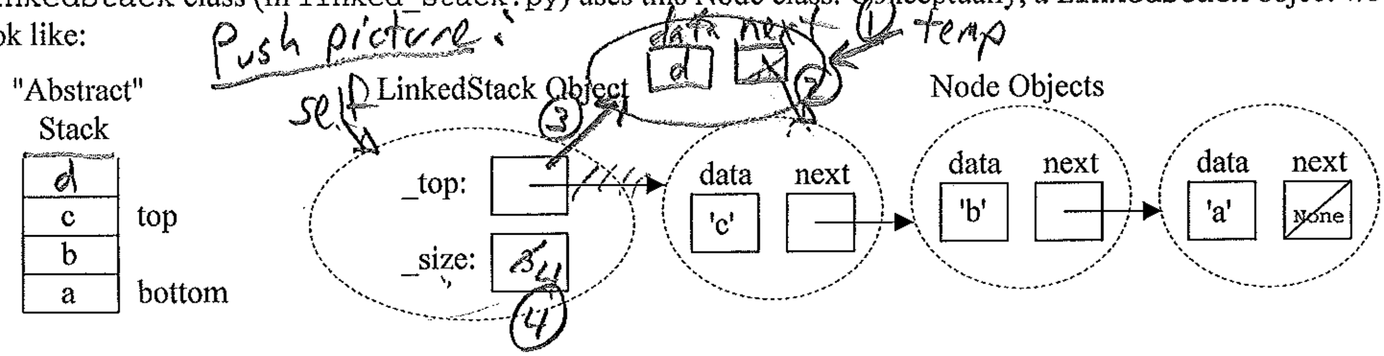
1. The Node class (in node.py) is used to dynamically create storage for a new item added to the stack. The LinkedStack class (in linked_stack.py) uses this Node class. Conceptually, a LinkedStack object would look like:




```
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
```

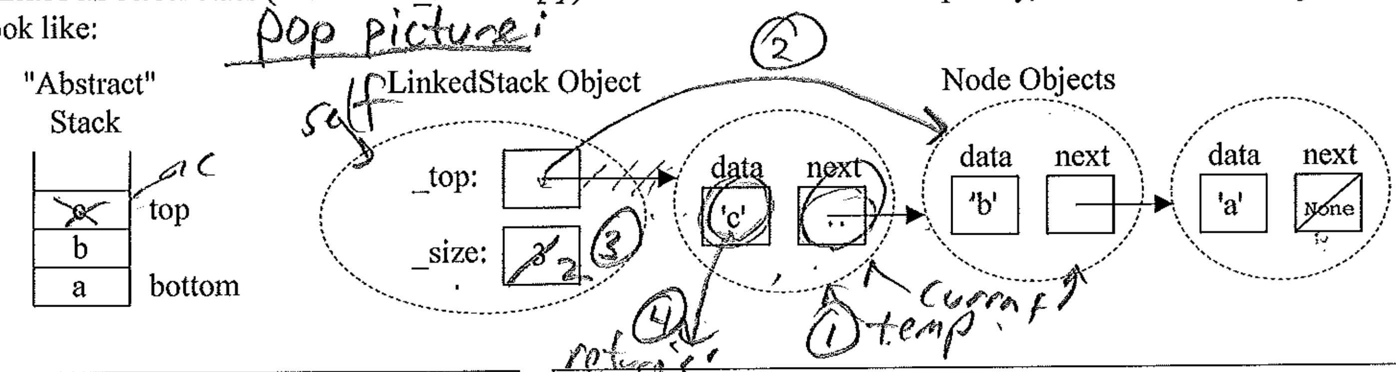
```
class LinkedStack(object):
    """ Link-based stack implementation."""

    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__:
- push(item):
- pop()
- peek()
- size()
- isEmpty()
- str()

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: pop picture:




```
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."""
        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 Exception("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) "
        current = self._top
        while current != None:
            resultStr += str(current.getData()) + " "
            current = current.getNext()
        resultStr += "(bottom)"
        return resultStr
```

Implementing Linked method

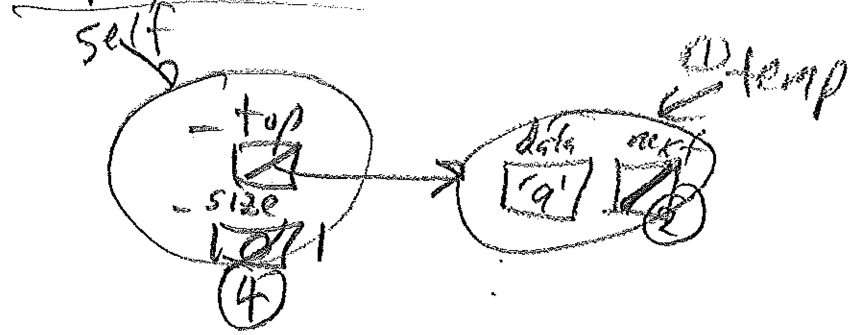
(1) Draw picture of "normal case"
(some items already)

(2) Numbered step to change

(3) wrote normal case code

(4) Consider special case(s): empty stack

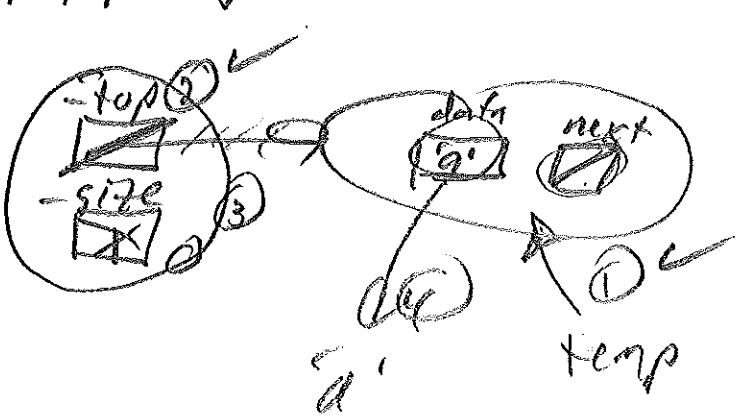
Empty stack



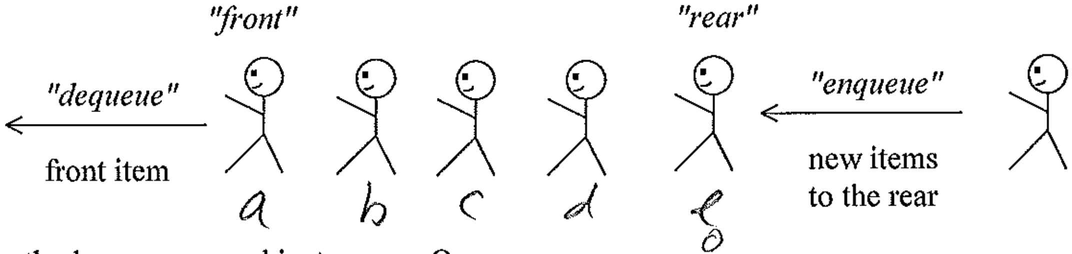
Pop special case(s)

(1) empty stack ✓ precond. check

(2) popping last item in stack



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



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()`	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 ...
        return self.items.pop()

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

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

    def __str__(self):          # (front) a b c ... e (rear)
        resultStr = "(front) "
        for index in range(len(self.items)-1,-1,-1):
            resultStr += str(self.items[index]) + " "
        resultStr += "(rear)"
```

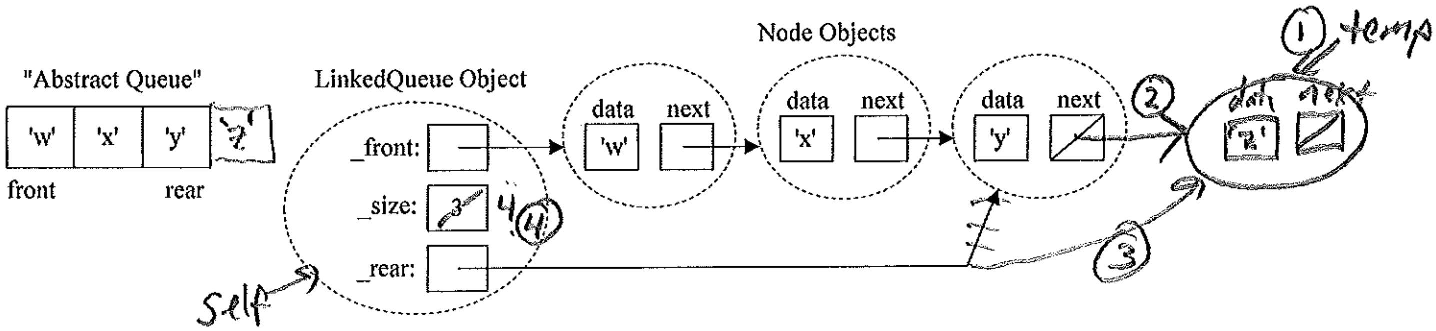
a) Complete the _peek, and __str__ methods

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)

$1+2+3+\dots+(n-1)+n = \frac{n}{2}*(n+1)$

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

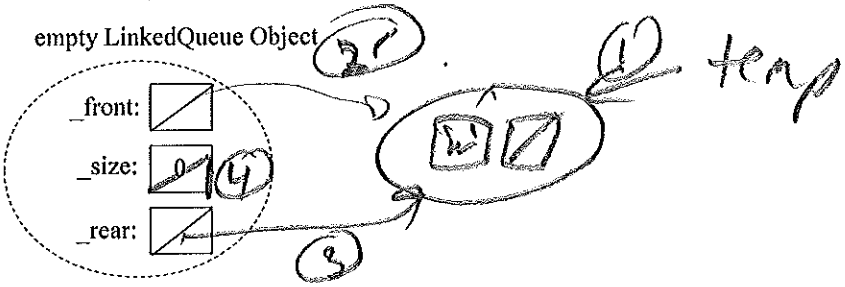


a) Draw on the picture and number the steps for the enqueue method of the "normal" case (non-empty queue)

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

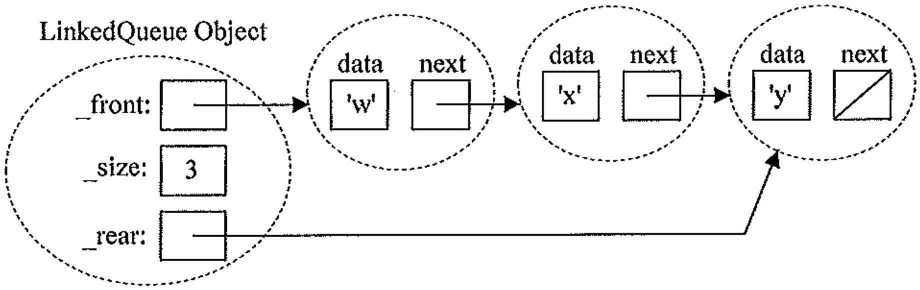
```
(1) temp = Node(newItem)
(2) self._rear.setNext(temp)
(3) self._rear = temp
(4) self._size += 1
```

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



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

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



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

f) Write the `dequeue` method code for the "normal" case:

g) What "special case(s)" does the `dequeue` method code need to handle?

h) Draw the picture for each special case and number the steps for the `dequeue` method in the "special" case(s)

i) Combine the "normal" and special case(s) code for a complete `dequeue` method.

j) Complete the big-oh notation for the `LinkedQueue` methods: ("n" is the # items)

	__init__	enqueue(item)	dequeue( )	peek( )	size( )	isEmpty( )	__str__
Big-oh							