Data Structures and Abstractions with Java™

FOURTH EDITION

Frank M. Carrano • Timothy M. Henry

ALWAYS LEARNING

PEARSON

ONLINE ACCESS

Thank you for purchasing a new copy of ***Data Structures and Abstractions with Java^{TM},* Fourth Edition, Global Edition.** Your textbook includes twelve months of prepaid access to the book's Premium Content. This prepaid subscription provides you with full access to the following student support areas:

- VideoNotes are step-by-step video tutorials specifically designed to enhance the programming concepts presented in this textbook
- Premium Web Chapters

Use a coin to scratch off the coating and reveal your student access code.
Do not use a knife or other sharp object as it may damage the code.

To access the ***Data Structures and Abstractions with Java^{TM},* Fourth Edition, Global Edition**, Premium Content for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

1. Go to **www.pearsonglobaleditions.com/Carrano**
2. Click on **Companion Website.**
3. Click on the **Register** button.
4. On the registration page, enter your student access code* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
5. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
6. Once your personal Login Name and Password are confirmed, you can begin using the *Data Structures and Abstractions with Java^{TM}* Companion Website!

To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **www.pearsonglobaleditions.com/Carrano** by providing your Login Name and Password when prompted.

*Important: The access code can only be used once. This subscription is valid for twelve months upon activation and is not transferable.

```
28       int indexOfMin = first;
29       for (int index = first + 1; index <= last; index++)
30       {
31          if (a[index].compareTo(min) < 0)
32          {
33             min = a[index];
34             indexOfMin = index;
35          } // end if
36          // Assertion: min is the smallest of a[first] through a[index].
37       } // end for
38
39       return indexOfMin;
40    } // end getIndexOfSmallest
41
42    // Swaps the array entries a[i] and a[j].
43    private static void swap(Object[] a, int i, int j)
44    {
45       Object temp = a[i];
46       a[i] = a[j];
47       a[j] = temp;
48    } // end swap
49 } // end SortArray
```

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

Recursive Selection Sort

8.6 Selection sort also has a natural recursive form. Often recursive algorithms that involve arrays operate on a portion of the array. Such algorithms use two parameters, `first` and `last`, to designate the portion of the array containing the entries `a[first]` through `a[last]`. The method `getIndexOfSmallest` in Listing 8-1 illustrates this technique. The recursive selection sort algorithm uses this notation as well:

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

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

After we place the smallest entry into the first position of the array, we ignore it and sort the rest of the array by using a selection sort. If the array has only one entry, sorting is unnecessary. In this case, `first` and `last` are equal, so the algorithm leaves the array unchanged.

8.7 When we implement the previous recursive algorithm in Java, the resulting method will have `first` and `last` as parameters. Thus, its header will differ from the header of the iterative method `selectionSort` given in Segment 8.5. We could, however, provide the following method to simply invoke the recursive method:

```
public static <T extends Comparable<? super T>>
       void selectionSort(T[] a, int n)
{
   selectionSort(a, 0, n - 1); // Invoke recursive method
} // end selectionSort
```

Whether you make the recursive method `selectionSort` private or public is up to you, but making it public provides the client with a choice of two ways in which to invoke the sort. In a similar fashion, you could revise the iterative selection sort given in Segment 8.5 to use the parameters `first` and `last` (see Exercise 6) and then provide the method just given to invoke it.

With these observations in mind, we will make the subsequent sorting algorithms more general by giving them three parameters—`a`, `first`, and `last`—so that they sort the entries `a[first]` through `a[last]`.

The Efficiency of Selection Sort

8.8 In the iterative method `selectionSort`, the `for` loop executes $n - 1$ times, so it invokes the methods `getIndexOfSmallest` and `swap` $n - 1$ times each. In the $n - 1$ calls to `getIndexOfSmallest`, `last` is $n - 1$ and `first` ranges from 0 to $n - 2$. Each time `getIndexOfSmallest` is invoked, its loop executes `last - first` times. Since `last - first` ranges from $(n - 1) - 0$, or $n - 1$, to $(n - 1) - (n - 2)$, or 1, this loop executes a total of

$$(n - 1) + (n - 2) + \ldots + 1$$

times. This sum is $n (n - 1)/2$. Therefore, since each operation in the loop is O(1), the selection sort is $O(n^2)$. Notice that our discussion does not depend on the nature of the data in the array. It could be wildly out of order, nearly sorted, or completely sorted; in any case, selection sort would be $O(n^2)$.

The recursive selection sort performs the same operations as the iterative selection sort, and so it is also $O(n^2)$.

Note: The time efficiency of selection sort
Selection sort is $O(n^2)$ regardless of the initial order of the entries in an array. Although the sort requires $O(n^2)$ comparisons, it performs only $O(n)$ swaps. Thus, the selection sort requires little data movement.

Insertion Sort

8.9 Another intuitive sorting algorithm is the **insertion sort**. Suppose again that you want to rearrange the books on your bookshelf by height, with the shortest book on the left. If the leftmost book on the shelf were the only book, your shelf would be sorted. But you also have all the other books to sort. Consider the second book. If it is taller than the first book, you now

have two sorted books. If not, you remove the second book, slide the first book to the right, and *insert* the book you just removed into the first position on the shelf. The first two books are now sorted.

VideoNote
Insertion sort

Now consider the third book. If it is taller than the second book, you now have three sorted books. If not, remove the third book and slide the second book to the right, as Parts *a* through *c* of Figure 8-3 illustrate. Now see whether the book in your hand is taller than the first book. If so, insert the book into the second position on the shelf, as shown in Figure 8-3d. If not, slide the first book to the right, and insert the book in your hand into the first position on the shelf. If you repeat this process for each of the remaining books, your bookshelf will be arranged by the heights of the books.

FIGURE 8-3 The placement of the third book during an insertion sort

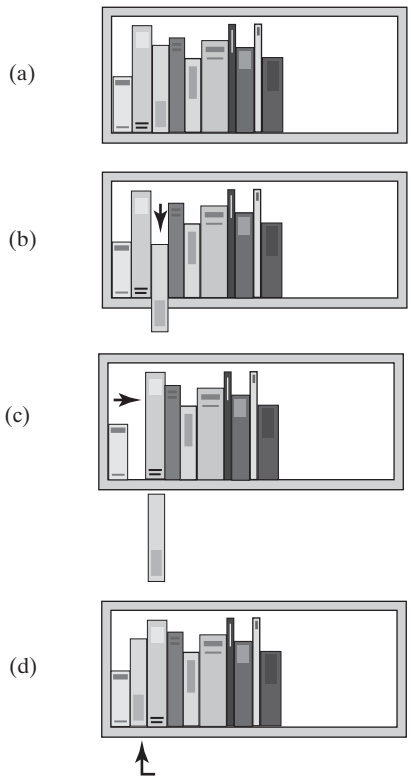
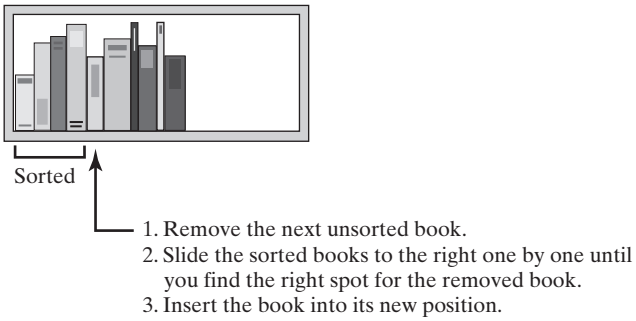


Figure 8-4 shows the bookshelf after several steps of the insertion sort. The books on the left side of the shelf are sorted. You remove the next unsorted book from the shelf and slide sorted books to the right, one at a time, until you find the right place for the book in your hand. You then insert this book into its new sorted location.

FIGURE 8-4 An insertion sort of books



Iterative Insertion Sort

8.10 An insertion sort of an array **partitions**—that is, divides—the array into two parts. One part is sorted and initially contains just the first entry in the array. The second part contains the remaining entries. The algorithm removes the first entry from the unsorted part and inserts it into its proper sorted position within the sorted part. Just as you did with the bookshelf, you choose the proper position by comparing the unsorted entry with the sorted entries, beginning at the end of the sorted part and continuing toward its beginning. As you compare, you shift array entries in the sorted part to make room for the insertion.

Figure 8-5 illustrates these steps for a sort that has already positioned the first three entries of the array. The 3 is the next entry that must be placed into its proper position within the sorted region. Since 3 is less than 8 and 5 but greater than 2, the 8 and 5 are shifted to make room for the 3.

FIGURE 8-5 Inserting the next unsorted entry into its proper location within the sorted portion of an array during an insertion sort

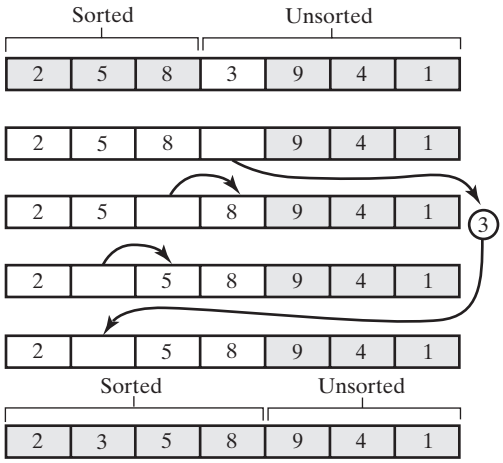
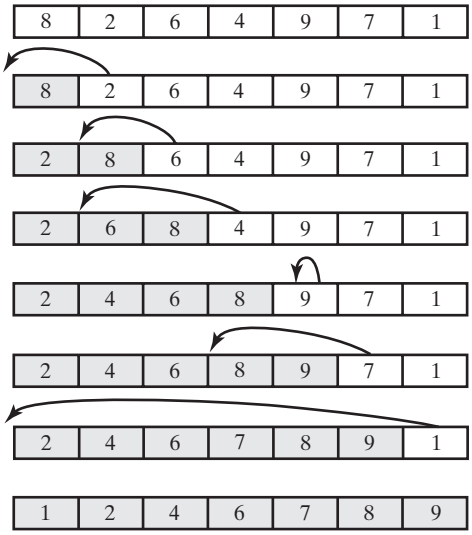


Figure 8-6 illustrates an entire insertion sort of an array of integers. At each pass of the algorithm, the sorted part expands by one entry as the unsorted part shrinks by one entry. Eventually, the unsorted part is empty and the array is sorted.

FIGURE 8-6 An insertion sort of an array of integers into ascending order



The following iterative algorithm describes an insertion sort of the entries at indices `first` through `last` of the array `a`. To sort the first n entries in the array, the call to the algorithm would be `insertionSort(a, 0, n - 1)`.

```
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)
}
```

The sorted part contains one entry, `a[first]`, and so the loop in the algorithm begins at index `first + 1` and processes the unsorted part. It then invokes another method—`insertInOrder`—to perform the insertions. In the pseudocode that follows for this method, `anEntry` is the value to be inserted into its proper position, and `begin` and `end` are array indices.

```
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.

Recursive Insertion Sort

8.11 You can describe an insertion sort recursively as follows. If you sort all but the last item in the array—a smaller problem than sorting the entire array—you then can insert the last item into its proper position within the rest of the array. The following pseudocode describes a recursive insertion sort:

```
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
}
```

We can implement this algorithm in Java as follows:

```
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
```

8.12 **The algorithm `insertInOrder`: first draft.** The previous method can call the iterative version of `insertInOrder`, given earlier, or the recursive version that we now describe. If the entry to insert is greater than or equal to the last item in the sorted portion of the array, the entry belongs immediately after this last item, as Figure 8-7a illustrates. Otherwise, we move the last sorted item to the next higher position in the array and insert the entry into the remaining portion, as shown in Figure 8-7b.

We can describe these steps more carefully as follows:

```
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)
}
```