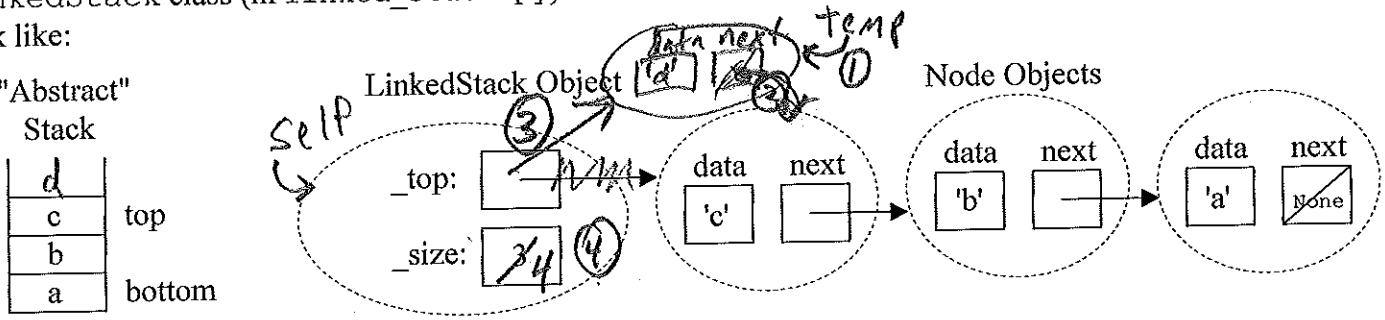
Data Structures (CS 1520) Lecture 5 Name:_____

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
```

accessor methods

mutators methods

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)

```
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."""
        if self._size == 0:
            raise IndexError("cannot pop empty stack")
        temp = self._top
        self._top = temp.getNext()
        self._size -= 1
        return temp.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."""
        strResult = "(top) "
        temp = self._top
        while temp != None:
            strResult += str(temp.getData()) + " "
            temp = temp.getNext()
        strResult += "(bottom)"
        return strResult
```

Steps for writing "linked" methods

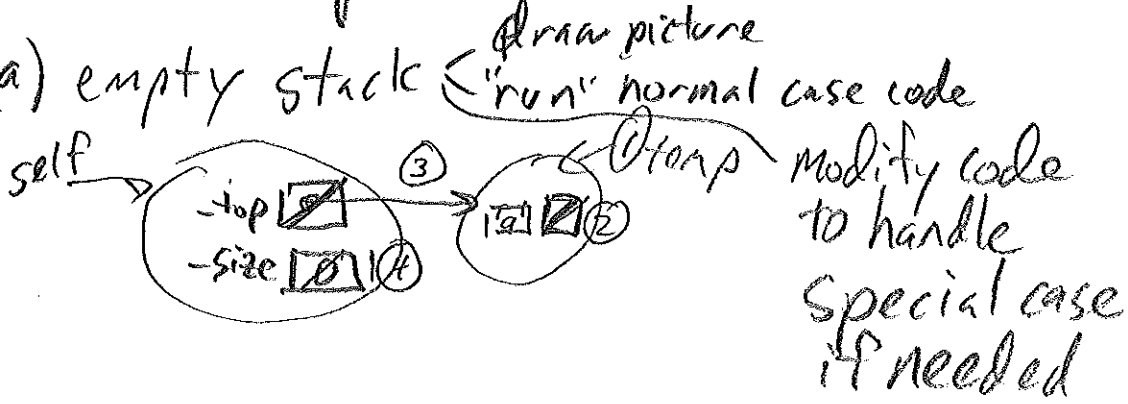
(1) Draw normal case picture (non empty)

(2) Draw the updated picture after method runs

(3) Number steps in drawing

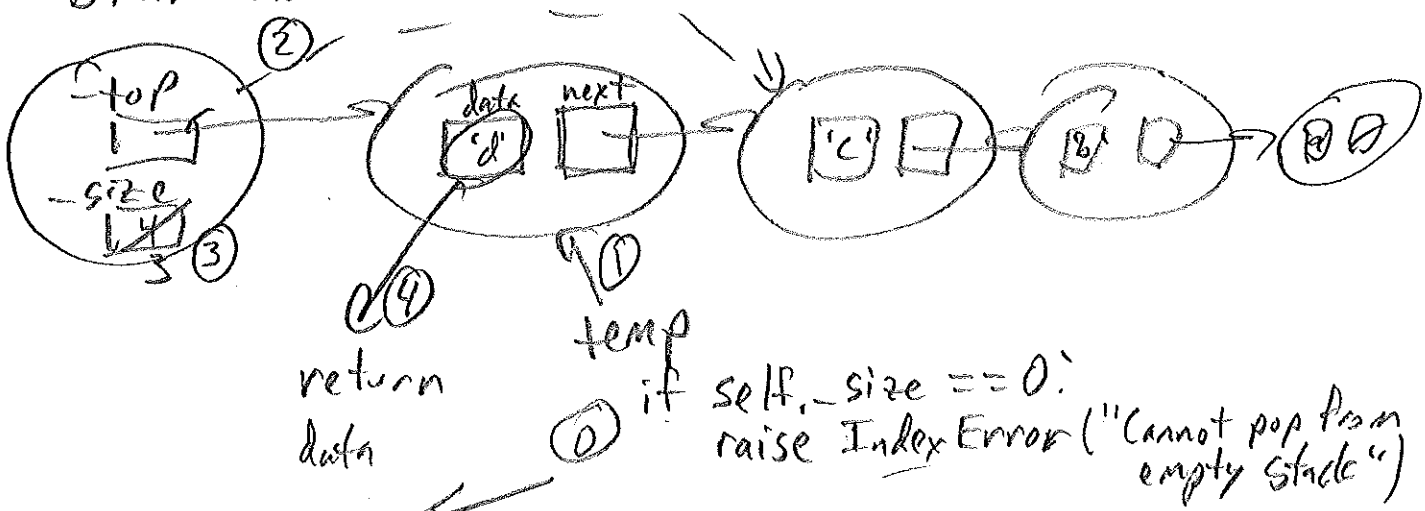
(4) write normal-case code from picture

(5) Consider "special cases"

(a) empty stack 

Pop – remove and return the top item

Draw normal case



① temp = self._top

② self._top = temp.getNext()

③ self._size = self._size - 1

④ return temp.getData()

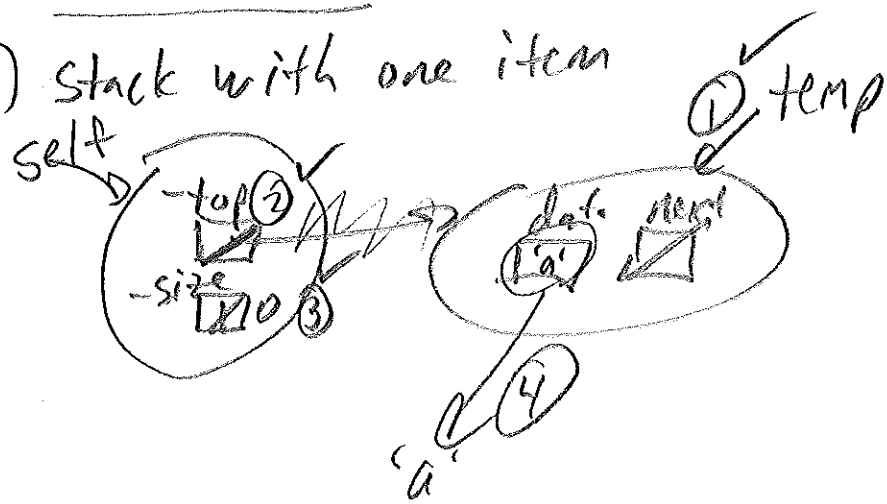
---

special cases:

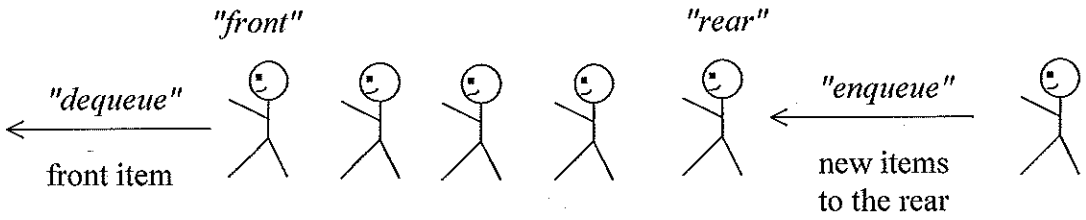
A) empty stack – precondition ~~Can't pop~~ stack is not empty

---

B) Stack with one item



A FIFO *queue* is basically what we think of as a waiting line.



The operations/methods on a queue object, say myQueue are:

Method Call on myQueue object	Description
`myQueue.dequeue()`	Removes and returns the front item in the queue.
`myQueue.enqueue(myItem)`	Adds `myItem` at the rear of the queue
`myQueue.peek()`	Returns the front item in the queue without removing it.
`myQueue.isEmpty()`	Returns `True` if the queue is empty, or `False` otherwise.
`myQueue.size()`	Returns the number of items currently in the queue
`str(myQueue)`	Returns the string representation of the queue

2. Complete the following table by indicating which of the queue operations should have preconditions. Write "none" if a precondition is not needed.

Method Call on myQueue object	Precondition(s)
`myQueue.dequeue()`	Queue is not empty
`myQueue.enqueue(myItem)`	None
`myQueue.peek()`	Queue is not empty
`myQueue.isEmpty()`	none
`myQueue.size()`	"
`str(myQueue)`	"

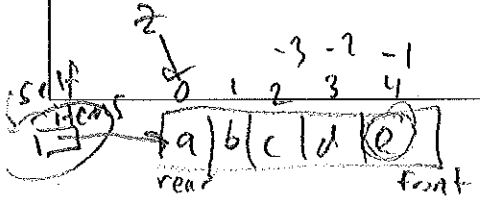
3. The textbook's Queue implementation use a Python list:

```
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):
```

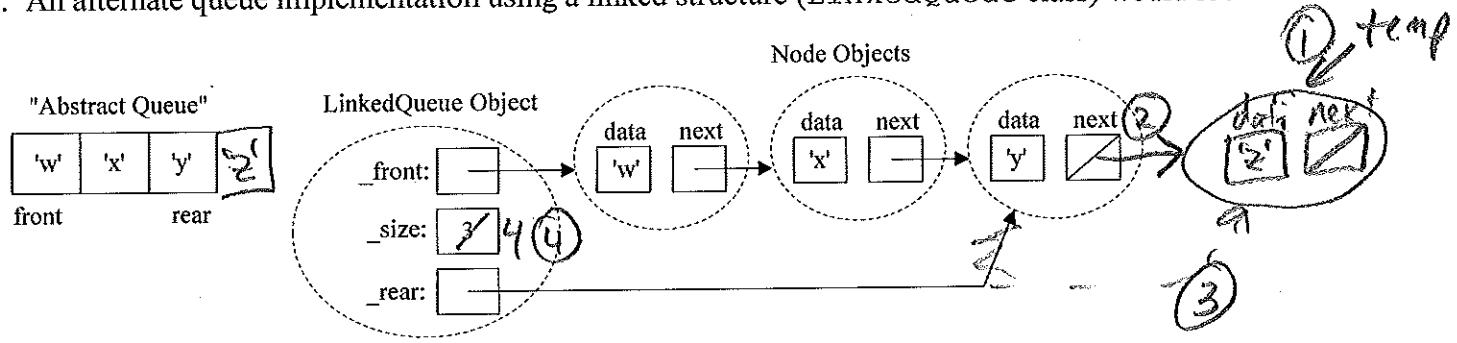
a) Complete the `_peek`, and `__str__` methods

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)



3. An alternate queue implementation using a linked structure (`LinkedQueue` class) would look like:

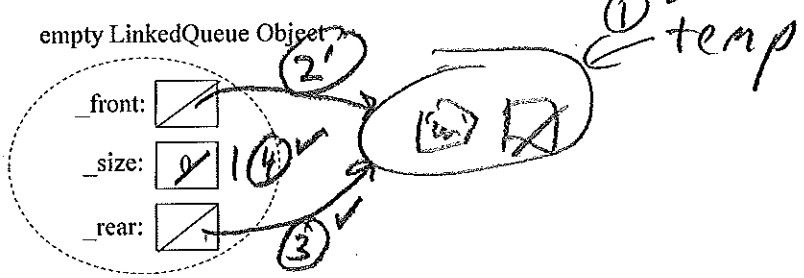


a) Draw on the picture and number the steps for the `enqueue` method of the "normal" case (non-empty queue)

b) Write the `enqueue` method code for the "normal" case:

```
(1) temp = Node(newItem)
(2) self._rear.setNext(temp)
(3) self._rear = temp
(4) self._size += 1
```

c) Starting with the empty queue below, draw the resulting picture after your "normal" case code executes.



(2) step causes error since _rear has value None
We want alternate step (2') which sets _front to first node

d) Fix your "normal" case code to handle the "special case" of an empty queue.

```
(1)  temp = Node(newItem)
     if self._size == 0:    # checks for special case
(2')     self._front = temp
     else:
(2)      self._rear.setNext(temp)
(3)  self._rear = temp
(4)  self._size += 1
```