Today we shift our emphasis from 'Object Oriented Design' to 'Data Structures and Algorithms.' We use a data structure that you are familiar with – an array – to maintain an ordered set of items.

In Chapter 3.1 of the textbook, an example is developed in which one keeps track of player scores in a game. A class `GameEntry` is defined which contains two fields: a person's name and the person's score in the game. A second class `Scores` is defined which keeps an array of type `GameEntry` that holds the entries with the top `maxEntries` scores, where `maxEntries` is a constant declared in the `Scores` class. Java code is given that allows one to keep track of a set of entries, for example, there are methods to `add` an entry, and `remove` an entry. You can find this code on the textbook's website. `http://ww0.java4.datastructures.net/source/`

The class `Scores` has a constructor, as well as two methods `add()` and `remove()` which add and remove an entry, respectively. The array of entries is initialized to being empty. As entries are added or removed, the array always maintains an ordering from highest to lowest. So entry at index 0 has the highest score. The class `Scores` has a field `numEntries` which keeps track of how many entries there are in an array `entries`.

In the lecture, I simplified this example by letting the array `entries` be an array of `double` values. The reason for this simplification is that it allows us to focus more on the sequence of operations that are involved in `add()` and `remove()` methods, i.e. algorithms and not get distracted by the Object Oriented stuff.

In presenting the methods/algorithms below, I went slowly through an example that showed what happens when there is a certain sequence of adds and removes. You should run this code and make sure you understand it. Here I briefly summarize the method and then give a compressed listing of the code. More complete code can be found on the public course web page.

add( )

Suppose that the array `entry` contains `numEntries` values where `numEntries` is some number in the set $\{0, \ldots$ `maxNumEntries`$\}$. We would like to add a new entry (a `double`) and put it in its correct order in the array. There are two cases to consider:

- `numEntries < maxNumEntries`: in this case we need to do two things: we need to increment `numEntries`, and we need to insert the new entry. This requires shifting the position of any *smaller* entries by one position to make room for the new entry in its correct position.

- `numEntries == maxNumEntries`: in this case, we only add the new entry if it is bigger than the smallest entry in the array. We are assuming the entries are sorted from largest (at index 0) to smallest (at entry `maxEntries-1`) and so we need to check if the new entry is larger than the entry at position `maxEntries-1`). If it is, then we need to find the correct place to insert it, shift the smaller elements by one, and insert the new entry at its correct place. Note that the smallest entry in the array will be removed.

The code follows on the next page.

```
public void   add(double e){

   if (numEntries == maxNumEntries){
      if (e < entries[numEntries-1])
        return;
   }
   else numEntries++;

   int  cur = numEntries - 1;   //     cur  >=  0

   while ((cur > 0) && (e > entries[cur - 1])){
      entries[cur] = entries[cur - 1];
      cur--;
   }
   entries[cur] = e;
}
```

remove( )

Next consider the method for removing an element at a particular index (input parameter of type `int`). There are three steps:

- Store the element in a temporary variable. You need to do this because eventually you return the element – that's how the `remove()` method is defined!
- Shift all the smaller elements in the array. Note the shift here is in the opposite direction as in the `add()` method.
- Decrement the `numEntries` counter.

And here is the code:

```
public double remove(int i){

//   throws an exception if index is invalue but
//   I am not including that code here.

    double tmp = entries[i];

    for (int cur = i; cur < numEntries-1; cur++){
      entries[ cur ] = entries[ cur +1];
    }
    numEntries--;
    return tmp;
}
```

insertionSort

The `add` and `remove` methods always maintain a sorted array, so we do not need a separate method for sorting a given array.

But let's instead suppose we are given an array which is not sorted. We would write a method which sorts this array, which uses the same idea as the `add` method.

The idea is to consider the first i elements of the array and let's *suppose they are already sorted.* (This would be the case, for example, if $i = 0$ or 1. If the first i elements are laready sorted, then we can insert the element $i + 1$ using the same steps as used in the `add` method. (The `add` method assumes we have a sorted array of elements.) Thus, by incrementing i from 0 (actually, from 1 – see code below), we can sort the entire array.

```
public void insertionSort(){

//  Here I do not include the Exception checking code.
//  See online code next to these lecture notes for
//  complete code.

  int cur;
  double tmp;
  for (int i=1; i < numEntries; i++){
     tmp = entries[i];
     cur = i;

//  Similar code to add()

    while ((cur > 0) && (tmp > entries[cur - 1])){
       entries[cur] = entries[cur - 1];
      cur--;
    }
    entries[cur] = tmp;
 }
}
```