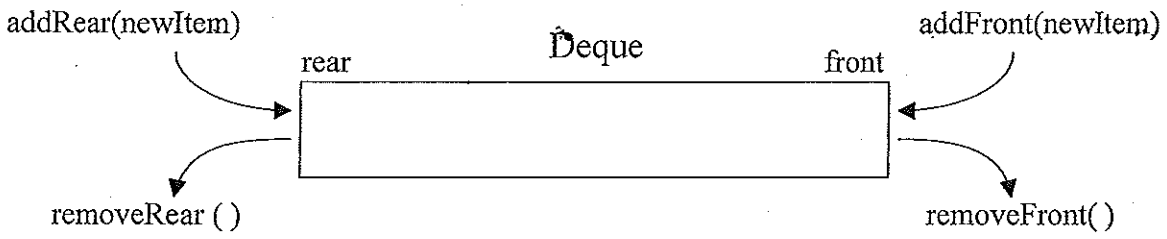
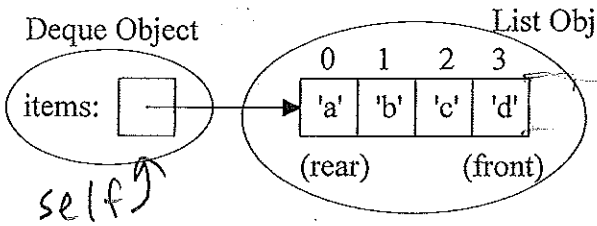
Data Structures (CS 1520) Lecture 6 Name: Mark F.

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)




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

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.

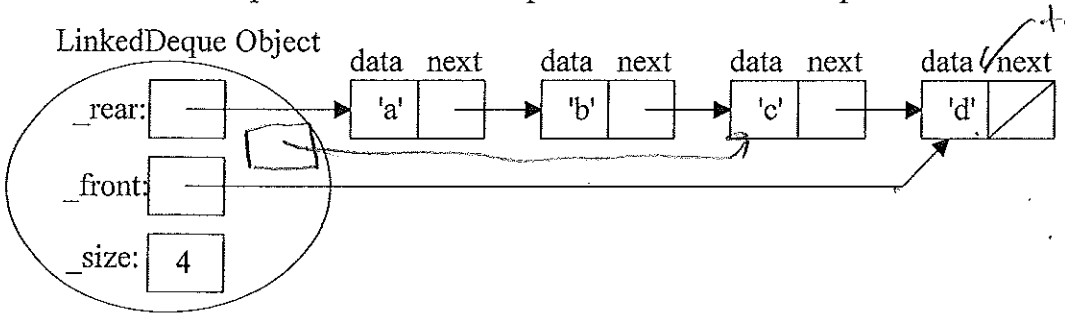
isEmpty	addFront	removeFront	addRear	removeRear	size
O(1)	O(1)	O(1)	O(n)	O(n)	O(1)

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

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

```
def removeRear(self):
    return self.items.pop(0)
```

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




```
class LinkedDeque(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.

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

b) Suggest an improvement to the above linked implementation of the Deque to speed up some of its operations.

double-linked list of nodes

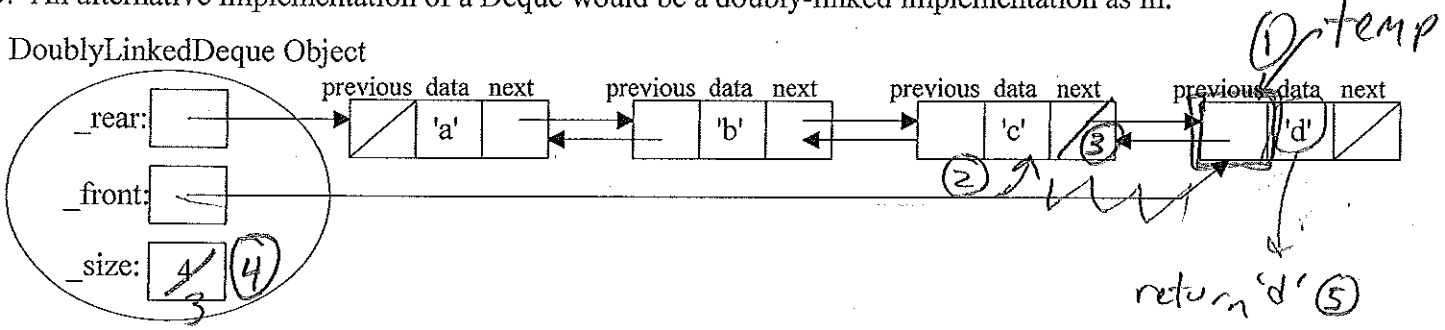
```
from node import Node

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:



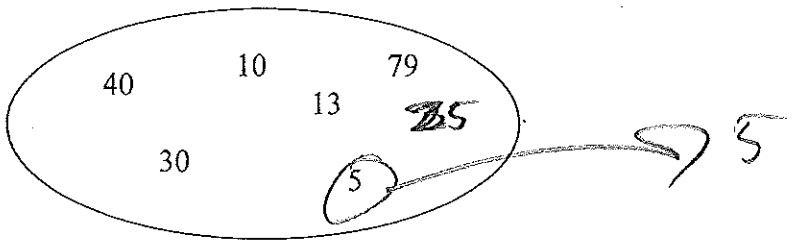
a) Determine the big-oh, *O* ( ), for each Deque operation assuming the above doubly-linked implementation. Let n be the number of items in the Deque.

isEmpty	addFront	removeFront	addRear	removeRear	size
O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

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 proirity that determines the order they are removed. A hospital emergence room operates like a priority queue -- the person with the most serious injure has highest priority even if they just arrived.

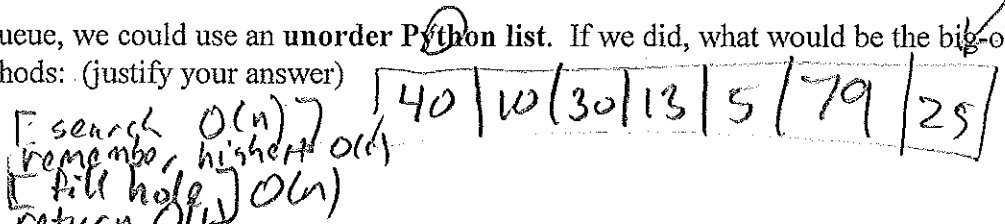
a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next?

priority queue:



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 decending order. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue:
- dequeue

DoublyLinkedDeque removeFront

normal case code

① temp = self._front // temp = self._front.getData()

② self._front = {self._front.getPrevious() / temp.getPrevious()}

③ self._front.setNext(None)

④ self._size -= 1

⑤ return temp.getData() // return temp

---

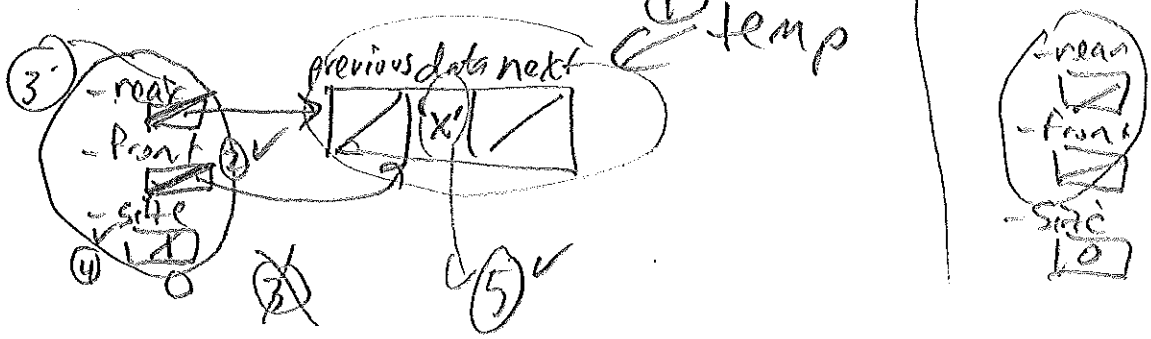
special cases

① empty Deque? precondition

```
if self._size == 0:
    raise ValueError("Cannot removeFront of an empty Deque.")
```

---

② removeFront the last item



① same

② same

```
if self._size == 1:
    self._rear = None    ③'
else:
    self._front.setNext(None)    ③
```

④ same

⑤ same

2 items in Deque - remove front