A Heap Implementation

Chapter 26

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

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

Heap and Maxheap

```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   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 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

An Array to Represent a Heap

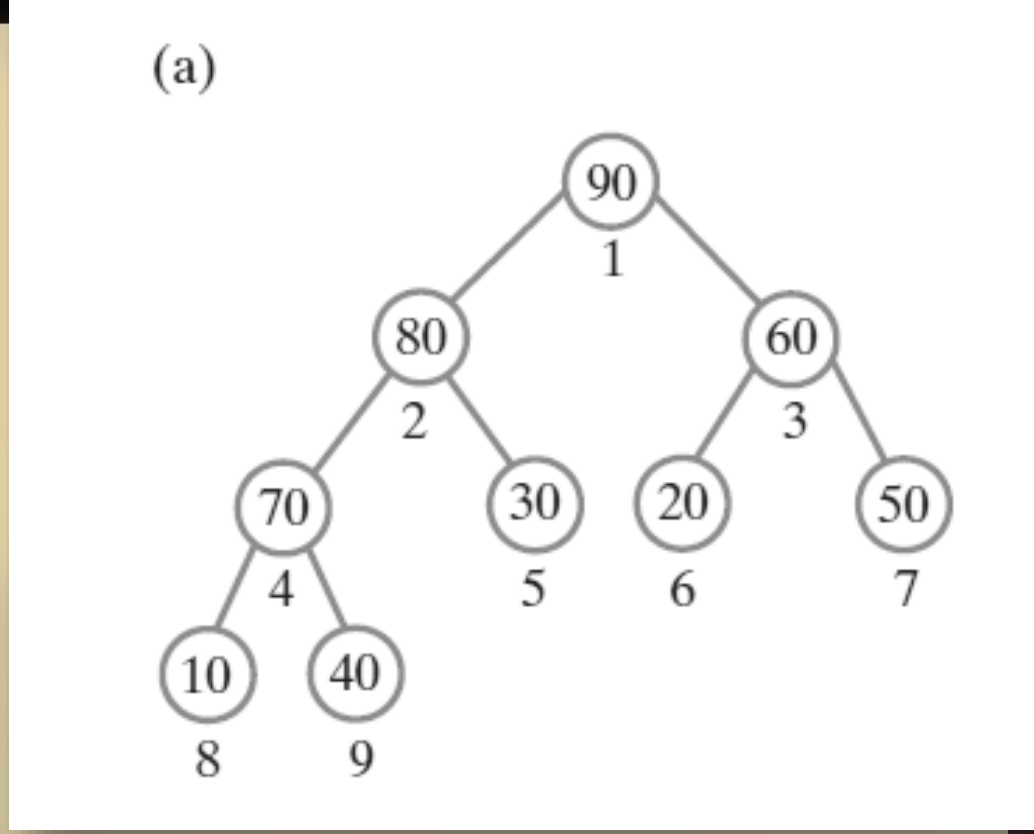


FIGURE 26-1 (a) A complete binary tree with its nodes numbered in level order; (b) its representation as an array

An Array to Represent a Heap

```
1  import java.util.Arrays;
2  public final class MaxHeap<T extends Comparable<? super T>>
3                implements MaxHeapInterface<T>
4  {
5     private T[] heap;       // Array of heap entries
6     private int lastIndex; // Index of last entry
7     private boolean initialized = false;
8     private static final int DEFAULT_CAPACITY = 25;
9     private static final int MAX_CAPACITY = 10000;
10
11    public MaxHeap()
12    {
13       this(DEFAULT_CAPACITY); // Call next constructor
14    } // end default constructor
15
```


LISTING 26-1 The class **MaxHeap**, partially completed

An Array to Represent a Heap

```
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 all null entries
   @SuppressWarnings("unchecked")
   T[] tempHeap = (T[]) new Comparable[initialCapacity + 1];
   heap = tempHeap;
   lastIndex = 0;
   initialized = true;
} // end constructor

public void add(T newEntry)
{
   < See Segment 26.8. >
} // end add

public T removeMax()
```

An Array to Represent a Heap

```
37    public T removeMax()
38    {
39       < See Segment 26.12. >
40    } // end removeMax
41
42    public T getMax()
43    {
44       checkInitialization();
45       T root = null;
46       if (!isEmpty())
47          root = heap[1];
48       return root;
49    } // end getMax
```

LISTING 26-1 The class **MaxHeap**, partially completed

An Array to Represent a Heap

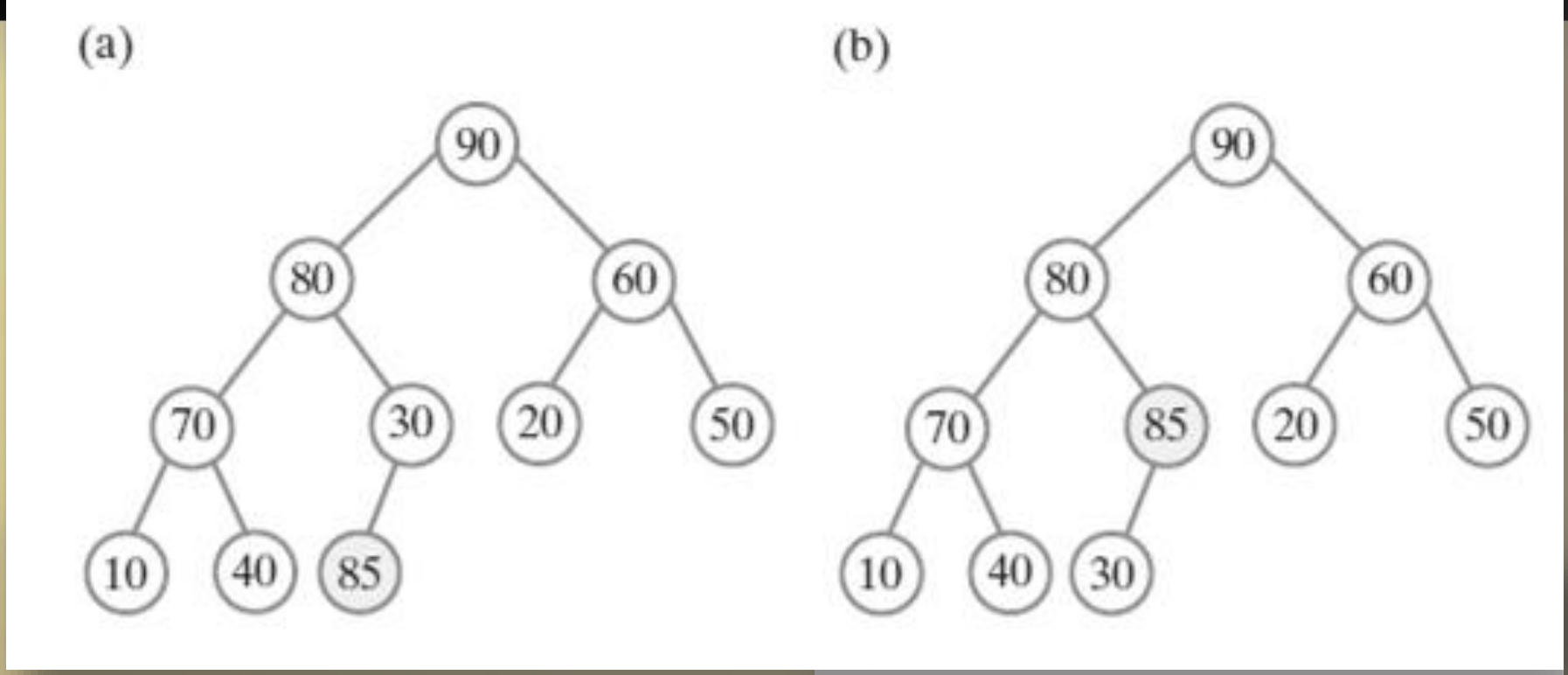
```
50
51      public boolean isEmpty()
52      {
53          return lastIndex < 1;
54      } // end isEmpty
55
56      public int getSize()
57      {
58          return lastIndex;
59      } // end getSize
60
```

LISTING 26-1 The class **MaxHeap**, partially completed

An Array to Represent a Heap

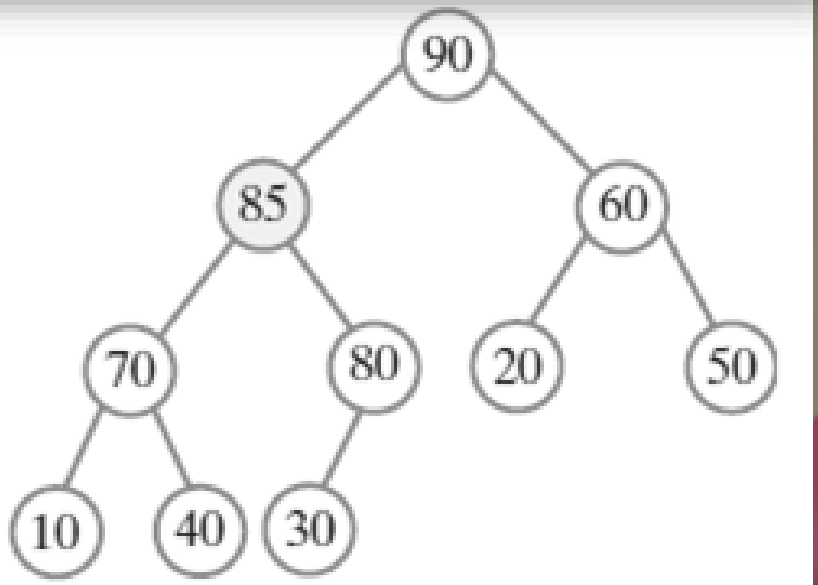
```
60
61      public void clear()
62      {
63         checkInitialization();
64         while (lastIndex > -1)
65         {
66            heap[lastIndex] = null;
67            lastIndex--;
68         } // end while
69         lastIndex = 0;
70      } // end clear
71      < Private methods >
72      . . .
73   } // end MaxHeap
```

LISTING 26-1 The class **MaxHeap**, partially completed



Adding an Entry

FIGURE 26-2 The steps in adding 85 to the maxheap in Figure 26-1a



Adding an Entry

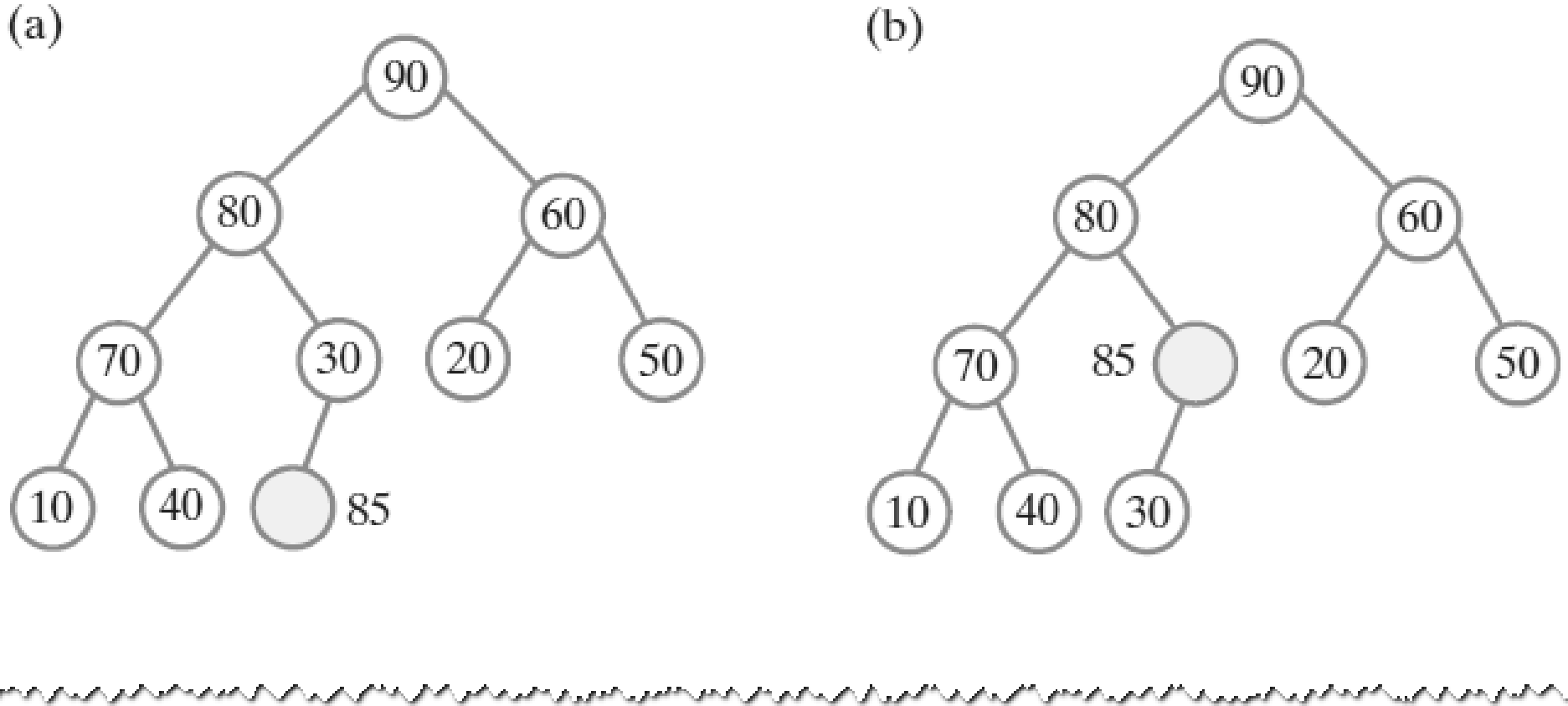


FIGURE 26-3 A revision of the steps shown in Figure 26-2, to avoid swaps

Adding an Entry

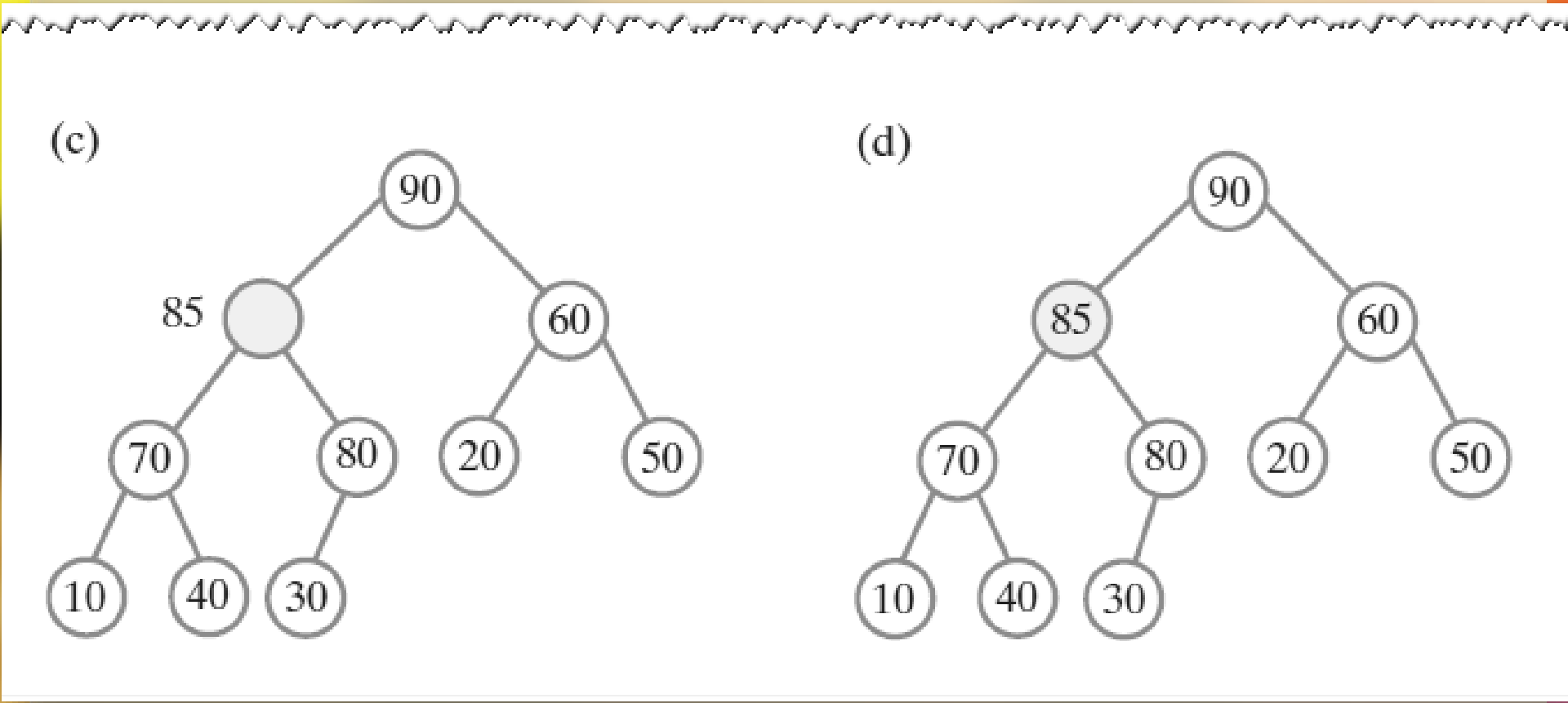


FIGURE 26-3 A revision of the steps shown in Figure 26-2, to avoid swaps

Adding an Entry

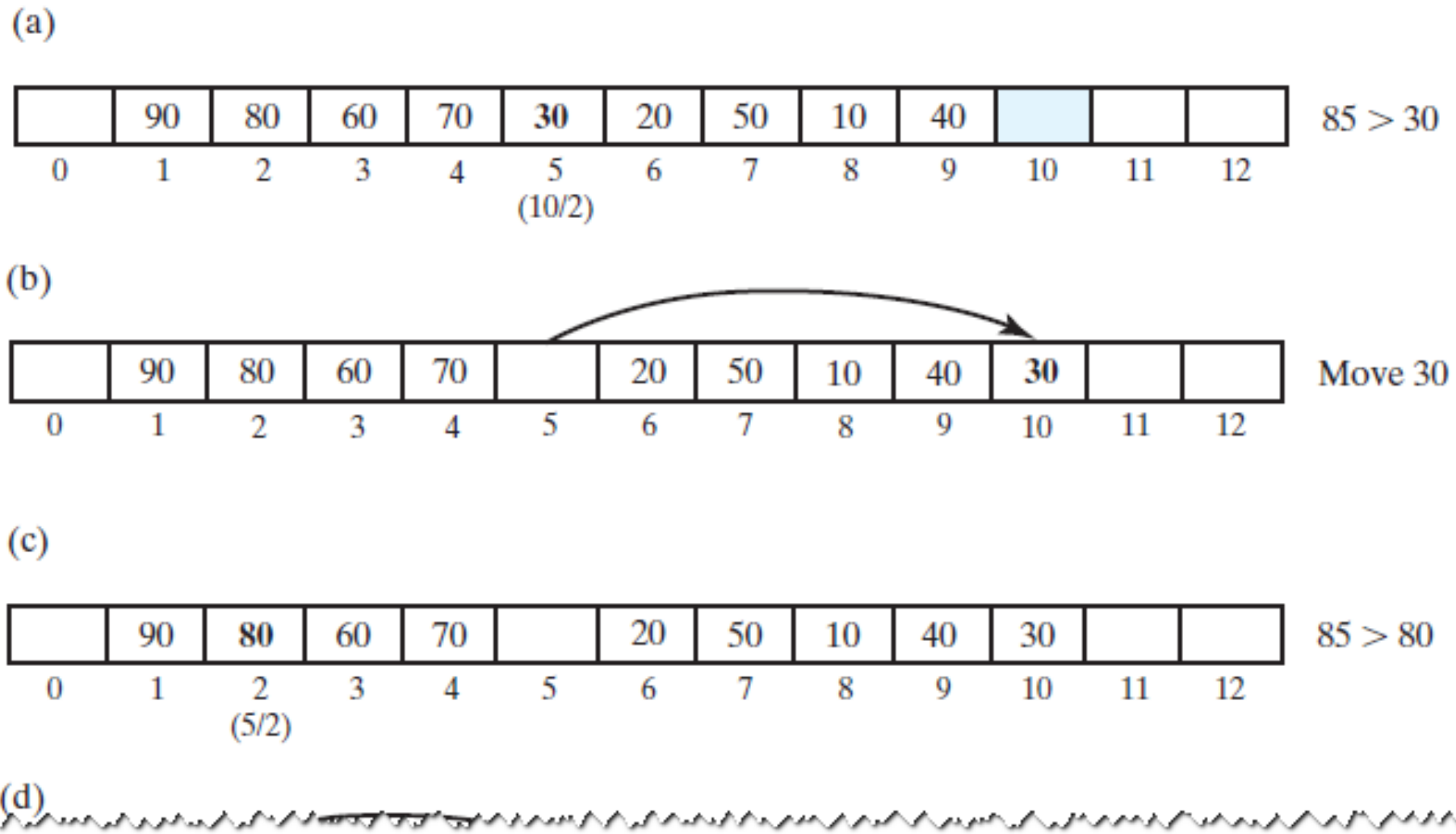


FIGURE 26-4 An array representation of the steps in Figure 26-3

Adding an Entry

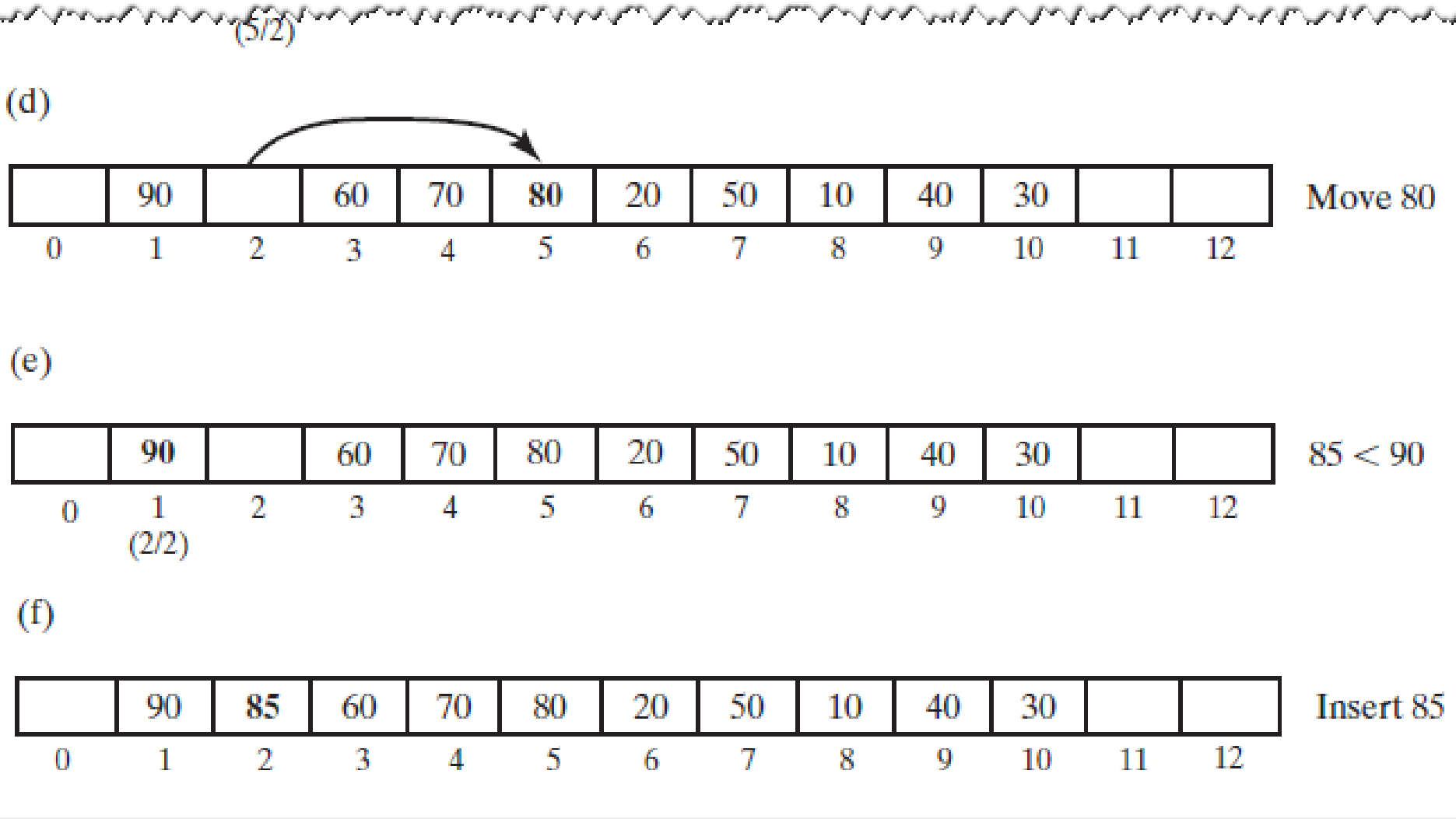


FIGURE 26-4 An array representation of the steps in Figure 26-3

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)
{
   checkInitialization();          // 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 the Root

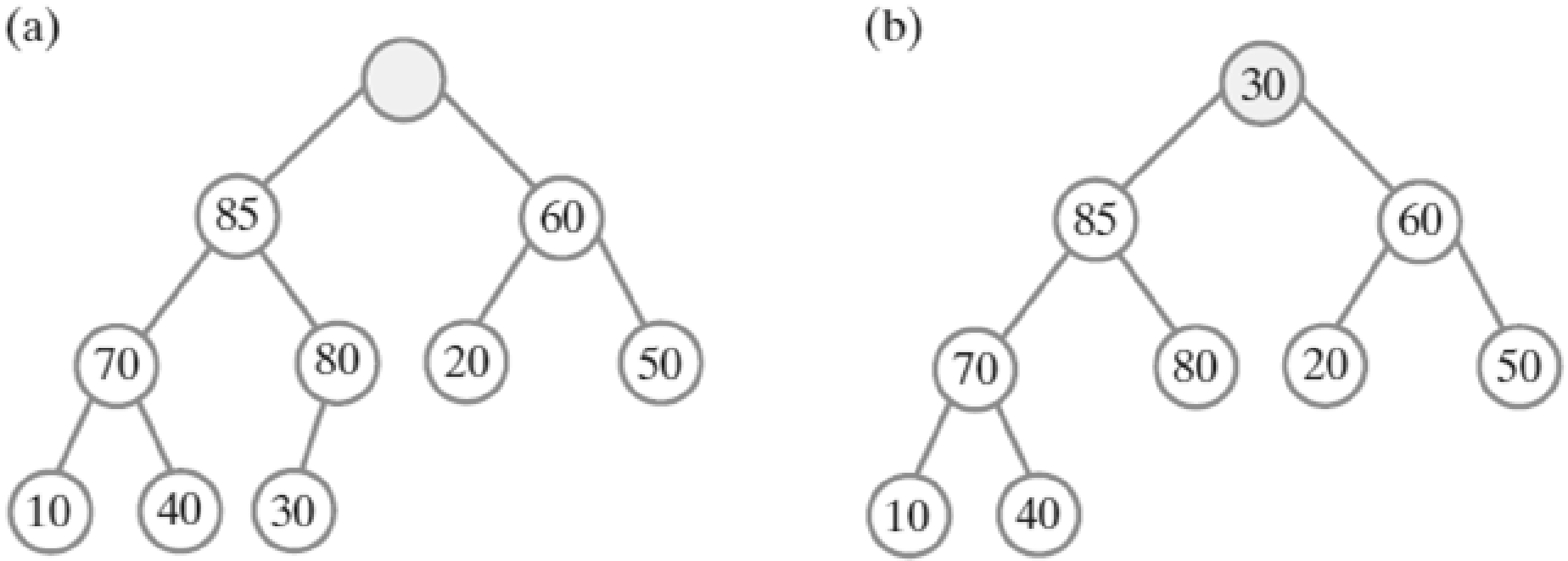


FIGURE 26-5 The steps to remove the entry in the root of the maxheap in Figure 26-3d

Removing the Root

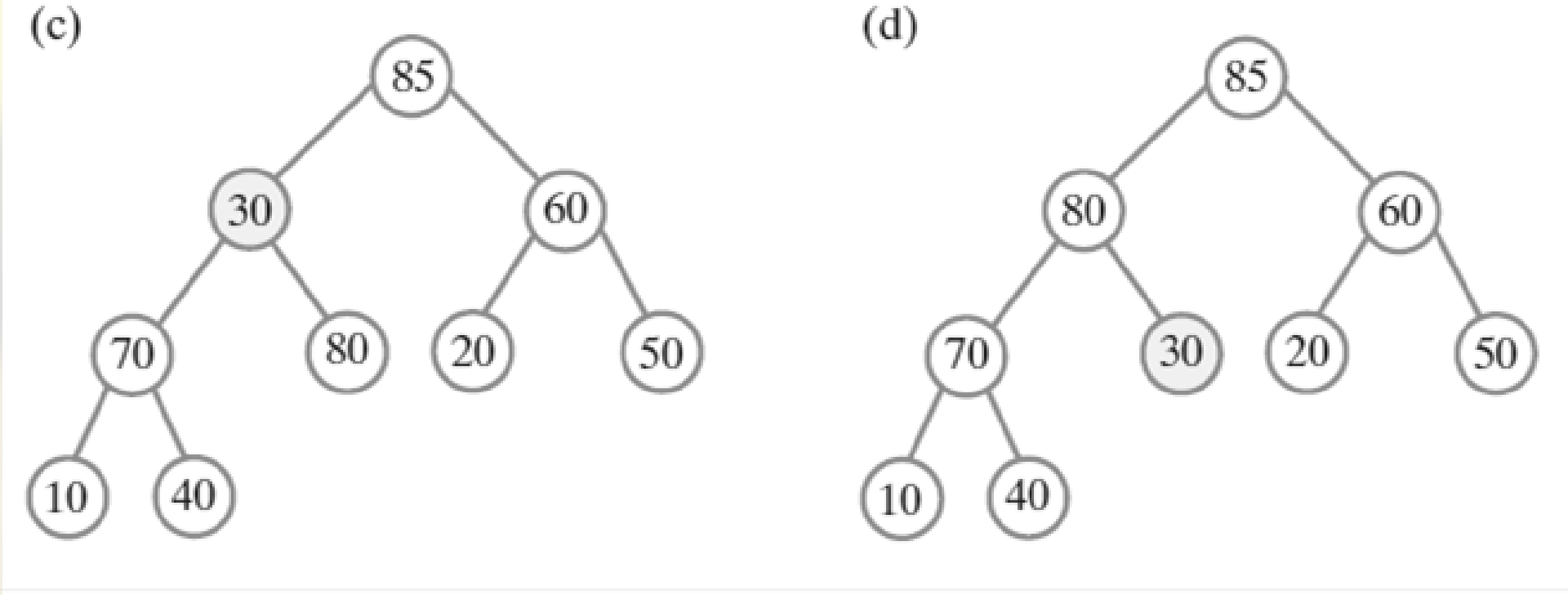


FIGURE 26-5 The steps to remove the entry in the root of the maxheap in Figure 26-3d

Removing the Root

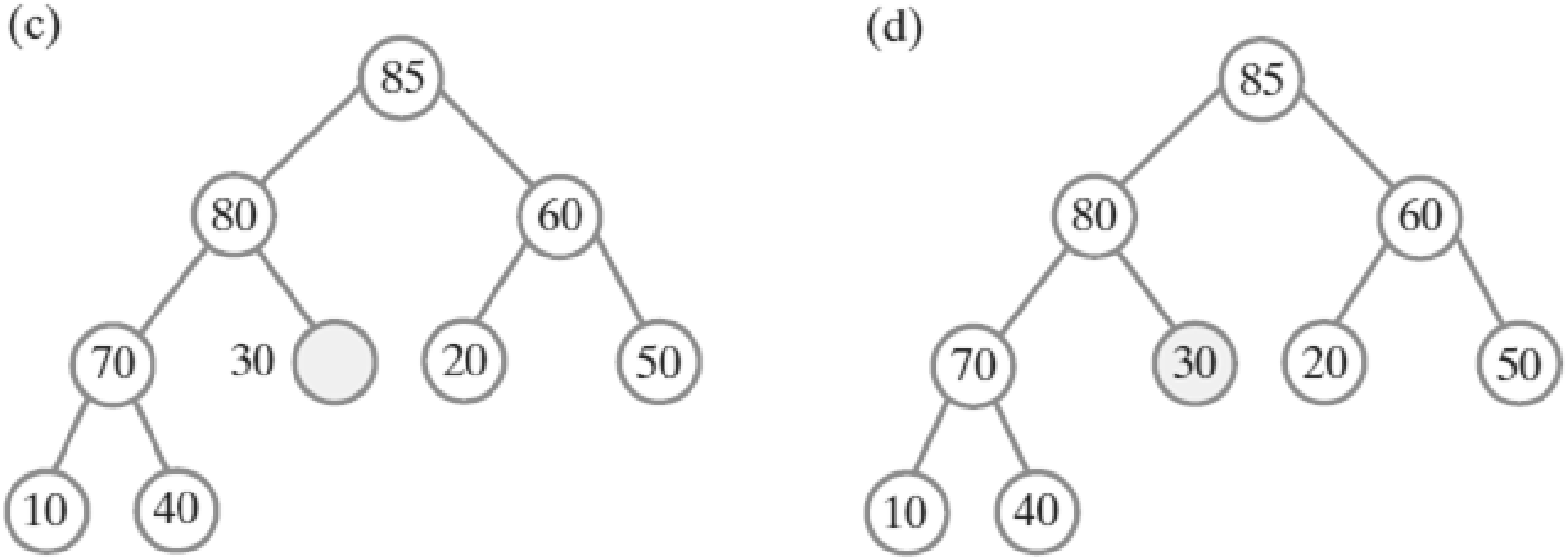


FIGURE 26-6 The steps that transform a semiheap into a heap without swaps

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

Removing the Root

```
public T removeMax()
{
   checkInitialization();          // 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

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)
```

Implementation of the **reheap** algorithm as a private method

Removing the Root

```
        } // 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

Creating a Heap

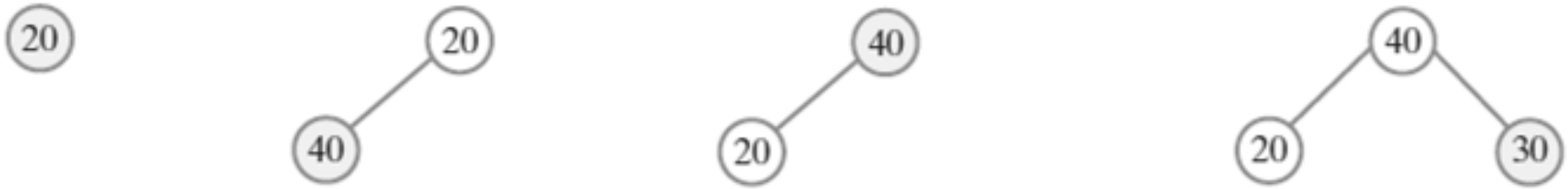


FIGURE 26-7 The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap

Creating a Heap

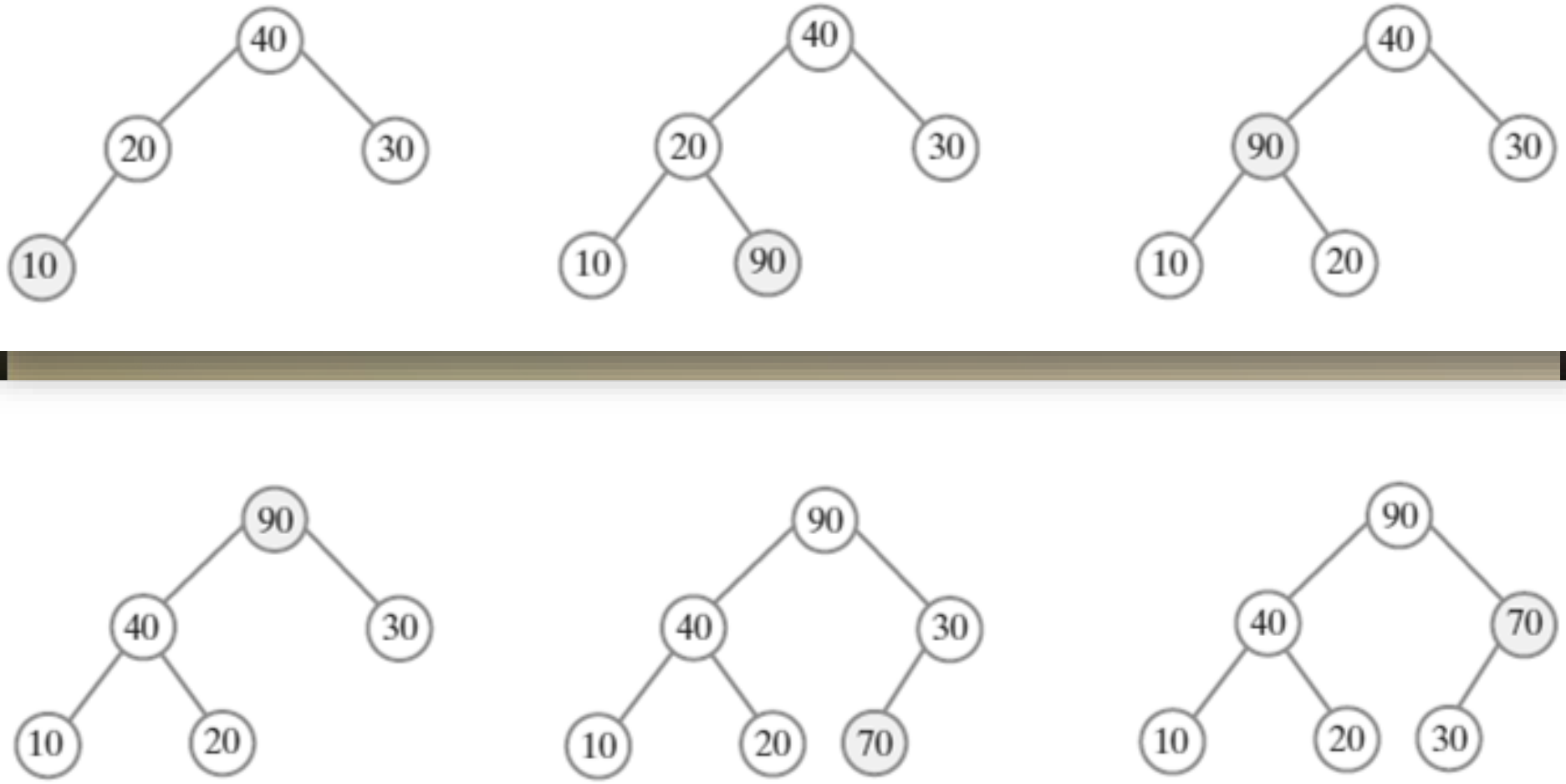


FIGURE 26-7 The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap

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

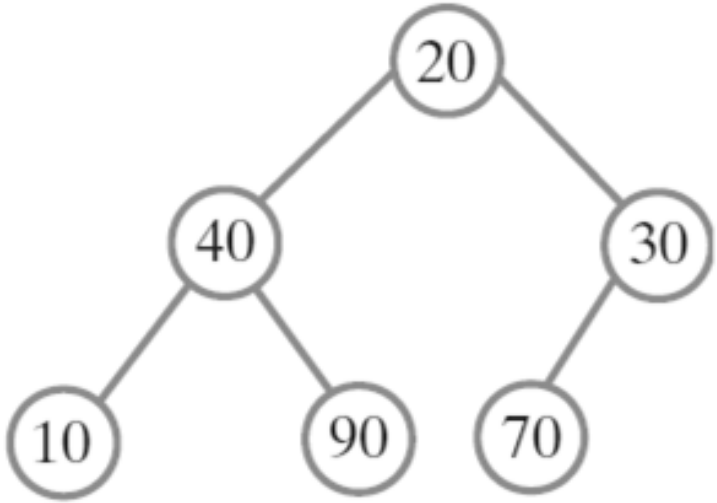


FIGURE 26-8 The steps in creating a heap of the entries 20, 40, 30, 10, 90, and 70 by using `reheap`

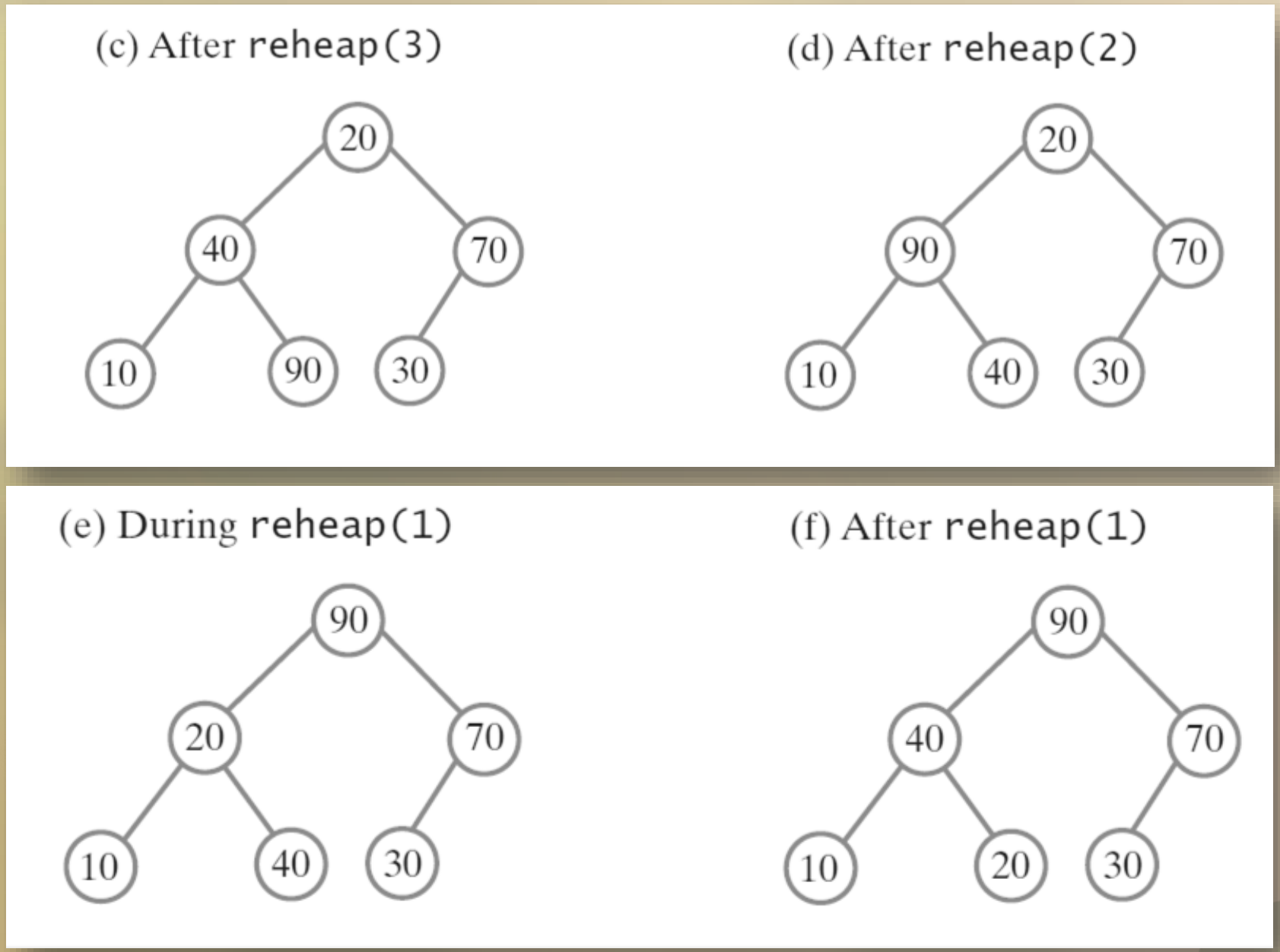


FIGURE 26-8 The steps in creating a heap of the entries 20, 40, 30, 10, 90, and 70 by using **reheap**

Creating a Heap

```
public MaxHeap(T[] entries)
{
   this(entries.length); // Call other constructor
   assert initialized = 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

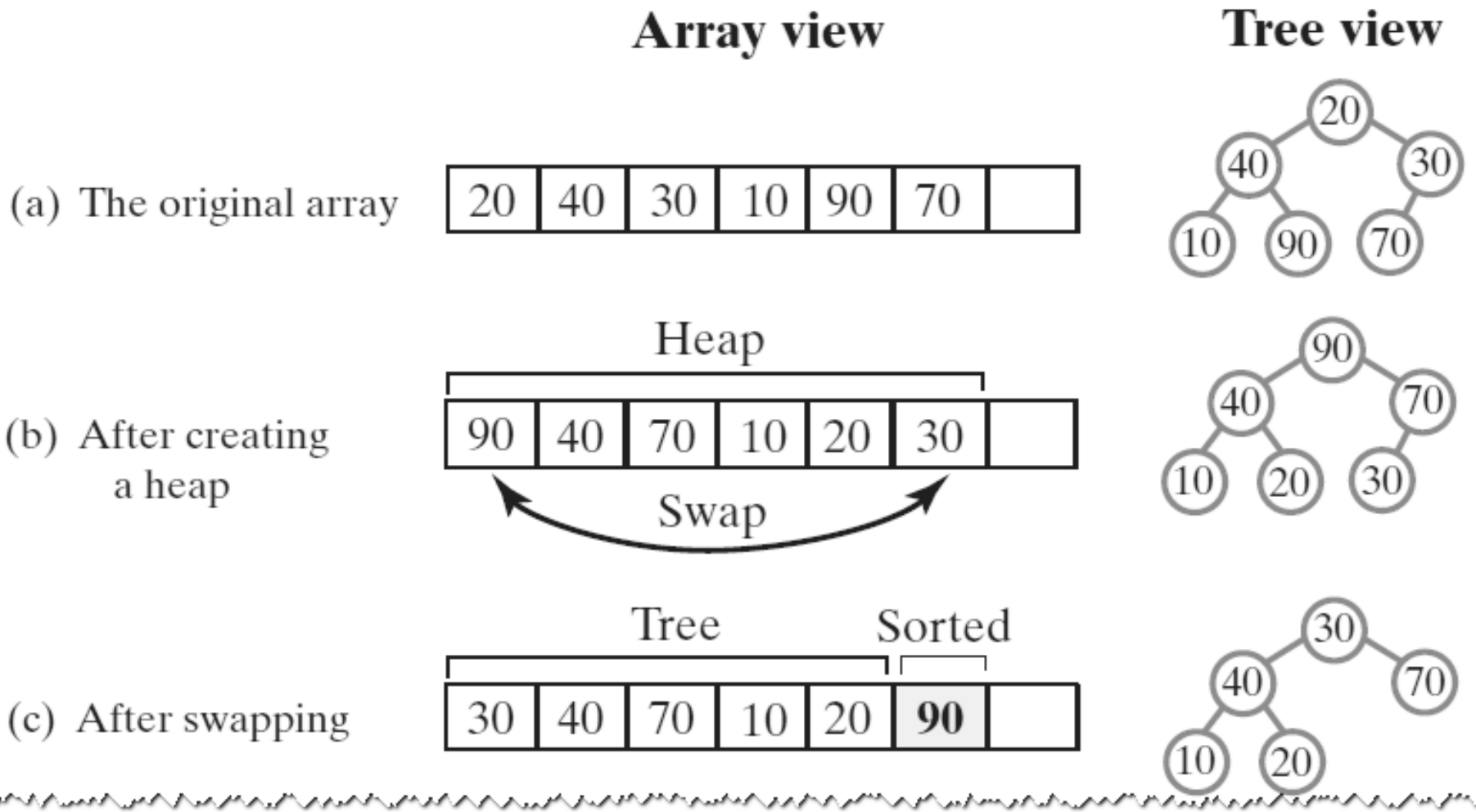


FIGURE 26-9 A trace of heap sort

Heap Sort

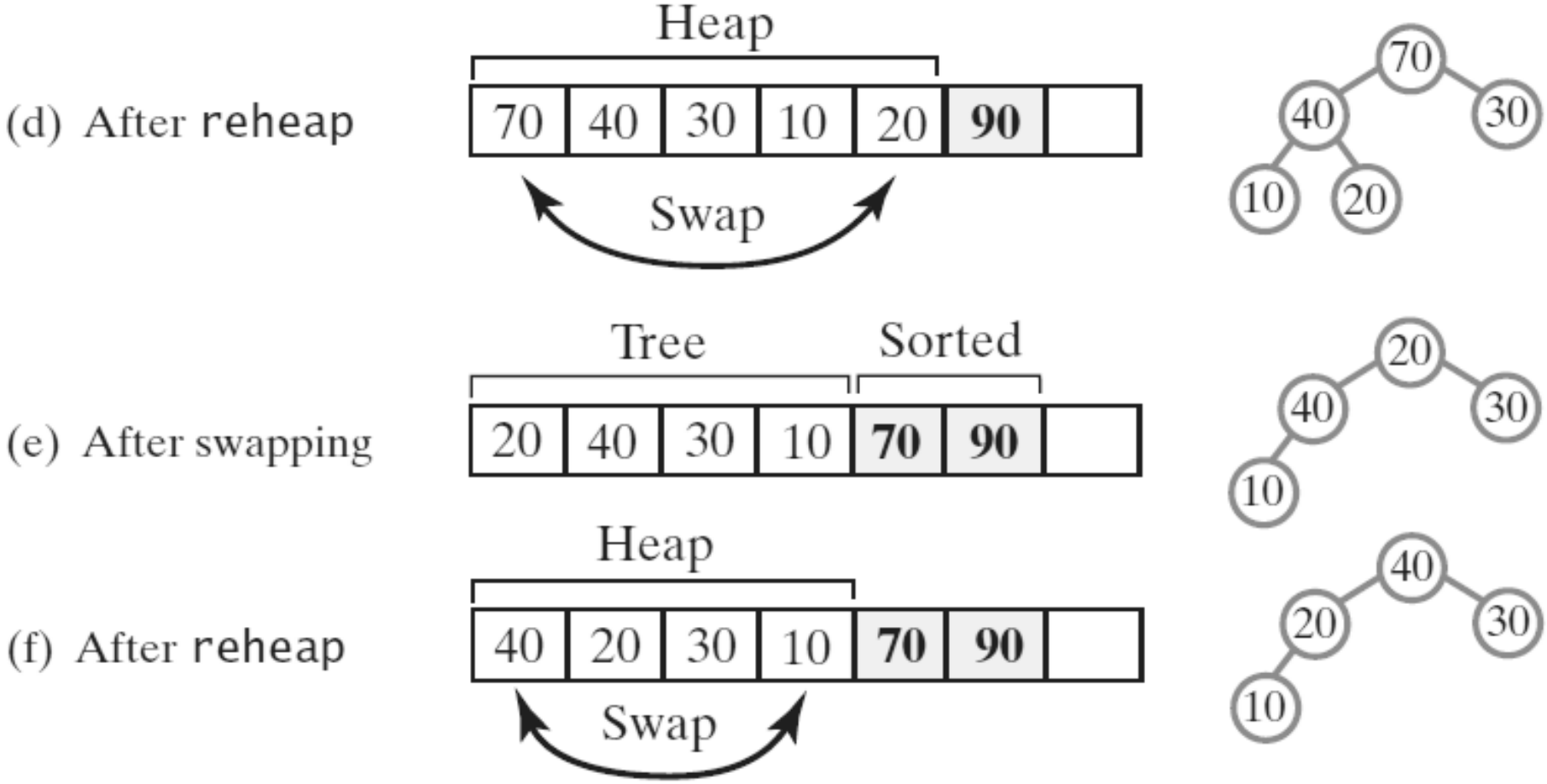


FIGURE 26-9 A trace of heap sort

Heap Sort

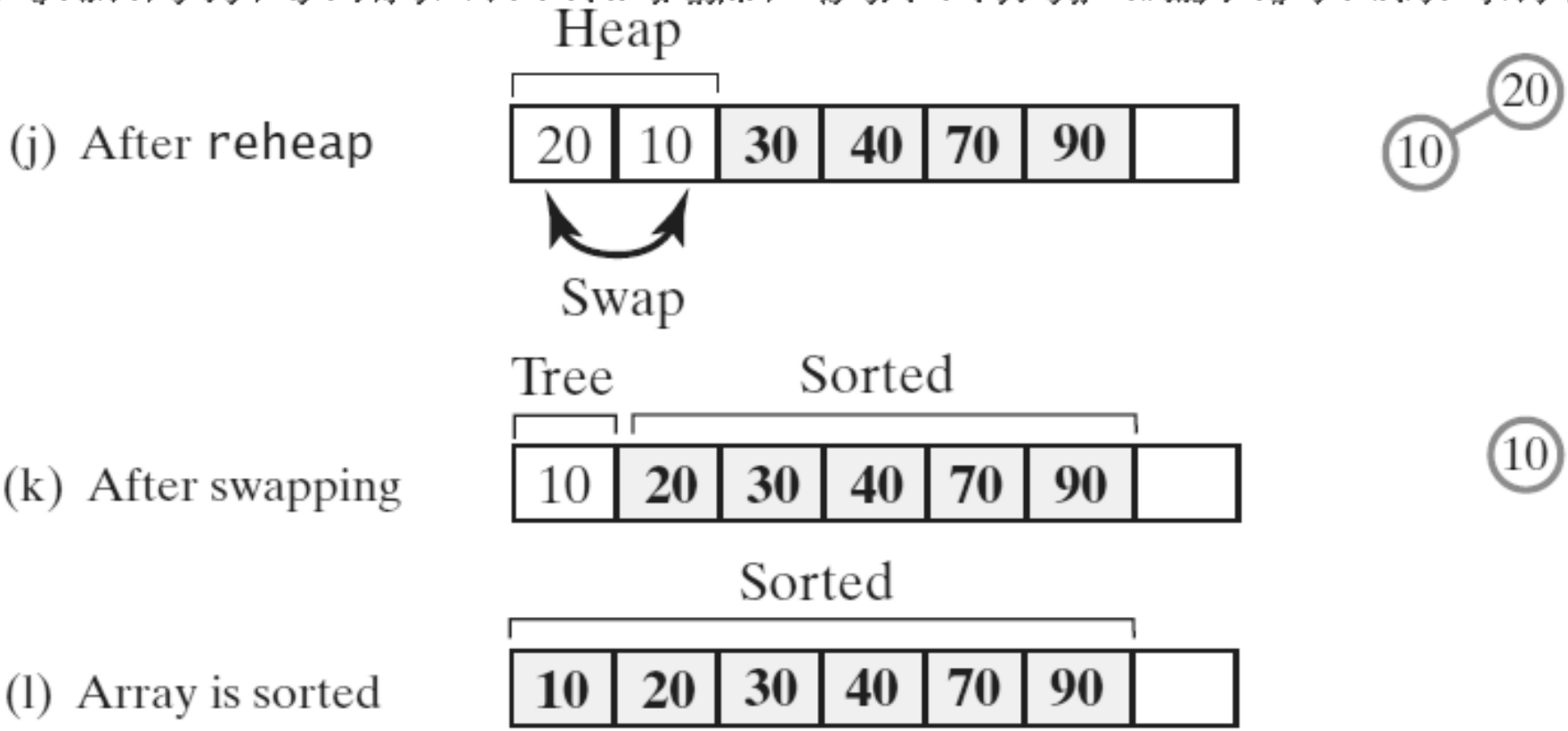


FIGURE 26-9 A trace of heap sort

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

Time efficiency is O(n log n)

Heap Sort

```
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
```


Revised `reheap` method

Heap Sort

```
            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
```


Revised **`reheap`** method

End

Chapter 26