3. An alternative implementation of a Deque would be a doubly-linked implementation as in:

DoublyLinkedDeque Object

4. A priority queue has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a priority that determines the order they are removed. A hospital emergency room operates like a priority queue -- the person with the most serious injuries has highest priority even if they just arrived.

b) To implement a priority queue, we could use an unordered Python list. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue: $O(1)$
- dequeue: $O(n)$

c) To implement a priority queue, we could use a Python list order by priorities in descending order. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue: $O(n)$
- dequeue $O(1)$
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

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: 2i
   - its right child if it exists: 2i + 1
   - its parent if it exists: 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(1) 2x = n

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

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

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

```python
class MinHeap:
    def __init__(self):
        self.heapList = [0] * 2
        self.currentSize = 0

    def minChild(self, i):
        if i * 2 + 1 <= self.currentSize:
            return i * 2 + 1
        else:
            return i * 2

    def delMin(self):
        ret = self.heapList[1]
        self.currentSize -= 1
        self.heapList.pop()
        self.percDown(1)
        return ret

    def percDown(self, self, currentIdx):
        while currentIdx <= self.currentSize:
            minChildIdx = self.minChild(currentIdx)
            if self.heapList[currentIdx] <= self.heapList[minChildIdx]:
                break
            else:
                self.heapList[currentIdx], self.heapList[minChildIdx] = self.heapList[minChildIdx], self.heapList[currentIdx]
                currentIdx = minChildIdx
```

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`? move last item to index 1

c) What new problem does this cause? destroys heap order property

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`.

f) What is the big-oh notation for `delMin`? $O(\log_2 n)$
Data Structures (CS 1520)  Lecture 7  Name:

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 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()
5
>>> 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"

```
\[\begin{array}{c}
\text{\textbackslash w} \\
\text{\textbackslash a} \\
\text{\textbackslash y} \\
\text{\textbackslash c} \\
\end{array}\]\]
```

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><code>L.insert(index, item)</code></td>
<td><code>L.add(item)</code></td>
<td><code>L.hasNext()</code></td>
</tr>
<tr>
<td><code>item = L[index]</code></td>
<td><code>L.remove(item)</code></td>
<td><code>L.next()</code></td>
</tr>
<tr>
<td><code>L[index] = newValue</code></td>
<td><code>L.search(item) #return Boolean</code></td>
<td><code>L.hasPrevious()</code></td>
</tr>
<tr>
<td><code>L.pop(index)</code></td>
<td><code>i = L.index(item)</code></td>
<td><code>L.previous()</code></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?