4CCS1DST – Data Structures

Lecture 5:

Lists (§ 6.1, 6.2)

Lists

- A collection **S** of elements stored in a certain linear order.

S: E1, E2, E3, ... , Ei, ... , En

0 1 2 n–1

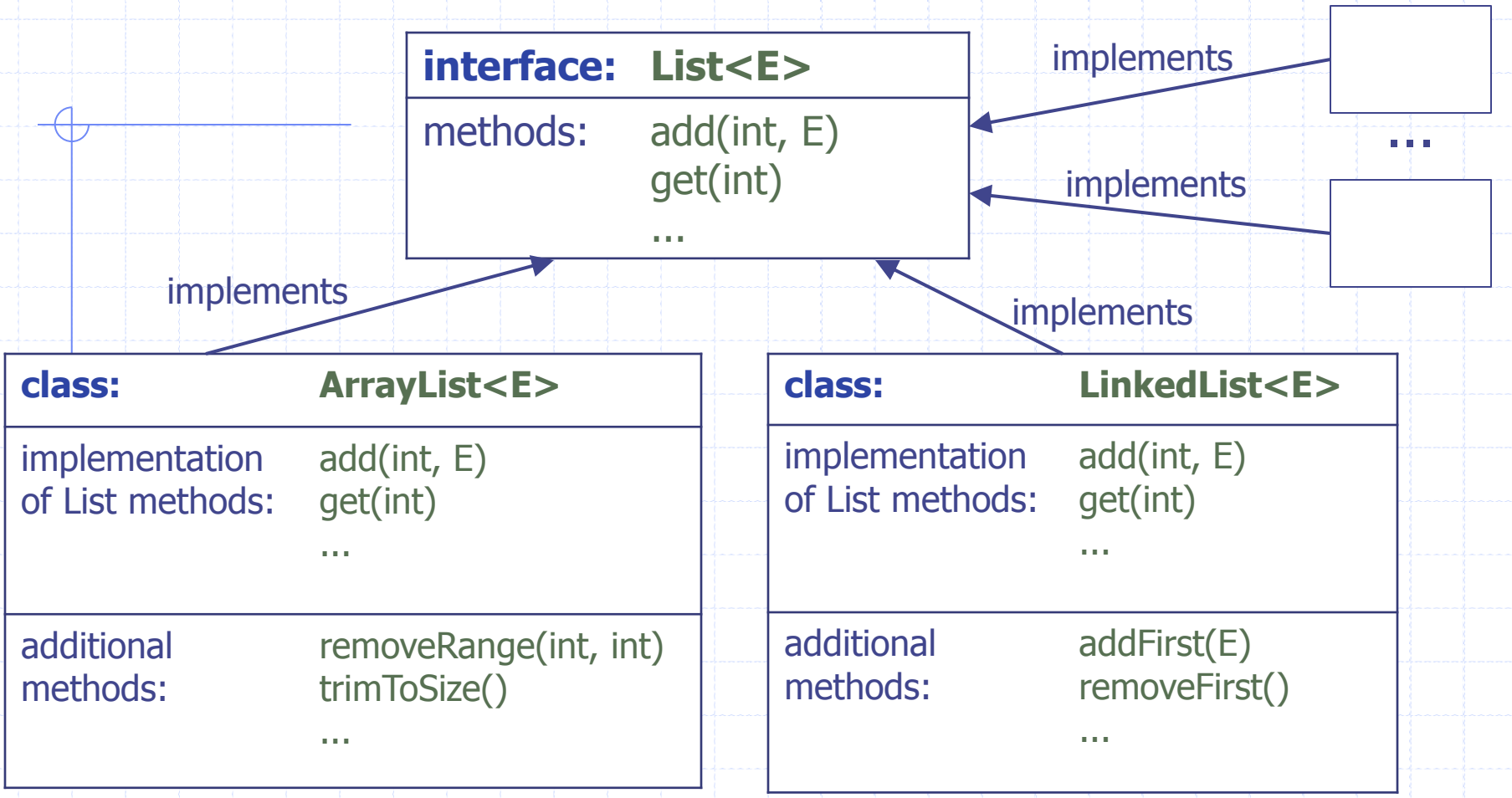
- We can refer to the elements in **S** as first, second, third, ...
- The **index** of an element **e** in **S** is the number of elements that are before **e** in **S**.
- The index can be used to specify where to insert a new element into the list, or where to remove an old element from.
- Stack and Queue can be viewed as **restricted** Lists.

Interface List<E> in Java Collections Framework

see:

http://docs.oracle.com/javase/8/docs/api/java/util/List.html

Lists in Java Collections Framework



Our Lists: ADTs, interfaces, classes

Array List ADT:

add(i,e), get(i), remove(i), ...

specifies

interface:	**IndexList<E>**
methods:	add(int, E) get(int) remove(int) ...

implements

class:	**ArrayIndexList<E>**
implementation of IndexList methods:	add(int, E) get(int) ...

array

Node List ADT:

addFirst(e), addBefore(p.e), ...

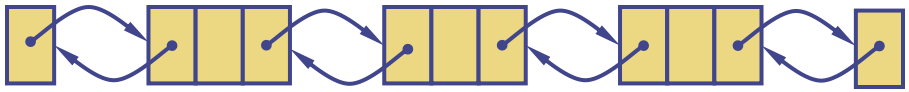
specifies

interface:	**PositionList<E>**
methods:	addFirst(E) addBefore(Position, E) remove(Position) ...

implements

class:	**NodePositionList<E>**
implementation of IndexList methods:	addFirst(E) addBefore(Position, E) ...

linked list



Array List ADT

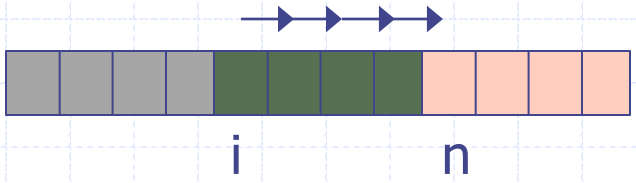
- An instance: a sequence S of elements
- Methods:
  - size(): return the number of elements in S;
  - isEmpty(): return true if S has no elements, otherwise false;
  - get(i): return the element of S with index i;
    an error condition occurs if i < 0 or i > size() – 1;
  - set(i,e): replace with e the element at index i and return the old element at this index;
    an error condition occurs if i < 0 or i > size() – 1;
  - add(i,e): insert a new element e into S to have index i;
    an error condition occurs if i < 0 or i > size();
  - remove(i): remove from S, and return, the element at index i;
    an error condition occurs if i < 0 or i > size() – 1.

Array List: example

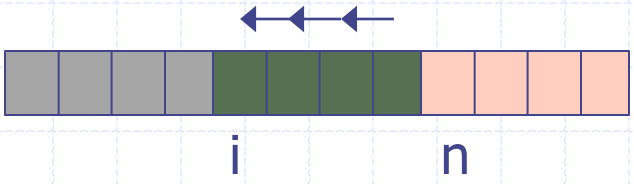
Operation	*Output*	*S*
add(0,5)	–	(5)
add(0,3)	–	(3, 5)
get(1)	*5*	(3, 5)
add(2,7)	–	(3, 5, 7)
get(3)	*"error"*	(3, 5, 7)
remove(1)	*5*	(3,7)
add(1,6)	–	(3, 6, 7)
add(1,7)	–	(3, 7, 6, 7)
add(4,9)	–	(3, 7, 6, 7, 9)
get(2)	*6*	(3, 7, 6, 7, 9)
set(3,8)	*7*	(3, 7, 6, 8, 9)
size()	*5*	(3, 7, 6, 8, 9)

Array-based implementation of Array List ADT

- add(i,e): shift up all elements with indices >= i to make room for the new element.



- remove(i): shift down all elements with indices >= i+1.



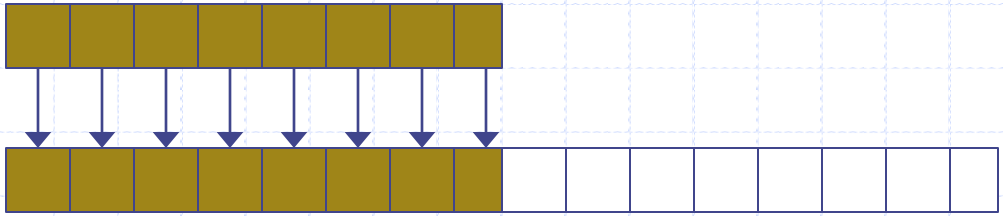
Algorithm *add*(*i*, *e*):
  for $j = n-1, n-2, \ldots, i$ **do**
    $A[j+1] \leftarrow A[j]$
  $A[i] \leftarrow e$
  $n \leftarrow n+1$

check first for error conditions!

Algorithm *remove*(*i*):
  $e \leftarrow A[i]$
  for $j = i, i+1, \ldots, n-2$ **do**
    $A[j] \leftarrow A[j+1]$
  $n \leftarrow n-1$
  return e

Extending full arrays (1)

- Inserting a new element to the list when the array is full: allocate a new larger array and copy all elements to the new array.



- What should be the capacity of the new array?
- Consider the following sequence of operations:
  start with an empty list and perform n "add" operations, adding each new element at the end of the list.
- If we keep increasing the capacity of a full array by 1, then the running time of this computation is

$$\Theta(1+2+3+ \ldots + n) = \Theta(n^2).$$

Extending full arrays (2)

- Increase the capacity by a fixed constant c and the running time is still $\Theta(n^2)$: $c+2c+3c+ \ldots +n = c(1+2+\ldots+n/c) \approx n^2/(2c) = \Theta(n^2)$.
- **Double the capacity**, then the running time is
  - $\Theta(n)$, for adding the elements when there is room in the array,
  - plus the time spent on increasing the array,

  which is (assuming that the initial capacity of the array is 2):

$$\Theta(4 + 8 + 16 + 32 + \ldots + N)$$

$$= \Theta(2N - 4) = \Theta(N) = \Theta(n),$$

$1+2+4+8+ \ldots + 2^k = 2^{k+1} - 1$

where $N = 2^k$ is the final capacity of the array; $n \leq N < 2n$.

- Therefore, when the array is full and the operation "add" is performed, we double the capacity of the array.

Interface IndexList<E>

```
public interface IndexList<E> {
  public int size();              // returns the number of elements in this list
  public boolean isEmpty();       // returns whether the list is empty
  public void add(int i, E e) throws IndexOutOfBoundsException;
                                  // inserts an element e to be at index i, shifting
                                  // all elements after this
  public E get(int i) throws IndexOutOfBoundsException;
                                  // returns the element at index i, without removing it
  public E remove(int i) throws IndexOutOfBoundsException;
                                  // removes and returns the element at index i,
                                  // shifting the elements after this
  public E set(int i, E e) throws IndexOutOfBoundsException;
                                  // replaces the element at index i with e, returning
                                  // the previous element at i
}
```

Class ArrayIndexList<E>

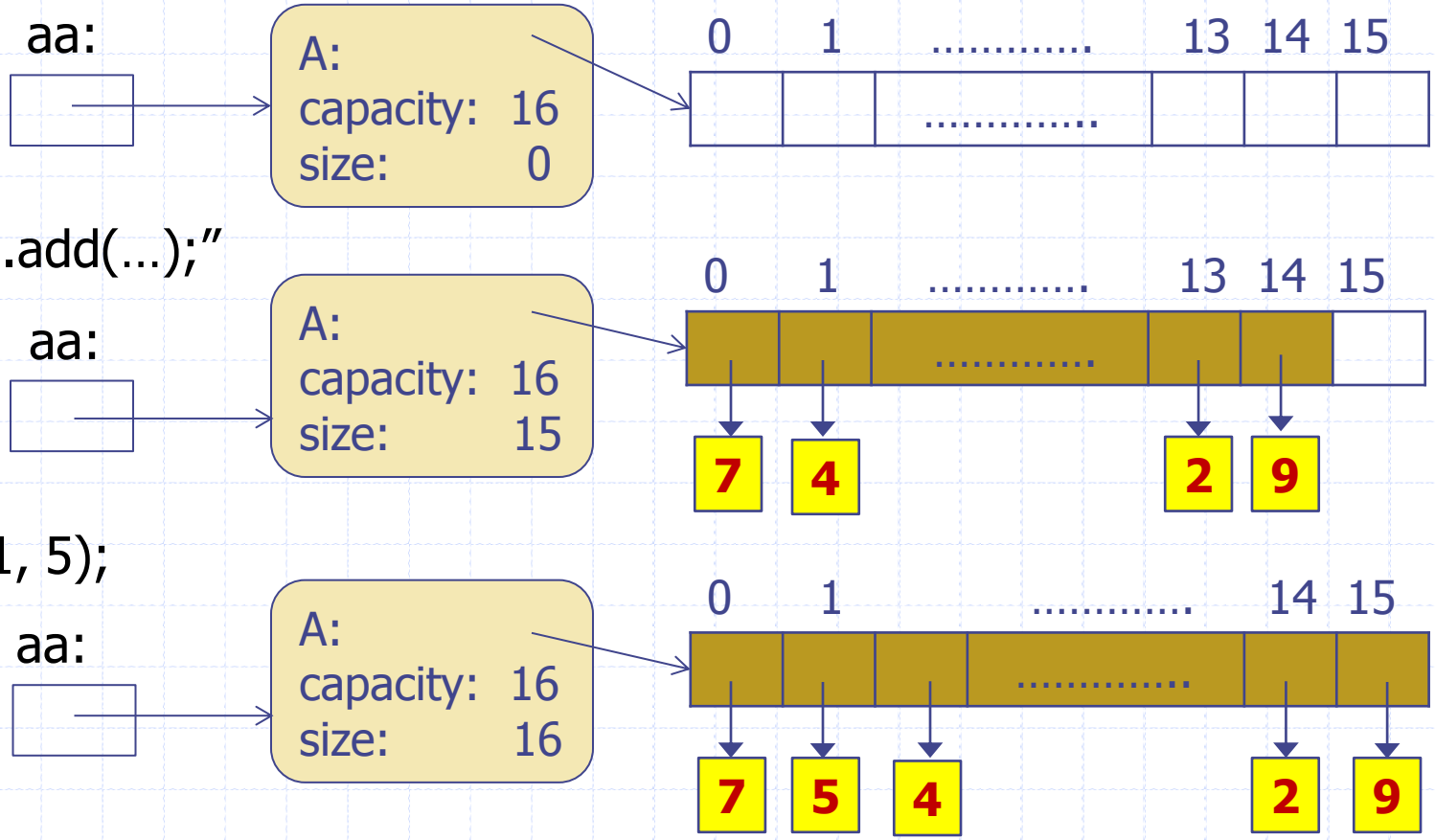
```
public class ArrayIndexList<E> implements IndexList<E> {
    private E[ ] A;                 // array storing the elements of the indexed list
    private int capacity = 16;      // the current capacity (length) of array A
    private int size = 0;           // number of elements stored in the indexed list

    public ArrayIndexList() {
      A = (E[ ]) new Object[capacity];  // initial array with the default capacity 16
    }

    // implementations of the IndexList methods
    public void add(int r, E e) throws IndexOutOfBoundsException { ... }
    ...

    // an auxiliary method; checks if the given index is in the range [0,n-1]
    protected void checkIndex(int r, int n) throws IndexOutOfBoundsException
      { if ( r < 0 || r >= n ) throw new IndexOutOfBoundsException(" ... "); }
}
```

Method add(int r, E e)

```
/** Inserts an element at the given index. */
public void add(int r, E e) throws IndexOutOfBoundsException {
    checkIndex(r, size() + 1);
    if ( size == capacity ) {                          // an overflow
        capacity *= 2;
        E[ ] B = (E[ ]) new Object[capacity];          // create a new array
        for (int i=0; i<size; i++) { B[i] = A[i]; }    // copy all elements
        A = B;                                         // the new array becomes the list array
    }
    for (int i=size – 1; i>=r; i--) { A[i+1] = A[i]; } // shift elements up
    A[r] = e;                                          // insert the new element
    size++;                                            // update the size of the list
}
```

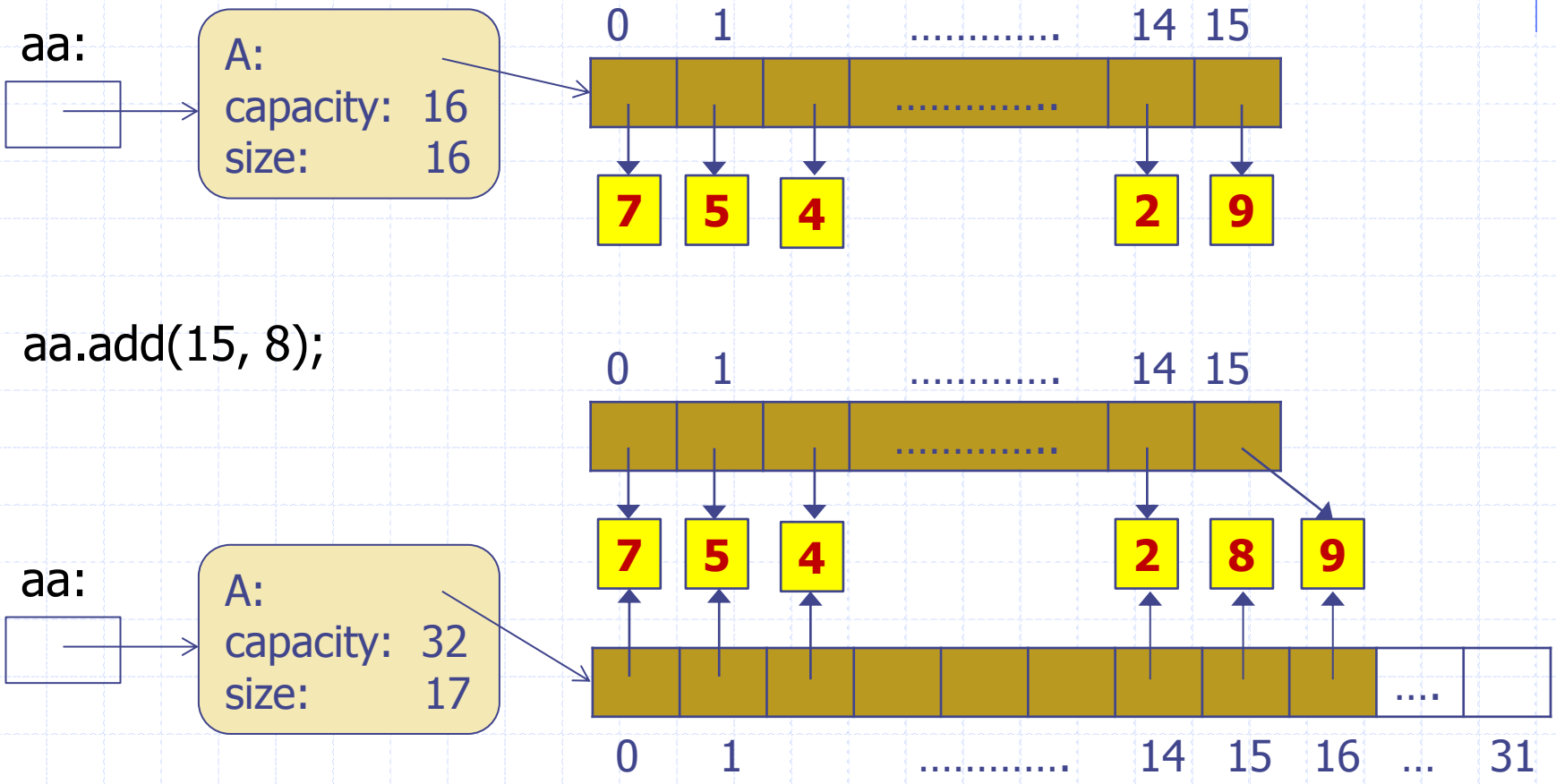
Method add(int r, E e): example

IndexList<Integer> aa = new ArrayIndexList<Integer>();



Method add(int r, E e): example, cont.

current aa:



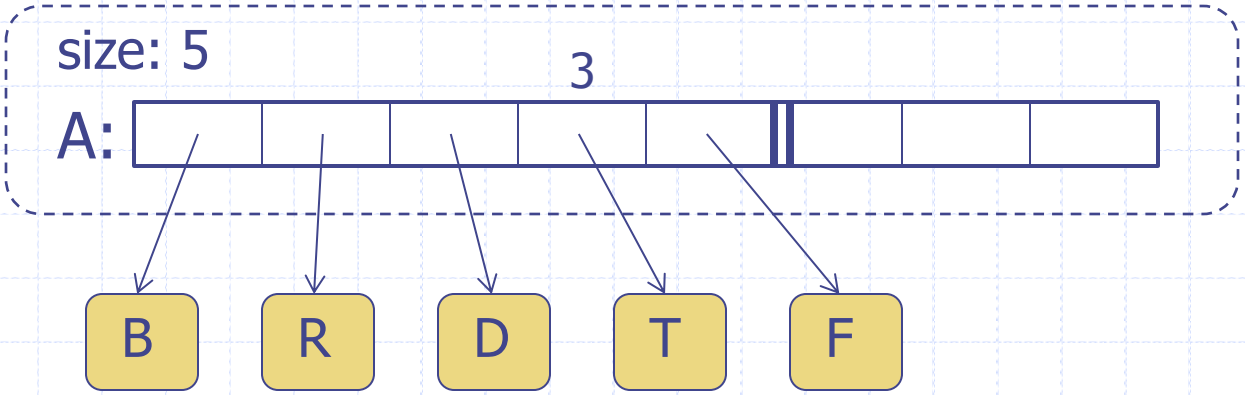
Performance

- In the array-based implementation of the Array List ADT, if the list has n elements:
  - The space used is $O(n)$.
  - Operations size, isEmpty and get and set run in $O(1)$ time (constant time).
  - Operation remove(i) runs in $O(n)$ time, or, more precisely in $\Theta(n-i)$ time.
  - Removing the last element takes $O(1)$ time.
  - Operation add(i,e) runs in $O(n)$ time.
  - Adding a new element at the end of the list takes $\Theta(n)$ time in the worst case (full array), but only $O(1)$ time on average.

Array list versus Node list

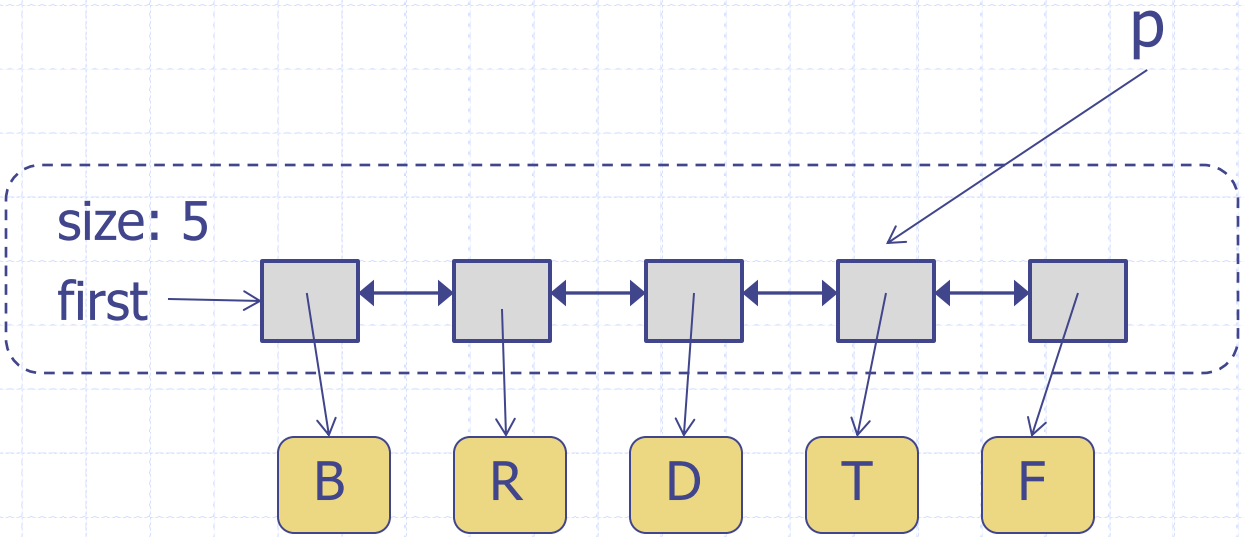
Array list:

element T is at index 3



Node list:

element T is at position p



Position

- Position: a place within a list where a single element is stored.

  In implementations, positions are nodes of lists
  (the references to nodes).

  If we know the position of some element, we might not know what the index of this element is, and knowing the index of an element doesn't mean that we know the "position" of this element.

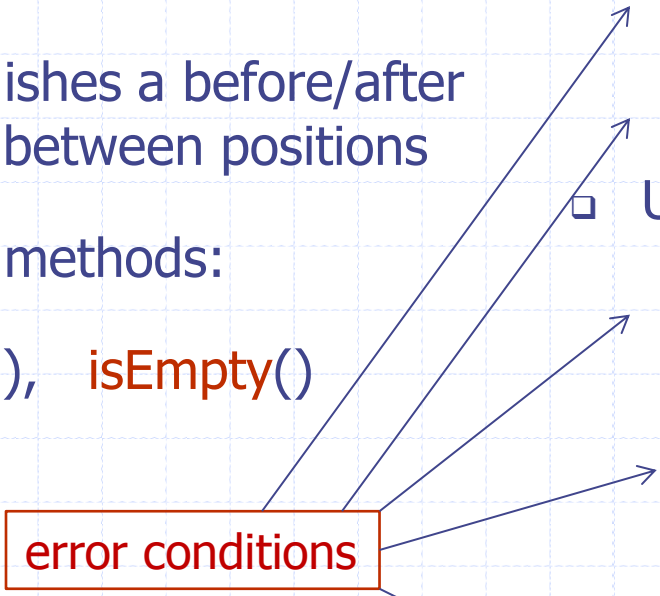
- The notion of "position" is also used in more complex node-based data structures (for example in trees).

- A user of Node List should only use the "position" of an element:
  - to get the element of the list which is stored at this position:

    p.element(): returns the element stored at position p;
  - as an argument in list operations, for example: addAfter(**p**, e).

Node List ADT

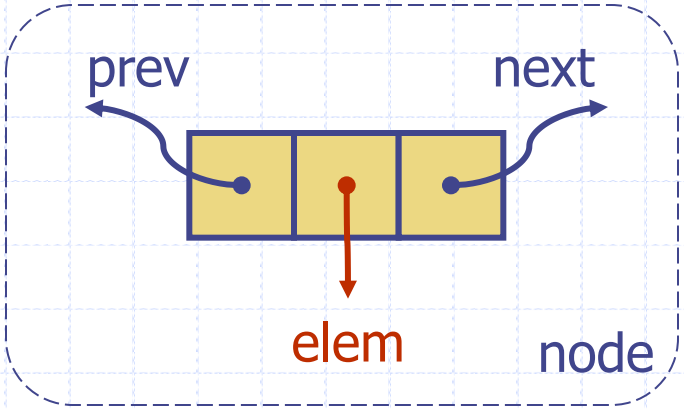
- The Node List ADT models a sequence of positions storing elements of some (arbitrary) type.
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()

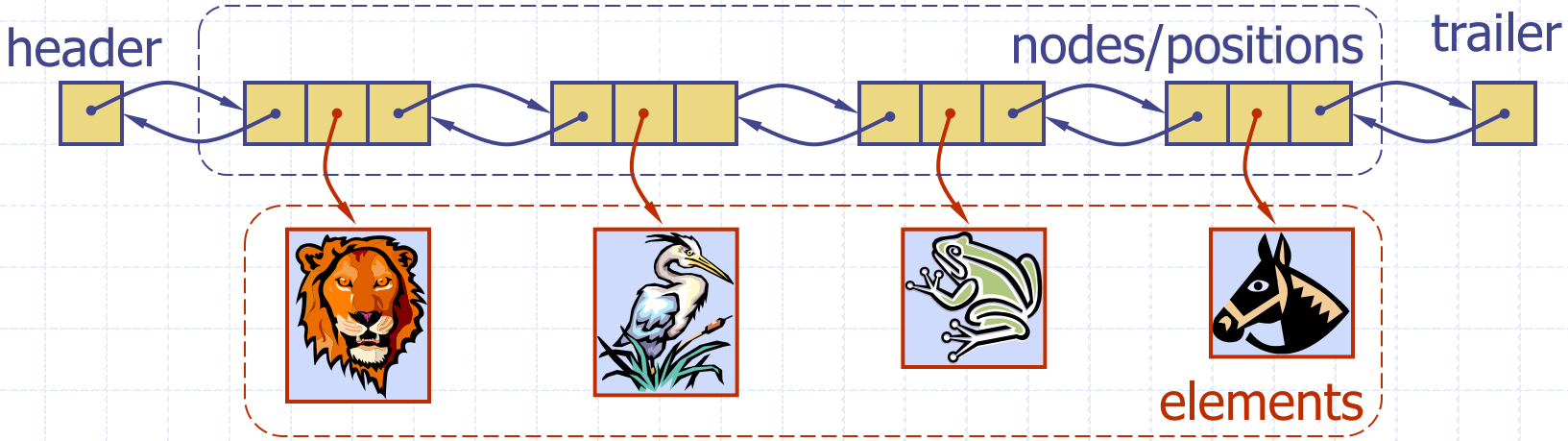


- The argument "p" is a position
- Accessor methods, return the position:
  - first(), last()
  - prev(p), next(p)
- Update methods:
  - set(p, e) - replace the element at p with e, return old element
  - remove(p) - remove and return the element at p
  - addFirst(e), addLast(e)
  - addBefore(p, e), addAfter(p, e)

Doubly Linked List

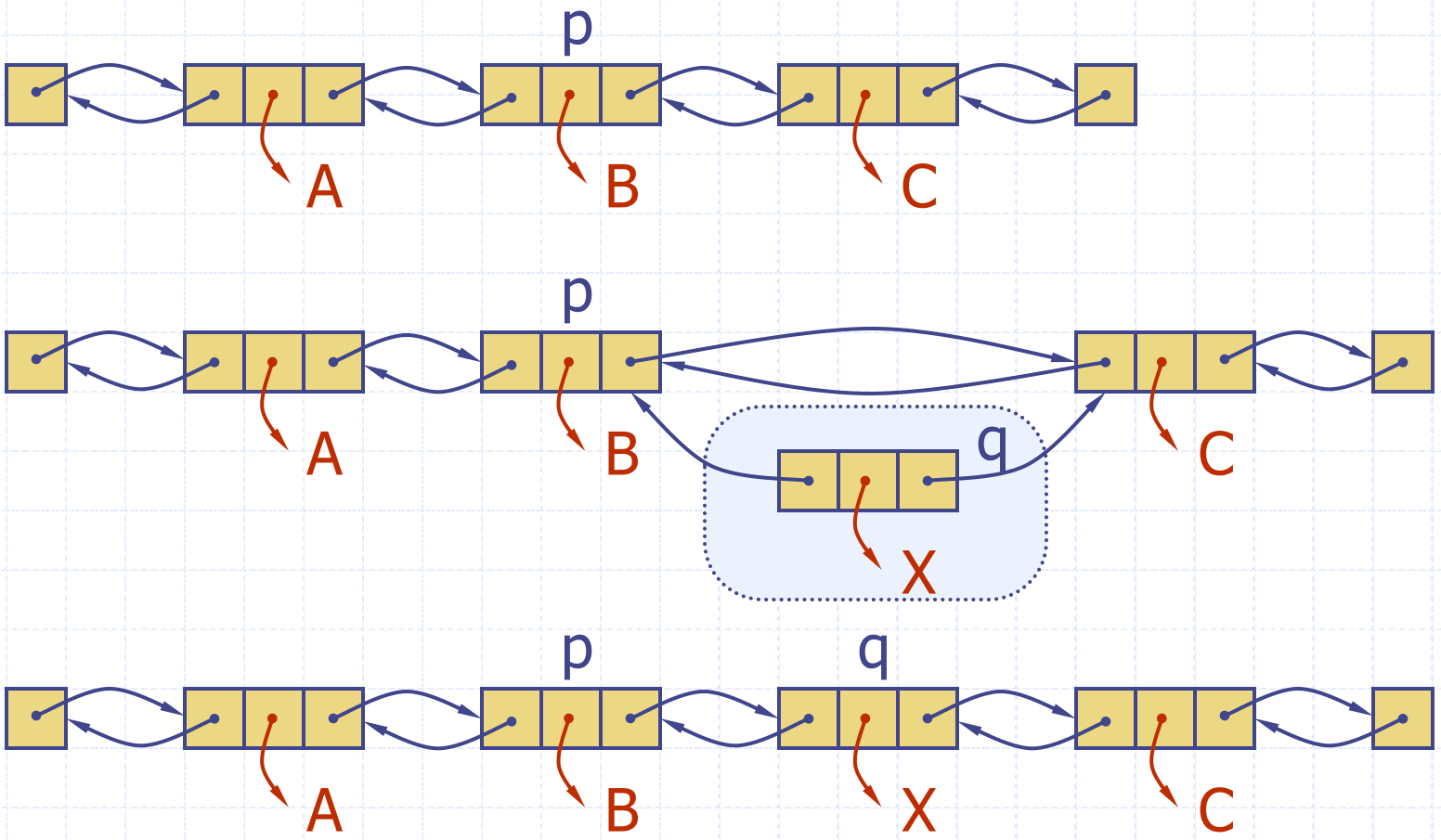
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link (reference) to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation addAfter(p, X)



Insertion Algorithm

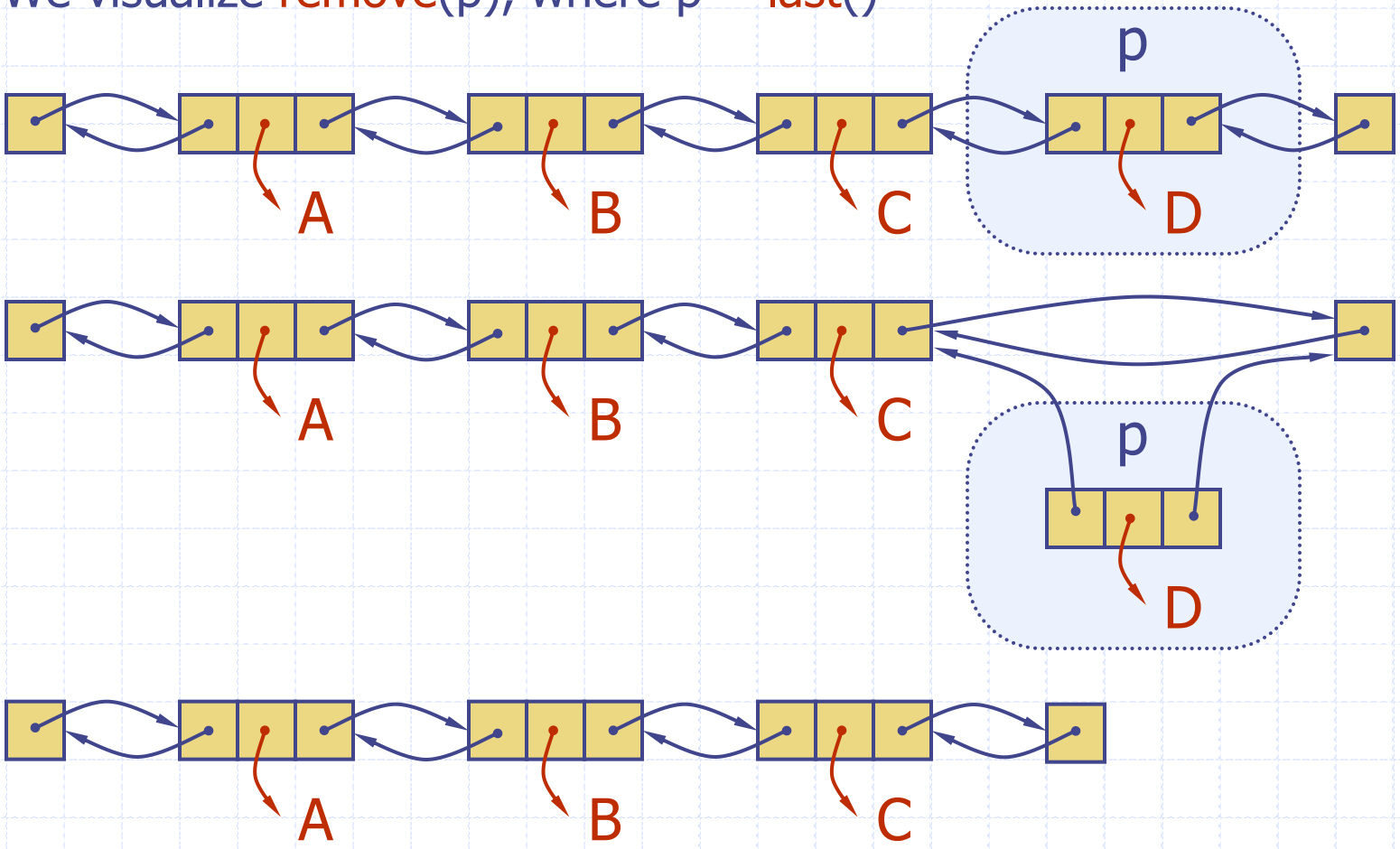
Algorithm addAfter(p,e):

Create a new node v

```
v.setElement(e)              { put e in the new node v }
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}

numberElements++
```

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**)

numberElements++

return t

Performance

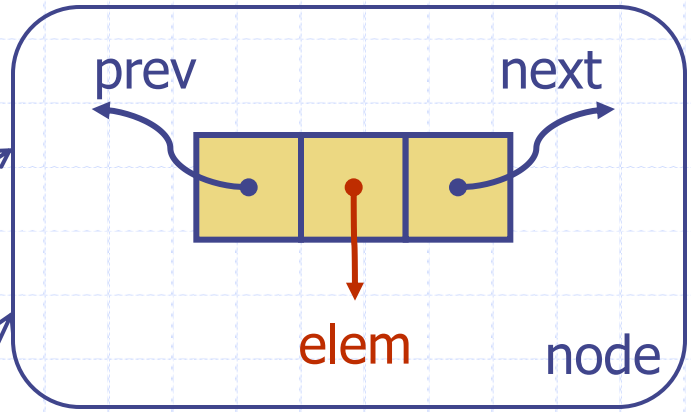
- In the implementation of the Node List ADT by means of a doubly linked list:
  - The space used by each node of the list is $O(1)$
  - The space used by a list with n elements is $\Theta(n)$
  - All the operations of the Node List ADT run in $O(1)$ time
  - The position operation element() runs in $O(1)$ time
  - No efficient implementation of operations which refer to the index of an element of a list.

    For example, no efficient implementation of "get(i)"

Interface Position<E>

```
public interface Position<E> {
    E element();          // return the element stored at this position
}
```

User of List, can only get the element from Node



p
(of type Position)

Implementation of List, access to the internal structure of Node

v
(of type DNode<E>, which implements Position)

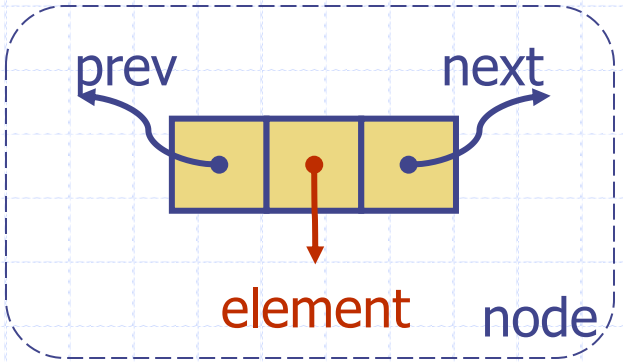
```
v = (DNode<E>) p;
```

Interface PositionList<E>

```
public interface PositionList<E> extends Iterable<E> {
    public int size();
    public boolean isEmpty();

    public Position<E> first() throws EmptyListException;  // returns the first node
    public Position<E> last() throws EmptyListException;   // returns the last node

    public Position<E> next(Position<E> p)
            throws InvalidPositionException, BoundaryViolationException;
    public Position<E> prev(Position<E> p) throws ... ;

    public void addFirst(E e);  // inserts an element at the front of the list
    public void addLast(E e);   // inserts an element at the back of the list
    public void addAfter(Position<E> p, E e) throws InvalidPositionException;
    public void addBefore(Position<E> p, E e) throws InvalidPositionException;

    public E remove(Position<E> p) throws InvalidPositionException;
    public E set(Position<E> p, E e) throws InvalidPositionException;

    // not discussed:
    public Iterable<Position<E>> positions();   public Iterator<E> iterator();    }
```

Implementation of a Node (1)

```
public class DNode<E> implements Position<E> {
    private  DNode<E>  prev,  next;      // references to the nodes before and after
    private  E  element;                 // element stored in this position

    // constructor
    public  Dnode (DNode<E> newPrev,  DNode<E> newNext,  E elem) {
            prev = newPrev;
            next = newNext;
            element = elem;
    }

    // method from interface Position
    public  E  element()  throws  InvalidPositionException {
            if ( (prev == null )  &&  ( next == null ) )
                        throw new InvalidPositionException("Position is not in a list!");
            return element;
    } ..... // class Dnode<E> continues on the next slide
```

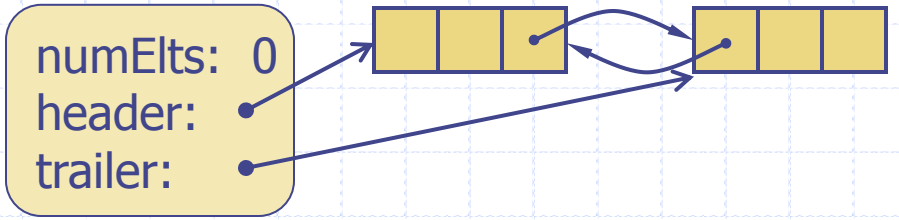


Implementation of a Node (2)

```
public class DNode<E> implements Position<E> {
    private  DNode<E>  prev,  next;      // references to the nodes before and after
    private  E  element;                 // element stored in this position

    // constructor
    public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {  ……  }

    // method from interface Position
    public  E  element()  throws  InvalidPositionException {  ……  }

    // accessor methods
    public DNode<E> getNext() { return next; }
    public DNode<E> getPrev() { return prev; }

    // update methods
    public void setNext(DNode<E> newNext) { next = newNext; }
    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
    public void setElement(E newElement) { element = newElement; }
}
```

Class NodePositionList<E> (1)

```
public class NodePositionList<E> implements PositionList<E> {
    protected int numElts;                    // number of elements in the list
    protected DNode<E> header, trailer;       // special sentinels

    // constructor
    // that creates an empty list
    public NodePositionList() {
        numElts = 0;
        header = new DNode<E>(null, null, null);          // create header
        trailer = new DNode<E>(header, null, null);       // create trailer
        header.setNext(trailer);     // header and trailer to point to each other
    }

    // checks if p is a valid position for this list and, if valid, converts it to DNode
    protected DNode<E> checkPosition(Position<E> p)
        throws InvalidPositionException { ..... }

    .....
```

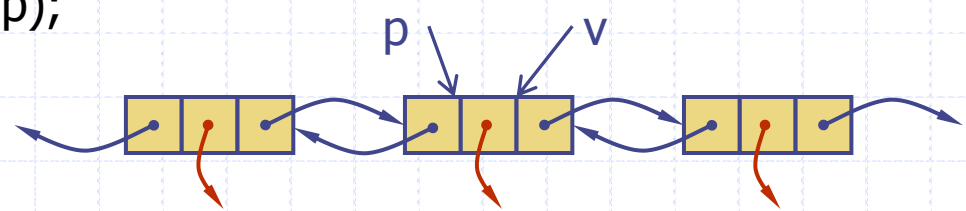


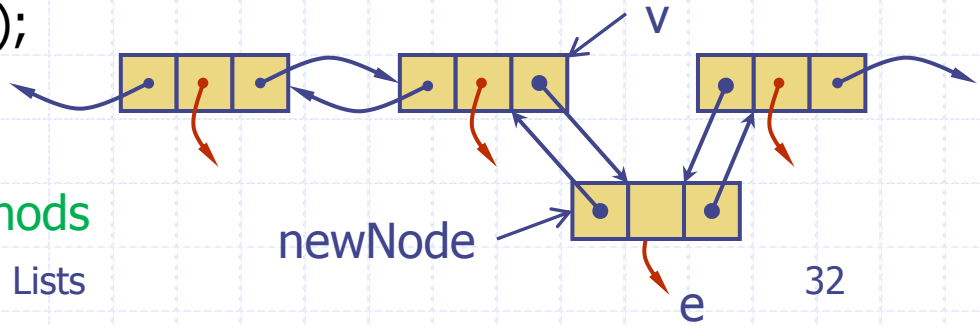
From "Position" to "DNode"

```
// checks if p is a valid position for this list and, if valid, converts it to DNode
protected  DNode<E>  checkPosition(Position<E> p)
          throws  InvalidPositionException {

     if (p == null)  throw new InvalidPositionException
                                   ("Null position passed to NodeList");
     if (p == header)  throw new InvalidPositionException
                                     ("The header node is not a valid position");

     ......
     DNode<E>  temp  =  (DNode<E>)  p;
     ......
     return temp;
}
```

Class NodePositionList<E> (cont.)

```
// returns the first position in the list
public Position<E> first() throws EmptyListException {
        if ( isEmpty() ) throw new EmptyListException("List is empty");
        return header.getNext();
}
// insert the given element after the given position
public void addAfter(Position<E> p, E e) throws InvalidPositionException {
    DNode<E> v = checkPosition(p);
    numElts++;


    DNode<E> newNode = new DNode<E>(v, v.getNext(), e);
    v.getNext().setPrev(newNode);
    v.setNext(newNode);
}
// implementations of other methods
```





Example

```
PositionList<Integer>  L1 =  new  NodePositionList<Integer>();
L1.addFirst(5);
L1.addFirst(9);
L1.addBefore(L1.last(),3);
…
// print the elements of list L1; assuming L1 is not empty
Position<Integer>  p = L1.first();     // p will access consecutive positions of L1
System.out.print( p.element() + " ");
for ( int i = 0;  i < L1.size()-1;  i++ ) {
    p = L1.next(p);
    System.out.print( p.element() + "  " );
}
```

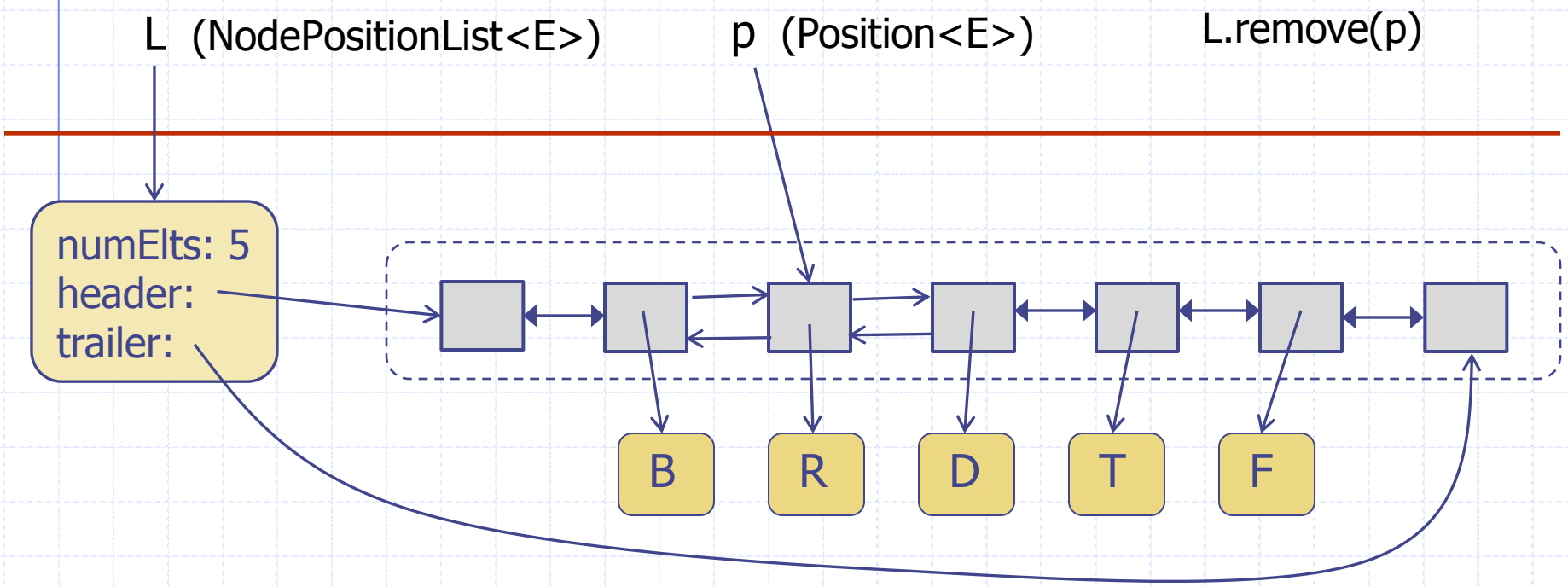
Exercises

Exercise 1

Give the code for method remove(p) in class NodePositionList.

Diagram for Exercise 1: Node List implementation

User's application



Node-List implementation

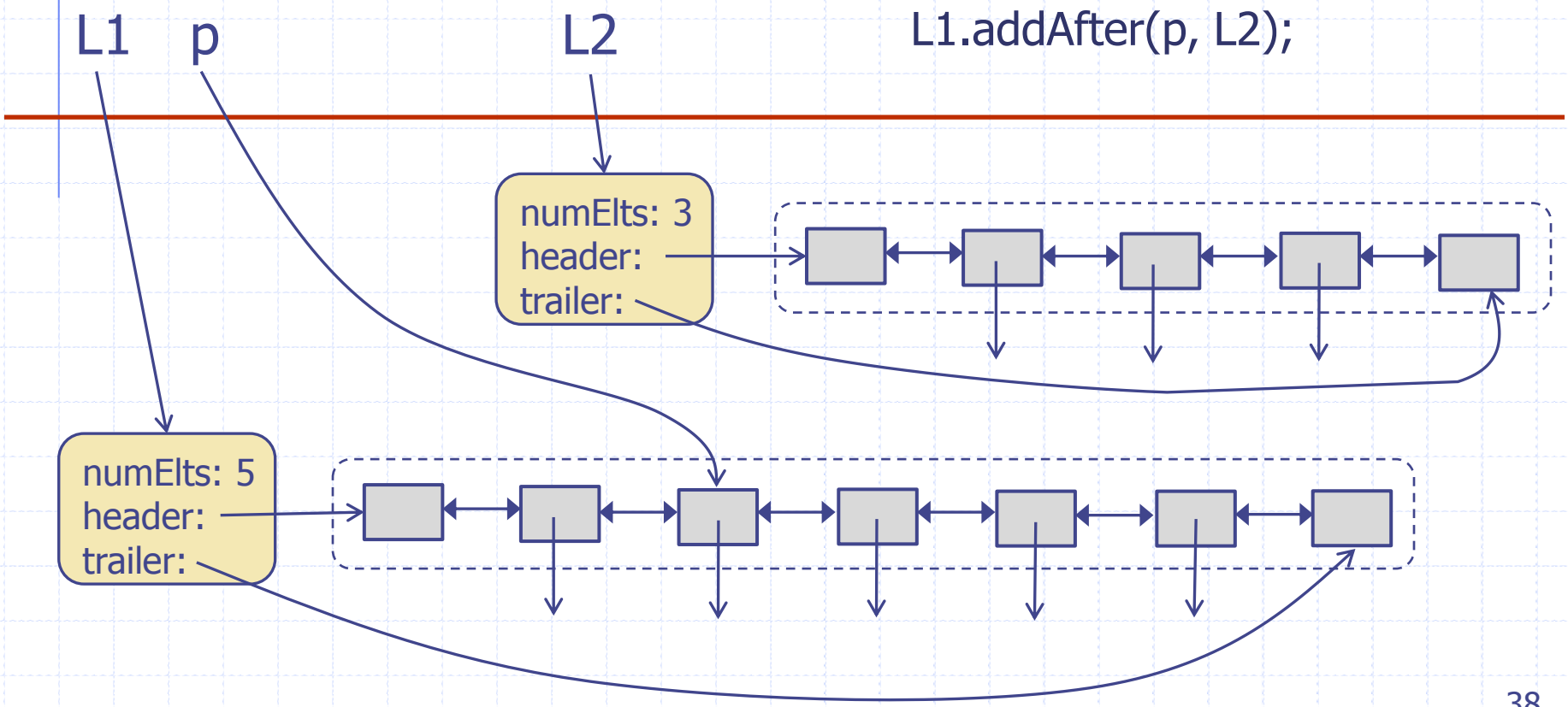
Exercise 2

Give the code for method addAfter(p, L) in the following extension of class NodePositionList. This method inserts list L to "this" list after the position p. The nodes from L should be put directly into "this" list. (Don't create new nodes.)

```
public class NodePositionListPlus<E> extends NodePositionList<E> {
    public void addAfter(Position<E> p, NodePositionListPlus<E> L)
            throws InvalidPositionException {  …..  }  // give your code here

    public static void main(String[] args) {
       NodePositionListPlus<Integer> L1 = new NodePositionListPlus<Integer>();
       NodePositionListPlus<Integer> L2 = new NodePositionListPlus<Integer>();
       L1.addLast(6);  L1.addLast(7);  L2.addLast(1);  L2.addLast(2);
       L1.addAfter(L1.first(), L2);
       System.out.print("L1 has " + L1.size() + " elements: ");
       // class NodePositionList has an appropriate toString() method
       System.out.println(L1); // prints: "L1 has 4 elements: [6, 1, 2, 7]"
    }
}
```

Diagram for Exercise 2

```
public void addAfter(Position<E> p, NodePositionListPlus<E> L)
        throws InvalidPositionException { .... }
```



Exercise 3

Consider the following class, which uses lists from Java Collections Framework.

```
import  java.util.*;  // we're using lists Java Collections Framework
public  class  ListTester {
    // count even numbers in a sequence of integers
    public  static  int  countEven(List<Integer>  seq) {
            int c = 0;
            for (int i = 0; i < seq.size(); i++) {
                    if ( seq.get(i) % 2 == 0 ) {  c++; }
            }
            return c;
    }
    public  static  void  main(String[] args) {
            List<Integer> seqArr = new  ArrayList<Integer>();
            List<Integer> seqLL =  new  LinkedList<Integer>();
            // continues on the next slide
```

Exercise 3 (cont.)

```
    for (int i = 0; i < 200000; i++) {
            int n = (int) (Math.random() * 100);
            seqArr.add(n);              // appends n at the end of the list seqArr
            seqLL.add(n);               // appends n at the end of the list seqLL
     }
    long  startTime  =  System.currentTimeMillis();
    int  c1 = countEven(seqArr);
    long  elapseTime = System.currentTimeMillis() – startTime;
    System.out.println( ...   c1  ...  elapseTime  ... );
    ... // similarly, run and time  countEven(seqLL)
  }
}
```

When run on sequences of 200,000 numbers:

count even numbers in array: 99704/200000 [0.007s]
count even numbers in linked list: 99704/200000 [26.971s]

Explain the difference in the running time.
Predict the running times for sequences of 400,000 numbers.