University of Virginia

CS 2100: Data Structures & Algorithms 1

Trees

~Introduction to Trees and Tree Traversals~

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements / Reminders

- **Reminder of Homework Late Policy:** [Announcement sent 02/14/2022]

  - "Homework 1 (coding)" for each module:
    - Official due date: **Wednesday** by 11:59pm ET
    - Late period (with 10% penalty): 1 week; until the following Wednesday by 11:59pm ET

  - "Homework 2 (analysis)" for each module *[if applicable]*:
    - Official due date: **Friday** by 11:59pm ET
    - Late period (with 10% penalty): 3 days; until following Monday by 11:59pm ET

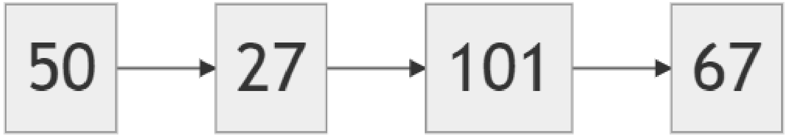
  - Manage your time wisely, seek help (TAs or Profs) when needed, ***use grace period as your extension*** if need be.

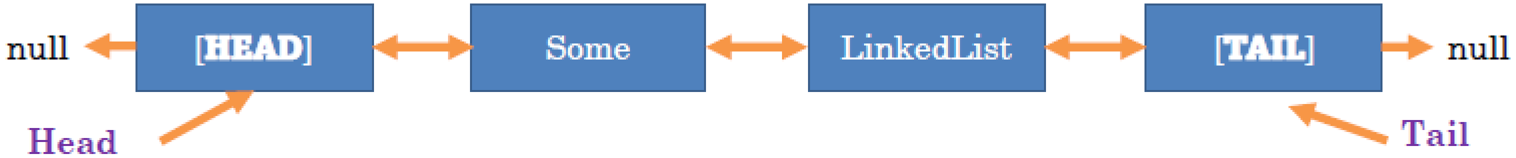
Data Structures

- If we have a good list implementation, do we need **any other** data structures?
- For computing: ***no***
  - We can compute everything with just lists (actually even less).
    The underlying machine memory can be thought of as a list
- For thinking: ***yes***
  - Lists are a very limited way of thinking about problems

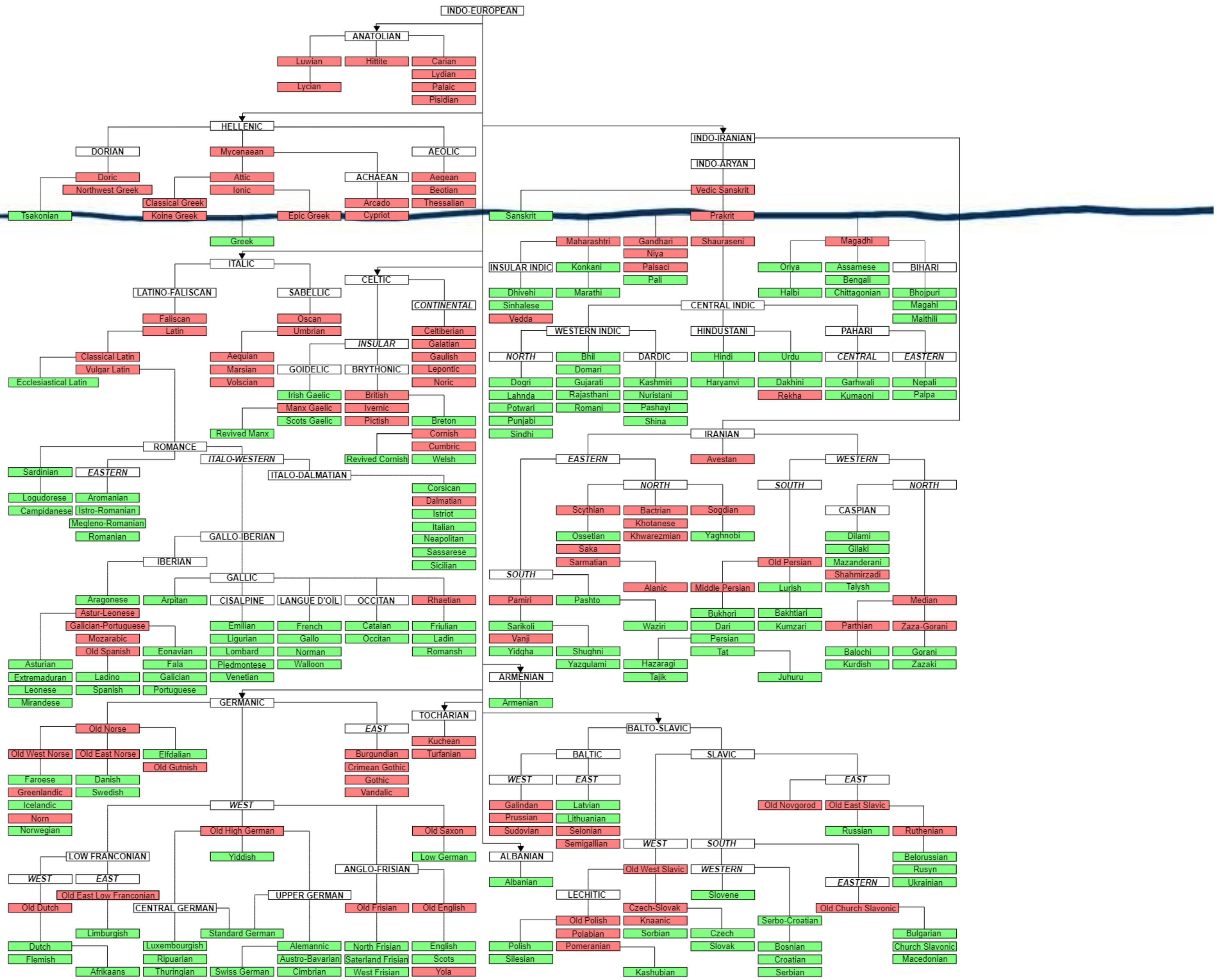
List Recursive Data Structure

- **Lists** keep things in order
  - Arrays
    - Keep things in a fixed block of memory – which is good for some operations and not as good for other operations
      - Example: Add at the end of a list vs. add at beginning or middle of list
  - Linked Lists
    - Use reference pointers between list ***nodes*** (elements) to maintain order
- **List Limitations**
  - In a list, every element has direct relationships *with only two others*: the predecessor and the successor
  - Access time: Θ(n)
  - Goal: Θ(log n)





Complex Relationships: Language Tree



Why Does This Matter Now?

- This illustrates (again) important design ideas
- The tree itself is what we're interested in
  - There are tree-level operations on it ("ADT level" operations)
  - A **tree** is an abstract data type!
- The implementation is a recursive data structure
  - There are recursive methods inside the **node-level** classes that are *closely related* (same name!) to the **tree-level** operation
- Principles?
  - abstraction (hiding details)
  - delegation (helper classes, methods)

Data Types vs. Data Structures

- Data types can be…
  - Simple or **Composite**

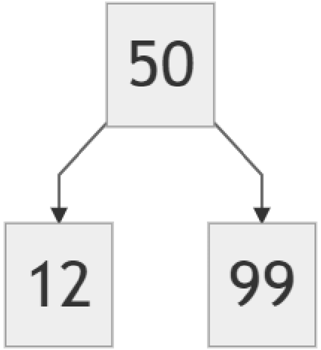
- **Data structures** are composite data types…
  - **Definition**: a collection of elements that are some combination of primitive and other composite data types

Trees

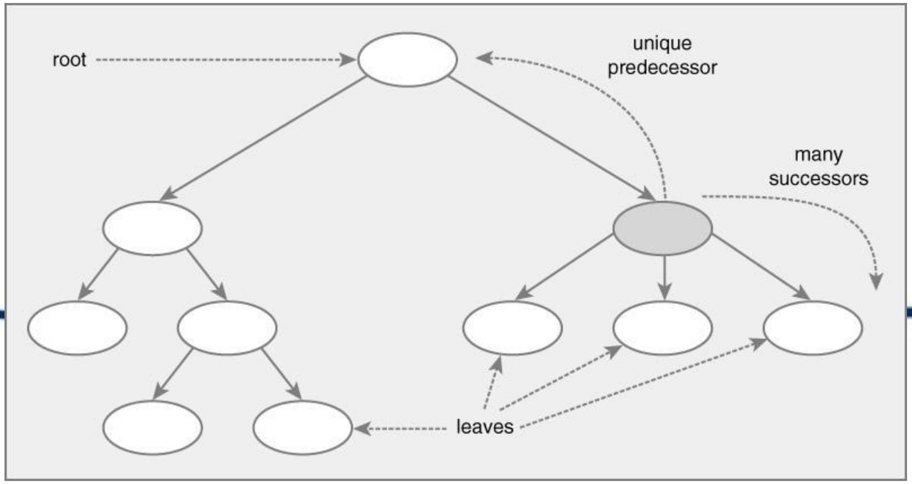
- **Trees** are a
  - **composite, hierarchical** and **graph-like** data structure in which each element has
    - Only one **predecessor**, and
    - Zero, one, or more **successors**

- In Computer Science, trees grow **down**, not up!
  - Predecessors are **up**
  - Successors are **down**

- **A tree is a special case of a list**



Tree Terminology



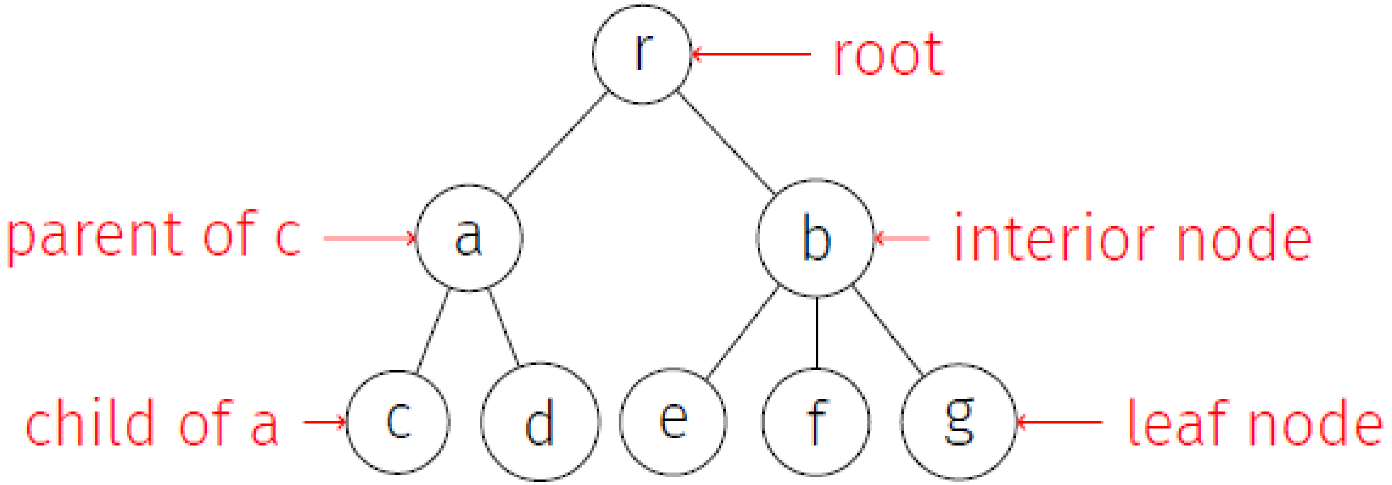
- Trees are composed of:
  - **Nodes**
    - Elements in the data structure (hold data)
    - Only one parent (***unique predecessor***)
    - Zero, one, or more children (***successors***)
    - **ROOT** node: **top** (or start) node; with no parent; there is only one root
    - **LEAF** nodes: nodes without children (***terminal***)
    - **INTERNAL** node: nodes with children (***non-terminal***)
    - **SIBLING** nodes: nodes with the same parent
    - Measure of **DEGREE**: how many children
  - **Edges**
    - Link parent node with children node (if applicable)

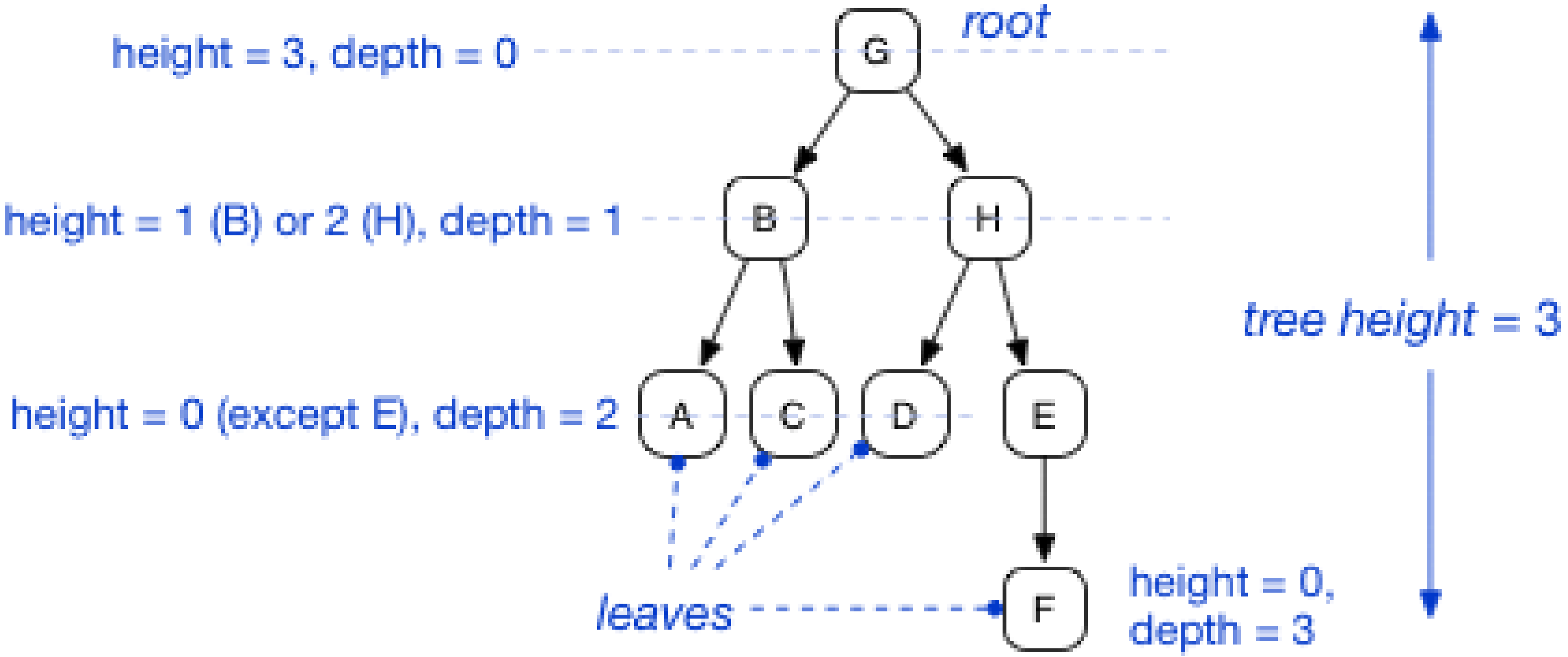
Tree Terminology ~ Relating to Height, Depth, Path

- **Height and Depth**
  - **HEIGHT** of a **node**: is the ***longest*** **path** ***(# edges)*** from that node to a **leaf**
    - Thus, all **leaves** have a height of **zero** (0)
  - **HEIGHT** of a **tree** is the ***maximum*** **depth** ***(# edges)*** of a node in that tree
    - Height of a tree = height of the **root**
  - **DEPTH** of a node: length of the **path** ***(# edges)*** from the **root** to that node
  - **PATH**: sequence of nodes n_1, n_2, ..., n_k such that n_i is parent of n_{i+1} for $1 \leq i \leq k$
  - **LENGTH**: number of **edges** in the **path**
  - **INTERNAL PATH LENGTH**: sum of the **depths** of all the nodes

Trees

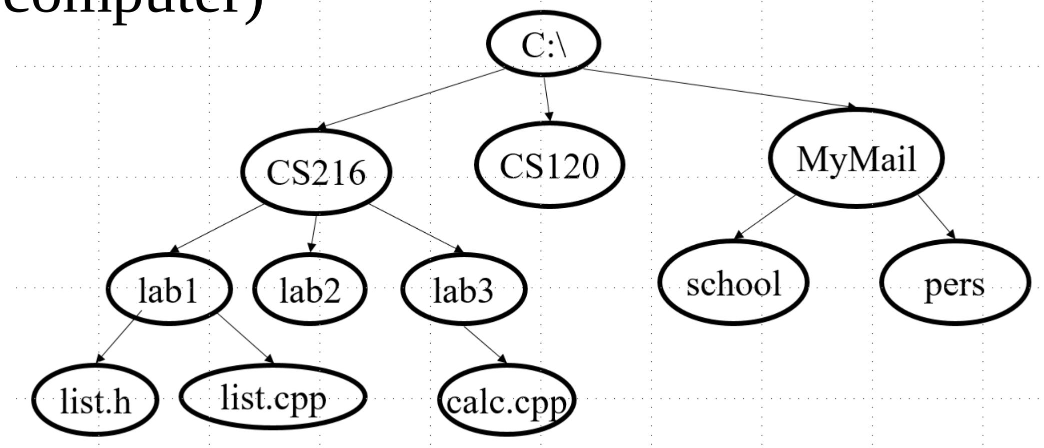
Tree height = 2





Trees are Important

- Trees are important for cognition and computation.
  What are some examples of trees and tree usages?
  - **Parse trees:** language processing, human or computer (compilers)
  - **Family (genealogy) trees** (can be complicated with some complex family relationships)
  - **The Linnaean taxonomy** (kingdom, phylum, ..., species)
  - **File systems** (directory structures on a computer)
  - … others?



Tree Definitions and Terms

- **Binary tree:**
  - A tree in which each node has at most **two** (2) children
  - Children denoted as **left child** or **right child**
- **General tree** definition:
  - A set of nodes T (*possibly empty*) with a **distinguished node**, the **root**
    - All other nodes form a set of disjoint subtrees T_i, in which
      - each is a tree in its own right
      - each is connected to the root with an edge
    - Note the **recursive definition**
      - Each node is the root of a ***subtree***
  - A tree with no nodes → **null** or empty tree

Trees: Recursive Data Structure

- **Recursive data structure:** a data structure that contains references (or pointers) to an instances of that same type

```
public class TreeNode<E> {
   private E data;
   private TreeNode<E> left;
   private TreeNode<E> right;
     …
 }
```

- Recursion is a natural way to express many data structures
- For these, it’s natural to have recursive algorithms
- **Tree operations may come in two flavors:**
  - **NODE-SPECIFIC (NODE CLASS)** (e.g. `hasParent()` or `hasChildren()`)
  - **TREE-WIDE (TREE CLASS)** (e.g. `size()` or `height()`) – requires tree traversal

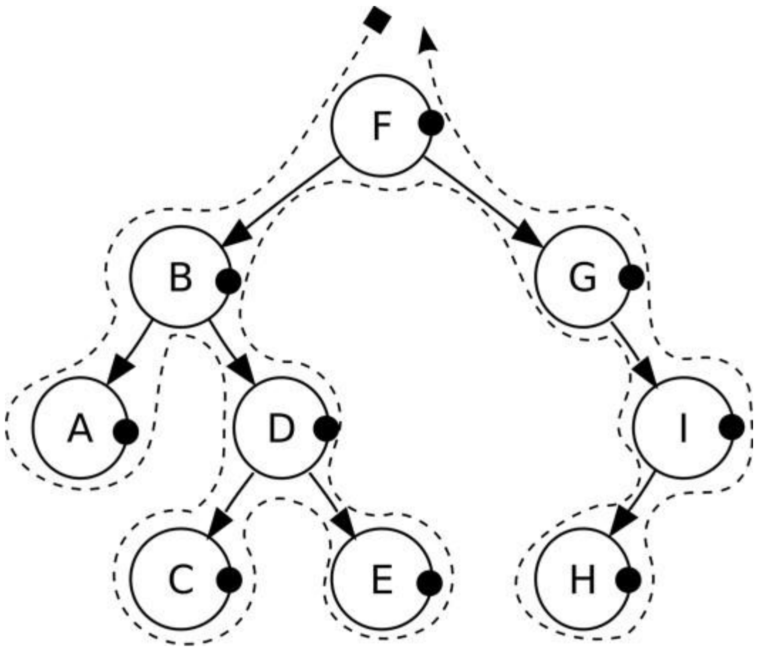
Tree Traversals

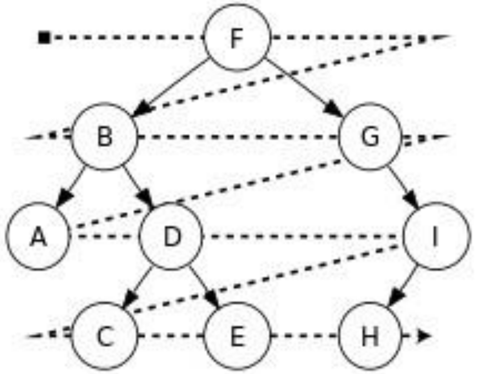
Some Motivation…

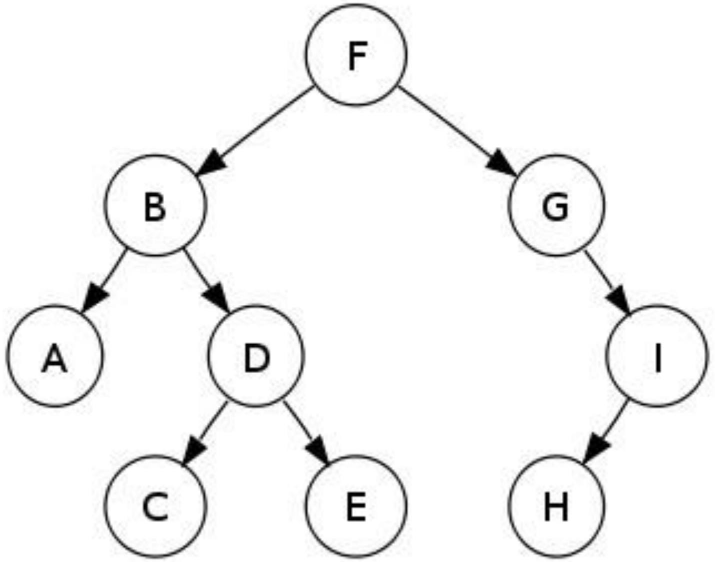
- Lists are great for keeping objects in order. They're less useful for searching
- **Searching** an unsorted list → O(n) (e.g. *linear search*)
- **Searching** a sorted list → O(lg n) (e.g. *binary search*)
  - However, takes O(n lg n) to sort…
  - And must be re-sorted as the list changes
- *We know how to traverse a list – the order is obvious… but for other structures?*

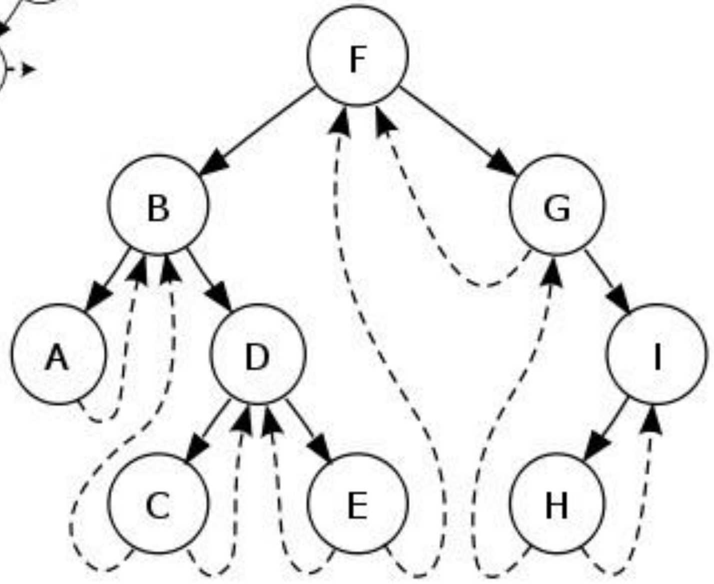
Tree Traversals – How?

- *There are many different ways to do this!*









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
- …

Tree Traversals

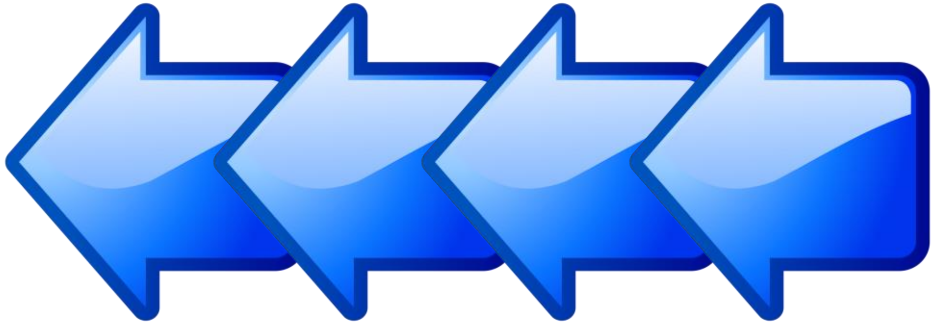
- A **tree traversal** is a specific order in which to trace the nodes of a tree
  - Visit *every* node ***once***
- There are **three** common tree traversals for binary trees: (**depth-first**)
  1. pre-order
  2. in-order
  3. post-order
  - This order is applied ***recursively***

Tree Traversals

- In each technique, the **left** subtree is traversed recursively, the **right** subtree is traversed recursively, and the **root** is visited
- What distinguishes the 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)

Tree Traversals

- In each technique, the **left** subtree is always traversed (recursively) **BEFORE** the **right** subtree is traversed!



Preoder, Inorder, Postorder

- 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

Preorder Traversal:
1. Visit the **root**
2. Traverse **left** subtree
3. Traverse **right** subtree

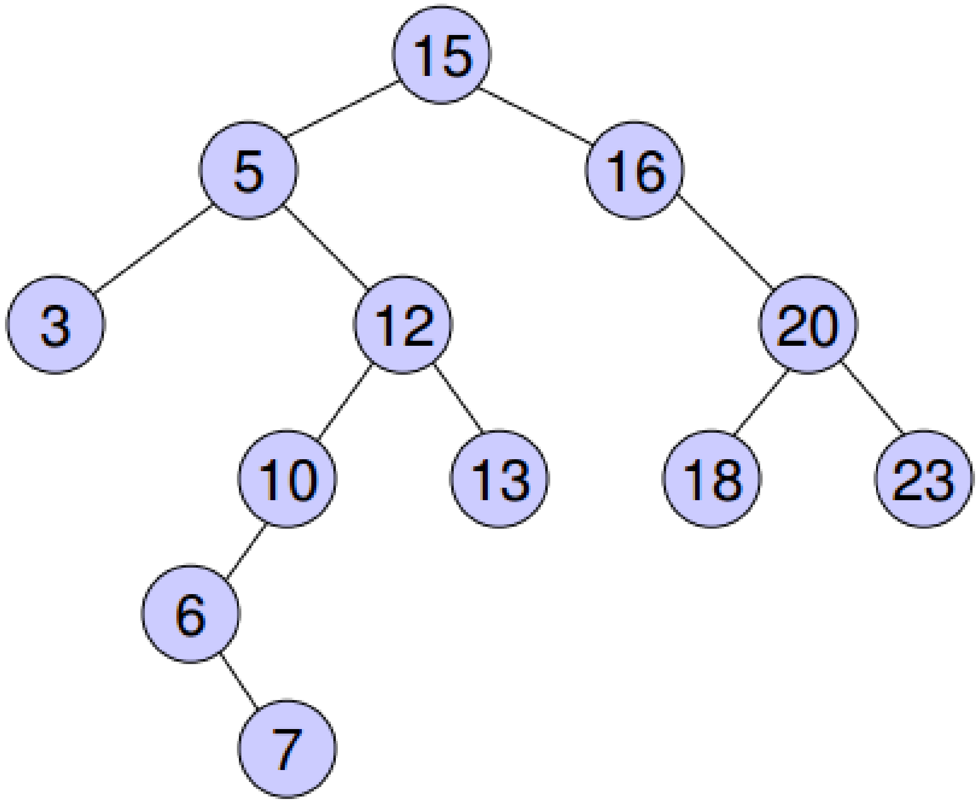
Inorder Traversal:
1. Traverse **left** subtree
2. Visit the **root**
3. Traverse **right** subtree

Postorder Traversal:
1. Traverse **left** subtree
2. Traverse **right** subtree
3. Visit the **root**

Tree Traversal Example [*3 methods*]

Let's do an example first…

(Notice: this is a Binary Search Tree!)



- **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)

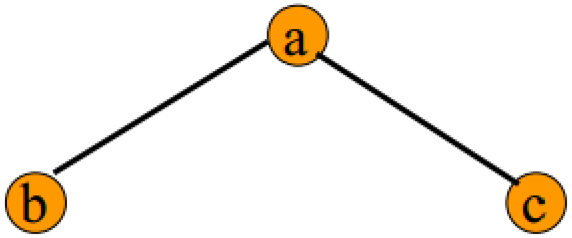
  3, 7, 6, 10, 13, 12, 5,

  18, 23, 20, 16, 15

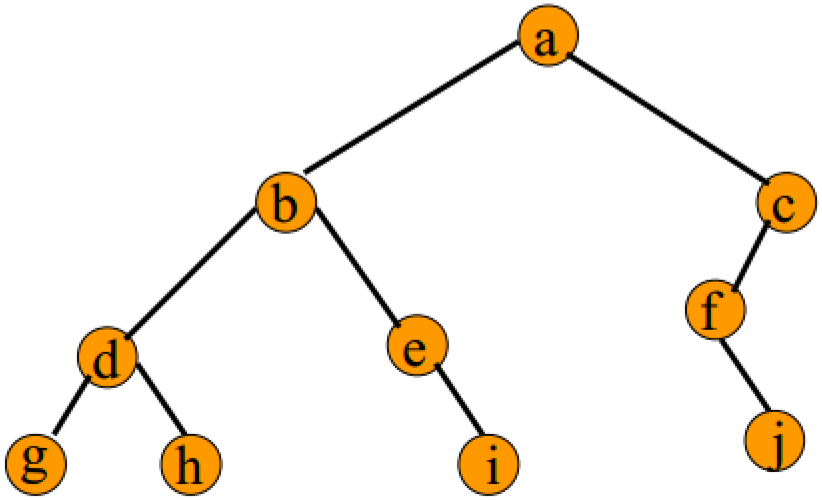
Pre-order Traversal

- Prints in order: **root**, left, right
- It is also the simple

depth-first search



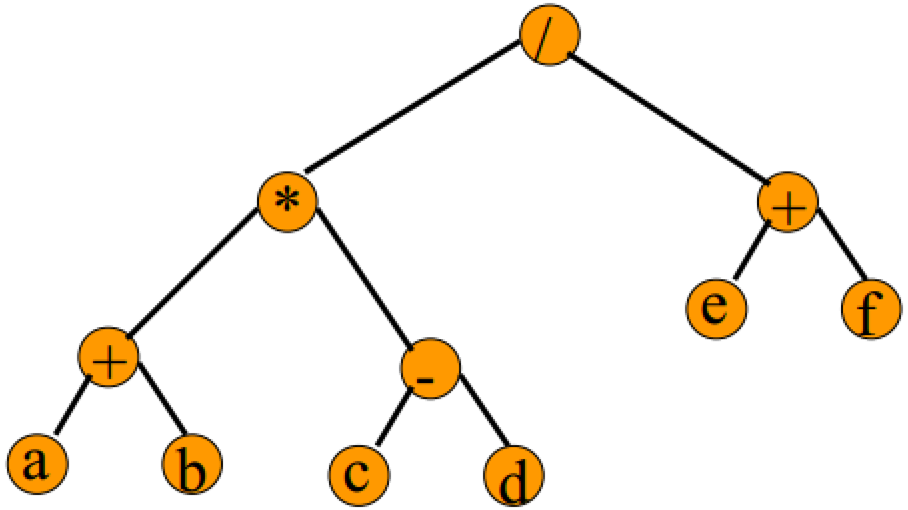
a b c



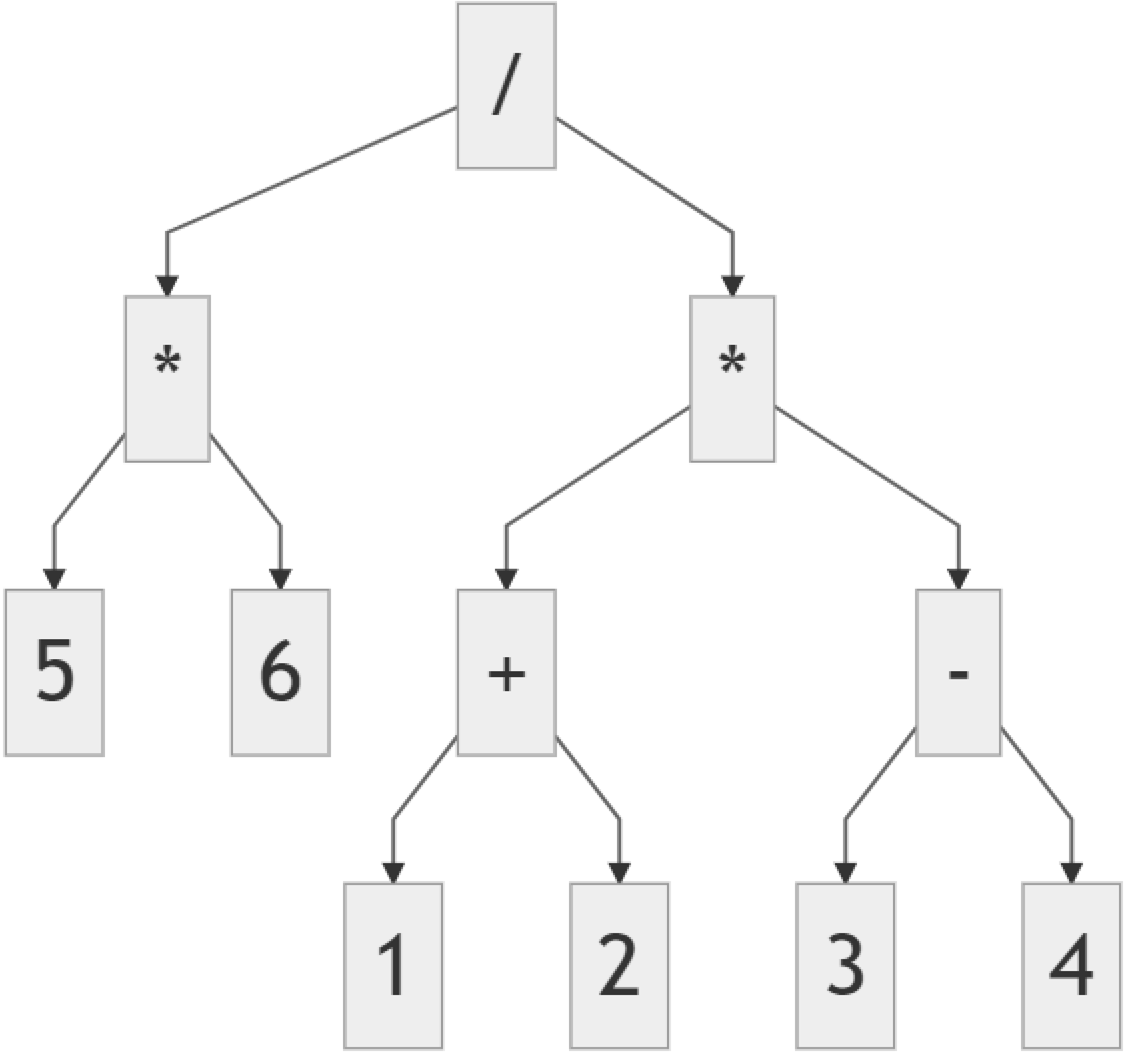
a b d g h e i c f j

Pre-order Traversal

- Gives **prefix** form of expression



/ * + a b - c d + e f



Pre-order: / * 5 6 * + 1 2 - 3 4

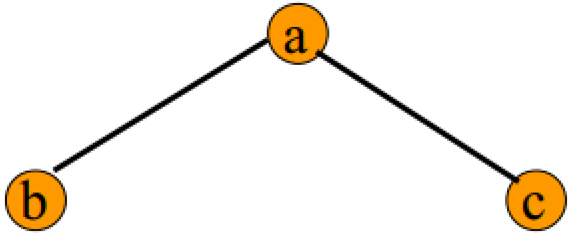
Pre-order Traversal – Java Code

- Pre-order: node first, then children (this is *pseudocode*):

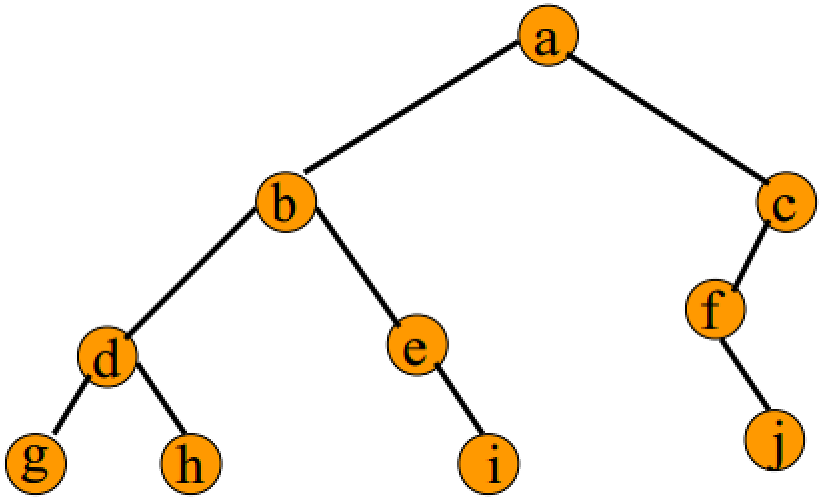
```java
public class Tree{
  private Node root;

  public void printTree(){
    printTree(root);
  }

  private void printTree(Node curNode) {
      if(curNode == null) return;

      System.out.println(curNode.value + " ");
      printTree(curNode.left);
      printTree(curNode.right);
  }
}
```

In-order Traversal

- The in-order traversal **sorts the values from smallest to largest** ***for a Binary Search Tree (BST)***
  (*See "**3 methods**" slide*)
- Prints in order: left, **root**, right



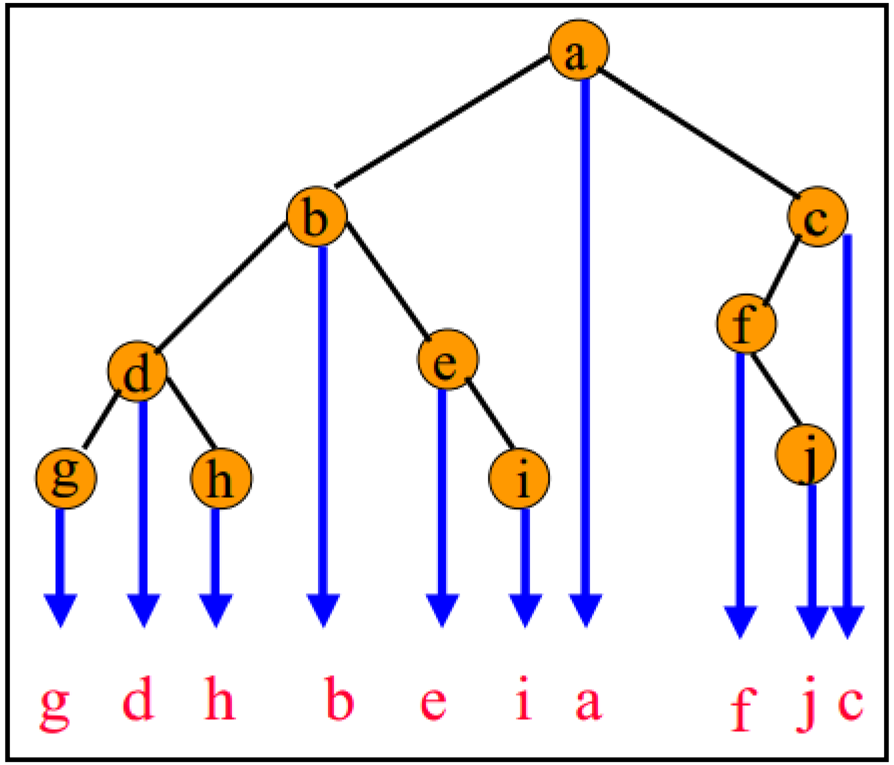
b a c

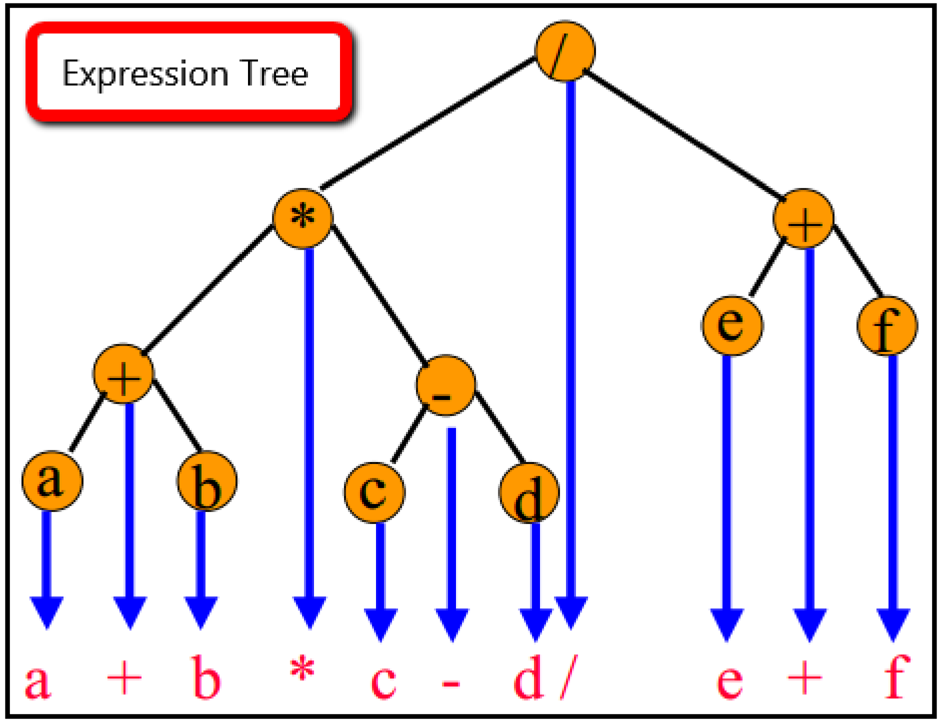


g d h b e i a f j c

In-order Traversal (Projection)

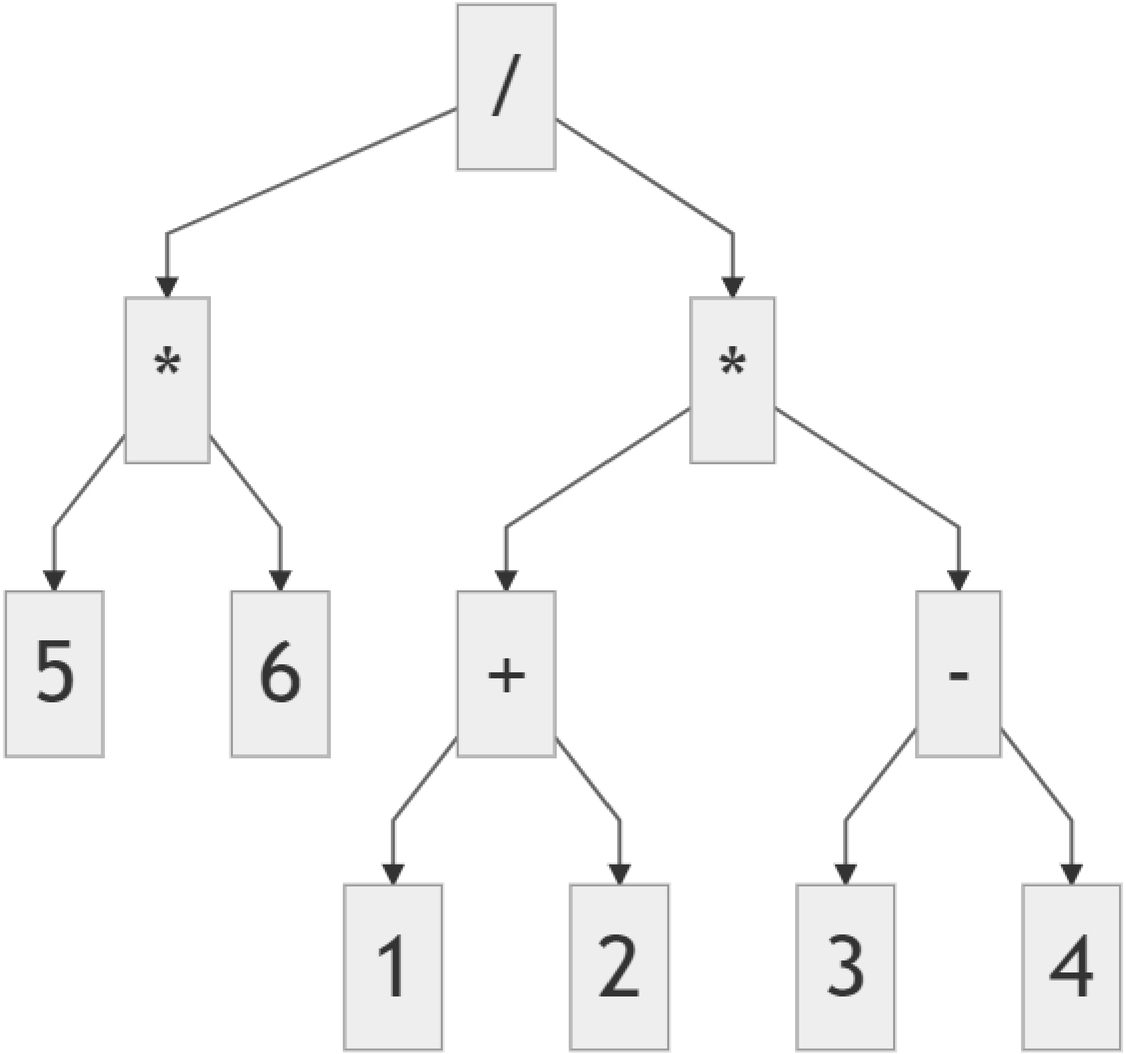
- Gives **infix** form of expression (sans parenthesis)





In-order Traversal

- Another example:



In-order: (5+6) / ((1+2)*(3-4))

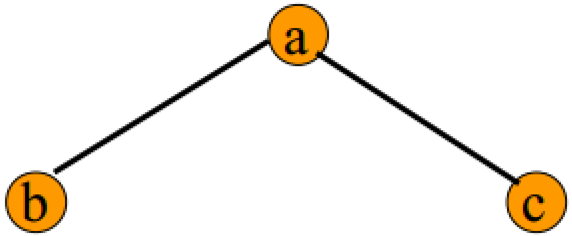
In-order Traversal – Java Code

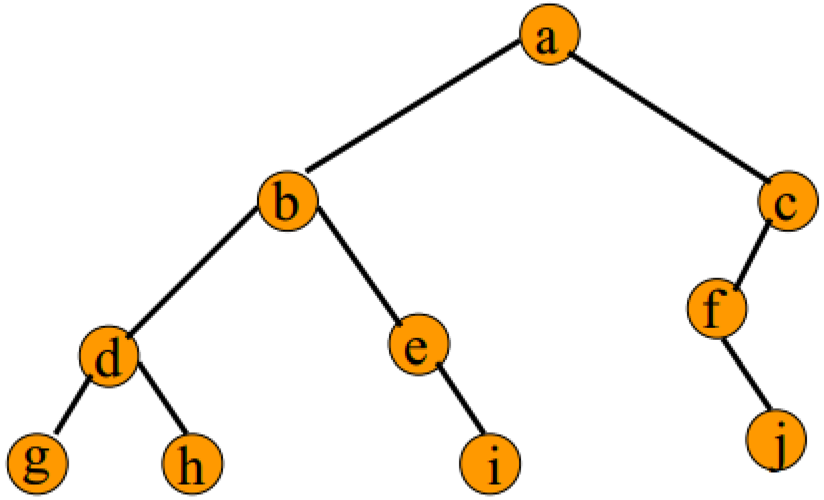
- In-order: left node first, then self, then right node:

```java
private void printTree(Node curNode) {
    if(curNode == null) return;

    printTree(curNode.left);
    System.out.println(curNode.value + " ");
    printTree(curNode.right);
}
```

Post-order Traversal

- Prints in order: left, right, **root**



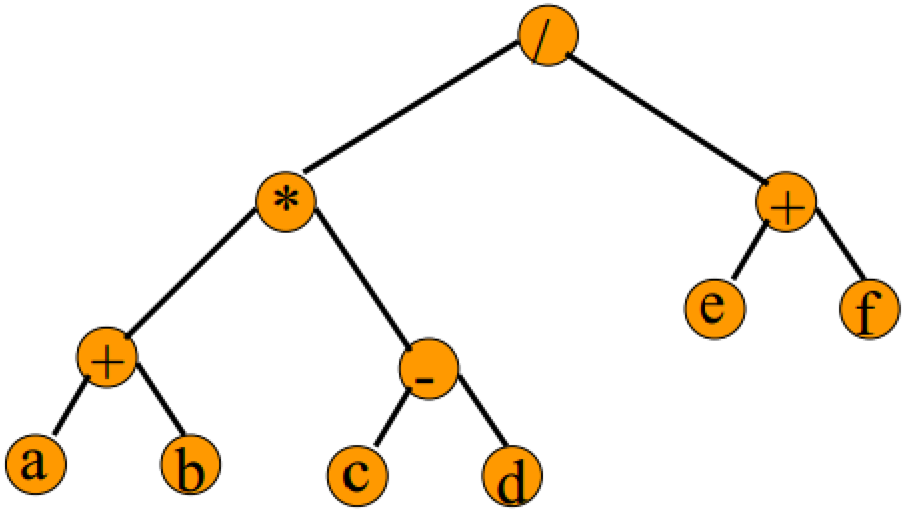


b c a

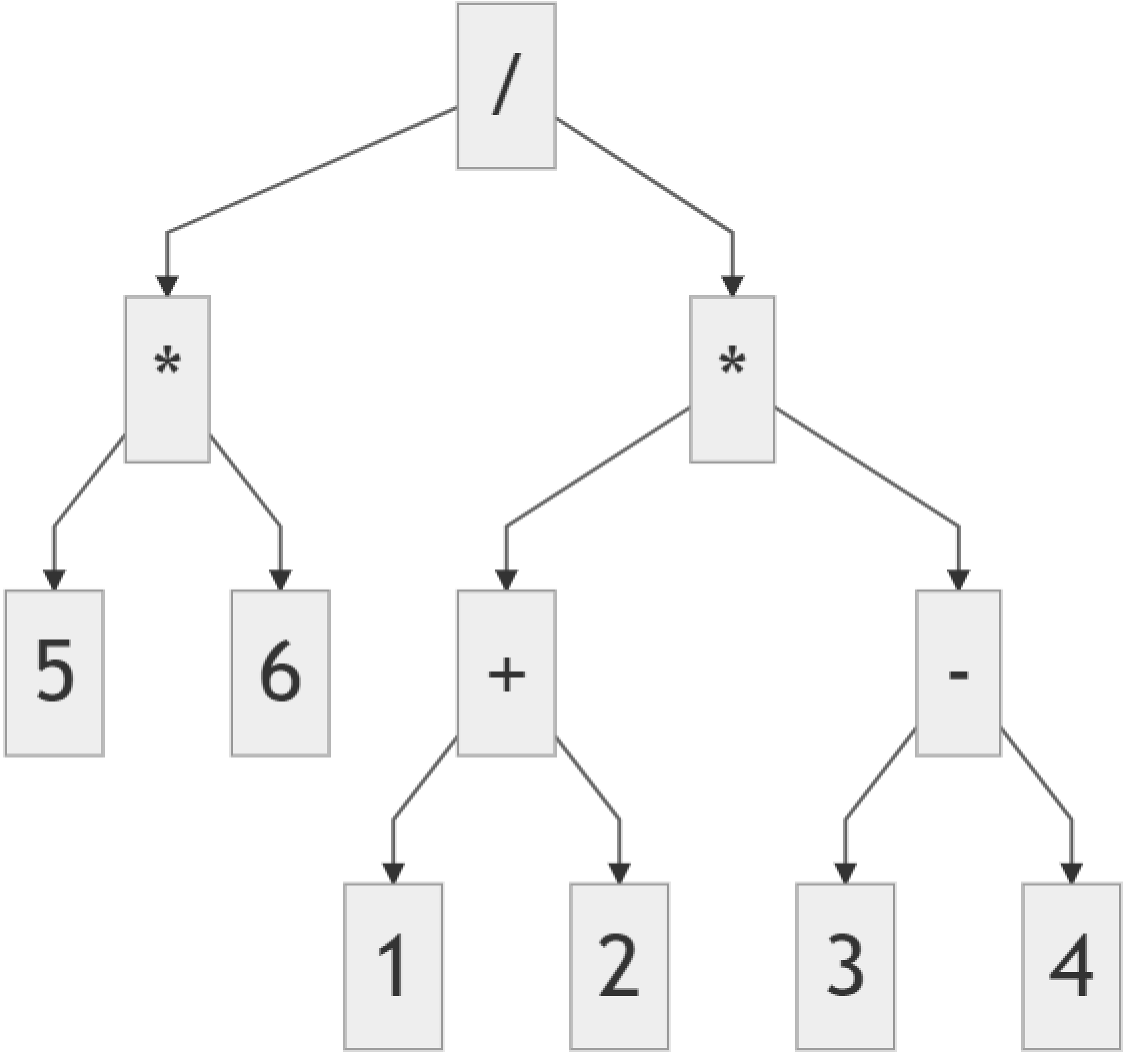
g h d i e b j f c a

Post-order Traversal

- Gives **postfix** form of expression



a b + c d - * e f + /



Post-order: 5 6 * 1 2 + 3 4 - * /

Post-order Traversal – Java Code

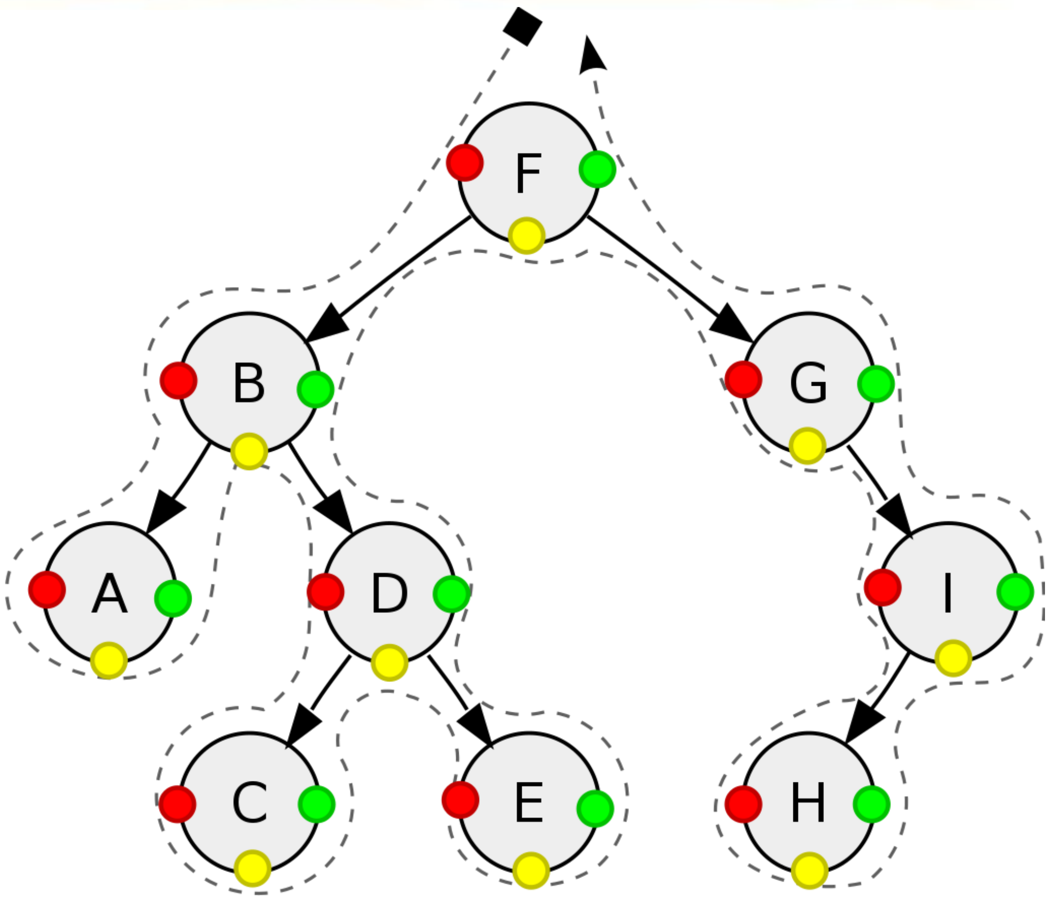
- Post-order: children first, then node
  - This method *counts the number of nodes*

```java
private void numNodes(Node root) {
    if(root == null) return 0;

    int sum = numNodes(root.left) + numNodes(root.right);
    return sum+1;
}
```

Tree Traversal "Trick"?

- Here's a trick to help you remember the traversal methods:

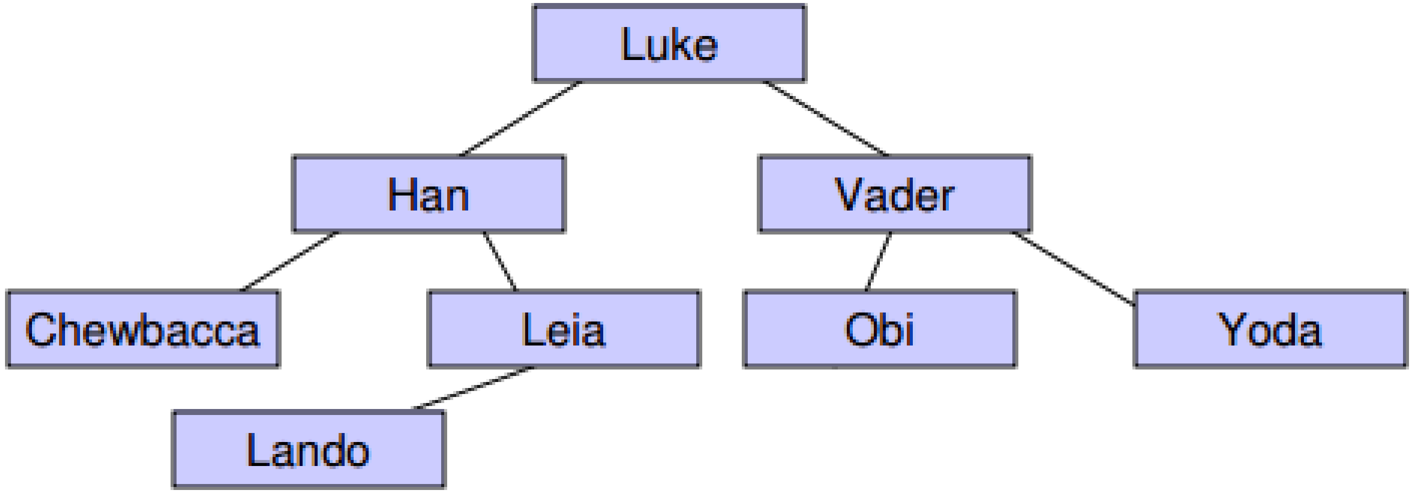
- ***pre-order* (*red*):**
  F, B, A, D, C, E, G, I, H
- ***in-order* (*yellow*):**
  A, B, C, D, E, F, G, H, I
- ***post-order* (*green*):**
  A, C, E, D, B, H, I, G, F



Picture credit: Pluke, Miles, and Jochen Burghardt (overlay)

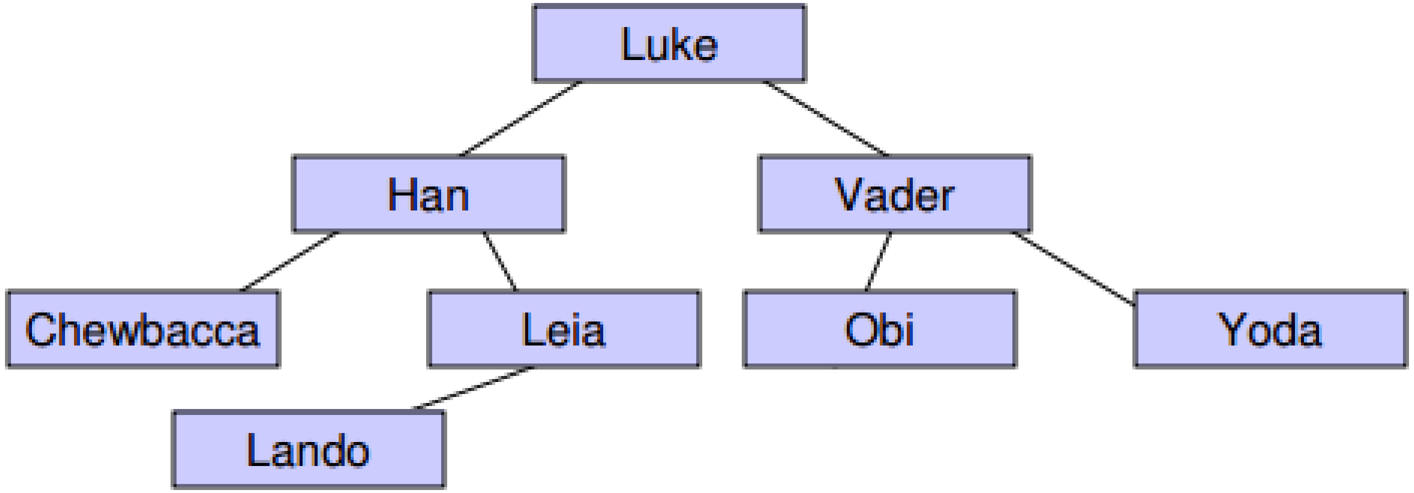
Tree Traversal Practice

- Given a tree, you are expected to know how to do the pre-, in-, and post-order traversals
- Example: Write the 3 traversals of the given tree



- In-order: ____
- Pre-order: _______________________________________
- Post-order: ______________________________________

Practice (Answers)

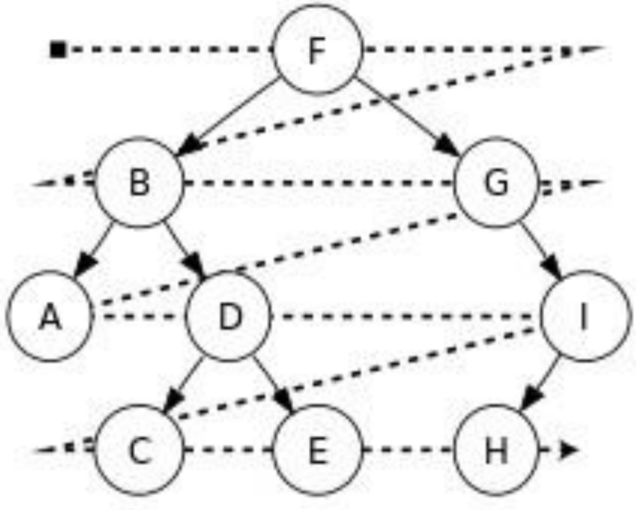


In-order: Chewbacca, Han, Lando, Leia, Luke, Obi, Vader, Yoda
Pre-order: Luke, Han, Chewbacca, Leia, Lando, Vader, Obi, Yoda
Post-order: Chewbacca, Lando, Leia, Han, Obi, Yoda, Vader, Luke

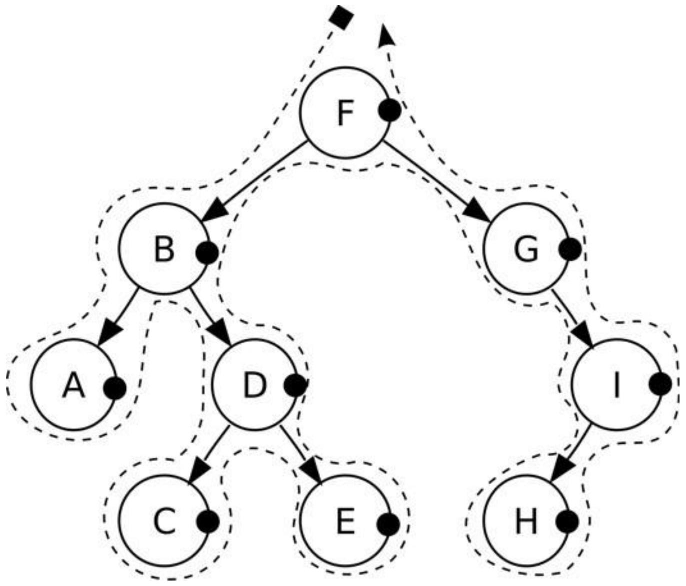
Interesting / Extra…!

Depth First vs. Breadth First

Breadth First



Depth First



Iterative Depth-First Search

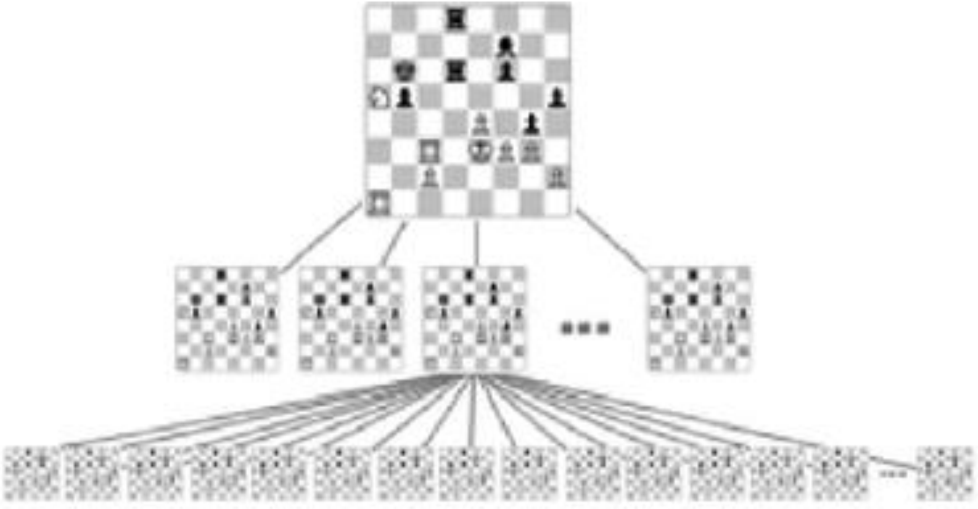
- **Depth-first search** (**DFS**) goes deeply into the tree and then backtracks when it reaches the leaves.
- DFS pseudocode algorithm uses a ***Stack***!

```
stack.push(root) // starting with empty stack, push root
while (stack is not empty):
      n = stack.pop()
      process(n) // "visit" or process this node
      // right child pushed first so that left is processed first
      if (right node not null):
            stack.push(right child)
      if (left node not null):
            stack.push(left child)
```

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

When would you use Depth-First?

- Often used when simulating games
- Populate a tree with all possible chess moves
- Perform a depth-first search to find a leaf node that ends in a **win**
- Follow the moves that lead to that leaf!



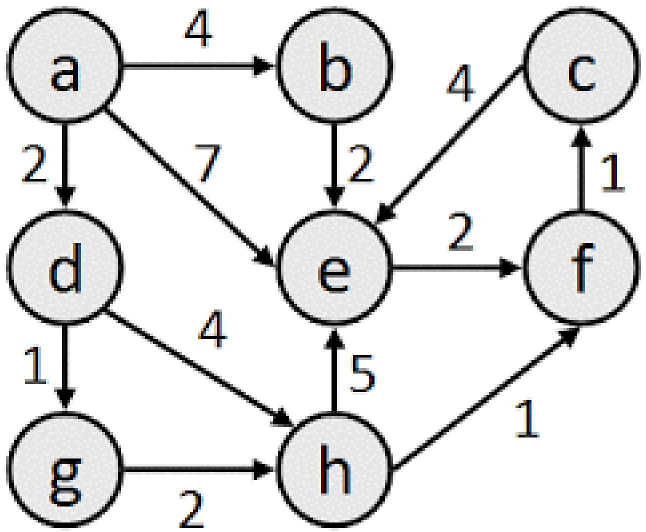
Iterative Breadth-First Search

- **Breadth-first search** (**BFS**) visits all notes on the same level before going to the next.
- BFS pseudocode algorithm uses a ***Queue***!

```
queue.add(root) // starting with empty queue, add root
while (queue is not empty):
      n = queue.remove()
      process(n) // "visit" or process this node
      // enqueue the left child before the right child
      // so that left is processed first
      if (left node not null):
             queue.add(left child)
      if (right node not null):
             queue.add(right 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)

Practice makes perfect!

Unofficial Exercise:

On this (*wavey!*) binary tree, show:

- **in-order**,
- **pre-order**, and
- **post-order** traversal