operations:
//returns true if x in tree, o/w returns false
boolean contains (object x, Node t)

Idea:
contains 3:

6
|    |
O 2 5
|  |
4 1

Start at root:
For each node, check element:
If = x, done! return true
If > x, recurse on left child
If < x, "right"

Base Case: when node is null, return false
contains (object x, Node t)?

if t == null
return false
else:
if t.element == x
return true
else if t.element > x
return contains(x, t.left)
else // t.element < x
return contains(x, t.right)
\text{contains} (4, \text{root}) = \text{contains} (4, (6)) \quad 6 > 4 \Rightarrow \text{left} \\
\quad \text{contains} (4, (2)) \quad 2 < 4 \Rightarrow \text{right} \\
\quad \text{contains} (4, (4)) \quad 4 = 4 \Rightarrow \text{return true} \\
\text{contains} (5, \text{root}) = \text{contains} (5, (6)) \quad 6 > 5 \Rightarrow \text{left} \\
\quad \text{contains} (5, (2)) \quad 2 < 5 \Rightarrow \text{right} \\
\quad \text{contains} (5, (4)) \quad 4 < 5 \Rightarrow \text{right} \\
\quad \text{contains} (5, \text{null}) \quad \text{t=null, return false} \\

\text{Run Time? For a tree with} \sqrt{n} \text{nodes} \\
\text{Worst-case: start at root, keep searching until we hit a leaf (traverse the depth of tree).} \\
\text{Worst-case depth?} \quad \text{Average case is log(n).} \\
\begin{align*}
\text{Top level has 1 node (root)} \\
\text{Each level has } \theta \text{ twice as many nodes as previous level} \\
\text{Bottom level has } \frac{n}{2} \text{ nodes (leaves)}
\end{align*}
\[ \text{depth} = \# \text{ levels} = \]
\[ \# \text{ times we can double from 1 until } \frac{n}{2} \]
\[ = O \left( \log \left( \frac{n}{2} \right) \right) = O \left( \log(n) \right) \]

\[ \therefore \text{ RunTime} = O(\text{depth}) = \text{ worst-case: } O(n) \]
\[ \quad \quad \text{ avg-case: } O(\log(n)) \]

Similar for most operations

Next operation: insert
Similar to contains

\[
\begin{array}{c}
\text{(6)} \\
\downarrow \\
\text{(2)} \quad \text{(8)} \\
\downarrow \quad \downarrow \\
\text{(1)} \quad \text{(4)} \\
\downarrow \\
\text{(3)} \quad \text{(5)}
\end{array}
\]

- ex: insert 5

- Where should it go?
  - Only one spot!

Idea: Search for element (contains()), if not in tree, insert.

Where? Child of last (non-null) node visited

will have another version of contains() that returns last non-null node visited in search for x in tree rooted at t

\[ \text{modified-contains}(x, t) \]
// One implementation approach

insert (object e, BinaryNode t) // insert element e in the with root t

// create new node
BinaryNode n = new BinaryNode (e)

// Get last visited non-null node in search for e
BinaryNode p = modifiedContains (e, t);  
  // pseudo-code for compareTo()

if (p.element == e) // add to left of p
  p.left = n
else if (p.element < e) // add to right of p
  p.right = n;
else // p.element == e (duplicate!)
  p.addtoduplicate (e);

3

2

1

(6) insert (5, 6) (4)

2

1

3

5

p = (4) \Rightarrow 4 < 5, right

What about duplicates? Not all every node has duplicate array implementations (add to BinaryNode class) do this!

Run Time: $O\text{(depth)} = O(n)$ worst $O(\log n)$ average
Keep in mind elements may not just be numbers.
We can store any objects.

Now that we're considering a data structure that orders the elements, it's unclear how to order objects.

ex. BST of BankAccounts, CalendarDate, City, Message

Each of these objects have many variables.

Make an object comparable:

1. assign one data field (or a function of data fields) as a key for comparison (your choice)
2. write a compareTo method using key (Java-recognized method like toString())
3. make class implement Comparable

ex. BankAccount key: balance

```java
public int compareTo(BankAccount other) {
    if (balance == other.balance)
        return 0;
    else if (balance < other.balance)
        return -1;
    else // balance > other.balance
        return 1;
}
```
public class BankAccount implements Comparable<BankAccount>

Sample Code: BankAccount, CalendarDate, Message