Spring 2018 Data Structures - Test 1 Name: Mark F.

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

```
i = n
while i > 1:
    print(i)
    i = i // 2

for j in range(n * n):
    print(j)
```

$\} O(\log_2 n)$

$\} O(n^2)$

Not nested, so $O(n^2)$ overall

4

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

+2 $n^2 \log_2 n$

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

```
for i in range(n):          ~n x
    for j in range(n):      ~n x
        k = 1
        while k < n:
            print(i, j, k)
            k = k + 2
```

$\}$ loops $\frac{n}{2}$ times

$O(n^3)$

4

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

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

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

```
def main(n):
    i = n
    while i > 0:               ~log2n x
        for j in range(n):     ~n x
            doSomething(n)     ~n^3
        i = i // 2

def doSomething(n):
    for k in range(n):         ~n x
        doMore(n*n)            ~n^2

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

main(n)
```

n^3

$O(n^4 \log_2 n)$

4

12

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

+2 $O(n^4)$ +2 $O(n^3 \log_2 n)$

Question 4. (5 points) Suppose a $O(n^5)$ algorithm takes 1 second when n = 1000. How long would the algorithm run when n = 10,000?

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

$T(1000) = c\,1000^5 = 1$ sec

$c = \frac{1}{1000^5} = \frac{1}{10^{15}}$ sec

5

$T(10{,}000) = c\,10000^5 = c\,10^{20} =$

$= \frac{1}{10^{15}} 10^{20} = 10^5$ sec

$= 100{,}000$ sec

Question 5. (8 points) Why should medium/large size programs be written using function definitions instead of a single block of monolithic code written at the top-level (i.e., all statements written outside of any function)?

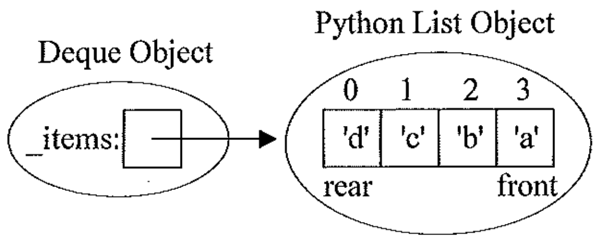
8

Medium/large size programs are hard to understand without decomposing into small subtasks. Coding the small subtasks as a function allows it to be tested separately, programmed by another team member, and possibly reused in a later program.

25

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 **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 Deque operation, assuming the above implementation. Let n be the number of items in the Deque.

6

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

b) (9 points) Complete the code for the `addRear` method, including any precondition check needed by raising an exception if it is violated.

9

```
def addRear(self, newItem):
    """Adds the newItem to the rear of the Deque
       Precondition:  none
       Postcondition: newItem has been added to the rear of the Deque"""
       self._items.insert(0,newItem)
```

c) (10 points) Complete the method for the `__str__` operation.

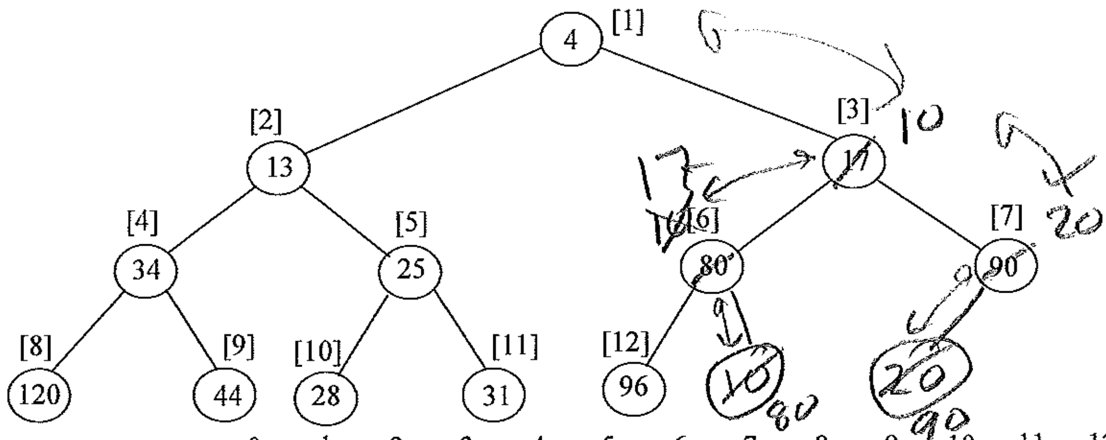
10

```
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 = "(front) "
    for index in range(len(self._items)-1,-1,-1):
        resultStr += str(self._items[index]) + " "
    resultStr += "(rear)"
    return resultStr
```

25

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



Python List actually used to store heap items

0	1	2	3	4	5	6	7	8	9	10	11	12
not used	4	13	17	34	25	80	90	120	44	28	31	96

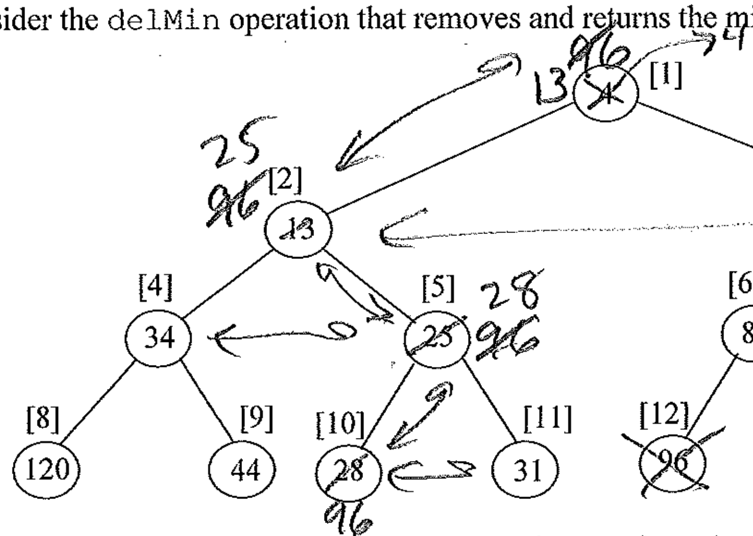
a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:

3
- 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) (7 points) What would the above heap look like after inserting 10 **and then** 20 (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? 4

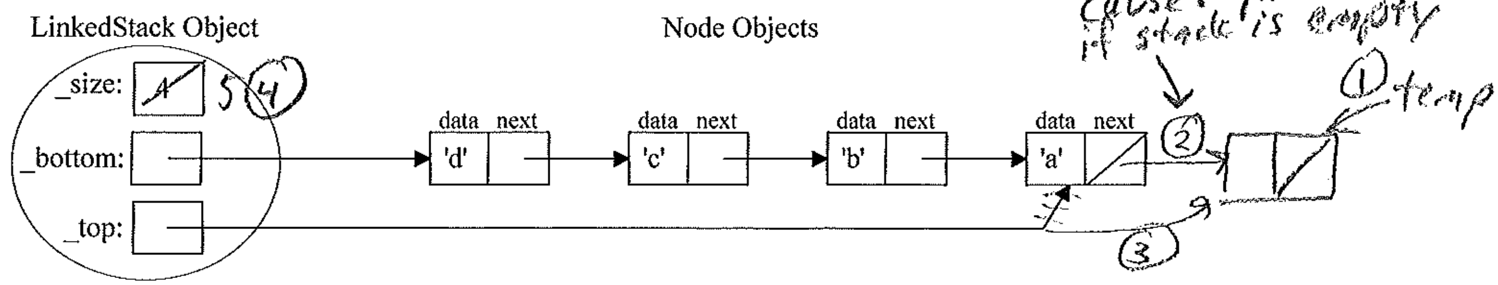
d) (7 points) What would the heap look like after `delMin`? (show the changes on tree in the middle of the page)

e) (6 points) What is the big-oh notation for the `delMin` operation? **(EXPLAIN YOUR ANSWER)**

6 The last item in the list is moved to index 1. At each iteration of the loop, it get swapped with its smaller child at 2*i or (2*i+1), so its index at least doubles. Since we can only double 1 $O(\log_2 n)$ times before reaching n, we only loop $O(\log_2 n)$ times.

25

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



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

6

isEmpty	size	pop	push(item)	__init__	__str__
O(1)	O(1)	O(n)	O(1)	O(1)	O(n)

b) (12 points) Complete the ~~addFront~~ push method for the above Linked~~Deque~~ Stack implementation.

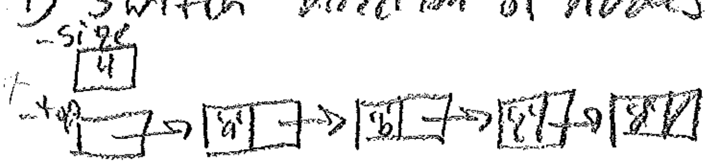
12

```
class LinkedStack(object):
    """ Singly-linked list based Stack implementation."""

    def __init__(self):
        self._size = 0
        self._bottom = None
        self._top = None

    def push(self, item):
        """ Adds the item to the top of the Stack.
            Precondition:  none """
        temp = Node(item)
        if self._size == 0:          +3 special case
            self._bottom = temp
        else:
            self._top.setNext(temp)
        self._top = temp
        self._size += 1
```

```
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
```

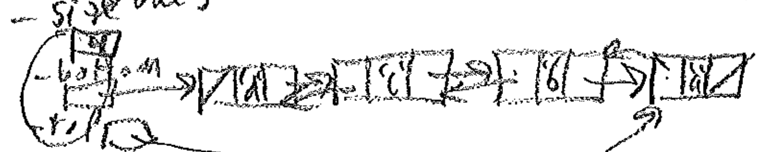
c) (7 points) Suggest an improvement to the above implementation to speed up some of the stack operations enough to change their big-oh notation? (Justify your answer) — Three okay answers:

1) Switch "direction" of nodes:



both pop + push are O(1)

2) use doubly-linked (Node2way) Nodes



both pop + push are O(1) 4

3) Use Python list with bottom at index 0.