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

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

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

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

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

    i = i * 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):
        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)
```

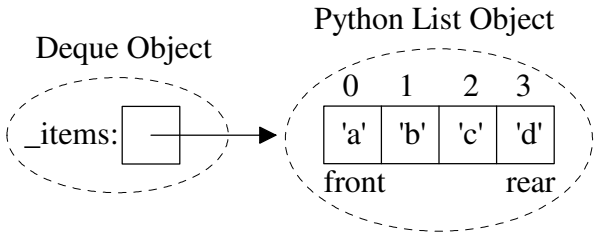
What is the big-oh notation $O(\ )$ for this code segment in terms of n?

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?

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

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.

isEmpty	addRear	removeRear	addFront	removeFront	__str__

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

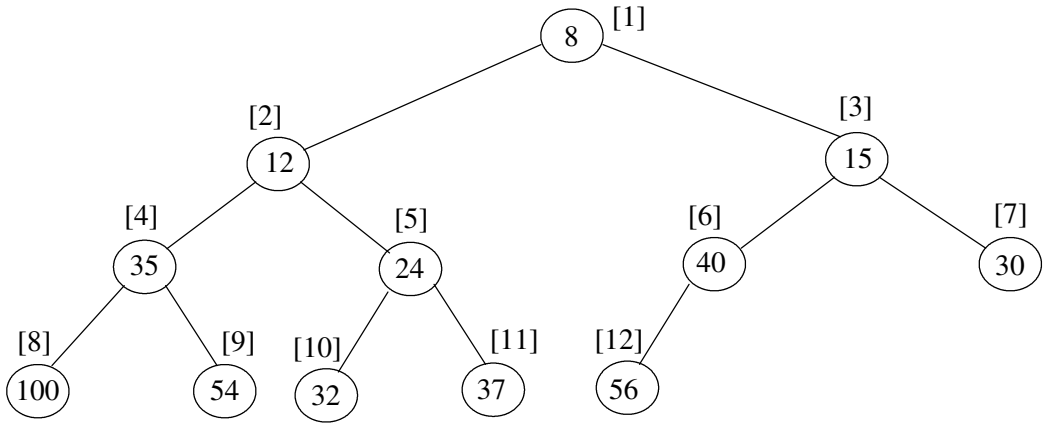
```
def removeFront(self):
    """Removes and returns the front item of the Deque
       Precondition:  the Deque is not empty.
       Postcondition: Front item is removed and returned from the Deque"""
```

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 = ""
```

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

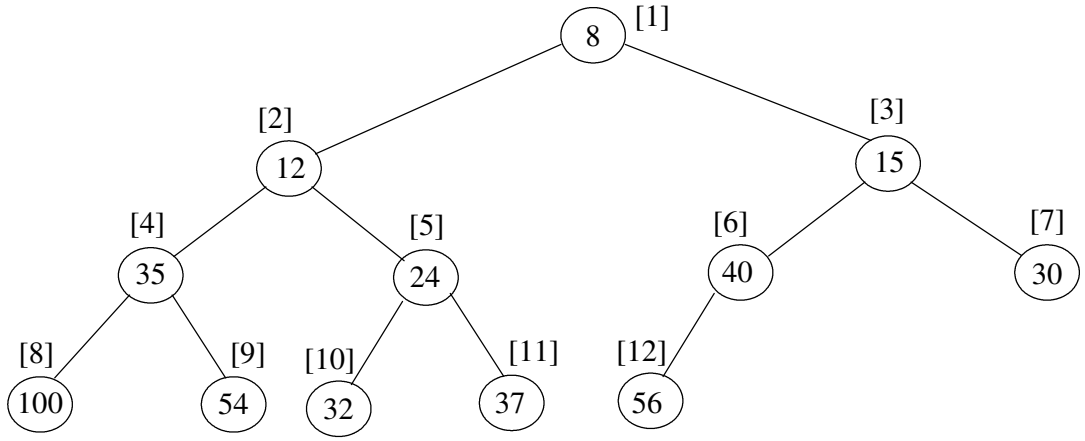
0	1	2	3	4	5	6	7	8	9	10	11	12
not used	8	12	15	35	24	40	30	100	54	32	37	56

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:
- its right child if it exists:
- its parent if it exists:

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.

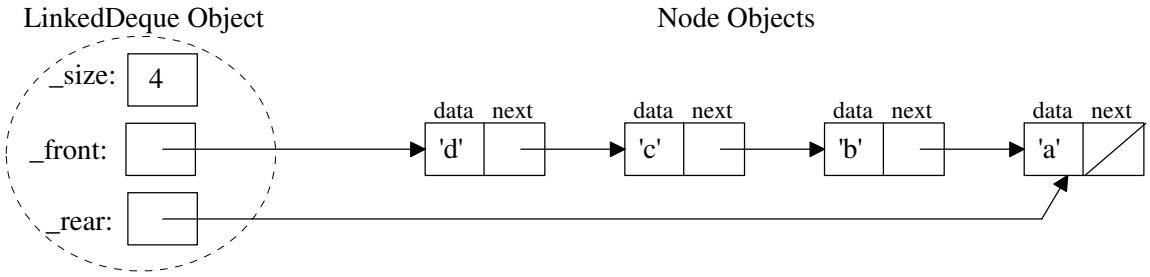


c) (2 point) What item would `delMin` remove and return from the above heap?

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.

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__

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 """
.
```

```
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.