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

```

```

c
d

Node Objects

```

```

Node

```

```

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)
    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."""
    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 stringed from top to bottom.""
    return str()
```

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():
   ```
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>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lazy Stack Object

1. **pop picture:**

```
LinkedStack Object

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_top</td>
<td>c</td>
</tr>
<tr>
<td>_size</td>
<td>2</td>
</tr>
</tbody>
</table>

Node Objects

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>'c'</td>
</tr>
<tr>
<td>next</td>
<td></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 Exception("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)"
        current = self._top
        while current != None:
            resultStr += str(current.getData()) + ',
            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()`: $O(n)$
Implementing Linked method

1. Draw picture of "normal case" (some items already)
2. Numbered step to change
3. Wrote normal case code
4. Consider special case(s): empty stack

```
Empty stack
```

Pop special case(s)

1. Empty stack - precondition check
2. Popping 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>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):
        raise ... if self.isEmpty()
        return self.items[-1]

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

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

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) \(O(1)\)
- 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:

```
"Abstract Queue"    
| 'w' | 'x' | 'y' |   
|-----|-----|-----|---
| front |       | rear |   

LinkedQueue Object

_data: Node
__front: Node
__size: int
__rear: Node

Node Objects

_data: Node
__next: Node

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

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(new Item)`
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.

```
empty LinkedQueue Object

__front: Node
__size: 0
__rear: Node

1) temp
```

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

```
1. `temp = Node(new Item)`
   if self._size == 0:  # empty queue special case
   2' `self._front = temp`
   else:
   3. `self._rear.setNext(temp)`  # normal case
5. `self._rear = temp`
4. `self._size += 1`
```
e) Draw on the above picture and number the steps for the `dequeue` method of the "normal" case (non-empty queue)

f) Write the `dequeue` method code for the "normal" case:

```
g) What "special case(s)" does the `dequeue` method code need to handle?

h) Draw the picture for each special case and number the steps for the `dequeue` method in the "special" case(s)

i) Combine the "normal" and special case(s) code for a complete `dequeue` method.
```

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

<table>
<thead>
<tr>
<th></th>
<th><em>init</em></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>