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
1 public int height() {
2     return height(this.root);
3 }

4 private int height(IntTreeNode current) {

5     // A null tree has height 0
6     if (current == null) {
7         return 0;
8     }
9     else {
10        // Find the largest path by taking the max
11        // of both branches recursively (and adding 1 for this node)
12        return 1 + Math.max(
13        height(current.left),
14        height(current.right)
15    );
16    }
17 }
18 }
```
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
```

Which nodes do we visit for contains(7)? That makes the code $O(n)$. Can we do better?
Back to contains

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>Diagram</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram 1" /></td>
<td>NO</td>
</tr>
<tr>
<td><img src="image2.png" alt="Diagram 2" /></td>
<td>YES</td>
</tr>
<tr>
<td><img src="image3.png" alt="Diagram 3" /></td>
<td>NO</td>
</tr>
</tbody>
</table>
Write a function `isBST` with the following signature:

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

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

```java
private boolean isBST(IntTreeNode current, Integer min, Integer max) {
    if (current == null) {
        return true;
    } 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 {
        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 \texttt{contains}

Runtime of (better) \texttt{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 \textbf{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))**

Before

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

After

```
      55
     /   
   29    87
  /     /  
-3    42   60
       /  
      49  91
```
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**

```
>> 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

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