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

```python
for i in range(n*n*n*n):
    for j in range(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.

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

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

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

```python
def main(n):
    for i in range(n):
        doSomething(n)
        doMore(n)

def doSomething(n):
    for j in range(n*n):
        doMore(n)

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

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

Question 4. (6 points) Suppose a $O(\ n^3 \ )$ algorithm takes 1 second when $n = 100$. How long would the algorithm run when $n = 1,000$?

Question 5. (7 points) Why should you design a program first instead of “jumping in” and start by writing code?
Question 6. Consider the following Stack implementation utilizing a Python list: **Note: top is at index 0.**

<table>
<thead>
<tr>
<th>Abstract Stack</th>
<th>Stack Object</th>
<th>Python list Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>top</td>
<td></td>
</tr>
<tr>
<td></td>
<td>items:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bottom</td>
<td>top</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>bottom</td>
</tr>
<tr>
<td>top is at index 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) (6 points) Complete the big-oh notation for the Stack methods assuming the above implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(newItem)</td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
</tr>
<tr>
<td>peek()</td>
<td></td>
</tr>
<tr>
<td>size()</td>
<td></td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
</tr>
<tr>
<td><strong>str</strong></td>
<td></td>
</tr>
</tbody>
</table>

b) (14 points) Complete the code for the `peek` and `push` methods, **including the precondition check to raise an exception if it is violated.**

```python
class Stack:
    def __init__(self):
        self._items = []

    def peek(self):
        """Returns the top item of the stack without removing it.
        Precondition: the stack is not empty.
        Postcondition: the top item is returned without removed it."""

    def push(self, newItem):
        """Adds the newItem to the top of the stack.
        Precondition: none.
        Postcondition: the newItem has been added to the top of the stack."""
```

c) (5 points) Suggest an improvement to the above Stack 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 arranged 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:

```
10
   / \
  13   19
 / \   / \\
34 25 30 90
|   |   |   |
44 88 91 46
|   |   |   |
120 88 91 46
```

Python List actually used to store heap items:
```
[10, 13, 19, 34, 25, 30, 46, 90, 120, 44, 88, 91]
```

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 14 and then 18 (show the changes on above tree)

Now consider the \texttt{delMin} operation that removes and returns the minimum item.

```
10
   / \
  13   19
 /     / \\
34 25 30 90
|   |   |   |
44 88 91 46
|   |   |   |
120 88 91 46
```

```
[10, 13, 19, 34, 25, 30, 46, 90, 120, 44, 88, 91]
```

c) (2 point) What item would \texttt{delMin} remove and return from the above heap?

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

e) (6 points) Explain why the \texttt{delMin} operation is \(O(\log_2 n)\), 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 queue using a singly-linked implementation as in:

```
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
```

```
Spring 2019
Data Structures - Test 1
Name: ______________________
```

LinkedQueue Object

```
LinkedQueue Object

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>'b'</td>
</tr>
<tr>
<td>'b'</td>
<td>'c'</td>
</tr>
<tr>
<td>'c'</td>
<td>'d'</td>
</tr>
</tbody>
</table>
```

a) (5 points) Complete the expected big-oh $O(\cdot)$, for each LinkedQueue operation, assuming the above implementation. Let $n$ be the number of items in the LinkedQueue.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>enqueue</td>
<td>$O(\cdot)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(\cdot)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>$O(\cdot)$</td>
</tr>
</tbody>
</table>

b) (20 points) Complete the enqueue method for the above LinkedQueue implementation.

```
class LinkedQueue(object):
    """ Singly-linked list based queue implementation."""
    def __init__(self):
        self._front = None
        self._rear = None
        self._size = 0

    def enqueue(self, newItem):
        """ Adds the newItem to the rear of the queue.
          Precondition: none """
        self._size += 1
        if self._size == 1:
            self._front = self._rear = Node(newItem)
        else:
            self._rear.next = Node(newItem)
            self._rear = self._rear.next
```