

Algorithmen und Datenstrukturen

Datenstrukturen: Arrays, Listen, Queue und Stack

Roland Gisler



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 selber implementieren.
- Sie können abhängig von Anforderungen die geeigneten Implementationen von Datenstrukturen auswählen.

Eigenschaften von Datenstrukturen

Eigenschaft: Reihenfolge und Sortierung

- 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 von Büchern.
- Datenstrukturen welche die Elemente (typisch beim Einfügen) implizit sortieren / ordnen.
 - Analogie: Vollautomatisches Hochregallager
- Achtung: Auch abhängig von der Implementierung bzw. Nutzung!

Eigenschaft: Operationen auf Datenstrukturen

- 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:
 - Nachfolger: nachfolgendes Datenelement.
 - Vorgänger: vorangehendes Datenelement.
 - Sortierung: Sortieren der Datenelemente nach best. Kriterien.
 - Maxima/Minima: kleinstes / grösstes Datenelement.

Eigenschaft: Statische vs. dynamische Datenstruktur

- 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.

Eigenschaft: Explizite vs. implizite Beziehungen

- Bei **expliziten** Datenstrukturen werden die Beziehungen zwischen Elementen von jedem Element selber **explizit** mit Referenzen festgehalten.
 - Analogie: Fahrradkette
 - Kettenglieder sind miteinander verknüpft.
 - Kettenglieder kennen ihre jeweiligen Nachbarglieder.
- Bei **impliziten** Datenstrukturen werden die Beziehungen zwischen den Datenelementen **nicht** von den Elementen selber festgehalten.
 - Die Beziehungen werden quasi von «aussen» definiert, z.B. über eine externe Nummerierung.
 - Analogie: Buchregal mit Büchern
 - Bücher stehen einfach (ggf. auch geordnet) nebeneinander.
 - Das einzelne Buch weiss nicht wo es steht bzw. hingehört.

Eigenschaft: Direkter vs. indirekter/sequenzieller Zugriff.

- 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 alle Teller sequenziell umstapeln bis der gewünschte Teller gefunden ist.

Eigenschaft: Aufwand von Operationen

- Der **Aufwand** (Rechen- und Speicherkomplexität) variiert sowohl für die verschiedenen Operationen als auch oft in Abhängigkeit der enthaltenen Elementmenge 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
 - 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 (Polynomial)

Array

(Sprachelement, 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];`

- **Implizite** Datenstruktur

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

- **Direkter** Zugriff

- Auf jedes Element kann über den Index direkt zugegriffen werden.

- **Reihenfolge:** Der Array behält die Positionen der Datenelemente so wie sie zugewiesen / eingeordnet werden bei.


Array – Suchen eines Elementes 'G'

- Unsortiert, aber fortlaufend 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>



- Sortiert:

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

- Voraussetzung:
Sortierte Datenmenge!
- Algorithmus:
 - Datenmenge in der Mitte teilen.
 - Auf Basis des Trennelementes entscheiden, ob man in der linken oder rechten Hälfte weitersucht.
 - Algorithmus **rekursiv** mit der ausgewählten Hälfte wiederholen.
 - 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'

- Unsortiert, aber fortlaufend befüllt:

Wenn wir uns den **Index** des jeweils nächsten freien Platzes merken 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>

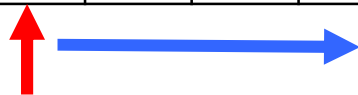
Ohne diesen Trick: $O(n)$

- 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>

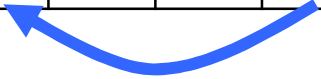
Array – Entfernen eines Elementes 'G'

- Unsortiert, 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>

- 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>

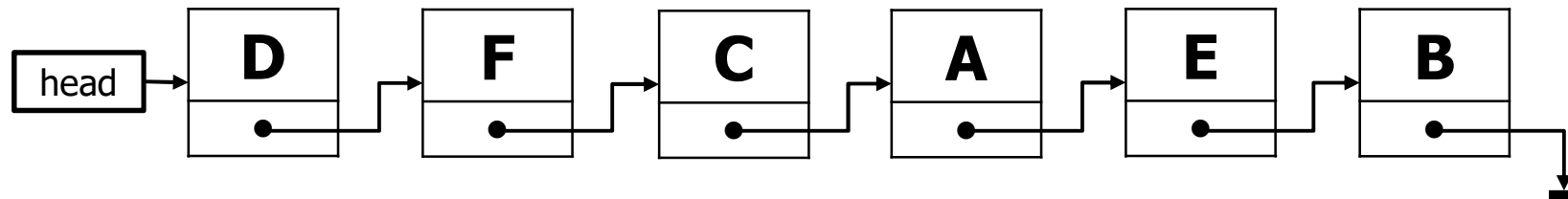
Verwendung von Arrays - Empfehlung

- 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» einzelne, elementare Datentypen ablegen muss.
 - Datentypen haben bekannte Grösse, und können somit in einer Reihung direkt als/im Array abgelegt werden. Mit dem Index kann direkt die Speicheradresse berechnet werden.
- In allen anderen Fällen sind Collections meist vorzuziehen, da sie wesentlich objektorientierter sind, und es z.B. mit der **ArrayList** ebenfalls Implementationen gibt, die einen direkten, schnellen Zugriff per Index erlauben.

Listen

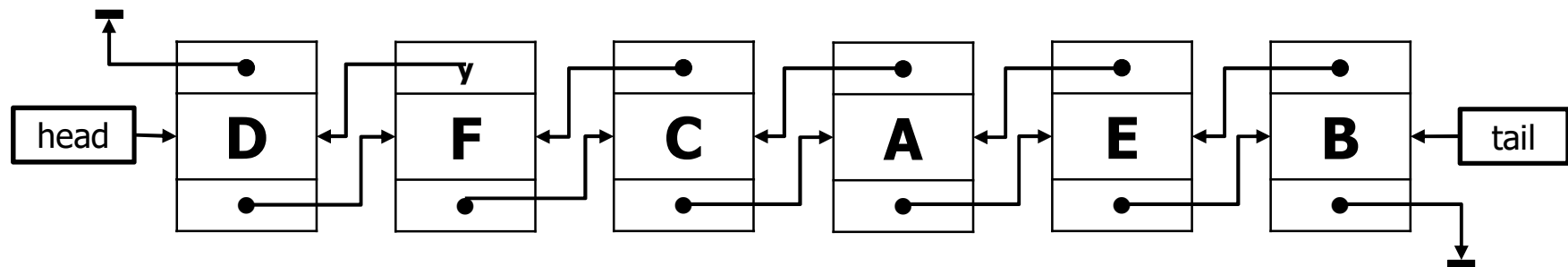
Einfach und doppelt verkettete Listen

▪ **Einfach** verkettete Liste:



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

▪ **Doppelt** verkettete 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 - Eigenschaften

- **Dynamische** Datenstruktur

- Die Grösse der Datenstruktur passt sich der Anzahl zu speichernden Element 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)

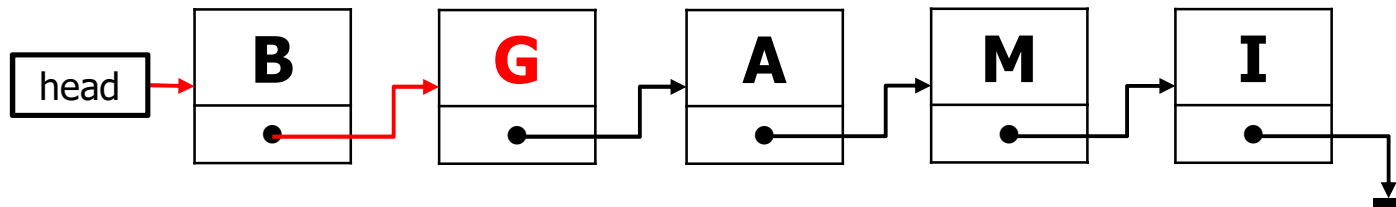
- **Nur indirekter** Zugriff

- 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.

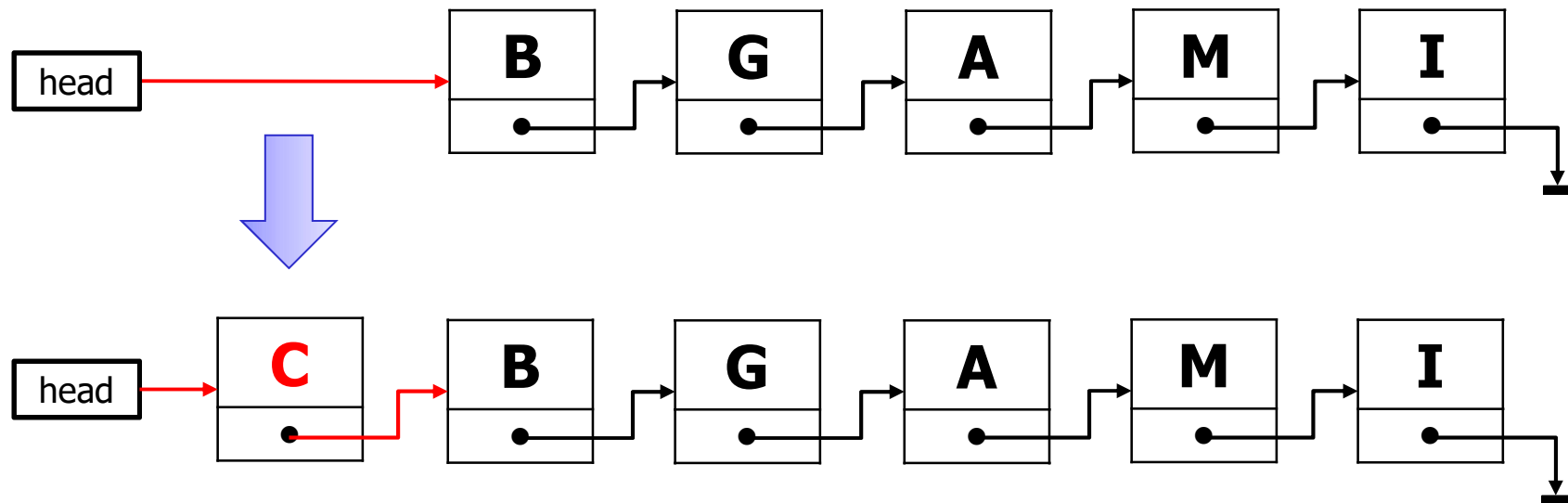


- Die Suche lässt sich aber bei sortierten Listen mit zusätzlichen Hilfsmitteln (→ Skiplisten) beschleunigen.
 - Damit wird ebenfalls $O(\log n)$ möglich.

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: **$O(1)$**

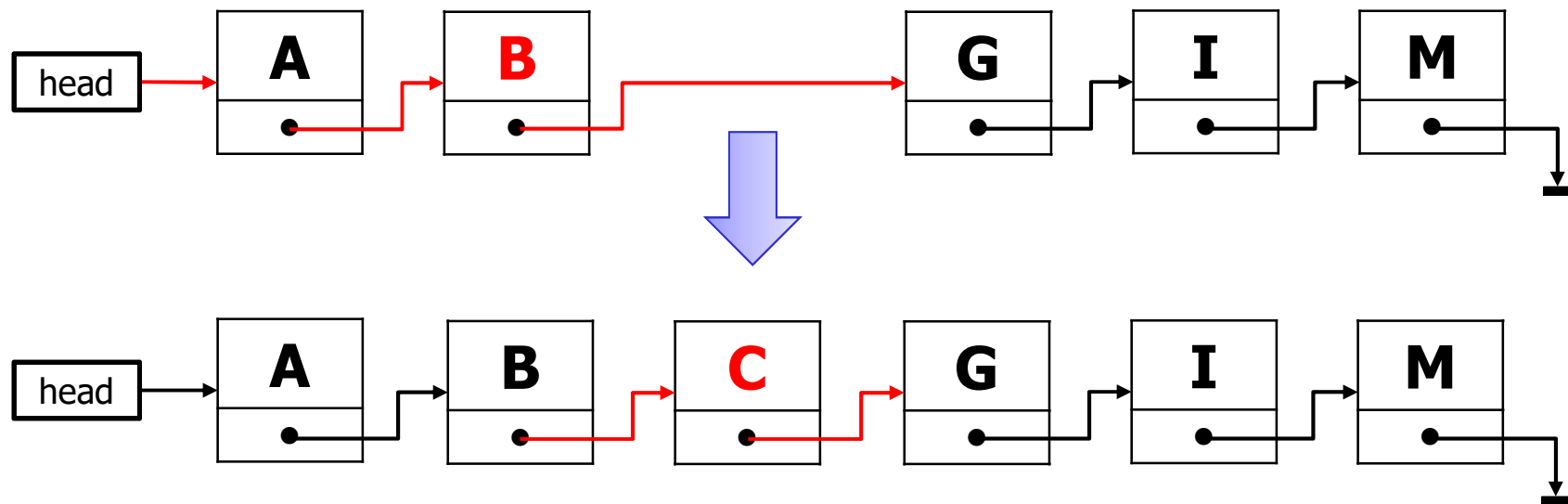


- Bei **doppelt** verketteter Liste:

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

Sortierte Listen – Einfügen eines Elementes 'C'

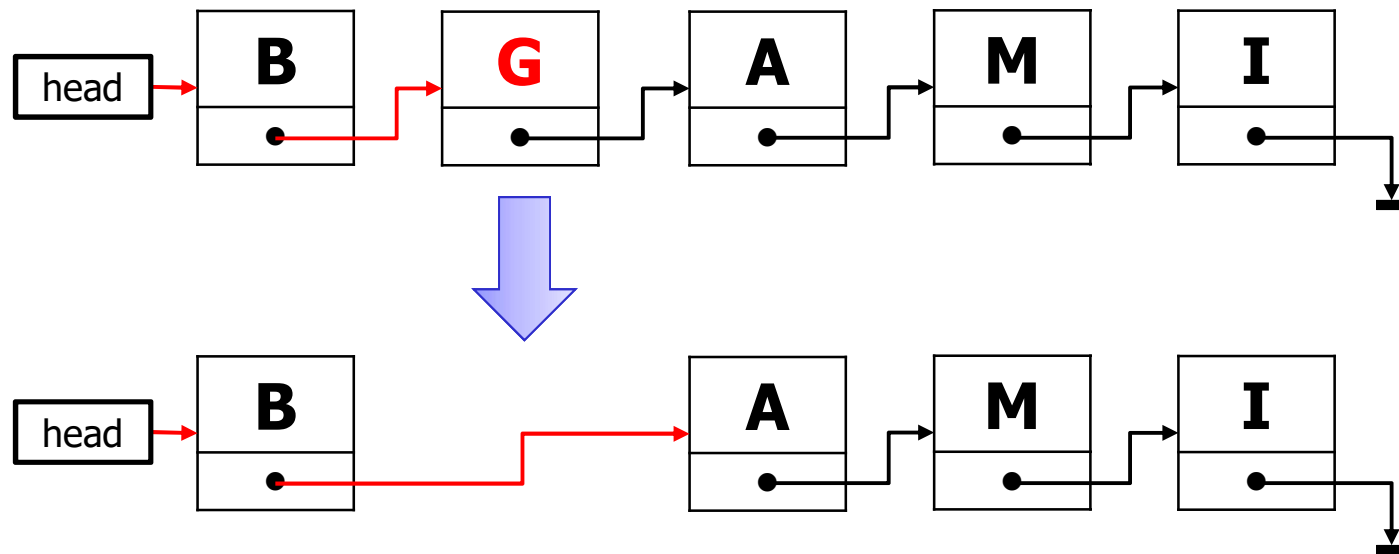
- Wir müssen sequenziell die richtige **Position** (**B**) suchen und ein neues Element einfügen, das Verschieben der restlichen Elemente **entfällt** hingegen! Der Aufwand ist trotzdem: **$O(n)$**



- 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.

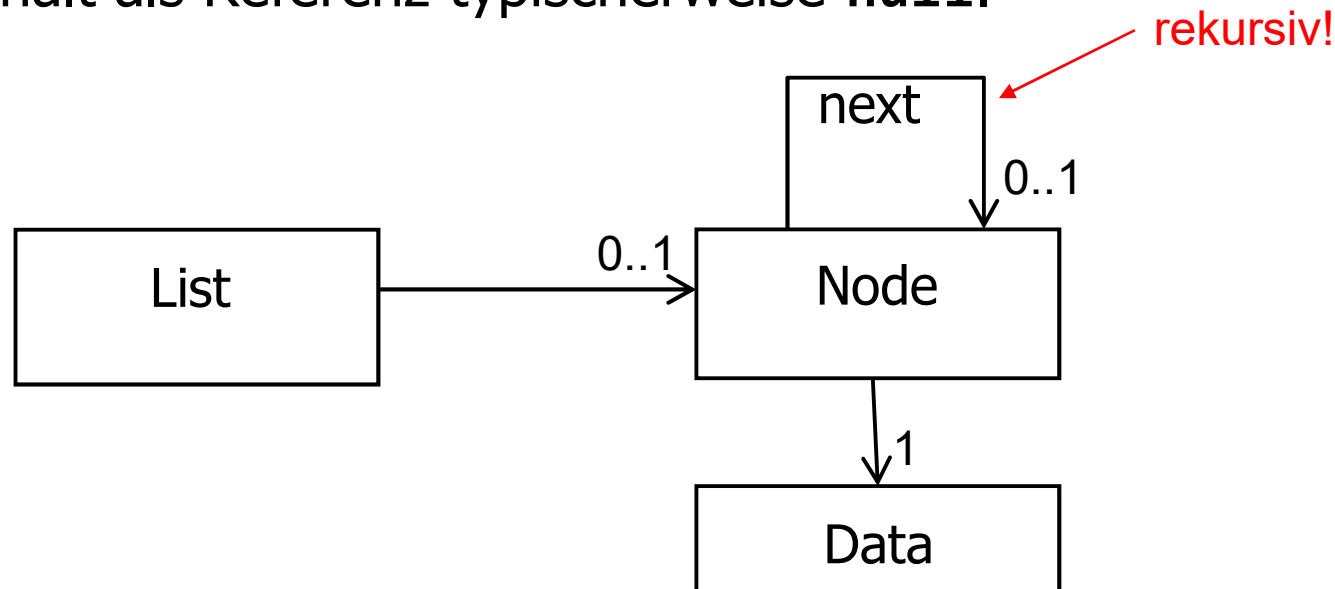
Modellierung von Listen

Modellierung von Listen

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

Konzeptionelles Modell: Einfach verkettete Liste

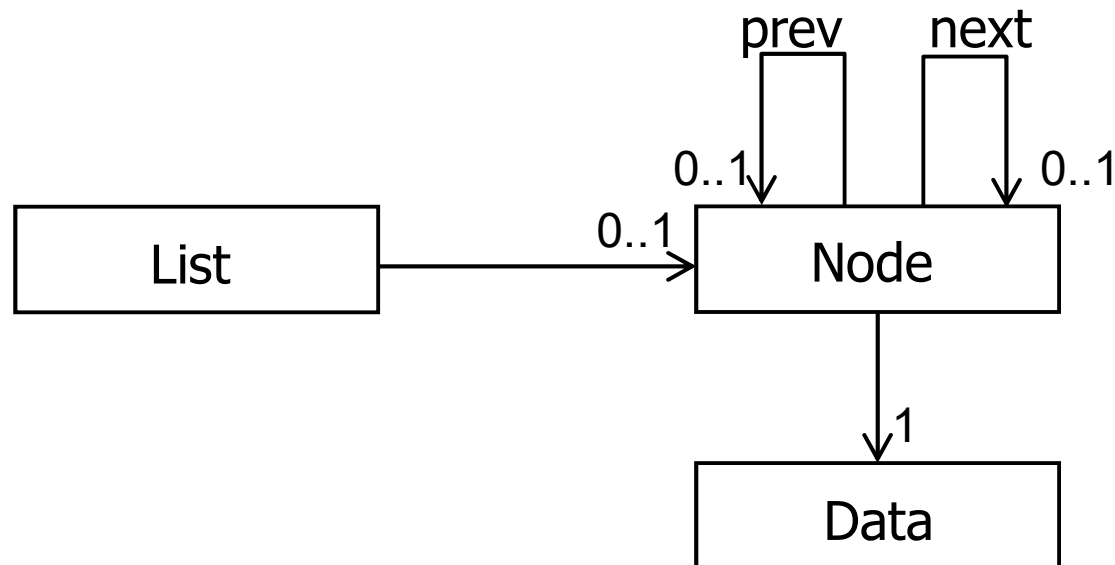
- Da jeder Node (Element) eine Referenz auf den Nachfolger (next) hält, resultiert eine rekursive Beziehung auf den selben Typ.
 - Das Modell enthält einen Zyklus.
- Hinweis: Das letzte Element (das somit keinen Nachfolger mehr hat) erhält als Referenz typischerweise `null`.



- Die Daten werden (als Objekte) meist als Referenz gespeichert.

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).



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 (sehr) grosse, stark variierende Datenmengen.
- Der Aufwand für das **reine** Einfügen in eine Liste ist an jeder beliebigen Position (sofern man diese bereits gefunden hat!) konstant (und schnell).
- Listen sind als Datenstrukturen objektorientiert implementiert, unterstützen Generics, und können dank der vorhandenen Interfaces je nach Anwendungsfall in spezifischen Implementationen eingesetzt / ausgetauscht werden.

Stack

Stack

- Der Stack ist eine Datenstruktur, der die Element 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 – **L**ast **I**n **F**irst **O**ut
- 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: Einfach verkettete Liste reicht aus.
- **push()**: Einfügen am Anfang der Liste, Aufwand **$O(1)$** .
- **pop()**: Entnehmen am Anfang der Liste, Aufwand **$O(1)$** .

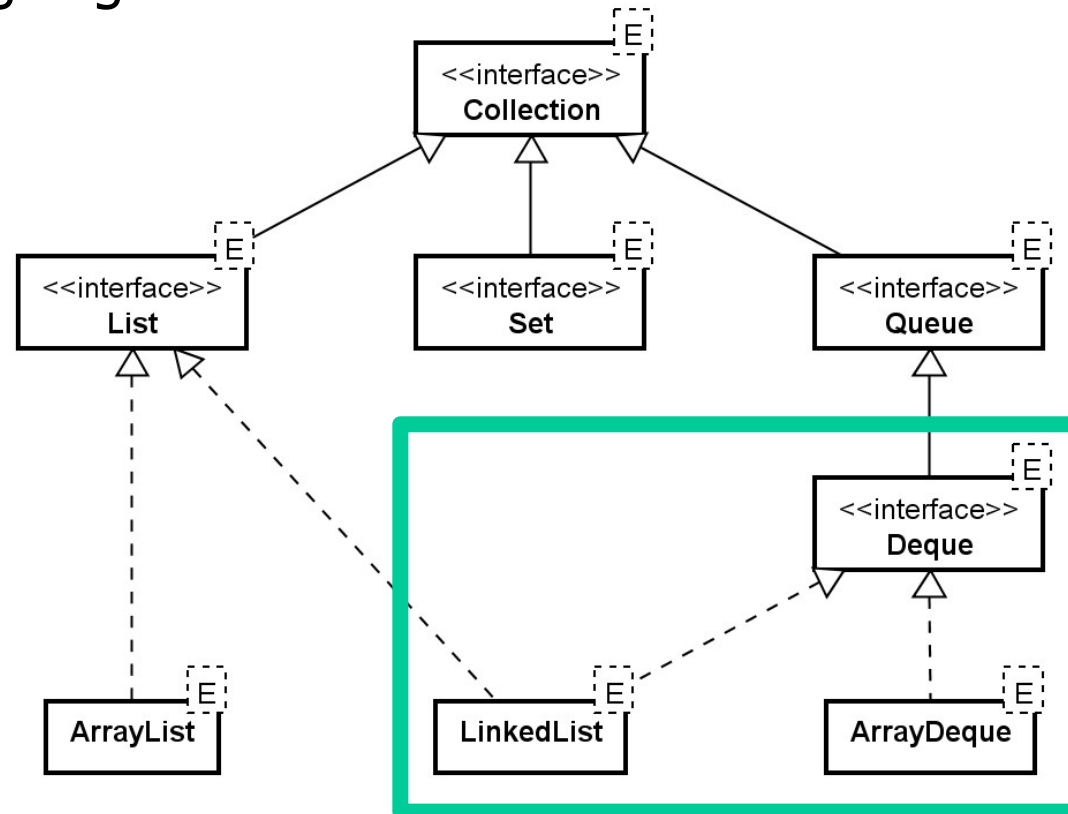
➔ Bei beiden Implementationen ist der Aufwand konstant und somit also unabhängig von der Datenmenge!
Welche Variante ist besser?

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 immer belegt und reserviert.
 - Darum aber auch sehr schnelles Einfügen möglich!
- Implementation mit Liste:
 - Einfach verkettete Liste reicht aus.
 - Leerer Stack benötigt (fast) keinen Platz.
 - Grösse dynamisch und nur durch Speicher begrenzt.
 - Speicheranforderung für neue Element notwendig, darum im Vergleich zum Array etwas langsamer!
- Dass in vielen Programmiersprachen eine **StackOverflow**-Exception/Fehler (o.ä.) existiert, bedeutet somit was?

Java - Stack mit Bibliotheksklassen

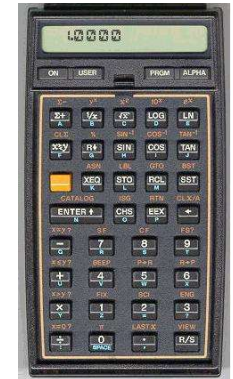
- Welche Klassen und Interfaces sind für Stack-Implementation bzw. Semantik geeignet?



- Wir haben also die Wahl!
(Nebenbei: 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 den Stack ab (push).



- Beispiel:
 - Berechnung des Ausdruckes:
 $(3 + 4) * 2$
 - Eingabe auf Taschenrechner:
 $3_{\langle \text{Enter} \rangle} 4_{\langle \text{Enter} \rangle} + 2_{\langle \text{Enter} \rangle} *$

Verlauf des Stacks:

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)` / `offer(E e)`: Element am Ende anhängen.
 - `E dequeue()` / `E poll()`: Element am Anfang entnehmen.
- Semantik: FIFO - **F**irst **I**n **F**irst **O**ut
- Analogie: Warteschlange an Kasse.
- Einsatz (Beispiele):
 - Zwischenspeicherung von Daten(-strömen).
 - Tastaturpuffer, Unix-Pipe.

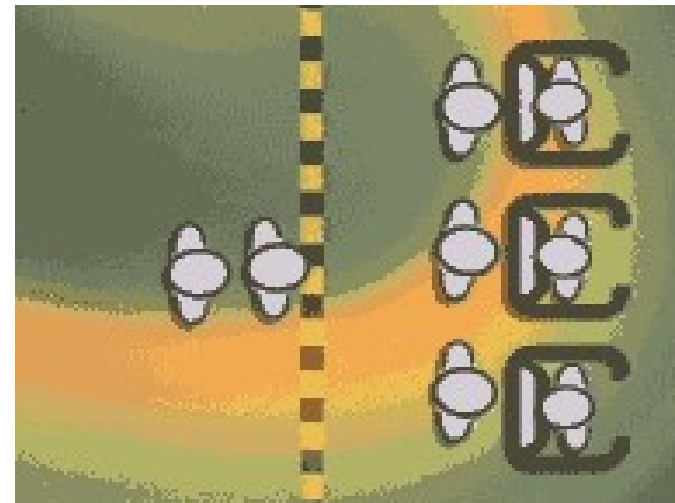


Bild: www.ku-eichstaett.de

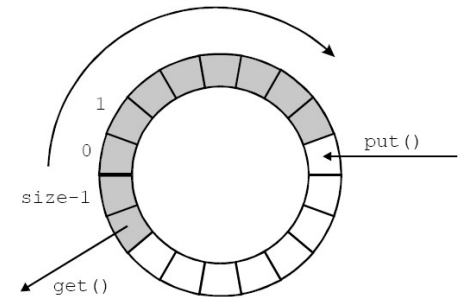
Queue – Aufwand der Operationen

- 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 **$O(1)$** .

- Implementation mit 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()**: Anhängen am Ende, Aufwand **$O(1)$** .
- **dequeue()**: Entnehmen am Ende, Aufwand **$O(1)$** .
- Hinweis: Indexe dürfen sich nicht gegenseitig überholen!

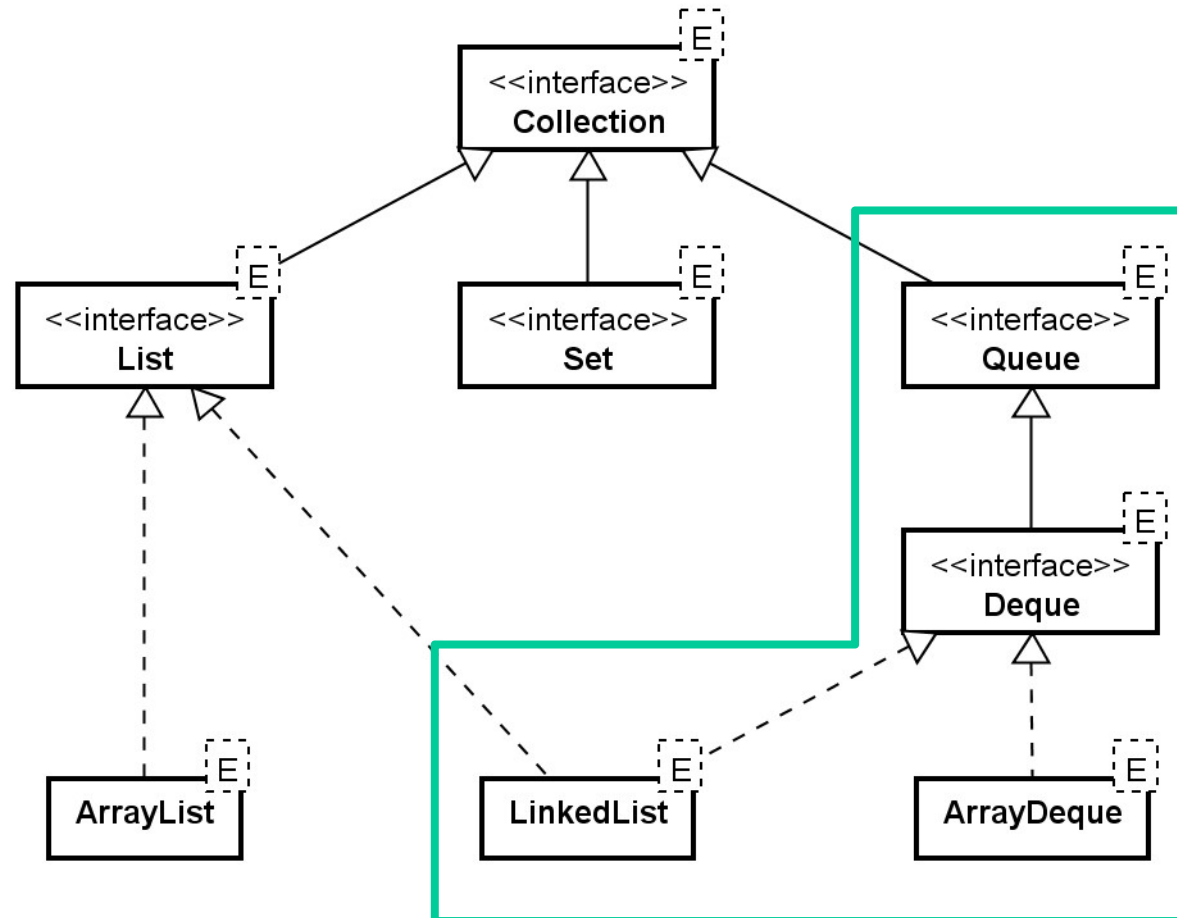


Queue – Implementation mit Liste oder mit Array?

- Implementation mit Array:
 - Trickreiche Implementation aus logischer Ringbuffer!
 - Array ist statisch, Grösse somit beschränkt.
 - Maximaler Platz immer belegt und reserviert.
 - Darum aber auch wieder sehr schnell!
- Implementation mit Liste:
 - Doppelt verkettete Liste notwendig.
 - Leere Queue benötigt (fast) keinen Platz.
 - Grösse dynamisch und nur durch Speicher begrenzt.
 - Speicheranforderung für neue Element notwendig, darum im Vergleich zum Array schon 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 / Operationen, sondern auch durch 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 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?