Data Structures
Lesson 6

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Outline

- PART I – Introduction to Recursion
- PART II - Trees

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Recursion

- A method that is partially defined in terms of itself is called recursive.
- Recursion, which is the use of recursive methods, is a powerful programming tool that in many cases can yield both short and efficient algorithms.
- We begin our discussion of recursion by examining the mathematical principle on which it is based: mathematical induction.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Recursion

- A recursive method is a method that either directly or indirectly makes a call to itself.
- This action may seem to be circular logic: How can a method F solve a problem by calling itself?
- The key is that the method F calls itself on a different, generally simpler, instance.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Recursion and infinite loops

- Recursion is a powerful problem-solving tool.
  - Many algorithms are most easily expressed in a recursive formulation.
  - Furthermore, the most efficient solutions to many problems are based on this natural recursive formulation.
- But you must be careful not to create circular logic that would result in infinite loops.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Proofs by mathematical induction

- Induction is commonly used to establish theorems that hold for positive integers.
- Theorem 7.1:

For any integer $N \geq 1$, the sum of the first N integers, given by $\sum_{i=1}^{N} i = 1 + 2 + \cdots + N$, equals $N(N+1)/2$.

- This particular theorem can be easily established by using other methods, but often a proof by induction is the simplest mechanism.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Proof by induction

- A proof by induction is carried out in two steps.
- First we show that the theorem is true for the smallest cases.
- We then show that if the theorem is true for the first few cases, it can be extended to include the next case.
- For instance, we show that a theorem that is true for all $1 \leq N \leq k$ must be true for $1 \leq N \leq k + 1$.
- Once we have shown how to extend the range of true cases, we have shown that it is true for all cases.
- The reason is that we can extend the range of true cases indefinitely.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

The basis

- Why does this constitute a proof?
- First, the theorem is true for $N = 1$, which is called the basis.
- We can view it as being the basis for our belief that the theorem is true in general.
- In a proof by induction, the basis is the easy case that can be shown by hand.
- Once we have established the basis, we use inductive hypothesis to assume that the theorem is true for some arbitrary k and that, under this assumption, if the theorem is true for k, then it is true for $k + 1$.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Sum of the first N integers

- Sometimes mathematical functions are defined recursively.
- For instance, let S(N) be the sum of the first N integers. Then S(1) = 1, and we can write
  - S(N) = S(N- 1) + N.
- Here we have defined the function S in terms of a smaller instance of itself.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Sum of the first N integers

figure 7.1

Recursive evaluation of the sum of the first *N* integers

```
1    // Evaluate the sum of the first n integers
2    public static long s( int n )
3    {
4        if( n == 1 )
5            return 1;
6        else
7            return s( n - 1 ) + n;
8    }
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Sum of the first N integers

- Note that, although ***s*** seems to be calling itself, in reality it is calling a clone of itself.
- That clone is simply another method with different parameters.
- At any instant only one clone is active; the rest are pending.
- It is the computer's job, not yours, to handle all the bookkeeping.
- If there were too much bookkeeping even for the computer, then it would be time to worry.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Rules of Recursion

- We thus have our first two (of four) fundamental rules of recursion.
  1. Base case: Always have at least one case that can be solved without using recursion.
  2. Make progress: Any recursive call must progress toward a base case.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Cost of bookkeeping

- A problem is that if the parameter n is large, but not so large that the answer does not fit in an int, the program can crash or hang.
- Our system, for instance, cannot handle N > 8,882.
- The reason is that the implementation of recursion requires some bookkeeping to keep track of the pending recursive calls, and for sufficiently long chains of recursion, the computer simply runs out of memory.
- This routine also is somewhat more time consuming than an equivalent loop because the bookkeeping also uses some time.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Third rule of induction

- When designing a recursive algorithm, we can always assume that the recursive calls work (if they progress toward the base case) because, when a proof is performed, this assumption is used as the inductive hypothesis.

  3. Always assume that the recursive call works.

- At first glance such an assumption seems strange.
  - However, recall that we always assume that method calls work, and thus the assumption that the recursive call works is really no different.
- Rule 3 tells us that when we design a recursive method, we do not have to attempt to trace the possibly long path of recursive calls.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

How is recursion implemented in programming languages?

- The implementation of recursion requires additional bookkeeping on the part of the computer.
- Said another way, the implementation of any method requires bookkeeping, and a recursive call is not particularly special (except that it can overload the computer's bookkeeping limitations by calling itself too many times).
- In Java, like other languages such as C++, implements methods by using an internal stack of activation records.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Activation records

- An activation record contains relevant information about the method, including, for instance, the values of the parameters and local variables.

- The actual contents of the activation record is system dependent.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

The stack and activation records

- The stack of activation records is used because methods return in reverse order of their invocation.
- Recall that stacks are great for reversing the order of things.
- In the most popular scenario, the top of the stack stores the activation record for the currently active method.
- When method G is called, an activation record for G is pushed onto the stack, which makes G the currently active method.
- When a method returns, the stack is popped and the activation record that is the new top of the stack contains the restored values.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Activation records

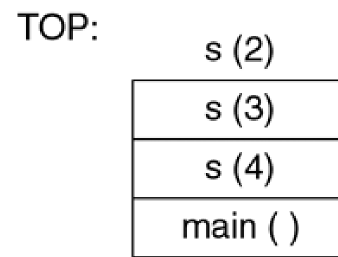


figure 7.5

A stack of activation records

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

A bad algorithm

figure 7.6

A recursive routine for Fibonacci numbers: A bad idea

```
1    // Compute the Nth Fibonacci number.
2    // Bad algorithm.
3    public static long fib( int n )
4    {
5        if( n <= 1 )
6            return n;
7        else
8            return fib( n - 1 ) + fib( n - 2 );
9    }
```

The underlying problem is that this recursive routine performs redundant calculations. To compute fib(n), we recursively compute fib(n-l). When the recursive call returns, we compute fib(n-2) by using another recursive call.

But we have already computed fib(n-2) in the process of computing fib(n-l), so the call to fib(n-2) is a wasted, redundant calculation.

In effect, we make two calls to fib(n-2) instead of only one.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

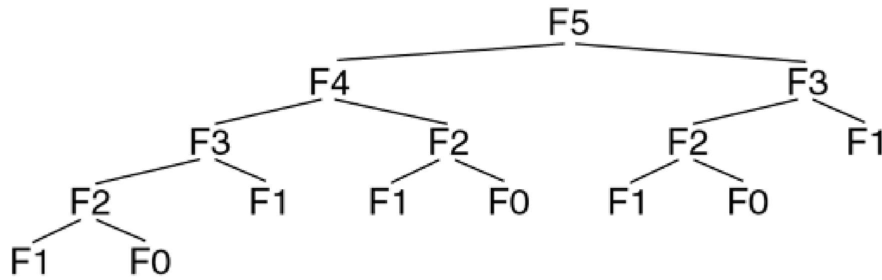


figure 7.7

A trace of the recursive calculation of the Fibonacci numbers

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Fourth rule of recursion

- Compound interest rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Factorial

```
1   // Evaluate n!
2   public static long factorial( int n )
3   {
4       if( n <= 1 )    // base case
5           return 1;
6       else
7           return n * factorial( n - 1 );
8   }
```

figure 7.10

Recursive implementation of the `factorial` method

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Greatest Common Divisor

figure 7.17

Computation of greatest common divisor

```
1   /**
2    * Return the greatest common divisor.
3    */
4   public static long gcd( long a, long b )
5   {
6       if( b == 0 )
7           return a;
8       else
9           return gcd( b, a % b );
10  }
```


Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

End of Part I

- Readings
  - Chapter 7

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

PART II - Trees

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Trees

- The tree is a fundamental structure in computer science.
- Almost all operating systems store files in trees or treelike structures.
- Trees are also used in compiler design, text processing, and searching algorithms.
- Artificial intelligence and many other subfields of computer science make use of trees heavily.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

General trees

- Trees can be defined in two ways: nonrecursively and recursively.
- The nonrecursive definition is the more direct technique, so we begin with it.
- The recursive formulation allows us to write simple algorithms to manipulate trees.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Tree definition

- Nonrecursively, a tree consists of a set of nodes and a set of directed edges that connect pairs of nodes.
- Throughout this text we consider only rooted trees.
- A rooted tree has the following properties.
  - One node is distinguished as the root.
  - Every node c, except the root, is connected by an edge from exactly one other node p. Node p is c's parent, and c is one of p's children.
  - A unique path traverses from the root to each node. The number of edges that must be followed is the path length.
- Parents and children are naturally defined. A directed edge connects the parent to the child.
- A node that has no children is called a leaf.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Depth and Height

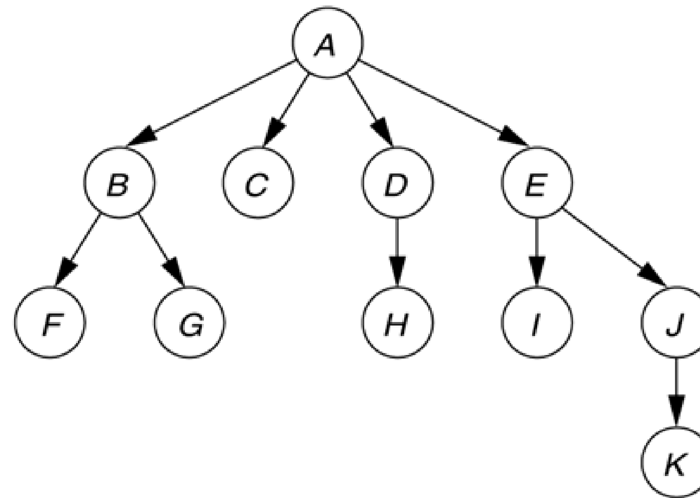
- The depth of a node in a tree is the length of the path from the root to the node.
- Thus the depth of the root is always 0, and the depth of any node is 1 more than the depth of its parent.
- The height of a node in a tree is the length of the path from the node to the deepest leaf.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Example of a tree

figure 18.1

A tree, with height and depth information



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Definitions

- Nodes with the same parent are called siblings; thus B, C, D, and E are all siblings.
- If there is a path from node u to node v, then u is an ancestor of v and v is a descendant of u.
- If $u \neq v$, then u is a proper ancestor of v and v is a proper descendant of u.
- The size of a node is the number of descendants the node has (including the node itself). Thus the size of B is 3, and the size of C is 1.
- The size of a tree is the size of the root.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Recursive Definition

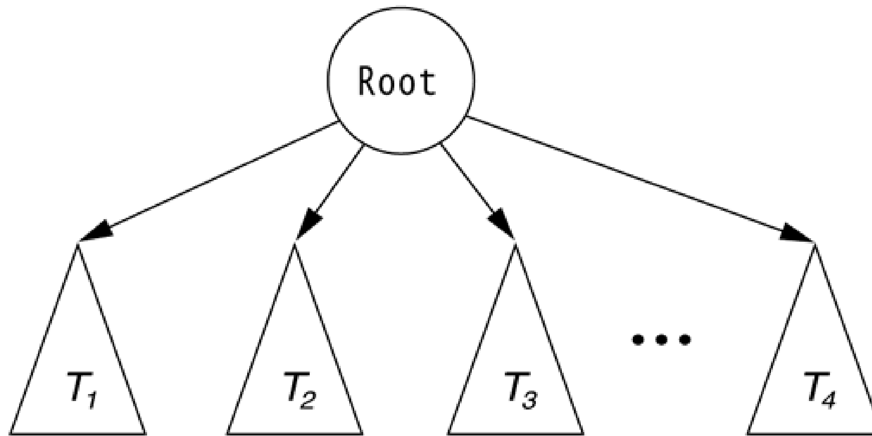


figure 18.2

A tree viewed recursively

An alternative definition of the tree is recursive: Either a tree is empty or it consists of a root and zero or more nonempty subtrees T_1, T_2,…, T_k, each of whose roots are connected by an edge from the root.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Implementation

- One way to implement a tree would be to have in each node a link to each child of the node in addition to its data.

- However, as the number of children per node can vary greatly and is not known in advance, making the children direct links in the data structure might not be feasible — there would be too much wasted space.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

First child/next sibling method

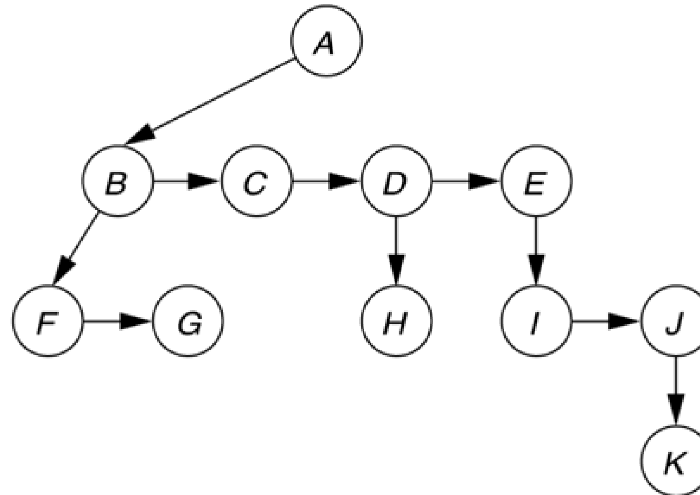
- The first child/next sibling method is simple:
- Keep the children of each node in a linked list of tree nodes, with each node keeping two links:
  - one to its leftmost child (if the node is not a leaf) and
  - one to its right sibling (if the node is not the rightmost sibling).

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

First child/next sibling method

figure 18.3

First child/next sibling representation of the tree in Figure 18.1

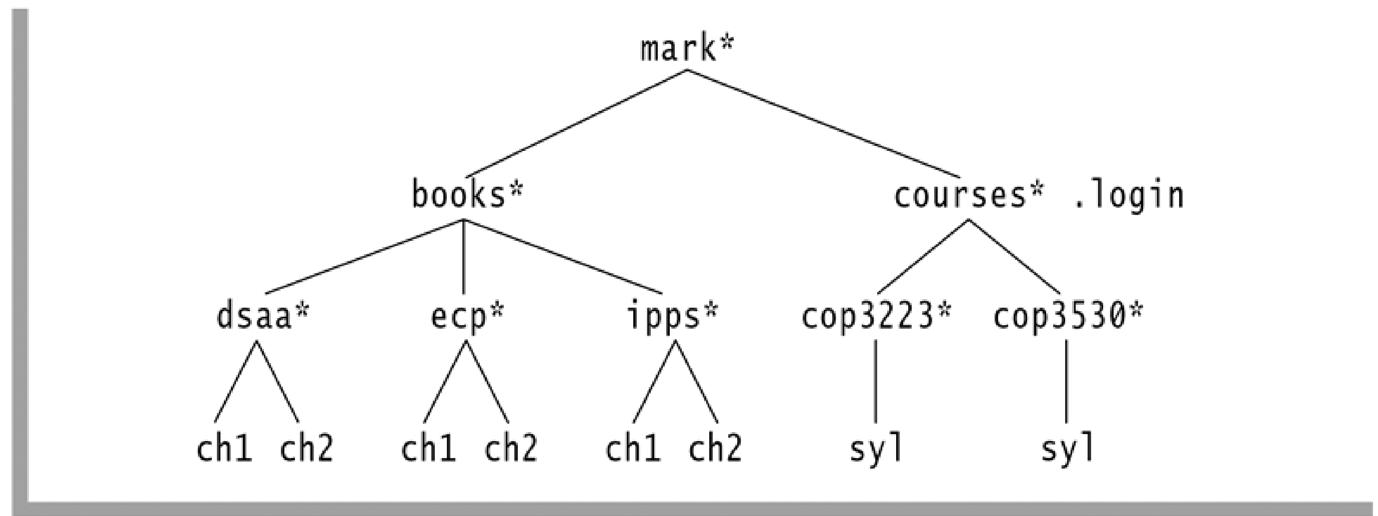


Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

File systems and trees

figure 18.4

A Unix directory



Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Listing a directory: pre-order visit

```
1    void listAll( int depth = 0 ) // depth is initially 0
2    {
3        printName( depth );        // Print the name of the object
4        if( isDirectory( ) )
5            for each file c in this directory (for each child)
6                c.listAll( depth + 1 );
7    }
```

figure 18.5

A routine for listing a directory and its subdirectories in a hierarchical file system

We first process the current node and then we go down the tree

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Preorder tree traversal

```
mark
      books
            dsaa
                  ch1
                  ch2
            ecp
                  ch1
                  ch2
            ipps
                  ch1
                  ch2
      courses
            cop3223
                  syl
            cop3530
                  syl
      .login
```

figure 18.6

The directory listing for the tree shown in Figure 18.4

In this algorithmic technique, known as a preorder tree traversal, work at a node is performed before (pre) its children are processed.

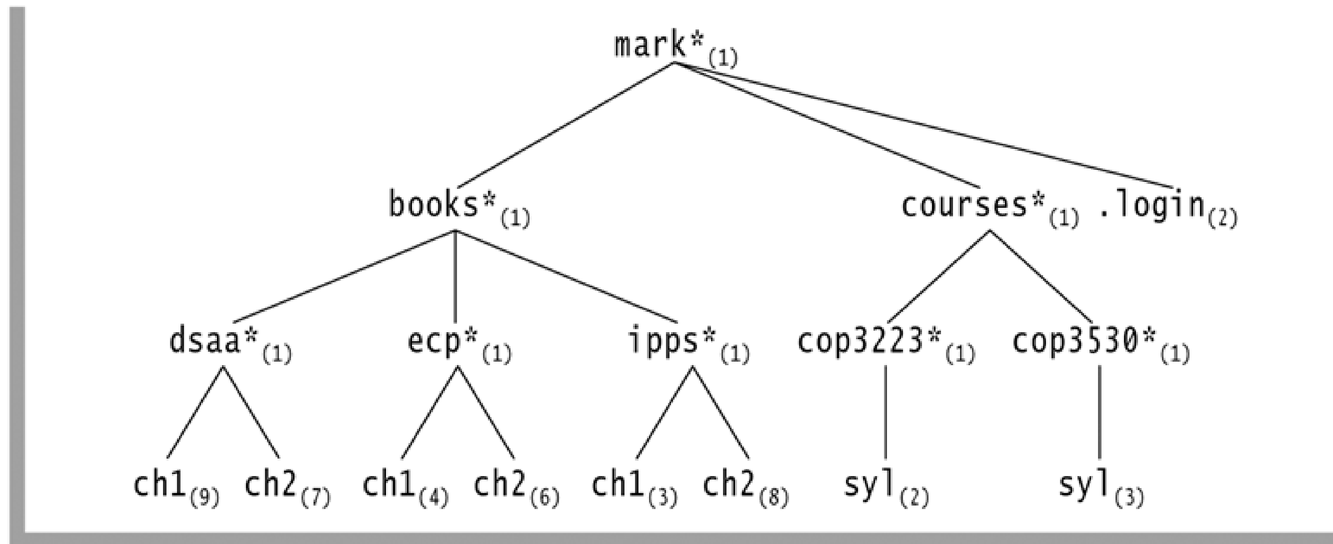
Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder tree traversal

- The work at a node is performed after (post) its children are evaluated.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder tree traversal



The numbers in parentheses represent the number of disk blocks taken up by each file.
The directories themselves are files, so they also use disk blocks (to store the names and information about their children).

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder tree traversal

- Suppose that we want to compute the total number of blocks used by all files in our example tree.
- The most natural way to do so is to find the total number of blocks contained in all the children (which may be directories that must be evaluated recursively): books (41), courses (8), and .login (2).
- The total number of blocks is then the total in all the children plus the blocks used at the root (1), or 52.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

The *size* routine: post-order visit

```
1     int size( )
2     {
3         int totalSize = sizeOfThisFile( );
4
5         if( isDirectory( ) )
6             for each file c in this directory (for each child)
7                 totalSize += c.size( );
8
9         return totalSize;
10    }
```

figure 18.8

A routine for calculating the total size of all files in a directory

We first go down the tree

and then

we return the size for each node

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder visit of a tree

```
            ch1                    9
            ch2                    7
        dsaa                      17
            ch1                    4
            ch2                    6
        ecp                       11
            ch1                    3
            ch2                    8
        ipps                      12
    books                         41
            syl                    2
        cop3223                    3
            syl                    3
        cop3530                    4
    courses                        8
    .login                         2
mark                              52
```

figure 18.9

A trace of the `size` method

We get a classic postorder signature because the total size of an entry is not computable until the information for its children has been computed.
The running time is linear.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
 1 import java.io.File;
 2
 3 public class FileSystem extends File
 4 {
 5         // Constructor
 6     public FileSystem( String name )
 7     {
 8         super( name );
 9     }
10
11         // Output file name with indentation
12     public void printName( int depth )
13     {
14         for( int i = 0; i < depth; i++ )
15             System.out.print( "\t" );
16         System.out.println( getName( ) );
17     }
18
19         // Public driver to list all files in directory
20     public void listAll( )
21     {
22         listAll( 0 );
23     }
24
25         // Recursive method to list all files in directory
26     private void listAll( int depth )
27     {
28         printName( depth );
29
30         if( isDirectory( ) )
31         {
32             String [ ] entries = list( );
33
34             for( String entry : entries)
35             {
36                 FileSystem child = new FileSystem( getPath( )
37                                 + separatorChar + entry );
38                 child.listAll( depth + 1 );
39             }
40         }
41     }
42
43         // Simple main to list all files in current directory
44     public static void main( String [ ] args )
45     {
46         FileSystem f = new FileSystem( "." );
47         f.ListAll( );
48     }
49 }
```

figure 18.10

Java implementation for a directory listing

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Binary Trees

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Binary trees

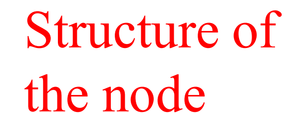
- A binary tree is a tree in which no node can have more than two children.
- Because there are only two children, we can name them left and right.
- Recursively, a binary tree is either empty or consists of a root, a left tree, and a right tree.
- The left and right trees may themselves be empty; thus a node with one child could have either a left or right child.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
 1 // BinaryNode class; stores a node in a tree.
 2 //
 3 // CONSTRUCTION: with no parameters, or an Object,
 4 //     left child, and right child.
 5 //
 6 // *******************PUBLIC OPERATIONS**********************
 7 // int size( )            --> Return size of subtree at node
 8 // int height( )          --> Return height of subtree at node
 9 // void printPostOrder( ) --> Print a postorder tree traversal
10 // void printInOrder( )   --> Print an inorder tree traversal
11 // void printPreOrder( )  --> Print a preorder tree traversal
12 // BinaryNode duplicate( )--> Return a duplicate tree
13
14 class BinaryNode<AnyType>
15 {
16     public BinaryNode( )
17       { this( null, null, null ); }
18     public BinaryNode( AnyType theElement,
19                        BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
20       { element = theElement; left = lt; right = rt; }
21
22     public AnyType getElement( )
23       { return element; }
24     public BinaryNode<AnyType> getLeft( )
25       { return left; }
26     public BinaryNode<AnyType> getRight( )
27       { return right; }
28     public void setElement( AnyType x )
29       { element = x; }
30     public void setLeft( BinaryNode<AnyType> t )
31       { left = t; }
32     public void setRight( BinaryNode<AnyType> t )
33       { right = t; }
34
35     public static <AnyType> int size( BinaryNode<AnyType> t )
36       { /* Figure 18.19 */ }
37     public static <AnyType> int height( BinaryNode<AnyType> t )
38       { /* Figure 18.21 */ }
39     public BinaryNode<AnyType> duplicate( )
40       { /* Figure 18.17 */ }
41
42     public void printPreOrder( )
43       { /* Figure 18.22 */ }
44     public void printPostOrder( )
45       { /* Figure 18.22 */ }
46     public void printInOrder( )
47       { /* Figure 18.22 */ }
48
49     private AnyType             element;
50     private BinaryNode<AnyType> left;
51     private BinaryNode<AnyType> right;
52 }
```

figure 18.12

The `BinaryNode` class skeleton



Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

figure 18.13

The `BinaryTree` class, except for `merge`

```
 1 // BinaryTree class; stores a binary tree.
 2 //
 3 // CONSTRUCTION: with (a) no parameters or (b) an object to
 4 //    be placed in the root of a one-element tree.
 5 //
 6 // ******************PUBLIC OPERATIONS*********************
 7 // Various tree traversals, size, height, isEmpty, makeEmpty.
 8 // Also, the following tricky method:
 9 // void merge( Object root, BinaryTree t1, BinaryTree t2 )
10 //                          --> Construct a new tree
11 // ******************ERRORS********************************
12 // Error message printed for illegal merges.
13
14 public class BinaryTree<AnyType>
15 {
16     public BinaryTree( )
17       { root = null; }
18     public BinaryTree( AnyType rootItem )
19       { root = new BinaryNode<AnyType>( rootItem, null, null ); }
20
21     public BinaryNode<AnyType> getRoot( )
22       { return root; }
23     public int size( )
24       { return BinaryNode.size( root ); }
25     public int height( )
26       { return BinaryNode.height( root ); }
27
28     public void printPreOrder( )
29       { if( root != null ) root.printPreOrder( ); }
30     public void printInOrder( )
31       { if( root != null ) root.printInOrder( ); }
32     public void printPostOrder( )
33       { if( root != null ) root.printPostOrder( ); }
34
35     public void makeEmpty( )
36       { root = null; }
37     public boolean isEmpty( )
38       { return root == null; }
39
40     public void merge( AnyType rootItem,
41                        BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
42       { /* Figure 18.16 */ }
43
44     private BinaryNode<AnyType> root;
45 }
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Recursion and Trees

- Because trees can be defined recursively, many tree routines, not surprisingly, are most easily implemented by using recursion.
- We will see recursive implementations for almost all the remaining BinaryNode and BinaryTree methods.
- The resulting routines are amazingly compact.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Duplicating a tree recursively

```
 1  /**
 2   * Return a reference to a node that is the root of a
 3   * duplicate of the binary tree rooted at the current node.
 4   */
 5  public BinaryNode<AnyType> duplicate( )
 6  {
 7      BinaryNode<AnyType> root =
 8                 new BinaryNode<AnyType>( element, null, null );
 9
10      if( left != null )            // If there's a left subtree
11          root.left = left.duplicate( );    // Duplicate; attach
12      if( right != null )          // If there's a right subtree
13          root.right = right.duplicate( );  // Duplicate; attach
14      return root;                          // Return resulting tree
15  }
```

figure 18.17

A routine for returning a copy of the tree rooted at the current node

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Calculating the size recursively

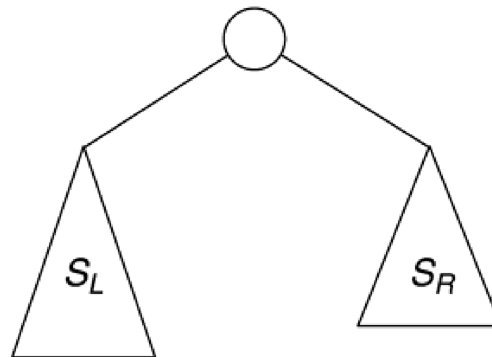


figure 18.18

Recursive view used to calculate the size of a tree: $S_T = S_L + S_R + 1$.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Calculating the size recursively

figure 18.19

A routine for computing the size of a node

```
1   /**
2    * Return the size of the binary tree rooted at t.
3    */
4   public static <AnyType> int size( BinaryNode<AnyType> t )
5   {
6       if( t == null )
7           return 0;
8       else
9           return 1 + size( t.left ) + size( t.right );
10  }
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Calculating the height recursively

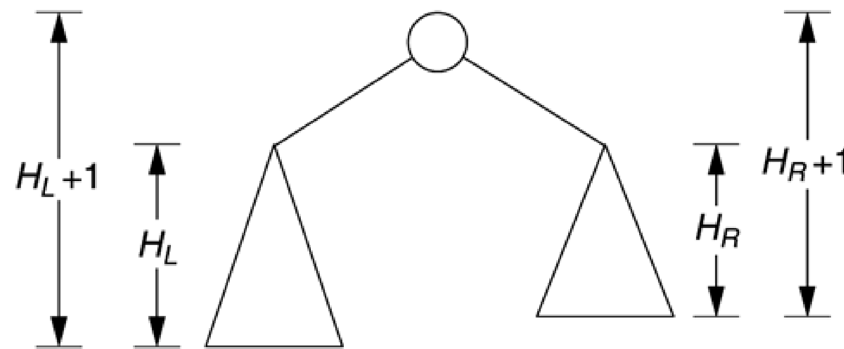


figure 18.20

Recursive view of the node height calculation:
$H_T = \text{Max}\ (H_L + 1, H_R + 1)$

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Calculating the height recursively

```
1   /**
2    * Return the height of the binary tree rooted at t.
3    */
4   public static <AnyType> int height( BinaryNode<AnyType> t )
5   {
6       if( t == null )
7           return -1;
8       else
9           return 1 + Math.max( height( t.left ), height( t.right ) );
10  }
```

figure 18.21

A routine for computing the height of a node

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Tree traversals

- When recursion is applied, we compute information about not only a node but also about all its descendants.
- We say then that we are traversing the tree.
- In a preorder traversal, the node is processed and then its children are processed recursively.
- In a postorder traversal, the node is processed after both children are processed recursively.
- In a inorder traversal, the left child is recursively processed, the current node is processed, and the right child is recursively processed.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Tree traversals

figure 18.22

Routines for printing nodes in preorder, postorder, and inorder

```
1   // Print tree rooted at current node using preorder traversal.
2   public void printPreOrder( )
3   {
4       System.out.println( element );      // Node
5       if( left != null )
6           left.printPreOrder( );          // Left
7       if( right != null )
8           right.printPreOrder( );         // Right
9   }
10
11  // Print tree rooted at current node using postorder traversal.
12  public void printPostOrder( )
13  {
14      if( left != null )                  // Left
15          left.printPostOrder( );
16      if( right != null )                 // Right
17          right.printPostOrder( );
18      System.out.println( element );      // Node
19  }
20
21  // Print tree rooted at current node using inorder traversal.
22  public void printInOrder( )
23  {
24      if( left != null )                  // Left
25          left.printInOrder( );
26      System.out.println( element );      // Node
27      if( right != null )
28          right.printInOrder( );          // Right
29  }
```

Print before

Print after

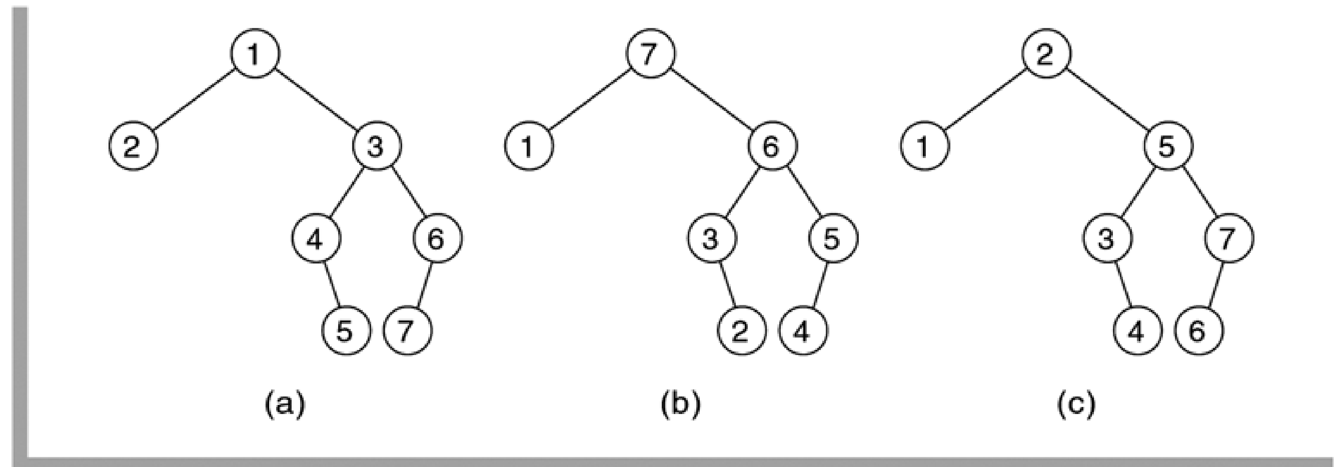
Print inorder

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Tree traversals

figure 18.23

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



Numbers represent the order in which each of the nodes is visited

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Running time for tree traversals

- The running time of each algorithm is linear. In every case, each node is output only once.
- Consequently, the total cost of an output statement over any traversal is O(N).
- As a result, each if statement is also executed at most once per node, for a total cost of O(N).
- The total number of method calls made (which involves the constant work of the internal run-time stack pushes and pops) is likewise once per node, or O(N).
- Thus the total running time is O(N).

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Trees: speed and recursion

- Must we use recursion to implement the traversals?
- The answer is clearly no, because, as discussed in Section 7.3, recursion is implemented by using a stack.
- Thus we could keep our own stack.
- We might expect that a somewhat faster program could result because we can place only the essentials on the stack rather than have the compiler place an entire activation record on the stack.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
1 import java.util.NoSuchElementException;
2
3 // TreeIterator class; maintains "current position"
4 //
5 // CONSTRUCTION: with tree to which iterator is bound
6 //
7 // ******************PUBLIC OPERATIONS**********************
8 //     first and advance are abstract; others are final
9 // boolean isValid( )    --> True if at valid position in tree
10 // AnyType retrieve( )   --> Return item in current position
11 // void first( )         --> Set current position to first
12 // void advance( )       --> Advance (prefix)
13 // ******************ERRORS*********************************
14 // Exceptions thrown for illegal access or advance
15
16 abstract class TreeIterator<AnyType>
17 {
18     /**
19      * Construct the iterator. The current position is set to null.
20      * @param theTree the tree to which the iterator is bound.
21      */
22     public TreeIterator( BinaryTree<AnyType> theTree )
23       { t = theTree; current = null; }
24
25     /**
26      * Test if current position references a valid tree item.
27      * @return true if the current position is not null; false otherwise.
28      */
29     final public boolean isValid( )
30       { return current != null; }
31
32     /**
33      * Return the item stored in the current position.
34      * @return the stored item.
35      * @exception NoSuchElementException if the current position is invalid.
36      */
37     final public AnyType retrieve( )
38     {
39         if( current == null )
40             throw new NoSuchElementException( );
41         return current.getElement( );
42     }
43
44     abstract public void first( );
45     abstract public void advance( );
46
47     protected BinaryTree<AnyType> t;        // The tree root
48     protected BinaryNode<AnyType> current;  // The current position
49 }
```

Abstract class

Structure of iterator

figure 18.24

The tree iterator abstract base class

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder traversal implementation

- The postorder traversal is implemented by using a stack to store the current state.
- The top of the stack will represent the node that we are visiting at some instant in the postorder traversal.
- However, we may be at one of three places in the algorithm:
  1. About to make a recursive call to the left subtree
  2. About to make a recursive call to the right subtree
  3. About to process the current node
- Consequently, each node is placed on the stack three times during the course of the traversal. If a node is popped from the stack a third time, we can mark it as the current node to be visited.

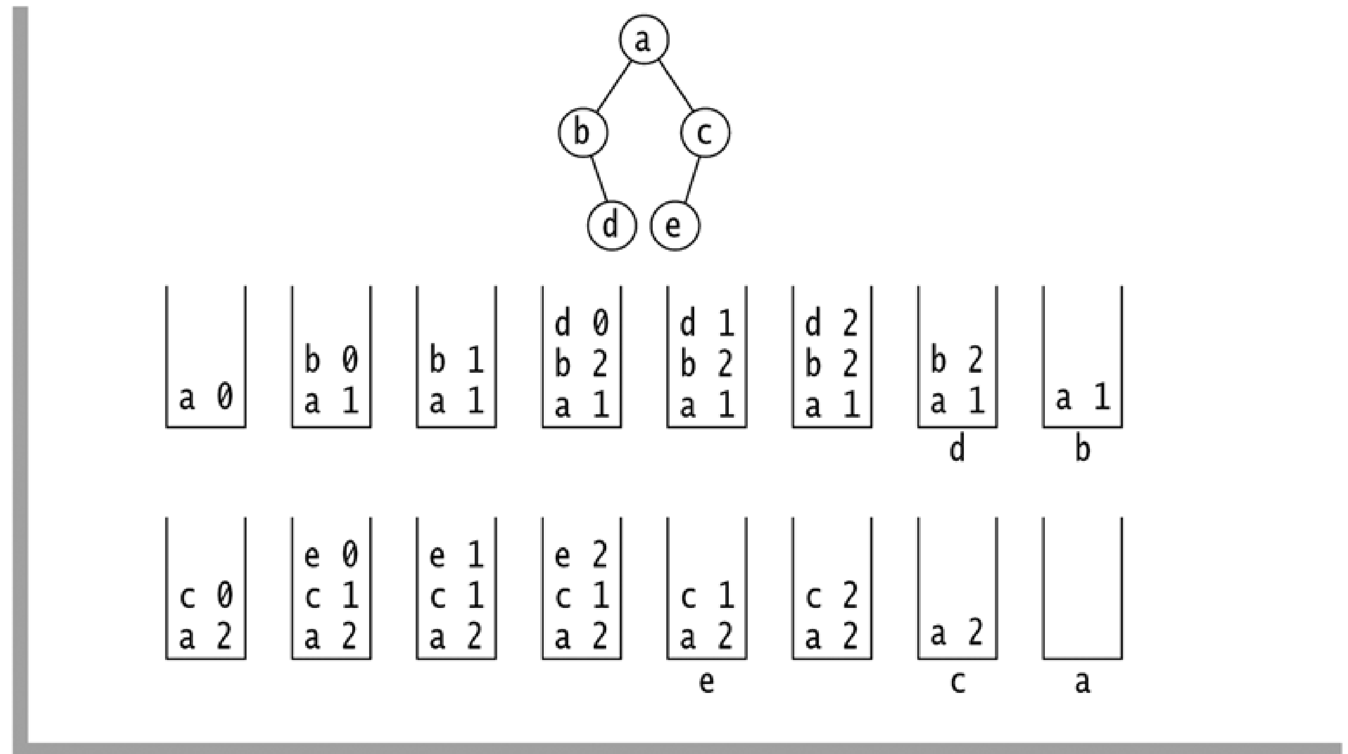
Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Postorder traversal implementation

- If the node is being popped for either the first time or the second time, it is not yet ready to be visited, so we push it back onto the stack and simulate a recursive call.
- If the node was popped for a first time, we need to push the left child (if it exists) onto the stack.
- Otherwise, the node was popped for a second time, and we push the right child (if it exists) onto the stack.
- In any event, we then pop the stack, applying the same test.
- Note that, when we pop the stack, we are simulating the recursive call to the appropriate child.
- If the child does not exist and thus was never pushed onto the stack, when we pop the stack we pop the original node again.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Tracing of the stack states



Therefore the order of visit is d, b, e, c, a

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
 1 import weiss.nonstandard.Stack;
 2 import weiss.nonstandard.ArrayStack;
 3
 4 // PostOrder class; maintains "current position"
 5 //     according to a postorder traversal
 6 //
 7 // CONSTRUCTION: with tree to which iterator is bound
 8 //
 9 // ******************PUBLIC OPERATIONS**********************
10 // boolean isValid( )   --> True if at valid position in tree
11 // AnyType retrieve( )  --> Return item in current position
12 // void first( )        --> Set current position to first
13 // void advance( )      --> Advance (prefix)
14 // ******************ERRORS*********************************
15 // Exceptions thrown for illegal access or advance
16
17 class PostOrder<AnyType> extends TreeIterator<AnyType>
18 {
19     protected static class StNode<AnyType>
20     {
21         StNode( BinaryNode<AnyType> n )
22           { node = n; timesPopped = 0; }
23         BinaryNode<AnyType> node;
24         int timesPopped;
25     }
26
27     /**
28      * Construct the iterator. The current position is set to null.
29      */
30     public PostOrder( BinaryTree<AnyType> theTree )
31     {
32         super( theTree );
33         s = new ArrayStack<StNode<AnyType>>( );
34         s.push( new StNode<AnyType>( t.getRoot( ) ) );
35     }
36
37     /**
38      * Set the current position to the first item.
39      */
40     public void first( )
41     {
42         s.makeEmpty( );
43         if( t.getRoot( ) != null )
44         {
45             s.push( new StNode<AnyType>( t.getRoot( ) ) );
46             advance( );
47         }
48     }
49
50     protected Stack<StNode<AnyType>> s; // The stack of StNode objects
51 }
```

figure 18.26

The PostOrder class (complete class except for advance)

The stack node

The stack

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```java
1   /**
2    * Advance the current position to the next node in the tree,
3    *     according to the postorder traversal scheme.
4    * @throws NoSuchElementException if the current position is null.
5    */
6   public void advance( )
7   {
8       if( s.isEmpty( ) )
9       {
10          if( current == null )
11              throw new NoSuchElementException( );
12          current = null;
13          return;
14      }
15
16      StNode<AnyType> cnode;
17
18      for( ; ; )
19      {
20          cnode = s.topAndPop( );
21
22          if( ++cnode.timesPopped == 3 )
23          {
24              current = cnode.node;
25              return;
26          }
27
28          s.push( cnode );
29          if( cnode.timesPopped == 1 )
30          {
31              if( cnode.node.getLeft( ) != null )
32                  s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
33          }
34          else  // cnode.timesPopped == 2
35          {
36              if( cnode.node.getRight( ) != null )
37                  s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
38          }
39      }
40  }
```

Remove from stack

Push left child

Push right child

figure 18.27

The `advance` routine for the `PostOrder` iterator class

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Inorder traversal implementation

- The inorder traversal is the same as the postorder traversal, except that a node is declared visited after it is popped a second time.
- Prior to returning, the iterator pushes the right child (if it exists) onto the stack so that the next call to *advance* can continue by traversing the right child.
- Because this action is so similar to a postorder traversal, we derive the InOrder class from the PostOrder class (even though an IS-A relationship does not exist).
- The only change is the minor alteration to *advance*.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
 1 // InOrder class; maintains "current position"
 2 //     according to an inorder traversal
 3 //
 4 // CONSTRUCTION: with tree to which iterator is bound
 5 //
 6 // ******************PUBLIC OPERATIONS**********************
 7 // Same as TreeIterator
 8 // ******************ERRORS*********************************
 9 // Exceptions thrown for illegal access or advance
10
11 class InOrder<AnyType> extends PostOrder<AnyType>
12 {
13     public InOrder( BinaryTree<AnyType> theTree )
14       { super( theTree ); }
15
16     /**
17      * Advance the current position to the next node in the tree,
18      *     according to the inorder traversal scheme.
19      * @throws NoSuchElementException if iteration has
20      *     been exhausted prior to the call.
21      */
22     public void advance( )
23     {
24         if( s.isEmpty( ) )
25         {
26             if( current == null )
27                 throw new NoSuchElementException( );
28             current = null;
29             return;
30         }
31
32         StNode<AnyType> cnode;
33         for( ; ; )
34         {
35             cnode = s.topAndPop( );
36
37             if( ++cnode.timesPopped == 2 )
38             {
39                 current = cnode.node;
40                 if( cnode.node.getRight( ) != null )
41                     s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
42                 return;
43             }
44                 // First time through
45             s.push( cnode );
46             if( cnode.node.getLeft( ) != null )
47                 s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
48         }
49     }
50 }
```

Push right child

Push left child

figure 18.28

The complete `InOrder` iterator class

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Preorder traversal implementation

- The preorder traversal is the same as the inorder traversal, except that a node is declared visited after it has been popped the first time.
- Prior to returning, the iterator pushes the right child onto the stack and then pushes the left child.
- Note the order: We want the left child to be processed before the right child, so we must push the right child first and the left child second.
- We could derive the PreOrder class from the InOrder or PostOrder class, but doing so would be wasteful because the stack no longer needs to maintain a count of the number of times an object has been popped.
- Consequently, the PreOrder class is derived directly from TreeIterator.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```java
1 // PreOrder class; maintains "current position"
2 //
3 // CONSTRUCTION: with tree to which iterator is bound
4 //
5 // ******************PUBLIC OPERATIONS**********************
6 // boolean isValid( )   --> True if at valid position in tree
7 // AnyType retrieve( )  --> Return item in current position
8 // void first( )        --> Set current position to first
9 // void advance( )      --> Advance (prefix)
10 // ******************ERRORS*********************************
11 // Exceptions thrown for illegal access or advance
12
13 class PreOrder<AnyType> extends TreeIterator<AnyType>
14 {
15     /**
16      * Construct the iterator. The current position is set to null.
17      */
18     public PreOrder( BinaryTree<AnyType> theTree )
19     {
20         super( theTree );
21         s = new ArrayStack<BinaryNode<AnyType>>( );
22         s.push( t.getRoot( ) );
23     }
24
25     /**
26      * Set the current position to the first item, according
27      * to the preorder traversal scheme.
28      */
29     public void first( )
30     {
31         s.makeEmpty( );
32         if( t.getRoot( ) != null )
33         {
34             s.push( t.getRoot( ) );
35             advance( );
36         }
37     }
38
39     public void advance( )
40       { /* Figure 18.30 */ }
41
42     private Stack<BinaryNode<AnyType>> s; // Stack of BinaryNode objects
43 }
```

figure 18.29

The PreOrder class skeleton and all members except advance

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

figure 18.30

The `PreOrder` iterator class `advance` routine

```
1      /**
2       * Advance the current position to the next node in the tree,
3       *     according to the preorder traversal scheme.
4       * @throws NoSuchElementException if iteration has
5       *     been exhausted prior to the call.
6       */
7      public void advance( )
8      {
9          if( s.isEmpty( ) )
10         {
11             if( current == null )
12                 throw new NoSuchElementException( );
13             current = null;
14             return;
15         }
16
17         current = s.topAndPop( );
18
19         if( current.getRight( ) != null )
20             s.push( current.getRight( ) );
21         if( current.getLeft( ) != null )
22             s.push( current.getLeft( ) );
23     }
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Level Order

- The level-order traversal processes nodes starting at the root and going from top to bottom, left to right.
- The name is derived from the fact that we output level 0 nodes (the root), level 1 nodes (root's children), level 2 nodes (grandchildren of the root), and so on.
- A level-order traversal is implemented by using a queue instead of a stack.
- The queue stores nodes that are yet to be visited.
- When a node is visited, its children are placed at the end of the queue where they are visited after the nodes that are already in the queue have been visited.
- This procedure guarantees that nodes are visited in level order.

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

```
1 // LevelOrder class; maintains "current position"
2 //     according to a level-order traversal
3 //
4 // CONSTRUCTION: with tree to which iterator is bound
5 //
6 // ******************PUBLIC OPERATIONS**********************
7 // boolean isValid( )   --> True if at valid position in tree
8 // AnyType retrieve( )  --> Return item in current position
9 // void first( )        --> Set current position to first
10 // void advance( )      --> Advance (prefix)
11 // ******************ERRORS*********************************
12 // Exceptions thrown for illegal access or advance
13
14 class LevelOrder<AnyType> extends TreeIterator<AnyType>
15 {
16     /**
17      * Construct the iterator.
18      */
19     public LevelOrder( BinaryTree<AnyType> theTree )
20     {
21         super( theTree );
22         q = new ArrayQueue<BinaryNode<AnyType>>( );
23         q.enqueue( t.getRoot( ) );
24     }
25
26     public void first( )
27       { /* Figure 18.32 */ }
28
29     public void advance( )
30       { /* Figure 18.32 */ }
31
32     private Queue<BinaryNode<AnyType>> q; // Queue of BinaryNode objects
33 }
```

figure 18.31

The `LevelOrder` iterator class skeleton

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

figure 18.32

The `first` and `advance` routines for the `LevelOrder` iterator class

```
1      /**
2       * Set the current position to the first item, according
3       * to the level-order traversal scheme.
4       */
5      public void first( )
6      {
7          q.makeEmpty( );
8          if( t.getRoot( ) != null )
9          {
10             q.enqueue( t.getRoot( ) );
11             advance( );
12         }
13     }
14
15     /**
16      * Advance the current position to the next node in the tree,
17      *     according to the level-order traversal scheme.
18      * @throws NoSuchElementException if iteration has
19      *     been exhausted prior to the call.
20      */
21     public void advance( )
22     {
23         if( q.isEmpty( ) )
24         {
25             if( current == null )
26                 throw new NoSuchElementException( );
27             current = null;
28             return;
29         }
30
31         current = q.dequeue( );
32
33         if( current.getLeft( ) != null )
34             q.enqueue( current.getLeft( ) );
35         if( current.getRight( ) != null )
36             q.enqueue( current.getRight( ) );
37     }
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Test the iterators

```
for( itr.first( ); itr.isValid( ); itr.advance( ) ){
        System.out.print( " " + itr.retrieve( ) );
    System.out.println( );
}
```

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

End of Lesson 6

- Readings
  - PART I – Chapter 7
  - PART II – Chapter 18

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved