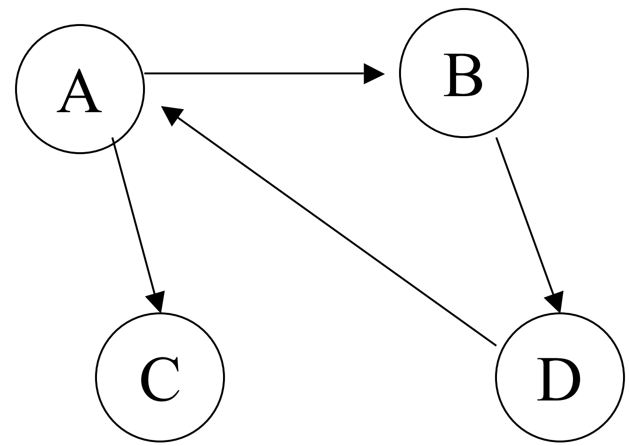
    **(A,B),(B,D)**

    **(A,C),(C,B)**



# Graph Concepts

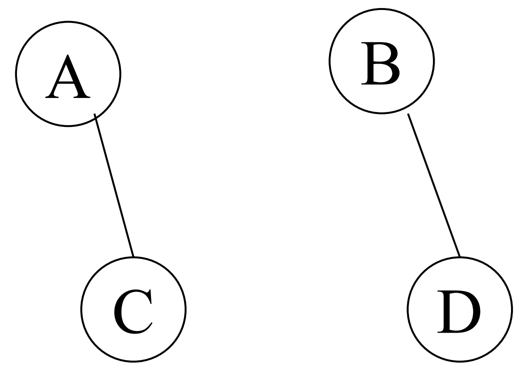
- **A cycle is a path from a node back to itself**
  - **(A, B)(B, D)(D, A)**
- **A graph with no cycles is called acyclic**



# Graph Concepts

- **Connected Graph**

  **For any two vertices X and Y there is a path from X to Y.**



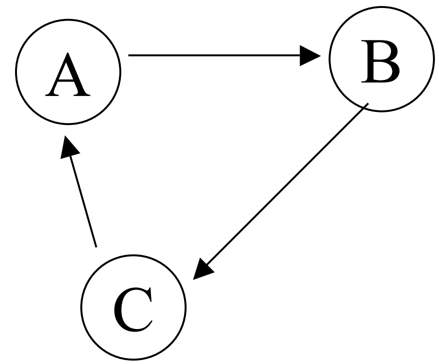
# Graph Concepts

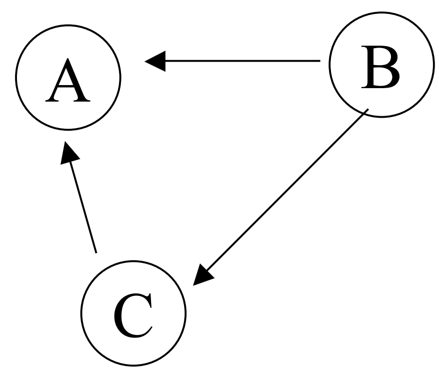
- **Strongly Connected Digraph**

  **For any two vertices X and Y there is a path from X to Y. (Paths must follow arc directions)**

- **Weakly Connected Digraph**

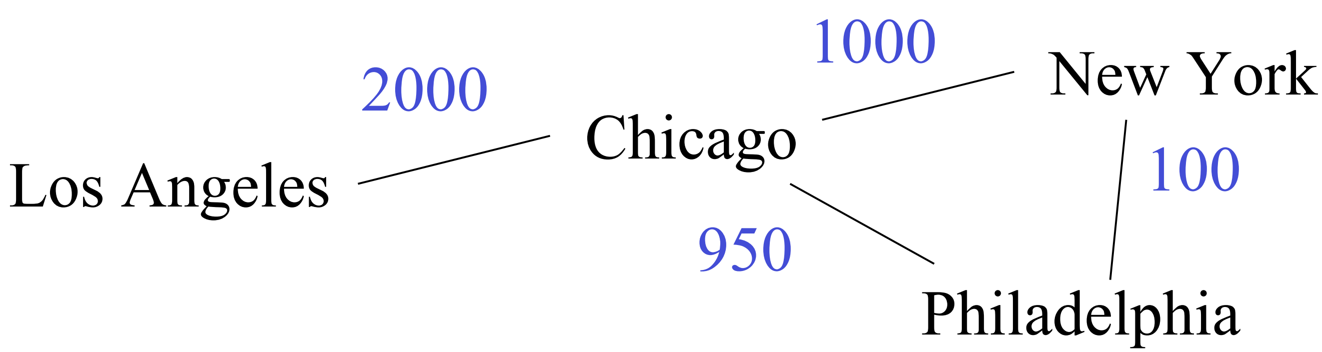
  **Corresponding graph is connected (i.e., ignoring arc direction)**



# Graph Concepts

- **Weighted graph: each arc has a numerical weight**



# Graph Traversals

- **Need to mark vertices to prevent infinite loops**
- **Need driver in case not connected**
- **Otherwise like tree traversals**
- **Depth first**

```
dfsG(v)
    if (marked(v)) return;
    process v;
    mark v;
    for each vn in neighbors(v)
      dfsG(vn)
```

# Graph Traversals

- **Need driver in case not connected**

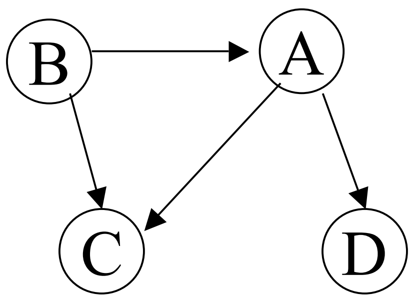
  **For v in vertices**

      **dfsG(v)**

# DFS Graph Traversal

- Enters a vertex v
- Visits all vertices reachable from v (that have not yet been visited
- Leaves v

# Graph Traversals



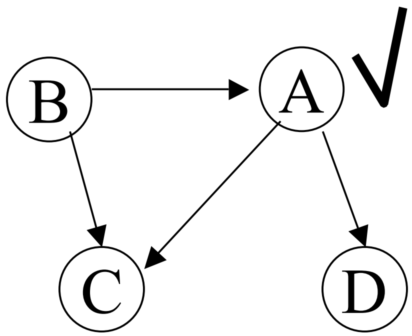

**Driver**

**v = <A>**

**dfsG**

**v = <A>**

# Graph Traversals




**Driver**

**v = <A>**

**dfsG**

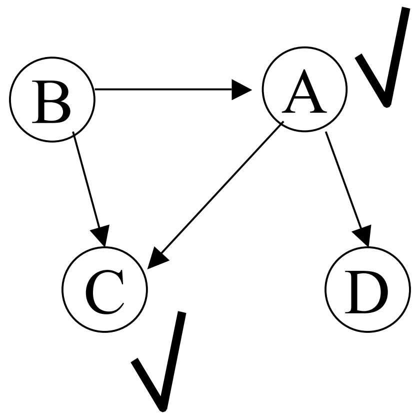
**v = <A>**

**vn = <C>**

**dfsG**

**v = <C>**

# Graph Traversals




**Driver**

**v = <A>**

**dfsG**

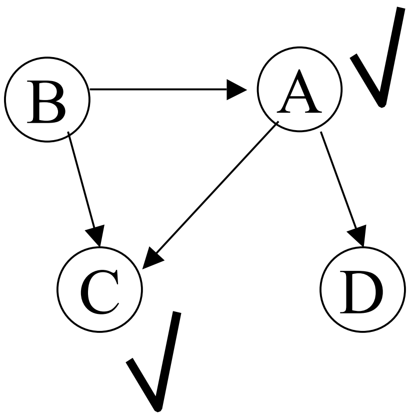
**v = <A>**

**vn = <C>**

**dfsG**

**v = <C>**

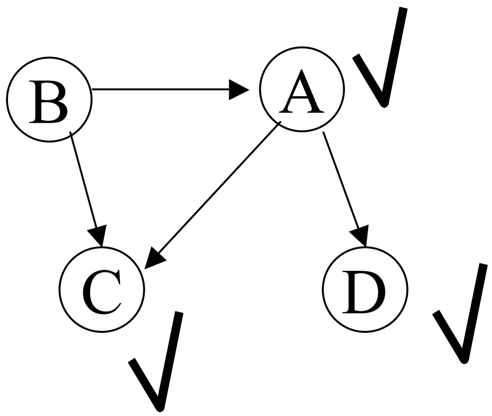
# Graph Traversals




**Driver**
**v = <A>**

**dfsG**
**v = <A>**
**vn = <D>**

# Graph Traversals

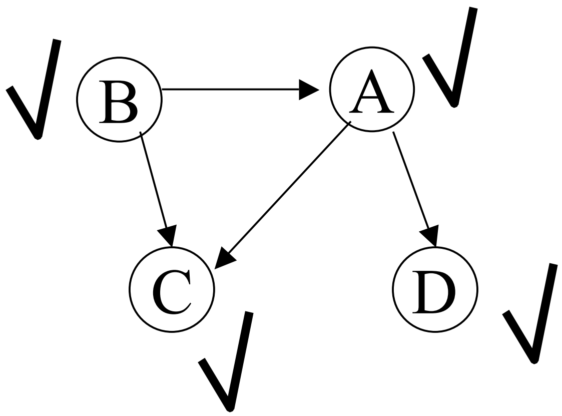



**Driver**
**v = <A>**

**dfsG**
**v = <A>**
**vn = <D>**

**dfsG**
**v = <D>**

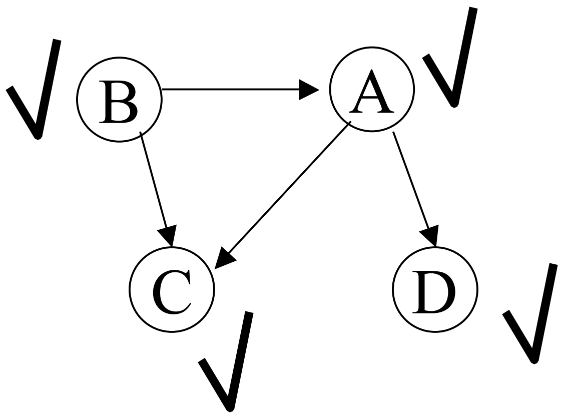
# Graph Traversals




**Driver**
**v = <B>**

**dfsG**
**v = <B>**

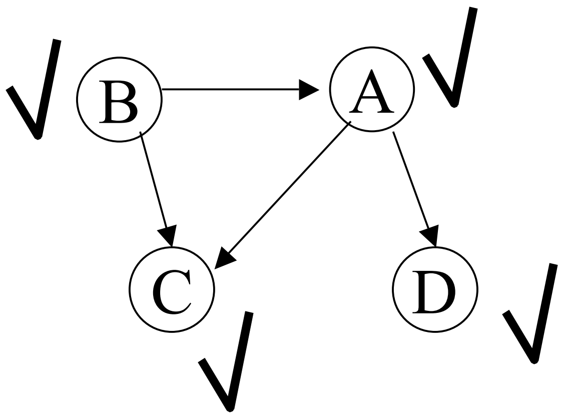
# Graph Traversals




**Driver**
**v = <C>**

**dfsG**
**v = <C>**

# Graph Traversals




**Driver**

**v = <D>**

**dfsG**

**v = <D>**

# Graph Traversals

- **Time:**
  - **Visit each vertex**
  - **inspect each arc**
  - **driver**

  **O(n + e)  n vertices, e edges**

# Topological Sort

- **Acyclic Digraph <=> partial order**
- **Topsort: find total order consistent with partial order**

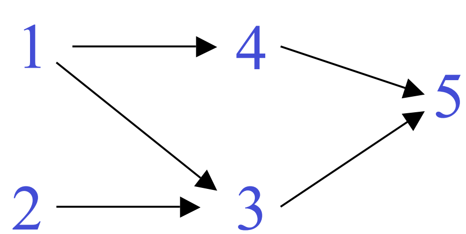
1 **a=1;**

2 **b=2;**

3 **c=a*b;**

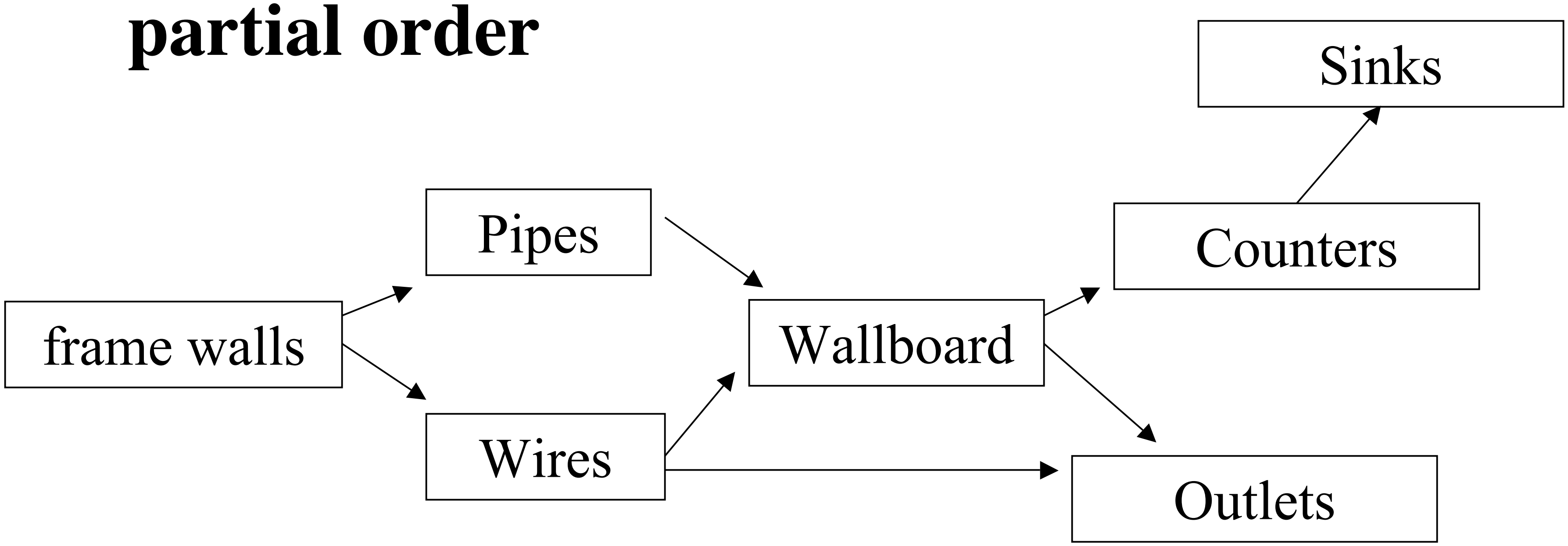
4 **d=a+4;**

5 **c=c+d**



# Topological Sort

- **Acyclic Digraph <=> partial order**
- **Topsort: find total order consistent with partial order**



# Topsort Algorithms

- Most work by assigning numbers to vertices
  - vertex order = numerical order
- Depth first
- Breadth First

# DFS Topsort Algorithm

- **Algorithm:**
  - Do DFS
  - Number vertices as you leave them
- **Problem: leave vertex *after* leave reachable vertices, but needs number *smaller* than reachable vertices**
  - Solution: number with decreasing numbers

# BFS Topsort Algorithm

- Keep a “predecessor count” for each vertex
  - Initially: in degree
  - When a predecessor is numbered, decrement count

# BFS Topsort Algorithm

- **enqueue all sources**
- **while not queue.isEmpty**

  **v = queue.dequeue( )**

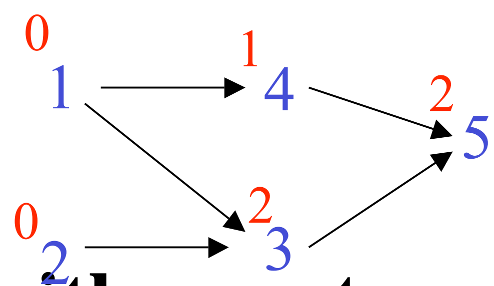
  **number v (increasing numbers)**

  **decrement predecessor counts of v's neighbors**

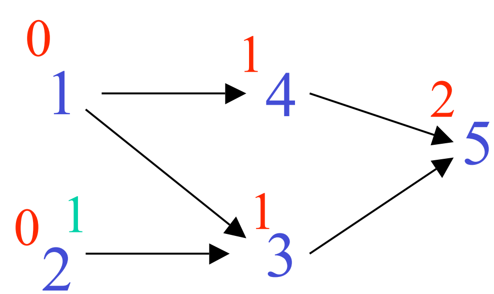
    **if count becomes 0, enqueue neighbor**

# BFS Topsort Algorithm

- Keep predecessor count for each vertex




- Find vertex with count == 0
  - number it
  - decrement counts of neighbors



# Shortest Path

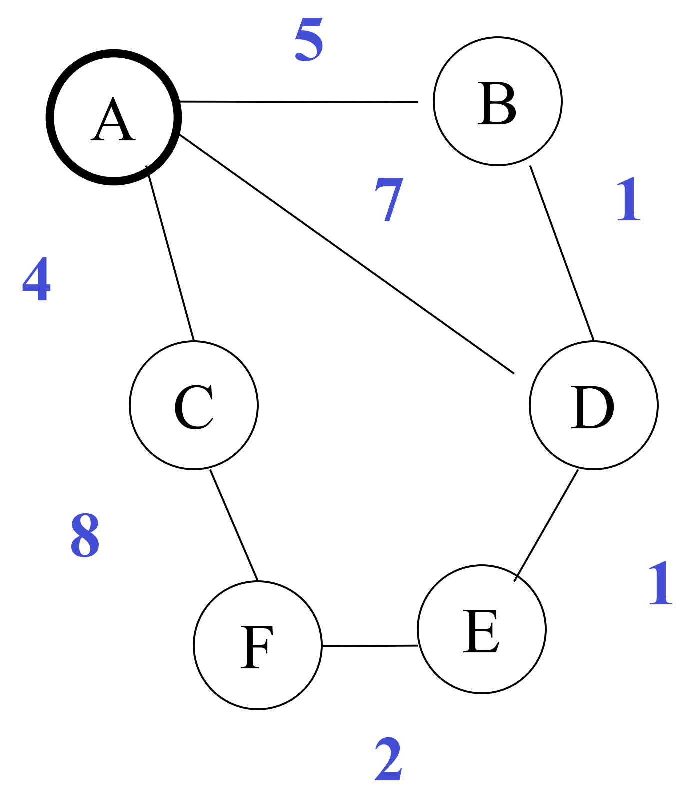
- **weighted digraph**
  - **weights are all > 0**
- **"length" of a path = sum of weights of arcs on path**
- **given start vertex, end vertex, find shortest path from start to end**

# Dijkstra's Algorithm

- Grow a tree of paths from start
  - tree is subgraph of original digraph
  - grow it one vertex at a time
  - only add a vertex when we know where to put it so that path to vertex from root in tree is shortest in digraph
  - when we add end vertex to tree the shortest path from start to end is given by path in tree

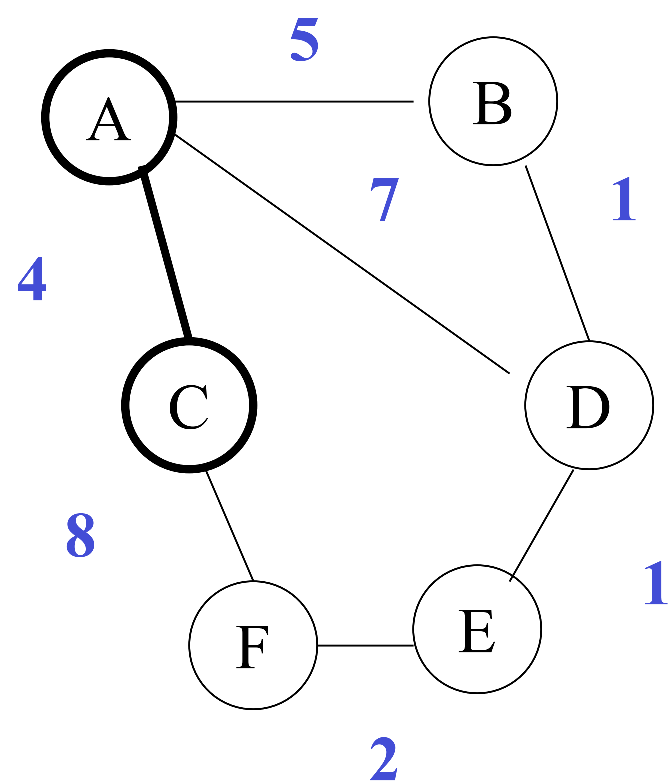
# Example

| Node | Status | LinK | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Fringe | A | 5 |
| C | Fringe | A | 4 |
| D | Fringe | A | 7 |
| E | | | |
| F | | | |



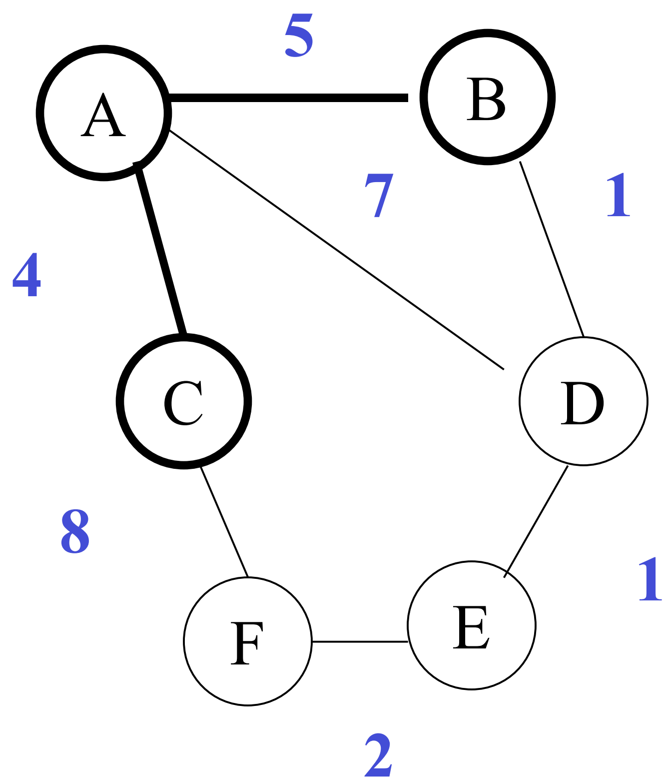
# Example

| Node | Status | LinK | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Fringe | A | 5 |
| C | Tree | A | 4 |
| D | Fringe | A | 7 |
| E | | | |
| F | Fringe | C | 12 |



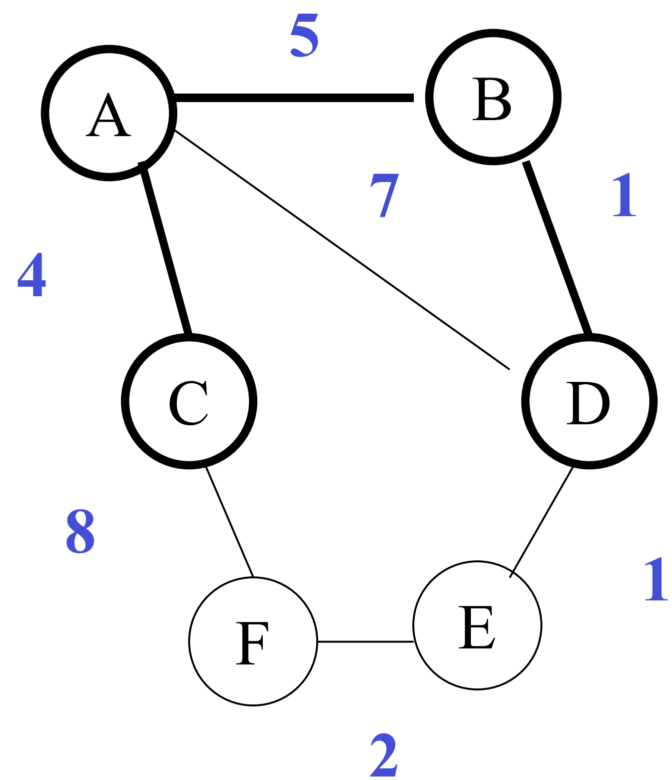
# Example

| Node | Status | LinK | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Tree | A | 5 |
| C | Tree | A | 4 |
| D | Fringe | B | 6 |
| E | | | |
| F | Fringe | C | 12 |



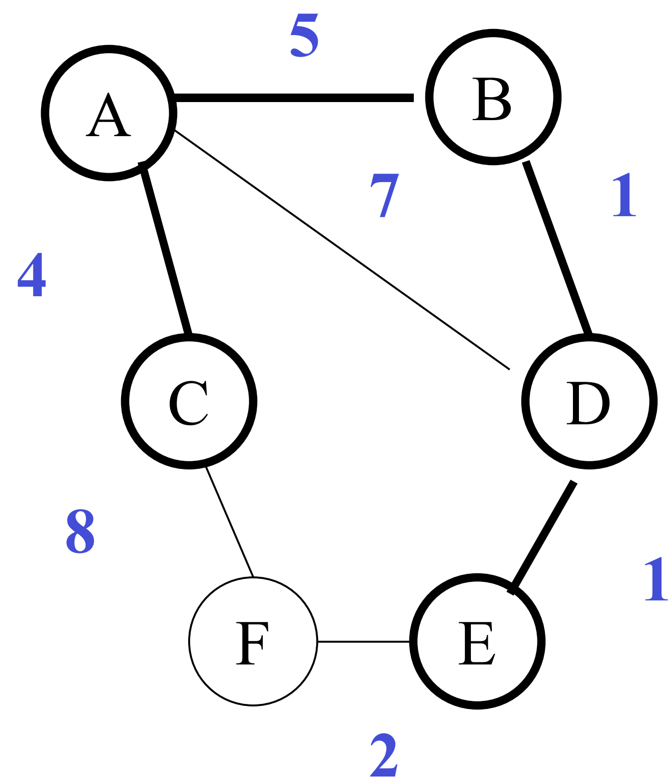
# Example

| Node | Status | LinK | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Tree | A | 5 |
| C | Tree | A | 4 |
| D | Tree | B | 6 |
| E | Fringe | D | 7 |
| F | Fringe | C | 12 |



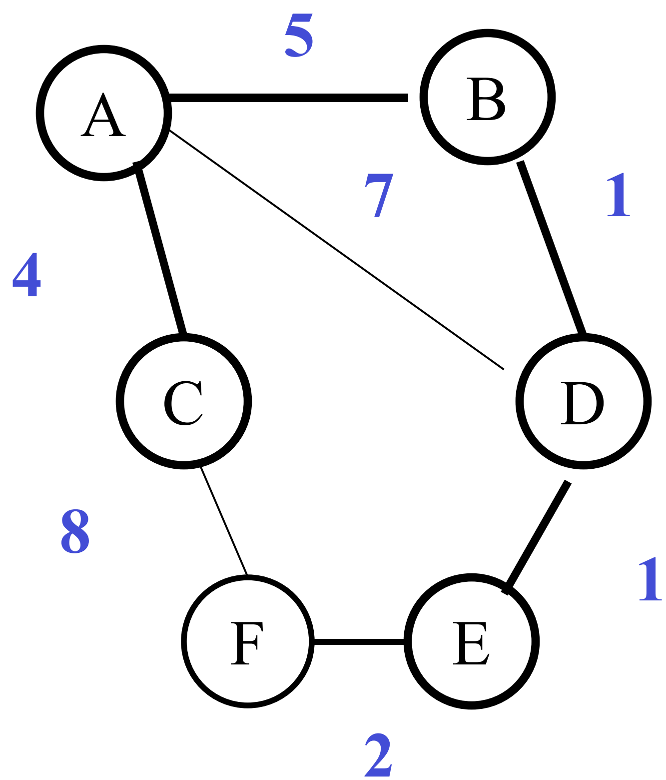
# Example

| Node | Status | LinK | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Tree | A | 5 |
| C | Tree | A | 4 |
| D | Tree | B | 6 |
| E | Tree | D | 7 |
| F | Fringe | E | 9 |



# Example

| Node | Status | Link | Distance |
|---|---|---|---|
| A | Tree | -- | 0 |
| B | Tree | A | 5 |
| C | Tree | A | 4 |
| D | Tree | B | 6 |
| E | Tree | D | 7 |
| F | Tree | E | 9 |



# Dijkstra's Algorithm

- **How can we be sure we are attaching vertex at right point?**
  - **assume tree so far is shortest paths**
  - **choose vertex X and arc (Y, X), Y in tree and X not:**
    - **choose X and Y such that path start, … , Y,X has minimum weight of all possible X and Y**

# Dijkstra's Algorithm

- **But what if some other path is shorter?**
  - **Other path must include some vertices in tree, some not in tree**
  - **Let (A,B) be arc in this shorter path such that A is in tree and B is not**
  - **Path start, …,A,B is longer than path we have found start, … , Y, X so path start,…,A,B,…,X must be longer than path start, … , Y, X**