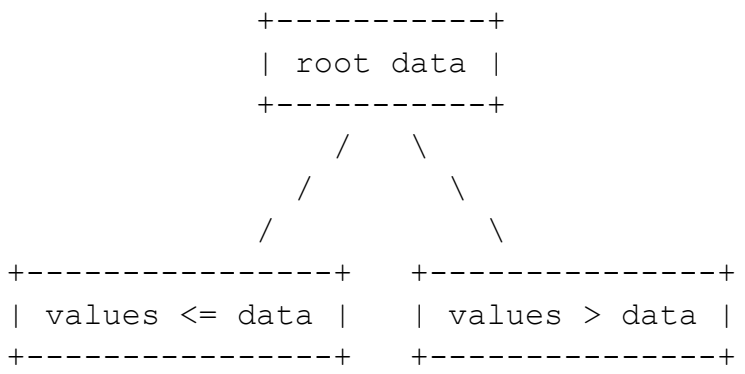
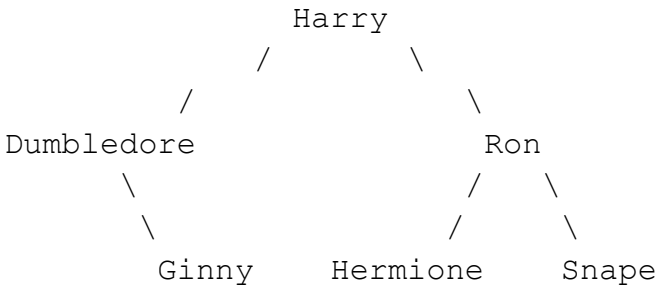
Lecture 18: Binary Search Trees

- There is a subtype of binary trees called BINARY SEARCH TREES
  - A binary search tree is organized so that each subtree has the following format:



  - This is the binary search tree property, and it holds for EVERY subtree, not just the overall root
  - What about duplicates?
    - 1) Don't allow it
    - 2) Put them on the left
    - 3) Put them on the right
    - What matters is consistency
- Example: Inserting into a binary search tree
  - Order: Harry, Ron, Dumbledore, Snape, Hermione, Ginny
  - Tree:



  - Now perform an inorder traversal -->
    - Dumbledore, Ginny, Harry, Hermione, Ron, Snape
  - Alphabetical!
  - Actually not that surprising given the way we built the tree

```
(values <= data)   root data   (values > data)
\------1st-----/   \--2nd--/   \-----3rd-----/
```

  - We repeated this pattern at every subtree, so it always prints alphabetically earlier names first
- How to implement this adding behavior
  - (show the client code) - modify the IntTree class
  - First, we need to start with an empty tree - add a constructor

```
public IntTree() {
    overallRoot = null;
}
```

- Then, we need to write add(value), which will do exactly what we did before - preserving the binary search tree property
- We'll follow the usual structure of a public/private pair, with the private method taking the root of the current subtree that we are looking at

```
public void add(int value) {
    add(overallRoot, value);
}

private void add(IntTreeNode root, int value) {
    ...
}
```

- So our private helper will be recursive - it always is with trees.
- What is our base case? When do we know when to stop traversing and just ADD the node?
  - When there is no node!
  - For example, when initially the root is null and we want to add the node there, or when we traverse down and get to an empty spot.
  - Replace the null reference with a new node

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

- And what if the root is not an empty tree?
  - What did we do when we were adding?
  - We compared the value we were given to the data in the root
  - If it was less than the root's data, we went left
  - If it was greater than the root's data, we went right

```
private void add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (value <= root.data) {
        // add to left
    } else {
        // add to right
    }
}
```

  - So what do we do in the tests? You might want to say

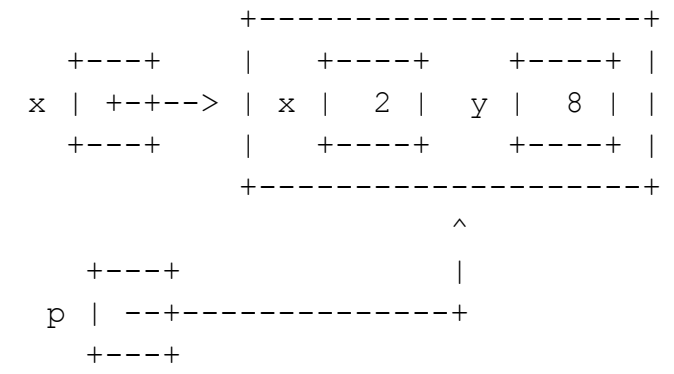
```
if (root.left == null) {
    root.left = new IntTreeNode(value);
}
```

  - But this is redundant, and not recursive zen

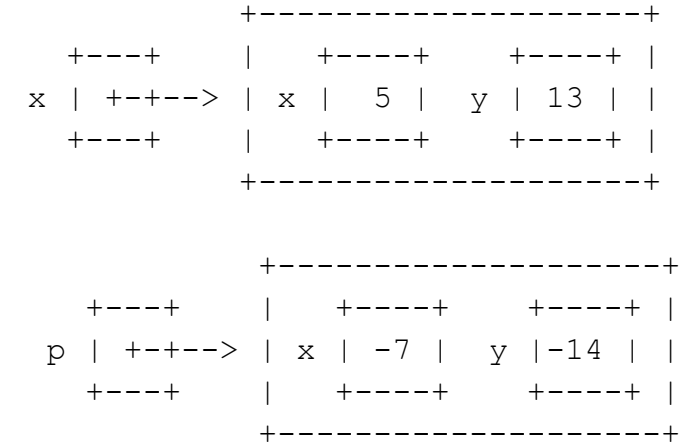
        - “if only we had a method...”

```
private void add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (value <= root.data) {
        add(root.left, value);
    } else {
        add(root.right, value);
    }
}
```

    - (run the code with some numbers)
    - Unfortunately, this doesn’t work
        - The reason has to do with the parameter “root”
        - “root” stores a copy of whatever is passed into it
        - As a result, changing what root points to doesn’t change what was passed in
- Look at PointTest
    - Does the translation reflected in the original object? YES - we are changing the object itself
    - Does changing p change anything in main? NO - we are changing a reference, which was copied



    - Originally p and q are the same phone number - calling the same person
    - But when we update p, it doesn’t affect x



- To solve this, we use what we call the “x = change(x)” idiom
- (change Point example so that the change() method returns a Point)
- The idea is that the method may change the object entirely (creating a new object), so we want to update x in case that happens
- (run PointTest again)

- How can we apply this to the add() example?
  - We return the root, and change the return type

```
private IntTreeNode add(IntTreeNode root, int value) {
    ...
    return root;
}
```

  - And we change our calls on the method so that we follow the x=change(x) pattern

```
public void add(int value) {
   overallRoot = add(overallRoot, value);
}
```

  - This says that we might get the old value (effectively x=x) or a new value back, but if we get a new value, we want to update the overall root to point to it
  - Analogy: your computer breaks, and you send it to the manufacturer to get it fixed. You don’t care if you get the old computer back fixed, or a brand new computer - you just care that you got a computer back
  - We also need to change all other calls on the recursive method

```
private IntTreeNode add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (value <= root.data) {
        root.left = add(root.left, value);
    } else {
        root.right = add(root.right, value);
    }
    return root;
}
```

  - Notice it’s not root = add(root.left, value) - the same thing you pass in is what you assign to - the same “x” on both sides
  - You’ll get practice with this in section next week, and on your homework. It’s a valuable tool when it’s used properly
  - Use when you want to MODIFY an EXISTING tree

- Another useful method for a binary search tree:

```
// post: returns true if overall tree contains value
public boolean contains(int value) {
    return contains(overallRoot, value);
}
// post: returns true if given tree contains value
private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (value == root.data) {
        return true;
    } else if (value &lt; root.data) {
        return contains(root.left, value);
    } else {
        // value > root.data
        return contains(root.right, value);
    }
}
```

- Assignment 6: 20Questions
    - Simulate how nodes are added
    - Show the intermediate questions.txt files
    - Run with bigquestions.txt