Konkatenation zweier Listen mit *concat*

- Was ist an der Konkatenation etwas unschön?
- Man muss die „vordere“ Liste einmal durchgehen, um den „letzten“ Nachfolger (urspr. **`null`** zu erhalten, der einen neuen Nachfolger bekommt (das erste Objekt der „hinteren“ Liste).
- Wie könnte man das verbessern?
- Die bisherige Liste nennt man *einfach verankerte, einfach verkettete Liste*.
- Man könnte die Liste *zweifach verankern*.
- Das geht einfach: in der Klasse `List<T>` gibt es ein zusätzliches Attribute `last` vom Typ `Entry<T>`, das den letzten Eintrag der Liste referenziert.

i-tes Element entfernen

- Nehmen wir an, wir wollen auch zusätzlich das i-te Element aus der Liste entfernen.
- Um das i-te Element aus einer Liste zu entfernen, müssen natürlich die ersten $i-1$ Elemente durchlaufen werden.
- Dann muss der Verweis des $i-1$-ten-Elementes auf den neuen Nachfolger “umgebogen” werden:

i-tes Element entfernen

```
...
public void delete(int index)
{
  if(this.head==null) {
    throw new NullPointerException("Empty List.");
  }
  if(index<1 || index>this.length()) {
    throw new IllegalArgumentException(index+" out of list index range.");
  }

  Entry<T> currentEntry = this.head;
  for(int j = 1; j<=index; j++) {
    currentEntry = currentEntry.getNext();
  }
  currentEntry.setNext(currentEntry.getNext().getNext());
  this.size--;
}
...
```

Element an Stelle i einfügen

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

Element an Stelle n einfügen

```
...
public void insert(T o, int index)
{
  if(index < 1 || index > this.length()) {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  if(this.head==null) {
    this.head = new Entry<T>(o,null);
  }

  else {
    Entry<T> currentEntry = this.head;
    while(index > 0) {
      currentEntry = currentEntry.getNext();
      index--;
    }
    Entry<T> newEntry = new Entry<T>(o,currentEntry.getNext());
    currentEntry.setNext(newEntry);
  }
  this.size++;
}
```

Suche nach einem Element

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

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

Effizienzverbesserung: Doppelt-verankerte Liste

Hier noch kurz angedeutet: die Verbesserung durch die doppelte Verankerung:

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

 private Entry<T> head;

 private Entry<T> last;

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

Symmetrische Variante

- Wie wir kurz angedeutet haben, können wir die Liste statt mit `prefix` auch mit `postfix` aufbauen und dann auf das letzte Element `last` und den vorderen Rest `lead`
- Wir können das natürlich mit einer doppelt-verankerten Liste genauso wie mit einer einfach verketteten Liste machen.
- In der Klasse `Entry<T>` wird statt der Nachfolger (Attribut `next`) nun der Vorgänger `prev` gespeichert.
- Die drei oben angedeuteten Methoden sind dann symmetrische Varianten der Methoden `prefix`, `first` und `rest`.

Anwendungs-Beispiel

- Die symmetrische Variante in Action:
- Ein Polynom P n-ten Grades ist eine Abbildung $P : \mathbb{R}^{n+2} \to \mathbb{R}$ mit

$$P(a_n, a_{n-1}, \ldots, a_1, a_0, x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

  d.h. für $n = 0$ ist der Wert von P der Wert von a_0.
- Modellieren wir die „Koeffizienten“ $a_n, \ldots a_0$ als Liste a über `Double`-Objekten (d.h. `T` wird mit `Double` zu `List<Double>` parametrisiert), können wir das Berechnungsschema direkt implementieren (müssen dabei aber die Liste von hinten durchlaufen):

Anwendungs-Beispiel

```
public class Polynom {
   public static Double polynomBerechnen(List<Double> a, double x) {
      if(a.isEmpty()) {
         throw new IllegalArgumentException("Coefficients must not be empty.");
      }

      if(a.lead().isEmpty()) {
         return a.last().getElement();
      } else {
         return polynomBerechnen(a.lead(), x*x) * x + a.last()
      }
   }

...
```

Überblick

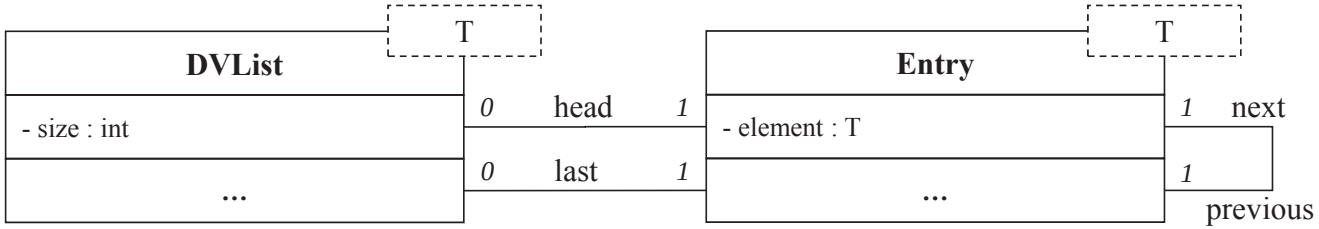
10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen
10.3 Doppelt-verkettete Listen
10.4 Bäume

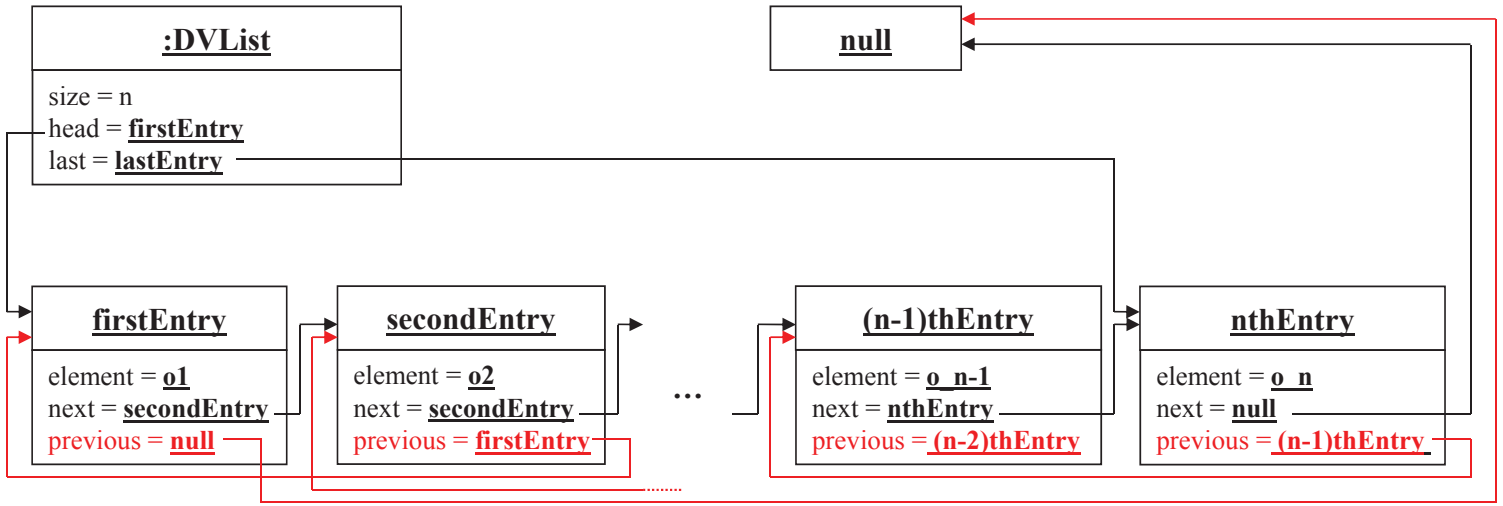
Verbesserung?

- Was ist eigentlich, wenn wir tatsächlich beide symmetrische Varianten gleichzeitig haben wollen, also sowohl `prefix`, `first` und `rest`, als auch `postfix`, `last` und `lead` haben wollen?
- Dann müssten wir beide Varianten in einem implementieren . . .
- Das nennt man dann *doppelt-verkettete Liste* bzw. doppelt-verkettet und doppelt-verankerte Liste.

Doppelt-verkettete Liste

Graphisch:





Doppelt-verkettete Liste: Entry

```
private class Entry<T> {
  private T element;

  private Entry<T> next;

  private Entry<T> prev;

  public Entry(T elem) {
    this.element = elem;
    this.next = null;
    this.prev = null;
  }
  ... // Getter and Setter
}
```

Doppelt-verkettete Liste: Liste

```
public class DVList<T> {

   private int size;

   private Entry<T> head;

   private Entry<T> tail;

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

   ...
```

Doppelt-verkettete Liste: *prefix*

```
...
public void prefix(T elem) {
   Entry<T> newHead = new Entry<T>(elem);
   if(this.size == 0) {  // leere Liste: beachte tail!
      this.head = newHead;
      this.tail = newHead;
   } else {  // Liste nicht mehr leer: tail egal!
      newHead.setNext(this.head);
      this.head.setPrev(newHead);
      this.head = newHead;
   }
   this.size++;
}
...
```

Doppelt-verkettete Liste: *postfix*

```
...
public void postfix(T elem) {
   Entry<T> newTail = new Entry<T>(elem);
   if(this.size == 0) {  // leere Liste: beachte head!
      this.head = newTail;
      this.tail = newTail;
   } else {  // Liste nicht mehr leer: head egal!
      newTail.setPrev(this.tail);
      this.tail.setNext(newTail);
      this.tail = newTail;
   }
   this.size++;
}
...
```

Doppelt-verkettete Liste: *first* und *last*

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

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

Doppelt-verkettete Liste: *rest*

```
  ...
  public DVList<T> rest() {
    if(this.head==null) {
       throw new NullPointerException("Empty List - no head element to cut off.");
    }
    DVList<T> erg = new DVList<T>();
    erg.size = this.size - 1;
    erg.head = this.head.getNext();
    return erg;
  }
  ...
}
```

Doppelt-verkettete Liste: *lead*

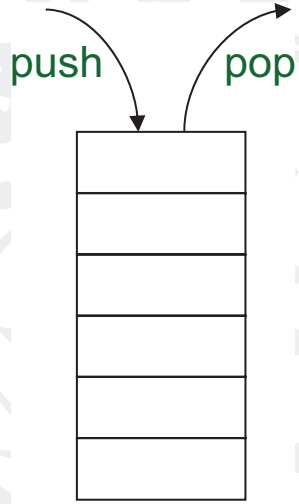
```
  ...
  public DVList<T> lead() {
    if(this.tail==null) {
       throw new NullPointerException("Empty List - no last element to cut off.");
    }
    DVList<T> erg = new DVList<T>();
    erg.size = this.size - 1;
    erg.tail = this.tail.getPrev();
    return erg;
  }
  ...
}
```

Doppelt-verkettete Liste als Standard

- Die weiteren Methoden sind recht ähnlich zur einfach verketteten Liste.
- Standard-Implementierungen einer (verketteten) Liste in imperativen Sprachen sind oft doppelt-verkettete Listen (z.B. `java.util.LinkedList`).

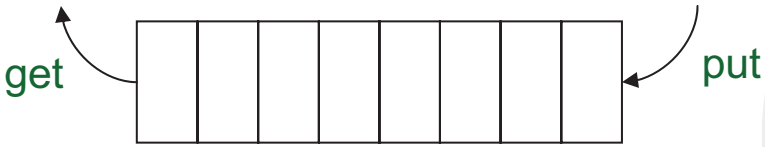
Beispiel: Keller (Stapel, Stack)

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



Beispiel: Warteschlange (Queue)

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



- Welches Listenmodell ist zur Implementierung am besten geeignet?

Beispiel: Spiegeln einer Queue

- Rekursiv lässt sich eine Queue sehr einfach spiegeln:
- Rekursionsanfang: wenn die Liste leer ist, ist sie automatisch gespiegelt.
- Ansonsten: Spiegele den Rest (ohne das erste Element) und hänge dieses erste Element am Ende an.