Data Structures - Test 2

Question 1. A Deque (pronounced "Deck") ADT is like a double-ended queue, i.e., it allows adding or removing items from either the front or the rear of the Deque. One possible implementation of a Deque would be to use an array (dequeArray) to store the Deque items such that

- the front item is **always stored at index 0,**
- an integer numItems maintains the number of items in the Deq
- an integer rear maintains the index of the rear item
- an integer maxDeqSize maintains the size of the array

```
        0  1  2  3  4  5      99
dequeArray: 'a' 'b' 'c' 'd'
```

| numItems: | 4 |
| rear:     | 3 |
| maxDeqSize: | 100 |

(a) (12 points) Complete the worst-case theta notation, \( \Theta(n) \), for each Deque operation, assuming the above implementation. Let \( n \) be the number of items in the Deque.

<table>
<thead>
<tr>
<th>isEmpty</th>
<th>addFront</th>
<th>addRear</th>
<th>removeFront</th>
<th>removeRear</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>

(b) (10 points) Suggest an improvement to the above array implementation of the Deque to speed up some of these operations.

- Allow the front to "float" from other array-like treating array circularly.

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

```
Deque Object
       (Linked Representation)
```

```
front: —-> 'a' ——> 'b' ——> 'c' ——> 'd' ——> rear:
```

| numItems: | 4 |

(a) (12 points) Complete the worst-case theta notation, \( \Theta(n) \), for each Deque operation assuming the above linked implementation. Let \( n \) be the number of items in the Deque.

<table>
<thead>
<tr>
<th>isEmpty</th>
<th>addFront</th>
<th>addRear</th>
<th>removeFront</th>
<th>removeRear</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>
b) (16 points) Complete the removeRear method of the linked Deque implementation:

```cpp
template <typename T>
class DequeNode {
public:
    T value; // Node value
    DequeNode<T> *next; // Pointer to next node

    // Constructor
    DequeNode (T nodeValue)
    { value = nodeValue;
      next = NULL;
    } // end class DequeNode

    template <typename T>
    class Deque {
private:
    DequeNode<T> *front; // Deque front pointer
    DequeNode<T> *rear; // Deque rear pointer
    int numItems; // Count of nodes

public:
    +3 Special case: deleting last elt.
    ...;
    +4 loop to find node before rear
    +3 delete node after getting value
    +3 decrement numbers

    Deque<T> & removeRear () { }  
    +3 General ptr. manipulation
    if (rear == NULL) { 
        assert("Cannot remove Rear from empty Deque");
    } else if (rear == front) { 
        value = rear->value;
        delete rear;
        rear = NULL;
        front = NULL;
    } else { 
        value = rear->value;
        rear = front;
        while (rear->next != NULL) { 
            rear = rear->next;
        } 
        delete rear->next;
    }
    return value = numItems--;
```
c) (5 points) Suggest a recommendation for improving the linked implementation of the Deque.

Doubly linked nodes so node before rear can be found easily.

Question 6. 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: \( 2i + 1 \)
   - its parent if it exists: \( \frac{i-1}{2} \)

b) (11 points) What would the above heap look like after adding 18, and then dequeuing an item?

The height of a complete binary tree is \( \log_2 n \) since the number of nodes on each level doubles.
Question 7. Recall that merge sort is a recursive divide-and-conquer algorithm such that:

Divide - splits list/array into two equal parts

\[ \text{Initial unsorted list of size } n \]

\[ \text{unsorted list of size } n/2 \quad \text{unsorted list of size } n/2 \]

Conquer - recursively merge sort each half

\[ \text{sorted list of size } n/2 \quad \text{sorted list of size } n/2 \]

Combine - merge the sorted halves back together

\[ \text{sorted list of size } n \]

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)

\[ (n-1) \text{ comparisons because in the worst case neither lists of size } n/2 \text{ run out until we compare the last items} \]

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)

\[ \log_2 n \text{ since } n \text{ is repeated list in half to } \text{array of size} \]

1

c) (10 points) Both Heap sort and Merge sort are \( \Theta(n \log_2 n) \), where \( n \) is the number of items being sorted, but Merge sort takes about twice the time of Heap sort. Why?

The merge sort does more moves when copying to smaller arrays and merging back to larger array.