Abschnitt 10: Datenstrukturen

10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen

Überblick

10. Datenstrukturen

10.1 Einleitung

10.2 Ein Datenmodell für Listen

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 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 $postfix(x, a) \in M^*$.

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

Überblick

10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen

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 *head*).
- Operationen *isEmpty* und *size* und die Projektion *proj*.

Erster Versuch mit Arrays

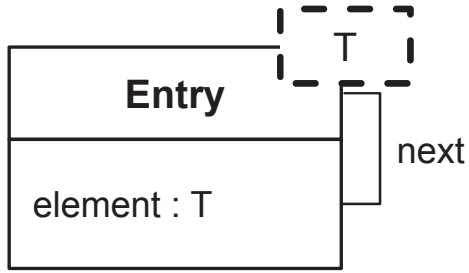
- Von der Objektorientierung herkommend, wollen wir Folgen als Objekte modellieren, die eine Liste von Elementen dynamisch verwaltet.
- Wir konzentrieren uns zunächst auf die Alternative, die Liste von vorne aufzubauen, also mit den Operationen *first*, *rest* und *prefix*.
- Eine erste Idee, in der Liste intern ein Array für die einzelnen Elemente zu halten, ist vermutlich keine gute Idee.
- 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

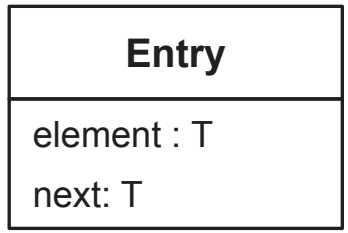
- Gehen wir objektorientiert vor: was sind die Objekte, die interagieren (und die wir modellieren müssen)?
- Eine Liste ist ein Objekt und eine ihrer wesentlichen Eigenschaften ist, dass sie eine Menge von Elementen verwaltet.
- Diese Elemente der Liste könnten wir natürlich auch als *eigenständige*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

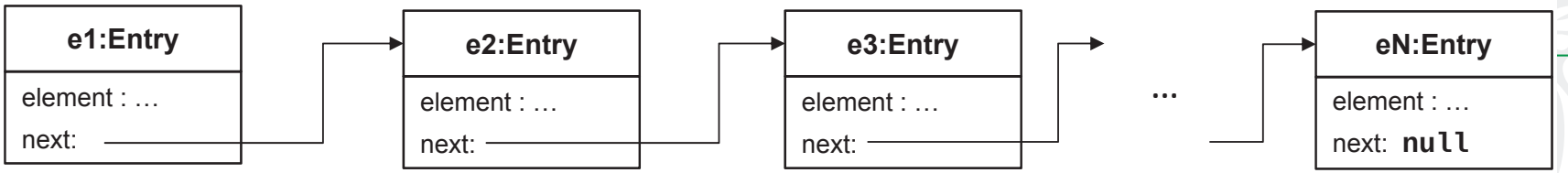
Element der Liste als besonderes Objekt

- 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



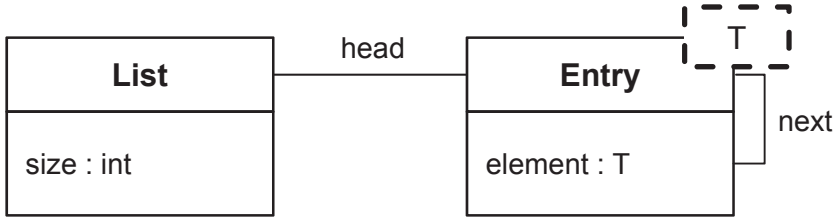


Die Liste selbst

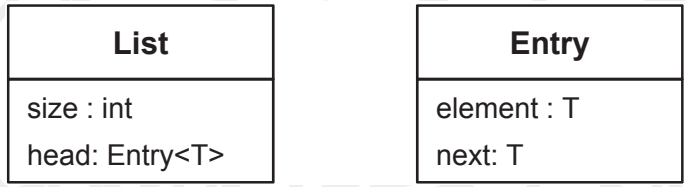
- Die Liste selbst muss nur das erste Element kennen, über dessen Verweise zum nächsten Element (Attribut `next`) können alle Nachfolger sequentiell erreicht werden.
- Dies entspricht auch der „Implementierung der Projektion“ von oben.
- Die eigentliche Liste benötigt also nur einen Verweis auf das erste 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

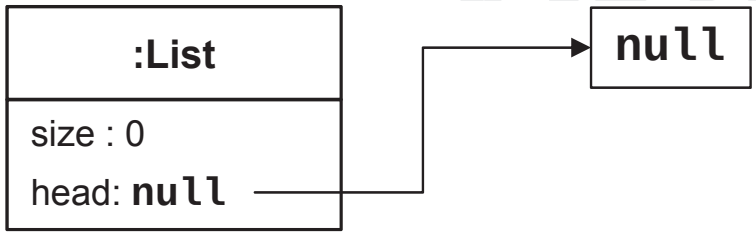
```
public class Entry<T> {
  private T element;          // das eigentliche Element vom Typ T
  private Entry<T> next;      // der Verweis auf das Nachfolgeelement

  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;
  }

  ...
}
```

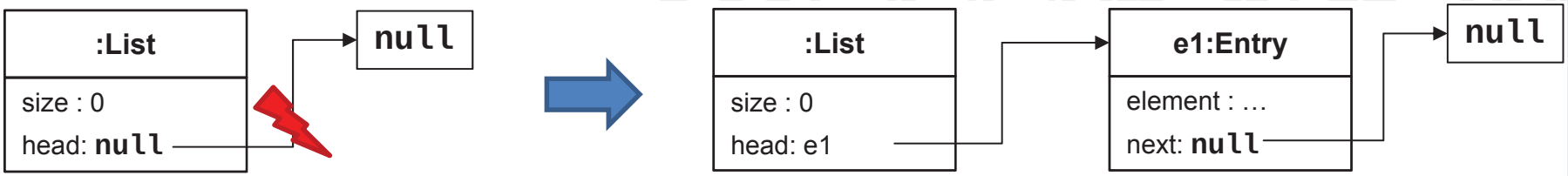
Erzeugen

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



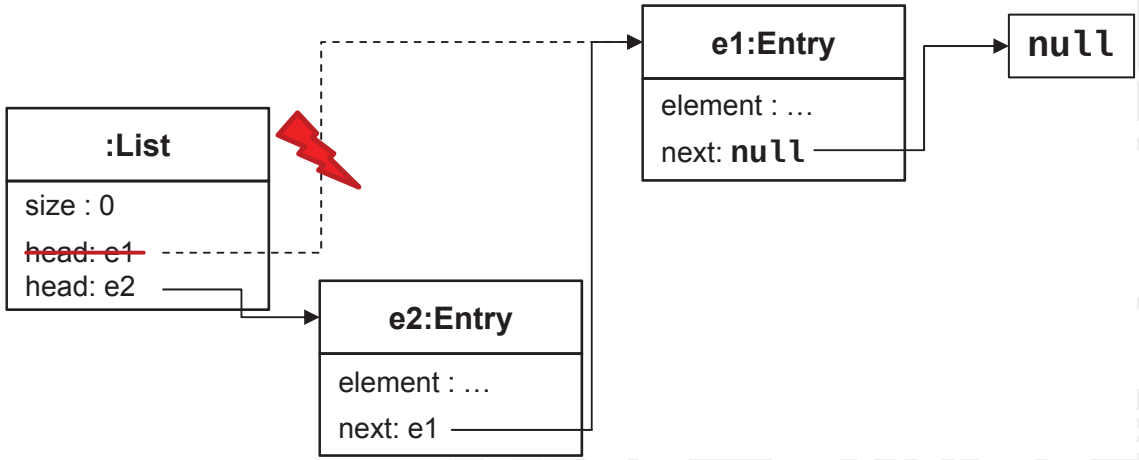
Hinzufügen mit *prefix*

- 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> entry) {
{
  newHead.setNext(this.head);
  this.head = newHead;
  this.size++;
}
...
```

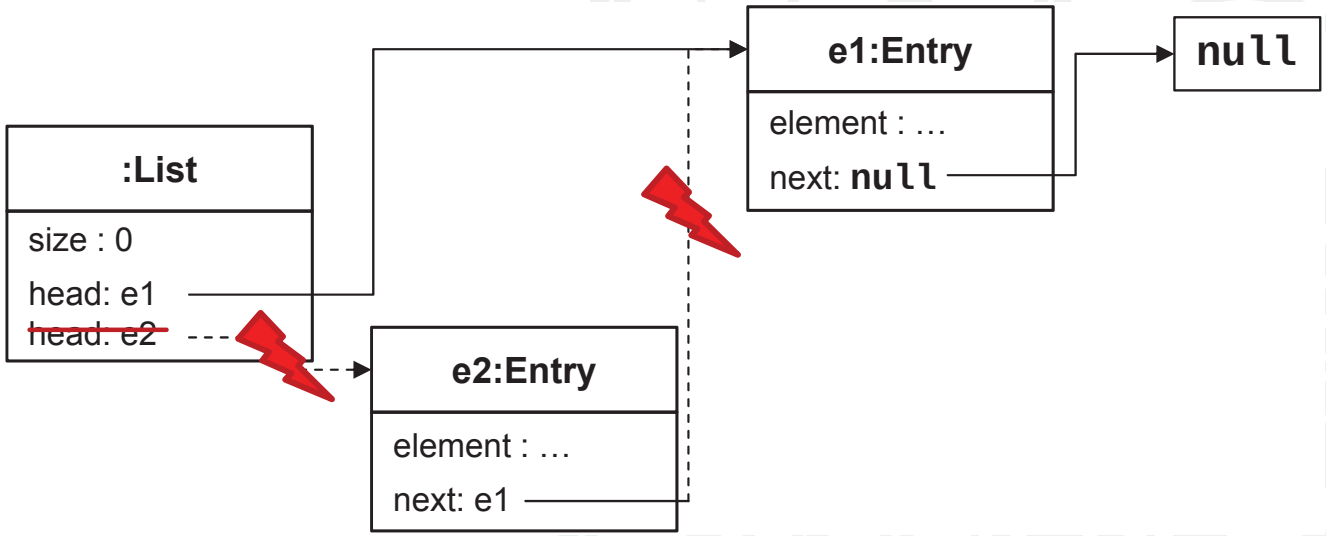
Zugriff auf das erste Element mit *first*

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

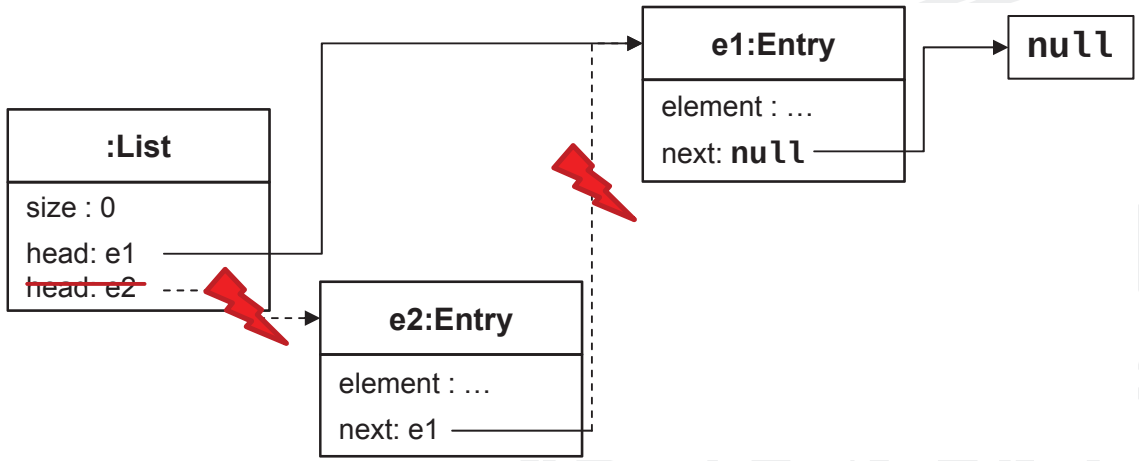
```
public T first() {
  if(this.head==null) {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}
```

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.length()) {
    throw new IllegalArgumentException(index+" exceeds length 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*

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

}
...
```