Figure 18.4
A Unix directory

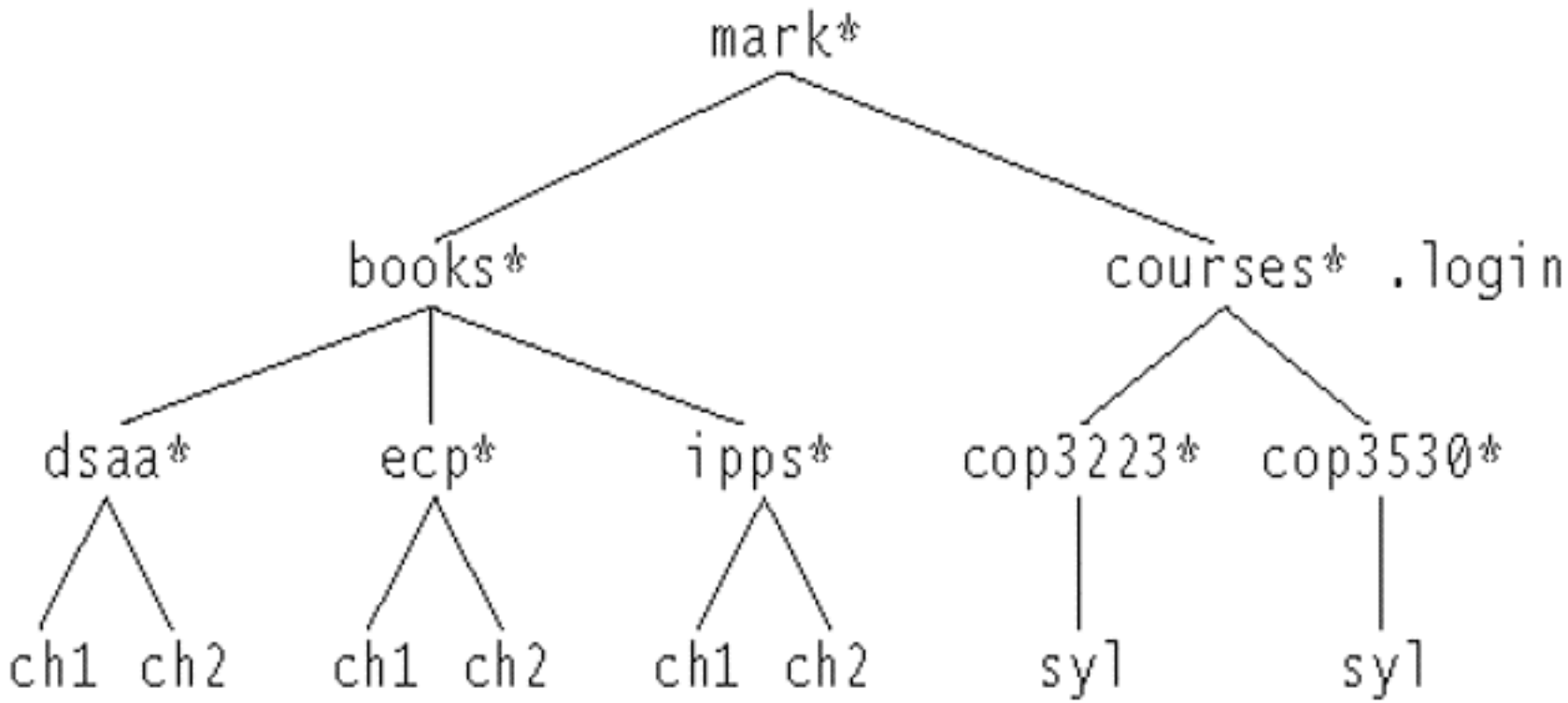


Figure 18.7

The Unix directory with file sizes

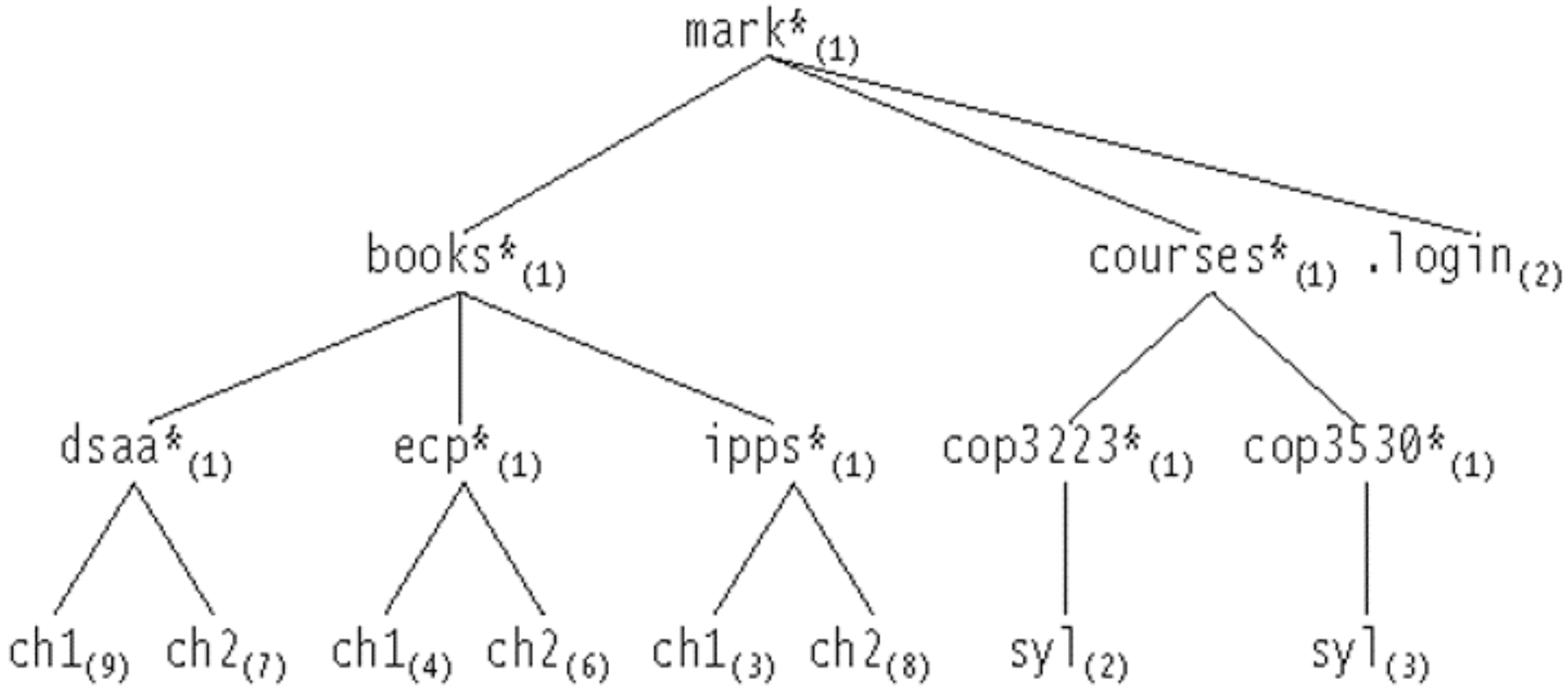
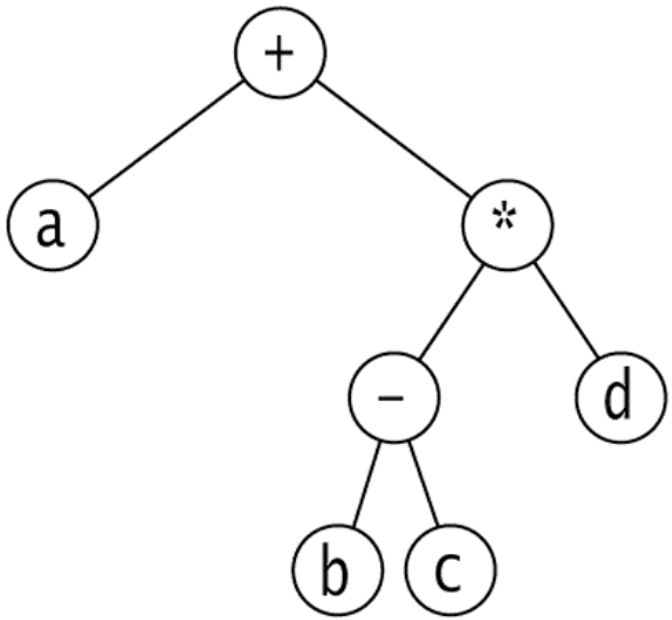
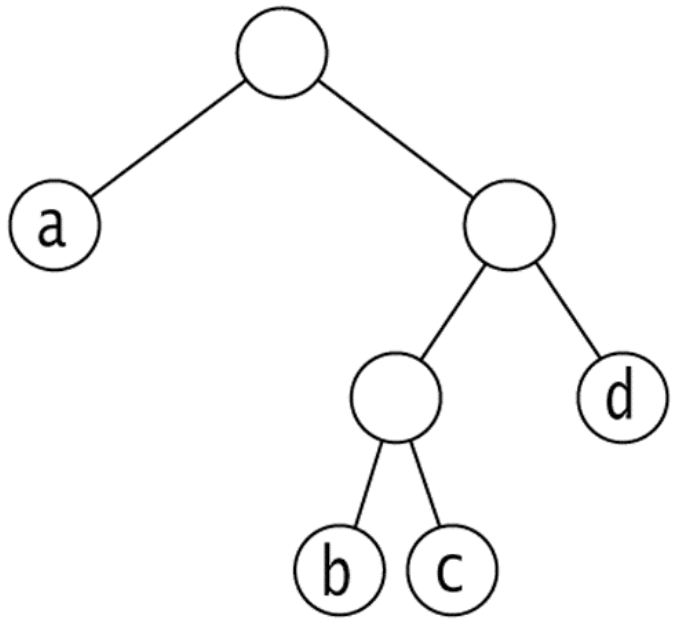


Figure 18.11

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



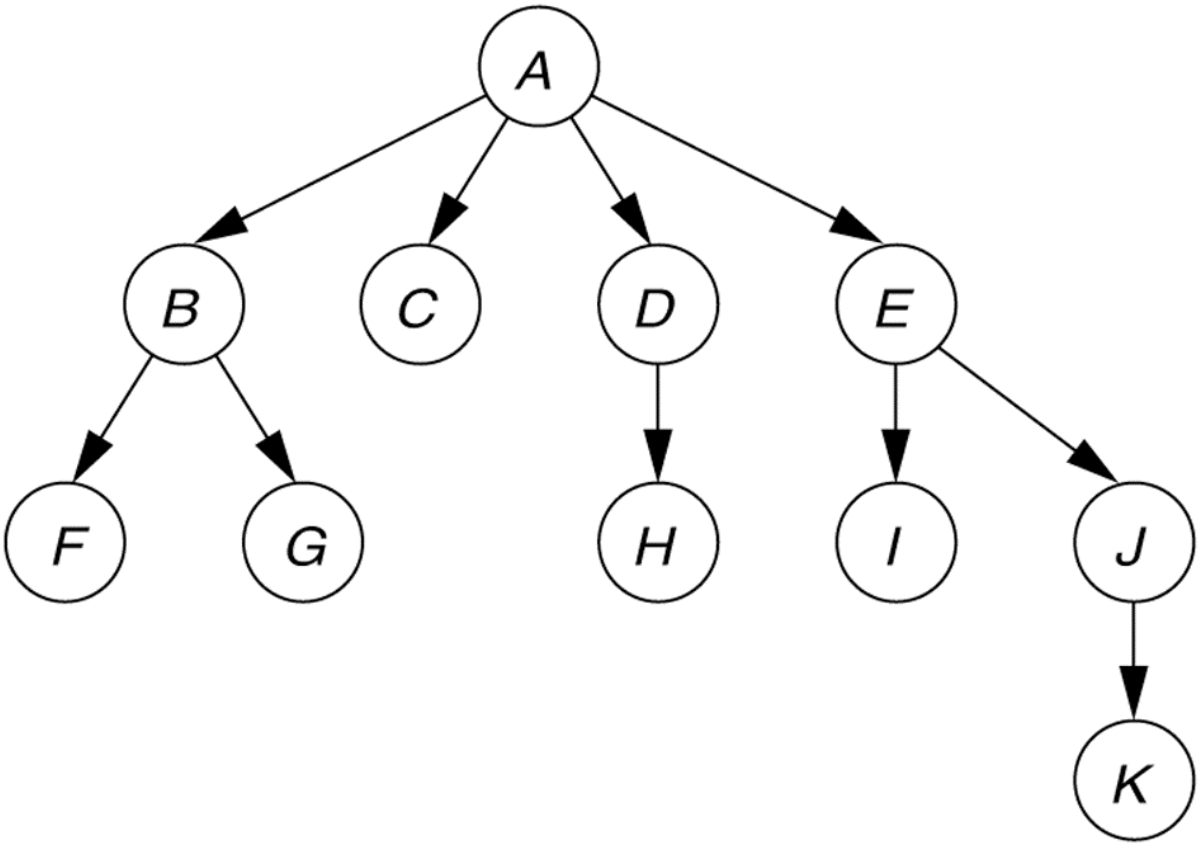
(a)



(b)

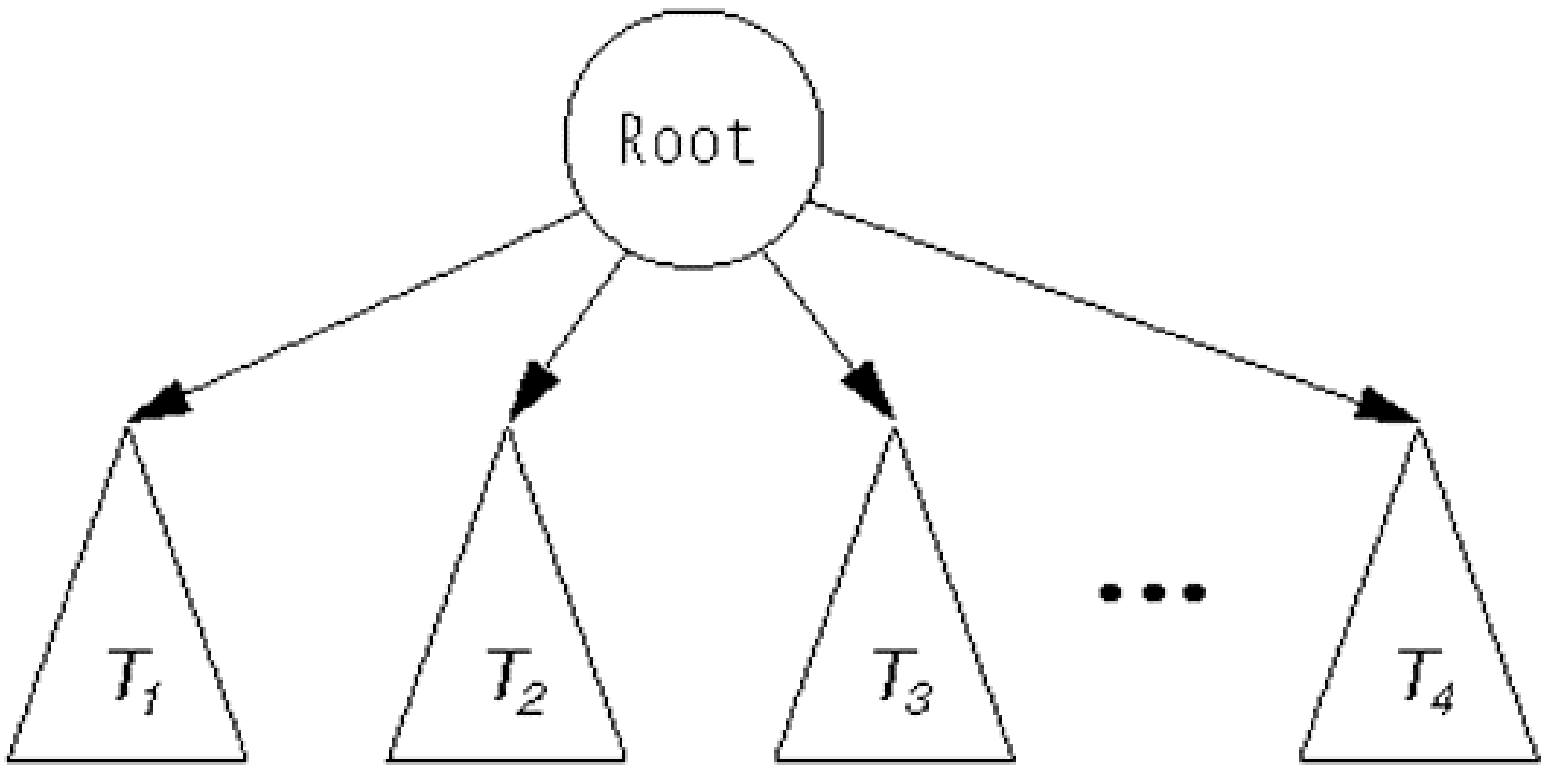
Figure 18.1

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 Trees

Constructors

Tree Properties

TreePrint

Get Info

Set Info

Data (private)

```
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

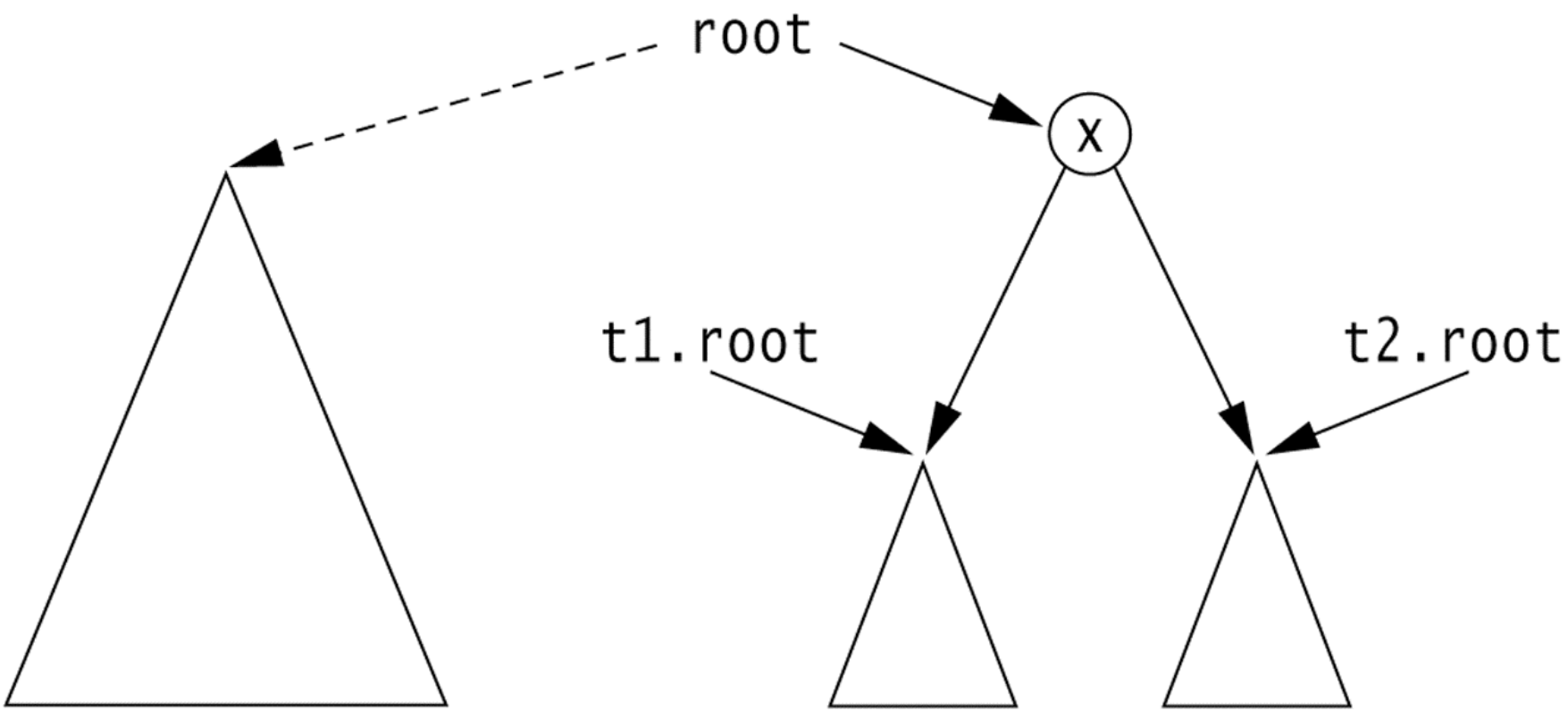
```
public class BinaryTree
{
   static public void main( String [ ] args )
   {
      BinaryTree t1 = new BinaryTree( "1" );   BinaryTree t3 = new BinaryTree( "3" );
      BinaryTree t5 = new BinaryTree( "5" );  BinaryTree t7 = new BinaryTree( "7" );
      BinaryTree t2 = new BinaryTree( );       BinaryTree 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

```
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

```
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;
 }

public BinaryNode duplicate( )
 {
    BinaryNode root = new BinaryNode( element, null, null );
    if( left != null )    root.left = left.duplicate( );
    if( right != null )  root.right = right.duplicate( );
    return root;                    // Return resulting tree
 }
```

Binary Search Trees

Values in the left subtree are smaller than the value stored at root.
Values in the right subtree are larger than the value stored at root.

Figure 19.1
Two binary trees: (a) a search tree;
(b) not a search tree

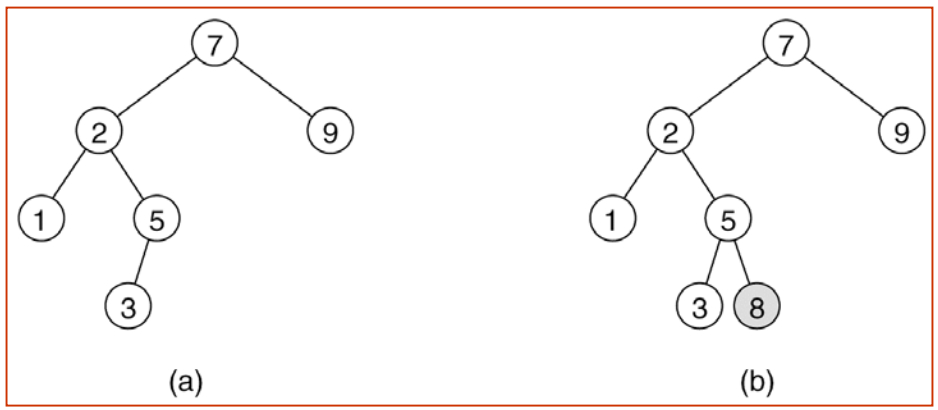


Figure 19.1

Two binary trees: (a) a search tree;
(b) not a search tree

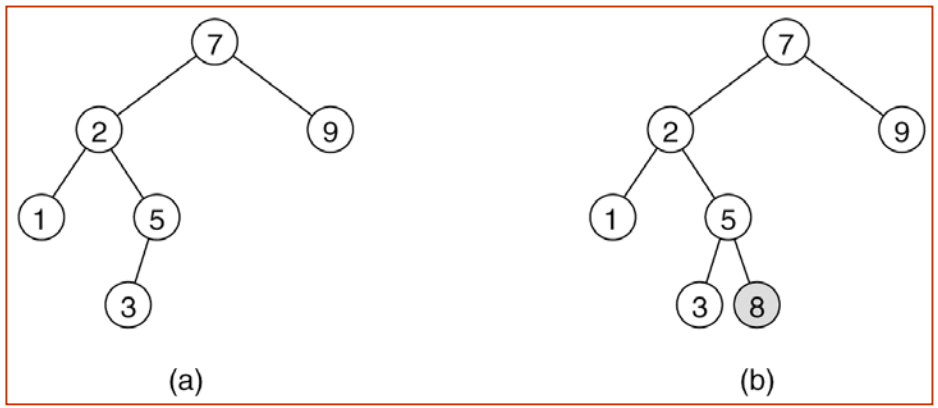


Figure 19.2

Binary search trees
(a) before and (b) after the insertion of 6

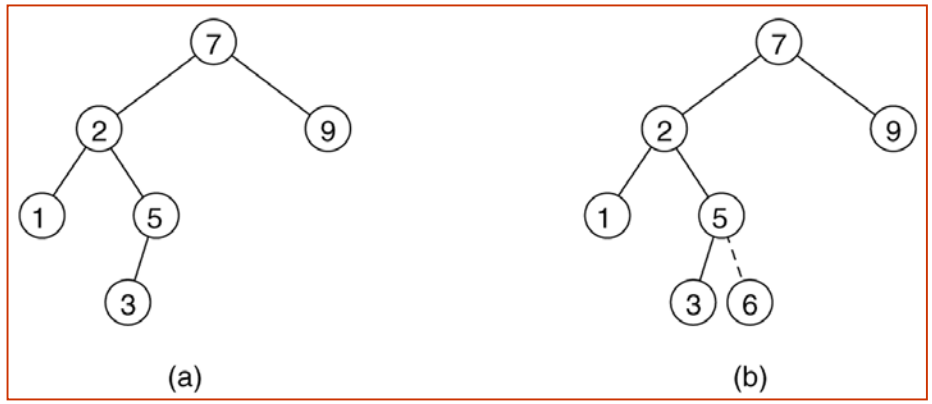


Figure 19.3

Deletion of node 5 with one child:
(a) before and (b) after

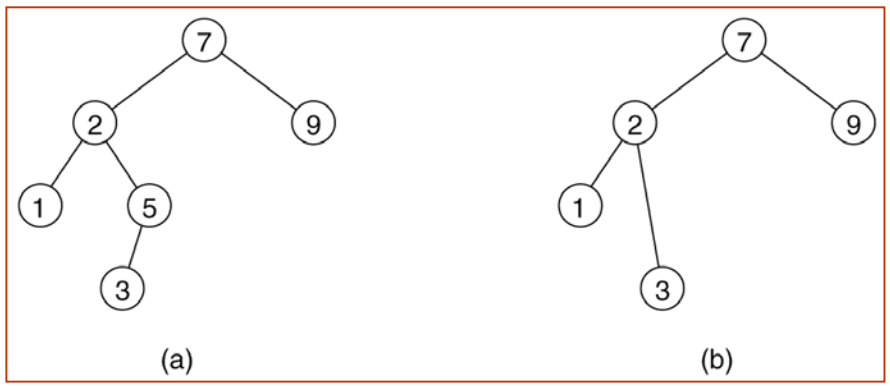
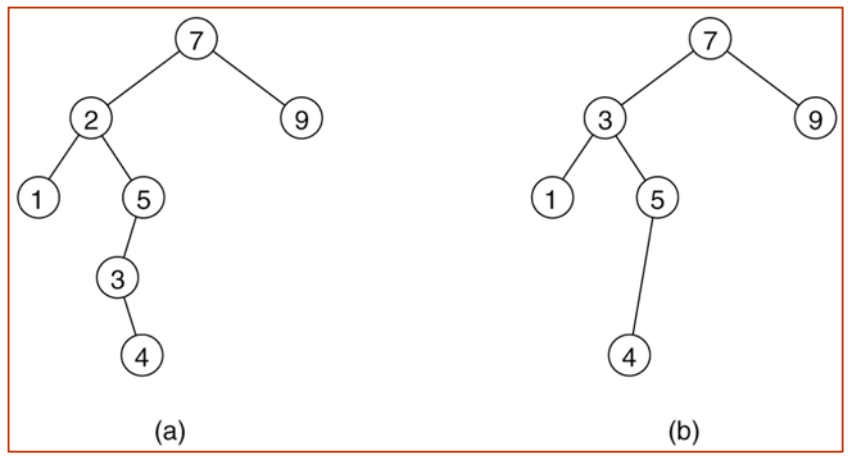


Figure 19.4

Deletion of node 2 with two children:
(a) before and (b) after



Binary Search Trees

```
// BinarySearchTree class
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// void removeMin( )      --> Remove minimum item
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( ) / findMax( ) --> Return smallest / largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
public class BinarySearchTree
{  private Comparable elementAt( BinaryNode t )  { return t == null ? null :
     t.element; }
   protected BinaryNode insert( Comparable x, BinaryNode t )
   protected BinaryNode remove( Comparable x, BinaryNode t )
   protected BinaryNode removeMin( BinaryNode t )
   protected BinaryNode findMin( BinaryNode t )
   private BinaryNode findMax( BinaryNode t )
   private BinaryNode find( Comparable x, BinaryNode t )

  protected BinaryNode root;
}
```

Binary Search Trees

```
public static void main( String [ ] args ) {
        BinarySearchTree t = new BinarySearchTree( );
        final int NUMS = 4000;
        final int GAP  =   37;
        System.out.println( "Checking... (no more output means success)" );
        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
           t.insert( new Integer( i ) );
        for( int i = 1; i < NUMS; i+= 2 )
           t.remove( new Integer( i ) );
        if( ((Integer)(t.findMin( ))).intValue( ) != 2 ||
           ((Integer)(t.findMax( ))).intValue( ) != NUMS - 2 )
           System.out.println( "FindMin or FindMax error!" );
        for( int i = 2; i < NUMS; i+=2 )
            if( ((Integer)(t.find( new Integer( i ) ))).intValue( ) != i )
               System.out.println( "Find error1!" );
        for( int i = 1; i < NUMS; i+=2 )
        {
           if( t.find( new Integer( i ) ) != null )
              System.out.println( "Find error2!" );
        }
    }
}
```

Binary Search Trees

```
protected BinaryNode insert( Comparable x, BinaryNode t )   {
      if( t == null )
         t = new BinaryNode( x );
      else if( x.compareTo( t.element ) < 0 )
         t.left = insert( x, t.left );
      else if( x.compareTo( t.element ) > 0 )
         t.right = insert( x, t.right );
      else   throw new DuplicateItemException( x.toString( ) );  // Duplicate
      return t;
   }
protected BinaryNode remove( Comparable x, BinaryNode t )   {
      if( t == null )   throw new ItemNotFoundException( x.toString( ) );
      if( x.compareTo( t.element ) < 0 )    t.left = remove( x, t.left );
      else if( x.compareTo( t.element ) > 0 )    t.right = remove( x, t.right );
      else if( t.left != null && t.right != null )    {
         t.element = findMin( t.right ).element;
         t.right = removeMin( t.right );
      }
      else
         t = ( t.left != null ) ? t.left : t.right;
      return t;
   }
```

Binary Search Trees

```
protected BinaryNode removeMin( BinaryNode t )
   {
      if( t == null )
         throw new ItemNotFoundException( );
      else if( t.left != null )
      {
         t.left = removeMin( t.left );
         return t;
      }
      else
         return t.right;
   }
```

```
protected BinaryNode findMin( BinaryNode t )
   {
      if( t != null )
         while( t.left != null )
            t = t.left;
      return t;
   }

private BinaryNode
            find( Comparable x, BinaryNode t )
   {
      while( t != null )
      {
         if( x.compareTo( t.element ) < 0 )
            t = t.left;
         else if( x.compareTo( t.element ) > 0 )
            t = t.right;
         else
            return t;    // Match
      }
      return null;        // Not found
   }
```