# Trees (Chapter 9)

A *Tree* is a non-linear structure defined by the concept that each *node* in the tree, other than the first node or *root* node, has exactly one parent

For trees, the operations are dependent upon the type of tree and it's use

In order to discuss trees, we must first have a common vocabulary

We have already introduced a couple of terms:

- *node* which refers to a location in the tree where an element is stored, and
- *root* which refers to the node at the base of the tree or the one node in the tree that does not have a parent
- Each node of the tree points to the nodes that are directly beneath it in the tree
- These nodes are referred to as its *children*
- A child of a child is then called a *grandchild*, a child of a grandchild called a *great-grandchild*
- A node that does not have at least one child is called a *leaf*
- A node that is not the root and has at least one child is called an *internal node*

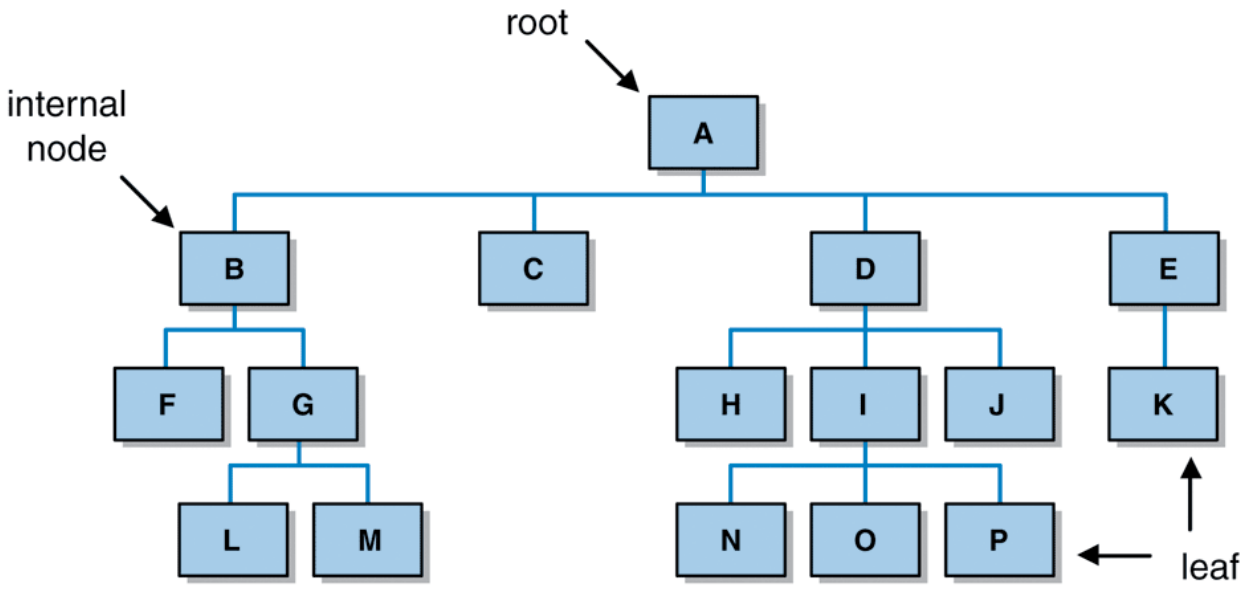


**FIGURE 9.1** Tree terminology

- Any node below another node and on a path from that node is called a *descendant* of that node
- Any node above another node on a connecting path from the root to that node is called an *ancestor* of that node
- All children of the same node are called *siblings*
- A tree that limits each node to no more than n children is called an n-ary tree

Each node of the tree is at a specific *level* or *depth* within the tree

The level of a node is the length of the path from the root to the node

This *pathlength* is determined by counting the number of links that must be followed to get from the root to the node

The root is considered to be level 0, the children of the root are at level 1, the grandchildren of the root are at level 2, and so on.

The *height* or *order* of a tree is the length of the longest path from the root to a leaf

Thus the height or order of the tree in the next slide is 3

The path from the root (A) to leaf (F) is of length 3

The path from the root (A) to leaf (C) is of length 1

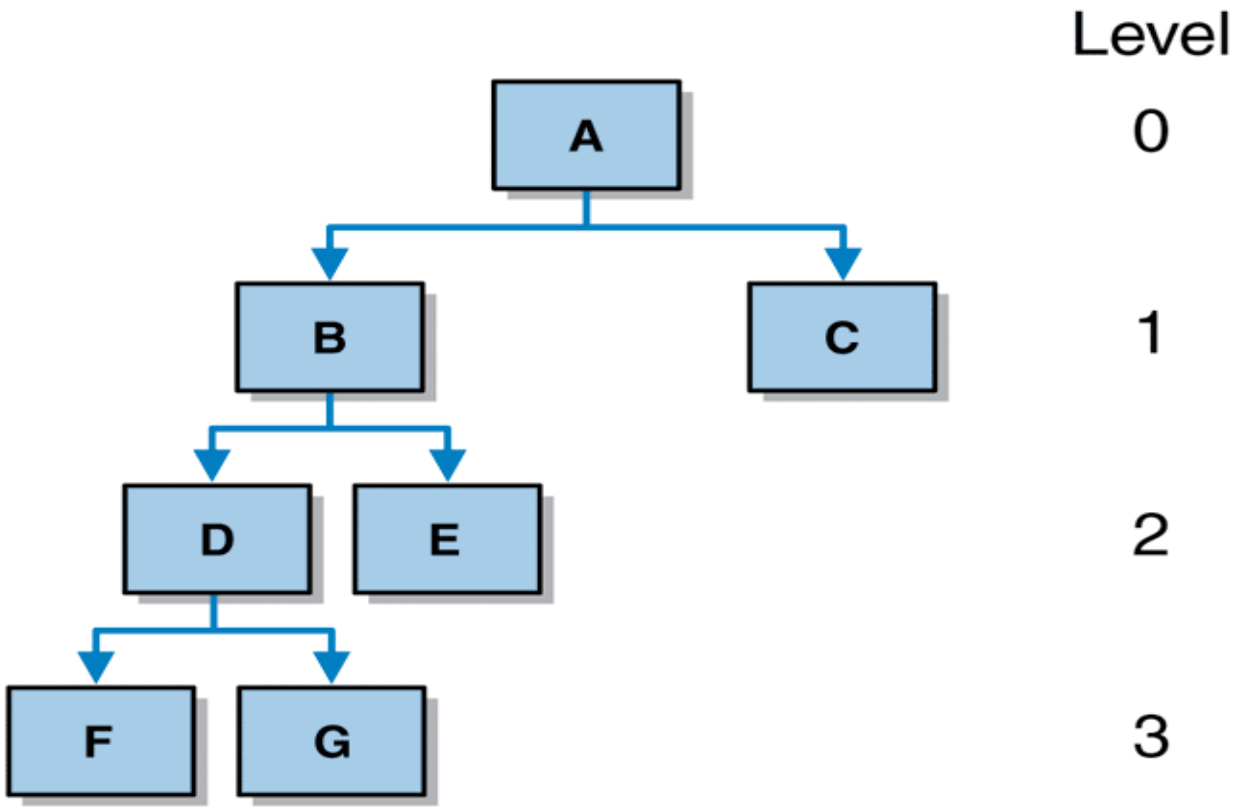


**FIGURE 9.2** Path length and level

A tree is considered to be *balanced* if all of the leaves of the tree are at roughly the same depth

While the use of the term “roughly” may not be intellectually satisfying, the actual definition is dependent upon the algorithm being used

Some algorithms define balanced as all of the leaves being at level h or h-1 where h is the height of the tree  and where h = $\log_N n$ for an N-ary tree.

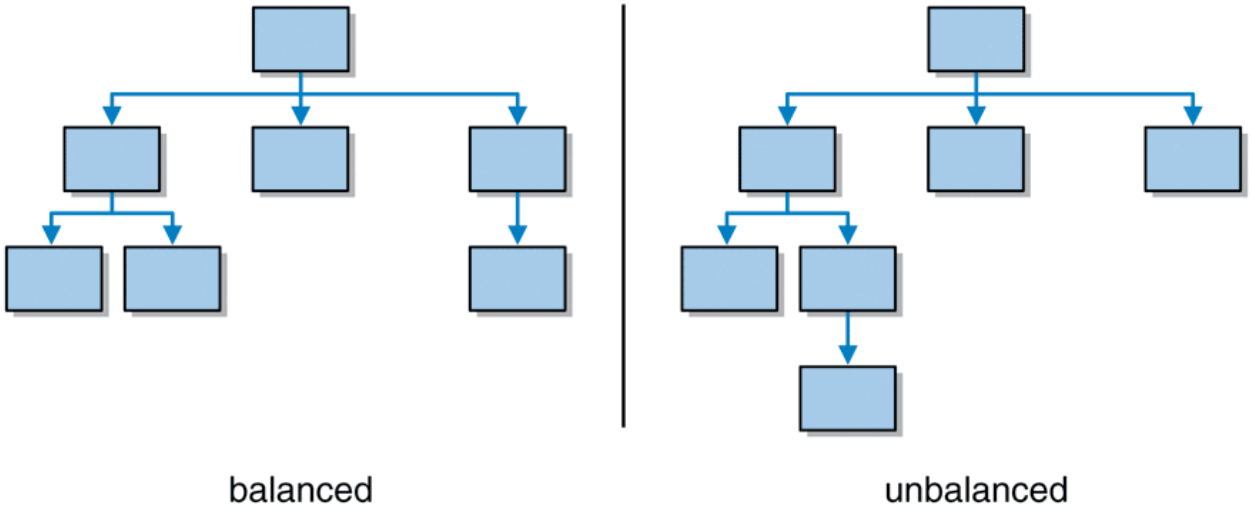


**FIGURE 9.3** Balanced and unbalanced trees

The concept of a *complete* tree is related to the balance of a tree

A tree is considered *complete* if it is balanced and all of the leaves at level h are on the left side of the tree

While a seemingly arbitrary concept, as we will discuss in later chapters, this definition has implications for how the tree is stored in certain implementations

Tree a and c on the next slide are complete while tree b is not

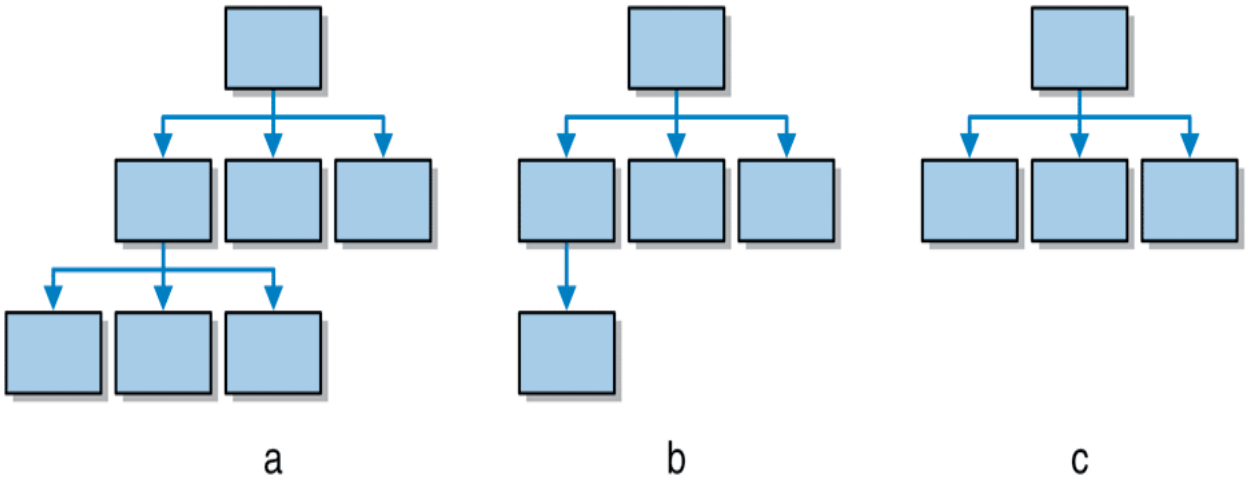


**FIGURE 9.4** Some complete trees

**Implementing Trees with Arrays**

For certain types of trees, specifically binary trees, a computational strategy can be used for storing a tree using an array

For any element stored in position n of the array, that elements left child will be stored in position ((2*n) + 1) and that elements right child will be stored in position (2*(n+1))

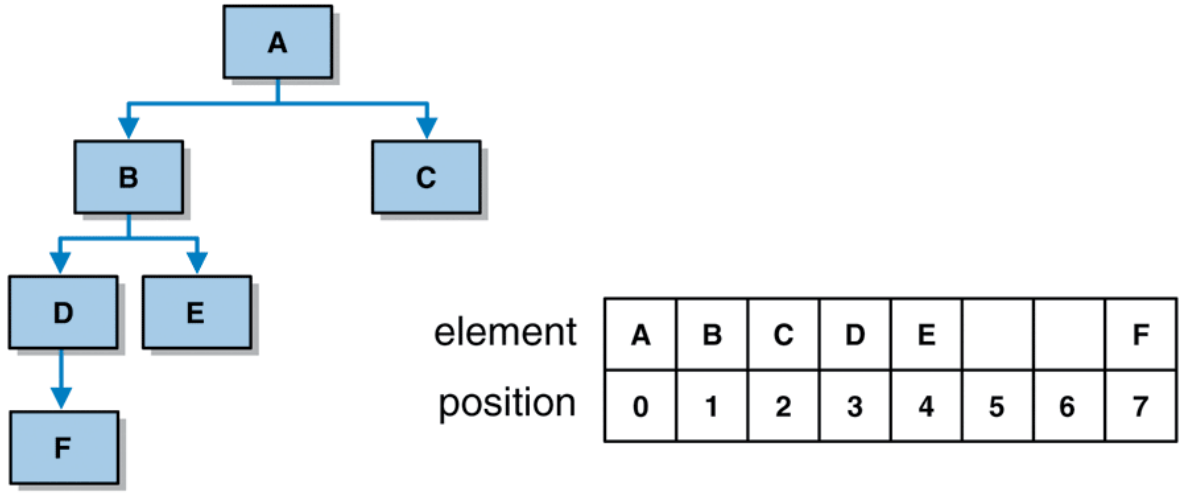


**FIGURE 9.5** Computational strategy for array implementation of trees

Space is wasted for “sparse” trees.

There are four basic algorithms for traversing a tree:

- Preorder traversal
- Inorder traversal
- Postorder traversal
- Levelorder traversal

- Preorder traversal is accomplished by visiting each node, followed by its children, starting with the root
- Given the complete binary tree on the next slide, a preorder traversal would produce the order: A B D E C

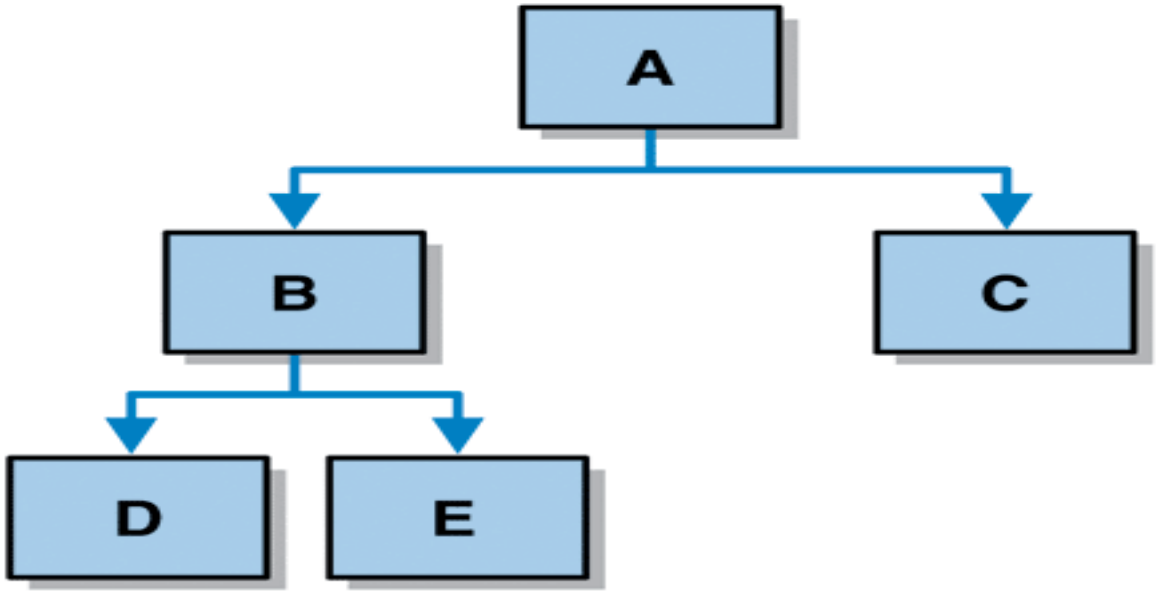


FIGURE 9.7 A complete tree

Stated in pseudocode, the algorithm for a preorder traversal of a binary tree is:

Visit node
Traverse(left child)
Traverse(right child)

Inorder traversal is accomplished by visiting the left child of the node, then the node, then any remaining child nodes starting with the root

An inorder traversal of the previous tree produces the order: D B E A C

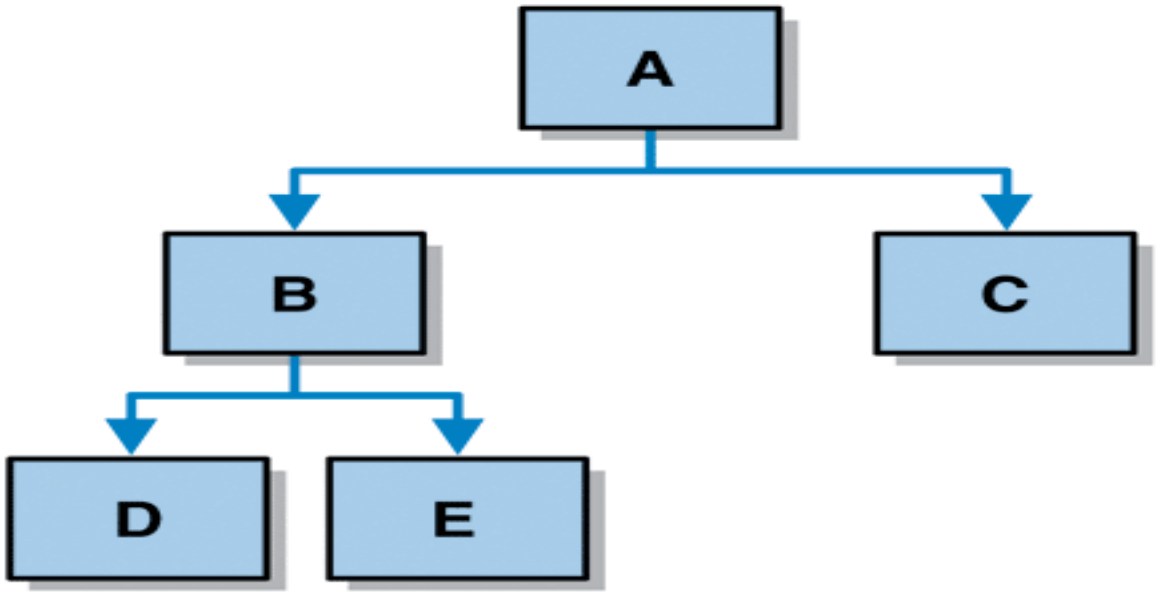


FIGURE 9.7 A complete tree

Stated in pseudocode, the algorithm for an inorder traversal of a binary tree is:

Traverse(left child)
Visit node
Traverse(right child)

Postorder traversal is accomplished by visiting the children, then the node starting with the root

Given the same tree, a postorder traversal produces the following order:
D E B C A

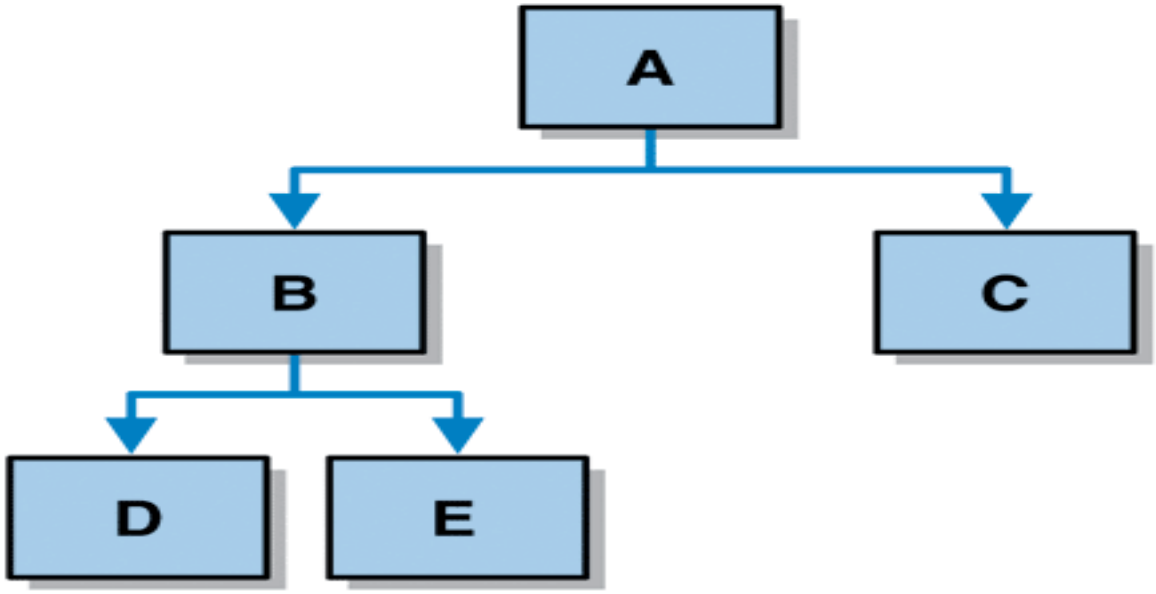


FIGURE 9.7 A complete tree

Stated in pseudocode, the algorithm for a postorder traversal of a binary tree is:

Traverse(left child)
Traverse(right child)
Visit node

Levelorder traversal is accomplished by visiting all of the nodes at each level, one level at at time, starting with the root

Given the same tree, a levelorder traversal produces the order:A B C D E

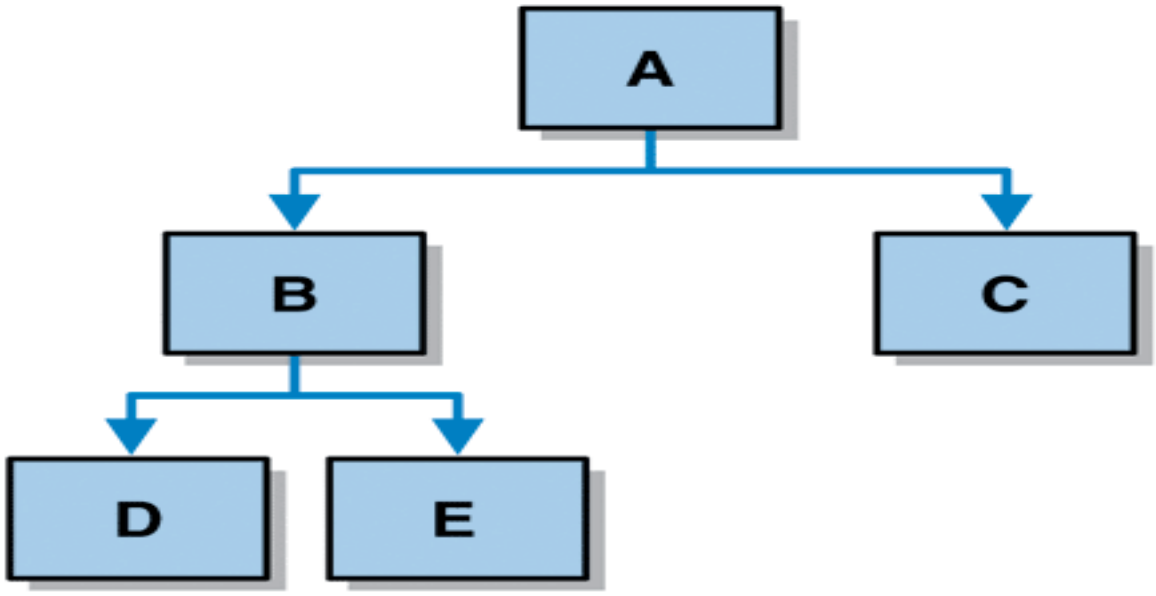


FIGURE 9.7 A complete tree

Stated in pseudocode, the algorithm for a level order traversal of a binary tree is:

```
Create a queue called nodes
Create an unordered list called results
Enqueue the root onto the nodes queue
While the nodes queue is not empty
{
  Dequeue the first element from the queue
  If that element is not null
     Add that element to the rear of the results list
     Enqueue the children of the element on the nodes queue
  Else
     Add null on the result list
}
Return an iterator for the result list
```

**Implementing trees with linked nodes**

```
public class BTNode<E> {
  private E data;
  private BTNode<E> left, right;
  public E getData( )  {
    return data;
  }

  public BTNode<E> getLeft( )  {     return left; }

  public E getLeftmostData( )  {
    if (left == null)
      return data;
    else
      return left.getLeftmostData( );
  }

  public BTNode<E> getRight( )  { return right;  }

  public E getRightmostData( )  {
    if (left == null)
      return data;
    else
      return left.getRightmostData( );
  }

  public boolean isLeaf( )  {      return (left == null) && (right == null);    }

 public void setData(E newData)      { data = newData; }

  public void setLeft(BTNode<E> newLeft)  {  left = newLeft;  }

  public void setRight(BTNode<E> newRight)  {  right = newRight;  }
```

```
public void inorderPrint( )   {
     if (left != null)
       left.inorderPrint( );
     System.out.println(data);
     if (right != null)
       right.inorderPrint( );
   }

   public void preorderPrint( )   {
     System.out.println(data);
     if (left != null)
       left.preorderPrint( );
     if (right != null)
       right.preorderPrint( );
   }

   public void postorderPrint( )   {
     if (left != null)
       left.postorderPrint( );
     if (right != null)
       right.postorderPrint( );
     System.out.println(data);
   }
```

```
// 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> rt )
      { 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;
}
```

**figure 18.12**

The `BinaryNode` class skeleton

**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 }
```

**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    }
```

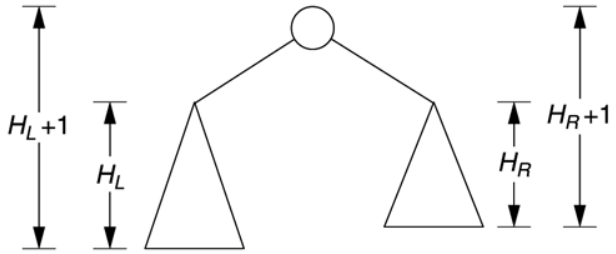


**figure 18.20**

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

```
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

**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    }
```

```
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 }
```

**figure 18.24**

The tree iterator abstract base class

```
// PreOrder class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS*********************
// boolean isValid( )   --> True if at valid position in tree
// AnyType 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

class PreOrder<AnyType> extends TreeIterator<AnyType>
{
    /**
     * Construct the iterator. The current position is set to null.
     */
    public PreOrder( BinaryTree<AnyType> theTree )
    {
        super( theTree );
        s = new ArrayStack<BinaryNode<AnyType>>( );
        s.push( t.getRoot( ) );
    }

    /**
     * Set the current position to the first item, according
     * to the preorder traversal scheme.
     */
    public void first( )
    {
        s.makeEmpty( );
        if( t.getRoot( ) != null )
        {
            s.push( t.getRoot( ) );
            advance( );
        }
    }

    public void advance( )
      { /* Figure 18.30 */ }

    private Stack<BinaryNode<AnyType>> s; // Stack of BinaryNode objects
}
```

**figure 18.29**

The `PreOrder` class skeleton and all members except `advance`

**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      }
```

Binary Search Trees

A binary search tree is a binary tree with the added property that for each node, the left child is less than the parent is less than or equal to the right child

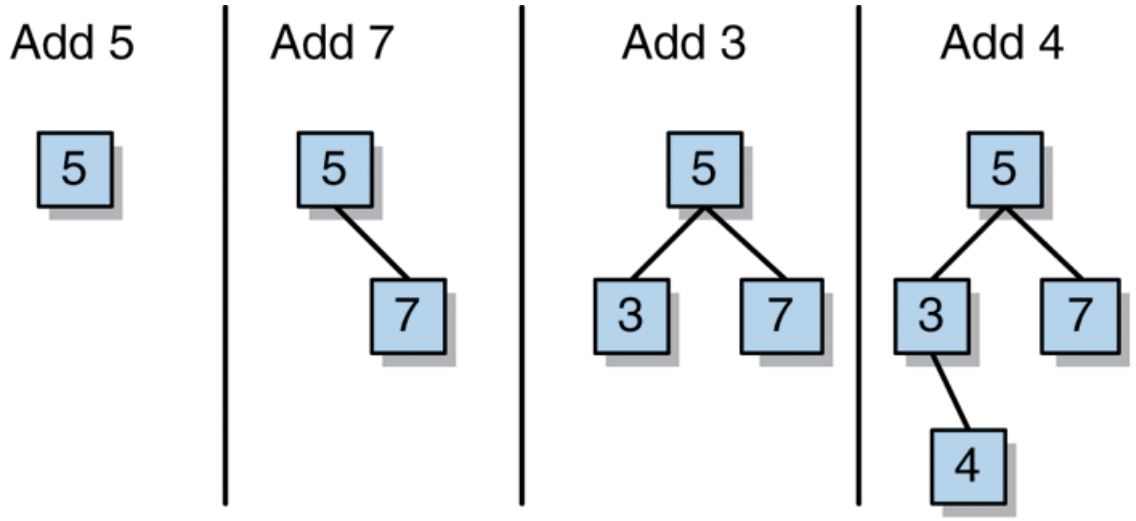


**FIGURE 10.3** Adding elements to a binary search tree

Removing elements from a binary search tree requires

- Finding the element to be removed
- If that element is not a leaf, then replace it with its inorder successor
- Return the removed element

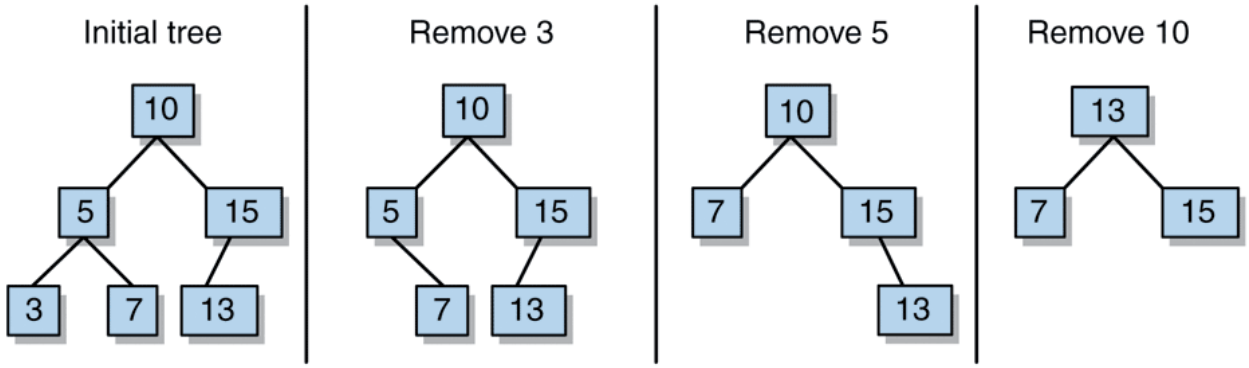


**FIGURE 10.4** Removing elements from a binary tree

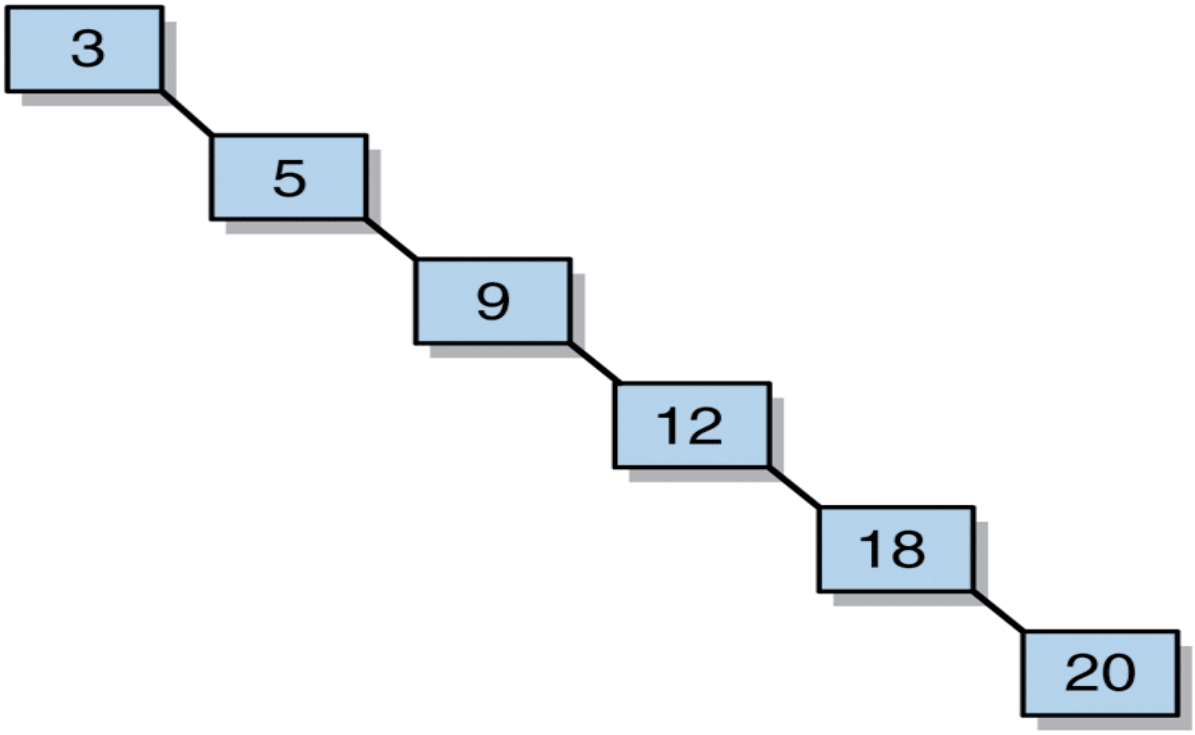


**FIGURE 10.9** A degenerate binary tree

Worst Case: O(n)
Average Case: O(lgn)

Method **add():** finds the proper location for the given element and adds it there as a leaf.

```
public void addElement (E element)  {
   BTNode <E> temp = new BTNode <E> (element);
   Comparable<E> comparableElement = (Comparable<E>) element;
   if (isEmpty())
        root = temp;
   else  { // not empty
        BinaryTreeNode<T> current = root;
        boolean added = false;
        while (!added)   {
          if (comparableElement.compareTo(current.element) < 0)    {
            if (current.left == null)  {
              current.left = temp;
              added = true;
            }
            else
              current = current.left;
          }
          else {
            if (current.right == null) {
              current.right = temp;
              added = true;
            }
            else
              current = current.right;
          }
        } // while
      } // else not empty
      count++;
}
```

Weiss:

```
1  package weiss.nonstandard;
2
3  // Basic node stored in unbalanced binary search trees
4  // Note that this class is not accessible outside
5  // this package.
6
7  class BinaryNode<AnyType>
8  {
9          // Constructor
10     BinaryNode( AnyType theElement )
11     {
12         element = theElement;
13         left = right = null;
14     }
15
16       // Data; accessible by other package routines
17     AnyType             element;  // The data in the node
18     BinaryNode<AnyType> left;     // Left child
19     BinaryNode<AnyType> right;    // Right child
20 }
```

**figure 19.5**

The `BinaryNode` class for the binary search tree

```
1  package weiss.nonstandard;
2
3  // BinarySearchTree class
4  //
5  // CONSTRUCTION: with no initializer
6  //
7  // ******************PUBLIC OPERATIONS*********************
8  // void insert( x )       --> Insert x
9  // void remove( x )       --> Remove x
10 // void removeMin( )      --> Remove minimum item
11 // Comparable find( x )   --> Return item that matches x
12 // Comparable findMin( )  --> Return smallest item
13 // Comparable findMax( )  --> Return largest item
14 // boolean isEmpty( )     --> Return true if empty; else false
15 // void makeEmpty( )      --> Remove all items
16 // ******************ERRORS********************************
17 // Exceptions are thrown by insert, remove, and removeMin if warranted
18
19 public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
20 {
21     public BinarySearchTree( )
22       {  root = null; }
23
24     public void insert( AnyType x )
25       { root = insert( x, root ); }
```

**figure 19.6a**

The `BinarySearchTree` class skeleton (*continues*)

```
26      public void remove( AnyType x )
27        { root = remove( x, root ); }
28      public void removeMin( )
29        { root = removeMin( root ); }
30      public AnyType findMin( )
31        { return elementAt( findMin( root ) ); }
32      public AnyType findMax( )
33        { return elementAt( findMax( root ) ); }
34      public AnyType find( AnyType x )
35        { return elementAt( find( x, root ) ); }
36      public void makeEmpty( )
37        { root = null; }
38      public boolean isEmpty( )
39        { return root == null; }
40
41      private AnyType elementAt( BinaryNode<AnyType> t )
42        { /* Figure 19.7 */ }
43      private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
44        { /* Figure 19.8 */ }
45      protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
46        { /* Figure 19.9 */ }
47      private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
48        { /* Figure 19.9 */ }
49      protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
50        { /* Figure 19.10 */ }
51      protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
52        { /* Figure 19.11 */ }
53      protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
54        { /* Figure 19.12 */ }
55
56      protected BinaryNode<AnyType> root;
57  }
```

**figure 19.6b**

The `BinarySearchTree` class skeleton (*continued*)

```
1     /**
2      * Internal method to get element field.
3      * @param t the node.
4      * @return the element field or null if t is null.
5      */
6     private AnyType elementAt( BinaryNode<AnyType> t )
7     {
8         return t == null ? null : t.element;
9     }
```

**figure 19.7**
The `elementAt` method

```
1     /**
2      * Internal method to find an item in a subtree.
3      * @param x is item to search for.
4      * @param t the node that roots the tree.
5      * @return node containing the matched item.
6      */
7     private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
8     {
9         while( t != null )
10        {
11            if( x.compareTo( t.element ) < 0 )
12                t = t.left;
13            else if( x.compareTo( t.element ) > 0 )
14                t = t.right;
15            else
16                return t;    // Match
17        }
18
19        return null;         // Not found
20    }
```

**figure 19.8**
The `find` operation for binary search trees

**figure 19.9**

The `findMin` and `findMax` methods for binary search trees

```
1       /**
2        * Internal method to find the smallest item in a subtree.
3        * @param t the node that roots the tree.
4        * @return node containing the smallest item.
5        */
6       protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
7       {
8           if( t != null )
9               while( t.left != null )
10                  t = t.left;
11
12          return t;
13      }
14
15      /**
16       * Internal method to find the largest item in a subtree.
17       * @param t the node that roots the tree.
18       * @return node containing the largest item.
19       */
20      private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
21      {
22          if( t != null )
23              while( t.right != null )
24                  t = t.right;
25
26          return t;
27      }
```

```
1       /**
2        * Internal method to insert into a subtree.
3        * @param x the item to insert.
4        * @param t the node that roots the tree.
5        * @return the new root.
6        * @throws DuplicateItemException if x is already present.
7        */
8       protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
9       {
10          if( t == null )
11              t = new BinaryNode<AnyType>( x );
12          else if( x.compareTo( t.element ) < 0 )
13              t.left = insert( x, t.left );
14          else if( x.compareTo( t.element ) > 0 )
15              t.right = insert( x, t.right );
16          else
17              throw new DuplicateItemException( x.toString( ) );  // Duplicate
18          return t;
19      }
```

**figure 19.10**

The recursive `insert` for the `BinarySearchTree` class

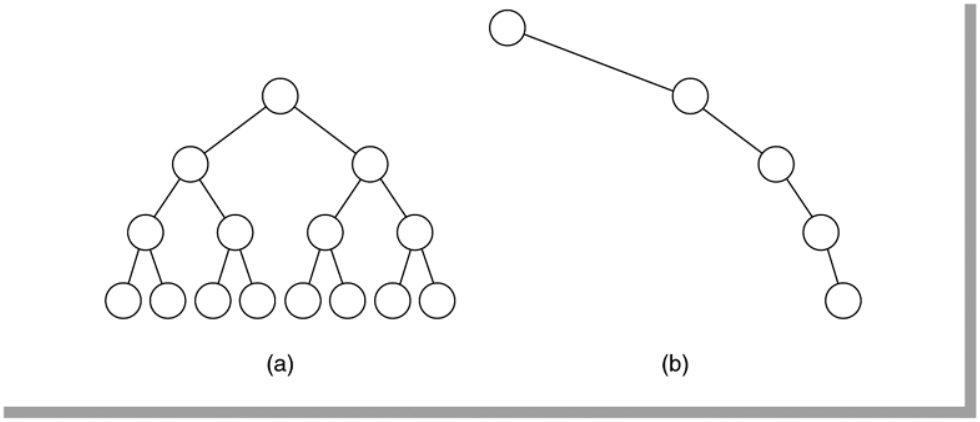


**figure 19.19**

(a) The balanced tree has a depth of $\lfloor \log N \rfloor$; (b) the unbalanced tree has a depth of $N - 1$.

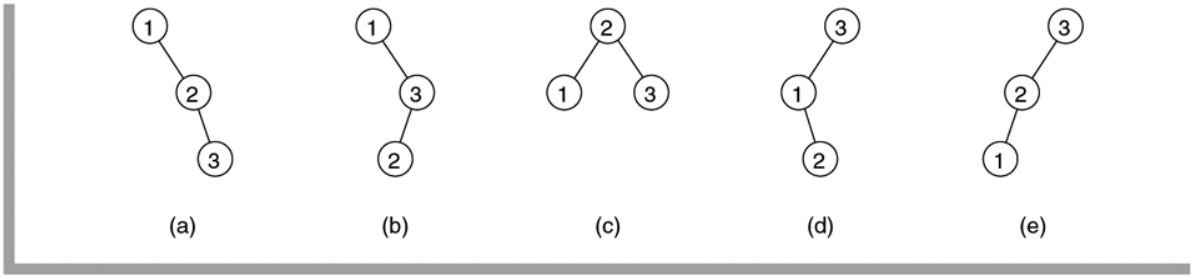


**figure 19.20**

Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree shown in part (c) is twice as likely to result as any of the others.