CSE 143

Computer Programming II
Binary Search Trees (BSTs)
Outline

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,
```java
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)
        );
    }
}
```
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:

```
4
 /   \
2     6
|     /   \
1   3     5   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)`
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?)**

<table>
<thead>
<tr>
<th>Tree</th>
<th>Right of Root</th>
<th>Left of Root</th>
<th>All Sub-Trees BST?</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Tree 1" /></td>
<td>2, 3</td>
<td>1</td>
<td>NO</td>
</tr>
<tr>
<td><img src="#" alt="Tree 2" /></td>
<td>3, 2</td>
<td>1, 3</td>
<td>YES</td>
</tr>
<tr>
<td><img src="#" alt="Tree 3" /></td>
<td>6, 12</td>
<td>1, 3</td>
<td>NO</td>
</tr>
</tbody>
</table>
Write a function `isBST` with the following signature:

```java
private boolean isBST(IntTreeNode current, int min, int max)
```

that returns true if the tree at root `current` is a BST.

```java
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);
  }
}
```
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:

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

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

**WARNING!**

Consider the following tree:

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

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:

Which nodes do we visit for contains (7)

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

private void add(IntTreeNode current, int value) {
    if (current == null) {
        current = new IntTreeNode(value);
    } 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:

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

OUTPUT

>> hello world

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

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
1   p = change(p); // in main
2   public static Point change(Point thePoint) {
3       thePoint = new Point(99, -1);
4       return thePoint;
5   }
```
Adding to a BST (Fixed)

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