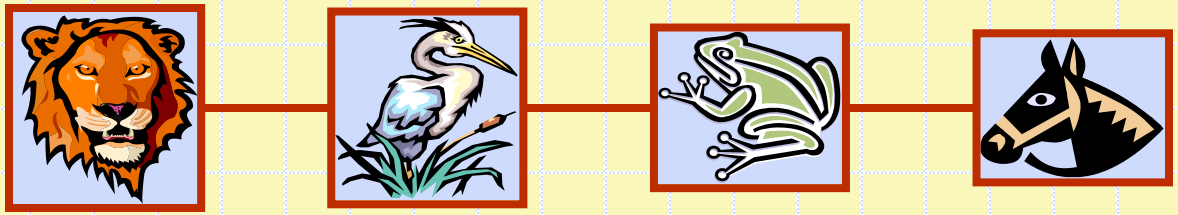
Linked Lists & Iterators

Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

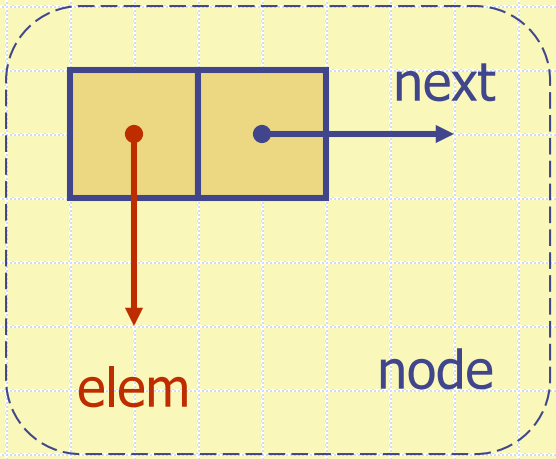
Coverage

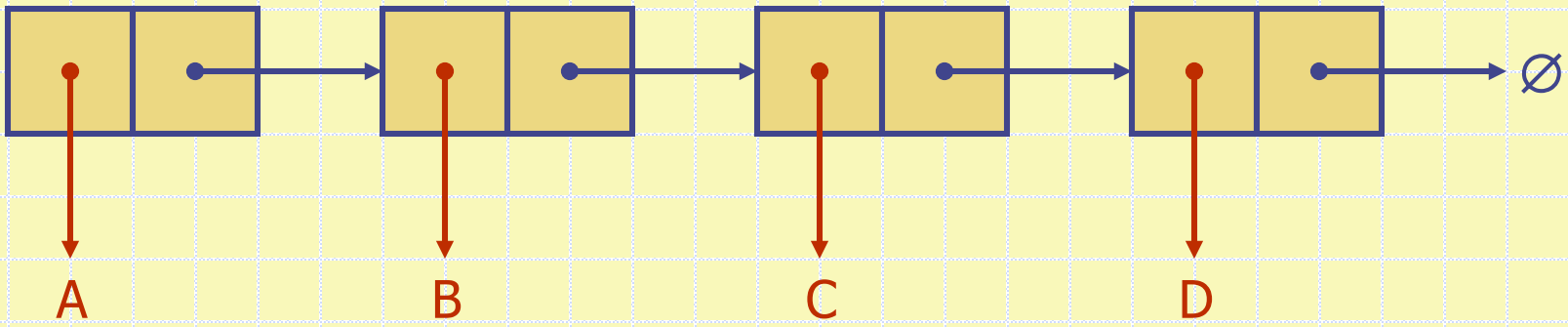


- Singly Linked Lists
- Doubly Linked Lists
- Circularly Linked Lists
- Iterators

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



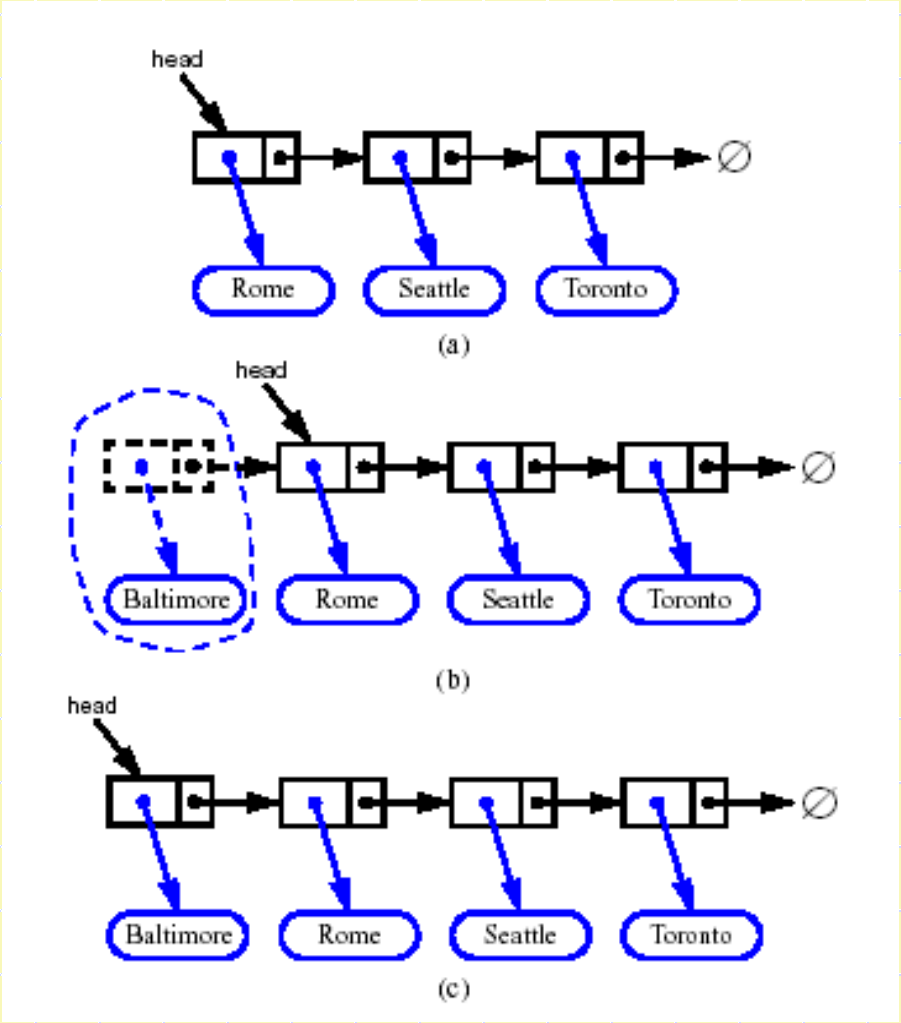


The Node Class for List Nodes

```
public class Node          {
   // Instance variables:
   private  Object element;
   private  Node next;
   /** Creates a node with null references to its element and next node. */
   public  Node()          {
     this(null,  null);
   }
   /** Creates a node with the given element and next node. */
   public  Node(Object e,  Node n)  {
       element  =  e;
       next  =  n;
   }
   // Accessor methods:
   public  Object getElement()  {
     return  element;
   }
   public  Node getNext()  {
     return  next;
   }
   // Modifier methods:
   public void  setElement(Object newElem)  {
       element  =  newElem;
   }
   public void  setNext(Node newNext)  {
       next  =  newNext;
   }
}
```

Inserting at the Head

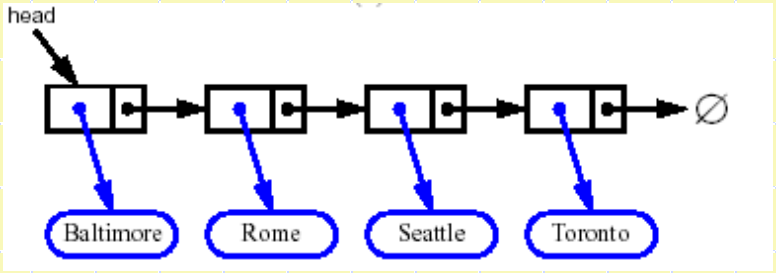
1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

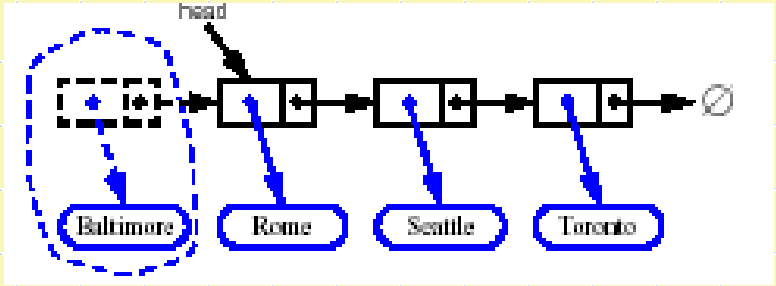


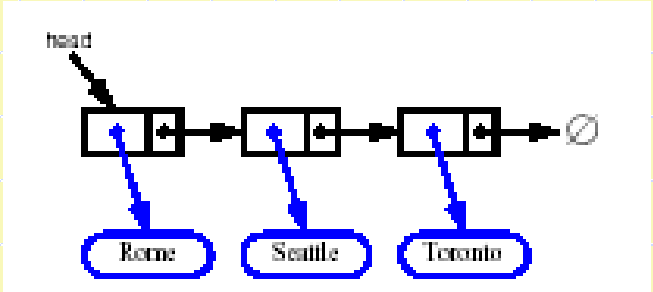
Removing at the Head

1. Update head to point to next node in the list

2. Allow garbage collector to reclaim the former first node

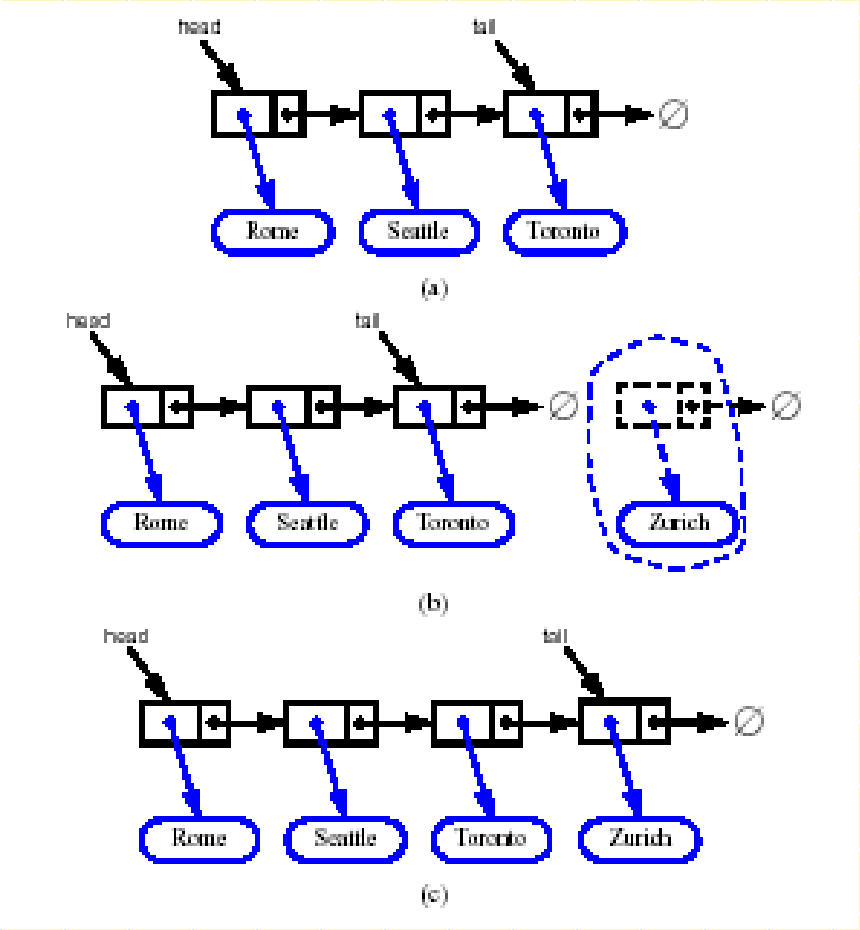






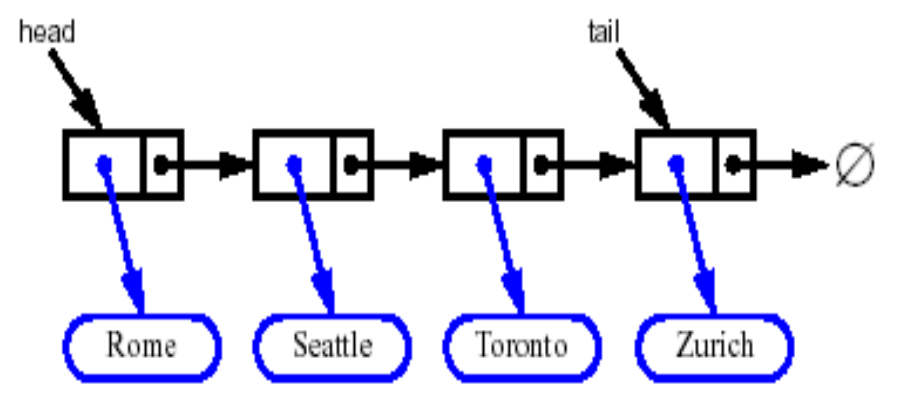
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



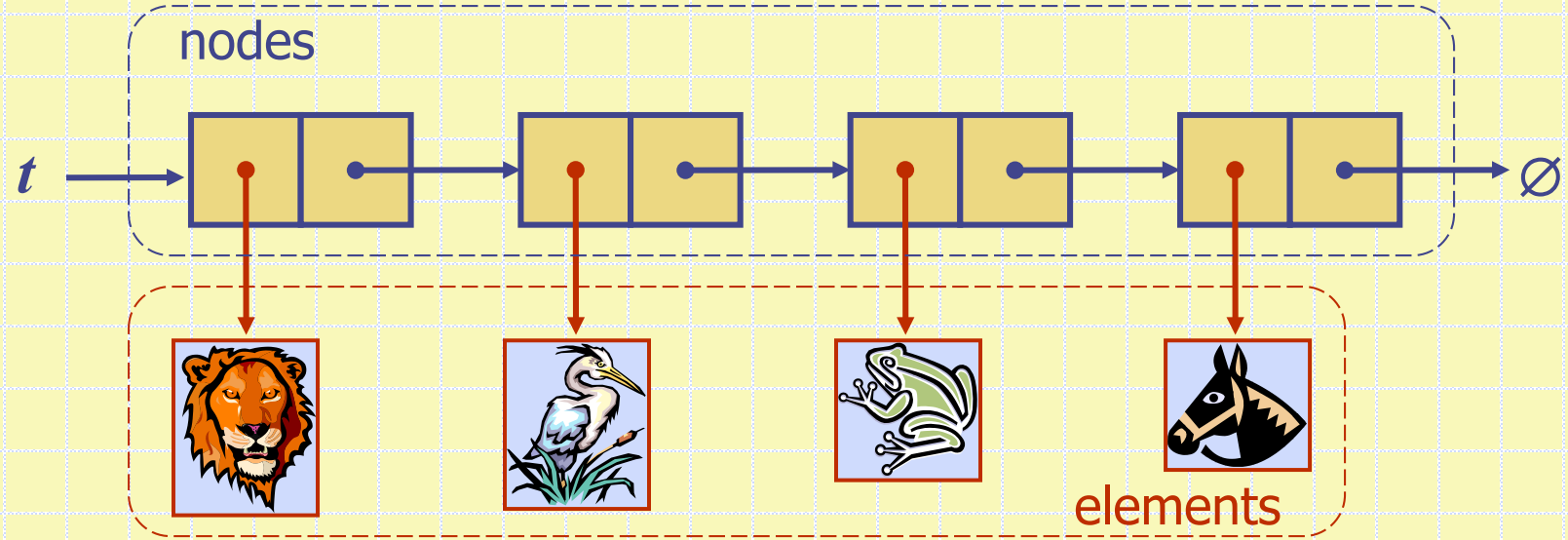
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



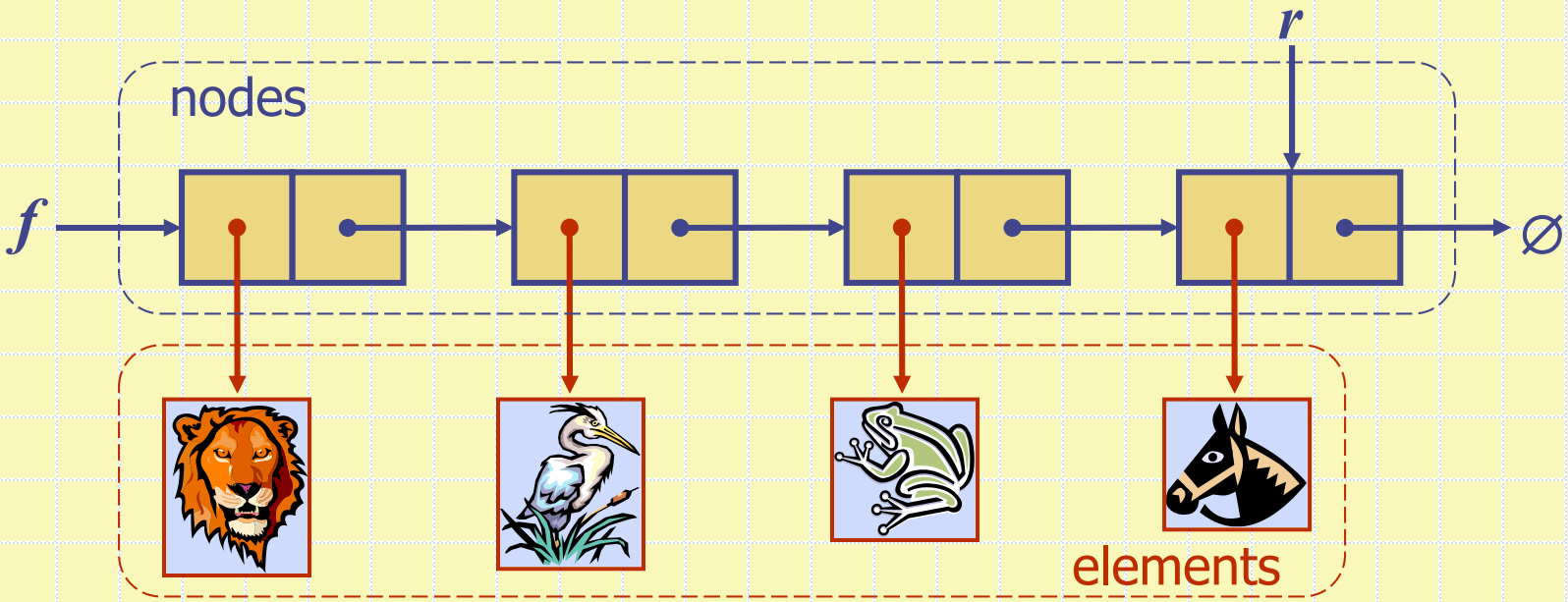
Stack as a Linked List (Review Section 5.1.3)

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



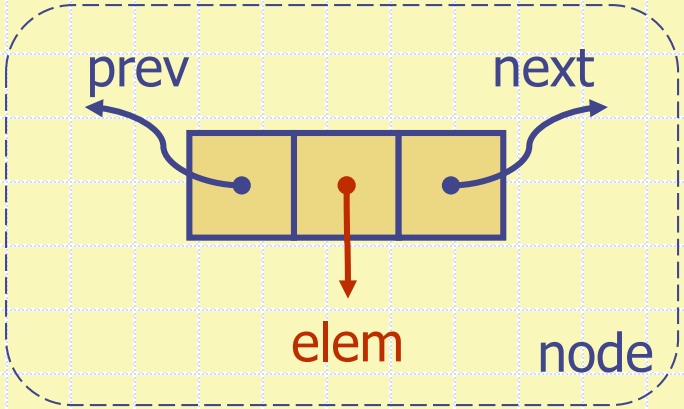
Queue as a Linked List (Review Section 5.2.3)

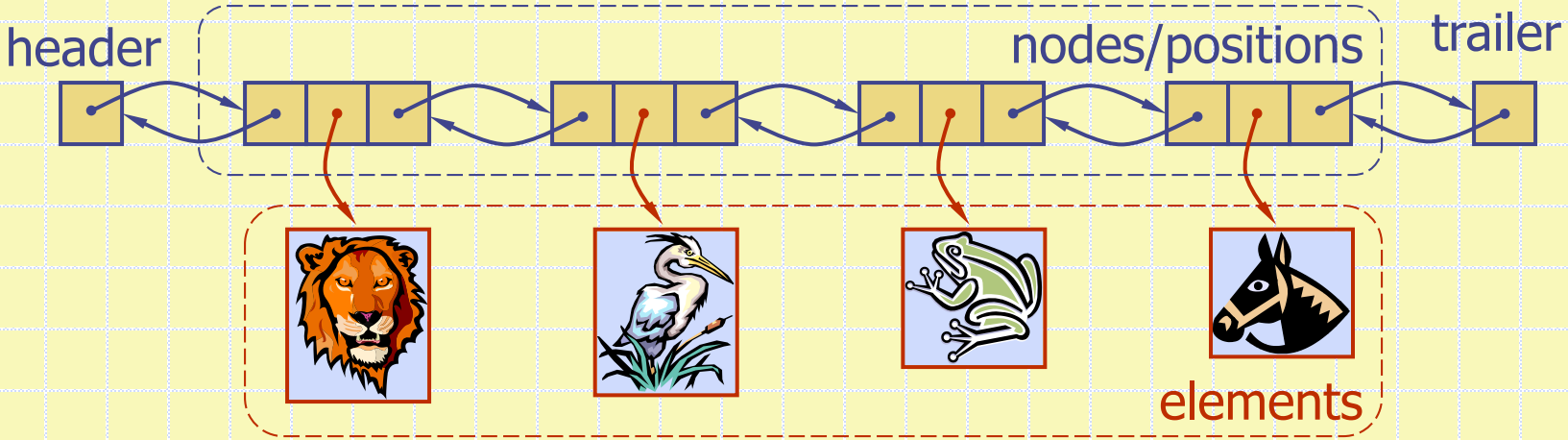
- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time (assuming that both head and rear are pointed to!)



Doubly Linked List

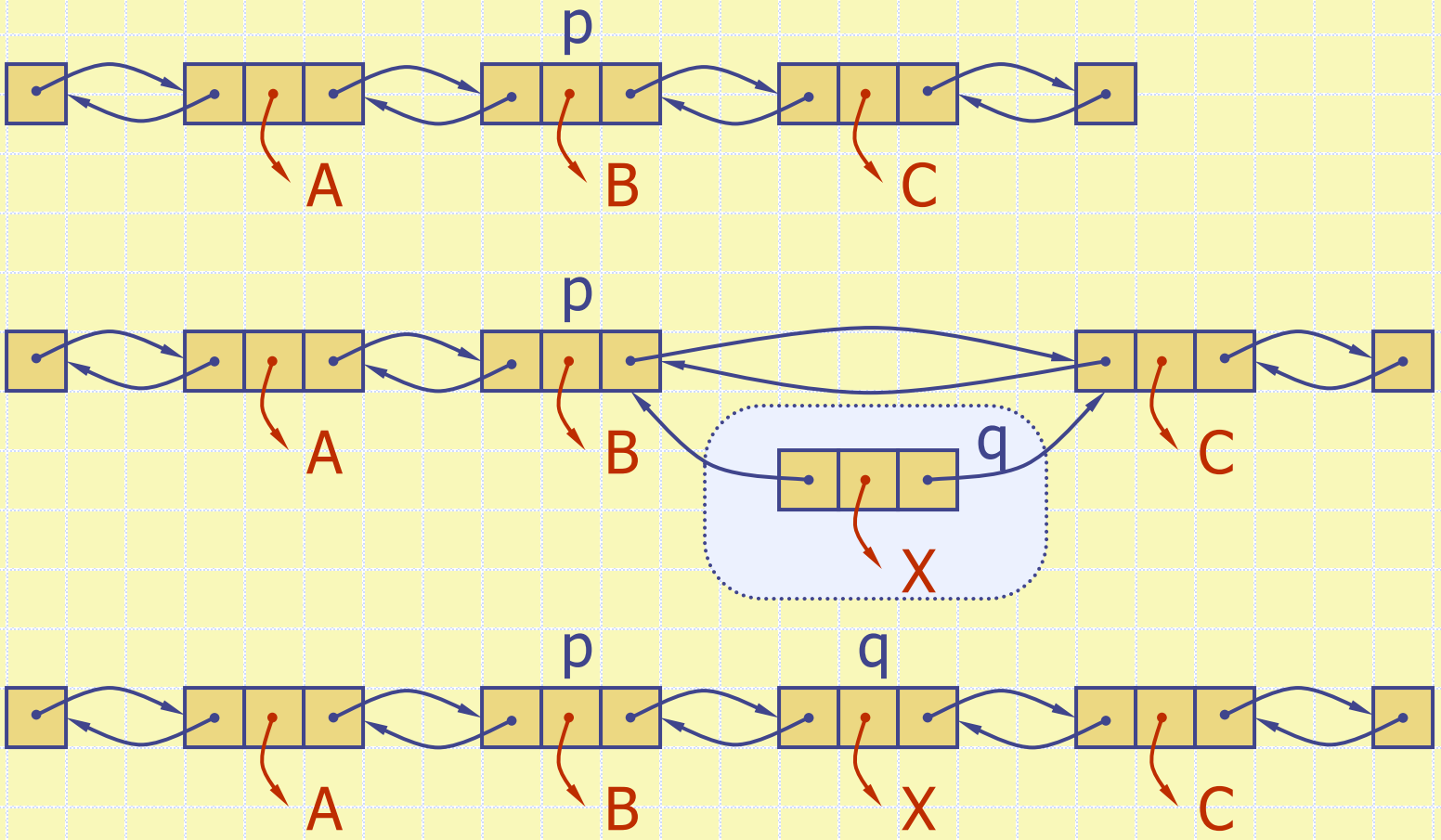
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special sentinel/dummy trailer and header nodes (simplify implementation)





Insertion

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



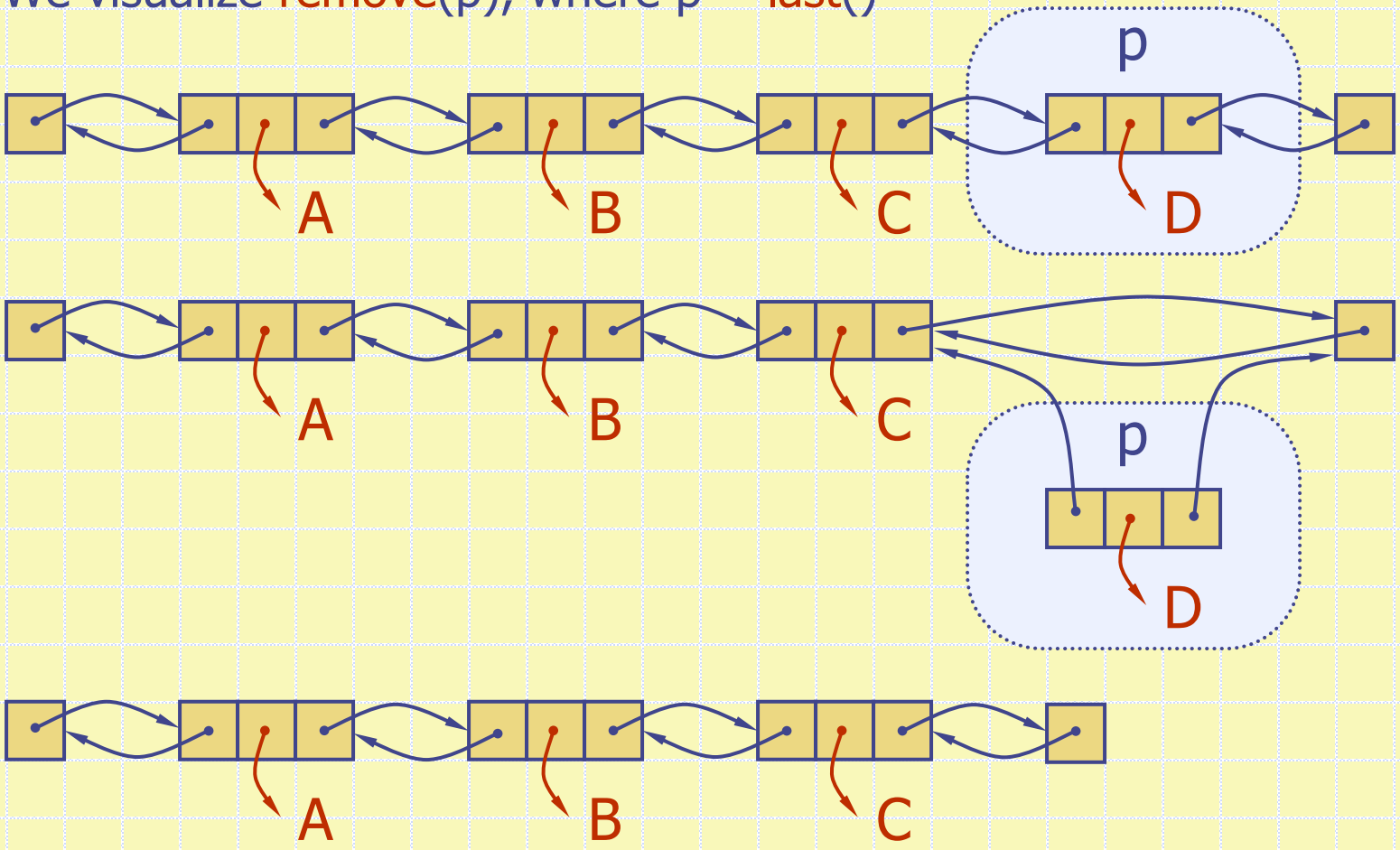
Insertion Algorithm

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
```

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 $O(1)$
  - Operations of the List ADT run in $O(1)$ time (as will be discussed later)
  - Operation element() of the Position ADT also runs in $O(1)$ time (as will be discussed later)

Iterators

- An iterator abstracts the process of scanning through a collection of elements
- It maintains a cursor that sits between elements in the list, or before the first or after the last element
- Methods of the Iterator ADT:
  - hasNext(): returns true so long as the list is not empty and the cursor is not after the last element
  - next(): returns the next element
- Extends the concept of position by adding a traversal capability

Iterable Classes

- An iterator is typically associated with an another data structure, which can implement the Iterable ADT
- We can augment the different ADTs (i.e. Stack, Queue, List, etc.) with method:
  - Iterator<E> iterator(): returns an iterator over the elements
  - In Java, classes with this method extend Iterable<E>
- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure
  - In Java: an iterator will fail (and throw an exception) if the underlying collection changes unexpectedly

The For-Each Loop

- Java provides a simple way of looping through the elements of an Iterable class:
  - for (type name: expression)
    loop_body
  - For example:

```
List<Car> ls1;
for (Car c : ls1)
  System.out.println(c); // Car is assumed to have a toString method defined.
```

List Iterators in Java

- Java uses a the ListIterator ADT for node-based lists.
- This iterator includes the following methods:
  - add(e): add e at the current cursor position
  - hasNext(): true if there is an element after the cursor
  - hasPrevious: true if there is an element before the cursor
  - previous(): return the element e before the cursor and move cursor to before e
  - next(): return the element e after the cursor and move cursor to after e
  - set(e): replace the element returned by last, next or previous operation with e
  - remove(): Removes from the list the last element that was returned by next() or previous(). This call can only be made once per call to next() or previous(). It can be made only if add(e) has not been called after the last call to next() or previous().