

CSE 143

Lecture 22

Binary Search Trees continued; Tree Sets

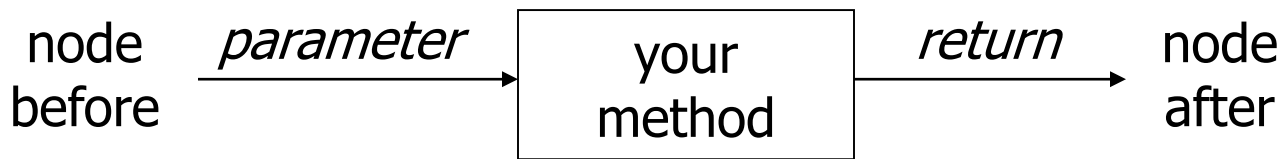
read 17.3 - 17.5

slides adapted from Marty Stepp and Hélène Martin

<http://www.cs.washington.edu/143/>

Recall: $x = \text{change}(x)$

- 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:

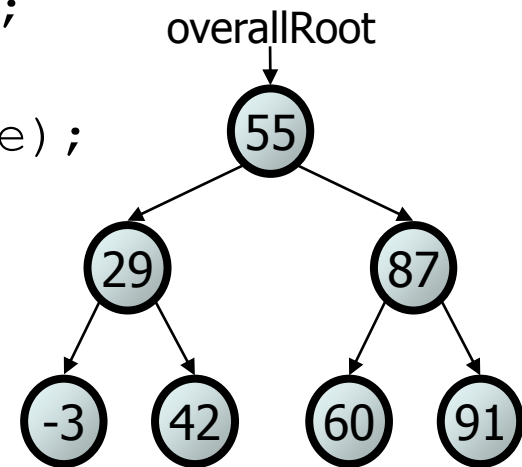
```
node           = change (node, parameters) ;  
node.left      = change (node.left, parameters) ;  
node.right     = change (node.right, parameters) ;  
overallRoot   = change (overallRoot, parameters) ;
```

Add method

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

    return node;
}
```



A poor correct solution

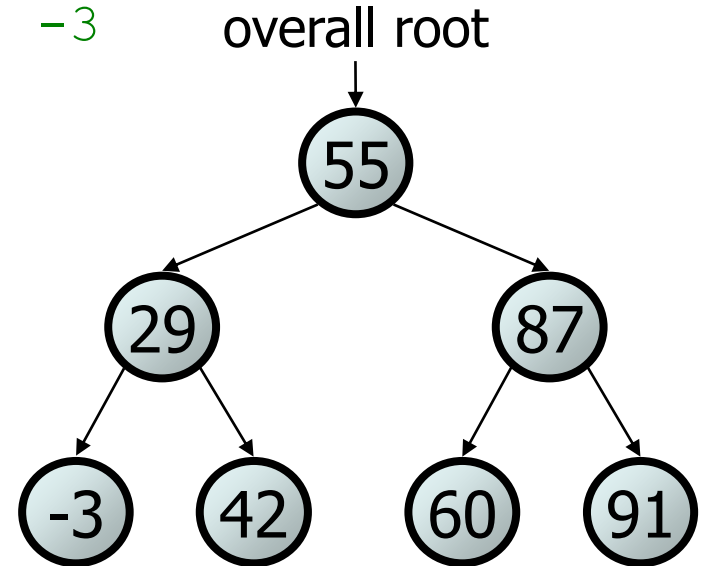
```
// Adds the given value to this BST in sorted order. (bad style)
public void add(int value) {
    if (overallRoot == null) {
        overallRoot = new IntTreeNode(value);
    } else if (overallRoot.data > value) {
        add(overallRoot.left, value);
    } else if (overallRoot.data < value) {
        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)
}
```

Exercise

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

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

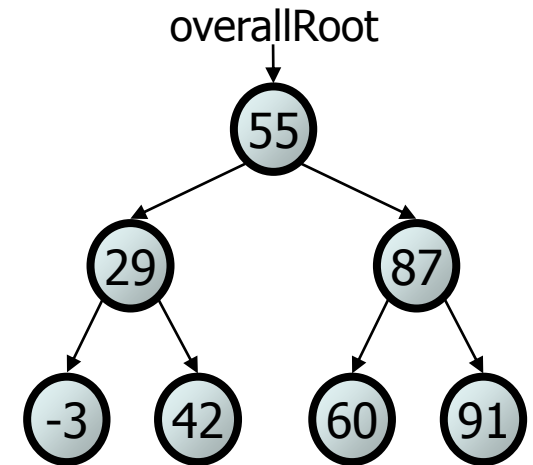


Exercise solution

```
// 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) ;  
}
```

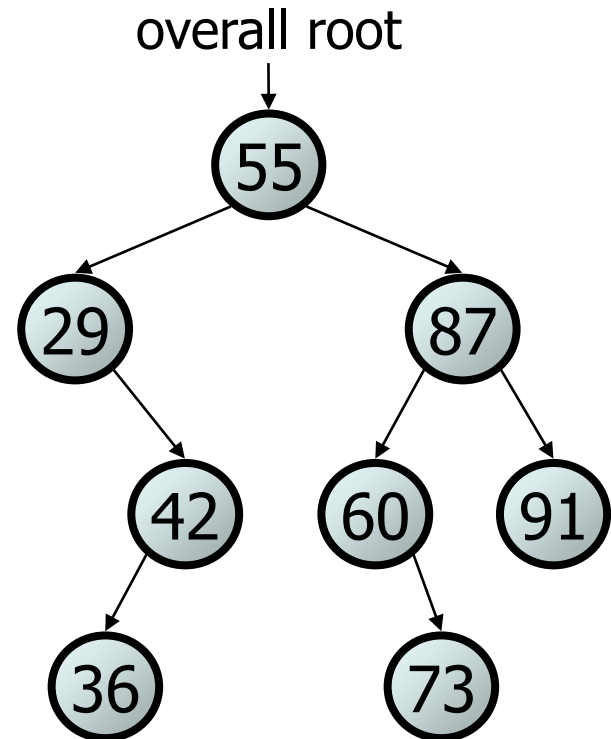
```
private int getMin(IntTreeNode root) {  
    if (root.left == null) {  
        return root.data;  
    } else {  
        return getMin(root.left);  
    }  
}
```



Exercise

- Add a method `remove` to the `BinarySearchTree` class that removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.

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



Cases for removal 1

1. a **leaf**:

2. a node with a **left child only**:

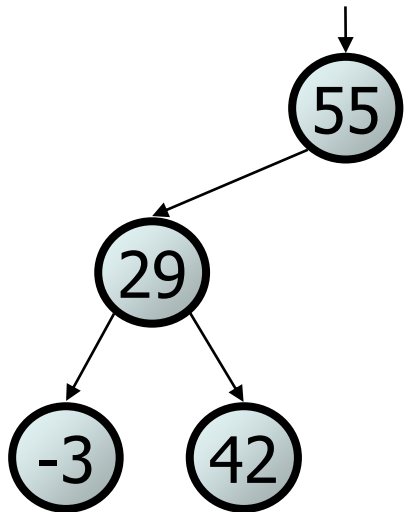
3. a node with a **right child only**:

replace with `null`

replace with left child

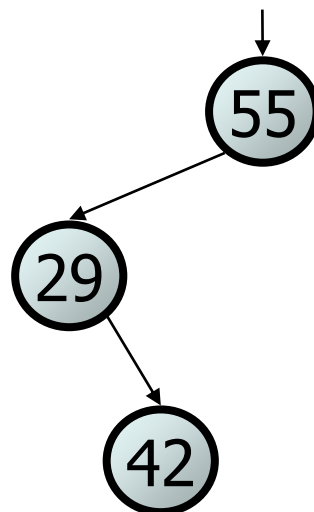
replace with right child

overall root



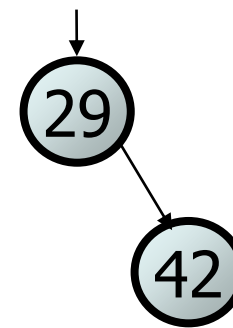
```
tree.remove(-3);
```

overall root



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

overall root



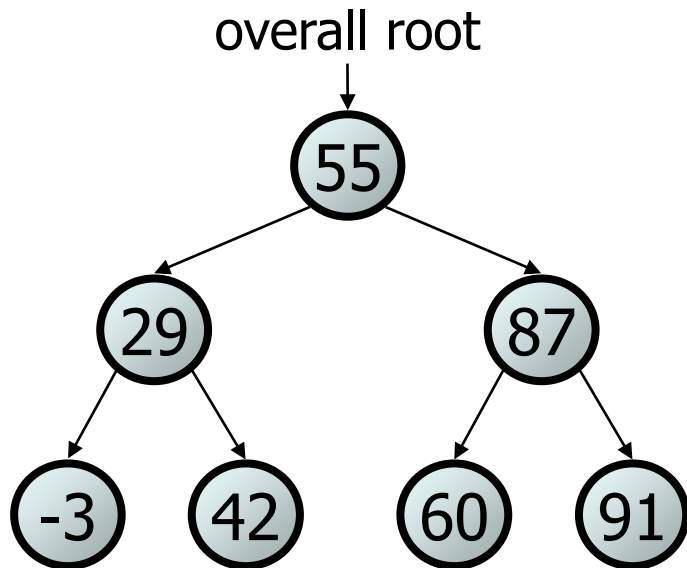
```
tree.remove(29);
```

overall root

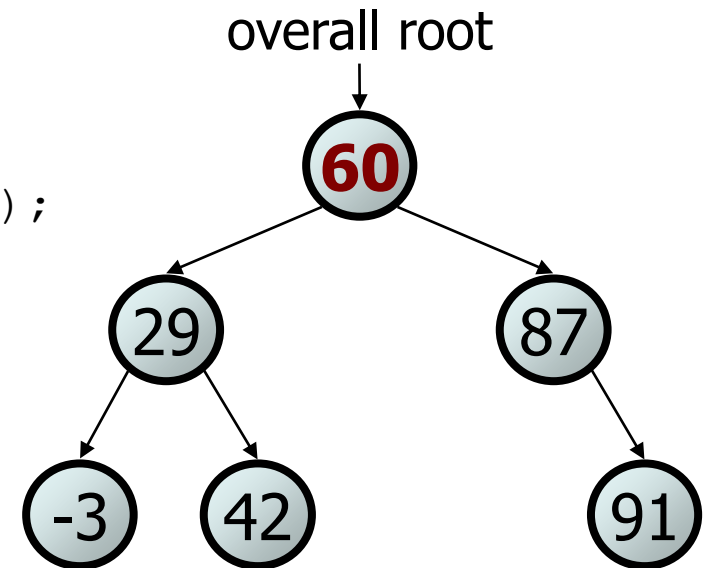


Cases for removal 2

4. a node with **both** children: replace with **min from right**
- (replacing with max from left would also work)



`tree.remove(55);`



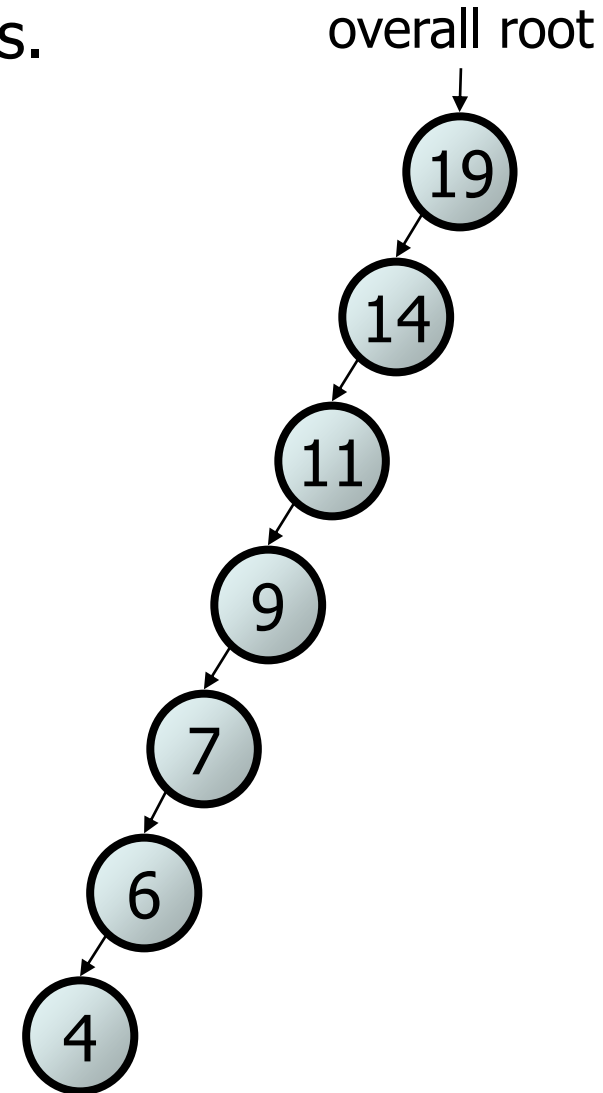
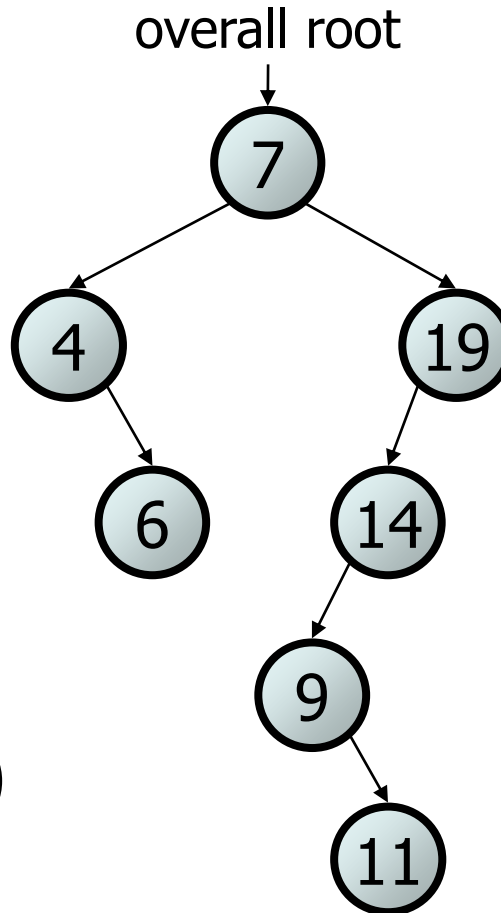
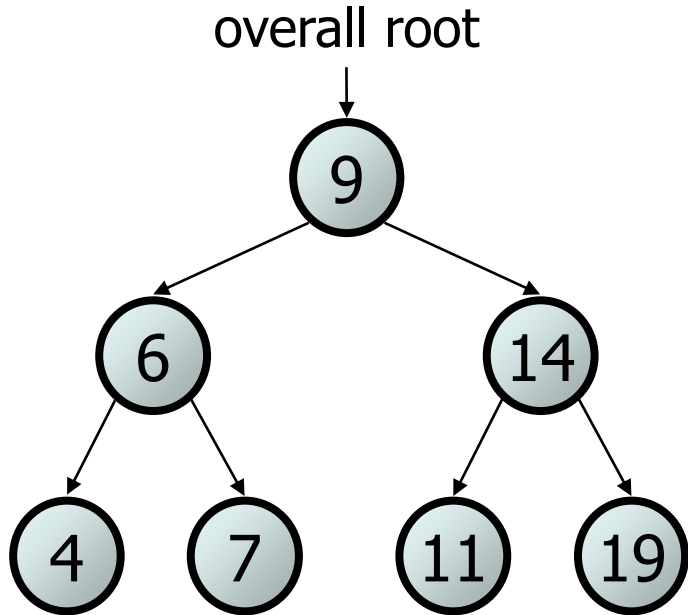
Exercise solution

```
// 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;
}
```

Searching BSTs

- The BSTs below contain the same elements.
 - What orders are "better" for searching?



Trees and balance

- **balanced tree:** One whose subtrees differ in height by at most 1 and are themselves balanced.
 - A balanced tree of N nodes has a height of $\sim \log_2 N$.
 - A very unbalanced tree can have a height close to N .
- The runtime of adding to / searching a BST is closely related to height.
- Some tree collections (e.g. `TreeSet`) contain code to balance themselves as new nodes are added.

