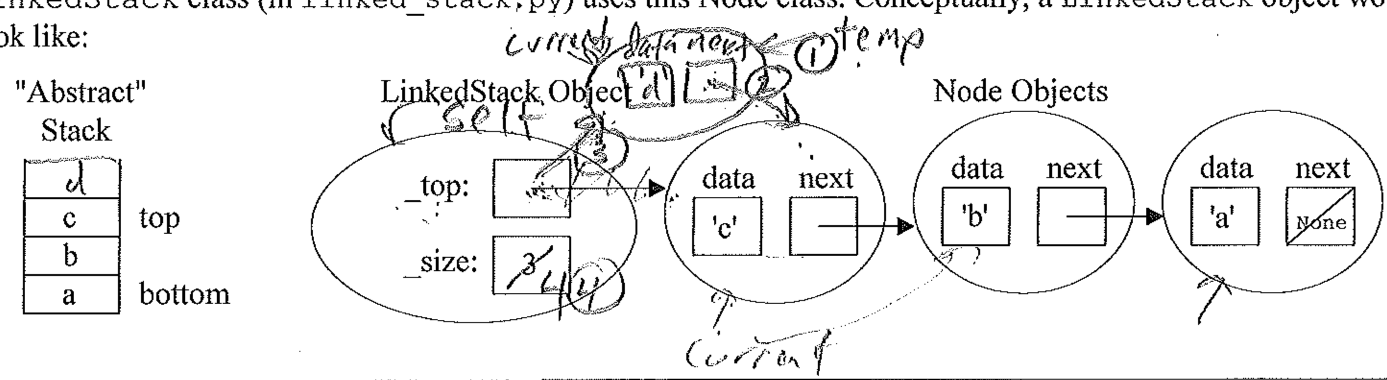
Data Structures (CS 1520) Lecture 5 Name:________________

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
```

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."""
 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._top == 0:
            raise Value("                ")
 ①      temp = self._top
 ②      self._top = temp.getNext()
 ③      self._size -= 1
 ④      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) d c b a (bottom)'
        """Items strung from top to bottom."""
        resultStr = '(top) '
        current = self._top
        while current != None:
            resultStr += str(current.getData()) + ' '
            current = current.getNext()
        return resultStr + '(bottom)'
```

Linked Data Structures

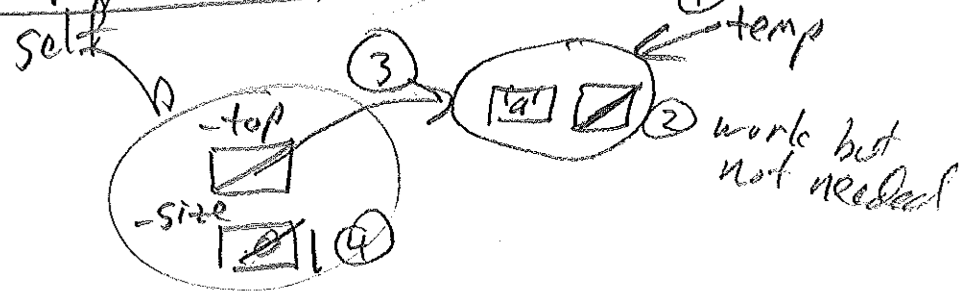
Normal Case - non empty

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

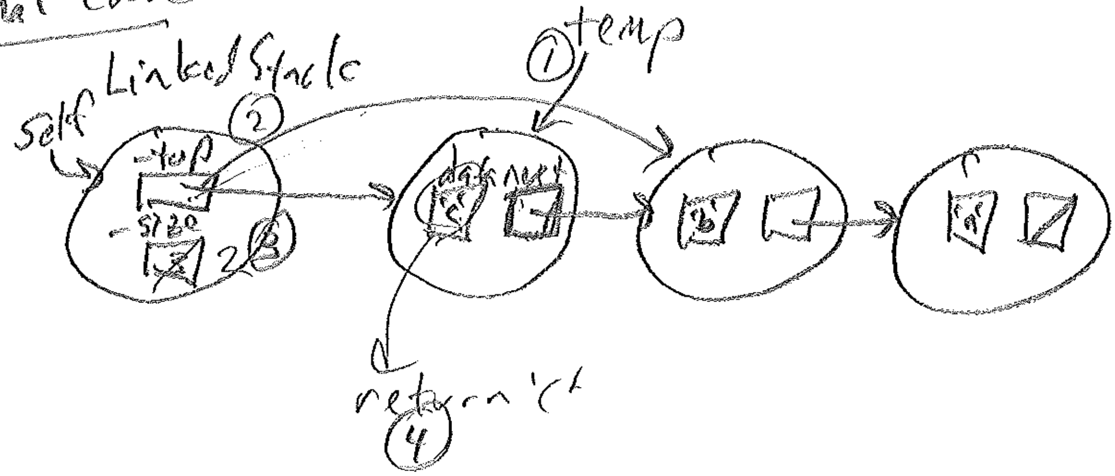
Identify special cases (empty data stack, single item stack,

(1) Draw picture of special case
(2) "Run" normal case code to see where it "breaks"
(3) Modify code to handle special case with if-statement

Empty stack



Pop Normal case



(1) temp = self._top

(2) self._top = temp.getNext()

(3) self._size -= 1

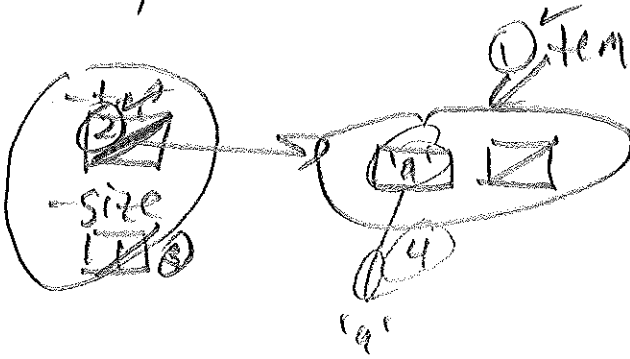
(4) return temp.getData()

special cases

(A) Empty Stack - precondition "Stack is not empty"

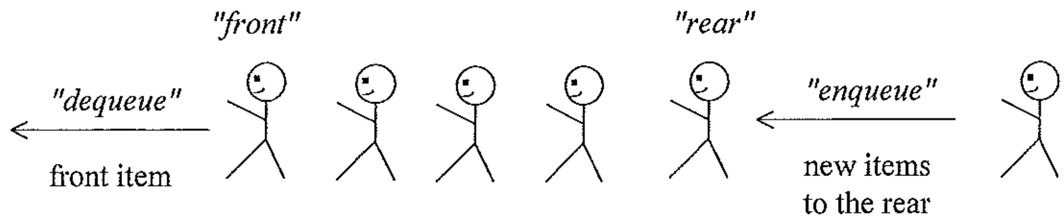
```
if self._size == 0:
    raise ValueError("Cannot pop empty stack")
```

(B) Pop only item in stack



Data Structures (CS 1520) Lecture 5 Name:________________

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



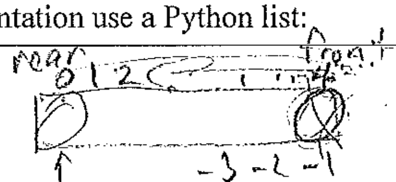
The operations/methods on a queue object, say myQueue are:

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 cannot be empty
`myQueue.enqueue(myItem)`	none
`myQueue.peek()`	" " " "
`myQueue.isEmpty()`	none
`myQueue.size()`	none
`str(myQueue)`	none

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):
        if self.isEmpty():
            raise ValueError('...')
        return self.items.pop()

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

    def size(self):
        return len(self.items)
    '(front) 1 2 3 ... n (rear)'
    def __str__(self):
        resultStr = '(front) '
        for index in range(len(self.items)-1, -1, -1):
            resultStr += str(self.items[index]) + ' '
        resultStr += '(rear)'
        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)