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

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

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

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

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)
```

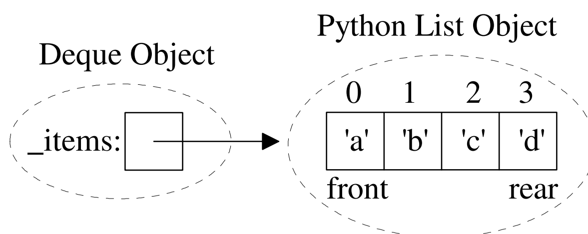
What is the big-oh notation $O(\ )$ for this code segment in terms of n?

Question 4. (9 points) Suppose a $O(n^3)$ algorithm takes 10 second when n = 1000. How long would you expect the algorithm to run when n = 10,000?

Question 5. (9 points) Why should any 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.

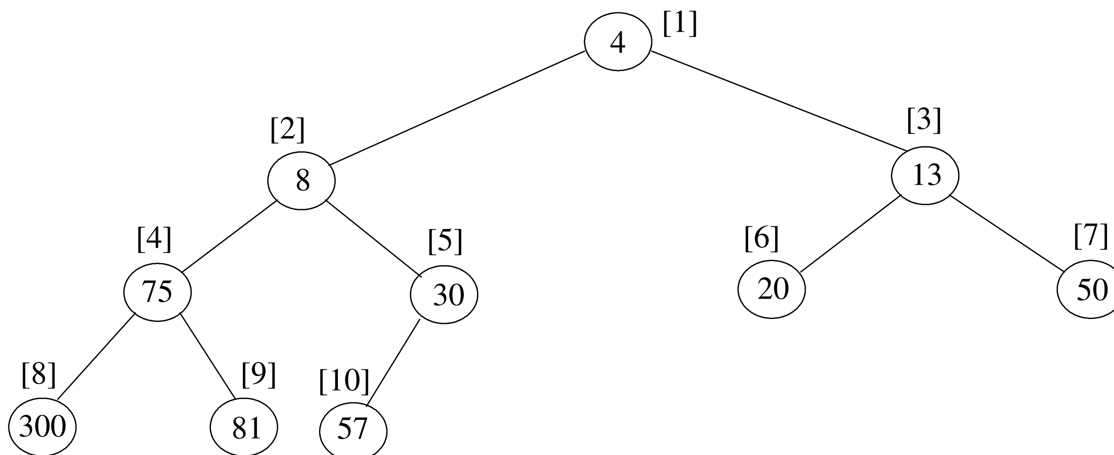
isEmpty	addRear	removeRear	addFront	removeFront	size

b) (9 points) Complete the method for the `removeRear` operation including the precondition check.

```
def removeRear(self):
    """Removes and returns the rear item of the Deque
       Precondition:  the Deque is not empty.
       Postcondition: Rear item is removed from the Deque and returned"""
```

c) (5 points) Suggest an alternate Deque implementation to speed up some of its operations.

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	4	8	13	75	30	20	50	300	81	57

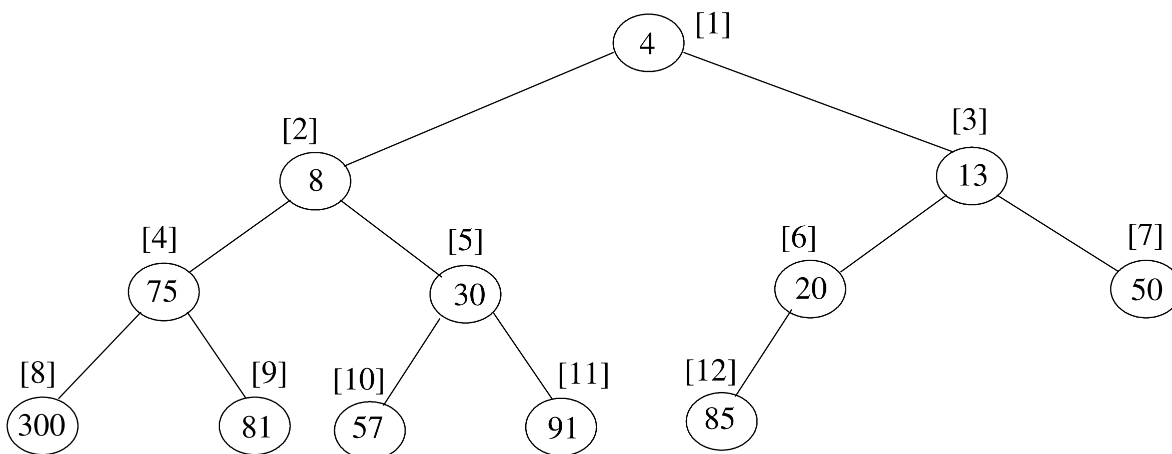
a) (3 points) For the above heap, the list indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:

- 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 6 and then 2 (show the changes on above tree)

c) (3 points) What is the big-oh notation for inserting a new item in the heap?

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



Python List actually used to store heap items

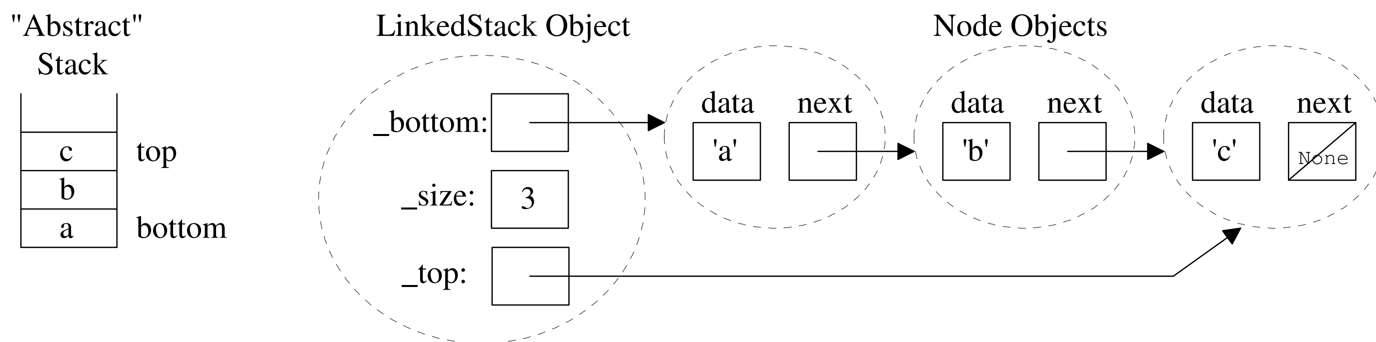
0	1	2	3	4	5	6	7	8	9	10	11	12
not used	4	8	13	75	30	20	50	300	81	57	91	85

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

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

f) (3 points) Why does a `delMin` operation typically take longer than an `insert` operation?

Question 8. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. Consider the following `LinkedStack` class using this Node class. Conceptually, a `LinkedStack` object would look like:




```
class LinkedStack(object):
    """ Link-based stack implementation."""

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


    def __str__(self):
        """ Returns a string with items strung from bottom to
            top. Each item should be separated by a space. """
```

```
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
```

a) (11 points) Complete the `__str__` method above.

b) (7 points) Assuming the stack ADT described above. Complete the big-oh *O* ( ) for each stack operation. Let n be the number of items in the stack.

push(item)	pop()	peek()	size()	__str__()

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