Barry Linnert
Institut für Informatik

Freie Universität Berlin

Programmieren

Barry Linnert

Sommersemester 2020

Gliederung der heutigen Vorlesung

- Kurze Wiederholung
- Container
- Listen
- Einfach verkette Listen
  - Implementierung des Stack durch eine verketteten Liste
  - Implementierung der Queue durch eine verketteten Liste
  - Einfügen und Löschen an beliebigen Stellen der verketteten Liste
- Doppelt verkettete Listen
- Zusammenfassung

Freie Universität Berlin

Dynamische Datenstrukturen

WIEDERHOLUNG

Freie Universität Berlin

Gliederung der heutigen Vorlesung

- Kurze Wiederholung
- Lists
  - Eigene Implementierung in Java
- Zusammenfassung

Terminologie

- Ein **abstrakter Datentyp (ADT)** besteht aus einem Wertebereich (d.h. einer Menge von Objekten) und darauf definierten Operationen.

- Die Menge der Operationen bezeichnet man auch als Schnitt-stelle des Datentyps.

- Eine **Datenstruktur** ist eine Realisierung bzw. Implementierung eines ADT.

Verbergen der Implementierung eines abstrakten Datentyps

Meyer, B., "Object-Oriented Software Construction", 2nd ed., Prentice Hall, 1997

Elementare Datenstrukturen – Stack und Queue

Elementare Datenstrukturen

- Stack
- Queue

Freie Universität Berlin

Dynamische Datenstrukturen

CONTAINER

Container?

- Containern sind Arrays, Mengen, Listen, HashSets, Maps, Heaps, Bäume, Streams, etc.
- Alle diese Container haben besondere Eigenschaften, die sie für die ein oder die andere Anwendung besonders geeignet erscheinen lassen
- Was ist ihre Gemeinsamkeit?
  - In allen Behältern können Elemente gleichen Typs gespeichert sein, und diese Elemente können systematisch traversiert werden.

Diese Funktionalität abstrahiert das
generische Interface `Iterable<E>` im Paket `java.util`.

Iterator in Java

- Mit der Methode `iterator()` wird ein Objekt Iterator erzeugt.
- Ein Iterator ist ein Traversierer, bildlich gesprochen so etwas wie ein guide, der alle Elemente des “Behälters” der Reihe nach präsentieren kann.
- Ein Iterator versteht die folgenden Methoden:
  - `boolean hasNext`: prüft, ob noch ein weiteres Element ansteht
  - `E next`: liefert dieses Element und schreitet zum nächsten,
  - `void remove`: entfernt das aktuelle Element.

Iterator in Java

- Gegeben ist Behälter b als ein Objekt einer Iterable<E> implementierenden Klasse
- Mit dem Aufruf b.iterator() wird ein Iterator erzeugt
- Dieser Iterator erlaubt es alle Elemente zu besuchen:

```
Iterator guide = b.iterator();
while(guide.hasNext()){ tuWas(next()); }
```

- Äquivalenter Aufruf

```
Iterator guide = b.iterator();
for(E e: b) tuWas(e)
```

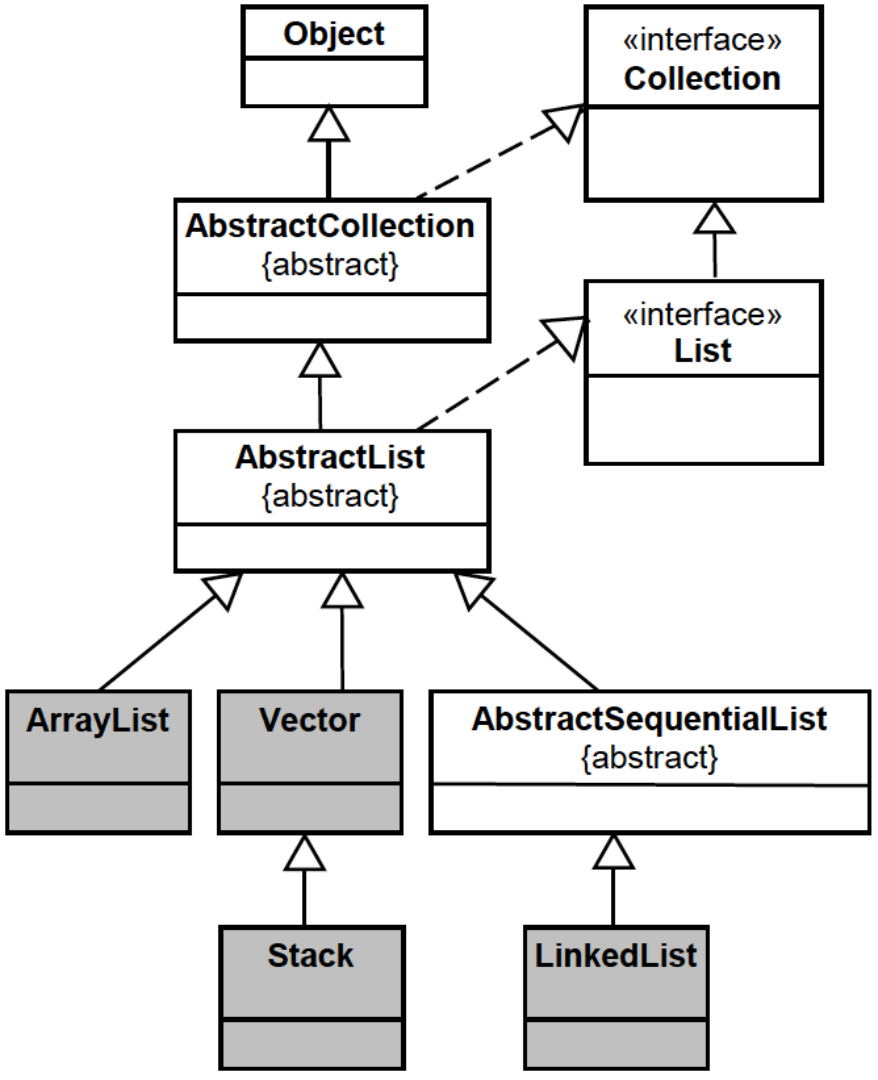
Freie Universität Berlin

Dynamische Datenstrukturen

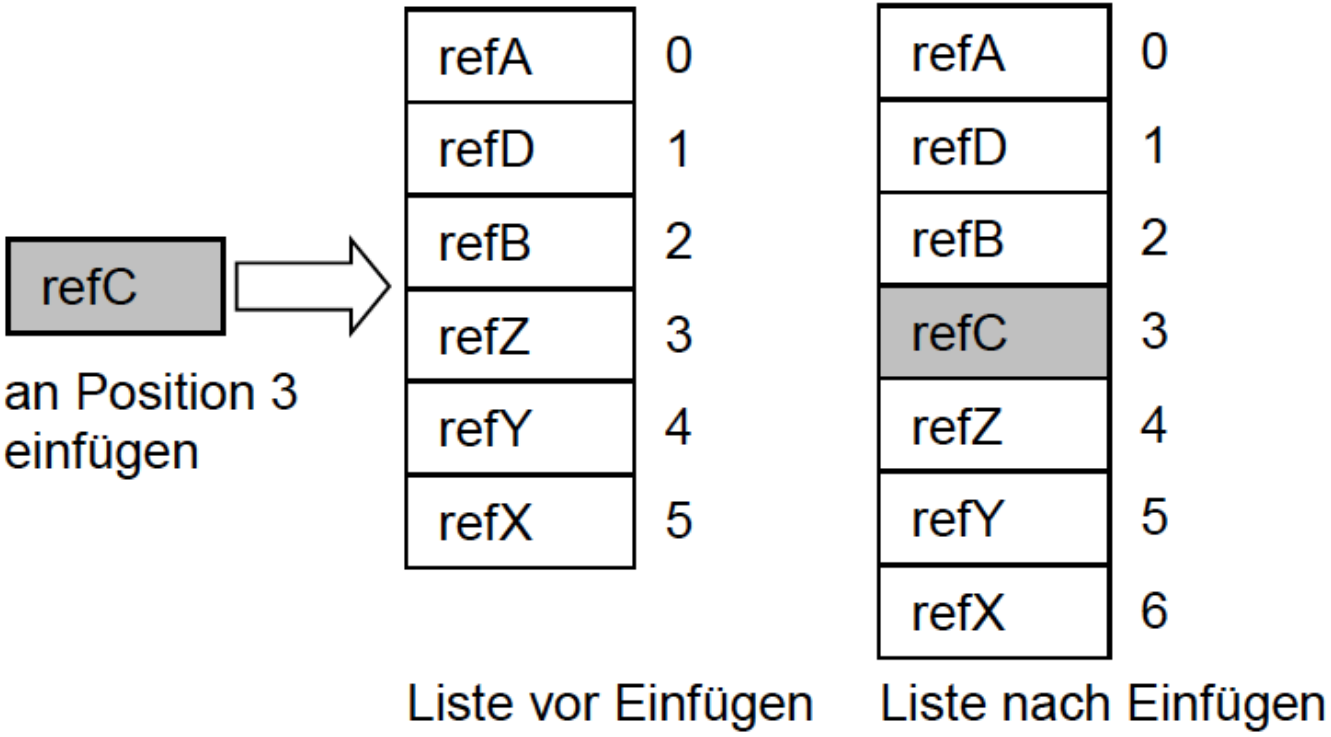
LISTEN

Listen

- Als Listen werden die Klassen bezeichnet, welche die Schnittstelle List<E> implementieren und das sind die Klassen
  - ArrayList<E>
  - Vector<E>
  - LinkedList<E> und
  - Stack<E>.



Zugriff auf Listenelemente



In Listen können Objektreferenzen an beliebiger Stelle eingefügt werden.

Nutzung von Listen in imperativen Sprachen

- In “klassischen” imperativen Sprachen wird die Arraygröße statisch, d.h. zur Compilezeit festgelegt. Um sicher zu gehen, musste der Programmierer ausreichend viel Speicherplatz reservieren, d.h. das sehr viel Platz unnötig verschenkt wurde.

- In “klassischen” imperativen Sprachen werden üblicherweise komplette Datenobjekte mitsamt ihren Feldern direkt in dem Array gespeichert.

- Java dagegen speichert nur einen Zeiger auf das Objekt. Dadurch muss in Java nur je ein Zeiger pro Objekt geschoben werden, nicht ganze Datenblöcke.

Vergleich Liste und Array

Listen

+ Größe passt sich der Anzahl der aktuell gespeicherten Daten an
+ Einfügen und entfernen von Elementen geht in konstanter Zeit
- Auffingen eines Elementes anhand seiner Position benötigt Zeit O(n)

Array

+ Auf Elemente kann man direkt (in konstanter Zeit) zugreifen
- Größe eines Arrays zur Laufzeit nicht mehr verändert werden

Array-Listen

ArrayList

```
import java.util.*;
public class ListBeispiel {
   public static void main (String[] args) {
   // Liste erzeugen und fünf Elemente anhängen
      List<String> liste = new ArrayList<>();
         liste.add ("Frieder");
         liste.add ("Marie");
         liste.add ("Laura");
         liste.add ("Uli");
         System.out.println (liste);
   // Referenz auf ein String-Objekt mit dem Inhalt Karl an Position 2 einfügen
        liste.add (2, "Karl");
        System.out.println (liste);
  // Die Referenz an Position 3 entfernen
        liste.remove (3);
        System.out.println (liste);
  // Die Referenz entfernen, die auf ein Objekt mit dem Inhalt "Karl" zeigt
  //  (das betroffene Objekt wird intern mit Hilfe der equals()-Methode gesucht).
      liste.remove ("Karl");
      System.out.println (liste);
       }
}
```

ArrayList

```
import java.util.*;
public class ListBeispiel {
  public static void main (String[] args) {
  // Liste erzeugen und fünf Elemente anhängen
    List<String> liste = new ArrayList<>();
      liste.add ("Frieder");
      liste.add ("Marie");
      liste.add ("Laura");
      liste.add ("Uli");
      System.out.println (liste);
  // Referenz auf ein String-Objekt mit dem Inhalt Karl an Position 2 einfügen
       liste.add (2, "Karl");
       System.out.println (liste);
  // Die Referenz an Position 3 entfernen
       liste.remove (3);
       System.out.println (liste);
  // Die Referenz entfernen, die auf ein Objekt mit dem Inhalt "Karl" zeigt
  //  (das betroffene Objekt wird intern mit Hilfe der equals()-Methode gesucht).
      liste.remove ("Karl");
      System.out.println (liste);
       }
}
```

Freie Universität Berlin

Dynamische Datenstrukturen

EINFACH VERKETTE LISTEN

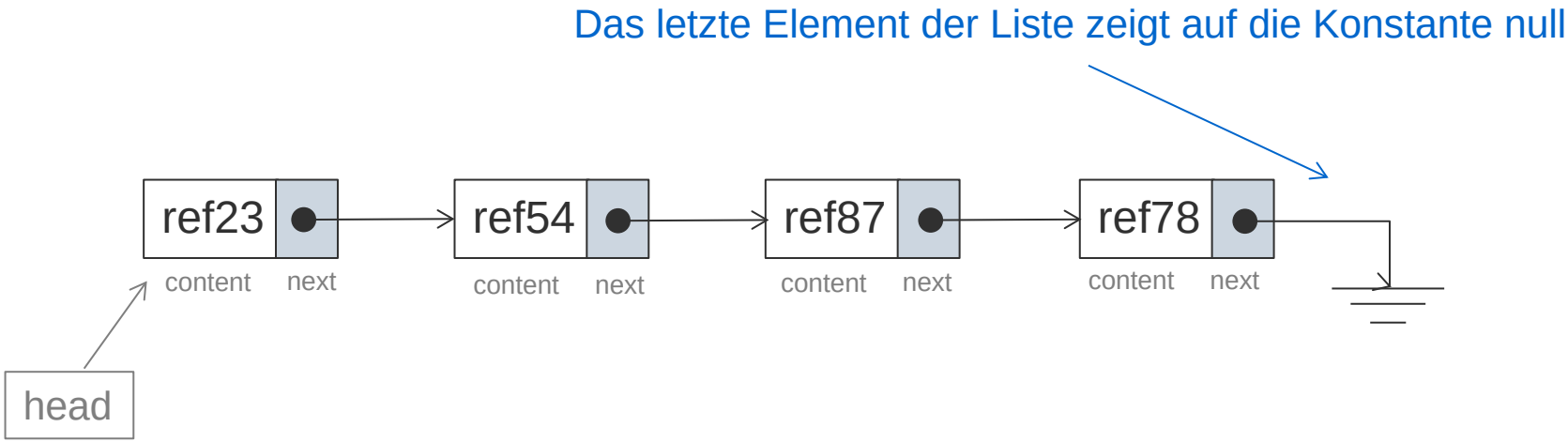
Einführung

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

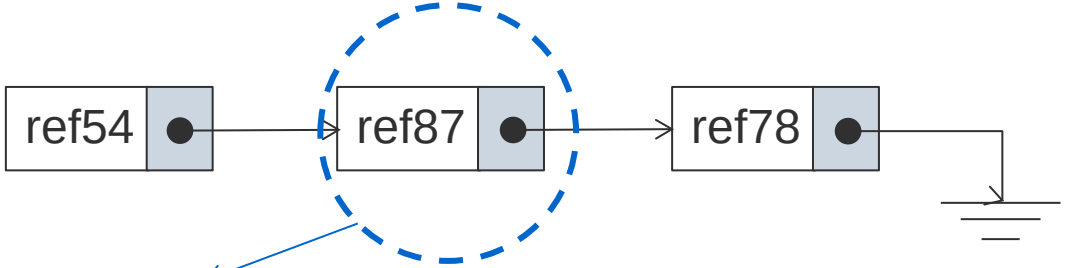
- Eine Liste besteht aus einer Menge von Knoten, die untereinander verkettet sind.

- Jeder Knoten besteht aus einer Referenz auf das eigentliche zu speichernde Objekt (e) und eine Referenz auf das nächste Element der Liste (`next`).

Eine Kette von vier Knoten



Implementierung der einfach verketteten Liste




```
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 ); }
}
```

Wir haben eine rekursive Klassendefinition.

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.

Dynamische Datenstrukturen

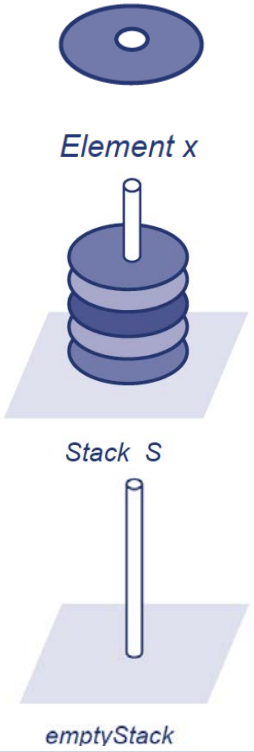
IMPLEMENTIERUNG DES STACKS DURCH EINE VERKETTETE LISTE

Wiederholung: Stackoperationen

- Der einfachste Stapel ist der *leere Stapel*, d.h. *emptyStack*

- Die Prüfung des Stacks kann mittels eines Prädikats *isEmpty* erfolgen

- Grundlegende Stackoperationen sind:
  - push(x, s) legt ein Element x auf den Stack s,
  - top(s) liefert das zuletzt auf den Stack s gelegte Element,
  - pop(s) entfernt das zuletzt auf den Stack s gelegte Element.

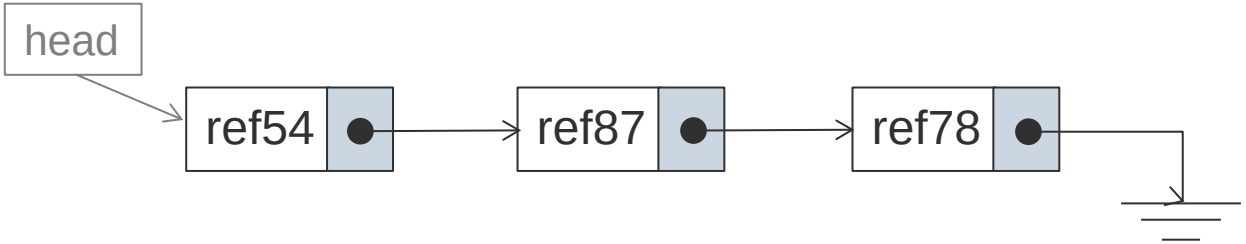


Stack als verkettete Liste

- 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 verketten oder entfernen, wenn eine `pop`-Operation stattfindet.

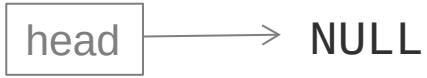


Einfache Implementierung der Stapel-Schnittstelle

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

Ein Konstruktor initialisiert head mit der Konstante null.

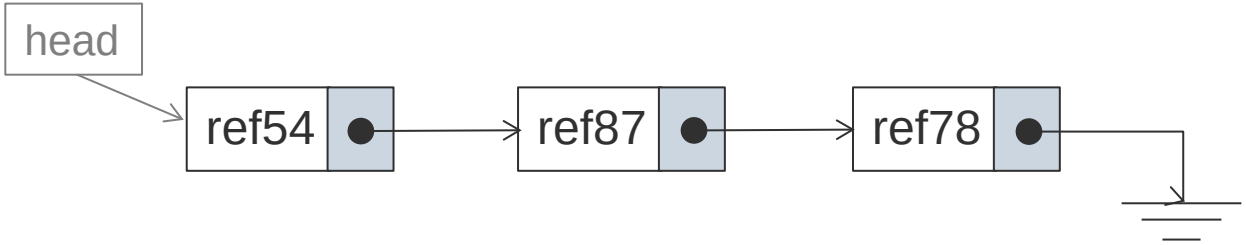
Implementierung der empty-Operation




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

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

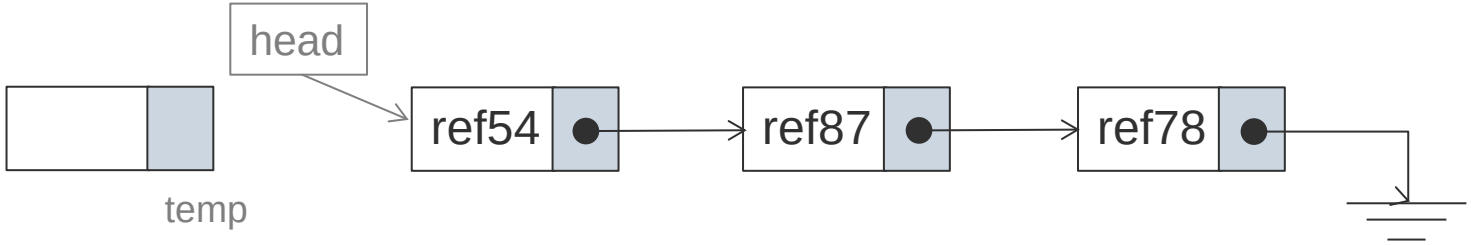
Implementierung der push-Operation




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

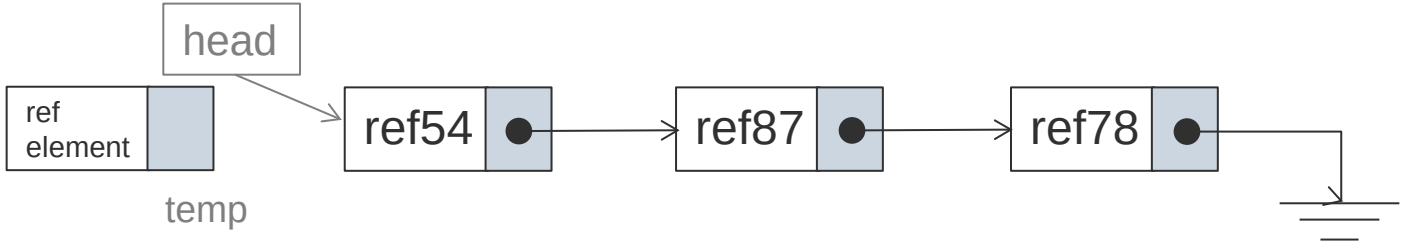
Das Objekt mit der Referenz `element` soll am Anfang der Liste eingefügt werden.

Implementierung der push-Operation



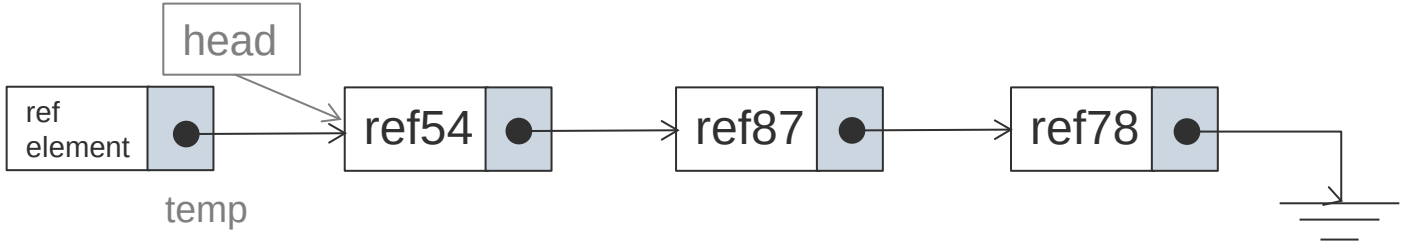

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation



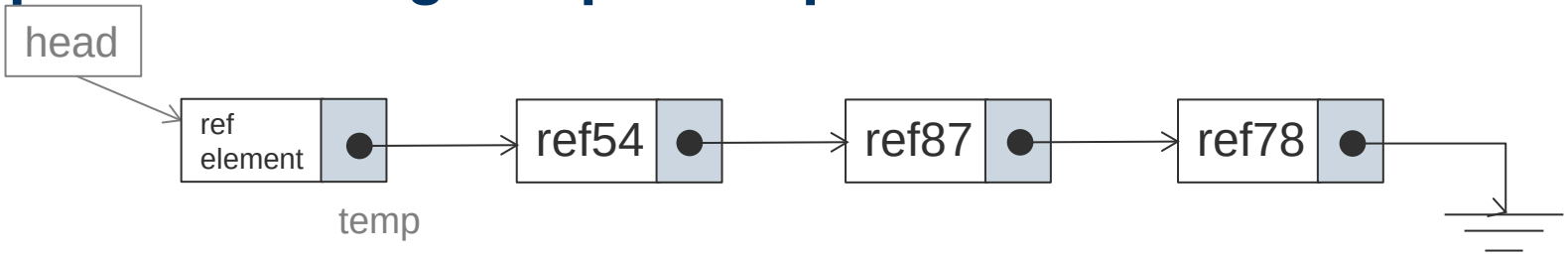

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation



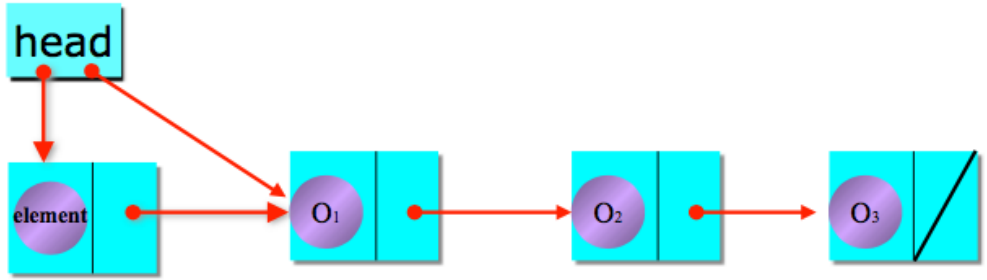

```
public void push ( T element ) {
    ListNode<T> temp = new ListNode<T> ();
    temp.element = element;
    temp.next = head;
    head = temp;
}
```

Implementierung der push-Operation




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

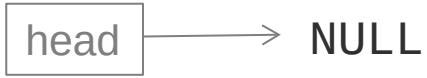
Implementierung der push-Operation



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 );
}
```

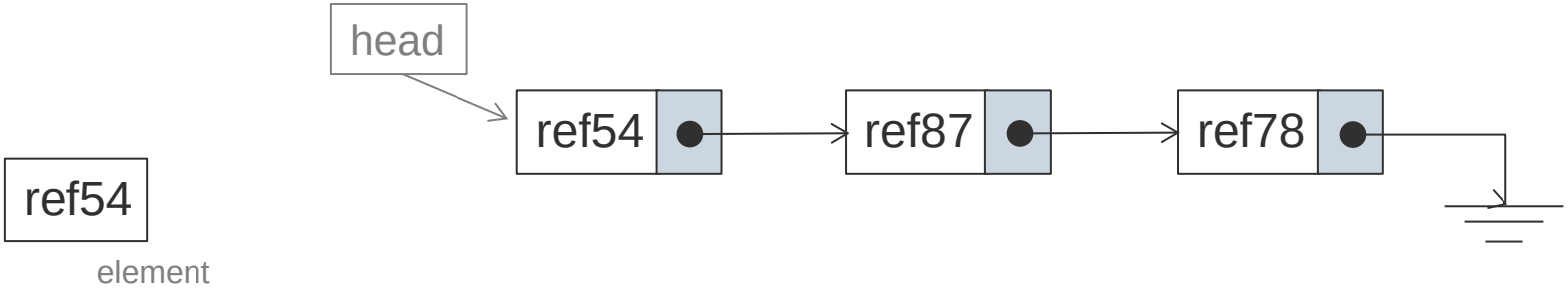
Implementierung der pop-Operation




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

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

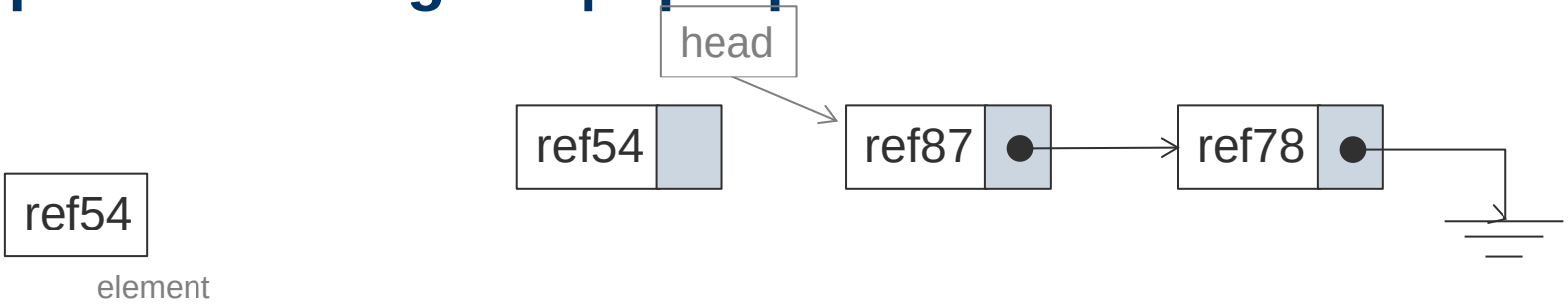
Implementierung der pop-Operation




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

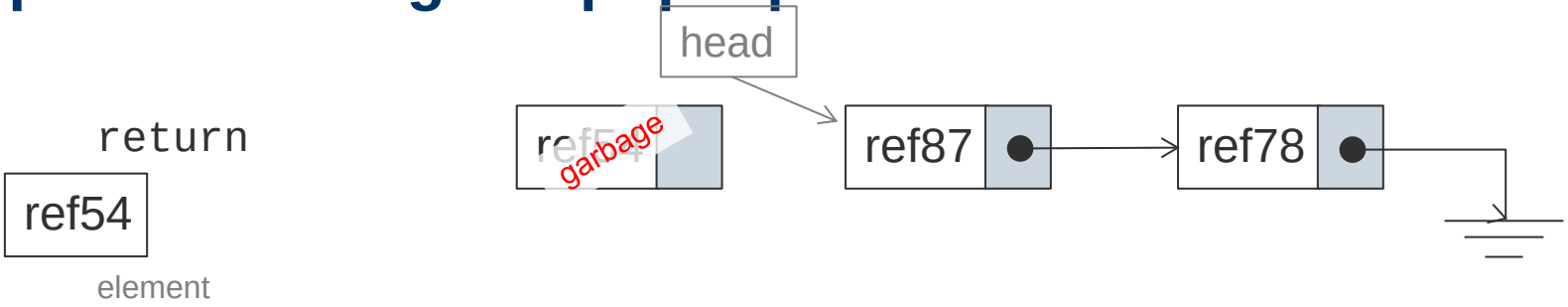
Die Objekt-Referenz, die sich in dem ersten Knoten befindet, wird in die lokale Variable `element` gespeichert.

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;
}
```

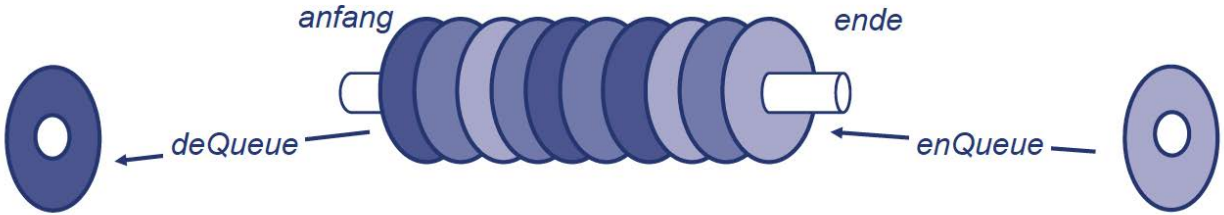
Das entfernte ListNode-Objekt bleibt ohne eine einzige Referenz, das auf es zeigt, und verwandelt sich in Datenspeichermüll, der später von dem Java-“garbage collector“ beseitigt wird.

Dynamische Datenstrukturen

IMPLEMENTIERUNG DER QUEUE DURCH EINE VERKETTETE LISTE

Wiederholung: Queue-Operationen

- Die einfachste Queue ist die *leere* Queue , d.h. empty*Queue*
- Die Prüfung der Queue kann mittels eines Prädikats *isEmpty oder isFull* erfolgen.
- Grundlegende Queue-Operationen sind:
  - enqueue ist die Einfüge-Operation.
  - dequeue ist die Lösch-Operation.

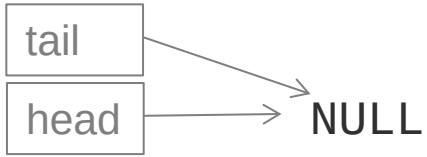


Einfache Implementierung der Queue-Schnittstelle

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

Der Konstruktor initialisiert head und tail mit der Konstante null.

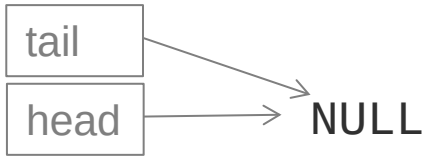
Implementierung der empty-Operation




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

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

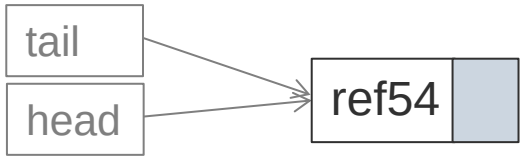
Implementierung der enqueue-Operation



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 );
}
```

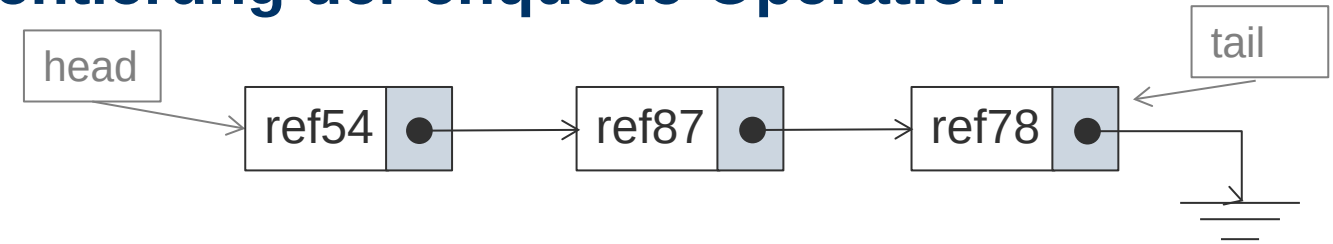
Implementierung der enqueue-Operation




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

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

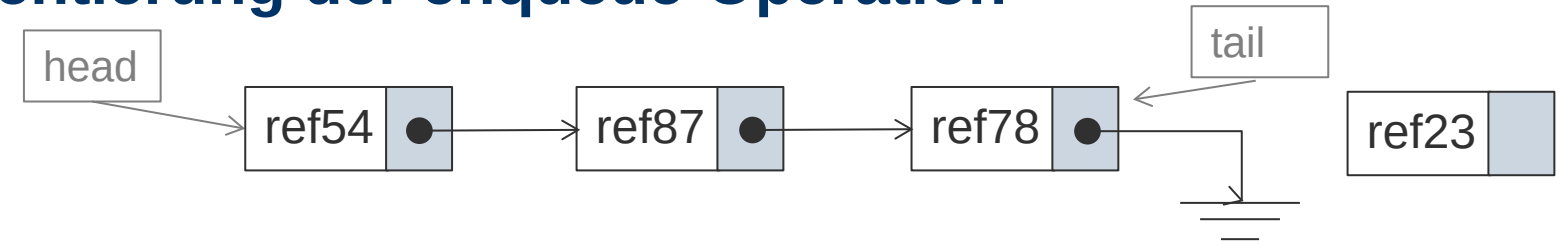
Implementierung der enqueue-Operation



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 );
}
```

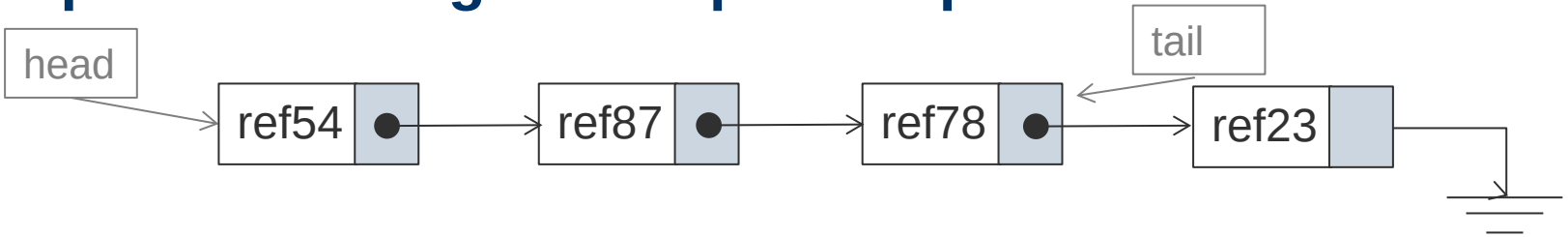
Implementierung der enqueue-Operation




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

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

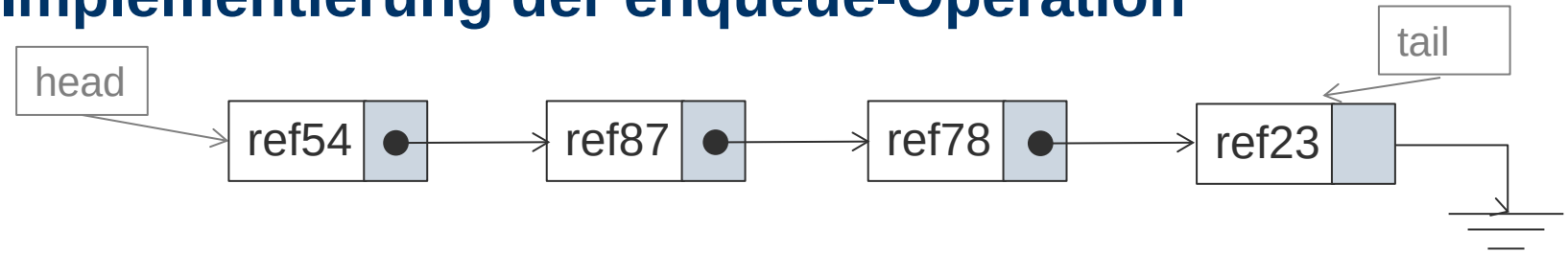
Implementierung der enqueue-Operation




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

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

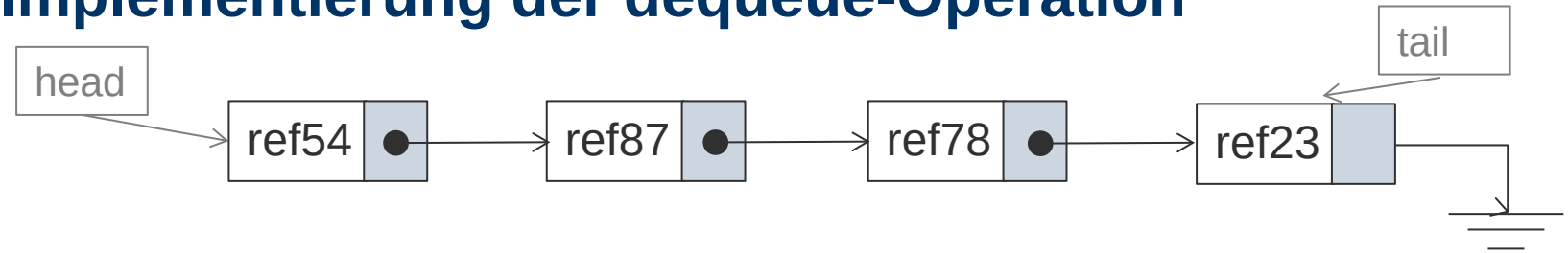
Implementierung der enqueue-Operation



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

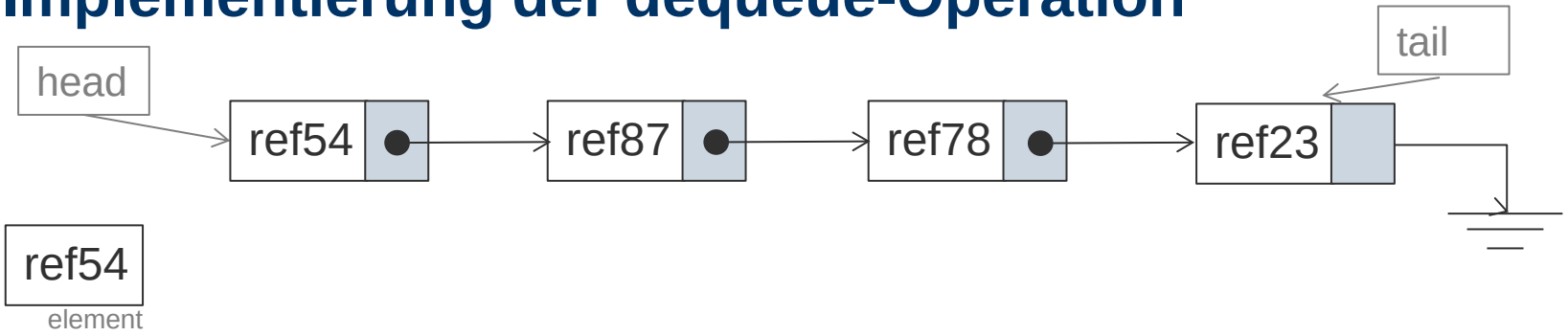
```
public void enqueue ( T newElement ) {
   if ( empty() )
      head = tail = new ListNode<T> ( newElement );
   else
      tail = tail.next = new ListNode<T>( newElement );
}
```

Implementierung der dequeue-Operation




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

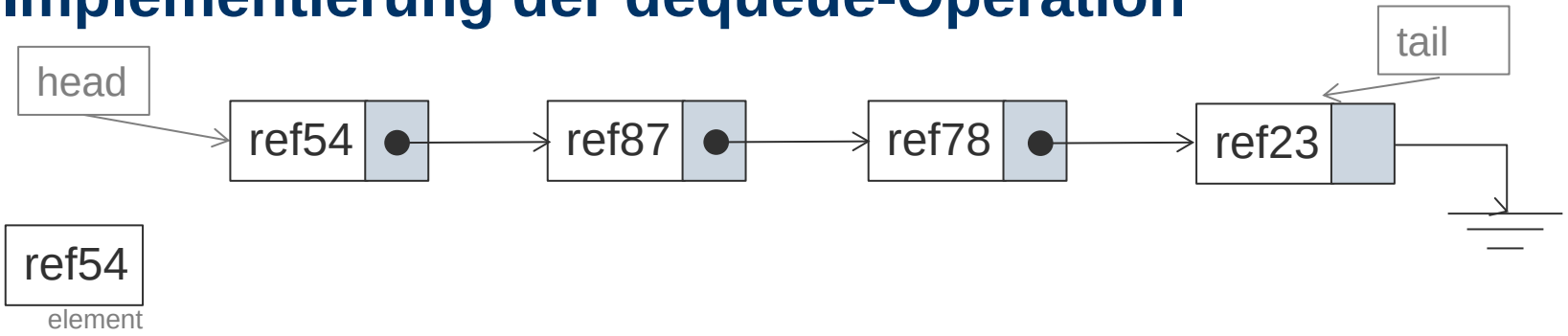
Implementierung der dequeue-Operation




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

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

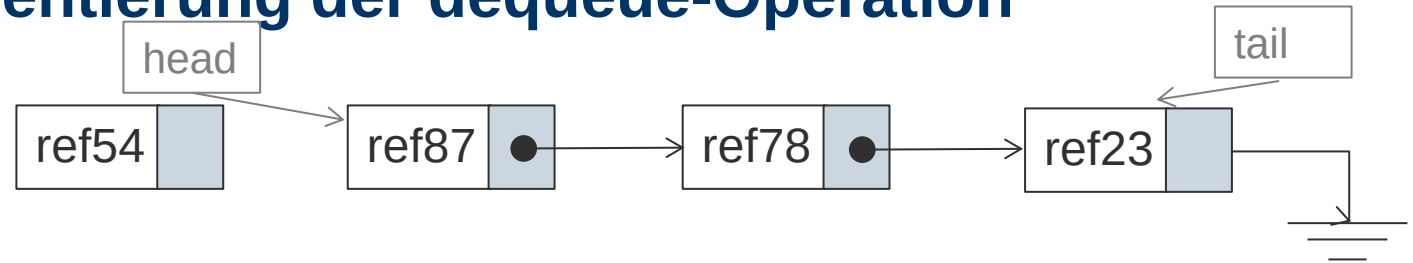
Implementierung der dequeue-Operation




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

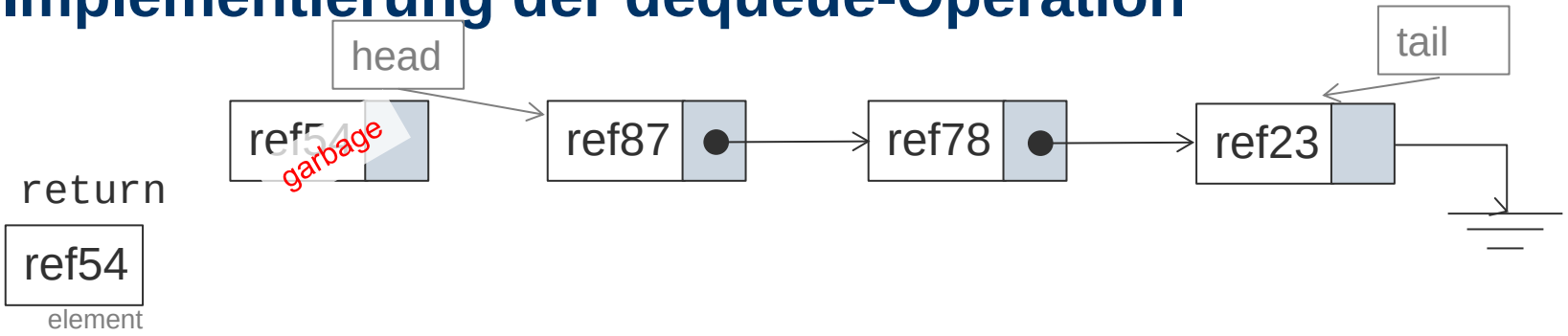
Die Referenz des Elements, das am Anfang der Warteschlange gespeichert ist, wird der Variablen element zugewiesen.

Implementierung der dequeue-Operation




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

Implementierung der dequeue-Operation




```
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.

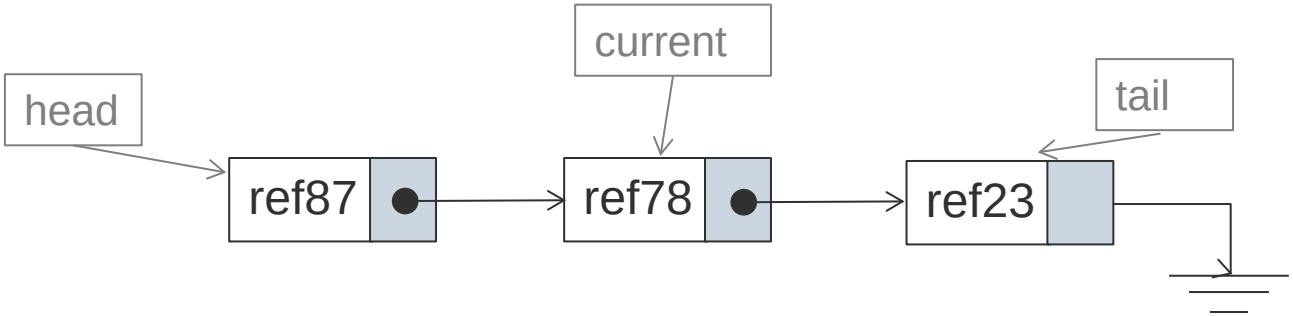
Freie Universität Berlin

Dynamische Datenstrukturen

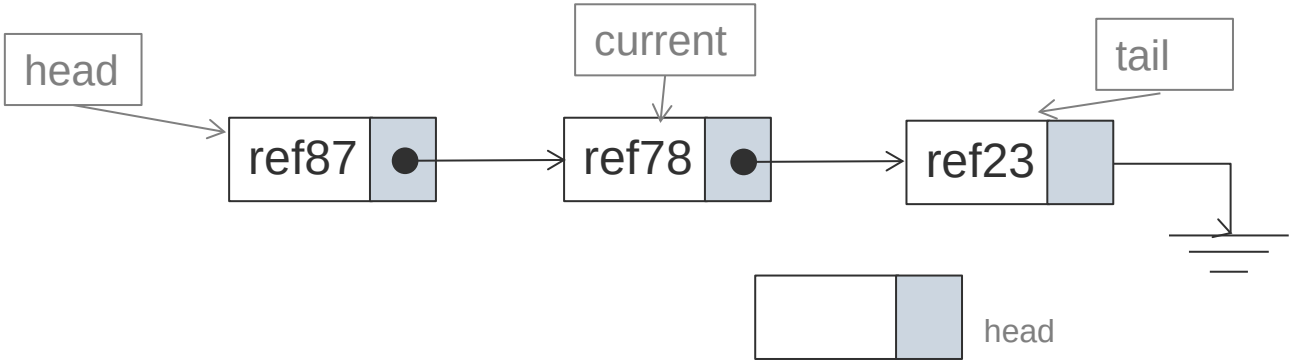
EINFÜGEN UND LÖSCHEN AN BELIEBIGEN STELLEN DER VERKETTETEN LISTE

Ansatz 1 – Referenz-Objekt `current`

- Wir wollen an einer beliebigen Stelle der Liste Elemente einfügen und löschen. Dafür brauchen wir ein weiteres Referenz-Objekt `current` das sich durch die Liste bewegt.

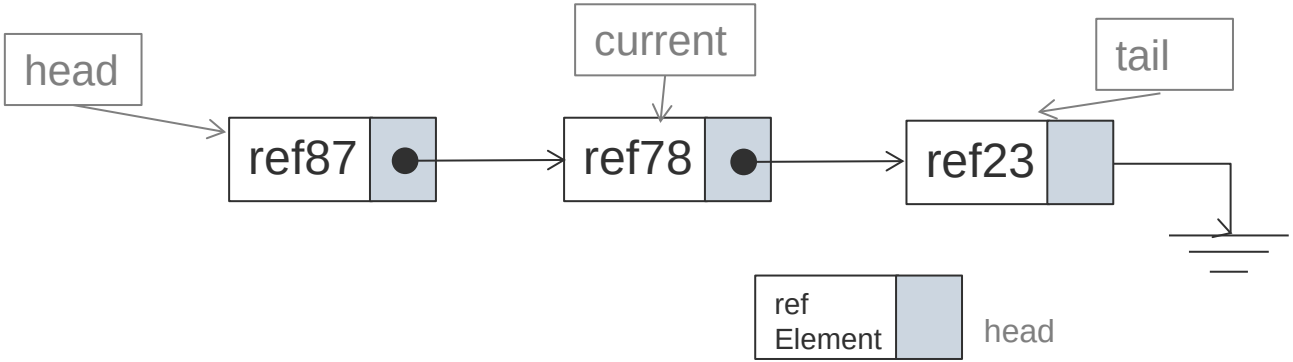


Einfügen an einer beliebigen Stelle der Liste



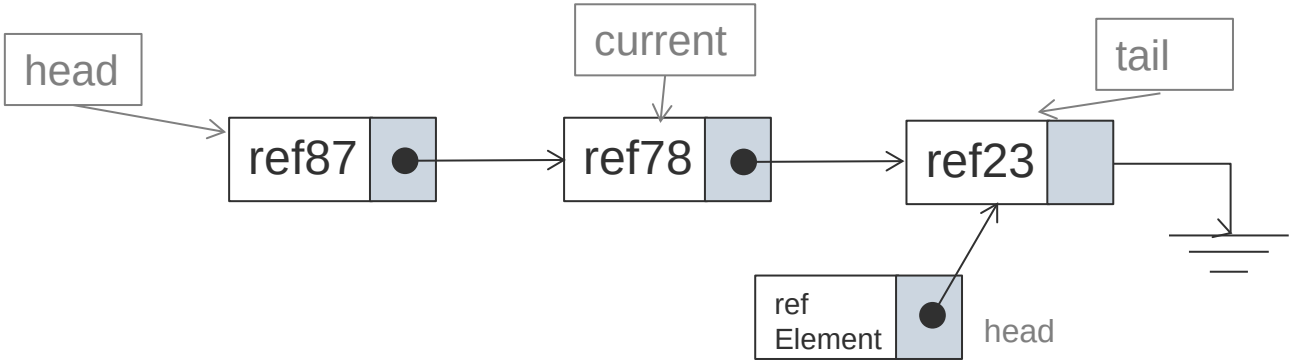

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = element;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an einer beliebigen Stelle der Liste



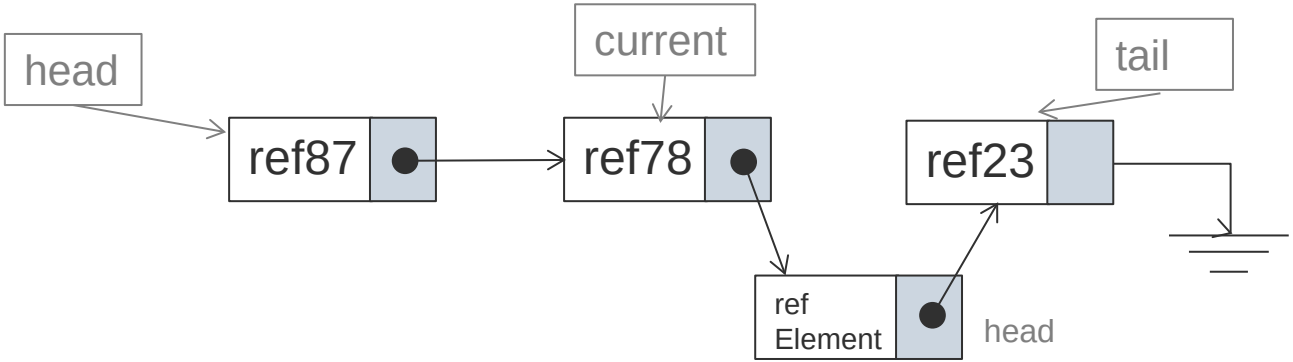

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = element;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an einer beliebigen Stelle der Liste



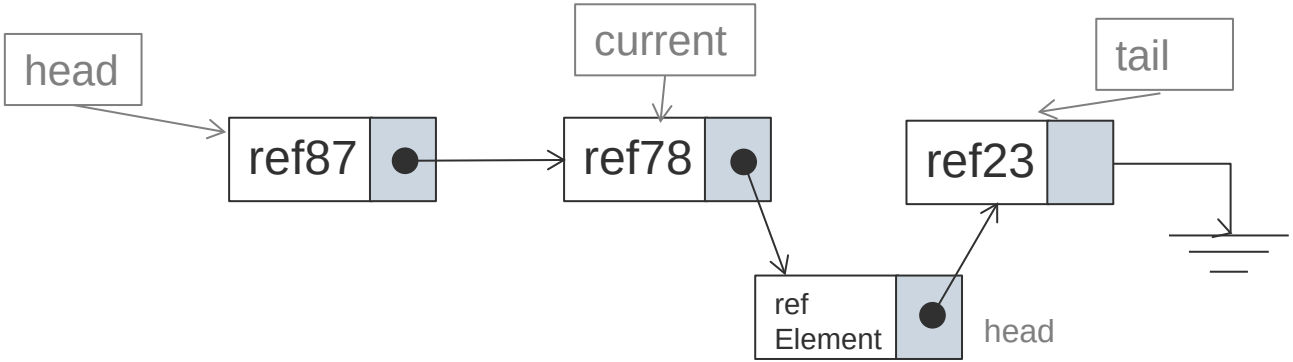

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = element;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an einer beliebigen Stelle der Liste



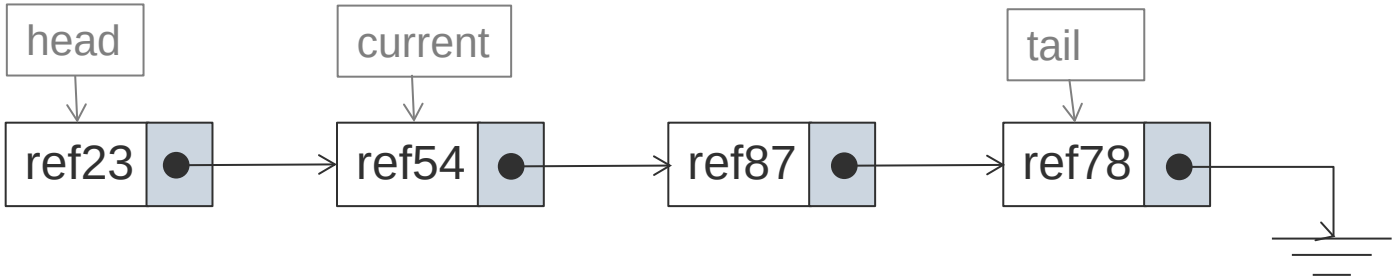

```
. . .
ListNode<T> temp = new ListNode<T>();
temp.element = element;
temp.next = current.next;
current.next = temp;
. . .
```

Einfügen an einer beliebigen Stelle der Liste



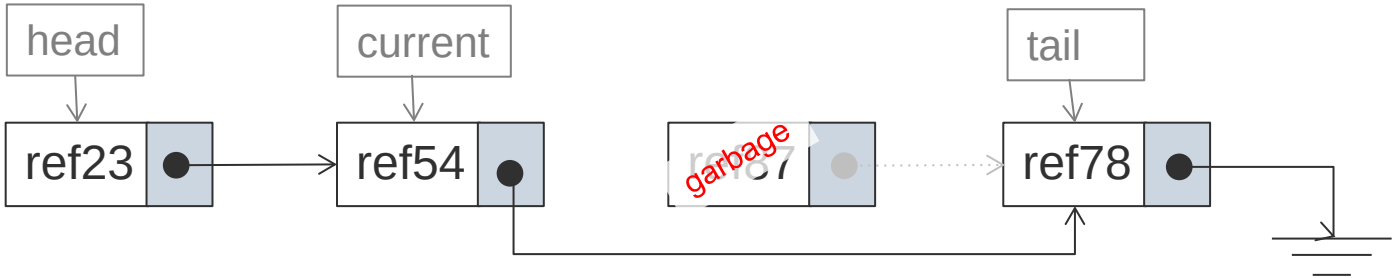

```
…
current.next = new ListNode<T>( element, current.next );
…
```

Löschen an einer beliebigen Stelle der Liste




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

Löschen an einer beliebigen Stelle der Liste



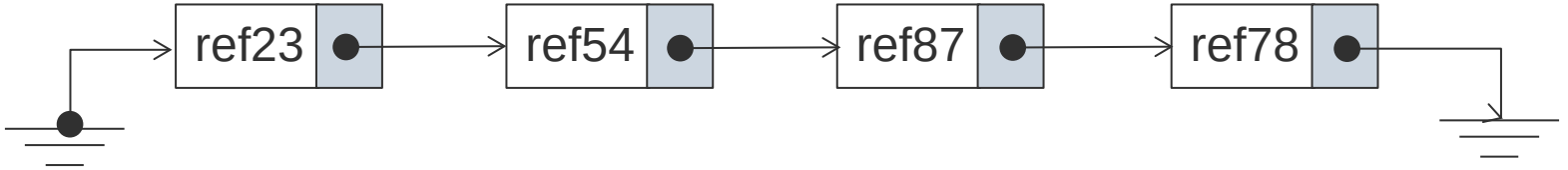

```
…
current.next = current.next.next;
…
```

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.

Ansatz 2: Dummy-Element

- 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 bestimmte spezielle Fälle nicht berücksichtigt werden müssen, wie das
  - Löschen des ersten Elements der Liste
  - Einfügen, wenn die Liste leer ist
- Daher führen wir einen dummy-Element ein.

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



Zwischenfazit

Verkettete Liste

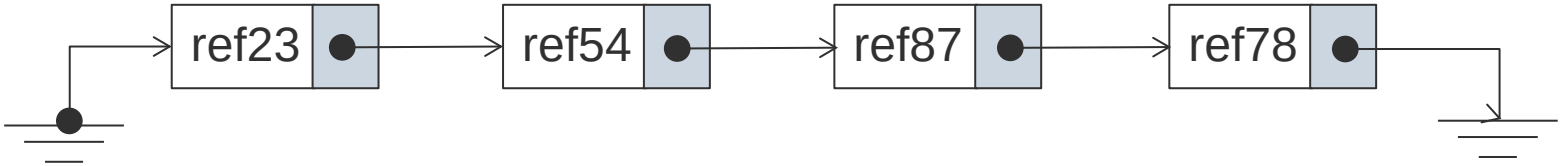
+ Einfaches Einfügen/Löschen
+ Gut für sequentiellen Zugriff auf Listenelemente
+ Einfaches Trennen oder Zusammenfügen von Listen
+ Gestaltung komplexer Datenstrukturen ist möglich
+ flexible Speichernutzung (dynamische Speicherverwaltung)
– zusätzlicher Speicherplatz
– Zugriff auf k-tes Element erfordert k Iterationen

Sequentielle Liste (Array)

+ Direkter Zugriff auf Listenelemente ist einfach
+ Zugriff auf k-tes Element benötigt immer eine konstante Zeit
– sind statisch und können verschwenderisch bzgl. des Speichers sein
– Einfügen/Löschen eines Elementes erfordert erheblichen Kopieraufwand.

Probleme bei einfach verketteten Listen

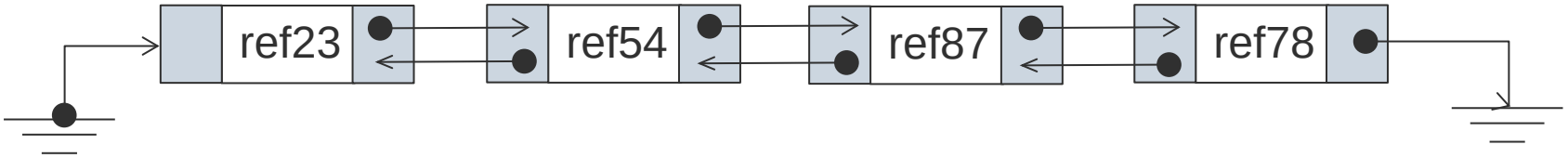
- Der `current`-Zeiger kann nur nach vorne bewegt werden.
- Um den Vorgänger desjenigen Knotens zu finden, auf den `current` zeigt, müssen wir die ganze Liste von head an durchlaufen.
- Das ist im allgemeinen sehr ineffizient.



Lösung: Einführung von zusätzlichen Zeigern

Erweiterung zu doppelt verketteten Listen

- Höhere Flexibilität bei der Manipulation von Listen Vorgehen:
  - Für jedes Element zwei Zeiger verwalten:
  - `prev` und `next`-Zeiger auf Vorgänger und Nachfolger
  - `prev` und `next` sind Zeigervariablen, die auf das linke bzw. das rechte Ende der Liste zeigen

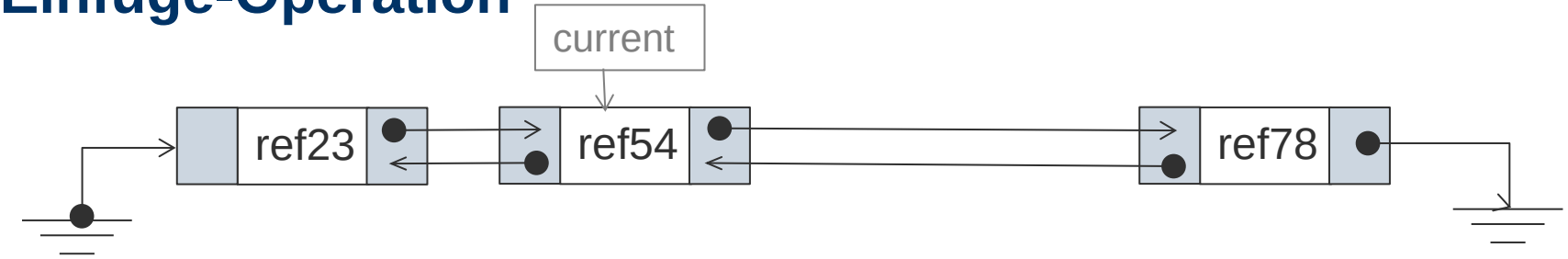


Freie Universität Berlin

Dynamische Datenstrukturen

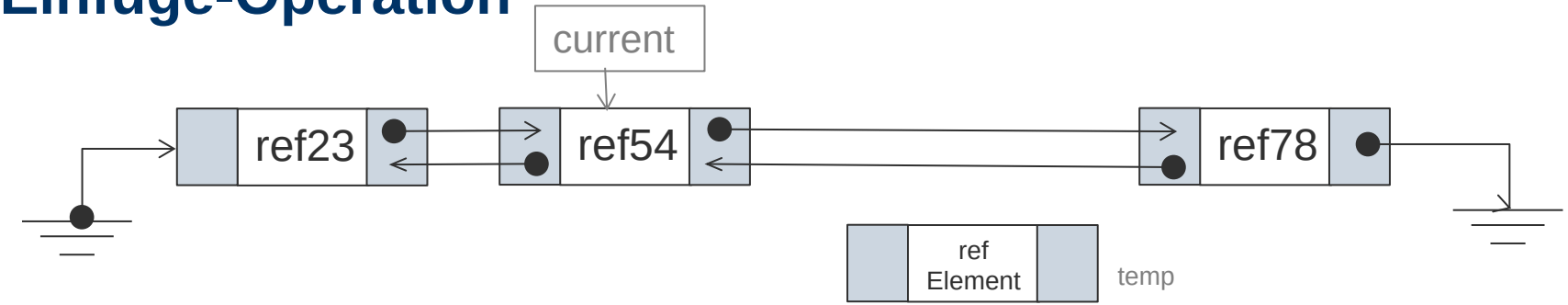
DOPPELT VERKETTETE LISTEN

Einfüge-Operation



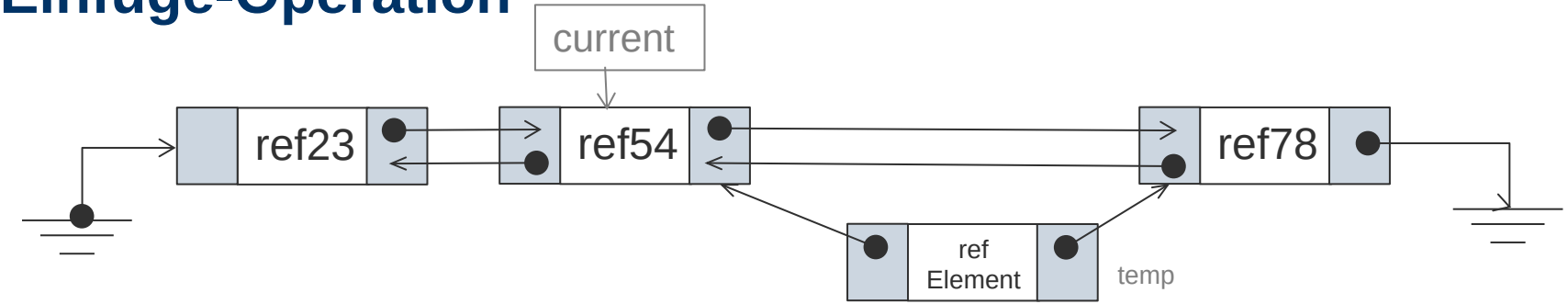

```
...
ListNode<T> temp = new ListNode<T>(element);
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;
...
```

Einfüge-Operation



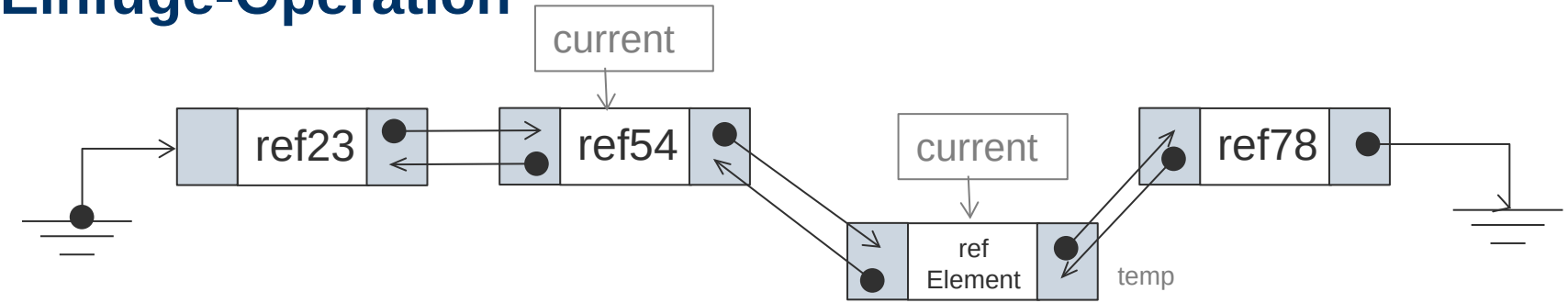

```
...
ListNode<T> temp = new ListNode<T>(element);
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;
...
```

Einfüge-Operation



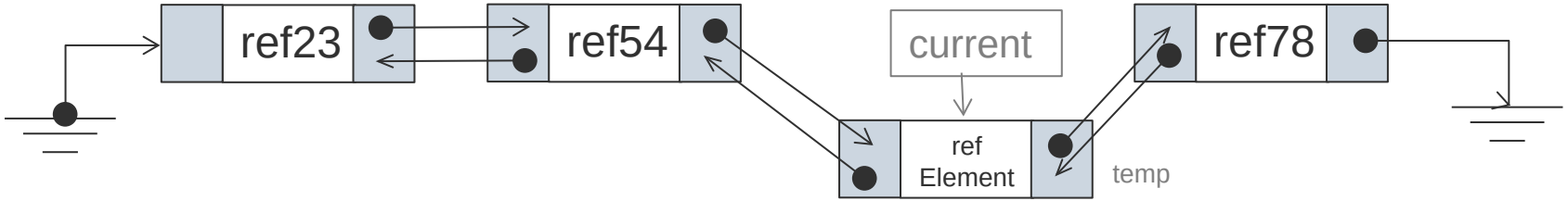

```
...
ListNode<T> temp = new ListNode<T>(element);
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;
...
```

Einfüge-Operation



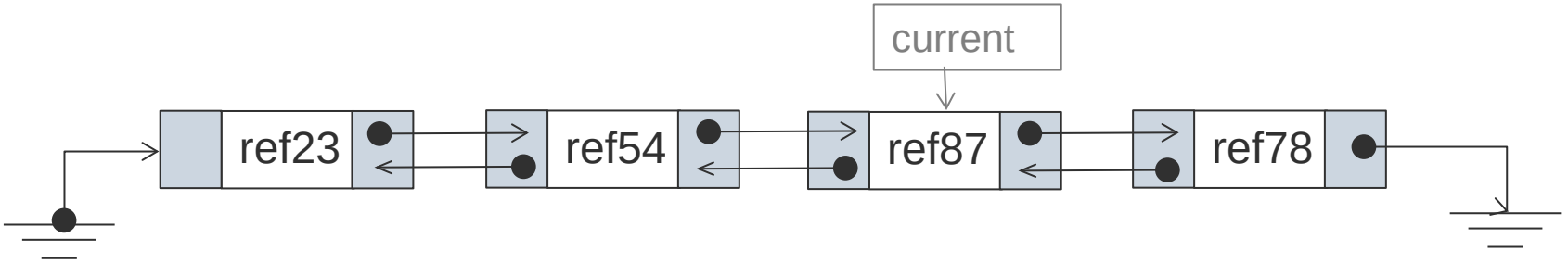

```
...
ListNode<T> temp = new ListNode<T>(element);
temp.prev = current;
temp.next = current.next;
temp.prev.next = temp;
temp.next.prev = temp;
current = temp;
...
```

Einfüge-Operation




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

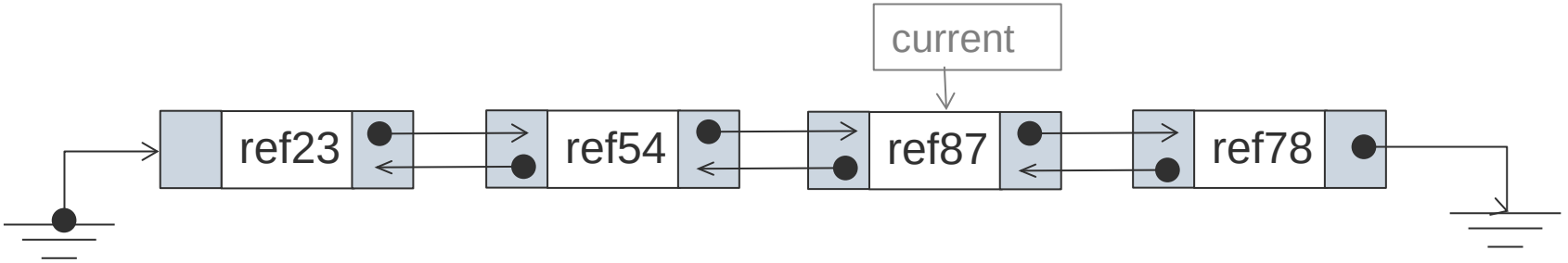
Lösch-Operation




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

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

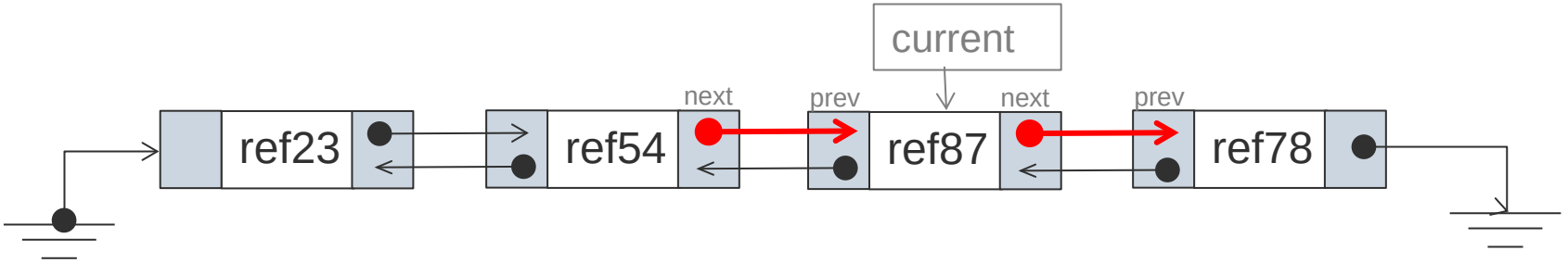
Lösch-Operation




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

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

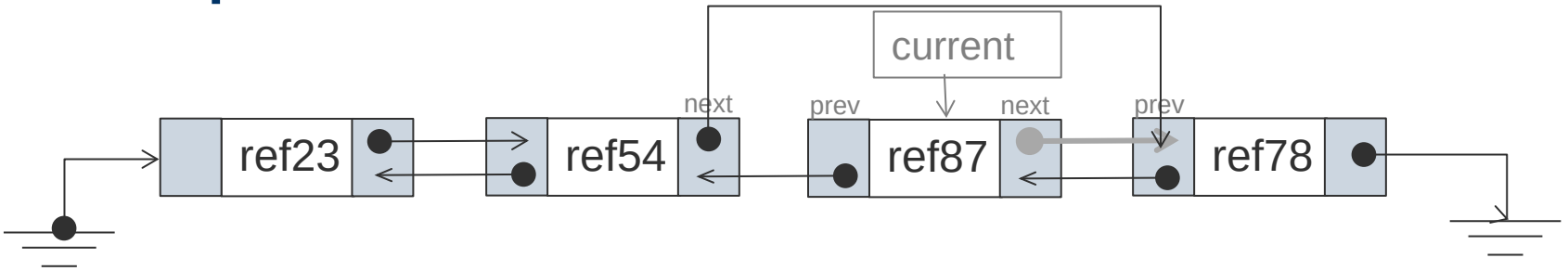
Lösch-Operation




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

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

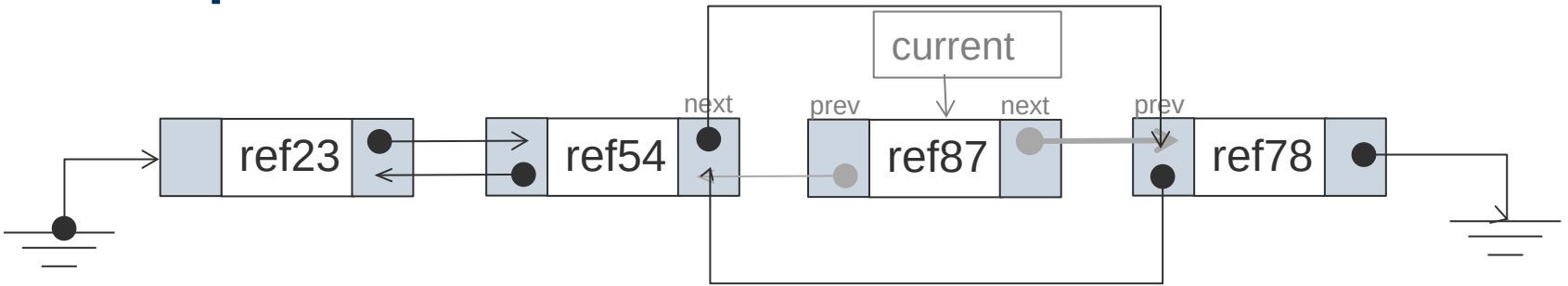
Lösch-Operation




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

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

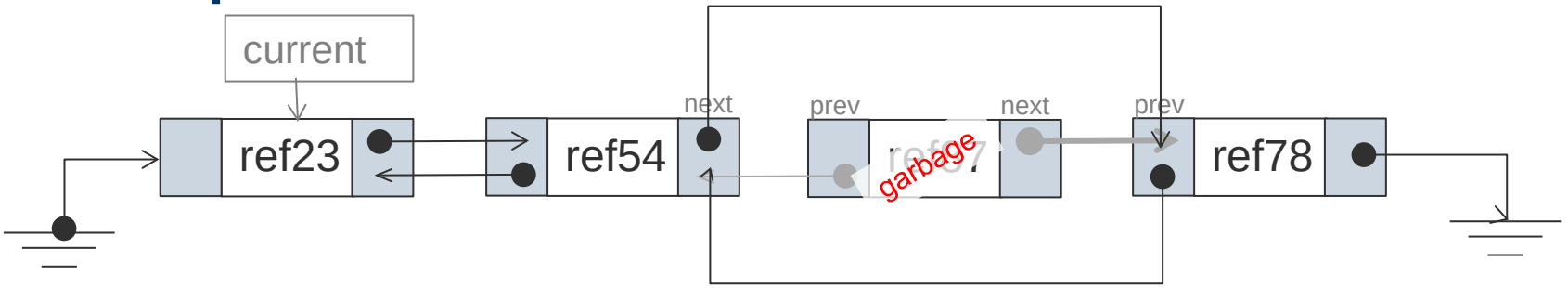
Lösch-Operation




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

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

Lösch-Operation




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

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

Fazit: Doppelt verkettete Listen

Vorteile

- Durchlauf durch Listen in beide Richtungen möglich
- Einfache Umsetzung von Operationen wie:
  - Füge Element zwischen zwei bestimmte Elemente ein
  - Finde das Element vor einem Element
  - Lösche Element

Nachteile

- Erfordert zusätzlichen Speicher für zusätzlichen Zeiger
- Mehr explizites Management für zusätzliche Zeiger
- Fehleranfälliges Programmieren aufgrund der Zeigerverwaltung

Fazit: Elementare Datenstrukturen

- Sequentielle und verkettete Listen
- relativ einfache Operationen
- hohe Flexibilität bei Zugriffen auf Elemente der Listen/Felder
- ABER: hohe Flexibilität kostet Aufwand, z.B.
  - Suchaufwand, Durchlaufen von Listen
  - Umkopieren, ‚Umhängen‘ von Zeigern etc.

- Einführen von Datenstrukturen mit effizienten Operationen, d.h. der Verzicht auf Flexibilität zugunsten von Effizienz
  - Mögliche Umsetzungen dafür sind Stapel und Queues ☺

Freie Universität Berlin

Dynamische Datenstrukturen

ZUSAMMENFASSUNG

Was haben wir besprochen

- Wir haben die grundlegenden Datenstrukturen Array, ArrayListe, einfach verkettete Liste und doppelt verkettete Liste kennengelernt.

- Sie kennen die Unterschiede zwischen den einzelnen Datenstrukturen und können die Vor- und Nachteile nennen.

- Sie wissen wie man mithilfe einer Datenstruktur andere Datenstrukturen implementiert, z.B. Stacks und Queues jeweils mit Hilfe einer einfach verketteten Liste implementiert werden.