Figure 18.4
A Unix directory

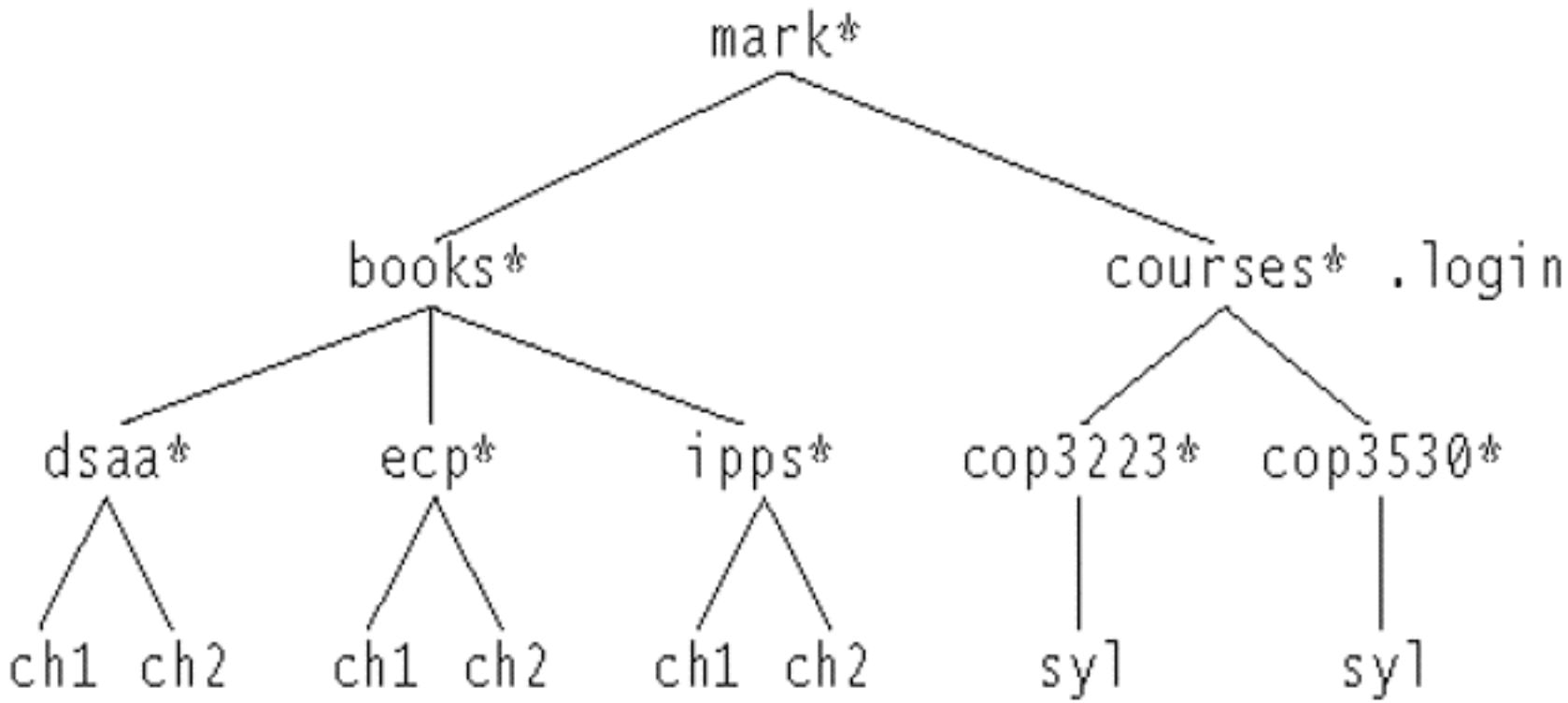


Figure 18.7

The Unix directory with file sizes

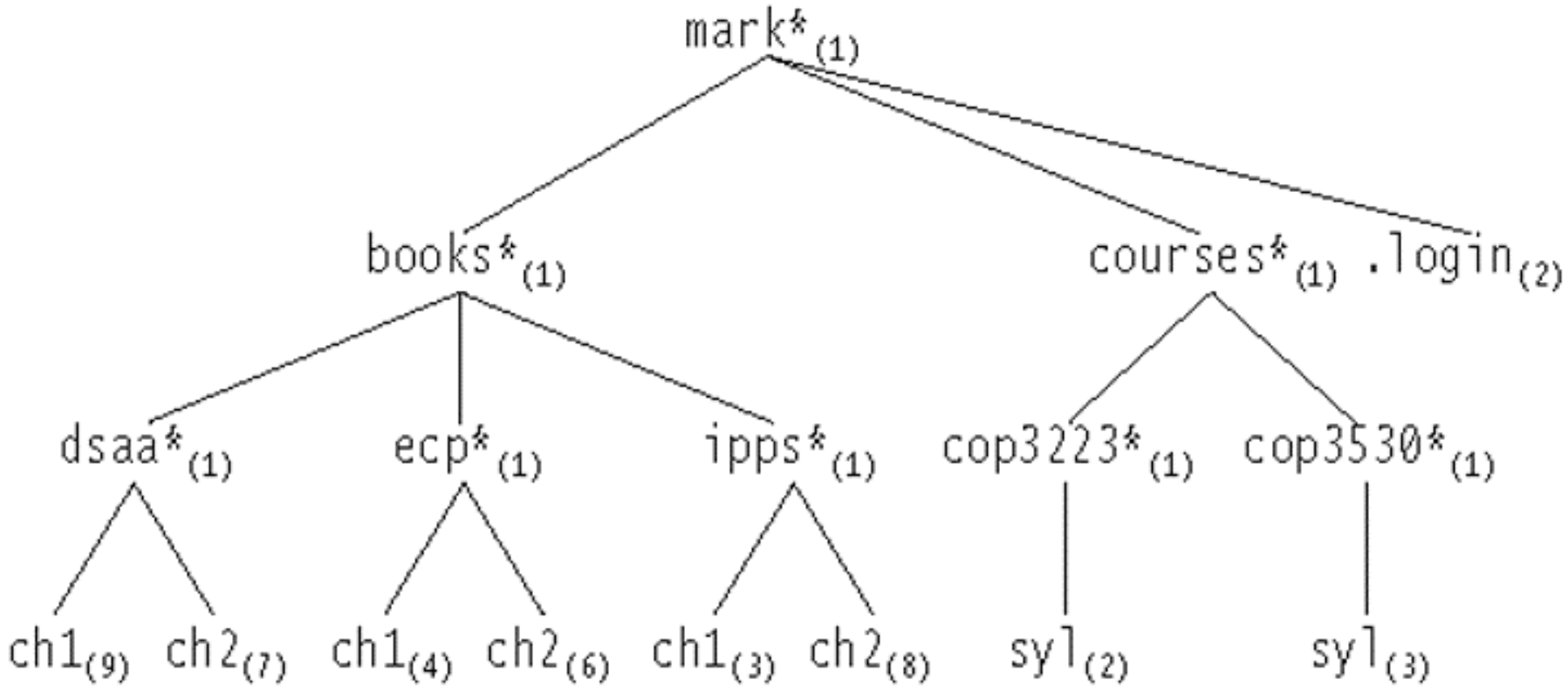
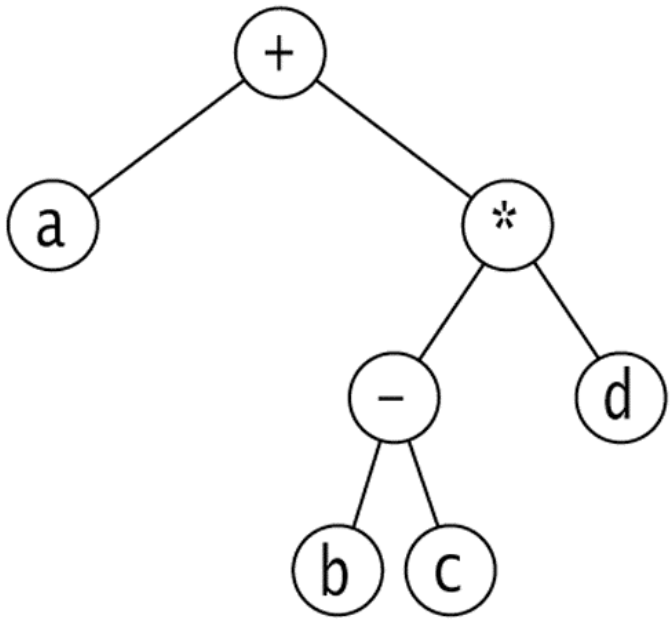
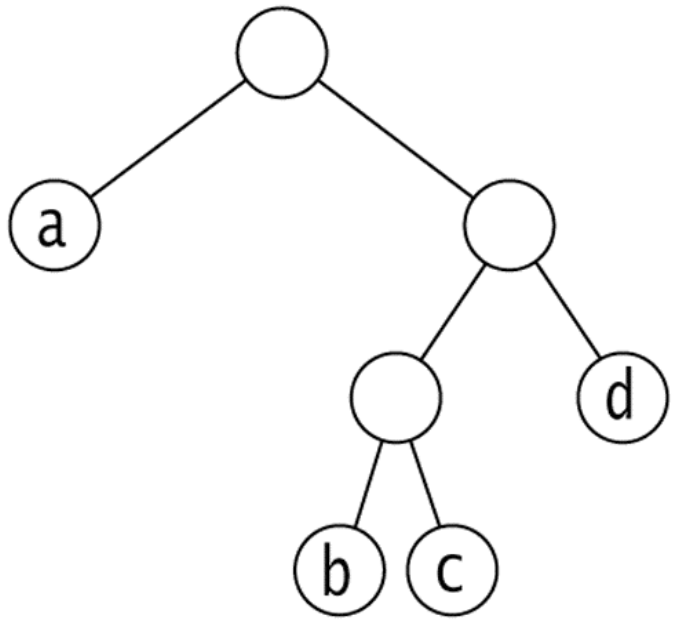


Figure 18.11

Uses of binary trees: (a) an expression tree and (b) a Huffman coding tree



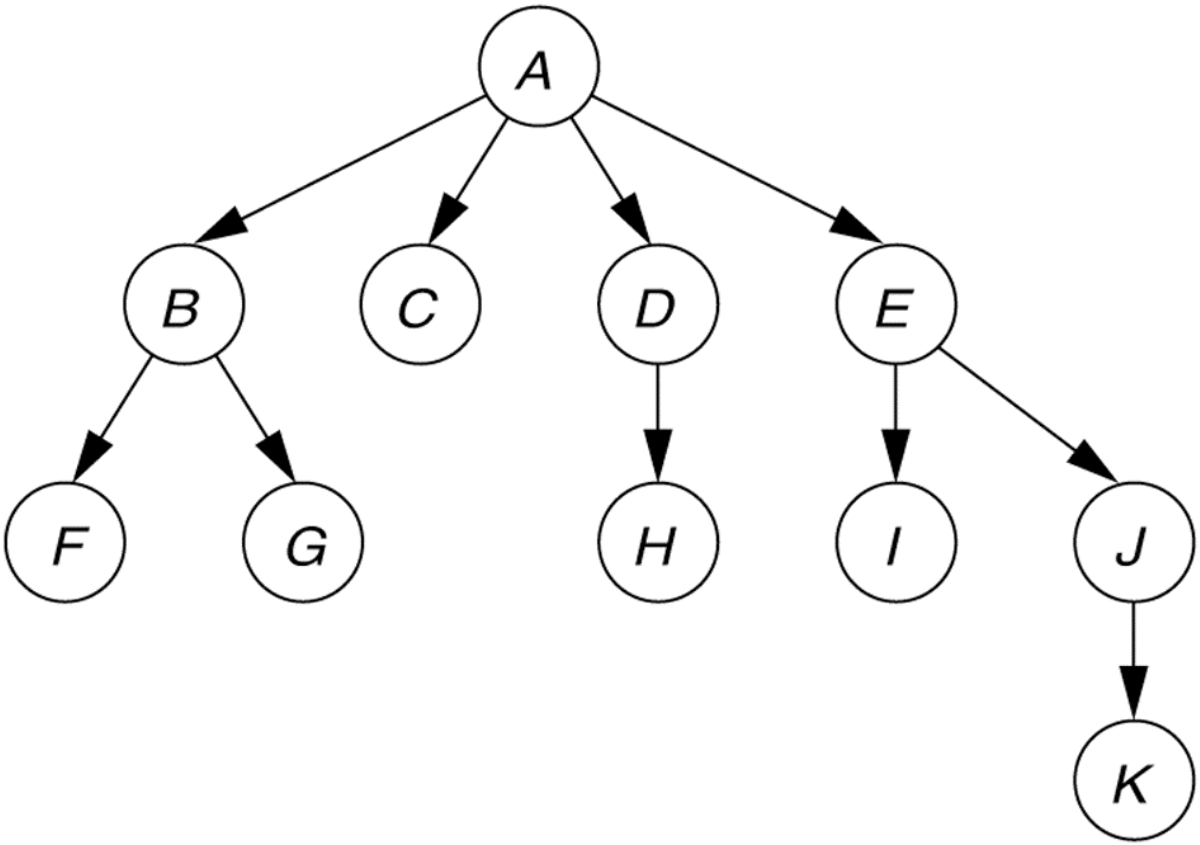
(a)



(b)

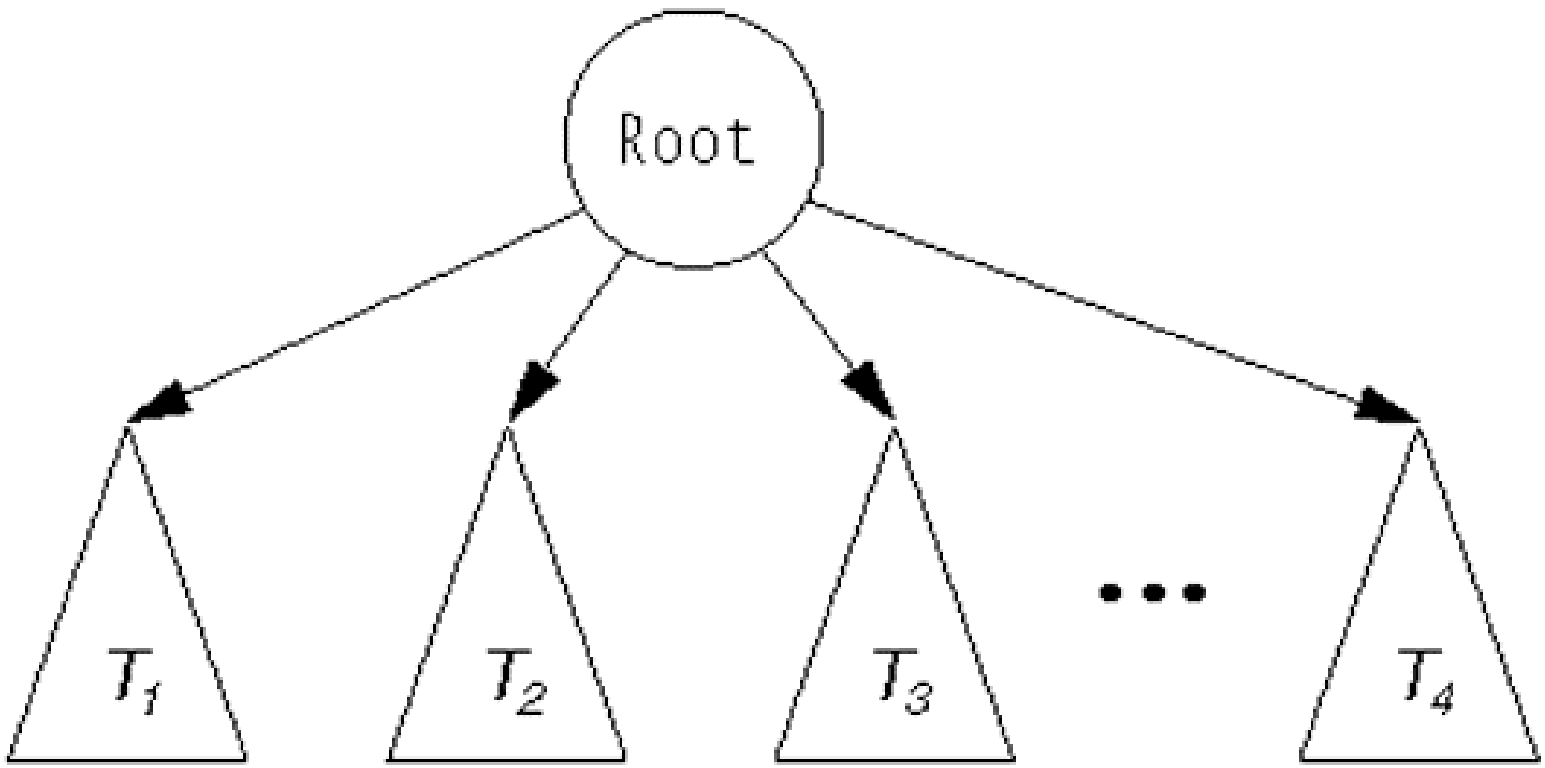
Figure 18.1

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

Figure 18.2
A tree viewed recursively



Binary Node

```
class BinaryNode
{
    public BinaryNode( )  {  this( null, null, null ); }
    public BinaryNode( Object theElement, BinaryNode lt, BinaryNode rt );
    public static int size( BinaryNode t );   // size of subtree rooted at t
    public static int height( BinaryNode t );
    public void printPreOrder( );
    public void printPostOrder( );
    public void printInOrder( );
    public BinaryNode duplicate( ); // make a duplicate tree and return root
    public Object getElement( );
    public BinaryNode getLeft( );
    public BinaryNode getRight( );
    public void setElement( Object x );
    public void setLeft( BinaryNode t );
    public void setRight( BinaryNode t );

    private Object     element;
    private BinaryNode left;
    private BinaryNode right;
}
```

Binary Trees

```
public class BinaryTree
{
    public BinaryTree( );
    public BinaryTree( Object rootItem );
    public void printPreOrder( );
    public void printInOrder( );
    public void printPostOrder( );
    public void makeEmpty( );
    public boolean isEmpty( );
   /** Forms a new tree from rootItem, t1 and t2. t1 not equal to t2. */
    public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 );
    public int size( );
    public int height( );
    public BinaryNode getRoot( );

    private BinaryNode root;
}
```

Binary Trees Cont'd

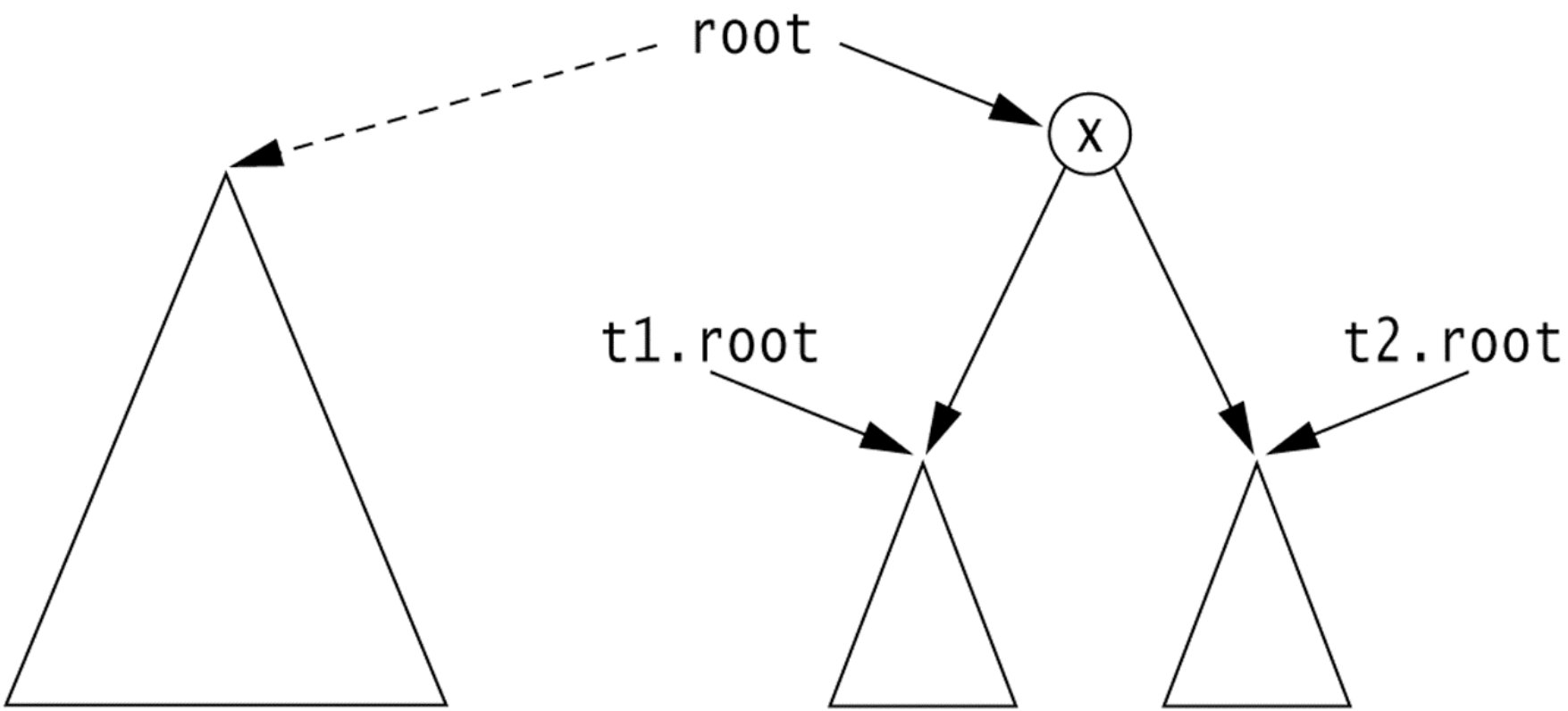
```
public class BinaryTree
{
   static public void main( String [ ] args )
   {
      BinaryTree t1 = new BinaryTree( "1" );   BinaryTree t3 = new BinaryTree( "3" );
      BinaryTree t5 = new BinaryTree( "5" );  BinaryTree t7 = new BinaryTree( "7" );
      BinaryTree t2 = new BinaryTree( );       BinaryTree t4 = new BinaryTree( );
      BinaryTree t6 = new BinaryTree( );
      t2.merge( "2", t1, t3 );   t6.merge( "6", t5, t7 );   t4.merge( "4", t2, t6 );

      System.out.println( "t4 should be perfect 1-7; t2 empty" );
      System.out.println( "----------------" );
      System.out.println( "t4" );
      t4.printInOrder( );
      System.out.println( "----------------" );
      System.out.println( "t2" );
      t2.printInOrder( );
      System.out.println( "----------------" );
      System.out.println( "t4 size: " + t4.size( ) );
      System.out.println( "t4 height: " + t4.height( ) );
   }
}
```

Binary Trees Cont'd

```
public void printPreOrder( )
  {
     System.out.println( element );      // Node
     if( left != null )  left.printPreOrder( );          // Left
     if( right != null )  right.printPreOrder( );         // Right
  }
  public void printPostOrder( )
  {
     if( left != null )   left.printPostOrder( );         // Left
     if( right != null )   right.printPostOrder( );       // Right
     System.out.println( element );      // Node
  }
  public void printInOrder( )
  {
     if( left != null )  left.printInOrder( );          // Left
     System.out.println( element );      // Node
     if( right != null )  right.printInOrder( );         // Right
  }
```

Figure 18.14

Result of a naive merge operation: Subtrees are shared.



Binary Trees Cont'd

```
public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 )
 {
    if( t1.root == t2.root && t1.root != null )  {
       System.err.println( "leftTree==rightTree; merge aborted" );
       return;
    }
   root = new BinaryNode( rootItem, t1.root, t2.root );
    if( this != t1 )    t1.root = null;
    if( this != t2 )    t2.root = null;
 }
```