CompSci 201, L16: Binary Search Trees

Logistics, Coming up

- APT7 (sorting problems) due today Wednesday 10/26
- Midterm exam 2 this Wednesday 10/26
- Project 4: Autocomplete due next Monday 10/31

Public Service Announcement: Spring 2023 Registration

Registration Windows for Spring 2023

All students must have a student record free of administrative and financial holds to be able to register in DukeHub. Divinity School, Pratt School of Engineering (undergraduate), School of Nursing, and Trinity College of Arts & Sciences all require students to meet with an advisor and be marked eligible to enroll for the term prior to registration. All other students are strongly encouraged to consult an advisor before registering for classes in DukeHub.

Graduate and Professional Students, *Wednesday, November 2, 7:00 AM*

Seniors, *Thursday, November 3, 7:00 AM*

Juniors - (last two digits of the Student ID)
50-99, *Friday, November 4, 7:00 AM*
00-49, *Monday, November 7, 7:00 AM*

Sophomores - (last two digits of the Student ID)
50-99, *Tuesday, November 8, 7:00 AM*
00-49, *Thursday, November 10, 7:00 AM*

First Year - (last two digits of the Student ID)
50-99, *Friday, November 11, 7:00 AM*
00-49, *Monday, November 14, 7:00 AM*

registrar.duke.edu/registration/about-registration

CS Advising and Book bagging

- Considering a major/idm/minor/concentration? There are many pathways!

- Computer Science Majors
  - B.S. or B.A., same core CS requirements, different math and electives requirements.
- Interdepartmental Majors: 7 from CS, 7 from another
  - CS+Stats Data Science, CS+Math Data Science, CS+Linguistics, CS+VSM Computational Media.
- Computer Science Minor: 5 CS courses

Current Major Requirements in CS

210 – software oriented
250 – hardware oriented

CS 210D Intro to Computer Systems OR CS 250D Computer Architecture

Systems, choose >=1 course

- CS 310 Operating Systems
- CS 316 Databases
- CS 350 Digital Systems
- CS 351 Security
- CS 356 Networks

CS 201 Data Structures & Algorithms

Plus electives and math/stats classes

CS 230 Discrete Math for Computer Science

CS 330 Design and Analysis of Algorithms

Can substitute with extra math/stats classes

Common post-201 CS Courses available in Fall 2022

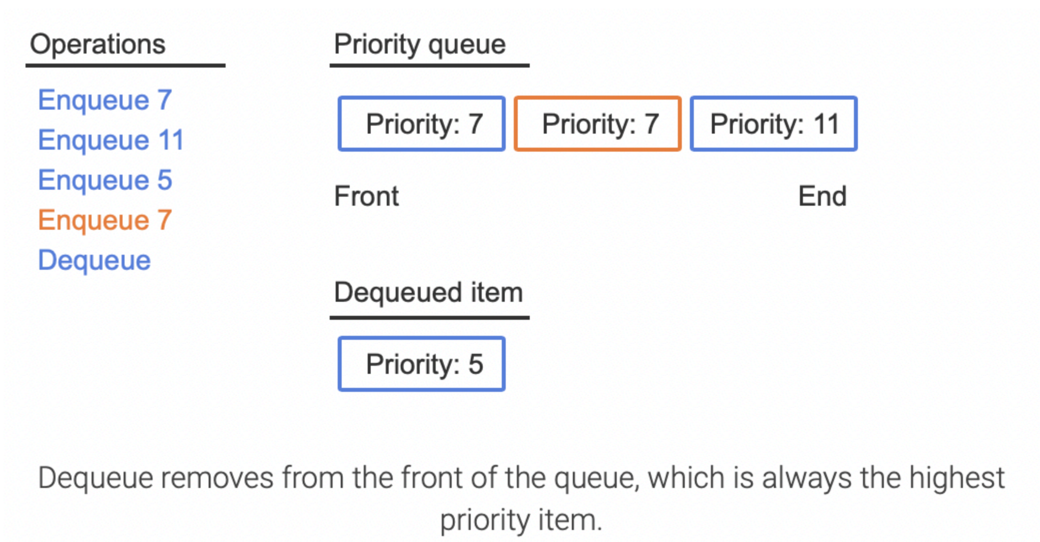
Next Required CS major courses:

- CS 210D Intro to Computer Systems OR CS 250D Computer Architecture.
- CS 230 Discrete Math for Computer Science.

Some electives in Sp 23 with no other prereqs (not exhaustive of the options!)

- CS 216 Everything Data
- CS 240 Race Gender Class & Computing
- CS 290-01 IOS Mobile App Development

Priority Queue in the Abstract



Queue sorted by priority instead of insertion order.

Zybook

java.util.PriorityQueue Class

- Kept in sorted order, smallest out first
  - Objects must be Comparable OR provide Comparator to priority queue

```
PriorityQueue<String> pq = new PriorityQueue<>();
pq.add("is");
pq.add("Compsci 201");
pq.add("wonderful");
while (! pq.isEmpty()) {
    System.out.println(pq.remove());
}
```

```
Compsci 201
is
wonderful
```

```
PriorityQueue<String> pq = new PriorityQueue<>(
        Comparator.comparing(String::length));
pq.add("is");
pq.add("Compsci 201");
pq.add("wonderful");
while (! pq.isEmpty()) {
    System.out.println(pq.remove());
}
```

```
is
wonderful
Compsci 201
```

WOTO

Go to duke.is/w7jv8

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

Inefficient DIY Priority Queue with Plain Old ListNodes

Design for a lazy priority queue:

- Invariant: Keep the list sorted
- Always remove first (least) node and update myFirst
- To add, need to search for correct in-order position

Class

```java
public class lazyPriorityQueue {
    private class ListNode {...}
    private ListNode myFirst;
    private int mySize;

    lazyPriorityQueue() {
        myFirst = null;
        mySize = 0;
    }
```

Complexity of Inefficient Lazy DIY Priority Queue

- Peek: O(1) – Just get value of first node
- Remove: O(1) – Just remove first node
- Add: O(N) – Need to search linked list

Alternative, if we had left the list unsorted…

- Peek/Remove: O(N) – Have to search
- Add: O(1) – Just add to front of list

Tradeoffs, Heaps, and Trees

- Fast add and remove?
- Binary Heap: Implements a priority queue with:
  - Peek: O(1)
  - Remove: O(log(N))
  - Add: O(log(N))
- `java.util` `PriorityQueue` is implemented as a binary heap

N	Log_2(N)
100	6.6
200	7.6
400	8.6
800	9.6
1600	10.6
3200	11.6
6400	12.6
12800	13.6
25600	14.6
51200	15.6

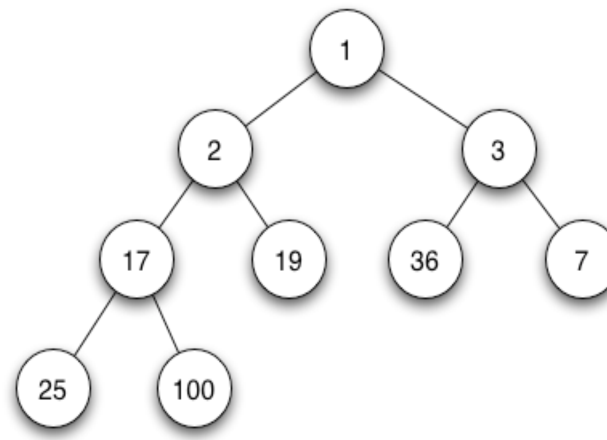
Binary Heap at a high level

Sorted list of nodes → ordered binary tree of nodes

- Maintain the **heap property** *that every node is less than or equal to its successors*, and
- The **shape property** *that the tree is full except for the rightmost positions on the last level.*

Operations:

- Peek: Return value of root node
- Remove: Remove root node and fix tree to reestablish properties.
- Add: Insert at first open position, fix to reestablish properties.



By Vikingstad at English Wikipedia - Transferred from en.wikipedia to Commons by LeaW., Public Domain, https://commons.wikimedia.org/w/index.php?curid=3504273

Binary Trees

Why another data structure? Trees!

- ArrayList: Fast, not very dynamic
  - O(1) get but O(N) add/remove (except at end)
- LinkedList: Dynamic, not very fast
  - O(N) get, O(1) add/remove (once you get there)

- Binary Search Tree: Fast AND dynamic
  - O(log(N)) search AND O(log(N)) add/remove.
  - ASSUMING the tree is ~balanced

Comparing TreeSet/Map with HashSet/Map

TreeSet/Map

- O(log(N)) add, contains, put, get *not amortized*.
- Stored in sorted order
- Can get range of values in sorted order efficiently

HashSet/Map

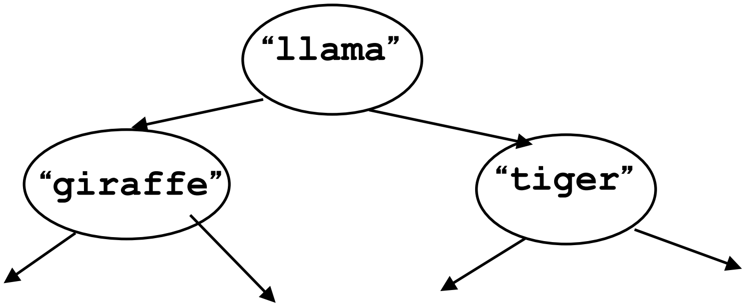
- O(1) add, contains, put, get, *amortized*.
- Unordered data structures
- Cannot get range efficiently, stored unordered

Binary Tree Nodes

Nodes for trees

```
public class TreeNode {
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
             TreeNode llink, TreeNode rlink){
        info = s;
        left = llink;
        right = rlink;
    }
}
```

Like LinkedList but each node has 2 pointers instead of 1



Nodes in the wild with Java 8

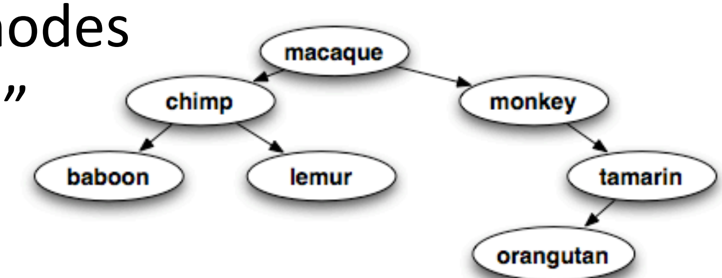
http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/8ed8e2b4b90e/src/share/classes/java/util/TreeMap.java

- In TreeMap the root has this type
  - maintains left, right, and parent pointers
    - Similar to: info and next, or info, left, right

```java
    static final class Entry<K,V> implements Map.Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left = null;
        Entry<K,V> right = null;
        Entry<K,V> parent;
```

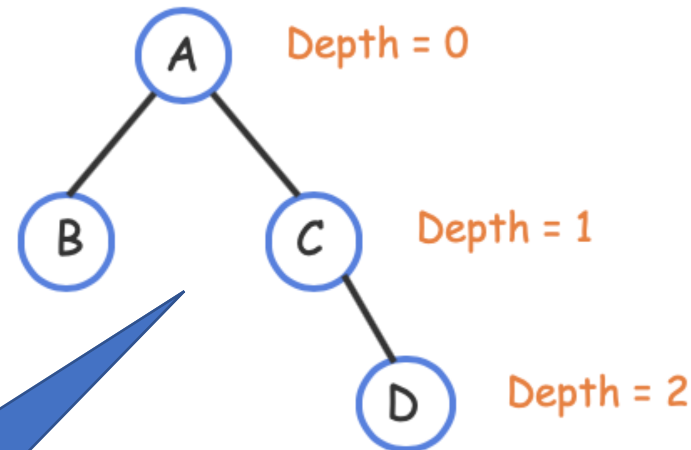
Tree terminology

- *Root*: "top node", has no parent, node you pass for the whole tree
  - "macaque". Subtrees also have a root: chimp, …
- *Leaf*: “bottom” nodes, have no children / both null
  - "baboon", "lemur", "organutan"
- *Path*: sequence of parent-child nodes
  - "macaque", "chimp", "lemur”
- *Subtree*: nodes at and beneath
  - ”chimp”, “baboon”, “lemur”



More Tree terminology

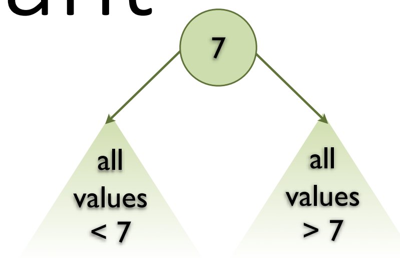
The **depth** of a node is the number of edges from the root to the node.



The **height** of a tree is the maximum depth of any node.

Height is max(0,1,2) = 2

Binary Search Tree Invariant



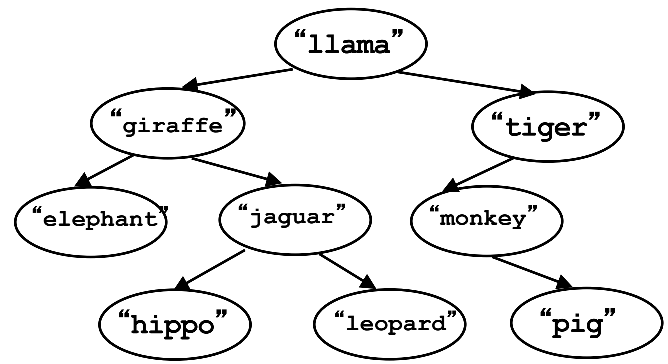
A binary tree is a binary **search** tree if *for every node*:

- Left subtree values are all less than the node's value

AND

- Right subtree values are all greater than the node's value

According to some ordering (comparable or comparator)



Enables efficient search, similar to binary search!

WOTO

Go to duke.is/8hjzt

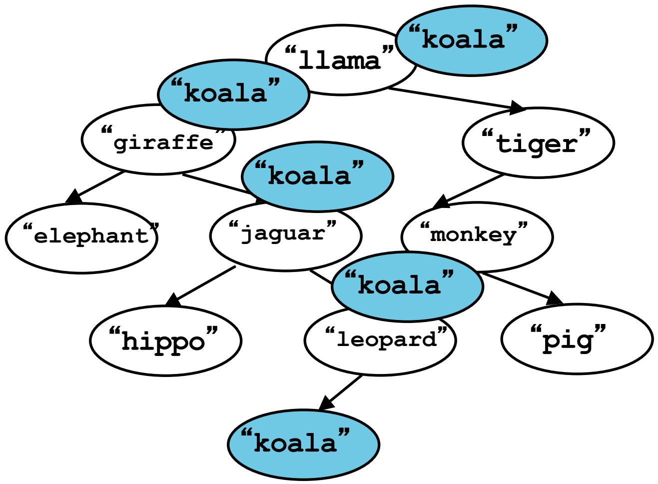
Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

Efficient Search in Binary Serach Tree

- Code for search
  - Insertion is very similar
  - **`target.compareTo(…)`**




```java
public boolean contains(TreeNode tree, String target) {
    if (tree == null) return false;
    int result = target.compareTo(tree.info);
    if (result == 0) return true;
    if (result < 0) return contains(tree.left,target);
    return contains(tree.right, target);
}
```

Recursion strikes again!

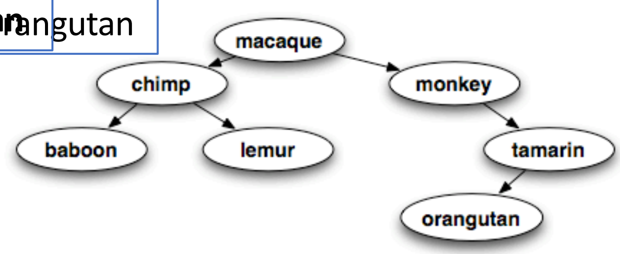
inOrder Traversal

- Illustrate with inOrder traversal and print
  - Search tree values printed in order
  - Could "visit" rather than print, every value

baboon, chimp, lemur, macaque, monkey, tamarin, orangutan

baboon, chimp, lemur
macaque
monkey, tamarin, orangutan

```java
public void inOrder(TreeNode root) {
    if (root != null) {
        inOrder(root.left);
        System.out.println(root.info);
        inOrder(root.right);
    }
}
```



Helper method to return List

```
public ArrayList<String> visit(TreeNode root) {
    ArrayList<String> list = new ArrayList<>();
    doInOrder(root,list);
    return list;
}

private void doInOrder(TreeNode root, ArrayList<String> list) {
    if (root!= null) {
        doInOrder(root.left,list);
        list.add(root.info);
        doInOrder(root.right,list);
    }
}
```

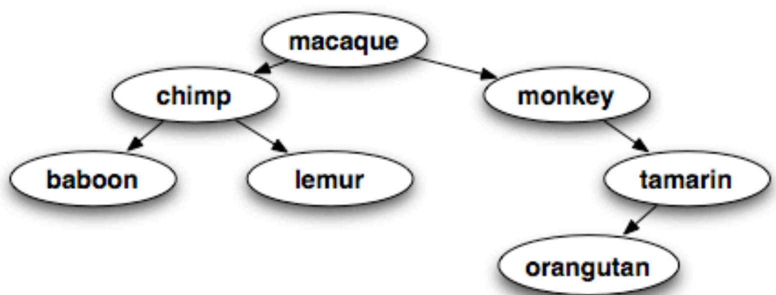
- In order traversal → list?
- Create list, call helper, return list
- values in returned list in order

Three ways to recursively traverse a tree

- Difference is in where the non-recursive part is

inorder	preorder	psotorder

void inOrder(TreeNode t) {
  if (t != null) {
    inOrder(t.left);
    System.out.println(t.info);
    inOrder(t.right);
  }
}
	void preOrder(TreeNode t) {
	  if (t != null) {
	    System.out.println(t.info);
	    preOrder(t.left);
	    preOrder(t.right);
	  }
	}
		void postOrder(TreeNode t) {
		  if (t != null) {
		    postOrder(t.left);
		    postOrder(t.right);
		    System.out.println(t.info);
		  }
		}

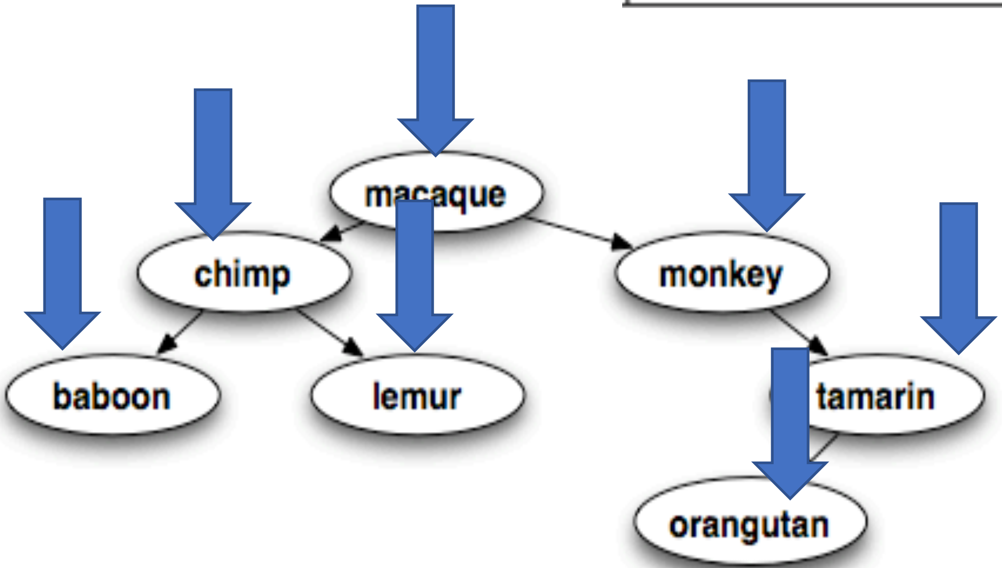


preOrder Traversal

- macaque
- chimp
- baboon
- lemur
- monkey
- tamarin
- orangutan

preorder

```
void preOrder(TreeNode t) {
  if (t != null) {
    System.out.println(t.info);
    preOrder(t.left);
    preOrder(t.right);
  }
}
```

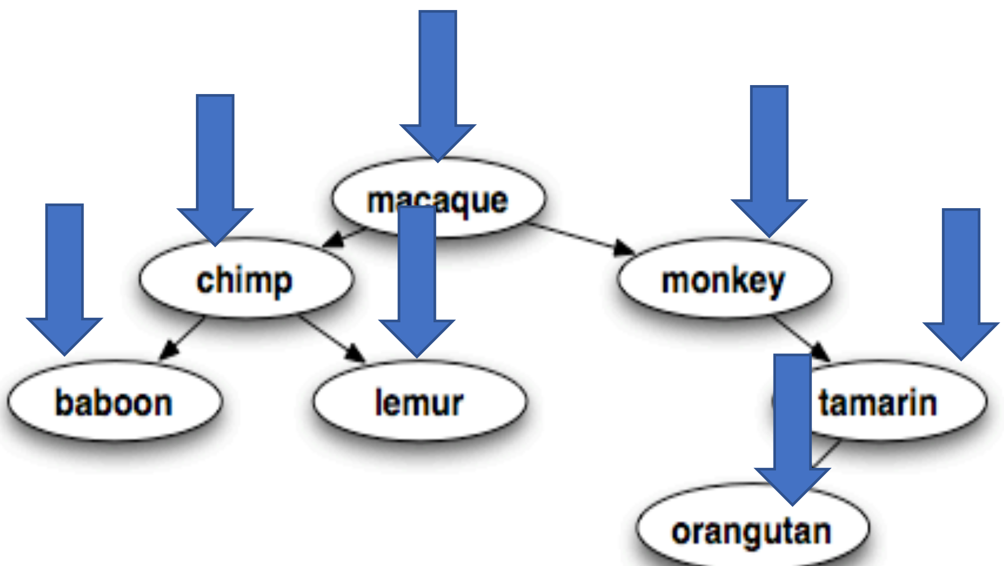


postOrder Traversal

- baboon
- lemur
- chimp
- orangutan
- tamarin
- monkey
- macaque

psotorder

```
void postOrder(TreeNode t) {
  if (t != null) {
    postOrder(t.left);
    postOrder(t.right);
    System.out.println(t.info);
  }
}
```

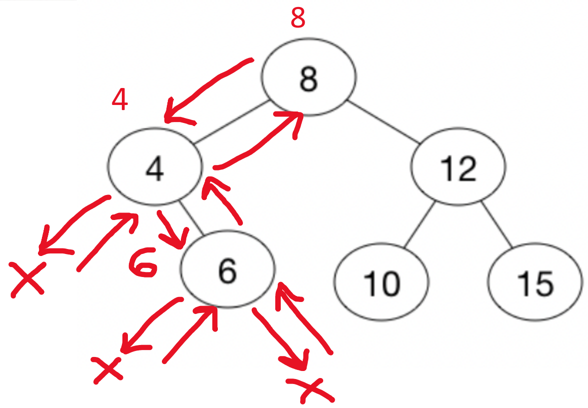


TreeCount APT

Problem Statement

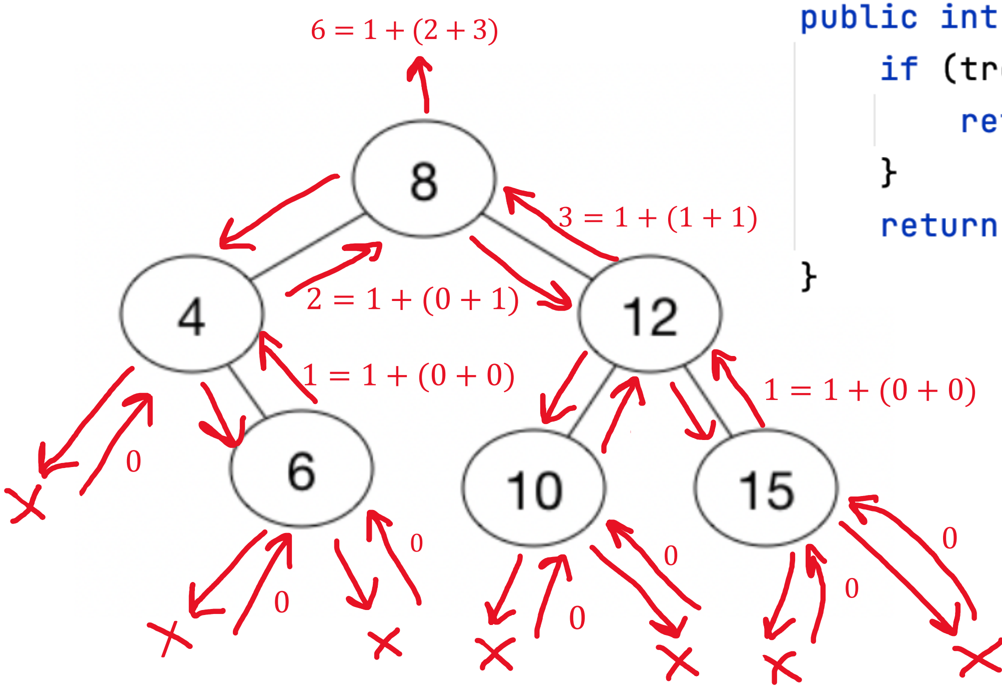
Write a method that returns the number of nodes of a binary tree. The TreeNode class will be accessible when your method is tested.

```
public class TreeCount {
    public int count(TreeNode tree) {
        // replace with working code
        return 0;
    }
}
```



is characterized by the pre-order string **8, 4, x, 6, x, x, 12, 10, x, x, 15, x, x**

Solving TreeCount in Picture & Code




```java
public int count(TreeNode tree) {
    if (tree == null) {
        return 0;
    }
    return 1 + count(tree.left) + count(tree.right);
}
```

FAQ: Can I make a tree?

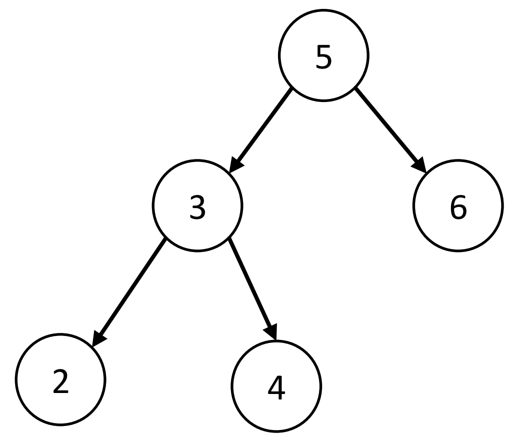
```java
public class TreeNode {
    int info;
    TreeNode left;
    TreeNode right;
    TreeNode(int x){
        info = x;
    }
    TreeNode(int x, TreeNode lNode, TreeNode rNode){
        info = x;
        left = lNode;
        right = rNode;
    }
```

Just call the TreeNode constructor for each new node and connect them.

```java
TreeNode root = new TreeNode( x: 5);
root.left = new TreeNode( x: 3);
root.right = new TreeNode( x: 6);
root.left.left = new TreeNode( x: 2);
root.left.right = new TreeNode( x: 4);
```

More terse version

```java
TreeNode myTree = new TreeNode( x: 5,
          new TreeNode( x: 3,
                    new TreeNode( x: 2),
                    new TreeNode( x: 4)),
          new TreeNode( x: 6));
```



Tree Recursion tips / common mistakes

1. Draw it out! Trace your code on small examples.
2. Return type of the method. Do you need a helper method?
3. Base case first, otherwise infinite recursion / null pointer exception.
4. If you make a recursive call, make sure to use what it returns.

Complexity of tree traversal

- Intuition: visit every node once and print it
  - If there are N nodes, should be O(N)
  - But what about recursive calls?

- More generally/formally:
  - We create a recurrence relation (an equation)
  - Solving the equation yields runtime

Developing runtime recurrence relation

- ***T(n)*** time **`inOrder(root)`** with ***n*** nodes
  - T(n) = T(n/2) + O(1) + T(n/2) = O(n)

```java
public void inOrder(TreeNode root) {
    if (root != null) {
        inOrder(root.left);
        System.out.println(root.info);
        inOrder(root.right);
    }
}
```

- Why T(n/2)?

Assumes the tree is *balanced:* About the same number of nodes in the left subtree as the right.

If Tree is not balanced?

- If every node has a right child but no left…
  - T(n) = T(0) + O(1) + T(n-1) = O(n)
- If not

Balanced

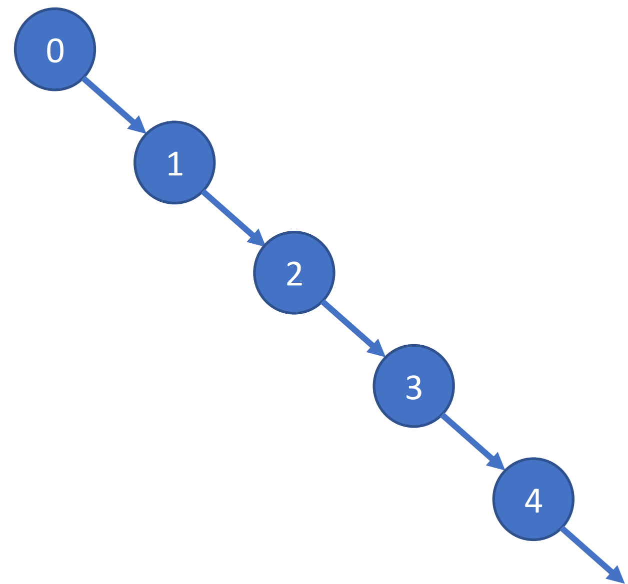
- But search?
- Insert?

```java
public void inOrder(TreeNode root) {
    if (root != null) {
        inOrder(root.left);
        System.out.println(root.info);
        inOrder(root.right);
    }
}
```

Why would a tree not be balanced?

Worse case:

- What if we insert sorted data?




```
For(int i=0; i<n; i++) {
    myTree.insert(i);
}
```

- Average case height O(log(n)) for random-ish order
- AVL trees, red-black trees (optional Zybook chapter) can dynamically ensure good balance.