Fall 2013 Name: Mark F.

Data Structures - Test 1

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

```
for i in range(n):        — n x
    for j in range(n*n):  — n^2 x
        print (i, j)
```

$n \times n^2 = n^3$

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

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

```
i = 1
while i < n:              — log n

    for j in range(n):    — n
        print(j)

    i = i * 2
```

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

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

```
def main(n):
    for i in range(n):    — n x
        doSomething(n)    — n x } 2n
        doMore(n)         — n x

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

def doMore(n):
    for k in range(n):    — n x
        print(k)

main(n)
```

$\cancel{n} \times 2n = 2n^2$

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

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

$O(n^3) \Rightarrow T(n) \doteq cn^3 \quad T(1000) = c\,1000^3 = 10\text{sec}$

$c = \frac{10\text{sec}}{1000^3} = \frac{10}{10^9}\text{sec} = 10^{-8}\text{sec} \quad T(10{,}000) = c\,10{,}000^3 = c\,10^{12}$

$= 10^{-8}\text{sec}\,10^{12} = 10^4\text{sec}$

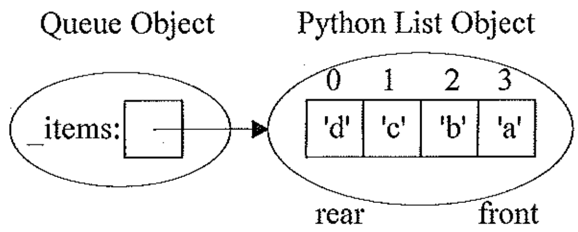
$= 10{,}000\text{sec}$

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

If precondition is violated, then the method/fn will not work correctly. Better to discovery the error as soon as possible rather than later when tracking the error down is more difficult.

Question 6. Consider the following FIFO (First-In-First-Out) Queue implementation utilizing a Python list: Recall that a queue is a linear data structure that allows adding new items at the rear and removing items from the front. One possible implementation of a queue would be to use a built-in Python list to store the 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) (8 points) Complete the big-oh O ( ), for each Queue operation, assuming the above implementation. Let n be the number of items in the Queue.

isEmpty	enqueue	dequeue	size
O(1)	O(n)	O(1)	O(1)

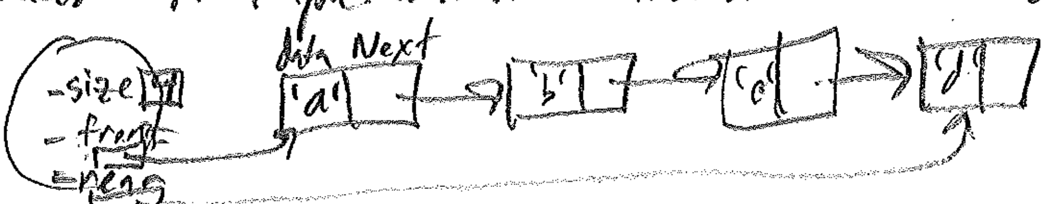
b) (10 points) Complete the method for the dequeue operation including the precondition check.

```
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"""
```

return self._items.pop()

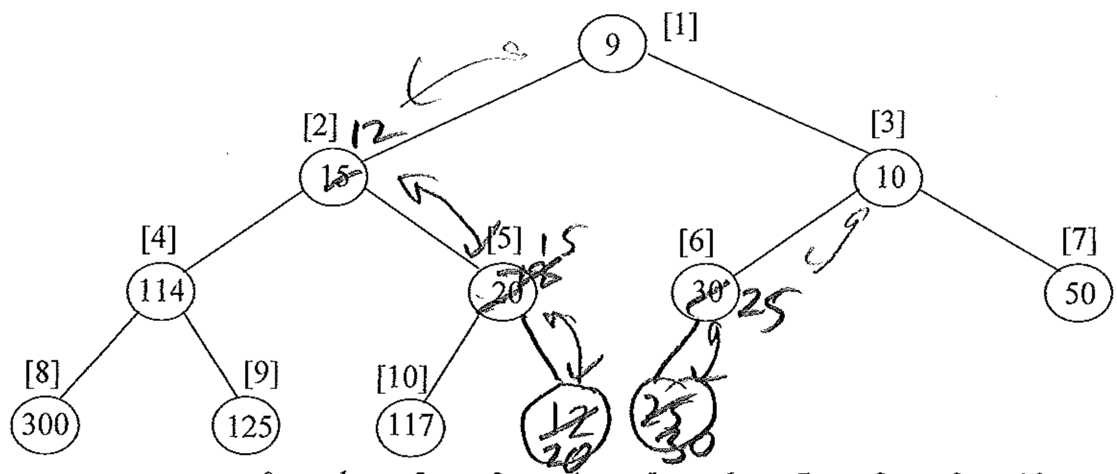
c) (7 points) Suggest an alternate Queue implementation to speed up some of its operations.

Linked list implementation would have all operations O(1)



(or "circular array")

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	9	10
not used	9	15	10	114	20	30	50	300	125	117

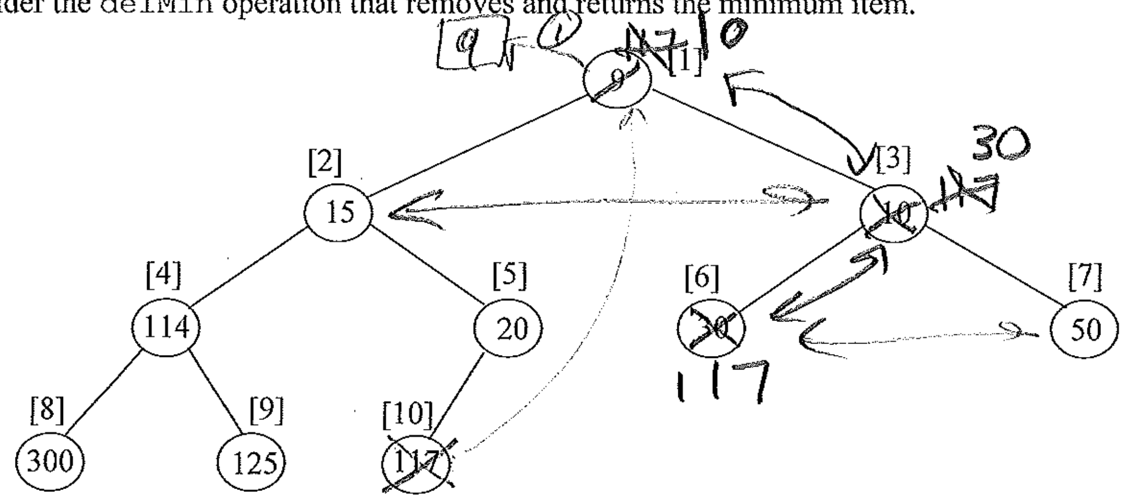
a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:

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

b) (7 points) What would the above heap look like after inserting 12 and then 25 (show the changes on above tree)

c) (3 points) What is the big-oh notation for inserting a new item in the heap? O(log n) since heap's height is $\log_2 n$

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



Python List actually used to store heap items

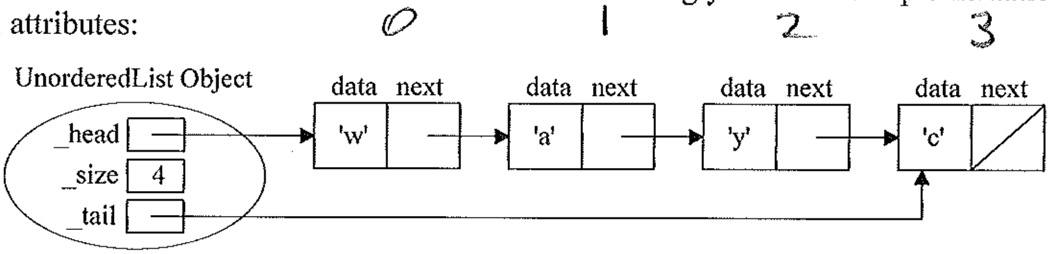
0	1	2	3	4	5	6	7	8	9	10
not used	9	15	10	114	20	30	50	300	125	117

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

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

f) (3 points) What is the big-oh notation for `delMin`? $O(\log_2 n)$

Question 8. The textbook's **unordered list** ADT uses a singly-linked list implementation. I added the `_size` and `_tail` attributes:



a) (15 points) The `insert(position, item)` method adds the `item` to the list at the specified `position`. Unlike the textbook's implementation, ASSUME that the list may contain duplicate items!!! The precondition is that `position` is a nonnegative integer. If `position` is 0, then add it to the head of the list. If `position` is `_size` **or bigger**, then add it to the tail of the list. Complete the `insert(position, item)` method code including the precondition check.

```
class UnorderedList:

   def __init__(self):
      self._head = None
      self._size = 0
      self._tail = None

   def insert(self, position, item):
      if position < 0:
         raise (IndexError, "position must be nonnegative")
      temp = Node(item)
      if position == 0:
         temp.setNext(self._head)
         self._head = temp
         if self._size == 0:
            self._tail = temp
      elif position >= size:
         if self._size == 0:
            self._head = temp
         else:
            self._tail.setNext(temp)
         self._tail = temp
      else:
         current = self._head
         for c in range(position - 1):
            current = current.getNext()
         temp.setNext(current.getNext())
         current.setNext(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
```

b) (10 points) Assuming the unordered list ADT described above **that allows duplicate items**. Complete the big-oh $O(\,)$ for each operation. Let n be the number of items in the list.

`insert(position,item)`	`pop()` removes and returns tail item	`length()` returns number of items in the list	`append(item)` adds item to the tail of list	`add(item)` adds item to the head of list
$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

↑ needs to reset _tail pointer.