Fall 2016 Data Structures - Test 1 Name: Mark F.

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

```
for i in range(n * n * n):
    j = 1
    while j < n:
        print( i, j )
        j = j * 2
```

$O(n^3 \log_2 n)$

$O(n^4)$ +2

$O(n \log n)$ +2

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 ~~n>1~~: j>1:
        print( i, j )
        j = j // 2

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

$O(n^2)$

$O(n \log_2 n)$ +2

$O(n^2 \log_2 n)$ +2

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

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

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

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

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

main(n)
```

$O(n^4)$

$O(n^5)$ +2

$O(n^3)$ +2

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

Question 4. (8 points) Suppose a $O(n^5)$ algorithm takes 10 second when n = 100. How long would the algorithm run when n = 1,000?

$T(n) = c\,n^5 \quad T(100) = c\,100^5 = 10\text{ sec}$

$c = \frac{10\text{ sec}}{100^5} = \frac{10\text{ sec}}{10^{10}}$

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

$T(1000) = c\,1000^5$

$= \frac{1\text{ sec}}{10^9} 1000^5$

$= \frac{10^{15}}{10^9} = 10^6\text{ sec}$

Question 5. (10 points) For a built-in Python list, explain each of the following average big-oh notations:

a) Why does `myList.insert(0, "item")` have an average big-oh of O(n), where n is the # of list items?

5 To insert an item at index 0, all n items in the list must be shifted right to create a hole for the new item.

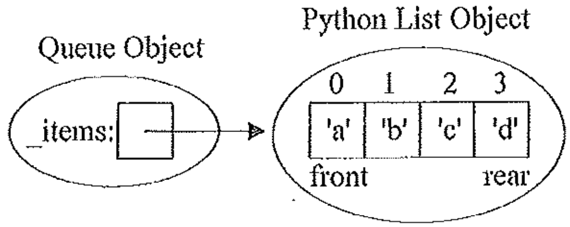
b) Why does `myList.append("item")` have an average big-oh of O(1), where n is the # of list items?

5 The Python list is assumed to have extra/unused space on its right end, so the new item can use the next free spot

30

Question 6. A FIFO 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 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 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(1)	O(n)	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

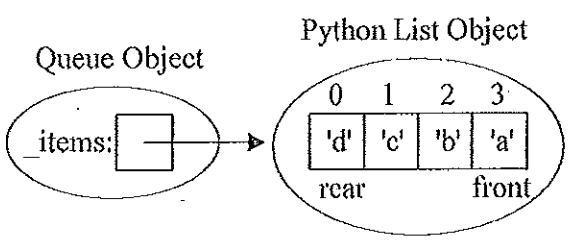
```
    if self.isEmpty():
        raise ValueError("Cannot dequeue from empty queue.")

    return self._items.pop(0)
```

$\underbrace{\text{return}}_{+2}$

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

5



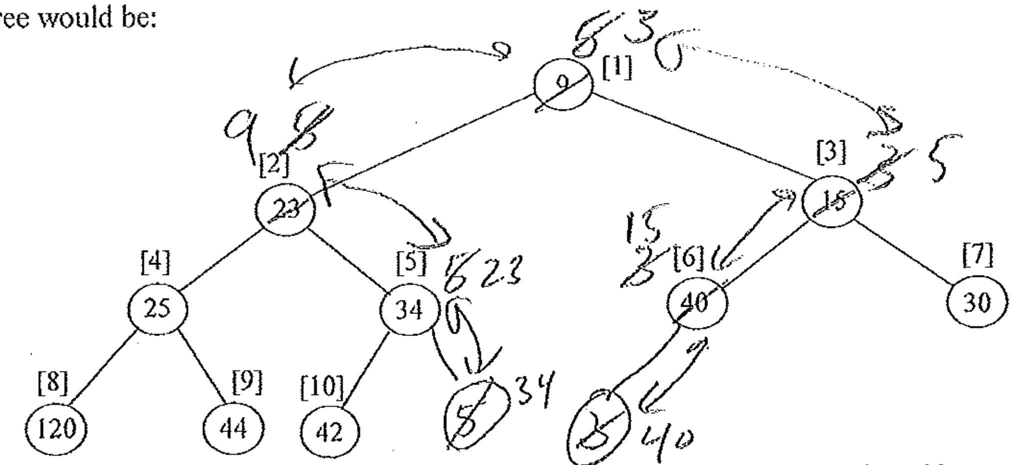
Why is this alternate implementation probably not very helpful with respect to the Queue's performance?

Now enqueue becomes O(n) and dequeue becomes O(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:

7



	0	1	2	3	4	5	6	7	8	9	10
Python List actually used to store heap items	not used	9	23	15	25	34	40	30	120	44	42

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:

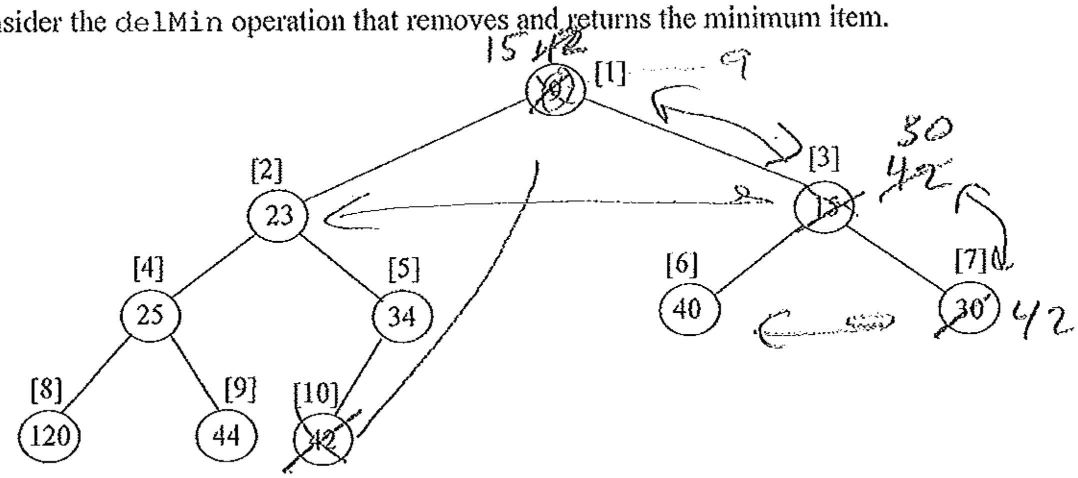
- its left child if it exists: 2*i
- its right child if it exists: 2*i + 1
- its parent if it exists: i//2

3

b) (7 points) What would the above heap look like after inserting 5 and then 3 (show the changes on above tree)

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

7



2 c) (2 point) What item would `delMin` remove and return from the above heap? 9

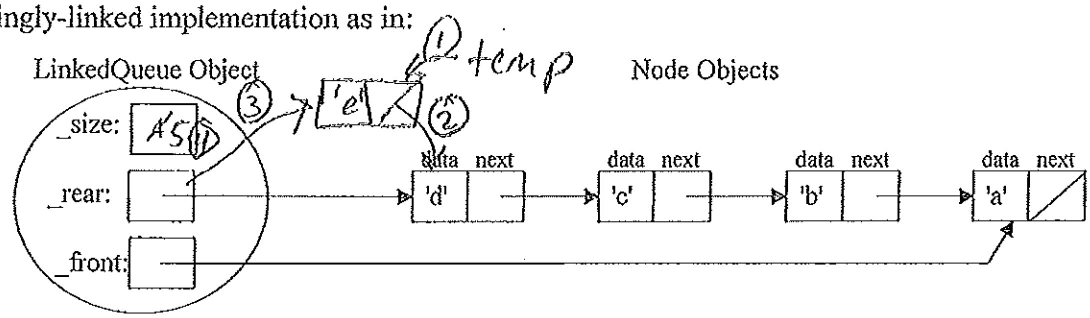
d) (7 points) What would the above heap look like after a `delMin` operation? (show the changes on above tree)

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 these 20,000 `delMin` operations take more time than the 20,000 `insert` operations? On an insert the new item is repeated compared with its parent until it finds the right spot. On a delMin the last item is moved to the root and percolates down by comparing its two children to find the min. Plus item moved in delMin is large and will likely percolate down more levels, than a new item percolates up.

6

25

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



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

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

b) (14 points) Complete the enqueue method for the above LinkedQueue implementation.

```
class LinkedQueue(object):
    """ Singly-linked list based Queue implementation."""

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

    def enqueue(self, newItem):
        """ Adds the newItem to the rear of the queue.
            Precondition:  none """
        temp = Node(newItem)
        temp.setNext(self._rear)
        self._rear = temp
        self._size += 1
        if self._size == 1:      # special case -- adding first item
            self._front = 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) Suggest an improvement to the above implementation to speed up some of the queue operations enough to change their big-oh notation? Justify your answer Two ways to get dequeue $O(1)$:

1) (best way) - have _front point to first Node in the linked list e.g., _front → ☐☐ → ☐☐ ... → ☐☐ ← _rear

2) Use doubly-linked list of Node2Way nodes.