CSE 143

Computer Programming II
Binary Search Trees (BSTs)
1 More Tree Methods

2 Introducing BSTs

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

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

that returns the number of nodes on the **longest path** from the root to any leaf. For example,
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)
        );
    }
}
private boolean contains(IntTreeNode current, int value) {
    /* If the tree is null, it definitely doesn’t contain value... */
    if (current == null) { return false; }

    /* If current *is* value, we found it! */
    else if (current.data == value) { return true; }

    else {
        return contains(current.left, value) || contains(current.right, value);
    }
}
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; }

    /* If current *is* value, we found it! */
    else if (current.data == value) { return true; }

    else {
        return contains(current.left, value) ||
                contains(current.right, value);
    }
}
```

Runtime of `contains(7)`

Consider the following tree: Which nodes do we visit for `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; }

    /* If current *is* value, we found it! */
    else if (current.data == value) { return true; }

    else {
        return contains(current.left, value) ||
               contains(current.right, value);
    }
}
```

Runtime of `contains(7)`

Consider the following tree: Which nodes do we visit for `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?)

```
[Diagram]
```

- The left tree is **NO**
- The middle tree is **YES**
- The right tree is **NO**
Write contains() for a BST

Fix contains so that it takes advantage of the BST properties.

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;
    }

    /* If current *is* value, we found it! */
    else if (current.data == value) {
        return true;
    }

    else if (current.data < value) {
        return contains(current.right, value);
    }
    else {
        return contains(current.left, value);
    }
}
```
Tracing the new contains

Runtime of (better) contains(7)

Consider the following tree:

That makes the code $\log n$. Much better!

WARNING!

Consider the following tree:

This is the same tree, but now we have to visit all the nodes!
Tracing the new `contains`

Runtime of (better) `contains(7)`

Consider the following tree:

```
   4
  / \
 2   6
/ \  /\
1  3 5 7
```

Which nodes do we visit for `contains(7)`?

```
   4
  /  \
 2   6
/ \  /  \
1  3 5 7
```

That makes the code $\log n$. Much better!

**WARNING!**

Consider the following tree:

```
   1
  /  \
 2   3
  |   |
 4   5
  |   |
 5   6
  |   |
 7
```

This is the same tree, but now we have to visit all the nodes!
Add a method `add` in the BST class with the following signature:

```java
public void add(int value);
```

that preserves the BST property.

Example (`tree.add(49)`)

Before

```
      55
     /   
   29    87
  /     / 
-3     42  60  91
```

After

```
      55
     /   
   29    87
  /     /   
-3     42   60  91
```

49
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:

1. `public static void main(String[] args) {`
2. `String s = "hello world";`
3. `s.toUpperCase();`
4. `System.out.println(s);`
5. `}`

```
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
p = change(p); // in main
```

```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**.
- 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 \( x = \text{change}(x) \) pattern. It won’t work otherwise.