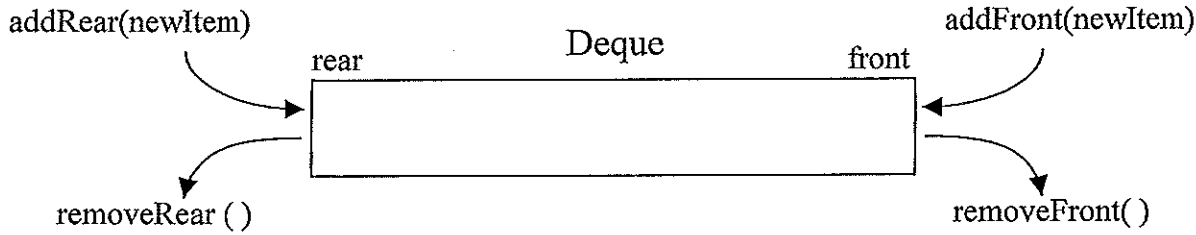
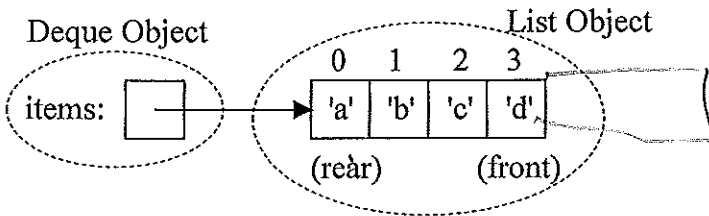


A Deque (pronounced "Deck") is a linear data structure which behaves like a double-ended queue, i.e., it allows adding or removing items from either the front or the rear of the Deque.



- 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 = 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.

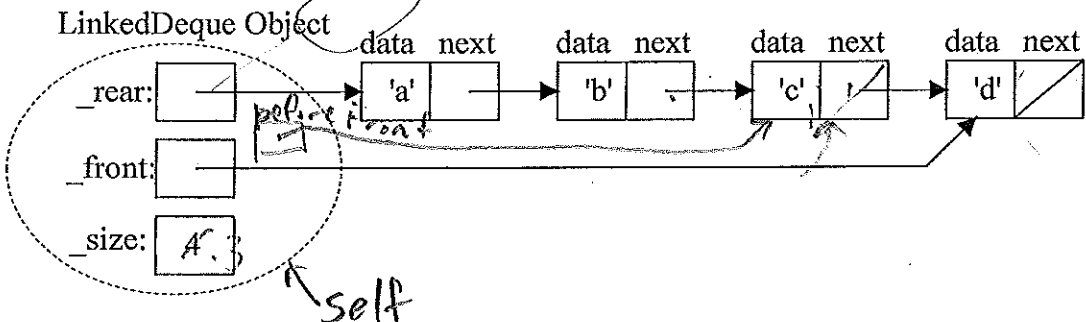
isEmpty	addFront	removeFront	addRear	removeRear	size
$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$

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

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

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

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
$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

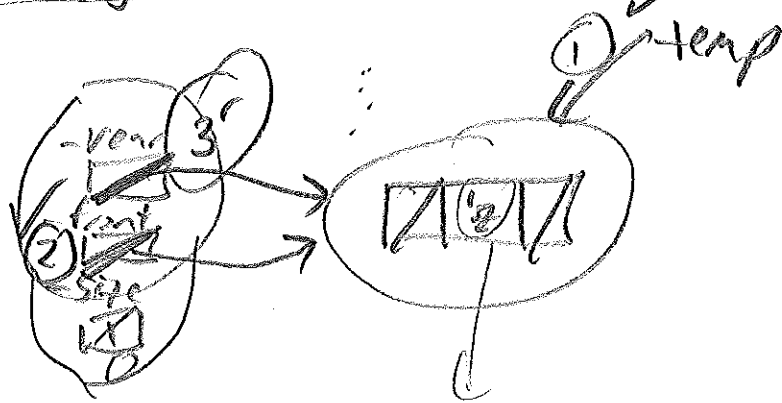
b) Suggest an improvement to the above linked implementation of the Deque to speed up some of its operations.

DoublyLinkedDeque removeFront

- ① temp = self._front
- ② self._front = temp.getPrevious()
- ③ self._front.setNext(None) if self._size == 1:
self._rear = None
else:
- ④ self._size -= 1
- ⑤ return temp.getData()

Special cases? Yes if self._size == 0:

- Empty Deque. — raise Exception ("cannot remove from empty deque")
- Removing last item in Deque



```

from node import Node

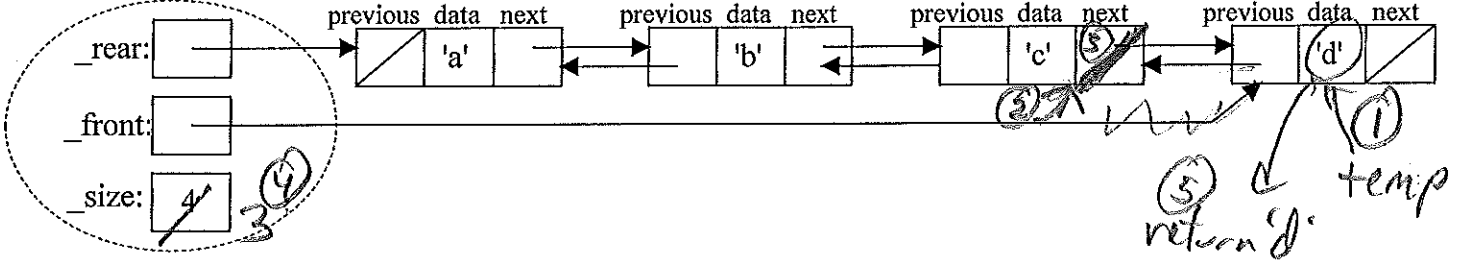
class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None

    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

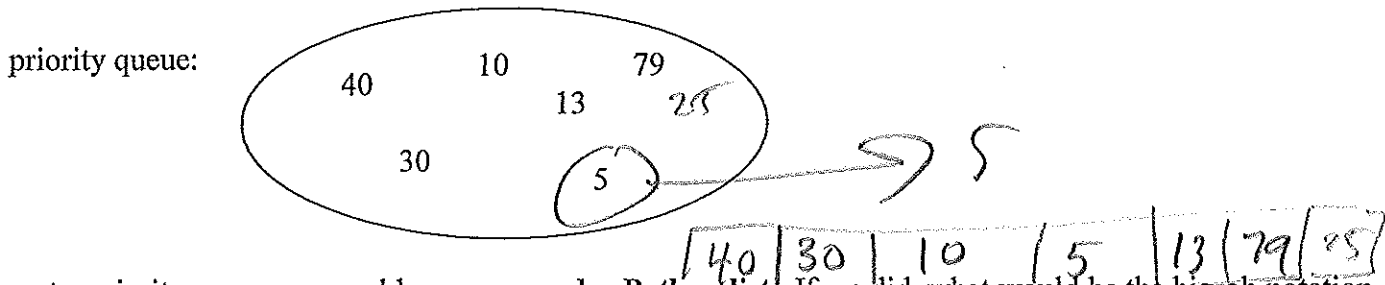


a) Determine the big-oh, $O()$, for each Deque operation assuming the above doubly-linked implementation. Let n be the number of items in the Deque.

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 priority 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?



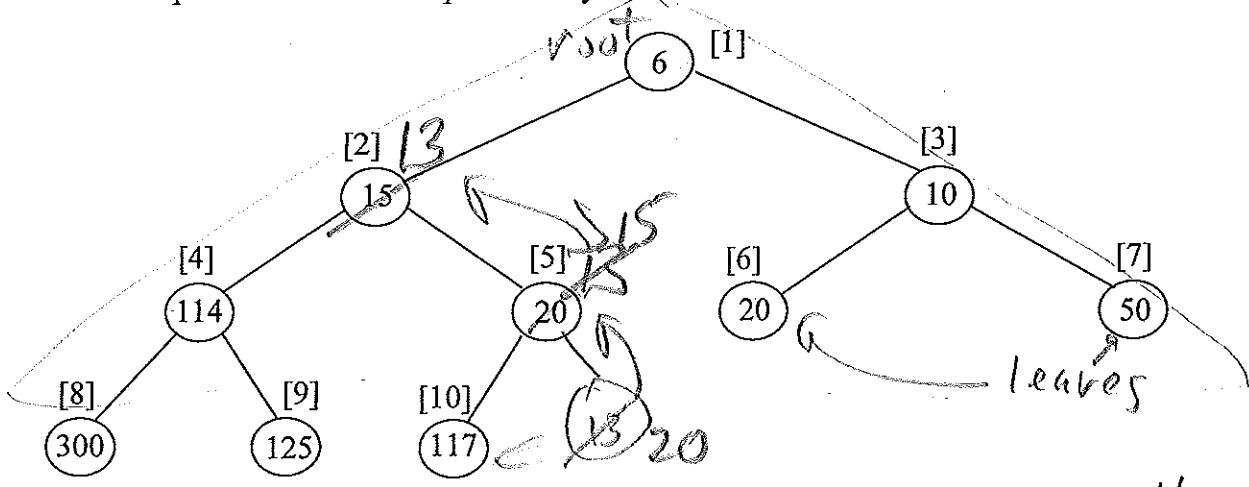
b) To implement a priority queue, we could use an **unordered Python list**. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue: $O(1)$ - append to right end of Python 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: $O(1)$

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 \leq either of its children. An example of a *min heap* “viewed” as a complete binary tree would be:



Python List actually used to store heap items

	0	1	2	3	4	5	6	7	8	9	10	11
	not used	6	15	10	114	20	20	50	300	125	117	13

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: $2 * i$
- its right child if it exists: $2 * i + 1$
- its parent if it exists: $i // 2$

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.

```

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

    def percUp(self, currentIndex):
        parentIndex =
        while

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