1. The Node class (in node.py) is used to dynamically create storage for a new item added to the stack. The LinkedStack class (in linked_stack.py) uses this Node class. Conceptually, a LinkedStack object would look like:

```
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
```

```
class LinkedStack(object):
    """ Link-based stack implementation. """
    def __init__(self):
        self.top = None
        self.size = 0
    def push(self, newItem):
        """Inserts newItem at top of stack."
        temp = Node(newItem)
        self.setNext(temp, self._top)
        self._top = temp
        self._size += 1
    def pop(self):
        """Removes and returns the item at top of the stack."
        Precondition: the stack is not empty.
        if self._size == 0:
            raise Exception("cannot pop from empty stack")
```

```
def peek(self):
    """Returns the item at top of the stack."
    Precondition: the stack is not empty.
    return self._top.getData()
```

```
def size(self):
    """Returns the number of items in the stack."
    return self._size
```

```
def isEmpty(self):
    return self._size == 0
```

```
def __str__(self):
    """Items strung from top to bottom."
    resultStr = "(top) "
    temp = self._top
    while temp != None:
        resultStr += str(temp.getData()) + "->"
        temp = temp.getNext()
    return resultStr + "(bottom)"
```

a) Complete the push, pop, and __str__ methods.

b) Stack methods big-o’s?
   (Assume "n" items in stack)
   - constructor __init__: \(O(1)\)
   - push(item): \(O(1)\)
   - pop(): \(O(1)\)
   - peek(): \(O(1)\)
   - size(): \(O(1)\)
   - isEmpty(): \(O(1)\)
   - __str__(): \(O(n)\)
Method pop

(1) Normal case picture

(2) update picture

(3) code for normal case

   1. temp = self._top
   2. self._top = temp.get_next()
   3. self._size -= 1
   4. return temp.get_data()

(4) special cases

   - empty stack => raise exception
   - removing list only item

   want

   done
A FIFO queue is basically what we think of as a waiting line.

The operations/methods on a queue object, say myQueue are:

<table>
<thead>
<tr>
<th>Method Call on myQueue object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myQueue.dequeue()</td>
<td>Removes and returns the front item in the queue.</td>
</tr>
<tr>
<td>myQueue.enqueue(item)</td>
<td>Adds item at the rear of the queue</td>
</tr>
<tr>
<td>myQueue.peek()</td>
<td>Returns the front item in the queue without removing it.</td>
</tr>
<tr>
<td>myQueue.isEmpty()</td>
<td>Returns True if the queue is empty, or False otherwise.</td>
</tr>
<tr>
<td>myQueue.size()</td>
<td>Returns the number of items currently in the queue</td>
</tr>
<tr>
<td>str(myQueue)</td>
<td>Returns the string representation of the queue</td>
</tr>
</tbody>
</table>

2. Complete the following table by indicating which of the queue operations should have preconditions. Write "none" if a precondition is not needed.

<table>
<thead>
<tr>
<th>Method Call on myQueue object</th>
<th>Precondition(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>myQueue.dequeue()</td>
<td>Queue is not empty</td>
</tr>
<tr>
<td>myQueue.enqueue(item)</td>
<td>none</td>
</tr>
<tr>
<td>myQueue.peek()</td>
<td>none</td>
</tr>
<tr>
<td>myQueue.isEmpty()</td>
<td>none</td>
</tr>
<tr>
<td>myQueue.size()</td>
<td>none</td>
</tr>
<tr>
<td>str(myQueue)</td>
<td>none</td>
</tr>
</tbody>
</table>

3. The textbook’s Queue implementation use a Python list:

```python
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def peek(self):
        if len(self.items) > 0:
            return self.items[-1]

    def size(self):
        return len(self.items)

    def __str__(self):
        resultStr = "(front)"
        for index in range(len(self.items)-1, -1, -1):
            resultStr += str(self.items[index])
        return resultStr + "(rear)"
```

a) Complete the _peek_ and _str_ methods.

b) What are the Queue methods big-oh’s? (Assume “n” items in the queue)

- constructor _init_: \( O(1) \)
- isEmpty(): \( O(1) \)
- enqueue(item): \( O(n) \)
- dequeue(): \( O(1) \)
- peek(): \( O(1) \)
- size(): \( O(1) \)
- str(): \( O(n) \)
3. An alternate queue implementation using a linked structure (LinkedQueue class) would look like:

![Diagram of LinkedQueue implementation]

a) Draw on the picture and number the steps for the enqueue method of the "normal" case (non-empty queue):

1. \( \text{temp} = \text{Node}(\text{new Item}) \)
2. if \( \text{self._size} == 0 \):
   - \( \text{self._front} = \text{temp} \)
3. \( \text{self._rear} = \text{temp} \)
4. \( \text{self._size} += 1 \)

b) Write the enqueue method code for the "normal" case:

\[
\begin{align*}
1. & \quad \text{temp} = \text{Node}(\text{new Item}) \\
2. & \quad \text{if } \text{self._size} == 0 : \\
3. & \quad \quad \text{self._front} = \text{temp} \\
4. & \quad \quad \text{self._rear} = \text{temp} \\
5. & \quad \text{self._size} += 1 \\
\end{align*}
\]

Starting with the empty queue below, draw the resulting picture after your "normal" case code executes.

![Diagram of empty LinkedQueue Object]

1. \( \text{temp} \)
2. \( \text{self._rear}.\text{setNext}(\text{temp}) \)

causes error "NoneType has no attribute setNext since self._rear is None and does not point at a None"

d) Fix your "normal" case code to handle the "special case" of an empty queue:

\[
\begin{align*}
1. & \quad \text{temp} = \text{Node}(\text{new Item}) \\
2. & \quad \text{if } \text{self._size} == 0 : \\
3. & \quad \quad \text{self._front} = \text{temp} \\
4. & \quad \text{else} \\
5. & \quad \quad \text{self._rear}.\text{setNext}(\text{temp}) \\
6. & \quad \text{self._rear} = \text{temp} \\
7. & \quad \text{self._size} += 1 \\
\end{align*}
\]