# Lists (Chapter 3)

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

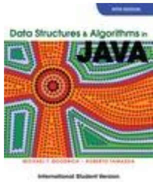
# Introduction

- Rank is not the only way of referring to the place where an element appears in a sequence.
- Problems with Arrays: Fixed size of N & use of integer indices to access its contents.

- If we were to use a linked list to implement a sequence, then it would be more natural to use elements (nodes) to describe the place where an element appears.

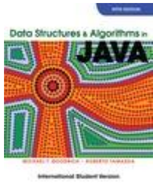
- For example:
  - X should be inserted after Y.
  - Replace the element at Z with e.
  - Delete the element at the node before X.

- A Linked List is a collection of **nodes** that together form a linear ordering. Each node stores a reference to an element and a reference, called (next) to another node.



# Introduction

- When we talk about “place”, we can do so in two ways:
  - 1: Insert “Arsenal” after “Liverpool”.
  - 2: Insert “Arsenal” at the position immediately after the position in which “Liverpool” is stored.
- The key difference here, is the idea of **position**.
  - A position is a place in the list that a piece of data is stored.
- In the first case, position is informal and relative to a key piece of known data.
- In the second case, position is explicit and not based directly on the associated data.
  - The data is stored in the position, and is not the position itself…



# Why all the fuss?

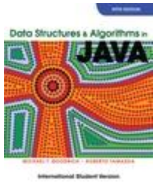
- The way in which we view "place" affects how we think about the concept of a List.
  - If we view things in terms of key values in the sequence, then we must first find the place in which that key value is stored.
  - If we view things in terms of positions, then we can design our ADT to work independent of values.

- Actually, we still need to find the position of interest, but this not of direct concern to the definition of the ADT.

- Decoupling the concept from the data is often a better solution, so we will model Lists in term of position.
  - To do this, we first need to define the concept of a position.

# Singly Linked Lists

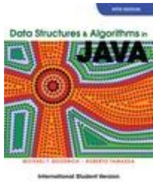
- A Linked List is a collection of **nodes** that together form a linear ordering. Each node stores a reference to an element and a reference, called *next* to another node.
- The *next* reference inside a node can be viewed as a **link** or **pointer** to another node.
- Moving from one node to the another by following the *next* reference is known as **link hopping** or **pointer hopping.**
- The first and last node of a linked list are called the **head** and **tail** of the list.
- We can identify the tail as a node having a null *next* reference, which indicates the end of the list. A linked list defined this way is known as **singly linked list.**

# Node List ADT

- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()

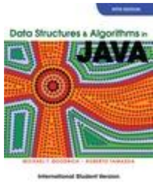
Accessor methods:
  - first(), last()
  - prev(p), next(p)
- Update methods:
  - set(p, e)
  - addBefore(p, e), addAfter(p, e),
  - addFirst(e), addLast(e)
  - remove(p)



# Implementing a Singly Linked List

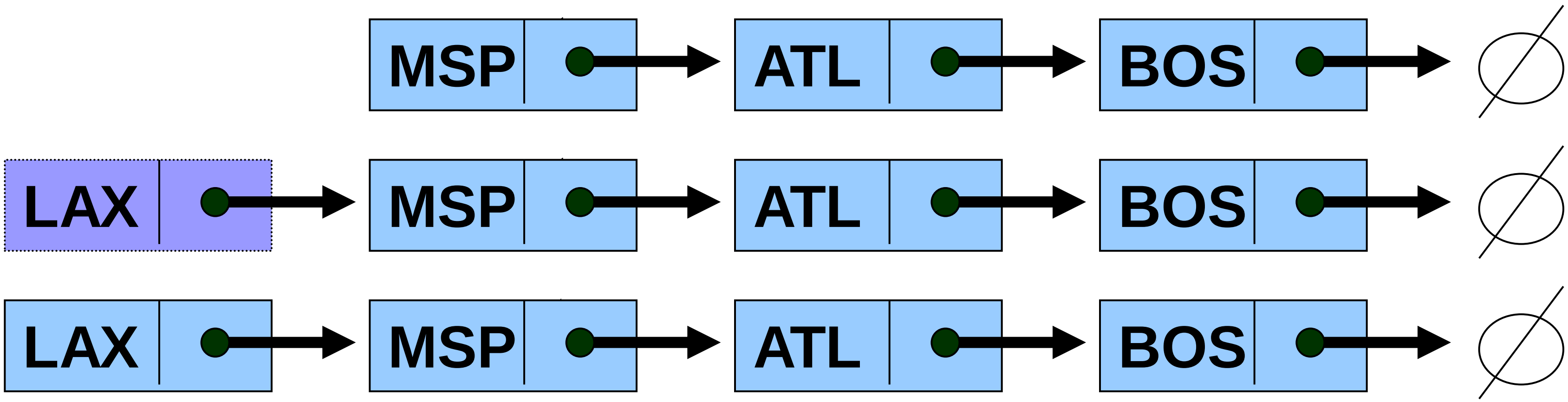
```
/** Node of a singly linked list of
    strings. */
public class Node { private String element;
// we assume elements are character strings
    private Node next;
    /** Creates a node with the given
    element and next node. */
    public Node(String s, Node n) {
          element = s; next = n; }
    /** Returns the element of this node.
    */
    public String getElement() { return
    element; }
    /** Returns the next node of this node.
    */
    public Node getNext() { return next; }
    // Modifier methods:
    /** Sets the element of this node. */
    public void setElement(String newElem)
    {
          element = newElem; }
    /** Sets the next node of this node. */
    public void setNext(Node newNext) {
    next = newNext; } }
```

```
/** Singly linked list .*/
public class SLinkedList {
    protected Node head;
    // head node of the list
    protected long size;
    // number of nodes in the list
    /** Default constructor that creates an
    empty list */
    public SLinkedList() { head = null;
    size = 0; }
     // ... update and search methods would
    go here ... }
```



# Insertion in a Singly Linked List

- Create a new node, set its *next* link to refer to the same object as *head* and then set head to point to the new node.

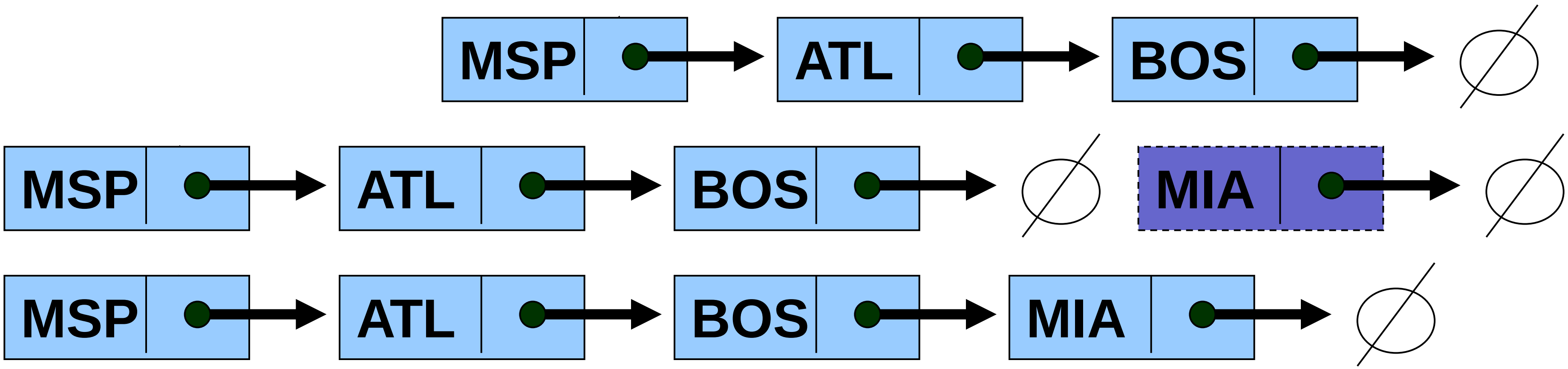



**Algorithm**: addFirst(v):

| | |
|---|---|
| **v.setNext(head)** | { make v point to the old head node} |
| head $\leftarrow$ v | {make variable head point to new node} |
| size $\leftarrow$ size + 1 | {increment the node count} |

# Insertion in a Singly Linked List

- Create a new node, set its *next* link to refer to the same object as *tail* and then set tail to point to the null object

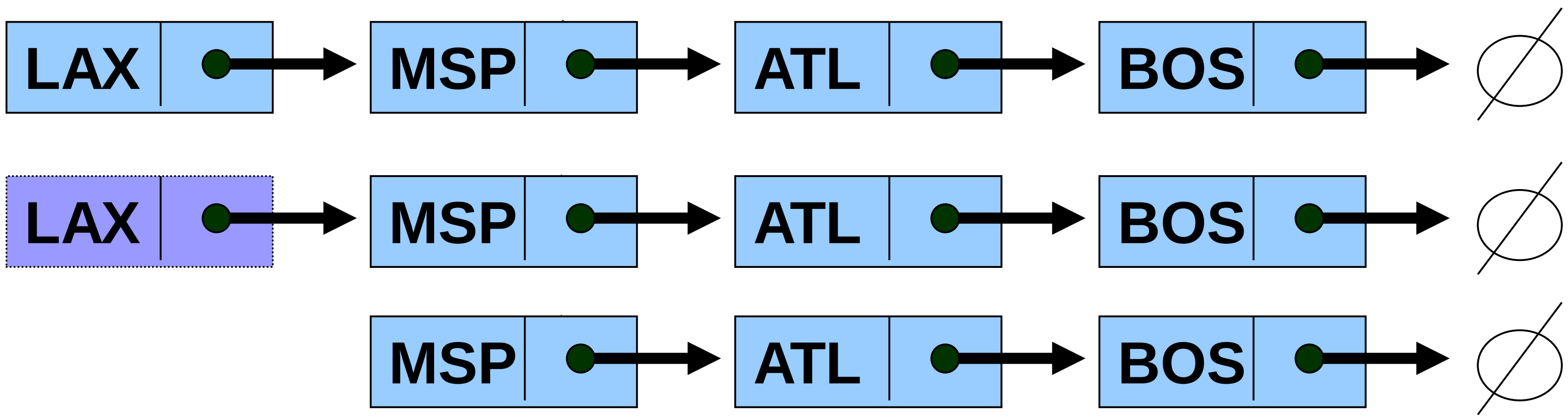



**Algorithm**: addLast(v):

| | |
|---|---|
| **v.setNext(null)** | { make new node v point to null object} |
| tail.setNext(v) | {make old tail node point to the new node} |
| tail ← v | {make variable tail point to new node} |
| size ← size + 1 | {increment the node count} |

# Removing an element in a Singly Linked List

- Removing an element at the head is the reverse operation of inserting an element at the head.
- Unfortunately we cannot easily delete the tail node of a singly linked list. It would take a long time to access the node just before the tail from the head of the list and search all the way through the list.




**Algorithm**: removeFirst(v):

If head = **null then**

    indicate an error: the list is empty

t ← head

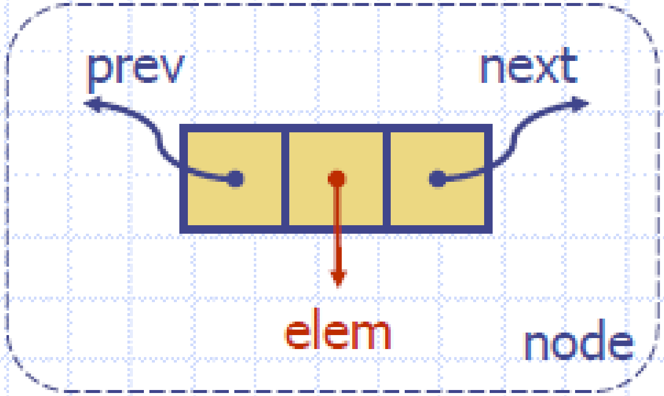
head ← head.getNext()    {make head point to next node or null}

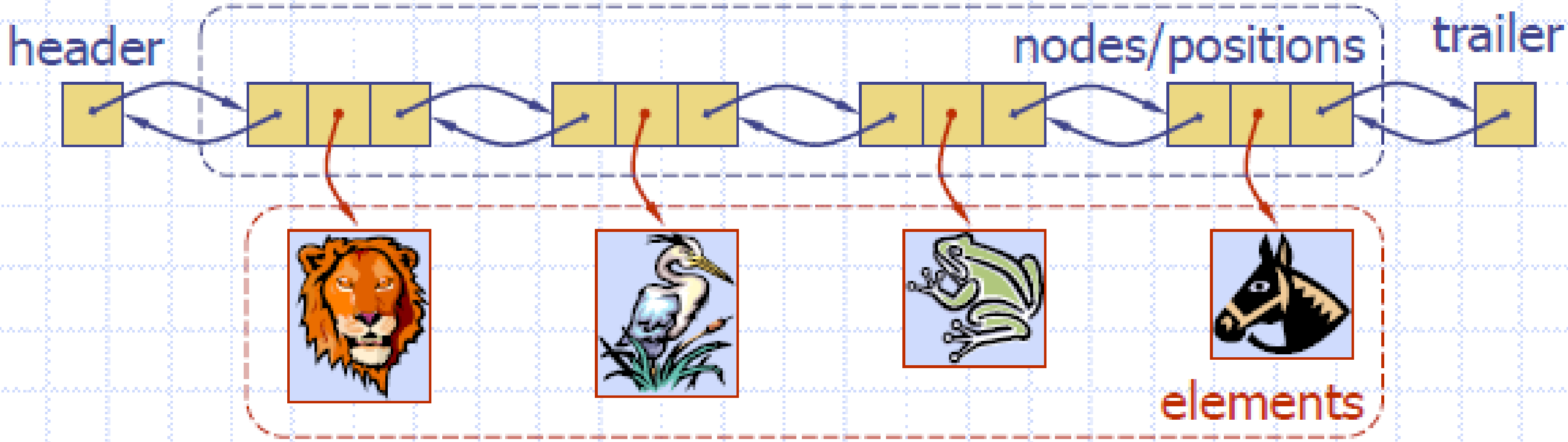
t.setNext(null)    {null out the next pointer of the removed node}

size ← size - 1    {decrement the node count}

# Doubly Linked List

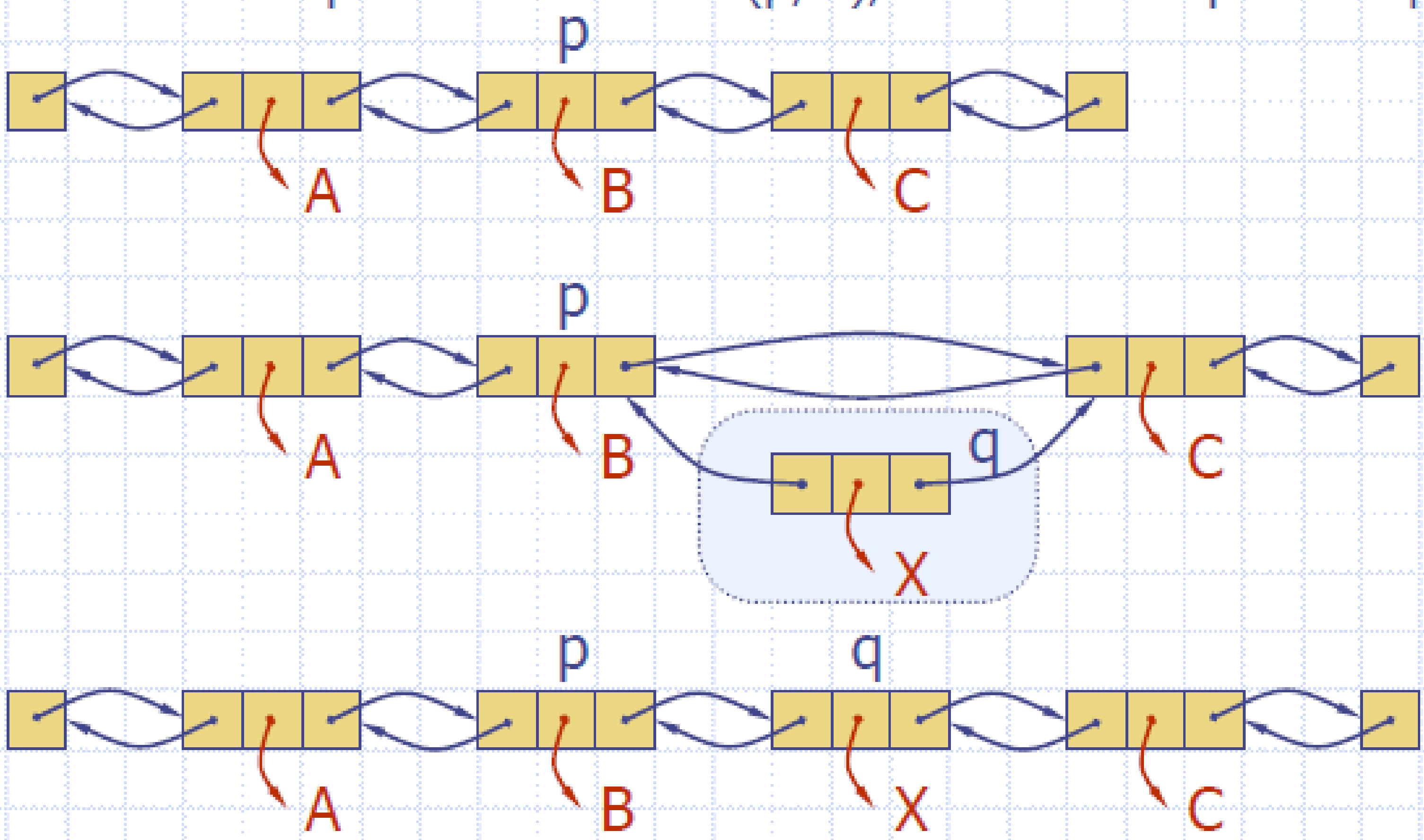
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



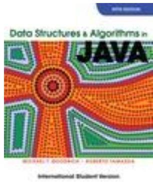

# Insertion

- We visualize operation insertAfter(p, X), which returns position q



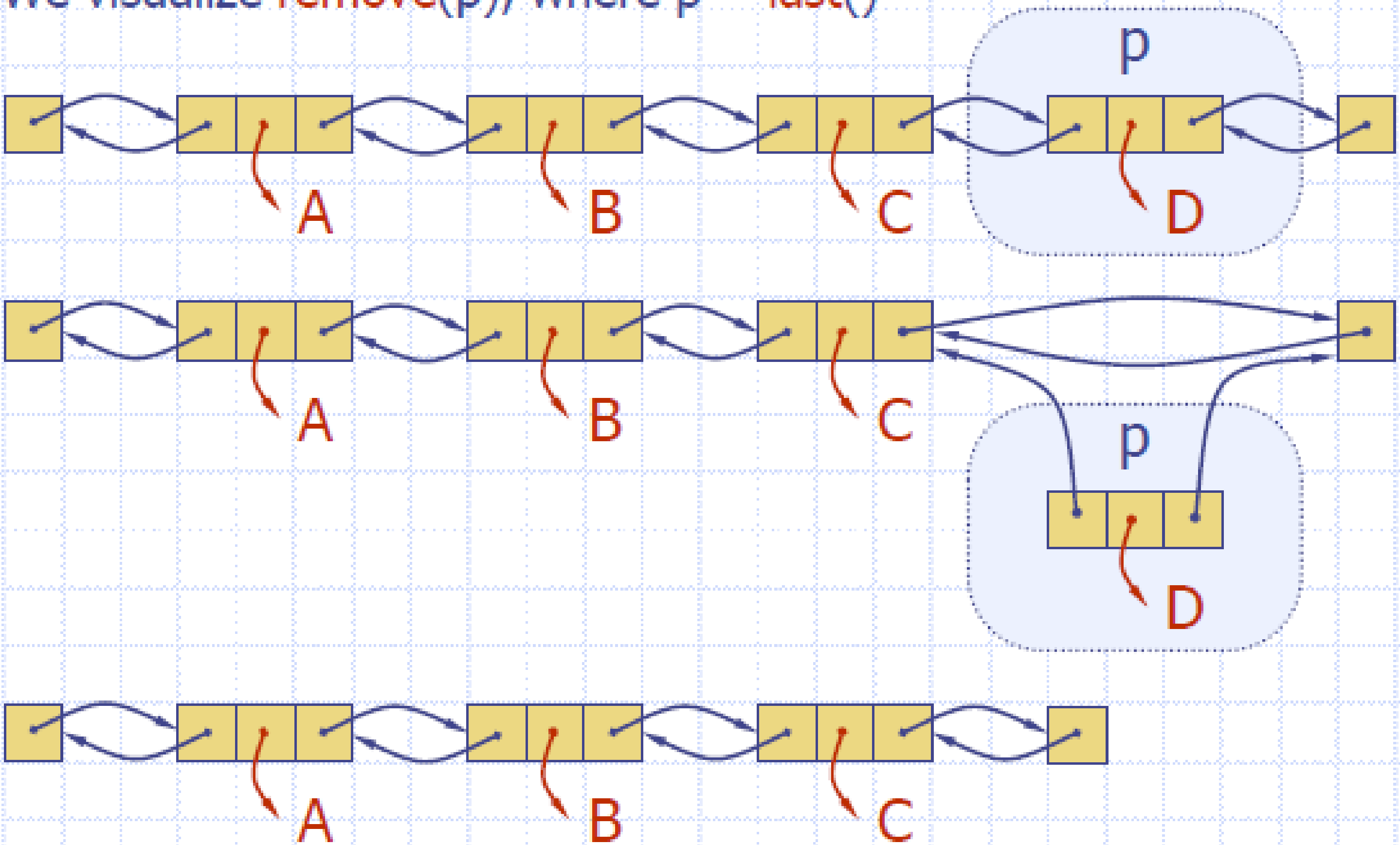
# Insertion Algorithm

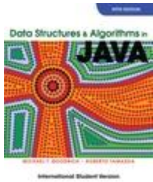
```
Algorithm addAfter(p,e):
    Create a new node v
    v.setElement(e)
    v.setPrev(p)          {link v to its predecessor}
    v.setNext(p.getNext())   {link v to its successor}
    (p.getNext()).setPrev(v) {link p's old successor to v}
    p.setNext(v)          {link p to its new successor, v}
    return v   {the position for the element e}
```



# Deletion

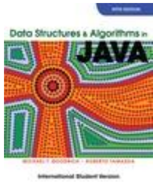
- We visualize remove(p), where p = last()



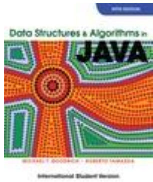
# Deletion Algorithm

```
Algorithm remove(p):
    t = p.element          {a temporary variable to hold the
                           return value}
    (p.getPrev()).setNext(p.getNext())     {linking out p}
    (p.getNext()).setPrev(p.getPrev())
    p.setPrev(null)        {invalidating the position p}
    p.setNext(null)
    return t
```



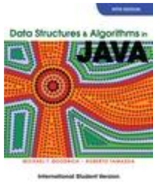
# Doubly Linked Lists

```
/** Doubly linked list with nodes of type DNode storing strings. */
public class DList {
  protected int size;                   // number of elements
  protected DNode header, trailer;      // sentinels
  /** Constructor that creates an empty list */
  public DList() {
    size = 0;
    header = new DNode(null, null, null);       // create header
    trailer = new DNode(null, header, null);    // create trailer
    header.setNext(trailer);    // make header and trailer point to each other
  }
  /** Returns the number of elements in the list */
  public int size() { return size; }
  /** Returns whether the list is empty */
  public boolean isEmpty() { return (size == 0); }
  /** Returns the first node of the list */
  public DNode getFirst() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return header.getNext();
  }
  /** Returns the last node of the list */
  public DNode getLast() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return trailer.getPrev();
  }
  /** Returns the node before the given node v. An error occurs if v
    * is the header */
  public DNode getPrev(DNode v) throws IllegalArgumentException {
    if (v == header) throw new IllegalArgumentException
      ("Cannot move back past the header of the list");
    return v.getPrev();
  }
  /** Returns the node after the given node v. An error occurs if v
    * is the trailer */
  public DNode getNext(DNode v) throws IllegalArgumentException {
    if (v == trailer) throw new IllegalArgumentException
      ("Cannot move forward past the trailer of the list");
   return v.getNext();
  }
```



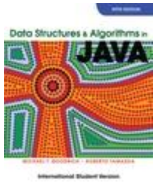
```java
/** Inserts the given node z before the given node v. An error
  * occurs if v is the header */
public void addBefore(DNode v, DNode z) throws IllegalArgumentException {
  DNode u = getPrev(v);      // may throw an IllegalArgumentException
  z.setPrev(u);
  z.setNext(v);
  v.setPrev(z);
  u.setNext(z);
  size++;
}
/** Inserts the given node z after the given node v. An error occurs
  * if v is the trailer */
public void addAfter(DNode v, DNode z) {
  DNode w = getNext(v);      // may throw an IllegalArgumentException
  z.setPrev(v);
  z.setNext(w);
  w.setPrev(z);
  v.setNext(z);
  size++;
}
/** Inserts the given node at the head of the list */
public void addFirst(DNode v) {
  addAfter(header, v);
}
/** Inserts the given node at the tail of the list */
public void addLast(DNode v) {
  addBefore(trailer, v);
}
/** Removes the given node v from the list. An error occurs if v is
  * the header or trailer */
public void remove(DNode v) {
  DNode u = getPrev(v);      // may throw an IllegalArgumentException
  DNode w = getNext(v);      // may throw an IllegalArgumentException
  // unlink the node from the list
  w.setPrev(u);
  u.setNext(w);
  v.setPrev(null);
  v.setNext(null);
  size--;
}
```

```java
/** Returns whether a given node has a previous node */
public boolean hasPrev(DNode v) { return v != header; }
/** Returns whether a given node has a next node */
public boolean hasNext(DNode v) { return v != trailer; }
/** Returns a string representation of the list */
public String toString() {
  String s = "[";
  DNode v = header.getNext();
  while (v != trailer) {
    s += v.getElement();
    v = v.getNext();
    if (v != trailer)
      s += ",";
  }
  s += "]";
  return s;
}
}
```



# Insertion Sort for Doubly Linked List

```
/** Insertion-sort for a doubly linked list of class DList.  */
public static void sort(DList L) {
  if (L.size() <= 1) return; // L is already sorted in this case
  DNode pivot;         // pivot node
  DNode ins;           // insertion point
  DNode end = L.getFirst();    // end of run
  while (end != L.getLast()) {
    pivot = end.getNext();     // get the next pivot node
    L.remove(pivot);           // remove it
    ins = end;                 // start searching from the end of the sorted run
    while (L.hasPrev(ins) &&
           ins.getElement().compareTo(pivot.getElement()) > 0)
      ins = ins.getPrev();     // move left
    L.addAfter(ins,pivot);     // add the pivot back, after insertion point
    if (ins == end)            // we just added pivot after end in this case
      end = end.getNext();     // so increment the end marker
  }
}
```



# Positions

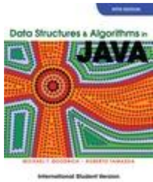
- Concept: A position is a place in the list that holds a value.
  - It is an auxiliary ADT for ADTS in which the values are stored at positions.

- Operations:
  - **element()**: Return the element stored at this position.

- Interface:

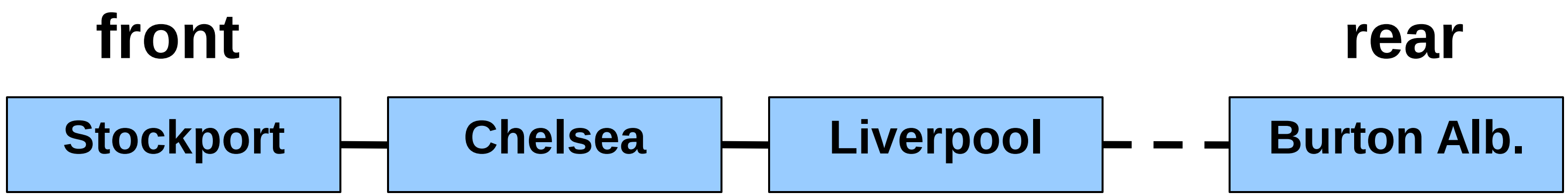
```
public interface Position {
   public Object element();
}
```

- Implementation: This depends on the primary ADT

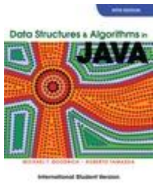


# Lists: Concept

- A List supports insertion and removal of objects based on **position**.
  - Insertion is carried out relative to a position or a known fixed point.
  - E.g. insert “Liverpool” after “Chelsea” / insert “Stockport County” at the front.
- Example:




- Lists are similar to Linked Lists:
  - As a concept, we say nothing about links / nodes



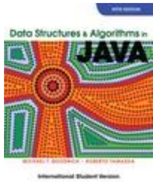
# Lists: Function Specification

- Core Operations:
  - replace(p,e): Replace the element at position p with e, returning the element formerly at p.
  - insertFirst(e): Insert a new element e into S as the first element and return the position of e.
  - insertLast(e): Insert a new element e into S as the last element and return the position of e.
  - insertBefore(p,e): Insert a new element e into S before position p and return the position of e.
  - insertAfter(p,e): Insert a new element e into S after position p and return the position of e.
  - remove(p): Remove from S the element at position p.

- Support Operations:
  - isEmpty() Returns true if the vector is empty, or false otherwise
  - size() Returns the number of elements in the vector

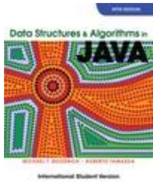
# Lists: Function Specification

- Vector traversal is easy: objects are stored sequentially based on rank.
- List traversal is more difficult: everything is relative to a position.
- Traversal Operations:
  - first(): Return the position of the first element of S; a list empty error occurs if S is empty.
  - last(): Return the position of the last element of S; a list empty error occurs if S is empty.
  - prev(p): Return the position of the element of S preceding the one at p; an boundary violation error occurs if p is the first position.
  - next(p): Return the position of the element of S following the one at p; an boundary violation error occurs if p is the last position.



# Lists: Java Interface

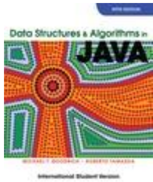
```
public interface List {
    public int size();
    public boolean isEmpty();
    public Position first() throws ListEmptyException;
    public Position last() throws ListEmptyException;
    public Position prev(Position p)
                throws BoundaryViolationException;
    public Position next(Position p)
                throws BoundaryViolationException;
    public Position insertFirst(Object e);
    public Position insertLast(Object e);
    public Position insertBefore(Position p, Object e);
    public Position insertAfter(Position p, Object e);
    public Object replace(Position p, Object e);
    public Object remove(Position p);
}
```



# Lists: Impl. Strategies

- Array-based Implementation:
  - Objects stored in an array of positions
    - Each position includes the index it is stored at in the array
    - Without this, we must find where the position is in the array
    - The index at which a position is stored changes with insertion and removal
  - Keep track of the current size of the list

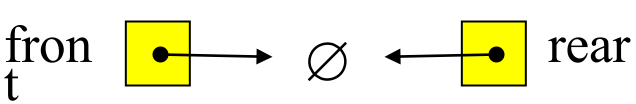
- Link-based Implementation:
  - Nodes are “positions”
  - Nodes maintain ordering information
    - Link to the next and previous objects in the Vector.
  - Auxiliary references maintained “front” and “rear” nodes.
  - Keep track of the current size of the list



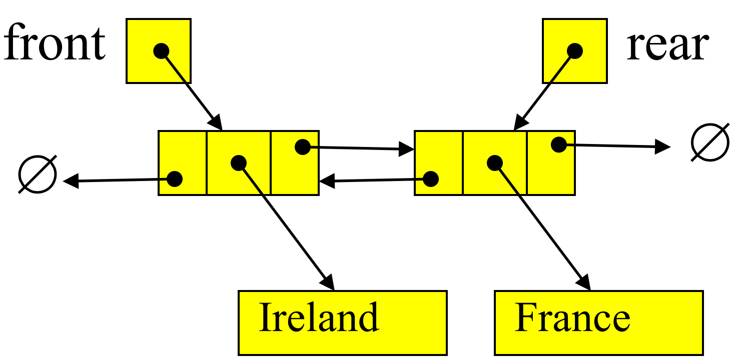
# Link-based Implementation

- Approach: Use a doubly linked list
  - A Node is basically a “position”
  - Finding adjacent nodes is very fast O(1) and simple to do
  - Key positions can be easily identified (front, rear)
  - Need to keep track of the size (size)

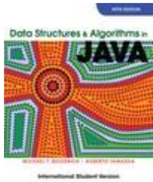
- Dry Run Format:




*Empty List*



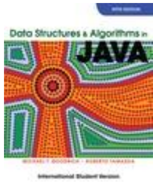

*List of size 2*



# Node Implementation

- Implemented as an inner class within the LinkedList class

```
private class Node implements Position {
  Object element;
  Node next;
  Node prev;

  public Node(Object element) {
   this.element = element;
  }

  public Object element() {
   return element;
  }
}
```



# Link-based Lists: Pseudo Code

```
Algorithm insertFirst(e):
    node ← new Node(e)
    node.next ← first
    if (first == null) then
      last ← node
    else
      first.prev ← node
    first ← node
    size ← size + 1
    return node
```

```
Algorithm: first():
    if (front = null) then
      throw a ListEmptyException
    return front
```

```
Algorithm size():
    return size
```

```
Algorithm insertLast(e):
    node ← new Node(e)
    node.prev ← last
    if (last == null) then
      first ← node
    else
      last.next ← node
    last ← node
    size ← size + 1
    return node
```

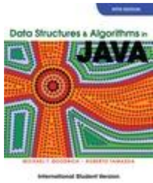
```
Algorithm: last():
    if (front = null) then
      throw a ListEmptyException
    return rear
```

```
Algorithm isEmpty():
    return size = 0
```

# Insert: Writing Pseudo Code

- Developing pseudo code can be tricky:
  - operations can work differently depending on the state of the data structure
  - one solution may not work in all cases

- An approach:
  - Identify Potential Use Cases
    - Describe different scenarios in which the operation is performed (e.g. empty list; non-empty list)
  - Use diagrams to understand what should happen in each case
    - Start with before and after cases, and try to work out the intermediary steps
  - Write pseudo code for each case
  - Integrate the code into a single algorithm

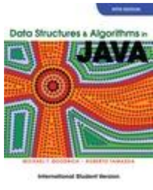


# Operation: insertAfter(p, e)

- Basic Idea:
  - Insert the data, e, so that it will be stored in the position in the list that is previous to position p.

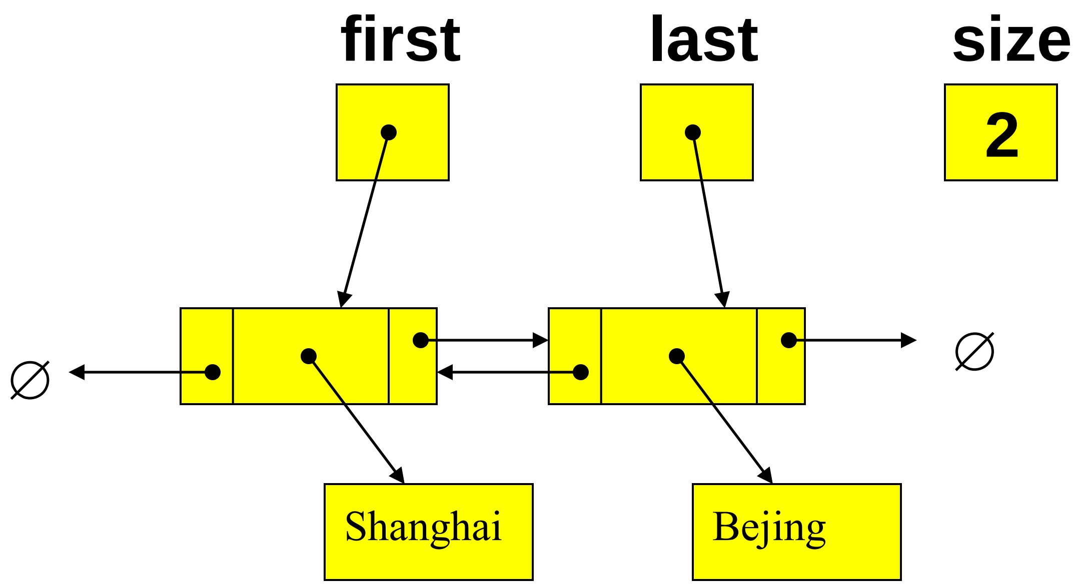
- Process (Informal):
  - Create a new node
  - Update the links so that the node is inserted relative to p
  - Update the size

- Potential Use Cases:
  1. Inserting after the last position in the list (already done).
  2. Inserting at any position in the list.



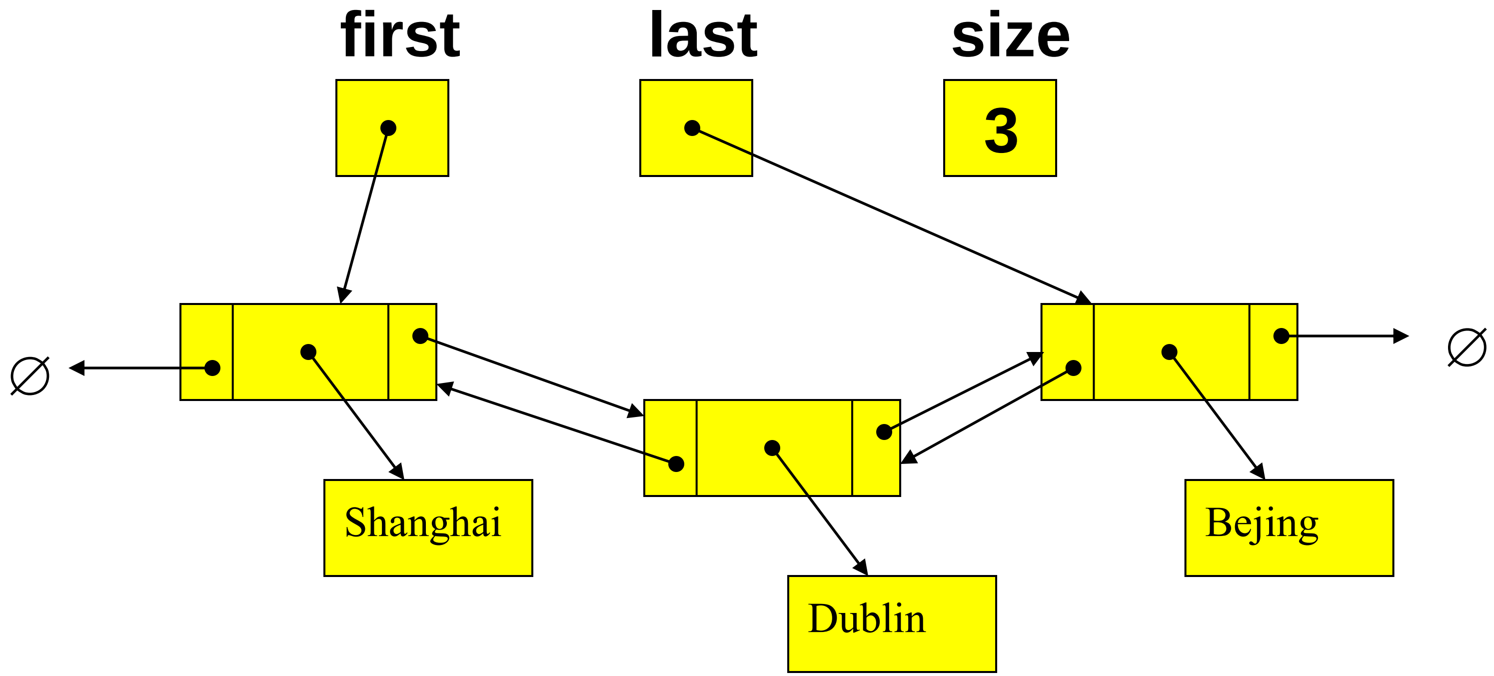
# Operation: insertAfter(p, e)

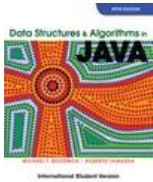
- Use Case 2: Insert after position p, where p is not the last position.
  - For example: Insert "Dublin" after the position containing "Shanghai"



# Operation: insertAfter(p, e)

- Use Case 2: Insert after position p, where p is not the last position.
  - For example: Insert “Dublin” after the position containing “Shanghai”



# Operation: insertAfter(p, e)

**Algorithm**: InsertAfter(p, e):

n ← new Node(e)

n.next ← p.next

n.prev ← p

p.next.prev ← n

p.next ← n

size ← size + 1

**return** n

# Operation: insertAfter(p, e)

**Algorithm**: InsertAfter(p, e):

***if** (p = last) **then** insertLast(e)*

n ← new Node(e)

n.next ← p.next

n.prev ← p

p.next.prev ← n

p.next ← n

size ← size + 1

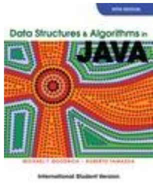
**return** n

# Operation: remove(p)

- Basic Idea:
  - Removes the element at position p from the List

- Process (Informal):
  - Modify the links to remove the node from the list.
  - Reduce the size
  - Return the value stored in the node

- Potential Use Cases:
  1. Removal of the element at the first position
  2. Removal of the element at the last position
  3. Removal of the last element
  4. Removal of element at other positions



# Operation: remove(p)

**Algorithm**: remove(p):

  **if** (p = first) **then**

    first ← first.next

  **else**

    p.prev.next ← p.next

  **if** (p = last) **then**

    last ← last.prev

  **else**

    p.next.prev ← p.prev

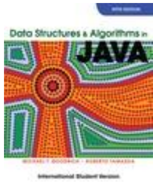
  size ← size - 1

  **return** p.element

# Operation: next(p)

- Basic Idea:
  - Returns the position of the next element in the list (if one exists)
- Process (Informal):
  - Check that we are not at the end of the list
  - Follow the link to the next node and return it
- Potential Use Cases:
  1. p is not the last position
  2. p is the last position
- Pseudo Code:

**Algorithm**: next(p):
    **if** (p = last) **then** throw a BoundaryViolationException
    **return** p.next

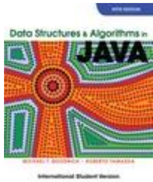


# Linked Lists: Implementation

- Class: `LinkedList` (implements `List`)
  - Inner Class:
    - A `Node` class that implements `Position`
  - Fields:
    - References to the `front` and `rear` of the doubly linked list
    - The `size` of the list (integer value)
  - Constructors:
    - One that creates an empty linked list
  - Methods:
    - One public method per operation

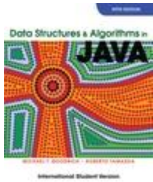
# Linked Lists: Implementation

```
public class LinkedList implements List {
   private Node first, last;
   private int size;

   private class Node implements Position {
        Object element;
        Node next, prev;
        public Node(Object element) {
             this.element = element;
        }
        public Object element() {
             return element;
        }
   }

   public LinkedList() {
        first = null;
        last = null;
        size = 0;
   }
```



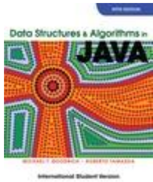
# Linked Lists: Implementation

```
public Position insertAfter(Position p, Object e) {
     if (p == last) insertLast(e);

     Node node = new Node(e);
     node.next = p.next;
     node.prev = p;

     p.next.prev = node;
     p.next = node;

     size++;
     return node;
}
```



# Linked Lists: Implementation

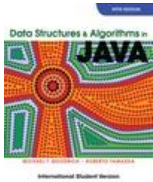
```
public Position insertAfter(Position p, Object e) {
      Node pos = (Node) p;

      if (pos == last) insertLast(e);

      Node node = new Node(e);
      node.next = pos.next;
      node.prev = pos;

      pos.next.prev = node;
      pos.next = node;

      size++;
      return node;
}
```



# List Traversal

- To loop through all of the objects stored in a Vector, the following code works (and is useful):

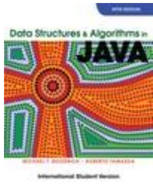
```
Vector v = new ArrayVector();
v.insertAtRank(v.size(), “H”);
v.insertAtRank(v.size(), “A”);
v.insertAtRank(v.size(), “P”);
v.insertAtRank(v.size(), “P”);
v.insertAtRank(v.size(), “Y”);
…
for (int j=0; j<v.size(); j++) {
  System.out.println(“v(“ + j + “)” + v.elemAtRank(j));
}
…
```



# List Traversal

- What about lists?

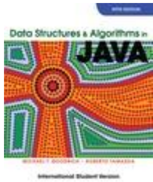
```
List l = new LinkedList();
l.insertLast("H");
l.insertLast("A");
l.insertLast("P");
l.insertLast("P");
l.insertLast("Y");
…
for (int j=0; j<l.size(); j++) {
  System.out.println("v(" + j + ")" + ???);
}
…
```



# List Traversal

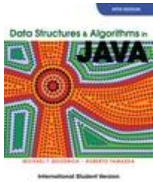
- A solution:

```
List l = new LinkedList();
l.insertLast(“H”);
l.insertLast(“A”);
l.insertLast(“P”);
l.insertLast(“P”);
l.insertLast(“Y”);
…
Position p = l.first();
for (int j=0; j<l.size(); j++) {
  System.out.println(“v(“ + j + “)” + p.element());
  p = l.next(p);
}
…
```



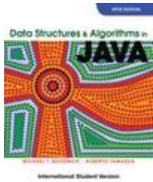
# Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with $n$ elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time



# Circularly Linked Lists

```java
/** Circulary linked list with nodes of type Node storing strings. */
public class CircleList {
  protected Node cursor;        // the current cursor
  protected int size;           // the number of nodes in the list
  /** Constructor that creates and empty list */
  public CircleList() { cursor = null; size = 0; }
  /** Returns the current size */
  public int size() { return size; }
  /** Returns the cursor */
  public Node getCursor() { return cursor; }
  /** Moves the cursor forward */
  public void advance() { cursor = cursor.getNext(); }
  /** Adds a node after the cursor  */
  public void add(Node newNode) {
    if (cursor == null) {    // list is empty
      newNode.setNext(newNode);
      cursor = newNode;
    }
    else {
      newNode.setNext(cursor.getNext());
      cursor.setNext(newNode);
    }
    size++;
  }
  /** Removes the node after the cursor */
  public Node remove() {
    Node oldNode = cursor.getNext();    // the node being removed
    if (oldNode == cursor)
      cursor = null; // list is becoming empty
    else {
      cursor.setNext(oldNode.getNext());        // link out the old node
      oldNode.setNext(null);
    }
    size--;
    return oldNode;
  }
  /** Returns a string representation of the list, starting from the cursor */
  public String toString() {
    if (cursor == null) return "[]";
    String s = "[..." + cursor.getElement();
    Node oldCursor = cursor;
    for (advance(); oldCursor != cursor; advance())
      s += ", " + cursor.getElement();
    return s + "...]";
  }
}
```



# Duck Duck Goose

```java
/** Simulation of Duck, Duck, Goose with a circularly linked list. */
public static void main(String[] args) {
  CircleList C = new CircleList();
  int N = 3; // number of iterations of the game
  Node it;    // the player who is "it"
  Node goose; // the goose
  Random rand = new Random();
  rand.setSeed(System.currentTimeMillis()); // use current time as seed
  // The players...
  String[] names = {"Bob","Jen","Pam","Tom","Ron","Vic","Sue","Joe"};
  for (int i = 0; i< names.length; i++) {
    C.add(new Node(names[i], null));
    C.advance();
  }
  for (int i = 0; i < N; i++) {        // play Duck, Duck, Goose N times
    System.out.println("Playing Duck, Duck, Goose for " + C.toString());
    it = C.remove();
    System.out.println(it.getElement() + " is it.");
    while (rand.nextBoolean() || rand.nextBoolean()) { // march around circle
      C.advance(); // advance with probability 3/4
      System.out.println(C.getCursor().getElement() + " is a duck.");
    }
    goose = C.remove();
    System.out.println(goose.getElement() + " is the goose!");
    if (rand.nextBoolean()) {
      System.out.println("The goose won!");
      C.add(goose); // add the goose back in its old place
      C.advance();  // now the cursor is on the goose
      C.add(it);    // The "it" person will be it again in next round
    }
    else {
      System.out.println("The goose lost!");
      C.add(it);     // add who's "it" back at the goose's place
      C.advance(); // now the cursor is on the "it" person
      C.add(goose);  // The goose will be "it" in the next round
    }
  }
  System.out.println("Final circle is " + C.toString());
}
```