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 as complete binary tree would be:

```
   1
  / \
 [2] [3]
/   /
[4] 15
|
[5]
|
[6]
|
[7]
300 125 117 20 20 50 300 125 117
```

Python List actually used to store heap items

```
[6 15 10 114 20 20 50 300 125 117]
```

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: \(i \times 2\)
- its right child if it exists: \(i \times 2 + 1\)
- its parent if it exists: \(i \div 2\)

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)
- 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? \(O(\log n)\)

d) Complete the code for the percUp method used by insert.

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
    
    def percUp(self, currentIdx): 
        parentIdx = currentIdx // 2 
        while currentIdx > 0 and self.heapList[currentIdx] < 
            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.

<table>
<thead>
<tr>
<th>Python List actually used to store heap items</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 15 10 114 20 20 50 300 125 117</td>
</tr>
</tbody>
</table>

a) What item would `delMin` remove and return from the above heap? **6**
b) What is the quickest way to fill the hole left by `delMin`? **move right most item to index 1**
c) What new problem does this cause? **moved item 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

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

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

```python
class BinHeap:
    def minChild(self, i):
        if i * 2 + 1 > self.currentSize: # only left child
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

def delMin(self):
    retval = self.heapList[1]
    self.currentSize = self.currentSize - 1
    self.heapList[0] = self.HeapifyDown(1)
    return retval

def percDown(self, current_index):
    while True:
        if current_index * 2 > self.currentSize:
            return
        minChild = self.minChild(current_index)
        if self.heapList[current_index] <=
            self.heapList[minChild]:
            return
        self.heapList[current_index],
        return
```

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
### File: priority_queue.py
from binheap import BinHeap

class PriorityQueue:
    def __init__(self):
        self._heap = BinHeap()

    def isEmpty(self):
        return self._heap.isEmpty()

    def enque(self, item):
        self._heap.insert(item)

    def deq(self):
        return self._heap.delMin()

    def size(self):
        return self._heap.size()

    def __str__(self):
        return str(self._heap)

>>> q = PriorityQueue()
>>> print(q)
[
>>> q.enqueue(5)
>>> q.enqueue(1)
>>> q.enqueue(7)
>>> print(q)
[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

    "w"   "a"   "y"   "c"

    head   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.,
  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.,
  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.,
  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] = new_value</td>
<td>L.search(item) #return Boolean</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.

a) Why would a singly-linked list be a bad choice for implementing a cursor-based list ADT?
def percolup(self, current_index):
    parent_index = current_index // 2
    while parent_index >= 1 and
        self.heap_list[current_index] <
        self.heap_list[parent_index]:
        temp = self.heap_list[current_index]
        self.heap_list[current_index] =
        self.heap_list[parent_index]
        self.heap_list[parent_index] = temp
        current_index = parent_index
        parent_index = current_index // 2
```python
def percDown(self, current_index):
    while True:
        if current_index // 2 > self.currentSize:
            return
        else:
            minChildIndex = self.MinChild(self, current_index)
            if self.heapList[current_index] <
                self.heapList[minChildIndex]:
                temp = self.heapList[current_index]
                self.heapList[current_index] = self.heapList[minChildIndex]
                self.heapList[minChildIndex] = temp
                current_index = minChildIndex
            else:
                return
```