Building Java Programs

Binary Search Trees

reading: 17.3 – 17.4
What is the output of this program?

```java
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change1(p);
    System.out.println(p);
    change2(p);
    System.out.println(p);
}

public static void change1(Point p) {
    p.x = 14;
}

public static void change2(Point p) {
    p = new Point(7, 8);
}
```
contains

```java
private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (root.data == value) {
        return true;
    } else {
        return contains(root.left, value)
            || contains(root.right, value);
    }
}
```
Case study: contains w/ arrays

- What is the Big-O efficiency to see if a value is contained in an unsorted array?

| -3 | 87 | 42 | 55 | 91 | 29 | 60 |

- What about if the array is sorted?

| -3 | 29 | 42 | 55 | 60 | 87 | 91 |
Binary search trees

- **binary search tree** ("BST"): a binary tree where each non-empty node R has the following properties:
  - elements of R's left subtree contain data "less than" R's data,
  - elements of R's right subtree contain 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.
Which of the trees shown are legal binary search trees?
Searching a BST

• Describe an algorithm for searching a binary search tree.
  • Try searching for the value 31, then 6.

• What is the maximum number of nodes you would need to examine to perform any search?
Exercise

- Convert the `IntTree` class into a `SearchTree` class.
  - The elements of the tree will form a legal binary search tree.

- Write a `contains` method that takes advantage of the BST structure.
  - `tree.contains(29) → true`
  - `tree.contains(55) → true`
  - `tree.contains(63) → false`
  - `tree.contains(35) → false`
Exercise solution

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

private boolean contains(IntTreeNode node, int value) {
    if (node == null) {
        return false; // base case: not found here
    } else if (node.data == value) {
        return true; // base case: found here
    } else if (node.data > value) {
        return contains(node.left, value);
    } else {
        // root.data < value
        return contains(node.right, value);
    }
}
Adding to a BST

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

- What is the general algorithm?
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
50
  20
   11
   31
  23  39
  80  150
  77
```
Exercise

- Add a method `add` to the `SearchTree` class that adds a given integer value to the BST.
  - Add the new value in the proper place to maintain BST ordering.
    - `tree.add(49);`
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 node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (node.data > value) {
        add(node.left, value);
    } else if (node.data < value) {
        add(node.right, value);
    } else node.data == value, so // it's a duplicate (don't add)
}

• Why doesn't this solution work?
The $x = \operatorname{change}(x)$ pattern

read 17.3
A tangent: Change a point

What is the state of the object referred to by \( p \) after this code?

```java
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static void change(Point thePoint) {
    thePoint.x = 3;
    thePoint.y = 4;
}

// answer: (3, 4)
```
What is the state of the object referred to by \( p \) after this code?

```java
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static void change(Point thePoint) {
    thePoint = new Point(3, 4);
}

// answer: (1, 2)
```
Changing references

• If a method *dereferences a variable* (with .) and modifies the object it refers to, that change will be seen by the caller.

```java
public static void change(Point thePoint) {
    thePoint.x = 3;       // affects p
    thePoint.setY(4);     // affects p
}
```

• If a method *reassigns a variable to refer to a new object*, that change will *not* affect the variable passed in by the caller.

```java
public static void change(Point thePoint) {
    thePoint = new Point(3, 4);       // p unchanged
    thePoint = null;                   // p unchanged
}
```

• What if we want to make the variable passed in become `null`?
What is the state of the object referred to by $p$ after this code?

```java
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static Point change(Point thePoint) {
    thePoint = new Point(3, 4);
    return thePoint;
}

// answer: (1, 2)
```

![Diagram showing the change in point coordinates](image)
What is the state of the object referred to by \( p \) after this code?

```java
public static void main(String[] args) {
    Point p = new Point(1, 2);
    p = change(p);
    System.out.println(p);
}

public static Point change(Point thePoint) {
    thePoint = new Point(3, 4);
    return thePoint;
}

// answer: (3, 4)
```
x = change(x);

- 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

    p = change(p); // in main

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

- We call this general algorithmic pattern **x = change(x)**;
  - also seen with strings: **s = s.toUpperCase();**
The problem

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

```java
private void add(IntTreeNode node, int value) {
    if (node == null) {
        node = 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
Applying \( x = \text{change}(x) \)

- 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:

  ```
  node = \text{change}(node, \text{parameters});
  node.left = \text{change}(node.left, \text{parameters});
  node.right = \text{change}(node.right, \text{parameters});
  overallRoot = \text{change}(overallRoot, \text{parameters});
  ```
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 (node.data > value) {
        node.left = add(node.left, value);
    } else if (node.data < value) {
        node.right = add(node.right, value);
    } // else a duplicate; do nothing
    return node;
}

- What happens when node is a leaf?