Prof. Dr. Sven Strickroth

Einführung in die Programmierung

Wintersemester 2021/22

Grundkonzepte der Programmierung III: Dynamische Datenstrukturen

Themen

1. Datenstrukturen (allgemein)
2. Listen
   a. Beispiel: Stack und Queue
3. Bäume (nichtlineare Datenstrukturen)

Was sind Datenstrukturen?

- Viele Computer-Programme sind natürlich in erster Linie dazu da, Daten zu verarbeiten.
  - Die Daten müssen dazu intern organisiert und verwaltet werden, dazu dienen *Datenstrukturen*.
  - Oft gibt es viele alternative Möglichkeiten mit verschiedenen Vor- und Nachteilen, um eine gegebene Menge von Daten zu verwalten.
- Beispiele:
  - *Arrays:* Einfache Datenstruktur zur Verwaltung gleichartiger Elemente
  - Verbünde: Einfache Datenstruktur zur Verwaltung weniger verschiedenartiger Elemente
- Mit Klassen können wir sehr flexible Datenstrukturen definieren.

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 Ausführungszeit und Speicherplatz sparen.

Datenstrukturen und Objektorientierung

- Die Konzepte der Objektorientierung eignen sich gut, um Datenstrukturen zu realisieren.
  - Daten werden als Ansammlung von Objekten repräsentiert.
  - Eine Klasse bietet eine genau definierte *Schnittstelle* (Typen und Methoden mit Signatur): Abstrakter Datentyp
  - Durch private Sichtbarkeit können Implementierungsdetails vor dem Anwender verstekt werden.

Abstrakter Datentyp

- Ein abstrakter Datentyp (ADT) ist ein Verbund von Daten zusammen mit der Definition aller zulässigen Operationen, die auf sie zugreifen.

- Spezifikation über Signatur und Semantik, z.B.
  - Interface und verbale Beschreibung der Operationen
  - mathematisch-algebraisch mit Axiomen
    (→ Vorlesung „Algorithmen und Datenstrukturen“)

- Abstrakter Datentyp kann verschiedene Implementierungen haben
  - → konkrete Datentypen

- Verschiedene Datenstrukturen können die gleiche Schnittstelle implementieren
  - Allgemein: konkrete Datentypen können andere abstrakte Datentypen „implementieren“
  - → Wahl der Datenstruktur ist Entwurfsentscheidung

Ziel des Kapitels

- Wir wollen (zunächst) Datenstrukturen für Listen im Detail besprechen.
  - Liste: Dynamische Menge gleichartiger Objekte mit festgelegter Reihenfolge
- Java enthält Standard-Implementierungen für viele Datenstrukturen (auch Listen): Collections-Framework (später in der Vorlesung)
- Trotzdem studieren wir „eigene" Implementierungen
  - zum besseren Verständnis von Vor- und Nachteilen und
  - als detailliertes Anwendungsbeispiel für die Konzepte der objektorientierten Programmierung.

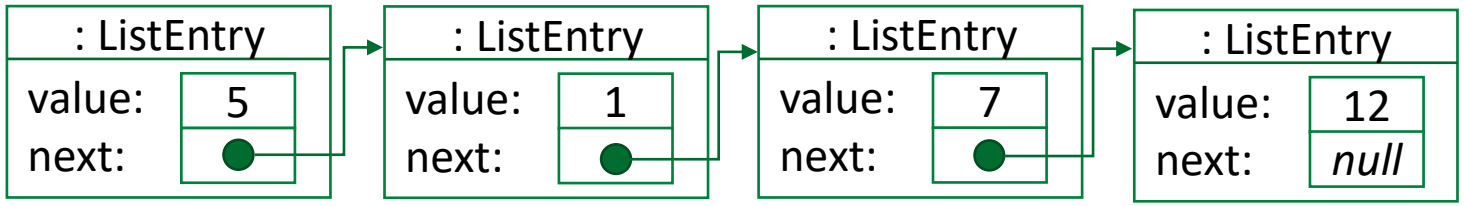
Themen

1. Datenstrukturen (allgemein)
2. Listen
   a. Beispiel: Stack und Queue
3. Bäume (nichtlineare Datenstrukturen)

Listen

- Nachteil von Arrays:
  - feste Größe (in Java)
  - beliebiges Einfügen neuer Elemente und Entfernen von Elementen nicht einfach möglich → umkopieren notwendig

- Idee:
  - Jedes Listenelement ist ein eigenes Objekt (hier der Klasse `ListEntry`).
  - Das Objekt enthält jeweils einen Elementwert (*value*) und eine Referenz auf den Rest der Liste (*next*).
  - „*Einfach verkettete Liste*“

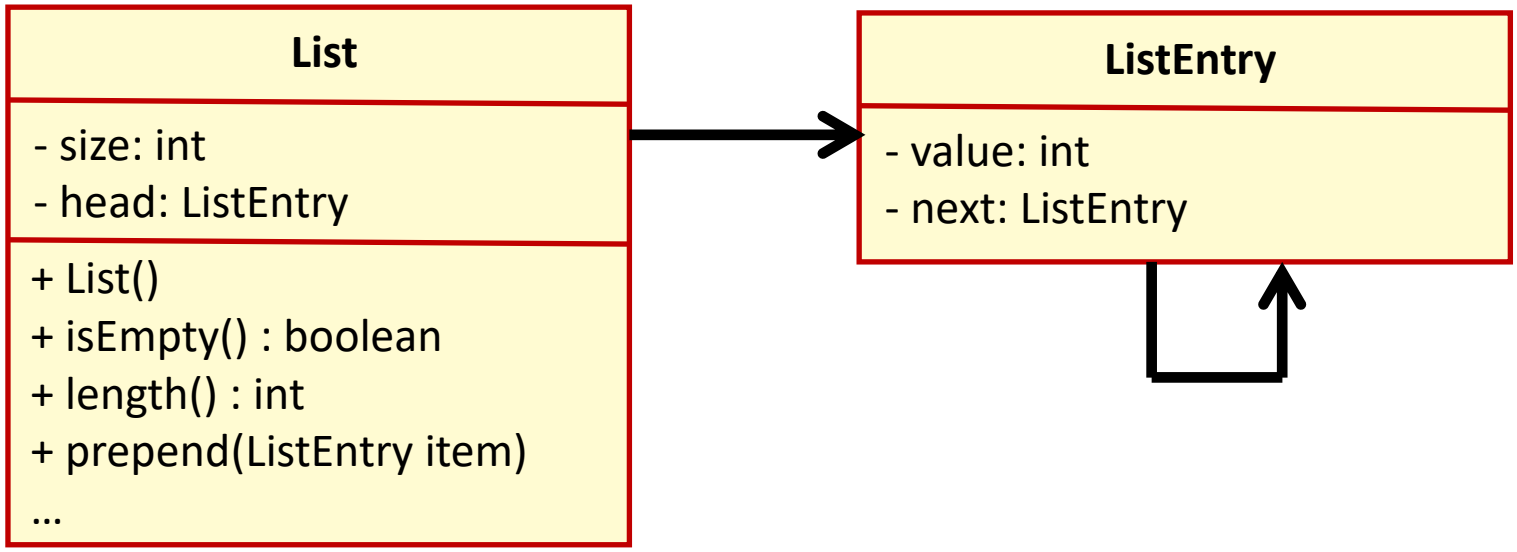


Schnittstelle für Listen

- Welche Methoden erwarten wir von einer Implementierung einer Liste? (Abstrakter Datentyp)
  - Erzeugen einer leeren Liste (Konstruktor)
  - Element an eine Liste anhängen wahlweise
    - vorne (*prepend*), hinten (*append*), als i-tes Element (*insertAt*)
  - Zugriff auf das
    - erste Element (*first*), letzte Element (*last*), i-te Element der Liste (*getValueAt*)
  - Entfernen des
    - ersten Elements (*deleteFirst*), letzten Elements (*deleteLast*), i-tes Element entfernen (*deleteAt*)
  - Test, ob die Liste leer ist (*isEmpty*)
  - Anzahl der Elemente der Liste (*length*)
  - komplette Liste löschen (clear)
  - …

Liste als Objekt

- Es ist oft hilfreich, ein eigenes Verwaltungsobjekt für eine Datenstruktur anzulegen.
  - eigene Klasse, hier `List` mit:
  - Referenz auf das erste Element (*head*)
  - Länge der Liste (*size*) (nur zur Effizienzverbesserung)
  - öffentlichen Methoden, um mit der Liste zu arbeiten



Element-Klasse in Java

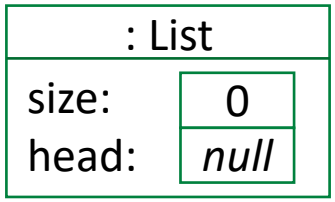
```
public class ListEntry {
  private int value; // Listeneintrag, hier exemplarisch ein int
  private ListEntry next; // Referenz auf Nachfolgeelemente

  public ListEntry(int value) {
    this.value = value;
    this.next = null;
  }

  public int getValue() {
    return value;
  }

  public ListEntry getNext() {
    return next;
  }

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

Listen-Klasse in Java

```
public class List {
  private int size; // Listenlänge, zum schnelleren Zugriff
  private ListEntry head;

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

  public void clear() {
    size = 0;
    head = null;
  }

  public int length() {
    return size;
  }

  …
}
```

Die leere Liste

- Der Konstruktor `List()` erzeugt ein Objekt, das eine leere Liste repräsentiert.
  - Die leere Liste ist also *nicht* durch die `null`-Referenz repräsentiert! → bessere Kapselung



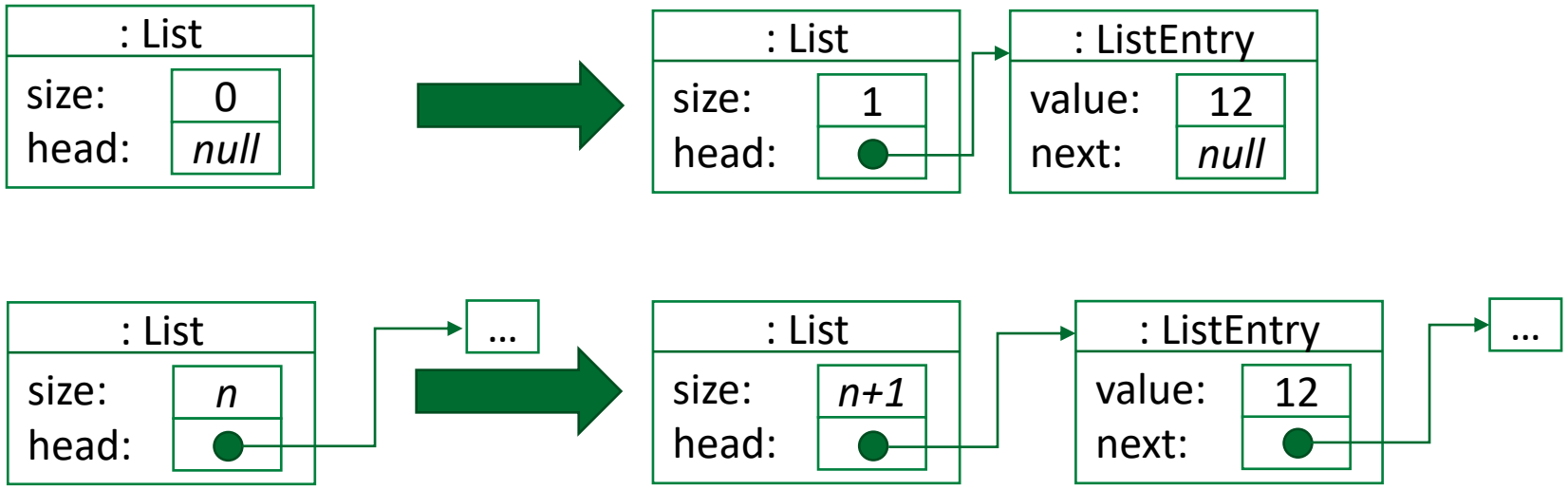
- Prüfen, ob die Liste leer ist:

```
public int isEmpty() {
  return head == null;
}
```

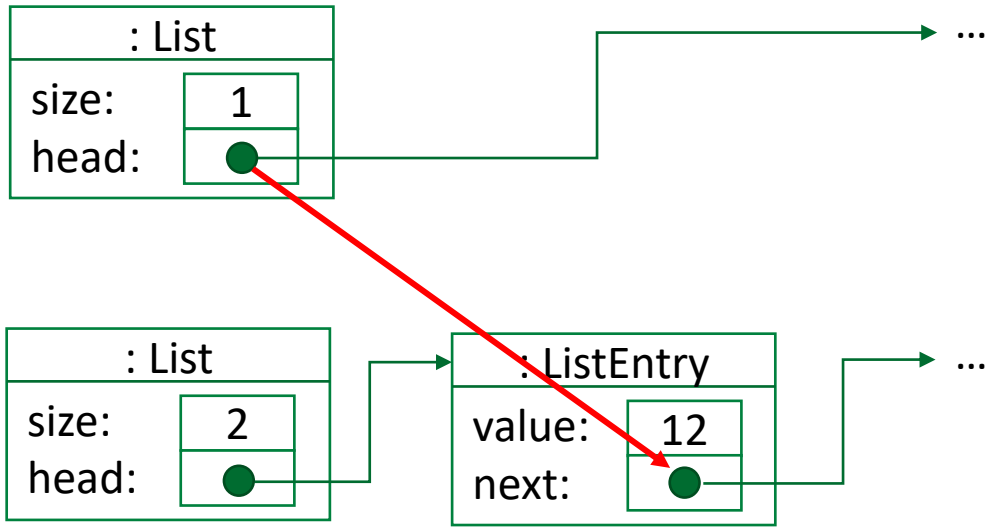
  - alternativ: `return size == 0;`

Hinzufügen mit *prepend* (1)

- Die Operation *prepend* fügt ein neues Element „vorne“ an die Liste an.
  - Neues ListEntry-Element erzeugen, wird neues erstes Element (*head*)
  - Nachfolger des neuen Elements ist das alte erste Element, also *head* umsetzen.
  - Die Länge der Liste erhöht sich um 1.



Hinzufügen mit *prepend* (2)




```
public void prepend(int value) {
  ListEntry newEntry = new ListEntry(value);
  newEntry.setNext(head);
  head = newEntry;
  size++;
}
```

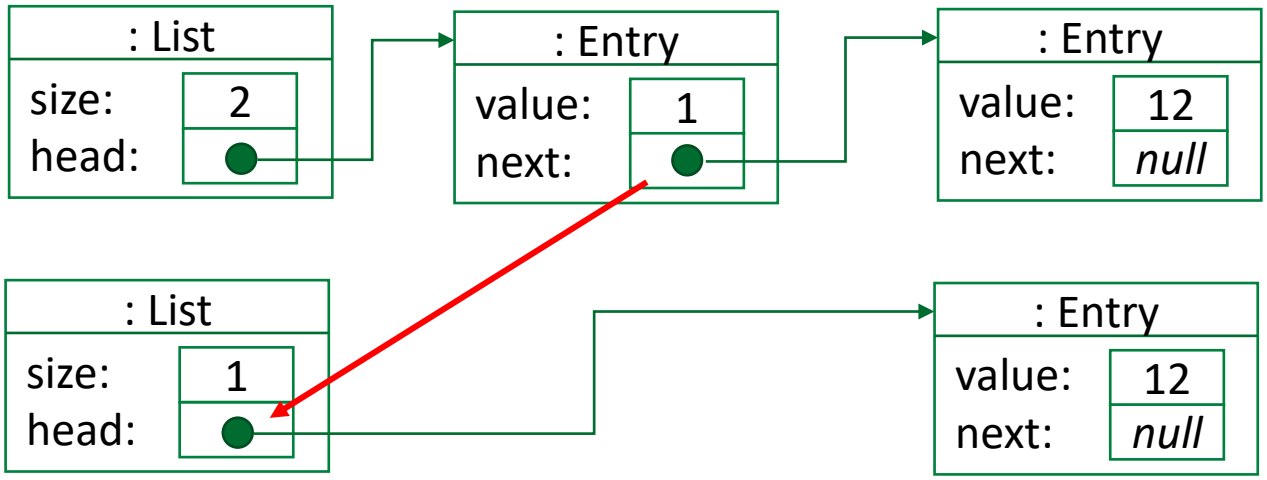
Zugriff auf das erste Element mit *first*

- Das erste Element ist leicht zu finden: *head*-Eintrag.
- Problem: Liste kann leer sein (dann ist *head* null).
  - Dann geben wir erst einmal einen Fehler aus und -1 zurück

```
public int first() {
  if (isEmpty()) {
    // Fehlerbehandlung studieren wir später
    System.out.println("Fehler, Liste ist leer.");
    return -1;
  }
  return head.getValue();
}
```

Entfernen des ersten Elements mit *deleteFirst*

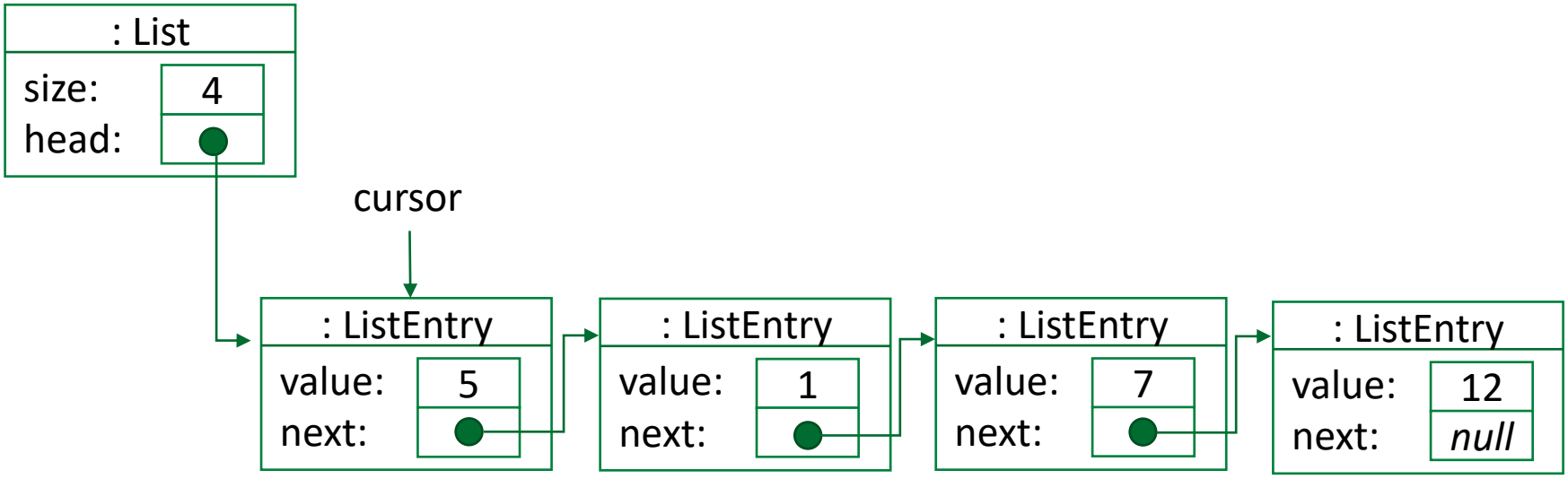
```
public void deleteFirst() {
  if (isEmpty()) { // Fehlerbehandlung später
    return;
  }
  head = head.getNext();
  size--;
}
```



Traversierung von Listen

```
public void printEntries() {
  ListEntry cursor = head;
  while (cursor != null) {
    System.out.println(cursor.getValue());
    cursor = cursor.getNext();
  }
}
```

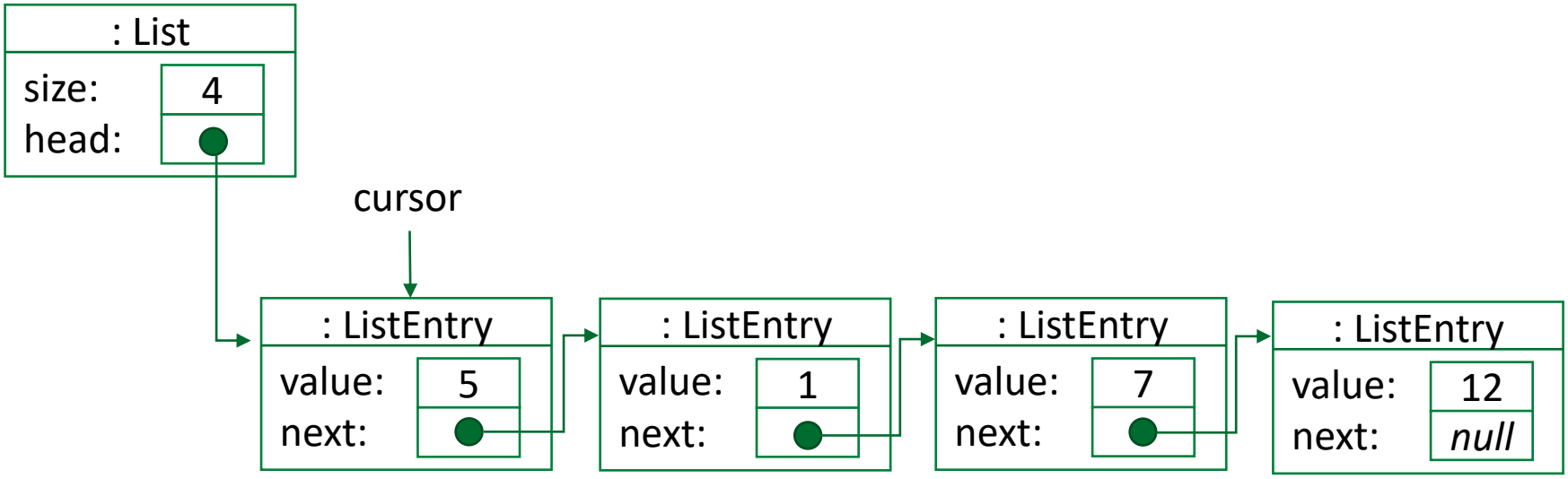
Ausgabe:



Traversierung von Listen

```
public void printEntries() {
  ListEntry cursor = head;
  while (cursor != null) {
    System.out.println(cursor.getValue());
    cursor = cursor.getNext();
  }
}
```

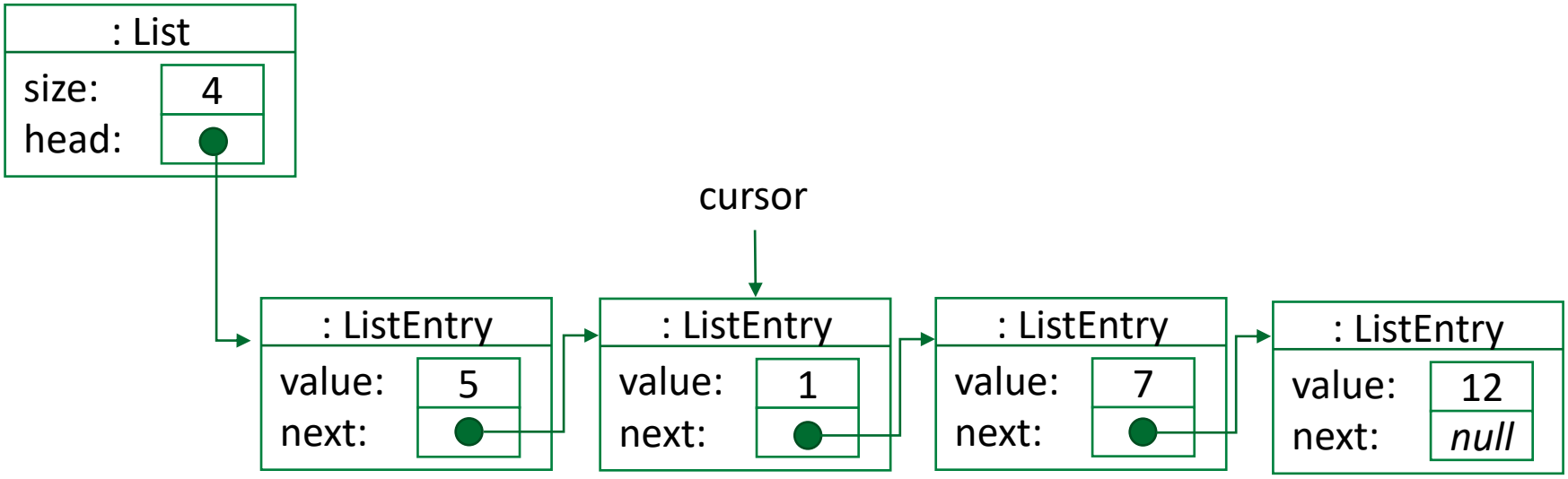
Ausgabe:
5



Traversierung von Listen

```
public void printEntries() {
  ListEntry cursor = head;
  while (cursor != null) {
    System.out.println(cursor.getValue());
    cursor = cursor.getNext();
  }
}
```

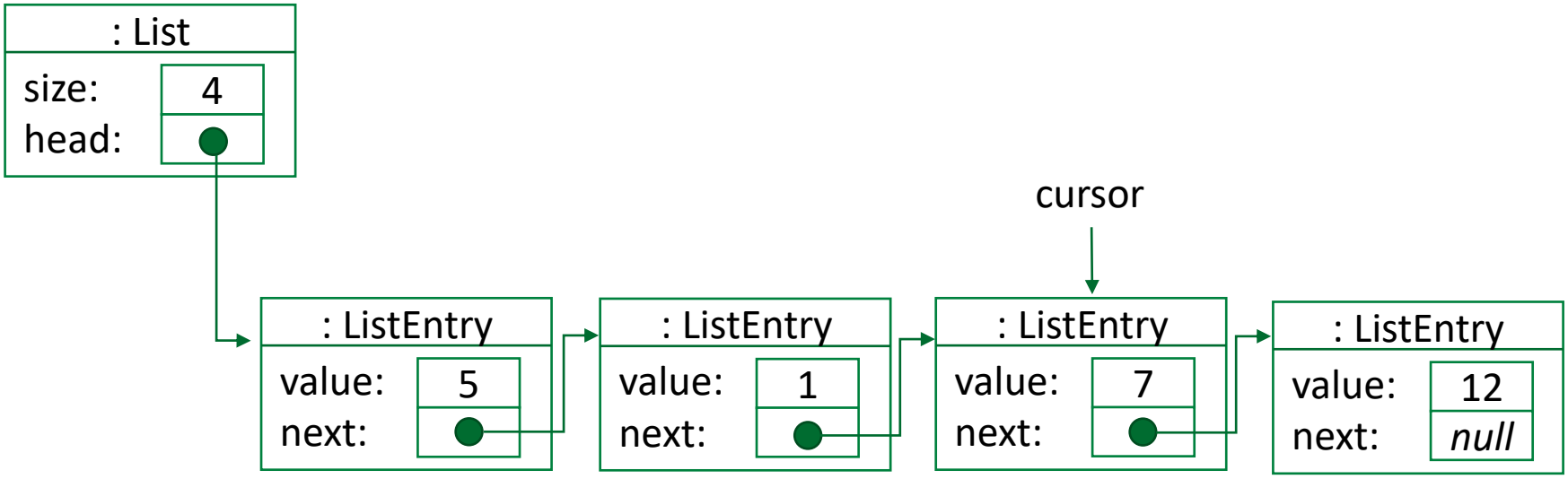
Ausgabe:
5
1



Traversierung von Listen

```
public void printEntries() {
  ListEntry cursor = head;
  while (cursor != null) {
    System.out.println(cursor.getValue());
    cursor = cursor.getNext();
  }
}
```

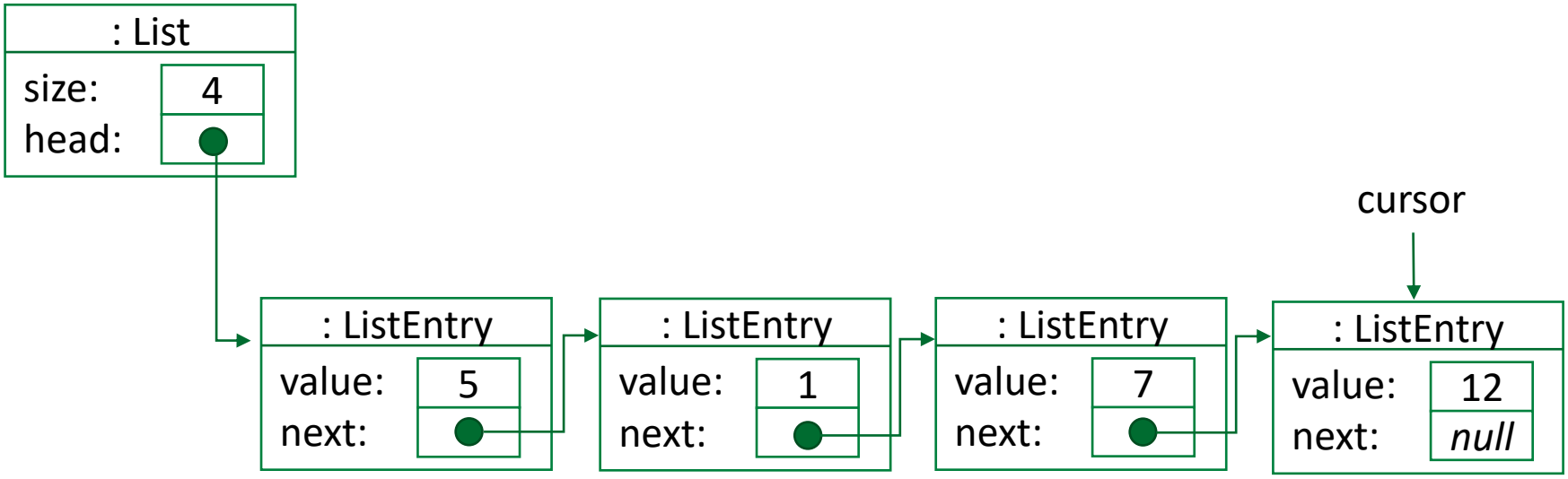
Ausgabe:
5
1
7



Traversierung von Listen

```
public void printEntries() {
  ListEntry cursor = head;
  while (cursor != null) {
    System.out.println(cursor.getValue());
    cursor = cursor.getNext();
  }
}
```

Ausgabe:
5
1
7
12



Generische toString()-Methode für Listen

- „Schönere Ausgabe“:
  - leere Liste: []
  - sonst, z.B. [1, 2, 3]

```
@Override
public String toString() {
  StringBuilder res = new StringBuilder("[");
  ListEntry cursor = head;
  while (cursor != null) {
    res.append(cursor.getValue());
    if (cursor.getNext() != null)
      res.append(", ");
    cursor = cursor.getNext();
  }
  res.append("]");
  return res.toString();
}
```

- Hier Benutzung von StringBuilder, weil String immutable ist und wir mit normalen Konkatenationen viele temporäre Strings anlegen würden.

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

- Index startet bei 0
- Der Zugriff, der bei Arrays sehr effektiv war, ist bei sequentiellen Listen umständlicher (ein *trade-off*).

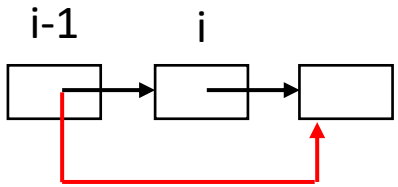
```
public int getValueAt(int index) {
  if (index < 0 | index >= size) {
    // TODO: Fehlerbehandlung studieren wir später
    System.out.println("index out of range");
    return -1;
  }

  ListEntry cursor = head;
  for (int j = 0; j < index; ++j)
    cursor = cursor.getNext();
  return cursor.getValue();
}
```

- Wahlfreier Zugriff auf das i-te Element benötigt i Schritte.

i-tes Element entfernen

- An die passende Stelle der Liste „laufen“
  - Achtung: bei i=0 muss head angepasst werden!
- dann Verweise passend „umbiegen“

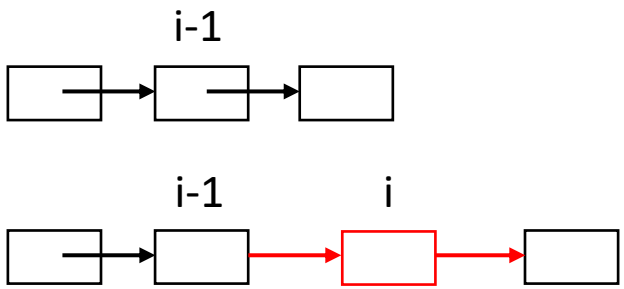
```
public void deleteAt(int index) {
  if (index < 0 || index >= size) return; // TODO: Optimize

  if (index == 0) { // Spezialfall für erstes Element
    head = head.getNext(); // Alternativ: deleteFirst()-Methode
    size--;
    return;
  }

  ListEntry cursor = head;
  for (int j = 0; j < index - 1; ++j)
    cursor = cursor.getNext();
  cursor.setNext(cursor.getNext().getNext());
  size--;
}
```



Element an Stelle *i* einfügen

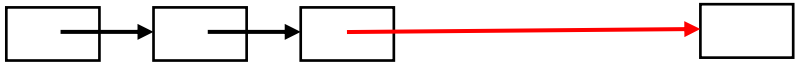
- Auch dazu an die passende Stelle der Liste „laufen“
  - Achtung einfügen *nach* dem (*i*-1)-ten Element! (Spezialfälle *i*=0 und *i*=*size*)
  - dann Verweise passend umsetzen

```
public void insertAt(int value, int index) {
  if (index < 0 || index > size) return; // TODO: Optimize

 if (index == 0) {
    prepend(value);
    return;
  }
  ListEntry cursor = head;
  for (int i = 0; i < index - 1; i++)
    cursor = cursor.getNext();
  ListEntry newEntry = new ListEntry(value);
  newEntry.setNext(cursor.getNext());
  cursor.setNext(newEntry);
  size++;
}
```

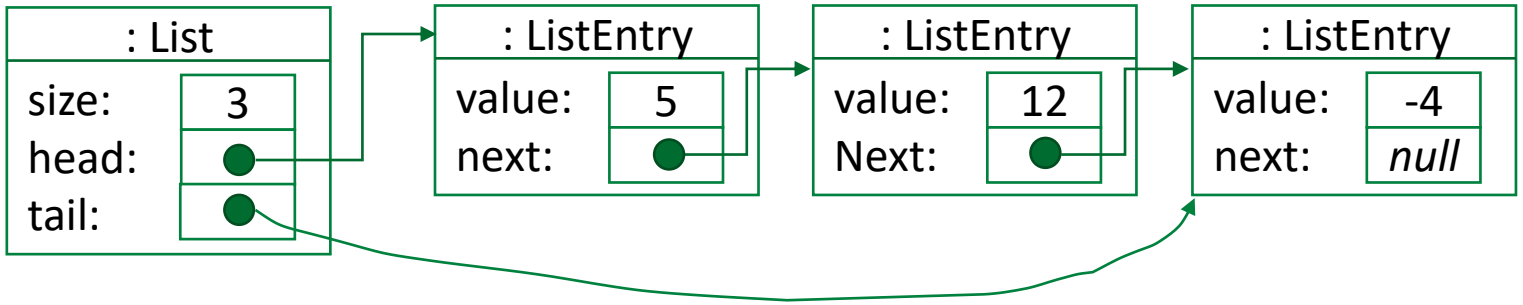


Einfügen am Ende

- Um ein Element am Ende einzufügen, müssen wir
  - die Liste komplett bis zum Ende durchlaufen
  - und das nur, um das letzte Element zu bekommen!

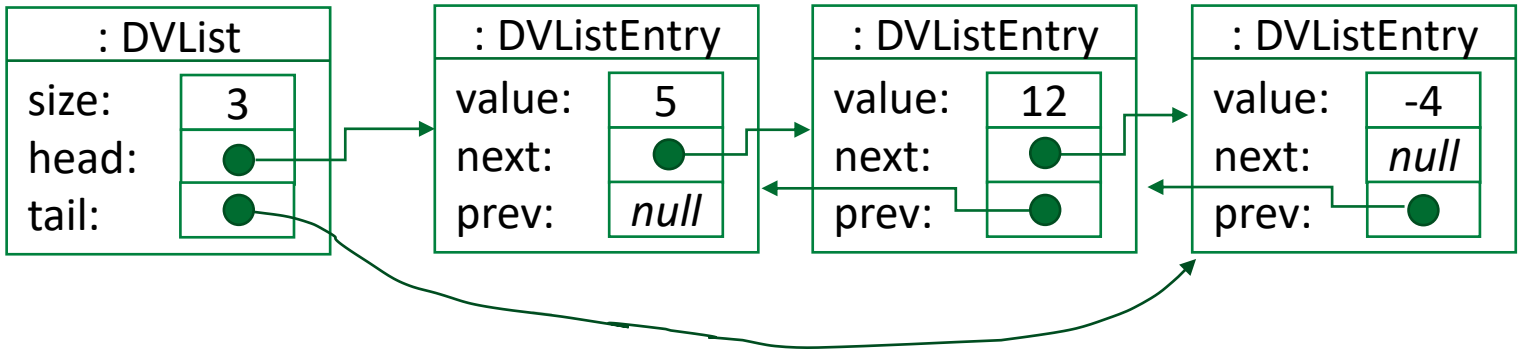


- Wir könnten zusätzlich eine Referenz auf das aktuell letzte Element im Listenobjekt führen
  - *Zweifach verankerte* statt *einfach verankerte* Liste



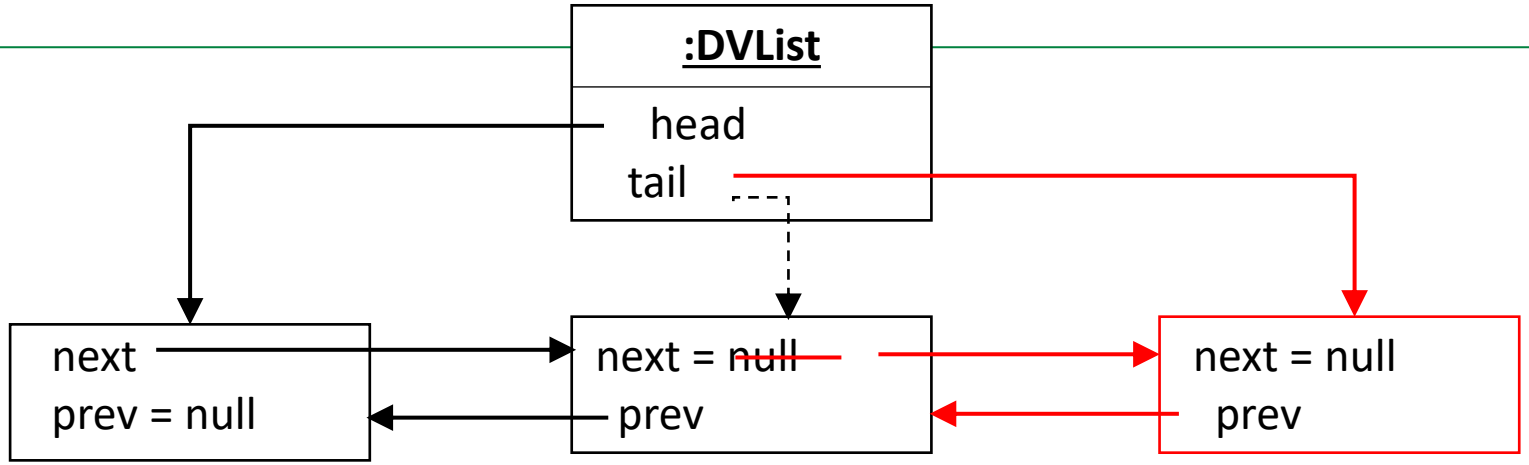
Doppelt verkettete Liste

- Wunsch: schneller Zugriff auch auf die letzten Elemente (z.B. letztes Element entfernen)
- Mögliche Lösung: Doppelt verkettet Liste (doubly linked list)
- Die Elemente sind in beide Richtungen verkettet
  - Symmetrie im Aufwand beim Durchlaufen der Kette
  - größerer Speicherbedarf
  - größerer Aufwand bei Entfernen/Einfügen



Doppelt verkettete Liste: *addLast*

```
public void addLast(int value) {
  DVListEntry newTail = new DVListEntry(value);
  if (tail == null) { // leere Liste: auch head setzen
    head = newTail;
    tail = newTail;
  } else { // nicht leer: head kann bleiben
    newTail.setPrev(tail);
    tail.setNext(newTail);
    tail = newTail;
  }
  size++;
}
```

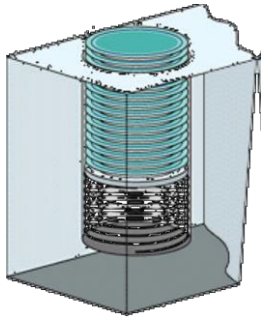


Themen

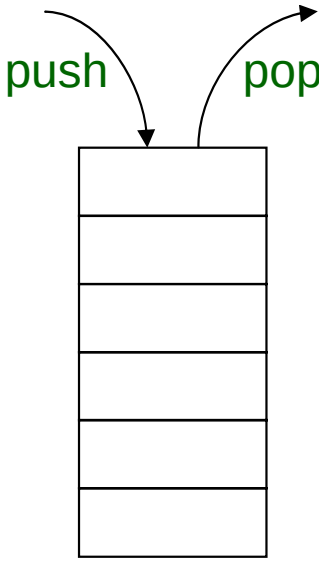
1. Datenstrukturen (allgemein)
2. Listen
   a. Beispiel: Stack und Queue
3. Bäume (nichtlineare Datenstrukturen)

Datenstruktur-Beispiel: Keller (Stapel, Stack)

- Ein *Keller (Stack)* ist eine *LIFO-(Last-In-First-Out)*-Datenstruktur
  - Cafeteriabehälter, „die letzten werden die ersten sein“, Speicher für Methodenaufrufe in Programm
  - direkter Zugriff nur auf das zuletzt eingefügte Element



- Stacks verfügen typischerweise über folgende Operationen:
  - `void push(TYPE value)`
    legt Wert „oben“ auf dem Stapel ab
  - `TYPE top()`
    gibt das „oberste“ Element zurück
  - `void pop()` oder `TYPE pop()`
    entfernt das „oberste“ Element vom Stapel
  - manchmal: nur `pop()` mit Ergebnis, kein `top()`
- Implementierung mit verketteten Listen gut realisierbar
  - einfache Verkettung, einfacher Anker genügt



Stack mit lokaler Klasse (1)

```
public class Stack {
  private static class Entry {
    Entry next;
    int value;

    Entry(int value) {
      this.value = value;
    }
  }

  private Entry first;

  public boolean isEmpty() {
    return first == null;
  }
  …
}
```

Stack mit lokaler Klasse (2)

```
public class Stack {
  …

  public void push(int value) {
    Entry newEntry = new Entry(value);
    newEntry.next = first;
    first = newEntry;
  }

  public int pop() {
    int res = first.value;
    first = first.next;
    return res;
  }
}
```

Lokale Hilfsklassen in Datenstrukturen

- Manche Klassen werden nur für interne Zwecke einer Datenstruktur benötigt
  - Beispiel: Die `Entry`-Klassen in unsere Stack-Implementierung
- Solche Klassen kann man auch lokal in einer anderen Klasse definieren (*`InnerClass`*).
  - z.B. *`Entry`* innerhalb von *`Stack`*

- Lokale Klassen haben Zugriff auf alle Komponenten (auch privater Komponenten) der Klasse, in die sie eingebettet sind.
- Nachteil: Lokale Klassen können nicht so gut wiederverwendet werden.

Stack mit Array (1)

- Der Stack könnte auch mit einem Array implementiert werden

- Hier mit Hilfe eines „dynamischen“ Arrays
  - Instanzvariablen:
    a = Array
    size = # Elemente auf dem Stack
  - push: speichere Element in a[size]
    wenn Array voll, erzeuge neues Array mit doppelter Größe
    → seltenes umkopieren
  - pop: entferne ein Element aus a[size-1]

Stack mit Array (2)

- StackArray s = new StackArray();

index	0
value	?

size = 0

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);

index	0
value	1

size = 1

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);

index	0	1
value	1	2

size = 2

Größe des Arrays verdoppeln…

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);
- s.push(3);

index	0	1	2	3
value	1	2	3	?

size = 3

Größe des Arrays verdoppeln…

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);
- s.push(3);
- s.pop(); // → 3

index	0	1	2	3
value	1	2	?	?

size = 2

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);
- s.push(3);
- s.pop();
- s.push(4);
- s.push(5);

index	0	1	2	3
value	1	2	4	5

size = 4

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);
- s.push(3);
- s.pop();
- s.push(4);
- s.push(5);
- s.push(6);

index	0	1	2	3	4	5	6	7
value	1	2	4	5	6	?	?	?

size = 5

Größe des Arrays verdoppeln…

Stack mit Array (2)

- StackArray s = new StackArray();
- s.push(1);
- s.push(2);
- s.push(3);
- s.pop();
- s.push(4);
- s.push(5);
- s.push(6);
- s.pop(); // → 6

index	0	1	2	3	4	5	6	7
value	1	2	4	5	?	?	?	?

size = 4

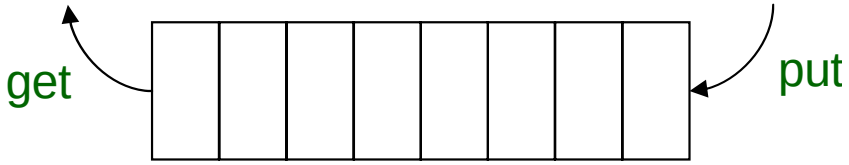
Stack mit Array: Quellcode

```
import java.util.Arrays;

public class StackArray {
  private int size = 0;
  private int[] array = new int[1];

  public boolean isEmpty() {
    return size == 0;
  }

  public void push(int value) {
    if (size >= array.length) {
      // verdoppeln der Länge
      array = Arrays.copyOf(array,
                  array.length * 2);
    }
    array[size] = value;
    ++size;
  }
…
```

```
…

  public int pop() {
    size--;
    return array[size];
  }
}
```

Datenstruktur-Beispiel: Warteschlange (Queue)

- Eine *Warteschlange (Queue)* ist eine *FIFO-(First-In-First-Out)*-Datenstruktur.
  - „Wer zuerst kommt malt zuerst“
- Queues verfügen typischerweise über folgende Operationen:
  - `void append(TYPE value)` (oder `put`) fügt Objekt „hinten“ an die Schlange an
  - `TYPE value()` (oder `get, element` oder `peek`) gibt das „vorderste“ Element zurück
  - `void remove()` oder `TYPE remove()` Entfernt das „vorderste“ Element aus der Schlange
  - Manchmal: nur `remove()` mit Ergebnis, kein `value()`



- Mit verketteten Listen gut realisierbar
  - zweifacher Anker, einfache Verkettung genügt

Themen

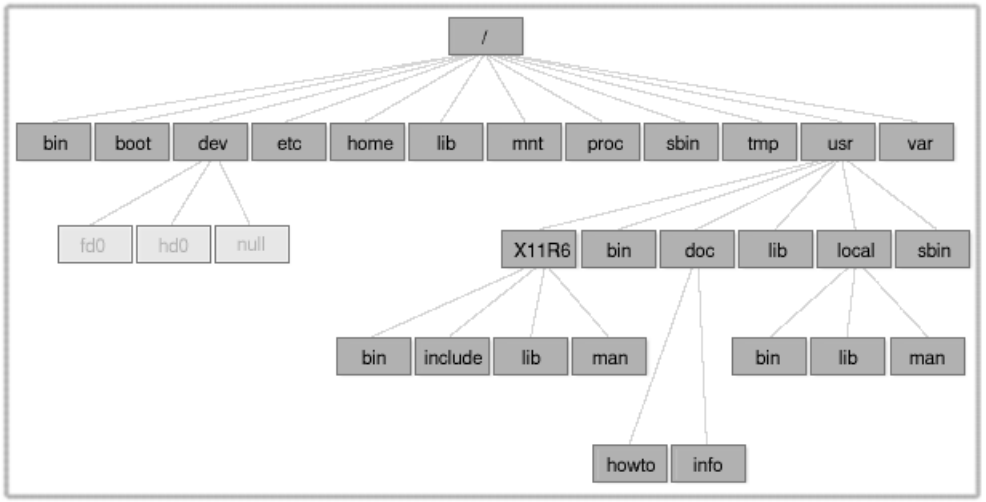
1. Datenstrukturen (allgemein)
2. Listen
   a. Beispiel: Stack und Queue
3. Bäume (nichtlineare Datenstrukturen)

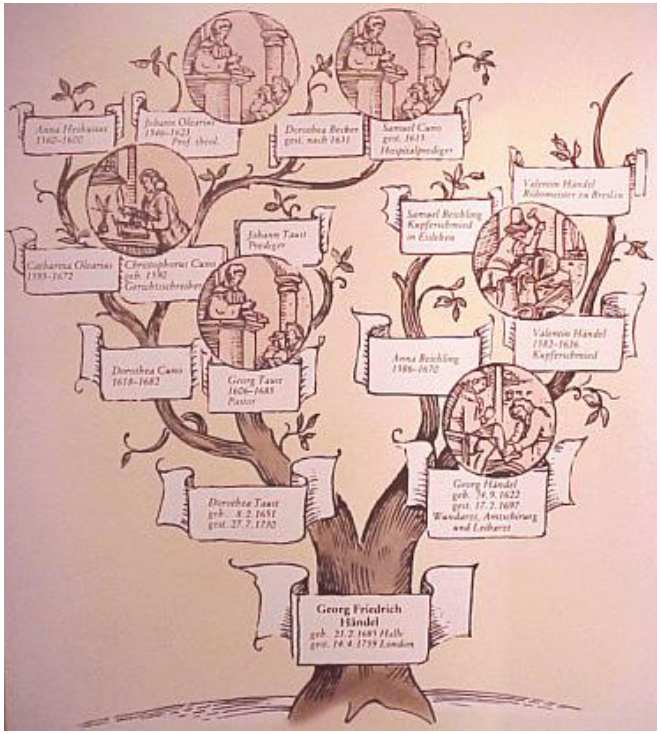
Nichtlineare Datenstrukturen

- Bisherige Datenstrukturen (Array, Liste, Stack, Queue) hatten lineare Struktur:
  - Elemente hatten eine Reihenfolge
  - jedes Element hatte max. einen Vorgänger bzw. Nachfolger
- Oft benötigt man Datenstrukturen, die nicht linear aufgebaut sind:
  - Darstellung von Produkten, die aus Einzelteilen bestehen und jedes dieser Einzelteile ist wiederum aus vielen Bestandteilen aufgebaut
  - Mitarbeiterverwaltung: Repräsentation von Hierarchien (ein Chef, mehrere Mitarbeiter, Mitarbeiter können ihrerseits Vorgesetzte von Anderen sein)
  - Datenstruktur zur Darstellung von familiären Beziehungen (jede Person hat genau zwei Elternteile)

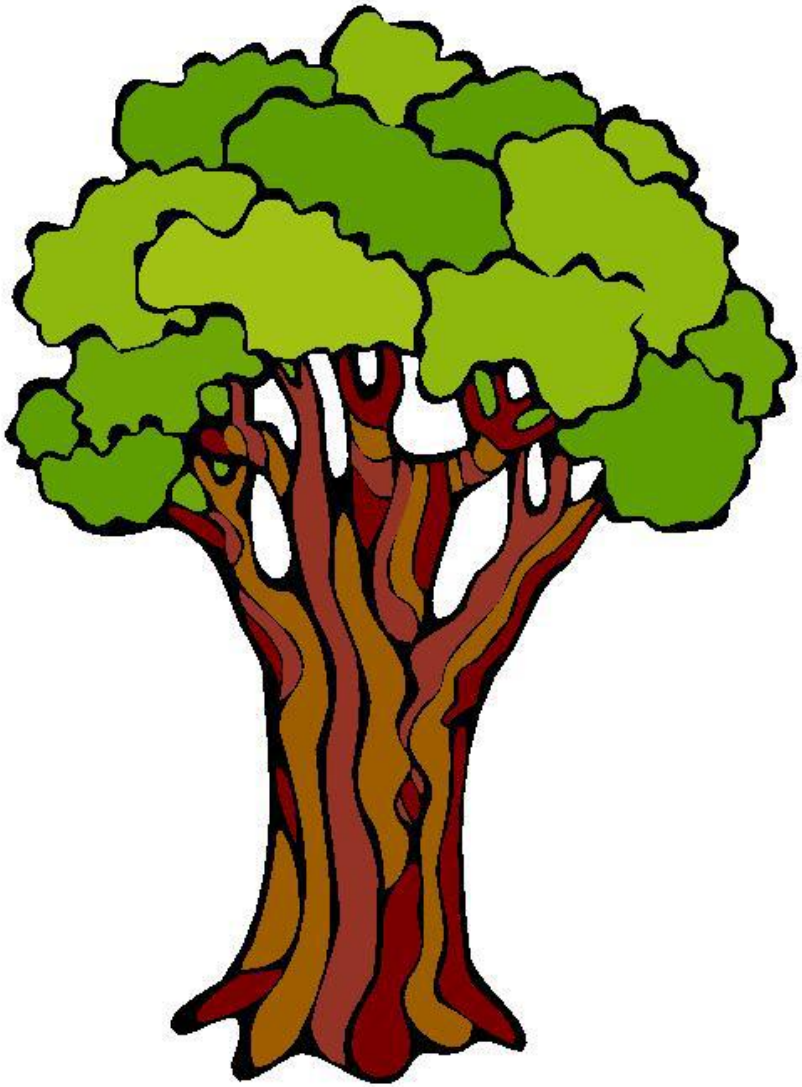
Beispiele für Bäume

- Stammbäume
- Verzeichnisbaum
- Vererbungshierarchie (in objektorientierten Sprachen)
- Webseiten (Document Object Model, DOM)
- …

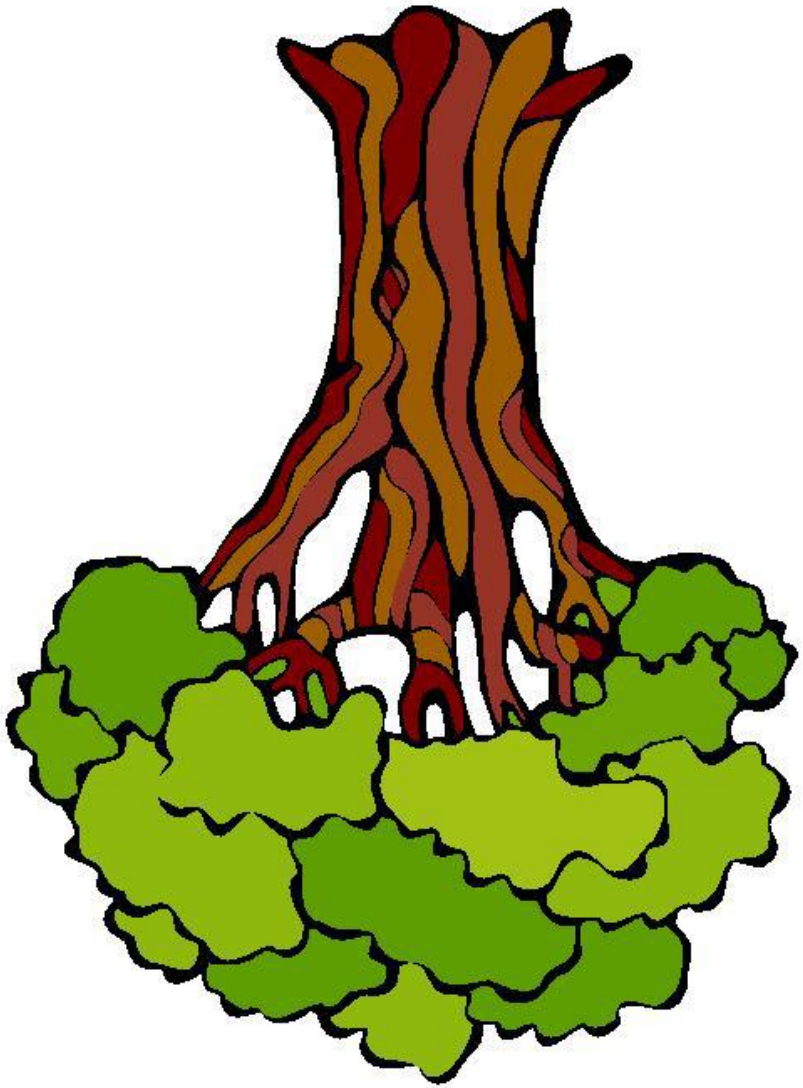




Bäume im realen Leben



Bäume in der Informatik



Definition: Baum

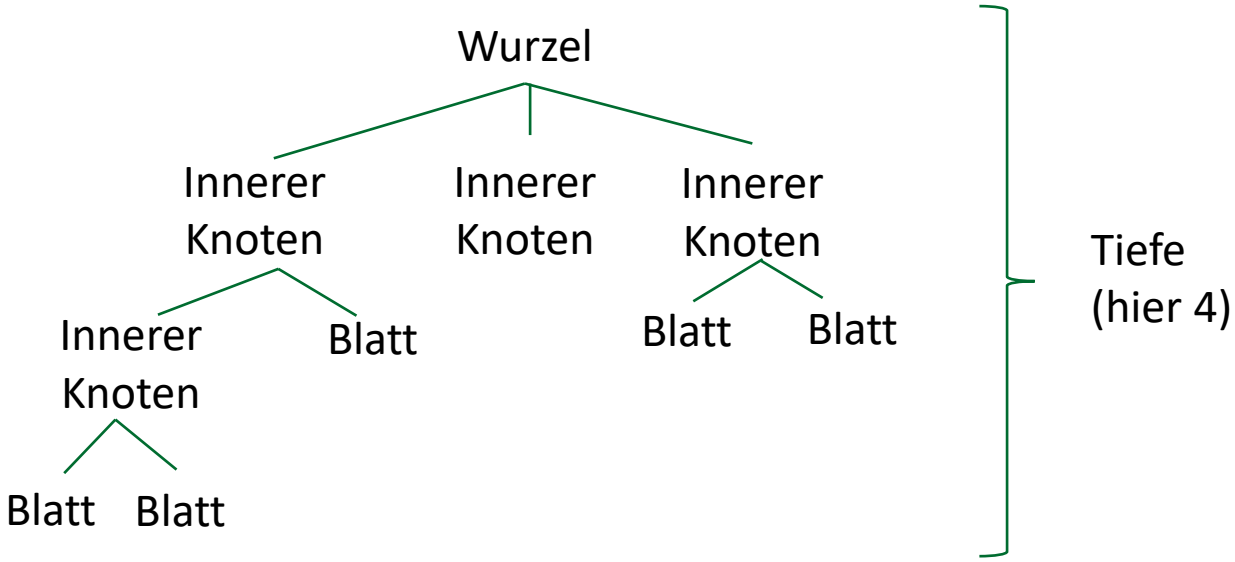
- Ein Baum besteht aus „Knoten“ (Einträge, beliebige Objekte).

Definition (Baum, Induktive Definition)

- Ein Baum ist entweder
  - ein leerer Baum,
  - ein Knoten, der mit einer Menge von Bäumen t_1, …, t_d verbunden ist.

- Es gibt diverse Varianten des Konzepts.
  - In der Vorlesung „Algorithmen und Datenstrukturen“ im Sommersemester wird das Thema im Detail behandelt.

Terminologie bei Bäumen (1)



Terminologie bei Bäumen (2)

- Baum: Menge von Knoten und Kanten
- Knoten: repräsentiert beliebiges Objekt
- Kante: Verbindung zwischen zwei Knoten
- Pfad: Folge unterschiedlicher, durch Kanten verbundener Knoten

- Wurzel: ausgezeichneter Knoten, der keine Vorgänger hat
- Blatt: Knoten ohne Nachfolger
- Elterknoten: Vorgänger eines Knotens
- Kind: Nachfolger eines Knotens
- Geschwister: Knoten mit gleichem Elterknoten

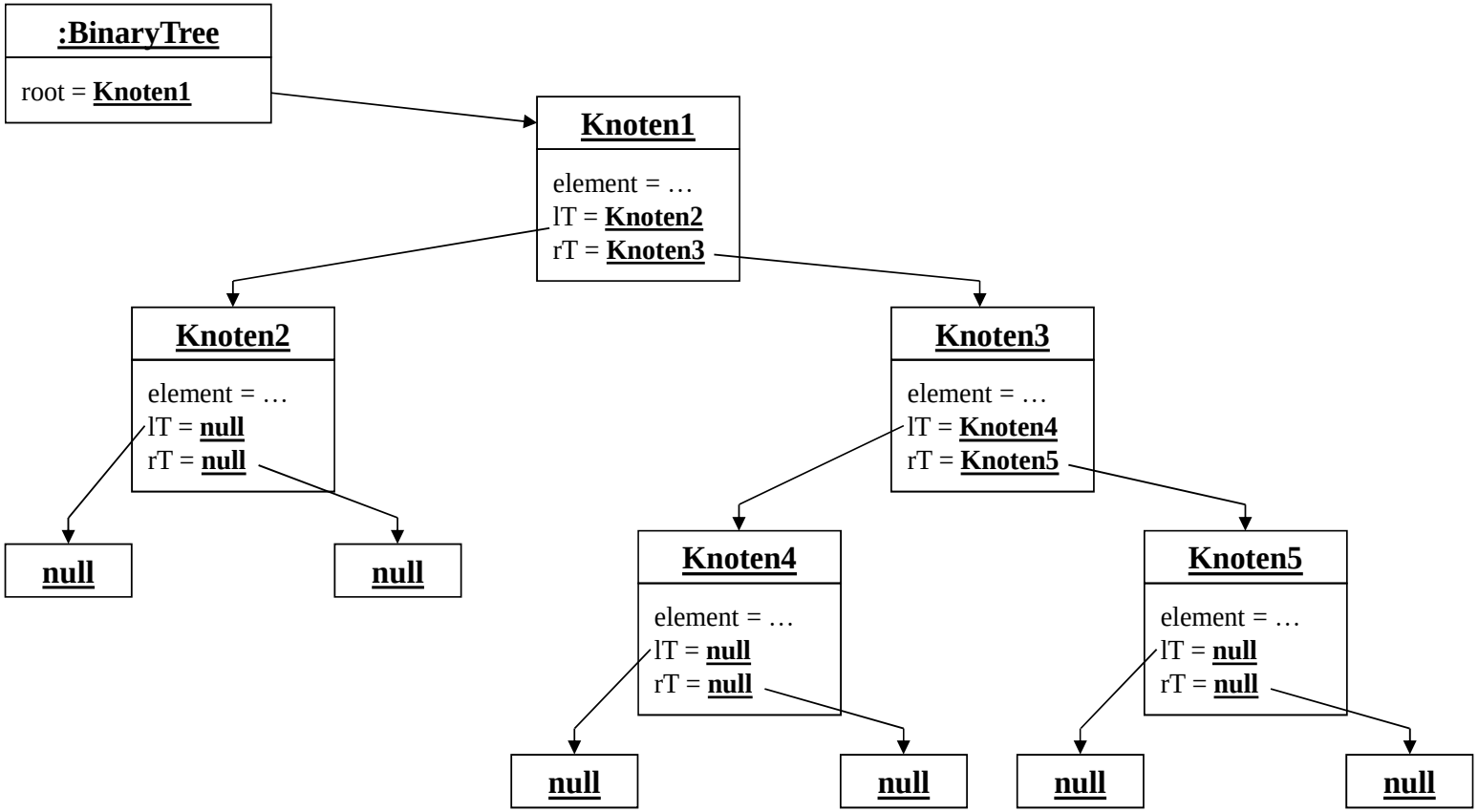
Binärbäume

- Wichtiger Spezialfall:
  - jeder Knoten hat höchstens zwei Kinder

Definition (Binärbaum, Induktive Definition)

- Ein Binärbaum ist entweder
  - ein leerer Binärbaum oder
  - ein Knoten mit zwei Binärbäumen als Kinder (linker und rechter Teilbaum).

Bäume: Beispiel



Beispiel: Binärbaum

```
public class BinaryTreeElement {
  private int value;
  private BinaryTreeElement left;
  private BinaryTreeElement right;

  … Konstruktoren etc.
}
```

Binärer Suchbaum

- Geordneter Baum mit dem Ziel Einträge schnell wieder zu finden.

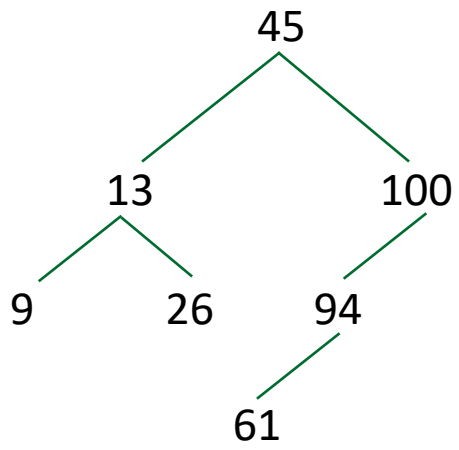
Definition (Binärer Suchbaum, Induktive Definition)

- Ein binärer Suchbaum ist entweder
  - leer oder
  - ein Knoten K mit zwei binären Suchbäumen als Kinder (linker und rechter Teilbaum). Dabei sind
    - die Werte aller Knoten des linken Teilbaumes kleiner oder gleich zum Wert des Knoten K,
    - die Werte aller Knoten des rechten Teilbaumes größer als der Wert des Knoten K.

Einfügen von Elementen in einen binären Suchbaum

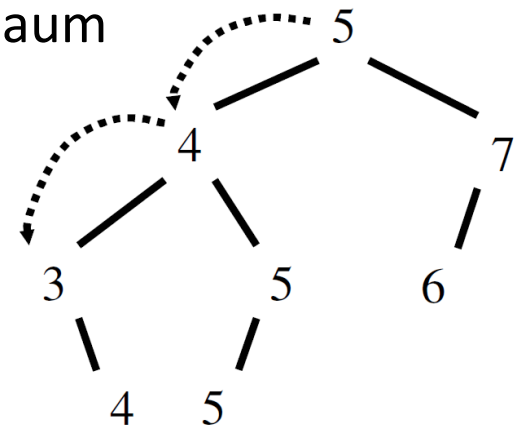
- Hinzufügen von x zum binären Suchbaum B
  - wenn B leer ist, erzeuge Knoten mit x
  - wenn x <= „Knoten“, füge x rekursiv zum linken Teilbaum hinzu
  - wenn x > „Knoten“, füge x rekursiv dem rechten Teilbaum hinzu

- Einfügen der Werte 45, 13, 100, 94, 26, 9 ,61



Suchen im binären Suchbaum

- Suche nach x im binären Suchbaum B:
  - wenn B leer ist, dann ist x nicht in B
  - wenn x = Wurzelelement gilt, dann haben wir x gefunden
  - wenn x < Wurzelelement, suche im linken Teilbaum
  - wenn x > Wurzelelement, suche im rechten Teilbaum
- Beispiel: Suche 3 im folgenden Baum



- Gibt es mehrere Knoten mit gleichem Wert, wird bei diesem Algorithmus offenbar derjenige mit der geringsten Tiefe gefunden.
- Suchzeit ist proportional zur Tiefe des Baums! Im besten Fall logarithmisch zur Tiefe.

Zusammenfassung

- Listen
  - lineare dynamische Datenstruktur
  - einfach- oder zweifachverkettete Listen
  - Spezialfälle: Stack und Queue

- Bäume
  - hierarchische dynamische Datenstruktur
  - Spezielfälle: Binäre Bäume und binäre Suchbäume

Prof. Dr. Sven Strickroth

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67
80538 München

Telefon: +49-89-2180-9300
sven.strickroth@ifi.lmu.de