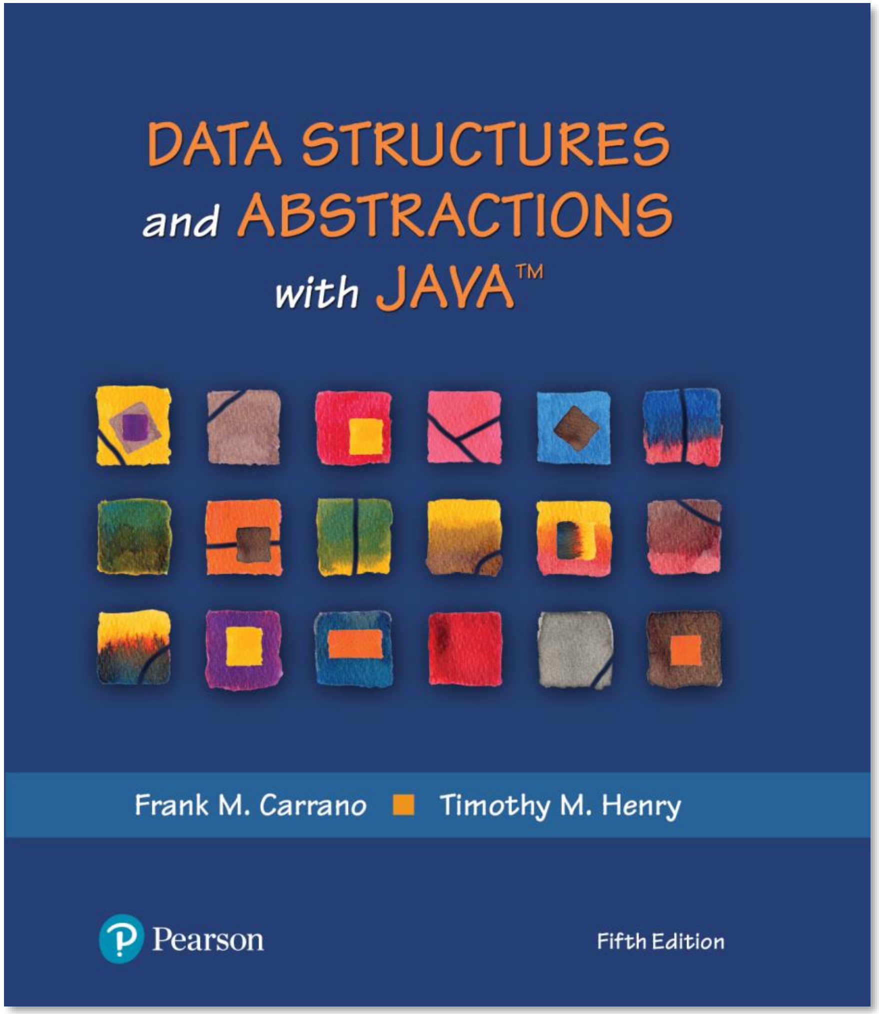
Data Structures and Abstractions with Java™

5th Edition



Chapter 27

A Heap Implementation

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

Pearson

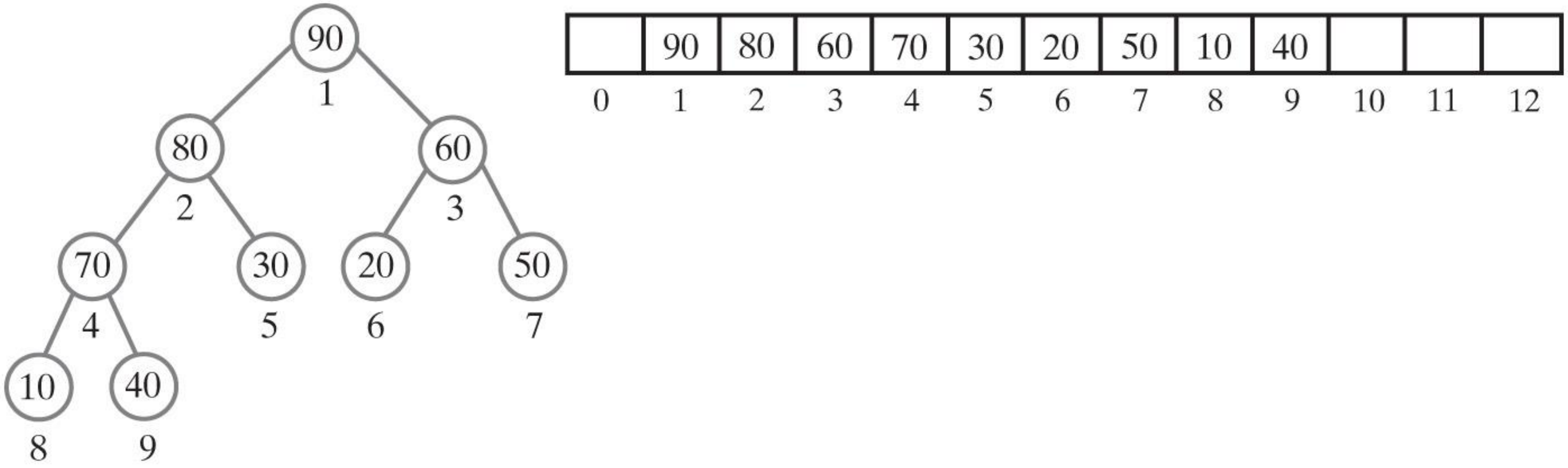
Heap and Maxheap

```
/**  An interface for the ADT maxheap.  */
public interface MaxHeapInterface<T extends Comparable<? super T>>
{  // See Segment 24.33 for a commented version.
   public void add(T newEntry);
   public T removeMax();
   public T getMax();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end MaxHeapInterface
```

Interface for the ADT Maxheap

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



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

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

An Array to Represent a Heap (Part 1)

```
/** A class that implements the ADT maxheap by using an array. */
public final class MaxHeap<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 boolean integrityOK = false;
      private static final int DEFAULT_CAPACITY = 25;
      private static final int MAX_CAPACITY = 10000;

 public MaxHeap(int initialCapacity)
  {
    // Is initialCapacity too small?
    if (initialCapacity < DEFAULT_CAPACITY)
      initialCapacity = DEFAULT_CAPACITY;
    else // Is initialCapacity too big?
      checkCapacity(initialCapacity);

    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] tempHeap = (T[])new Comparable[initialCapacity + 1];
    heap = tempHeap;
    lastIndex = 0;
    integrityOK = true;
  } // end constructor
```

LISTING 27-1 The class MaxHeap, partially completed

An Array to Represent a Heap (Part 2)

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

public T getMax()
{
     checkIntegrity();
   T root = null;
   if (!isEmpty())
      root = heap[1];
   return root;
} // end getMax

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

public int getSize()
{
   return lastIndex;
} // end getSize
```

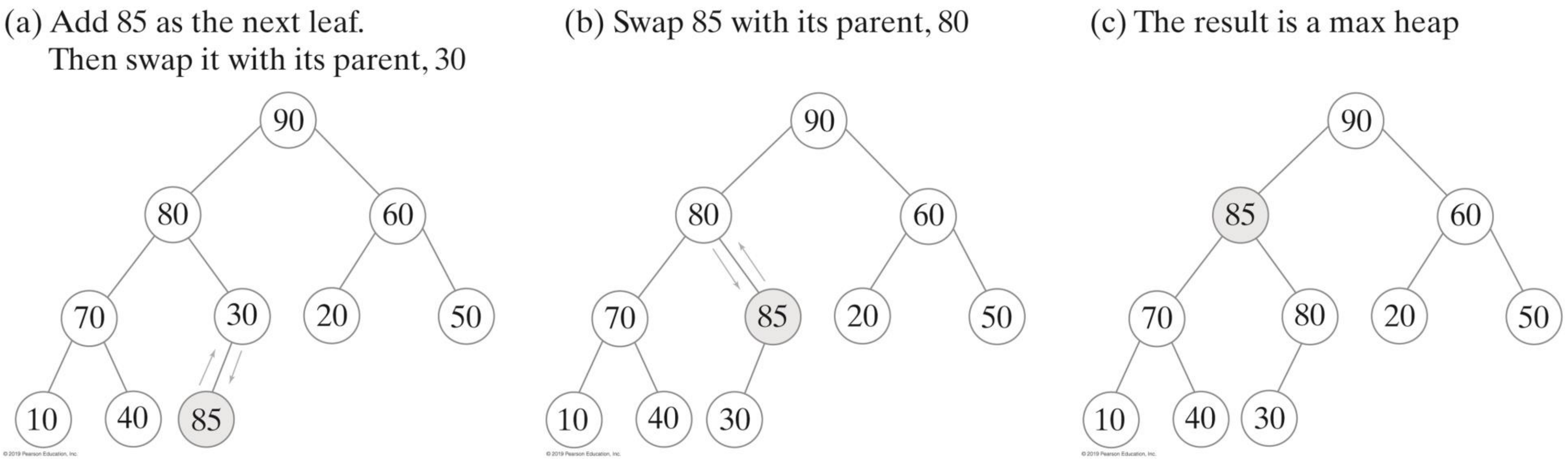
LISTING 27-1 The class MaxHeap, partially completed

An Array to Represent a Heap (Part 3)

```
  public void add(T newEntry)
  {
  // Will address later — See Segment 27.8.
  } // end add

  public T removeMax()
  {
  // Will address later — See Segment 27.12.
  } // end removeMax

  public void clear()
  {
     checkIntegrity();
    while (lastIndex > -1)
    {
     heap[lastIndex] = null;
     lastIndex--;
    } // end while
    lastIndex = 0;
  } // end clear

// Private methods
// . . .
} // end MaxHeap
```

LISTING 27-1 The class MaxHeap, partially completed

Adding an Entry


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

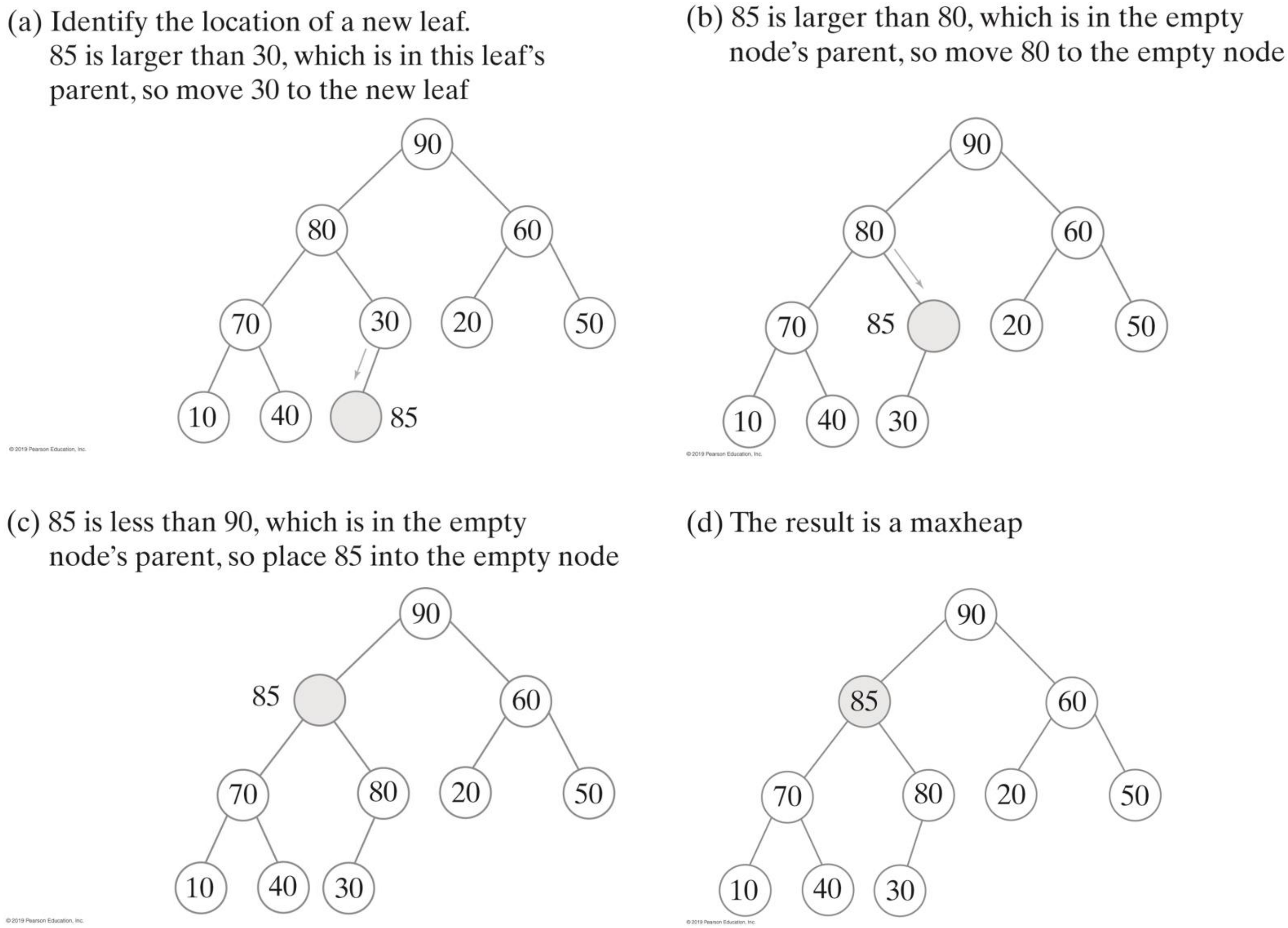
(b) Swap 85 with its parent, 80

(c) The result is a max heap

90
80
60
70
30
20
50
10
40
85
90
80
60
70
85
20
50
10
40
30
90
85
60
70
80
20
50
10
40
30
© 2019 Pearson Education, Inc.
© 2019 Pearson Education, Inc.
© 2019 Pearson Education, Inc.

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

Adding an Entry without 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

© 2019 Pearson Education, Inc.

© 2019 Pearson Education, Inc.

© 2019 Pearson Education, Inc.

© 2019 Pearson Education, Inc.

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

Pearson

Adding an Entry to Heap (Part 1)

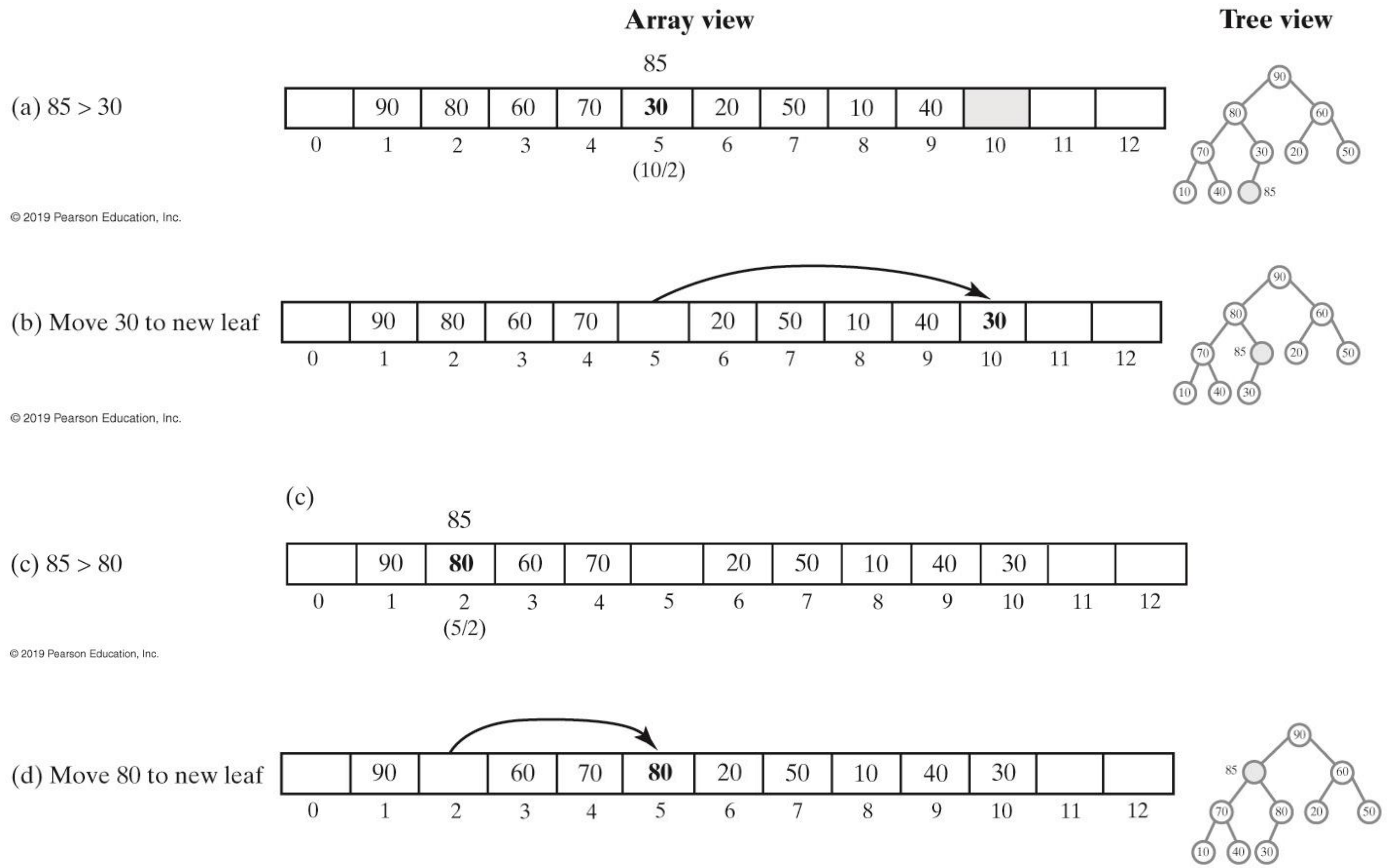
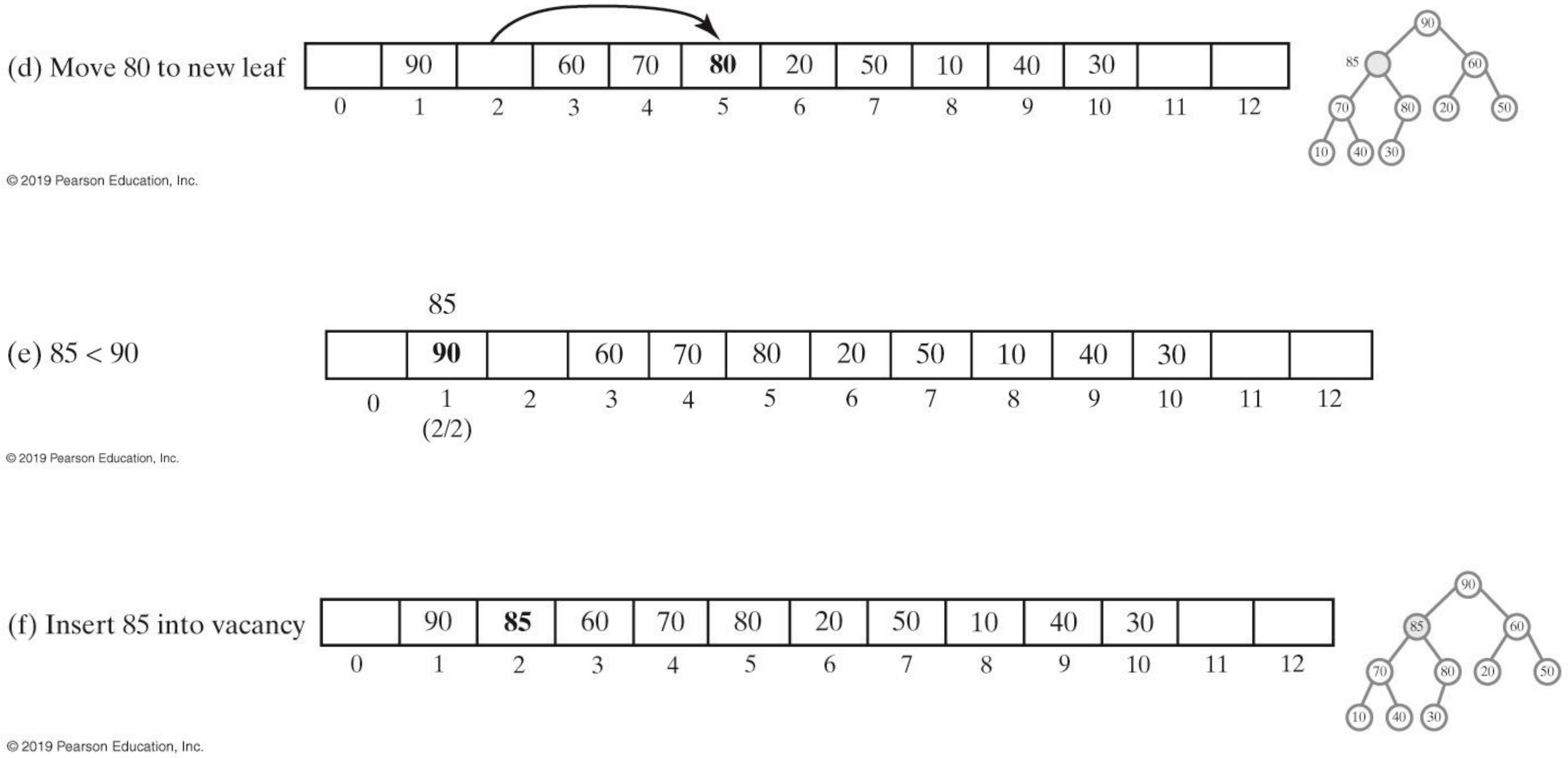


FIGURE 27-4 An array representation of the steps in Figure 27-3

Pearson

Adding an Entry to Heap (Part 2)



(d) Move 80 to new leaf

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

(e) 85 < 90

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

(f) Insert 85 into vacancy

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

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

Adding an Entry

```
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
```

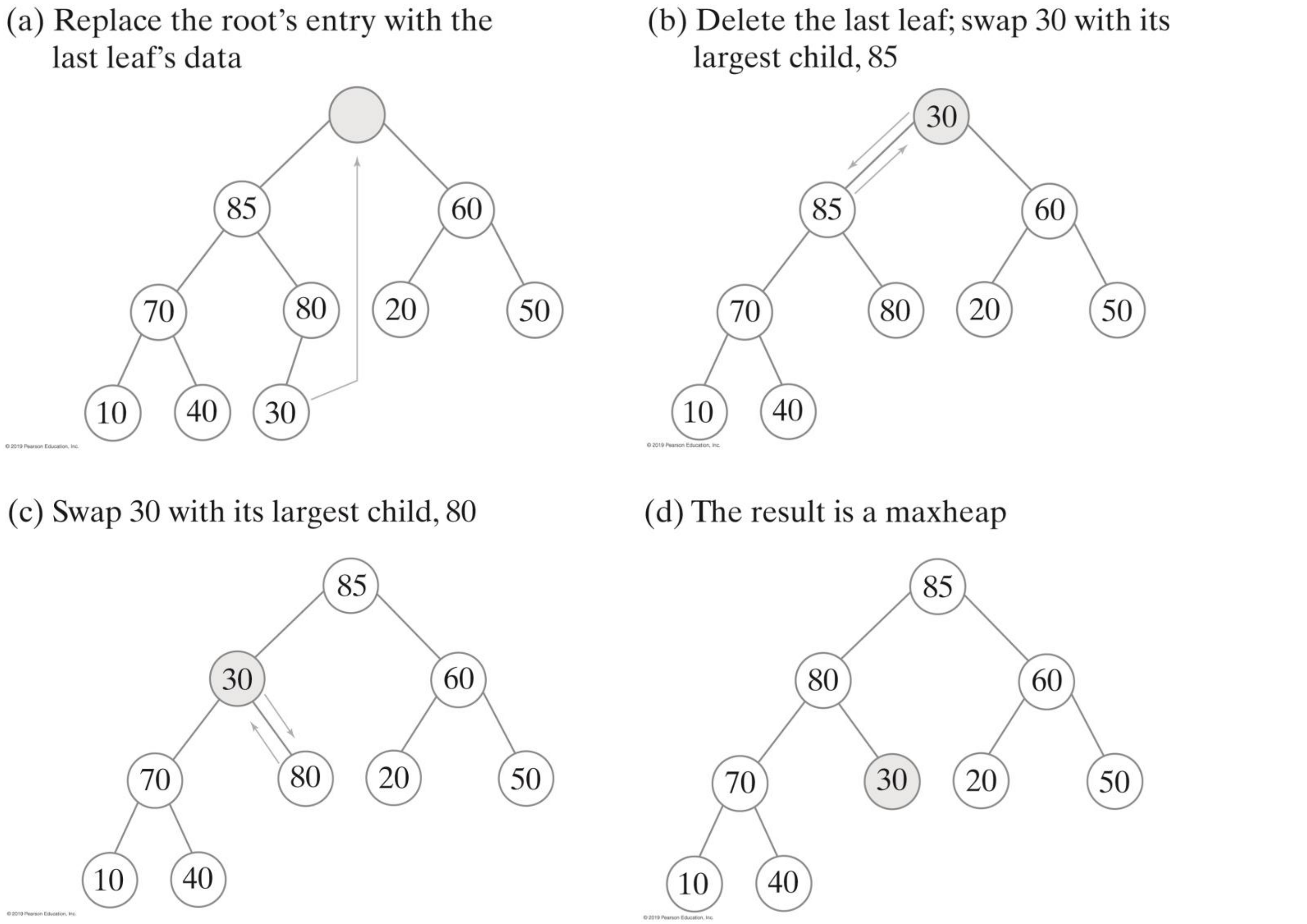
Algorithm to add a new entry to a heap

Adding an Entry

```
public void add(T newEntry)
{
   checkIntegrity();        // Ensure initialization of data fields
   int newIndex = lastIndex + 1;
   int parentIndex = newIndex / 2;
   while ( (parentIndex > 0) && newEntry.compareTo(heap[parentIndex]) > 0)
   {
      heap[newIndex] = heap[parentIndex];
      newIndex = parentIndex;
      parentIndex = newIndex / 2;
   } // end while

   heap[newIndex] = newEntry;
   lastIndex++;
   ensureCapacity();
} // end add
```

The method add

Removing a Value from A Heap



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

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

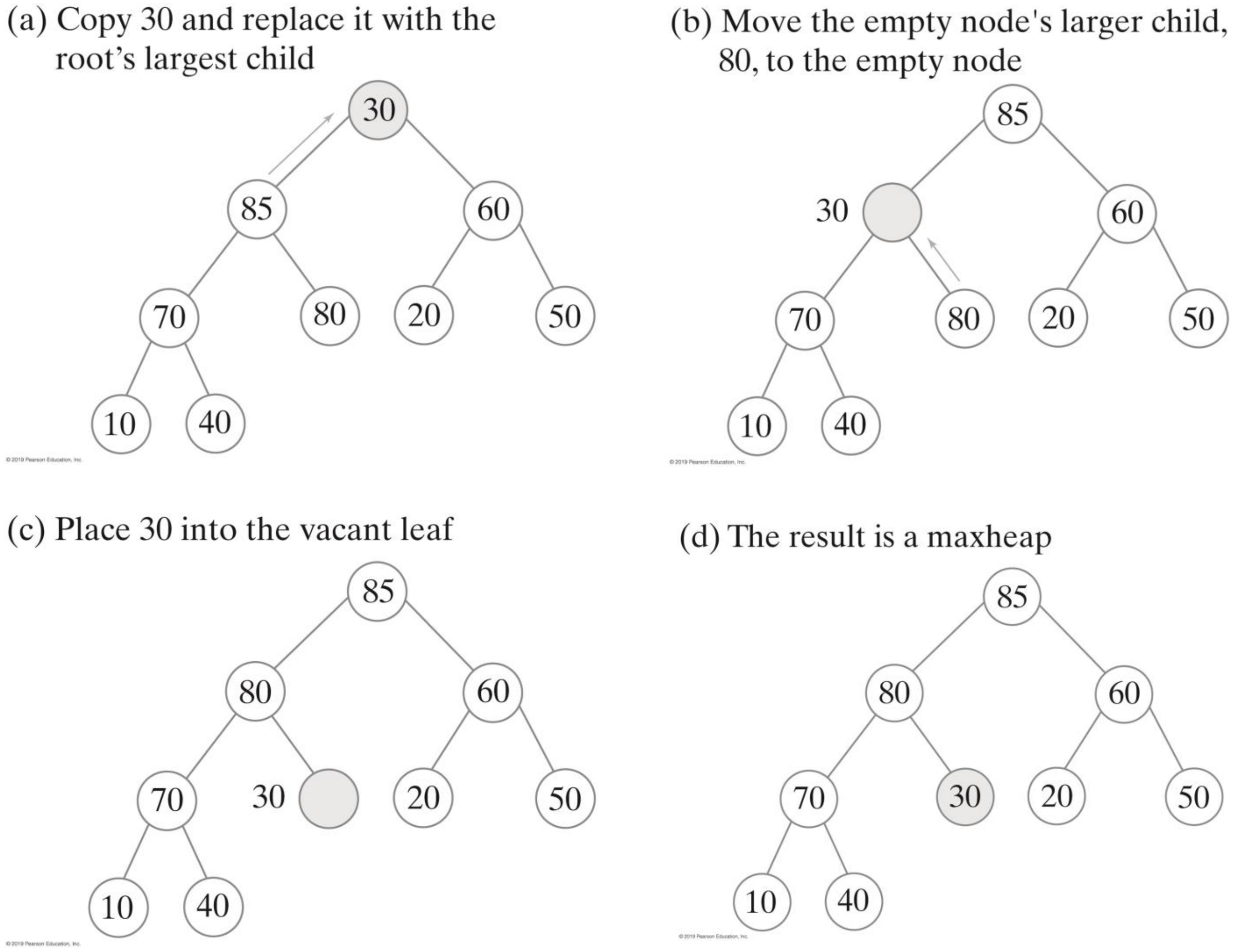
(c) Swap 30 with its largest child, 80

(d) The result is a maxheap

© 2019 Pearson Education, Inc.

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

Removing a Value without 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

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

Removing the Root

```
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
```

Algorithm to transform a semiheap to a heap

Pearson

Removing the Root

```
private void reheap(int rootIndex)
{
   boolean done = false;
   T orphan = heap[rootIndex];
   int leftChildIndex = 2 * rootIndex;

   while (!done && (leftChildIndex <= lastIndex) )
   {
      int largerChildIndex = leftChildIndex; // Assume larger
      int rightChildIndex = leftChildIndex + 1;

      if ( (rightChildIndex <= lastIndex) &&
            heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)
      {
         largerChildIndex = rightChildIndex;
      } // end if

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

Implementation of the reheap algorithm as a private method

Removing the Root

```
public T removeMax()
{
   checkIntegrity();          // Ensure initialization of data fields
   T root = null;

   if (!isEmpty())
   {
      root = heap[1];         // Return value
      heap[1] = heap[lastIndex]; // Form a semiheap
      lastIndex--;            // Decrease size
      reheap(1);              // Transform to a heap
   } // end if

   return root;
} // end removeMax
```

Implementation of the `removeMax` method

Creating a Heap

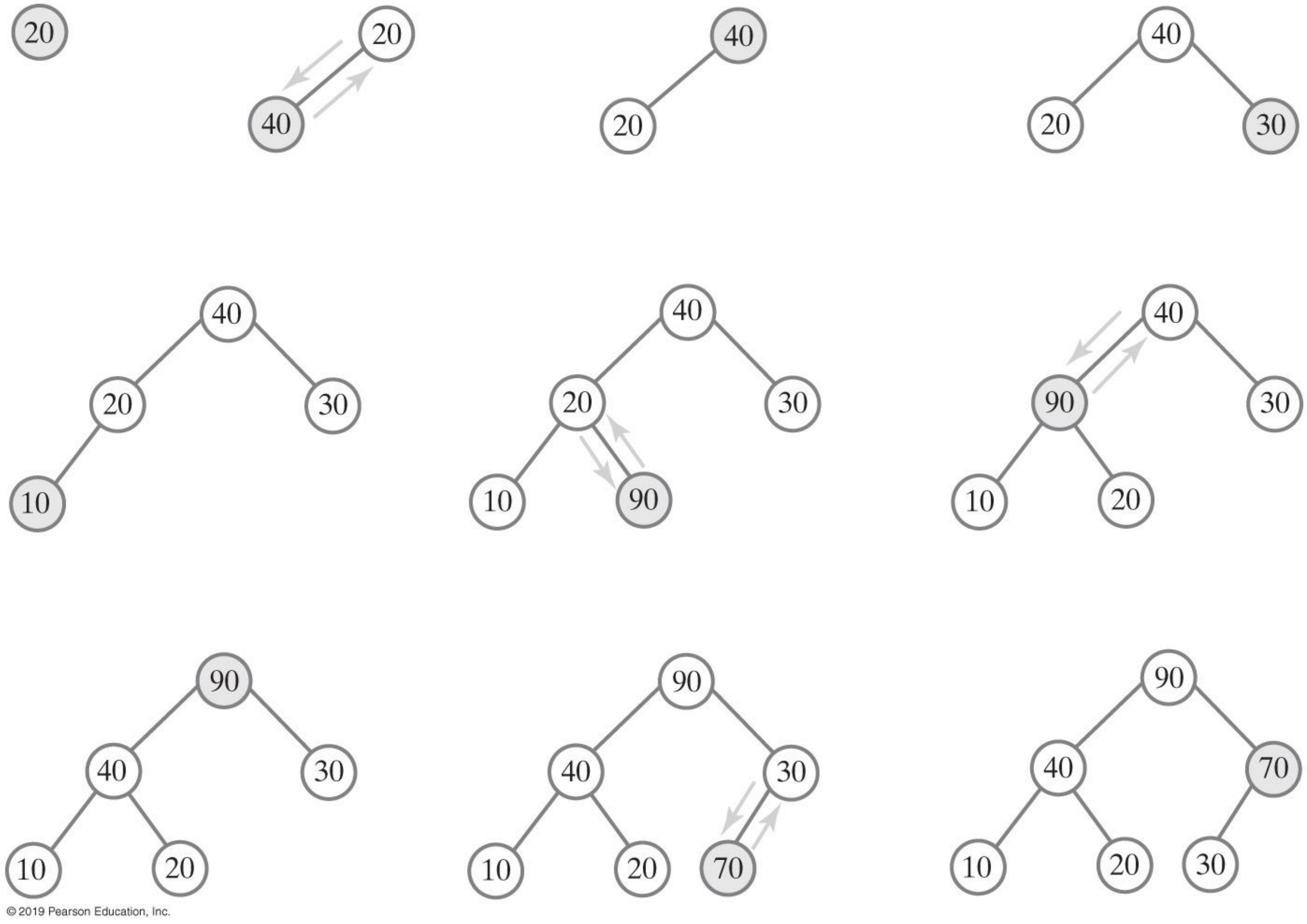
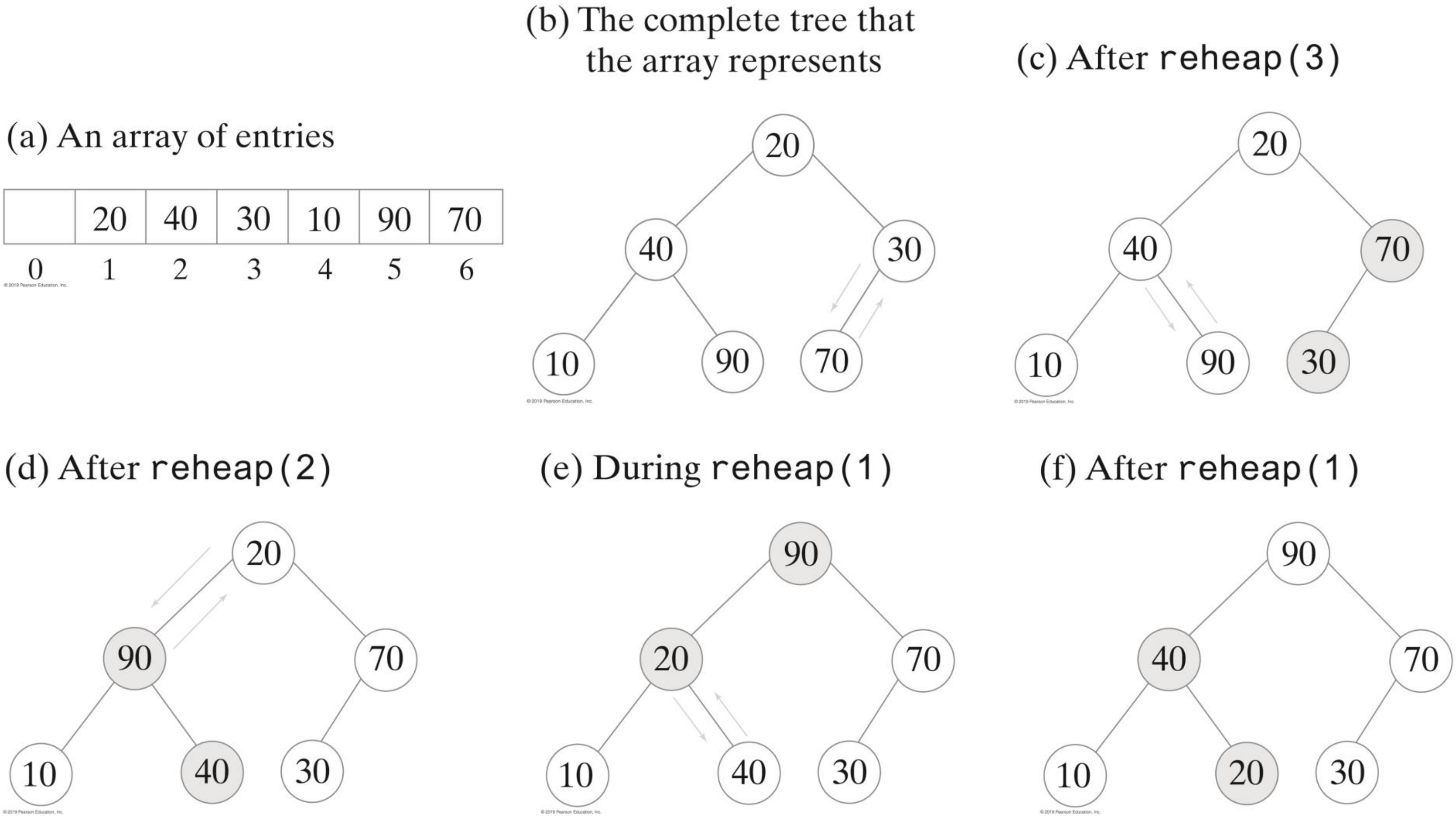


FIGURE 27-7 The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap

Creating a Heap



(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)

FIGURE 27-8 The steps in creating a heap of the entries 20, 40, 30, 10, 90, and 70 by using `reheap`

Creating a Heap

```
public MaxHeap(T[] entries)
{
   this(entries.length); // Call other constructor
   lastIndex = entries.length;
   // Assertion: integrityOK = true

   // Copy given array to data field
   for (int index = 0; index < entries.length; index++)
      heap[index + 1] = entries[index];

   // Create heap
   for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
      reheap(rootIndex);
} // end constructor
```

Another constructor for the class **`MaxHeap`**

Heap Sort

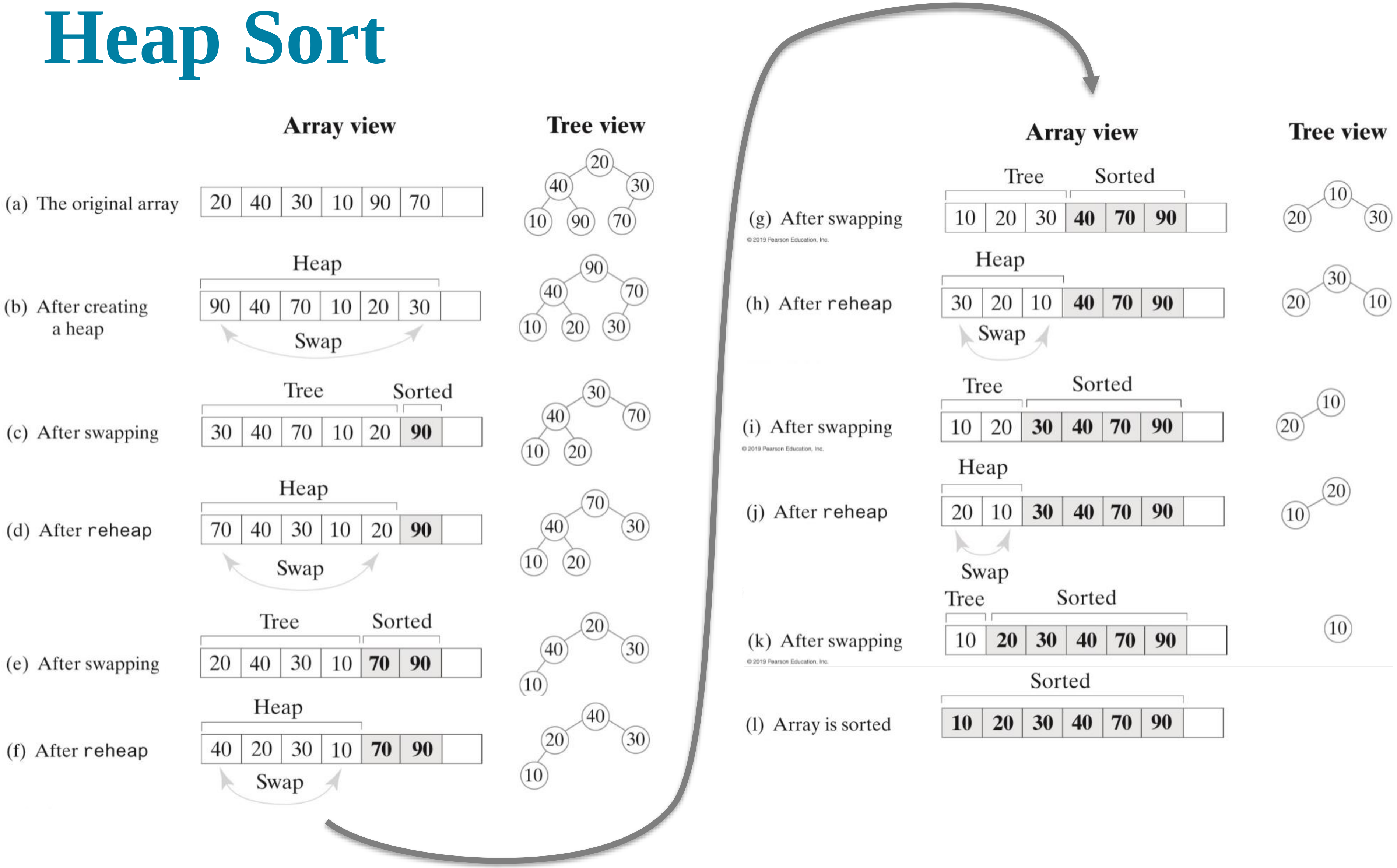


FIGURE 27-9 A trace of heap sort

Heap Sort - Revised reheap Method

```
private static <T extends Comparable<? super T>>
      void reheap(T[] heap, int rootIndex, int lastIndex)
{
  boolean done = false;
  T orphan = heap[rootIndex];
  int leftChildIndex = 2 * rootIndex + 1;

  while (!done && (leftChildIndex <= lastIndex))
  {
    int largerChildIndex = leftChildIndex;
    int rightChildIndex = leftChildIndex + 1;

    if ( (rightChildIndex <= lastIndex) &&
        heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)
    {
      largerChildIndex = rightChildIndex;
    } // end if

    if (orphan.compareTo(heap[largerChildIndex]) < 0)
    {
      heap[rootIndex] = heap[largerChildIndex];
      rootIndex = largerChildIndex;
      leftChildIndex = 2 * rootIndex + 1;
    }
    else
      done = true;
  } // end while
  heap[rootIndex] = orphan;
} // end reheap
```

Heap Sort

```
public static <T extends Comparable<? super T>>
       void heapSort(T[] array, int n)
{
   // Create first heap
   for (int rootIndex = n / 2 - 1; rootIndex >= 0; rootIndex--)
      reheap(array, rootIndex, n - 1);

   swap(array, 0, n - 1);

   for (int lastIndex = n - 2; lastIndex > 0; lastIndex--)
   {
      reheap(array, 0, lastIndex);
      swap(array, 0, lastIndex);
   } // end for
} // end heapSort
```

The `heapSort` method with time efficiency is $O(n \log n)$

End

Chapter 27