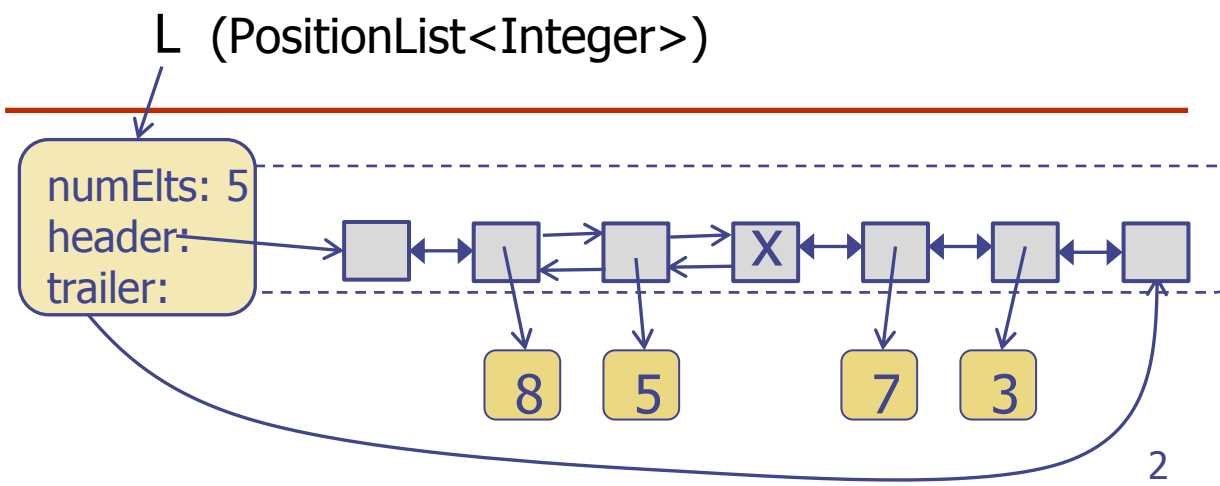
4CCS1DST – Data Structures

Exercises for Lecture 5

Exercise 0

Give the code for the following method:

```
public static void addToAll(PositionList<Integer> L, int k) {
    // add k to each number in the list L
```



Exercise 0 (cont.)

Answer:

```
public static void addToAll(PositionList<Integer> L, int k) {
// add k to each number in the list L

        if ( (L != null) && !L.isEmpty() ) {

                Position<Integer> p;
                p = L.first(); // start from the first position in the list

                for (int i=0; i < L.size(); i++) {
                        if ( p.element() != null ) {   // add k
                                L.set(p, p.element() + k);
                        }
                        if ( p != L.last() )  {          // move to the next position
                                p = L.next(p);
                        }
                }
        }
}
```

Exercise 1

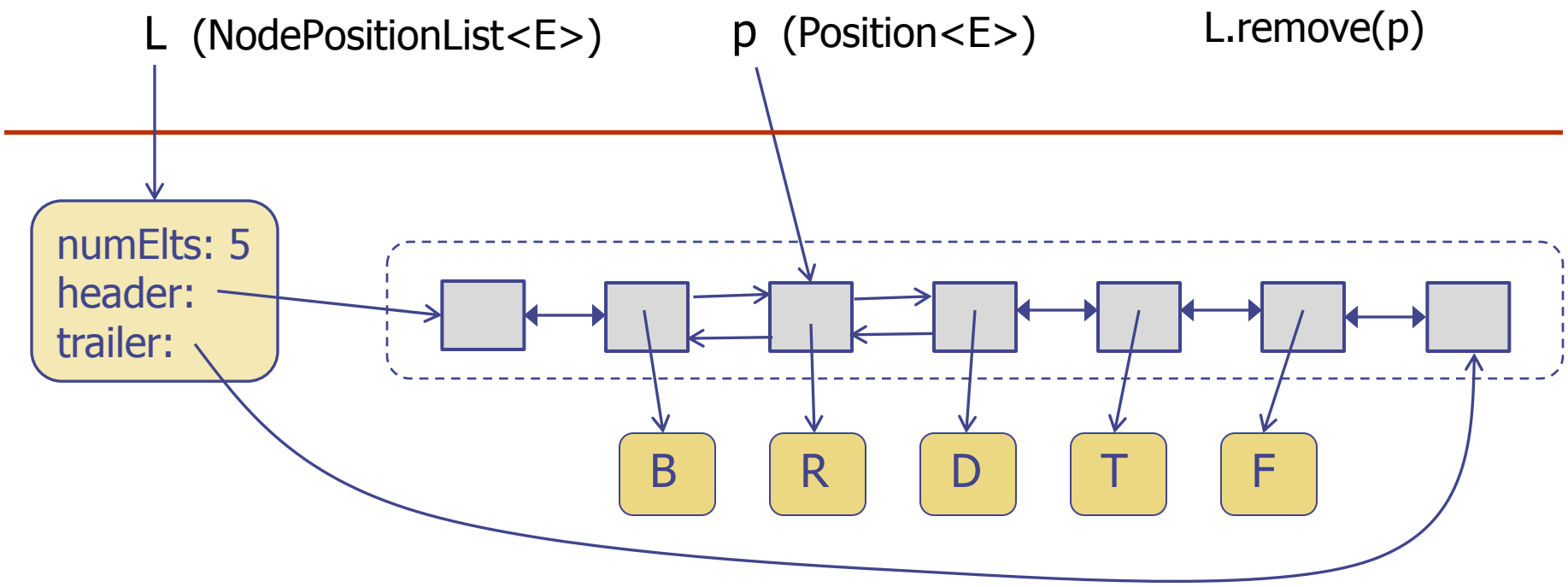
Give the code for method remove(p) in class NodePositionList.

Answer:

```
public E  remove(Position<E> p)  throws  InvalidPositionException {

        DNode<E> v  =  checkPosition(p);
        // p is valid; v is the same reference as p, butof the type DNode<E>

        numElts--;

        // unlink the position from the list
        v.getPrev().setNext( v.getNext() );
        v.getNext().setPrev( v.getPrev() );

        // make the position invalid
        v.setNext(null);
        v.setPrev(null);
        E elem = v.element();
        v.setElement(null);

        return elem;
}
```

Diagram for Exercise 1: Node List implementation

User's application



Node-List implementation

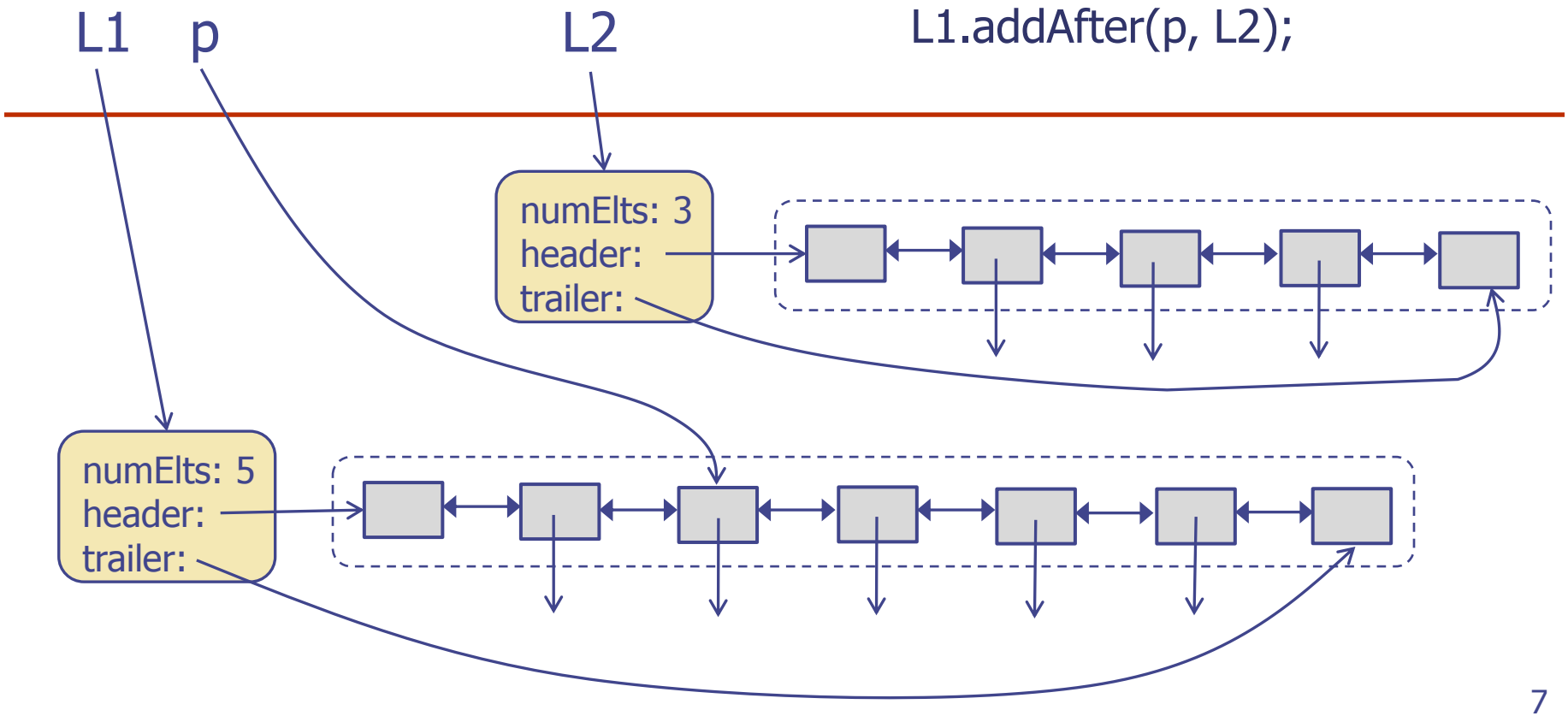
Exercise 2

Give the code for method addAfter(p, L) in the following extension of class NodePositionList. This method inserts the 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> list)
            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.println("L1 has " + L1.size() + " elements: ");
       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 2 (cont.)

Answer:

```
public void addAfter(Position<E> p, NodePositionListPlus<E> L)
        throws InvalidPositionException {
        DNode<E> v = checkPosition(p);
        // p is valid; v is the same reference as p, but cast to type DNode<E>

        if ( (L != null) && !L.isEmpty() ) {
                L.trailer.getPrev().setNext(v.getNext());
                L.header.getNext().setPrev(v);
                v.getNext().setPrev(L.trailer.getPrev());
                v.setNext(L.header.getNext());
                this.numElts += L.size();

                L.header.setNext(L.trailer);          // make L an empty list
                L.trailer.setPrev(header);
                L.numElts = 0;
        }
}
```

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.

Exercise 3 (cont.)

Answer:

The running time of the method countEven depends on the running time of the list method get(i). The running time of this method is constant for ArrayList (direct access to the i-th element of the list) and O(i) for LinkedList (i nodes of the list have to be traversed to reach the i-th element). Therefore, the running time of the method countEven is O(n) for ArrayList, and O(1 + 2 + ... + i + ... + n) = O(n^2) time for LinkedList. This difference in the running time (linear for ArrayList, but quadratic for LinkedList) explains the huge difference in the observed running times.

If the running time is linear (that is, O(n)), then if the input size doubles, then we would expect that the observed running times double as well.

If the running time is quadratic (that is, O(n^2)), then if the input size doubles, then we would expect that the observed running times increase 4 times.

Thus, for sequesnces of 400,000 numbers, we should expect the running times ~0.014 sec. for ArrayList, and ~110 sec. for LinkedList.