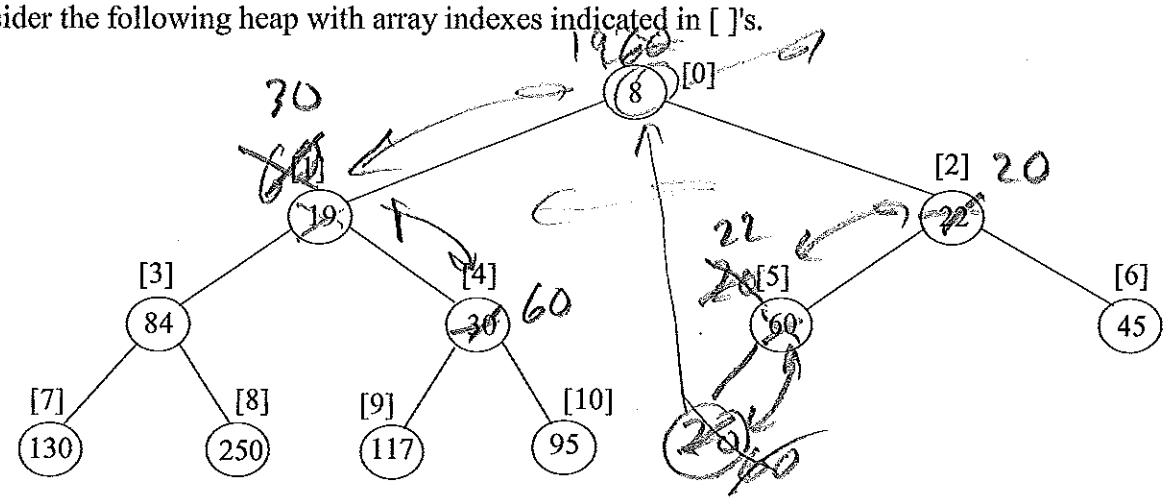
Spring 2011 Name: Mark F.

Data Structures - Test 2

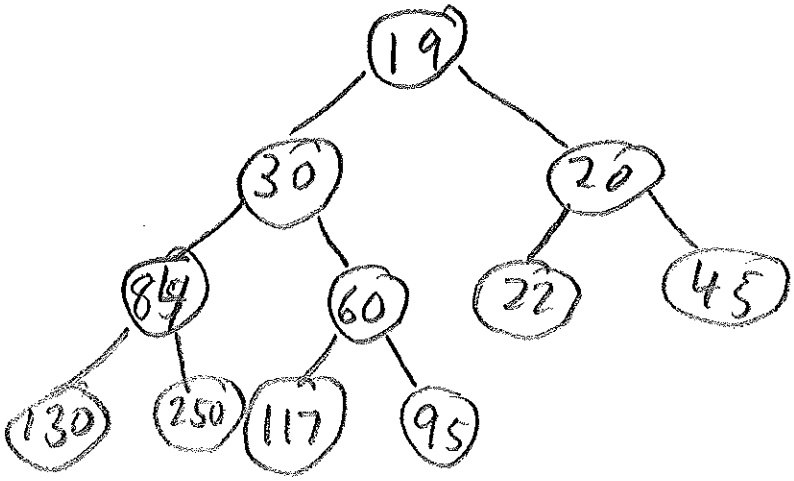
Question 1. Perhaps the best way to implement a priority queue uses a array/Python list organized as a heap. Consider the following heap with array indexes indicated in [ ]'s.



a) (6 points) For a node at index *i*, what is the index of:

- its left child if it exists: 2*i+1
- its right child if it exists: 2*i+2
- its parent if it exists: (i-1)/2

b) (20 points) What would the above heap look like after adding 20, **and then** popping (dequeuing) an item?



c) (9 points) Explain why adding a new item to a heap has a worst-case big-oh of $O(\log_2 n)$, where n is the number of items in the heap.

The complete binary tree with n items has a height of $\log_2 n$. Since we add as a leaf and compare with its parent(s) up the tree, it can only move up $\log_2 n$ times

Question 2. (25 points) Below is the textbook's partial `ArrayHeap` class showing the `add` method.

```
class ArrayHeap(object):

    def __init__(self):
        self._heap = []

    def add(self, item):
        self._heap.append(item)
        curPos = len(self._heap) - 1
        while curPos > 0:
            parent = (curPos - 1) / 2
            parentItem = self._heap[parent]
            if parentItem <= item:
                break
            else:
                self._heap[curPos] = self._heap[parent]
                self._heap[parent] = item
                curPos = parent
```

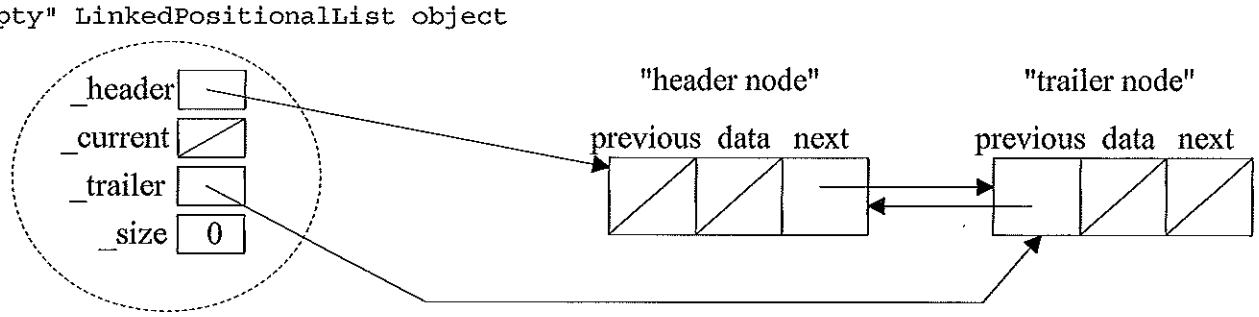
You are to complete the below version of the `add` method which uses a **recursive** `siftUp` function to move the new item to its correct spot in the heap. Your `siftUp` function should have one recursive case:

- if item is not at the root already and the parent > item, then move parent down to the item's current position and siftUp the item from the parent's position.

If it is not a recursive case, i.e., it is a base case, then we do not need to do anything since the item is at the correct spot in the heap.

```
class ArrayHeap(object):

    def __init__(self):
        self._heap = []

    def add(self, item):

        def siftUp(curPos):
            parentPos = (curPos - 1) / 2
            parent = self._heap[(curPos - 1) / 2]
            if curPos > 0 and item < self._heap[(curPos - 1) / 2]:
                self._heap[curPos] = parent
                self._heap[parentPos] = item
                siftUp(parentPos)

        # start of add method's code
        self._heap.append(item)       # add item as leaf
        siftUp(len(self._heap) - 1)   # call siftUp to move item to correct spot
```

Question 3. In lab 7, we implemented a positional-list using a doubly-linked list with a *header* node and *trailer* node to reduce the number of "special cases" (e.g., inserting first item in an empty list). An empty list looks like:



Instead of thinking of a cursor between two list items like the textbook, we have a *current item* which is always defined as long as the list is not empty. We inserted and deleted relative to the curent item.

Positional-based operations	**Description of operation**
`L.insertAfter(item)`	Inserts item after the current item, or as the only item if the list is empty. The new item is the current item.
`L.insertBefore(item)`	Inserts item before the current item, or as the only item if the list is empty. The new item is the current item.
`L.remove()`	Removes and returns the current item. Making the next item the current item if one exists; otherwise the tail item in the list is the current item. Precondition: the list is not empty.
`L.getItem()`	Returns the current item without removing it or changing the current position. Precondition: the list is not empty.
`L.hasNext()`	Returns True if the current item has a next item; otherwise return False. Precondition: the list is not empty.
`L.next()`	Precondition: hasNext returns True. Postcondition: The current item is has moved right one item
`L.hasPrevious()`	Returns True if the current item has a previous item; otherwise return False. Precondition: the list is not empty.
`L.previous()`	Precondition: hasPrevious returns True. Postcondition: The current item is has moved left one item
`L.first()`	Makes the first item the current item. Precondition: the list is not empty.
`L.last()`	Makes the last item the current item. Precondition: the list is not empty.
`L.replace(newValue)`	Replaces the current item by the newValue. Precondition: the list is not empty.

a) (6 points) Complete the worst-case big-oh notation for each `LinkedPositionalList` operation assuming the above implementation. Let n be the number of items in the list.

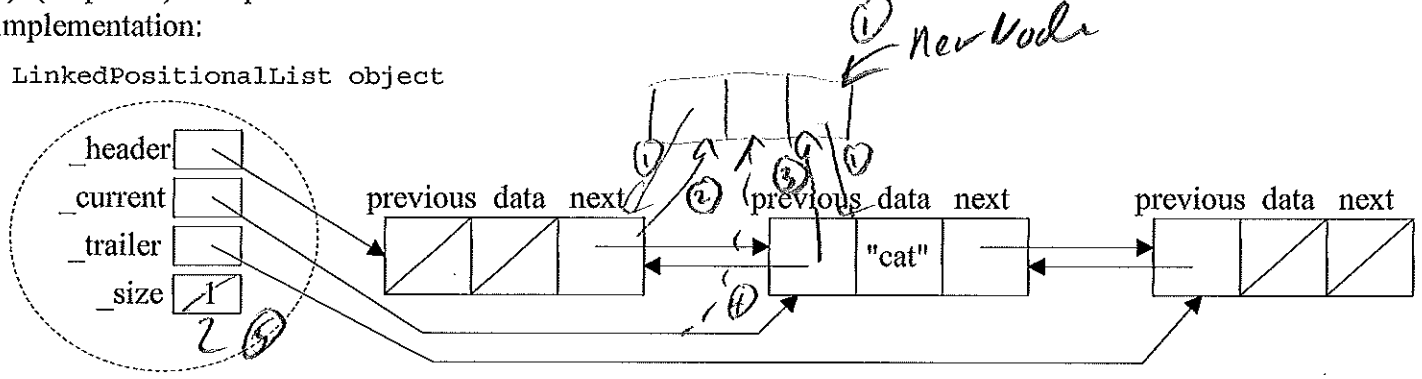
insertAfter	insertBefore	remove	first	next	replace
O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

b) (4 points) Provide a sentence of justification for your answers in part (a) for each of the following operations:

`insertBefore`: Since the list is doubly-linked, we just need to manipulate a few pointers to link in the new node

`first`: _header can be used to find "first" node easily.

c) (30 points) Complete the `insertBefore` and `first` methods of the `LinkedPositionalList` implementation:




```
from node import TwoWayNode # Has __init__ method: TwoWayNode(myData,myPrevious,myNext)
                            # Has public data attributes: data, previous, and next

class LinkedPositionalList(object):
    """ Linked implementation of a positional list."""

    def __init__(self):
        self._header = TwoWayNode(None, None, None)
        self._trailer = TwoWayNode(None, self._header, None)
        self._header.next = self._trailer
        self._current = self._header
        self._size = 0

    def insertBefore(self, item):
        """Inserts item before the current item, or as the only item if the list is empty.
           The new item is the current item."""
        if self._size == 0:
            self._current = self._trailer
        newNode = TwoWayNode(item, self._current.previous, self._current)
        self._current.previous.next = newNode
        self._current.previous = newNode
        self._current = newNode
        self._size += 1

    def first(self):
        """Moves the cursor to the first item if there is one.
           Precondition:  the list is not empty."""
        if self._size == 0:
            raise Exception, "list is empty, so first item does not exist"
        self._current = self._header.next
```

(20) (10)

36