1. An “abstract” view of the stack:

Using an array implementation would look something like:

<table>
<thead>
<tr>
<th>Items:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top:</th>
<th></th>
<th></th>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>max:</td>
<td></td>
<td></td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

Complete the big-oh notation for the following stack methods assuming an array implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(item)</td>
<td>O(1)</td>
</tr>
<tr>
<td>pop()</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1)</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isFull()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

2. Since Python does not have a (directly accessible) built-in array, we can use a list.

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

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

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

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

Since Python uses an array of references (pointers) to list items in their implementation of a list.

"Abstract" Stack Object

```
myStack = Stack()

myStack.push('a')
myStack.push('b')
myStack.push('c')
```

list Object

```
0 1 2 3
a b c d
```

a) Complete the big-oh notation for the stack methods assuming this Python list implementation: ("n" is the # items)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(item)</td>
<td>O(1)</td>
</tr>
<tr>
<td>pop()</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1)</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isFull()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

b) Which operations should have what preconditions?

- pop - stack is not empty
- peek - stack is not empty
3. The text's alternative stack implementation also using a Python list is:

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

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

    def peek(self):
        return self.items[0]

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

Since an array is used to implement a Python list, the alternate Stack implementation using a list:

<table>
<thead>
<tr>
<th>&quot;Abstract&quot; Stack</th>
<th>&quot;alternate&quot; Stack Object</th>
<th>list Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>items:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c b a</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>e b a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>f c a</td>
</tr>
</tbody>
</table>

a) Complete the big-oh notation for the "alternate" Stack methods: ("n" is the # items)

<table>
<thead>
<tr>
<th></th>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>size()</th>
<th>isEmpty()</th>
<th><strong>init</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-oh</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

4. How could we use a stack to check if a word is a palindrome (e.g., radar, toot)?

5. How could we check to see if we have a balanced string of nested symbols? ("((())")
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:

![Diagram of LinkedStack object with Node objects]

```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)
        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 strung from top to bottom."""
```

a) Complete the `push`, `pop`, and `__str__` methods.

b) Stack big-oh's?
   (Assume "n" items in stack)
   - constructor __init__:
   - push(item):
   - pop():
   - peek():
   - __size__:
   - isEmpty():
   - __str__():
Process for writing linked data structure:
(1) Draw "normal" case picture (e.g., several items already)
(2) Modify picture to reflect changes of the method
(3) Number the steps to order the changes in step (2)
(4) Write normal case code.