Einführung in die Programmierung

Teil 11: Datenstrukturen

Prof. Dr. Peer Kröger,
Florian Richter, Michael Fromm

Wintersemester 2018/2019

Übersicht

Was sind Datenstrukturen?

- Viele Computer-Programme sind in erster Linie dazu da, Daten zu verarbeiten.
- Eine Datenmenge muss dazu intern durch eine *Datenstruktur* organisiert und verwaltet werden.
- Als einfache Datenstruktur zur Verwaltung gleichartiger Elemente haben wir für imperative Sprachen z.B. das Array kennengelernt.
- Im ersten Teil der Vorlesung haben wir bei den Wechselgeldalgorithmen *Folgen* als Datenstruktur verwendet.
- Ein Äquivalent zum mathematischen Konzept der Folge findet sich in vielen Programmiersprachen als *Liste*.

Was sind Datenstrukturen?

- Als spezielle Datenstruktur können wir auch die Strings (und verwandte Klassen) betrachten, die für eine Menge von Zeichen stehen.
- Auch eine Klasse dient zunächst der Darstellung von Objekten, die einen Ausschnitt der Wirklichkeit abstrahiert repräsentieren; hier können theoretisch sogar verschiedenartige Elemente verwaltet werden.

Die Bedeutung von Datenstrukturen

- 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.
- Manche Datenstrukturen sind dynamisch (veränderbar), andere statisch (nicht veränderbar).
- Die Auswahlmöglichkeiten für Algorithmus und Datenstruktur sind eng miteinander verflochten. Durch eine geeignete Wahl möchte man Zeit und Platz sparen.

Datenstrukturen als Objekte

- Eine Datenstruktur können wir auch wieder als Objekt auffassen und mit einer Klasse entsprechend modellieren.
- Das bedeutet, dass eine Datenstruktur Eigenschaften und Fähigkeiten hat, also z.B. typische Operationen ausführen kann.
- Eine Datenstruktur hat also auch wieder eine *Schnittstelle*, die angibt, wie man sie verwenden kann.
- Das oo Paradigma (insbesondere die Aspekte Abstraktion und Kapselung) eignet sich bestens, um eigene Datenstrukturen (durch Klassen) als abstrakten Datentypen zu entwickeln.

Ziel dieses Kapitels

- In diesem Kapitel wollen wir Datenstrukturen für Listen besprechen.
- Obwohl es eigentlich alle Varianten, die wir entwickeln werden, bereits in der Java-API (im sog. Collections-Framework) gibt, dient dieses Kapitel zwei Aspekten:
  1. Wir besprechen den internen Aufbau von verscheidenen Listen-Modellen und diskutieren ihre Vor- und Nachteilen; das ist insb. wichtig, da Sie, wenn Sie Java API Klassen verwenden, sich darüber im Klaren sein müssen, was die verwendeten Klassen für Eigenschaften haben.
  2. Wir sehen auch nochmal die Kernideen der oo Programmierparadigma in Aktion:
     Sowohl die oo Modellierung der Listen, als auch deren Verwendung sind gute Beispiele dafür.

Arrays vs. Listen

- Wie wir gesehen haben, erlauben Arrays effizient den sogenannten *wahlfreien Zugriff*, d.h. wir können auf ein beliebiges Element direkt zugreifen, wenn wir dessen Stelle im Array kennen.
- Bei der Spezifikation von Folgen gilt das nicht:
  Der Zugriff auf das n-te Element erfordert, vom Beginn der Folge alle $n-1$ Elemente *abzugehen*, bis man an der entsprechenden Stelle angekommen ist.
  Zur Erinnerung: Definition der Projektion

$$\pi(x,i) = \begin{cases} first(x), & \text{falls } i = 1, \\ \pi(rest(x), i-1) & \text{sonst.} \end{cases}$$

Arrays vs. Listen

- Dafür können Folgen (und entsprechendene Umsetzungen als Listen) beliebig wachsen, während wir die Größe eines Arrays von vornherein festlegen müssen.
  Zur Erinnerung: Folgen über der Menge M wurden (induktiv) von vorne (mit *postfix*) bzw. von hinten (mit *prefix*) aus einer leeren Folge $()$ aufgebaut:
  1. $() \in M^*$
  2. Ist $x \in M^*$ und $a \in M$, dann ist $\mathit{postfix}(x, a) \in M^*$.

  bzw. alternativ
  1. $() \in M^*$
  2. Ist $a \in M$ und $x \in M^*$, dann ist $\mathit{prefix}(a, x) \in M^*$.

Kapitel 2: Ein Datenmodell für Listen i

Ein einfaches Listenmodell

- Bevor es mit der oo Modellierung einer Liste los geht, sollten wir uns zunächst überlegen, welche Schnittstelle, d.h. welche Methoden, die Liste haben sollte.
- Dabei orientieren wir uns an den Funktionen, die wir für Folgen spezifiziert haben.
- Eine Liste kann außerdem leer sein, d.h. wir könnten zusätzlich eine Methode *isEmpty* $\mapsto \mathbb{B}$ zur Verfügung stellen.
- Eine Liste sollte möglicherweise Auskunft über ihre Länge geben können, d.h. eine Methode *size* $\mapsto \mathbb{N}_0$ wäre sinnvoll.

Schnittstelle für die Liste-Klasse

In Summe hätten wir folgende Operationen:

- Ein Element wird vorne an eine Liste angehängt (Operation *prefix*).
- Alternativ: ein Element wird hinten an eine Liste angehängt (Operation *postfix*).
- Konkatenation zweier Listen (Operation *concat*).
- Zugriff auf das erste Element (Operation *first*).
- Alternativ: Zugriff auf das letzte Element (Operation *last*).
- Liste nach Entfernen des ersten Elementes (Operation *rest*).
- Alternativ: Liste nach Entfernen des letzten Elementes (Operation *lead*).
- Operationen *isEmpty* und *size* und die Projektion *proj*.

Erster Versuch mit Arrays

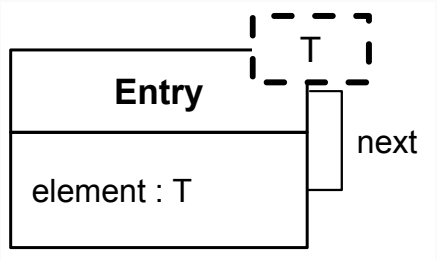
- Von der Objektorientierung herkommend, wollen wir Folgen als Objekte modellieren, die eine Liste von Elementen dynamisch verwaltet.
- Wir konzentrieren uns zunächst auf die Alternative, die Liste „vorne“ zu erweitern (von hinten aufzubauen), also mit den Operationen *first*, *rest* und *prefix*.
- Eine erste Idee, in der Liste intern ein Array für die einzelnen Elemente zu halten, ist vermutlich nicht so clever.
- Wir müssten beim Wachsen oder Schrumpfen der Liste immer neue Arrays erzeugen und tiefe Kopien anfertigen, was sehr aufwändig zu programmieren und auszuführen ist.

Zweiter Versuch: oo Modellierung

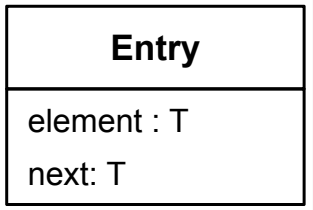
- Gehen wir objektorientiert vor: was sind die Objekte, die interagieren (und die wir modellieren müssen)?
- Eine Liste ist ein Objekt und eine ihrer wesentlichen Eigenschaften ist, dass sie eine Menge von Elementen verwaltet.
- Diese Elemente der Liste könnten wir natürlich auch als *eigenständige* Objekte ansehen und entsprechend modellieren.

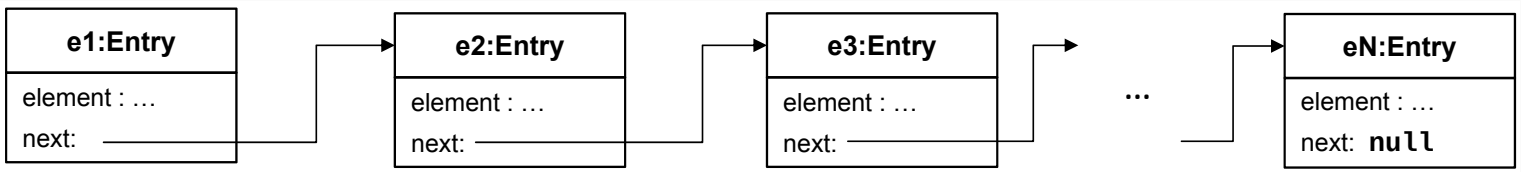
Element der Liste als besonderes Objekt

- Welche Eigenschaften hat ein Element der Liste (als eigenständiges Objekt modelliert)?
- Dieses Objekt hält zunächst das eigentlich gespeicherte Element (am besten mit einem generischen Typ!).)
- Dieses Objekt steht in Beziehung z.B. zum nächsten Element der Liste, d.h. es kennt einen *Verweis* auf das nächste Element, den Nachfolger:



bzw. entspricht



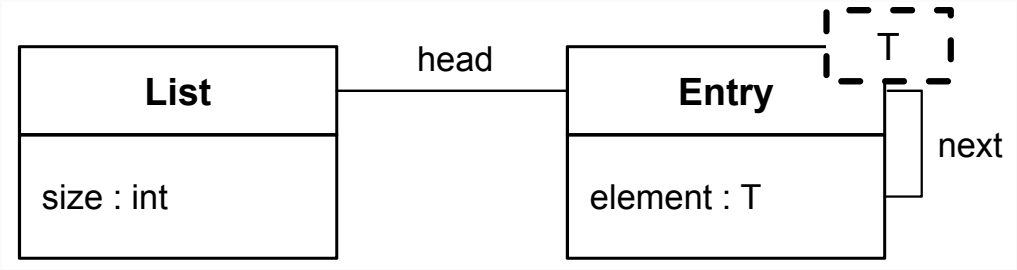


Die Liste selbst

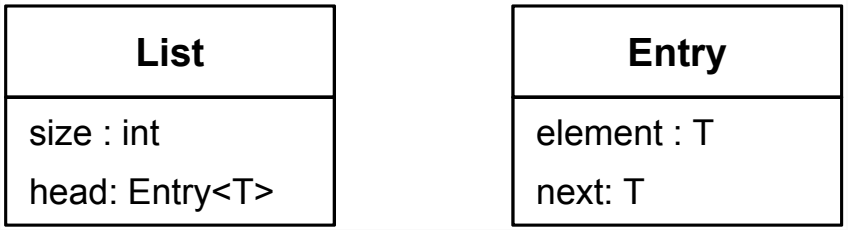
- Die Liste selbst muss nur das erste Element kennen, über dessen Verweise zum nächsten Element (Attribut `next`) können alle Nachfolger sequentiell erreicht werden.
- Dies entspricht auch der „Implementierung der Projektion“ von oben.
- Die eigentliche Liste benötigt also nur einen Verweis auf das erste Element, d.h. die Klasse, die die Liste modelliert, hat als Attribute ein Objekt vom Typ `Entry<T>`.
- In der leeren Liste ist dieser Verweis leer, d.h. das Element ist **`null`**.

Die Liste selbst

- Zusätzlich speichert die Liste noch die Anzahl der Elemente in einem entsprechenden Attribut (um die Operation *size* effizient umzusetzen)
- In UML:



bzw.



Implementierung in Java

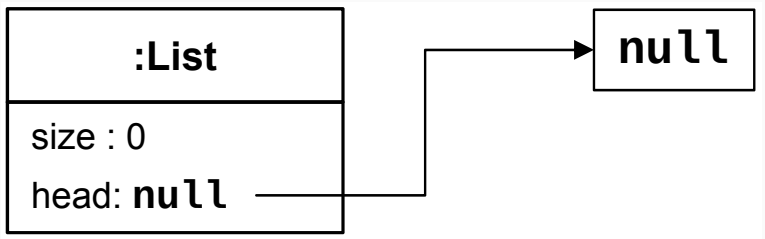
```
public class Entry<T> {
  private T element;          // das eigentliche Element vom Typ T
  private Entry<T> next;      // der Verweis auf das Nachfolgeelement

  public Entry(T elem) {
    this.element = elem;
    this.next = null;
  }

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

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

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

Implementierung in Java

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

  private Entry<T> head;

  public List() {      // leere Liste wird angelegt
    this.size = 0;
    this.head = null;
  }

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

  ...
}
```

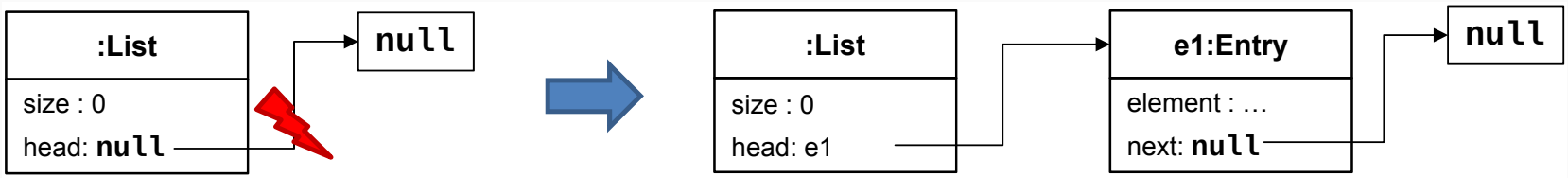
Erzeugen

- Veranschaulichung: der Konstruktors `List()` erzeugt eine leere Liste:



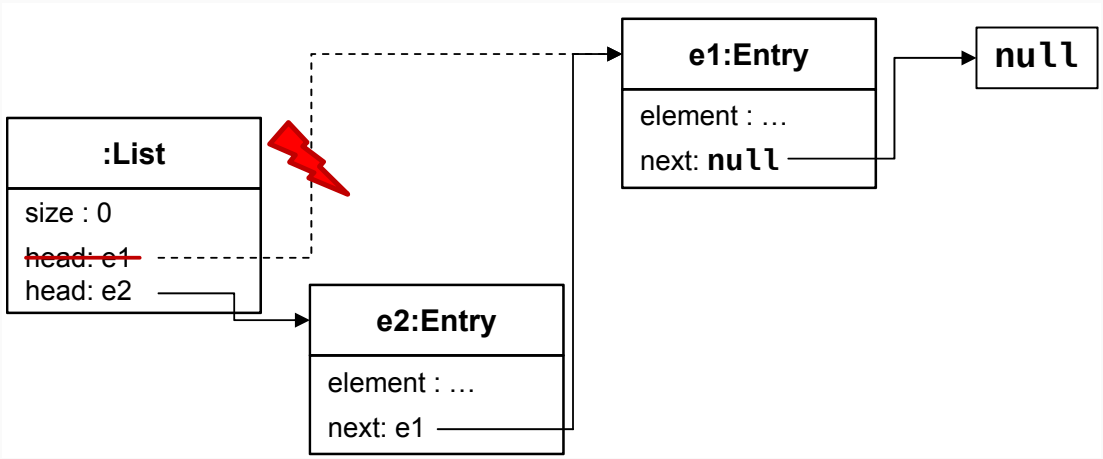
Hinzufügen mit *prefix*

- Die Operation *prefix* fügt ein neues Element „vorne“ an die Liste an.
- Es muss ein neues `Entry`-Element erzeugt werden und als neues erstes Element gesetzt werden („vorne“ anhängen).
- Dessen Nachfolger ist das alte erste Element, d.h. das neue Element bekommt dieses als Nachfolger.
- Die Länge der Liste erhöht sich um 1:



Hinzufügen

- Und so wächst die Liste weiter:



In Java:

```
...
public void prefix(Entry<T> newHead) {
{
  newHead.setNext(this.head);
  this.head = newHead;
  this.size++;
}
...
```

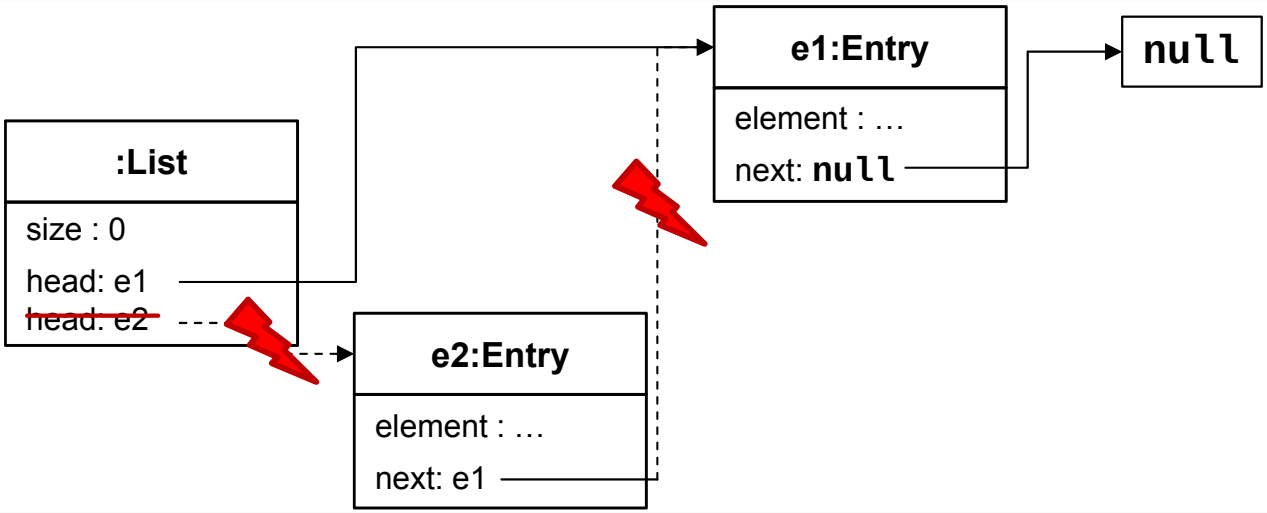
Zugriff auf das erste Element mit *first*

- Nur den Wert des ersten Elementes zu bekommen, ist einfach, aber die Liste könnte leer sein.
- Wie abfangen? Natürlich: z.B. mit einer Exception!

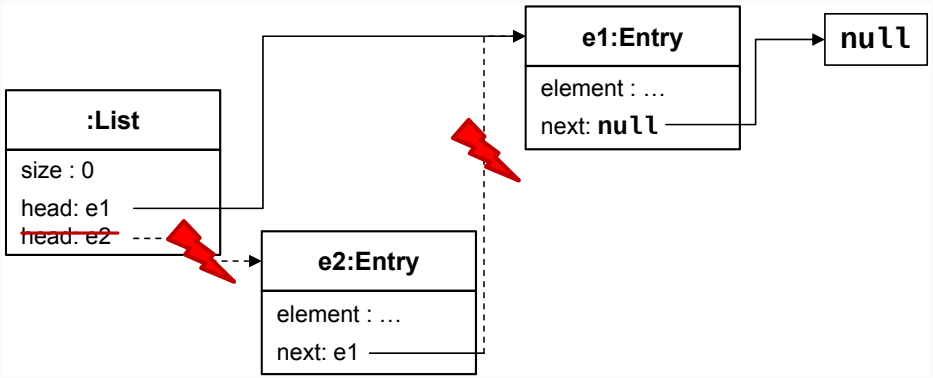
```
public T first() {
  if(this.head==null) {
    throw new
       NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}
```

Entfernen des ersten Elementes mit *rest*

- Um das erste Element zu entfernen, muss der Head-Zeiger der Liste auf den Nachfolger des ersten Elementes zeigen.
- Die Ergebnisliste enthält ein Element weniger.
- Achtung wieder bei leeren Listen!!!



Liste: Entfernen des ersten Elementes




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

Zugriff auf das *i*-te Element mit *proj*

- Zur Erinnerung: Definition der Projektion auf das *i*-te Element einer Liste *x* kann wie folgt „berechnet“ werden:

$$proj(x,i) = \begin{cases} first(x), & \text{falls } i = 1, \\ \pi(rest(x), i-1) & \text{sonst.} \end{cases}$$

- Diese rekursive Funktion können wir 1-zu-1 so als Algorithmus umsetzen.

Zugriff auf das *i*-te Element mit *proj*

```
...
public T proj(int index) {
  if(this.head == null) {
    throw new NullPointerException("Empty List.");
  }
  if(index >= this.size()) {
    throw new IllegalArgumentException(index+" exceeds size of list.");
  }

  // Hier der eigentliche Algorithmus

  if(index == 1) {
     return this.head.getElement();
  } else {
     return this.rest().proj(index - 1);
  }
}
...
```

Überprüfung der leeren Listen mit *isEmpty*

```
...
public boolean isEmpty() {
  return this.head==null;
}
...
```

Das war leicht, aber jetzt fehlt uns noch die Konkatenation . . .

Konkatenation zweier Listen mit *concat*

Idee: siehe Tafel :-)

```
...
public void concat(List<T> list) {

  // siehe Uebung

}
...
```

Übrigens: was müsste man beachten, wenn der Ergebnistyp `List<T>` wäre?

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, daher verwenden die symmetrische Variante mit `postfix`):

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()
      }
   }

 ...
```

Bemerkung: tiefe Kopien

- Grundsätzlich haben wir uns bisher wieder keine Gedanken darüber gemacht, dass wir bei Gettern, Settern und Konstruktoren Referenzen direkt übergeben, statt tiefe Kopien anzufertigen.
- Beispiel beim Getter von `Entry<T>`:

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

- Eine Möglichkeit wäre, die Typvariable `T` vom Interface `Cloneable` abzuleiten und überall wo benötigt die Methode `clone()` zu verwenden (damit ist es ggfls. in der Verantwortung des Verwenders, diese Methode entsprechend zur Verfügung zu stellen).

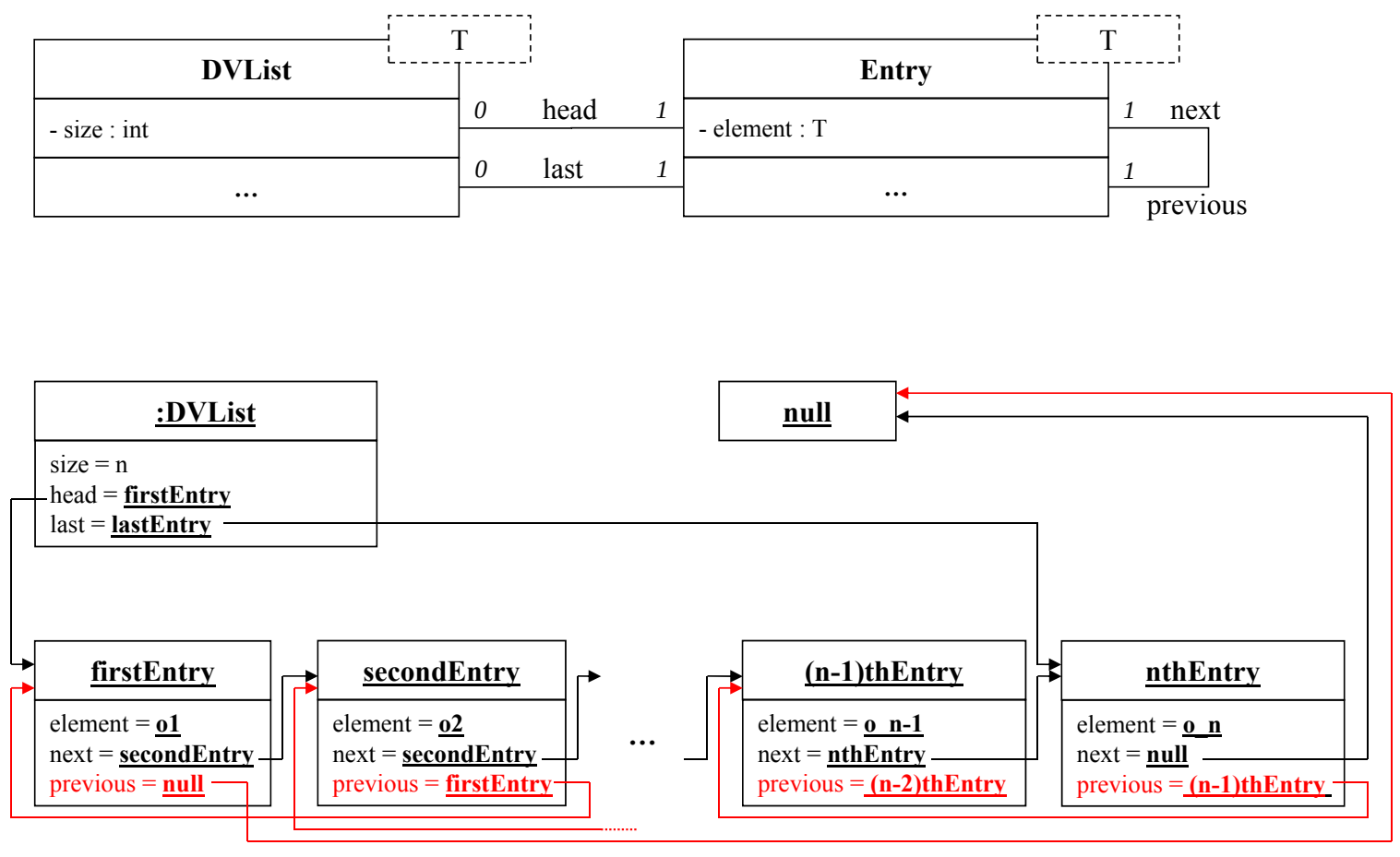
Kapitel 3: Doppelt-verkettete Listen i

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 . . .
- Können wir natürlich machen, das Resultat 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*

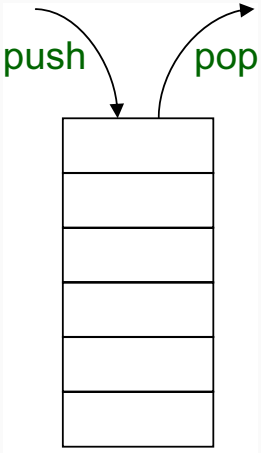
```
  ...
  public DVList<T> lead() {
    if(this.tail==null) {
       throw new
          NullPointerException("Empty List - no last element to cut off.");
    }
    DVList<T> erg = new DVList<T>();
    erg.size = this.size - 1;
    erg.tail = this.tail.getPrev();
    return erg;
  }
  ...
}
```

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 (in Java ist dies z.B. die Klasse `java.util.LinkedList`).

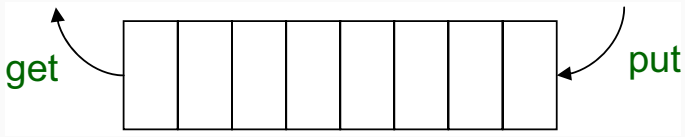
Beispiel: Keller (Stapel, Stack)

- 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 (entspricht `prefix`).
  - `T top()` – gibt das oberstes Element (vom Typ `T`) zurück (entspricht `first`).
  - `T pop()` – entfernt oberstes Element (entspricht `rest`).
    Anmerkung: in vielen Implementierungen entfernt `pop` auch zusätzlich das oberste Element (Seiteneffekt!).
- Welches Listenmodell ist zur Implementierung am besten geeignet?



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`)
  - `T rest()` – entfernt vorderstes Element (entspricht `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 (mit `put`).
- Iterativ geht es z.B. mit Hilfe eines Stacks.

Spiegeln einer Liste mit Keller

```
    public DVList<T> spiegeln() {
    Stapel<T> stack = new Stapel<T>();
    int max = this.size();
    for(int i=1; i<=max; i++) {
       stack.push(this.proj(i));
    }

    DVList<T> erg = new DVList<T>();
    for(int i=1; i<=max; i++) {
       erg.postfix(stack.top());
       stack.pop();
    }
    return erg;
}
```

Lokale Klassen

- Eigentlich muss man die Klasse `Entry<T>` gar nicht unbedingt kennen, wenn man einen Stack oder eine Queue (oder auch eine Liste) benutzt.
- Wir hatten bereits bei den Record-Typen gesehen, dass es in Java die Möglichkeit gibt, innerhalb einer Klassendefinition weitere Klassen zu definieren, sog. *lokale* oder *eingebettete* Klassen.
- Die lokalen Klassen haben Zugriff auf alle Komponenten (auch der **`private`** Komponenten) der Klasse, in die sie eingebettet sind.

Keller mit lokaler Entry-Klasse

```
public class Keller<T> {

private Entry first;

// Eingebettete Klasse:

private class Entry {
   Entry next;
   T element;

   Entry(T element) {
      this.next = null;
      this.element = element;
   }
}

// Die eigentlichen Methoden von Keller (naechste Seite):
...
```

Keller mit lokaler Entry-Klasse

```
    ...
    public Keller() {
        this.first = null;
    }

    public boolean isEmpty() {
        return (this.first == null);
    }

    public void push(T element) {
        Entry newEntry = new Entry(element);
        newEntry.next = this.first;
        this.first = newEntry;
    }

    public T pop() {
        T erg = this.first.element;
        this.first = this.first.next;
        return erg;
    }
}
```

Lokale Klassen

- Die Schnittstelle der Klasse `Keller` besteht nur aus folgenden Methoden (zzgl. des Konstruktors):

`boolean isEmpty()`	Ist der Keller leer?
`void push(T element)`	Legt ein Element auf dem Keller ab.
`T pop()`	Entfernt das oberste Objekt vom Keller und gibt es zurück.

- Die Klasse `Entry` ist nach außen hin nicht sichtbar.
- Wegen der Abhängigkeit der Referenzen (tiefe Kopie!!!) gelten die o.g. Überlegungen.

Schlange mit lokaler Entry-Klasse

- Und dasselbe auch mit der Schlange:

```
public class Schlange<T> {

private Entry first;

private Entry last;

// Eingebettete Klasse:

private class Entry {
   Entry next;
   T element;
}

// Die eigentlichen Methoden von Schlange (naechste Seite):
...
```

- Aufgemerkt: Der Konstruktor von `Entry` fehlt hier.

Schlange mit lokaler Entry-Klasse

```
    ...
    // Auch hier sparen wir uns den Konstruktor.

    public boolean isEmpty() {
       return (this.first == null);
    }

    public void put(T item) {
       Entry newEntry = new Entry();
       newEntry.element = item;
       if(this.first == null) {
          this.first = newEntry;
          this.last = newEntry;
       } else {
       this.last.next = newEntry;
       this.last = newEntry;
    }

    public T get() {
       T erg = this.first.element;
       this.first = this.first.next;
       if(this.isEmpty()) {
          this.last = null;
       }
       return erg;
    }
}
```

Diskussion

- Vorteil der lokalen Klassen ist, dass sie von außen nicht sichtbar sind.
- Das macht Sinn: der Benutzer der Datenstrukturen muss deren Aufbau nicht verstehen, die Schnittstellen der Listen sind klar definiert und verwenden keine Entry-Objekte.
- Nachteil: keine Wiederverwendbarkeit der Klasse `Entry<T>`.
- Die beiden Entry-Klassen sind zwar unterschiedlich, aber z.B. könnte die Entry-Klasse für Schlangen von der Entry-Klasse des Kellers abgeleitet werden, etc.

Noch ein paar Tests der beiden Klassen

```
public class Test {

   public static void main(String[] args) {
                Schlange<Integer> queue = new Schlange<Integer>();
                Keller<Integer> stack = new Keller<Integer>();

                // Test von Schlange und Keller
                queue.put(1);
                stack.push(1);
                queue.put(2);
                stack.push(2);
                queue.put(3);
                stack.push(3);

                ausgabe(queue, stack); // Dannach sind beide leer!!!

                // Test von spiegeln
                queue.put(1);
                queue.put(2);
                queue.put(3);

                spiegeln(queue);

                System.out.println("Inhalt Queue:");
                while(!queue.isEmpty()) {
                        System.out.print(queue.get().toString());
                }
                System.out.println();

        }

   ...
```

Noch ein paar Tests der beiden Klassen

```
...

public static void ausgabe(Schlange<Integer> queue, Keller<Integer> stack)
{
   System.out.println("Inhalt Queue:");
   while(!queue.isEmpty()) {
      System.out.print(queue.get().toString());
   }
   System.out.println();

   System.out.println("Inhalt Stack:");
   while(!stack.isEmpty()) {
      System.out.print(stack.pop().toString());
   }
   System.out.println();
}

...
```

Noch ein paar Tests der beiden Klassen

Ausgabe nach Aufruf von `ausgabe(queue, stack)` in `main`:

```
Inhalt Queue:
123
Inhalt Stack:
321
```

Danach sind beide Listen leer!!!

Noch ein paar Tests der beiden Klassen

```
    ...

    public static void spiegeln(Schlange<Integer> q) {
        Keller<Integer> s = new Keller<Integer>();
        while(!q.isEmpty()) {
        Integer top = q.get();
        System.out.println("Lege "+top.toString()+" auf den Stack!");
        s.push(top);
    }

    while(!s.isEmpty()) {
        Integer top = s.pop();
        System.out.println("Lege "+top.toString()+" zurueck!");
        q.put(top);
    }
}
```

Noch ein paar Tests der beiden Klassen

Ausgabe nach Aufruf von `spiegeln(queue)` in `main`:

```
Lege 1 auf den Stack!
Lege 2 auf den Stack!
Lege 3 auf den Stack!
Lege 3 zurueck!
Lege 2 zurueck!
Lege 1 zurueck!
Inhalt Queue:
321
```