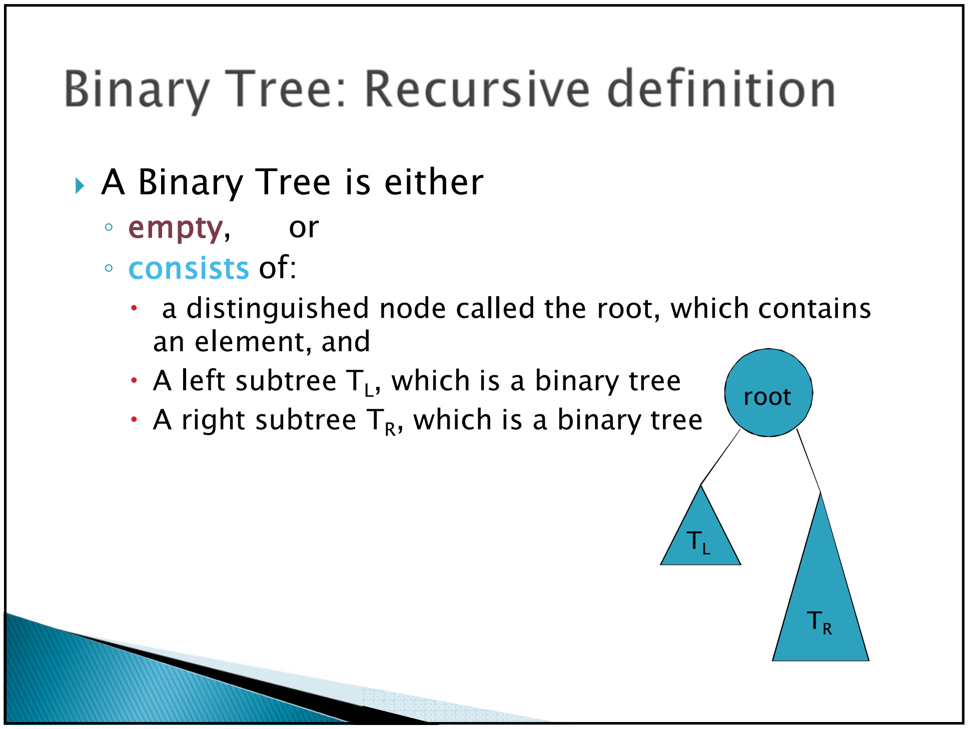
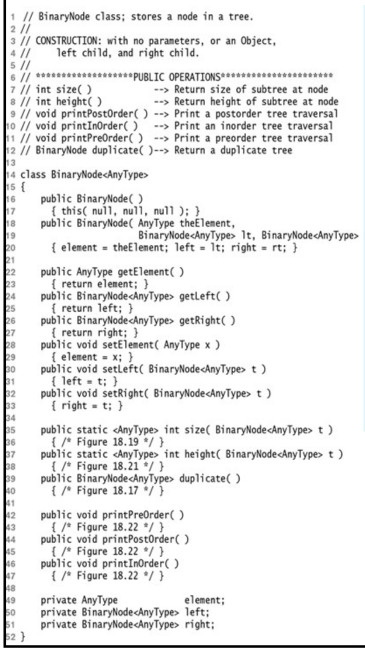
Binary Trees



Binary Tree: Recursive definition

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

Binary Tree implementation



```
// BinaryNode class; stores a node in a tree.
//
// CONSTRUCTION: with no parameters, or an Object,
//     left child, and right child.
//
// *******************PUBLIC OPERATIONS**********************
// int size( )            --> Return size of subtree at node
// int height( )          --> Return height of subtree at node
// void printPostOrder( ) --> Print a postorder tree traversal
// void printInOrder( )   --> Print an inorder tree traversal
// void printPreOrder( )  --> Print a preorder tree traversal
// BinaryNode duplicate( )--> Return a duplicate tree

class BinaryNode<AnyType>
{
    public BinaryNode( )
      { this( null, null, null ); }
    public BinaryNode( AnyType theElement,
                       BinaryNode<AnyType> lt, BinaryNode<AnyType>
      { element = theElement; left = lt; right = rt; }

    public AnyType getElement( )
      { return element; }
    public BinaryNode<AnyType> getLeft( )
      { return left; }
    public BinaryNode<AnyType> getRight( )
      { return right; }
    public void setElement( AnyType x )
      { element = x; }
    public void setLeft( BinaryNode<AnyType> t )
      { left = t; }
    public void setRight( BinaryNode<AnyType> t )
      { right = t; }

    public static <AnyType> int size( BinaryNode<AnyType> t )
      { /* Figure 18.19 */ }
    public static <AnyType> int height( BinaryNode<AnyType> t )
      { /* Figure 18.21 */ }
    public BinaryNode<AnyType> duplicate( )
      { /* Figure 18.17 */ }

    public void printPreOrder( )
      { /* Figure 18.22 */ }
    public void printPostOrder( )
      { /* Figure 18.22 */ }
    public void printInOrder( )
      { /* Figure 18.22 */ }

    private AnyType             element;
    private BinaryNode<AnyType> left;
    private BinaryNode<AnyType> right;
}
```

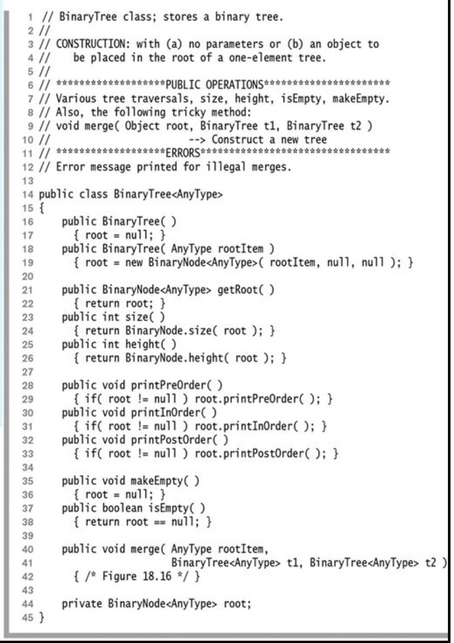
Build up a particular tree by calling the constructor.

size, height for tree, node

duplicate

contains

merge



```
// BinaryTree class; stores a binary tree.
//
// CONSTRUCTION: with (a) no parameters or (b) an object to
//    be placed in the root of a one-element tree.
//
// *******************PUBLIC OPERATIONS**********************
// Various tree traversals, size, height, isEmpty, makeEmpty.
// Also, the following tricky method:
// void merge( Object root, BinaryTree t1, BinaryTree t2 )
//                                --> Construct a new tree
// *******************ERRORS*********************************
// Error message printed for illegal merges.

public class BinaryTree<AnyType>
{
    public BinaryTree( )
      { root = null; }
    public BinaryTree( AnyType rootItem )
      { root = new BinaryNode<AnyType>( rootItem, null, null ); }

    public BinaryNode<AnyType> getRoot( )
      { return root; }
    public int size( )
      { return BinaryNode.size( root ); }
    public int height( )
      { return BinaryNode.height( root ); }

    public void printPreOrder( )
      { if( root != null ) root.printPreOrder( ); }
    public void printInOrder( )
      { if( root != null ) root.printInOrder( ); }
    public void printPostOrder( )
      { if( root != null ) root.printPostOrder( ); }

    public void makeEmpty( )
      { root = null; }
    public boolean isEmpty( )
      { return root == null; }

    public void merge( AnyType rootItem,
                       BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
      { /* Figure 18.16 */ }

    private BinaryNode<AnyType> root;
}
```

Merge method

- Public void merge(AnyType rootItem,
  BinaryTree<AnyType > t1,
  BinaryTree<AnyType > t2)
- Simple approach:
  - this.root = new BinaryNode<AnyType>(rootItem,
    t1.root,
    t2.root);
  - What can go wrong?

Problems With Naïve Merge

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

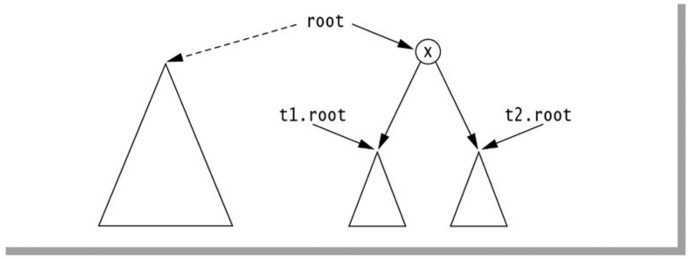
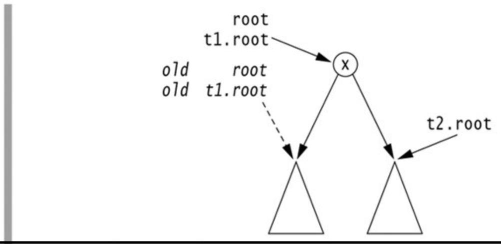


figure 18.14

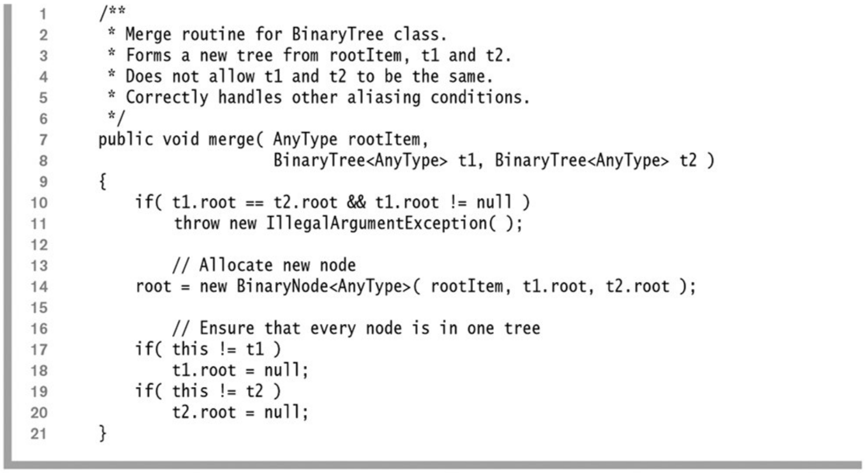
Result of a naive `merge` operation: Subtrees are shared.

figure 18.15

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



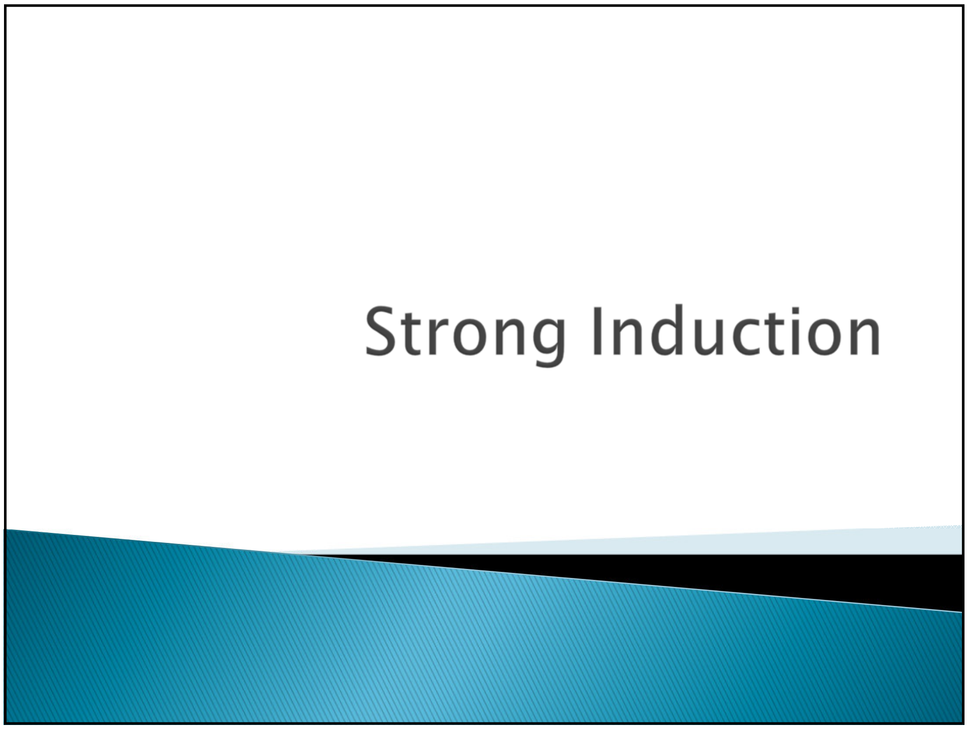
Correct Merge Method



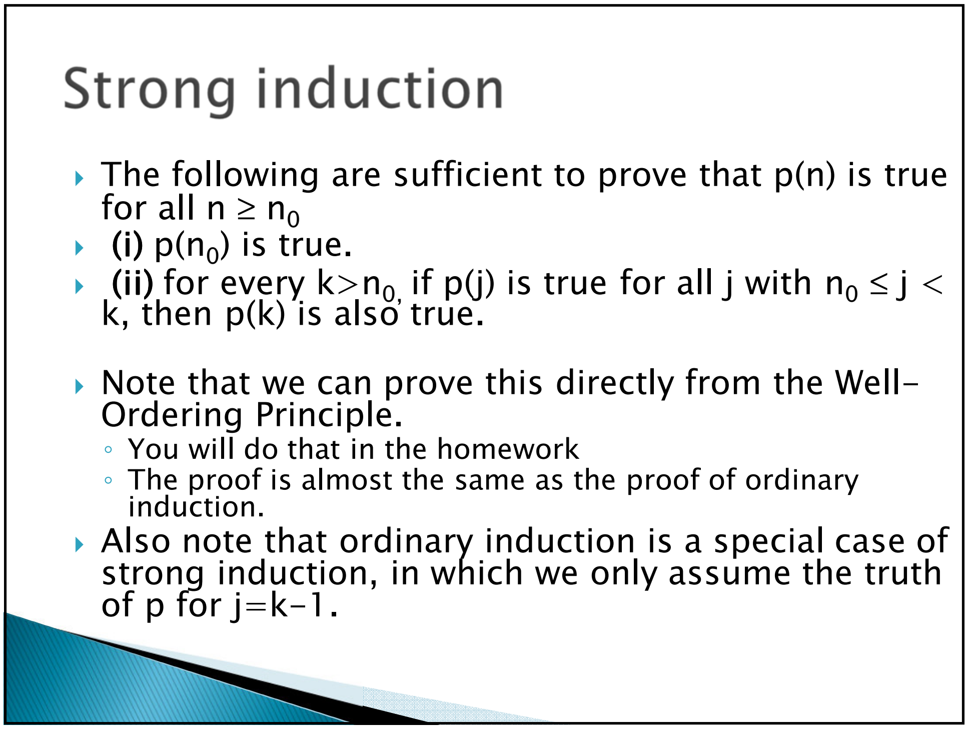
```
/**
 * Merge routine for BinaryTree class.
 * Forms a new tree from rootItem, t1 and t2.
 * Does not allow t1 and t2 to be the same.
 * Correctly handles other aliasing conditions.
 */
public void merge( AnyType rootItem,
                   BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
{
    if(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

figure 18.16

The `merge` routine for the `BinaryTree` class



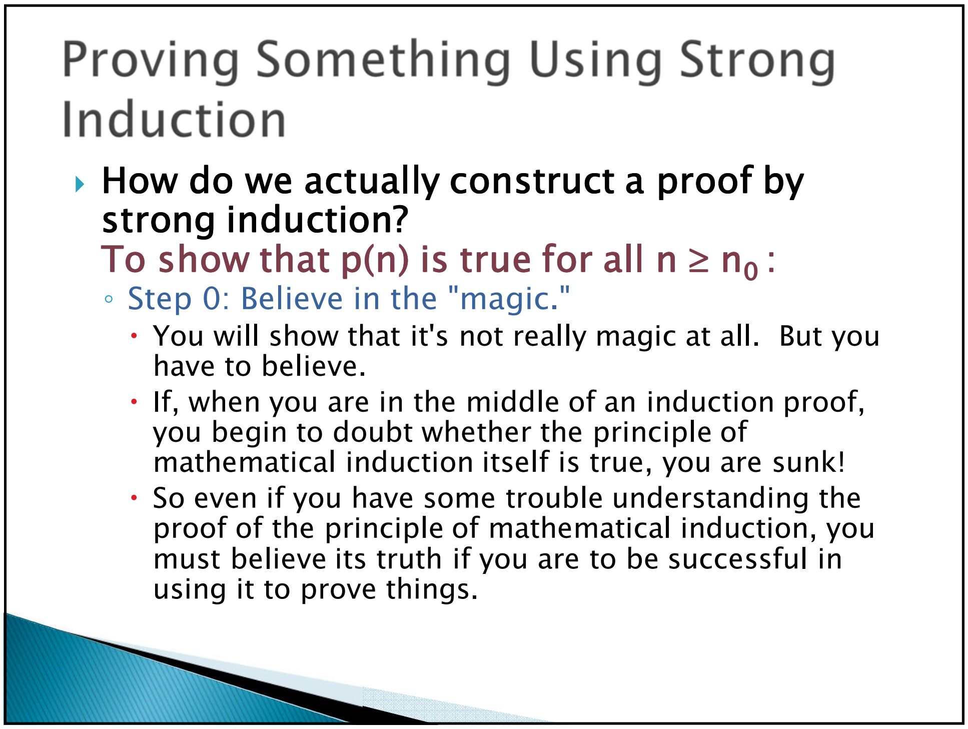
Strong Induction



Strong induction

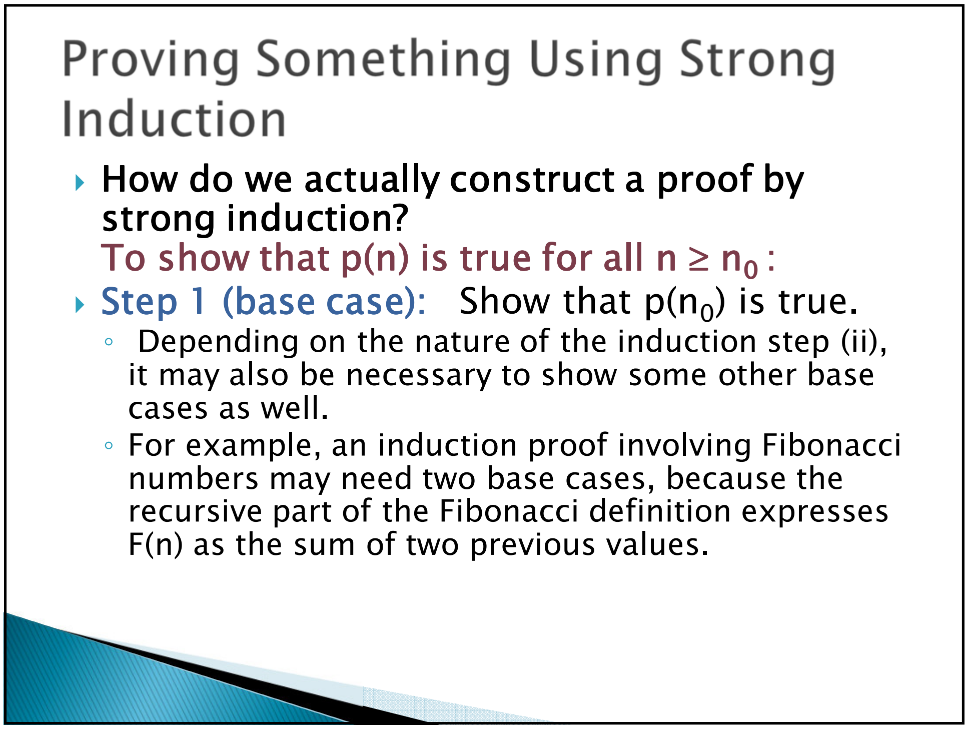
- The following are sufficient to prove that p(n) is true for all $n \geq n_0$
- **(i)** $p(n_0)$ is true.
- **(ii)** for every $k>n_0$, if p(j) is true for all j with $n_0 \leq j < k$, then p(k) is also true.

- Note that we can prove this directly from the Well-Ordering Principle.
  - You will do that in the homework
  - The proof is almost the same as the proof of ordinary induction.
- Also note that ordinary induction is a special case of strong induction, in which we only assume the truth of p for $j=k-1$.



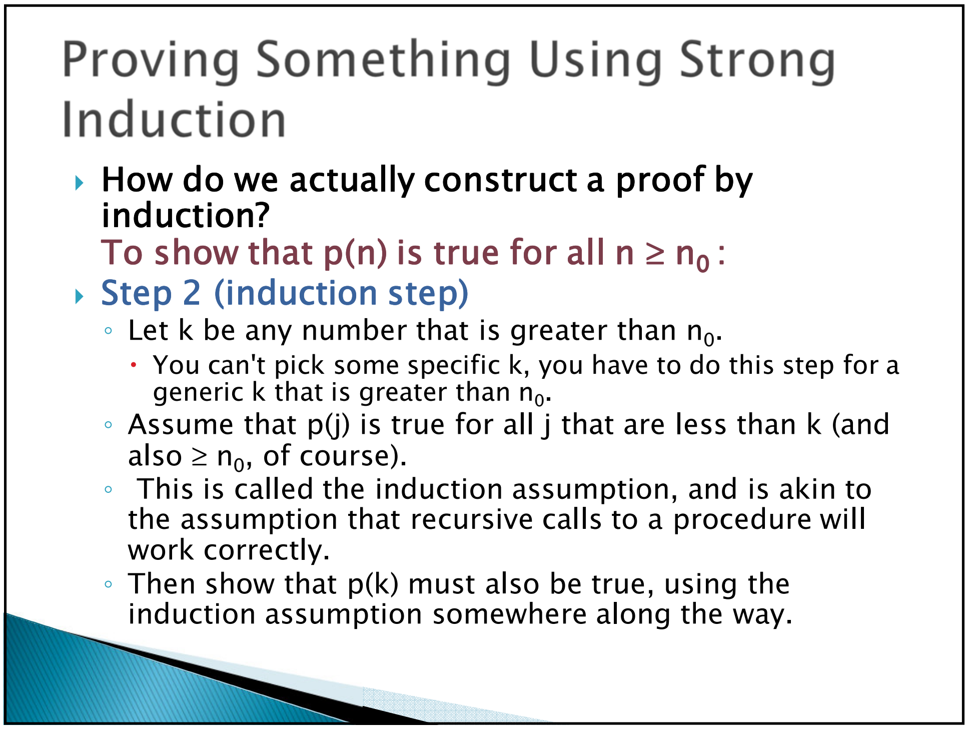
Proving Something Using Strong Induction

- **How do we actually construct a proof by strong induction?**
  To show that p(n) is true for all n ≥ n_0 :
  - Step 0: Believe in the "magic."
    - You will show that it's not really magic at all. But you have to believe.
    - If, when you are in the middle of an induction proof, you begin to doubt whether the principle of mathematical induction itself is true, you are sunk!
    - So even if you have some trouble understanding the proof of the principle of mathematical induction, you must believe its truth if you are to be successful in using it to prove things.



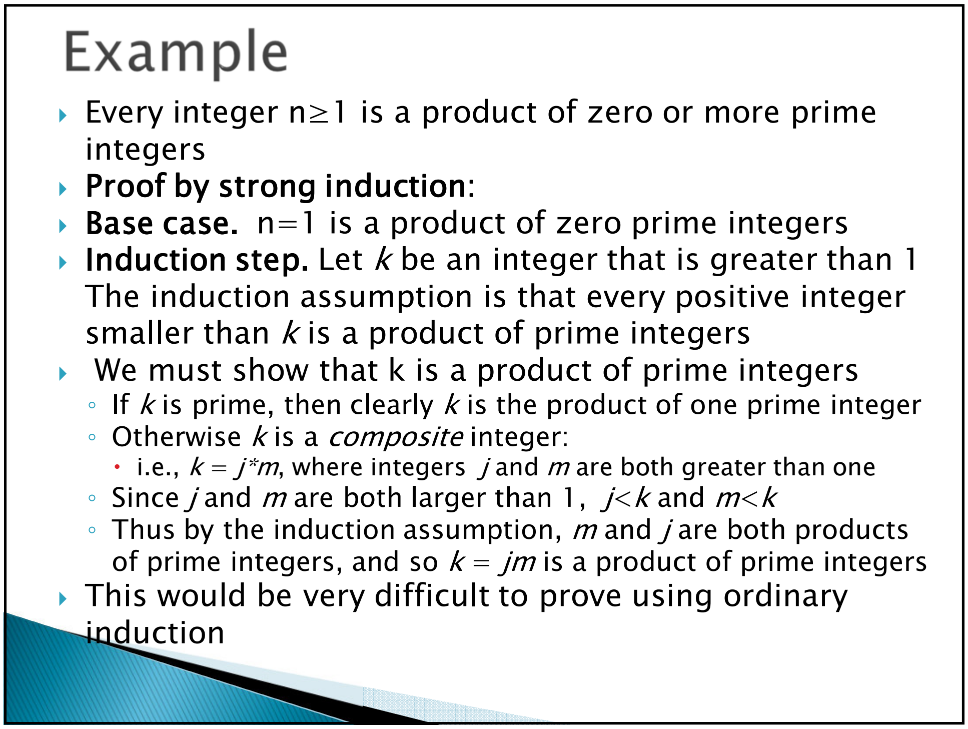
Proving Something Using Strong Induction

- **How do we actually construct a proof by strong induction?**
  To show that p(n) is true for all n ≥ n_0 :
- Step 1 (base case): Show that p(n_0) is true.
  - Depending on the nature of the induction step (ii), it may also be necessary to show some other base cases as well.
  - For example, an induction proof involving Fibonacci numbers may need two base cases, because the recursive part of the Fibonacci definition expresses F(n) as the sum of two previous values.



Proving Something Using Strong Induction

- **How do we actually construct a proof by induction?**
  To show that p(n) is true for all n ≥ n_0 :
- Step 2 (induction step)
  - Let k be any number that is greater than n_0.
    - You can't pick some specific k, you have to do this step for a generic k that is greater than n_0.
  - Assume that p(j) is true for all j that are less than k (and also ≥ n_0, of course).
  - This is called the induction assumption, and is akin to the assumption that recursive calls to a procedure will work correctly.
  - Then show that p(k) must also be true, using the induction assumption somewhere along the way.

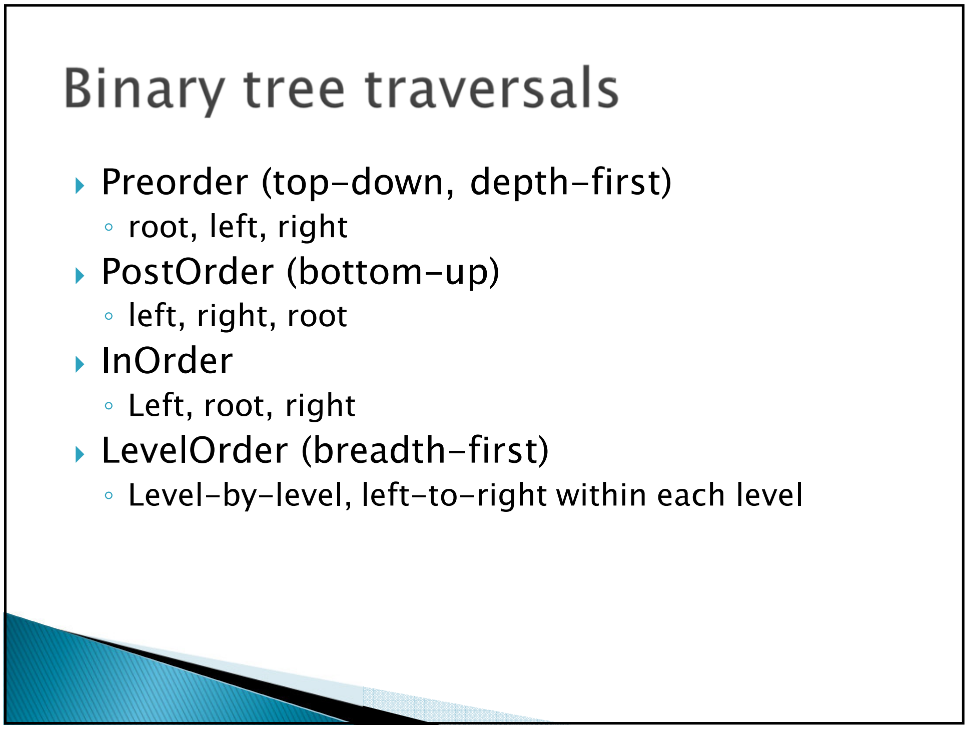


Example

- Every integer n≥1 is a product of zero or more prime integers
- **Proof by strong induction**:
- **Base case.** n=1 is a product of zero prime integers
- **Induction step.** Let *k* be an integer that is greater than 1 The induction assumption is that every positive integer smaller than *k* is a product of prime integers
- We must show that k is a product of prime integers
  - If *k* is prime, then clearly *k* is the product of one prime integer
  - Otherwise *k* is a *composite* integer:
    - i.e., *k = j*m*, where integers *j* and *m* are both greater than one
  - Since *j* and *m* are both larger than 1, *j<k* and *m<k*
  - Thus by the induction assumption, *m* and *j* are both products of prime integers, and so *k = jm* is a product of prime integers
- This would be very difficult to prove using ordinary induction

Binary Tree Traversals

PreOrder, PostOrder, InOrder, LevelOrder

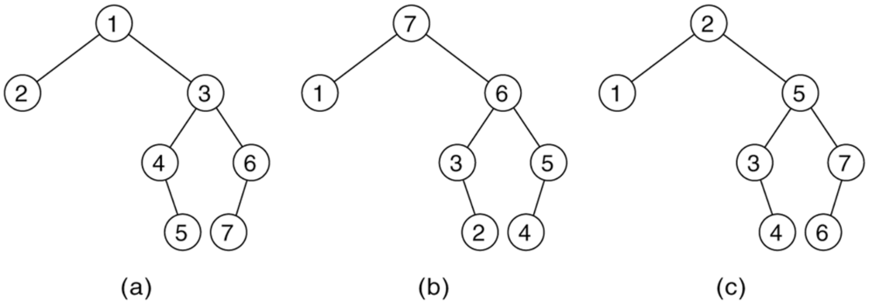


Binary tree traversals

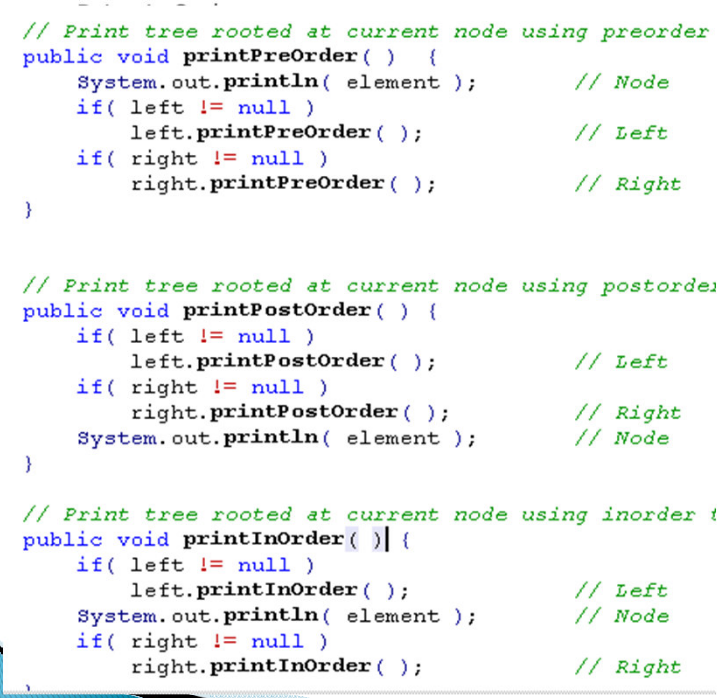
- Preorder (top-down, depth-first)
  - root, left, right
- PostOrder (bottom-up)
  - left, right, root
- InOrder
  - Left, root, right
- LevelOrder (breadth-first)
  - Level-by-level, left-to-right within each level

Tree Traversal orders

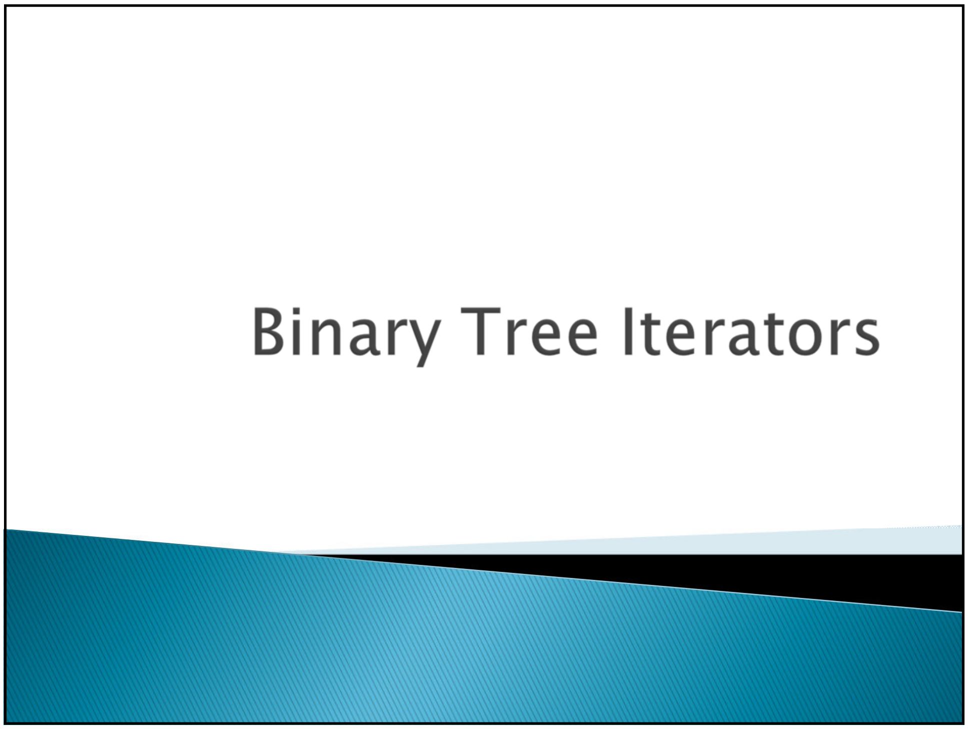
Figure 18.23
(a) Preorder, (b) postorder, and (c) inorder visitation routes



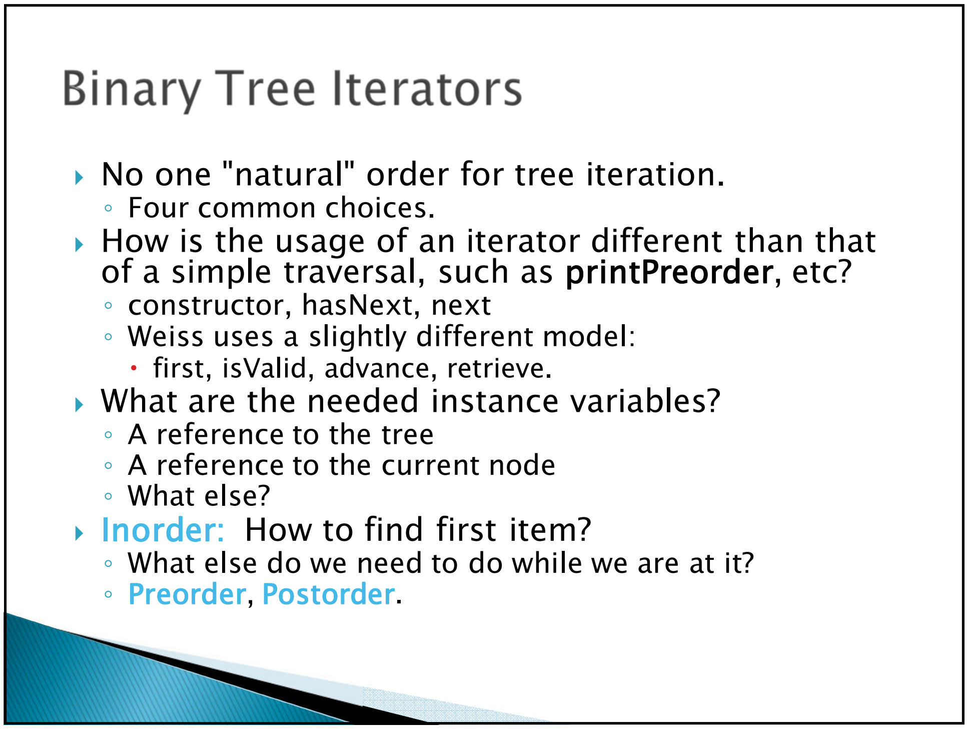
Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley



```
// Print tree rooted at current node using preorder
public void printPreOrder( )  {
    System.out.println( element );        // Node
    if( left != null )
        left.printPreOrder( );            // Left
    if( right != null )
        right.printPreOrder( );           // Right
}


// Print tree rooted at current node using postorde
public void printPostOrder( ) {
    if( left != null )
        left.printPostOrder( );           // Left
    if( right != null )
        right.printPostOrder( );          // Right
    System.out.println( element );        // Node
}

// Print tree rooted at current node using inorder t
public void printInOrder( )| {
    if( left != null )
        left.printInOrder( );             // Left
    System.out.println( element );        // Node
    if( right != null )
        right.printInOrder( );            // Right
```



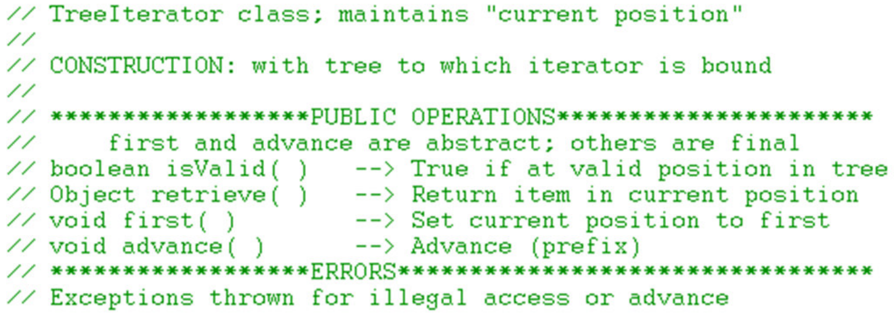
Binary Tree Iterators



Binary Tree Iterators

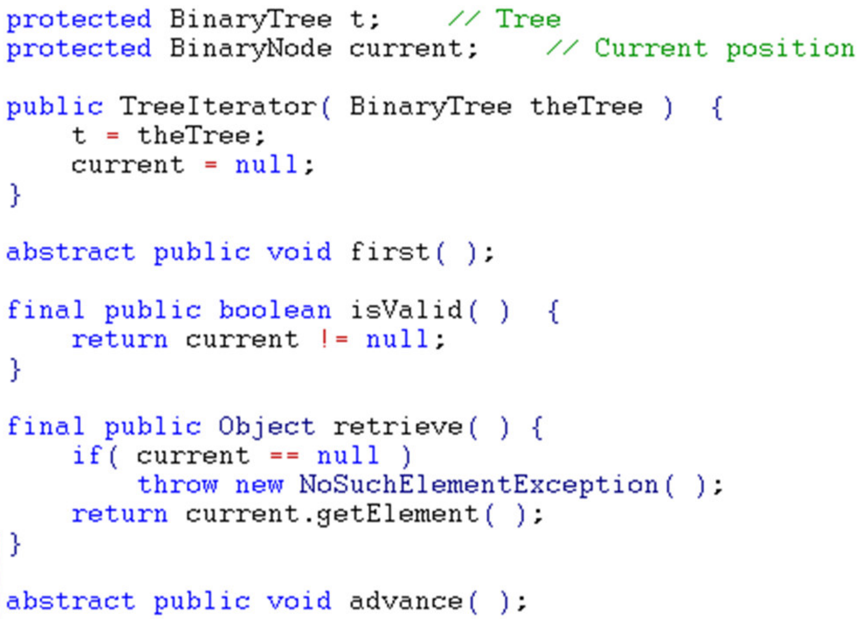
- No one "natural" order for tree iteration.
  - Four common choices.
- How is the usage of an iterator different than that of a simple traversal, such as **printPreorder**, etc?
  - constructor, hasNext, next
  - Weiss uses a slightly different model:
    - first, isValid, advance, retrieve.
- What are the needed instance variables?
  - A reference to the tree
  - A reference to the current node
  - What else?
- Inorder: How to find first item?
  - What else do we need to do while we are at it?
  - Preorder, Postorder.

TreeIterator abstract class

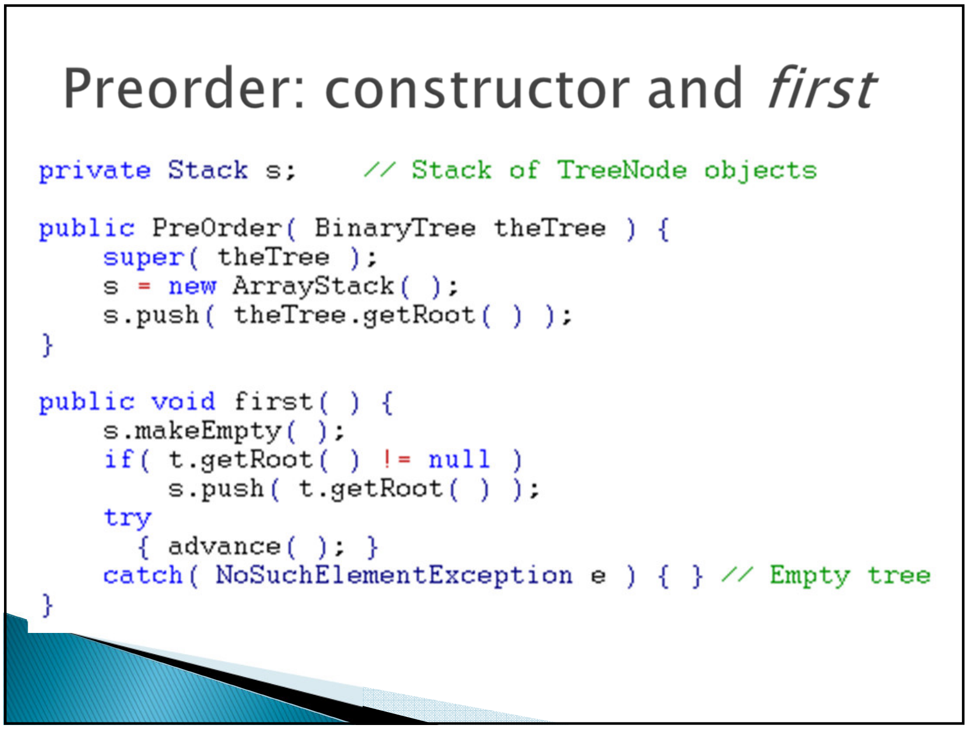


```
// TreeIterator class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS**********************
//     first and advance are abstract; others are final
// boolean isValid( )   --> True if at valid position in tree
// Object 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

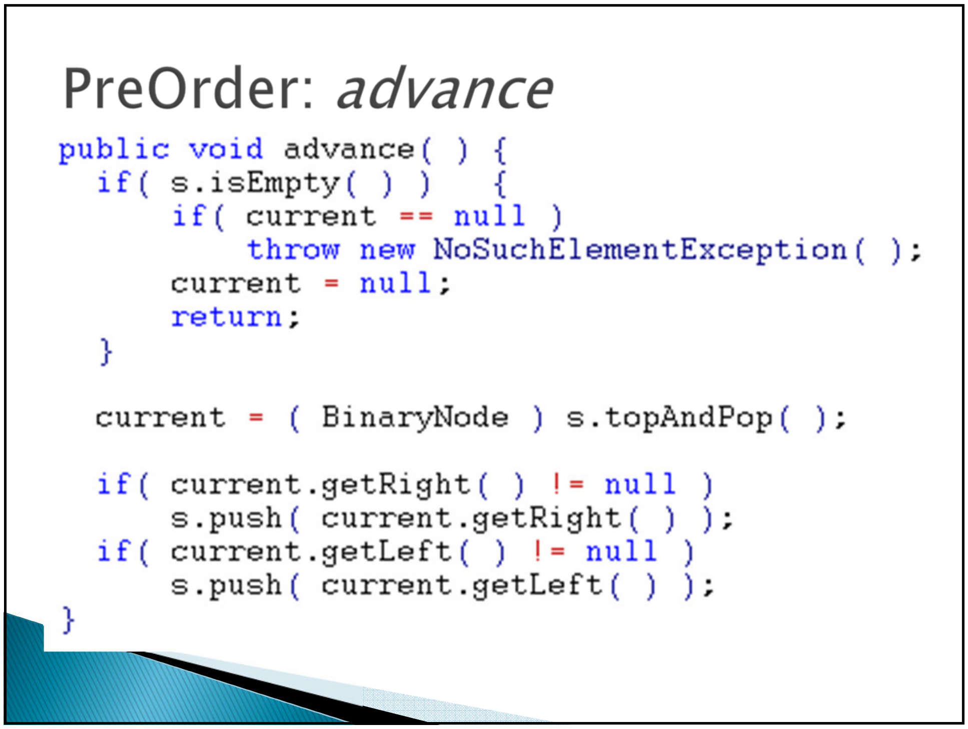


```
protected BinaryTree t;    // Tree
protected BinaryNode current;    // Current position

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*

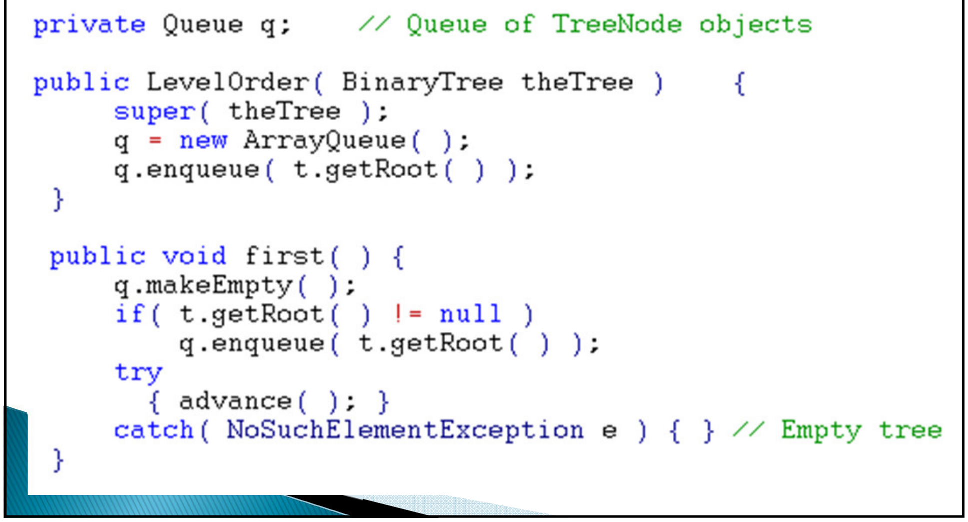
```
private Stack s;    // Stack of TreeNode objects

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*

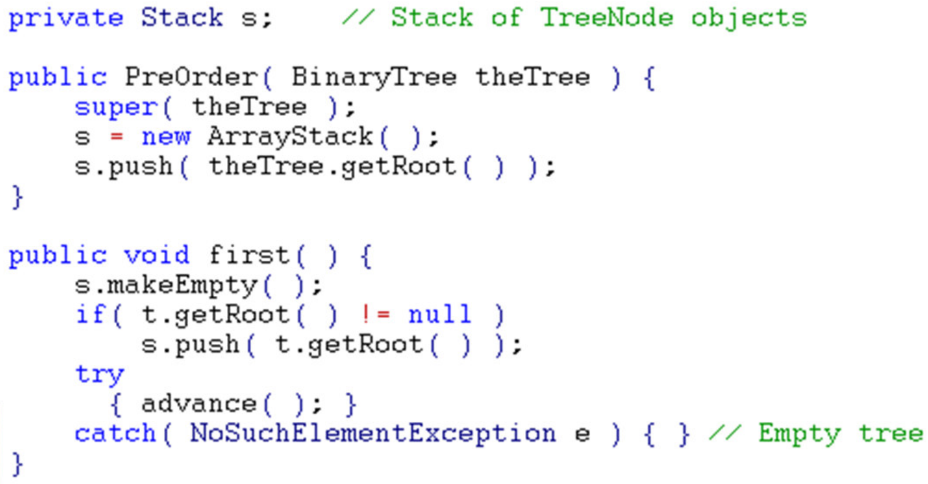
```
public void advance( ) {
  if( s.isEmpty( ) )    {
      if( current == null )
          throw new NoSuchElementException( );
      current = null;
      return;
  }

  current = ( BinaryNode ) s.topAndPop( );

  if( current.getRight( ) != null )
      s.push( current.getRight( ) );
  if( current.getLeft( ) != null )
      s.push( current.getLeft( ) );
}
```

LevelOrder: constructor and *first*



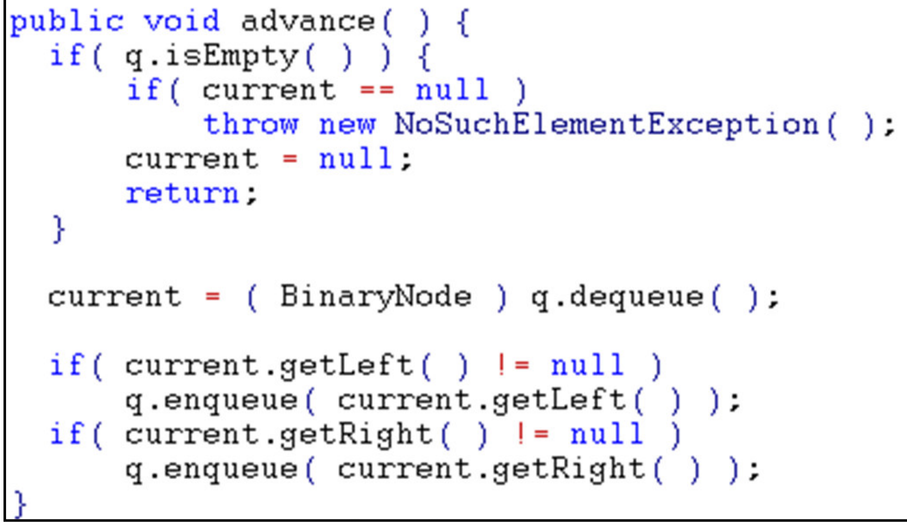
```
private Queue q;    // Queue of TreeNode objects

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*



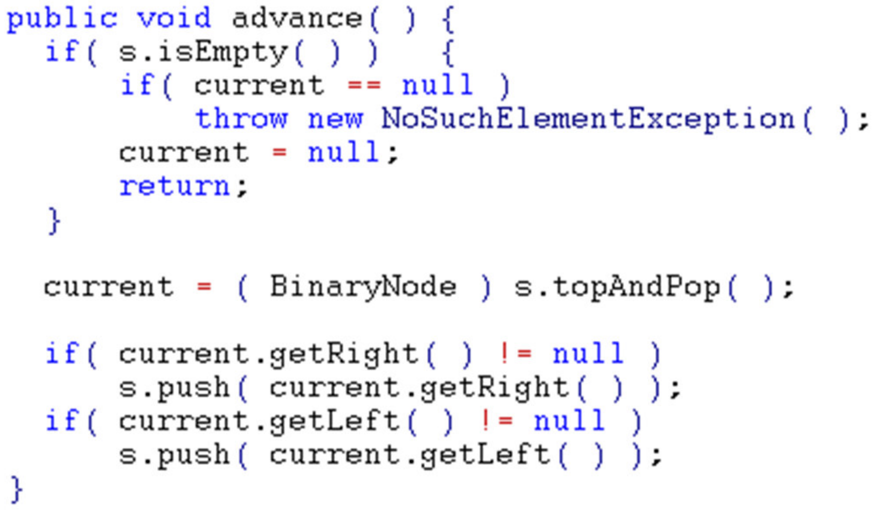
```
private Stack s;    // Stack of TreeNode objects

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*



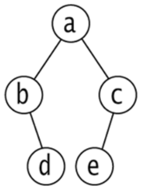
```
public void advance( ) {
  if( q.isEmpty( ) ) {
      if( current == null )
          throw new NoSuchElementException( );
      current = null;
      return;
  }

  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

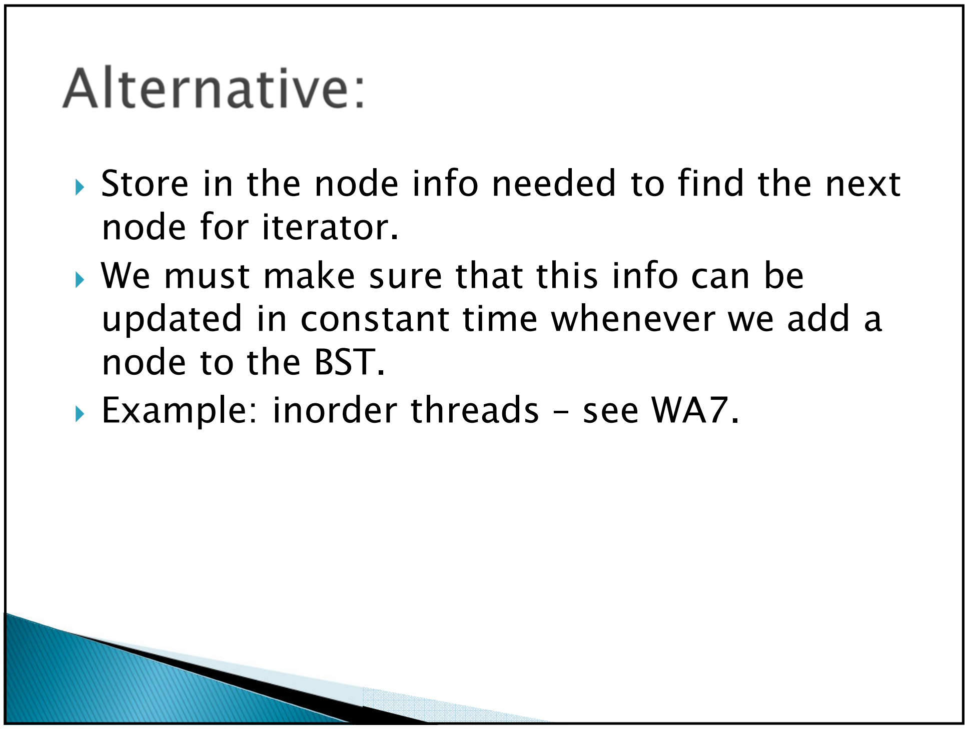


			d 0	d 1	d 2		
	b 0	b 1	b 2	b 2	b 2	b 2	
a 0	a 1	a 1	a 1	a 1	a 1	a 1	a 1
						d	b

	e 0	e 1	e 2				
c 0	c 1	c 1	c 1	c 1	c 2		
a 2	a 2	a 2	a 2	a 2	a 2	a 2	
				e		c	a

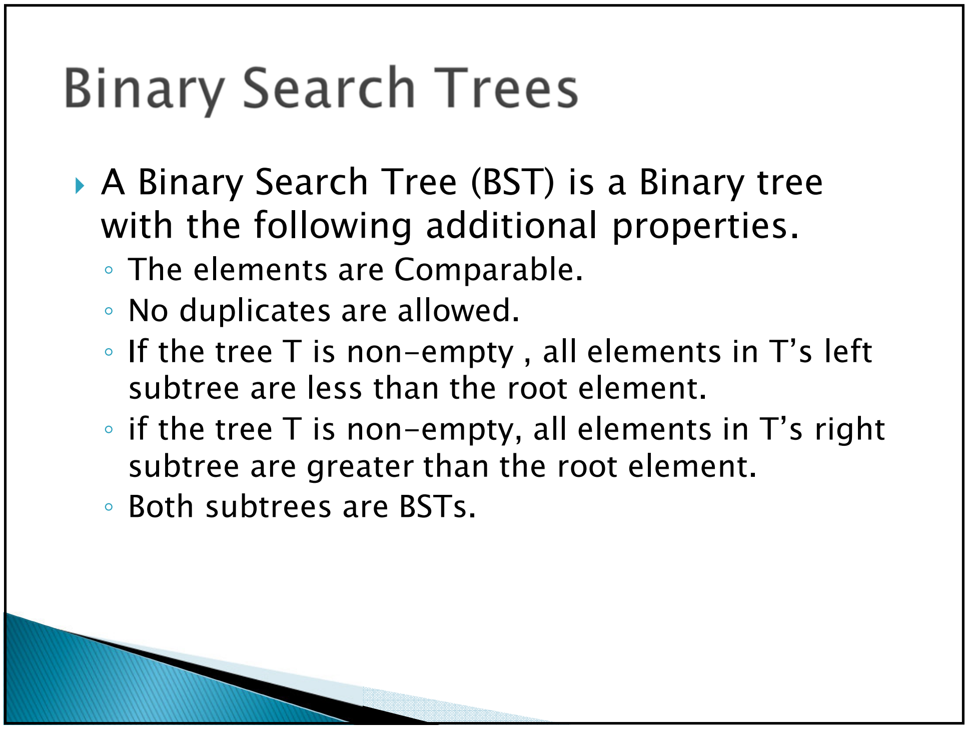
Wouldn’t it be nice?

- 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.



Alternative:

- Store in the node info needed to find the next node for iterator.
- We must make sure that this info can be updated in constant time whenever we add a node to the BST.
- Example: inorder threads – see WA7.



Binary Search Trees

- A Binary Search Tree (BST) is a Binary tree with the following additional properties.
  - The elements are Comparable.
  - No duplicates are allowed.
  - If the tree T is non-empty , all elements in T's left subtree are less than the root element.
  - if the tree T is non-empty, all elements in T's right subtree are greater than the root element.
  - Both subtrees are BSTs.