Another Tree Method

Write a tree method called height (inside the IntTree class) with the following method signature:

```java
class IntTree {
    // More Tree Methods
    // Introducing BSTs
    // BST Methods
}
```

```java
public int height()
```

that returns the number of nodes on the longest path from the root to any leaf. For example,

```
<table>
<thead>
<tr>
<th>Height</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3 4 5</td>
</tr>
<tr>
<td>3</td>
<td>3 4 5</td>
</tr>
</tbody>
</table>
```

```java
public int height() {
    return height(this.root);
}
```

```java
private int height(IntTreeNode current) {
    if (current == null) {
        return 0;
    } else {
        return 1 + Math.max(height(current.left), height(current.right));
    }
}
```

runtime of contains(7)

Recall contains()

```java
private boolean contains(IntTreeNode current, int value) {
    // If the tree is null, it definitely doesn't contain value... */
    if (current == null) { return false; } // A null tree has height 0
    else if (current.data == value) { return true; } // If current *is* value, we found it! */
    else { // Find the largest path by taking the max
        return contains(current.left, value) || contains(current.right, value); // of both branches recursively (and adding 1 for this node)
    }
}
```

That makes the code $O(n)$. Can we do better?
Which nodes do we visit for value we found? 

```
def contains(IntTreeNode current, Integer min, Integer max) {
    if (current == null) {
        return false;
    } else if ((min != null && current.data < min) ||
        (max != null && current.data > max)) {
        return false;
    } else if (!isBST(current.left, min, current.data)) {
        return false;
    } else if (!isBST(current.right, current.data, max)) {
        return false;
    } else {
        return contains(current.left, value);
    }
}
```

Definition (Binary SEARCH Tree (BST))
A binary tree is a BST when an in-order traversal of the tree yields a sorted list.
To put it another way, a binary tree is a BST when:
- All data “to the left of” a node is less than it.
- All data “to the right of” a node is greater than it.
- All sub-trees of the binary tree are also BSTs.

Example (Which of the following are BSTs?)

```
root
1 2
NO

root
1 2
YES

root
1 2
NO
```

contains (AGAIN!) 6

Write contains() for a BST 
Fix contains so that it takes advantage of the BST properties.

```
private boolean contains(IntTreeNode current, int value) {
    if (current == null) { return false; }
    if (current.data < value) { return contains(current.right, value); }
    else if (current.data > value) { return contains(current.left, value); }
    else { return true; }
}
```

Adding to a BST (Attempt #1) 9

Attempt #1
```
public void add(int value) {
    add(this.root, value);
}
```

Runtime of (better) contains(7)
Consider the following tree: Which nodes do we visit for contains(7)?

```
 It yields a sorted list. Much better!
```

This is the same tree, but now we have to visit all the nodes!

Adding to a BST (Attempt #1) 9

```
public void add(Node current, int value) {
    if (current == null) {
        return;
    } else if (current.data > value) {
        add(current.left, value);
    } else if (current.data < value) {
        add(current.right, value);
    }
}
```

What's wrong with this solution?
Just like with LinkedLists where we must change front or .next, we’re not actually changing anything here. We’re discarding the result.
Consider the following code:

```java
public static void main(String[] args) {
    String s = "hello world";
    s.toUpperCase();
    System.out.println(s);
}
```

OUTPUT

```text
>> hello world
```

We must **USE** the result; otherwise, it gets discarded.

**Adding to a BST (Fixed)**

```java
Fixed Attempt
public void add(int value) {
    this.root = add(this.root, value);
}

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

This works because we **always update the result, always return the result, and always update the root.**

**BST Tips!**

- BSTs can make searching/inserting/etc. much faster.

- Make sure that you can figure out if a tree is a BST or not.

- Whenever you are writing a BST method, you **must** use the `x = change(x)` pattern. It won’t work otherwise.