# Class 07 – Sorting

CSIS 3475

Data Structures and Algorithms

# Sorting

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

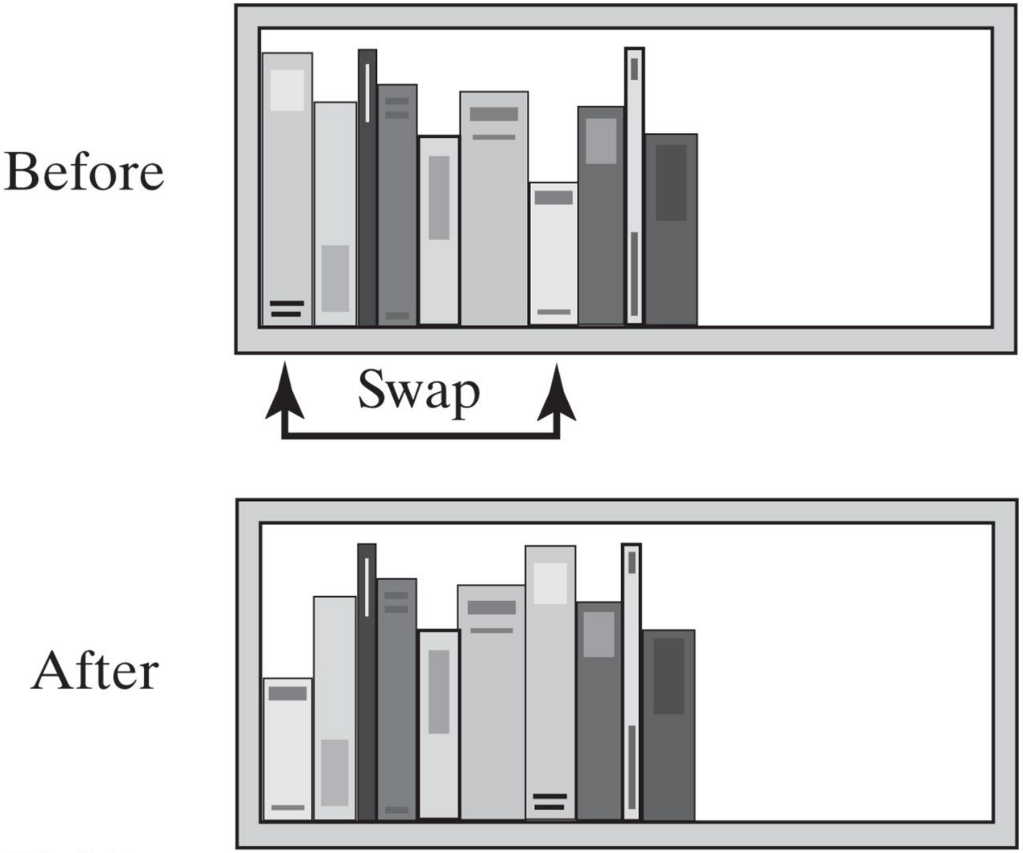
  **entry 1 ≤ entry 2 ≤ . . . ≤ entry n**
- Sorting an array is usually easier than sorting a chain of linked nodes
- Efficiency of a sorting algorithm is significant

# SortUtilities

- Contains methods implementing all sort algorithms contained in subsequent chapters
- Also contains a swap method
- All require that objects implement the revised ListInterface
- Sort implementations use ListInterface methods getEntry(), replace() and others.
  - Generally, these are efficient for arrays, but not so for lists.

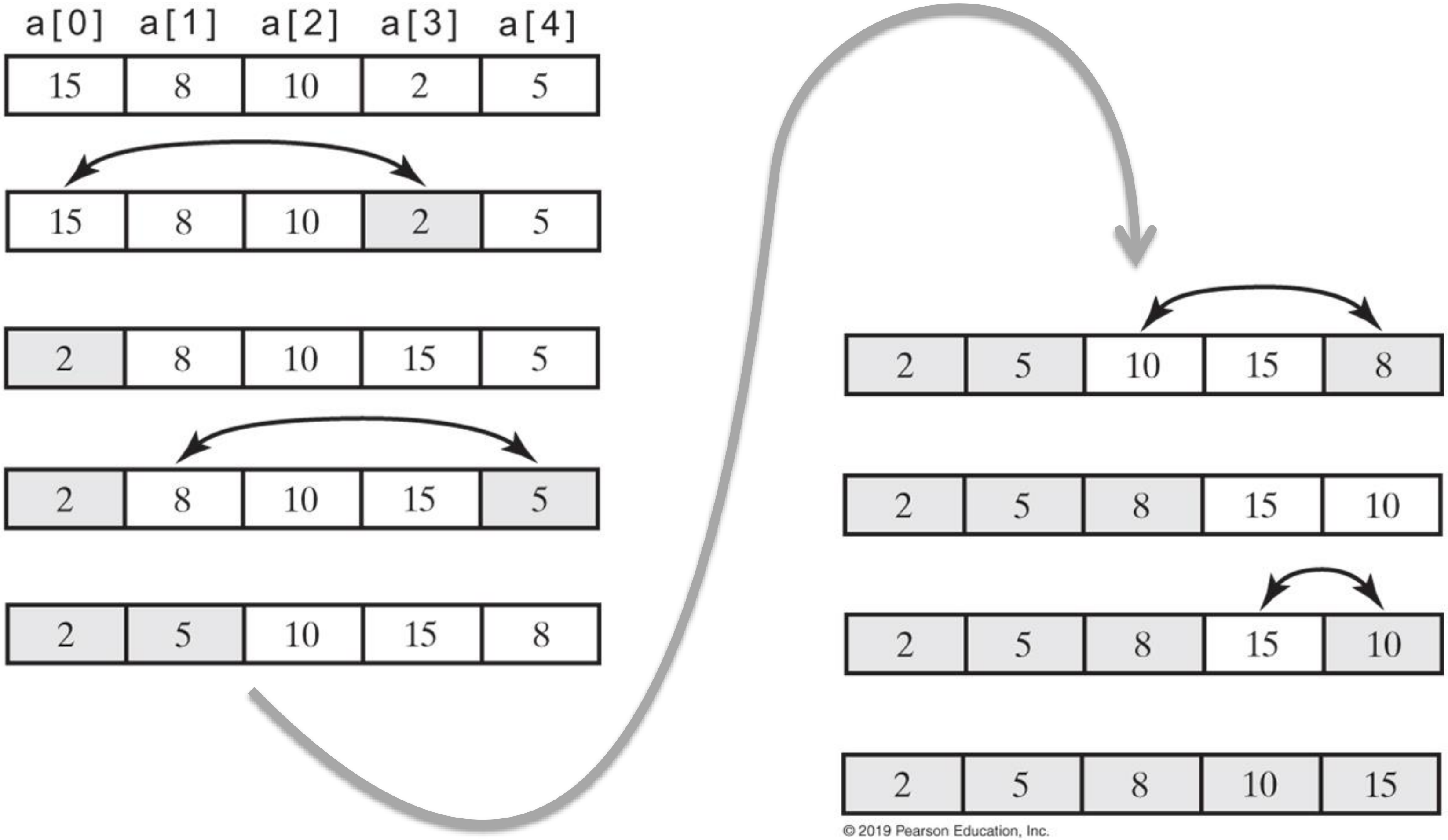
# Selection Sort

- For each element, find the smallest in the rest of the list and swap
- If the current element is the smallest, do nothing
- Example: Before and after exchanging the shortest book and the first book



© 2019 Pearson Education, Inc.

# Selection Sort of an integer array



| a[0] | a[1] | a[2] | a[3] | a[4] |
|---|---|---|---|---|
| 15 | 8 | 10 | 2 | 5 |
| 15 | 8 | 10 | 2 | 5 |
| 2 | 8 | 10 | 15 | 5 |
| 2 | 8 | 10 | 15 | 5 |
| 2 | 5 | 10 | 15 | 8 |
| 2 | 5 | 10 | 15 | 8 |
| 2 | 5 | 8 | 15 | 10 |
| 2 | 5 | 8 | 15 | 10 |
| 2 | 5 | 8 | 10 | 15 |

© 2019 Pearson Education, Inc.

# Iterative Selection Sort

- This pseudocode describes an iterative algorithm for the selection sort

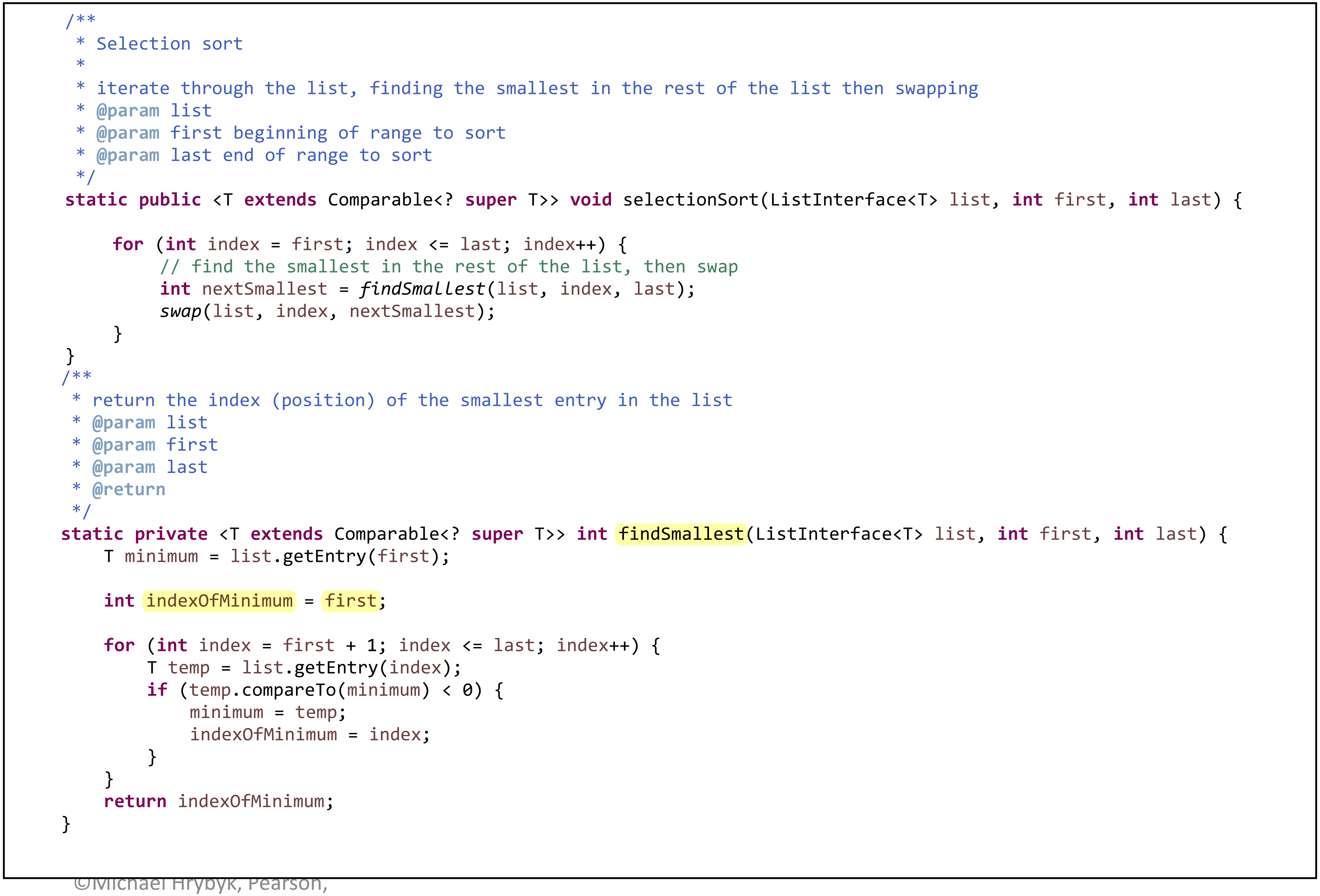
```
Algorithm selectionSort(a, n)
// Sorts the first n entries of an array a.
for (index = 0; index < n − 1; index++)
{
        indexOfNextSmallest = the index of the smallest value among
                                                  a[index], a[index + 1], . . . , a[n − 1]
        Interchange the 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].
}
```

# Swap

- This is used by other sort methods as well
- Note use of replace() and getEntry()
- Inefficient for lists, needs to traverse for each method call

```
/**
 * Swap the data in the given positions in the list using getEntry()
 * and replace() methods from ListInterface.
 *
 * Need to catch IndexOutOfBoundsException
 *
 * @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;

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

        list.replace(first, secondEntry);
        list.replace(second, firstEntry);
    } catch (IndexOutOfBoundsException e) {
        return;
    }
}
```

# Selection Sort - Iterative



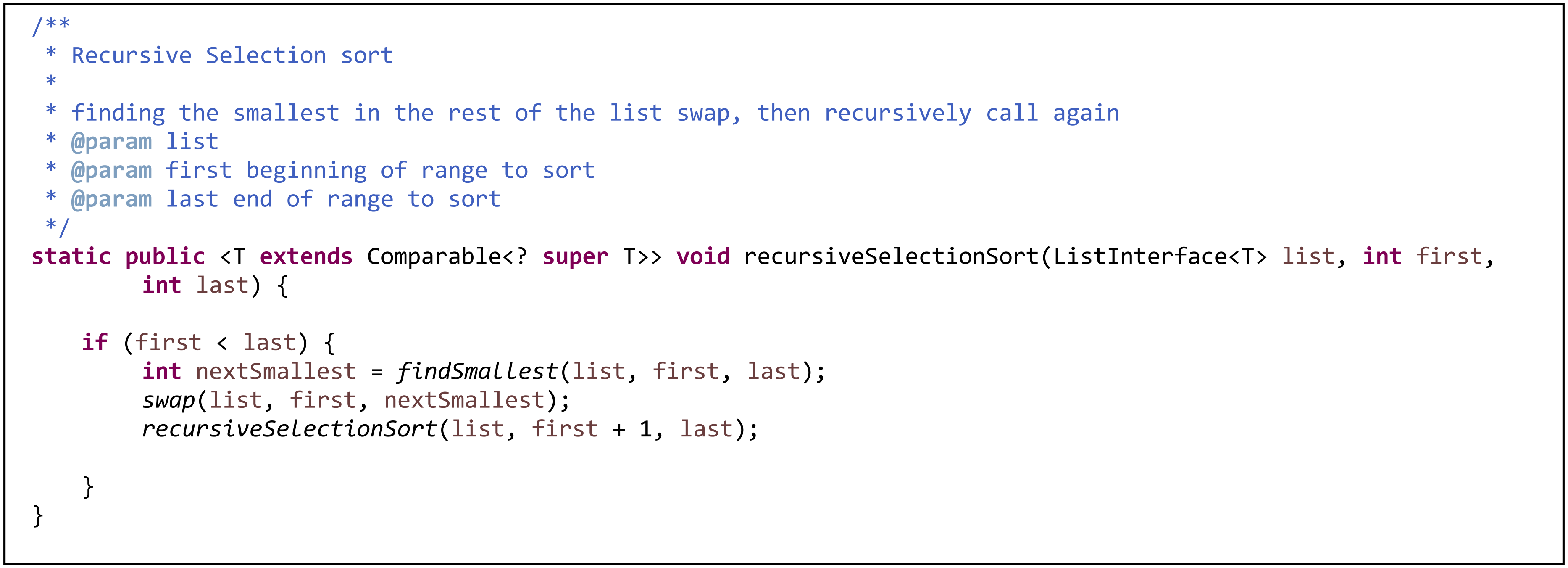
```
/**
 * Selection sort
 *
 * iterate through the list, finding the smallest in the rest of the list then swapping
 * @param list
 * @param first beginning of range to sort
 * @param last end of range to sort
 */
static public <T extends Comparable<? super T>> void selectionSort(ListInterface<T> list,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

# 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)
}
```

# Selection Sort - recursive

- Uses findSmallest(), as done in iterative version



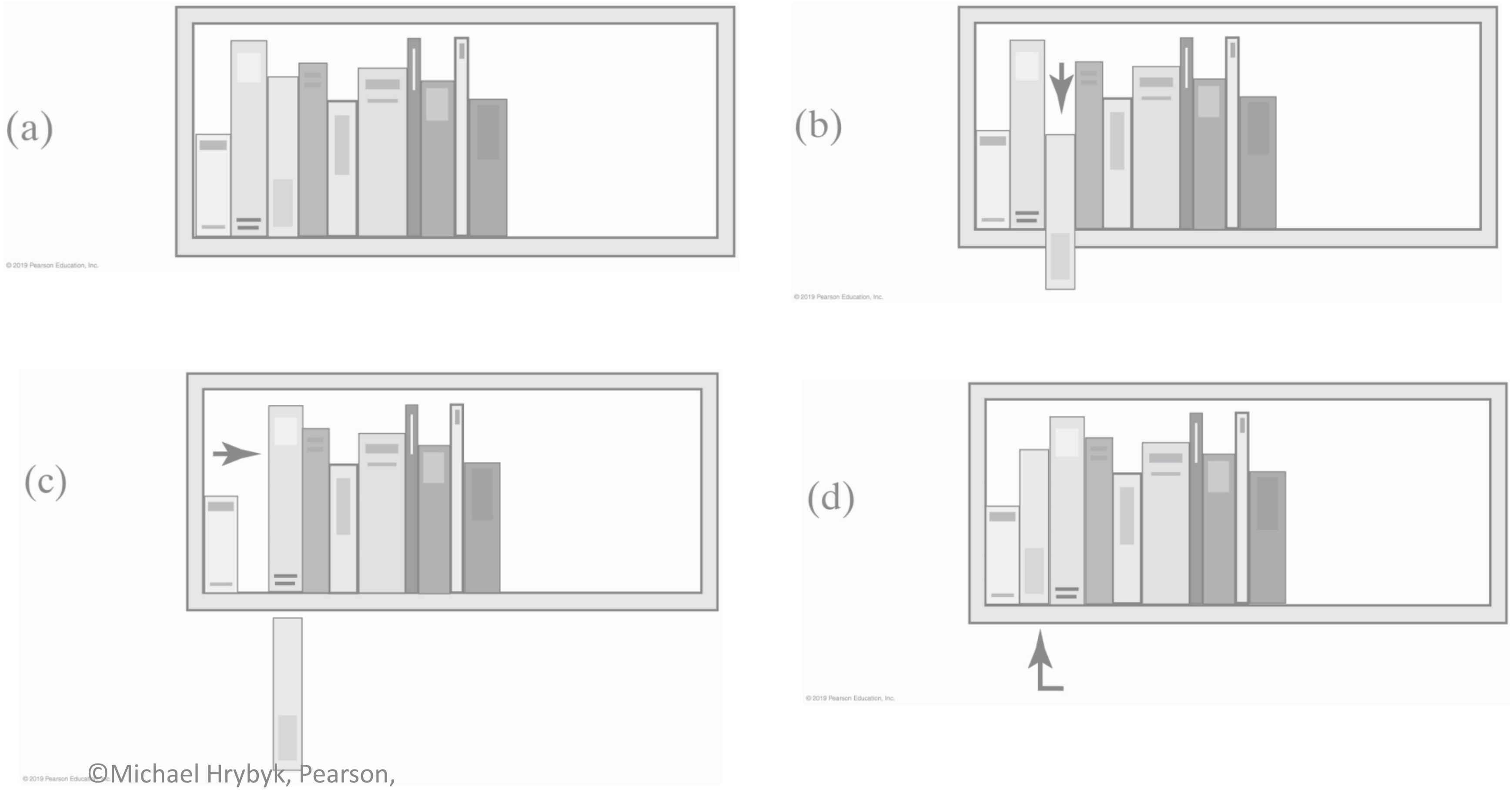
```java
/**
 * Recursive Selection sort
 *
 * finding the smallest in the rest of the list swap, then recursively call again
 * @param list
 * @param first beginning of range to sort
 * @param last end of range to sort
 */
static public <T extends Comparable<? super T>> void recursiveSelectionSort(ListInterface<T> list, int first,
        int last) {

    if (first < last) {
        int nextSmallest = findSmallest(list, first, last);
        swap(list, first, nextSmallest);
        recursiveSelectionSort(list, first + 1, last);

    }
}
```

# Efficiency of Selection Sort

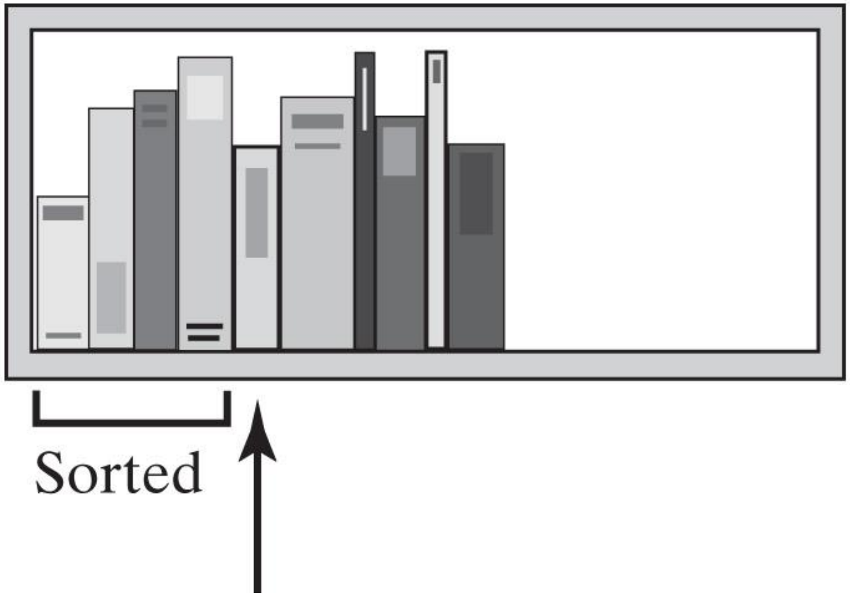
- Selection sort is $O(n^2)$ regardless of the initial order of the entries.
    - Requires $O(n^2)$ comparisons
    - Does only $O(n)$ swaps

# Insertion Sort

- Start a new sorted list with the first element
- For each other element, insert into the sorted list
- Example: The placement of the third book during an insertion sort


(a)
© 2019 Pearson Education, Inc.
(b)
© 2019 Pearson Education, Inc.
(c)
© 2019 Pearson Education, Inc.
(d)
© 2019 Pearson Education, Inc.

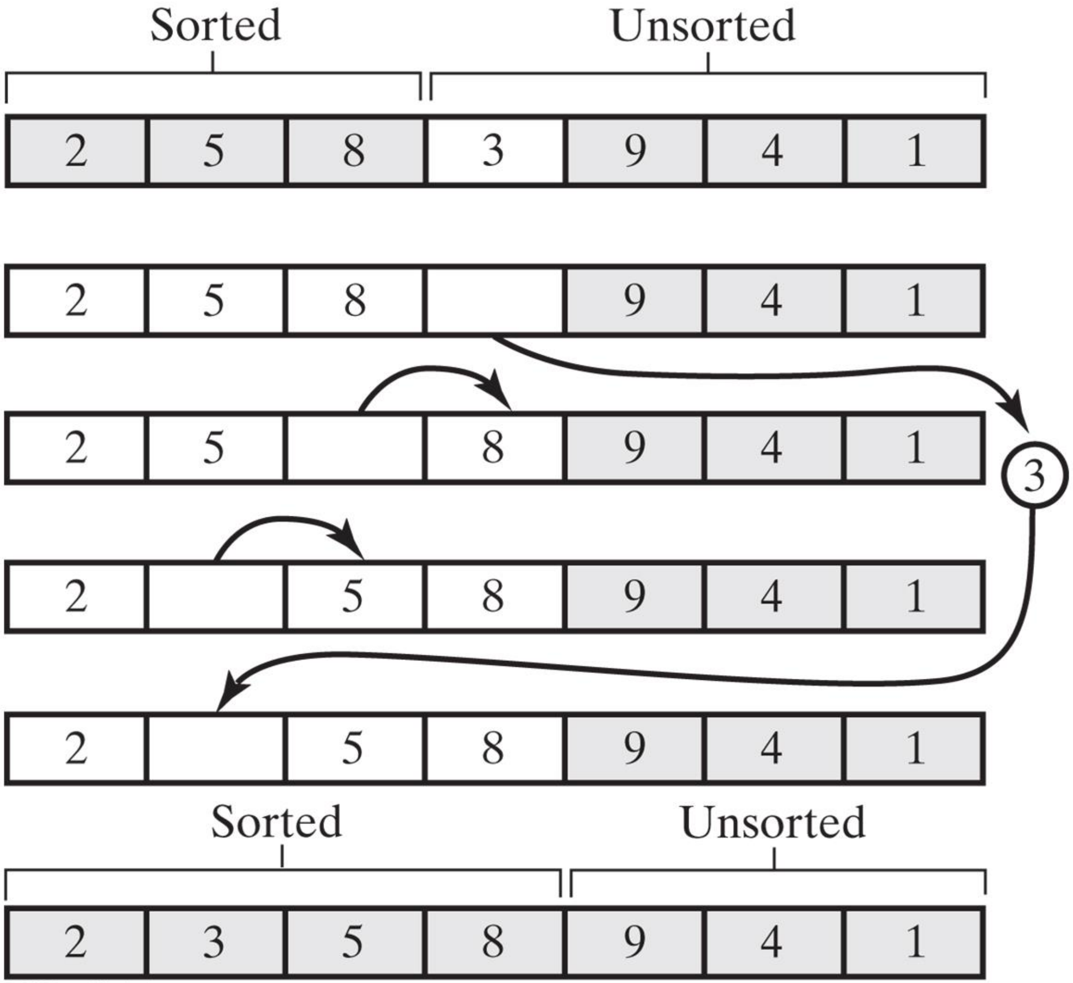
# Insertion Sort of books



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

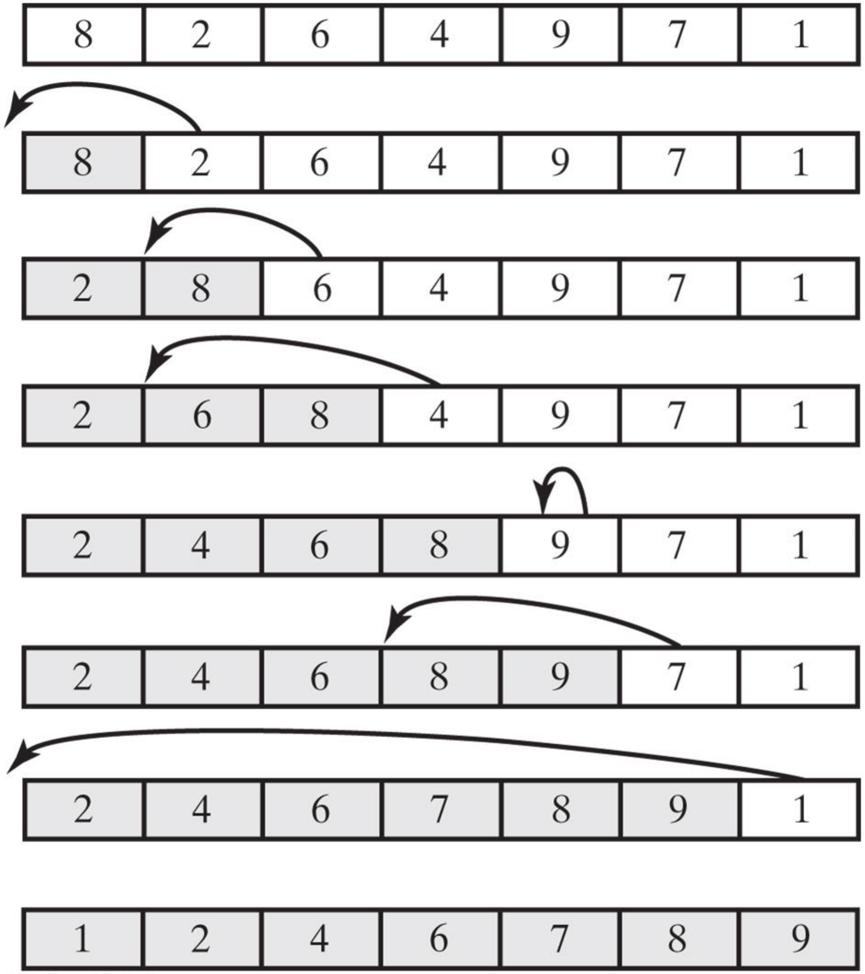
# Insertion Sort

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



© 2019 Pearson Education, Inc.

# Insertion Sort of integers



8 2 6 4 9 7 1

8 2 6 4 9 7 1

2 8 6 4 9 7 1

2 6 8 4 9 7 1

2 4 6 8 9 7 1

2 4 6 8 9 7 1

2 4 6 7 8 9 1

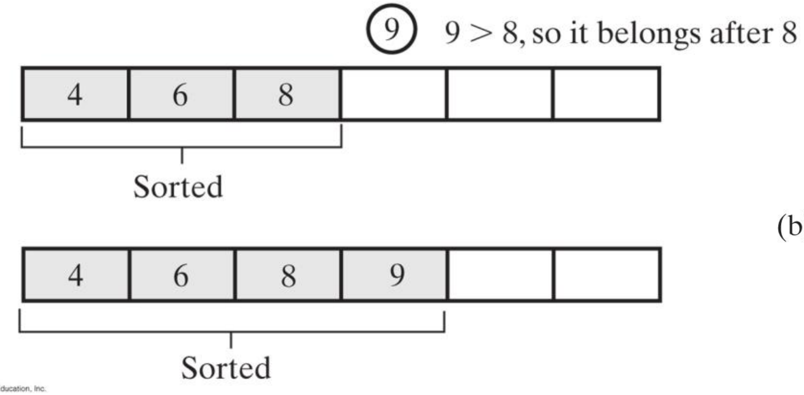
1 2 4 6 7 8 9

© 2019 Pearson Education, Inc.

# Insertion Sort

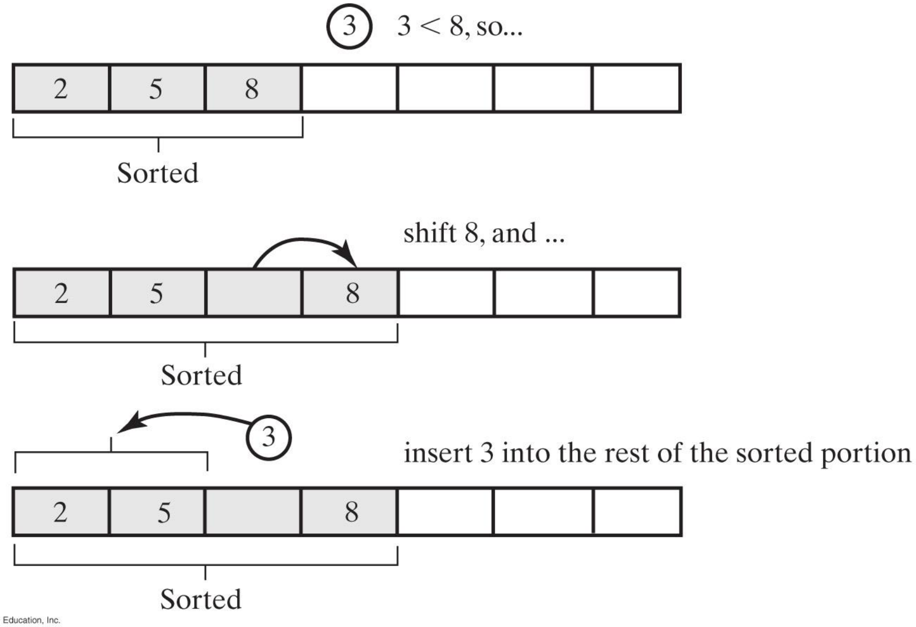
- Inserting the first unsorted entry into the sorted portion of the array

(a) The entry is greater than or equal to the last sorted entry



© 2019 Pearson Education, Inc.

(b) The entry is smaller than the last sorted entry



© 2019 Pearson Education, Inc.

# Iterative Insertion Sort

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

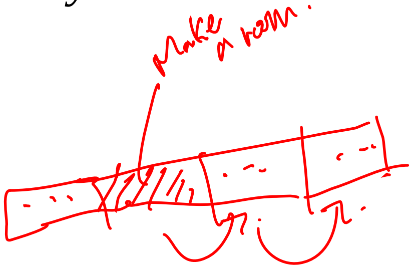
```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.
for (unsorted = first + 1 through last)
{
        nextToInsert = a[unsorted]
        insertInOrder(nextToInsert, a, first, unsorted – 1)
}
```

sorted list

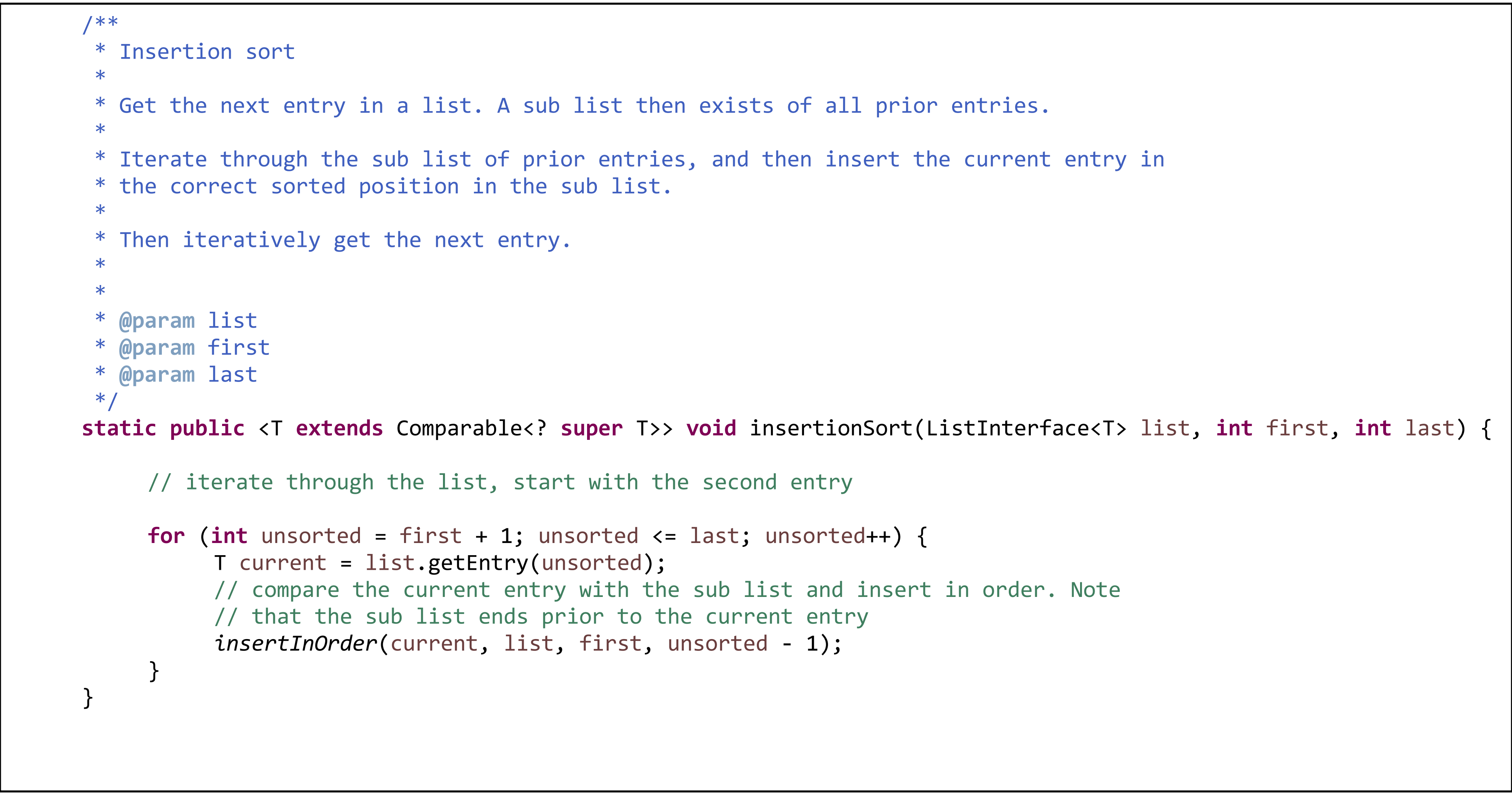
# insertInOrder pseudocode

- performs the insertions.

```
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
```



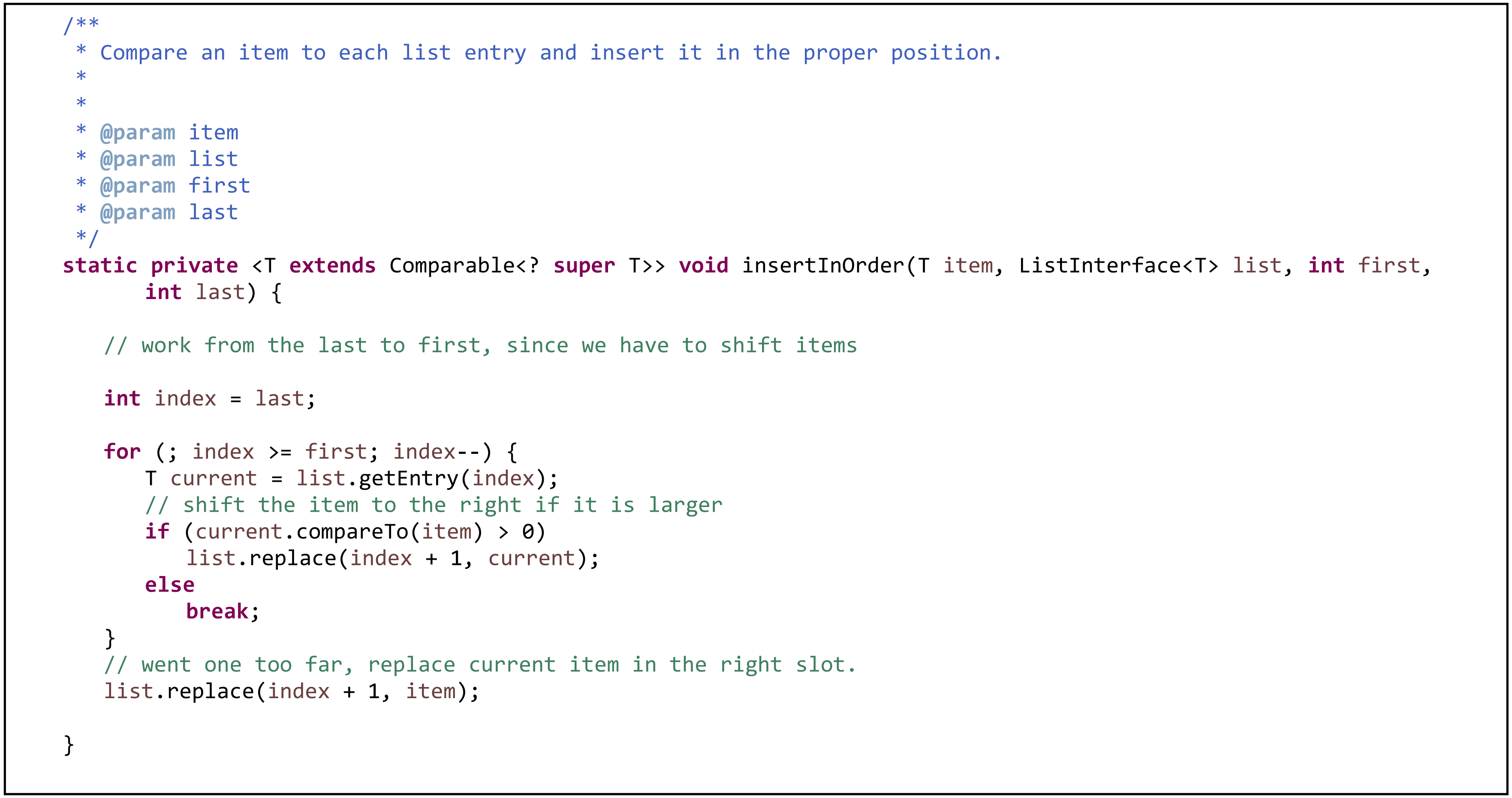
# Insertion Sort - Iterative



```java
/**
 * Insertion sort
 *
 * Get the next entry in a list. A sub list then exists of all prior entries.
 *
 * Iterate through the sub list of prior entries, and then insert the current entry in
 * the correct sorted position in the sub list.
 *
 * Then iteratively get the next entry.
 *
 *
 * @param list
 * @param first
 * @param last
 */
static public <T extends Comparable<? super T>> void insertionSort(ListInterface<T> list, int first, int last) {

     // iterate through the list, start with the second entry

     for (int unsorted = first + 1; unsorted <= last; unsorted++) {
          T current = list.getEntry(unsorted);
          // compare the current entry with the sub list and insert in order. Note
          // that the sub list ends prior to the current entry
          insertInOrder(current, list, first, unsorted - 1);
     }
}
```

# insertInOrder

- If the item is larger, shift it to the right
- Work from the end of the array



```java
/**
 * Compare an item to each list entry and insert it in the proper position.
 *
 *
 * @param item
 * @param list
 * @param first
 * @param last
 */
static private <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);

}
```

# 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
}
```

# Insertion Sort - Recursive

```
    /**
     * 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) {
          recursiveInsertionSort(list, first, last - 1);
          T next = list.getEntry(last);
          insertInOrder(next, list, first, last - 1);
       }
    }
```

Recursive

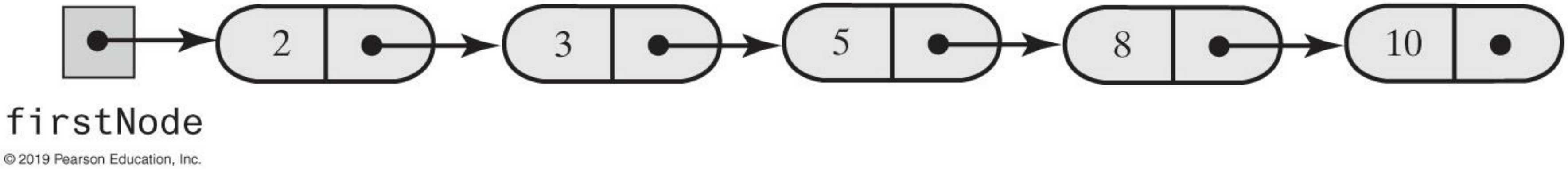
All the way down

and

Back up.

# Insertion Sort with a Linked Chain

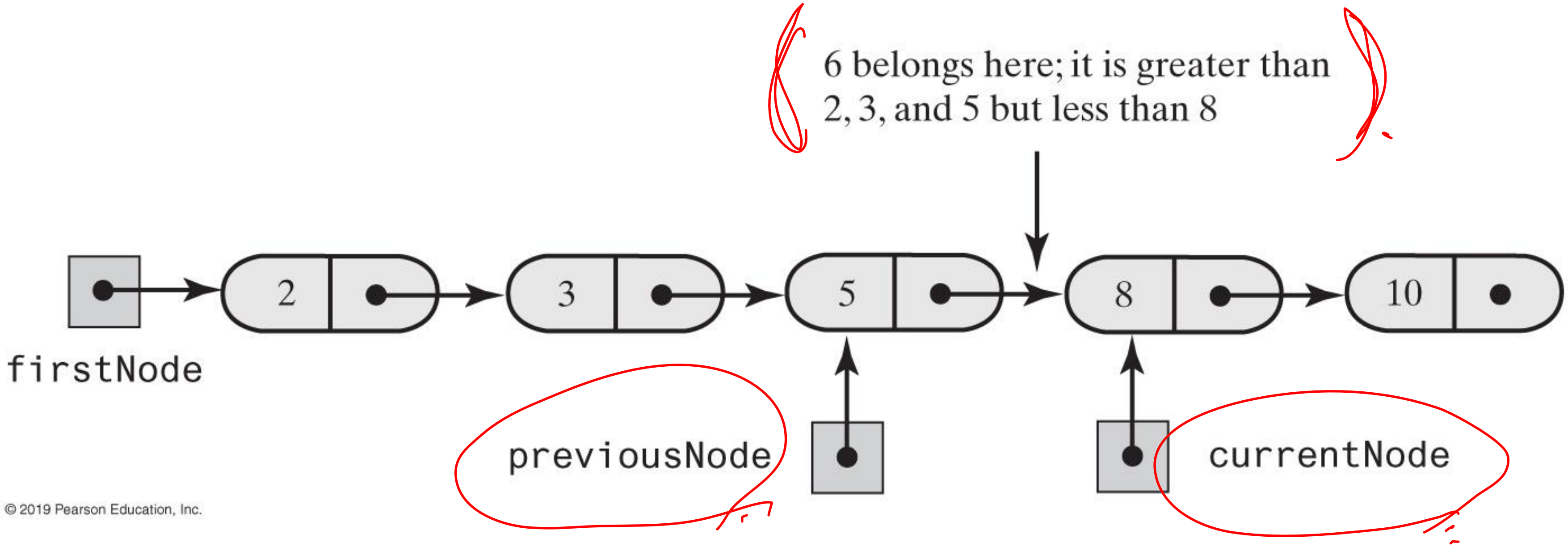
- A chain of integers sorted into ascending order



© 2019 Pearson Education, Inc.

# Insertion Sort with a Linked Chain

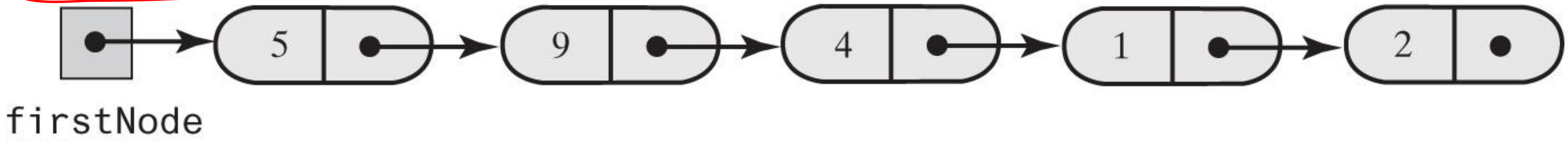
- During the traversal of a chain to locate the insertion point, save a reference to the node before the current one


6 belongs here; it is greater than 2, 3, and 5 but less than 8
2
3
5
8
10
firstNode
previousNode
currentNode

© 2019 Pearson Education, Inc.

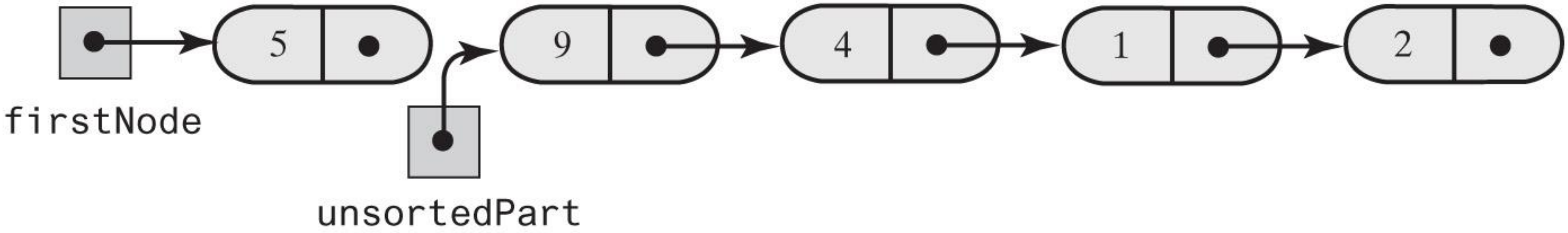
# Insertion Sort with a Linked Chain

(a) The original chain



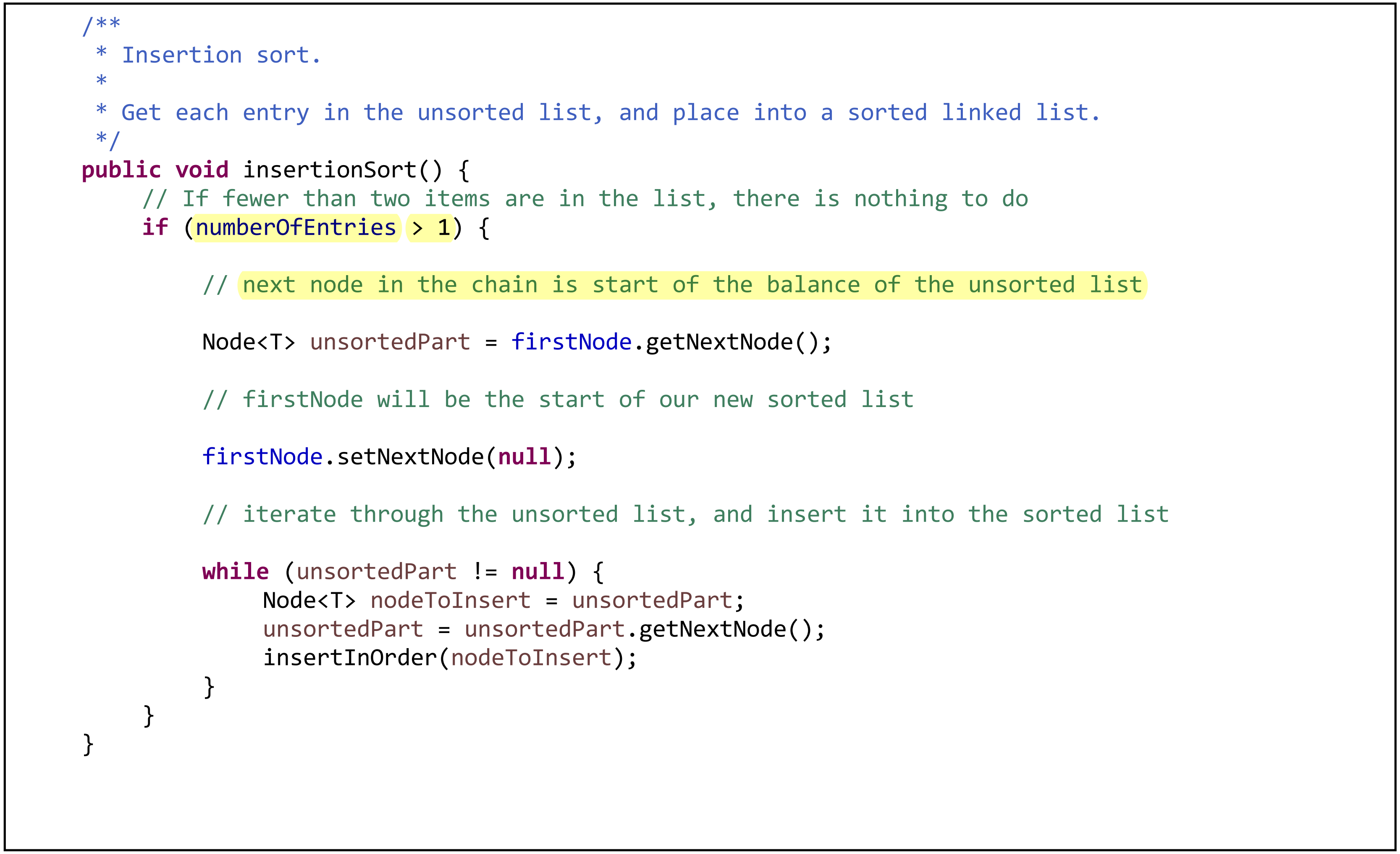
© 2019 Pearson Education, Inc.

(b) The two pieces



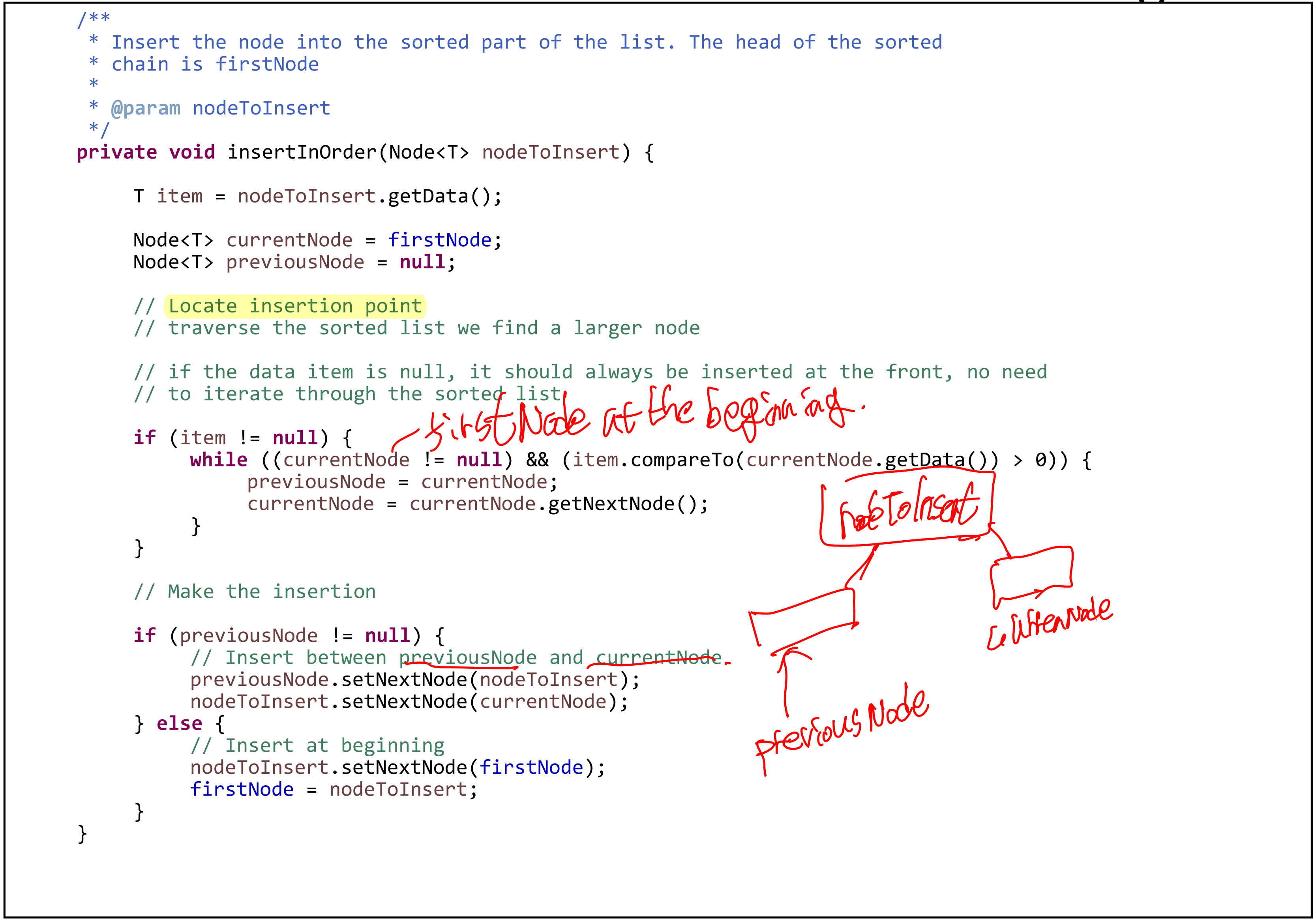
© 2019 Pearson Education, Inc.

# InsertionSortLList – insertionSort()



```
/**
 * Insertion sort.
 *
 * Get each entry in the unsorted list, and place into a sorted linked list.
 */
public void insertionSort() {
    // If fewer than two items are in the list, there is nothing to do
    if (numberOfEntries > 1) {

        // next node in the chain is start of the balance of the unsorted list

        Node<T> unsortedPart = firstNode.getNextNode();

        // firstNode will be the start of our new sorted list

        firstNode.setNextNode(null);

        // iterate through the unsorted list, and insert it into the sorted list

        while (unsortedPart != null) {
            Node<T> nodeToInsert = unsortedPart;
            unsortedPart = unsortedPart.getNextNode();
            insertInOrder(nodeToInsert);
        }
    }
}
```

# InsertionSortLList - insertInOrder()



```
/**
 * Insert the node into the sorted part of the list. The head of the sorted
 * chain is firstNode
 *
 * @param nodeToInsert
 */
private void insertInOrder(Node<T> nodeToInsert) {

     T item = nodeToInsert.getData();

     Node<T> currentNode = firstNode;
     Node<T> previousNode = null;

     // Locate insertion point
     // traverse the sorted list we find a larger node

     // if the data item is null, it should always be inserted at the front, no need
     // to iterate through the sorted list

     if (item != null) {
          while ((currentNode != null) && (item.compareTo(currentNode.getData()) > 0)) {
               previousNode = currentNode;
               currentNode = currentNode.getNextNode();
          }
     }

     // 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;
     }
}
```

firstNode at the beginning.

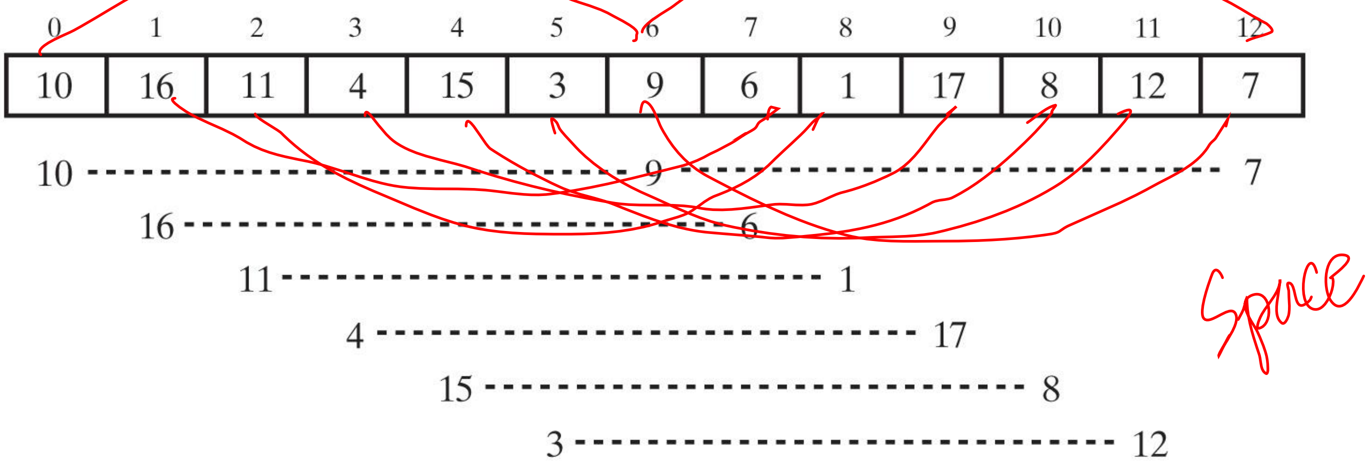
# Shell Sort

- 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

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

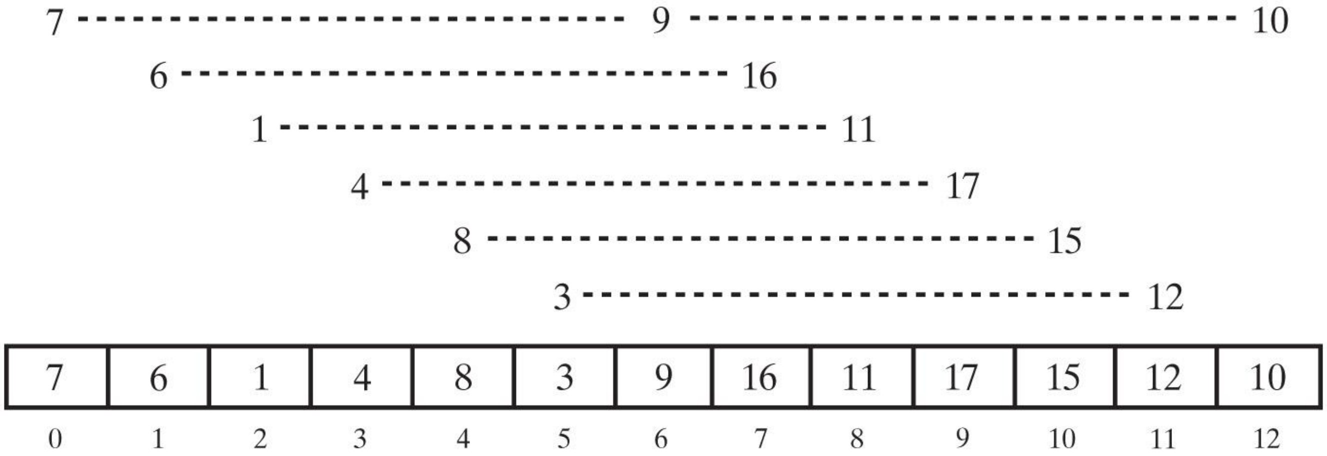
Before Ordering



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 16 | 11 | 4 | 15 | 3 | 9 | 6 | 1 | 17 | 8 | 12 | 7 |

Space:6

After Ordering



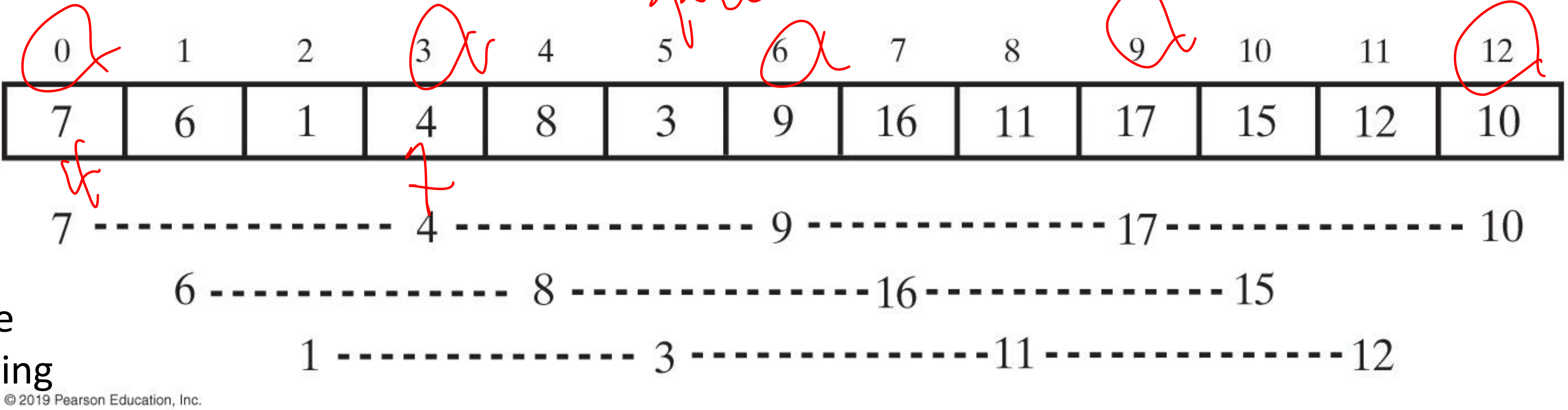
| 7 | 6 | 1 | 4 | 8 | 3 | 9 | 16 | 11 | 17 | 15 | 12 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Shell Sort

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

Space = 3.

Before Ordering



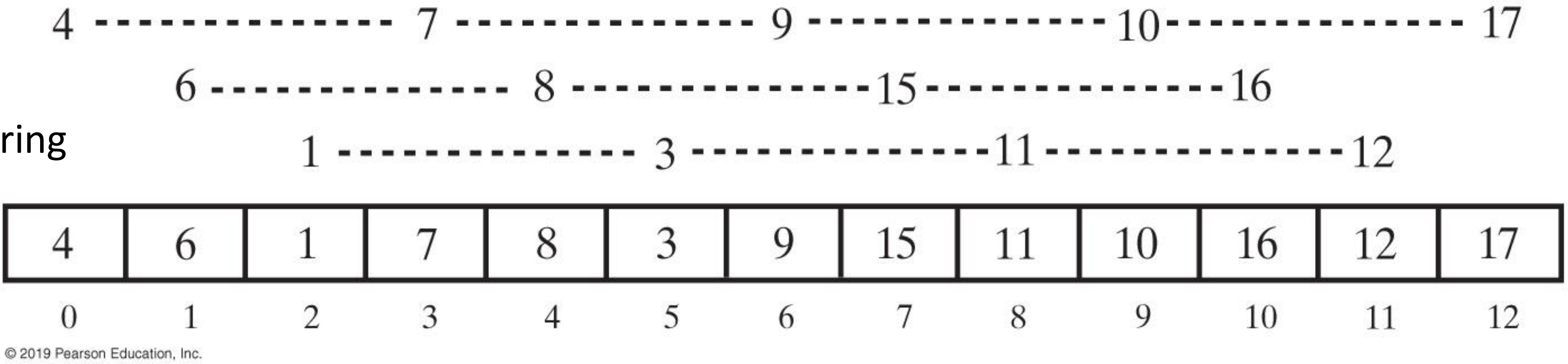
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 1 | 4 | 8 | 3 | 9 | 16 | 11 | 17 | 15 | 12 | 10 |

4 7

```
7 --------------- 4 --------------- 9 --------------- 17 --------------- 10
      6 --------------- 8 --------------- 16 --------------- 15
            1 --------------- 3 --------------- 11 --------------- 12
```

© 2019 Pearson Education, Inc.

After Ordering



```
4 --------------- 7 --------------- 9 --------------- 10 --------------- 17
      6 --------------- 8 --------------- 15 --------------- 16
            1 --------------- 3 --------------- 11 --------------- 12
```

| 4 | 6 | 1 | 7 | 8 | 3 | 9 | 15 | 11 | 10 | 16 | 12 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

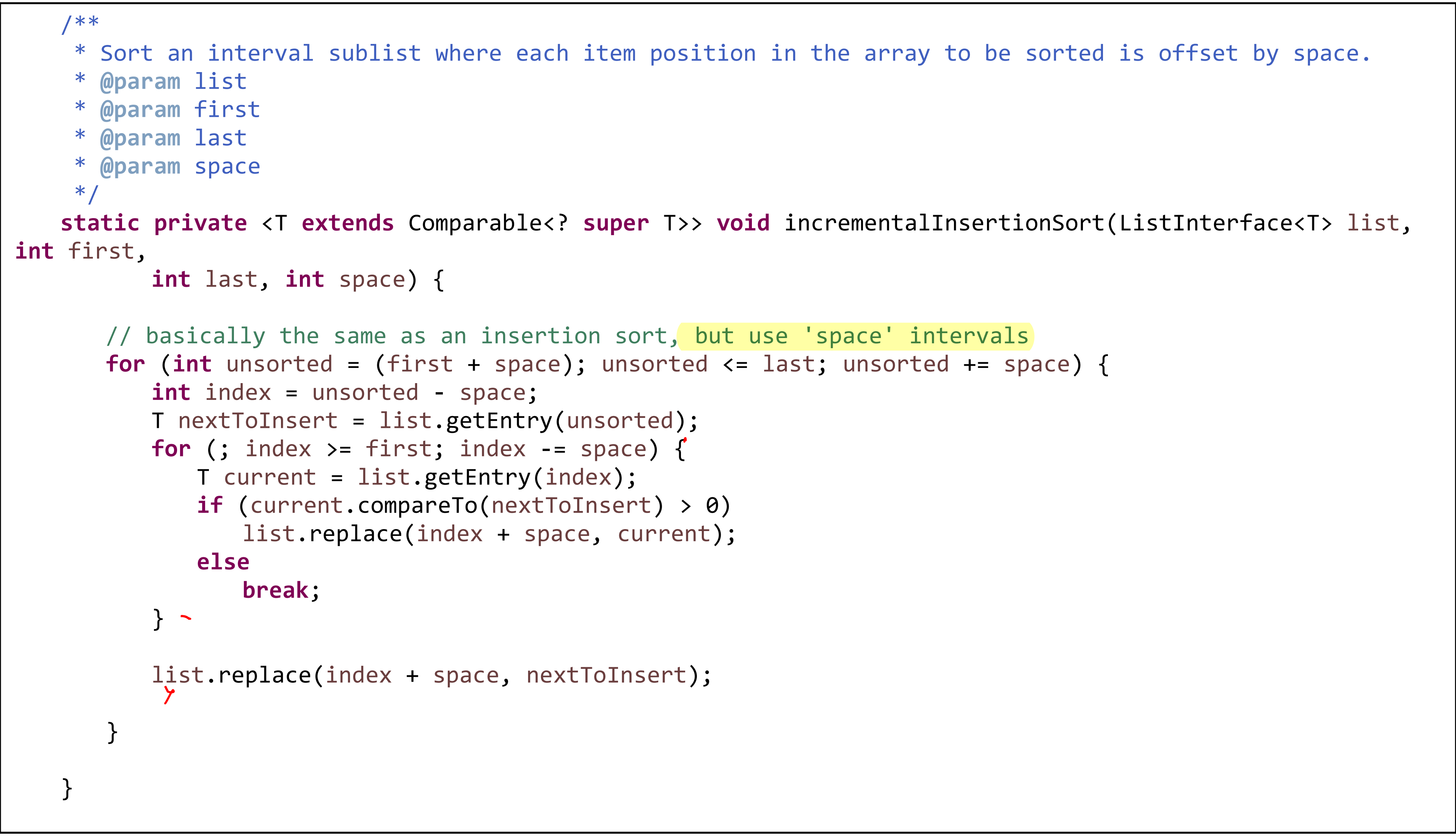
© 2019 Pearson Education, Inc.

- 4 6 1 7 8 3 9 15 11 10 16 12 17
  - ○ 4 1 8 9 11 16 17
    - 1 4 8 9 11 16 17
  - ○ 6 7 3 15 10 12
    - 3 6 7 12 10 15
- 1 3 4 6 8 7 9 12 11 10 16 15 17
- 1 3 4 6 7 8 9 12 10 11 15 16 17

# shellSort()

```
   /**
    * Shell Sort
    *
    * Variation on insertion sort by using a set of intervals and then gradually decreasing the
intervals
    * @param list
    * @param first
    * @param last
    */

   static public <T extends Comparable<? super T>> void shellSort(ListInterface<T> list, int
first, int last) {

      int space = (last - first + 1) / 2; // initial interval is array size divided by 2

      while (space > 0) {
         // move begin up iteratively until end of interval
         for (int begin = first; begin <= (first + space - 1); begin++) {
            incrementalInsertionSort(list, begin, last, space);
         }
         // decrease interval
         space /= 2;
      }
   }
```

# incrementalInsertionSort()



```java
    /**
     * Sort an interval sublist where each item position in the array to be sorted is offset by space.
     * @param list
     * @param first
     * @param last
     * @param space
     */
    static private <T extends Comparable<? super T>> void incrementalInsertionSort(ListInterface<T> list,
int first,
            int last, int space) {

        // basically the same as an insertion sort, but use 'space' intervals
        for (int unsorted = (first + space); unsorted <= last; unsorted += space) {
            int index = unsorted - space;
            T nextToInsert = list.getEntry(unsorted);
            for (; index >= first; index -= space) {
                T current = list.getEntry(index);
                if (current.compareTo(nextToInsert) > 0)
                    list.replace(index + space, current);
                else
                    break;
            }

            list.replace(index + space, nextToInsert);

        }

    }
```

# Comparing Algorithms

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

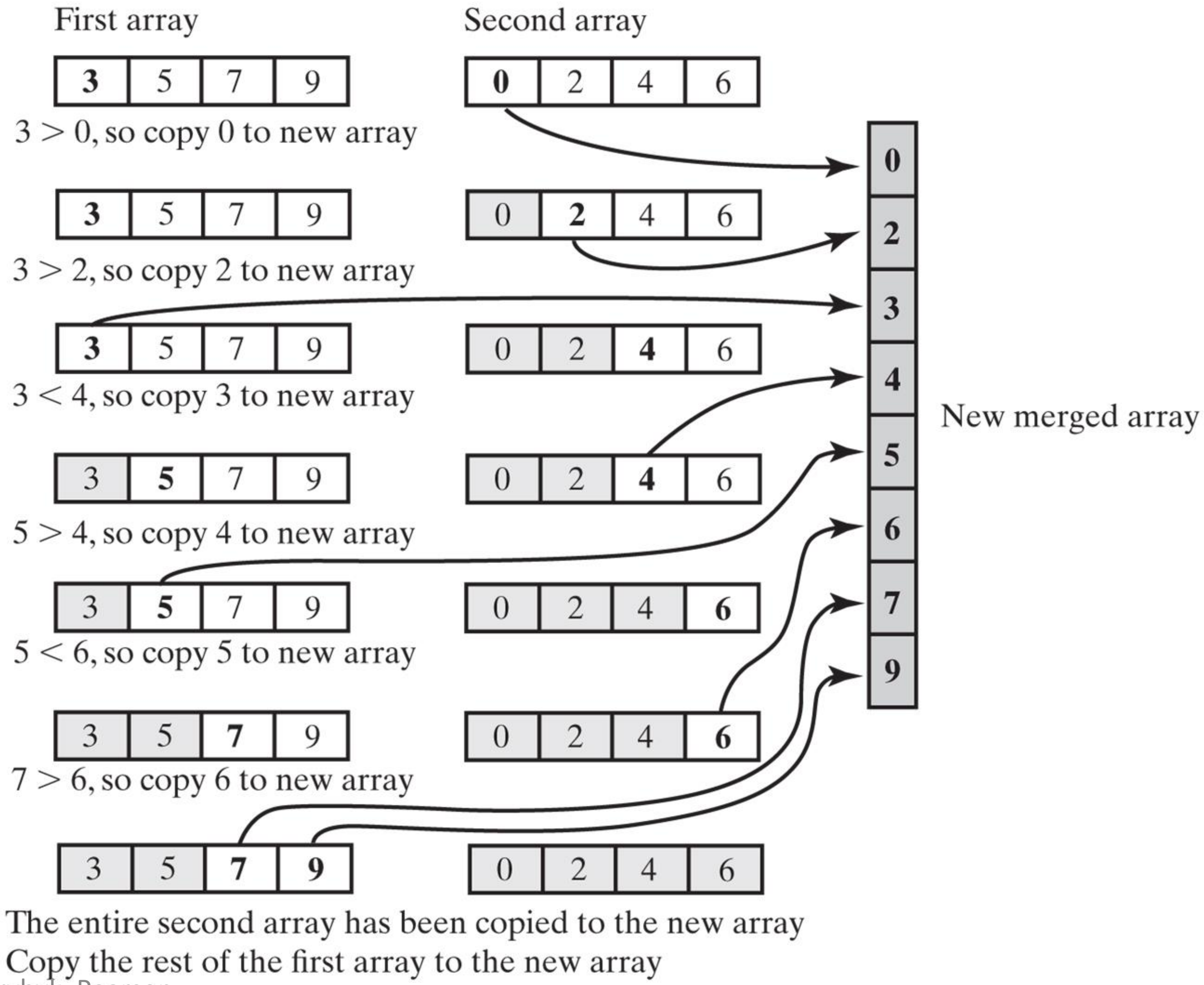
| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Shell Sort | $O(n)$ | $O(n^{1.5})$ | $O(n^{1.5})$ |

# Merge Sort

- Divides an array into halves
- Sorts the two halves,
  - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

# Merge Sort

- Merging two sorted arrays into one sorted array



3 5 7 9 | 0 2 4 6

3 > 0, so copy 0 to new array

3 5 7 9 | 0 2 4 6

3 > 2, so copy 2 to new array

3 5 7 9 | 0 2 4 6

3 < 4, so copy 3 to new array

3 5 7 9 | 0 2 4 6

5 > 4, so copy 4 to new array

3 5 7 9 | 0 2 4 6

5 < 6, so copy 5 to new array

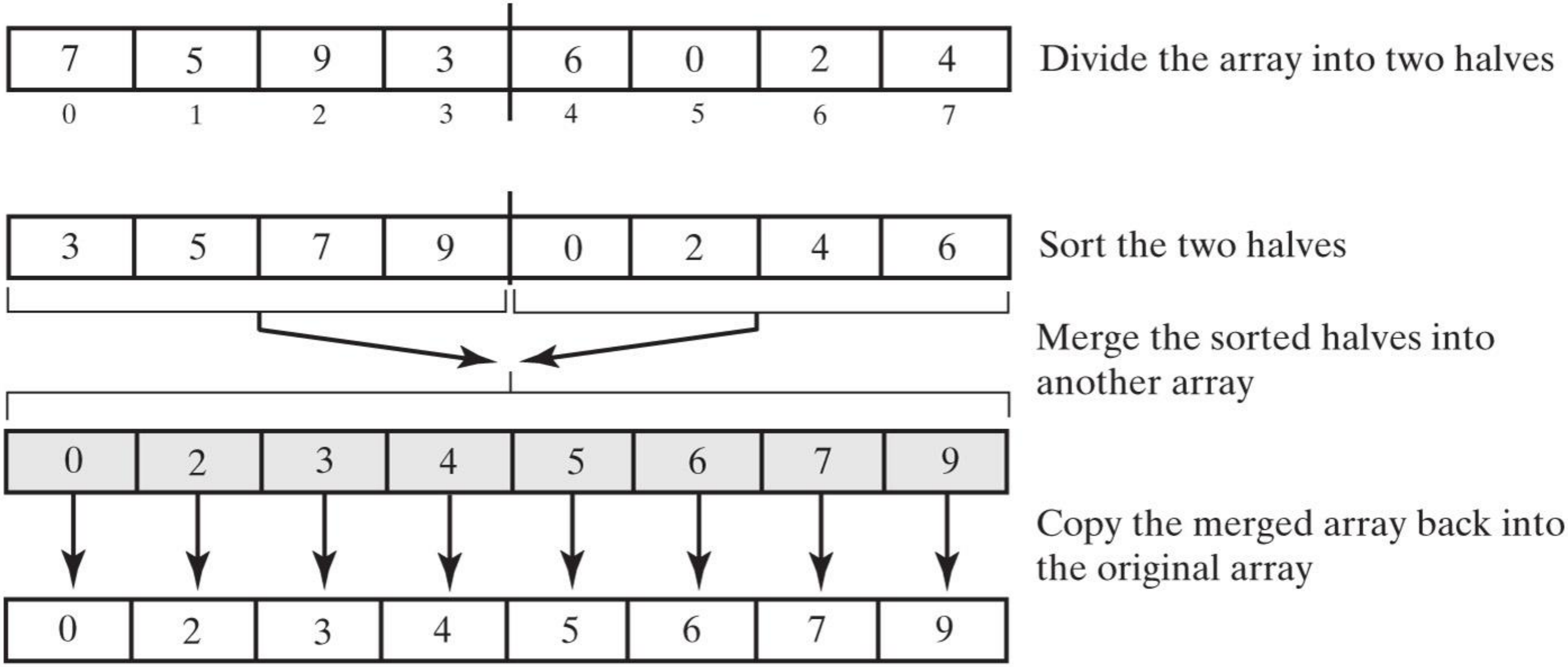
3 5 7 9 | 0 2 4 6

7 > 6, so copy 6 to new array

3 5 7 9 | 0 2 4 6

The entire second array has been copied to the new array
Copy the rest of the first array to the new array

# Merge Sort steps



Divide the array into two halves

Sort the two halves

Merge the sorted halves into another array

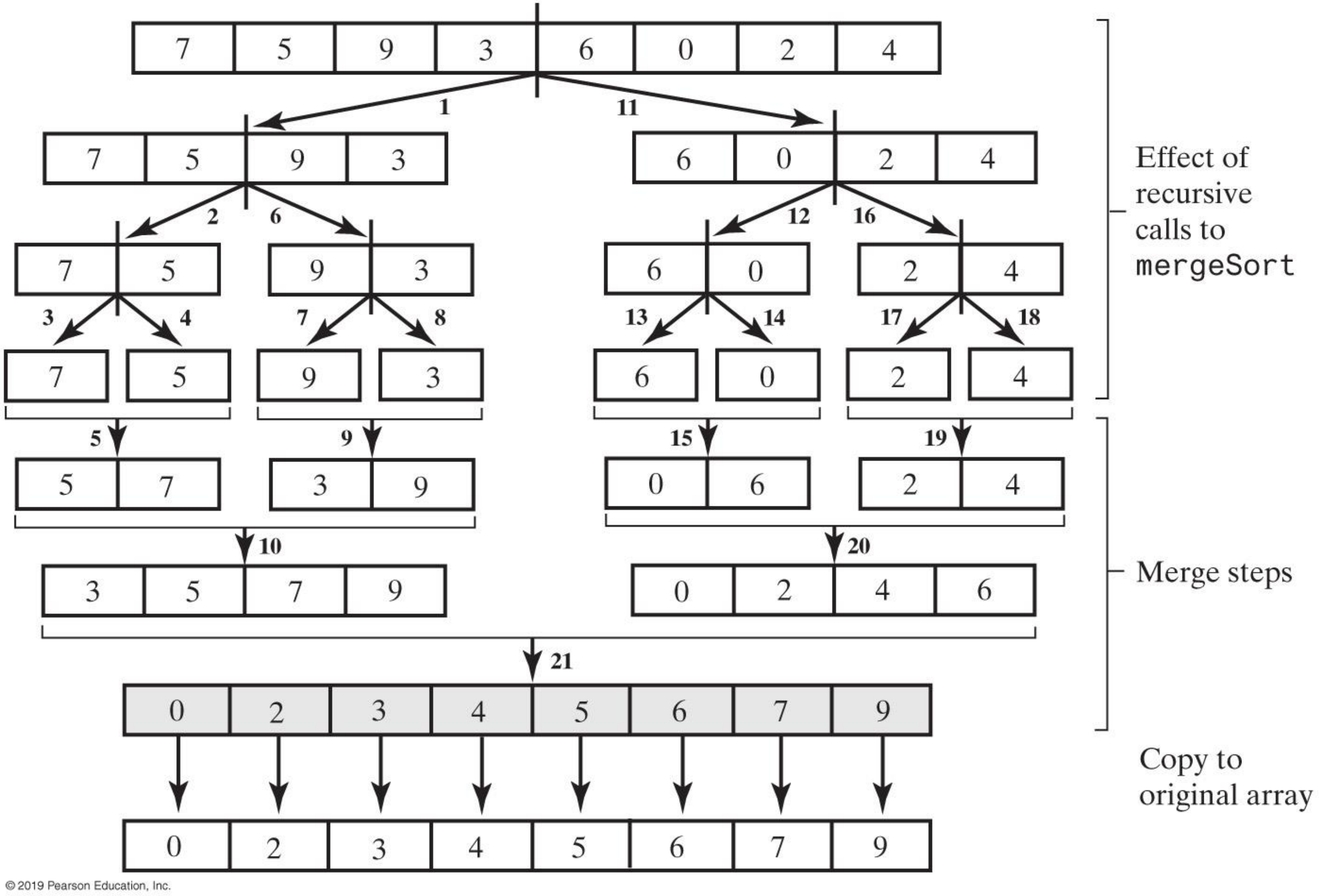
Copy the merged array back into the original array

# Recursive Merge Sort

- Recursive algorithm for merge sort.

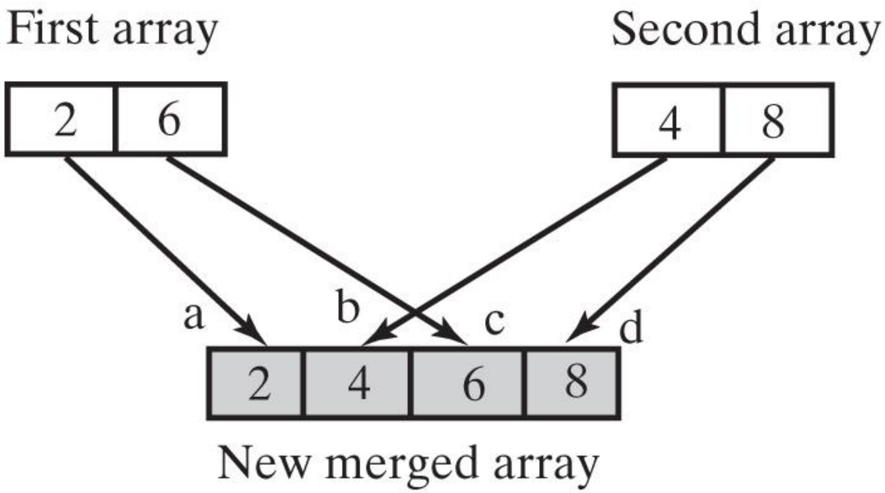
```
Algorithm  mergeSort(a,  tempArray,  first, last)
// Sorts the array entries a[first..last] recursively.
if (first < last)
{
      mid = approximate midpoint between first and last
      mergeSort(a, tempArray,  first,  mid)
      mergeSort(a, tempArray, mid + 1, last)
      Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

# Merge Sort – recursive calls and merges



© 2019 Pearson Education, Inc.

# A worst-case merge of two sorted arrays



© 2019 Pearson Education, Inc.

a. 2 < 4, so copy 2 to new array

b. 6 > 4, so copy 4 to new array

c. 6 < 8, so copy 6 to new array

d. Copy 8 to new array

**Efficiency is O(*n* log *n*)**

# Iterative Merge Sort

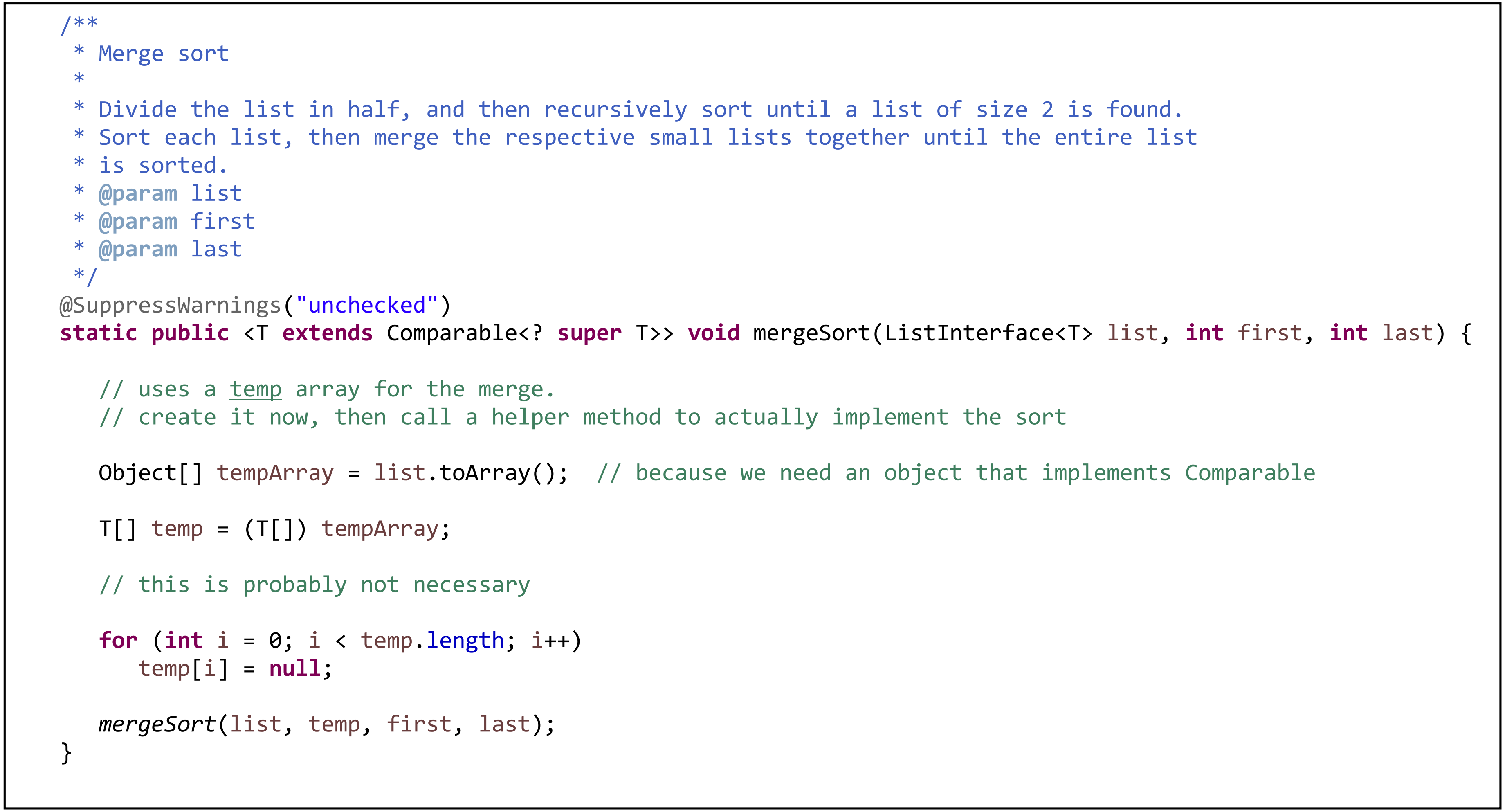
- Less simple than recursive version.
  - Need to control the merges.
- Will be more efficient of both time and space.
  - But trickier to code without error.

# Iterative Merge Sort

- Starts at beginning of array
    - Merges pairs of individual entries to form two-entry subarrays
- Returns to the beginning of array and merges pairs of the two-entry subarrays to form four-entry subarrays
    - And so on
- After merging all pairs of subarrays of a particular length, might have entries left over.

# mergeSort()

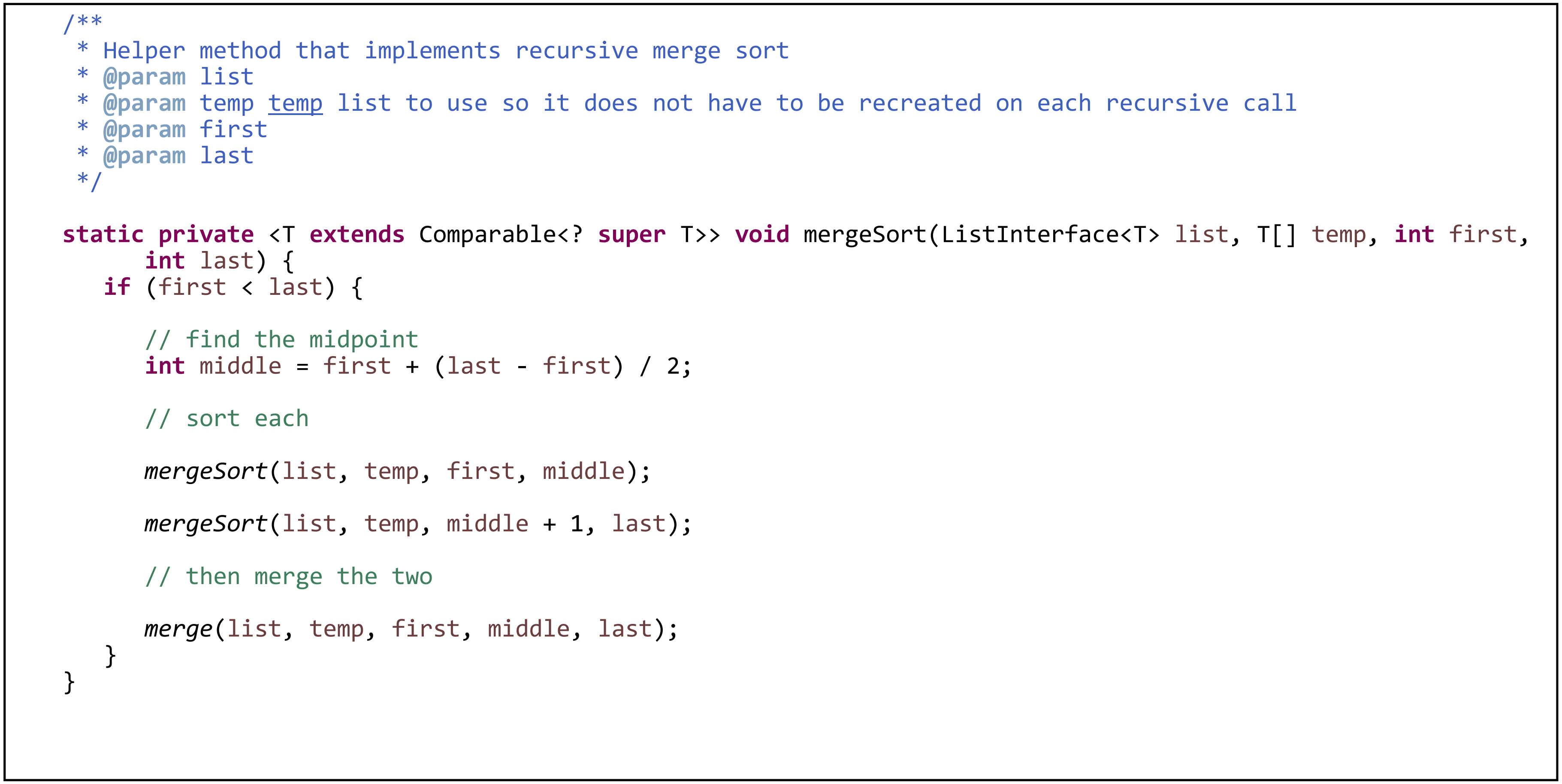
- Create temp array to use so no need to recreate each time merge is called



```
/**
 * Merge sort
 *
 * Divide the list in half, and then recursively sort until a list of size 2 is found.
 * Sort each list, then merge the respective small lists together until the entire list
 * is sorted.
 * @param list
 * @param first
 * @param last
 */
@SuppressWarnings("unchecked")
static public <T extends Comparable<? super T>> void mergeSort(ListInterface<T> list, int first, int last) {

   // uses a temp array for the merge.
   // create it now, then call a helper method to actually implement the sort

   Object[] tempArray = list.toArray();  // because we need an object that implements Comparable

   T[] temp = (T[]) tempArray;

   // this is probably not necessary

   for (int i = 0; i < temp.length; i++)
      temp[i] = null;

   mergeSort(list, temp, first, last);
}
```

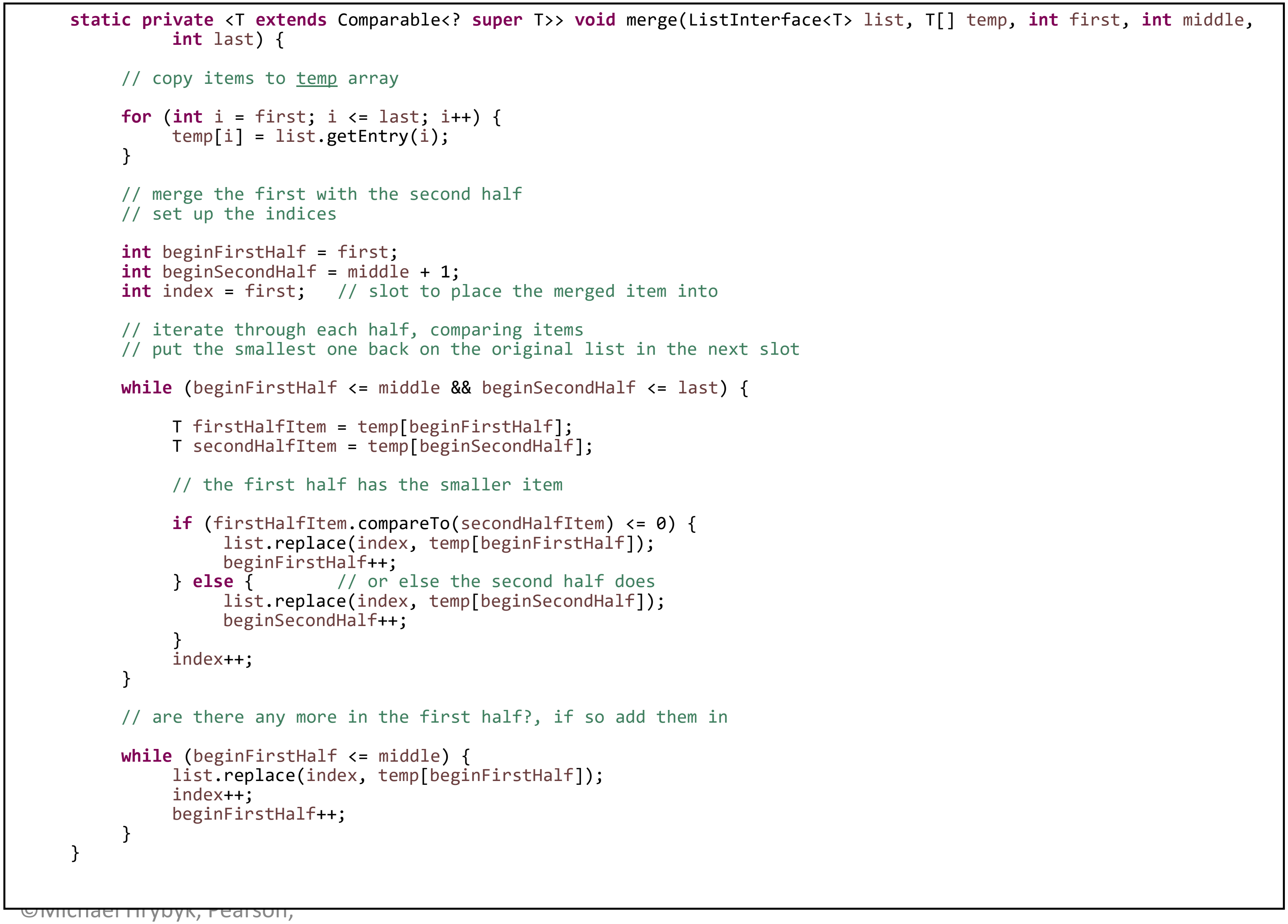
# mergeSort() – recursive calls

- sort each half, then merge the two



```
/**
 * Helper method that implements recursive merge sort
 * @param list
 * @param temp temp list to use so it does not have to be recreated on each recursive call
 * @param first
 * @param last
 */

static private <T extends Comparable<? super T>> void mergeSort(ListInterface<T> list, T[] temp, int first,
      int last) {
   if (first < last) {

      // find the midpoint
      int middle = first + (last - first) / 2;

      // sort each

      mergeSort(list, temp, first, middle);

      mergeSort(list, temp, middle + 1, last);

      // then merge the two

      merge(list, temp, first, middle, last);
   }
}
```

# merge() – interleave two arrays



```java
static private <T extends Comparable<? super T>> void merge(ListInterface<T> list, T[] temp, int first, int middle,
        int last) {

    // copy items to temp array

    for (int i = first; i <= last; i++) {
        temp[i] = list.getEntry(i);
    }

    // merge the first with the second half
    // set up the indices

    int beginFirstHalf = first;
    int beginSecondHalf = middle + 1;
    int index = first;   // slot to place the merged item into

    // iterate through each half, comparing items
    // put the smallest one back on the original list in the next slot

    while (beginFirstHalf <= middle && beginSecondHalf <= last) {

        T firstHalfItem = temp[beginFirstHalf];
        T secondHalfItem = temp[beginSecondHalf];

        // the first half has the smaller item

        if (firstHalfItem.compareTo(secondHalfItem) <= 0) {
            list.replace(index, temp[beginFirstHalf]);
            beginFirstHalf++;
        } else {        // or else the second half does
            list.replace(index, temp[beginSecondHalf]);
            beginSecondHalf++;
        }
        index++;
    }

    // are there any more in the first half?, if so add them in

    while (beginFirstHalf <= middle) {
        list.replace(index, temp[beginFirstHalf]);
        index++;
        beginFirstHalf++;
    }
}
```

# MergeSort in the Java Class Library

- Class **`Arrays`** in the package **`java.util`** defines versions of a static method sort
- All array items must implement Comparable
- The implementation is a stable, adaptive, iterative mergesort that requires far fewer than nlog(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered.
- If the input array is nearly sorted, the implementation requires approximately n comparisons.
- Uses QuickSort when array consists of primitives (int, char, …)

```
public static void sort(Object[] a)

public static void sort(Object[] a, int first, int after)
```

# sortUsingLibrary() implementation

```
    /**
     * sort using the Java Library method.
     *
     * Uses merge sort
     *
     * @param list
     * @param first
     * @param last
     */
    static public <T extends Comparable<? super T>> void sortUsingLibrary(ListInterface<T> list, int first,
int last) {

        // get the array

        Object[] tempArray = list.toArray();

        @SuppressWarnings("unchecked")
        T[] listCopy = (T[]) tempArray;

        // uses merge sort. hover over to see implementation notes
        // note that the third arg is toIndex exclusive, which effectively means the next index
        //   or for the entire array, array size
        Arrays.sort(listCopy, first, last + 1);

        // now copy it back
        list.clear();
        for(T item : listCopy)
            list.add(item);
    }
```

# Quick Sort

- Divides an array into two pieces
    - Pieces are not necessarily halves of the array
    - Chooses one entry in the array—called the pivot
- Partitions the array

# Quick Sort

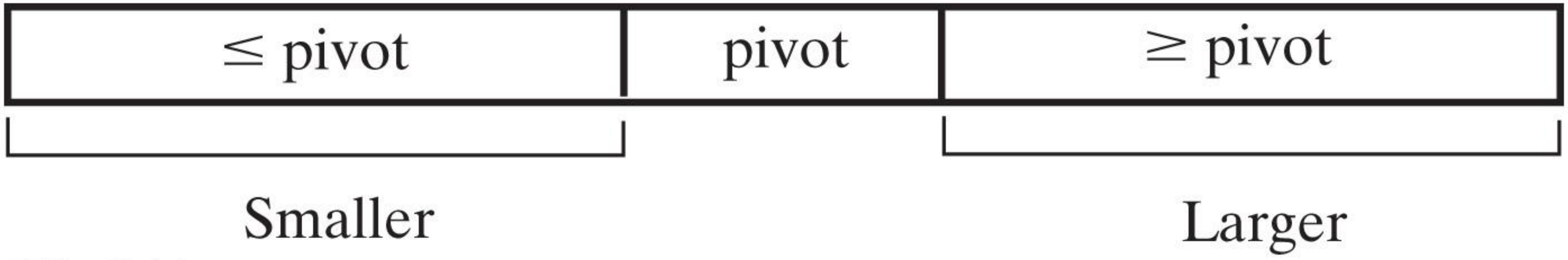
- When pivot chosen, array rearranged such that:
  - Pivot is in position that it will occupy in final sorted array
  - Entries in positions before pivot are less than or equal to pivot
  - Entries in positions after pivot are greater than or equal to pivot

# Quick Sort

- Algorithm that describes our sorting strategy

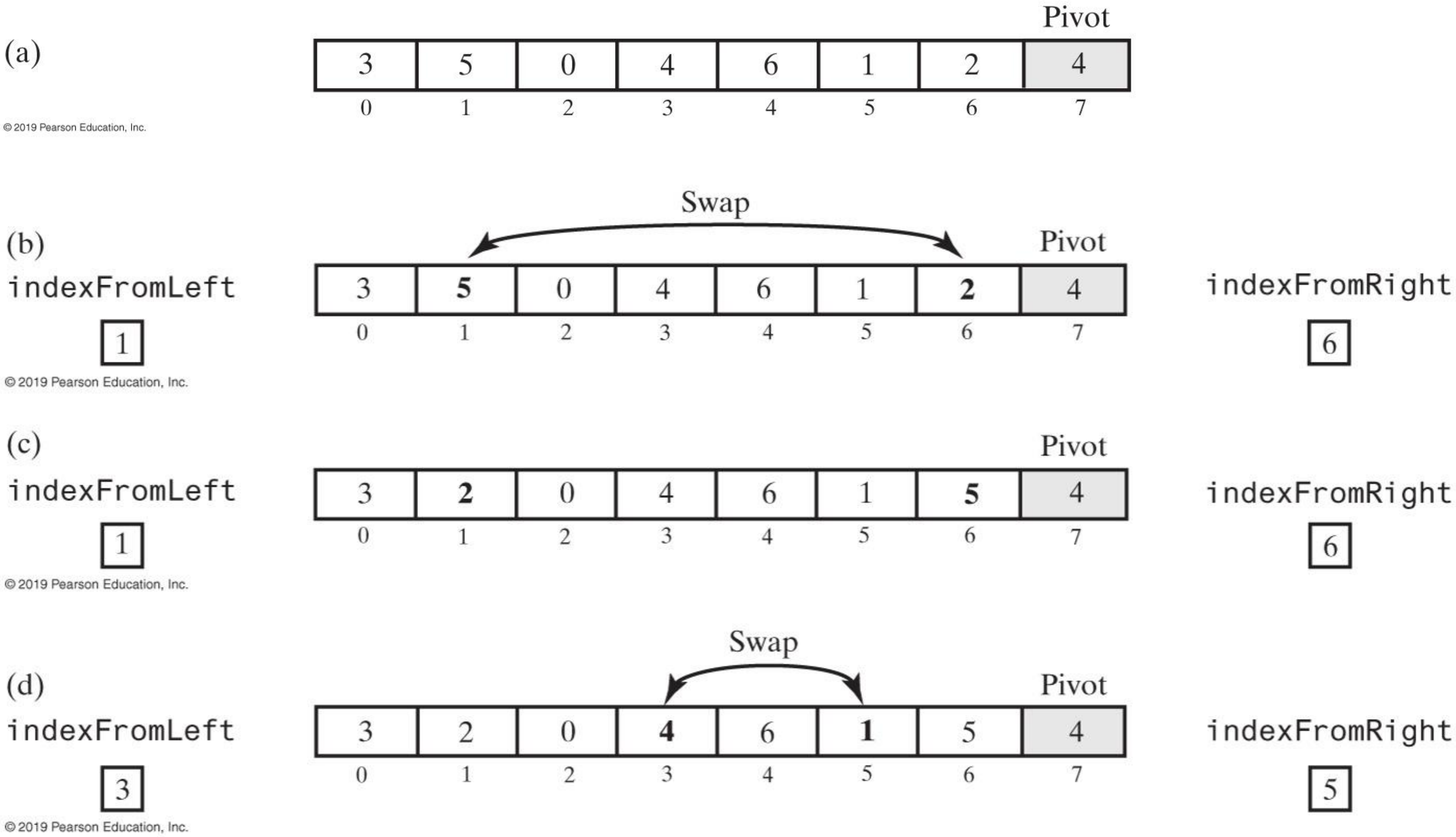
```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first..last] recursively.
if (first < last)
{
        Choose a pivot
        Partition the array about the pivot
        pivotIndex = index of pivot
        quickSort(a, first, pivotIndex − 1) // Sort Smaller
        quickSort(a, pivotIndex + 1, last) //Sort Larger
}
```

# A partition of an array during a quick sort



© 2019 Pearson Education, Inc.

# Quick Sort Partitioning – last element as pivot



(a)

| 3 | 5 | 0 | 4 | 6 | 1 | 2 | 4 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

(b)

| 3 | **5** | 0 | 4 | 6 | 1 | **2** | 4 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

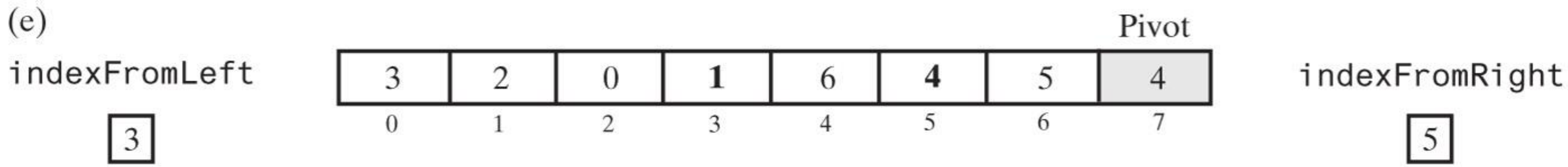
(c)

| 3 | **2** | 0 | 4 | 6 | 1 | **5** | 4 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

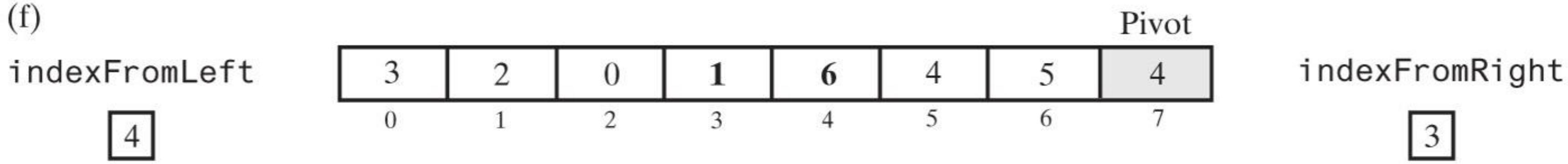
(d)

| 3 | 2 | 0 | **4** | 6 | **1** | 5 | 4 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

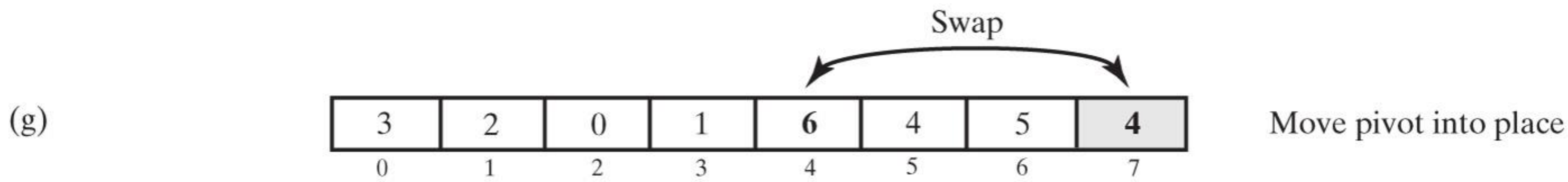
# Quick Sort Partitioning



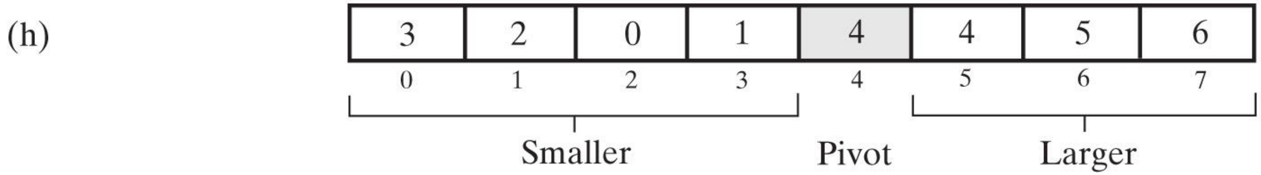
© 2019 Pearson Education, Inc.



© 2019 Pearson Education, Inc.



© 2019 Pearson Education, Inc.



© 2019 Pearson Education, Inc.

# QuickSort

- 8 3 9 4 1 7 2
- **8** 3 9 4 **1** 7 2
- 1 **3** 9 4 8 7 2
- 1 2 9 4 8 7 3
- 1 2 9 4 8 7 3
- 1 2 3 4 8 7 9
- 1 2 3 4 7 8 9
- 3 + 1 + 4 + 2 = 10 compares

- Divide and Conquer!

# QuickSort

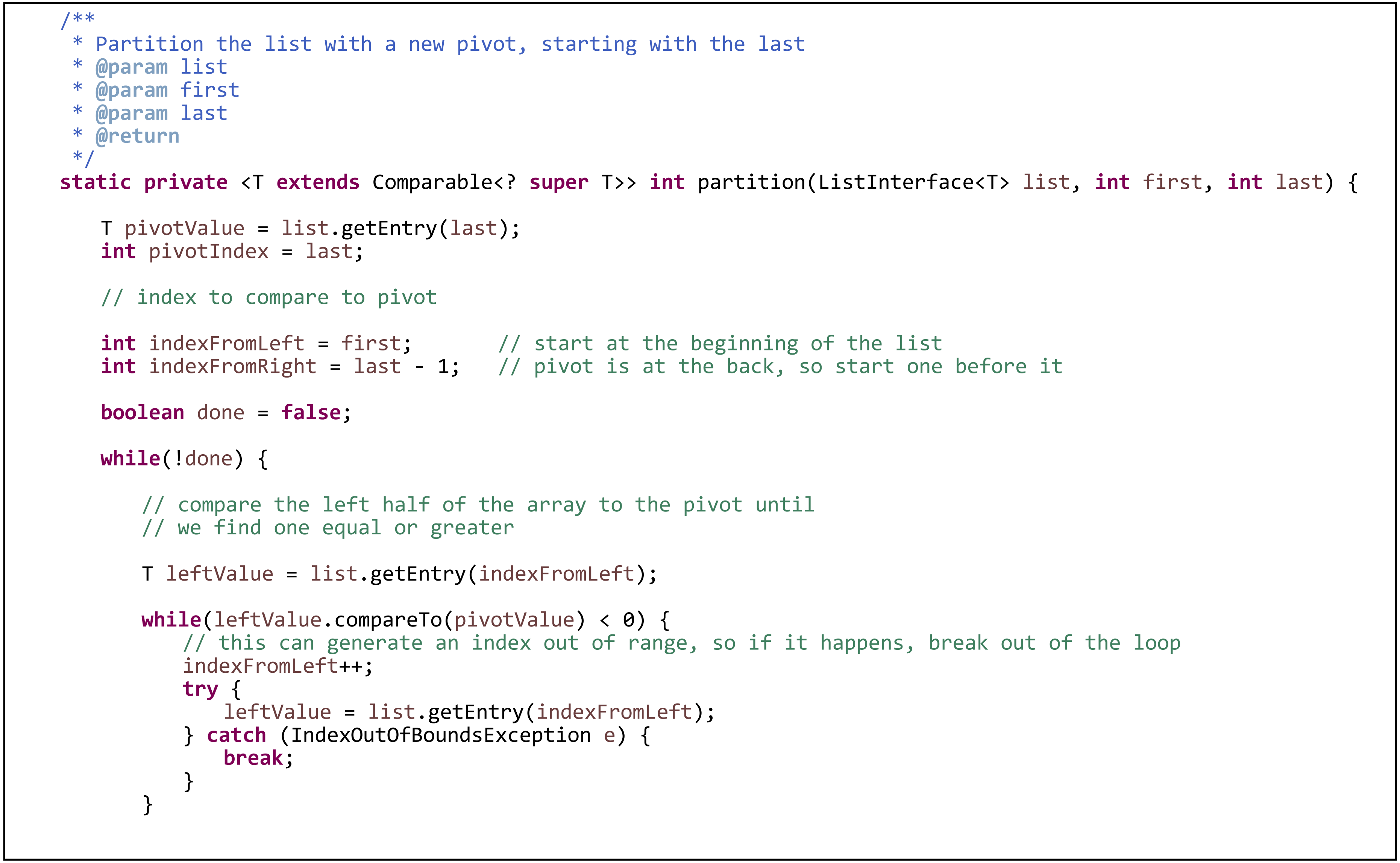
- 1 2 3 4 5 6 7 8
- 1 2 3 4 5 6 7 8
- 1 2 3 4 5 6 7 8
- (7 * 8)/2 = 28 compares

- 8 7 6 5 4 3 2 1
- 1 7 6 5 4 3 2 8
- 1 7 6 5 4 3 2 8
- 1 2 6 5 4 3 7 8
- 1 2 6 5 4 3 7 8
- 1 2 3 5 4 6 7 8
- 1 2 3 5 4 6 7 8
- 1 2 3 4 5 6 7 8
- (7 * 8)/2 = 28 compares

# QuickSort implementation

```
   /**
    * Quick sort
    *
    * Sort using a pivot value. In this implementation, we use the last element as the initial pivot
    * @param list
    * @param first
    * @param last
    */
   static public <T extends Comparable<? super T>> void quickSort(ListInterface<T> list, int first,
int last) {
      if (first < last) {
         int pivotIndex = partition(list, first, last);
         // sort around the pivot index, splitting the list
         quickSort(list, first, pivotIndex - 1);
         quickSort(list, pivotIndex + 1, last);
      }

   }
```

# QuickSort – partition()



```
/**
 * Partition the list with a new pivot, starting with the last
 * @param list
 * @param first
 * @param last
 * @return
 */
static private <T extends Comparable<? super T>> int partition(ListInterface<T> list, int first, int last) {

    T pivotValue = list.getEntry(last);
    int pivotIndex = last;

    // index to compare to pivot

    int indexFromLeft = first;       // start at the beginning of the list
    int indexFromRight = last - 1;   // pivot is at the back, so start one before it

    boolean done = false;

    while(!done) {

        // compare the left half of the array to the pivot until
        // we find one equal or greater

        T leftValue = list.getEntry(indexFromLeft);

        while(leftValue.compareTo(pivotValue) < 0) {
            // this can generate an index out of range, so if it happens, break out of the loop
            indexFromLeft++;
            try {
                leftValue = list.getEntry(indexFromLeft);
            } catch (IndexOutOfBoundsException e) {
                break;
            }
        }
```

# QuickSort – partition()

```
            // do the same for right half, but find one less than or equal to

            T rightValue = list.getEntry(indexFromRight);

            while(rightValue.compareTo(pivotValue) > 0) {
                // this can generate an index out of range, so if it happens, break out of the loop
                indexFromRight--;
                try {
                    rightValue = list.getEntry(indexFromRight);
                } catch (IndexOutOfBoundsException e) {
                    break;
                }
            }

            // we now have one that is greater on the left, and less on the right
            // if the indices aren't equal, swap the values

            if(indexFromLeft < indexFromRight) {
                swap(list, indexFromLeft, indexFromRight);
                indexFromLeft++;
                indexFromRight--;
            }
            else done = true;
        }

        // finally, the left index value is the largest, so swap with pivot

        swap(list, pivotIndex, indexFromLeft);
        pivotIndex = indexFromLeft;

        return pivotIndex;
    }
```

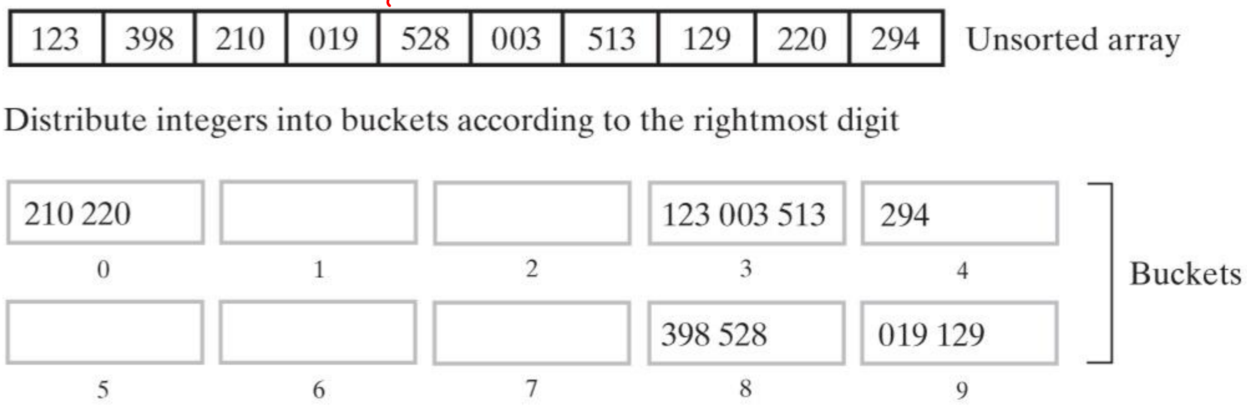
# Radix Sort

- Does not use comparison
- Treats array entries as if they were strings that have the same length.
  - Group integers according to their rightmost character (digit) into “buckets”
  - Repeat with next character (digit), etc.

# Radix Sort

works for only.
integer.

(a) Distribution of the original array into buckets



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 123 | 398 | 210 | 019 | 528 | 003 | 513 | 129 | 220 | 294 | Unsorted array |

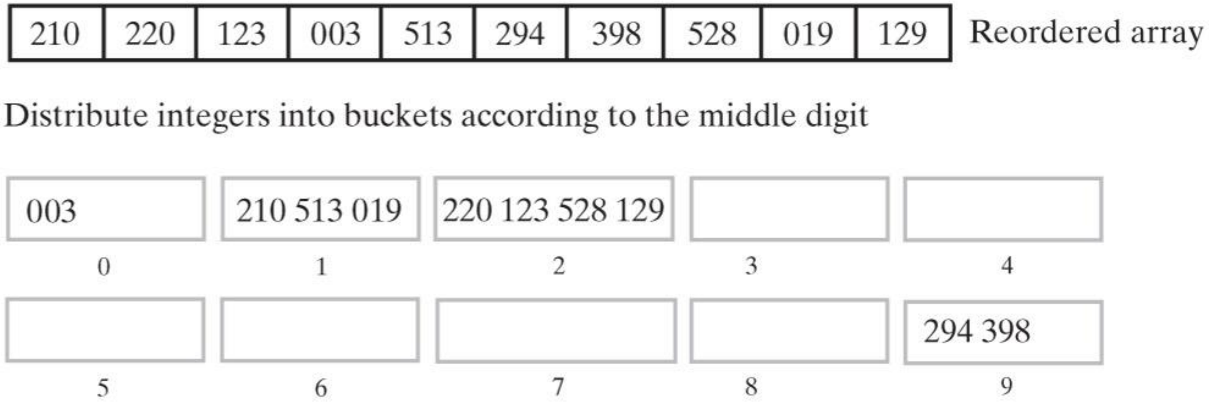
Distribute integers into buckets according to the rightmost digit

Buckets

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 210 220 | | | 123 003 513 | 294 |

| 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| | | | 398 528 | 019 129 |

(b) Distribution of the reordered array into buckets



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 210 | 220 | 123 | 003 | 513 | 294 | 398 | 528 | 019 | 129 | Reordered array |

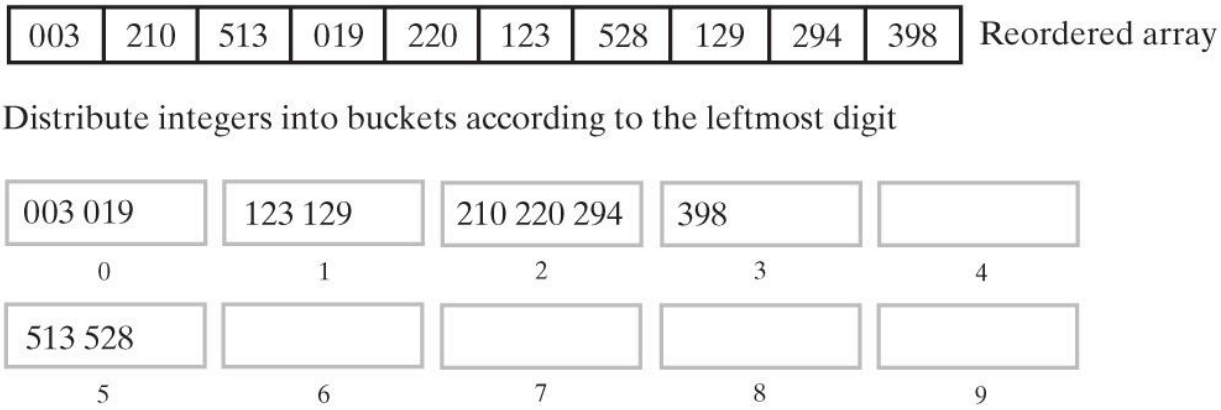
Distribute integers into buckets according to the middle digit

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 003 | 210 513 019 | 220 123 528 129 | | |

| 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| | | | | 294 398 |

# Radix Sort

(c) Distribution of the reordered array into buckets

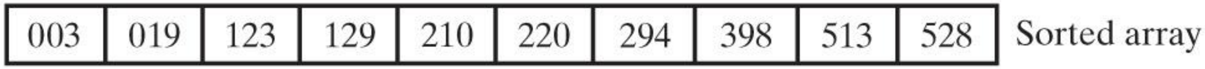


|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 003 | 210 | 513 | 019 | 220 | 123 | 528 | 129 | 294 | 398 | Reordered array |

Distribute integers into buckets according to the leftmost digit

|  |  |  |  |  |
|---|---|---|---|---|
| 003 019 | 123 129 | 210 220 294 | 398 |  |
| 0 | 1 | 2 | 3 | 4 |
| 513 528 |  |  |  |  |
| 5 | 6 | 7 | 8 | 9 |

(d) Sorting is complete



|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 003 | 019 | 123 | 129 | 210 | 220 | 294 | 398 | 513 | 528 | Sorted array |

# Algorithm Comparison

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Radix Sort | $O(n)$ | $O(n)$ | $O(n)$ |
| Merge Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Quick Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ |
| Shell Sort | $O(n)$ | $O(n^{1.5})$ | $O(n^{1.5})$ |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |

# Comparing Function Growth Rates

- A comparison of growth-rate functions as n increases

| | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|---|
| ***n*** | 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
| ***n* log *n*** | 33 | 664 | 9,966 | 132,877 | 1,660,964 | 19,931,569 |
| $n^{1.5}$ | 32 | 1,000 | 31,623 | 1,000,000 | 319,622,777 | 109 |
| $n^2$ | 100 | 10,000 | 1,000,000 | 108 | 1,010 | 1,012 |

# GrowthRateFunctionsDemo

```
N                   2, N^1.5                   2, N^2                   4, NlogN                   2
N                   4, N^1.5                   8, N^2                  16, NlogN                   8
N                   8, N^1.5                  22, N^2                  64, NlogN                  24
N                  16, N^1.5                  64, N^2                 256, NlogN                  64
N                  32, N^1.5                 181, N^2                1024, NlogN                 160
N                  64, N^1.5                 512, N^2                4096, NlogN                 384
N                 128, N^1.5                1448, N^2               16384, NlogN                 896
N                 256, N^1.5                4096, N^2               65536, NlogN                2048
N                 512, N^1.5               11585, N^2              262144, NlogN                4608
N                1024, N^1.5               32768, N^2             1048576, NlogN               10240
N                2048, N^1.5               92681, N^2             4194304, NlogN               22528
N                4096, N^1.5              262144, N^2            16777216, NlogN               49152
N                8192, N^1.5              741455, N^2            67108864, NlogN              106496
N               16384, N^1.5             2097152, N^2           268435456, NlogN              229376
N               32768, N^1.5             5931641, N^2          1073741824, NlogN              491520
N               65536, N^1.5            16777216, N^2          4294967296, NlogN             1048576
N              131072, N^1.5            47453132, N^2         17179869184, NlogN             2228224
N              262144, N^1.5           134217728, N^2         68719476736, NlogN             4718592
N              524288, N^1.5           379625062, N^2        274877906944, NlogN             9961472
N             1048576, N^1.5          1073741824, N^2       1099511627776, NlogN            20971520
N             2097152, N^1.5          3037000499, N^2       4398046511104, NlogN            44040192
N             4194304, N^1.5          8589934592, N^2      17592186044416, NlogN            92274688
N             8388608, N^1.5         24296003999, N^2      70368744177664, NlogN           192937984
N            16777216, N^1.5         68719476736, N^2     281474976710656, NlogN           402653184
N            33554432, N^1.5        194368031998, N^2    1125899906842624, NlogN           838860800
N            67108864, N^1.5        549755813888, N^2    4503599627370496, NlogN          1744830464
N           134217728, N^1.5       1554944255987, N^2   18014398509481984, NlogN          3623878656
N           268435456, N^1.5       4398046511104, N^2   72057594037927936, NlogN          7516192768
N           536870912, N^1.5      12439554047901, N^2  288230376151711744, NlogN         15569256448
N          1073741824, N^1.5      35184372088832, N^2 1152921504606846976, NlogN         32212254720
```

# Sorted Lists

- Entries in a list are ordered simply by positions within list
- Can add a sort operation to the ADT list
- Add an entry to, remove an entry from sorted list
  - Provide only the entry.
  - No specification where entry belongs or exists

# Specifications for ADT Sorted List

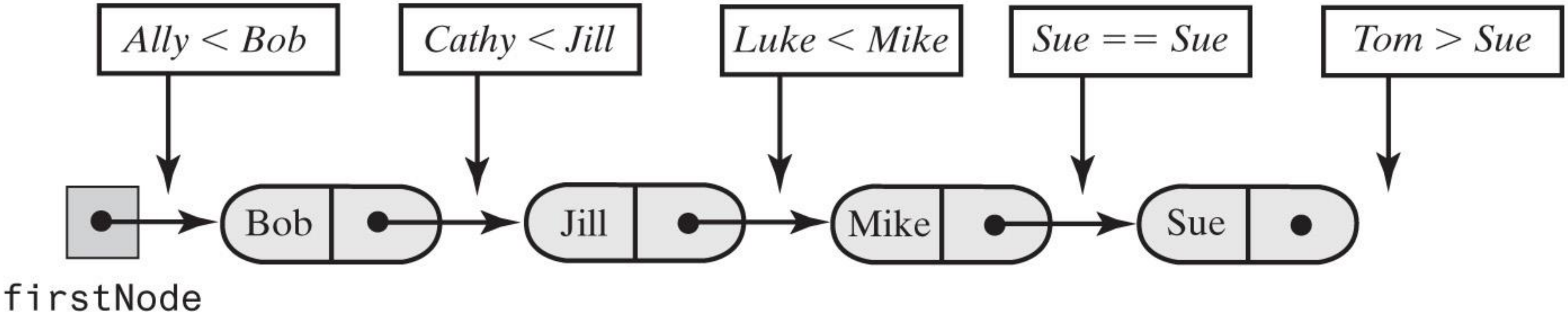
- DATA
  - A collection of objects in sorted order and having the same data type
  - The number of objects in the collection
- We will simply reuse revised ListInterface and then use a subclass of AList or LList to implement
- Operations requiring special consideration
  - **`add(newEntry) – just overridden for AList/LList`**
  - **`add(position, entry) – not allowed`**
  - **`replace() – not allowed`**

# Specifications for ADT Sorted List

- Additional Operations
  - These behave as they do for the ADT List
    - **`getEntry(givenPosition)`**
    - **`contains(anEntry)`**
    - **`findEntry(anEntry)`**
    - **`remove(givenPosition)`**
    - **`remove(anEntry)`**
    - **`clear()`**
    - **`size()`**
    - **`isEmpty()`**
    - **`toArray()`**

# SortedLList Implementation

- Places to insert additional names into a sorted chain of linked nodes



© 2019 Pearson Education, Inc.

# SortedLList Implementation

- Algorithm for `add` method.

```
Algorithm add(newEntry)
// Adds a new entry to the sorted list.
Allocate a new node containing newEntry
Search the chain until either you find a node containing newEntry or you pass
                         the point where it should be
Let nodeBefore reference the node before the insertion point
if (the chain is empty or the new node belongs at the beginning of the chain)
   Add the new node to the beginning of the chain
else
   Insert the new node after the node referenced by nodeBefore

Increment the length of the sorted list
```

# SortedLList/AList add()

- Iterative implementation
- Uses inheritance (LListWithIterator)
- Same code for sorted LList and AList

```
public class SortedLList<T extends Comparable<? super T>>
     extends CompletedLListWithIterator<T> {

     public SortedLList() {
          super();
     }

     @Override
     final public boolean add(T entry) {

          // find the position in sorted order that the entry precedes

          int position = findEntryBefore(entry);

          // if it is not found, simply add it to the end.

          if (position < 0) {
               return(super.add(entry));
          }

          // otherwise add it into the slot found

          return(super.add(position, entry));

     }
```

# add(position,entry) and replace()

- Not allowed

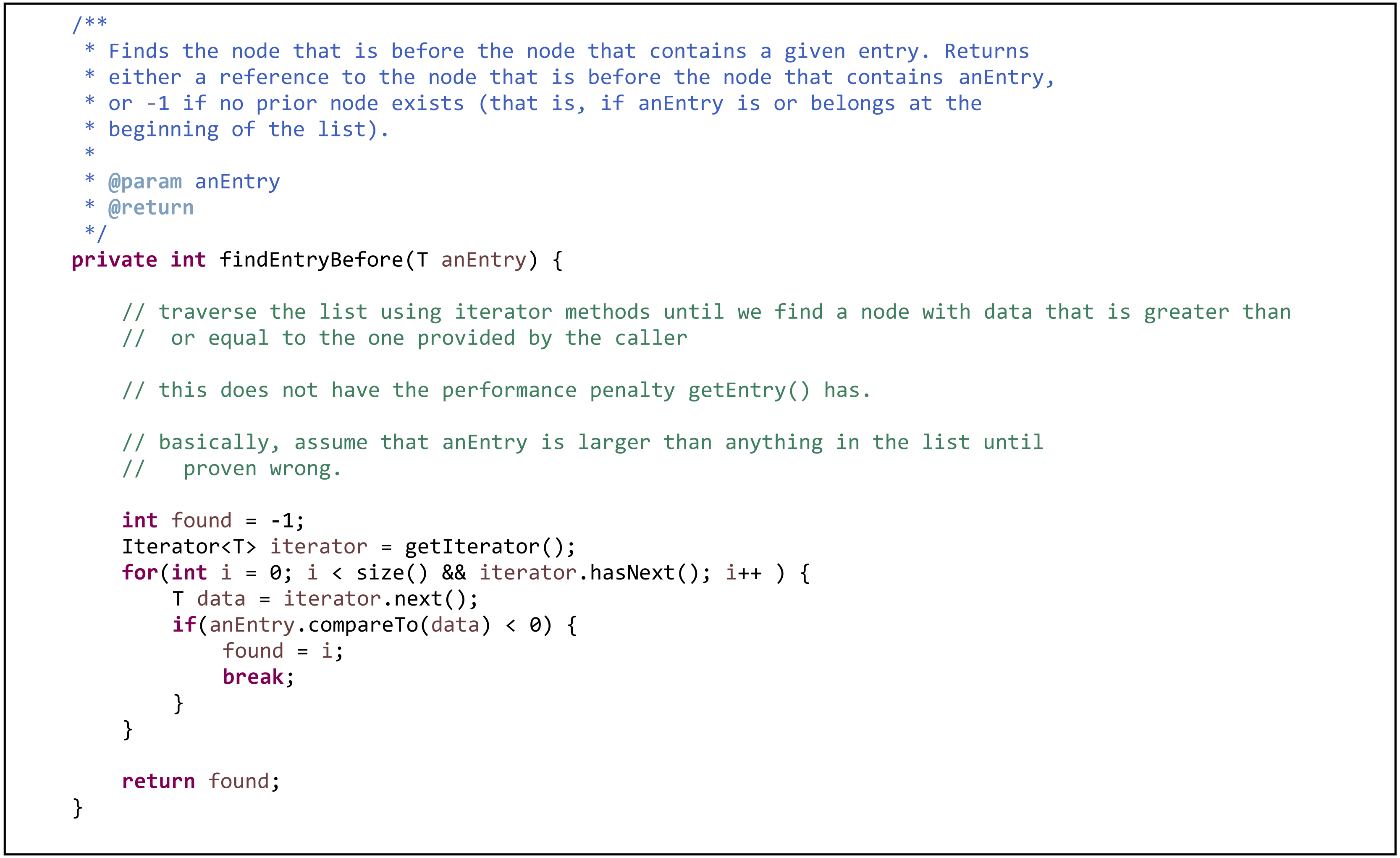
```
/**
 * This is not allowed, or list will be unsorted.
 */
@Override
public boolean add(int newPosition, T newEntry) {
   throw new UnsupportedOperationException("add() at a position is not legal for a sorted list");
}

@Override
public T replace(int givenPosition, T newEntry) {
   throw new UnsupportedOperationException("replace() at a position is not legal for a sorted list");
}
```

# SortedAList – findEntryBefore()

```java
/**
 * Finds a position in the list that an entry precedes in sorted order (ascending).
 *
 * We can use getEntry() here as there is no performance issue with the array implementation.
 *
 * This is different than SortedLList (although we could use the same code with an iterator)
 * @param entry
 * @return -1 if not found
 */
public int findEntryBefore(T entry) {

        int found = -1;

        for(int i = 0; i < size(); i++) {
            if(entry.compareTo(getEntry(i)) < 0) {
                        found = i;
                        break;
            }
        }
        return found;
}
```

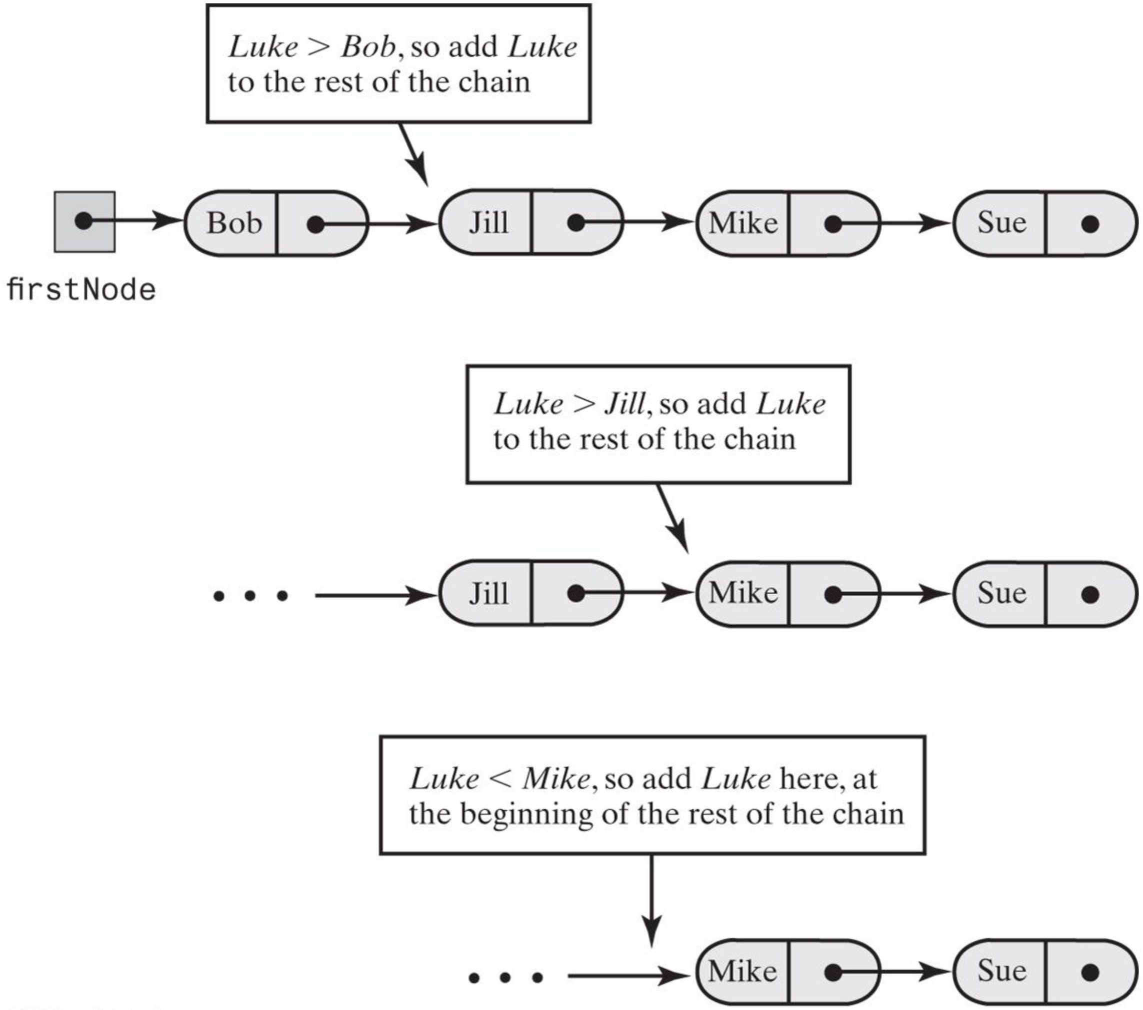
# SortedLList – findEntryBefore()

- uses iterator



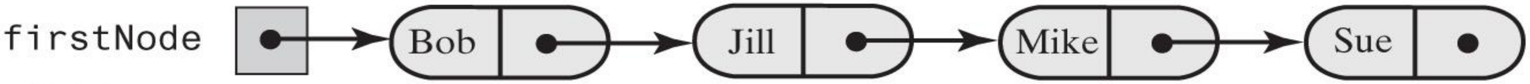
```java
/**
 * Finds the node that is before the node that contains a given entry. Returns
 * either a reference to the node that is before the node that contains anEntry,
 * or -1 if no prior node exists (that is, if anEntry is or belongs at the
 * beginning of the list).
 *
 * @param anEntry
 * @return
 */
private int findEntryBefore(T anEntry) {

    // traverse the list using iterator methods until we find a node with data that is greater than
    //  or equal to the one provided by the caller

    // this does not have the performance penalty getEntry() has.

    // basically, assume that anEntry is larger than anything in the list until
    //   proven wrong.

    int found = -1;
    Iterator<T> iterator = getIterator();
    for(int i = 0; i < size() && iterator.hasNext(); i++ ) {
        T data = iterator.next();
        if(anEntry.compareTo(data) < 0) {
            found = i;
            break;
        }
    }

    return found;
}
```

# Recursive Add to Sorted List


*Luke* > *Bob*, so add *Luke* to the rest of the chain
Bob
Jill
Mike
Sue
firstNode
*Luke* > *Jill*, so add *Luke* to the rest of the chain
• • •
Jill
Mike
Sue
*Luke* < *Mike*, so add *Luke* here, at the beginning of the rest of the chain
• • •
Mike
Sue

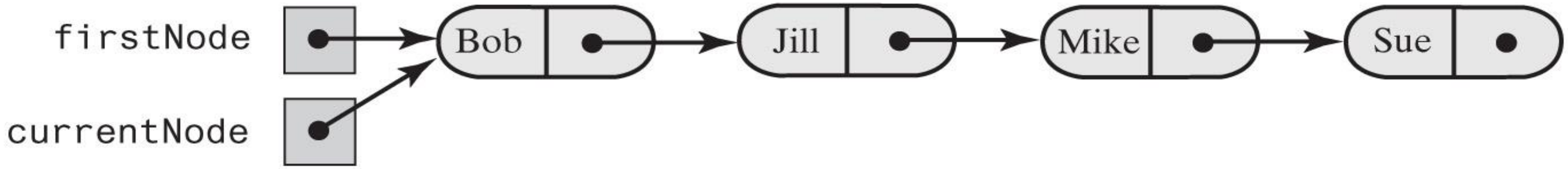
# Recursive Add to Sorted List (Part 1)

(a) The list before any additions



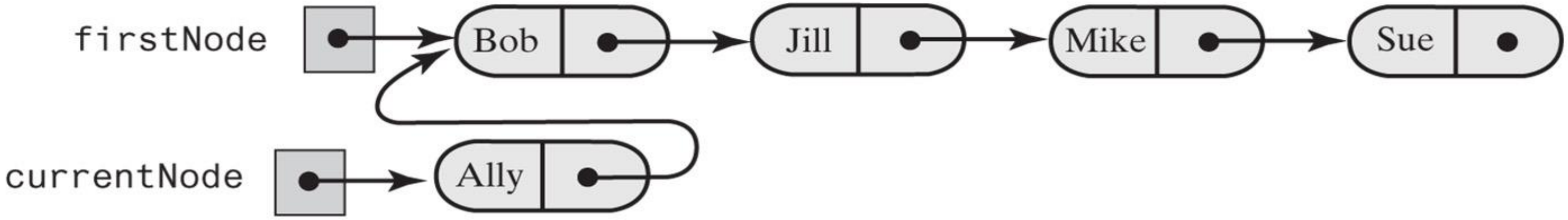
© 2019 Pearson Education, Inc.

(b) As `add("Ally", firstNode)` begins execution



© 2019 Pearson Education, Inc.

(c) After a new node is created (the base case)



The private method returns the reference that is in `currentNode`

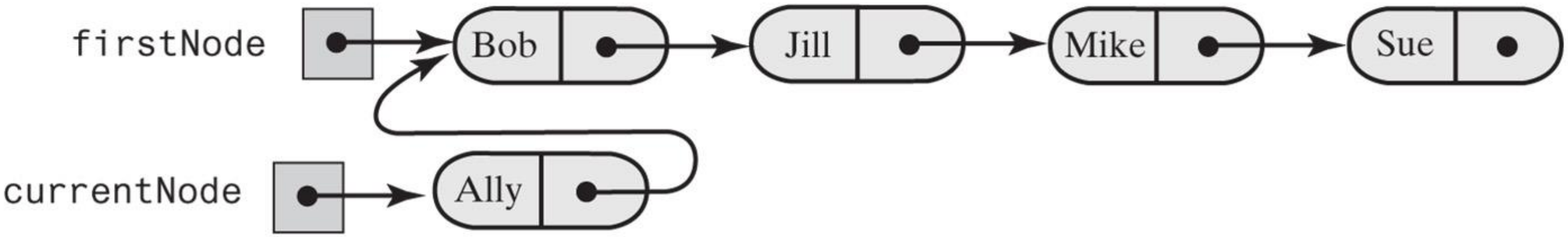
© 2019 Pearson Education, Inc.

# Recursive Add to Sorted List (Part 2)

*[from previous slide]*

- Recursively adding a node at the beginning of a chain

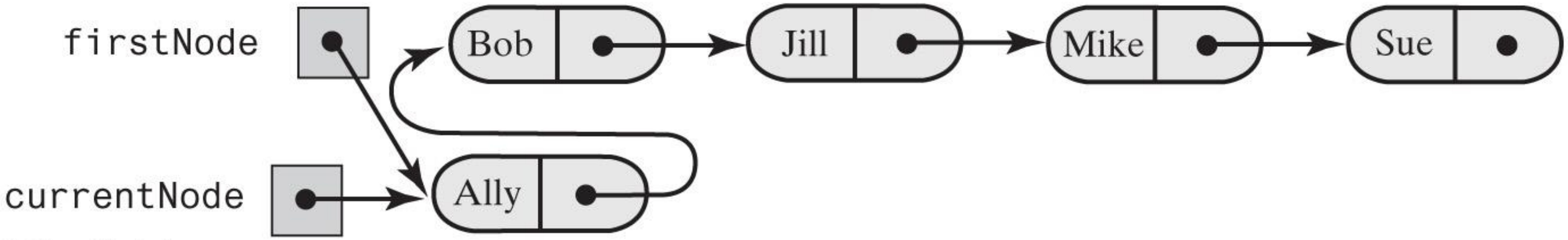
(c) After a new node is created (the base case)



The private method returns the reference that is in `currentNode`

© 2019 Pearson Education, Inc.

(d) After the public `add` assigns the returned reference to `firstNode`

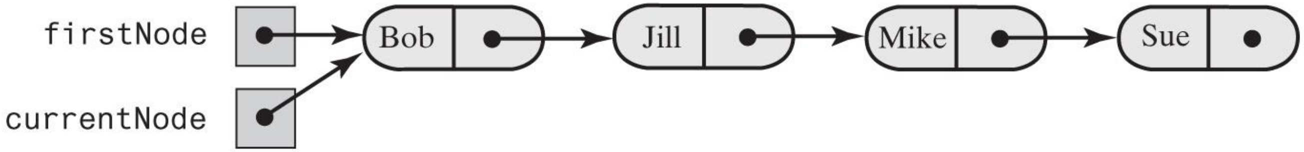


© 2019 Pearson Education, Inc.

# Recursive Add to Sorted List (Part 1)

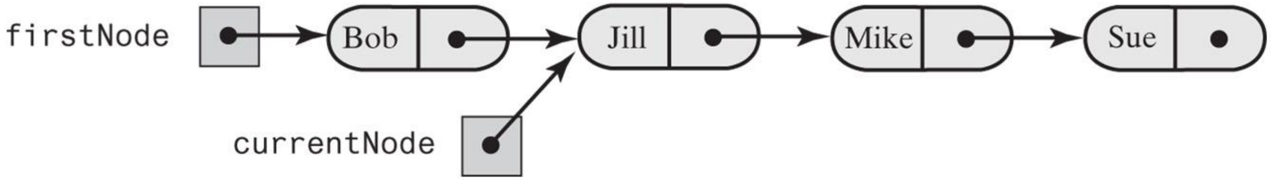
- Recursively adding a node between existing nodes in a chain

(a) As add("Luke", firstNode) begins execution



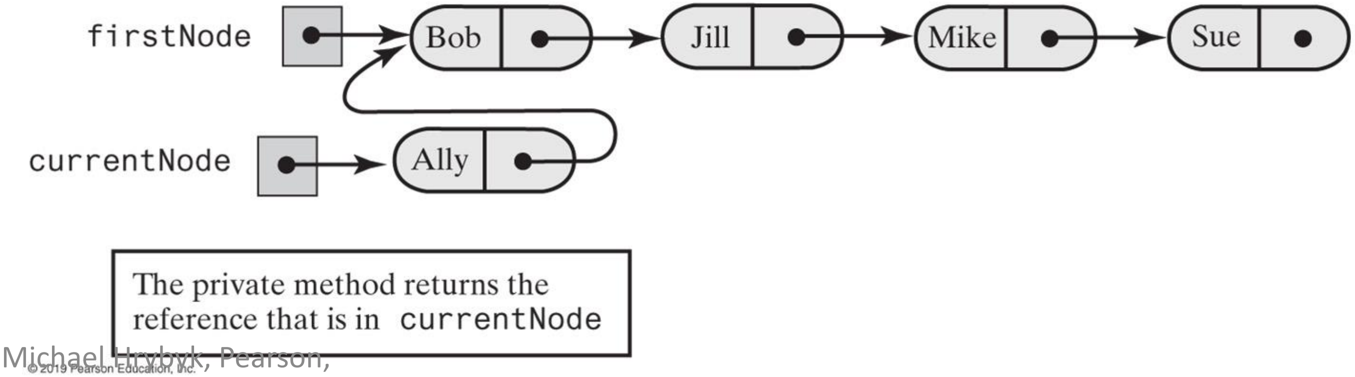
© 2019 Pearson Education, Inc.

(b) As the recursive call add("Luke", currentNode.getNextNode()) begins execution



© 2019 Pearson Education, Inc.

(c) After a new node is created (the base case)



The private method returns the reference that is in currentNode

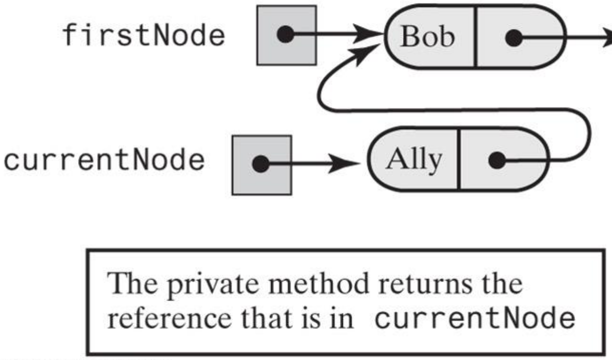
© 2019 Pearson Education, Inc.

# Recursive Add to Sorted List (Part 2)

*[from previous slide]*

- Recursively adding a node between existing nodes in a chain

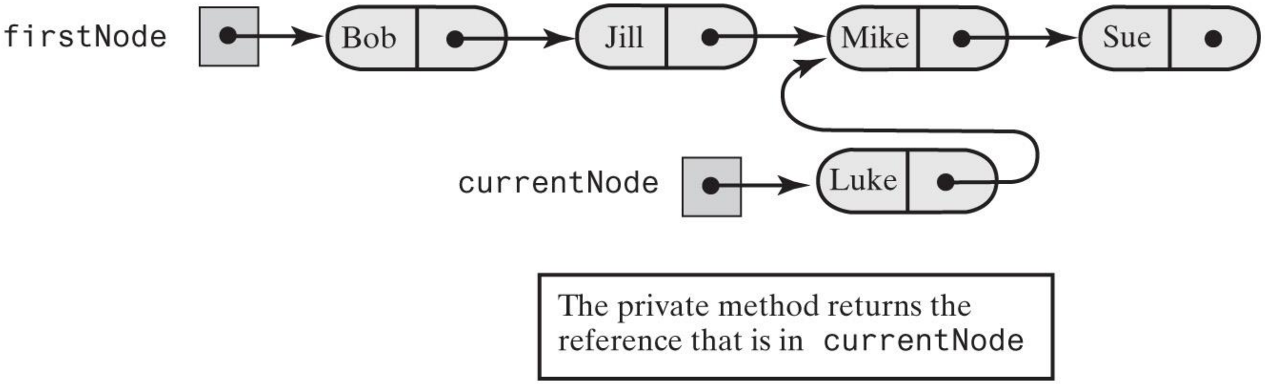
(c) After a new node is created (the base case)



The private method returns the reference that is in currentNode

© 2019 Pearson Education, Inc.

(d) After a new node is created (the base case)



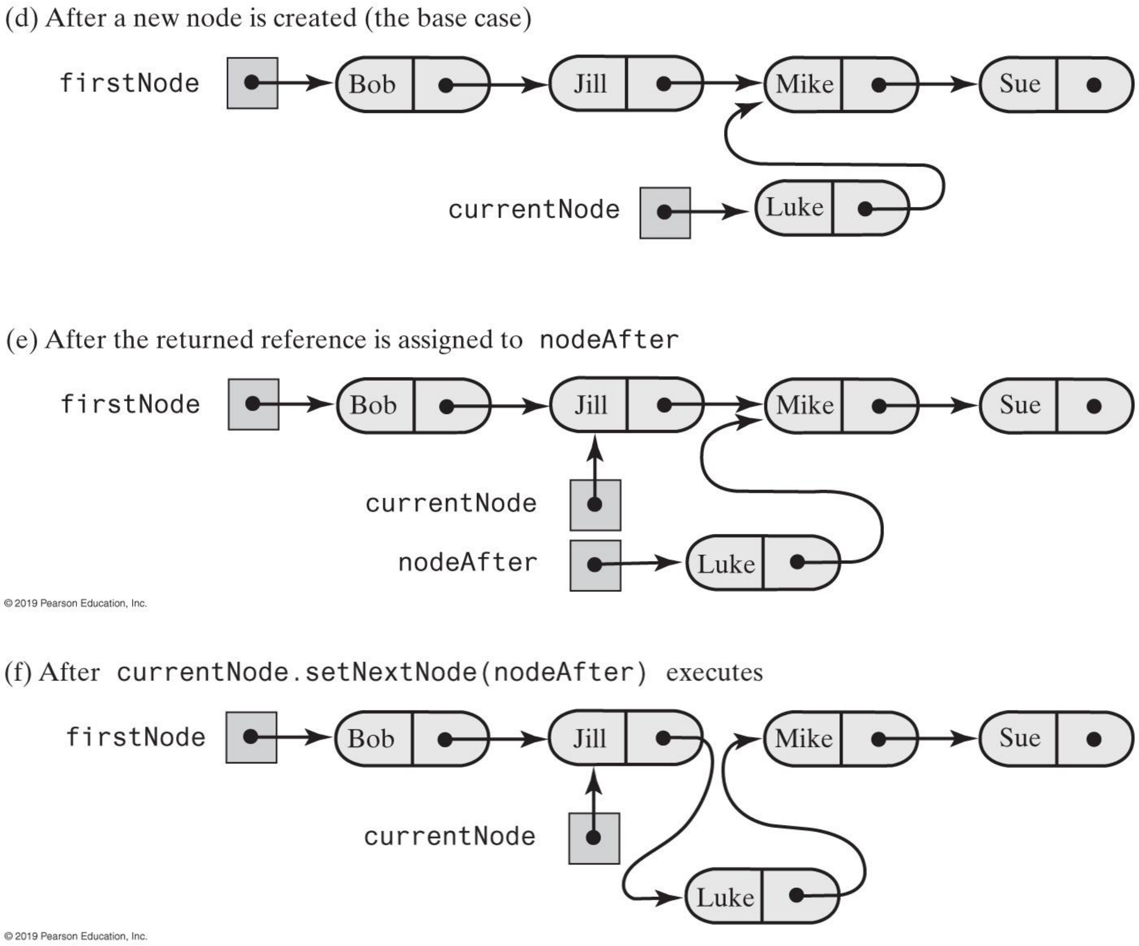
The private method returns the reference that is in currentNode

© 2019 Pearson Education, Inc.

# Recursive Add to Sorted List (Part 2)

*[from previous slide]*

- Recursively adding a node between existing nodes in a chain



(d) After a new node is created (the base case)

(e) After the returned reference is assigned to `nodeAfter`

© 2019 Pearson Education, Inc.

(f) After `currentNode.setNextNode(nodeAfter)` executes

© 2019 Pearson Education, Inc.

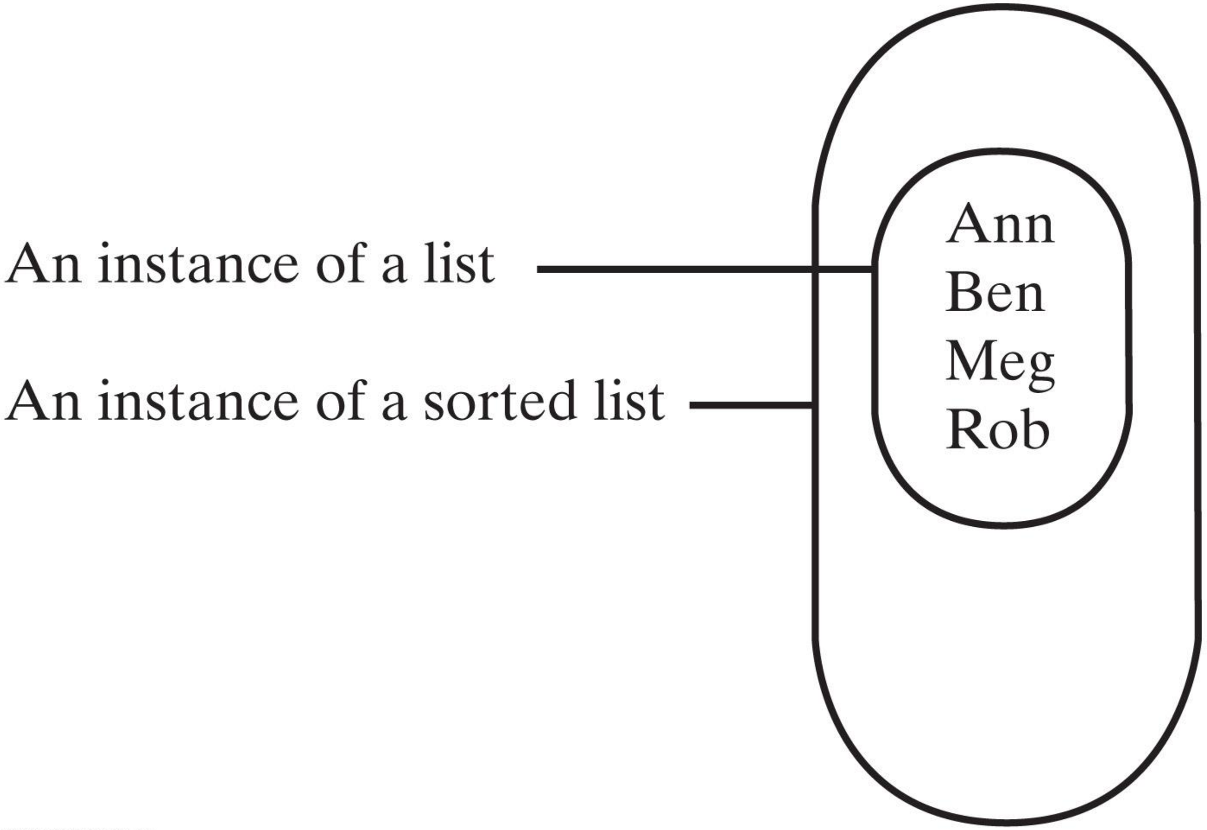
# Comparison of Implementations

- The worst-case efficiencies of the operations on the ADT sorted list for two implementations

| Operation | Array | Linked |
|---|---|---|
| `add(newEntry)` | $O(n)$ | $O(n)$ |
| `remove(anEntry)` | $O(n)$ | $O(n)$ |
| `getPosition(anEntry)` | $O(n)$ | $O(n)$ |
| `getEntry(givenPosition)` | $O(1)$ | $O(n)$ |
| `contains(anEntry)` | $O(n)$ | $O(n)$ |
| `remove(givenPosition)` | $O(n)$ | $O(n)$ |
| `display()` | $O(n)$ | $O(n)$ |
| `clear(), size(), isEmpty()` | $O(1)$ | $O(1)$ |

# Implementation that uses ADT List

- An instance of a sorted list that contains a list of its entries

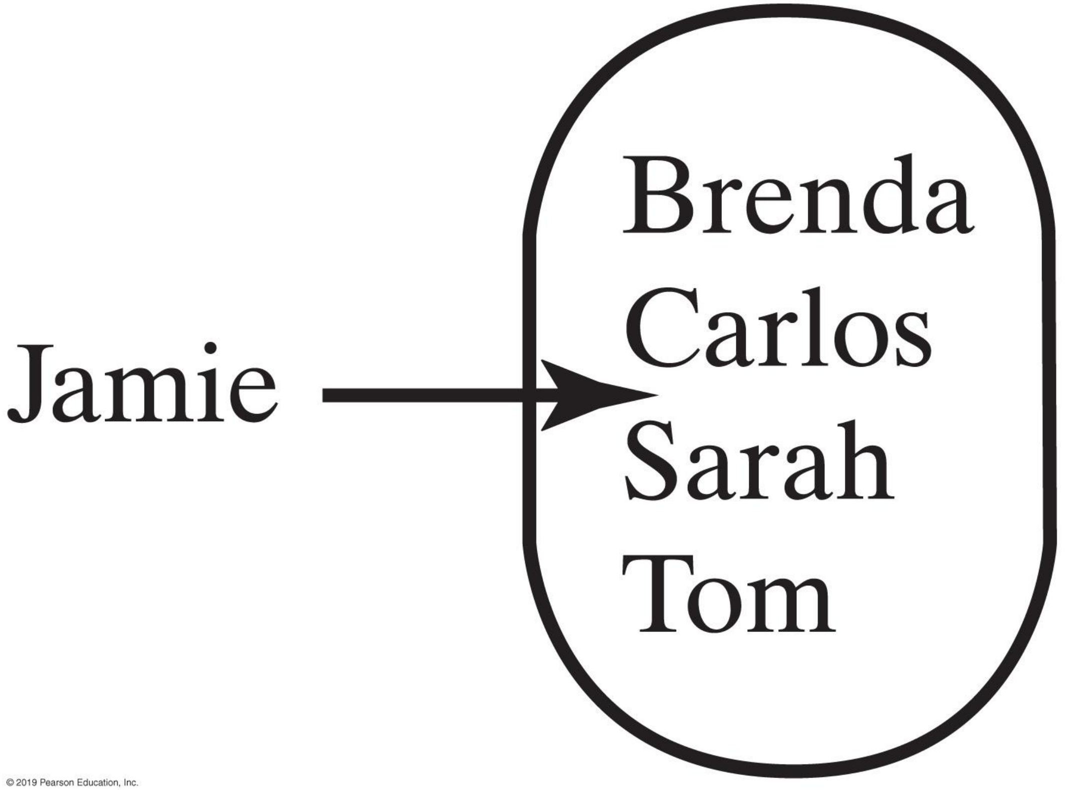


# SortedListDemo

```
   ListInterface<String> originalListOfIntegers = new CompletedAList<>(30);

   // first generate a list of integers, and display them

   DemoUtilities.generateListOfNumbers(originalListOfIntegers, 10, 30);
   DemoUtilities.display(originalListOfIntegers, "Original List of random Integers");

   // comment out one SortedList to test the other

   SortedLList<String> sortedListOfIntegers = new SortedLList<>();
//   SortedAList<String> sortedListOfIntegers = new SortedAList<>();

   // copy the original list to the sorted list. copy method uses add() so
   //  items will be inserted in order
   DemoUtilities.copyListOfNumbers(originalListOfIntegers, sortedListOfIntegers);


   DemoUtilities.display(sortedListOfIntegers, "List of Sorted Integers");

   sortedListOfIntegers.add("31");
   sortedListOfIntegers.add("32");
   sortedListOfIntegers.add("20");

   DemoUtilities.display(sortedListOfIntegers, "Sorted Integers with 20, 31 and 32 added");

   sortedListOfIntegers.removeEntry("31");
   sortedListOfIntegers.remove(2);
   DemoUtilities.display(sortedListOfIntegers, "Sorted Integers 31 and position 2 removed");
```

# Implementation That Uses the ADT List

- A sorted list in which Jamie belongs after Carlos but before Sarah



# Comparison of Implementations

- The worst-case efficiencies of the operations on the ADT sorted list for two implementations

| Operation | Array | Linked |
|---|---|---|
| `add(newEntry)` | $O(n)$ | $O(n)$ |
| `remove(anEntry)` | $O(n)$ | $O(n)$ |
| `findEntry(anEntry)` | $O(n)$ | $O(n)$ |
| `getEntry(givenPosition)` | $O(1)$ | $O(n)$ |
| `contains(anEntry)` | $O(n)$ | $O(n)$ |
| `remove(givenPosition)` | $O(n)$ | $O(n)$ |
| `clear(), size(), isEmpty()` | $O(1)$ | $O(1)$ |

# Comparison of Implementations

- The worst-case efficiencies of the ADT sorted list operations when implemented using an instance of the ADT list

| Operation | Array | Linked | AList | LList |
|---|---|---|---|---|
| **`add(newEntry)`** | O($n$) | O($n$) | O($n$) | O($n^2$) |
| **`remove(anEntry)`** | O($n$) | O($n$) | O($n$) | O($n^2$) |
| **`findEntry(anEntry)`** | O($n$) | O($n$) | O($n$) | O($n^2$) |
| **`getEntry(givenPosition)`** | O(1) | O($n$) | O(1) | O($n$) |
| **`contains(anEntry)`** | O($n$) | O($n$) | O($n$) | O($n$) |
| **`remove(givenPosition)`** | O($n$) | O($n$) | O($n$) | O($n$) |
| **`display()`** | O($n$) | O($n$) | O($n$) | O($n$) |
| **`clear(), size(), isEmpty()`** | O(1) | O(1) | O(1) | O(1) |