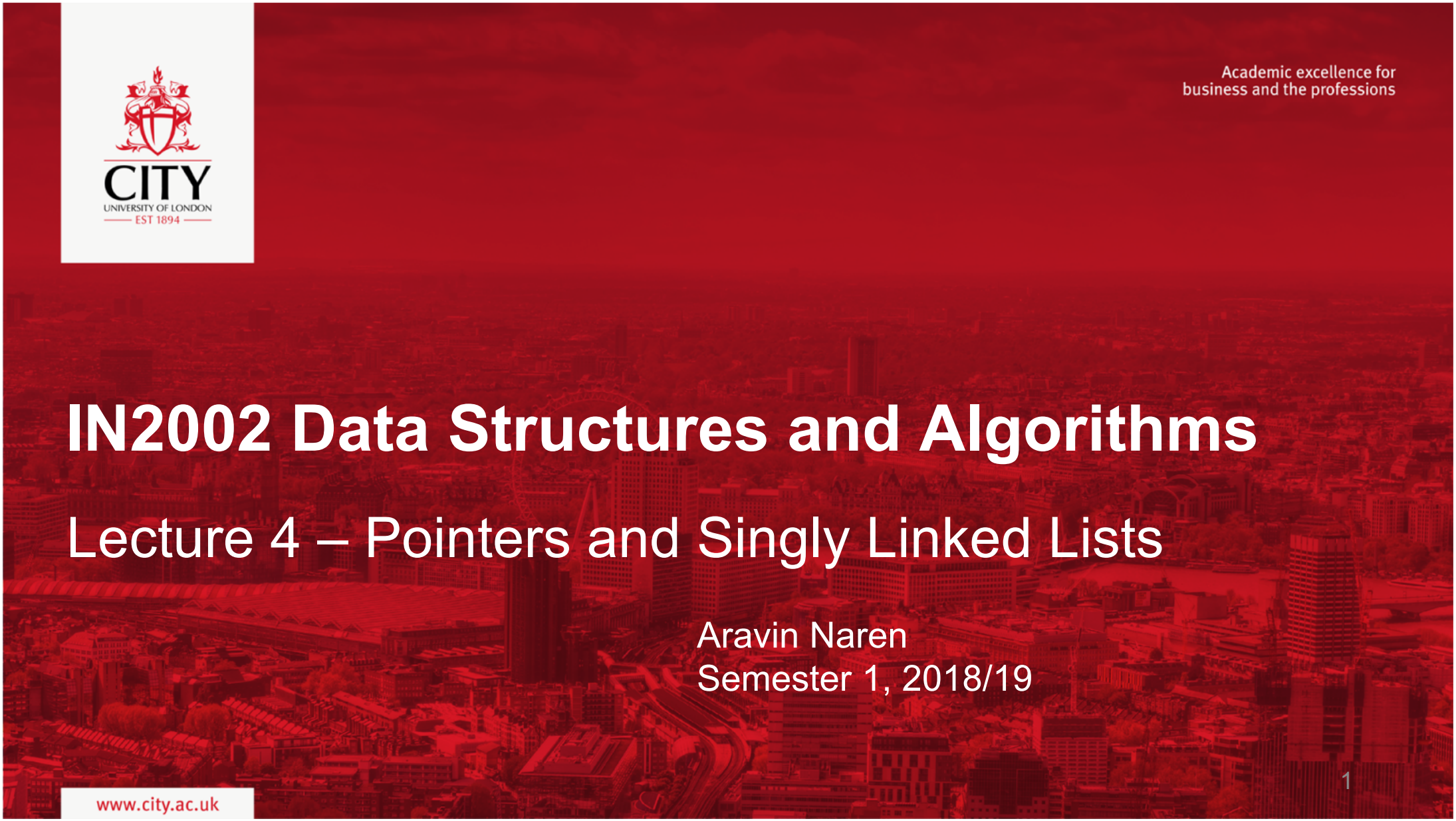
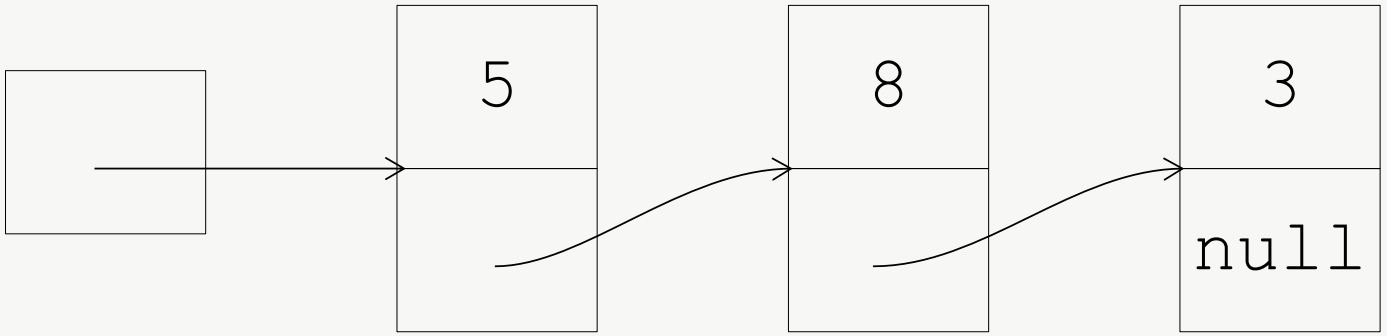

CITY
UNIVERSITY OF LONDON
EST 1894
Academic excellence for
business and the professions
IN2002 Data Structures and Algorithms
Lecture 4 – Pointers and Singly Linked Lists
Aravin Naren
Semester 1, 2018/19
www.city.ac.uk
1

Learning Objectives

- Understand and be able to use the data structure singly linked lists
- Be able to understand, apply and develop algorithms to handle singly linked lists. Including:
  - Adding elements
  - Deleting elements
  - Traversing a list

Introduction

What is this?



Database Records

- A record is a fixed collection of things of different types.
- E.g. records at a mobile phone provider:

Client ID	1234
Phone number	+44 7887 4444444
Owner	Rupert Taylor
Start date	04/10/2001
Credit	£24.53
Type of contract	Ultra
International opt.	No
Free minutes left	42

Records as Objects

```
class Mobile {
        public int clientID;
        public String number;
        public String name;
        public Date startdate;
        public int credit;
        public char contract;
        public boolean intl;
        public int seconds;
        ...
        public Mobile(int id, String nu, String na, ...) {
            clientID = c; number = nu; name = na; ....
        }
}
```

Pointers to Records

- In Java, variables cannot contain objects (record instances, in this case), but pointers to them
- Every reference type can have value null, which refers to no object at all

```
Mobile x = null;
```

- Several variables may refer to the same object:

```
Mobile y = new Mobile(1234,
      “+44 7887 4444444”,“Rupert”,...);
Mobile z = y;
```

Are two records equal?

Consider:

```
Mobile x = new Mobile(1234,
   “+44 7887 4444444”,“Rupert”,...);
Mobile y = new Mobile(1234,
   “+44 7887 4444444”,“Rupert”,...);
Mobile z = x;
```

`x == z` is

`x == y` is

`x.compareTo(y),` however, can be defined to return 0 to indicate that both objects contain the same values

What would happen here?

Consider:

```
Mobile x = new Mobile(1234,
    “+44 7887 4444444”,“Rupert”,...);
Mobile y = new Mobile(1234,
    “+44 7887 4444444”,“Rupert”,...);
Mobile z = x;
z.name = “Anne”;
y.clientID = 555;
```

What are the values of:

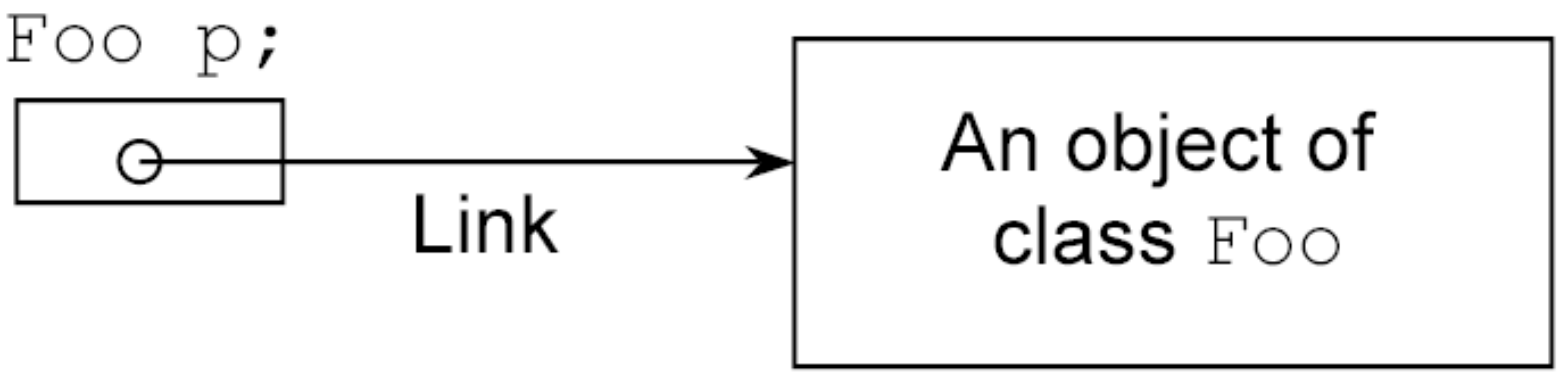
x.name	x.clientID
y.name	y.clientID
z.name	z.clientID

What are links?

All object variables in Java are actually pointers

A pointer is a link to some object, i.e. a piece of memory:

- Links can be thought of as arrows, addresses or references
- They point at an object that is somewhere in the system's memory



Pointers to an address

```
Foo p;
```

null

When you create and assign objects, the system allocates memory and assigns addresses

```
Foo p = new Foo();
```

E1E42FB

memory **address** **E1E42FB:**

New object of
class Foo

```
Foo q = p;
```

E1E42FB

Creating a pointer to the same object
at address **E1E42FB**

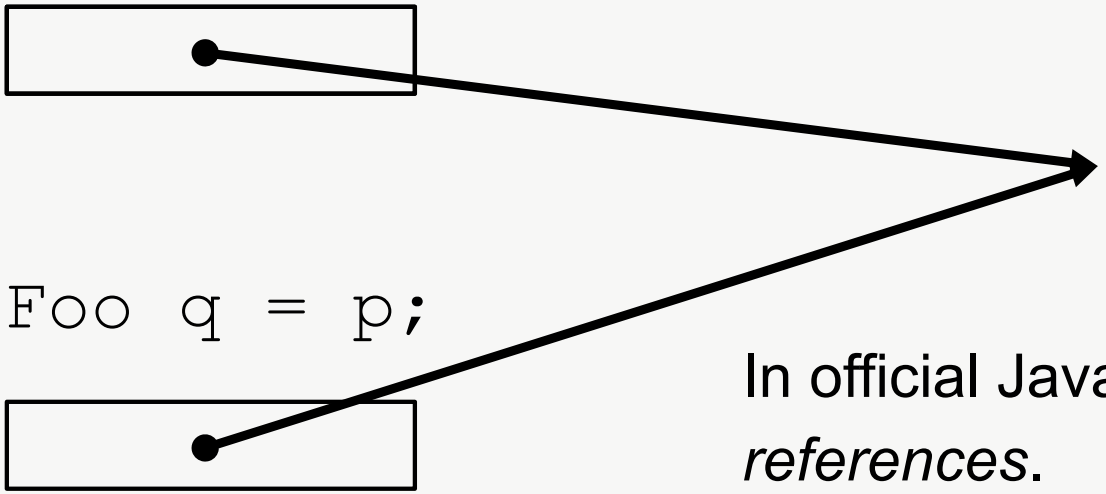
Pointers are links

```
Foo p;
```

In Java you don't need to worry about the address, just what you are pointing at.

```
Foo p = new Foo();
```

memory address **E1E42FB:**




```
Foo q = p;
```

In official Java parlance, they are called *references*.

Links in Java

All variables whose type is a class (or interface) of objects are actually pointers (“references”) to an instance of such class (or to null if empty).

This means that the **same** object can be assigned to different variables

... and, importantly, the same object’s methods and parameters can be accessed through different variables

Singly Linked Lists

Singly linked lists

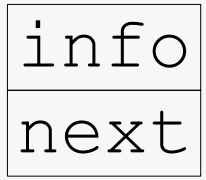
How do you remove a value from the middle of an array?
How would you do it in real life, e.g. in a long written list?
In real life, to add something at the middle of a long list what do you do?
How would you do it with arrays?

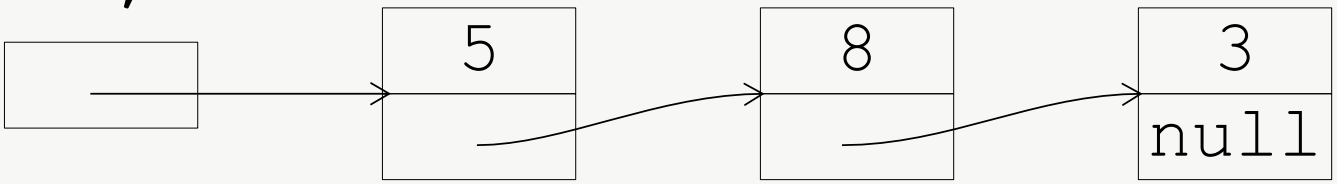
Singly linked lists use pointers as links to the next element in the list.

Nodes in singly linked lists

Each node contains a value and a pointer to the next node:

```
public class Node {
    public int info;
    public Node next;
    ...
}

Node h;
```



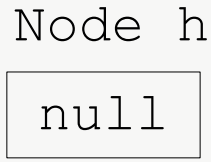


A node implementation in a singly linked list

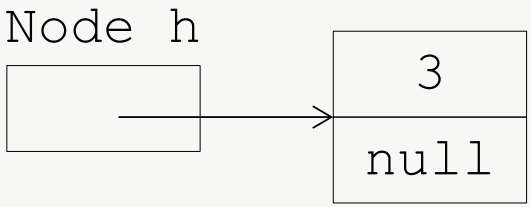
```
public class Node {
    public int info;
    public Node next;

    public Node(int i, Node n) {
       info = i; next = n;
    }

    public Node(int i) {
       this(i, null);
    }
}
```

Adding elements at the head of a list

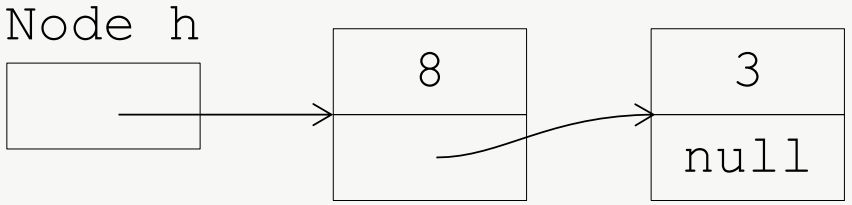
```
Node h = null;
```



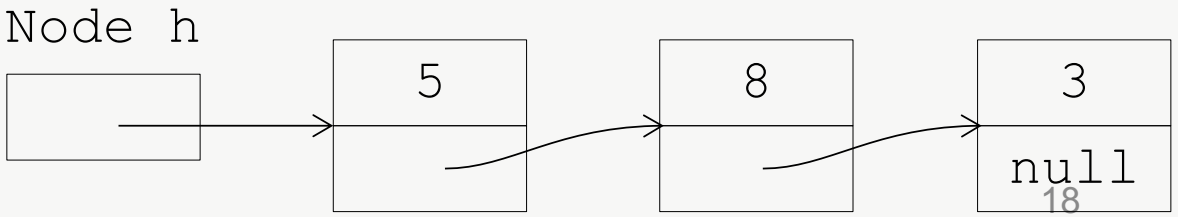

```
h = new Node(3);
```




```
h = new Node(8,h);
```

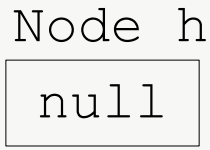



```
h = new Node(5,h);
```

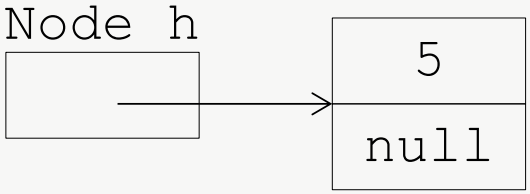


Adding elements at the tail

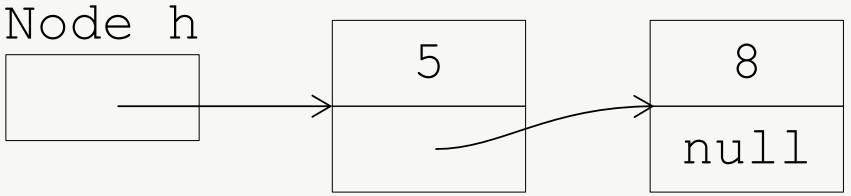
```
Node h = null;
```



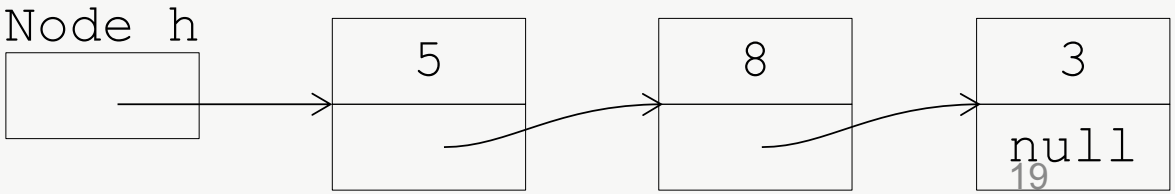

```
h = new Node(5);
```




```
h.next = new Node(8);
```

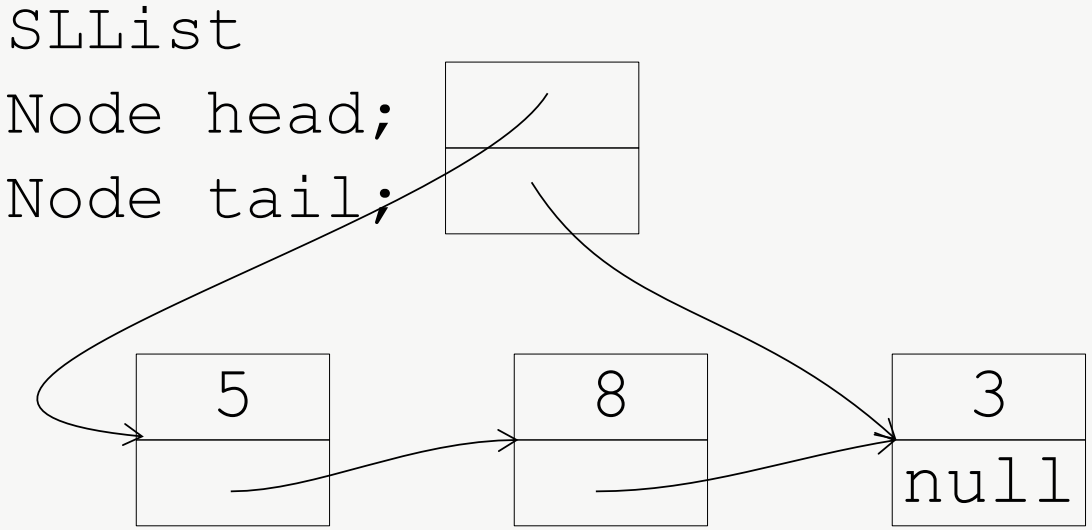



```
h.next.next = new Node(3);
```



Pointers to the head and tail

Adding elements at the tail is faster[1] if we keep a pointer to the last element (if any). We define a new class to hold the head and tail pointers:



1. Faster than adding elements at the tail while having only a pointer to the head.

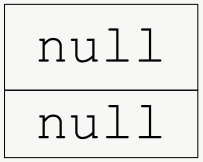
The singly linked list class

```
public class SLList {
    private Node head = null;
    private Node tail = null;

    public boolean isEmpty() {
       return head == null;
    }

    public void addToHead(int el)    {...}

    public void addToTail(int el) { ... }

    public int deleteFromHead() {...}
}
```

Special cases

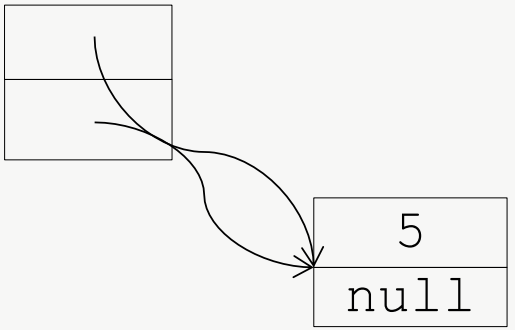
An empty list:

```
SLList
Node head;
Node tail;
```



A list with one element:

```
SLList
Node head;
Node tail;
```



What shouldn’t happen in singly linked lists

- Loops
- Errors with pointers, e.g.
  tail does not point to head.next. ... .next
- Tail node with non-null pointer
- Non-tail node with null pointer

Operations in singly linked lists

- Adding an element at the head
- Adding an element at the tail
- Deleting the head element
- Deleting the tail element

Adding an element at the head

There are two cases:

- The list contains at least one element
  (so the tail stays the same)
- The list is empty (head = tail = null)
  (we need to set the tail)

```
public void addToHead(int el) {
    head = new Node(el, head);
    if(tail == null)
       tail = head;
}
```

Adding an element at the tail

Two cases again:

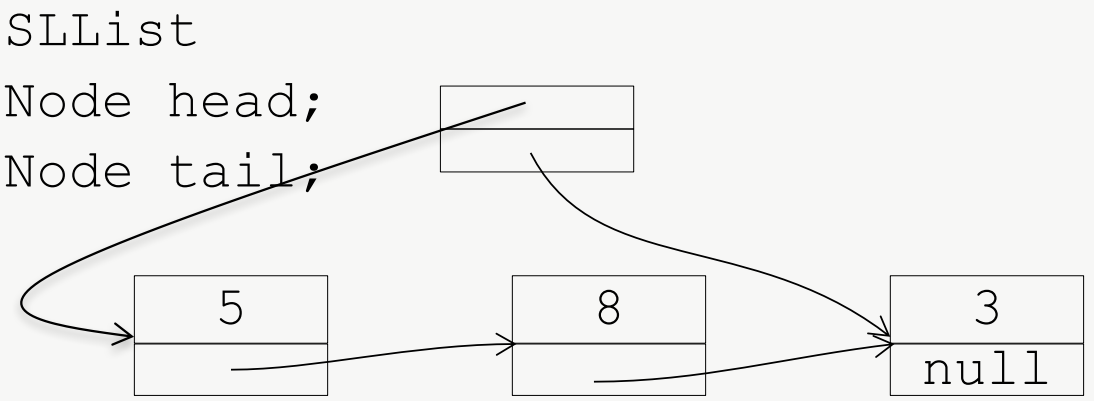
- If the list is empty, we need to set the head
- Otherwise, we need to set the pointer of the old tail

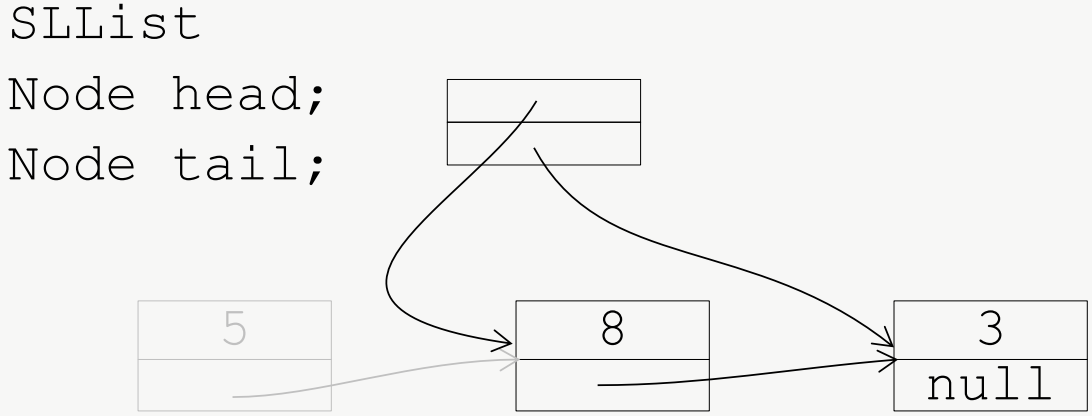
```
public void addToTail(int el) {
 if (! isEmpty()) {
    tail.next = new Node(el);
    tail = tail.next;
 }
 else
    head = tail = new Node(el);
}
```

Deleting the head element

Just move the head pointer:

(Special case:
head == tail
->
head = null;
tail = null;)





Deleting the head element (2)

If the list has more than one element, change the head
Otherwise, make the list empty

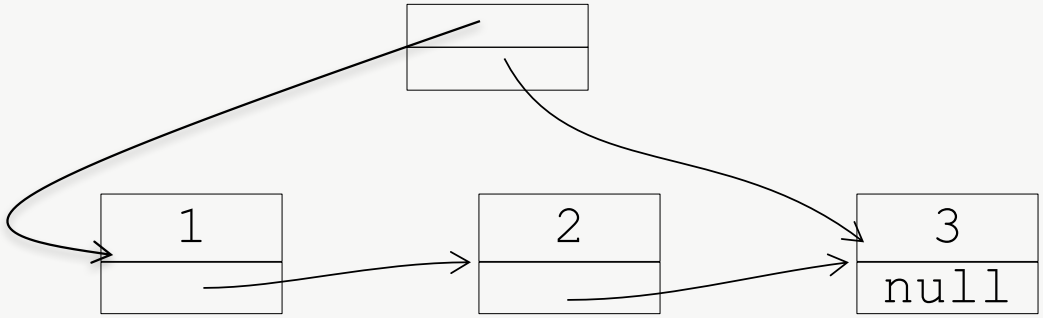
```
public int deleteFromHead() {
 int el = head.info;
 if(head == tail) // one or no elements
    head = tail = null;
 else
    head = head.next;
 return el;
}
```

Deleting the last element

We need to set `tail` to the second last node
... but it's more easily said than done:

```
if(head == tail) // one or no elements
    head = tail = null;
else
    aux = sll.head
    while (aux.next!=tail)
       aux = aux.next
    aux.next = null
    sll.tail = aux
```

Traversing a list




```
Function printList()
aux = sll.head
While (aux! = null)
    print aux.info
    aux = aux.next
```

... so to traverse the list, iteratively (or recursively!) evaluate the rest of the list (i.e., next)

Applications of singly linked lists

- A list requires O(n) space
- The operations we have defined each take O(1) time
- Lists can represent the following ADTs:
  - Stack uses
    - for push
    - for pop
  - Queue uses
    - for enqueue
    - for dequeue
- Lists may also be used directly in many algorithms
- Important high-level programming languages strongly rely on linked lists (e.g., Lisp)

Other uses?

It is possible to implement priority queues, stacks, queues and heaps using linked lists. However, is this efficient?

Let us have a closer look:

- priority queues: inserting a new element in a sorted singly linked list takes O(?)
- heaps: searching a parent or child in a singly linked list is O(?)

Some observations

- Extensible arrays efficiently implement the Stack ADT, as well as the array operations, overhead for additional space
- Singly linked lists:
  - efficiently implement the Stack and Queue ADTs, and have many other uses
  - can only move in one direction, accessing elements in the middle or rear part is difficult
  - incur overhead for the pointers

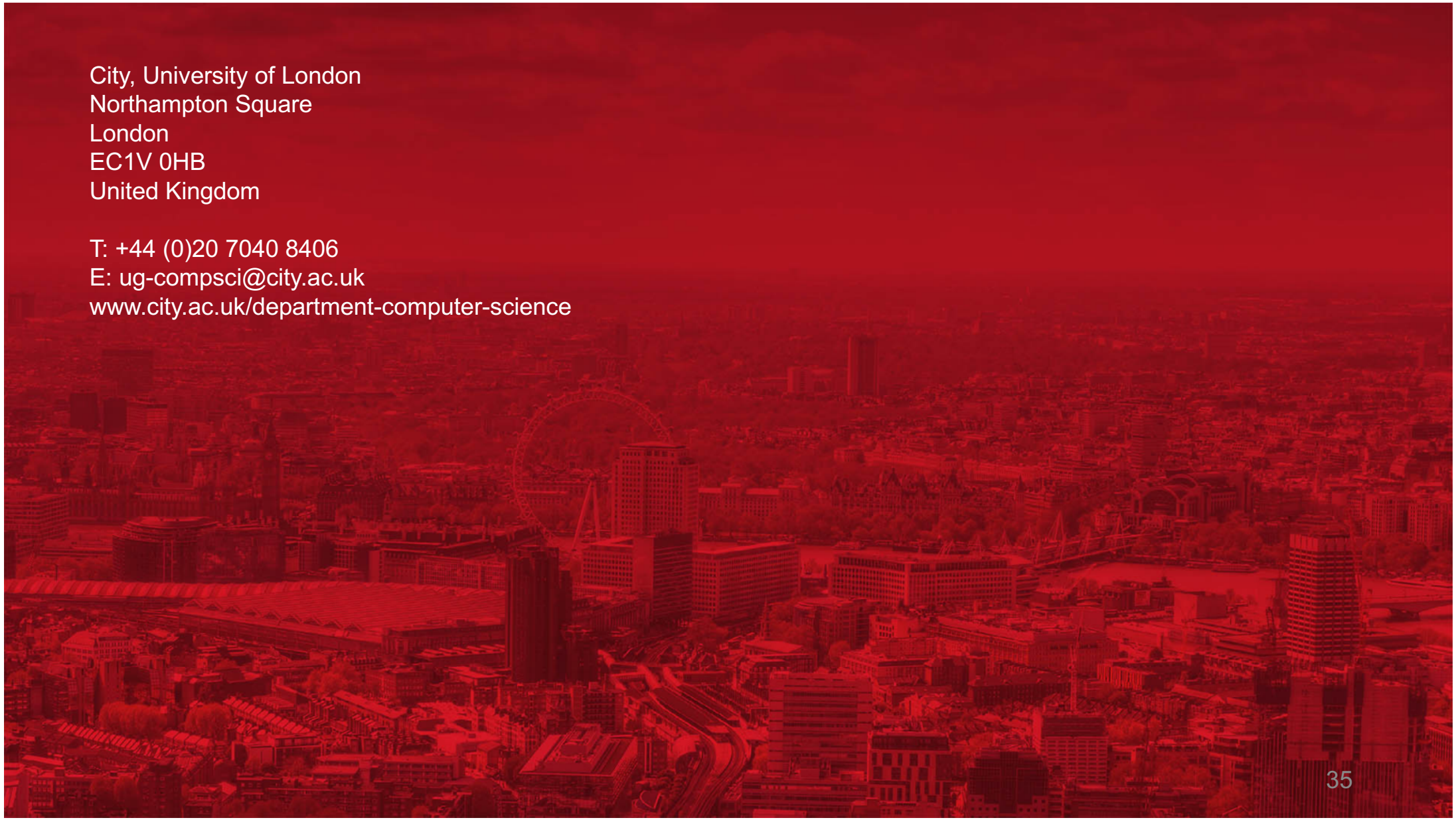
Reading

- Weiss: Chapter 15 (Stacks and Queues), and section 16.1 and 16.2 (Linked Lists)

- Drozdek: Sections 1.4 (Java and Pointers), 3.1 (Singly Linked Lists), 4.1 (Stacks) and 4.2 (Queues)

Next week: Circular lists and Doubly linked lists

Drozdek: sections 3.2 (doubly linked lists) and 3.3 (circular lists) OR Weiss: section 16.3 (doubly linked lists and circular lists)



City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science