Building Java Programs

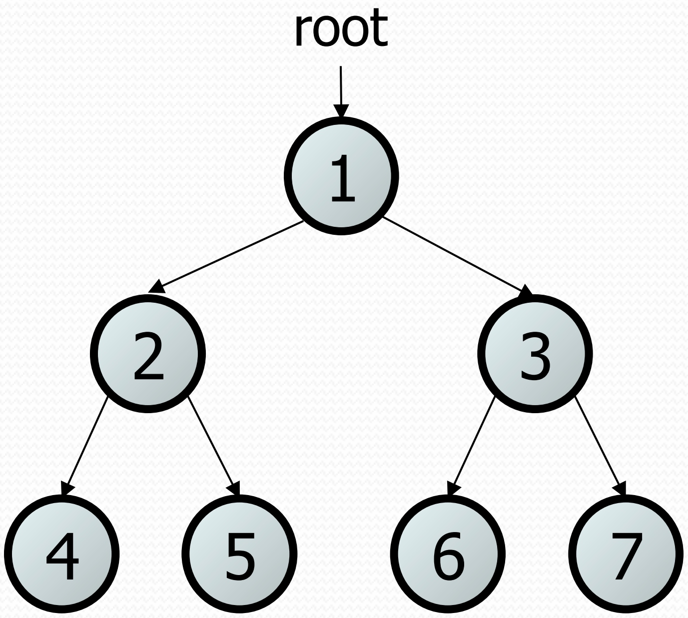
Binary Trees

reading: 17.1 – 17.3

Trees

- **tree**: A directed, acyclic structure of linked nodes.
  - *directed* : Has one-way links between nodes.
  - *acyclic* : No path wraps back around to the same node twice.

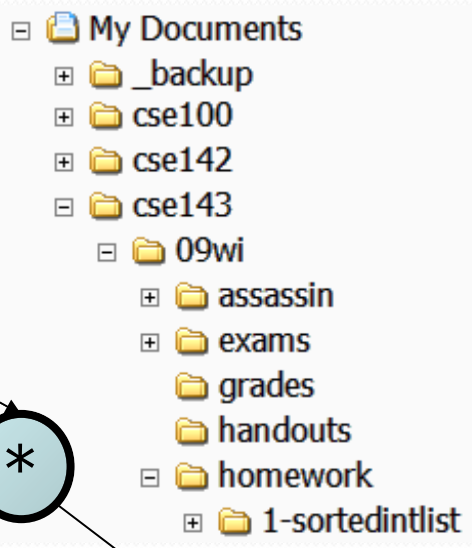
– **binary tree**: One where each node has at most two children.

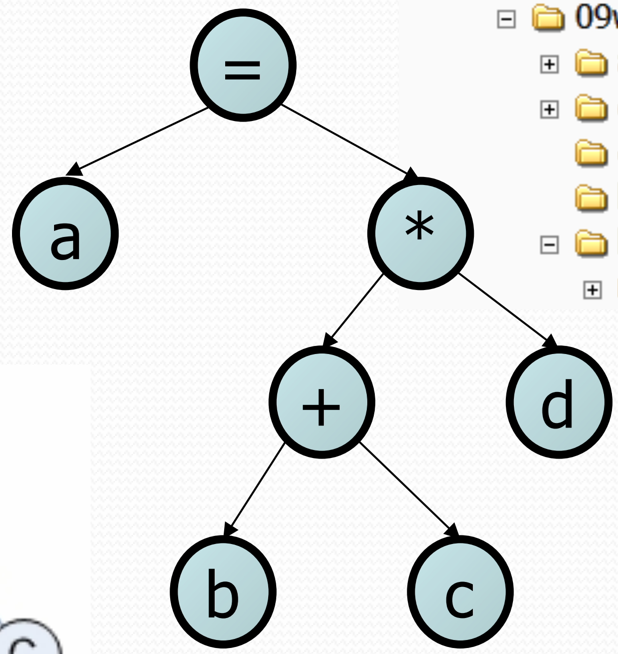


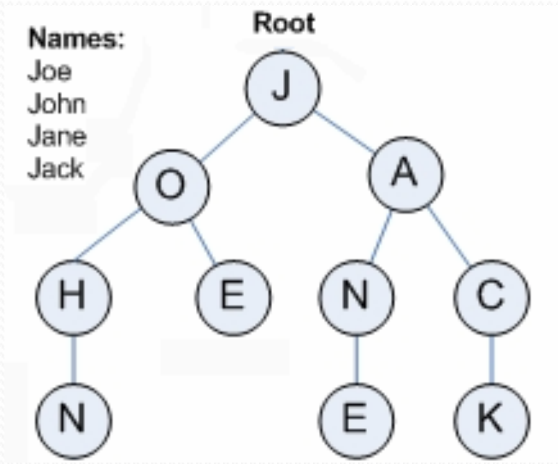
- *Recursive definition:* A tree is either:
  - empty (`null`), or
  - a **root** node that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.
      - (The left and/or right subtree could be empty.)

Trees in computer science

- folders/files on a computer
- family genealogy; organizational charts
- AI: decision trees
- compilers: parse tree
  - a = (b + c) * d;
- cell phone T9

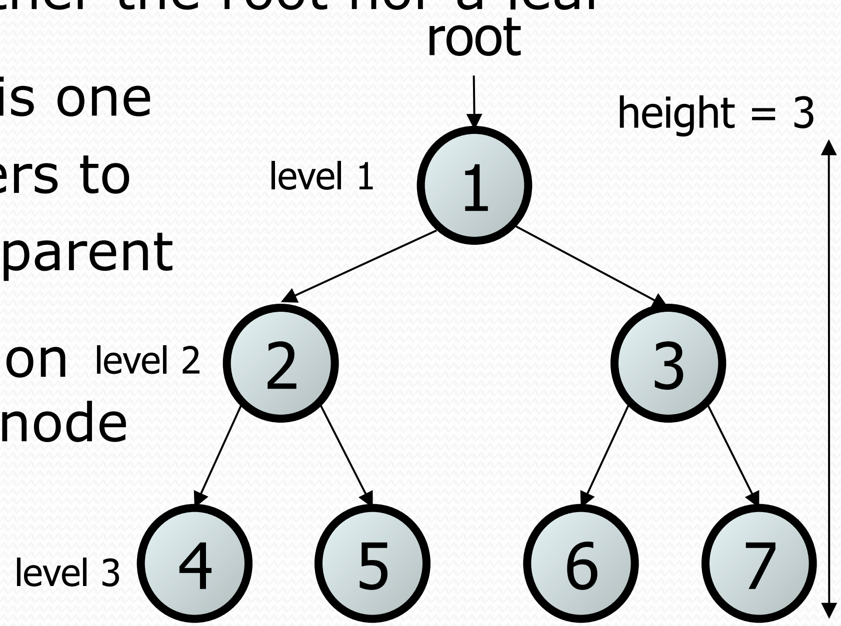






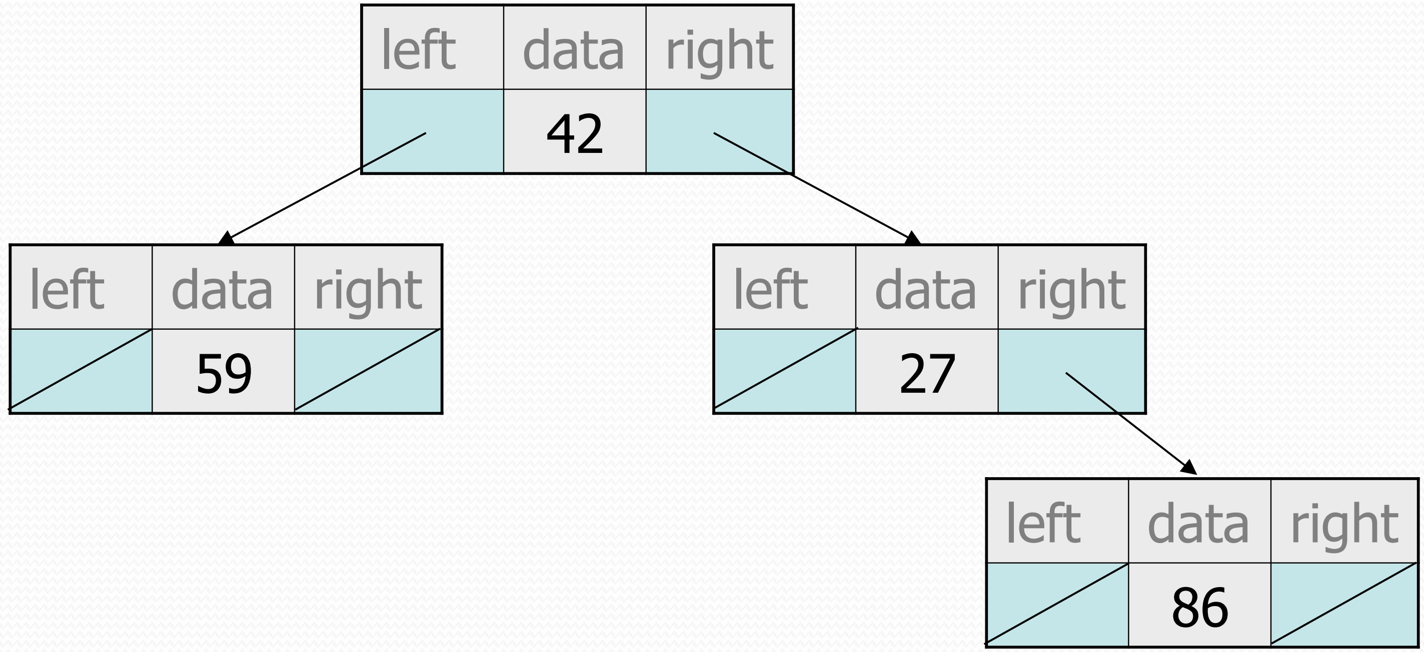
Terminology

- **node**: an object containing a data value and left/right children
  - **root**: topmost node of a tree
  - **leaf**: a node that has no children
  - **branch**: any internal node; neither the root nor a leaf
  - **parent**: a node that refers to this one
  - **child**: a node that this node refers to
  - **sibling**: a node with a common parent
- **subtree**: the smaller tree of nodes on the left or right of the current node
- **height**: length of the longest path from the root to any node
- **level** or **depth**: length of the path from a root to a given node



A tree node for integers

- A basic **tree node object** stores data, refers to left/right
  - Multiple nodes can be linked together into a larger tree



IntTreeNode class

```
// An IntTreeNode object is one node in a binary tree of ints.
public class IntTreeNode {
    public int data;              // data stored at this node
    public IntTreeNode left;      // reference to left subtree
    public IntTreeNode right;     // reference to right subtree

    // Constructs a leaf node with the given data.
    public IntTreeNode(int data) {
        this(data, null, null);
    }

    // Constructs a branch node with the given data and links.
    public IntTreeNode(int data, IntTreeNode left,
                                 IntTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

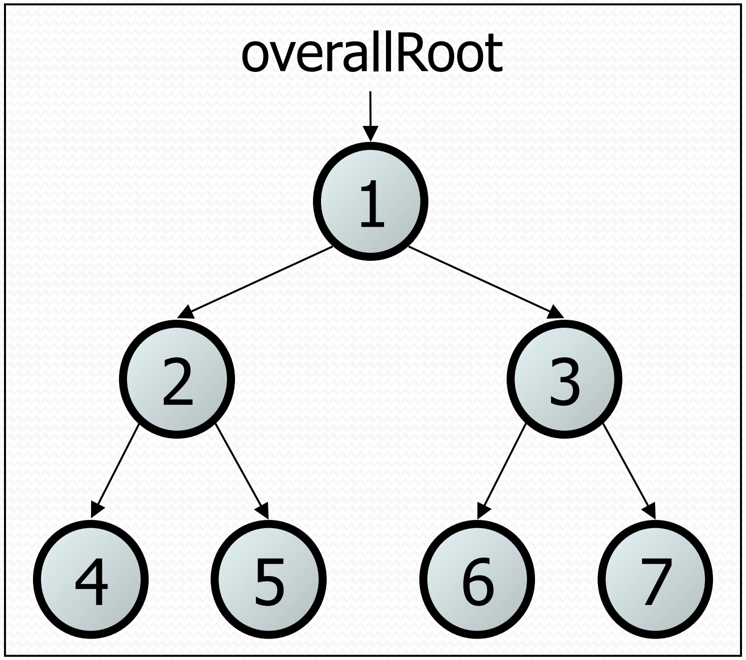
left	data	right

`IntTree` class

```
// An IntTree object represents an entire binary tree of ints.
public class IntTree {
    private IntTreeNode overallRoot;    // null for an empty tree

    methods
}
```

- Client code talks to the `IntTree`, not to the node objects inside it.

- Methods of the `IntTree` create and manipulate the nodes, their data and links between them.



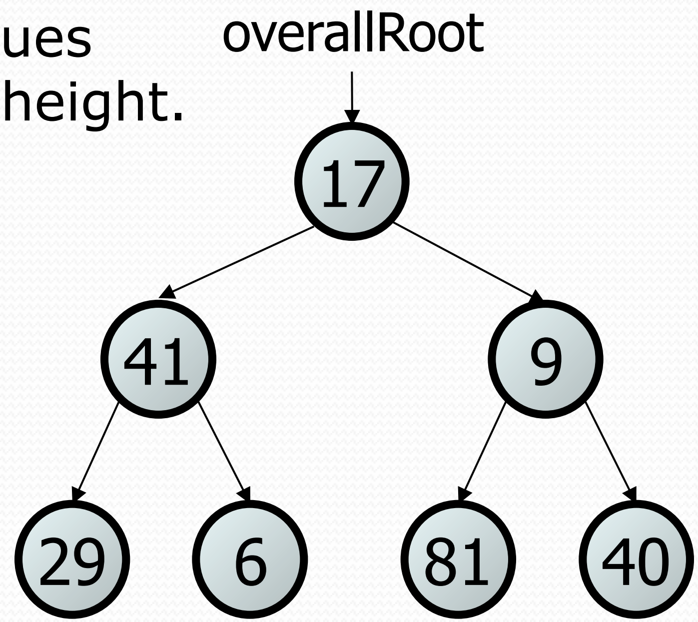
`IntTree` constructors

- For now, assume we have the following constructors:

```
public IntTree(IntTreeNode overallRoot)
public IntTree(int height)
```

  - The 2nd constructor will create a tree and fill it with nodes with random data values from 1-100 until it is full at the given height.

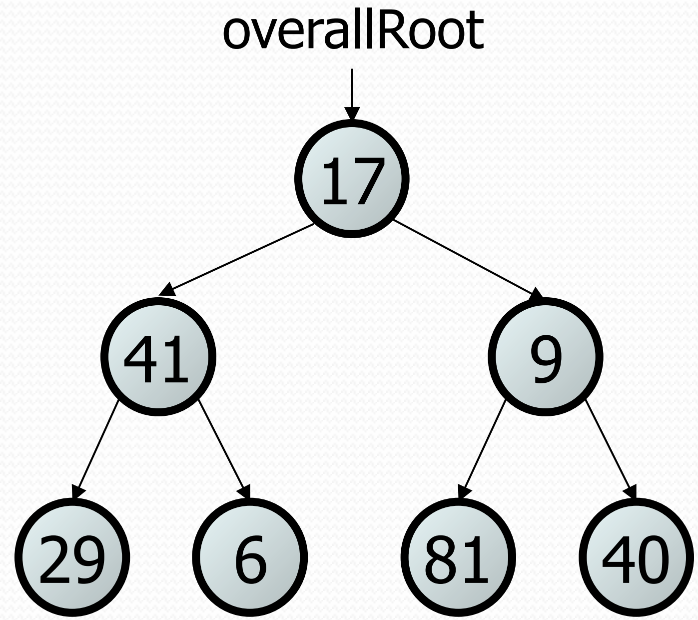
```
IntTree tree = new IntTree(3);
```



Exercise

- Add a method `print` to the `IntTree` class that prints the elements of the tree, separated by spaces.
  - A node's left subtree should be printed before it, and its right subtree should be printed after it.
  - Example: `tree.print();`

```
29 41 6 17 81 9 40
```



Exercise solution

```
// An IntTree object represents an entire binary tree of ints.
public class IntTree {
    private IntTreeNode overallRoot;    // null for an empty tree
    ...

    public void print() {
        print(overallRoot);
        System.out.println();    // end the line of output
    }

    private void print(IntTreeNode root) {
        // (base case is implicitly to do nothing on null)
        if (root != null) {
            // recursive case: print left, center, right
            print(overallRoot.left);
            System.out.print(overallRoot.data + " ");
            print(overallRoot.right);
        }
    }
}
```

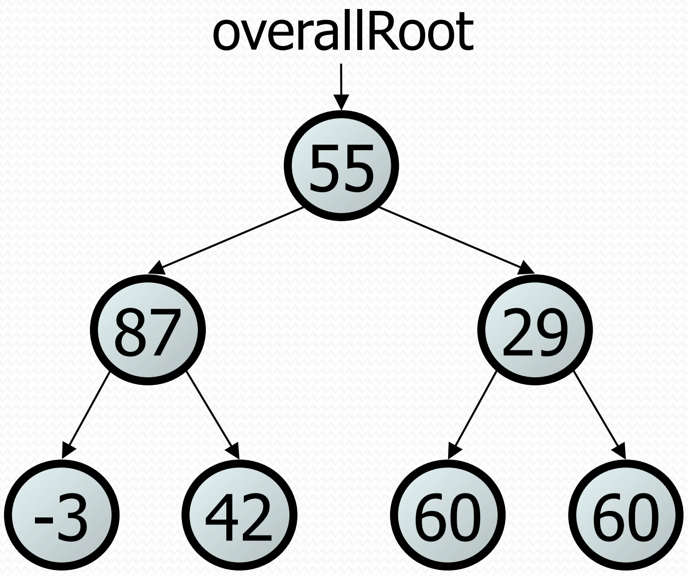
Template for tree methods

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...

    public type name(parameters) {
        name(overallRoot, parameters);
    }

    private type name(IntTreeNode root, parameters) {
        ...
    }
}
```

- Tree methods are often implemented recursively
  - with a public/private pair
  - the private version accepts the root node to process

Exercise

- Add a method `contains` to the `IntTree` class that searches the tree for a given integer, returning `true` if it is found.
  - If an `IntTree` variable `tree` referred to the tree below, the following calls would have these results:
    - `tree.contains(87)` → `true`
    - `tree.contains(60)` → `true`
    - `tree.contains(63)` → `false`
    - `tree.contains(42)` → `false`



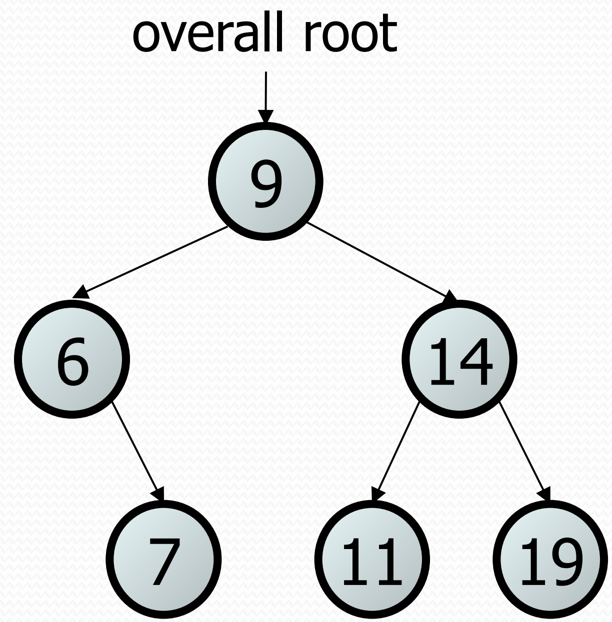
Exercise solution

```
// Returns whether this tree contains the given integer.
public boolean contains(int value) {
    return contains(overallRoot, value);
}

private boolean contains(IntTreeNode node, int value) {
    if (node == null) {
        return false;   // base case: not found here
    } else if (node.data == value) {
        return true;    // base case: found here
    } else {
        // recursive case: search left/right subtrees
        return contains(node.left, value) ||
               contains(node.right, value);
    }
}
```

Exercise

- Add a method named `printSideways` to the `IntTree` class that prints the tree in a sideways indented format, with right nodes above roots above left nodes, with each level 4 spaces more indented than the one above it.
  - Example: Output from the tree below:

```
        19

    14

        11

9

        7

    6
```



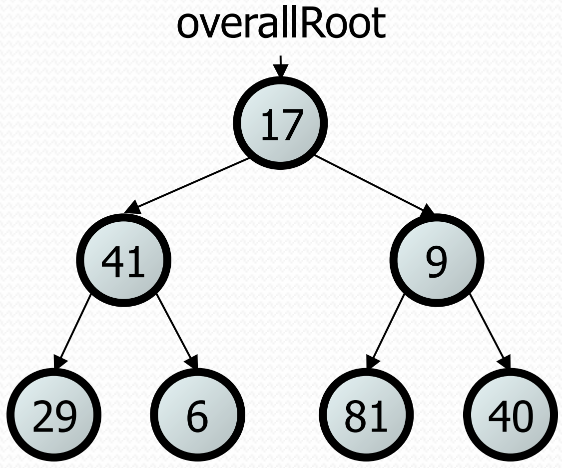
Exercise solution

```
// Prints the tree in a sideways indented format.
public void printSideways() {
    printSideways(overallRoot, "");
}

private void printSideways(IntTreeNode root,
                           String indent) {
    if (root != null) {
        printSideways(root.right, indent + "    ");
        System.out.println(indent + root.data);
        printSideways(root.left, indent + "    ");
    }
}
```

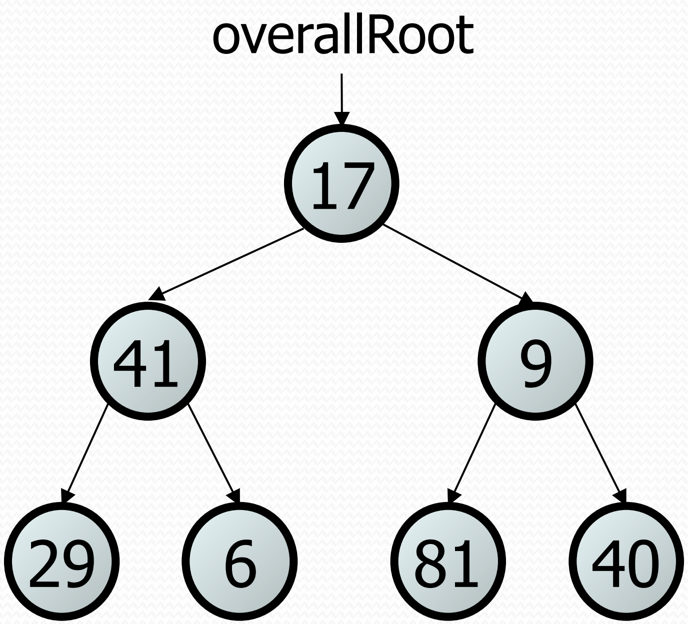
Traversals

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node



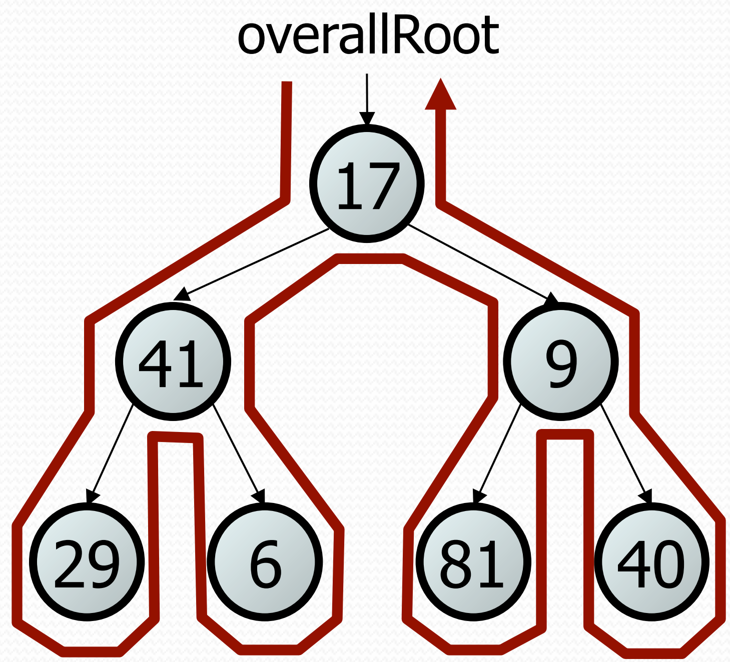
Traversal example



- pre-order: `17 41 29 6 9 81 40`
- in-order: `29 41 6 17 81 9 40`
- post-order: `29 6 41 81 40 9 17`

Traversal trick

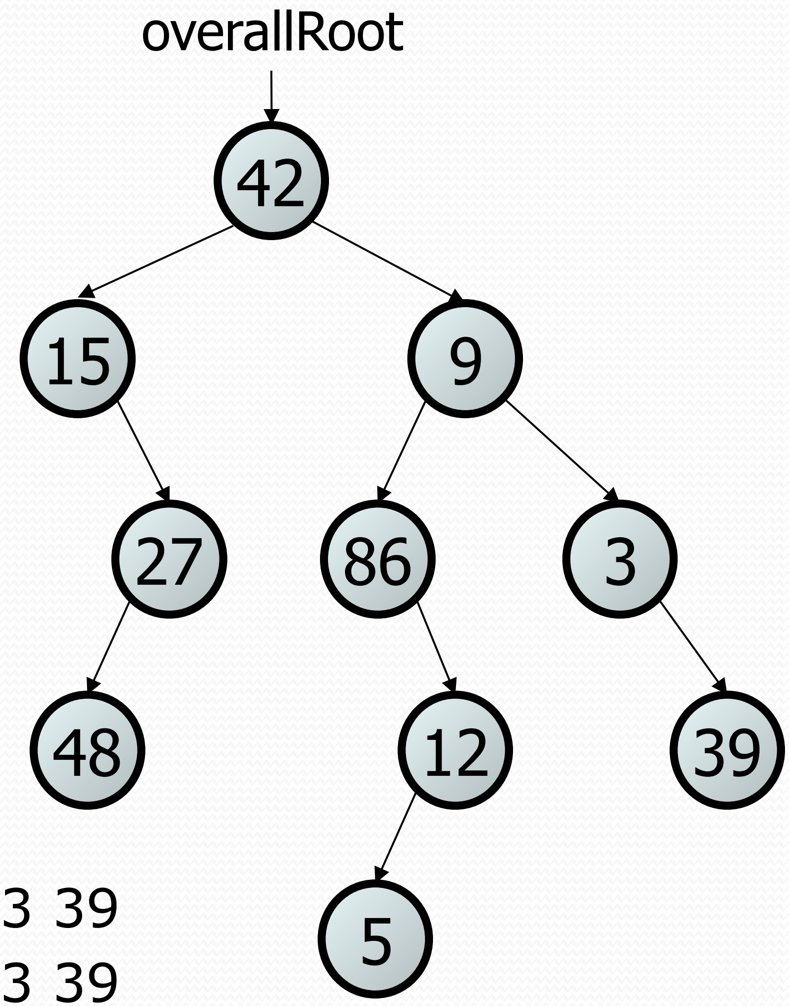
- To quickly generate a traversal:
  - Trace a path around the tree.
  - As you pass a node on the proper side, process it.
    - pre-order: left side
    - in-order: bottom
    - post-order: right side



- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

Exercise

- Give pre-, in-, and post-order traversals for the following tree:



  - pre: 42 15 27 48 9 86 12 5 3 39
  - in: 15 48 27 42 86 5 12 9 3 39
  - post: 48 27 15 5 12 86 39 3 42