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” as a complete binary tree would be:

```
[1]
  [2]  [3]
 [15] [10]
 [114] [20] [20] [50]
 [8]  [9]  [10]
 [300] [125] [117]
```

Python List actually used to store heap items

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>not used</td>
<td>6</td>
<td>15</td>
<td>10</td>
<td>114</td>
<td>20</td>
<td>20</td>
<td>50</td>
<td>300</td>
<td>125</td>
<td>117</td>
</tr>
</tbody>
</table>

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) 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?

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,currentIndex):
        parentIndex =...

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
```

```python
def percUp(self,currentIndex):
    parentIndex =...
    while
```
```python
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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>not used</td>
<td>6</td>
<td>15</td>
<td>10</td>
<td>114</td>
<td>20</td>
<td>20</td>
<td>50</td>
<td>300</td>
<td>125</td>
<td>117</td>
<td></td>
</tr>
</tbody>
</table>

a) What item would `delMin` remove and return from the above heap?

b) What is the quickest way to fill the hole left by `delMin`?

c) What new problem does this cause?

**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:  # if 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.pop()
        self.percDown(1)
        return retval

def percDown(self, currentIndex):  
```

f) What is the big-oh notation for `delMin`?
### File: priority_queue.py
from binheap import BinHeap

class PriorityQueue:
    def __init__(self):
        self._heap = BinHeap()
    
    def isEmpty(self):
        return self._heap.isEmpty()
    
    def enqueue(self, item):
        self._heap.insert(item)
    
    def dequeue(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

    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.,
  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.add(item)        # adds the item to the list

- **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] = newValue</td>
<td>L.search(item)</td>
<td>L.hasPrevious()</td>
</tr>
<tr>
<td>L.pop(index)</td>
<td>L.index(item)</td>
<td>L.previous()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.first()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.last()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.insert(item)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.replace(item)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.remove()</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?