**Let’s make some Heaps**

We are going to implement a MaxHeap (and later a MinHeap) using an ArrayList.

**Use the Schaffer implementation as the template (see JAVA3elatest.pdf, Chapter 5.5, page 170).** He uses an Array, I am requiring you to use an ArrayList. We are using an ArrayList so the Heap will dynamically grow. I am expecting your implementation will have the removemax method. Along with the other methods he describes.

Along with the methods Schaffer describes, you will also have to implement:

* A default constructor

public MaxHeap(){

…

}

* A swap method that will allow you to swap items in the ArrayList
  + The chapter refers to a DSUtil.swap method – just make your own.
* An isEmpty method that will return true if the MaxHeap is empty, false otherwise.
* A toString method, that returns a String that represents the contents of the filled indices of the ArrayList.

Heaps have two properties:

* Complete binary trees
* Partial Ordering

A complete binary tree means all levels except possibly the last are completely full, and the last level has all its nodes filled from the left to right.

If we number the nodes from 0 at the root and place:

* the left child of node *k* at position 2*k* +1
* the right child of node *k* at position 2*k*+2
* parent of node k is at position (k-1)/2 (using integer division)

Then the 'fill from the left' nature of the complete tree ensures that the heap can be stored in consecutive locations in an ArrayList.

The values stored in a heap are partially ordered. This means that there is a relationship between the value stored at any node and the values of its children. A max-heap has the property that every node stores a value that is ***greater*** than or equal to its children. On the other hand, a min-heap has the property that every node stores a value that is ***less***than or equal to that of its children.

When you insert into a heap, you add the value to the last place in the ArrayList. The value is then percolated up the heap through a series of comparisons and swaps. You compare the newly added value to its parent. For a **max-heap**, if it is ***greater*** than its parent, you swap the parent and child values. This process is repeated until the newly added value is no longer greater than the parent or it is the new root. For a **min-heap**, if it is ***less*** than its parent, you swap the parent and child values. This process is repeated until the newly added value is no longer less than the parent or it is the new root.

When you remove, you always remove the root (position 0). The value in the last position in the array is placed in the root and then sifted down the heap to the proper location through a series of comparisons and swaps.

I have provided you with a JUnit test file, TestMaxHeap.java and a Cat.java file. It makes MaxHeaps with Strings, Integers, and Cats and tests basic MaxHeap functionality. Make sure your MaxHeap passes the JUnit tests in the file.

**Next, make a MinHeap**

Now – implement a MinHeap. You will need a removemin method instead of a removemax method. You will have the same remaining methods as MaxHeap.

I have provided you with a JUnit test file, TestMinHeap.java and a Cat.java file. It makes MinHeaps with Strings, Integers, and Cats and tests basic MinHeap functionality. Make sure your MinHeap passes the JUnit tests in the file.