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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>top</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>bottom</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 new item 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 = self._top.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 strung from top to bottom. """
        resultStr = "(top)"
        temp = self._top
        while temp != None:
            resultStr += str(temp.getData()) + " "
            temp = temp.getNext()
        resultStr += "(bottom)"
        return resultStr
```
Linked Data Structure - new method push

1. Draw "normal" case picture
   a. non empty

2. Update picture for method
3. Number steps in picture
4. Write normal case code

5. Identify special cases
   a. drew picture
   b. "trace" normal case code to see if/where it breaks
(4) Normal case picture

1. \( \text{self\_top} = \text{self\_top} \cdot \text{getNext()} \)

2. \( \text{self\_size} = 1 \)

3. \( \text{temp} = \text{self\_top} \)

4. \( \text{return temp \cdot \text{getData()} } \)

(5) Special cases
- empty
- pop last item in stack
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><code>myQueue.dequeue()</code></td>
<td>Removes and returns the front item in the queue.</td>
</tr>
<tr>
<td><code>myQueue.enqueue(myItem)</code></td>
<td>Adds <code>myItem</code> at the rear of the queue</td>
</tr>
<tr>
<td><code>myQueue.peek()</code></td>
<td>Returns the front item in the queue without removing it.</td>
</tr>
<tr>
<td><code>myQueue.isEmpty()</code></td>
<td>Returns True if the queue is empty, or False otherwise.</td>
</tr>
<tr>
<td><code>myQueue.size()</code></td>
<td>Returns the number of items currently in the queue</td>
</tr>
<tr>
<td><code>str(myQueue)</code></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><code>myQueue.dequeue()</code></td>
<td>queue is not empty</td>
</tr>
<tr>
<td><code>myQueue.enqueue(myItem)</code></td>
<td></td>
</tr>
<tr>
<td><code>myQueue.peek()</code></td>
<td></td>
</tr>
<tr>
<td><code>myQueue.isEmpty()</code></td>
<td></td>
</tr>
<tr>
<td><code>myQueue.size()</code></td>
<td></td>
</tr>
<tr>
<td><code>str(myQueue)</code></td>
<td></td>
</tr>
</tbody>
</table>

3. The textbook's Queue implementation uses 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):
        returnStr = "(rear)"
        for item in self.items:
            returnStr += str(item)
        returnStr = "(front)" + returnStr
        returnStr = returnStr + returnStr
```

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() \( O(n) \)
   - str() \( O(n) \)