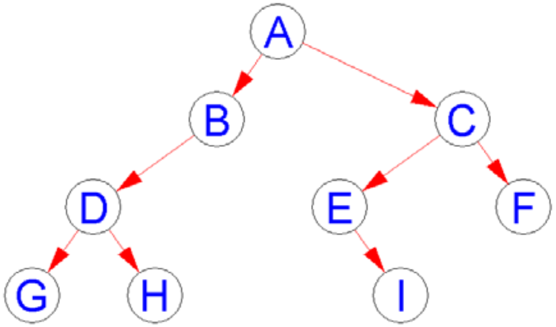
CSSE 230 Day 10

Binary Tree Iterators
and Properties

Displayable Binary Trees



Displayable Binary Trees
(next assignment)

Picnic time tomorrow afternoon!

DEPARTMENT OF COMPUTER SCIENCE & SOFTWARE ENGINEERING

PICNIC

SATURDAY SEPTEMBER 28, 2013

HAWTHORN PARK
BURKEYBYLE SHELTER

3:00 – 6:00 P.M.

Outdoor games at 3:00 p.m.

Food served 4:30 p.m.

Food catered by Qdoba Mexican Grill

FOOD SPONSORED BY X-by-2

Agenda

- Displayable Binary Trees
- Binary Tree iterators
- Another induction example
- WA4 hints, questions

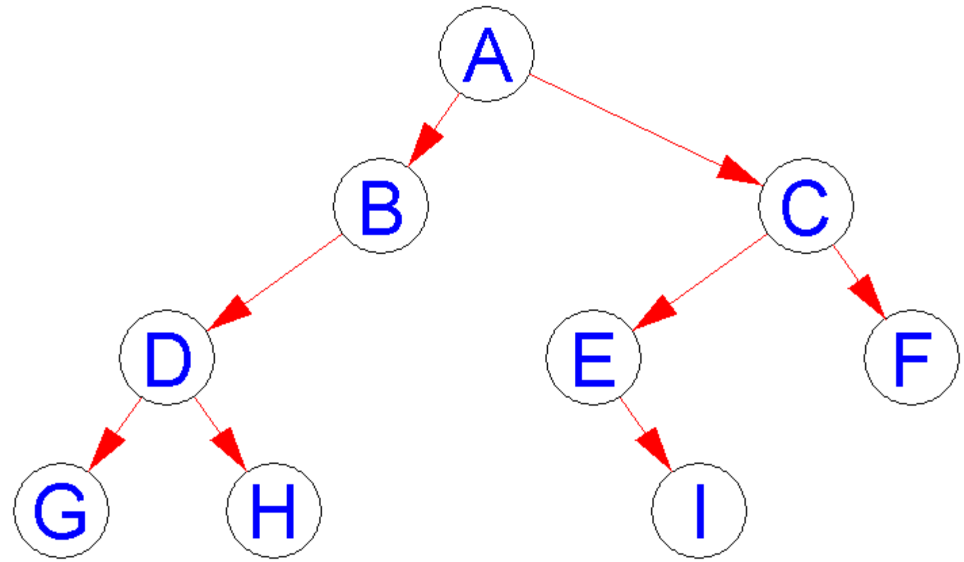
- # of nodes in Binary tree with height h

Displayable Binary Trees

Individual assignment

Gist of the assignment

- Levels all spaced evenly.
- Level-to-right spaced evenly, ordered by in-order traversal.
- Node and font sizes depend on window size



See demo of solution, trees #9 and #7

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!

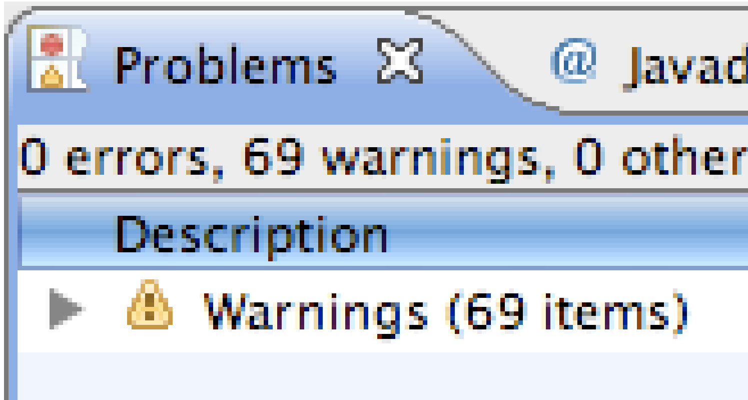
Hints and getting started

- 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 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 they 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 *CheckDisplaybleBinaryTree* 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()*

```
                    tp.dbTree.display();
                } catch (InternalError e) {
                    System.out
                            .println("You must
                }

            }
        } catch (Exception e) {
            System.out.println(e.toString());
            e.printStackTrace();
        }

    }

    private static String inOrder(int index) {
        switch (index) {
```

preOrderBuild Hints

- Like WA4, problem 3
- Consider:
  - *chars = ‘ROSEHULMAN’*
  - *children = ‘22002R0RL0’*

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 …
    }
    ```
    ```

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.

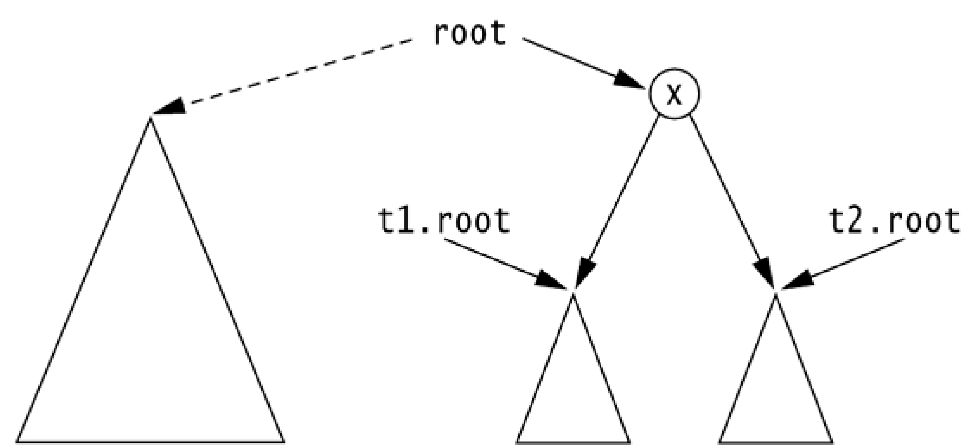
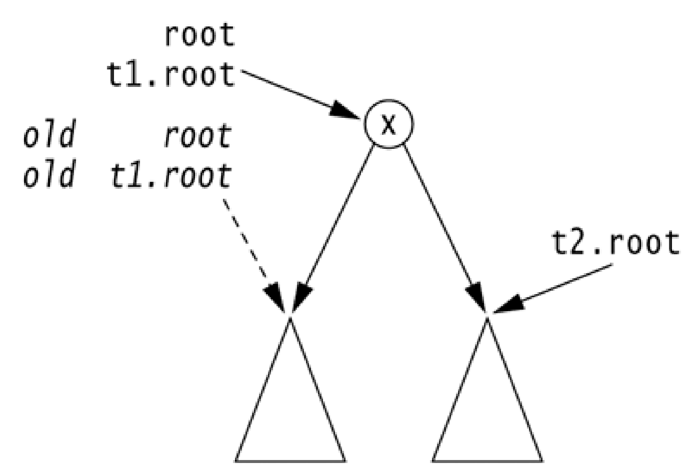


figure 18.14

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

```java
/**
 * Merge routine for BinaryTree class.
 * Forms a new tree from rootItem, t1 and t2.
 * Does not allow t1 and t2 to be the same.
 * Correctly handles other aliasing conditions.
 */
public void merge( AnyType rootItem,
                   BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
{
    if( t1.root == t2.root && t1.root != null )
        throw new IllegalArgumentException( );

        // Allocate new node
    root = new BinaryNode<AnyType>( rootItem, t1.root, t2.root );

        // Ensure that every node is in one tree
    if( this != t1 )
        t1.root = null;
    if( this != t2 )
        t2.root = null;
}
```

Weiss, figure 18.16

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?

Big questions

- How do you “slow down” recursion to be one step at a time?
  - Hint: what data structure is used to hold recursive calls at runtime?

- How many times is each node visited in a traversal/iterator?
  - See the visualization linked to from day 9 schedule.

Implementing Binary Tree Iterators

- What methods does an iterator typically provide?
- How do we get to the first item in:
  - a pre-order traversal?
  - an in-order traversal?
  - a post-order traversal?
- In what order should we advance?
- What instance variables do we need?

- The *`Displayable`* project has Weiss's *`TestTreeIterators`* implementation
  - Most of the code is on the next slides.

TreeIterator 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
```

TreeIterator 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*

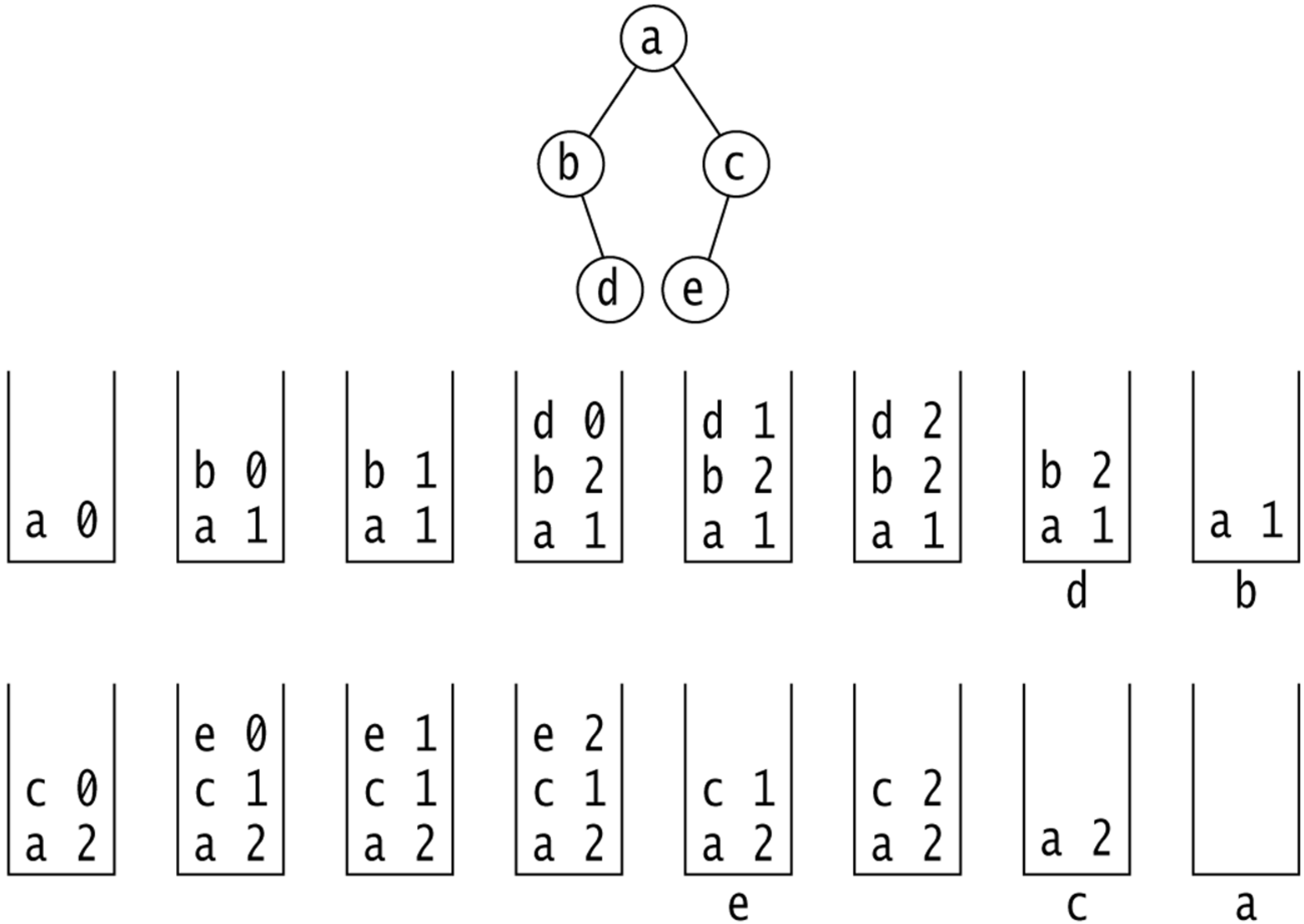
```
public void advance( ) {
   if( s.isEmpty( ) )     {
       if( current == null )
           throw new NoSuchElementException( );
       current = null;
       return;
   }

   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

Q1–4



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”

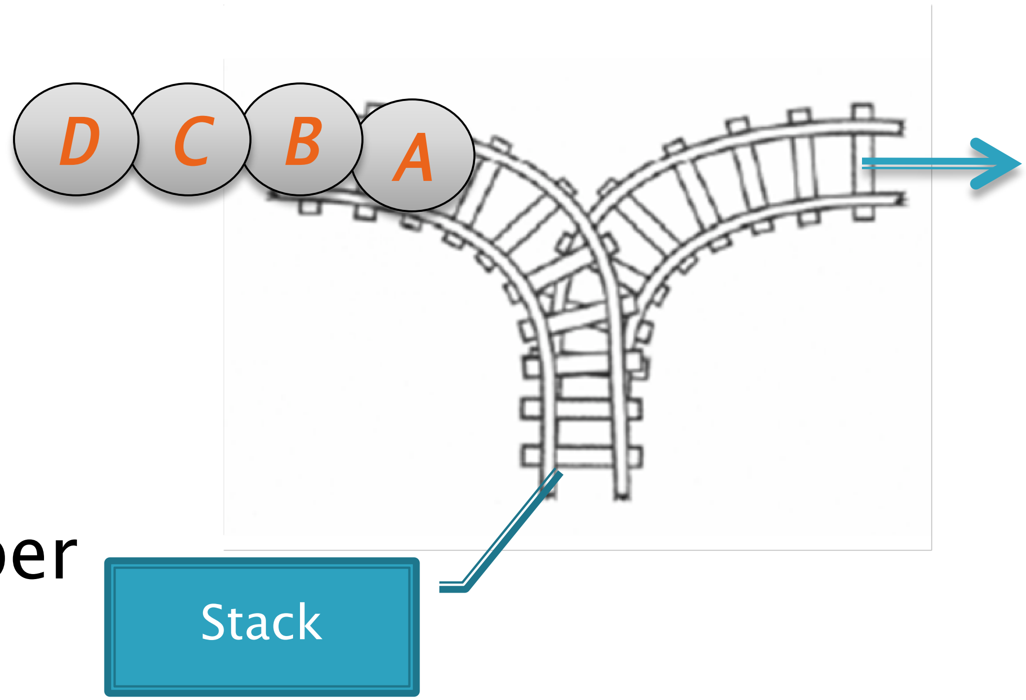
Wouldn't it be nice?

- 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

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

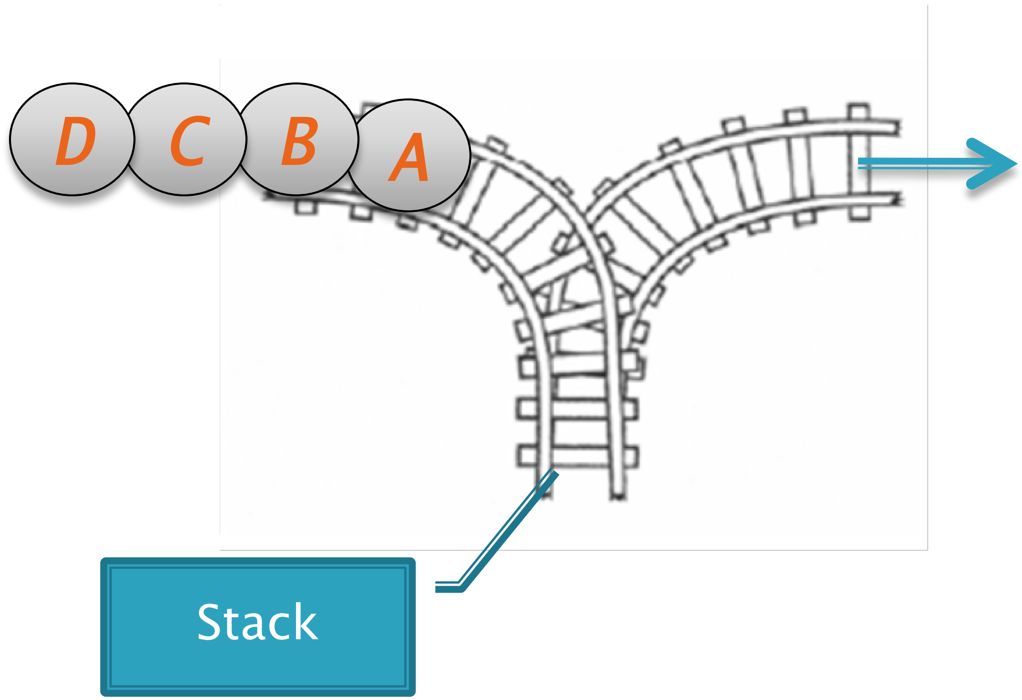
- Make up tiny examples like the given problem
  - No really tiny, I'm serious

- 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?



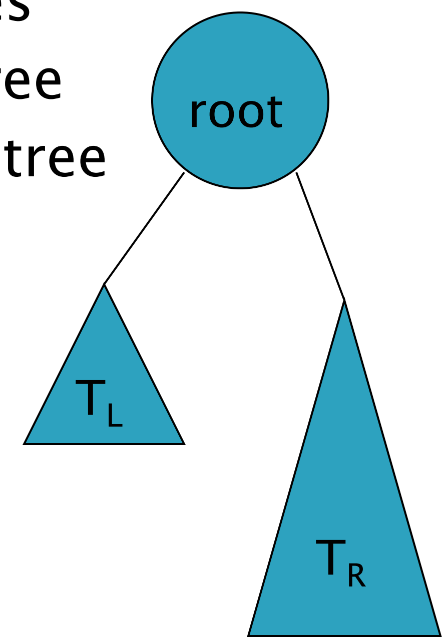
More Binary Trees

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



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))

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 >= 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 $n_0 \le j < k$, 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 to knock down the rest!