CS151 Intro to Data Structures

Sets, Graphs

Announcements

HW08 due 12/14

- Dropping lowest homework assignment so no penalty for not submitting HW08

Final – Self-Scheduled

Last day to resubmit assignments is end of finals period

Outline

Sets

Graphs

Set

A set is an unordered collection of elements, without duplicates

A set supports an efficient search

A hashtable is a set

A multi-set (bag) allows duplicates

A multi-map allows the same key to be mapped to multiple values

set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$
\begin{aligned}
S \cup T &= \{e : e \text{ is in } S \text{ or } e \text{ is in } T\}, \\
S \cap T &= \{e : e \text{ is in } S \text{ and } e \text{ is in } T\}, \\
S - T &= \{e : e \text{ is in } S \text{ and } e \text{ is not in } T\}.
\end{aligned}
$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

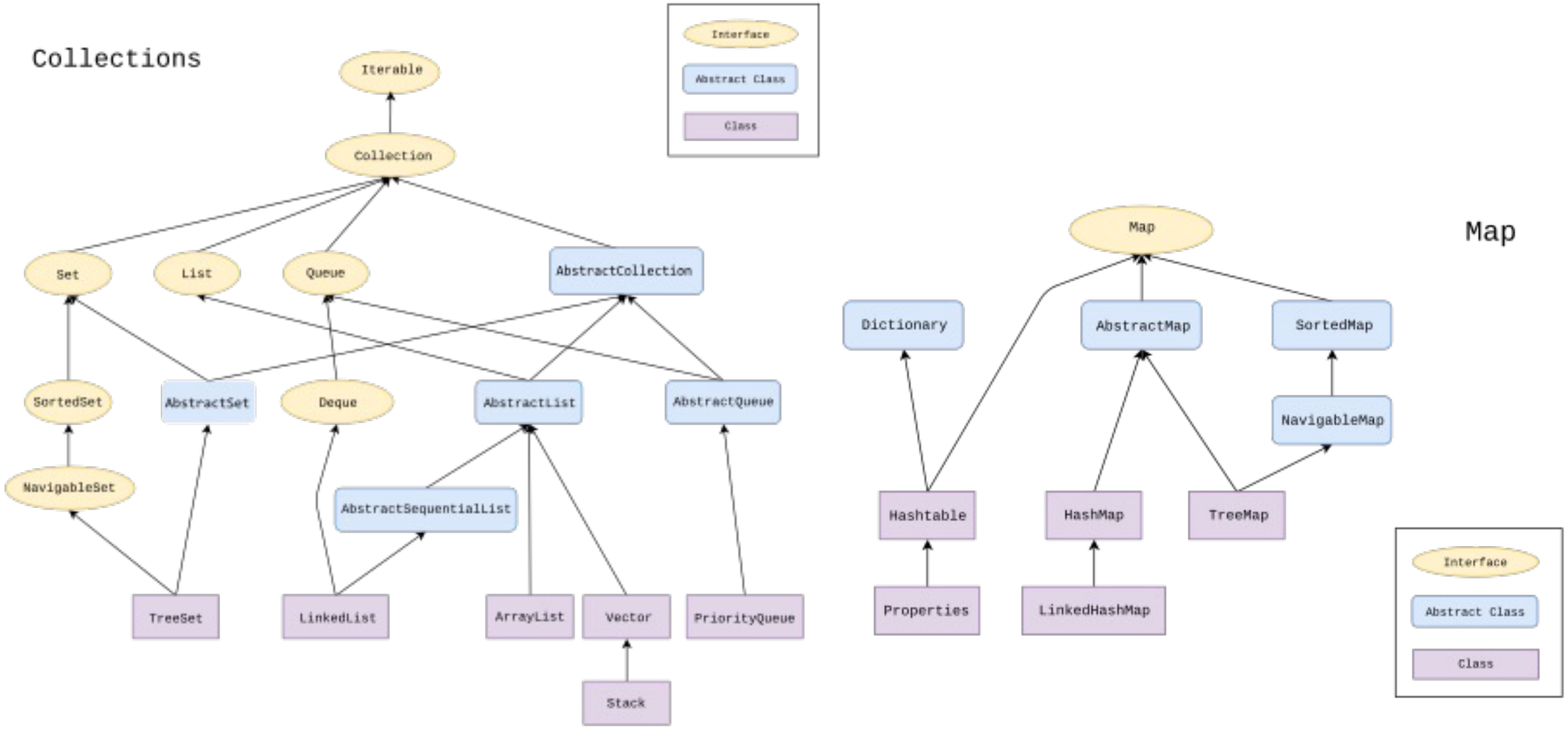
Implementation

- Recall that maps do not allow duplicate keys
- A set is simply a map in which keys have no associated values (or null)
- `java.util.HashSet`
- `java.util.Concurrent.ConcurrentSkipListSet`
- `java.util.TreeSet`

Java Built-ins: `java.util.*`

- Linked List
  - `LinkedList`
- Stack
  - `Stack` (linked)
- Queue
  - `ArrayDqueue`
- BST (unbalanced)
  - none
- Heap
  - `PriorityQueue`
- Hashtable
  - `HashMap` (chained)
- Set
  - `HashSet`
- Balanced BST
  - `TreeMap` (R&B)
- Search/Sort
  - `Collections.binarySearch`
  - `Collections.sort`

Framework Diagram



Outline

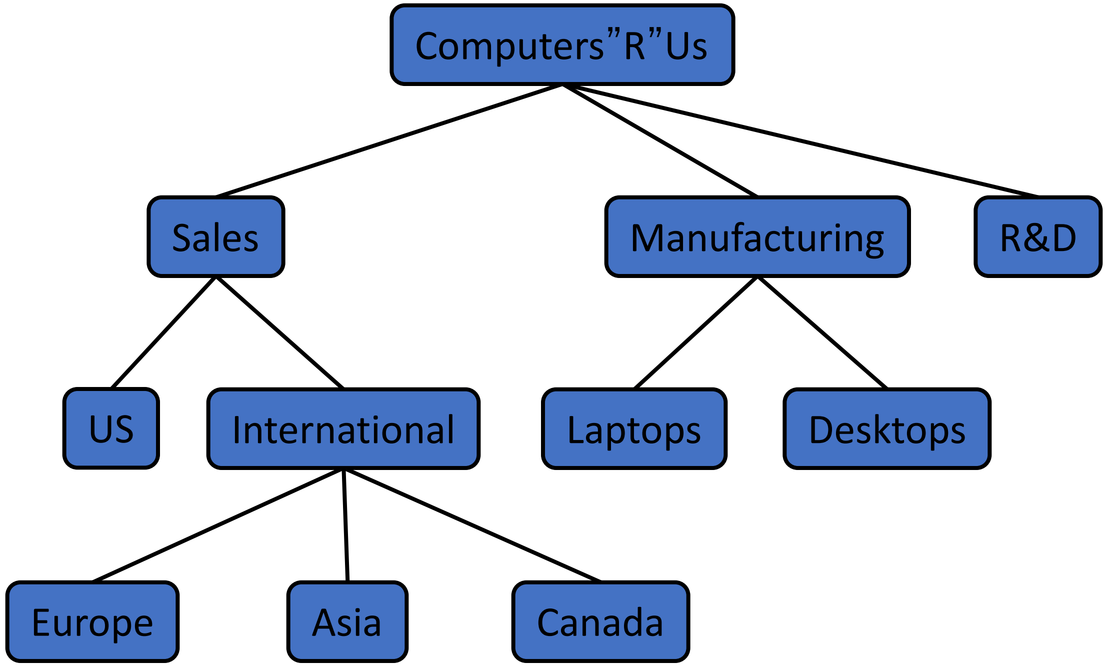
Sets

Graphs

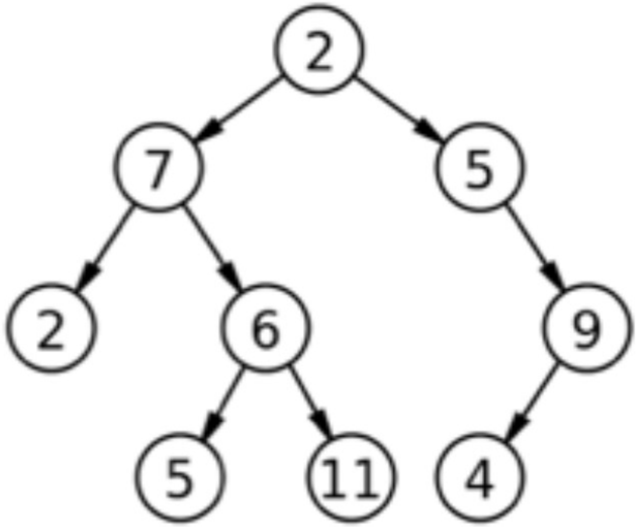
Tree

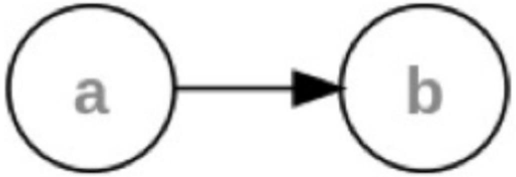
A tree is an abstract model of a hierarchical structure

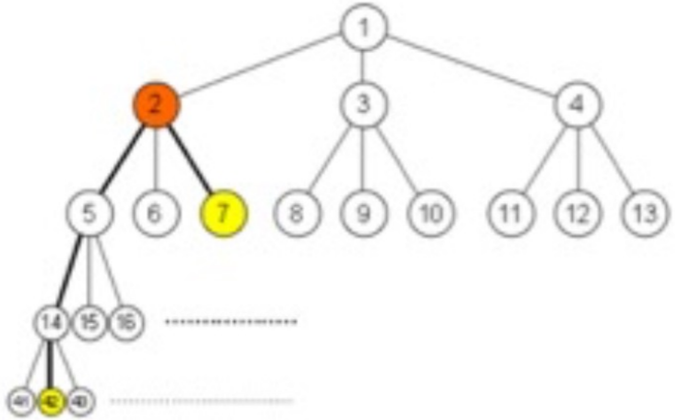
Nodes have a parent-child relation



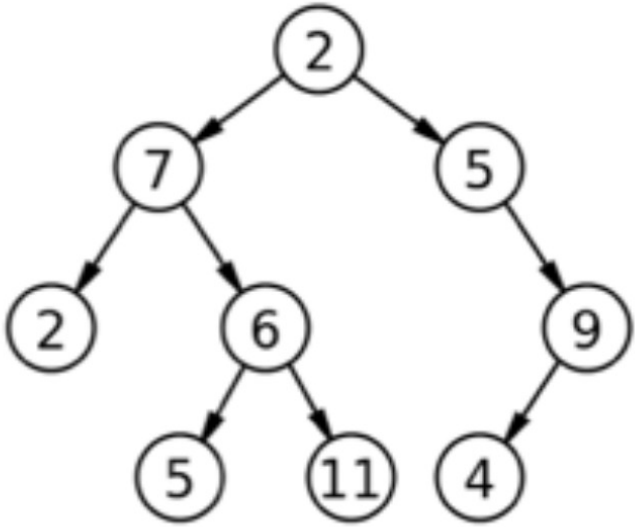
Types of Trees



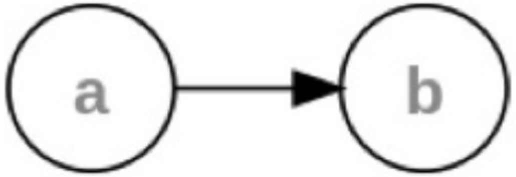




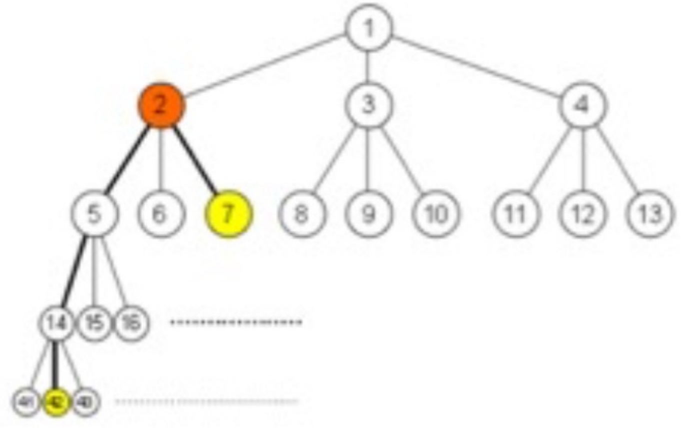
Types of Trees



Unordered
Binary tree



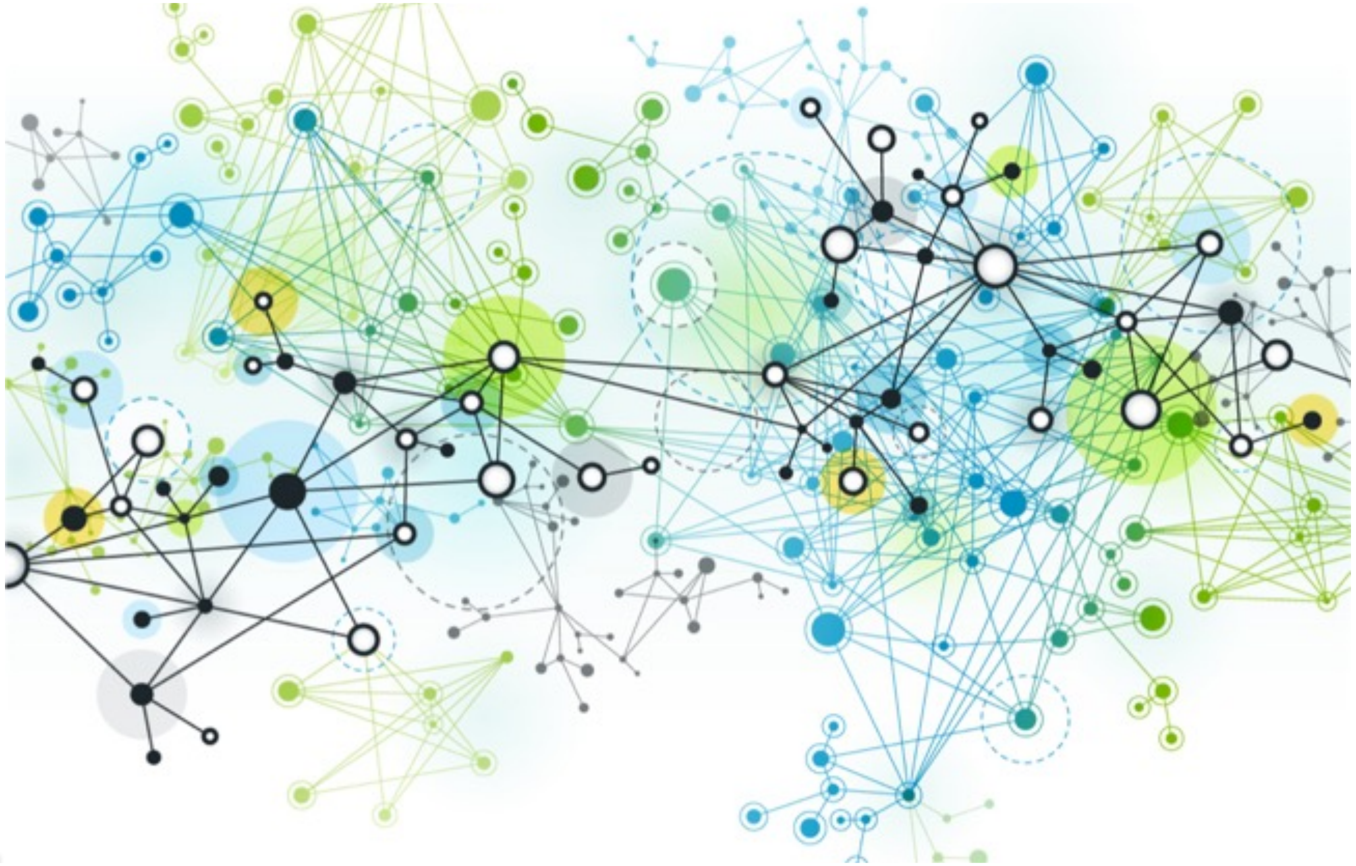
Linear
List



3-ary Tree
(k-ary tree has k children)

Graphs are everywhere!

- Computers in a network,
- Friends on Facebook,
- Roads & Cities on GoogleMaps,
- Webpages on Internet,
- Cells in your body,
- …



1

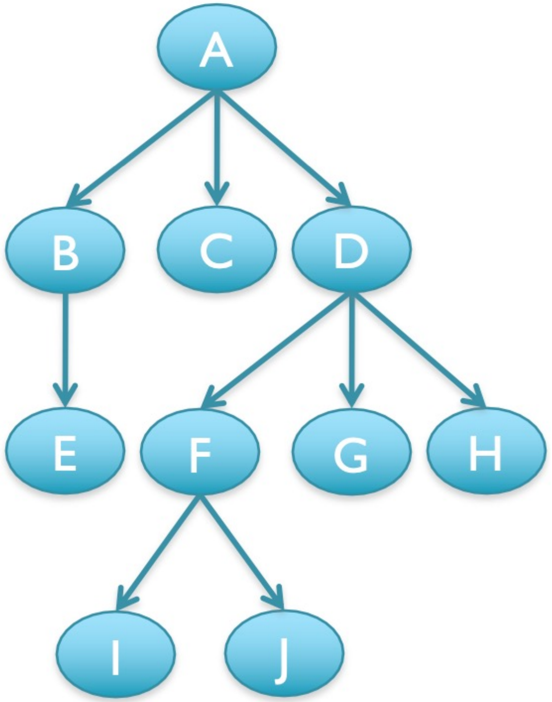
Graphs

- Node aka vertices
  - People, Cities, Friends, …

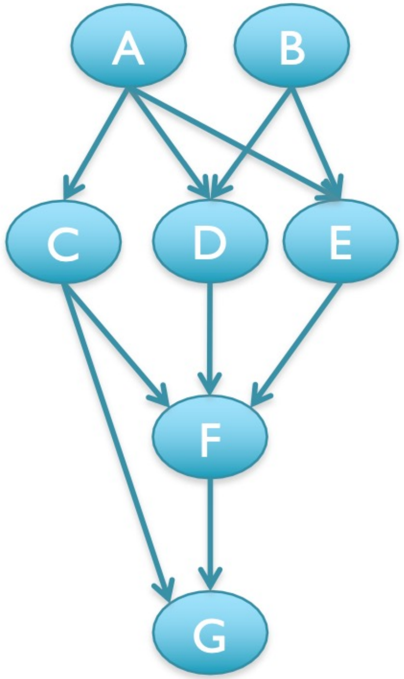
- Edges aka arcs
  - A is connected to B
  - A is related to B
  - A activates B
  - A interacts with B
  - …

Types of Graphs

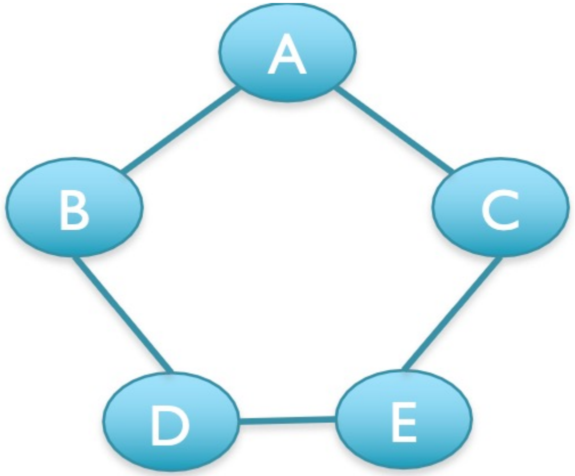
List



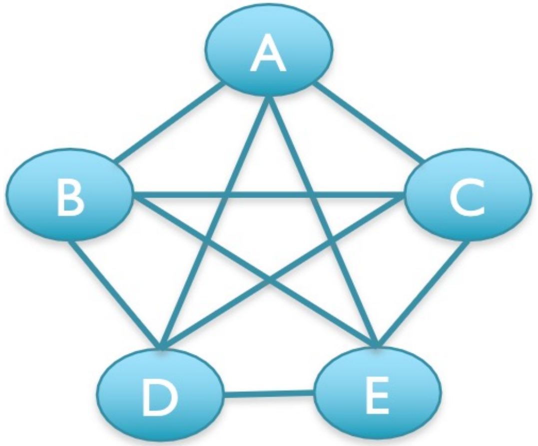
Tree



Directed
Acyclic
Graph

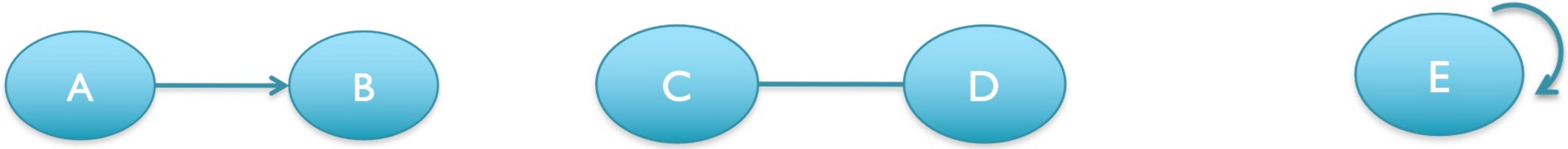


Cycle



Complete

Definitions



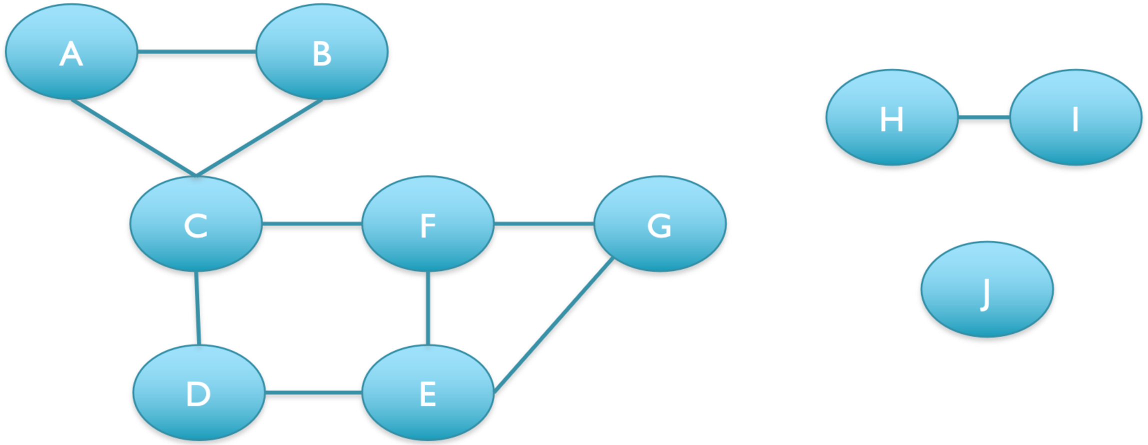
Directed Edge

Undirected Edge

Self Edge
(Unusual but usually allowed)

- A and B are **adjacent**
- An edge connected to a vertex is **incident** on that vertex
- The number of edges incident on a vertex is the **degree** of that vertex
- For directed graphs, we report the **indegree** and **outdegree**
- A **multigraph** allows multiple edges between the same pair of nodes, a **simple** graph does not (most common)

Definitions



- A **path** is a sequence of edges $\mathrm{e}_1 \mathrm{e}_2, \ldots e_n$ in which each edge starts from the vertex where the previous edge ended
- A path that starts and ends at the same node is a **cycle**
- The number of edges in a path is called the **length** of the path
- A graph is **connected** if there is a path between every pair of nodes, otherwise it is **disconnected** into >1 **connected components**

ADT

- numVertices(): Returns the number of vertices of the graph.
- vertices(): Returns an iteration of all the vertices of the graph.
- numEdges(): Returns the number of edges of the graph.
- edges(): Returns an iteration of all the edges of the graph.
- getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
- endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
- outDegree(v): Returns the number of outgoing edges from vertex v.
- inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

ADT

outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.

incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

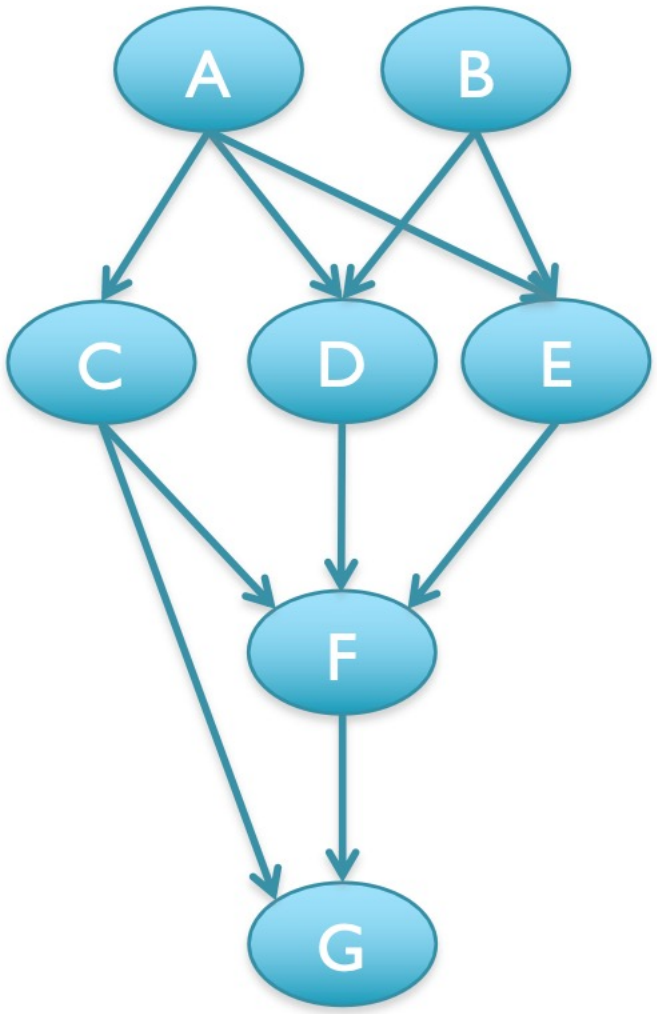
insertVertex(x): Creates and returns a new Vertex storing element x.

insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.

removeVertex(v): Removes vertex v and all its incident edges from the graph.

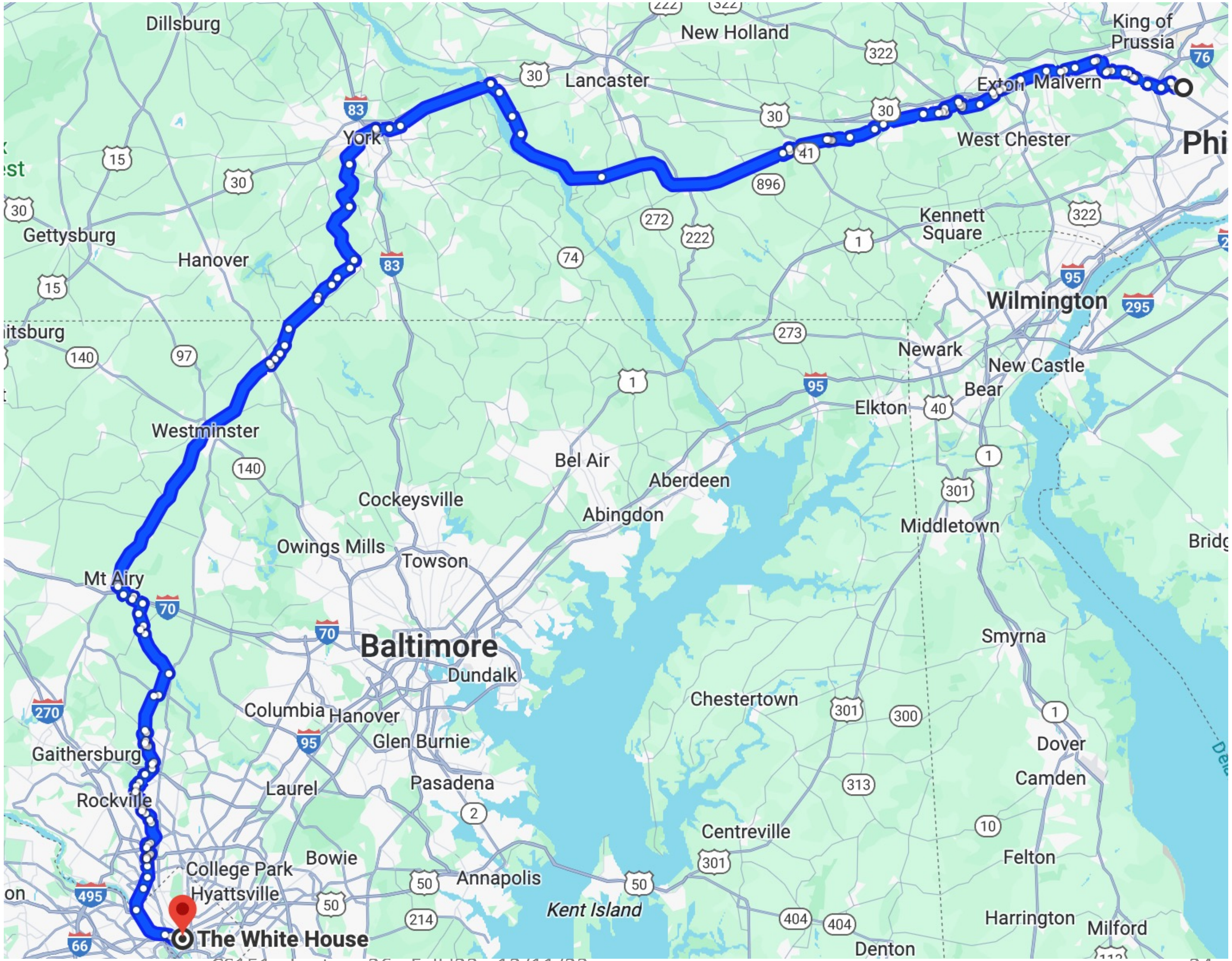
removeEdge(e): Removes edge e from the graph.

Implement a graph

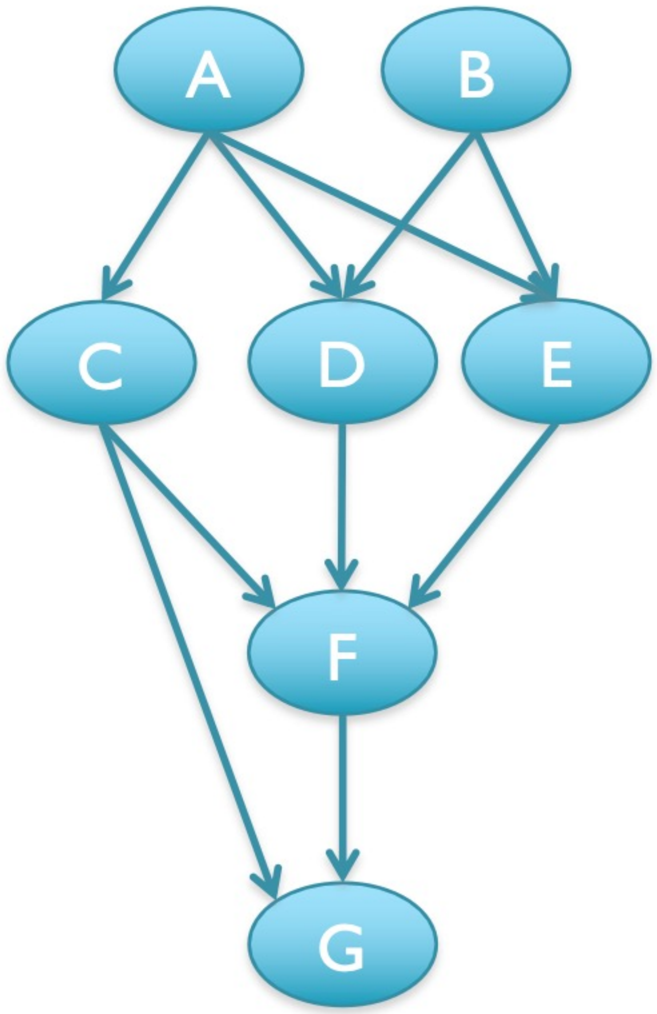


Finding a path

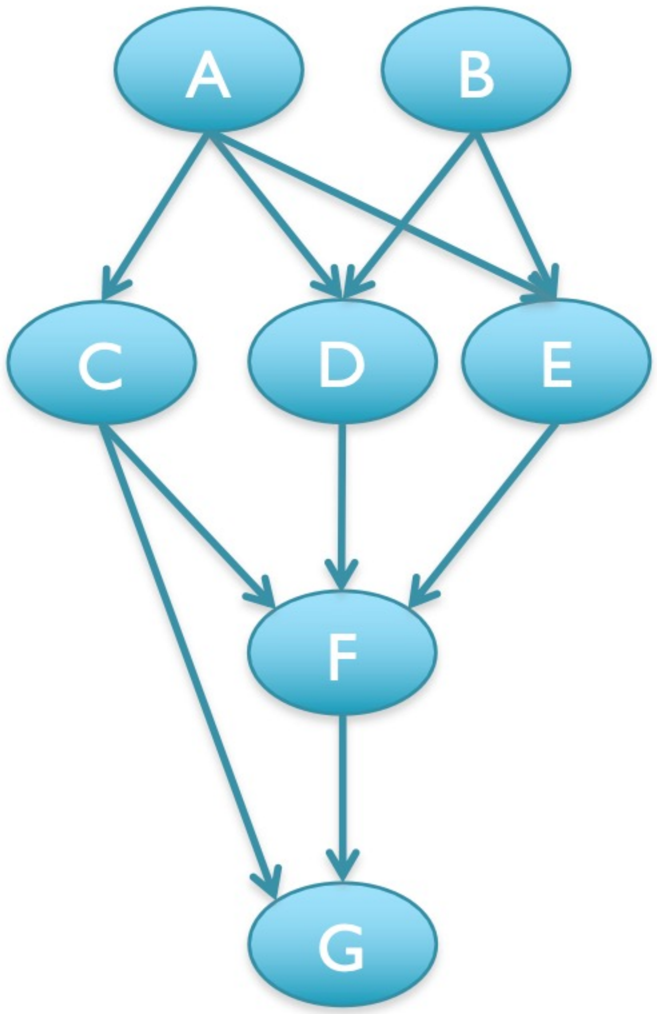
Want to find a path from **source** to **target**



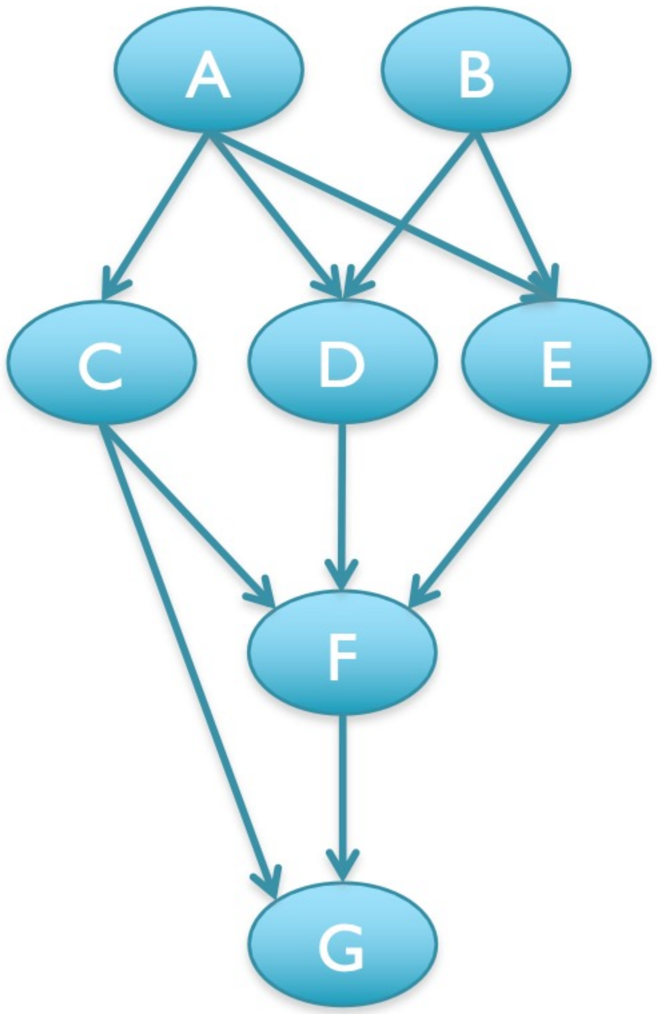
Find a path from A -> G



Find a path from A -> B



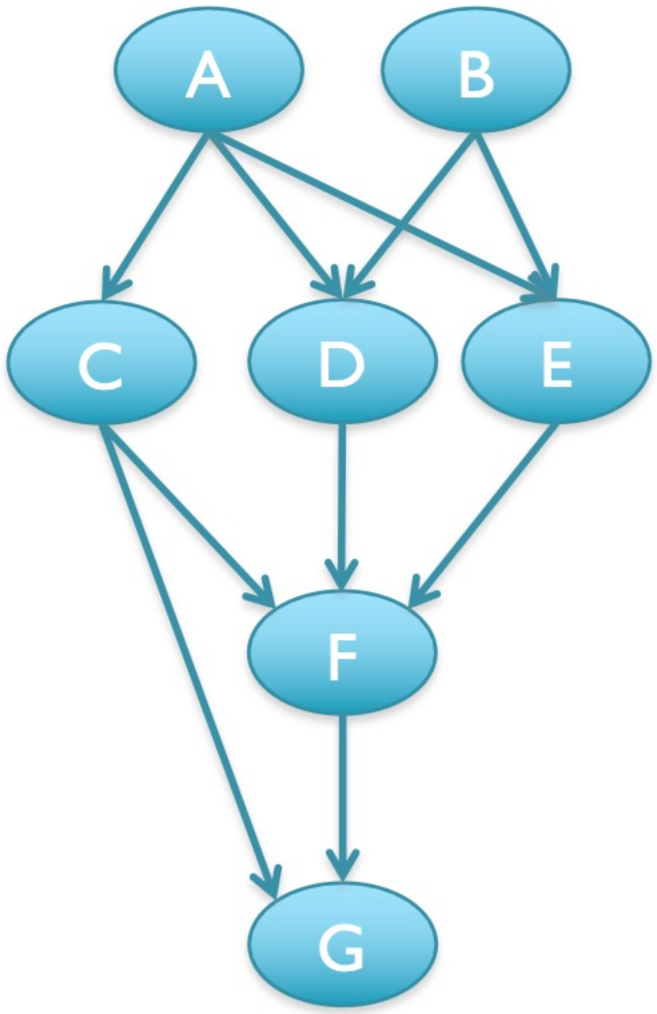
Path finding algorithim



Graph Traversals

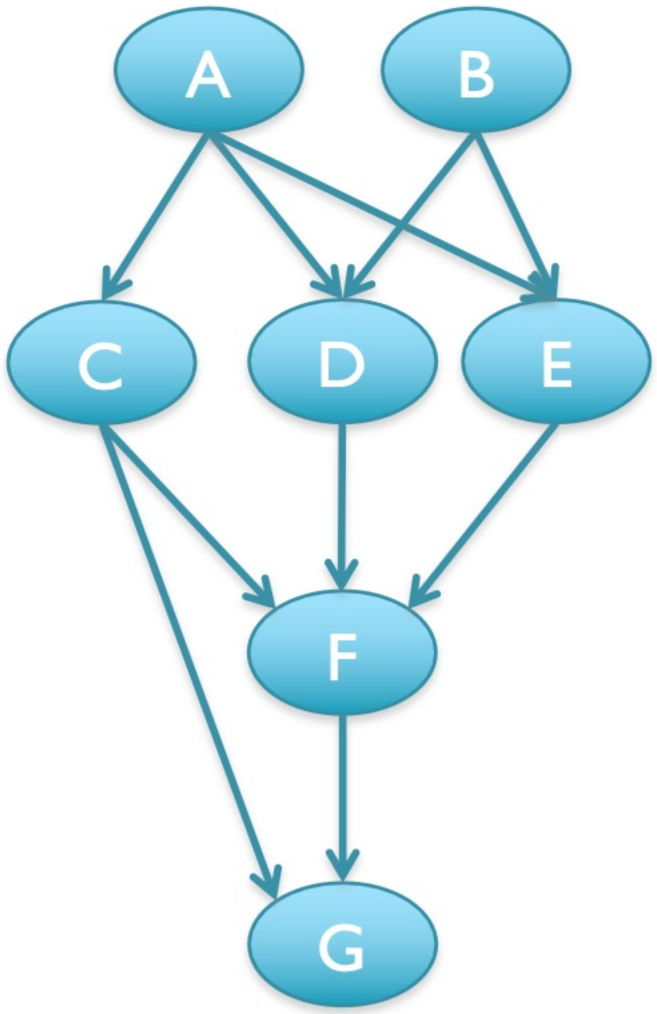
- Breadth-First Search
  - We've seen this so far in tree
- Depth-First Search

Depth-First Search




```
DFS(src, tgt):
        list.insert (src)
        while (!list.isEmpty()):
                curr = list.remove();
                if curr == tgt:
                        found! //stop
                foreach node n adject to curr:
                        list.insert(n)
```

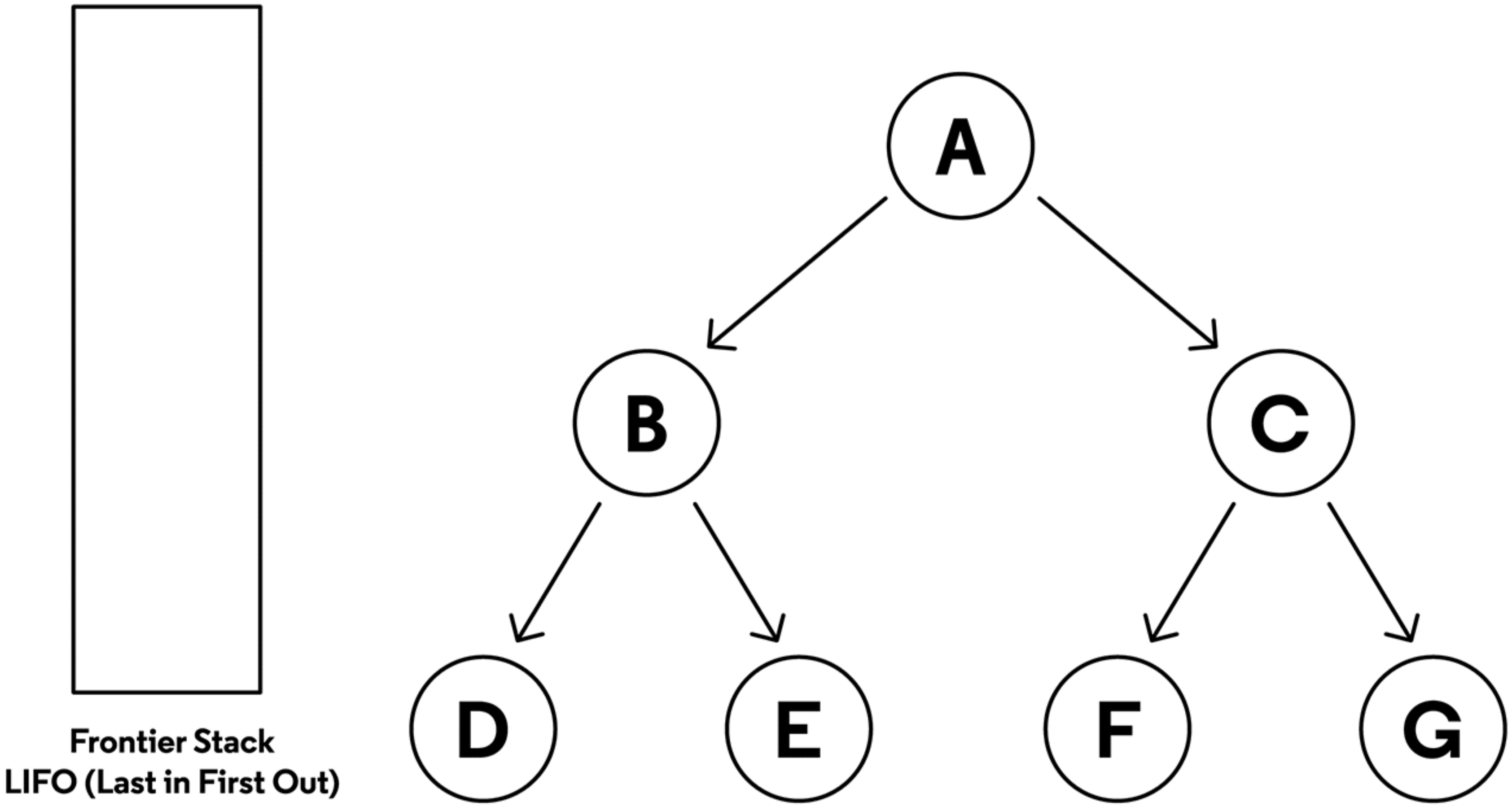
Depth-First Search




```
DFS(src, tgt):
        list.addEnd(src)
        while (!list.isEmpty()):
                curr = list.end();
                if curr == tgt:
                        found! //stop
                foreach node n adject to curr:
                        list.addend(n)
```

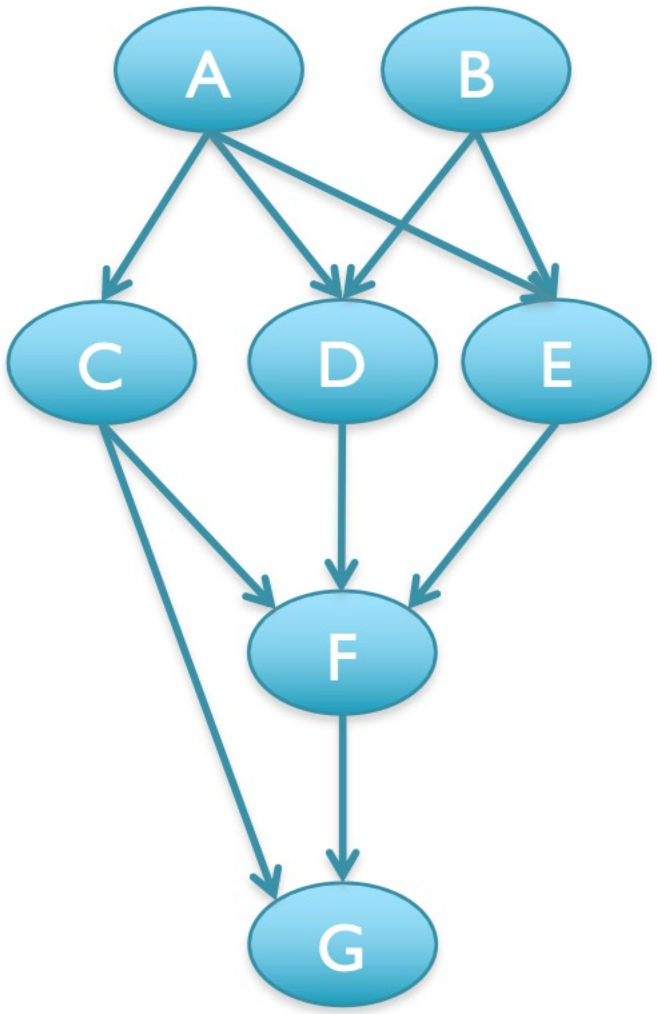
• ł

Tree with an Empty Stack



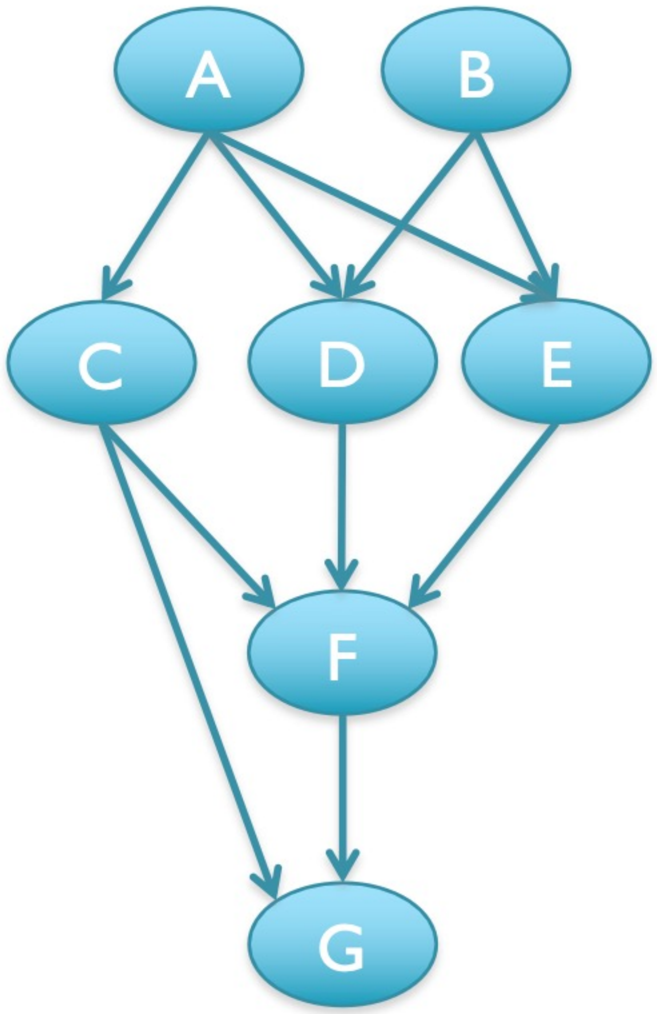
https://www.codecademy.com/article/tree-traversal

Depth-First Search




```
DFS(src, tgt):
        list.addEnd(src)
        while (!list.isEmpty()):
                curr = list.end();
                if curr == tgt:
                        found! //stop
                foreach node n adject to curr:
                        list.addend(n)
```

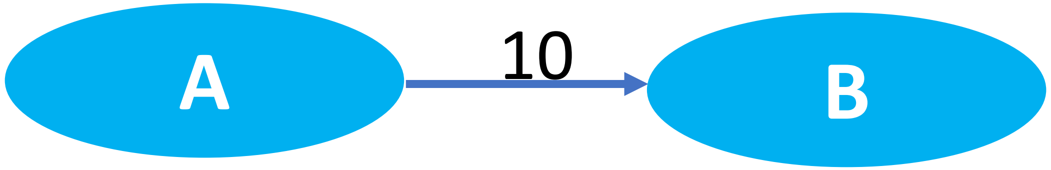
Depth-First Search



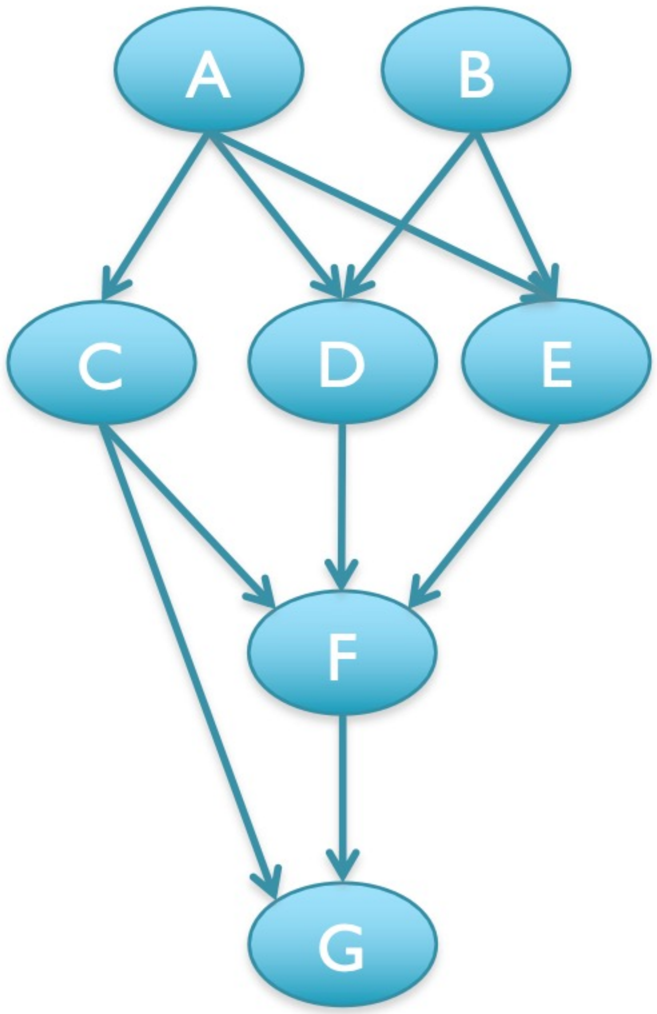

```
DFS(src, tgt):
        list.addEnd(src)
        while (!list.isEmpty()):
                curr = list.end();
                if curr == tgt:
                        found! //stop
                foreach node n adject to curr:
                        if !n.hasBeenVisited():
                                list.addend(n)
```

Weighted Graphs

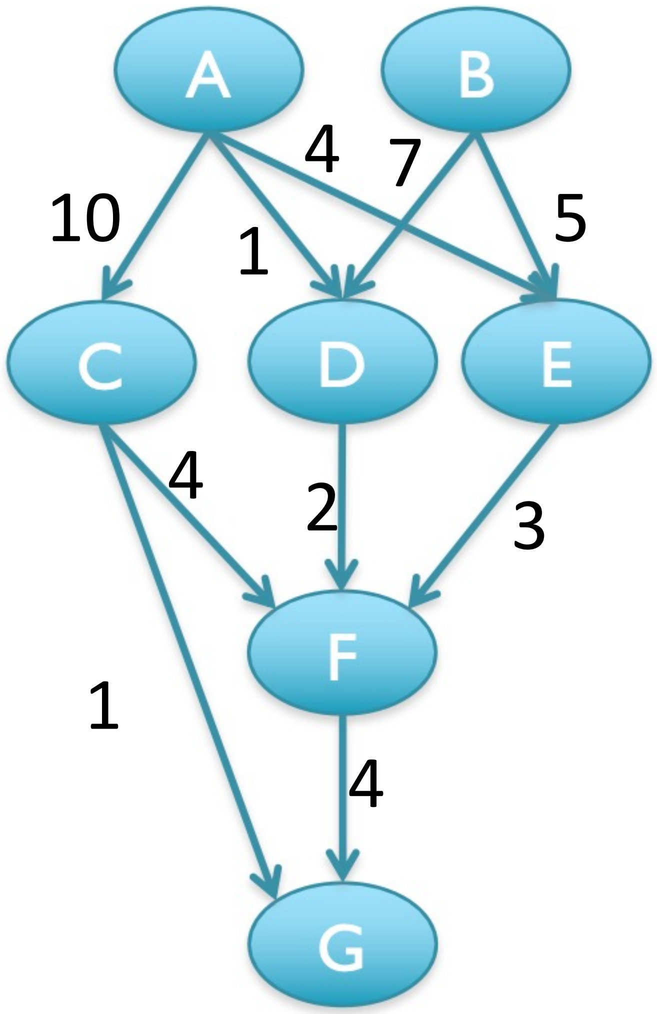
- Edges have weights/costs



Find the shortest path from A -> G

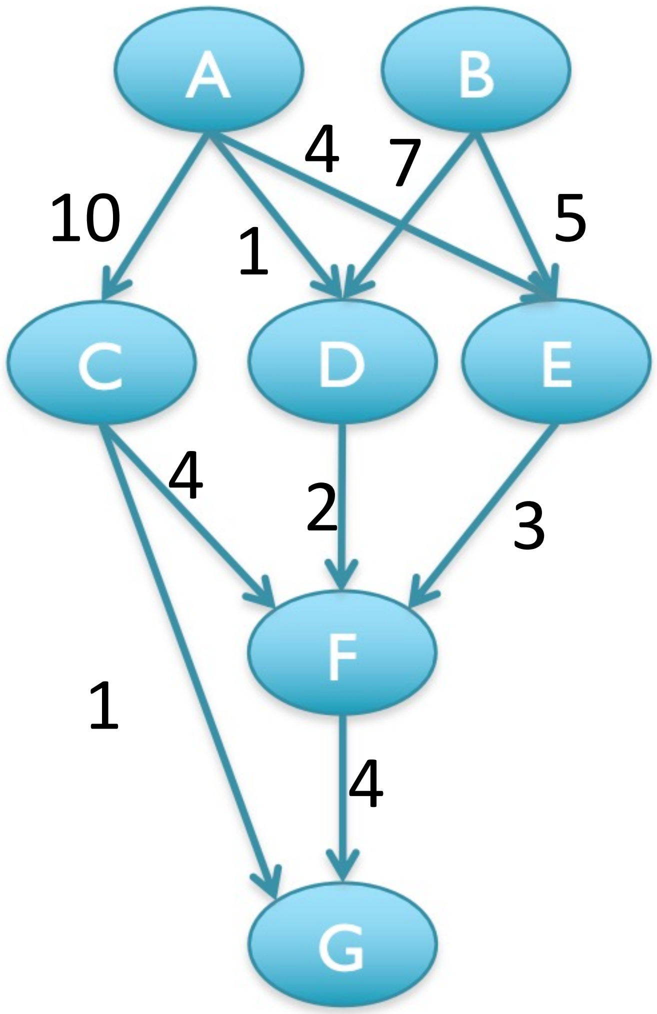


Find the shortest path from A -> G



- Dijkstra’s algorithm
  - Visit the node with the lowest cost

Minimum number of edges to visit all nodes



- Minimum spanning tree