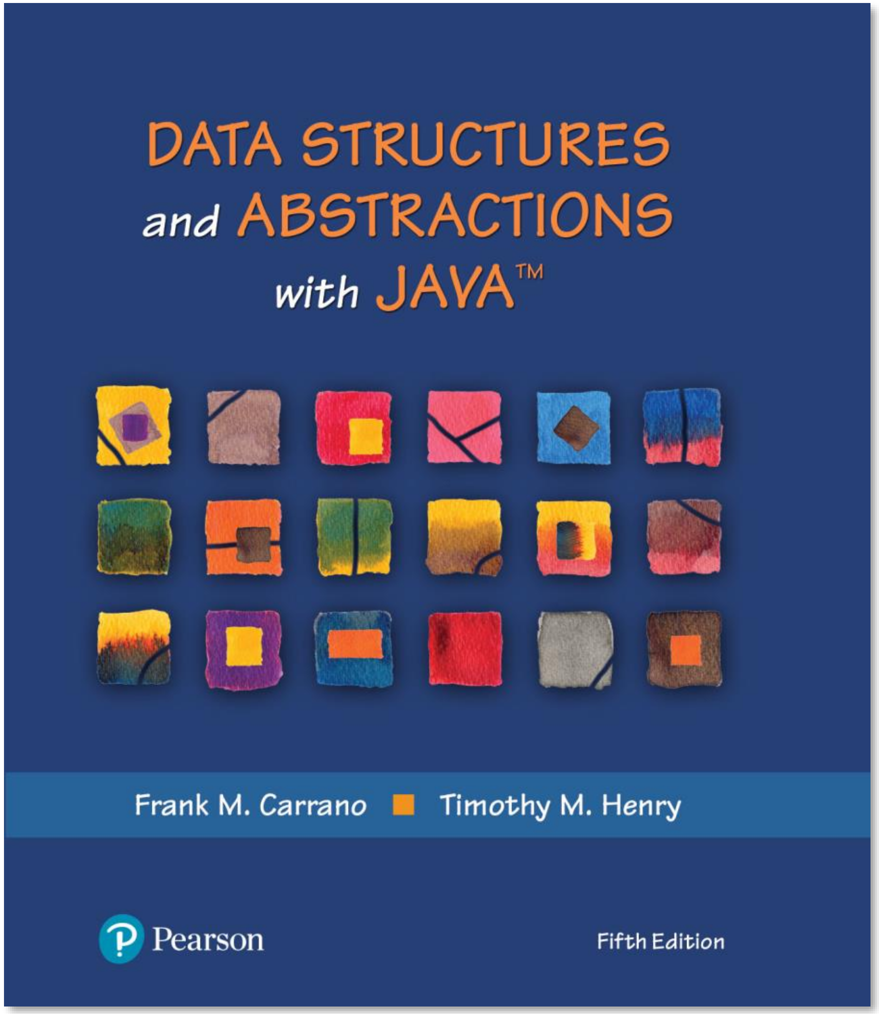
Data Structures and Abstractions with Java™

5th Edition



Chapter 15

An Introduction to Sorting

Sorting

- We seek algorithms to arrange items, $\mathbf{a_i}$ such that:

$$\texttt{entry 1} \le \texttt{entry 2} \le \ldots \le \texttt{entry n}$$

- Sorting an array is usually easier than sorting a chain of linked nodes
- Efficiency of a sorting algorithm is significant

SortUtilities

- Contains methods implementing all sort algorithms contained in subsequent chapters
- Also contains a swap method
- All require that objects implement the revised ListInterface

Selection Sort

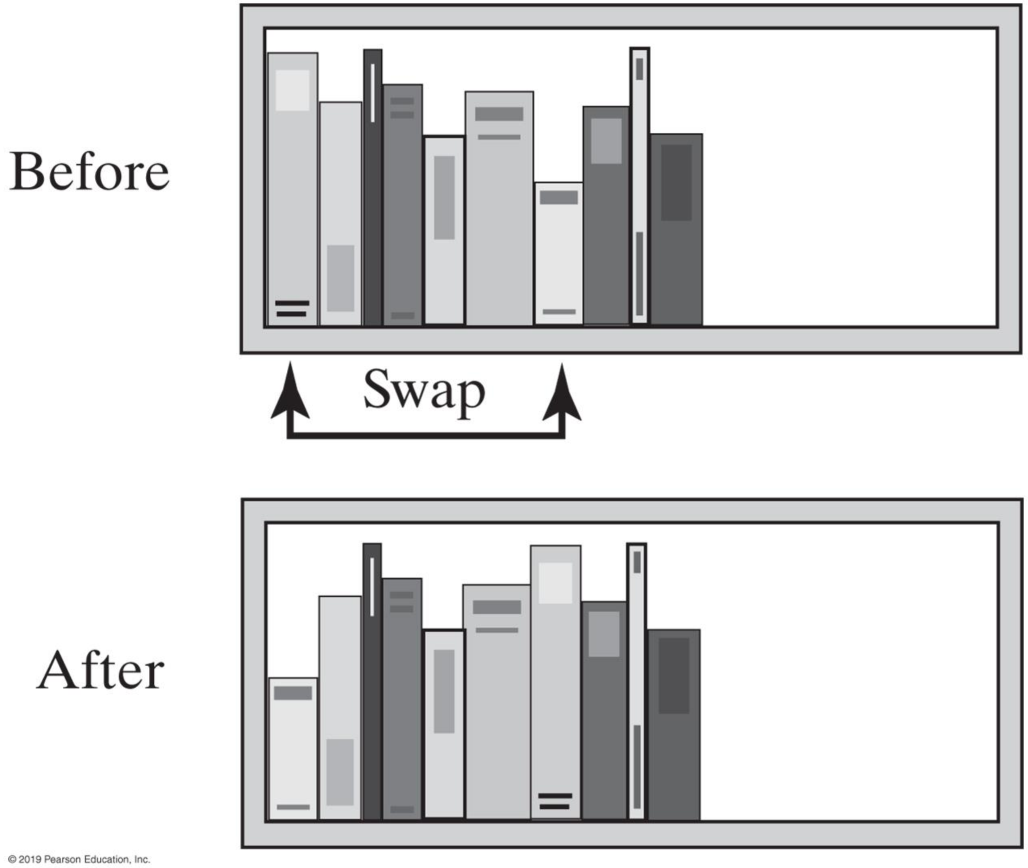


FIGURE 15-1 Before and after exchanging the shortest book and the first book

Selection Sort

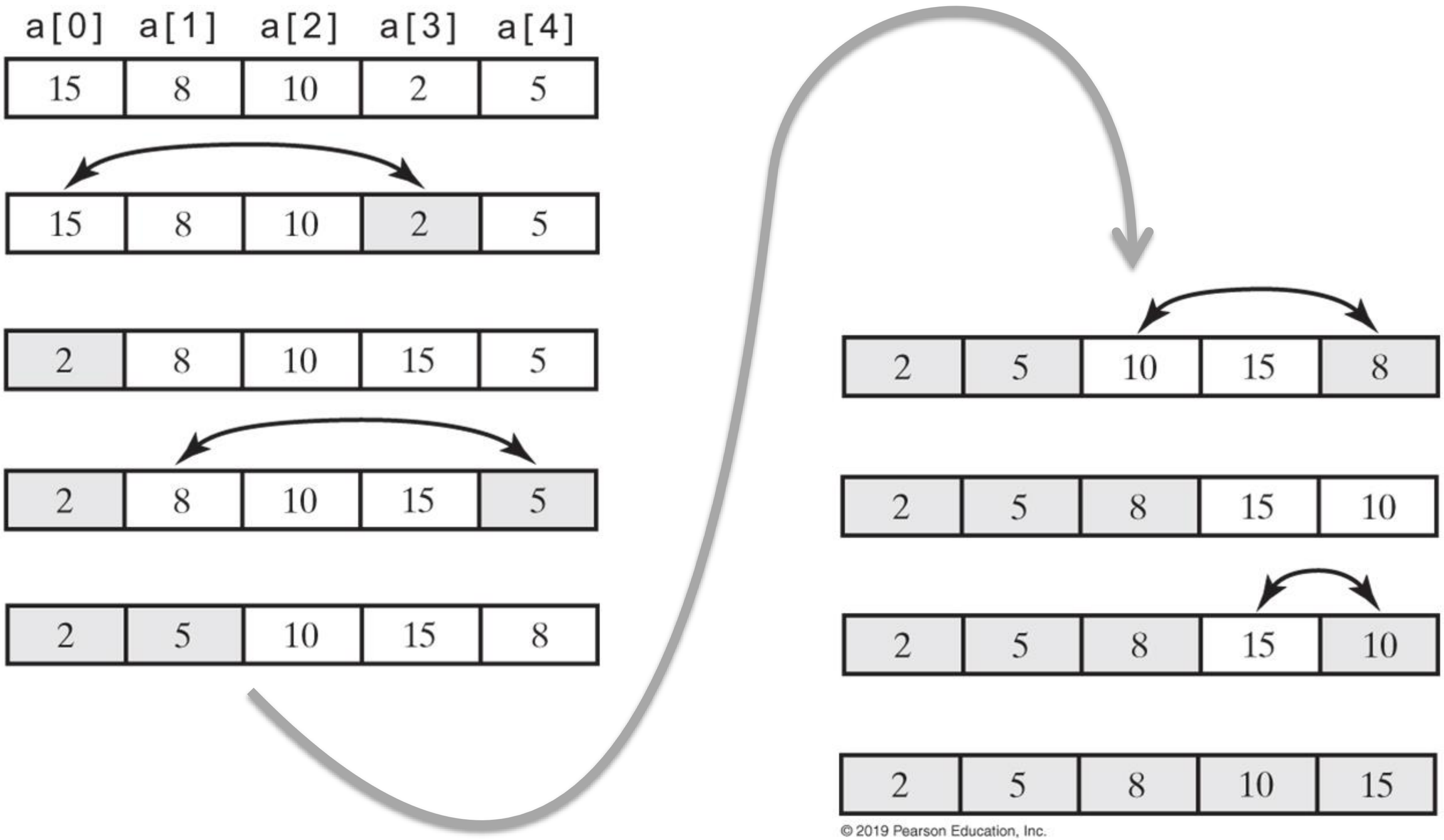


FIGURE 15-2 A selection sort of an array of integers into ascending order

Iterative Selection Sort

```
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].
}
```

This pseudocode describes an iterative algorithm for the selection sort

Selection Sort

```
/**
 * Selection sort
 *
 * iterate through the list, finding the smallest in the rest of the list then swapping
 * @param list
 * @param first beginning of range to sort
 * @param last end of range to sort
 */
static public <T extends Comparable<? super T>> void selectionSort(ListInterface<T> list, int first, int last) {

    for (int index = first; index <= last; index++) {
        // find the smallest in the rest of the list, then swap
        int nextSmallest = findSmallest(list, index, last);
        swap(list, index, nextSmallest);
    }
}
/**
 * return the index (position) of the smallest entry in the list
 * @param list
 * @param first
 * @param last
 * @return
 */
static private <T extends Comparable<? super T>> int findSmallest(ListInterface<T> list, int first, int last) {
    T minimum = list.getEntry(first);

    int indexOfMinimum = first;

    for (int index = first + 1; index <= last; index++) {
        T temp = list.getEntry(index);
        if (temp.compareTo(minimum) < 0) {
            minimum = temp;
            indexOfMinimum = index;
        }
    }

    return indexOfMinimum;

}
```

Swap

- This is used by other sort methods as well
- Note use of replace() and getEntry()

```java
/**
 * Swap the data in the given positions in the list
 *
 * @param list
 * @param first  position to swap
 * @param second position to swap
 */
static public <T> void swap(ListInterface<T> list, int first, int second) {

    if (first == second || list == null)
            return;

    T firstEntry = list.getEntry(first);
    T secondEntry = list.getEntry(second);

    list.replace(first, secondEntry);
    list.replace(second, firstEntry);

}
```

Recursive Selection Sort

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)
}

Recursive selection sort algorithm

Pearson

Efficiency of Selection Sort

- Selection sort is $O(n^2)$ regardless of the initial order of the entries.
  - Requires $O(n^2)$ comparisons
  - Does only $O(n)$ swaps

Insertion Sort

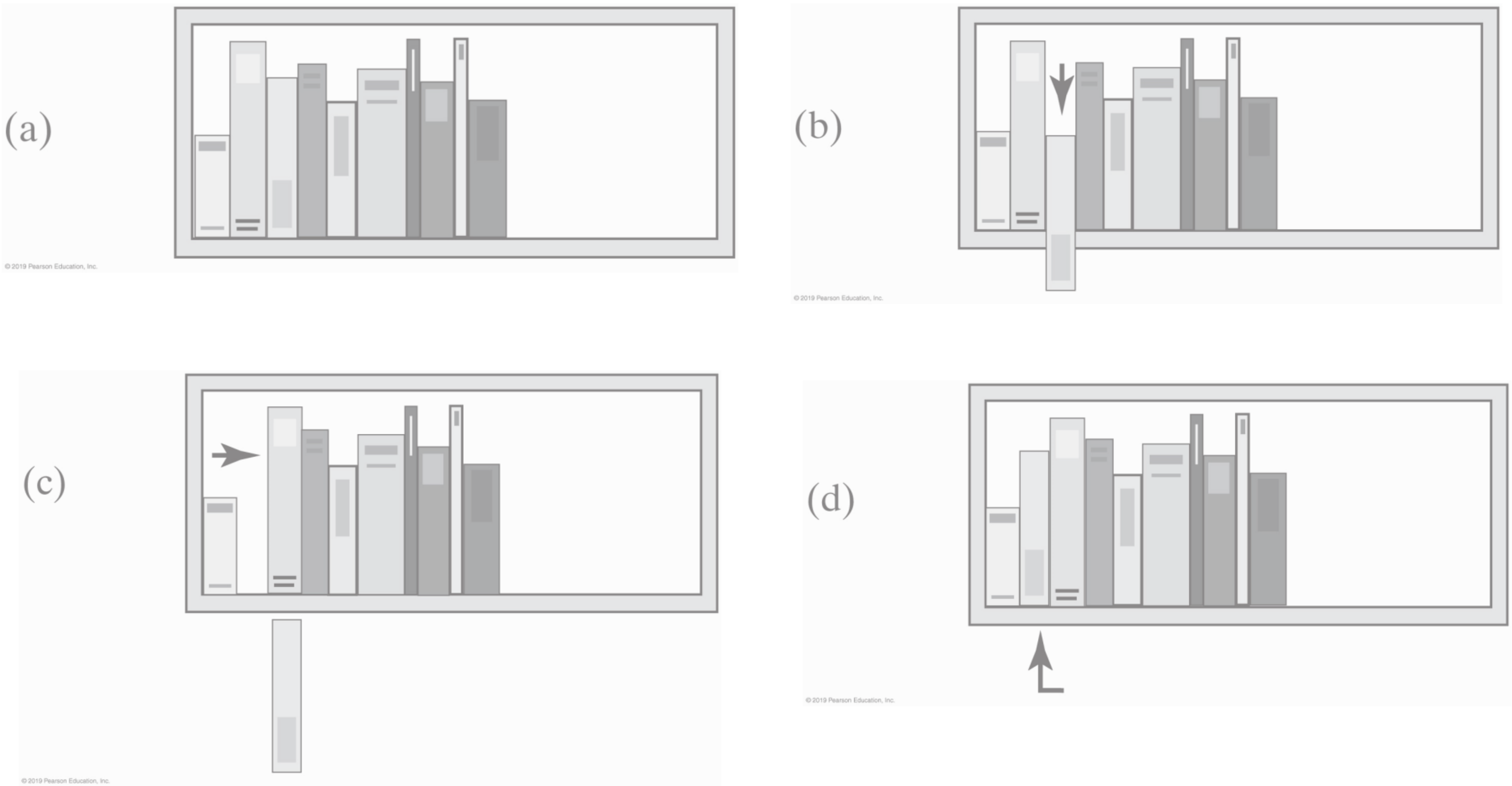
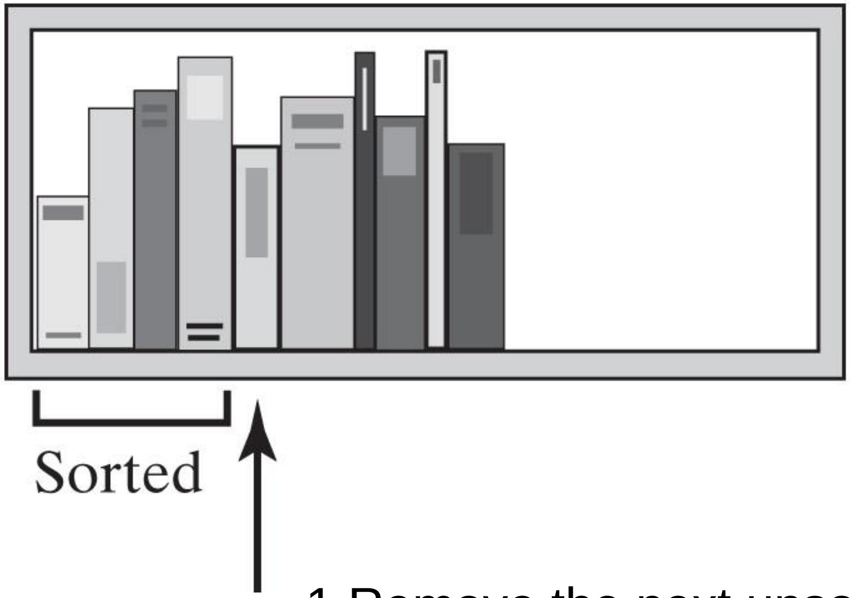


FIGURE 15-3 The placement of the third book during an insertion sort

Pearson

Insertion Sort



1. Remove the next unsorted book.
2. Slide the sorted books to the right one by one until you find the right spot for the removed book.
3. Insert the book into its new position

FIGURE 15-4 An insertion sort of books

Pearson

Insertion Sort

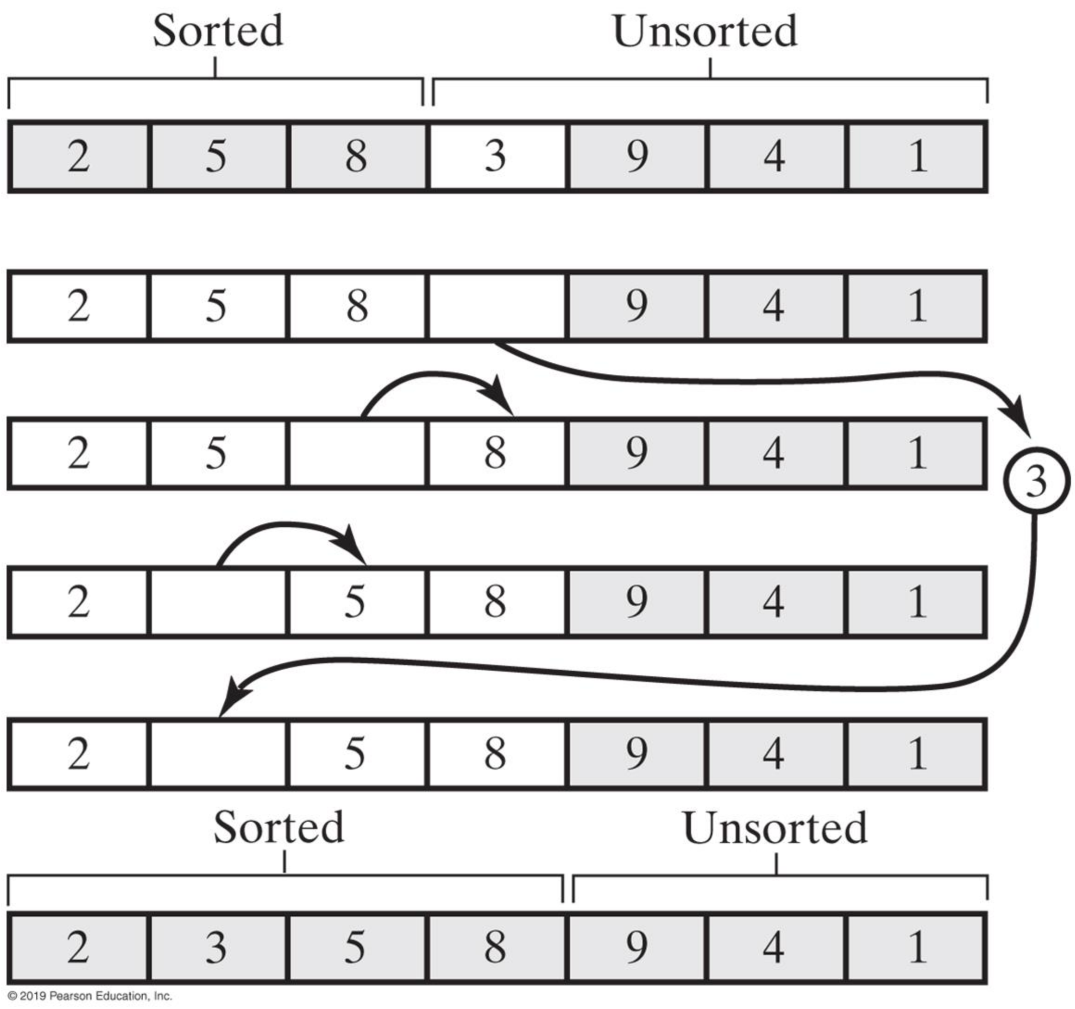


FIGURE 15-5 Inserting the next unsorted entry into its proper location within the sorted portion of an array during an insertion sort

Insertion Sort

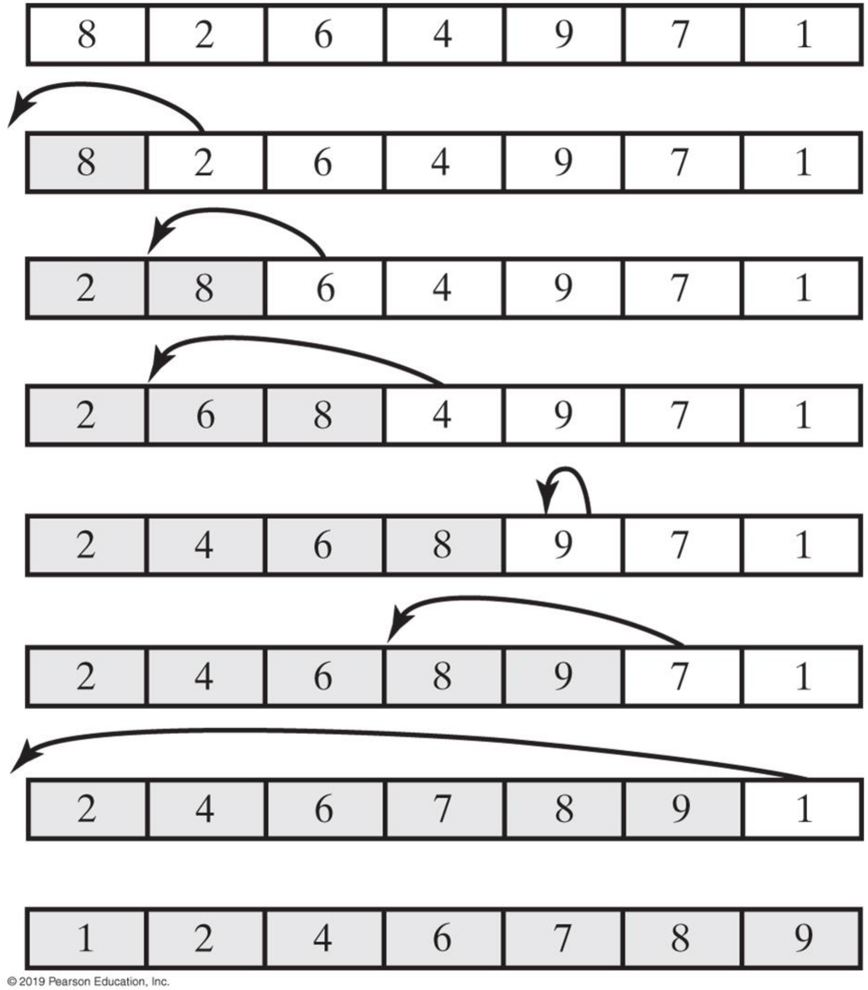


FIGURE 15-6 An insertion sort of an array of integers into ascending order

Iterative Insertion Sort

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)

}

Iterative algorithm describes an insertion sort of the entries at indices first through last of the array a

Iterative Insertion Sort

```
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
```

Pseudocode of method, `insertInOrder`, to perform the insertions.

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*

}

This pseudocode describes a recursive insertion sort.

Recursive Insertion Sort

- See SortUtilities code

```java

    /**
     * Recursive insertion sort
     *
     * Recursively call insertion sort until we only have a list of one, then begin to insert
     * the current entry into the sorted sub list portion.
     * @param list
     * @param first
     * @param last
     */
    static public <T extends Comparable<? super T>> void recursiveInsertionSort(ListInterface<T> list, int first,
            int last) {
        if (first < last) {
            insertionSort(list, first, last - 1);
            T next = list.getEntry(last);
            insertInOrder(next, list, first, last - 1);
        }
    }

    /**
     * Compare an item to each list entry and insert it in the proper position.
     *
     *
     * @param item
     * @param list
     * @param first
     * @param last
     */
    static public <T extends Comparable<? super T>> void insertInOrder(T item, ListInterface<T> list, int first,
            int last) {

        // work from the last to first, since we have to shift items

        int index = last;

        for (; index >= first; index--) {
            T current = list.getEntry(index);
            // shift the item to the right if it is larger
            if (current.compareTo(item) > 0)
                list.replace(index + 1, current);
            else
                break;
        }
        // went one too far, replace current item in the right slot.
        list.replace(index + 1, item);

    }

    /**
```

Insertion Sort

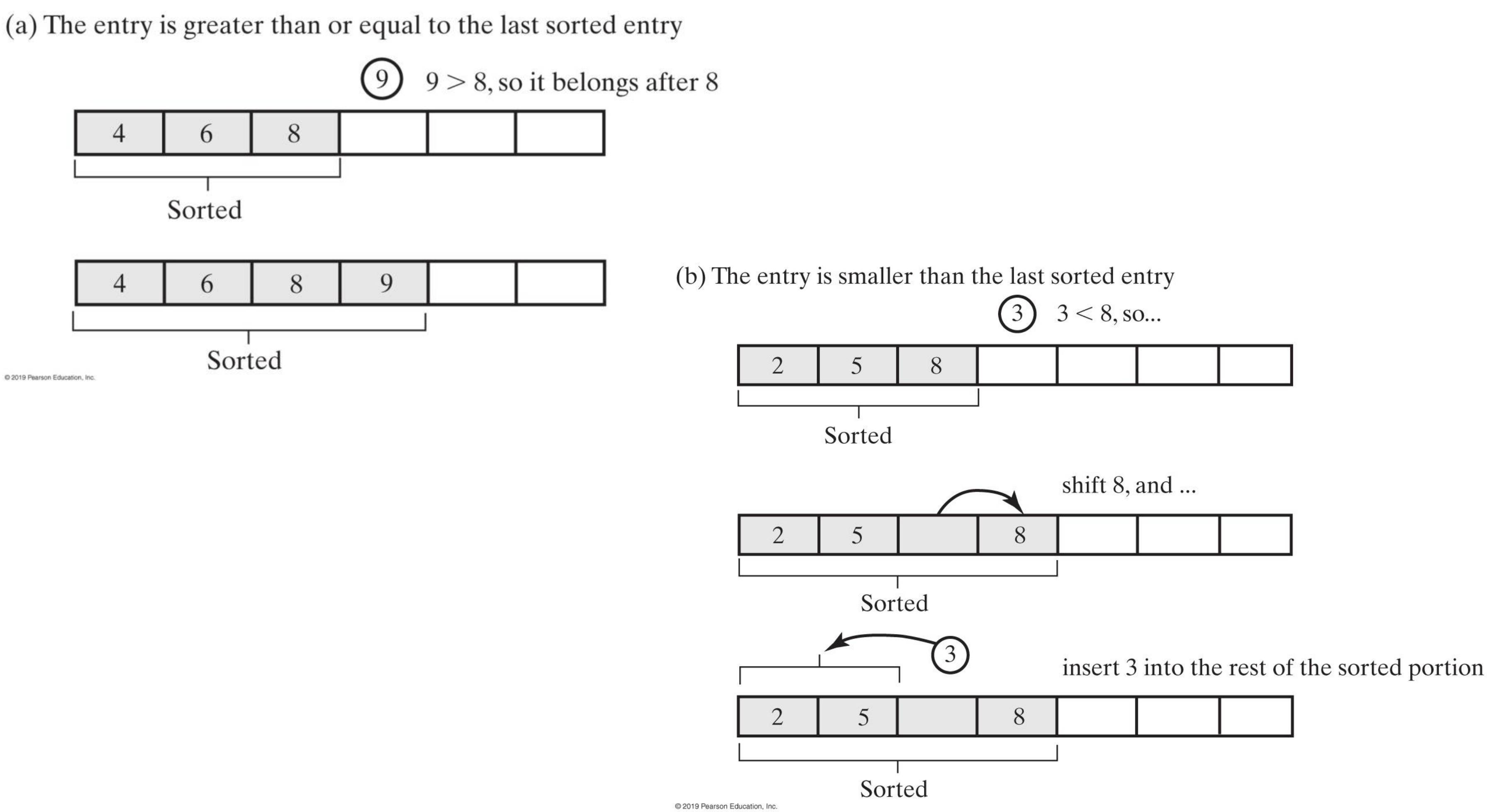


FIGURE 15-7 Inserting the first unsorted entry into the sorted portion of the array

Insertion Sort with a Linked Chain

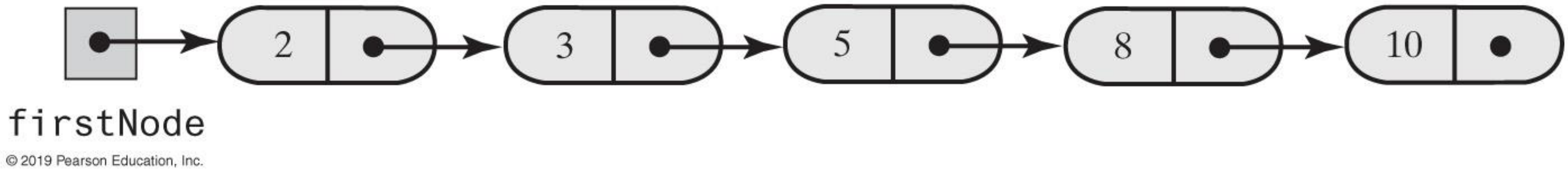


FIGURE 15-8 A chain of integers sorted into ascending order

Insertion Sort with a Linked Chain

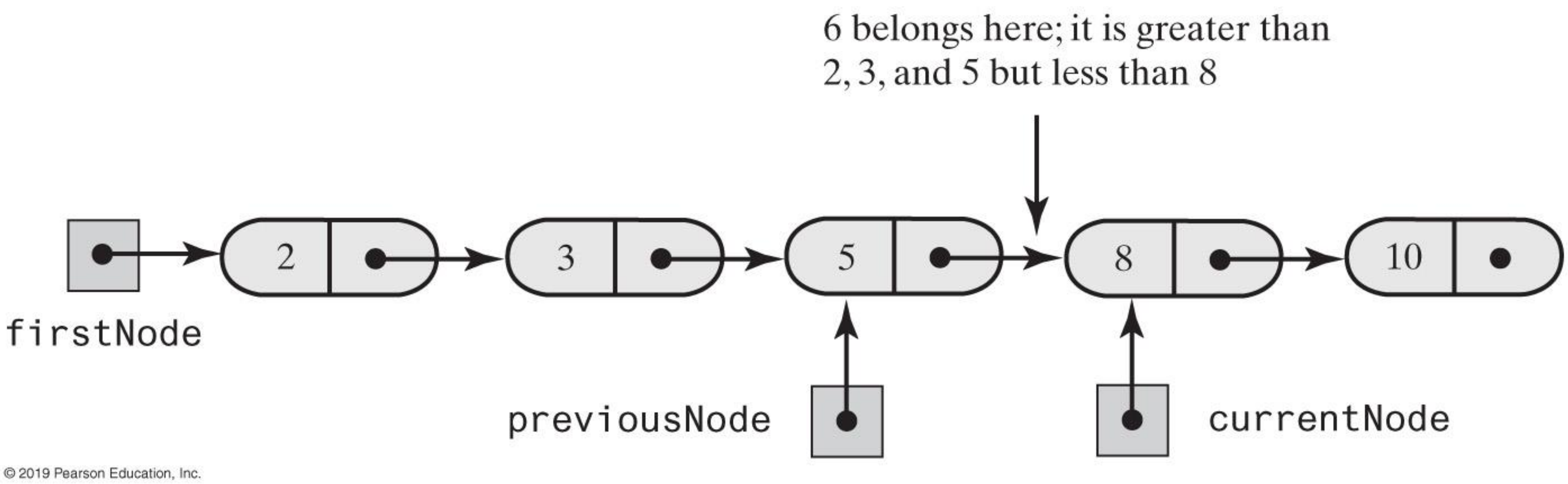


FIGURE 15-9 During the traversal of a chain to locate the insertion point, save a reference to the node before the current one

Insertion Sort with a Linked Chain

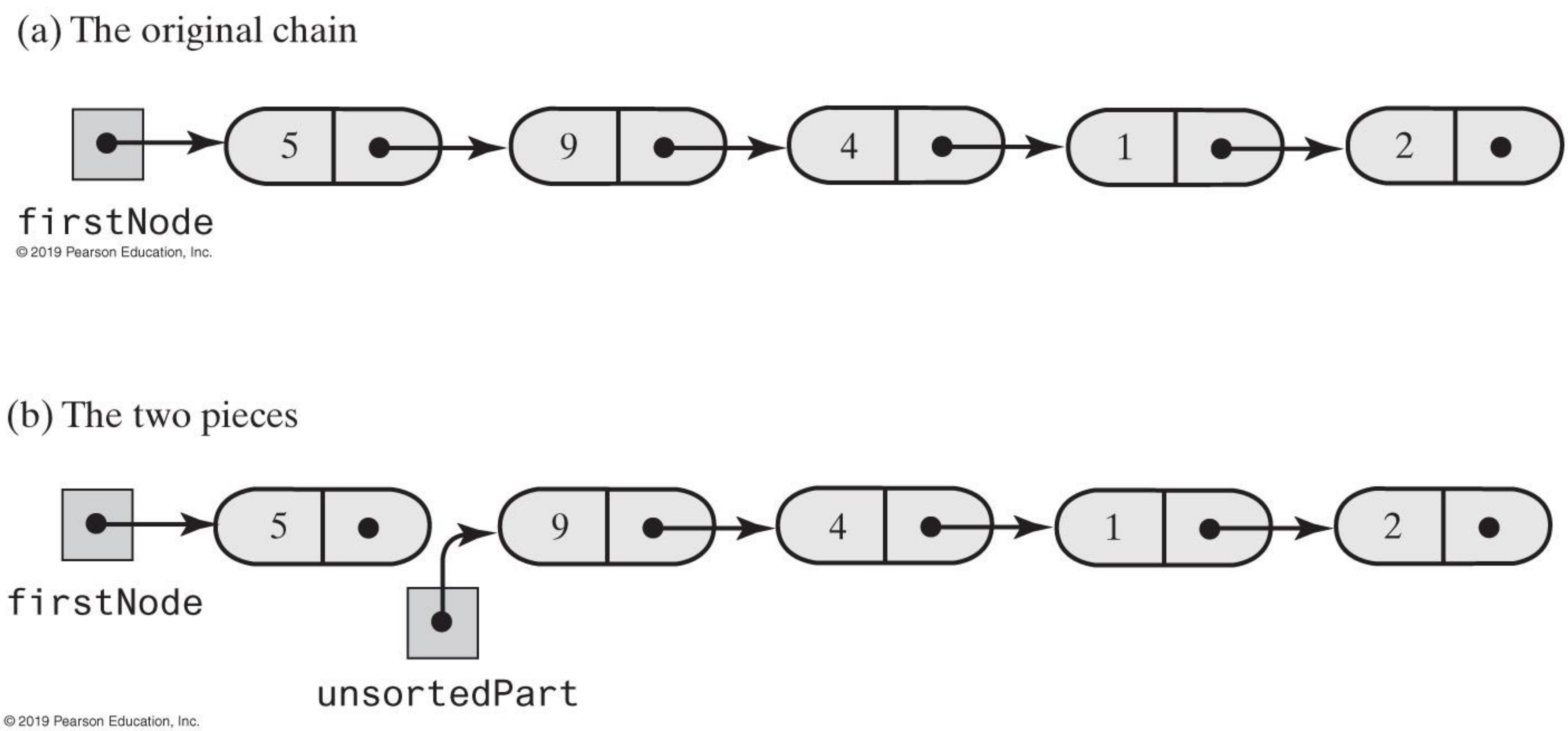


FIGURE 15-10 Breaking a chain of nodes into two pieces as the first step in an insertion sort

Insertion Sort with a Linked Chain

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

   //  . . .
        private class Node
        {
            //  private inner class Node is implemented here.
         }
```

Add a sort method to a class `LinkedGroup` that uses a linked chain to represent a certain collection

Insertion Sort with a Linked Chain

```
private void insertInOrder(Node nodeToInsert)
  {
    T item = nodeToInsert.getData();
    Node currentNode = firstNode;
    Node previousNode = null;

    // Locate insertion point
    while ( (currentNode != null) &&
          (item.compareTo(currentNode.getData()) > 0) )
    {
      previousNode = currentNode;
      currentNode = currentNode.getNextNode();
    } // end while

    // Make the insertion
    if (previousNode != null)
    {  // Insert between previousNode and currentNode
      previousNode.setNextNode(nodeToInsert);
      nodeToInsert.setNextNode(currentNode);
    }
    else // Insert at beginning
    {
      nodeToInsert.setNextNode(firstNode);
      firstNode = nodeToInsert;
    } // end if
  } // end insertInOrder
```

Class has an inner class `Node` with `set` and `get` methods

Insertion Sort with a Linked Chain

```
public void insertionSort()
{
   // If fewer than two items are in the list, there is nothing to do
   if (length > 1)
   {
      // Assertion: firstNode != null

      // Break chain into 2 pieces: sorted and unsorted
      Node unsortedPart = firstNode.getNextNode();
      // Assertion: unsortedPart != null
      firstNode.setNextNode(null);

      while (unsortedPart != null)
      {
         Node nodeToInsert = unsortedPart;
         unsortedPart = unsortedPart.getNextNode();
         insertInOrder(nodeToInsert);
      } // end while
   } // end if
} // end insertionSort
```

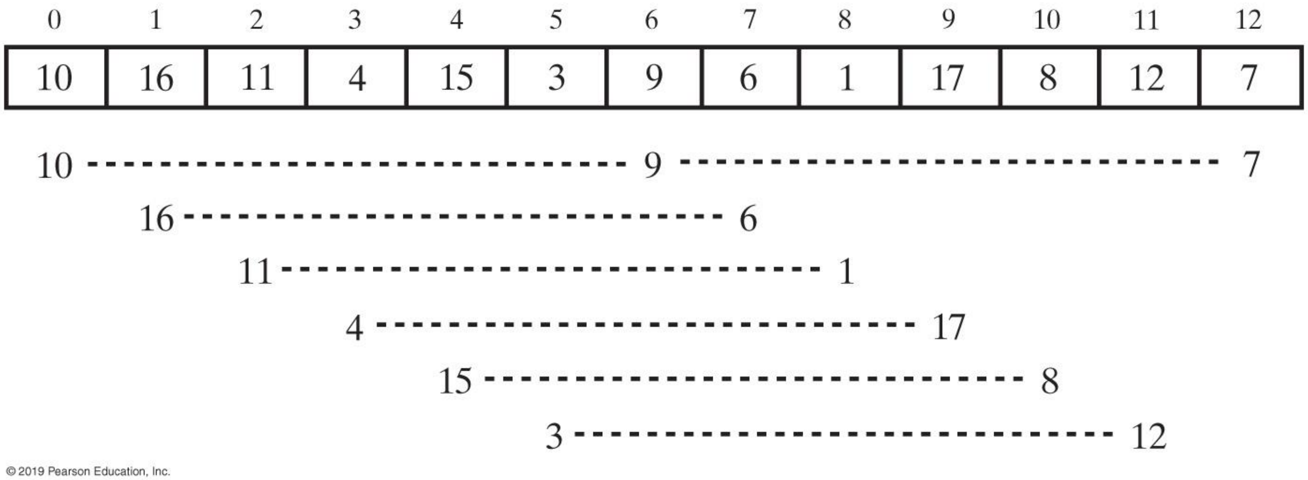
Insertion sort method

Shell Sort

- Algorithms so far are simple
  - but inefficient for large arrays at $O(n^2)$
- The more sorted an array is, the less work `insertInOrder` must do
- Improved insertion sort developed by Donald Shell

Shell Sort

Before Ordering



After Ordering

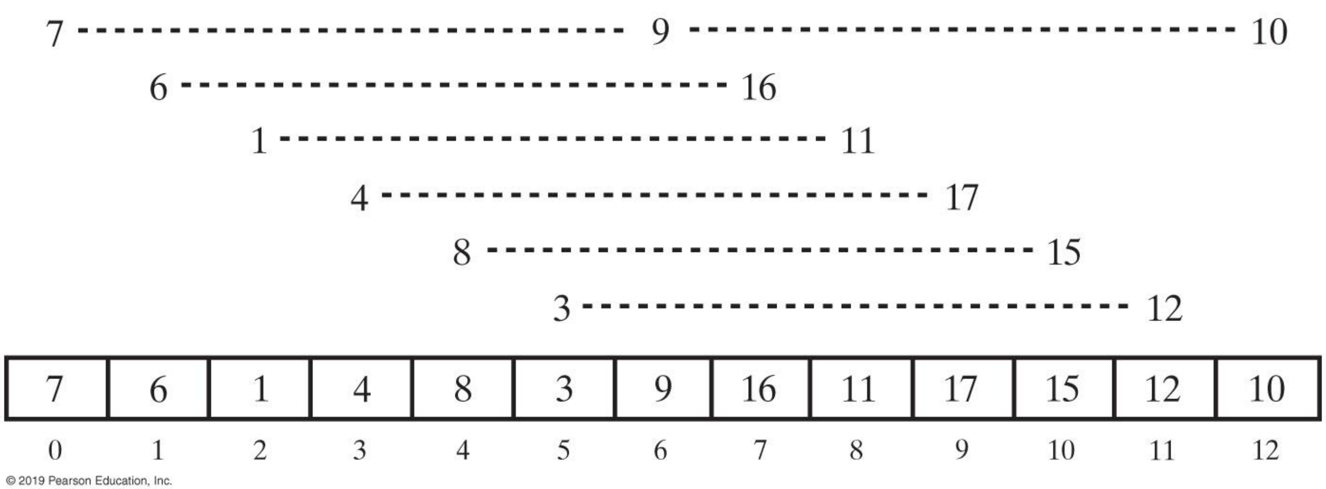
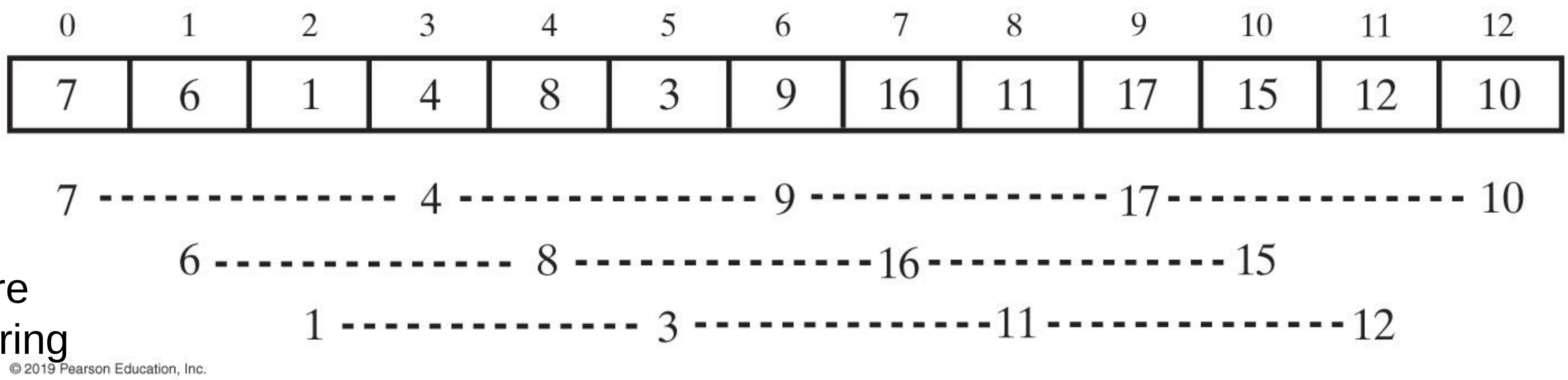


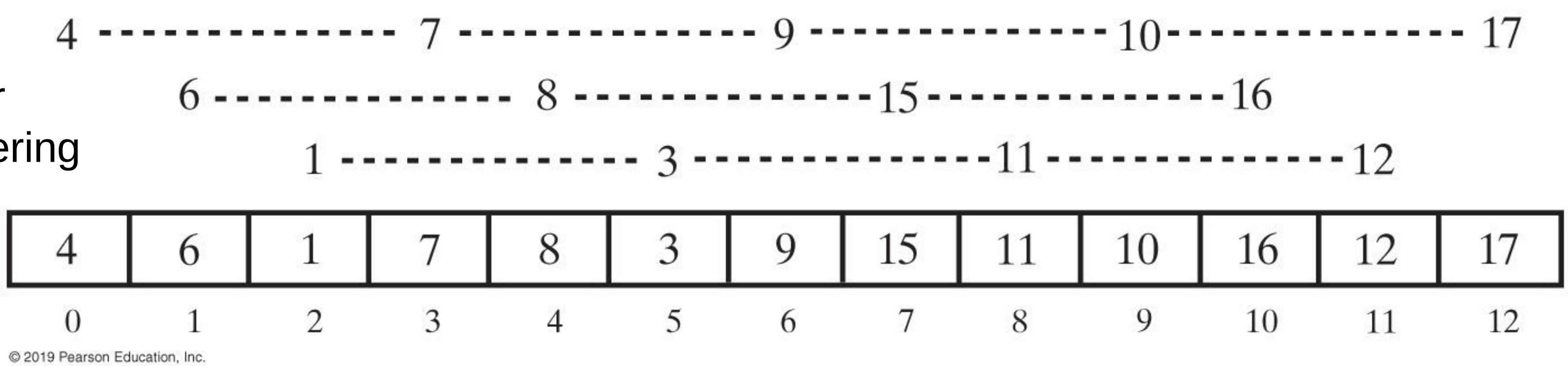
FIGURE 15-11 An array and the groups of entries whose indices are 6 apart before and after ordering groups

Shell Sort



Before Ordering

After Ordering



Grouped entries in the array in Figure 15-12 whose indices are 3 apart before and after ordering groups

Comparing Algorithms

	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	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$

FIGURE 15-15 The time efficiencies of three sorting algorithms, expressed in Big Oh notation

End

Chapter 15