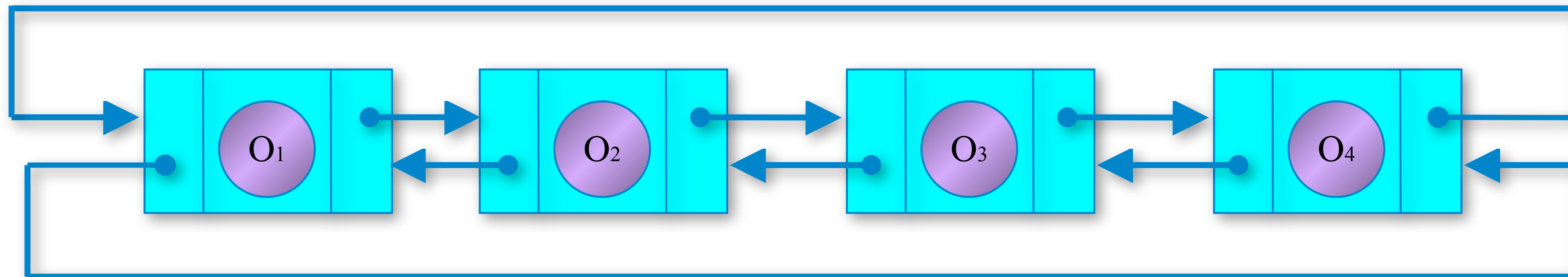
Freie Universität Berlin

ALP II

Dynamische Datenmengen

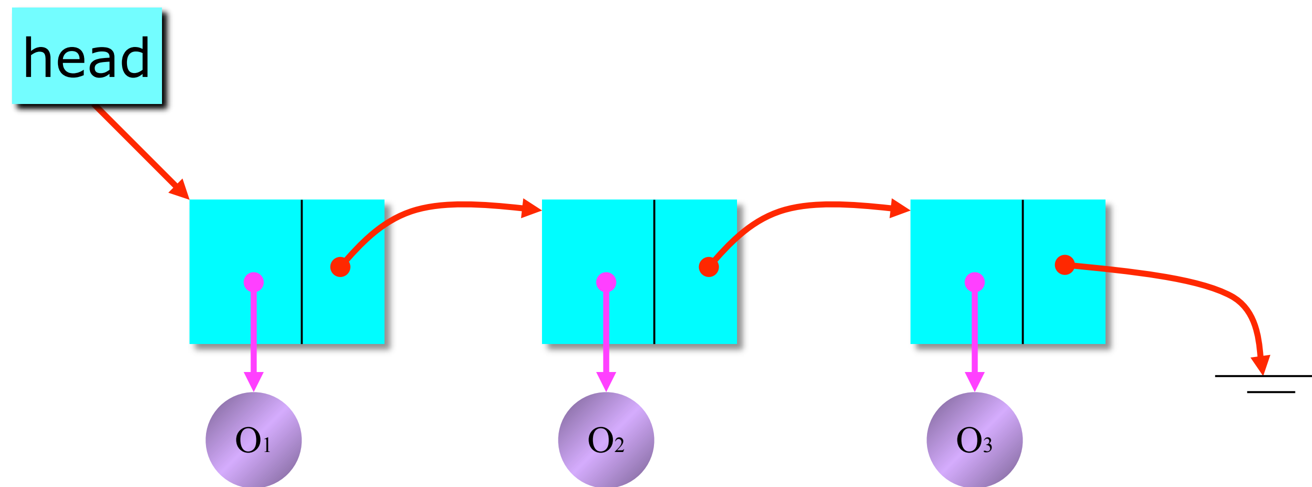
Datenabstraktion (Teil 2)



SS 2012

Prof. Dr. Margarita Esponda

Einfach verkettete Listen

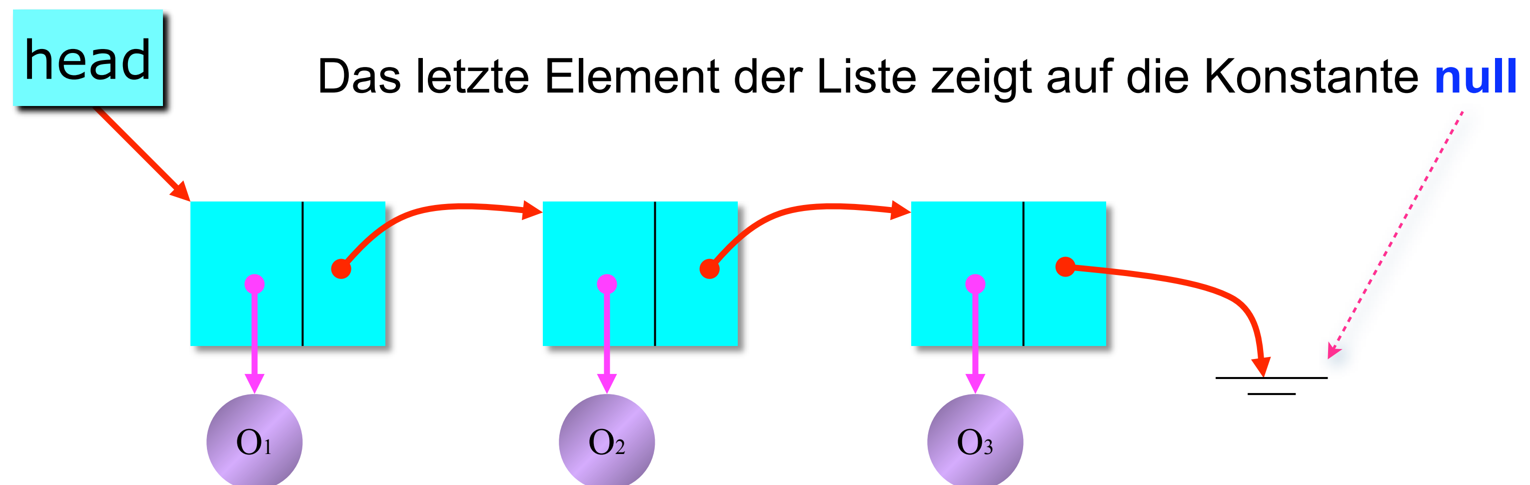


Einführung

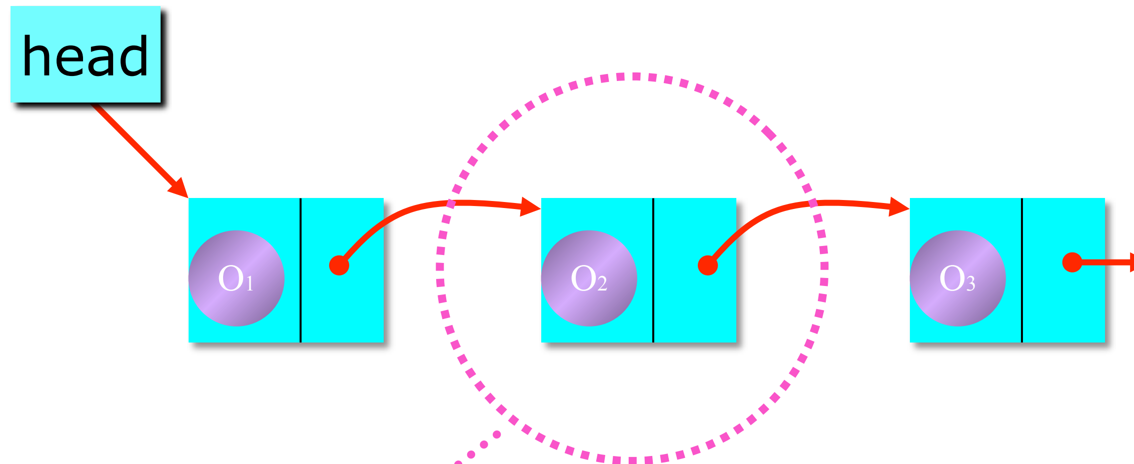
Einfach verkettete Listen sind die einfachsten dynamischen Datenstrukturen, die zur Laufzeit an den tatsächlichen Speicherbedarf anpassen können.

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.



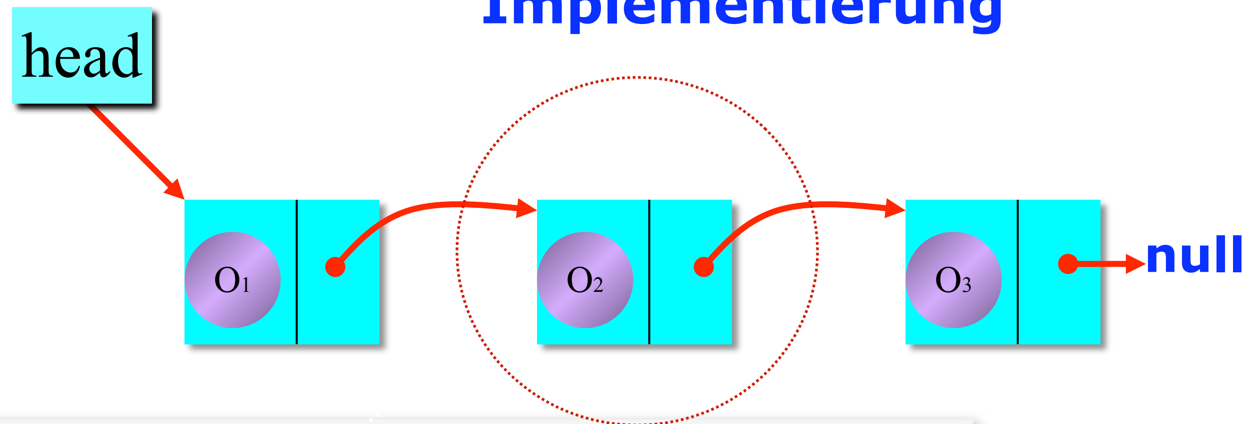
Implementierung




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

Wir haben eine **rekursive Klassendefinition**, weil das **next**-Element zu der gleichen Objekt-Klasse gehört, die wir gerade definieren.

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:

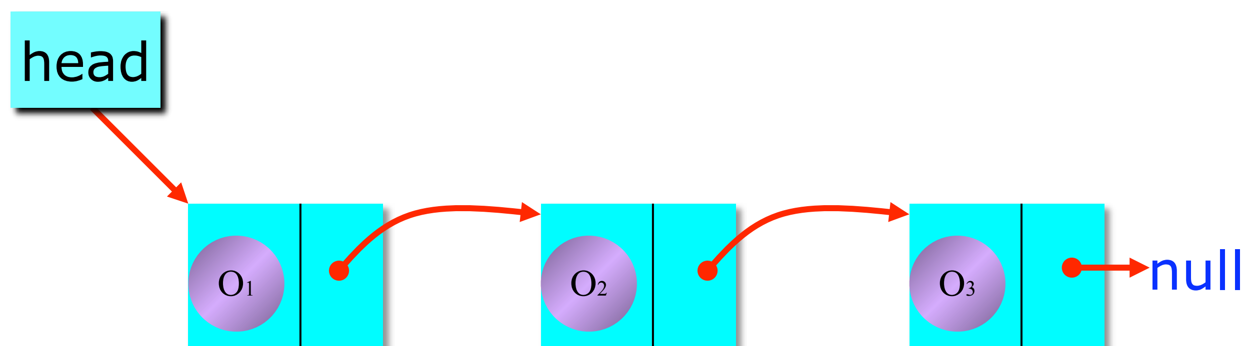
Einer, der nur einen leeren Knoten erzeugt und einen, der gleichzeitig ein Objekt in dem Knoten speichert und die Referenz auf den nächsten Knoten bekommt.

Stapel als verkettete Liste

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

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

Wir brauchen 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 verketten oder entfernen, wenn eine **pop**-Operation stattfindet.



Stapel-Schnittstelle

In dieser Implementierung werden wir eine EmptyStackException erzeugen bei dem Versuch, ein Element zu entfernen oder zu lesen (**pop**- und **top**-Operationen), wenn die Liste leer ist.

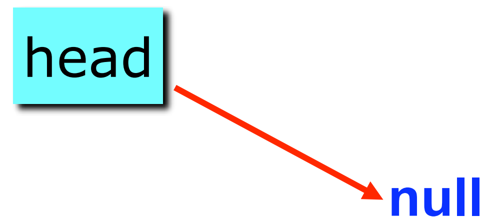
```
public interface Stack <T>{
    public void push( T element );
    public T pop() throws EmptyStackException;
    public T top() throws EmptyStackException;
    public boolean empty();
}
```

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 */
. . .
```

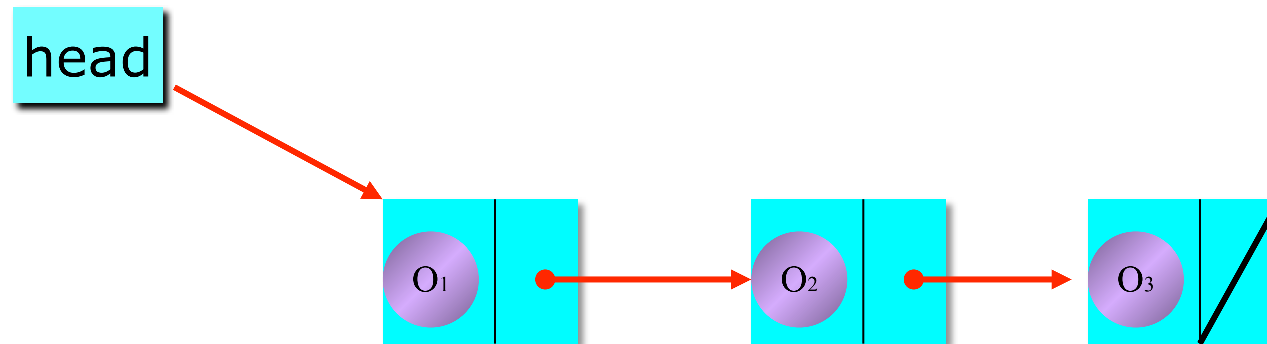
Implementierung der empty-Operation



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

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

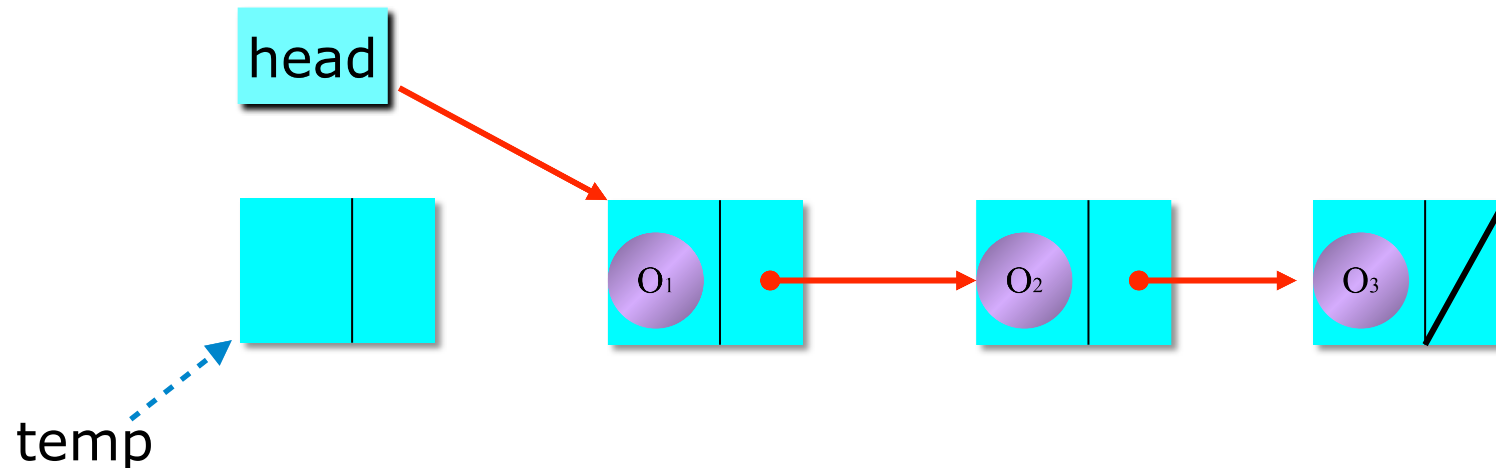
Implementierung der push-Operation



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

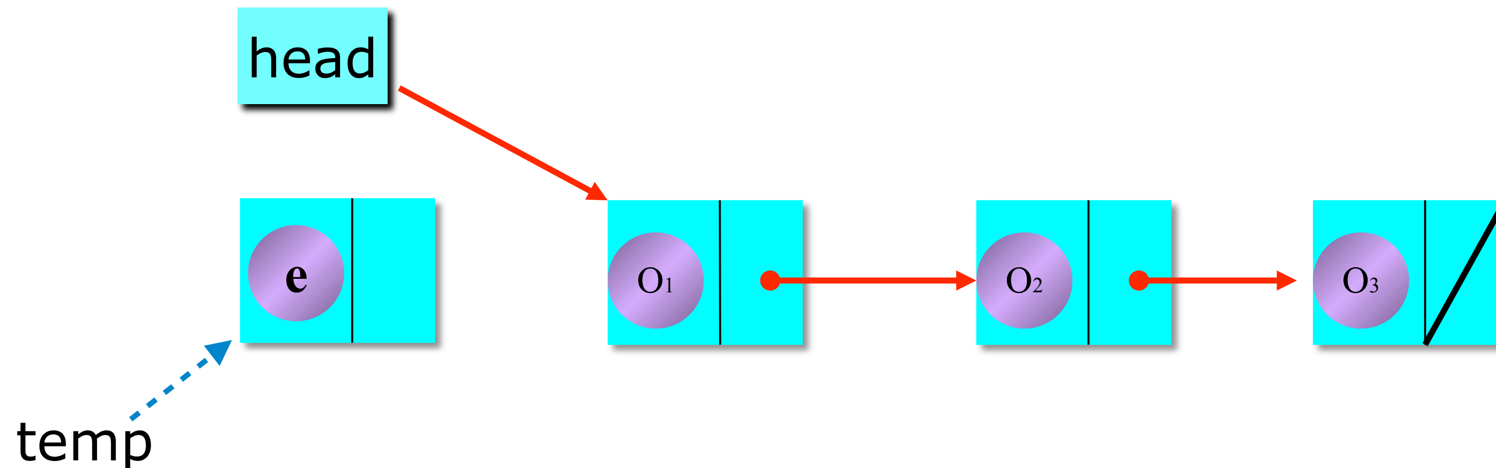
```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation



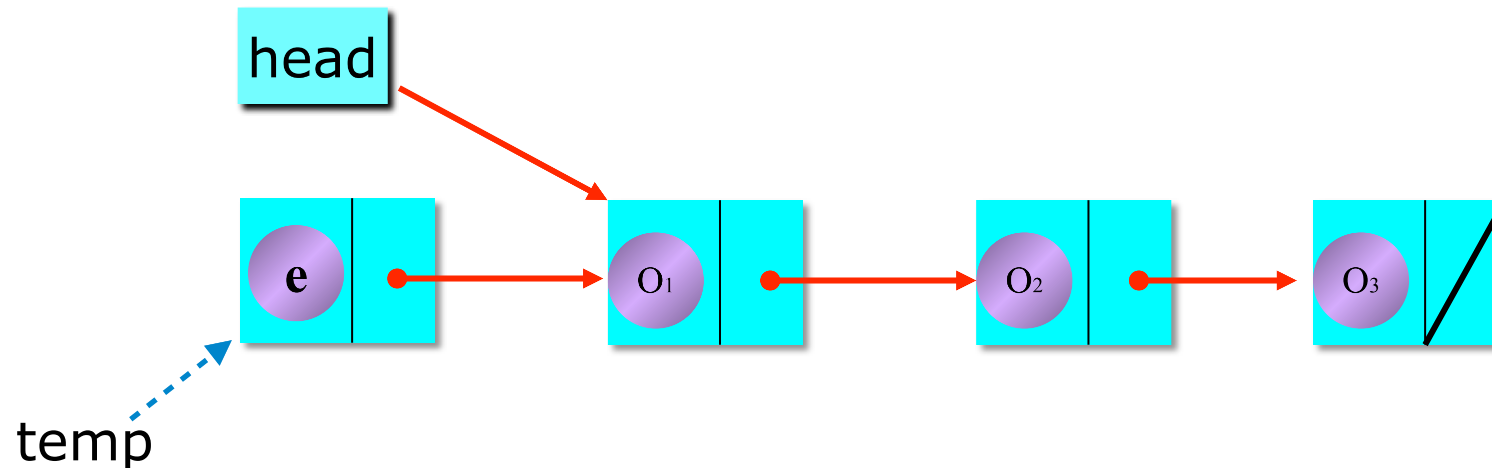

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation



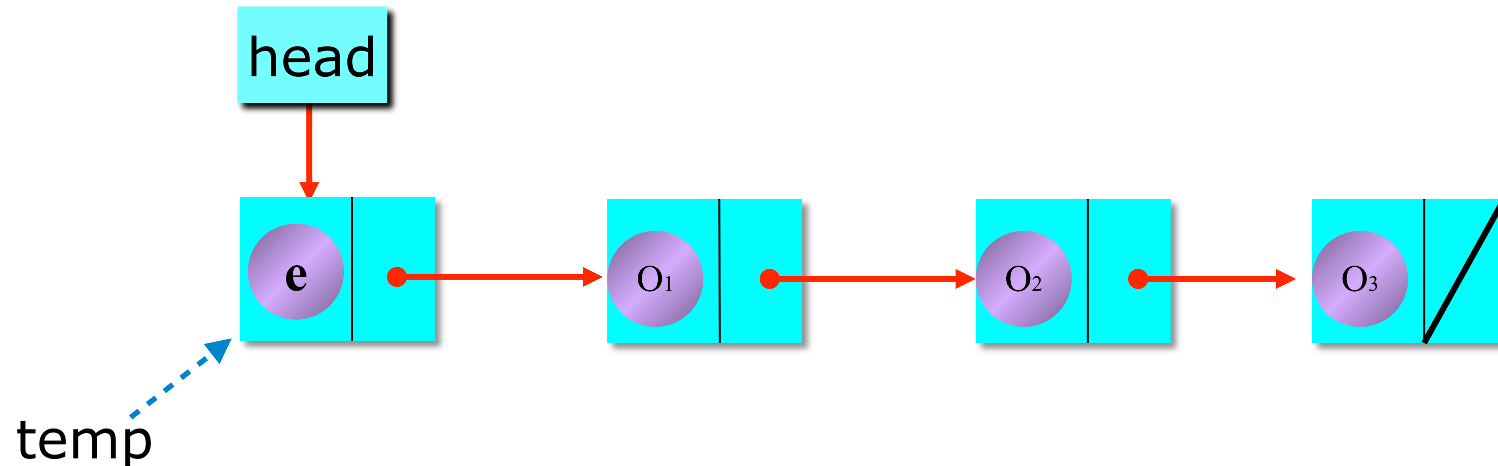

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation



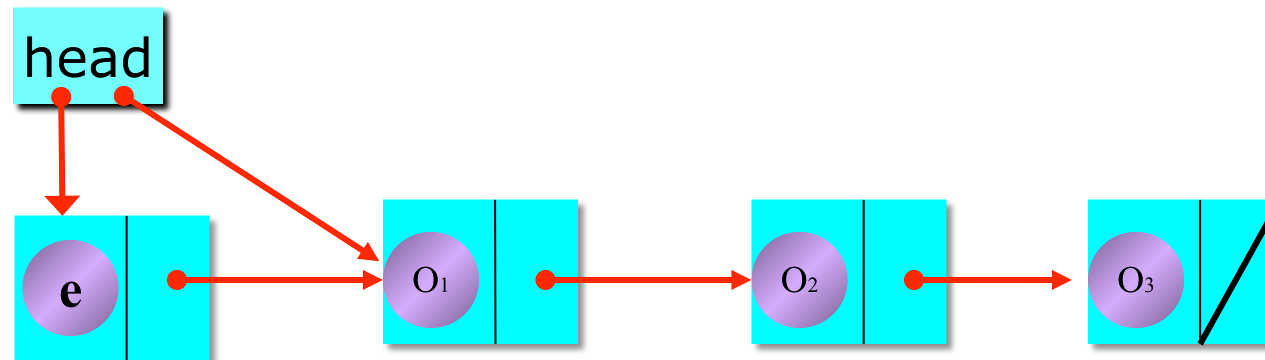

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation




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

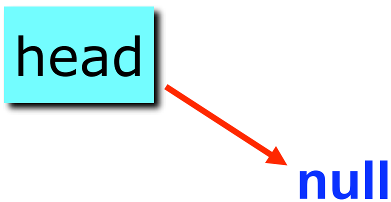
Implementierung der push-Operation



Mit Hilfe eines **ListNode-**Konstruktors, der als erster Parameter das zu speichernde Objekt element bekommt, und als zweiter Parameter eine Referenz auf ein ListNode-Objekt bekommt, können wir sehr einfach unsere **push**-Operation wie folgt implementieren.

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

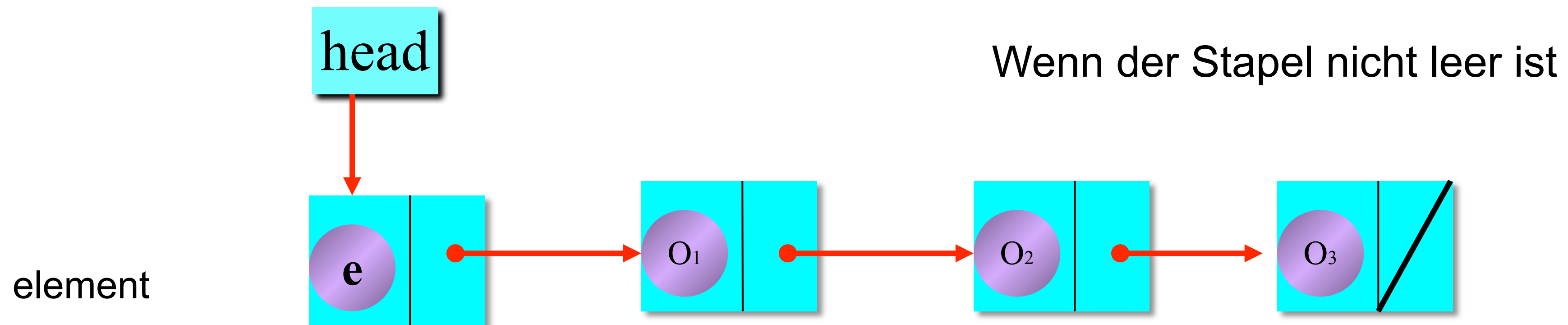
Implementierung der pop-Operation



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

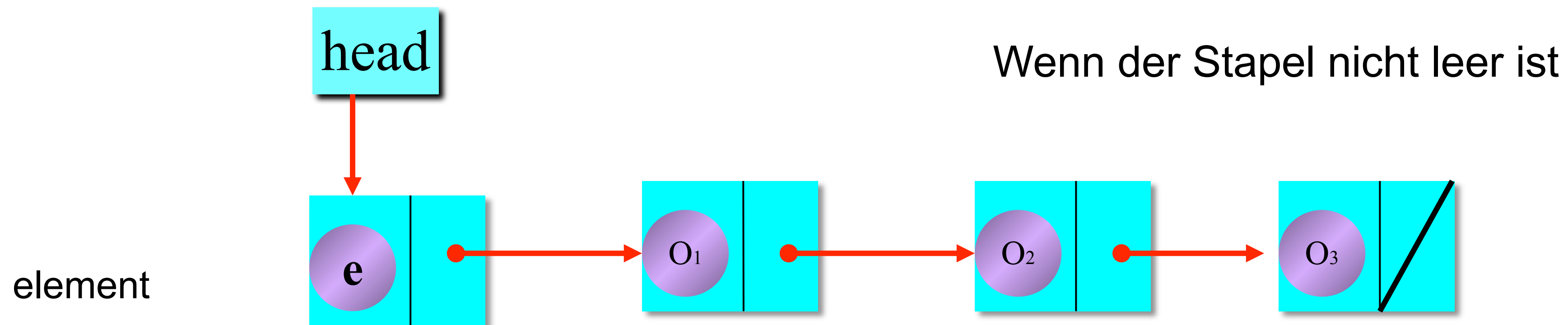
```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Implementierung der pop-Operation

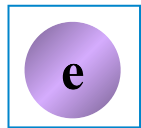



```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Implementierung der pop-Operation



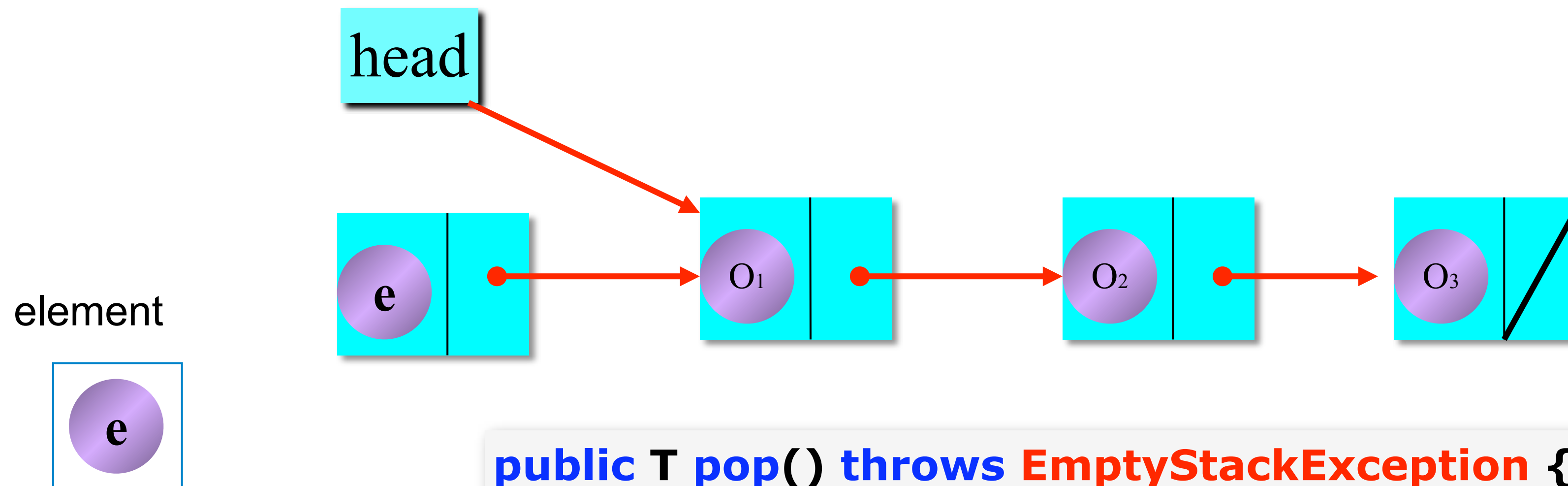
element



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

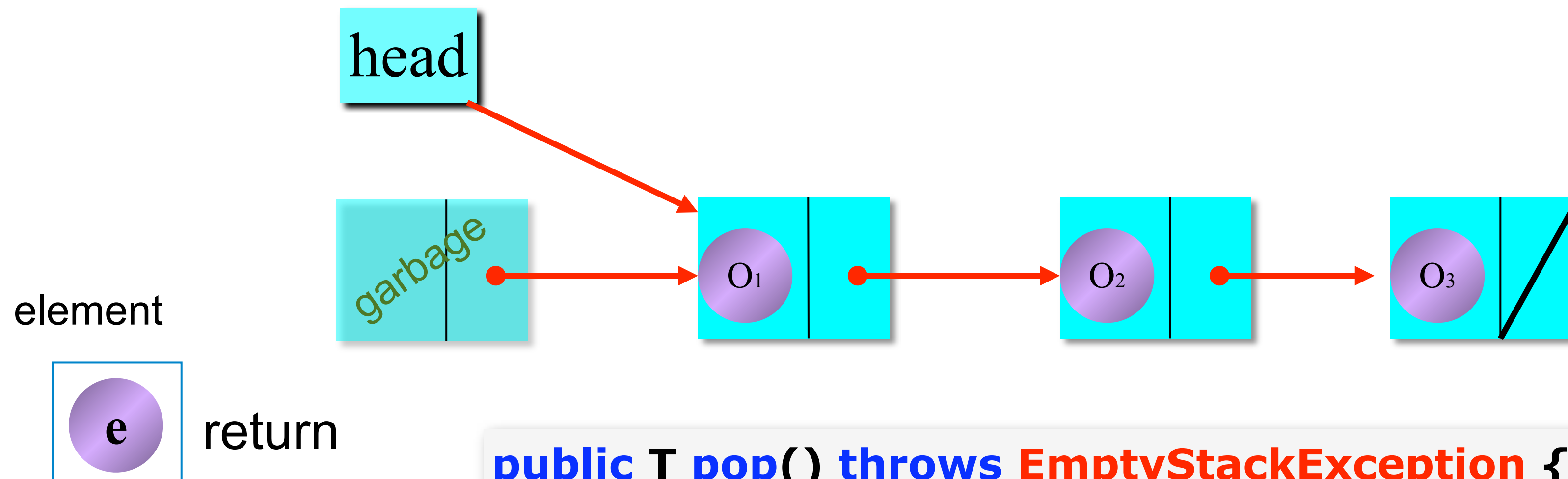
```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Implementierung der pop-Operation



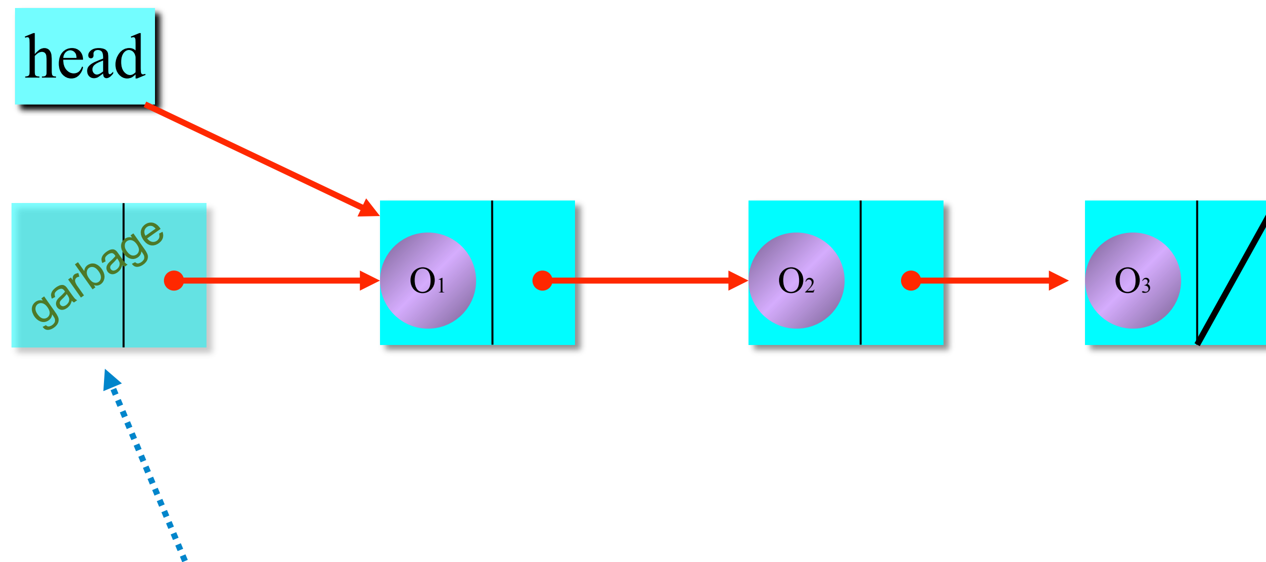

```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

Implementierung der pop-Operation



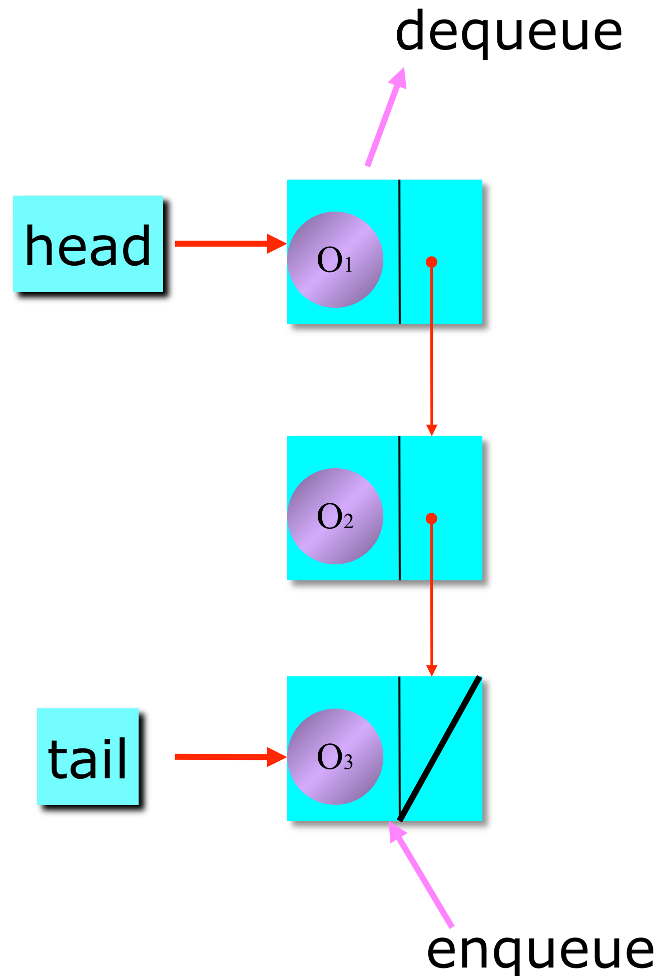

```
public T pop() throws EmptyStackException {
    if ( empty() )
        throw new EmptyStackException();
    T element = head.element;
    head = head.next;
    return element;
}
```

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



FIFO - Datenstruktur

"**F**irst **I**n - **F**irst **O**ut"

Wenn wir eine Warteschlange mit Hilfe einer verketteten Liste implementieren, brauchen wir nicht mehr zu überprüfen, ob die Warteschlange voll ist.

Operationen der Warteschlange

enqueue
dequeue
empty
head

Warteschlange-Schnittstelle

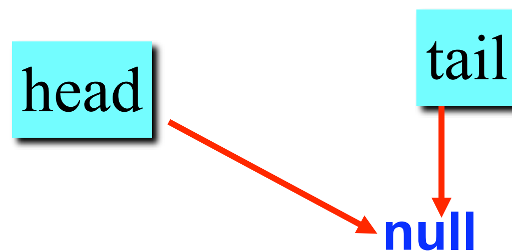
In dieser Implementierung werden wir eine EmptyQueueException erzeugen bei dem Versuch, ein Element zu entfernen oder zu lesen (dequeue-Operation und head-Operationen), wenn die Liste leer ist.

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

Warteschlange-Implementierung

```
public class ListQueue<T> implements Queue<T> {

        ListNode<T> head;
        ListNode<T> tail;

        public ListQueue() {
                this.head = null;
                this.tail = null;

        . . .
}
```

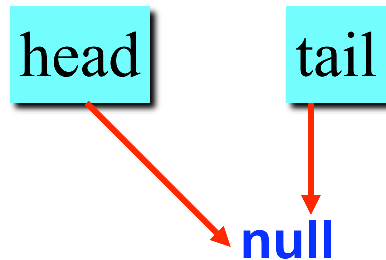
Implementierung der **empty**-Operation



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

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

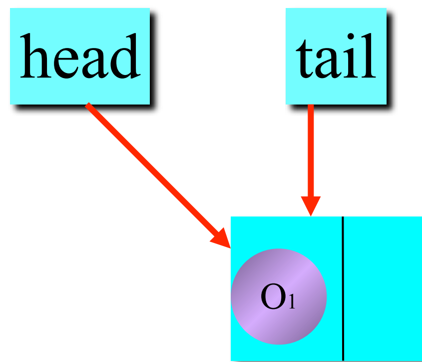
enqueue-Operation



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 );
}
```

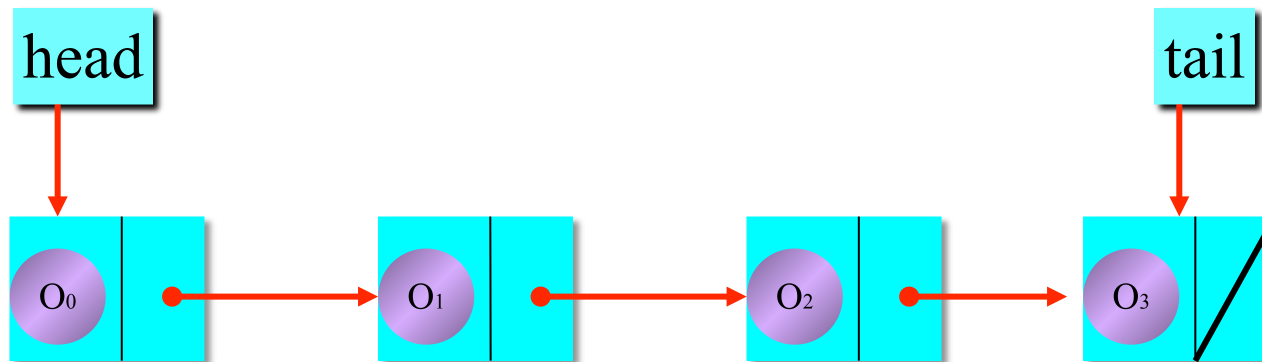
enqueue-Operation



Das zu speichernde Element muss zuerst in ein ListNode-Objekt verpackt werden, und head und tail bekommen eine Referenz aus den neuen Knoten zugewiesen.

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

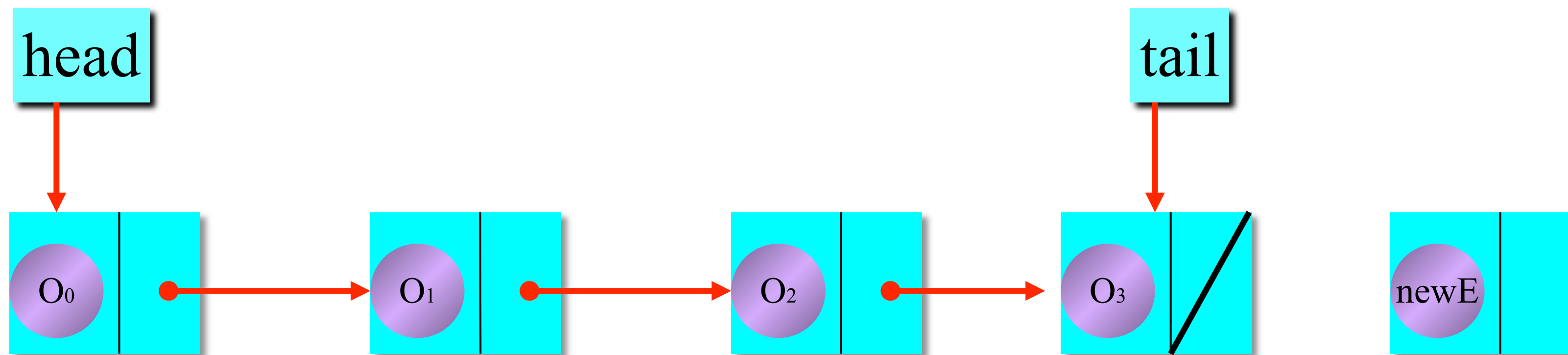
enqueue-Operation



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 );
}
```

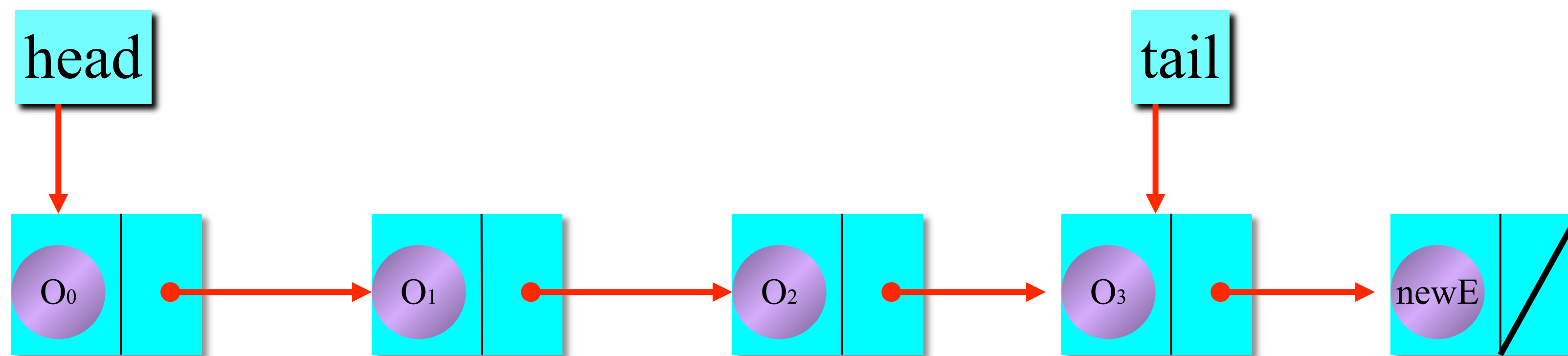
enqueue-Operation



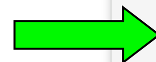
Zuerst wird das zu speichernde neue Objekt in einen neu erzeugten Listenknoten (ListNode) verpackt.

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

enqueue-Operation

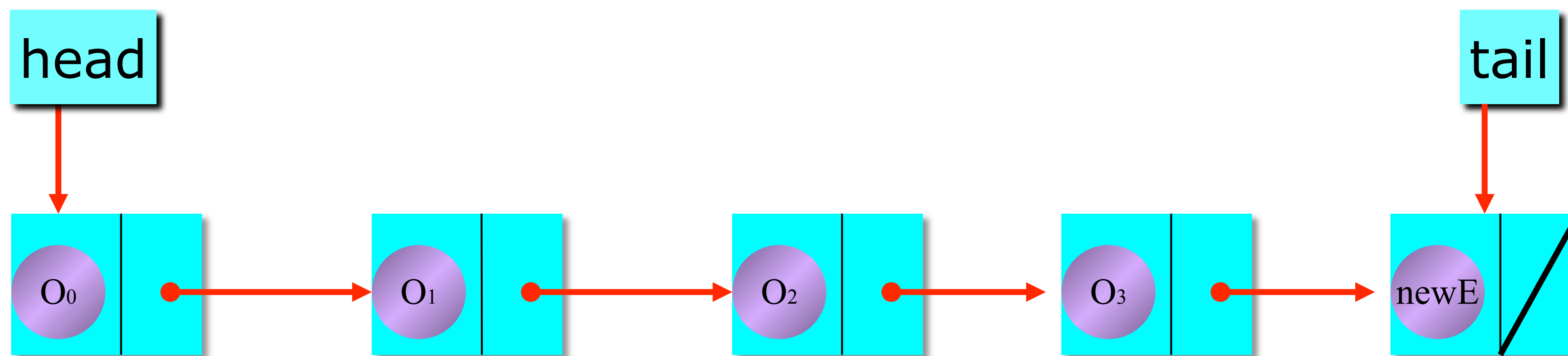


Die Referenz, die von dem ListNode-Konstruktor erzeugt wird, bekommt tail.next zugewiesen.

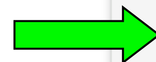


```
public void enqueue ( T newElement ) {
    if ( empty() )
        head = tail = new ListNode<T> ( newElement );
    else
        tail = tail.next = new ListNode<T>( newElement );
}
```

enqueue-Operation

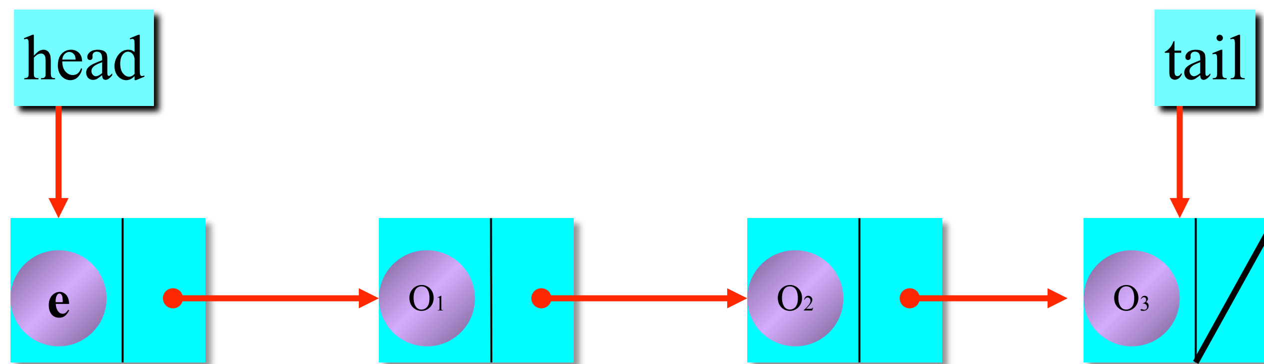


Zum Schluss bekommt tail auch die gleiche Referenz wie tail.next.



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

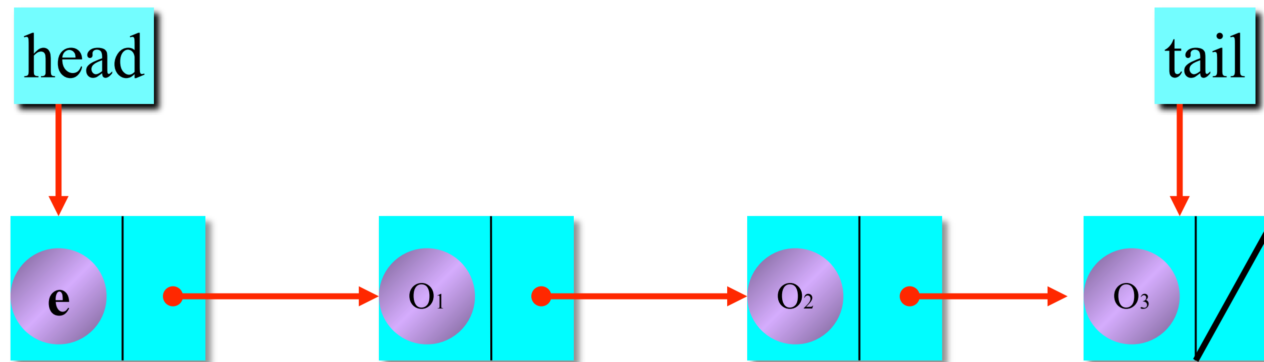
dequeue-Operationen



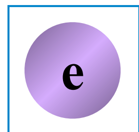
Wenn die Liste nicht leer ist

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

dequeue-Operationen



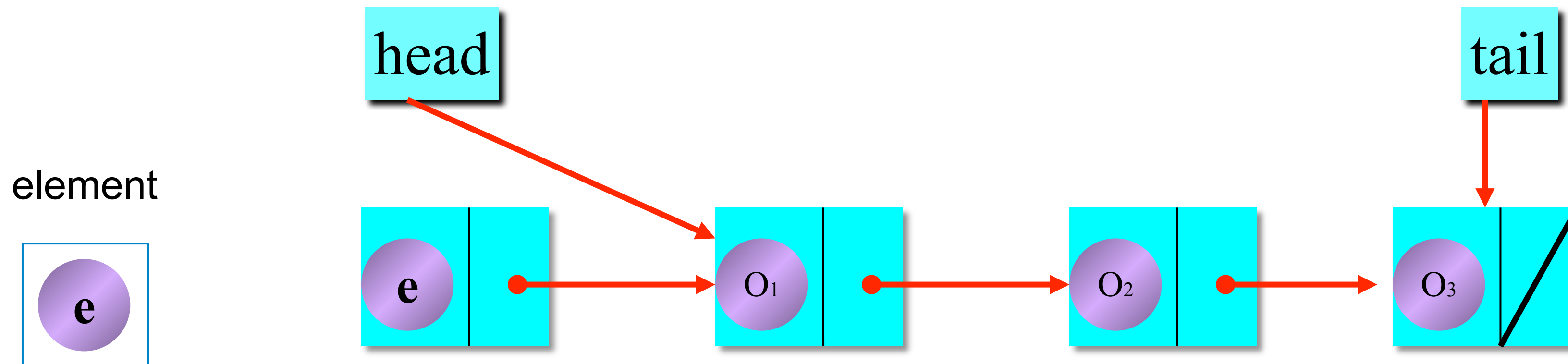
element



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

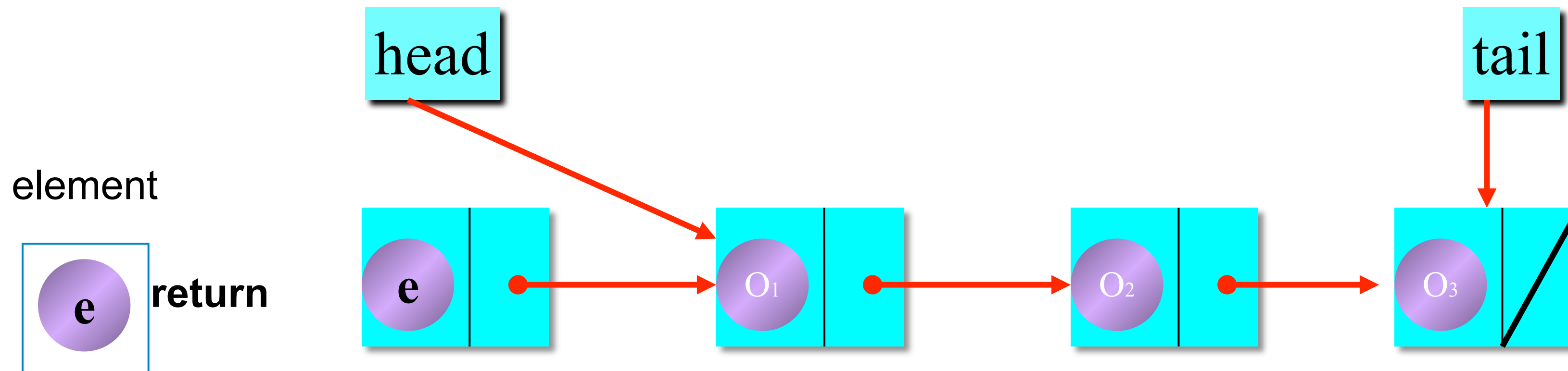
```
public T dequeue() throws EmptyQueueException {
    if (empty () )
      throw new EmptyQueueException();
    T element = head.element;
    head = head.next;
    return element;
}
```

dequeue-Operationen




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

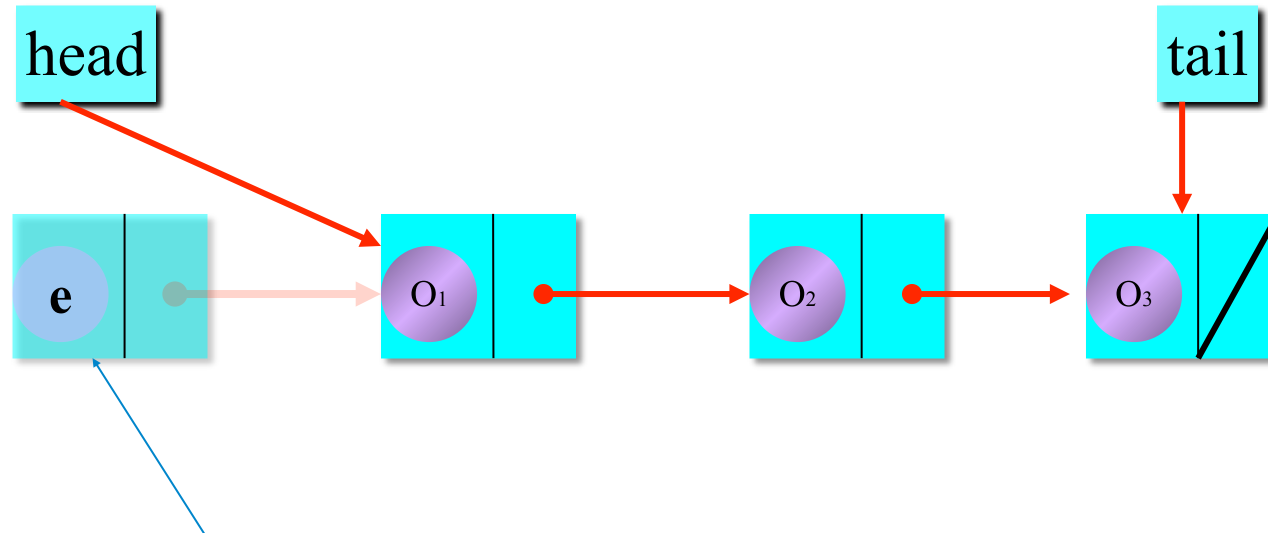
dequeue-Operationen




```
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

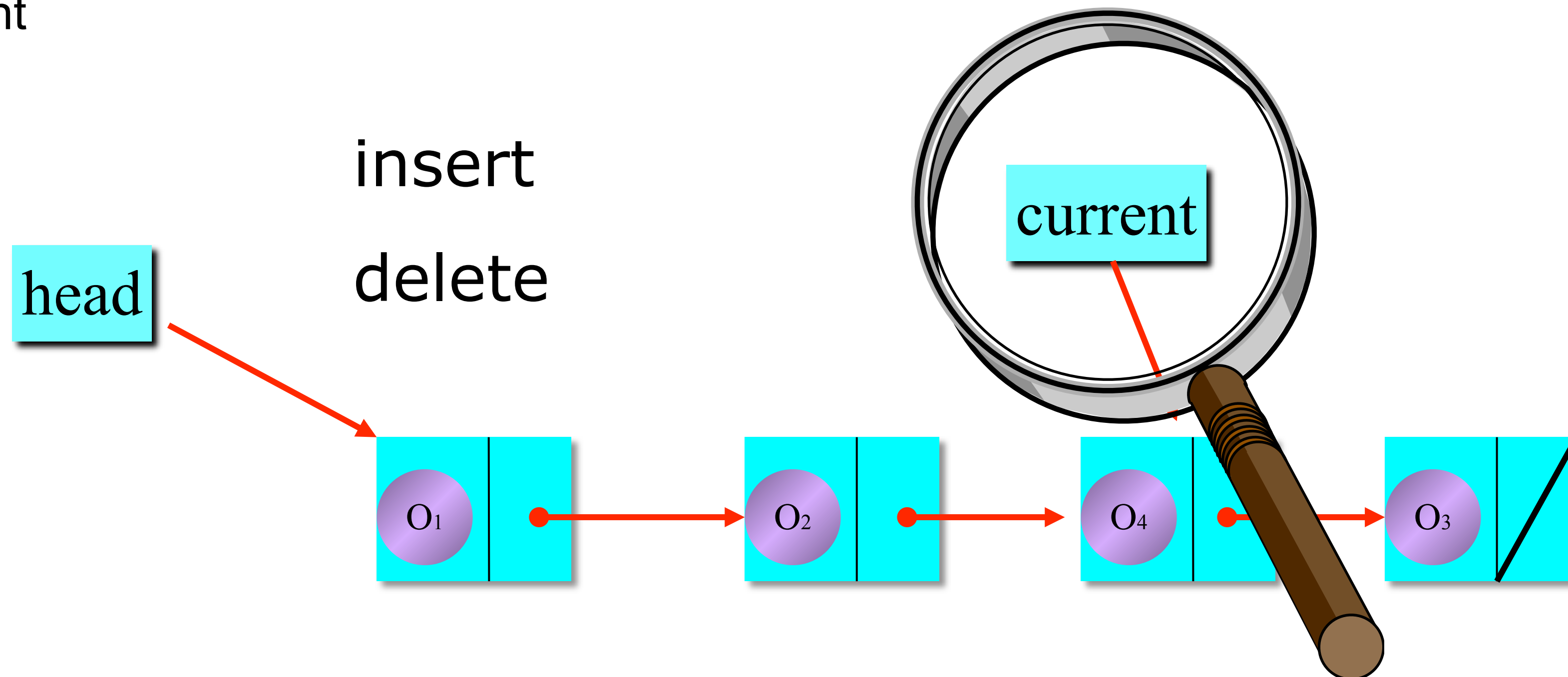


Weil keine Referenz auf das entfernte ListNode-Objekt zeigt, bleibt es Speicherdatenmüll, bis es vom Java-“garbage collector“ gelöscht wird.

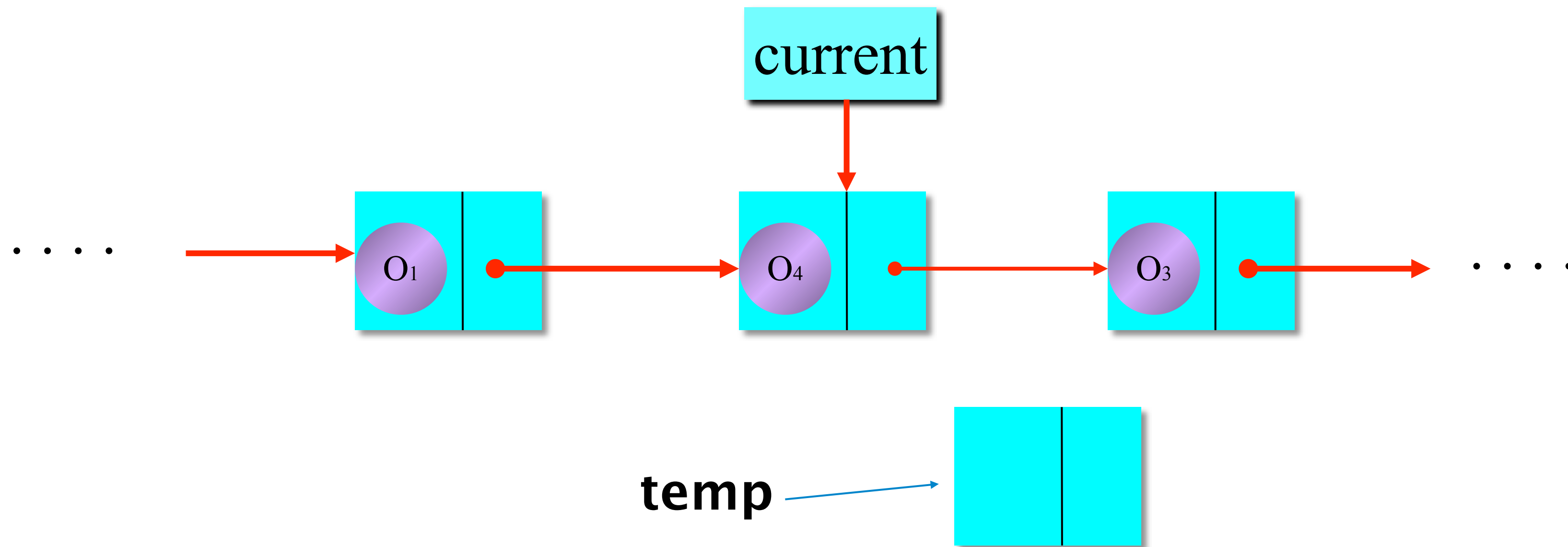
Allgemeine dynamische Datenmenge

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

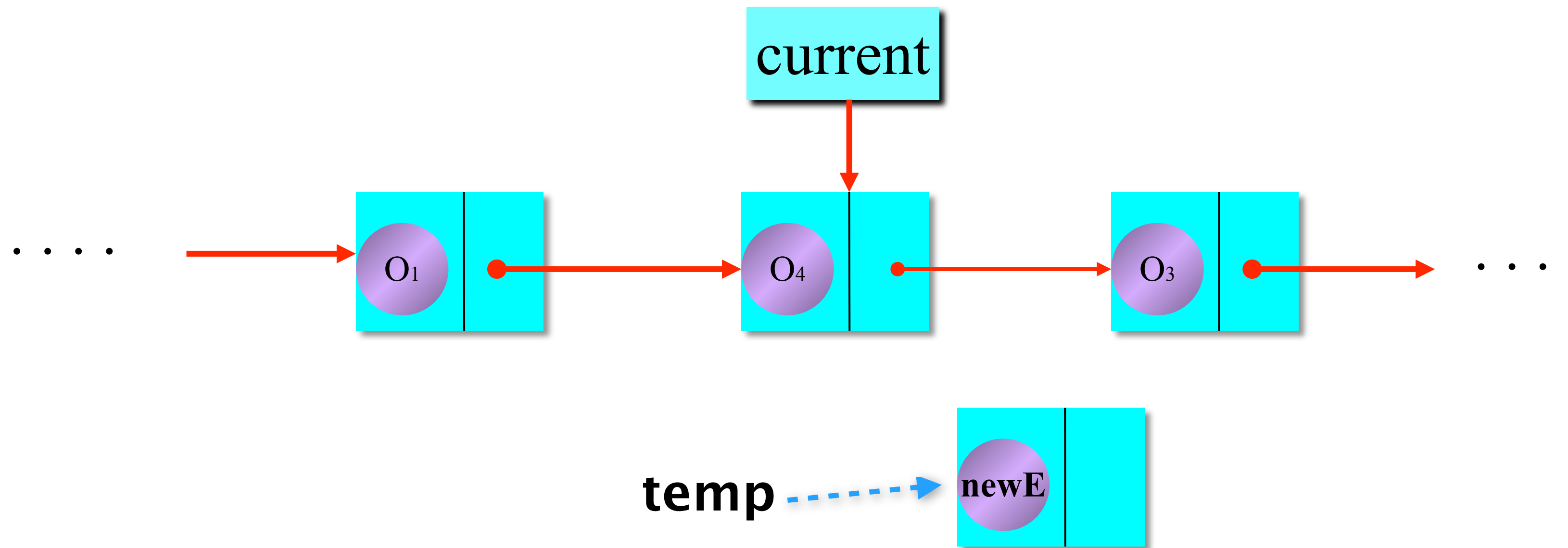


Einfügen an einer beliebigen Stelle der Liste



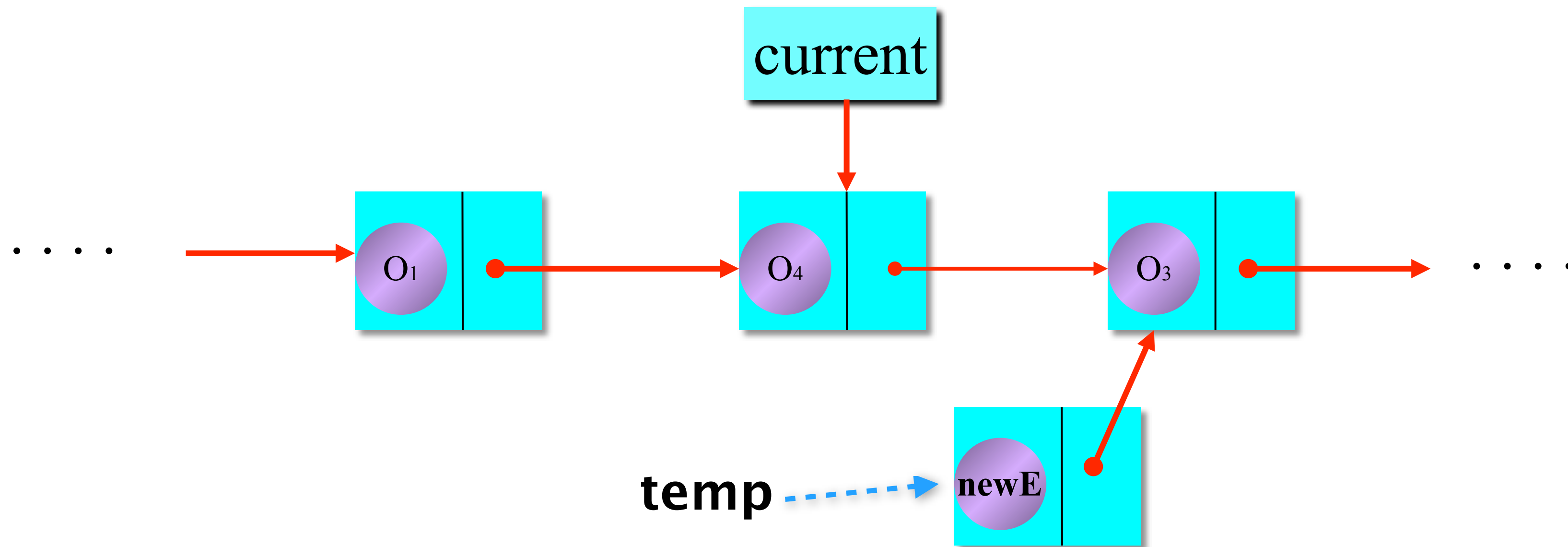

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = newE;
temp.next = current.next;
current.next = temp;
 . . .
```

Einfügen an eine beliebige Stelle der Liste



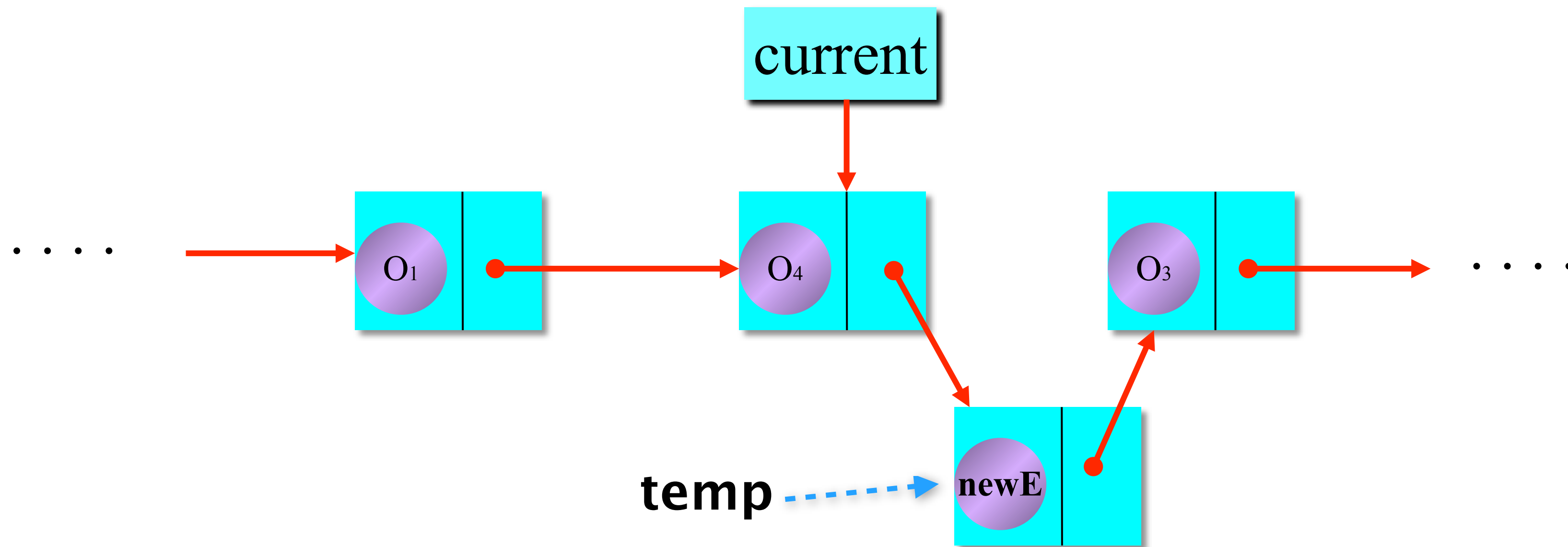

```
...
ListNode<T> temp = new ListNode<T>();
temp.element = newE;
temp.next = current.next;
current.next = temp;
...
```

Einfügen an eine beliebige Stelle der Liste



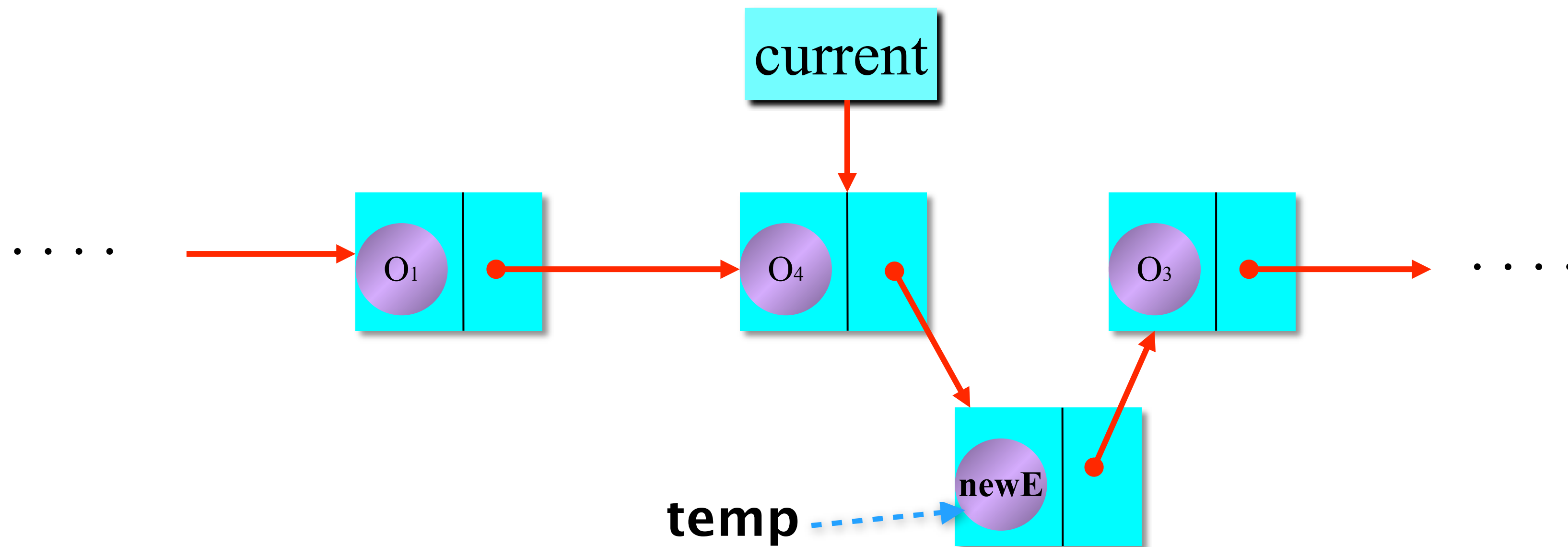

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = newE;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an eine beliebige Stelle der Liste



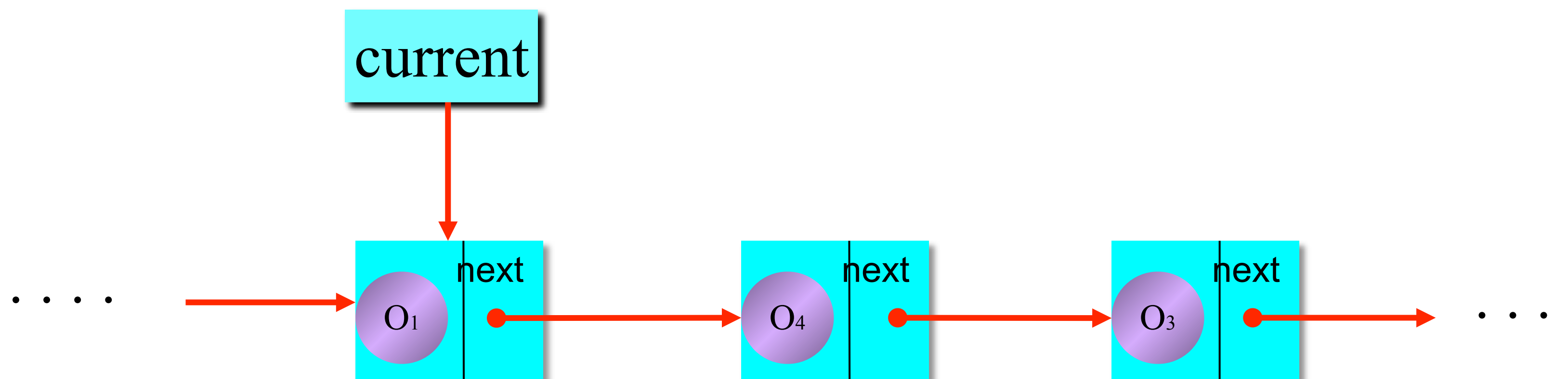

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = newE;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an eine beliebige Stelle der Liste



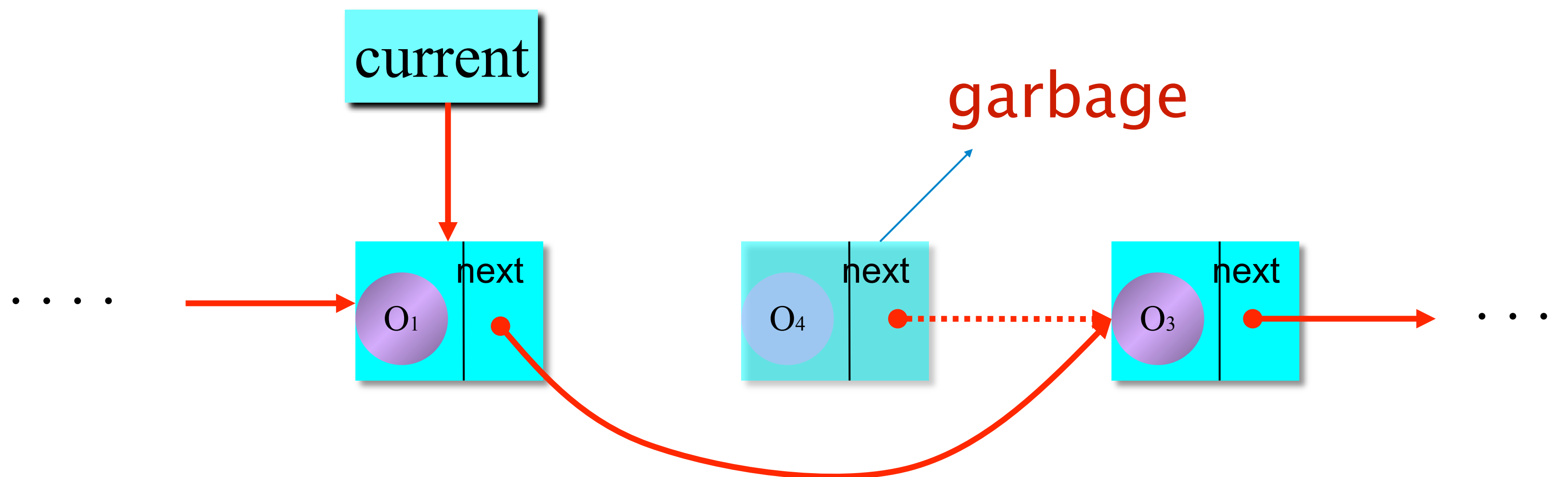

```
...
current.next = new ListNode<T>( newE, current.next );
...
```

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 Löschoperationen, 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

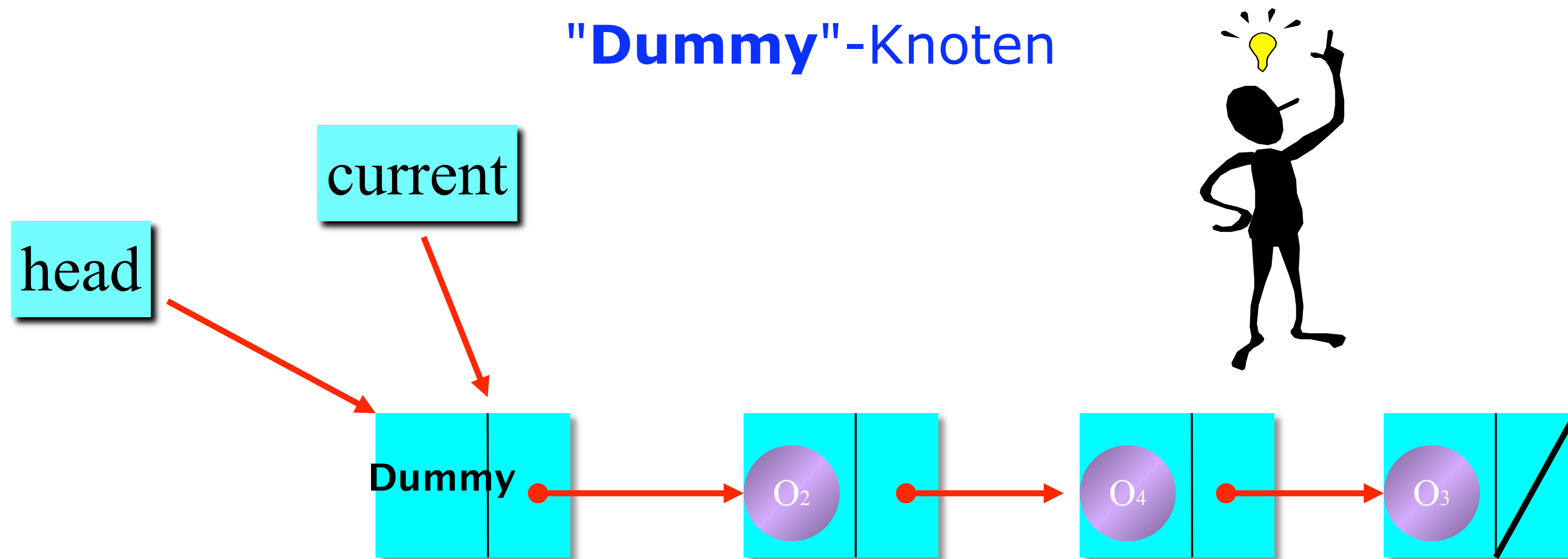
- Löschen des ersten Elements der Liste-

- Einfügen, wenn die Liste leer ist-

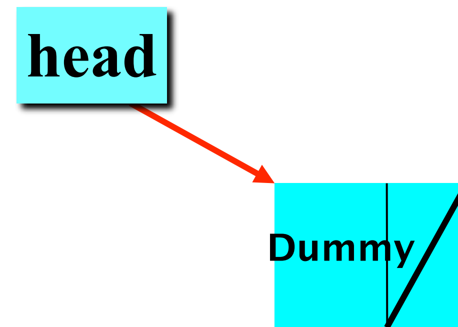
nicht berücksichtigt werden müssen.

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.

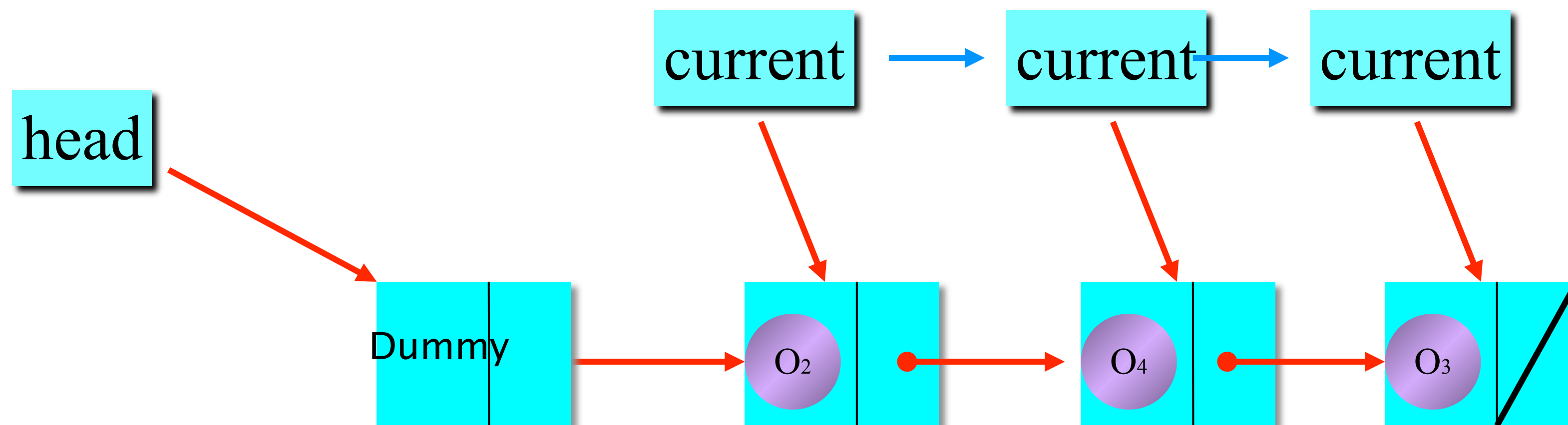



```
public boolean  empty () {

      return head.next == null;

}
```

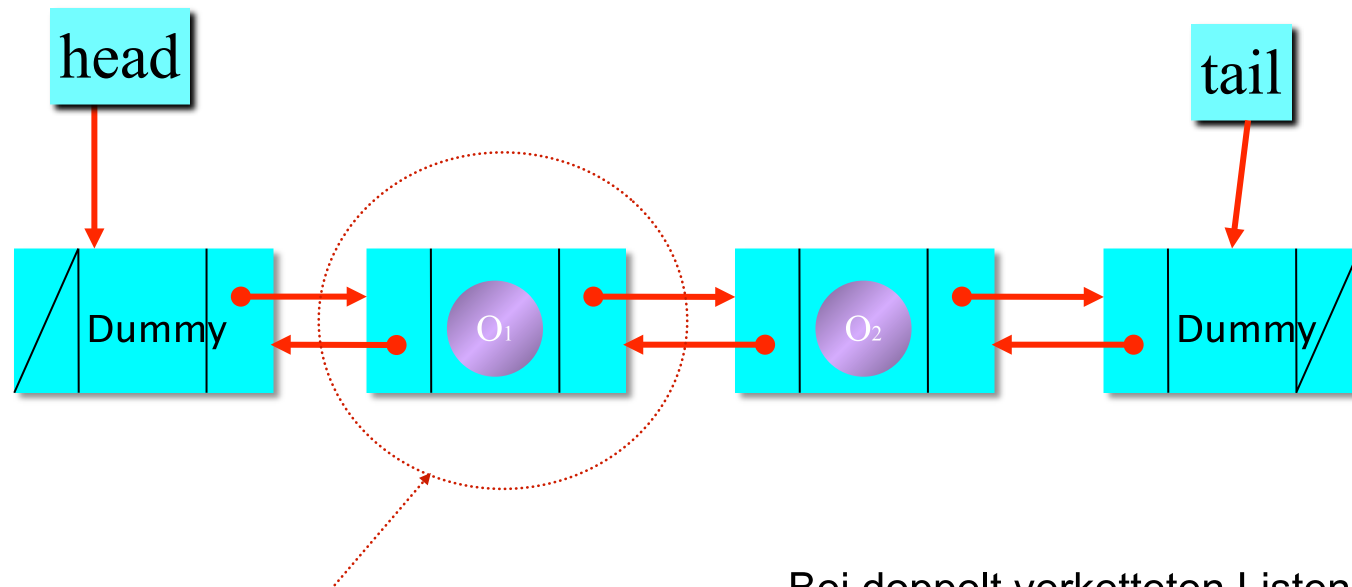
Einschränkungen bei einfach verketteten Listen

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.



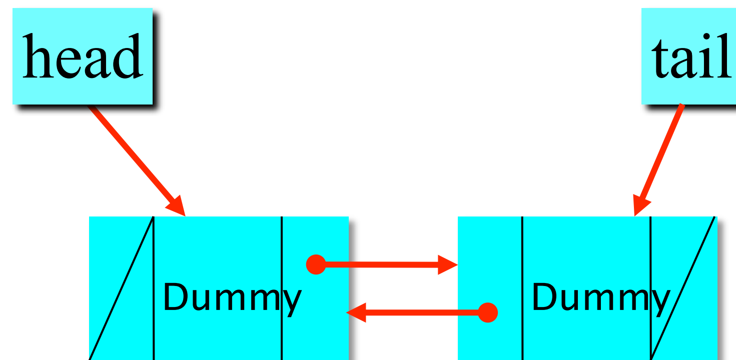
Doppelt verkettete Listen




```
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 Lösch-Operation ändert sich, weil wir jetzt immer eine doppelte Verkettung erstellen müssen.

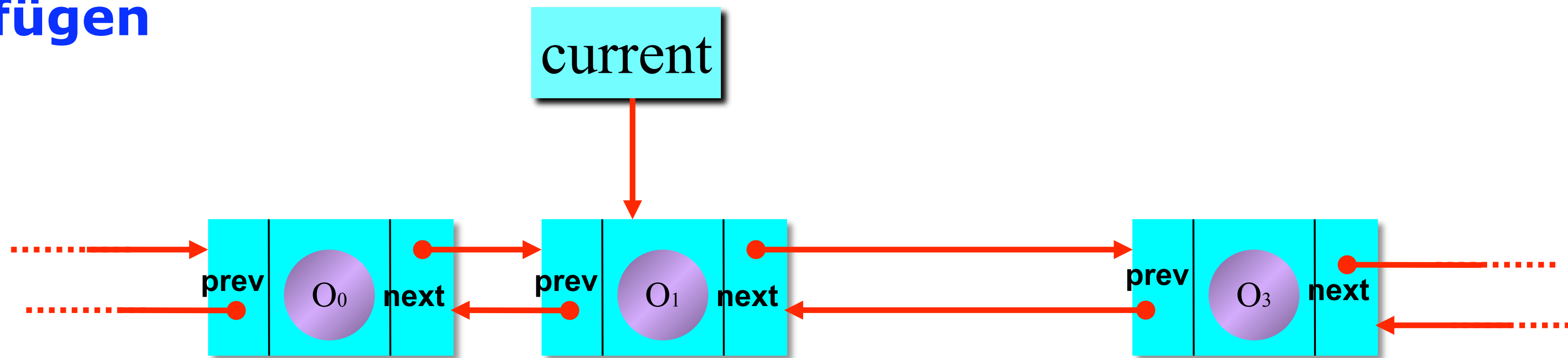
Doppelt verkettete leere Liste

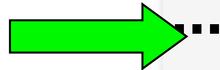


Bei doppelt verketteten Listen ist es sinnvoll, einen Zeiger auf das letzte Element zu haben.

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

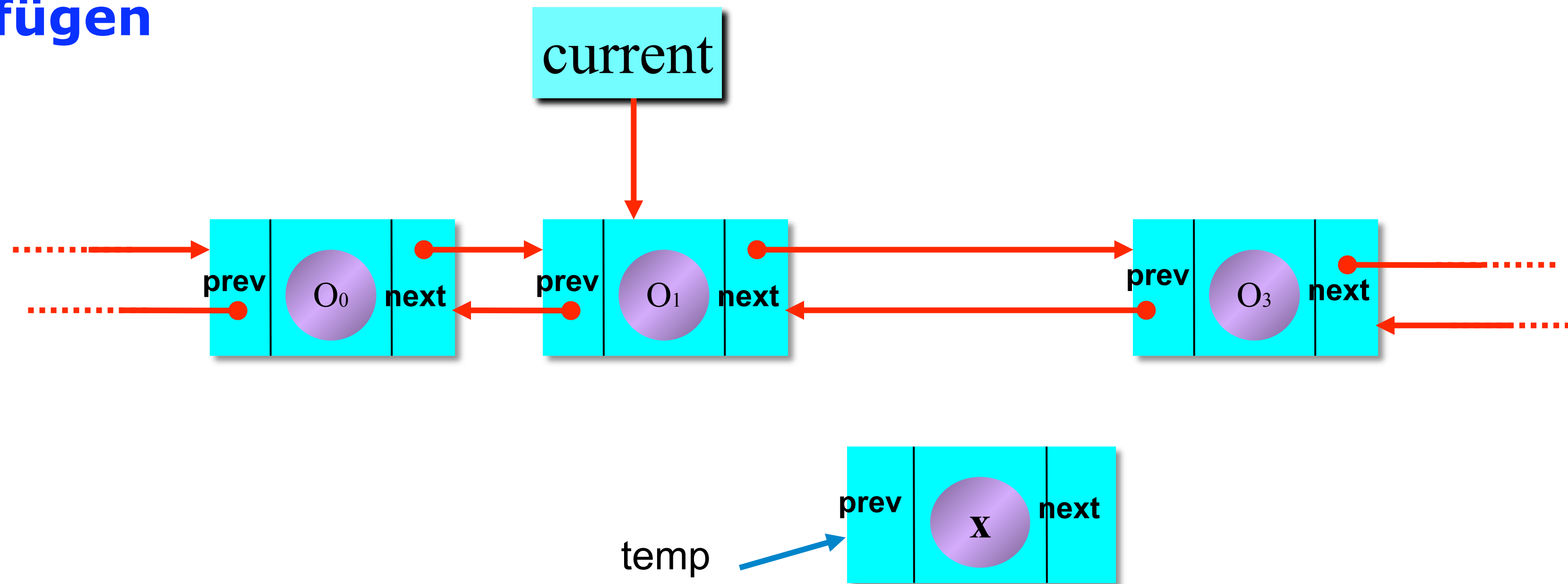
Einfügen





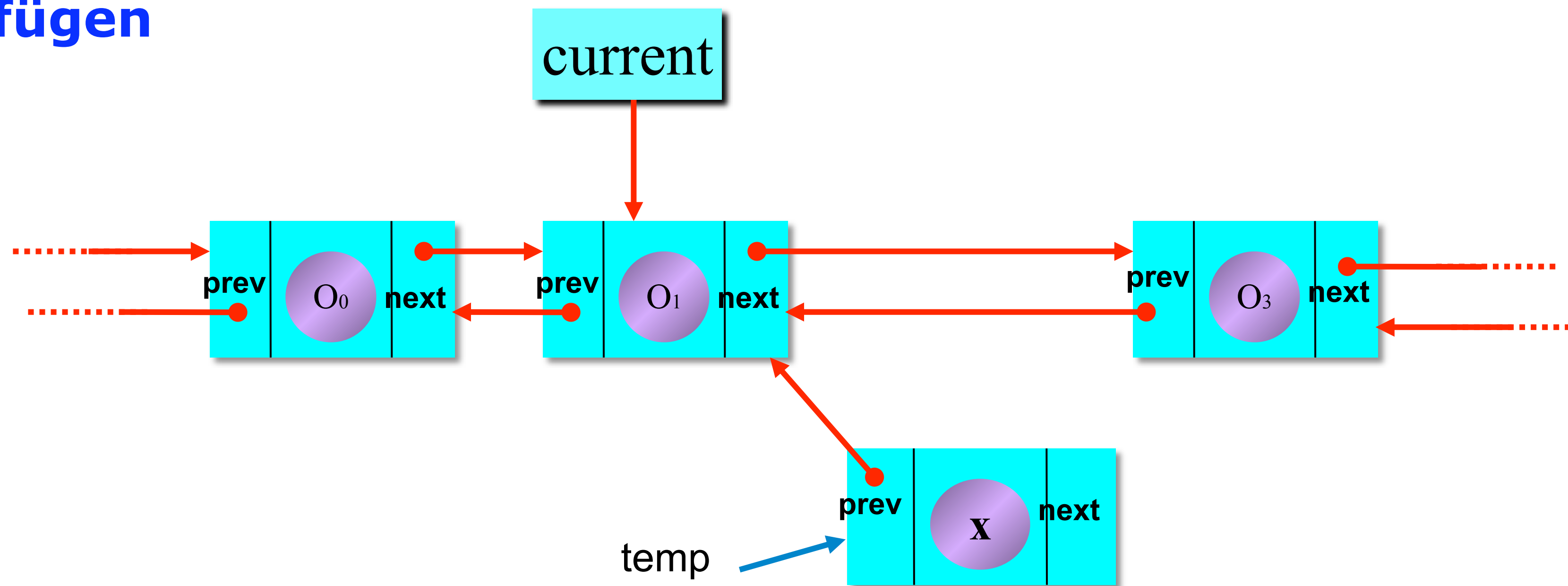
```
...
    ListNode<T> temp = new ListNode<T>( x );
    temp.prev = current;
    temp.next = current.next;
    temp.prev.next = temp;
    temp.next.prev = temp;
    current = temp;
...
```

Einfügen



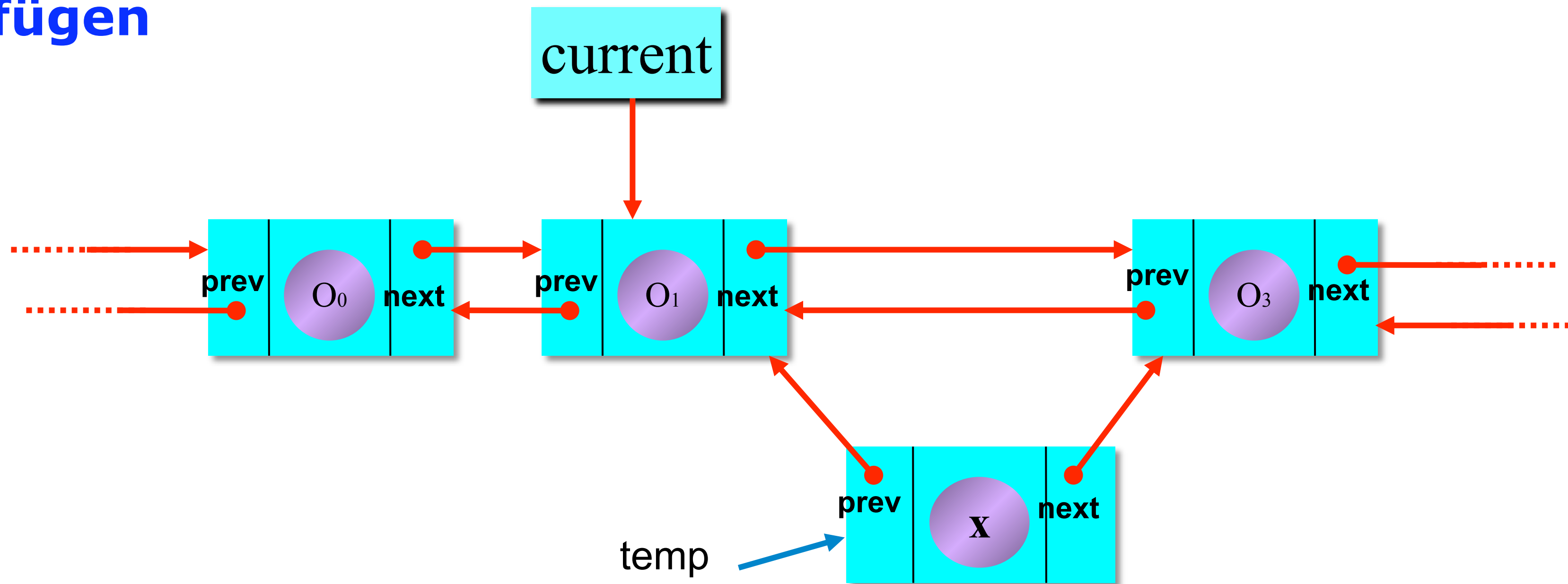

```
...
ListNode<T> temp = new ListNode<T>( x );
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;
...
```

Einfügen



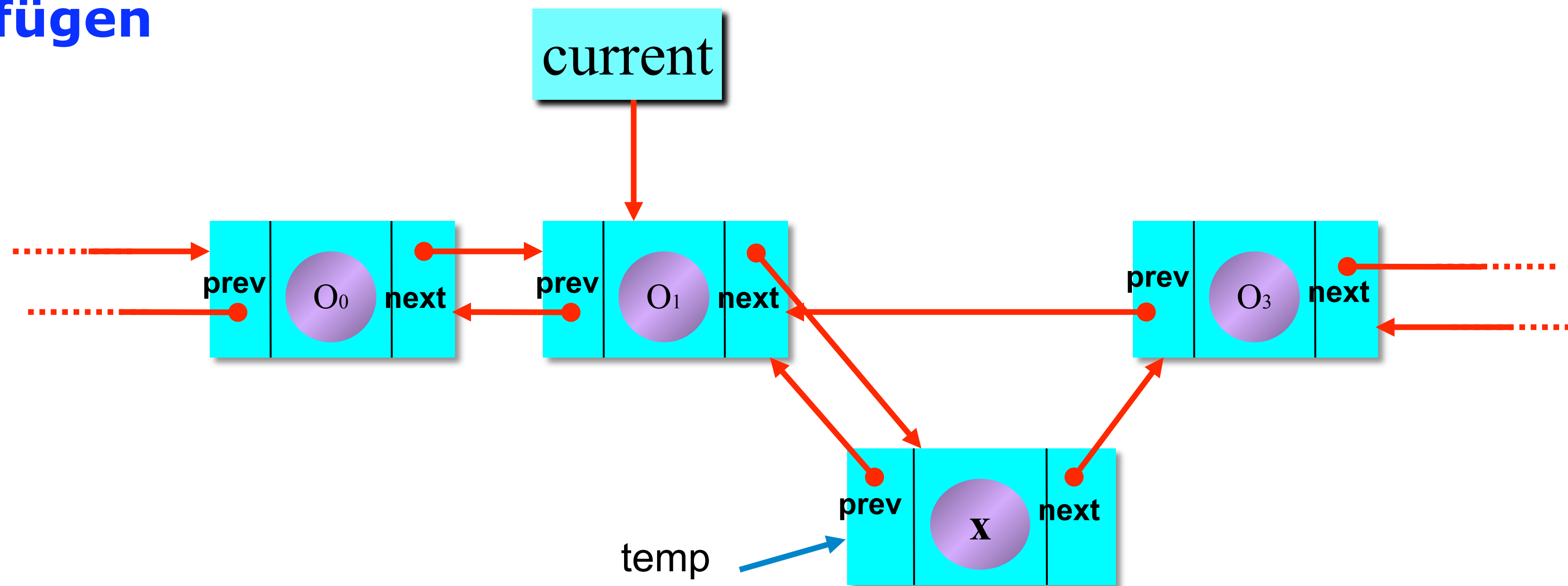

```
...
    ListNode<T> temp = new ListNode<T>( x );
    temp.prev = current;
    temp.next = current.next;
    temp.prev.next = temp;
    temp.next.prev = temp;
    current = temp;
...
```

Einfügen



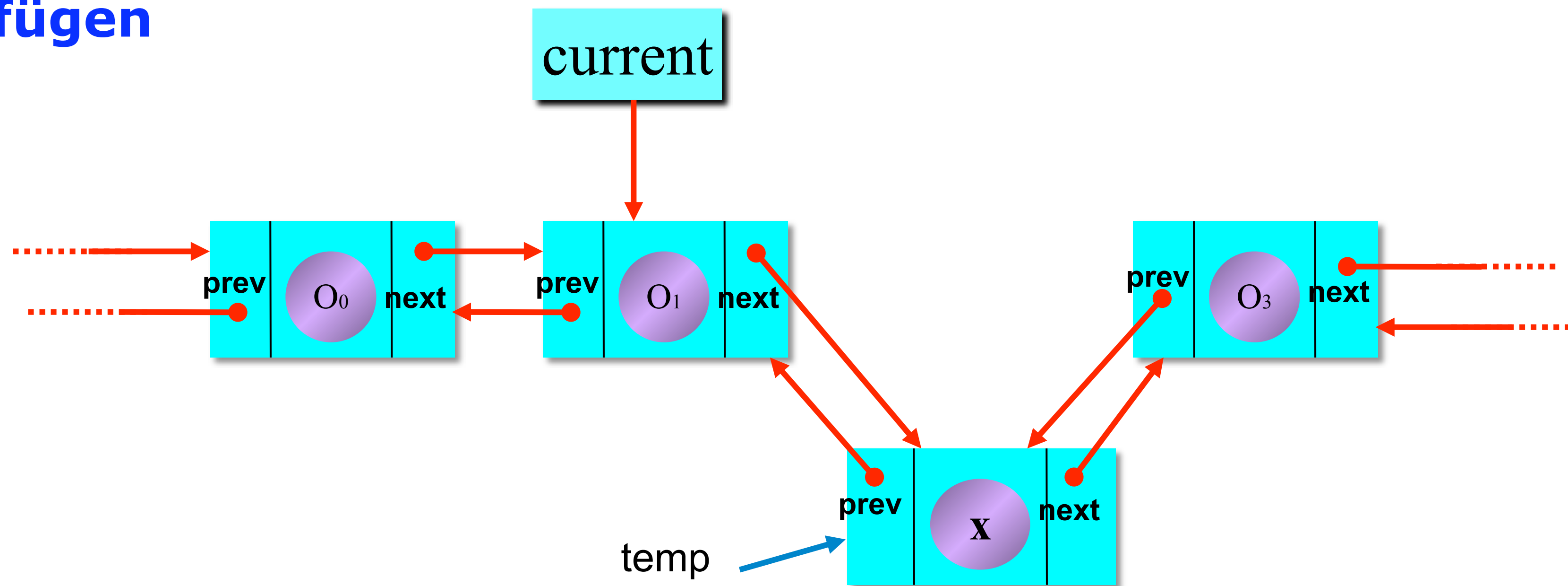

```
...
    ListNode<T> temp = new ListNode<T>( x );
    temp.prev = current;
    temp.next = current.next;
    temp.prev.next = temp;
    temp.next.prev = temp;
    current = temp;
...
```

Einfügen



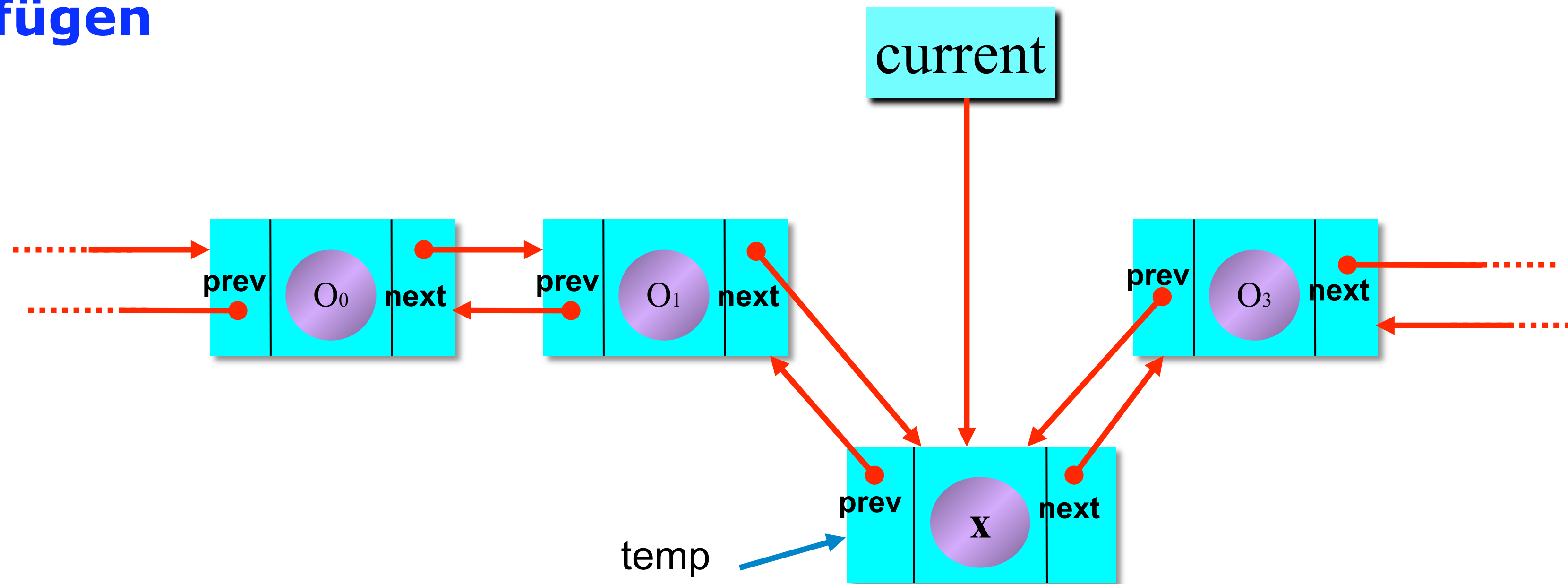

```
...
    ListNode<T> temp = new ListNode<T>( x );
    temp.prev = current;
    temp.next = current.next;
    temp.prev.next = temp;
    temp.next.prev = temp;
    current = temp;
...
```

Einfügen



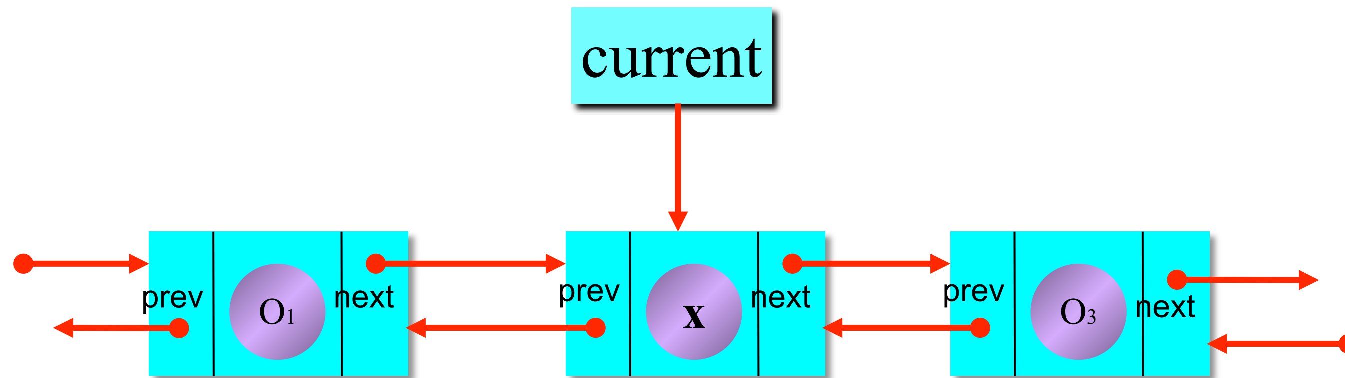

```
...
    ListNode<T> temp = new ListNode<T>( x );
    temp.prev = current;
    temp.next = current.next;
    temp.prev.next = temp;
→   temp.next.prev = temp;
    current = temp;
...
```

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;
...
```

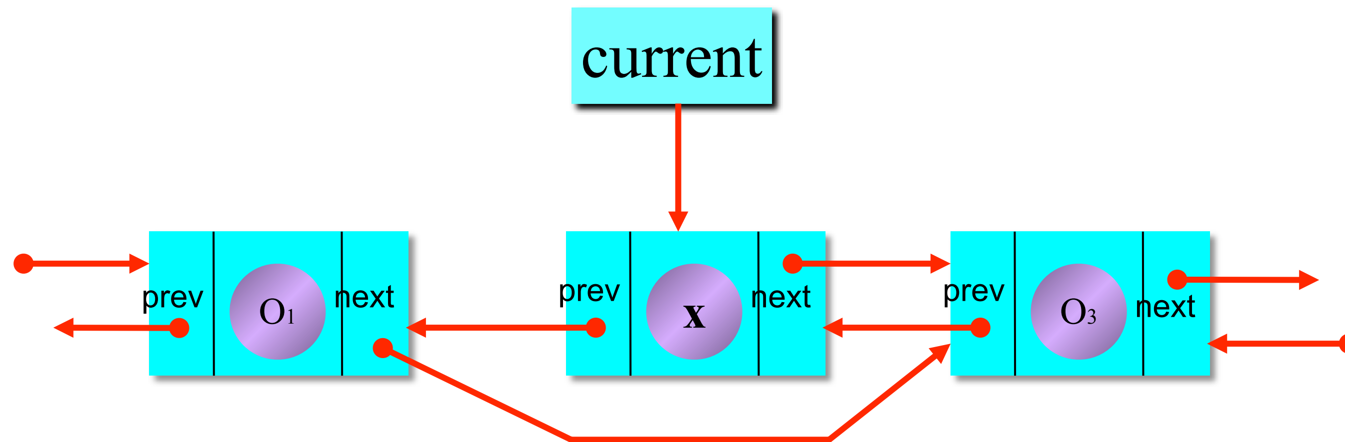
Löschen



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

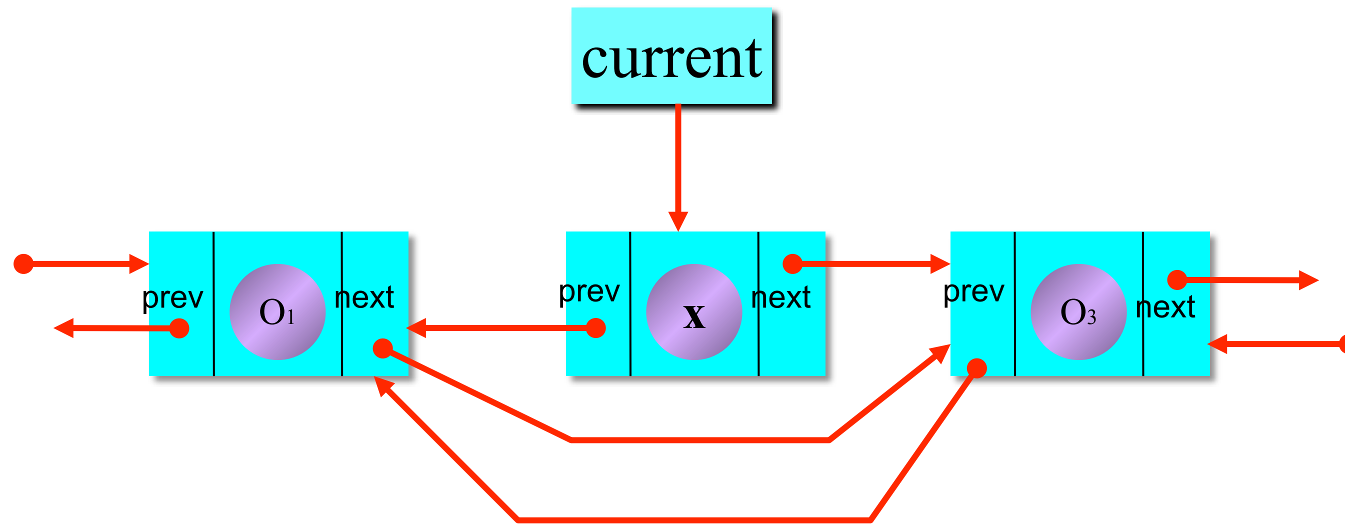
```
. . .
  current.prev.next = current.next;
  current.next.prev = current.prev;
  current = head;
. . .
```

Löschen



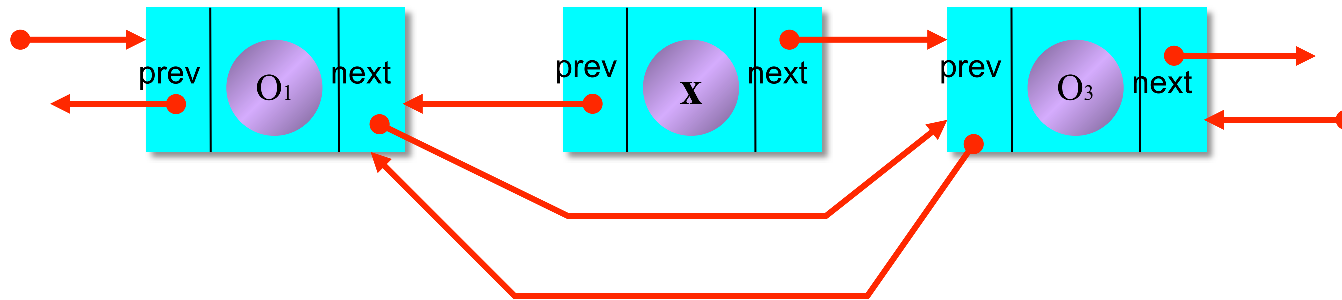

```
. . .
current.prev.next = current.next;
current.next.prev = current.prev;
current = head;
. . .
```

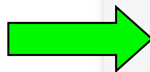
Löschen




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

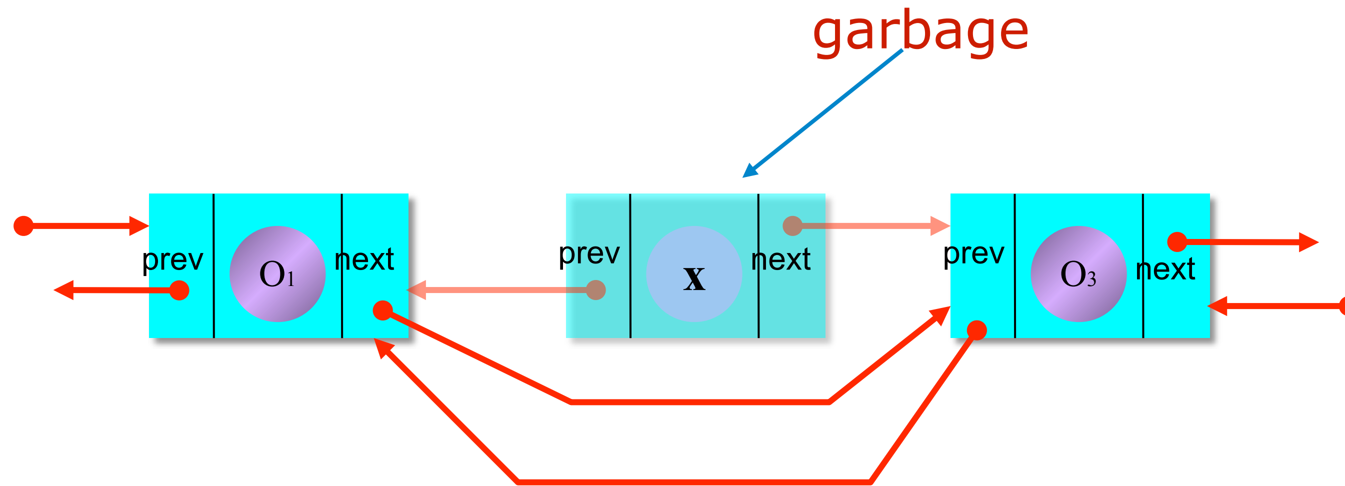
Löschen

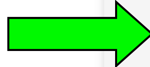




```
. . .
  current.prev.next = current.next;
  current.next.prev = current.prev;
  current = head;
. . .
```

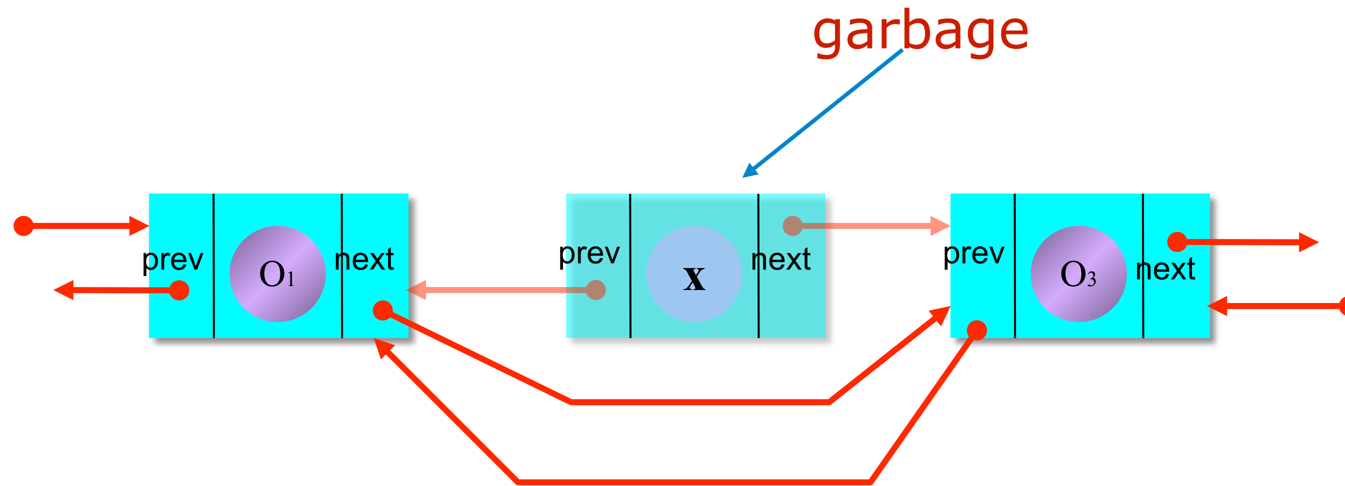
Löschen

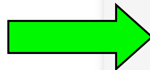




```
. . .
  current.prev.next = current.next;
  current.next.prev = current.prev;
  current = head;
. . .
```

Löschen





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