CSE 143
Lecture 20

Binary Search Trees continued; Tree Sets

read 17.3 - 17.5

slides created by Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/
Recall: \( x = \text{change}(x) \)

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

  \[
  \text{node before} \quad \text{parameter} \quad \text{your method} \quad \text{return} \quad \text{node after}
  \]

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

  \[
  \begin{align*}
  \text{node} & \quad = \quad \text{change}(\text{node, parameters}) ; \\
  \text{node.left} & \quad = \quad \text{change}(\text{node.left, parameters}) ; \\
  \text{node.right} & \quad = \quad \text{change}(\text{node.right, parameters}) ; \\
  \text{overallRoot} & \quad = \quad \text{change}(\text{overallRoot, parameters}) ;
  \end{align*}
  \]
Add method

// 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;
}
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. Remove the value in such a way as to maintain BST ordering.

```java
• tree.remove(73);
• tree.remove(29);
• tree.remove(87);
• tree.remove(55);
```
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

```
overall root

29

55

-3 42

tree.remove(-3);  tree.remove(55);

overall root

29

55

42

tree.remove(55);

tree.remove(29);

overall root

29

42

overall root

42
```
Cases for removal 2

4. a node with **both** children: replace with **min from right**
   • (replacing with max from left would also work)

overall root

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

tree.remove(55);

overall root

```
60
  29     87
   -3    42   91
```

// 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;
}
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.

Diagram:
- Overall root
- Height = 4 (balanced)
Implementing a Tree Set

read 17.4 - 17.5
A tree set

- **Our SearchTree class is essentially a set.**
  - **operations:** add, remove, contains, size, isEmpty
  - **similar to the TreeSet class in java.util**

- **Let's actually turn it into a full set implementation.**
  - **step 1:** create ADT interface; implement it
  - **step 2:** get rid of separate node class file
  - **step 3:** make tree capable of storing any type of data (not just int)
• **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  – Describes *what* a collection does, not *how* it does it.

• Java's collection framework describes ADTs with interfaces:
  – Collection, Deque, List, Map, Queue, Set, SortedMap

• An ADT can be implemented in multiple ways by classes:
  – ArrayList and LinkedList implement List
  – HashSet and TreeSet implement Set
  – LinkedList, ArrayDeque, etc. implement Queue
An IntSet interface

// Represents a list of integers.
public interface IntSet {
    public void add(int value);
    public boolean contains(int value);
    public boolean isEmpty();
    public void remove(int value);
    public int size();
}

public class IntTreeSet implements IntSet { ...
Inner classes

To get rid of our separate node file, we can use an *inner class*.

- **inner class**: A class defined inside of another class.
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as `OuterClassName.this`
Recall: Type Parameters

ArrayList<Type> name = new ArrayList<Type>();

• When constructing a java.util.ArrayList, you specify the type of elements it will contain in < and >.
  - ArrayList accepts a type parameter; it is a generic class.

ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Helene Martin");
names.add(42); // compiler error
// a parameterized (generic) class
public class name<Type> {
    ...
}

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as E or T.
    - You can require multiple type parameters separated by commas.

- The rest of your class's code can refer to that type by name.
Generics and inner classes

```java
public class Foo<T> {
    private class Inner<T> {...}  // incorrect
    private class Inner {...}    // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- The inner class should NOT redeclare the type parameter.
  - (If you do, it will create a second type param with the same name.)
Issues with generic objects

```java
public class TreeSet<E> {

    ... public void example(E value1, E value2) {

        // BAD:  value1 == value2  (they are objects)
        // GOOD:  value1.equals(value2)

        // BAD:  value1 < value2
        // GOOD:  value1.compareTo(value2) < 0

    }

}
```

- When testing objects of type `E` for equality, must use `equals`
- When testing objects of type `E` for `<` or `>`, must use `compareTo`
  - Problem: By default, `compareTo` doesn't compile! What's wrong!
// a parameterized (generic) class
public class name<Type extends Class/Interface> {
    ...
}

- A type constraint forces the client to supply a type that is a subclass of a given superclass or implements a given interface.
  
  • Then the rest of your code can assume that the type has all of the methods in that superclass / interface and can call them.
// Represents a list of values.
public interface Set<E> {
    public void add(E value);
    public boolean isEmpty();
    public boolean contains(E value);
    public void remove(E value);
    public int size();
}

public class TreeSet<E extends Comparable<E>> implements Set<E> {
    ...
}