Abschnitt 8: Datenstrukturen

8. Datenstrukturen
8.1 Einleitung
8.2 Ein Datenmodell für Listen

Überblick

8. Datenstrukturen
8.1 Einleitung
8.2 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
- 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`
- Bemerkung: das oo Paradigma (insbesondere die Aspekte Abstraktion und Kapselung) eignet sich bestens, um eigene Datenstrukturen (durch Klassen) als abstrakten Datentypen zu entwickeln

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*
  Zur Erinnerung: Definition der Projektion

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

- Dafür können Folgen (und entsprechendene Umsetzungen als Listen) beliebig wachsen, während wir die Größe eines Arrays von vornherein festlegen müssen

Überblick

8. Datenstrukturen
8.1 Einleitung
8.2 Ein Datenmodell für Listen

Wiederholung: Folgen

- Konkatenation einer Folge $x \in M^*$ an ein Element $a \in M$:

$$prefix : M \times M^* \to M^*$$

mit $prefix(a, x) = (a) \circ x$

- Induktive Definition von M^*:
  1. $() \in M^*$
  2. Ist $a \in M$ und $x \in M^*$, dann ist $prefix(a, x) \in M^*$.
- Zugriff auf das erste Element:

$$first : M^+ \to M \text{ mit } first(prefix(a, x)) = a$$

- Liste nach Entfernen des ersten Elementes:

$$rest : M^+ \to M^* \text{ mit } rest(prefix(a, x)) = x$$

Ein einfaches Listenmodell

Um uns die Implementierung (in Java) von Listen auf Basis dieser Spezifikationen/Definitionen zu überlegen, halten wir fest, dass das Interface eines abstrakten Datentyps *Liste* aus folgenden Funktionalitäten besteht:

- Eine Liste kann leer sein (eine Operation $isEmpty$, die dies prüft, wäre denkbar)
- Ein Element wird vorne an eine Liste angehängt (Operation $prefix$)
- Wir können nur auf das erste Element einer Liste zugreifen und auch nur dieses entfernen (Operationen $first$ und $rest$)
- Zusätzlich: Eine Liste kann Auskunft über ihre Länge (= Anzahl der Elemente) geben durch folgende Operation:

$$size(x) = \begin{cases} 0, & \text{falls } x = (), \\ 1 + size(rest(x)) & \text{sonst.} \end{cases}$$

Erster Versuch mit Arrays

- Von der Objektorientierung herkommend, wollen wir Folgen als Objekte modellieren
- Erste Idee: wir verwalten intern ein Array für die einzelnen Einträge der Liste und implementieren die Schnittstelle als öffentliche Methoden *darum herum*
- Um generisch zu bleiben, könnten wir als Eintrag Objekte vom `Typ java.lang.Object` verwenden
- Vorteil wäre, dass diese Liste wegen der Typ-Polymorphie dann Objekte aller möglichen Klassen aufnehmen können
- Nachteil: genau wie im Kapitel über *generics* besprochen, wissen wir dann aber im Zweifel nicht, von welcher konkreten Klasse wir Objekte in die Liste eingestellt haben, d.h. Typfehler sind beinahe schon vorprogrammiert

Erster Versuch mit Arrays

- Daher ist es vermutlich sinnvoller, generische Typvariablen für die Einträge zu verwenden und die Liste so typsicher zu machen

```
public class ListeErsterVersuch<T> {
   /** Die Entr&auml;ge der Liste. */
   private T[] entries;

   /** Konstruktor zum Erzeugen einer leeren Liste. */
   public ListeErsterVersuch() {
       this.entries = new T[0];
   }

   // die weiteren Methoden der Schnittstelle ...
   public int size() { ... }

   public T first() { ... }

   public ListeErsterVersuch<T> rest() { ... }

   public void prefix(T entry) { ... }
```

Erster Versuch mit Arrays

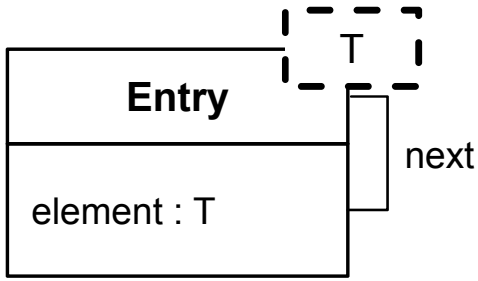
- Problem: Die Implementierungen der Methoden `prefix` und `rest` werden leider etwas komplizierter, da Arrays nicht dynamisch sind
- Wir müssten daher immer neue Arrays erzeugen und tiefe Kopien anfertigen (d.h. alle elemntweise kopieren), was sehr aufwändig zu programmieren und auszuführen ist
- Fazit: die Idee, Arrays zu verwenden, ist nicht so gut (da sie semi-dynamisch sind und sich daher offenbar für die Modellierung andere Dinge besser eignen)

Zweiter Versuch: oo Modellierung

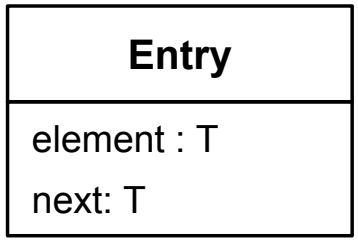
- Gehen wir objektorientiert vor: was sind die Objekte, die wir modellieren müssen?
- Eine Liste ist ein Objekt; was hat dieses Objekt für Eigenschaften?
- Es hält insb. eine Menge von Elementen
- Diese Elemente der Liste könnten wir natürlich auch als *eigenständige* Objekte ansehen und entsprechend modellieren

Element der Liste als besonderes Objekt

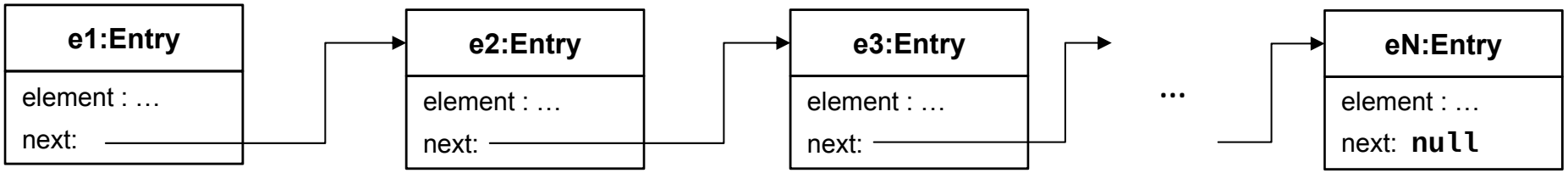
- Welche Eigenschaften hat ein element der Liste (als eigenständiges Objekt modelliert)?
- Dieses Objekt hält zunächst das eigentlich gespeicherte Element
- Dieses Objekt steht in Beziehung z.B. zum nächsten Element der Liste, d.h. es kennt einen *Verweis* auf das nächste Element, den Nachfolger



bzw. entspricht

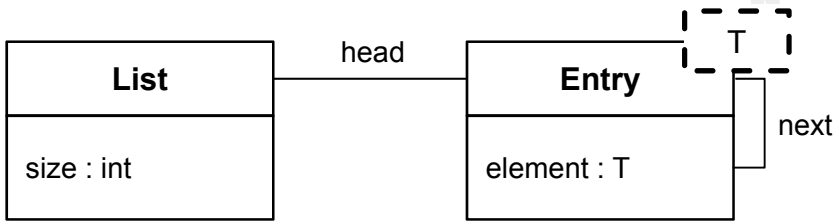


Bildlich:

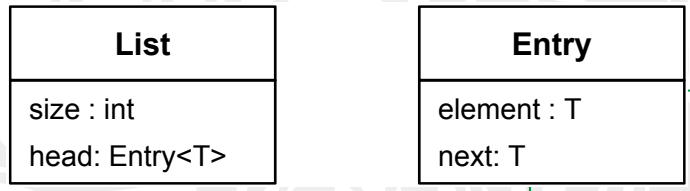


Die Liste selbst

- Die Liste muss nur das erste Element kennen, über dessen Verweise zum nächsten Element (Attribut `next`) können alle Nachfolger erreicht werden
- 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`**
- 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

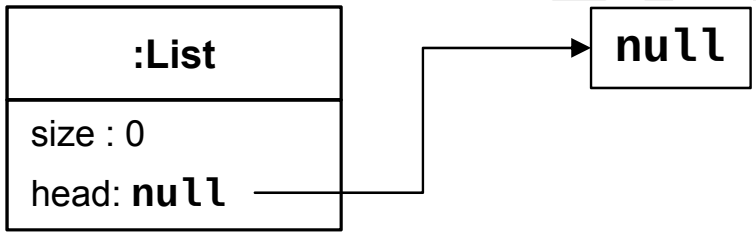
```
public class Entry<T> {
  private T element;
  private Entry<T> next;

  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

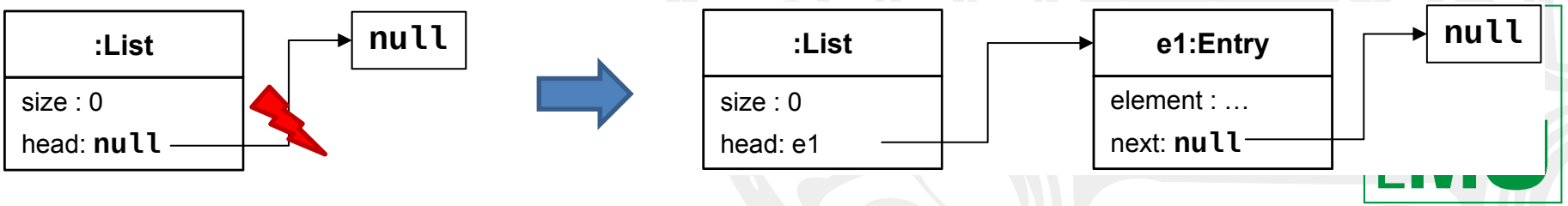
```
public class List<T>
{
  private int size;

  private Entry<T> head;

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

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

  ...
}
```

Erzeugen und Hinzufügen

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

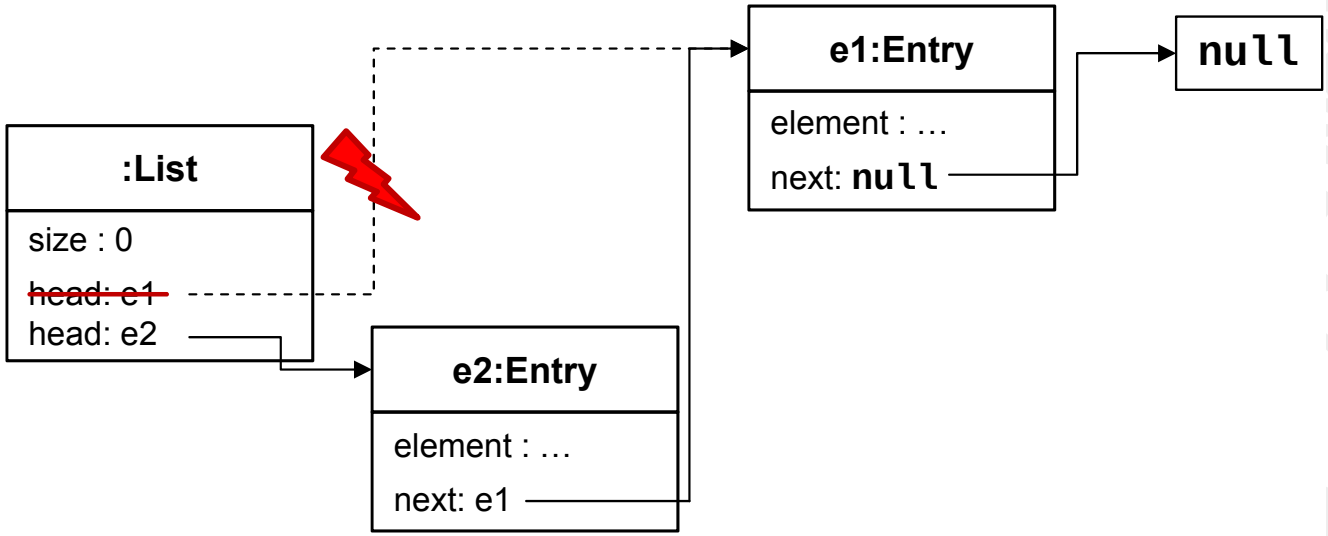


- Um ein neues Element hizuzufügen, wird ein neues `Entry`-Element erzeugt und als neues erstes Element gesetzt
- Dessen Nachfolger ist das alte erste Element, d.h. das neue Element bekommt dieses als Nachfolger
- Die Länge der Liste erhöht sich um 1



Erzeugen und Hinzufügen

- Und so gehts weiter:




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

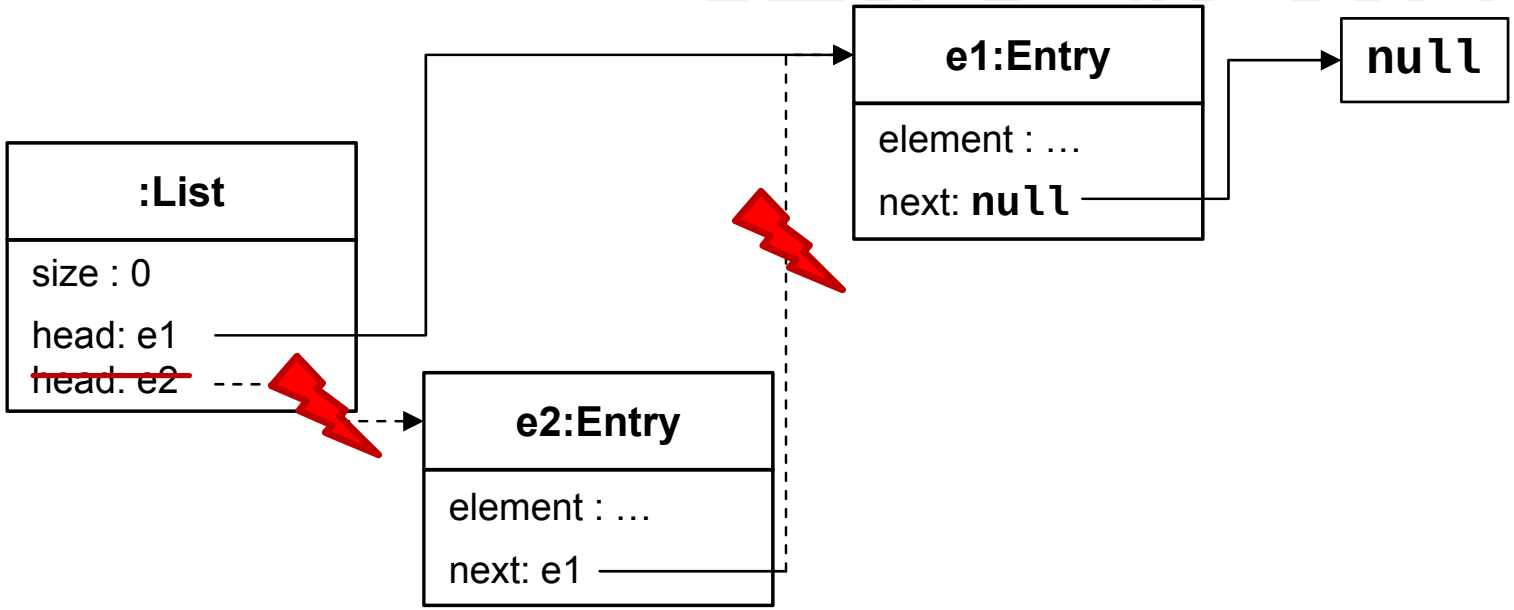
Zugriff auf das erste Elemente

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

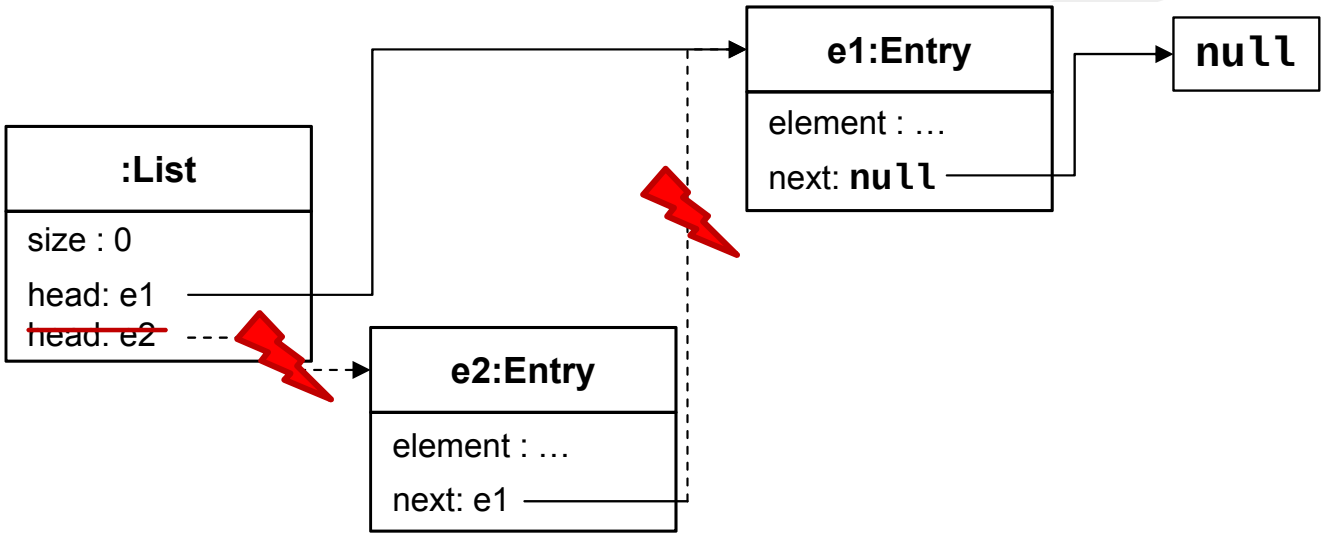
```
public T first() {
  if(this.head==null) {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}
```

Entfernen des ersten Elementes

- 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;
}
```