



Klaus Schwarz (schwarzk@HTW-Berlin.de)

(Material von Prof. Dr. Frank Burghardt, Prof. Dr. Erik Rodner,
Prof. Dr. Mohammad Abuosba)

Grundvorlesung Informatik

Algorithmen, Komplexität, Datenstrukturen

Lernziele

	Inhalt
Verstehen	Komplexität von Algorithmen, O-Notation
	Suchalgorithmen
	Sortieralgorithmen
	Datenstrukturen
Anwenden	Algorithmen entwerfen
	Laufzeitverhalten bewerten

Theorie: Komplexität von Algorithmen ($O(n)$)

Zeitkomplexität von Algorithmen

- Die Laufzeit von Algorithmen hängt oft von der benötigten Rechenzeit ab
- Man möchte die benötigte Rechenzeit (Zeitkomplexität, Komplexität) von Algorithmen abschätzen
 - Üblich: Durchschnitt (best case) oder Extremfälle (best, worst case)
- Die Komplexität ist typischerweise abhängig von der Größe des Problems (z.B. Anzahl zu sortierender Zahlen)
- Die Komplexität wird in Rechenschritten (Schritten) in Abhängigkeit der Problemgröße angegeben
 - Man nimmt vereinfachend an, dass jeder Schritt gleich viel Zeit benötigt
 - Genauer: Es werden Komplexitätsklassen benutzt, nicht die genaue Anzahl von Rechenschritten. Also die Größenordnung.
 - Z.B. werden nicht $3+5n^2$ Schritte ermittelt, sondern nur die Komplexität von ca. n^2 Schritten
 - Dazu werden alle im Wesentlichen gleich schnell wachsenden Funktionen zusammengefasst.
- Komplexitätsklassen werden mit dem (Landau-Symbol O , "Groß O") gekennzeichnet.
- Komplexitätsklassen werden in O-Notation angegeben.
- Mit $O(f(n))$ werden Funktionenklassen bezeichnet, bei denen das Wachstum nicht schneller als beim aufgeführten Repräsentanten f erfolgt, beispielsweise $O(\log n)$ oder $O(n^2)$.

Komplexitäten und Beispiele

- $O(1)$: Konstante Komplexität
 - Zugriff auf Element eines Arrays* der Größe n per Index. Egal wie groß n , Zugriff `array[index]` immer gleich schnell
- $O(\log n)$: Logarithmische Komplexität
 - auch große Probleme können in wenig Zeit, d.h. mit wenigen Berechnungsschritten, ausgeführt werden. Z.B. Suche in sortierter Liste (Wörterbuch)
- $O(n)$: Lineare Komplexität
 - Z.B. Summierung der Werte eines Arrays
- $O(n \log n)$
 - Effiziente Sortieralgorithmen (Quicksort)
- $O(n^2)$ quadratische Komplexität. Wenn das Problem 8 Mal größer wird, dann wird der Algorithmus 64 mal mehr Rechenzeit benötigt!
 - Bubble Sort (Paarweise Element einer Liste sortieren, immer wieder bis nichts mehr zu sortieren ist)



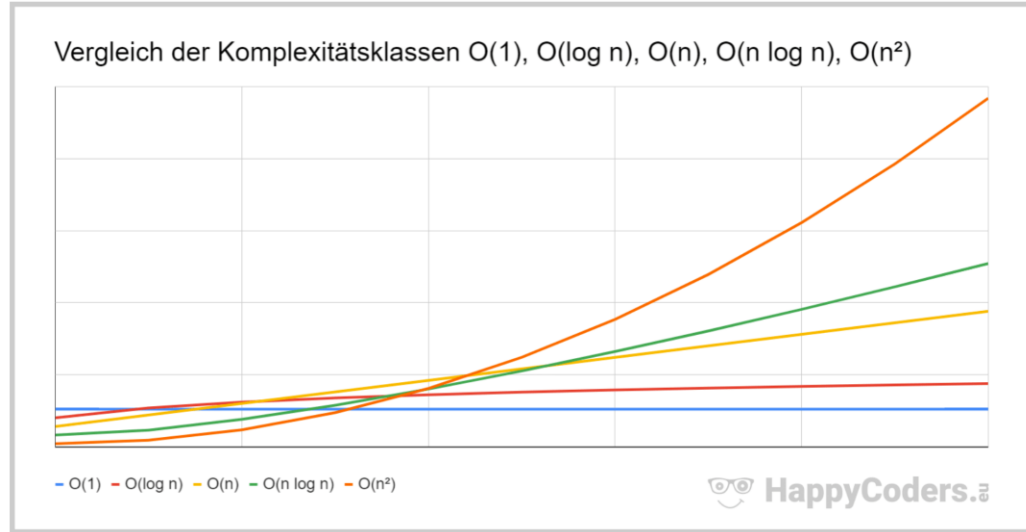
* Später: Datenstrukturen. Kurz: Array = fixe Länge, fixe Elementtypen, indizierter Zugriff

Komplexitäten und Beispiele

- Es gibt aber noch schneller wachsende und daher in der Praxis „unangenehmere“ Komplexitätsklassen; schnellere Prozessoren helfen kaum!
 - $O(n^m)$ – polynomieller Aufwand
 - $O(2^n)$ – exponentieller Aufwand
 - $O(n!)$ – faktorieller Aufwand

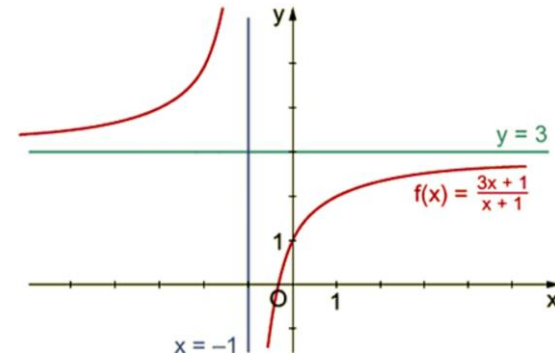


Vermeiden Sie Algorithmen hoher Komplexität!



Komplexität $O(f(x))$, mathematischer Hintergrund

- Die Komplexität wird in der Informatik bei der Analyse von Algorithmen verwendet. Ist ein Maß für die Anzahl der für die Lösung eines Problems benötigten Schritte.
- $f = O(g)$ heißt: f wächst nicht wesentlich schneller als g
- $O(g)$ von Herrn Landau eingeführt, auch *Landau* Symbol.
- Hintergrund: Beschreibt das asymptotische Verhalten von Funktionen und Folgen.
 - (Eine Asymptote ist in der Mathematik eine Linie, der sich der Graph einer Funktion im Unendlichen immer weiter annähert.)
- Bsp.: Binäre Suche hat $O(\log_2(n))$



Laufzeitbewertung von Algorithmen

Laufzeit eines Algorithmus

- Wie lange benötigt eine Algorithmus zur Bestimmung der Lösung?
- Möglichkeit 1: Zeit stoppen
 - Unter Linux/Unix mit dem Befehl: `time` (real gibt die reale Laufzeit an)
 - In python: Verwendung des Modules `timeit`
 - Abhängig von Hardware und anderen aktiven Programmen, etc.
 - Direkte Laufzeit ist schwer zu vergleichen



Laufzeit eines Algorithmus

- Möglichkeit 2: Zählen von Operationen
- Operationen:
 - ① Einzelne Rechenoperation (+, -, *, /)
 - ② Vergleichsoperationen (==, <, >)
 - ③ Zugriff auf Feldelemente
 - ④ ...

(Zählen der Schritte,
engl. steps)

Beispiel, Problem für das Laufzeit gemessen wird: Suche von Elementen

- Typische Aufgabenstellung in der Informatik: Suchen von Elementen in einer Liste
 - Gibt es Daten eines bestimmten Kunden in einer Datenbank?
 - Suche nach einer Datei eines gewissen Namens
 - etc.
- Gegeben ist ein Feld mit Elementen x_0, \dots, x_{n-1}
- Wie stelle ich für einen Wert z fest ob dieser enthalten ist?



Analyse der Laufzeit

(n steps)

- Ungünstigster Fall: Durchlauf des ganzen Feldes, daher n Feldzugriffe
(worst case analysis)
- Durchschnittliche Laufzeit: $\frac{n}{2}$ Feldzugriffe
(average case analysis)
- worst case analysis ist oft einfacher durchzuführen




Gibt es ein besseres Suchverfahren?

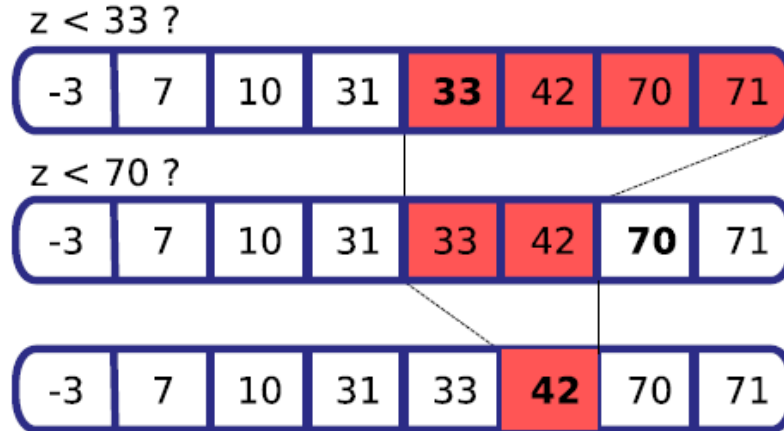
- Vollständige Suche funktioniert, aber ist sie auch effizient genug?
- ... auch bei Millionen von Datenelementen?
- Gibt es einen Algorithmus der weniger als n Feldzugriffe/Vergleiche benötigt?
- Antwort: **ja**,^{*} **ABER** nur wenn das Feld vorher sortiert ist

* In der Praxis: Noch mehr *ABERs*

Ein Spiel: <https://www.bildungs-lsa.de/files/9f4964f1900cd4ba2765a539466550fa/schueler-zahlenraten02.htm>

Ein besseres Suchverfahren: Binäre Suche

- Sortiertes Feld: $x_0 \leq x_k \leq x_n$
- Prinzip: Wie beim Erraten einer Zahl zwischen 0 und 100
- Binäre Suche hat $O(\log_2(n))$
- Bsp.: Suche nach 42: 
- Mittleres Element:
 - $>$, $<$, $=$?
 - Wenn $=$ dann fertig
 - Sonst wenn $</>$ in jeweiliger Hälfte wieder mittleres Element



Binäre Suche – Das Konzept

Wie oft müssen wir einen Vergleich durchführen?

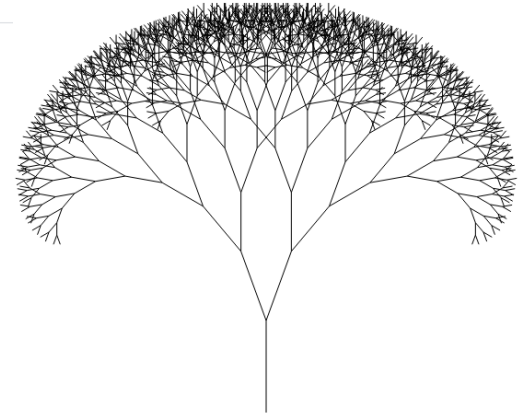
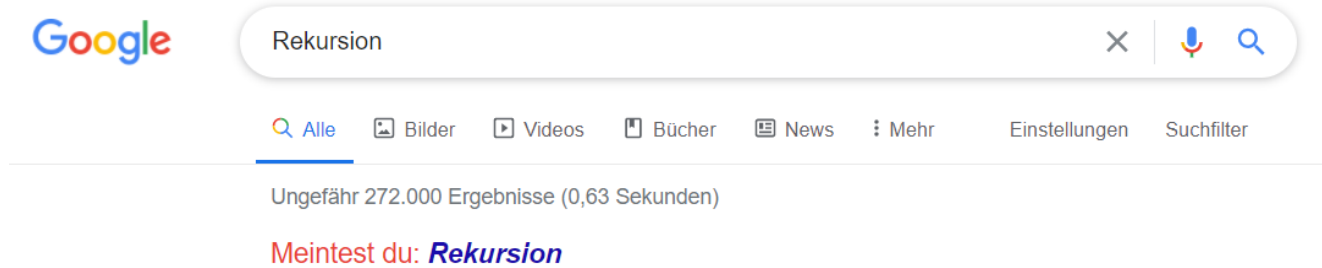


Wachstum der Laufzeiten bei verschiedenen $O()$

$f(n)$	$n = 2$	$2^4 = 16$	$2^8 = 256$	$2^{10} = 1024$	$2^{20} = 1048576$
$\log(n)$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \log(n)$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^9$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$
Anzahl der Atome im Weltall $\approx 10^{77}$ (nach A. Beutelspacher)					

Mittel in Algorithmen: Rekursion

Rekursion: Wiederholung



Rekursion

- Rekursion: Zurückführen eines Problemes "auf sich selbst"
- *"Um Rekursion zu verstehen muss man erst einmal Rekursion verstehen."*
- Beispiel: Fakultät

$$n! = \begin{cases} 1 & n \leq 1 \\ (n-1)! \cdot n & \text{sonst} \end{cases}$$

- Voraussetzung für eine Rekursion:
 - 1 Es existiert eine Lösung für "einfache" Fälle
 - 2 "Komplexe" Fälle lassen sich auf einfache Fälle zurückführen

Rekursion: Ergänzung

- Oft ist eine Funktion einfacher mit Rekursion zu programmieren als mit einer Iterationsvorschrift (Schleife)
- Aber: Jeder rekursive Algorithmus lässt sich in einen iterativen Algorithmus überführen
- Weitere Beispiele: Werte der Fibonacci Folge, Sortieralgorithmen, .
- Rekursive Akronyme: :)
 - ① VISA = Visa International Service Association
 - ② Liste: http://de.wikipedia.org/wiki/Rekursives_Akronym

Rekursive Formeln

- Rekursion kann auch zur Definition mathematischer Formeln verwendet werden
- Vorheriges Beispiel der Fakultät
- **Weitere Beispiele:**

1. $T(n) = T(n - 1) + n, \quad T(0) = 0$

2. $T(n) = T(n - 1) + T(n - 2), \quad T(0) = 1, \quad T(1) = 1$ (Fibonacci Sequenz)

Wie lässt sich die Funktion aus 2. als nicht-rekursive Formel ausdrücken?



Fibonacci iterativ

```
class Fibo {  
    public static void main(String[] args) {  
        int fib = 20;  
        int hilf;  
        int h1, h2;  
        h1 = 0;  
        h2 = 1;  
        System.out.println("0");  
        System.out.println("1");  
        for (int i = 1; i < fib; i++) {  
            hilf = h1 + h2;  
            h1 = h2;  
            h2 = hilf;  
            System.out.println(hilf);  
        }  
    }  
}
```

Wesentlich komplizierter, oder?

Sortiervverfahren

Sortierverfahren

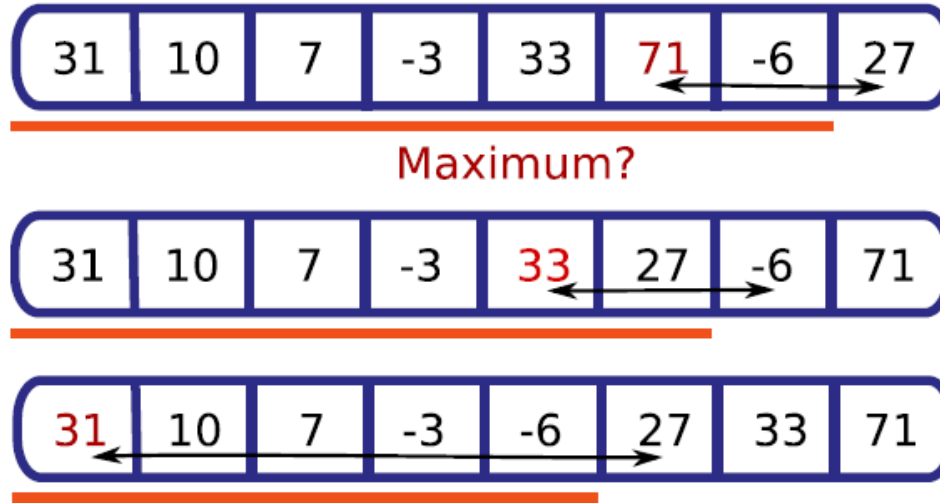
- Eingabe: unsortiertes Feld von n Elementen: x_1, \dots, x_n
- Ausgabe: sortiertes Feld
- Es existieren zahlreiche Sortieralgorithmen:
 - 1 Bubble Sort
 - 2 Insertion Sort
 - 3 Selection Sort
 - 4 Quick Sort
 - 5 Merge Sort
 - 6 Distribution Sort
 - 7 ...

Das schlechteste Sortierverfahren der Welt!

- "Forever" Sort:
 - 1 Erzeuge eine neue Reihenfolge der Elemente
 - 2 Überprüfe ob die Elemente richtig sortiert sind
 - 3 Wenn dies nicht der Fall ist, gehe zu Schritt 1, ansonsten ist eine Lösung gefunden
- Aufwand des Algorithmus? Ungünstigster Fall?
- Ungünstigster Fall: Alle möglichen Reihenfolgen müssen überprüft werden
- Anzahl der Reihenfolgen: $n!$ (n Fakultät)
- Asymptotische Laufzeit: $n! \in O(n^n)$

Ca: Würfeln einer Straße bei Kniffel

Selection-Sort: „Maxima nach hinten!“



https://www.youtube.com/watch?v=f8hXR_Hvybo

Selection-Sort: Algorithmus

- Bestimmung des Maximums kleiner werdender Teilfelder
- Verschieben des Maximums auf die hintere Position nach dem Teilfeld
- Erster Schritt: Maximum gelangt an die hintere Position
- Zweiter Schritt: Zweitgrößter Wert gelangt an die vorletzte Position
... u.s.w.

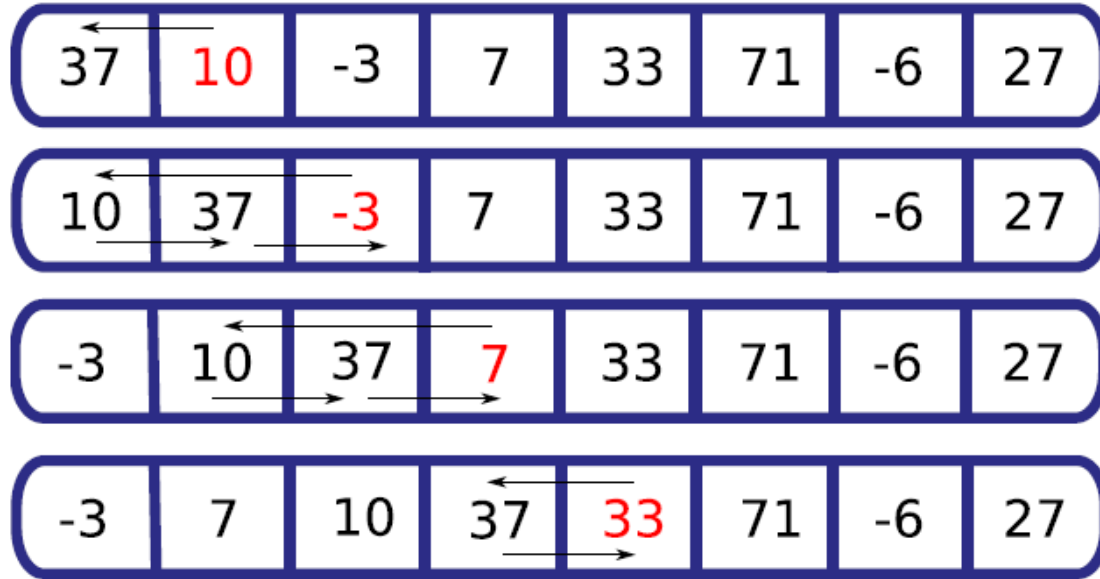
Selection-Sort: Analyse der Laufzeit

- In Schritt k wird das Maximum von $n - k + 1$ Elementen bestimmt
- Anzahl der Gesamtschritte: n

$$\begin{aligned} f(n) &= \sum_{k=1}^n (n - k + 1) \\ &= n^2 - \frac{n(n+1)}{2} + n = \frac{n^2 + n}{2} \in O(n^2) \end{aligned}$$

- Worst case und best case: $O(n^2)$

Insertion-Sort: „Herstellen der Ordnung“



<https://www.youtube.com/watch?v=DFG-XuyPYUQ>

Insertion Sort: Algorithmus

Wiederhole folgenden Vorgang vom zweiten bis zum letzten Element:

- ❶ Suchen einer Position vor dem Element mit kleinerem Vorgänger (oder Anfang des Feldes falls unmöglich)
- ❷ Verschieben der Elemente zwischen neuer und alter Position um ein Feld nach hinten
- ❸ Setze das aktuelle Element in die neue Position
- ❹ Springe ein Feld weiter

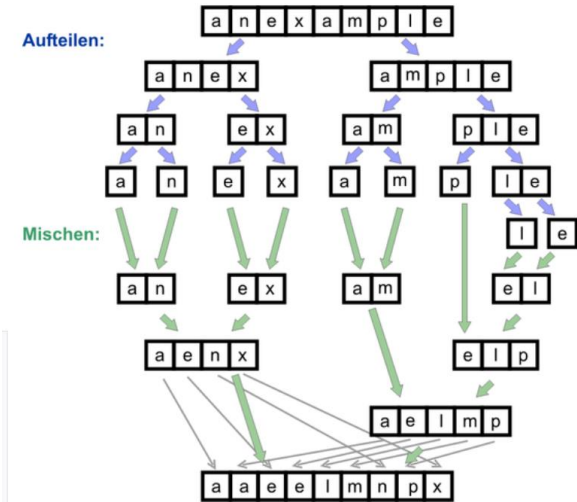
Analyse ähnlich zu Selection Sort,
aber best case $O(n)$ und worst case $O(n^2)$

Merge-Sort: Sortiere zwei Hälften!

- Typischer "Teile und Herrsche" Algorithmus
- Grundidee:
 - 1 Teile das Feld in zwei Teilfelder
 - 2 Sortiere jeweils die Teilfelder durch **rekursive** Anwendung des Algorithmus
 - 3 Mische die zwei sortierten Teilfelder zum Gesamtergebnis

Wer hat das erfunden?

John von Neumann, 1945



Merge-Sort: Analyse

- Mischen zweier sortierter Liste benötigt $O(n)$
- Anwendung der Rekursion für die Bestimmung der Anzahl der Rechenoperationen

$$f(n) = \underbrace{2 \cdot f\left(\frac{n}{2}\right)}_{\text{Sortieren der Teilfelder}} + \underbrace{c \cdot n}_{\text{Mischen der sortierten Teilfelder}}$$
$$= 2 \left(2 \cdot f\left(\frac{n}{4}\right) + c \cdot \frac{n}{2} \right) + c \cdot n$$

$$\begin{aligned} f(n) &= 4f\left(\frac{n}{4}\right) + 2c \cdot n \\ &= 8f\left(\frac{n}{8}\right) + 3c \cdot n \\ &= 16f\left(\frac{n}{16}\right) + 4c \cdot n \\ &= n \cdot f(1) + \log_2(n) \cdot c \cdot n \in O(n \cdot \log_2(n)) \end{aligned}$$

Quick-Sort

Wer hat das erfunden?

→ Tony Hoare

$O(n \log(n))$ – besser geht es nicht

Prinzip Quick Sort:

Pivotelement wählen. Elemente kleiner Pivot in linke Liste, größer in rechte. Quicksort rekursiv auf links und rechts anwenden.



Datenstrukturen

Arrays, Listen, Stacks und Queues

Algorithmen benötigen grundlegende Datenstrukturen. Hier und heute:
Arrays, Listen, Stacks, Queues, Maps.

Arrays

Arrays gehören zu den einfachsten Datenstrukturen und sind in so gut wie jeder Programmiersprache vorhanden. Bei Erzeugung eines Arrays wird die Größe festgelegt und kann später nicht mehr geändert werden.

Ein Array besteht aus einer Anzahl von Speicherzellen die über ihren Index direkt adressierbar sind.

Es gibt ein und mehr dimensionale Arrays.

43	32	5	18	77	0	...	17	...	56
1	2	3	4	5	6		i		n

Vorteil: Direkte Adressierung, effizient

Nachteil: Statisch, Verschiebung teuer

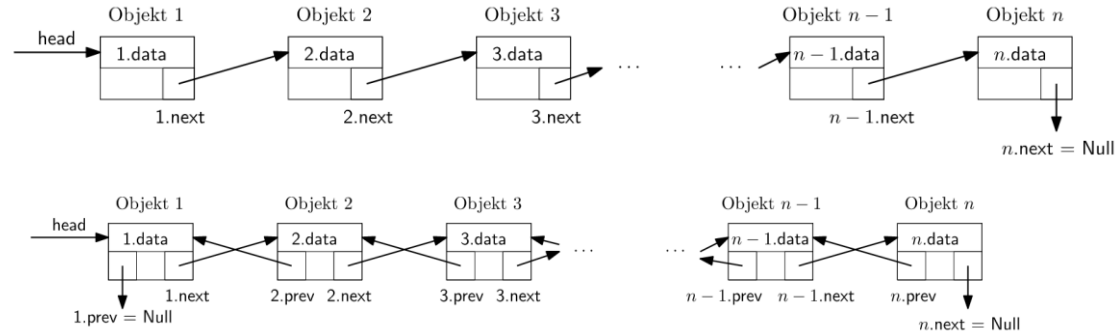
	1					j		m
1								
			...					
		...						
⋮								
i								
n								

Listen I

Im Gegensatz zu Arrays ist bei Listen die Anzahl der Elemente variable, es können neue Elemente eingefügt werden.

Eine Liste ist eine Verkettung von Objekten bestehend aus zu speicherndem Datum und einen Zeiger auf das Nachfolgeobjekt. Es gibt auch Listen mit Zeiger auf das Vorgängerobjekt, diese sind dann doppelt verkettet (vs. einfach verkettet).

Es gibt einen Zeiger auf *head*, das erste Element der Liste. Das letzte Element einer Liste hat keinen Nachfolger.



<https://hpi.de/friedrich/teaching/units/arrays-listen-stacks-und-queues.html>

Listen II

Vorteile gegenüber Arrays:

Variabler, ändern leichter

z.B. Verschieben oder einfügen, Größenänderungen

Nachteile:

Mehr Speicher Verbrauch,

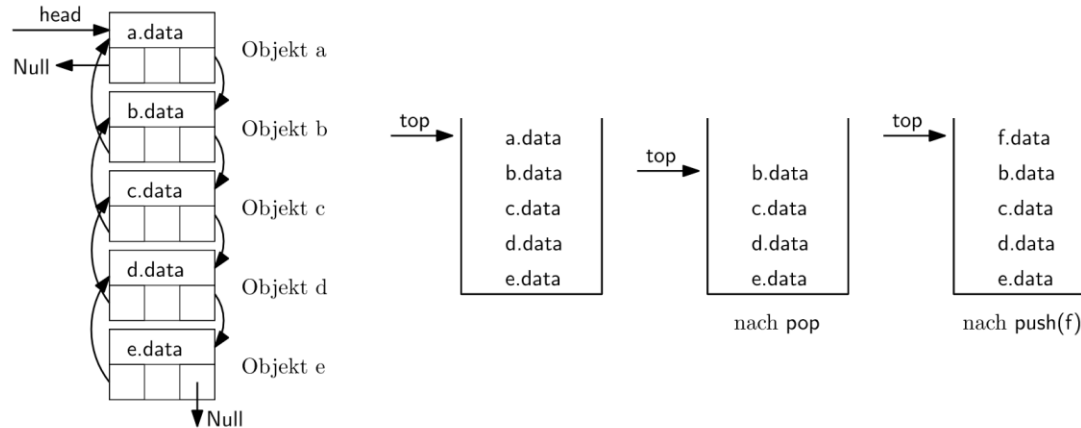
Zugriff auf Element teurer

<https://hpi.de/friedrich/teaching/units/arrays-listen-stacks-und-queues.html>

Listen: Die Basis für komplexere Datenstrukturen

Mithilfe von doppelt verketteten Listen können weitere abstrakte Datentypen realisiert werden, z.B. Stacks und Queues.

Stack: Compilerbau, Unterprogramme mit Parametern, push, pop



Queue: FIFO, Warteschlange

<https://hpi.de/friedrich/teaching/units/arrays-listen-stacks-und-queues.html>

Maps:

Eine Map besteht aus key-value Paaren (Schlüssel, Wert). Es können neue Paare hinzugefügt oder entfernt werden.

Jeder key darf in einer Map nur genau einmal vorhanden sein, wodurch jedes key-value Paar unique (einmalig) ist.

Operationen:

get(Object key)

put(K key, V value)

remove(Object key)

isEmpty()

Bsp.:

Schlüssel (key)	Wert (value)
12345	Student {Waldfee, Holla, 12345}
12355	Student {Stilzchen, Humpel, 12355}



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

www.htw-berlin.de