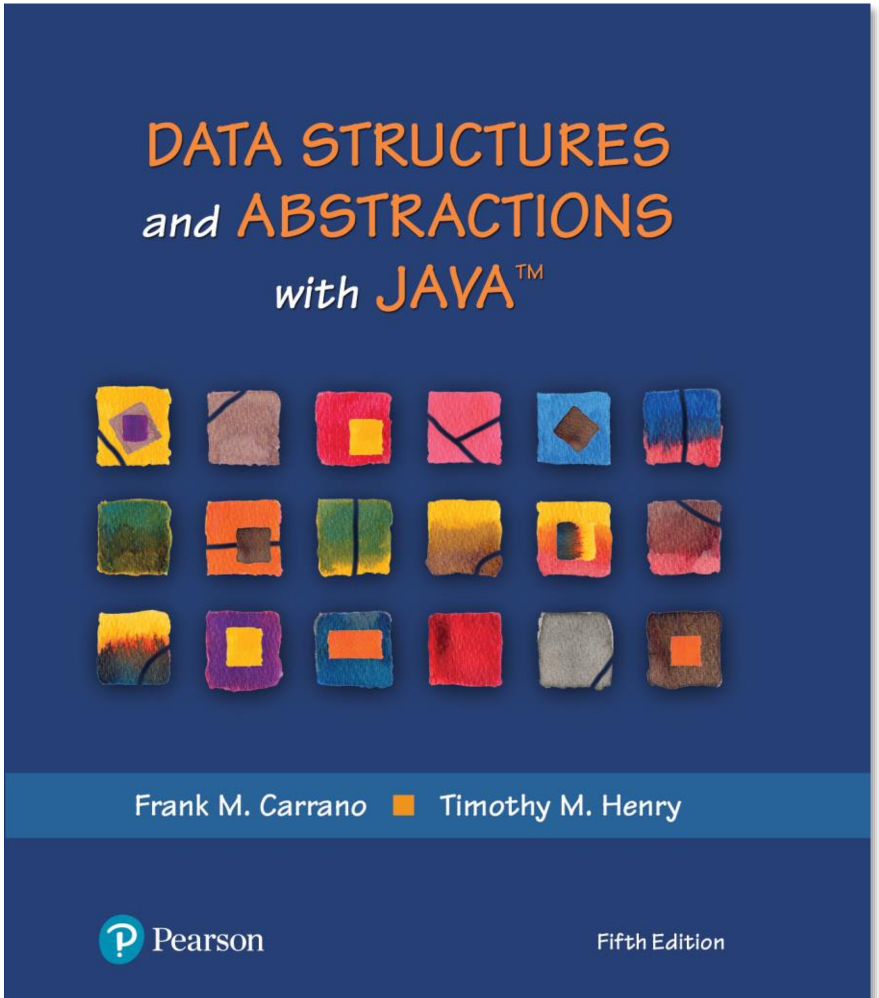
Data Structures and Abstractions with Java™

5th Edition



Chapter 28

Balanced Search Trees

AVL Trees

- Possible to form several differently shaped binary search trees
  - From the same collection of data
- AVL tree is a binary search tree that
  - Rearranges its nodes whenever it becomes unbalanced.

Single AVL Tree Rotations

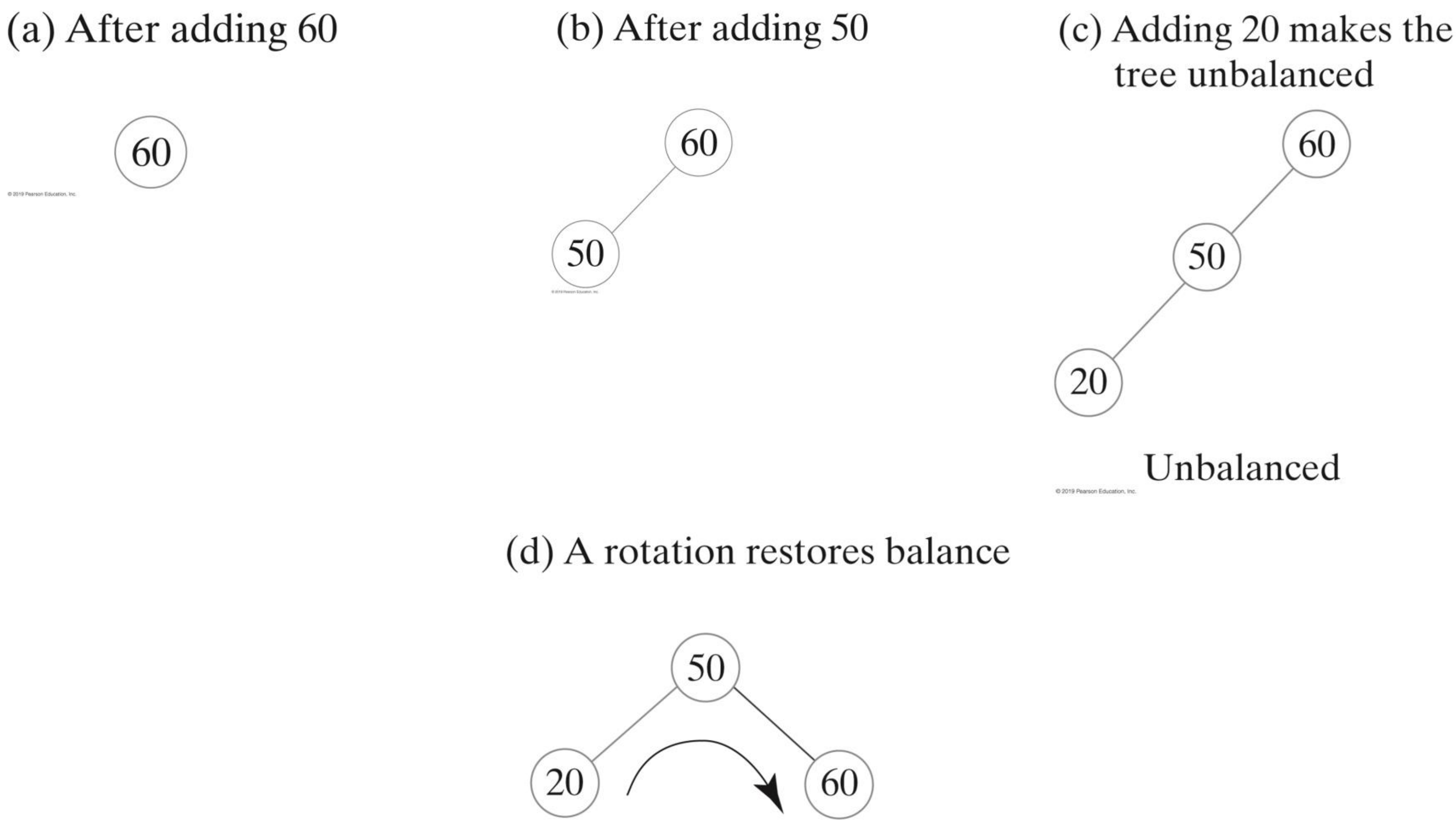


FIGURE 28-1 Additions to an initially empty AVL tree

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

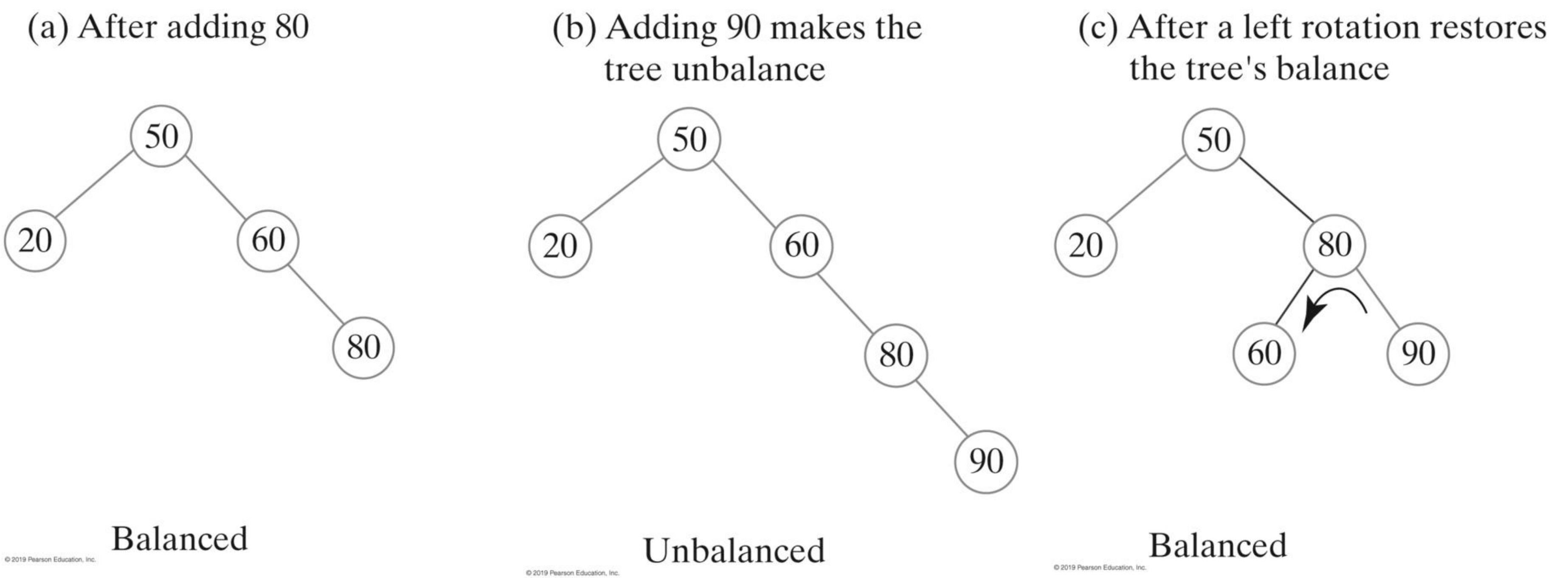


FIGURE 28-2 Additions to the AVL tree in Figure 28-1

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

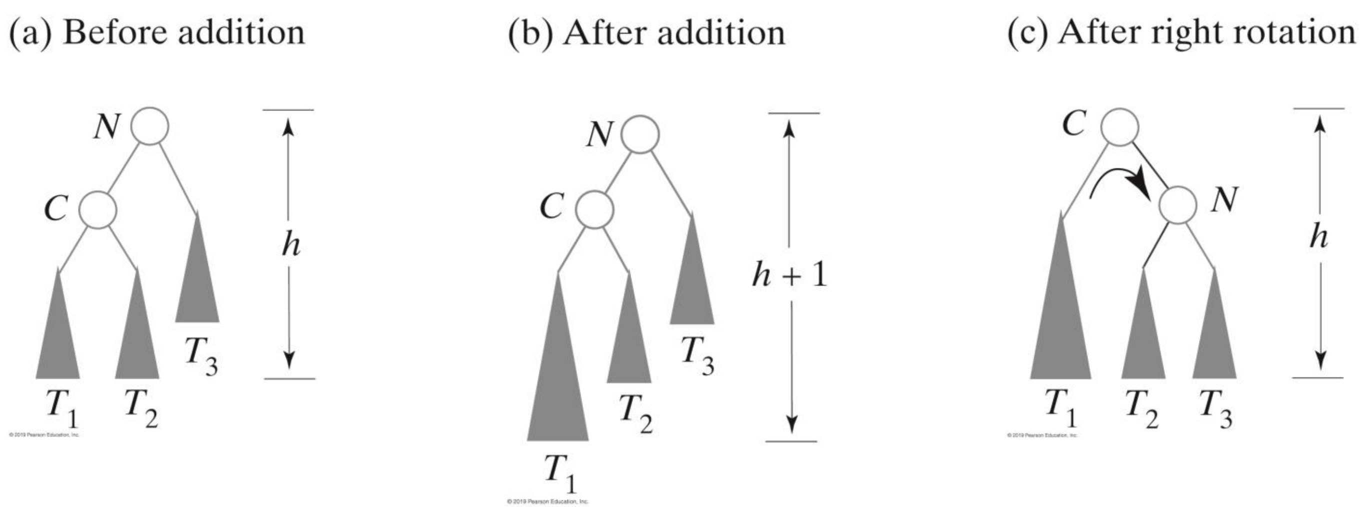


FIGURE 28-3 Before and after an addition to an AVL subtree that requires a right rotation to maintain its balance

Single Rotation

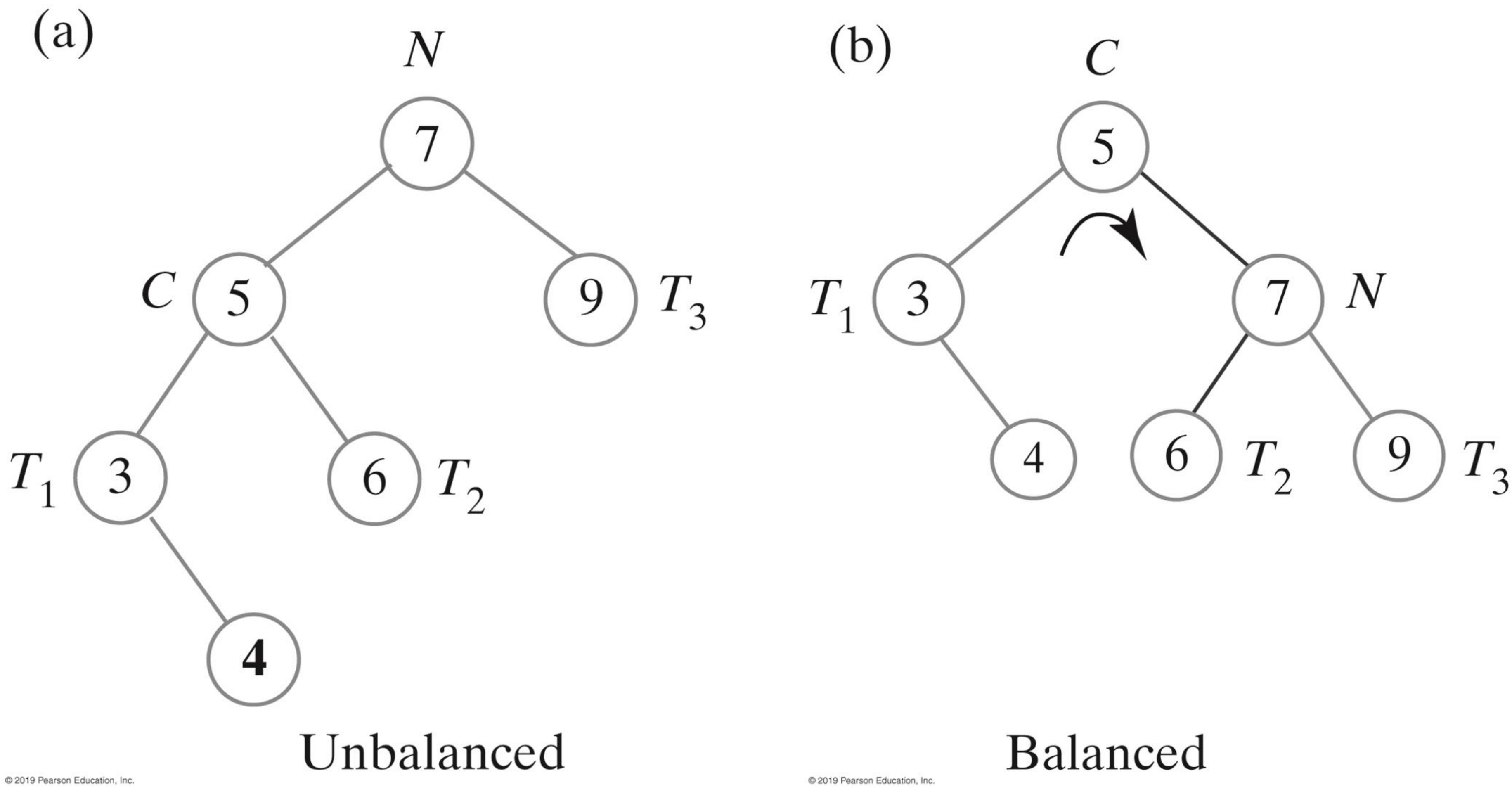


FIGURE 28-4 Before and after a right rotation restores balance to an AVL tree

Single Rotation

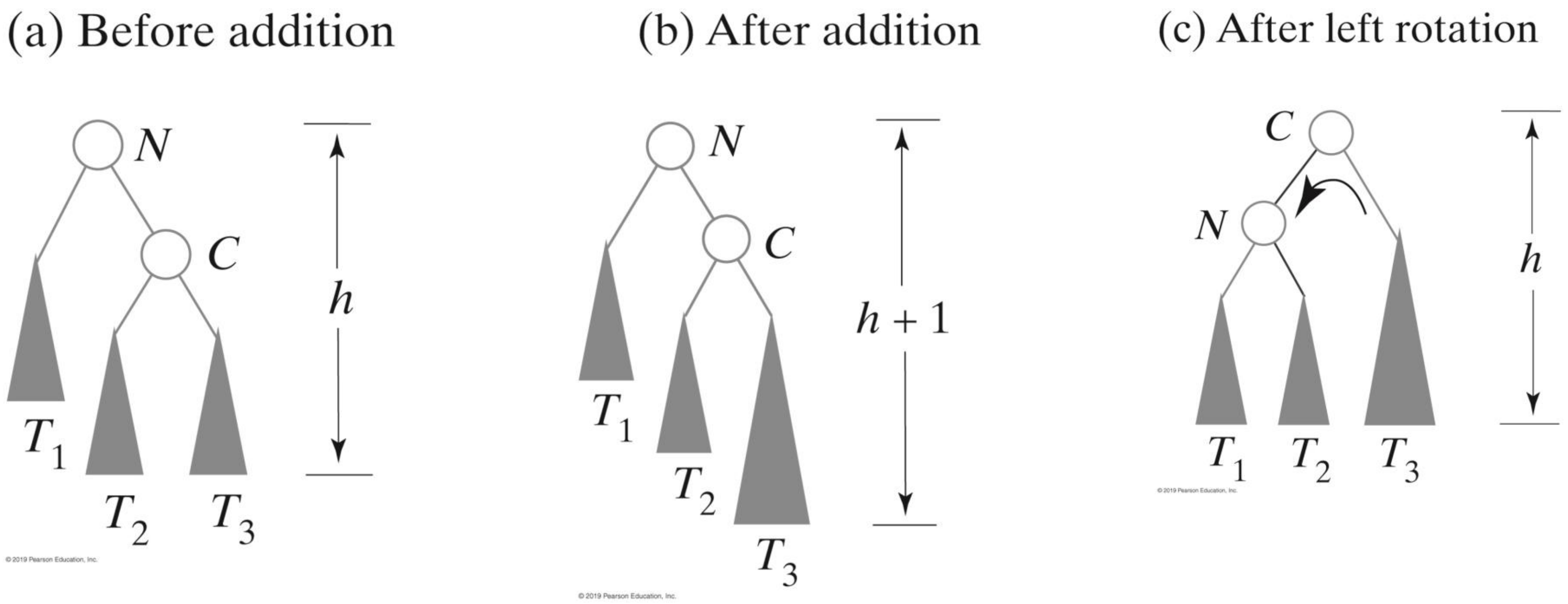


FIGURE 28-5 Before and after an addition to an AVL subtree that requires a left rotation to maintain its balance

Double Rotation

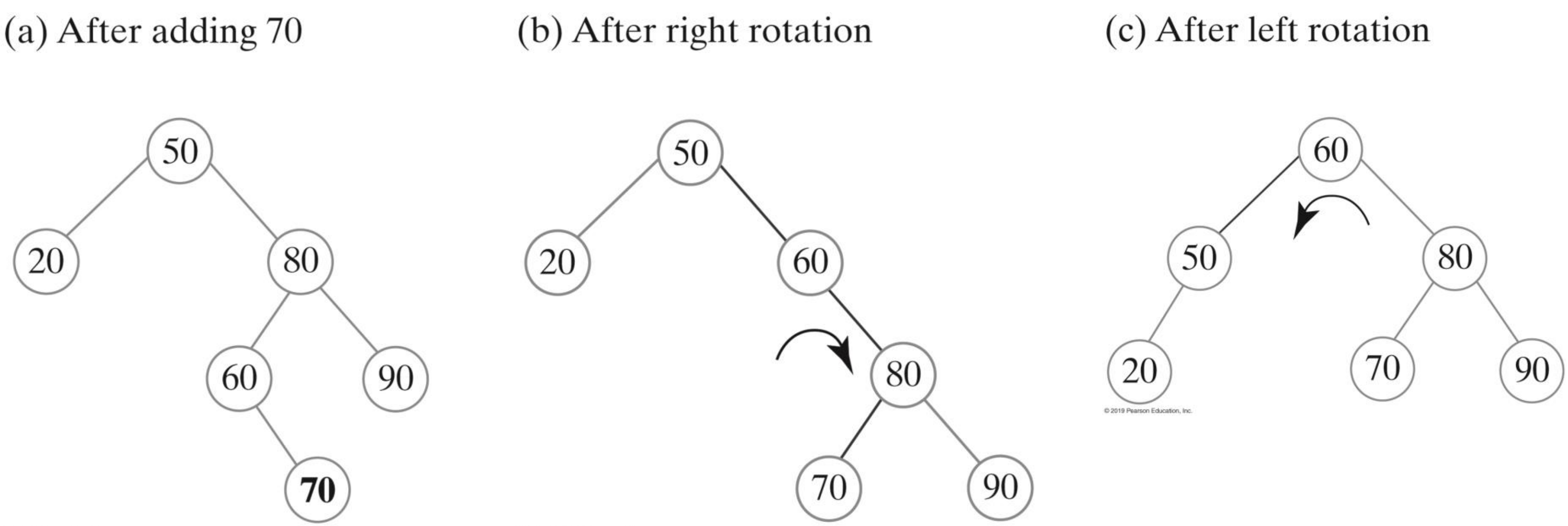


FIGURE 28-6 Adding 70 to the AVL tree in Figure 28-2c requires both a right rotation and a left rotation to maintain its balance

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

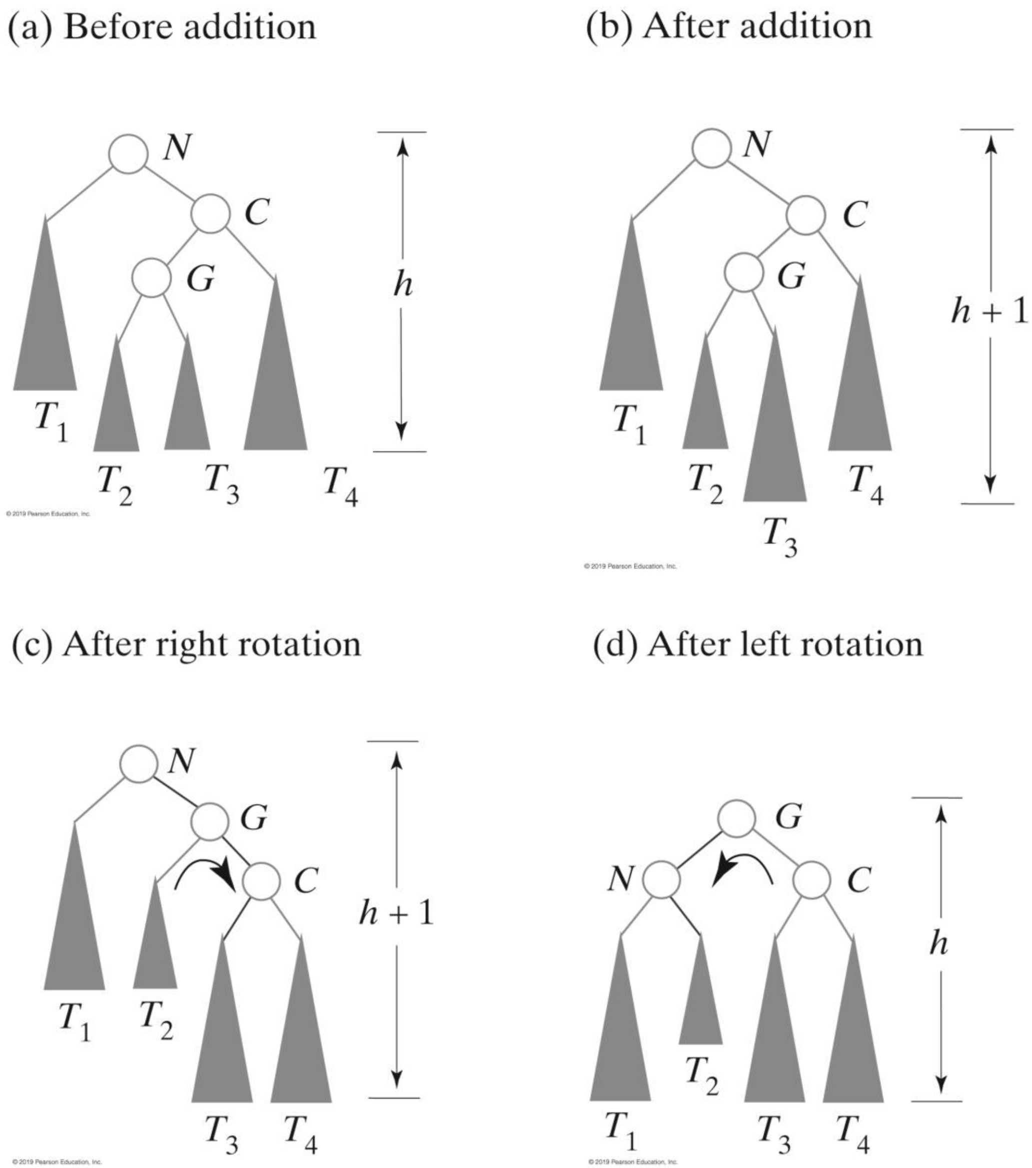


FIGURE 28-7 Before and after an addition to an AVL subtree that requires both a right rotation and a left rotation to maintain its balance

Double Rotations

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)

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

Double Rotations

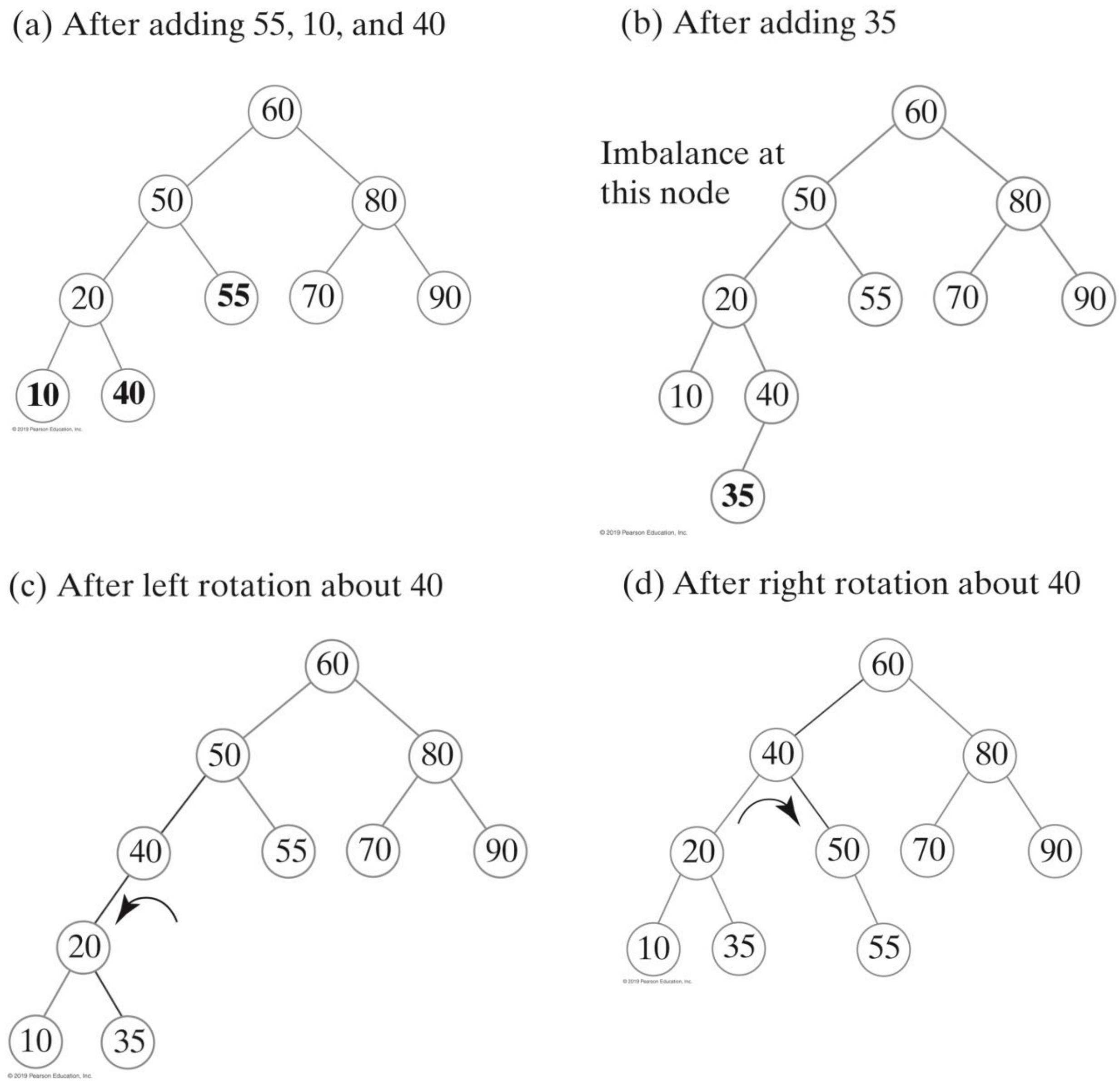


FIGURE 28-8 Adding 55, 10, 40, and 35 to the AVL tree in Figure 28-6c

Double Rotations

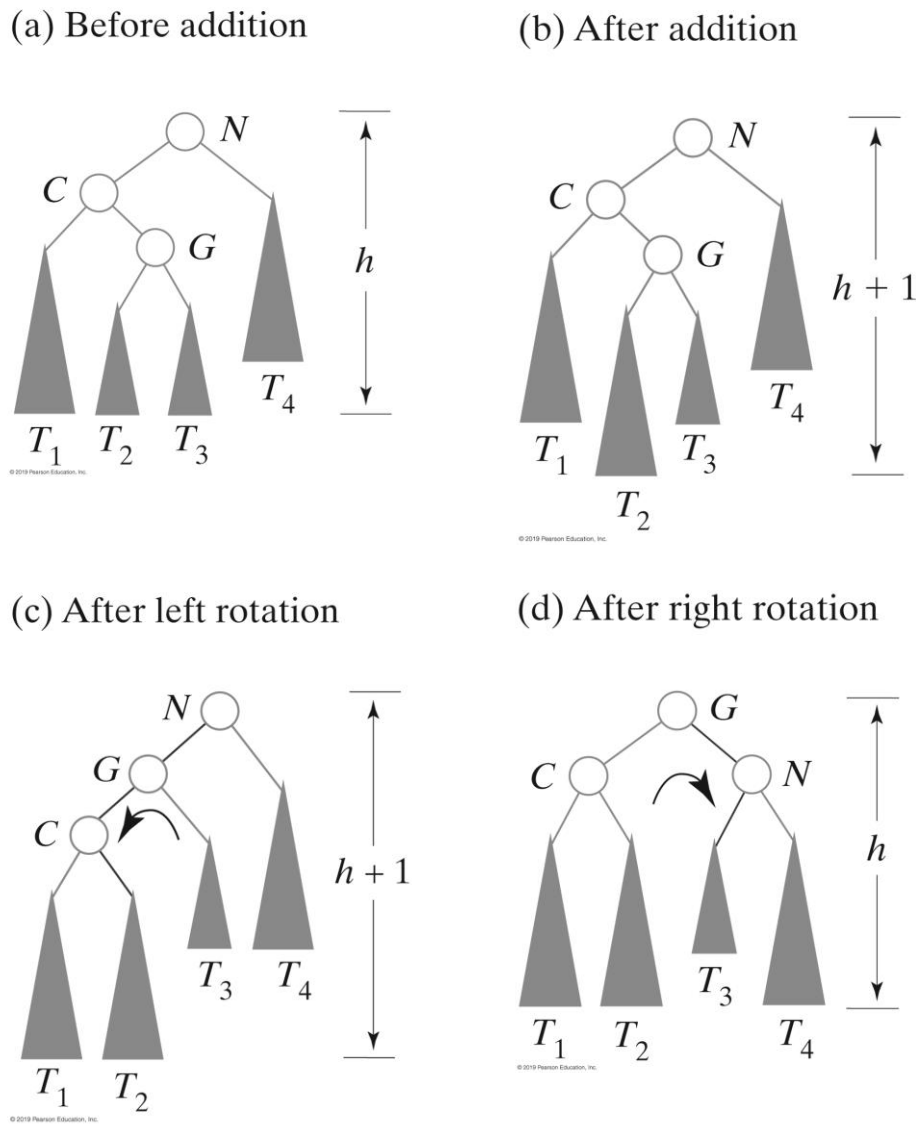


FIGURE 28-9 Before and after an addition to an AVL subtree that requires both a left rotation and a right rotation to maintain its balance

Left-right Double Rotations

Algorithm **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)

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

An AVL Tree Versus a Binary Search Tree

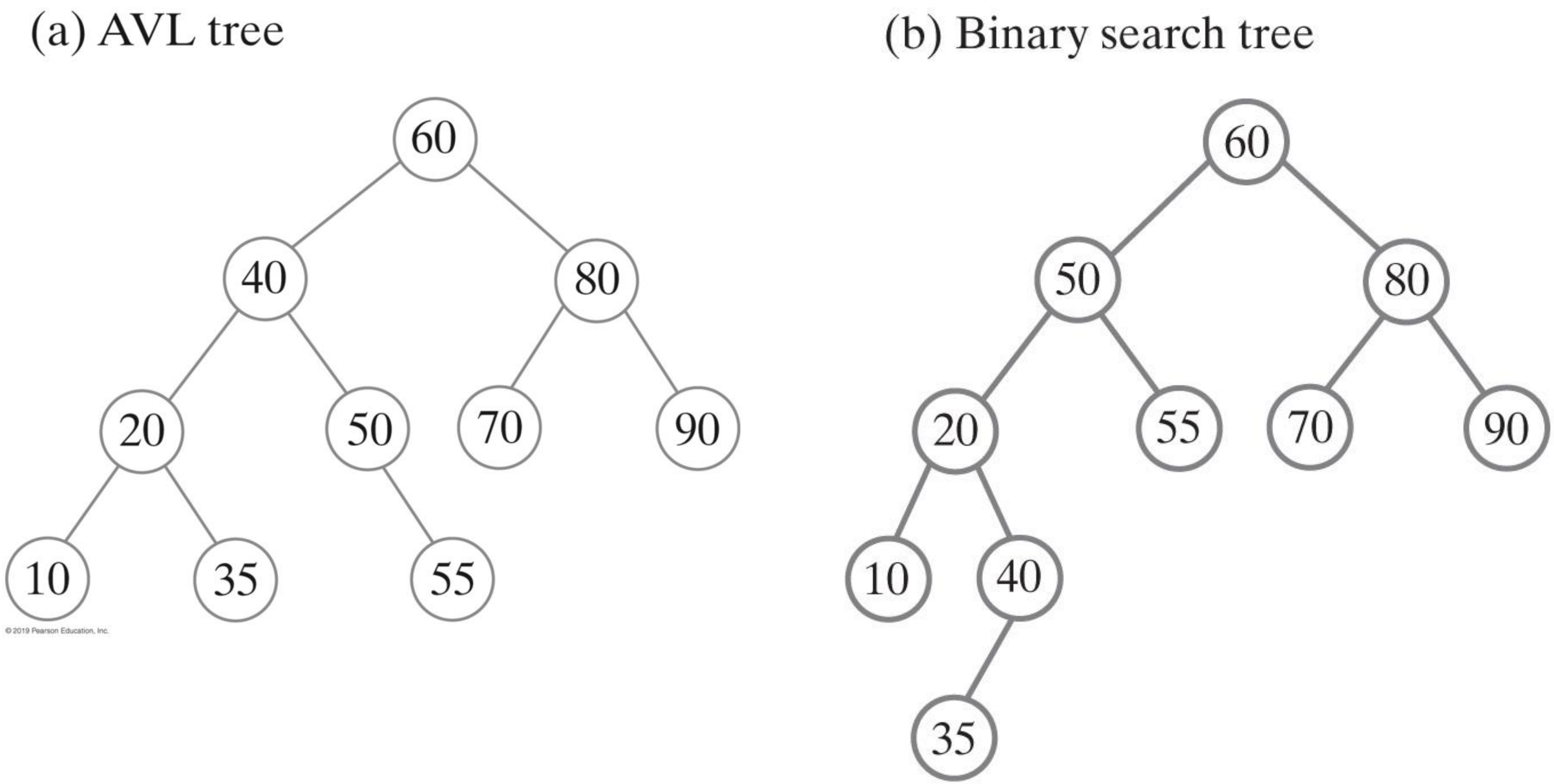


FIGURE 28-10 The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty AVL tree and a binary search tree

Implementation Details

```java
/**  A class that implements the ADT AVL tree by extending BinarySearchTree.
  The remove operation is not supported.  */
public class AVLTree<T extends Comparable<? super T>>
         extends BinarySearchTree<T> implements SearchTreeInterface<T>
{
      public AVLTree()
      {
      super();
      } // end default constructor

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

/*  Implementations of add and remove are here.
    A definition of add appears in Segment 28.12 of
    this chapter. Other methods in SearchTreeInterface are inherited.
  . . .
    Implementations of private methods to rebalance the tree
    using rotations are here.
  . . . */
} // end AVLTree
```

LISTING 28-1 An outline of the class AVLTree

Rotations

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

```
// Corrects an imbalance at the node closest to a structural
// change in the left 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.

private BinaryNode<T> rotateRight(BinaryNode<T> nodeN)
{
  BinaryNode<T> nodeC = nodeN.getLeftChild();
  nodeN.setLeftChild(nodeC.getRightChild());
  nodeC.setRightChild(nodeN);
  return nodeC;
} // end rotateRight
```

The implementation of the method for a single right rotation

Rotations

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)

```
// 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.
private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN)
{
  BinaryNode<T> nodeC = nodeN.getRightChild();
  nodeN.setRightChild(rotateRight(nodeC));
  return rotateLeft(nodeN);
} // end rotateRightLeft
```

Implementation for a right- left double rotation

Rotations

```
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
}
```

Pseudocode to rebalance the tree

Rotations

```
private BinaryNode<T> rebalance(BinaryNode<T> nodeN)
{
  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);
  } // end if
  // Else nodeN is balanced
  return nodeN;
} // end rebalance
```

Implementation for rebalancing within the class `AVLTree`

Methods to Add

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

   if (isEmpty())
      setRootNode(new BinaryNode<>(newEntry));
   else
   {
      BinaryNode<T> rootNode = getRootNode();
      result = addEntry(rootNode, newEntry);
      setRootNode(rebalance(rootNode));
   } // end if

   return result;
} // end add
```

AVL Tree Method `add`

Methods to Add — `addEntry` (Part 1)

```
private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
  // Assertion: rootNode != null
  T result = null;
  int comparison = newEntry.compareTo(rootNode.getData());

  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
```

AVL Tree Method `addEntry`

Methods to Add — `addEntry` (Part 1)

```
      else
         rootNode.setLeftChild(new BinaryNode<>(newEntry));
   }
   else
   {
      // Assertion: comparison > 0

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

   return result;
} // end addEntry
```

AVL Tree Method `addEntry`

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

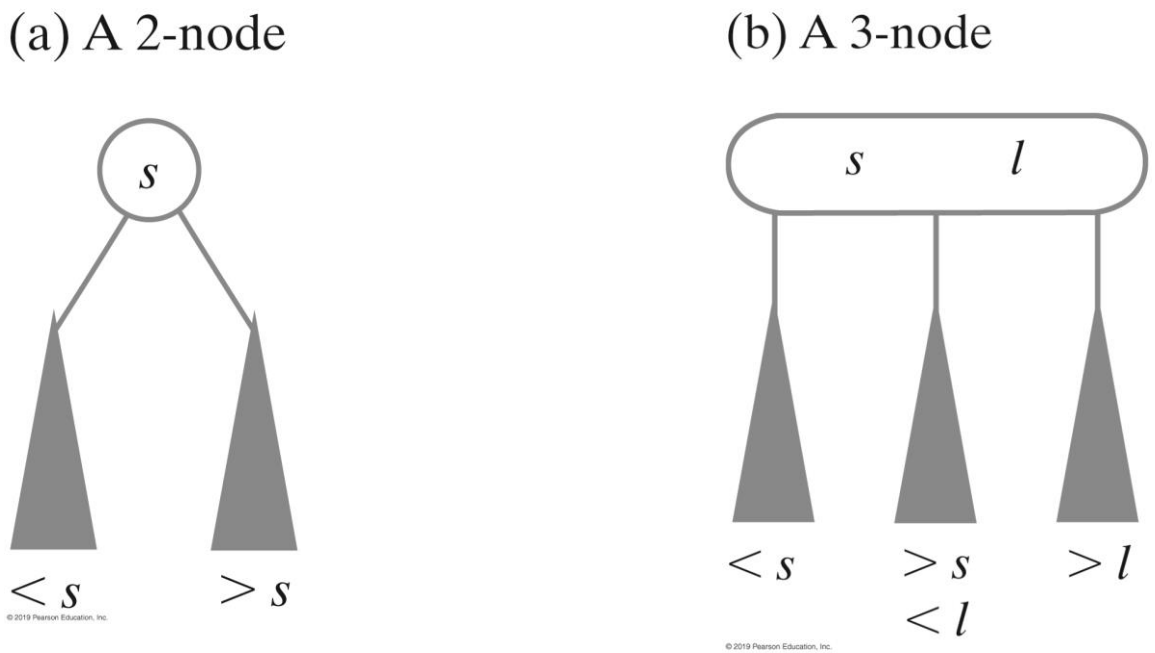


FIGURE 28-11 Nodes in a 2-3 tree

2-3 Trees

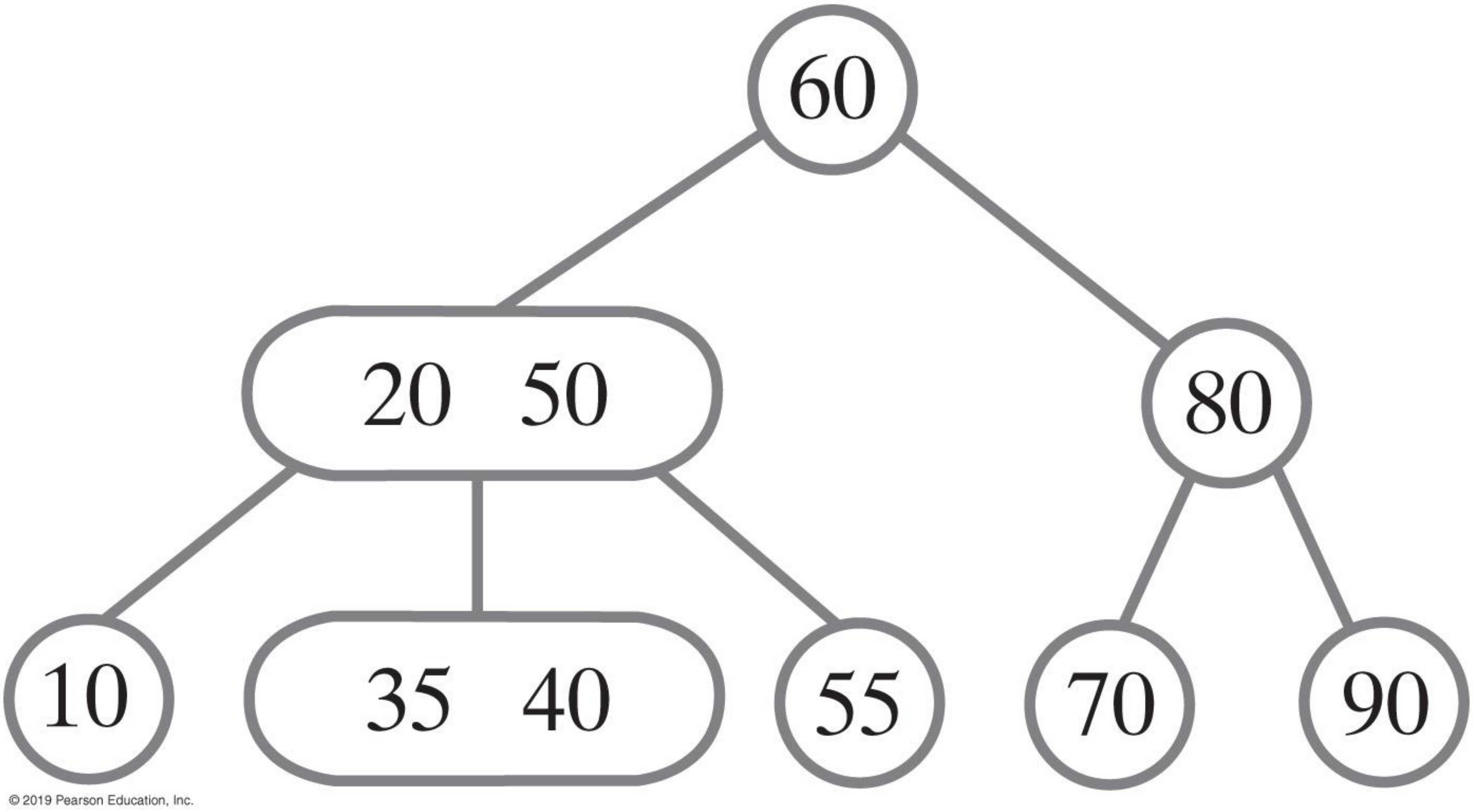


FIGURE 28-12 A 2-3 tree

Building 2-3 Trees

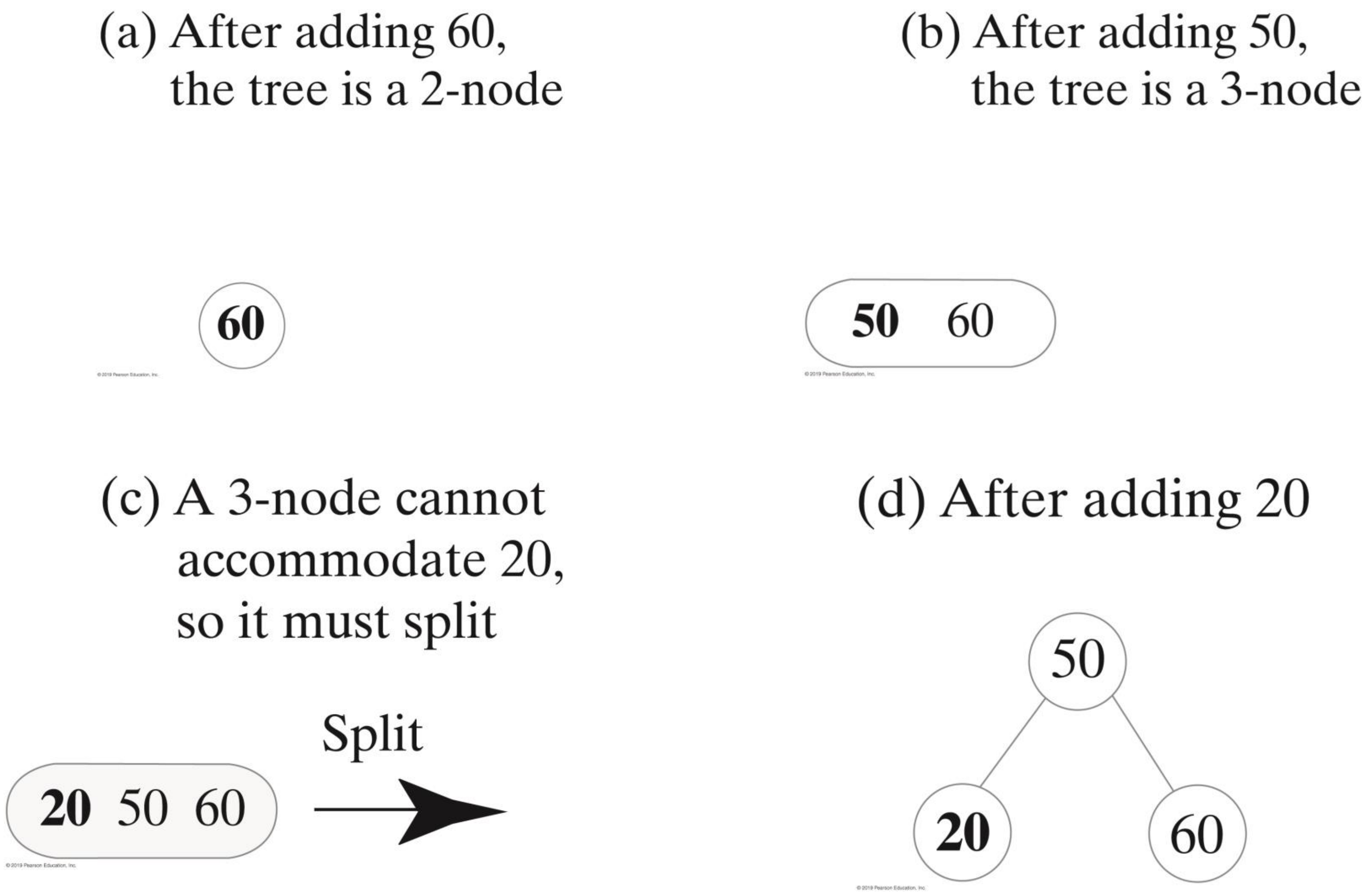
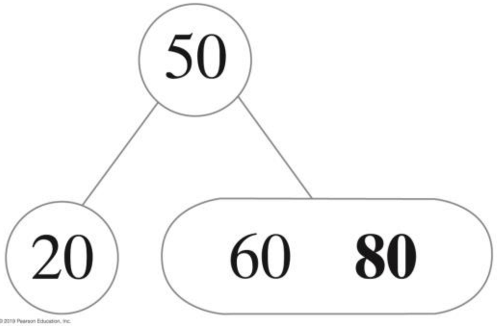


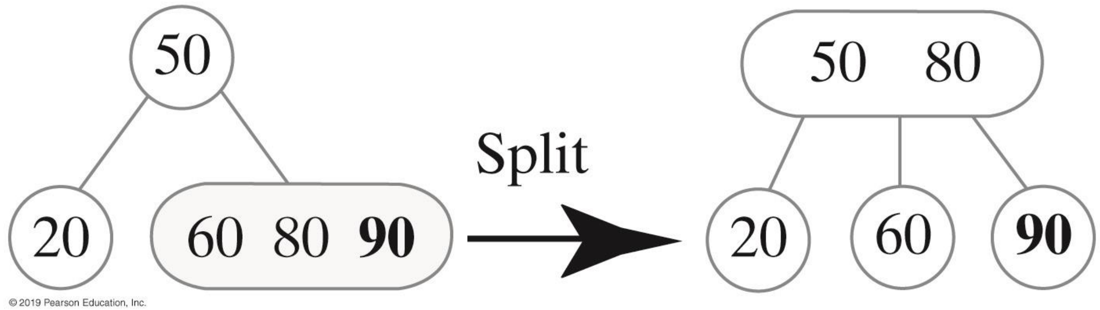
FIGURE 28-13 An initially empty 2-3 tree after three additions

Building 2-3 Trees

(a) After adding 20



(b) Splitting the leaf and adding 90



(c) After adding 70

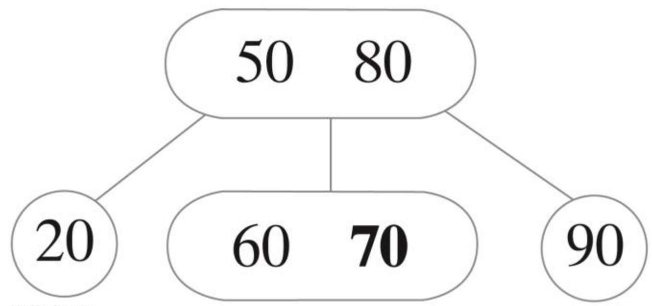
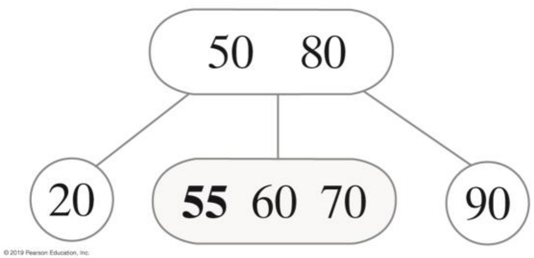


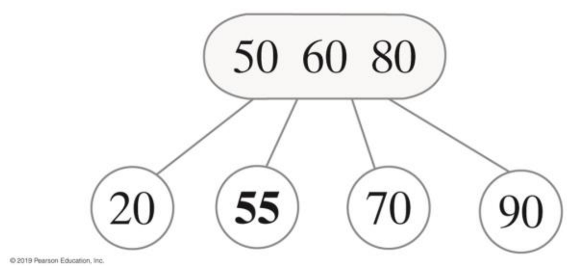
FIGURE 28-14 The 2-3 tree after three additions

Building 2-3 Trees

(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

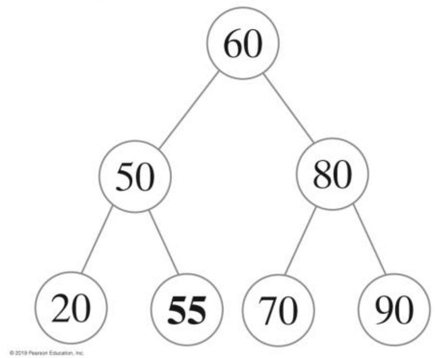


FIGURE 28-15 Adding 55 to the 2-3 tree in Figure 28-14c causes a leaf and then the root to split

Building 2-3 Trees

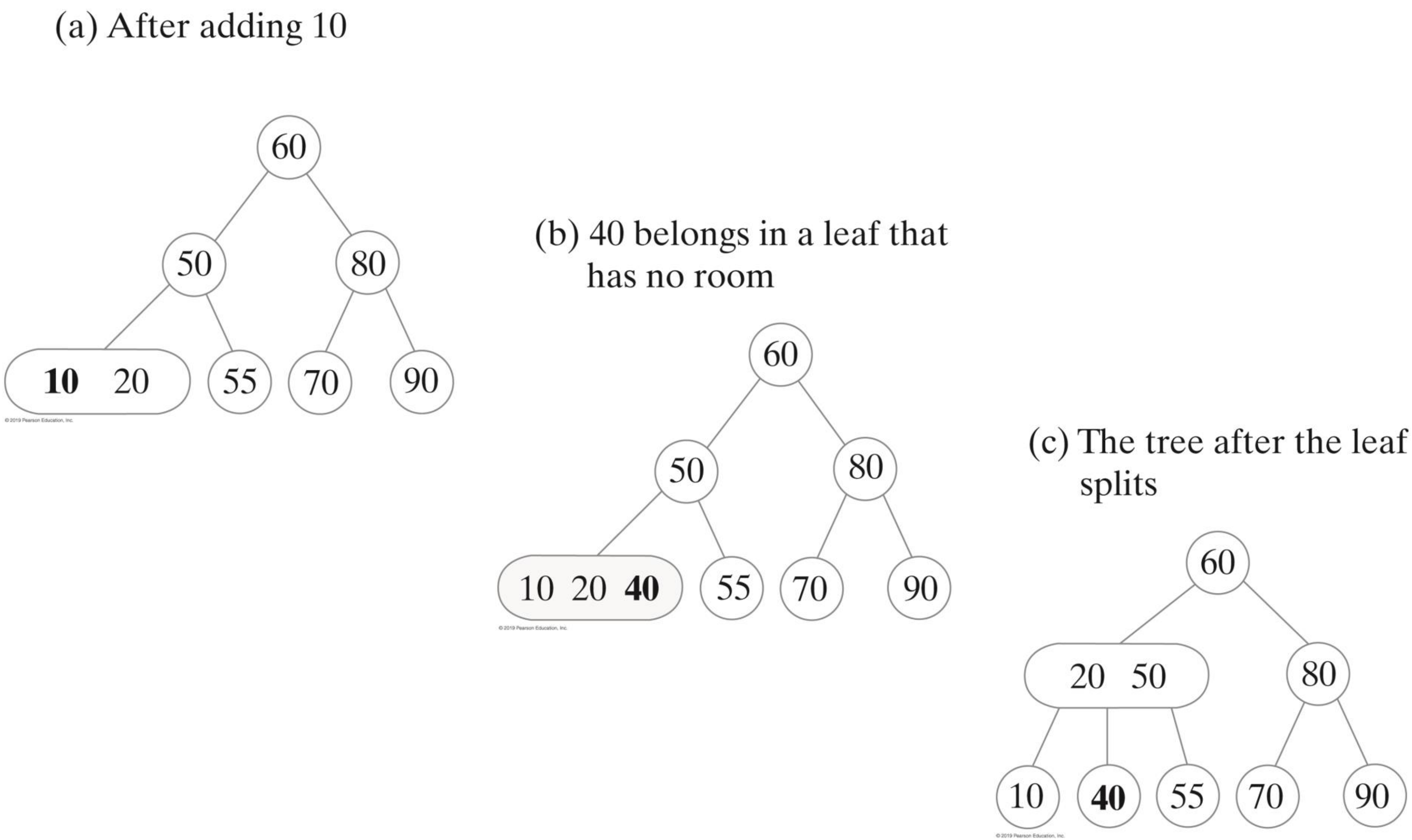


FIGURE 28-16 Adding 10 and 40 to the 2-3 tree in Figure 28-15c

Building 2-3 Trees

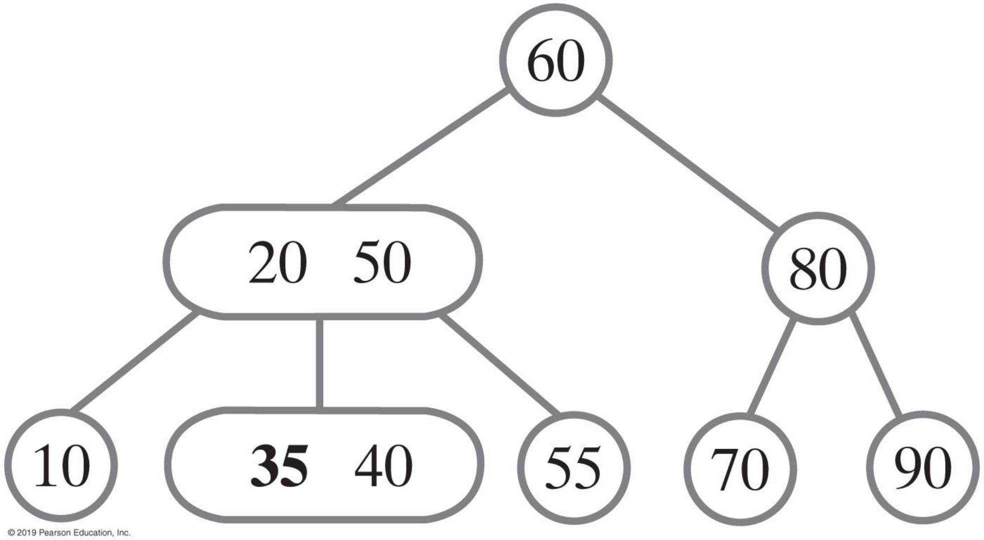


FIGURE 28-17 The 2-3 tree in Figure 28-16c after adding 35

Splitting Nodes During Addition

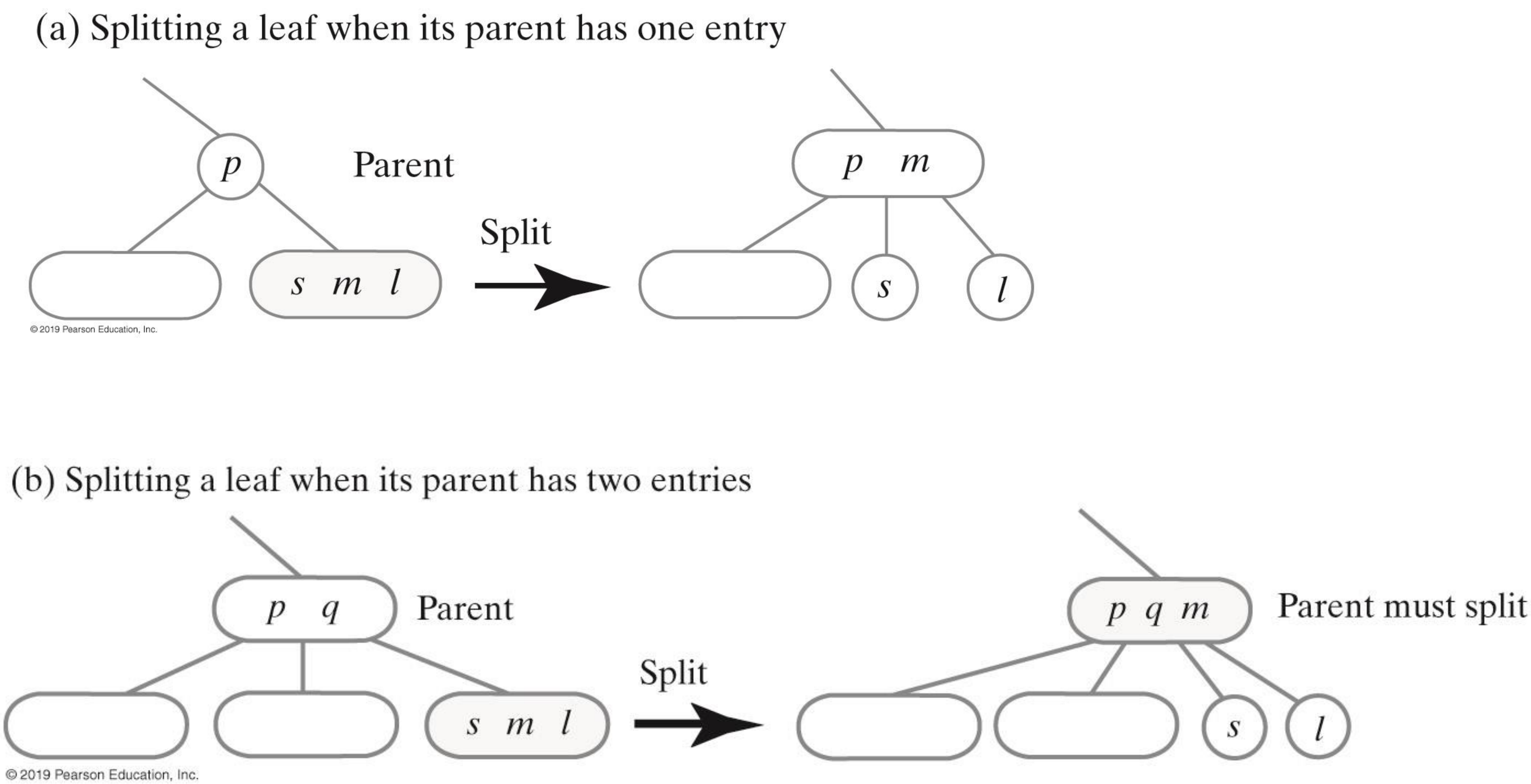


FIGURE 28-18 Splitting a leaf to accommodate a new entry

Splitting Nodes During Addition

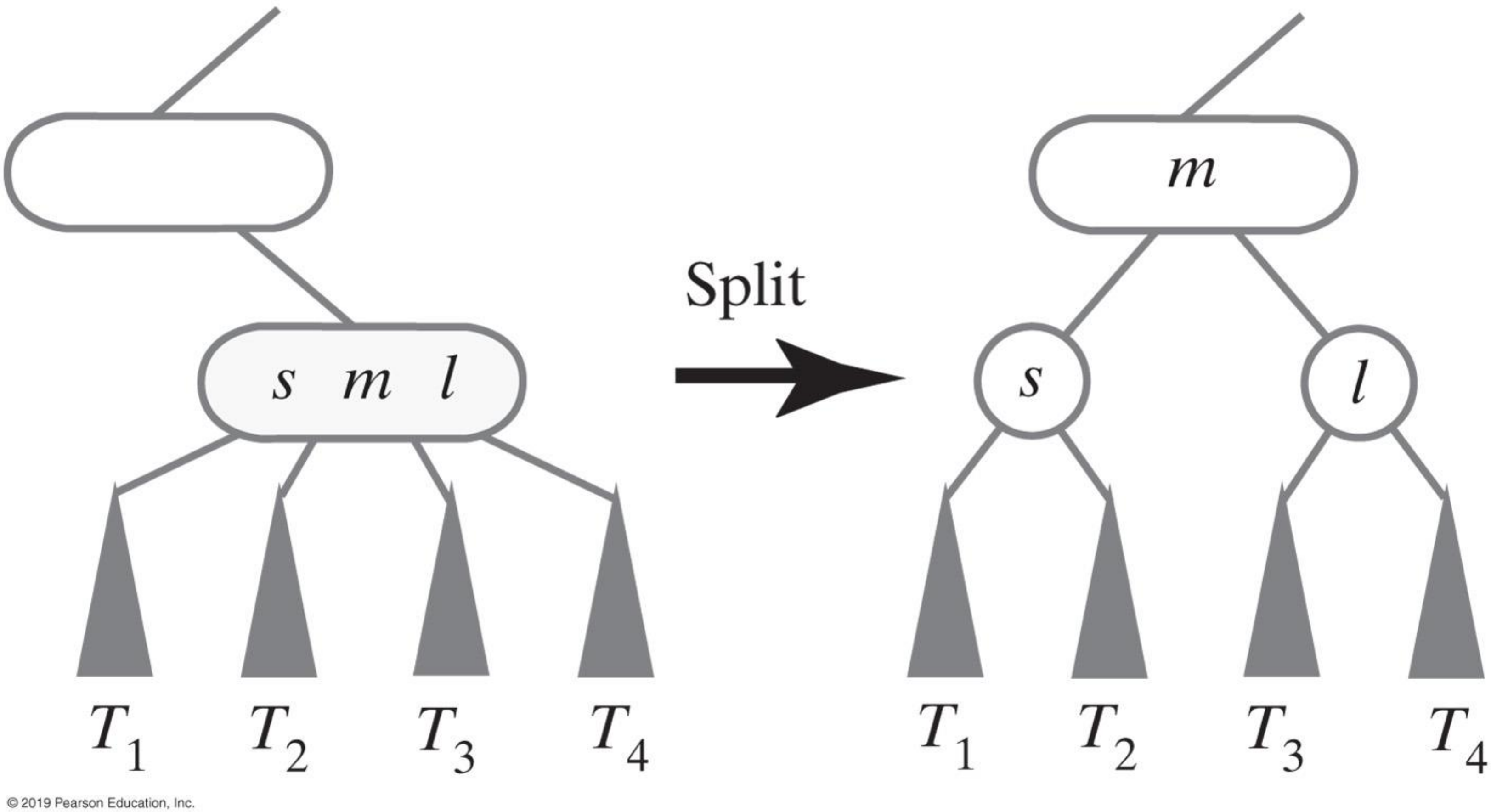


FIGURE 28-19 Splitting an internal node to accommodate a new entry

Splitting Nodes During Addition

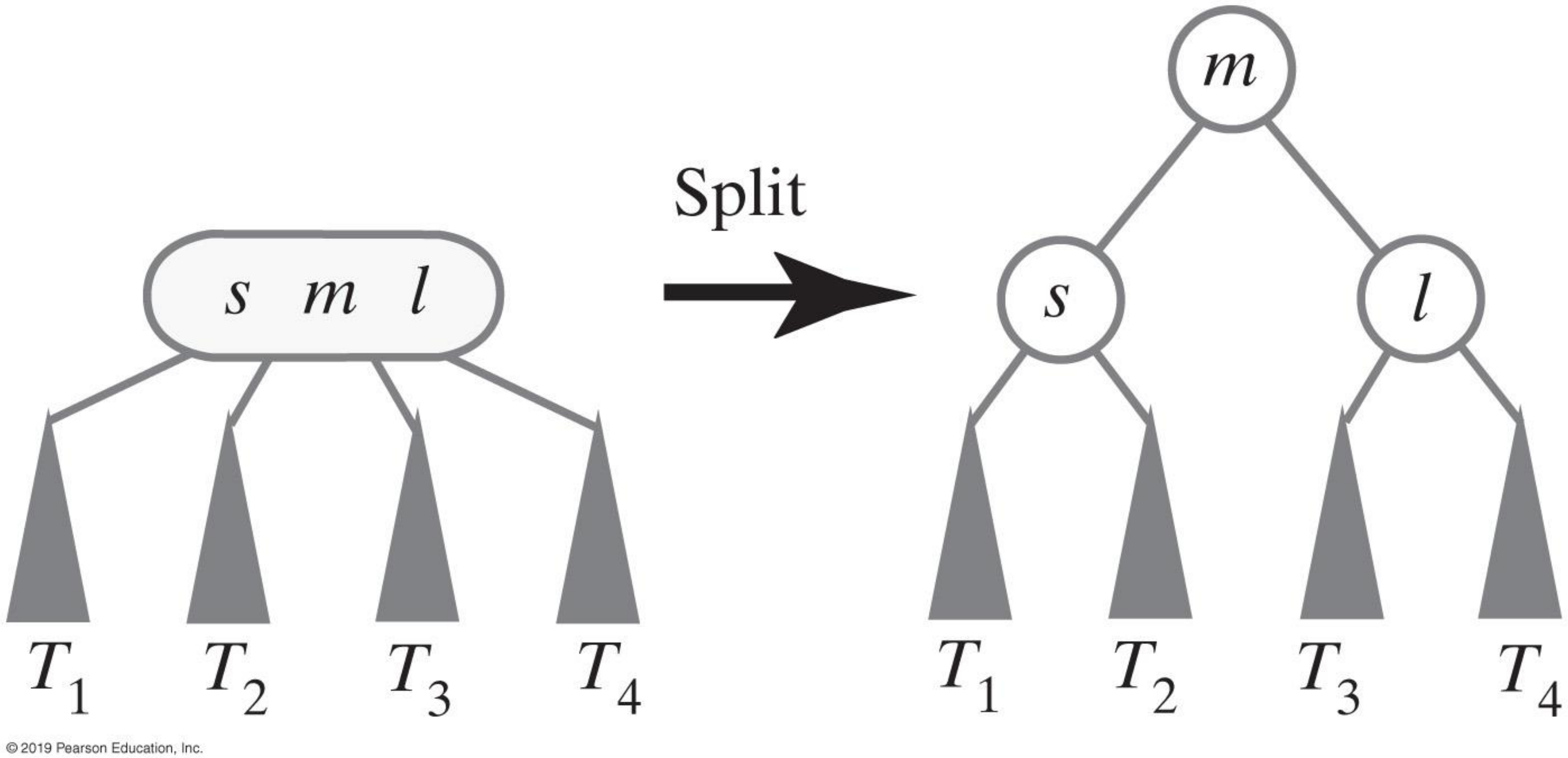


FIGURE 28-20 Splitting the root to accommodate a new entry

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.

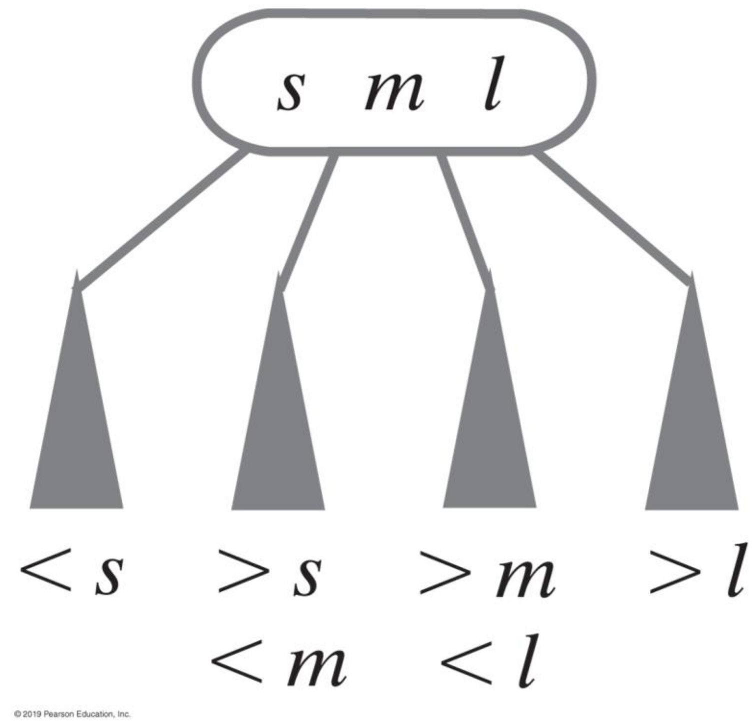


FIGURE 28-21 A 4-node

Adding Entries to a 2-4 Tree

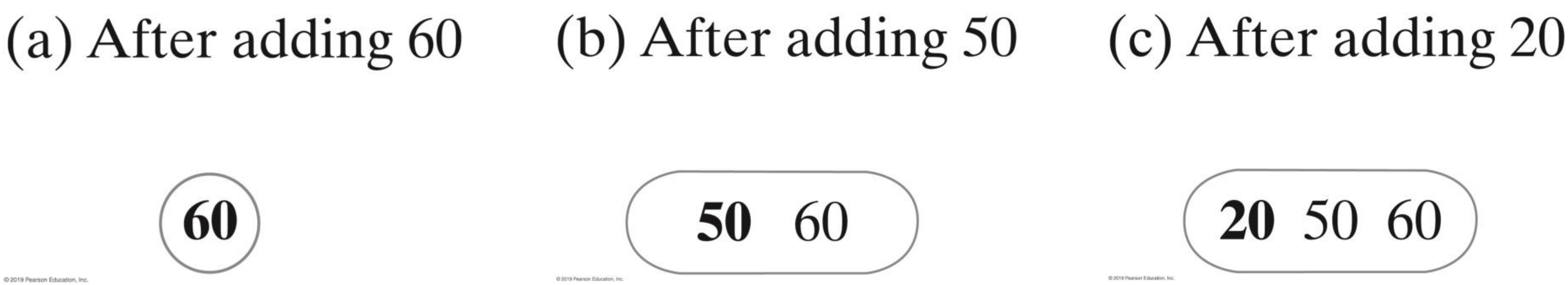


FIGURE 28-22 Adding 60, 50, and 20 to an initially empty 2-4 tree

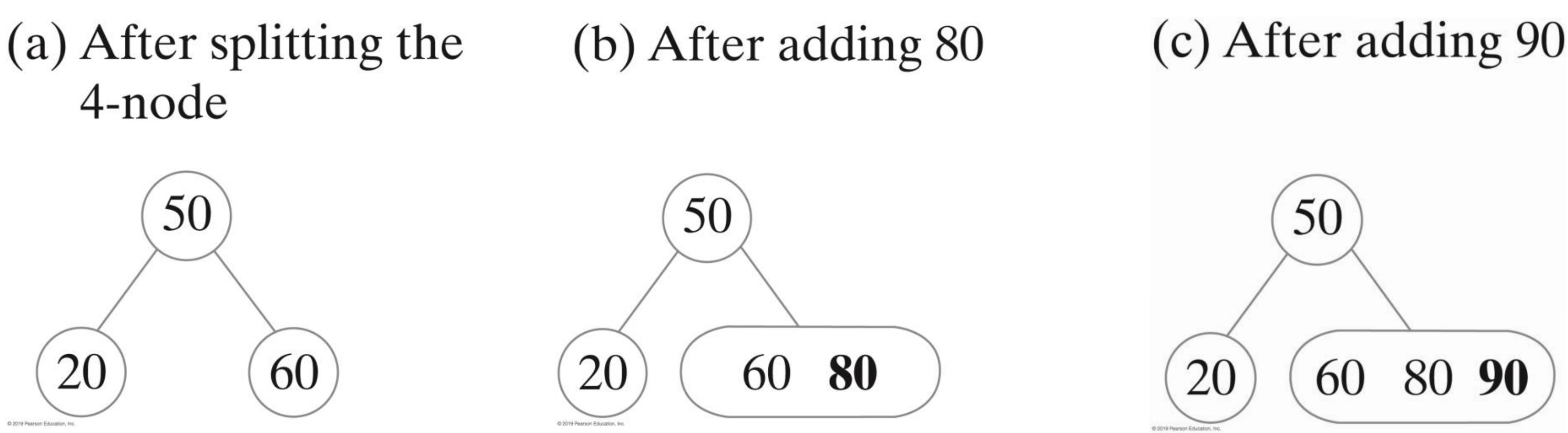
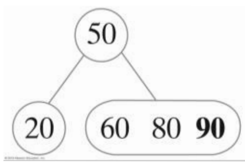


FIGURE 28-23 Adding 80 and 90 to the tree in Figure 28-22c

Adding Entries to a 2-4 Tree



(a) After splitting the 4-node leaf

(b) After adding 70

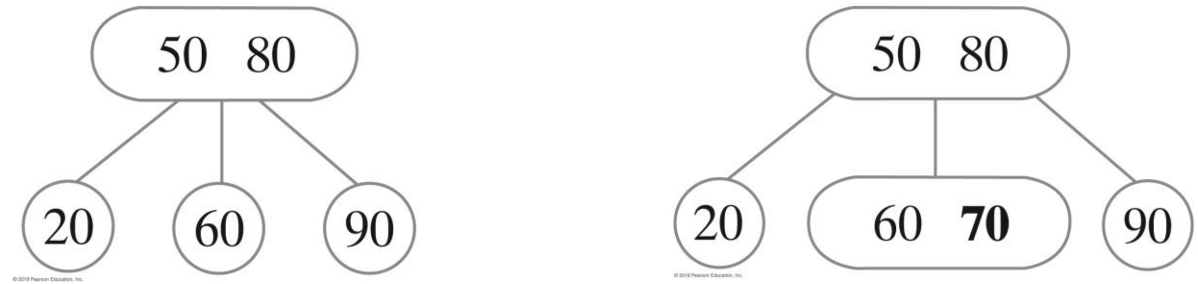


FIGURE 28-24a Adding 70 to the 2-4 tree in Figure 28-23

(a) After adding 55

(b) After adding 10

(c) After adding 40

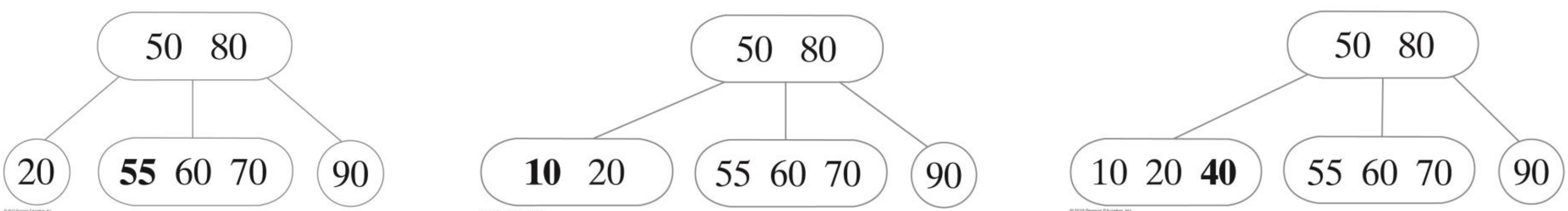
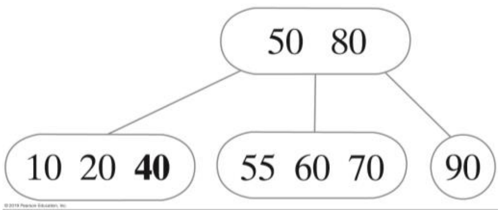
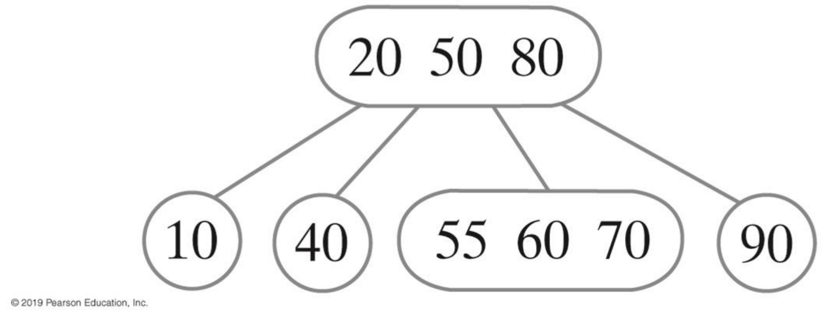


FIGURE 28-25 Adding 55, 10, and 40 to the 2-4 tree in Figure 28-24b

Adding Entries to a 2-4 Tree



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



(b) After adding 35

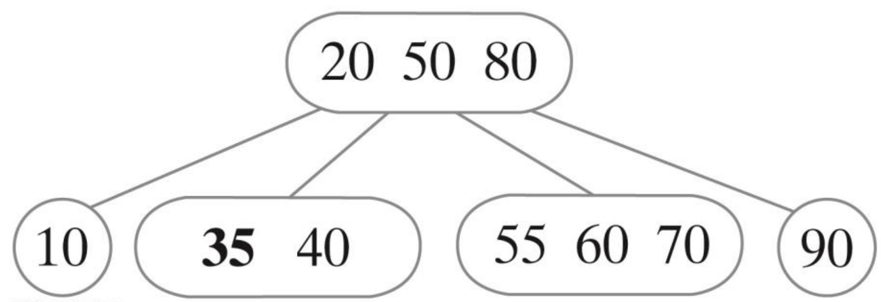


FIGURE 28-26 Adding 35 to the 2-4 tree in Figure 28-25c

Building 2-4 Trees - A Comparison

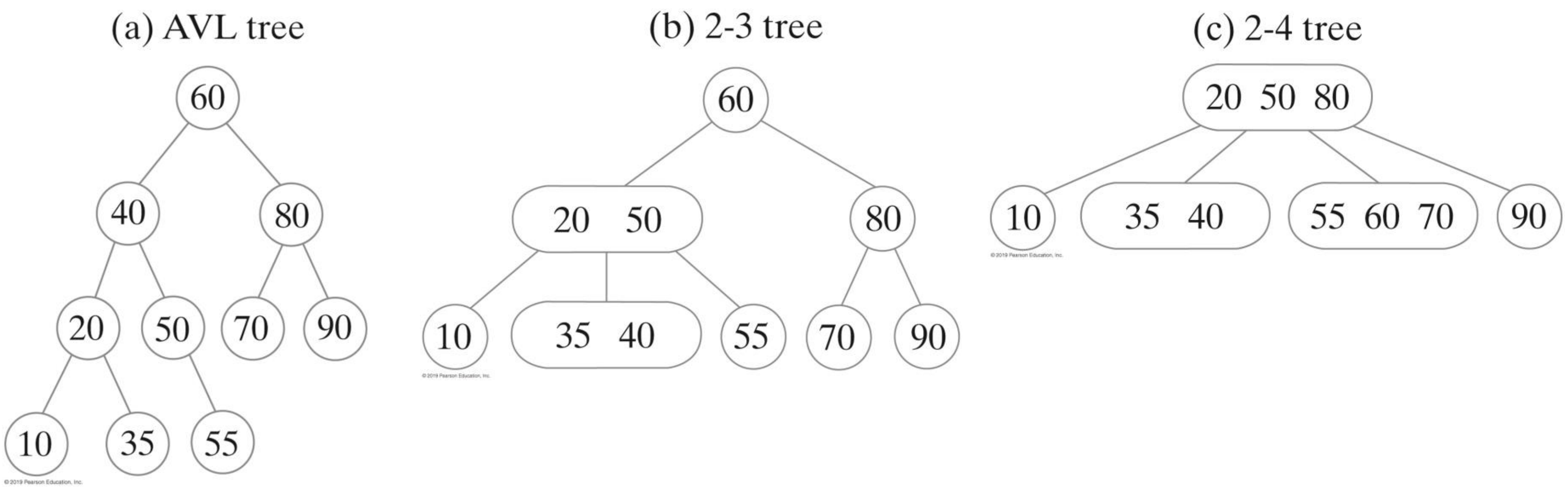
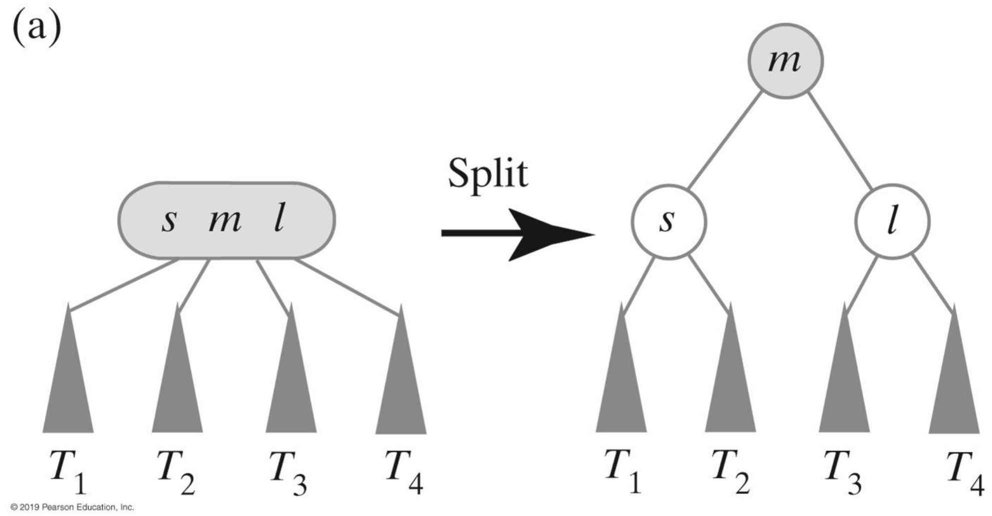


FIGURE 28-28 Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35

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



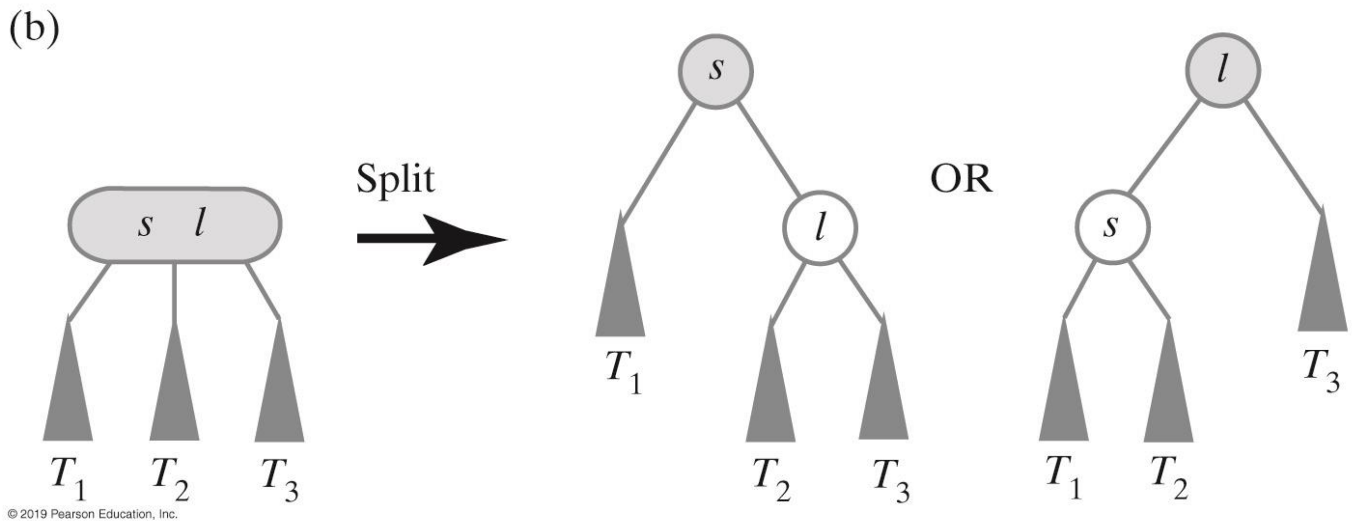


FIGURE 28-28 Using 2-nodes to represent (a) a 4-node; (b) a 3-node

Red-Black Trees

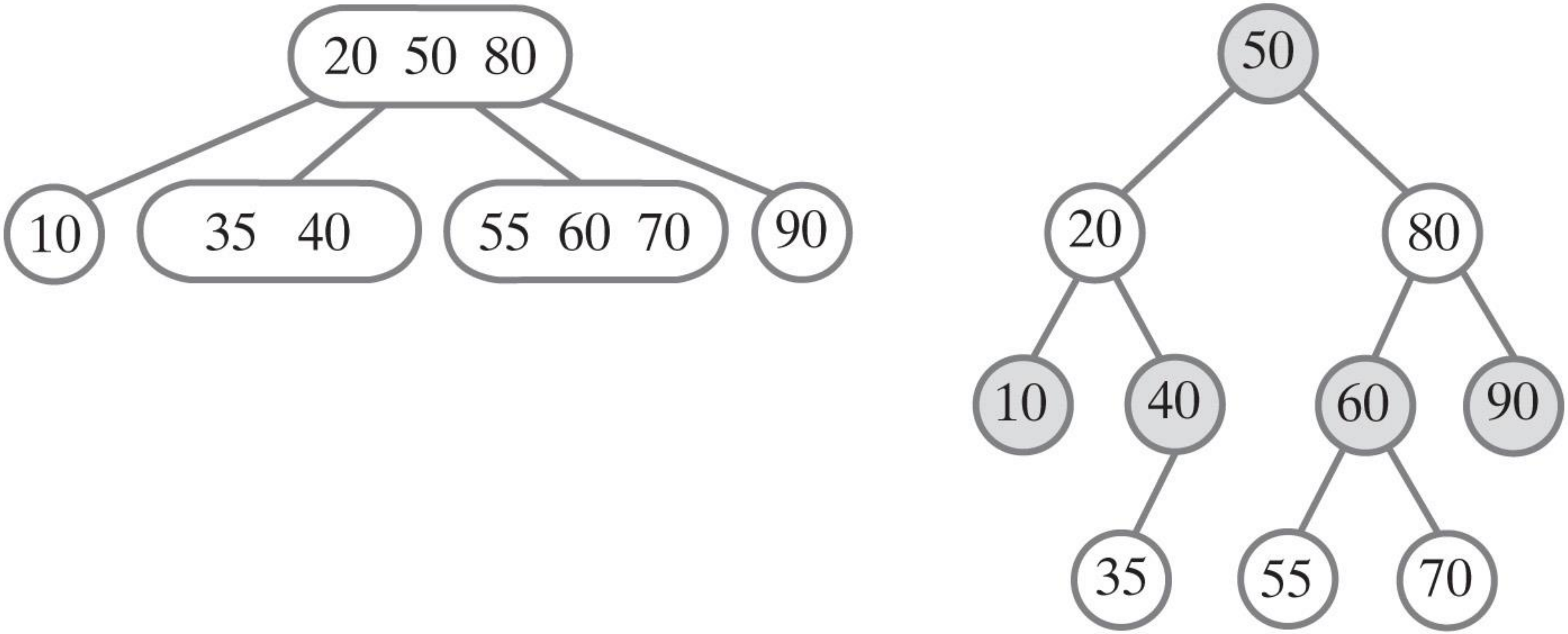


FIGURE 28-29 A 2-4 tree (Figure 28-28c) and its equivalent red-black tree

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.

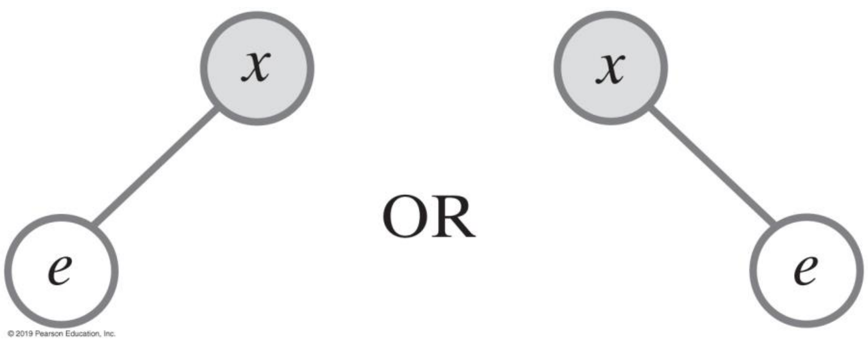
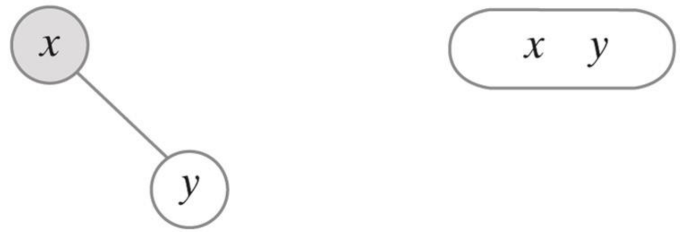


FIGURE 28-30 The result of adding a new entry e to a one-node red-black tree

Adding Entries to a Red-Black Tree

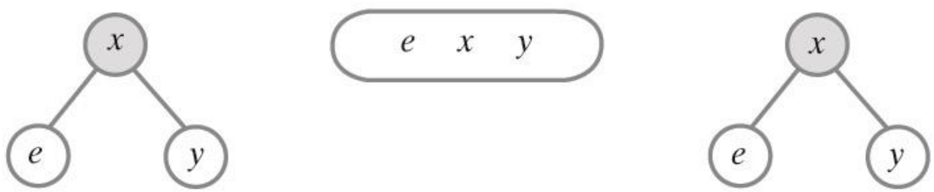
Red-black tree **Equivalent 2-4 tree**

(a) Before addition



After adding *e* to the red-black tree | **After adding *e* to the 2-4 tree** | **Red-black equivalent of the 2-4 tree** | **Action after addition to transform column 1 into column 3**

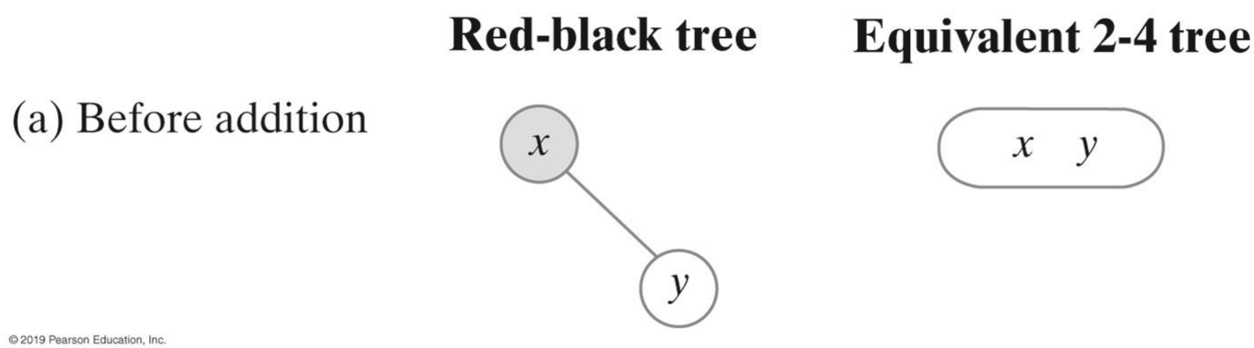
(b) Case 1:
The tree is balanced



None

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



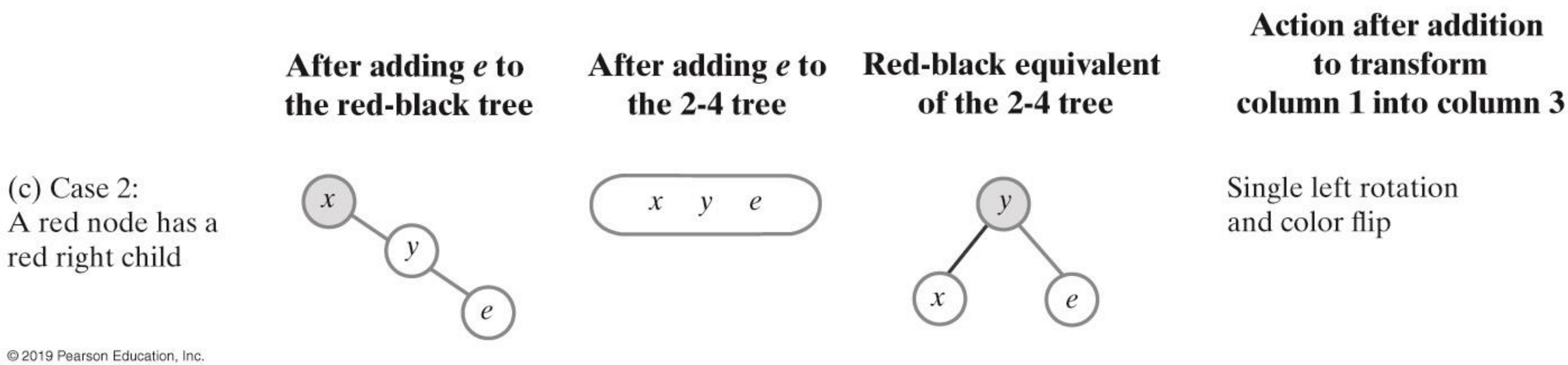
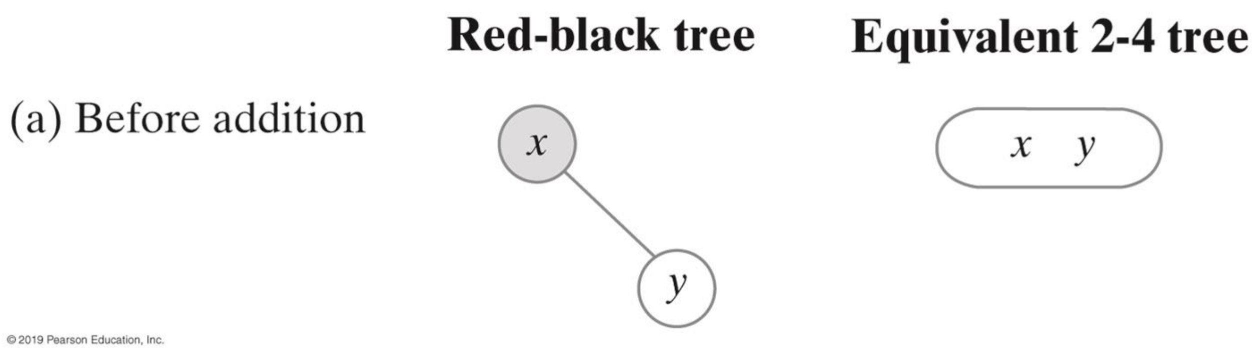


FIGURE 28-31c The possible results of adding a new entry *e* to a two-node red-black tree

Adding Entries to a Red-Black Tree



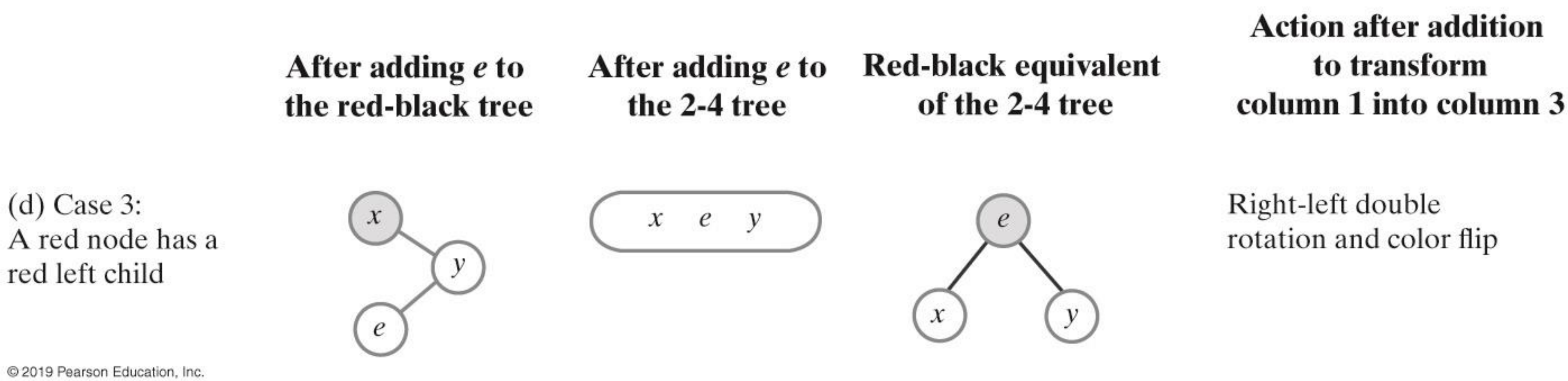


FIGURE 28-31d The possible results of adding a new entry *e* to a two-node red-black tree

Adding Entries to a Red-Black Tree

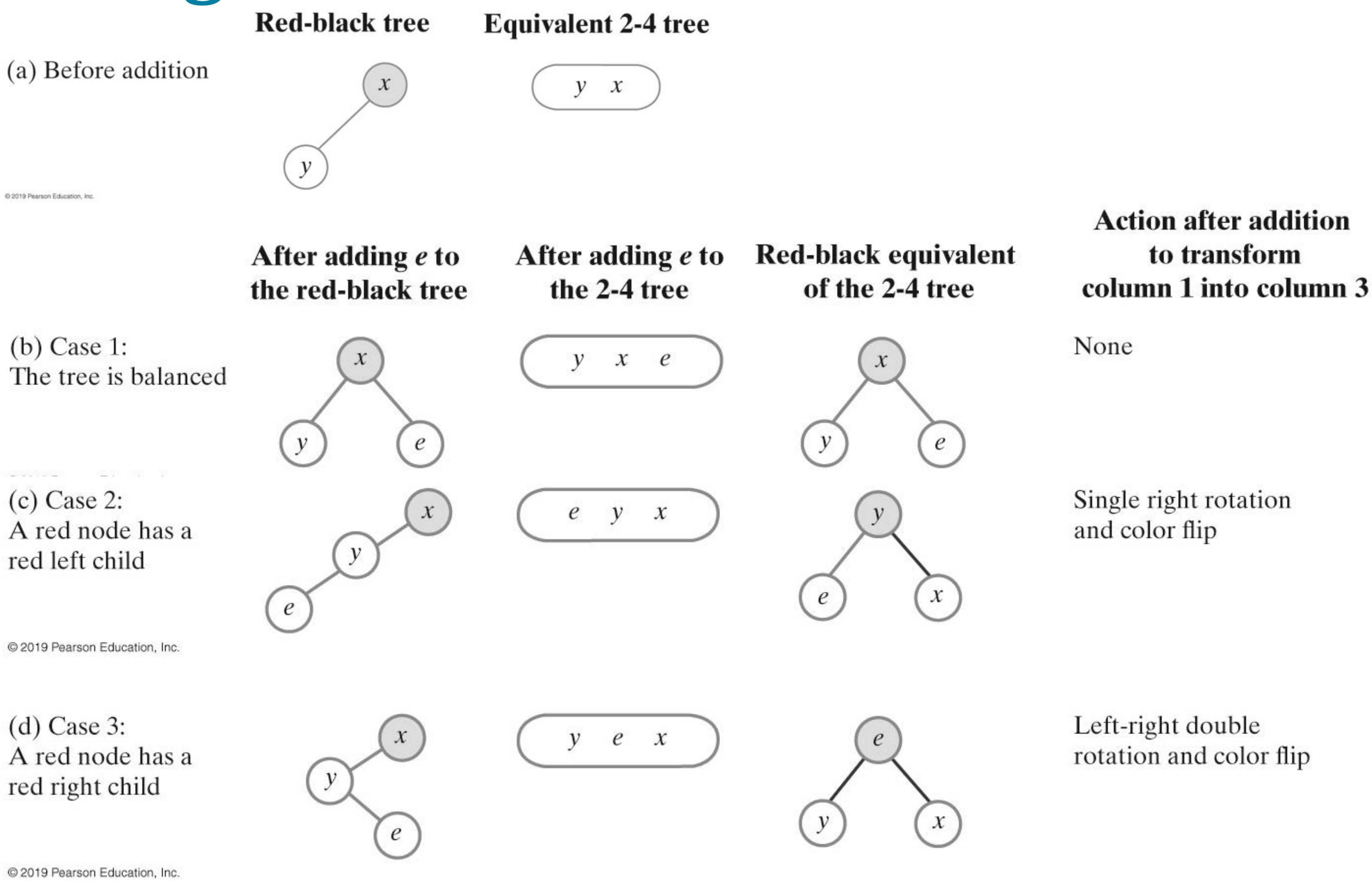
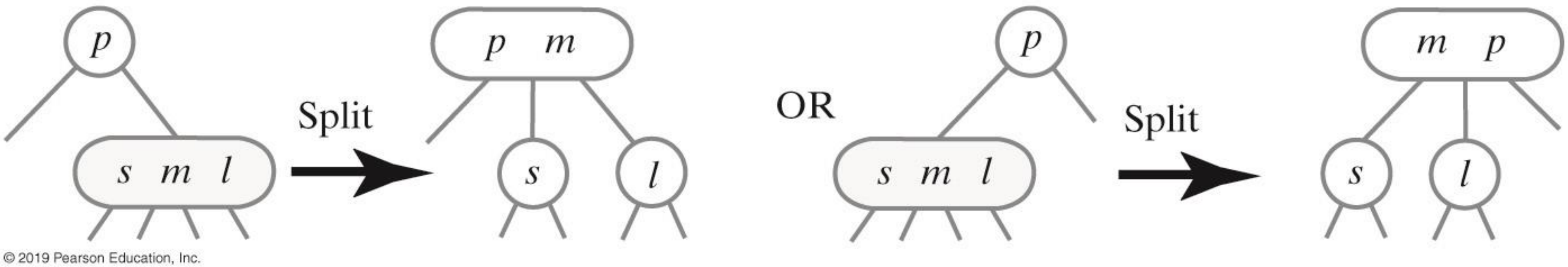


FIGURE 28-32 The possible results of adding a new entry e to a two-node red-black tree: mirror images of Figure 28-31

Splitting a 4-node

(a) In a 2-4 tree



(b) In a red-black tree

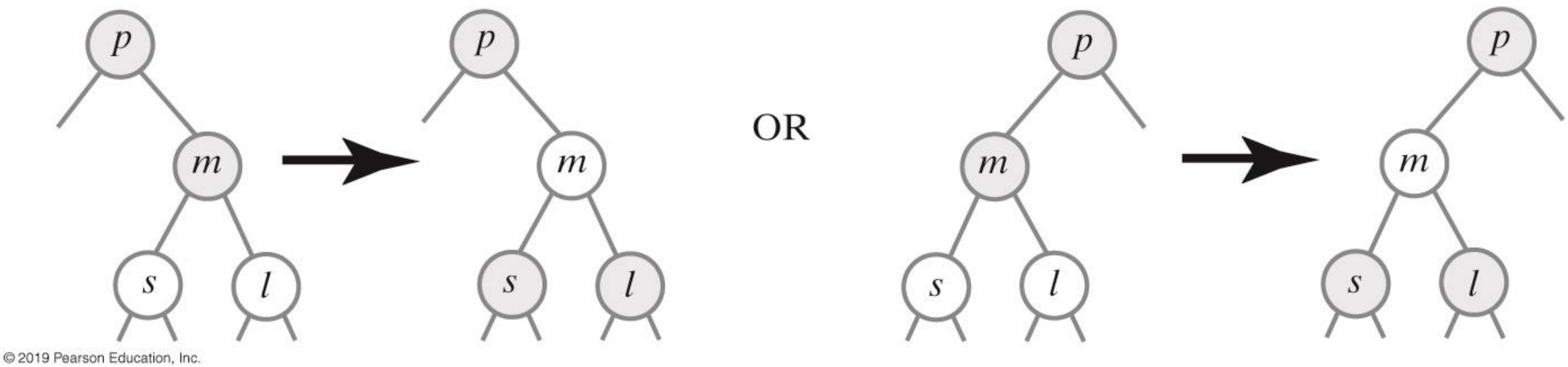


FIGURE 28-33 Splitting a 4-node whose parent is a 2-node

Splitting a 4-node

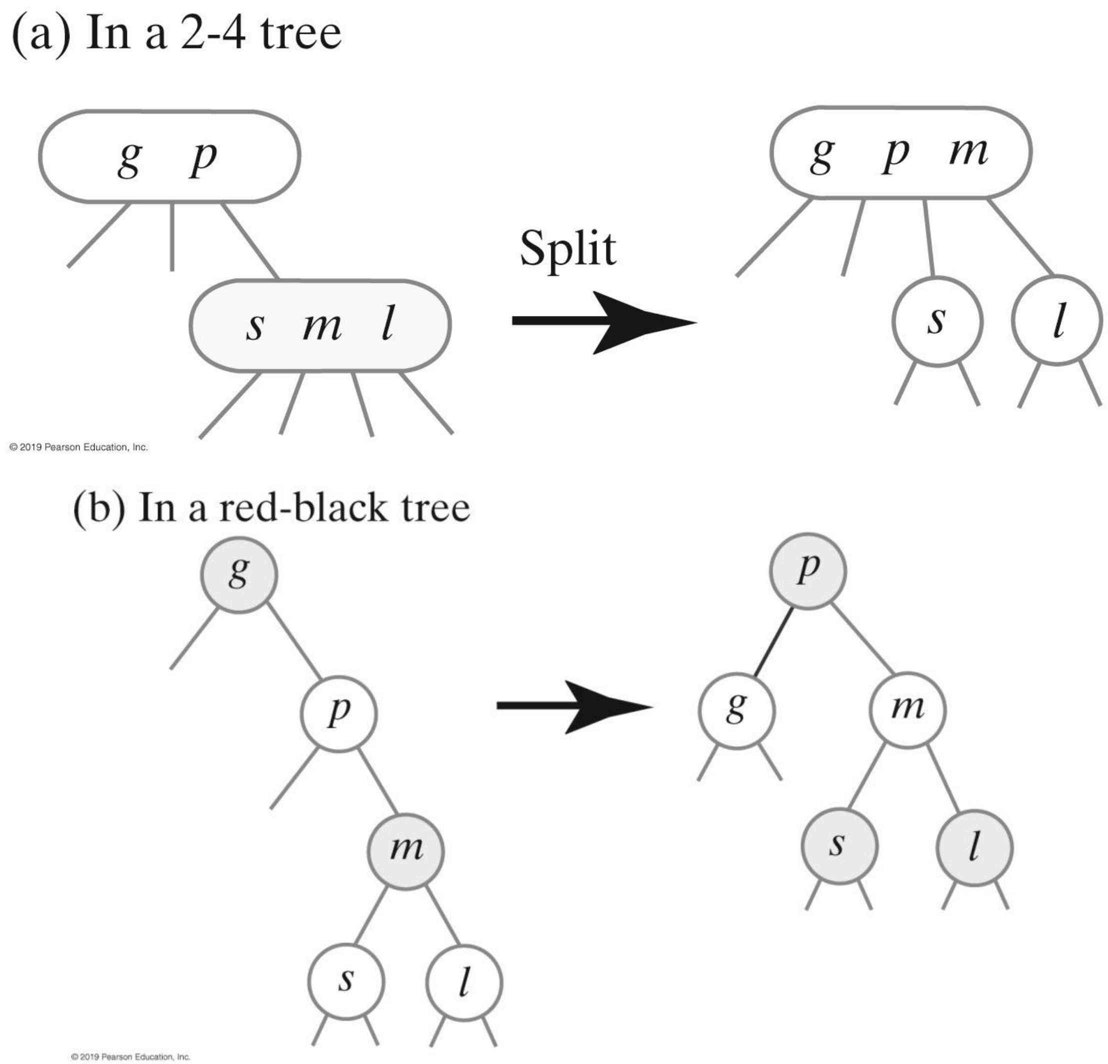


FIGURE 28-34 Splitting a 4-node whose parent is a 3-node

Splitting a 4-node

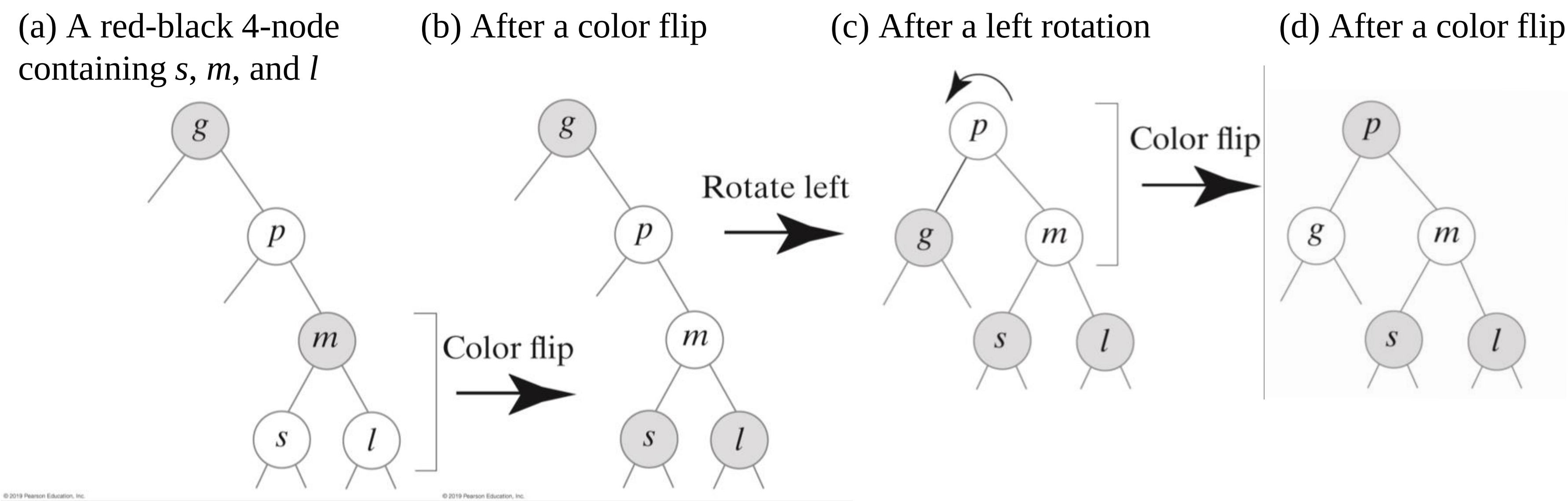


FIGURE 28-35 Splitting a 4-node that has a red parent within a red-black tree: Case 1

Splitting a 4-node

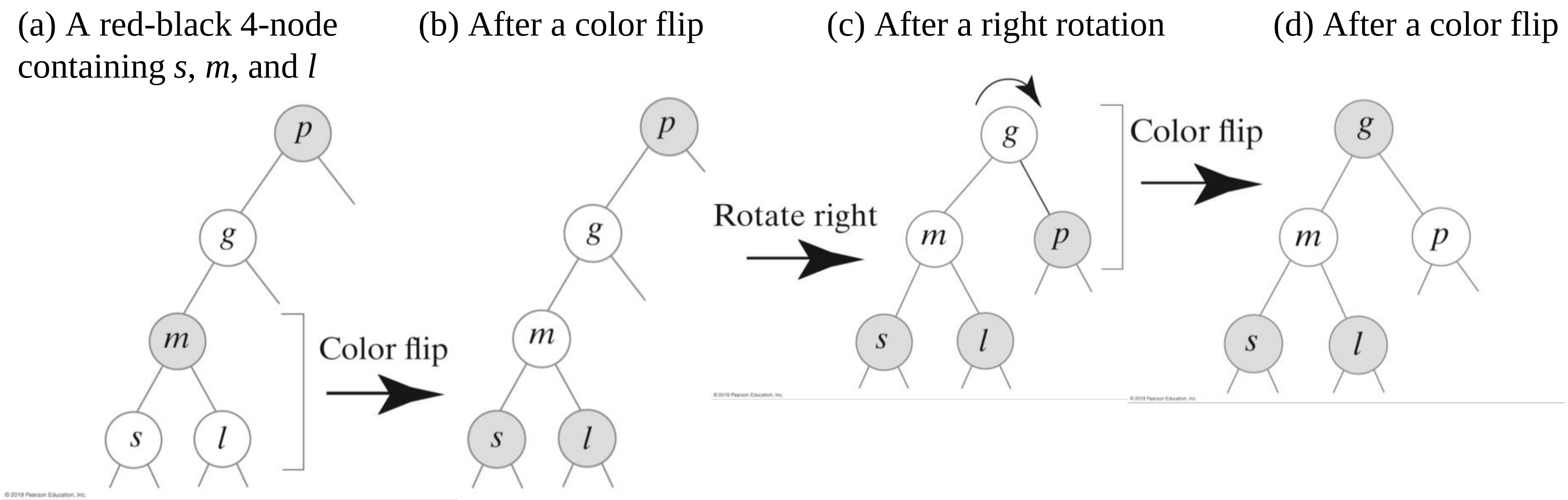


FIGURE 28-36 Splitting a 4-node that has a red parent within a red-black tree: Case 2

Splitting a 4-node

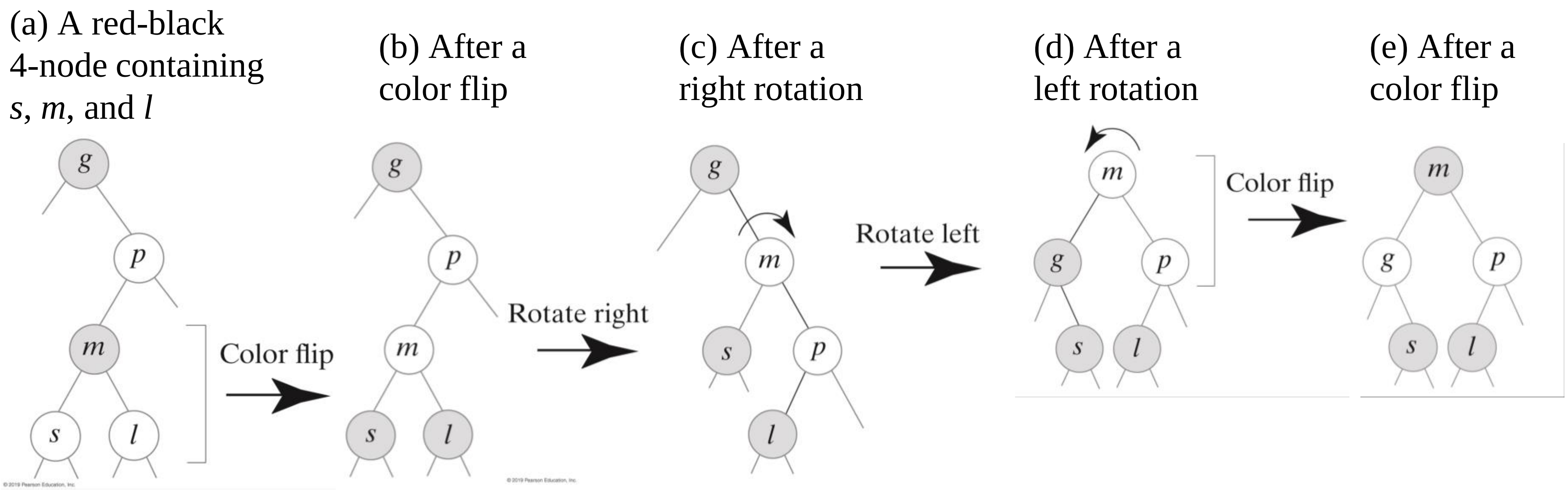


FIGURE 28-37 Splitting a 4-node that has a red parent within a red-black tree: Case 3

Splitting a 4-node

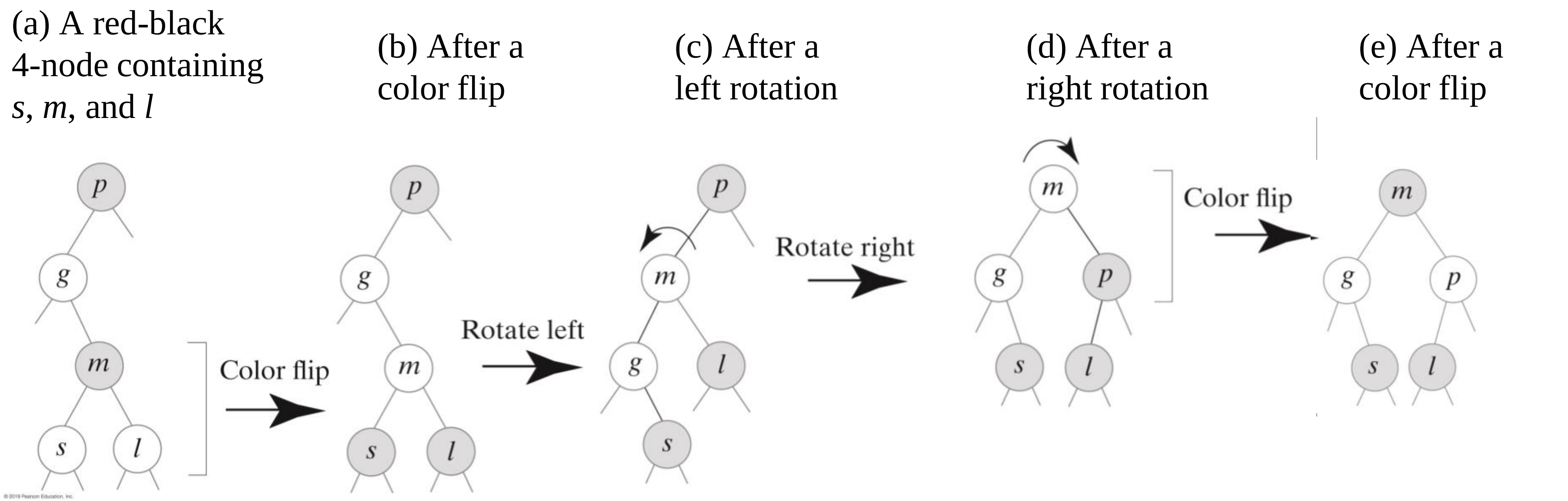


FIGURE 28-38 Splitting a 4-node that has a red parent within a red-black tree: Case 4

End

Chapter 28