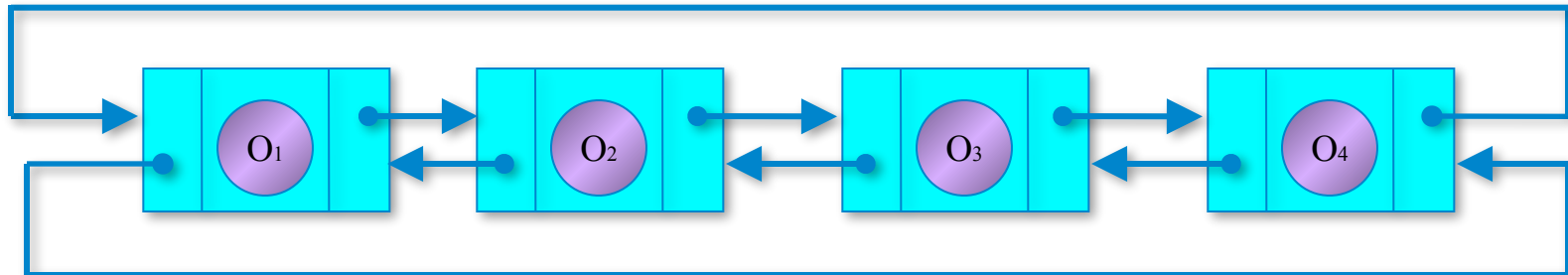


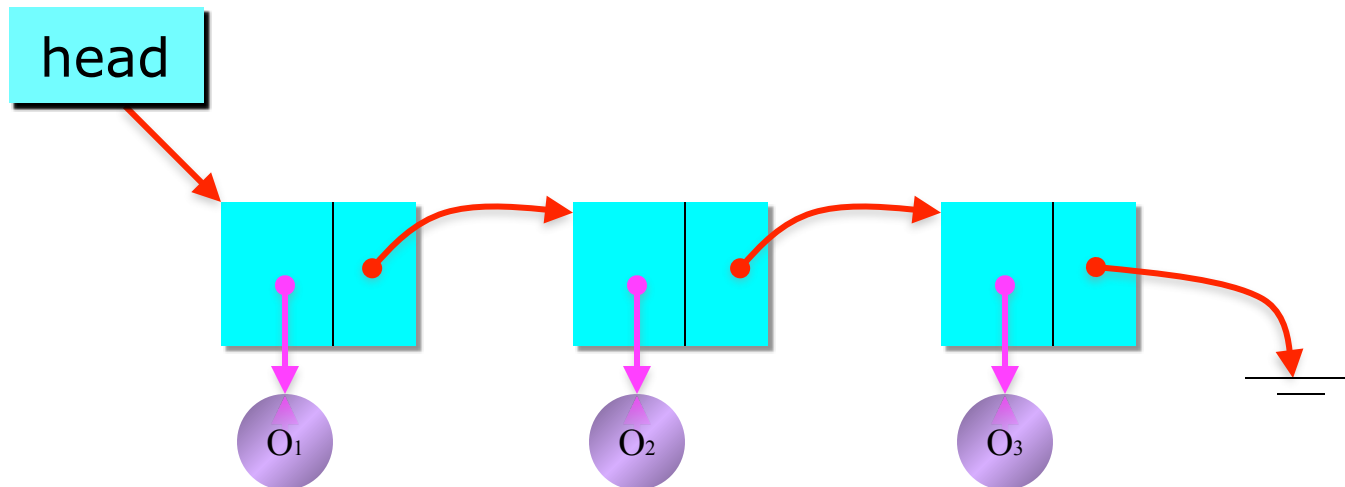
Dynamische Datenmengen

Datenabstraktion (Teil 2)



Oliver Wiese

Einfach verkettete Listen

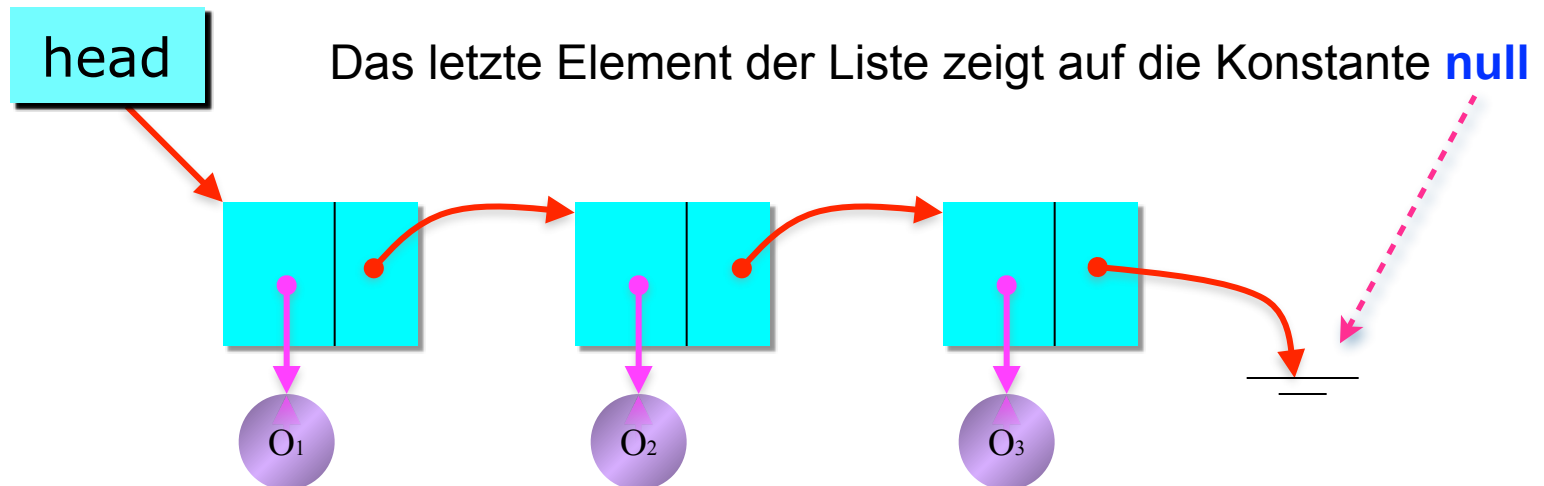


Einführung

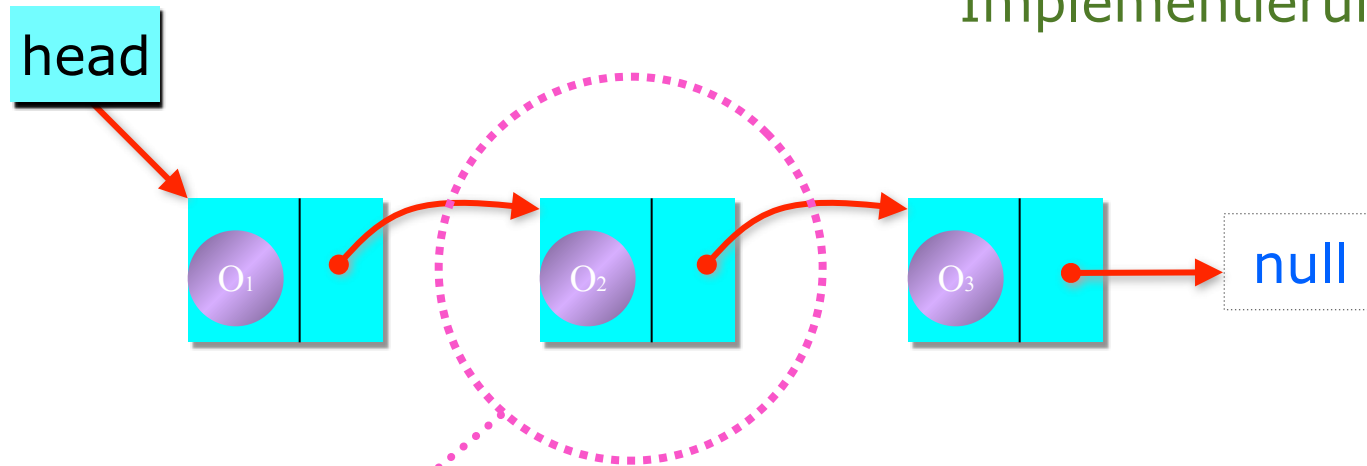
Einfach verkettete Listen sind die **einfachsten dynamischen Datenstrukturen**, die zur Laufzeit an den tatsächlichen Speicherbedarf anpassen können.

Eine Liste besteht aus einer **Menge von Knoten**, die **untereinander verkettet sind**.

Jeder Knoten besteht aus einer **Referenz auf das eigentliche zu speichernde Objekt** und **eine Referenz auf das nächste Element der Liste**.

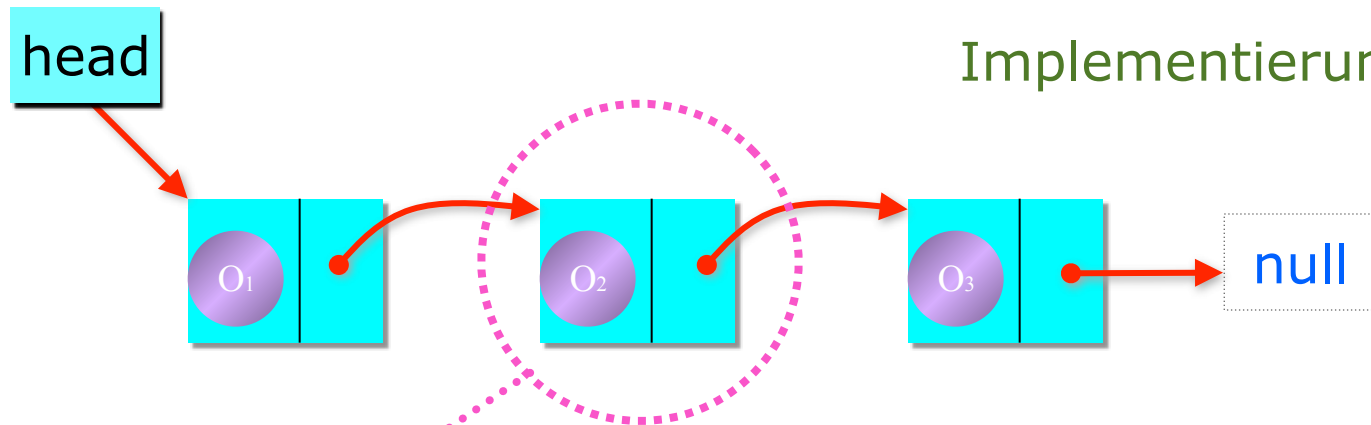


Implementierung



```
class ListNode <T> {  
    T element;  
    ListNode <T> next;  
    ...  
}
```

Wir haben eine **rekursive**
Klassendefinition,



Implementierung

```
class ListNode <T> {  
    T element;  
    ListNode<T> next;  
  
    // Konstruktoren  
    ListNode( T element, ListNode<T> next ){  
        this.element = element;  
        this.next = next;  
    }  
    ListNode() { this( null, null ); }  
}
```

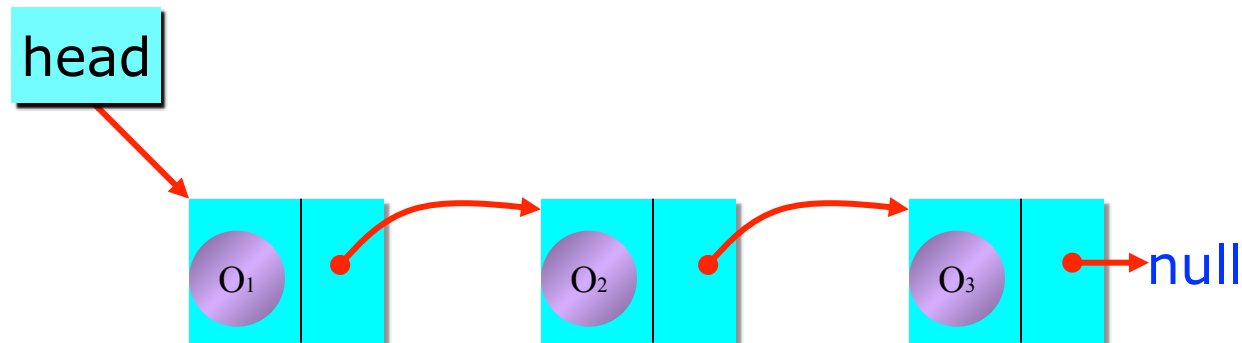
Zwei Konstruktoren:

Einer, der nur einen leeren Knoten erzeugt und einen, der gleichzeitig ein Objekt in dem Knoten speichert und die Referenz auf den nächsten Knoten bekommt.

Stapel als verkettete Liste

Mit Hilfe von verketteten Listen lässt sich sehr einfach und elegant ein Stapel implementieren.

Wir müssen dabei nicht überprüfen, ob der Stapel voll ist.



Wir brauchen nur ein **head**-Element, das eine **Referenz** auf ein **ListNode**-Objekt ist. Mit dieser Referenz können wir bei einer **push**-Operation neue Elemente am Anfang der Liste verketteten oder entfernen, wenn eine **pop**-Operation stattfindet.

Stapel-Schnittstelle

```
public interface Stack <E> {  
    public boolean empty();  
    public void push( E element );  
    public E pop() throws EmptyStackException;  
    public E peek() throws EmptyStackException;  
}
```

Eine **EmptyStackException** wird bei dem Versuch, ein Element zu entfernen oder zu lesen (**pop**- und **peek**-Operationen) erzeugt, wenn die Liste leer ist.

Einfache Implementierung der Stapel-Schnittstelle

Ein Konstruktor wird definiert, der **head** mit der Konstante **null** initialisiert.

```
public class ListStapel<T> implements Stack <T> {  
    /* Instanzvariablen */  
    private ListNode<T> head;  
    /* Konstruktor */  
    public ListStapel() {  
        head = null;  
    }  
    /* Methoden */  
    . . .  
}
```

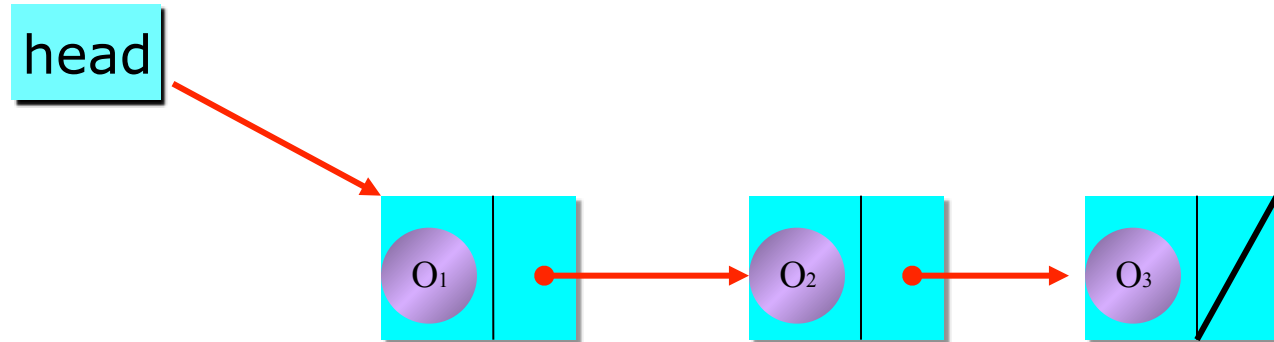

Implementierung der empty-Operation



Der Stapel ist leer, wenn das **head**-Element auf die Konstante **null** zeigt

```
public boolean empty () {  
    return head == null;  
}
```

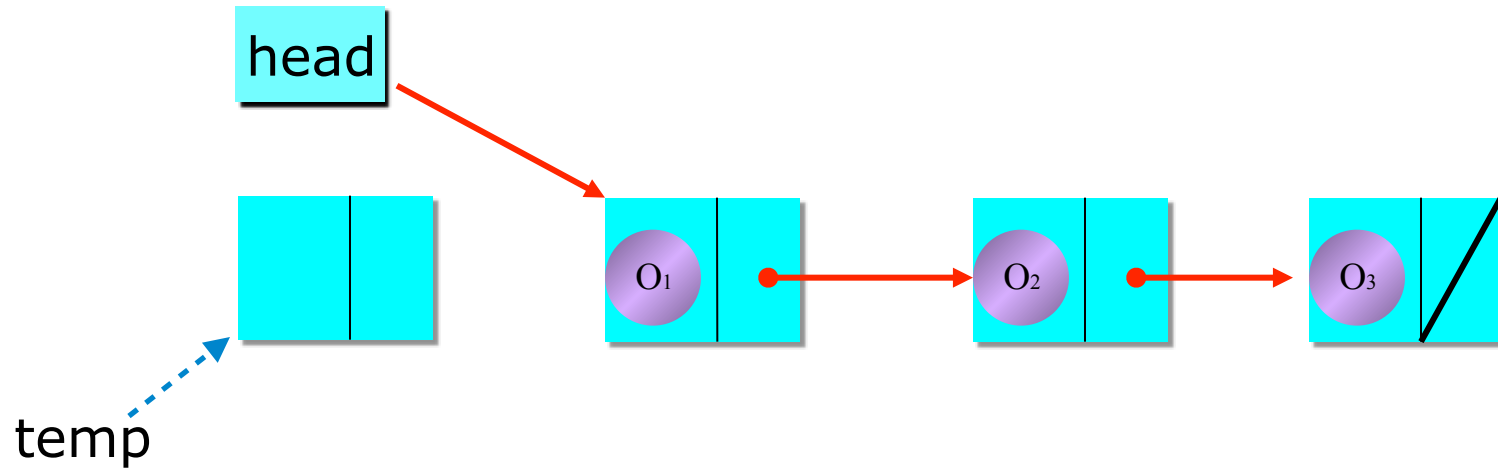
Implementierung der push-Operation



Das **T-Objekt** **element** soll am Anfang der Liste eingefügt werden

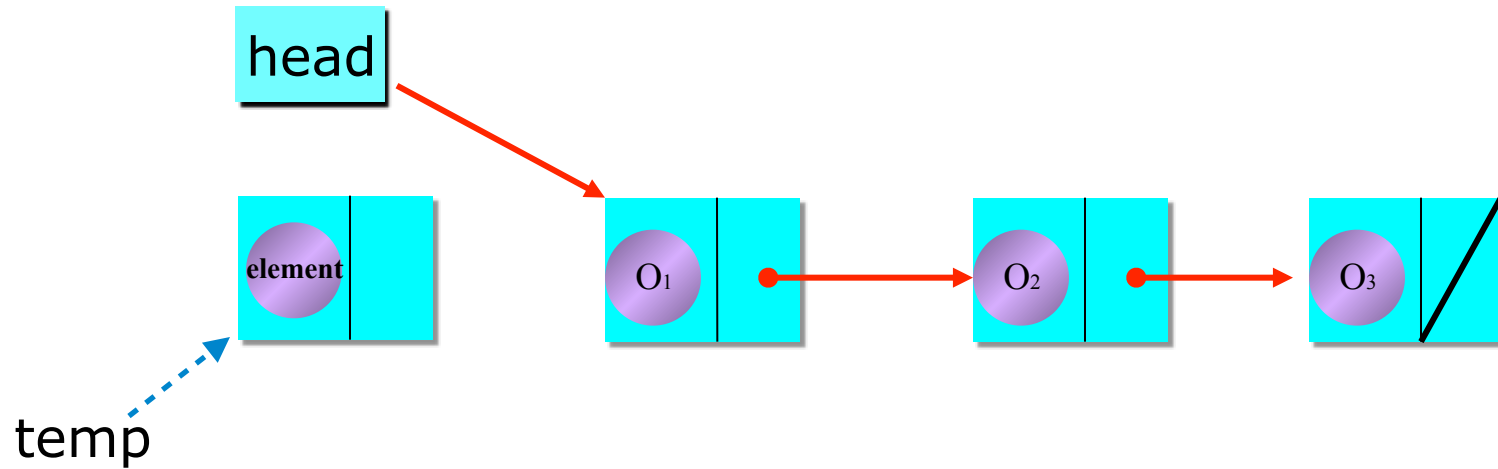
➡ **public void push (T element) {**
 ListNode<T> temp = new ListNode<T> ();
 temp.element = element;
 temp.next = head;
 head = temp;
}

Implementierung der push-Operation



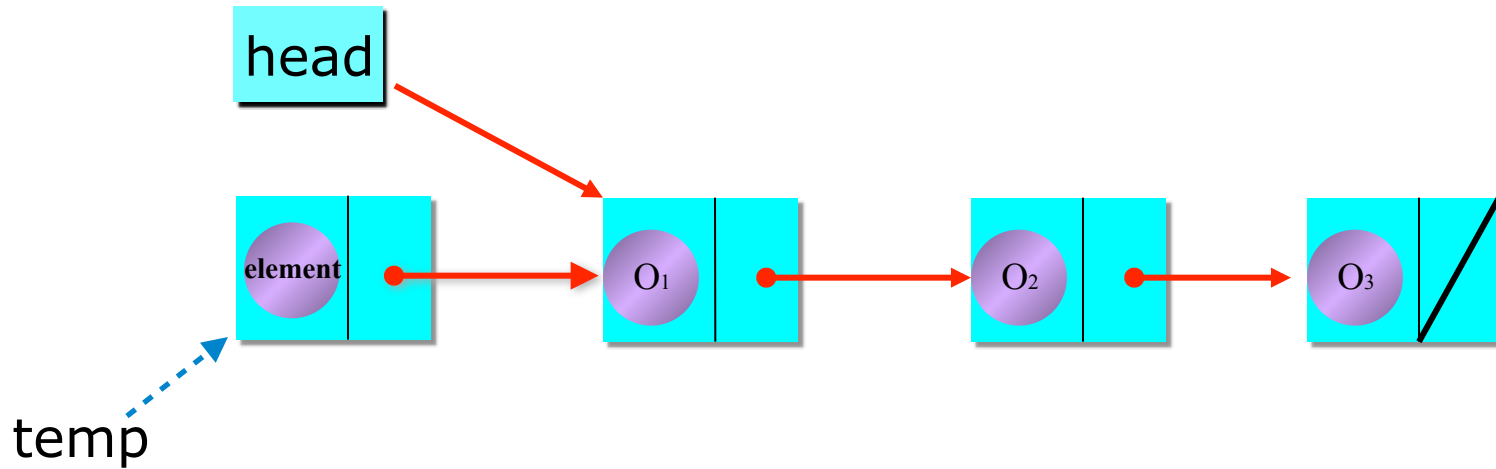
```
public void push ( T element ) {  
    ListNode<T> temp = new ListNode<T> ();  
    temp.element = element;  
    temp.next = head;  
    head = temp;  
}
```

Implementierung der push-Operation



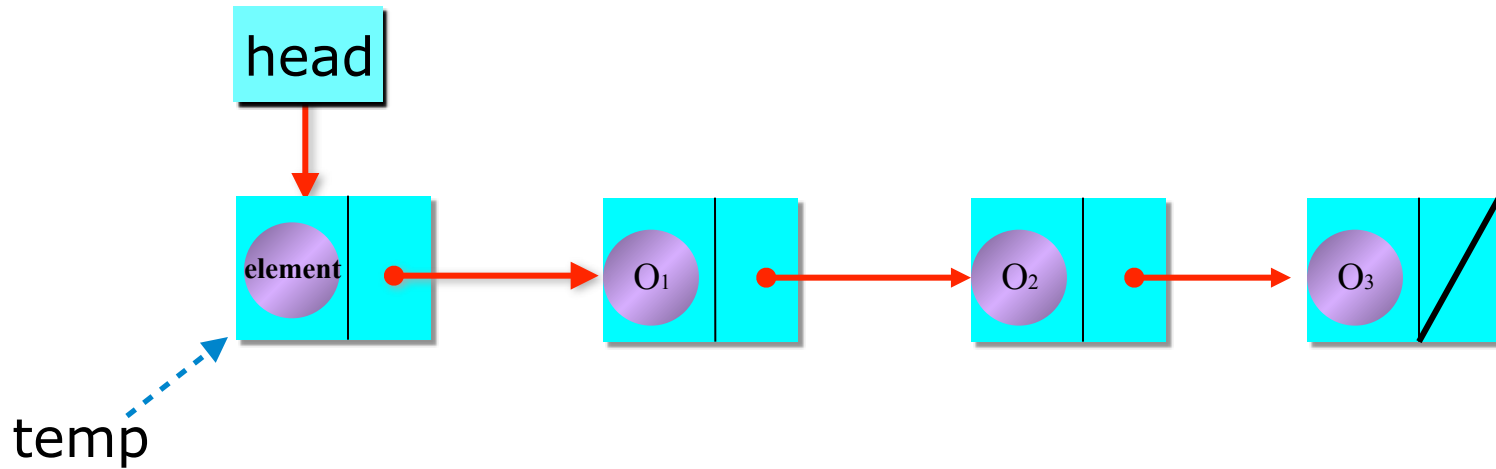
```
public void push ( T element ) {  
    ListNode<T> temp = new ListNode<T> ();  
    temp.element = element;  
    temp.next = head;  
    head = temp;  
}
```

Implementierung der push-Operation



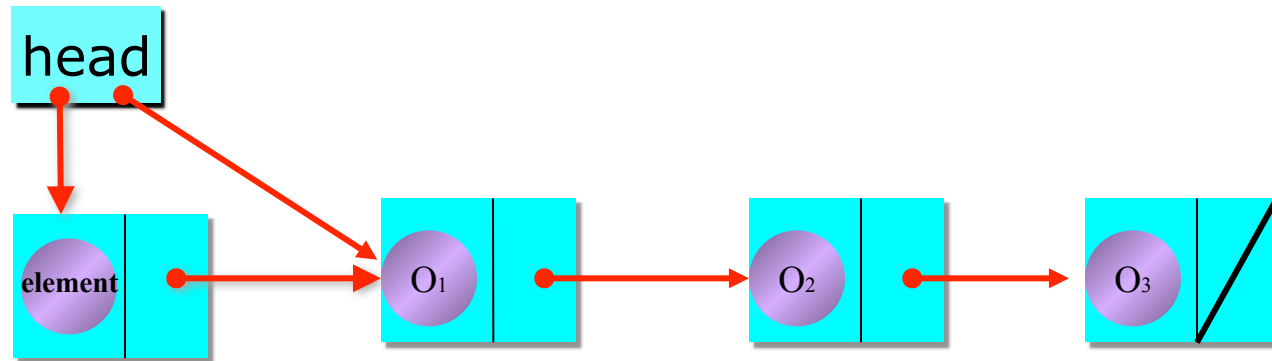
```
public void push ( T element ) {  
    ListNode<T> temp = new ListNode<T> ();  
    temp.element = element;  
    temp.next = head;  
    head = temp;  
}
```

Implementierung der push-Operation



```
public void push ( T element ) {  
    ListNode<T> temp = new ListNode<T> ();  
    temp.element = element;  
    temp.next = head;  
    head = temp;  
}
```

Implementierung der push-Operation

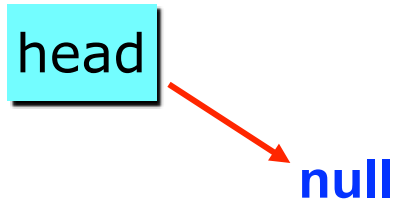


Mit Hilfe eines **ListNode**-Konstruktors, der das zu speichernde Objekt **element** und eine Referenz auf ein **ListNode**-Objekt bekommt, kann die **push**-Operation wie folgt implementiert werden.

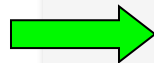
```
public void push ( T element ) {
    head = new ListNode<T> ( element, head );
}
```

A green arrow points to the line `head = new ListNode<T> (element, head);`.

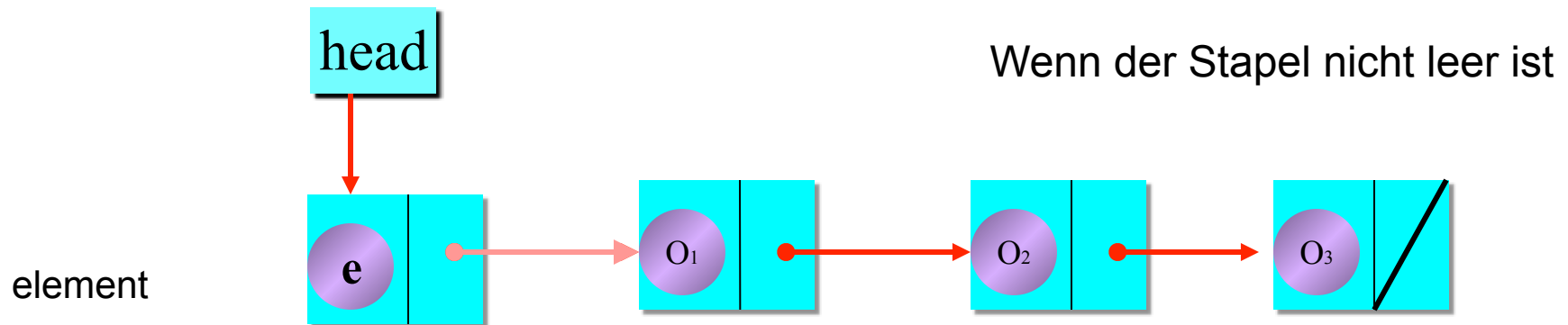
Implementierung der pop-Operation



Wenn innerhalb einer **pop**-Operation festgestellt wird, dass der Stapel leer ist, wird ein **EmptyStackException** geworfen und die **pop**-Operation unterbrochen.

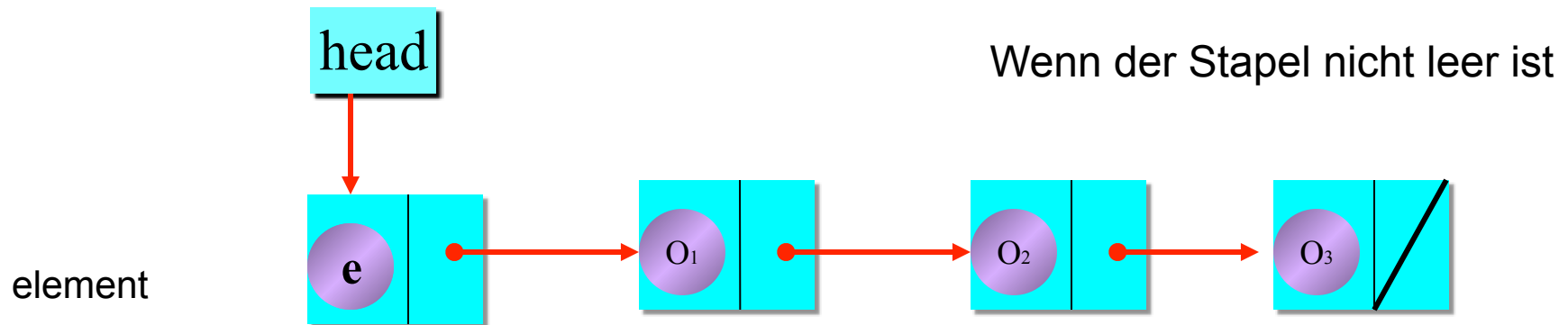
```
public T pop() throws EmptyStackException {
    if ( empty() )
         throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```


Implementierung der pop-Operation



```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Implementierung der pop-Operation

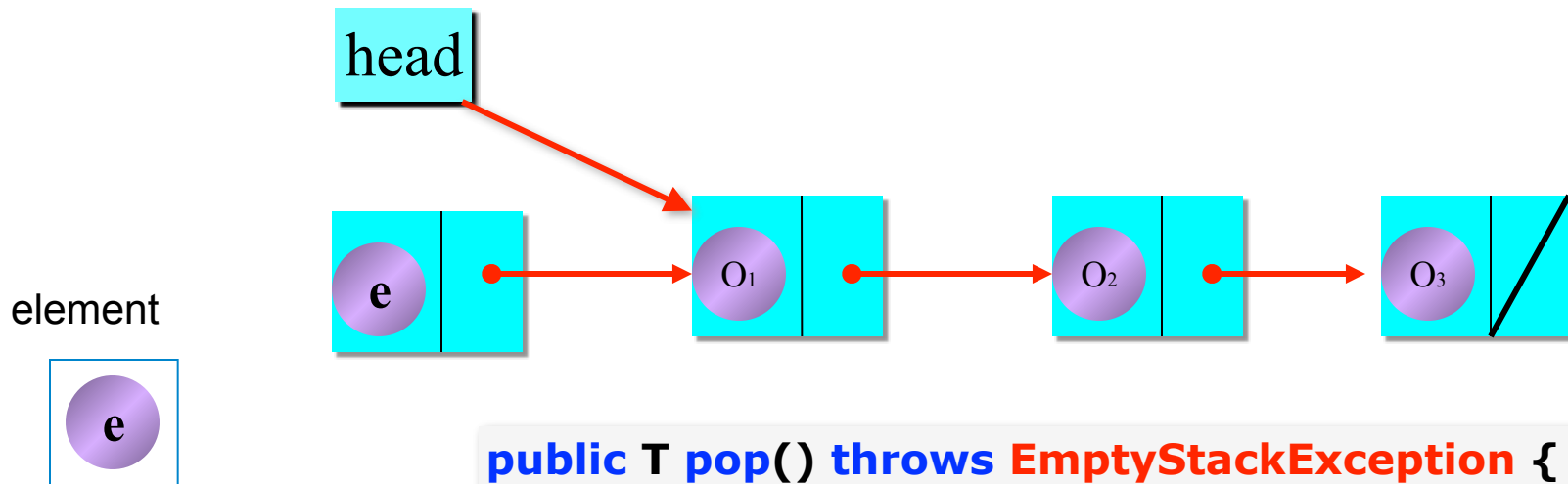


```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```



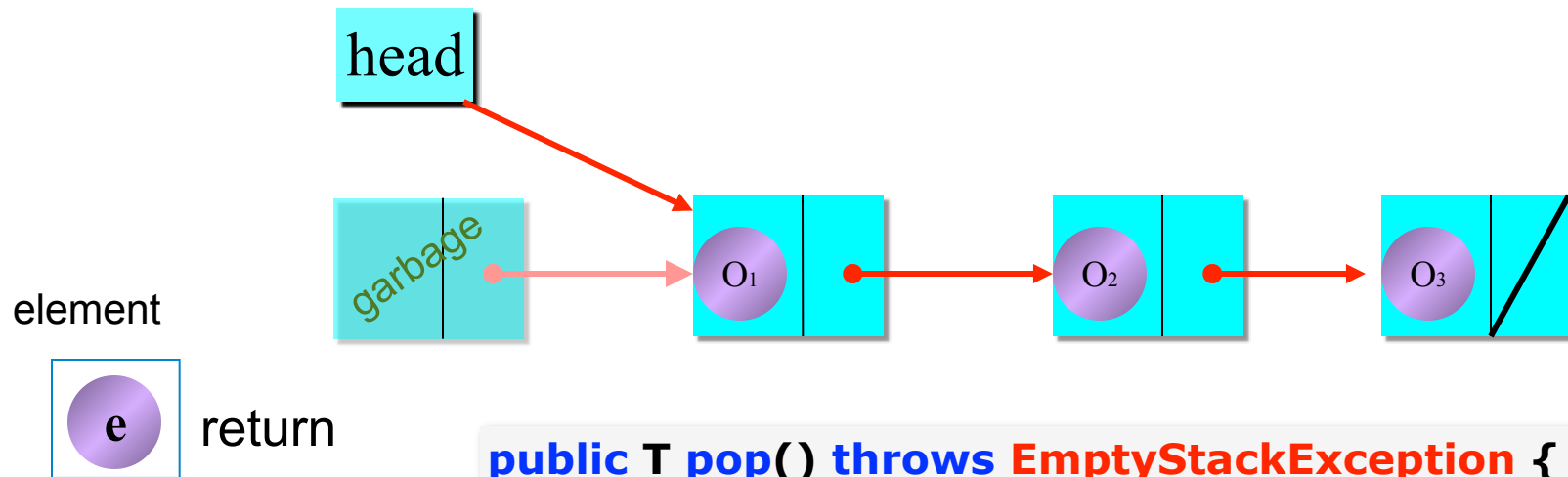
Die Objekt-Referenz,
die sich in dem ersten
Knoten befindet, wird
in die lokale Variable
element gespeichert.

Implementierung der pop-Operation



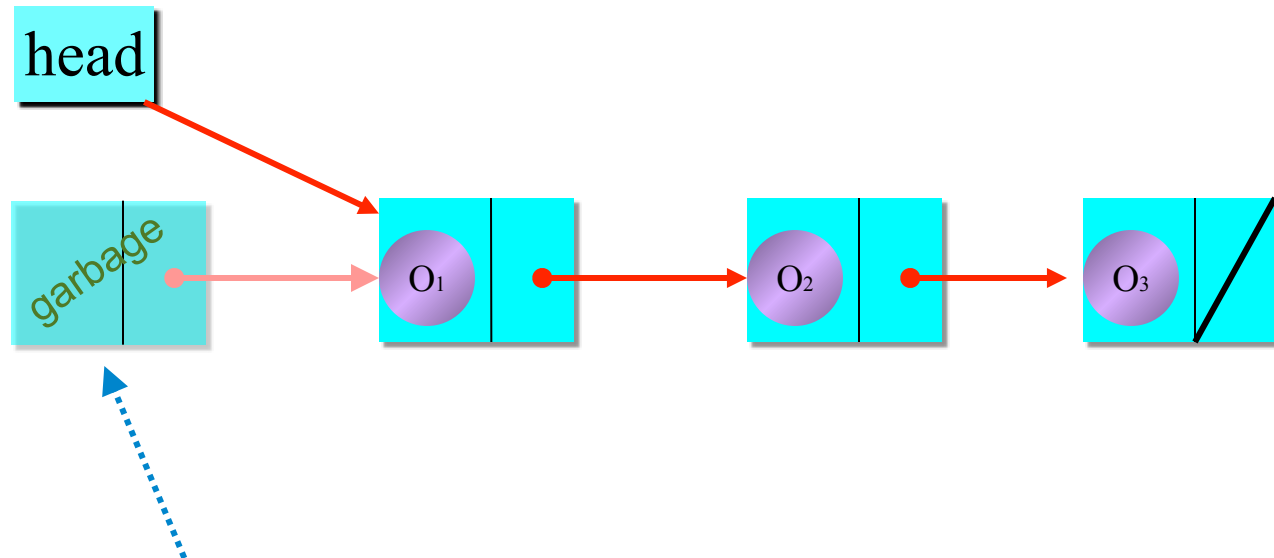
```
public T pop() throws EmptyStackException {  
    if ( empty() )  
        throw new EmptyStackException();  
    T element = head.element;  
    head = head.next;  
    return element;  
}
```

Implementierung der pop-Operation



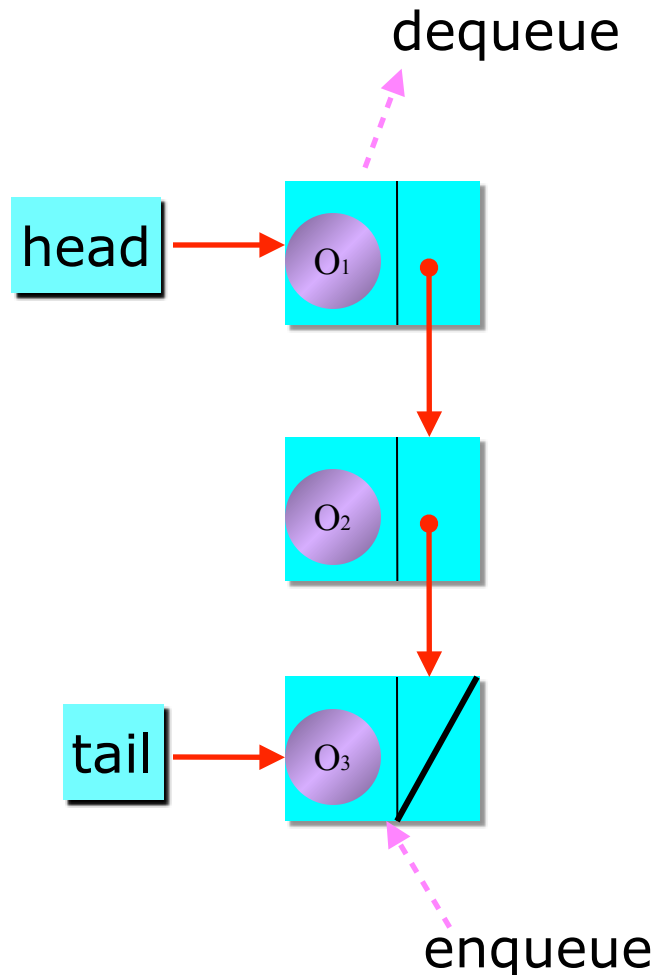
```
public T pop() throws EmptyStackException {  
    if ( empty() )  
        throw new EmptyStackException();  
    T element = head.element;  
    head = head.next;  
    return element;  
}
```

Implementierung der pop-Operation



Das entfernte **ListNode**-Objekt bleibt ohne eine einzige Referenz, das auf es zeigt, und verwandelt sich in Datenspeichermüll, der später von dem Java-“garbage collector“ beseitigt wird.

Einfache Implementierung einer Warteschlange



Operationen der Warteschlange

enqueue

dequeue

empty

head

Wir brauchen nicht zu überprüfen, ob die Warteschlange voll ist.

Warteschlange-Schnittstelle

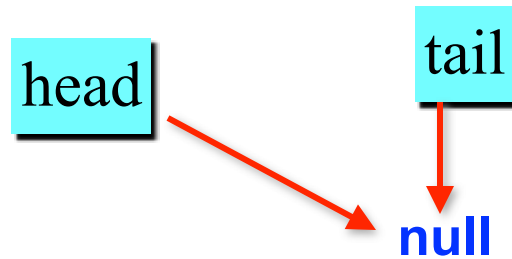
```
public interface Queue<T> {  
    public void enqueue( T newElement ) ;  
    public T dequeue() throws EmptyQueueException;  
    public T head() throws EmptyQueueException;  
    public boolean empty();  
}
```

Eine **EmptyQueueException** wird, bei dem Versuch ein Element zu entfernen oder zu lesen (**dequeue**-Operation und **head**-Operationen), wenn die Liste leer ist, produziert.

Warteschlange-Implementierung

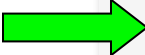
```
public class ListQueue<T> implements Queue<T> {  
  
    private ListNode<T> head;  
    private ListNode<T> tail;  
  
    public ListQueue() {  
        this.head = null;  
        this.tail = null;  
  
        ...  
    }  
}
```


Implementierung der **empty**-Operation

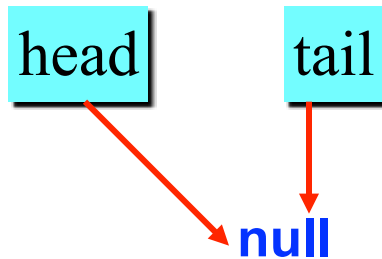


Die Warteschlange ist leer, wenn das **head**-Element auf die Konstante **null** zeigt

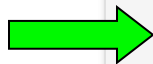
```
public boolean empty () {  
    return head == null;  
}
```



enqueue-Operation

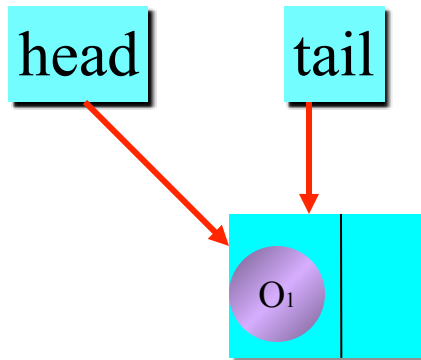


Wenn die
Liste leer
ist



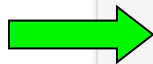
```
public void enqueue ( T newElement ) {  
    if ( empty() )  
        head = tail = new ListNode<T> ( newElement );  
    else  
        tail = tail.next = new ListNode<T>( newElement );  
}
```

enqueue-Operation

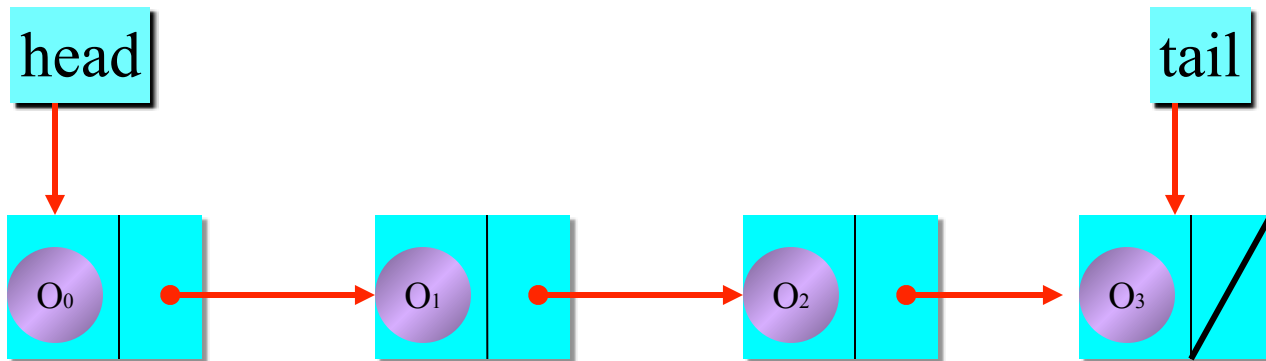


Das zu speichernde Element muss zuerst in ein `ListNode`-Objekt verpackt werden, und `head` und `tail` bekommen die Referenz des neuen `ListNode`-Objekts zugewiesen.

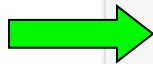
```
public void enqueue ( T newElement ) {
    if ( empty() )
        head = tail = new ListNode<T> ( newElement );
    else
        tail = tail.next = new ListNode<T>( newElement );
}
```



enqueue-Operation

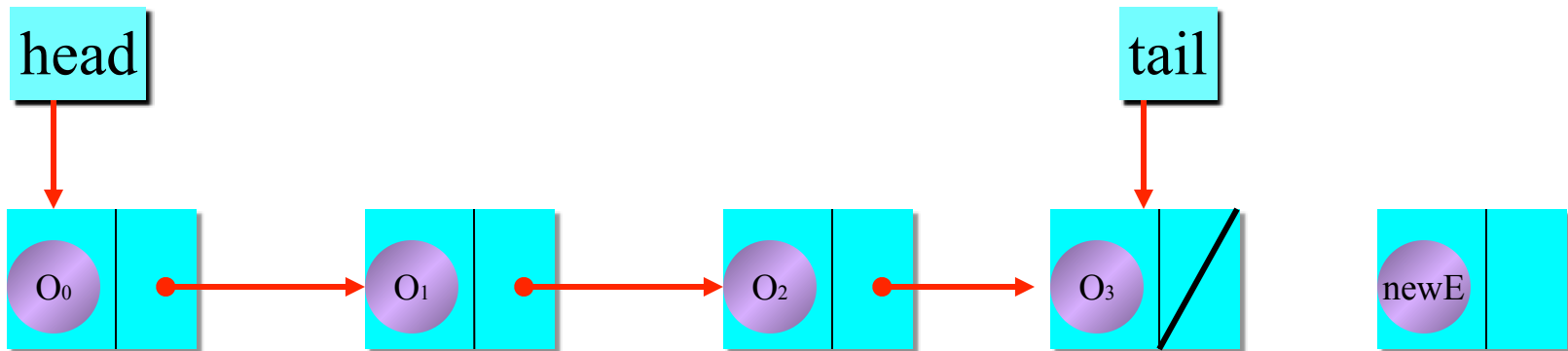


Wenn die
Liste nicht
leer ist



```
public void enqueue ( T newElement ) {
    if ( empty() )
        head = tail = new ListNode<T> ( newElement );
    else
        tail = tail.next = new ListNode<T>( newElement );
}
```

enqueue-Operation

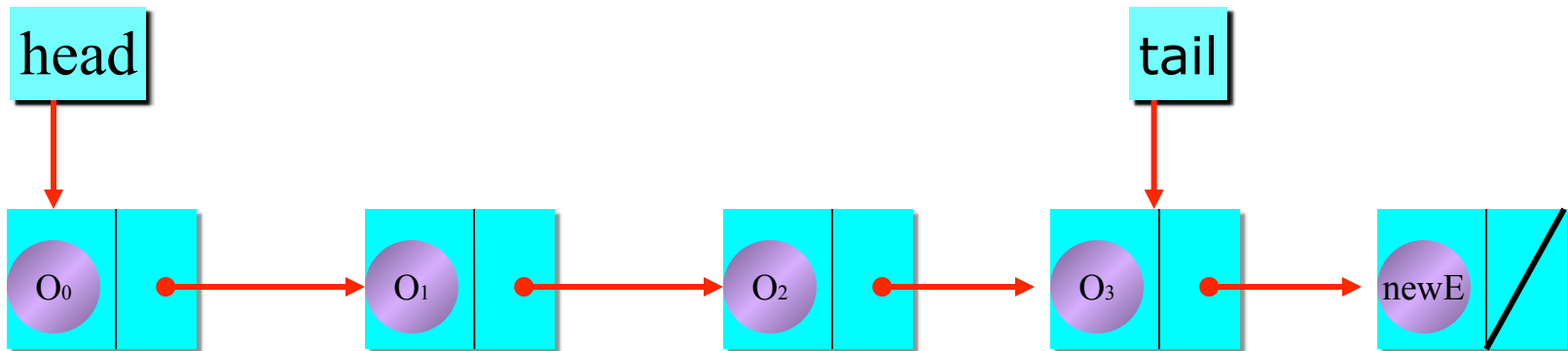


Zuerst wird das zu
speichernde neue
Objekt in einen neu
erzeugten
Listenknoten
(ListNode) verpackt.



```
public void enqueue ( T newElement ) {  
    if ( empty() )  
        head = tail = new ListNode<T> ( newElement );  
    else  
        tail = tail.next = new ListNode<T>( newElement );  
}
```

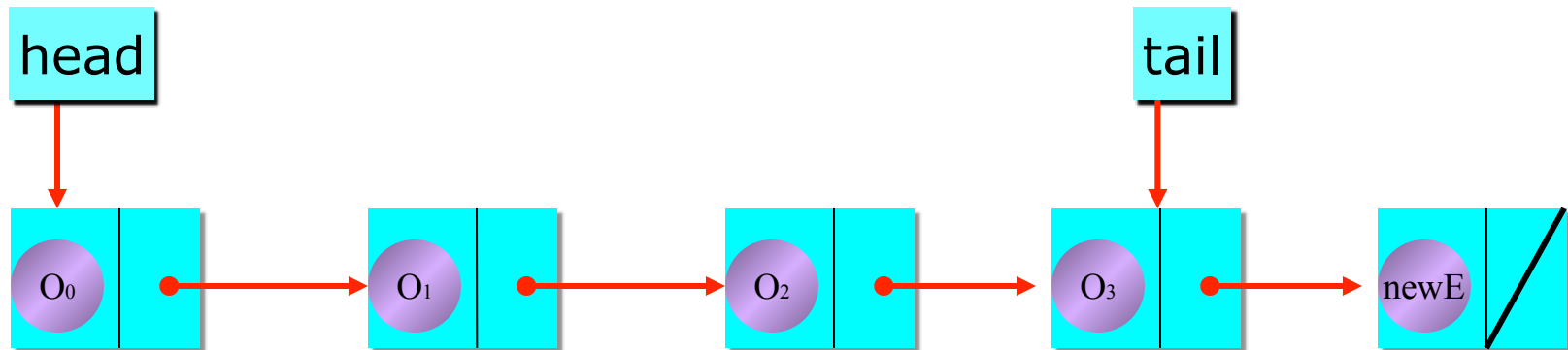
enqueue-Operation



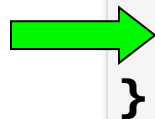
Die Referenz, die von dem ListNode-Konstruktor erzeugt wird, bekommt tail.next zugewiesen

```
public void enqueue ( T newElement ) {
    if ( empty() )
        head = tail = new ListNode<T> ( newElement );
    else
        tail = tail.next = new ListNode<T>( newElement );
}
```

enqueue-Operation

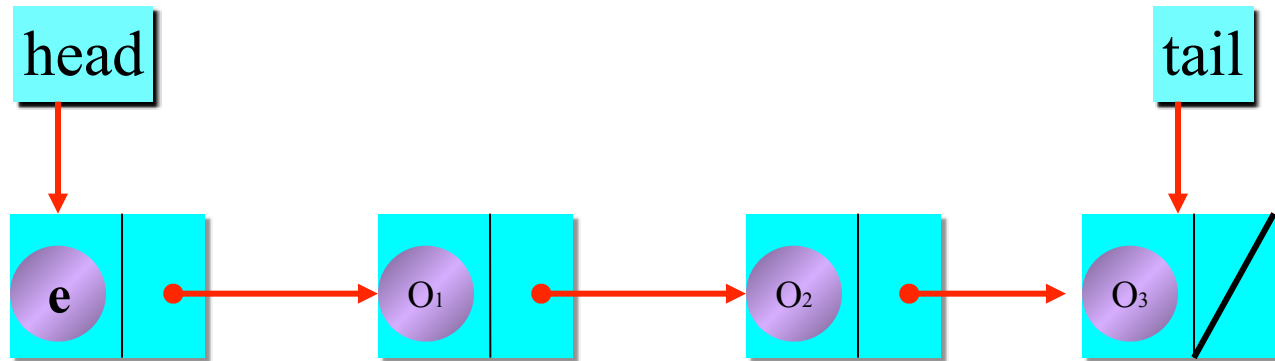


Zum Schluss
bekommt tail
auch die gleiche
Referenz wie
tail.next.

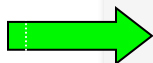


```
public void enqueue ( T newElement ) {
    if ( empty() )
        head = tail = new ListNode<T> ( newElement );
    else
        tail = tail.next = new ListNode<T>( newElement );
}
```

dequeue-Operationen



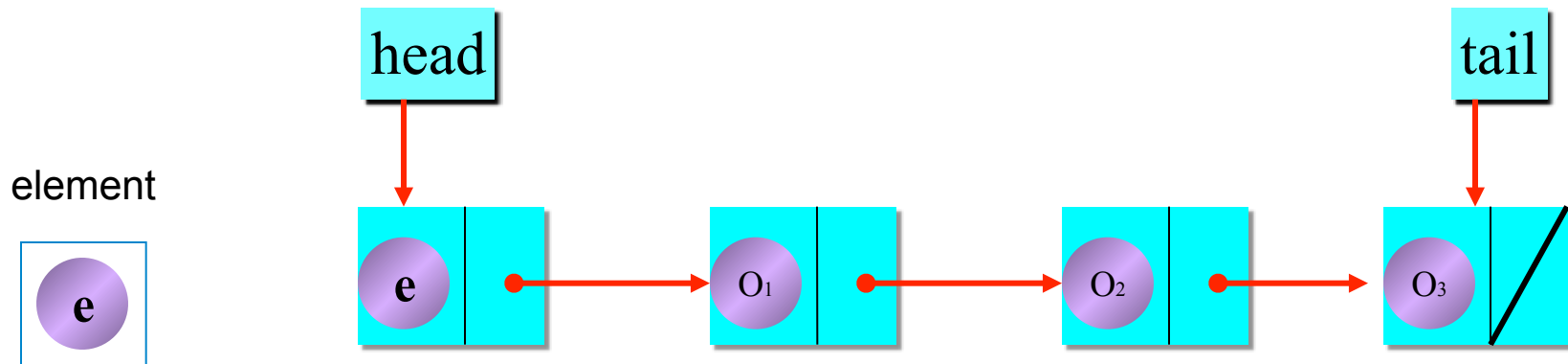
Wenn die
Liste nicht
leer ist



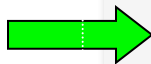
```

public T dequeue() throws EmptyQueueException {
    if (empty ())
        throw new EmptyQueueException();
    T element = head.element;
    head = head.next;
    return element;
}
  
```


dequeue-Operationen

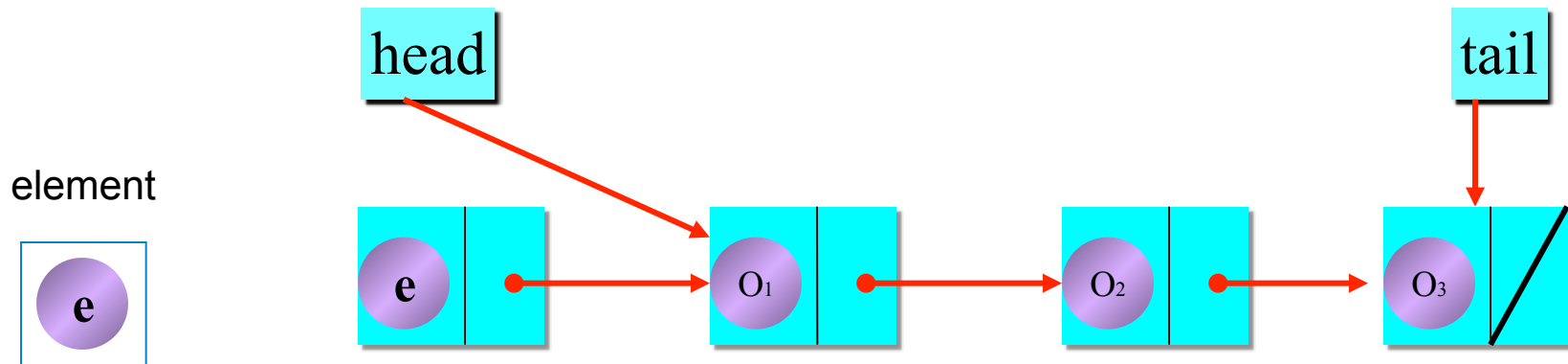


Die Referenz des Elements, das am Anfang der Warteschlange gespeichert ist, wird der Variablen **element** zugewiesen.



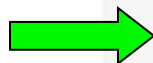
```
public T dequeue() throws EmptyQueueException {
    if (empty () )
        throw new EmptyQueueException();
    T element = head.element;
    head = head.next;
    return element;
}
```

dequeue-Operationen

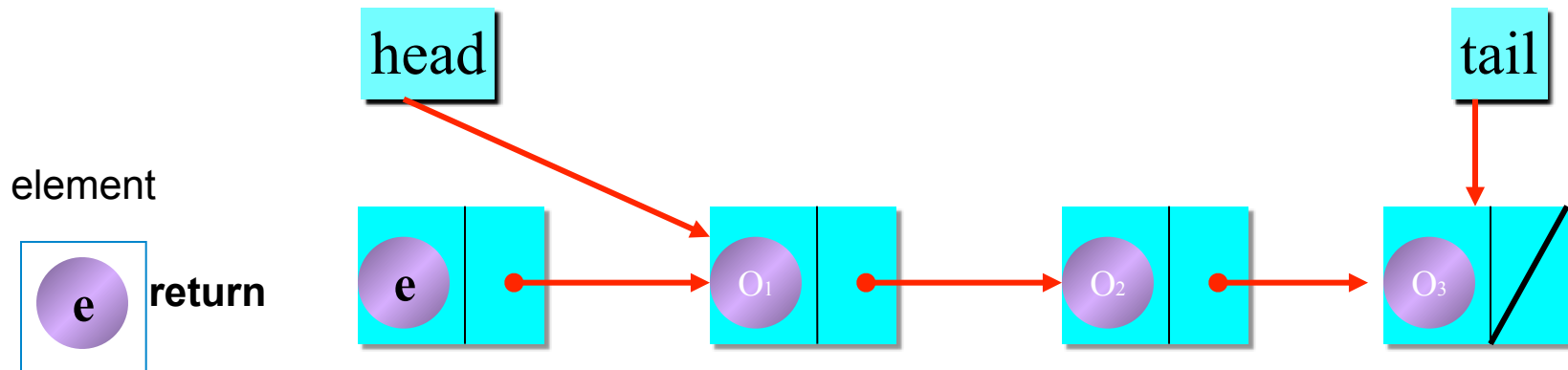


```

public T dequeue() throws EmptyQueueException {
    if (empty () )
        throw new EmptyQueueException();
    T element = head.element;
    head = head.next;
    return element;
}
    
```

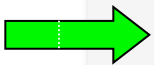


dequeue-Operationen

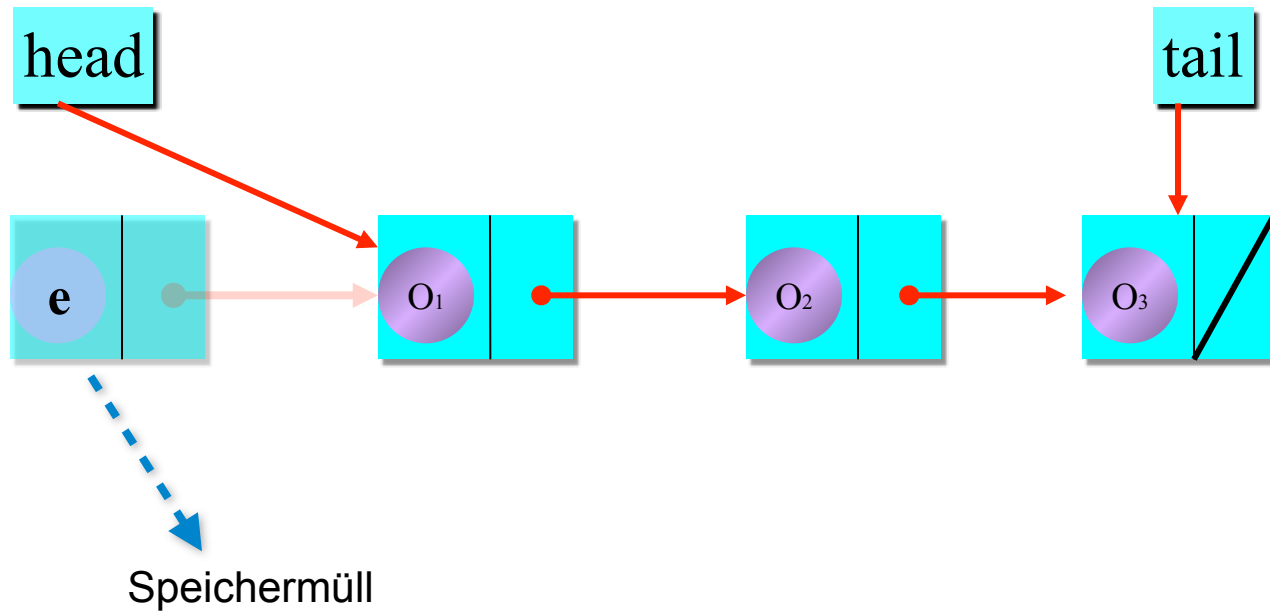


```
public T dequeue() throws EmptyQueueException {
    if (empty () )
        throw new EmptyQueueException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Die Referenz des entfernten Objekts wird als Ergebnis zurückgegeben.



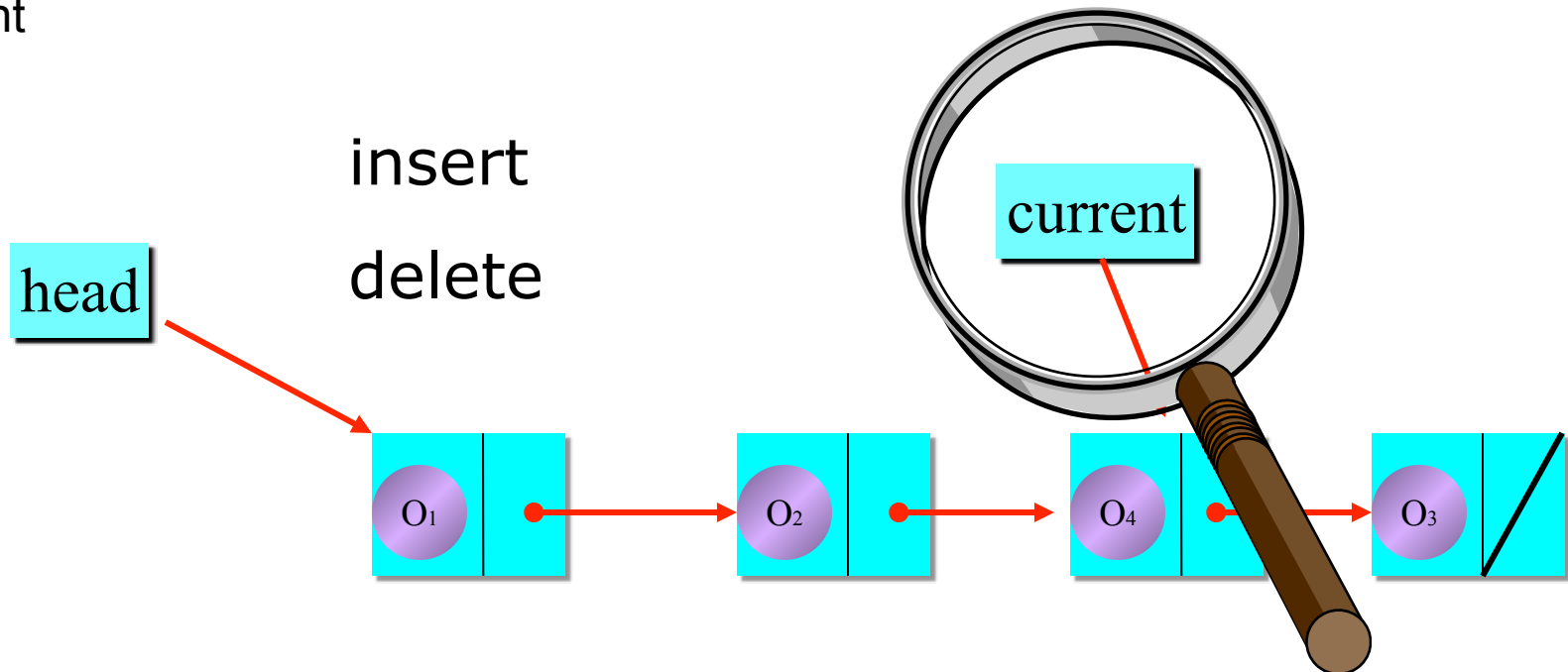
dequeue-Operationen



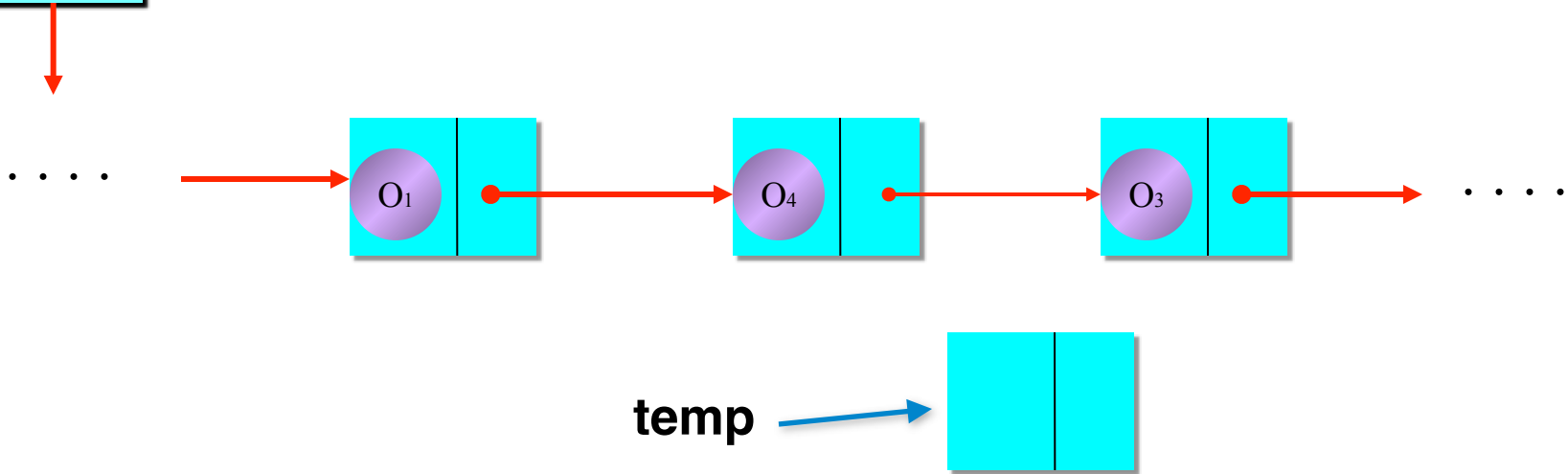
Allgemeine dynamische Datenmenge

Wenn wir allgemeine dynamische Datenmengen mit Hilfe von Listen implementieren möchten, müssen wir an einer beliebigen Stelle der Liste Elemente einfügen und löschen können. Dafür brauchen wir ein weiteres Referenz-Objekt **current** das sich durch die Liste bewegt.

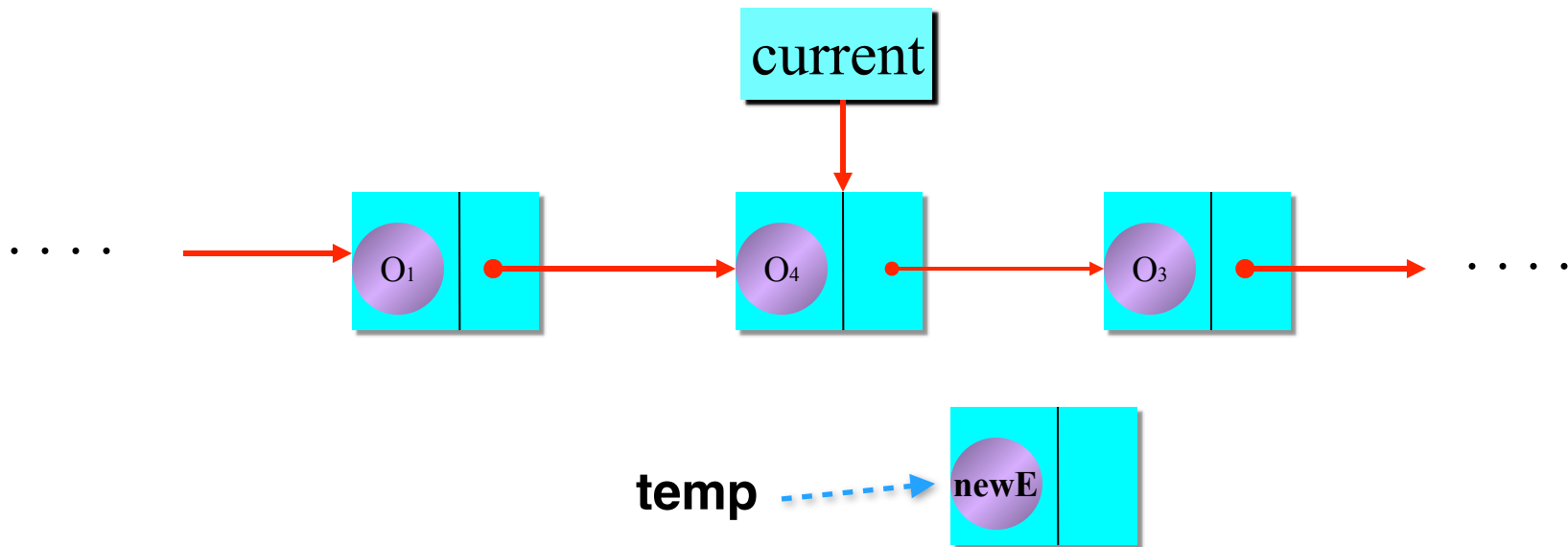
Wir werden eine Einfüge-Operation definieren, die ein Element nach dem **current** - Zeiger einfügt und eine Lösch-Operation, die ein Element nach dem **current-Zeiger** löscht



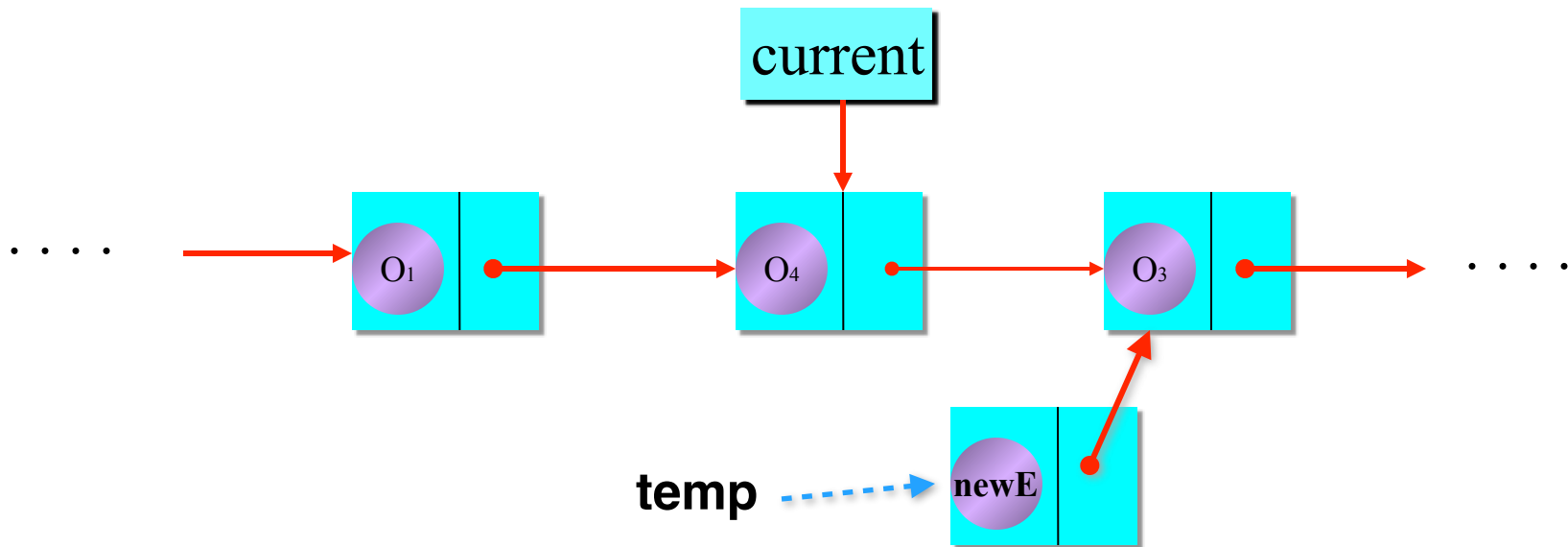
current



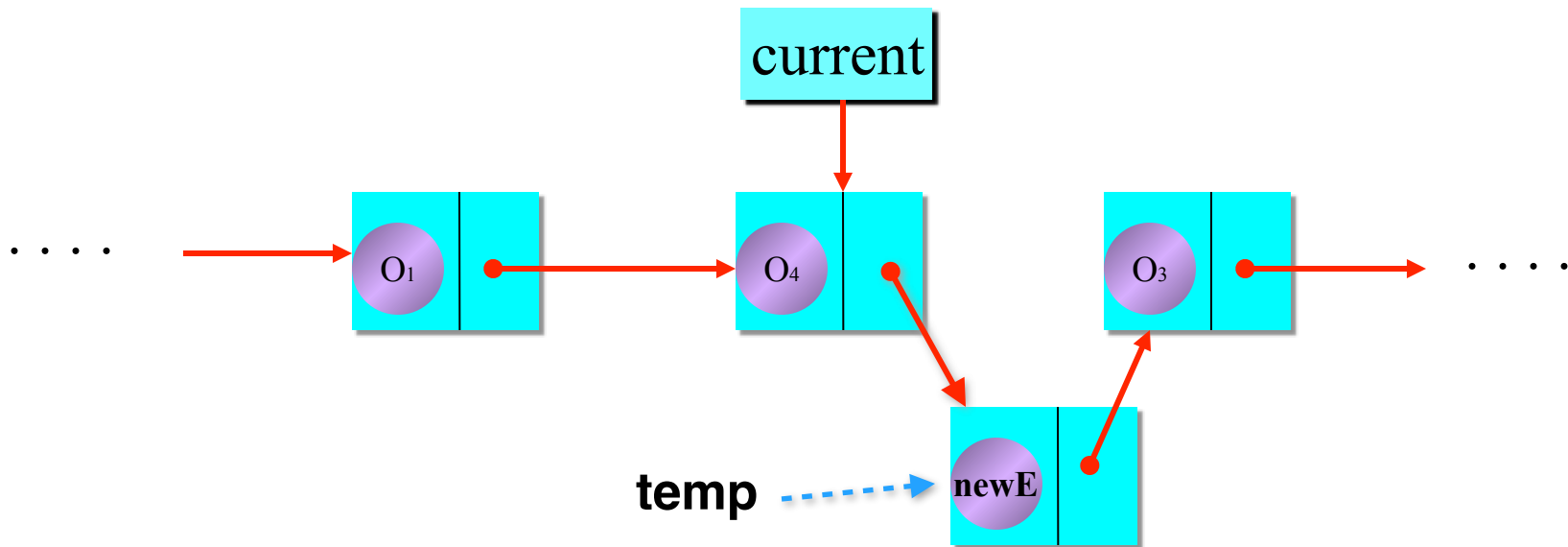
```
...  
➔ ListNode<T> temp = new ListNode<T>();  
temp.element = new E;  
temp.next = current.next;  
current.next = temp;  
...
```



```
...  
ListNode<T> temp = new ListNode<T>();  
→ temp.element = newE;  
temp.next = current.next;  
current.next = temp;  
...
```

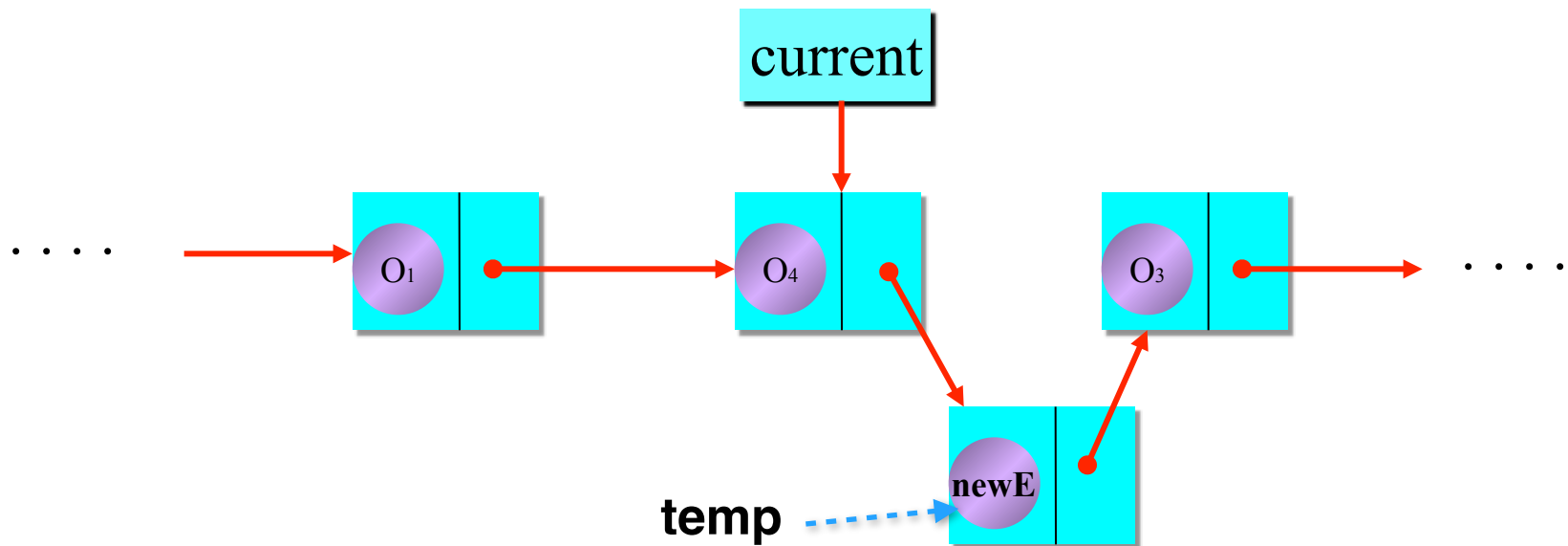


```
...  
ListNode<T> temp = new ListNode<T>();  
temp.element = newE;  
→ temp.next = current.next;  
current.next = temp;  
...
```

```
...  
ListNode<T> temp = new ListNode<T>();  
temp.element = newE;  
temp.next = current.next;  
➡ current.next = temp;  
...
```

Einfügen an eine beliebige Stelle der Liste

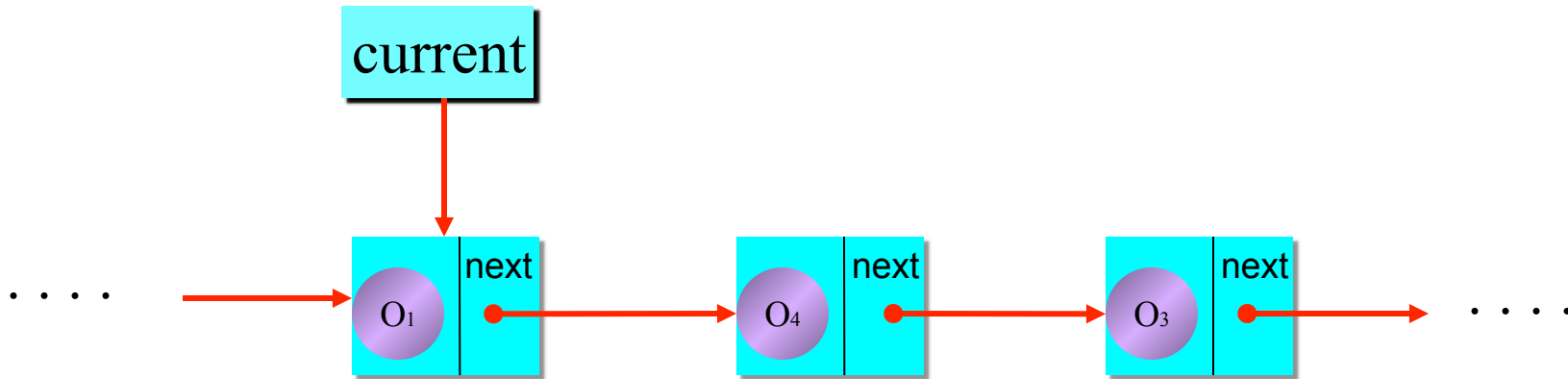


```

...
➔ current.next = new ListNode<T>( newE, current.next );
...

```

Löschen an einer beliebigen Stelle der Liste

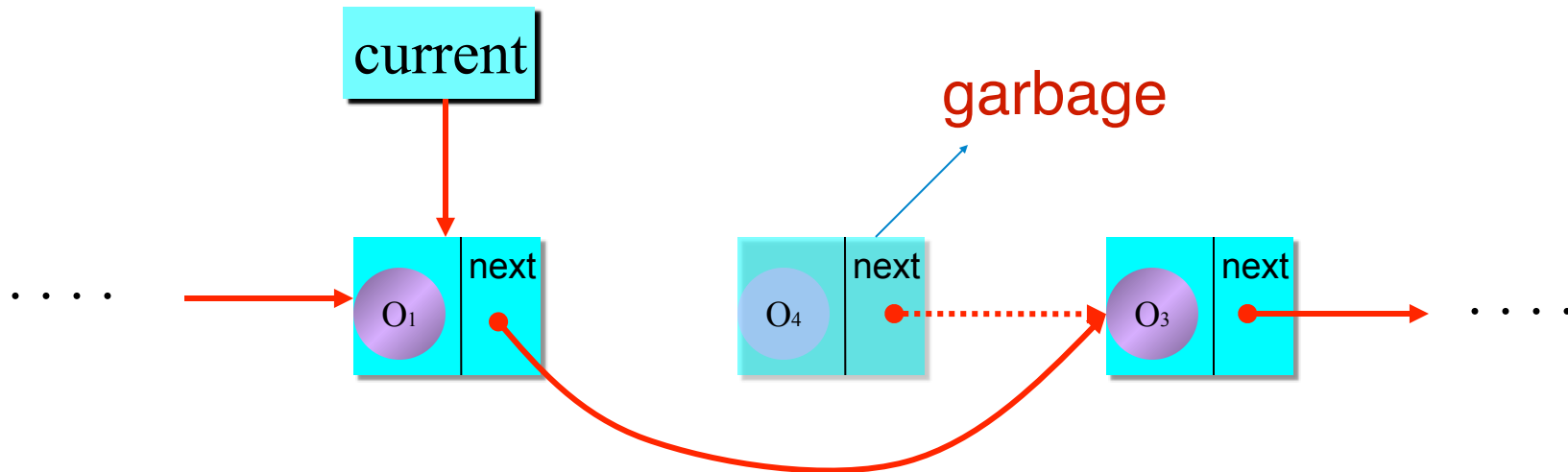


...

```
current.next = current.next.next;
```

...

Löschen an einer beliebigen Stelle der Liste



...

```
current.next = current.next.next;
```

...

Das entfernte **ListNode**-Objekt bleibt ohne eine einzige Referenz, die auf es zeigt, und verwandelt sich in Datenspeichermüll, der später von dem Java-“garbage collector“ beseitigt wird.

"Header"-Knoten ("Dummy"-Knoten)

Die Einfüge- und Löschooperationen, wie wir bis jetzt diskutiert haben, gehen davon aus, dass immer ein Vorgänger-Element vorhanden ist. Das macht unsere Implementierung einfacher, weil die speziellen Fälle

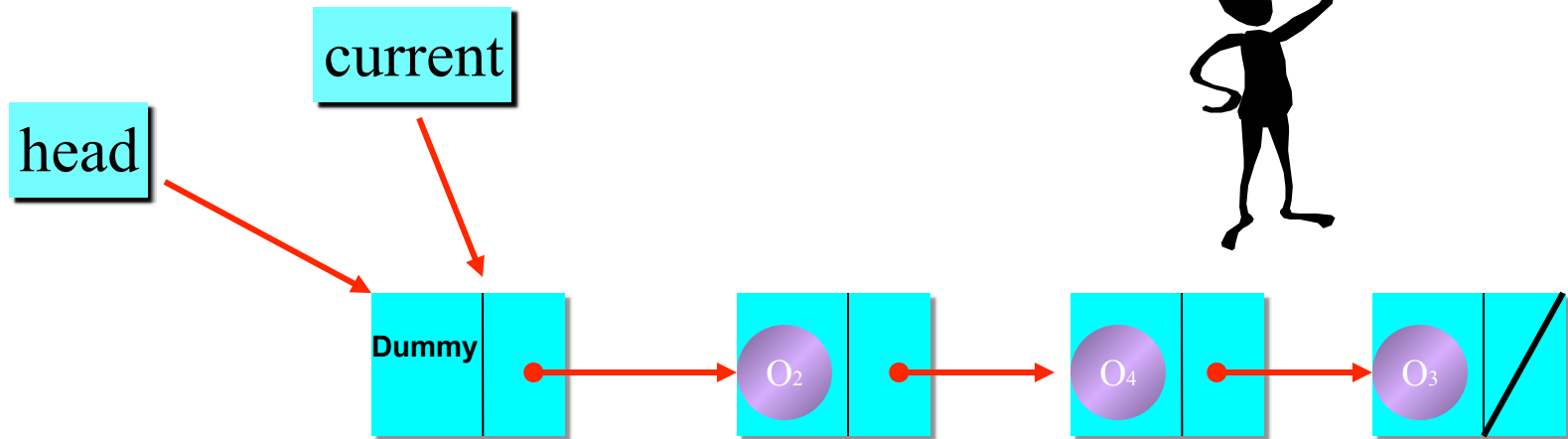
- Löschen des ersten Elements der Liste -

- Einfügen, wenn die Liste leer ist -

nicht berücksichtigt werden müssen.

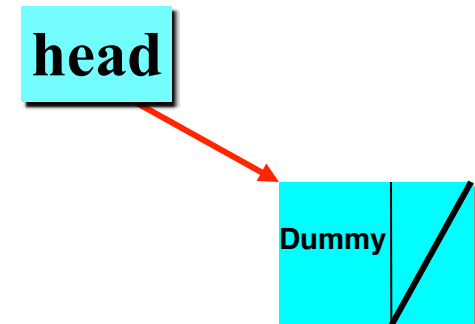
Um unsere Implementierung übersichtlich und einfach zu halten, benutzen wir einen **Dummy**-Knoten (**sentinel**), der selbst keine Elemente speichert und nur benutzt wird, um diese speziellen Fälle zu vermeiden, weil auf diese Weise jeder Knoten der Liste immer einen Vorgänger haben wird.

"Dummy"-Knoten



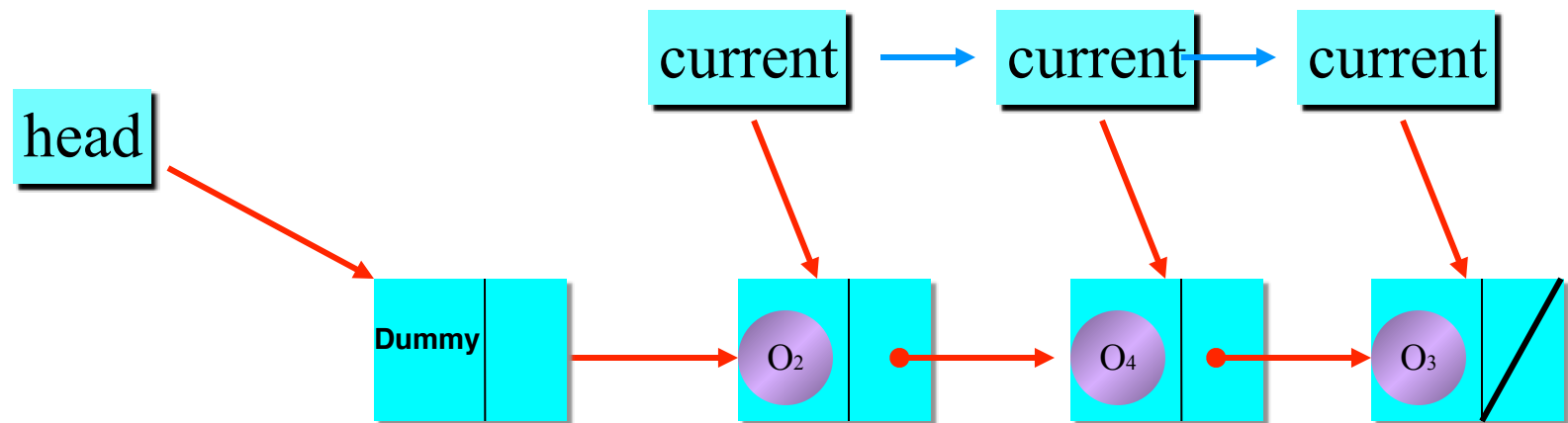
Unsere Liste ist jetzt leer, wenn sich nur der "**Dummy**"-Knoten in ihr befindet.

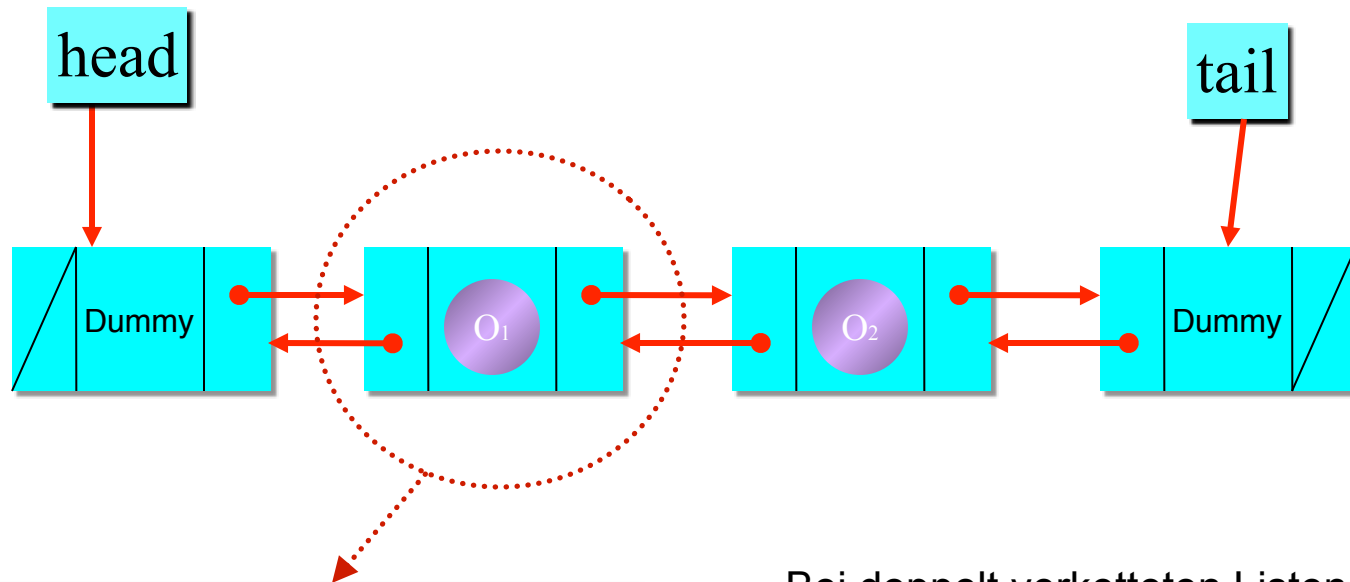
```
public boolean empty () {  
    return head.next == null;  
}
```



Probleme:

- Der **current**-Zeiger kann **nur nach vorne** bewegt werden.
- Um den Vorgänger desjenigen Knotens zu finden, auf den **current** zeigt, müssen wir die ganze Liste von **head** an durchlaufen.
- Das ist im allgemeinen **sehr ineffizient**.

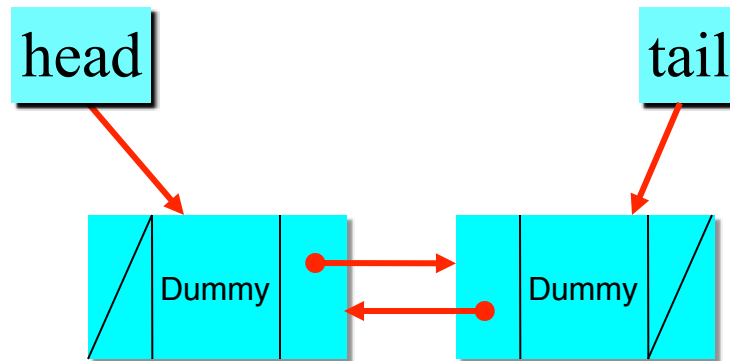




```
class ListNode <T> {  
    T element;  
    ListNode <T> next;  
    ListNode <T> prev;  
    ...  
}
```

Bei doppelt verketteten Listen kann sich **current** problemlos in beide Richtungen bewegen. Die Einfüge- und Löschoption ändert sich, weil wir jetzt immer eine doppelte Verkettung erstellen müssen.

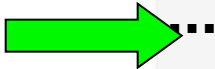
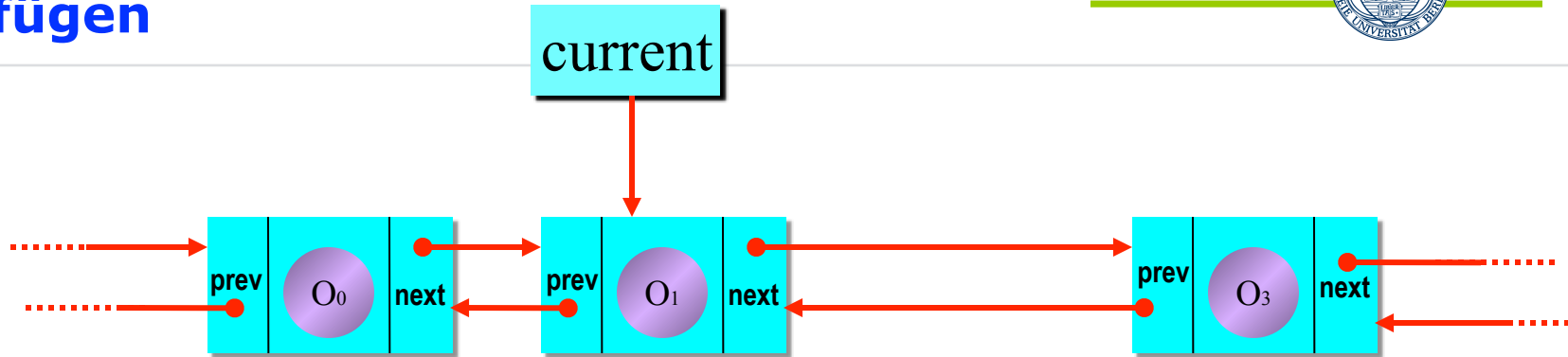
Doppelt verkettete leere Liste



Bei doppelt verketteten Listen ist es sinnvoll, einen Zeiger auf das letzte Element zu haben.

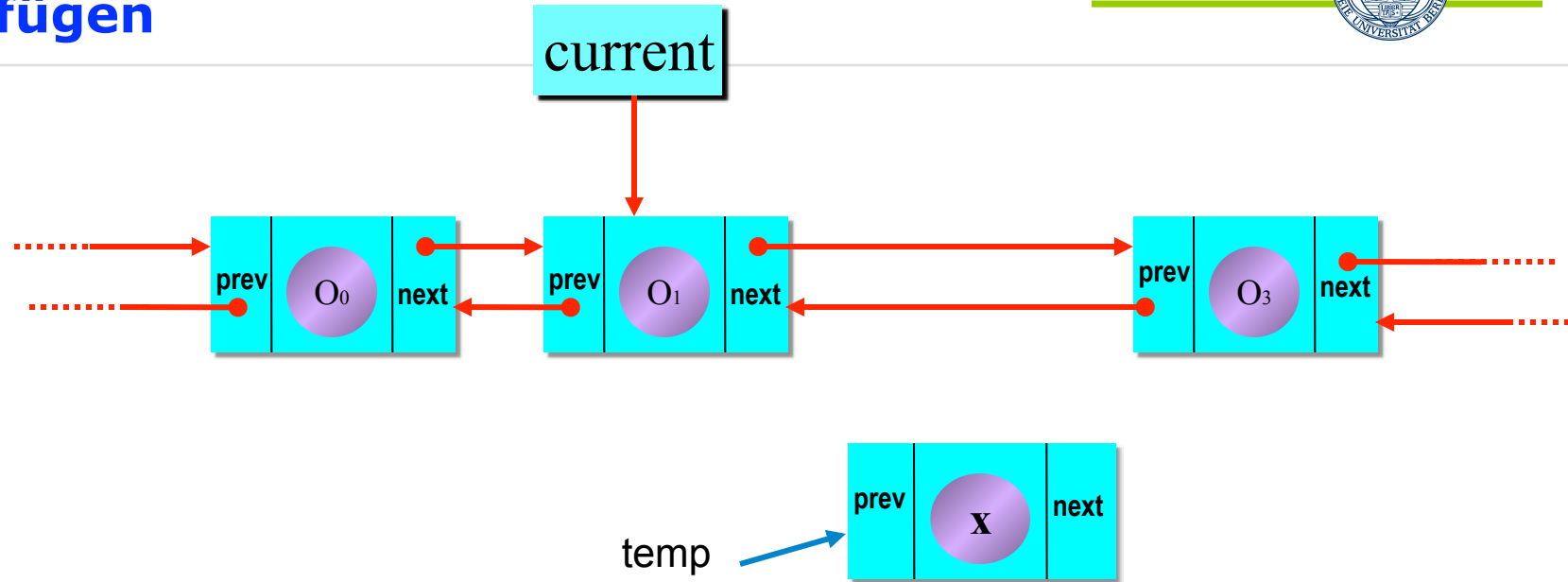
```
public boolean empty {  
    return head.next == tail;  
}
```

Einfügen



```
ListNode<T> temp = new ListNode<T>( x );  
temp.prev = current;  
temp.next = current.next;  
temp.prev.next = temp;  
temp.next.prev = temp;  
current = temp;
```

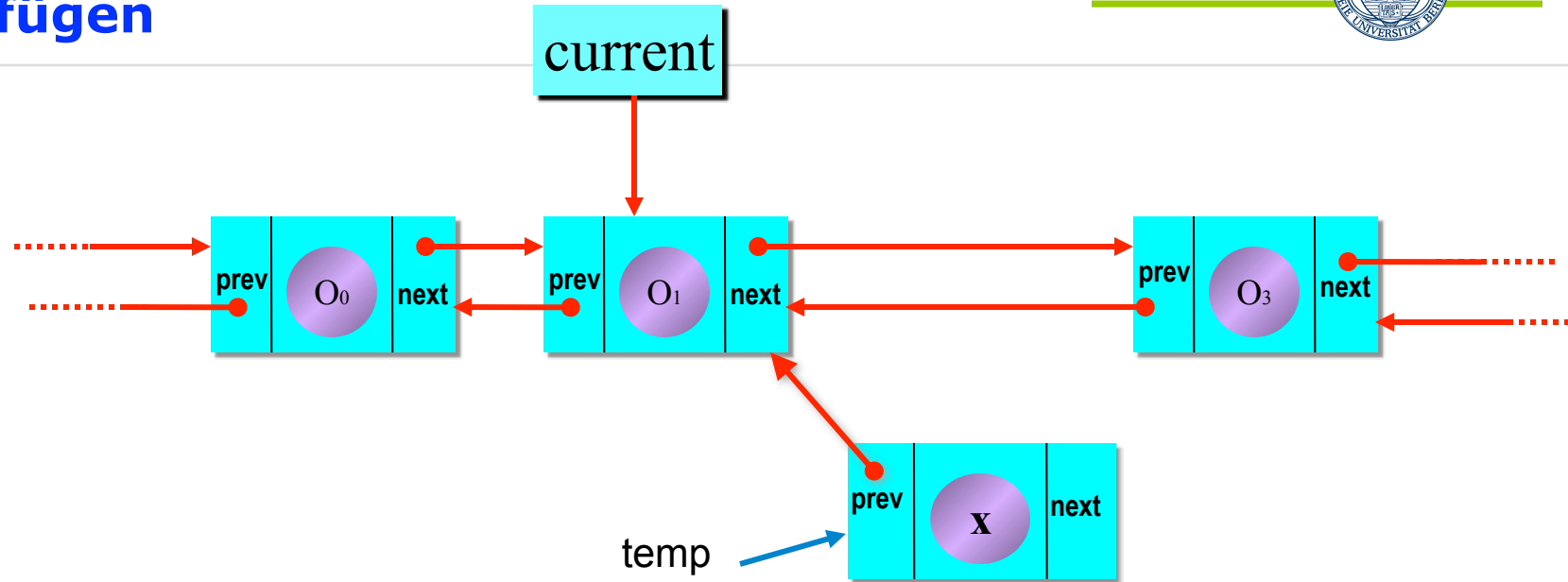
...



...

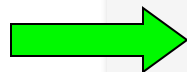
➔ **ListNode<T> temp = new ListNode<T>(x);**
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;

...



...

```
ListNode<T> temp = new ListNode<T>( x );
```



```
temp.prev = current;
```

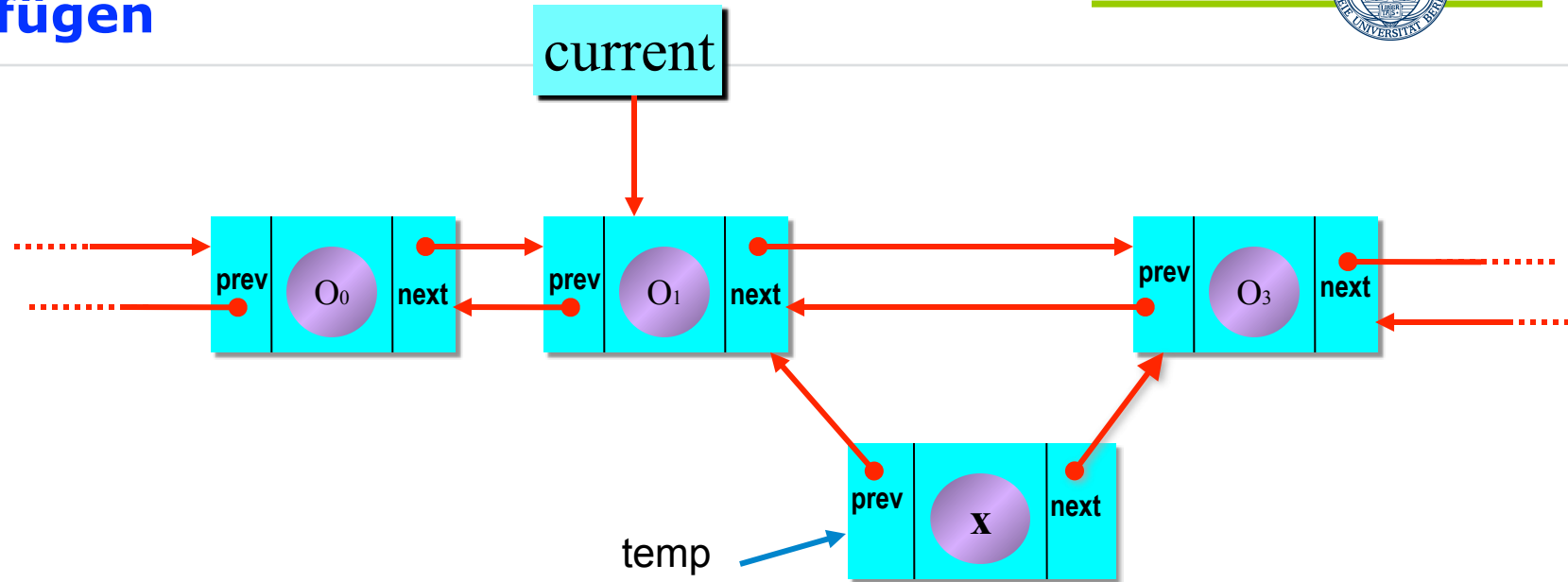
```
temp.next = current.next;
```

```
temp.prev.next = temp;
```

```
temp.next.prev = temp;
```

```
current = temp;
```

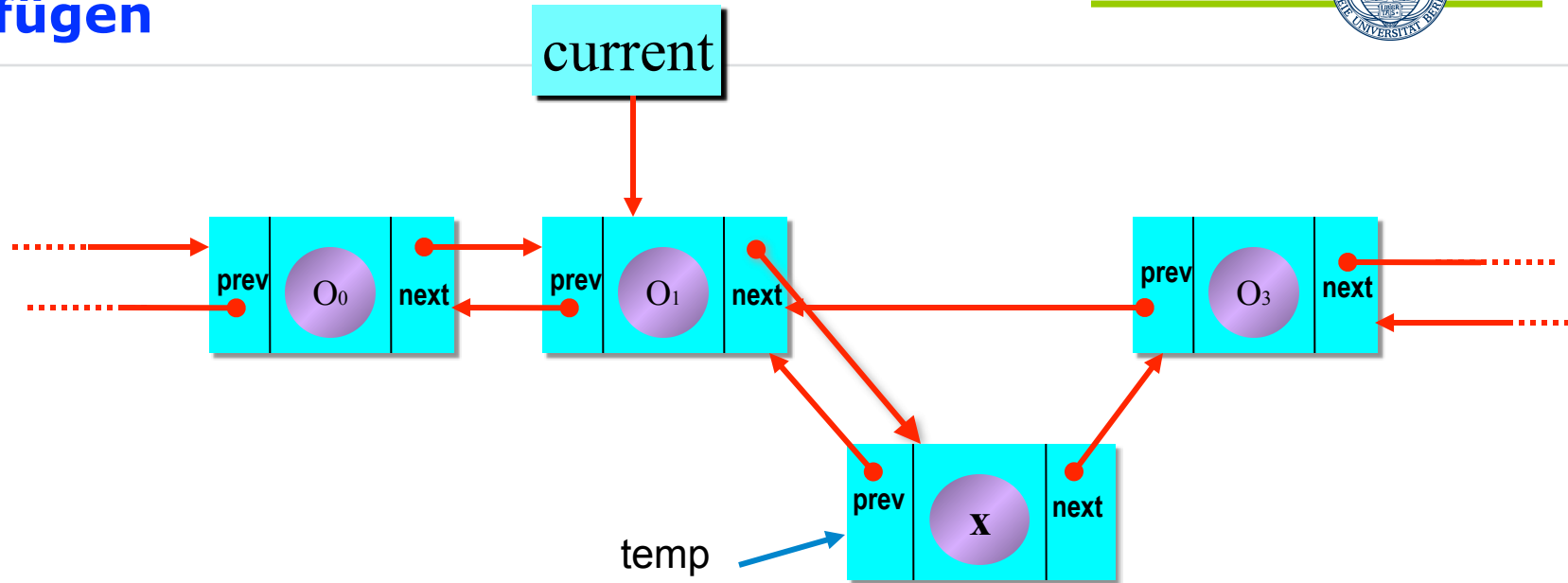
...



...

```
ListNode<T> temp = new ListNode<T>( x );  
temp.prev = current;  
temp.next = current.next;  
temp.prev.next = temp;  
temp.next.prev = temp;  
current = temp;
```

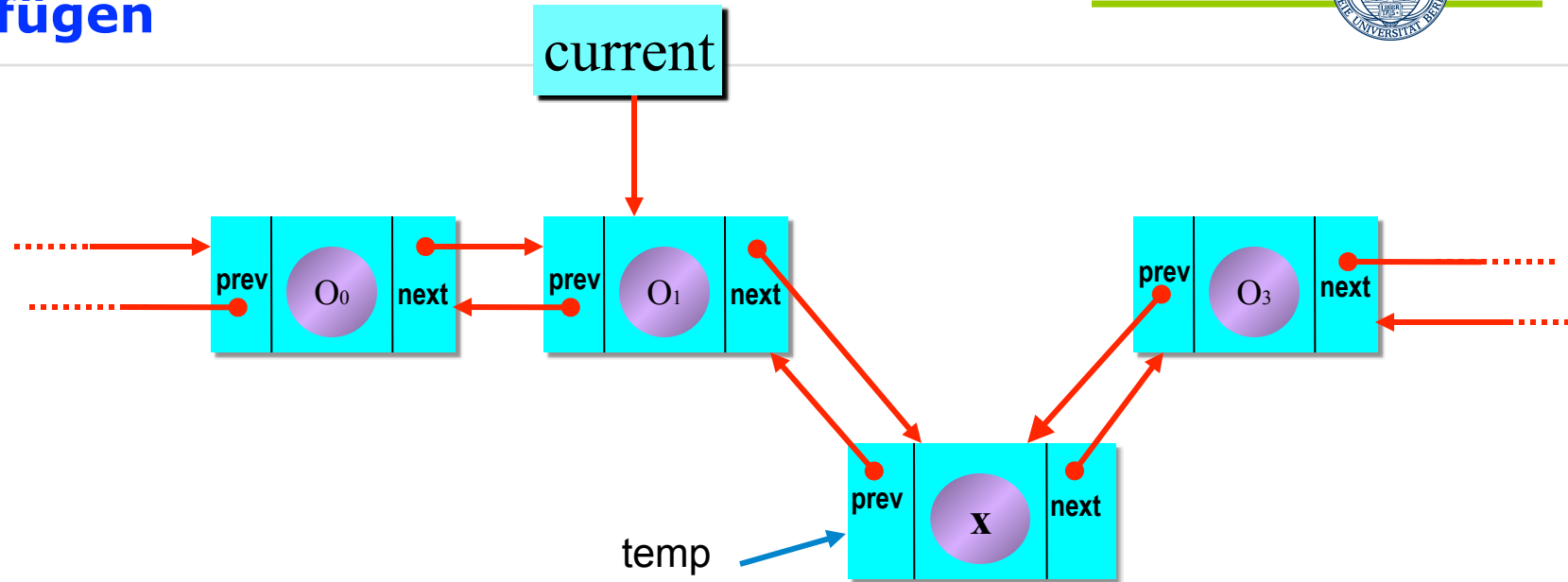
...



...

```
ListNode<T> temp = new ListNode<T>( x );  
temp.prev = current;  
temp.next = current.next;  
→ temp.prev.next = temp;  
temp.next.prev = temp;  
current = temp;
```

...

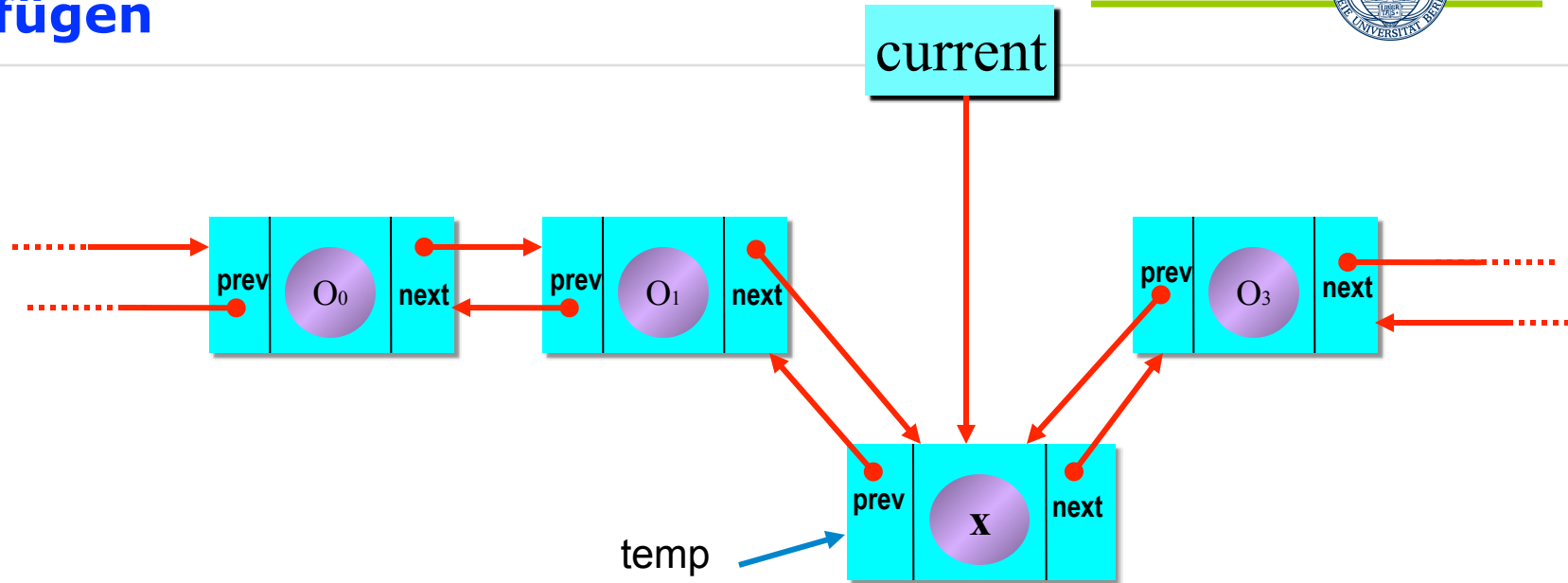


...

```
ListNode<T> temp = new ListNode<T>( x );  
temp.prev = current;  
temp.next = current.next;  
temp.prev.next = temp;  
temp.next.prev = temp;  
current = temp;
```

...

Einfügen

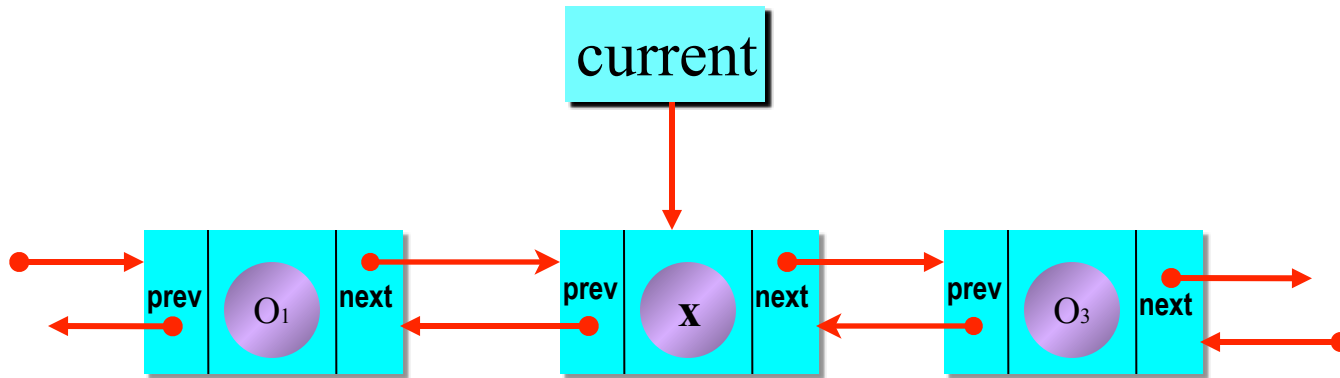


...

```
ListNode<T> temp = new ListNode<T>( x );  
temp.prev = current;  
temp.next = current.next;  
temp.prev.next = temp;  
temp.next.prev = temp;  
current = temp;
```

...

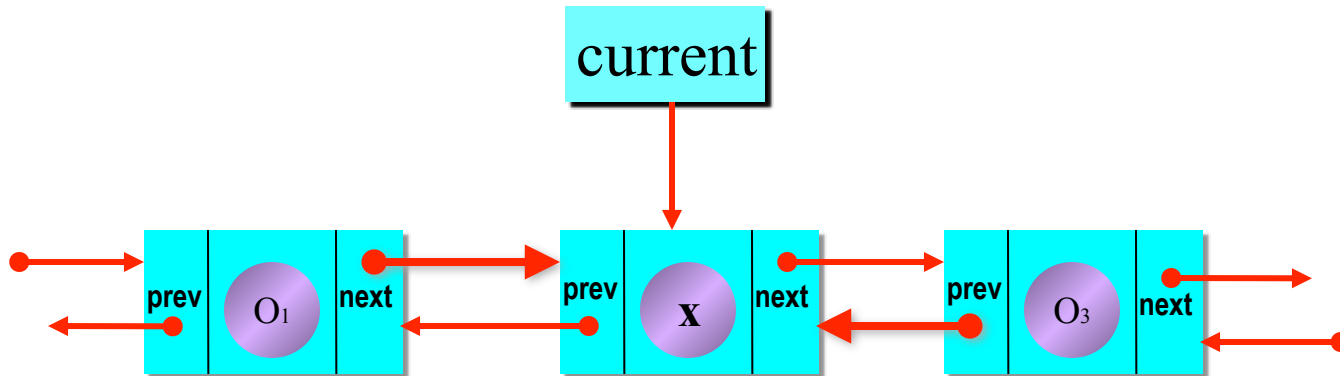
Löschen



Mit doppelt verketteten Listen können wir genau das Objekt löschen, auf das **current** zeigt.

```
...  
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;  
...
```

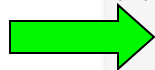
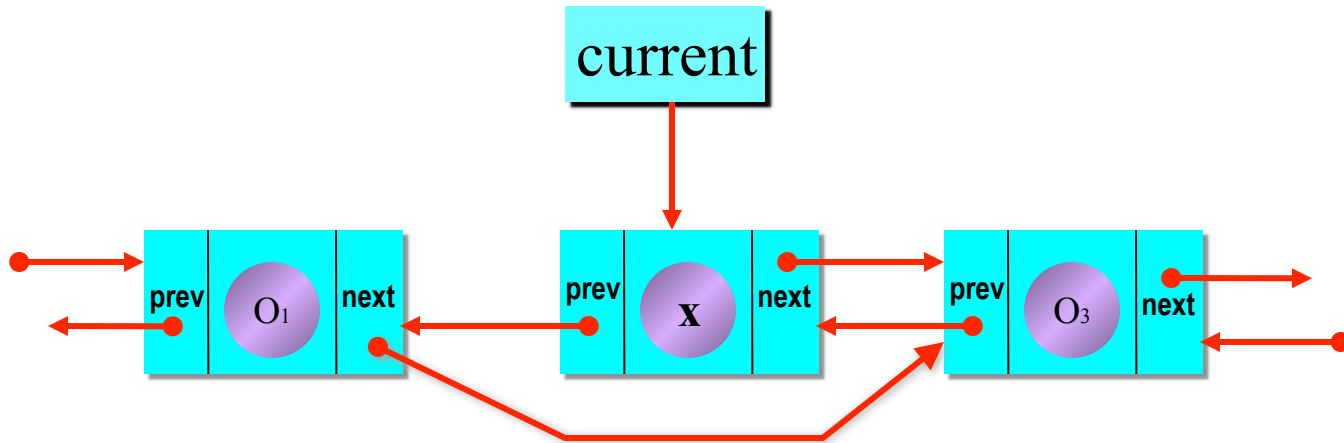
Löschen



Mit doppelt verketteten Listen können wir genau das Objekt löschen, auf das **current** zeigt.

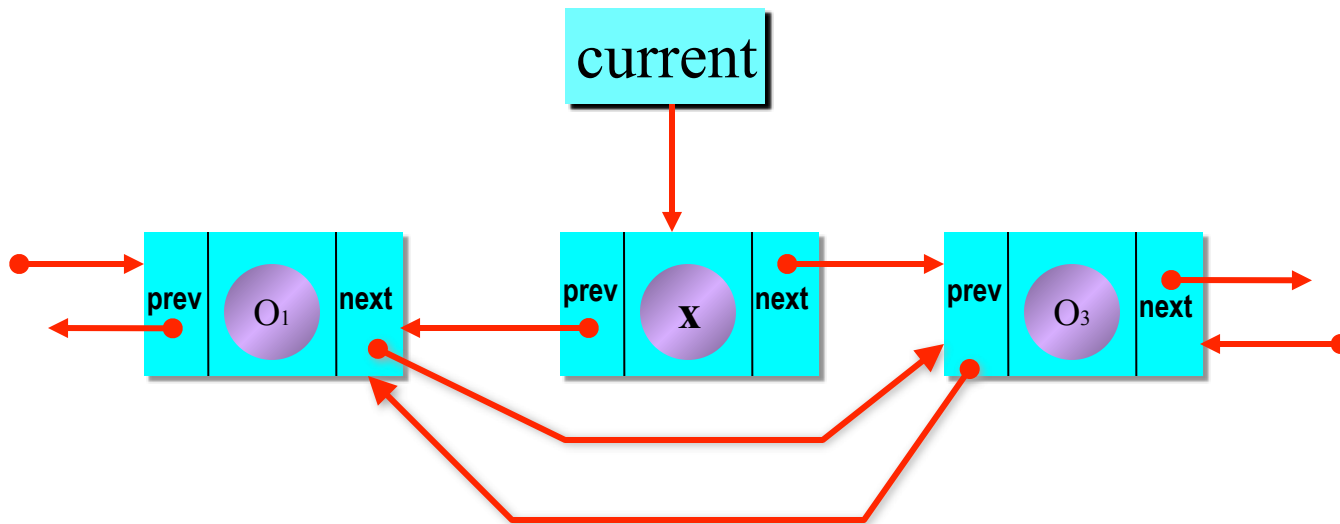
```
...  
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;  
...
```

Löschen



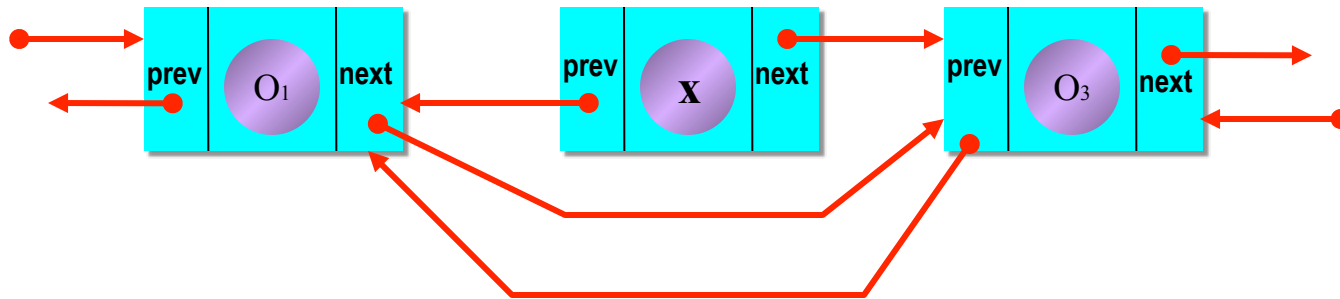
```
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;
```

Löschen

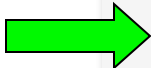


```
...  
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;  
...
```

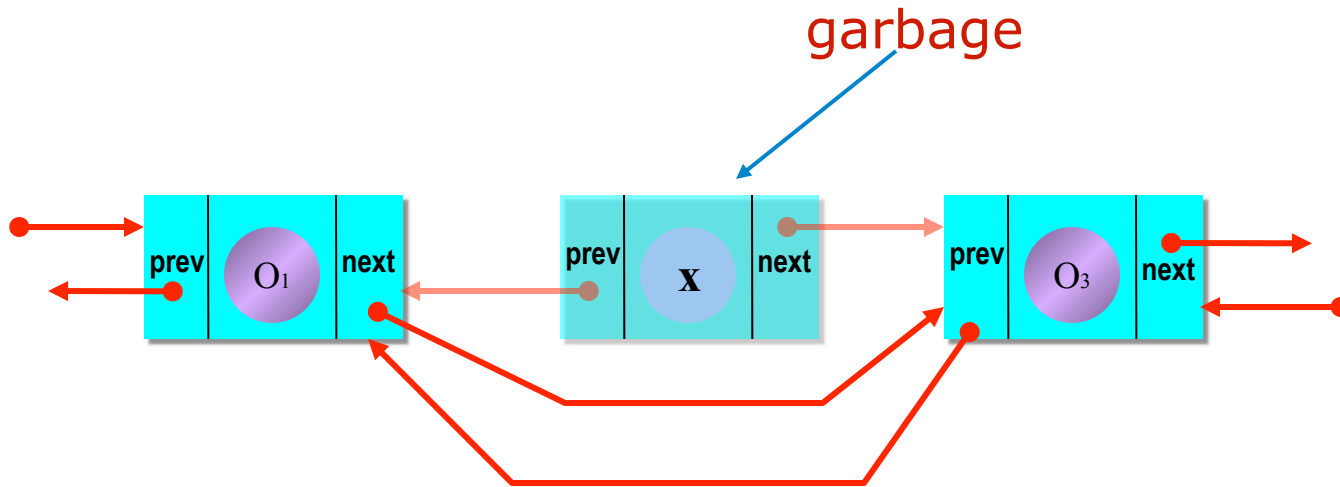
Löschen



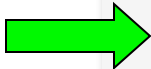
```
...  
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;  
...
```



Löschen



```
...  
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = head;  
...
```

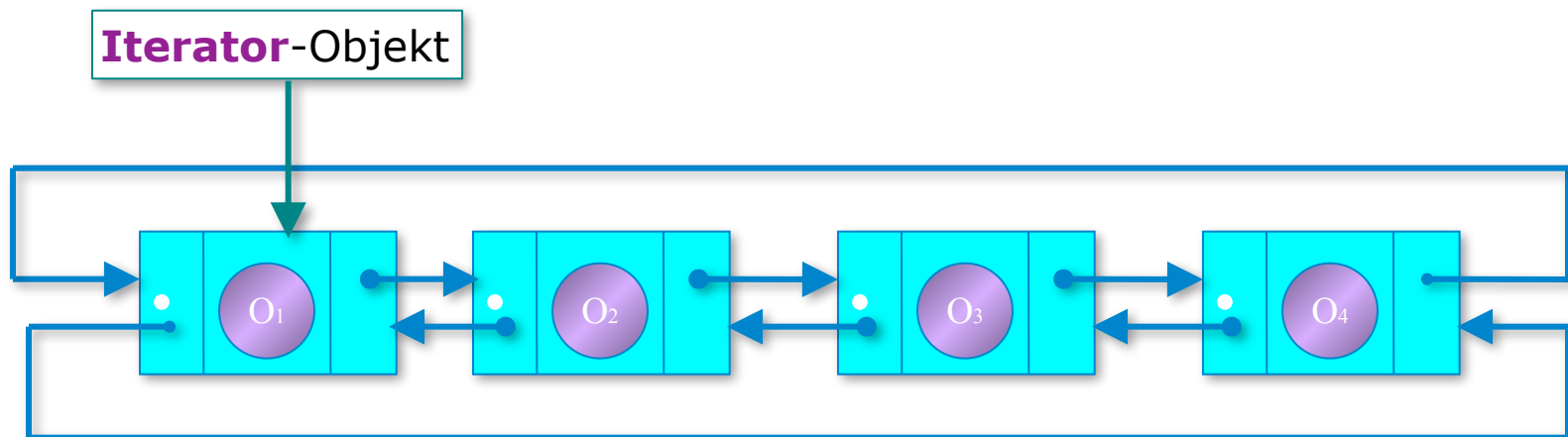


Zusammenfassung

1. Wir haben gesehen, wie **Stapel** und **Warteschlange** mit Hilfe von einfach verketteten Listen implementiert werden können.
2. Wir haben grundlegende Programmiertechniken für die Implementierung von **dynamischen Datenmengen** mit Hilfe von **einfach** und **doppelt verketteten Listen** besprochen.
3. Wir haben gesehen, wie die Anzahl der Fälle in unseren Algorithmen für die **Einfüge** und **Lösch**-Operationen reduziert werden kann mit der Einführung von **Dummy-Knoten**.

Dynamische Datenmengen

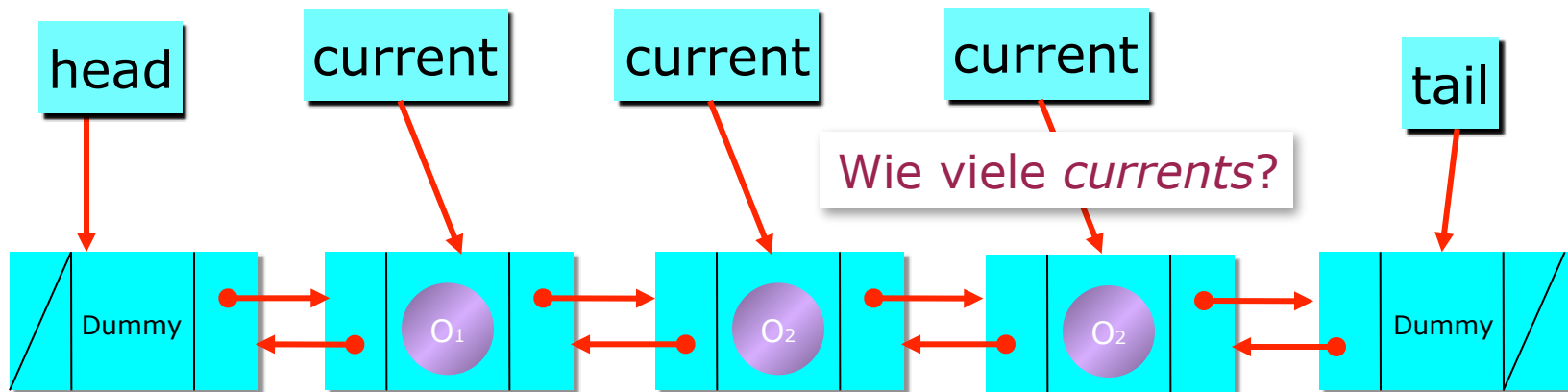
Iteratoren



Iteratoren

Motivation:

Wir haben für die Implementierung dynamischer Datenmengen mittels verketteter Listen einen **current**-Zeiger als Teil unserer Listen-Objekte verwendet, der sich durch die Liste bewegen kann und diese Position für weitere Operationen bereitstellt.



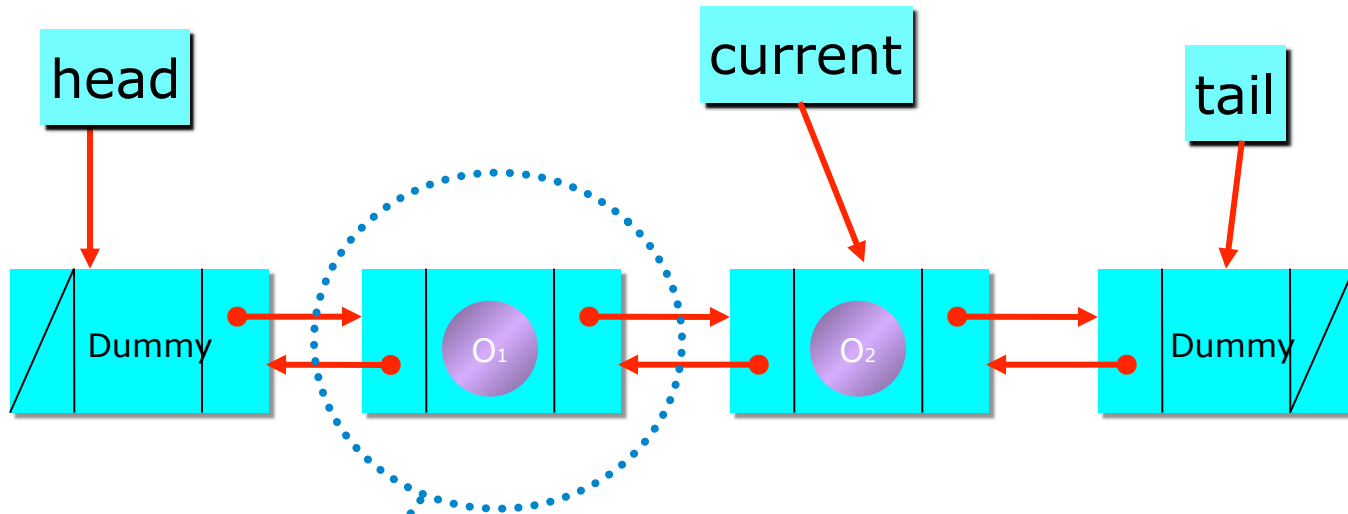
Oft müssen wir für bestimmte Algorithmen mehr als eine Position in einer dynamischen Datenmenge gleichzeitig festhalten.

Iteratoren

Motivation:

Wenn wir den Benutzern unserer Klassen erlauben, beliebige solcher Zeiger selber zu definieren und dadurch direkten Zugriff auf die interne Struktur einer dynamischen Datenmenge (z.B. auf die Listenknoten einer Liste) zu haben, würden wir nicht nur das **Konzept der Kapselung verletzen**, sondern die Gefahr, dass die Benutzer **Fehler** programmieren, wäre zu groß.

Doppelt verkettete Listen



```
class ListNode <T> {
    T element;
    ListNode <T> next;
    ListNode <T> prev;
    ...
}
```

Um an einer beliebigen Stelle der Liste Elemente einfügen und löschen zu können, brauchen wir ein Referenz-Objekt (**current**), das sich durch die Liste bewegt.

Doppelt verkettete Listen

```
class ListNode <T> {
    T element;
    ListNode <T> next;
    ListNode <T> prev;
    ...
}
```

```
public class DoubleChainList<T> {

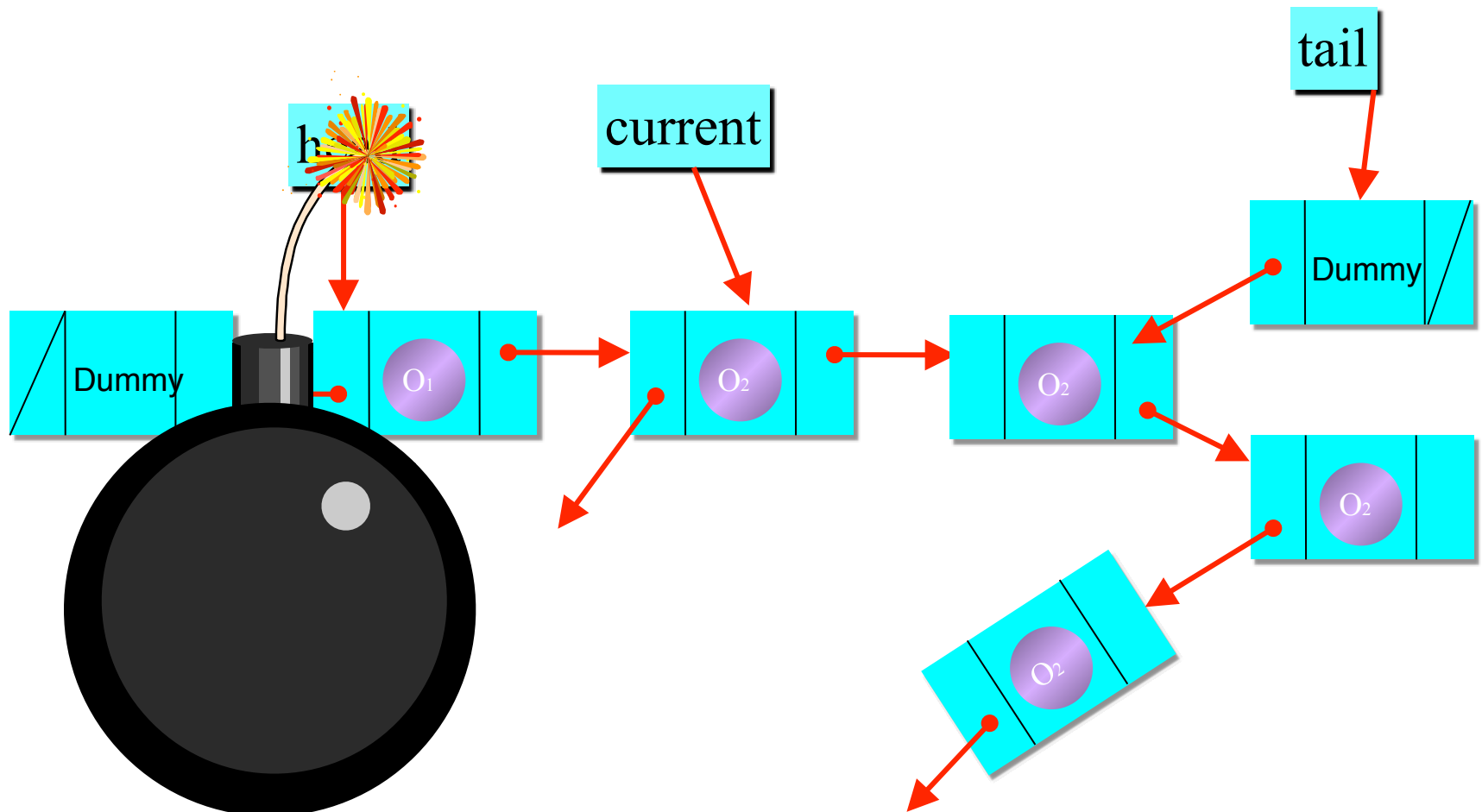
    private ListNode<T> head;
    private ListNode<T> tail;
    private ListNode<T> current;

    public DoubleChainList () {
        head = tail = null;
    }
    /* weitere Methoden */ ...
}
```

Dynamische Datenstrukturen sind sehr empfindlich.

Ein Programmierer kann z.B. sehr leicht die Knoten-Referenzen zweier verschiedener Listen verwechseln und die Listen unwiederbringlich beschädigen, weil durch die rekursive Definition der Knoten einer Liste, der Zeiger auf den Listenanfang, der Zeiger der die Liste durchläuft und der Zeiger, der zwei Knoten der Liste verkettet, den gleichen Datentyp haben.

Doppelt verkettete Listen



Iteratoren

Lösung:

Die **objektorientierte Lösung** zu diesem Problem ist die Verwendung von **Iteratoren**.

Iteratoren sind Objekte, die eine Datenstruktur erhalten und Operationen zur Verfügung stellen, mit Hilfe derer man diese Datenstruktur durchlaufen kann.

Iteratoren

Iteratoren sind ein Softwarepattern zur Datenkapselung. Sie abstrahieren von der darunter liegenden Repräsentation der Daten.

Beispiel:

```
public interface Iterator <E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

Das Iteratormuster

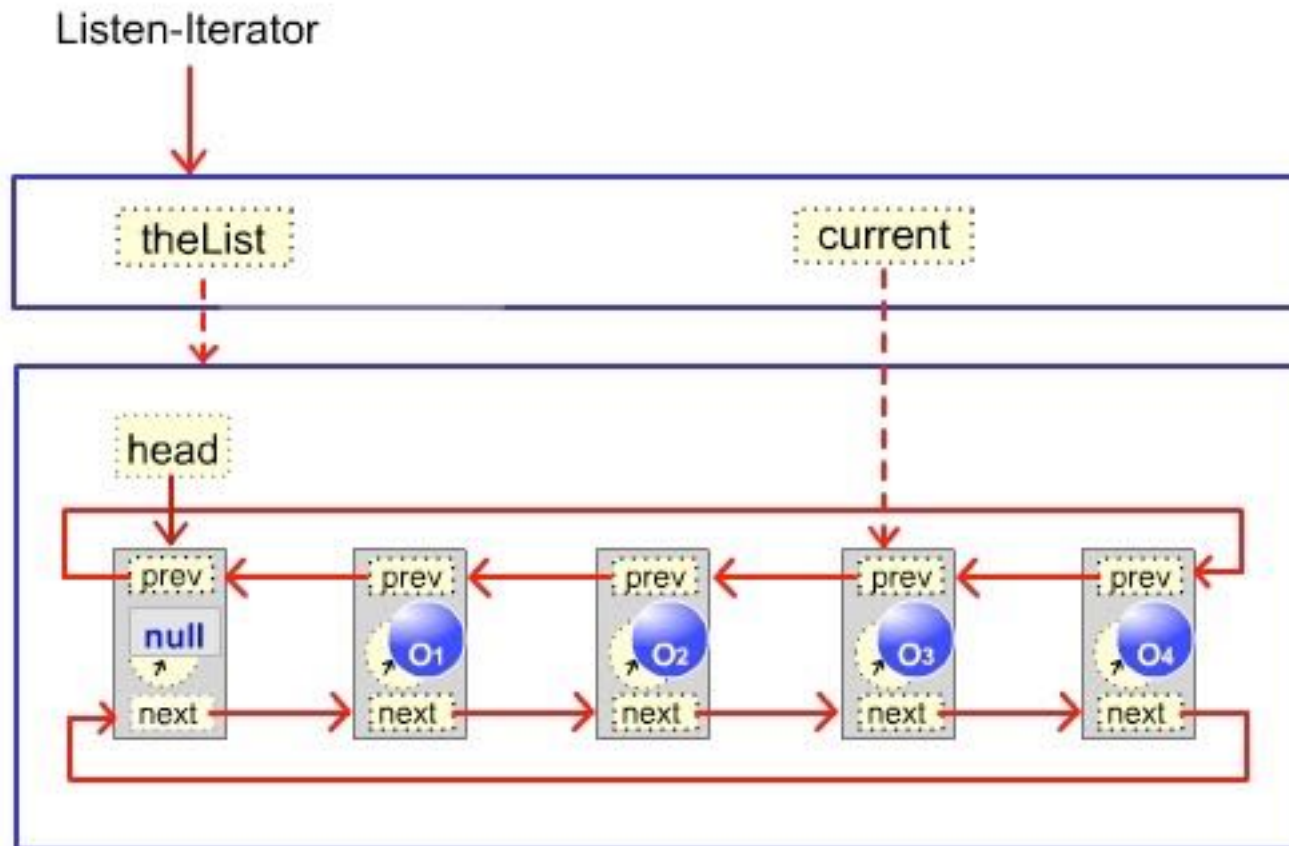
Iterator Pattern

Um eine **einheitliche Schnittstelle** zur Traversierung unterschiedlicher zusammengesetzter Strukturen (das heißt, um **polymorphe Iteration**) zu ermöglichen, bieten die Standardbibliotheken vieler Sprachen das **Iteratormuster**.

Mehrere Iteratoren können mit der gleichen Datenstruktur gestartet werden, und der Benutzer hat somit keinen direkten Zugriff auf die interne Struktur der dynamischen Datenmenge wie z.B. auf den internen **current**-Zeiger einer Liste.

Das Iteratormuster

Grundprinzip



Die Java-Implementierung des Iteratormusters

Die Java-Bibliothek bietet eine Interface für die Implementierung von Iteratoren. Man muss allerdings folgende Probleme beachten:

1. Der Iterator muss immer seinen Zustand innerhalb der Iteration speichern.
2. Die Iteratoren sollen invalidiert werden, wenn sich die Sammlung oder die Datenstruktur durch Einfüge- oder Löschoperationen verändert hat.

(**CurrentModificationException**).

Das Iteratormuster von Java

Einfaches

Beispiel:

```
public class LinkedList<T> implements Iterable<T> {  
    ListNode<T> head;  
    ListNode<T> tail;  
    ... // List-Konstruktor fehlt! Methode und kein Konstruktor!  
  
    public ListIterator<T> iterator() {  
        return new ListIterator<T>(head);  
    }  
  
    public void insert( T elem ){  
        ListNode<T> temp;  
        if (head==null) {  
            temp = new ListNode<T>(elem,null,null);  
        }else{  
            temp = new ListNode<T>(elem,head,null);  
            head.prev = temp;  
        }  
        head = temp;  
    }  
    ...  
}
```

Das Iteratormuster von Java

Innere-Klasse

```
...
private class ListNode<E> {
    E elem;
    ListNode<E> next;
    ListNode<E> prev;

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

Das Iteratormuster von Java

Einfaches
Beispiel:

Innere-Klasse

```
...
class ListIterator<T> implements Iterator<T> {
    ListNode<T> current;
    ListIterator(ListNode<T> head) {
        current = head;
    }
    public boolean hasNext() {
        return (current != null);
    }
    public T next() {
        if (current == null){
            return null;
        }
        T ret = current.elem;
        current = current.next;
        return ret;
    }
    public void remove(){ // not implemented }
}
...
```

Das Iteratormuster von Java

```
public class TestListIterator {  
  
    public static void main( String[] args ){  
        LinkedList<Integer> ll = new LinkedList<Integer>();  
        ll.insert(1);  
        ll.insert(2);  
        ll.insert(3);  
        ll.insert(4);  
        ll.insert(1);  
        ll.insert(5);  
        ListIterator li = ll.iterator();  
        for ( ; li.hasNext() ; ) {  
            System.out.println( li.next() );  
        }  
    }  
}
```

Das Iteratormuster von Java

Beispiel:

```
public class PrimesIterator implements Iterator<Integer>{
    int current = 1;
    public boolean hasNext() {
        return true;
    }
    public Integer next() {
        while ( true ) {
            current++;
            for ( int i = 2; current % i != 0; i++) {
                if (i*i >= current) return current;
            }
        }
    }
    public void remove(){...}
}
```

generiert beliebige
Primzahlen >2

Das Iteratormuster von Java

Beispiel:

```
public class TestPrimesIterator {  
    . . .  
    public static void main(String[] args) {  
        PrimesIterator gen = new PrimesIterator();  
        int n = 2;  
        while(n<1000){  
            n = gen.next();  
            System.out.println(n);  
        }  
    }  
} // end of class TestPrimesIterator
```


Iteratoren

```
public interface Iterator <E>
```



```
public interface ListIterator <E>
```

```
public interface Iterator <E> {  
    void      add( E o )  
    boolean   hasNext();  
    boolean   hasPrevious()  
    E         next();  
    E         previous();  
    void      remove();  
    void      set( E o ) ;  
    int       nextIndex();  
    int       previousIndex()  
}
```

(optionale
Operationen)



UnsupportedOperationException
if the operation is not supported by
this list iterator