Disclaimer

This complementary study package is provided by Easy 4.0 Education Inc. and its affiliated mentors. This study package seeks to support your study process and should be used as a complement, **NOT** substitute to course material, lecture notes, problem sets, past tests and other available resources.

We acknowledge that this package contains some materials provided by professors and staff of the University of Toronto, and the sources of these materials are cited in details wherever they appear.

This package is distributed for free to students participating in Easy 4.0's review seminars, and are not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in this package.

Thank you for choosing Easy 4.0. We sincerely wish you the best of luck in all of your exams.

Easy 4.0 Education Inc.

Section 1. Object Oriented Design

所谓OO design指的是 object oriented design， 面向对象设计。 利用class设计来模拟出一种对象所需的data，达到描述此种对象， 并对此对象提供可操作性的方法。

Class:

Four important components:
1. DOCSTRING!!!!!!!!!!
2. Attribute: variables that belong to the instance of the class.
3. Build-in methods: Functions that provide some build-in functionality.
(eg. __init__, __eq__, __repr__, __str__, __hash__)
4. Methods: Functions for the class, shared by all instances.

Docstring:

```
class Human:
    """ A Human representation.

    name - The human's name.
    cell - The human's cellphone number
    """
    name: str
    cell: str.

    def __init__(self, name: str, cell: str) -> None.
        """ The init method.

        example.

        """
        self.name = name
        self.cell = cell.
```

Attribute:

Three types of attributes in init:

```
def __init__(self, name, cell=None):
    self.id = name
    self.cell = cell
    self.friends = []
```

optional.

normal.

Private-ness:

Human('Gary')

Human(self, n, c=None, d)

In case of we have some attributes in function that does not want them expose to client, we want to have some privacy on variables, and limit the access of the variables (methods as well). However, since python does not have strict definition of private-ness, 'underscore' before the variable name means '大家都是成年人，请尊重别人的隐私'.

Should not access.

Build-in methods:

Methods that are used to preform some build-in functionality.

def __eq__(self, other):

```
return type(self) == type(other) and
       self.name == other.name
```

def __str__(self):

```
template = "Human, {}  cell: {}"
return template.format(self.name,
                       self.cell)
```

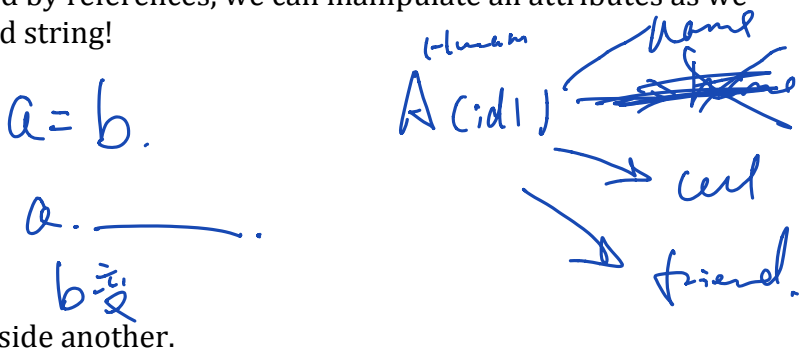
Method:

Function that manipulates the objects, that's why we need self as parameter.

a. Interface : 在csc148中 我们可以 粗略的将interface理解为 the function signatures of class methods.

b. DOCSTRING!!!!!!

c. As objects are passed by references, we can manipulate all attributes as we want, even integer and string!



d. Composition: one inside another.
You may need to use methods of the attribute to manipulate composited attribute.

Property function:

如果对于某个变量, 定义了property, 那么这个function的access将通过property中传入的function决定.

```
class property(object)
 |  property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
 |
 |  fget is a function to be used for getting an attribute value, and likewise
 |  fset is a function for setting, and fdel a function for del'ing, an
 |  attribute.  Typical use is to define a managed attribute x:
 |
 |  class C(object):
 |      def getx(self): return self._x
 |      def setx(self, value): self._x = value
 |      def delx(self): del self._x
 |      x = property(getx, setx, delx, "I'm the 'x' property.")
```

As in the example above, x is a MASK of attribute _x, we defined functions for getting, setting and delete x so that it looks like we have a variable x. However, the attribute we stored in class C is _x.

```
 |  class C(object):
 |      def getx(self): return self._x
 |      def setx(self, value):
 |          if value > 100:
 |              self._x = value
 |          else:
 |              print('Can not assign')
 |      def delx(self): del self._x
 |      x = property(getx, setx, delx, "I'm the 'x' property.")
```

```
>>> test = C()
>>> C.x = 30    # C.setx(30)
Can not assign.
```

Exception Class:

本学期Heap没有着重讲Exception, 但是对于Exception, 要知道如何raise, 如何declare.

Declaration of Exception:

```
class TaiShuaiException(Exception):
    pass
```

Raise Exception:

```
raise TaiShuaiException('Wo De Tian A!')
```

Try Except block: # 目测不会考..

```
try:
    # Code that may cause an exception
except SomeException:
    # Do something
except Exception:
    # Do something else
else:
    # Do if no exception happens
finally:
    # Always do at the end.
```

Assert:

Assert 就是简化版的Exception 使用...
Assert的使用方法是

```
assert _____condition_____
```

在condition得到False的时候， 会raise AssertionError（）

Consider a Help Queue system, like the one in the CSC Help Centre. Students can optionally choose a course they want help with (or not), then swipe their TCard to be added to the help queue. Then, instructors and TAs can choose the next student from the help queue to help. Assume that there are no duplicate student names.

Students can swipe their card again if they're already in the help queue to be reminded of their number. Instructors and TAs can either choose the next student from the whole help queue, or the next student that matches a particular course.

On the following page, write the **HelpQueue** class so that it implements the Help Queue system described above, and interacts properly with the client code in the **if __name__ == '__main__'** block below. (That is, that the client code runs without any AssertionErrors.)

```
class HelpQueueEntry:
    """ A simple class to represent one entry in the help queue """

     def __init__(self, name, number, course=''):
        """ Create this new HelpQueueEntry representing a student name,
        the queue position number, and an optional course.

        @param self HelpQueueEntry: this help queue entry
        @param name str: the name of the student
        @param number int: the position in the queue
        @param course str: the course to get help with
        """
        self._name, self._number, self._course = name, number, course

    def _get_name(self):
        return self._name

    def _get_number(self):
        return self._number

    def _get_course(self):
        return self._course

    name = property(_get_name)
    number = property(_get_number)
    course = property(_get_course)


if __name__ == '__main__':
    hq = HelpQueue()
    assert hq.process_swipe('Amy', 'CSC108') == 0
    assert hq.process_swipe('Bo') == 1
    assert hq.process_swipe('Chen', 'CSC148') == 2
    assert hq.process_swipe('Amy') == 0
    assert hq.get_next_student('CSC148').number == 2
    assert hq.get_next_student('CSC148') is None
    assert hq.get_next_student('CSC165') is None
    assert hq.get_next_student().number == 0
    assert hq.get_next_student().number == 1
    assert hq.get_next_student() is None
```

```
class HelpQueue:

    """ A class to represent a help queue.

    === Attributes ===
    @param list[HelpQueueEntry] helpqueue: student entries waiting for help
    @param int next_number: number to assign to next entry to join helpqueue
    """
```

```
def __init__(self):
    self.helpqueue = []
    self.next_number = 0

def process_swipe(self, name, course=None):
    for entry in self.helpqueue:
        if entry.name == name:
            return entry.number

    if course == None:
        course = ''

    entry = HelpQueueEntry(name, self.next_number,
                           course)

    self.helpqueue.append(entry)
    self.next_number += 1

    return entry.number
```

```
def get_next_student(self, course=''):
    if not course:
        if len(self.helpqueue) > 0:
            return self.helpqueue.pop(0)
        else: return None
    else:
        for entry in self.helpqueue:
            if entry.course == course:
                self.helpqueue.remove(entry)
                return entry
        return None.
```

Section 2. Inheritance

继承可以使子类（subclass） 获得父类（superclass）的全部method， 可使我们的oo design 更加系统化， 并且eliminate duplicates.

自己没有，向上找

Python在class中寻找function declaration时, 是由下向上寻找的, 先在自身寻找有没有, 如果自身没有, 就到父类中寻找, 然后向爷爷类寻找 ……

① reduce duplicate code

② raise NotImplementedError to force subclass to implement.

Declaration of inheritance:

```
class A(B):
```

class A inherits from class B

Assume we have superclass Gary as follow:

```
class Gary:

    def __init__(self):
        print('initializing Gary')
        self.name = 'Gary'
```

Three ways to use inheritance:

1. 完全啃老型：

```
class LittleGary(Gary):
    pass
```

如果在inheritance中, 完全没有declare过superclass中的某些method, 那么代表这个method被完全继承。

2. Override 完全覆盖

```
class Jacky(Gary):

    def __init__(self):
        self.name = 'Jacky'
```

如果在class中declare了父类中相同signature的function, 那么就会使用当前function中的 function definition 而不再向上查询, 换言之，覆盖掉了继承来的function definition. 常见的， 我们会在superclass中是 raise NotImplementedError() 的时候， 在subclass中使用override的情况。

3. Extend 锦上添花

```
class GaryB(Gary):

    def __init__(self):
        super().__init__()
        print('initializing GaryB')
        self.like = 0
```

>>> a = GaryB().
Initializing Gary
Initializing GaryB
>>> a.name
'Gary'

在使用这种方法inherit的时候, 和完全覆盖一样, python 也会使用当前definition， 但是， 由于我们使用了super() method来call 父类当中的method. 我们相当于对父类中的method进行了更改.

常见的有一下两种情况：

1. 我们会在init class中很多的使用Extend的方式。
2. body中， 出现了对superclass中同一function的reference.

super().method——.

Note: super() 代表把当前object convert到super class, 因此, 我们不需要在parameter当中加入self.

Gary.__init__(self).

This question is about Object-Oriented Design.
You will design a set of classes and methods to satisfy the following specifications:

You have to maintain a simple social network that consists of individual profiles. Each profile holds a person's name, date of birth (including day, month and year) and a data structure that stores that person's friends and makes it possible to retrieve a friend's profile given his or her name. Each profile also holds a list of posts that person has created.
Each post has a date of publication and a list of names of people who are tagged in it, and it supports tagging additional friends after it has been created. Each post is either a note, containing the text of the note, or a picture, containing a string-based path to the image file.

Each profile supports:

- adding or removing a friend given their profile
- checking whether it is that person's birthday
  (assume you can retrieve today's date without passing it in)
- posting a note given its content and a list of friend names to tag in it
- posting a picture given its path and a list of friend names to tag in it

For every class you decide to implement:

- Write the class definition line. ✓
- Write a full `__init__` method and docstring (including signature line). ✓
- Write the method definition line and docstring (including signature line) for the rest of the methods in the class.

Things you do *not* need to write:

- the class docstring ✗
- any code inside methods that are not the `__init__` method of each class ✗

```
class Profile:
    def __init__(self, name: str, dob: str) -> None:
        self.name = name
        self.dob = dob
        self.friends = {}
        self.posts = []  # List[Post]
    def add_friend(self, friend: Profile) -> None:
    def delete_friend(self, friend: Profile) -> None.
    def retrieve_friend(self, name: str) -> Profile:
    def check_birthday(self) -> bool.
    def make_post(self, post: Post) -> None:
```

```
class Post:
    def __init__(self, date: str, names: List[str]) -> None:
        self.date = date
        self.names = names[:]
    def add_name(self, name: str) -> None:

class Note(Post):
    def __init__(self, date: str, names: List[str],
                 text: str):
        # Extend.
        super().__init__(date, names)
        self.text = text

class Picture(Post):
    def __init__(self, date: str, names: List[str],
                 path: str):
        # Extend.
        Post.__init__(self, date, names)
        self.path = path
```

Section 3. Abstract Data Type

Focuses on what can it do in real world rather than how we implement it in detail. It Specifies the intended meaning of the data it stores, and the operations it provides on data but **DOES NOT TALK HOW** do we achieve that in any particular programming language.

ADT examples: Stack, Queue, Hash table.

API for Stack

```
class Stack(object):
    def __init__self)
    def add(self, value)
    def remove(self)
    def is_empty(self)
```

Stack is LIFO, when you put everything in from a Stack to another, you reverse everything in the stack.

API for Queue

```
class Queue(object):
    def __init__self)
    def add(self, value)
    def remove(self)
    def is_empty(self)
```

Queue is FIFO,

题型一：颠来倒去，记得放回去

```
def remove_bottom(s: Stack) -> object:
    """ Return the item at the bottom of the stack and remove it from
    stack
    """
    temp = Stack()
    while not S.is_empty():
        temp.add( S.remove())

    result = temp.remove()
    while not temp.is_empty()
        S.add(temp.remove())

    return result .
```

```
def contains(s: Stack, item: object) -> bool:
    """ Return True iff s contains item, otherwise False.
    """
    result = False.
    temp = Stack()
    while not s.is_empty().
        obj = S.remove()
        if obj == item:
            result = True
        temp.add(obj)

    while not temp.is_empty():
        s.add( temp.remove()).

    return result
```

题型二： 继承&使用ADT

We want to create a subclass of `Stack` called `GuardedStack`, which has the following differences:

- It has a new attribute `is_good`, which is a **function** that takes an object and returns a boolean value. This attribute is initialized through a parameter to the `GuardedStack` constructor.
- When pushing a new item onto a `GuardedStack`, the stack's `is_good` attribute is first called on the item. If this returns `True` then the item is pushed onto the stack, otherwise a `ValueError` is raised.

Here is a sample usage of `GuardedStack`.

```
>>> def is_even(num):
...     return num % 2 == 0
...
>>> g = GuardedStack(is_even)
>>> g.push(2)
>>> g.push(4)
>>> g.push(5)
ValueError  # error raised here
>>> g.push(16)
>>> g.pop()
16
>>> g.pop()
4
```

In the space below, show how to override the relevant `Stack` methods in the `GuardedStack` class. You *must* properly call superclass methods when appropriate. (Note: neither method is very long.)

```
class GuardedStack(Stack):
    def __init__(self, is_good):
        # Extend
        super().__init__()
        self.is_good = is_good

    def add(self, item):
        if not self.is_good(item):
            raise ValueError()
        super().add(item)
```

Callable[[object]]

Stack.add(self, item)

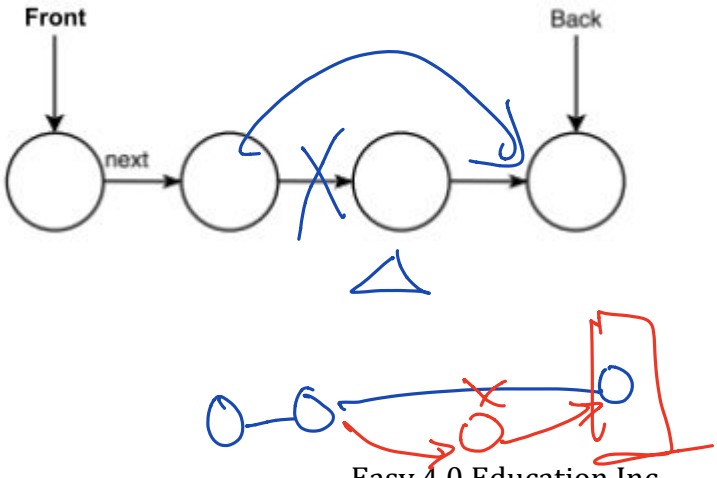
Section 4. Linkedlist

Linkedlist 是148中我们接触到第一个全新的data structure, 他作为最主要的linear data structure之一, 是每一个computer scientist都要掌握的。Linkedlist的主要考点在于对data structure的理解和loop 的应用。 Final中出现的linkedlist问题， 一般多为**变种题目**。

Linkedlist class implementation from lecture

```
class LinkedListNode:
    """
    Node to be used in linked list

    === Attributes ===
    next_ - successor to this LinkedListNode
    value - data represented by this LinkedListNode
    """
    next_: Union["LinkedListNode", None]

    def __init__(self, value: object,
                 next_: Union["LinkedListNode", None]=None) -> None:
        """
        Create LinkedListNode self with data value and successor next

        >>> LinkedListNode(5).value
        5
        >>> LinkedListNode(5).next_ is None
        True
        """
        self.value, self.next_= value, next_


class LinkedList:
    """
    Collection of LinkedListNodes

    === Attributes ==
    front - first node of this LinkedList
    back - last node of this LinkedList
    size - number of nodes in this LinkedList, >= 0
    """
    front: Union[LinkedListNode, None]
    back: Union[LinkedListNode, None]
    size: int

    def __init__(self) -> None:
        """
        Create an empty linked list.
        """
        self.front, self.back = None, None
        self.size = 0
```

一定要记住这张图！！！



LinkedList 注意事项：

1. 图中front, back 和 next都是箭头，代表pointer, 他们都是**指向一个node**, 或者**指向None**！

2. 圆圈代表一个node, 每个node当中有value和next_, 也就是pointer to next。

3. Front是整个linkedlist里面最重要的node, 由于linkedlist是线性的, 我们是在通front来访问到他后面的所有node, 所以head是不能乱动的. 因此, 当我们要loop over一个linkedlist的时候, 我们要定义一个temporary pointer, 然后将这个 temporary pointer向后移动来大达到loop over整个linkedlist的效果。

4. Back是整个linkedlist中唯一一个next_为None的node, 当我们在loop over 一个linkedlist的时候, 要利用这一性质来判断是否达到linkedlist的尾部。

Note: sometimes we check temp, sometimes we check temp.next_

5. Loop 先Temp

6. 记得还有个size哟，千万别忘记update。

7. 无论在对linkedlist 进行添加或者删除的时候，都要注意他前后的node，在添加时要注意不要忘记将两侧都连起来（需要记录当前位置的前一个node），在删除只需要把前一个node和当前node的next连接到一起即可。头尾一定要注意！如果删除的或者添加是头或者尾 需要额外进行赋值，来重新规划头尾。

LinkedList 基本操作：

```
def add_front(self, item):
    self.front = LinkedListNode(item, self.front)
    self.size += 1
    if self.size == 1:
        self.back = self.front
```

```
def remove_front(self):
    self.front = self.front.next_
    self.size -= 1
    if not self.front:
        self.back = None
```

基础题型一： loop over

```
def double_all_passr(self, passr):
    """ Double all nodes by add the same node after them.

    Passr is a function that you pass an value of node in,
    and it returns bool.

    @type self: Linkedlist
    @type passr: function(obj) -> bool
    @rtype: None

    >>> a = LinkedList([1, 2, 3])
    >>> def foo(tmp): return tmp % 2 == 1
    >>> a.double_all_passr(foo)
    >>> print(a)
    1 -> 1 -> 2 -> 3 -> 3
    """
```

```
temp = self.front
while temp is not None:
    if passr(temp.value):
        temp.next_ = LinkedListNode(temp.value,
                                    temp.next_)
        self.size += 1
        temp = temp.next_.next_
    else:
        temp = temp.next_
if self.back and passr(self.back.value):
    self.back = self.back.next_
```

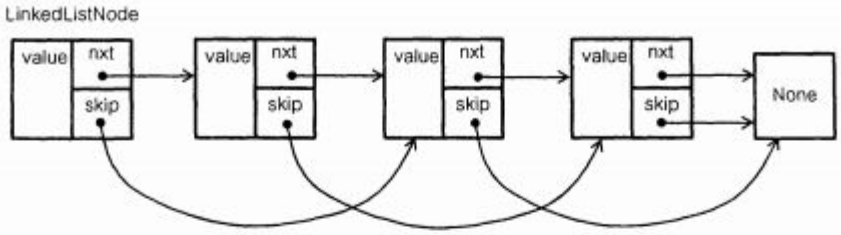
基础题型二： 找Node

```
def delete_first_odd(self) -> None:
    """ delete all nodes that contains odd value.
    Delete the first odd node in self.

    @type self: Linkedlist
    @rtype: None

    >>> a = LinkedList()
    >>> a.load_list([1, 2, 3, 4, 5])
    >>> a.delete_first_odd()
    >>> print(a)
    2 -> 3 -> 4 -> 5 ->|
    >>> a.delete_first_odd()
    >>> print(a)
    2 -> 4 -> 5 ->|
    """
```

```
if self.front and self.front.value % 2 == 1:
    self.front = self.front.next_
    self.size -= 1
    if self.size == 0:
        self.back = None
else:
    temp = self.front
    while temp.next_ and temp.next_.value % 2 != 1:
        temp = temp.next_
    if temp.next_:
        temp.next_ = temp.next_.next_
        self.size -= 1
        if not temp.next_:
            self.back = temp
```

165Final式题型：

A skiplist is defined as a sorted linked list which in addition to the nxt reference, stores an additional "skip" reference. A skip reference refers to the node located immediately after nxt, if such a node exists, or None otherwise, as shown in the diagram below.



Your task is to implement two methods: precursors, and insert. These two methods are necessary to insert a new element with a given value in the list, in a way that the list remains sorted in ascending order of the node values, and the skip references are correctly maintained for every list node.

1. The insert method takes as a parameter a value (the value of the new node to be created and inserted in the list by this method), and the two nodes that would precede the new node once inserted (the two nodes with the highest two values less than the value passed as a parameter). This method will insert the new node into the list, and adjust all links necessary. Please be careful to correctly adjust the front, back, and size as well, when necessary.

2. The precursors method takes a value as a parameter and returns a tuple containing two nodes. The second node in the tuple is the list node with the largest value less than the value passed as a parameter, if such a node exists, or None otherwise. The first node is the predecessor of the second node from the tuple, if such a node exists, or None otherwise. If the list is empty, or the insertion must be done before the front node, then this method will return a tuple containing two None values.

Read the declaration of classes LinkedListNode and LinkedList below, and then implement the two methods of class LinkedList, on Pages 14 and 15. Hint: draw diagrams first!

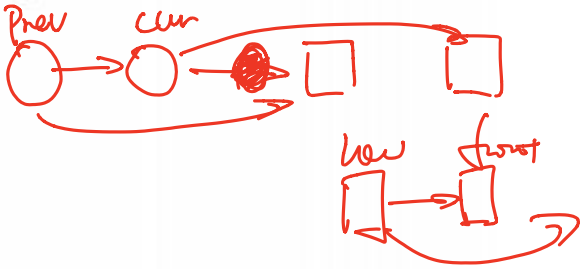
```
class LinkedListNode:
    """ Node to be used in linked list """

    def __init__(self, value, nxt=None):
        """ Create LinkedListNode self with data value and successor nxt.

        @param LinkedListNode self: this LinkedListNode
        @param int value: data of this linked list node
        @param LinkedListNode|None nxt: successor to this LinkedListNode.
        @rtype: None
        """
        self.value, self.nxt = value, nxt
        if nxt is not None:
            self.skip = nxt.nxt
        else:
            self.skip = None

        # continued on following page
```

```
    def __str__(self):
        """ Return a user-friendly representation of this LinkedListNode.

        @param LinkedListNode self: this LinkedListNode
        @rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        s = "{} ->".format(self.value)
        cur_node = self
        while cur_node is not None:
            if cur_node.nxt is None:
                s += "|"
            else:
                s += " {} ->".format(cur_node.nxt.value)
            cur_node = cur_node.nxt
        return s

    # end of LinkedListNode class

class LinkedList:
    """ Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList a non-negative integer
    """

    def __init__(self):
        """ Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back = None, None
        self.size = 0

    def __str__(self):
        """ Return a human-friendly string representation of LinkedList self.

        @param LinkedList self: this LinkedList
        @rtype: str

        >>> lnk = LinkedList()
        >>> lnk.insert(5, lnk.precursors(3)[0], lnk.precursors(3)[1])
        >>> print(lnk)
        5 ->|
        """
        return str(self.front)
```

```
def precursors(self, value):
    """
    Returns a tuple containing the two list nodes with the two highest
    values which are less than the method argument 'value'.

    @param LinkedList self: this LinkedList
    @param int value: value to insert
    @rtype: (LinkedListNode|None, LinkedListNode|None)

    >>> lnk = LinkedList()
    >>> lnk.precursors(3)
    (None, None)
    >>> a = LinkedListNode(3)
    >>> lnk.front, lnk.back, lnk.size = a, a, 1
    >>> lnk.precursors(1)
    (None, None)
    >>> b = LinkedListNode(1, a)
    >>> lnk.front, lnk.size = b, 2
    >>> pre1 = lnk.precursors(5)[0]
    >>> pre2 = lnk.precursors(5)[1]
    >>> pre1.value, pre2.value
    (1, 3)
    """
```

first = None
second = None
cur = self.front
while cur and cur.value < value:
    first = second
    second = cur
    cur = cur.next
return first, second

```
def insert(self, value, prev, cur):
    """
    Inserts a new node with value after node cur. Updates all links correctly.
    This is a method of class LinkedList.

    @param LinkedList self: this LinkedList
    @param int value: value to insert
    @param LinkedListNode|None cur: node before the one we are inserting
    @param LinkedListNode|None prev: node before cur
    @rtype: None

    >>> lnk = LinkedList()
    >>> lnk.insert(3, lnk.precursors(3)[0], lnk.precursors(3)[1])
    >>> lnk.insert(0, lnk.precursors(0)[0], lnk.precursors(0)[1])
    >>> lnk.insert(2, lnk.precursors(2)[0], lnk.precursors(2)[1])
    >>> lnk.insert(1, lnk.precursors(1)[0], lnk.precursors(1)[1])
    >>> print(lnk.front)
    0 -> 1 -> 2 -> 3 ->|
    >>> print(lnk.back)
    3 ->|
    >>> lnk.size
    4
    >>> print(lnk.front.skip)
    2 -> 3 ->|
    >>> print(lnk.front.nxt.skip)
    3 ->|
    """
```




```
new_node = LinkedListNode(value).
if prev:
    prev.skip = new_node

if not cur:
    new_node.next = self.front
    if self.front:
        new_node.skip = self.front.next
    else:
        self.back = new_node.
    self.front = new_node.
else: origin_next = cur.next
      origin_skip = cur.skip
      new_node.next = origin_next
      new_node.skip = origin_skip.
      cur.next = new_node
      cur.skip = origin_next
      if not new_node.next:
```

self.back = new_node

self.size += 1

Question 6. [12 MARKS]

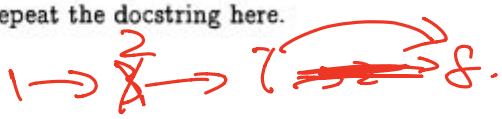
LeakyQueue is a subclass of Queue that implements a First In Usually First Out (FIUFO) Queue. LeakyQueue has one additional method defer(from_value, to_value). What defer does is find the first occurrence of from_value and replace it with the first occurrence of to_value that follows it, removing to_value from its previous position in the queue. If there is no instance of from_value earlier than an instance of to_value, then this method does nothing.

Read over the implementation of class Queue below.

```
class LLNode:
    def __init__(self, value, nxt=None):
        ''' (LLNode, object, LLNode) -> NoneType

        Create LLNode (self) with data value and successor nxt.
        '''
        self.value, self.nxt = value, nxt


class Queue:
    def __init__(self):
        ''' (Queue) -> NoneType

        Create and initialize new queue self.
        '''
        self._front = self._back = None

    def enqueue(self, o):
        ''' (Queue, object) -> NoneType

        Add o at the back of this queue.
        '''
        new_node = LLNode(o)
        if self._back:
            self._back.nxt = new_node
            self._back = new_node
        else:
            self._back = self._front = new_node

    def dequeue(self):
        ''' (Queue) -> object

        Remove and return front object from self.
        '''
        new_value = self._front.value
        self._front = self._front.nxt
        return new_value

    def is_empty(self):
        ''' (Queue) -> bool

        Return True queue self is empty,
        False otherwise.
        '''
        return self._front == None
```

Now implement the method defer below. You don't need to repeat the docstring here.

```
def defer(self, from_value, to_value):
```



```
    temp = self.front
    while temp and temp.value != from_value:
        temp = temp.next_
    if temp:
        from_node = temp
        while temp.next_ and temp.next_.value != to_value:
            temp = temp.next_
        if temp.next_:
            temp.next_ = temp.next_.next_
            self.size -= 1
            if not temp.next_:
                self.back = temp
        from_node.value = to_value
```

Section 5. List Comprehension

Use one line of code to accomplish the construction of new list.

a) No condition

[表达式 for loop]

b) Filter

[表达式 for loop if条件]

c) If and else

[表达式 if条件 else 其他活 for loop]