



---

## Ordered Structures

Wellesley College CS230  
Lecture 11  
Thursday, March 8  
Handout #21  
*Revised Friday, March 9*

Exam 1 due Friday, March 16

11-1



---

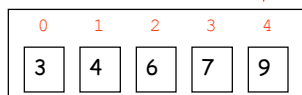
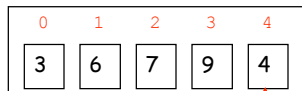
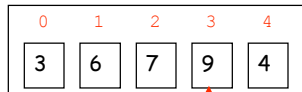
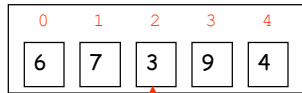
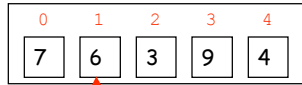
## Overview of Today's Lecture

1. Insertion sort on integer arrays.
2. Using `Comparable<T>` and `Comparator<T>` for more general orderings.
3. Some other approaches to sorting
4. Priority Queues = ordered queues in which maximal element is dequeued.

11-2



## Insertion Sort on Integer Arrays

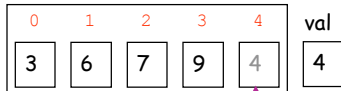


```
public static void insertionSort (int[] a) {
  for (int i = 1; i < a.length; i++) {
    insert(a, i); // Assume a[0..(i-1)] is sorted
                // Insert a[i] so that a[0..i] is sorted
  }
}
```

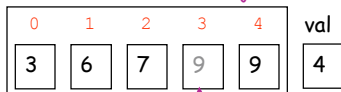
11-3



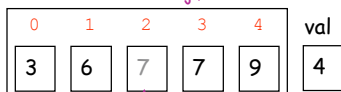
## Inserting One Element



val 4



val 4



val 4



val 4



val 4

```
// Assume a[0..(i-1)] is sorted
// Insert a[i] so that a[0..i] is sorted
public static void insert (int[] a, int i) {
```

}

11-4



## Insertion Sort in One Method

```
public static void insertionSort (int [] a) {  
    // Outer loop to process elements left-to-right  
    for (int i = 1; i < a.length; i++) {  
        int val = a[i];  
        int j = i;  
        // Inner loop to insert an element to left in sorted order  
        while ((j > 0) && (val < (a[j-1]))) { // Order of tests is crucial!  
            a[j] = a[j-1]; // Shift value right = shift hole left.  
            j = j-1;  
        }  
        // Now, either (j = 0) or ((j > 1) && (val >= a[j-1]))  
        a[j] = val;  
    }  
}
```

11-5



## Generalizing to an Array of Comparables

The Comparable<T> interface:

```
public interface Comparable<T> {  
    // Compare me to the otherGuy  
    public int compareTo (T otherGuy);  
}
```

A version of insertionSort() that works for any array of comparable elements:

```
public static <T extends Comparable<T>> void insertionSort (T[] a) {  
    for (int i = 1; i < a.length; i++) {  
        T val = a[i];  
        int j = i;  
        while ((j > 0) && (val.compareTo(a[j-1]) < 0)) { // Order of tests is crucial!  
            a[j] = a[j-1]; // Shift value right = shift hole left.  
            j = j-1;  
        }  
        // Now, either (j = 0) or ((j > 1) && (val.compareTo(a[j-1]) >= 0))  
        a[j] = val;  
    }  
}
```

11-6



## Testing insertionSort() on Comparables

```
[fturbak@puma utils] java ArrayOps insertionSort "{I,do,not,like,green,eggs,and,ham}"  
Before: a = {"I", "do", "not", "like", "green", "eggs", "and", "ham"};  
insertionSort(a);  
After: a = {"I", "and", "do", "eggs", "green", "ham", "like", "not"}
```

```
[fturbak@puma utils] java ArrayOps insertionSort "{i,do,not,like,green,eggs,and,ham}"  
Before: a = {"i", "do", "not", "like", "green", "eggs", "and", "ham"};  
insertionSort(a);  
After: a = {"and", "do", "eggs", "green", "ham", "i", "like", "not"}
```

11-7



## Using an Explicit Comparator

The `Comparator<T>` interface:

```
public interface Comparator<T> {  
    public int compare (T x, T y); // Compare x to y  
}
```

A version of `insertionSort()` that takes an explicit comparator:

```
public static <T> void insertionSort (Comparator<T> comp, T[] a) {  
    for (int i = 1; i < a.length; i++) {  
        T val = a[i];  
        int j = i;  
        while ((j > 0) && (comp.compare(val, a[j-1]) < 0)) { // Order of tests is crucial!  
            a[j] = a[j-1]; // Shift value right = shift hole left.  
            j = j-1;  
        }  
        // At this point, either (j = 0) or ((j > 1) && (comp.compare(val, a[j-1]) >= 0))  
        a[j] = val;  
    }  
}
```

11-8



## Another String Comparator

Here's a sample nonstandard String comparator. Shorter strings are considered "less" than longer strings; strings of the same length are compared alphabetically.

```
class LengthComparator implements Comparator<String> {  
    public LengthComparator() {} // Can omit this constructor (Java will supply it).  
    public int compare (String s1, String s2) {  
        if (s1.length() < s2.length()) return -1;  
        else if (s1.length() > s2.length()) return 1;  
        else return s1.compareTo(s2);  
    }  
}
```

Let's try it out:

```
[fturbak@puma utils] java ArrayOps insertionSortLengthComparator  
    "{I,do,not,like,green,eggs,and,ham}"  
Before: a = {"I","do","not","like","green","eggs","and","ham"};  
insertionSort(new LengthComparator(), a);  
After: a = {"I","do","and","ham","not","eggs","like","green"}
```

11-9



## Some Other Comparators

Here are some other Comparator classes:

```
public class ComparableComparator<T extends Comparable<T>> implements Comparator<T> {  
    // Constructs a comparator that compares Comparables  
    public ComparableComparator () {} // Can omit this constructor, since Java will supply it.  
    public int compare (T x, T y) { return (x.compareTo(y)); }  
}  
  
public class ComparatorInverter<T> implements Comparator<T> {  
    private Comparator<T> comparator; // Instance variable that remembers comparator to invert  
    // Constructs a comparator that inverts the sense of comp  
    public ComparatorInverter (Comparator<T> comp) { comparator = comp; }  
    public int compare (T x, T y) { return comparator.compare(y,x); } // Invert comparator's comparison.  
}
```

Let's see them in action:

```
[fturbak@puma utils] java ArrayOps insertionSortInverted  
    "{I,do,not,like,green,eggs,and,ham}"  
Before: a = {"I","do","not","like","green","eggs","and","ham"};  
insertionSort(new ComparatorInverter<String>(new ComparableComparator<String>()),a);  
After: a = {"not","like","ham","green","eggs","do","and","I"}  
11-10
```



## A Priority Queue (PQueue) Interface

---

```
public interface PQueue<T> extends Queue<T> {
    public Comparator<T> comparator();
        // Returns comparator used by this pqueue to determine priority.
        // Returns null if the natural Comparable ordering is being used.
    public void enq(T elt); // Add elt to this pqueue.
    public T deq(); // Remove and return highest-priority element from this pqueue.
    public T front(); // Return highest-priority element from this pqueue
        // without removing it.
    public Iterator<T> iterator();
        // Return an iterator that yields this pqueue's elements from
        // highest priority to lowest priority.
    public PQueue<T> copy(); // Return a shallow copy of this pqueue
    public String toString(); // Return string listing elements of this pqueue
        // from highest priority down.

    // Inherits the following methods with their usual behavior from Queue<T>:
    // boolean isEmpty(), int size(); void clear();

    // A pqueue implementation should also provide two constructor methods:
    // 1. A nullary constructor method in which the Comparator is assumed null
    // 2. A unary constructor method explicitly supplying the Comparator method
}
```

11-11