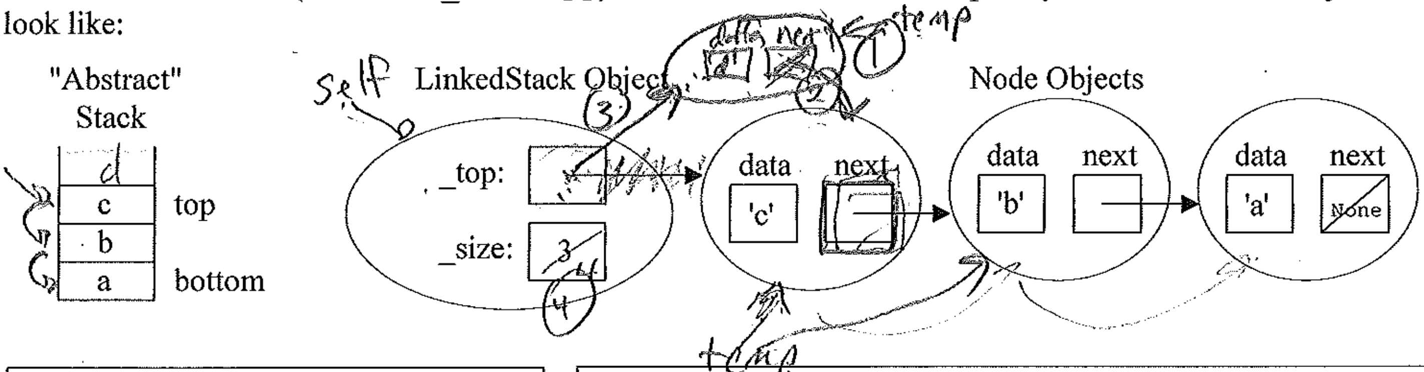
1. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. The `LinkedStack` class (in `linked_stack.py`) uses this Node class. Conceptually, a `LinkedStack` object would look like:




```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):          "assessor methods"
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):  mutator method
        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."""
        if self._size == 0:
            raise ValueError("Cannot pop empty stack")
        temp = self._top
        self._top = self._top.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):    "(top) c b a (bottom)"
        """Items strung from top to bottom."""
        resultStr = "(top) "
        temp = self._top
        while temp != None:
            resultStr += str(temp.getData()) + " "
            temp = temp.getNext()
        resultStr += "(bottom)"
        return resultStr
```

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

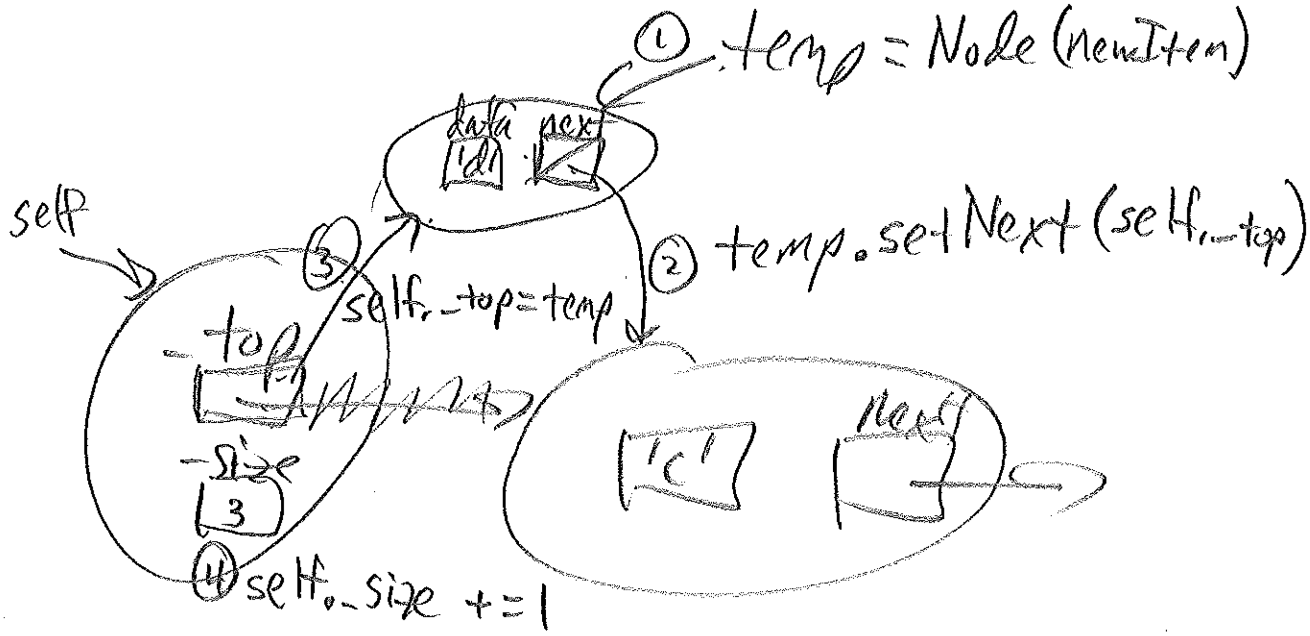
b) Stack methods big-oh's? (Assume "n" items in stack)

- constructor `__init__`: O(1)
- `push(item)`: O(1)
- `pop()` O(1)
- `peek()` O(1)
- `size()` O(1)
- `isEmpty()` O(1)
- `str()` O(n)

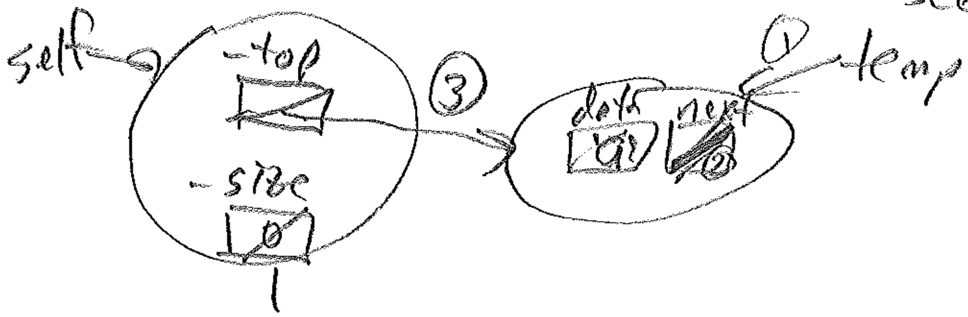
Linked Data Structure - new method push

(1) Draw "normal" case picture
^
non empty

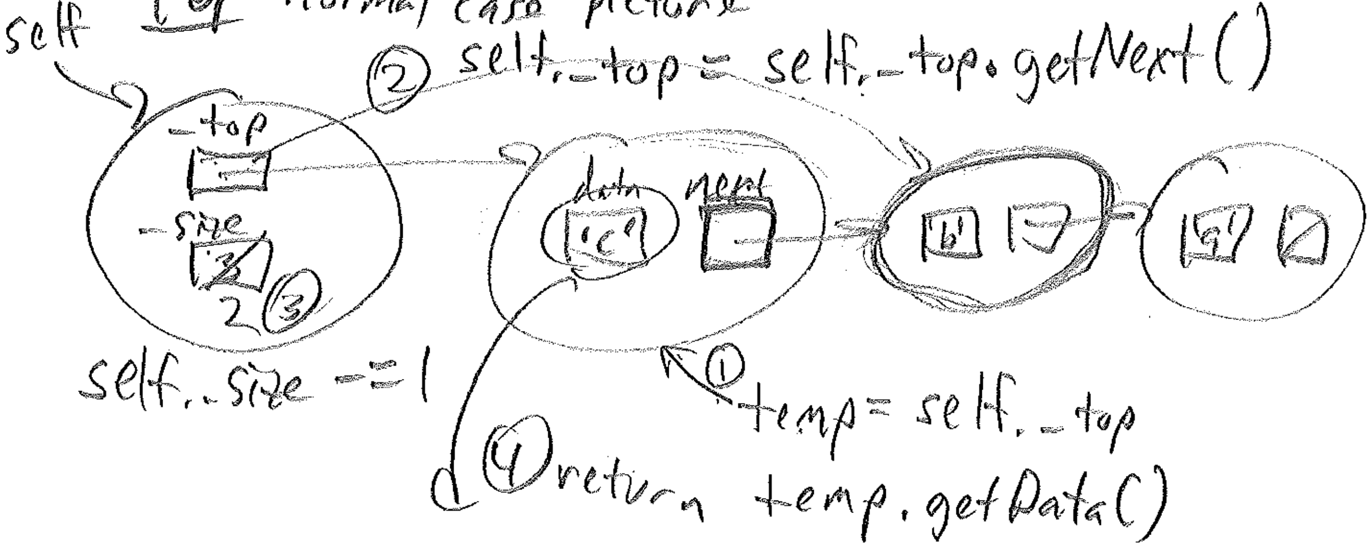
(2) Update picture for method
(3) Number steps in picture
(4) Write normal case code



(5) Identify Special case(s)
empty (a) draw picture (b) "trace" normal case code to see if/where it breaks

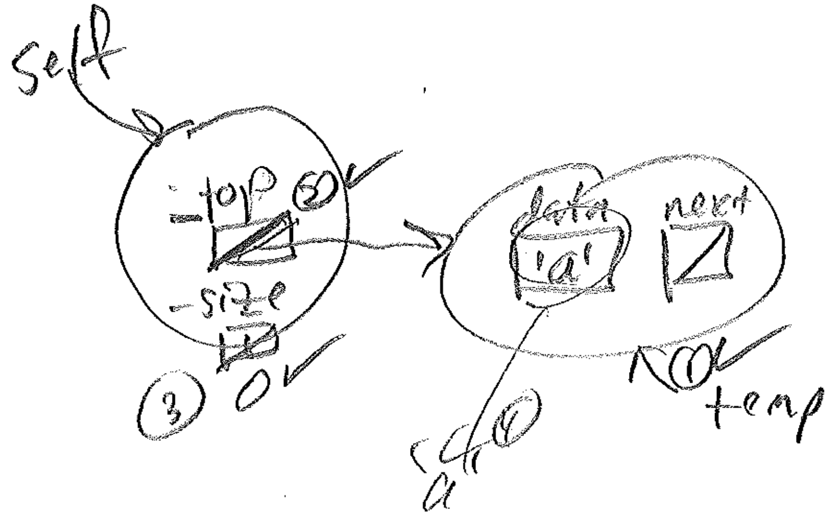


(4) pop normal case picture



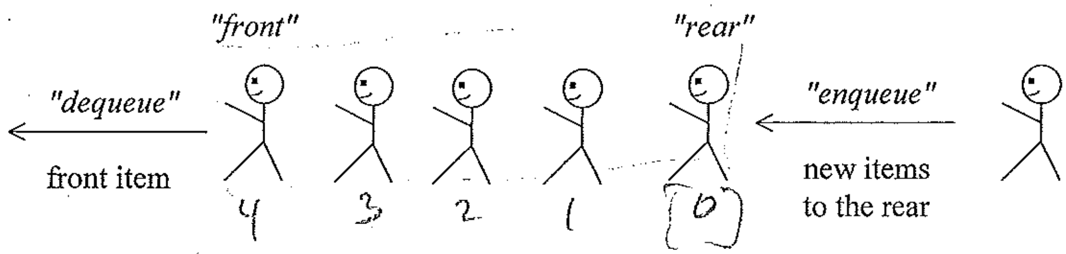
a   a + 1

(5) Special cases
- empty ✓
- pop last item in stack



Data Structures (CS 1520) Lecture 5 Name:____________

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)`	
`myQueue.peek()`	" " " "
`myQueue.isEmpty()`	
`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[-1]

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

    def __str__(self):
      resultStr = " (rear)"
      for item in self.items:
         resultStr = " "+str(item) + resultStr
      resultStr = "(front)" + resultStr
      return resultStr
```

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)