Academic excellence for
business and the professions

IN2002 Data Structures and Algorithms

Lecture 5 – Other Linked Lists

Aravin Naren
Semester 1, 2018/19

Learning Objectives

- Understand and be able to use the data structures circular lists and doubly linked lists

- Be able to understand, apply and develop algorithms to handle the lists above. Including:
  - Adding elements
  - Deleting elements
  - Traversing a list

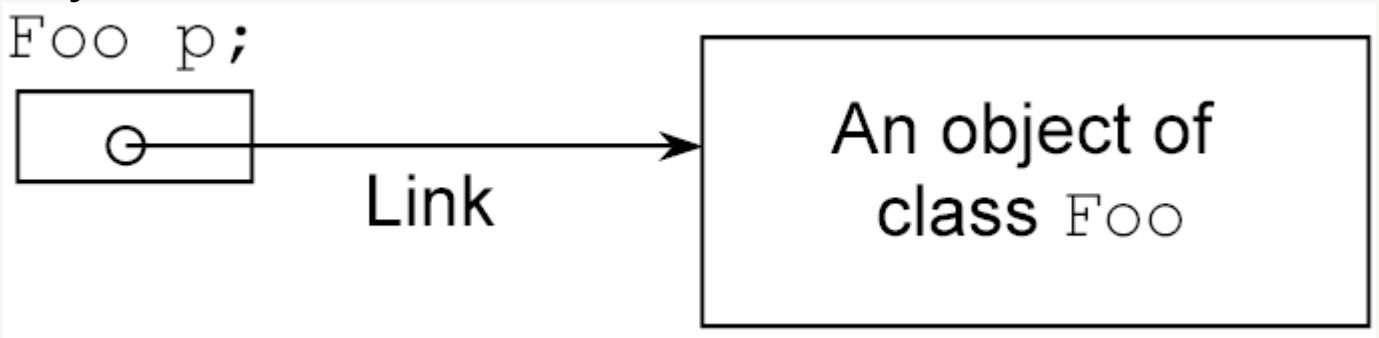
Pointers and Linked Lists

Database Records

All object variables in Java are actually pointers

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

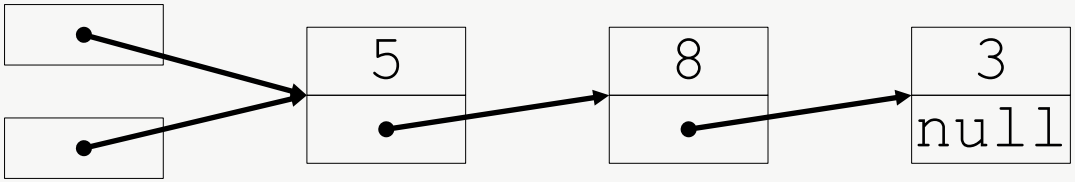
- Links can be thought of as arrows, addresses or references
- They point at an object that is somewhere in the system's memory



Links and their behaviour

Links point at objects

Node p;
Node q;



`p` is

`q.info` is

`p.next` is

`q.next.info` is

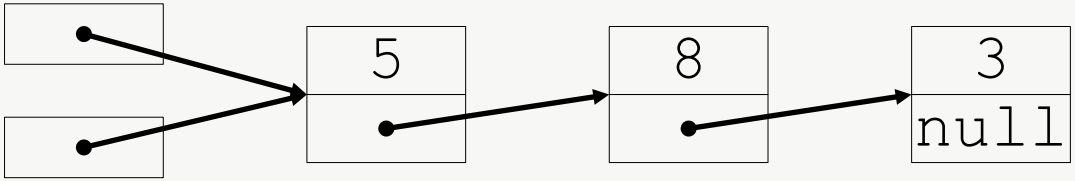
`q.next.next` is

`p.next.next.info` is

Links and their behaviour

Links point at objects

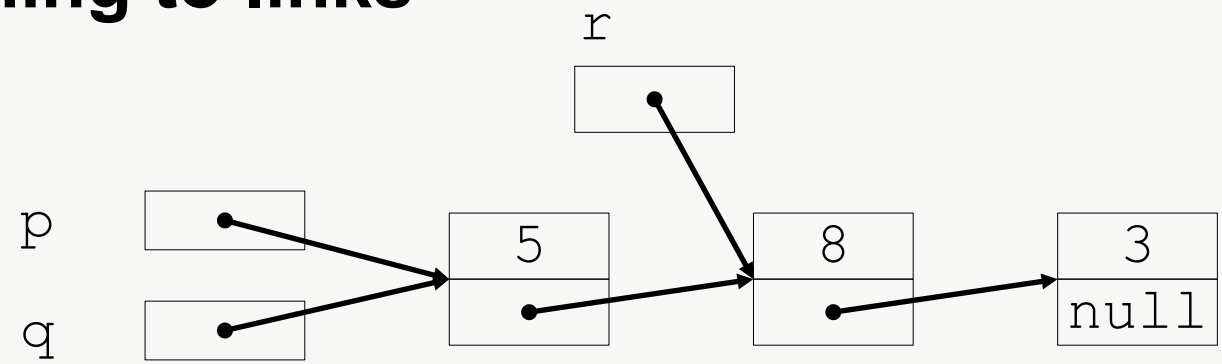
Node p;
Node q;



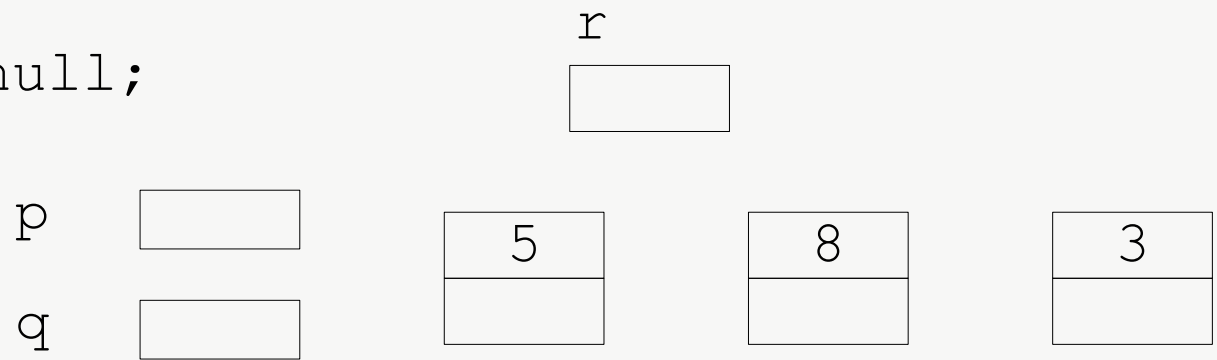

```
p.next = null;
```

```
p = null;
```

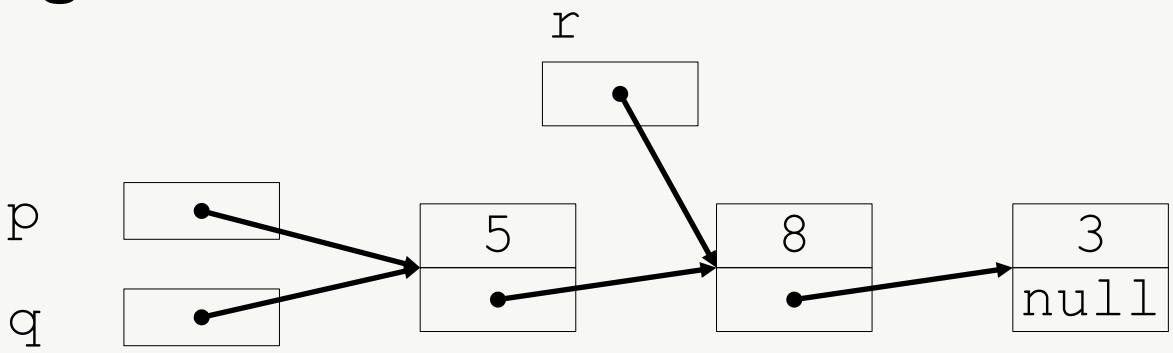
Assigning to links



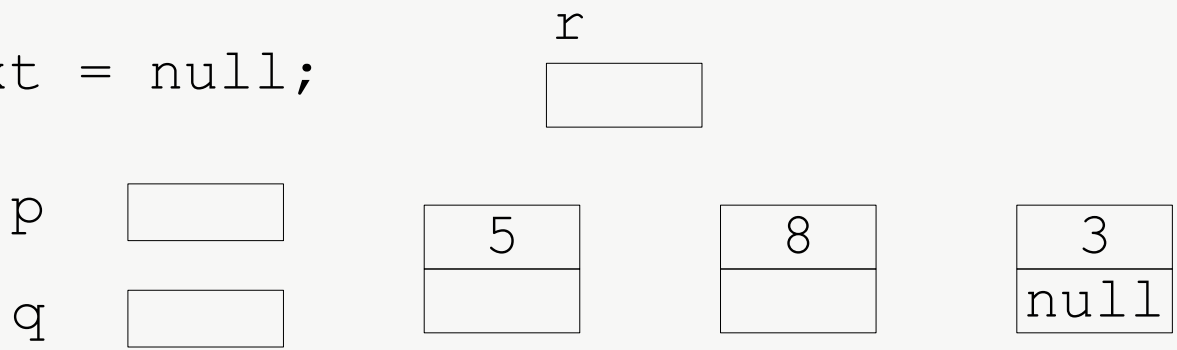

```
p = null;
```



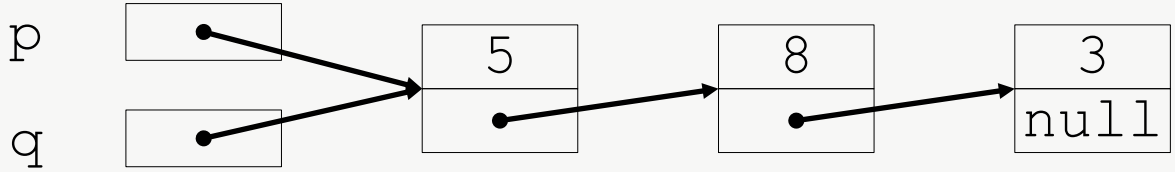
Assigning to links




```
p.next = null;
```

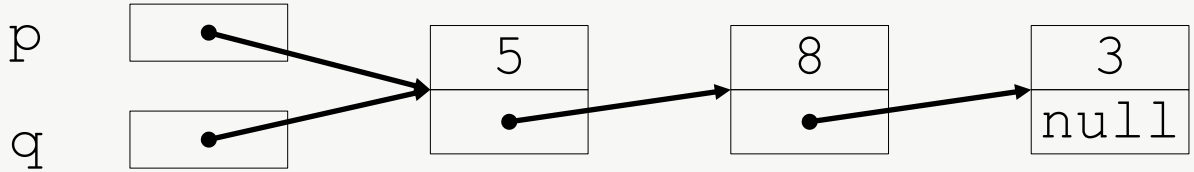


Assigning to links

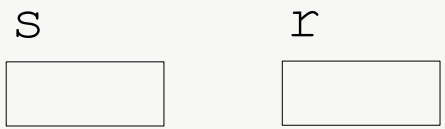



```
p.next = null;
```

Evaluation of links




```
r = q.next.next;
s = p.next;
```



Singly linked lists

Our implementation uses two types of objects:

- *SLList*, which has two components: *head* and *tail*
- *Node*, which has two components: *info* and *next*

head, *tail* and *next* are of type *Node*, so they contain links
info is of the type of information elements you want to store

Traversing a singly linked list

Function printList():
node1 ← head of list
WHILE node1 is not null
print value of node1
node1 ← next of node1

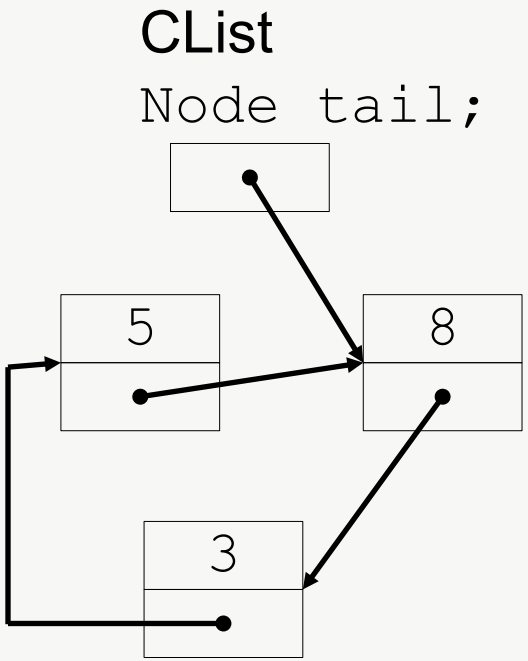
... so to traverse the list, iteratively (or recursively!) evaluate the rest of the list (i.e., use next)

Circular Lists

A Circular List

We use object type *CList*, which has one component: *tail*

The head is *tail.next*



Circular lists in Java

```
public class CList {

    private Node tail = null;

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

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

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

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

Some special cases

An empty list

CList

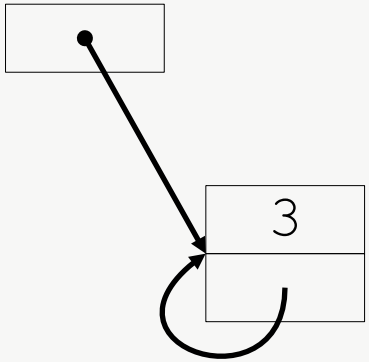
```
Node tail;
```

`null`

A list with one element

CList

```
Node tail;
```



Adding at the head of a circular list

Adding at the head of a circular list… in Java

Remember that `head` is really `tail.next`

```
public void addToHead(int el) {
    if (isEmpty()) {
       // create a new node pointing at itself
       tail = new Node(el);
       tail.next = tail;
    } else { // update the head
       tail.next = new Node(el, tail.next);
    }
}
```

Adding at the tail of a circular list

Adding at the tail of a circular list … in Java

```
public void addToTail(int el) {
    addToHead(el);
    tail = tail.next;
}
```

Deleting the head of a circular list

Deleting the head of a circular list … in Java

```
public int deleteFromHead() {
    int el = tail.next.info;
    if (tail == tail.next)
       tail=null;
    else
       tail.next = tail.next.next;
    return el;
}
```

Traversing a circular list

Traversing a circular list… in Java

Consider that the end of the circular list is not marked by *null*, but by *tail*:

```
public void printList() {
    if (! isEmpty()) {
       Node p = tail;
       do {
          p = p.next;
          System.out.println(p.info);
       } while (p != tail);
    }
}
```

Example Analysis: Traversing a circular list

```
public void printList() {
    if (! isEmpty()) {
        Node p = tail;
        do {
            p = p.next;
            System.out.println(p.info);
        } while (p != tail);
    }
}
```

One loop. Execution starts after tail, and ends at tail. Therefore loop executed once for each element. Running time is $O(n)$.

Only Node p uses memory, none allocated in loop. Memory usage is $O(1)$.

Both apply to best, worst, and average case.

Analysis of circular lists

They perform like singly linked lists:

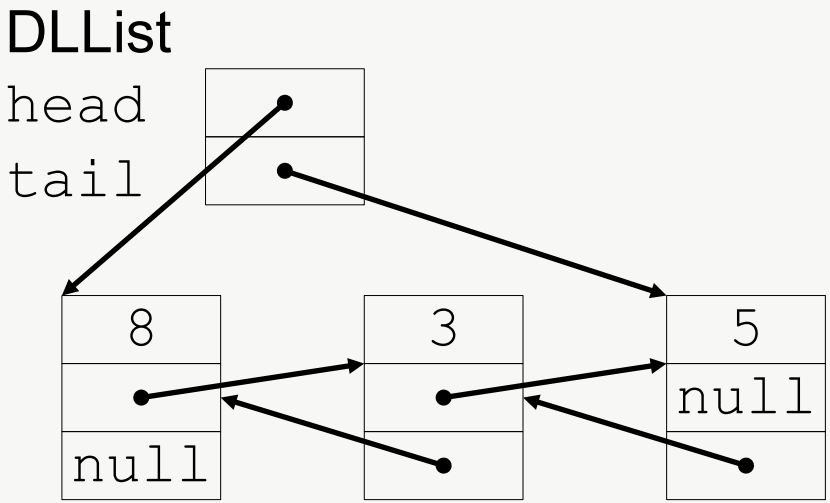
- efficient support of the `Stack` and `Queue` ADTs
- store n elements in O(n) space
- all defined operations take O(1) time
- cannot efficiently delete last element
- can be traversed from head to tail, but not the other way around

Applications: resource sharing, load balancing

Doubly Linked Lists

Doubly Linked Lists

Each node has two pointers: one to its successor and another to its predecessor



A node in a doubly linked list

```
public class DLLNode {

    public int info;
    public DLLNode next, prev;

    public DLLNode(int i, DLLNode n, DLLNode p) {
        info = i; next = n; prev = p;
    }

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

The doubly linked list class

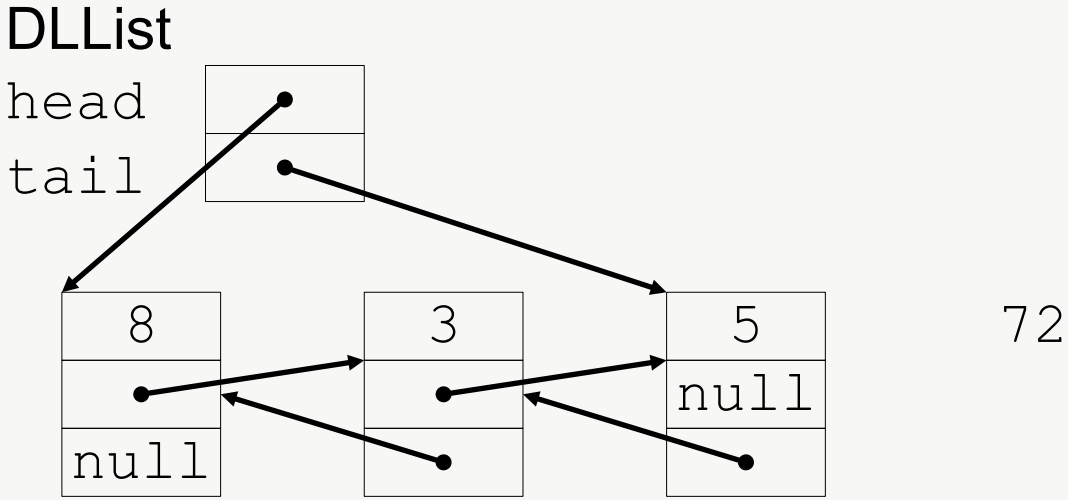
```
public class DLList {
    private DLLNode head = null;
    private DLLNode tail = null;

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

    public void addToHead(int el) {...}
    public void addToTail(int el) {...}
    public int deleteFromHead() {...}
    public int deleteFromTail() {...}
}
```

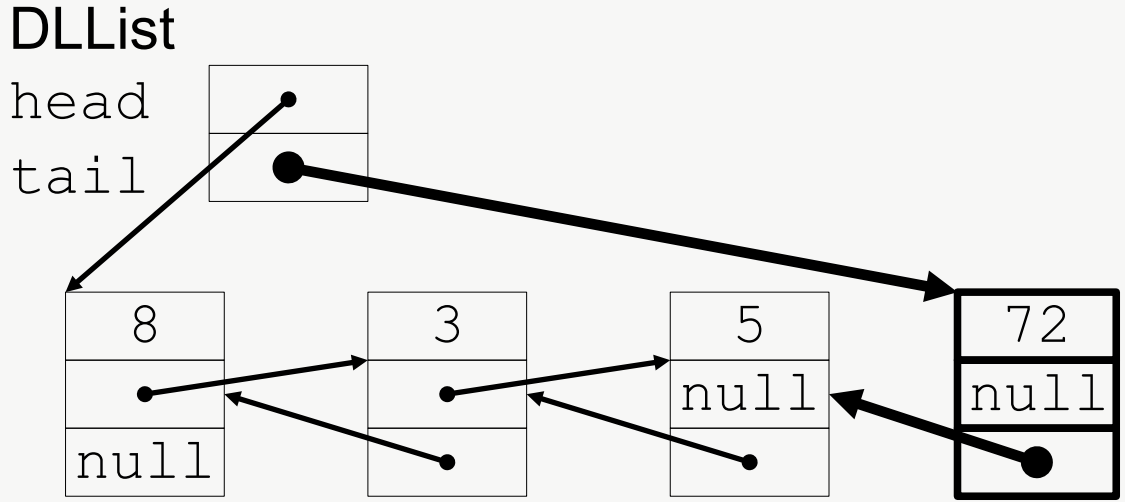
Adding an element at the tail

Suppose we want to add an element `72` to the tail of the list



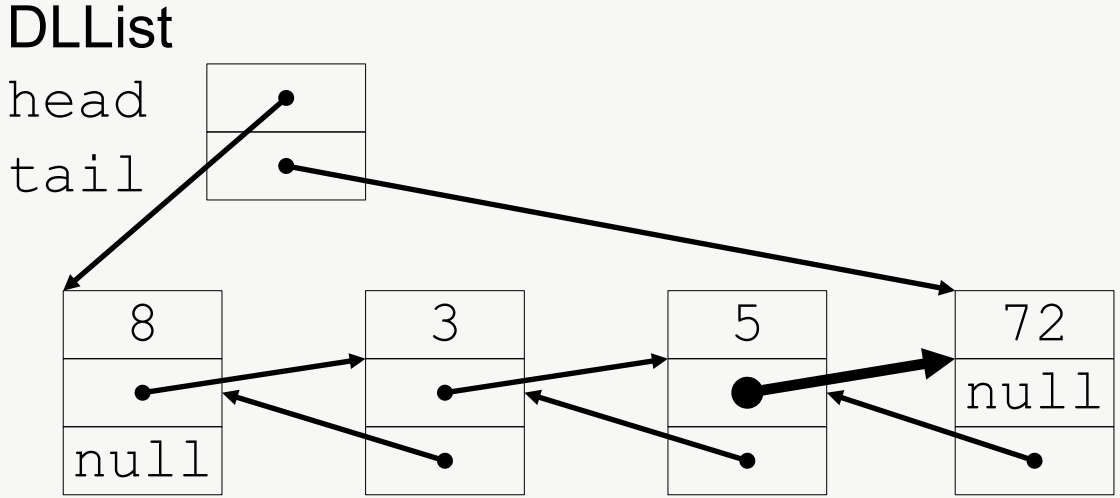
Adding an element at the tail (step 1): create a new node and reset tail

```
tail = new DLLNode(el, null, tail);
```



Adding an element at the tail (step 2): connect the node to the list

```
tail.prev.next = tail;
```

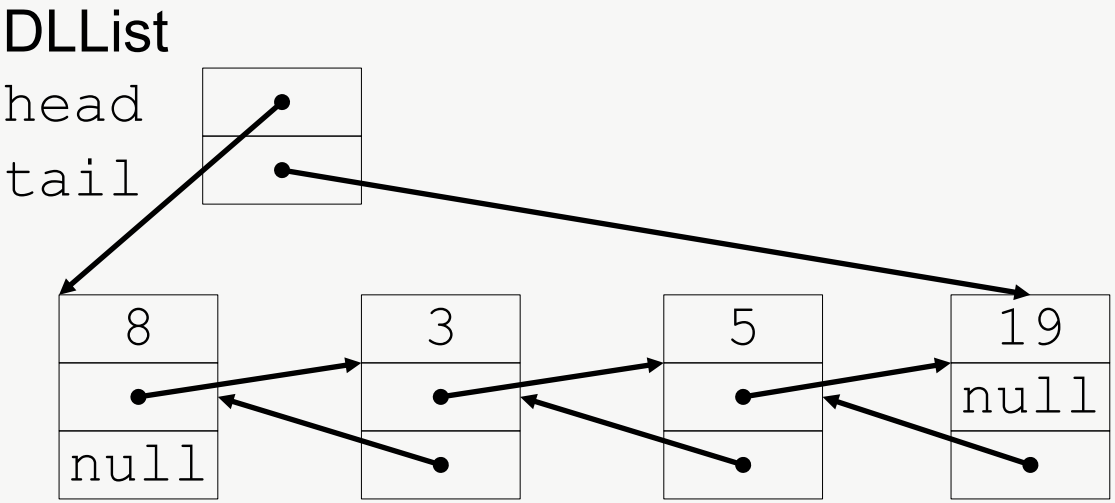


Adding an element at the tail… in Java

If the list is not empty, create and connect the node
Otherwise, just create a new node.

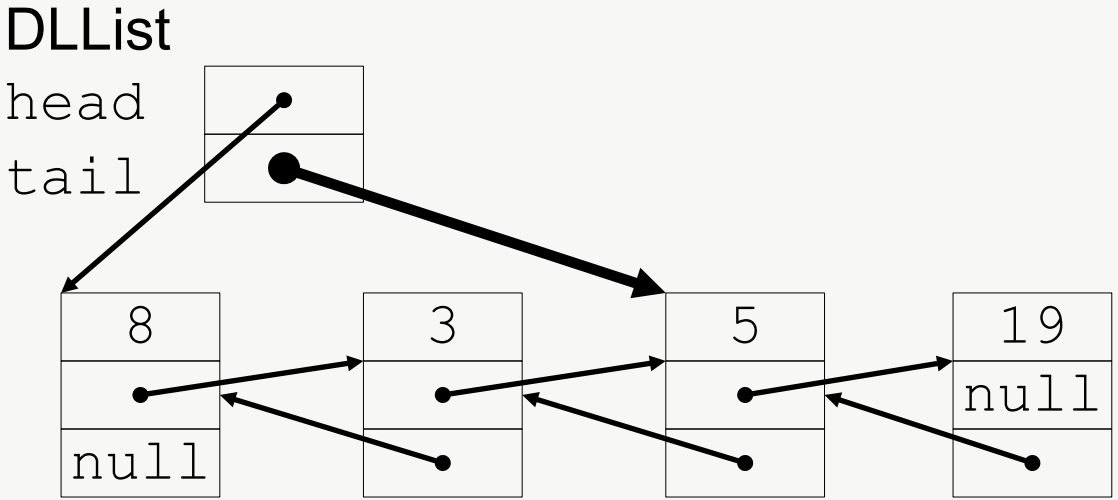
```
public void addToTail(int el) {
    if (! isEmpty()) {
       tail = new DLLNode(el, null, tail);
       tail.prev.next = tail;
    } else
       head = tail = new DLLNode(el);
}
```

Deleting the last node



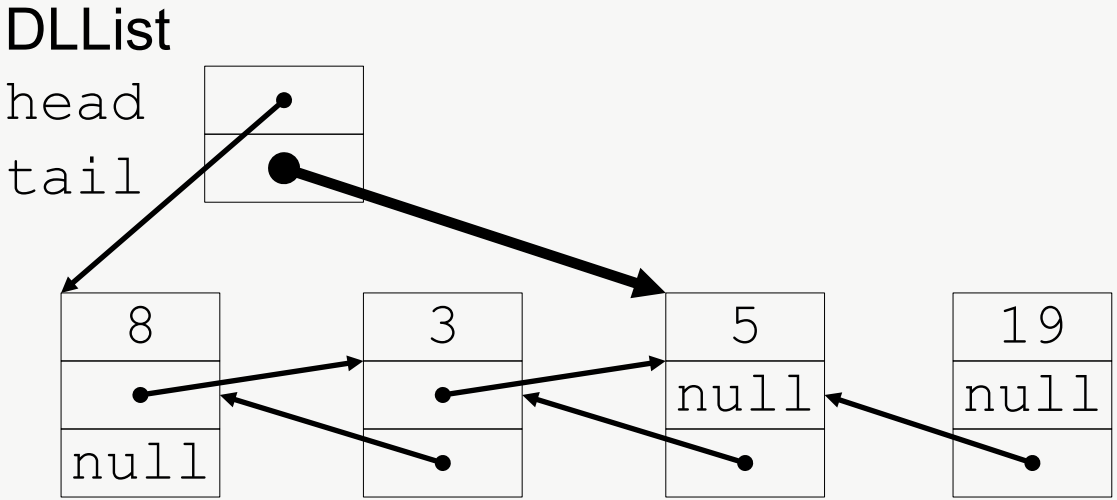
Deleting the last node (step 1): move the tail pointer

```
tail = tail.prev;
```



Deleting the last node (step 2): disconnect the last node

```
tail.next = null;
```



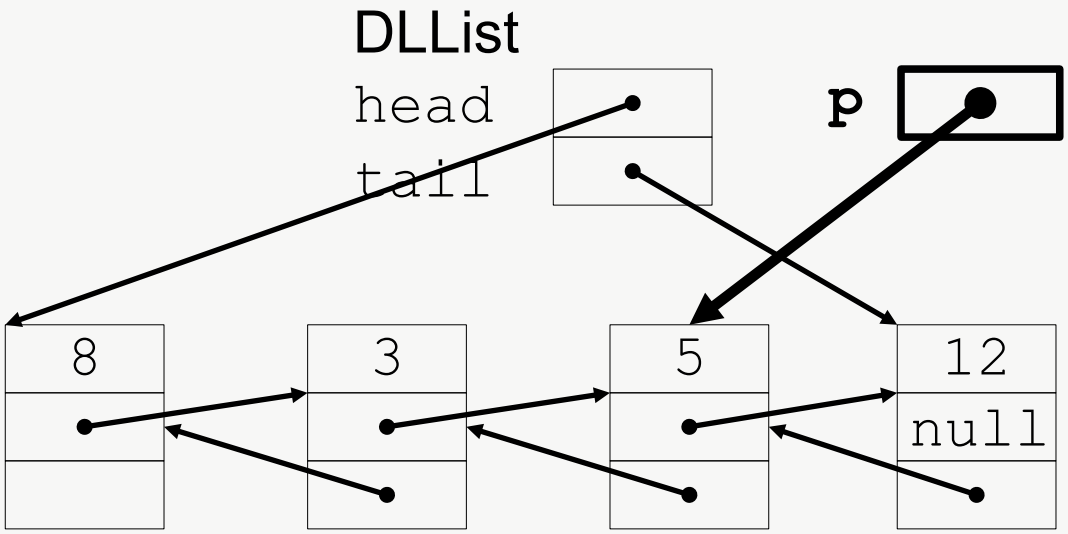
Deleting the last node… in Java

If the list has more than one node, move `tail` and disconnect last node. Otherwise, empty the list

```
public int deleteFromTail() {
    int el = tail.info;
    if (head == tail)
       head = tail = null;
    else {
       tail = tail.prev;
       tail.next = null;
    }
    return el;
}
```

Deleting any node

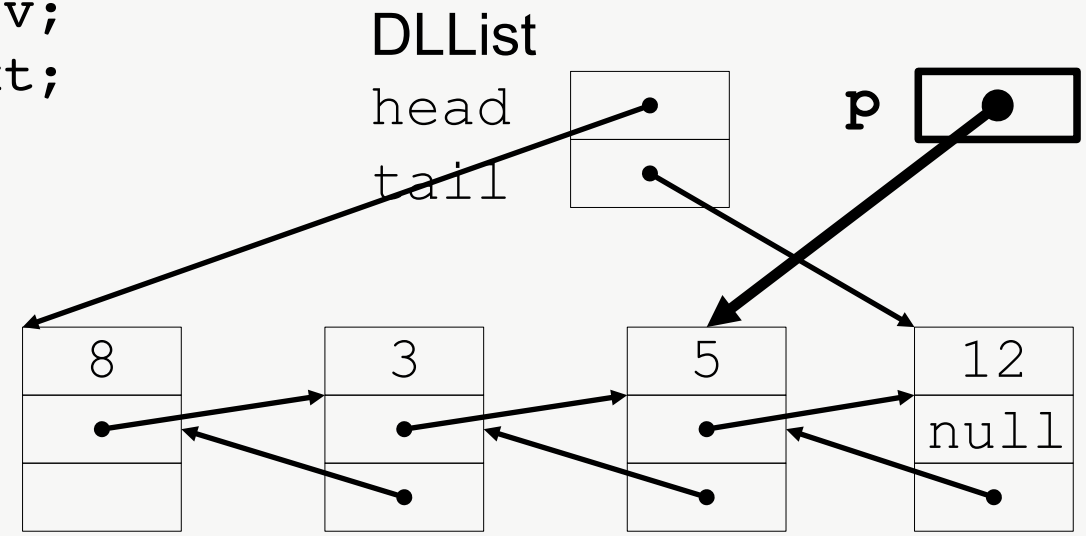
If we wish to delete the node `p` points to



Deleting any node

If we wish to delete the node `p` points to, we need to bridge over it:

```
p.next.prev = p.prev;
p.prev.next = p.next;
```



We may also need to adjust `head` and/or `tail`.

Deleting any node… in Java

```
public void delete(DLLNode p) {
    if (p.prev == null)
       head = p.next;
    else
       p.prev.next = p.next;
    if (p.next == null)
       tail = p.prev;
    else
       p.next.prev = p.prev;
}
```

What happens if it has only one node?

Traversing a doubly linked list… in Java

It is just as easy to go in either direction:

```
public void forwards() {
    for (DLLNode p = head; p != null; p = p.next)
       System.out.println(p.info);
}

public void backwards() {
    for (DLLNode p = tail; p != null; p = p.prev)
       System.out.println(p.info);
}
```

Java Traversal Specialist Classes: Iterators

Iterators encapsulate traversal. In Java defined in `java.util`

```
interface Iterator<E> {

    boolean  hasNext(); // Returns true if the
                        // iteration has more elements.
    E next(); // Returns the next element in the
              // iteration.
    void  remove(); // Removes from the underlying
          // collection the last element returned by
          // the iterator.
}
```

Java Traversal Specialist Classes: ListIterators

ListIterators support backwards traversal and addition.

```
interface ListIterator<E> extends Iterator<E> {
    boolean  hasPrevious(); // Returns true if the
       // iterator has further elements when traversing
       // in reverse direction.
    E previous(); // Returns the previous element in
                  // the list.
    void add(E e); // Inserts the specified element
                   // into the list.
    ... // more methods
}
```

Reading

- Weiss: Section 16.3 (doubly linked lists and circular lists)

- Drozdek: Sections 3.2 (doubly linked lists) and 3.3 (circular lists)

Next session (IN 2 WEEKS TIME): Hash tables

Drozdek: Chapter 10 (Hashing) OR Weiss: Chapter 19 (Hash tables)

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