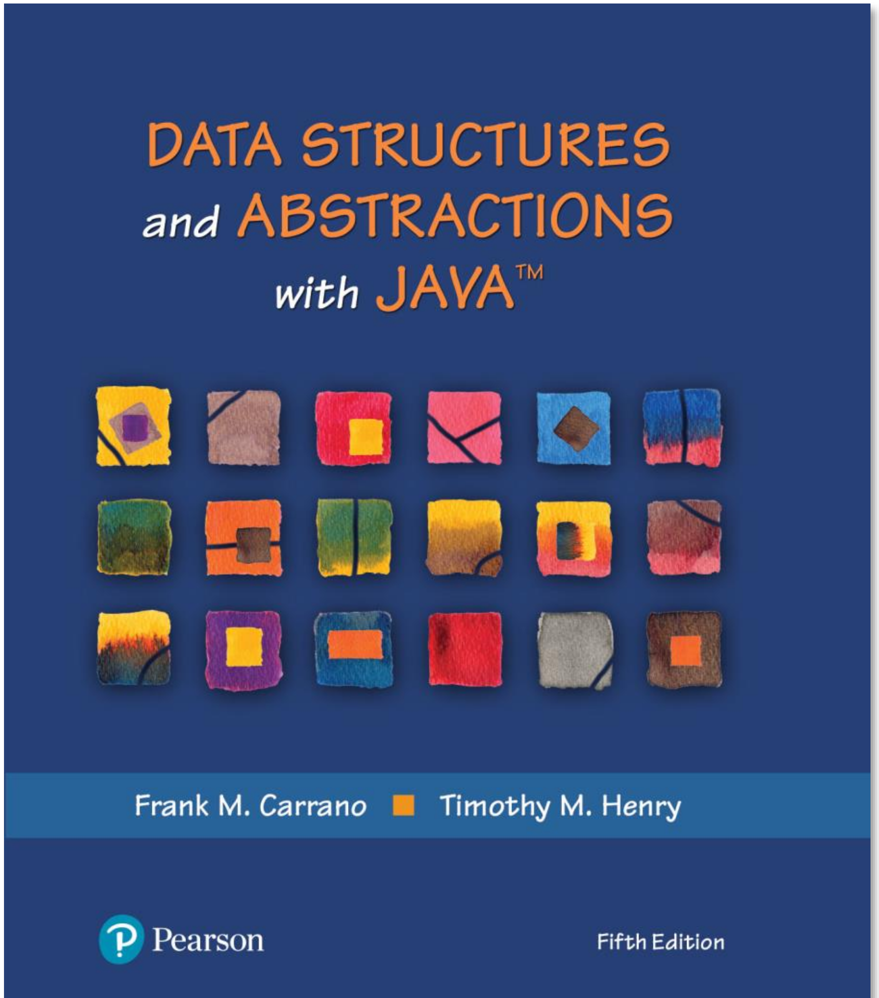
Data Structures and Abstractions with Java™

5th Edition



Chapter 24

Trees

Hierarchical Organizations

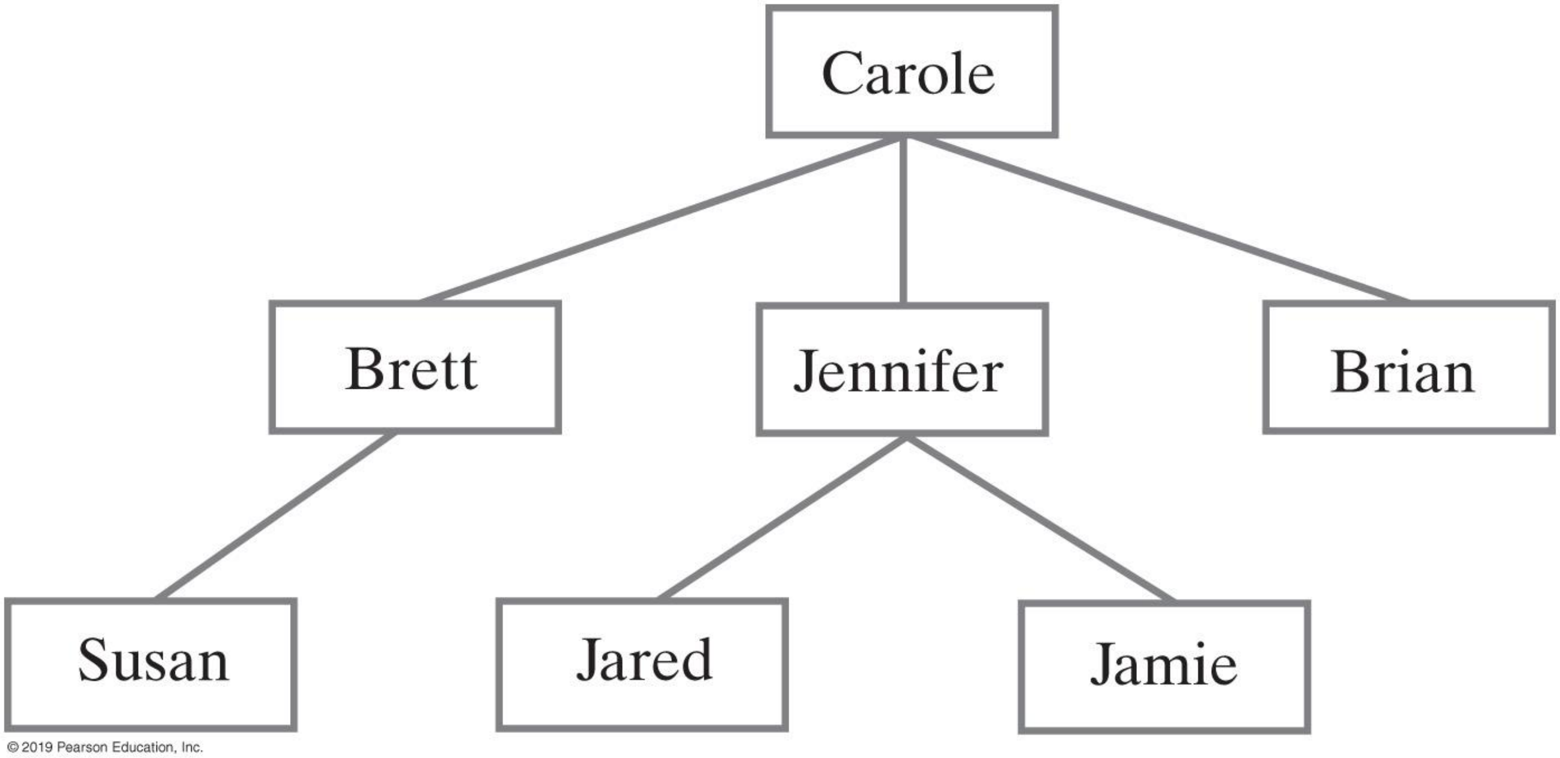


FIGURE 24-1 Carole's children and grandchildren

Hierarchical Organizations

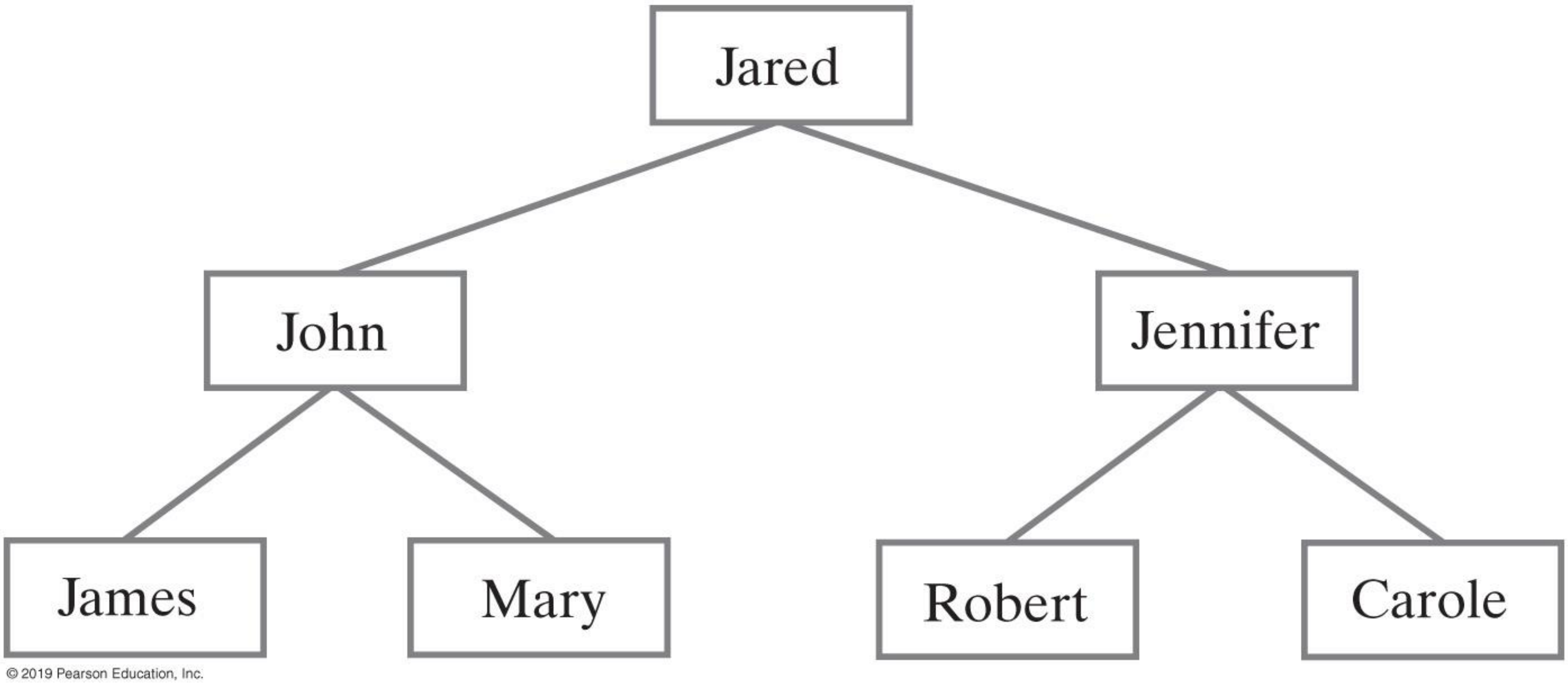


FIGURE 24-2 Jared's parents and grandparents

Hierarchical Organizations

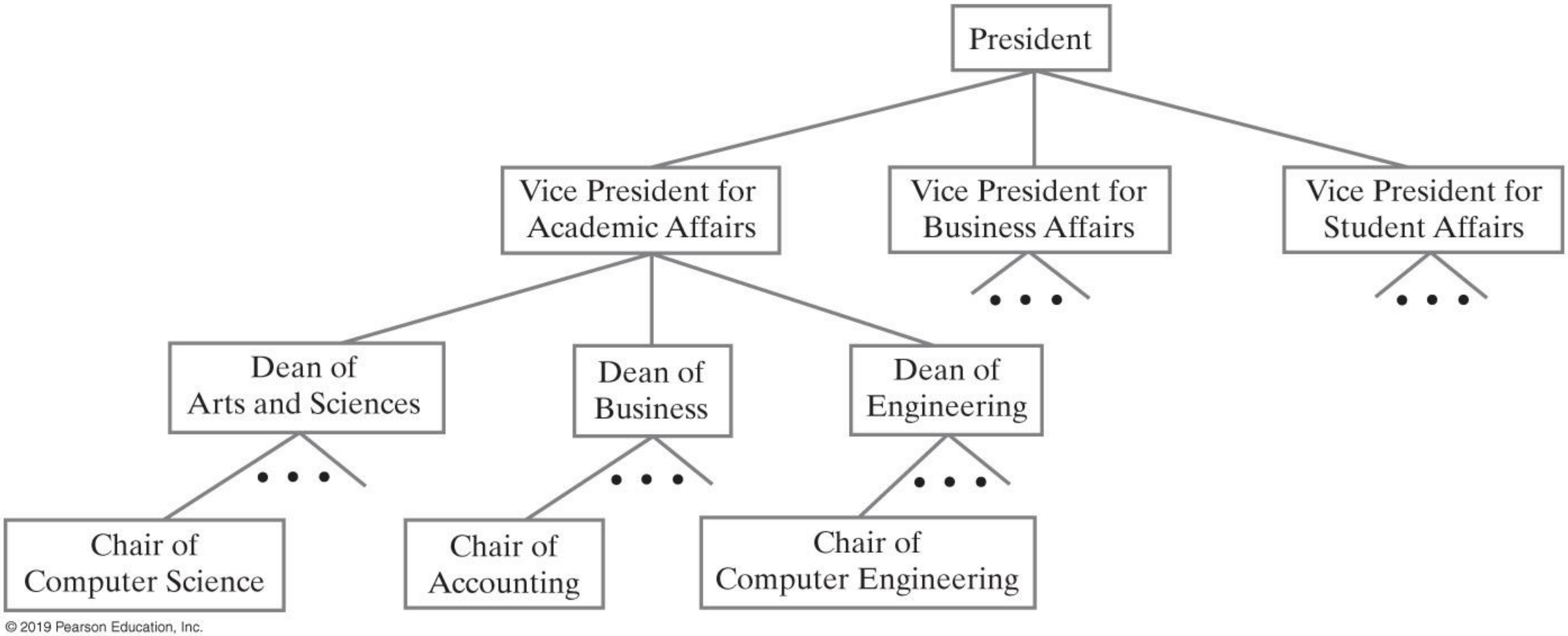


FIGURE 24-3 A portion of a university's administrative structure

Hierarchical Organizations

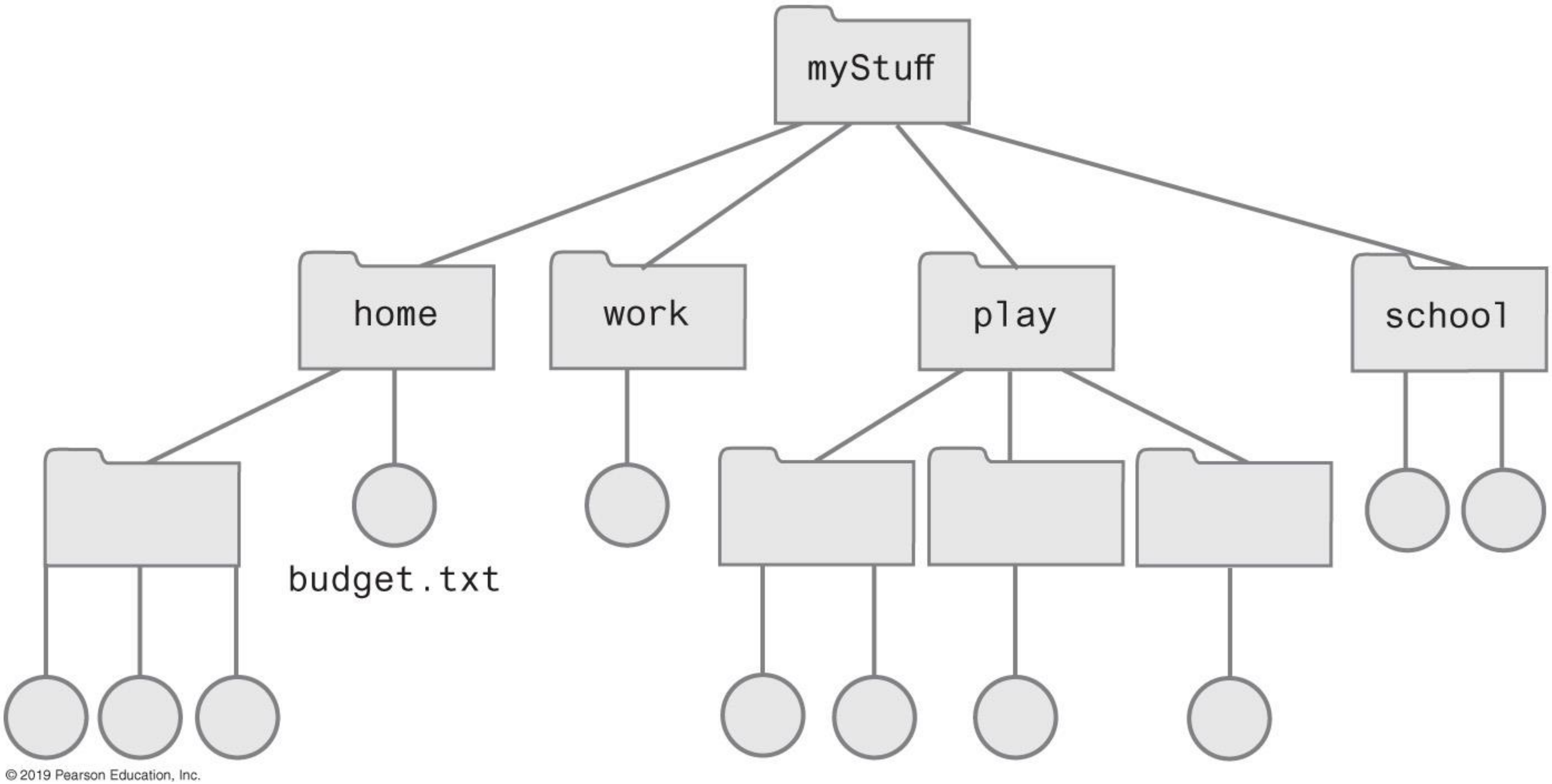
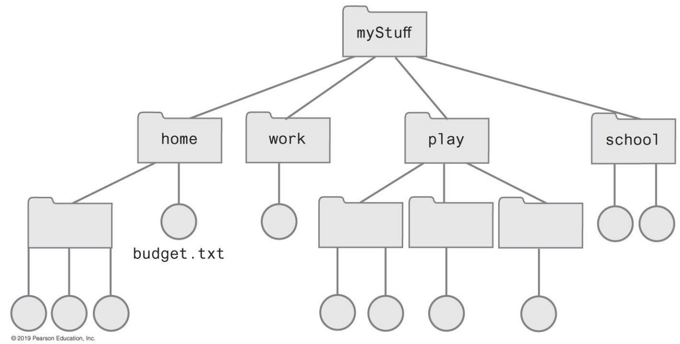


FIGURE 24-4 Computer files organized into folders

Tree Terminology



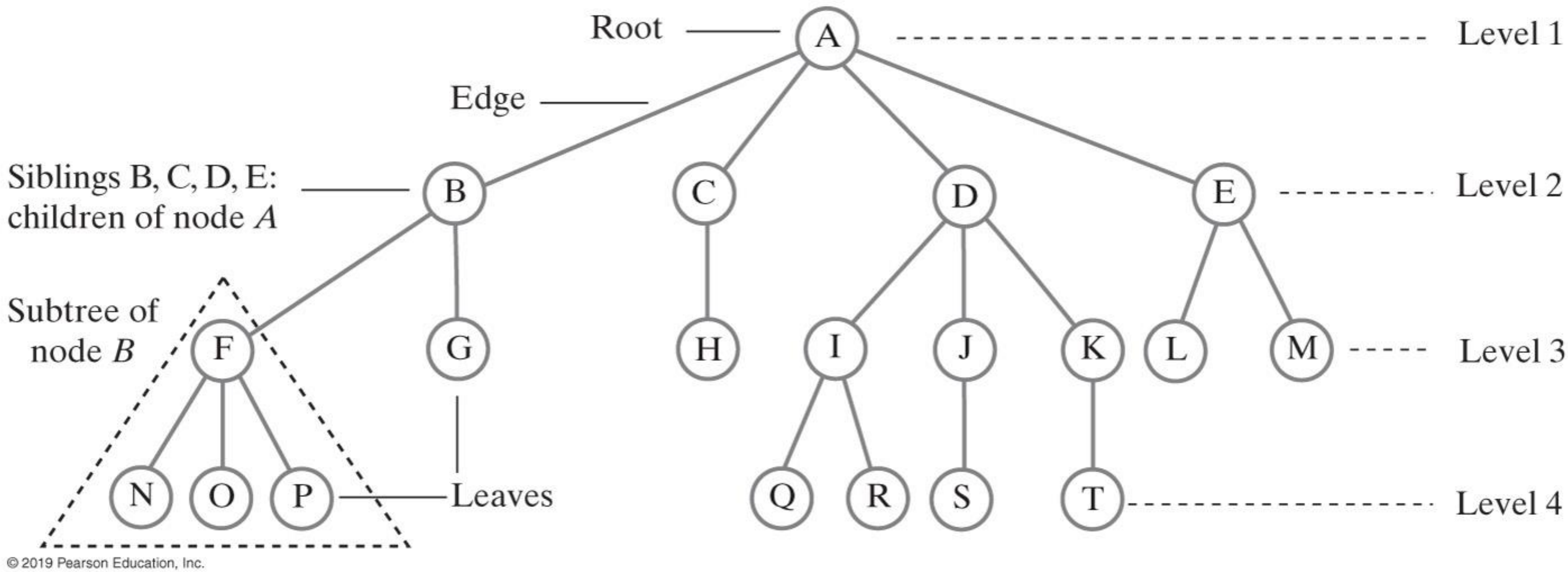
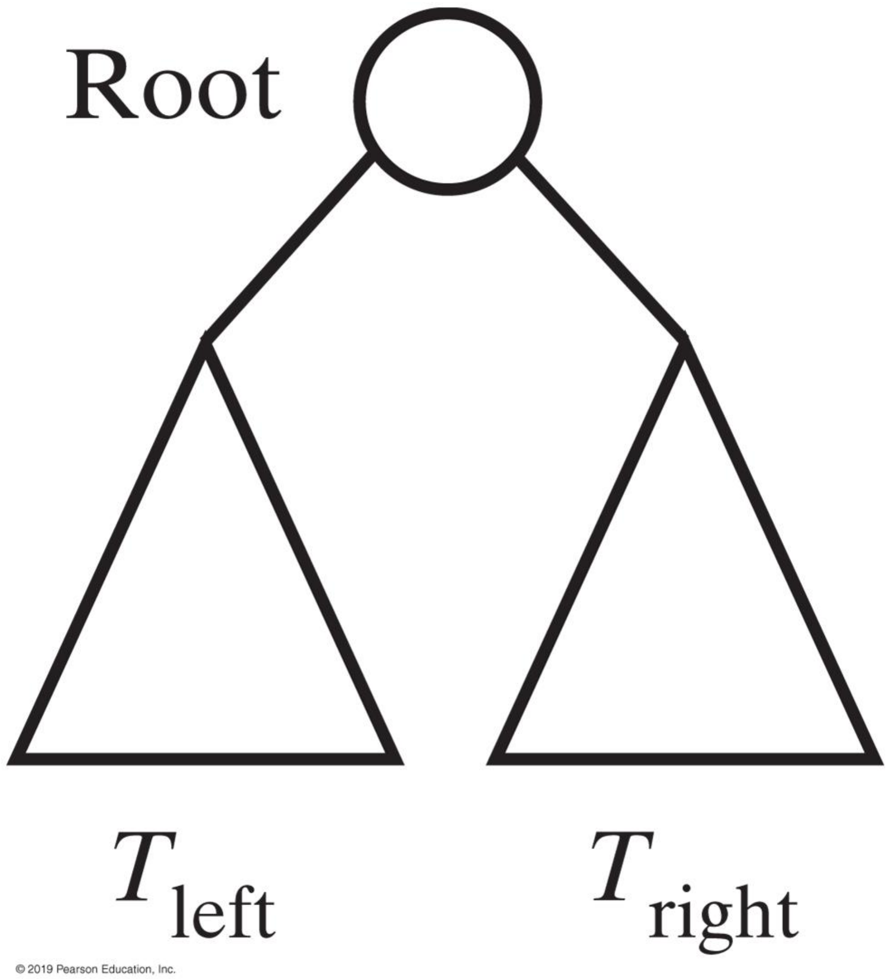


FIGURE 24-5 A tree equivalent to the tree in Figure 24-4

Tree Terminology

- Contrast plants with root at bottom
  - ADT tree with root at top
  - Root is only node with no parent
- A tree can be empty
- Any node and its descendants form a subtree of the original tree
- The height of a tree is the number of levels in the tree

Binary trees



Binary Trees

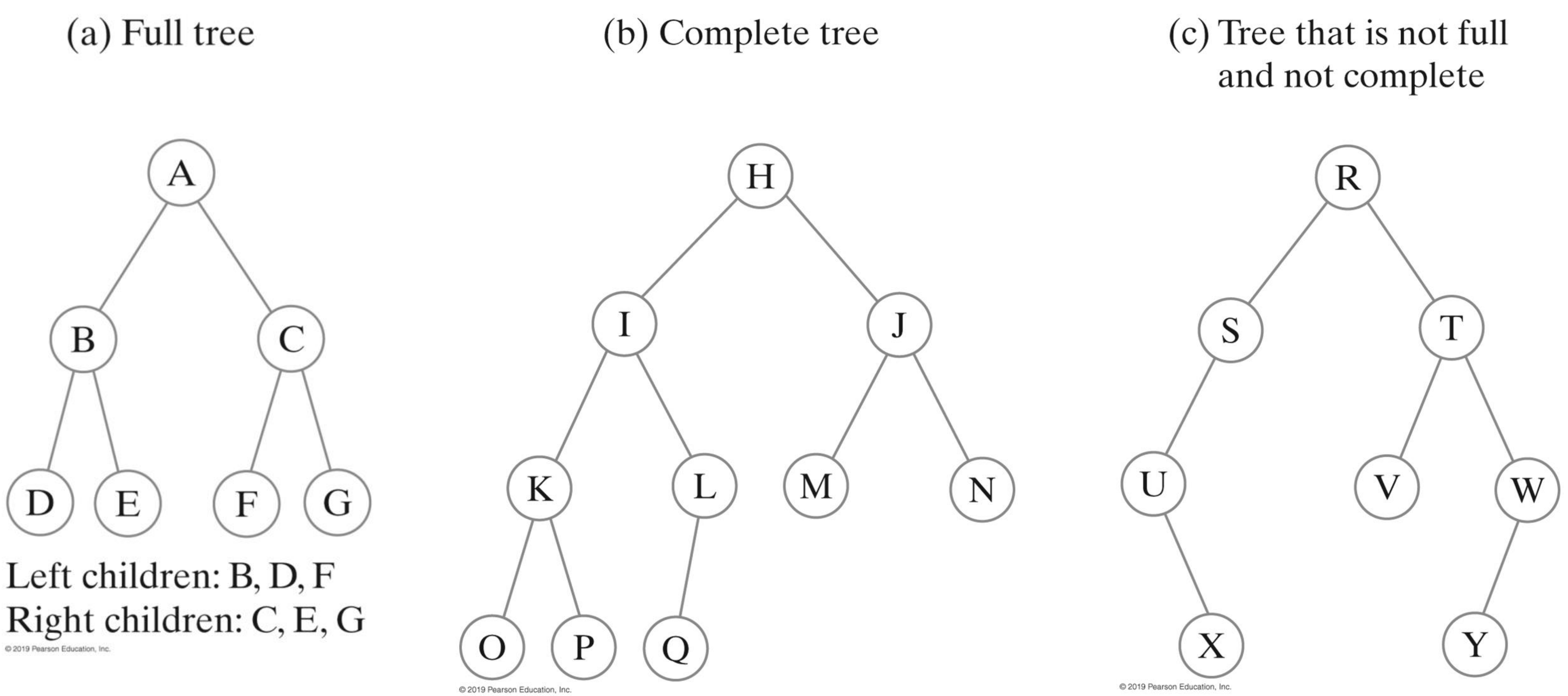
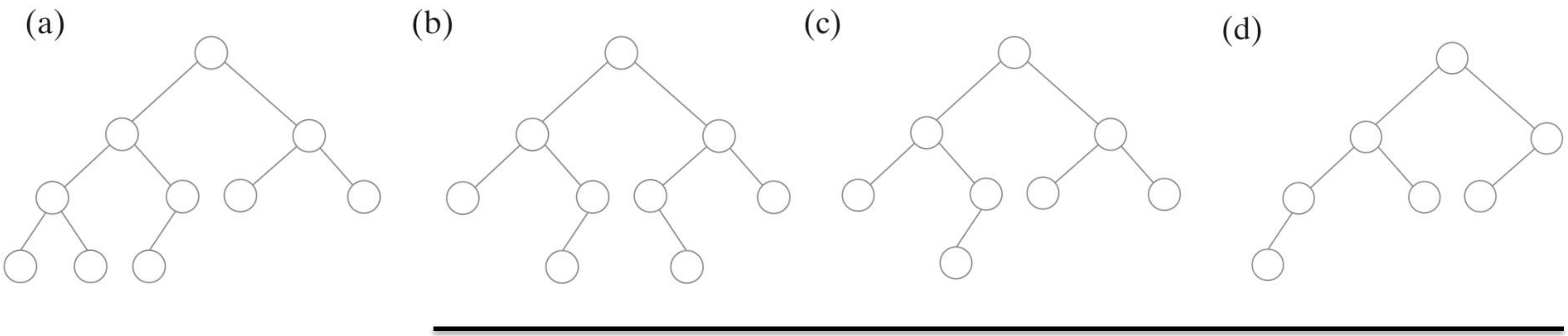


FIGURE 24-6 Three binary trees

Binary Trees

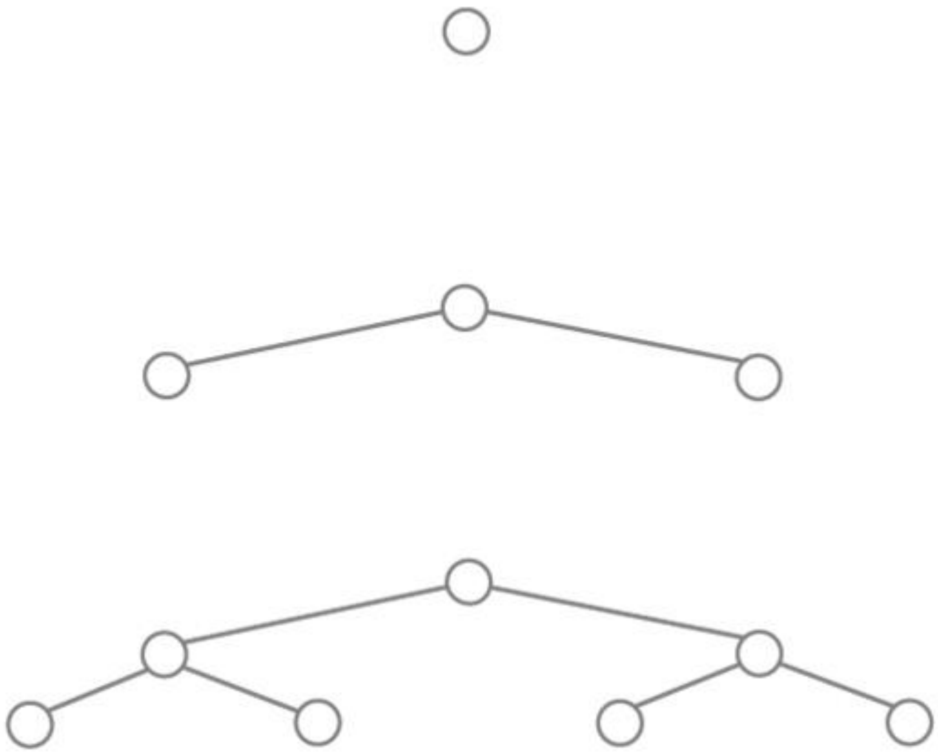


Balanced and complete

Balanced, but not complete

FIGURE 24-7 Some binary trees that are height balanced

Binary Tree Height (Part 1)



Full Tree	Height	Number of Nodes
	1	$1 = 2^1 - 1$
	2	$3 = 2^2 - 1$
	3	$7 = 2^3 - 1$

FIGURE 24-8 The number of nodes in a full binary tree as a function of the tree's height

Binary Tree Height (Part 2)

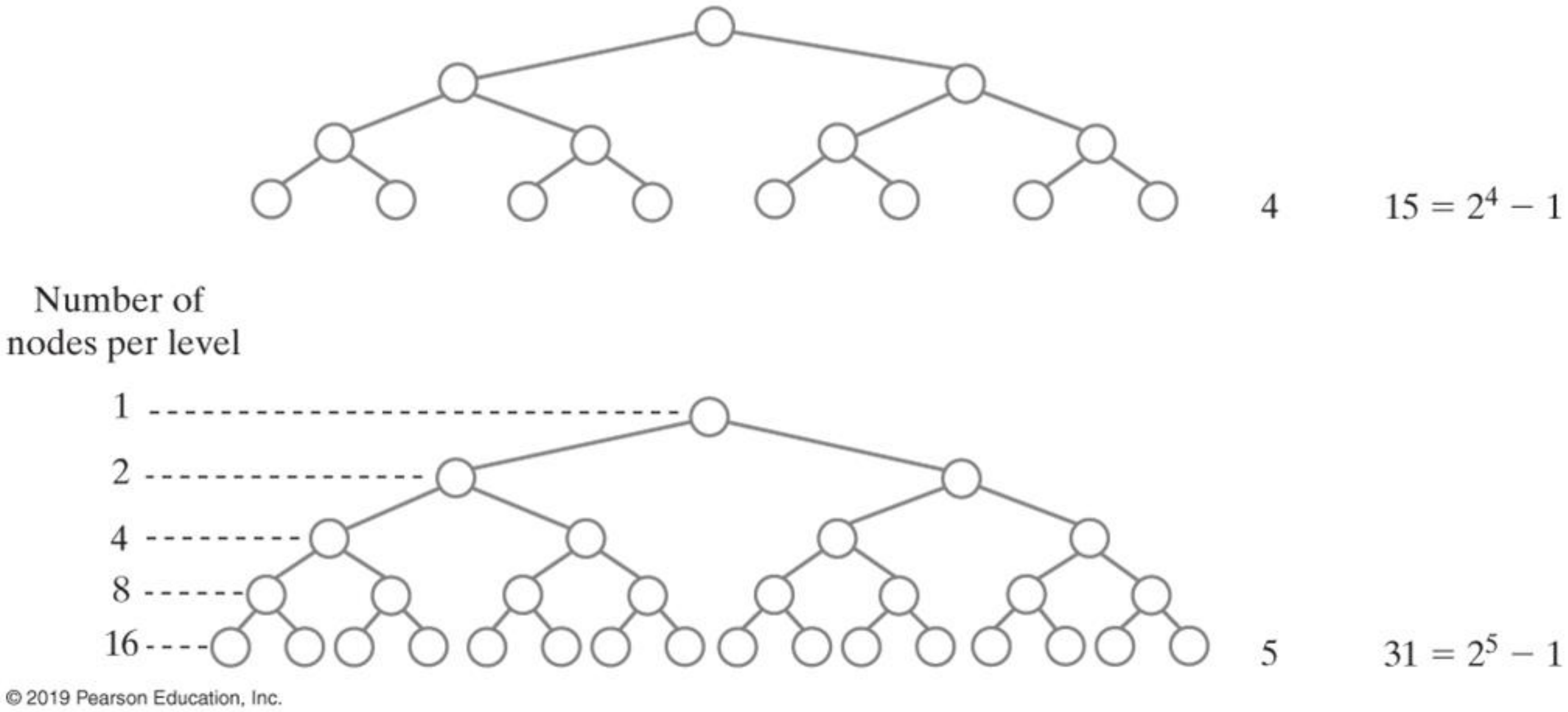


FIGURE 24-8 The number of nodes in a full binary tree as a function of the tree's height

Traversals of A Tree

- Traversal:
  - Visit, or process, each data item exactly once
- We will say that traversal can pass through a node without visiting it at that moment.
- Order in which we visit items is not unique
- Traversals of a binary tree are somewhat easy to understand

Traversals of a Binary Tree

- We use recursion
- To visit all the nodes in a binary tree, we must
  - Visit the root
  - Visit all the nodes in the root's left subtree
  - Visit all the nodes in the root's right subtree

Traversals of a Binary Tree

- **Preorder traversal**
  - Visit root before we visit root's subtrees
- **Inorder traversal**
  - Visit root of a binary tree between visiting nodes in root's subtrees.
- **Postorder traversal**
  - Visit root of a binary tree after visiting nodes in root's subtrees
- **Level-order traversal**
  - Begin at root and visit nodes one level at a time

Traversals of a Binary Tree

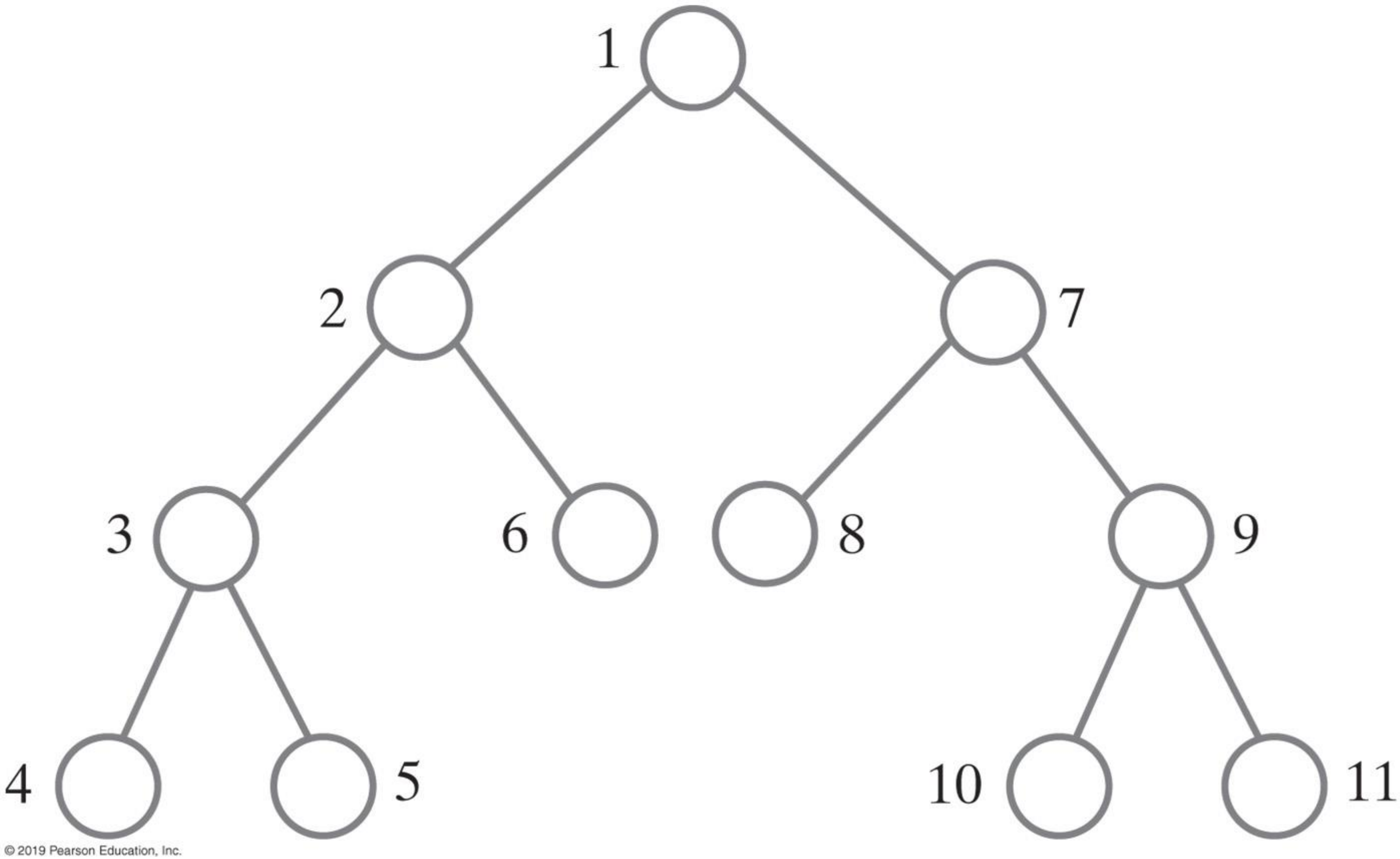


FIGURE 24-9 The visitation order of a preorder traversal

Traversals of a Binary Tree

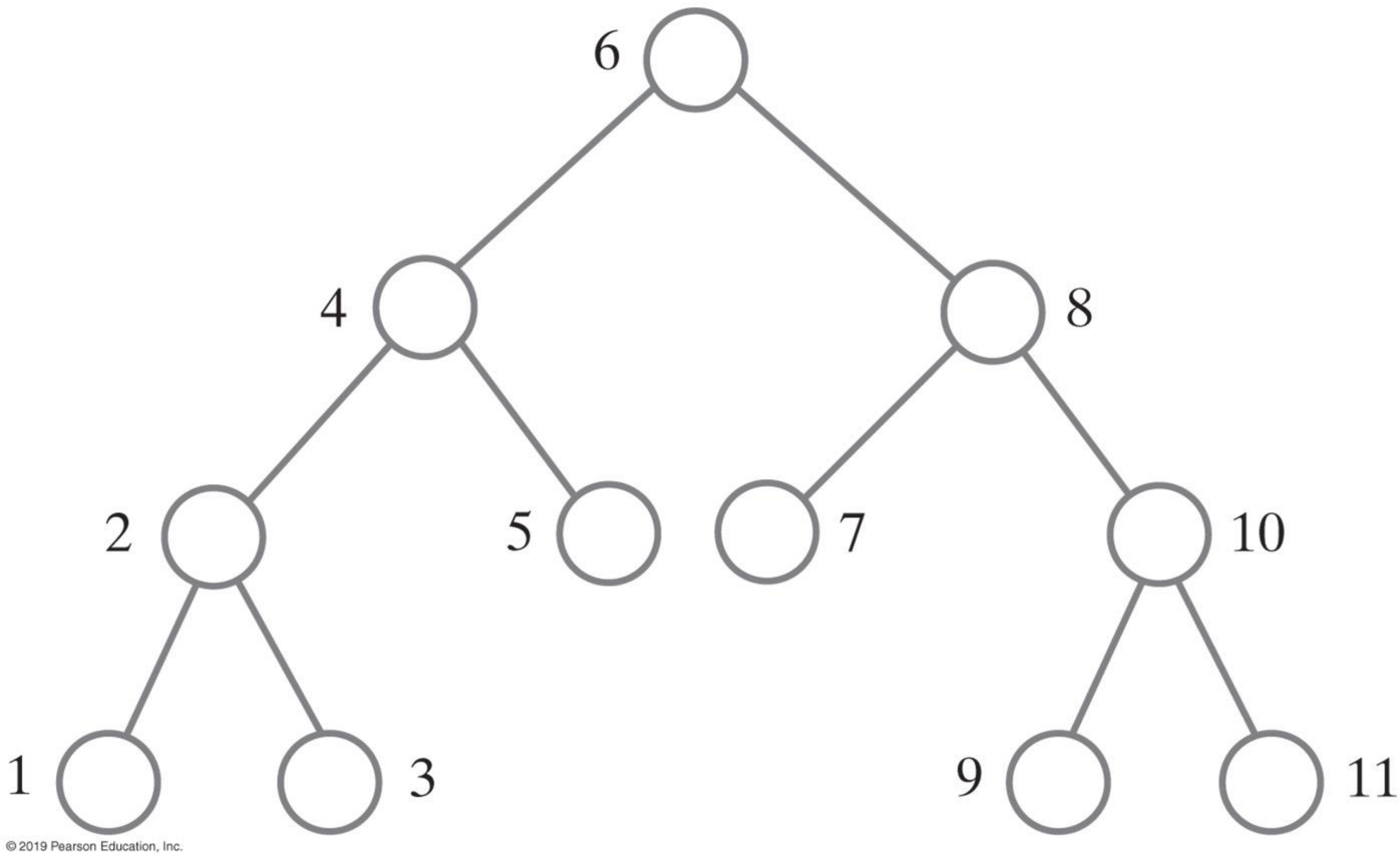


FIGURE 24-10 The visitation order of an in-order traversal

Traversals of a Binary Tree

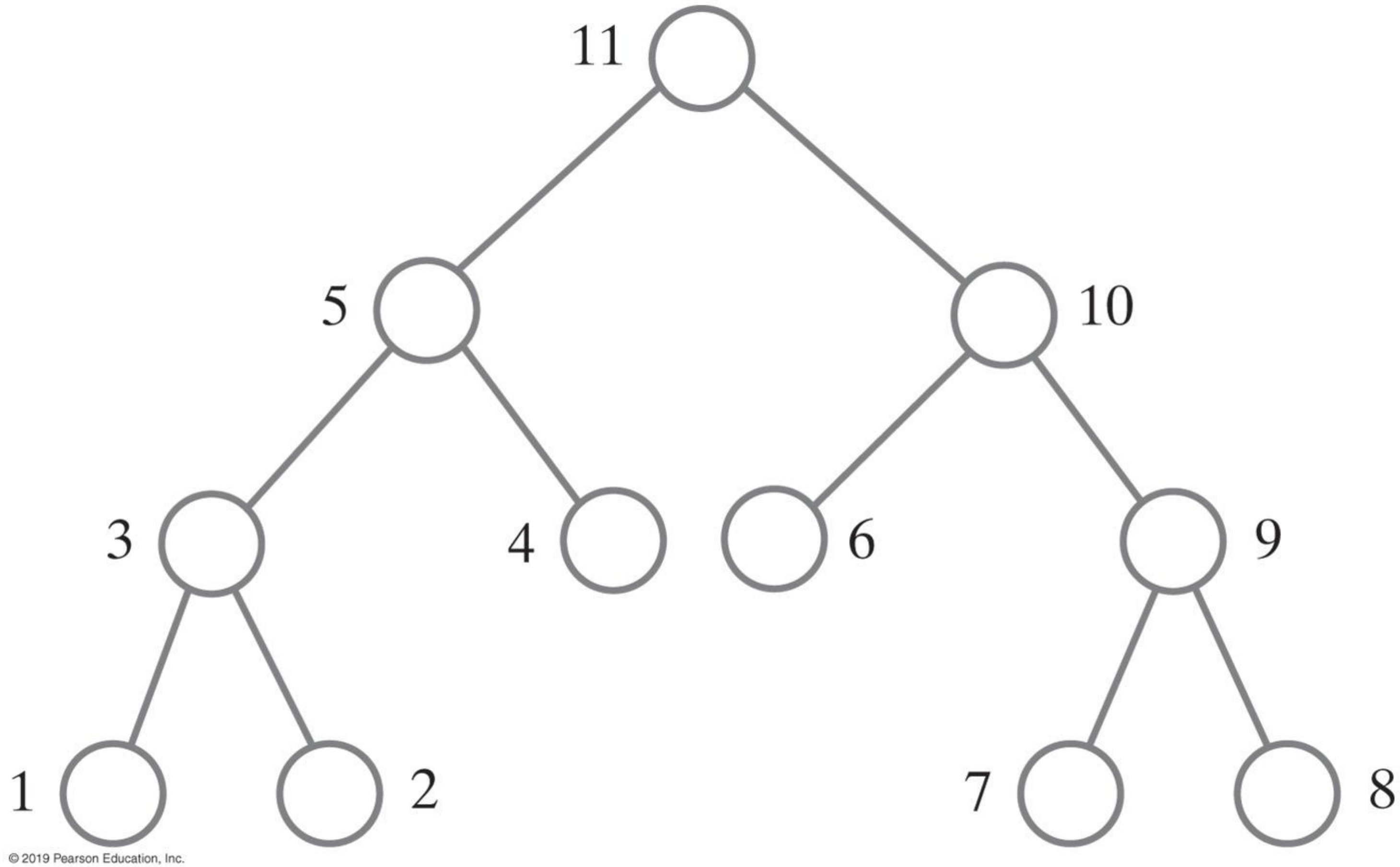


FIGURE 24-11 The visitation order of a postorder traversal

Traversals of a Binary Tree

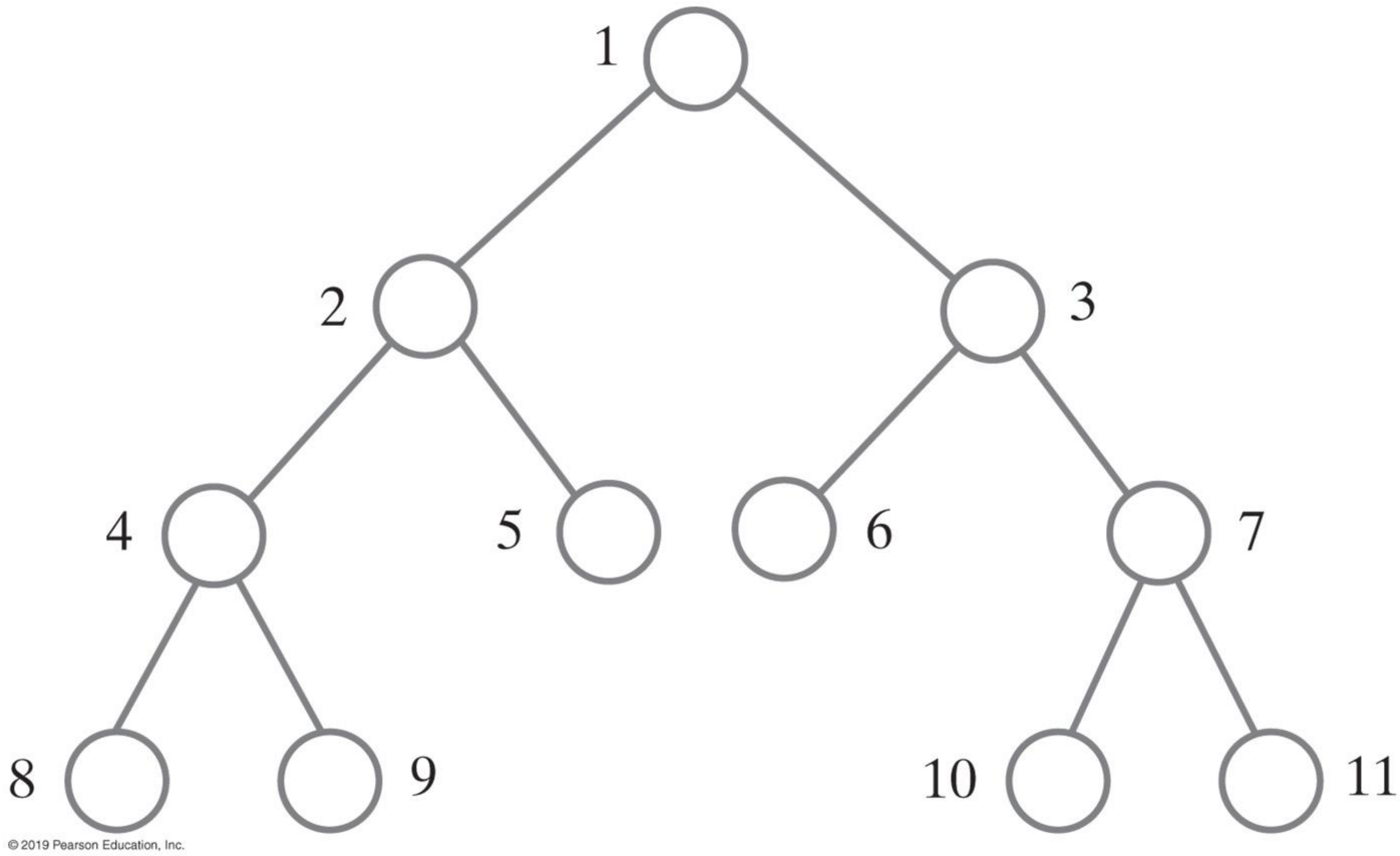
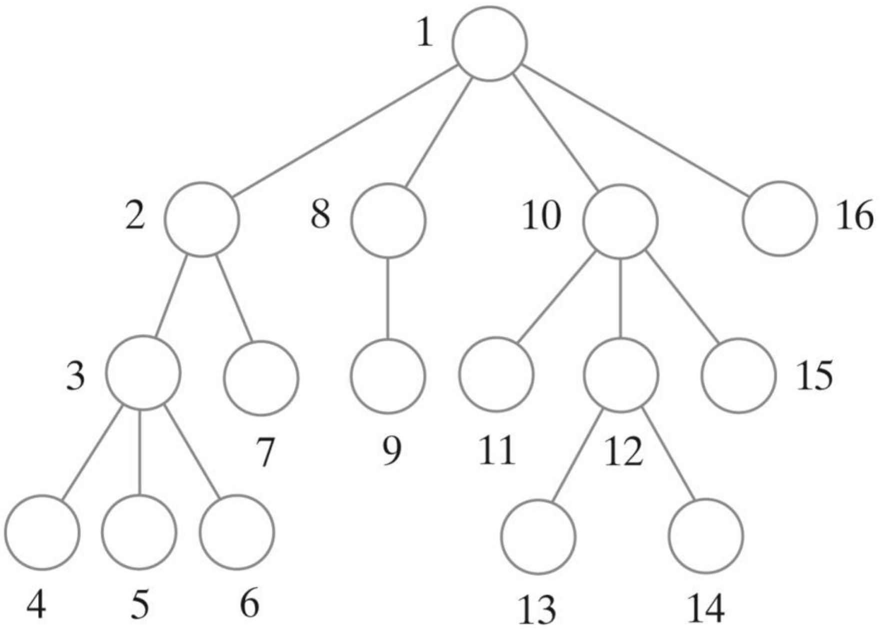


FIGURE 24-12 The visitation order of a level-order traversal

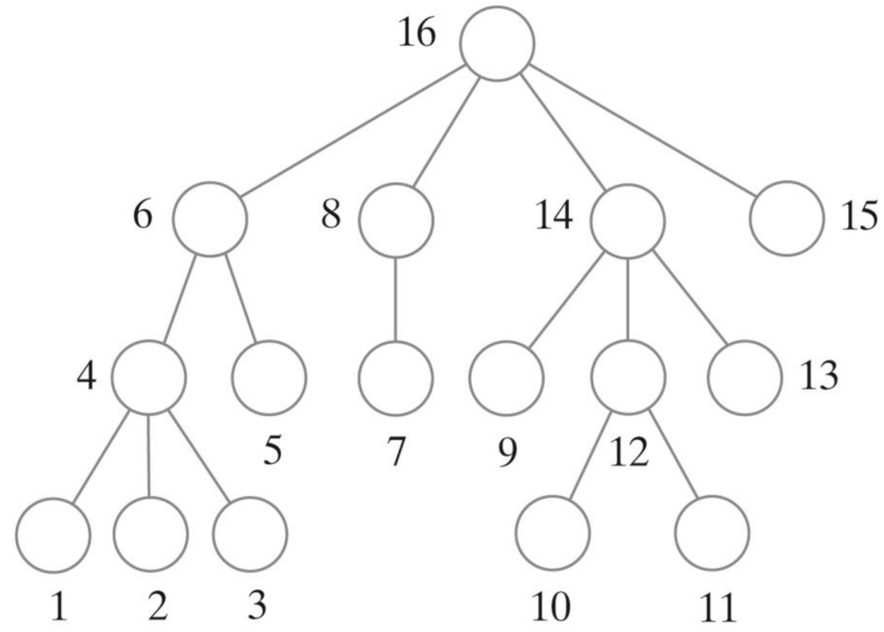
Traversals of a General Tree

- Types of traversals for general tree
  - Level order
  - Preorder
  - Postorder
- Not suited for general tree traversal
  - Inorder

Traversals of a General Tree



(a) Preorder traversal



(b) Postorder traversal

FIGURE 24-13 The visitation order of two traversals of a general tree

Interfaces for All Trees

```
package TreePackage;
/** An interface of basic methods for the ADT tree.  */
public interface TreeInterface<T>
{
  public T getRootData();
  public int getHeight();
  public int getNumberOfNodes();
  public boolean isEmpty();
  public void clear();
} // end TreeInterface
```

LISTING 24-1 An interface of methods common to all trees

Traversals

```
package TreePackage;
import java.util.Iterator;
/** An interface of iterators for the ADT tree. */
public interface TreeIteratorInterface<T>
{
   public Iterator<T> getPreorderIterator();
   public Iterator<T> getPostorderIterator();
   public Iterator<T> getInorderIterator();
   public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

LISTING 24-2 An interface of traversal methods for a tree

Interface for Binary Trees

```
package TreePackage;
/*   An interface for the ADT binary tree. */
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                      TreeIteratorInterface<T>
{
   /** Sets the data in the root of this binary tree.
       @param rootData  The object that is the data for the tree's root.
   */
   public void setRootData(T rootData);

   /** Sets this binary tree to a new binary tree.
      @param rootData   The object that is the data for the new tree's root.
      @param leftTree   The left subtree of the new tree.
      @param rightTree  The right subtree of the new tree. */
   public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                               BinaryTreeInterface<T> rightTree);
} // end BinaryTreeInterface
```

LISTING 24-3 An interface for a binary tree

Building a Binary Tree

```
BinaryTreeInterface<String> dTree = new BinaryTree<>();
dTree.setTree("D", null, null);

BinaryTreeInterface<String> fTree = new BinaryTree<>();
fTree.setTree("F", null, null);

BinaryTreeInterface<String> gTree = new BinaryTree<>();
gTree.setTree("G", null, null);

BinaryTreeInterface<String> hTree = new BinaryTree<>();
hTree.setTree("H", null, null);

BinaryTreeInterface<String> emptyTree = new BinaryTree<>();

// Form larger subtrees
BinaryTreeInterface<String> eTree = new BinaryTree<>();
eTree.setTree("E", fTree, gTree); // Subtree rooted at E

BinaryTreeInterface<String> bTree = new BinaryTree<>();
bTree.setTree("B", dTree, eTree); // Subtree rooted at B

BinaryTreeInterface<String> cTree = new BinaryTree<>();
cTree.setTree("C", emptyTree, hTree); // Subtree rooted at C

BinaryTreeInterface<String> aTree = new BinaryTree<>();
aTree.setTree("A", bTree, cTree); // Desired tree rooted at A
```

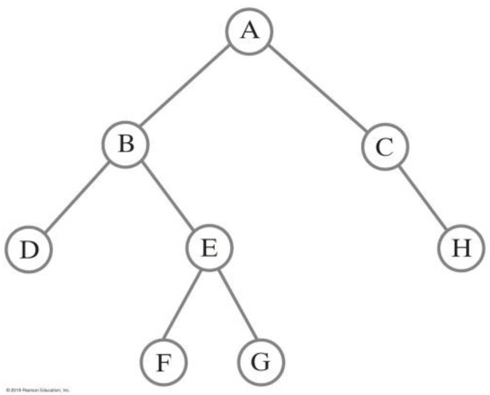


FIGURE 24-14 A binary tree whose nodes contain one-letter strings

Java statements that build a tree

Building a Binary Tree

```
// Display root, height, number of nodes
System.out.println("Root of tree contains " + aTree.getRootData());
System.out.println("Height of tree is " + aTree.getHeight());
System.out.println("Tree has " + aTree.getNumberOfNodes() + " nodes");

// Display nodes in preorder
System.out.println("A preorder traversal visits nodes in this order:");
Iterator<String> preorder = aTree.getPreorderIterator();
while (preorder.hasNext())
   System.out.print(preorder.next() + " ");
System.out.println();
```

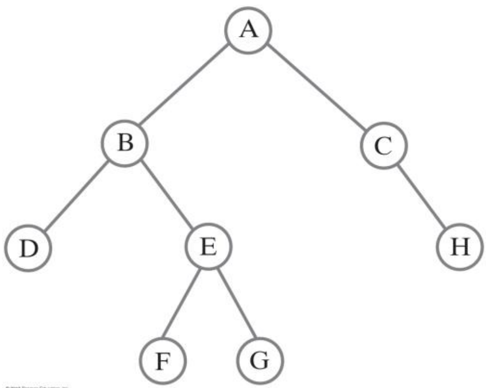


FIGURE 24-14 A binary tree whose nodes contain one-letter strings

Java statements that build a tree and then display some of its characteristics:

Expression Trees

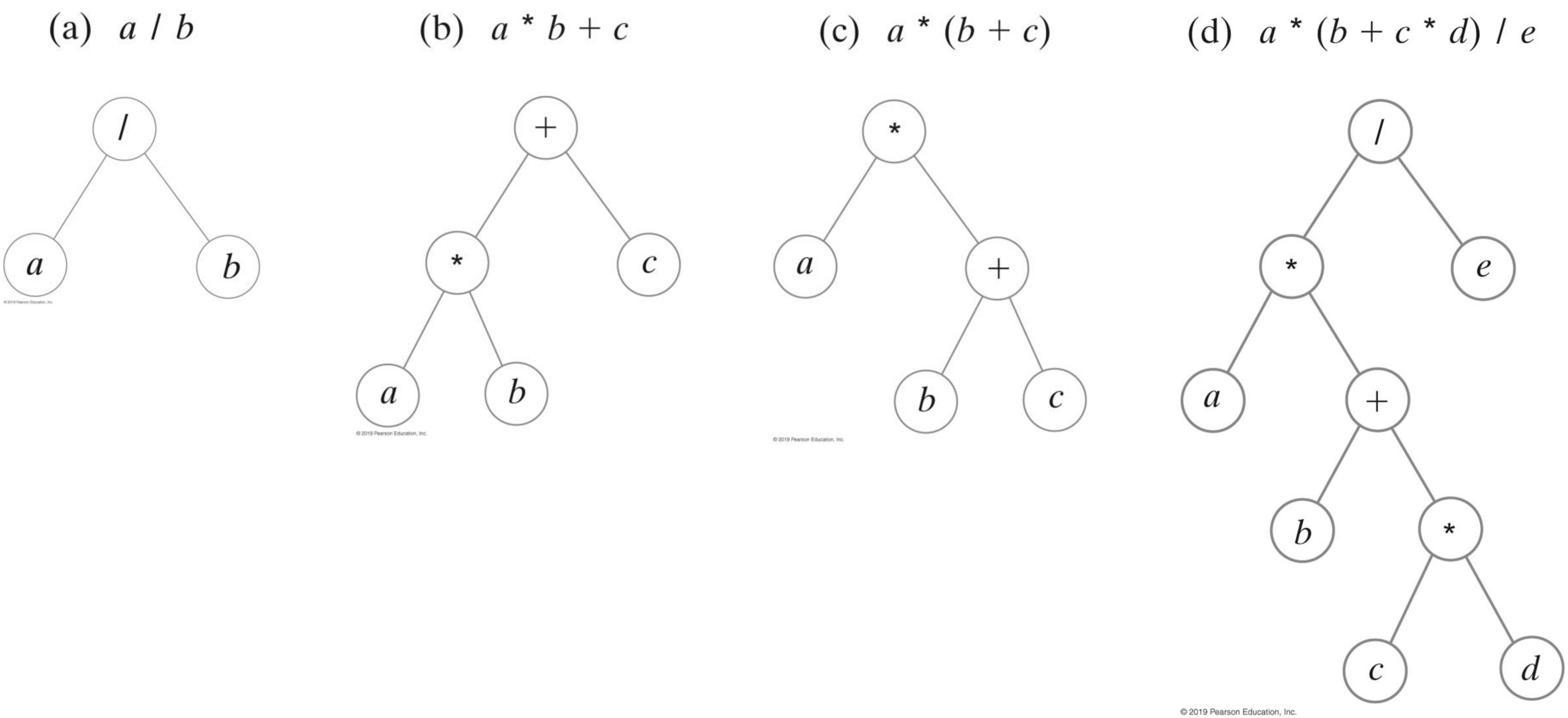


FIGURE 24-15 Expression trees for four algebraic expressions

Pearson

Expression Trees

```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and its operands firstOperand
                    and secondOperand
}
```

Algorithm for postorder traversal of an expression tree.

Binary Search Tree

- For each node in a binary search tree
  - Node’s data is greater than all data in node’s left subtree
  - Node’s data is less than all data in node’s right subtree
- Every node in a binary search tree is the root of a binary search tree

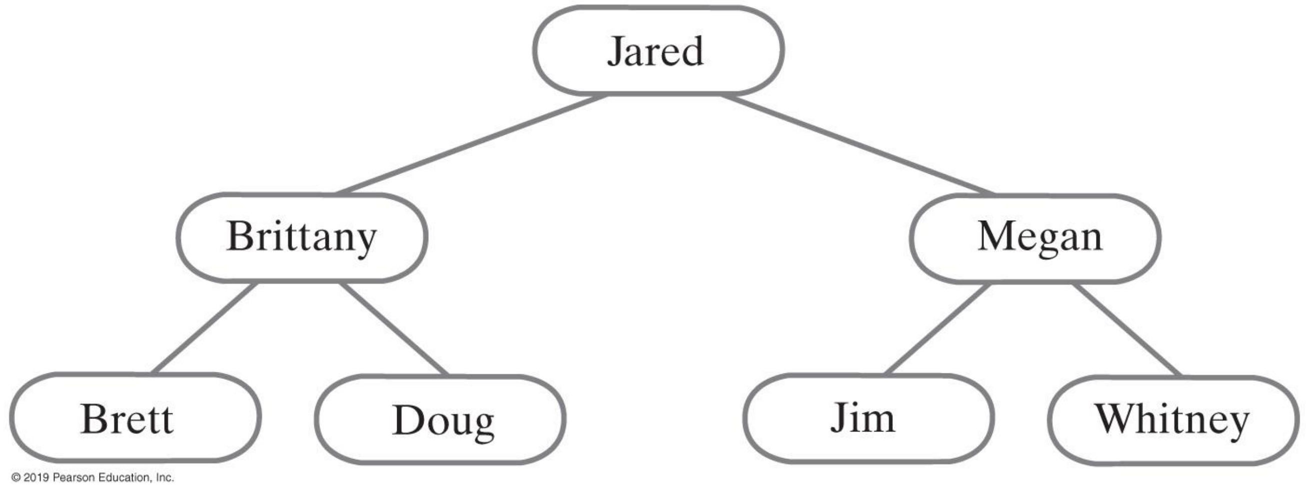


FIGURE 24-19 A binary search tree of names

Binary Search Tree

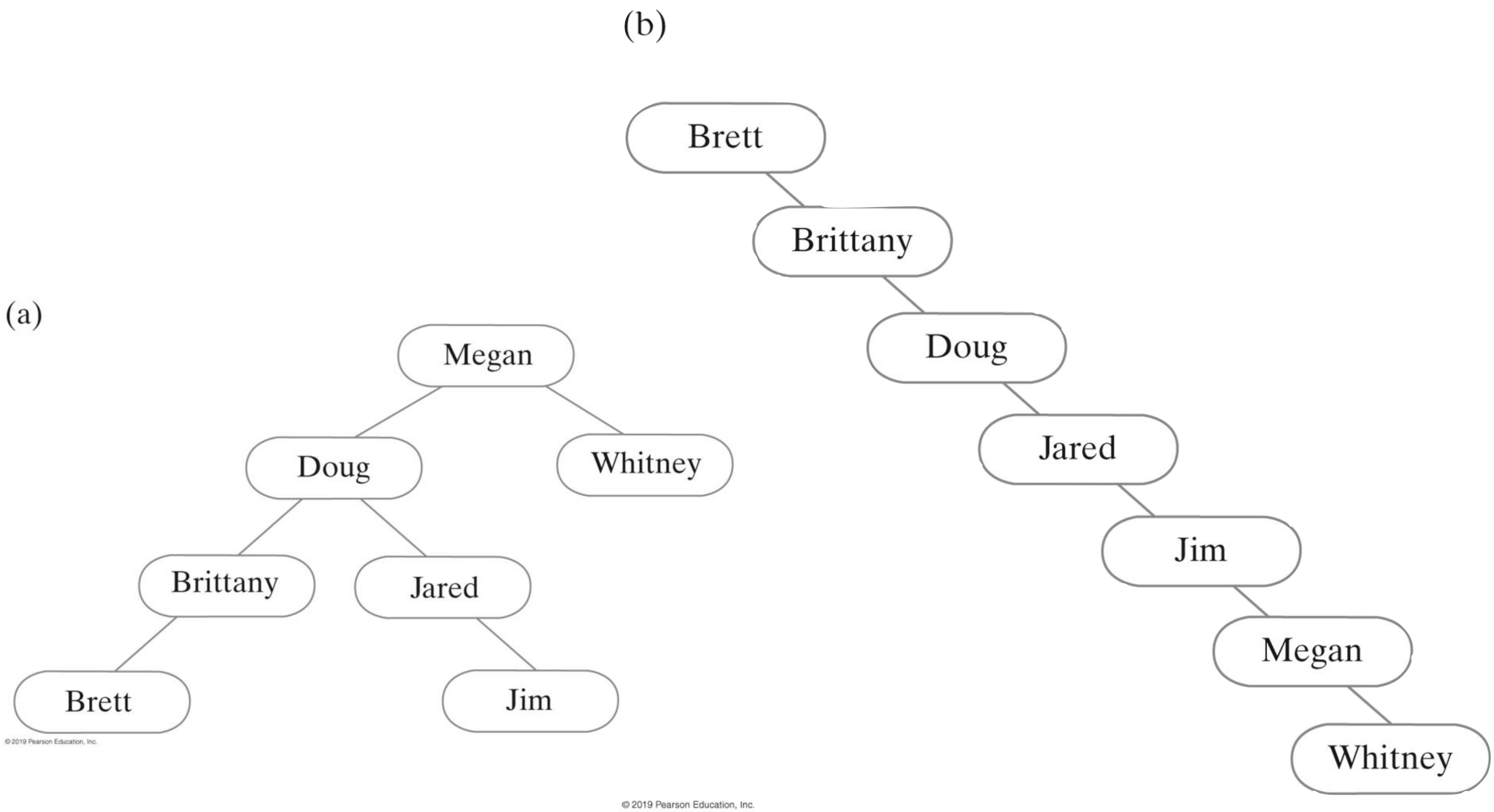


FIGURE 24-20 Two binary search trees containing the same data as the tree in Figure 24-19

Binary Search Tree

```
Algorithm bstSearch(binarySearchTree, desiredObject)
// Searches a binary search tree for a given object.
//Returns true if the object is found.
if (binarySearchTree is empty)
  return false
else if (desiredObject == object in the root of binarySearchTree)
  return true
else if (desiredObject < object in the root of binarySearchTree)
  return bstSearch(left subtree of binarySearchTree, desiredObject)
else
  return bstSearch(right subtree of binarySearchTree, desiredObject)
```

Pseudocode for recursive search algorithm

Pearson

Binary Search Tree

- Efficiency of a search
  - Searching a binary search tree of height h is $O(h)$
- To make searching a binary search tree efficient:
  - Tree must be as short as possible.

Heaps

- Complete binary tree whose nodes contain `Comparable` objects and are organized as follows:
  - Each node contains an object no smaller/larger than objects in its descendants
  - ***Maxheap***: object in node greater than or equal to its descendant objects
  - ***Minheap***: object in node less than or equal to its descendant objects

Heaps

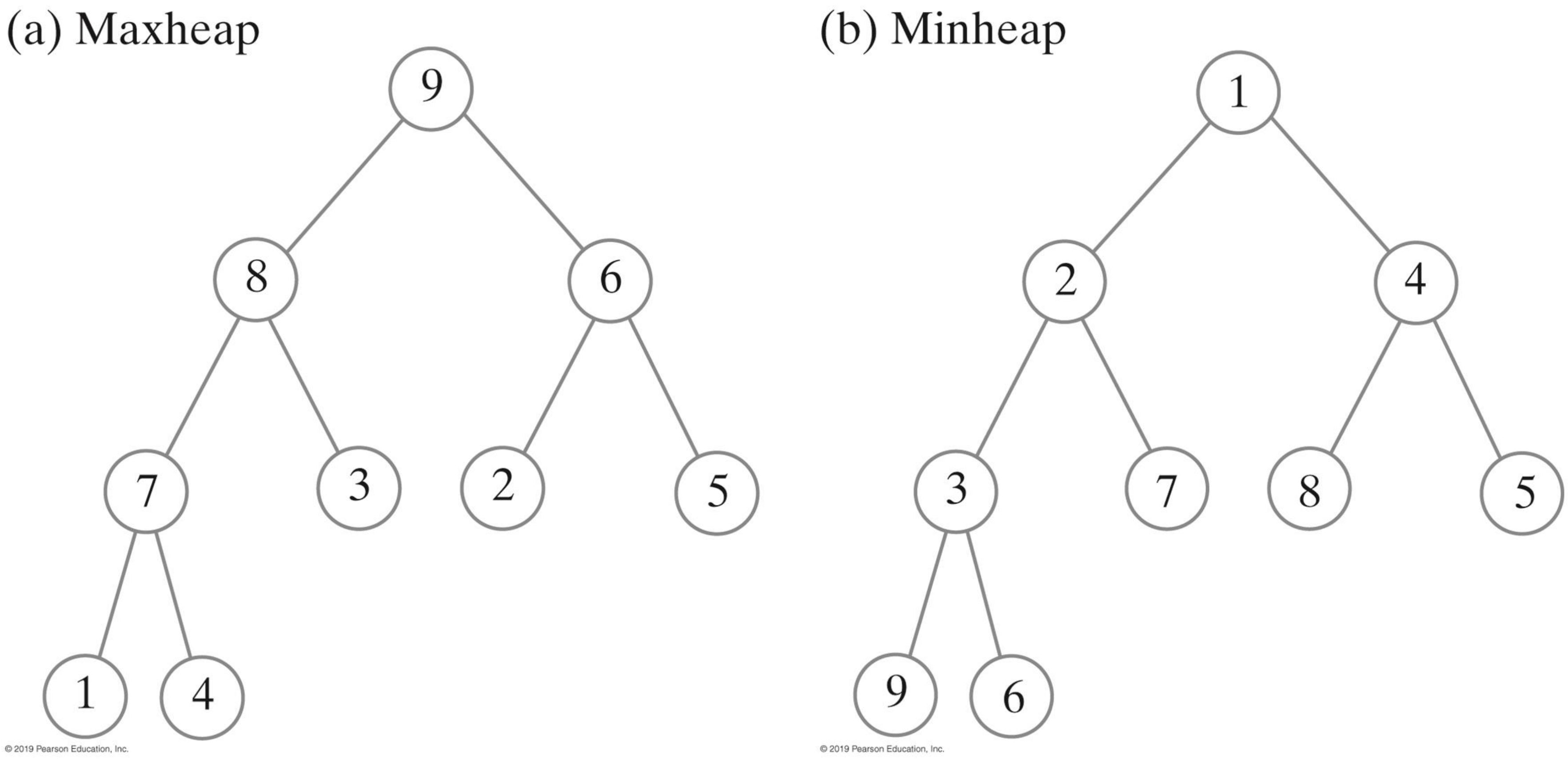


FIGURE 24-21 Two heaps that contain the same values

Heaps

```java
/**  An interface for the ADT maxheap. */
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
  /** Adds a new entry to this heap.
     @param newEntry  An object to be added. */
  public void add(T newEntry);

  /** Removes and returns the largest item in this heap.
     @return  Either the largest object in the heap or,
             if the heap is empty before the operation, null. */
  public T removeMax();

  /** Retrieves the largest item in this heap.
     @return  Either the largest object in the heap or,
             if the heap is empty, null. */
  public T getMax();

  /** Detects whether this heap is empty.
     @return  True if the heap is empty, or false otherwise. */
  public boolean isEmpty();

  /** Gets the size of this heap.
     @return  The number of entries currently in the heap. */
  public int getSize();

  /** Removes all entries from this heap. */
  public void clear();
} // end MaxHeapInterface
```

LISTING 24-6 An interface for a maxheap

End

Chapter 24