1. In this lab we will be writing the code to implement a heap as an array, and then use the heap to sort data. We will want to compare this method of sorting empirically with other sorting algorithms in subsequent labs, so we will want to have a common starting point. So our first step will be to get data to be sorted all into an array, and that array will then be “read from” by each of the sorting algorithms. Today we will only work with heap sort. So begin by creating a new Java project called Sorting, and in that project create a new package called heapsort. In that package, create a new class called Heap. Add protected member variables that are arrays of integers called array and heap. Also add a protected integer member variable called nextAvailable. Add a constructor that takes an integer argument representing the size of the arrays, and that instantiates both arrays to the size of that argument. Add a method called randomInit that initializes array to random integers between zero and the size of the array. Add a JUnit test for this method that makes sure the random numbers are in the range you expect. The array array will contain random numbers, but the array heap will be constructed as a heap from the values that are already in array.

2. Create methods called leftChild, rightChild, and parent that take integer arguments and return integers. These methods will compute the subscript of the array that contains the appropriate tree value for the given argument. Begin by writing JUnit test methods for each of these. Here are some facts you can use. Add more as needed. leftChild(0) returns 1, rightChild(0) returns 2, leftChild(1) returns 3, rightChild(1) returns 4, leftChild(2) returns 5, parent(1) returns 0, parent(2) returns 0, parent(3) returns 1, parent(0) throws an illegal argument exception. Implement the methods using the formulas you saw in class (or that you should be able to derive for yourself).

3. We are now ready to start building the heap from the array of random values. Create a new method called buildHeap that sets nextAvailable to zero, and then for each element of array, places it into the next available element of heap, increments nextAvailable, and calls a method called restoreHeapAfterInsert that is yet to be written. Create the restoreHeapAfterInsert method, adding to it a local variable called current initially set to nextAvailable-1, and a loop that continues as long as current is not the root and the data in the parent of the current node is bigger than the data in the current node. Inside that loop, swap the current node and its parent, then change current to the parent subscript. At this point your buildHeap method should work, and you can make a JUnit test for it by creating a Heap object of 100 elements, calling randomInit and buildHeap, and then asserting that each node has data that is less than or equal to the data in its children.

4. After filling array with random integers and running buildHeap, all of the random integers will be in the heap array, which will satisfy the minheap property. All we need to do is sort the data is to repeatedly remove the smallest node, which is at the root of the heap, put the last node data into the root, and restore the minheap property. If you feel especially confident about your programming abilities at this point, you do not need to read the rest of this lab. Just write a method sortFromHeap (and any helper methods it needs) that places the sorted elements back in array, and test it by making sure array is sorted afterwards. You can write the test now, even if you read on.
5. It will be useful to have a helper method called `getSmaller` that takes two integer arguments representing subscripts, compares the corresponding elements of the `heap` array, and returns the subscript of the smaller one. But there is an extra catch: we would like this method to work even if one of the subscripts is out of bounds, in which case it should just return the subscript that is in bounds. If both subscripts are out of bounds, it should throw an illegal argument exception. The size of the heap as it is being built is determined by `nextAvailable`. Here is an example of a test for this method. Instantiate the `heap` array with 8 elements, set `nextAvailable` to 8, and fill `heap` with the same values as the subscripts, but then set `heap[5]` to 17. Draw on paper the tree this represents so you can see what is going on. `getSmaller` should return 1 if you give it 1 and 2, it should return 3 if you give it 3 and 7, it should return 6 if you give it 5 and 6, and it should return 7 if you give it 7 and 8. Make sure you understand this method and that it works right, because you will find it very helpful.

6. Write a method called `sortFromHeap` that contains a loop that goes through all of `array` and does the following: places the first element of `heap` into the current element of `array`, places the last element in `heap` at the top of the heap, decreases `nextAvailable`, and calls a method yet to be written called `restoreHeapStartingAt` that takes an integer argument. Also create a JUnit test for `sortFromHeap` that creates a `Heap` object, initializes the array to random integers, builds a heap, calls `sortFromHeap`, and asserts that `array` is now sorted.

7. It is probably easiest to make `restoreHeapStartingAt` a recursive method. The base case comes up when the argument passed in refers to a leaf, and in that case you can just have the method return. Otherwise, the method can use `getSmaller` to find the smaller of the two possible children, and if the node passed has bigger data than the smaller of the two children, swap those two nodes and make a recursive call passing in the smaller subscript. That’s it. Make sure this all works before the next lab.