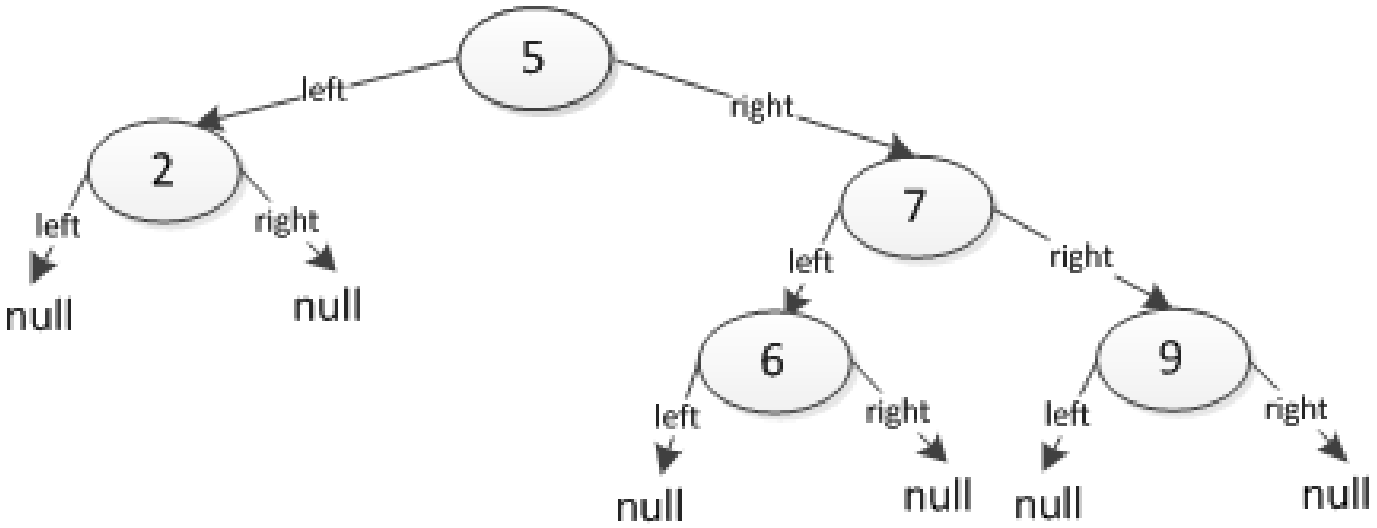
Recitation 12: Trees



Above is my solution to this week's prep.

Please snarf the code for today's class

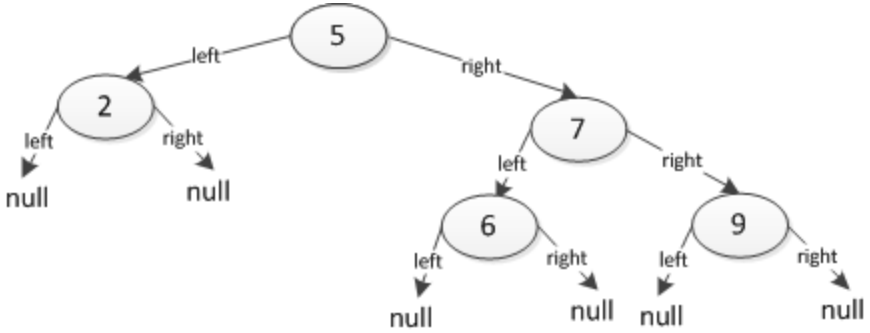
(it may be out of order)

Please also grab a handout.

What We Will Do In Class Today

1. You will learn the definition of a binary tree, and many related words (e.g. “root”, “leaf”, “binary search tree")
2. You will write some recursive code to manipulate binary trees, and you will be really impressed how easy and fun it is. We may high-five each other at the end. It will be that cool.
3. You will be able to articulate what makes binary search trees are so powerfully efficient – including understanding the runtime of the mysterious TreeSet

Binary Tree: In Diagram and In Code



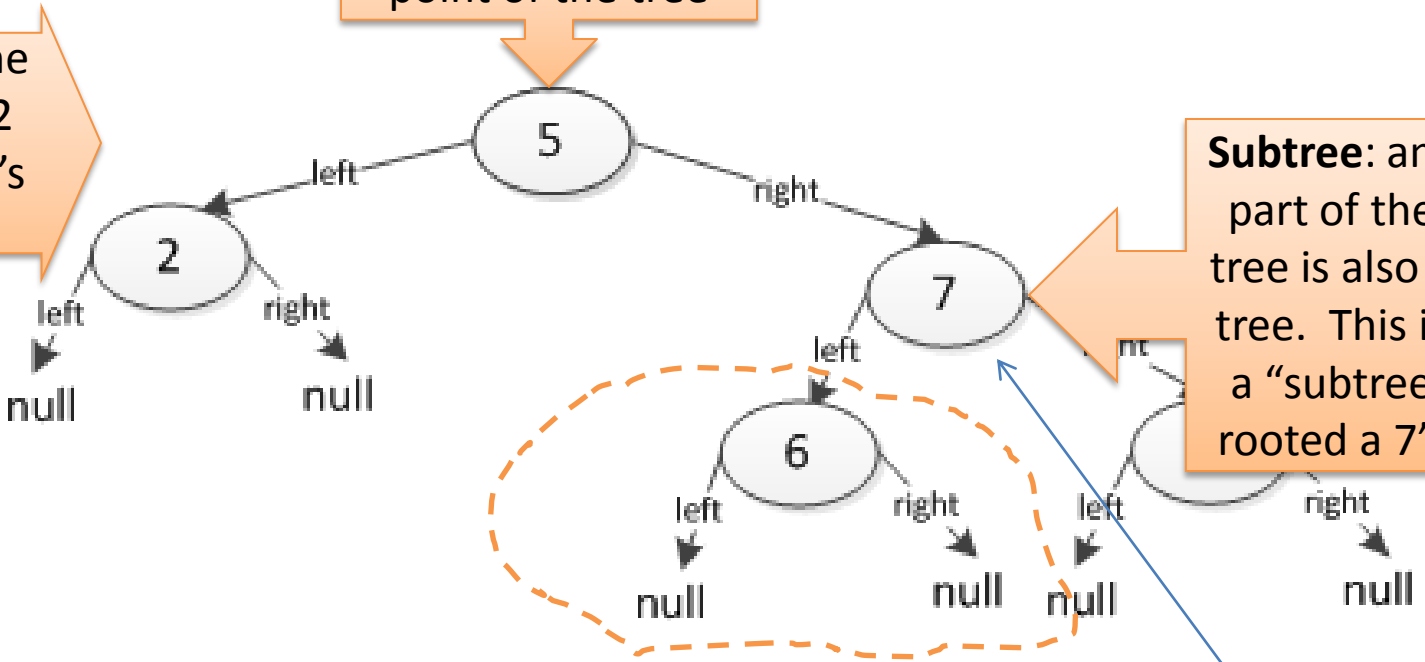

```
public class MikesIntBinaryTree {

    MikesIntTreeNode root = null;

    public class MikesIntTreeNode {
        public int value;
        public MikesIntTreeNode left; //holds smaller tree nodes
        public MikesIntTreeNode right; //holds larger tree nodes
    }

    //many handy functions like add, remove, and find go here
}
```

Binary Tree: Some terms to memorize



Root: the starting point of the tree

We'll often call 5 the "**parent of**" node 2 and node 2 node 5's "**left child**"

Subtree: any part of the tree is also a tree. This is a "subtree rooted a 7".

Leaf: a node that has no "child nodes"

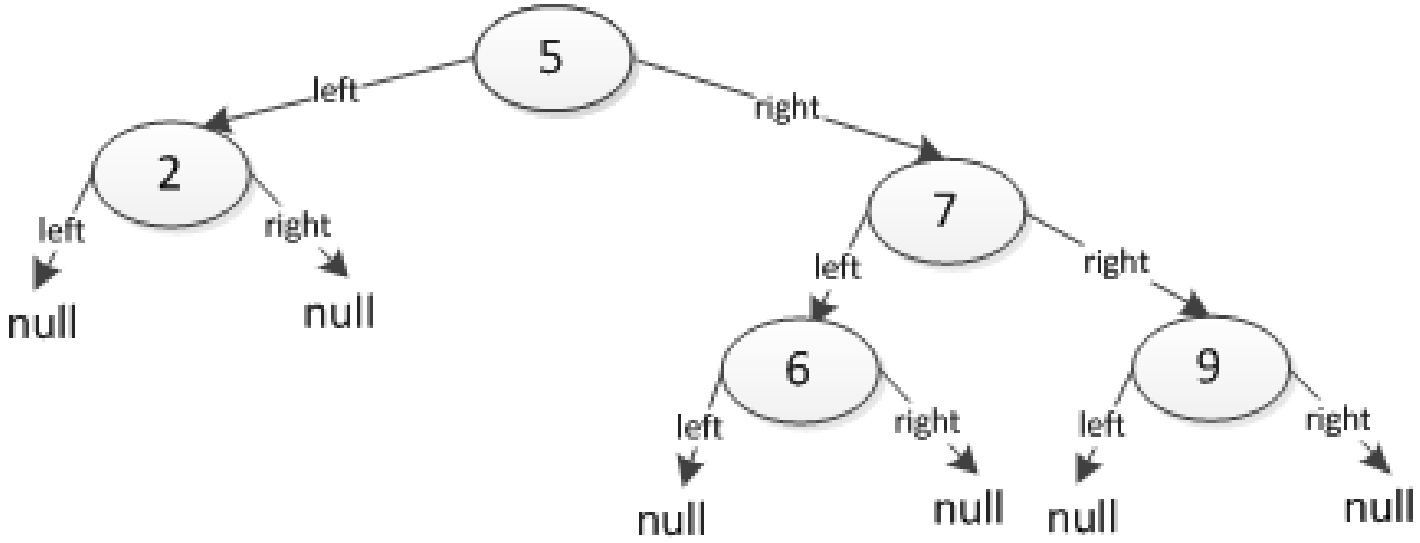
Internal node: a node that has 1 or 2 childen

To ponder:

What's so "binary" about this tree anyway?

Could there be loops? (like could the left child of node 6 point back at the root?)

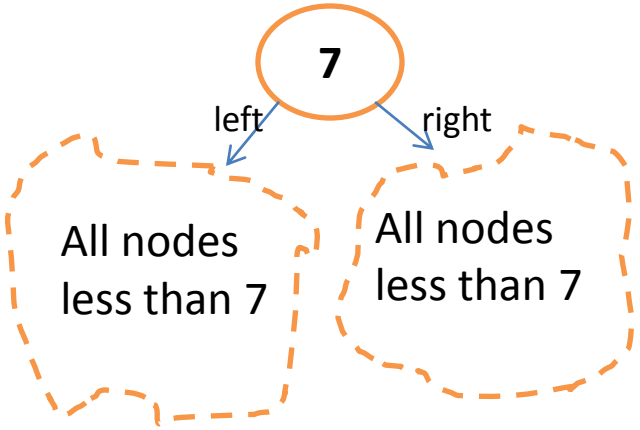
Yet more terms to memorize



- **Depth** is the distance of a node from the root. *Root is depth 1 (please note that I screwed this up on your handout).* So node 2 is depth 2, node 6 is depth 3.
- **Height** is the maximum depth of the tree

Binary Search Tree

In a “binary search tree” each node has a value. Nodes that have a value less than the node are in the *left subtree*. Nodes that have a value greater than the node are in the *right subtree*.



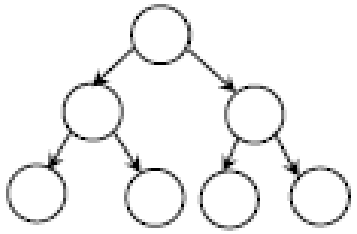
Height Balanced

A tree is **height-balanced** if

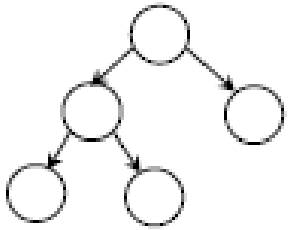
- left and right subtrees are both height balanced
- The heights of left and right subtrees do no differ by more than 1

Informally, we can see if a tree is height-balanced, it has nearly equal number of nodes on each side. This will turn out to matter hugely when we start talking about efficiency.

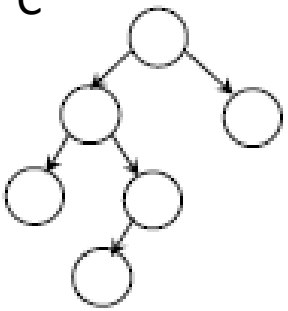
A



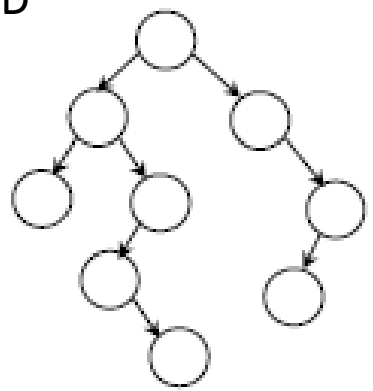
B



C

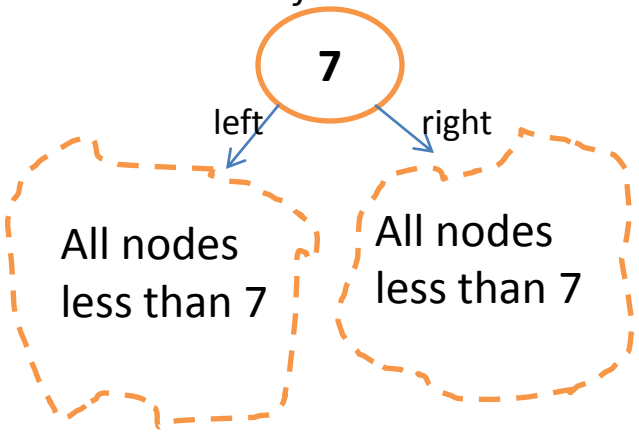


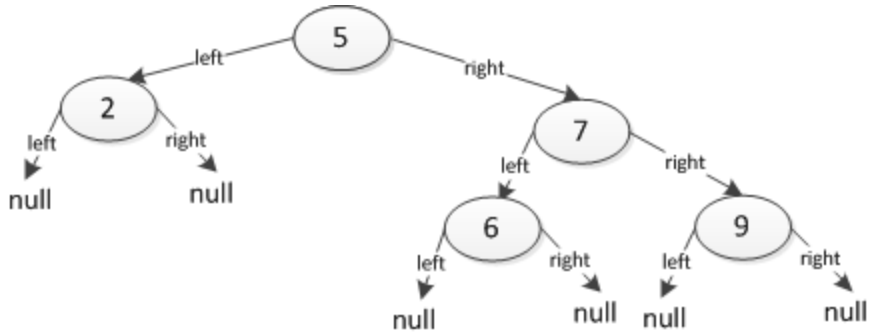
D



Binary Search Tree

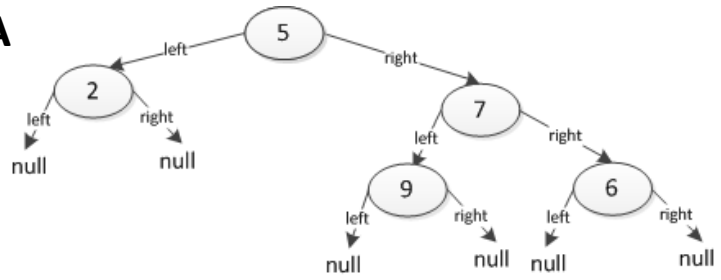
In a “binary search tree” each node has a value. Nodes that have a value less than the node are in the *left subtree*. Nodes that have a value greater than the node are in the *right subtree*.



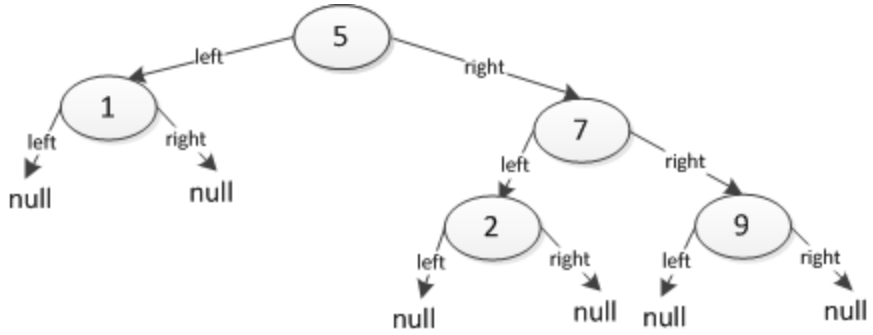


Are these binary search trees?

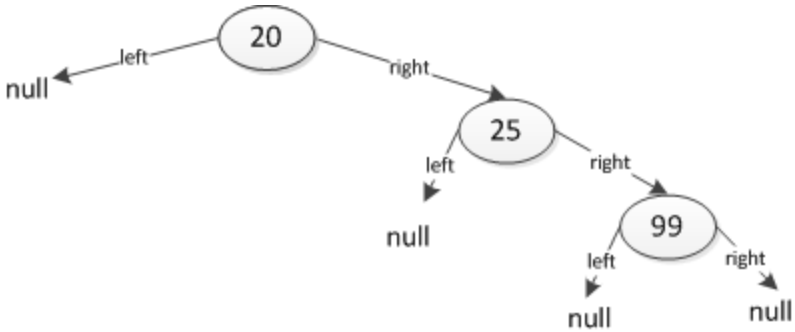
A



B



C



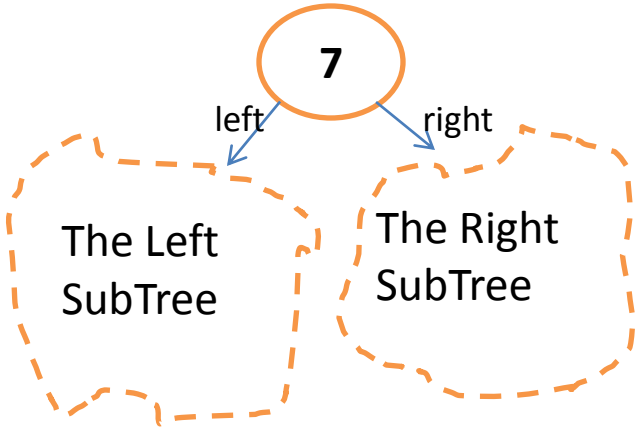
What We Will Do In Class Today

1. You know the meaning of : root, leaf, binary search tree, parent/child, subtree, height, depth, height-balanced

Coming next:

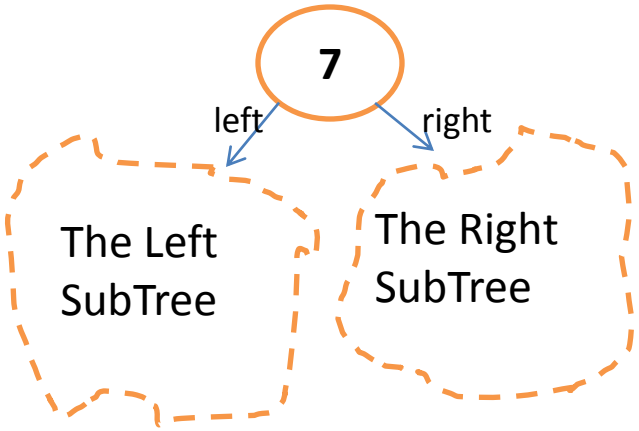
1. You will write some recursive code to manipulate binary trees, and you will be really impressed how easy and fun it is. We may high-five each other at the end. It will be that cool.
2. You will be able to articulate what makes binary search trees are so powerfully efficient – including understanding the runtime of the mysterious TreeSet

Recursion and Trees: Prepare Your World for Rocking



Recursion and Trees: How Your Code Will Look

```
public int computeTreeThing(TreeNode current) {
if (we are at the base case) {
  return obviousValue;
} else {
   int lResult = computeTreeThing(current.left);
   int rResult = computeTreeThing(current.right);
   int result = //combine those values;
   return result;
}
```



Coding Exercise

```
// my sample template
public int computeTreeThing(TreeNode current) {
if (we are at the base case) {
  return obviousValue;
} else {
   int lResult = computeTreeThing(current.left);
   int rResult = computeTreeThing(current.right);
   int result = //combine those values;
   return result;
}
```

Do as many of countNodes, containsNode, findMax as you can.
If you get stuck on countNodes (I've provided most of the code), definitely raise your hand.
If you finish earily, go ahead and modify your functions to work assuming a Binary Search Tree.

Once you're finished…go ahead and submit your code through Ambient.

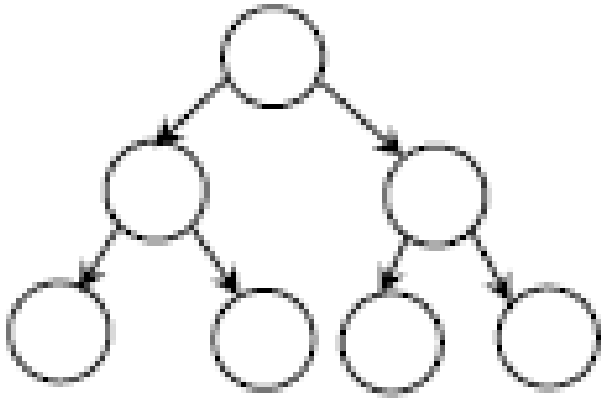
Where We Are

1. You know the meaning of : root, leaf, binary search tree, parent/child, subtree height, depth, height-balanced.
2. You wrote some slick recursive tree code.

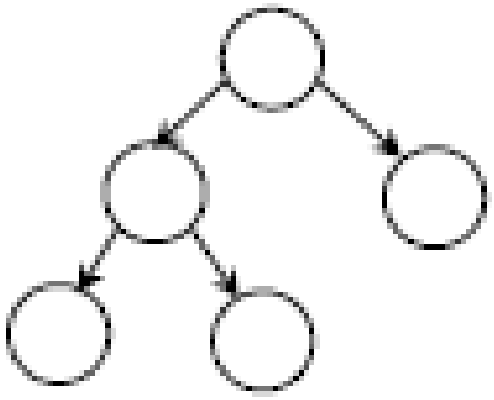
Coming next:

1. You will be able to articulate what makes binary search trees are so powerfully efficient – including understanding the runtime of the mysterious TreeSet

Given a tree that's height-balanced, what is its *height*?



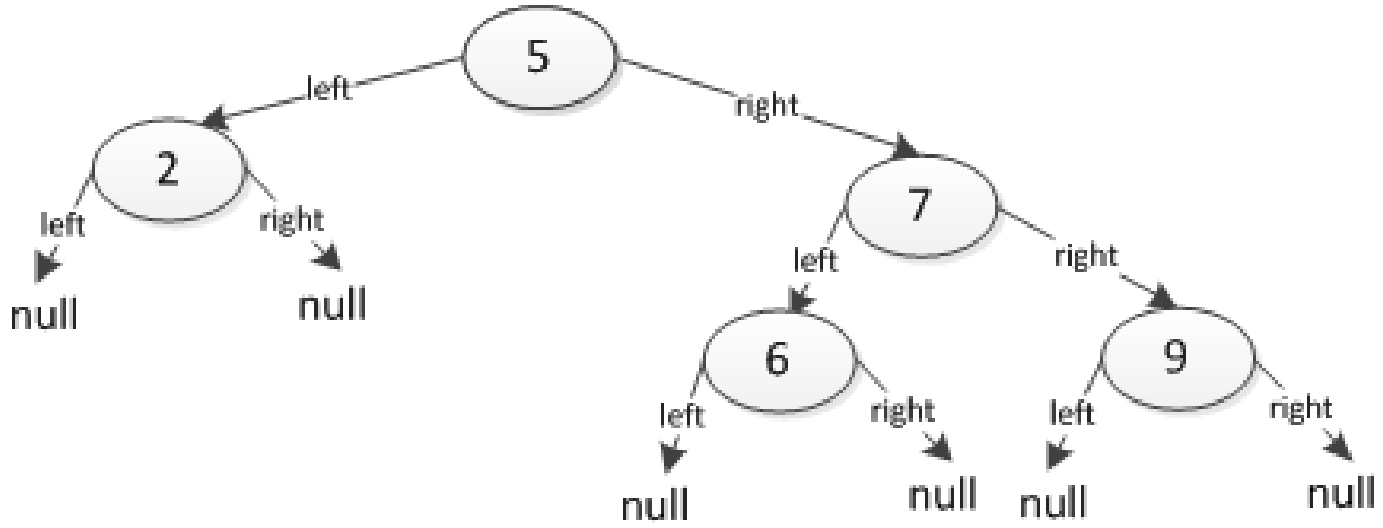
Number of nodes = 7
Height = 3



Number of nodes = 5
Height = 3

A. O(n)
B. O(n log n)
C. O(log n)
D. $O(n^2)$

What is the maximum amount of time it could take to insert a node in a binary search tree? How about find a node?

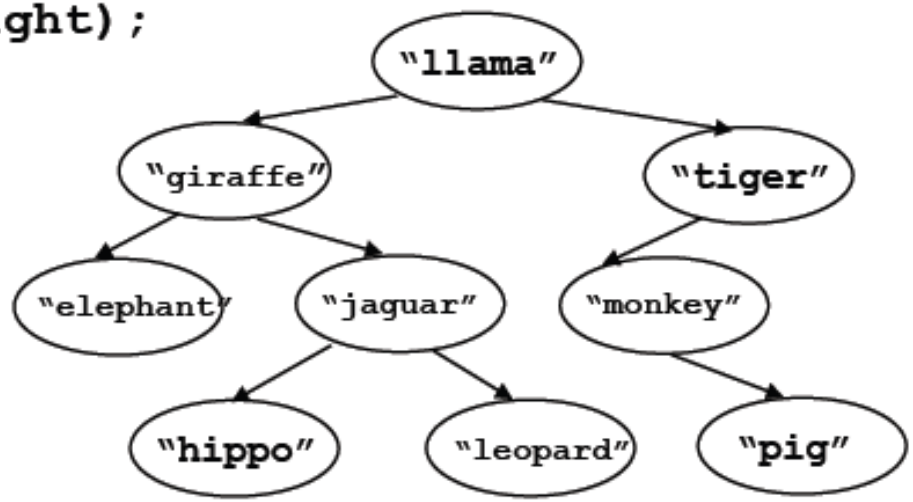


O(tree height)

Printing a search tree in order

- When is *root* printed?
  - After left *subtree*, before right *subtree*.

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



- *Inorder traversal*

What We Did In Class Today

1. You know the meaning of : root, leaf, binary search tree, parent/child, subtree, height, depth, height-balanced.
2. You wrote some slick recursive tree code.
3. You know why inserting and removing in a balanced binary search tree are O(log n), and iterating over the elements in sorted order is O(n)