CSE 143
Lecture 20

Binary Search Trees

read 17.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/
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 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);
    }
}
• 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:

```
50
20
75
98
80
31
150
39
23
11
77
```
• 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.

```java
tree.add(49);
```

![Diagram of a binary search tree with nodes labeled 55, 29, 87, -3, 42, 60, 91, and 49. The root is 55, with 29 and 87 as its children, and 49 is added as a new node.]
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?
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 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
// 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)
}
• 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 call this general algorithmic pattern \( x = \text{change}(x); \)
– We will use this approach when writing methods that modify the structure of a binary tree.
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:

  \[
  \begin{align*}
  \text{root} &= \text{change}(\text{root}, \ \text{parameters}); \\
  \text{root.left} &= \text{change}(\text{root.left}, \ \text{parameters}); \\
  \text{root.right} &= \text{change}(\text{root.right}, \ \text{parameters});
  \end{align*}
  \]
// 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...
Searching BSTs

- The BSTs below contain the same elements.
  - What orders are "better" for searching?
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 $\sim \log_2 N$.
  - 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.
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
```
// 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.

```java
• 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);
```
// 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;
}