Another Tree Method

Write a tree method called \textit{height} (inside the \texttt{IntTree} class) with the following method signature:

\begin{verbatim}
public int height()
\end{verbatim}

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

\begin{itemize}
  \item height is 1
  \item height is 5
  \item height is 3
\end{itemize}

\textbf{height Solution}

\begin{verbatim}
public int height() {
    return height(this.root);
}

private int height(IntTreeNode current) {
    // A null tree has height 0
    if (current == null) {
        return 0;
    } else {
        // Find the largest path by taking the max
        // of both branches recursively (and adding 1 for this node)
        return 1 + Math.max(
            height(current.left),
            height(current.right)
        );
    }
}
\end{verbatim}

\textbf{Back to contains}

Consider the following tree: Which nodes do we visit for \textit{contains(7)}?

That makes the code $O(n)$. Can we do better?
In general, we can’t do better. BUT, sometimes, we can!

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?)

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

Example (tree.add(49))

```
add(IntTreeNode current, int value) {
  if (current == null) {
    return false;
  }
  add(current.left, value);
  return false;
  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.

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

```
public boolean add(int value) {
  add(this.root, value);
  return false;
}
```

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

Runtime of (better) contains(7)

Consider the following tree: Which nodes do we visit for contains(7)

That makes the code logn. Much better!

```
private boolean isBST(IntTreeNode current, int min, int max) {
  if (current == null) {
    return true;
  }
  else if (current.data < min || current.data > max) {
    return false;
  }
  else if (!isBST(current.left, min, current.data)) {
    return false;
  }
  else {
    return isBST(current.right, current.data, max);
  }
}
```

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

```
public boolean add(int value) {
  return contains(current.right, value);
}
```

```
private boolean isBST(IntTreeNode current, int min, int max) {
  if (current == null) {
    return true;
  }
  else if (current.data < min || current.data > max) {
    return false;
  }
  else if (!isBST(current.left, min, current.data)) {
    return false;
  }
  else if (current.data > max) {
    return false;
  }
  else {
    return isBST(current.right, current.data, max);
  }
}
```

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

```
private boolean isBST(IntTreeNode current, int min, int max) {
  if (current == null) {
    return true;
  }
  else if (current.data < min || current.data > max) {
    return false;
  }
  else if (!isBST(current.left, min, current.data)) {
    return false;
  }
  else {
    return isBST(current.right, current.data, max);
  }
}
```

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

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

```
public boolean add(int value) {
  return contains(current.right, value);
}
```
Consider the following code:

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

**OUTPUT**
```
>> hello world
```

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

**OUTPUT**
```
>> HELLO WORLD
```

We must use the result; otherwise, it gets discarded.

If you want to write a method that can change the object that a variable refers to, you must do three things:
1. Pass in the original state of the object to the method
2. Return the new (possibly changed) object from the method
3. Re-assign the caller’s variable to store the returned result

```java
public static Point change(Point thePoint) {
    thePoint = new Point(99, -1);
    return thePoint;
}
```

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.