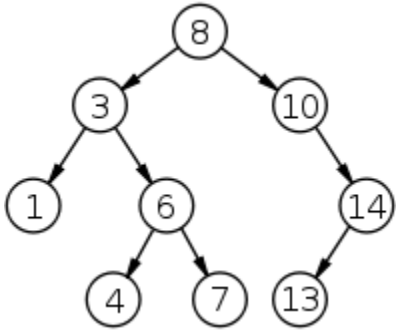


Q1-3

CSSE 230 Day 7

More BinaryTree methods
Tree Traversals

After today, you should be able to…
… traverse trees on paper & in code

Announcements

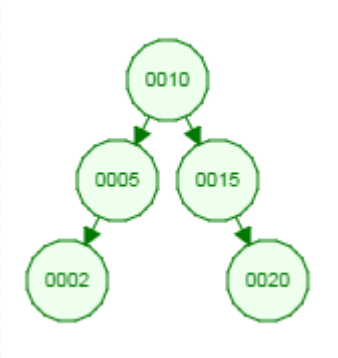
- Please complete the StacksAndQueues partner evaluation in Moodle after you submit your final code.
  - Due Friday
- Doublets is next programming assignment.
  - Solve it with a partner – meet later during today's class.
  - Instructor demo later too.
- Questions (Exam, Stacks & Queues, HW3)?

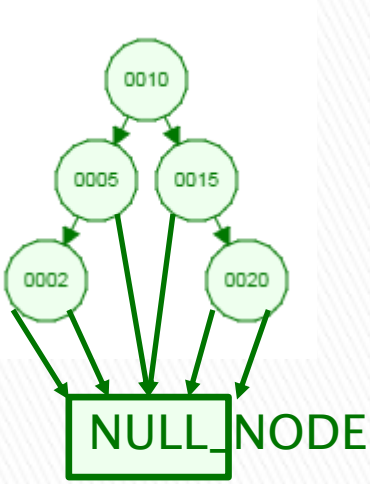
Questions?

Quiz question: What became clear to you as a result of class?

CSSE230 student: I was **TREE**ted to some good knowledge by the time I **LEAF**t the classroom.

A dummy NULL_NODE lets you recurse to a simpler base case while avoiding null pointer exceptions





4 possibilities for children (leaf, Left only, Right only, Both)

1 possibility for children: Both (which could be NULL_NODE)

A dummy NULL_NODE lets you recurse to a simpler base case while avoiding null pointer exceptions

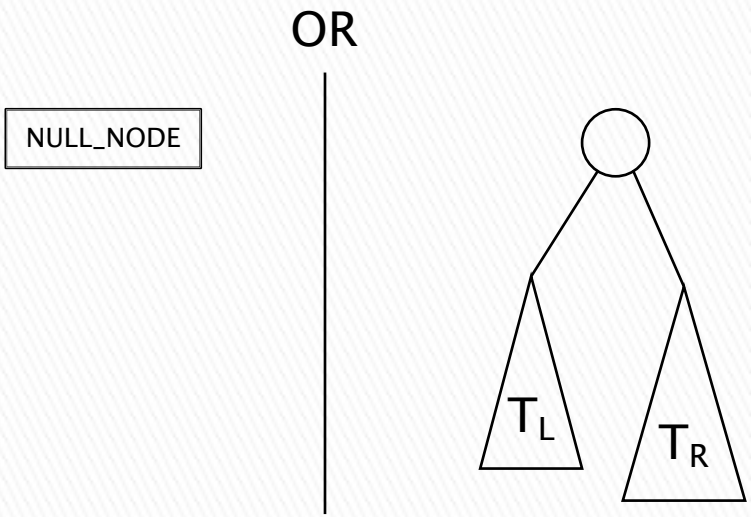
```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    public BinarySearchTree() {
        root = null;
    }

    public int size() {
        if (root == null) {
            return 0;
        }
        return root.size();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public int size() {
            if (left == null && right == null) {
                return 1;
            } else if (left == null) {
                return right.size() + 1;
            } else if (right == null) {
                return left.size() + 1;
            } else {
                return left.size() + right.size() + 1;
            }
        }
```

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    private final BinaryNode NULL_NODE = new BinaryNode();

    public BinarySearchTree() {
        root = NULL_NODE;
    }

    public int size() {
        return root.size();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public BinaryNode(T element) {
            this.data = element;
            this.left = NULL_NODE;
            this.right = NULL_NODE;
        }

        public int size() {
            if (this == NULL_NODE) {
                return 0;
            }
            return left.size() + right.size() + 1;
        }
```

Simpler

Simpler

Definition of Binary Tree, NULL_NODE version



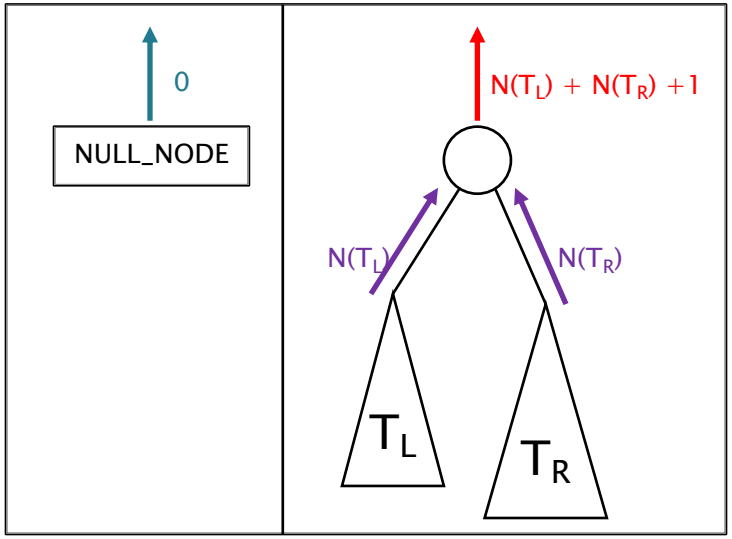
More Trees

Comment out unused tests and uncomment as you go

Write `containsNonBST(T item)` now.

Notice the pattern: size

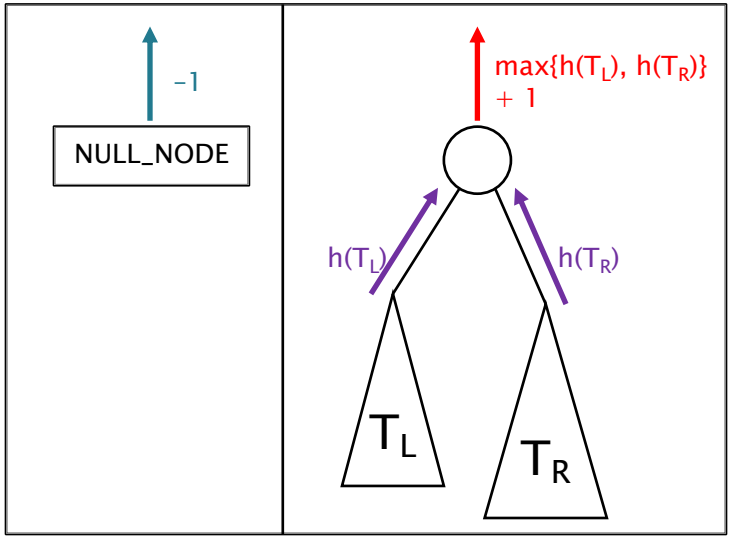
- If (node is null)
  - Return something simple
- Recurse to the left
- Recurse to the right
- Combine results with this node




```
1  public class BinarySearchTree<T> {
2      private BinaryNode root;
3
4      private final BinaryNode NULL_NODE = new BinaryNode();
5
6      public BinarySearchTree() {
7          root = NULL_NODE;
8      }
9
10     public int size() {
11         return root.size();
12     }
13
14     class BinaryNode {
15         private T data;
16         private BinaryNode left;
17         private BinaryNode right;
18
19         public BinaryNode() {
20             this.data = null;
21             this.left = null;
22             this.right = null;
23         }
24
25         public int size() {
26             if (this == NULL_NODE) {
27                 return 0;
28             }
29             return left.size() + right.size() + 1;
30         }
31     }
32 }
```

Notice the pattern: height

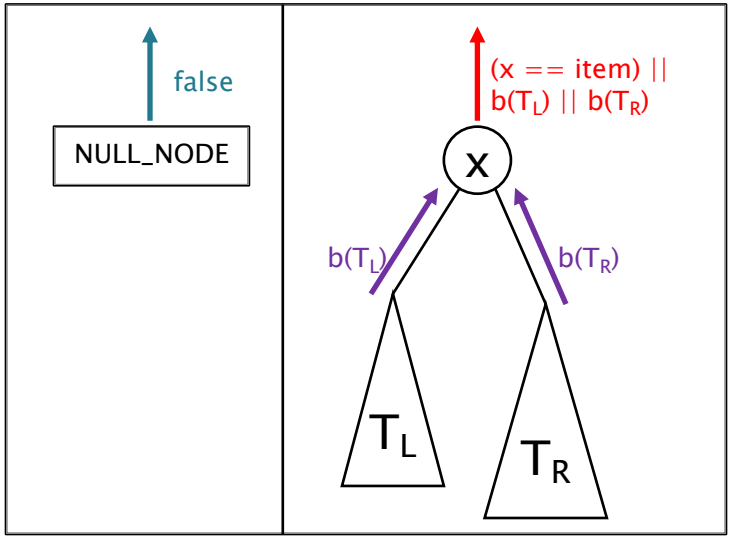
- If (node is null)
  - Return something simple
- Recurse to the left
- Recurse to the right
- Combine results with this node




```
1  public class BinarySearchTree<T> {
2      private BinaryNode root;
3
4      private final BinaryNode NULL_NODE = new BinaryNode();
5
6      public BinarySearchTree() {
7          root = NULL_NODE;
8      }
9
10     public int height() {
11         return root.height();
12     }
13
14     class BinaryNode {
15         private T data;
16         private BinaryNode left;
17         private BinaryNode right;
18
19         public BinaryNode() {
20             this.data = null;
21             this.left = null;
22             this.right = null;
23         }
24
25         public int height() {
26             if (this == NULL_NODE)
27                 return -1;
28             }
29             return Math.max(left.height(), right.height()) + 1;
30         }
31     }
```

Notice the pattern: contains

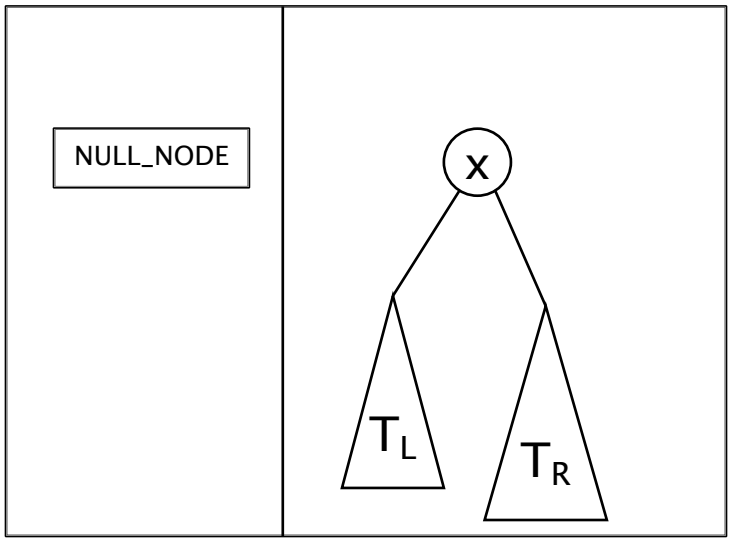
- If (node is null)
  - Return something simple
- Recurse to the left
- Recurse to the right
- Combine results with this node




```
1  public class BinarySearchTree<T> {
2      private BinaryNode root;
3
4      private final BinaryNode NULL_NODE = new BinaryNode();
5
6      public BinarySearchTree() {
7          root = NULL_NODE;
8      }
9
10     public boolean containsNonBST(T item) {
11         return root.containsNonBST(item);
12     }
13
14     class BinaryNode {
15         private T data;
16         private BinaryNode left;
17         private BinaryNode right;
18
19         public BinaryNode() {
20             this.data = null;
21             this.left = null;
22             this.right = null;
23         }
24
25         public boolean containsNonBST(T item) {
26             if (this == NULL_NODE) {
27                 return false;
28             }
29             return this.data.equals(item) ||
30                    left.containsNonBST(item) ||
31                    right.containsNonBST(item);
32         }
33     }
34 }
```

What else could you do with this recursive pattern?

- If (node is null)
  - Return something simple
- Recurse to the left
- Recurse to the right
- Combine results with this node

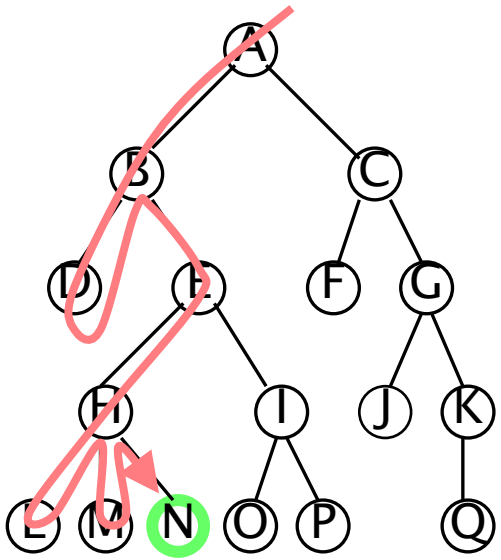


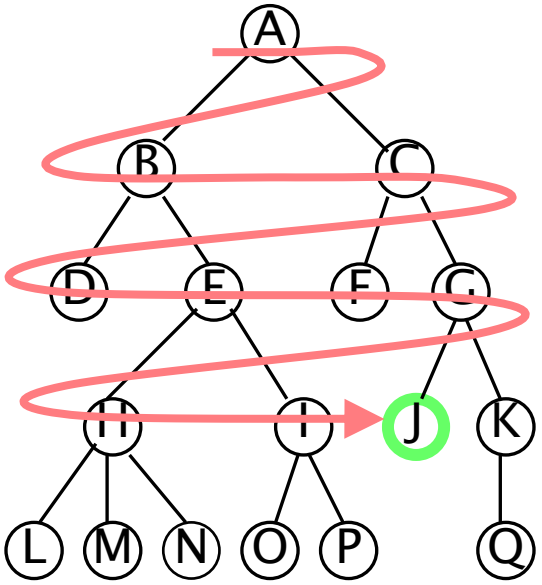
- Print the tree contents
- Sum the values of the nodes
- Dump the contents to an array list
- …

- All involve a *recursive traversal* of the tree.
- Question: in what *order* to visit nodes?

Binary tree traversals

- Depth-first
  - PreOrder ("top-down")
    - root, left, right
  - InOrder ("left-to-right")
    - left, root, right
  - PostOrder ("bottom-up")
    - left, right, root
- Breadth-first / LevelOrder
  - Level-by-level, left-to-right





Depth-first traversals using recursion

```
public void printPreOrder() {
        if (this == NULL_NODE) return;
        System.out.println(this.data.toString());
        left.printPreOrder();
        right.printPreOrder();
}

public void printInOrder() {
        if (this == NULL_NODE) return;
        left.printInOrder();
        System.out.println(this.data.toString());
        right.printInOrder();
}

public void printPostOrder() {
        if (this == NULL_NODE) return;
        left.printPostOrder();
        right.printPostOrder();
        System.out.println(this.data.toString());
}
```

If the tree has N nodes, what's the big-O run-time of each traversal?

Converting the tree to an ArrayList gives an easy solution for toString()

- Brainstorm how to write:

```
public ArrayList<T> toArrayList()
```

- Then BST toString() will simply be:

```
return toArrayList().toString();
```

Efficiency of toArrayList()

- **`toArrayList()`** is most efficient if we
  - Create the list only once, in the header
  - Pass (a reference to) the list down the recursion
    - All the "communication" is top-down (parent-to-child)

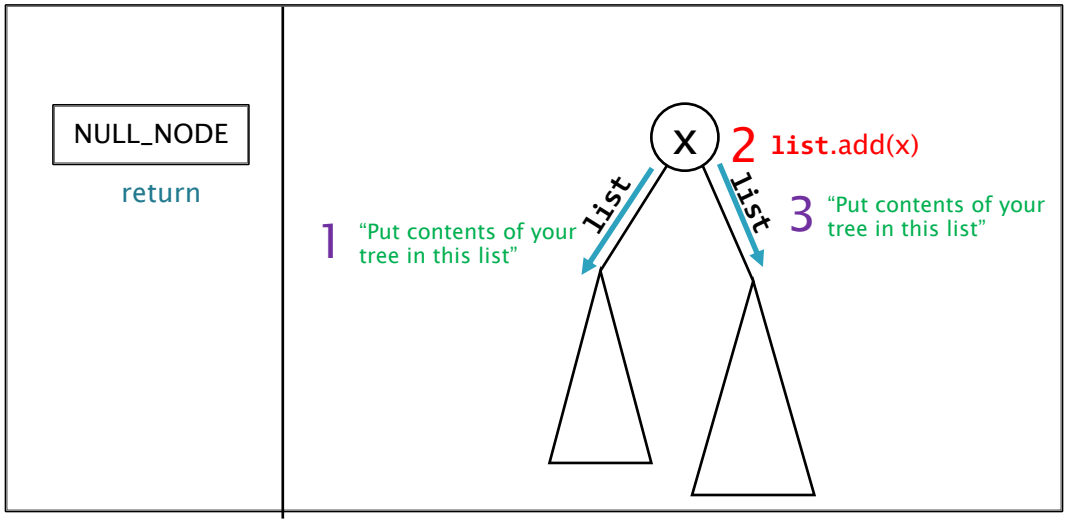
Tree level (header)

```
ArrayList list
        = new ArrayList();
```

`list`

"Put contents of your tree in this list"

Node level (recursion)



Use the recursive pattern when you want to process the whole tree at once

Size(), height(), contains(), toArrayList(), toString(), etc.

What if we want an iterator (one element at a time)?

Next class

Doublets Intro