Heaps

Chapter 17

The ADT Heap

- A heap is a complete binary tree that either is
  - Empty or …
  - Whose root contains a value ≥ each of its children and has heaps as its subtrees
- It is a special binary tree … different in that
  - It is ordered in a weaker sense
  - it will always be a complete binary tree

The ADT Heap

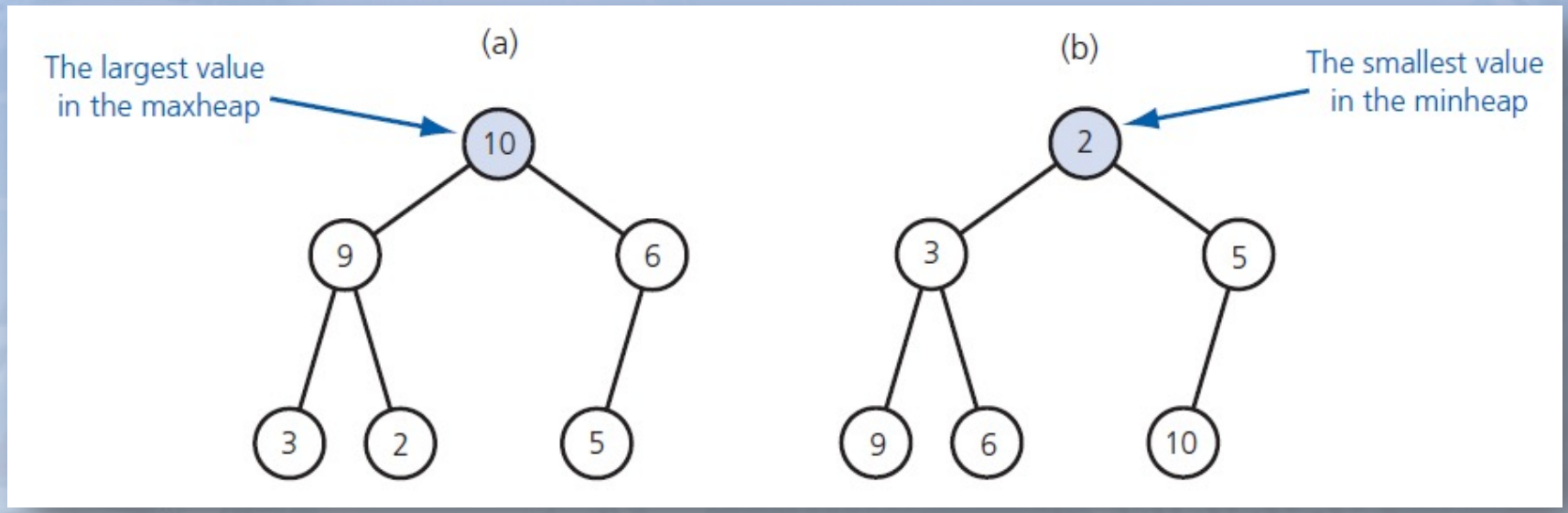

FIGURE 17-1 (a) A maxheap and (b) a minheap

The ADT Heap

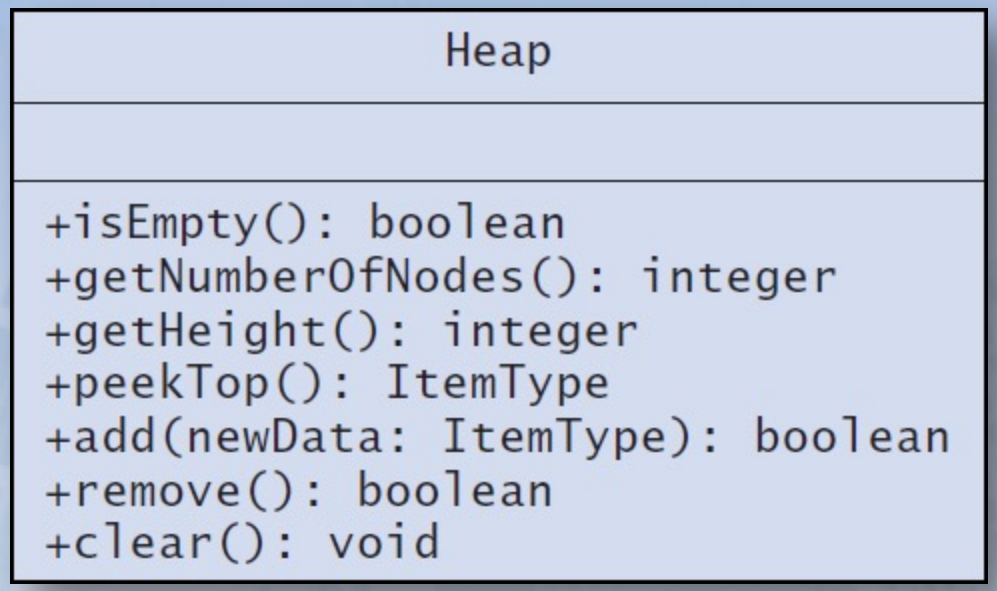

FIGURE 17-2 UML diagram for the class Heap

The ADT Heap

```
1  /** Interface for the ADT heap.
2   @file HeapInterface.h */
3
4  #ifndef HEAP_INTERFACE_
5  #define HEAP_INTERFACE_
6
7  template<class ItemType>
8  class HeapInterface
9  {
10 public:
11    /** Sees whether this heap is empty.
12     @return  True if the heap is empty, or false if not. */
13    virtual bool isEmpty() const = 0;
14
15    /** Gets the number of nodes in this heap.
16     @return  The number of nodes in the heap. */
17    virtual int getNumberOfNodes() const = 0;
```

LISTING 17-1 An interface for the ADT heap

The ADT Heap

```
18
19      /** Gets the height of this heap.
20       @return  The height of the heap. */
21      virtual int getHeight() const = 0;
22
23      /** Gets the data that is in the root (top) of this heap.
24          For a maxheap, the data is the largest value in the heap;
25          for a minheap, the data is the smallest value in the heap.
26       @pre  The heap is not empty.
27       @post  The root's data has been returned, and the heap is unchanged.
28       @return  The data in the root of the heap. */
29      virtual ItemType peekTop() const = 0;
30
31      /** Adds a new data item to this heap.
```

LISTING 17-1 An interface for the ADT heap

The ADT Heap

```
30
31      /** Adds a new data item to this heap.
32       @param newData  The data to be added.
33       @post  The heap has a new node that contains newData.
34       @return  True if the addition is successful, or false if not. */
35      virtual bool add(const ItemType& newData) = 0;
36
37      /** Removes the data that is in the root (top) of this heap.
38       @return  True if the removal is successful, or false if not. */
39      virtual bool remove() = 0;
40
41      /** Removes all data from this heap. */
42      virtual void clear() = 0;
43
44      /** Destroys this heap and frees its assigned memory. */
45       virtual ~HeapInterface() {  }
46  }; // end HeapInterface
47  #endif
```

LISTING 17-1 An interface for the ADT heap

Array-Based Implementation of a Heap

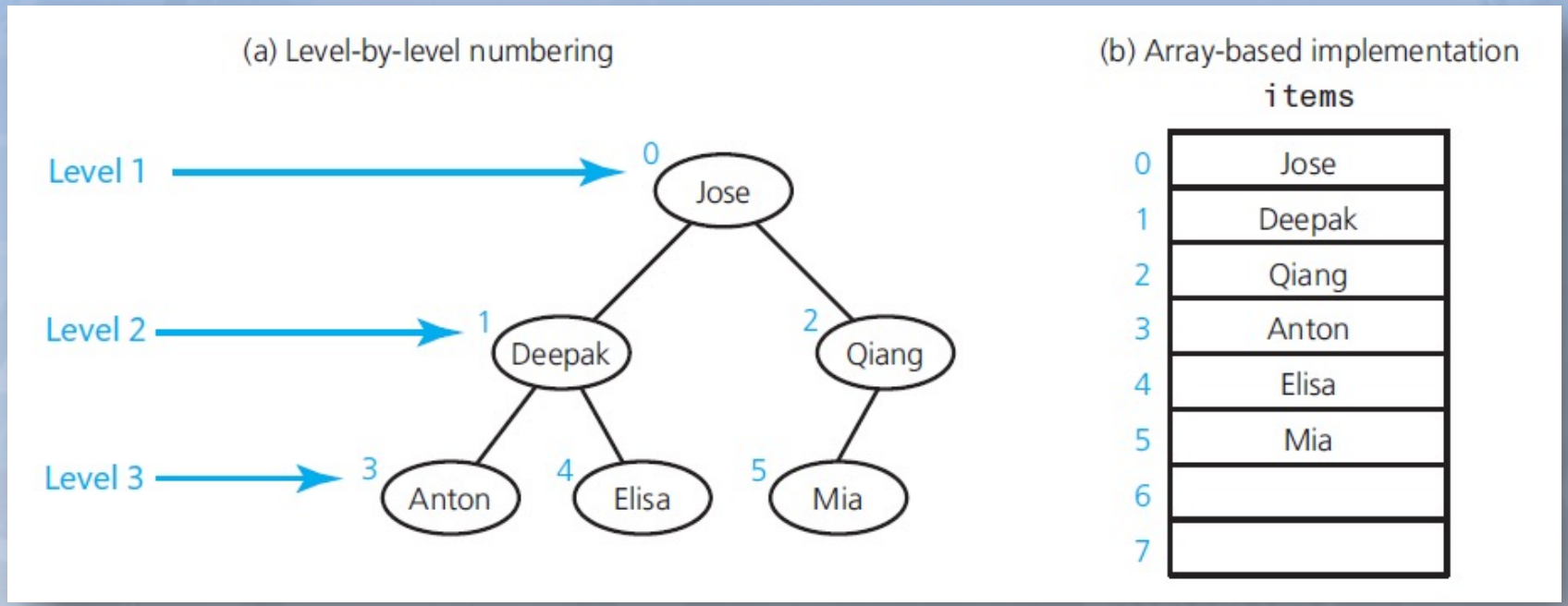

Figure 17-3 A complete binary tree and its array-based implementation

Algorithms for Array-Based Heap Operations

- Assume following private data members
  - items: an array of heap items
  - itemCount: an integer equal to the number of items in the heap
  - maxItems: an integer equal to the maximum capacity of the heap

Algorithms for Array-Based Heap Operations

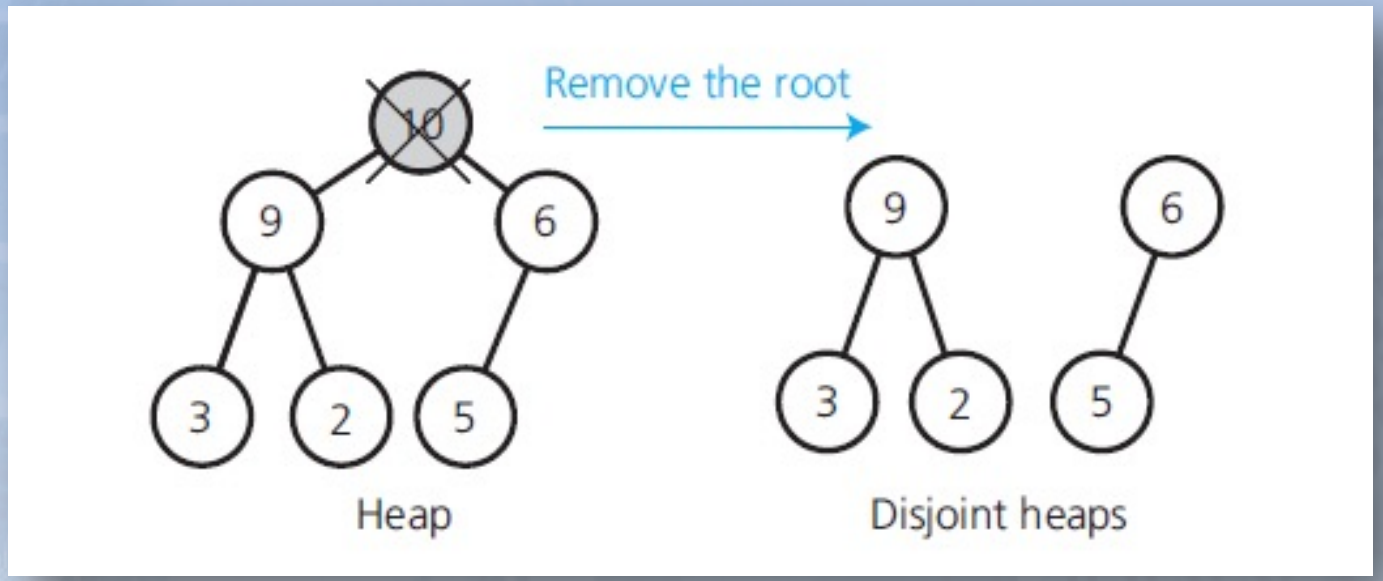

FIGURE 17-4 Disjoint heaps after removing the heap's root

Algorithms for Array-Based Heap Operations

```
// Converts a semiheap rooted at index nodeIndex into a heap.
heapRebuild(nodeIndex: integer, items: ArrayType, itemCount: integer): void
{
   // Recursively trickle the item at index nodeIndex down to its proper position by
   // swapping it with its larger child, if the child is larger than the item.
   // If the item is at a leaf, nothing needs to be done.
   if (the root is not a leaf)
   {
      // The root must have a left child; find larger child
      leftChildIndex = 2 * rootIndex + 1
      rightChildIndex = leftChildIndex + 1
      largerChildIndex = rightChildIndex // Assume right child exists and is the larger
```

Recursive algorithm to transform semiheap to heap.

Algorithms for Array-Based Heap Operations

```
        // Check whether right child exists; if so, is left child larger?
        // If no right child, left one is larger
        if ((largerChildIndex >= itemCount) ||
                                (items[leftChildIndex] > items[rightChildIndex]))
            largerChildIndex = leftChildIndex; // Assumption was wrong
        if (items[nodeIndex] < items[largerChildIndex])
        {
            Swap items[nodeIndex] and items[largerChildIndex]

            // Transform the semiheap rooted at largerChildIndex into a heap
            heapRebuild(largerChildIndex, items, itemCount)
        }
    }
    // Else root is a leaf, so you are done
}
```

Recursive algorithm to transform semiheap to heap.

Algorithms for Array-Based Heap Operations

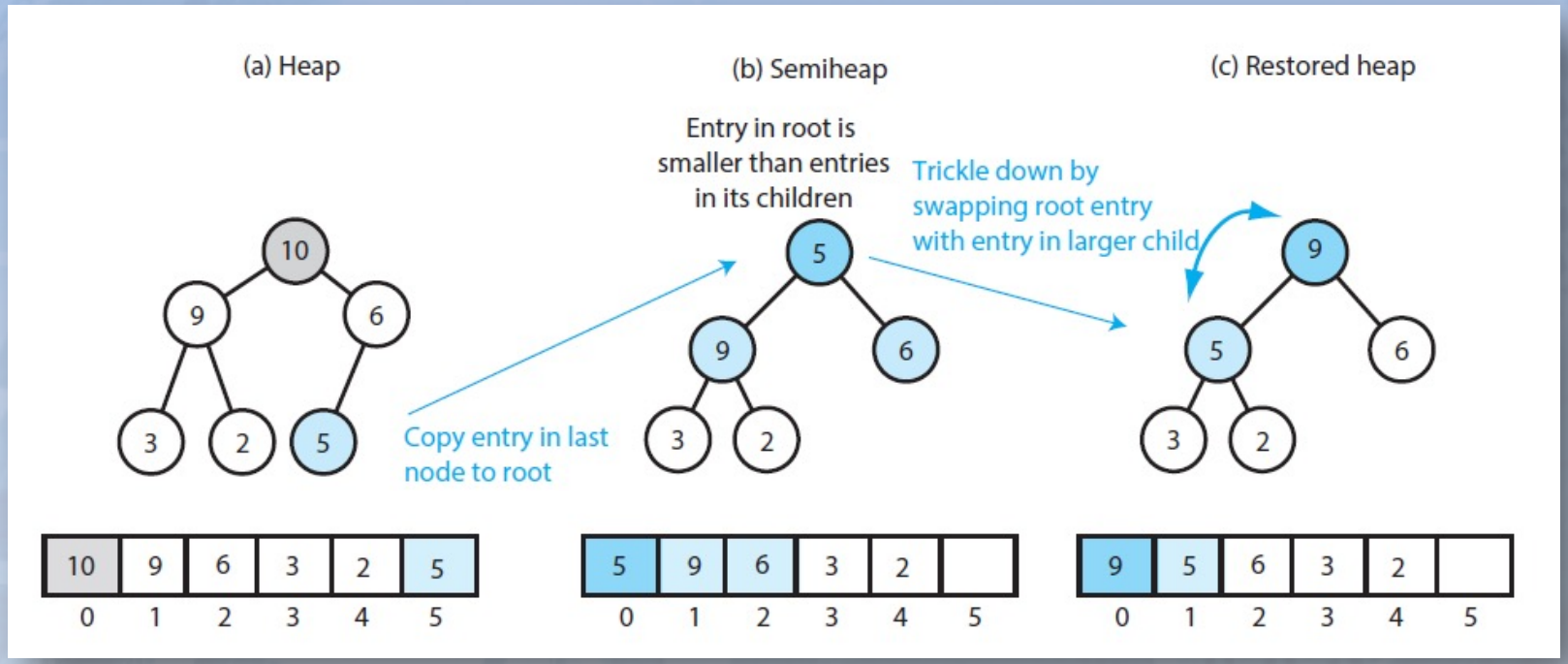

FIGURE 17-5

Algorithms for Array-Based Heap Operations

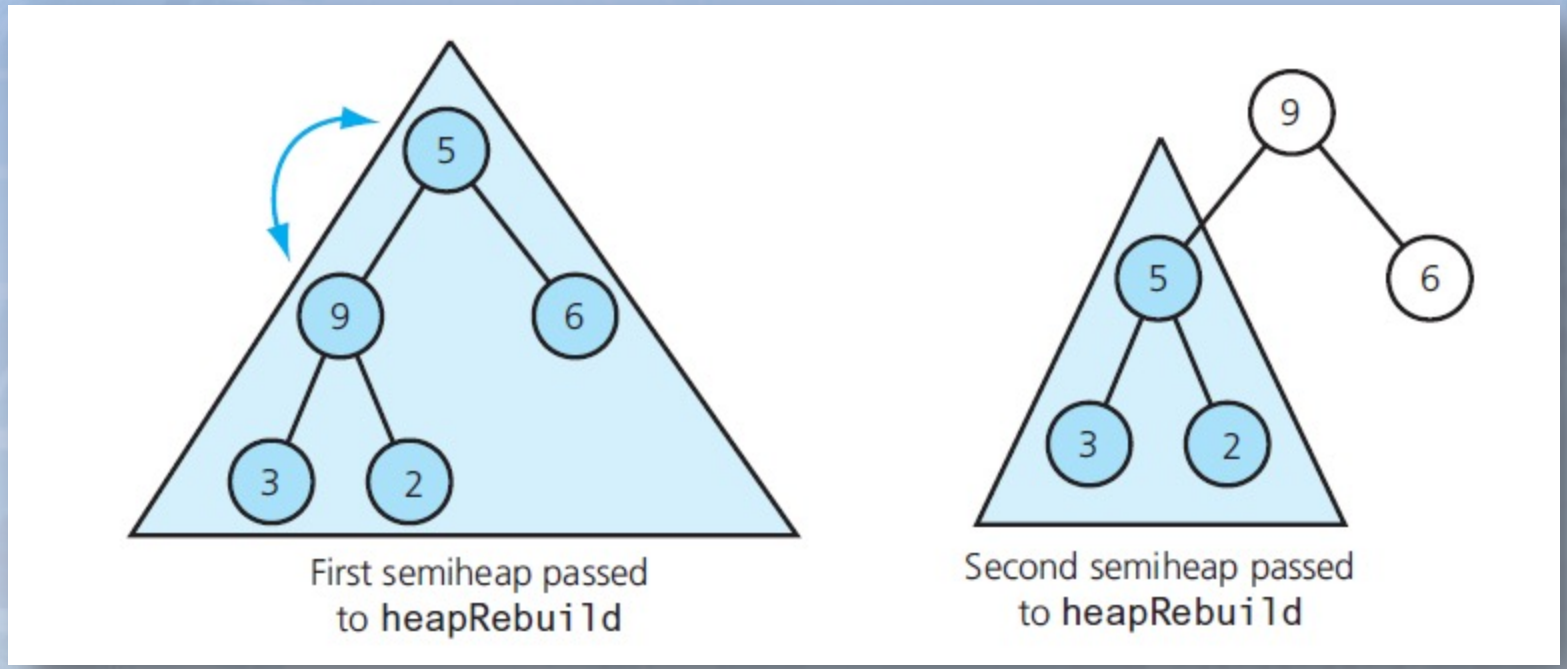

FIGURE 17-6 Recursive calls to heapRebuild

Algorithms for Array-Based Heap Operations

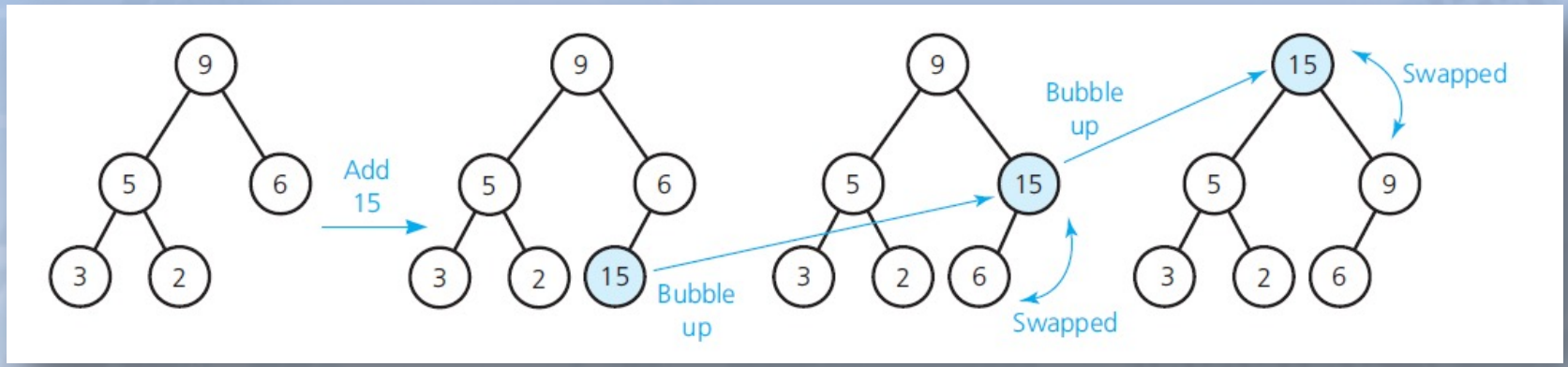

FIGURE 17-5 Adding 15 to a heap

Algorithms for Array-Based Heap Operations

```
add(newData: itemType): boolean
{
   // Place newData at the bottom of the tree
   items[itemCount] = newData

   // Make new item bubble up to the appropriate spot in the tree
   newDataIndex = itemCount
   inPlace = false
   while ((newDataIndex >= 0) and !inPlace)
   {
      parentIndex = (newDataIndex – 1) / 2
      if (items[newDataIndex] <= items[parentIndex])
         inPlace = true
      else
      {
         Swap items[newDataIndex] and items[parentIndex]
         newDataIndex = parentIndex
      }
   }
   itemCount++
   return inPlace
}
```

Pseudocode for add

The Implementation

```
1  /** Array-based implementation of the ADT heap.
2   @file ArrayMaxHeap.h */
3  #ifndef ARRAY_MAX_HEAP_
4  #define ARRAY_MAX_HEAP_
5
6  #include "HeapInterface.h"
7  #include "PrecondViolatedExcept.h"
8
9  template<class ItemType>
10 class ArrayMaxHeap : public HeapInterface<ItemType>
11 {
12 private:
13    static const int ROOT_INDEX = 0;          // Helps with readability
14    static const int DEFAULT_CAPACITY = 21; // Small capacity for testing
15    std::unique_ptr<ItemType[]> items;        // Array of heap items
16    int itemCount;                            // Current count of heap items
17    int maxItems;                             // Maximum capacity of the heap
18
```

LISTING 17-2 The header file for the class ArrayMaxHeap

The Implementation

```
19      // ------------------------------------------------------------
20      // Most of the private utility methods use an array index as a parameter
21      // and in calculations. This should be safe, even though the array is an
22      // implementation detail, since the methods are private.
23      // ------------------------------------------------------------
24
25      // Returns the array index of the left child (if it exists).
26      int getLeftChildIndex(const int nodeIndex) const;
27
28      // Returns the array index of the right child (if it exists).
29      int getRightChildIndex(int nodeIndex) const;
30
31      // Returns the array index of the parent node.
32      int getParentIndex(int nodeIndex) const;
33
34      // Tests whether this node is a leaf.
35      bool isLeaf(int nodeIndex) const;
36
```

LISTING 17-2 The header file for the class ArrayMaxHeap

The Implementation

```
37      // Converts a semiheap to a heap.
38      void heapRebuild(int nodeIndex);
39
40      // Creates a heap from an unordered array.
41      void heapCreate();
42
43  public:
44      ArrayMaxHeap();
45      ArrayMaxHeap(const ItemType someArray[], const int arraySize);
46      virtual ~ArrayMaxHeap();
47
48      // HeapInterface Public Methods:
49      bool isEmpty() const;
50      int getNumberOfNodes() const;
51      int getHeight() const;
52      ItemType peekTop() const throw(PrecondViolatedExcept);
53      bool add(const ItemType& newData);
54      bool remove();
55      void clear();
56  }; // end ArrayMaxHeap
57  #include "ArrayMaxHeap.cpp"
58  #endif
```

LISTING 17-2 The header file for the class ArrayMaxHeap

The Implementation

```
template<class ItemType>
int ArrayMaxHeap<ItemType>::getLeftChildIndex(const int nodeIndex) const
{
   return (2 * nodeIndex) + 1;
} // end getLeftChildIndex
```

Definition of method getLeftChildIndex

The Implementation

```
template<class  ItemType>
ArrayMaxHeap<ItemType>::
ArrayMaxHeap(const ItemType someArray[], const int arraySize):
             itemCount(arraySize), maxItems(2 * arraySize)
{
    // Allocate the array
    items = std::make_unique<ItemType[]>(maxItems);

    // Copy given values into the array
    for (int i = 0; i < itemCount; i++)
       items[i] = someArray[i];

    // Reorganize the array into a heap
    heapCreate();
 }  // end constructor
```

Definition of the constructor

The Implementation

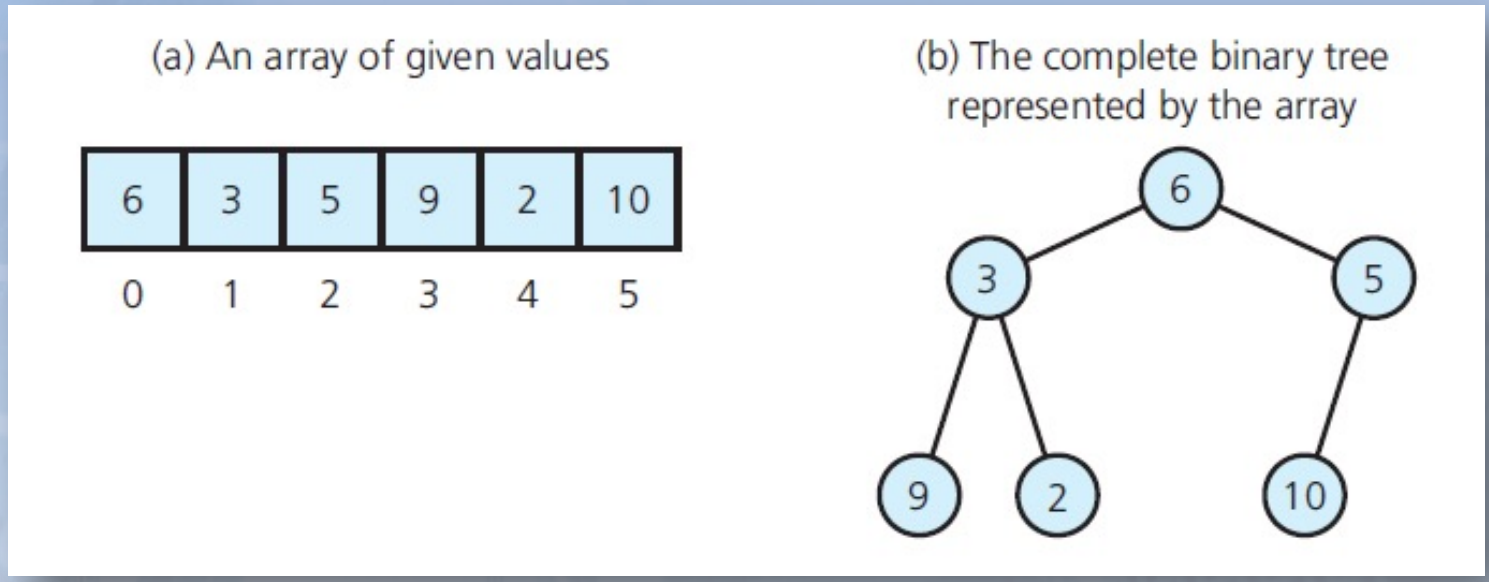

FIGURE 17-8 Array and its corresponding complete binary tree

The Implementation

```
for (index = itemCount - 1 down to 0)
{
   // Assertion: The tree rooted at index is a semiheap
   heapRebuild(index)
   // Assertion: The tree rooted at index is a heap
}
```

Building a heap from an array of data

The Implementation

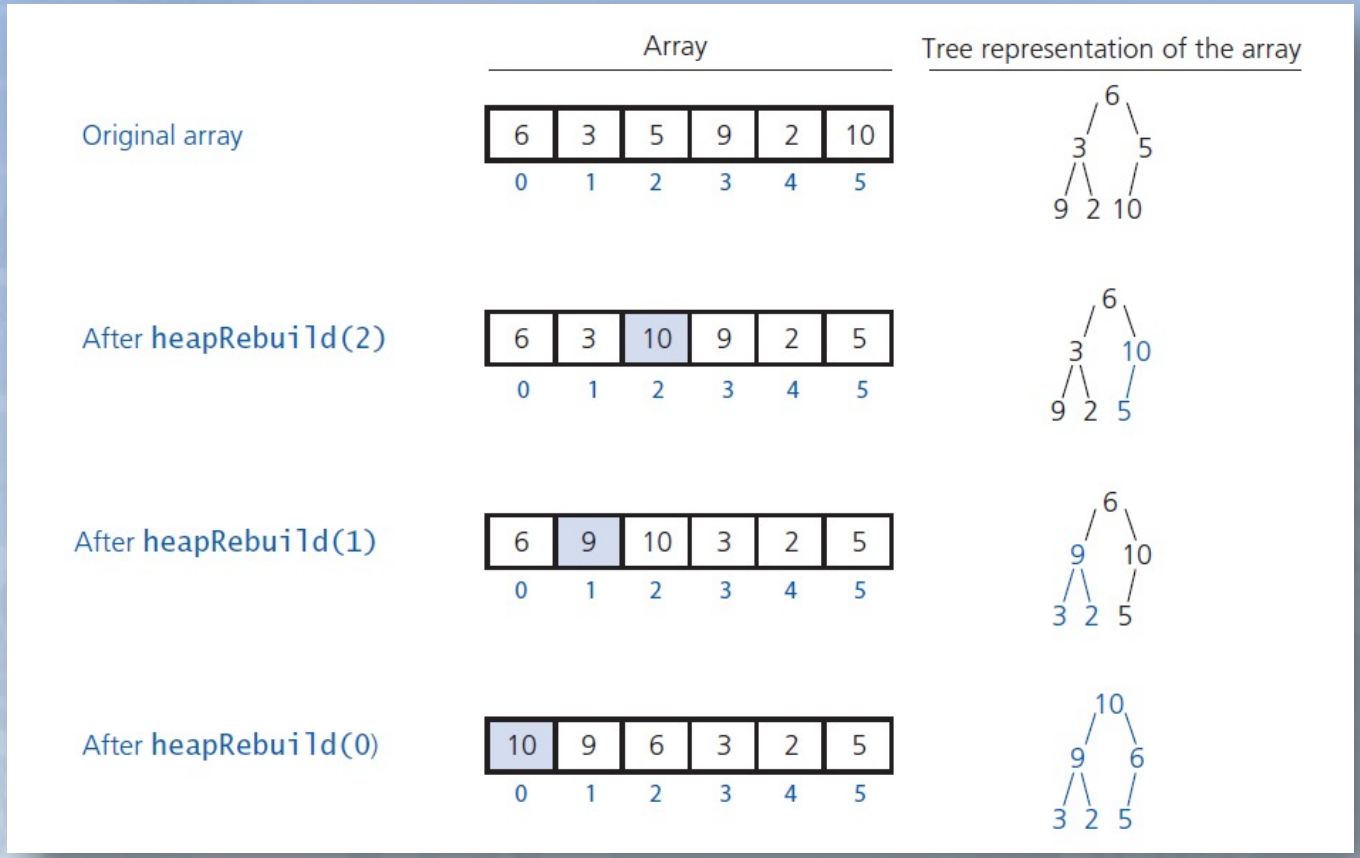

FIGURE 17-9 Transforming an array into a heap

The Implementation

```
template<class ItemType>
void ArrayMaxHeap<ItemType>::heapCreate()
{
    for (int index = itemCount / 2; index >= 0; index--)
       heapRebuild(index);
}  // end heapCreate
```

C++ method heapCreate

The Implementation

```
template<class ItemType>
ItemType ArrayMaxHeap<ItemType>::peekTop() const throw(PrecondViolatedExcept)
{
   if (isEmpty())
      throw PrecondViolatedExcep("Attempted peek into an empty heap.");

   return items[0];
}  // end peekTop
```

C++ method peekTop which tests for an empty heap

Heap Implementation of the ADT Priority Queue

```
1  /** ADT priority queue: Heap-based implementation.
2   @file HeapPriorityQueue.h */
3  #ifndef HEAP_PRIORITY_QUEUE_
4  #define HEAP_PRIORITY_QUEUE_
5  #include "ArrayMaxHeap.h"
6  #include "PriorityQueueInterface.h"
7
8  template<class ItemType>
9  class HeapPriorityQueue : public PriorityQueueInterface<ItemType>,
10                            private ArrayMaxHeap<ItemType>
11 {
12 public:
13    HeapPriorityQueue();
14    bool isEmpty() const;
15    bool enqueue(const ItemType& newEntry);
16    bool dequeue();
17
18    /** @pre  The priority queue is not empty. */
19    ItemType peekFront() const throw(PrecondViolatedExcept);
20 }; // end HeapPriorityQueue
21
22 #include "HeapPriorityQueue.cpp"
23 #endif
```

LISTING 17-3 A header file for the class HeapPriorityQueue

Heap Implementation of the ADT Priority Queue

```
1  /** Heap-based implementation of the ADT priority queue.
2   @file HeapPriorityQueue.cpp */
3
4  #include "HeapPriorityQueue.h"
5
6  template<class ItemType>
7  HeapPriorityQueue<ItemType>::HeapPriorityQueue()
8  {
9     ArrayMaxHeap<ItemType>();
10 }  // end constructor
11
12 template<class ItemType>
13 bool HeapPriorityQueue<ItemType>::isEmpty() const
14 {
15    return ArrayMaxHeap<ItemType>::isEmpty();
16 }  // end isEmpty
17
18 template<class ItemType>
19 bool HeapPriorityQueue<ItemType>::enqueue(const ItemType& newEntry)
20 {
21    return ArrayMaxHeap<ItemType>::add(newEntry);
22 }  // end add
```

LISTING 17-4 An implementation of the class HeapPriorityQueue

Heap Implementation of the ADT Priority Queue

```
23
24  template<class ItemType>
25  bool HeapPriorityQueue<ItemType>::dequeue()
26  {
27     return ArrayMaxHeap<ItemType>::remove();
28  }  // end remove
29
30  template<class ItemType>
31  ItemType HeapPriorityQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
32  {
33     try
34     {
35        return ArrayMaxHeap<ItemType>::peekTop();
36     }
37     catch (PrecondViolatedExcept e)
38     {
39        throw PrecondViolatedExcept("Attempted peek into an empty priority queue.");
40     }  // end try/catch
41  }  // end peekFront
```

LISTING 17-4 An implementation of the class HeapPriorityQueue

Heap Implementation of the ADT Priority Queue

- Heap versus a binary search tree
  - If you know maximum number of items in the priority queue, heap is the better implementation.
- Finite, distinct priority values
  - Many items likely have same priority value
  - Place in same order as encountered

Heap Sort

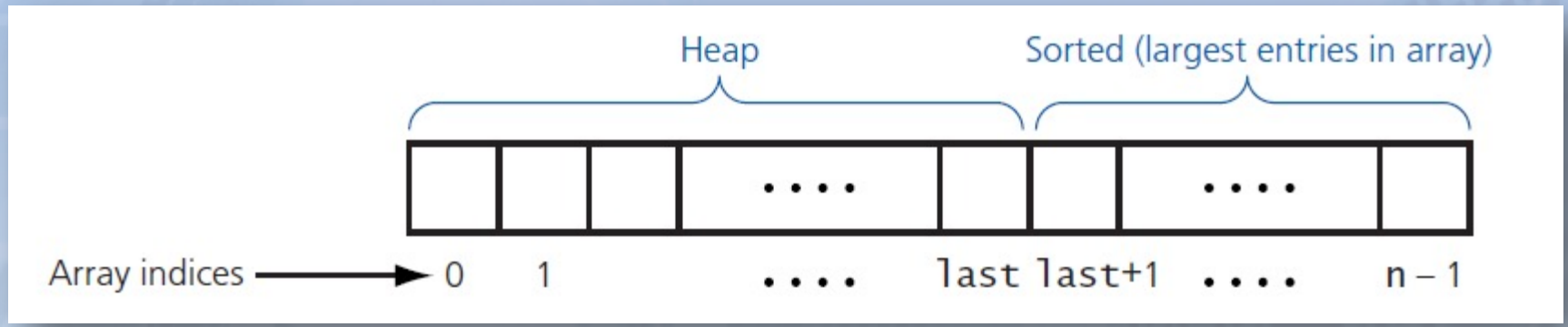

FIGURE 17-10 Heap sort partitions an array into two regions

Heap Sort

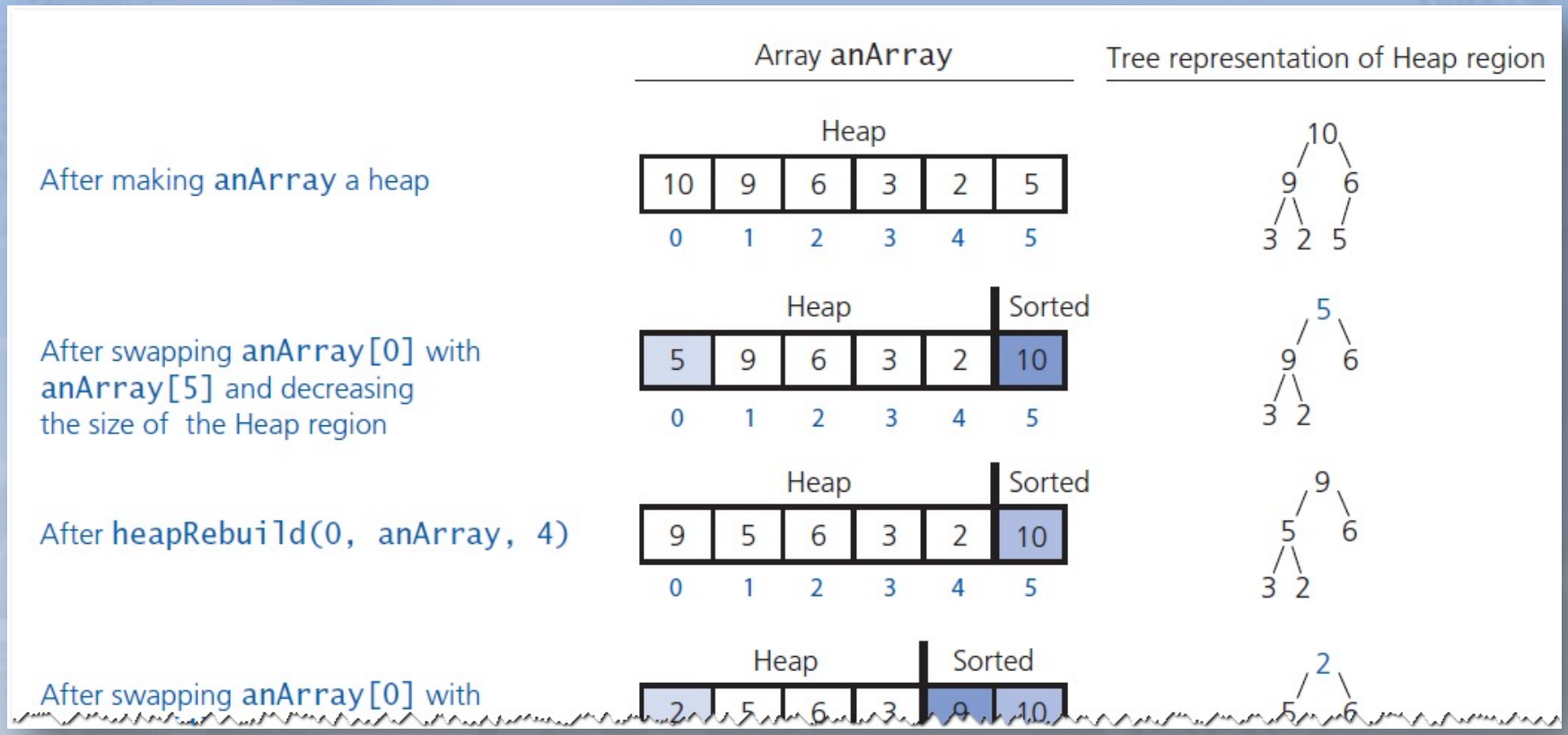

FIGURE 17-11 A trace of heap sort, beginning with the heap in Figure 17-9

Heap Sort

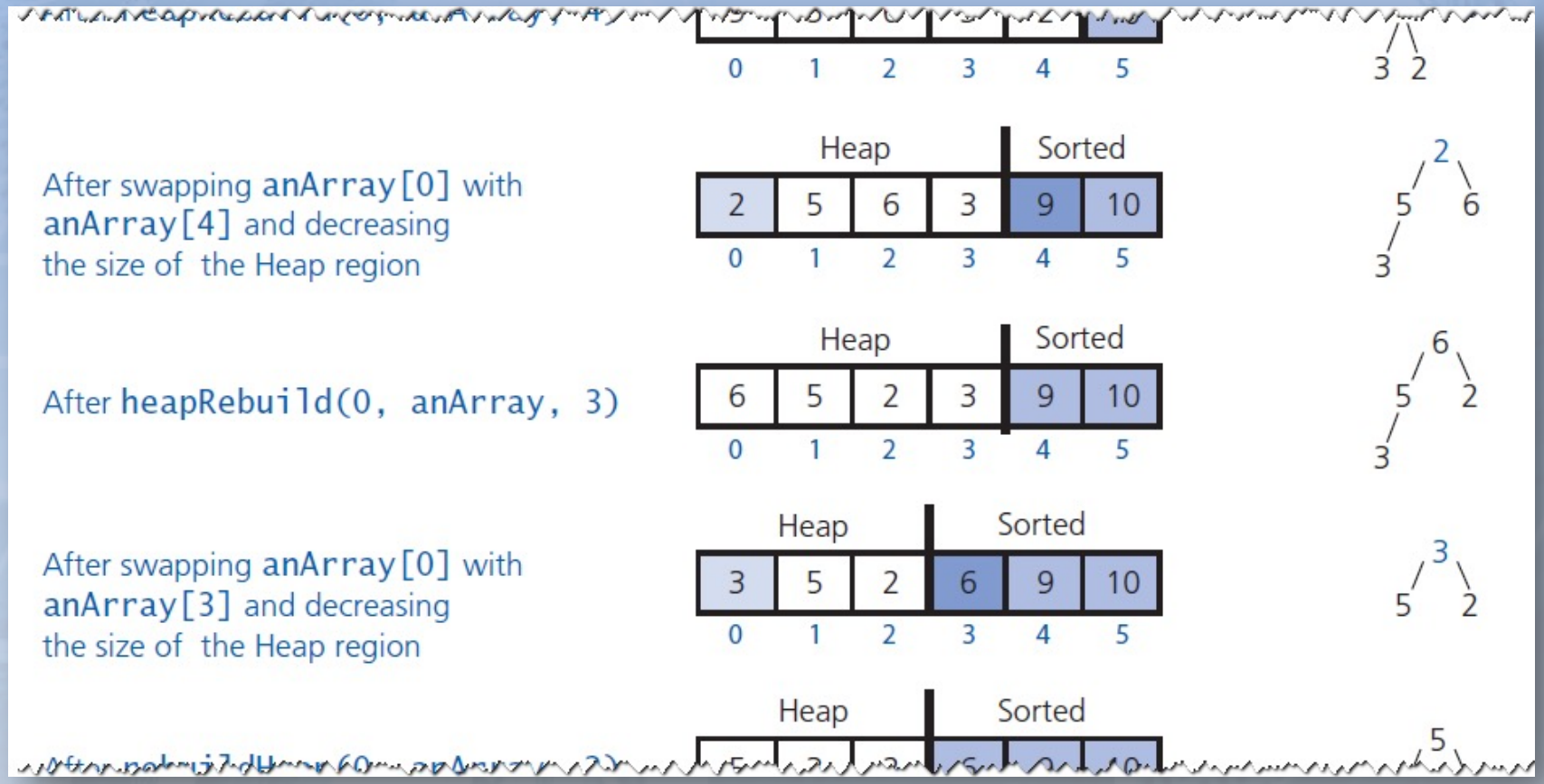

FIGURE 17-11 A trace of heap sort, beginning with the heap in Figure 17-9

Heap Sort

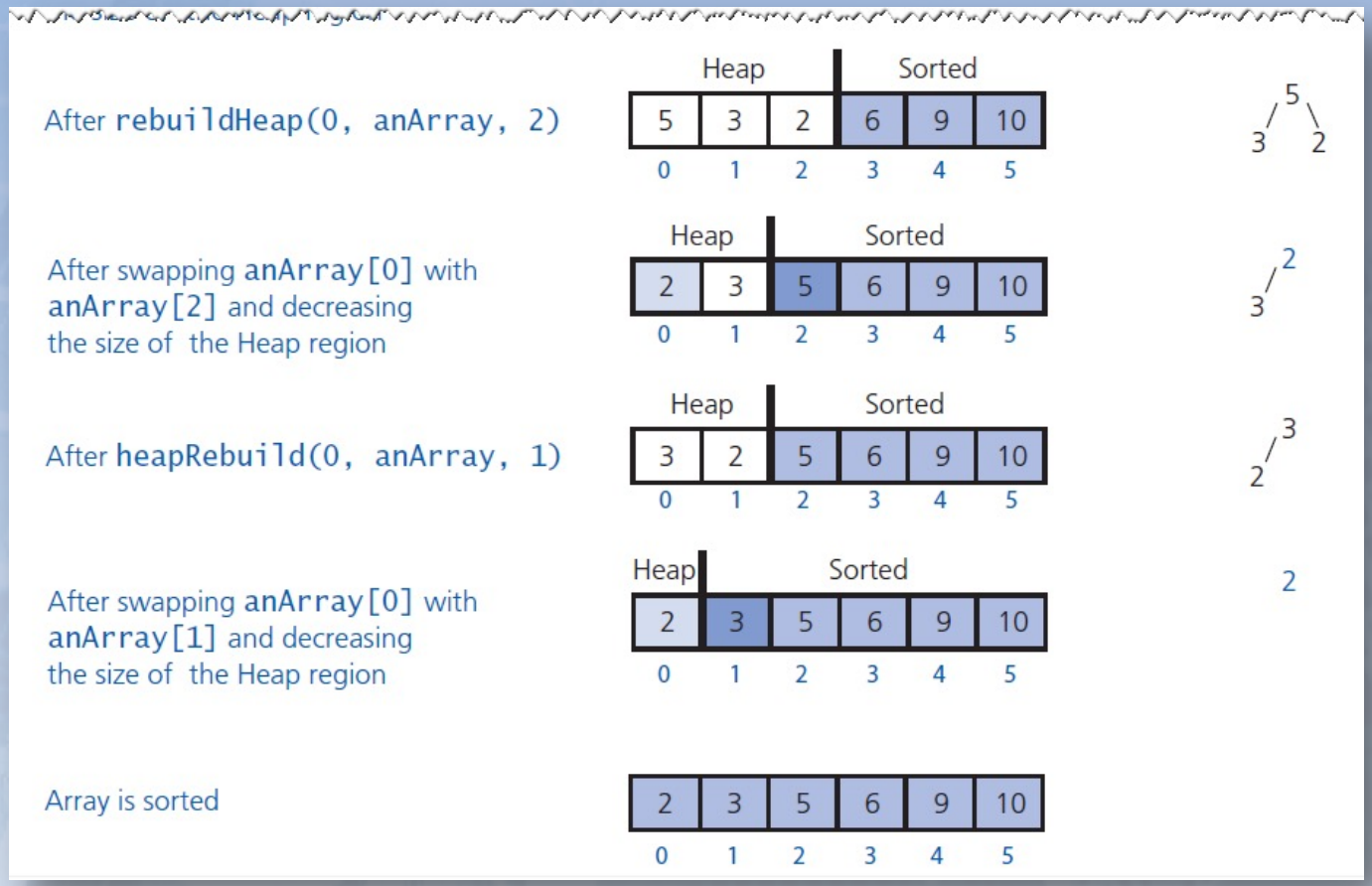

FIGURE 17-11 A trace of heap sort, beginning with the heap in Figure 17-9

End

Chapter 17