

17. Datenstrukturen

17.1 Einleitung

17.2 Listen

17.3 Assoziative Speicher

17.4 Mengen

17.5 Bäume

17.6 Das Collections-Framework in Java

17.7 Zusammenfassung

17. Datenstrukturen

17.1 Einleitung

17.2 Listen

17.3 Assoziative Speicher

17.4 Mengen

17.5 Bäume

17.6 Das Collections-Framework in Java

17.7 Zusammenfassung

- Viele Computer-Programme sind in erster Linie dazu da, Daten zu verarbeiten.
- Eine Datenmenge muß dazu intern organisiert und verwaltet werden.
- Als einfache Datenstruktur zur Verwaltung gleichartiger Elemente haben wir für imperative Sprachen das Array kennengelernt.
- In funktionalen Sprachen kennen wir zum gleichen Zweck die Liste.
- Auch eine Klasse dient zunächst der Darstellung von Objekten, die einen Ausschnitt der Wirklichkeit abstrahiert repräsentieren.
- Als spezielle Datenstruktur können wir auch die Strings (und verwandte Klassen) betrachten, die für eine Menge von Zeichen stehen.

- Bei vielen Anwendungen besteht die wichtigste Entscheidung in Bezug auf die Implementierung darin, die passende Datenstruktur zu wählen.
- Verschiedene Datenstrukturen erfordern für dieselben Daten mehr oder weniger Speicherplatz als andere.
- Für dieselben Operationen auf den Daten führen verschiedene Datenstrukturen zu mehr oder weniger effizienten Algorithmen.
- Die Auswahlmöglichkeiten für Algorithmus und Datenstruktur sind eng miteinander verflochten. Durch eine geeignete Wahl möchte man Zeit und Platz sparen.

- Eine Datenstruktur können wir auch wieder als Objekt auffassen und entsprechend modellieren.
- Das bedeutet, dass eine Datenstruktur Eigenschaften und Fähigkeiten hat, also z.B. typische Operationen ausführen kann.
- Für Arrays haben wir z.B. die typischen Operationen:
 - Setze das i -te Element von a auf Wert x : $a[i] = x$;
 - Gib mir das j -te Element von a : $a[j]$;
 - Gib mir die Anzahl der Elemente in a : $a.length$

- Wie wir gesehen haben, erlauben Arrays effizient den sogenannten “wahlfreien Zugriff”, d.h. wir können auf ein beliebiges Element in $O(1)$ zugreifen.
- Bei der Listen-Implementierung, die wir von SML kennen, gilt das nicht. Der Zugriff auf das n -te Element erfordert einen Aufwand in $O(n)$.
- Dafür können Listen beliebig wachsen, während wir die Größe eines Arrays von vornherein festlegen müssen.
- Betrachten wir nun im folgenden die Listen etwas genauer, wozu wir uns jetzt natürlich objektorientierte Implementierungen in Java vornehmen wollen.

17. Datenstrukturen

17.1 Einleitung

17.2 Listen

17.3 Assoziative Speicher

17.4 Mengen

17.5 Bäume

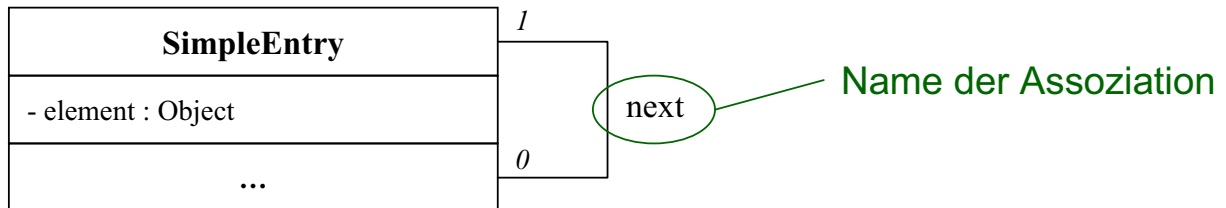
17.6 Das Collections-Framework in Java

17.7 Zusammenfassung

Erstellen wir zunächst eine einfache Listenimplementierung analog zu dem Verhalten von Listen, das wir aus SML kennen:

- Eine Liste kann leer sein.
- Ein Element wird vorne an eine Liste angehängt.
- Wir können nur das vordere Element einer Liste entfernen.
- Eine Liste kann Auskunft über ihre Länge (= Anzahl der Elemente) geben.

- Von der Objektorientierung herkommend, können wir ein Element der Liste zunächst als eigenständiges Objekt ansehen.
- Dieses Objekt hält das eigentlich gespeicherte Element.
- Andererseits hält das Element-Objekt einen Verweis auf das nächste Element, den Nachfolger.



```

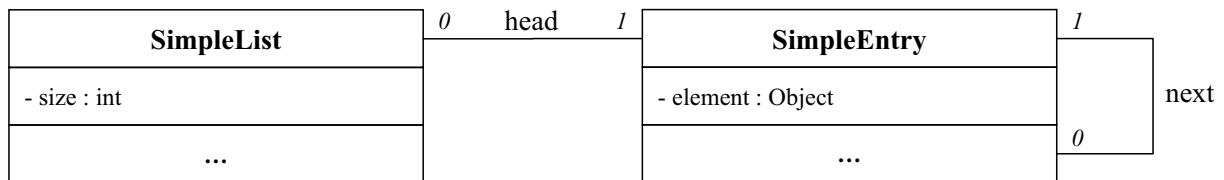
public class SimpleEntry
{
    private Object element;
    private SimpleEntry next;

    public SimpleEntry(Object o, SimpleEntry next)
    {
        this.element = o;
        this.next = next;
    }

    public Object getElement()
    {
        return this.element;
    }

    public SimpleEntry getNext()
    {
        return this.next;
    }

    public void setNext(SimpleEntry next)
    {
        this.next = next;
    }
}
  
```



Liste hält Verweis auf erstes Element

- Die Liste muss nur das erste Element kennen, über dessen Zeiger zum nächsten Element können alle Nachfolger erreicht werden.
- In der leeren Liste ist das erste Element **null**.

```

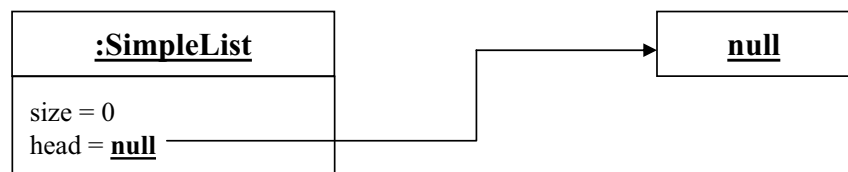
public class SimpleList
{
    private int size;

    private SimpleEntry head;

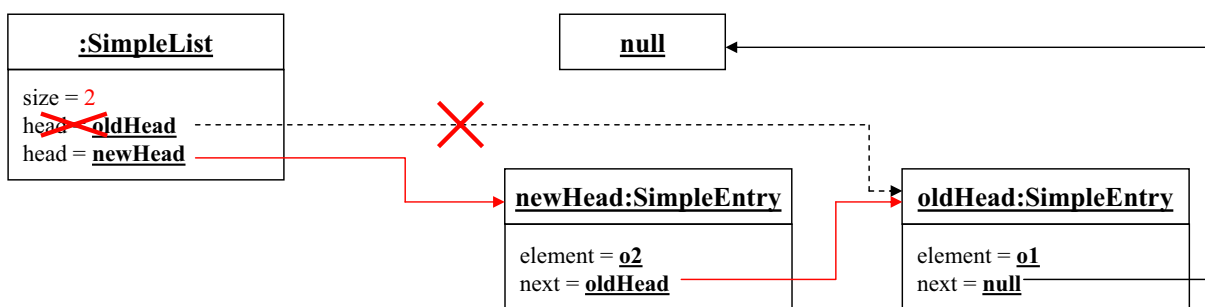
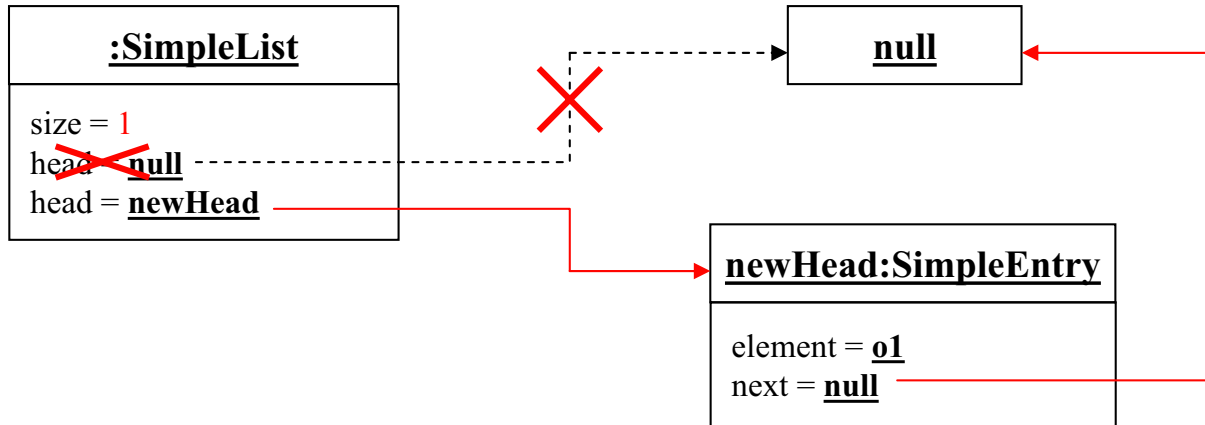
    public SimpleList()
    {
        this.size = 0;
        this.head = null;
    }

    public int length()
    {
        return this.size;
    }

    ...
}
  
```



- Um ein neues Element hinzuzufügen, wird ein neues SimpleEntry-Element erzeugt und als neues erstes Element gesetzt.
- Dessen Nachfolger ist das alte erste Element.
- Die Länge der Liste erhöht sich um 1.



```

...
public void add(Object o)
{
    SimpleEntry newHead = new SimpleEntry(o, this.head);
    this.head = newHead;
    this.size++;
}
...

```

- Nur den Wert des ersten Elementes zu bekommen, ist einfach.
- Probleme können bei der leeren Liste auftreten.

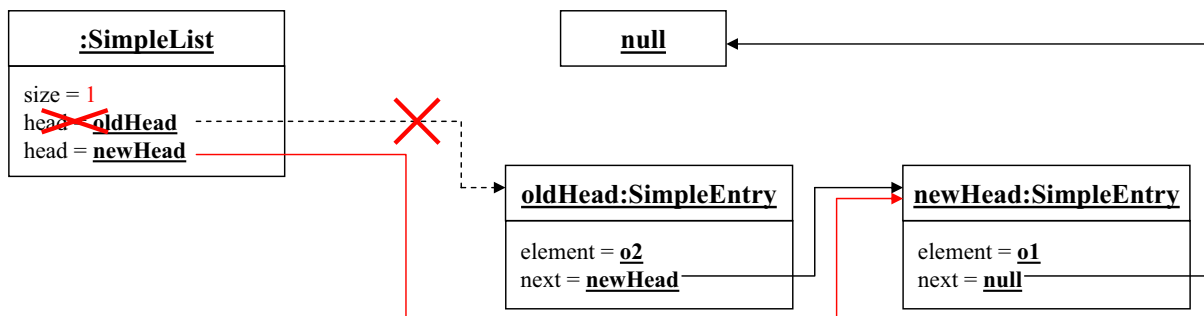
...

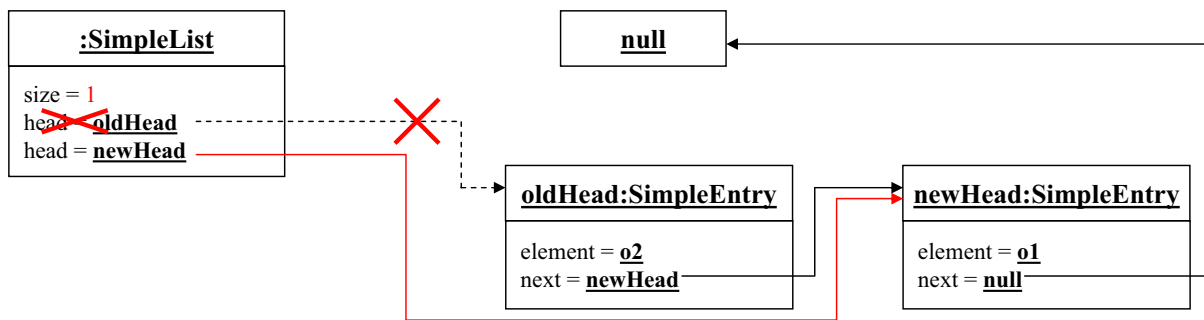
```
public Object head()
{
    if (this.head == null)
    {
        throw new NullPointerException("Empty List - no head element available.");
    }
    return this.head.getElement();
}

```

...

- Um das erste Element zu entfernen, muss der Head-Zeiger der Liste auf den Nachfolger des ersten Elementes zeigen.
- Die Länge der Liste wird erniedrigt.
- Danach gibt es keine Zugriffsmöglichkeit mehr für das erste Element.
- Wiederum können Probleme bei der leeren Liste auftreten.





```

public void removeHead()
{
    if (this.head == null)
    {
        throw new NullPointerException("Empty List - no head element available.");
    }
    this.head = this.head.getNext();
    this.size--;
}
  
```

Unterschied der Modellierung zum funktionalen Paradigma

- Im funktionalen Paradigma sind Listenfunktionen (wie alle Funktionen) Abbildungen, die die Eingabeparameter nicht verändern, sondern auf andere (neue) Werte *abbilden*.
- Wird eine Liste in der funktionalen Sprache um ein Element erweitert, dann gibt es anschließend beide Werte: die alte Liste und die um ein Element erweiterte Liste.
- Hier haben wir Prozeduren, die das aufrufende Objekt *verändern*. Die alte Liste ohne das neue Element gibt es nicht mehr.
- Vorteil im funktionalen Paradigma: Man hat keine Nebeneffekte.
- Vorteil im imperativen Paradigma: Man belegt den Speicherplatz nicht mehrfach.

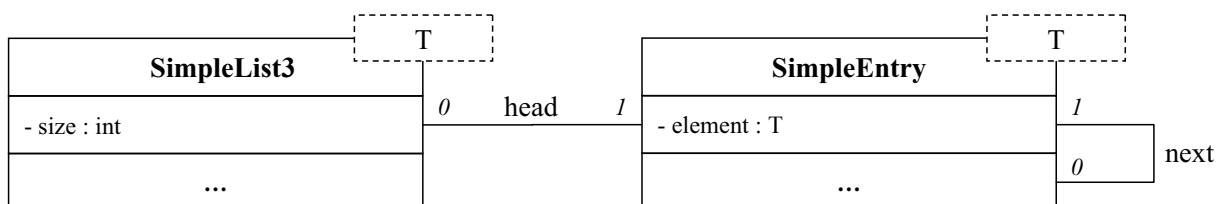
- Die Klasse `SimpleEntry` wird niemals außerhalb der Liste gebraucht, sondern dient nur als Wrapper für das eigentliche Element und die Verkettung zum nächsten Element.
- Java bietet eine Möglichkeit, um diese enge Beziehung auszudrücken: `SimpleEntry` kann man als innere Klasse der Liste definieren.

```
public class SimpleList2
{
    private int size;
    ...

    private static class SimpleEntry
    {
        private Object element;
        ...
    }
}
```

- Innere Klassen sollte man nur sehr zurückhaltend verwenden, aber in diesem Fall ist die Verwendung sinnvoll.

- Bisher können wir jedes beliebige Objekt in unserer Liste ablegen.
- Damit sind in der Verwendung dem Auftreten von Typfehlern Tür und Tor geöffnet.
- Wir kennen schon eine Möglichkeit, die Liste typsicher zu machen: Typisierung der Klasse.



```
public class SimpleList3<T>
{
    private int size;

    private SimpleEntry<T> head;

    public SimpleList3()
    {
        this.size = 0;
        this.head = null;
    }

    public int length()
    {
        return this.size;
    }
    ...
}
```

```
...
public void add(T o)
{
    SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
    this.head = newHead;
    this.size++;
}

public T head()
{
    if(this.head==null)
    {
        throw new NullPointerException("Empty List - no head element available.");
    }
    return this.head.getElement();
}

public void removeHead()
{
    if(this.head==null)
    {
        throw new NullPointerException("Empty List - no head element available.");
    }
    this.head = this.head.getNext();
    this.size--;
}
...
}
```

```
...
private static class SimpleEntry<T>
{
    private T element;

    private SimpleEntry<T> next;

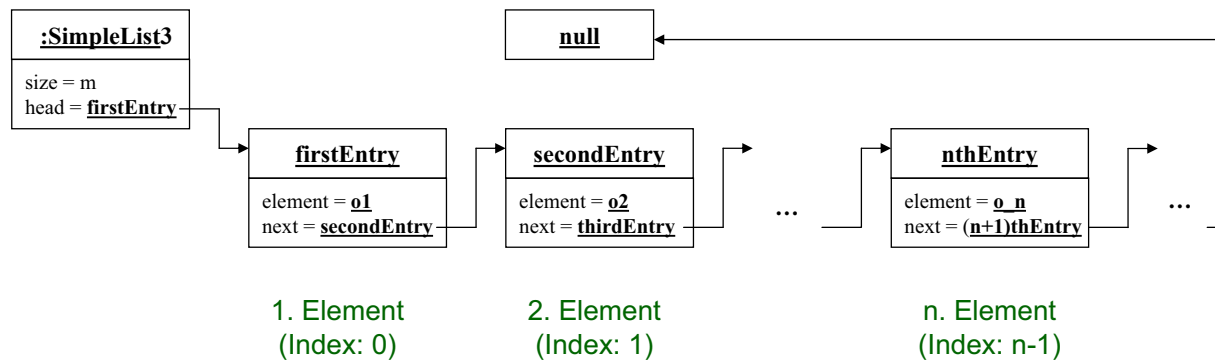
    public SimpleEntry(T o, SimpleEntry<T> next)
    {
        this.element = o;
        this.next = next;
    }
    ...
}
```

```
...
public T getElement()
{
    return this.element;
}

public SimpleEntry<T> getNext()
{
    return this.next;
}

public void setNext(SimpleEntry<T> next)
{
    this.next = next;
}
}
```

Um das n -te Element einer Liste zu bekommen, müssen die ersten $n - 1$ Elemente durchlaufen werden:

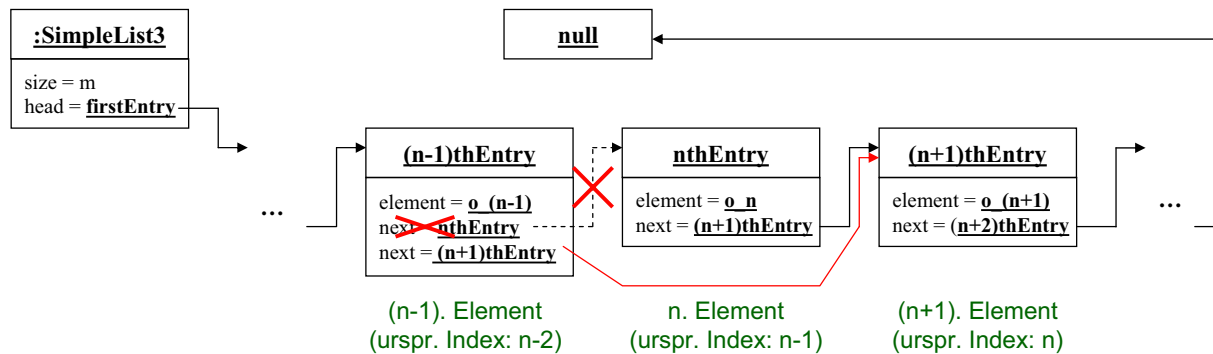


```

...
public T get(int index)
{
    if(this.head==null)
    {
        throw new NullPointerException("Empty List.");
    }
    if(index>=this.length())
    {
        throw new IllegalArgumentException(index+" exceeds length of list.");
    }
    SimpleEntry<T> currentEntry = this.head;
    while(index>0)
    {
        currentEntry = currentEntry.getNext();
        index--;
    }
    return currentEntry.getElement();
}
...

```

- Um das n -te Element aus einer Liste zu entfernen, müssen ebenfalls die ersten $n - 1$ Elemente durchlaufen werden.
- Dann muss der Verweis des $n - 1$ -ten-Elementes auf den neuen Nachfolger "umgebogen" werden:

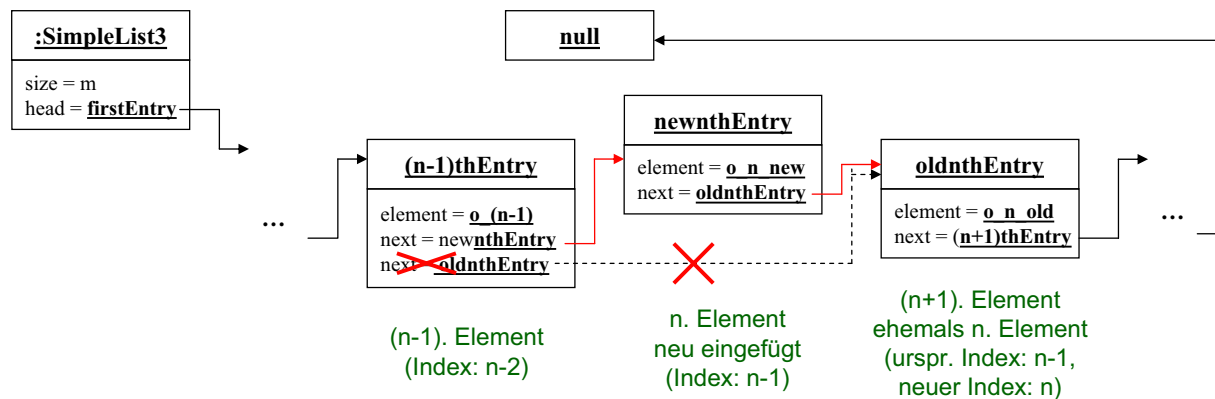


```

...
public void delete(int index)
{
    if (this.head == null)
    {
        throw new NullPointerException("Empty List.");
    }
    if (index >= this.length())
    {
        throw new IllegalArgumentException(index + " exceeds length of list.");
    }
    SimpleEntry<T> currentEntry = this.head;
    while (index > 1)
    {
        currentEntry = currentEntry.getNext();
        index--;
    }
    currentEntry.setNext(currentEntry.getNext().getNext());
    this.size--;
}
...

```

- Um ein Element an einer bestimmten Stelle n einzufügen, müssen wiederum die ersten $n - 1$ Elemente durchlaufen werden.
- Dann muss der Verweis des $n - 1$ -Elementes auf den neuen Nachfolger “umgebogen” werden.
- Zuvor brauchen wir aber den alten Verweis, weil der neue Nachfolger des $n - 1$ -ten Elements als Nachfolger den alten Nachfolger des $n - 1$ -ten Elements haben muss.



```

...
public void insert(T o, int index)
{
    if (this.head == null)
    {
        throw new NullPointerException("Empty List.");
    }
    if (index >= this.length())
    {
        throw new IllegalArgumentException(index + " exceeds length of list.");
    }
    SimpleEntry<T> currentEntry = this.head;
    while (index > 1)
    {
        currentEntry = currentEntry.getNext();
        index--;
    }
    SimpleEntry<T> newEntry = new SimpleEntry<T>(o, currentEntry.getNext());
    currentEntry.setNext(newEntry);
    this.size++;
}
...

```

Häufige Anforderung: Stelle fest, ob die Liste ein Objekt bestimmter Art enthält.

```
...
public boolean contains(T o)
{
    SimpleEntry<T> currentEntry = this.head;
    while(currentEntry!=null && !currentEntry.getElement().equals(o))
    {
        currentEntry = currentEntry.getNext();
    }
    return currentEntry!=null;
}
...
```

Um sich den aktuellen Zustand der Liste anzuschauen, überschreibt man am besten die toString-Methode in geeigneter Weise.

```
...
public String toString()
{
    StringBuilder builder = new StringBuilder();
    builder.append("[");
    for(SimpleEntry<T> entry = this.head; entry!=null; entry=entry.getNext())
    {
        builder.append(entry.getElement().toString());
        if(entry.getNext()!=null)
        {
            builder.append(", ");
        }
    }
    builder.append("]");
    return builder.toString();
}
...
```


Wenn sich die Länge einer Liste nicht mehr ändert, ist es oft praktischer, mit einem Array weiterzuarbeiten.

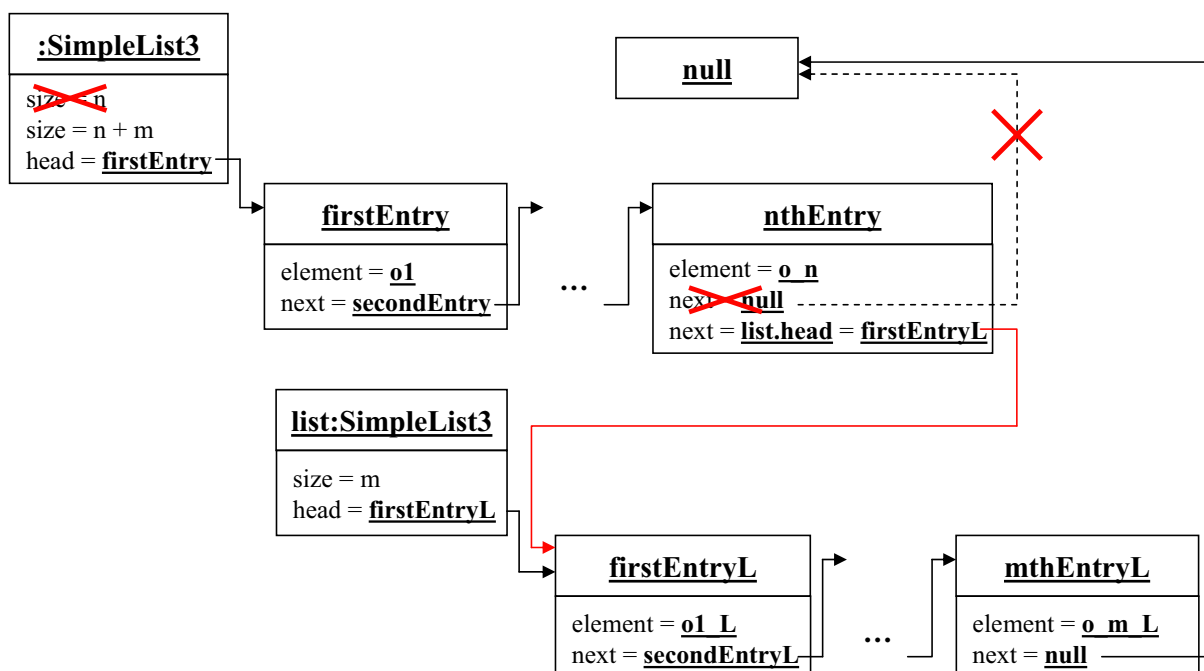
```

...
public T[] asArray()
{
    // hier wird sich der Compiler besorgt zeigen:
    // Type safety: The cast from Object[] to T[]
    // is actually checking against the erased type Object[]
    // Erinnerung: wir koennen keine generisch typisierten Arrays erzeugen
    T[] array = (T[]) new Object[this.size];
    int index = 0;
    for(SimpleEntry<T> entry = this.head; entry!=null; entry=entry.getNext())
    {
        array[index++] = entry.getElement();
    }
    return array;
}
...

```

Append: Eine Liste anhängen

- Die Methode `append` soll eine andere Liste an diese Liste anhängen.
- Das letzte Element dieser Liste soll dann nicht mehr auf `null` verweisen, sondern auf das erste Element der anderen Liste:



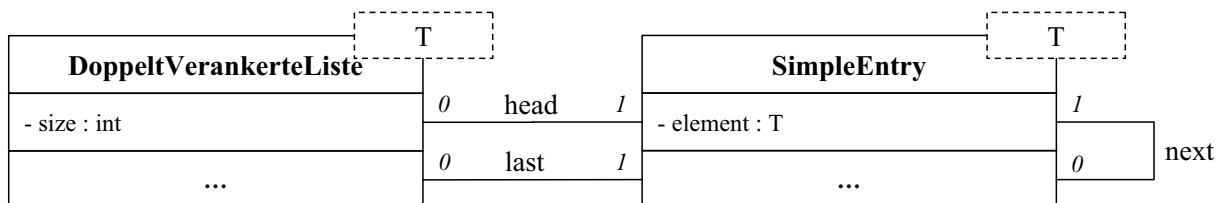
```

public void append(SimpleList3<T> list)
{
    SimpleEntry<T> last = this.head;
    while (last.getNext() != null)
    {
        last = last.getNext();
    }
    last.setNext(list.head);
    this.size += list.length();
}

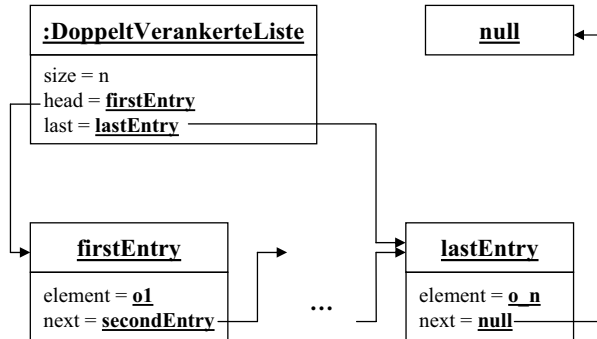
```

- In der bisherigen Implementierung muss man für die append-Methode die erste Liste ganz durchwandern ($O(n)$).
- Man kann die Liste aber auch so implementieren, dass sich der Zeitbedarf für append auf $O(1)$ reduziert.

- Lösung: halte den Verweis auf das letzte Element als Attribut der Liste:



Allgemeines Schema:



Nach Instanziierung:



```

public class DoppeltVerankerteListe<T>
{
    private int size;

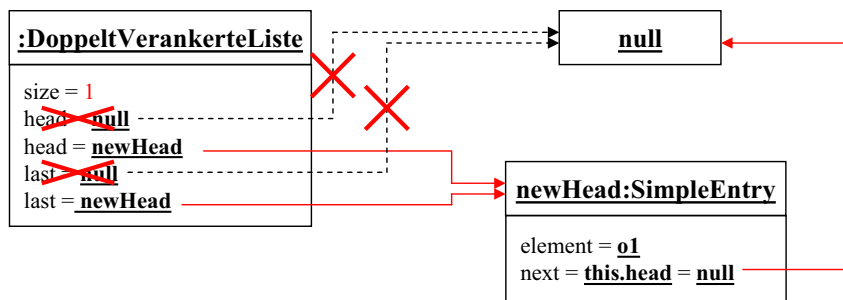
    private SimpleEntry<T> head;

    private SimpleEntry<T> last;

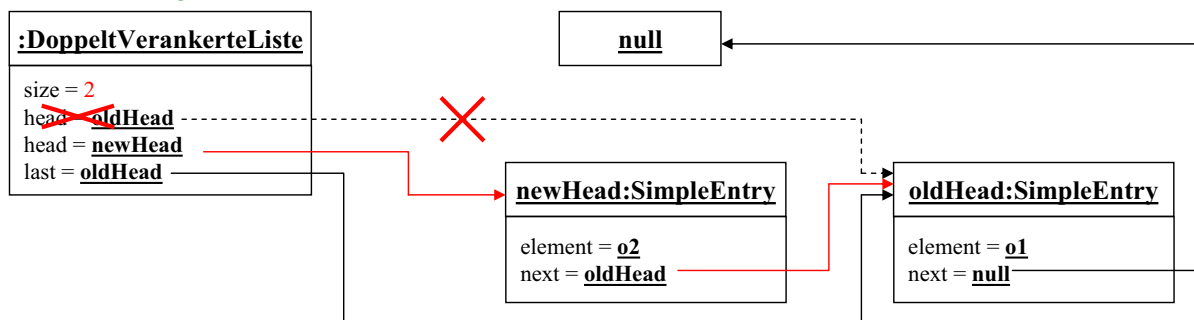
    public DoppeltVerankerteListe()
    {
        this.size = 0;
        this.head = null;
        this.last = null;
    }
    ...
}

```

Erstes Einfügen:



Zweites Einfügen:



```

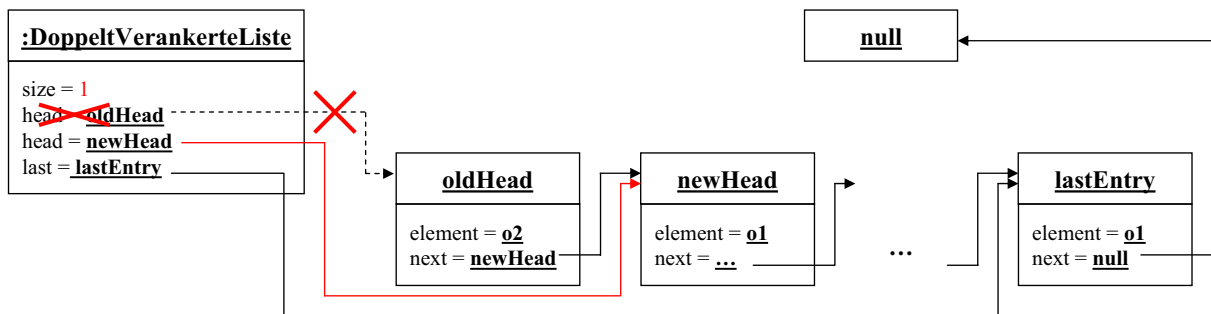
public void add(T o)
{
    SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
    this.head = newHead;
    if (this.last == null)
    {
        this.last = newHead;
    }
    this.size++;
}

```

```

public void removeHead()
{
    if (this.head == null)
    {
        throw new NullPointerException("Empty List - no head element available.");
    }
    this.head = this.head.getNext();
    if (this.head == null)
    {
        this.last = null;
    }
    this.size--;
}

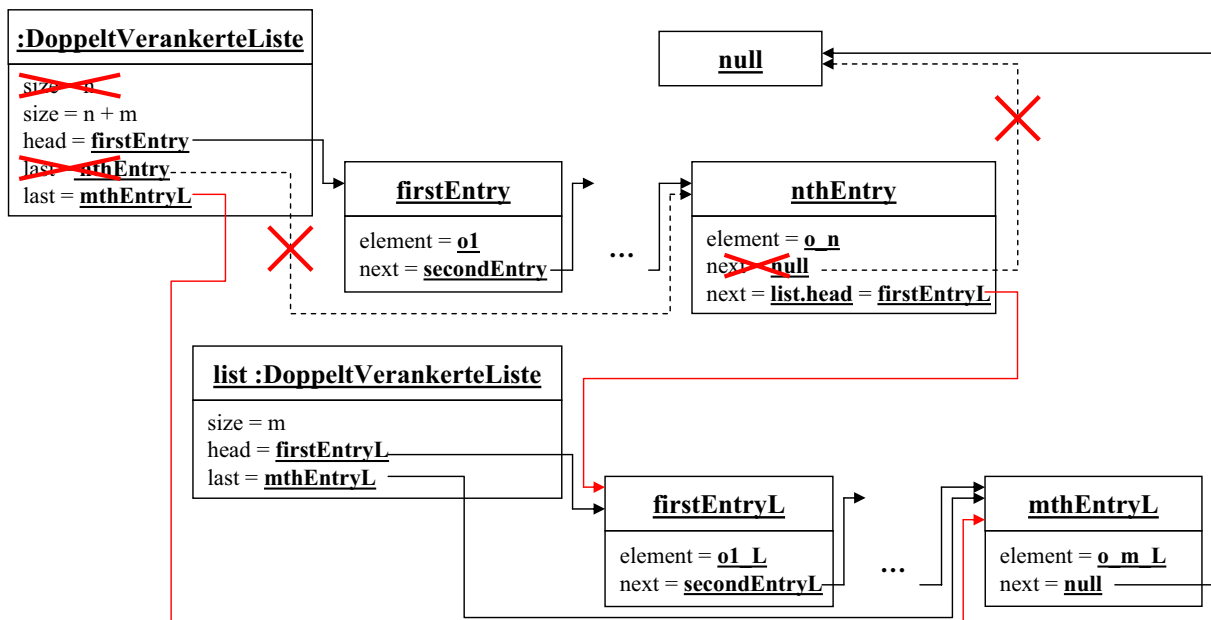
```



```

public void append(DoppeltVerankerteListe<T> list)
{
    this.last.setNext(list.head);
    this.last = list.last;
    this.size += list.length();
}

```



```

public void delete(int index)
{
    if(this.head==null)
    {
        throw new NullPointerException("Empty List.");
    }
    if(index>=this.length())
    {
        throw new IllegalArgumentException(index+" exceeds length of list.");
    }
    SimpleEntry<T> currentEntry = this.head;
    while(index>1)
    {
        currentEntry = currentEntry.getNext();
        index--;
    }
    if(currentEntry.getNext()==this.last)
    {
        this.last = currentEntry;
        this.last.setNext(null);
    }
    else
    {
        currentEntry.setNext(currentEntry.getNext().getNext());
    }
    this.size--;
}

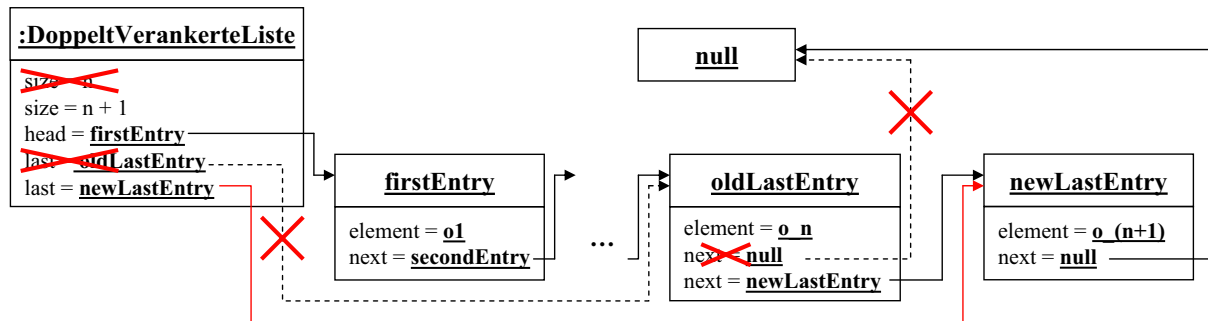
```

```

...
public void insert(T o, int index)
{
    if(this.head==null)
    {
        throw new NullPointerException("Empty List.");
    }
    if(index>=this.length())
    {
        throw new IllegalArgumentException(index+" exceeds length of list.");
    }
    SimpleEntry<T> currentEntry = this.head;
    while(index>1)
    {
        currentEntry = currentEntry.getNext();
        index--;
    }
    SimpleEntry<T> newEntry = new SimpleEntry<T>(o,currentEntry.getNext());
    if(currentEntry==this.last)
    {
        this.last=newEntry;
    }
    currentEntry.setNext(newEntry);
    this.size++;
}
...

```

- Damit haben wir gleichzeitig eine effiziente Möglichkeit gewonnen, ein Element am Ende der Liste einzufügen.

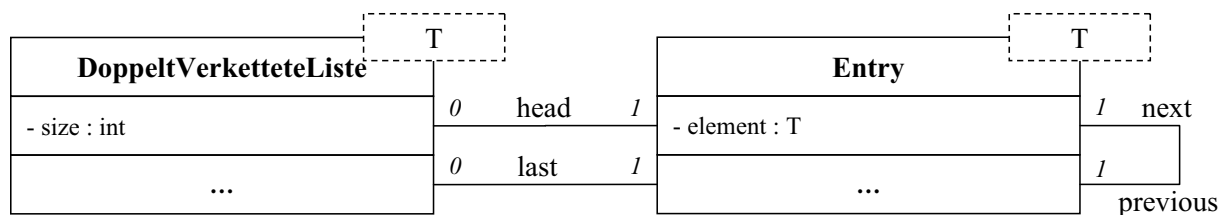


```
public void append(T o)
{
    SimpleEntry<T> newLast = new SimpleEntry<T>(o, null);
    this.last.setNext(newLast);
    this.last = newLast;
    this.size++;
}
```

- Ist auch das Entfernen am Ende der Liste möglich?

- Der Zeiger für das letzte Element muss auf das vorletzte Element “umgebogen” werden.
- In der bisherigen Implementierung müssten wir dazu wieder von vorne die Liste durchlaufen.
- Lösung: Doppelt-verkettete Liste.

- Jeder Eintrag `Entry` enthält auch einen Verweis auf seinen Vorgänger.
- Dadurch kann man die Liste auch von hinten nach vorne durchlaufen.
- Entfernen des letzten Elementes: $O(1)$.




```

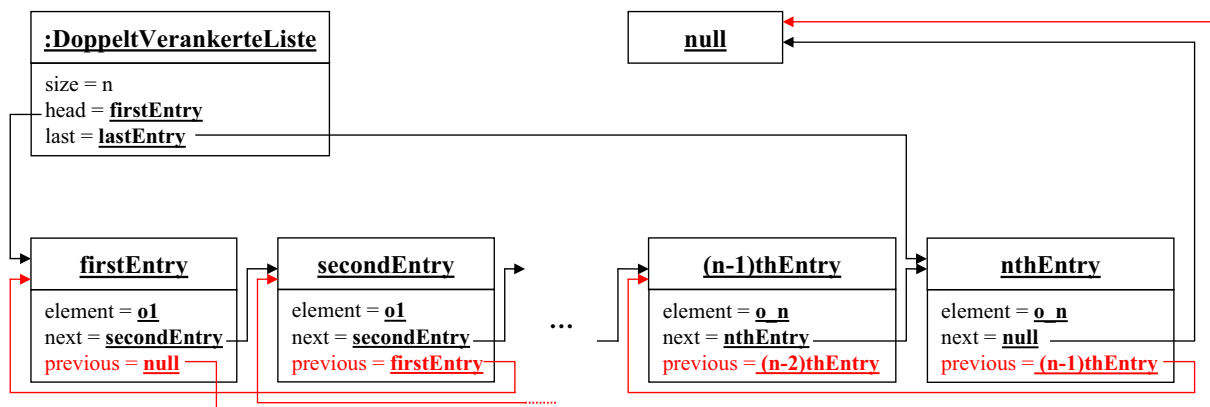
...
private static class Entry<T>
{
    private T element;

    private Entry<T> next;

    private Entry<T> previous;

    public Entry(T o, Entry<T> next, Entry<T> previous)
    {
        this.element = o;
        this.next = next;
        this.previous = previous;
    }
    ...
}

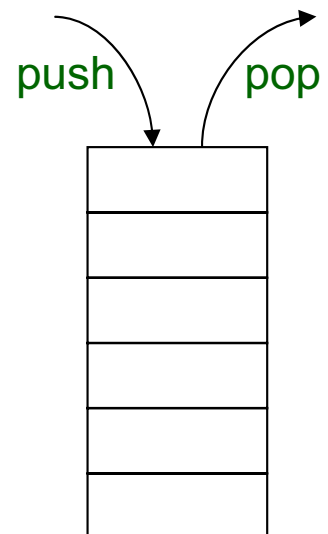
```



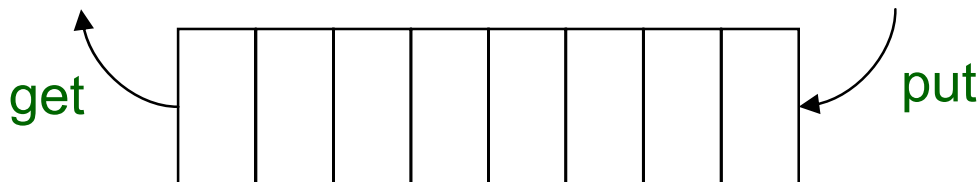
- Übung: Welche Änderungen werden in den bisherigen Methoden dadurch nötig?

- Im imperativen Paradigma wächst eine Liste normalerweise nicht nach vorne, sondern nach hinten.
- Die einfache Anfüge-Operation hängt ein neues Element hinten an.
- Einfügen und Löschen ist auch an beliebiger Stelle möglich.
- Standard-Implementierungen einer verketteten Liste sind oft doppelt-verkettete Listen (z.B. `java.util.LinkedList`).

- Den Keller als LIFO-Datenstruktur haben wir bereits kennengelernt.
- Wir wollen einen Kellerspeicher mit zwei Operationen:
 - **void** `push(o)` – legt Objekt `o` auf dem Stapel ab.
 - `E pop()` – entfernt oberstes Element (vom Typ `E`) und gibt es zurück.
- Welche Mittel benötigen wir *minimal* zur Implementierung:
 - Einfach verkettete Liste?
 - Doppelt verankerte Liste?
 - Doppelt verkettete Liste?



- Die Warteschlange (Queue) ist eine häufig benötigte FIFO-Datenstruktur.
- Typischerweise zwei Operationen:
 - `void put (o)` – fügt das Objekt `o` an die Schlange an.
 - `E get ()` – entfernt vorderstes Element (vom Typ `E`) und gibt es zurück.



- Welche Mittel sind zur Implementierung am besten geeignet:
 - Einfach verkettete Liste?
 - Doppelt verankerte Liste?
 - Doppelt verkettete Liste?

Flexible Datenstrukturen mit effizientem wahlfreiem Zugriff

- Den Vorteil der flexiblen Länge haben wir in den bisherigen Implementierungen dadurch erkaufte, dass der Zugriff auf das n -te Element eine Zeitkomplexität von $O(n)$ hat.
- Andere Implementierungen verwirklichen die flexible Länge durch ein internes Array.
- Wird das interne Array zu kurz, wird es durch ein längeres ersetzt und der Inhalt des alten Arrays in das neue kopiert.
- Beispiele für diese Implementierung kennen Sie bereits mit `StringBuilder` und `StringBuffer`.
- Allgemeine, array-basierte Listen-Implementierungen: `java.util.ArrayList` und `java.util.Vector`.
- Der Vorteil durch Zeiteffizienten wahlfreien Zugriff ($O(1)$) wird erkaufte durch höheren Speicherplatzbedarf und gelegentlichen Kopieraufwand beim Wachsen der Liste.
- Verkettete Listen sind also dann vorteilhaft, wenn nur am Anfang und/oder am Ende Änderungsoperationen (Wachsen/Schrumpfen) auftreten.