Fall 2017 Data Structures - Test 1 Name: Mark F.

Question 1. (4 points) Consider the following Python code.

```
for i in range(n):
    for j in range(n):
        print(i, j)

for k in range(n):
    print( k )
```

$O(n^2)$

+2 $O(n^3)$

What is the big-oh notation $O(\,)$ for this code segment in terms of n?

Question 2. (4 points) Consider the following Python code.

```
for i in range(n):
    j = n
    while j > 0:
        for k in range(n):
            print(i, j, k)

    j = j // 2
```

$O(n^2 \log_2 n)$

What is the big-oh notation $O(\,)$ for this code segment in terms of n?

+2 $O(n \log n)$

Question 3. (4 points) Consider the following Python code.

```
def main(n):
    for i in range(n):      ~ n x
        doSomething(n)      ~ n^3
        doMore(n)           ~ n^2

def doSomething(n):         n^3
    for k in range(n):      ~
        doMore(n)           ~ n^2
        print(k)

def doMore(n):
    for j in range(n*n):
        print(j)

main(n)
```

$O(n^4)$

What is the big-oh notation $O(\,)$ for this code segment in terms of n? +2 $O(n^3)$

Question 4. (8 points) Suppose a $O(n^4)$ algorithm takes 10 second when n = 1000. How long would the algorithm run when n = 10,000?

$T(n) = c\,n^4$

$T(1000) = c\,1000^4 = 10\text{ sec}$

$c = \frac{10\text{ sec}}{1000^4} = \frac{10}{10^{12}} = \frac{1}{10^{11}}\text{ sec}$

$T(10{,}000) = c\,10000^4 = c\,10^{16}$

$= \frac{1\text{ sec}}{10^{11}}\,10^{16}$

$= 10^5\text{ sec}$

$= 100{,}000\text{ sec}$

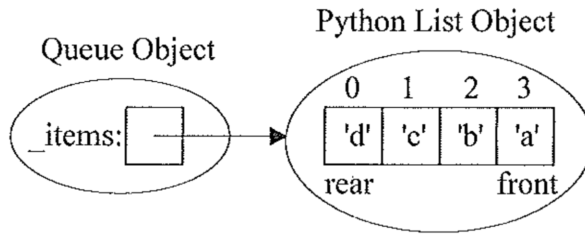
Question 5. (10 points) Why should you design a program instead of "jumping in" and start by writing code?

Without designing first you will be writing code that you'll need to modify or throw away later as you work further.

30

Question 6. A FIFO (First-In-First-Out) queue allows adding a new item at the rear using an enqueue operation, and removing an item from the front using a dequeue operation. One possible implementation of a queue would be to use a built-in Python list to store the queue items such that

- the **rear** item is **always stored at index 0,**
- the **front** item is always at index len(self._items) -1, or -1



a) (6 points) Complete the big-oh O ( ), for each Queue operation, assuming the above implementation. Let n be the number of items in the queue.

6

isEmpty	enqueue(item)	dequeue	peek - returns front item without removing it	__str__	size
O(1)	O(n)	O(1)	O(1)	O(n)	O(1)

b) (9 points) Complete the method for the dequeue operation, **including the precondition check to raise an exception if it is violated.**

```
def dequeue(self):
    """Removes and returns the Front item of the Queue
       Precondition:  the Queue is not empty.
       Postcondition: Front item is removed from the Queue and returned"""
```

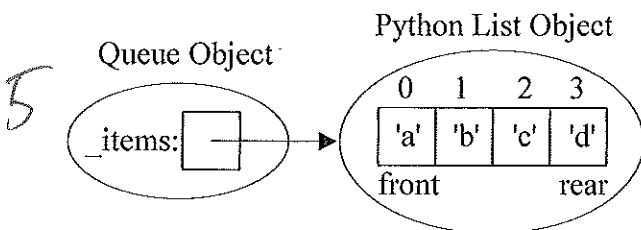
9

+4 (if len(self._items) == 0:
raise ValueError ("Cannot dequeue from empty queue")

+5 (return self._items.pop()

c) (5 points) An alternate Queue implementation would swap the location of the front and rear items as in:

5

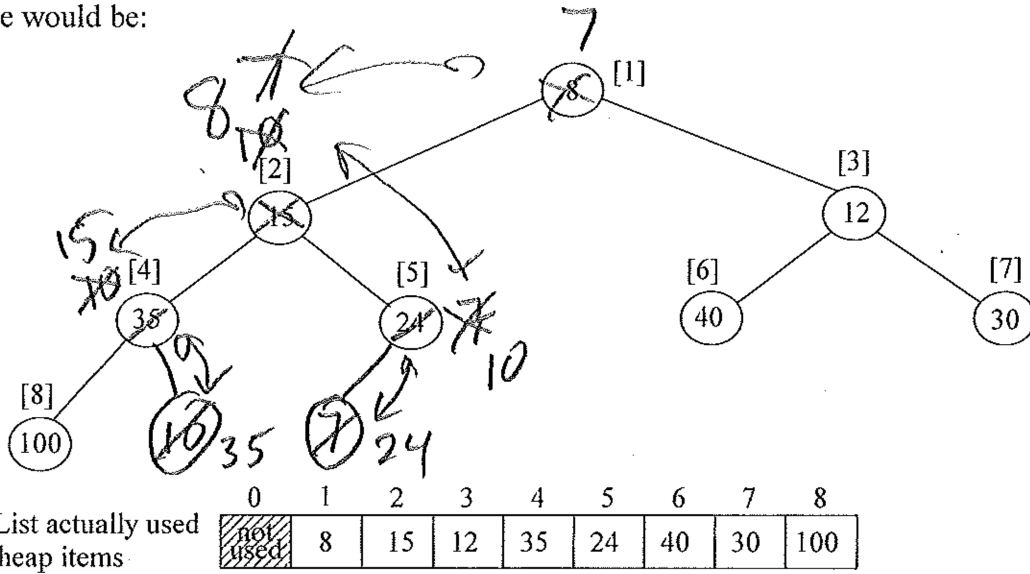


Why is this alternate implementation probably not very helpful with respect to the Queue's performance?

Makes O(1) for enqueue, but dequeue goes from O(1) to O(n).

20

Question 7. Consider the binary heap approach to implement a priority queue. A Python list is used to store a *complete binary tree* (a full tree with any additional leaves as far left as possible) with the items being arranges 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:

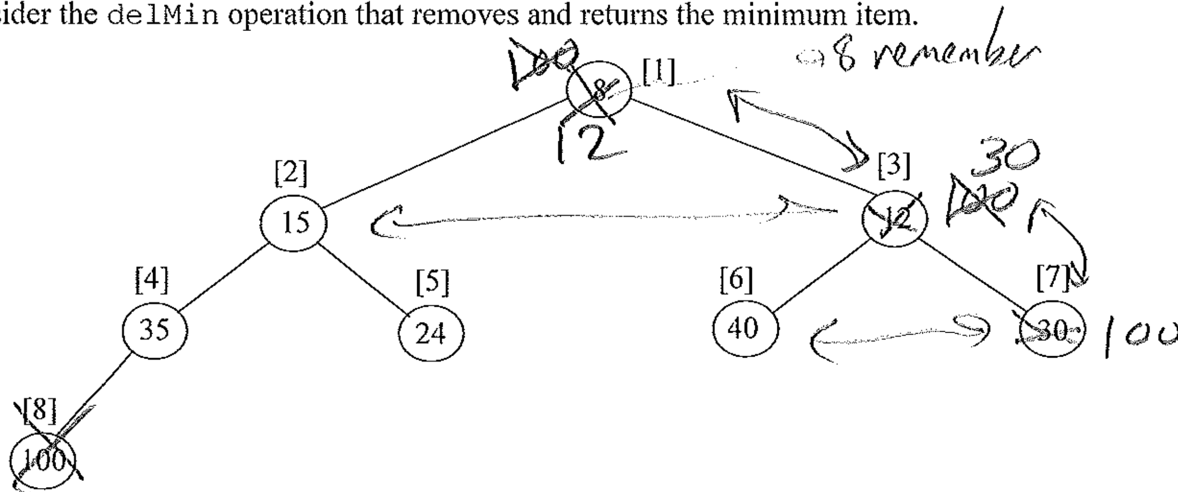


Python List actually used to store heap items

0	1	2	3	4	5	6	7	8
not used	8	15	12	35	24	40	30	100

a) (7 points) What would the above heap look like after inserting 10 and then 7 (show the changes on above tree)

Now consider the `delMin` operation that removes and returns the minimum item.



b) (2 point) What item would `delMin` remove and return from the above heap? 8

c) (7 points) What would the above heap look like after a `delMin` operation? (show the changes on above tree)

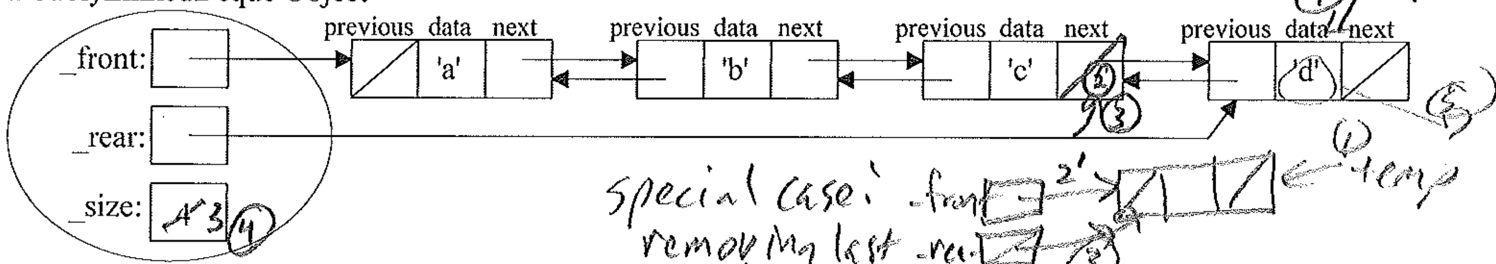
d) (3 points) What is the big-oh notation for **both** of the `insert` and `delMin` operations, where n is the number of items in the heap? $O(\log_2 n)$ of random integers +20(n log₂ n)

e) (6 points) Performing 20,000 `inserts` into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 `delMin` operations, it takes 0.39 seconds. Explain why 20,000 `delMin` operations take more time than the 20,000 `insert` operations?

An inserted item starts as a leaf and percolates up a branch of the tree by comparing with its parent. A delMin cause the "last" item to move to the root and percolates down each level by comparing its two children and then comparing with the smaller child. DelMin needs two comparison to move down a level while insert only needs one.

25

Question 8. The `Node2Way` and `Node` classes can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:



a) (6 points) Complete the big-oh O ( ), for each DoublyLinkedDeque operation, assuming the above implementation. Let n be the number of items in the DoublyLinkedDeque.

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

b) (14 points) Complete the `removeRear` method for the above DoublyLinkedDeque implementation.

```
class DoublyLinkedDeque(object):
    """ Doubly-linked list based deque implementation."""

    def __init__(self):
        self._front = None
        self._rear = None
        self._size = 0

    def removeRear(self):
        """Removes and returns the rear item of the Deque
           Precondition:  the Deque is not empty.
           Postcondition: Rear item is removed from the Deque
                          and returned. """
        if self._size == 0:
            raise ValueError("Cannot remove Rear from Empty Deque")
        temp = self._rear
        if self._size == 1:   special case
            self._front = None
        else:
            temp.getPrevious().setNext(None)
        self._rear = temp.getPrevious()
        self._size -= 1
        return temp.getData()
```

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

```
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
```

+2 +3

c) (5 points) Why would using singly-linked nodes (i.e., only Node objects with data and next) to implement the Deque lead to poor performance (i.e., cause some Deque operations to have worse big-oh notations)? Justify your answer. Use singly-linked nodes would cause removeRear to have a O(n)

25