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

| d | top |
| c |     |
| b |     |
| a | bottom |
```

```
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 IndexError("cannot pop empty stack")
        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 strung from top to bottom.""
        strResult = "(top)"
        temp = self._top
        while temp != None:
            strResult += str(temp.getData()) + "\n"
            temp = temp.getNext()
        strResult += "(bottom)"
        return strResult
```
Steps for writing "linked" methods

1. Draw normal case picture
2. Draw the updated picture after method runs
3. Number steps in drawing
4. Write normal-case code from picture

5. Consider special cases:
   a. empty stack ("run" normal case code)
   b. draw picture
      self
      -top
      -size
      0
      1
      2
      3
      4
      "top" modifies code to handle special case if needed
Pop — remove and return the top item

Draw normal case

1. temp = self._top
2. self._top = temp.getNext()
3. self._size = self._size - 1
4. return temp.getData()

Special cases:
A) empty stack — precondition: stack is not empty
B) stack with one item
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()
    def peek(self):
        return self.items[0]
    def size(self):
        return len(self.items)
    def __str__(self):
        return str(self.items)
```

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:

1. `temp = Node(newItem)`
2. `self._rear.setNext(temp)`
3. `self._rear = temp`
4. `self._size += 1`

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.

1. `temp = Node(newItem)`
2. `if self._size == 0:`  # checks for special case
   
3. `self._front = temp`
   
4. `else:`
   
5. `self._rear.setNext(temp)`
6. `self._rear = temp`
7. `self._size += 1`