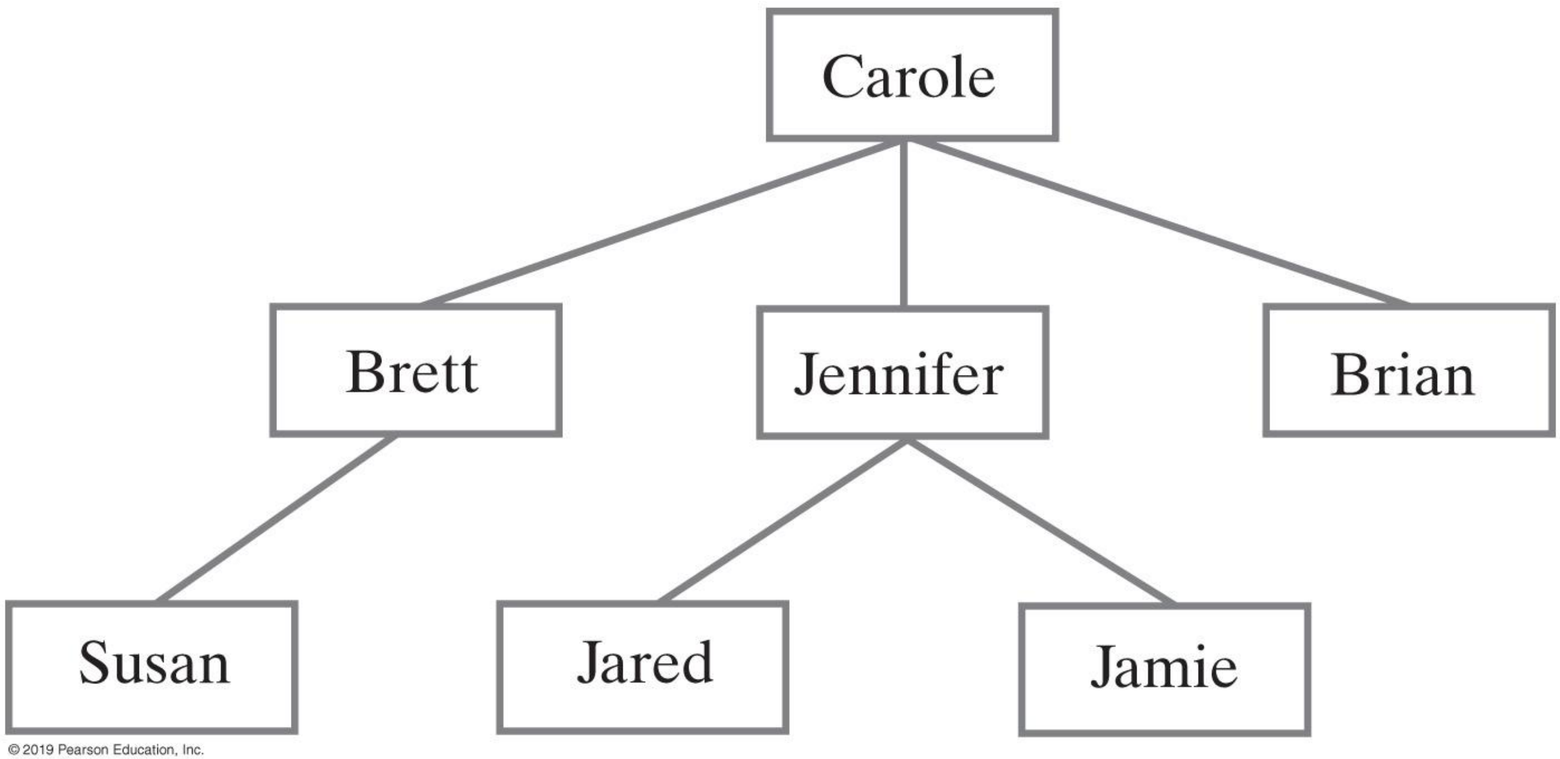
Class 10 - Trees

CSIS 3475 Data Structures and Algorithms

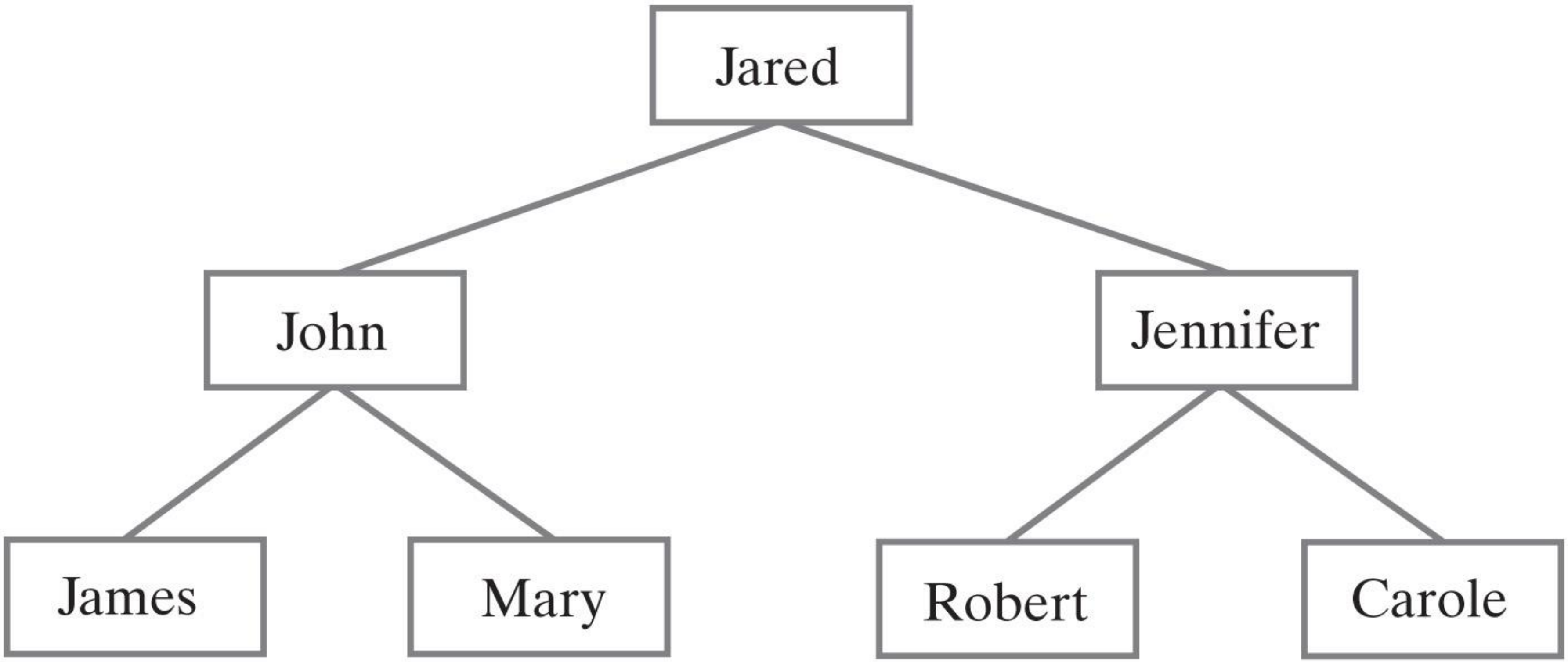
Hierarchical Organizations

- FIGURE 24-1 Carole's children and grandchildren



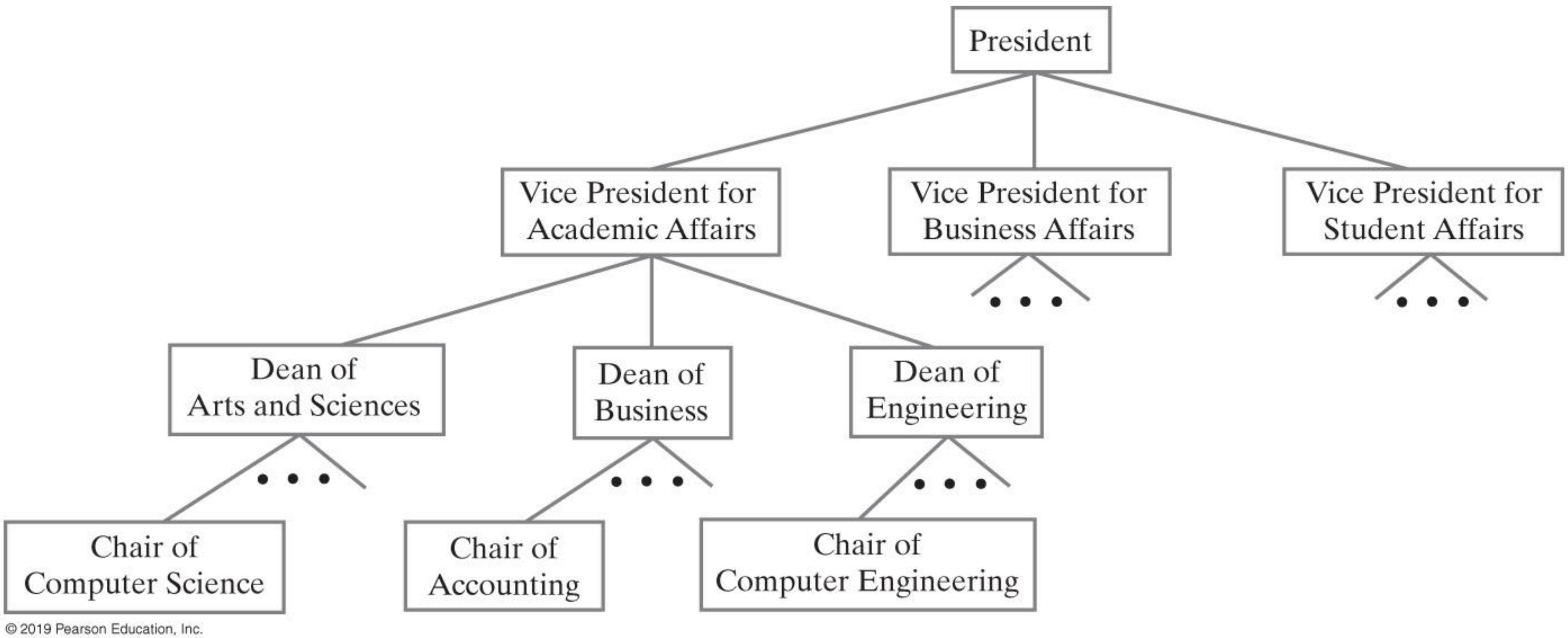
Hierarchical Organizations

- FIGURE 24-2 Jared's parents and grandparents

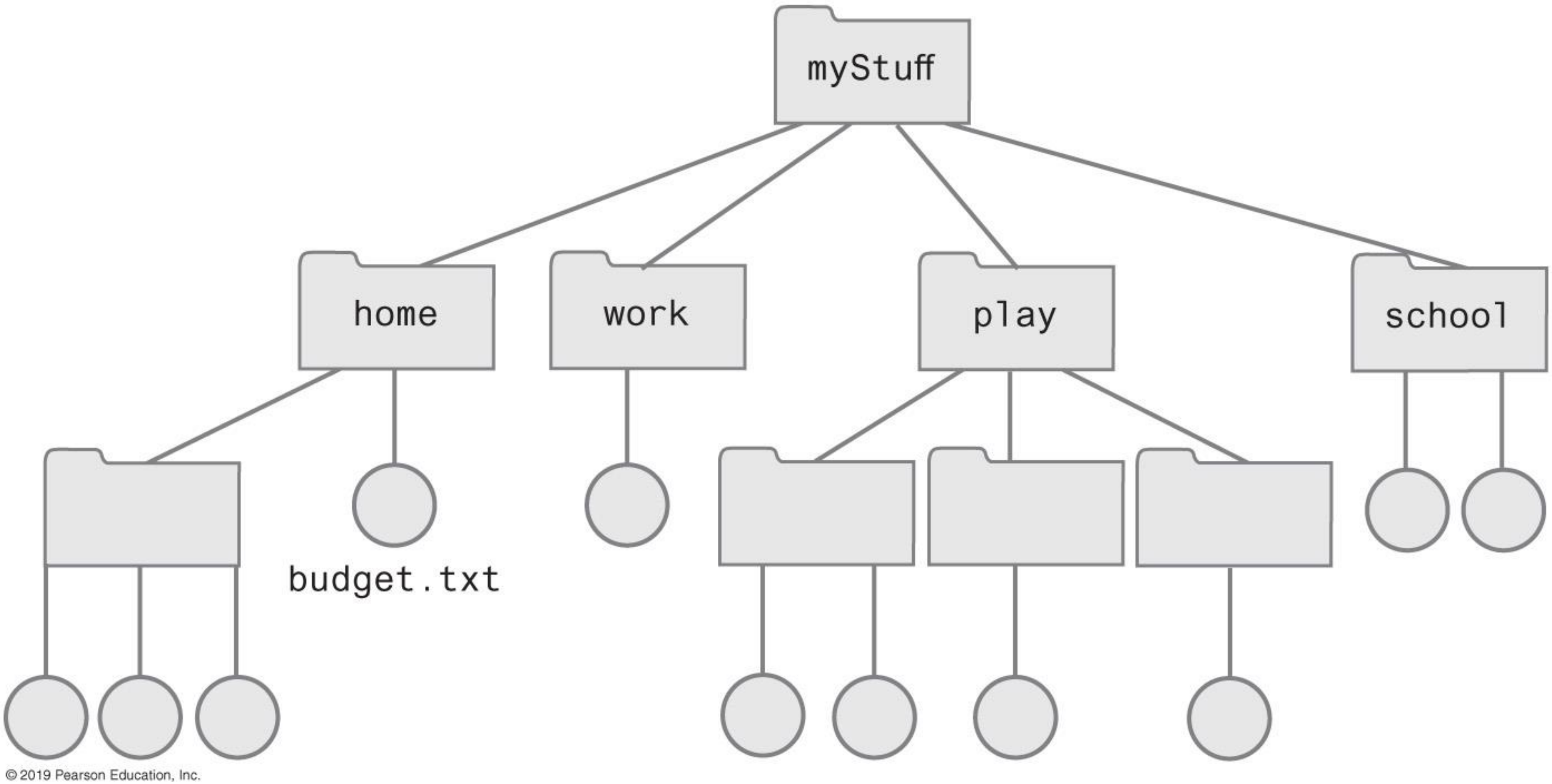


Hierarchical Organizations

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

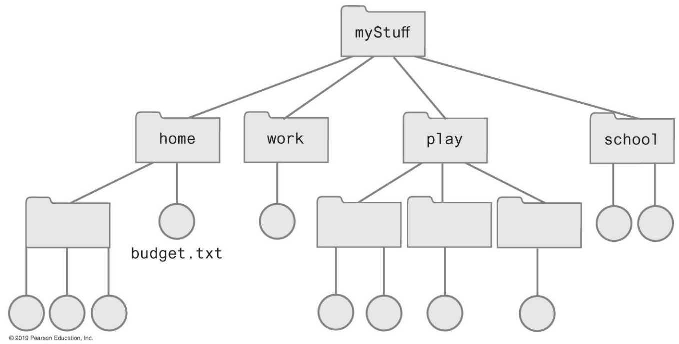


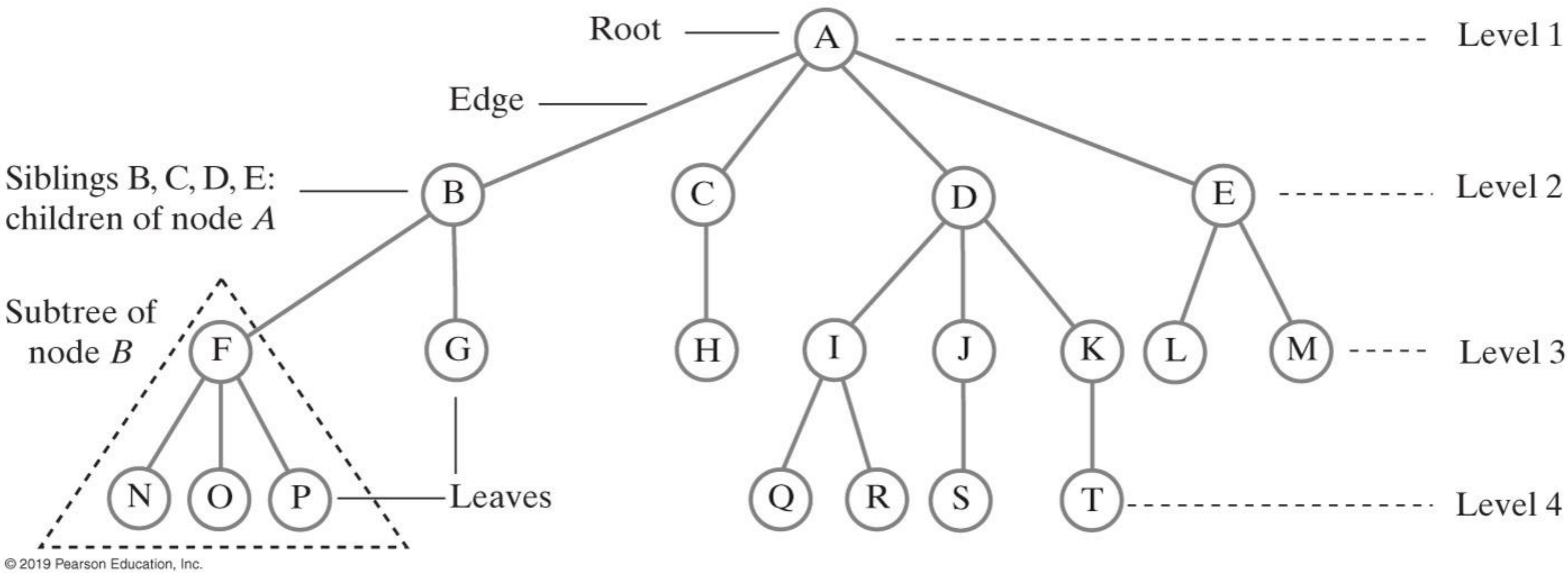
Hierarchical Organizations



Tree Terminology

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

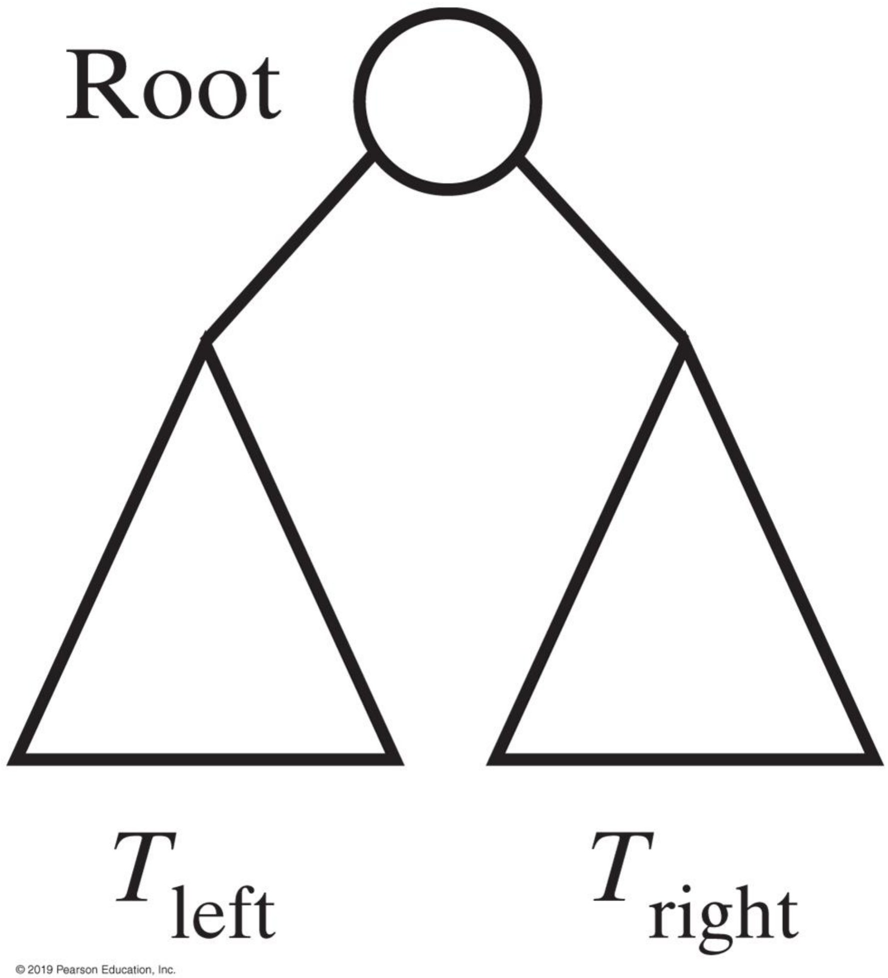




Tree Terminology

- 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

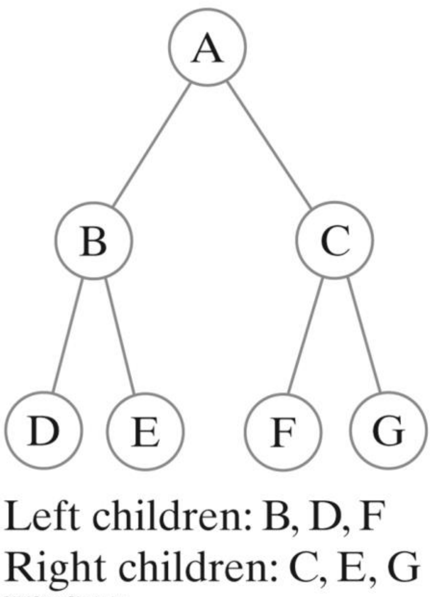


Tree types

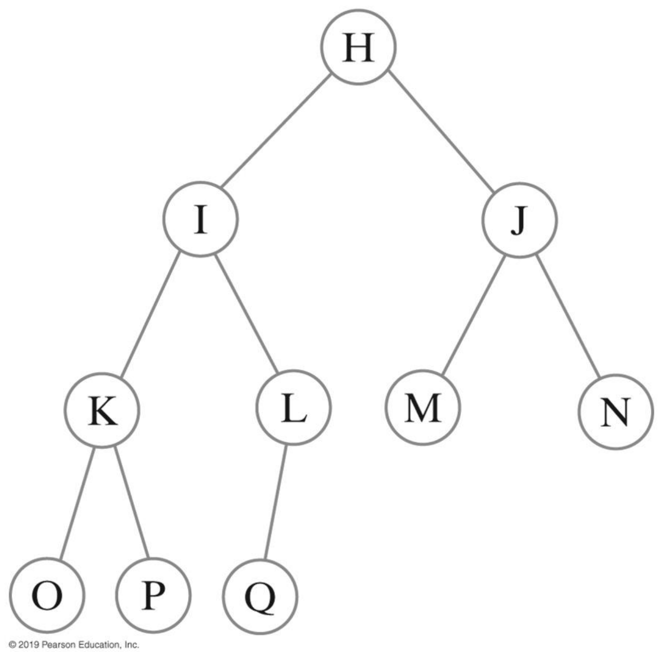
- Tree is said to be **full** when a a binary tree of height h has all of its leaves at level h and every parent has exactly two children
- Tree is said to be **complete** when all levels but the last contain as many nodes as possible, and the nodes on the last level are filled in from left to tight. (Not full, but complete)
- When each node in a binary tree has two subtrees who's heights are exactly the same the tree is said to be **completely balanced**
- Completely balanced trees are full
- A tree is **height balanced** or simply **balanced** if the subtrees of a node differ by no more than one

Three binary trees

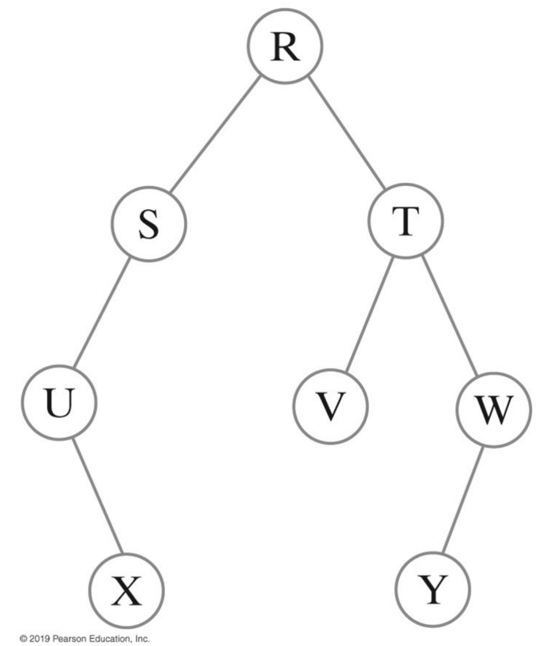
(a) Full tree



(b) Complete tree

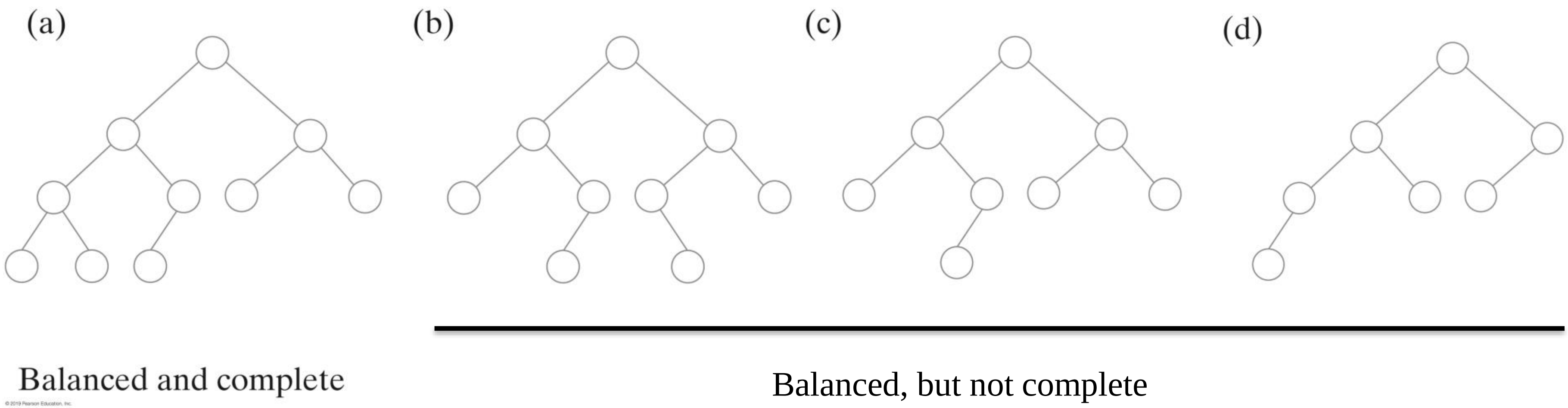


(c) Tree that is not full and not complete



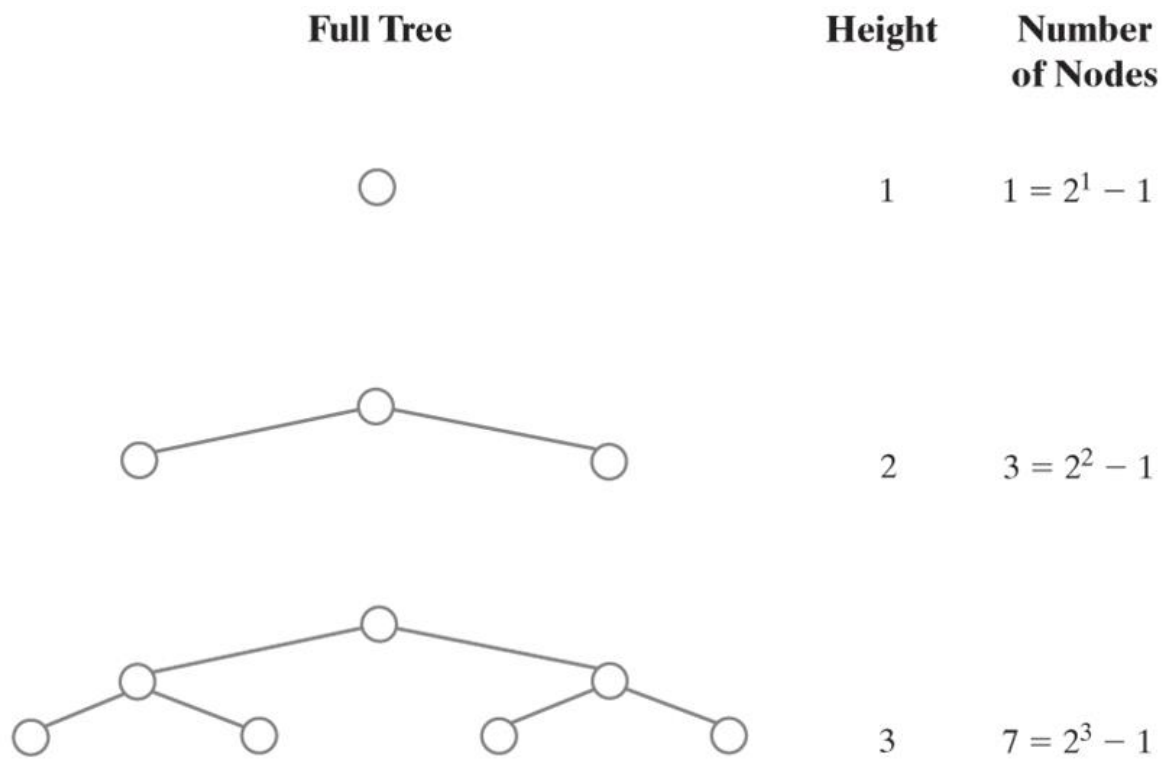
Binary Trees

- FIGURE 24-7 Some binary trees that are height balanced

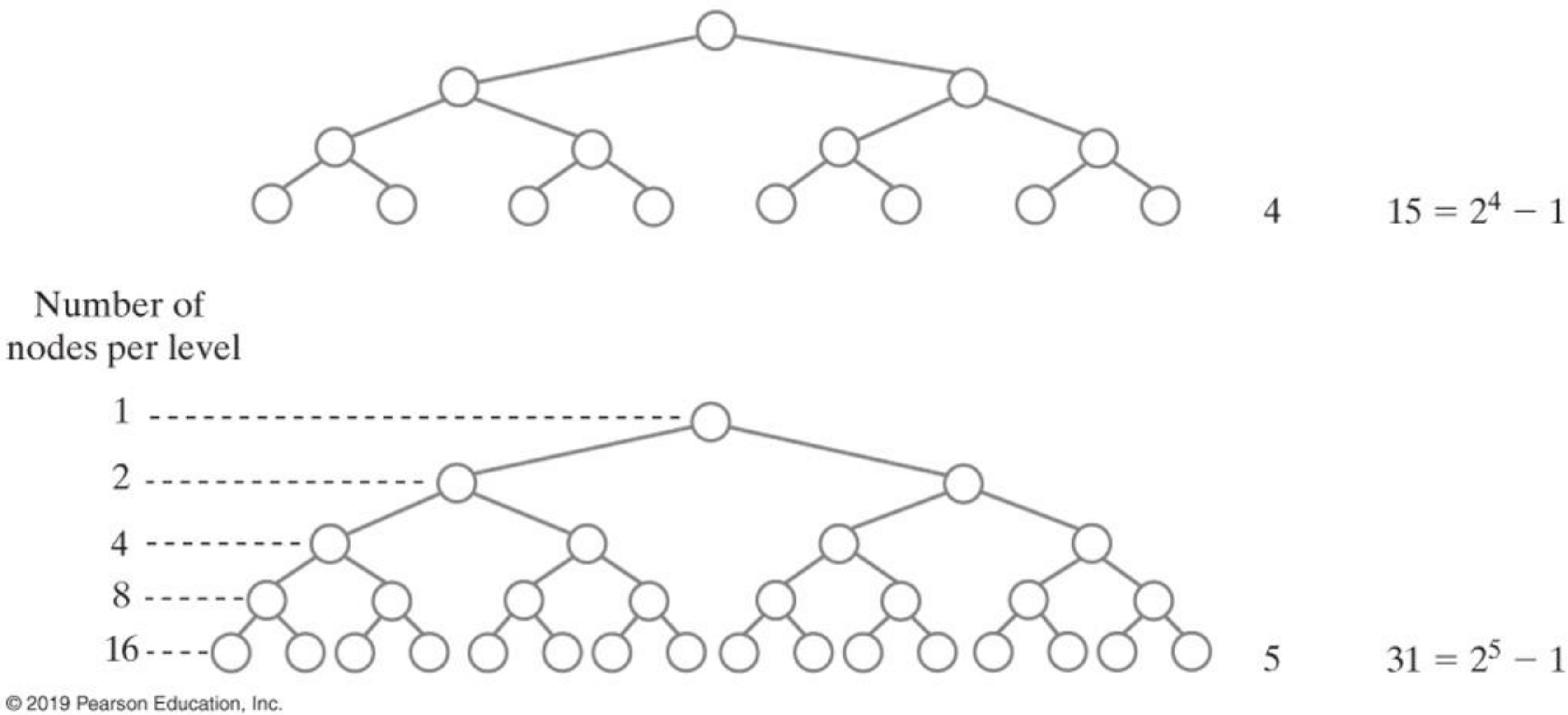


Binary Tree Height (Part 1)

- FIGURE 24-8 The number of nodes in a full binary tree as a function of the tree's height.
- Number of Nodes = $2^N - 1$, where N is height.



Binary Tree Height (Part 2)



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

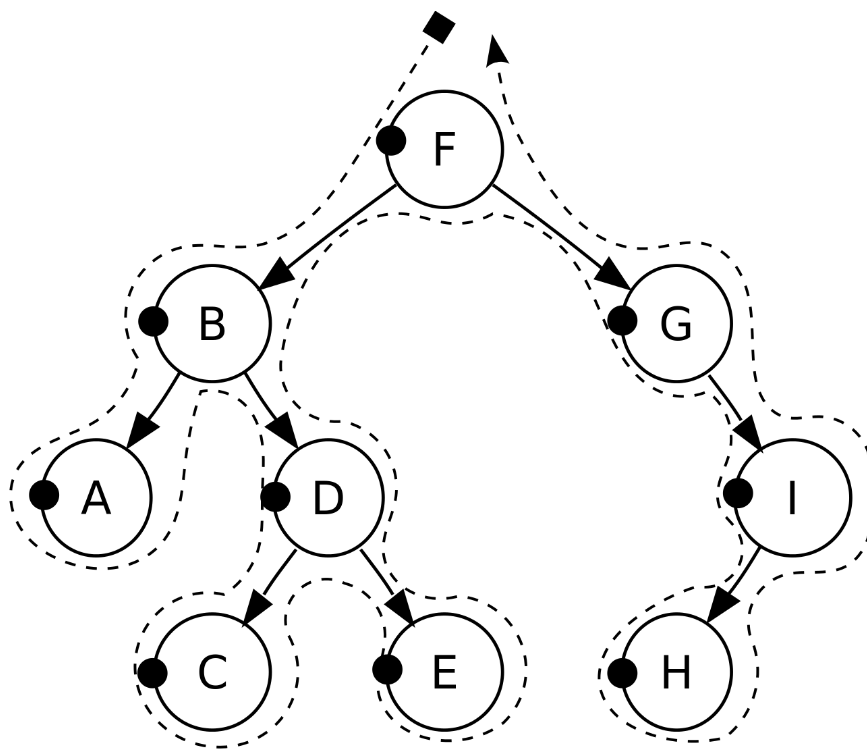
- We use recursion
- To visit all the nodes in a binary tree, we must
  - Visit the root
  - Visit all the nodes in the root's left subtree
  - Visit all the nodes in the root's right subtree

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

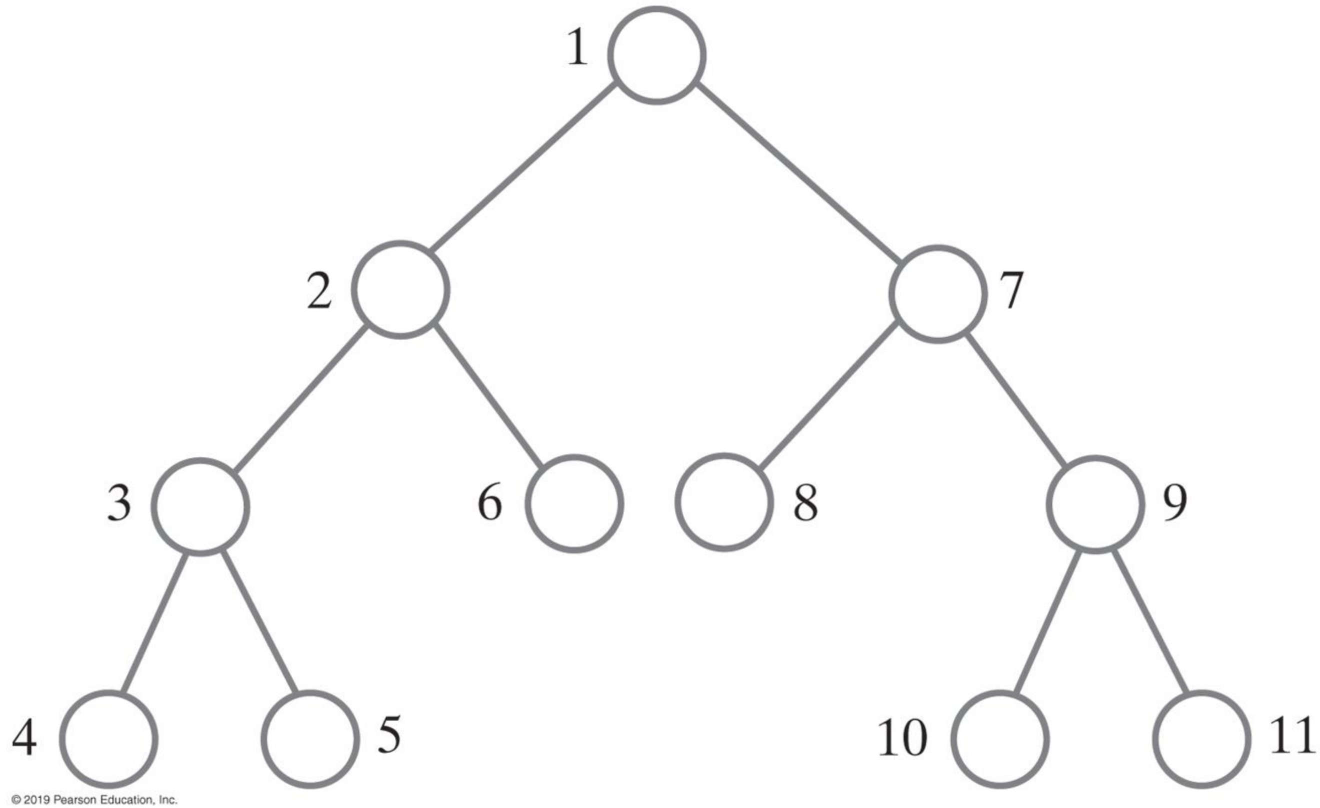
Preorder traversal

- Check if the current node is empty or null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.
- Follow the dots and dashed lines below: FBADCEGIH



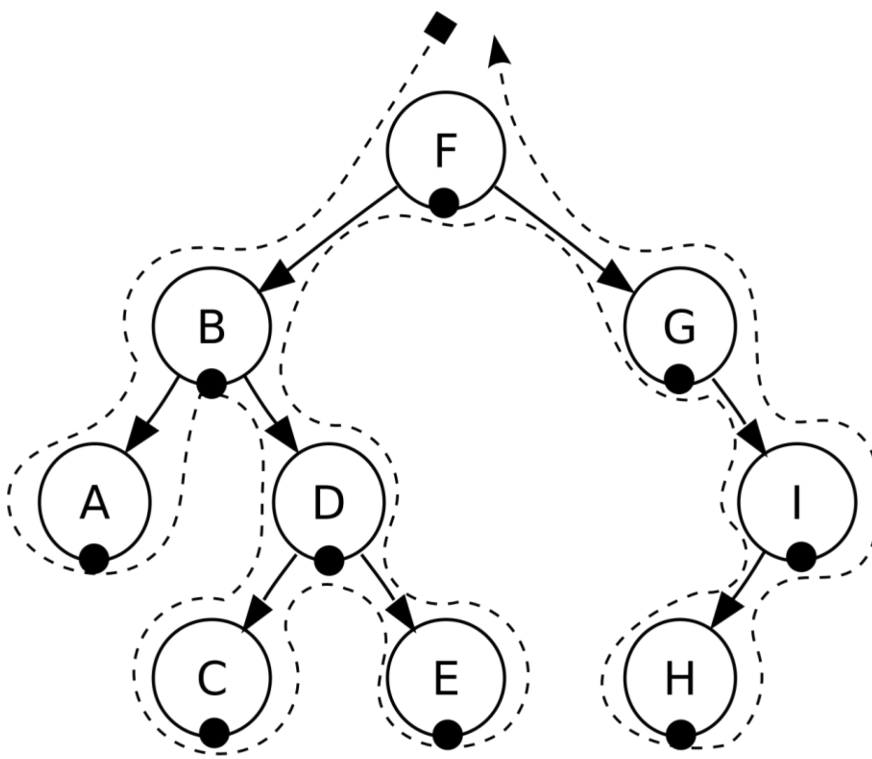
Source: Wikipedia
https://en.wikipedia.org/wiki/Tree_traversal

Preorder traversal



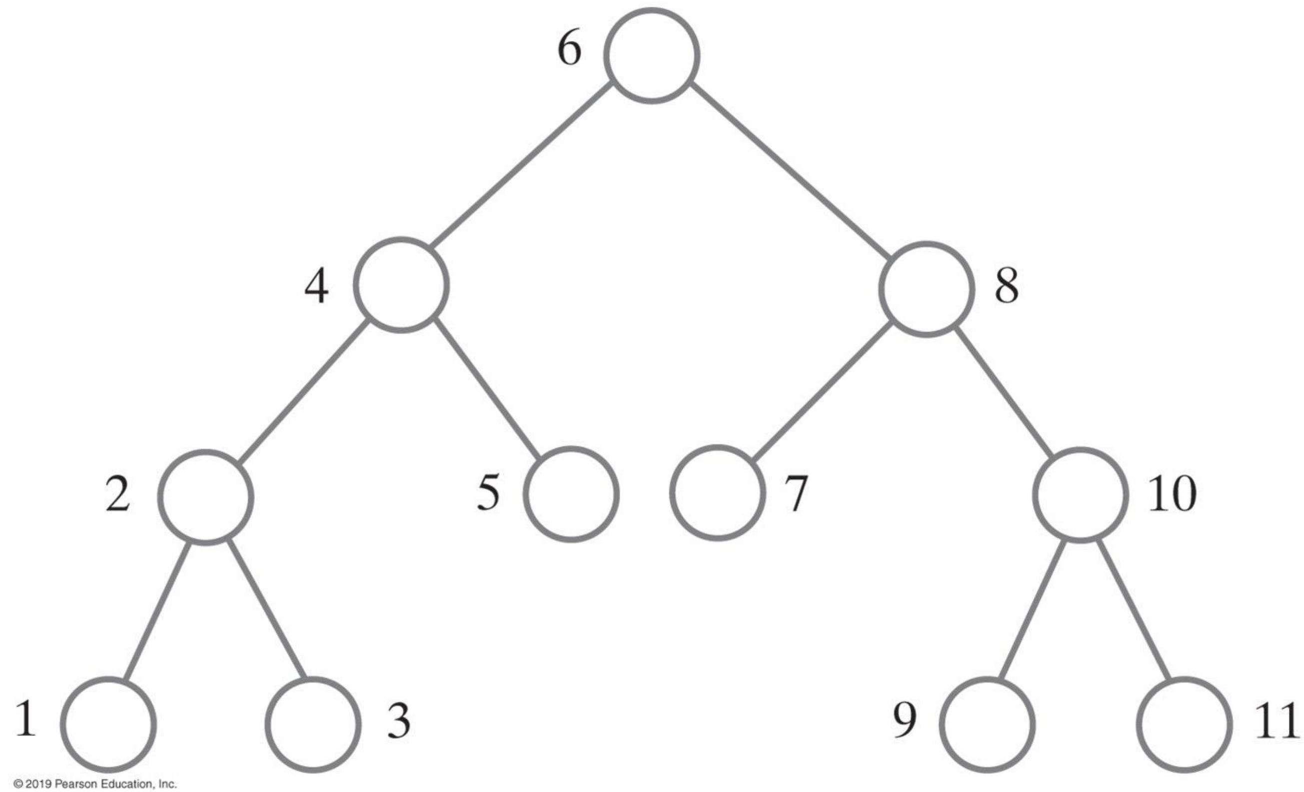
Inorder traversal

- Check if the current node is empty or null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.
- In a [binary search tree](), in-order traversal retrieves data in sorted order.
- Order: ABCDEFGHI



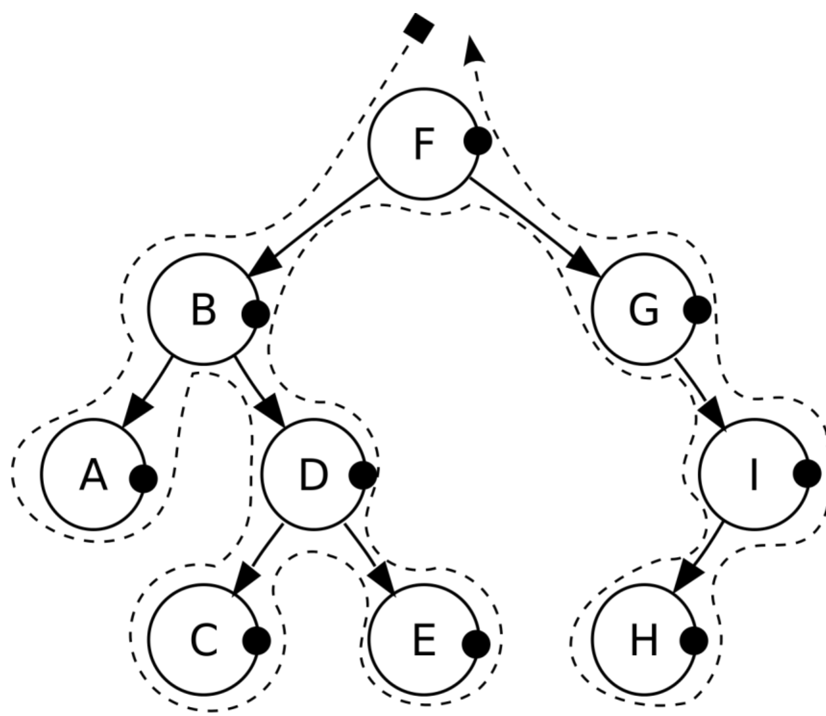
Source: Wikipedia
https://en.wikipedia.org/wiki/Tree_traversal

Inorder traversal



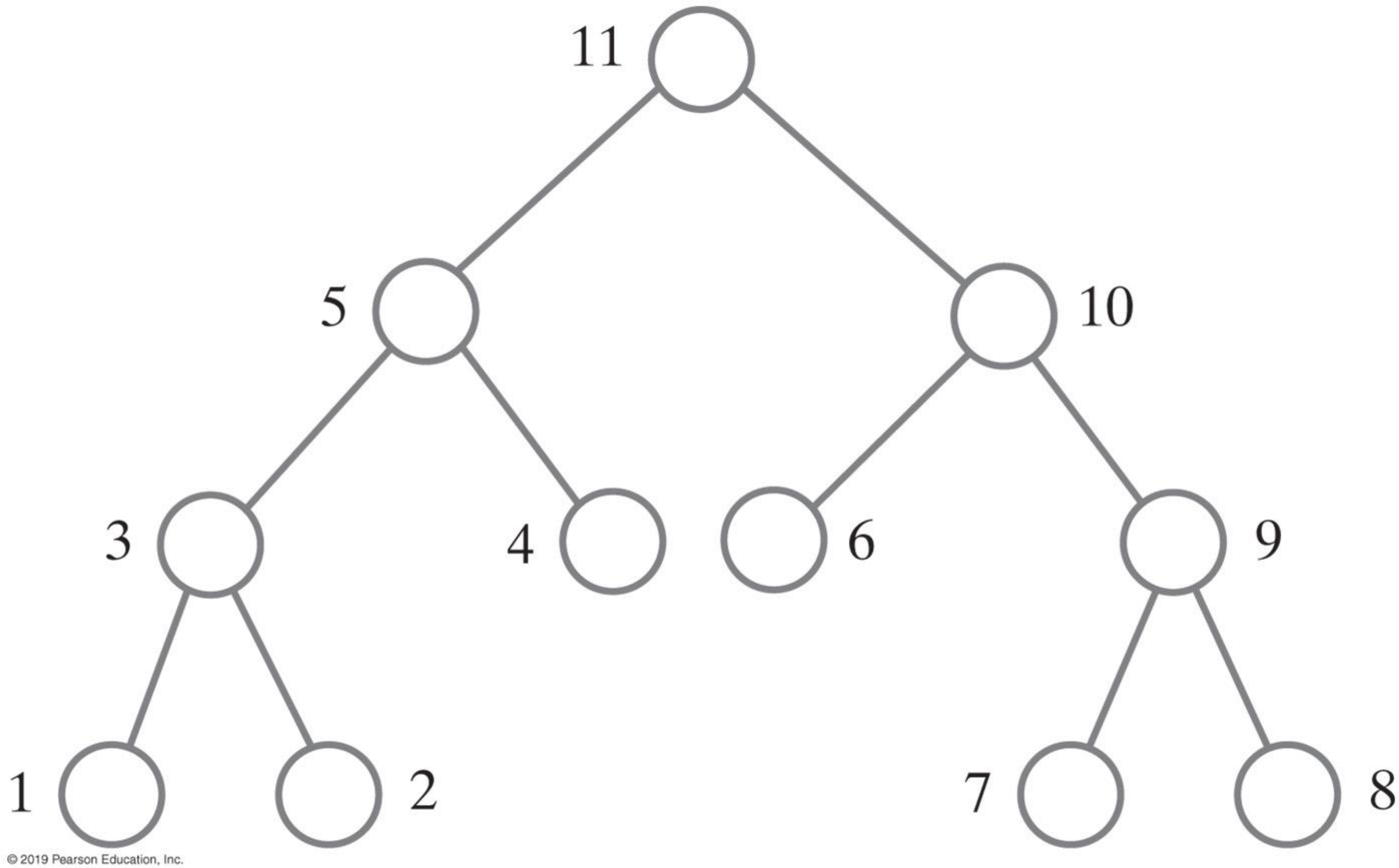
Postorder traversal

- Check if the current node is empty or null.
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of the root (or current node).
- Order: ACEDBHIGF



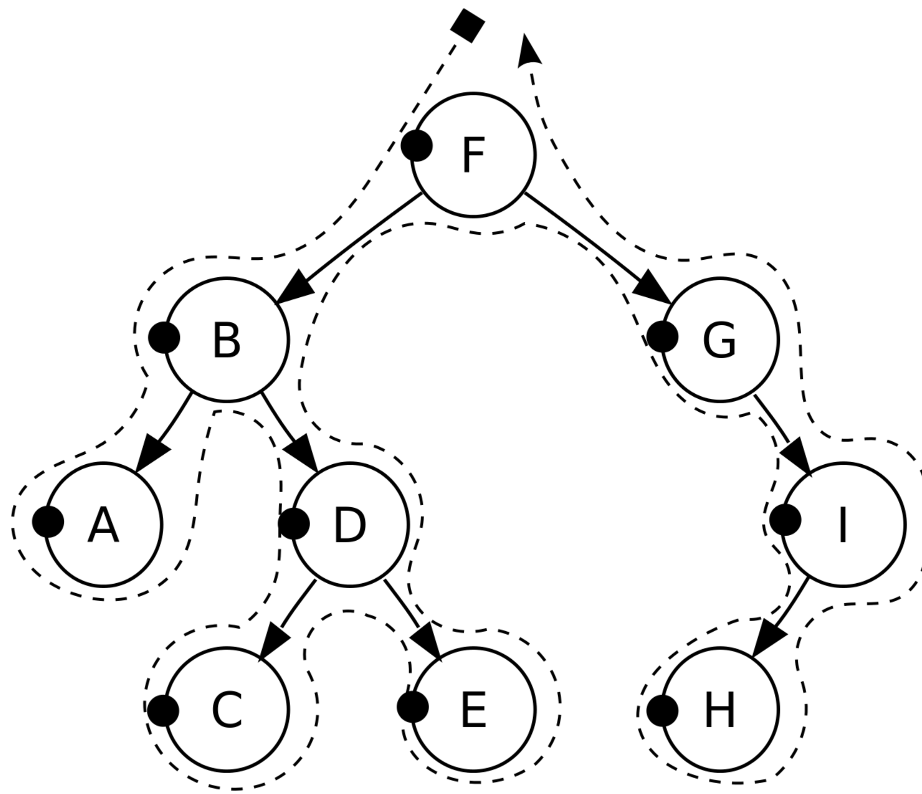
Source: Wikipedia
https://en.wikipedia.org/wiki/Tree_traversal

Postorder traversal



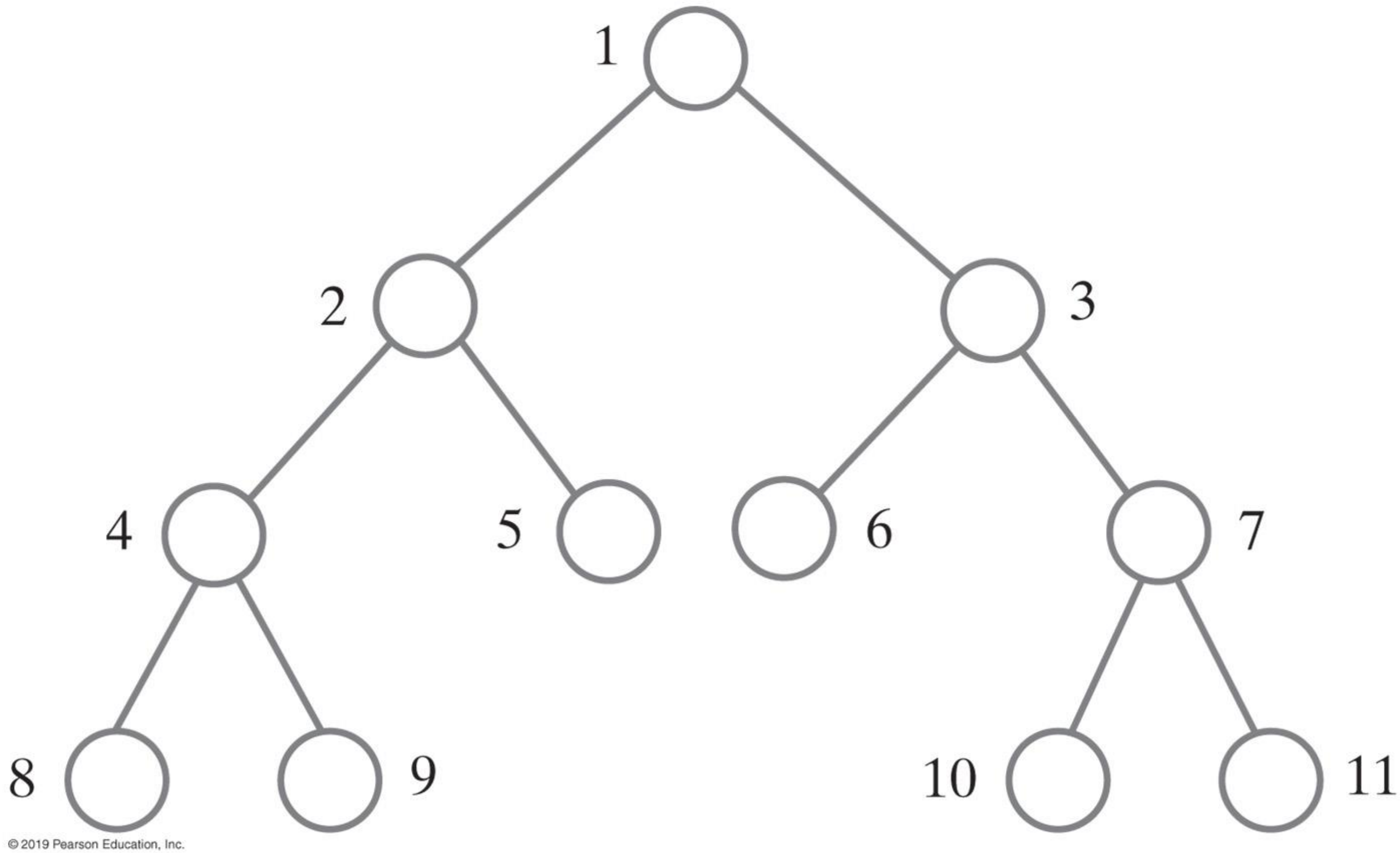
Preorder traversal

- Check if the current node is empty or null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.



Source: Wikipedia
https://en.wikipedia.org/wiki/Tree_traversal

Level-order traversal

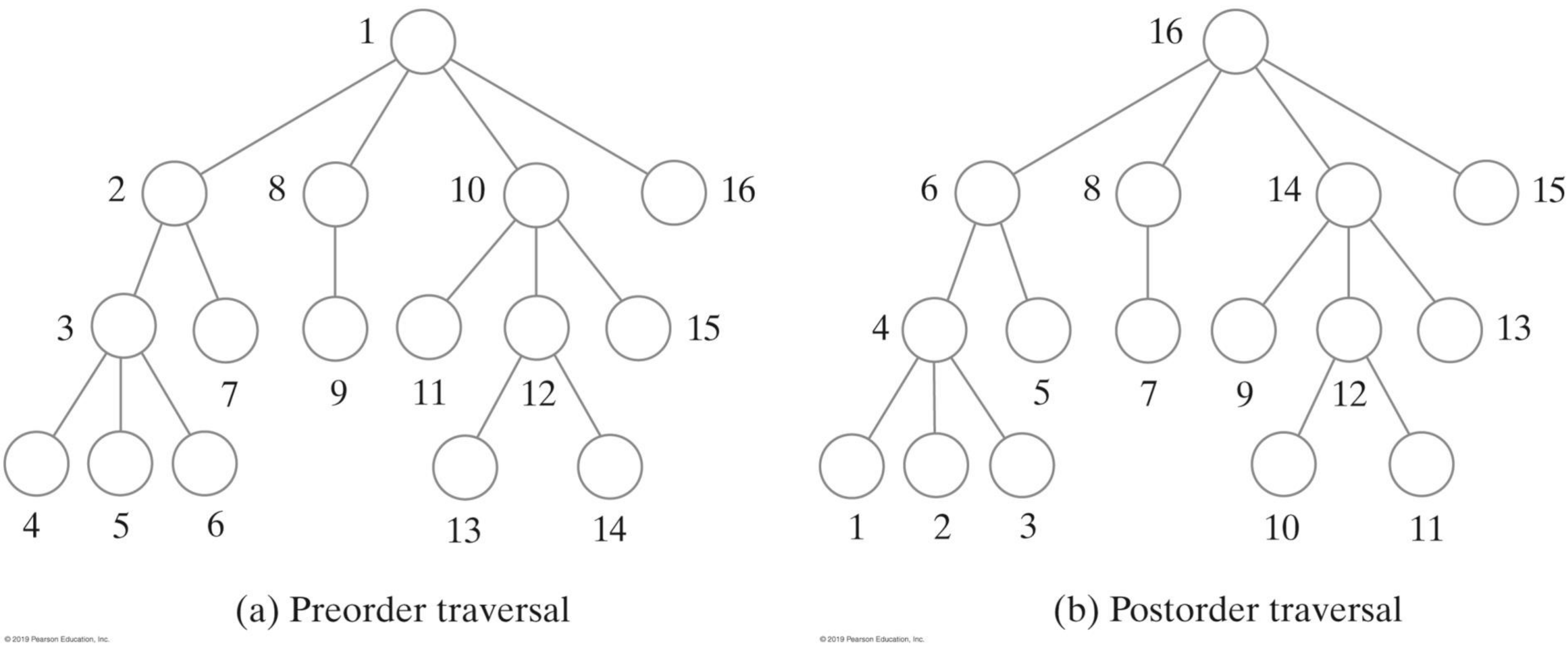


Traversals of a General Tree

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

Traversals of a General Tree

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



Basic Tree Interface

```java
public interface TreeInterface<T> {

		/**
		 * Gets the data at the root node of the tree
		 * @return data of type T in the node
		 */
		public T getRootData();

		/**
		 * Get the height of the tree
		 * @return tree height
		 */
		public int getHeight();

		/**
		 * Get the number of nodes in the tree from the root
		 * @return number of nodes
		 */
		public int getNumberOfNodes();

		/**
		 * Checks to see if the tree has any nodes
		 * @return true if there are no nodes in the tree
		 */
		public boolean isEmpty();

		/**
		 * Clears all nodes in the tree, leaving it empty
		 */
		public void clear();
}
```

Traversals

- Iterator interface specifies iterators for each type of tree traversal

```
public interface TreeIteratorInterface<T> {
            public Iterator<T> getPreorderIterator();

            public Iterator<T> getPostorderIterator();

            public Iterator<T> getInorderIterator();

            public Iterator<T> getLevelOrderIterator();
}
```

Binary Tree Interface

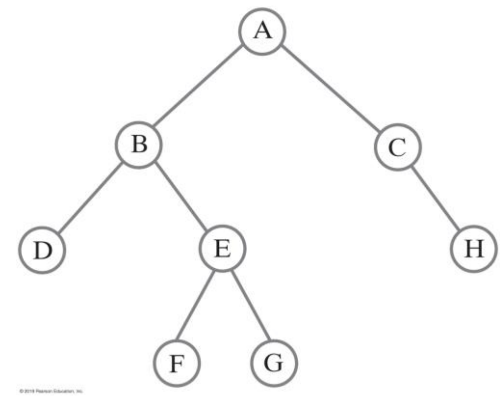
- In a binary tree, nodes only have two children (left and right).
- Interface adds setRootData() method and the ability to set the tree to a new root to TreeInterface

```
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);
}
```

Building a binary tree

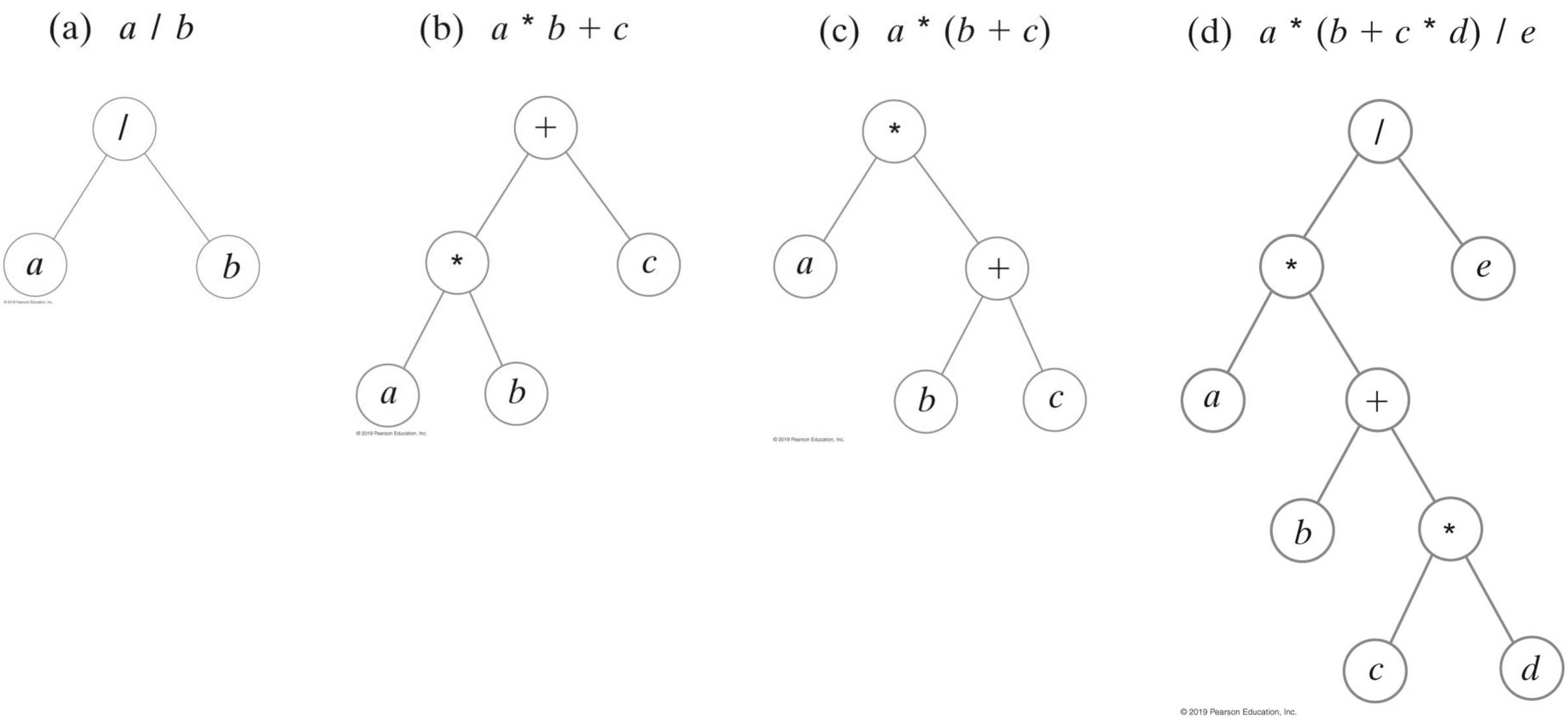
```java
// from BinaryTreeTestDriver.java
public static void TestSeg2421() {
     System.out.println("Testing code in Segment 24.21:");

     // Represent each leaf as a one-node 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

     // 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();
}
```

A binary tree whose nodes contain one-letter strings (Segment 24.21)



Expression Trees

- FIGURE 24-15 Expression trees for four algebraic expressions



Expression Trees

- Algorithm for postorder traversal of an expression tree.

Algorithm **evaluate(expressionTree)**

if (expressionTree *is empty*)

return 0

else

{

firstOperand = evaluate(*left subtree of* expressionTree)

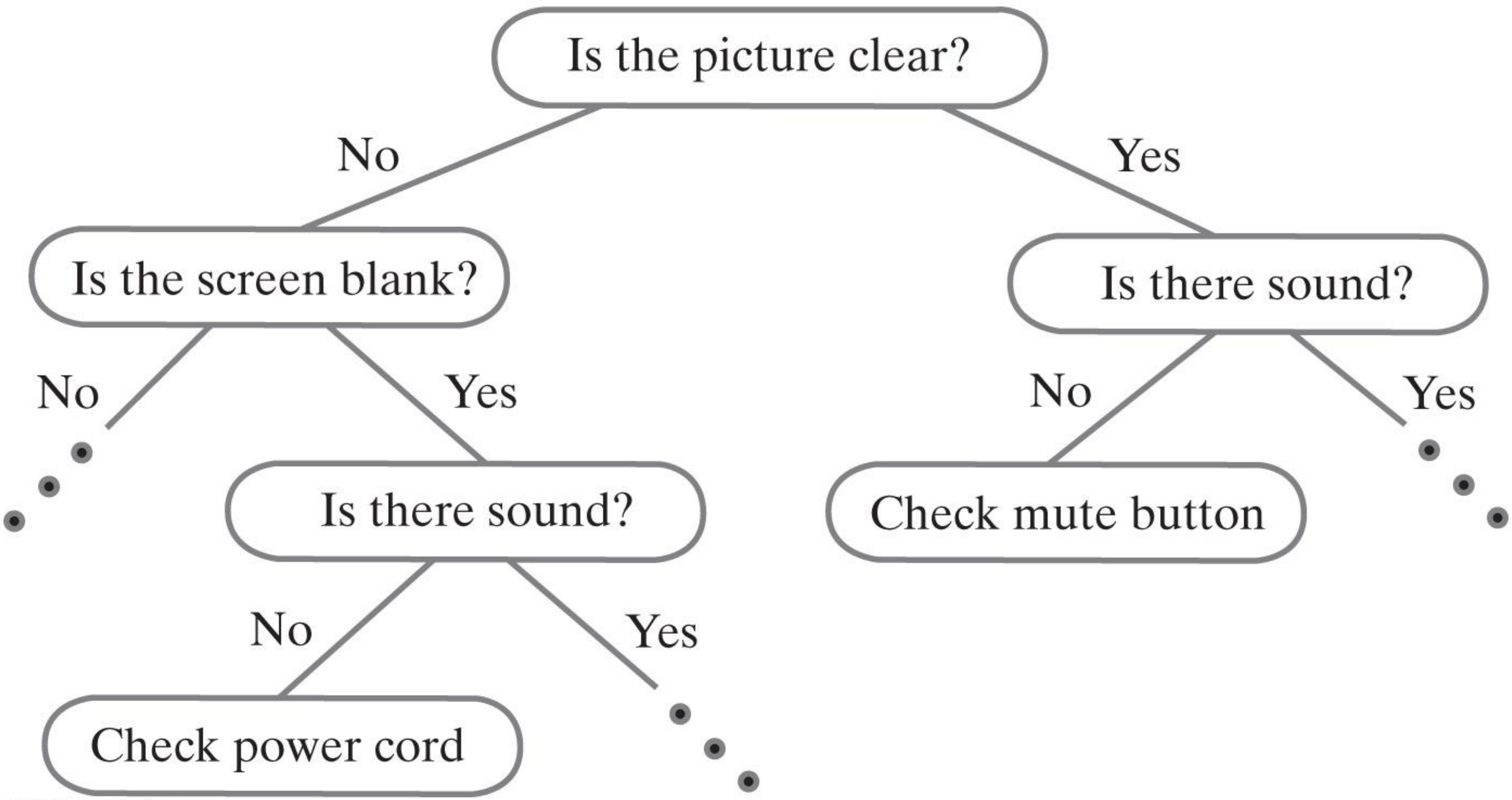
secondOperand = evaluate(*right subtree of* expressionTree)

operator = *the root of* expressionTree

return *the result of the operation* operator *and its operands* firstOperand *and secondOperand*

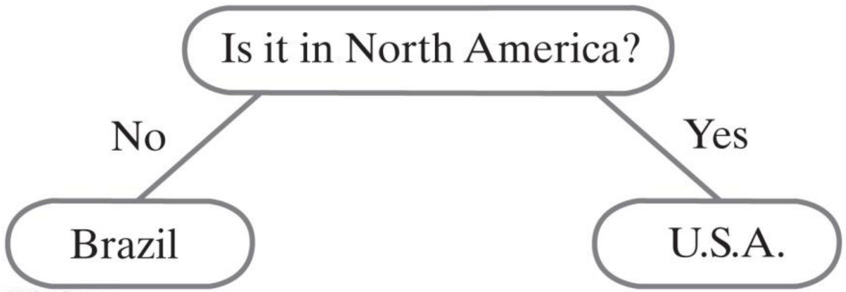
}

Expert System Using A Decision Tree



Expert System Using A Decision Tree

- FIGURE 24-17 An initial decision tree for a guessing game



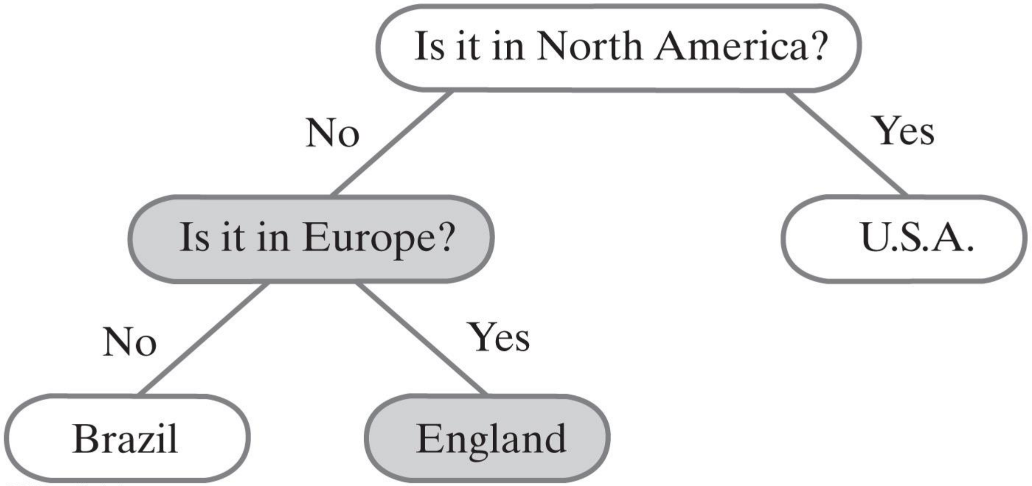


FIGURE 24-18 The decision tree for a guessing game after acquiring another fact

Decision Tree Interface

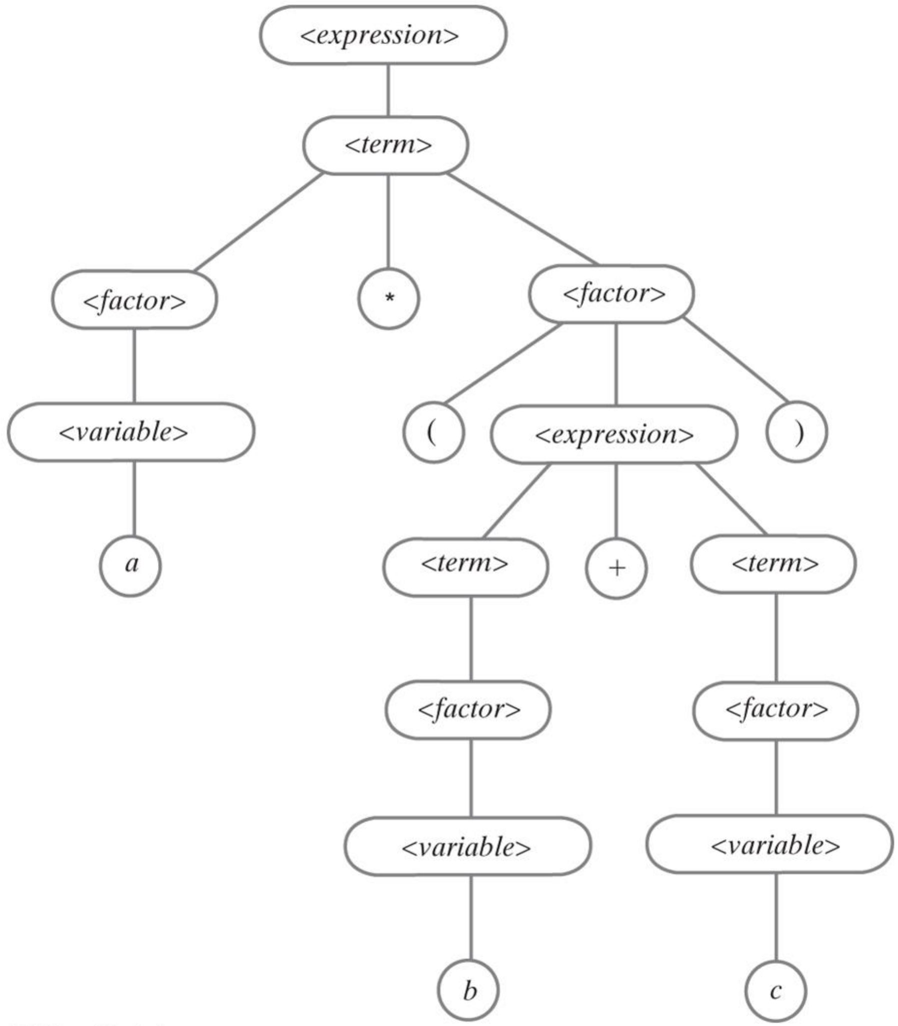
```
public interface DecisionTreeInterface<T> extends BinaryTreeInterface<T> {
                /**
                 * Gets the data in the current node.
                 *
                 * @return The data object in the current node, or null if the current node is
                 *         null.
                 */
                public T getCurrentData();

                /**
                 * Sets the data in the current node. Precondition: The current node is not
                 * null.
                 *
                 * @param newData The new data object.
                 */
                public void setCurrentData(T newData);

                /**
                 * Sets the data in the children of the current node, creating them if they do
                 * not exist. Precondition: The current node is not null.
                 *
                 * @param responseForNo  The new data object for the left child.
                 * @param responseForYes The new data object for the right child.
                 */
                public void setResponses(T responseForNo, T responseForYes);

                /**
                 * Sees whether the current node contains an answer.
                 *
                 * @return True if the current node is a leaf, or false if it is a nonleaf.
                 */
                public boolean isAnswer();

                /**
                 * Sets the current node to its left child. If the child does not exist, sets
                 * the current node to null.
                 */
                public void advanceToNo();

                /**
                 * Sets the current node to its right child. If the child does not exist, sets
                 * the current node to null.
                 */
                public void advanceToYes();

                /** Sets the current node to the root of the tree. */
                public void resetCurrentNode();
}
```

More on General Trees

- Parse tree
  - Check syntax of a string for valid algebraic expression
  - If valid can be expressed as a parse tree
- Parse tree must be a general tree
  - So it can accommodate any expression
- Compilers use parse trees
  - Check syntax, produce code

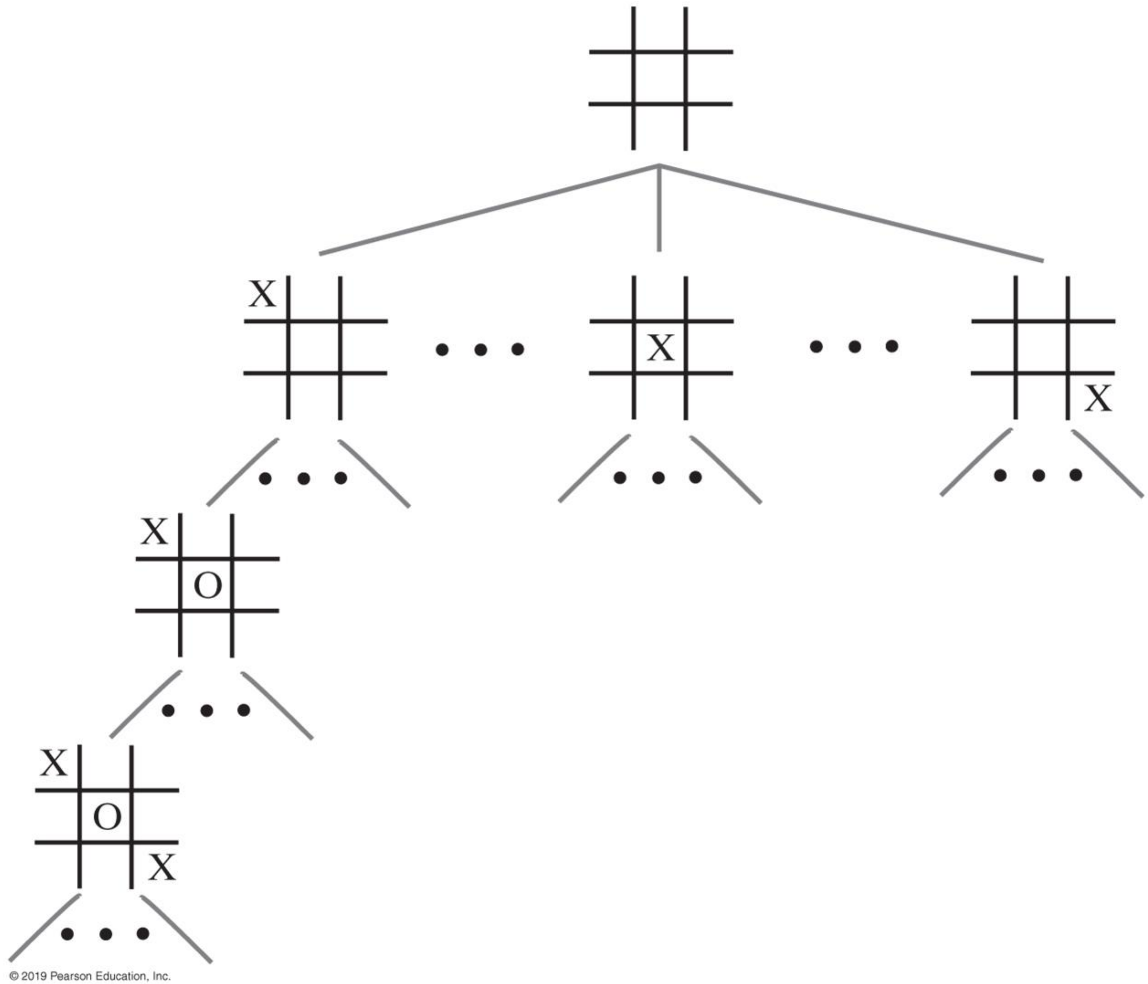
Parse Tree for an Equation

- FIGURE 24-22 A parse tree for the algebraic expression `a * (b + c)`



Parse Tree for a Game

- FIGURE 24-23 A portion of a game tree for tic-tac-toe

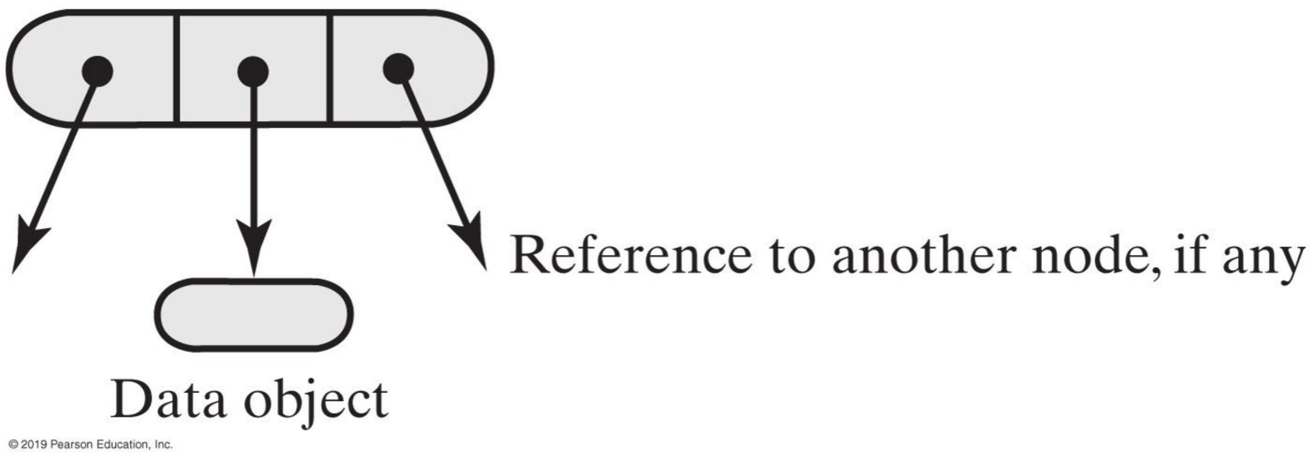


Binary Tree

- Define a BinaryTreeInterface
  - be able to set the root of the tree and retrieve data
  - in addition to TreeInterface
- BinaryNode class
  - Able to use left/right children
  - Get other characteristics of the node
    - Height of tree from this node down
    - Copy all nodes to another tree
    - Get number of nodes from this node down

Nodes in a Binary Tree

- FIGURE 25-1 A node in a binary tree



BinaryNode class – constructors and data

```
class BinaryNode<T> {
		private T data;
		private BinaryNode<T> leftChild; // Reference to left child
		private BinaryNode<T> rightChild; // Reference to right child

		public BinaryNode() {
			this(null); // Call next constructor
		} // end default constructor

		public BinaryNode(T dataPortion) {
			this(dataPortion, null, null); // Call next constructor
		} // end constructor

		public BinaryNode(T dataPortion, BinaryNode<T> newLeftChild, BinaryNode<T> newRightChild) {
			data = dataPortion;
			leftChild = newLeftChild;
			rightChild = newRightChild;
		} // end constructor

		/**
		 * Retrieves the data portion of this node.
		 *
		 * @return The object in the data portion of the node.
		 */
		public T getData() {
			return data;
		} // end getData

		/**
		 * Sets the data portion of this node.
		 *
		 * @param newData The data object.
		 */
		public void setData(T newData) {
			data = newData;
		} // end setData
```

BinaryNode – children methods

```
/**
 * Retrieves the left child of this node.
 *
 * @return A reference to this node's left child.
 */
public BinaryNode<T> getLeftChild() {
		return leftChild;
} // end getLeftChild

/**
 * Sets this node’s left child to a given node.
 *
 * @param newLeftChild A node that will be the left child.
 */
public void setLeftChild(BinaryNode<T> newLeftChild) {
		leftChild = newLeftChild;
} // end setLeftChild

/**
 * Detects whether this node has a left child.
 *
 * @return True if the node has a left child.
 */
public boolean hasLeftChild() {
		return leftChild != null;
} // end hasLeftChild
```

BinaryNode – children methods

```
/**
 * Retrieves the right child of this node.
 *
 * @return A reference to this node's right child.
 */
public BinaryNode<T> getRightChild() {
		return rightChild;
} // end getRightChild

/**
 * Sets this node’s right child to a given node.
 *
 * @param newRightChild A node that will be the right child.
 */
public void setRightChild(BinaryNode<T> newRightChild) {
		rightChild = newRightChild;
} // end setRightChild

/**
 * Detects whether this node has a right child.
 *
 * @return True if the node has a right child.
 */
public boolean hasRightChild() {
		return rightChild != null;
} // end hasRightChild

/**
 * Detects whether this node is a leaf.
 *
 * @return True if the node is a leaf.
 */
public boolean isLeaf() {
		return (leftChild == null) && (rightChild == null);
}
```

BinaryNode utility methods

- use recursive traversal to get the tree height and number of nodes from the current node.

```java
/**
 * Counts the nodes in the subtree rooted at this node.
 * Do this by recursively descending the tree, counting as we go
 * @return The number of nodes in the subtree rooted at this node.
 */
public int getNumberOfNodes() {
		int leftNumber = 0;
		int rightNumber = 0;
		if (leftChild != null)
			leftNumber = leftChild.getNumberOfNodes();
		if (rightChild != null)
			rightNumber = rightChild.getNumberOfNodes();
		return 1 + leftNumber + rightNumber;
} // end getNumberOfNodes
/**
 * Computes the height of the subtree rooted at this node.
 * @return The height of the subtree rooted at this node.
 */
public int getHeight() {
		return getHeight(this); // Call private getHeight
} // end getHeight
/**
 * Copies the subtree rooted at this node.
 *
 * @return The root of a copy of the subtree rooted at this node.
 */
public BinaryNode<T> copy() {
		BinaryNode<T> newRoot = new BinaryNode<>(data);
		if (leftChild != null)
			newRoot.setLeftChild(leftChild.copy());

		if (rightChild != null)
			newRoot.setRightChild(rightChild.copy());

		return newRoot;
} // end copy
/**
 * recursively descend the tree, counting as we go
 * @param node
 * @return tree height
 */
private int getHeight(BinaryNode<T> node) {
		int height = 0;
		if (node != null)
			height = 1 + Math.max(getHeight(node.getLeftChild()), getHeight(node.getRightChild()));
		return height;
}
```

BinaryTreeInterface

- Make sure that all binary trees can set a root node and data

```
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);
}
```

Binary Tree class

- Uses a private BinaryNode as the root
- Options to set up the tree with two existing subtrees

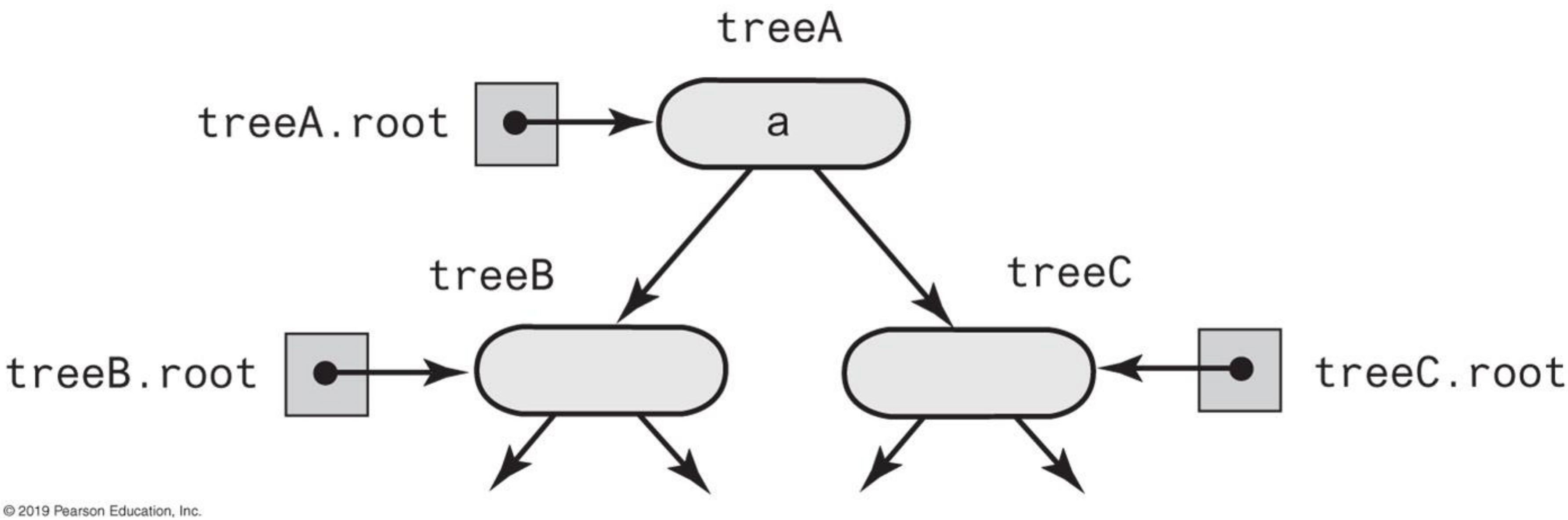
```
public class BinaryTree<T> implements BinaryTreeInterface<T> {
		private BinaryNode<T> root;

		public BinaryTree() {
				root = null;
		} // end default constructor

		/**
		 * Create the tree with a single root node from data
		 * @param rootData
		 */
		public BinaryTree(T rootData) {
				root = new BinaryNode<>(rootData);
		} // end constructor

		/**
		 * Create the tree from a new root node and two other trees that will now become subtrees from root.
		 * @param rootData
		 * @param leftTree
		 * @param rightTree
		 */
		public BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree) {
				initializeTree(rootData, leftTree, rightTree);
		} // end constructor

		public void setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree) {
				initializeTree(rootData, (BinaryTree<T>) leftTree, (BinaryTree<T>) rightTree);
		} // end setTree
```

Initialize the tree

```java
/**
 * Combine two subtrees into a single tree with a new root.
 * @param rootData
 * @param leftTree
 * @param rightTree
 */
private void initializeTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree) {
     // < FIRST DRAFT - See Segments 25.4 - 25.7 for improvements. >
     root = new BinaryNode<T>(rootData);

     if (leftTree != null)
          root.setLeftChild(leftTree.root);

     if (rightTree != null)
          root.setRightChild(rightTree.root);
}
```

Creating a Binary Tree

- FIGURE 25-2 The binary tree `treeA` shares nodes with `treeB` and `treeC`



treeA.setTree(a, treeB, treeC);

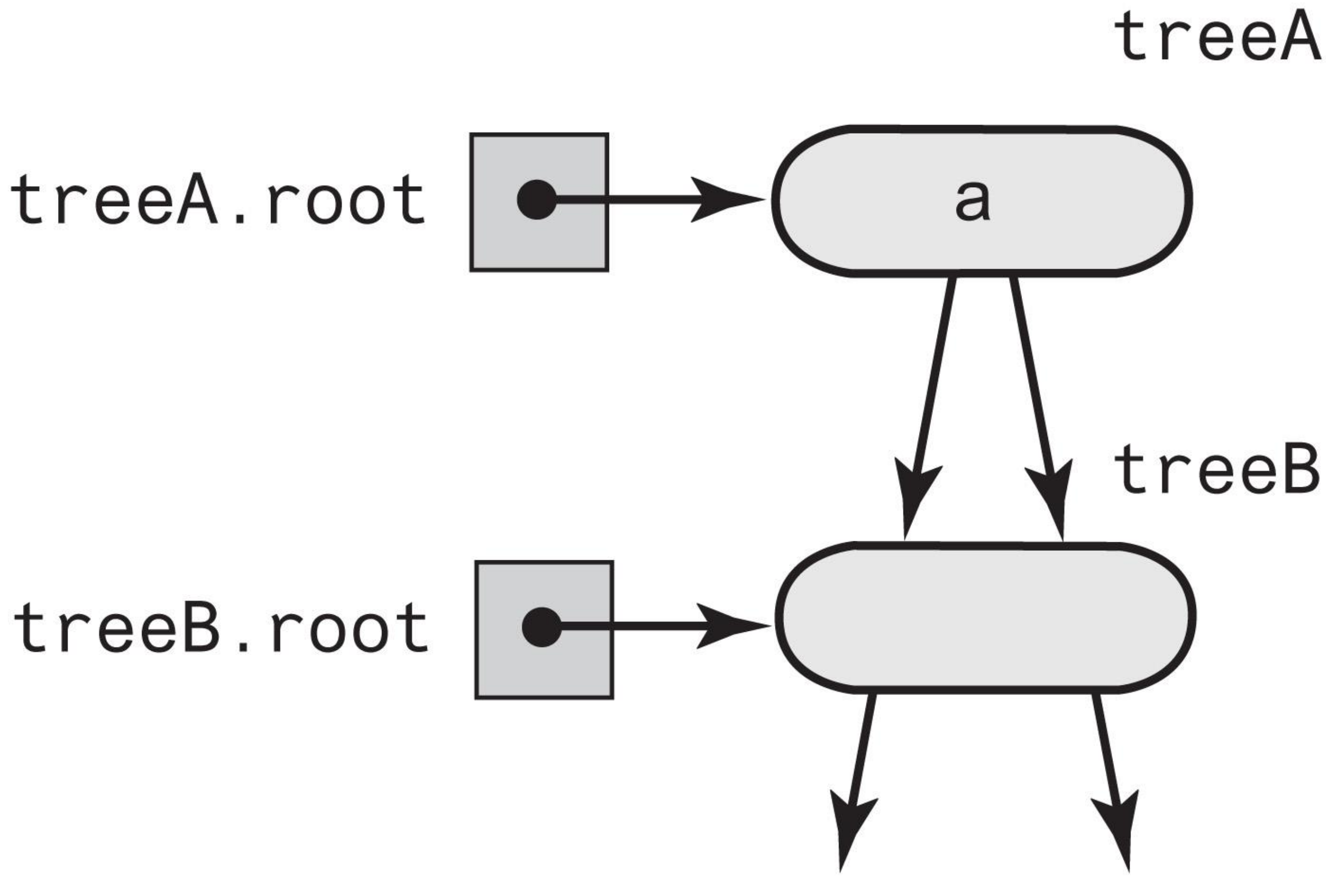
The copy method

- recursively copies all nodes by copying itself then all children.
- We can use this to revise the initialize method.

```
/**
 * Copies the subtree rooted at this node.
 *
 * @return The root of a copy of the subtree rooted at this node.
 */
public BinaryNode<T> copy() {
        BinaryNode<T> newRoot = new BinaryNode<>(data);
        if (leftChild != null)
                newRoot.setLeftChild(leftChild.copy());

        if (rightChild != null)
                newRoot.setRightChild(rightChild.copy());

        return newRoot;
}
```

Additional Challenges

- FIGURE 25-3 `treeA` has identical subtrees
- We will want these to be distinct



Method `initializeTree` Solution

- If left subtree exists and not empty,
    - attach root node to **`r`** as left child.
- Create root node **`r`** containing given data.
- If right subtree exists, not empty, and distinct from left subtree,
    - attach root node to **`r`** as a right child.
- But if right and left subtrees are same,
    - attach copy of right subtree to **`r`** instead.
- If the left subtree exists and differs from the tree object used to call **`initializeTree`**,
    - set the subtree's data field root to null.
- If right subtree exists and differs from the tree object used to call **`initializeTree`**,
    - set subtree's data field root to null.

Revised initialize method

- Copy all nodes to new tree, then clear the old subtrees.

```
private void initializeTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
  {
     root = new BinaryNode<>(rootData);

     if ((leftTree != null) && !leftTree.isEmpty())
        root.setLeftChild(leftTree.root);

     if ((rightTree != null) && !rightTree.isEmpty())
     {
        if (rightTree != leftTree)
           root.setRightChild(rightTree.root);
        else
           root.setRightChild(rightTree.root.copy());
     } // end if

     if ((leftTree != null) && (leftTree != this))
        leftTree.clear();

     if ((rightTree != null) && (rightTree != this))
        rightTree.clear();
  }
```

Binary Tree accessor/mutator methods

```
public void setRootData(T rootData)
  {
     root.setData(rootData);
  } // end setRootData

  public T getRootData()
  {
     if (isEmpty())
        return null;
     else
        return root.getData();
  } // end getRootData

  public boolean isEmpty()
  {
     return root == null;
  } // end isEmpty

  public void clear()
  {
     root = null;
  } // end clear

  protected void setRootNode(BinaryNode<T> rootNode)
  {
     root = rootNode;
  } // end setRootNode

  protected BinaryNode<T> getRootNode()
  {
     return root;
  }
```

Binary Tree – counting and height

- These methods use recursive BinaryNode methods.

```
public int getHeight()
  {
     int height = 0;
     if (root != null)
        height = root.getHeight();
     return height;
  } // end getHeight

  public int getNumberOfNodes()
  {
     int numberOfNodes = 0;
     if (root != null)
        numberOfNodes = root.getNumberOfNodes();
     return numberOfNodes;
  }
```

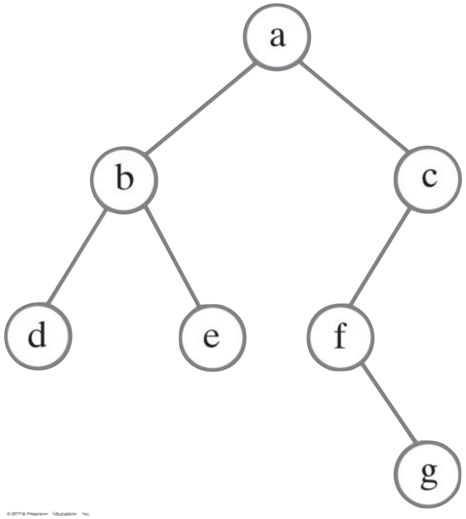
BinaryNode – counting and height

- Use of recursive traversal

```java
/**
 * Counts the nodes in the subtree rooted at this node.
 * Do this by recursively descending the tree, counting as we go.
 *
 * @return The number of nodes in the subtree rooted at this node.
 */
public int getNumberOfNodes() {
    int leftNumber = 0;
    int rightNumber = 0;
    if (leftChild != null)
        leftNumber = leftChild.getNumberOfNodes();
    if (rightChild != null)
        rightNumber = rightChild.getNumberOfNodes();
    return 1 + leftNumber + rightNumber;
} // end getNumberOfNodes

/**
 * Computes the height of the subtree rooted at this node.
 *
 * @return The height of the subtree rooted at this node.
 */
public int getHeight() {
    return getHeight(this); // Call private getHeight
}

/**
 * recursively descend the tree, counting as we go
 * @param node
 * @return tree height
 */
private int getHeight(BinaryNode<T> node) {
    int height = 0;
    if (node != null)
        height = 1 + Math.max(getHeight(node.getLeftChild()), getHeight(node.getRightChild()));
    return height;
}
```

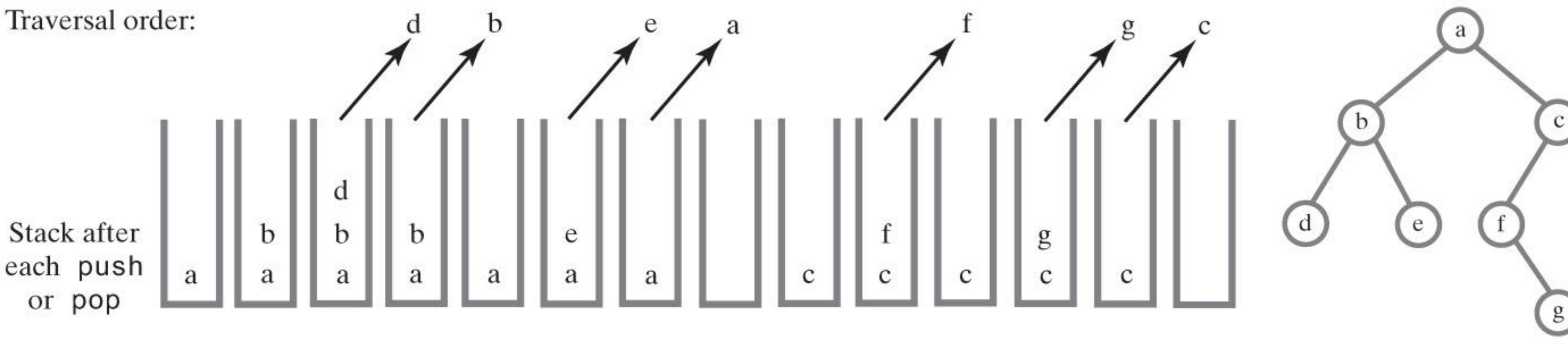
Traversing a binary tree recursively

```
public void recursiveInorderTraverse() {
        recursiveInorderTraverse(root);
}

private void recursiveInorderTraverse(BinaryNode<T> node) {

        if(node != null) {
                recursiveInorderTraverse(node.getLeftChild());
                System.out.print(node.getData() + " ");
                recursiveInorderTraverse(node.getRightChild());
        }
}
```



Non-recursive Traversal

- FIGURE 25-5 Using a stack to perform an in-order traversal of a binary tree



Non-recursive Inorder traversal

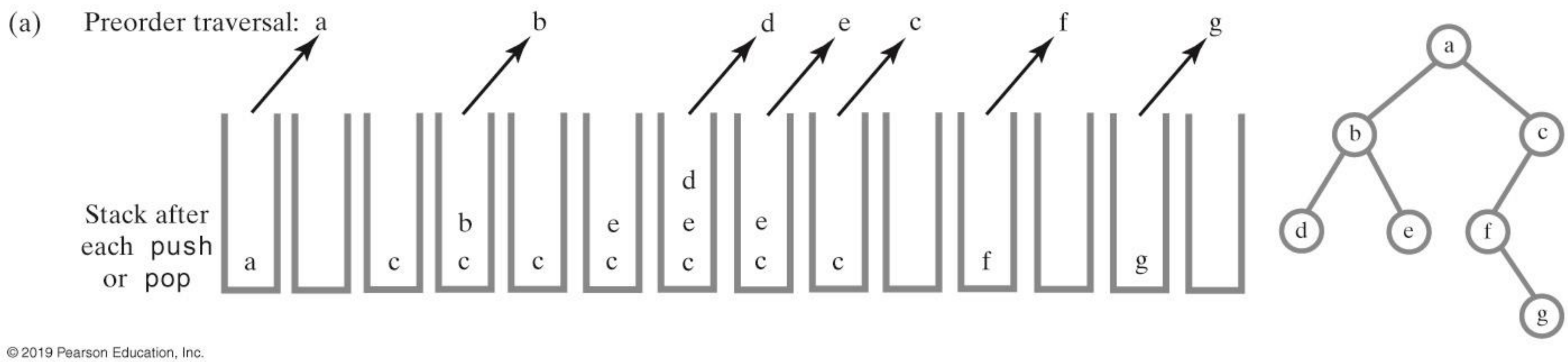
- Uses a Stack

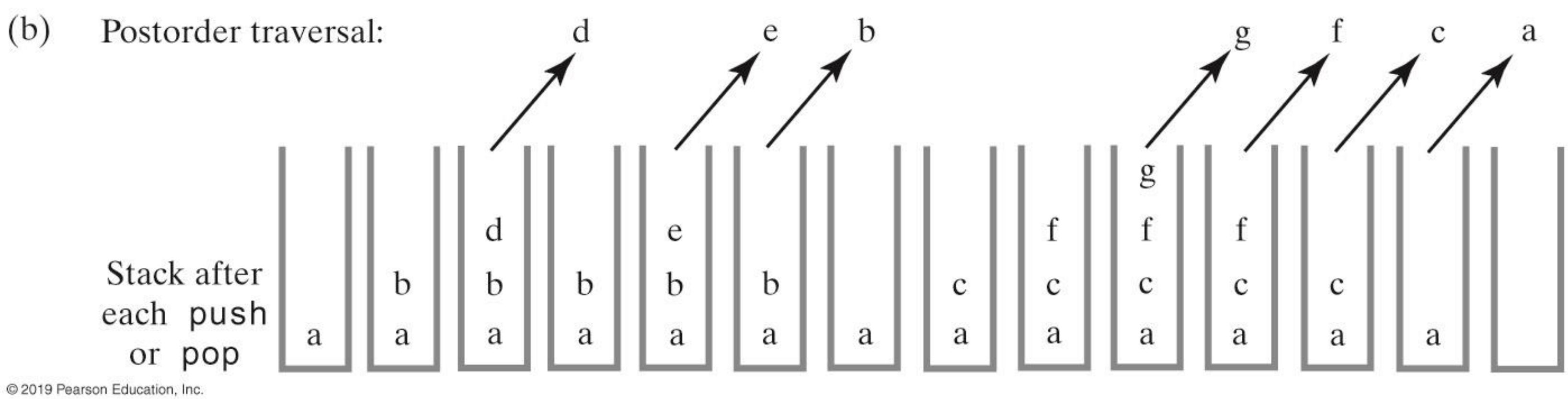
```
public void iterativeInorderTraverse()
   {
      StackInterface<BinaryNode<T>> nodeStack = new LinkedStack<>();
      BinaryNode<T> currentNode = root;

      while (!nodeStack.isEmpty() || (currentNode != null))
      {
         // Find leftmost node with no left child
         while (currentNode != null)
         {
            nodeStack.push(currentNode);
            currentNode = currentNode.getLeftChild();
         } // end while

         // Visit leftmost node, then traverse its right subtree
         if (!nodeStack.isEmpty())
         {
            BinaryNode<T> nextNode = nodeStack.pop();
            // Assertion: nextNode != null, since nodeStack was not empty
            // before the pop
            System.out.print(nextNode.getData() + " ");
            currentNode = nextNode.getRightChild();
         } // end if
      } // end while
   }
```

Inorder traverse using Iterator

```
private class InorderIterator implements Iterator<T>
   {
      private StackInterface<BinaryNode<T>> nodeStack;
      private BinaryNode<T> currentNode;

      public InorderIterator()
      {
         nodeStack = new LinkedStack<>();
         currentNode = root;
      } // end default constructor

      public boolean hasNext()
      {
         return !nodeStack.isEmpty() || (currentNode != null);
      } // end hasNext

      public T next()
      {
         BinaryNode<T> nextNode = null;

         // Find leftmost node with no left child
         while (currentNode != null)
         {
            nodeStack.push(currentNode);
            currentNode = currentNode.getLeftChild();
         } // end while

         // Get leftmost node, then move to its right subtree
         if (!nodeStack.isEmpty())
         {
            nextNode = nodeStack.pop();
            // Assertion: nextNode != null, since nodeStack was not empty
            // before the pop
            currentNode = nextNode.getRightChild();
         }
         else
            throw new NoSuchElementException();

         return nextNode.getData();
      } // end next

      public void remove()
      {
         throw new UnsupportedOperationException();
      }
```

Using a Stack to Traverse a Binary Tree





Preorder traverse using Iterator

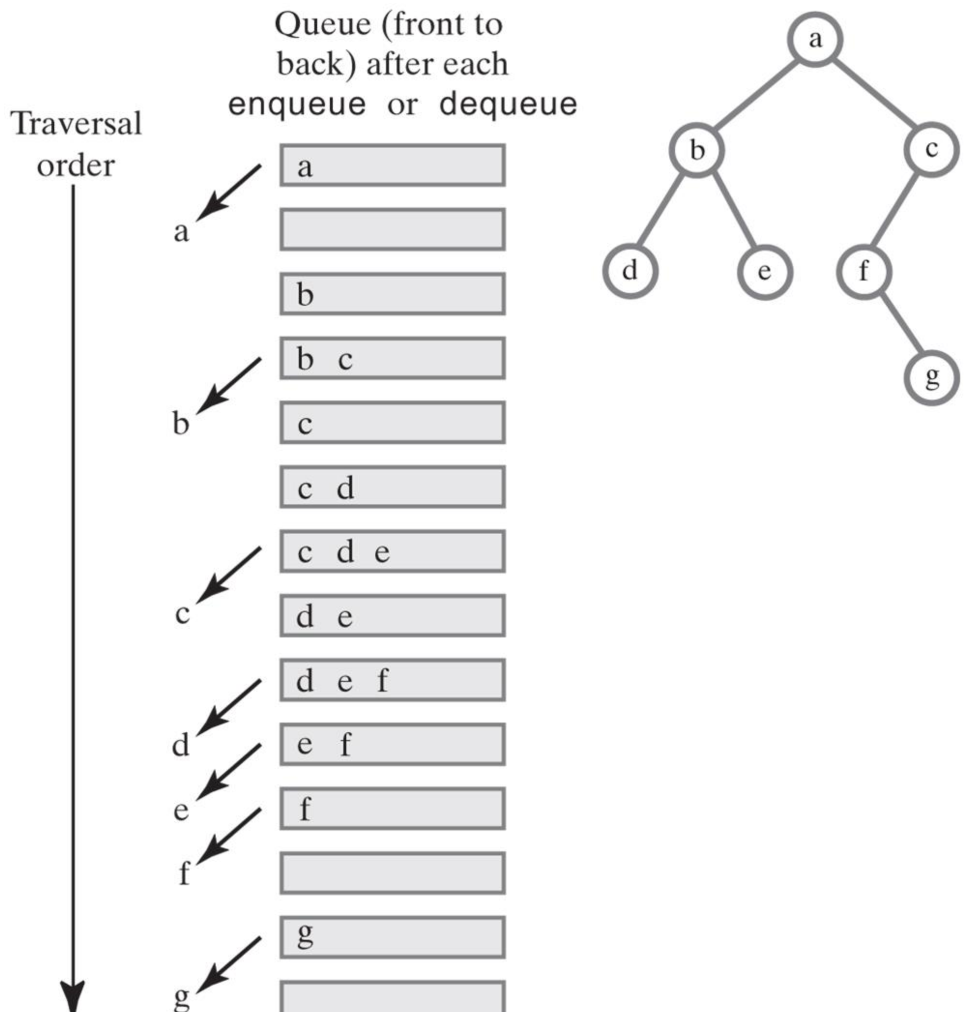
```
private class PreorderIterator implements Iterator<T>
  {
     private StackInterface<BinaryNode<T>> nodeStack;

     public PreorderIterator()
     {
        nodeStack = new LinkedStack<>();
        if (root != null)
           nodeStack.push(root);
     } // end default constructor

     public boolean hasNext()
     {
        return !nodeStack.isEmpty();
     } // end hasNext

     public T next()
     {
        BinaryNode<T> nextNode;

        if (hasNext())
        {
           nextNode = nodeStack.pop();
           BinaryNode<T> leftChild = nextNode.getLeftChild();
           BinaryNode<T> rightChild = nextNode.getRightChild();

           // Push into stack in reverse order of recursive calls
           if (rightChild != null)
              nodeStack.push(rightChild);

           if (leftChild != null)
              nodeStack.push(leftChild);
        }
        else
        {
           throw new NoSuchElementException();
        }
        return nextNode.getData();
     } // end next

     public void remove()
     {
        throw new UnsupportedOperationException();
     } // end remove
  }
```

Postorder traverse using Iterator

```
private class PostorderIterator implements Iterator<T>
  {
     private StackInterface<BinaryNode<T>> nodeStack;
     private BinaryNode<T> currentNode;
     public PostorderIterator()
     {
        nodeStack = new LinkedStack<>();
        currentNode = root;
     } // end default constructor
     public boolean hasNext()
     {
        return !nodeStack.isEmpty() || (currentNode != null);
     } // end hasNext
     public T next()
     {
        BinaryNode<T> leftChild, rightChild, nextNode = null;
        // Find leftmost leaf
        while (currentNode != null)
        {
           nodeStack.push(currentNode);
           leftChild = currentNode.getLeftChild();
           if (leftChild == null)
              currentNode = currentNode.getRightChild();
           else
              currentNode = leftChild;
        } // end while
        // Stack is not empty either because we just pushed a node, or
        // it wasn't empty to begin with since hasNext() is true.
        // But Iterator specifies an exception for next() in case
        // hasNext() is false.
        if (!nodeStack.isEmpty())
        {
           nextNode = nodeStack.pop();
           // nextNode != null since stack was not empty before pop

           BinaryNode<T> parent = null;
           if (!nodeStack.isEmpty())
           {
              parent = nodeStack.peek();
              if (nextNode == parent.getLeftChild())
                 currentNode = parent.getRightChild();
              else
                 currentNode = null;
           }
           else
              currentNode = null;
        }
        else
        {
           throw new NoSuchElementException();
        } // end if
        return nextNode.getData();
     } // end next
     public void remove()
     {
        throw new UnsupportedOperationException();
     } // end remove
  }
```

Using a Queue for Level-Order Traversal

- FIGURE 25-7 Using a queue to traverse a binary tree in level order



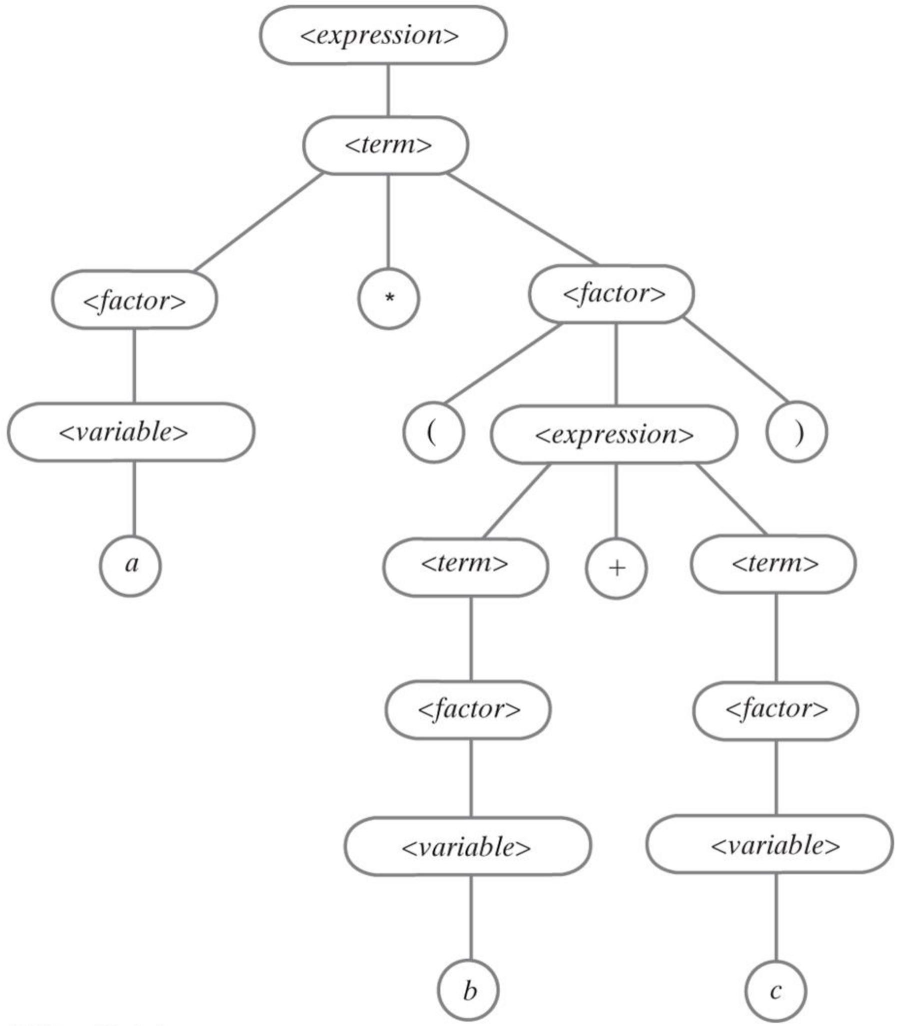
Levelorder traverse using Iterator

- Note use of Queue

```
private class LevelOrderIterator implements Iterator<T>
  {
     private QueueInterface<BinaryNode<T>> nodeQueue;

     public LevelOrderIterator()
     {
        nodeQueue = new LinkedQueue<>();
        if (root != null)
           nodeQueue.enqueue(root);
     } // end default constructor

     public boolean hasNext()
     {
        return !nodeQueue.isEmpty();
     } // end hasNext

     public T next()
     {
        BinaryNode<T> nextNode;

        if (hasNext())
        {
           nextNode = nodeQueue.dequeue();
           BinaryNode<T> leftChild = nextNode.getLeftChild();
           BinaryNode<T> rightChild = nextNode.getRightChild();

           // Add to queue in order of recursive calls
           if (leftChild != null)
              nodeQueue.enqueue(leftChild);

           if (rightChild != null)
              nodeQueue.enqueue(rightChild);
        }
        else
        {
           throw new NoSuchElementException();
        }

        return nextNode.getData();
     } // end next

     public void remove()
     {
        throw new UnsupportedOperationException();
     } // end remove
  }
```

Parse Tree for an Equation

- FIGURE 24-22 A parse tree for the algebraic expression `a * (b + c)`



Expression Tree interface

- Just add an evaluation method to the Binary Tree

```
public interface ExpressionTreeInterface extends BinaryTreeInterface<String> {
/**
 * Computes the value of the expression in this tree.
 *
 * @return The value of the expression.
 */
  public double evaluate();
}
```

Expression Tree – evaluate()

- evaluate() walks through expression tree recursively
  - at leaf, gets value of the data and returns.
  - If the node has children, it is an operator, and invoke compute()

```
public class ExpressionTree extends BinaryTree<String> implements ExpressionTreeInterface {
            public ExpressionTree() {
            } // end default constructor

            public ExpressionTree(String rootData) {
                        super(rootData);
            } // end constructor

            public double evaluate() {
                        return evaluate(getRootNode());
            } // end evaluate

            private double evaluate(BinaryNode<String> rootNode) {
                        double result;

                        if (rootNode == null)
                                    result = 0;
                        else if (rootNode.isLeaf()) {
                                    String variable = rootNode.getData();
                                    result = getValueOf(variable);
                        } else {
                                    double firstOperand = evaluate(rootNode.getLeftChild());
                                    double secondOperand = evaluate(rootNode.getRightChild());
                                    String operator = rootNode.getData();
                                    result = compute(operator, firstOperand, secondOperand);
                        } // end if

                        return result;
            }
```

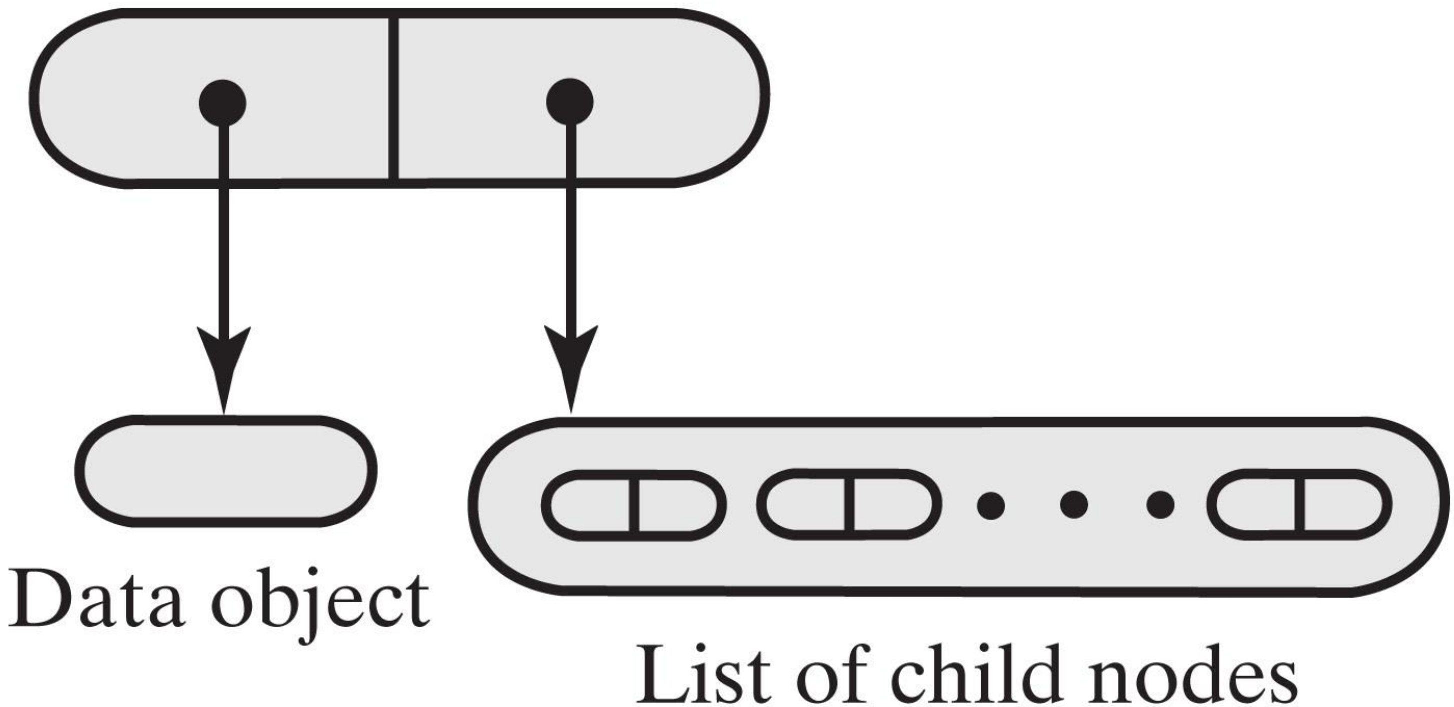
Expression Tree – values and compute

- We would typically set values in other methods, but this demonstrates how values could be intrinsic
- compute() looks at the operator
  - decides on the operation, and then operates on the operands

```
	private double getValueOf(String variable) { // Strings allow multicharacter variables

		double result = 0;

		if (variable.equals("a"))
			result = 2;
		else if (variable.equals("b"))
			result = 3;
		else if (variable.equals("c"))
			result = 4;
		else if (variable.equals("d"))
			result = 5;
		else if (variable.equals("e"))
			result = 2;

		return result;
	} // end getValueOf

	private double compute(String operator, double firstOperand, double secondOperand) {
		double result = 0;

		if (operator.equals("+"))
			result = firstOperand + secondOperand;
		else if (operator.equals("-"))
			result = firstOperand - secondOperand;
		else if (operator.equals("*"))
			result = firstOperand * secondOperand;
		else if (operator.equals("/"))
			result = firstOperand / secondOperand;

		return result;
	}
```

Representing General Trees

- FIGURE 25-8 A node for a general tree



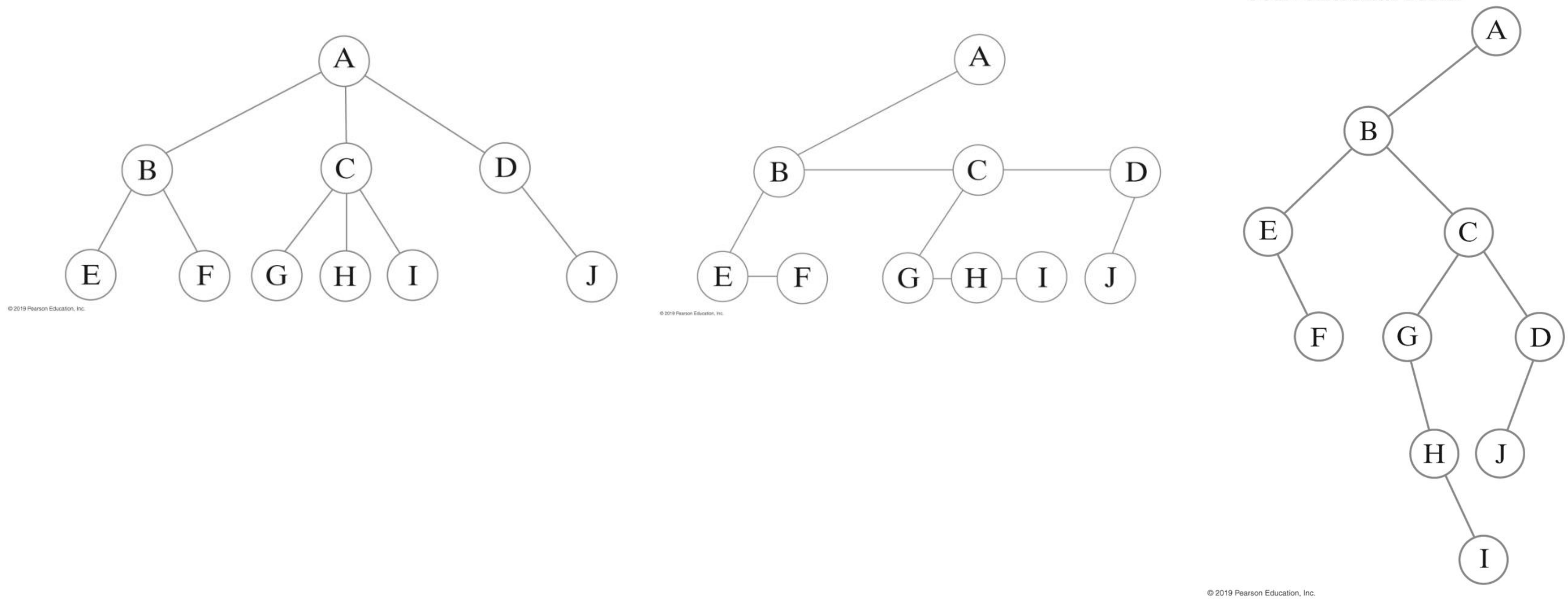
Representing General Trees

- FIGURE 25-9 A general tree and two views of an equivalent binary

(a) A general tree

(b) An equivalent binary tree

(c) The same binary tree in a conventional form



General Tree Node interface

- General tree node can have many children, not just two

```
interface GeneralNodeInterface<T> {
		/**
		 * Get the data from a node
		 * @return
		 */
		public T getData();

		/**
		 * Set the data in a node
		 * @param newData
		 */
		public void setData(T newData);

		/**
		 * Tests to see if this node is a leaf node
		 * @return true if a leaf node with no children
		 */
		public boolean isLeaf();

		/**
		 * Gets and iterator to traverse all child nodes
		 * @return
		 */
		public Iterator<GeneralNodeInterface<T>> getChildrenIterator();

		/**
		 * Adds a child to this node
		 * @param newChild
		 */
		public void addChild(GeneralNodeInterface<T> newChild);
}
```