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

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

```
for i in range(n):          - n
    for j in range(n * n):  - n^2
        for k in range(n):  - n
            print( i, j, k )
```

What is the big-oh notation $O()$ for this code segment in terms of n? $O(n^4)$

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

```
i = 1
while i < n:                ← log2n
    for j in range(n):      n
        print(i, j)
    for k in range(n):      n
        print(i, k)

    i = i * 2
```

Not nested $n+n = 2n$ $O(n)$

What is the big-oh notation $O()$ for this code segment in terms of n? $O(n \log_2 n)$

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

```
def main(n):
    for i in range(n):
        doSomething(n)

def doSomething(n):
    for k in range(n):
        doMore(n)
        print(k)

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

main(n)
```

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

Question 4. (8 points) Suppose a $O(n^3)$ algorithm takes 1 second when n = 1000. How long would the algorithm run when n = 10,000?

$O(n^3) \Rightarrow T(n) = c\,n^3$

$T(1000) = c\,1000^3 = 1\text{ sec}$

$c = \frac{1\text{ sec}}{1000^3} = 10^{-9}\text{ sec}$

$T(10{,}000) = c\,10{,}000^3 = c\,10^{12}$

$= 10^{-9}_{\text{sec}} \cdot 10^{12}$

$= 10^3\text{ sec}$

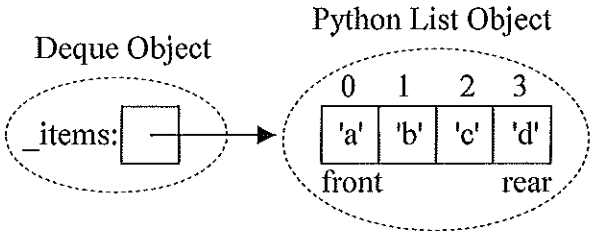
Question 5. (10 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?

The programmer using the method/function is using it incorrectly. By raising an exception they immediately find out about the error instead of later during the execution of program when the real cause of the error is harder to track down.

30

Question 6. 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 built-in Python list to store the Deque items such that

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



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

6

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

b) (7 points) Complete the method for the `removeFront` operation including the precondition check.

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

7

```
if len(self._items) == 0:
    raise ValueError("Cannot remove from empty Deque")
return self._items.pop(0)
```

c) (7 points) Complete the method for the `__str__` operation.

```
def __str__(self):
    """Returns the string representation of the Deque.
       Precondition:  none
       Postcondition: Returns a string representation of the Deque from the
       front item thru the rear item with a blank space between each item."""

    resultStr = ""
```

7

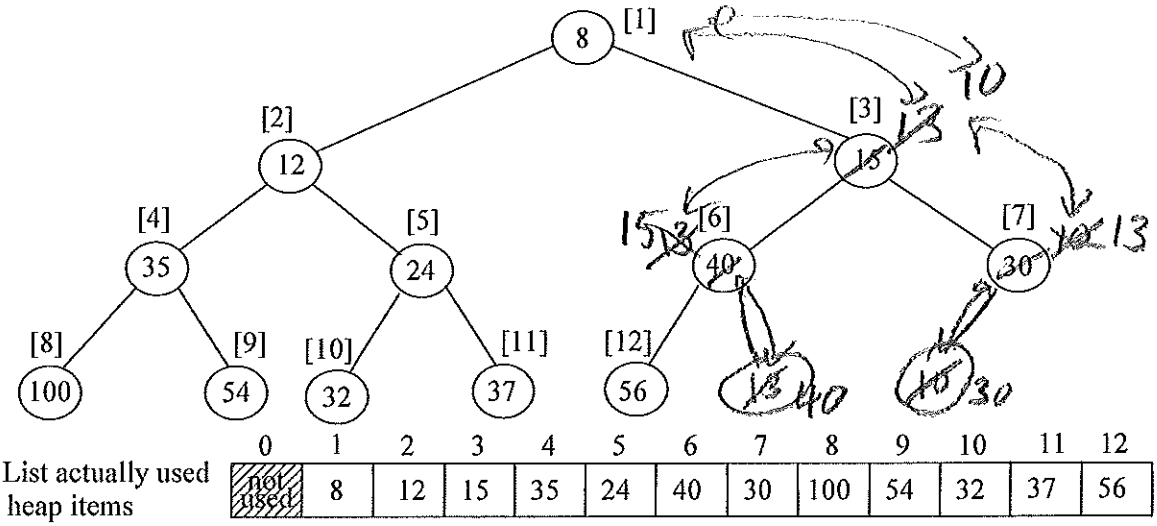
```
for item in self._items:
    resultStr += str(item) + " "
return resultStr
```

+1

+1

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:



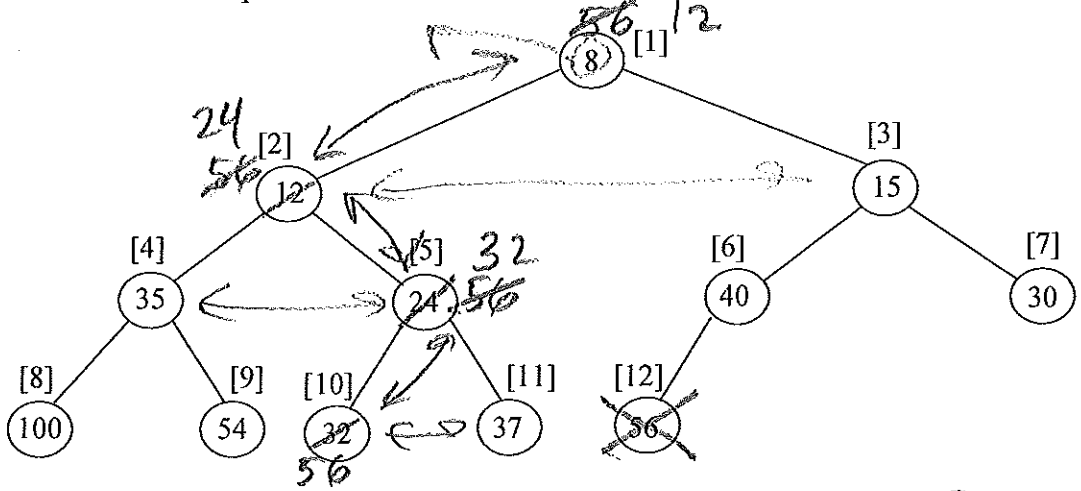
	0	1	2	3	4	5	6	7	8	9	10	11	12
Python List actually used to store heap items	not used	8	12	15	35	24	40	30	100	54	32	37	56

a) (3 points) For the above heap, the list 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 (integer division)

b) (7 points) What would the above heap look like after inserting 13 and then 10 (show the changes on above tree)

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



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

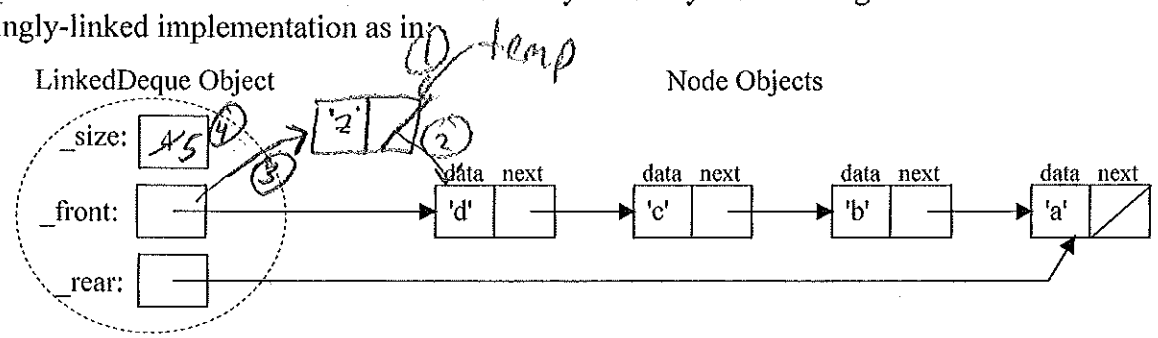
d) (7 points) What would the above heap look like after a `delMin` operation? (show the changes on above tree)

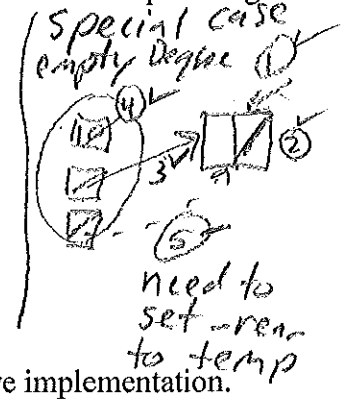
e) (6 points) Explain why **both** of the `insert` and `delMin` operations are $O(\log_2 n)$, where n is the number of items in the heap.

insert: new item added at index n and it repeatly gets compared and maybe swap with its parent at index i//2. Since we can repeatedly divide n in half $O(\log_2 n)$ times, the new item can only perolate up $O(\log_2 n)$ levels of the tree.

delMin: similar, the last item gets moved to index 1 (root) and

Question 8. The Node class can be used to dynamically create storage for each new item added to a Deque using a singly-linked implementation as in:





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

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

b) (14 points) Complete the addFront method for the above LinkedDeque implementation.

```
class LinkedDeque(object):
    """ Singly-linked list based deque implementation."""

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

    def addFront(self, newItem):
        """ Adds the newItem to the front of the Deque.
            Precondition:  none """
```

```
temp = Node(newItem)
temp.setNext(self._front)
self._front = temp
self._size += 1
if self._size == 1:
    self._rear = temp
```

```
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
```

c) (5 points) Would using doubly-linked nodes (Node2Way with previous, data, and next) improvement the above implementation (i.e., speed up some of the queue operations enough to change their big-oh notation)? Justify your answer.

Yes, removeRear would go to O(1) since we can find the node before the rear by follow the rear nodes previous (e.g. self._rear.getPrevious())