Lecture 28: Balanced Search Trees

CS 0445: Data Structures

Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Dec 3, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

AVL Trees

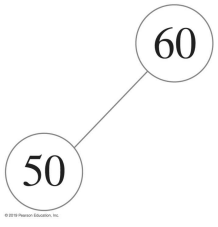
- Possible to form several differently shaped binary search trees
  - From the same collection of data
- AVL tree is a binary search tree that
  - Rearranges its nodes whenever it becomes unbalanced.

Single AVL Tree Rotations

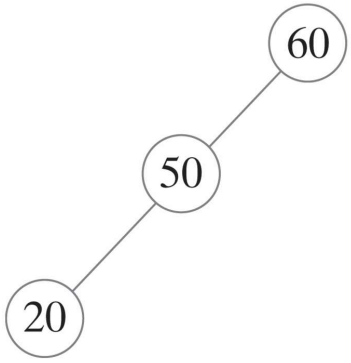
- FIGURE 28-1 Additions to an initially empty AVL tree

(a) After adding 60

(b) After adding 50



(c) Adding 20 makes the tree unbalanced



Unbalanced

(d) A rotation restores balance

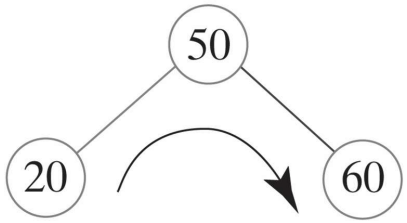
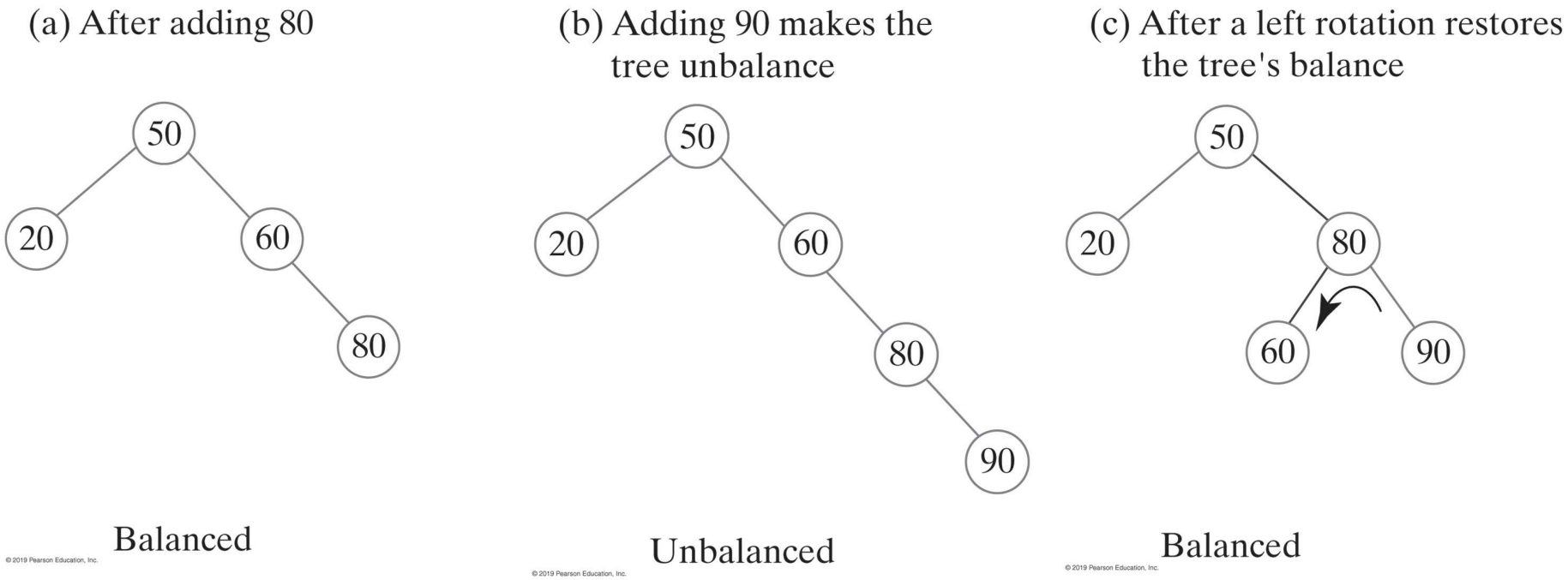


FIGURE 28-2 Additions to the AVL tree in Figure 28-1

- 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

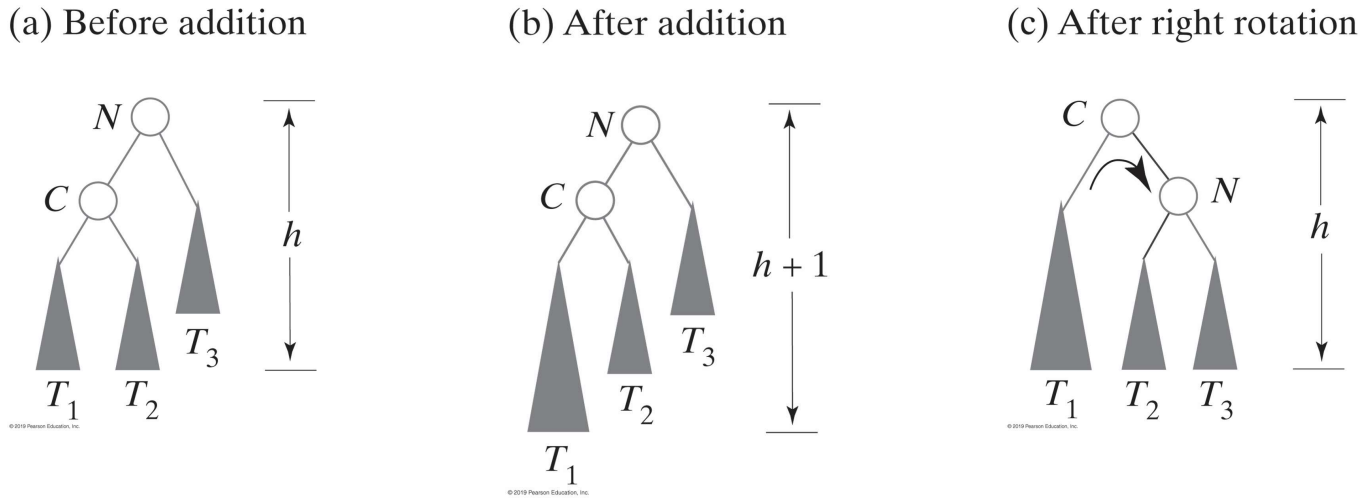
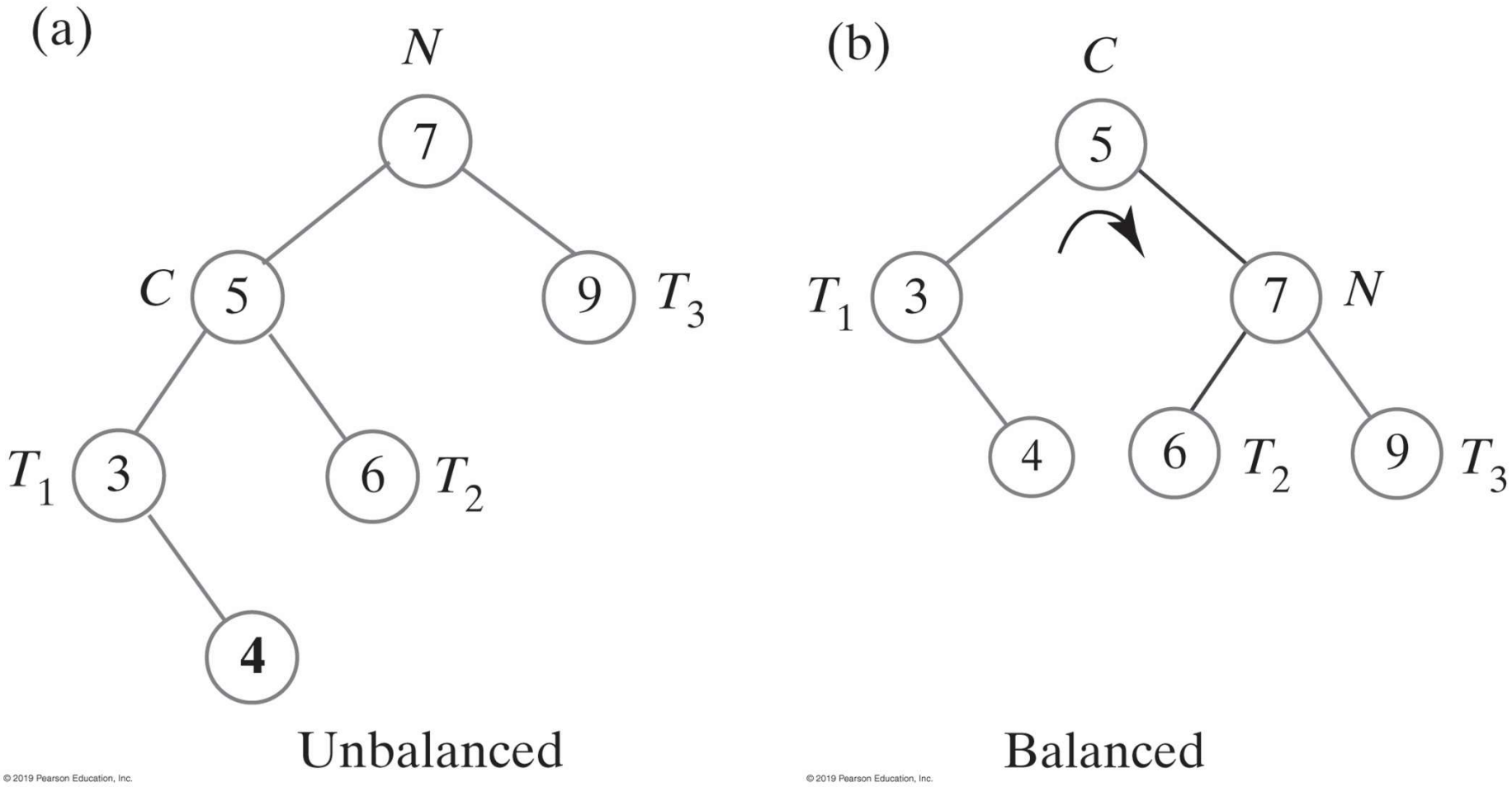


FIGURE 28-3 Before and after an addition to an AVL subtree that requires a right rotation to maintain its balance

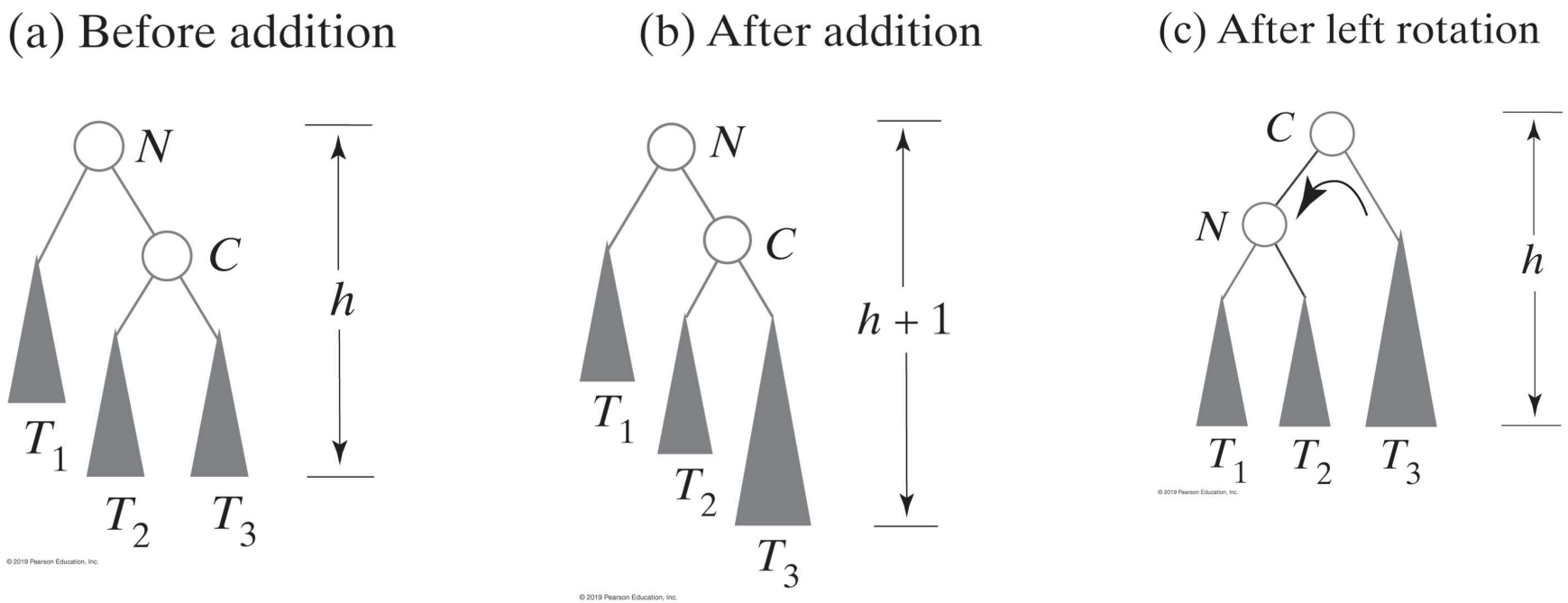
Single Rotation

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



Single Rotation

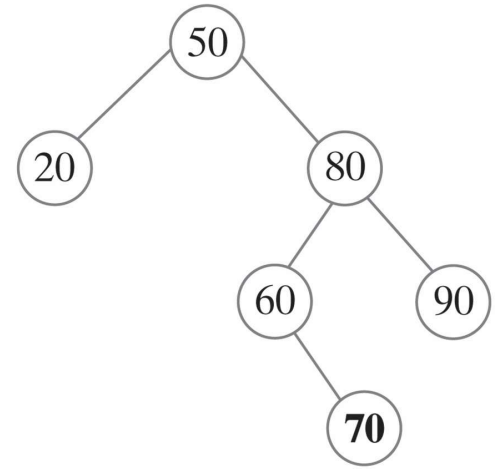
- FIGURE 28-5 Before and after an addition to an AVL subtree that requires a left rotation to maintain its balance



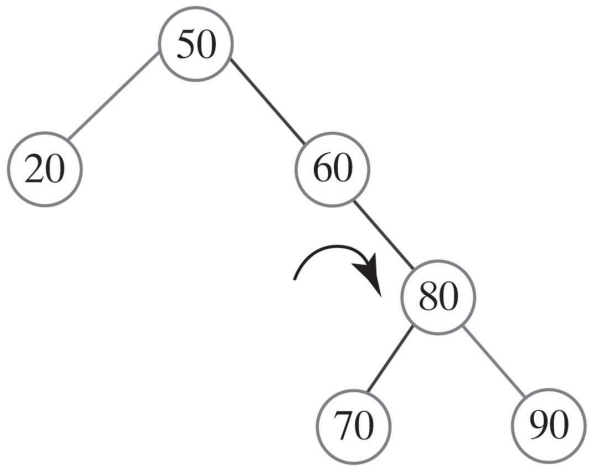
Double Rotation

- 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

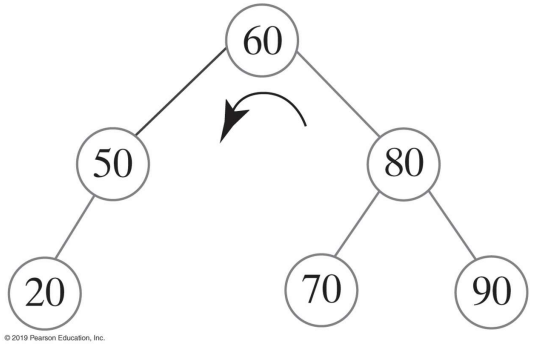
(a) After adding 70



(b) After right rotation



(c) After left rotation



Left-right Double Rotations

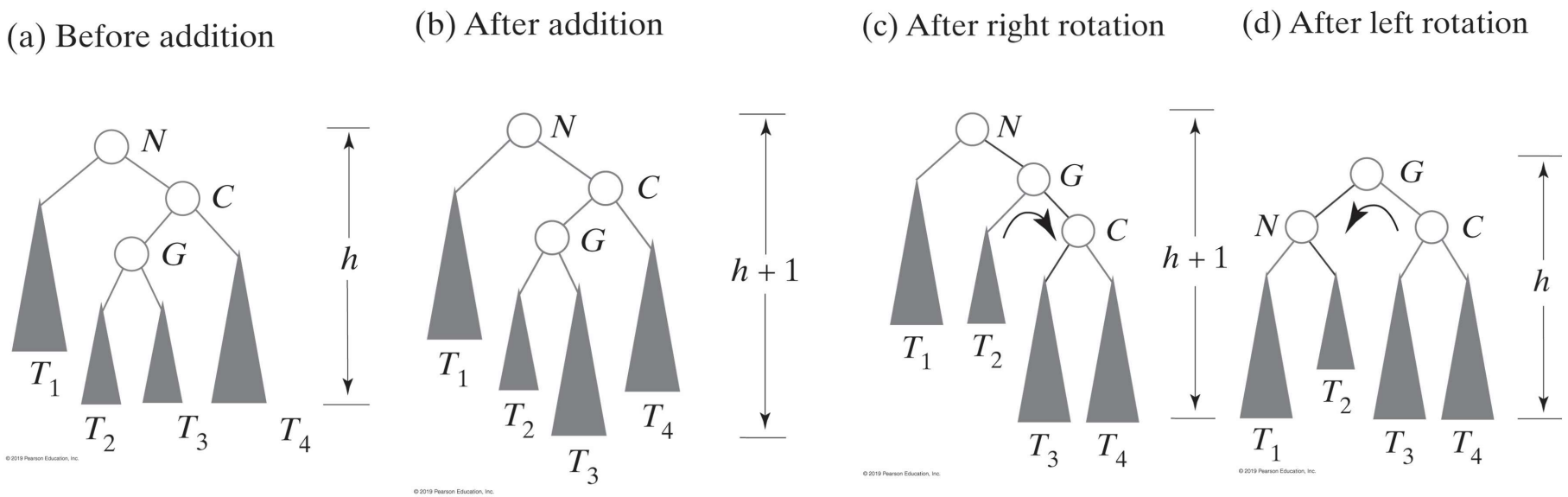
- A double rotation is accomplished by performing two single rotations
- A rotation about node *N*'s grandchild *G* (its child's child)
- A rotation about node *N*'s new child

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

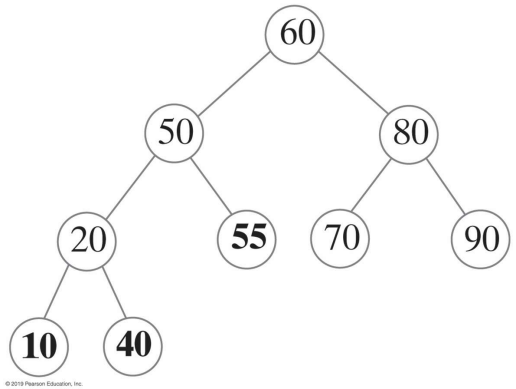
- This algorithm performs the right-left double rotation illustrated in Figure 28-7

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)

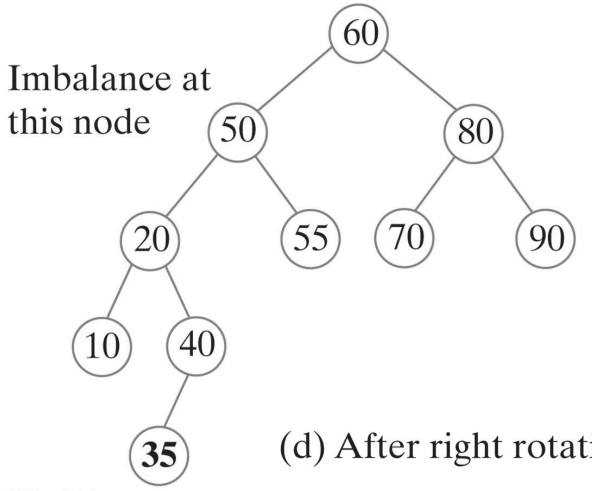
Double Rotations

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

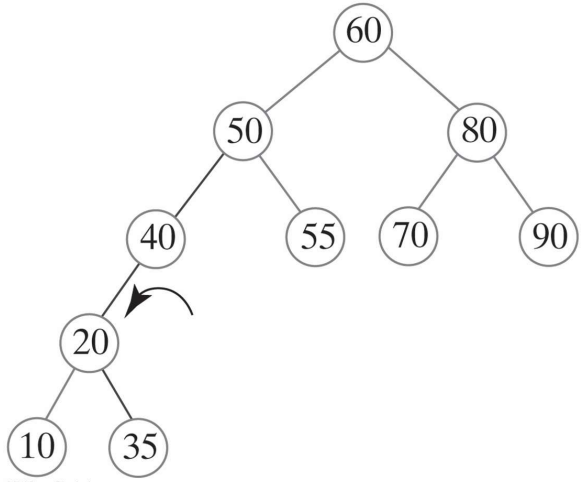
(a) After adding 55, 10, and 40



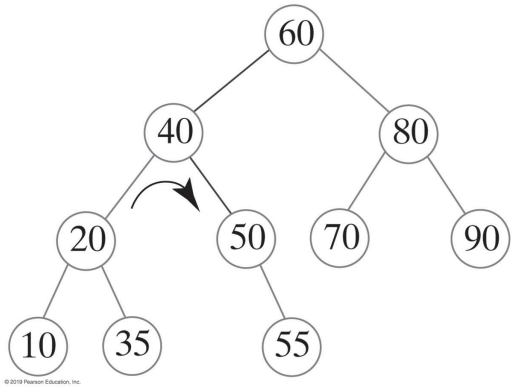
(b) After adding 35



(c) After left rotation about 40

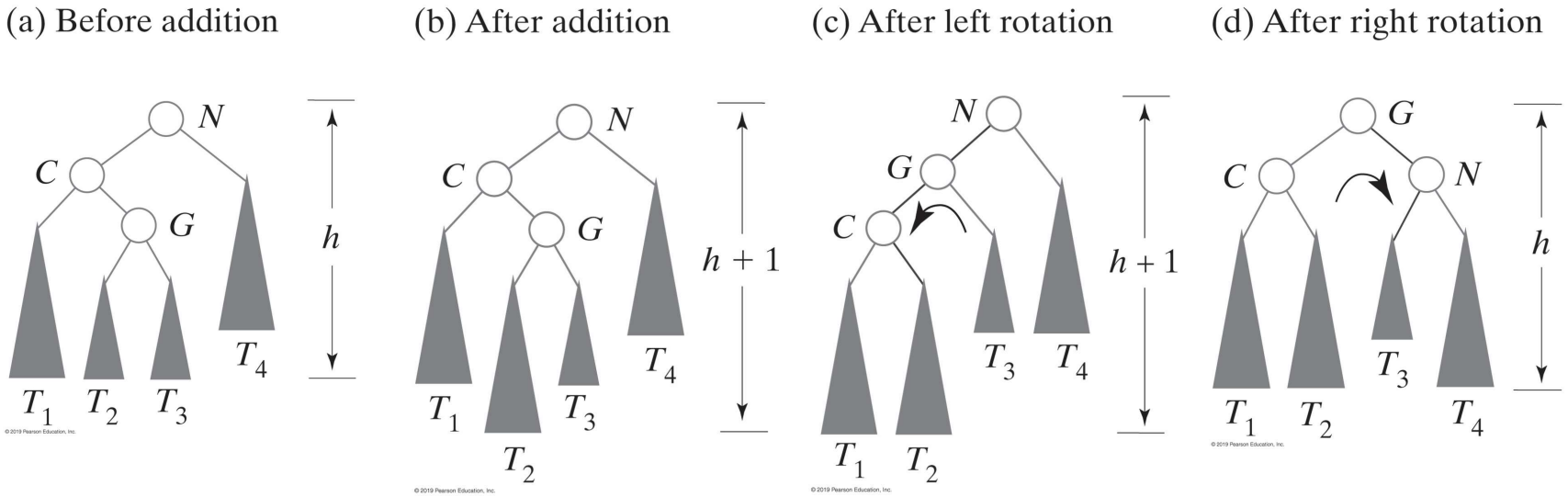


(d) After right rotation about 40



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



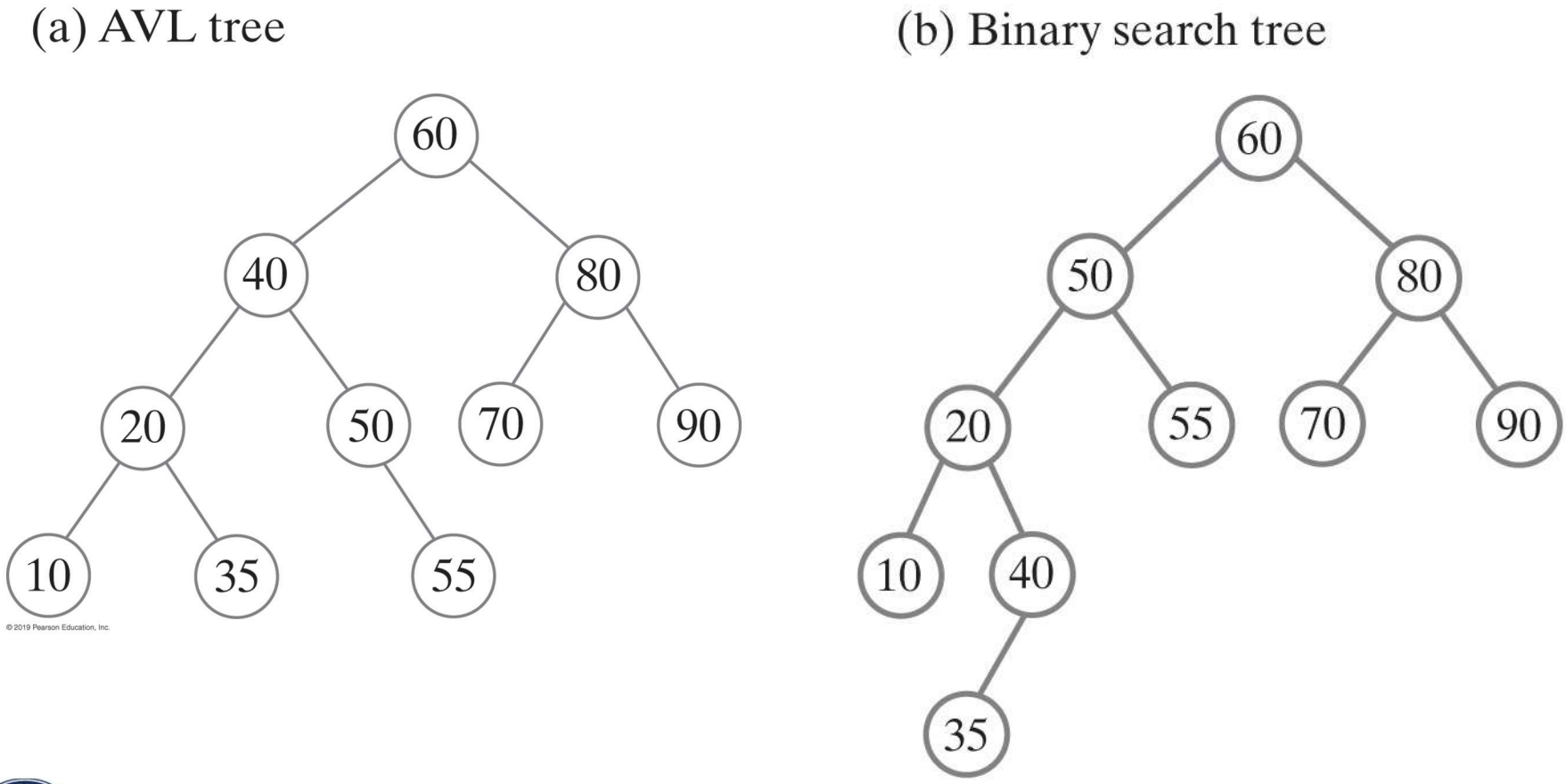
Left-right Double Rotations

- 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



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

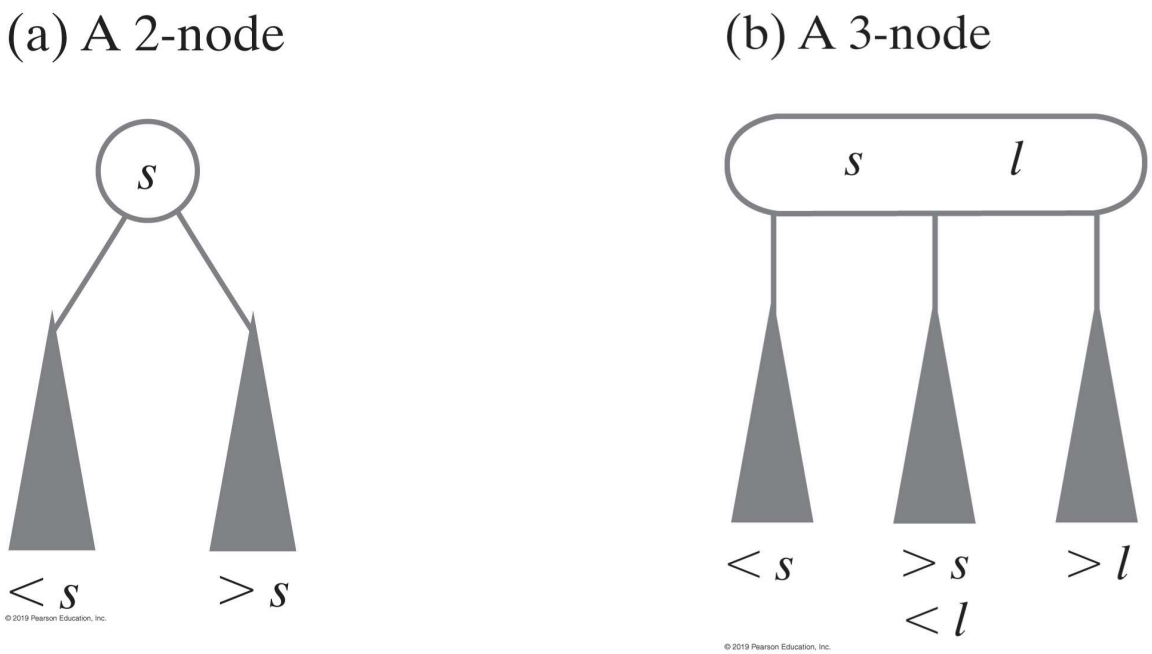
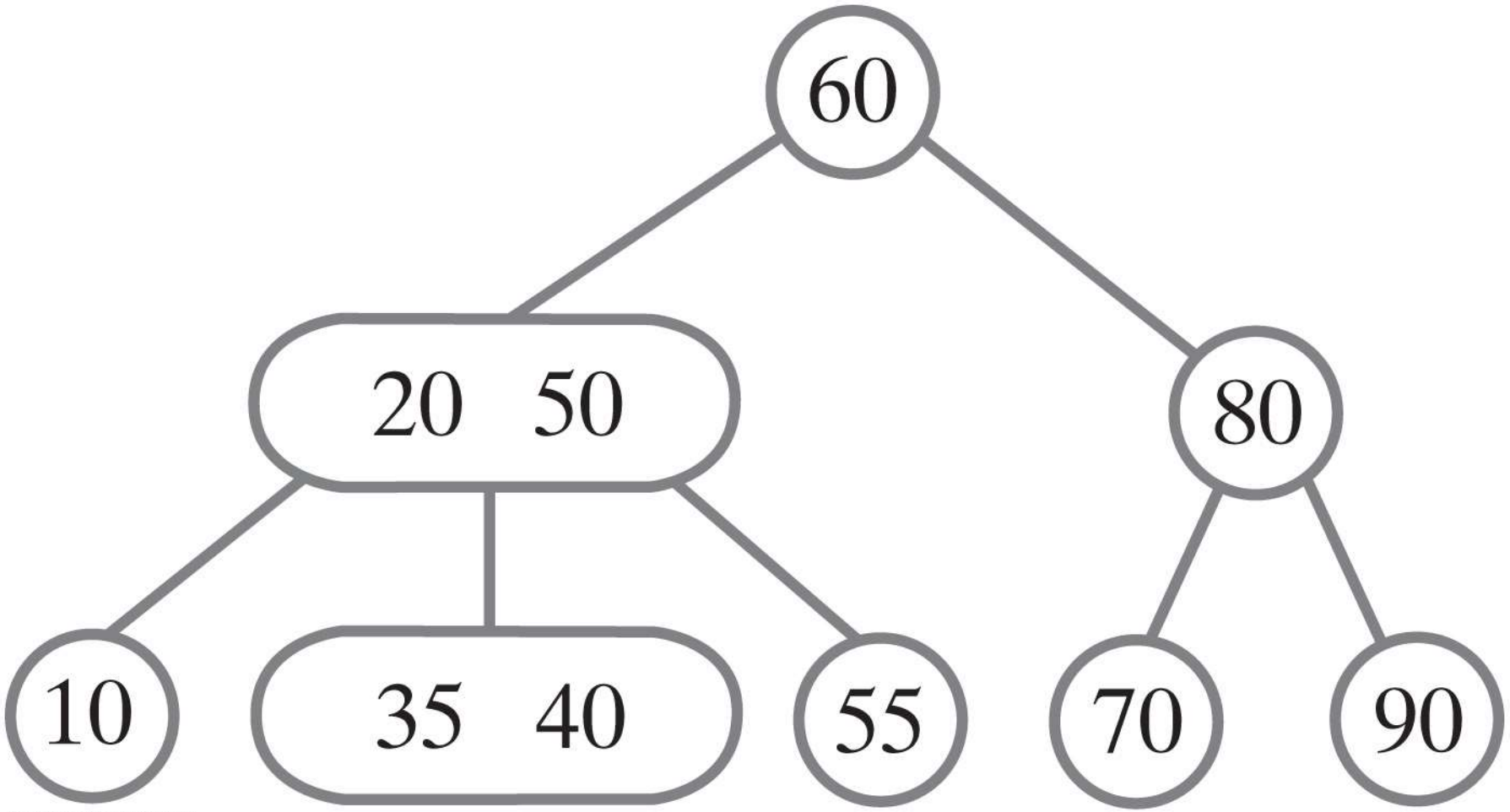


FIGURE 28-11 Nodes in a 2-3 tree

2-3 Trees

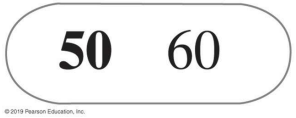


Building 2-3 Trees

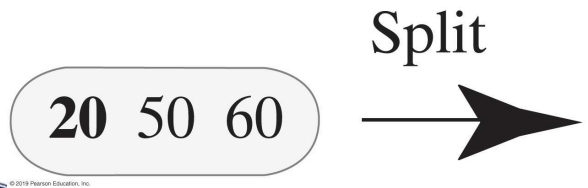
- FIGURE 28-13 An initially empty 2-3 tree after three additions

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

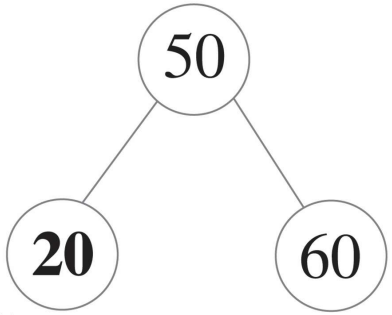
(b) After adding 50, the tree is a 3-node



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

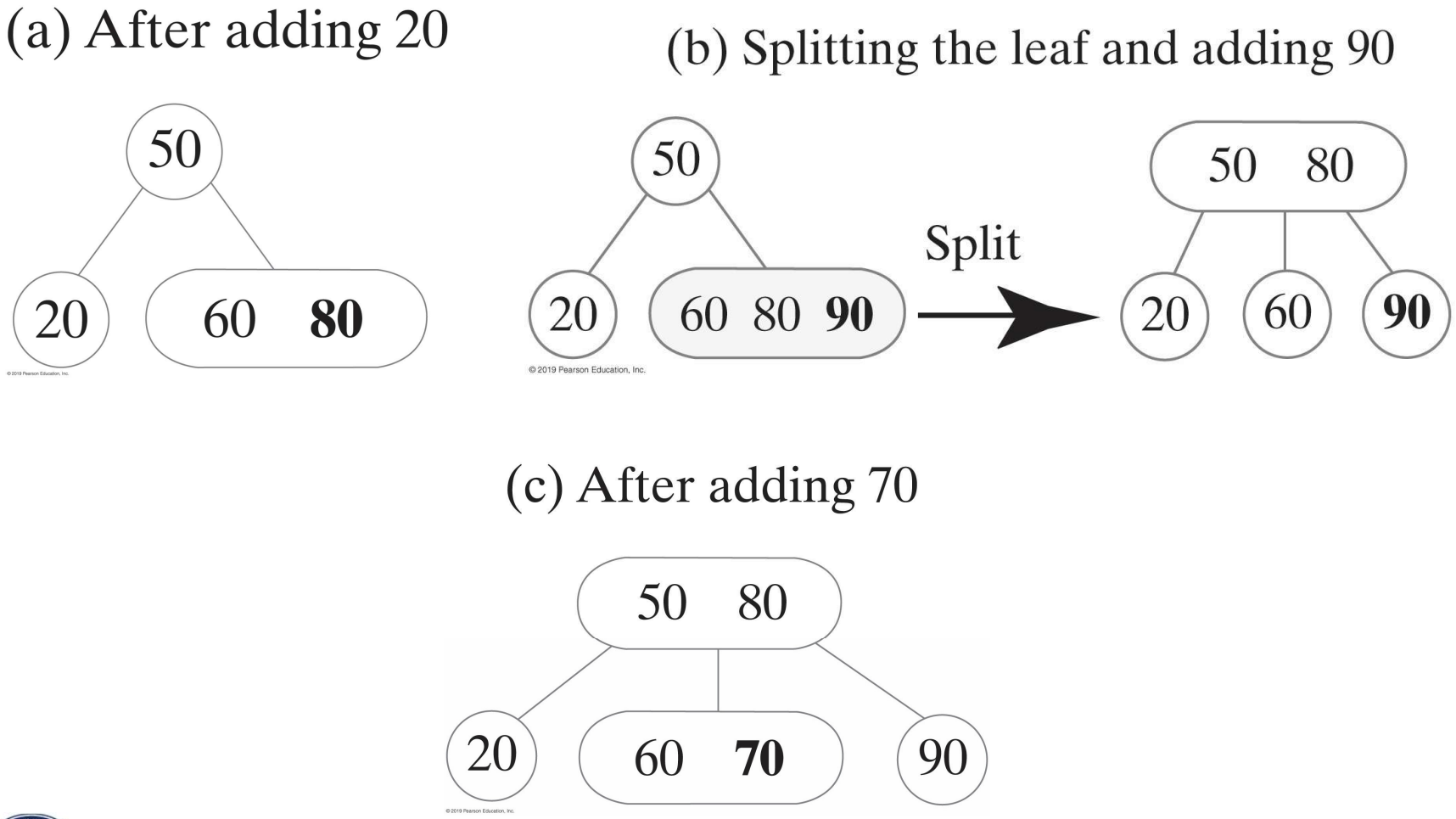


(d) After adding 20



Building 2-3 Trees

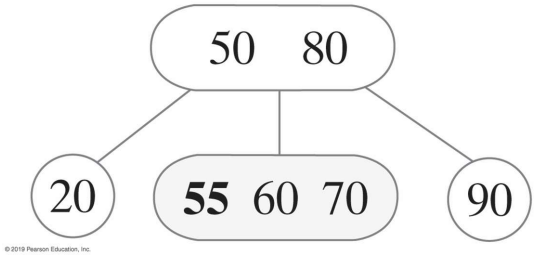
- FIGURE 28-14 The 2-3 tree after three additions



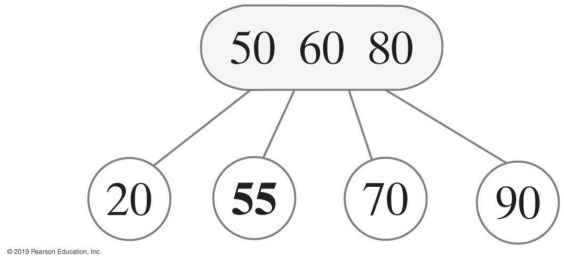
Building 2-3 Trees

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

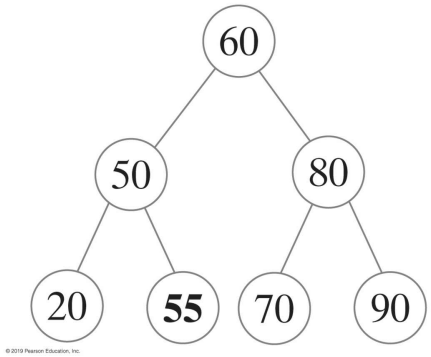
(a) 55 belongs in the middle leaf, but it has no room



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

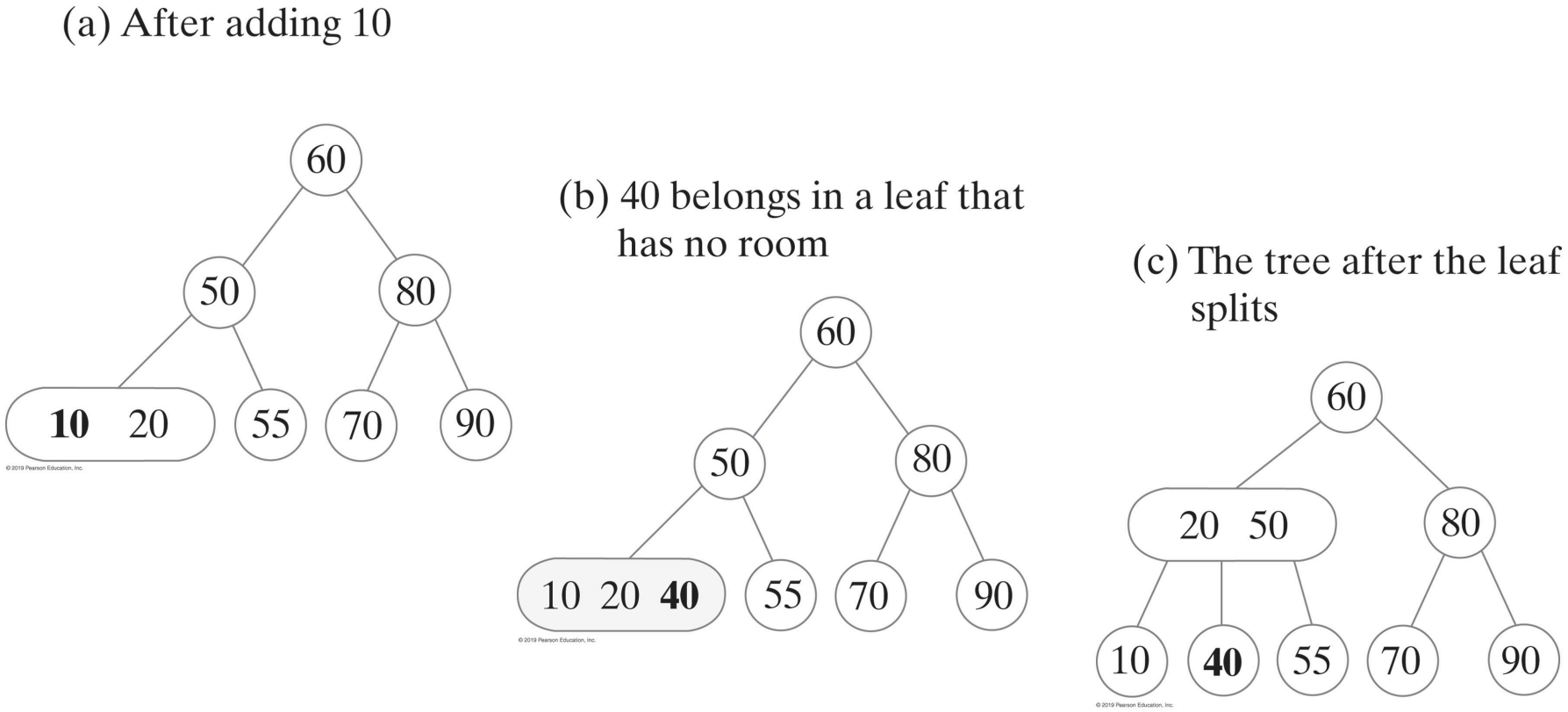


(c) The tree after the root splits



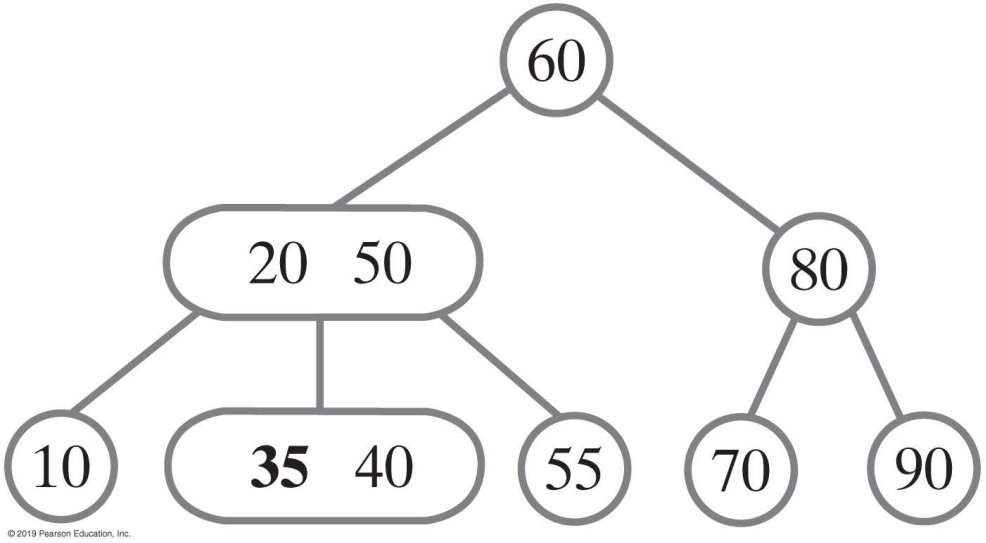
Building 2-3 Trees

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



Building 2-3 Trees

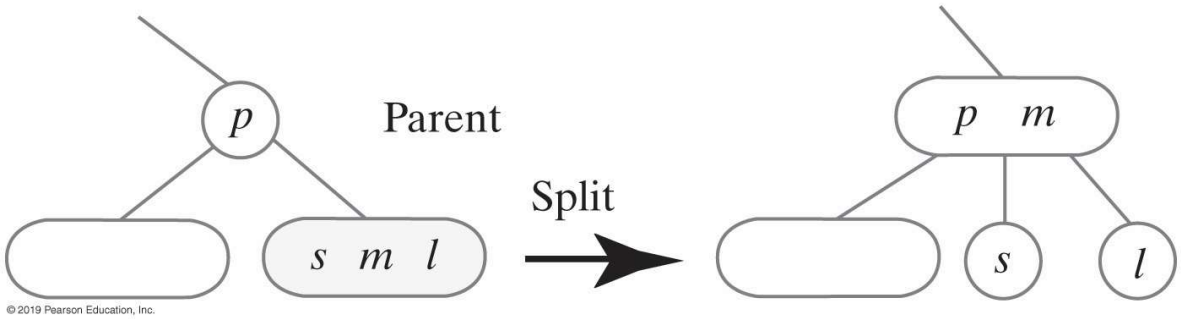
- FIGURE 28-17 The 2-3 tree in Figure 28-16c after adding 35



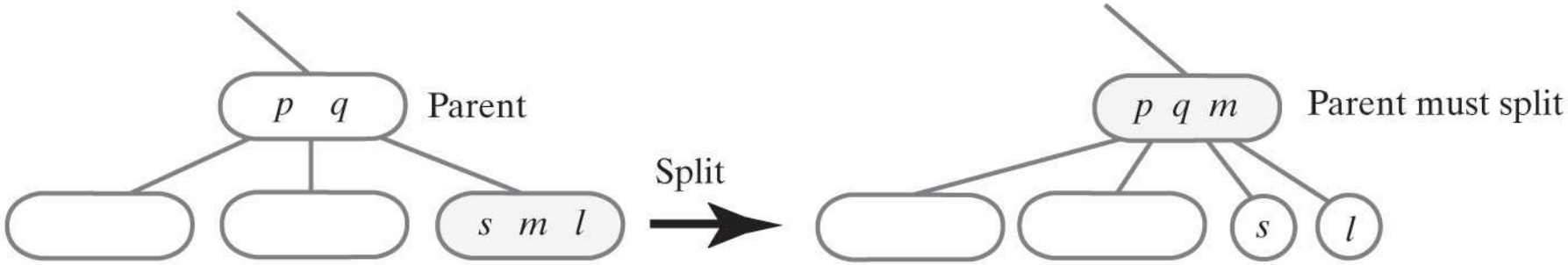
Splitting Nodes During Addition

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

(a) Splitting a leaf when its parent has one entry

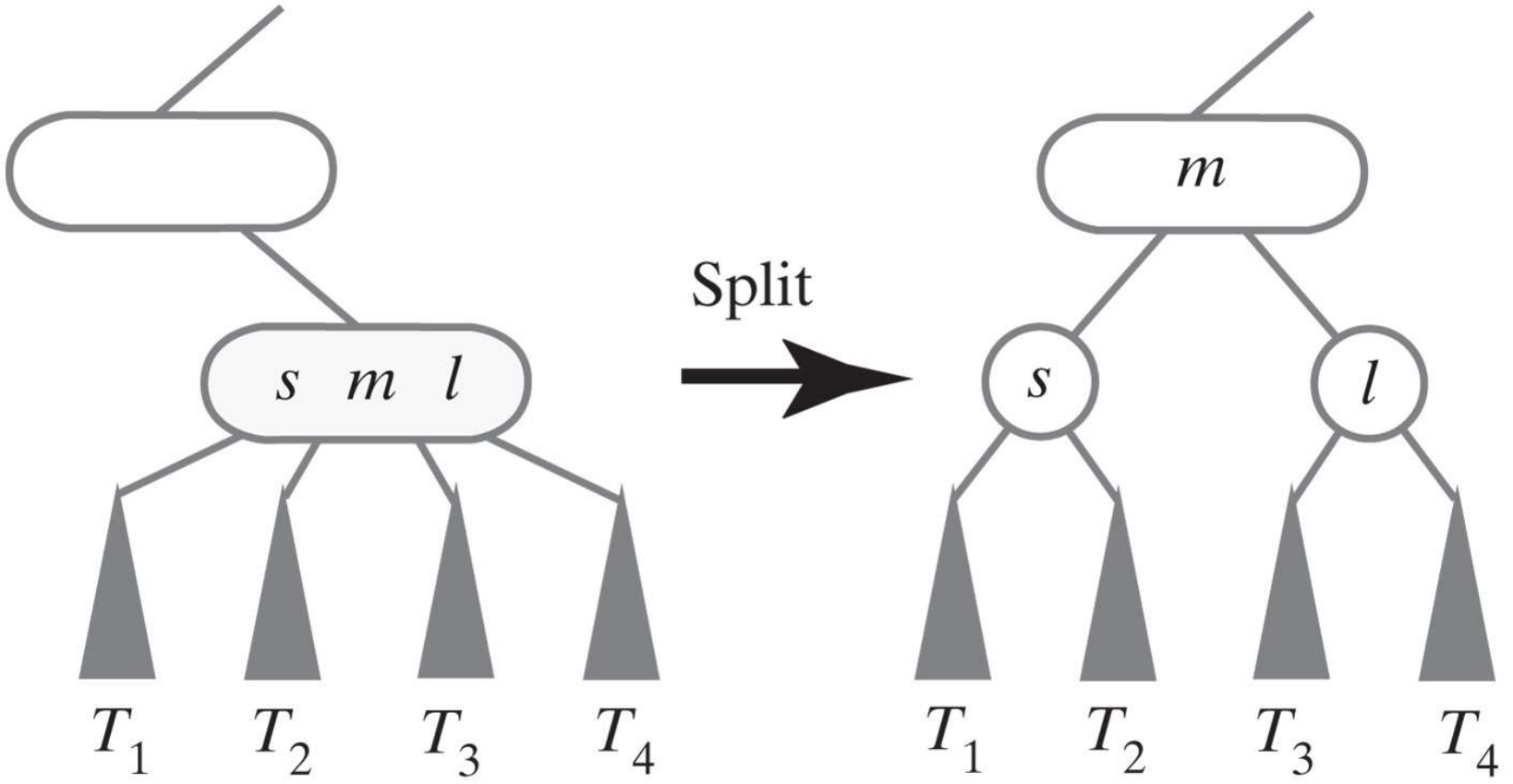


(b) Splitting a leaf when its parent has two entries



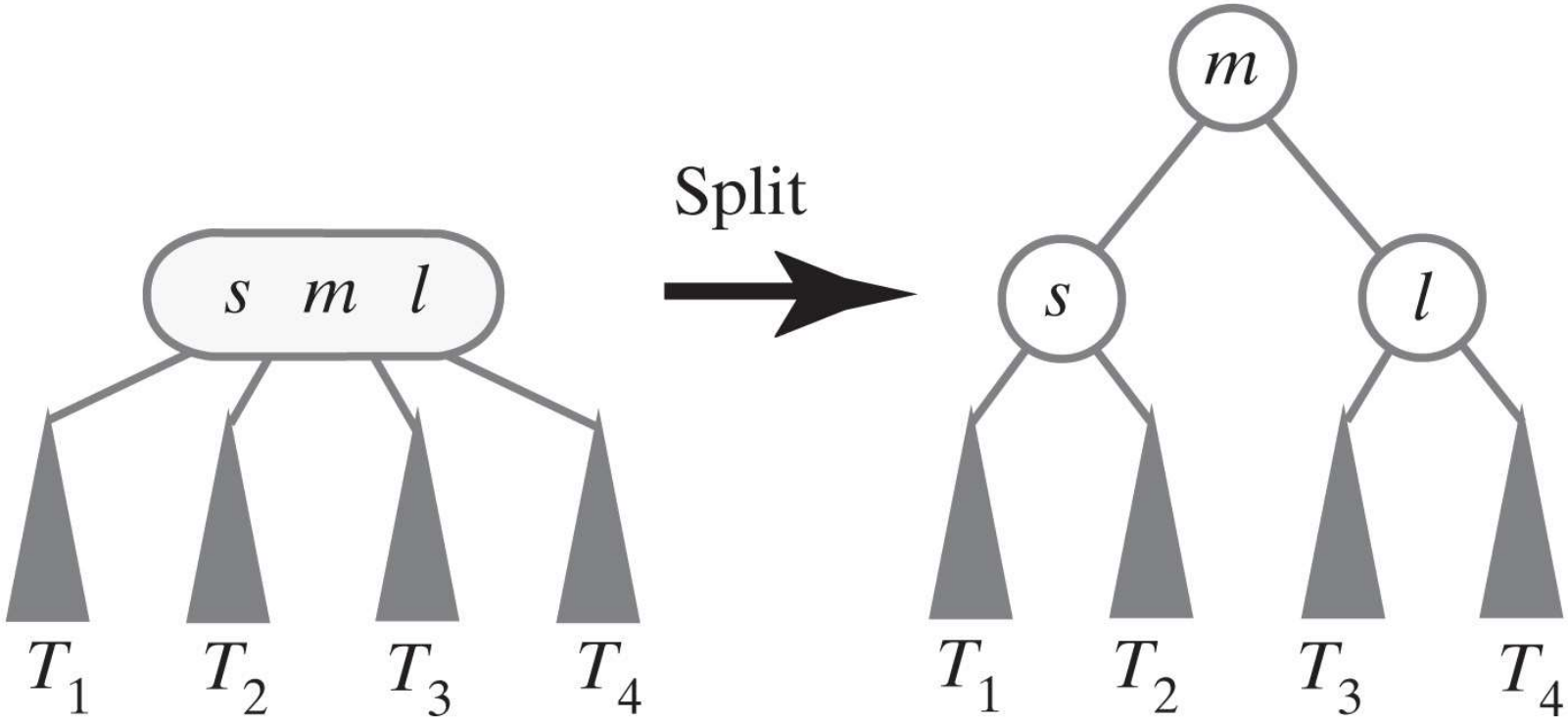
Splitting Nodes During Addition

- 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.

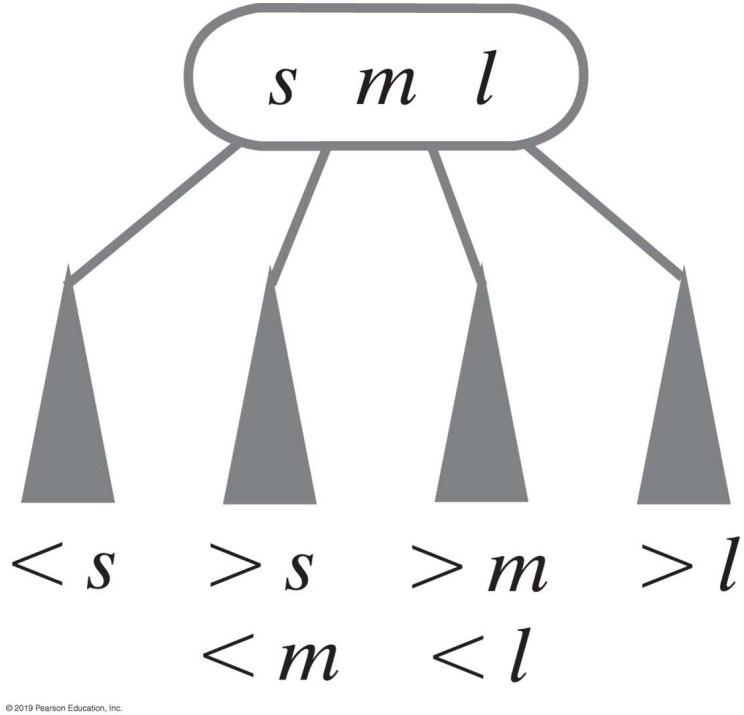


FIGURE 28-21 A 4-node

Adding Entries to a 2-4 Tree

(a) After adding 60 (b) After adding 50 (c) After adding 20

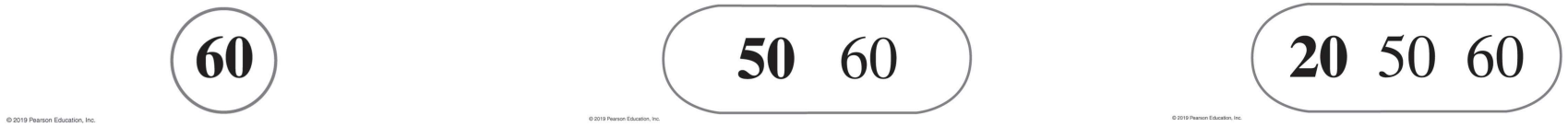


FIGURE 28-22 Adding 60, 50, and 20 to an initially empty 2-4 tree

(a) After splitting the 4-node (b) After adding 80 (c) After adding 90

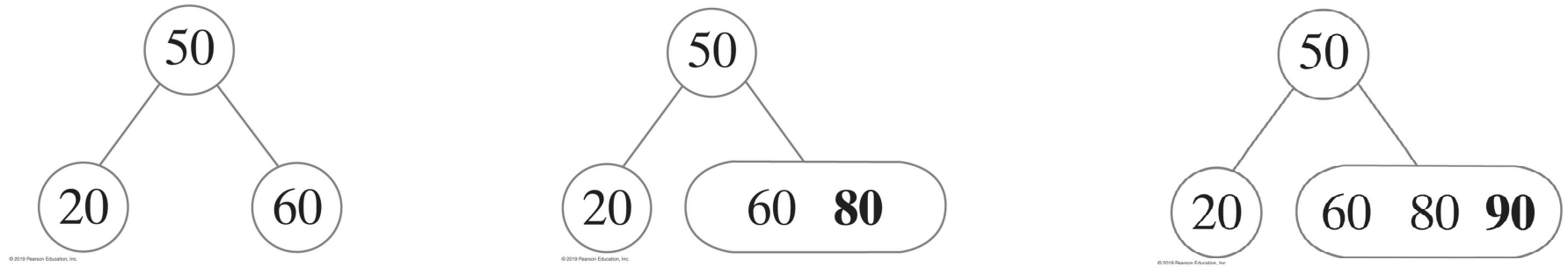


FIGURE 28-23 Adding 80 and 90 to the tree in Figure 28-22c

Adding Entries to a 2-4 Tree

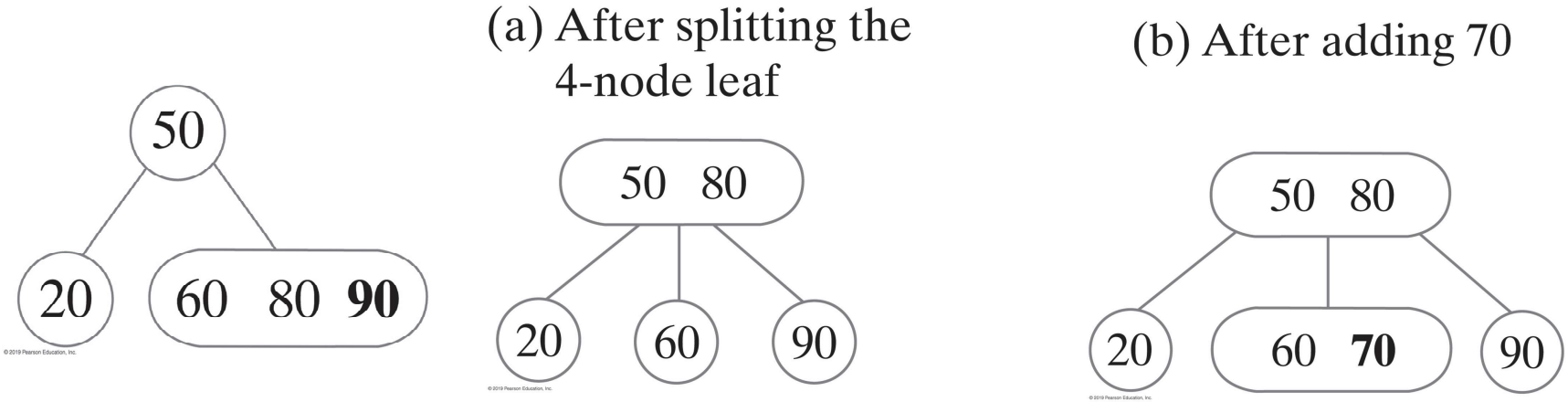


FIGURE 28-24a Adding 70 to the 2-4 tree in Figure 28-23

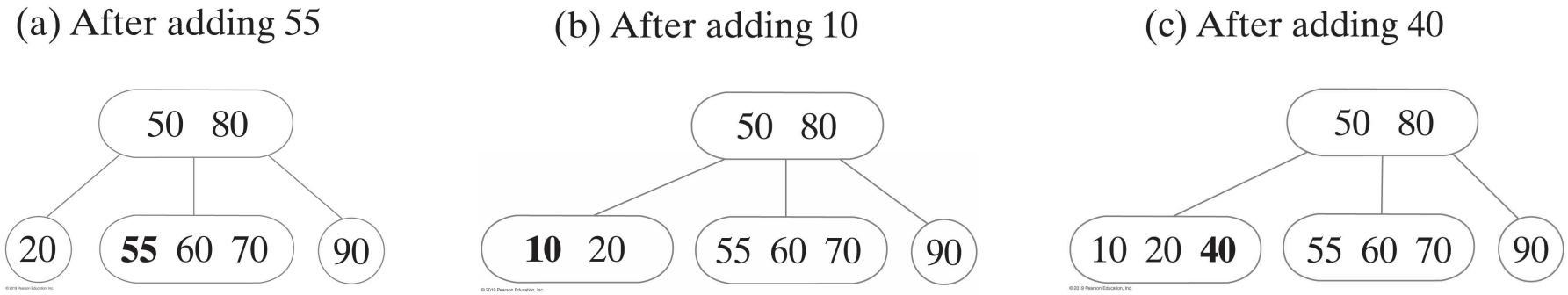
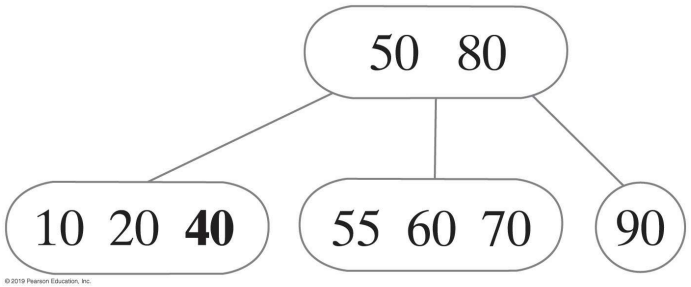


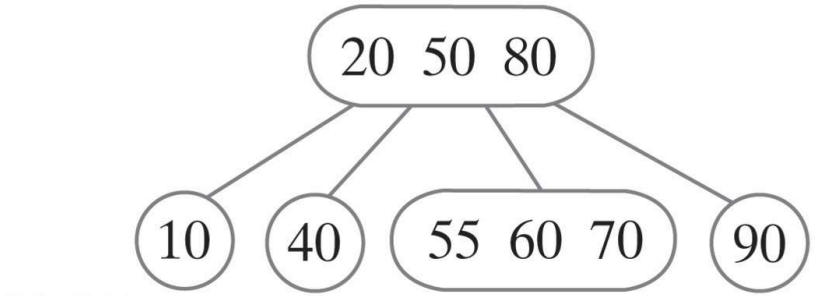
FIGURE 28-25 Adding 55, 10, and 40 to the 2-4 tree in Figure 28-24b

Adding Entries to a 2-4 Tree

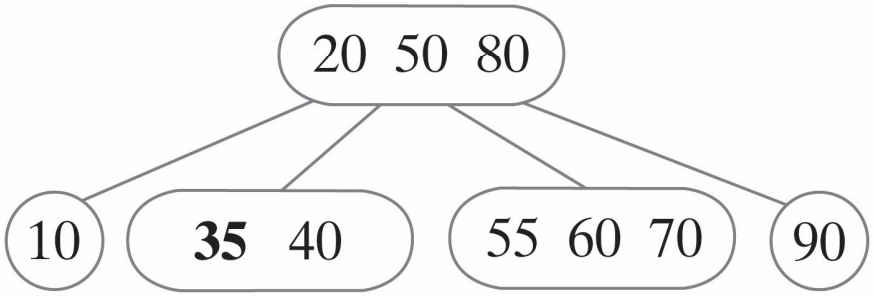
- FIGURE 28-26 Adding 35 to the 2-4 tree in Figure 28-25c



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

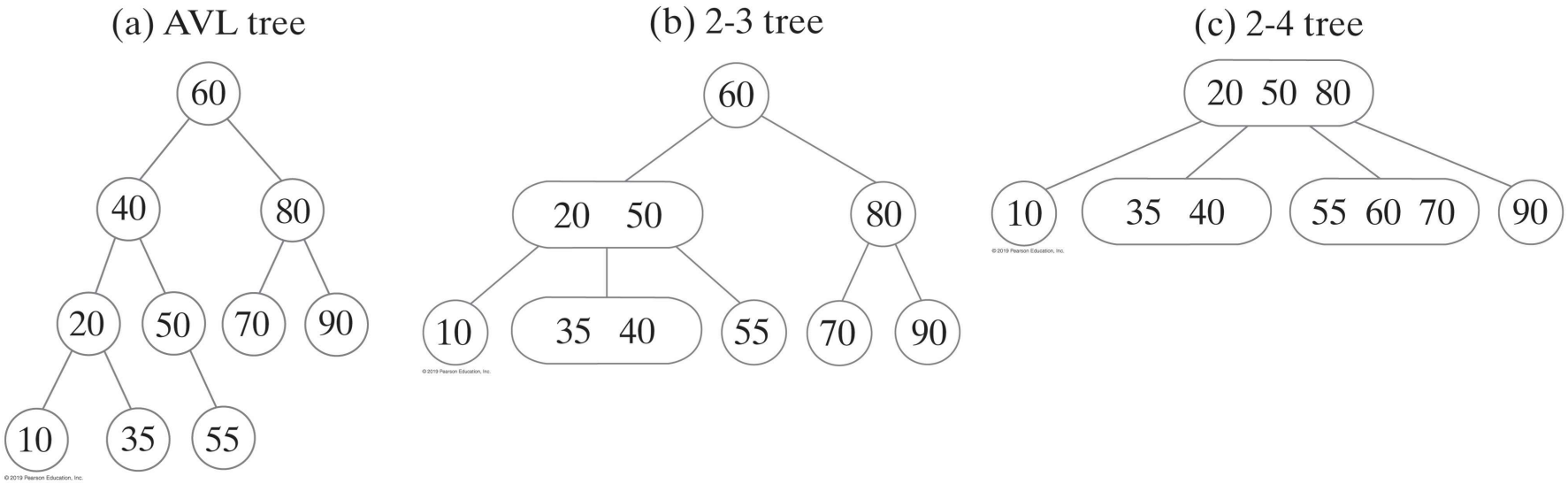


(b) After adding 35



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

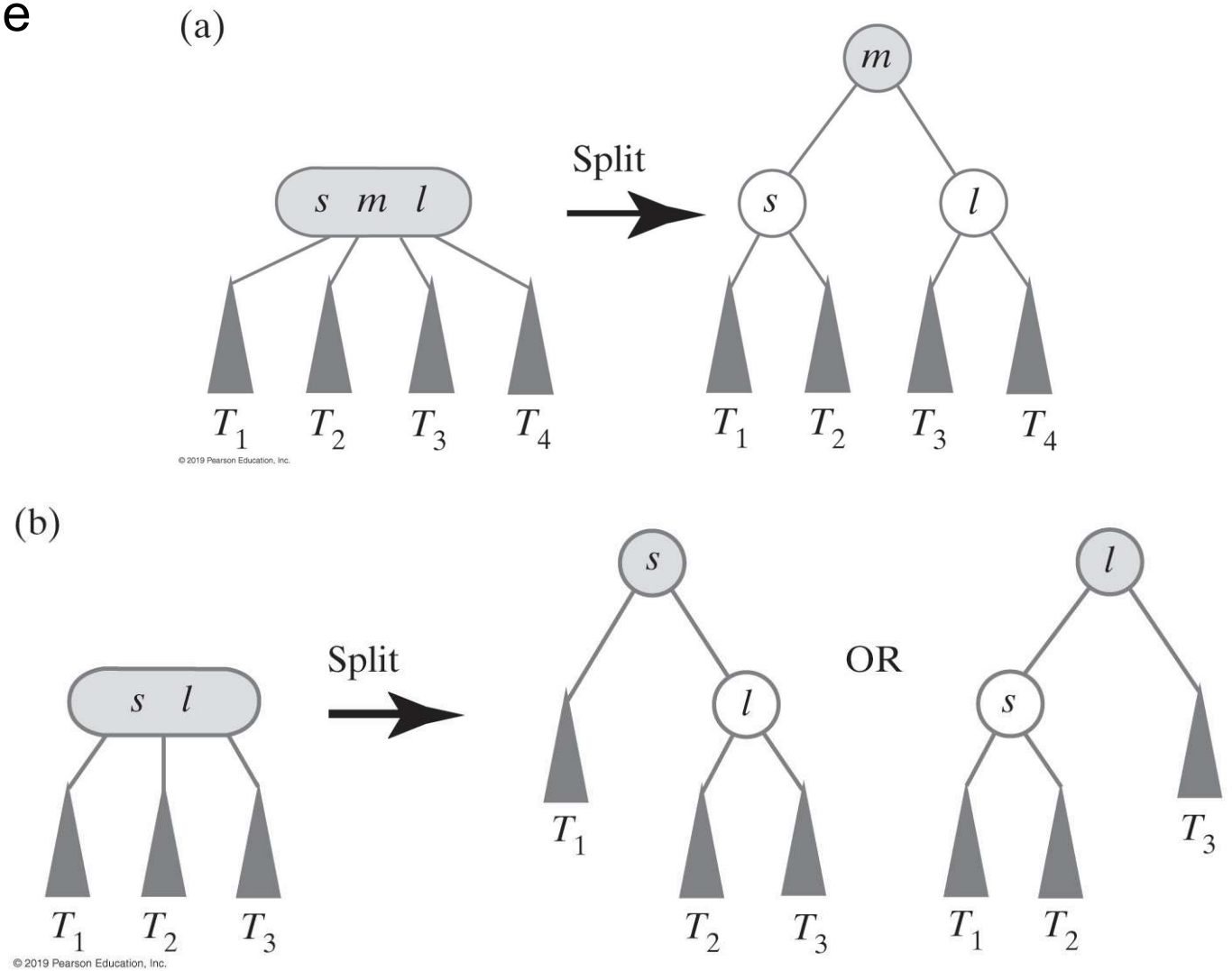


Red-Black Trees

- 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

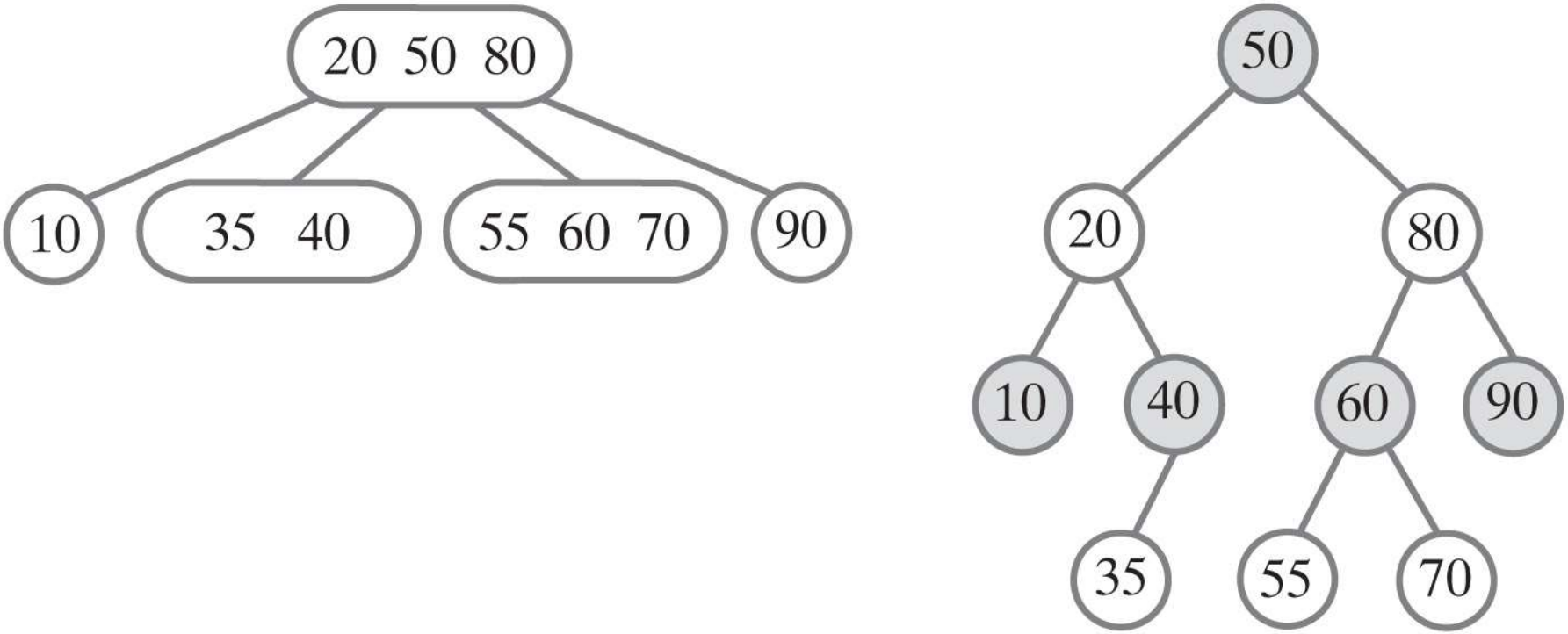
Red-Black Trees

- 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.