Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez’s and Dr. Farnan’s CS 0445 slides.)

Announcements

- Upcoming Deadlines:
  - Lab 8: next Monday 11/14 @ 11:59 pm
  - Homework 8: next Monday 11/14 @ 11:59 pm
  - Midterm reattempts: tonight @ 11:59 pm

Today …

- Sorting Algorithms

Muddiest Points

- **Q: Request some guidance on Lab 8. Just like Lab 7 the provided PowerPoint and PDF provides very unclear (to no!) instruction.**
- The PowerPoint, the PDF, **and your recitation TA** should give you a clear idea of how to finish the lab in a short amount of time

Muddiest Points

- **Q: I was confused why you had to switch all your instances of Node to Node<T>**
- Since Node was a static class, it cannot use any non-static data and types, including the type parameter T of class SortingAlgorithms<T>
- So, we had to define another (static) type parameter for the static Node class
    - The type parameter could also have been named S or any other name

Muddiest Points

- **Q: Why the keyword "static" was tripping up your code? I don't think I have a firm understanding on when static should/is required to be used.**
- I had to use static because I was calling the sorting methods from the static method main
- Alternatively, I could have called the methods from the class constructor, in which case static won't be needed
  - SortingAlgorithms.java now uses that approach

Muddiest Points

- **Q: Can you post the code you did today in class? My code doesn't compile and for some reason it isn't working.**
- The code is always accessible from the Draft Slides and Code Handouts link on Canvas

Sorting Algorithms

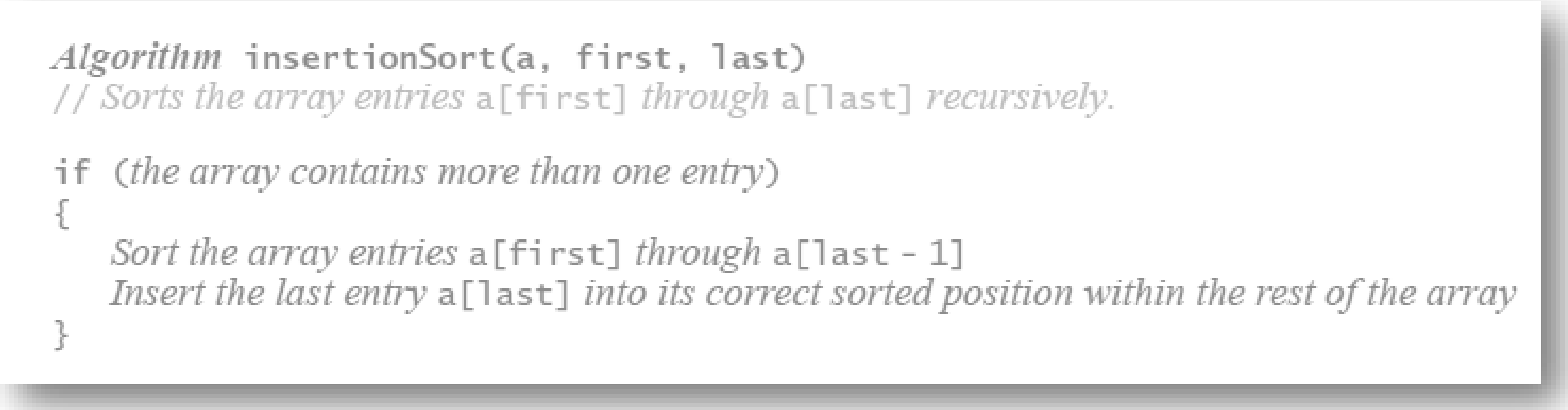
- $O(n^2)$
  - Selection Sort
  - Insertion Sort

Sorting Algorithms

- For each algorithm
  - understand the main concept using an example
  - implement the algorithm
    - on an Array
      - iterative
      - recursive
    - on a linked list
      - iterative
      - recursive

Recursive Insertion Sort

- This pseudocode describes a recursive insertion sort.

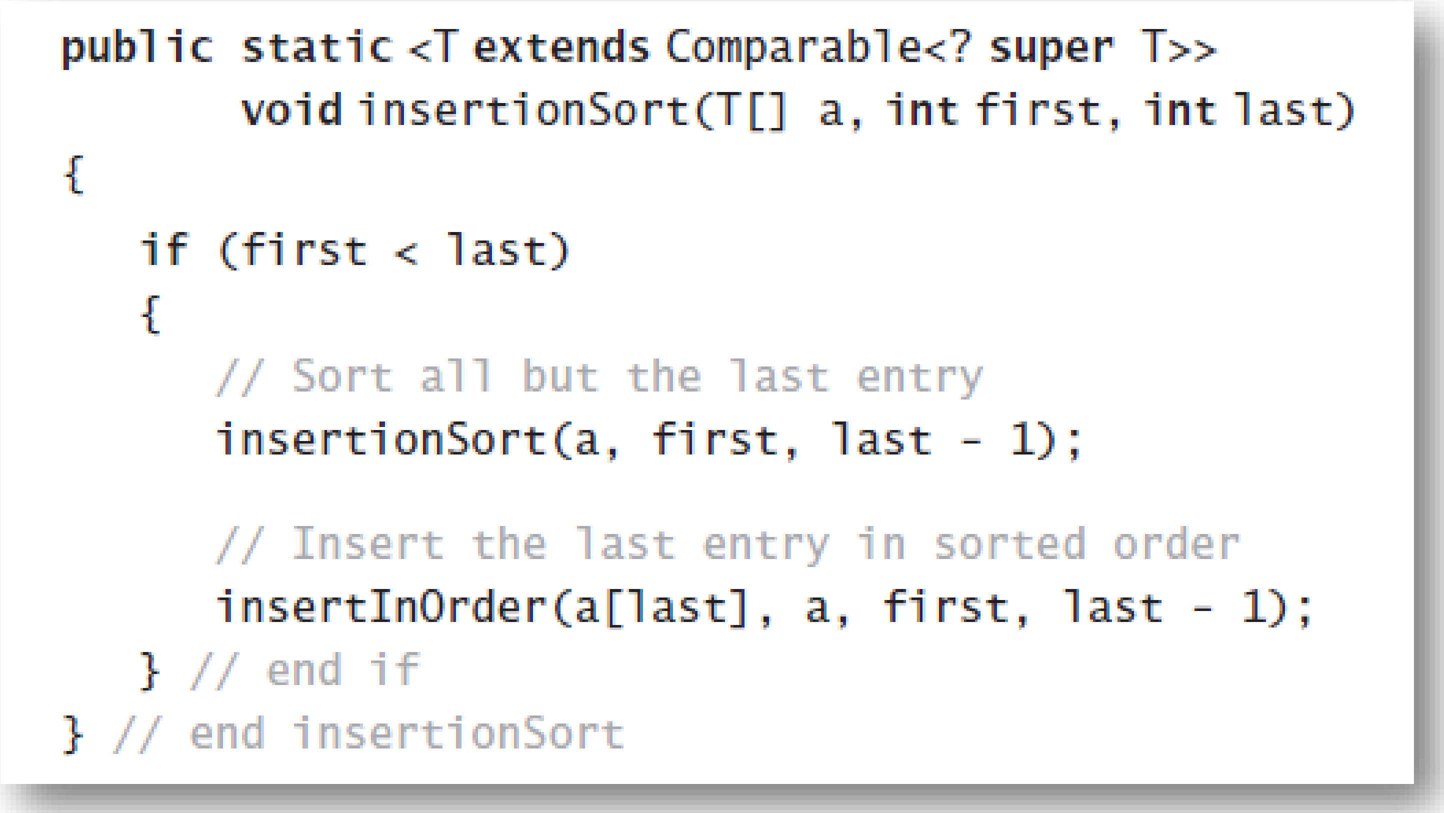


```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

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
}
```

Recursive Insertion Sort

- Implementing the algorithm in Java

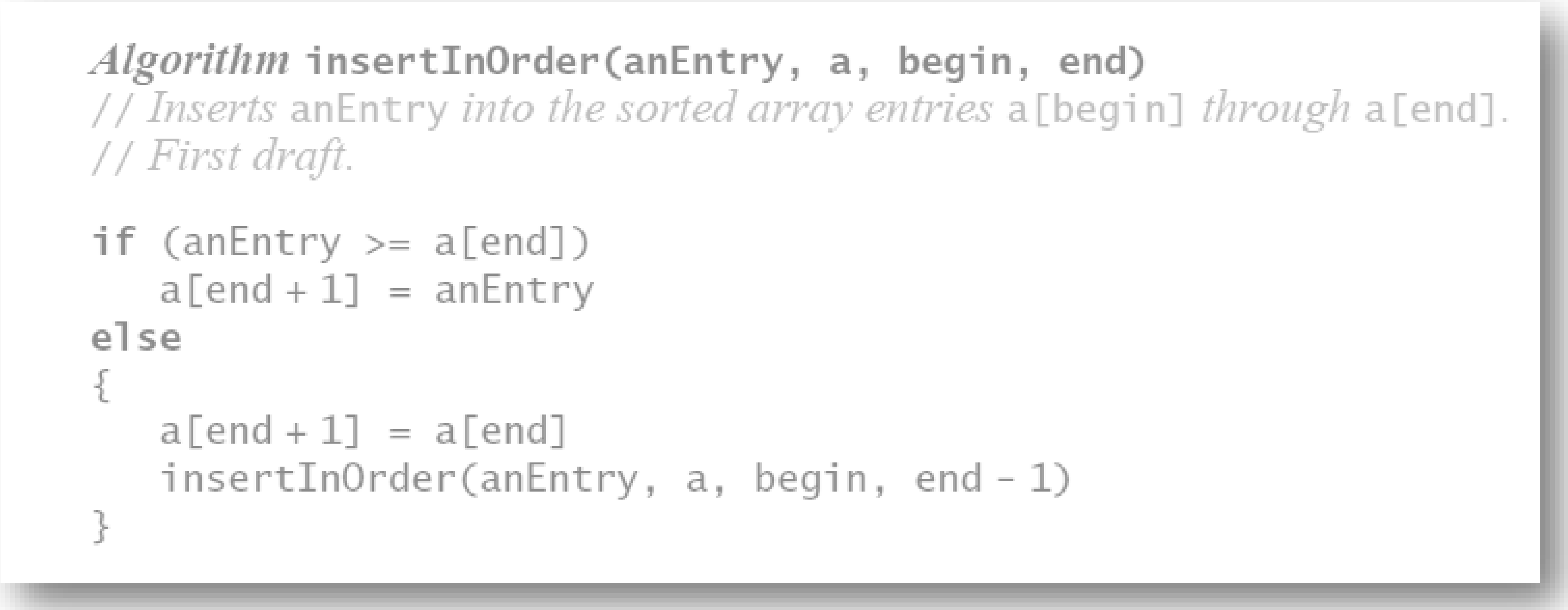


```
public static <T extends Comparable<? super T>>
       void insertionSort(T[] a, int first, int last)
{
   if (first < last)
   {
      // Sort all but the last entry
      insertionSort(a, first, last - 1);

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

Recursive Insertion Sort

- First draft of `insertInOrder` algorithm.

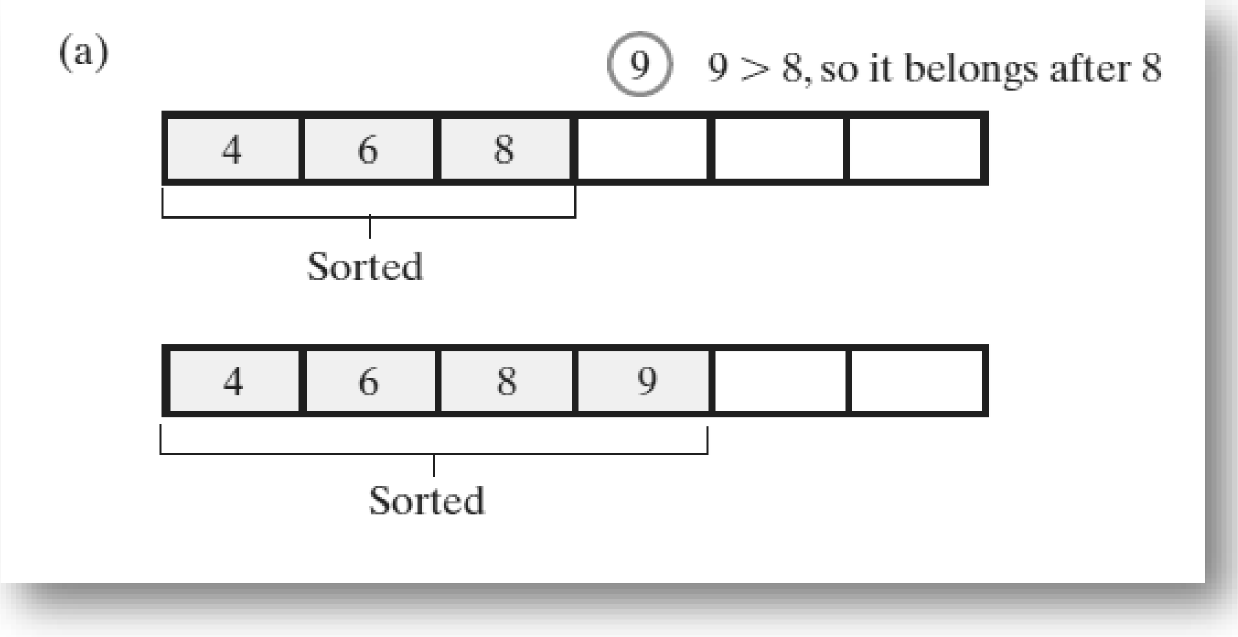


```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// First draft.

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

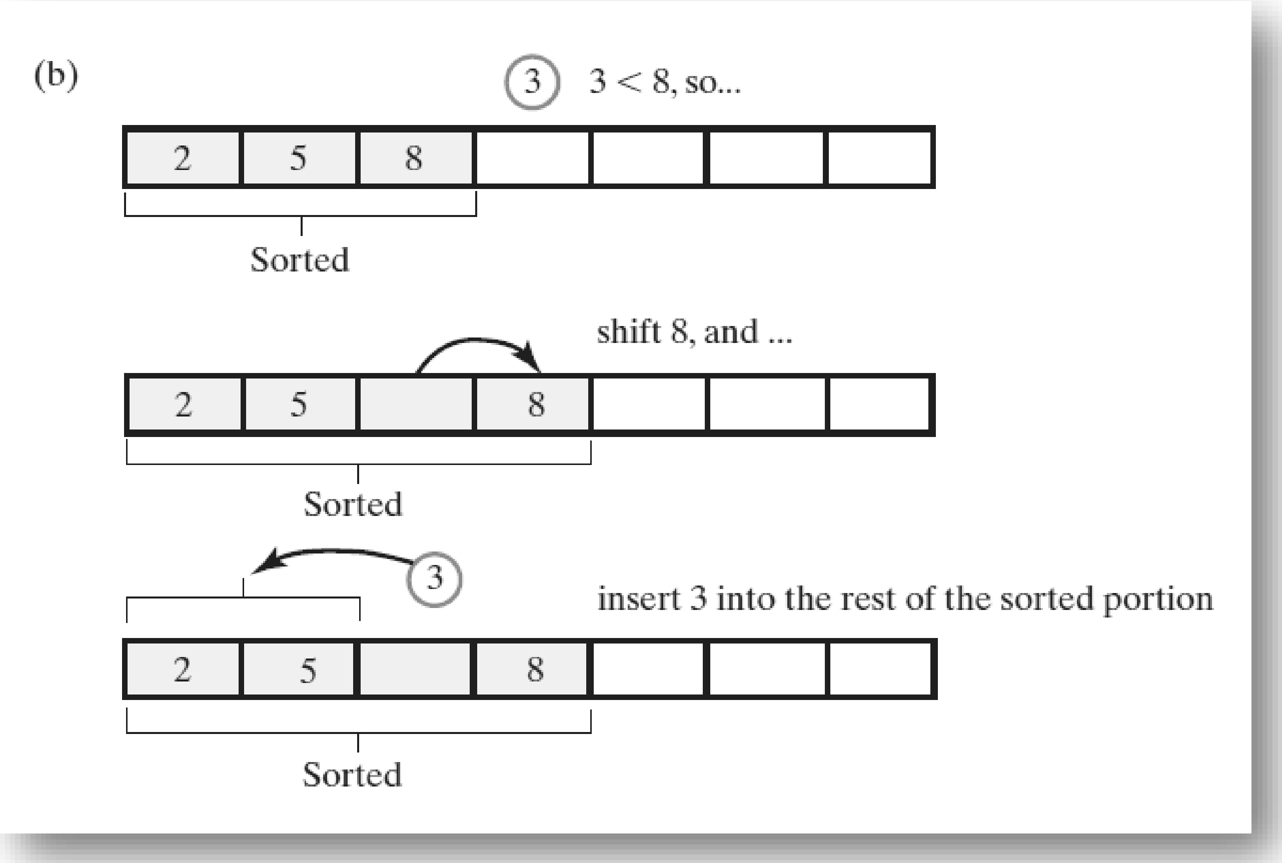
Recursive Insertion Sort

- 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



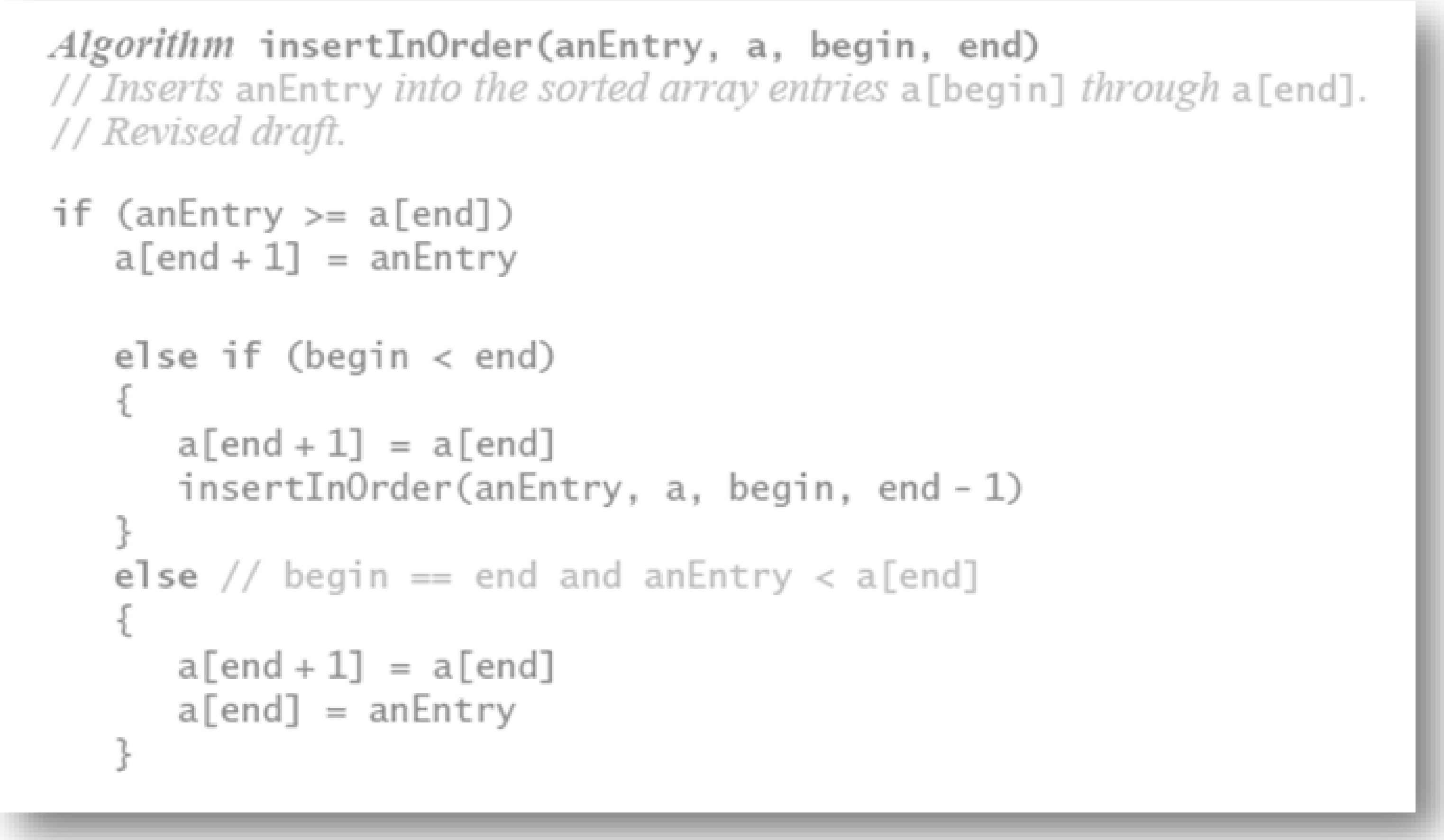
Recursive Insertion Sort

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



Recursive Insertion Sort

- The algorithm **insertInOrder**: final draft.
  Note: insertion sort efficiency (worst case) is $O(n^2)$



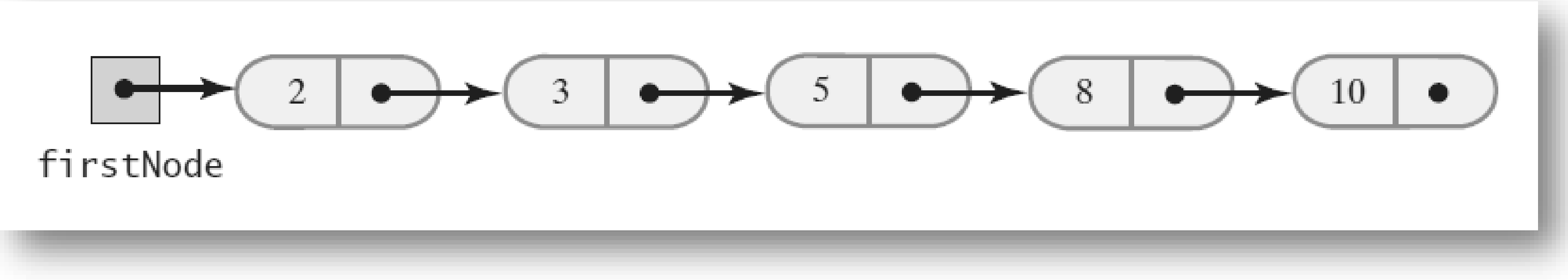
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

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
   }
```

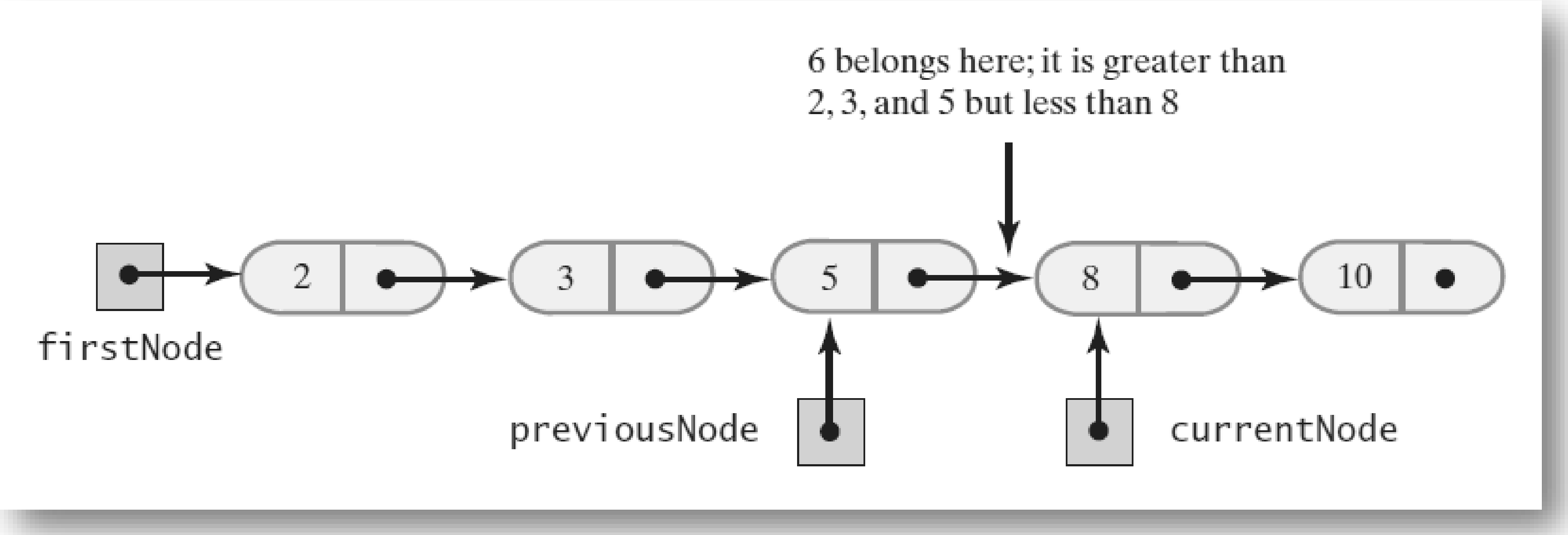
Insertion Sort of a Chain of Linked Nodes

- A chain of integers sorted into ascending order



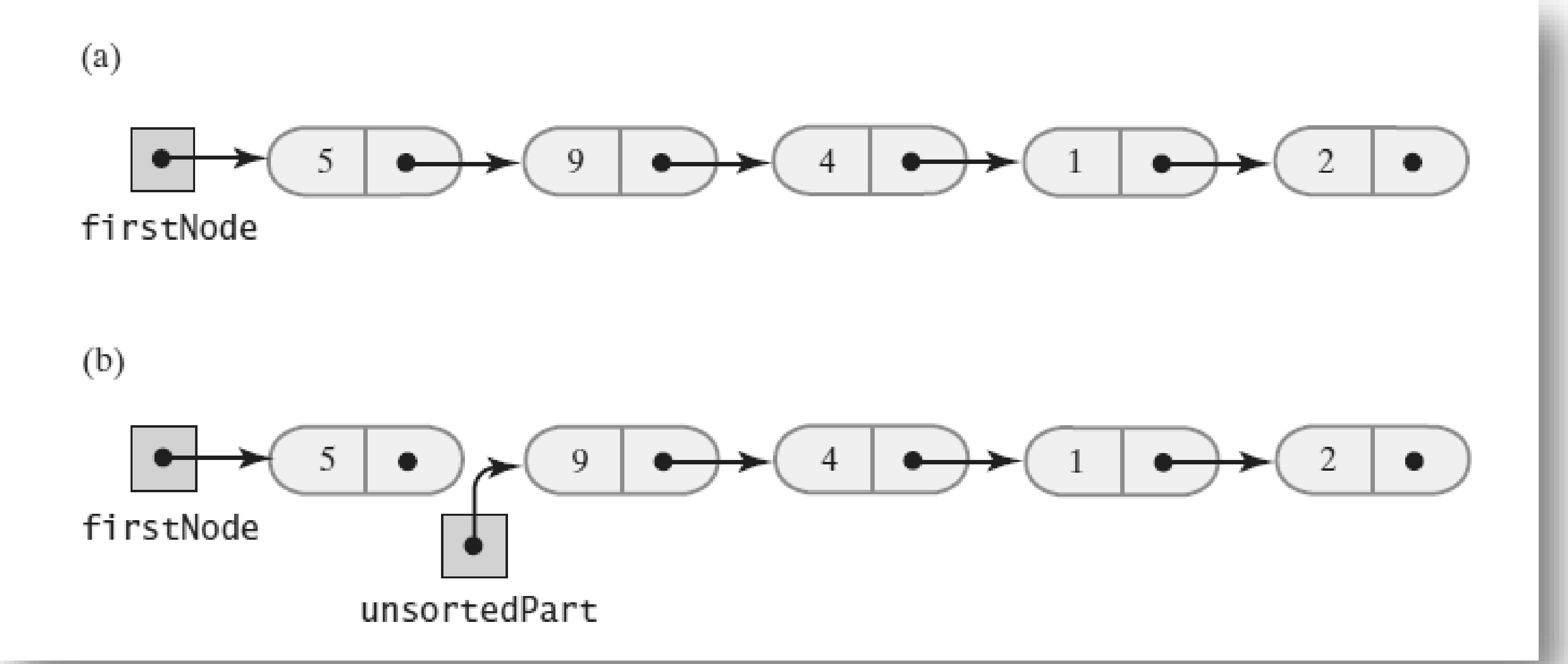
Insertion Sort of a Chain of Linked Nodes

- 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

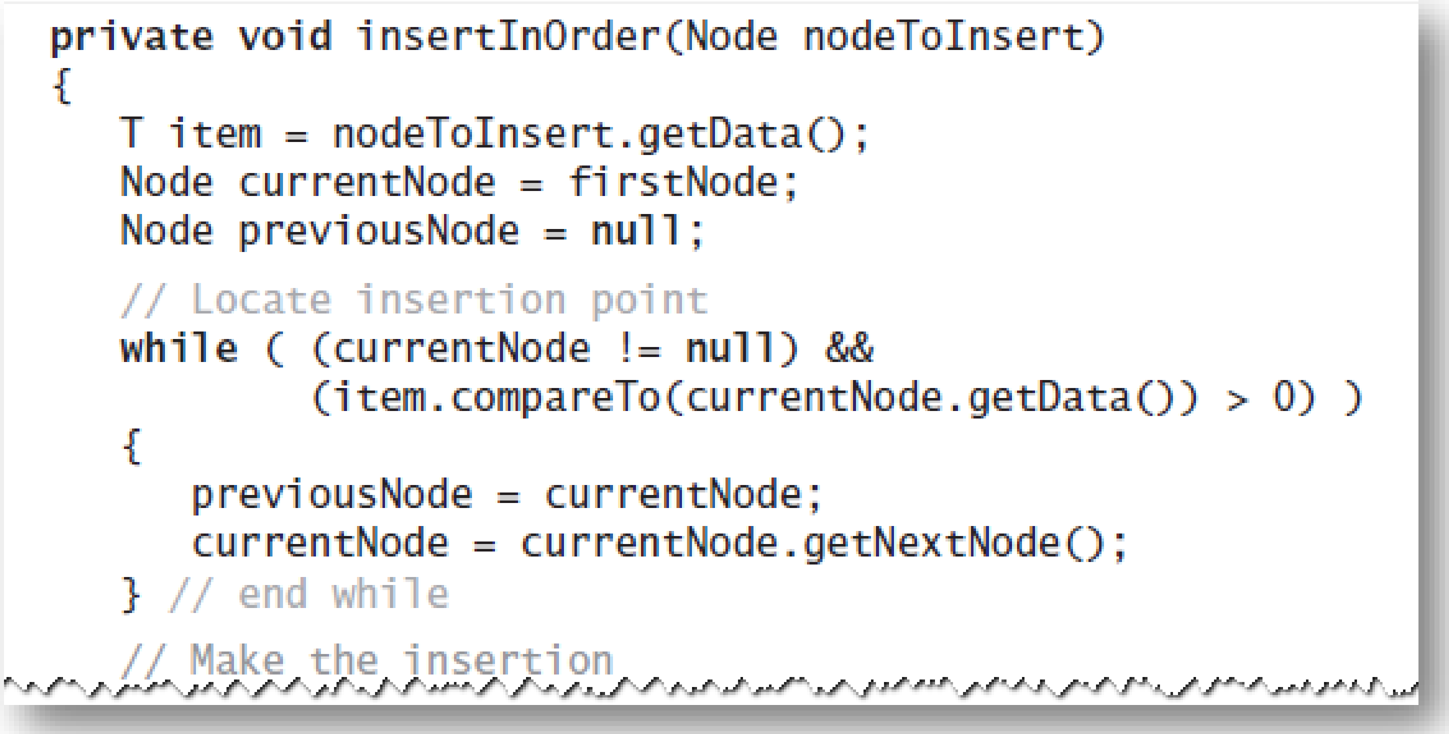
Insertion Sort of a Chain of Linked Nodes

- 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

- Define an inner class **Node** that has set and get methods

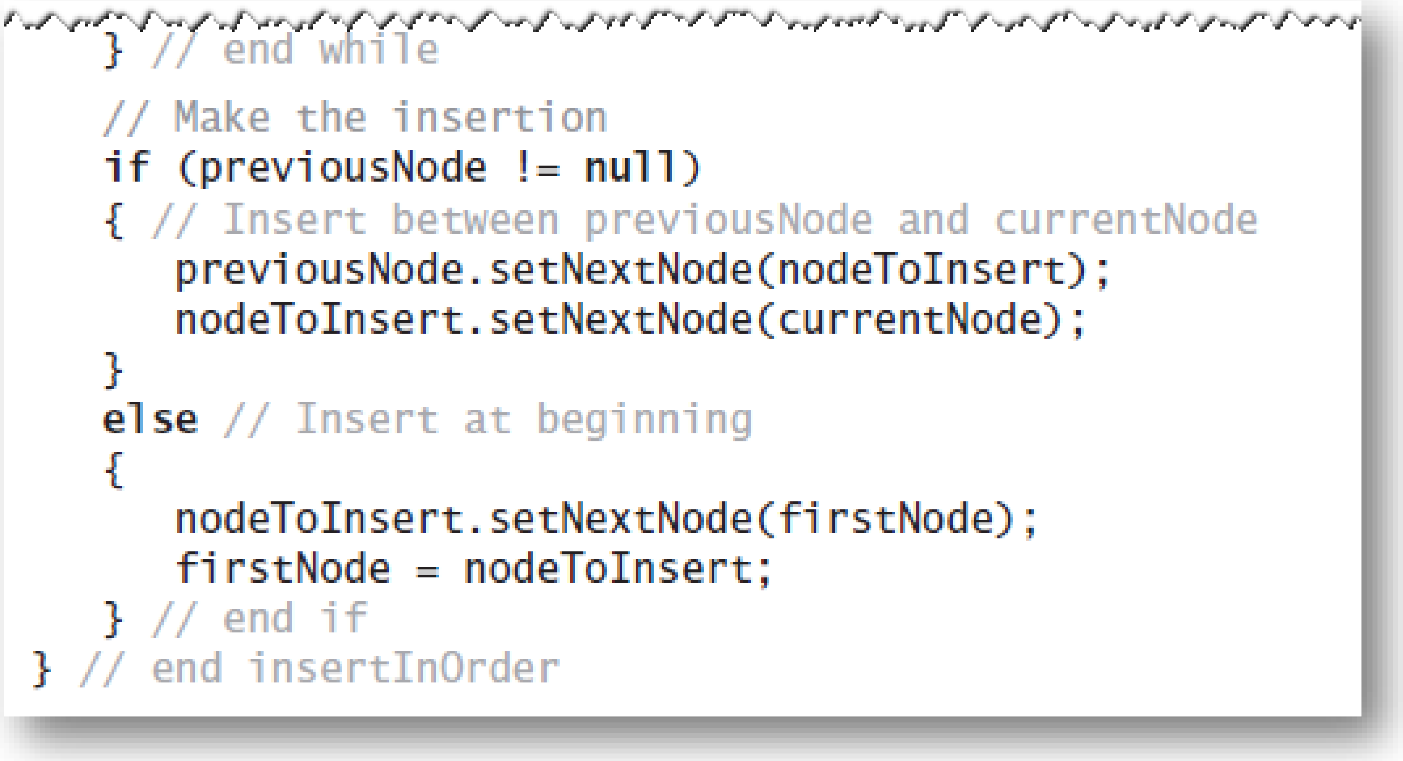


```
private void insertInOrder(Node nodeToInsert)
{
   T item = nodeToInsert.getData();
   Node currentNode = firstNode;
   Node previousNode = null;

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

   // Make the insertion
```

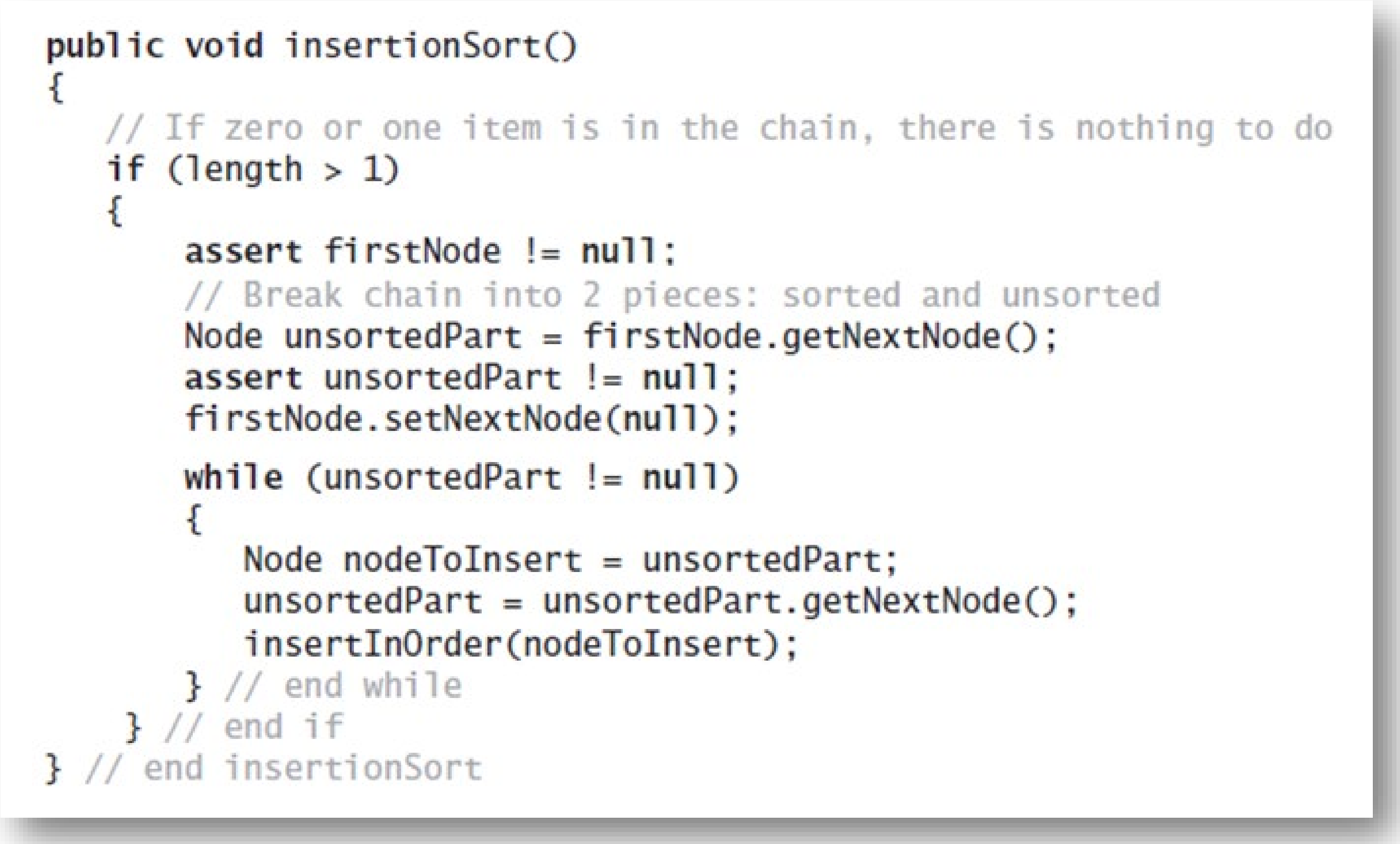
Insertion Sort of a Chain of Linked Nodes



```
   } // 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
```

Insertion Sort of a Chain of Linked Nodes

- The method to perform the insertion sort.



```
public void insertionSort()
{
   // If zero or one item is in the chain, there is nothing to do
   if (length > 1)
   {
       assert firstNode != null;
       // Break chain into 2 pieces: sorted and unsorted
       Node unsortedPart = firstNode.getNextNode();
       assert unsortedPart != null;
       firstNode.setNextNode(null);

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

Efficiency of Selection and Insertion Sorts

- 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 is O(n^2) in the worst-case
    - Requires O(n^2) comparisons and swaps
    - O(n) in the best case

Some properties of Selection and Insertion Sorts

- Selection sort is
  - not stable
  - in-place
  - non-adaptive
  - provides partial solution when interrupted in the middle of execution
- Insertion sort
  - stable
  - in-place
  - adaptive
    - the more sorted an array is, the less work `insertInOrder` must do
  - very fast on small arrays
    - small constant factors