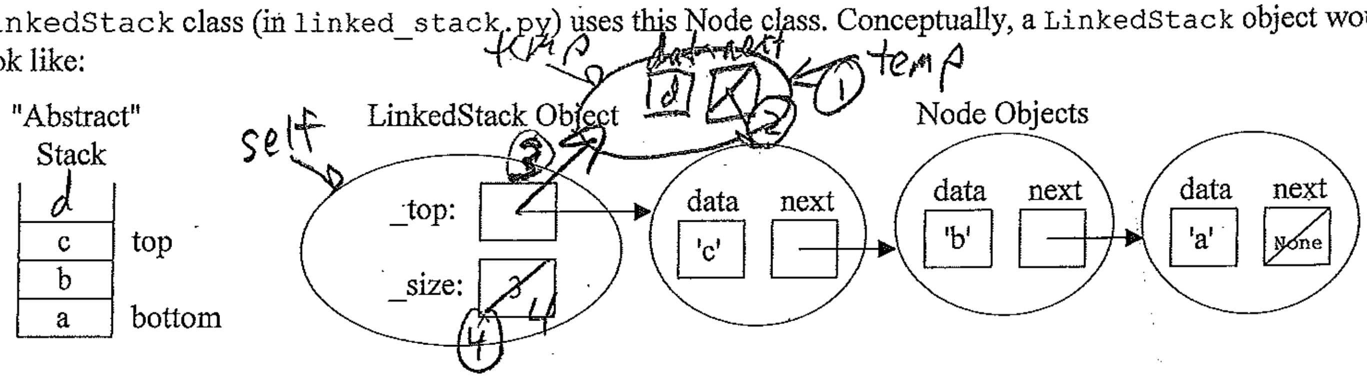
1. The `Node` class (in `node.py`) is used to dynamically create storage for a new item added to the stack. The `LinkedStack` class (in `linked_stack.py`) uses this Node class. Conceptually, a `LinkedStack` object would look like:




```
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."""
        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 ValueError("Cannot pop empty stack")
normal temp = self._top
case    self._top = temp.getNext()
code    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 = ""
        temp = self._top
        for count in range(self._size):
            strResult += str(temp.getData()) + " "
            temp = temp.getNext()
        return strResult
```

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)

Process for writing linked data structure method:

(1) Draw "normal" case picture (e.g. "several items already)

(2) Modify picture to reflex changes of the method

(3) Number the steps to order the changes in step (2)

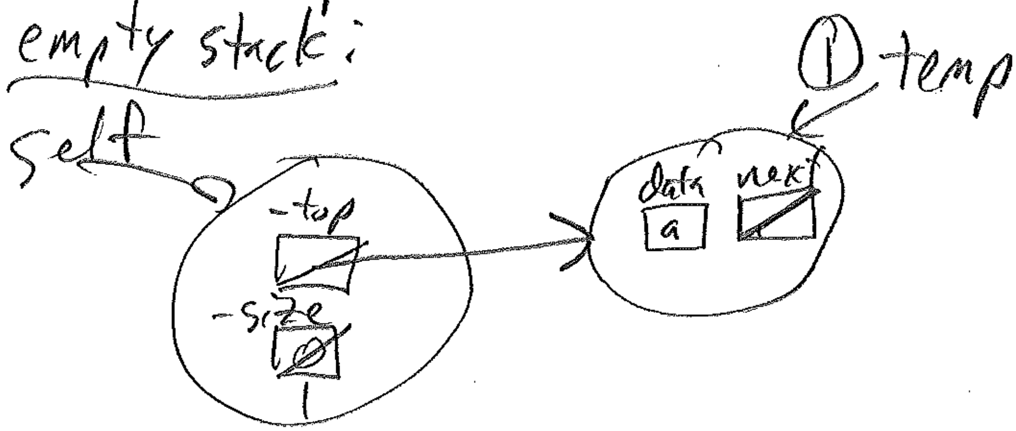
(4) Write normal case code

---

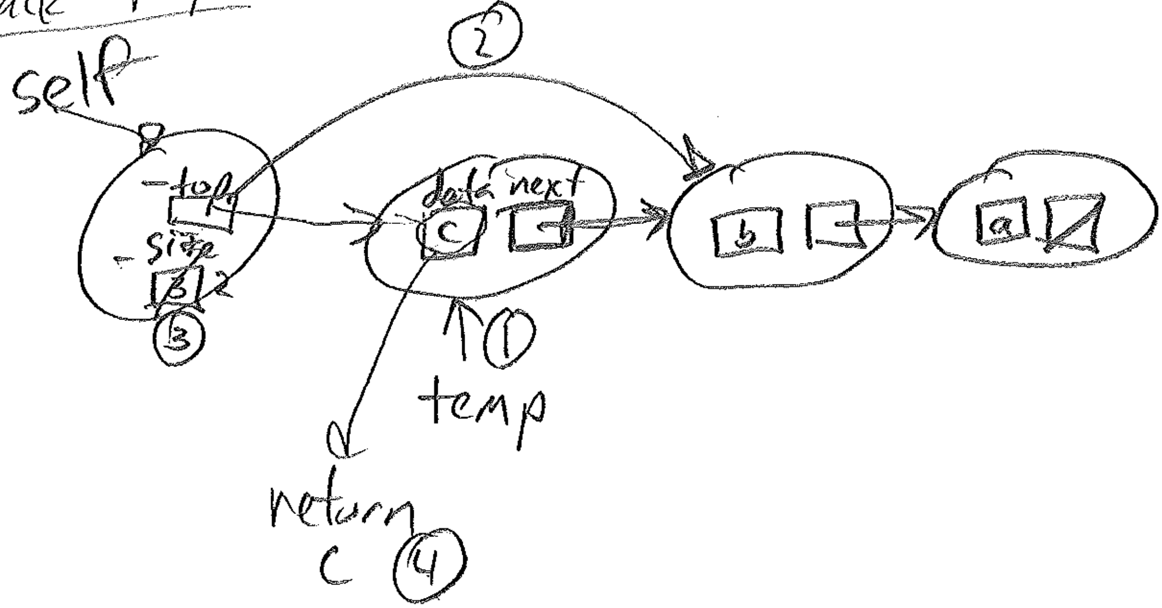
(5) Consider "special" cases (e.g. stack is empty)

Push: - Run normal case code on special case picture

empty stack:



Stack Pop

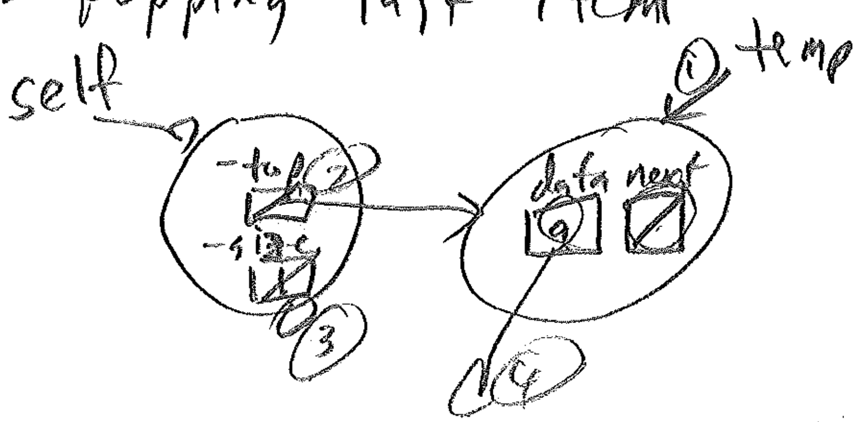


Normal case code

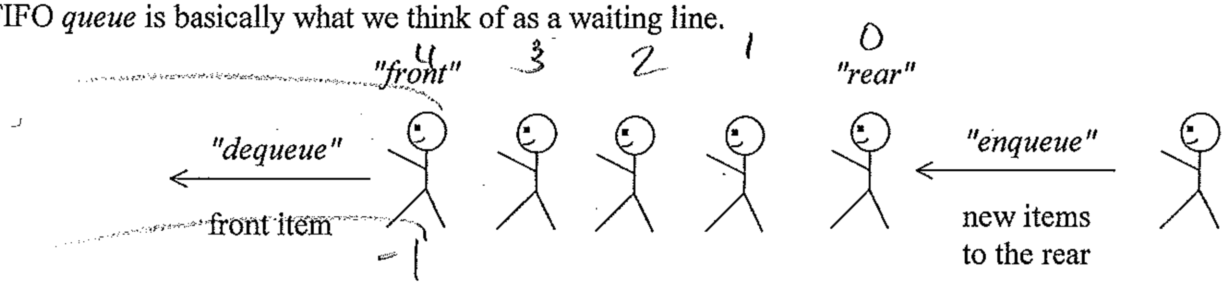
```
temp = self._top
          temp.getNext()
self._top = self._top.getNext()
self._size -= 1
return temp.getData()
```

Special cases

- popping last 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()`	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):
        return self.items.pop()
                                  self.items[len(self.items)-1]
    def peek(self):
        return self.items[-1]

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

    def __str__(self):
        strResult = ""
        for index in range(len(self.items)-1,-1,-1):
            strResult += str(self.items[index])+" "
        return strResult
```

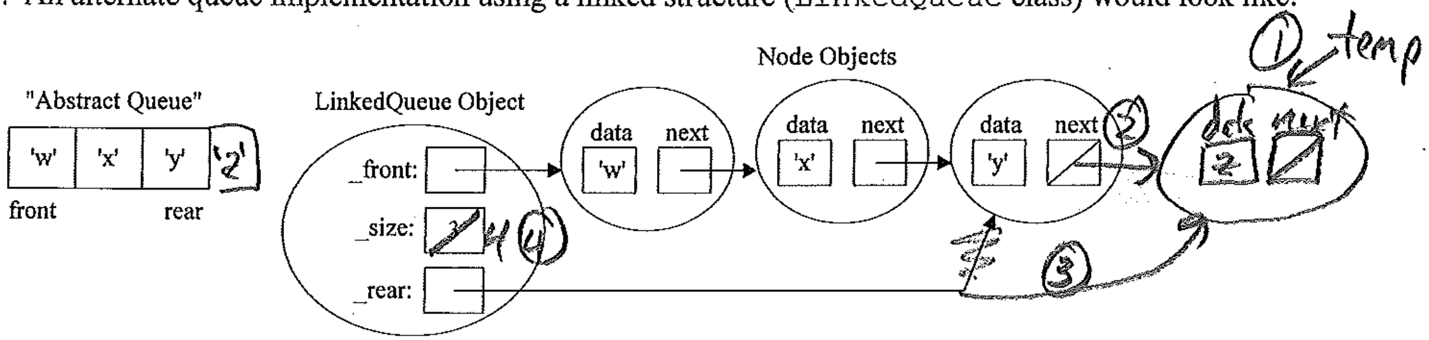
a) Complete the _peek, and __str__ methods

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(n2)

Strings are immutable

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



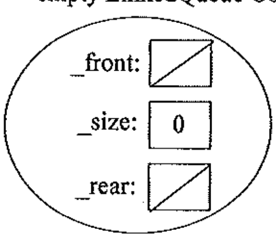
a) Draw the picture and number the steps for the `enqueue` method of the "normal" case (non-empty queue) above?

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

```
temp = Node(item)
self._rear.setNext(temp)
self._rear = temp
self._site += 1
```

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

empty LinkedQueue Object



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