Building Java Programs Chapter 17

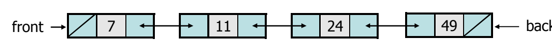
Binary Trees

Creative use of arrays/links

- Some data structures (such as hash tables and binary trees) are built around clever ways of using arrays and/or linked lists.
  - What array order can help us find values quickly later?

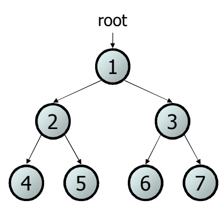
index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

  - What if linked list nodes each had more than one link?



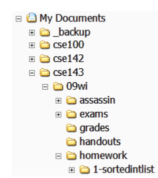
Trees

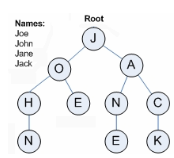
- **tree**: A directed, acyclic structure of linked nodes.
  - *directed* : Has one-way links between nodes.
  - *acyclic* : No path wraps back around to the same node twice.
  - **binary tree**: One where each node has at most two children.
- A tree can be defined as either:
  - empty (`null`), or
  - a **root** node that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.
      - (The left and/or right subtree could be empty.)

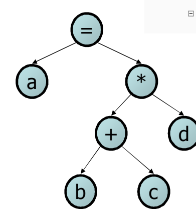


Trees in computer science

- folders/files on a computer
- family genealogy; organizational charts
- AI: decision trees
- compilers: parse tree
  - a = (b + c) * d;
- cell phone T9





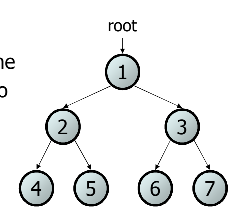


Programming with trees

- Trees are a mixture of linked lists and recursion
  - considered very elegant (perhaps beautiful!) by CSE nerds
  - difficult for novices to master
- Common student remark #1:
  - "My code doesn't work, and I don't know why."
- Common student remark #2:
  - "My code works, and I don't know why."

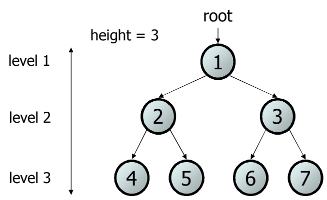
Terminology

- **node**: an object containing a data value and left/right children
- **root**: topmost node of a tree
- **leaf**: a node that has no children
- **branch**: any internal node; neither the root nor a leaf
- **parent**: a node that refers to this one
- **child**: a node that this node refers to
- **sibling**: a node with a common



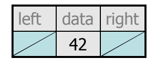
Terminology 2

- **subtree**: the tree of nodes reachable to the left/right from the current node
- **height**: length of the longest path from the root to any node
- **level** or **depth**: length of the path from a root to a given node
- **full tree**: one where every branch has 2 children

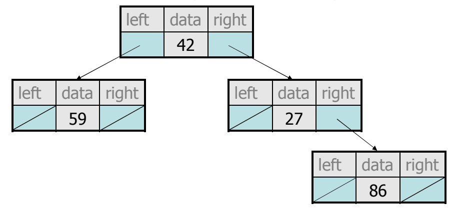


A tree node for integers

- A basic **tree node object** stores data and refers to left/right



- Multiple nodes can be linked together into a larger tree



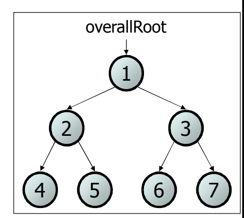
`IntTreeNode` class

```
// An IntTreeNode object is one node in a binary tree of ints.
public class IntTreeNode {
    public int data;            // data stored at this node
    public IntTreeNode left;    // reference to left subtree
    public IntTreeNode right;   // reference to right subtree

    // Constructs a leaf node with the given data.
    public IntTreeNode(int data) {
        this(data, null, null);
    }

    // Constructs a branch node with the given data and links.
    public IntTreeNode(int data, IntTreeNode left,
                                 IntTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```


`IntTree` class

```
// An IntTree object represents an entire binary tree of ints.
public class IntTree {
    private IntTreeNode overallRoot;   // null for an empty tree

    methods
}
```

- Client code talks to the `IntTree`, not to the node objects inside it
- Methods of the `IntTree` create and manipulate the nodes, their data and links between them



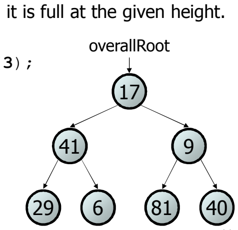
`IntTree` constructor

- Assume we have the following constructors:

```
public IntTree(IntTreeNode overallRoot)
public IntTree(int height)
```

  - The 2nd constructor will create a tree and fill it with nodes with random data values from 1-100 until it is full at the given height.

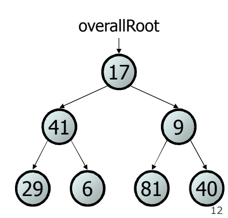
```
IntTree tree = new IntTree(3);
```



Exercise

- Add a method `print` to the `IntTree` class that prints the elements of the tree, separated by spaces.
  - A node's left subtree should be printed before it, and its right subtree should be printed after it.
  - Example: `tree.print();`

```
29 41 6 17 81 9 40
```



Exercise solution

```
// An IntTree object represents an entire binary tree of ints.
public class IntTree {
    private IntTreeNode overallRoot;    // null for an empty tree
    ...

    public void print() {
        print(overallRoot);
        System.out.println();    // end the line of output
    }

    private void print(IntTreeNode root) {
        // (base case is implicitly to do nothing on null)
        if (root != null) {
            // recursive case: print left, center, right
            print(overallRoot.left);
            System.out.print(overallRoot.data + " ");
            print(overallRoot.right);
        }
    }
}
```

13

Template for tree methods

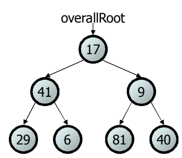
```
public class IntTree {
    private IntTreeNode overallRoot;
    ...

    public type name(parameters) {
        name(overallRoot, parameters);
    }

    private type name(IntTreeNode root, parameters) {
        ...
    }
}
```

- Tree methods are often implemented recursively
  - with a public/private pair
  - the private version accepts the root node to process

14

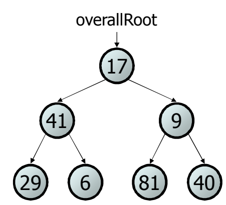
Traversals

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods
- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node



15

Traversal example

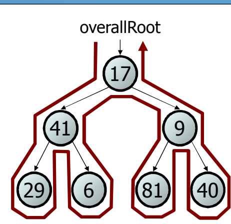


- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

16

Traversal trick

- To quickly generate a traversal:
  - Trace a path around the tree.
  - As you pass a node on the proper side, process it.
    - pre-order: left side
    - in-order: bottom
    - post-order: right side

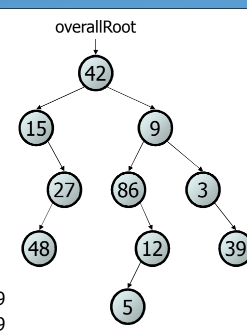


- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

17

Exercise

- Give pre-, in-, and post-order traversals for the following tree:



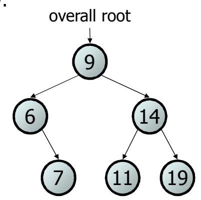
  - pre: 42 15 27 48 9 86 12 5 3 39
  - in: 15 48 27 42 86 5 12 9 3 39
  - post: 48 27 15 5 12 86 39 3 42

18

Exercise

- Add a method named `printSideways` to the `IntTree` class that prints the tree in a sideways indented format, with right nodes above roots above left nodes, with each level 4 spaces more indented than the one above it.
  - Example: Output from the tree below:

```
        19
    14
        11
9
        7
    6
```



19

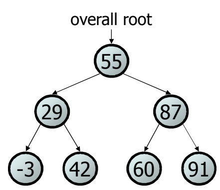
Exercise solution

```
// Prints the tree in a sideways indented format.
public void printSideways() {
    printSideways(overallRoot, "");
}

private void printSideways(IntTreeNode root,
                           String indent) {
    if (root != null) {
        printSideways(root.right, indent + "    ");
        System.out.println(indent + root.data);
        printSideways(root.left, indent + "    ");
    }
}
```

20

Binary search trees

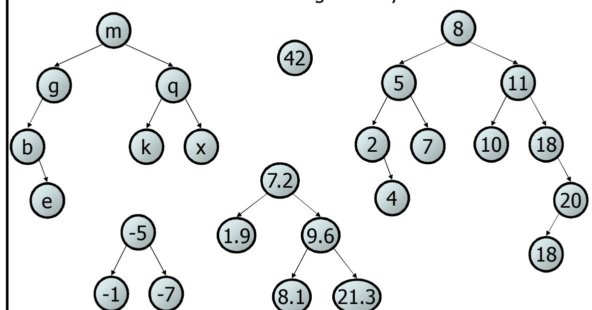
- **binary search tree** ("BST"): a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.
- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



21

Exercise

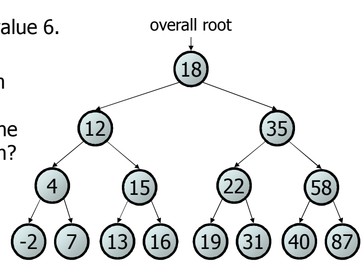
- Which of the trees shown are legal binary search trees?



22

Searching a BST

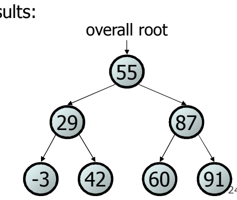
- Describe an algorithm for searching the tree below for the value 31.
- Then search for the value 6.
- What is the maximum number of nodes you would need to examine to perform any search?



23

Exercise

- Convert the `IntTree` class into a `SearchTree` class.
  - The elements of the tree will constitute a legal binary search tree.
- Add a method `contains` to the `SearchTree` class that searches the tree for a given integer, returning `true` if found.
  - If a `SearchTree` variable `tree` referred to the tree below, the following calls would have these results:
    - `tree.contains(29)` → `true`
    - `tree.contains(55)` → `true`
    - `tree.contains(63)` → `false`
    - `tree.contains(35)` → `false`



24

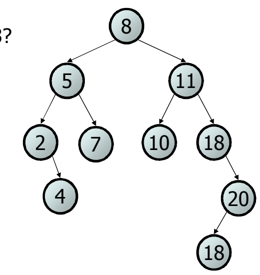
Exercise solution

```
// Returns whether this tree contains the given integer.
public boolean contains(int value) {
    return contains(overallRoot, value);
}

private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (root.data == value) {
        return true;
    } else if (root.data > value) {
        return contains(root.left, value);
    } else {    // root.data < value
        return contains(root.right, value);
    }
}
```

25

Adding to a BST

- Suppose we want to add the value 14 to the BST below.
  - Where should the new node be added?
- Where would we add the value 3?
- Where would we add 7?
- If the tree is empty, where should a new value be added?
- What is the general algorithm?

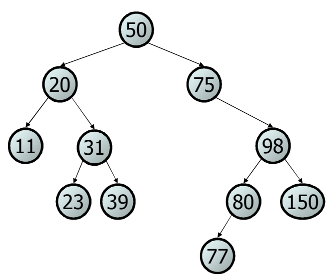


26

Adding exercise

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

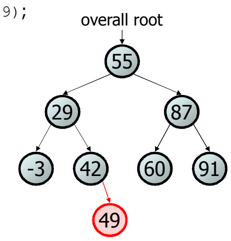
50
20
75
98
80
31
150
39
23
11
77



27

Exercise

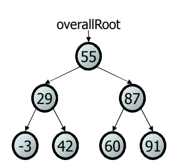
- Add a method `add` to the `SearchTree` class that adds a given integer value to the tree. Assume that the elements of the `SearchTree` constitute a legal binary search tree, and add the new value in the appropriate place to maintain ordering.
  - `tree.add(49);`



28

An incorrect solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    add(overallRoot, value);
}

private void add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (root.data > value) {
        add(root.left, value);
    } else if (root.data < value) {
        add(root.right, value);
    }
    // else root.data == value;
    // a duplicate (don't add)
}
```

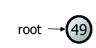


- Why doesn't this solution work?

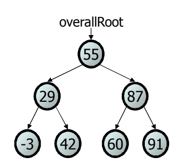
29

The problem

- Much like with linked lists, if we just modify what a local variable refers to, it won't change the collection.




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



  - In the linked list case, how did we actually modify the list?
    - by changing the `front`
    - by changing a node's `next` field

30

A poor correct solution

```
// Adds the given value to this BST in sorted order. (bad style)
public void add(int value) {
    if (overallRoot == null) {
        overallRoot = new IntTreeNode(value);
    } else if (overallRoot.data > value) {
        add(overallRoot.left, value);
    } else if (overallRoot.data < value) {
        add(overallRoot.right, value);
    }
    // else overallRoot.data == value; a duplicate (don't add)
}

private void add(IntTreeNode root, int value) {
    if (root.data > value) {
        if (root.left == null) {
            root.left = new IntTreeNode(value);
        } else {
            add(overallRoot.left, value);
        }
    } else if (root.data < value) {
        if (root.right == null) {
            root.right = new IntTreeNode(value);
        } else {
            add(overallRoot.right, value);
        }
    }
    // else root.data == value; a duplicate (don't add)
}
```

31

x = change(x);

- String methods that modify a string actually return a new one.
  - If we want to modify a string variable, we must re-assign it.

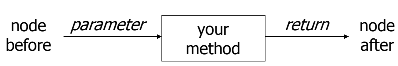
```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);    // lil bow wow
s = s.toUpperCase();
System.out.println(s);    // LIL BOW WOW
```

  - We call this general algorithmic pattern **x = change(x);**
  - We will use this approach when writing methods that modify the structure of a binary tree.

32

Applying x = change(x)

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node



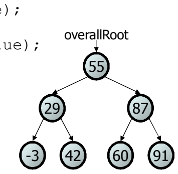
- In order to actually change the tree, you must reassign:

```
root = change(root, parameters);
root.left = change(root.left, parameters);
root.right = change(root.right, parameters);
```

33

A correct solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    overallRoot = add(overallRoot, value);
}

private IntTreeNode add(IntTreeNode root, int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else if (root.data > value) {
        root.left = add(root.left, value);
    } else if (root.data < value) {
        root.right = add(root.right, value);
    } // else a duplicate

    return root;
}
```

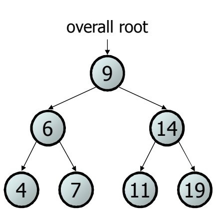


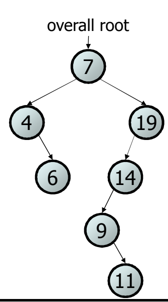
- Think about the case when `root` is a leaf...

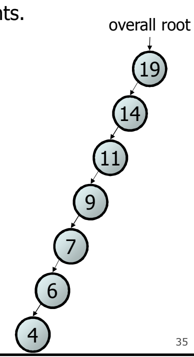
34

Searching BSTs

- The BSTs below contain the same elements.
  - What orders are "better" for searching?





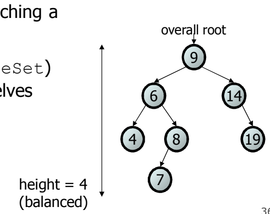


35

Trees and balance

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of N nodes has a height of ~ $\log_2$ N.
  - A very unbalanced tree can have a height close to N.

  - The runtime of adding to / searching a BST is closely related to height.
  - Some tree collections (e.g. `TreeSet`) contain code to balance themselves as new nodes are added.

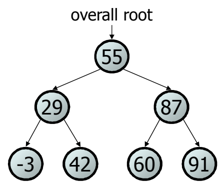


36

Exercise

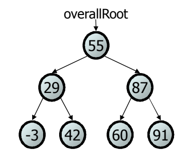
- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

```
int min = tree.getMin();   // -3
```



Exercise solution

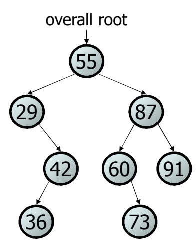
```
// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public int getMin() {
    if (overallRoot == null) {
        throw new NoSuchElementException();
    }
    return getMin(overallRoot);
}

private int getMin(IntTreeNode root) {
    if (root.left == null) {
        return root.data;
    } else {
        return getMin(root.left);
    }
}
```



Exercise

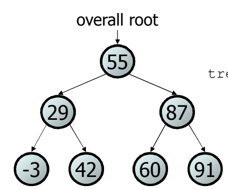
- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Assume that the elements of the `IntTree` constitute a legal binary search tree, and remove the value in such a way as to maintain ordering.

  - `tree.remove(73);`
  - `tree.remove(29);`
  - `tree.remove(87);`
  - `tree.remove(55);`

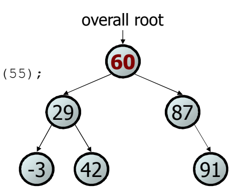


Cases for removal

- Possible states for the node to be removed:
  - a leaf: replace with null
  - a node with a left child only: replace with left child
  - a node with a right child only: replace with right child
  - a node with both children: replace with min value from right



`tree.remove(55);`



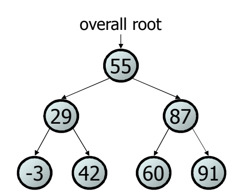
Exercise solution

```
// Removes the given value from this BST, if it exists.
public void remove(int value) {
    overallRoot = remove(overallRoot, value);
}

private IntTreeNode remove(IntTreeNode root, int value) {
    if (root == null) {
        return null;
    } else if (root.data > value) {
        root.left = remove(root.left, value);
    } else if (root.data < value) {
        root.right = remove(root.right, value);
    } else {  // root.data == value; remove this node
        if (root.right == null) {
            return root.left;     // no R child; replace w/ L
        } else if (root.left == null) {
            return root.right;    // no L child; replace w/ R
        } else {
            // both children; replace w/ min from R
            root.data = getMin(root.right);
            root.right = remove(root.right, root.data);
        }
    }
    return root;
}
```


Binary search trees

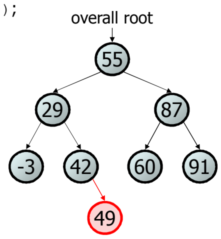
- **binary search tree** ("BST"): a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



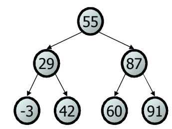
Exercise

- Add a method `add` to the `IntTree` class that adds a given integer value to the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree, and add the new value in the appropriate place to maintain ordering.
  - `tree.add(49);`



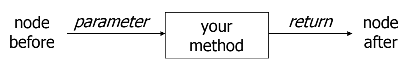
The incorrect solution

```
public class SearchTree {
 private IntTreeNode overallRoot;

 // Adds the given value to this BST in sorted order.
 // (THIS CODE DOES NOT WORK PROPERLY!)
 public void add(int value) {
     add(overallRoot, value);
 }

 private void add(IntTreeNode node, int value) {
     if (node == null) {
         node = new IntTreeNode(value);
     } else if (value < node.data) {
         add(node.left, value);
     } else if (value > node.data) {
         add(node.right, value);
     }
     // else a duplicate (don't add)
 }
```



Applying x = change(x)

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node

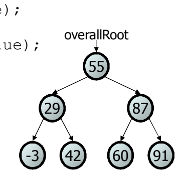


- In order to actually change the tree, you must reassign:

```
overallRoot = change(overallRoot, ...);
node = change(node, ...);
node.left = change(node.left, ...);
node.right = change(node.right, ...);
```


A correct solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    overallRoot = add(overallRoot, value);
}

private IntTreeNode add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (value < node.data) {
        node.left = add(node.left, value);
    } else if (value > node.data) {
        node.right = add(node.right, value);
    } // else a duplicate

    return node;
}
```

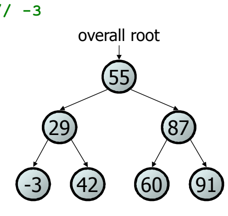


- Think about the case when `root` is a leaf...

Exercise

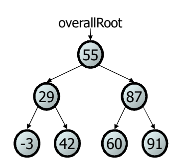
- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

```
int min = tree.getMin();   // -3
```



Exercise solution

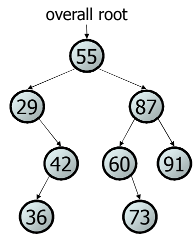
```
// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public int getMin() {
    if (overallRoot == null) {
        throw new NoSuchElementException();
    }
    return getMin(overallRoot);
}

private int getMin(IntTreeNode root) {
    if (root.left == null) {
        return root.data;
    } else {
        return getMin(root.left);
    }
}
```



Exercise

- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Assume that the elements of the `IntTree` constitute a legal binary search tree, and remove the value in such a way as to maintain ordering.

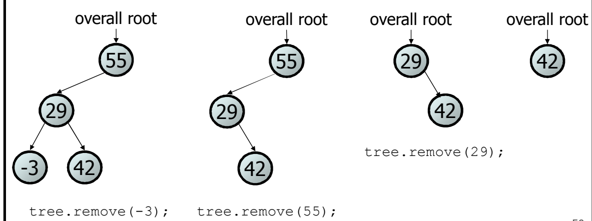
  - `tree.remove(73);`
  - `tree.remove(29);`
  - `tree.remove(87);`
  - `tree.remove(55);`



49

Cases for removal 1

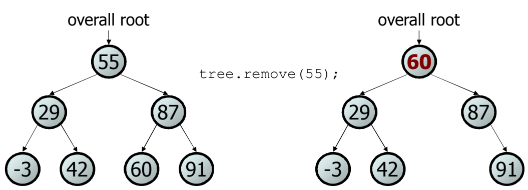
1. a leaf: replace with null
2. a node with a left child only: replace with left child
3. a node with a right child only: replace with right child



50

Cases for removal 2

4. a node with **both** children: replace with **min from right**



51

Exercise solution

```
// Removes the given value from this BST, if it exists.
public void remove(int value) {
    overallRoot = remove(overallRoot, value);
}

private IntTreeNode remove(IntTreeNode root, int value) {
    if (root == null) {
        return null;
    } else if (root.data > value) {
        root.left = remove(root.left, value);
    } else if (root.data < value) {
        root.right = remove(root.right, value);
    } else {  // root.data == value; remove this node
        if (root.right == null) {
            return root.left;    // no R child; replace w/ L
        } else if (root.left == null) {
            return root.right;   // no L child; replace w/ R
        } else {
            // both children; replace w/ min from R
            root.data = getMin(root.right);
            root.right = remove(root.right, root.data);
        }
    }
    return root;
}
```

52