1. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. The `LinkedListStack` class (in `linked_stack.py`) uses this `Node` class. Conceptually, a `LinkedListStack` 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 LinkedListStack(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)
        temp.setNext(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 ValueError("Cannot pop empty stack")
        temp = self._top
        self._top = temp.getNext()
        return temp.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."""
        strResult = ""
        temp = self._top
        for count in range(self._size):
            strResult += str(temp.getData()) + """"
            temp = temp.getNext()
        return strResult
```

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

b) Stack methods big-oh’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__()` : \( O(n) \)
Process for writing linked data structure method:
(1) Draw "normal" case picture (e.g. several items already)
(2) Modify picture to reflex changes of the method
(3) Number the steps to order the changes in step (2)
(4) Write normal case code
(5) Consider "special" cases (e.g. stack is empty)
Push: - Run normal case code on special case picture

- empty stack:
  - self
  - size: 0

  - top
  - data
  - next

  0 temp
Normal case code:

```python
    temp = self._top
    temp = temp.get_next()
    self._top = self._top.get_next()
    self._size -= 1
    return temp.get_data()
```

Special cases:

- Popping last item
  ```python
  self
  ```
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>Queue is not empty</td>
</tr>
<tr>
<td>myQueue.enqueue(myItem)</td>
<td>none</td>
</tr>
<tr>
<td>myQueue.peek()</td>
<td>Queue is not empty</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(0)
    def peek(self):
        return self.items[-1]
    def size(self):
        return len(self.items)
    def __str__(self):
        strResult = ""
        for index in range(len(self.items)-1,-1,-1):
            strResult += str(self.items[index]) + ""
        return strResult
```

a) Complete the __peek__, and __str__ methods

b) What are the Queue methods big-oh’s?
(Provide an O(n) time or O(1) time estimate.)

- **constructor __init__:** Strings are immutable
- **isEmpty()**: O(1)
- **enqueue(item)**: O(1)
- **dequeue()**: O(n)
- **peek()**: O(1)
- **size()**: O(1)
- **str()**: O(n²)
3. An alternate queue implementation using a linked structure (LinkedQueue class) would look like:

"Abstract Queue"

front: 'w'
rear: 'y'
_size: 3

LinkedQueue Object

data: 'w'
_next: -
_front: -
_size: 0
_rear: -

data: 'x'
_next: -
_front: -
_size: 1
_rear: -

data: 'y'
_next: -
_front: -
_size: 2
_rear: -

Node Objects

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

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

```python
    temp = Node(item)
    self._rear.setNext(temp)
    self._rear = temp
    self._size += 1
```

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

empty LinkedQueue Object

_front: -
_size: 0
_rear: -

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