Abstract Data Structure

LINEAR DATA STRUCTURE
LINKED LIST

APIs

- In computer programming, an Application Programming Interface (API) is a set of routines, protocols, and tools for building software applications.
- An API expresses a software component in terms of its operations, inputs, outputs, and underlying types.

APIs

- An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising each other.

- A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

API in Object Oriented Programming

- In its simplest form, an object API is a description of how objects work in a given object-oriented language
  - usually it is expressed as a set of classes with an associated list of class methods.

- The API in this case can be conceived of as the totality of all the methods publicly exposed by the classes (usually called the class interface).
  - This means that the API prescribes the methods by which one interacts with/handles the objects derived from the class definitions.

The Library Project API

Item

```
BOOK = 'BOOK'
CD = 'CD'
DVD = 'DVD'
OTHER = 'OTHER'
VHS = 'VHS`
-----------------------
__init__(title, author,
         media, uid)
get_UID()
get_author()
get_borrower()
get_media()
get_title()
isavailable()
loan_to(member_uid)
returned()
```

Member

```
__init__(firstname, surname,
         postcode, uid)
add_borrowed_item(item_uid)
get_UID()
get_borrowed()
get_firstname()
get_postcode()
get_surname()
has_items()
remove_borrowed_item(item_uid)
set_postcode(new_postcode)
set_surname(new_name)
```

Library

```
__init__()
add_item(item)
add_member(member)
borrow(item_uid, member_uid)
delete_member(member_uid)
get_member(uid)
return_item(item_uid)
...
```

Using the Library Project API

- See code library_test_suit.py

Abstract Data Type

IMPLEMENTATION OF STACKS & QUEUES

ADT API

We will focus on two Abstract Data Types:

- FIFO (Queue)
  - Constructor
  - `enqueue(obj)`: add obj at the end of the queue
  - `dequeue()`: remove and return the object at the front of the queue
- LIFO (Stack)
  - Constructor
  - `push(obj)`: add obj at the top of the stack
  - `pop()` : remove and return the object at the top of the stack

Classes Skeleton

Stack

```
class LStack:

   def __init__(self):
      pass

   def push(self, obj):
      pass

   def pop(self):
      pass
```

Queue

```
class LQueue:

   def __init__(self):
      pass

   def enqueue(self, obj):
      pass

   def dequeue(self):
      pass
```

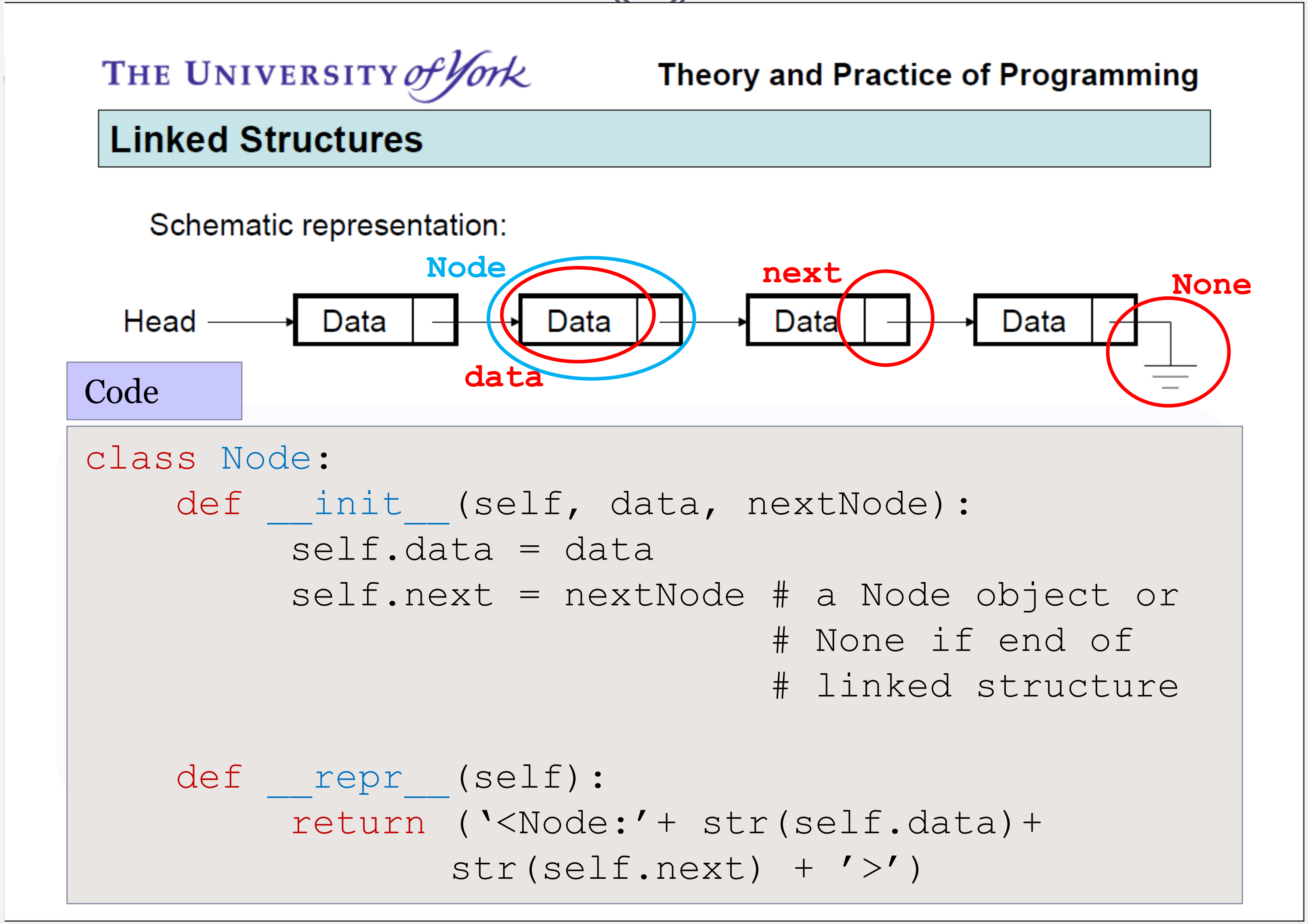
Internal Representation

- Need to decide how data will be stored
  - Arrays
  - Dynamic arrays
  - Maps (bad choice)
  - Linked lists

- The Choice MUST be hidden from API users

- Implementation could be changed, however it MUST NOT affect programs using previous implementation.

Defining Entities



THE UNIVERSITY *of York* Theory and Practice of Programming

Linked Structures

Schematic representation:

Code

```
class Node:
    def __init__(self, data, nextNode):
        self.data = data
        self.next = nextNode # a Node object or
                             # None if end of
                             # linked structure

    def __repr__(self):
        return ('<Node:'+ str(self.data)+
                str(self.next) + '>')
```

Back to our Problem

Stack

```
class LStack:

   def __init__(self):
      pass

   def push(self, obj):
      pass

   def pop(self):
      pass
```

Queue

```
class LQueue:

   def __init__(self):
      pass

   def enqueue(self, obj):
      pass

   def dequeue(self):
      pass
```

Using the class Node as an **Inner class** of LQueue

Inner Class

Code

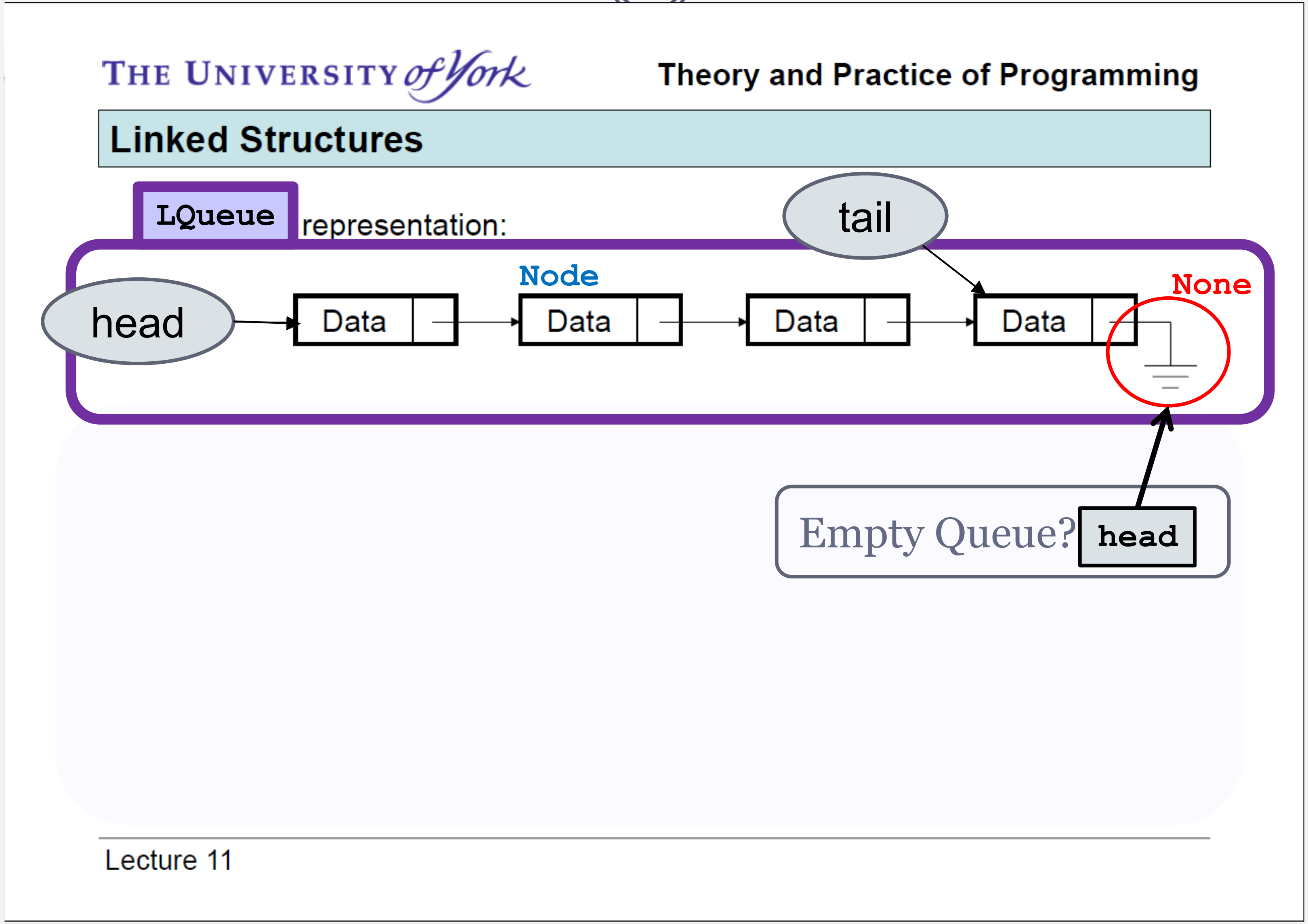
```
class LQueue:

    ## inner class, watch the indentation
    ## internal representation using linked structure
    class Node:
        def __init__(self, data, nextNode):
            self.data = data
            self.next = nextNode

        def __repr__(self):
            return (‘<Node:’+ str(self.data)+
                        str(self.next) + ’>’)

    def __init__(self):
        pass
…
```

Defining Entities



THE UNIVERSITY *of York* Theory and Practice of Programming

Linked Structures

LQueue representation:

Empty Queue? head

Implementing Queues using Linked List

Code

```
class LQueue:

   ## inner class, watch the indentation
   ## internal representation using linked structure
   class Node:
      ...

   def __init__(self):
      ’’’Construct an empty queue’’’
      self._head = None # a Node object
      self._tail = None # a Node object
      self._size = 0
…
```

Implementing enQueue

Code

```
class LQueue:
   ## inner class
   class Node: …
      . . .

   def __init__(self):…

   def enqueue(self, obj):
      '''Add obj at the end of the queue'''
      new_node = LQueue.Node(obj, None)
      self._tail = new_node
      self._size += 1
```

WRONG!

Node not correctly added to queue. You should draw the pointers to understand.

Implementing enQueue

Code

```
class LQueue:
   ## inner class
   class Node: …
      ...

   def __init__(self):…

   def enqueue(self, obj):
      '''Add obj at the end of the queue'''
      new_node = LQueue.Node(obj, None)
      self._tail.next = new_node
      self._tail = new_node
      self._size += 1
```

WRONG!

Problem when queue is empty

Implementing enQueue

Code

```
class LQueue:
    class Node: …
    def __init__(self):…

    def enqueue(self, obj):
        ’’’Add obj at the end of the queue’’’
        new_node = LQueue.Node(obj, None)
        if self._size == 0:
            self._tail = new_node
            self._head = new_node
        else:
            self._tail.next = new_node
            self._tail = new_node
        self._size += 1
```

CORRECT!

Implementing deQueue

Code

```
class LQueue:
    ## inner class
    class Node: …
        . . .

    def __init__(self):…
    def enqueue(self, obj): …

    def dequeue(self):
        ’’’Remove obj at the front of the queue’’’
        front_node = self._head
        self._head = self._head.next
        self._size -= 1
        return front_node
```

CORRECT!

Summary

We have seen:

- Inner classes

- How to implement a linear data structure
  - Queues using Linked List as opposed to arrays