1. Section 6.6 discusses a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary 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 arranged by heap-order property, i.e., each node is ≤ either of its children. An example of a min heap "viewed" an a complete binary tree would be:

```
Python List actually used to store heap items
[114, 15, 20, 20, 50, 300, 125, 117, 18, 13, 12]
```

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: 
  \[ \left\lfloor \frac{i}{2} \right\rfloor \]
- its right child if it exists: 
  \[ 2i + 1 \]
- its parent if it exists: 
  \[ \frac{i}{2} \]

b) What would the above heap look like after inserting 13 and then 3? (show the changes on above tree)

General Idea of insert(newItem):
- append newItem to the end of the list (easy to do, but violates heap-order property) \( O(1) \)
- restore the heap-order property by repeatedly swapping the newItem with its parent until it percolates to correct spot

c) What is the big-oh notation for inserting a new item in the heap? 
\( O(\log n) \)

\( O(\log n) \)

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self, currentIdx):
        parentIdx = int(currentIdx / 2)
        while currentIdx > 1 and self.heapList[currentIdx] < self.heapList[parentIdx]:
            temp = self.heapList[currentIdx]
            self.heapList[currentIdx] = self.heapList[parentIdx]
            self.heapList[parentIdx] = temp
            currentIdx = parentIdx

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
```
2. Now let us consider the `delMin` operation that removes and returns the minimum item.

Python List actually used to store heap items:

```
0  15  10  114  20  20  50  300  125  117
```

a) What item would `delMin` remove and return from the above heap? 
6 at [C]

b) What is the quickest way to fill the hole left by `delMin`? with item on right end (117)

c) What new problem does this cause? 117 at root violates heap-order property

General Idea of `delMin()`:
- remember the minimum value so it can be returned later (easy to find - at index 1)
- copy the last item in the list to the root, delete it from the right end, decrement size
- restore the heap-order property by repeatedly swapping this item with its smallest child until it percolates down to the correct spot
- return the minimum value

\[ O(\log n) \]

d) What would the above heap look like after `delMin`? (show the changes on above tree)

```
def percDown(self, currentIndex):
    while currentIndex * 2 <= self.currentSize:
        minChildIndex = self.minChild(currentIndex)
        if self.heapList[currentIndex] > self.heapList[minChildIndex]:
            self.heapList[currentIndex], self.heapList[minChildIndex] = 
            temp = self.heapList[currentIndex]
            self.heapList[currentIndex] = self.heapList[minChildIndex]
            self.heapList[minChildIndex] = temp
        else:
            return
```

e) Complete the code for the percDown method used by `delMin`.

f) What is the big-oh notation for `delMin`? \( O(\log n) \)
Once we have a working BinHeap, then implementing the PriorityQueue class using a BinHeap is a piece of cake:

```python
>>> q = PriorityQueue()
>>> print(q)
[]
>>> q.enqueue(5)
[5]
>>> q.enqueue(1)
[1, 5]
>>> q.enqueue(7)
[1, 5, 7]
>>> q.dequeue()
1
>>> print(q)
[5, 7]
```

3. A "list" is a generic term for a sequence of items in a linear arrangement. Unlike stacks, queues and deques access to list items is not limited to either end, but can be from any position in the list. The general terminology of a list is illustrated by:

```
"Abstract view of a list"                 [0] [1] [2] [3] ← index/position in the list

head                                 'w' 'a' 'y' 'c'

                                    ↓

                      tail
```

There are three broad categories of list operations that are possible:

- **index-based operations** - the list is manipulated by specifying an index location, e.g.,
  ```python
  myList.insert(3, item)  # insert item at index 3 in myList
  ```

- **content-based operations** - the list is manipulated by specifying some content (i.e., item value), e.g.,
  ```python
  myList.remove(item)       # removes the item from the list based on its value
  ```

- **cursor-based operations** - a cursor (current position) can be moved around the list, and it is used to identify list items to be manipulated, e.g.,
  ```python
  myList.first()            # sets the cursor to the head item of the list
  myList.next()             # moves the cursor one position toward the tail of the list
  myList.remove()           # deletes the second item in the list because that's where the cursor is currently located
  ```

The following table summarizes the operations from the three basic categories on a list, L:

<table>
<thead>
<tr>
<th>Index-based operations</th>
<th>Content-based operations</th>
<th>Cursor-based operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.insert(index, item)</td>
<td>L.add(item)</td>
<td>L.hasNext()</td>
</tr>
<tr>
<td>item = L[index]</td>
<td>L.remove(item)</td>
<td>L.next()</td>
</tr>
<tr>
<td>L[index] = newValue</td>
<td>L.search(item)</td>
<td>L.hasNext()</td>
</tr>
<tr>
<td>L.pop(index)</td>
<td>i = L.index(item)</td>
<td>L.previous()</td>
</tr>
</tbody>
</table>

Built-in Python lists are unordered with a mixture of index-based and content-based operations. We know they are implemented using a contiguous block of memory (i.e., an array). The textbook talks about an unordered list ADT, and a sorted list ADT which is more content-based. Both are implemented using a singly-linked list. Node

a) Why would a singly-linked list be a bad choice for implementing a cursor-based list ADT?