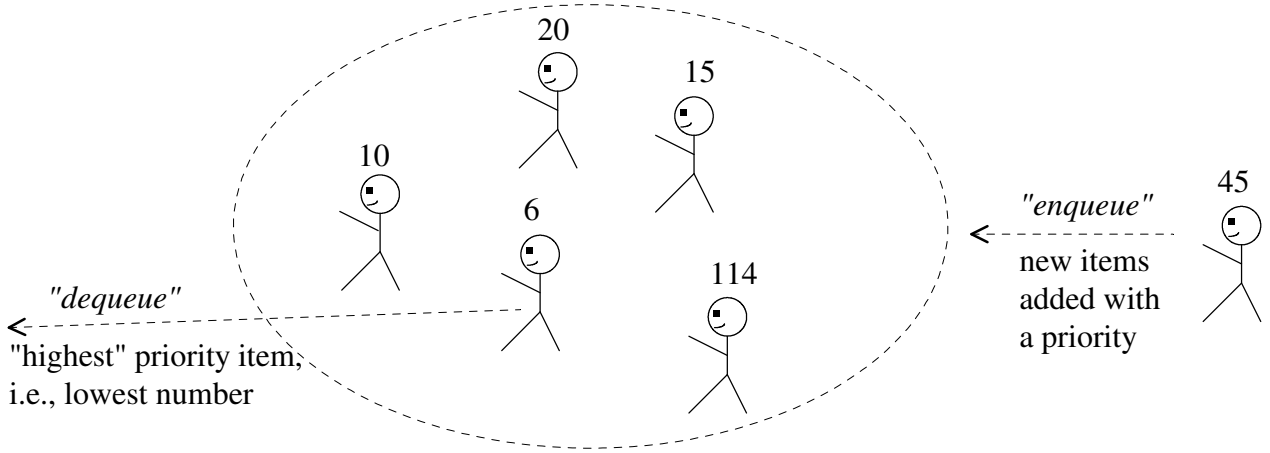
Data Structures (CS 1520) Lab 7 Name:_____

Objective: To understand priority queue implementations in Python including linked and heap-based array implementations (including being able to determine the big-oh of each operation), and to practice recursion.

Background: Read sections 15.6 and 18.9 - 18.11 from the Lambert text. A priority queue is NOT a FIFO queue. Instead, each item added to a priority queue has an associated priority. When a dequeue occurs, the item with the highest associated priority is returned. (Unfortunately, the textbook defines the highest priority to be the lowest valued priority.)



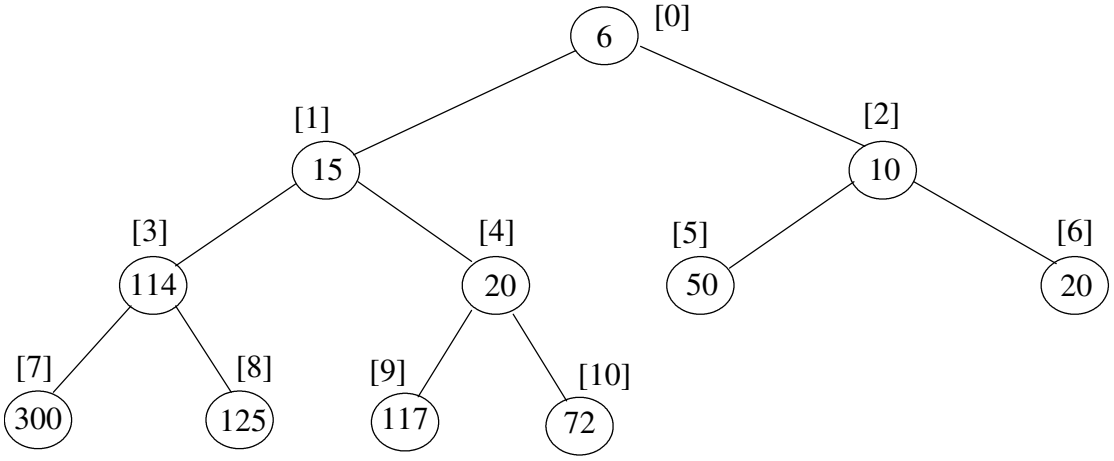
To start the lab: Download and unzip the file lab7.zip

Part A: The `queue.py` file contains a subclass of `LinkedQueue`, called `LinkedPriorityQueue`, which maintains the items in the linked list in sorted order by priority. To do this, the enqueue method searches and inserts the new item into the correct spot based on its priority.

a) An advantage of this implementation is that `LinkedPriorityQueue` can inherit all of `LinkedQueue` with only the `enqueue` method being overridden. What would be the expected big-oh notation for the `LinkedPriorityQueue`'s `enqueue` method? Assume "n" items in the queue.

b) What would be the expected big-oh notation for the `LinkedPriorityQueue`'s `dequeue` method inherited from `LinkedQueue`? Assume "n" items in the queue.

From lecture (Sections 18.9 - 18.11) recall the very "non-intuitive", but powerful list/array-based approach to implement a priority queue, called a *heap*. The list/array is used to store a *complete binary tree* (a full tree with any additional leaves as far left as possible) with the items being arranges by *heap-order property*, i.e., each node is less than either of its children. An example of a min-heap "viewed" an a complete binary tree would be:



c) For the above heap, the list/array indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:

- its left child if it exists:
- its right child if it exists:
- its parent if it exists:

d) Because of the heap-order property, where would the smallest node in the heap be located?

e) Modify the above heap to show the result after adding 3?

f) In general, what would be the height of a heap containing n nodes?

g) Use the file `timePriorityQueues.py` to time the enqueuing of 40,000 items onto a priority queue and then dequeuing them off. Complete the following timing table:

	Time (seconds) to enqueue 40,000 items	Time (seconds) to dequeue 40,000 items
LinkedPriorityQueue		
HeapPriorityQueue		

h) Explain why enqueuing into the LinkedPriorityQueue is so much slower than enqueuing into the HeapPriorityQueue.

i) Examine the code for the HeapPriorityQueue to explain why it takes less time to enqueue the same number of elements than to dequeue them?

j) The pop code for the heap is straight from the textbook. What is strange about the variable names used in the code?

k) Fix the variable names to better reflect their usage.

After answering the above questions and fixing the variable names, raise you hand and explain your answers.

Part B. Below is the textbook's partial `ArrayHeap` class showing the `add` method.

```
class ArrayHeap(object):

    def __init__(self):
        self._heap = []

    def add(self, item):
        self._heap.append(item)
        curPos = len(self._heap) - 1
        while curPos > 0:
            parent = (curPos - 1) / 2
            parentItem = self._heap[parent]
            if parentItem <= item:
                break
            else:
                self._heap[curPos] = self._heap[parent]
                self._heap[parent] = item
                curPos = parent
    def addRec(self, item):

        def siftUp(curPos):









        # start of add method's code
        self._heap.append(item)      # add item as leaf
        siftUp(len(self._heap) - 1)  # call siftUp to move iten to correct spot
```

You are to complete the `addRec` method which uses a **recursive** `siftUp` function to move the new item to its correct spot in the heap. Your `siftUp` function should have one recursive case:

- if item is not at the root already and the parent > item, then move parent down to the item's current position and siftUp the item from the parent's position.

If it is not a recursive case, i.e., it is a base case, then we do not need to do anything since the item is at the correct spot in the heap.

After completing and testing your addRec method, raise your hand and demonstrate your code.

Part C. EXTRA CREDIT. Below is the textbook's partial `ArrayHeap` class showing the a partial `popRec` method. You are to complete the `popRec` method which uses a **recursive** `siftDown` function to move the item dumped at the root to its correct spot in the heap.

```
class ArrayHeap(object):

    def __init__(self):
        self._heap = []

    def popRec(self):

        def siftDown(curPos):
```

```
       # Start of popRec code
       if self.isEmpty():
            raise Exception, "Heap is empty"

        topItem = self._heap[0]
        bottomItem = self._heap.pop(len(self._heap) - 1)
        if len(self._heap) == 0:
            return bottomItem

        self._heap[0] = bottomItem
        lastIndex = len(self._heap) - 1

        siftDown(0)
        return topItem
```

After completing and testing your popRec method, raise your hand and demonstrate your code.