CSSE 230 Day 9

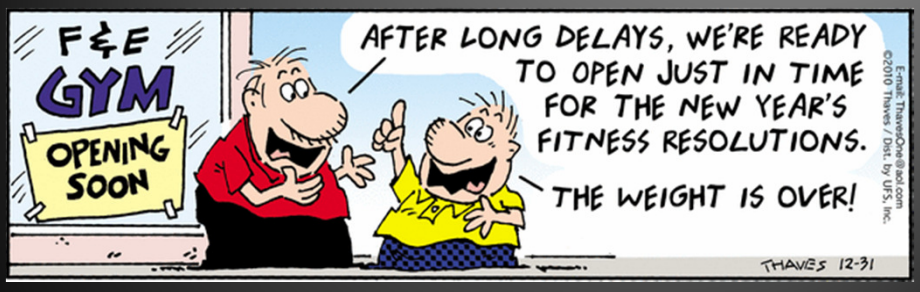
More simple BinaryTree methods
Tree Traversals and Iterators

Reminders/Announcements

- Hardy/Colorize programs due **Monday**.
- Exam 1 **Wednesday 7 PM (O267-269)**
  - Coverage:
    - Everything from reading and lectures, Sessions 1-10
    - 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 workspace 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

Questions?



Agenda

- Another induction example
- Implementing Binary Trees
- Binary Tree Traversals
- Binary Tree Iterators

Another induction proof example

Show by induction that $2n + 1 < n^2$ for all integers $n \geq 3$

There are other ways that we could show this (using calculus, for example)

But for now the goal is to have another example that can illustrate how to do proofs by induction

2

Growing Trees

- Let's continue implementing a `BinaryTree<T>` class including methods `size()`, `height()`, `duplicate()`, and `contains(T)`.

3-6

Binary tree traversals

- PreOrder (top-down, depth-first)
  - root, left, right
- PostOrder (bottom-up)
  - left, right, root
- InOrder (left-to-right, if tree is spread out)
  - Left, root, right
- LevelOrder (breadth-first)
  - Level-by-level, left-to-right within each level

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?

7-8

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?

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( ) );
}
```

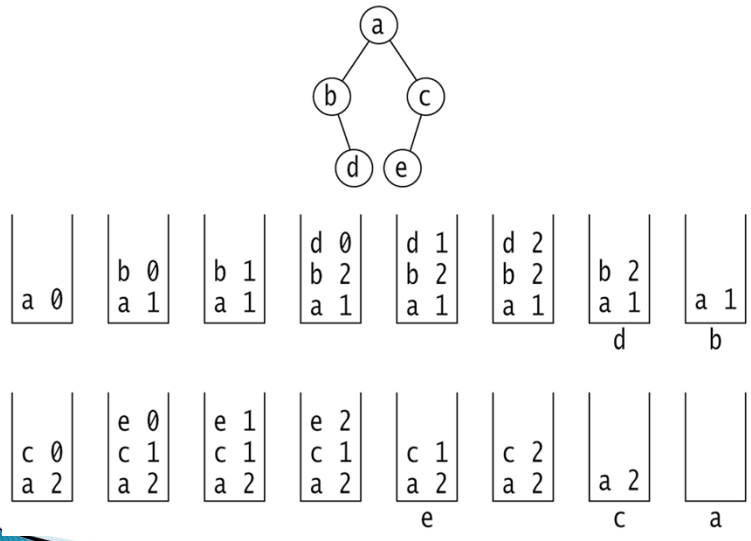
PreOrder: *advance*

```
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



9-11

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