# Computer Science 112 - Fundamentals of Programming II
# Lab Project 8
**Due on Github 11:59pm Monday 15 November**

## **Part I:** Heap Heap Hooray!

Create a file **heap.py** defining a **`Heap`** class. As with our other container classes, you'll want an **`add`** method and a __**`contains`**__ method as the minimal API for this class, plus a __**`str`**__method to aid in debugging. Feel free to copy/paste from the lecture slides (slide deck 28) to get your heap working. To keep things simple, your __**`str`**__method can just return a string representation of the heap's array. (I tried having __**`str`**__ return a string showing the hierarchy like a BT, but I didn't get too far and gave up: maybe an extra-credit opportunity!)

## **Part II:** Priority Queue Redux

As we discussed, the efficient way to implement a priority queue is with a heap. To get started on your new priority queue, add a **pop** method to your **Heap** class, using the code from the lecture notes.

Next we'll want a new **`PriorityQueue`** class to compare with the one we built in Lab 5 a few weeks ago. What I did was to copy the **priorityqueue.py** file from that lab into this lab and rename it to **priorityqueue2.py**, and replace the **`LinkedList`** code with code from my new **`Heap`** class. Doing that should involve simply replacing your complicated **`PriorityQueue`** code with one-line calls to the **`Heap`** methods; for example:

```
    def push(self, datum):

        self.heap.add(datum)
```

After running some simple tests as usual, you're ready to have your new and old priority queue classes go head-to-head in an efficiency contest. (Don't forget to copy your **priorityqueue.py** and its supporting **linkedlist.py** to this lab folder to enable the comparison.) Create a new file **pqtest.py** to run this comparison. By making good use of the import statement, you can refer to your two queue implementations elegantly in your main code:

```
from priorityqueue import PriorityQueue as PQ1
from priorityqueue2 import PriorityQueue as PQ2
```

As you did last week, code up a little timing test to show the difference in cost associated with inserting an element into each of these queue implementations, making sure to put a helpful **`print`** statement before any code that you know is going to take a nontrivial amount of time to run.

## **Part III:** BucketSort with Charlie the Parrot

Implement the BucketSort algorithm we described in class. Write **bucketsort.py** containing (for now) a single function **`bucketSort`** (no class definition necessary), which will sort (mutate) the integer list you pass to it.

Your **`LinkedList`** class will be helpful for implementing the buckets. Instead of using the code from the slides, I found it easier to code up **`bucketSort`** myself from first principles, avoiding the **Arrays** class and other complications from the lecture slides (Lecture 29, Slide 18). Instead, I used an ordinary Python list for my array and followed the general algorithm in the slide.

Once you've tested your **`bucketSort`** function on some integer lists you build by hand (make sure to include some duplicates!), it's time to try it out on a list of random integers. To help with these, write a function **`randints(n)`** that accepts a numbers *n* and returns a list of *n* random integers in the interval [0, *n*-1]. Test your **`bucketSort`** again on a small list created by using this function.

Finally, as you may have expected, it's time to run some speed contests! Since the claim is that BucketSort can be faster than QuickSort, we'll compare these two sorting algorithms. Copying your **sorts.py**, **counter.py**, and **tools.py** files from Lab #3 will give you access to QuickSort, by adding **`from sorts import quickSort`** at the top of **bucketsort.py.** To compare the two sorting functions, use your **`randints`** function to generate a large list of random numbers. Then make a copy of this list, so you can give one copy to **`quicksort`** and the other to **`bucketSort`**. In reporting the time values, I avoided using **`int()`** to convert the elapsed time to a whole number of seconds; otherwise, the differences were too small to notice.

As I discovered through some googling, Quicksort is the best choice in general, but BucketSort can win when you have a small list. To figure out what size list counts as "small", run some experiments with different values of *n*. For example, you could start with *n* = 1000 and double *n* until you seem some interesting results.

To finish up, summarize your results in a plot using Excel (or Numbers on a Mac, or whatever spreadsheet program you like). Your plot should show time in seconds against list size *n*, with a cross-over for some value of *n*. Include your plot as a PDF in your submission. Another extra-credit opportunity would be to have your **bucketsort.py** program create a .CSV file that you can open in a spreadsheet program, so you don't have to copy the numbers into the spreadsheet by hand.

*NOTE: No parrots were harmed in the making of this assignment.*