UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Trees and Graphs

Mai Dahshan

September 29, 2024

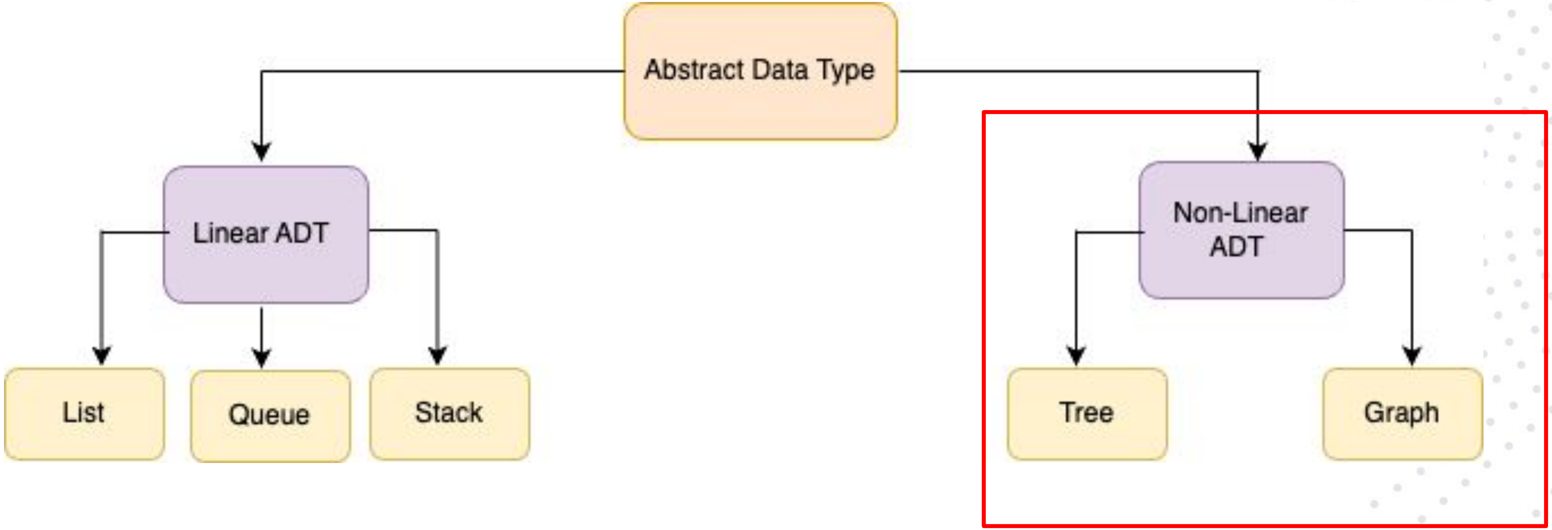
Learning Objectives

- Define and differentiate between linear and non-linear data structures.
- Understand the hierarchical or network-based relationships that characterize non-linear structures, including trees and graphs
- Implement and demonstrate traversal algorithms for non-linear data structures
- Applications of non-linear data structures

Non-Linear Data Structure

- A non-linear data structure does not arrange data in a sequential manner, unlike linear data structures (e.g., arrays, linked lists)
- **Key Features**:
  - Multiple levels of data (hierarchical relationships)
  - More complex relationships between elements
  - Useful for representing real-world scenarios like hierarchical structures or networks

Non Linear ADTs



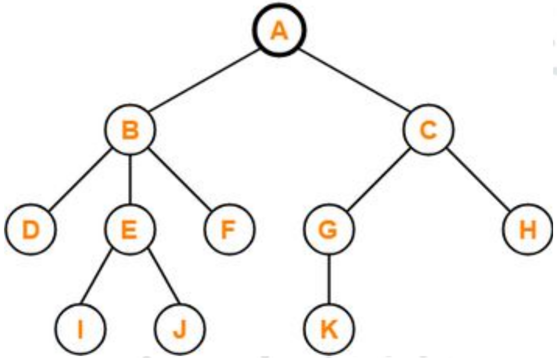
UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Trees

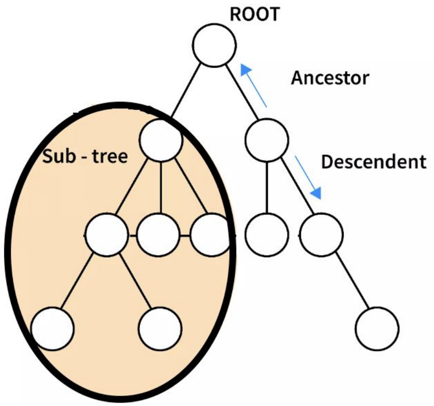
Tree ADT

- **Tree Abstract Data Type (ADT)** structure data hierarchically based on relationships between the data elements. It abstracts the operations that can be performed on the tree, without specifying the details of how these operations are implemented



Tree ADT

- Key characteristics of Tree ADT
  - **Hierarchical Structure**
  - **Parent-Child Relationship**
  - **Subtrees**



Tree ADT Operations

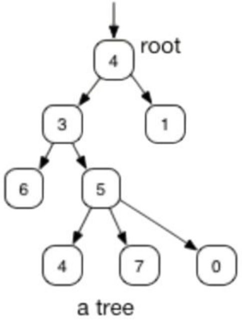
- Key Operations of a Tree ADT
  - **Insertion**: Add a new node to the tree while maintaining the tree structure
  - **Deletion**: Remove a node from the tree, and adjust the tree to maintain its structure
  - **Traversal**: Visiting all nodes in the tree systematically
  - **Search**: Find a node with a specific value

Tree ADT Operations

Operation/Interface	Descriptions
root()	Return the tree's root; error if tree is empty
parent(v)	Return v's parent; error if v is a root
children(v)	Return v's children (an iterable collection of nodes)
isRoot(v)	Test whether v is a root
isExternal(v)	Test whether v is an external node
isInternal(v)	Test whether v is an internal node

Trees

- Trees are composite, hierarchical and graph-like data structures
- A Tree is a collection nodes with a distinguished node, the root. All other nodes form a set of disjoint subtrees, in which each is a tree in its own
- In Computer Science, trees grow *down*, not up!



Tree in DS

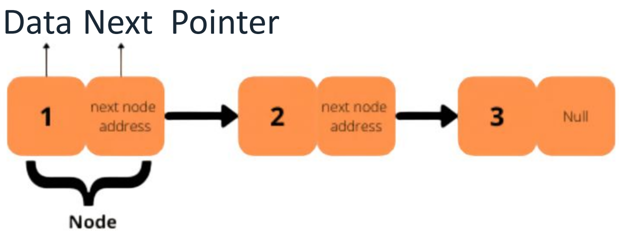
Physical Tree

Recursive Definition of Trees

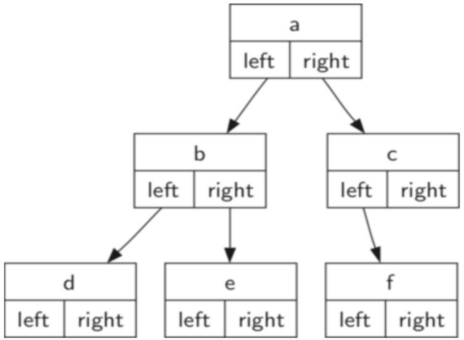
- A **recursive data structure** is a data structure that is defined in terms of itself. This means that the structure contains smaller instances of the same type of structure, which allows for self-referential definitions and recursive algorithms to process or manipulate it.
- **Recursive Definition**: A tree is a collection of nodes where it is either:
  - An **empty tree**, which is a tree that contains no nodes.
  - A **non-empty tree**, which consists of a root node and zero or more subtrees.
    - A **root node** (which may contain a value or data).
    - A collection of **subtrees**, each of which is itself a tree.

Recursive Definition of Trees

- Examples
  - Each node in a **linked list** contains a reference to another node
  - Each node in a **tree** can have references to other nodes



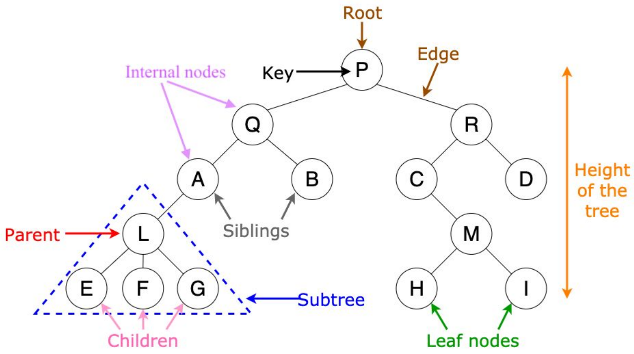
Linked List recursive DS



Tree recursive DS

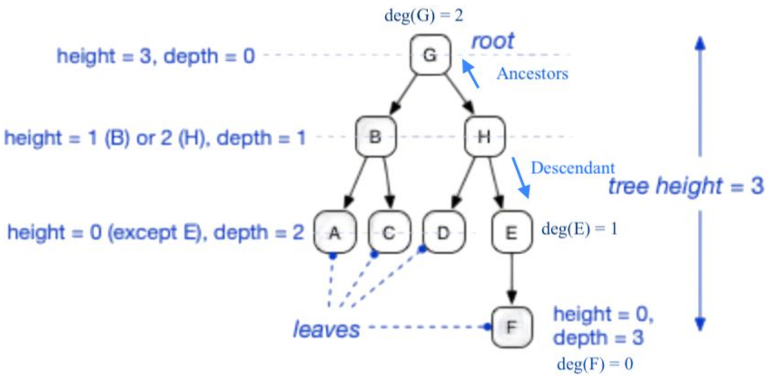
Trees

- Trees are composed of
  - **Nodes**
    - are elements in the data structure that hold data
    - have only one parent (**unique predecessor**)
    - have zero, one, or more children (**successors**)
    - Nodes with same parent are siblings
    - **Root** node: **top** or start node; with no parent
    - **Leaf** nodes (**external** nodes): nodes without children (***terminal***)
    - **Internal** node: nodes with children (***non-terminal***)
  - **Edges**
    - Link parent node with children node (if applicable)



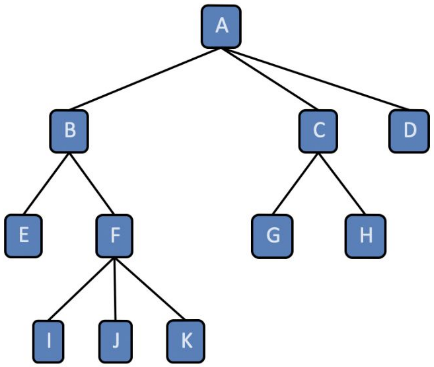
Trees

- The **degree** of a node is the number of its children
- **Depth** of a node: the length of the (unique) path from the root down to that node. The length of a path is the number of edges in the path.
- **Height** of a node: the length of the longest path from the node down to the leaf node.
- **Descendant** of a node: child, grandchild, great-grandchild, etc.
- **Ancestors** of a node: parent, grandparent, great-grandparent, etc.



Quiz

- Answer the following questions about the tree shown below:
    - Classify each node of the tree as a root, leaf, or internal node
    - List the ancestors of nodes B, F, G, and A. Which are the parents?
    - List the descendents of nodes B, F, G, and A. Which are the children?
    - List the depths of nodes B, F, G, and A.
    - What is the height of the tree?



Kinds of Trees

- General Tree (N-ary Tree): A tree in which each node has N (any) number of children
- Binary tree: Each node has at most 2 children (branching factor 2)
    - Binary Search Tree (BST)
    - Balanced Trees(e.g., AVL Tree)
    - Heap
    - K-D Tree
- B-tree
- Decision Tree
- and more

Why Study Trees?

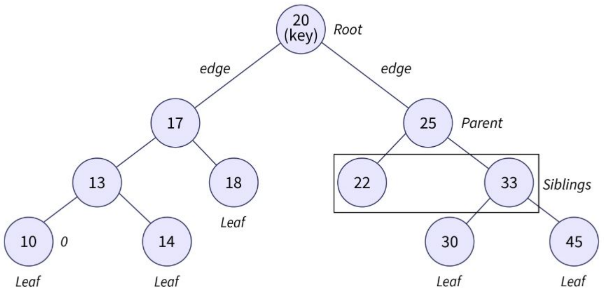
- Data Organization
- Dynamic Data Handling
- Hierarchical Data Representation

Applications of Trees

- Databases
- File Systems
- Artificial Intelligence
- Game Development
- HTML/XML Parsing
- Compiler Design

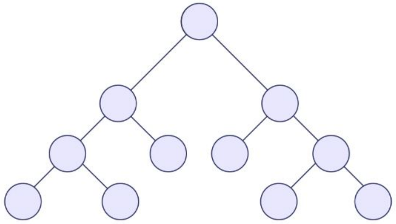
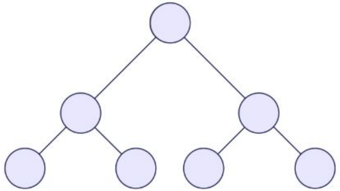
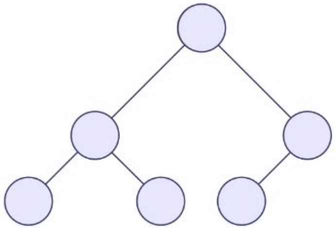
Binary Tree

- A ***binary tree*** is a tree with the following properties:
  - Every node has at most two children
  - Each child node is labeled as being either a ***left child*** or a ***right child***
- **Alternative recursive definition:** a binary tree is either empty or consists of:
  - A node r, called the root of T, that stores an element
  - A binary tree (possibly empty), called the left subtree of T
  - A binary tree (possibly empty), called the right subtree of T



Types of Binary Tree

- **Full Binary Tree** (proper binary tree) is a special type of binary tree in which all non-leaf nodes have exactly two children
- **Perfect Binary Tree** is a special type of binary tree with all the internal nodes have exactly two children and all the leaf nodes are at the same depth
- **Complete Binary Tree** is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.

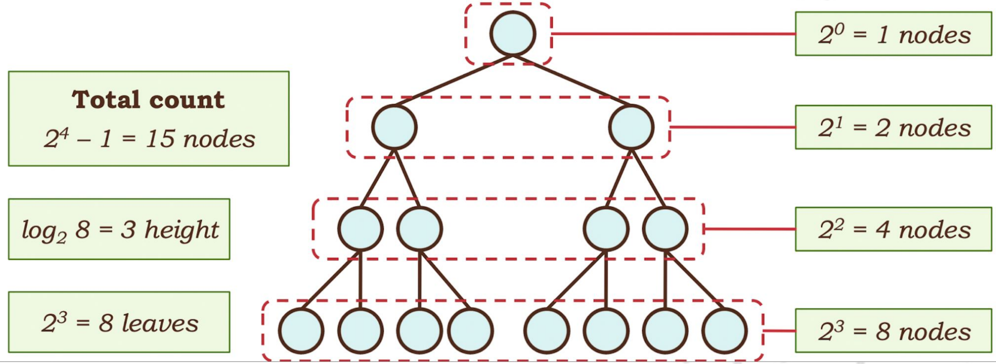




Types of Binary Tree - Quiz

- What is the maximum height of a full binary tree with 11 nodes?
  - 3
  - 5
  - 7
  - 10
  - Not possible to have full binary tree with 11 nodes

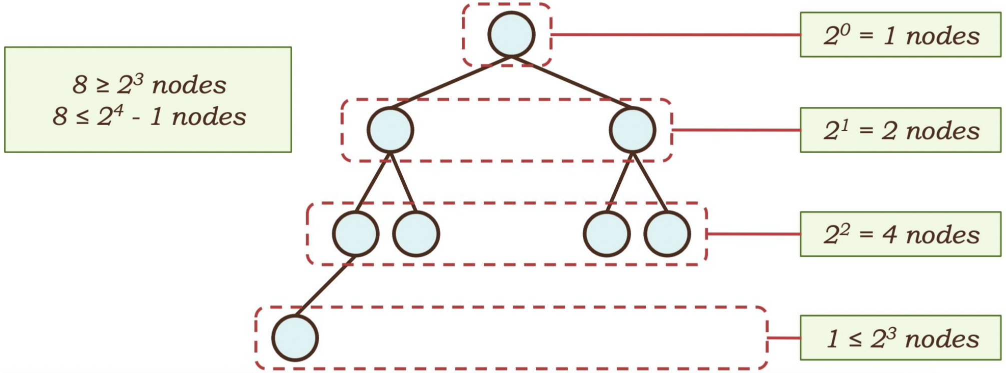
Types of Binary Tree

- A perfect binary tree has
  - 2^L nodes at level L
  - 2^h leaves for height h
  - 2^{h+1} - 1 nodes for height h or 2n-1 nodes for n leaves
  - Height $\log_2 n$ for n leaves



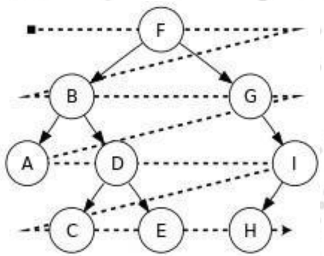
Types of Binary Tree

- A complete binary tree has
  - at most 2^L nodes at level L
  - at least 2^h leaves for height h
  - at most 2^{h+1} - 1 nodes for height h

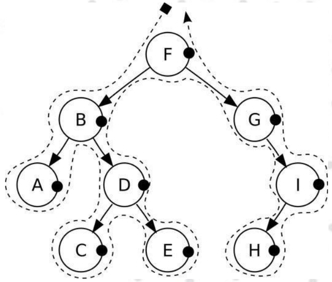


Binary Tree Traversal

- Many algorithms require all nodes of a binary tree be visited and the contents of each node processed or examined
- A **tree traversal** is a specific order in which to trace the nodes of a tree
- Types of traversals
    - Breadth First Traversal
    - Depth First Traversal



Breadth First

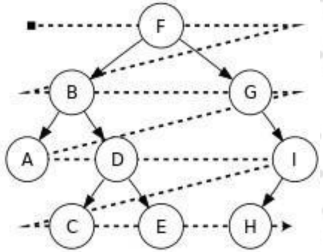


Depth First

Binary Tree Traversal: https://tree-visualizer.netlify.app/

Binary Tree Traversal

- Breadth First Traversal or level order traversal: starting from the root of a tree, process all nodes at the same depth from left to right, then proceed to the nodes at the next depth.



Breadth First

Binary Tree Traversal: https://tree-visualizer.netlify.app/

Binary Tree Traversal

- Depth First Traversal has three variations of traversal
  - preorder traversal: process the root, then process all sub trees (left to right)
  - in order traversal: process the left sub tree, process the root, process the right subtree
  - post order traversal: process the left sub tree, process the right subtree, then process the root

Binary Tree Traversal: https://tree-visualizer.netlify.app/

Binary Tree Traversal

- In Depth First Traversal techniques, the **left** subtree is traversed recursively, the **right** subtree is traversed recursively, and the **root** is visited
- What distinguishes these techniques from one another is ***the order of those 3 tasks***
- Visiting a node entails doing some processing at that node (often it is just **printing** – node label or its data)
- Note "**in**", "**pre**", and "**post**" refer to when we visit the root (of that subtree)

Binary Tree Traversal

- In Preorder, the root is visited **before** (pre) the subtrees traversals
- In Inorder, the root is visited **in- between** left and right subtree traversal
- In Postorder, the root is visited **after** (post) the subtrees traversals

Algorithm for Preorder Traversal

```
1.  if the tree is empty
2.      Return.
    else
3.      Visit the root.
4.      Preorder traverse the left subtree.
5.      Preorder traverse the right subtree.
```

Algorithm for Inorder Traversal

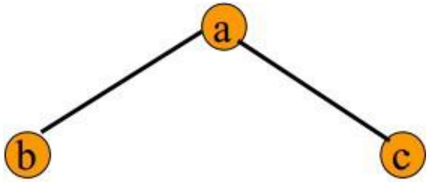
```
1.  if the tree is empty
2.      Return.
    else
3.      Inorder traverse the left subtree.
4.      Visit the root.
5.      Inorder traverse the right subtree.
```

Algorithm for Postorder Traversal

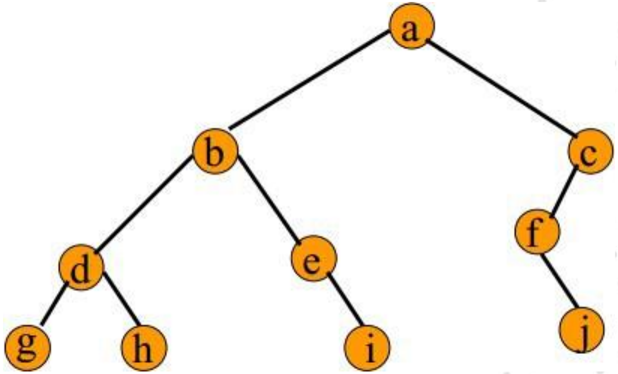
```
1.  if the tree is empty
2.      Return.
    else
3.      Postorder traverse the left subtree.
4.      Postorder traverse the right subtree.
5.      Visit the root.
```

Binary Tree Traversal

- Pre-Order Traversal
  - Prints in order: **root**, left, right



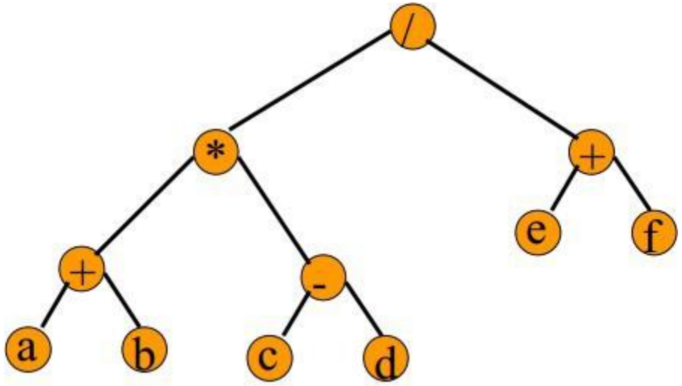
a b c



a b d g h e i c f j

Binary Tree Traversal

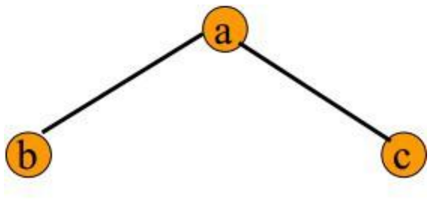
- Pre-Order Traversal
  - Gives **prefix** form of expression



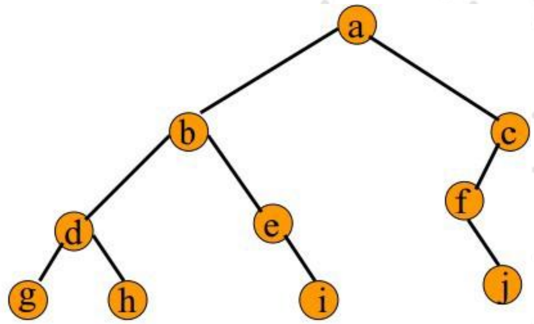
/ * + a b - c d + e f

Binary Tree Traversal

- In-Order Traversal
  - Prints in order: left, **root** , right



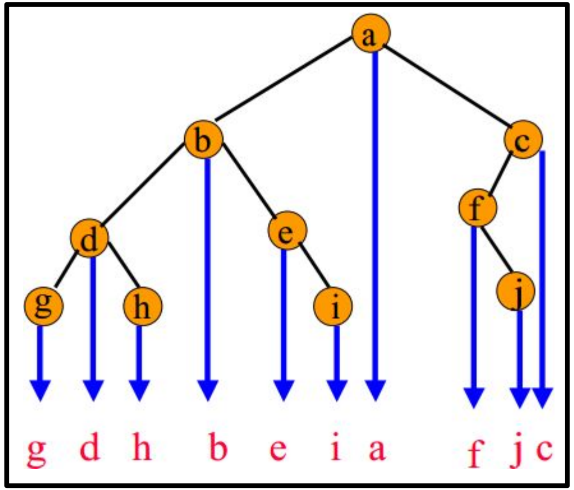
b a c

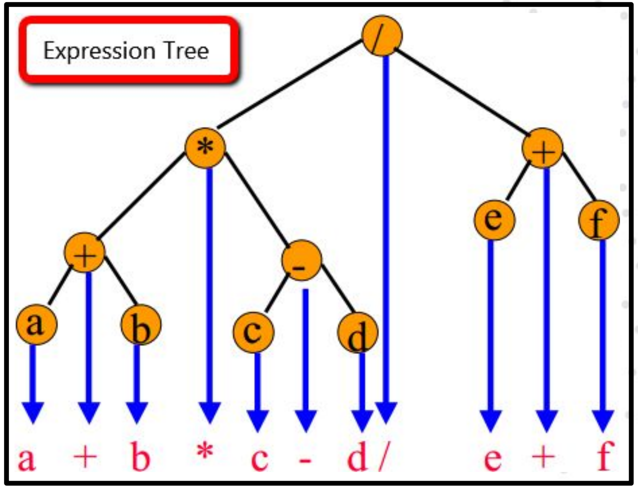


g d h b e i a f j c

Binary Tree Traversal

- In-Order Traversal
  - Gives **infix** form of expression (sans parenthesis)

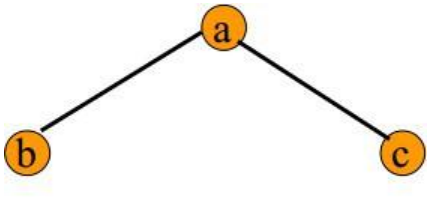




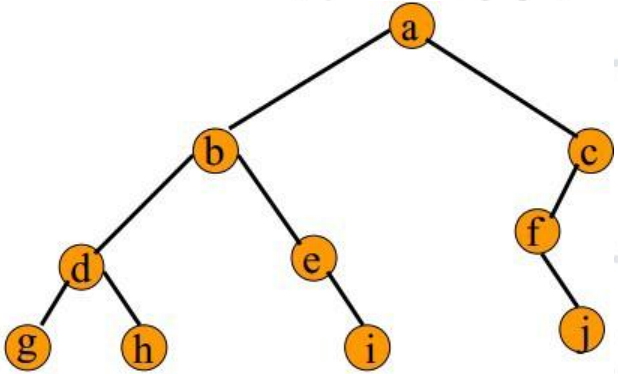
(will revisit this later)

Binary Tree Traversal

- Post- Order Traversal
    - Prints in order: left, right, **root**



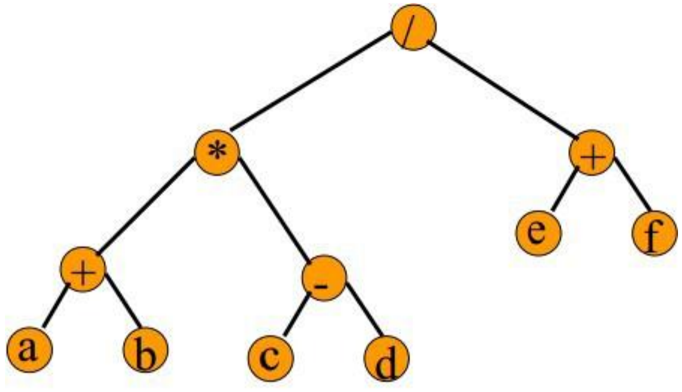
b c a



g h d i e b j f c a

Binary Tree Traversal

- Post-order Traversal
  - Gives **postfix** form of expression

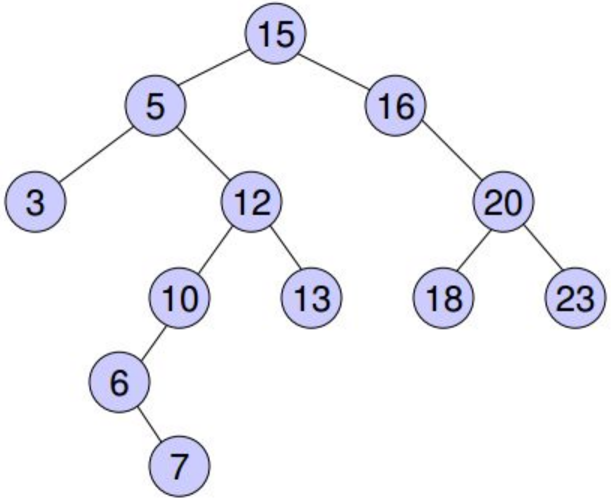


a b + c d - * e f + /

(will revisit this later)

Binary Tree Traversal

- Let's do an example first…

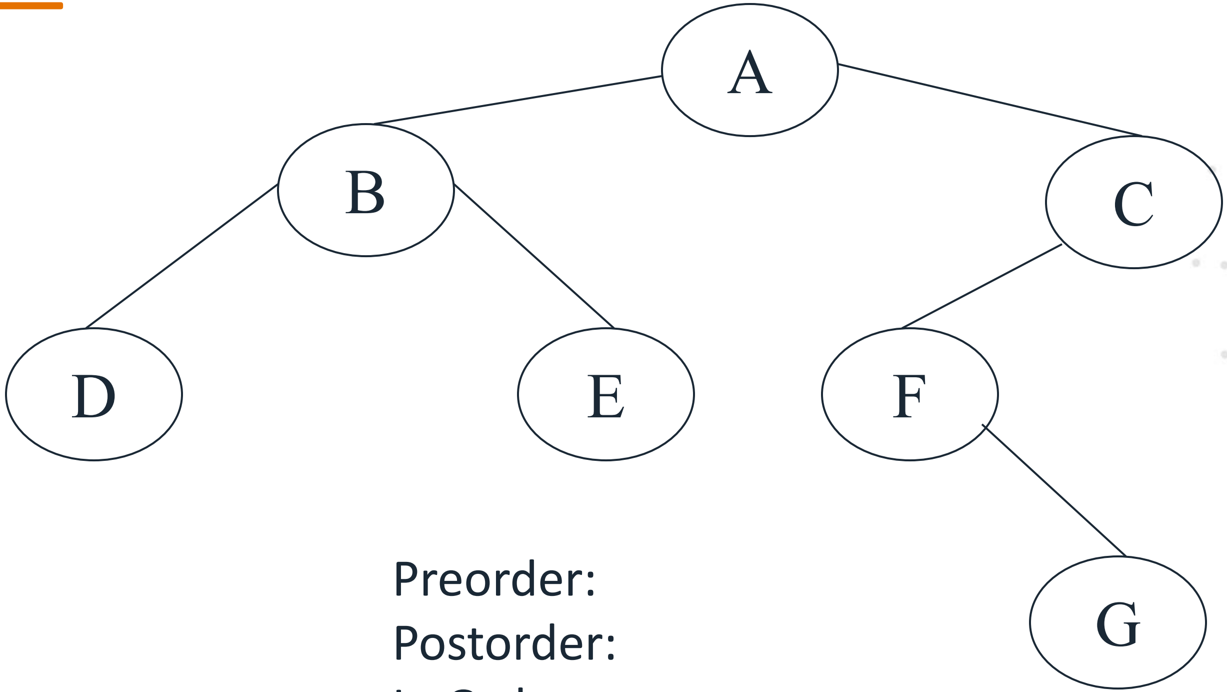


pre-order: (root, left, right)
15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23

in-order: (left, root, right)
3, 5, 6, 7, 10, 12, 13, 15, 16, 18, 20, 23

post-order: (left, right, root)
3, 7, 6, 10, 13, 12, 5, 18, 23, 20, 16, 15

Binary Tree Traversal

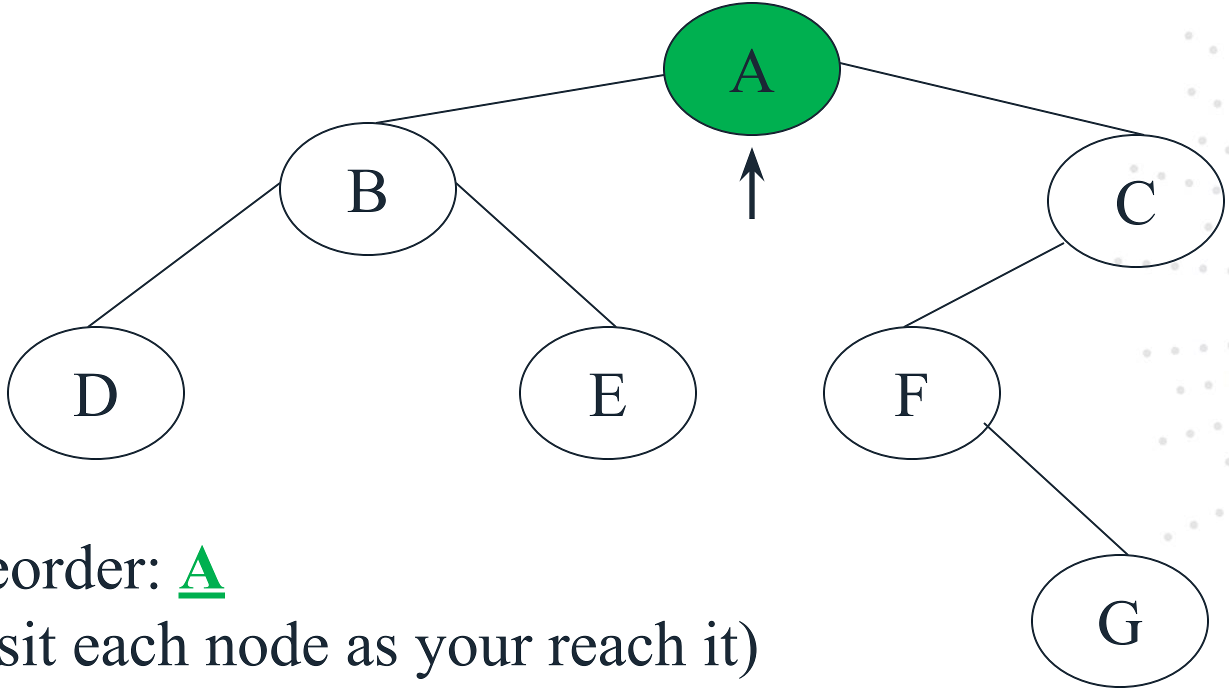


Preorder:

Postorder:

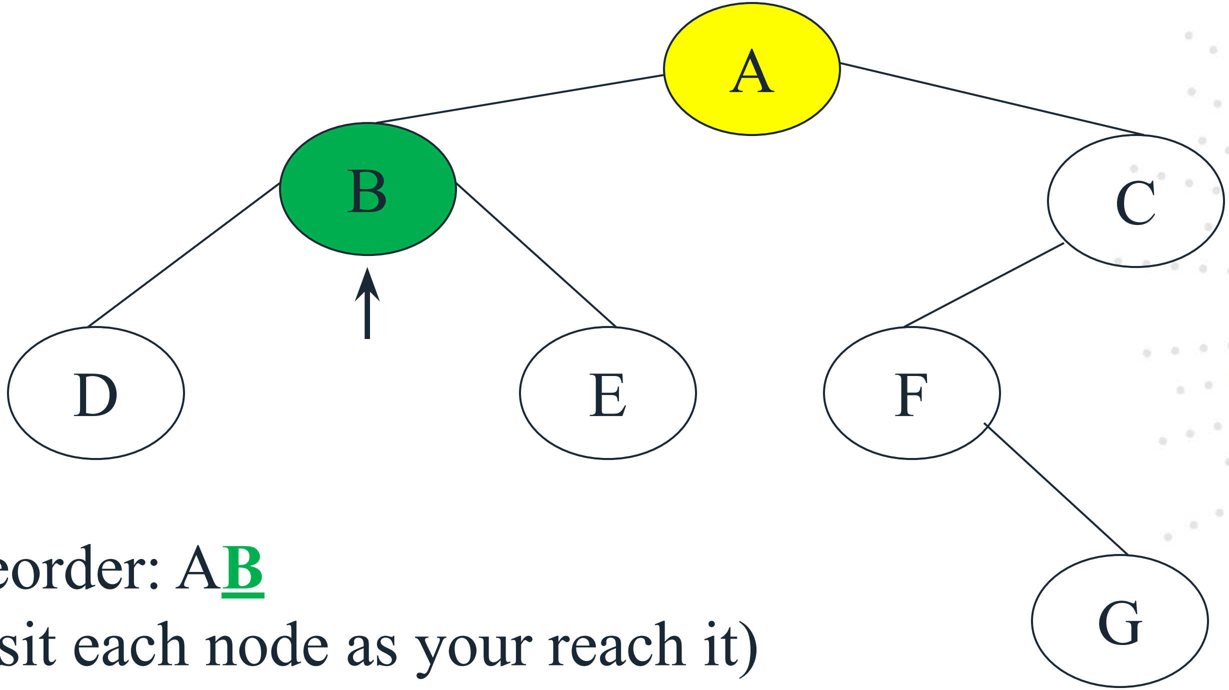
In Order:

Binary Tree Traversal



Preorder: A
(visit each node as your reach it)

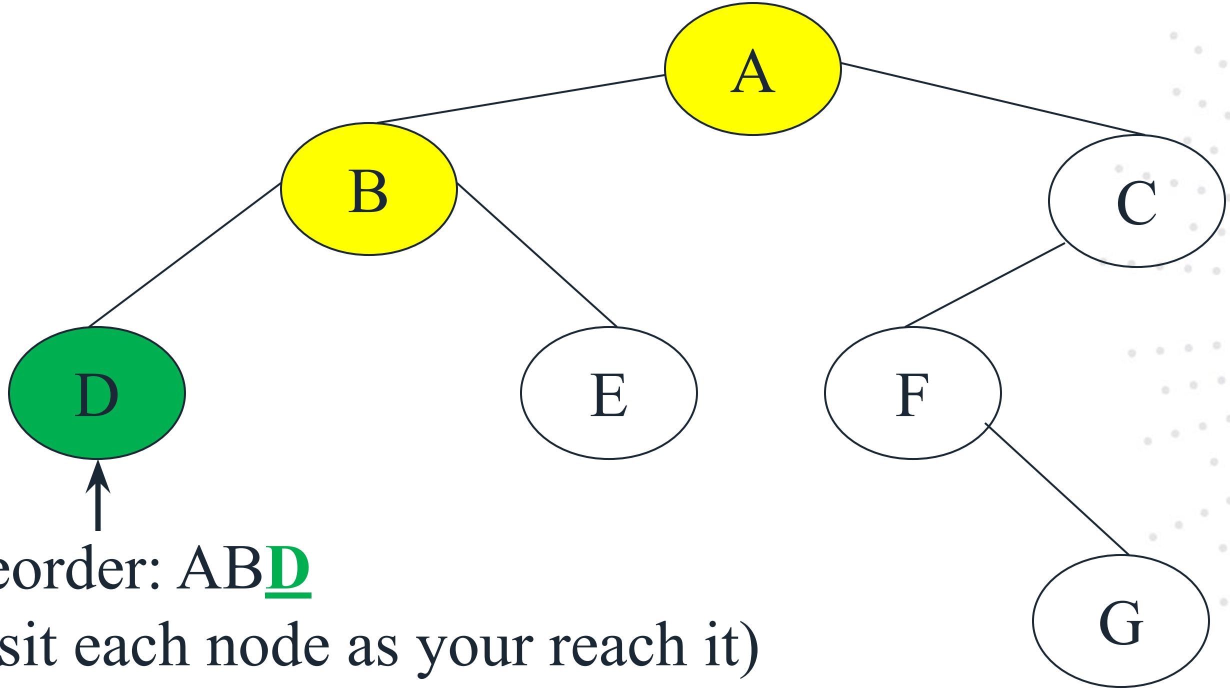
Binary Tree Traversal



Preorder: A**B**

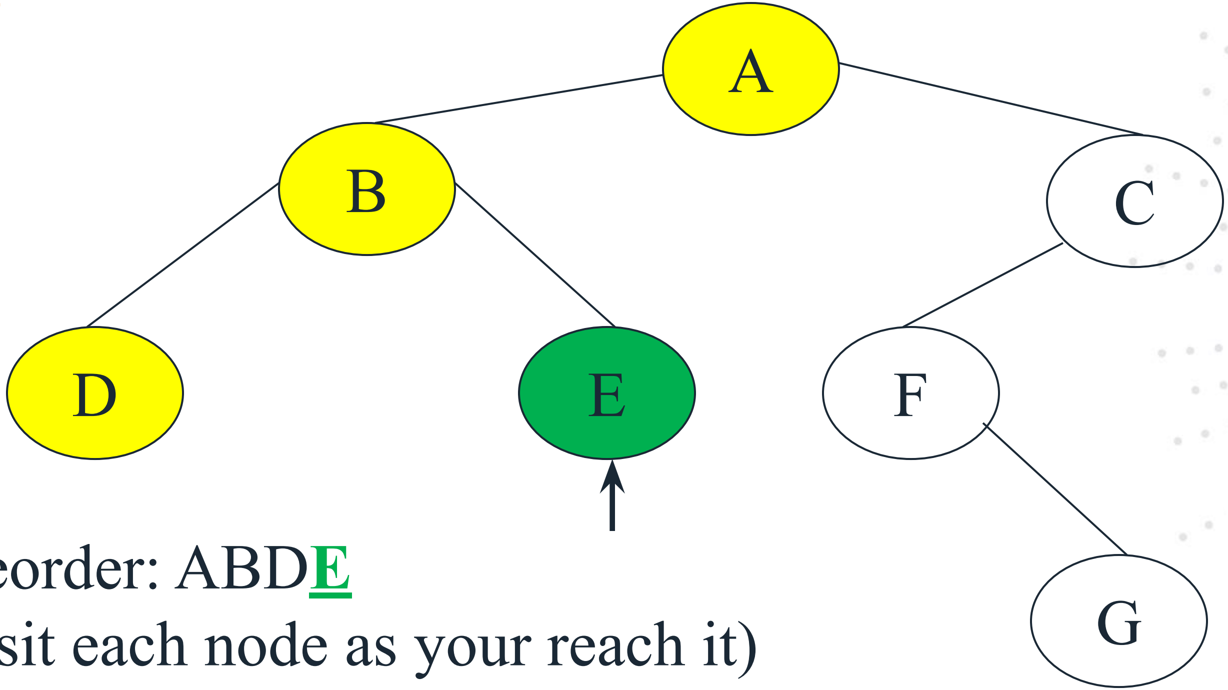
(visit each node as your reach it)

Binary Tree Traversal



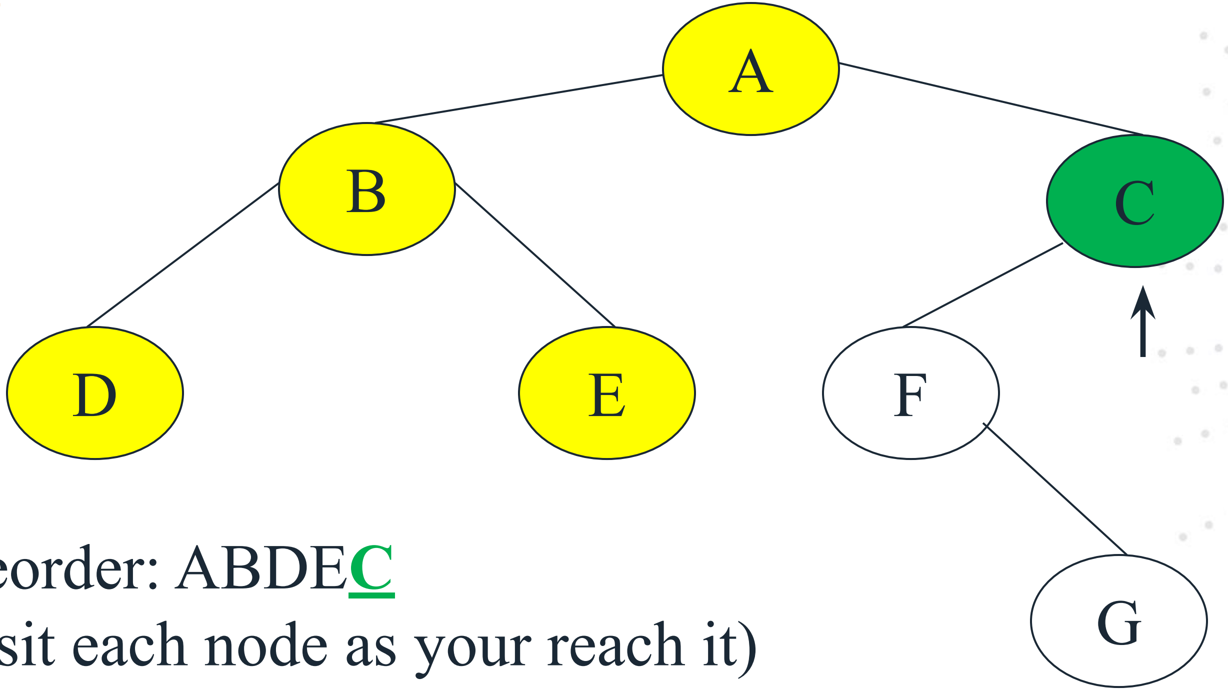
Preorder: AB**D**
(visit each node as your reach it)

Binary Tree Traversal



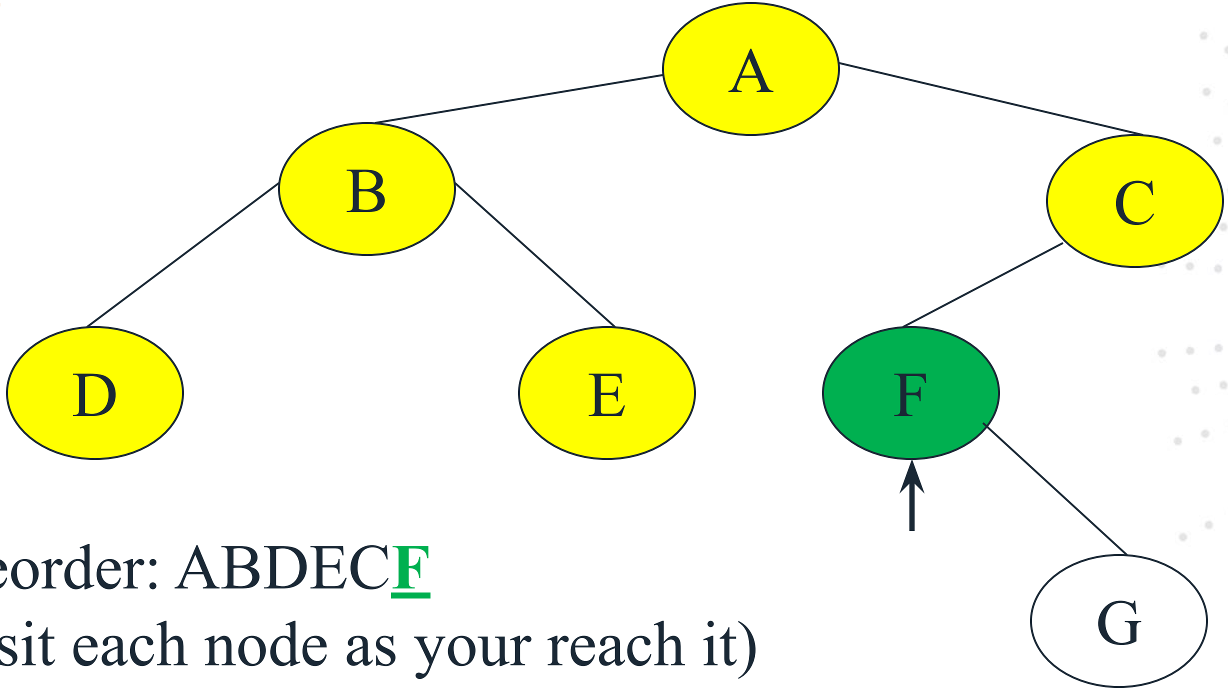
Preorder: ABD**E**
(visit each node as your reach it)

Binary Tree Traversal



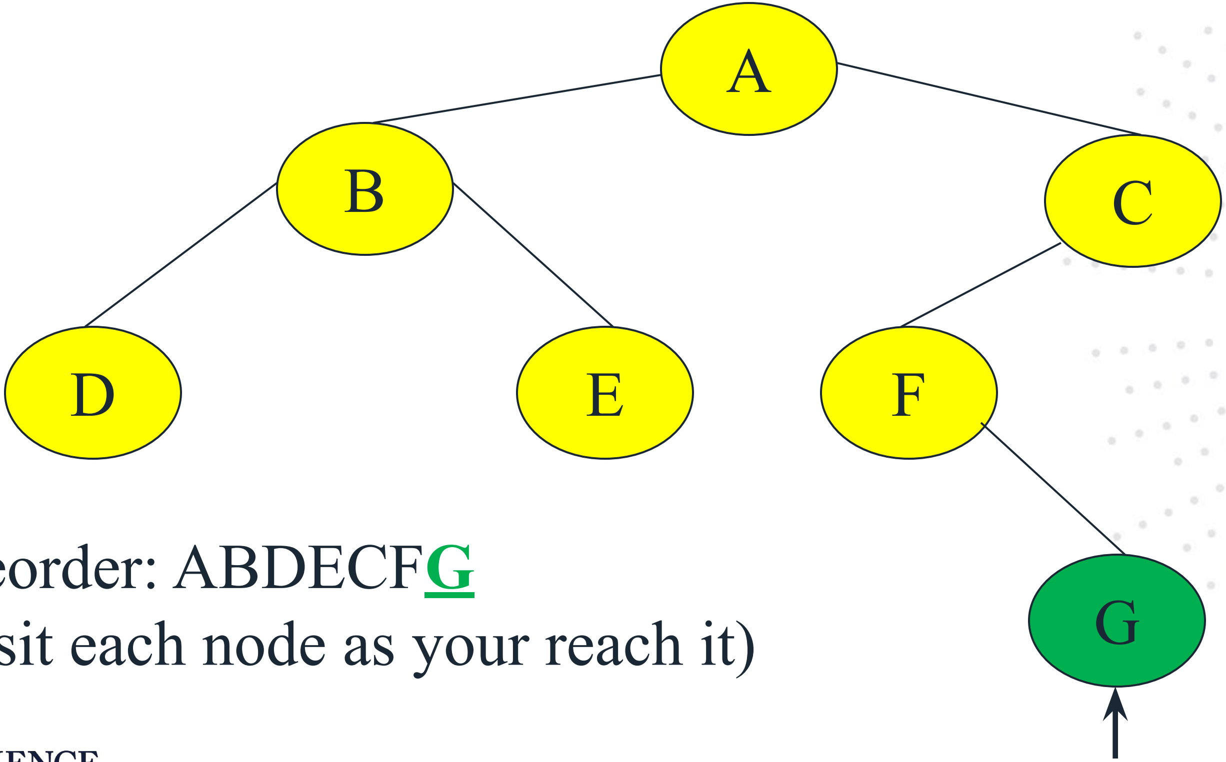
Preorder: ABDE<u>**C**</u>
(visit each node as your reach it)

Binary Tree Traversal



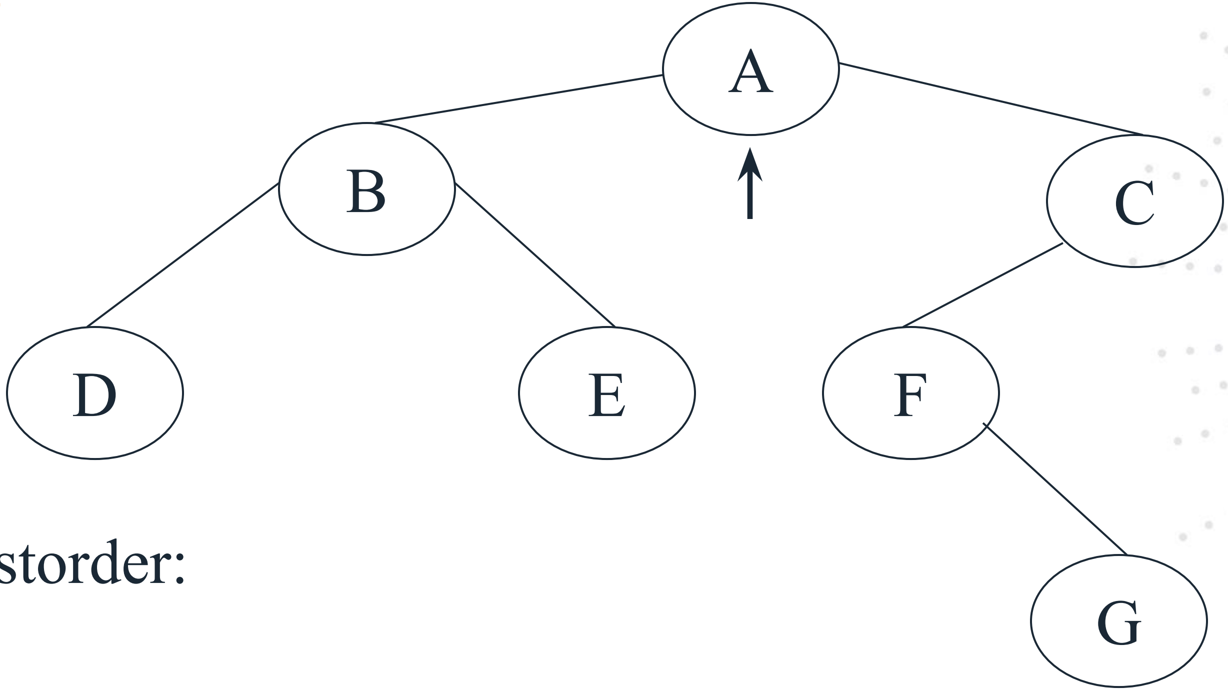
Preorder: ABDEC**F**
(visit each node as your reach it)

Binary Tree Traversal



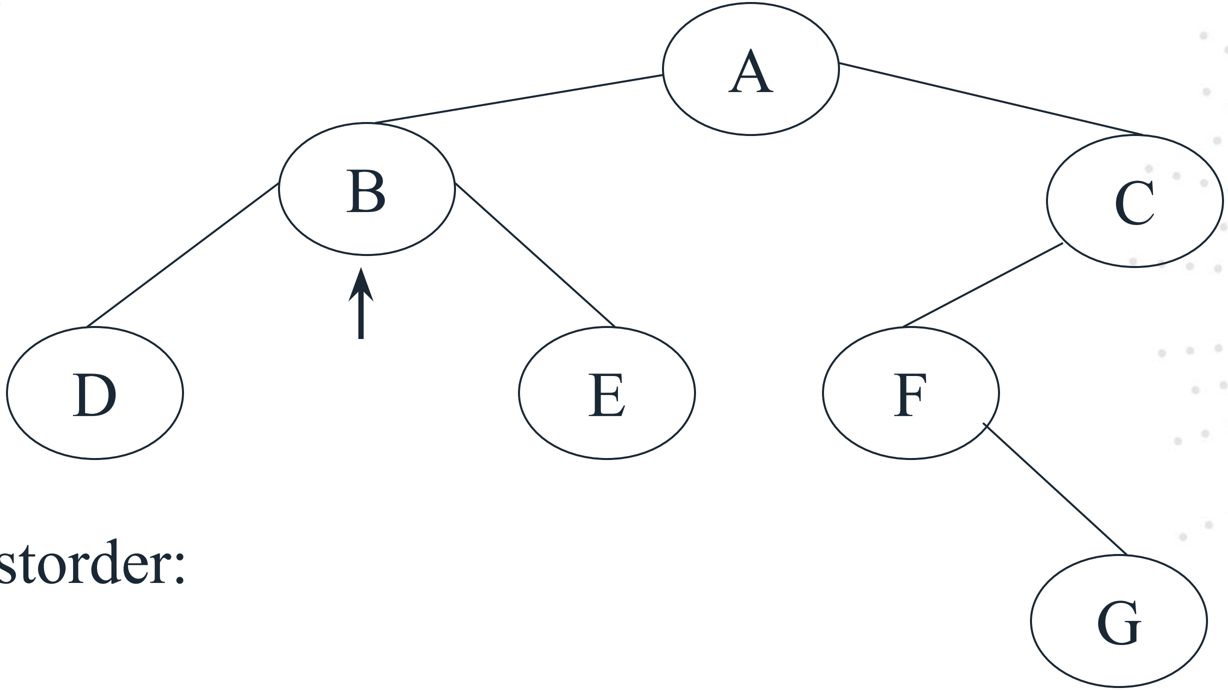
Preorder: ABDECF**G**
(visit each node as your reach it)

Binary Tree Traversal



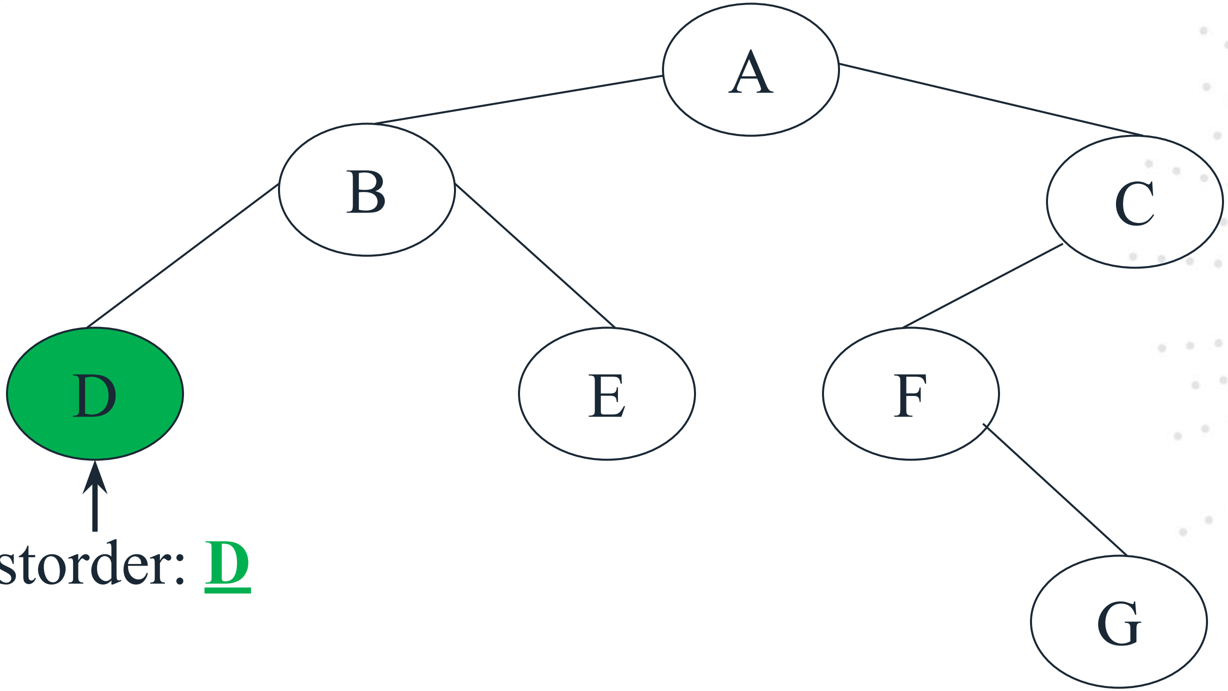
Postorder:

Binary Tree Traversal



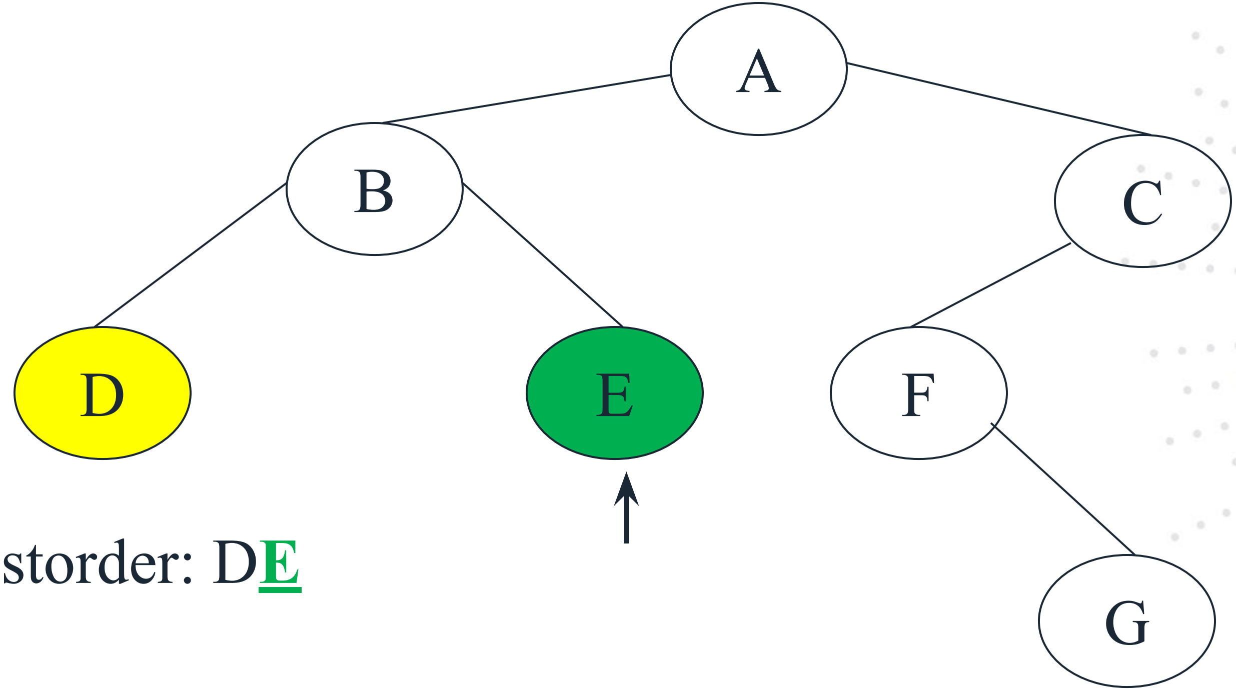
Postorder:

Binary Tree Traversal



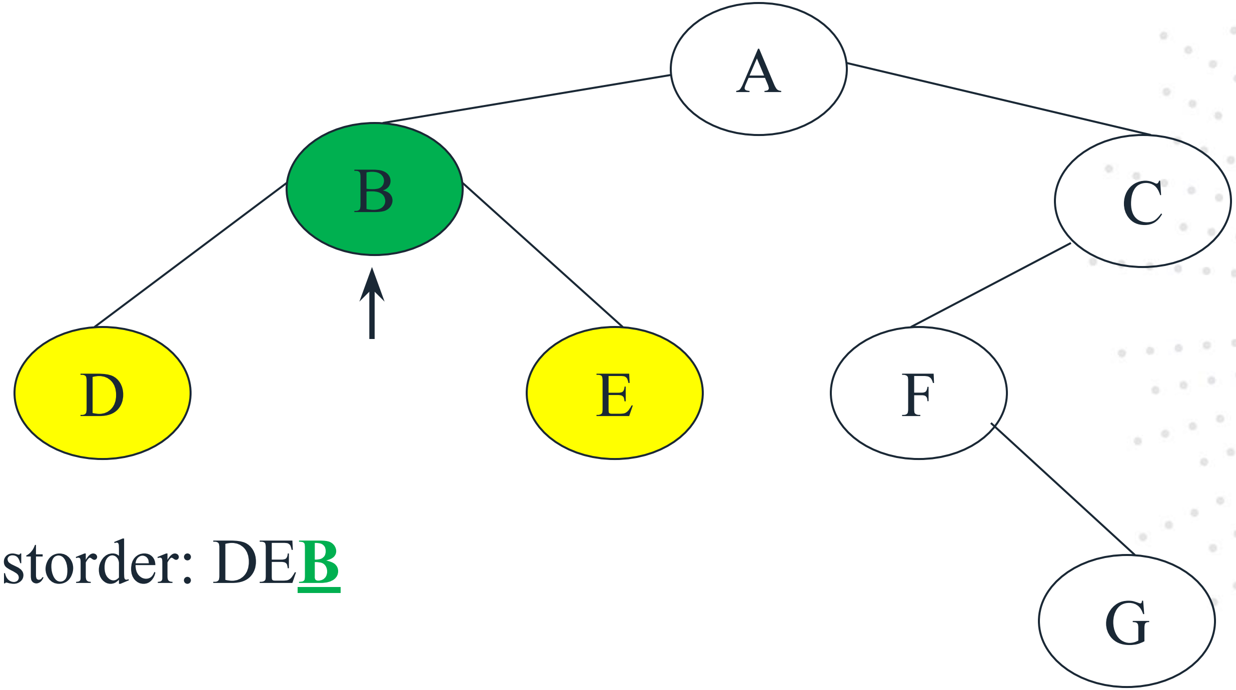
Postorder: **D**

Binary Tree Traversal



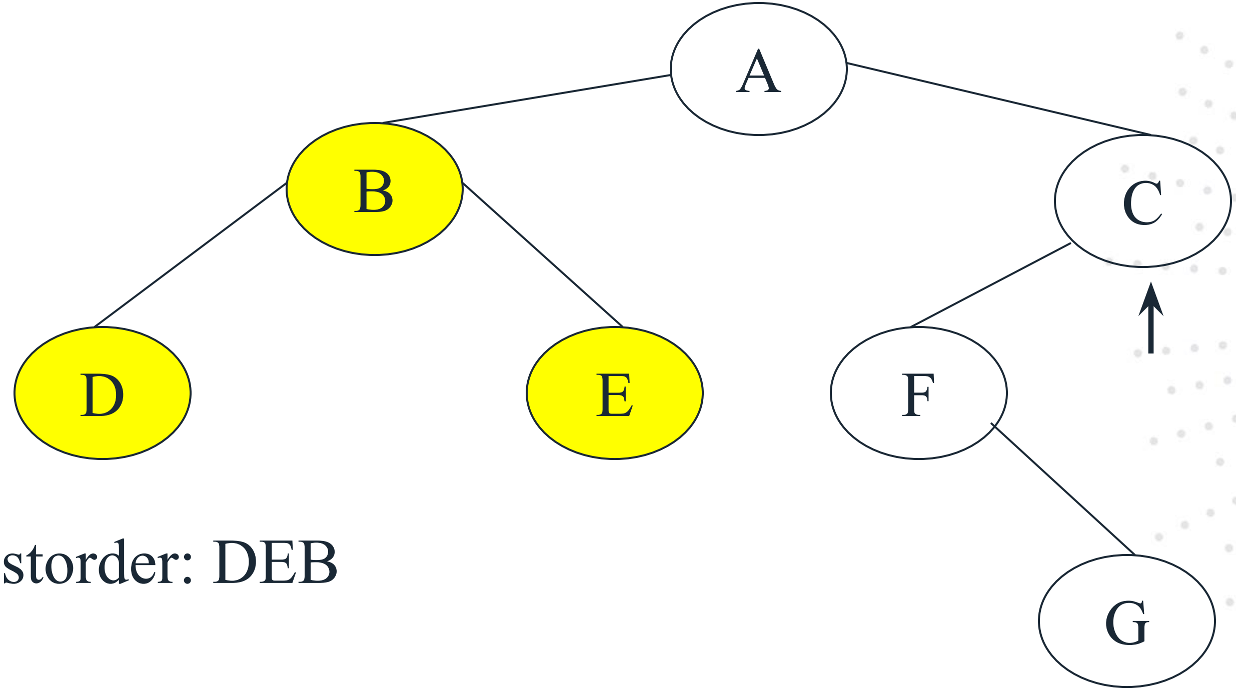
Postorder: D**E**

Binary Tree Traversal



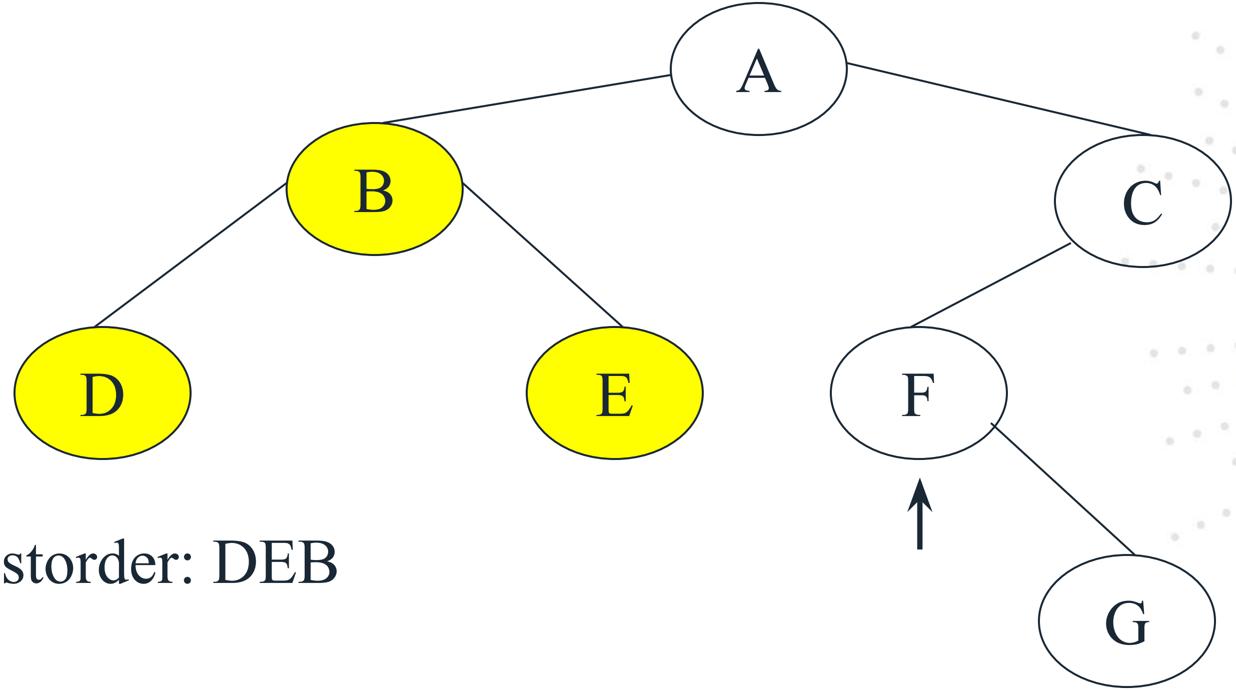
Postorder: DE**B**

Binary Tree Traversal



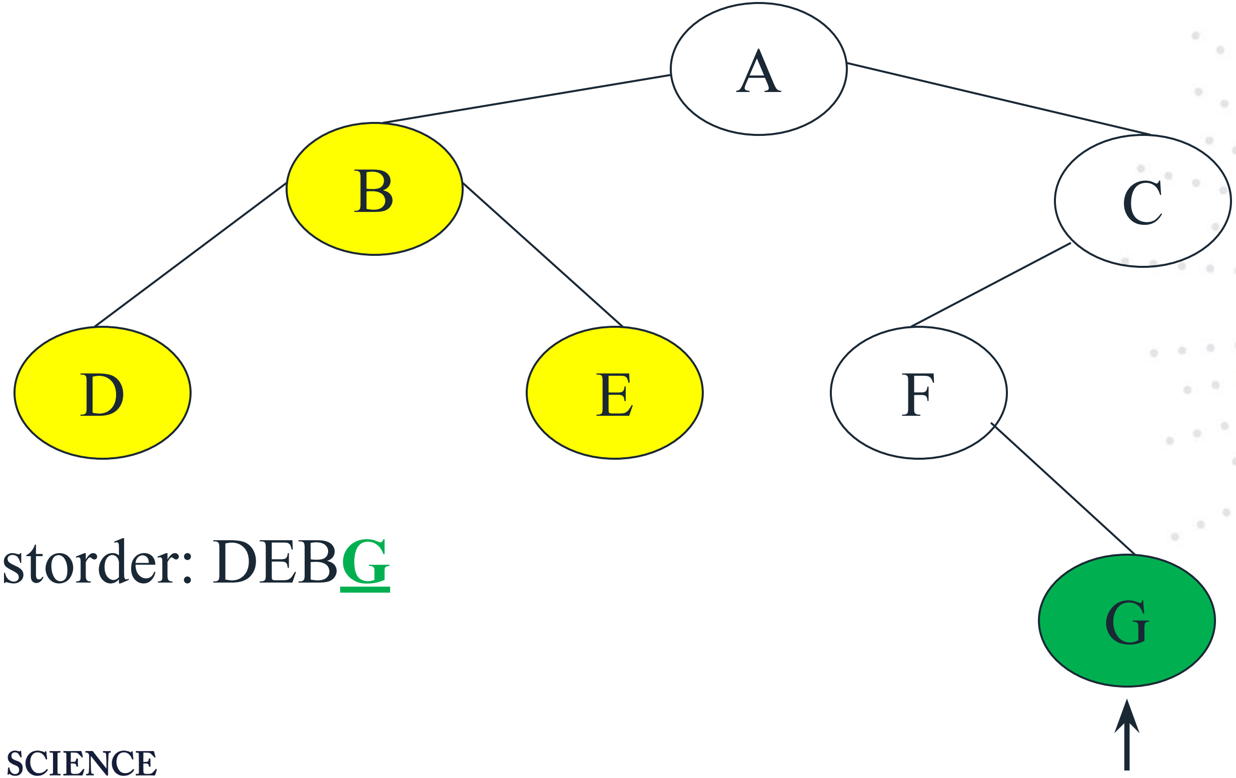
Postorder: DEB

Binary Tree Traversal



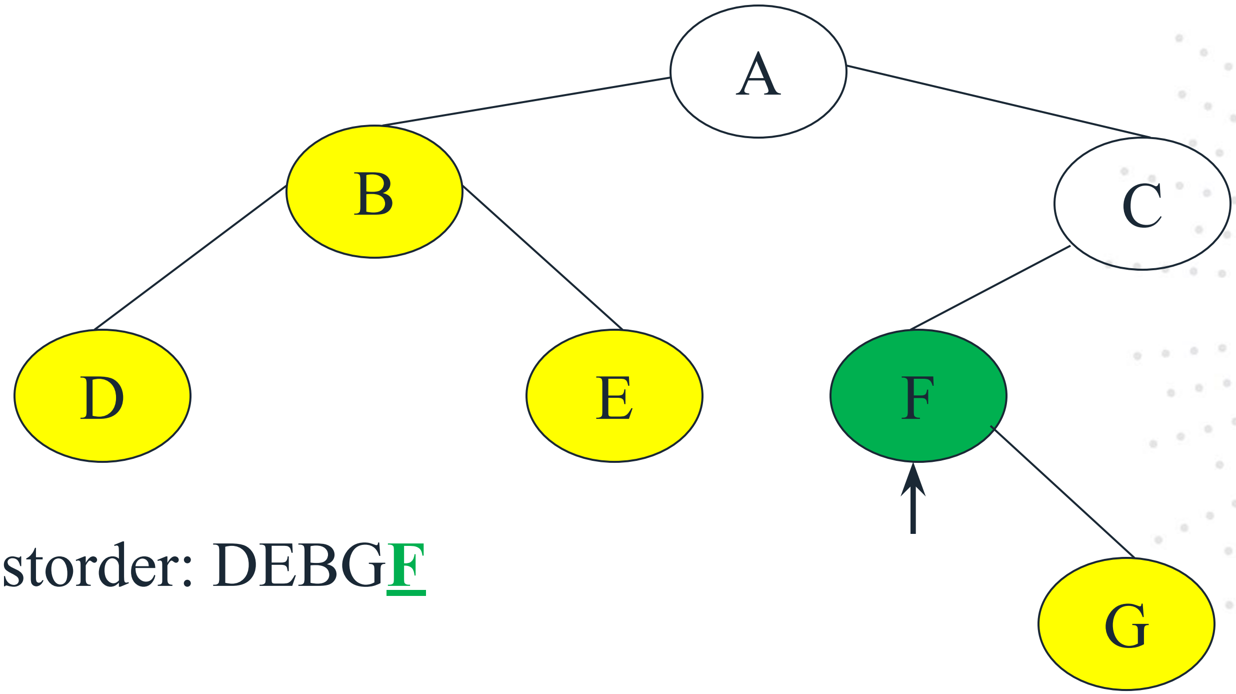
Postorder: DEB

Binary Tree Traversal



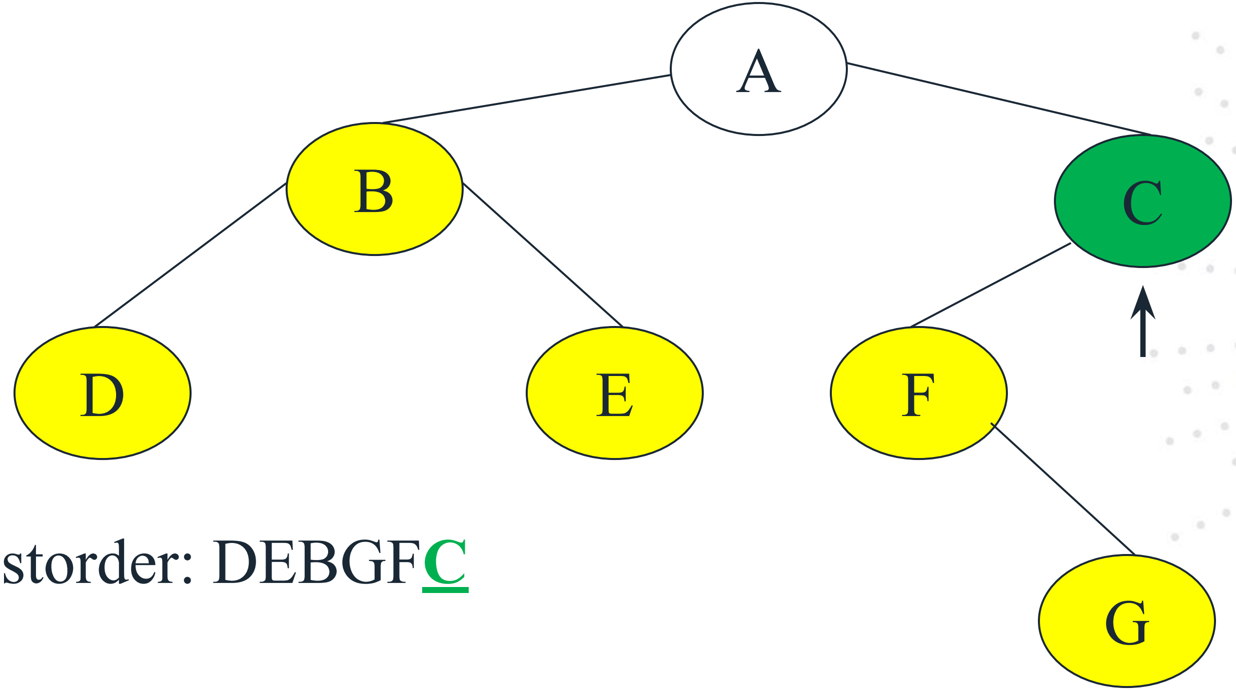
Postorder: DEB**G**

Binary Tree Traversal



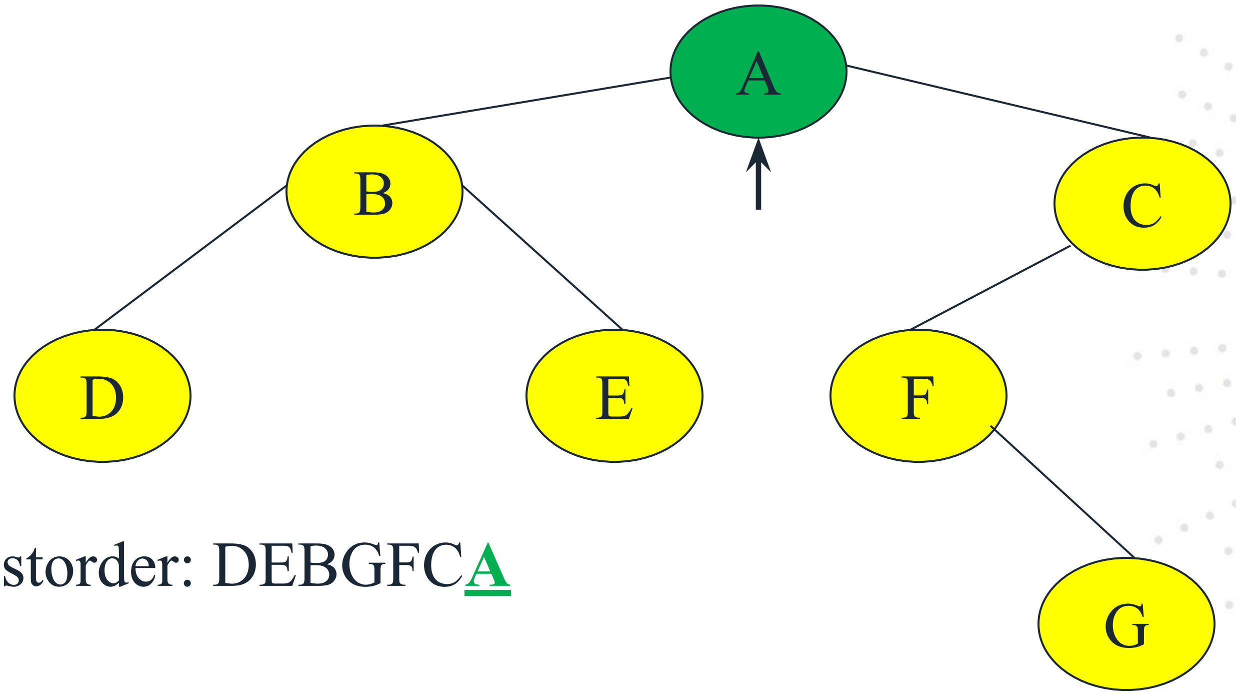
Postorder: DEBG<ins>F</ins>

Binary Tree Traversal



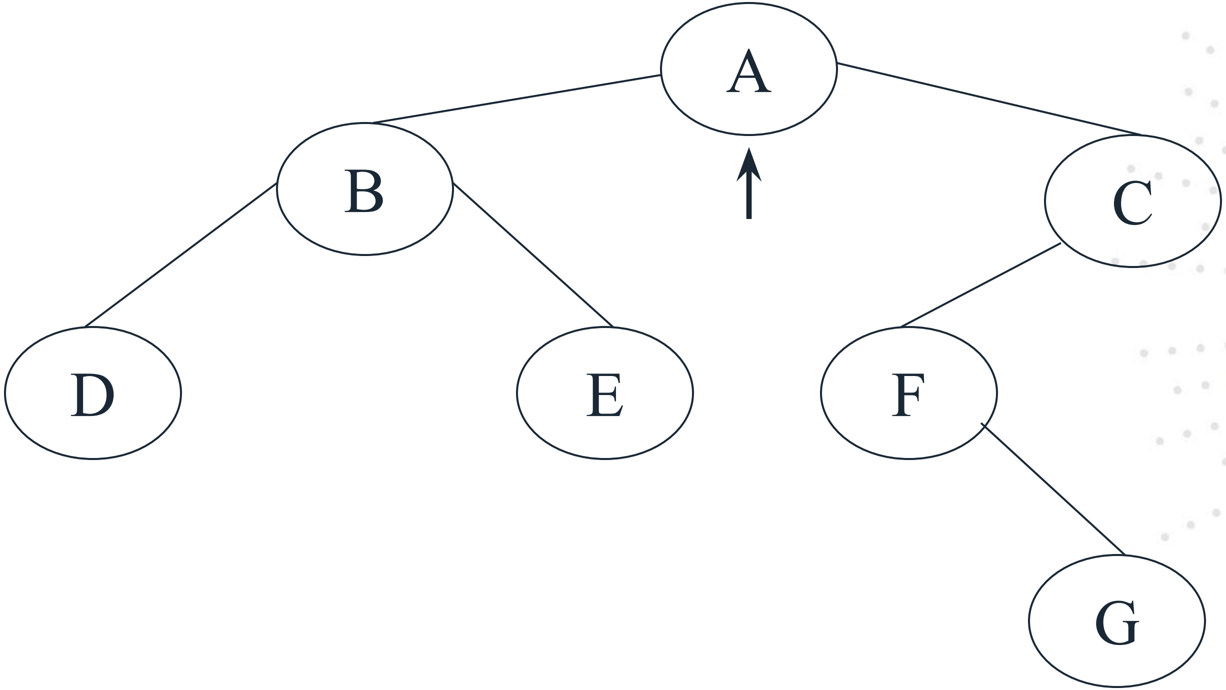
Postorder: DEBGF**C**

Binary Tree Traversal



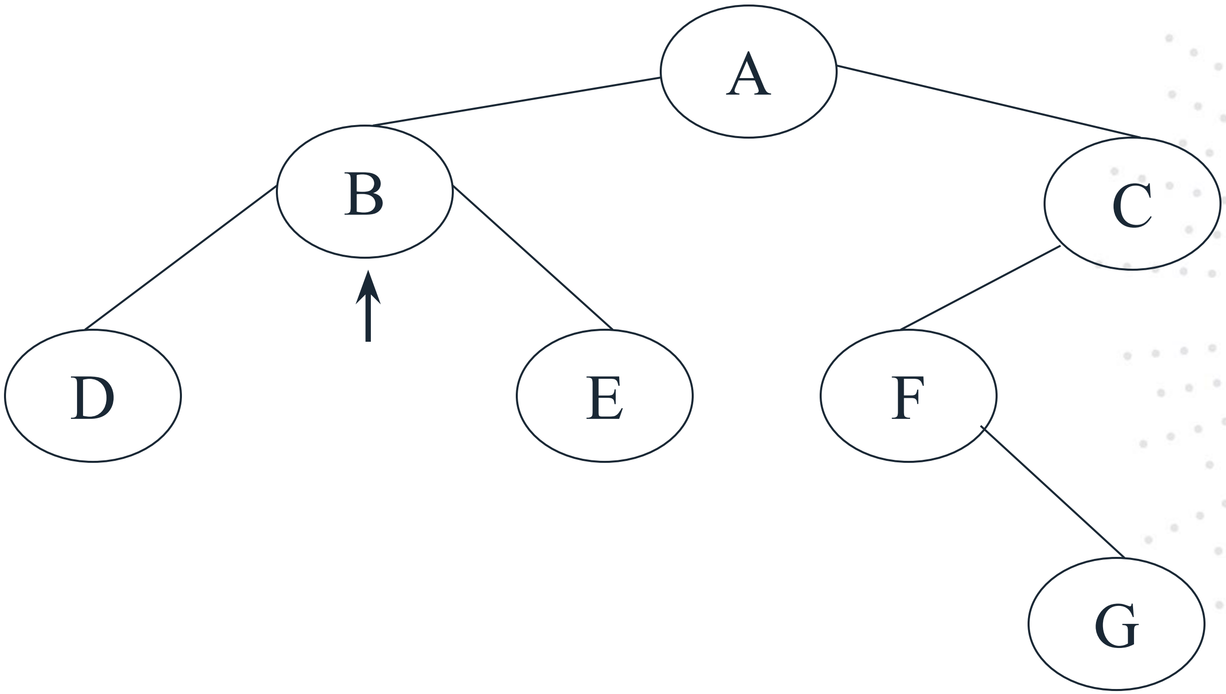
Postorder: DEBGFC**A**

Binary Tree Traversal



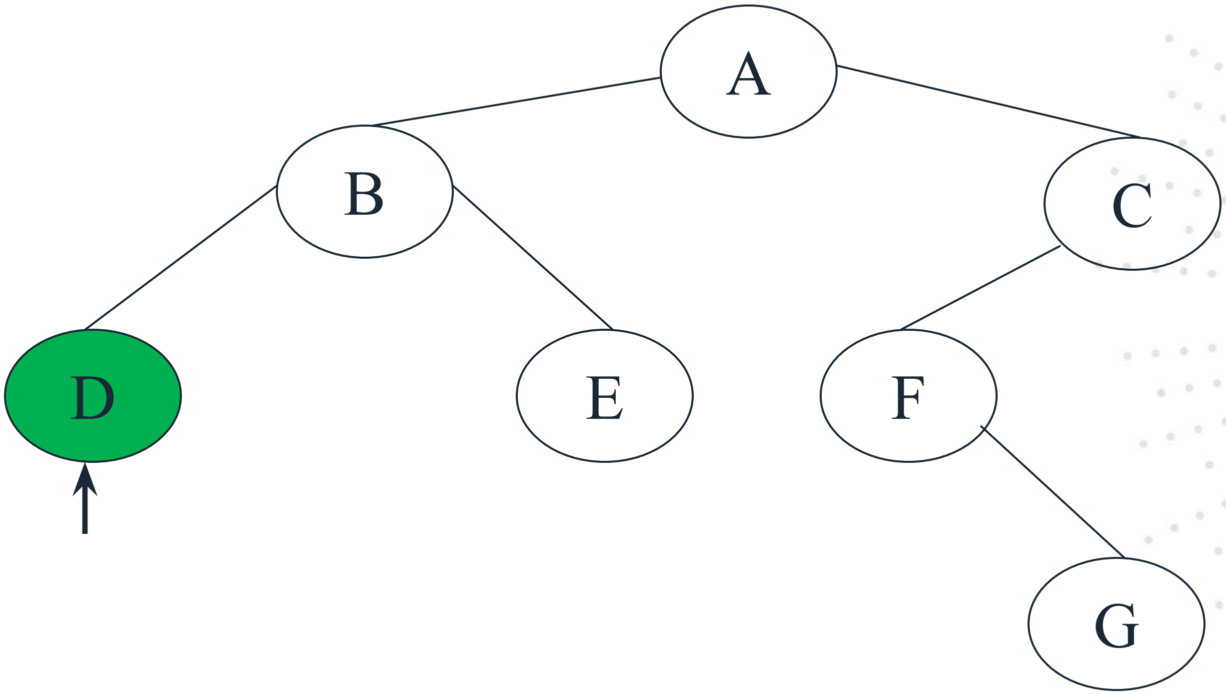
In Order:

Binary Tree Traversal



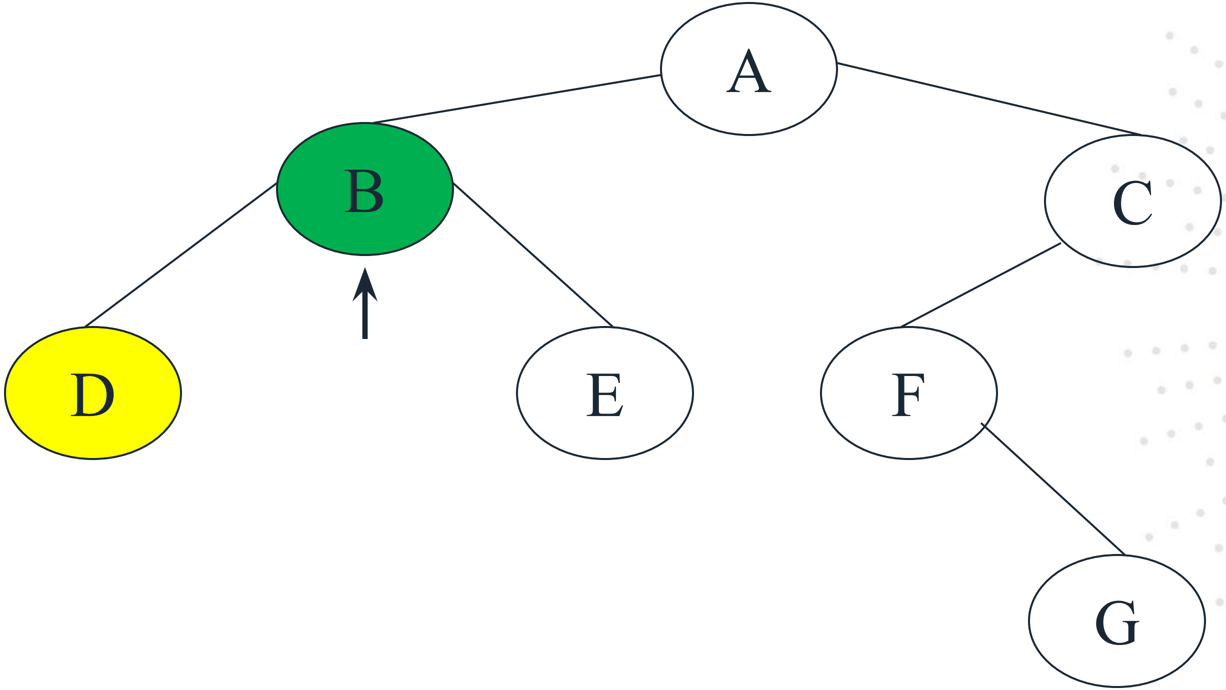
In Order:

Binary Tree Traversal



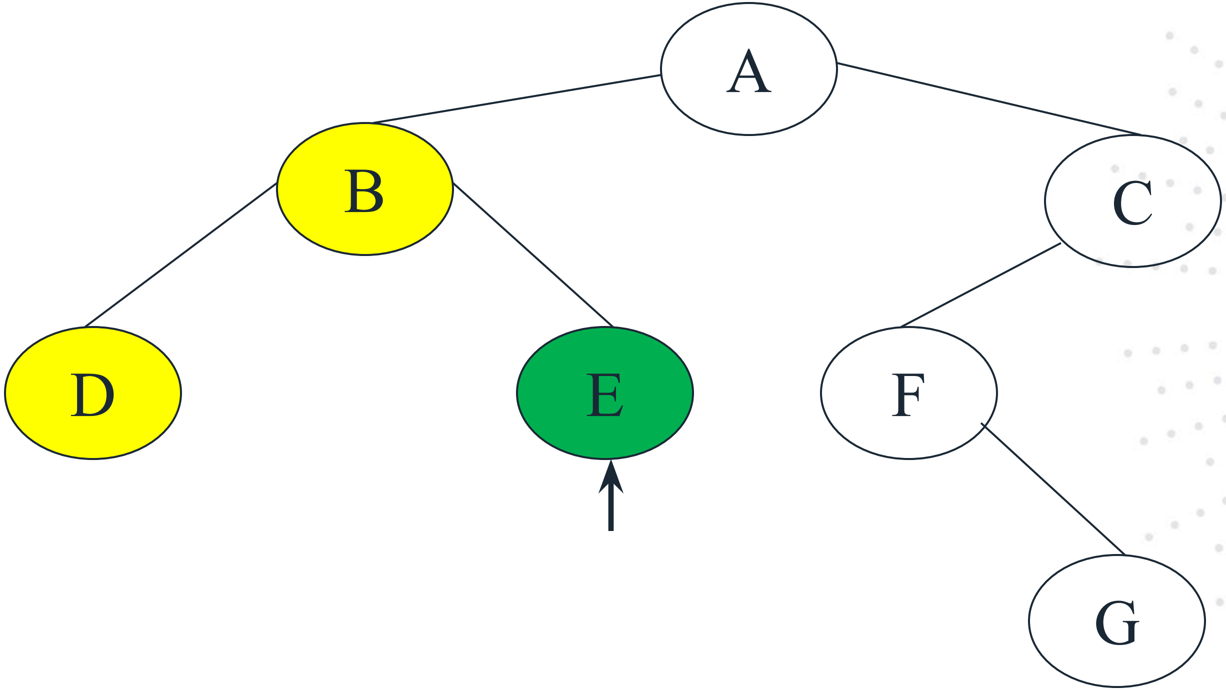
In Order: **D**

Binary Tree Traversal



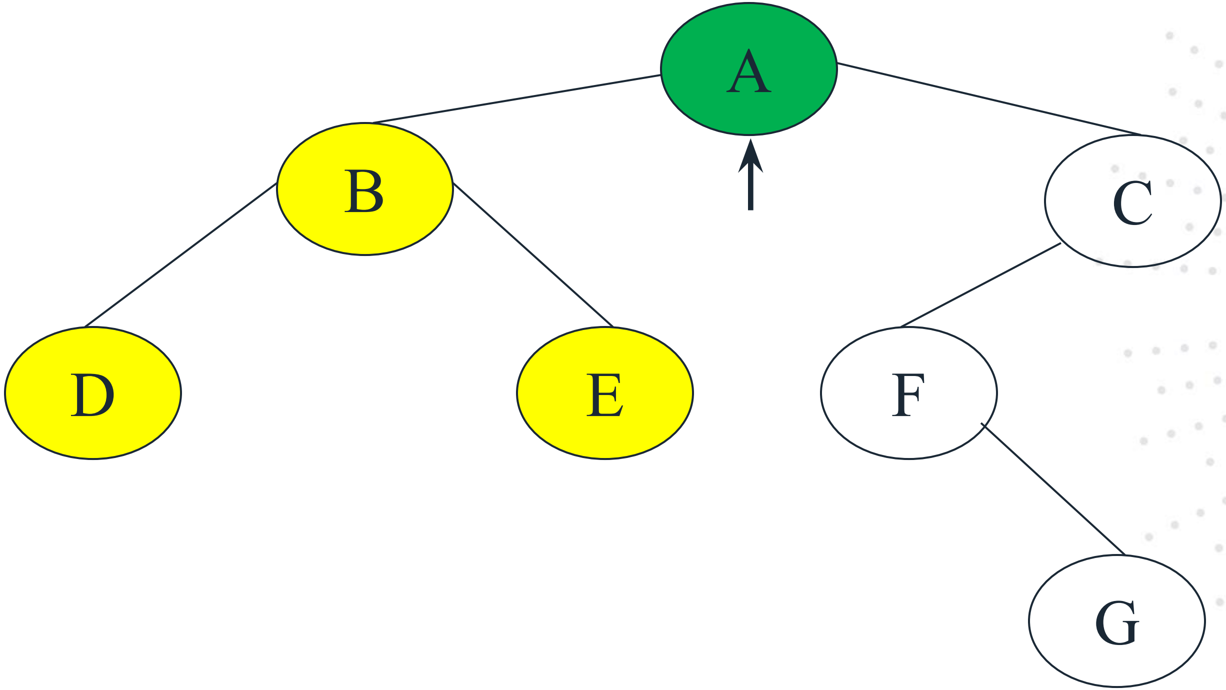
In Order: D**B**

Binary Tree Traversal



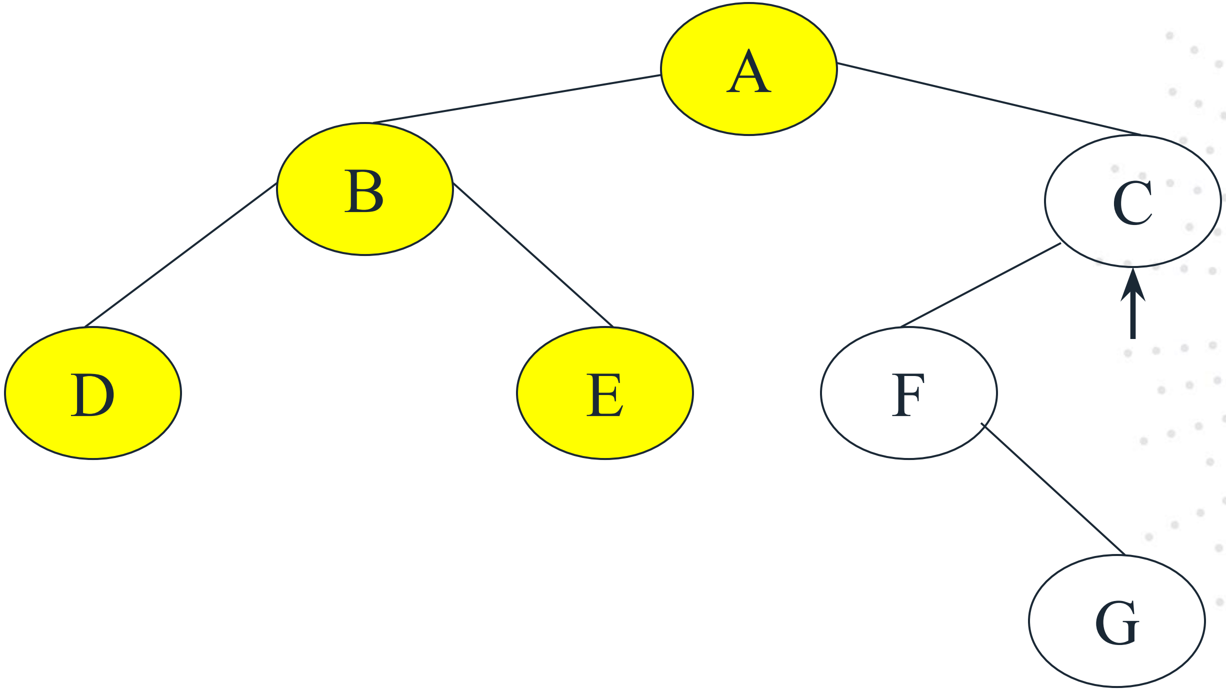
In Order: DBE

Binary Tree Traversal



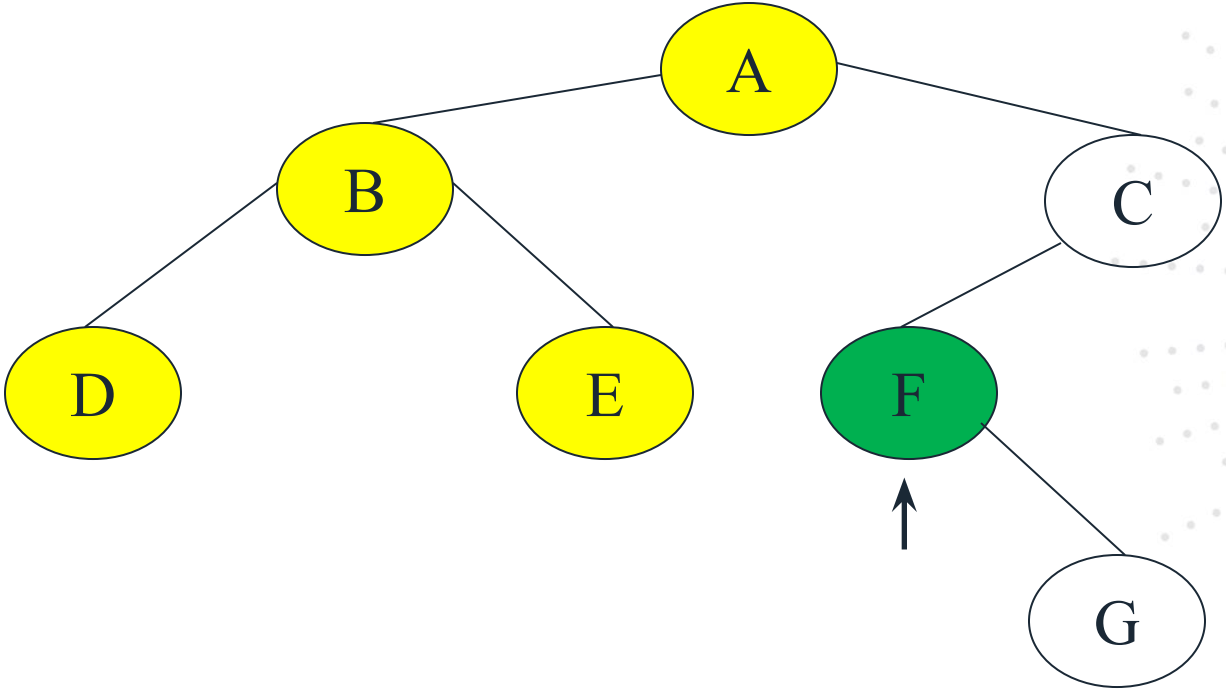
In Order: DBEA

Binary Tree Traversal



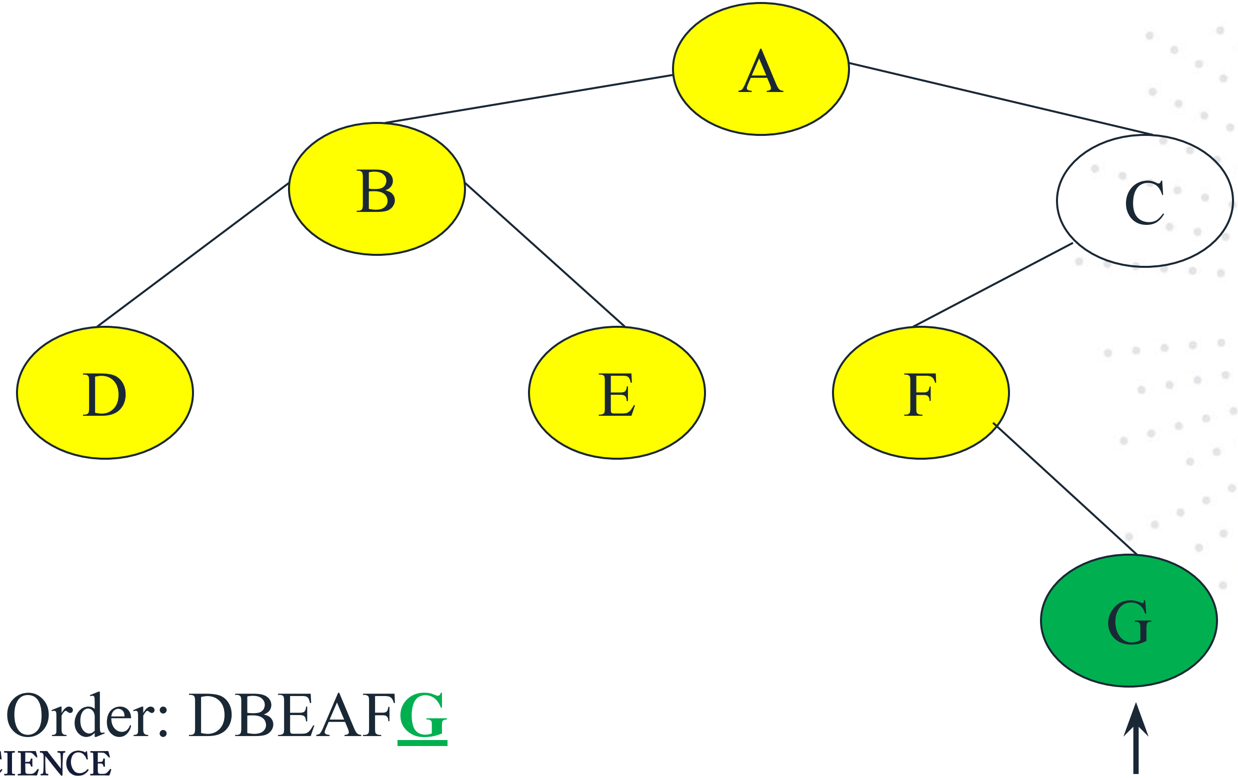
In Order: DBEA

Binary Tree Traversal



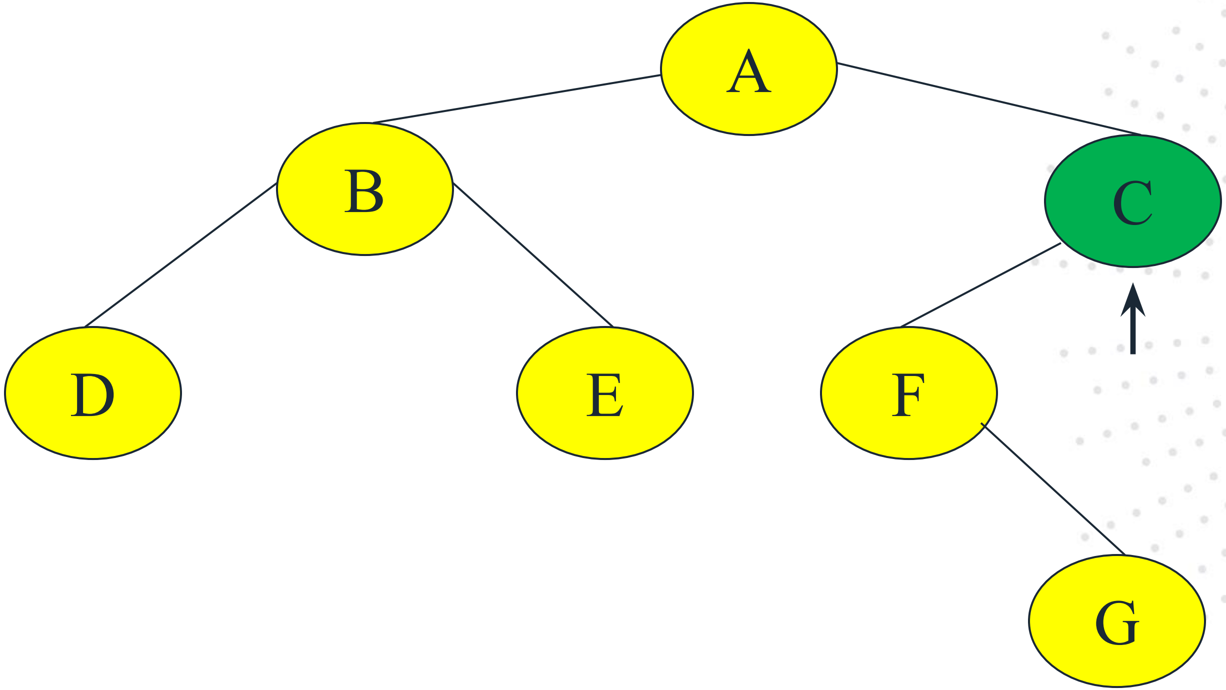
In Order: DBEAF

Binary Tree Traversal



In Order: DBEAFG

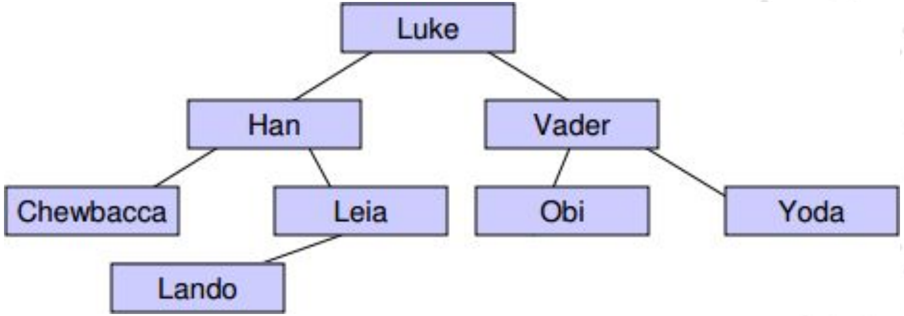
Binary Tree Traversal



In Order: DBEAFGC

Binary Tree Traversal Practice

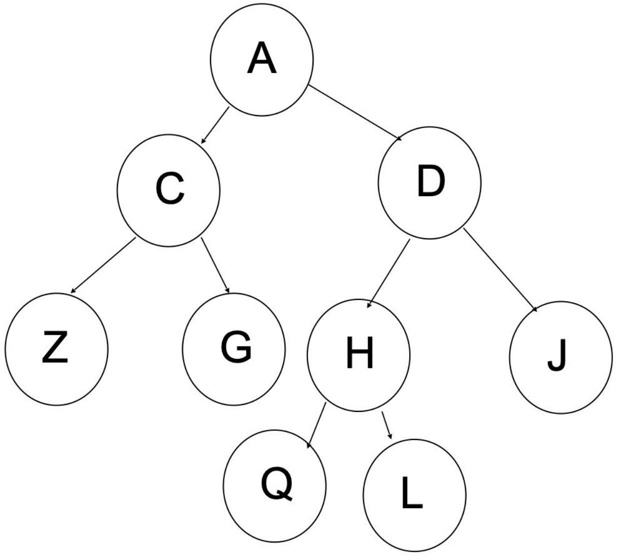
- Write the pre-, in-, and post-order traversals of the following tree:



- In-order:
- Pre-order:
- Post-order:

Binary Tree Traversal - Quiz

- What is a the result of a post order traversal of the tree to the left?
  - Z G C A Q H L D J
  - Z G C Q L H J D A
  - A C Z G D H Q L J
  - A C D Z G H J Q L
  - None of these



Traversal Applications

- When would we want to traverse a tree? What are some applications?
  - Processing tree elements
  - Make a clone (deep copy) of a tree
  - Determine tree height
  - Determine tree size (number of nodes)
  - Searching
  - …

Iterative Depth-First Search

- **Depth-first search** (**DFS**) goes deeply into the tree and then backtracks when it reaches the leaves.
- DFS pseudocode algorithm using a ***Stack***!

```
stack.push(root)

while (stack is not empty):
         n = stack.pop()

         process(n) // “visit”

         for (each child of n, starting with the last one):

                    stack.push(child)
```

This algorithm accomplishes a **pre-order** traversal

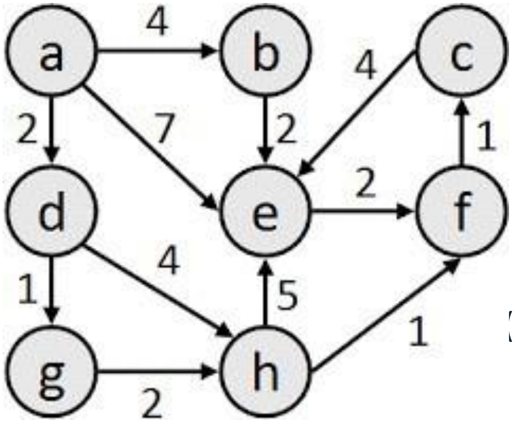
Iterative Breadth-First Search

- **Breadth-first search** (**BFS**) visits all notes on the same level before going to the next.
- Harder to do recursively than DFS (harder to simulate a queue)
- BFS pseudocode algorithm using a ***Queue***!

```
queue.add(root)
while (queue is not empty):
    n = queue.remove()
    process(n) // "visit"
    for (each child of n):
        queue.add(child)
```

When would you use Breadth-First?

- Breadth-First Search has an interesting property in that it can be used to find the **shortest path** between two nodes
- See **Dijkstra's algorithm**



(not a tree, but a graph!)

UVA DATA SCIENCE

Foundation of Computer Science for Data Science

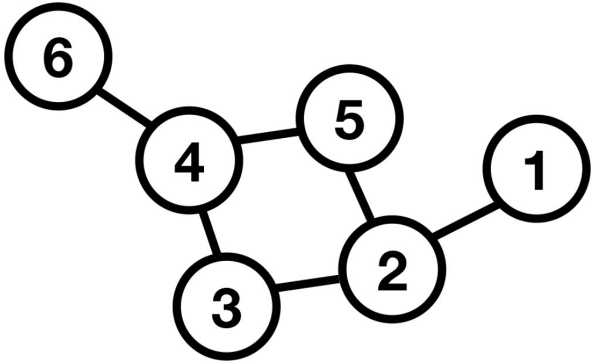
Graphs

Graph

- Have you ever heard of a company called *Meta*? How about *LinkedIn*?
  - Companies that connect people together use a *social graph* to do so. Facebook has the world's largest social graph, connecting well over a billion people together. Facebook keeps track of connections in a graph data structure, which we will discuss today.

Graph

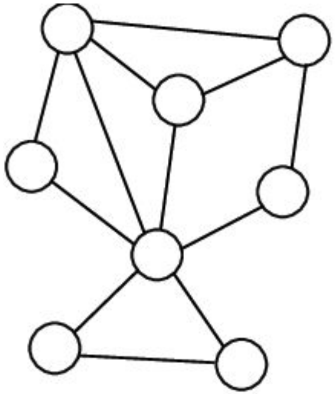
- A *graph* is a mathematical structure for representing relationships using ***vertices*** and edges.
    - A ***graph*** G is a set V of ***vertices*** and a collection E of pairs of vertices from V, called ***edges***.
- This is just like a tree, but without any rules!
- Some books use different terminology for graphs and refer to what we call vertices as nodes and what we call edges as arcs.



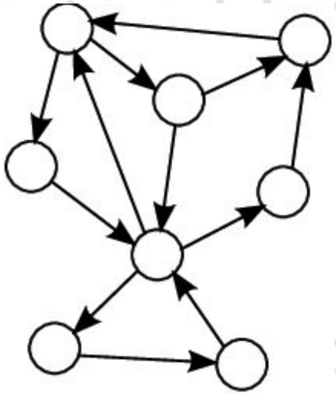
Graph Visualization: https://visualgo.net/en/graphds

Graph

- Graphs do not have any notion of a parent-child relationship, and they can have cycles
- Graphs also do not have roots
- Graphs can have *directed* edges or undirected edges, which means that you can get from one vertex to another but not necessarily in the other direction



Undirected Graph

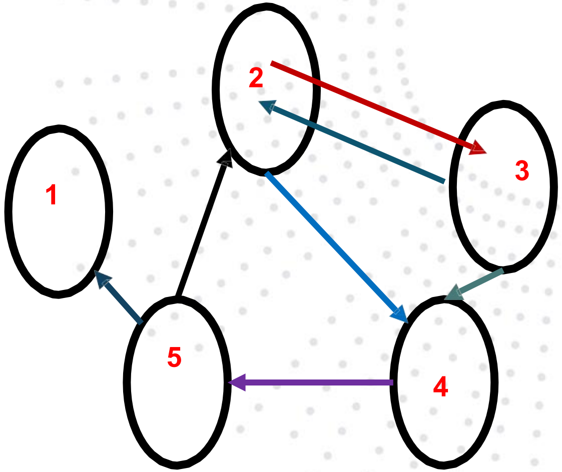


Directed Graph

Graph Visualization: https://visualgo.net/en/graphds

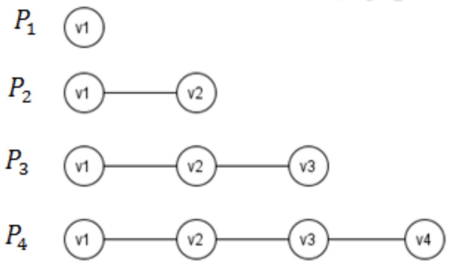
Graph

- Let G be any directed graph.
- Let the set E represent the edges of the graph where Eij is a directed edge that goes from vertex i to vertex j
- Eij can be written as (i,j)
- Let V be the set of all vertices (nodes) in the graph
  - EXAMPLE
    - V = {1, 2, 3, 4, 5}
    - E= {(2,3), (2,4), (3,2), (3,4), (4,5), (5,1), (5,2)}



Graph

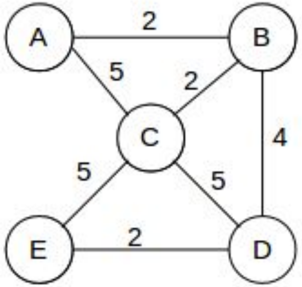
- **Directed edge**: An edge that only goes in one direction (e.g., from **a** to **b**) but not the other way. In a diagram directed edges are arrows.
- **Undirected edge**: An edge that can go in both directions (e.g., from **a** to **b** and from **b** to **a**). In a diagram undirected edges are lines.
- **Path**: A *path* from vertex **a** to vertex **b** is a sequence of edges that can be followed starting from **a** to reach **b**.
  - A path can be represented as vertex visited, or edges taken.
  - path length is he number of nodes or edges contained in a path.
- **Neighbor** or **adjacent**: Two vertices connected directly by an edge



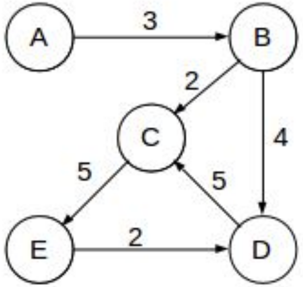
Graph Paths

Graph

- A weighted graph is a graph that has numbers associated with edges.
- Weighted graphs are particularly useful for applications such as mapping between two locations, to take into consideration traffic, distance, time, etc.
    - weighted graph of airline flights, with the weights representing the miles between airports



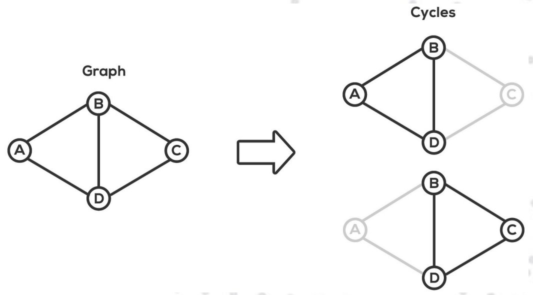
Undirected

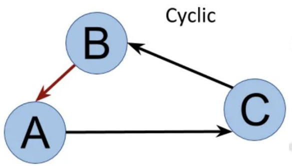


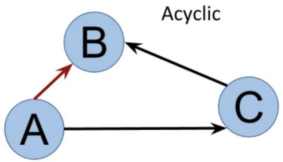
Directed

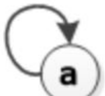
Graph

- **Cycle**: A path that begins and ends at the same node.
- **acyclic graph**: a graph that does not contain any cycles
- **loop**: an edge directly from a node to itself
    - Many graphs do not allow loops (you can't be friends with yourself on Facebook)





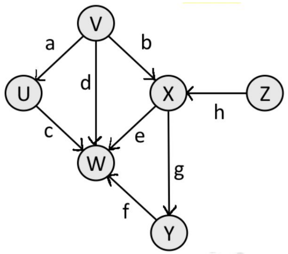




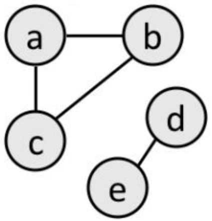
Loop

Graph

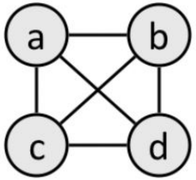
- **Reachable**: A vertex is *reachable* from another vertex if a path exists between the two vertices.
- **Connected**: A graph is *connected* if every vertex is reachable from every other vertex
- **Complete**: A graph is *complete* if every vertex has an edge to every other edge



Vertex X is reachable from Vertex V



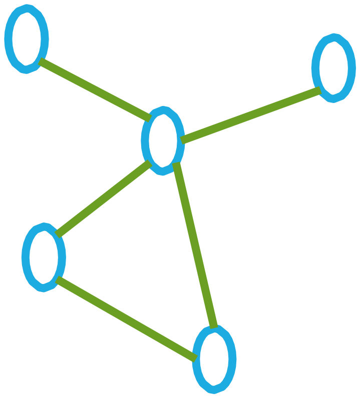
Unconnected graph



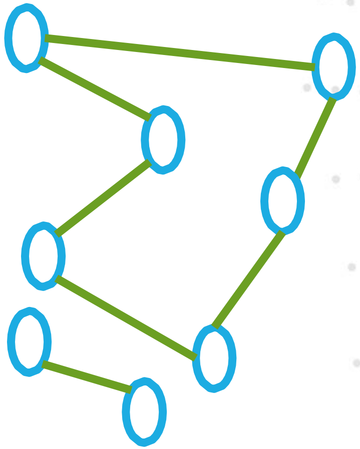
Complete graph

Graph

- How many connected components?



This Graph has 1
connected component

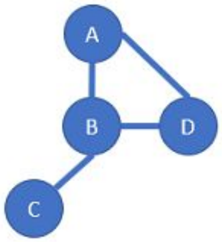


This Graph has 2
connected components.

Adjacency Matrix

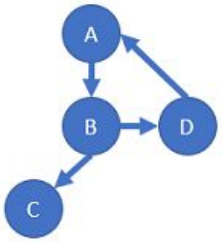
- An **adjacency matrix** is a way of representing a graph using a square matrix, where the rows and columns correspond to the graph vertices, and the entries indicate whether there is an edge between pairs of vertices
- Given a graph G=(V,E) with V vertices and E edges, the **adjacency matrix** A is a V×V matrix where each element A[i][j] represents the connection between vertex i and vertex j
- In an **undirected graph**, the adjacency matrix is **symmetric**.
- In a **directed graph**, the adjacency matrix is **not necessarily symmetric**.

Graph Visualization: https://visualgo.net/en/graphds



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 1 & 0 & 0 \\ D & 1 & 1 & 0 & 0 \end{array}$$

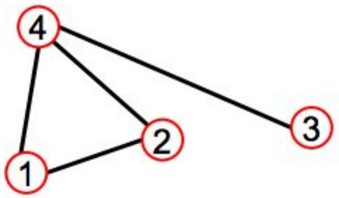
Undirected

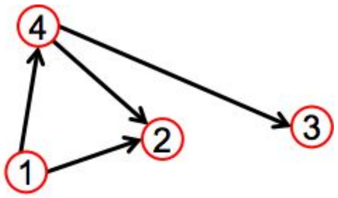


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 0 \\ B & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 0 & 0 & 0 \end{array}$$

Directed

Adjacency Matrix





$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

SYMMETRIC **NOT SYMMETRIC**

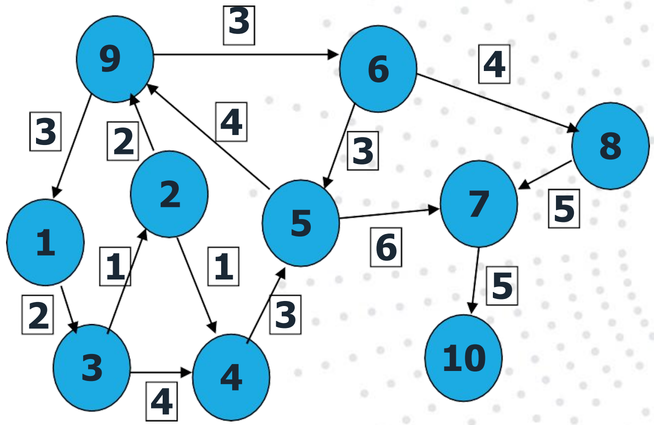
Adjacency Matrix

Here, vertex 1 has a link to vertex 3 with a value of 2. Vertex 3 is connected to vertex 2 with a value of 1.

So, in the matrix, row 1 column 3 has a value of 2 and row 3 column 2 has a value of 1.

0	0	2	0	0	0	0	0	0	0
0	0	1	0	0	0	0	2	0	0
0	1	0	4	0	0	0	0	0	0
0	0	0	0	3	0	0	0	0	0
0	0	0	0	0	6	0	4	0	
0	0	0	0	3	0	0	4	0	0
0	0	0	0	0	0	0	0	0	5
0	0	0	0	0	0	5	0	0	0
3	0	0	0	0	3	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Adjacency Matrix



Graph Visualization: https://visualgo.net/en/graphds

Adjacency List

- An **adjacency list** is a data structure used to represent a graph. It consists of a list (or array) where each element corresponds to a vertex in the graph, and each element contains a list of the vertices that are directly connected to it by an edge.
- The adjacency list is efficient for storing sparse graphs (where the number of edges is much smaller than the square of the number of vertices), and it uses less memory than an adjacency matrix.
- In an **undirected graph**, if two vertices A and B are connected by an edge, both A and B will appear in each other's adjacency list. In a **directed graph**, if there is a directed edge from A to B, only B will appear in A's list.

Adjacency List

Keep track of all the edges in the graph

Edge list

2 3

2 4

3 2

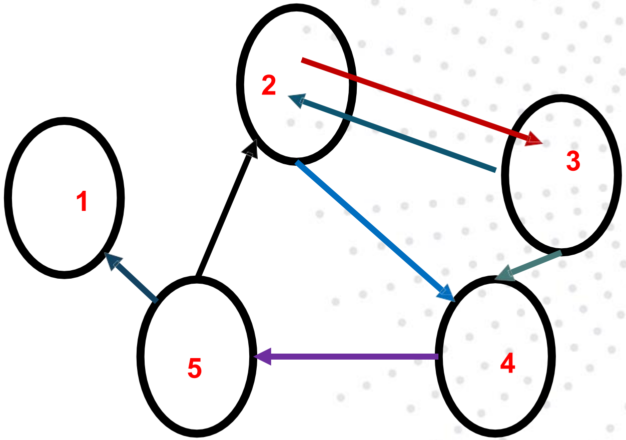
3 4

4 5

5 2

5 1

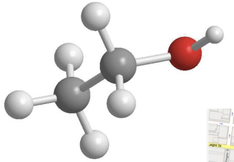
- Node list with edges
  - 1: no exiting edges
  - 2: 3 4 (we have edges (2,3) and (2,4)
  - 3: 2 4 (we have edges (3, 2) and (4,4)
  - 4: 5 (we have edge (4,5)
  - 5: 1 2 (we have edges (5, 1) and (5, 2)

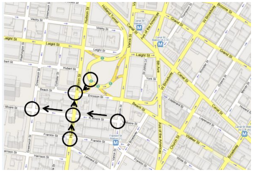


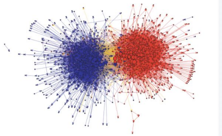
Graph Visualization: https://visualgo.net/en/graphds

Graph Applications

- Graphs are used in many domains such as Geography, Chemistry and Engineering sciences.
    - Chemical bonds
    - Road maps (Google Map)
    - Partisanship
    - Circuit routing
    - Amazon product relationships







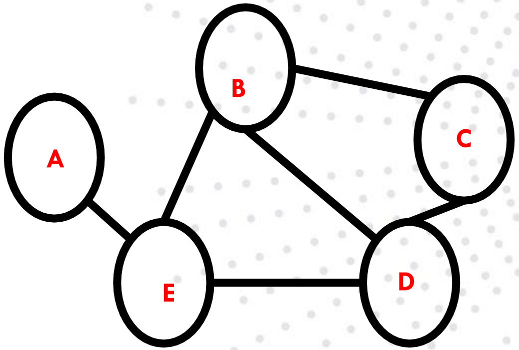
The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

Graph Density

- **Graph Density** is a measure of how many edges are in a graph compared to the maximum number of edges possible.
- The density D of a graph is defined as the ratio of the number of edges present in the graph to the maximum possible number of edges in that graph.
- For a graph with V vertices and E edges:
  - In an **undirected graph**: $D = \frac{2E}{V(V-1)}$
  - In a **directed graph**: $D = \frac{E}{V(V-1)}$
  - Where:
    - V is the number of vertices
    - E is the number of edges
    - V(V−1) is the maximum possible number of edges in a directed graph, and V(V−1)/2 is the maximum in an undirected graph (since edges are bidirectional).

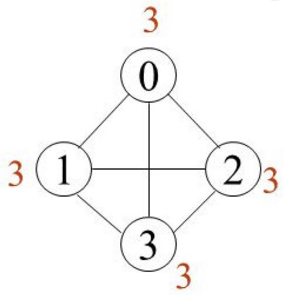
Graph Density

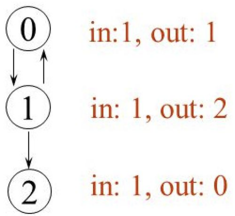
- Example
  - Graph Density $D = \frac{2E}{V(V-1)}$
  - we have E = 6 and V = 5
  - D =(2*6)/5*(5-1) = 12/20 = 0.6
- So, the density of this graph is **0.6**, meaning that 60% of the possible edges are present



Node Degree

- **Node Degree in Graphs** refers to the number of edges connected to a particular vertex
- **Degree of a node**: In an undirected graph, the degree of a vertex is the number of edges connected to it.
- **In directed graphs**, each edge has a direction (from one vertex to another). Therefore, the degree of a vertex is split into:
    - **In-Degree**: The number of edges directed **toward** the vertex
    - **Out-Degree**: The number of edges directed **outward** from the vertex





Betweenness Centrality

- **Betweenness Centrality** is a measure of centrality in a graph that quantifies the importance of a vertex based on the number of shortest paths that pass through it.
- The **betweenness centrality** of a vertex v is the sum of the fraction of all-pairs shortest paths that pass through v
- For a graph G, the betweenness centrality $C_B(v)$ of a vertex v is given by: $C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma(s,t|v)}{\sigma(s,t)}$
  - s and t are different vertices in the graph.
  - σ(s,t) is the total number of shortest paths from vertex s to vertex t
  - σ(s,t|v) is the number of those shortest paths that pass through vertex v

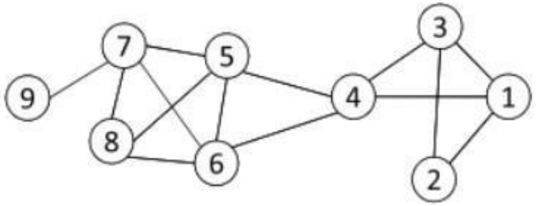
Betweenness Centrality

- Vertices with high betweenness centrality often serve as "bridges" or "gateways" in the network, playing a critical role in the flow of information or resources.
- They are also the ones whose removal from the graph will most **disrupt communications** between other vertices because they lie on the largest number of paths taken by

Betweenness Centrality Example

- Betweenness Centrality Example

RE: Lei Tang and Huan Liu, Morgan & Claypool, September, 2010.



$C_B(4) = 15$

Table 2.2: $\sigma_{st}(4)/\sigma_{st}$

	$s=1$	$s=2$	$s=3$
$t=5$	1/1	2/2	1/1
$t=6$	1/1	2/2	1/1
$t=7$	2/2	4/4	2/2
$t=8$	2/2	4/4	2/2
$t=9$	2/2	4/4	2/2

What's the betweenness centrality for node 5?

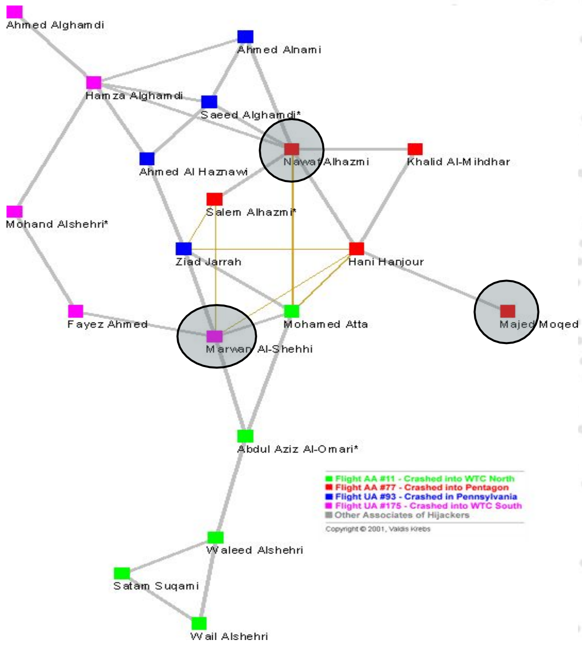
σ_{st} : The number of shortest paths between s and t

$\sigma_{st}(v_i)$: The number of shortest paths between s and t that pass v_i

$$C_B(v_i) = \sum_{v_s \neq v_i \neq v_t \in V, s<t} \frac{\sigma_{st}(v_i)}{\sigma_{st}}$$

Betweenness Centrality

- Where are the high and low betweenness nodes?
- Do all high betweenness nodes have to be high degree?
- Why do we care?



Communities in Graphs

- **Communities in Graphs** refer to groups of vertices that are more densely connected to each other than to the rest of the network
- A **community** (or cluster) in a graph is a subset of vertices that have more internal connections (within the group) than external connections (to vertices outside the group). The goal of community detection is to identify these densely connected regions

Graph WITH CLEAR COMMUNITIES

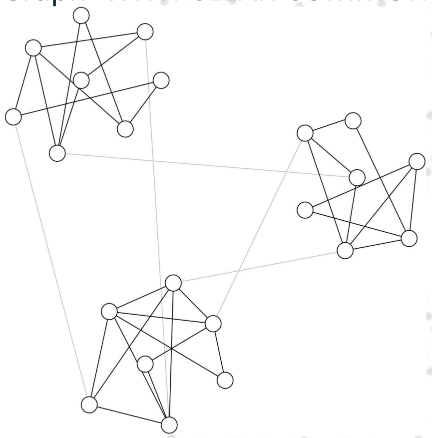
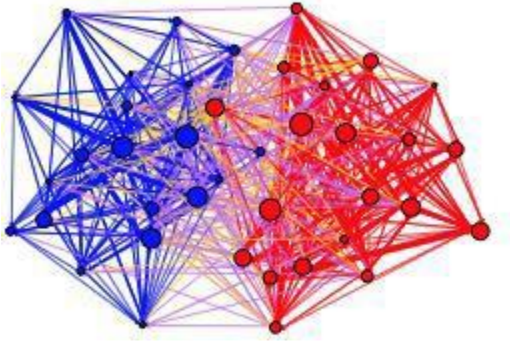


Image Source: M. Girvan, and M. E. J. Newman PNAS 2002;99:7821-7826

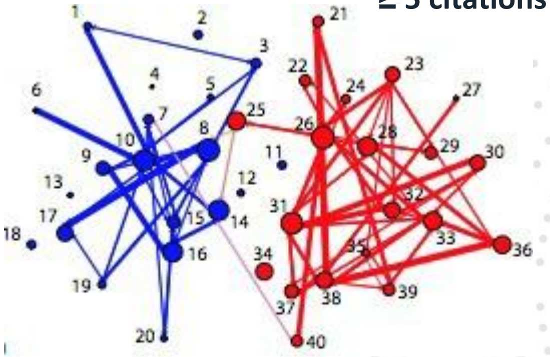
Communities in Graphs

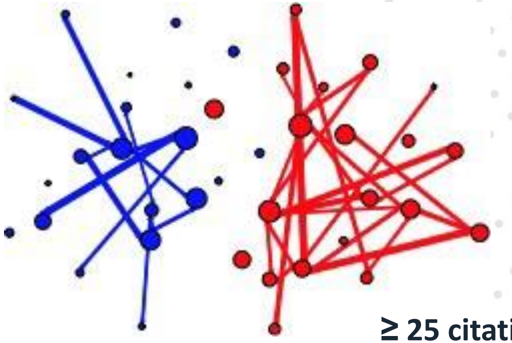
- **Strength of a Community** in a graph refers to the cohesiveness or tightness of connections within a community relative to the connections outside it



Citations of posts of top 20 liberal & conservative blogs

≥ 5 citations



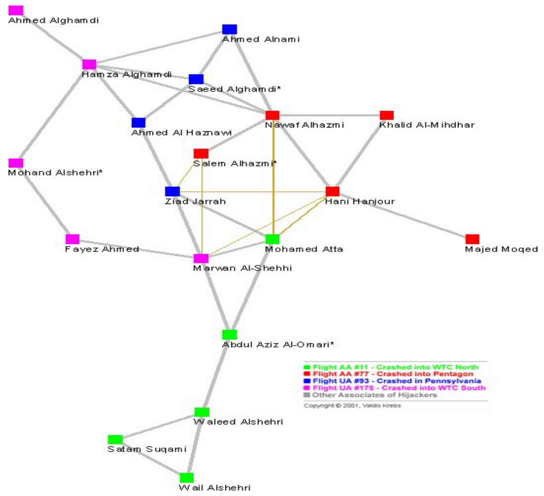


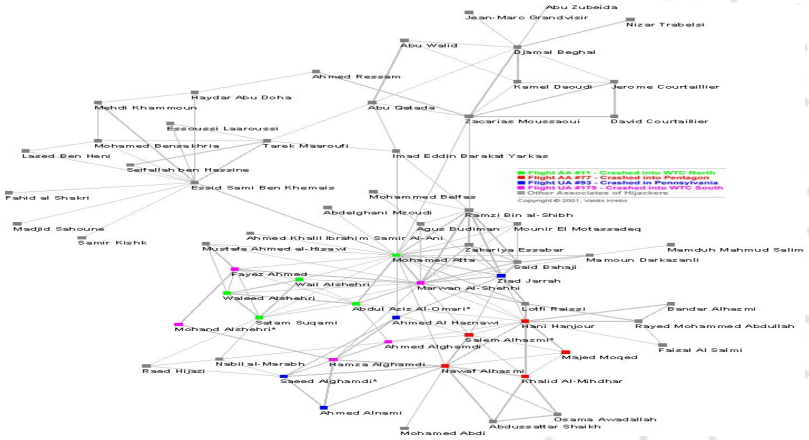
≥ 25 citations

Network

- A **network** is a graph that models real-world systems, where the nodes (vertices) represent entities, and the edges (links) represent the relationships or interactions between those entities.
- While graphs are abstract mathematical objects, **networks** are graphs that have a specific interpretation in various fields like social sciences, biology, and computer science.
- Networks are mathematical **graphs**, with vertices (nodes) and edges
- They are **simple** abstractions that let us **model** problems involving **relationships** between individuals, groups, organizations, and societies.

Examples of Networks



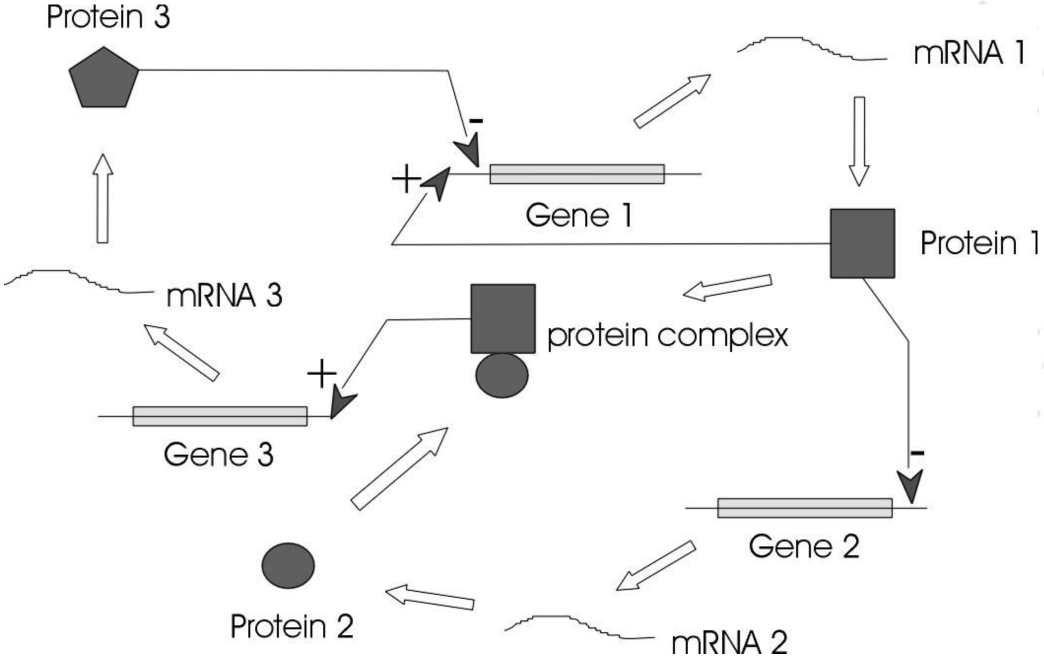


V. Krebs. Uncloaking terrorist networks. First Monday. 7(4). April 1, 2002.

Network Analysis

Examples of Networks

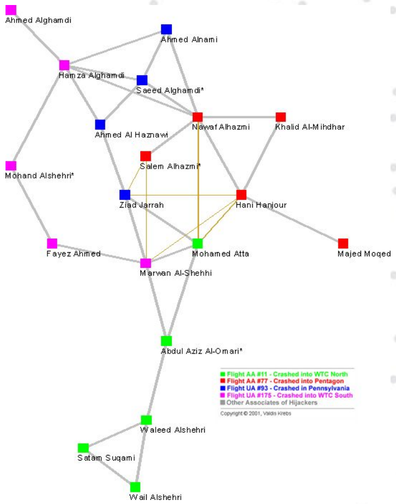
Gene Regulatory Network



Source: http://www.zaik.uni-koeln.de/bioinformatik/regulatorynets.html.en

Network Properties

- Network properties only depend on the **structure** (or topology) of the network
  - Not on the labeling of the network
  - Not on the drawing of the network



Networks

(Quiz)What are other real world networks?