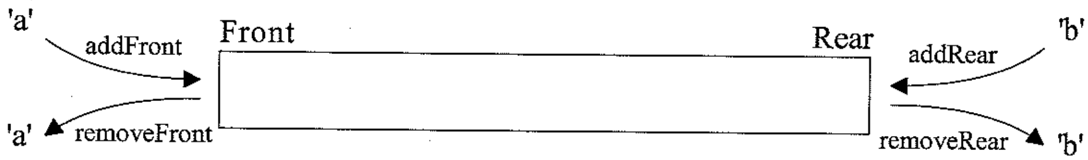
Fall 2010

Name: Mark F.

Data Structures - Test 2

Question 1. A Deque (pronounced "Deck") ADT is like a queue, but it allows adding or removing items from either the front or the rear of the Deque. Abstractly, the Deque behaves as:



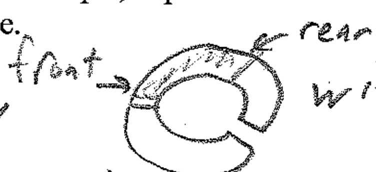
Consider the following Deque implementation which uses a Python list representation.

```
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addRear(self, item):
        self.items.append(item)

    def addFront(self, item):
        self.items.insert(0,item)

    def removeRear(self):
        return self.items.pop()

    def removeFront(self):
        return self.items.pop(0)

    def __len__(self):
        return len(self.items)
```

a) (10 points) Complete the worst-case big-oh notation for each Deque operation assuming the above implementation. Let n be the number of items in the Deque.

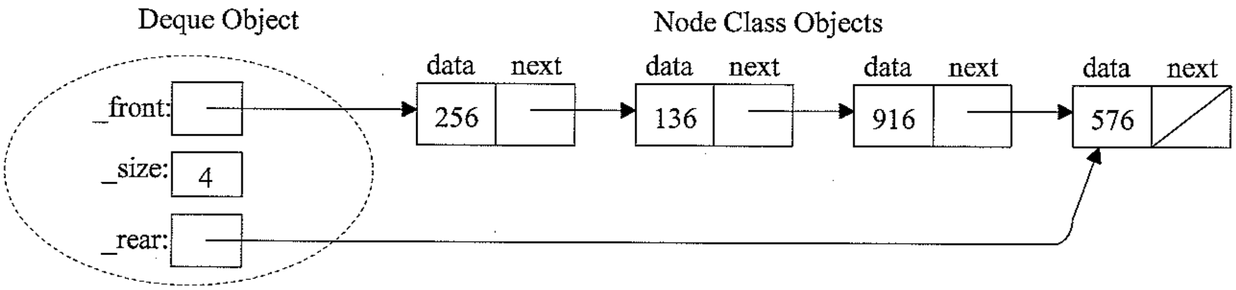
isEmpty	addFront	addRear	removeFront	removeRear	len
O(1)	O(n)	O(1)	O(n)	O(1)	O(1)

b) (8 points) Instead of the above list representation of a Deque, explain how an Array (the textbook Array class) can be used to improve performance of the Deque.

If use a circular Array 

with "floating" front and rear indexes, then we could get O(1) performance

Question 2. An alternative implementation of a Deque would be a linked implementation as in:



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

6

isEmpty	addFront	addRear	removeFront	removeRear	len
O(1)	O(1)	O(1)	O(1)	O(n)	O(1)

b) (9 points) Provide a sentence or two of justification for your answers in part (a) for each of the following operations:

4 removeFront: We just need to manipulate a few "pointer" values to remove the front node

5 removeRear - We need to traverse down list of nodes to locate the new rear node, so it is Θ(n)

c) (20 points) Complete the addFront and removeFront methods of the linked Deque implementation:

```
from node import Node    # Has constructor method: Node(myData, myNext)
                         # Has public data attributes: data and next

class Deque:
    def __init__(self):
        self._front = None
        self._rear = None
        self._size = 0

    def addFront(self, item):
```

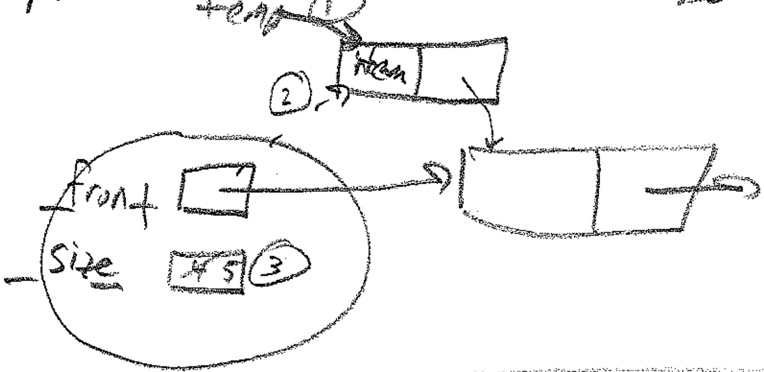
10

```
    def removeRear(self):
```

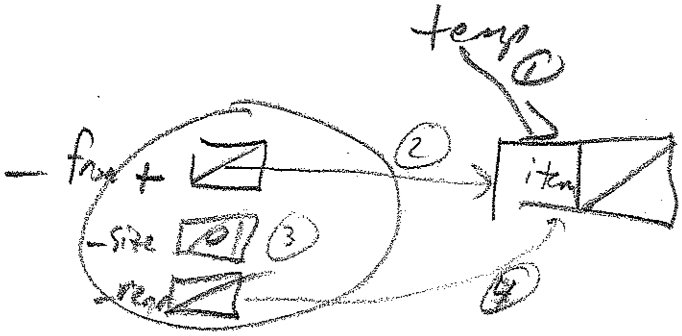
10

35

Add Front "normal case" - non empty Deque



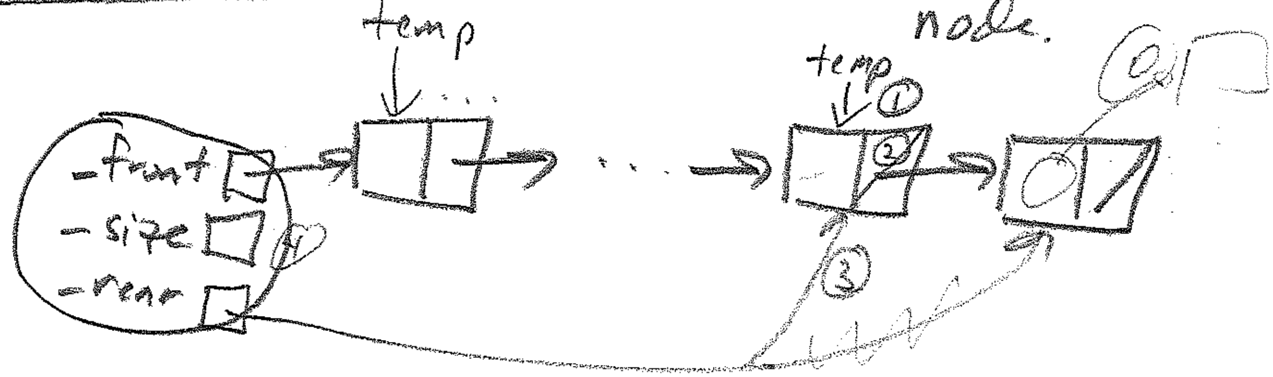
add to empty Deque special case



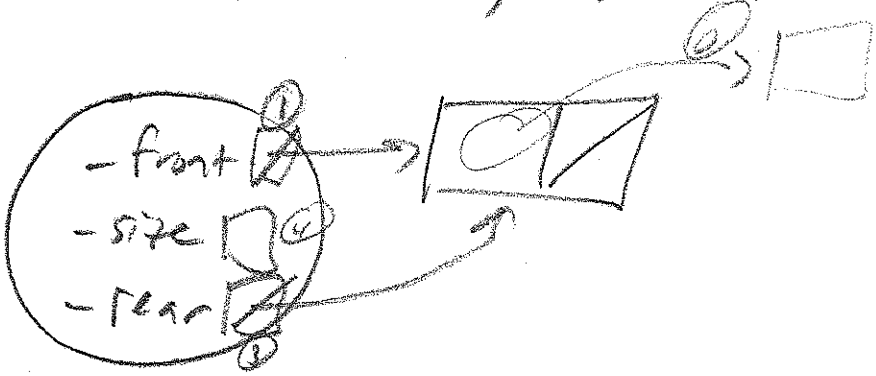

```
def addFront(self, item):
    temp = Node(item, self._front)
    self._front = temp
    if self._size == 0:
        self._rear = temp
    self._size += 1
```

+3

removeRear "normal case" - not deleting last node.



special case deleting last node



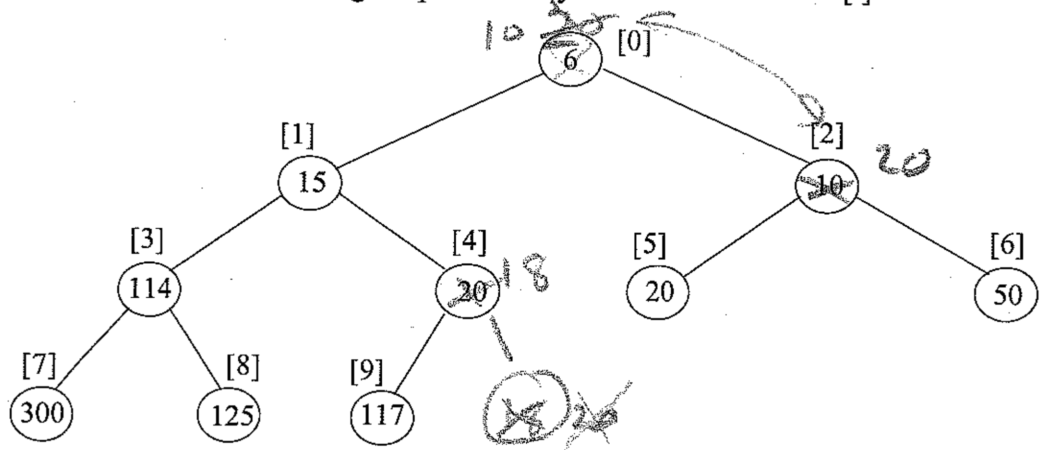

```
def removeRear(self):
    itemToReturn = self._rear.data
    if self._size == 1:
        self._front = None
        self._rear = None
    else:
        temp = self._front
        while temp.next != self._rear:
            temp = temp.next
        temp.next = None
        self._rear = temp
    self._size -= 1
    return itemToReturn
```

d) (7 points) Suggest a recommendation for improving the linked implementation of the Deque.

Use doubly-linked nodes. previous data next

to aid finding node before rear node.

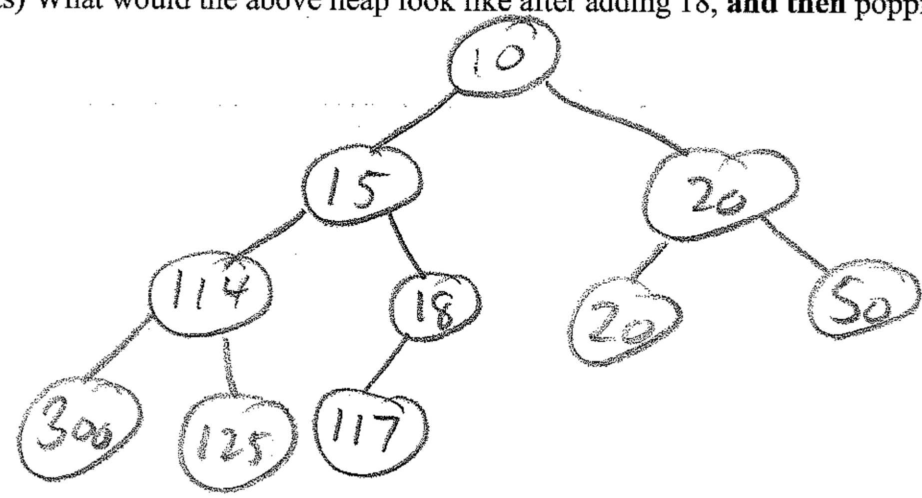
Question 3. Consider the following heap with array indexes indicated in [ ]'s.



a) (4 points) For a node at index i, what is the index of:

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

b) (10 points) What would the above heap look like after adding 18, **and then** popping (dequeuing) an item?



c) (6 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

A complete binary tree used by a heap has a height $\Theta(\log_2 n)$. When we add a new item as a leaf, we need to sift it up the tree at most to the root which is $\Theta(\log_2 n)$ away.

Name: ______________________

Question 4. Consider implementing a **sorted** list ADT that includes the following operations:

- indexed-based operations: [ ] as an accessor and `remove` (e.g., print myList[i] and myList.remove(i))
- content-based operations: `insert` and `index` (e.g., myList.insert(item) and i = myList.index(item))

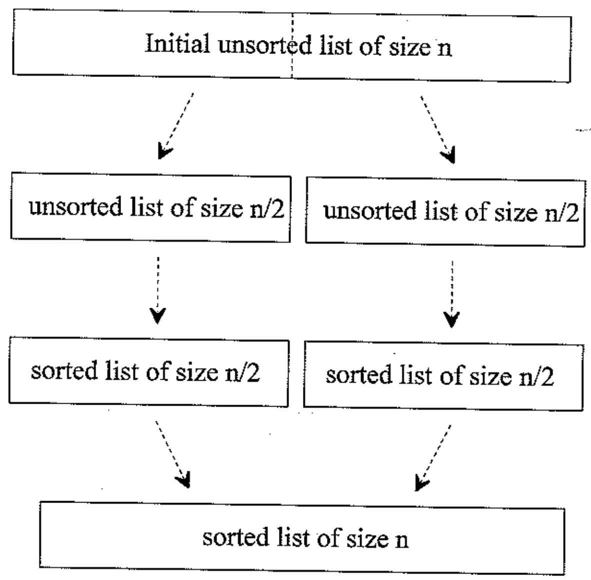
a) (5 points) If the underlying representation is an Array sorted by item values, then complete the worst-case big-oh notation for each sorted list operation. Assume that a binary search is used to find an item. Let n be the number of items in the sorted list.

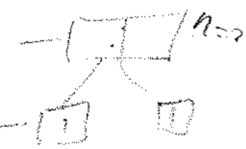
myList[i]	myList.remove(i)	myList.insert(item)	i = myList.index(item)
$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n)$

b) (5 points) If the underlying representation is a doubly-linked list sorted by item values, then complete the worst-case big-oh notation for each sorted list operation. Let n be the number of items in the sorted list.

myList[i]	myList.remove(i)	myList.insert(item)	i = myList.index(item)
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Question 5. Recall that merge sort is a recursive divide-and-conquer algorithm such that:





Divide - splits list/array into two equal parts

Conquer - recursively merge sort each half

Combine - merge the sorted halves back together

a) (5 points) When merging two sorted lists of size n/2 each, what is the worst-case number of comparisons that must be performed? (justify your answer for partial credit)

n-1, When one list runs out, the rest of the other can be copied without compares. In the worst-case we compare the last two items in each, then get to copy last one without a compare.

b) (5 points) What maximum depth of recursion does the merge sort algorithm require when sorting a list of size n? (justify your answer for partial credit)

$\sim \log_2 n$ since the list is split in half each recursive call.