Arrays - Linked Lists

cse2011
sections 3.1,3.2,3.3

Arrays

A common programming task is to keep track of a group of related objects!

Array – **sequence of indexed components with the following properties:**

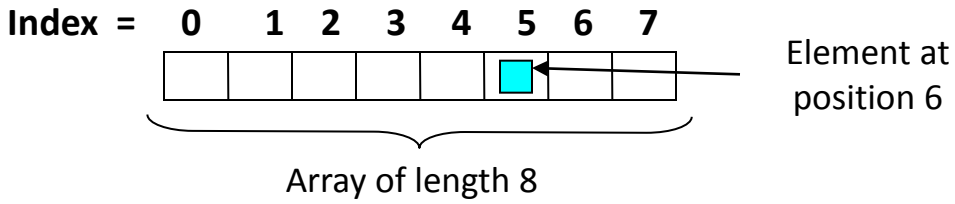
- **array size is fixed at the time of array's construction**

```
int[] numbers = new numbers [10];
```

- **array elements are placed contiguously in memory** ⇒ address of any element can be calculated directly as its offset from the beginning of the array

- consequently, **array components can be efficiently inspected or updated in O(1) time, using their indices**

```
randomNumber = numbers[5];
numbers[2] = 100;
```



Arrays in Java – Properties

(1) **For an array of length n, the index bounds are 0 to (n-1).**

(2) **Java arrays are homogeneous - all array components must be of the same (object or primitive) type.**

- **but, an array of an object type can contain objects of any respective subtype**

(3) **An array is itself an object.**

- **it is allocated dynamically by means of "new", it is automatically deallocated when no longer referred to**

(4) **when an array is first created, all values are initialized with**

- **0 – for an array of int[] or double[] type**
- **false – for a boolean[] array**
- **null – for an array of objects**

Example 1 [*common error* – uninitialized arrays]

```
int[] numbers;
numbers[2] = 100;
```

Arrays in Java – Properties (cont.)

(5) The length of any array (object) can be accessed through its instance variable 'length'.

- **the cells of an array A are numbered: 0, 1, .., (A.length-1) !!!**

(6) ArrayIndexOutOfBoundsException - thrown at an attempt to index into array A using a number larger than (A.length-1).

- **helps Java avoid 'buffer overflow attacks'**

Example 2 [declaring, defining and determining the size of an array]

```
int[] A={12, 24, 37, 53, 67}
for (int i=0; i < A.length; i++) {
   … }
```

Array is defined at the time of its 'declaration'.

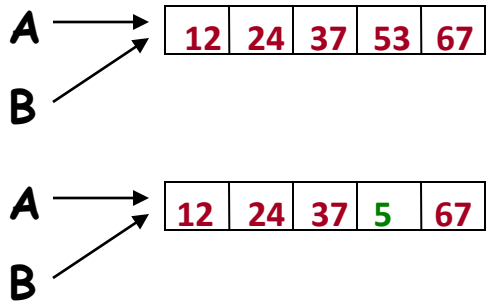
Arrays in Java – Properties (cont.)

(7) Since an array is an object, the name of the array is actually a reference (pointer) to the place in memory where the array is stored.

- **reference to an object holds the address of the actual object**

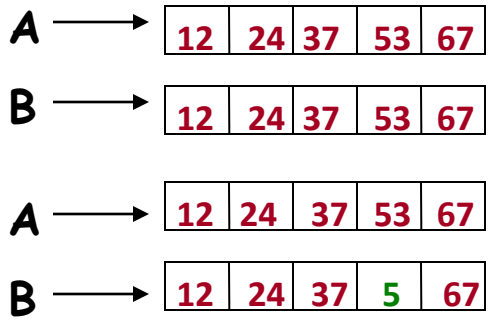
Example 3 [arrays as objects]

```
int[] A={12, 24, 37, 53, 67};
int[] B=A;
B[3]=5;
```



Example 4 [cloning an array, making a **distinct** copy of the array with its own reference]

```
int[] A={12, 24, 37, 53, 67};
int[] B=A.clone();
B[3]=5;
```



Arrays in Java: a few useful methods (java.util.Arrays)

equals(A,B) – returns true if A and B have an equal number of elements and every corresponding pair of elements in the two arrays are referencing to the same object

fill(A,x) – store element x into every cell of array A

sort(A) – sort the array A in the natural ordering of its elements. All elements in the array must implement the Comparable interface.

binarySearch([Object] A, Object key)

Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the natural ordering of its elements

Arrays (cont.)

Example **[** What will be the output ?! **]**

```
…
import java.util.Arrays;
…
        int[] A={12, 24, 37, 53, 67};

        int[] B=A.clone();

        if (A==B)  System.out.println("1");

        if (A.equals(B))  System.out.println("2");

        if (Arrays.equals(A,B))  System.out.println("3");

…
```

MainC.java

```
import java.util.*;
public class MainC {
    public static void main(String [] args){
        CC[] C1 = new CC[1];
        CC d1 = new CC("a");
        C1[0] = d1;

        CC[] C2 = new CC[1];
        CC d2 = new CC("a");
        C2[0] = d2;

        System.out.println(Arrays.equals(C1,C2));
    }
}

class CC{
    public CC(String name){
        this.name = name;
    }
    String name;
}
```

Output? true or false?

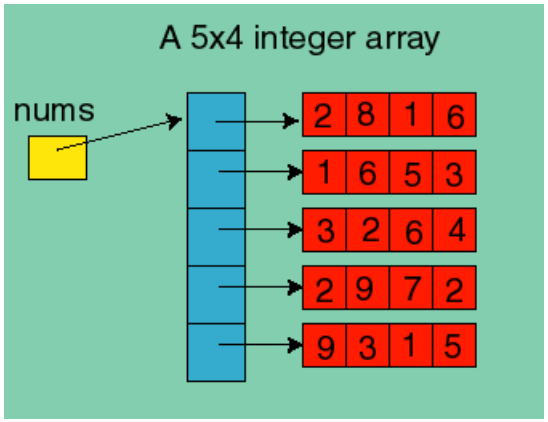
What if C2[0] = d1 ??

Arrays (cont.)

Example 4 [2D array in Java = array of arrays !!!]

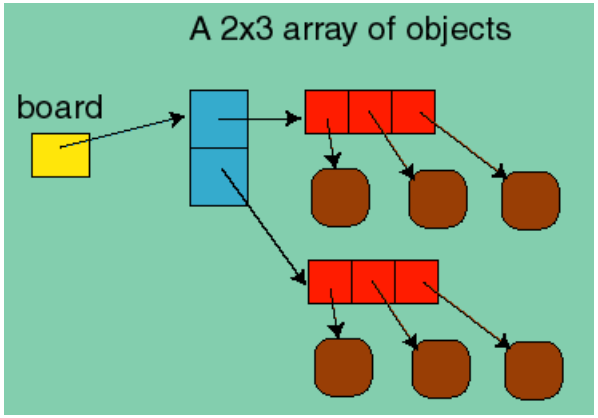
```
int[][] nums = new int[5][4];
```

```
int[][] nums;
nums = new int[5][];
for (int i=0; i<5; i++) {
        nums[i] = new int[4]; }
```



Example 5 [2D array of objects in Java = an array of arrays of references !!!]

```
Square[][] board = new Square[2][3];
```



Arrays in General – Major Limitations!

(1) static data structure – size must be fixed at the time the program creates the array – once set, array size cannot be changed

- **if: *number of entered items* > *declared array size* ⇒ out of memory**
- **fix 1: use *array size* > *number of expected items* ⇒ waste of memory**
- **fix 2: increase array size to fit the number of items ⇒ extra time**

Example 5 [time complexity of "growing" an array]

```
if (numberOfItems > numbers.length) {
    int[] newNumbers = new int[2*numbers.length];
    System.arraycopy(numbers, 0, newNumbers, 0, numbers.length);
    numbers = newNumbers;
}
```

Number of elements to be copied.

Starting position in source array.

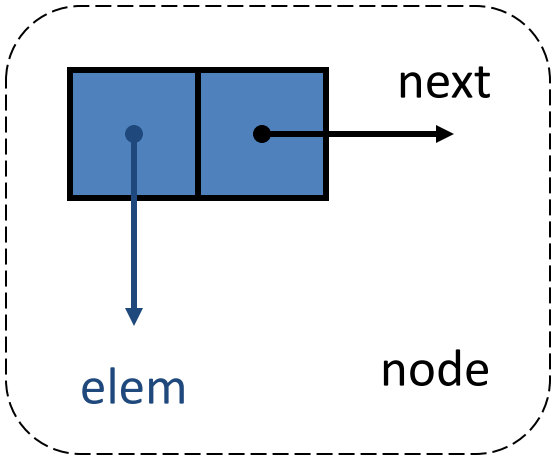
Starting position in destination array.

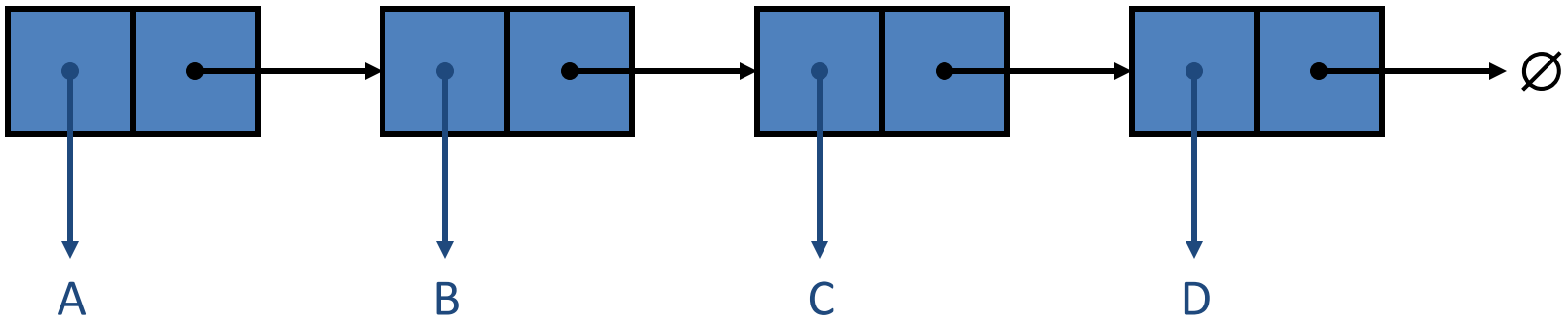
(2) insertion / deletion in an array is time consuming – all the elements following the inserted element must be shifted appropriately

Linked Lists

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

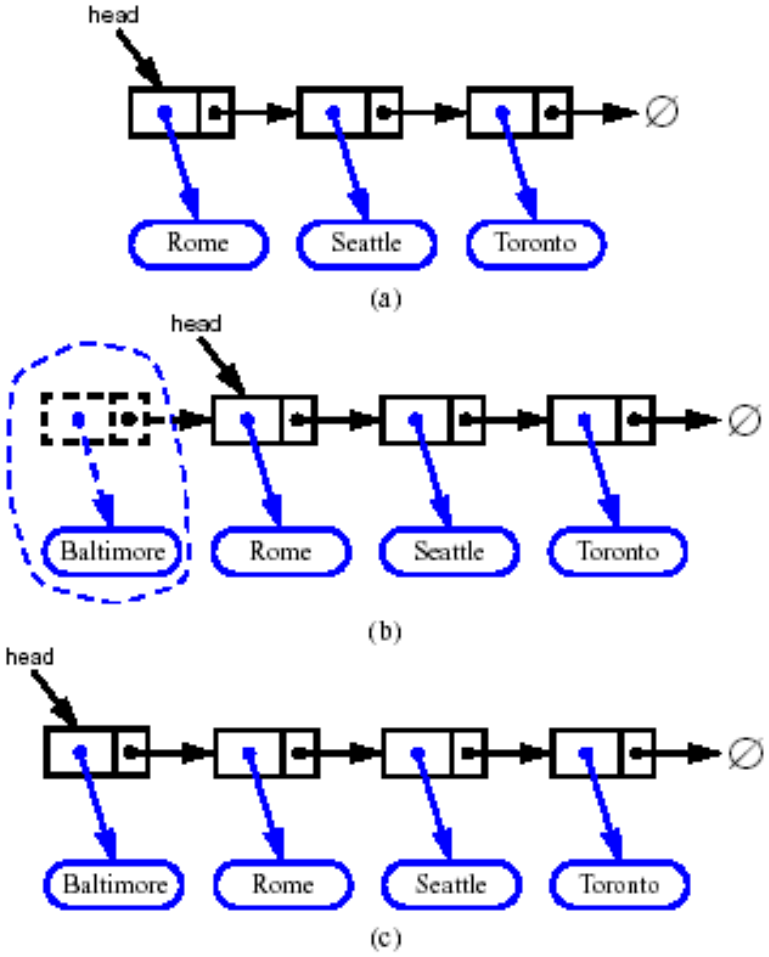
```
/** Singly linked list .*/
public class SLinkedList {
  protected Node head;                    // head node of the list
  protected long size;                    // number of nodes in the list
  /** Default constructor that creates an empty list */
  public SLinkedList() {
    head = null;
    size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

  // ... update and search methods would go here ...
}
```

Inserting at the Head

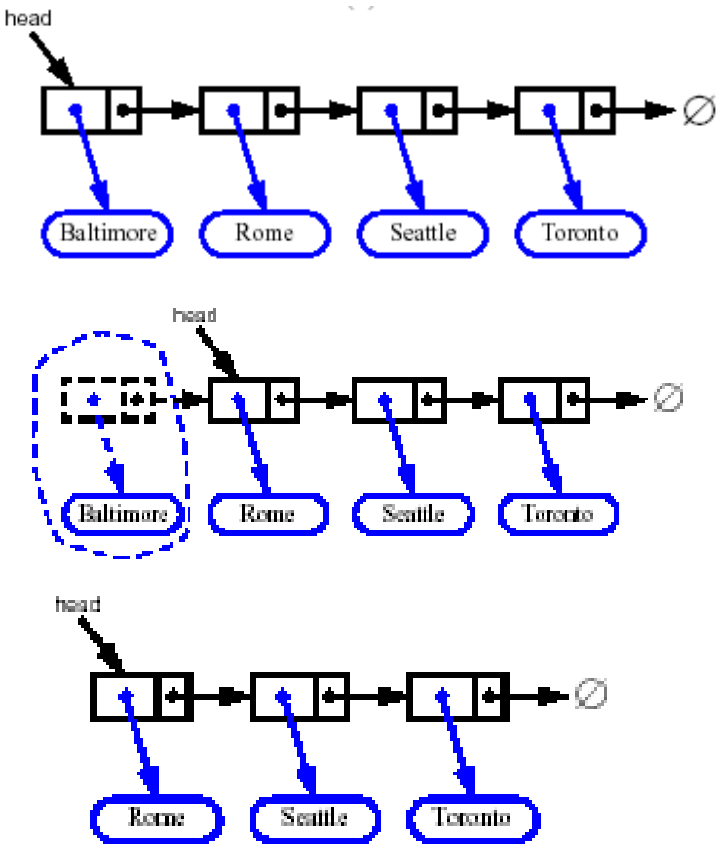
1. Allocate a new node.
2. Insert new element.
3. Have new node point to old head.
4. Update head to point to new node.

The order of operations is very important



Removing at the Head

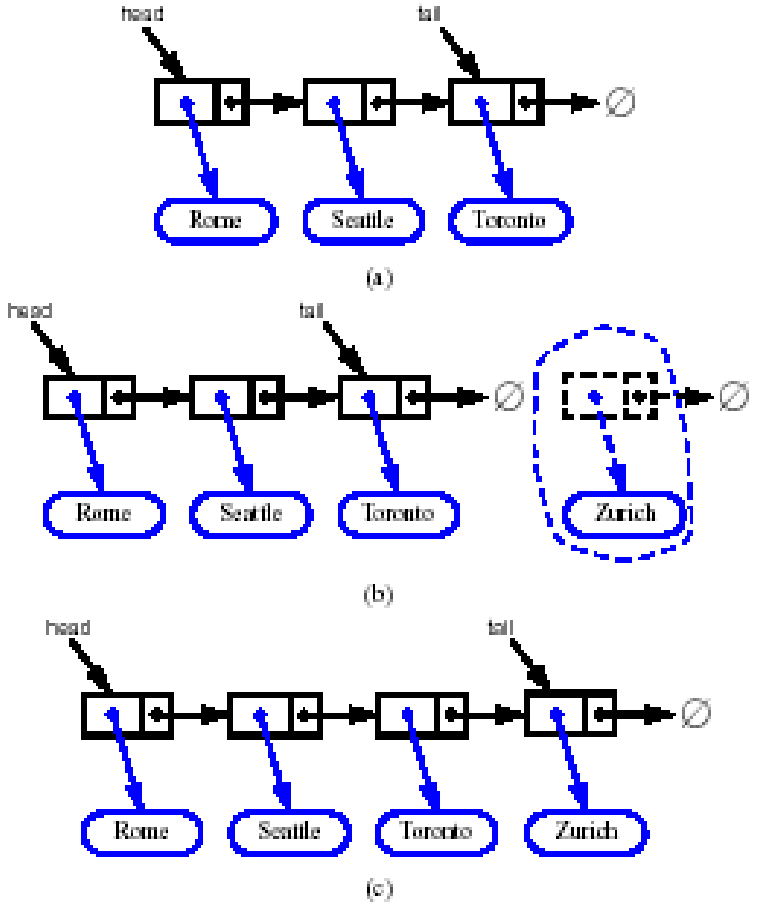
1. Update head to point to next node in the list.
2. Allow garbage collector to reclaim the former first node.



Inserting at the Tail

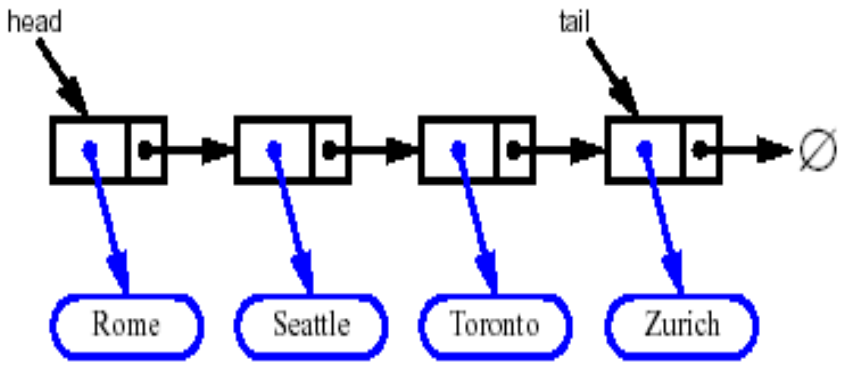
Assume that we keep a pointer to the last element of the list ("tail").

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



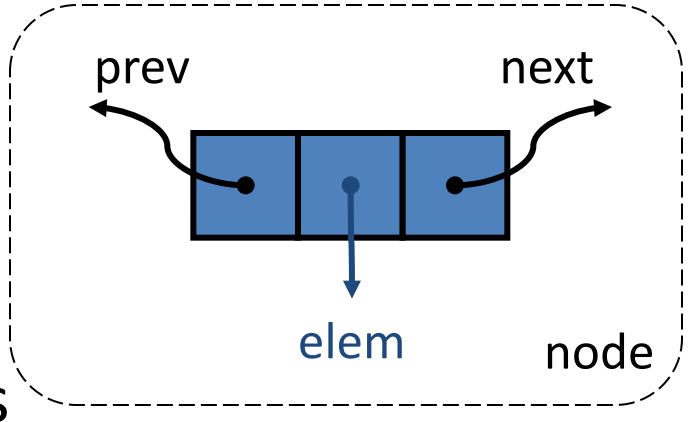
Removing at the Tail

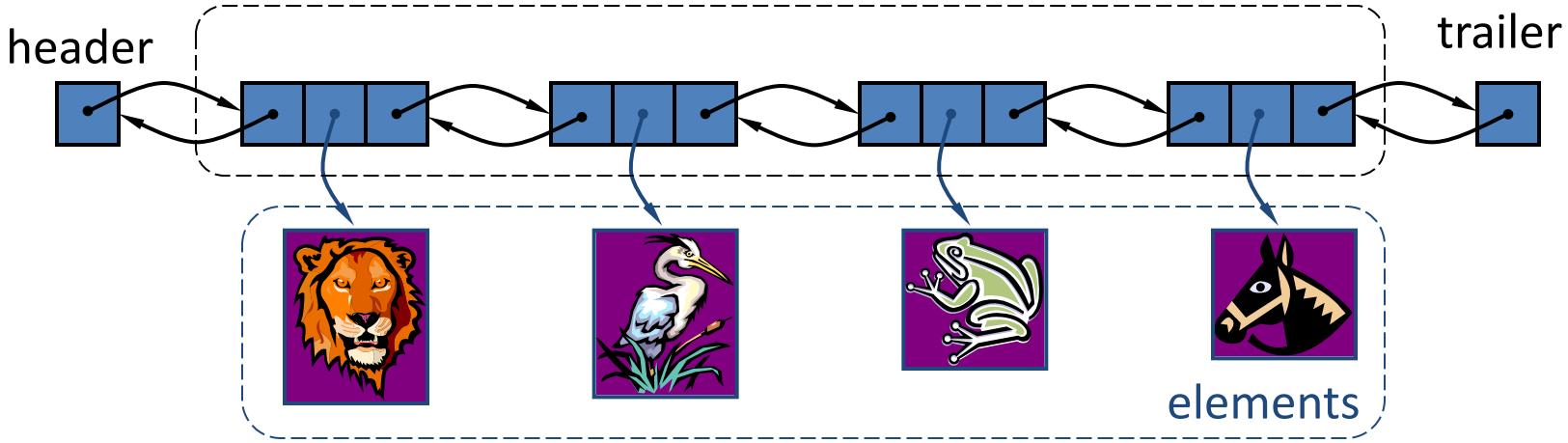
- Removing at the tail of a singly linked list is not efficient!
- There is **no** constant-time way to update the tail to point to the previous node.
- Homework: write Java code to remove at the tail of a singly linked list.



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.

- Homework: implement the operations of a doubly linked list with NO header and trailer sentinels.
  - Compare the code of the 2 implementations.

“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)

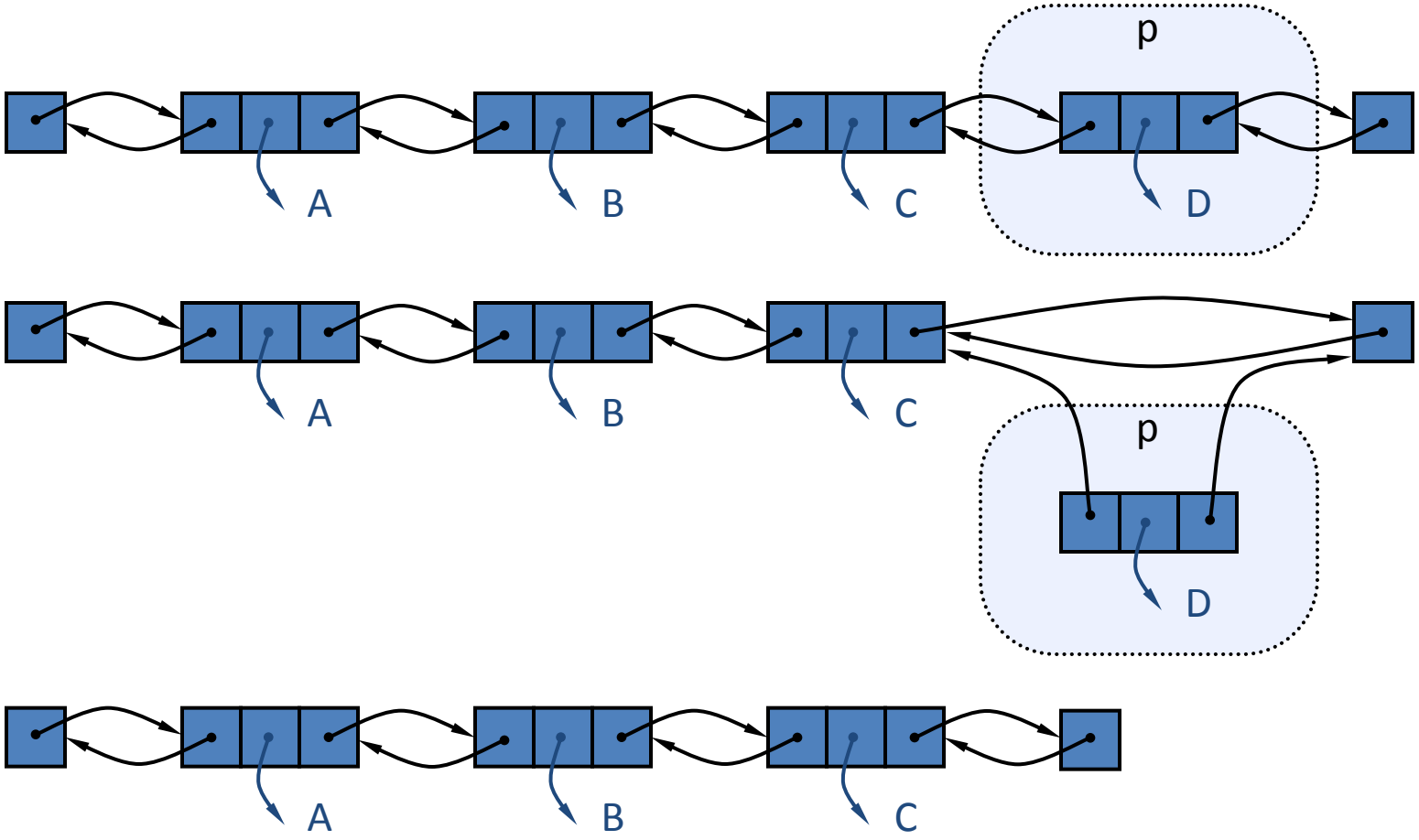
void remove(DNode v)

Setting the Prev pointer

Insert/Remove at Either End

- Straightforward.
- Example 1: removing the last node.
  - next slide …
- Example 2: inserting a new node at the beginning of the list (head).
  - To be discussed …

Removal at the Tail of the List

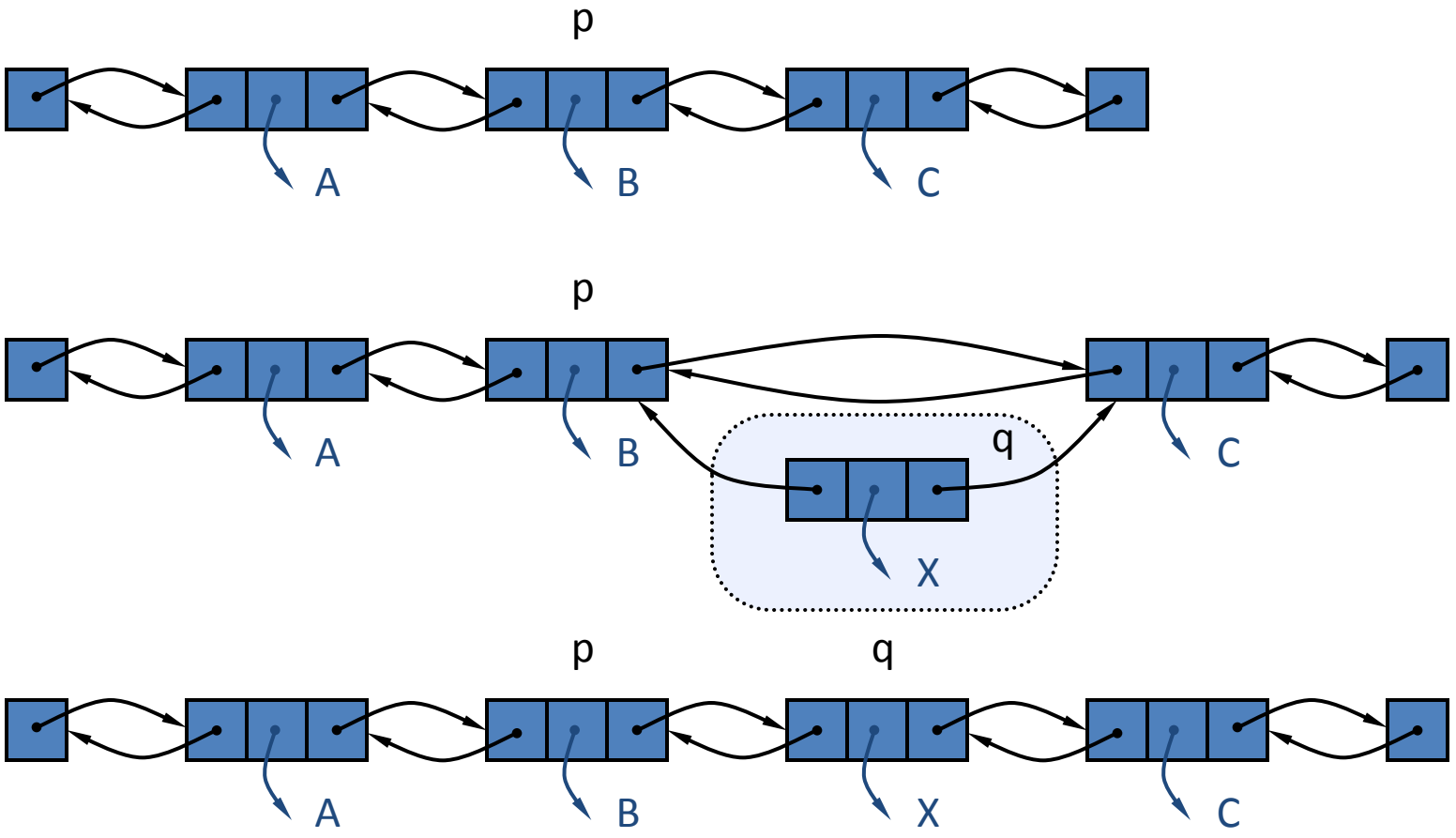


Removal at the Tail: Algorithm

```
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*).

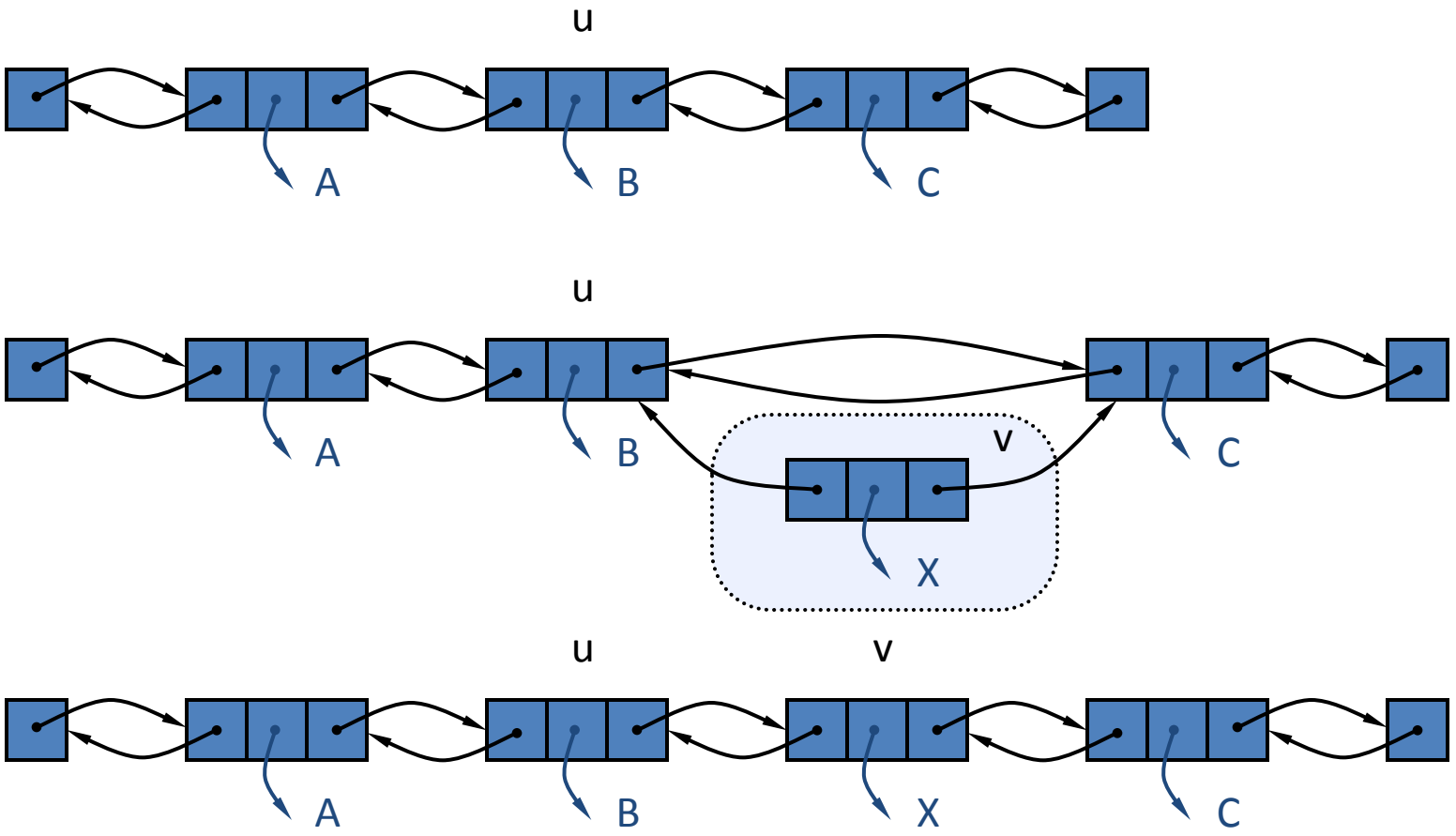


Insertion Algorithm

```
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.131 (p.125 in 4th edition).

- Homework: re-do the implementation without using the header and trailer sentinels.