Poll Everywhere

PollEv.com/javabear text `javabear` to 22333

Suppose we `push()` 1, 2, 3, 4 onto a `Stack` (in that order), calling `pop()` at various points during and after this.

In which of the following orders could the elements *not* have been `pop()`ped?

4, 3, 2, 1 **(A)**	1, 2, 3, 4 **(B)**	2, 1, 4, 3 **(C)**
1, 3, 2, 4 **(D)**	3, 2, 4, 1 **(E)**	3, 1, 4, 2 **(F)**



Lecture 16: Trees and their Iterators

CS 2110

October 20, 2026

Today's Learning Outcomes

60. Identify parent, child, root, leaf, ancestor, and descendent nodes in a tree.

61. Given a tree, identify its size and height along with the depth and arity of its nodes.

62. Write recursive methods on general and binary trees.

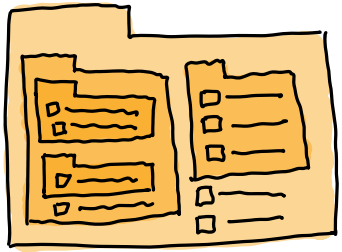
63. Given an image of a binary tree, write out the pre-order, in-order, and post-order traversals of its nodes, and vice versa.

64. Implement iterators on trees.

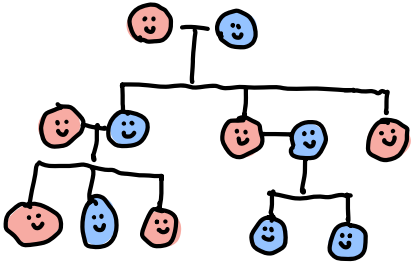
Hierarchical Data

Sometimes, a linear organization of data fails to capture its structure.

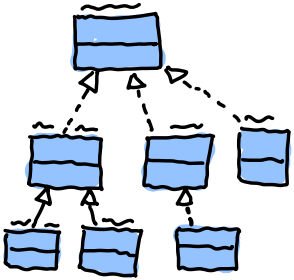
File Systems



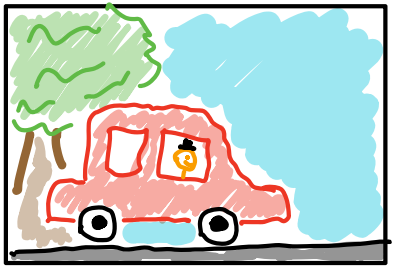
Family Trees



Inheritance Hierarchies

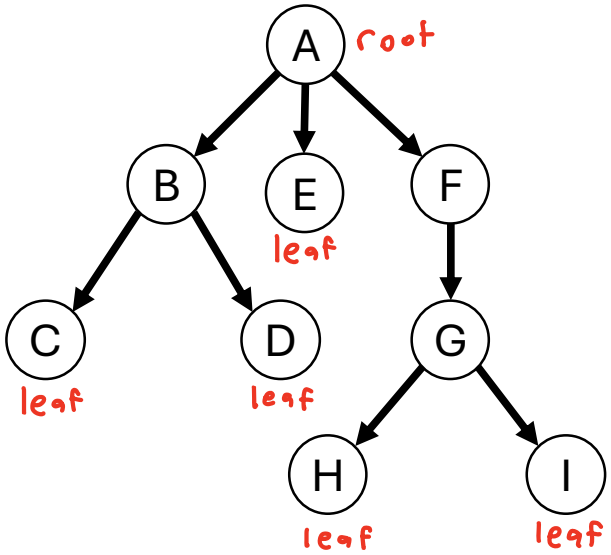


Graphics Rendering



Visualizing a Tree

A **tree** is a linked data structure with nodes arranged in a branching structure.

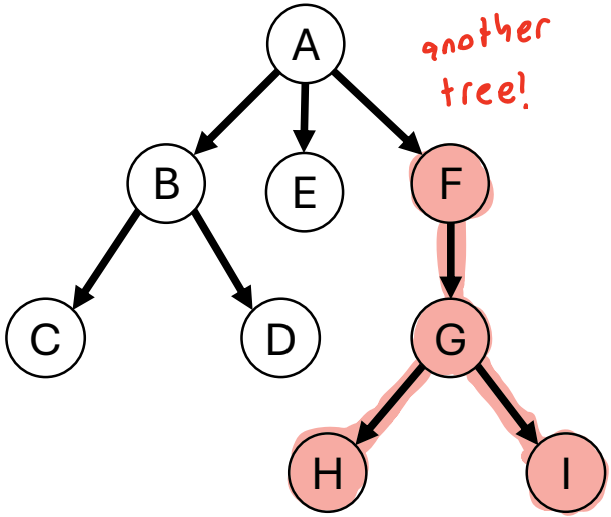


root node = source of branching (no incoming connections)

- all connections oriented away from root
- every other node has one incoming connection from its parent
- nodes each have outgoing connections to zero or more children
- leaf nodes have no children

Subtrees

The **subtree** rooted at a node v consists of v and all other nodes reachable by following connections out of v.



another tree!

Node u is an ancestor of node v (equiv. v is a descendant of u) if v belongs to the subtree rooted at u.

f is an ancestor of g, h, i (and f)

h is a descendant of g, f, a (and h)

hmm... seems like recursion is coming

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Which of the following statements is *not necessarily true* about nodes u, v, and w in a tree?

If u is an ancestor of v and v is an ancestor of w, then u is an ancestor of w. True transitivity **(A)**

If u is an ancestor of w and v is an ancestor of w, then either u is an ancestor of v or v is an ancestor of u. True w's lineage is a linked chain **(B)**

If u is an ancestor of v and u is an ancestor of w, then either v is an ancestor of w or w is an ancestor of v. False



C

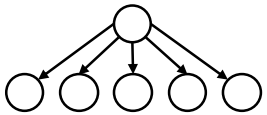
Measuring Trees

The **size** of a tree is the number of nodes that it contains.

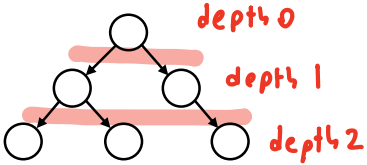
Different branching structures allow for many trees with the same size.

Height = max # connections to get from root → leaf

Depth of node = # connections needed to get to it from root



height 1
arity 5



height 2
arity 2



height 5
arity 1

Arity = max # of outgoing connections from a node

Binary trees have arity ≤ 2. General trees have any arity.

Binary Tree Representation

First Idea: Model as a de-centralized linked structure with an auxiliary Node class, this time with two pointers to its child nodes.

```
public class LinkedBinaryTree<T> {
  private static class Node<T> {
    T data;  // The element stored in this tree node
    Node<T> left; // The left child of this node
    Node<T> right; // The right child of this node
    // Node constructor
  }

  private Node<T> root; // The root node of this tree
  // LinkedBinaryTree methods
}
```

every Node reachable from here

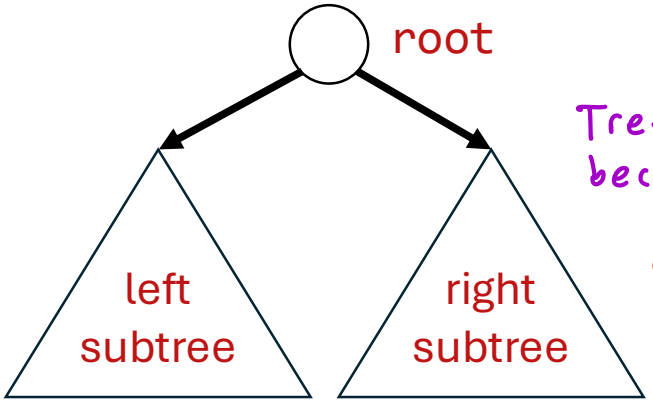
Issue: To visit all nodes, we need to worry about branching and backtracking

... clunky to track state ...

Big idea: Use runtime stack and recursion

The Recursive View of Trees

A `BinaryTree` consists of the element at its root, plus two (possibly `null`) references to its left and right *subtrees*.



↑ not children!

Tree class itself (not inner node class) becomes recursive.

Client calls methods on the root subtree (i.e., whole tree) which recursively delegates computation to its subtrees.



Coding Demo: `BinaryTree` class



Key Ideas:

- `protected` visibility
- `abstract` `left()` and `right()` methods (as opposed to fields)

- Nested constructor calls to build `ImmutableBinaryTrees`

Recursive Methods on Trees

Big Idea: Combine the answers from the `left()` and `right()` subtrees (along with the `root`) to get the answer for the whole tree.

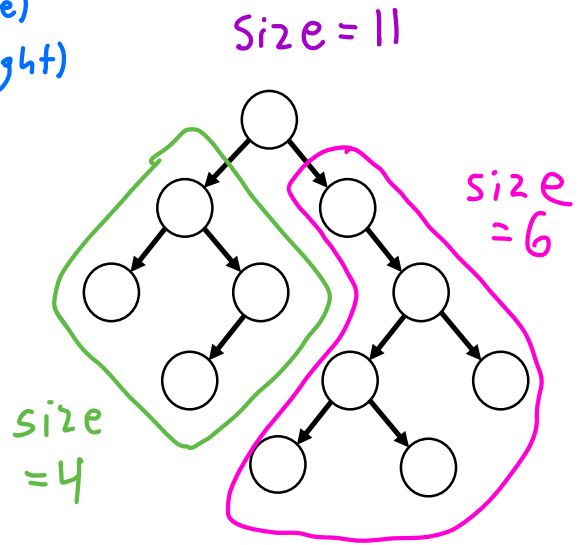
Be careful about `null` children!

O(N) time (size)
O(H) space (height)

BinaryTree.java

```
/** Returns the number of nodes in this tree. */
public int size() {
 return 1 +
  (left() == null ? 0 : left().size()) +
  (right() == null ? 0 : right().size())

}
```



Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

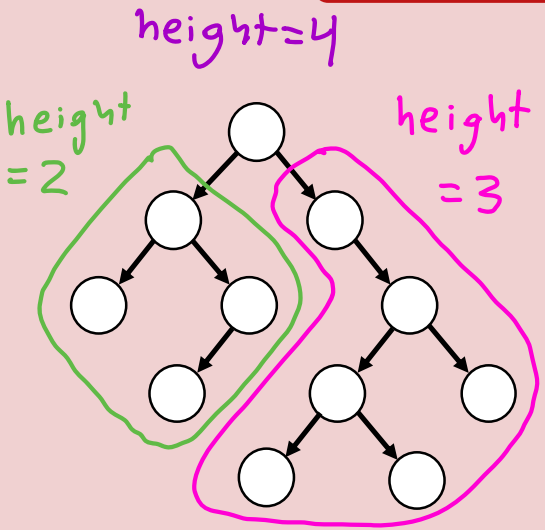
What values should we fill in for x and y in this definition?

BinaryTree.java

```
/** Returns the height of this tree. */
public in height() {
  return [ x ] + Math.max(
    left() == null ? [ y ] : left().height(),
    right() == null ? [ y ] : right().height()
  );
}
```

x = 1 : root adds another level

y = -1 : tree of one node has height 0



Exercise:

Work with your neighbors to complete the definition of this method.

BinaryTree.java

```
/** Returns the number of leaves in this binary tree. */
public int numLeaves() {
  if (left()==null && right()==null) { return 1;}
  return   (left()==null ? 0 : left().numLeaves())
         + (right()==null ? 0 : right().numLeaves());

}
```

this is a leaf

Tree Traversals

When we **traverse** a tree, we "visit" each of its nodes once in some systematic order.

For branching (non-linear) structures, this order becomes non-obvious...

From the recursive view, we can split traversal into 3 "steps":

- visit the tree root
- recursively traverse the left subtree
- recursively traverse the right subtree

Different ways to "slot" root visit in lead to three different traversal strategies.

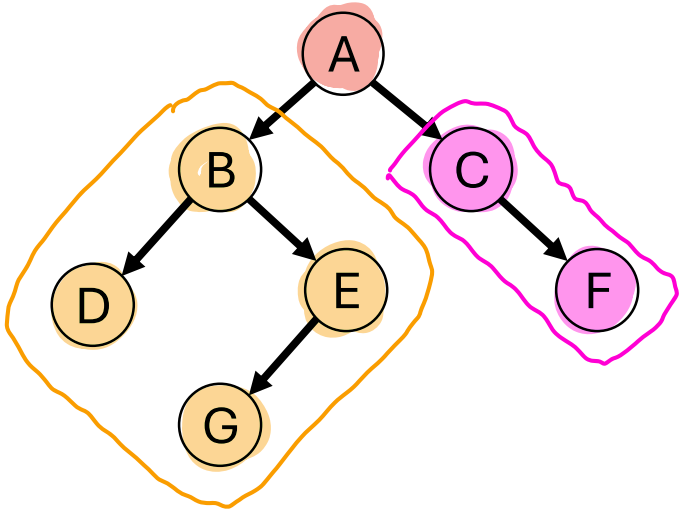
* Different traversal orders useful for different applications (e.g., sorting, parsing)

Pre-Order Traversal

Visiting the root happens ***before*** traversing the subtrees.

1. Visit the root
2. Pre-order traverse the left subtree
3. Pre-order traverse the right subtree





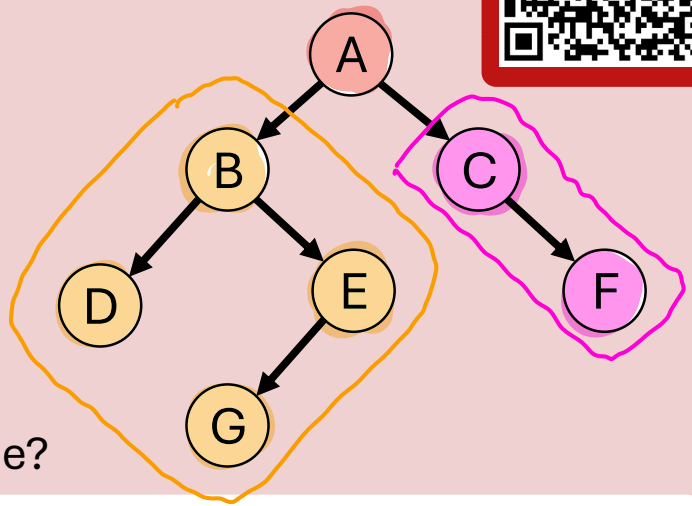
Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

In an **In-Order Traversal**, we:

1. In-order traverse the left subtree
2. Visit the root node
3. In-order traverse the right subtree

DBGE A C F



Which is the in-order traversal of this tree?

A B D E G C F **(A)**

D B E G A C F **(B)**

D G B E A C F **(C)**

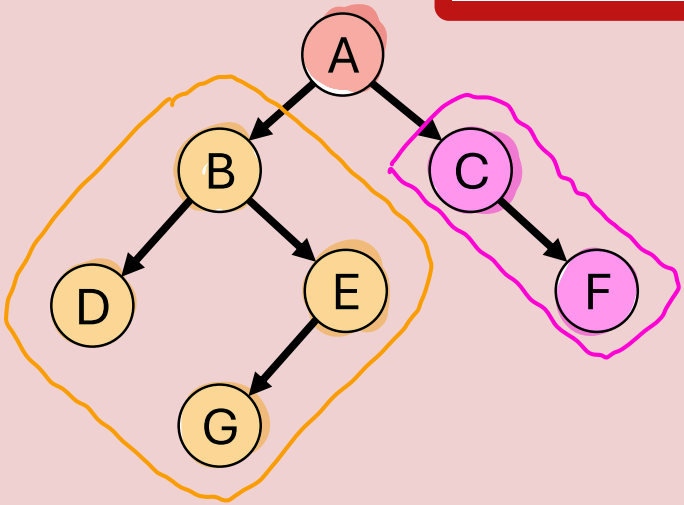
D B G E A C F **(D)**

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

In a **Post-Order Traversal**, we:

1. Post-order traverse the left subtree
2. Post-order traverse the right subtree
3. Visit the root node



DGEB FC A

Write out the post-order traversal of this tree.

Binary Tree Iterators

We can develop inner `Iterator` classes for `BinaryTree` that yield its elements in each of the three traversal orders.

We'll need to introduce field to the iterator to track the "traversal state"

- status at each level of the tree (have we visited root, left, right yet)
- model using a stack (if we're traversing lower levels, need to finish this before returning to root)

Common invariant: Top of stack will have next() root to return

Empty stack = done.

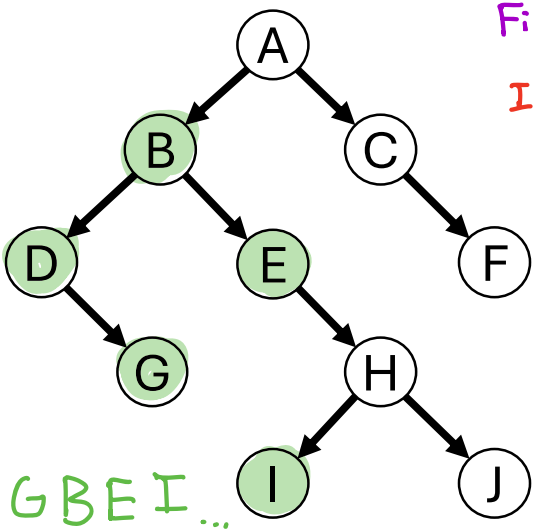


Coding Demo: Pre-Order Iterator



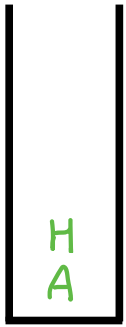
Designing an In-Order Iterator

New Challenge: Need to "drill down" to locate the first node to return (etc.)



First node to return = d ("leftmost" node)

Idea: keep following left() pointers until we reach a node without one.



cascadeLeft() helper method

strategy for next():

pop() then cascadeLeft() on right subtree



Coding Demo: In-Order Iterator

