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:

```
<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>top</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>bottom</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>LinkedStack Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>_top:</td>
</tr>
<tr>
<td>_size:</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Node Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>next</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>next</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```

```

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."
    self._top = newItem
    self._size += 1

def pop(self):
    """Removes and returns the item at top of the stack.
    Precondition: the stack is not empty."
    return self._top.getData()

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.""
```

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

b) Stack methods big-oh’s?
   (Assume “n” items in stack)
   • constructor __init__:
   • push(item):
   • pop()
   • peek()
   • size()
   • isEmpty()
   • str()
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(myItem)</td>
<td>Adds myItem 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>none</td>
</tr>
<tr>
<td>myQueue.enqueue(myItem)</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):
        pass
    def size(self):
        return len(self.items)
    def __str__(self):
        pass
```

a) Complete the _peek, and __str__ methods

b) What are the Queue methods big-oh’s? (Assume “n” items in the queue)
   - constructor __init__:
   - isEmpty()
   - enqueue(item)
   - dequeue()
   - peek()
   - size()
   - str()
3. An alternate queue implementation using a linked structure (LinkedQueue class) would look like:

![Diagram of LinkedQueue Object and Node Objects]

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

b) Write the `enqueue` method code for the “normal” case:

```python
# Method to add an element to the queue
def enqueue(self, data):
    # Add code here
```

c) Starting with the empty queue below, draw the resulting picture after your “normal” case code executes.

![Diagram of empty LinkedQueue Object]

d) Fix your “normal” case code to handle the “special case” of an empty queue.
e) Draw on the above picture and number the steps for the deque method of the “normal” case (non-empty queue)

f) Write the deque method code for the “normal” case:

g) What “special case(s)” does the deque method code need to handle?

h) Draw the picture for each special case and number the steps for the deque method in the “special” case(s)

i) Combine the “normal” and special case(s) code for a complete deque method.

j) Complete the big-oh notation for the LinkedQueue methods: ("n" is the # items)

<table>
<thead>
<tr>
<th></th>
<th><strong>init</strong></th>
<th>enqueue(item)</th>
<th>dequeue()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th><strong>str</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-oh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>