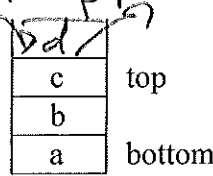
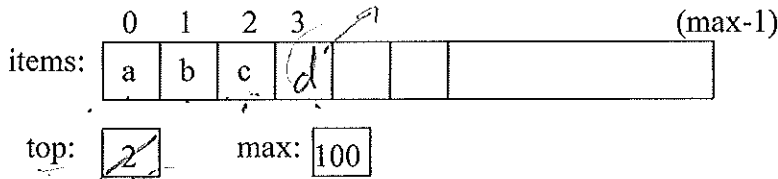


1. An "abstract" view of the stack:



Using an array implementation would look something like:



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

	push(item)	pop()	peek()	size()	isEmpty()	isFull()	Constructor
Big-oh	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	

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

```

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()

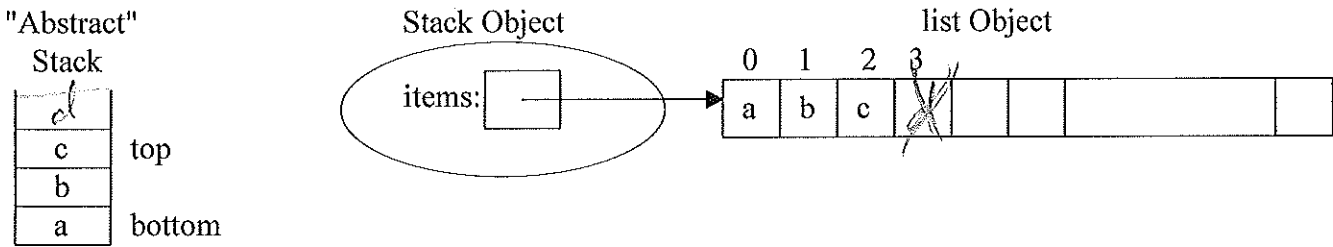
    def peek(self):
        return self.items[len(self.items)-1]

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

*myStack = Stack()*  
*myStack* → Stack Object (items: [ ])  
 List

*len(self.items) == 0*

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



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

	push(item)	pop()	peek()	size()	isEmpty()	__init__
Big-oh	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

b) Which operations should have what preconditions?

*pop and peek "the stack cannot be empty"*

3. The text's alternative stack implementation also using a Python list is:

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

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

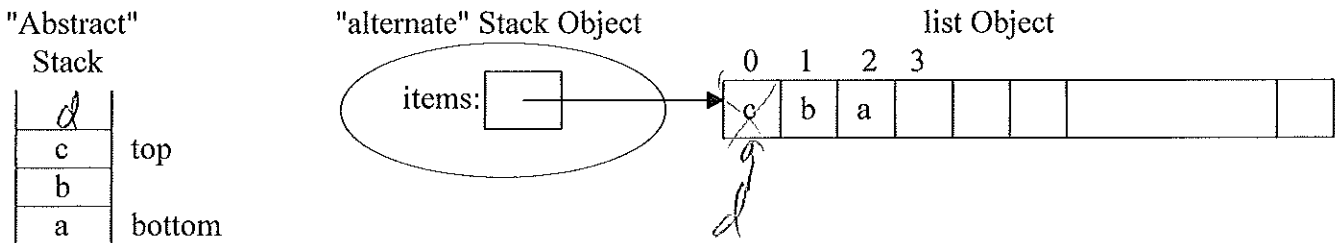
    def push(self, item):
        self.items.insert(0, item)

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

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

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

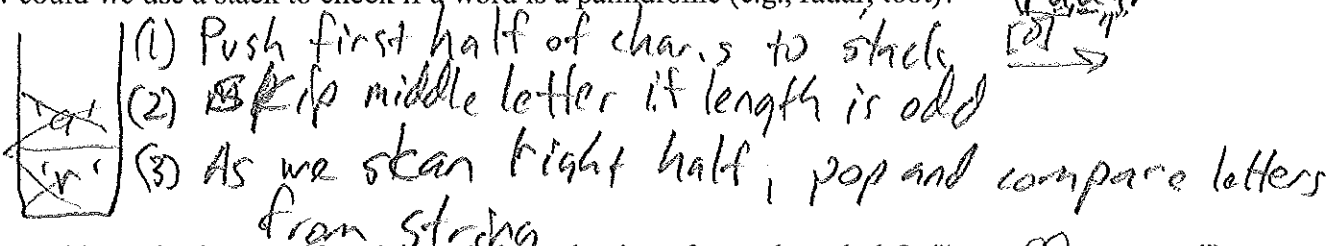
Since an array is used to implement a Python list, the alternate Stack implementation using a list:



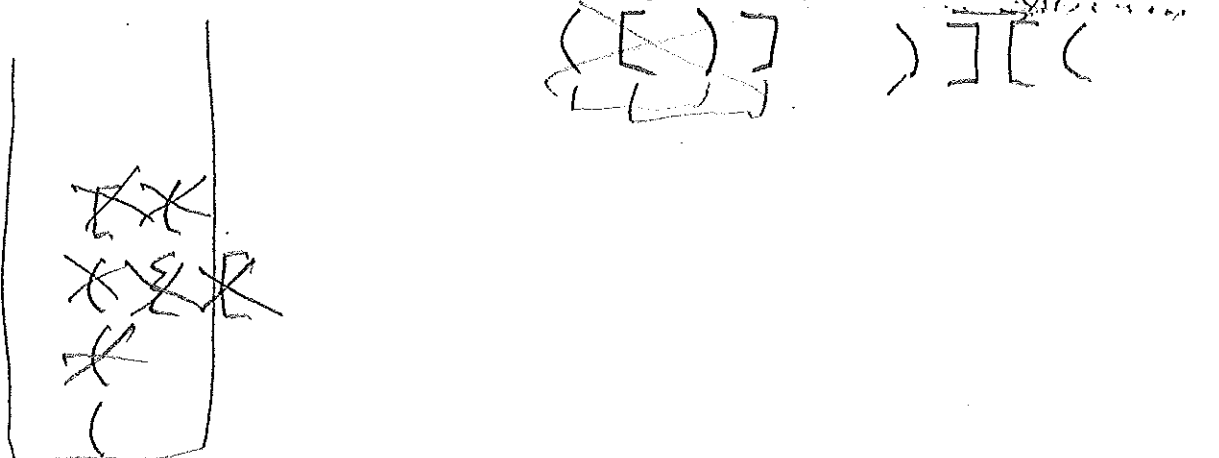
a) Complete the big-oh notation for the "alternate" Stack methods: ("n" is the # items)

	push(item)	pop()	peek()	size()	isEmpty()	__init__
Big-oh	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

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

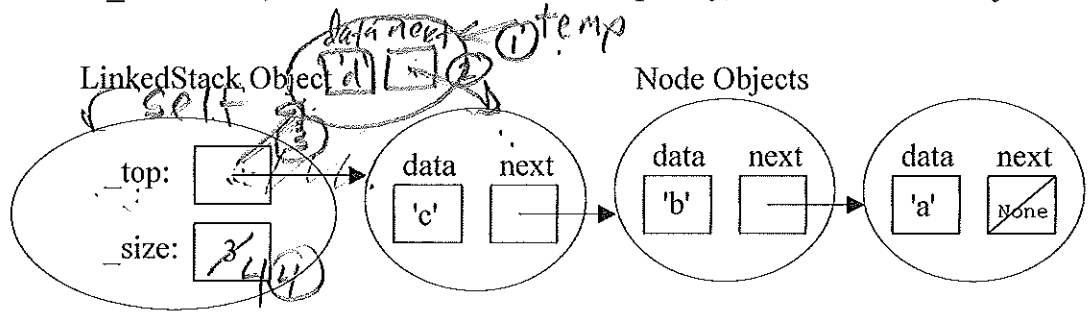
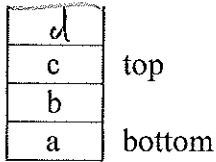


5. How could we check to see if we have a balanced string of nested symbols? ("((( [ ] ) { } [ ] )")



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:

"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 newItem at top of stack. """
        ① temp = Node(newItem)
        ② temp.setNext(self._top)
        ③ self._top = temp
        ④ self._size += 1

    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__`:  $O(1)$
- `push(item)`:  $O(1)$
- `pop()`
- `peek()`  $O(1)$
- `size()`  $O(1)$
- `isEmpty()`  $O(1)$
- `str()`