A Deque (pronounced "Deck") is a linear data structure which behaves like a double-ended queue, i.e., it allows adding or removing items from either the front or the rear of the Deque.

1. One possible implementation of a Deque would be to use a Python list to store the Deque items such that:
   - the rear item is always stored at index 0,
   - the front item is always stored at the highest index (or -1)

   ```python
   class Deque(object):
       def __init__(self):
           self.items = list()
   ```

   a) Complete the `__init__` method and determine the big-oh, $O()$, 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>removeFront</th>
<th>addRear</th>
<th>removeRear</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

   b) Write the methods for the addRear and removeRear operation.

   ```python
def addRear(self, newItem):
    self.items.insert(0, newItem)

def removeRear(self):
    if self.size() == 0:
        raise Exception("cannot...")
    return self.items.pop(0)
   ```

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

```python
class LinkedListDeque(object):
    def __init__(self):
        self.rear = None
        self.front = None
        self.size = 0
```

a) Complete the `__init__` method and determine the big-oh, $O()$, 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>removeFront</th>
<th>addRear</th>
<th>removeRear</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

b) Suggest an improvement to the above linked implementation of the Deque to speed up some of its operations.
3. An alternative implementation of a Deq would be a doubly-linked implementation as in:

DoublyLinkedDeque Object

a) Determine the big-oh, $O()$, for each Deq operation assuming the above doubly-linked implementation. Let $n$ be the number of items in the Deq.

<table>
<thead>
<tr>
<th>isEmpty</th>
<th>addFront</th>
<th>removeFront</th>
<th>addRear</th>
<th>removeRear</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

4. A priority queue has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a priority that determines the order they are removed. A hospital emergency room operates like a priority queue -- the person with the most serious injury has highest priority even if they just arrived.

a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next?

priority queue: 40 10 13 79 30 5

b) To implement a priority queue, we could use an unorder Python list. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)
- enqueue: $O(1)$
- dequeue: $O(n)$

c) To implement a priority queue, we could use a Python list order by priorities in descending order. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)
- enqueue
- dequeue
Dequeue removeFront "doubly-linked list"

(1) Normal-case Pix
(2) Update Pix for remove front
(3) Number steps
(4) Write normal-case code.

1. temp = self. prev
2. self. prev = temp.getPrevious()
3. self. prev.setNext(None)
4. self. size -= 1
5. return temp.getData()

(6) if self.size == 0:
    raise Exception("="
else:
    self. rear = None

(5) Consider special case(s):
- empty Deque
- remove last item
  (i) draw special case
  (ii) track/record normal-case calls
  (iii) cause error
  (iv) self.rear = None
\[
\begin{align*}
\text{minValue} &= \min (\text{self.items}) \quad O(n) \\
\text{minIndex} &= \text{self.items.index} (\text{minValue}) \quad O(n/2) \\
\text{self.items}[\text{minIndex}] &= \text{self.items.pop()} \quad O(1) \\
\hline
O \left( \frac{3n}{2} \right)
\end{align*}
\]

\[
\begin{align*}
\text{minValue} &= \text{self.items}[0] \\
\text{minIndex} &= 0 \\
\text{for } \text{testIndex} \text{ in range } (1, \text{len(self.items)}) : \\
O(n) & \text{ if self.items[testIndex] \leq minValue:} \\
\text{minValue} &= \text{self.items[testIndex]} \\
\text{minIndex} &= \text{testIndex}
\end{align*}
\]