Fachhochschule
Nordwestschweiz

03 Iteratoren

Algorithmen und Datenstrukturen 2

Themen Heute

Iteratoren

ListIteratoren

Doppelt Verkettete Listen

Lernziele

Iteratoren

- Sie kennen die Vor- und Nachteile von verschiedenen Zugriffsarten auf die Elemente einer Collection (oder Set oder Liste).
- Sie programmieren einen Iterator zu einer Collection und zu einer Liste.
- Sie erklären mögliche Konflikte, wenn die Collection während der Iteration verändert wird und erläutern eine Lösung dafür.

Doppelt Verkettete Liste

- Sie programmieren eine Doubly Linked List und einen Iterator dazu.

Motivation

Ziele:

- Operation mit allen Elementen einer Collection durchführen
  - Summe aller Elemente
  - Elemente mit bestimmter Eigenschaft zählen
  - Löschen von Duplikaten
  - …
- Operation mit den ersten x Elementen einer Collection durchführen

Problem:

- geht nicht gut mit ausschliesslich add, contains, remove

```java
public interface Collection<E> {
   Object[] toArray();                          // includes new Object[size()]
   <T> T[] toArray(T[] a);                      // reuses a, if large enough, otherwise new T[]
   <T> T[] toArray(IntFunction<T[]> gen);       // calls gen to create an array to be filled

   Stream<E> stream();                          // create streams
   Stream<E> parallelStream();

   void forEach(Consumer<? super E> action);  // calls action for each element in collection

   Iterator<E> iterator();                      // creates iterator (Interface see below)
}
```

```java
public interface Iterator<E> {
   boolean hasNext();                  // checks if there are more elements available
   E next();                           // returns next element
   void remove();                      // removes least recently returned element
}
```

Beispiel: Verwendung eines Integer-Iterators

For-Loop

```
for (int e : c) {
    do something with e
}
```

While-Loop

```
Iterator<Integer> it = c.iterator();
while (it.hasNext()) {
    e = it.next();
    do something with e
}
```

Der erzeugte Byte-Code ist identisch.

Arbeitsblatt Aufgabe 1

a) `int sumOf(Collection<Integer> c)`

b) `void removeSecondElem(Collection<Integer> c, int v)`

- Benützen Sie dazu einen Iterator

c) Eigenschaften

Lösen Sie die Aufgaben zu zweit.

Begründen Sie Ihre Wahl für a) und b)!

sumOf - mit toArray

```
private static int sumToArray(Collection<Integer> c){
  int sum = 0;
  Integer[] A = c.toArray(new Integer[c.size()]);
  for(int i=0; i<A.length; i++) {
     sum += A[i];
  }
  return sum;
}
```

sumOf - mit forEach

```
private static final class Action implements Consumer<Integer> {
    int sum = 0;

    @Override
    public void accept(Integer e) {
        sum += e;
    }
}

private static int sumForEach(Collection<Integer> c) {
    Action action = new Action();
    c.forEach(action);
    return action.sum;
}
```

sumOf - mit stream und parallelstream

```
private static int sumStream(Collection<Integer> c) {
   return c.stream().mapToInt(Integer::intValue).sum();
}

private static int sumStreamReduce(Collection<Integer> c) {
   return c.stream().mapToInt(Integer::intValue).reduce(0, (x, y) -> x + y);
}

private static int sumParallelStream(Collection<Integer> c) {
   return c.parallelStream().mapToInt(Integer::intValue).sum();
}

private static int sumParallelStreamReduce(Collection<Integer> c) {
   return c.parallelStream().mapToInt(Integer::intValue).reduce(0, (x, y) -> x + y);
}
```

sumOf - mit Iterator

```
private static int sumForLoop(Collection<Integer> c) {
    int sum = 0;
    for (Integer e : c)
        sum += e;
    return sum;
}

private static int sumIterator(Collection<Integer> c) {
    int sum = 0;
    Iterator<Integer> it = c.iterator();
    while (it.hasNext()) {
        sum += it.next();
    }
    return sum;
}
```

removeSecondElem - mit Iterator

```
private static <T> void removeSecondElemIterator(Collection<T> c, T v) {
  int cnt = 0;
  Iterator<T> it = c.iterator();
  while (it.hasNext() && cnt < 2) {
     T e = it.next();
     if (e.equals(v)) {
        cnt++;
        if (cnt == 2) it.remove();
     }
  }
}
```

Einzige Möglichkeit, um frühzeitig abzubrechen und Collection direkt zu verändern.

Eigenschaft	toArray	stream	forEach	Iterator
Es werden immer alle Elemente der Collection bearbeitet. Vorzeitiger Abbruch, wie z.B. bei der sequenziellen Suche, ist nicht möglich.	x		x	
Unterstützt funktionales Programmieren mit immutable Collections.		x		
Es entsteht eine vollständige Kopie der Datenstruktur. Diese kann verändert werden, ohne dass die originale Datenstruktur davon beeinflusst wird.	x			
Einzelne bearbeitete Elemente können bei Bedarf direkt aus der Collection entfernt werden.			x	x
Der Aufwand sowohl für Speicher als auch für Laufzeit entspricht O(n), wobei n die Anzahl der Elemente in der Collection ist.	x			
Erlaubt unter gewissen Randbedingungen sehr leicht Multi-Core Prozessoren durch parallele Verarbeitung auszunutzen.		x		

ListIterator

```
public interface List<E> extends Collection<E> {
  ...
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);
}
```

Analog zu den Methoden `get(idx)`, `add(idx, e)`, `remove(idx)` gibt es einen `listIterator(idx)`, der direkt an einem bestimmten Index startet.

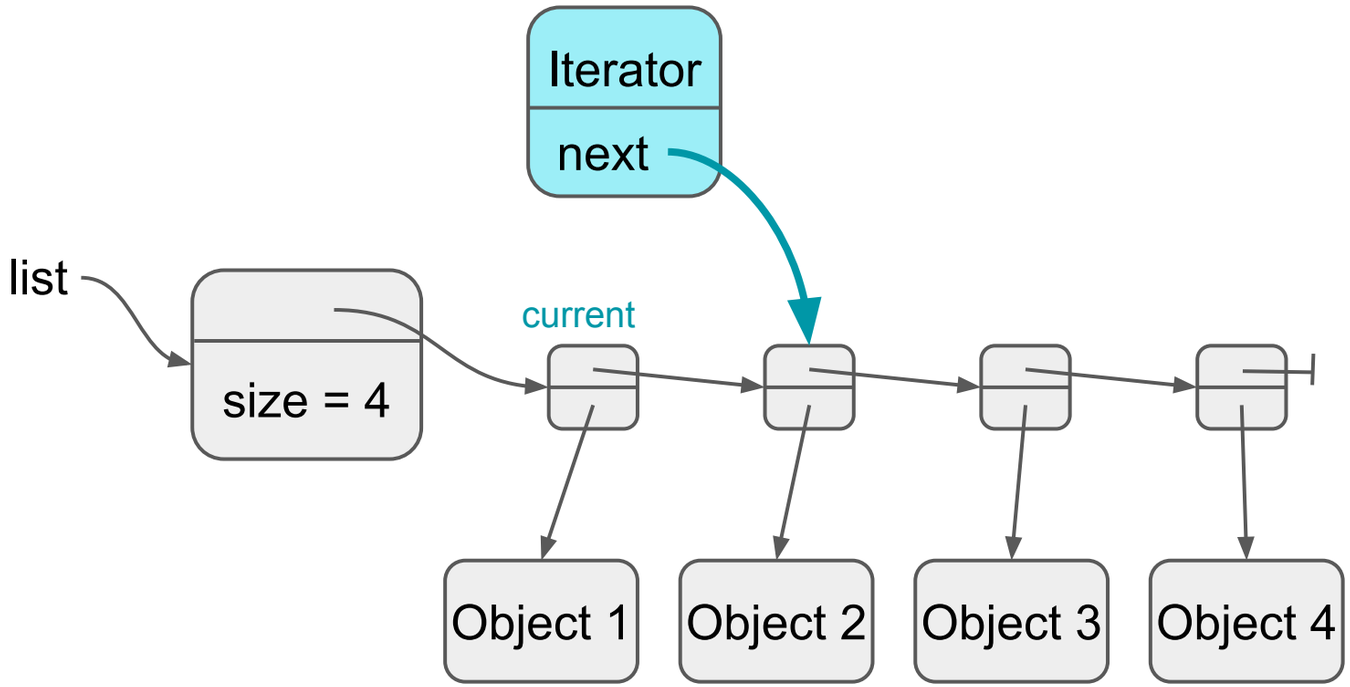
ListIterator

```
public interface ListIterator<E> extends Iterator<E> {
   // Query Operations
   boolean hasNext();         // as for simple Iterator
   E next();
   boolean hasPrevious();     // moves in opposite direction
   E previous();
   int nextIndex();           // returns position left and right of Iterator
   int previousIndex();

   // Modification Operations
   void remove();     // removes least recently (by next() or previous()) returned element
   void set(E e);     // replaces least recently returned element
   void add(E e);     // adds a new element at the current iterator position
}
```

Der `ListIterator` erlaubt das Iterieren in beide Richtungen. (Für den Moment betrachten wir nur “next”.)

Interner Iterator auf MyLinkedList

- Direkter Zugriff auf interne Struktur der Liste
- Interne Iterator-Klasse, die das Iterator-Interface implementiert
- next-Zeiger jeweils auf den nächsten Knoten (Node), dessen Element zurückgegeben wird



Interner Iterator auf MyLinkedList -- Programmieren 1.B

```
class MyIterator<E> implements Iterator<E> {
    TODO
    public boolean hasNext() { TODO }
    public E next() { TODO }
    public void remove() { throw new UnsupportedOperationException("Don't know yet."); }
}
```

- next-Zeiger jeweils auf den nächsten Knoten (Node), dessen Element zurückgegeben wird
- Aufruf von next() gibt das Element zurück auf dessen Knoten der next-Zeiger zeigt, anschliessend wird der Zeiger iteriert
- «NoSuchElementException», falls der next-Zeiger auf keinen gültigen Node mehr zeigt

Interner Iterator auf MyLinkedList -- Lösung

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() {
        return next != null;
    }
    public E next() {
        if (next == null) throw new NoSuchElementException();
        E e = next.elem;
        next = next.next;
        return e;
    }
    public void remove() { throw new UnsupportedOperationException("Don't know yet."); }
}
```

Laufzeiten?

Interner Iterator auf MyLinkedList -- Lösung

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() {
        return next != null;
    }
    public E next() {
        if (next == null) throw new NoSuchElementException();
        E e = next.elem;
        next = next.next;
        return e;
    }
    public void remove() { throw new UnsupportedOperationException("Don't know yet."); }
}
```

Laufzeiten?

Interner Iterator auf MyLinkedList -- Lösung

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() {
        return next != null;
    }
    public E next() {
        if (next == null) throw new NoSuchElementException();
        E e = next.elem;
        next = next.next;
        return e;
    }
    public void remove() { throw new UnsupportedOperationException("Don't know yet."); }
}
```

1 Vergleich -> O(1)

O(1)

O(1)

O(1)

-> Total: O(1)

Laufzeiten - Schlechtes Beispiel einer Implementierung

```
class MyIterator<E> implements Iterator<E> {
   private List<E> list;
   private int next = 0;

   MyIterator(List<E> list) { this.list = list; }

   public boolean hasNext() { return next < list.size(); }

   public E next() { return list.get(next++); }

   public void remove() { … }
}
```

Warum ist das schlecht? Wie lange dauern die jeweiligen Operationen (vs. sollten sie dauern)?

Laufzeiten - Schlechtes Beispiel einer Implementierung

```
class MyIterator<E> implements Iterator<E> {
   private List<E> list;
   private int next = 0;

   MyIterator(List<E> list) { this.list = list; }

   public boolean hasNext() { return next < list.size(); }

   public E next() { return list.get(next++); }

   public void remove() { … }
}
```

Die Laufzeit von next() ist in **O(n)***.

Einmal durch die ganze Liste zu Iterieren dauert deshalb **O(n²)**.

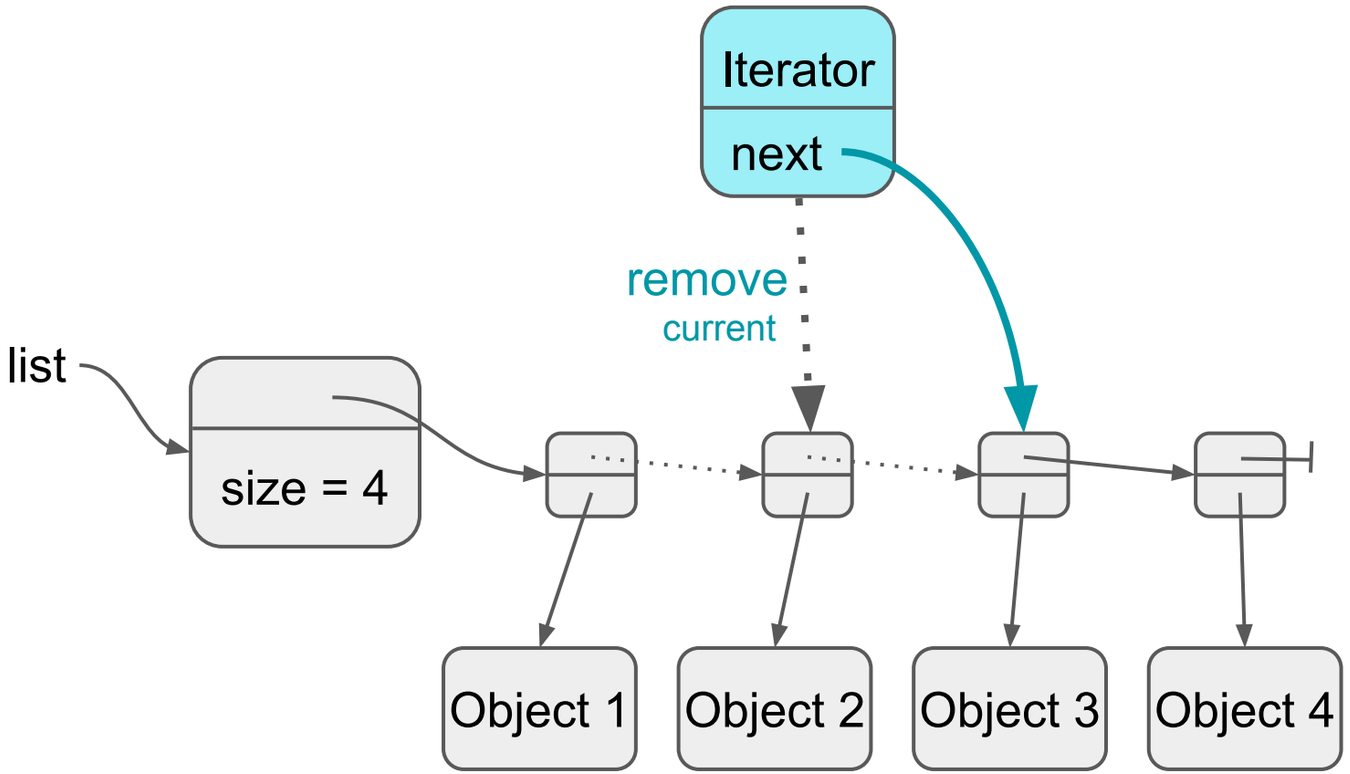
Dauern sollte ein Durchlauf nur O(n).

Warum ist das schlecht? Wie lange dauern die jeweiligen Operationen (vs. sollten sie dauern)?

* genauer gesagt ist es O(idx). Im Total also O(1)+O(2)+....+O(n) = O(n²).

Interner Iterator auf MyLinkedList -- Löschen

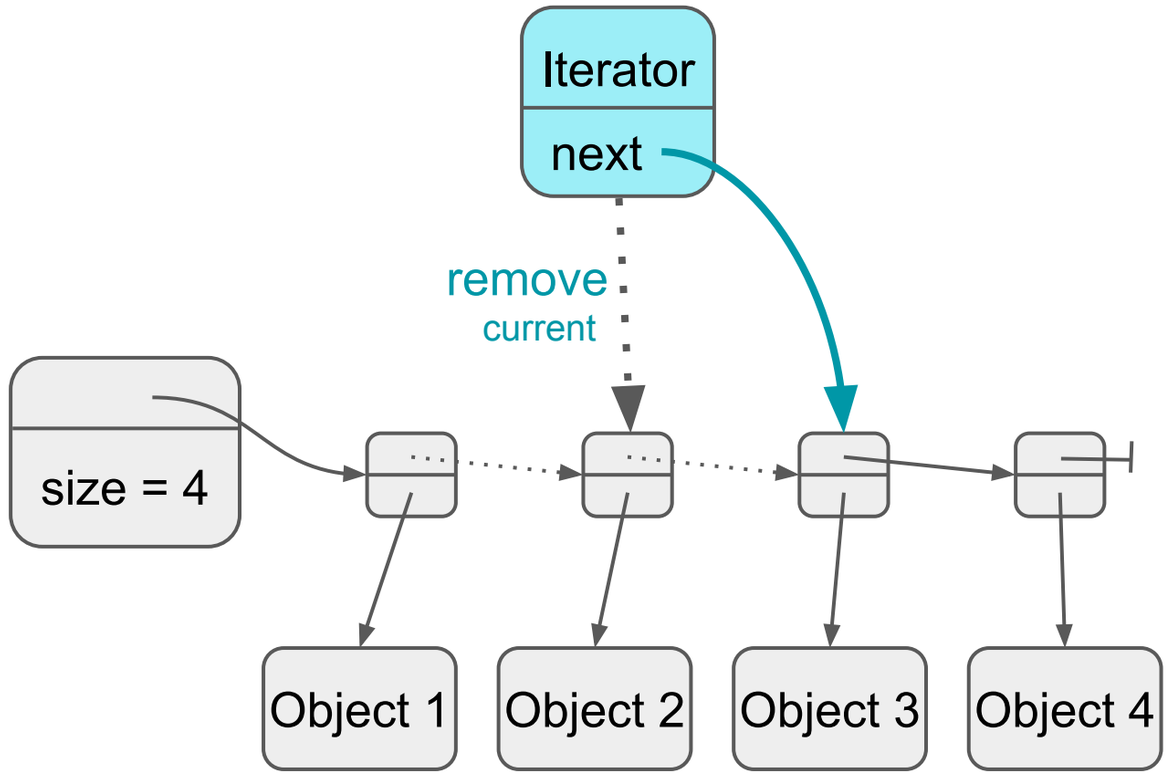
- Methode remove() löscht *zuletzt zurückgegebenes* Element (also “current” den Vorgänger von next).
- Einem remove()-Aufruf muss *mindestens* ein next()-Aufruf vorangehen.
  - Sonst: IllegalStateException
  - **Warum?**



Interner Iterator auf MyLinkedList -- Löschen

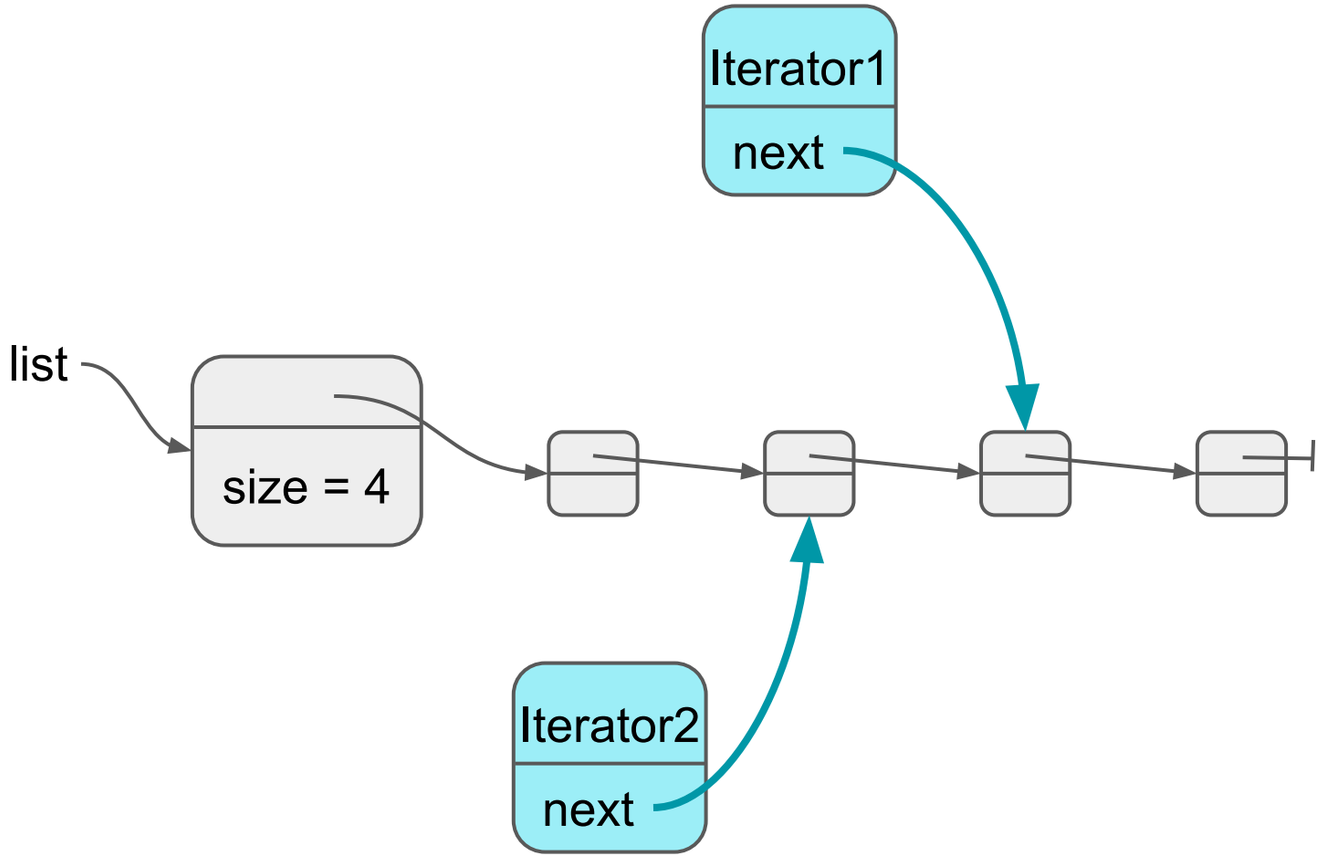
- Methode remove() löscht *zuletzt zurückgegebenes* Element (also “current” den Vorgänger von next).
- Einem remove()-Aufruf muss *mindestens* ein next()-Aufruf vorangehen.
  - Sonst: IllegalStateException
  - **Warum?**

Weil nicht beliebig viele Vorgängerknoten von next erinnert werden können.



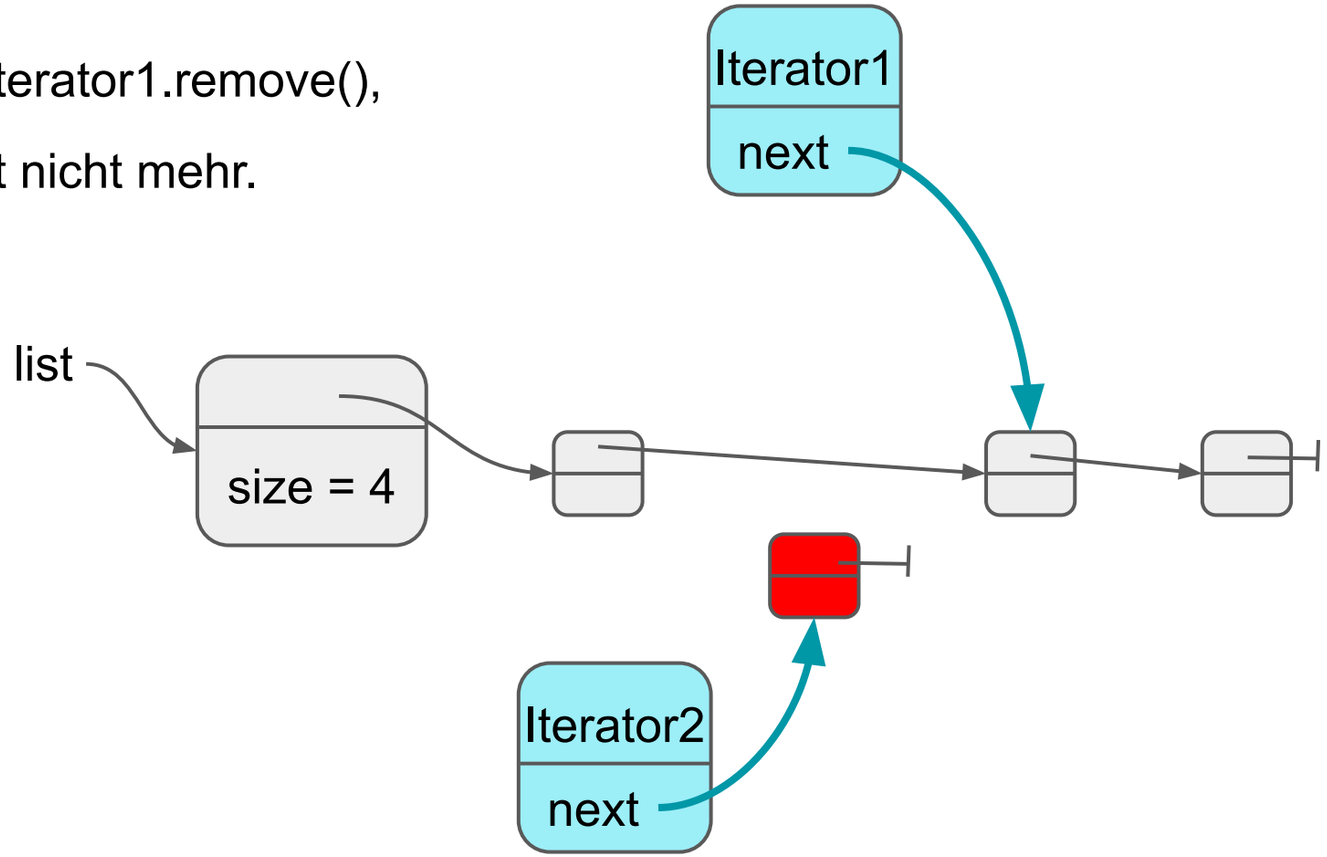
Mehrere Iteratoren gleichzeitig

Wo liegt das Problem?



Mehrere Iteratoren gleichzeitig

- Problematisch, wenn z.B. Iterator1.remove(),
- dann existiert Iterator2.next nicht mehr.



Modification Counter -- Generationenzähler

- Liste hält eine Zählvariable **«modCount»** (Modification Counter)
- Listenstruktur verändernde Operationen erhöhen modCount in der Liste um 1:
  - add()
  - remove()
- Iterator kopiert bei Instanziierung den modCount
- Iterator prüft regelmässig, ob die beiden modCount (Liste und Iterator) gleich sind:
  - next(), remove(), set(), add(), previous()
- *ConcurrentModificationException*, falls Werte unterschiedlich sind:
  - **Ausnahme:** iterator.remove() inkrementiert auch den modCount im Iterator

Aufgabe

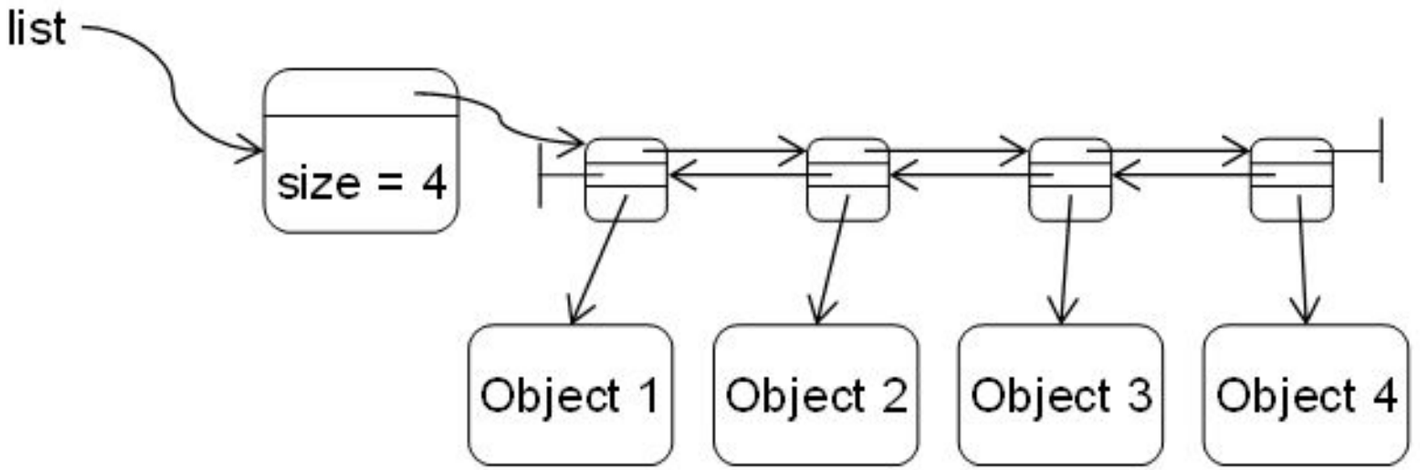
Programmieren 1.C (mod count) & 1.D (remove)

ListIterator

```
public interface ListIterator<E> extends Iterator<E> {
   // Query Operations
   boolean hasNext();         // as for simple Iterator
   E next();
   boolean hasPrevious();     // moves in opposite direction
   E previous();
   int nextIndex();           // returns position left and right of Iterator
   int previousIndex();

   // Modification Operations
   void remove();     // removes least recently (by next() or previous()) returned element
   void set(E e);     // replaces least recently returned element
   void add(E e);     // adds a new element at the current iterator position
}
```

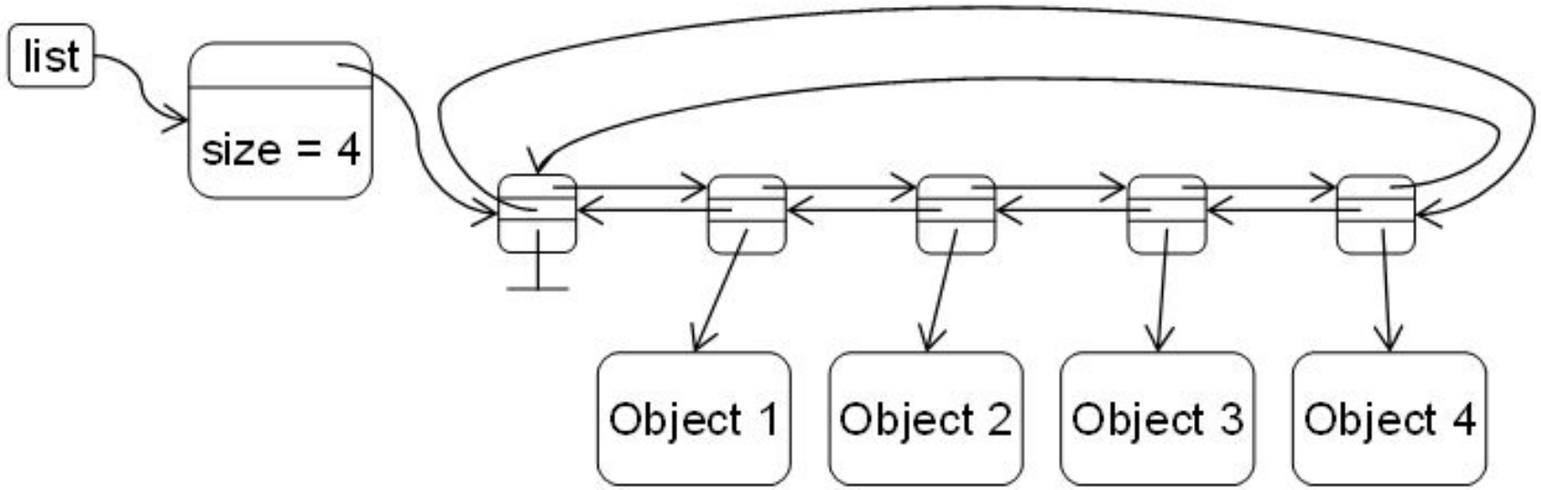
Der `ListIterator` erlaubt das Iterieren in **beide Richtungen**.

Doppelt Verkettete Liste (Doubly Linked List)




```
private static class Node<E> {
    private E elem;
    private Node<E> prev, next;

    private Node(E elem) { this.elem = elem; }
    private Node(Node<E> prev, E elem, Node<E> next) {
        this.prev = prev; this.elem = elem; this.next = next;
    }
}
```

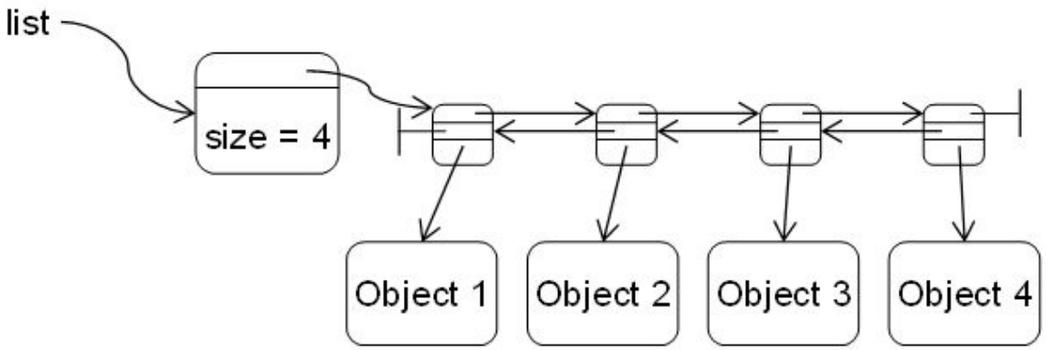
Zyklisch verkettete Liste mit Kopfelement




```
private static class Node<E> {
    private E elem;
    private Node<E> prev, next;

    private Node(E elem) { this.elem = elem; }
    private Node(Node<E> prev, E elem, Node<E> next) {
        this.prev = prev; this.elem = elem; this.next = next;
    }
}
```

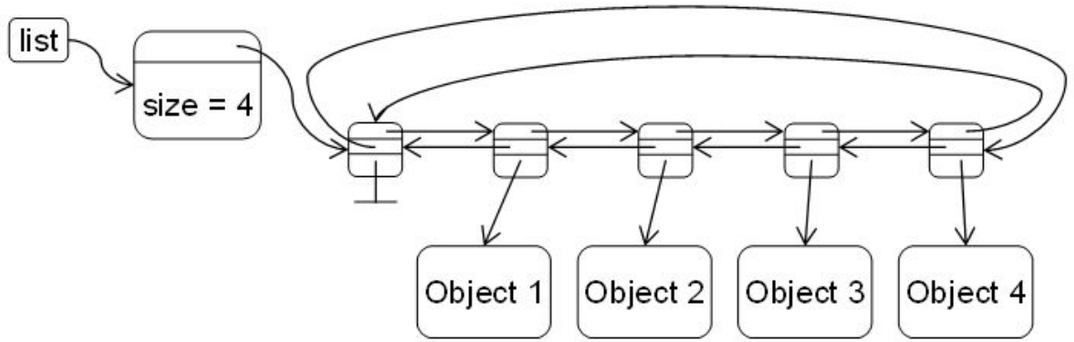
Doppelt Verkettete Liste

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() { TODO }
}
```



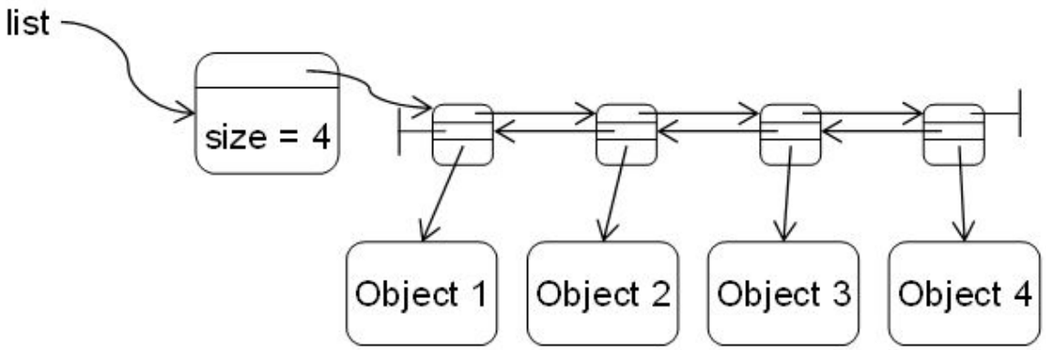
Zyklisch verkettete Liste mit Kopfelement

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() { TODO }
}
```



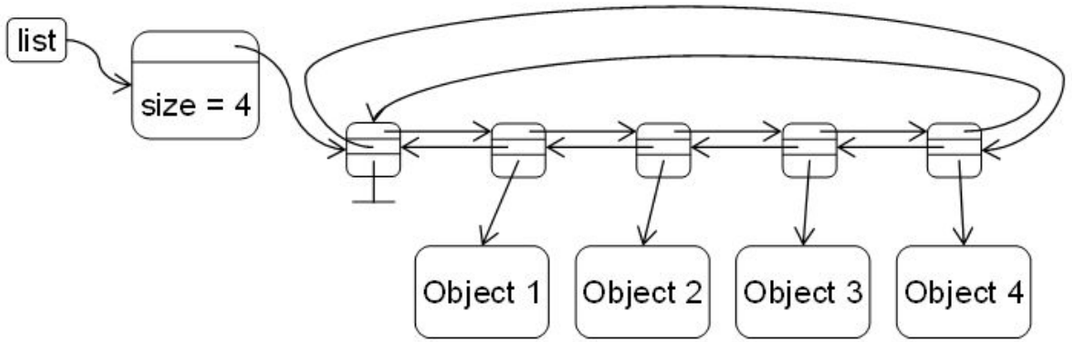
Doppelt Verkettete Liste

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() {
        return next != null;
    }
}
```



Zyklisch verkettete Liste mit Kopfelement

```
class MyIterator<E> implements Iterator<E> {
    private Node<E> next = first;

    public boolean hasNext() {
      return next != first;
    }
}
```



Hausaufgaben

Programmieren, Aufgabe 1 (fertig) & Aufgabe 2