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)

   ![Deque Object](image)

   ![List Object](image)

   ```python
   class Deque(object):
       def __init__(self):
           self.items = []
   
   def addFront(self, newItem):
       self.items.insert(0, newItem)
   
   def removeFront(self):
       if len(self.items) == 0:
           raise Exception("Cannot...")
       return self.items.pop(0)
   
   def addRear(self, newItem):
       self.items.append(newItem)
   
   def removeRear(self):
       if len(self.items) == 0:
           raise Exception("Cannot...")
       return self.items.pop()
   
   def size(self):
       return len(self.items)
   
   isEmpty(self):
   
   addFront(self, newItem):
   
   removeFront(self):
   
   addRear(self, newItem):
   
   removeRear(self):
   
   size(self):
   
   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>Operation</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addFront()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeFront()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addRear()</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>removeRear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size()</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

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

   ![LinkedDeque Object](image)

   ```python
   class LinkedDeque(object):
       def __init__(self):
           self._rear = None
           self._front = None
           self._size = 0
   
   def addFront(self, newItem):
       ... 
   
   def removeFront(self):
       ... 
   
   def addRear(self, newItem):
       ... 
   
   def removeRear(self):
       ... 
   
   def size(self):
       ... 
   
   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>Operation</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addFront()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeFront()</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>addRear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeRear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size()</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.
from node import Node

class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None

    def getPrevious(self):
        return self.previous

    def setPrevious(self, newprevious):
        self.previous = newprevious

3. An alternative implementation of a Deque would be a doubly-linked implementation as in:

DoublyLinkedDeque Object

<table>
<thead>
<tr>
<th>previous</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>'b'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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?

```python
<table>
<thead>
<tr>
<th>priority queue:</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 10 13 79 30 20</td>
</tr>
</tbody>
</table>
```

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

- enqueue: \( O(1) \) (append to right end of list)
- dequeue: \( O(n) \)

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

- enqueue: \( O(n) \)
- dequeue: \( O(n) \)
Normal-case removeFront Deque:

1. \( \text{temp} = \text{self.\_front} \)
   - if \( \text{self.\_size} == 0 \)
     - raise Exception()
2. \( \text{self.\_front} = \text{self.\_front.\_getPrevious()} \)
3. \( \text{self.\_front.\_setNext}(\text{None}) \)
   - if \( \text{self.\_size} == 1 \)
     - \( \text{self.\_rear} = \text{None} \)
   - else:
4. \( \text{self.\_size} -= 1 \)
5. return \( \text{temp, get\_Data()} \)

---

Special cases

1. removingFront of only/last node
2. precondition - empty Deque

---

Diagram:

- \( \text{node} \)
- \( \text{prev\_next} \)
- \( \text{size} \)
- \( \text{front} \)
- \( \text{rear} \)
- \( \text{temp} \)
- \( \text{prev} \)
- \( \text{next} \)
- \( \text{data} \)
1. Section 6.6 discusses a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array is used to store a complete binary tree (a full tree with any additional leaves as far left as possible) with the items being arranged by heap-order property, i.e., each node is ≤ either of its children. An example of a min heap "viewed" as a complete binary tree would be:

```
[6]
[1]
[2]  
[4]  [5]  
15    20  
114  
300  125 117
```

Python List actually used to store heap items
```
[6]  15  10  114  20  20  50  300  125  117
```

a) For the above heap, the list/array indexes are indicated in [ ]'s. For a node at index i, what is the index of:
   • its left child if it exists: \[2 \times i + 1\]
   • its right child if it exists: \[2 \times i + 1\]
   • its parent if it exists: \[\frac{i - 1}{2}\]

b) What would the above heap look like after inserting 13 and then 3? (show the changes on above tree)

General Idea of insert(newItem):
   • append newItem to the end of the list (easy to do, but violates heap-order property)
   • restore the heap-order property by repeatedly swapping the newItem with its parent until it percolates to correct spot

c) What is the big-oh notation for inserting a new item in the heap?

d) Complete the code for the percUp method used by insert.

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self, self, currentIndex):
        parentIndex = 
        while

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self, self.currentSize)
```

Lecture 7 - Page 1