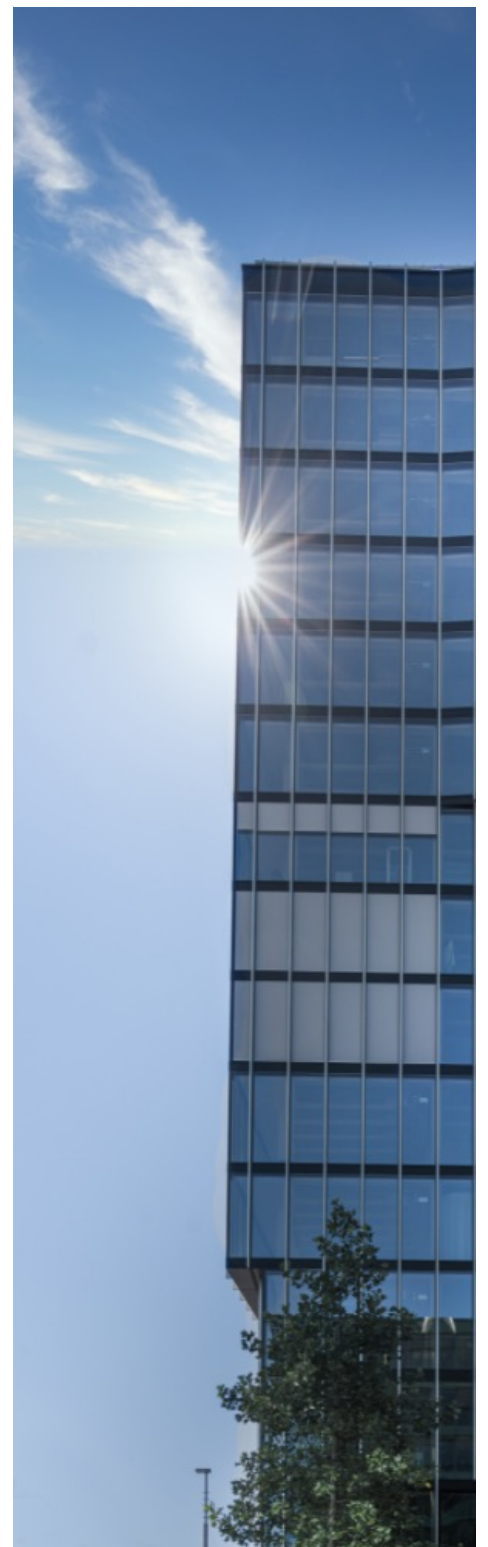


Algorithmen und Datenstrukturen

Datenstrukturen: Arrays, Listen, Queue und Stack

Roland Gisler / Ingmar Baetge



Inhalt

- Eigenschaften von Datenstrukturen
- Array (fixed-size Array, Sprachelement)
- Listen (einfach und doppelt verknüpft)
- Modellierung von Listen
- Stack
- Queue

Lernziele

- Sie kennen Eigenschaften von Datenstrukturen.
- Sie können die Komplexität von Operationen auf unterschiedlichen Datenstrukturen beurteilen.
- Sie kennen den Aufbau, die Eigenschaften und die Funktionsweise ausgewählter Datenstrukturen.
- Sie können Datenstrukturen exemplarisch selbst implementieren.
- Sie können abhängig von Anforderungen die geeigneten Implementationen von Datenstrukturen auswählen.

Eigenschaften von Datenstrukturen

Eigenschaften von Datenstrukturen - Übersicht

- ① **Reihenfolge / Sortierung**: In welcher Reihenfolge werden die Elemente abgelegt.
- ② **Operationen**: Welche Operationen stehen zur Verfügung.
- ③ **Statische oder dynamische Datenstruktur**: Kann die Datenstruktur ihre Grösse dynamisch verändern oder ist sie statisch.
- ④ **Explizite oder implizite Beziehungen**: Bestehen zwischen den Elementen explizite oder implizite Beziehungen.
- ⑤ **Zugriffsmöglichkeiten**: Besteht direkter oder nur indirekter Zugriff auf die einzelnen Datenelemente.
- ⑥ **Aufwand der Operationen**: Wie gross ist der Aufwand für die einzelnen Operationen, speziell in Abhängigkeit zur Datenmenge.

① Reihenfolge und Sortierung – 1/6

- Datenstrukturen als reine Sammlung: Die einzelnen Datenelemente sind darin ungeordnet abgelegt und die Reihenfolge ist nicht deterministisch.
 - Analogie: Steinhaufen.
- Datenstrukturen welche die Datenelemente in einer bestimmten Reihenfolge (z.B. in der Folge des Einfügens) enthalten und diese implizit beibehalten.
 - Analogie: Stapel oder Reihe von Büchern.
- Datenstrukturen welche die Elemente (typisch beim Einfügen) implizit sortieren / ordnen.
 - Analogie: Vollautomatisches Hochregallager
- Achtung: Auch abhängig von der Implementierung bzw. Nutzung!

② Operationen auf Datenstrukturen – 2/6

- Es gibt einige elementare Methoden, die auf Datenstrukturen angewendet werden können:
 - Einfügen von Elementen
 - Suchen von Elementen
 - Entfernen von Elementen
 - Ersetzen von Elementenin Datenstrukturen.
- Operationen in Abhängigkeit einer (optionalen) Reihenfolge oder Sortierung (natürlich oder speziell):
 - Nachfolger: Nachfolgendes Datenelement.
 - Vorgänger: Vorangehendes Datenelement.
 - Sortierung: Sortieren der Datenelemente nach Attributwerten.
 - Maxima und Minima: Kleinstes und grösstes Datenelement.

③ Statische vs. dynamische Datenstruktur – 3/6

- Eine **statische** Datenstruktur hat nach ihrer Initialisierung eine feste, unveränderlich Grösse. Sie kann somit nur eine beschränkte Anzahl Elemente aufnehmen.
 - Analogie: Getränkeflasche
 - Grösse der Flasche ist gegeben, ebenso maximaler Inhalt.
 - Die Flasche selber nimmt immer den selben Platz ein!
- Eine **dynamische** Datenstruktur hingegen kann ihre Grösse während der Lebensdauer verändern. Sie kann somit eine beliebige* Anzahl Elemente aufnehmen.
 - Analogie: Luftballon
 - Je nach Gasvolumen dehnt sich der Luftballon räumlich aus oder zieht sich wieder zusammen.
 - In leerem Zustand (fast) kein Platzbedarf.

*natürlich begrenzt durch den verfügbaren Speicher.

④ Explizite vs. implizite Beziehungen - 4/6

- Bei **expliziten** Datenstrukturen werden die Beziehungen zwischen den Daten von jedem Element **selber explizit** mit Referenzen festgehalten.
 - Analogie: Fahrradkette
 - Kettenglieder sind explizit miteinander verknüpft.
 - Jedes Kettenglied kennt seine zwei Nachbarglieder.
- Bei **impliziten** Datenstrukturen werden die Beziehungen zwischen den Daten **nicht** von den Elementen selber festgehalten.
 - Die Beziehungen werden quasi von «aussen» definiert, z.B. über eine externe Nummerierung (Index).
 - Analogie: Buchregal mit Büchern
 - Bücher stehen einfach (ggf. auch geordnet) nebeneinander.
 - Das Buch selber weiss nicht, wo es in der Reihe steht.

5 Direkter vs. indirekter/sequenzieller Zugriff – 5/6

- Bei **direktem** Zugriff hat man auf jedes einzelne Element direkten und unmittelbaren Zugriff.
 - Analogie: Buchregal mit Büchern.
 - Alle Bücher stehen nebeneinander im Regal.
 - Man kann jedes Buch **direkt** herausnehmen.
- Bei **indirektem** Zugriff hat man **keinen** direkten Zugriff auf bestimmte, einzelnen Datenelemente. Man kann allenfalls sequenziell ein Element nach dem anderen erhalten.
 - Analogie: Tellerstapel in der Mensa
 - Man kann «einen» Teller nehmen, aber keinen bestimmten.
 - Möchte man einen bestimmten Teller (oder alle), muss man alle Teller **sequenziell** umstapeln, bis der gewünschte Teller gefunden ist.

⑥ Aufwand von Operationen – 6/6

- Der **Aufwand** (Rechen- und Speicherkomplexität) variiert sowohl für die verschiedenen Operationen als auch (oft) in Abhängigkeit der enthaltenen Datenmenge in einer Datenstruktur.
- Meistens interessiert uns «nur» die Ordnung, also wie sich der Aufwand in Abhängigkeit zur Anzahl der Elemente verhält.
- Beispiele:
 - Buch auf einen Stapel legen (ungeordnet):
 $O(1)$ → Konstant
 - Einzelnes Buch in der Bibliothek alphabetisch einordnen:
im schlechtesten Fall **$O(n)$** → Linear
 - Eine unsortierte Menge von Büchern alphabetisch ordnen:
im schlechtesten Fall **$O(n^2)$** → Quadratisch (polynomiell)

Array

(Sprachelement, indexierte Reihung)

Array - Beispiele

■ Beispiel 1:

- Ein **char**-Array mit Platz für maximal 16 (**length**) Elemente.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

- Der Array ist «voll», alle Positionen sind belegt.
- Die Elemente sind sortiert eingefügt.

■ Beispiel 2:

- Ein **char**-Array mit Platz für maximal 8 (**length**) Elemente.

0	1	2	3	4	5	6	7
B	<leer>	A	M	I	<leer>	<leer>	<leer>

- Der Array hat noch vier freie Plätze.
- Die Elemente sind weder sortiert noch fortlaufend eingefügt.

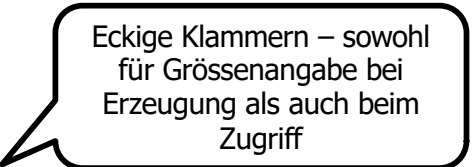
- Empfehlung: Leer Plätze mitten im Array möglichst **vermeiden!**

Array - Eigenschaften

- **Statische** Datenstruktur

- Grösse wird bei Initialisierung festgelegt:

Beispiel: `char[] demo = new char[8];`



Eckige Klammern – sowohl für Grössenangabe bei Erzeugung als auch beim Zugriff

- **Implizite** Datenstruktur

- Die einzelnen Elemente haben keine Beziehung untereinander bzw. keine Referenzen aufeinander.

- **Direkter** Zugriff

- Auf jedes Element kann über den Index direkt zugegriffen werden. `demo[0] = 'B';`

- **Reihenfolge:** Der Array behält die Positionen der Datenelemente (so wie sie zugewiesen/eingeordnet wurden) unverändert bei.

Array - Eigenschaften

▪ Indizierung

- Arrays der Grösse n sind in Java immer von $0 \dots n-1$ indiziert
- als Index wird in Java eine positive Integer-Zahl verwendet, d.h. die Grösse eines Arrays ist durch den Datentyp Integer begrenzt; ein Array kann maximal 2,147,483,647 Elemente beinhalten.

▪ Länge

keine Methode!

- Die Eigenschaft `arr.length` beinhaltet die Länge des Arrays (unabhängig davon, ob die Plätze auch alle genutzt werden)

▪ Arrays sind Objekte

- Darum müssen wir Arrays mit `new` instanzieren!
- Arrays können als Objektreferenz z.B. an eine Methode übergeben werden.

Array - Eigenschaften

▪ Arrays sind ein Sprachelement von Java

- Im Gegensatz zu anderen Datenstrukturen (z.B. List) sind Arrays ein echtes Sprachelement von Java, d.h. sie haben ihre eigene Syntax:

```
int[] numbers = new int[10];  
Person[] group = new Person[5];
```

▪ Arrays können aus beliebigen Typen bestehen

- Alle Datentypen, auch eigene Klassen können als Array genutzt werden

▪ Mehrdimensionale Arrays

- Arrays können auch mehrere Dimensionen haben (siehe Übung)

```
char[][] tictactoe = new char[3][3];
```


Array – Code-Beispiel

Die Argumente bei
Programmstart werden
auch als Array übergeben

```
public class ... {  
    public static void main(final String[] args) {  
        // Erzeugung eines Arrays  
        char[] demo = new char[8];  
  
        // Zugriff auf Array-Elemente  
        // Index von 0 ... n-1  
        demo[0] = 'B';  
        demo[1] = 'G';  
  
        // Kurzschreibweise für Initialisierung  
        char[] demo2 = {'A', 'B', 'C'};  
  
        // Länge  
        int l = demo.length;  
  
        // Was passiert hier?  
        char elem = demo[8];  
  
        ...  
    }  
}
```

demo

0	1	2	3	4	5	6	7
B	G	<leer>	<leer>	<leer>	<leer>	<leer>	<leer>

demo2

0	1	2
A	B	C

8


ArrayIndexOutOfBoundsException

Array – Suchen eines Elementes - G

- Fall **1**: Daten nicht sortiert, aber fortlaufend (ohne Lücken) befüllt:
Wir müssen den Array sequenziell durchsuchen.


Der Aufwand beträgt: **$O(n)$**

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>



- Fall **2**: Daten sortiert und fortlaufend befüllt:
Wir können **binär** Suchen (siehe nächste Folie), der Aufwand beträgt somit: **$O(\log n)$**

0	1	2	3	4	5	6	7
A	B	G	I	M	<leer>	<leer>	<leer>



Beispiel für binäre Suche in 8 Elementen:

$\log_2 8 = 3$, somit maximal **drei** Vergleiche notwendig!

Binäres Suchen - Algorithmus

- Wichtige Voraussetzung:
Eine **sortierte** Datenmenge!
- Algorithmus:
 1. Datenmenge in der Mitte teilen.
 2. Auf Basis des Trennelementes entscheiden, ob man in der linken oder rechten Hälfte weitersucht.
 3. Algorithmus **rekursiv** mit der ausgewählten Hälfte wiederholen.
 4. Algorithmus endet, wenn das Element gefunden wurde, oder wenn nur noch ein Element vorhanden ist.

Suche nach Element **3**:

Element in der Mitte (**4**) prüfen

1	2	3	4	5	6	7
---	---	---	---	---	---	---

4 ist **nicht** das gesuchte Element und **grösser** als 3.

→ wir nehmen die **linke** Hälfte und wiederholen damit den Algorithmus:
Element in der **Mitte** (**2**) prüfen

1	2	3	4	5	6	7
---	---	---	---	---	---	---

2 ist **nicht** das gesuchte Element und **kleiner** als 3.

→ wir nehmen die **rechte** Hälfte und wiederholend damit den Algorithmus:
Element in der **Mitte** (**3**) prüfen

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Gesuchtes Element 3 gefunden!

Array – Anhängen bzw. Einfügen eines Elementes - C

- Fall **1**: Daten nicht sortiert, aber fortlaufend befüllt:

Trick: Wir merken uns den **Index** des jeweils nächsten freien Platzes! Dann beträgt der Aufwand: **$O(1) \rightarrow$ Konstant**

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
B	G	A	M	I	C	<leer>	<leer>

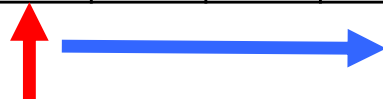
Ohne diesen Trick: $O(n)$

- Fall **2**: Die Daten sind sortiert:

Wir können zwar binär mit $O(\log n)$ die **Position** suchen, müssen dann aber die restlichen Elemente mit $O(n)$ nach rechts schieben!

Aufwand: $O(\log n) + O(n) \rightarrow \mathbf{O(n)}$

0	1	2	3	4	5	6	7
A	B	G	I	M	<leer>	<leer>	<leer>



0	1	2	3	4	5	6	7
A	B	C	G	I	M	<leer>	<leer>

Array – Entfernen eines Elementes - G

- Fall **1**: Daten nicht sortiert, aber fortlaufend befüllt:

Wir müssen den Array sequenziell mit $O(n)$ durchsuchen und können die Lücke mit dem letzten Element mit $O(1)$ schliessen.

Aufwand: $O(n) + O(1) \rightarrow O(n)$

0	1	2	3	4	5	6	7
B	G	A	M	I	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
B	I	A	M	<leer>	<leer>	<leer>	<leer>

- Fall **2**: Die Daten sind sortiert:

Wir können zwar binär mit $O(\log n)$ suchen, müssen die entstehende Lücke aber durch Linksrücken mit $O(n)$ schliessen.

Aufwand: $O(\log n) + O(n) \rightarrow O(n)$

0	1	2	3	4	5	6	7
A	B	G	I	M	<leer>	<leer>	<leer>

0	1	2	3	4	5	6	7
A	B	I	M	<leer>	<leer>	<leer>	<leer>

Array - Bilanz

- Arrays sind eigentlich gar nicht mal so schlecht?
- Positiv:
 - Tatsächlich sehr effizient und sehr schnell.
 - Direkter Zugriff, einfach.
- Negativ:
 - Array sind statisch! Konstanter Platzbedarf, unabhängig vom effektiven Füllgrad.
 - Arrays sind funktional relativ primitiv, z.B. beim Einfügen müssen die Elemente verschoben werden.
 - Wichtig: Arrays sind (in Java) nicht wirklich kompatibel mit Generics, darum werden Collections meistens bevorzugt.

Hinweise zur Verwendung von Arrays in Java

- Arrays sind in Java ein Sprachelement. Beispiele:

```
int[] numbers = new int[10];  
Person[] group = new Person[5];
```

- **Achtung:** Man kann zwar Arrays von generischen Typen deklarieren, aber keine Arrays von generischen Typen erzeugen!
 - Arrays und Generics «vertragen» sich **nicht** wirklich!
 - Gründe für diese Situation sind die Realisierung von Generics über «type erasure» und die Invarianz der Generics (typ zur Compiletime) vs. der Covarianz der Arrays (Runtime).





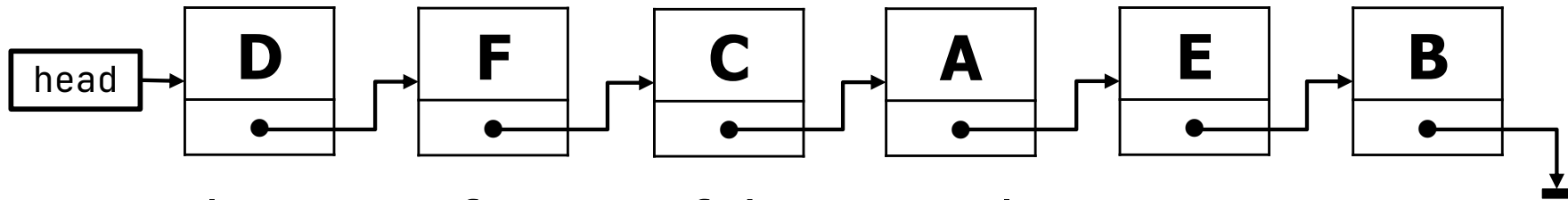
Verwendung von Arrays - Empfehlung

- Arrays sind statische Datenstrukturen, darum sollten sie nur verwendet werden, wenn die Datenmenge klar **beschränkt**, von Anfang an **bekannt**, und eher **klein** ist.
 - Arrays sind effizient, wenn man nur wenige, oder nur elementare Datentypen ablegen muss.
 - Datentypen haben eine feste Grösse, und können somit in einer Reihung direkt im Array abgelegt werden. Technisch kann mit dem Index direkt die Speicheradresse berechnet werden.
 - In **allen anderen Fällen** sind Collections **vorzuziehen**, da sie wesentlich objektorientierter sind und es Implementationen gibt, die einen direkten Zugriff per Index erlauben (z.B. **ArrayList**).
- ➔ Vermeiden Sie **unbedingt** Arrays auf Schnittstellen!

Listen

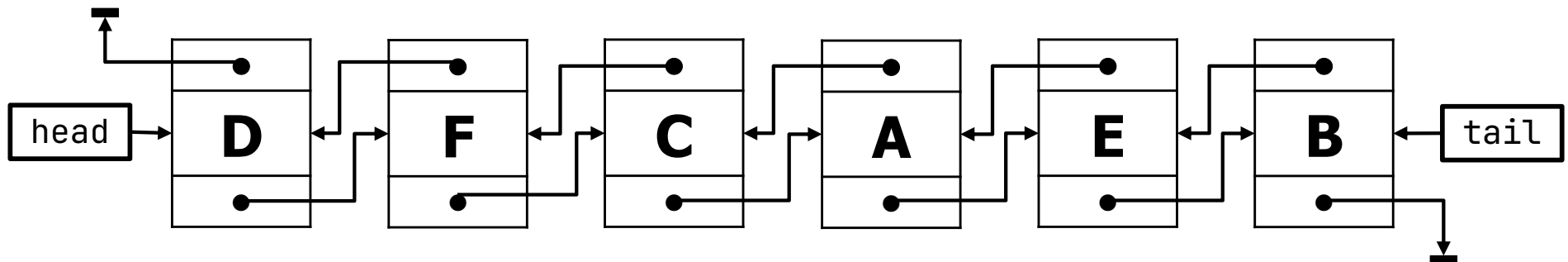
Einfach und doppelt verkettete Listen

- Visualisierung einer **einfach** verketteten Liste:



- Es gibt eine Referenz auf das erste Element ➔ **head**.
- Jedes Element kennt seinen direkten Nachfolger.

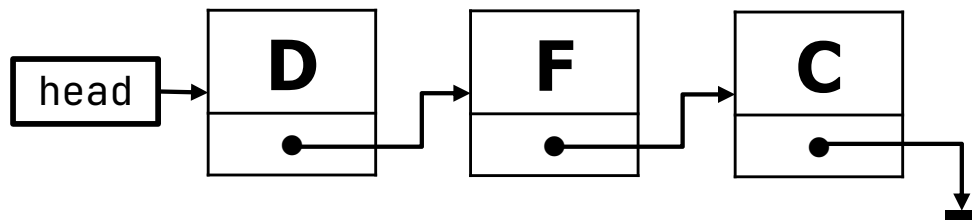
- Visualisierung einer **doppelt** verketteten Liste:



- Es gibt Referenzen auf das erste (➔ **head**) und das letzte (➔ **tail**) Element in der Liste.
- Jedes Element kennt seinen direkten Vorgänger **und** Nachfolger.

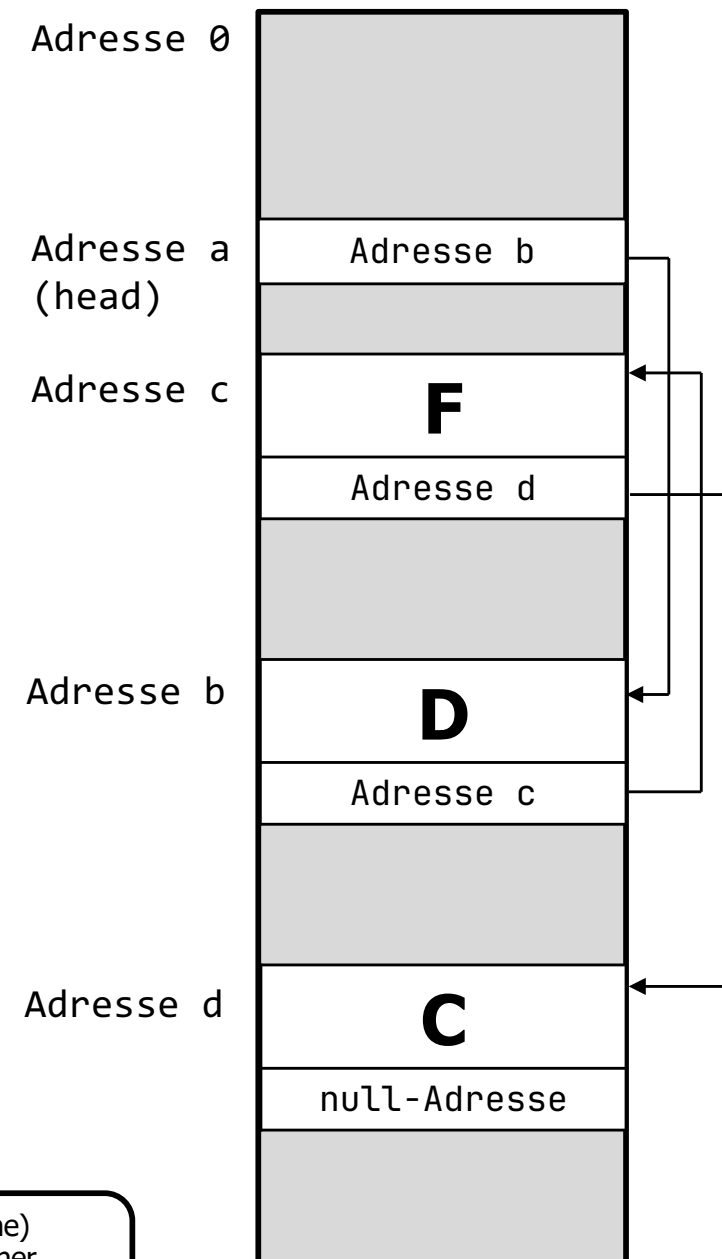
Listen und Speicher

- Listen werden in den meisten Darstellungen als Ketten dargestellt, bei denen die Elemente nebeneinander stehen – im tatsächlichen Speicher können diese Elemente aber willkürlich verteilt sein.



typische logische Darstellung
einer verketteten Liste

tatsächliche (physische)
Organisation im Speicher
(in Java haben wir keinen Zugriff
auf echte Speicheradressen!)

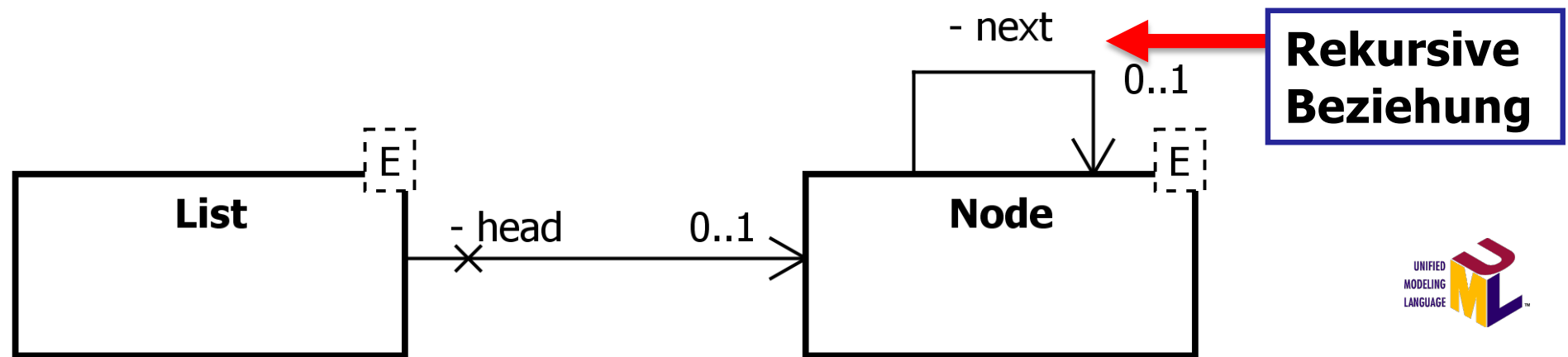


Modellierung von Listen

- Listen werden typisch mit **zwei** Klassen modelliert:
 - **Erste** Klasse (z.B. «**List**») repräsentiert die **Liste** selber:
 - Enthält die Referenz auf das **erste** Element (**head**).
 - Hilfsattribute z.B. für die Anzahl enthaltener Elemente.
 - Methoden für die verschiedenen Operationen.
 - **Zweite** Klasse repräsentiert einen Behälter für die **Elemente** und wird häufig als «**Knoten**» oder «**Node**» bezeichnet:
 - Enthält je nach Listentyp (einfach/doppelt) ein oder zwei Referenzen auf den Vorgänger bzw. den Nachfolger.
 - Enthält Attribut(e) für die effektiv enthaltenen Daten.
- Bei der Implementation mit Java kann das Datenattribut generisch sein und somit für beliebige Typen parametrisiert werden.
 - ➔ siehe Implementationen des Java Collection Frameworks.

Konzeptionelles Modell: Einfach verkettete Liste

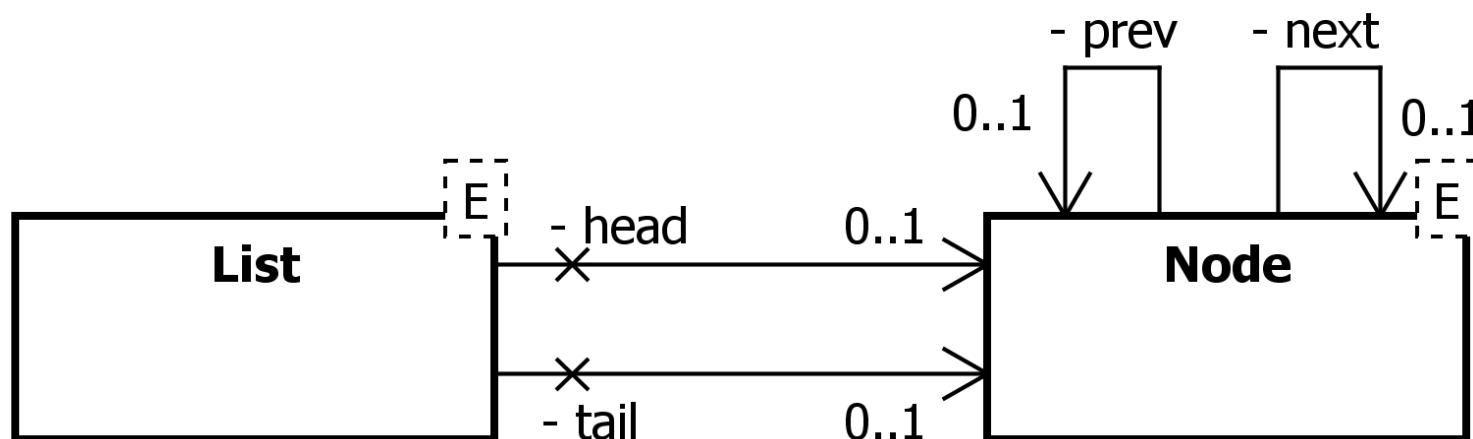
- Jeder **Node** (Element) enthält eine Referenz auf seinen Nachfolger (**next**), es resultiert somit eine rekursive Beziehung.
 - Das Modell enthält somit einen (unproblematischen) Zyklus.
- Hinweis: Das letzte Element (das somit keinen Nachfolger mehr hat) erhält als Referenz typischerweise den Wert **null**.



- Die Daten werden im **Node** gespeichert.
- Die **List** hält die Referenz auf das erste Element (**head**).

Konzeptionelles Modell: Doppelt verkettete Liste

- Analog zur einfach verketteten Liste, der **Node** (Element) hat nun aber **zwei** Referenzen: Je eine auf seinen unmittelbaren Vorgänger (**prev**) **und** seinen Nachfolger (**next**).



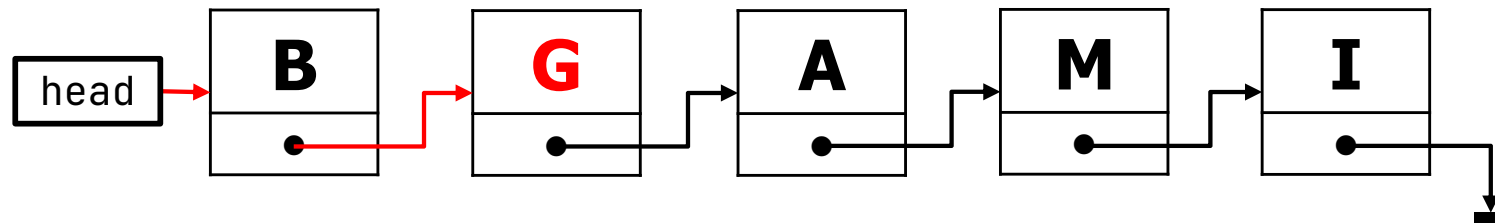
- Die **List** enthält nur sowohl die Referenz auf das erste (**head**) und das letzte (**tail**) Element.

Listen - Eigenschaften

- **Dynamische** Datenstruktur
 - Die Grösse der Datenstruktur passt sich der Anzahl der zu speichernden Elemente an und ist somit dynamisch.
- **Explizite** Datenstruktur
 - Die Elemente haben explizite Beziehungen untereinander.
 - Jedes Element kennt seinen Nachfolger (einfach verkettete Liste) und ggf. auch den Vorgänger (doppelt verkettete Liste)
- **Nur indirekter** Zugriff
 - Auf die Elemente kann beginnend vom Head aus **nur** sequenziell vorwärts (einfach verkettete Liste), bzw. auch vom Tail aus rückwärts (doppelt verkettete Liste) zugegriffen werden.
- **Reihenfolge:** Die Liste behält die Positionen der Datenelemente so wie sie eingefügt bzw. zugewiesen werden.

Listen – Suchen eines Elementes - G

- Da wir keinen direkten Zugriff haben (sondern nur sequenziell) beträgt der Aufwand für die Suche eines Elementes in einer Liste grundsätzlich **$O(n)$** .
 - Unabhängig davon, ob sortiert oder unsortiert.
 - Unabhängig davon, ob einfach oder doppelt verkettet.

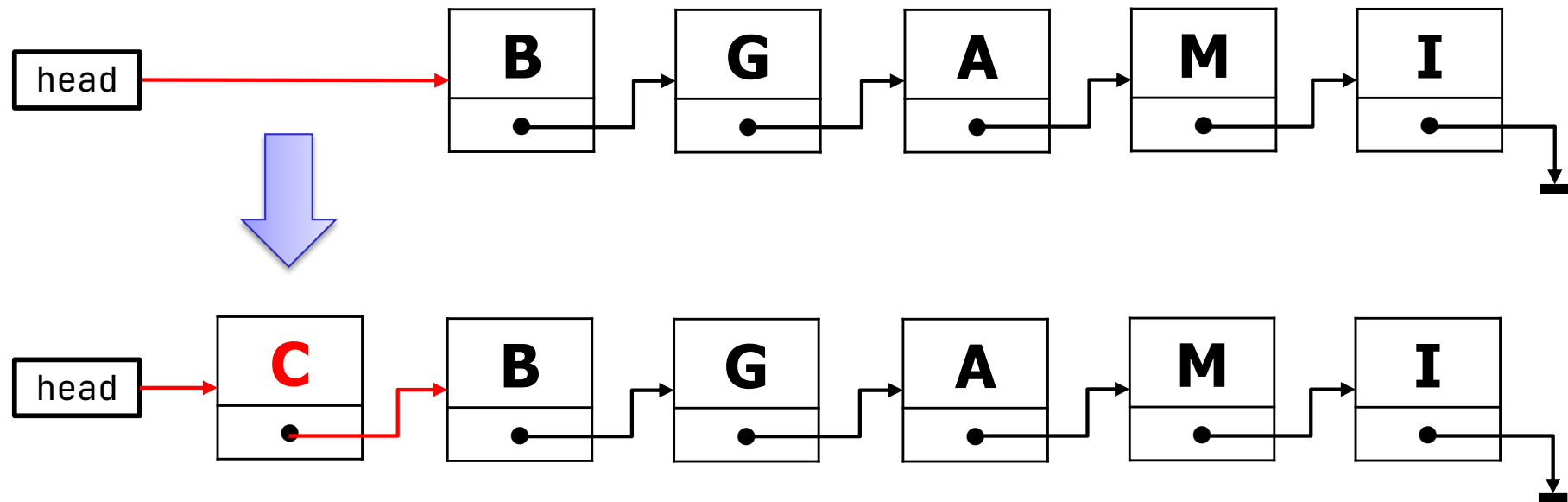


- Die Suche lässt sich aber bei sortierten Listen mit zusätzlichen Hilfsmitteln (→ Skiplisten) beschleunigen.
 - Damit wird ebenfalls $O(\log n)$ möglich.

Unsortierte Listen – Ergänzen eines Elementes - C

- **Einfach** verkettete Liste:

Da wir mit dem Head eine Referenz auf das erste Element haben, können neue Element am Anfang einfach und schnell eingefügt werden. Der Aufwand beträgt: **$O(1)$**

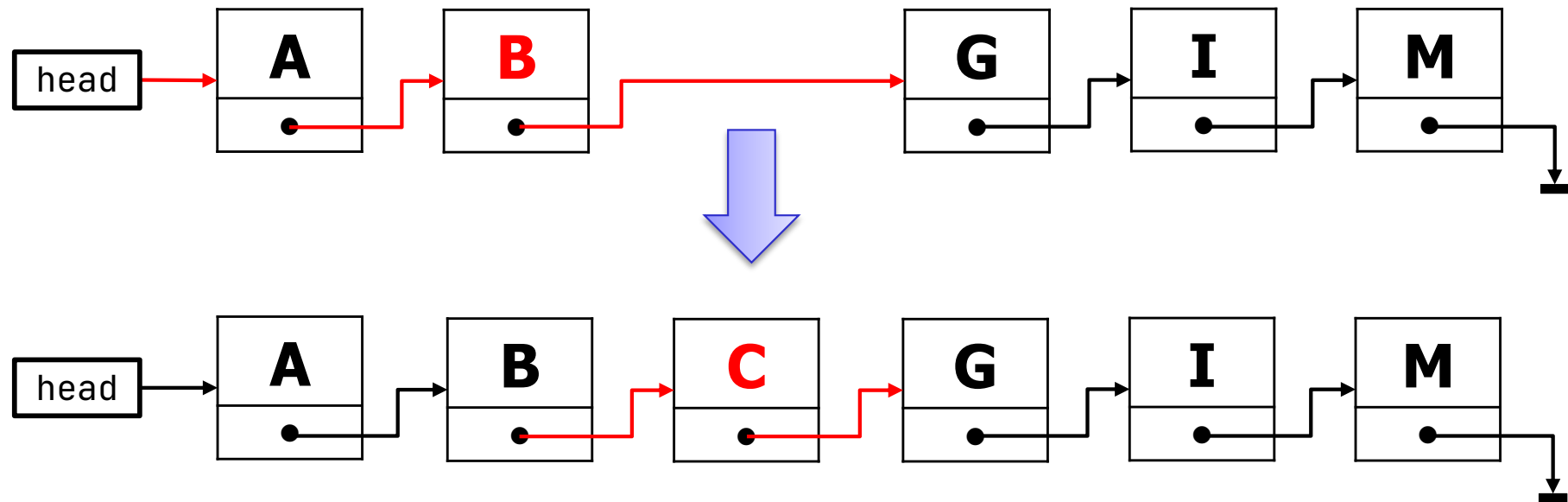


- Bei **doppelt** verketteter Liste:

Analog kann zusätzlich auch am Ende der Liste (tail) mit Aufwand **$O(1)$** eingefügt bzw. angehängt werden.

Sortierte Listen – Einfügen eines Elementes - C

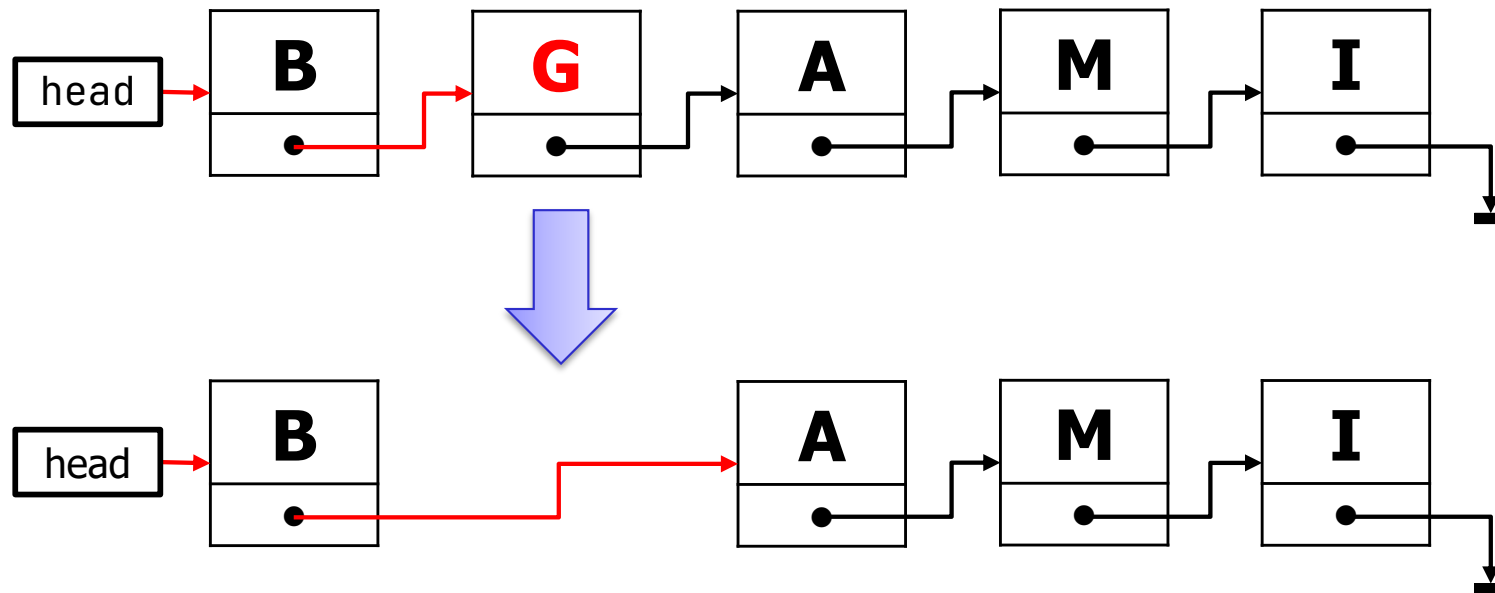
- Wir müssen sequenziell die richtige **Position** (**B**) suchen und ein neues Element einfügen, das Verschieben der restlichen Elemente **entfällt** hingegen! Der Aufwand ist trotzdem: **$O(n)$**



- Aufwand für einfach und doppelt verkettete Liste identisch.

Listen – Entfernen eines Elementes - G

- Wir müssen sequenziell das gewünschte Element **suchen** und es aus der Liste entfernen. Auch hier ist **kein** Nachrücken von Elementen notwendig. Aufwand (bedingt durch Suche): **$O(n)$**



- Aufwand ist für einfach und doppelt verkettete Liste identisch.



Listen – Vorteile gegenüber Arrays

- Der Hauptvorteil von Listen ist, dass sie **dynamisch** sind:
Sie können eine beliebige Datenmenge aufnehmen, belegen selber aber keinen «festen» Platz, sondern wachsen und schrumpfen mit der Datenmenge mit.
 - Prädestiniert für grosse, stark variierende Datenmengen.
- Der Aufwand für das **reine** Einfügen in eine Liste ist an jeder beliebigen Position (wenn diese gefunden ist) konstant und schnell.
- Listen sind als Datenstrukturen objektorientiert implementiert, unterstützen Generics, und können dank der vorhandenen Interfaces je nach Situation/Anwendungsfall durch optimale(re) Implementationen ausgetauscht werden.

Stack

Stack

- Ein Stack ist eine Datenstruktur, der Elemente als «Stapel» speichert:
 - **push(E e)**: Neue Elemente werden immer oben auf den Stapel abgelegt.
 - **E pop()**: Es kann jeweils nur das oberste Element entnommen werden.
- Semantik: LIFO – **Last In First Out**
(oder FILO – **First In Last Out**)
- Analogie: Tellerstapel
- Einsatz (Beispiele):
 - Datenablage bei Funktionsaufrufen.
 - Umkehren der Reihenfolge.



Bild: www.zlb.de

Stack – Aufwand der Operationen

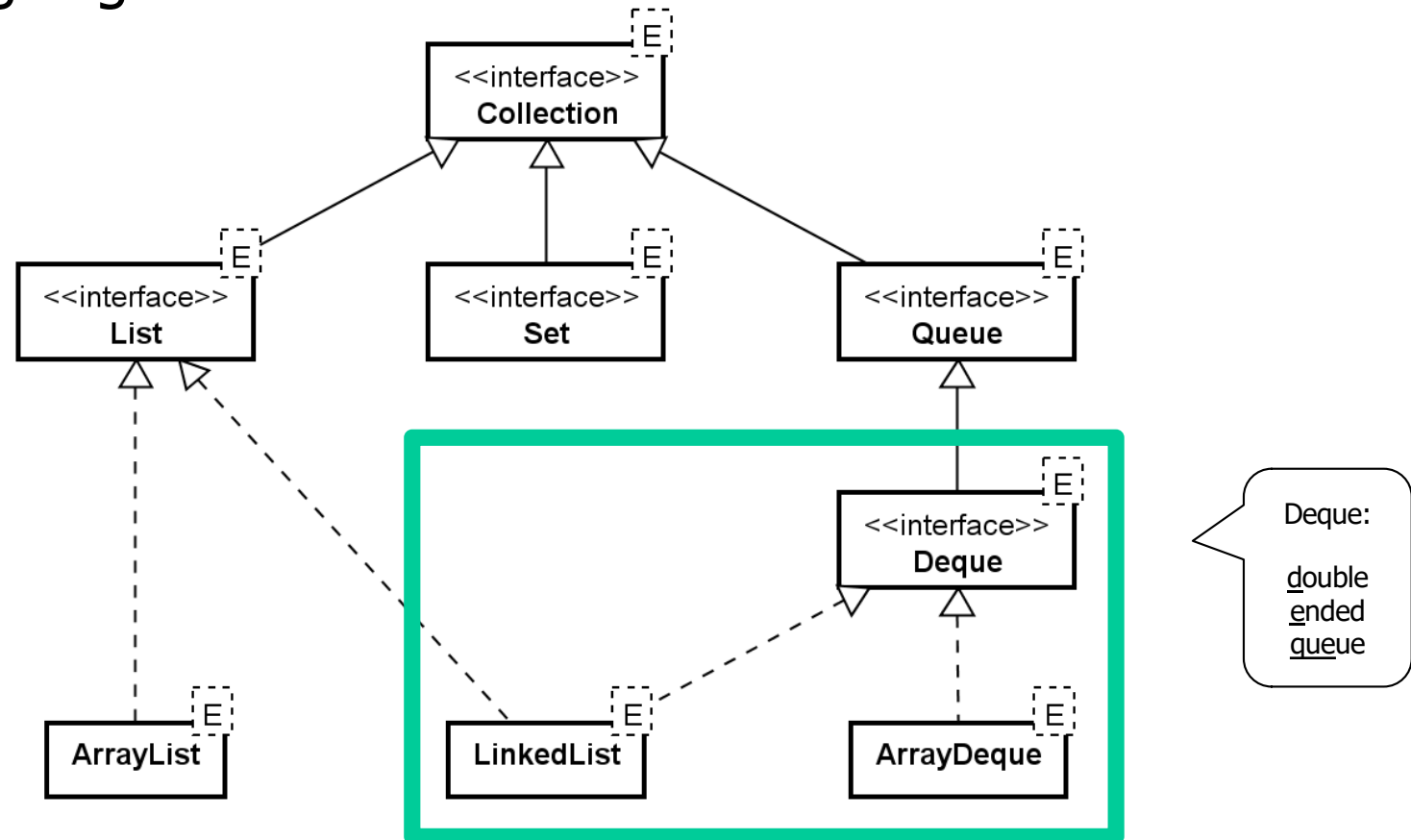
- Implementation mit **Array**:
 - Hinweis: Index des jeweils letzten Elementes wird gespeichert.
 - **push()**: Anhängen am Ende, Aufwand **$O(1)$** .
 - **pop()**: Entnehmen am Ende, Aufwand **$O(1)$** .
 - Implementation mit **Liste**:
 - Hinweis: Eine einfach verkettete Liste reicht aus.
 - **push()**: Einfügen am Anfang der Liste, Aufwand **$O(1)$** .
 - **pop()**: Entnehmen am Anfang der Liste, Aufwand **$O(1)$** .
 - Bei beiden Implementationen ist der Aufwand konstant und somit unabhängig von der Datenmenge!
- ➔ Welche Variante ist besser?

Stack – Implementation mit Liste oder mit Array?

- Implementation mit Array:
 - Man merkt sich jeweils den Index des letzten Elementes.
 - Array ist statisch, Grösse somit beschränkt.
 - Maximaler Platz ist immer belegt, weil bereits reserviert.
 - Dadurch ist ein **maximal schnelles** Einfügen möglich!
- Implementation mit Liste:
 - Einfach verkettete Liste reicht aus.
 - Ein leerer Stack benötigt fast keinen Platz.
 - Grösse dynamisch und nur durch Speicher begrenzt.
 - Speicheranforderung für neue Elemente notwendig, darum im Vergleich zum Array leicht langsamer!
- Dass in vielen Programmiersprachen eine **StackOverflow**-Exception/Fehler (o.ä.) existiert, bedeutet somit was?

Java - Stack mit Bibliotheksklassen

- Welche Klassen und Interfaces sind für Stack-Implementation bzw. Semantik geeignet?



- Wir haben also die Wahl!
(Nebenbei: Die Array-Variante ist auch dynamisch implementiert!)

Beispiel für fachlichen Einsatz eines Stacks*

- Wer kennt noch die legendären Taschenrechner von Hewlett Packard mit «reverser polnischen Notation» (RPN), auch als Postfix-Notation bekannt?
- Funktionsweise:
 - Zahlen werden auf Stack abgelegt (push)
 - Operationen konsumieren die benötigte Anzahl Argumente (pop) und legen das Resultat wieder auf dem Stack ab (push).



- Beispiel:
 - Berechnung des Ausdruckes:
 $(3 + 4) * 2$
 - Eingabe auf Taschenrechner:
3<Enter> 4<Enter> + 2<Enter> *

Verlauf des Stacks:

	t →						
3							
2			[+]			[*]	
1		4	4		2	2	
0	3	3	3	7	7	7	14

Queue

Queue

- Die Queue ist eine Datenstruktur, welche Elemente in einer (Warte-)Schlange speichert.
 - **enqueue(E e)** oder **offer(E e)**:
Ein Datenelement am **Ende** der Queue anhängen.
 - **E dequeue()** oder **poll()**:
Ein Datenelement am **Anfang** der Queue entnehmen.
- Semantik: FIFO - **F**irst **I**n **F**irst **O**ut
- Analogie: Warteschlange an Kasse.
- Einsatz (Beispiele):
 - Zwischenspeicherung von Daten(-strömen).
 - Tastaturpuffer, Unix-Pipe.

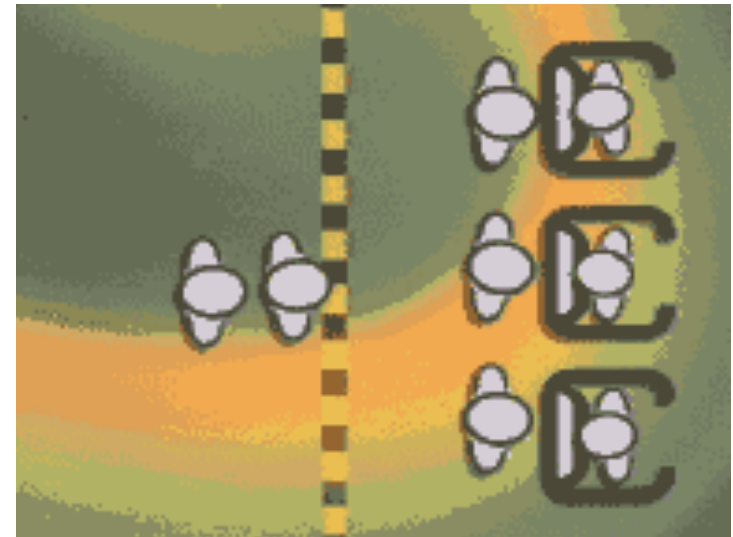


Bild: www.ku-eichstaett.de

Queue – Aufwand der Operationen

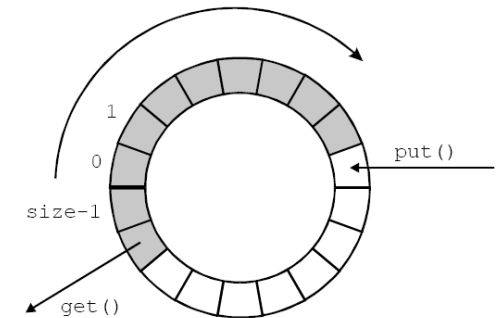
- Implementation mit Liste:

- Man verwendet eine doppelt verkettete Liste, damit man schnellen Zugriff auf head **und** tail hat.
- **enqueue()**: Einfügen am Ende der Liste, Aufwand **$O(1)$** .
- **dequeue()**: Entnehmen am Anfang der Liste, Aufwand **$O(1)$** .

- Trickreiche Implementation mit (statischem) Array:

- Man implementiert einen «Ringbuffer» so, dass man die Elemente **nicht** verschieben muss!
Es gibt je einen Index für das erste und das letzte Element welche «rotieren». Somit gilt:

- **enqueue()**: Einfügen «am Ende» (**put**), Aufwand **$O(1)$** .
- **dequeue()**: Entnehmen «am Anfang» (**get**), Aufwand **$O(1)$** .
- Wichtig: Die Indexe dürfen sich nicht gegenseitig überholen!

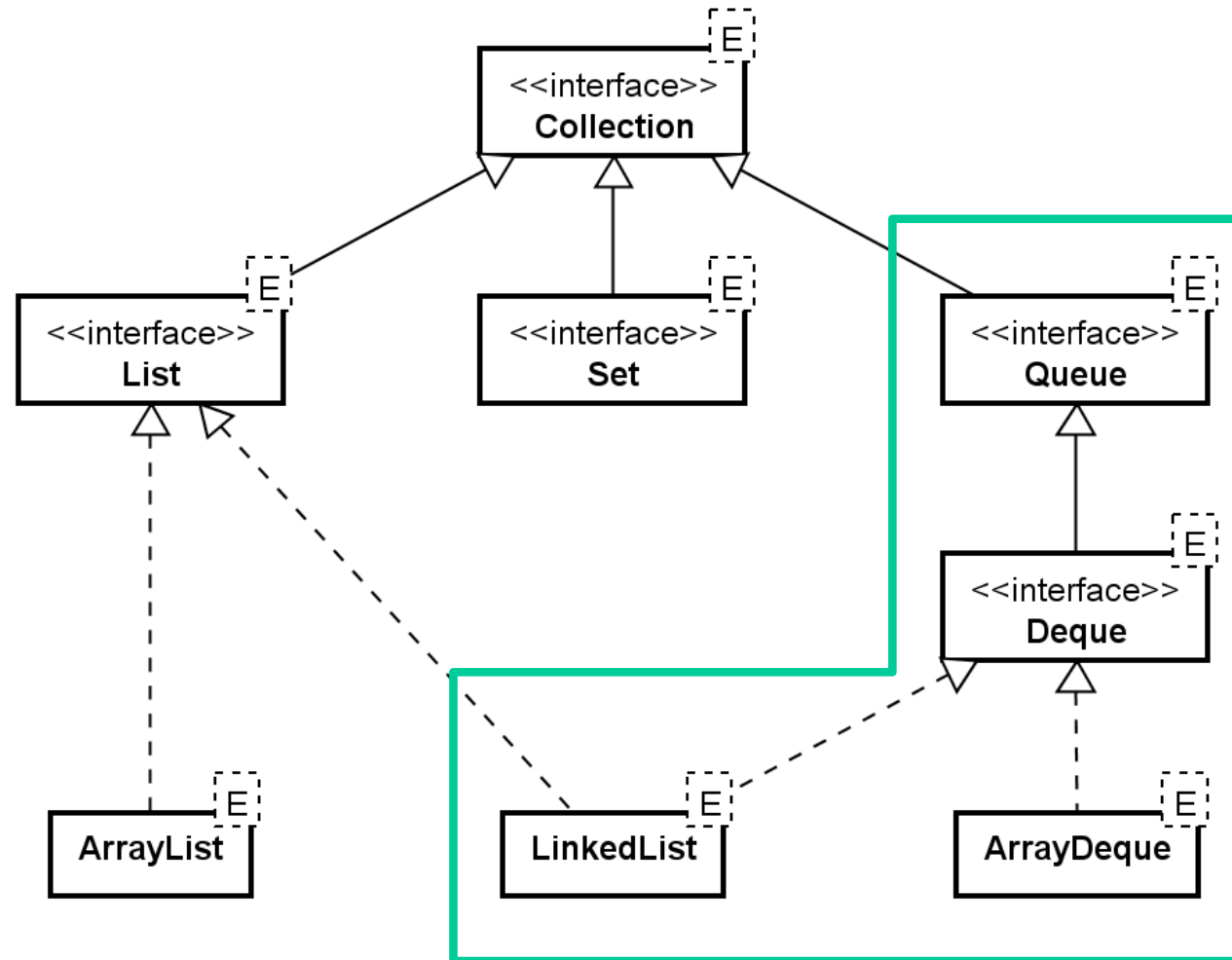


Queue – Implementation mit Liste oder mit Array?

- Implementation mit Array:
 - Trickreiche Implementation als logischer Ringbuffer!
 - Array ist statisch, maximale Grösse somit festgelegt.
 - Maximaler Platz immer belegt und reserviert.
 - Darum wieder sehr schnell (vgl. → Stack)!
- Implementation mit Liste:
 - Doppelt verkettete Liste notwendig.
 - Leere Queue benötigt fast keinen Platz.
 - Grösse dynamisch und nur durch Speicher begrenzt.
 - Speicheranforderung für neue Elemente notwendig, darum im Vergleich zum Array wieder etwas langsamer!

Java - Queue mit Bibliotheksklassen

- Welche Klassen und Interfaces von Java sind für Queues geeignet?



Zusammenfassung

- Datenstrukturen unterscheiden sich nicht nur durch verschiedene Semantiken und Operationen, sondern auch durch weitere, spezifische Eigenschaften:
 - Statische oder dynamische Grösse, explizite oder implizite Beziehungen, direkter oder sequenzieller Zugriff.
- Aufwände für Operationen sind (auch) abhängig davon ob eine Datenstruktur sortiert ist oder nicht.
- Auf sortierten Arrays (bzw. Datenstruktur mit direktem Zugriff) kann mittels binärer Suche die Suche massiv beschleunigt werden.
- Listen können einfach oder doppelt verkettet implementiert sein.
- Trickreiche Implementationen (z.B. Ringbuffer) können Datenstrukturen (im Beispiel: Array) deutlich beschleunigen.

Fragen?