CSE 143
Lecture 20

Binary Search Trees, continued

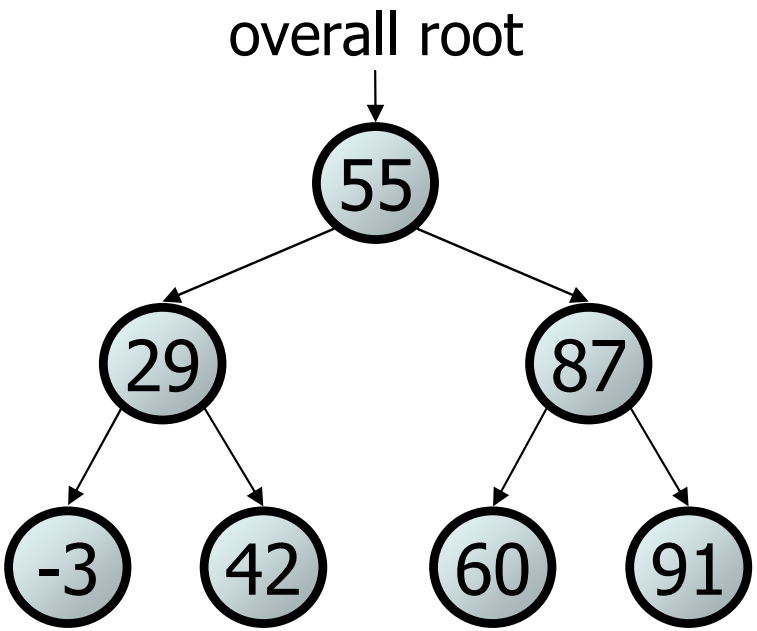
read 17.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Binary search trees

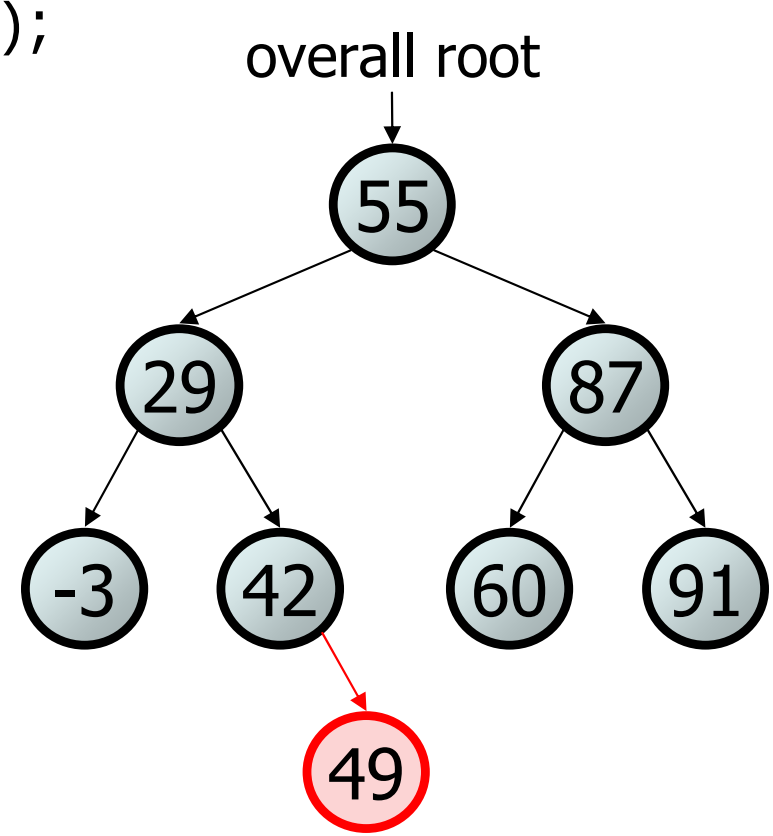
- **binary search tree** ("BST"): a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



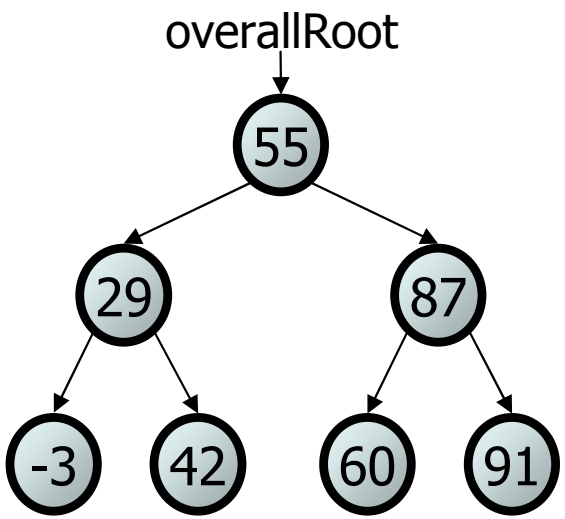
Exercise

- Add a method `add` to the `IntTree` class that adds a given integer value to the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree, and add the new value in the appropriate place to maintain ordering.
  - `tree.add(49);`



An incorrect solution

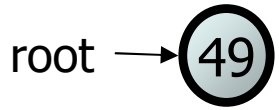
```
// Adds the given value to this BST in sorted order.
// (THIS CODE DOES NOT WORK PROPERLY!)
public void add(int value) {
    add(overallRoot, value);
}

private void add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (root.data > value) {
        add(root.left, value);
    } else if (root.data < value) {
        add(root.right, value);
    }
    // else root.data == value;
    // a duplicate (don't add)
}
```



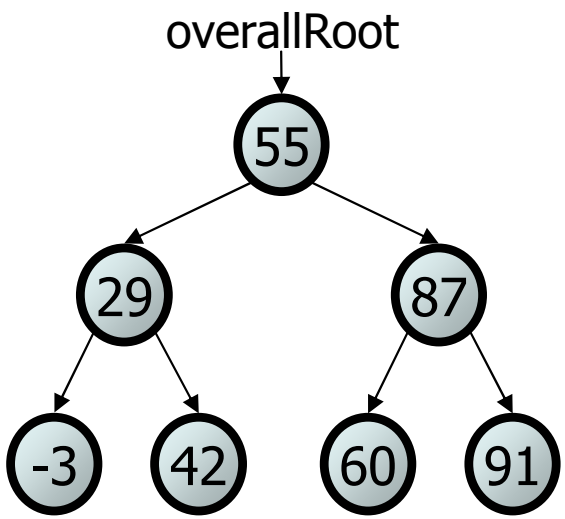
- Why doesn't this solution work?

The problem

- Much like with linked lists, if we just modify what a local variable refers to, it won't change the collection.




```
private void add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
```



  - In the linked list case, how did we correct this problem? How did we actually modify the list?

A poor correct solution

```
// Adds the given value to this BST in sorted order. (bad style)
public void add(int value) {
    if (overallRoot == null) {
        overallRoot = new IntTreeNode(value);
    } else if (overallRoot.data > value) {
        add(overallRoot.left, value);
    } else if (overallRoot.data < value) {
        add(overallRoot.right, value);
    }
    // else overallRoot.data == value; a duplicate (don't add)
}

private void add(IntTreeNode root, int value) {
    if (root.data > value) {
        if (root.left == null) {
            root.left = new IntTreeNode(value);
        } else {
            add(overallRoot.left, value);
        }
    } else if (root.data < value) {
        if (root.right == null) {
            root.right = new IntTreeNode(value);
        } else {
            add(overallRoot.right, value);
        }
    }
    // else root.data == value; a duplicate (don't add)
}
```

x = change(x);

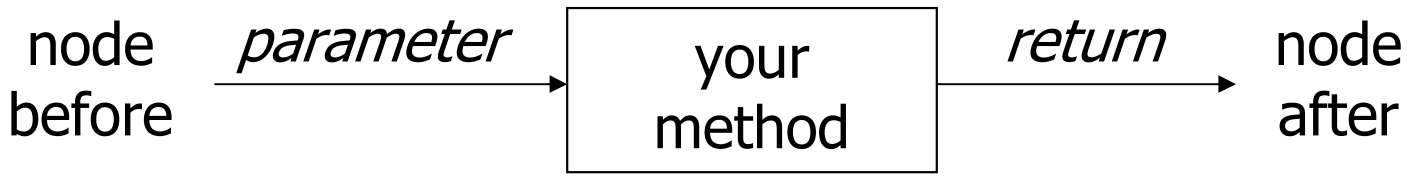
- All String object methods that modify a String actually return a new String object.
  - If we want to modify a string variable, we must re-assign it.

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // lil bow wow
s = s.toUpperCase();
System.out.println(s);   // LIL BOW WOW
```

  - We call this general algorithmic pattern **x = change(x);**
  - We will use this approach when writing methods that modify the structure of a binary tree.

Applying x = change(x)

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node

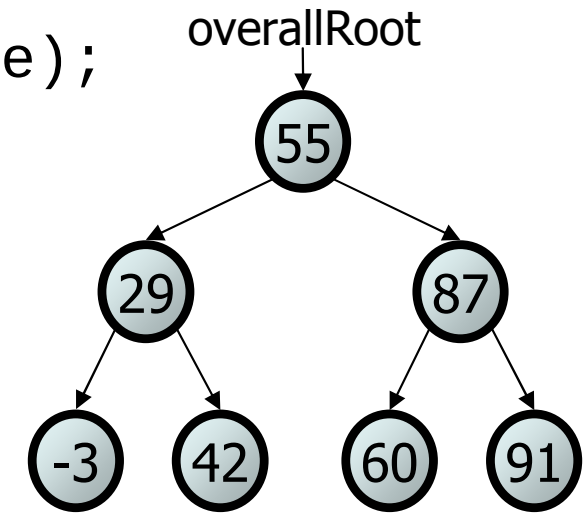


- In order to actually change the tree, you must reassign:

```
overallRoot = change(overallRoot, parameters);
root.left = change(root.left, parameters);
root.right = change(root.right, parameters);
```

A correct solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    overallRoot = add(overallRoot, value);
}

private IntTreeNode add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (root.data > value) {
        root.left = add(root.left, value);
    } else if (root.data < value) {
        root.right = add(root.right, value);
    } // else a duplicate

    return root;
}
```

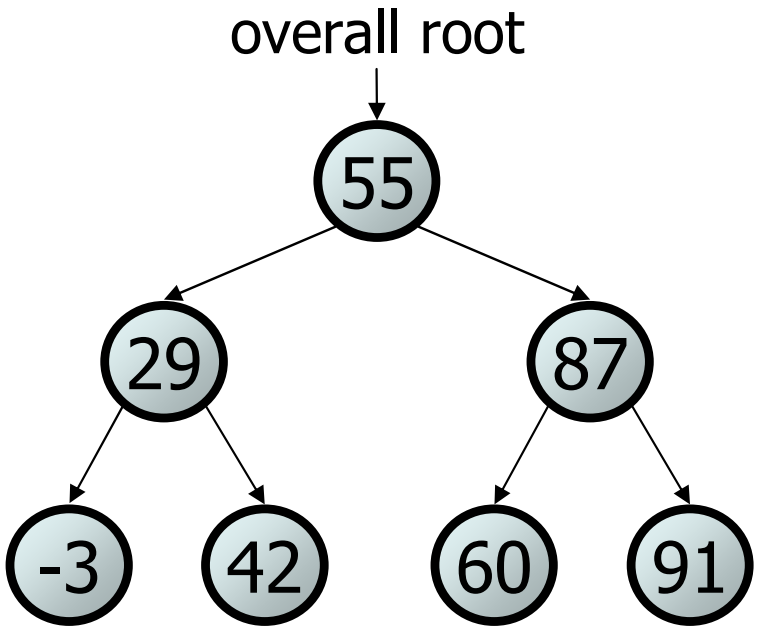


- Think about the case when `root` is a leaf...

Exercise

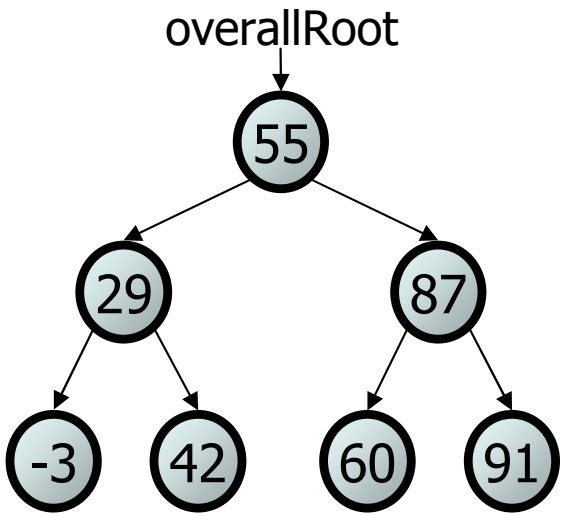
- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

```
int min = tree.getMin();  // -3
```



Exercise solution

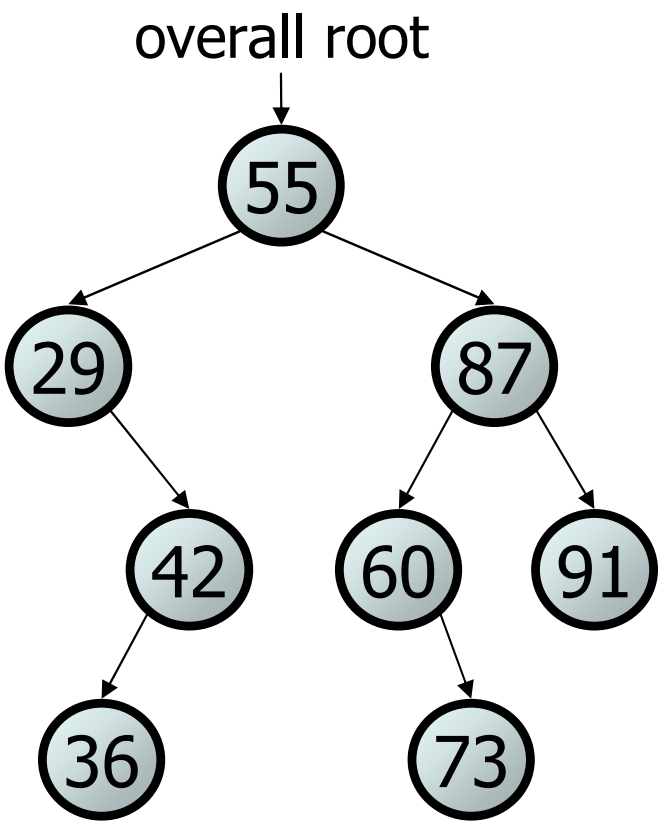
```
// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public int getMin() {
    if (overallRoot == null) {
        throw new NoSuchElementException();
    }
    return getMin(overallRoot);
}

private int getMin(IntTreeNode root) {
    if (root.left == null) {
        return root.data;
    } else {
        return getMin(root.left);
    }
}
```



Exercise

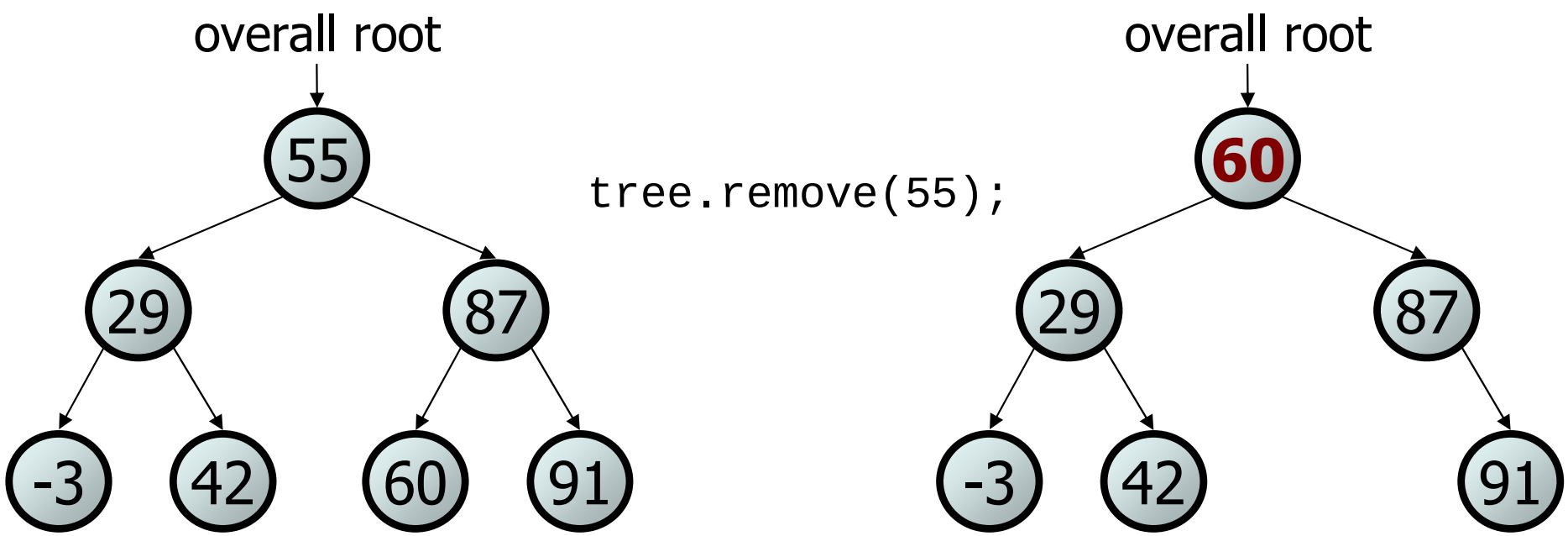
- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Assume that the elements of the `IntTree` constitute a legal binary search tree, and remove the value in such a way as to maintain ordering.

  - `tree.remove(73);`
  - `tree.remove(29);`
  - `tree.remove(87);`
  - `tree.remove(55);`



Cases for removal

- Possible states for the node to be removed:
  - a leaf: replace with null
  - a node with a left child only: replace with left child
  - a node with a right child only: replace with right child
  - a node with both children: replace with min value from right



Exercise solution

```
// Removes the given value from this BST, if it exists.
public void remove(int value) {
    overallRoot = remove(overallRoot, value);
}

private IntTreeNode remove(IntTreeNode root, int value) {
    if (root == null) {
        return null;
    } else if (root.data > value) {
        root.left = remove(root.left, value);
    } else if (root.data < value) {
        root.right = remove(root.right, value);
    } else {  // root.data == value; remove this node
        if (root.right == null) {
            return root.left;    // no L child; replace w/ R
        } else if (root.left == null) {
            return root.right;   // no R child; replace w/ L
        } else {
            // both children; replace w/ min from R
            root.data = getMin(root.right);
            root.right = remove(root.data, root.right);
        }
    }
    return root;
}
```