Recursive Data Structures: Trees

University of Virginia
CS 2110
Prof. N. Basit

MSD Sections 6.1, 7.1-7.6; **Big Java** Ch 17

Announcements

- **Homework 5 – Concurrency and Recursion**
  - **Due:** by 11:30pm on Friday, April 17, 2020
  - Submit on Web-CAT
- **Homework 6** [**Last HW!**]
  - **Released:** Monday, April 20, 2020
  - **Due:** by 11:30pm on Tuesday, April 28, 2020
  - Submit on Web-CAT
- **Weekly Quiz**
  - **Released:** this afternoon (*Friday*)
  - **Due:** by 11:30pm on Sunday, as usual

Recursive Data Structures

- **Recursive Data Structure:** a data structure that contains references (or pointers) to instances of that same type
  - **Example: Linked Lists**

```
public class ListNode {
    Object nodeItem;
    ListNode next;
    ListNode previous;
    ...
}
```

Linked Lists [High-level / Brief]

Lists keep things *in order* - we have mainly discussed **ArrayLists.**

- **Arrays** keep thing 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

Linked Lists

```
public class LinkedList<T> {
        ListNode<T> head;
        ...
}
public class ListNode<T> {
        T nodeItem;
        ListNode<T> next;
        ...
}
```

Compared to

```
public class ArrayList<T> {
        T[] items;
        ...
}
```

Goals for this Unit

- Continue focus on data structures and algorithms
- Understand concepts of reference-based data structures (e.g. linked lists, binary trees)
  - Some implementation for binary trees
- Understand usefulness of trees and hierarchies as useful data models
  - Recursion used to define data organization
- Topics:
  - Trees
  - Heaps ("binary heaps")
  - BST
  - Tree Traversals

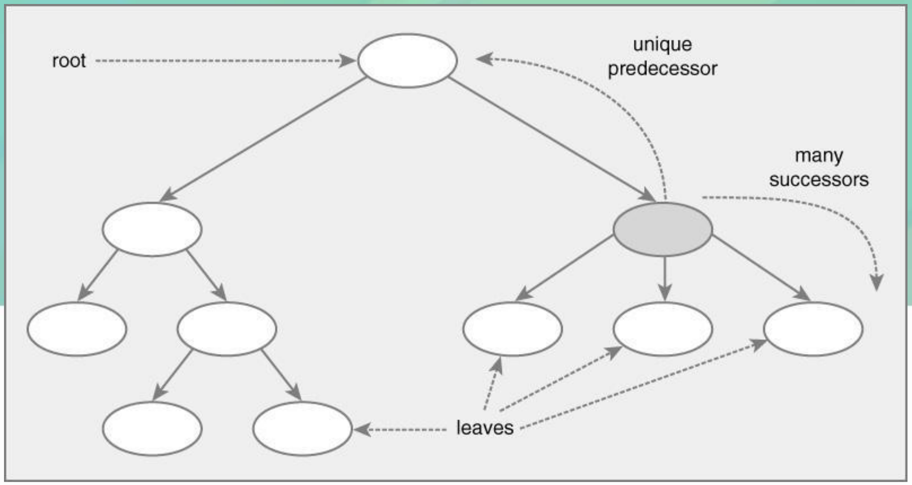
Why Does This Matter Now?

- 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
- <u>Principles</u>?
  - abstraction (hiding details)
  - delegation (helper classes, methods)

Trees

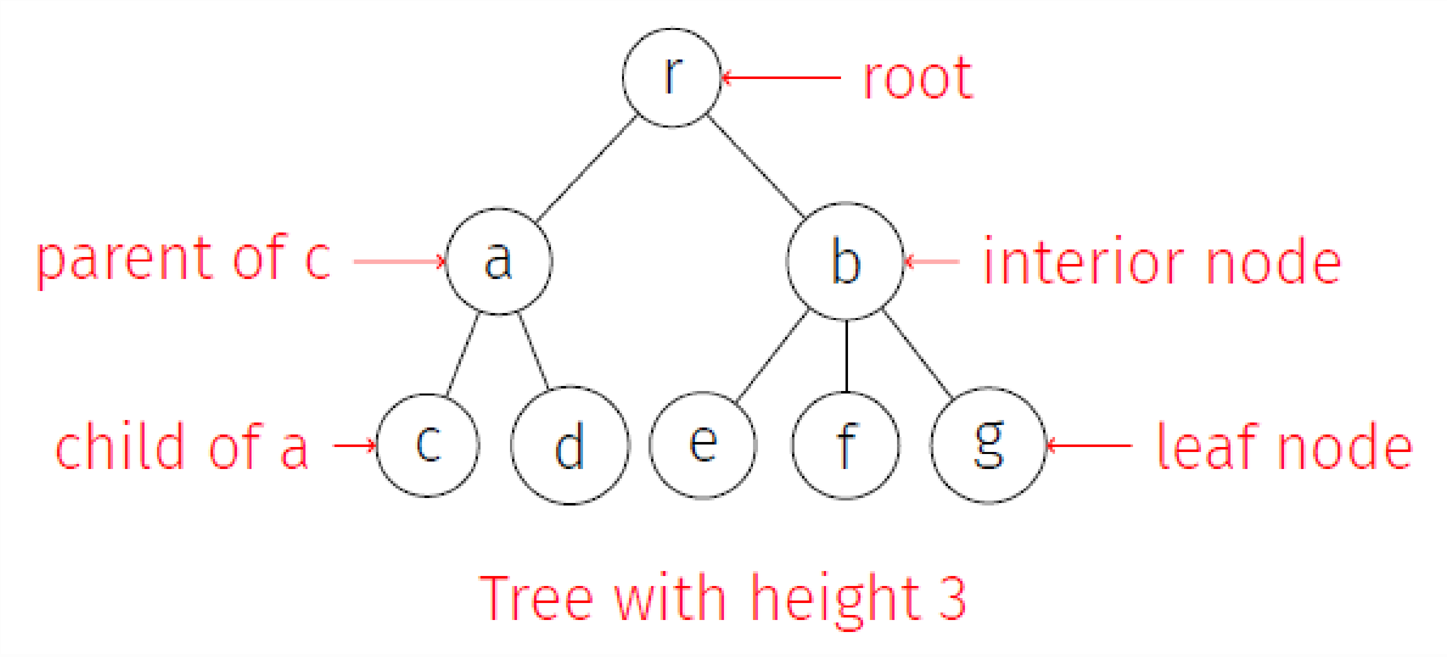
- Data types can be …
  - Simple or composite
- Data structures are composite data types …
  - Definition: a collection of elements that are some combination of primitive and other composite data types
- Tree Classification:
  - **Trees** are a
    - **composite**
    - **hierarchical** and
    - **graph-like** data structure
  - In Computer Science, trees grow down, not up!
    - Predecessors are up
    - Successors are down

Trees



- Trees are composed of:
  - **Nodes**
    - Elements in the data structure (hold data)
    - Only one parent (***unique predecessor***)
    - Zero, one, or more children (***successors***)
    - **LEAF** nodes: nodes without children (***terminal***)
    - **ROOT** node: **top** or start node; with no parent
    - **INTERNAL** node: notes with children (***non-terminal***)
    - Measure of **DEGREE**: how many children
  - **Edges**
    - Link parent node with children node (if applicable)
- The **HEIGHT** of a tree is the longest path (# nodes) from root to leaf

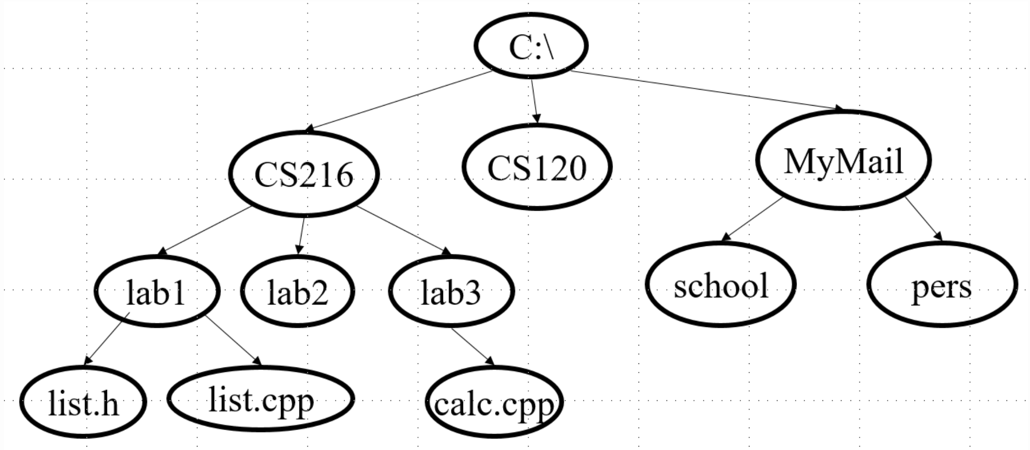
Trees



r
root
parent of c
a
b
interior node
child of a
c
d
e
f
g
leaf node
Tree with height 3

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
  - Family trees
  - The Linnaean taxonomy (kingdom, phylum, ..., species)
  - File systems (directory structures)
  - … others?



Tree Data Structures

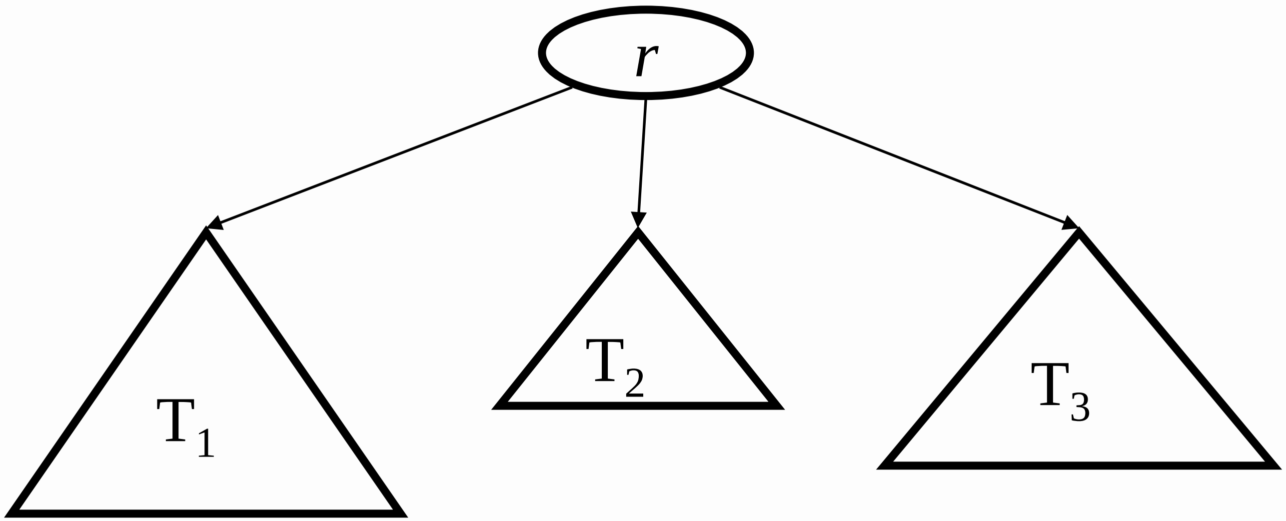
- Why are we talking about trees now?
  - Very useful in coding
  - An example of recursive data structures
  - Methods to act on trees are **recursive algorithms**

Tree Definitions and Terms

- **Binary tree:**
  - A tree in which each node has at most **two** 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

General Tree Depiction

- All (sub)trees are **recursively** defined as:
  - a root node with…
  - subtrees attached to it (e.g. T_1, T_2, and T_3 are attached to r)



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** (e.g. `hasParent()` or `hasChildren()`)
  - **TREE-WIDE** (e.g. `size()` or `height()`) – requires tree traversal

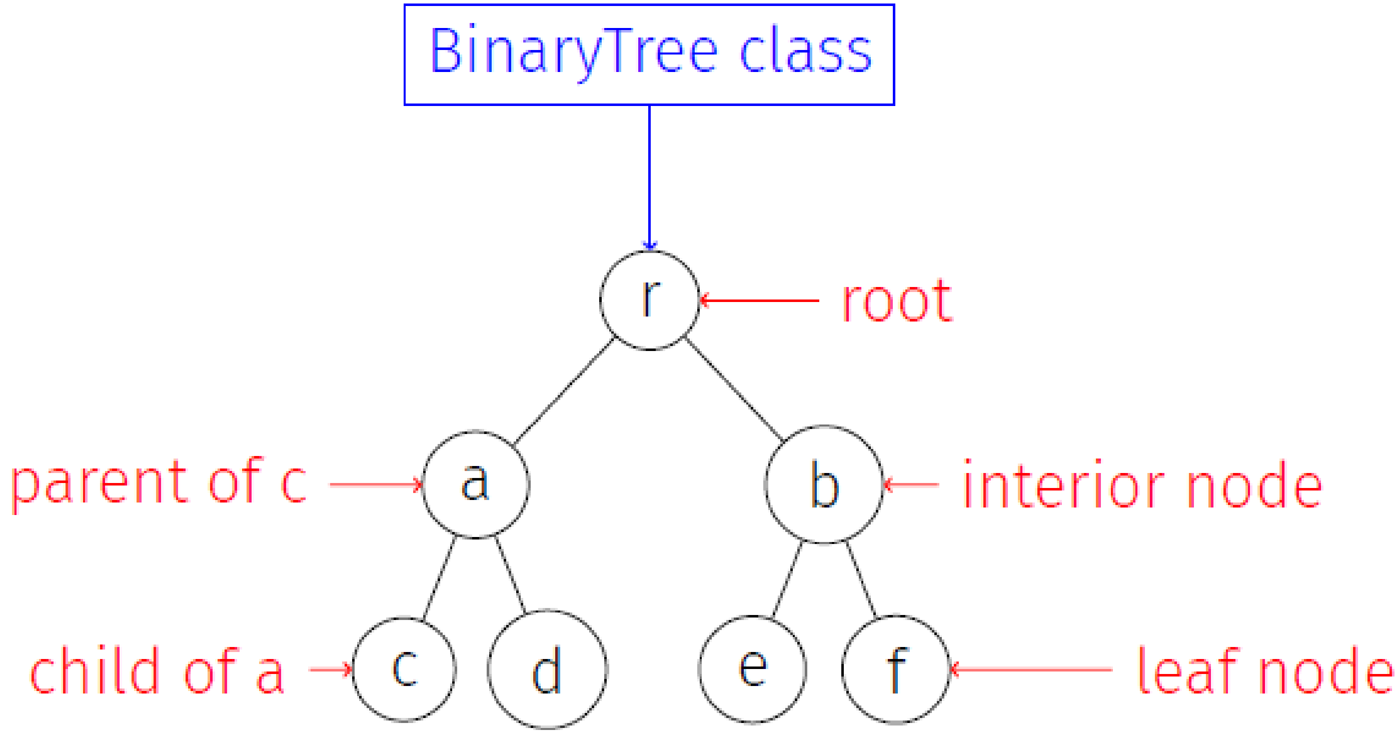
Classes for Binary Trees

We will define **TWO** classes (a simplified version of a binary tree)

- class **BinaryTree {..}** – defines the tree
  - reference pointer to the **root node**
  - methods: tree-level operations (like size())

- class **BiniaryTreeNode {..}** – defines a node in the tree!
  - **data**: an object (usually of some **Comparable** type)
  - **left**: references root of left-subtree (or null)
  - **right**: references root of right-subtree (or null)
  - **parent**: this node's parent node (optional)
    - Could this be null? When should it be?
  - **methods**: node-level operations

Binary Trees



Two-class Strategy for Recursive Data Structures

- This is a common design pattern: use one class for a Tree/List, another for Nodes

- **"Top" (tree) class**
  - Reference to "first" node
  - Methods and fields that apply to the entire data structure (i.e. the tree-object)

- **Node class**
  - Recursively defined: references to other node objects
  - Contains data stored at the node
  - Methods defined in this class are specific to this node or *recursive* (this node and its references)

Some Tree Methods

Discussion: How might we write the following methods?

- size()
- height()
- find() [*which assumes no order of nodes in the tree*]
- …

- **DEMO:** size() method: number of nodes in the tree
  - **Tree-wide size()** should check for empty tree (root is null), then ask root for its size
  - **Node-level size()** should count its children's sizes, add one for itself, and return the result (to be used by its parent)

Let's Go To Eclipse!

- ***Code on Trees:***
- *BinaryTree.java*
- *BinaryTreeNode.java*

- Note the use of generics! (See example method below)

```
/**
 * constructor
 * @param newRoot - root provided to construct the BinaryTree
 */
public BinaryTree(BinaryTreeNode<T> newRoot) {
    this.root = newRoot;
}
```

Size() method… [in Tree Class]

- **Tree-wide size()** should check for empty tree (root is null), then ask root for its size (call the node-version size() method on root)

```
public int size() {
    if (root == null) { // empty tree
        return 0; // size is zero
    }
    // otherwise, call size starting at
    // the ROOT of the tree
    return root.size(); // node level method
}
```

Size() method… [in Node Class]

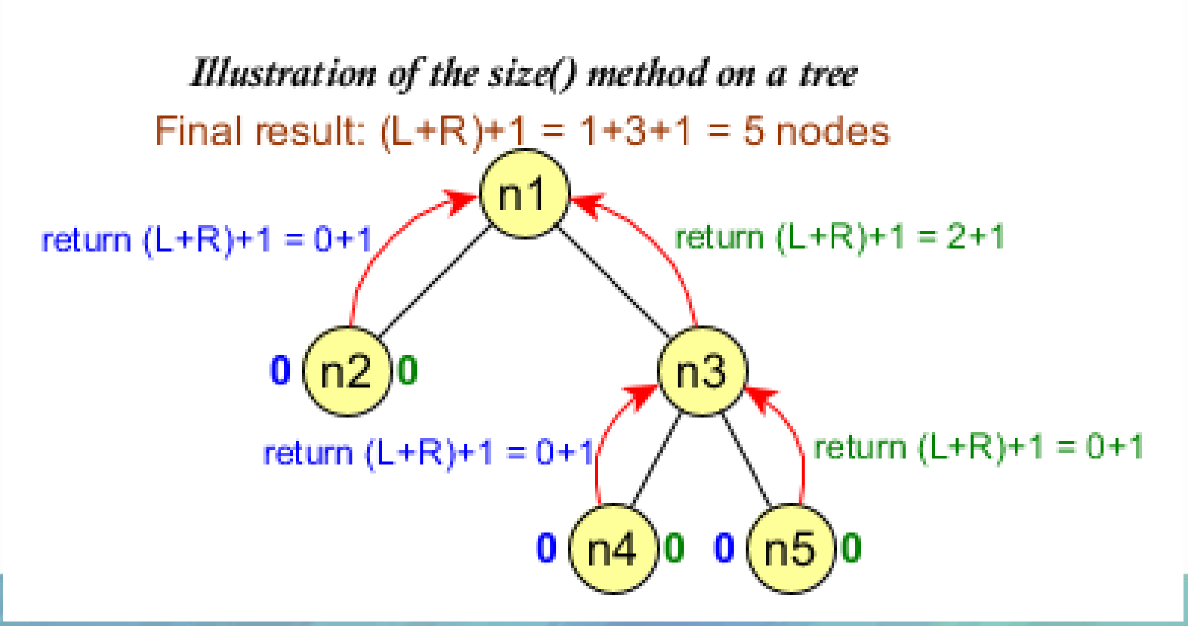
- **Node-level size()** should count its children's sizes, add one for itself, and return the result (to be used by its parent)

- Initialize size variable to 0 (variable to keep track of # nodes)
- The size of the tree rooted at **this node** is one more than the sum of the sizes of its children [size-of-left + size-of-right + 1]

- Check if current node has a **LEFT** child (not null)
  - If so, accumulate size to be **size + size of the left subtree** (that is, recursive call to size on the left node)
- Also check if current node has a **RIGHT** child (not null)
  - If so, accumulate size to be **size + size of the right subtree** (that is, recursive call to size on the right node)
- Finally, **return size + 1** *(adding 1 to account for the current node)*

public int size() method [in Node Class]

- Initialize size variable to 0 (variable to keep track of # nodes)
- Check if current node has a **LEFT** child (not null)
  - If so, size = **size + left.size()**
- Check if current node has a **RIGHT** child (not null)
  - If so, size = **size + right.size()**
- Finally, **return size + 1** *(adding 1 to account for the current node)*



Size() method… [in Node Class]

- **Node-level size()** should count its children's sizes, add one for itself, and return the result (to be used by its parent)

```
public int size() {
   int size = 0;
   if(left != null) // there is a left subtree
      size += left.size(); // recursively call size() on left
   if(right != null) // there is a right subtree
      size += right.size(); // recursively call size() on right
   size += 1; // add one to account for current (this) node
   return size; // return
}
```

Size() method… [in Node Class] - Alternative

- **Node-level size()** should count its children's sizes, and itself; then return the result (to be used by its parent)

```
public int size() {
    int size = 1; // set to one to account for current (this) node
    if(left != null) // there is a left subtree
        size += left.size(); // recursively call size() on left
    if(right != null) // there is a right subtree
        size += right.size(); // recursively call size() on right
    size += 1;
    return size; // return
}
```

Binary Trees

In-Class Activity-Trees (Day 1): Connecting Nodes

In-Class Activity: Binary Trees

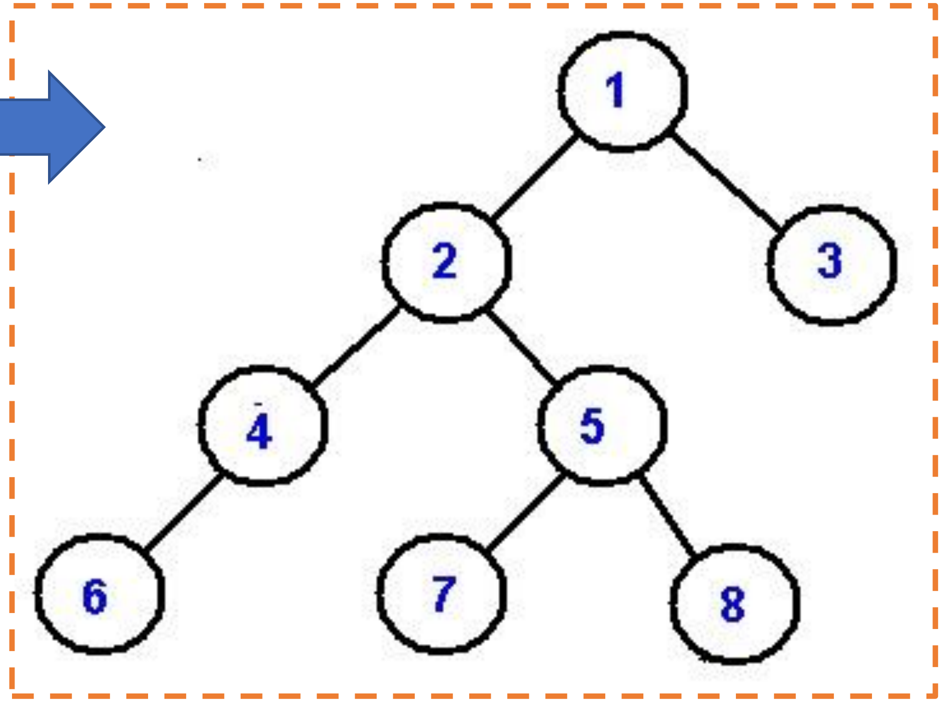
1. Download BinaryTree.java and BinaryTreeNode.java
2. In the main method of BinaryTreeNode.java, create the nodes 1 through 8
   - Use the **Integer** data type: **`BinaryTreeNode<Integer>`**
   - Create all of your BinaryTreeNodes first (`b1`→`b8`)

   E.g.: `BinaryTreeNode<Integer> b1 = new BinaryTreeNode<Integer>(1);`
   - Use `b1` as the **root**
3. Then create the connections to recreate the following tree (connect nodes in the same way) --- *see next page!*
   - Use **`setLeft()`** and **`setRight()`** methods to build tree.

   E.g.: `b1.setLeft(b2); //b2 is the left child of b1`

5. When finished, take the **root node** and call **`toString()`** to print out the result. If done correctly the **<u>output should be</u>**: (6)(4)(7)(8)(5)(2)(3)(1)
6. **<u>SUBMIT</u>**: your BinaryTreeNode.java file on Collab

In-Class Activity: Binary Trees



1. In the main method of BinaryTreeNode.java, create the nodes 1 through 8
   - **E.g.:** `BinaryTreeNode<Integer> b1 = new BinaryTreeNode<Integer>(1);`
2. Then create the connections to recreate the following tree (connect nodes in the same way.) Use `setLeft()` and `setRight()` methods to build tree.
   E.g.: `b1.setLeft(b2);`
5. When finished, take the **root node** and call `toString()` to print out the result.
   If done correctly the **output should be**: `(6)(4)(7)(8)(5)(2)(3)(1)`
6. **SUBMIT**: your BinaryTreeNode.java file on Collab