1. An "abstract" view of the stack:

Using an array implementation would look something like:

```
0 1 2 3 (max-1)
items: a b c [ ]
top: [ ]
max: [00]
```

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

```
<table>
<thead>
<tr>
<th>d</th>
<th>c</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
</table>
  top
  bottom
```

Stack Object

```
items: [ ]
```

List Object

```
0 1 2 3 ( )
a b c [ ]
```

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>
</tbody>
</table>

b) Which operations should have what preconditions?

\[
\text{peek} - \text{stack is not empty} \\
\text{pop} - \text{stack is not empty}
\]
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>Abstract Stack</th>
<th>&quot;alternate&quot; Stack Object</th>
<th>List Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>items:</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>a b a</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>pop()</td>
<td>$O(n)$</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><strong>init</strong></td>
<td>$X$</td>
</tr>
</tbody>
</table>

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

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:

```
<table>
<thead>
<tr>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>b</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 newItem at top of stack."""
def pop(self):
    """Removes and returns the item at top of the stack.
    Precondition: the stack is not empty."""
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 methods big-oh's?
   (Assume "n" items in stack)

   - constructor `__init__`
   - push(item):
   - pop()
   - peek()
   - size()
   - isEmpty()
   - str()
Implement (linked) data structure method

1. Draw normal-case picture
   (non-empty)

2. Update picture for method being implemented

3. Number the steps in picture

4. Write code for normal-case

   1. `temp = Node(newItem)`
   2. `temp.next = self._top`
   3. `self._top = temp`
   4. `self._size += 1`