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:

```
"Abstract"
Stack

   a  b  c  d  
  / \     \   |
bottom \     \  top

LinkedStack Object

   data  next

   \      /  \      /
     0  1

Node Objects

   data  next
   \      /
    a    b
```

```python
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._top.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."""
    temp = self._top
    self._top = temp.getNext()
    self._size -= 1
    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 stored from top to bottom."""
    resultStr = "(top)"
    current = self._top
    while current != None:
        resultStr += str(current.getData()) + "\n"
        current = current.getNext()
    resultStr += "(bottom)"
    return resultStr
```

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():  \(O(n)\)
Implement (linked) data structure method

1. Draw normal-case picture (non-empty)
2. Update picture for method being implemented
3. Number the steps in picture
4. Write code for normal-case
   1. `temp = Node (newItem)`
   2. `temp.setNext (self._top)`
   3. `self._top = temp`
   4. `self._size += 1`
5. Consider special cases - empty stack
6. Run/trace normal case code
7. Fix for special case if necessary
Normal case:
1. temp = self._top
2. self._top = temp, get_next()
3. self._size -= 1
4. return temp.get_data()

Special cases:
(1) Precondition check if stack empty
(2) Stack with single node

Return
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>queue is not empty</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.append(item)

    def dequeue(self):
        return self.items.pop(0)

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

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

    def __str__(self):
        front = "[" + str(self.items[0]) + ""
        for index in range(1, len(self.items) - 1):
            front += str(self.items[index]) + "," + str(self.items[index])
        return front + "]
```

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:

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:

```
if self._size == 0:  # Special case: empty queue
    self._front = self._rear = None
    self._size = 0
    return
```

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

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