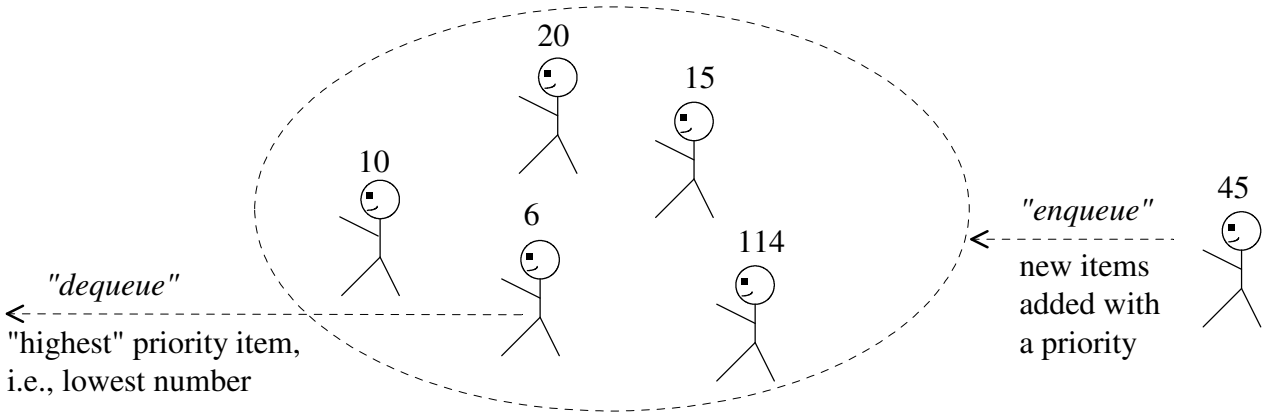
Data Structures (810:052) Lab 6 Name:_____

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.

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 lab6.zip

Two problems occur when implementing a priority queue:

1) some items don't have a nature priority associated with them, so a Comparable wrapper class can be used to associate a priority. The code for the Comparable class (section 15.6) is:

```
class Comparable(object):
    """Wrapper class for items that are not comparable."""

    def __init__(self, item, priority):
        self._item = item
        self._priority = priority

    def __cmp__(self, other):
        if type(other) != type(self):
            raise TypeError, "Type must be Comparable"
        return cmp(self._priority, other._priority)

    def getItem(self):
        return self._item

    def __str__(self):
        return str(self._item)
```

2) the items must be ordered by priority on enqueuing or items must be search by priority on dequeuing.

Part A of the Lab looks at:

- `LinkedPriorityQueue` - which maintains the items in the linked list in sorted order by priority
- Array-based priority queue maintaining the items in the Array in sorted priority order
- Array-based priority queue maintaining the items in the Array in NO order

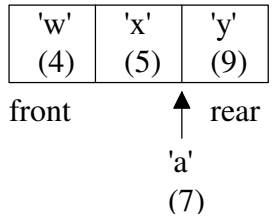
Part B of the Lab looks at:

- `HeapPriorityQueue` - uses a partially ordered Python list in a heap

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. For example, consider enqueuing 'a' with a priority of 7 into the following priority queue, `myQueue` by the method call:

```
myQueue.enqueue(Comparable('a', 7)):
```

"Abstract Priority Queue myQueue"
(priorities in parenthesis)



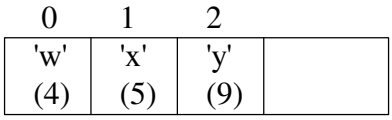
a) An advantage of this implementation is that `LinkedPriorityQueue` can inherit all of `LinkedQueue` with only the `enqueue` method being overridden. What would be the expected big-oh notation for the `LinkedPriorityQueue`'s `enqueue` method? Assume "n" items in the queue.

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

c) An intuitive Array-based priority queue implementation would involve either:

I. maintaining the items in the Array in sorted priority order, e.g.,

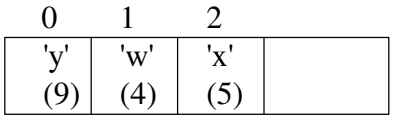
"Abstract Priority Queue"
(priorities in parenthesis)



Enqueuing a new item would be very much like the inner-loop of insertion sort, i.e., scan the "sorted" items from right-to-left looking for (and shifting to the right) the spot where to insert.

II. maintaining the items in NO order (and just add a new item to the right end of the Array), e.g.,

"Abstract Priority Queue"
(priorities in parenthesis)



Dequeuing involves searching for the "highest" priority item and removing it by sliding all the item to its right "down" to fill the hole.

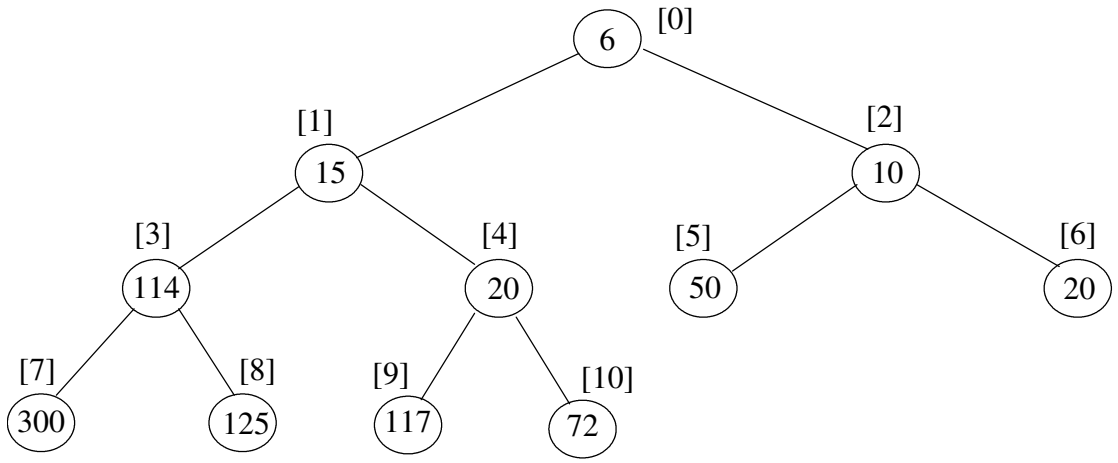
(**You DO NOT need to actual implement them**.)

Complete the expected big-oh notation for each queue method using the above Array-based implementatons. (**You DO NOT need to actual implement them**.)

Implementation	`dequeue()`	`enqueue(item)`	`peek()`	`isEmpty()`	`__len__()`	`__str__()`
I. sorted items						
II. unsorted items						

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

<u>Part B</u>: 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 heap "viewed" an a complete binary tree would be:



a) 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:

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

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

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

e) 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		

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

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

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

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

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