CSE 143
Lecture 20

Binary Search Trees

read 17.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/
**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

• Which of the trees shown are legal binary search trees?
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?
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
// Returns whether this tree 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;
    } else if (node.data == value) {
        return true;
    } 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 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?
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:
• 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.

```python
• tree.add(49);
```

![Binary search tree diagram]
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 { // else node.data == value, so
        // it's a duplicate (don't add)
    }

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

read 17.3
A tangent: Change a point

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

```
public static void main(String[] args) {
    Point p = new Point(3, 25);
    change(p);
    System.out.println(p);
}

public static void change(Point thePoint) {
    thePoint.x = 99;
    thePoint.y = -1;
}

// answer: (99, -1)
```
• 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(3, 25);
    change(p);
    System.out.println(p);
}
```
```
public static void change(Point thePoint) {
    thePoint = new Point(99, -1);
}
```

// answer: (3, 25)
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 = 99; // affects p
    thePoint.setY(-12345); // 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(99, -1); // p unchanged
    thePoint = null; // p unchanged
}
```
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(3, 25);
    change(p);
    System.out.println(p);
}
```

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

// answer: (3, 25)
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(3, 25);
    p = change(p);
    System.out.println(p);
}

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

// answer: (99, -1)
```
• 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

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

• We call this general algorithmic pattern 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.

```java
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 use `x = change(x)` in methods that modify a binary tree.
  – We will **pass** in a node as a parameter and **return** a node result.
  – The node passed in must be **re-assigned** via `x = change(x)`.
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:

```plaintext
node = \text{change}(\text{node}, \text{parameters});
node.left = \text{change}(\text{node.left}, \text{parameters});
node.right = \text{change}(\text{node.right}, \text{parameters});
overallRoot = \text{change}(\text{overallRoot}, \text{parameters});
```
// 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
    return node;
}

• Think about the case when node 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.

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

![Binary search tree diagram]
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);
    }
}