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

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._top == None:
            raise ValueError('')

        self._top = self._top.getNext()
        self._size -= 1

    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."

        current = self._top
        while current != None:
            resultStr += str(current.getData())
            current = current.getNext()

        return resultStr + '(bottom)'
```

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(): $O(n)$
Linked Data Structures

Normal case - non empty

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

Identity special cases (empty data stack, single item stack,)

1. Draw picture of special case
2. "Run" normal case code to see where if "breaks"
3. Modify code to handle special case with if-then-else statement

Empty stack

- Stack
- Top
- Size
- Work but not needed

1. Temp
Normal case

1. `temp = self._top`
2. `self._top = temp.next()`
3. `self._size -= 1`
4. `return temp.data()`

Special cases

(A) Empty Stack - precondition "Stack is not empty"

if `self._size <= 0`:
    raise ValueError("Cannot pop empty stack")

(B) Pop only 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 CANNOT be empty</td>
</tr>
<tr>
<td>myQueue.enqueue(myItem)</td>
<td>none</td>
</tr>
<tr>
<td>myQueue.peek()</td>
<td>n &lt;= i &lt;= l</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[len(self.items)-1]
    
    def size(self):
        return len(self.items)
    
    def __str__(self):
        resultStr = 'front | ' + ' | '.join([str(x) for x in self.items]) + ' | 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): \(O(n)\)
- dequeue(): \(O(1)\)
- peek(): \(O(1)\)
- size(): \(O(1)\)
- str(): \(O(n)\)