Lecture 28

Class Invariants:
How tall is a tree?
Previously on COMP213. . .

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
- Binary Search Trees — halving = depth
- Look-up and insertion — logarithmic
- height as a field — constant time. . .
Previously on COMP213…

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
- Binary Search Trees — halving = depth
- Look-up and insertion — logarithmic
- height as a field — constant time…
Previously on COMP213. . .

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
  - Binary Search Trees — halving = depth
  - Look-up and insertion — logarithmic
- height as a field — constant time. . .
Previously on COMP213... 

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
- Binary Search Trees — halving = depth
- Look-up and insertion — logarithmic
- height as a field — constant time...
Previously on COMP213. . .

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
- Binary Search Trees — halving = depth
- Look-up and insertion — logarithmic
- height as a field — constant time. . .
Previously on COMP213. . .

- Linear search — in an array
- Binary search — in a sorted array (halve the array)
- Binary Search Trees — halving = depth
- Look-up and insertion — logarithmic
- height as a field — constant time. . .
How Tall is a Tree?

var l : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / l \ T2) = max(height(T1), height(T2)) + 1.

public int height() {
    return (root == null) ? 0 : root.height();
}
How Tall is a Tree?

According to Maude

```maude
var l : Int.
vars T1 T2 : BSTree.
eq height(null) = 0 .
eq height(T1 / l \ T2) = max(height(T1), height(T2)) + 1 .
```

According to Java: `BinarySearchTree#height()`

```java
public int height() {
    return (root == null) ? 0 : root.height();
}
```
How Tall is a Tree?

**according to Maude**

```plaintext
var l : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / I \ T2) = max(height(T1), height(T2)) + 1.
```

**according to Java: BinarySearchTree#height()**

```java
public int height() {
    return (root == null) ? 0 : root.height();
}
```
How Tall is a Tree?

according to Maude

\[
\begin{align*}
\text{var } & \text{l : Int.} \\
\text{vars } & \text{T1 T2 : BSTree.} \\
\text{eq } & \text{height(null) = 0.} \\
\text{eq } & \text{height(T1 / I \ T2) = max(height(T1), height(T2)) + 1.}
\end{align*}
\]

according to Java: BinarySearchTree#height()

```java
public int height() {
    return (root == null) ? 0 : root.height();
}
```
How Tall is a Tree?

according to Maude

```maude
var l : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / l \ T2) = max(height(T1), height(T2)) + 1.
```

according to Java: TreeNode#height()

```java
public int height() {
    int lh = (leftSubtree == null) ? 0 : leftSubtree.height();
    int rh = (rightSubtree == null) ? 0 : rightSubtree.height();
    return Math.max(lh, rh) + 1;
}
```
How Tall is a Tree?

according to Maude

\[
\begin{align*}
\text{var} & \; l : \text{Int}. \\
\text{vars} & \; T1 \; T2 : \text{BSTree}.
\end{align*}
\]

\[
\begin{align*}
\text{eq} & \; \text{height}(\text{null}) = 0. \\
\text{eq} & \; \text{height}(T1 / l \setminus T2) = \max(\text{height}(T1), \text{height}(T2)) + 1.
\end{align*}
\]

according to Java: TreeNode#height()

```java
public int height() {
    int lh = (leftSubtree == null) ? 0 : leftSubtree.height();
    int rh = (rightSubtree == null) ? 0 : rightSubtree.height();
    return Math.max(lh, rh) + 1;
}
```
How Tall is a Tree?

according to Maude

```plaintext
var l : Int .
vars T1 T2 : BSTree .

eq height(null) = 0 .

eq height(T1 \ l / T2) = max(height(T1), height(T2)) + 1 .
```

according to Java: TreeNode#height()

```java
public int height() {
    int lh = (leftSubtree == null) ?
        0 : leftSubtree.height();
    int rh = (rightSubtree == null) ?
        0 : rightSubtree.height();
    return Math.max(lh, rh) + 1;
}
```
How Tall is a Tree?

according to Maude

var l : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / l \ T2) = max(height(T1), height(T2)) + 1.

according to Java: TreeNode#height()

```java
public int height() {
    int lh = (leftSubtree == null) ? 0 : leftSubtree.height();
    int rh = (rightSubtree == null) ? 0 : rightSubtree.height();
    return Math.max(lh, rh) + 1;
}
```
Efficiency

The recursive calls are not very efficient: there is one call of `height()` for each node in the tree.

If there are 1000 nodes in the tree, we need 1000 recursive calls of `height()`; double the number of nodes in the tree, and that doubles the number of calls of `height()`.

If each `TreeNode` stores its height as a field, then we can compute `height()` in constant time:
just get the value of the field

if we double the number of nodes in the tree, it doesn’t matter; we still only need get the value of the field
Efficiency

The recursive calls are not very efficient: there is one call of `height()` for each node in the tree.

If there are 1000 nodes in the tree, we need 1000 recursive calls of `height()`; double the number of nodes in the tree, and that doubles the number of calls of `height()`.

If each `TreeNode` stores its height as a field, then we can compute `height()` in constant time:
just get the value of the field

if we double the number of nodes in the tree, it doesn’t matter; we still only need get the value of the field
The recursive calls are not very efficient: there is one call of `height()` for each node in the tree. If there are 1000 nodes in the tree, we need 1000 recursive calls of `height()`; double the number of nodes in the tree, and that doubles the number of calls of `height()`.

If each `TreeNode` stores its height as a field, then we can compute `height()` in constant time: just get the value of the field. If we double the number of nodes in the tree, it doesn’t matter; we still only need get the value of the field.
Correctness

If the proposed height field is to do the work of the height() method, we must make sure that whenever the height of the tree is changed, the value of the height field is updated appropriately.

This is exactly what we mean by class invariant.
Correctness

If the proposed **height field** is to do the work of the `height()` method, we must make sure that whenever the height of the tree is changed, the value of the **height** field is updated appropriately.

This is exactly what we mean by **class invariant**.
The property that we want to be true for all BinarySearchTree instances is:

For all TreeNode nodes in the tree:

height really is the height of the node; i.e.,

\[
\text{height} = \text{Math.max(leftSubtree.height, rightSubtree.height)} + 1
\]
Height as a field

private class TreeNode {
    private int value;
    private TreeNode leftSubTree;
    private TreeNode rightSubTree;
    private int height;

    public TreeNode(int v) {
        value = v;
        height = 1;
    }
}

Establishing the invariant: height really is height
Height as a field

```java
private class TreeNode {
    private int value;
    private TreeNode leftSubTree;
    private TreeNode rightSubTree;
    private int height;

    public TreeNode(int v) {
        value = v;
        height = 1;
    }
}
```

Establishing the invariant: height really is height
public int height() {
    if (root == null) {
        return 0;
    } else {
        return root.height;
    }
}
Height as a field

in class BinarySearchTree v2

```java
public int height() {
    if (root == null) {
        return 0;
    } else {
        return root.height;
    }
}
```

or even:

```java
public int height() {
    return (root == null) ? 0 : root.height;
}
```
Checking the Class Invariant

In order for ‘height is height’ to be a class invariant, it has to be made true by all constructors.

In this case we just have the one constructor that has root set to null, which does indeed establish the class invariant (trivially, because there are no nodes in the tree).

```java
BinarySearchTree constructor

public BinarySearchTree() {
}
```
Checking the Class Invariant

In order for ‘height is height’ to be a class invariant, it has to be made true by all constructors.

In this case we just have the one constructor that has root set to null, which does indeed establish the class invariant (trivially, because there are no nodes in the tree).

```java
public BinarySearchTree() {
}
```
Checking the Invariant

We need the property to be preserved by all public methods. We’ll concentrate on `insert(int)`, as this could alter the height of the tree.

```java
public void insert(int v) {
    if (root == null) {
        root = new TreeNode(v);
    } else {
        root.insert(v);
    }
}
```

So we need to implement `TreeNode#insert(int)` in such a way that the class invariant is preserved.
Checking the Invariant

We need the property to be preserved by all public methods. We’ll concentrate on `insert(int)`, as this could alter the height of the tree.

```java
public void insert(int v) {
    if (root == null) {
        root = new TreeNode(v);
    } else {
        root.insert(v);
    }
}
```

So we need to implement `TreeNode#insert(int)` in such a way that the class invariant is preserved.
Checking the Invariant

We need the property to be preserved by all public methods. We’ll concentrate on `insert(int)`, as this could alter the height of the tree.

```java
public void insert(int v) {
    if (root == null) {
        root = new TreeNode(v);
    } else {
        root.insert(v);
    }
}
```

So we need to implement `TreeNode#insert(int)` in such a way that the class invariant is preserved.
Checking the Invariant

We need the property to be preserved by all public methods. We’ll concentrate on `insert(int)`, as this could alter the height of the tree.

```java
public void insert(int v) {
    if (root == null) {
        root = new TreeNode(v);
    } else {
        root.insert(v);
    }
}
```

So we need to implement `TreeNode#insert(int)` in such a way that the class invariant is preserved.
We need to update height fields if inserting the value increases the height of the tree. We’ll make the method tell us whether the height has changed (so we can update height fields if necessary) and make it return a boolean value:

```java
/**
 * @param v the value to insert
 * @return true if the height of the tree has increased, false if the height is unchanged
 */
public boolean insert(int v) {
    boolean heightIncreased = false;
    // this is the value to be returned: initially false, as we don’t yet know that the height has increased
```
Insertion

We need to update height fields if inserting the value increases the height of the tree. We’ll make the method tell us whether the height has changed (so we can update height fields if necessary) and make it return a boolean value:

```java
public boolean insert(int v) {
    boolean heightIncreased = false;
    // this is the value to be returned: initially false, as we don't yet know that the height has increased
```
Insertion

We need to update height fields if inserting the value increases the height of the tree. We’ll make the method tell us whether the height has changed (so we can update height fields if necessary) and make it return a boolean value:

```java
/**
 * @param v the value to insert
 * @return true if the height of the tree has increased, false if the height is unchanged
 */
public boolean insert(int v) {
    boolean heightIncreased = false;
}
```

this is the value to be returned: initially false, as we don’t yet know that the height has increased.
We need to update `height` fields if inserting the value increases the height of the tree. We’ll make the method tell us whether the height has changed (so we can update `height` fields if necessary) and make it return a `boolean` value:

```java
/**
 * @param v the value to insert
 * @return true if the height of the tree has increased, false if the height is unchanged
 */
public boolean insert(int v) {
    boolean heightIncreased = false;
    // this is the value to be returned: initially false, as we don’t yet know that the height has increased
```
A sketch of what we need to do is:

```plaintext
insert(v) pseudocode

if v < value
    insert v in the left subtree
    update height
    update heightIncreased
else if value < v
    insert v in the right subtree
    update height
    update heightIncreased

return heightIncreased
```

Just as before.
New.
A sketch of what we need to do is:

```plaintext
insert(v) pseudocode

if v < value
    insert v in the left subtree
    update height
    update heightIncreased
else if value < v
    insert v in the right subtree
    update height
    update heightIncreased

return heightIncreased
```

Just as before.

New.
A sketch of what we need to do is:

```
insert(v) pseudocode

if v < value
    insert v in the left subtree
    update height
    update heightIncreased
else if value < v
    insert v in the right subtree
    update height
    update heightIncreased

return heightIncreased
```

Just as before.

New.
More specifically, if we’re inserting in the left subtree, `heightIncreased` will tell us whether the height of the left subtree has increased.

Similarly, if we’re inserting in the right subtree, `heightIncreased` will tell us whether the height of the right subtree has increased.
Insertion, etc., in left subtree

pseudocode for: insert v in the left subtree, etc.

if leftSubtree is empty
    leftSubtree = new TreeNode(v)
    heightIncreased = true
else
    heightIncreased = leftSubtree.insert(v)

As before.
height of left subtree has increased from 0 to 1
height of left subtree may or may not have increased
Insertion, etc., in left subtree

pseudocode for: insert v in the left subtree, etc.

if leftSubtree is empty
    leftSubtree = new TreeNode(v)
    heightIncreased = true
else
    heightIncreased = leftSubtree.insert(v)

As before.
height of left subtree has increased from 0 to 1
height of left subtree may or may not have increased
Insertion, etc., in left subtree

**pseudocode for: insert v in the left subtree, etc.**

if leftSubtree is empty
    leftSubtree = new TreeNode(v)
    heightIncreased = true
else
    heightIncreased = leftSubtree.insert(v)

As before.
height of left subtree has increased from 0 to 1
height of left subtree may or may not have increased
Insertion, etc., in left subtree

pseudocode for: insert v in the left subtree, etc.

if leftSubtree is empty
   leftSubtree = new TreeNode(v)
   heightIncreased = true
else
   heightIncreased = leftSubtree.insert(v)

As before.
height of left subtree has increased from 0 to 1
height of left subtree may or may not have increased
Insertion, etc., in left subtree

**pseudocode for: insert v in the left subtree, etc.**

if leftSubtree is empty
   leftSubtree = new TreeNode(v)
   heightIncreased = true
else
   heightIncreased = leftSubtree.insert(v)

As before.
height of left subtree has increased from 0 to 1
height of left subtree may or may not have increased
Insertion, etc., in left subtree

---

**pseudocode for: insert v in the left subtree, etc.**

```java
if leftSubtree is empty
    leftSubtree = new TreeNode(v)
    heightIncreased = true
else
    heightIncreased = leftSubtree.insert(v)
```

As before.  
height of left subtree has increased from 0 to 1  
height of left subtree may or may not have increased
Now we’ve (pseudo) inserted $v$ in the left subtree, and updated `heightIncreased` so that it’s true if, and only if, the height of the left subtree has actually increased.

We still need to update `height`, but what we’ve done so far in pseudocode can be translated fairly straightforwardly into working Java code....
Now we’ve (pseudo) inserted $v$ in the left subtree, and updated `heightIncreased` so that it’s true if, and only if, the height of the left subtree has actually increased.

We still need to update `height`, but what we’ve done so far in pseudocode can be translated fairly straightforwardly into working Java code...
if (v < value) { // insert in left subtree
    if (leftSubtree == null) {
        leftSubtree = new TreeNode(v);
        heightIncreased = true;
    } else { // leftSubtree != null
        heightIncreased = leftSubtree.insert(v);
    }
}

the height of the left subtree has increased from 0 to 1
insert at the appropriate point in the left subtree and make a
note of whether the height of the tree has increased
if (v < value) { // insert in left subtree
   if (leftSubtree == null) {
      leftSubtree = new TreeNode(v);
      heightIncreased = true;
   } else { // leftSubtree != null
      heightIncreased = leftSubtree.insert(v);
   }
}

the height of the left subtree has increased from 0 to 1
insert at the appropriate point in the left subtree and make a
note of whether the height of the tree has increased
Insertion, etc.

```
if (v < value) {  // insert in left subtree
    if (leftSubtree == null) {
        leftSubtree = new TreeNode(v);
        heightIncreased = true;
    }
} else {  // leftSubtree != null
    heightIncreased = leftSubtree.insert(v);
}
```

the height of the left subtree has increased from 0 to 1
insert at the appropriate point in the left subtree and make a note of whether the height of the tree has increased
if (v < value) { // insert in left subtree
    if (leftSubtree == null) {
        leftSubtree = new TreeNode(v);
        heightIncreased = true;
    } else { // leftSubtree != null
        heightIncreased = leftSubtree.insert(v);
    }
}

the height of the left subtree has increased from 0 to 1
insert at the appropriate point in the left subtree and make a
note of whether the height of the tree has increased
if (v < value) {  // insert in left subtree
    if (leftSubtree == null) {
        leftSubtree = new TreeNode(v);
        heightIncreased = true;
    } else {  // leftSubtree != null
        heightIncreased = leftSubtree.insert(v);
    }
}

The height of the left subtree has increased from 0 to 1.

Insert at the appropriate point in the left subtree and make a note of whether the height of the tree has increased.
Insertion, etc., in left subtree

Now we’ve (really) inserted $v$ in the left subtree, and updated $\text{heightIncreased}$ so that it’s true if, and only if, the height of the left subtree has actually increased.

We may assume that the $\text{height}$ field of the left subtree is correct (in the sense of our class invariant) because:

- if the left subtree was empty, the $\text{TreeNode}$ constructor makes the class invariant true
- if the left subtree wasn’t empty, the recursive call of insert($v$) will do everything we code it to do:
  - insert $v$
  - update $\text{heightIncreased}$
  - update $\text{height}$

so long as we write correct code!
Now we’ve (really) inserted $v$ in the left subtree, and updated `heightIncreased` so that it’s true if, and only if, the height of the left subtree has actually increased.

We may assume that the `height` field of the left subtree is correct (in the sense of our class invariant) because:

- if the left subtree was empty, the `TreeNode` constructor makes the class invariant true
- if the left subtree wasn’t empty, the recursive call of `insert(v)` will do everything we code it to do:
  - insert $v$
  - update `heightIncreased`
  - update `height`

so long as we write correct code!
Insertion, etc., in left subtree

Now we’ve (really) inserted $v$ in the left subtree, and updated `heightIncreased` so that it’s true if, and only if, the height of the left subtree has actually increased.

We may assume that the `height` field of the left subtree is correct (in the sense of our class invariant) because:

- if the left subtree was empty, the `TreeNode` constructor makes the class invariant true
- if the left subtree wasn’t empty, the recursive call of `insert(v)` will do everything we code it to do:
  - insert $v$
  - update `heightIncreased`
  - update `height`

so long as we write correct code!
Insertion, etc., in left subtree

Now we’ve (really) inserted $v$ in the left subtree, and updated `heightIncreased` so that it’s true if, and only if, the height of the left subtree has actually increased.

We may assume that the `height` field of the left subtree is correct (in the sense of our class invariant) because:

- if the left subtree was empty, the `TreeNode` constructor makes the class invariant true
- if the left subtree wasn’t empty, the recursive call of `insert(v)` will do everything we code it to do:
  - insert $v$
  - update `heightIncreased`
  - update `height`

so long as we write correct code!
Insertion, etc., in left subtree

Now we’ve (really) inserted $v$ in the left subtree, and updated $\text{heightIncreased}$ so that it’s true if, and only if, the height of the left subtree has actually increased.

We may assume that the $\text{height}$ field of the left subtree is correct (in the sense of our class invariant) because:

1. If the left subtree was empty, the $\text{TreeNode}$ constructor makes the class invariant true.

2. If the left subtree wasn’t empty, the recursive call of $\text{insert}(v)$ will do everything we code it to do:
   - $\text{insert} \ v$
   - $\text{update} \ \text{heightIncreased}$
   - $\text{update} \ \text{height}$

so long as we write correct code!
Insertion, etc., in left subtree

We’ve inserted $v$ in `leftSubtree` and updated `heightIncreased`. Now we need to update `height`.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted $v$ in $\text{leftSubtree}$ and updated $\text{heightIncreased}$. Now we need to update $\text{height}$.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
We’ve inserted \( v \) in \texttt{leftSubtree} and updated \texttt{heightIncreased}. Now we need to update \texttt{height}.

The height of the \texttt{current} node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree
  (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees
  i.e., left height was greater than or equal to right height)
- \textbf{and} the height of the left subtree has increased
  (so left height is now greater than right height)

before/after recursive call
We’ve inserted $v$ in $\text{leftSubtree}$ and updated $\text{heightIncreased}$. Now we need to update $\text{height}$.

The height of the $\text{current}$ node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted $v$ in `leftSubtree` and updated `heightIncreased`. Now we need to update `height`.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)

- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted $v$ in leftSubtree and updated heightIncreased. Now we need to update height.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted \( v \) in `leftSubtree` and updated `heightIncreased`. Now we need to update `height`.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)

- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted $v$ in leftSubtree and updated heightIncreased. Now we need to update height.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Insertion, etc., in left subtree

We’ve inserted $v$ in leftSubtree and updated heightIncreased. Now we need to update height.

The height of the current node has increased if, and only if,

- the height of the current node was determined by the height of the left subtree (i.e., the height of the left subtree was the maximum of the heights of the left and right subtrees i.e., left height was greater than or equal to right height)
- and the height of the left subtree has increased (so left height is now greater than right height)

before/after recursive call
Updating height and heightIncreased

Soooo...

```plaintext
pseudocode for: update height

if heightIncreased
    if right height < left Height
        increment height
    else heightIncreased = false

if height of left subtree has increased
    height of this node has increased
    left height has simply caught up with right height
```
Updating height and heightIncreased

Soooo...

pseudocode for: update height

if heightIncreased
  if right height < left Height
    increment height
  else heightIncreased = false

if height of left subtree has increased
  height of this node has increased
  left height has simply caught up with right height
Updating height and heightIncreased

Soooo...

<table>
<thead>
<tr>
<th>pseudocode for: update height</th>
</tr>
</thead>
<tbody>
<tr>
<td>if heightIncreased</td>
</tr>
<tr>
<td>if right height &lt; left Height</td>
</tr>
<tr>
<td>increment height</td>
</tr>
<tr>
<td>else heightIncreased = false</td>
</tr>
</tbody>
</table>

if height of left subtree has increased
height of this node has increased
left height has simply caught up with right height
We need to update the `height` field: the left subtree is taller; we need to compare its height to that of the right subtree.
We need to update the `height` field: the left subtree is taller; we need to compare its height to that of the right subtree.
Insertion

insert(int), contd.

```java
if (rHeight < leftSubtree.height) {
    height = leftSubtree.height + 1;
    // heightIncreased remains true
} else {
    heightIncreased = false;
}
} // if (heightIncreased)
```

height of the current node is determined by the height of the left subtree
update correct value of `height`
height was determined by height of right subtree
so even though the left subtree is taller than it was, it’s just caught up with the right subtree
height of the current node is determined by the height of the left subtree
update correct value of height
height was determined by height of right subtree
so even though the left subtree is taller than it was, it’s just caught up with the right subtree
Insertion

insert(int), contd.

```java
if (rHeight < leftSubtree.height) {
    height = leftSubtree.height + 1;
    // heightIncreased remains true
} else {
    heightIncreased = false;
}
} // if (heightIncreased)
```

height of the current node is determined by the height of the left subtree

update correct value of `height`

height was determined by height of right subtree

so even though the left subtree is taller than it was, it’s just caught up with the right subtree
Insertion

insert(int), contd.

```java
if (rHeight < leftSubtree.height) {
    height = leftSubtree.height + 1;
    // heightIncreased remains true
} else {
    heightIncreased = false;
}
} // if (heightIncreased)
```

height of the current node is determined by the height of the left subtree
update correct value of `height`
height was determined by height of right subtree
so even though the left subtree is taller than it was, it’s just caught up with the right subtree
height of the current node is determined by the height of the left subtree
update correct value of height
height was determined by height of right subtree
so even though the left subtree is taller than it was, it’s just caught up with the right subtree
Insertion

**insert(int), contd.**

```java
if (rHeight < leftSubtree.height) {
    height = leftSubtree.height + 1;
    // heightIncreased remains true
} else {
    heightIncreased = false;
}
} // if (heightIncreased)
```

Height of the current node is determined by the height of the left subtree.
Update correct value of `height`.
Height was determined by height of `right` subtree.
So even though the left subtree is taller than it was, it’s just caught up with the right subtree.
We’ve finished the case where we need to insert in the left subtree; this is the case where we need to insert in the right subtree
done: value is inserted, and height updated;
let recursive callers do their stuff
We’ve finished the case where we need to insert in the left subtree; this is the case where we need to insert in the right subtree. done: value is inserted, and height updated; let recursive callers do their stuff.
We’ve finished the case where we need to insert in the left subtree; this is the case where we need to insert in the right subtree done: value is inserted, and height updated; let recursive callers do their stuff
We’ve finished the case where we need to insert in the left subtree; this is the case where we need to insert in the right subtree.  

done: value is inserted, and height updated; let recursive callers do their stuff.
We’ve implemented a recursive insertion method that inserts values in logarithmic time.

The `height()` method takes constant time (it doesn’t depend upon how many elements are in the tree), and `insert(int)` preserves the class invariant ‘height is height’.

We could implement a slightly more efficient version of `insert(int)` that uses a loop rather than recursive calls, but first, let’s go back to the efficiency....

* * well...
We’ve implemented a recursive insertion method that inserts values in logarithmic time*

The `height()` method takes constant time (it doesn’t depend upon how many elements are in the tree), and `insert(int)` preserves the class invariant ‘height is height’.

We could implement a slightly more efficient version of `insert(int)` that uses a loop rather than recursive calls, but first, let’s go back to the efficiency . . .

* well . . .
And That’s It

We’ve implemented a recursive insertion method that inserts values in logarithmic time*

The `height()` method takes constant time (it doesn’t depend upon how many elements are in the tree), and `insert(int)` preserves the class invariant ‘`height` is height’.

We could implement a slightly more efficient version of `insert(int)` that uses a loop rather than recursive calls, but first, let’s go back to the efficiency…

* well …
We’ve implemented a recursive insertion method that inserts values in logarithmic time*

The `height()` method takes constant time (it doesn’t depend upon how many elements are in the tree), and `insert(int)` preserves the class invariant ‘`height` is height’.

We could implement a slightly more efficient version of `insert(int)` that uses a loop rather than recursive calls, but first, let’s go back to the efficiency....

* well ...
We’ve implemented a recursive insertion method that inserts values in logarithmic time.

The `height()` method takes constant time (it doesn’t depend upon how many elements are in the tree), and `insert(int)` preserves the class invariant ‘height is height’.

We could implement a slightly more efficient version of `insert(int)` that uses a loop rather than recursive calls, but first, let’s go back to the efficiency...
A Test Case

We might test our implementation with:

```java
dpublic static void main(String[] args) {
    BinarySearchTree t = new BinarySearchTree();
    for (int i = 0; i < 2000000; i++) {
        t.insert(i);
    }
}
```

and we can see that this is just as bad as using an array!
A Test Case

We might test our implementation with:

```java
class BinarySearchTree {
    public static void main(String[] args) {
        BinarySearchTree t = new BinarySearchTree();
        for (int i = 0; i < 2000000; i++) {
            t.insert(i);
        }
    }
}
```

and we can see that this is just as bad as using an array!
Yet Another Class Invariant

Our claim that look-up and insertion works in logarithmic time for Binary Search Trees was based on the assumption that values would be ‘evenly spread’ through the tree.

I.e., every time we move to the left- or right-subtree, we should halve the number of elements we need to consider.

Clearly, this doesn’t work if the tree is ‘unbalanced’: what we need are trees where the number of elements in the left subtree is roughly the same as the number of elements in the right subtree.
Yet Another Class Invariant

Our claim that look-up and insertion works in logarithmic time for Binary Search Trees was based on the assumption that values would be ‘evenly spread’ through the tree.

I.e., every time we move to the left- or right-subtree, we should halve the number of elements we need to consider.

Clearly, this doesn’t work if the tree is ‘unbalanced’: what we need are trees where the number of elements in the left subtree is roughly the same as the number of elements in the right subtree.
Our claim that look-up and insertion works in logarithmic time for Binary Search Trees was based on the assumption that values would be ‘evenly spread’ through the tree.

I.e., every time we move to the left- or right-subtree, we should halve the number of elements we need to consider.

Clearly, this doesn’t work if the tree is ‘unbalanced’: what we need are trees where the number of elements in the left subtree is roughly the same as the number of elements in the right subtree.
Rather than just wishing this were the case, we’ll *make* it the case.

We impose a new class invariant:

For every node $t$ in the tree:

\[
\text{\texttt{t.leftSubtree.height} differs from \texttt{t.rightSubtree.height} by at most one.}
\]

Such trees are sometimes called ‘balanced trees’, or **AVL Trees** (after Adelson-Velski and Landis, who invented them).
Rather than just wishing this were the case, we’ll *make* it the case.

We impose a new class invariant:

**class invariant for ‘balanced’ trees**

for every node \( t \) in the tree:

\[ t\.leftSubtree\.height \text{ differs from } t\.rightSubtree\.height \text{ by at most one.} \]

Such trees are sometimes called ‘balanced trees’, or **AVL Trees** (after Adelson-Velski and Landis, who invented them).
Rather than just wishing this were the case, we’ll *make* it the case.

We impose a new class invariant:

```plaintext
class invariant for ‘balanced’ trees

for every node t in the tree:

    t.leftSubtree.height differs from t.rightSubtree.height
by at most one.
```

Such trees are sometimes called ‘balanced trees’, or **AVL Trees** (after Adelson-Velski and Landis, who invented them).
Keeping Balance

An example of a balanced tree:

If we insert 1 or 7 or 28 or 54, it will remain balanced; if we insert 9 or 21 or 35, it will become unbalanced.
Keeping Balance

An example of a balanced tree:

If we insert 1 or 7 or 28 or 54, it will remain balanced; if we insert 9 or 21 or 35, it will become unbalanced.
Keeping Balance?

**Problem**: how can we insert a value into a balanced tree in such a way that the result is also a balanced tree?
That’s All, Folks!

Summary

- Loops v Recursion v Fields
- Class Invariants
- AVL Trees
- Tree Rotations

Next:

insert();
last few class invariants