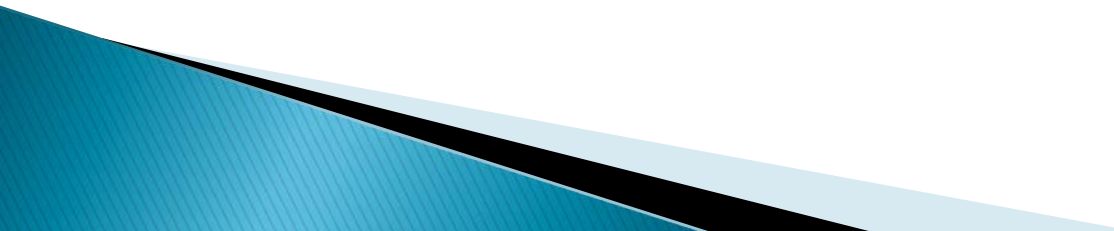


BINARY
SU DOKU

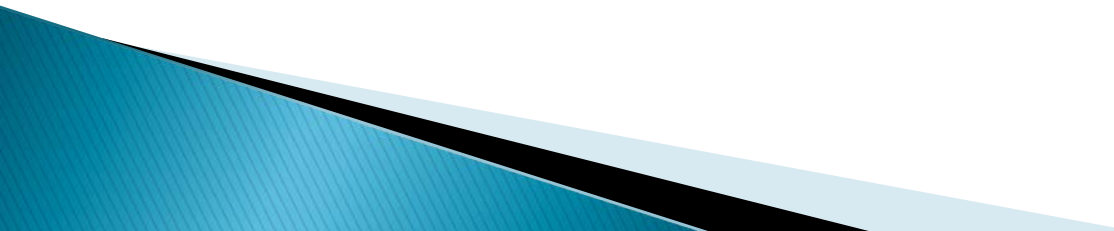
CSSE 230 Day 10

Binary Tree Iterators
and Properties
Displayable Binary Trees

Upcoming due dates: see schedule

- ▶ Exam 1
 - ▶ Hardy/Colorize, partner evals, too.
 - ▶ WA4 due
 - ▶ Displayable Binary Tree
- 

Announcements

- ▶ Exam 1 **Tonight 7–9 PM**
 - O-269 (Sec 1), O-267 (Sec 2)
 - ▶ Hardy/Colorize Partner Evaluation due **Friday at 5 PM**
 - ▶ WA4 due **Friday at 8:00 AM**
 - ▶ Displayable due **Tuesday, April 9 at 8 AM**
- 

Exam Announcements

▶ Exam 1

◦ Coverage:

- Everything from reading and lectures, Sessions 1–9
- Programs through Hardy + Colorize
- Written assignments 1–3

◦ Allowed resources:

- Written part: One side of one 8.5 x 11 sheet of paper
- Programming part:
 - Textbook
 - Eclipse (including programs in your repositories)
 - Course web pages and materials on ANGEL
 - Java API documentation
- A previous 230 Exam 1 is available on ANGEL

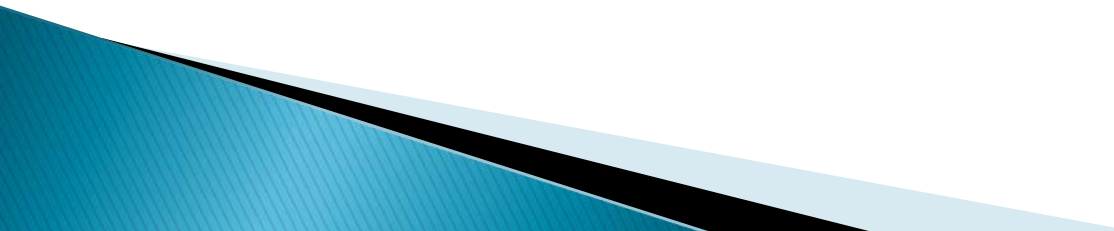
No devices with
headphones or
earbuds are
allowed

Exam 1 Possible Topics

▶ Sessions 1–11

- Terminology
- OOP and inheritance
- Growable Arrays
- Homework and Programs
- Big-oh, Big-Omega, and Big-Theta
- Limits and asymptotic behavior
- Basic data structures
- Comparable and Comparator
- MCSS
- Finite State Machines
- Recursion, stack frames
- Recursive binary search
- Binary trees
- Binary tree traversals
- Size vs. height for binary trees
- Binary Search Tree basics
- No induction problems yet.

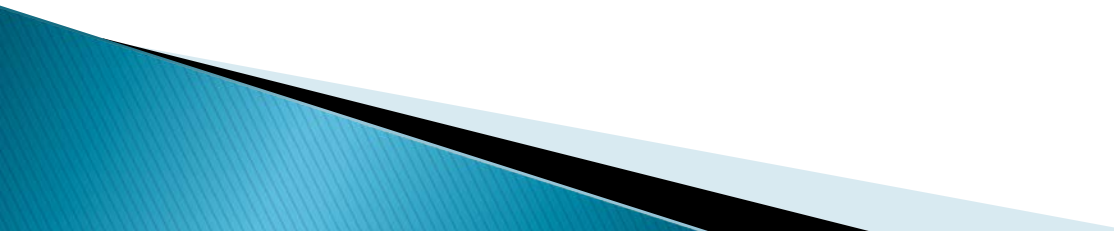
Agenda

- ▶ Binary Tree iterators
 - ▶ Another induction example
 - ▶ WA4 hints, questions
 - ▶ Displayable Binary Trees
 - ▶ # of nodes in Binary tree with height h
- 

Binary Tree Iterators

- »» What if we want to iterate over the elements in the nodes of the tree one-at-a-time instead of just printing all of them?

Implementing Binary Tree Iterators

- ▶ What methods does an iterator typically provide?
 - Weiss uses: *first*, *isValid*, *advance*, *retrieve*
 - ▶ In what order should we return the elements?
 - ▶ What instance variables do we need?
 - ▶ How do we get to the first item in:
 - a pre-order traversal?
 - an in-order traversal?
 - a post-order traversal?
- 

Weiss's Implementation

- ▶ See *TestTreeIterators* in the *Displayable* project.
- ▶ Most of the code is on the next slides.

Treeliterator abstract class

```
// TreeIterator class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// *****PUBLIC OPERATIONS*****
//     first and advance are abstract; others are final
// boolean isValid( )    --> True if at valid position in tree
// Object retrieve( )    --> Return item in current position
// void first( )         --> Set current position to first
// void advance( )       --> Advance (prefix)
// *****ERRORS*****
// Exceptions thrown for illegal access or advance
```

Treeliterator fields and methods

```
protected BinaryTree t;      // Tree
protected BinaryNode current; // Current position

public TreeIterator( BinaryTree theTree ) {
    t = theTree;
    current = null;
}

abstract public void first( );

final public boolean isValid( ) {
    return current != null;
}

final public Object retrieve( ) {
    if( current == null )
        throw new NoSuchElementException( );
    return current.getElement( );
}

abstract public void advance( );
```

Preorder: constructor and *first*

```
private Stack s;    // Stack of TreeNode objects

public PreOrder( BinaryTree theTree ) {
    super( theTree );
    s = new ArrayStack( );
    s.push( theTree.getRoot( ) );
}

public void first( ) {
    s.makeEmpty( );
    if( t.getRoot( ) != null )
        s.push( t.getRoot( ) );
    try
        { advance( ); }
    catch( NoSuchElementException e ) { } // Empty tree
}
```

PreOrder: *advance*

```
public void advance( ) {  
    if( s.isEmpty( ) ) {  
        if( current == null )  
            throw new NoSuchElementException( );  
        current = null;  
        return;  
    }  
  
    current = ( BinaryNode ) s.topAndPop( );  
  
    if( current.getRight( ) != null )  
        s.push( current.getRight( ) );  
    if( current.getLeft( ) != null )  
        s.push( current.getLeft( ) );  
}
```

LevelOrder: constructor and *first*

```
private Queue q;    // Queue of TreeNode objects

public LevelOrder( BinaryTree theTree )    {
    super( theTree );
    q = new ArrayQueue( );
    q.enqueue( t.getRoot( ) );
}

public void first( ) {
    q.makeEmpty( );
    if( t.getRoot( ) != null )
        q.enqueue( t.getRoot( ) );
    try
        { advance( ); }
    catch( NoSuchElementException e ) { } // Empty tree
}
```

Preorder: constructor and *first*

```
private Stack s;      // Stack of TreeNode objects

public PreOrder( BinaryTree theTree ) {
    super( theTree );
    s = new ArrayStack( );
    s.push( theTree.getRoot( ) );
}

public void first( ) {
    s.makeEmpty( );
    if( t.getRoot( ) != null )
        s.push( t.getRoot( ) );
    try
        { advance( ); }
    catch( NoSuchElementException e ) { } // Empty tree
}
```

LevelOrder: *advance*

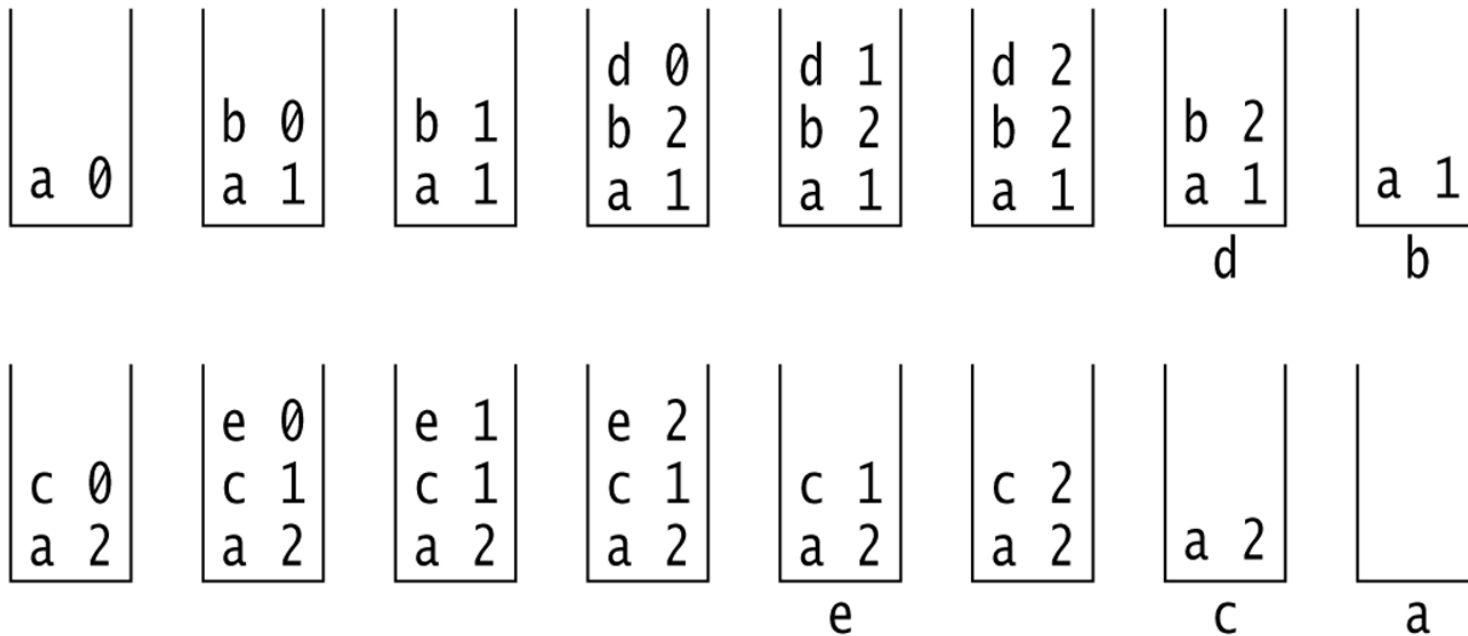
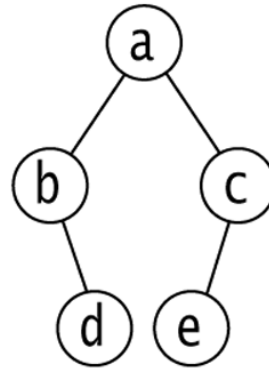
```
public void advance( ) {  
    if( q.isEmpty( ) ) {  
        if( current == null )  
            throw new NoSuchElementException( );  
        current = null;  
        return;  
    }  
  
    current = ( BinaryNode ) q.dequeue( );  
  
    if( current.getLeft( ) != null )  
        q.enqueue( current.getLeft( ) );  
    if( current.getRight( ) != null )  
        q.enqueue( current.getRight( ) );  
}
```


PreOrder: *advance*

```
public void advance( ) {  
    if( s.isEmpty( ) ) {  
        if( current == null )  
            throw new NoSuchElementException( );  
        current = null;  
        return;  
    }  
  
    current = ( BinaryNode ) s.topAndPop( );  
  
    if( current.getRight( ) != null )  
        s.push( current.getRight( ) );  
    if( current.getLeft( ) != null )  
        s.push( current.getLeft( ) );  
}
```

The Stack in a PostOrder iterator

Q1



Other Approaches to Tree Iterators

» Weiss's way isn't the only one

Alternative:

- ▶ Each node can store pointer to the next node in a traversal
- ▶ Must update extra info in constant time as tree changes

An upcoming written assignment will include these “threaded binary trees”

Wouldn't it be nice?

- ▶ If we did not have to maintain the stack for these iterators?
- ▶ If we could somehow “tap into” the stack used in the recursive traversal?
 - I.e. Take a “snapshot of that call stack, and restore it later when we need it.
 - This is called a **continuation**.
 - A big subject in the PLC course, CSSE 304

Python Makes This Easy

```
class BinaryTreeNode:
    def __init__(self, element, left, right):
        self.element = element
        self.left = left
        self.right = right

    def postorder(self):
        if self.left != None:
            for e in self.left.postorder():
                yield e
        if self.right != None:
            for e in self.right.postorder():
                yield e
        yield self.element
```

...

A lightweight form
of continuation

```
t = BinaryTree(...)
print "Post-order:"
for e in t.postorder():
    print e
```

Induction practice

»» Another example.

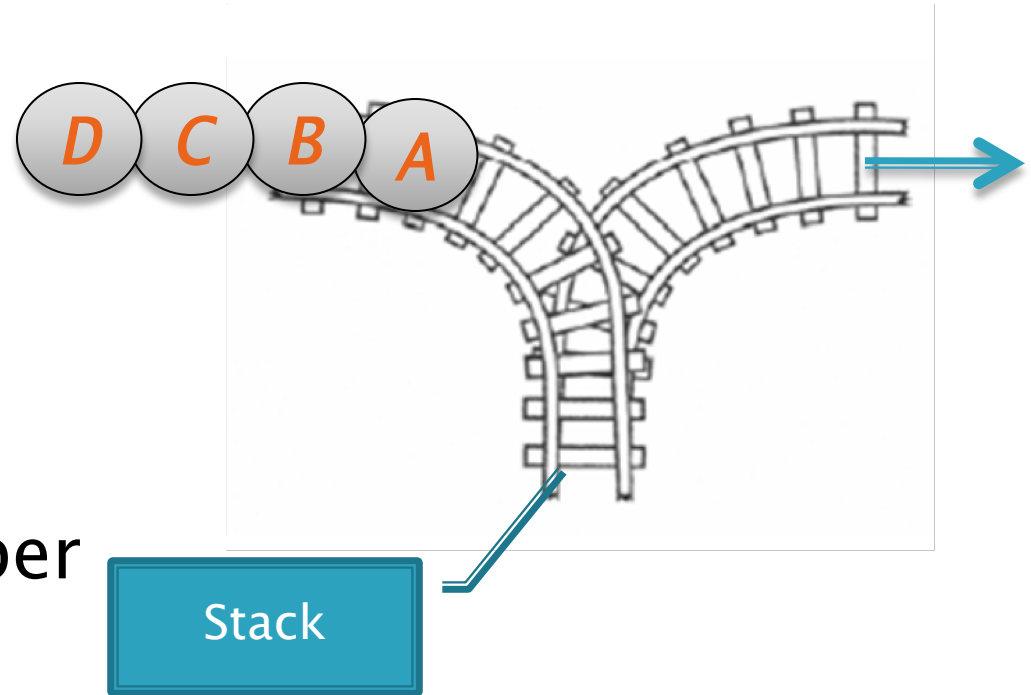
Another induction proof

- ▶ Show by induction that $n(n+1)(n+2)$ is divisible by 6 for all non-negative integers n .

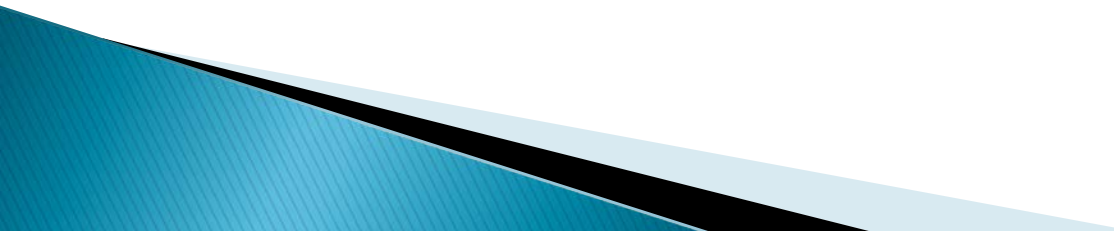
Tips on WA4

WA4, Problem 2 Application

- ▶ Railroad switching
- ▶ Problem is equivalent to counting the number of possible orders the cars can leave the station

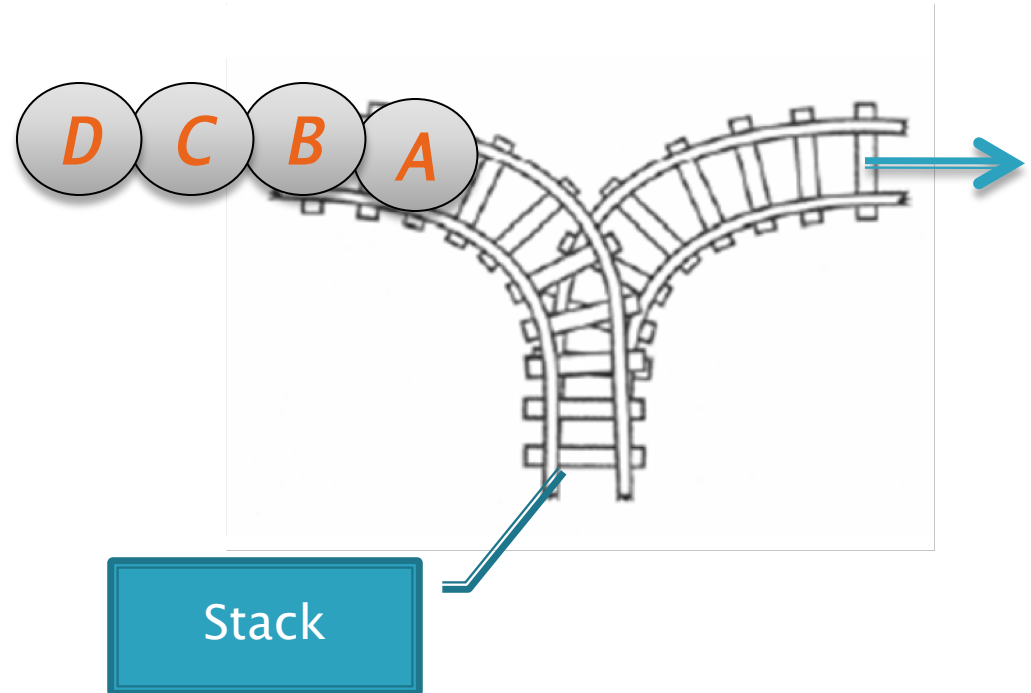


General Approach to Puzzle Problems

- ▶ Make up tiny examples like the given problem
 - No really tiny, I'm serious
 - ▶ Solve the tiny problem
 - ▶ Solve a slightly larger problem
 - ▶ Solve a slightly larger problem than that
 - ▶ Once you see the pattern, then try to solve the given problem
- 

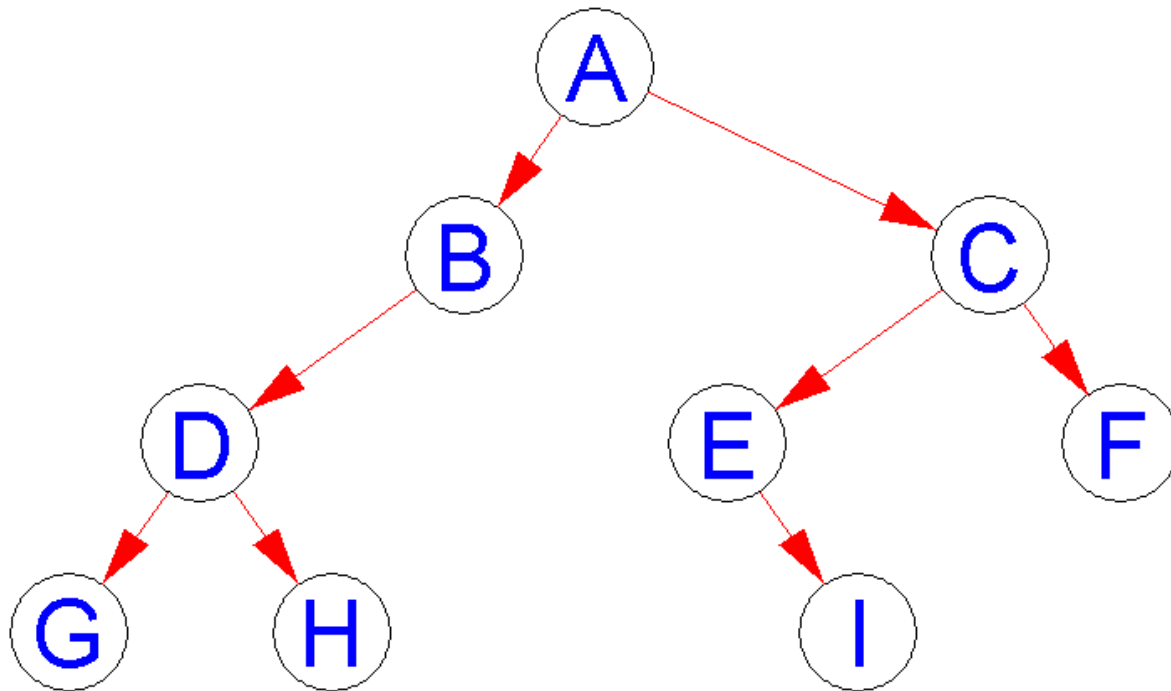
What's the smallest problem like this?

- ▶ In how many possible orders can the cars leave the station?



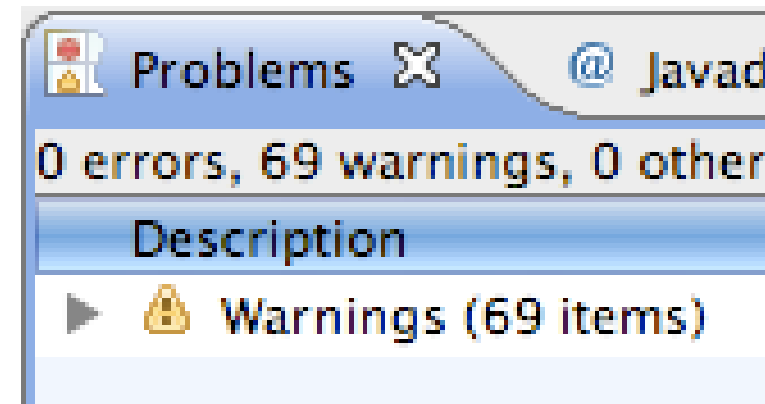
Displayable Binary Trees

»» Some suggestions



Displayable Binary Tree is an individual assignment

- ▶ Check out **Displayable** from your individual repo.
- ▶ If you get errors on the Weiss imports like
`import weiss.nonstandard.Stack;`
then install the Weiss packages now (see link from today on Schedule page)
- ▶ Should be no errors.
- ▶ If errors, see next slide.



Troubleshooting the Weiss install

- ▶ Close all Eclipse projects except *Displayable*
- ▶ Did you put jars in the right folder?
- ▶ Are the jars and not zips?
- ▶ Is Eclipse using that JRE?
 - See *Windows → Preferences*,
then *Java → Installed JREs → Edit*.
 - They should be in that list.

Get help now if you're stuck.
Help others if you aren't.

Displayable Binary Trees Steps

- ▶ Solve the sub-problems in this order:
 - *BuildTree.preOrderBuild()*
 - *BinaryTree.inOrder()*
 - Graphics
- ▶ Run *CheckDisplayableBinaryTree* to test
 - Doesn't use JUnit
 - Tests *preOrderBuild* and *inOrder* first
 - Prompts for test case for which to display graphics
 - Each tree should be displayed in a separate window.

Better Exception Reporting in `CheckDisplayableBinaryTrees`

- ▶ Add a stack trace in *main()*

```
70         tp.dbTree.display();
71     } catch (InternalError e) {
72         System.out
73             .println("You must i
74     }
75
76     }
77 } catch (Exception e) {
78     System.out.println(e.toString());
79     e.printStackTrace();
80 }
81
82 }
83
84 private static String inOrder(int index) {
85     switch (index) {
```

preOrderBuild Hints

- ▶ Like WA4, problem 3
- ▶ Consider:
 - *chars* = 'ROSEHULMAN'
 - *children* = '22002R0RLO'

inOrder Hints

- ▶ The iterators in TestTreeIterators.java are there for a reason!
- ▶ Recall how we can use Weiss iterators in a for loop:
 - ```
for(iter.first();iter.isValid();iter.advance()) {
 Object elem = iter.retrieve();
 // ... do something with elem ...
}
```

# Graphics Hints

- ▶ Suggested order for your graphics work:
  - Figure out how to calculate node locations
  - Get code to display correctly sized windows
  - Add code to draw nodes
  - Add code to draw lines
  - Only work on arrow heads if all the rest works!

# More Binary Trees

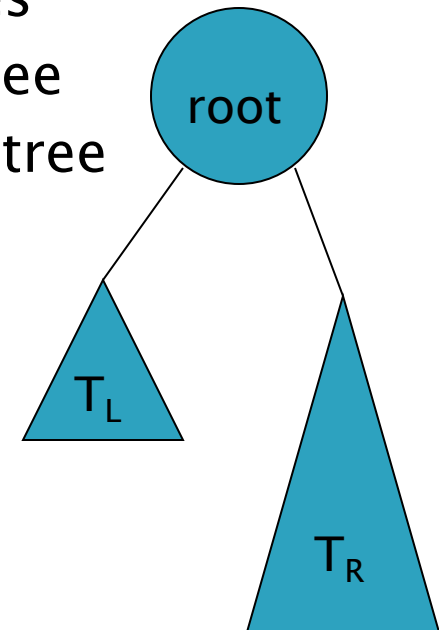
- » If a tree falls in the forest and there are two people around to hear it...

# Size vs. Height in Binary Trees



# Binary Tree: Recursive definition

- ▶ A Binary Tree is either
  - **empty**, or
  - **consists of**:
    - a distinguished node called the root, which contains an element, and two disjoint subtrees
    - A left subtree  $T_L$ , which is a binary tree
    - A right subtree  $T_R$ , which is a binary tree



# Time out for math!

- ▶ Want to prove some properties about trees
- ▶ Weak induction isn't enough
- ▶ Need strong induction instead:



The former  
governor of  
California



# Strong Induction

- ▶ To prove that  $p(n)$  is true for all  $n \geq n_0$ :
  - Prove that  $p(n_0)$  is true, and
  - For all  $k > n_0$ , prove that if we assume  $p(j)$  is true for  $n_0 \leq j < k$ , then  $p(k)$  is also true
- ▶ Weak induction uses the previous domino to knock down the next
- ▶ Strong induction uses a whole box of dominoes!

# Size and Height of Binary Trees

- ▶ Notation:
  - Let  $T$  be a tree
  - Write  $h(T)$  for the height of the tree, and
  - $N(T)$  for the size (i.e., number of nodes) of the tree
- ▶ Given  $h(T)$ , what are the bounds on  $N(T)$ ?
- ▶ Given  $N(T)$ , what are the bounds on  $h(T)$ ?

# Extreme Trees

- ▶ A tree with the maximum number of nodes for its height is a **full tree**.
  - Its height is  $O(\log N)$
- ▶ A tree with the minimum number of nodes for its height is essentially a \_\_\_\_\_
  - Its height is  $O(N)$
- ▶ Height matters!
  - We will see that the algorithms for search, insertion, and deletion in a Binary search tree are  $O(h(T))$

# Merge Method (from Weiss chapter 18)

- ▶ */\*\* Replaces the root element of this  
\* tree with the given item and the  
\* subtrees with the given ones.  
\* ... \*/*

*public void merge(T rootItem,  
BinaryTree<T> left,  
BinaryTree<T> right)*

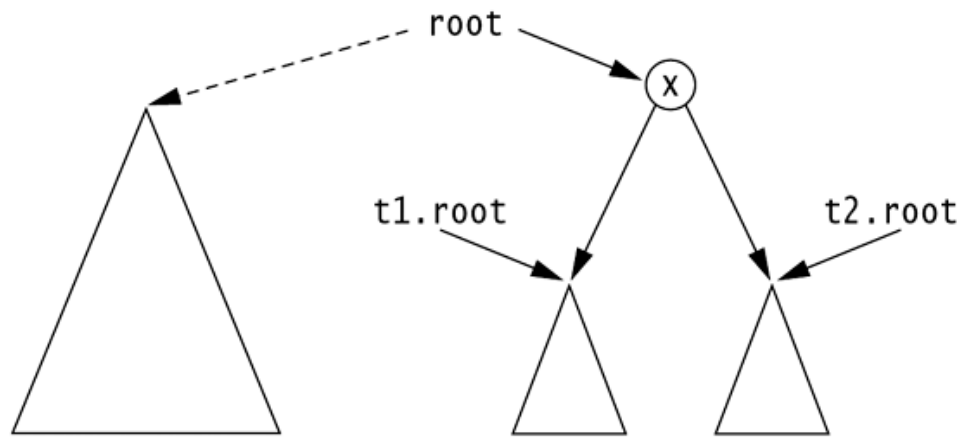
- ▶ Simple approach:

- *this.root = new BinaryTreeNode<T>(rootItem,  
left.root,  
right.root);*

What could go wrong?

# Problems With Naïve Merge

- ▶ A node should be part of one and only one tree.

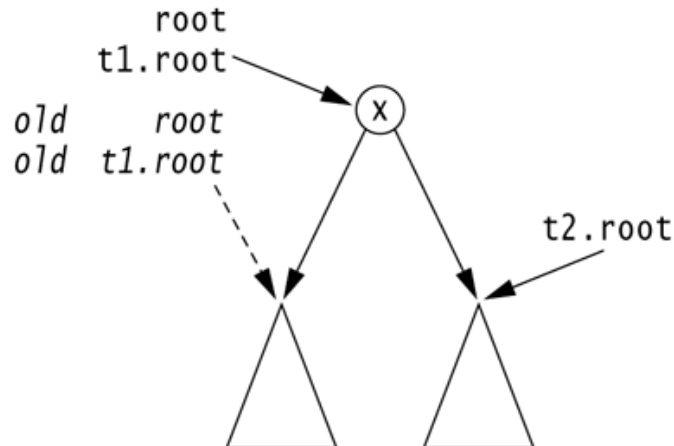


**figure 18.14**

Result of a naive merge operation:  
Subtrees are shared.

**figure 18.15**

Aliasing problems in the merge operation;  
t1 is also the current object.



# Correct Merge Method

```
1 /**
2 * Merge routine for BinaryTree class.
3 * Forms a new tree from rootItem, t1 and t2.
4 * Does not allow t1 and t2 to be the same.
5 * Correctly handles other aliasing conditions.
6 */
7 public void merge(AnyType rootItem,
8 BinaryTree<AnyType> t1, BinaryTree<AnyType> t2)
9 {
10 if(t1.root == t2.root && t1.root != null)
11 throw new IllegalArgumentException();
12
13 // Allocate new node
14 root = new BinaryNode<AnyType>(rootItem, t1.root, t2.root);
15
16 // Ensure that every node is in one tree
17 if(this != t1)
18 t1.root = null;
19 if(this != t2)
20 t2.root = null;
21 }
```

Weiss, figure 18.16