CSE 143
Lecture 16

Binary Trees

slides created by Alyssa Harding
http://www.cs.washington.edu/143/

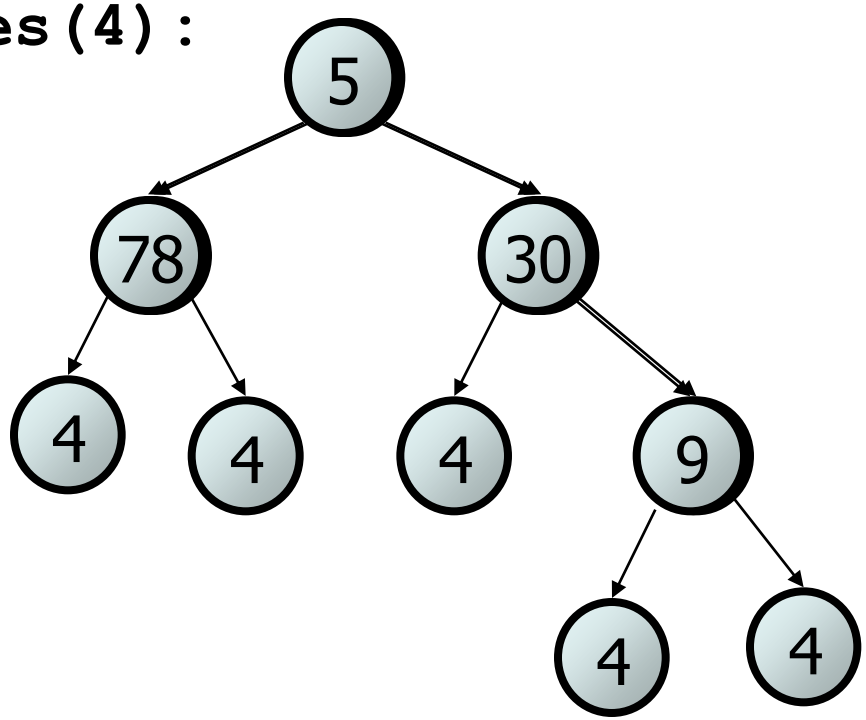
Binary trees

- Before, we wrote methods that *traversed* a tree
- Now we want to *change the structure* of a tree

Example: `addLeaves`

- Our first example, **`addLeaves`**, will take an **`int`** parameter and add leaves with that value

- For instance, with a tree variable **`t`**, a call of **`t.addLeaves(4)`**:



Example: `addLeaves`

- We start with our public/private pair:

```
public void addLeaves(int n) {
  addLeaves(overallRoot, n);
}

private void addLeaves(IntTreeNode root, int n){
   ...
}
```

The public method starts the whole process by calling the private method with the **`overallRoot`** while the private method focuses on one subtree

Example: addLeaves

- Then we focus on our base case:

```
private void addLeaves(IntTreeNode root, int n) {
  if ( root == null ) {
    root = new IntTreeNode(n);
  } else {
    ...
  }
}
```

If we've reached a null node, it means we've reached an empty subtree where we can add a leaf

Example: addLeaves

- Then we focus on our recursive case:

```
private void addLeaves(IntTreeNode root, int n) {
   if ( root == null ) {
      root = new IntTreeNode(n);
   } else {
      addLeaves(root.left,n);
      addLeaves(root.right,n);
   }
}
```

Otherwise, I want to add leaves to both of the subtrees.
Good thing I have a method that does that!

Example: addLeaves

- So we're done...

```
private void addLeaves(IntTreeNode root, int n) {
   if ( root == null ) {
      root = new IntTreeNode(n);
   } else {
      addLeaves(root.left,n);
      addLeaves(root.right,n);
   }
}
```

But this code doesn't change anything.

x = change(x)

- Why not?
- Let's look at an example using **`Point`** objects

x	y
7	42

x = change(x)

```
import java.awt.*;

public class PointTest {
  public static void main(String[] args) {
    Point x = new Point(7, 42);
    System.out.println("x = " + x);
    change(x);
    System.out.println("now x = " + x);
}

public static void change(Point p) {
   p.translate(3, 8);
    p = new Point(-33, -17);
    System.out.println("p = " + p);
  }
}
```

x = change(x)

- Given this code that manipulates a **`Point`**, what will it output?

- The first two lines are straightforward, but the last one might be surprising:

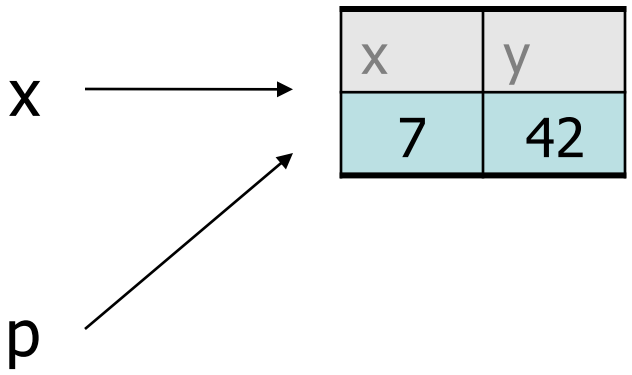
```
x = java.awt.Point[x=7,y=42]
p = java.awt.Point[x=-33,y=-17]
now x = java.awt.Point[x=10,y=50]
```

- **`x`** does not refer to our new **`Point`**, but the old translated **`Point`**

x = change(x)

- When we pass an object as a parameter, the parameter gets a copy of the reference to the object

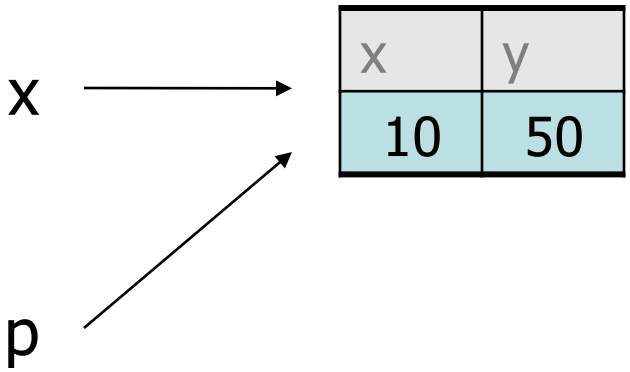
```
change(x);
```



x = change(x)

- So when call a method on **`p`**, it affects the object that **`x`** refers to because they refer to the same object

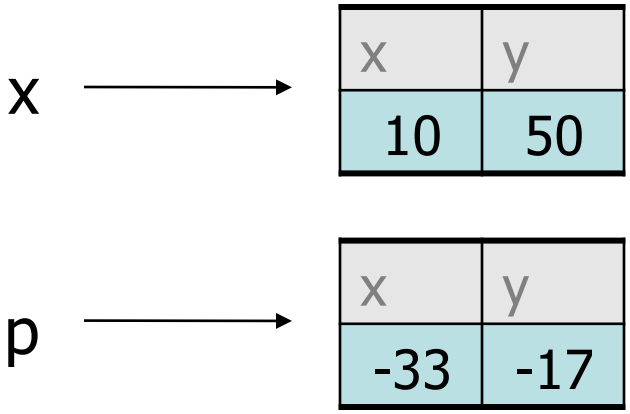
```
p.translate(3, 8);
```



x = change(x)

- But **p** can't change **x** itself

```
p = new Point(-33, -17);
```



x = change(x)

- We can also think of these references like cell phone numbers
- Just like when we passed the parameter, I can give you a copy of Ethan's cell phone number
- Just like the method call, you can call him too
- But you can't scratch out your copy of his number and assume that mine is destroyed as well

x = change(x)

- How do we get around this?
- We want our original variable, **`x`**, to refer to the same object as **`p`**
- A strategy we call "x assign change of x"
  - x = change(x)

x = change(x)

- To get the reference to our new object back to our original variable, we can return it

```
public static Point change(Point p) {
    p.translate(3, 8);
    p = new Point(-33, -17);
    System.out.println("p = " + p);
    return p;
  }
}
```

x = change(x)

- Then, when we call the method, we assign our original variable to the returned reference

```
public class PointTest {
  public static void main(String[] args) {
    Point x = new Point(7, 42);
    System.out.println("x = " + x);
    x = change(x);
    System.out.println("now x = " + x);
}
```

Now our last print statement uses our new `Point` object

Example: addLeaves

- We can apply this to **addLeaves** to change the references

```
private IntTreeNode addLeaves(IntTreeNode root,
                              int n) {
  if ( root == null ) {
    root = new IntTreeNode(n);
  } else {
    addLeaves(root.left,n);
    addLeaves(root.right,n);
  }
  return root;
}
```

First we make sure that the private method returns the node object

Example: addLeaves

- We also want to use the return value

```
private IntTreeNode addLeaves(IntTreeNode root,
                              int n) {
  if ( root == null ) {
    root = new IntTreeNode(n);
  } else {
    root.left = addLeaves(root.left,n);
    root.right = addLeaves(root.right,n);
  }
  return root;
}
```

Then we make sure that we assign variables to the return value anytime we call the method

Example: `addLeaves`

- The public method also needs to be modified

```
public void addLeaves(int n) {
  overallRoot = addLeaves(overallRoot, n);
}
```

Now we're done!

Example: `removeIfLeaf`

- Now let's do the opposite: **`removeIfLeaf`** takes an **`int`** and removes all leaf nodes storing that data

```
public void removeIfLeaf(int n) {
  overallRoot = removeIfLeaf(overallRoot, n);
}

private IntTreeNode removeIfLeaf(
                        IntTreeNode root, int n) {
  ...
}
```

The public method header looks the same, but our private method returns a node

Example: `removeIfLeaf`

- Now we think of our base case:

```
private IntTreeNode removeIfLeaf(
                        IntTreeNode root, int n) {
  if ( root != null ) {
    if ( root.data == n && root.left == null
                          && root.right == null ) {
       root = null;
    }
    ...
}
```

We'll stop if the root node is null or if we've found a leaf with the same data

Example: `removeIfLeaf`

- Now we think of our recursive case:

```
private IntTreeNode removeIfLeaf(
                        IntTreeNode root, int n) {
  if ( root != null ) {
    if ( root.data == n && root.left == null
                        && root.right == null ) {
       root = null;
    } else {
       root.left = removeIfLeaf(root.left, n);
       root.right = removeIfLeaf(root.right, n);
    }
  }
  return root;
}
```

Otherwise, we want to remove leaves from the left and right subtrees