CSCI 241
A Very Brief Intro to Generics
AVL Trees I: Rotations, Balance Factor
Announcements

- Schedule adjustments:
  - A2 is due Friday 2/15 instead of 2/11
  - Midterm exam is 2/22 instead of 2/15

- A1 grading is underway
  - A lot of people submitted A1 late
  - Not a lot of people visited me in office hours

- You may resubmit once for half of unit test credit back
Goals

• Know why Java has generics, and how to use and implement them.

• Be prepared to implement rotations in BSTs

• (probably next time) Be prepared to implement AVL rebalancing.
A Very Brief Intro to Generics
(because your lab depends on it)
Before Generics

/** A collection that contains no duplicate elements. */
interface Set {
    /** Return true iff the collection contains ob */
    boolean contains(Object ob);
    /** Add ob to the collection; return true iff
     * the collection is changed. */
    boolean add(Object ob);
    /** Remove ob from the collection; return true iff
     * the collection is changed. */
    boolean remove(Object ob);
    ...
}

Can contain anything that extends Object (any class at all)
• But **not primitive types**: int, double, float, boolean, …
The Problem

Set c = ...
c.add(“Hello”)
c.add(“World”);
...
for (Object ob : c) {
    String s = (String) ob;
    // do things with s
}

Notice: Arrays don’t have this problem!

String[] a = ...
a[0]= (“Hello”)
a[1]= (“World”);
...
for (String s : a) {
    System.out.println(s);
}
The Solution: Generics

Object[] oa = ... // array of Objects
String[] sa = ... // array of Strings
ArrayList<Object> oA = ... // ArrayList of Objects
ArrayList<String> oA = ... // ArrayList of Strings

Now the Set interface written like this:

```java
interface Set<T> {
    /** Return true iff the collection contains x */
    boolean contains(T x);

    /** Add x to the collection; return true iff
    * the collection is changed. */
    boolean add(T x);

    /** Remove x from the collection; return true iff
    * the collection is changed. */
    boolean remove(T x);
    ...
}
```
The Solution: Generics

The Set interface is now written like this:

```java
interface Set<T> {
    /** Return true iff the collection contains x */
    boolean contains(T x);

    /** Add x to the collection; return true iff
    * the collection is changed. */
    boolean add(T x);

    /** Remove x from the collection; return true iff
    * the collection is changed. */
    boolean remove(T x);
    ...
}
```

**Key idea:** I don’t need to know what T is to implement these!
The Solution: Generics

Key idea: I don’t need to know what T is to implement these!

```java
Set<String> c = ... 
c.add("Hello")   /* Okay */
c.add(1979);     /* Illegal: compile error! */
```

Generally speaking,

```java
Collection<String>
behaves like the parameterized type
Collection<T>
where all occurrences of T have been replaced by String.
```
The Solution: Generics

The bummer: T must extend Object - no primitive types. Can’t do:

```
Collection<int> c = ...;
```

Have to use:
```
Collection<Integer>
```

Java often seamlessly converts int to Integer and back.
```
Integer x = 5; // works
int x = new Integer(5); // works
```

"Autoboxing/unboxing"
ArraySet<T>

class ArraySet<T> implements Set<T> {
    T[] a;
    int size;
    /** Return true iff the collection contains x */
    boolean contains(T x) {
        for (int i = 0; i < size; i++) {
            if (a[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /** Add x to the collection; return true iff
     * the collection is changed. */
    boolean add(T x) {
        if (!contains(x)) {
            a[size] = x; // let’s hope a is big enough...
            size++;
            return true;
        }
        return false;
    }
}
Questions to Ponder

• What’s the runtime of each ArraySet operation?

• Sketch out the operations for a LinkedListSet and analyze their runtime.

• Sketch out the operations for a BSTSet and analyze their runtime.
Back to BSTs

Long ago, we built some trees:

```java
    t = new BST();
t.insert(10);    t = new BST();
t.insert(15);    t.insert(-1)
t.insert(16);    t.insert(8)
t.insert(8);     t.insert(9)
t.insert(16);    t.insert(10)
t.insert(16);    t.insert(11)
t.insert(9);     t.insert(15)
t.insert(11);    t.insert(16)
t.insert(-1);    t.insert(16)
```
Same values, different trees

Bad tree =(

Good tree =)
Can we make a tree less bad?

Bad tree =(  

Good tree =)
Can we make a tree less bad?
Can we make a tree less bad?
Can we make a tree less bad?
Can we make a tree less bad?
Can we make a tree less bad?
Tree Rotations

modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

CLRS Fig 13.2, pg 313
Tree rotations: DIY

Steps in left rotation (move y up to x’s position):
1. Transfer β (11)
2. Transfer the parent (n/a!)
3. Transfer x itself (10)

Perform a left rotation on the root of this tree:
Tree Rotations modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

Details: need to update child, parent, and (possibly) root pointers.

CLRS Fig 13.2, pg 313
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. **Transfer β**: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. **Transfer \( \beta \):** x’s right subtree becomes y’s old left subtree (\( \beta \))
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. **Transfer the parent:** y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

\[
\begin{align*}
x.R & \text{ gets } y.L \\
y.L.p & \text{ gets } x \\
y.p & \text{ gets } x.p \\
p.[L/R] & \text{ gets } y
\end{align*}
\]
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. **Transfer the parent**: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

\[
\begin{align*}
\text{x.R gets y.L} \\
\text{y.L.p gets x}
\end{align*}
\]

\[
\begin{align*}
\text{y.p gets x.p} \\
\text{p.[L/R] gets y}
\end{align*}
\]

(what if $\rho$ is null / x was root?)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. **Transfer the parent:** y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
y.p gets x.p
p.[L/R] gets y

(what if ρ is null / x was root?)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
y.p gets x.p
p.[L/R] gets y

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself**: x becomes y’s left subtree

- $x.R$ gets $y.L$
- $y.L.p$ gets x
- $y.p$ gets $x.p$
- $p.[L/R]$ gets y

$y.L$ gets x
$x.p$ gets y
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself:** x becomes y’s left subtree

- x.R gets y.L
- y.L.p gets x
- y.p gets x.p
- p.[L/R] gets y
- y.L gets x
- x.p gets y
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself:** x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
y.p gets x.p
p.[L/R] gets y

y.L gets x
x.p gets y
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
y.p gets x.p
p.[L/R] gets y
y.L gets x
x.p gets y

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

\[
\begin{align*}
\text{x.R gets y.L} \\
\text{y.L.p gets x} \\
\text{y.p gets x.p} \\
\text{p.[L/R] gets y} \\
\text{y.L gets x} \\
\text{x.p gets y}
\end{align*}
\]
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

$x.R$ gets $y.L$
$y.L.p$ gets x
$y.p$ gets $x.p$
$p.[L/R]$ gets y

$y.L$ gets x
$x.p$ gets y

**Overall Transformation**

```
LEFT ROTATE(T, x)
```

```
RIGHT ROTATE(T, y)
```
### LEFT-ROTATE\((T,x)\)

1. \(y = x.\text{right}\) \hspace{1cm} // set \(y\)
2. \(x.\text{right} = y.\text{left}\) \hspace{1cm} // turn \(y\)'s left subtree into \(x\)'s right subtree
3. \(\text{if } y.\text{left} \neq T.\text{nil}\)
   4. \(y.\text{left}.p = x\) \hspace{1cm} // link \(x\)'s parent to \(y\)
   5. \(y.p = x.p\)
4. \(\text{if } x.p == T.\text{nil}\)
5. \(T.\text{root} = y\)
6. \(\text{elseif } x == x.p.\text{left}\)
7. \(x.p.\text{left} = y\)
8. \(\text{else } x.p.\text{right} = y\)
9. \(y.\text{left} = x\) \hspace{1cm} // put \(x\) on \(y\)'s left
10. \(x.p = y\)

Notational quirk: assume \(T.\text{nil}\) means “null”
Tree Rotations

Steps in **left** rotation (move y up to x’s position):
1. Transfer $\beta$
2. Transfer the parent
3. Transfer x itself

- $x.R$ gets $y.L$
- $y.L.p$ gets x
- $y.p$ gets $x.p$
- $p.[L/R]$ gets y
- $y.L$ gets x
- $x.p$ gets y
We can make a tree less bad. Let’s quantify badness:

**Balance Factor**\( n \) = height\( n.\text{right} \) - height\( n.\text{left} \)

For convenience: define height\( \text{null} \) = -1
Can we improve balance?

Balance Factor(n) = height(n.right) - height(n.left)
Can we improve balance?

\[
\text{Balance Factor}(n) = \text{height}(n.\text{right}) - \text{height}(n.\text{left})
\]
Can we improve balance?

**Balance Factor**\( (n) = \text{height}(n.\text{right}) - \text{height}(n.\text{left}) \)
Can we improve balance?

Balance Factor(n) = height(n.right) - height(n.left)
Can we improve balance?

Balance Factor\( (n) \) = \text{height}(n.\text{right}) - \text{height}(n.\text{left})
Balance Factor

Balance Factor $b(n) = \text{height}(n.\text{right}) - \text{height}(n.\text{left})$

$T$: 10
  8
  -1
  9
  11
  15
  16

$U$: 10
  8
  -1
  9
  11
  15
  16

ABCD: What’s the largest absolute balance factor of any node in each tree?

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
AVL Trees

Balance Factor $b(n) = \text{height}(n.\text{right}) - \text{height}(\text{left})$

- Devised by Adelson-Velsky and Landis
- An AVL tree is a Binary Search Tree in which the following property holds:

**AVL property:** $-1 \leq b(n) \leq 1$ for all nodes $n$. 