5. Arrays in most HLLs are static in size (i.e., cannot grow at run-time), so arrays are constructed to hold the "maximum" number of items. For example, an array with 1,000 slots might only contain 3 items:

size: ~~3~~ 4

	0	1	2	3	4			999
scores:	20	10	30	25				

a) The *physical size* of the array is the number of slots in the array. What is the physical size of scores? 1000

b) The *logical size* of the array is the number of items actually in the array. What is the logical size of scores? 3

c) The *load factor* is faction of the array being used. What is the load factor of scores? 3/1000

d) What is the $O(\,)$ for "appending" a new score to the "right end" of the array? $O(1)$

e) What is the $O(\,)$ for adding a new score to the "left end" of the array? $O(n)$ $n \equiv$ logical size

f) What is the *average* $O(\,)$ for adding a new score to the array? $O(\frac{n}{2}) = O(n)$ $T(n) = c\frac{n}{2} = dn$

g) During run-time if an array fills up and we want to add another item, the program can usually:

- Create a bigger array than the one that filled up
- Copy all the items from the old array to the bigger array
- Add the new item
- Delete the smaller array to free up its memory

When creating the bigger array, how much bigger than the old array should it be?

double size over the old array

h) What is the $O(\,)$ of moving to a larger array? $O(n)$

6. Consider the following list methods in Python: $n \equiv$ len(myList) $m \equiv$ len(otherList)

Method	Usage	Average $O(\,)$ for myList containing n items
index []	`itemValue = myList[i]`	$O(1)$
	`myList[i] = newValue`	$O(1)$
append	`myList.append(item)`	$O(1)$ ($O(n)$ worst-case))
extend	`myList.extend(otherList)`	$O(m)$
insert	`myList.insert(i, item)`	$O(n)$
pop	`myList.pop( )`	$O(1)$
pop(i)	`myList.pop(i)`	$O(n)$
del	`del myList[i]`	$O(n)$
remove	`myList.remove(item)`	$O(n)$
index	`myList.index(item)`	$O(n)$
iteration	`for item in myList:`	$O(n)$
reverse	`myList.reverse( )`	$O(n)$

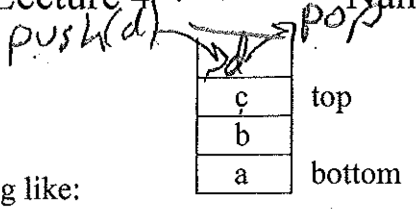
Dictionary Operations:

Method	Usage	Explanation	Average $O(\,)$ for n keys
get item	`myDictionary.get(myKey)` `value = myDictionary[myKey]`	Returns the value associated with myKey; otherwise *None*	$O(1)$
set item	`myDictionary[myKey]=value`	Change or add myKey:value pair	$O(1)$
in	`myKey in myDictionary`	Returns True if myKey is in myDictionary; otherwise False	$O(1)$
del	`del myDictionary[myKey]`	Deletes the mykey:value pair	$O(1)$

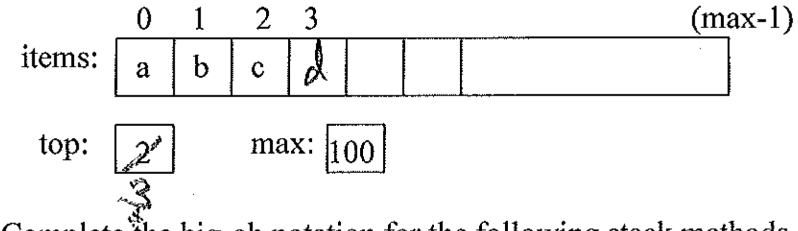
Data Structures (CS 1520) Lecture 4 Name:____________

"LIFO"

1. An "abstract" view of the stack:



Using an array implementation would look something like:

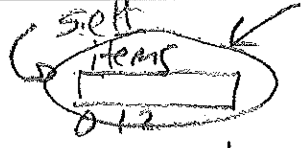


Complete the big-oh notation for the following stack methods assuming an array implementation: ("n" is the # items)

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

2. Since Python does not have a (directly accessable) 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)
```



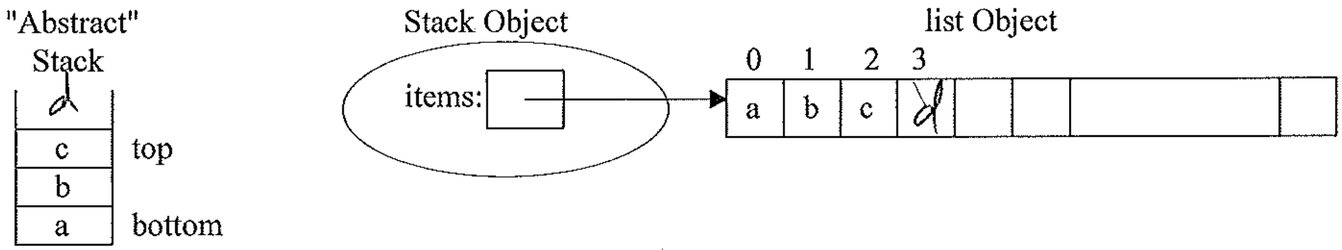
```
return len(self.item)==0
```

-? -1

```
myStack = Stack()

if myStack.isEmpty():
```

```
if self.isEmpty():
    raise(ValueError, "Cannot pop/peek from empty stack")
```

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) ?

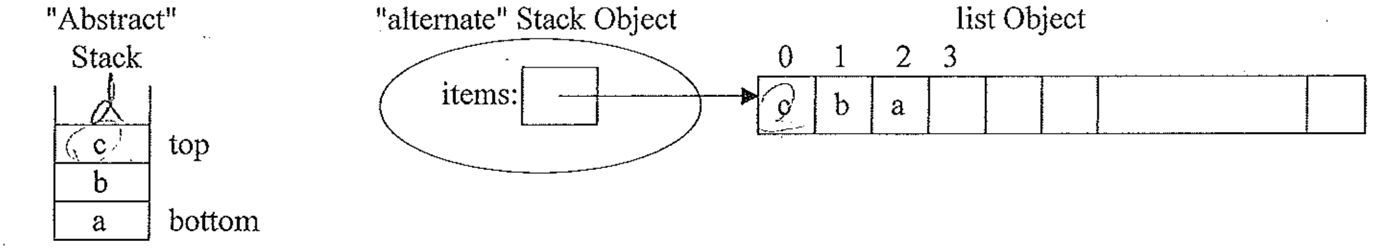
b) Which operations should have what preconditions?

Peek, Pop - stack is not 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)
```

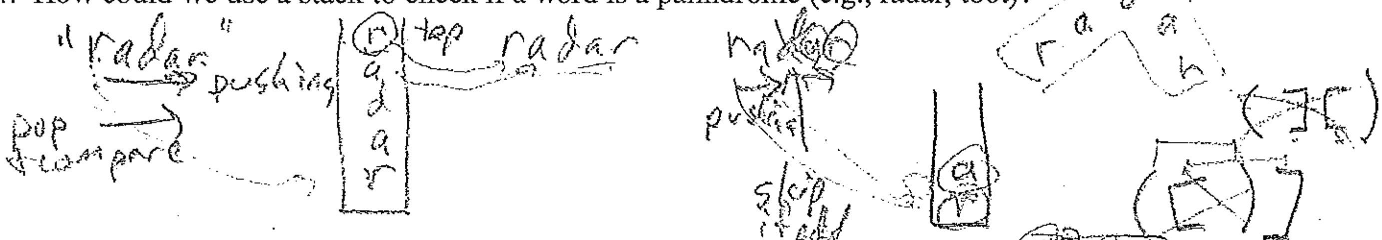
Since an array is used to implement a Python list, the alternate Stack implementation using a list:



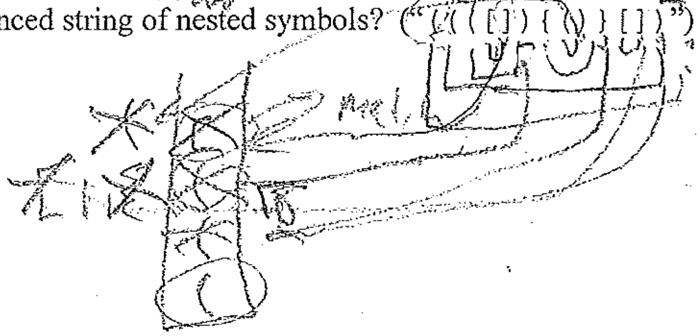
a) Complete the big-oh notation for the "alternate" Stack methods: ("n" is the # items)

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

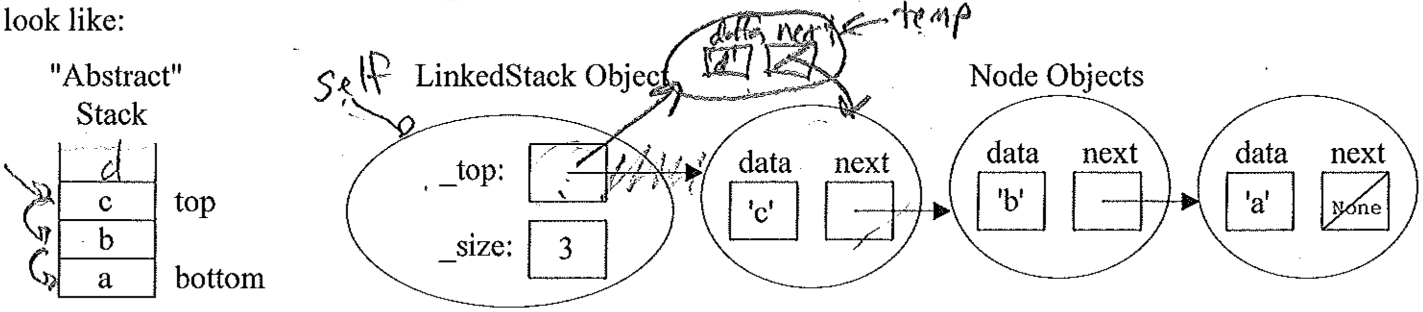
4. How could we use a stack to check if a word is a palindrome (e.g., radar, toot)?



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:



```
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
```

"assessor methods"

mutator method

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()`

```
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)

    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."""
```