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

Idea:
contains 3?

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, BinaryNode 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)
ex1 contains (4, root) = contains (4, 6) 6 > 4 \rightarrow \text{left} \downarrow
contains (4, 2) 2 < 4 \rightarrow \text{right} \downarrow
contains (4, 4) 4 = 4 \checkmark \text{return true}

ex2 contains (5, 6) 6 > 5 \rightarrow \text{left} \downarrow
contains (5, 2) 2 < 5 \rightarrow \text{right}

contains (5, 4) 4 < 5 \rightarrow \text{right}

contains (5, \text{null}) t=\text{null}, \text{return false}

Run Time? For a tree with \( \sqrt{n} \) nodes

Worst-case: start at root, keep searching until we hit a leaf (traverse the depth of tree)

Worst-case depth? Average Case is \( \log(n) \)

\( \begin{align*}
\text{Top level has 1 node (root)} \\
\text{Each level has } \approx \text{ twice as many nodes as previous level} \\
\text{Bottom level has } \approx \frac{n}{2} \text{ nodes (leaves)}
\end{align*} \)
\[
\text{depth} = \# \text{ levels} = \\
= \left\lceil \frac{\log(n)}{2} \right\rceil = O\left(\log\left(\frac{n}{2}\right)\right) = O\left(\log(n)\right)
\]

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

Similar for most operations

Next operation: insert
Similar to contains

```
(6)  
(2)  (8)  
(1) (4)  
(3) (5)
```

\(\text{ex: insert } 5\)
\(\text{Where should it go?}\)
\(\text{Only one spot!}\)

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 tree 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);
  //pseudocode 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 !)

\[ \text{insert } (5, 6) \]

\( \text{p = (4) } \Rightarrow 4 < 5, \text{ right} \)

What about duplicates?

Not all 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 elements, it's unclear how to order objects.

ex: BST of BankAccounts, Calendar Date, City, Message

Each of these objects has many data fields.

BankAccounts

\[ \text{ID: } \text{key} \]
\[ \text{name} \]
\[ \text{balance} \]

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

```
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
BST

remove Object x. //remove the node containing x

3 Cases:
- In all 3, need to:
  - Search for node containing x, call it n
  - Keep track of n's parent

- case 1: (simplest) n is a leaf:
  
  \[
  \begin{array}{c}
  2 \\
  8 \\
  1 \\
  4 \\
  3 \\
  \end{array}
  \]

  ex: remove 3?

  \( \Rightarrow \) Set n's parent's left or right ptr to null

remove (Object x, BinaryNode t)
//get the node containing x
1. \( n = \text{get}(x, t) \)

// if n is a leaf:
2. \( \text{if } (n.\text{left} == n.\text{right} == \text{null}) \)

//get last non-null node visited in search for x
3. \( p = \text{modified-contains}(x, t) \)

// set n to null
4. \( \text{if } (p.\text{right} == n), p.\text{right} = \text{null} \)
5. \( \text{else } \{(p.\text{left} == n), n.\text{left} = \text{null} \} \)
(2) \( n \) has one child:

- remove 4

\[
\begin{array}{c}
\text{(6)} \\
\text{(2)} & \text{(8)} \\
\text{(1)} & \text{(4)} \\
\end{array} = \begin{array}{c}
\text{(6)} \\
\text{(2)} & \text{(8)} \\
\text{(1)} & \text{(3)} \\
\end{array}
\]

Make (4)'s parent point to (4)'s child

\[(n \text{ is node containing } x) >
\text{Make } n's \text{ parent's left/right ptr. point to } n's \text{ child}
\]

(3) \( n \) has 2 children:

\[\text{new example}\]

- remove 2?

\[
\begin{array}{c}
\text{(6)} \\
\text{(2)} & \text{(10)} \\
\text{(1)} & \text{(5)} & \text{(8)} & \text{(11)} \\
\text{(3)} & \text{(5)} \\
\end{array}
\]

First some terminology

- predecessor of \( n \) - highest-valued node in left subtree of \( n \)

\[\text{ex: } \text{pred(6)} = 5\]

\[\text{pred(2)} = 1\]
- successor of \( n \): lowest-valued node in right subtree of \( n \)

\[
\text{ex: } \text{successor}(6) = 8
\]

\[
\text{successor}(3) = 3
\]

Can replace \( n \) with either (we will use predecessor)

Let \( y = n \)'s predecessor

(Main Idea: Replace \( n \) with \( y \\
Reorder using fewest # of pointer changes)

2 sub-cases

- Case 3(a): \( y \)'s parent = \( n \)

\[
\begin{array}{c}
\text{remove } (2) \ (n=2) \\
\text{Replace } n \text{ with } y
\end{array}
\]
Case 3(b): Y's parent ≠ n.

\[ \begin{array}{c}
2 \\
\downarrow
\end{array} \quad \begin{array}{c}
10 \\
\downarrow
\end{array} \quad \begin{array}{c}
8 \\
\downarrow
\end{array} \quad \begin{array}{c}
11 \\
\downarrow
\end{array} \quad \begin{array}{c}
6 \\
\downarrow
\end{array}
\]

Y = 5

\[ \begin{array}{c}
2 \\
\downarrow
\end{array} \quad \begin{array}{c}
10 \\
\downarrow
\end{array} \quad \begin{array}{c}
8 \\
\downarrow
\end{array} \quad \begin{array}{c}
11 \\
\downarrow
\end{array} \quad \begin{array}{c}
6 \\
\downarrow
\end{array}
\]

remove (6) n = (6)

1. Replace n with Y.
2. Make Y's old parent point to Y's (only) child

(Note if Y had another child, this child would be the pred of n)

\[ \begin{array}{c}
6 \\
\downarrow
\end{array} \quad \begin{array}{c}
2 \\
\downarrow
\end{array} \quad \begin{array}{c}
1 \\
\downarrow
\end{array} \\
\begin{array}{c}
0 \\
\end{array} \quad \begin{array}{c}
3 \\
\end{array} \quad \begin{array}{c}
5.5 \\
\rightarrow \end{array}
\]

\[ \begin{array}{c}
5.5 \\
\downarrow
\end{array} \quad \begin{array}{c}
2 \\
\downarrow
\end{array} \quad \begin{array}{c}
1 \\
\downarrow
\end{array} \\
\begin{array}{c}
0 \\
\end{array} \quad \begin{array}{c}
3 \\
\end{array} \quad \begin{array}{c}
5 \\
\end{array}
\]

(Anything else need to change?)

(root should now point to 6)

In all cases, check:

1. root = n
2. root = Y.
BST remove (more examples)

ex (1)  
```
   17
  /    
11     24
 /     /  
 9     14
 /     /
13     19
```

\( n = y \)'s parent

\( y = 19 \)

```
   11
  /    
 9     14
 /     
13     19
```

\( n \) replace \( n \) w/ \( y \)

\( n = 20 \)

remove (20)

\( y = 14 \)

```
   17
  /    
 11     24
 /     /  
 9     14
 /     
13     19
```

\( n \)'s parent

\( n \) = 14

\( 14 \)

```
   11
  /    
 9     24
```

Step 1: replace \( n \) with \( y \)

\( n \) = 13

```
   14
  /    
 9     13
```

\( y \) = 17

```
   11
  /    
 9     24
```

Step 2: Make \( y \)'s old parent point to \( y \)'s child.