CSC 273 – Data Structures

Lecture 5 - Introduction to Sorting

The Interface `Comparable`

- Consider the method **`compareTo`** for class **`String`**
- if *s* and *t* are strings, **`s.compareTo(t)`** is
  - Negative if *s* comes before *t*
  - Zero if *s* and *t* are equal
  - Positive if *s* comes after *t*

The Interface **`Comparable`**

- By invoking **`compareTo`**, you compare two objects of the class **`T`**.

```
package java.lang;
public interface Comparable<T>
{
   public int compareTo(T other);
} // end Comparable
```

The Interface **`Comparable`**

- By invoking **`compareTo`**, you compare two objects of the class **`T`**.

```
public class Circle implements Comparable<Circle> {
   private double radius;

   //Constructors and methods go here
   … …
}
```

```
public int compareTo(Circle other) {
  int  result;

  if (this.equals(other))
    result = 0;
  else if (radius < other.radius)
    result = -1;
  else
    result = 1;

  return result;
}
```

Generic Methods – An Example

```
public class Example {
  public static <T> void displayArray
                              (T[] anArray) {
    for (T arrayEntry : anArray) {
      System.out.print(arrayEntry);
      System.out.print(' ');
    }
    System.out.println();
  }
```

```
  public static void main(String[] args) {
    String[] stringArray
       = {"apple", "banana", "carrot",
        "dandelion"};

    System.out.print("stringArray contains ");
    displayArray(stringArray);

    Character[] characterArray
                = {'a', 'b', 'c', 'd'};
    System.out.print("characterArray contains ");
    displayArray(characterArray);
  }
}
```

Output

```
stringArray contains apple banana carrot dandelion
characterArray contains a b c d
```

Bounded Type Parameters

Consider this simple class of squares:

```
public class Square<T>
{
   private T side;

   public Square(T initialSide)
   {
      side = initialSide;
   } // end constructor

   public T getSide()
   {
      return side;
   } // end getSide
} // end Square
```

Bounded Type Parameters

Note the different types of square objects possible

```
Square<Integer> intSquare = new Square<>(5);
Square<Double> realSquare = new Square<>(2.1);
Square<String> stringSquare= new Square<>("25");
```

Bounded Type Parameters

Imagine that we want to write a static method that returns the smallest object in an array. Suppose that we wrote our method:

```
public MyClass
{
   // First draft and INCORRECT:
   public static <T> T arrayMinimum(T[] anArray)
   {
      T minimum = anArray[0];
      for (T arrayEntry : anArray)
      {
         if (arrayEntry.compareTo(minimum) < 0)
            minimum = arrayEntry;
      } // end for

      return minimum;
   } // end arrayMinimum
```

Bounded Type Parameters

The header should look like this:

```
public MyClass
{
public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)  // Correct header

   {
      T minimum = anArray[0];
      for (T arrayEntry : anArray)
      {
         if (arrayEntry.compareTo(minimum) < 0)
            minimum = arrayEntry;
      } // end for

      return minimum;
   } // end arrayMinimum
```

Wildcards

- Question mark, **?**, is used to represent an unknown class type
  - Referred to as a wildcard
- Consider following method and objects

```
public static void displayPair(OrderedPair<?> pair)
{
   System.out.println(pair);
} // end displayPair

...

OrderedPair<String> aPair = new OrderedPair<>("apple", "banana");
OrderedPair<Integer> anotherPair = new OrderedPair<>(1, 2);
```

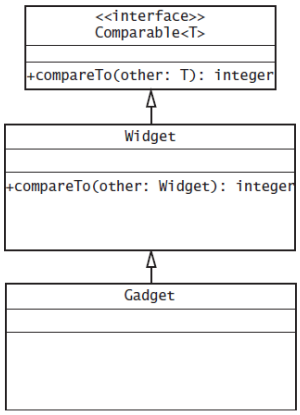
Wildcards

- Method **displayPair** will accept as an argument a pair of objects whose data type is any one class

```
displayPair(aPair);
displayPair(anotherPair);
```

Bounded Wildcards

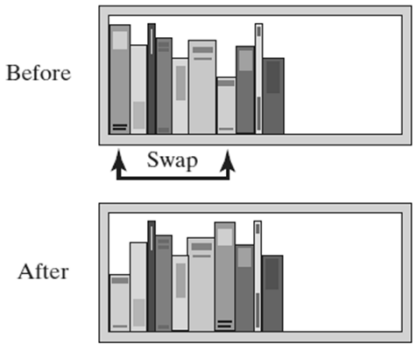
- The class **`Gadget`** is derived from the class **`Widget`**, which implements the interface **`Comparable`**



Sorting

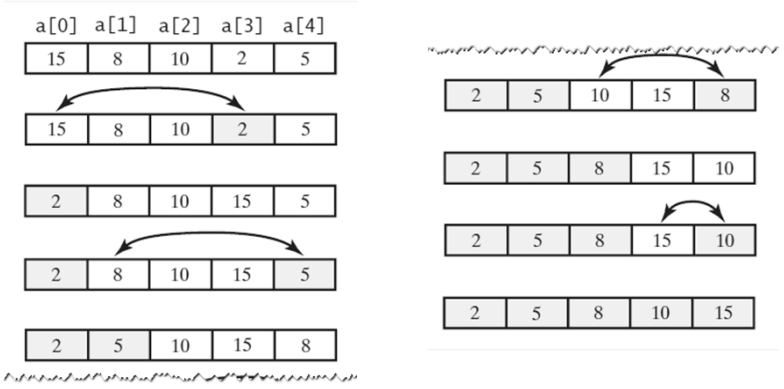
- We seek algorithms to arrange items, a_i such that
  entry 1 ≤ *entry 2* ≤ . . . ≤ *entry n*
- Sorting an array is usually easier than sorting a chain of linked nodes
- Efficiency of a sorting algorithm is significant

Selection Sort



Before and after exchanging the shortest book and the first book

Selection Sort



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].
}
```

SelectionSort

```
// Sorts the first n objects in an array
// into ascending order.
//   @param a  An array of Comparable objects.
//   @param n  An integer > 0. */
public static <T extends Comparable<? super T>>
      void selectionSort(T[] a, int n)    {
  for (int index = 0; index < n - 1; index++) {
    int indexOfNextSmallest =
          getIndexOfSmallest(a, index, n - 1);
    swap(a, index, indexOfNextSmallest);
   // Assertion: a[0] <= a[1] <= . . . <=
   //             a[index] <= all other a[i]
  }
}
```

```
// Finds the index of the smallest value in a
// portion of an array a.
// Precondition: a.length > last >= first >= 0.
// Returns the index of the smallest value among
// a[first], a[first + 1], . . . , a[last].
private static <T extends Comparable<? super T>>
      int getIndexOfSmallest(T[] a, int first,
                                    int last)    {
  T min = a[first];
  int indexOfMin = first;
```

```
  for (int index = first + 1; index <= last;
                                      index++)    {
    if (a[index].compareTo(min) < 0)  {
      min = a[index];
      indexOfMin = index;
    } // end if

    // Assertion: min is the smallest of
    // a[first] through a[index].
  } // end for

  return indexOfMin;
} // end getIndexOfSmallest
```

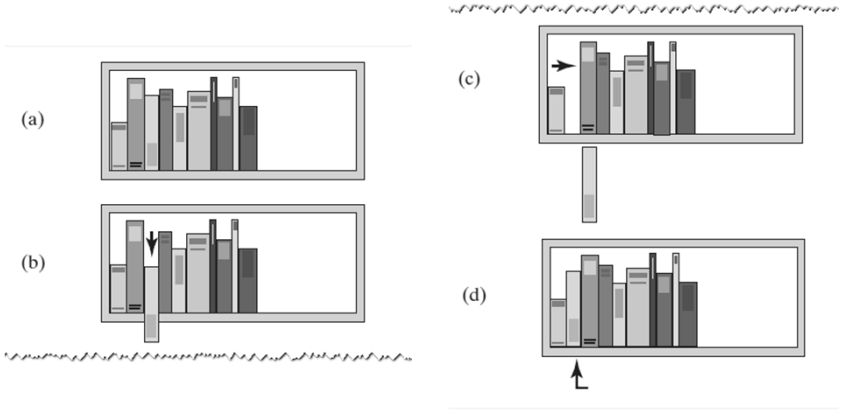
Recursive Selection Sort

```
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)
}
```

Efficiency of Selection Sort

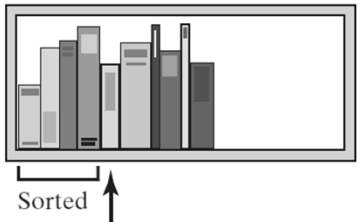
- Selection sort is $O(n^2)$ regardless of the initial order of the entries.
  - Requires $O(n^2)$ comparisons
  - Does only O(n) swaps

Insertion Sort



The placement of the third book during an insertion sort

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.

An insertion sort of books

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.*

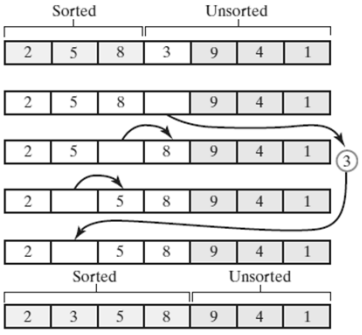
insertionSort()

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

insertionSort()

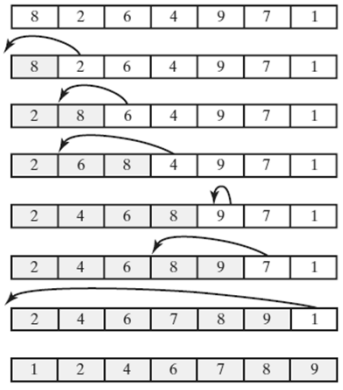
```
// insertionSort() - the iterative method
public static <T extends Comparable<? super T>>
         void insertionSort(T[] a, int first,
                                    int last)  {
  for (int unsorted = first + 1;
              unsorted <= last; unsorted++)   {

    // Assertion: a[first] <= a[first + 1]
                     <= ... <= a[unsorted - 1]
    T firstUnsorted = a[unsorted];
    insertInOrder(firstUnsorted, a, first,
                                    unsorted - 1);
  }
}
```

Iterative Insertion Sort



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

Iterative Insertion Sort



An insertion sort of an array of integers into ascending order

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

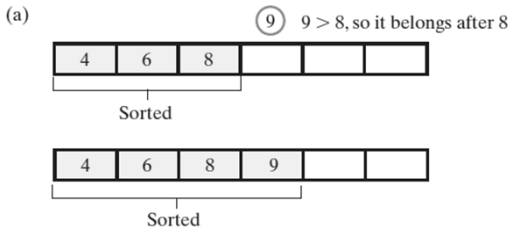
```
public static <T extends Comparable<? super T>>
      void insertionSort(T[] a, int first,
                                 int last)  {
  if (first < last) {
    insertionSort(a, first, last-1);
    insertInOrder(a[last], a, first, last -1);
  }
}
```

Recursive Insertion Sort

```
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)
}
```

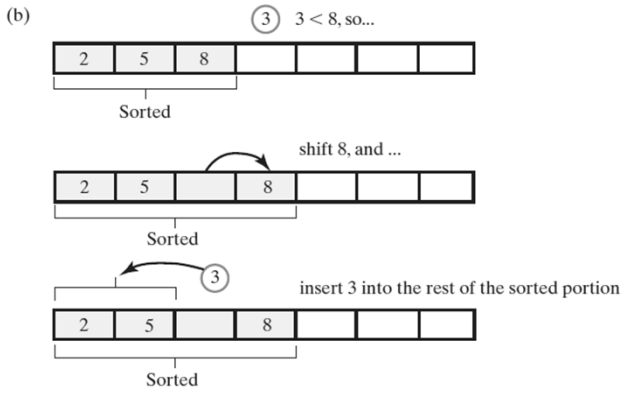
First draft of **`insertInOrder`** *algorithm.*

Recursive Insertion Sort



Inserting the first unsorted entry into the sorted portion of the array.
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. The entry is smaller than the last sorted entry

Recursive Insertion Sort

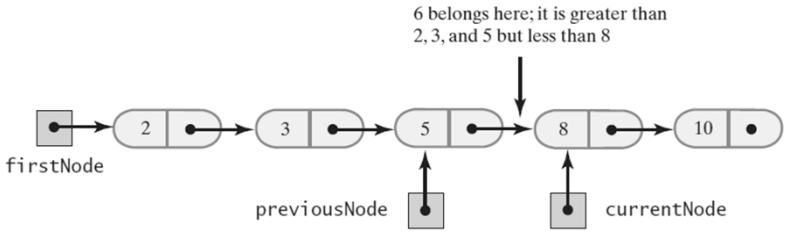
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

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
   }
```

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

Insertion Sort of a Chain of Linked Nodes

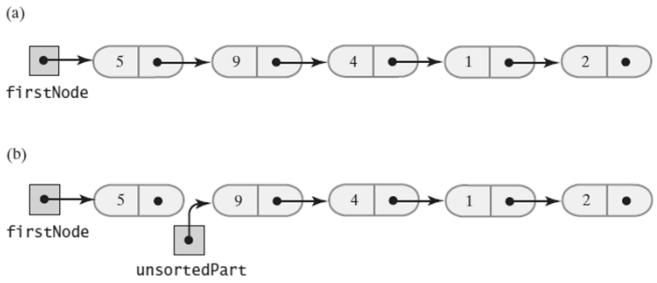
A chain of integers sorted into ascending order

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

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

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

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

Insertion Sort of a Chain of Linked Nodes

```
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
```

This class has an inner class **Node** *that has set and get methods*

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
```

This class has an inner class **Node** *that has set and get methods*

Insertion Sort of a Chain of Linked Nodes

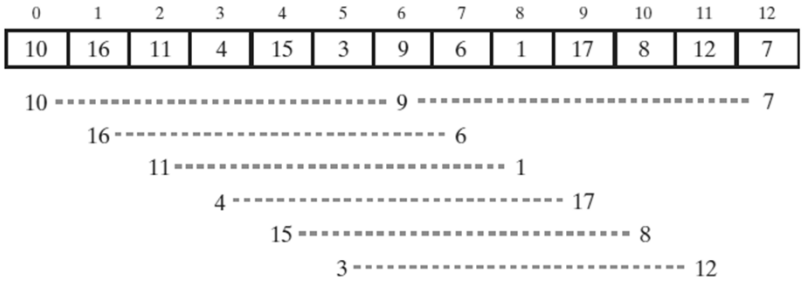
```
public void insertionSort()
{
   // If zero or one item is in the chain, there is nothing to do
   if (length > 1)
   {
       assert firstNode != null;
       // Break chain into 2 pieces: sorted and unsorted
       Node unsortedPart = firstNode.getNextNode();
       assert unsortedPart != null;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

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

The method to perform the insertion sort.

Shell Sort

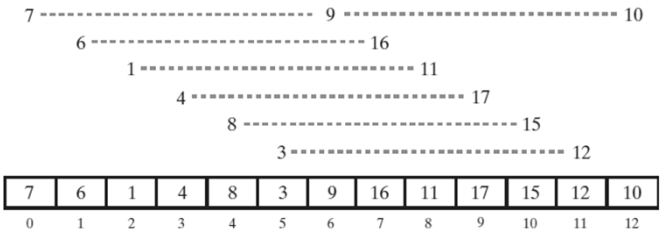
- Algorithms seen so far are simple but inefficient for large arrays at $O(n^2)$
- Note, the more sorted an array is, the less work **`insertInOrder`** must do
- Improved insertion sort developed by Donald Shell

Shell Sort



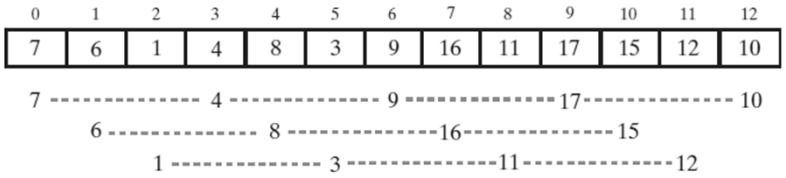
An array and the subarrays formed by grouping entries whose indices are 6 apart

Shell Sort

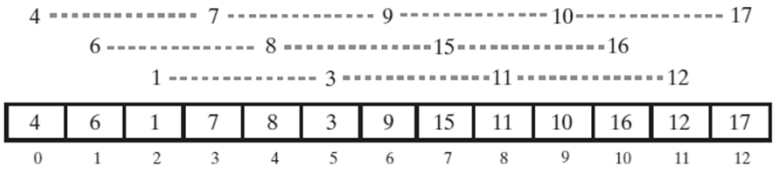


The subarrays after each is sorted, and the array that contains them

Shell Sort



The subarrays of the array formed by
grouping entries whose indices are 3 apart



The subarrays after each is sorted,
and the array that contains them

Shell Sort

```
Algorithm incrementalInsertionSort(a, first, last, space)
// Sorts equally spaced entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length;
// space is the difference between the indices of the entries to sort.

   for (unsorted = first + space through last at increments of space)
   {
      nextToInsert = a[unsorted]
      index = unsorted - space
      while ( (index >= first) and (nextToInsert.compareTo(a[index]) < 0) )
      {
         a[index + space] = a[index]
         index = index - space
      }
      a[index + space] = nextToInsert
   }
```

Algorithm that sorts array entries whose indices are separated by an increment of **`space`**.

Shell Sort

```
Algorithm shellSort(a, first, last)
// Sorts the entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length.

   n = number of array entries
   space = n / 2
   while (space > 0)
   {
      for (begin = first through first + space - 1)
      {
         incrementalInsertionSort(a, begin, last, space)
      }
      space = space / 2
   }
```

Algorithm to perform a Shell sort will invoke **`incrementalInsertionSort`** *and supply any sequence of spacing factors. Efficiency (worst) can be* $O(n^{1.5})$

Comparing the 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	$O(n)$	$O(n^{1.5})$	$O(n^2)$ or $O(n^{1.5})$

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