Test 1 will be Thursday February 21<sup>th</sup> in class. It will be closed-book and notes, except for one 8.5” x 11” sheet of paper (you can use front and back) containing any notes that you want AND the Python Summary handout. The test will cover the following topics (and maybe more).

**Chapter 1. Introduction**

Python control structures: if, while, for,

Python built-in data structures: list, dictionary, string

Preconditions, postconditions, and raising exceptions to enforce the precondition

Defining classes (e.g., Dic) including inheritance, overriding methods

**Chapter 2. Algorithm Analysis**

Machine dependent measures of performance: program running time and instruction count

Machine independent measures of performance: big-o, orders of complexity: constant O(1), logarithmic O(log n), linear O(n), “n log n”/log linear O(n log n), quadratic O(n²), cubic O(n³), exponential O(2ⁿ)

Complexity analysis of an algorithm to determine its big-oh notation

Implementation of Python lists as an array of object references with implications on operation big-oh (e.g., pop() is O(1) while pop(0) is O(n), etc.)

**Chapter 3. Basic Data Structures**

General concept of a stack: LIFO, top and bottom

Stack Operations: pop, push, peek, size, isEmpty, __str__

Stack Implementations: Python list to store stack items and linked list of Nodes to store stack items including big-oh of operations

Stack Applications: general idea of using a stack to do parentheses matching and palindrome checking

General concept of a queue: FIFO, front and rear

Queue Operations: enqueue, dequeue, peek, size, isEmpty, __str__

Queue Implementations: Python list to store queue items and linked list of Nodes to store stack items including big-oh of operations

General concept of a deque: double ended queue, front and rear

Deque Operations: addFront, addRear, removeFront, removeRear, size, isEmpty, __str__

Deque Implementations: Python list to store deque items, singly-linked list of Nodes to store deque items, and doubly-linked list of Nodes (e.g, Node2Way) including big-oh of operations

Deque Applications: general idea of using deque to do palindrome checking

General concept of a list: head, tail, index

Categories of List operations: index-based, content-based, cursor-based

Unordered List operations and implementation with a singly-linked list of Nodes including big-oh of operations

Ordered List operations and implementation with a singly-linked list of Nodes including big-oh of operations

**Section 6.6. Priority Queue with Binary Heaps**

General concept of a priority queue: remove highest priority next

Priority Queue Operations: enqueue, dequeue, peek, size, isEmpty, __str__

Priority Queue Implementations: Python list unordered, Python list ordered by priority, Binary Heap including big-oh of operations

Binary Heap implementation: insert, findMin, delMin, isEmpty, size, buildHeap including big-oh of operations, and binary-tree diagrams after insert and delMin operations
Question 1. (4 points) Consider the following Python code.

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

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

\[ O(n \log n) \]

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

```python
for i in range(n):
    k = n
    while k > 1:
        print(k)
        k = k // 2
    for j in range(n):
        print(i, j)
```

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

\[ O(n^2 \log n) \]

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

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

def doSomething(n):
    for j in range(int(n/2)):
        doMore(j)

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

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

\[ O(n^5) \]

Question 4. (6 points) Suppose a $O(n^4)$ algorithm takes 10 seconds when $n = 1000$. How long would the algorithm run when $n = 10,000$?

\[
T(n) = c \cdot n^4
\]

\[
T(1000) = c \cdot 1000^4 = 10 \text{ sec}
\]

\[
c = \frac{10 \text{ sec}}{1000^4}
\]

\[
T(10000) = c \cdot 10000^4 = 10 \text{ sec}
\]

\[
c = \frac{10 \text{ sec}}{10000^4}
\]

\[
\frac{10000^4}{1016} \approx 10^{16} \text{ sec}
\]

\[
\frac{10 \text{ sec}}{1016} = 10^{5.5} \text{ sec}
\]

Question 5. (8 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?

Raise exception immediately where error occurs instead of later in code where the source of the error is harder to track down.
Question 6. A FIFO queue allows adding a new item at the front using an enqueue operation, and removing an item from the rear using a dequeue operation. One possible implementation of a queue would be to use a built-in Python list to store the queue 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

<table>
<thead>
<tr>
<th></th>
<th>isEmpty</th>
<th>enqueue(item)</th>
<th>dequeue</th>
<th>peek - returns front item without removing it</th>
<th><strong>str</strong></th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

a) (6 points) Complete the expected big-oh O(), for each Queue operation, assuming the above implementation. Let n be the number of items in the queue.

```python
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"
    if len(self._items) == 0:
        raise Exception("Cannot dequeue from empty queue.")
    return self._items.pop(0)
```

b) (8 points) Complete the method for the dequeue operation, including the precondition check to raise an exception if it is violated.

c) (8 points) Complete the method for the __str__ operation:
```python
def __str__(self):
    """Returns a string representation of items from front to rear."
    strResult = "(front) "
    for item in self._items:
        strResult += str(item) + ""
    return strResult + "(rear)"
```
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:

![Binary Heap Diagram]

Python List actually used to store heap items: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

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: 2*i + 1
   - its right child if it exists: 2*i + 2
   - its parent if it exists: \( \lfloor i/2 \rfloor \)

b) (7 points) What would the above heap look like after inserting 40 and then 20 (show the changes on above tree)

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

![Updated Heap Diagram]

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) Performing 20,000 inserts into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 \texttt{delMin} operations, it takes 0.39 seconds. Explain why 20,000 \texttt{delMin} operations take more time than the 20,000 insert operations?

Insert does one compare per level and \texttt{delMin} does two compares per level (compare two children to find min, then compare with min, child). Plus, inserted item rarely percolates to root, but \texttt{delMin} item moved to root likely percolated back down to being a leaf.
Question 8. The Node2Way and Node classes can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:

DoublyLinkedDeque Object

\[ \_\text{front:} \quad \_\text{rear:} \quad \_\text{size:} \]

\[ \begin{array}{cccccc}
\text{previous data} & \text{next} & \text{previous data} & \text{next} & \text{previous data} & \text{next} \\\
\_\text{front:} & 'a' & \text{next} & 'b' & \text{next} & 'c' \\\n\_\text{rear:} & 'd' & \text{next} & 'e' & \text{next} & 'f' \\\n\_\text{size:} & 5 & \text{next} & 3 & \text{next} & 4
\end{array} \]

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

<table>
<thead>
<tr>
<th>isEmpty</th>
<th>addRear</th>
<th>removeRear</th>
<th>addFront</th>
<th>removeFront</th>
<th>_str_</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

b) (16 points) Complete the addRear method for the above DoublyLinkedDeque implementation.

```python
class DoublyLinkedDeque(object):
    """ Double-linked list based deque implementation."""
    def __init__(self):
        self._size = 0
        self._front = None
        self._rear = None

    def addRear(self, newItem):
        """ Adds the newItem to the rear of the Deque."
        """ Precondition: none """
        temp = Node2Way(newItem)
        temp.setPrevious(self._rear)
        if self._size == 0:
            self._front = temp
        else:
            self._rear.setNext(temp)
        self._size += 1
```

c) (5 points) Why would using singly-linked nodes (i.e., only Node objects with data and next) to implement the Deque lead to poor performance (i.e., cause some Deque operations to have worse big-oh notations)? Justify your answer.

The removeRear method would be \( O(n) \) because resetting the \_rear pointer requires traversing from \_front to find next to last Node.