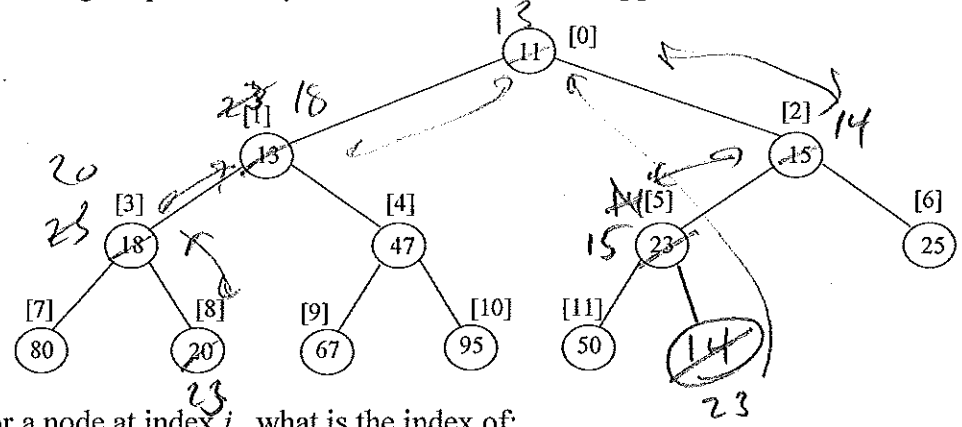
Spring 2012 Name: Mark F.

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/list indexes indicated in [ ]'s.



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

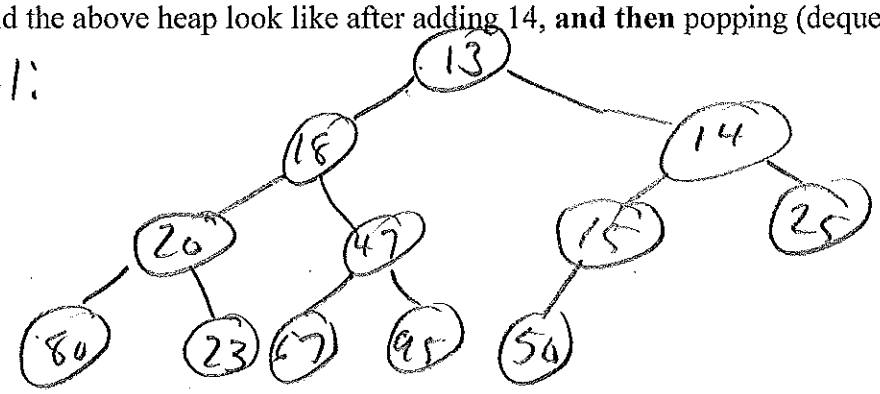
6

- 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) (14 points) What would the above heap look like after adding 14, **and then** popping (dequeuing) an item?

Pop returns 11:

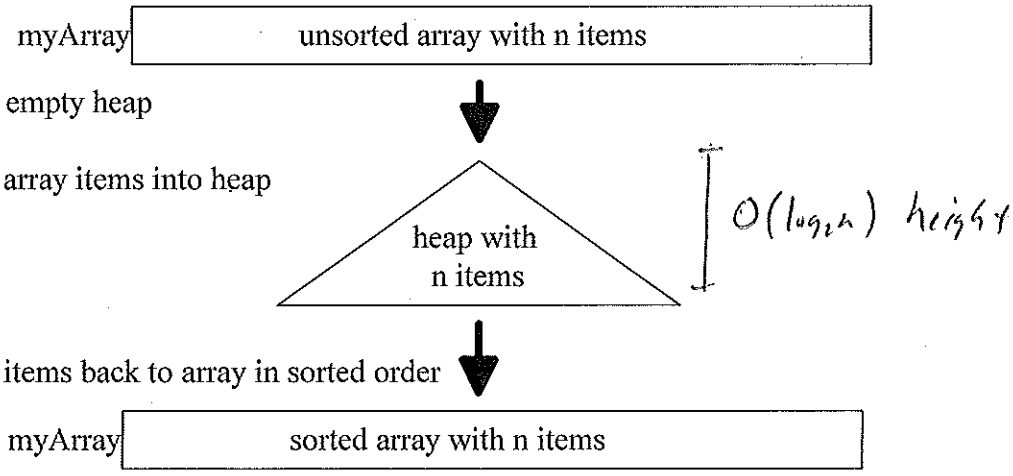
14



c) (10 points) We discussed using a heap to perform a sort. The steps of the algorithm are:

Steps:

1. Create an empty heap
2. Add all n array items into heap
3. Pop heap items back to array in sorted order



10

Determine and explain the *O*( ) for this algorithm. Each heap add is O(log n) since the height of the heap is at most O(log n), so step 2. is at most O(n log n) Step 3 is O(n log n) since each pop of heap is at most O(log n)

30

Question 2. (15 points) *Quick sort* general idea is as follows. For a piece of the list greater than one in size:

- *Partition* the list by "randomly" selecting an item (called the *pivot*) in the unsorted part of the list and rearranging the unsorted items such that:

Pivot Index

All items < to Pivot	Pivot Item	All items >= to Pivot

- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

a) Given the following `partition` function which performs the first step and returns the index of the pivot after this rearrangement, **complete the recursive `quicksortHelper` function.**

```
def partition(lyst, left, right):
    # Returns pivot index after partitioning
    middle = (left + right) / 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in xrange(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot and boundary items
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

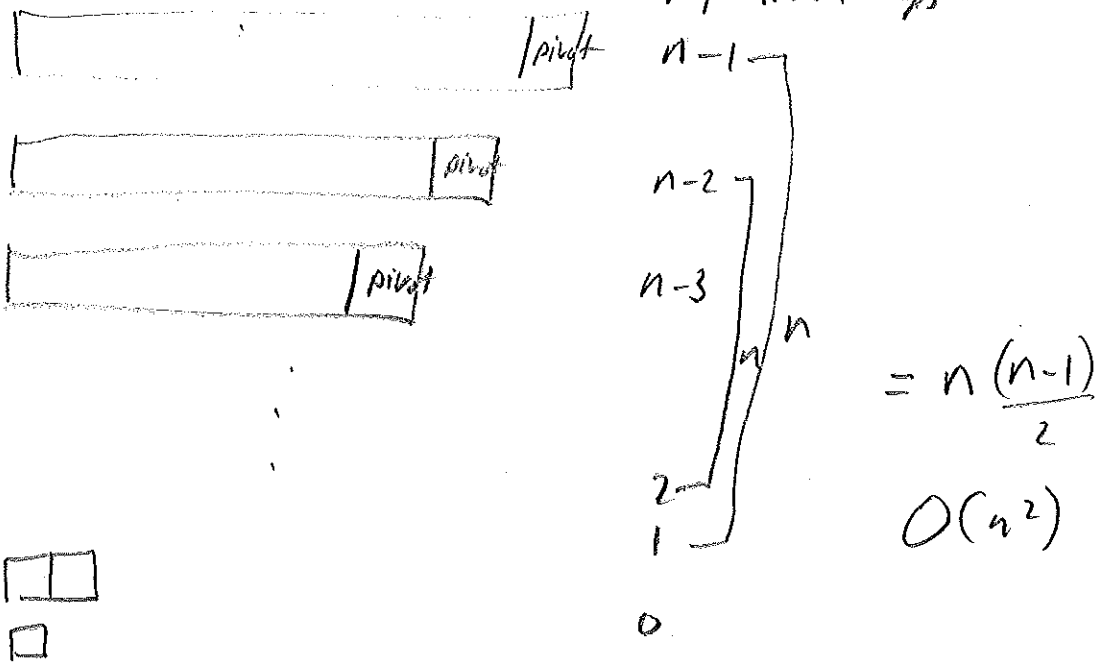
```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
    if left < right:
        pivotIndex = partition(lyst, left, right)
        quicksortHelper(lyst, left, pivotIndex - 1)
        quicksortHelper(lyst, pivotIndex + 1, right)
```

4 b) Let "n" be the number of items between left and right. What is the worst-case $O(\ )$ for partition? $O(n)$

since for-loop iterates from left to right

c) Ideally, the pivot item splits the list into two equal size problems, but in the worst case the pivot is all the way to one end. What would be the big-oh for Quick Sort in the **worst case**? **(Explain your answer)**

4

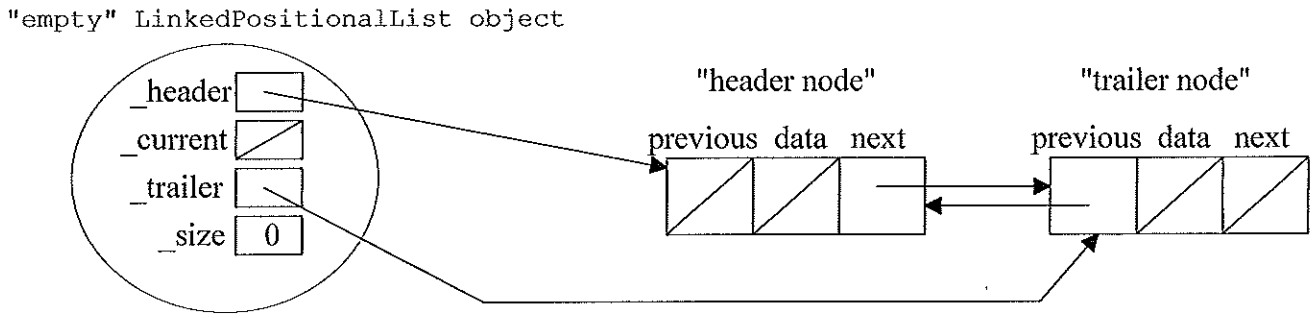


$= n\frac{(n-1)}{2}$

$O(n^2)$

15

Question 3. In lab 6 and homework #4, 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.getCurrent()`	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.

6

insertBefore	insertAfter	remove	last first?	next	previous
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:

2

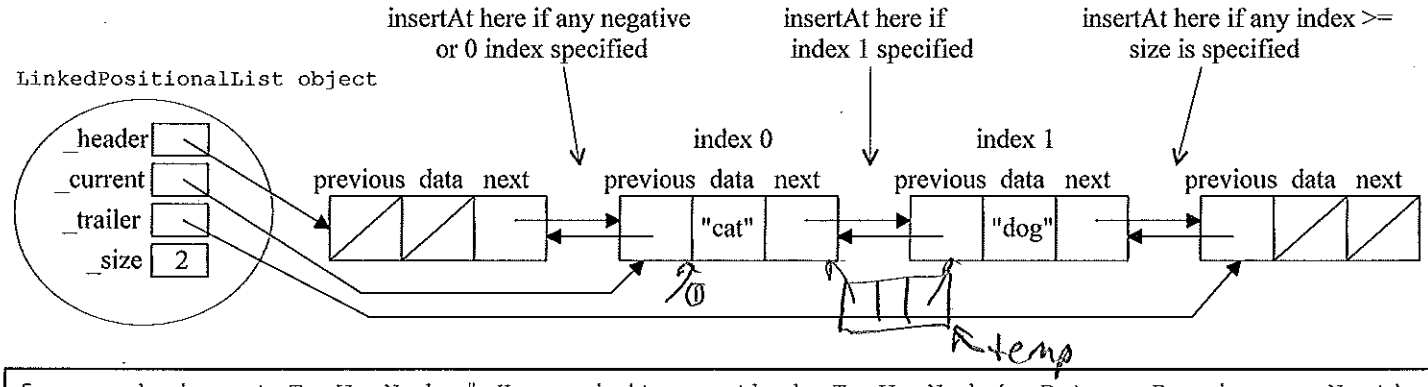
~~first~~ last: since _header points to heade node the first actual node can be found in constant time

2

insertBefore: Because of doubly-linked list, we just need to manipulate pointers to link in new node before _current item.

8

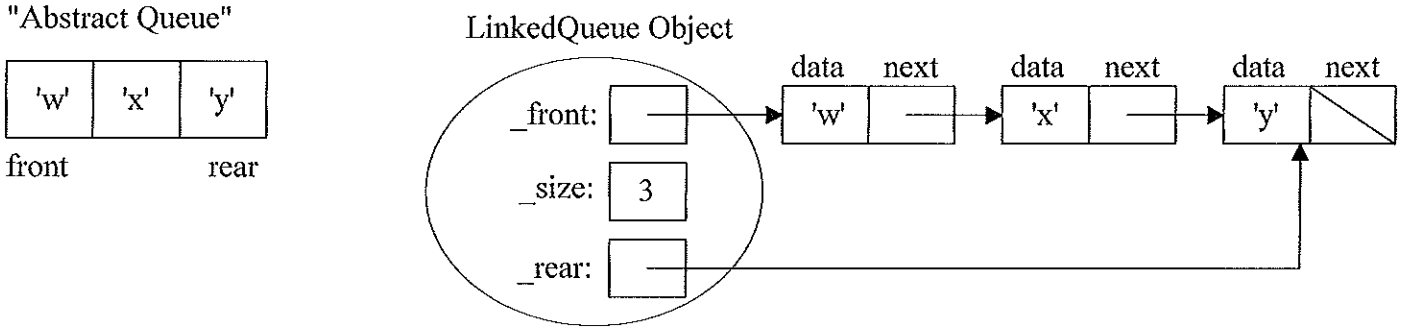
c) (30 points) Complete the code for a new method `insertAtIndex(self, item, index)` which inserts `item` at the specified `index` position and makes the new item the current item. Assume the first index in the list is 0. If the specified `index` is negative or 0, then insert at the beginning of the list. If the specified `index` is bigger than or equal to the lists size, then insert at the end of the list.




```
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 insertAtIndex(self, item, index):
       """Inserts item at the specified index position and makes the new item the
          current item.  Assume the first index in the list is 0. If the specified index
          is negative or 0, then insert at the beginning of the list.  If the specified
          index is bigger than or equal to the lists size, then insert at the end of the
          list."""
          if index <= 0:
              self._current = self._header
          elif index >= self._size:
              self._current = self._trailer.previous
          else:
              cursor = self._header
              for count in xrange(index):
                  cursor = cursor.next
              temp = TwoWayNode(item, cursor, cursor.next)
              temp.previous.next = temp
              temp.next.previous = temp
              self._current = temp
              self._size += 1
```

Question 4. (15 points) The Python `for` loop allows traversal of built-in data structures (strings, lists, tuple, etc) by an *iterator*. To accomplish this with ***our*** data structures we need to include an `__iter__(self)` method that gets used by the built-in `iter` function to create a special type of object called a *generator object*. The generator object executes as a separate process running concurrently with the process that created and uses the data structure being iterated.

A singly-linked list implementation of the queue (`LinkedQueue` class in the text) conceptually would look like:



The iterator (`__iter__`) method for the `LinkedQueue` class would be:

```
class LinkedQueue(object):
    """ Link-based queue implementation."""


  def __iter__(self):
      """An iterator for a linked queue"""
      cursor = self._front
      while True:
          if cursor == None:
              raise StopIteration
          yield cursor.data
          cursor = cursor.next
```

Write an `__iter__` method for the `LinkedPositionalList` class.

```
def __iter__(self):
    cursor = self._header.next
    while True:
        if cursor == self._trailer:
            raise StopIteration
        yield cursor.data
        cursor = cursor.next
```