

CSSE 230 Day 7

More BinaryTree methods
Tree Traversals

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

Announcements

- ▶ Doublets is next assignment.
- ▶ Also with a partner – meet later during today's class.
- ▶ Instructor demo later too.

Announcements

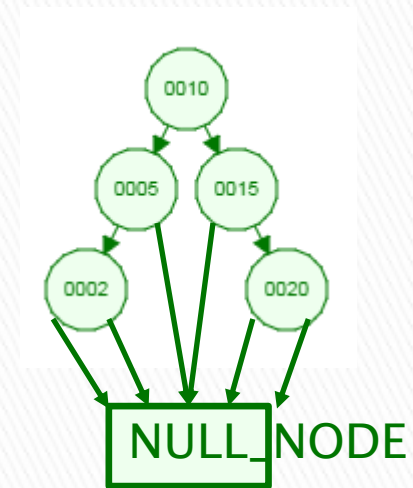
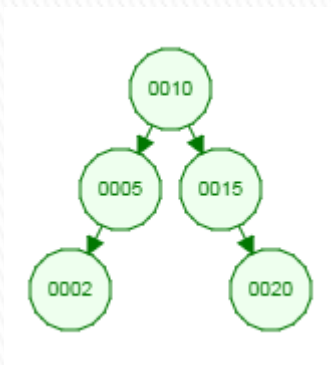
- ▶ Please complete the Stacks&Queues partner evaluation in Moodle after you submit your final code. Due as soon as you submit the project (or by Mon night).
- ▶ Optional Exam 1 review session:
 - Tue 9/18 from 8–11pm, Percopo classroom, led by Connor Boyle
- ▶ Questions (exam, Stacks & Queues, HW3)?

Questions?

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

CSSE230 student: I was **treated** to some good knowledge by the time I **leaft**.

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

```
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;
            }
        }
    }
}
```

```
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(T element) {
20            this.data = element;
21            this.left = NULL_NODE;
22            this.right = NULL_NODE;
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 }
```

Simpler

Simpler

More Trees

Comment out unused tests and
uncomment as you go

Write `containsNonBST(T item)` now.

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 }
```


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    }
32 }
```

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
- ▶ Lots more
- ▶ In what order should we print nodes?

Binary tree traversals

- ▶ InOrder (left-to-right, if tree is spread out)
 - Left, root, right
- ▶ PreOrder (top-down, depth-first)
 - root, left, right
- ▶ PostOrder (bottom-up)
 - left, right, root
- ▶ LevelOrder (breadth-first)
 - Level-by-level, left-to-right within each level

If the tree has N nodes, what's the (worst-case) big-Oh run-time of each traversal?

```
// Print tree rooted at current node using preorder
public void printPreOrder( ) {
    System.out.println( element );           // Node
    if( left != null )
        left.printPreOrder( );              // Left
    if( right != null )
        right.printPreOrder( );             // Right
}

// Print tree rooted at current node using postorder
public void printPostOrder( ) {
    if( left != null )
        left.printPostOrder( );             // Left
    if( right != null )
        right.printPostOrder( );           // Right
    System.out.println( element );          // Node
}

// Print tree rooted at current node using inorder
public void printInOrder( ) {
    if( left != null )
        left.printInOrder( );               // Left
    System.out.println( element );          // Node
    if( right != null )
        right.printInOrder( );             // Right
}
```

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

- ▶ Brainstorm how to write:

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

- ▶ Then BST toString() will simply be:

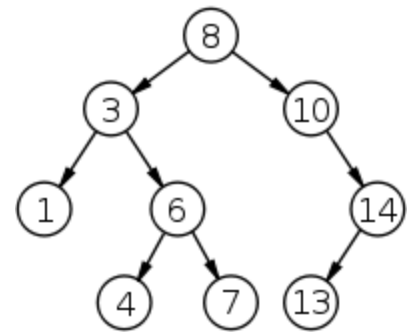
```
return toArrayList().toString();
```

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