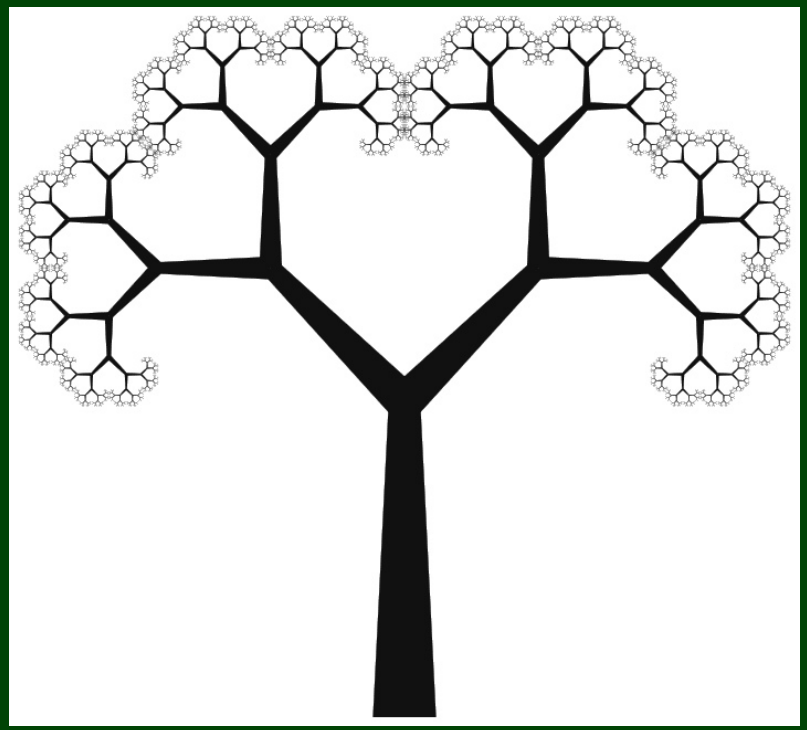
CSE 143X

Accelerated Computer Programming I/II

Binary Search Trees (BSTs)



Outline

1 More Tree Methods

2 Introducing BSTs

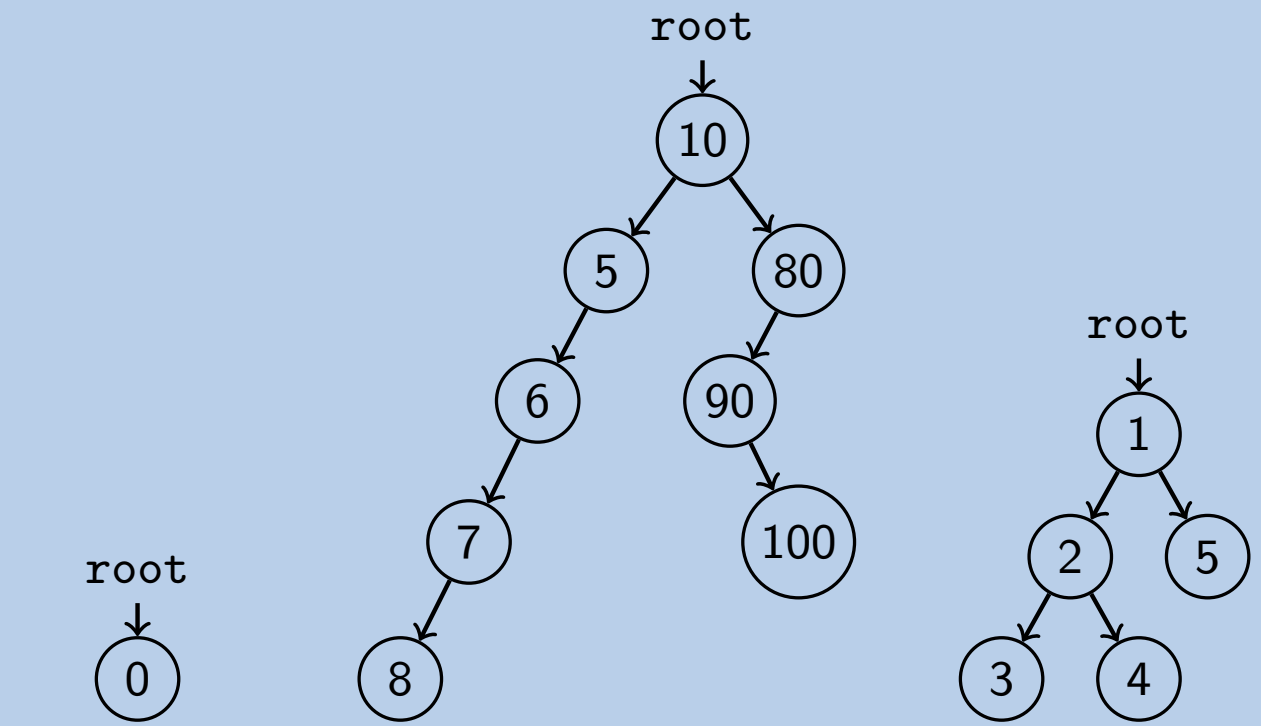
3 BST Methods

height

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

```
public int height()
```

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



`height` is 1 `height` is 5 `height` is 3

`height` Solution

```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)
      );
   }
}
```

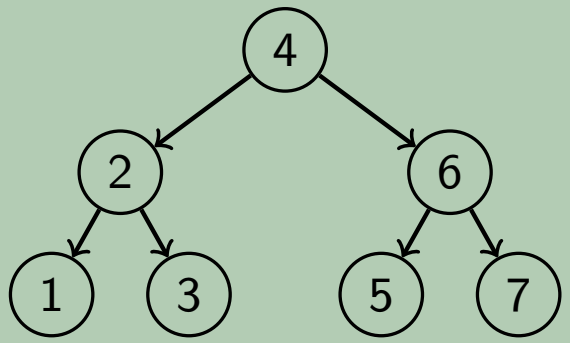
Back to contains

Recall contains()

```
1  private boolean contains(IntTreeNode current, int value) {
2     /* If the tree is null, it definitely doesn't contain value... */
3     if (current == null) { return false; }
4
5     /* If current *is* value, we found it! */
6     else if (current.data == value) { return true; }
7
8     else {
9        return contains(current.left, value) ||
10              contains(current.right, value);
11    }
12 }
```

Runtime of contains(7)

Consider the following tree:



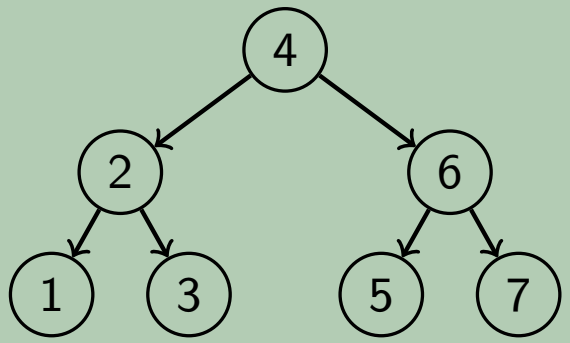
Back to contains

Recall contains()

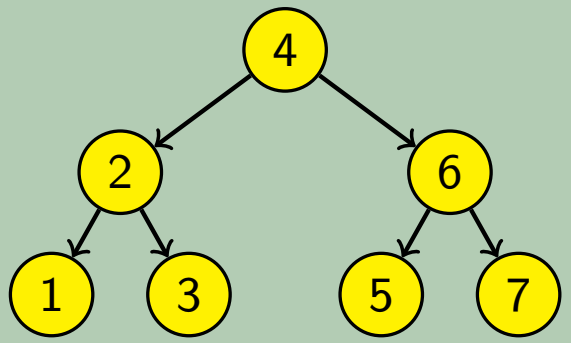
```
1  private boolean contains(IntTreeNode current, int value) {
2     /* If the tree is null, it definitely doesn't contain value... */
3     if (current == null) { return false; }
4
5     /* If current *is* value, we found it! */
6     else if (current.data == value) { return true; }
7
8     else {
9        return contains(current.left, value) ||
10              contains(current.right, value);
11    }
12 }
```

Runtime of contains(7)

Consider the following tree:



Which nodes do we visit for contains(**7**)

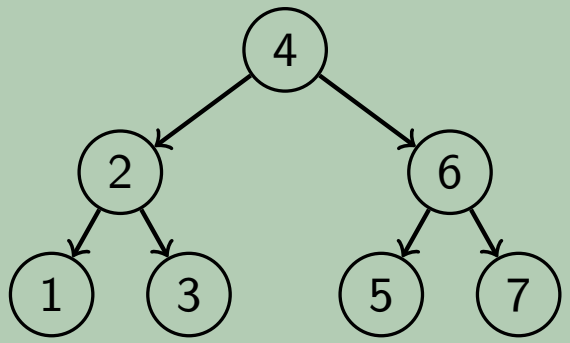


Recall contains()

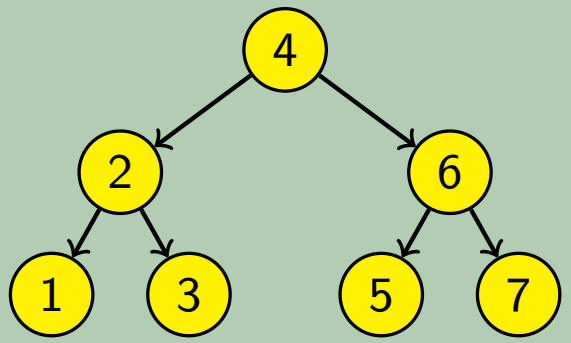
```
1  private boolean contains(IntTreeNode current, int value) {
2     /* If the tree is null, it definitely doesn't contain value... */
3     if (current == null) { return false; }
4
5     /* If current *is* value, we found it! */
6     else if (current.data == value) { return true; }
7
8     else {
9        return contains(current.left, value) ||
10              contains(current.right, value);
11    }
12 }
```

Runtime of contains(7)

Consider the following tree:



Which nodes do we visit for contains(**7**)



That makes the code $\mathcal{O}(n)$. Can we do better?

Doing Better!

In general, **we can't do better**. BUT, sometimes, we can!

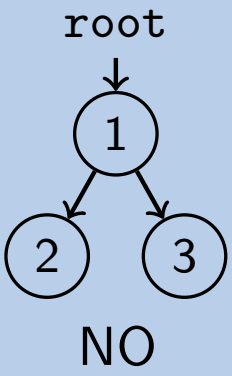
Definition (Binary **SEARCH** Tree (BST))

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

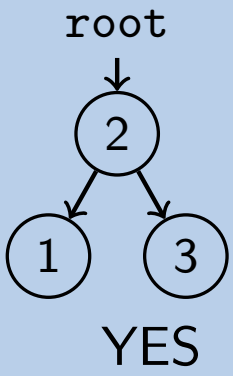
To put it another way, a binary tree is a **BST** when:

- 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

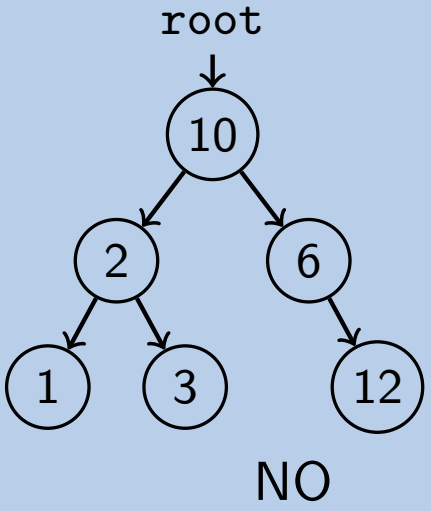
Example (Which of the following are BSTs?)



NO



YES



NO

isBST

Write a function isBST with the following signature:

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

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

```
1  private boolean isBST(IntTreeNode current, int min, int max) {
2     if (current == null) {
3        return true;
4     }
5     else if (current.data < min || current.data > max) {
6        return false;
7     }
8     else if (!isBST(current.left, min, current.data)) {
9        return false;
10    }
11    else {
12       return isBST(current.right, current.data, max);
13    }
14 }
```

Write contains() for a BST

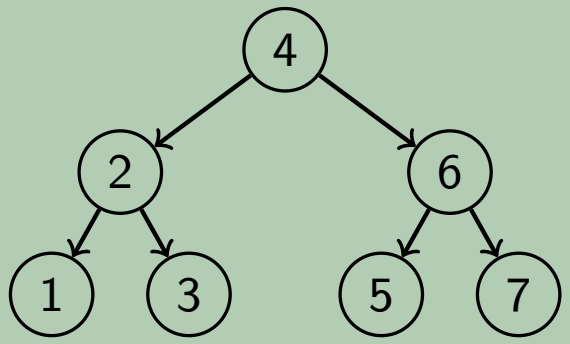
Fix contains so that it takes advantage of the BST properties.

Recall contains()

```
1  private boolean contains(IntTreeNode current, int value) {
2     /* If the tree is null, it definitely doesn't contain value... */
3     if (current == null) { return false; }
4
5     /* If current *is* value, we found it! */
6     else if (current.data == value) { return true; }
7
8     else if (current.data < value) {
9        return contains(current.right, value);
10    }
11    else {
12       return contains(current.left, value);
13    }
14 }
```

Runtime of (better) `contains(7)`

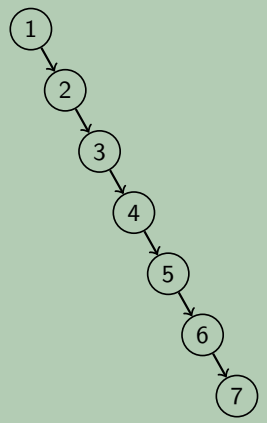
Consider the following tree:



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

WARNING!

Consider the following tree:

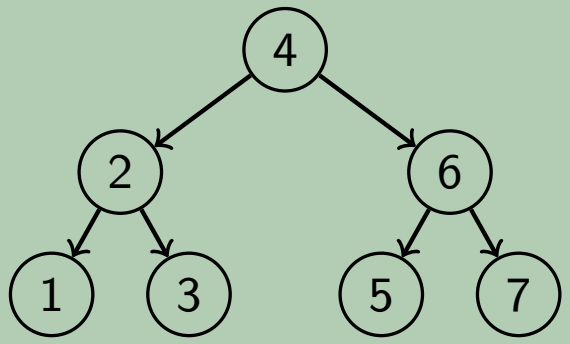


This is the same tree, but now **we have to visit all the nodes!**

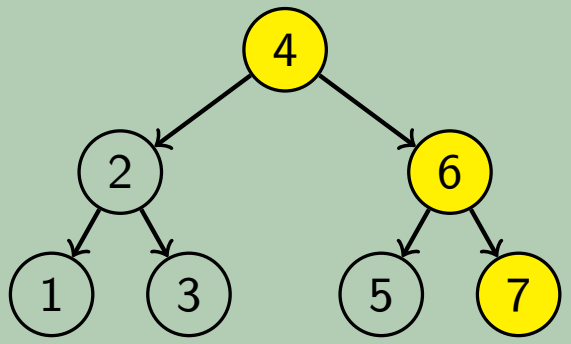
Tracing the new `contains`

Runtime of (better) `contains(7)`

Consider the following tree:



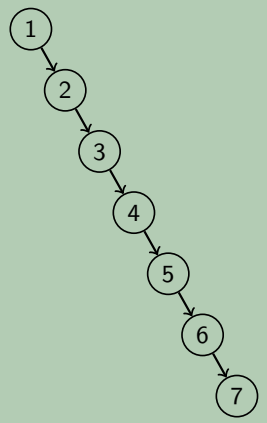
Which nodes do we visit for `contains(7)`



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!**

Adding to a BST!

`add`

Write a method `add` in the BST class with the following signature:

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

that preserves the BST property.

Example (`tree.add(49)`)

Adding to a BST (Attempt #1)

Attempt #1

```
1  public void add(int value) {
2     add(this.root, value);
3  }
4  private void add(IntTreeNode current, int value) {
5     if (current == null) {
6        current = new IntTreeNode(value);
7     }
8     else if (current.data > value) {
9        add(current.left, value);
10    }
11    else if (current.data < value) {
12       add(current.right, value);
13    }
14 }
```

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

```
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)

Fixed Attempt

```
1  public void add(int value) {
2     this.root = add(this.root, value);
3  }
4  private IntTreeNode add(IntTreeNode current, int value) {
5     if (current == null) {
6        current = new IntTreeNode(value);
7     }
8     else if (current.data > value) {
9        current.left = add(current.left, value);
10    }
11    else if (current.data < value) {
12       current.right = add(current.right, value);
13    }
14    return current;
15 }
```

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.