CSE 143
Lecture 21

Binary Search Trees, continued

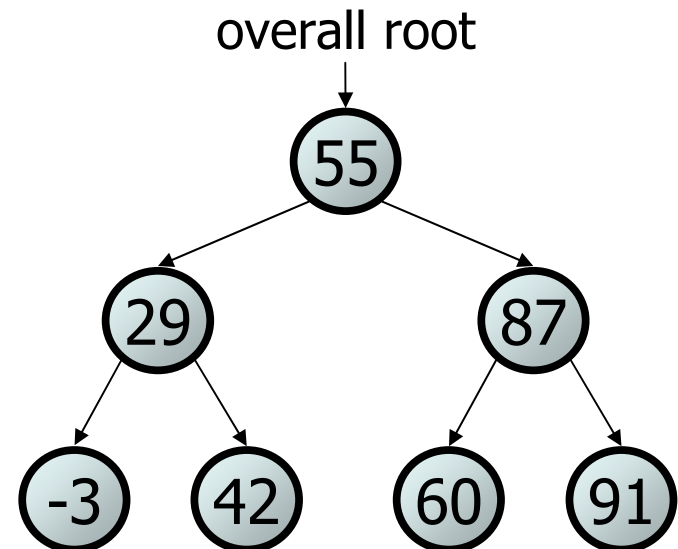
read 17.3

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

Binary search trees

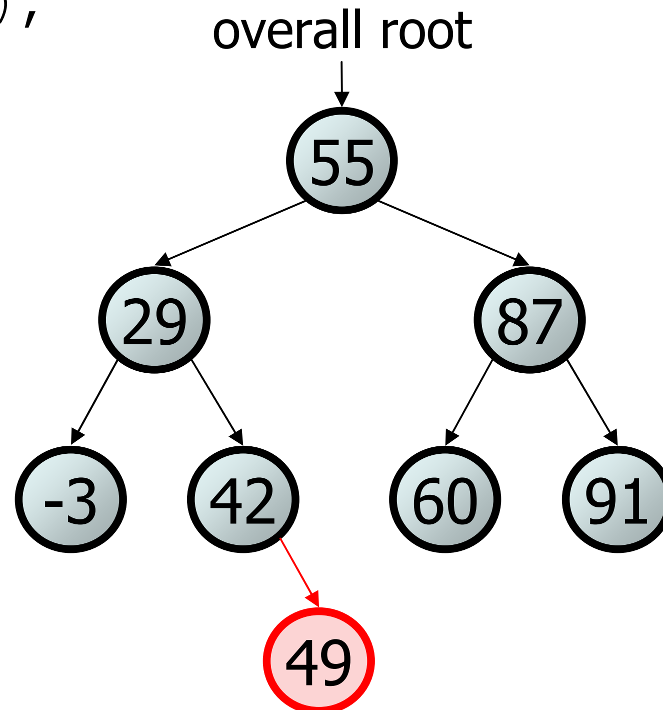
- **binary search tree** ("BST"): a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees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

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



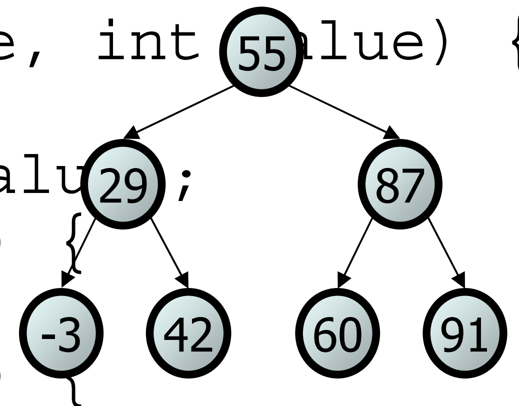
Exercise

- 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);`



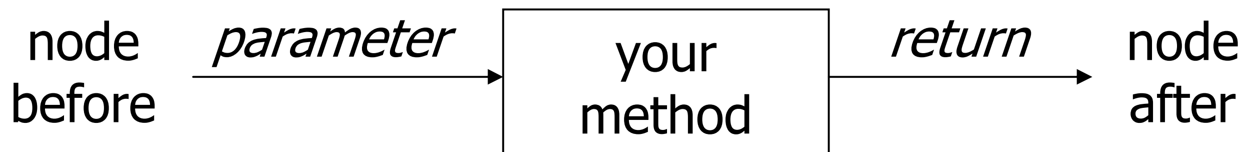
The incorrect solution

```
public class SearchTree {
  private IntTreeNode overallRoot;

  // 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 node, int value) {
      if (node == null) {
          node = new IntTreeNode(value);
      } else if (value < node.data) {
          add(node.left, value);
      } else if (value > node.data) {
          add(node.right, value);
      }
```



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

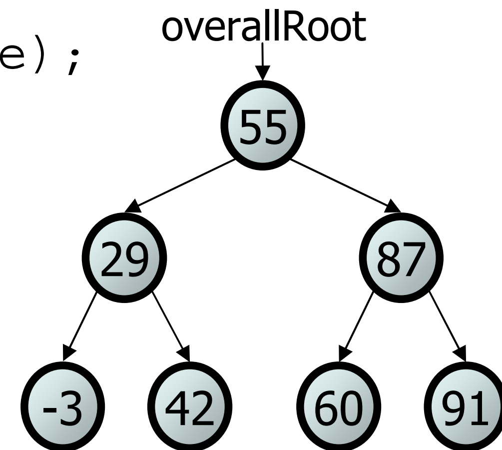


- In order to actually change the tree, you must reassign:

```
overallRoot = change(overallRoot, ...);
node = change(node, ...);
node.left = change(node.left, ...);
node.right = change(node.right, ...);
```

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

    return node;
}
```

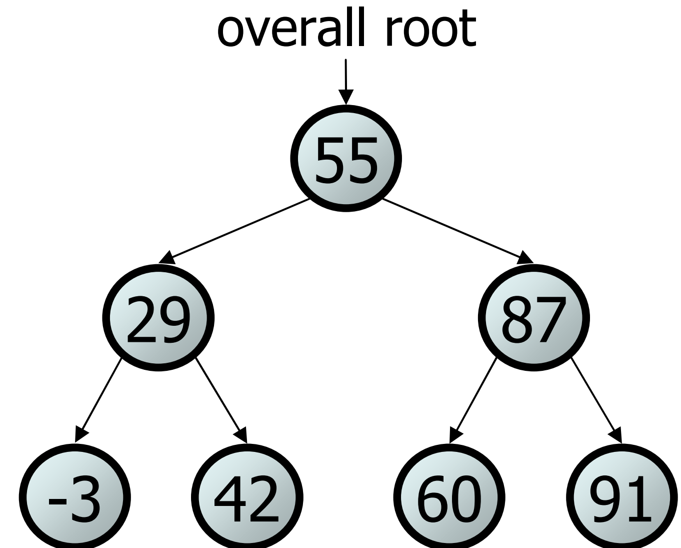


- Think about the case when `root` is a leaf...

Exercise

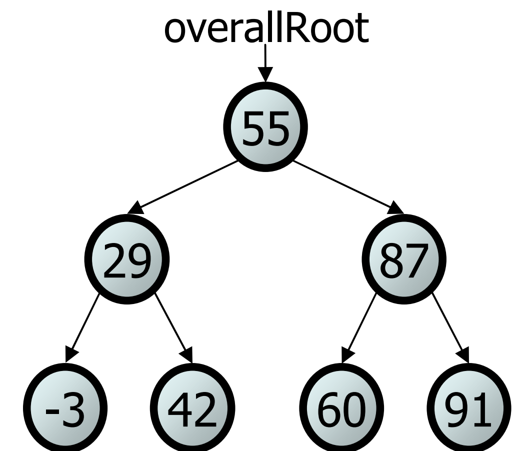
- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

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



Exercise solution

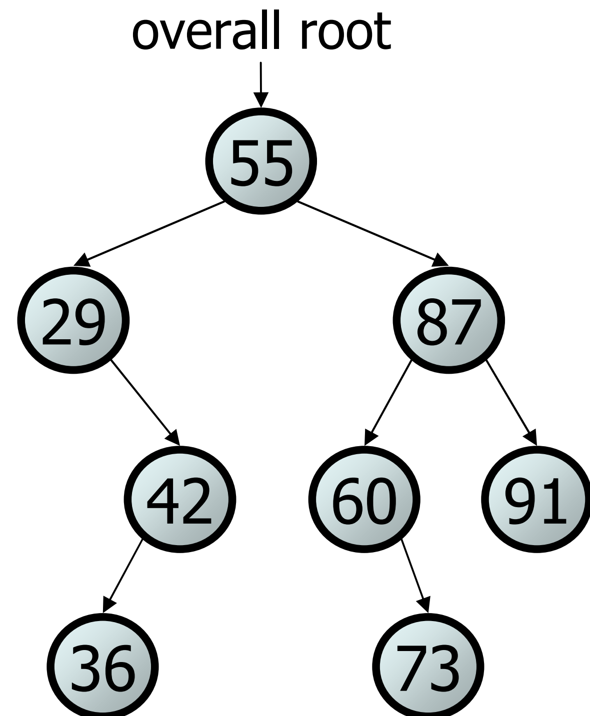
```
// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public int getMin() {
    if (overallRoot == null) {
        throw new NoSuchElementException();
    }
    return getMin(overallRoot);
}

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



Exercise

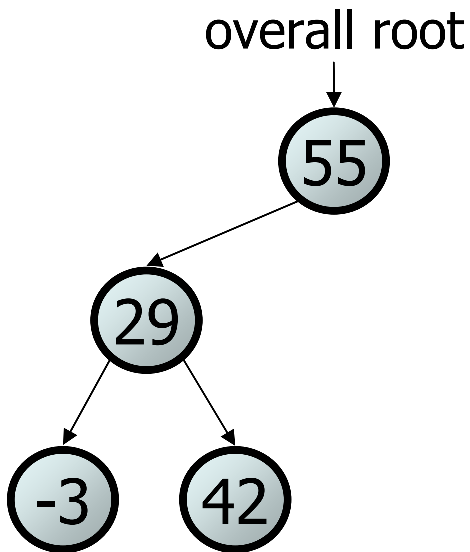
- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Assume that the elements of the `IntTree` constitute a legal binary search tree, and remove the value in such a way as to maintain ordering.

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

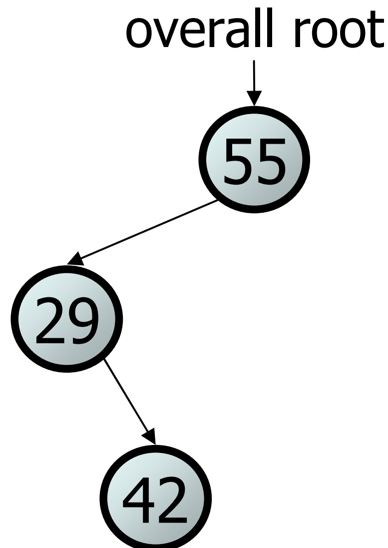


Cases for removal 1

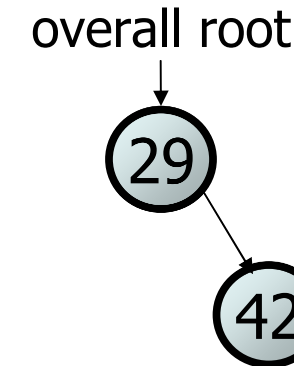
1. a leaf: replace with null
2. a node with a left child only: replace with left child
3. a node with a right child only: replace with right child



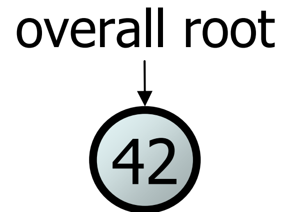

```
tree.remove(-3);
```




```
tree.remove(55);
```

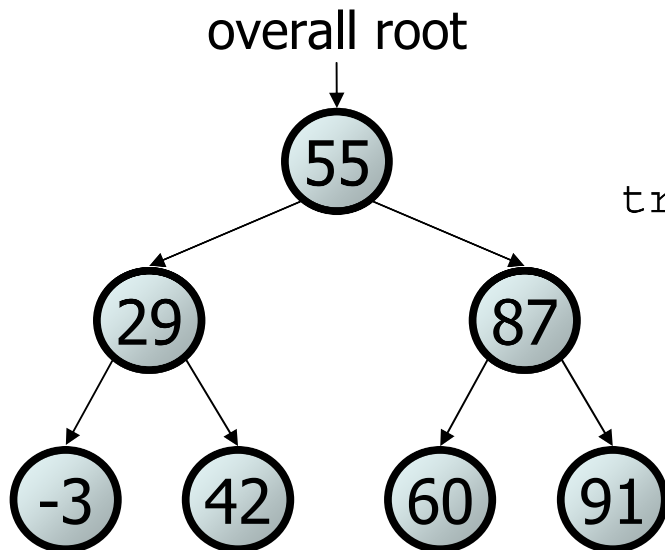



```
tree.remove(29);
```

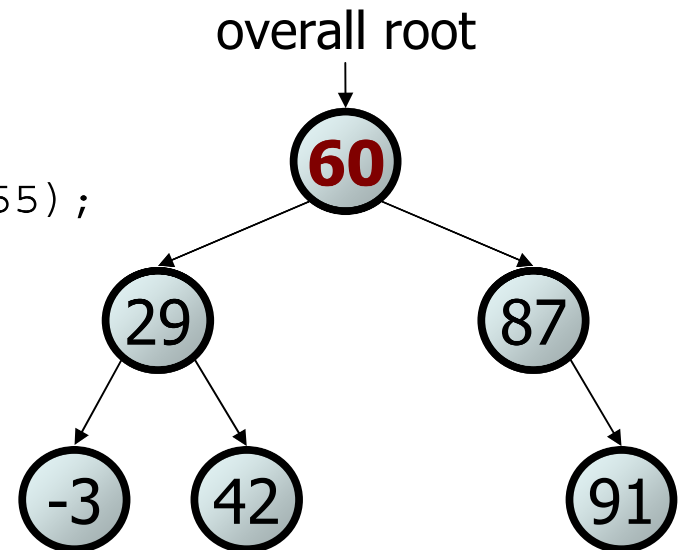


Cases for removal 2

4. a node with **both** children: replace with **min from right**



`tree.remove(55);`



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 R child; replace w/ L
        } else if (root.left == null) {
            return root.right;   // no L child; replace w/ R
        } else {
            // both children; replace w/ min from R
            root.data = getMin(root.right);
            root.right = remove(root.right, root.data);
        }
    }
    return root;
}
```