An Introduction to Sorting

Chapter 8

Contents

- Organizing Java Methods That Sort an Array
- Selection Sort
  - Iterative Selection Sort
  - Recursive Selection Sort
  - The Efficiency of Selection Sort

Contents

- Insertion Sort
  - Iterative Insertion Sort
  - Recursive Insertion Sort
  - The Efficiency of Insertion Sort
  - Insertion Sort of a Chain of Linked Nodes
- Shell Sort
  - The Java Code
  - The Efficiency of Shell Sort
- Comparing the Algorithms

Objectives

- Sort array into ascending order using
  - Selection sort
  - Insertion sort
  - Shell sort
- Sort a chain of linked nodes into ascending order using insertion sort
- Assess efficiency of a sort, discuss relative efficiencies of various methods

Sorting

- Arranging things into either ascending or descending order is called “sorting”
- This chapter discusses, implements simple algorithms that sort items into ascending order
  - Sort into descending order with a few changes
- In Java, possible to create class of static methods which sort objects of an array

Sorting an Array

- For an array to be sortable, objects must be comparable
  - Must implement interface **`Comparable`**

```
<T extends Comparable<T>>
```

- We could begin our class with

```
public class SortArray
{
   public static <T extends Comparable<T>> void sort(T[] a, int n)
   {  . . .
```

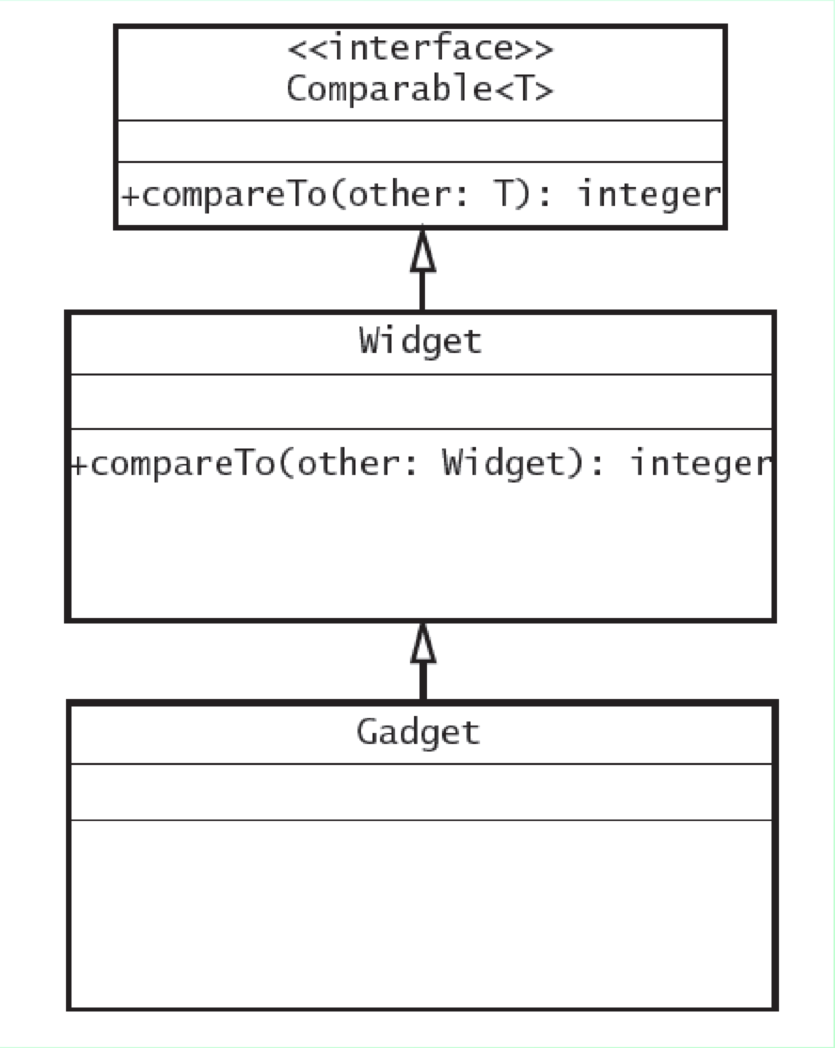


Figure 8-1 The class **Gadget** is derived from the class **Widget**, which implements the interface **Comparable**

Selection Sort

- Example of sorting books by height
  - Take all books off shelf
  - Select shortest , replace on shelf
  - Continue until all books
- Alternative
  - Look down shelf, select shortest
  - Swap first with selected shortest
  - Move to second slot, repeat process

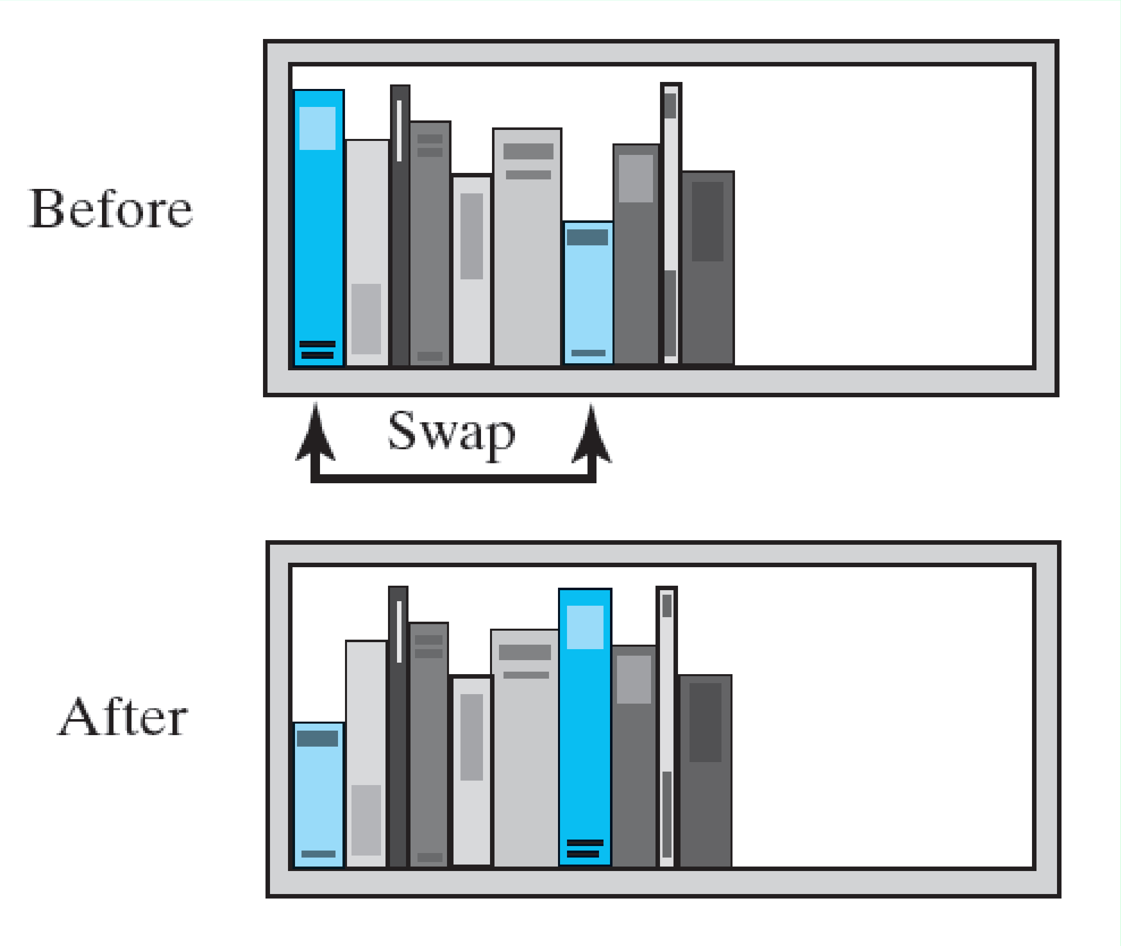


Figure 8-2 Before and after exchanging the shortest book and the first book

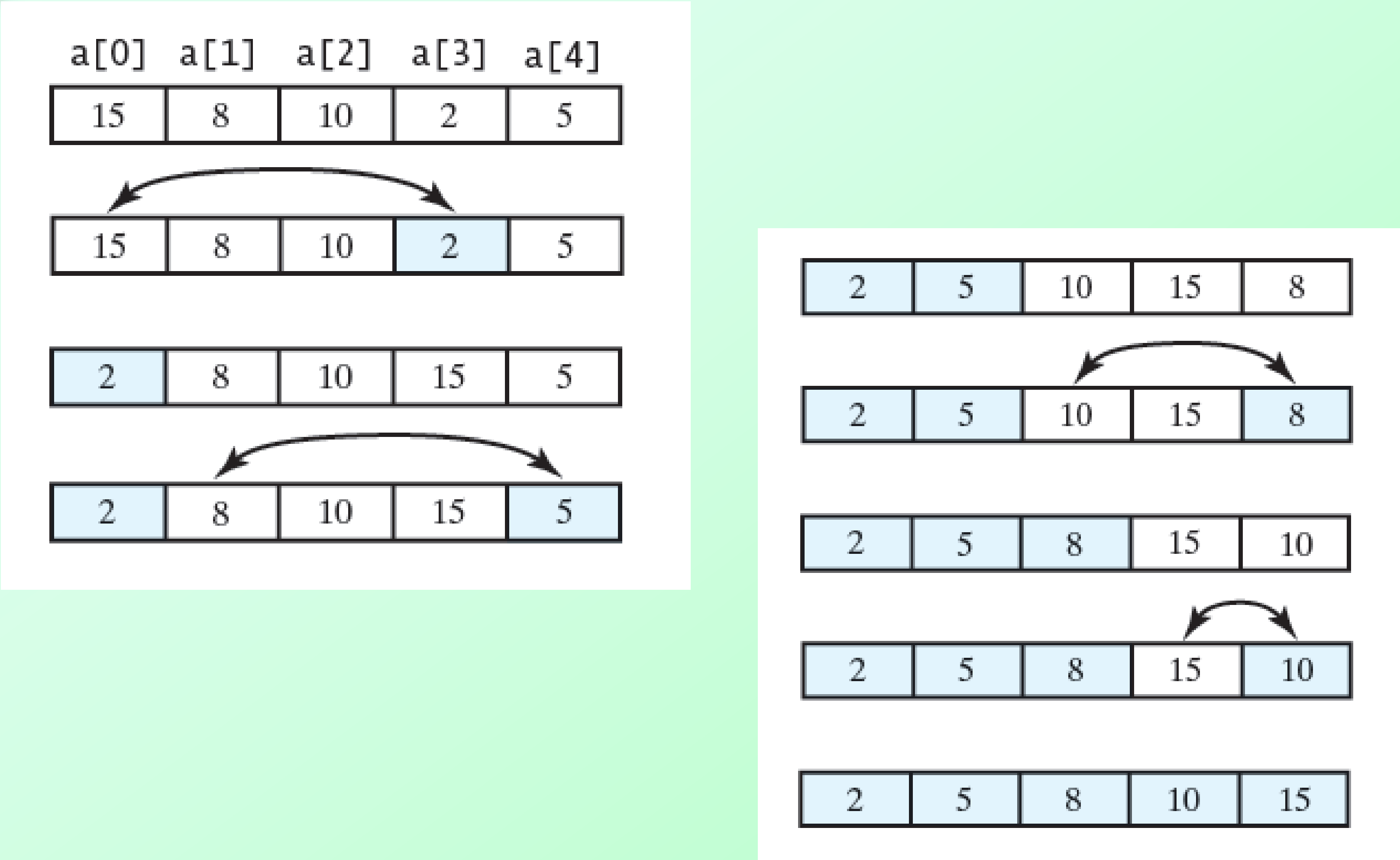


Figure 8-3 A selection sort of an array of integers into ascending order

Selection Sort

- Pseudocode for algorithm

```
Algorithm selectionSort(a, n)
// Sorts the first n entries of an array a.

for (index = 0; index < n - 1; index++)
{
   indexOfNextSmallest = the index of the smallest value among
                         a[index], a[index + 1], . . . , a[n - 1]
   Interchange the values of a[index] and a[indexOfNextSmallest]
   // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[index], and these are the smallest
   // of the original array entries. The remaining array entries begin at a[index + 1].
}
```

- View source code, [Listing 8-1]
- Efficiency of s (n^2)

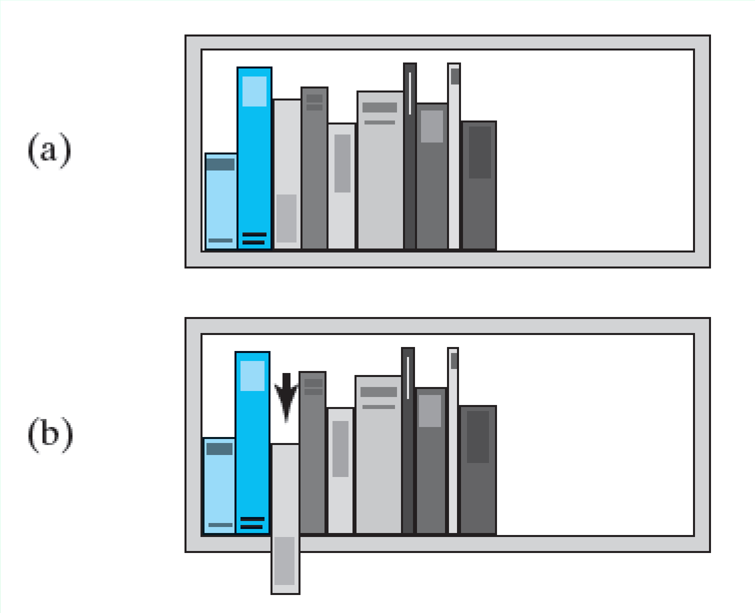
Note: Code listing files must be in same folder as PowerPoint files for links to work

Question 1 Trace the steps that a selection sort takes when sorting the following array into ascending order: 9 6 2 4 8.

1.	9	6	2	4	8
	2	6	9	4	8
	2	4	9	6	8
	2	4	6	9	8
	2	4	6	8	9

Insertion Sort

- When book found taller than one to the right
  - Remove book to right
  - Slide taller book to right
  - Insert shorter book into that spot
- Compare shorter book just moved to left
  - Make exchange if needed
- Continue …



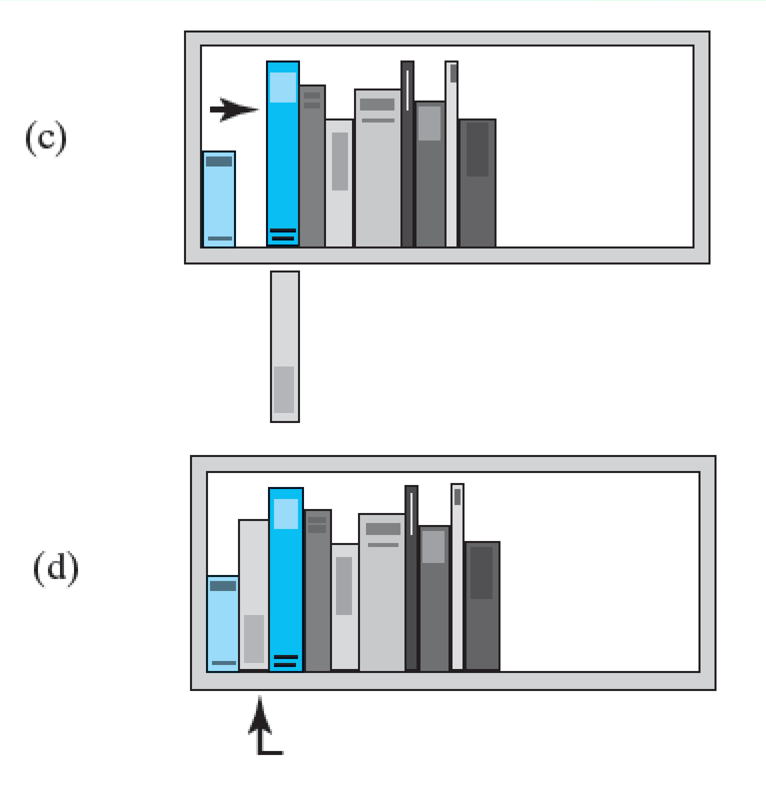


Figure 8-4 The placement of the third book during an insertion sort

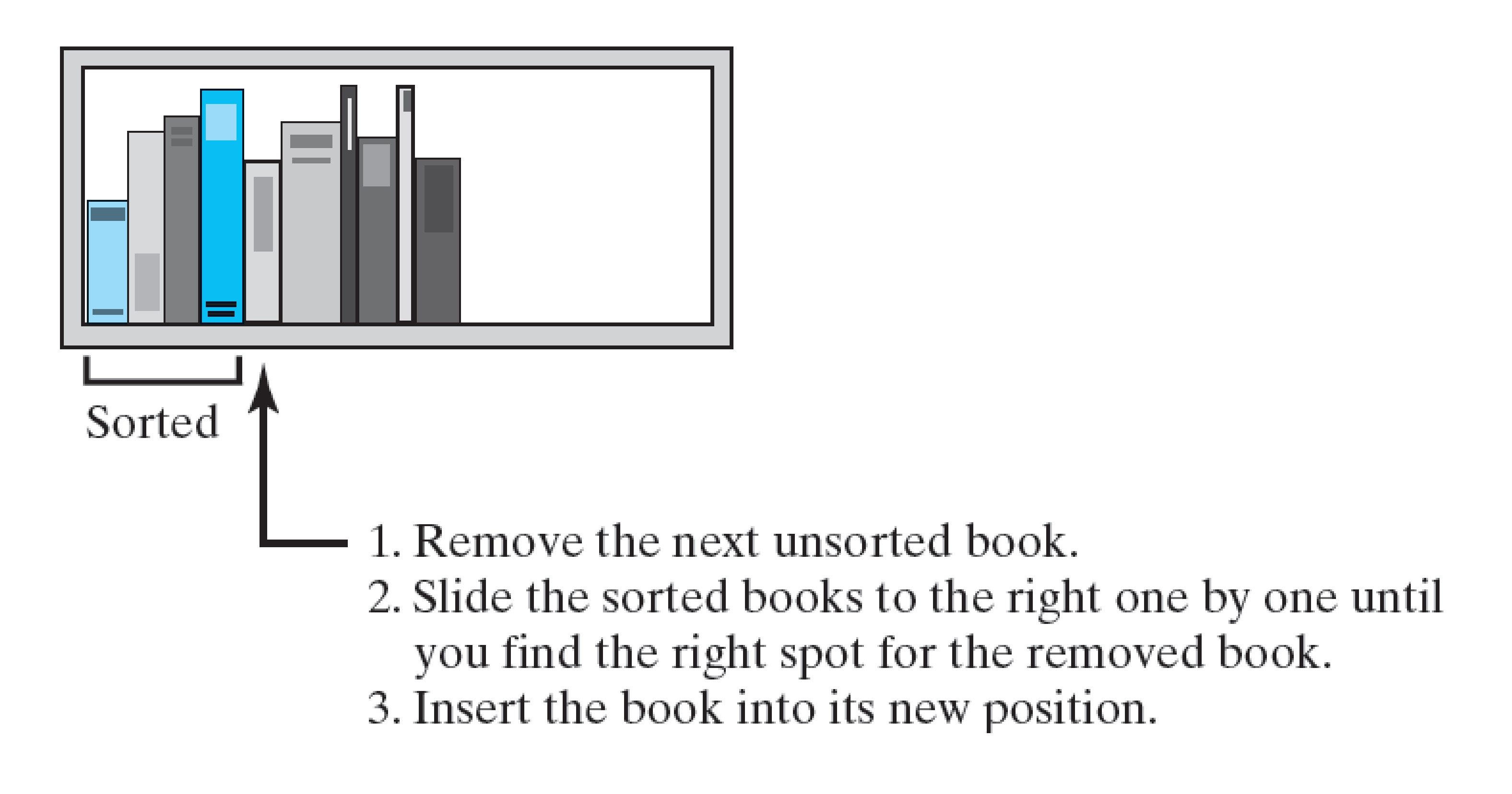


Figure 8-5 An insertion sort of books

Insertion Sort

- Algorithms

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.

for (unsorted = first + 1 through last)
{
   nextToInsert = a[unsorted]
   insertInOrder(nextToInsert, a, first, unsorted - 1)
}
```

Insertion Sort

- Algorithms

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted entries a[begin] through a[end].

index = end                    // index of last entry in the sorted portion
// make room, if needed, in sorted portion for another entry
while ( (index >= begin) and (anEntry < a[index]) )
{
   a[index + 1] = a[index] // make room
   index--
}
// Assertion: a[index + 1] is available.

a[index + 1] = anEntry         // insert
```

Question 2 Trace the steps that an insertion sort takes when sorting the following array into ascending order: 9 6 2 4 8.

2.

9	6	2	4	8
6	9	2	4	8
2	6	9	4	8
2	4	6	9	8
2	4	6	8	9

Insertion Sort

- Recursive algorithm

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (the array contains more than one entry)
{
    Sort the array entries a[first] through a[last - 1]
    Insert the last entry a[last] into its correct sorted position within the rest of the array
}
```

Insertion Sort

- Recursive Java method

```
public static <T extends Comparable<? super T>>
       void insertionSort(T[] a, int first, int last)
{
   if (first < last)
   {
      // sort all but the last entry
      insertionSort(a, first, last - 1);

      // insert the last entry in sorted order
      insertInOrder(a[last], a, first, last - 1);
   } // end if
} // end insertionSort
```

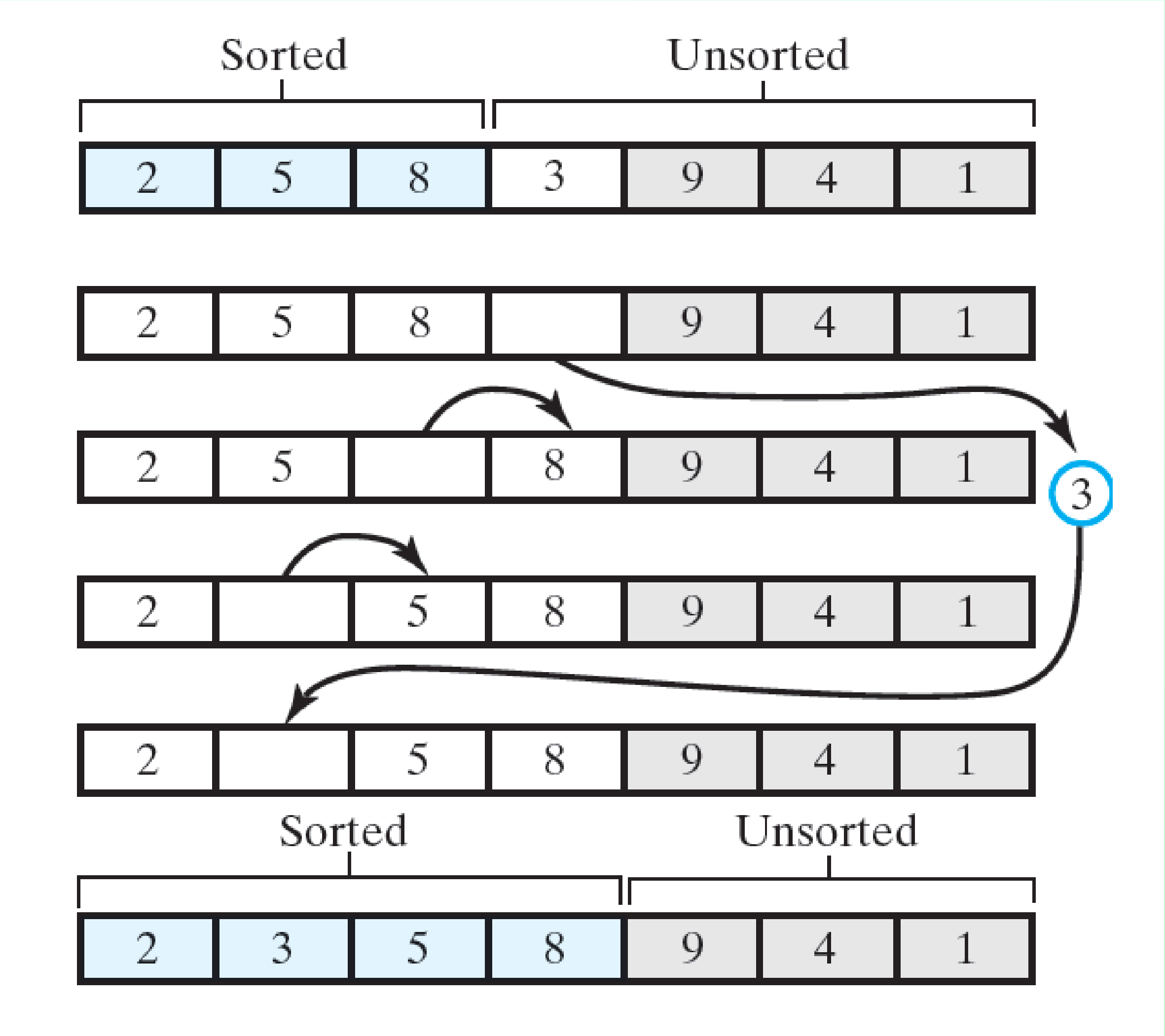


Figure 8-6 Inserting the next unsorted entry into its proper location within the sorted portion of an array during an insertion sort

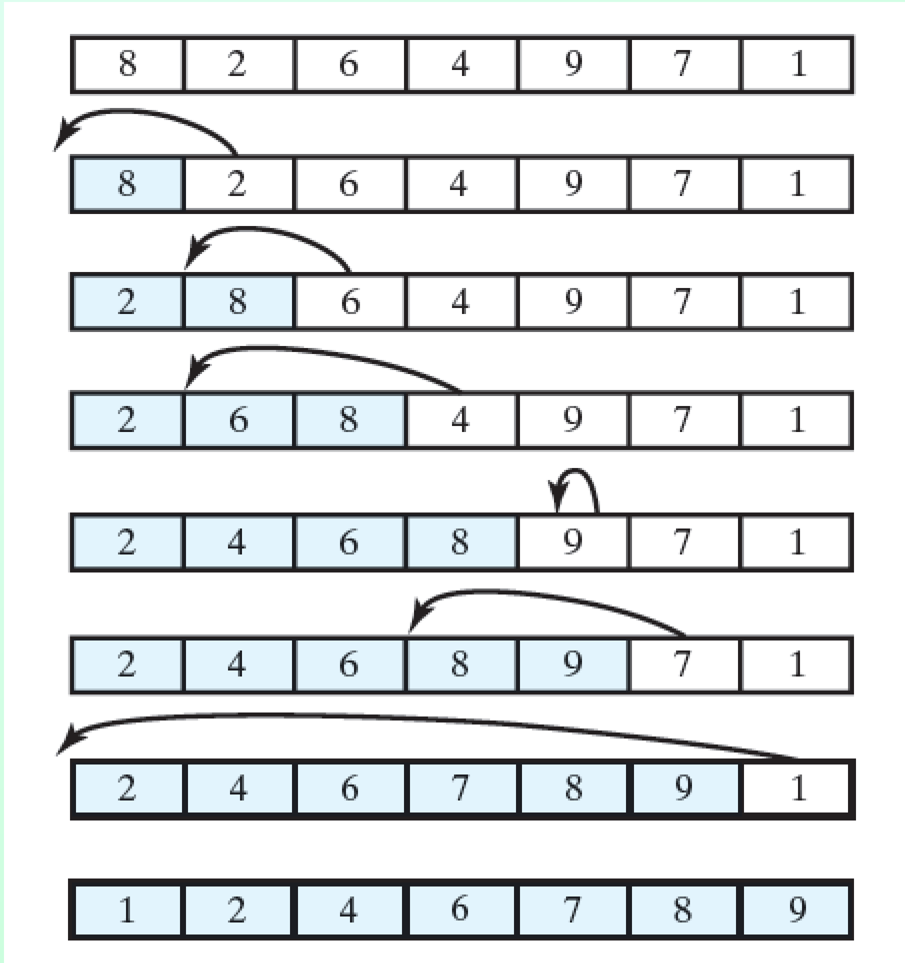


Figure 8-7 An insertion sort of an array of integers into ascending order

Insertion Sort

- The algorithm **`insertInOrder`**: first draft.

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// First draft.

if (anEntry >= a[end])
   a[end + 1] = anEntry
else
{
   a[end + 1] = a[end]
   insertInOrder(anEntry, a, begin, end - 1)
}
```

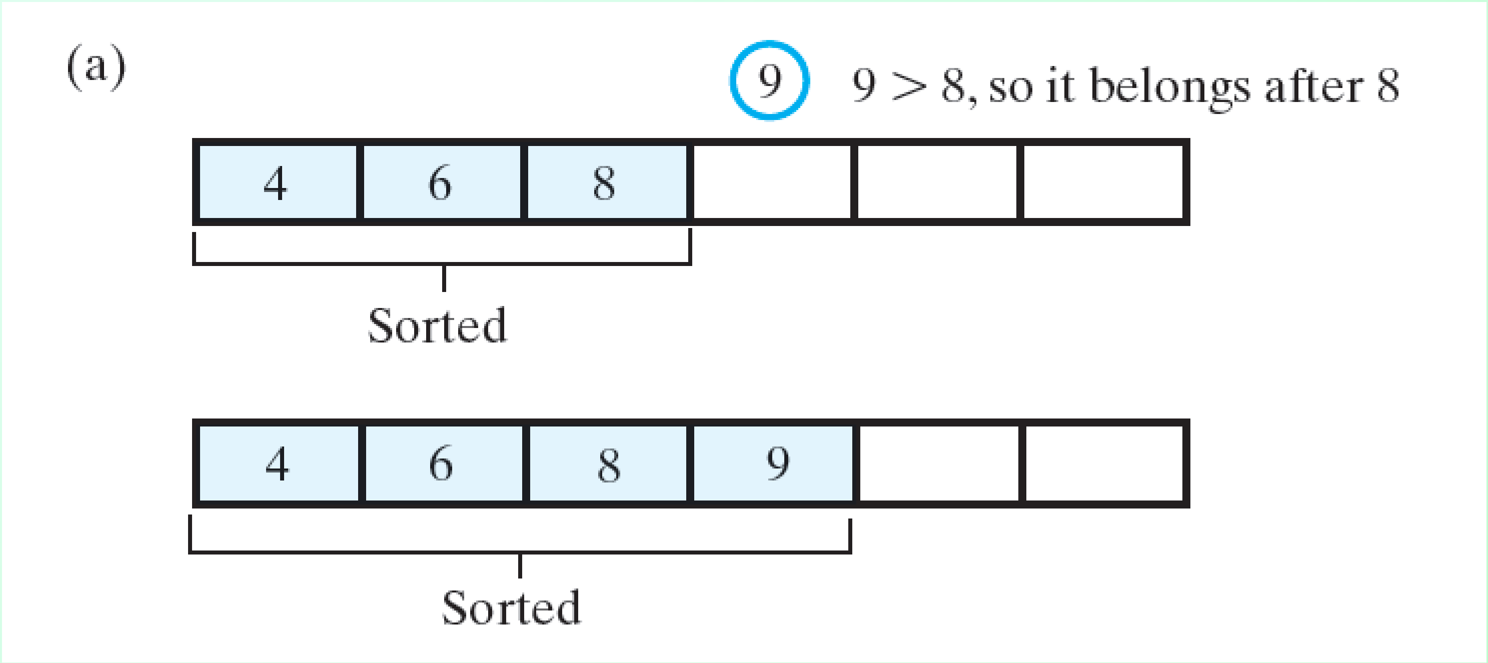


Figure 8-8 Inserting the first unsorted entry into the sorted portion of the array. (a) The entry is greater than or equal to the last sorted entry;

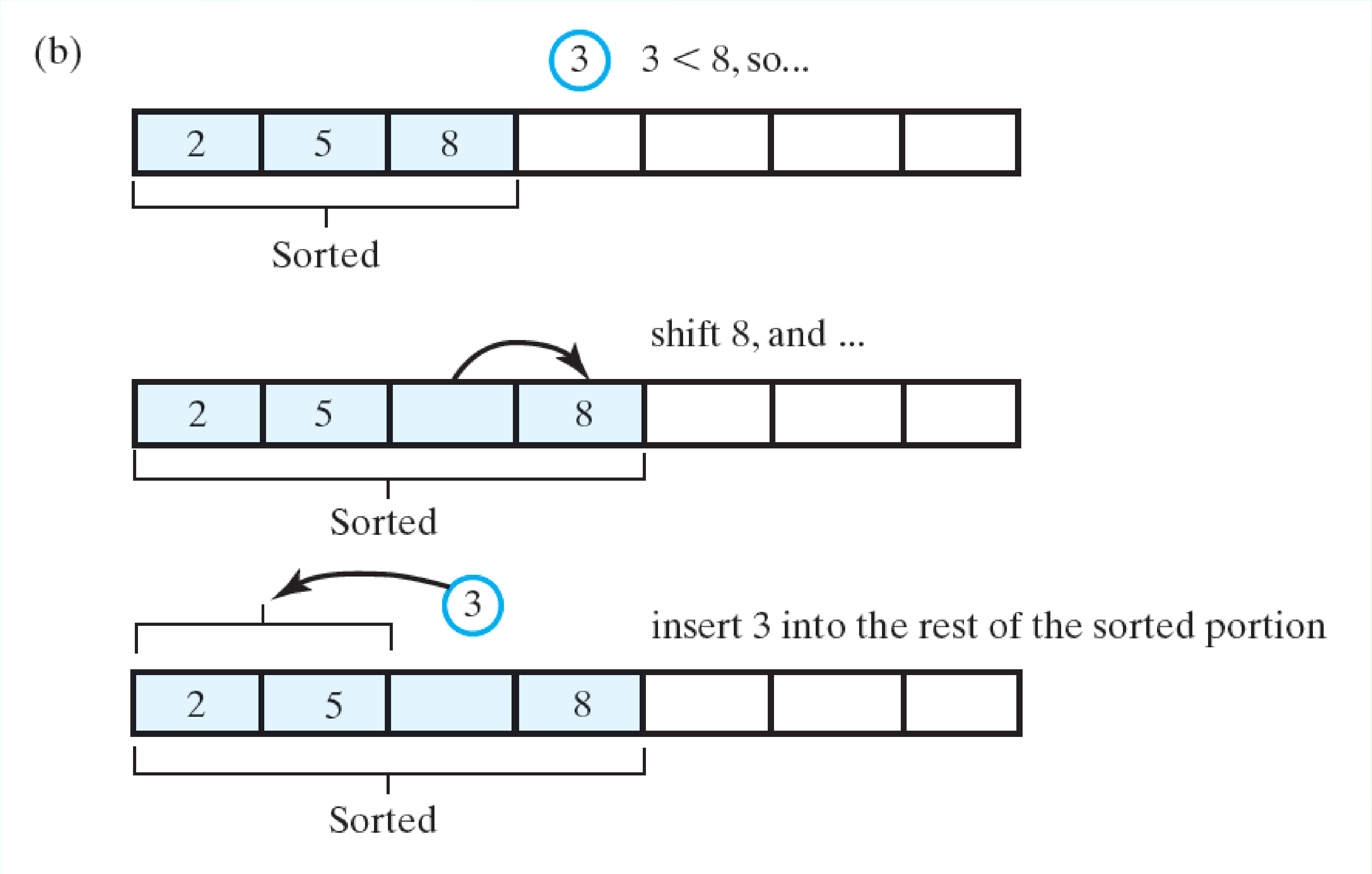


Figure 8-8 Inserting the first unsorted entry into the sorted portion of the array. (b) the entry is smaller than the last sorted entry

Insertion Sort

- The algorithm **`insertInOrder`**: final draft.

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

if (anEntry >= a[end])
   a[end + 1] = anEntry
else if (begin < end)
{
   a[end + 1] = a[end]
   insertInOrder(anEntry, a, begin, end - 1)
}
else // begin == end and anEntry < a[end]
{
   a[end + 1] = a[end]
   a[end] = anEntry
}
```

Insertion Sort

- Efficiency
  - Loop executes at most
    1 + 2 + … (n – 1) times
  - Sum is $$\frac{n \cdot (n-1)}{2}$$
  - Which gives $O(n^2)$
- Best case – array already in order, O(n)

Insertion Sort of a Chain of Linked Nodes

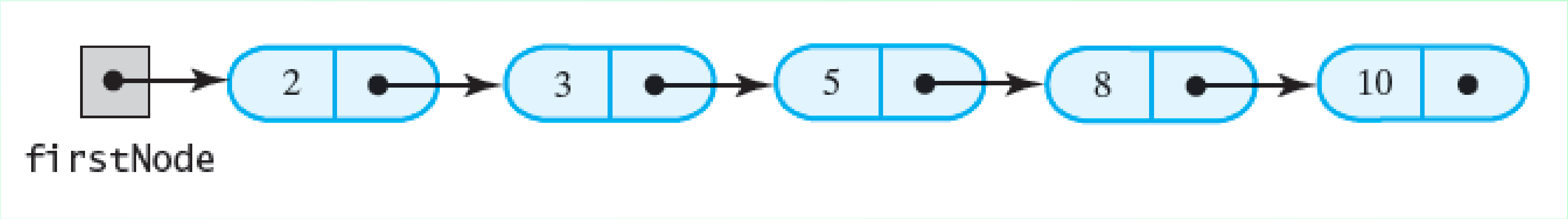


Figure 8-9 A chain of integers sorted into ascending order

Insertion Sort of a Chain of Linked Nodes

- Consider inserting node in chain in correct position
- First locate where it should go
  - Make comparisons from head towards end of chain
  - During chain traversal, keep reference to previous node of comparison

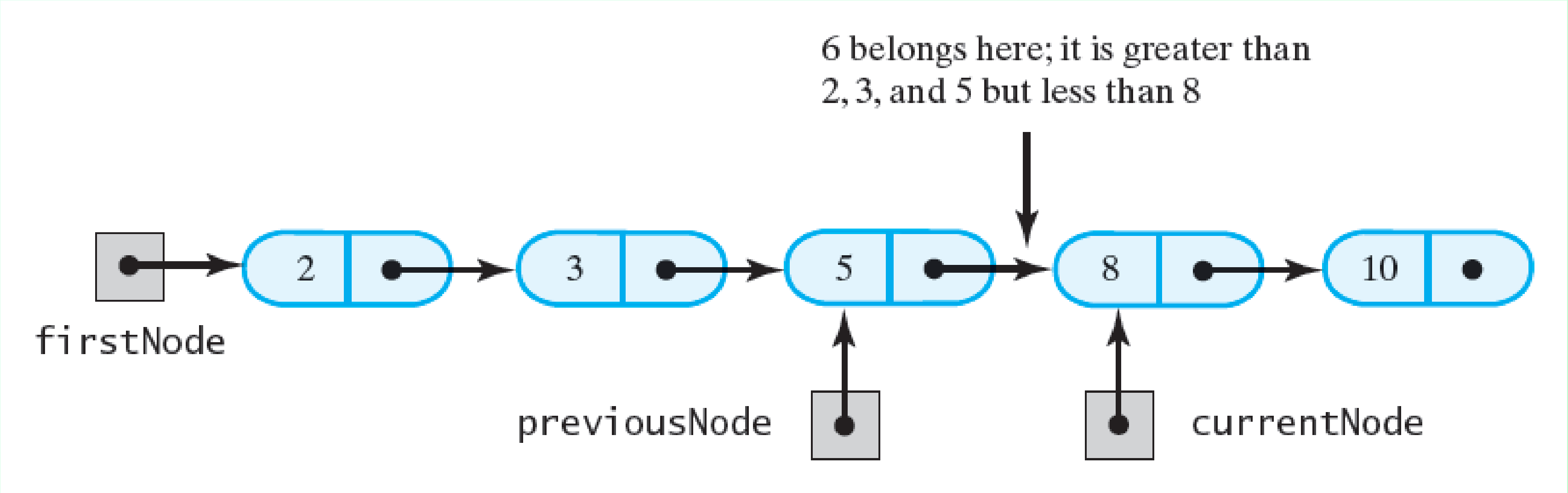


Figure 8-10 During the traversal of a chain to locate the insertion point, save a reference to the node before the current one

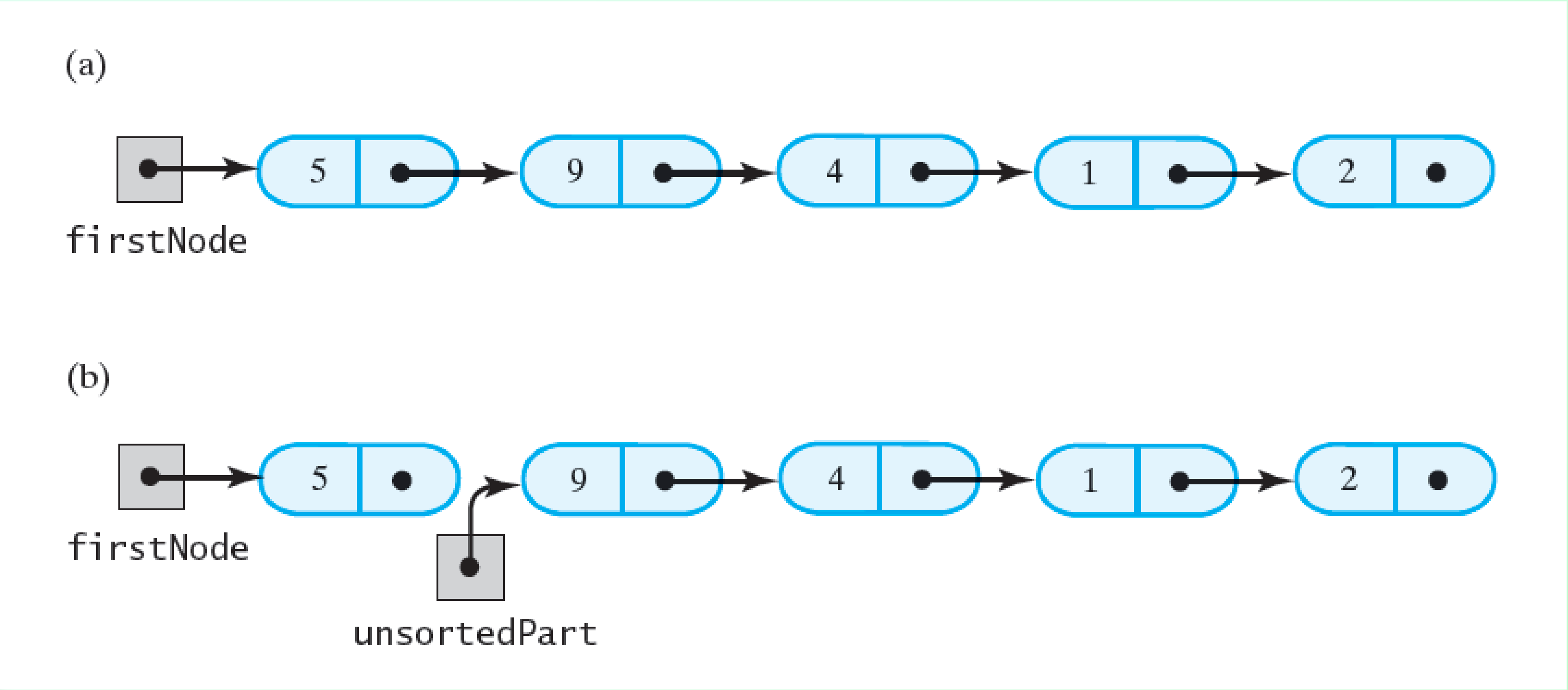


Figure 8-11 Breaking a chain of nodes into two pieces as the first step in an insertion sort: (a) the original chain; (b) the two pieces

Insertion Sort of a Chain of Linked Nodes

- Consider a class which holds a collection with linked list

```
public class LinkedGroup<T extends Comparable<? super T>>
        implements GroupInterface<T>
{
    private Node firstNode;
    int length; // number of objects in the group
    . . .
```

- We will add a sort method to this class
- Note source code listings, Listing 8-A, 8-B

Insertion Sort of a Chain of Linked Nodes

- Efficiency
  - As before, loop executes at most
    1 + 2 + … (n – 1) times
  - Results in efficiency of $O(n^2)$
- Insertion sort is reasonable way to sort chain of linked nodes

Question 3 In the previous method `insertionSort`, if you move the line

```
unsortedPart = unsortedPart.getNextNode();
```

after the call to `insertInOrder`, will the method still work? Explain.

Question 4 The previous method `insertionSort` is not a static method. Why?

3. No; `insertInOrder` links the node to be inserted into the sorted part of the chain so that the node no longer references the rest of the unsorted part. Since `unsortedPart` still references the inserted node, executing the line in question next would make `unsortedPart` either reference a node in the sorted part or be `null`.

4. The public method `insertionSort` is to be invoked by using an object of `LinkedGroup`, which is the class that defines this method. Thus, the method should not be static.

Shell Sort

- Previously mentioned sorts are simple, often useful
  - However can be inefficient for large arrays
  - Array entries move only to adjacent locations
- Shell sort moves entries beyond adjacent locations
  - Sort sub arrays of entries at equally spaced indices

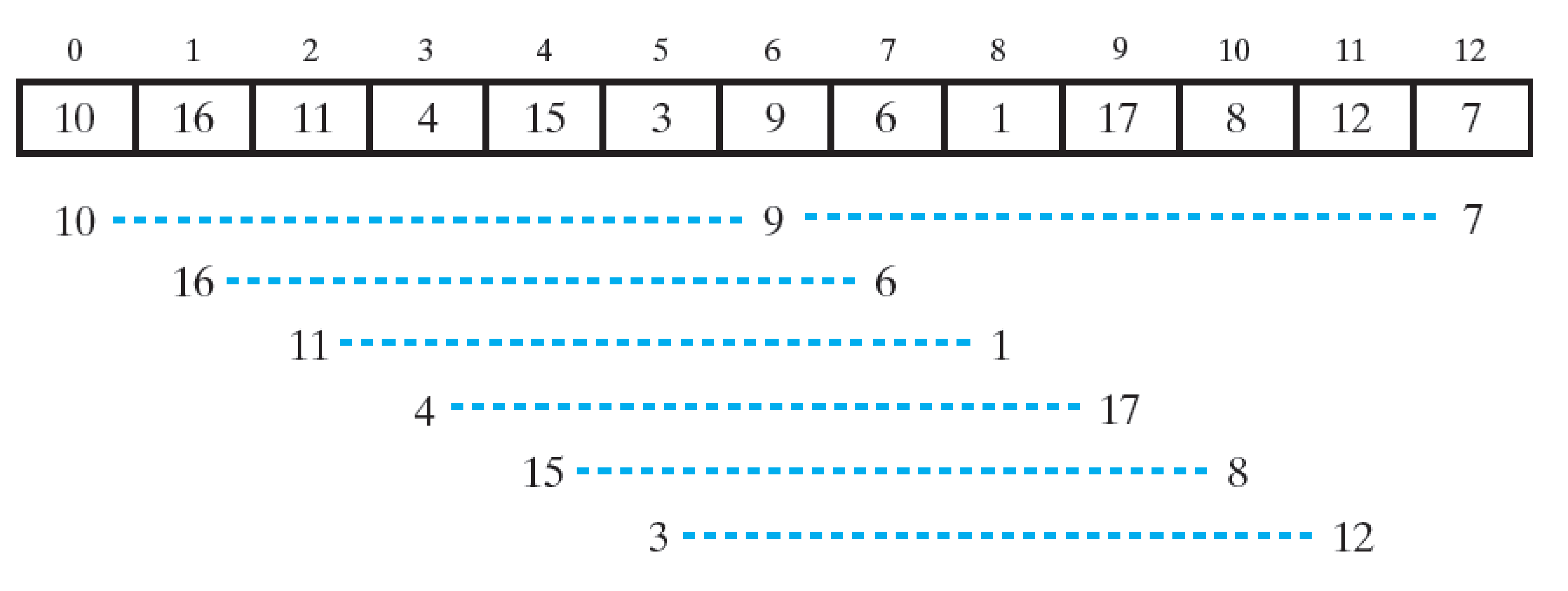


Figure 8-12 An array and the subarrays formed by grouping entries whose indices are 6 apart

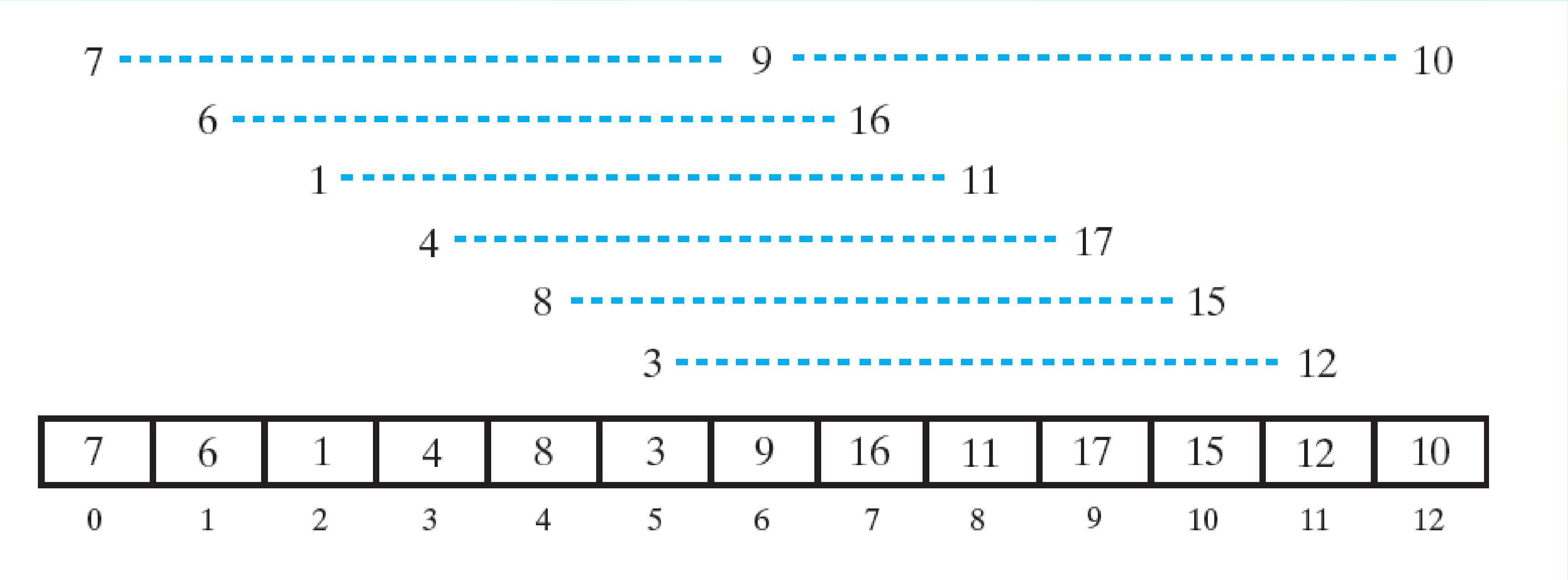


Figure 8-13 The subarrays of Figure 8-12 after each is sorted, and the array that contains them

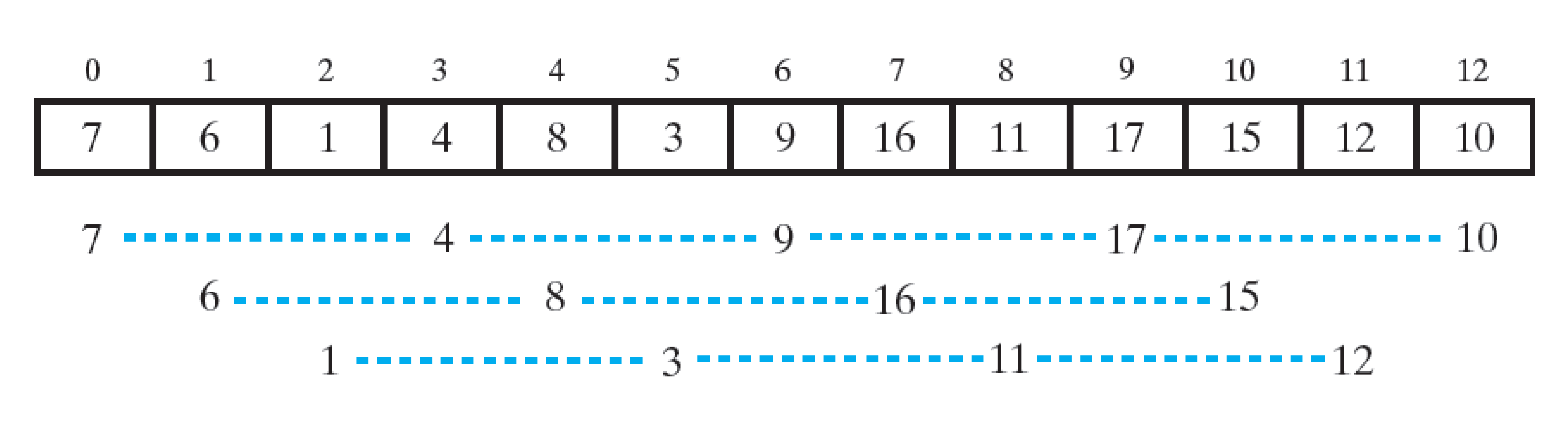


Figure 8-14 The subarrays of the array in Figure 8-13 formed by grouping entries whose indices are 3 apart

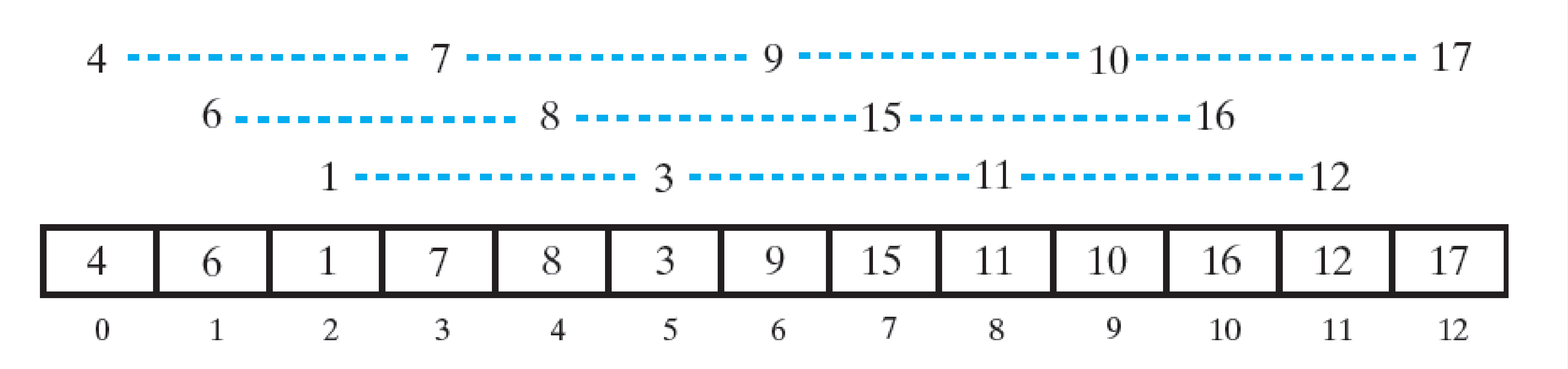


Figure 8-15 The subarrays of Figure 8-14 after each is sorted, and the array that contains them

Question 5 Apply the Shell sort to the array 9 8 2 7 5 4 6 3 1, with index separations of 4, 2, and 1. What are the intermediate steps?

5. First, you consider the subarray of equally spaced integers at the indices 0, 4, and 8 (they appear in bold):

9 8 2 7 **5** 4 6 3 **1**

Now sort them to get

1 8 2 7 **5** 4 6 3 **9**

The indices 0, 4, and 8 have a separation of 4. Next, consider the integers at indices 1 and 5:

1 **8** 2 7 5 **4** 6 3 9

Sort them to get

1 **4** 2 7 5 **8** 6 3 9

Then sort the integers at indices 2 and 6; they already are in order:

1 4 **2** 7 5 8 **6** 3 9

Next, consider the integers at indices 3 and 7. Sort them to get

1 4 2 **3** 5 8 6 7 9

Now decrease the separation between indices to 2. You consider the integers at the indices 0, 2, 4, 6, and 8:

1 4 **2** 3 **5** 8 **6** 7 **9**

You find that they are sorted. Then consider the integers at indices 1, 3, 5, and 7:

1 **4** 2 **3** 5 **8** 6 7 9

Sort them to get

1 **3** 2 **4** 5 7 6 **8** 9

Decreasing the separation to 1 results in an ordinary insertion sort of an array that is almost sorted.

Java Code

- Incremental Insertion Sort Listing 8-C
- Method which calls the **IncrementalInsertionSort**

```
public static <T extends Comparable<? super T>>
       void shellSort(T[] a, int first, int last)
{
   int n = last - first + 1; // number of array entries

   for (int space = n / 2; space > 0; space = space / 2)
   {
      for (int begin = first; begin < first + space; begin++)
         incrementalInsertionSort(a, begin, last, space);
   } // end for
} // end shellSort
```

Question 6 Trace the steps that a Shell sort takes when sorting the following array into ascending order: 9 6 2 4 8 7 5 3.

6.

9	6	2	4	**8**	7	5	3
8	6	2	4	**9**	7	5	3
8	**6**	2	4	9	**7**	5	3
8	6	**2**	4	9	7	**5**	3
8	6	2	**4**	9	7	5	**3**
8	6	2	**3**	9	7	5	**4**
8	6	**2**	3	**9**	7	**5**	4
2	6	**5**	3	**8**	7	**9**	4
2	**6**	5	**3**	8	**7**	9	**4**
2	**3**	5	**4**	8	**6**	9	**7**

Now apply a regular insertion sort.

	Best Case	Average Case	Worst Case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^2)$ or $O(n^{1.5})$

Figure 8-16 The time efficiencies of three sorting algorithms, expressed in Big Oh notation

End

Chapter 8