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 the 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:

Python List actually used to store heap items:

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>13</td>
<td>3</td>
<td></td>
<td></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: \[ i \times 2 \]
- its right child if it exists: \[ i \times 2 + 1 \]
- its parent if it exists: \[ i / 2 \]

b) What would the above heap look like after inserting 13 and then 37 (show the changes on above tree)

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, currentInder):
        parentInd = (currentIndex // 2) and self.heapList[currentIndex] < self.heapList[ParentIndex]
        while parentInd:
            temp = self.heapList[currentIndex]
            self.heapList[currentIndex] = self.heapList[parentIndex]
            self.heapList[parentIndex] = temp
            currentIndeX = ParentIndex
            parentInd = currentIndex // 2

    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.

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:
            return i * 2 + 1
        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):
    while True:
        if currentIndex > self.currentSize:
            return
        minIndex = minChild(self, currentIndex)
        if self.heapList[currentIndex] < self.heapList[minIndex]:
            self.heapList[currentIndex], self.heapList[minIndex] = self.heapList[minIndex], self.heapList[currentIndex]
            currentIndex = minIndex
        else:
            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
>>> 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.,
  ```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-base 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.append(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>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?
binheap.py

# Bradley N. Miller, David L. Ranum
# Introduction to Data Structures and Algorithms in Python
# Copyright 2005
# import unittest

# this heap takes key value pairs, we will assume that the keys are integers
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        print(len(self.heapList), i)
        while (i > 0):
            print(self.heapList, i)
            self.percDown(i)
            i = i - 1
        print(self.heapList, i)

    def percDown(self, self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
                i = mc

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self, self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
                i = i // 2

    def insert(self, k):