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.

```python
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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>O(1)</td>
</tr>
<tr>
<td>addFront</td>
<td>O(n)</td>
</tr>
<tr>
<td>addRear</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeFront</td>
<td>O(n)</td>
</tr>
<tr>
<td>removeRear</td>
<td>O(1)</td>
</tr>
<tr>
<td>len</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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:

![Deque Object Diagram]

Node Class Objects

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td></td>
</tr>
<tr>
<td>916</td>
<td></td>
</tr>
<tr>
<td>576</td>
<td></td>
</tr>
</tbody>
</table>

Node class objects:

- data: [256, 136, 916, 576]
- next: [next, None, None, None]

- isEmpty: [0]
- addFront: [O(1)]
- addRear: [O(1)]
- removeFront: [O(1)]
- removeRear: [O(n)]
- len: [O(1)]

(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.

<table>
<thead>
<tr>
<th></th>
<th>isEmpty</th>
<th>addFront</th>
<th>addRear</th>
<th>removeFront</th>
<th>removeRear</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>isE</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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

- removeFront: We just need to manipulate a few pointer values to remove the front node.
- removeRear: We need to traverse down list of nodes to locate the new rear node, so it is O(n).

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

```python
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):
        # Add item to front of list

    def removeFront(self):
        # Remove item from front of list
```

```python
10
```

```python
10
```

```python
35
```
Add Front: "normal case" - non empty Deque

```
def add_front(self, item):
    temp = Node(item, self._front)
    self._front = temp
    if self._size == 0:
        self._rear = temp
    self._size += 1
```
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 to aid finding node before rear node.

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

```
   6
  / \
 15  20
 / \ / \  
114 40 20 50
 /   /   
300 125 117
```

a) (4 points) For a node at index i, what is the index of:
- its left child if it exists: \( 2 \times i + 1 \)
- its parent if it exists: \( \frac{i-1}{2} \)

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

```
   10
  / \  
15  20
 / \ / \  
114 18 20 50
 /       /  
300 125 117
```

c) (6 points) Explain why adding a new item to a heap has a worst-case big-oh of \( O(\log n) \), where \( n \) is the number of items in the heap.

A complete binary tree used by a heap has a height \( \Theta(\log n) \). When we add a new item as a leaf, we need to sift it up the tree at most to the root which is \( \Theta(\log n) \) times.
Question 4. Consider implementing a sorted list ADT that includes the following operations:

- indexed-based operations: [i] 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.

<table>
<thead>
<tr>
<th>myList[i]</th>
<th>myList.remove(i)</th>
<th>myList.insert(item)</th>
<th>i = myList.index(item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log_2 n)$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>myList[i]</th>
<th>myList.remove(i)</th>
<th>myList.insert(item)</th>
<th>i = myList.index(item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

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

Initial unsorted list of size n

Divide - splits list/array into two equal parts

Conquer - recursively merge sort each half

Combine - merge the sorted halves back together

<table>
<thead>
<tr>
<th>Initial unsorted list of size n</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted list of size n/2</td>
</tr>
<tr>
<td>unsorted list of size n/2</td>
</tr>
<tr>
<td>sorted list of size n/2</td>
</tr>
<tr>
<td>sorted list of size n/2</td>
</tr>
<tr>
<td>sorted list of size n</td>
</tr>
</tbody>
</table>

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)

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