Advanced Object-oriented Programming

Lecture 30
Class Invariants: AVL trees, contd.
When a value is inserted into a balanced tree, it might cause the tree to become unbalanced (the height of some left subtree differs from the right subtree by two).

We need to implement insert(int) in such a way that it preserves the ‘balanced’ class invariant.

It’s fairly straightforward to implement this with recursive calls, but loops are more efficient, so we’re going to try that.
Where we were: Keeping Balance

When a value is inserted into a balanced tree, it might cause the tree to become unbalanced (the height of some left subtree differs from the right subtree by two).

We need to implement insert(int) in such a way that it preserves the ‘balanced’ class invariant.

It’s fairly straightforward to implement this with recursive calls, but loops are more efficient, so we’re going to try that.
Where we were: Keeping Balance

When a value is inserted into a balanced tree, it might cause the tree to become unbalanced (the height of some left subtree differs from the right subtree by two).

We need to implement insert(int) in such a way that it preserves the `balanced` class invariant.

It’s fairly straightforward to implement this with recursive calls, but loops are more efficient, so we’re going to try that.
Where we were: Keeping Balance

When a value is inserted into a balanced tree, it might cause the tree to become unbalanced (the height of some left subtree differs from the right subtree by two).

We need to implement `insert(int)` in such a way that it preserves the ‘balanced’ class invariant.

It’s fairly straightforward to implement this with recursive calls, but loops are more efficient, so we’re going to try that.
The program we end up with will be quite similar to the previous version of `insert(int)`:

insert the value, then correct stuff.

For the recursive version, the general pattern is:

```
recursive call
correct stuff
```

For a loop implementation, this translates to:

```
traverse the tree downwards to insertion point
then traverse back up, maintaining invariants
```
The program we end up with will be quite similar to the previous version of `insert(int)`:

insert the value, then correct stuff.

For the recursive version, the general pattern is:

- recursive call
- correct stuff

For a loop implementation, this translates to

- traverse the tree downwards to insertion point
- then traverse back up, maintaining invariants
Keeping Balance

The program we end up with will be quite similar to the previous version of `insert(int)`:

insert the value, then correct stuff.

For the recursive version, the general pattern is:

- recursive call
- correct stuff

For a loop implementation, this translates to

- traverse the tree downwards to insertion point
- then traverse back up, maintaining invariants
Traversing Upwards

In order to traverse backwards up the tree, we need to change the `TreeNode` class, to include a pointer to the ‘parent’ node (a bit like doubly-linked lists):

```java
class AVLTreeNode {
    private int value;
    private int height;
    private AVLTreeNode leftSubtree;
    private AVLTreeNode rightSubtree;
    private AVLTreeNode parent;

    public AVLTreeNode(int v) {
        value = v;
        height = 1;
    }
}
```
In order to traverse backwards up the tree, we need to change the `TreeNode` class, to include a pointer to the ‘parent’ node (a bit like doubly-linked lists):

class AVLTreeNode {
    private int value;
    private int height;
    private AVLTreeNode leftSubtree;
    private AVLTreeNode rightSubtree;
    private AVLTreeNode parent;

    public AVLTreeNode(int v) {
        value = v;
        height = 1;
    }
}
AVL Trees

We’ll also take the opportunity to include some useful methods.

```java
public int leftHeight() {
    return (leftSubtree == null) ? 0 : leftSubtree.height;
}

public int rightHeight() {
    return (rightSubtree == null) ? 0 : rightSubtree.height;
}

public void setHeight() {
    height = Math.max(leftHeight(), rightHeight()) + 1;
}
```
And we have to take care of parent links:

```java
static AVLTreeNode leftRotate(AVLTreeNode w) {
    // temp = v
    AVLTreeNode temp = w.leftSubtree;
    temp.parent = w.parent;
    // replace v with t3
    w.leftSubtree = temp.rightSubtree;
    temp.rightSubtree.parent = w;
    // replace t3 with w
    temp.rightSubtree = w;
    w.parent = temp;
    return temp;
}
```
And we have to take care of parent links:

class AVLTreeNode, contd.

static AVLTreeNode leftRotate(AVLTreeNode w) {
    // temp = v
    AVLTreeNode temp = w.leftSubtree;
    temp.parent = w.parent;

    // replace v with t3
    w.leftSubtree = temp.rightSubtree;
    temp.rightSubtree.parent = w;

    // replace t3 with w
    temp.rightSubtree = w;
    w.parent = temp;

    return temp;
}
And we have to take care of parent links:

class AVLTreeNode, contd.

static AVLTreeNode leftRotate(AVLTreeNode w) {
    // temp = v
    AVLTreeNode temp = w.leftSubtree;
    temp.parent = w.parent;
    // replace v with t3
    w.leftSubtree = temp.rightSubtree;
    temp.rightSubtree.parent = w;
    // replace t3 with w
    temp.rightSubtree = w;
    w.parent = temp;
    return temp;
}
And we have to take care of parent links:

```java
class AVLTreeNode, contd.

static AVLTreeNode leftRotate(AVLTreeNode w) {
    // temp = v
    AVLTreeNode temp = w.leftSubtree;
    temp.parent = w.parent;

    // replace v with t3
    w.leftSubtree = temp.rightSubtree;
    temp.rightSubtree.parent = w;

    // replace t3 with w
    temp.rightSubtree = w;
    w.parent = temp;

    return temp;
}
```
And we have to take care of parent links:

class AVLTreeNode, contd.

```java
static AVLTreeNode leftRotate(AVLTreeNode w) {
    // temp = v
    AVLTreeNode temp = w.leftSubtree;
    temp.parent = w.parent;
    // replace v with t3
    w.leftSubtree = temp.rightSubtree;
    temp.rightSubtree.parent = w;
    // replace t3 with w
    temp.rightSubtree = w;
    w.parent = temp;
    return temp;
}
```
Now we’re ready to implement `insert(int)`.

The method is split into two halves:

```java
public void insert(int v) {
    // insert v
    while (not inserted) {
        ...
    }
    // maintain invariants
    while (not finished) {
    }
}
```

loop down through the tree to find the insertion point, and insert v

loop up through the tree from the insertion point to the root, checking height and balance
Now we’re ready to implement `insert(int)`.

The method is split into two halves:

```java
in class AVLTree

public void insert(int v) {
    // insert v
    while (not inserted) {
        ...
    }
    // maintain invariants
    while (not finished) {
    }
}
```

loop down through the tree to find the insertion point, and insert `v`

loop up through the tree from the insertion point to the root, checking height and balance
Now we’re ready to implement `insert(int)`. The method is split into two halves:

```java
// in class AVLTree
public void insert(int v) {
    // insert v
    while (not inserted) {
        ...
    }
    // maintain invariants
    while (not finished) {
    }
    // loop down through the tree to find the insertion point, and insert v
    loop up through the tree from the insertion point to the root, checking height and balance
```
boolean heightIncreased = false;
AVLTreeNode tn = root;
while (!heightIncreased) {
    if (tn.value == v) {
        return;
    }
}

This variable will be set true once v has been inserted; this will end the first loop
v is already in the list; do nothing
boolean heightIncreased = false;
AVLTreeNode tn = root;
while (! heightIncreased) {
    if (tn.value == v) {
        return;
    }
}

This variable will be set true once \textit{v} has been inserted; this will end the first loop
\textit{v} is already in the list; do nothing
This variable will be set true once \( v \) has been inserted; this will end the first loop

\( v \) is already in the list; do nothing
```java
boolean heightIncreased = false;
AVLTreeNode tn = root;
while (!heightIncreased) {
    if (tn.value == v) {
        return;
    }
}
```

This variable will be set true once \( v \) has been inserted; this will end the first loop.

\( v \) is already in the list; do nothing.
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}

insert \(v\) in the left subtree
this is the insertion point
Add the value…
set the parent value…
… and exit the loop
carry on in left subtree on the next iteration of the loop
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}

insert v in the left subtree
this is the insertion point
Add the value...
set the parent value...
...and exit the loop
carry on in left subtree on the next iteration of the loop
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}

Insertion Point
in method insert(int); first loop

insert v in the left subtree
this is the insertion point
Add the value...
set the parent value...
...and exit the loop
carry on in left subtree on the next iteration of the loop
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}

insert $v$ in the left subtree
this is the insertion point
Add the value...
set the parent value...
...and exit the loop
carry on in left subtree on the next iteration of the loop
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}

insert v in the left subtree
this is the insertion point
Add the value...
set the parent value...
...and exit the loop
carry on in left subtree on the next iteration of the loop
else if (v < tn.value) {
    if (tn.leftSubtree == null) {
        tn.leftSubtree = new AVLTreeNode(v);
        tn.leftSubtree.parent = tn;
        heightIncreased = true;
    } else {
        tn = tn.leftSubtree;
    }
}
else if (v > tn.value) {
    // same thing for right subtree
}
} // while

/*
 * v has been inserted; now traverse back up
 * the tree, checking heights and balance
 */
else if (v > tn.value) {
    // same thing for right subtree
}
} // while

/*
 * v has been inserted; now traverse back up
 * the tree, checking heights and balance
 */
Maintaining Invariants

The second loop follows the parent pointers up from the insertion point to the root of the tree:

```java
in method insert(int)
while (tn != null && heightIncreased) {
    // check height and balance of tn
    tn = tn.parent;
}
```

we end the loop when either:

- we’re at the root (root.parent is null), or
- we know that the height of tn has not increased after the insertion of v
Maintaining Invariants

The second loop follows the parent pointers up from the insertion point to the root of the tree:

```java
while (tn != null && heightIncreased) {
    // check height and balance of tn
    tn = tn.parent;
}
```

we end the loop when either:

- we’re at the root (root.parent is null), or
- we know that the height of tn has not increased after the insertion of v
Maintaining Invariants

The second loop follows the **parent** pointers up from the insertion point to the root of the tree:

```java
while (tn != null && heightIncreased) {
    // check height and balance of tn
    tn = tn.parent;
}
```

we end the loop when either:

- we’re at the root (**root.parent** is **null**), or
- we know that the height of **tn** has not increased after the insertion of **v**
Maintaining Invariants

The second loop follows the parent pointers up from the insertion point to the root of the tree:

```java
in method insert(int)
while (tn != null && heightIncreased) {
    // check height and balance of tn
    tn = tn.parent;
}
```

we end the loop when either:

- we’re at the root (root.parent is null), or
- we know that the height of \( t_n \) has not increased after the insertion of \( v \)
Maintaining Invariants

Throughout this loop, we keep the following true:

- the subtrees of $tn$ are balanced (heights differ by no more than one); and
- $tn$ and all its subtrees have correct $height$ values; and
- $heightIncreased$ is true if — and only if — the height of $tn$ really has increased because of the insertion of $v$
Maintaining Invariants

Throughout this loop, we keep the following true:

- the subtrees of \( t_n \) are balanced (heights differ by no more than one); and

- \( t_n \) and all its subtrees have correct \texttt{height} values; and

- \texttt{heightIncreased} \ is true if — and only if — the height of \( t_n \) really has increased because of the insertion of \( v \)
Maintaining Invariants

Throughout this loop, we keep the following true:

- the subtrees of $tn$ are balanced (heights differ by no more than one); and
- $tn$ and all its subtrees have correct height values; and
- $heightIncreased$ is true if — and only if — the height of $tn$ really has increased because of the insertion of $v$
Maintaining Invariants

Throughout this loop, we keep the following true:

- the subtrees of \( t_n \) are balanced (heights differ by no more than one); and
- \( t_n \) and all its subtrees have correct height values; and
- \( \text{heightIncreased} \) is true if — and only if — the height of \( t_n \) really has increased because of the insertion of \( v \).
Maintaining Invariants

in method insert(int)

// check height and balance of tn
if (tn.height > tn.leftHeight() &&
    tn.height > tn.rightHeight()) {
    heightIncreased = false;
}

even though one subtree of tn has increased in height, the
height value of tn is correct
(the other subtree must have been taller)
exit the loop
Maintaining Invariants

in method insert(int)

// check height and balance of tn
if (tn.height > tn.leftHeight() &&
    tn.height > tn.rightHeight()) {
    heightIncreased = false;
}

even though one subtree of tn has increased in height, the height value of tn is correct
(the other subtree must have been taller)
exit the loop
Maintaining Invariants

in method insert(int)

// check height and balance of tn
if (tn.height > tn.leftHeight() &&
    tn.height > tn.rightHeight()) {
    heightIncreased = false;
}

even though one subtree of tn has increased in height, the height value of tn is correct
(the other subtree must have been taller)
exit the loop
Maintaining Invariants

in method insert(int)

// check height and balance of tn
if (tn.height > tn.leftHeight() &&
    tn.height > tn.rightHeight()) {
    heightIncreased = false;
}

even though one subtree of tn has increased in height, the height value of tn is correct
(the other subtree must have been taller)
exit the loop
Maintaining Invariants

in method insert(int)

// check height and balance of tn
if (tn.height > tn.leftHeight() &&
    tn.height > tn.rightHeight()) {
    heightIncreased = false;
}

even though one subtree of tn has increased in height, the height value of tn is correct
(the other subtree must have been taller)
exit the loop
Maintaining Invariants

else if (tn.leftHeight() - tn.rightHeight() > 1) {
    Boolean left;
    if (tn.parent != null) {
        left = tn.parent.leftSubtree == tn;
    }
}

The tree is unbalanced (bigger on the left):
find out if this is Case 1 or Case 2
But first find out if tn is the left or right subtree
(so we know where to put the rotated tree)
else if (tn.leftHeight() - tn.rightHeight() > 1) {
    Boolean left;
    if (tn.parent != null) {
        left = tn.parent.leftSubtree == tn;
    }
}

The tree is unbalanced (bigger on the left):
find out if this is Case 1 or Case 2
But first find out if tn is the left or right subtree
(so we know where to put the rotated tree)
Maintaining Invariants

else if (tn.leftHeight() - tn.rightHeight() > 1) {
    Boolean left;
    if (tn.parent != null) {
        left = tn.parent.leftSubtree == tn;
    }
}

The tree is unbalanced (bigger on the left):
find out if this is Case 1 or Case 2
But first find out if \(tn\) is the left or right subtree
(so we know where to put the rotated tree)
Maintaining Invariants

```java
else if (tn.leftHeight() - tn.rightHeight() > 1) {
    Boolean left;
    if (tn.parent != null) {
        left = tn.parent.leftSubtree == tn;
    }
}
```

The tree is unbalanced (bigger on the left):
find out if this is Case 1 or Case 2

But first find out if tn is the left or right subtree
(so we know where to put the rotated tree)
Maintaining Invariants

in method insert(int): Case 1 or 2?

```java
if (tn.leftSubtree.leftHeight() 
    > tn.leftSubtree.rightHeight()) {
    tn = leftRotate(tn);
} else {
    tn = doubleLeftRotate(tn);
}
```

Case 1 or Case 2
Do the appropriate rotation
Maintaining Invariants

in method insert(int): Case 1 or 2?

```java
if (tn.leftSubtree.leftHeight() > tn.leftSubtree.rightHeight()) {
    tn = leftRotate(tn);
} else {
    tn = doubleLeftRotate(tn);
}
```

Case 1 or Case 2
Do the appropriate rotation
Maintaining Invariants

in method insert(int): Case 1 or 2?

if (tn.leftSubtree.leftHeight() > tn.leftSubtree.rightHeight()) {
    tn = leftRotate(tn);
} else {
    tn = doubleLeftRotate(tn);
}

Case 1 or Case 2
Do the appropriate rotation
Maintaining Invariants

in method insert(int): Case 1 or 2?

if (tn.leftSubtree.leftHeight() > tn.leftSubtree.rightHeight()) {
    tn = leftRotate(tn);
} else {
    tn = doubleLeftRotate(tn);
}

Case 1 or Case 2
Do the appropriate rotation
...and put the rotated (balanced) tree in the appropriate place:

```java
if (tn.parent == null) {
    root = tn;
} else if (left) {
    tn.parent.leftSubtree = tn;
} else {
    tn.parent.rightSubtree = tn;
}
heightIncreased = false;
```

The rotation has decreased the height of the tree: exit the loop.
Maintaining Invariants

...and put the rotated (balanced) tree in the appropriate place:

```java
if (tn.parent == null) {
    root = tn;
} else if (left) {
    tn.parent.leftSubtree = tn;
} else {
    tn.parent.rightSubtree = tn;
}
heightIncreased = false;
```

The rotation has decreased the height of the tree: exit the loop
Maintaining Invariants

...and put the rotated (balanced) tree in the appropriate place:

```java
if (tn.parent == null) {
    root = tn;
} else if (left) {
    tn.parent.leftSubtree = tn;
} else {
    tn.parent.rightSubtree = tn;
}
heightIncreased = false;
```

The rotation has decreased the height of the tree: exit the loop
Maintaining Invariants

Cases 3 and 4 are handled symmetrically.

So far, what we’ve done is:

```plaintext
if height is okay
    exit the loop
else if tree is unbalanced
    balance it
    exit loop
```

The remaining case is that the height at $t_n$ has increased, and not unbalanced (so no rotations have been done to decrease the height).
Maintaining Invariants

Cases 3 and 4 are handled symmetrically.

So far, what we’ve done is:

```plaintext
if height is okay
    exit the loop
else if tree is unbalanced
    balance it
    exit loop
```

The remaining case is that the height at $t_n$ has increased, and not unbalanced (so no rotations have been done to decrease the height).
Maintaining Invariants

Cases 3 and 4 are handled symmetrically.

So far, what we’ve done is:

```
if height is okay
  exit the loop
else if tree is unbalanced
  balance it
  exit loop
```

The remaining case is that the height at tn has increased, and not unbalanced (so no rotations have been done to decrease the height).
So we simply increase the height, and carry on:

```java
else {
    tn.height++;
}
```
Summary

```
while tn is not null and height increased
  if tn’s height is okay
    exit the loop
  else if tn is unbalanced
    balance it
    exit loop
  else
    adjust tn’s height
  tn = tn.parent
```
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root
  (cf. start and end of doubly-linked lists)
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root
  (cf. start and end of doubly-linked lists)
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root
  (cf. start and end of doubly-linked lists)
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root (cf. start and end of doubly-linked lists)
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root
  (cf. start and end of doubly-linked lists)
Class Invariants

AVL trees give an example where we have several class invariants, which both create hard work for the programmer, but also provide guidance for the creation of code (the code has to maintain these invariants)

- small values to left; big values to right
- height values are correct
- tree is balanced
- pointers are correct (cf. doubly-linked lists)
- parent is null only for root
  (cf. start and end of doubly-linked lists)
Back to Queues

Recall the CardHand example from a long time ago. We implemented the list of Cards using an array, with pointers to the start and end of the list.

We wanted the Class Invariant

\[ 0 \leq startIndex \leq endIndex \leq \text{theCards.length} \]

(which would prevent ArrayIndexOutOfBoundsExceptions)
Back to Queues

Recall the CardHand example from a long time ago. We implemented the list of Cards using an array, with pointers to the start and end of the list.

```
| b1 | b2 | b3 | b4 | b5 | b6 | b7 |
```

We wanted the Class Invariant

\[
0 \leq startIndex \leq endIndex \leq theCards.length
\]

(which would prevent ArrayIndexOutOfBoundsException)
Invariants for Queues

This property was not a class invariant. It is possible to call methods that make this property false.

For example, adding a Card to a full array will increase the pointer at the end of the list, and attempt to assign to the array at that point (even with the ‘shuffle’ implementation).

Suppose the add(BandCard) method is called when startIndex is 0 and endIndex is theCards.length.
Invariants for Queues

This property was *not* a class invariant. It is possible to call methods that make this property false.

For example, adding a Card to a full array will increase the pointer at the end of the list, and attempt to assign to the array at that point (even with the ‘shuffle’ implementation).

Suppose the `add(BandCard)` method is called when `startIndex` is 0 and `endIndex` is `theCards.length`.
This property was *not* a class invariant. It is possible to call methods that make this property false.

For example, adding a Card to a full array will increase the pointer at the end of the list, and attempt to assign to the array at that point (even with the ‘shuffle’ implementation).

Suppose the `add(BandCard)` method is called when `startIndex` is 0 and `endIndex` is `theCards.length`.
add(Bandcard b)

// if there’s no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
// if there's no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex: ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there's no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0; i < len; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there’s no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0; i < len; i++) {
        theCards[i] = theCards[startIndex + i];
    }

    // move the pointers
    startIndex = 0;
    endIndex = len;
}

// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there's no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0; i < len; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there's no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there’s no room, make room
if (endIndex == theCards.length) {

    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }

    // move the pointers
    startIndex = 0;
    endIndex = len;
}

// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex:
ArrayIndexOutOfBoundsException
add(Bandcard b)

// if there's no room, make room
if (endIndex == theCards.length) {
    // how many times?
    int len = endIndex - startIndex;
    for (int i = 0 ; i < len ; i++) {
        theCards[i] = theCards[startIndex + i];
    }
    // move the pointers
    startIndex = 0;
    endIndex = len;
}
// add the card
theCards[endIndex++] = b;

len = endIndex; no change to endIndex: ArrayIndexOutOfBoundsException
Exceptions

To document this possibility, we can write a checked exception class, `CardHandOverflowException`, and change the implementation of `add(BandCard)` as follows:

```java
public void add(BandCard b)
    throws CardHandOverflowException
{
    if (endIndex - startIndex == theCards.length)
    {
        throw new CardHandOverflowException();
    }

    // as before
}
```
Exceptions

To document this possibility, we can write a checked exception class, `CardHandOverflowException`, and change the implementation of `add(BandCard)` as follows:

```java
public void add(BandCard b) throws CardHandOverflowException {
    if (endIndex - startIndex == theCards.length) {
        throw new CardHandOverflowException();
    }
    // as before
}
```
We change the definition of class invariant to allow for cases like this.

A property is a class invariant for a class $C$ if:

- all constructors of $C$: make the property true, and
- all public methods of $C$: preserve the property

or throw an exception.
We change the definition of class invariant to allow for cases like this.

A property is a class invariant for a class \( C \) if:

- all constructors of \( C \): make the property true, and
- all public methods of \( C \): preserve the property

or throw an exception.
Exercise

Write checked exception classes `StackOverflowException` and `StackUnderflowException` (for pushing values onto a full bounded stack, and popping values from an empty stack, respectively), and modify the implementation of bounded stacks from Lecture 5, so that the following are class invariants:

- $0 \leq \text{pointer}$
- $\text{pointer} \leq 100$
That’s All, Folks!

Summary

- AVL Trees
- Class Invariants . . .
- . . . and Exceptions

Next:

Generics