Class 11 – Search Trees and Heaps

CSIS 3475 Data Structures and Algorithms

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

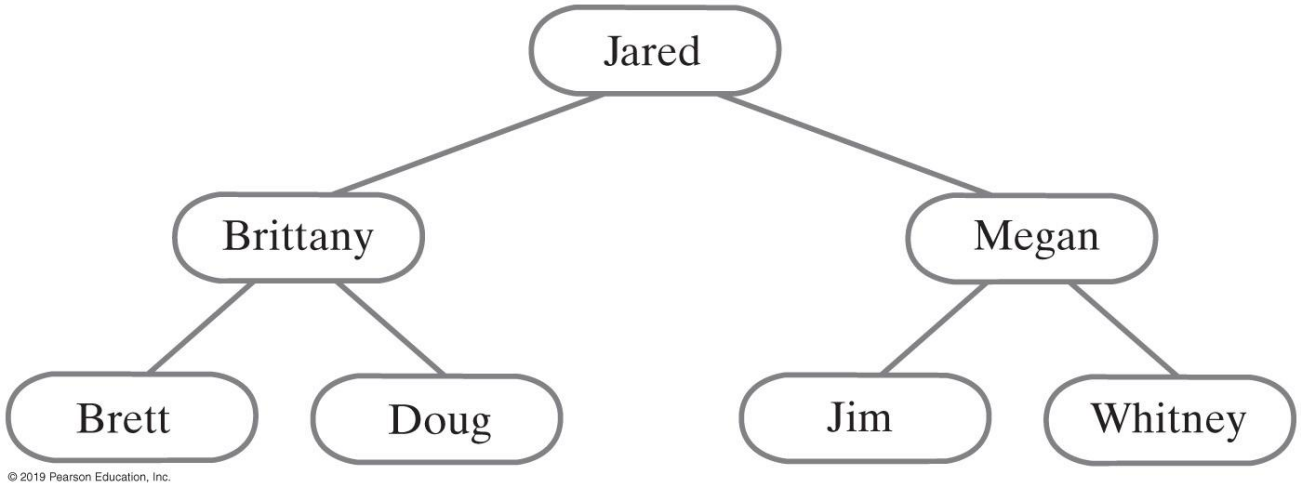
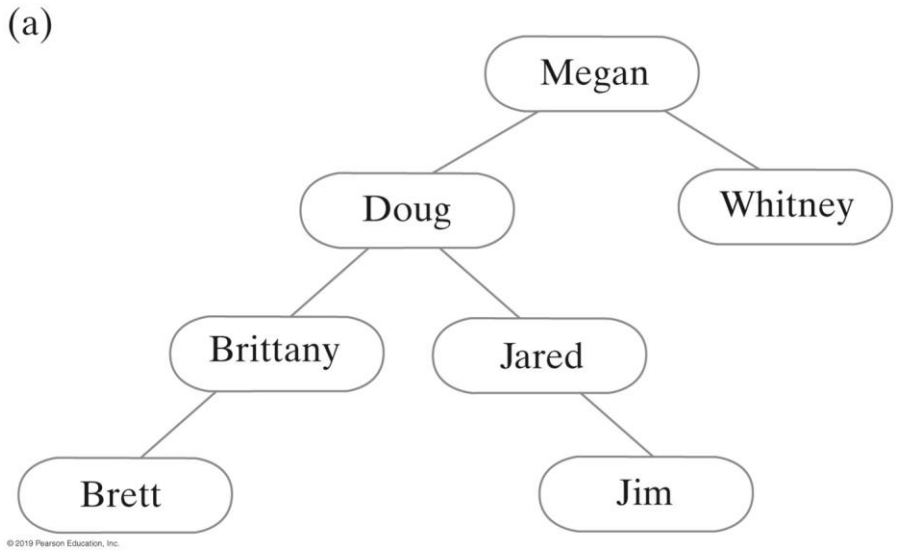
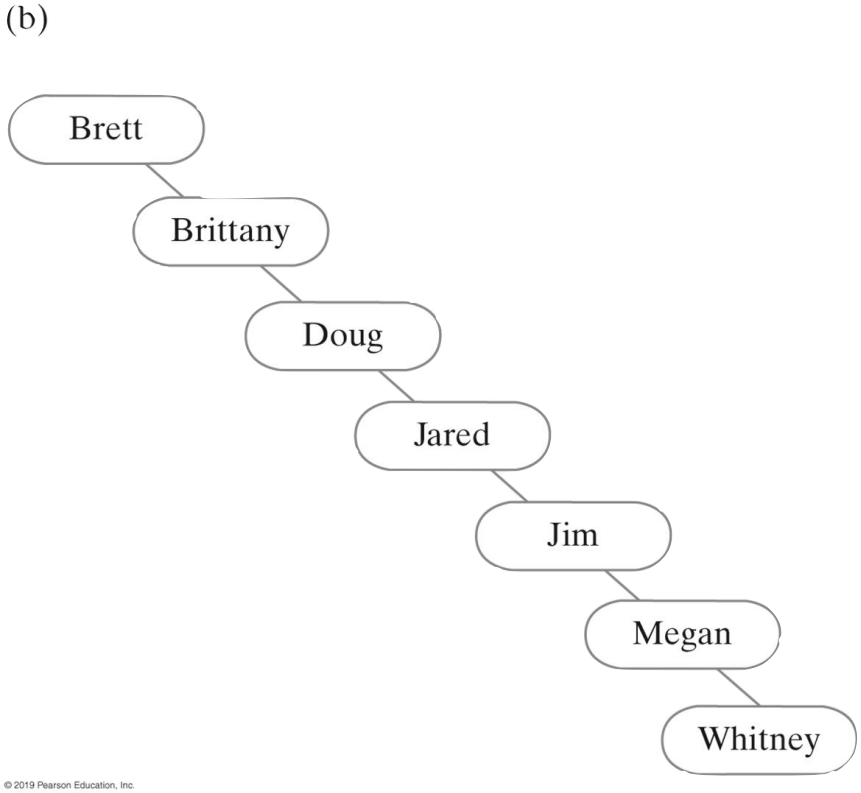


FIGURE 24-19 A binary search tree of names

Binary Search Tree

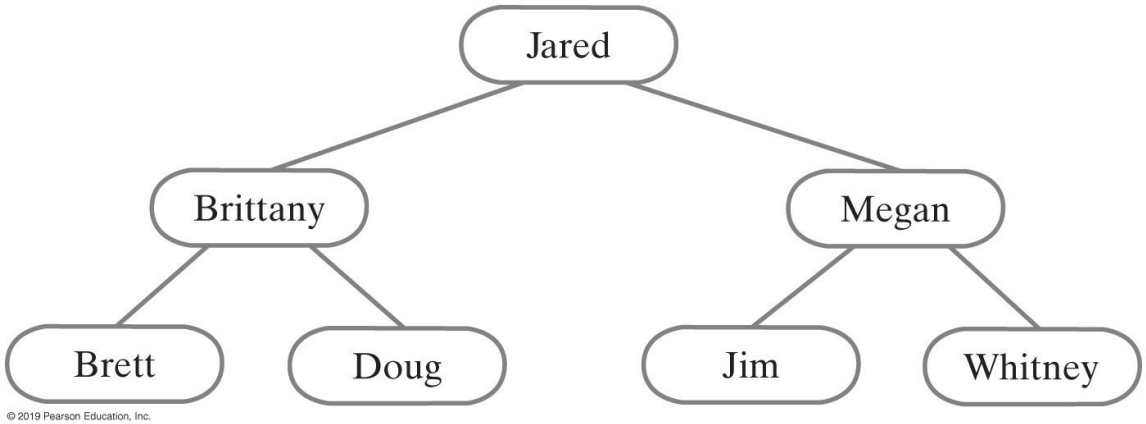
- FIGURE 24-20 Two binary search trees containing the same data as the tree in Figure 24-19





Binary Search Tree

- 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



Search Tree Interface

```
public interface SearchTreeInterface<T extends Comparable<? super T>> extends TreeInterface<T> {
                /**
                 * Searches for a specific entry in this tree.
                 *
                 * @param anEntry An object to be found.
                 * @return True if the object was found in the tree.
                 */
                public boolean contains(T anEntry);

                /**
                 * Retrieves a specific entry in this tree.
                 *
                 * @param anEntry An object to be found.
                 * @return Either the object that was found in the tree or null if no such
                 *         object exists.
                 */
                public T getEntry(T anEntry);

                /**
                 * Adds a new entry to this tree, if it does not match an existing object in the
                 * tree. Otherwise, replaces the existing object with the new entry.
                 *
                 * @param anEntry An object to be added to the tree.
                 * @return Either null if anEntry was not in the tree but has been added, or the
                 *         existing entry that matched the parameter anEntry and has been
                 *         replaced in the tree.
                 */
                public T add(T anEntry);

                /**
                 * Removes a specific entry from this tree.
                 *
                 * @param anEntry An object to be removed.
                 * @return Either the object that was removed from the tree or null if no such
                 *         object exists.
                 */
                public T remove(T anEntry);

                /**
                 * Creates an iterator that traverses all entries in this tree.
                 *
                 * @return An iterator that provides sequential and ordered access to the
                 *         entries in the tree.
                 */
                public Iterator<T> getInorderIterator();
}
```

Understanding the Specifications

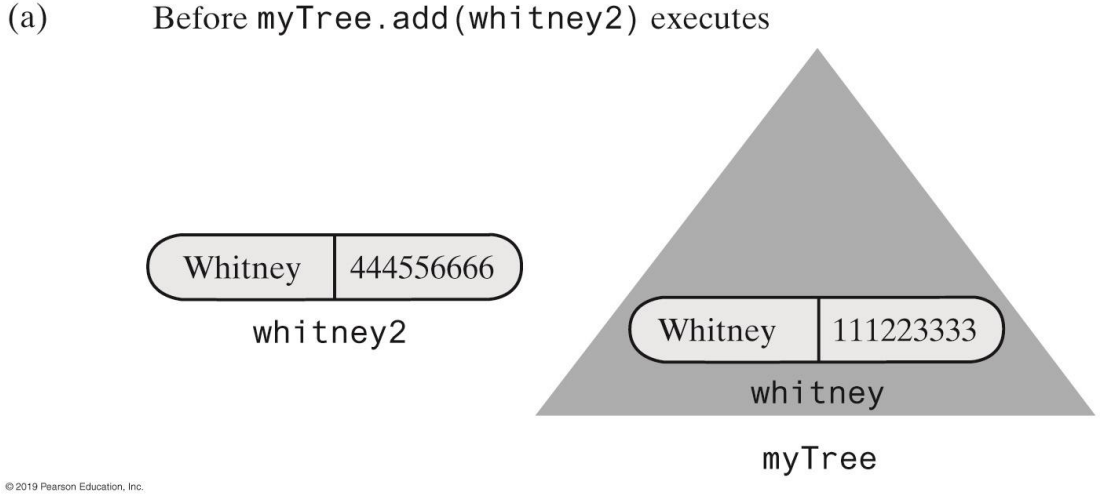
- Methods will use return values instead of exceptions to indicate whether an operation has failed
  - **`getEntry`**, returns same entry it is given to find
  - **`getEntry`** returns an object in tree and matches given entry according to the entry's **`compareTo`** method

Binary Search Tree

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

Adding to a Binary Search Tree

- FIGURE 26-2 Adding an entry that matches an entry already in a binary search tree



Duplicate Entries

- If any entry ***e*** has a duplicate entry ***d***, we arbitrarily require that ***d*** occur in the right subtree of ***e***'s node
- For each node in a binary search tree:
  - Data in a node is greater than data in node's left subtree
  - Data in a node is less than or equal to data in node's right subtree

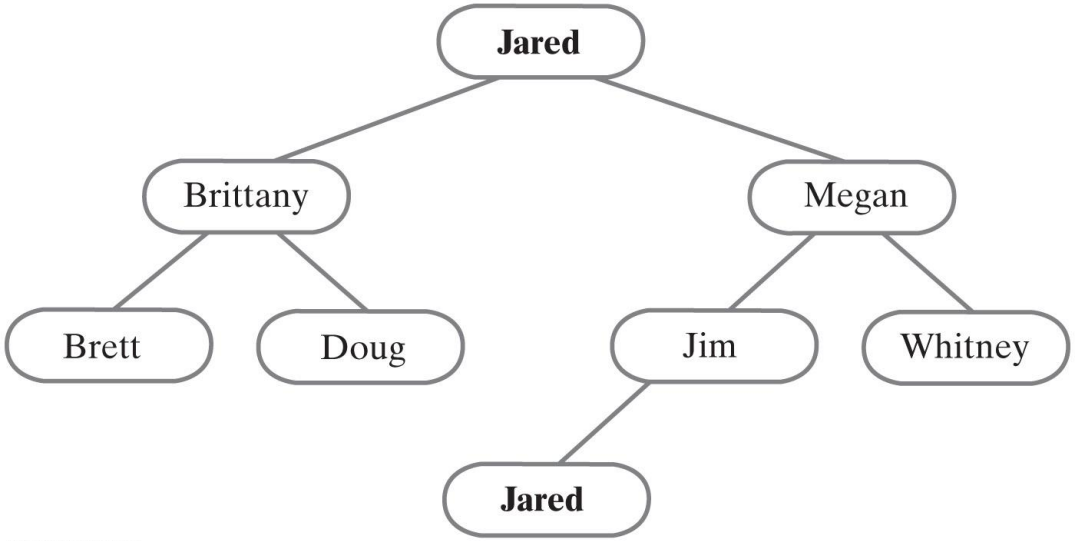


FIGURE 26-3
A binary search tree with duplicate entries

BinarySearchTree constructors

- Note super()
- Allow constructor to set root node. This is preferred.
- setTree() is not supported, as we need to keep the tree sorted.

```
package SearchTrees;
import TreePackage.*;
public class BinarySearchTree<T extends Comparable<? super T>> extends BinaryTree<T>
                        implements SearchTreeInterface<T> {
            public BinarySearchTree() {
                        super();
            } // end default constructor

            public BinarySearchTree(T rootEntry) {
                        super();
                        setRootNode(new BinaryNode<T>(rootEntry));
            } // end constructor

            /**
             * Disable setTree (see Segment 26.6)
             */
            public void setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree) {
                        throw new UnsupportedOperationException();
            }
```

Binary Search Tree

- Pseudocode for recursive search algorithm

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)

Searching and Retrieving

- Recursive algorithm to search a 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)

Searching and Retrieving

- Algorithm that describes actual implementation more closely

Algorithm **bstSearch(binarySearchTreeRoot, desiredObject)**
// Searches a binary search tree for a given object.
//Returns true if the object is found.
if (binarySearchTreeRoot *is* **null**)
    return false
else if (desiredObject == *object in* binarySearchTreeRoot)
    return true
else if (desiredObject < *object in* binarySearchTreeRoot)
    return bstSearch(*left child of* binarySearchTreeRoot, desiredObject)
else
    return bstSearch(*right child of* binarySearchTreeRoot, desiredObject)

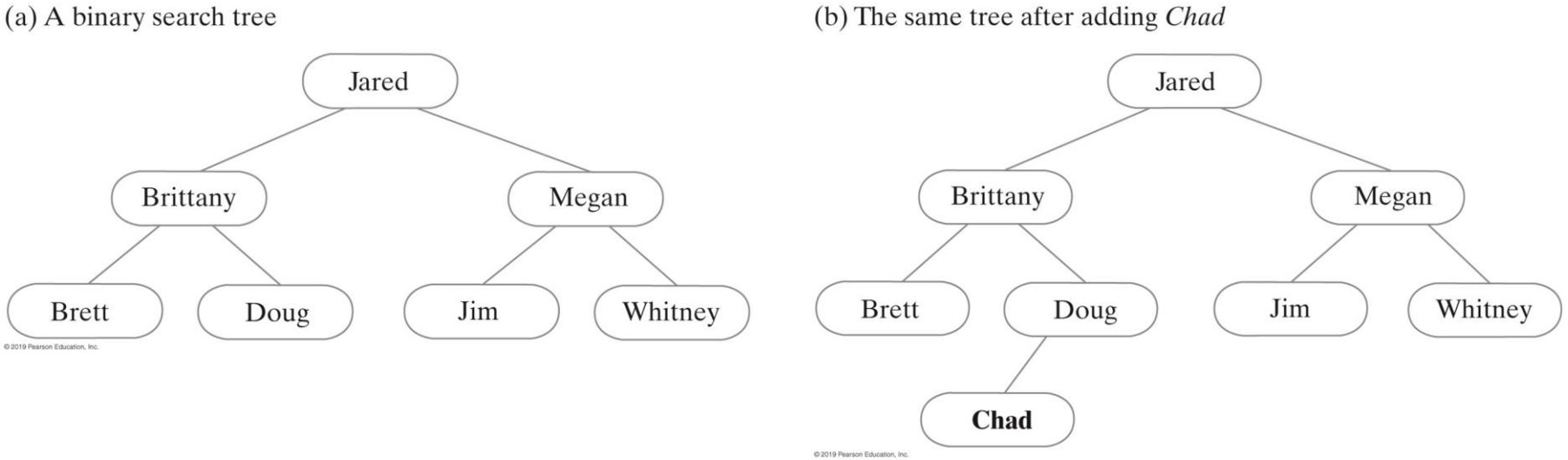
Search implementation

- Descend the tree recursively to find data

```
public T getEntry(T anEntry) {
        return findEntry(getRootNode(), anEntry);
} // end getEntry

/**
 * Find an entry by recursively comparing root node data to entry
 * If equals, we found it
 * Otherwise descend the tree recursively left (less than) or
 * right (greater than)
 * @param rootNode tree containing data
 * @param anEntry search data
 * @return null if not found, otherwise node data
 */
private T findEntry(BinaryNode<T> rootNode, T anEntry) {
        T result = null;

        if (rootNode != null) {
                T rootEntry = rootNode.getData();

                if (anEntry.equals(rootEntry))
                        result = rootEntry;
                else if (anEntry.compareTo(rootEntry) < 0)
                        result = findEntry(rootNode.getLeftChild(), anEntry);
                else
                        result = findEntry(rootNode.getRightChild(), anEntry);
        } // end if

        return result;
}

public boolean contains(T entry) {
        return getEntry(entry) != null;
}
```

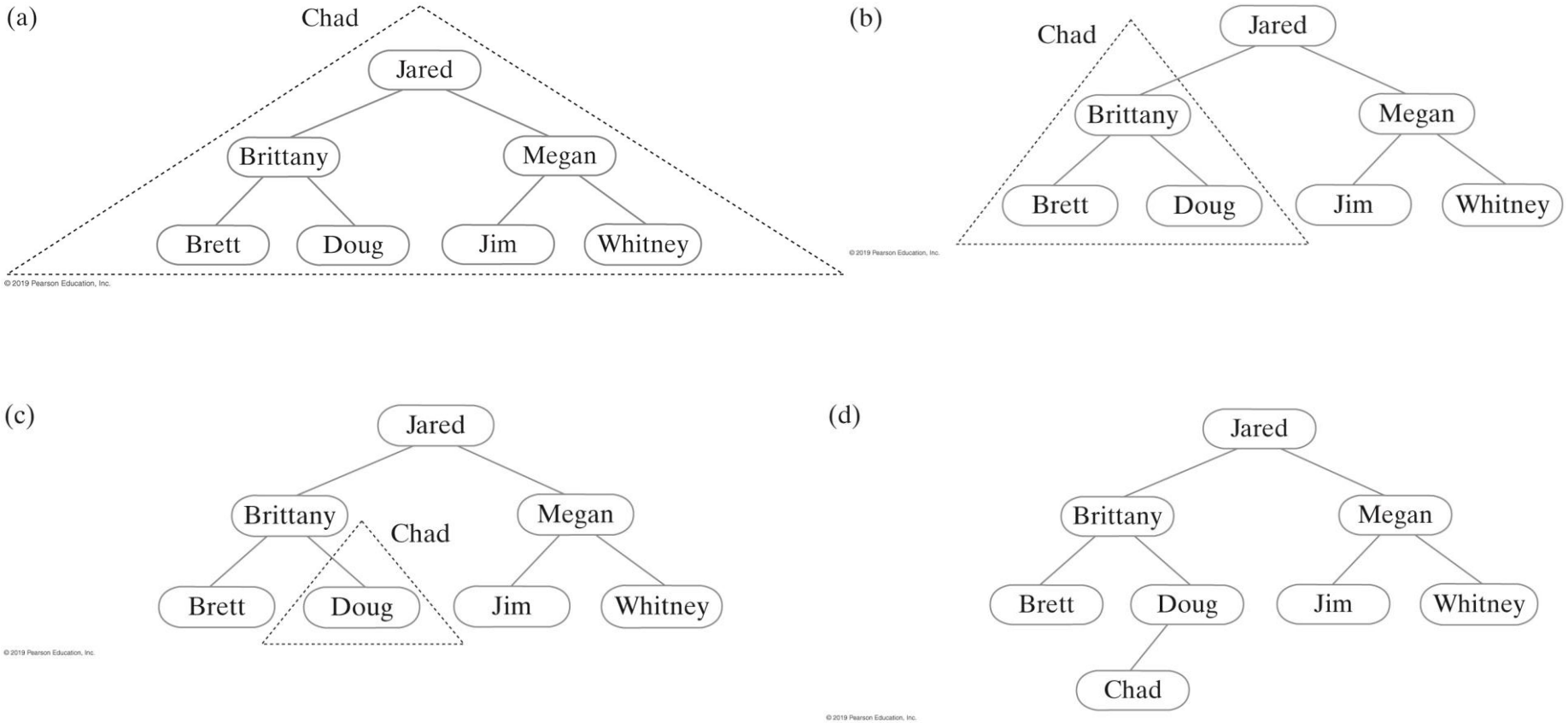
Adding to a Binary Search Tree

- FIGURE 26-4 A binary search tree before and after adding Chad



Adding to a Binary Search Tree

- FIGURE 26-5 Recursively adding Chad to smaller subtrees of a binary search tree



Recursive algorithm for adding a new entry

```
Algorithm addEntry(binarySearchTree, anEntry)
// Adds an entry to a binary search tree that is not empty.
//  Returns null if anEntry did not exist already in the tree. Otherwise, returns the
//tree entry that matched and was replaced by anEntry.
result = null
if (anEntry matches the entry in the root of binarySearchTree)
{
        result = entry in the root
        Replace entry in the root with anEntry
}
else if (anEntry < entry in the root of binarySearchTree)
{
        if (the root of binarySearchTree has a left child)
                result = addEntry(left subtree of binarySearchTree, anEntry)
        else
Give the root a left child containing anEntry
}
```

Recursive algorithm for adding a new entry

```
Algorithm addEntry(binarySearchTree, anEntry)
// Adds an entry to a binary search tree that is not empty.
//  Returns null if anEntry did not exist already in the tree. Otherwise, returns the
//tree entry that matched and was replaced by anEntry.
result = null
if (anEntry matches the entry in the root of binarySearchTree)
{
        result = entry in the root
        Replace entry in the root with anEntry
        }
else if (anEntry < entry in the root of binarySearchTree)
{
        if (the root of binarySearchTree has a left child)
                result = addEntry(left subtree of binarySearchTree, anEntry)
        else
        Give the root a left child containing anEntry
}
else // anEntry > entry in the root of binarySearchTree
{
        if (the root of binarySearchTree has a right child)
                result = addEntry(right subtree of binarySearchTree, anEntry)
        else
                Give the root a right child containing anEntry
}
return result
```

Addition to empty tree as special case

Algorithm **add(binarySearchTree, anEntry)**

// Adds an entry to a binary search tree.

// Returns null *if* anEntry *did not exist already in the tree. Otherwise, returns the*

// tree entry that matched and was replaced by anEntry.

result = **null**

if (binarySearchTree *is empty*)

Create a node containing anEntry *and make it the root of* binarySearchTree

else

result = addEntry(binarySearchTree, anEntry)

return result;

Recursive add() method

- Simply set the root to the entry if the tree is empty
- Otherwise call addEntry(), which recursively walks down the tree looking for a leaf node

```
public T add(T newEntry) {
        T result = null;

        if (isEmpty())
                setRootNode(new BinaryNode<>(newEntry));
        else
                result = addEntry(getRootNode(), newEntry);

        return result;
}
```

Recursive addEntry() method

- Walk down the tree until leaf is reached, then add a new node to either left or right (depending on comparison)

```
/**
 * Adds anEntry to the nonempty subtree rooted at rootNode.
 *
 * Do this by recursively calling addEntry() until a leaf node is reached.
 * @param rootNode
 * @param anEntry
 * @return
 */
private T addEntry(BinaryNode<T> rootNode, T anEntry) {
     // Assertion: rootNode != null

     if(rootNode == null || anEntry == null)
          return null;

     T result = null;

     // root to entry
     // If they are equal, return the old data to the caller
     // and replace with new data in the node

     // if comparison is less or greater than root, then recursively
     // traverse the tree until a leaf is reached and create
     // a new node.

     int comparison = anEntry.compareTo(rootNode.getData());

     if (comparison == 0) {
          result = rootNode.getData();
          rootNode.setData(anEntry); // replace the data
     } else if (comparison < 0) {
          if (rootNode.hasLeftChild())
               result = addEntry(rootNode.getLeftChild(), anEntry);
          else
               rootNode.setLeftChild(new BinaryNode<>(anEntry));
     } else {
          // Assertion: comparison > 0

          if (rootNode.hasRightChild())
               result = addEntry(rootNode.getRightChild(), anEntry);
          else
               rootNode.setRightChild(new BinaryNode<>(anEntry));
     } // end if

     return result;
}
```

Iterative algorithm for adding a new entry

```
Algorithm addEntry(binarySearchTree, anEntry)
// Adds a new entry to a binary search tree that is not empty.
// Returns null if anEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by anEntry.
result  = null
currentNode = root node of binarySearchTree found = false
while (found is false)
{
        if (anEntry matches the entry in currentNode)
        {
                found = true
                result = entry in currentNode
                Replace entry in currentNode with anEntry
        }
        else  if (newEntry  < entry in currentNode)
                        {if (currentNode has a left child)
                                currentNode = leftchildofcurrentNode
                        else
                        {
                                found  = true
                                Give currentNode a left child containing anEntry
                        }
        } // end if-else
        else // anEntry > entry in currentNode
        {
                        if (currentNode has a right child)
                                currentNode = rightchildofcurrentNode
                        else
                        {
                                found  = true
                                Give currentNode a right child containing anEntry
                        }
        }  // end if
} // end while
return result
```

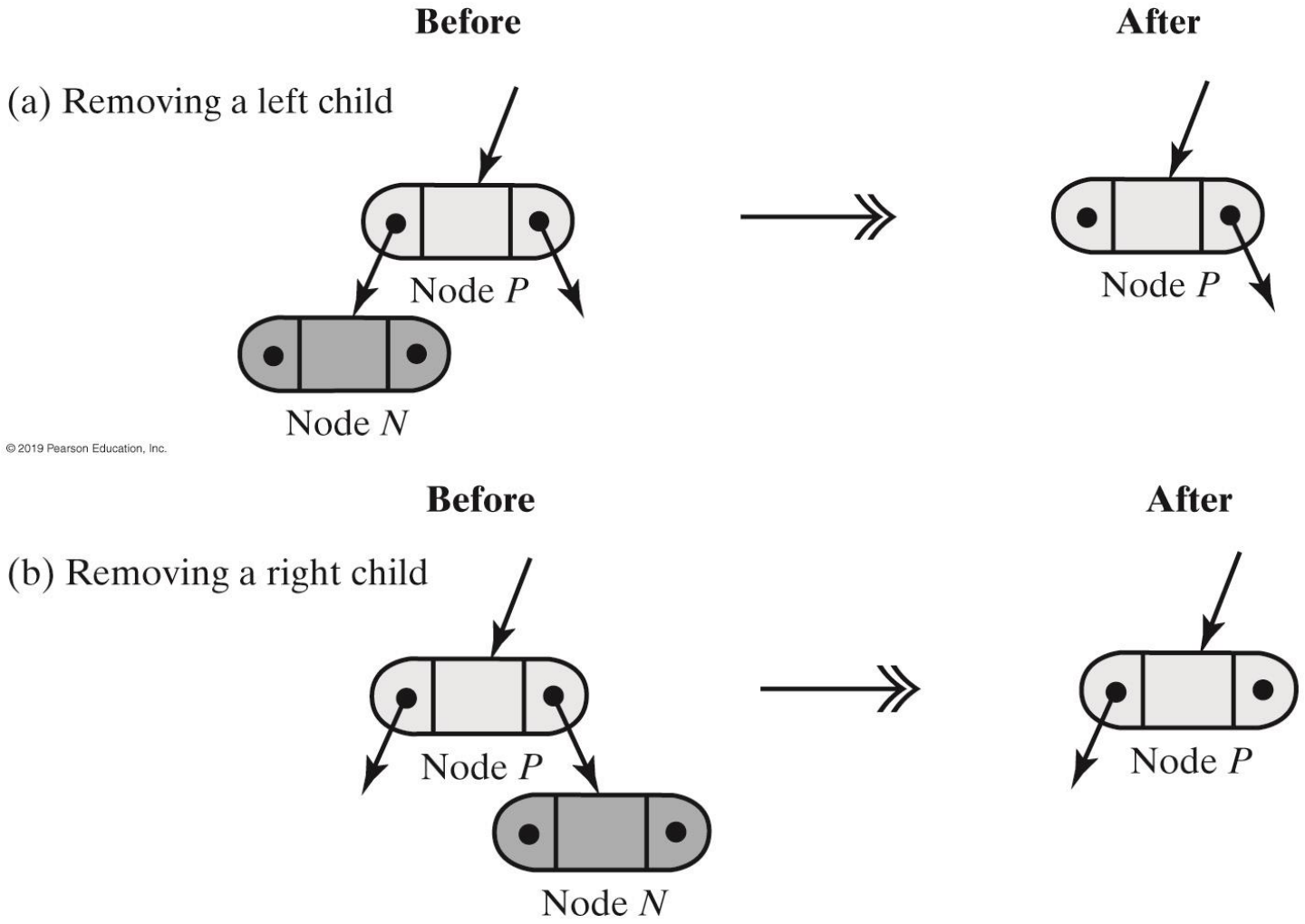
Interative addEntry() method

- Traverse the tree iteratively until a leaf is found by setting the current node to a child node.

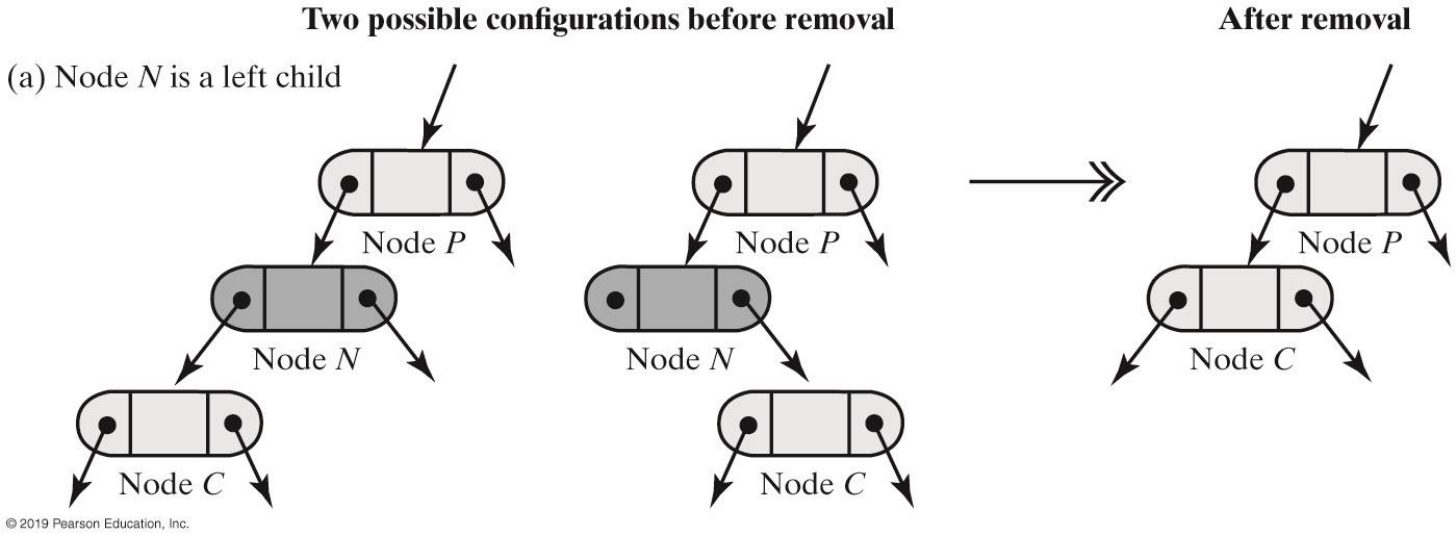
```
private T addEntry(T anEntry) {
    if(isEmpty())
        return null;
    // start with the root node
    BinaryNode<T> currentNode = getRootNode();

    T result = null;
    boolean found = false;

    while (!found) {
        // compare the data. If equals, replace the data
        // if not, walk the tree by setting currentNode to the left
        // or right child until a leaf is found.
        T currentEntry = currentNode.getData();
        int comparison = anEntry.compareTo(currentEntry);

        if (comparison == 0) {
            found = true;
            result = currentEntry;
            currentNode.setData(anEntry);
        } else if (comparison < 0) {
            if (currentNode.hasLeftChild())
                currentNode = currentNode.getLeftChild();
            else {
                // we are at a leaf, create a new node and add it
                found = true;
                currentNode.setLeftChild(new BinaryNode<>(anEntry));
            } // end if
        } else {
            if (currentNode.hasRightChild())
                currentNode = currentNode.getRightChild();
            else {
                // we are at a leaf, create a new node and add it
                found = true;
                currentNode.setRightChild(new BinaryNode<>(anEntry));
            }
        }
    }
    return result;
}
```

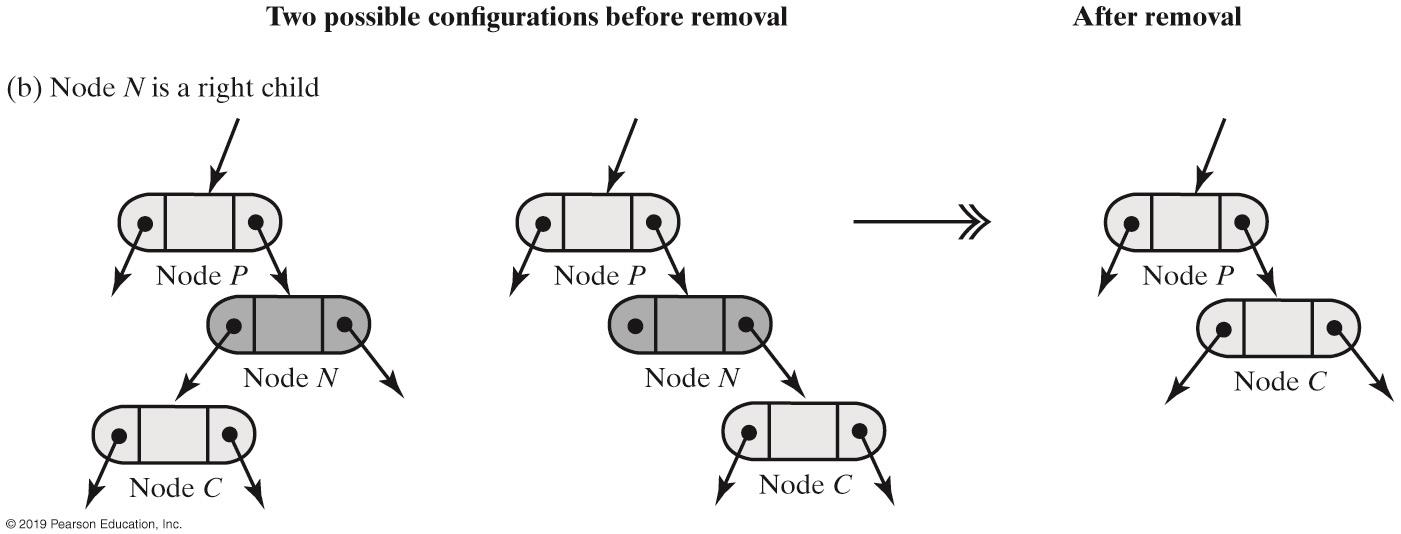
Removing a Value

- FIGURE 26-6 Removing a leaf node N from its parent node P

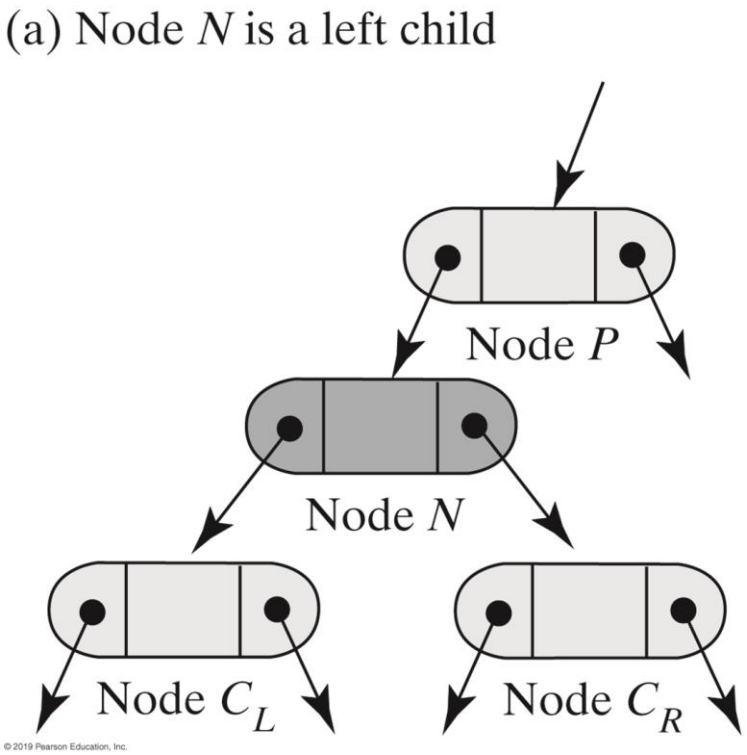


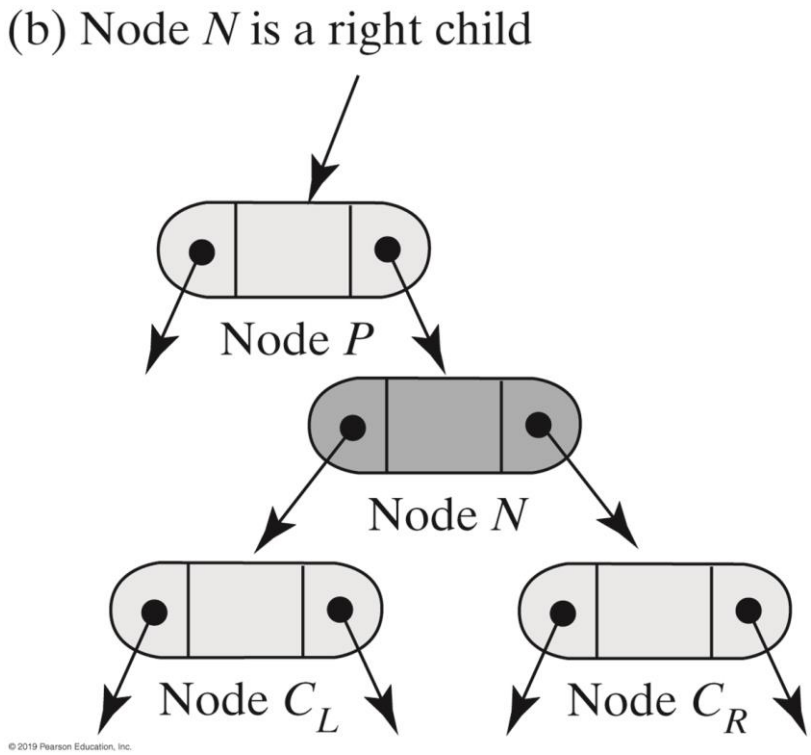
Removing a node *N* from its parent node when *N* has one child



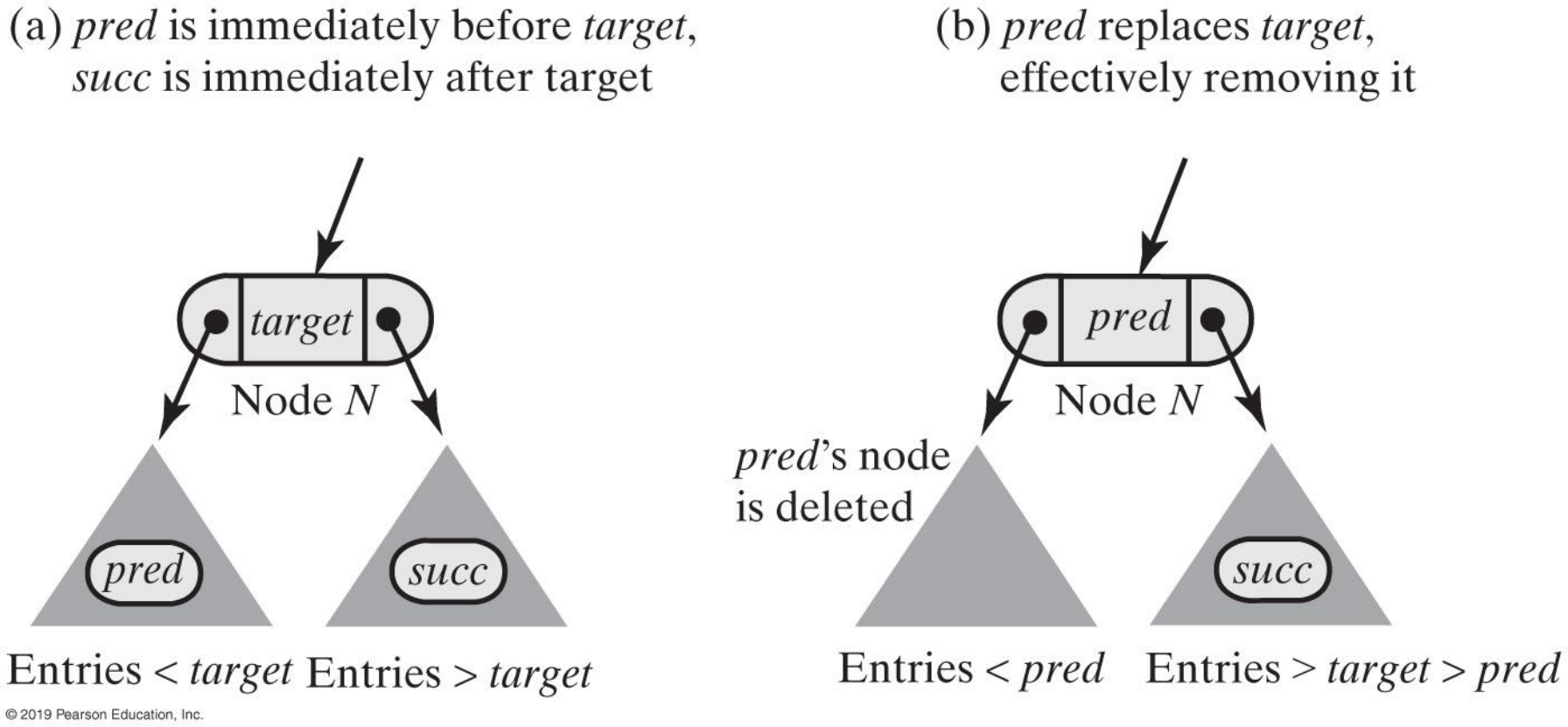


Two possible configurations of a node that has two children



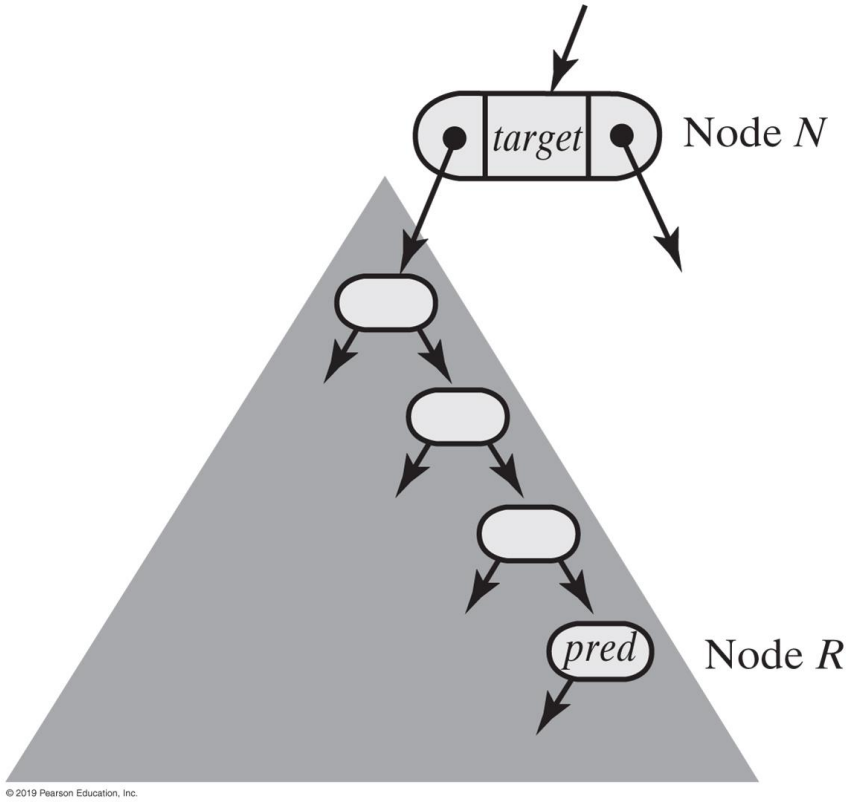


Node and its subtrees before and after removing target



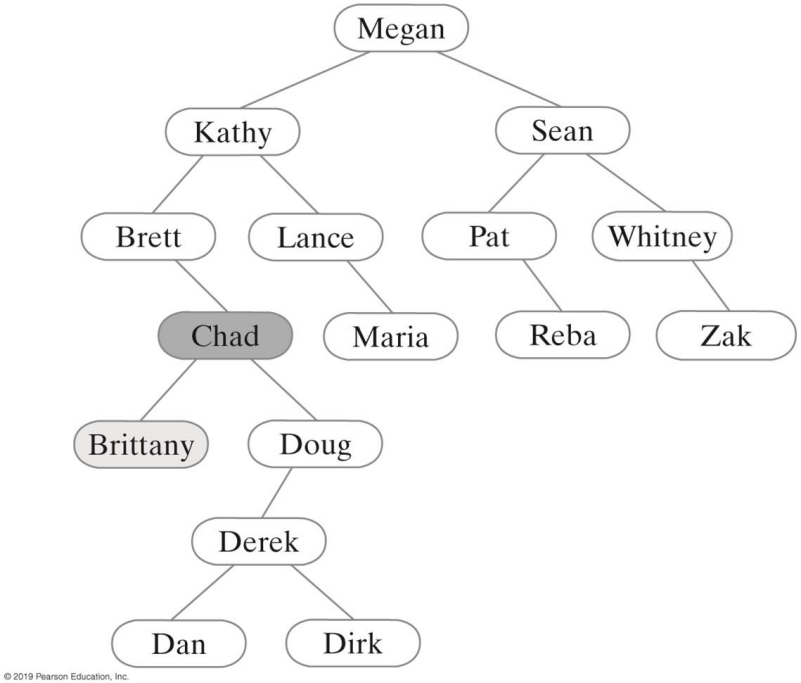
Removing a Value

- The largest entry *pred* (predecessor) in node *N's* left subtree occurs in the subtree's rightmost node *R*

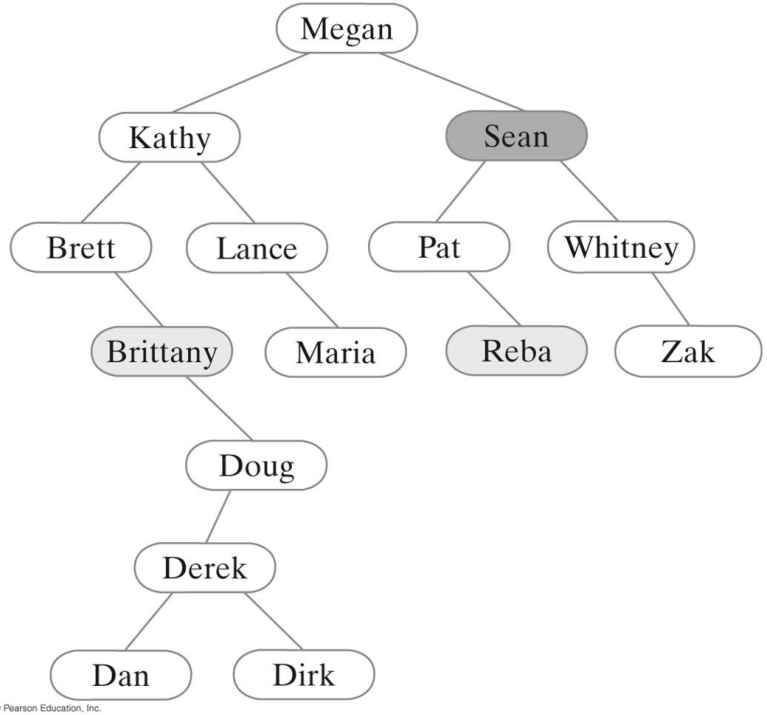


Successive removals from a binary search tree (Part 1)

(a) A binary search tree

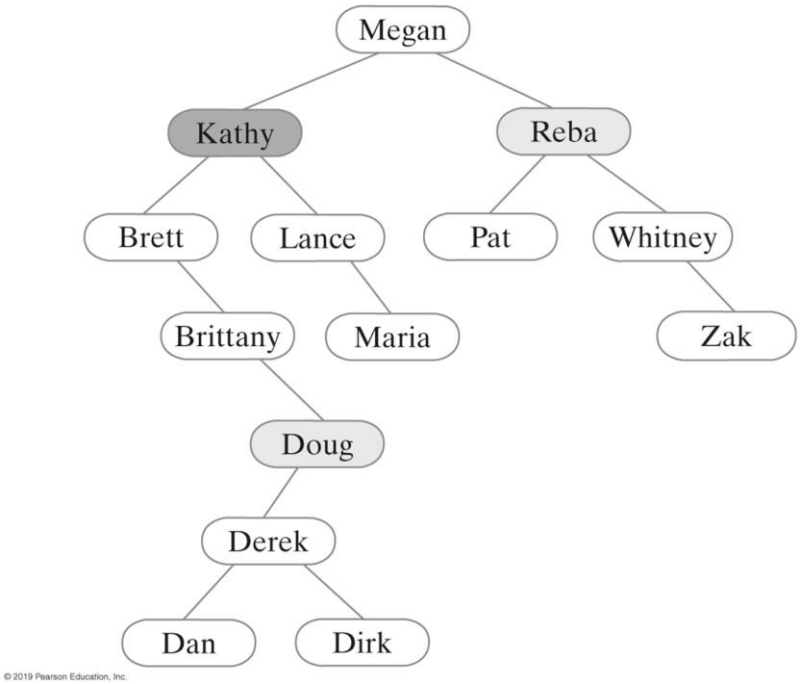


(b) The tree after removing *Chad*

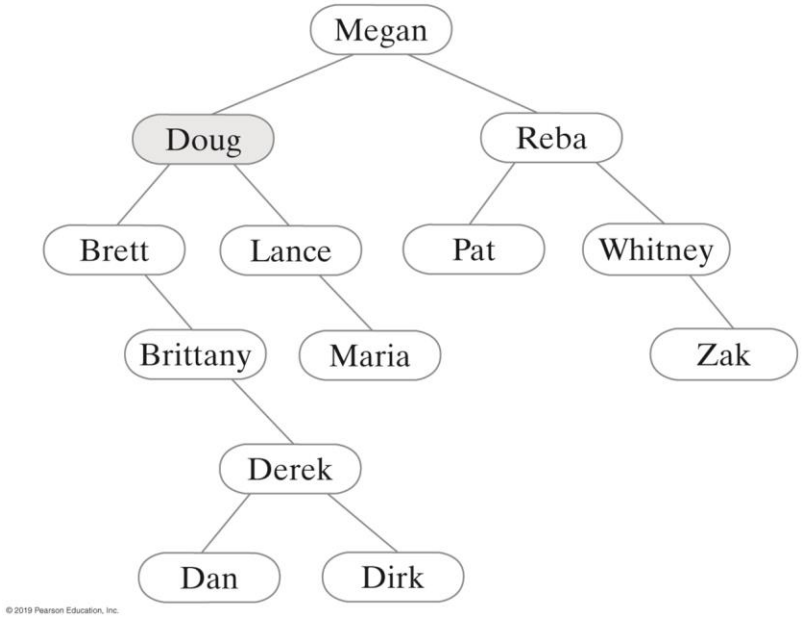


Successive removals from a binary search tree (Part 2)

(c) The tree after removing *Sean*

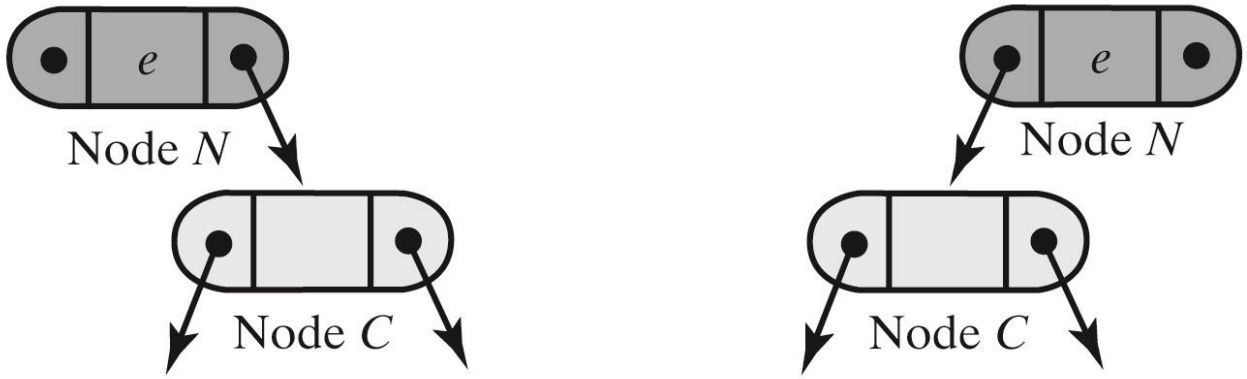


(d) The tree after removing *Kathy*

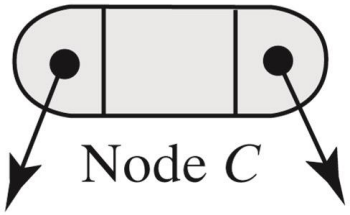


Removing the root when it has one child

(a) Two possible configurations of a tree's root with one child



(b) The tree after removing its root



Recursive Implementation

- Recursive algorithm describes the method's logic at a high level

```
Algorithm remove(binarySearchTree, anEntry)
oldEntry = null
if (binarySearchTree is not empty)
{
    if (anEntry matches the entry in the root of binarySearchTree)
    {
        oldEntry = entry in root
        removeFromRoot(root of binarySearchTree)
    }
    else if (anEntry < entry in root)
        oldEntry = remove(left subtree of binarySearchTree, anEntry)
    else // anEntry > entry in root
        oldEntry = remove(right subtree of binarySearchTree, anEntry)
}
return oldEntry
```

Recursive Implementation – remove()

- Create a null object.
- Call removeEntry(), which recursively traverses until the entry matches.
- Return the old entry

```
public T remove(T anEntry) {
     ReturnObject oldEntry = new ReturnObject(null);
     BinaryNode<T> newRoot = removeEntry(getRootNode(), anEntry, oldEntry);
     setRootNode(newRoot);

     return oldEntry.get();
}
```

Recursive Implementation - removeEntry()

```
/**
 * Removes an entry from the tree rooted at a given node.
 * Recursive implementation
 * @param rootNode root of the tree
 * @param anEntry object to be removed
 * @param oldEntry object whose data field is null
 * @return The root node of the resulting tree. If anEntry matches
 *   an entry in the tree, oldEntry's data field is the entry
 *   that was removed from the tree. Otherwise it is null.
 */
private BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T anEntry, ReturnObject oldEntry) {
    if (rootNode != null) {
        T rootData = rootNode.getData();
        int comparison = anEntry.compareTo(rootData);

        if (comparison == 0) // anEntry == root entry
        {
            oldEntry.set(rootData);
            rootNode = removeFromRoot(rootNode);
        } else if (comparison < 0) // anEntry < root entry
        {
            BinaryNode<T> leftChild = rootNode.getLeftChild();
            BinaryNode<T> subtreeRoot = removeEntry(leftChild, anEntry, oldEntry);
            rootNode.setLeftChild(subtreeRoot);
        } else // anEntry > root entry
        {
            BinaryNode<T> rightChild = rootNode.getRightChild();
            // A different way of coding than for left child:
            rootNode.setRightChild(removeEntry(rightChild, anEntry, oldEntry));
        } // end if
    } // end if

    return rootNode;
}
```

Recursive Implementation

- The algorithm `removeFromRoot`

Algorithm **removeFromRoot(rootNode)**
// Removes the entry in a given root node of a subtree.
if (rootNode *has two children*)
{
    largestNode = *node with the largest entry in the left subtree of* rootNode
    Replace the entry in rootNode *with the entry in* largestNode
    Remove largestNode *from the tree*
}
else if (rootNode *has a right child*)
    rootNode = rootNode*'s right child*
else
    rootNode = rootNode*'s left child // Possibly* null
    // Assertion: If rootNode *was a leaf, it is now* null
return rootNode

Recursive Implementation - removeFromRoot()

```
/**
 * Removes the entry in a given root node of a subtree.
 * @param rootNode root of subtree
 * @return root node of of the revised subtree
 */
private BinaryNode<T> removeFromRoot(BinaryNode<T> rootNode) {
     // Case 1: rootNode has two children
     if (rootNode.hasLeftChild() && rootNode.hasRightChild()) {
          // Find node with largest entry in left subtree
          BinaryNode<T> leftSubtreeRoot = rootNode.getLeftChild();
          BinaryNode<T> largestNode = findLargest(leftSubtreeRoot);

          // Replace entry in root
          rootNode.setData(largestNode.getData());

          // Remove node with largest entry in left subtree
          rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
     } // end if

     // Case 2: rootNode has at most one child
     else if (rootNode.hasRightChild())
          rootNode = rootNode.getRightChild();
     else
          rootNode = rootNode.getLeftChild();

     // Assertion: If rootNode was a leaf, it is now null

     return rootNode;
}
```

Recursive Implementation - findLargest()

```
/**
 * Finds the node containing the largest entry in a given tree.
 * @param rootNode root node of tree
 * @return node containing largest entry in a given tree
 */
private BinaryNode<T> findLargest(BinaryNode<T> rootNode) {
    if (rootNode.hasRightChild())
        rootNode = findLargest(rootNode.getRightChild());

    return rootNode;
}
```

Recursive Implementation - removeLargest()

```
/**
 * Removes the node containing the largest entry in a given tree.
 * @param rootNode root node of tree
 * @return root node of revised tree
 */
private BinaryNode<T> removeLargest(BinaryNode<T> rootNode) {
        if (rootNode.hasRightChild()) {
                BinaryNode<T> rightChild = rootNode.getRightChild();
                rightChild = removeLargest(rightChild);
                rootNode.setRightChild(rightChild);
        } else
                rootNode = rootNode.getLeftChild();

        return rootNode;
}
```

Iterative Implementation

- Pseudocode that describes `remove`

```
Algorithm remove(anEntry)
result = null
currentNode = node that contains a match for anEntry
parentNode  =  currentNode's parent
if (currentNode != null) // That is, if entry is found
{
    result = currentNode's data (the anEntry to be removed from the tree)
    // Case 1
    if (currentNode has two children)
    {
            // Get node to remove and its parent
            nodeToRemove = node containing anEntry inorder predecessor; it has at most one child
            parentNode = nodeToRemove's parent
            Copy entry from nodeToRemove to currentNode
            currentNode = nodeToRemove
            // Assertion: currentNode is the node to be removed; it has at most one child
            // Assertion: Case 1 has been transformed to Case 2
    }
    //Case 2: currentNode has at most one child
    Delete currentNode from the tree
}
return result
```

Iterative Implementation – remove()

```
public T remove(T entry) {
    T result = null;
    // Locate node (and its parent) that contains a match for entry
    NodePair pair = findNode(entry);
    BinaryNode<T> currentNode = pair.getFirst();
    BinaryNode<T> parentNode = pair.getSecond();

    if (currentNode != null) // Entry is found
    {
        result = currentNode.getData(); // Get entry to be removed

        // Case 1: currentNode has two children
        if (currentNode.hasLeftChild() && currentNode.hasRightChild()) {
            // Replace entry in currentNode with the entry in another node
            // that has at most one child; that node can be deleted

            // Get node to remove (contains inorder predecessor; has at
            // most one child) and its parent
            pair = getNodeToRemove(currentNode);
            BinaryNode<T> nodeToRemove = pair.getFirst();
            parentNode = pair.getSecond();

            // Copy entry from nodeToRemove to currentNode
            currentNode.setData(nodeToRemove.getData());

            currentNode = nodeToRemove;
            // Assertion: currentNode is the node to be removed; it has at
            // most one child
            // Assertion: Case 1 has been transformed to Case 2
        } // end if

        // Case 2: currentNode has at most one child; delete it
        removeNode(currentNode, parentNode);
    }
    return result;
}
```

Class NodePair

- Used to hold two nodes, usually current node and parent

```
/**
 * Class that holds two nodes, typically a node and its parent
 *
 */
private class NodePair {
     private BinaryNode<T> first, second;

     public NodePair() {
          first = null;
          second = null;
     } // end default constructor

     public NodePair(BinaryNode<T> firstNode, BinaryNode<T> secondNode) {
          first = firstNode;
          second = secondNode;
     } // end constructor

     public BinaryNode<T> getFirst() {
          return first;
     } // end getFirst

     public BinaryNode<T> getSecond() {
          return second;
     } // end getSecond
}
```

Iterative Implementation – findNode()

```
/**
 * Locate node that contains a match for entry
 * @param entry
 * @return node that matches and parent
 */
private NodePair findNode(T entry) {
      NodePair result = new NodePair();
      boolean found = false;

      BinaryNode<T> currentNode = getRootNode();
      BinaryNode<T> parentNode = null;

      while (!found && (currentNode != null)) {
            T currentEntry = currentNode.getData();
            int comparison = entry.compareTo(currentEntry);

            if (comparison < 0) {
                  parentNode = currentNode;
                  currentNode = currentNode.getLeftChild();
            } else if (comparison > 0) {
                  parentNode = currentNode;
                  currentNode = currentNode.getRightChild();
            } else // comparison == 0
                  found = true;
      } // end while

      if (found)
            result = new NodePair(currentNode, parentNode);
      // Located entry is currentNode.getData()

      return result;
}
```

Iterative Implementation

- Pseudocode for the private method `getNodeToRemove`

```
// Find the in-order predecessor by searching the left subtree; it will be the largest
// entry in the subtree, occurring in the node as far right as possible
leftSubtreeRoot = left child of currentNode
rightChild = leftSubtreeRoot
priorNode = currentNode
while (rightChild has a right child)
{
      priorNode = rightChild
      rightChild = right child of rightChild
}
// Assertion: rightChild is the node to be removed and has no more than one child
```

Iterative Implementation – getNodeToRemove()

```
/**
 * Gets the node that contains the inorder predecessor of currentNode.
 * Assumes node has two children
 * @param currentNode
 * @return
 */
private NodePair getNodeToRemove(BinaryNode<T> currentNode) {
     // Find node with largest entry in left subtree by
     // moving as far right in the subtree as possible
     BinaryNode<T> leftSubtreeRoot = currentNode.getLeftChild();
     BinaryNode<T> rightChild = leftSubtreeRoot;
     BinaryNode<T> priorNode = currentNode;

     while (rightChild.hasRightChild()) {
          priorNode = rightChild;
          rightChild = rightChild.getRightChild();
     } // end while

     // rightChild contains the inorder predecessor and is the node to
     // remove; priorNode is its parent

     return new NodePair(rightChild, priorNode);
}
```

Iterative Implementation – removeNode()

```
/**
 * Removes a node having at most one child.
 * @param nodeToRemove
 * @param parentNode
 */
private void removeNode(BinaryNode<T> nodeToRemove, BinaryNode<T> parentNode) {
    BinaryNode<T> childNode;

    if (nodeToRemove.hasLeftChild())
        childNode = nodeToRemove.getLeftChild();
    else
        childNode = nodeToRemove.getRightChild();

    // Assertion: if nodeToRemove is a leaf, childNode is null

    if (nodeToRemove == getRootNode())
        setRootNode(childNode);
    else if (parentNode.getLeftChild() == nodeToRemove)
        parentNode.setLeftChild(childNode);
    else
        parentNode.setRightChild(childNode);
}
```

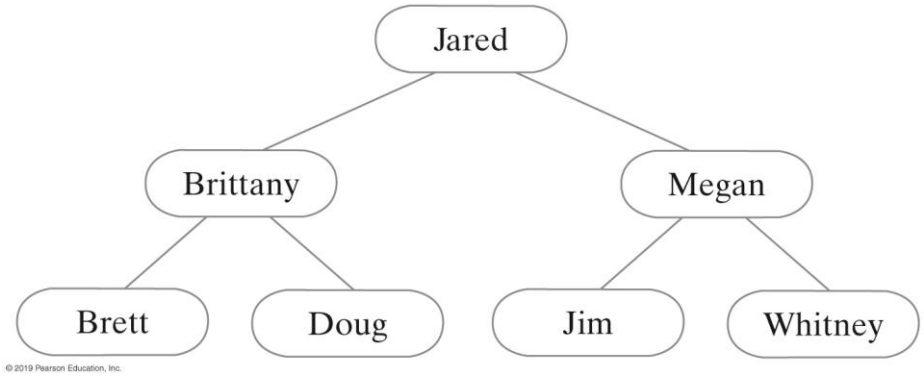
Efficiency of Operations

- For tree of height h
  - The operations add, remove, and **`getEntry`** are $O(h)$
- If tree of n nodes has height $h = n$
  - These operations are $O(n)$
- Shortest tree is full
  - Results in these operations being $O(\log n)$

Efficiency of Operations

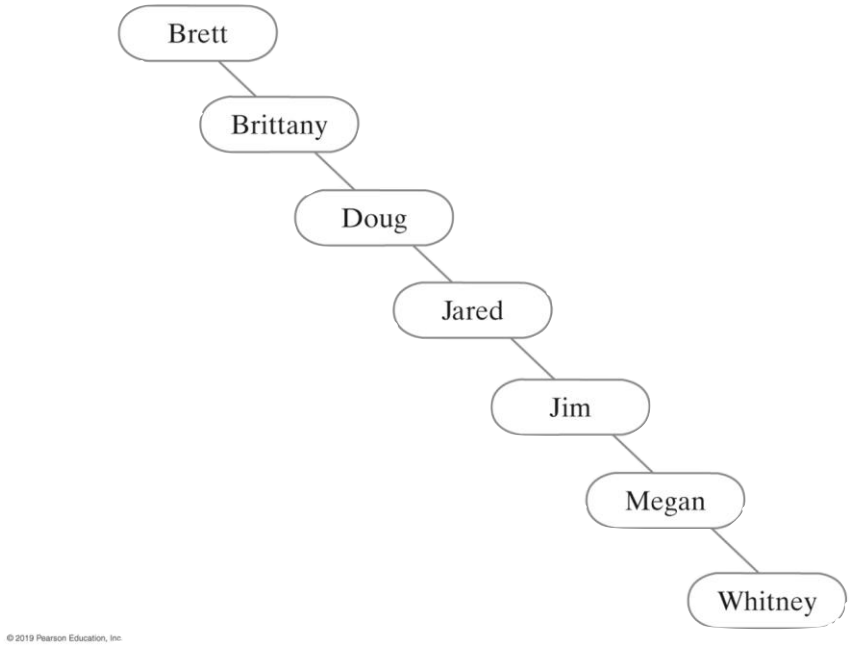
- FIGURE 26-13 Two binary search trees that contain the same data

(a) The shortest binary search tree having seven nodes



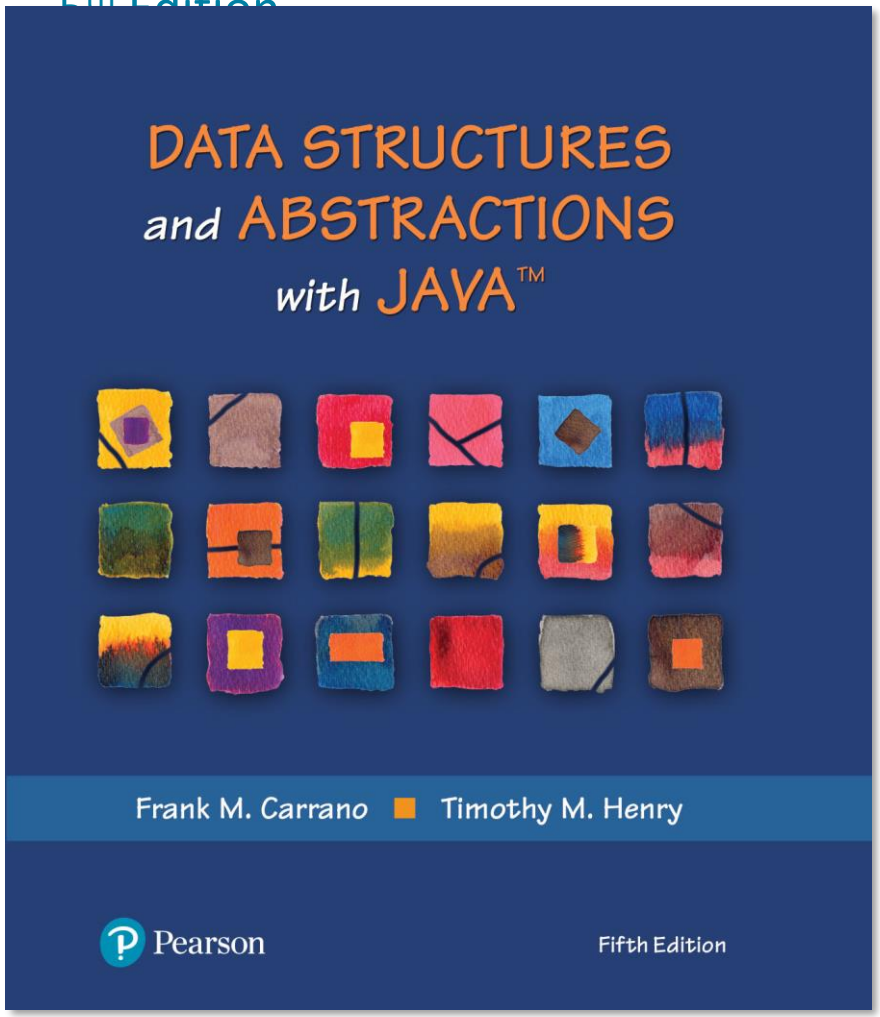
Operations are $O(\log n)$

(b) The tallest binary search tree having seven nodes



Operations are $O(n)$

Data Structures and Abstractions with Java™



Chapter 27

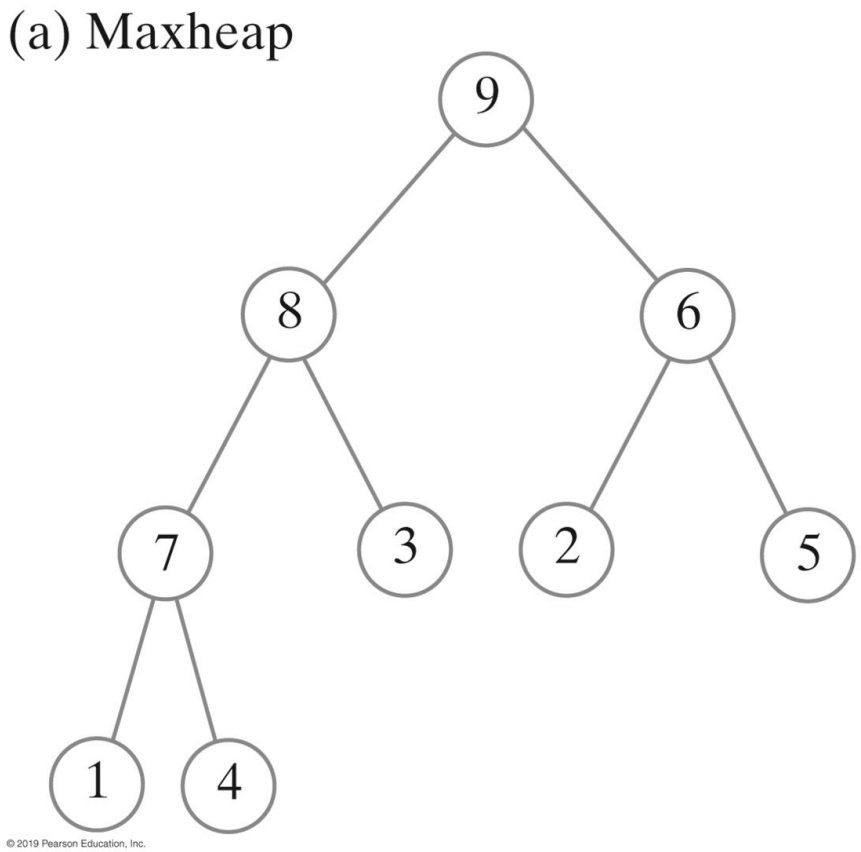
A Heap Implementation

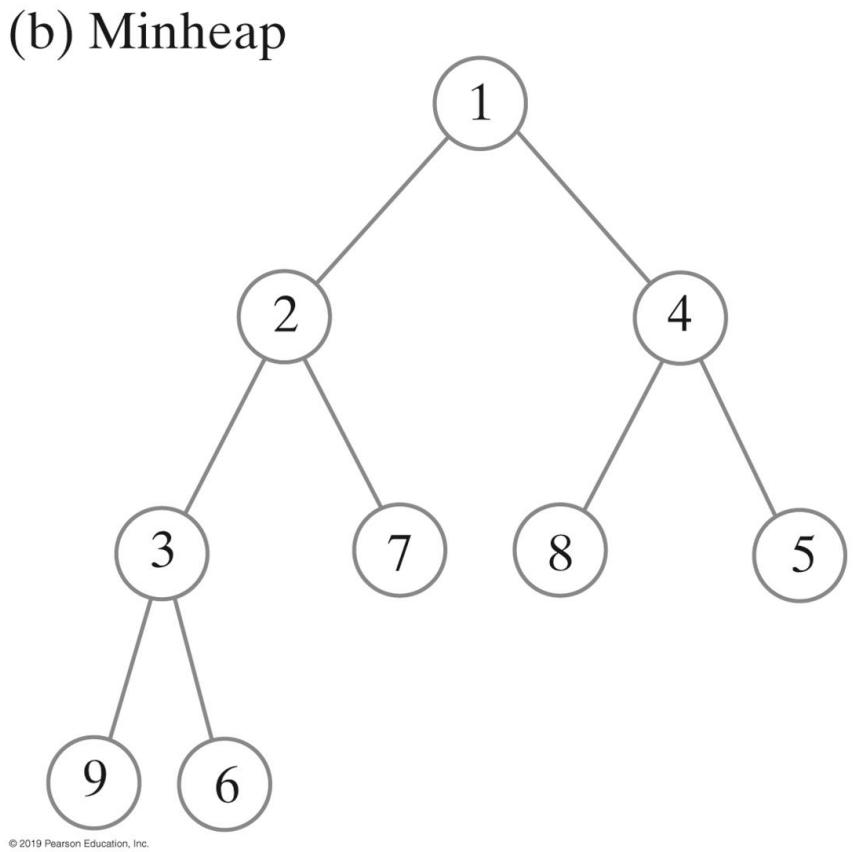
Heaps

- 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 – an interface for MaxHeap

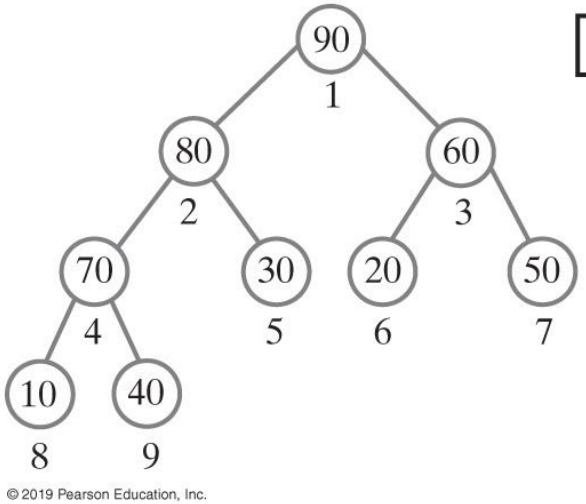
```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   /** Adds a new entry to this heap.
       @param newEntry  An object to be added. */
   public void add(T newEntry);

   /** 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();
}
```

Heap and Maxheap

- Heap
  - ***Complete binary tree*** whose nodes contain Comparable objects
- Maxheap
  - Object in each node is greater than or equal to the objects in the node's descendants

An Array to Represent a Heap

- Use an array to represent a complete binary tree
- Number nodes in the order in which a level-order traversal would visit them
- Can locate either the children or the parent of any node
  - Perform a simple computation on the node's number



	90	80	60	70	30	20	50	10	40			
0	1	2	3	4	5	6	7	8	9	10	11	12

left = currnet_left X 2

right = current_right X 2 + 1

FIGURE 27-1 A complete binary tree with its nodes numbered in level order and its representation as an array

MaxHeap constructor

- Uses an array with element 0 unused.
- Makes it easy to calculate parents and children.

```
public final class CompletedMaxHeap<T extends Comparable<? super T>> implements MaxHeapInterface<T> {
     private T[] heap; // Array of heap entries; ignore heap[0]
     private int lastIndex; // Index of last entry and number of entries
     private static final int FRONT = 1;  // the first element in the heap

     private static final int DEFAULT_CAPACITY = 5; // NB: Changed to 5 from 25 for testing convenience
     private static final int MAX_CAPACITY = 10000;


     public CompletedMaxHeap() {
          this(DEFAULT_CAPACITY); // Call next constructor
     } // end default constructor

     public CompletedMaxHeap(int initialCapacity) {
          // NOTE: This code deviates from the book somewhat in that checkCapacity throws
          // an exception
          // if initialCapacity is either too small or too large.

          checkCapacity(initialCapacity);

          // The cast is safe because the new array contains null entries
          @SuppressWarnings("unchecked")
          T[] tempHeap = (T[]) new Comparable[initialCapacity + FRONT]; // First array element is not used
          heap = tempHeap;
          lastIndex = 0;

     }
```

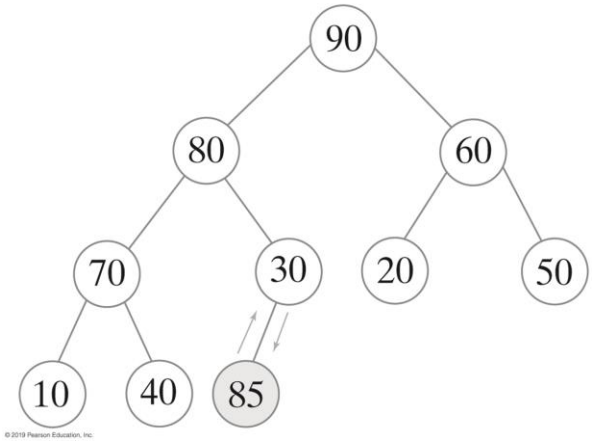
MaxHeap – standard methods

```
public T getMax() {
     T root = null;

     if (!isEmpty())
          root = heap[1];

     return root;
} // end getMax

public boolean isEmpty() {
     return lastIndex < 1;
}

public int getSize() {
     return lastIndex;
}

public void clear() {
     while (lastIndex > -1) {
          heap[lastIndex] = null;
          lastIndex--;
     }

     lastIndex = 0;
}
```

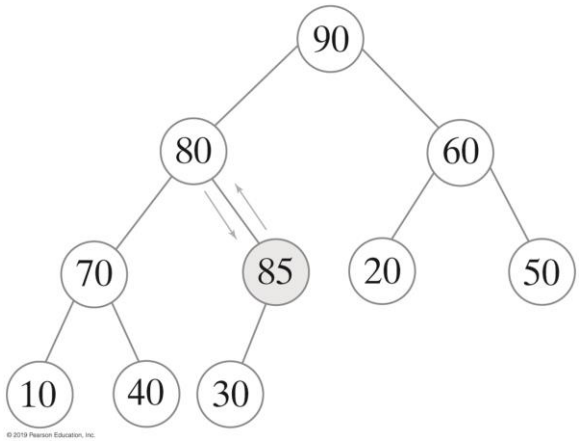
Adding an Entry

- FIGURE 27-2 The steps in adding 85 to the maxheap in Figure 27-1a

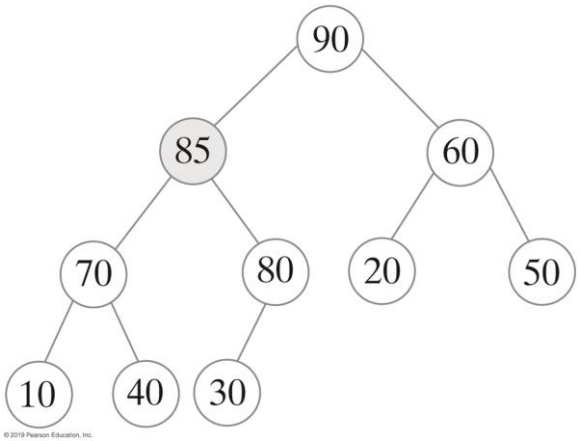
(a) Add 85 as the next leaf.
Then swap it with its parent, 30



(b) Swap 85 with its parent, 80



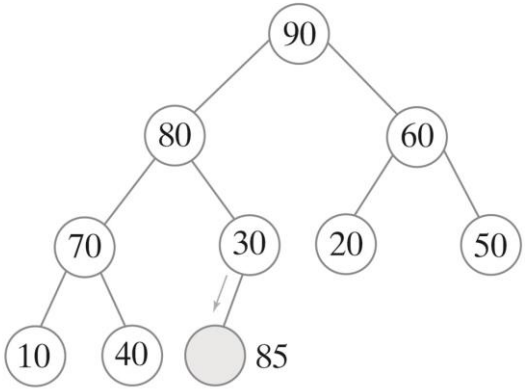
(c) The result is a max heap



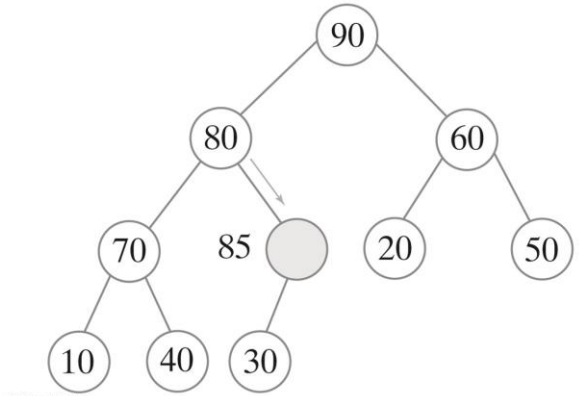
Adding an Entry without Swaps

- FIGURE 27-3 A revision of the steps to add 85, as shown in Figure 27-2, to avoid swaps

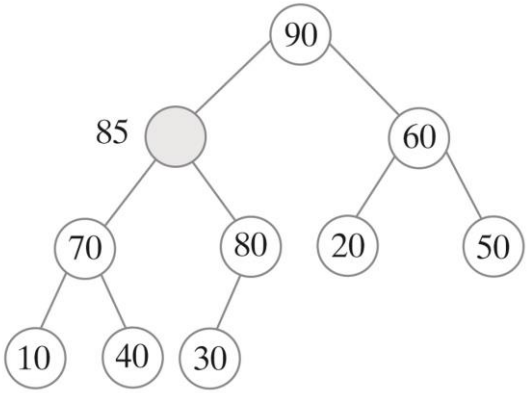
(a) Identify the location of a new leaf. 85 is larger than 30, which is in this leaf's parent, so move 30 to the new leaf



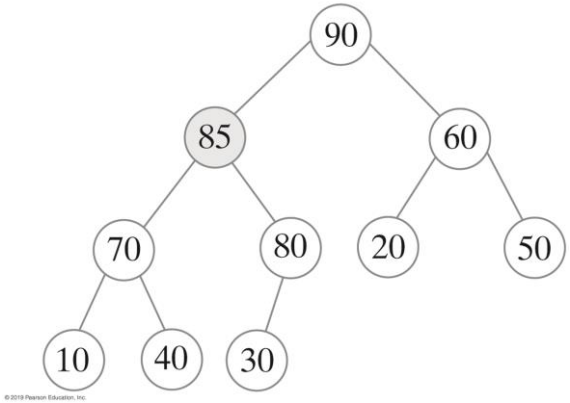
(b) 85 is larger than 80, which is in the empty node's parent, so move 80 to the empty node



(c) 85 is less than 90, which is in the empty node's parent, so place 85 into the empty node

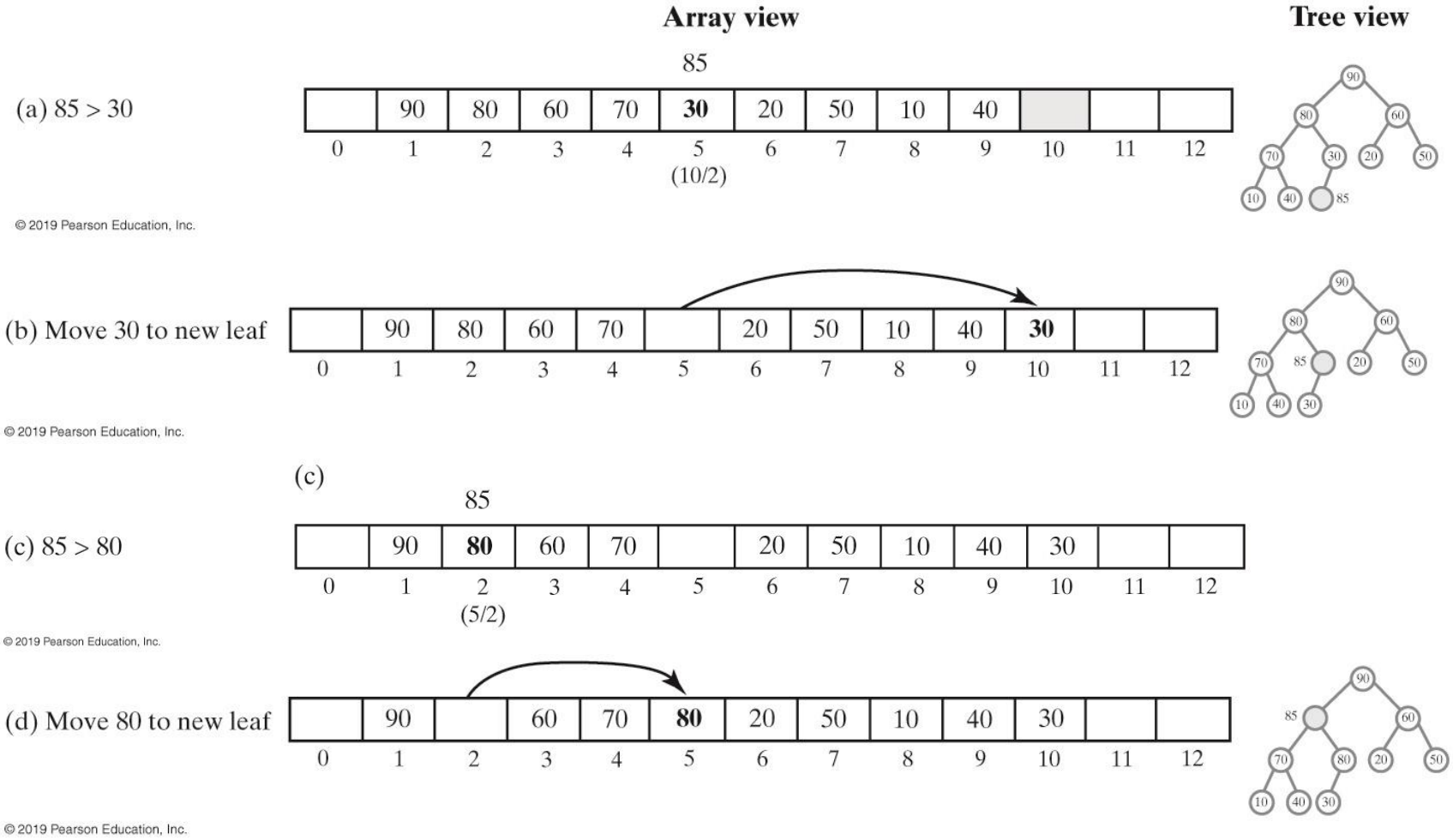


(d) The result is a maxheap



Adding an Entry to Heap (Part 1)

- FIGURE 27-4 An array representation of the steps in Figure 27-3

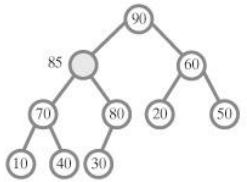


Adding an Entry to Heap (Part 2)

- FIGURE 27-4 An array representation of the steps in Figure 27-3

(d) Move 80 to new leaf

	90		60	70	**80**	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12



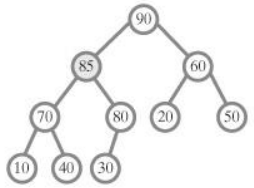
(e) 85 < 90

85

	90		60	70	80	20	50	10	40	30		
0	1 (2/2)	2	3	4	5	6	7	8	9	10	11	12

(f) Insert 85 into vacancy

	90	**85**	60	70	80	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12



Adding an Entry

- Algorithm to add a new entry to a heap

```
Algorithm add(newEntry)
// Precondition: The array heap has room for another entry.
newIndex = index of next available array location
parentIndex = newIndex/2  //Index of parent of available location
while (parentIndex > 0 and newEntry > heap[parentIndex])
{
      heap[newIndex] = heap[parentIndex] // Move parent to available location
      // Update indices
      newIndex = parentIndex parentIndex = newIndex/2
}
heap[newIndex] = newEntry                    // Place new entry in correct location
if (the array heap is full)
      Double the size of the array
```

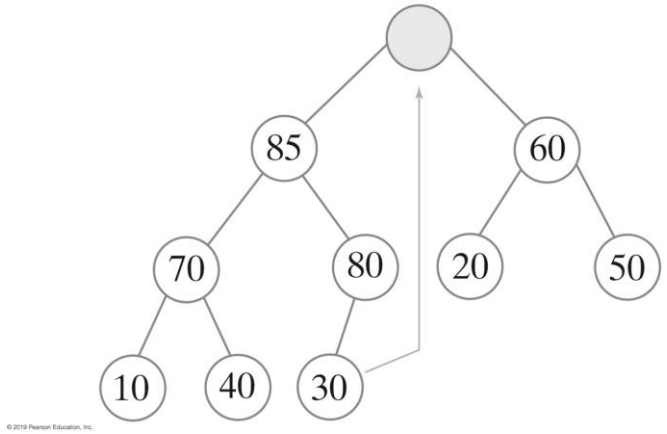
Adding an entry

```
public void add(T newEntry) {
    int newIndex = lastIndex + 1;  // start at the last index of the tree
    int parentIndex = newIndex / 2;  // parent is always halfway up

    while ((parentIndex > 0) && newEntry.compareTo(heap[parentIndex]) > 0) {
        // newEntry is larger than the parent, so move the parent to the leaf node or vacancy

        heap[newIndex] = heap[parentIndex];

        // vacancy is now where the parent previously existed

        newIndex = parentIndex;

        // go to the next parent up the tree

        parentIndex = newIndex / 2;
    } // end while

    // place the entry in the vacancy/leaf and bump the last index (last node of the tree)
    heap[newIndex] = newEntry;
    lastIndex++;

    ensureCapacity();
}
```

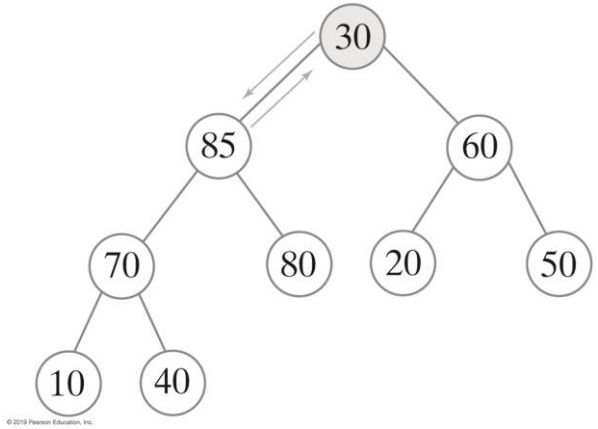
Removing a Value from A Heap

- FIGURE 27-5 The steps to remove the entry in the root of the maxheap in Figure 27-3d

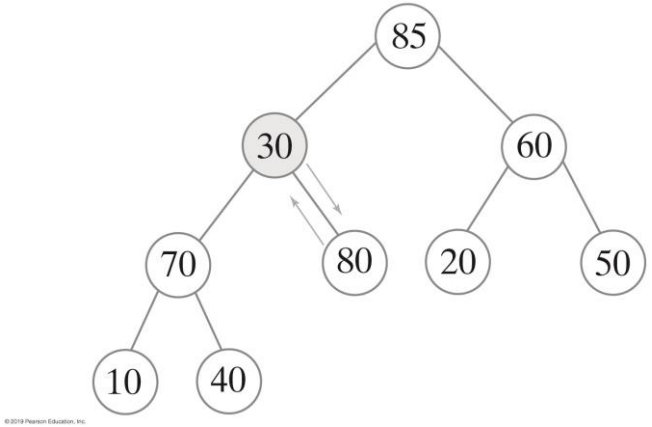
(a) Replace the root's entry with the last leaf's data



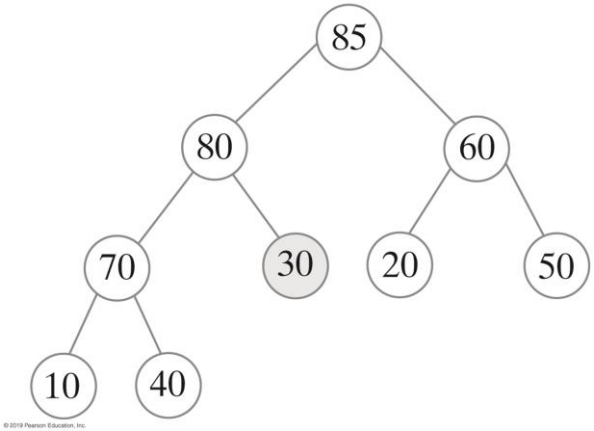
(b) Delete the last leaf; swap 30 with its largest child, 85



(c) Swap 30 with its largest child, 80



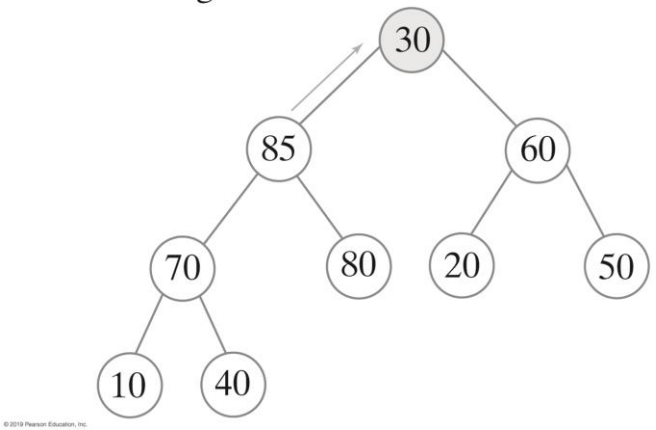
(d) The result is a maxheap



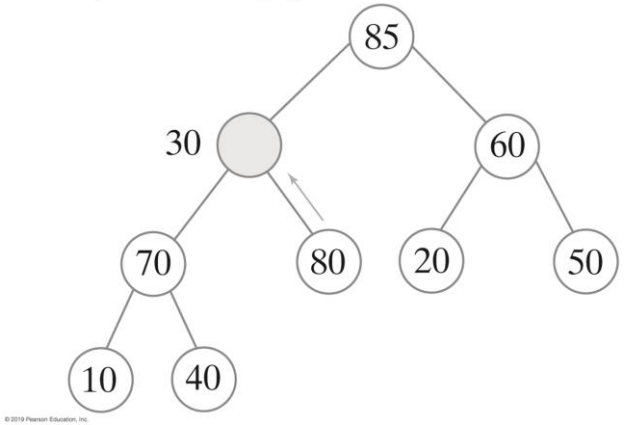
Removing a Value without Swaps

- FIGURE 27-6 The steps to transform the semiheap in Figure 27-5b into a heap without using swaps

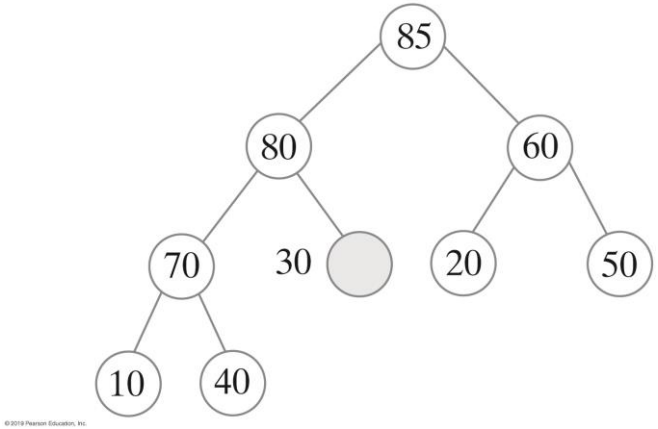
(a) Copy 30 and replace it with the root's largest child



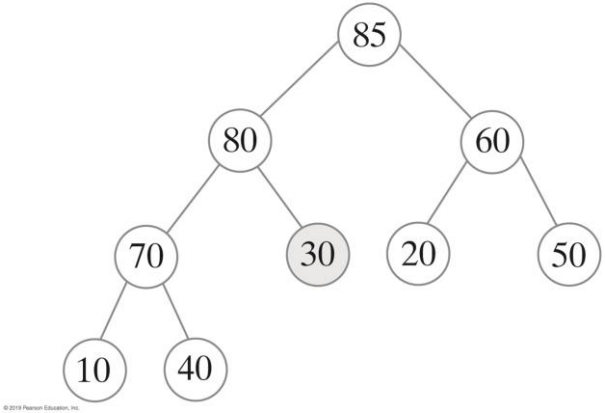
(b) Move the empty node's larger child, 80, to the empty node



(c) Place 30 into the vacant leaf



(d) The result is a maxheap



Removing the Root

- Algorithm to transform a semiheap to a heap

```
Algorithm reheap(rootIndex)
// Transforms the semiheap rooted at rootIndex into a heap
done = false
orphan = heap[rootIndex]
while (!done and heap[rootIndex] has a child)
{
  largerChildIndex = index of the larger child of heap[rootIndex]
  if (orphan < heap[largerChildIndex])
  {
    heap[rootIndex] = heap[largerChildIndex]
    rootIndex = largerChildIndex
  }
  else
    done = true
}
heap[rootIndex] = orphan
```

Removing the root – reheap()

```
/**
 * Transform a semiheap into a heap.
 * @param rootIndex
 */
private void reheap(int rootIndex) {
	boolean done = false;

	// this is the starting point. we are not sure where this will end up,
	// so it is an orphan to start

	T orphan = heap[rootIndex];

	// left child is always double the index of the parent
	// right child is one more than the left

	int leftChildIndex = 2 * rootIndex;

	// iterate through the tree until the orphan is the larger one

	while (!done && (leftChildIndex <= lastIndex)) {

		// larger child is always the left one
		int largerChildIndex = leftChildIndex;

		// right child is one position more than the left

		int rightChildIndex = leftChildIndex + 1;

		// set the larger child to the right child if the right child is larger

		if ((rightChildIndex <= lastIndex) && heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0) {
			largerChildIndex = rightChildIndex;
		}
		// now compare our orphan to the largest child.
		// if the orphan is smaller, move the larger child to the root,
		// and adjust the rootIndex to be the larger child
		// then look at the next left child and start over

		// if orphan is the same or larger, then we are done

		if (orphan.compareTo(heap[largerChildIndex]) < 0) {
			heap[rootIndex] = heap[largerChildIndex];
			rootIndex = largerChildIndex;
			leftChildIndex = 2 * rootIndex;
		} else
			done = true;
	} // end while

	// the rootIndex has been potentially reset to a child's
	// in any case, copy the orphan to the root

	// note that if the orphan was the largest to start, it just remains at the root
	heap[rootIndex] = orphan;
}
```

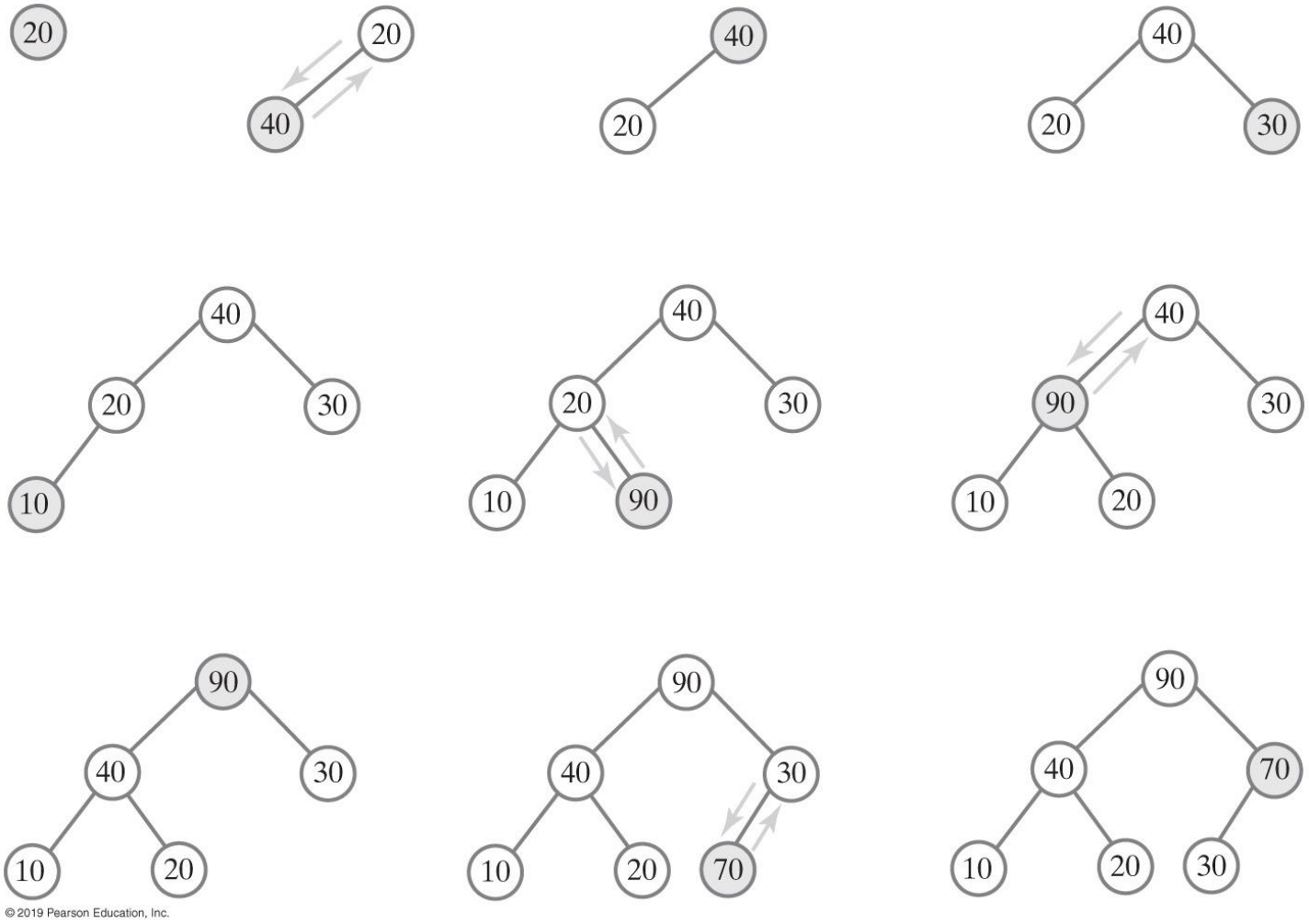
Removing the max via reheap()

- To remove the max, copy the last element to the front, and reheap()
- This will move all of the elements so that a true heap is reestablished.

```
public T removeMax() {

      T root = null;

      if (!isEmpty()) {
            root = heap[FRONT]; // return the front of the heap
            heap[FRONT] = heap[lastIndex]; // Form a semiheap
            lastIndex--; // Decrease size
            reheap(FRONT); // Transform to a heap
      }

      return root;
}
```

Creating a Heap

- FIGURE 27-7 The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap



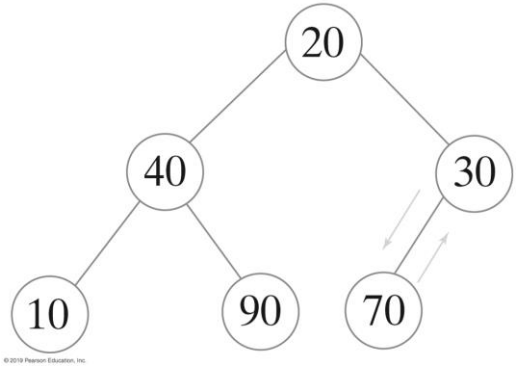
Creating a Heap from an array

- The steps in creating a heap of the entries 20, 40, 30, 10, 90, and 70 by using `reheap`

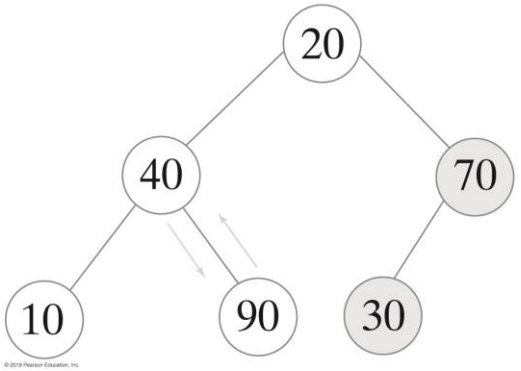
(a) An array of entries

	20	40	30	10	90	70
0	1	2	3	4	5	6

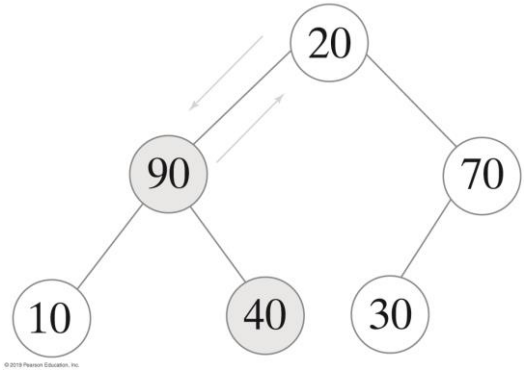
(b) The complete tree that the array represents



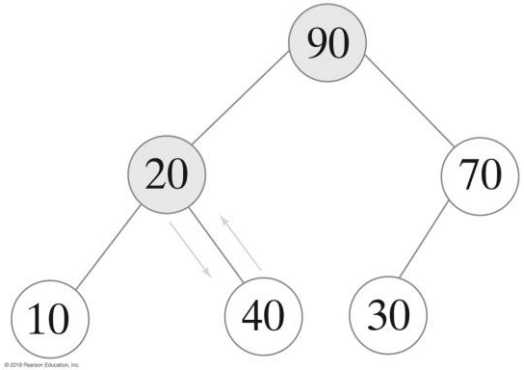
(c) After `reheap(3)`



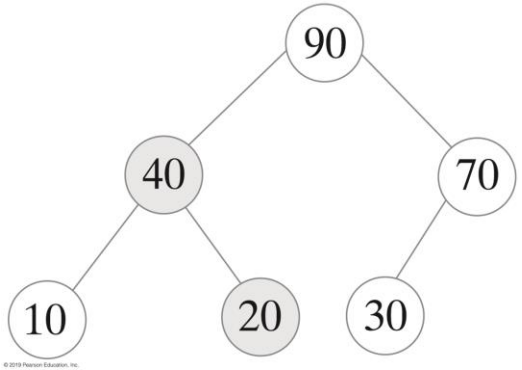
(d) After `reheap(2)`



(e) During `reheap(1)`



(f) After `reheap(1)`

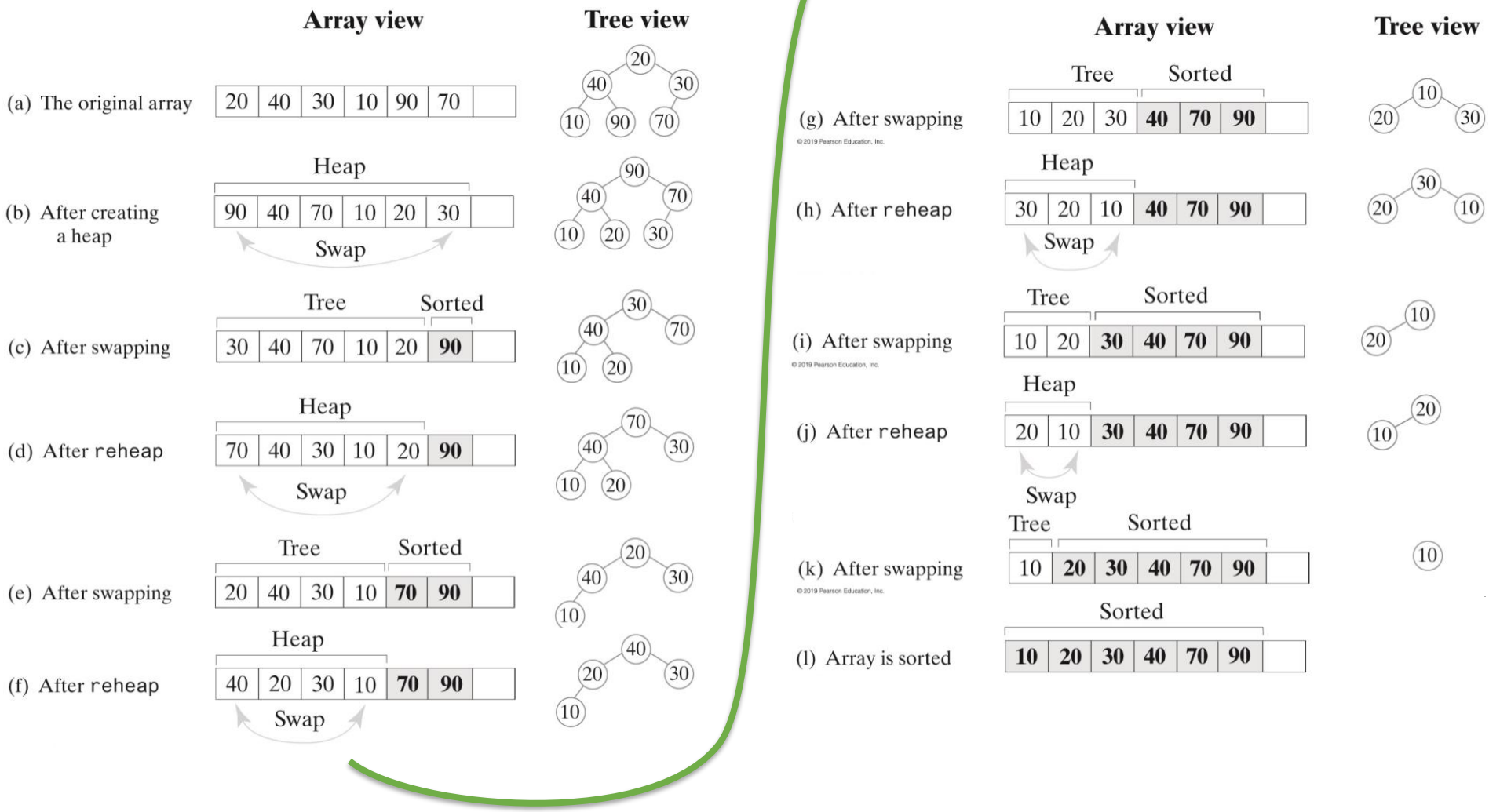


Creating a heap from an array

- Start with the end of the array (last index), and reheap its parent.
- Continue reheap() for each parent as we traverse the tree in level order

```
/**
 * Create a heap from an array of entries by reheap()ing as each is added
 * @param entries
 */
public MaxHeap(T[] entries) {
     this(entries.length); // Call other constructor
     // Assertion: integrityOK = true
     lastIndex = entries.length;

     // Copy given array to data field
     for (int index = FRONT; index <= lastIndex; index++)
          heap[index] = entries[index - 1];

     // Create heap. Start with the parent of the last index.
     for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
          reheap(rootIndex);
}
```

A trace of Heap Sort



Heap Sort using ListInterface – reheap()

- See SortUtilities and SortDemo
- Note use of AList instead of an array

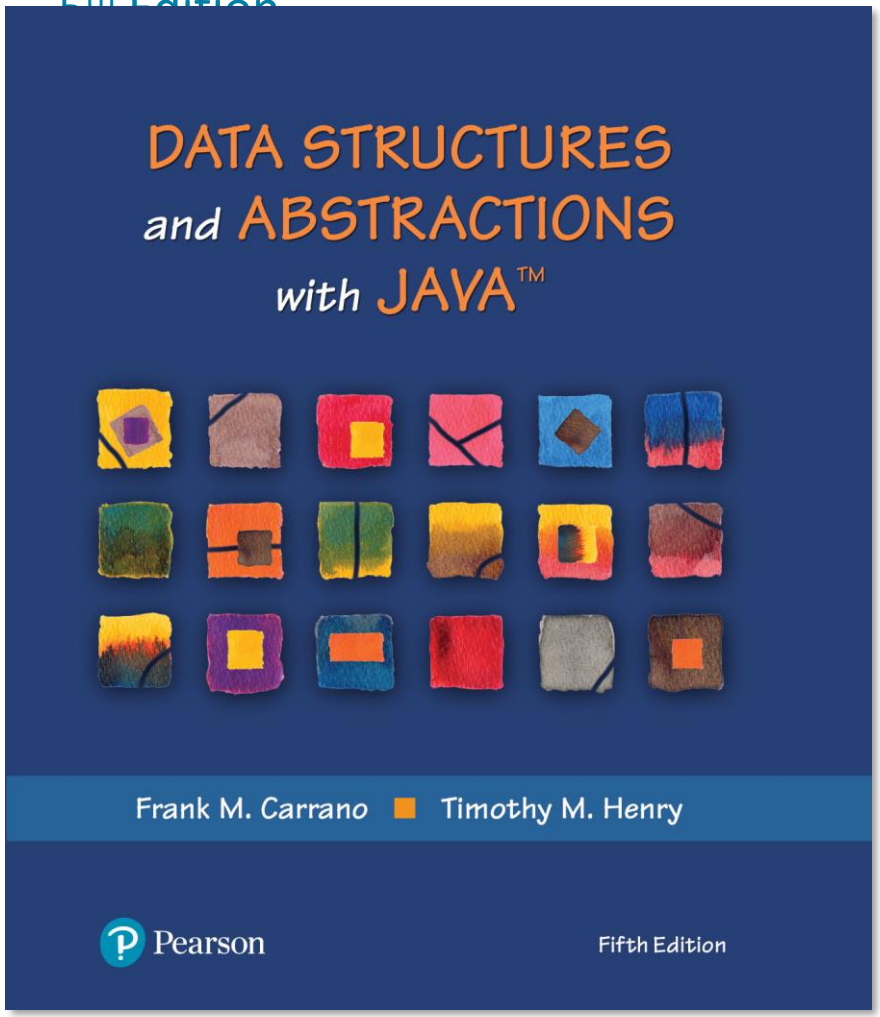
```
static public <T extends Comparable<? super T>> void reheap(ListInterface<T> list, int rootIndex, int lastIndex) {
     boolean done = false;

     T orphan = list.getEntry(rootIndex);
     System.out.println("reheap rootindex " + rootIndex + " last " + lastIndex + " orphan " + orphan);

     // children start at double the root index

     // left child is one more

     int leftChildIndex = (2 * rootIndex) + 1;

     while (!done && (leftChildIndex <= lastIndex)) {

          // assume left child is the larger one

          int largerChildIndex = leftChildIndex;

          // right child is one more than left

          int rightChildIndex = leftChildIndex + 1;

          // get the right child data. left child data is the largest
          T rightChild = list.getEntry(rightChildIndex);
          T largerChild = list.getEntry(largerChildIndex);

          // if right is larger than left, then set larger to right child
          if ((rightChildIndex <= lastIndex) && rightChild.compareTo(largerChild) > 0) {
               largerChildIndex = rightChildIndex;
               largerChild = rightChild;
          }

          // if our current root list smaller, move the larger to the root
          // and then go to its child
          // otherwise we are done
          if (orphan.compareTo(largerChild) < 0) {
               list.replace(rootIndex, largerChild);
               rootIndex = largerChildIndex;
               leftChildIndex = (2 * rootIndex) + 1;
          } else
               done = true;
     }

     // set the current root value to the (possibly redefined) root slot

     list.replace(rootIndex, orphan);
}
```

HeapSort using reheap()

- Heap sort is implemented using swap() and reheap()
- Note use of ListInterface

```
/**
 * Heap sort using reheap to create an initial semiheap.
 * Then swap last element with root and reheap until complete.
 * @param list
 * @param first
 * @param last
 */
static public <T extends Comparable<? super T>> void heapSort(ListInterface<T> list, int first, int last) {
     // Create first heap
     for (int rootIndex = (last / 2); rootIndex >= first; rootIndex--)
          reheap(list, rootIndex, last);

     // exchange the last with the root, and reheap.

     // continue reheap until we have iterated through all of the elements
     // starting with the last element

     swap(list, 0, last);

     for (int lastIndex = last - 1; lastIndex > 0; lastIndex--) {
          reheap(list, 0, lastIndex);
          swap(list, 0, lastIndex);  // swap and reheap
     }
}
```

Data Structures and Abstractions with Java™



Chapter 28

Balanced Search Trees

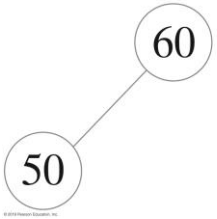
AVL Trees

- Possible to form several differently shaped binary search trees
  - From the same collection of data
- The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and Evgenii Landis
- AVL tree is a binary search tree that
  - Rearranges its nodes whenever it becomes unbalanced.

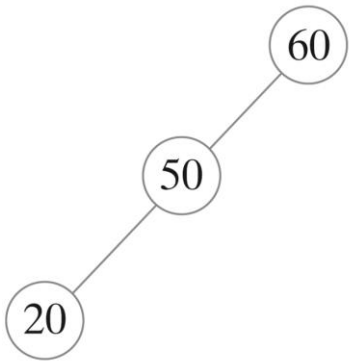
Single AVL Tree Rotations

(a) After adding 60

(b) After adding 50

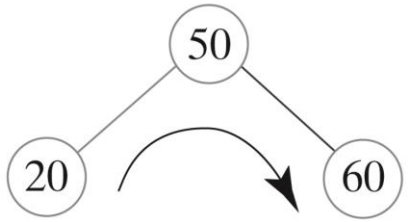


(c) Adding 20 makes the tree unbalanced



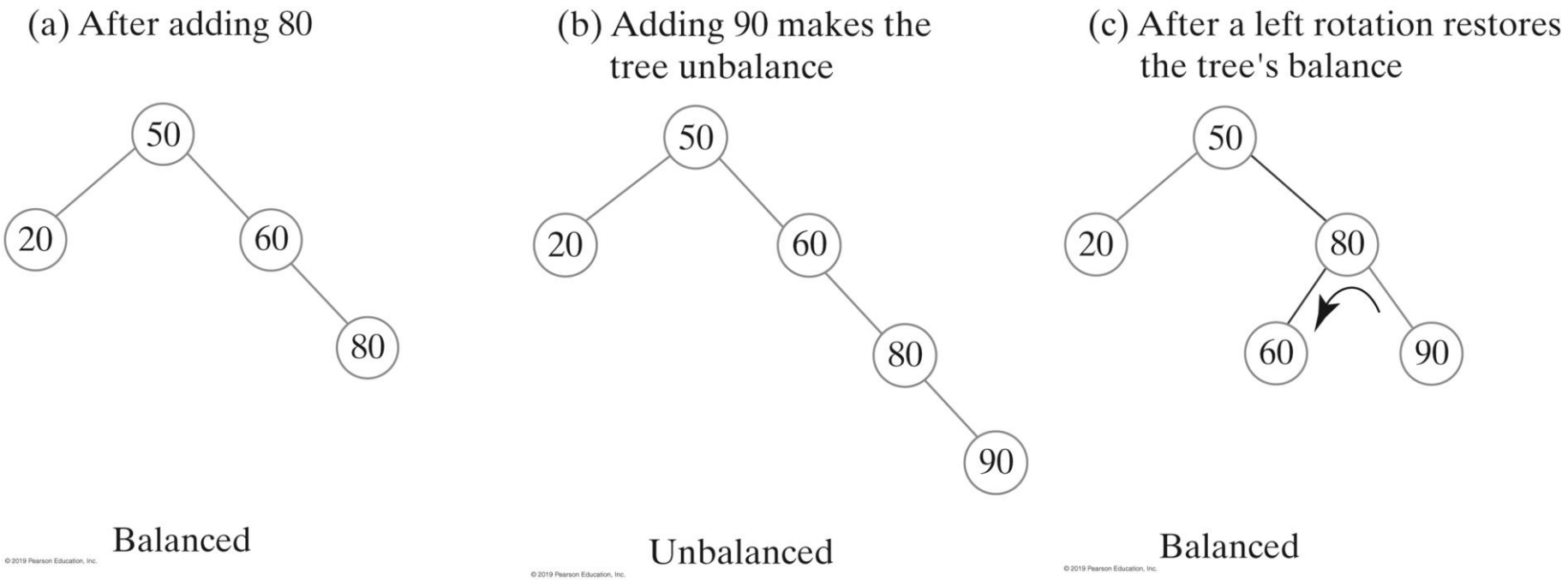
Unbalanced

(d) A rotation restores balance



- Additions to an initially empty AVL tree

FIGURE 28-2 Additions to the AVL tree in Figure 28-1



Single Rotations

- This algorithm performs the right rotation illustrated in Figures 28-3 and 28-4

Algorithm **rotateRight(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
//in the left subtree of nodeN*'s left child.*
nodeC = *left child of* nodeN
Set nodeN*'s left child to* nodeC*'s right child*
Set nodeC*'s right child to* nodeN
return nodeC

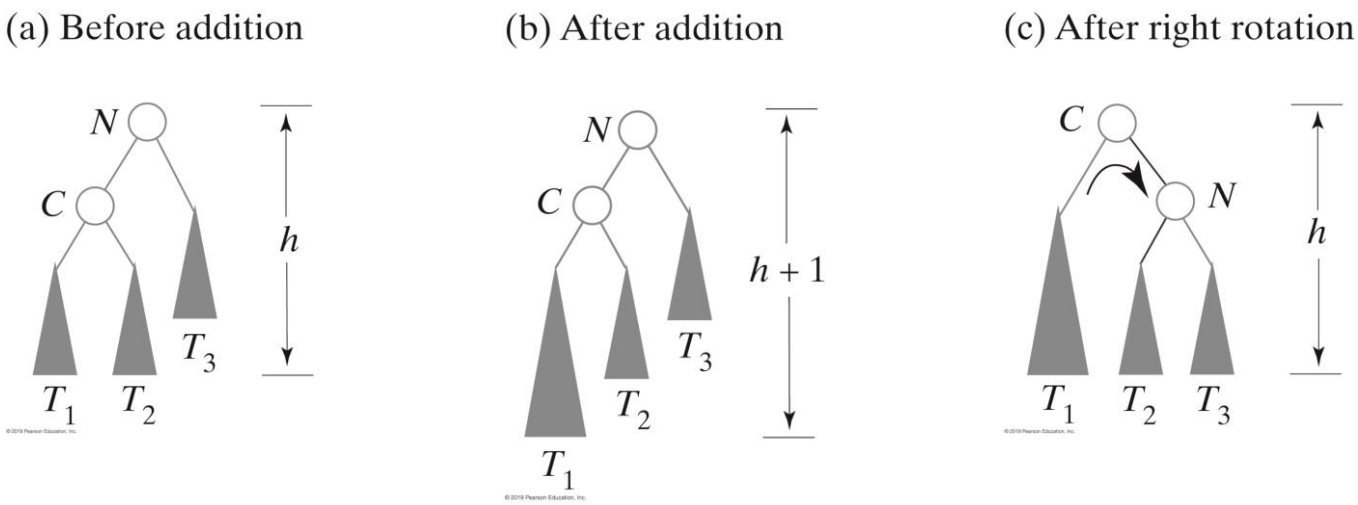
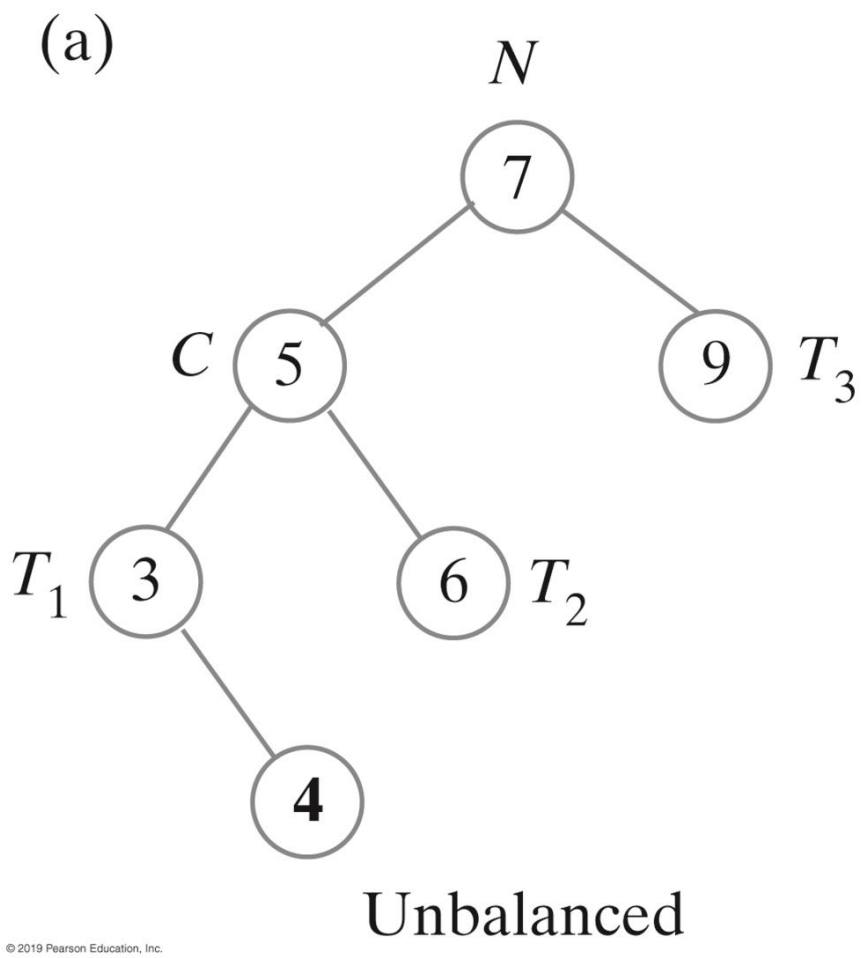
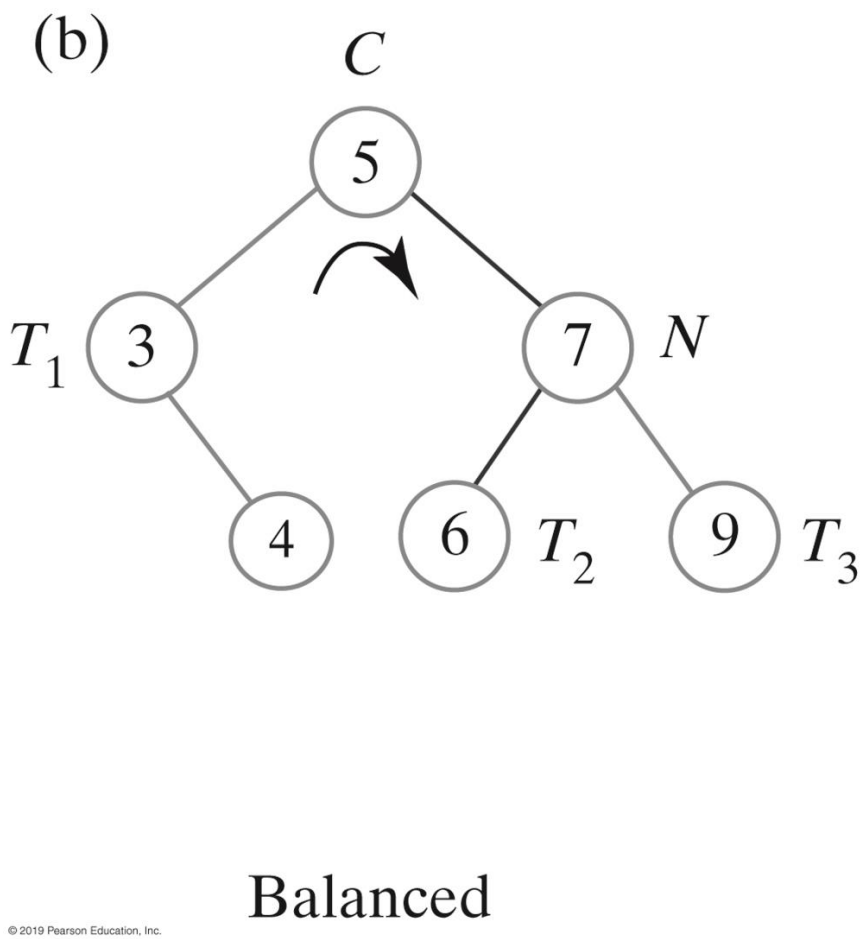


FIGURE 28-3 Before and after an addition to an AVL subtree that requires a right rotation to maintain its balance

Single Rotation

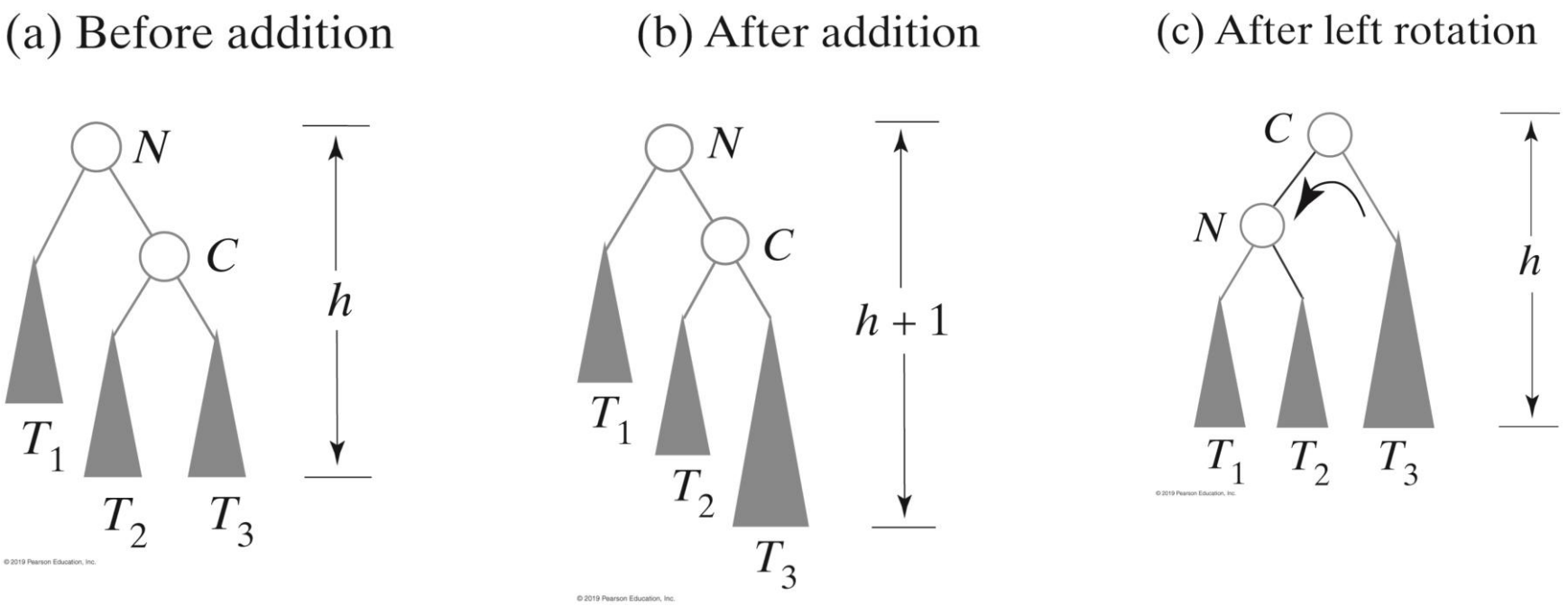
- FIGURE 28-4 Before and after a right rotation restores balance to an AVL tree





Single Rotation

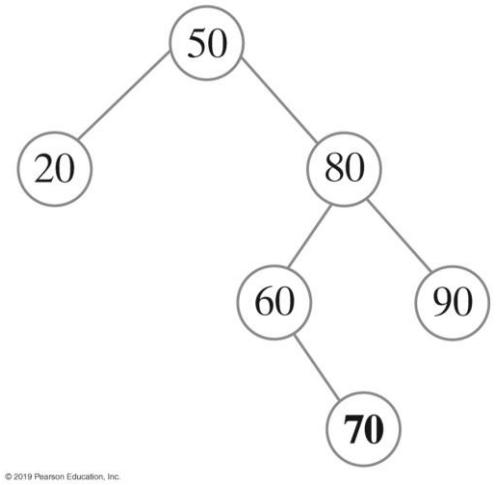
- FIGURE 28-5 Before and after an addition to an AVL subtree that requires a left rotation to maintain its balance



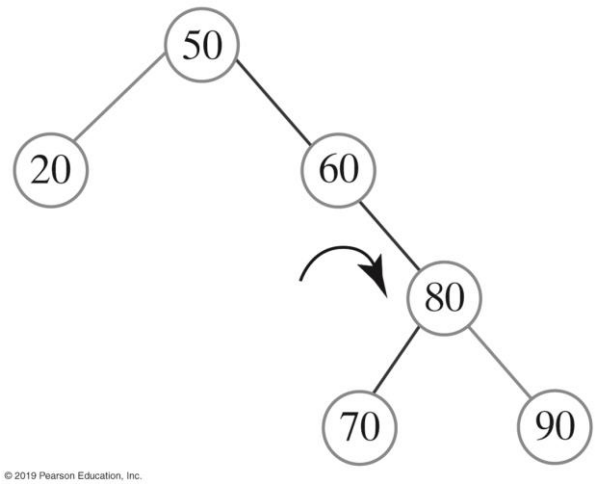
Double Rotation

- FIGURE 28-6 Adding 70 to the AVL tree in Figure 28-2c requires both a right rotation and a left rotation to maintain its balance

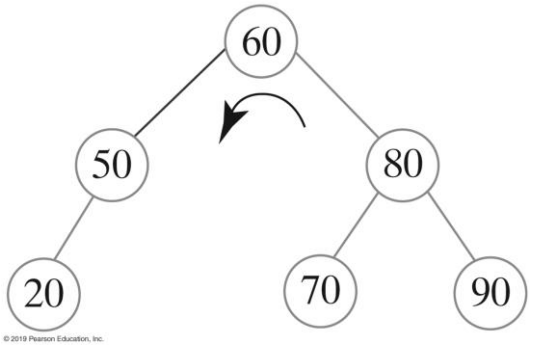
(a) After adding 70



(b) After right rotation



(c) After left rotation



Left-right Double Rotations

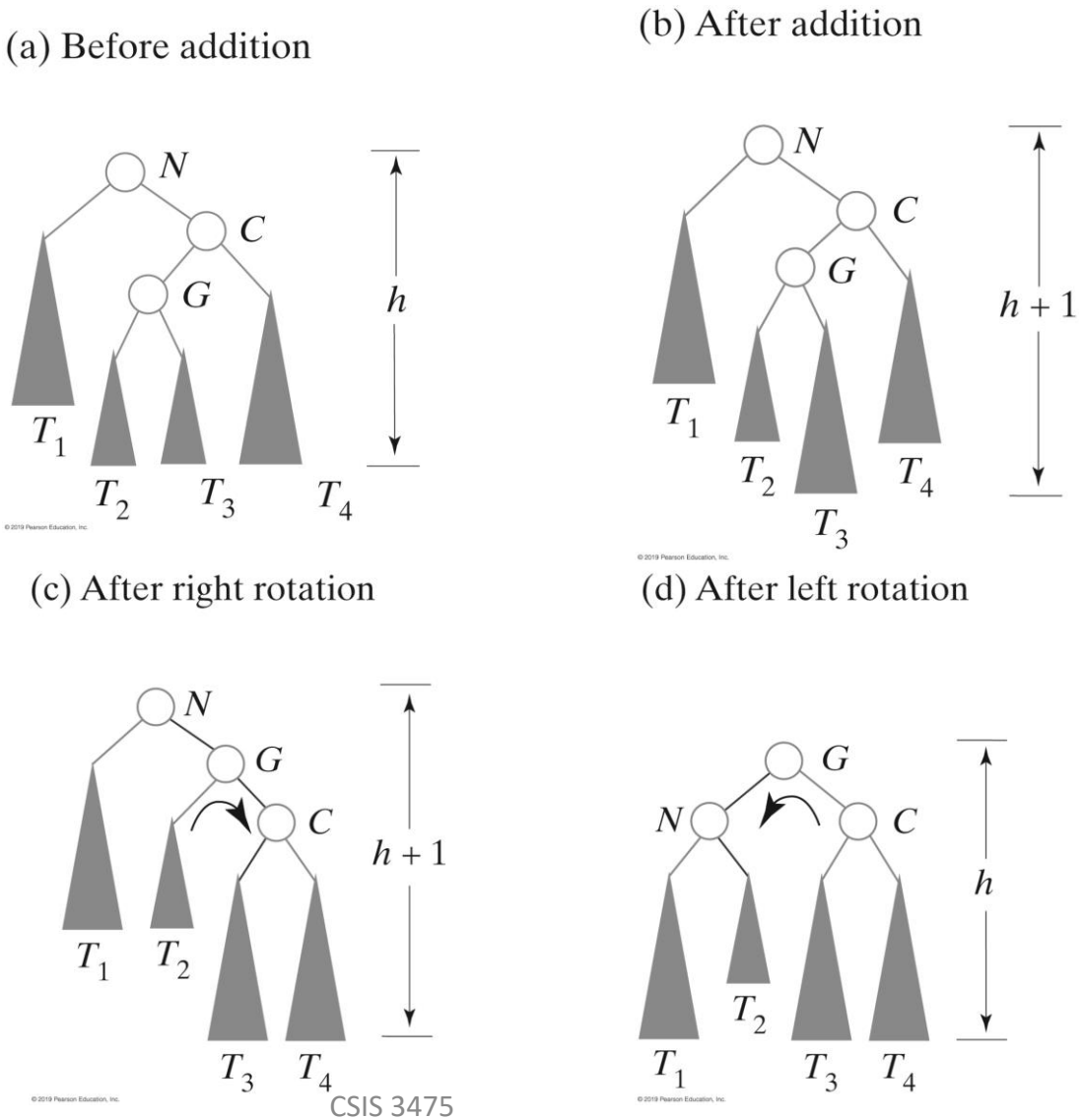
- A double rotation is accomplished by performing two single rotations
- A rotation about node *N*'s grandchild *G* (its child's child)
- A rotation about node *N*'s new child

Left-right Double Rotations

- Four rotations cover the only four possibilities for the cause of the imbalance at node *N*
- The addition occurred in …
  - the left subtree of *N's* left child (right rotation)
  - the right subtree of *N's* left child (left-right rotation)
  - the left subtree of *N's* right child (right-left rotation)
  - the right subtree of *N's* right child (left rotation)

Double Rotations

- FIGURE 28-7 Before and after an addition to an AVL subtree that requires both a right rotation and a left rotation to maintain its balance



Double Rotations

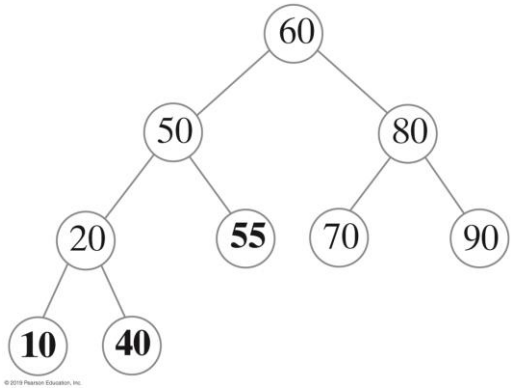
- This algorithm performs the right-left double rotation illustrated in Figure 28-7

Algorithm **rotateRightLeft(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
//in the left subtree of nodeN*'s right child.*
nodeC = *right child of* nodeN
Set nodeN*'s right child to the node returned by* rotateRight(nodeC)
return rotateLeft(nodeN)

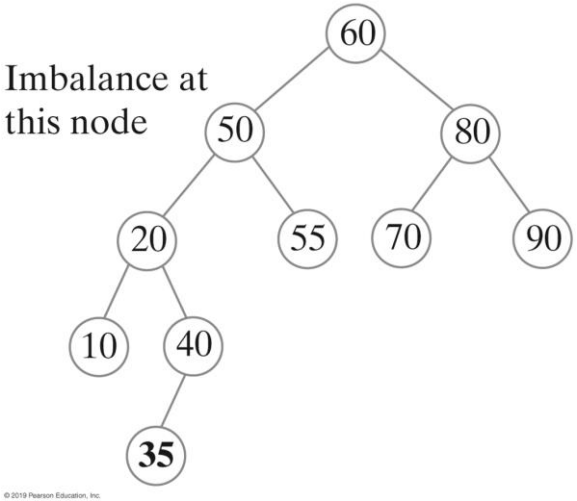
Double Rotations

- FIGURE 28-8 Adding 55, 10, 40, and 35 to the AVL tree in Figure 28-6c

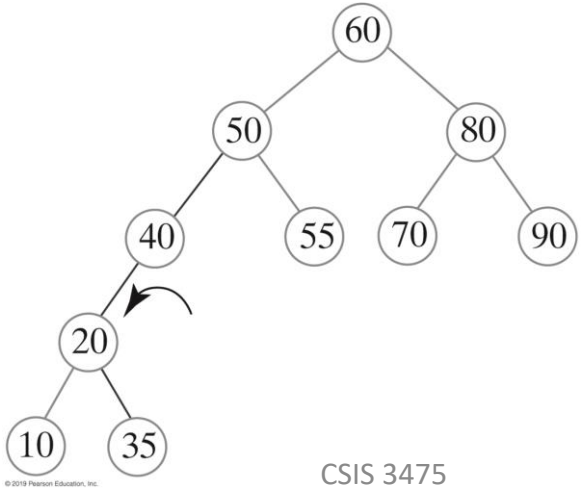
(a) After adding 55, 10, and 40



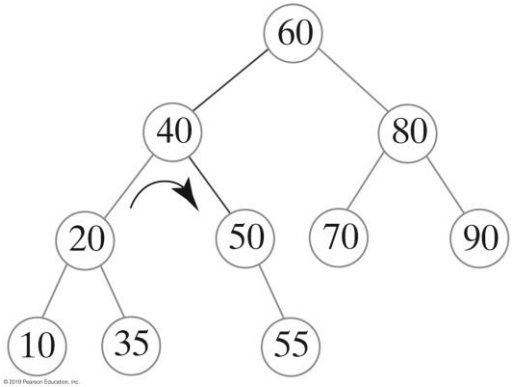
(b) After adding 35



(c) After left rotation about 40

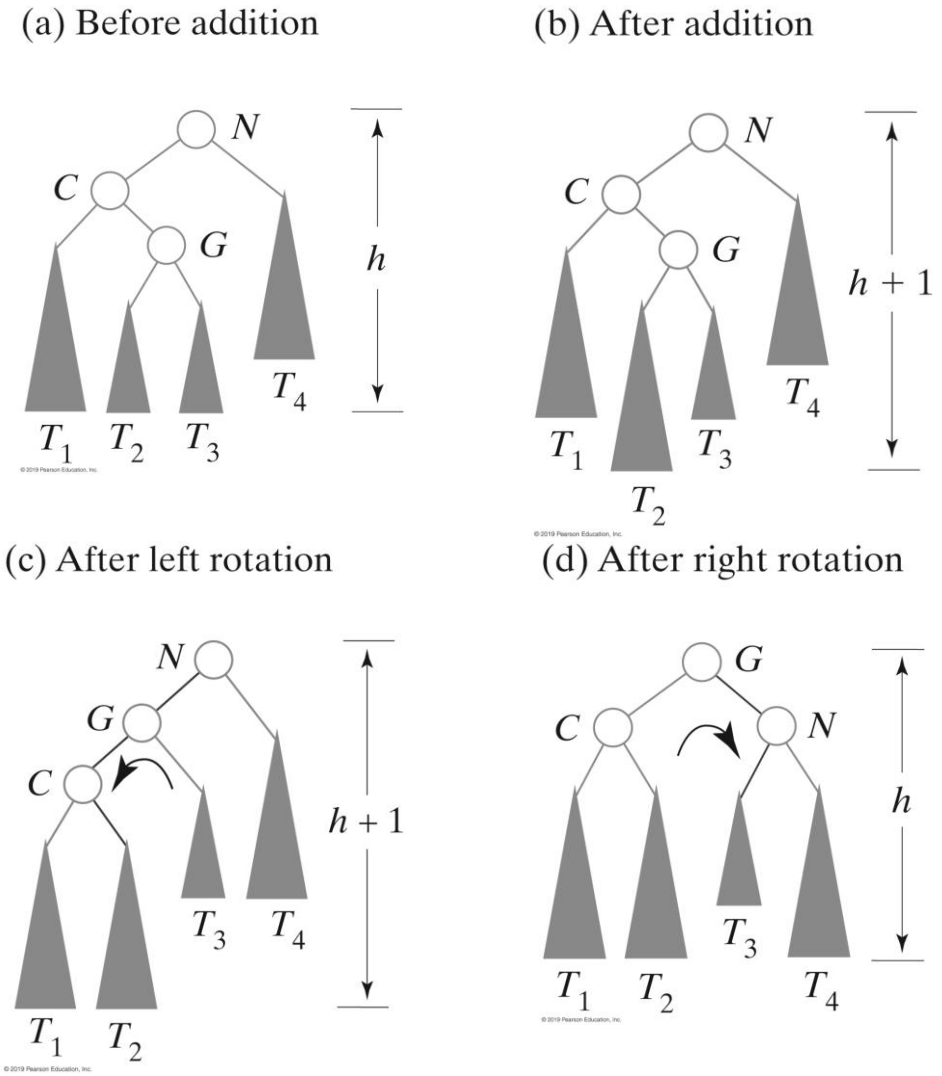


(d) After right rotation about 40



Double Rotations

- FIGURE 28-9 Before and after an addition to an AVL subtree that requires both a left rotation and a right rotation to maintain its balance



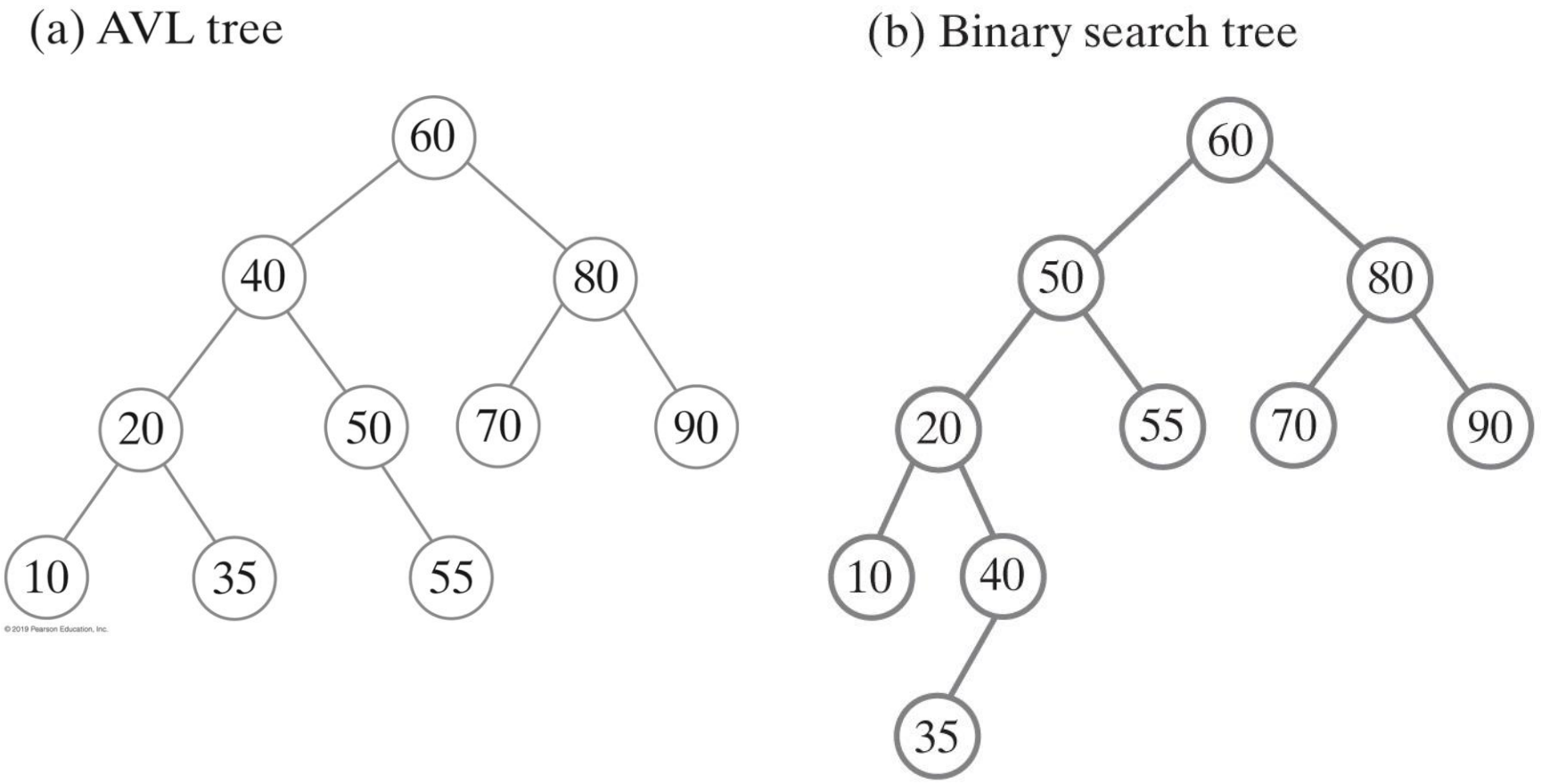
Left-right Double Rotations

- Algorithm that performs the left-right double rotation illustrated in Figure 28-9

Algorithm **rotateLeftRight(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
//in the right subtree of nodeN*'s left child.*
nodeC = *left child of* nodeN
Set nodeN*'s left child to the node returned by* rotateLeft(nodeC)
return rotateRight(nodeN)

An AVL Tree Versus a Binary Search Tree

- FIGURE 28-10 The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty AVL tree and a binary search tree



AVLTree class

- Inherits methods from BinarySearchTree
- Needs to override add() method.
- remove() not supported
- Needs private rotation methods

```
public class AVLTree<T extends Comparable<? super T>> extends BinarySearchTree<T>
//           implements SearchTreeInterface<T>
// Optional since BinarySearchTree implements this interface
{
   public AVLTree() {
      super();
   } // end default constructor

   public AVLTree(T rootEntry) {
      super(rootEntry);
   } // end constructor
```

Right rotation algorithm

Algorithm **rotateRight(nodeN)**
// Corrects an imbalance at a given node nodeN *due to an addition*
//in the left subtree of nodeN*'s left child.*
nodeC = *left child of* nodeN
Set nodeN*'s left child to* nodeC*'s right child*
Set nodeC*'s right child to* nodeN
return nodeC

Single right rotation implementation

```
/**
 * Corrects an imbalance at a given node nodeN due to an addition
 * in the left subtree of nodeN's left child.
 * @param nodeN
 * @return
 */
private BinaryNode<T> rotateRight(BinaryNode<T> nodeN) {
      // get the left child
      BinaryNode<T> nodeC = nodeN.getLeftChild();
      // get the right child our node's child, and set the
      // node's left child to that one.

      nodeN.setLeftChild(nodeC.getRightChild());

      // finally set the child's right child to this one
      nodeC.setRightChild(nodeN);

      // return the left child
      return nodeC;
}
```

Single left rotation implementation

```
/**
 * Corrects an imbalance at the node closest to a structural
 * change in the right subtree of the node's left child.
 * nodeN is a node, closest to the newly added leaf, at which
 * an imbalance occurs and that has a left child.
 * @param nodeN node closest to newly added leaf
 * @return
 */
private BinaryNode<T> rotateLeftRight(BinaryNode<T> nodeN) {
     BinaryNode<T> nodeC = nodeN.getLeftChild();
     nodeN.setLeftChild(rotateLeft(nodeC));
     return rotateRight(nodeN);
}
```

Right-left double rotation algorithm

Algorithm **rotateRightLeft(nodeN)**

// Corrects an imbalance at a given node nodeN *due to an addition*

//in the left subtree of nodeN*'s right child.*

nodeC = *right child of* nodeN

Set nodeN*'s right child to the node returned by* rotateRight(nodeC)

return rotateLeft(nodeN)

Right left double rotation implementation

```
/**
 * Corrects an imbalance at the node closest to a structural
 * change in the left subtree of the node's right child.
 * nodeN is a node, closest to the newly added leaf, at which
 * an imbalance occurs and that has a right child.
 * @param nodeN
 * @return
 */
private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN) {
     BinaryNode<T> nodeC = nodeN.getRightChild();
     nodeN.setRightChild(rotateRight(nodeC));
     return rotateLeft(nodeN);
}
```

Left right double rotation implementation

```
/**
 * Corrects an imbalance at the node closest to a structural
 * change in the right subtree of the node's left child.
 * nodeN is a node, closest to the newly added leaf, at which
 * an imbalance occurs and that has a left child.
 * @param nodeN node closest to newly added leaf
 * @return
 */
private BinaryNode<T> rotateLeftRight(BinaryNode<T> nodeN) {
          BinaryNode<T> nodeC = nodeN.getLeftChild();
          nodeN.setLeftChild(rotateLeft(nodeC));
          return rotateRight(nodeN);
}
```

Pseudocode to rebalance the tree

```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{       //Addition was in nodeN's left subtree
        if(the left child of nodeN has a left subtree that is taller than its right subtree)
                rotateRight(nodeN) //Addition was in left subtree of left child
        else
                rotateLeftRight(nodeN) //Addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{                       //Addition was in nodeN's right subtree
        if (the right child of nodeN has a right subtree that is taller than its left subtree)
                rotateLeft(nodeN)               //Addition was in right subtree of right child
        else
                rotateRightLeft(nodeN) //Addition was in left subtree of right child
}
```

Rebalance implementation

- Done after adding a node

```
/**
 * rebalance the tree, usually after an addition was made
 * @param nodeN
 * @return
 */
private BinaryNode<T> rebalance(BinaryNode<T> nodeN) {

      // get the height difference between the left and right subtrees

      int heightDifference = getHeightDifference(nodeN);

      if (heightDifference > 1) {
            // Left subtree is taller by more than 1,
            // so addition was in left subtree
            if (getHeightDifference(nodeN.getLeftChild()) > 0)
                  // Addition was in left subtree of left child
                  nodeN = rotateRight(nodeN);
            else
                  // Addition was in right subtree of left child
                  nodeN = rotateLeftRight(nodeN);
      } else if (heightDifference < -1) {
            // Right subtree is taller by more than 1,
            // so addition was in right subtree
            if (getHeightDifference(nodeN.getRightChild()) < 0)
                  // Addition was in right subtree of right child
                  nodeN = rotateLeft(nodeN);
            else
                  // Addition was in left subtree of right child
                  nodeN = rotateRightLeft(nodeN);
      }

      return nodeN;
}
```

getHeightDifference()

- Uses getHeight() from BinaryNode to compute the height differences in the left and right subtrees.

```
/**
 * Returns the difference in heights of the given node's
 * left and right subtrees.
 * @param node
 * @return
 */
private int getHeightDifference(BinaryNode<T> node) {
    BinaryNode<T> left = node.getLeftChild();
    BinaryNode<T> right = node.getRightChild();

    int leftHeight, rightHeight;

    if (left == null)
        leftHeight = 0;
    else
        leftHeight = left.getHeight();

    if (right == null)
        rightHeight = 0;
    else
        rightHeight = right.getHeight();

    return leftHeight - rightHeight;
}
```

add() method

- calls addEntry() then rebalances
- note: no remove() method

```
public T add(T newEntry) {
      T result = null;

      if (isEmpty())
            setRootNode(new BinaryNode<>(newEntry));
      else {

            // add an entry to the tree, then rebalance

            BinaryNode<T> rootNode = getRootNode();
            result = addEntry(rootNode, newEntry);
            setRootNode(rebalance(rootNode));
      }

      return result;
}
```

addEntry() method

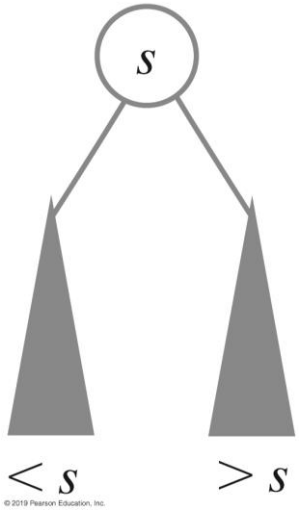
- Traverses the tree recursively, adding a node then rebalancing.

```
/**
 * Traverse the tree recursively to add an entry until a leaf is reached.
 * Then rebalance.
 * @param rootNode
 * @param newEntry
 * @return
 */
private T addEntry(BinaryNode<T> rootNode, T newEntry) {
     // Assertion: rootNode != null
     T result = null;
     int comparison = newEntry.compareTo(rootNode.getData());

     // if the node is the same just return the data
     // if not, if there is a child, keep traversing the tree
     //    if at a leaf, create a new node there, then rebalance after
     //    recursive call returns.
     if (comparison == 0) {
          result = rootNode.getData();
          rootNode.setData(newEntry);
     } else if (comparison < 0) {
          if (rootNode.hasLeftChild()) {
               BinaryNode<T> leftChild = rootNode.getLeftChild();
               result = addEntry(leftChild, newEntry);
               rootNode.setLeftChild(rebalance(leftChild));
          } else
               rootNode.setLeftChild(new BinaryNode<>(newEntry));
     } else {

          if (rootNode.hasRightChild()) {
               BinaryNode<T> rightChild = rootNode.getRightChild();
               result = addEntry(rightChild, newEntry);
               rootNode.setRightChild(rebalance(rightChild));
          } else
               rootNode.setRightChild(new BinaryNode<>(newEntry));
     }

     return result;
}
```

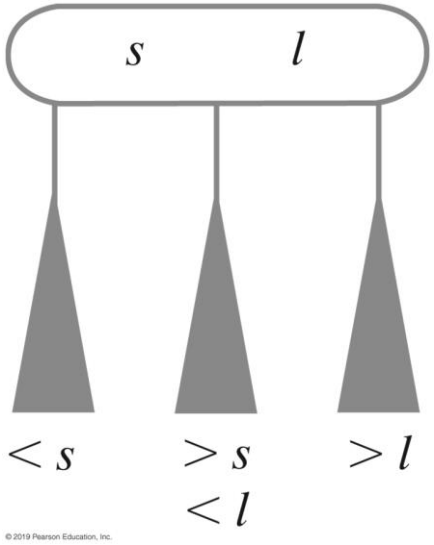
2-3 Trees

- General search tree whose interior nodes must have either two or three children
  - A 2-node contains one data item s and has two children
  - A 3-node contains two data items, s and l, and has three children

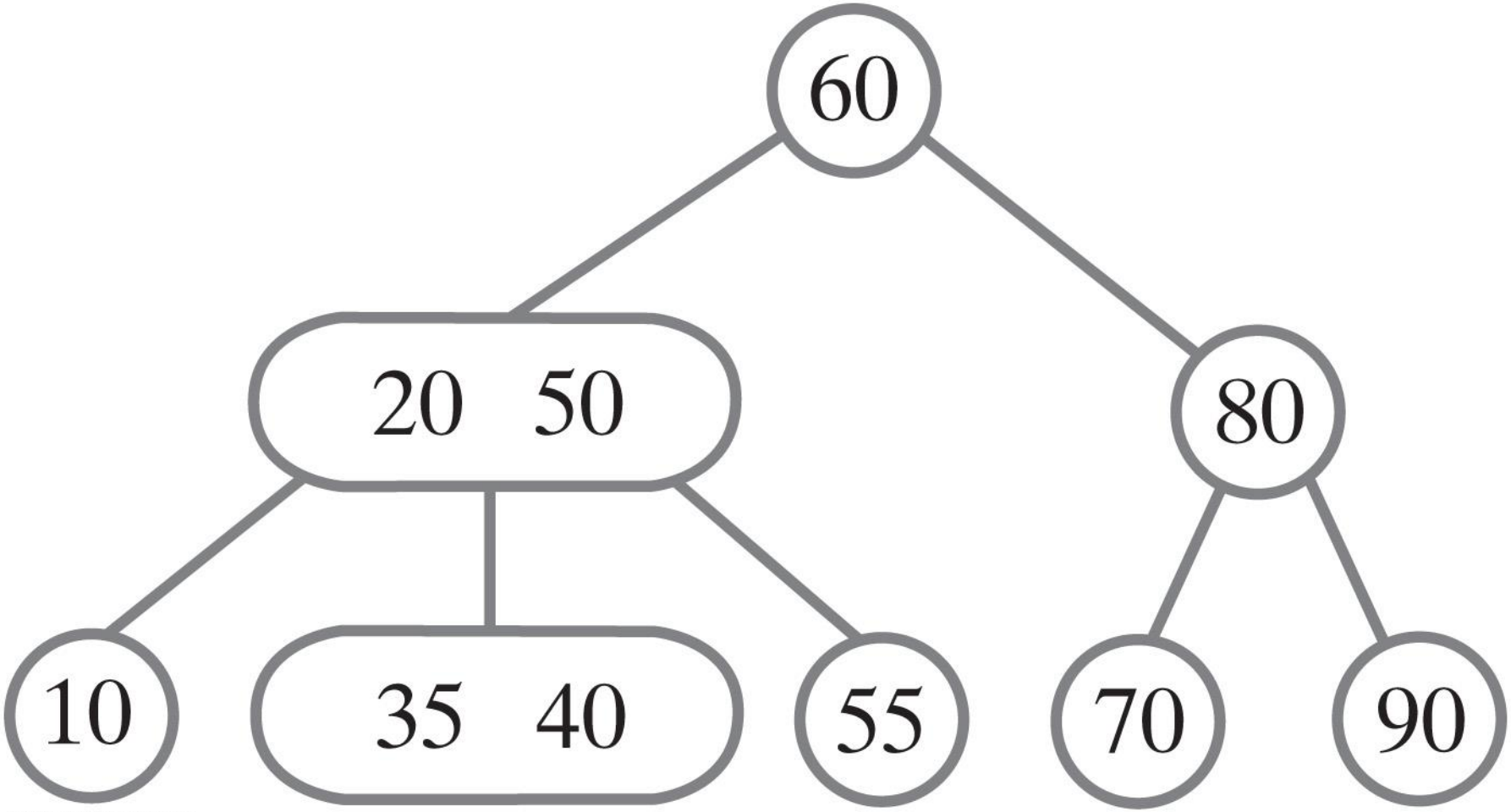
(a) A 2-node



(b) A 3-node



2-3 Trees

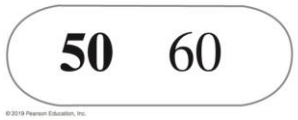


Building 2-3 Trees

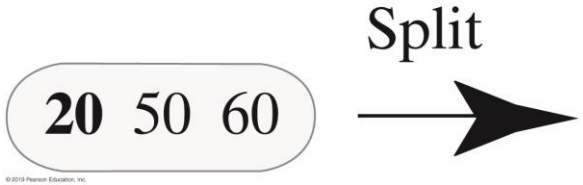
- FIGURE 28-13 An initially empty 2-3 tree after three additions

(a) After adding 60, the tree is a 2-node

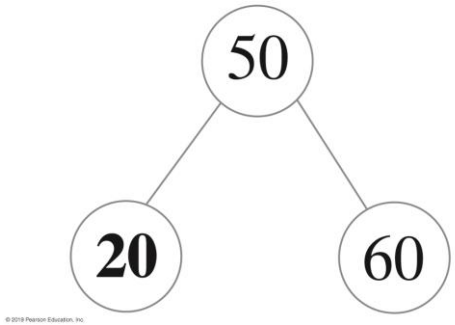
(b) After adding 50, the tree is a 3-node



(c) A 3-node cannot accommodate 20, so it must split



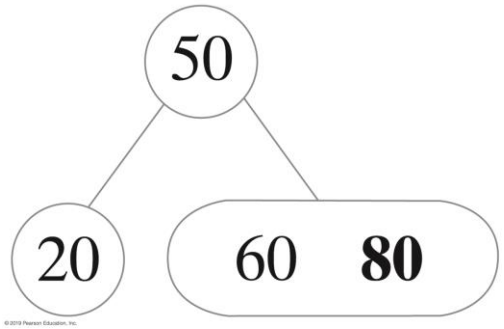
(d) After adding 20



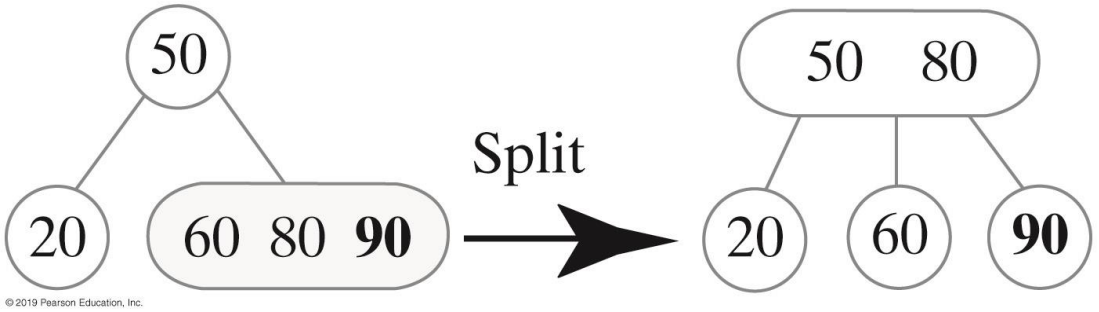
Building 2-3 Trees

- FIGURE 28-14 The 2-3 tree after three additions

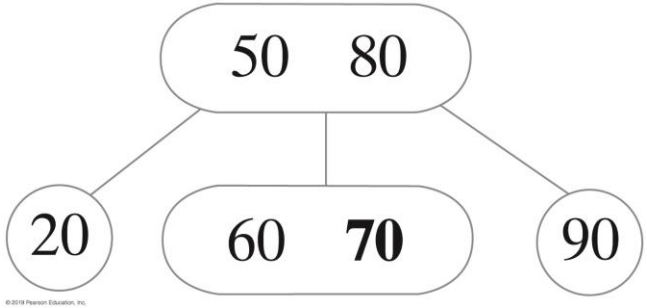
(a) After adding 20



(b) Splitting the leaf and adding 90



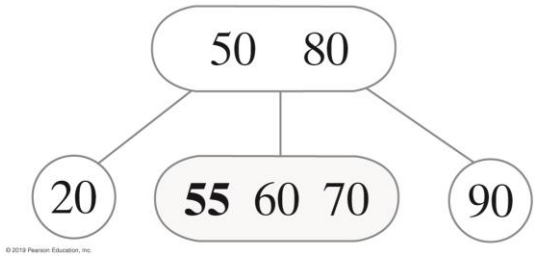
(c) After adding 70



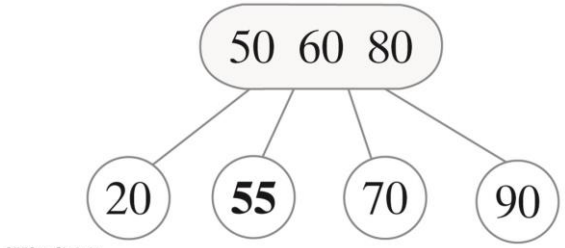
Building 2-3 Trees

- FIGURE 28-15 Adding 55 to the 2-3 tree in Figure 28-14c causes a leaf and then the root to split

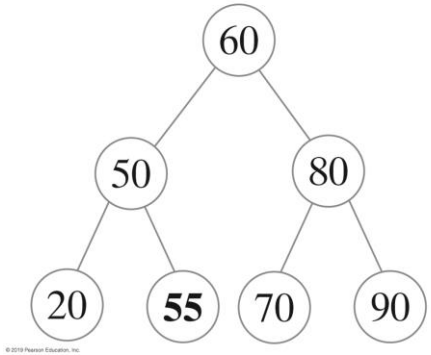
(a) 55 belongs in the middle leaf, but it has no room



(b) The leaf splits, but the root has no room for the 60 that moves up



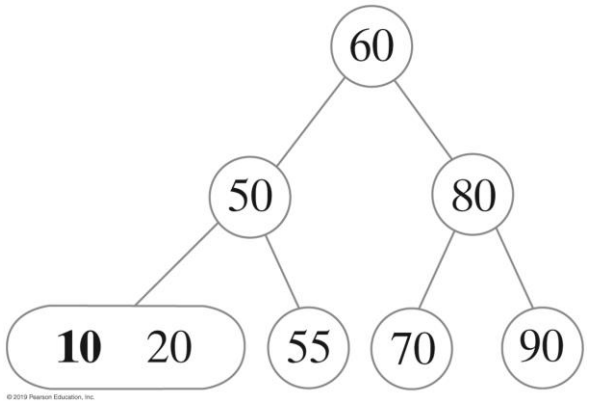
(c) The tree after the root splits



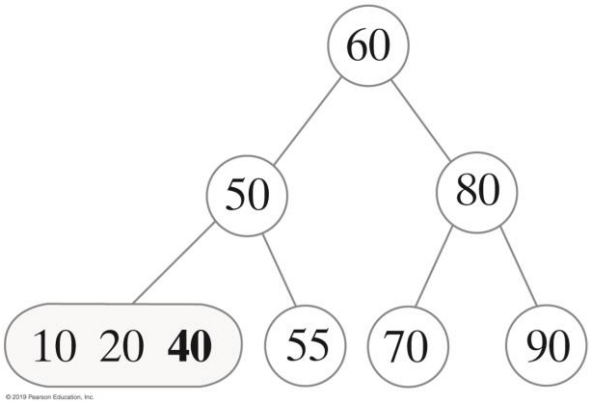
Building 2-3 Trees

- FIGURE 28-16 Adding 10 and 40 to the 2-3 tree in Figure 28-15c

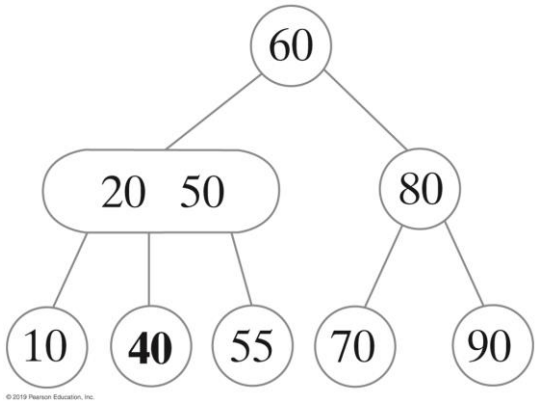
(a) After adding 10



(b) 40 belongs in a leaf that has no room

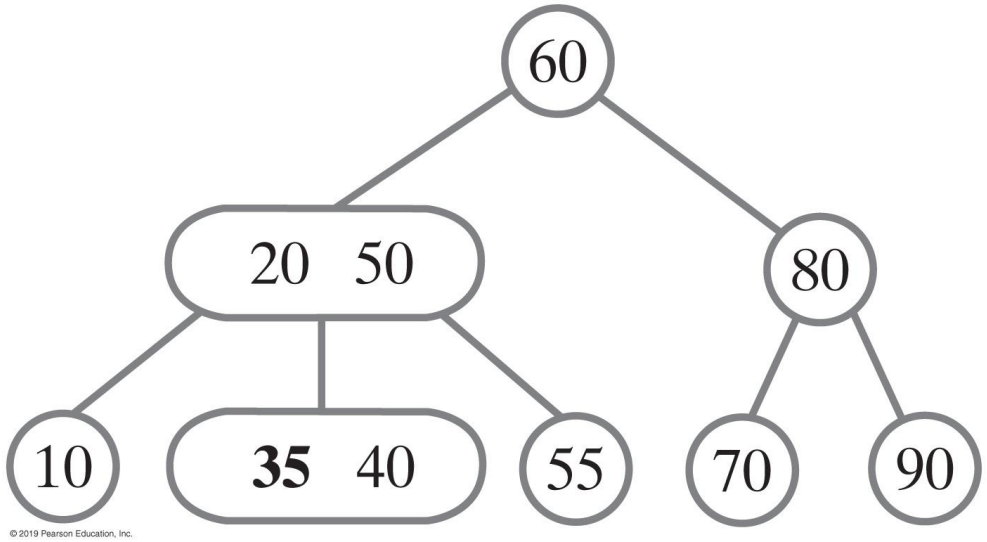


(c) The tree after the leaf splits



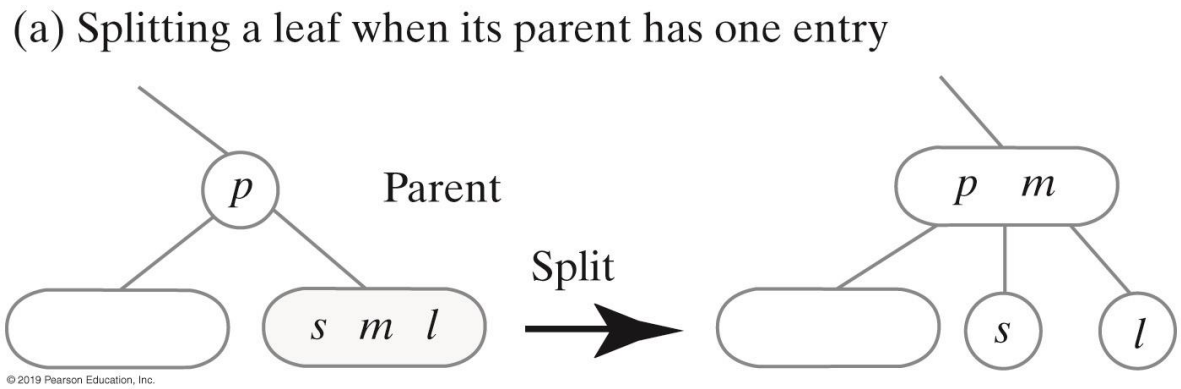
Building 2-3 Trees

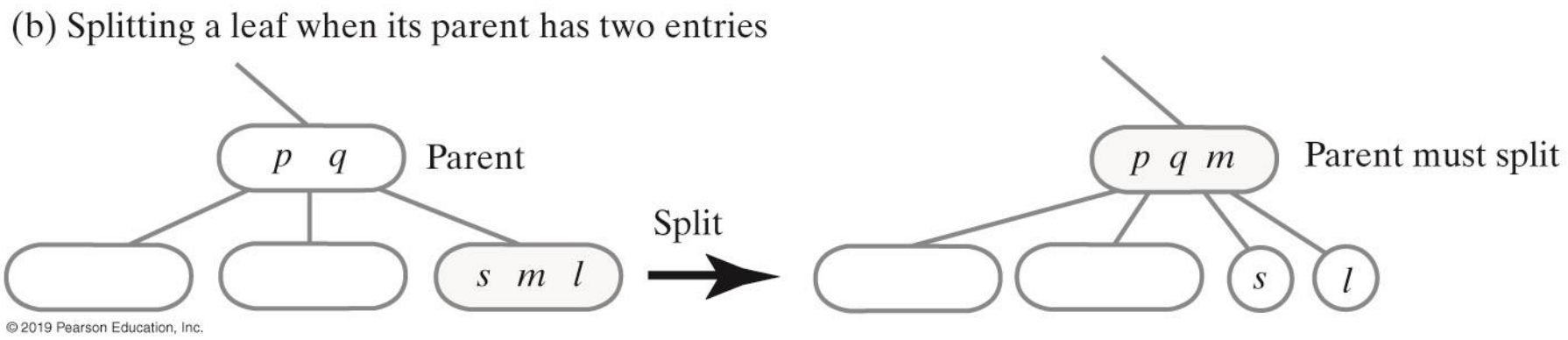
- FIGURE 28-17 The 2-3 tree in Figure 28-16c after adding 35



Splitting Nodes During Addition

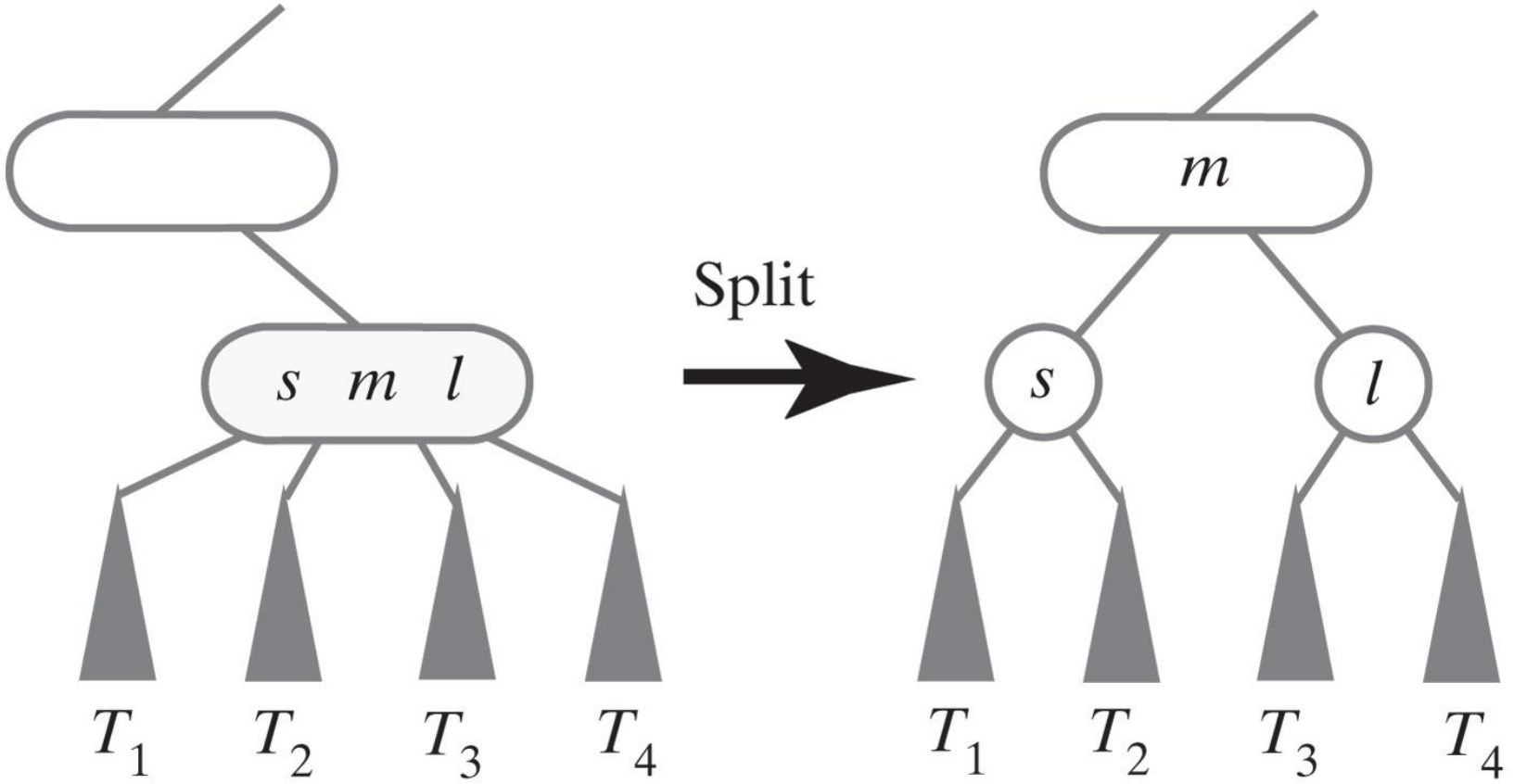
- FIGURE 28-18 Splitting a leaf to accommodate a new entry





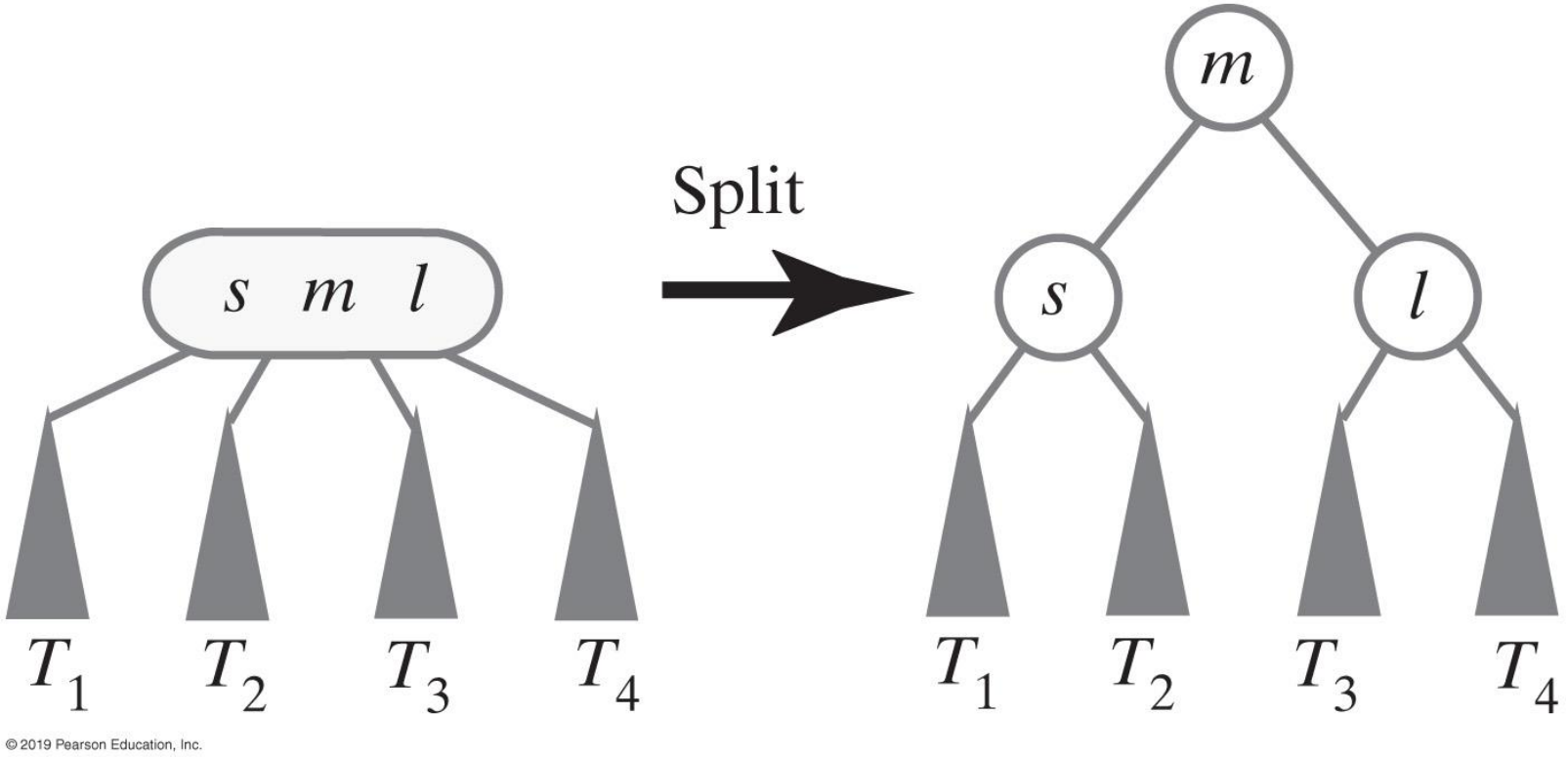
Splitting Nodes During Addition

- FIGURE 28-19 Splitting an internal node to accommodate a new entry



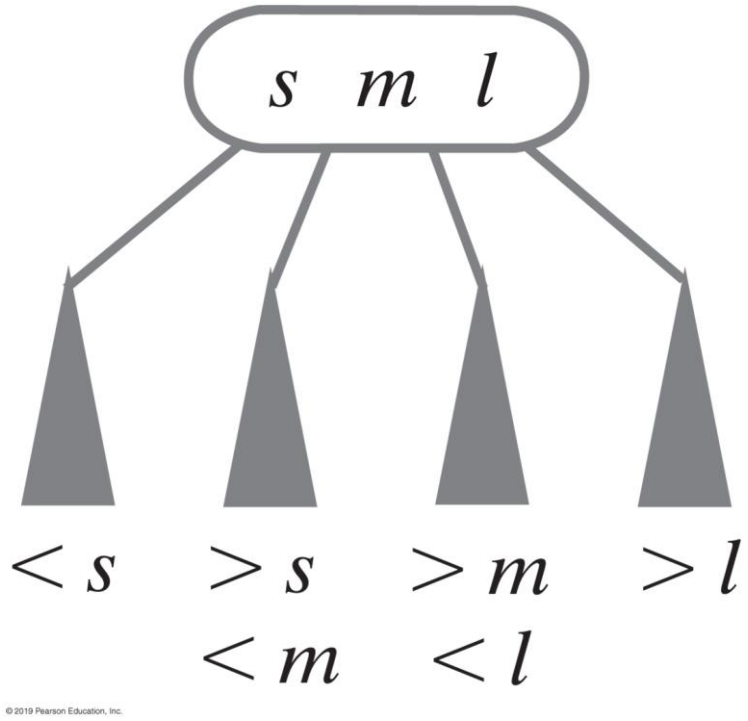
Splitting Nodes During Addition

- FIGURE 28-20 Splitting the root to accommodate a new entry



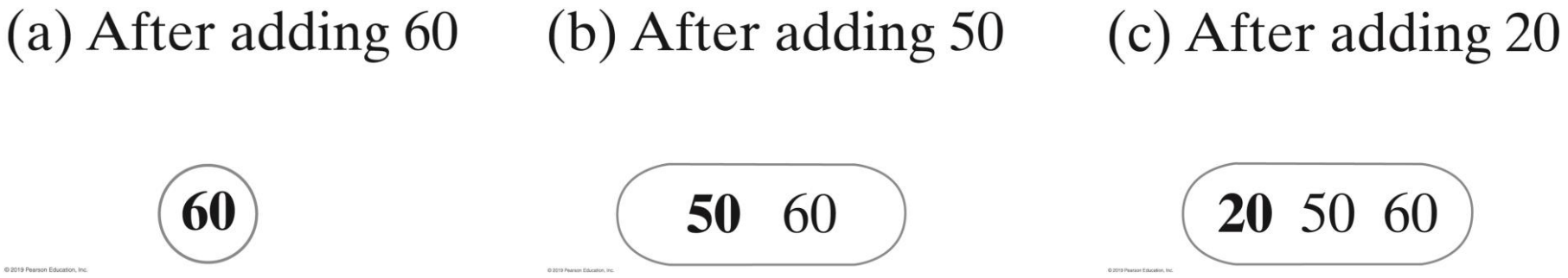
2-4 Trees

- Sometimes called a 2-3-4 tree
  - General search tree
  - Interior nodes must have either two, three, or four children
  - Leaves occur on the same level
- This tree also contains 4-nodes.
  - A 4-node contains three data items s, m, and l and has four children.



Adding Entries to a 2-4 Tree

- FIGURE 28-22 Adding 60, 50, and 20 to an initially empty 2-4 tree



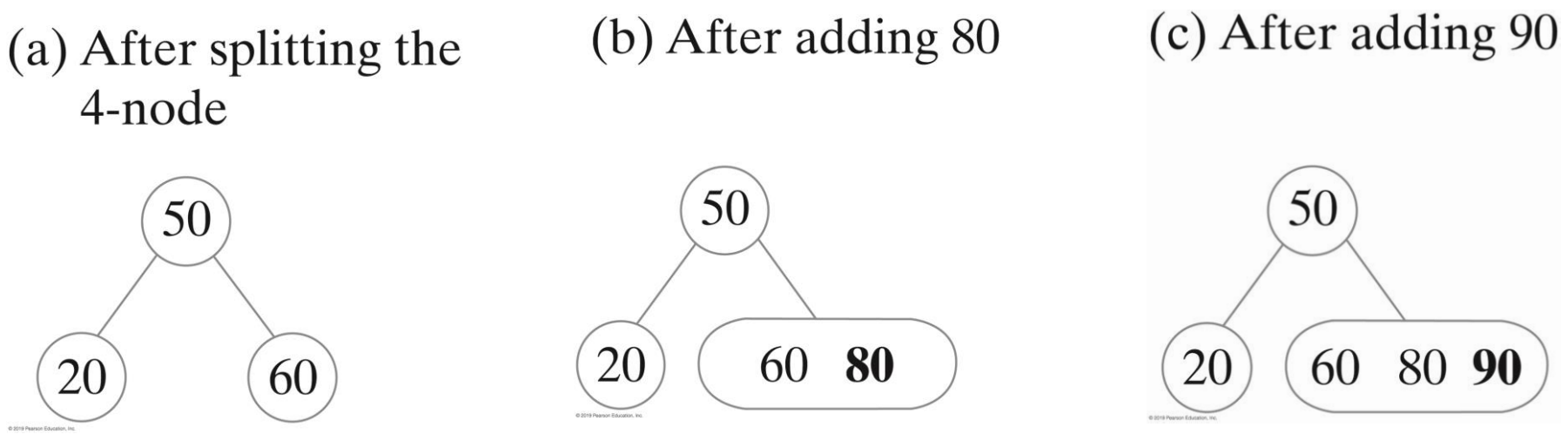
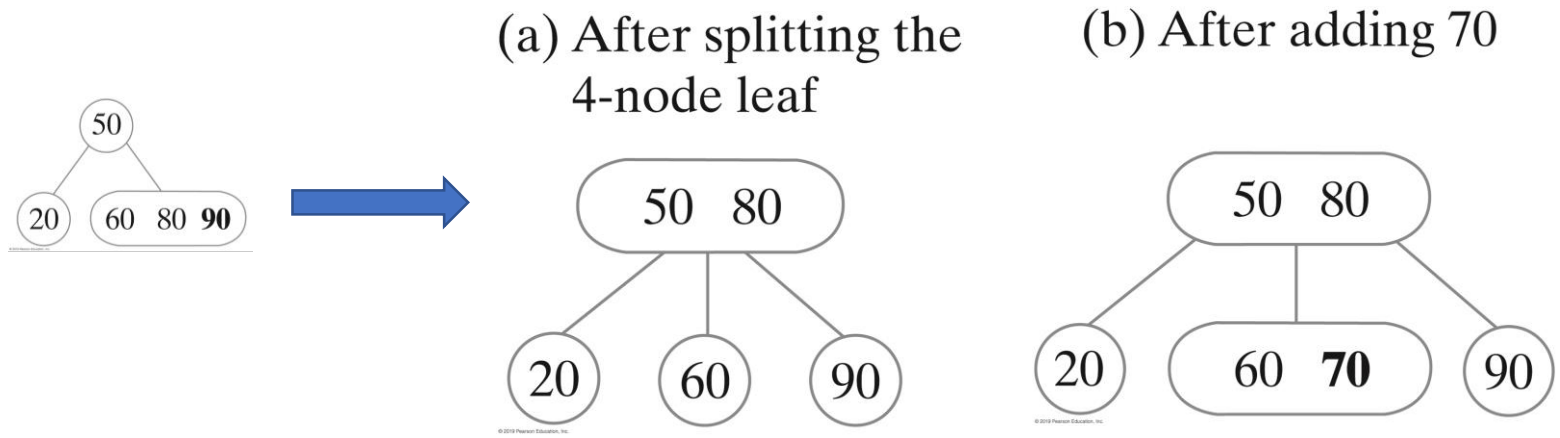


FIGURE 28-23 Adding 80 and 90 to the tree in Figure 28-22c

Adding Entries to a 2-4 Tree

- FIGURE 28-24a Adding 70 to the 2-4 tree in Figure 28-23



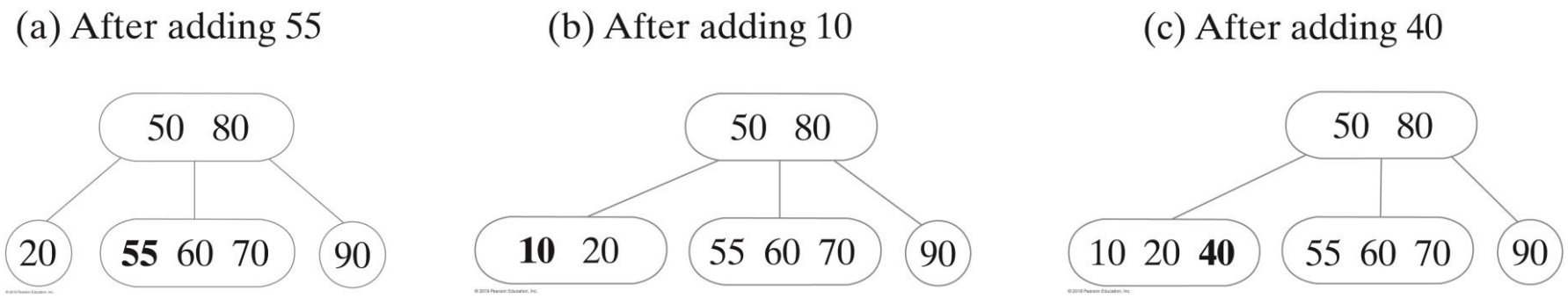
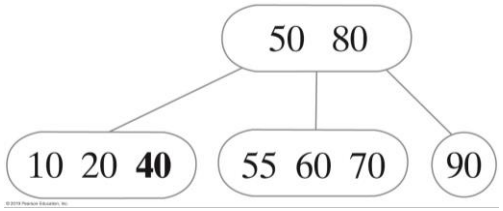


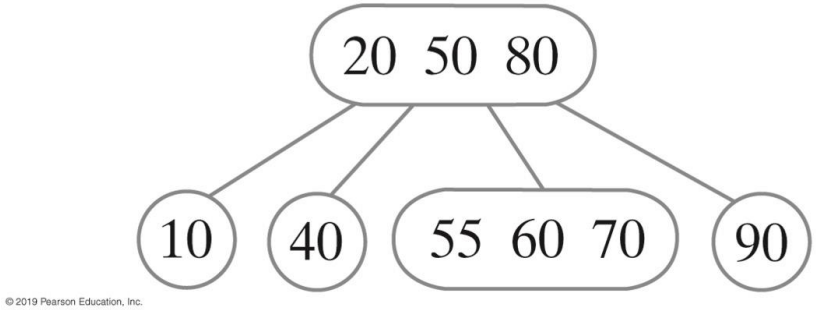
FIGURE 28-25 Adding 55, 10, and 40 to the 2-4 tree in Figure 28-24b

Adding Entries to a 2-4 Tree

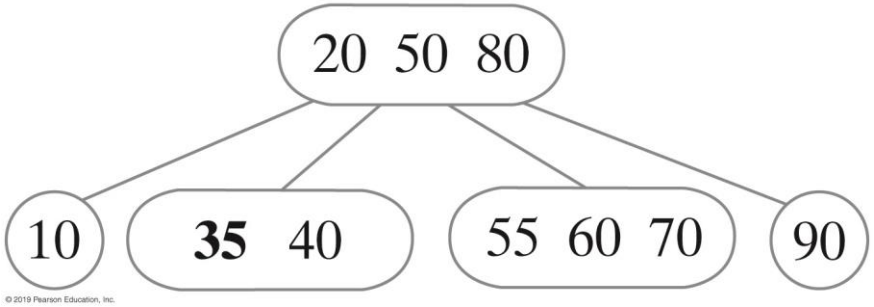
- FIGURE 28-26 Adding 35 to the 2-4 tree in Figure 28-25c



(a) After splitting the 4-node leaf encountered while searching for a place for 35

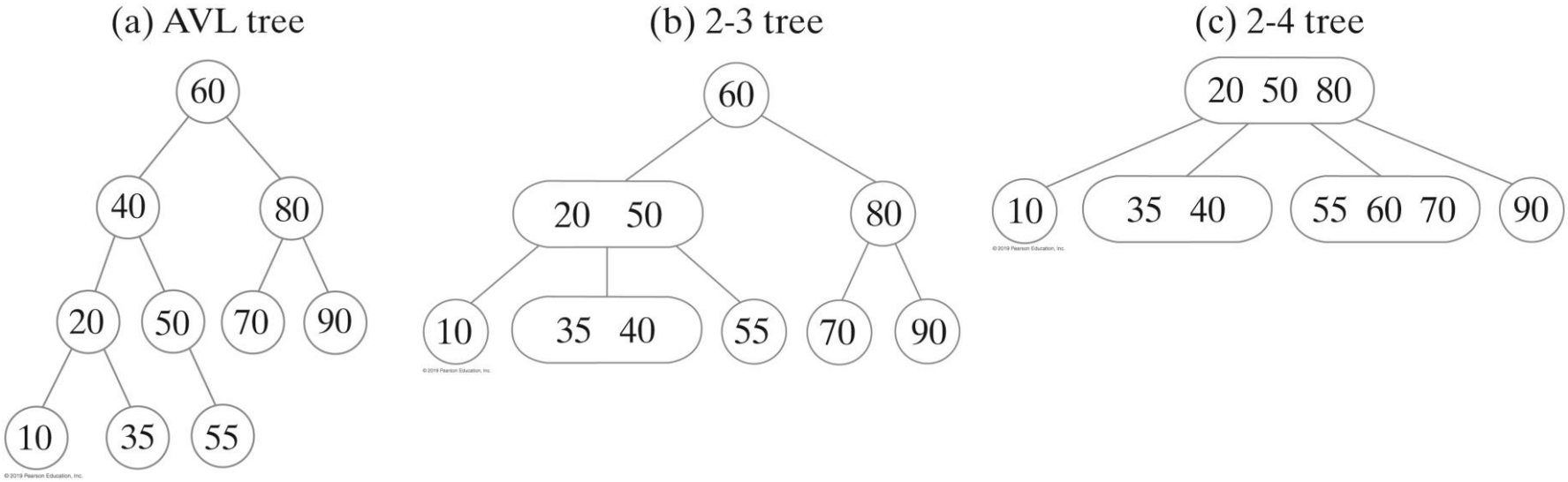


(b) After adding 35



Building 2-4 Trees - A Comparison

- FIGURE 28-28 Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35

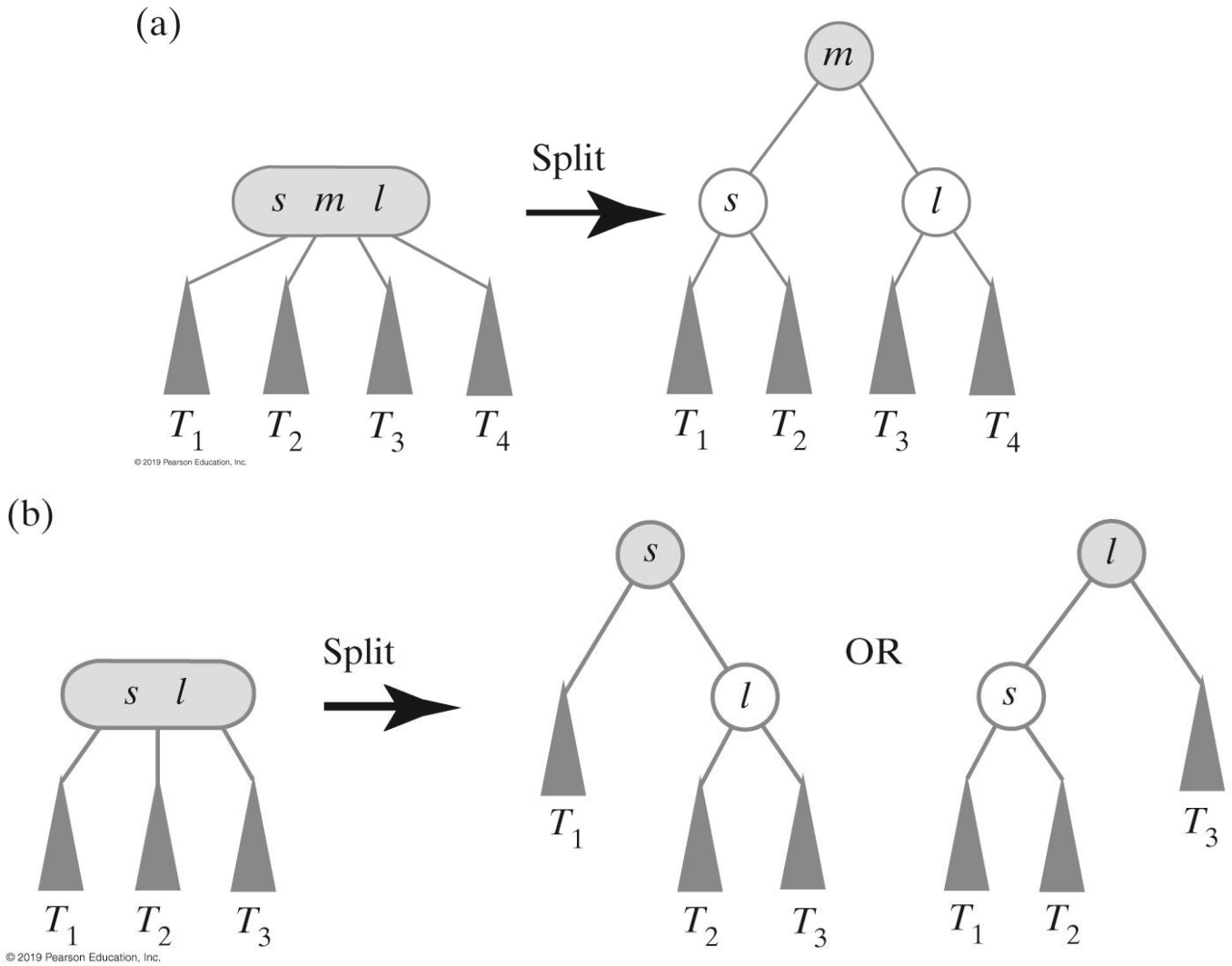


Red-Black Trees

- Binary tree that is equivalent to a 2-4 tree
  - Conceptually more involved
  - Uses only 2-nodes and so is more efficient.
- Adding an entry to a red-black tree is like adding an entry to a 2-4 tree
  - Since it is a binary tree, uses simpler operations to maintain its balance than a 2-4 tree

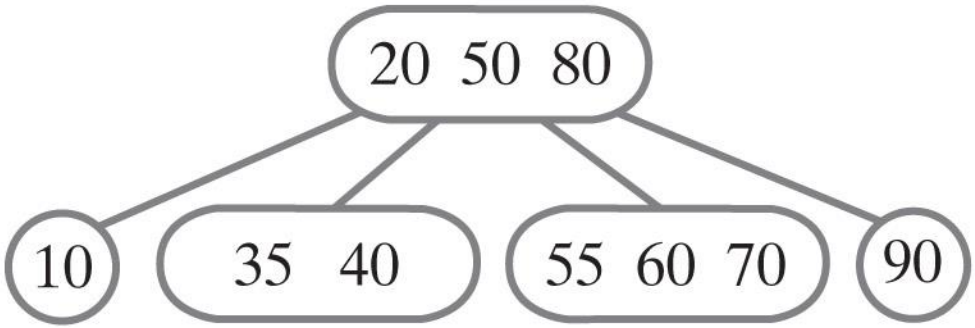
Red-Black Trees

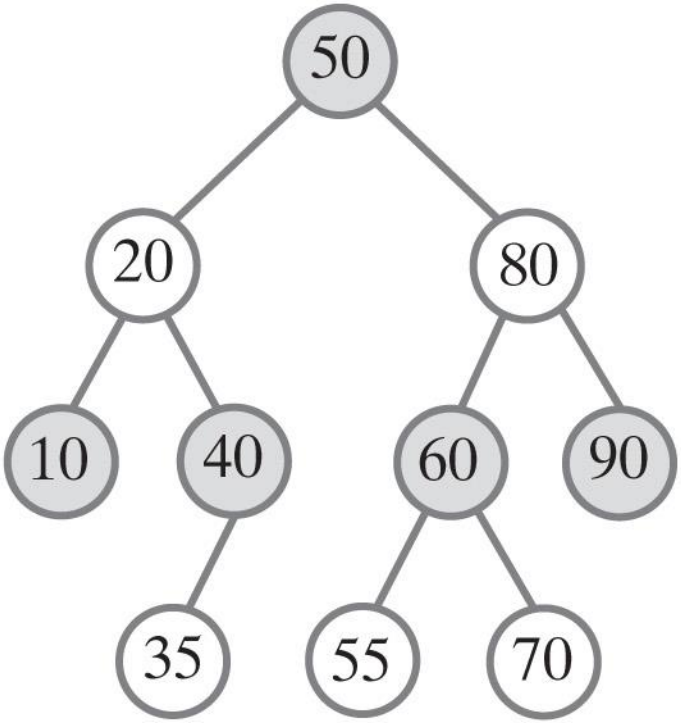
- FIGURE 28-28 Using 2-nodes to represent (a) a 4-node; (b) a 3-node



Red-Black Trees

- FIGURE 28-29 A 2-4 tree (Figure 28-28c) and its equivalent red-black tree





Properties of a Red-Black Tree

- The root is black.
- Every red node has a black parent.
- Any children of a red node are black; that is, a red node cannot have red children.
- Every path from the root to a leaf contains the same number of black nodes.

Adding Entries to a Red-Black Tree

- Adding an entry to a red-black tree results in a new red leaf.
- The color of this leaf can change later when other entries are added or removed.

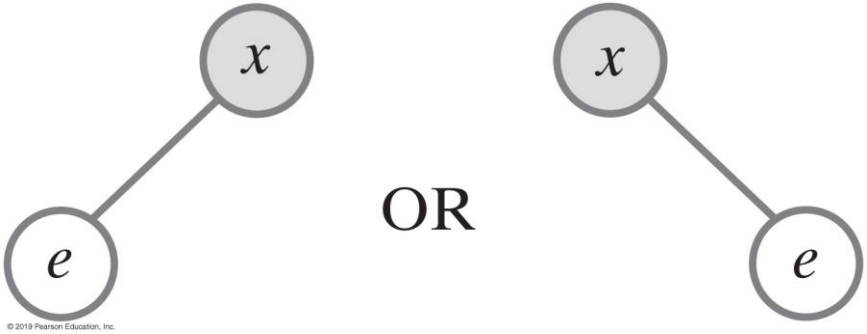
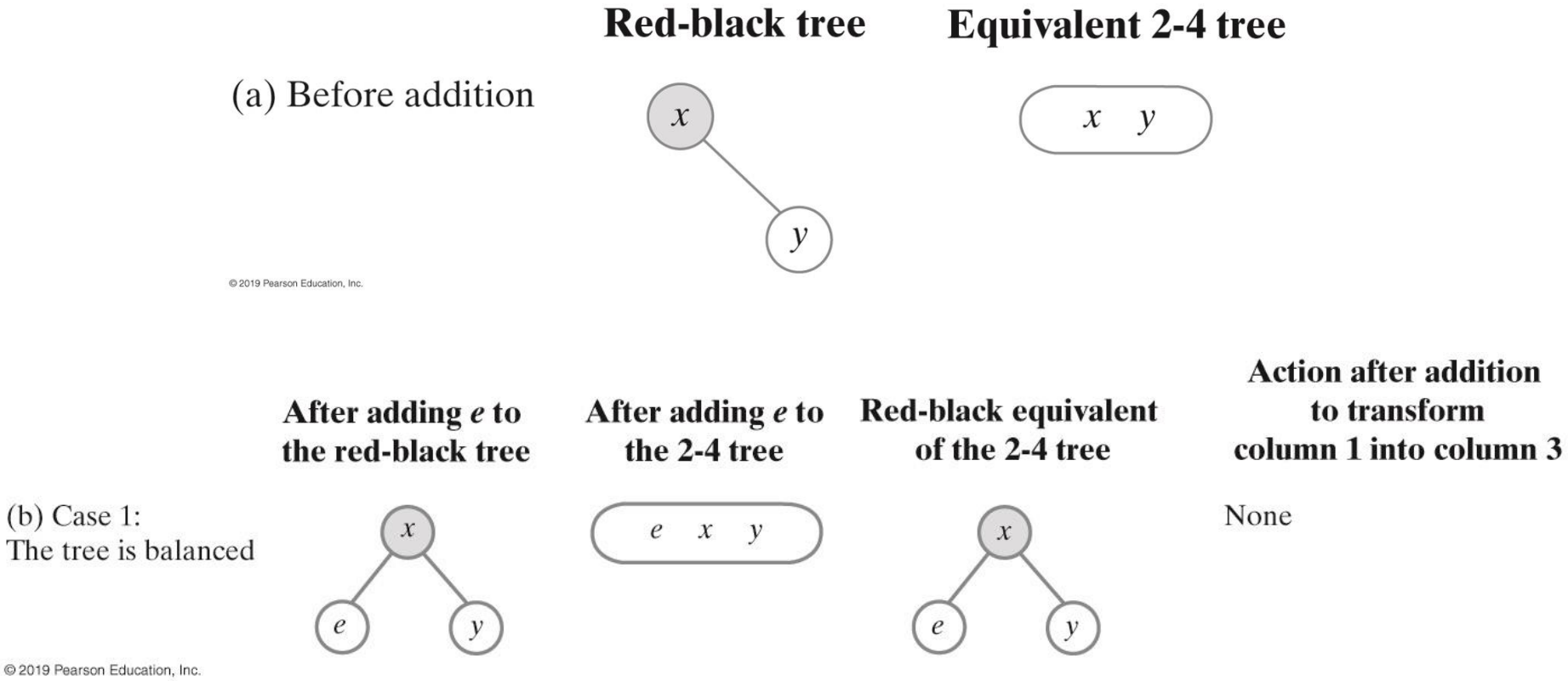


FIGURE 28-30 The result of adding a new entry e to a one-node red-black tree

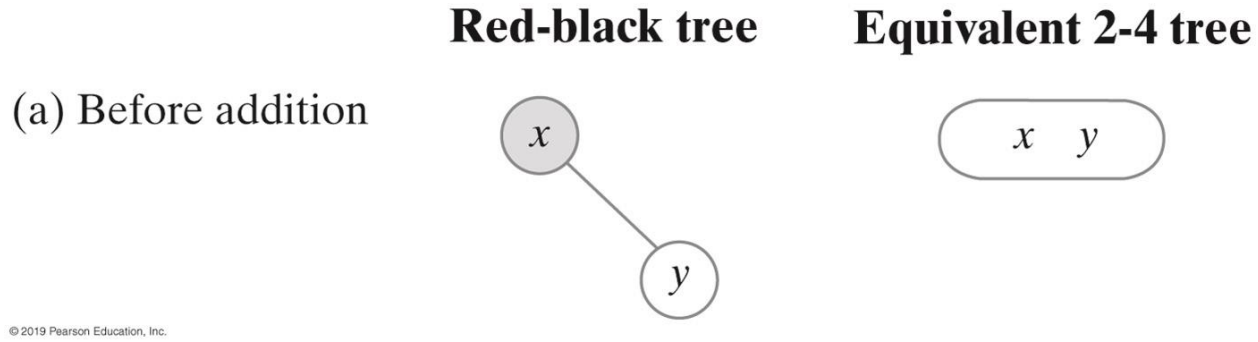
Adding Entries to a Red-Black Tree

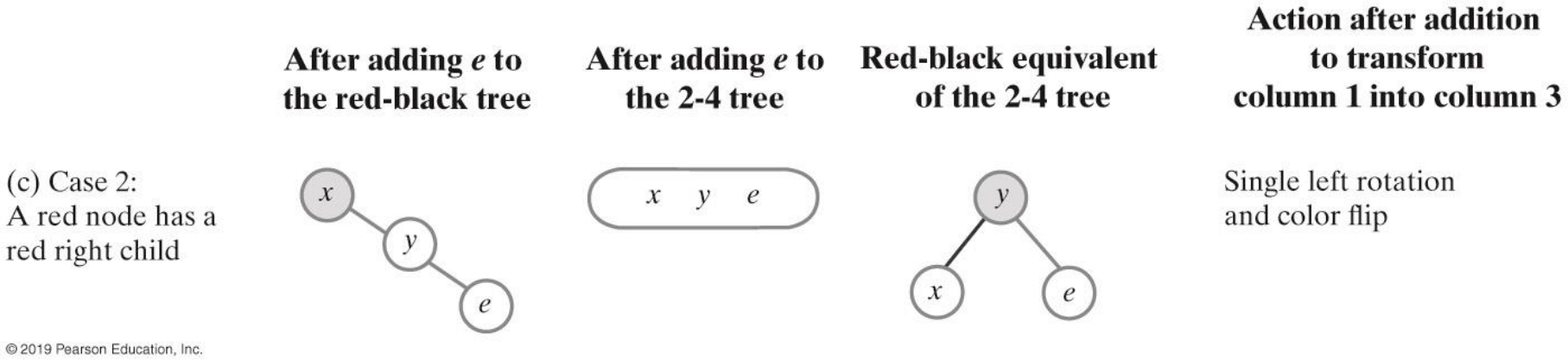
- FIGURE 28-31b The possible results of adding a new entry *e* to a two-node red-black tree



Adding Entries to a Red-Black Tree

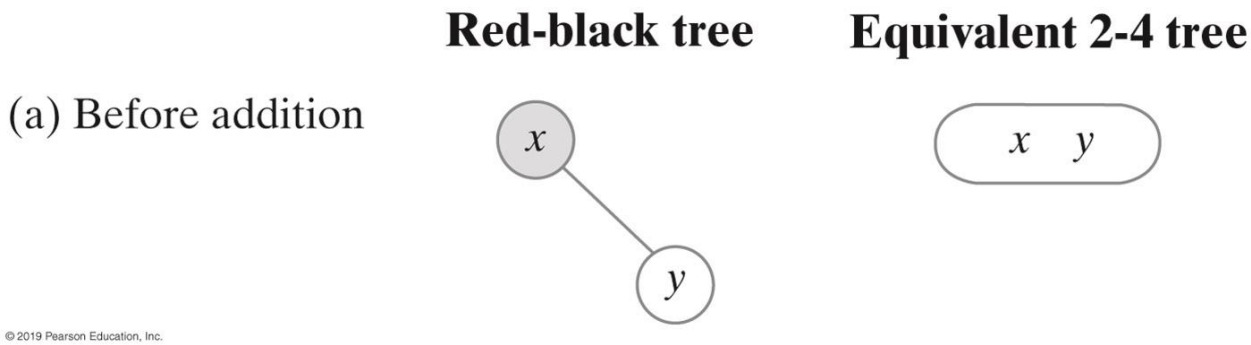
- FIGURE 28-31c The possible results of adding a new entry *e* to a two-node red-black tree

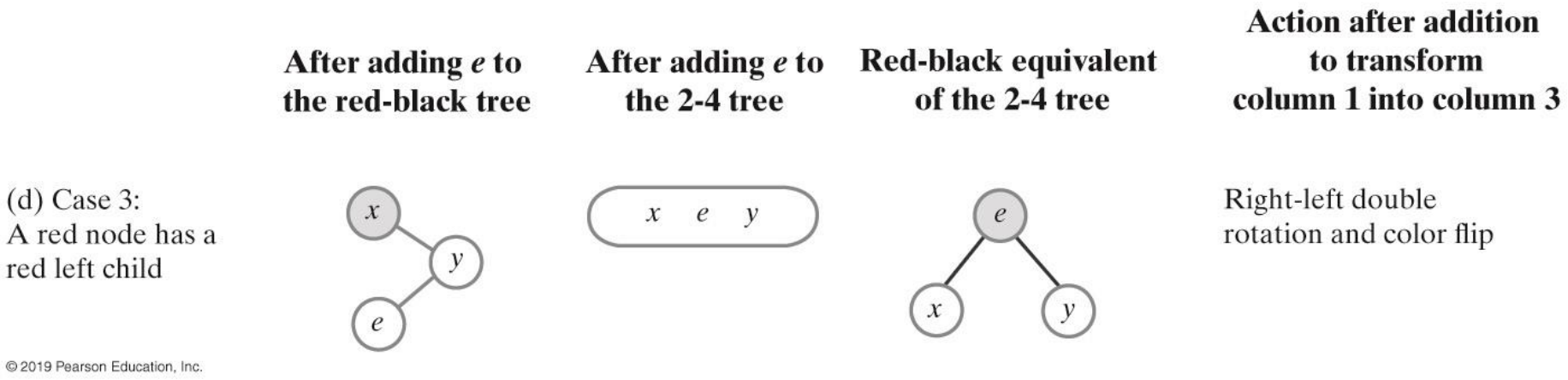




Adding Entries to a Red-Black Tree

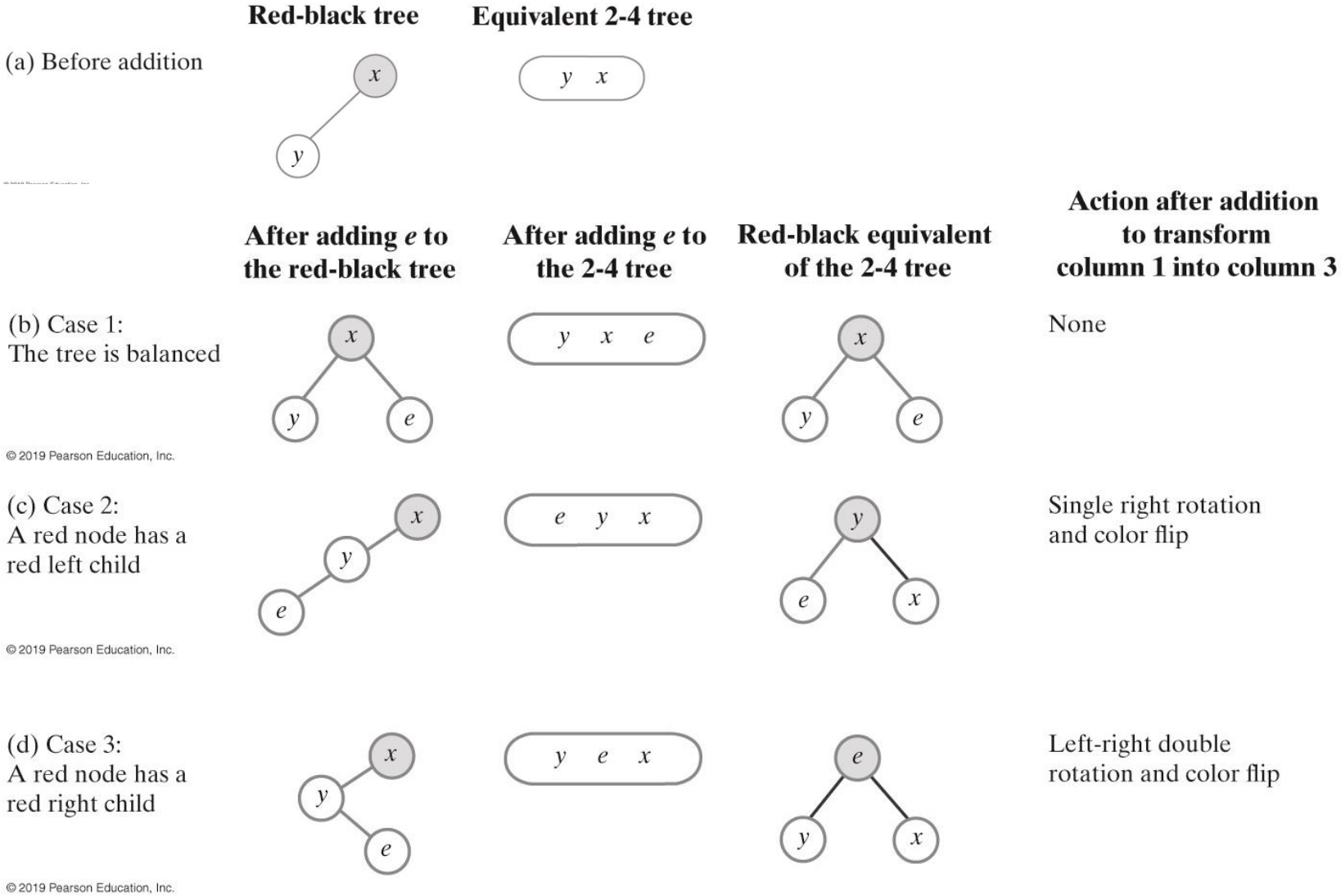
- FIGURE 28-31d The possible results of adding a new entry *e* to a two-node red-black tree





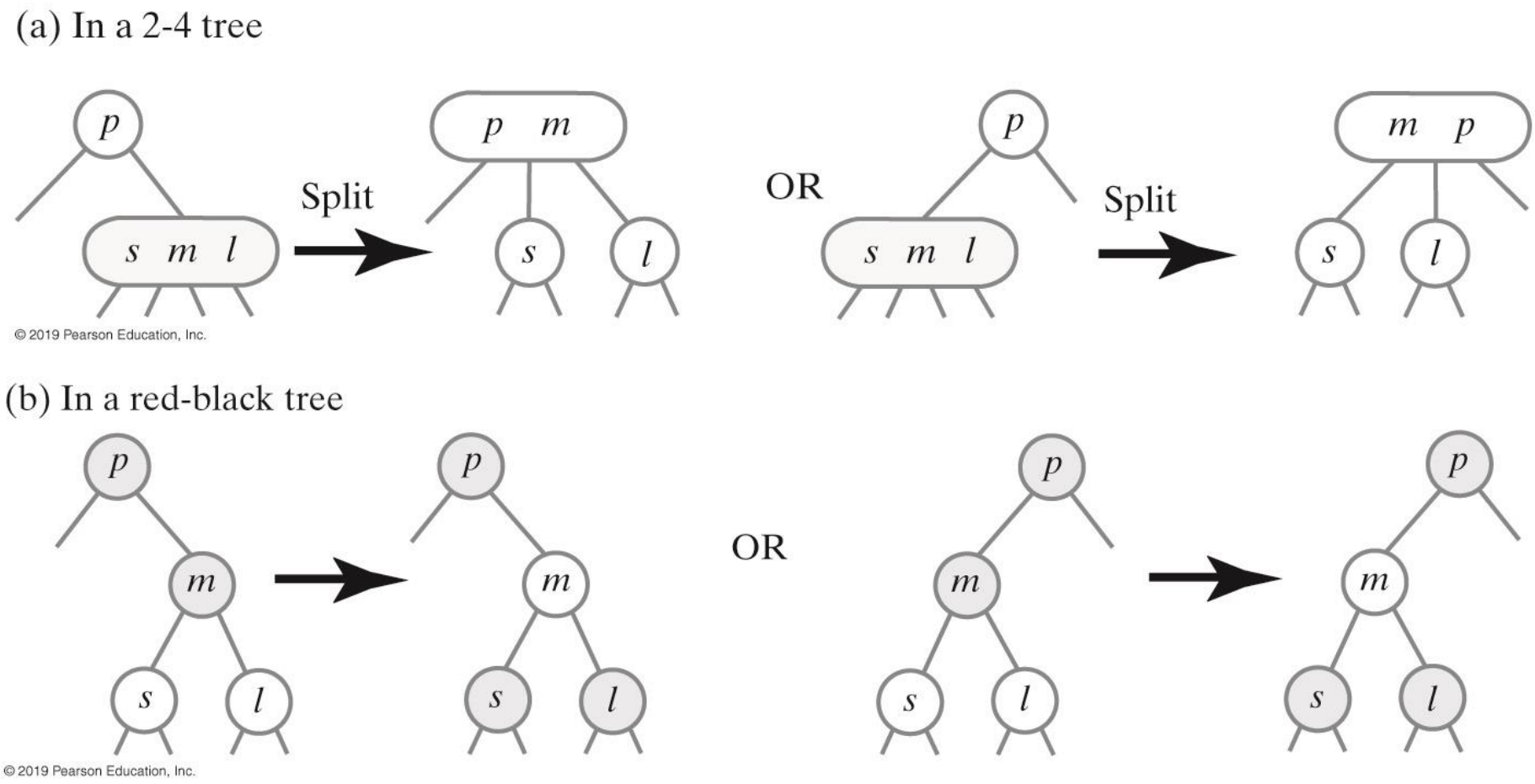
Adding Entries to a Red-Black Tree

- FIGURE 28-32 The possible results of adding a new entry e to a two-node red-black tree: mirror images of Figure 28-31



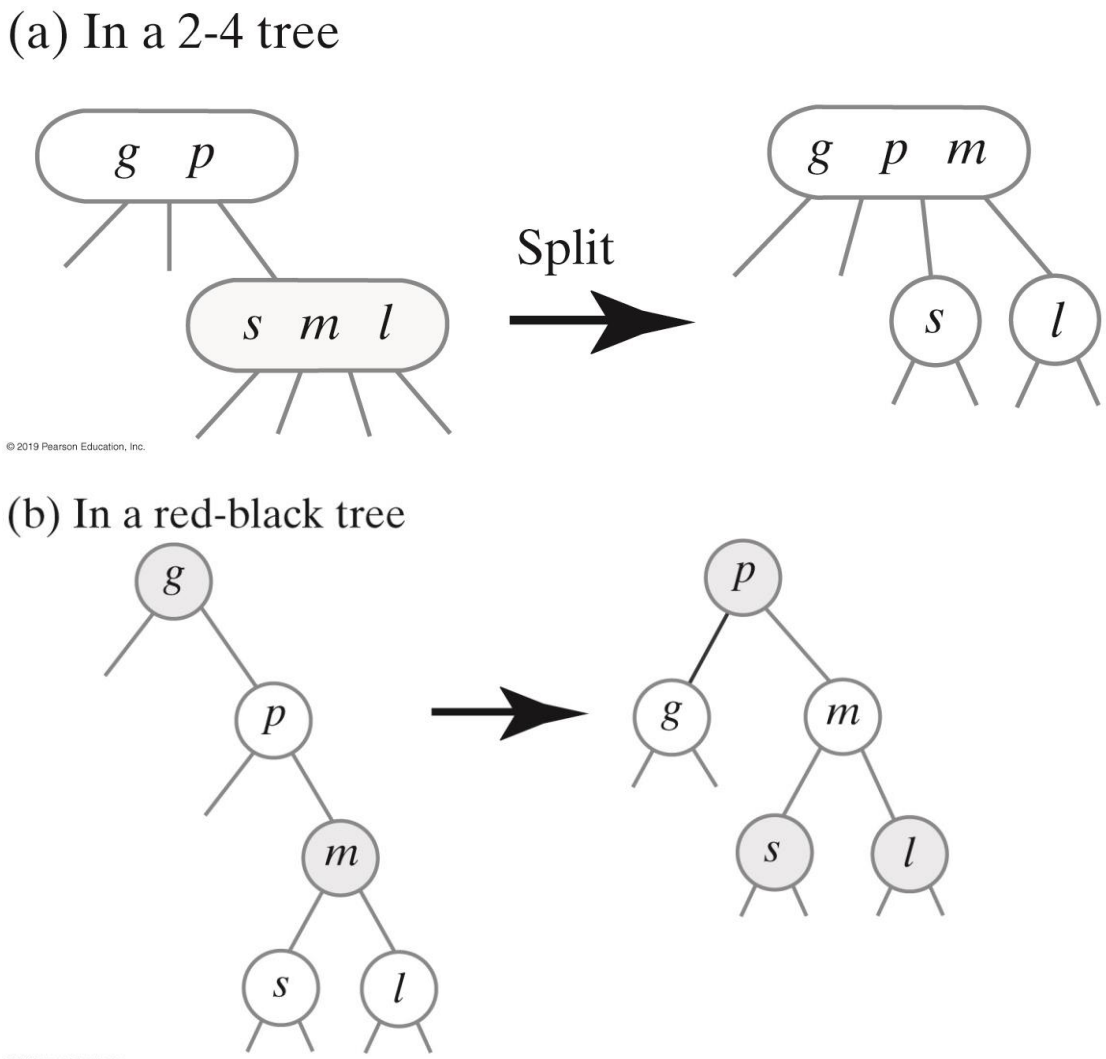
Splitting a 4-node

- FIGURE 28-33 Splitting a 4-node whose parent is a 2-node



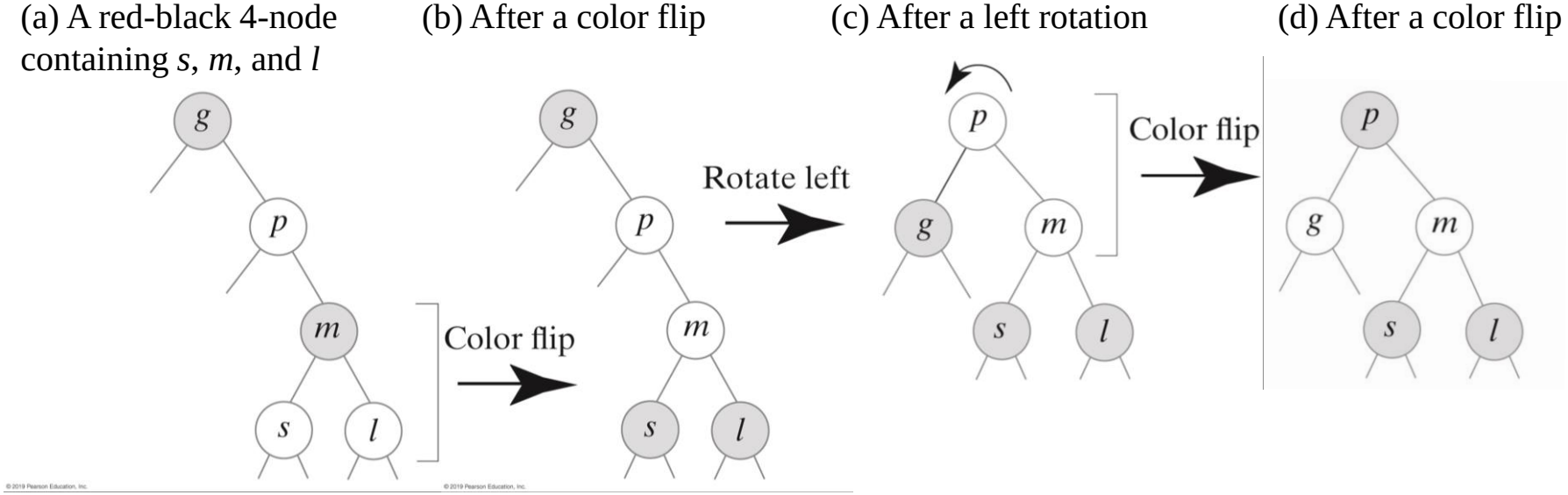
Splitting a 4-node

- FIGURE 28-34 Splitting a 4-node whose parent is a 3-node



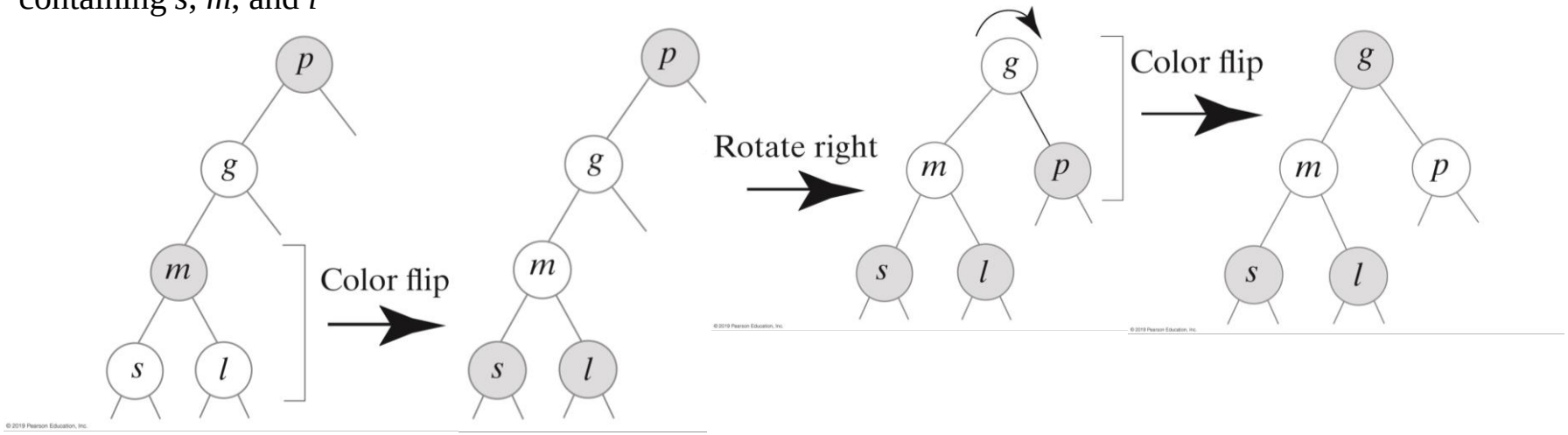
Splitting a 4-node

- FIGURE 28-35 Splitting a 4-node that has a red parent within a red-black tree: Case 1



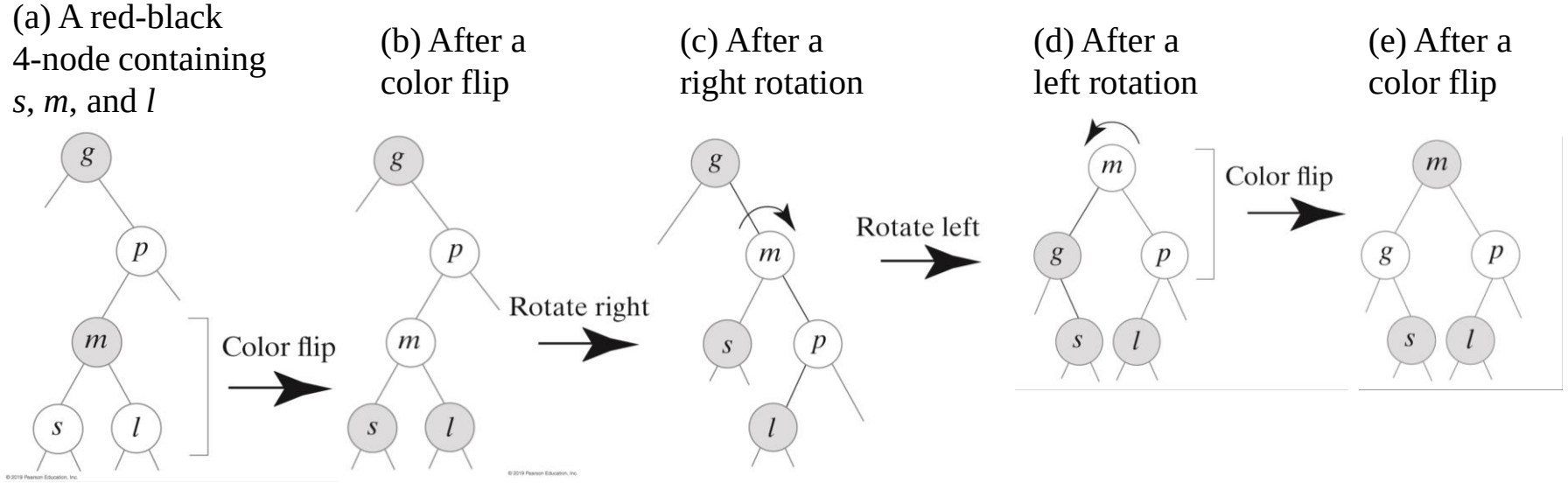
Splitting a 4-node

- FIGURE 28-36 Splitting a 4-node that has a red parent within a red-black tree: Case 2



Splitting a 4-node

- FIGURE 28-37 Splitting a 4-node that has a red parent within a red-black tree: Case 3



Splitting a 4-node

- FIGURE 28-38 Splitting a 4-node that has a red parent within a red-black tree: Case 4