# Linked Lists

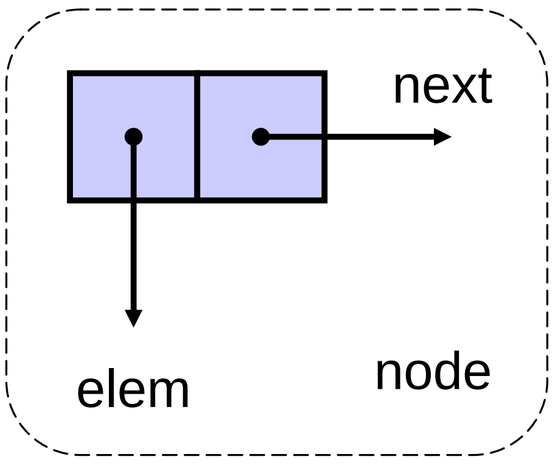
CSE 2011

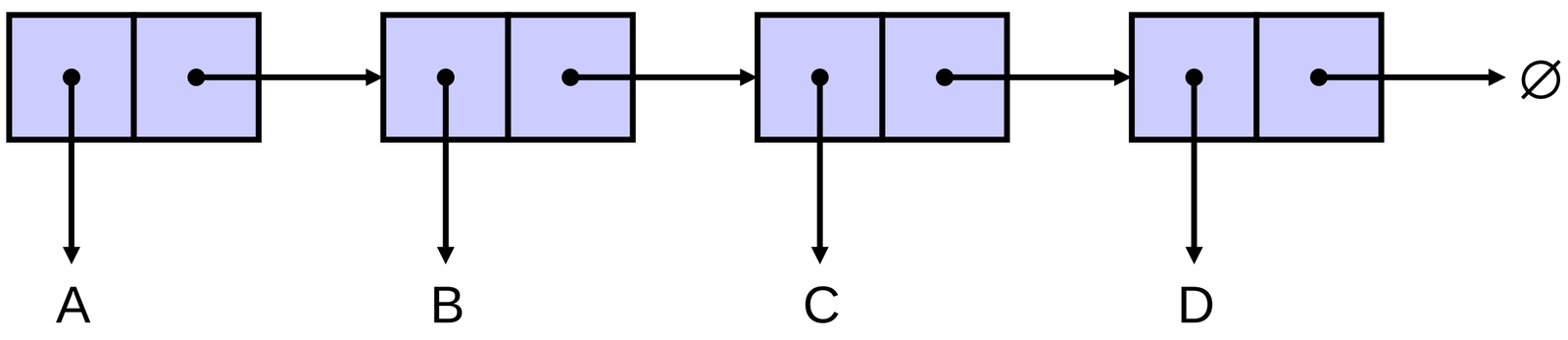
Fall 2009



## Singly Linked Lists (3.2)

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



# “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;
  }
}
```



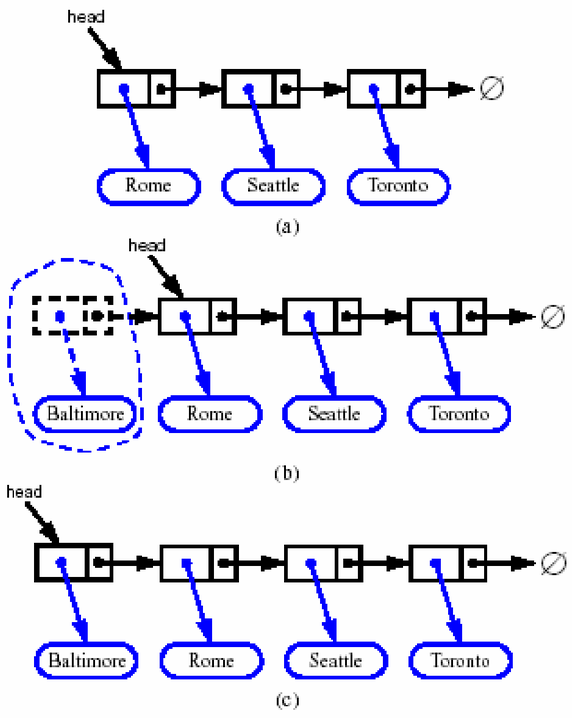
# SLinkedList class

```
/** 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 ...
}
```

# Inserting at the Head

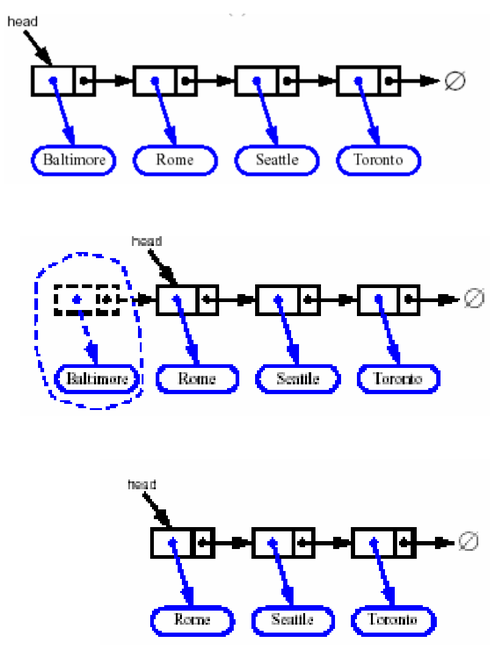
1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node






# Removing at the Head

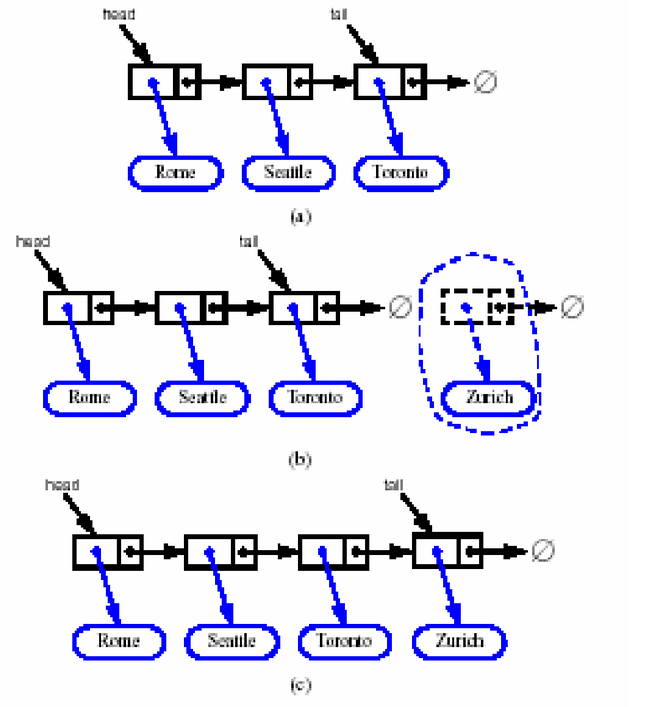
1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



# Inserting at the Tail

Assume that we keep a pointer to the last element of the list (“tail”).

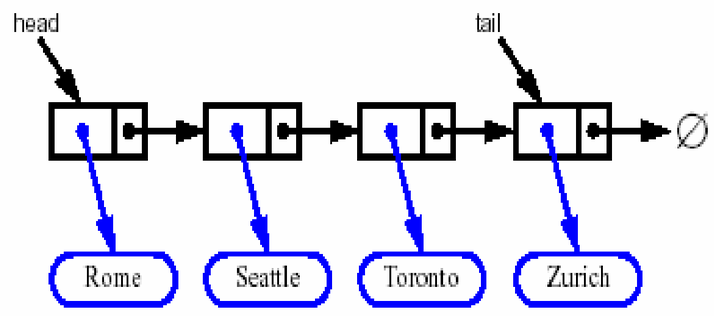
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node





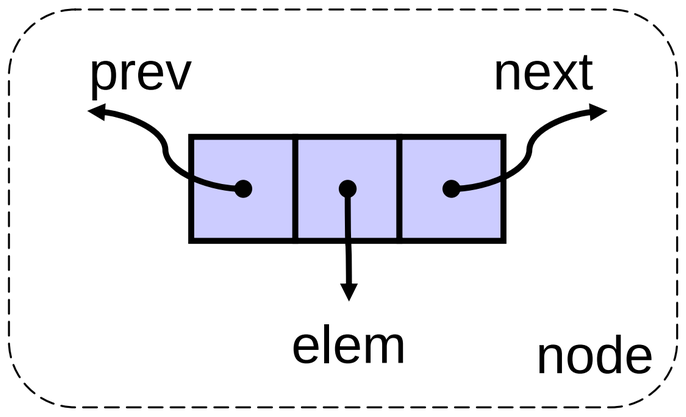

# Removing at the Tail

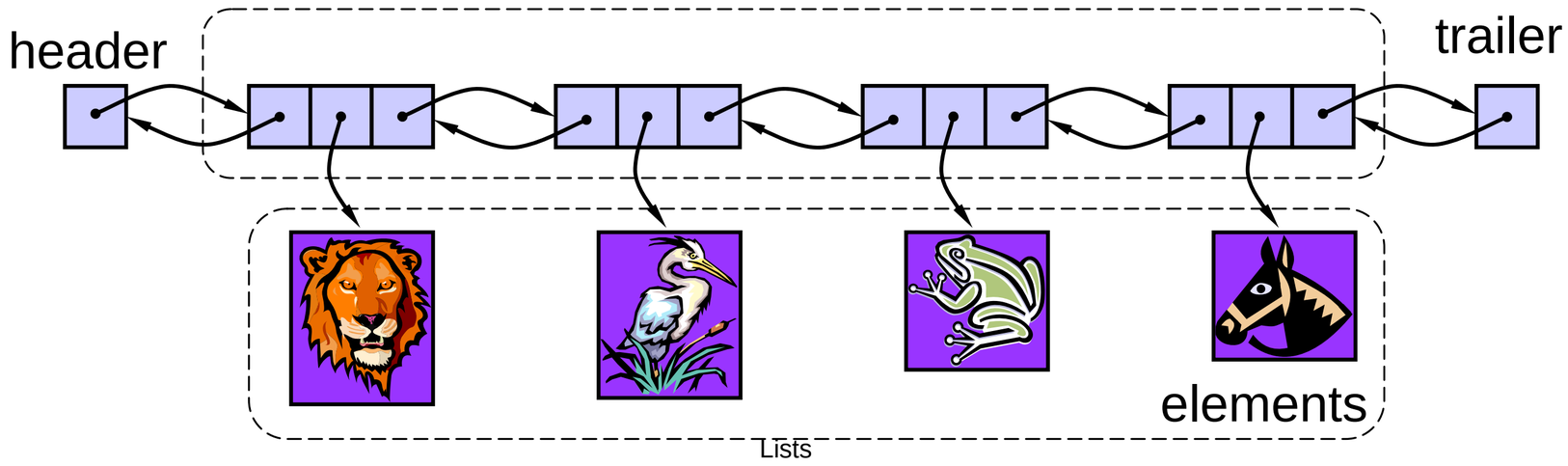
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node.



# Doubly Linked List (3.3)

- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes






# Header and Trailer Sentinels

- Dummy nodes which do not store any elements
- To simplify programming

# “Dnode” Class

```
/** Node of a doubly linked list of strings */
public class DNode {
  protected String element;      // String
     element stored by a node
  protected DNode next, prev;  // Pointers
     to next and previous nodes
  /** Constructor that creates a node with
     given fields */
  public DNode(String e, DNode p, DNode
     n) {
    element = e;
    prev = p;
    next = n;
  }
```

```
  /** Returns the element of this node */
  public String getElement() { return
     element; }
  /** Returns the previous node of this
     node */
  public DNode getPrev() { return prev; }
  /** Returns the next node of this node */
  public DNode getNext() { return next; }
  /** Sets the element of this node */
  public void setElement(String newElem) {
     element = newElem; }
  /** Sets the previous node of this node */
  public void setPrev(DNode newPrev) {
     prev = newPrev; }
  /** Sets the next node of this node */
  public void setNext(DNode newNext) {
     next = newNext; }
}
```



# “DList” Class

```
/** 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 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
  }
... // Implementation of methods
}
```

**Methods:**

int size()

boolean isEmpty()

DNode getFirst()

DNode getLast()

DNode getPrev(DNode v)

DNode getNext(DNode v)

void addBefore(DNode v, DNode z)

void addAfter(DNode v, DNode z)

void addFirst(DNode v)

void addLast(DNode v)

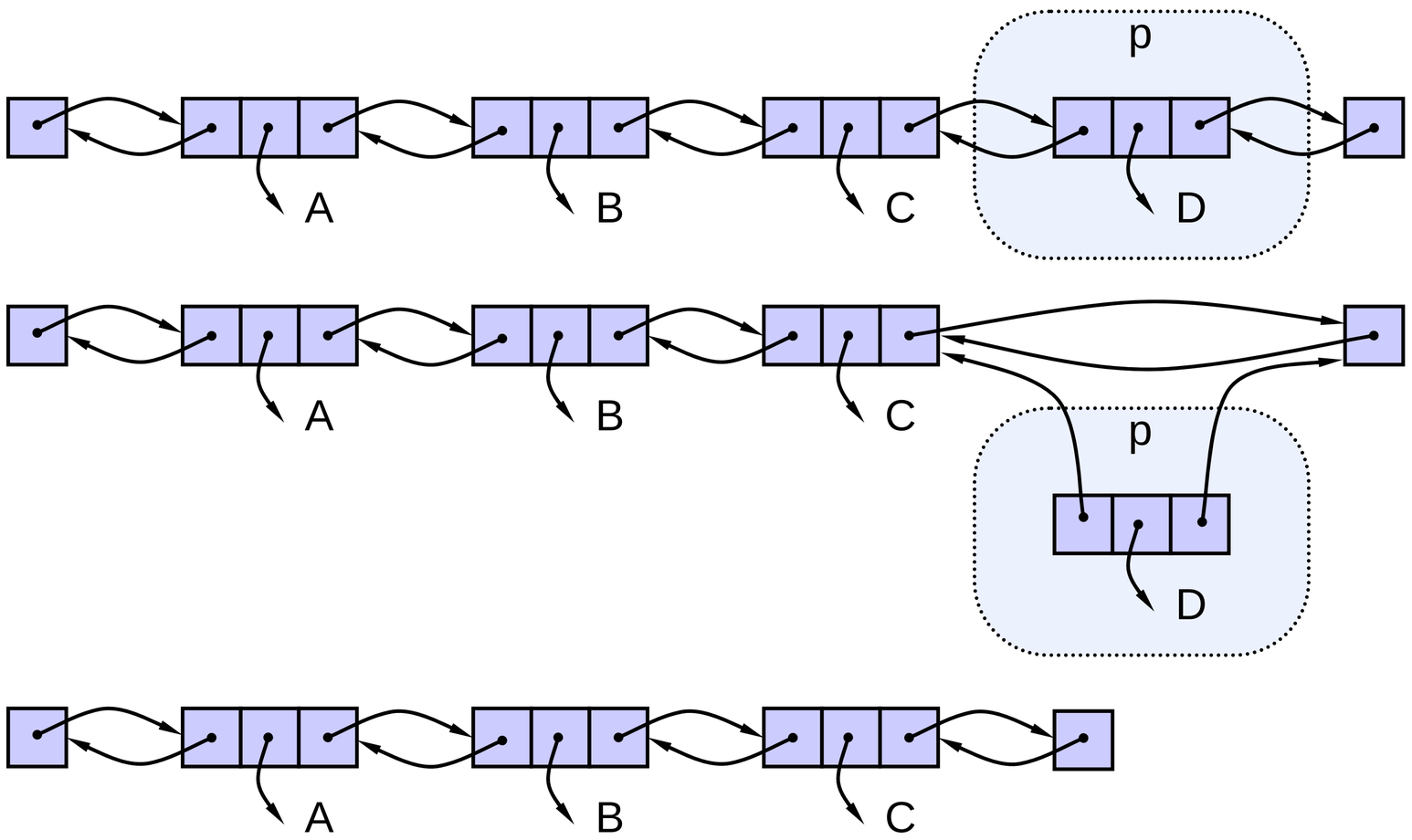
void remove(DNode v)

# Insert/Remove at Either End

- Straightforward.
- Example 1: removing the last node.
  - Figure 3.15 (next slide)
- Example 2: inserting a new node at the beginning of the list (head).
  - Figure 3.16



# Removal at the Tail of the List



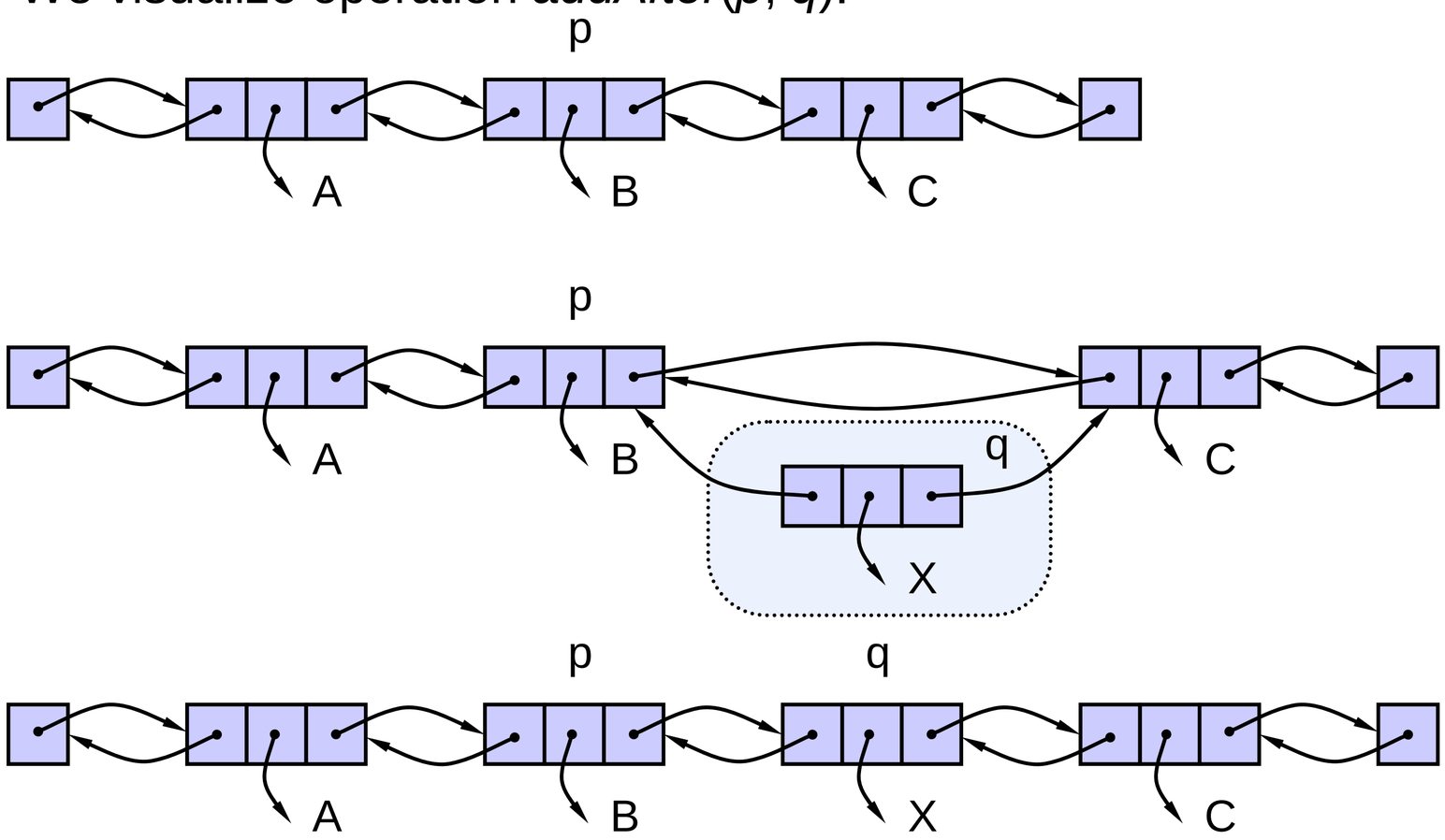
# Removal at the Tail:Algorithm

```
Algorithm removeLast() {
if size == 0 then
   Indicate error “empty list”;
v = trailer.getPrev();  // last node
u = v.getPrev();        // node before last node
trailer.setPrev(u)
u.setNext(trailer);
v.setPrev(null);
v.setNext(null);
size = size – 1;
}
```



# Insertion in the Middle of the List

- We visualize operation *addAfter*(*p*, *q*).



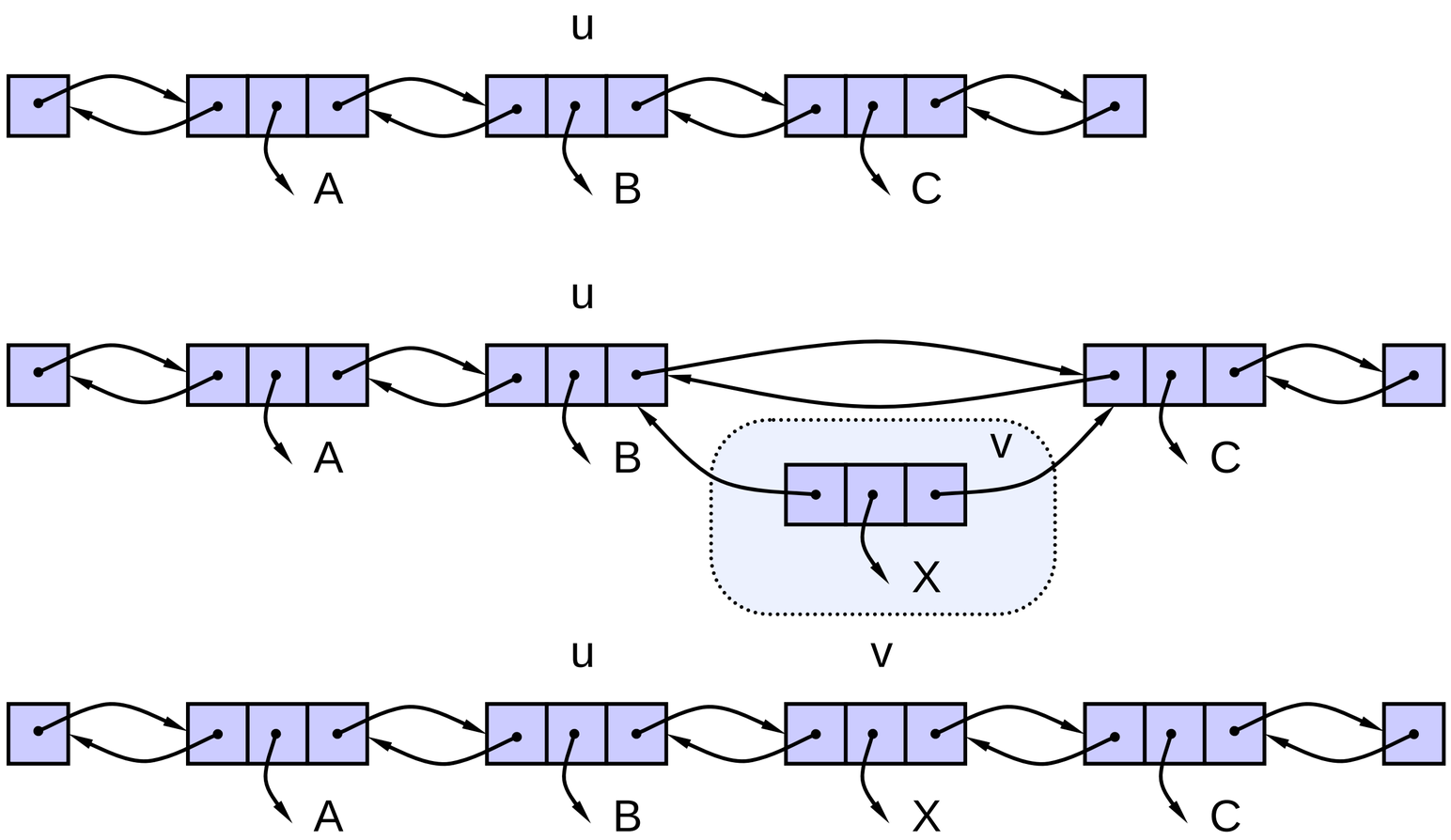
# Insertion Algorithm

Algorithm *addAfter*(p, q) {
r = p.getNext;  // node after p
q.setPrev(p);  // link q to its predecessor, p
q.setNext(r);  // link q to its successor, r
r.setPrev(q) );  // link r to its new predecessor, q
p.setNext(q);  // link p to its new successor, q
size = size + 1;
}



# Removal in the Middle of the List

- We visualize operation *remove*(*v*).



# Removal Algorithm

```
Algorithm remove(v) {
u = v.getPrev();    // node before v
w = v.getNext();    // node after v
w.setPrev(u);       // link out v
u.setNext(w);
v.setPrev(null);    // null out the fields of v
v.setNext(null);
size = size – 1;
}
```



# Implementation of Doubly Link Lists

- Section 3.3.3, p.125.
- Homework: re-do the implementation without using the header and trailer sentinels.

# Next time …

- Stacks (5.1)
- Queues (5.2)