2 Datenstrukturen / Algorithmen

2.1 Laufzeitverhalten

2.2 Elementare Datenstrukturen
2.2.1 Listen (Beispiel ArrayList / Collection-Interface)
2.2.2 Stack
2.2.3 Queues
2.2.4 Assoziative Speicher
2.2.5 Bäume

2.3 Elementare Sortieralgorithmen
2.3.1 Bubble-Sort
2.3.2 Quick-Sort
2.3.3 Baumstrukturen
2.3.4 Bucketsort

BIBI 7aWI **Schuljahr 2014/2015**
4(4)h BAT / KAUF

2 Datenstrukturen / Algorithmen

- Eine Datenstruktur ist eine bestimmte Art, Daten im Speicher eines Computers anzuordnen und zu verwalten.
- Daten werden gekapselt abgelegt (OO Prinzip Kapselung/Information Hiding) – der Zugriff ist nur über bestimmte Methoden möglich.
- Eine Datenstruktur bietet mindestens Operationen an, um *neue Daten aufzunehmen sowie vorhandene Daten zu löschen und zu lesen.*
- Es gibt statische (=fixe Größe) und dynamische (=variable Datengröße) Datenstrukturen!

2.1 Laufzeitverhalten (1)

Laufzeitverhalten / Aufwandanalyse

- Algorithmen / Datenstrukturen werden klassifiziert, wie schwierig oder aufwändig sie zur Berechnung sind
- Wird mit Landau-Symbolen ausgedrückt - z. Bsp. O (Sprich: **O-Notation**) – Edmund Landau war ein deutscher Zahlentheorektiker
- O-Notation wurde ebenfalls vom deutschen Zahlentheoretiker Paul Bachmann verwendet (1894)

asymptotische obere Schranke,
Funktion f wächst nicht wesentlich schneller als Funktion g

http://de.wikipedia.org/wiki/Landau-Notation

2.1 Laufzeitverhalten (1)

Ziele der O-Notation

- Es soll eine Funktion f in Abhängigkeit der Anzahl der Eingabewerte n gefunden werden, welche die Laufzeit eines Algorithmus' wiedergibt: f(n).
- Die angegebene Funktion f(n) soll unabhängig sein von systemspezifischen Details (tatsächlicher Zeitaufwand), daher ist von konstanten Faktoren c zu abstrahieren.
- Es soll eine Untersuchung des "ungünstigsten Falls" stattfinden und die Funktion f(n) soll mit einer Konstanten c als Faktor eine Asymptote bzw. Majorante zur tatsächlichen Laufzeit g(n) darstellen.

- [http://www.linux-related.de/index.html?/coding/o-notation.htm]

http://de.wikipedia.org/wiki/Landau-Notation

2.1 Laufzeitverhalten (2)

Notation	Bedeutung
$f \in \mathcal{O}(1)$	*f ist beschränkt* / f überschreitet einen Wert nicht.
$f \in \mathcal{O}(\log x)$	*f logarithmisches Wachstum* / f wächst ungefähr um einen konstanten Betrag, wenn sich das Argument verdoppelt. Merke: Die Basis des Logarithmus ist egal.
$f \in \mathcal{O}(x)$	*f lineares Wachstum* / f wächst ungefähr auf das doppelte, wenn sich das Argument verdoppelt.
$f \in \mathcal{O}(x^2)$	*f quadratisches Wachstum* / f wächst ungefähr auf das vierfache, wenn sich das Argument verdoppelt
$f \in \mathcal{O}(2^x)$	*f exponentielles Wachstum* / f wächst ungefähr auf das doppelte, wenn sich das Argument um eins erhöht

http://de.wikipedia.org/wiki/Landau-Notation

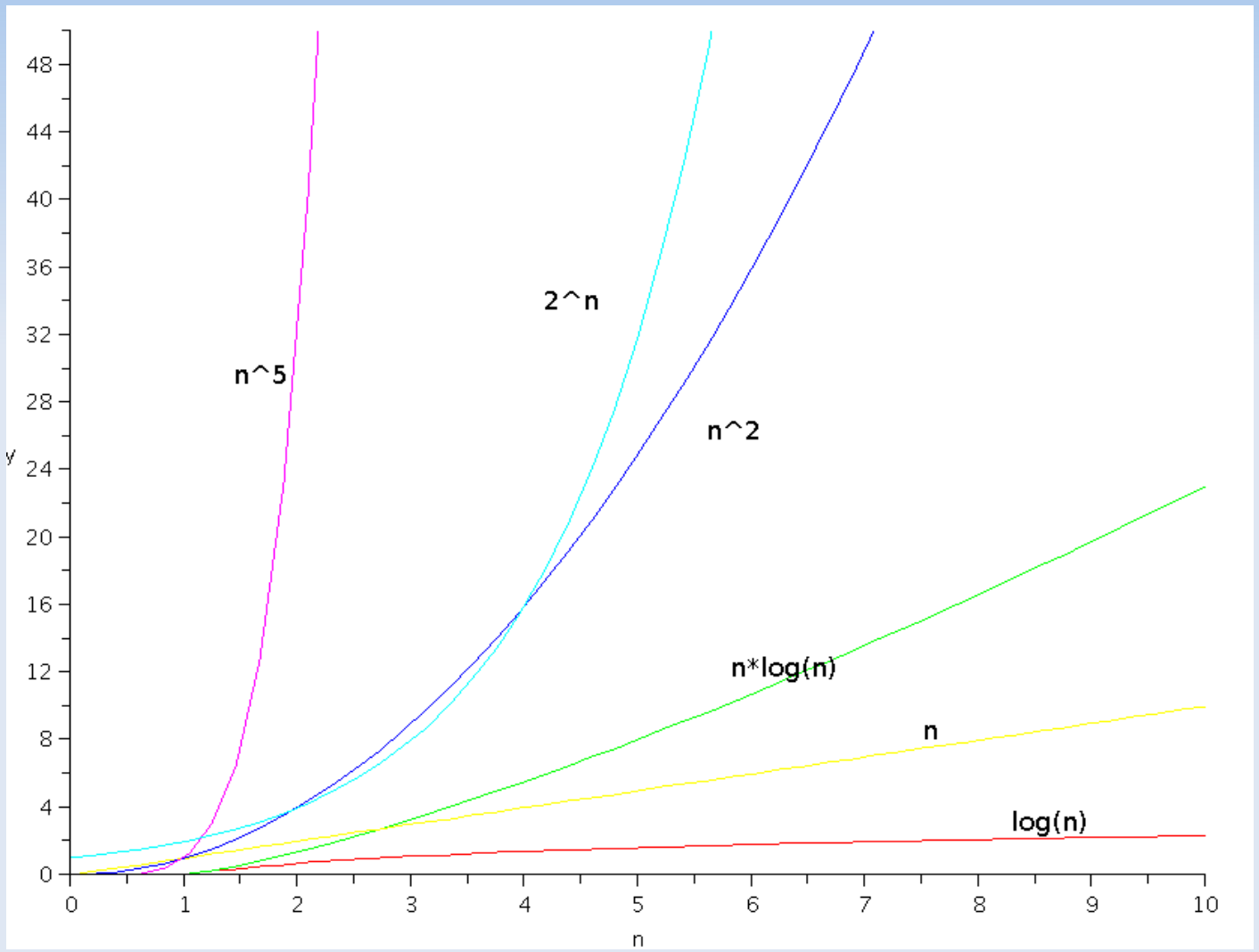
2.1 Laufzeitverhalten (3)

Beispiele:

- $f(n) = 2n + 5$ $= O(n)$
- $f(n) = 2n^2 + 5$ $= O(n^2)$
- $f(n) = n^3 + n^2$ $= O(n^3)$
- $f(n) = 10 \ln n$ $= O(\log_x n)$ (Basis x des Logarithmus ist egal für die O-Notation)

http://de.wikipedia.org/wiki/Landau-Notation

2.1 Laufzeitverhalten (4)



http://de.wikipedia.org/wiki/Landau-Notation

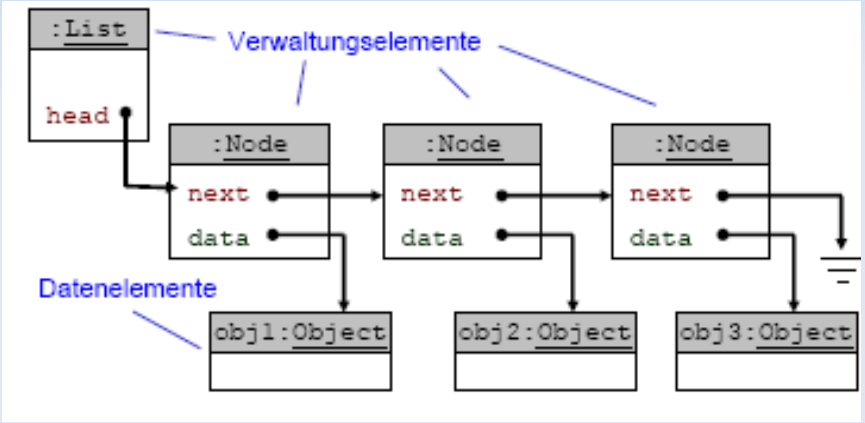
2.2 Elementare Datenstrukturen

Unterteilungen:

2.2.1 Listen (Beispiel ArrayList / Collection-Interface)
2.2.2 Stack
2.2.3 Queues
2.2.4 Assoziative Speicher
2.2.5 Bäume

2.2.1 Elementare Datenstrukturen (Listen 1)

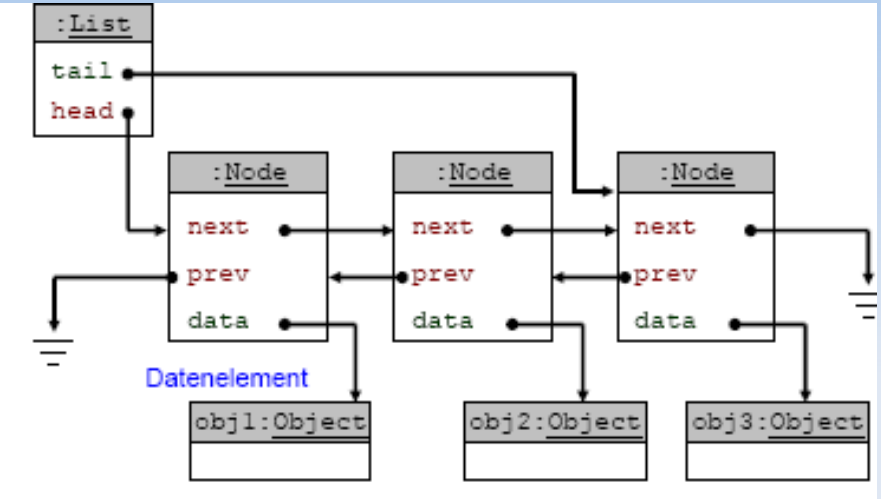
- Eine der einfachsten dynamischen Datenstrukturen
- Einfach verkettete Listen (Referenz auf den Nachfolger)
- Doppelt verkettete Listen (Referenz zum Vorgänger und Nachfolger)



Einfach verkettete Liste

Quelle: http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2005/STQM/Info1/skript/kap7-4.pdf

2.2.1 Elementare Datenstrukturen (Listen 2)



Doppelt verkettete Liste

Einfach verkettete Liste
- Einfügen in O(n)
- Nachfolger O(1)
- Vorgänger O(n)

Doppelt verkettete Liste
- Einfügen in O(n)
- Nachfolger in O(1)
- Vorgänger in O(1)

2.2.1 Java- Collections I

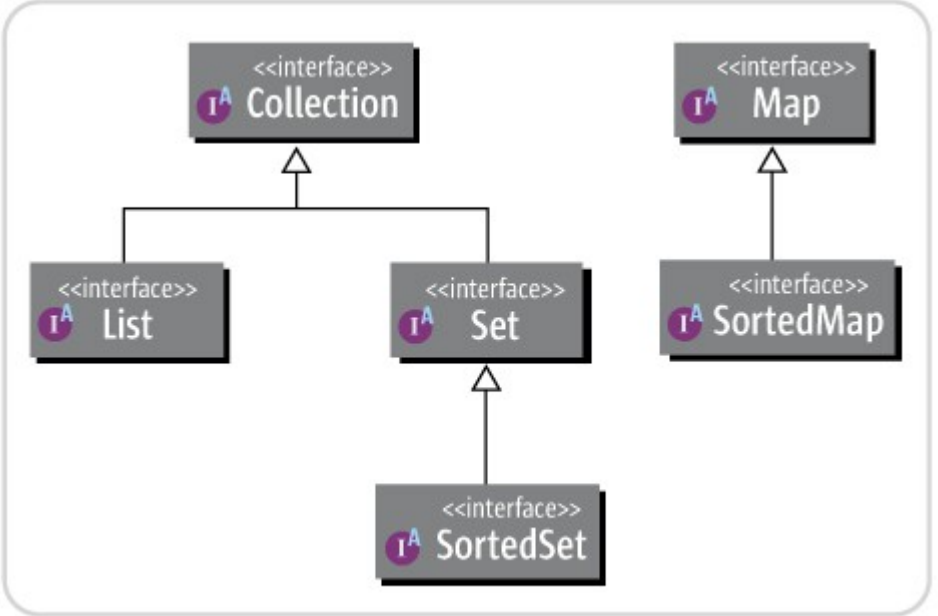


Figure 1 Collections Framework major interfaces

List: Beinhaltet Objekte / gleiches Objekt (vergleich mit der Methode equals() kann mehrfach vorkommen
Set: Duplikate sind nicht möglich (siehe Definition Menge)
Map: Key-Value Werte (wie zum Beispiel Property-Files-Einträge)

2.2.1 Java- Collections II

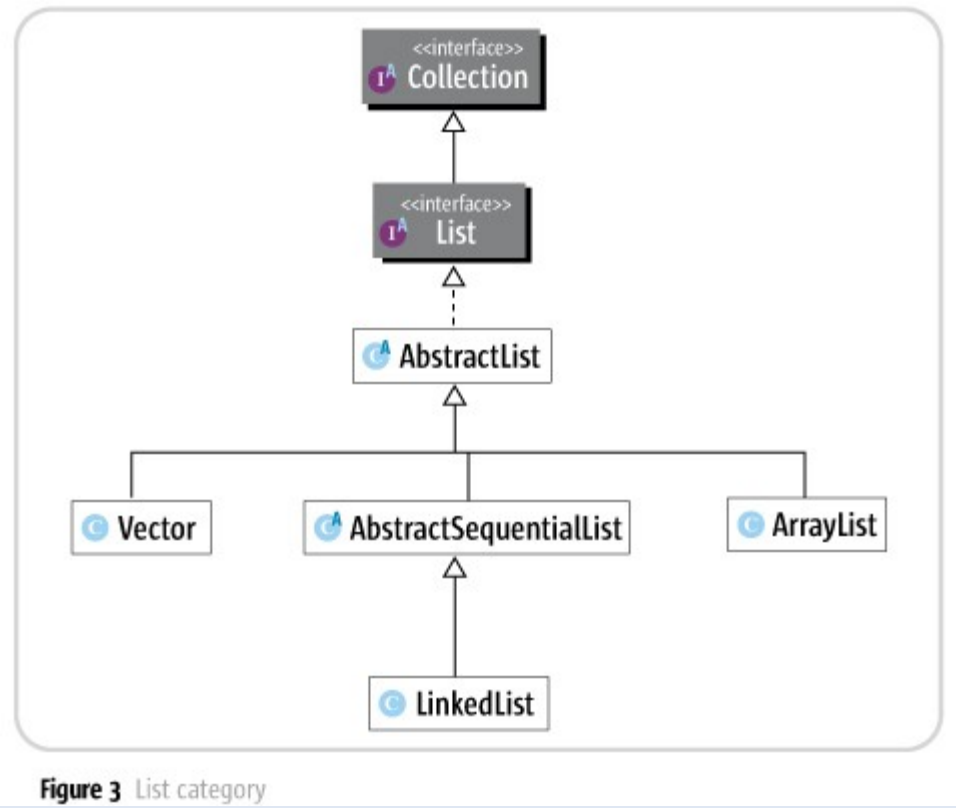


Figure 3 List category

- **ArrayList**: Bevorzugung wenn Elemente am Ende eingefügt werden
- Wahlfreier (random) Access auf die Elemente

- **LinkedList**: Bevorzugung wenn Elemente wahlweise gefügt werden sollten
- Sequenzieller Zugriff

- **Vector**: nur interessant für „threadsafed“ Zugriff

2.2.1 Java- Collections III

- Eine **1:n Beziehung** (analog zur relationalen Datenbank) wird üblicherweise unter Verwendung bestimmter *Datenstrukturen für Sammlungen (Collections) von Objekten* realisiert:
- z.B: Eine Schule „besitzt“ Adressen ihrer Lehrer und Schüler

```
public class Schule
{
   private Collection<Adresse> _coAdresse = new
      ArrayList<Adresse>();
}
```

2.2.2 Elementare Datenstrukturen (Stack 1)

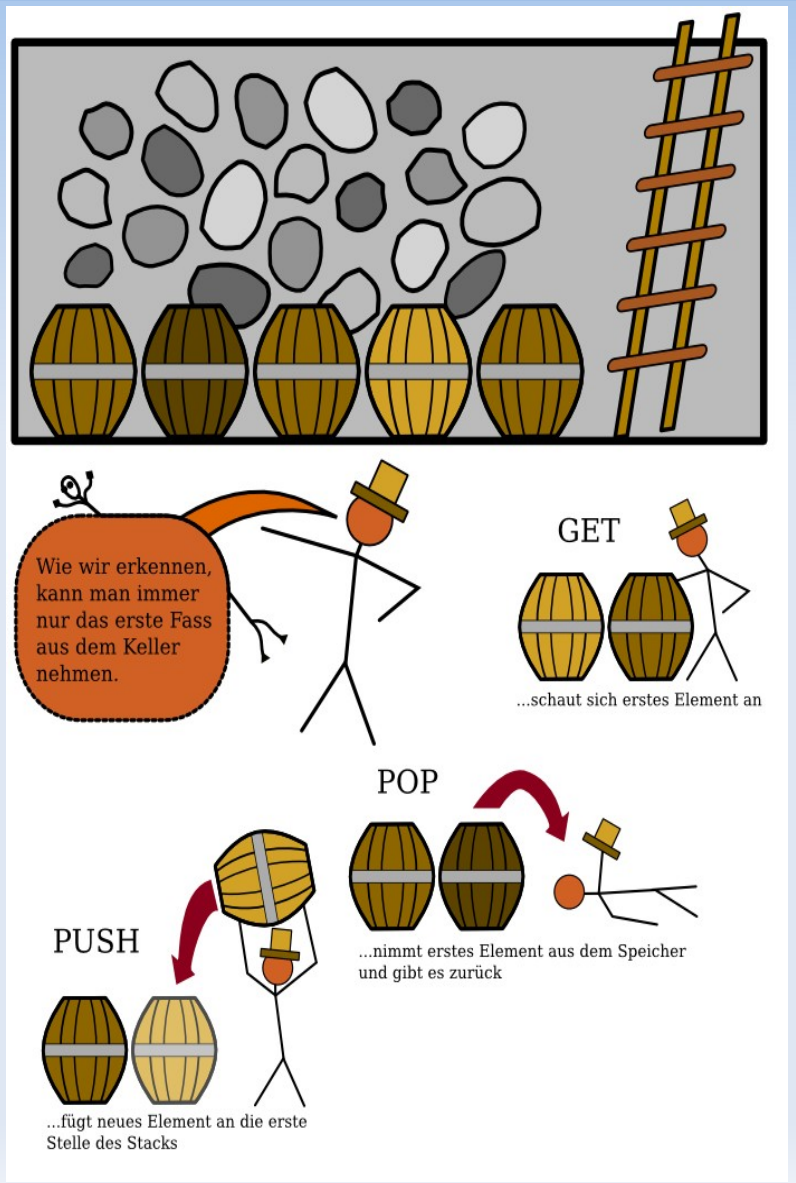
Stack (Kellerspeicher)

- Beispiele für Keller im Alltag:
  - Tellerstapel, Überfüllte Busse, Rekursive Probleme
- Operationen:
  - Anfügen nur an einem Ende (hinten)
  - Vorne sind die ältesten Elemente
  - Entfernen am gleichen Ende
- Verwendung in der Informatik:

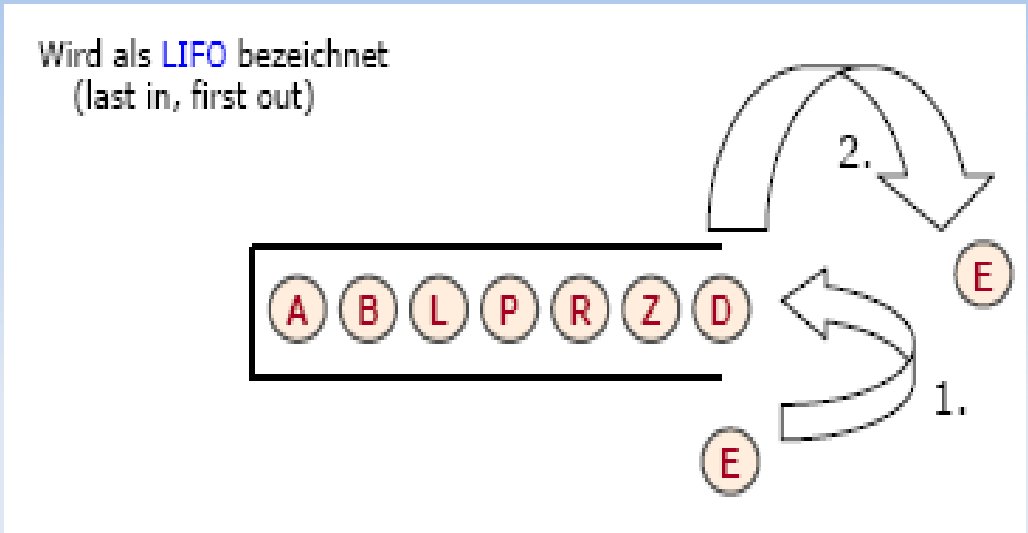
Sichern von lokalen Variablen bei Funktionsaufrufen, Labyrinth oder Schachprobleme, Syntaxanalysen

Wird als LIFO bezeichnet (last in, first out)

2.2.2 Elementare Datenstrukturen (Stack 2)



2.2.2 Elementare Datenstrukturen (Stack 2)



Kellerspeicher/Stack

- Einfügen in O(1)
- Nachfolger O(n)
- Vorgänger O(n)
- Auslesen Last-in Element O(1)

2.2.3 Elementare Datenstrukturen (Queue 1)

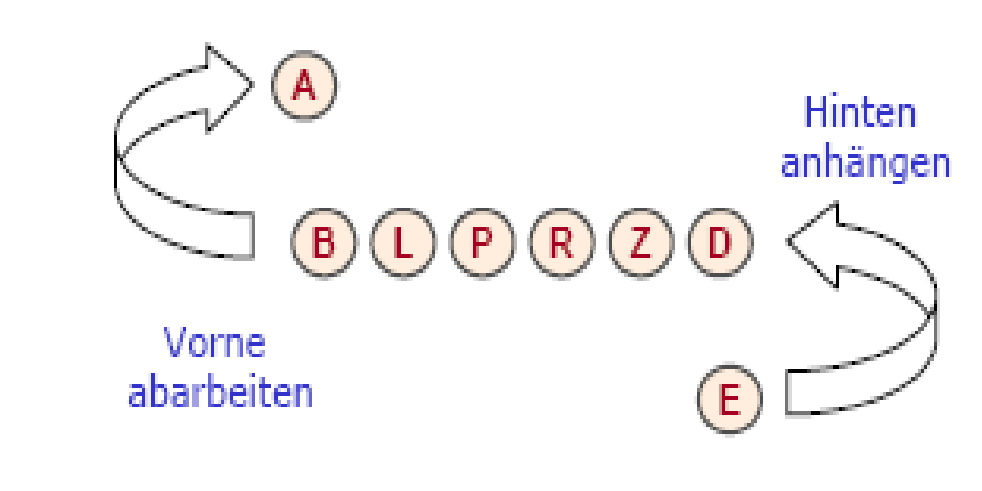
Queue (Schlangen)

- Beispiele für eine Queue im Alltag:
  Bankschalter, Supermarkt,
  Wartezimmer beim Arzt
- Operationen:
  Das erste Element wird „bedient“
  Ein neues Element wird hinten angefügt

- Verwendung in der Informatik:
  Verwaltung von Druckaufträgen oder Prozessen

Wird als **FIFO** bezeichnet (first in, first out)

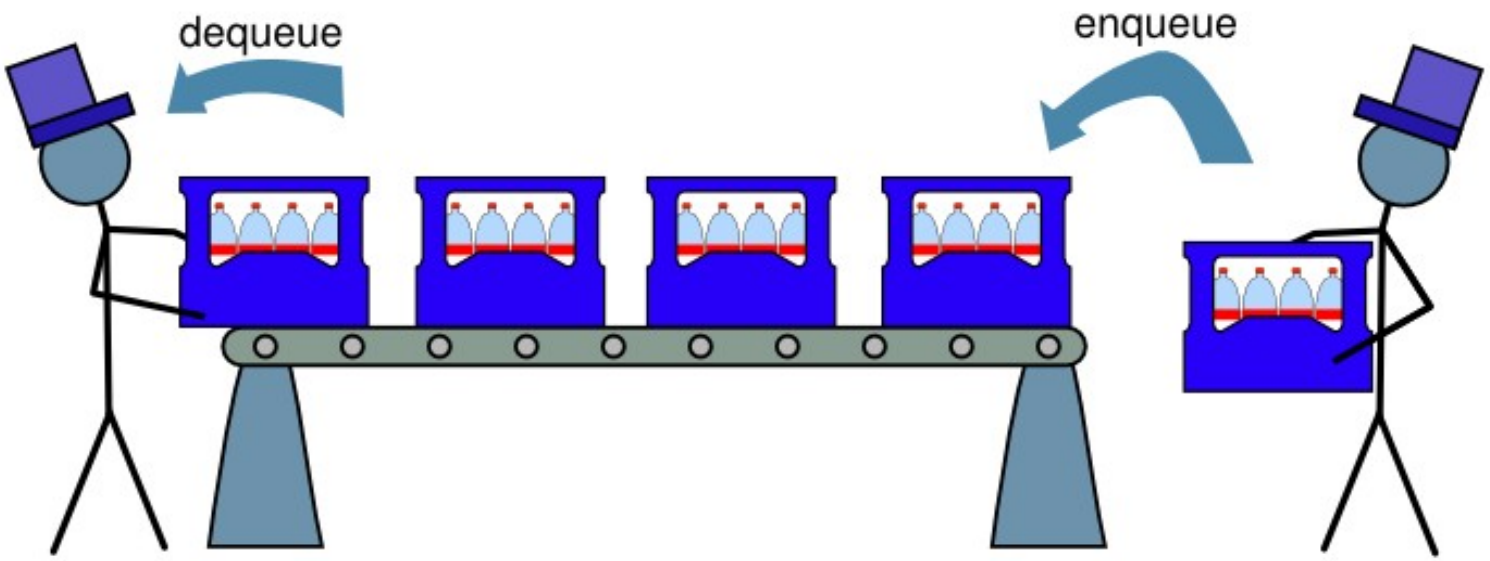
2.2.3 Elementare Datenstrukturen (Queue 2)

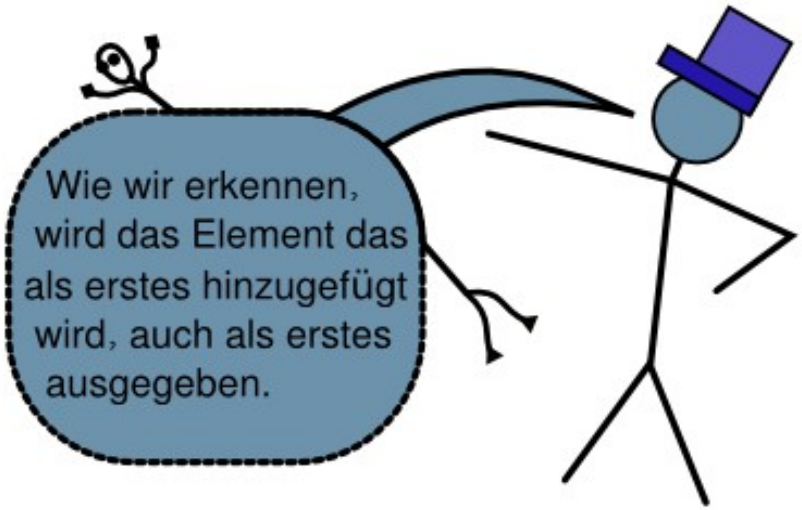


Schlangen / Queue

- Einfügen in O(1)
- Nachfolger O(n)
- Vorgänger O(n)
- Auslesen First-in Element O(1)

2.2.3 Elementare Datenstrukturen (Queue 2)





2.2.4 Elementare Datenstrukturen (Assoziative Speicher)

- Eindeutiger Objektwert wird mit einer Funktion in einen Hashwert umgeschlüsselt (Hash-Funktion)
- Die Hash-Funktion kann für unterschiedliche Objektwerte gleiche Hash-Funktionswerte liefern (=gleicher Bucket/Slot) - innerhalb des Buckets/Slots können die Werte mittels verketteter Liste abgelegt werden)
- Die Berechnung und Suche mittels der Hash-Funktion ist 'schnell' (= O(1) für insert, search, delete)
- Beispiel für assoziative Speicher:

```
java.util.HashMap
```

- Operationen:

```
put(Object oKey, Object oValue)
get(Object oKey) : Object
```

2.2.4 Elementare Datenstrukturen (Assoziative Speicher)

Beispiel: Sortierung der Schüler einer Klasse nach Größe:

Hash-Funktion: f(n) -> Größe des Schülers
Buckets (mögliche) --> 150-160 Bucket 1
161-170 Bucket 2
171-180 Bucket 3
181-225 Bucket 4

2.4 Elementare Datenstrukturen (Assoziative Speicher)

Einfügen: Schüler Huber (172 cm) --> Bucket 3
Maier (165 cm) --> Bucket 2
Muster (172 cm) --> Bucket 3

Auslesen: Schüler Muster – Größe 172 cm --> Bucket 3
Bucket 3 hat zwei Einträge – diese werden nun mittels Suche nach Namen verglichen

2.2.4 Elementare Datenstrukturen (Assoziative Speicher) / Beispiel 2

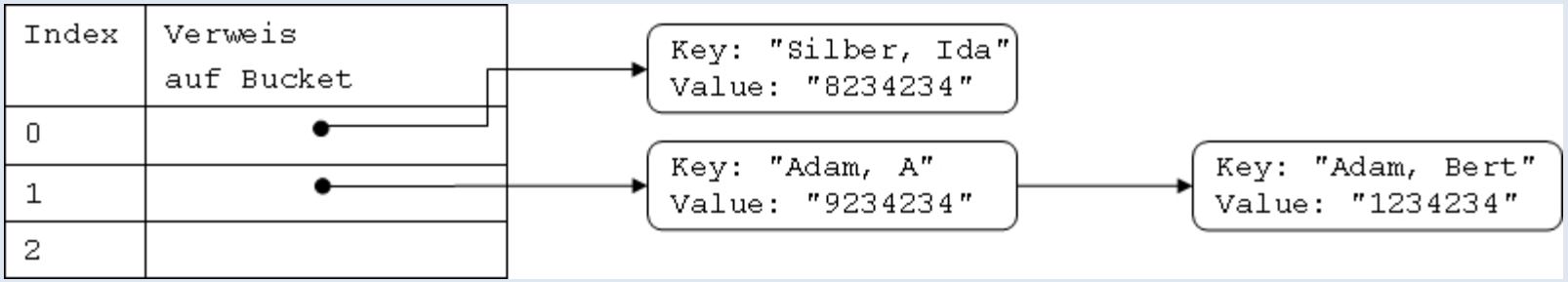
Beispiel:

put("Silber, Ida", "8234234")

hashcode = "Silber, Ida".hashCode() ---> 1831921689

index = hashcode % 3 --> 0

Jetz wird das Bucket an der zugehörigen Position eingefügt



Quelle: http://fbim.fh-regensburg.de/~hab39652/PGII/skriptum/

2.2.4 Elementare Datenstrukturen (Assoziative Speicher)

Einfügen: O(1)
Auslesen: O(1)
Löschen: O(1)

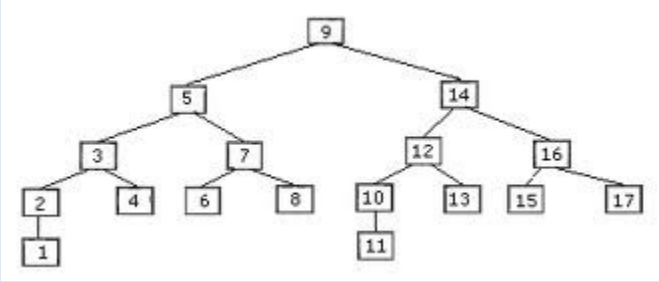
- Anzahl Elemente muss beachtet werden ('vernüftige' Größe -> Datenstruktur wird im Speicher abgelegt)
- Hashfunktion muss eine 'gute' Verteilung liefern - die schlechteste Hashfunktion (alle Werte im gleichen Slot) benötigt Laufzeiten von O(n)

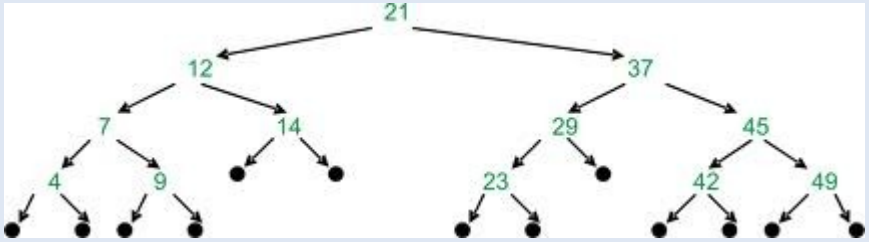
2.2.5 Elementare Datenstrukturen (Baumstrukturen - Definition)

In der Informatik werden Bäume häufig als Datenstruktur eingesetzt.

Durch Entfernen einer Kante zerfällt ein Baum in zwei Teilbäume und bildet damit einen so genannten Wald.

Quelle: http://de.wikipedia.org/wiki/Baum_%28Graphentheorie%29





2.2.5 Elementare Datenstrukturen (Baumstrukturen - Definition)

- Sei N eine nicht-leere Menge von Knoten ni (nodes).
- Ein Knoten n0 besitzt eine (möglicherweise leere) Menge von Nachfolger-Knoten n1 ... nm

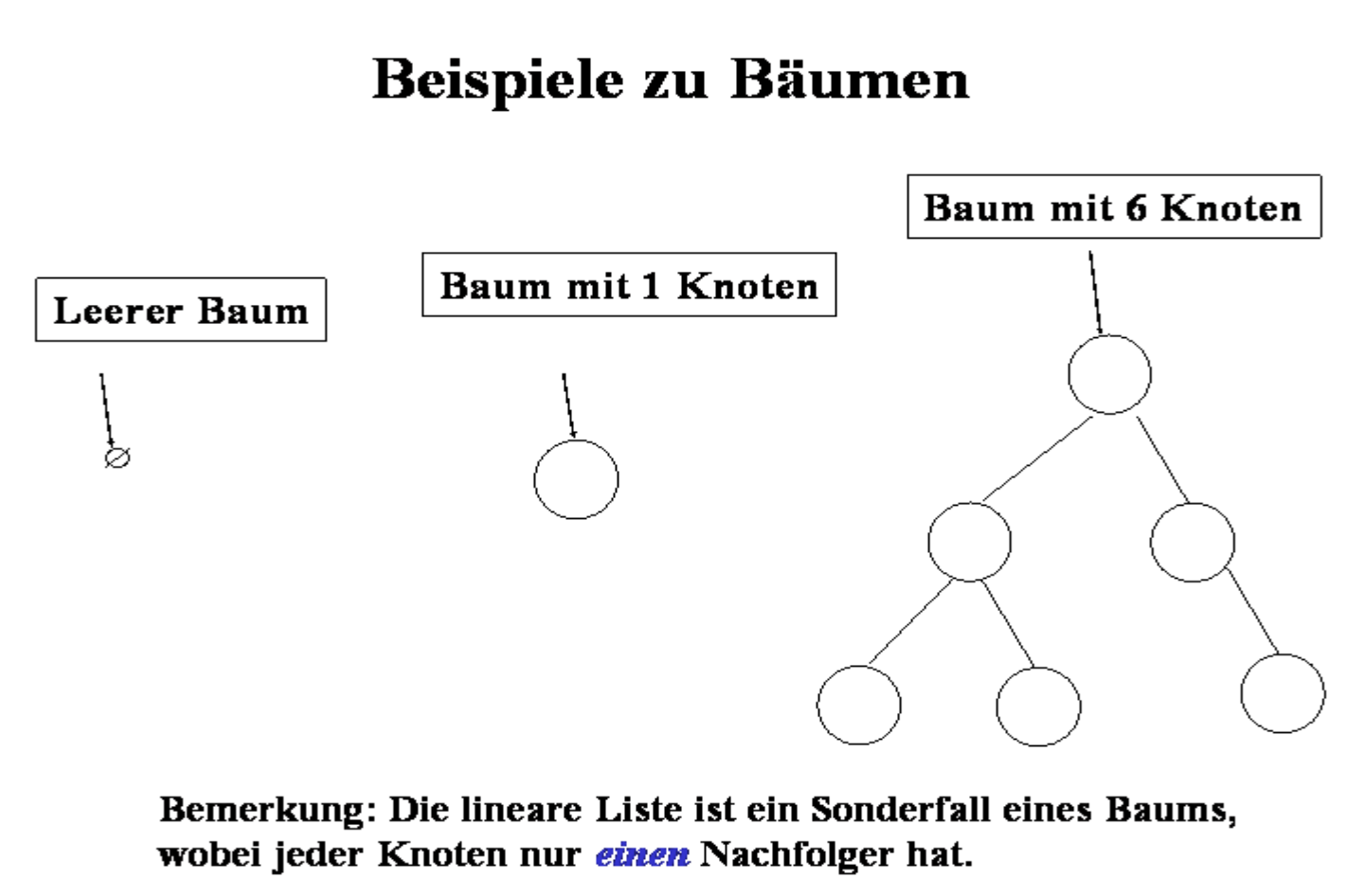
Diese werden auch **Kinder**, children, sons von n genannt.

- b ist Vorgänger (auch Elternteil, Vater, parent) von a
  <-> a ist ein Kind von b
- Ein Knoten heißt **Wurzel** des Baumes, wenn er keinen Vater hat
- ni und nj sind Geschwister (**siblings**)

<-> Vater(ni) = Vater(nj)

- Eine Kante geht von n0 nach ni , 1 ≤ i ≤ m (m Anzahl Kinder)
- Ein Knoten, der keine Nachkommen besitzt, wird **Blatt** (leaf) genannt.

2.5 Elementare Datenstrukturen (Baumstrukturen - Definition)



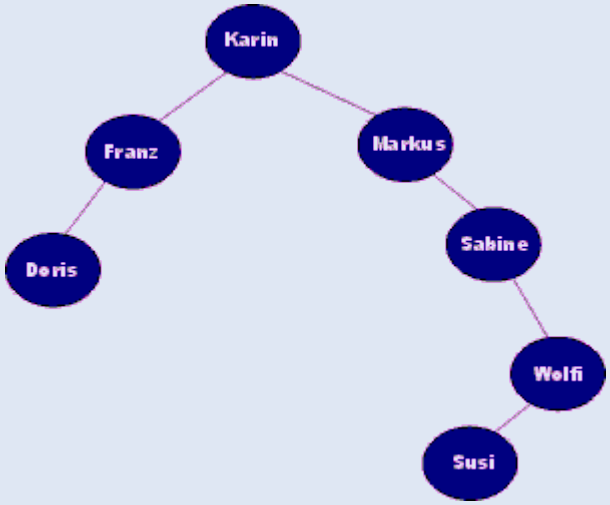
Quelle: http://www.saar.de/~awa/images/jbaum03.gif

2.2.5 Elementare Datenstrukturen (binäre Bäume)

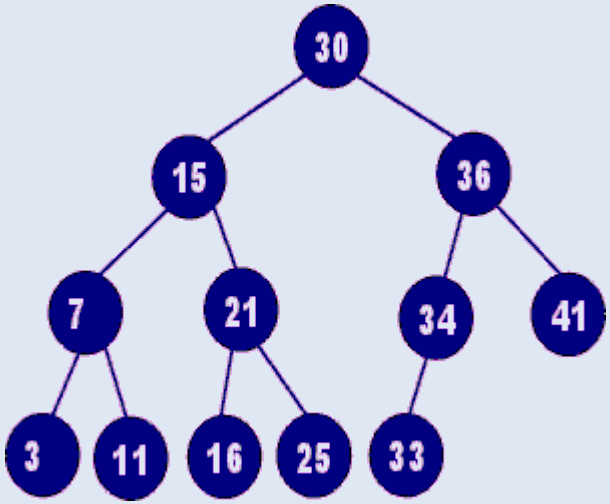
- Ein Baum der Ordnung n heißt n-ärer Baum
- Ein Baum der Ordnung 2 heißt binärer Baum

d.h. Jeder Knoten eines Binärbaums hat maximal 2 Kinder

- Ein Balancierter Baum ist ein Baum, bei dem für alle Paare von Blättern(a,b) gilt: | Tiefe(a) – Tiefe(b) | ≤ 1



Degenerierter Baum

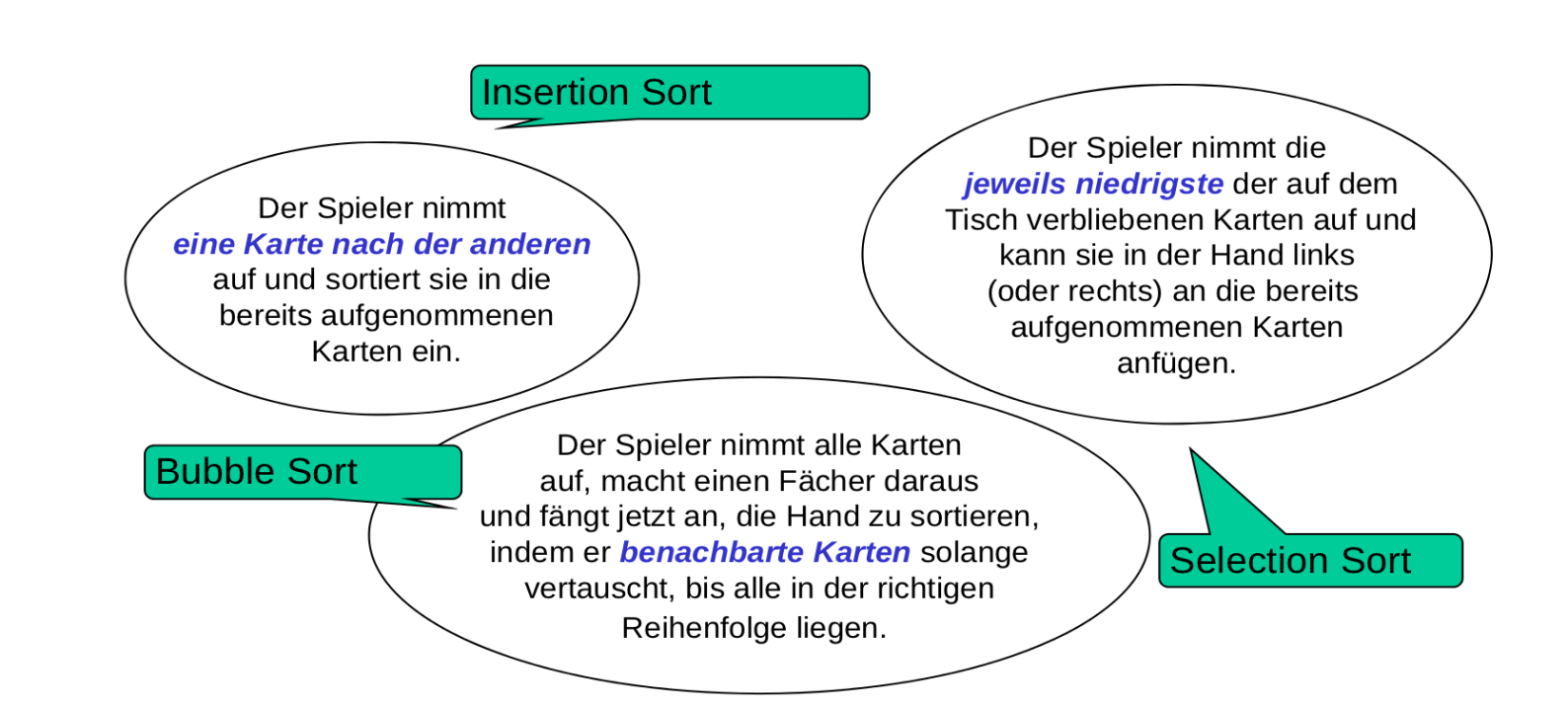


Balancierter Baum

2.3 Elementare Sortieralgorithmen

Motivation:

Sortieren zur Steigerung der Effizienz eines Algorithmus: gewisse (schnelle) Algorithmen funktionieren nur wenn die Daten sortiert sind, wie zum Beispiel Binäre Suche



[Quelle: https://home.zhaw.ch/~frup/ADS/Unterlagen/10_Sortieren1_4.pdf]

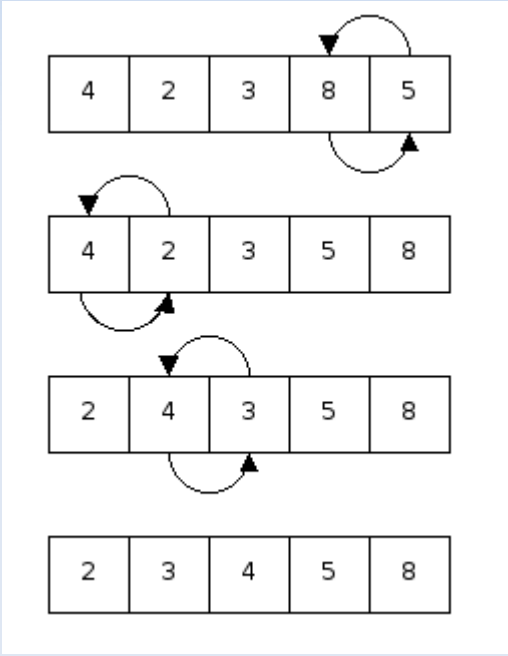
2.3.1 Elementare Sortieralgorithmen Vergleichbasiert - Bubble-Sort

Video: http://www.youtube.com/watch?v=lyZQPjUT5B4&feature=player_embedded#!

Das Sortieren durch Aufsteigen (englisch Bubble sort, "Blasensortierung") bezeichnet einen einfachen, stabilen Sortieralgorithmus, der eine Reihe zufällig angeordneter Elemente (etwa Zahlen) der Größe nach ordnet.

Bubblesort wird von Donald E. Knuth als vergleichsbasierter Sortieralgorithmus bezeichnet [1]. Das bedeutet, dass der Sortieralgorithmus sämtliche Entscheidungen alleine auf Basis des Größenvergleichs je zweier Elemente fällt, nicht etwa aufgrund der Inspektion der Binärdarstellung eines Elements.

Bubblesort ist der einfachste solcher Algorithmen. Einzige Anforderung an den Vergleichsoperator ist, dass er eine Totalordnung der Liste ermöglicht.



2.3.1 Elementare Sortieralgorithmen Vergleichbasiert - Bubble-Sort

Bubblesort gehört zur Klasse der In-place-Verfahren, was bedeutet, dass der Algorithmus zum Sortieren keinen zusätzlichen Arbeitsspeicher außer den lokalen Laufvariablen der Prozedur benötigt. Dadurch, dass grundsätzlich nur aneinandergrenzende Elemente miteinander vertauscht werden, eignet sich dieses Verfahren auch zum Sortieren von Listen mit unterschiedlich großen Elementen.

Pseudo-Code für den Algorithmus:

```
prozedur bubbleSort( A : Liste sortierbarer Elemente )
  n := Länge( A )
  wiederhole
    vertauscht := falsch
    für jedes i von 1 bis n - 1 wiederhole
      falls A[ i ] > A[ i + 1 ] dann
        vertausche( A[ i ], A[ i + 1 ] )
        vertauscht := wahr
      ende falls
    ende für
    n := n - 1
  solange vertauscht und n > 1
prozedur ende
```

2.3.1 Elementare Sortieralgorithmen
Vergleichbasiert - Bubble-Sort

Bubblesort hat die Laufzeit **$O(n^2)$** für Listen der Länge n.

Bester Fall

Falls die Liste bereits sortiert ist, wird Bubblesort die Liste
nur einmal durchgehen, um festzustellen, dass die Liste bereits
sortiert ist, weil keine benachbarten Elemente vertauscht
werden mussten.
Daher benötigt Bubblesort **O(n)** Schritte,
um eine bereits sortierte Liste zu bearbeiten.

Hasen und Schildkröten

Die Positionen der Elemente vor dem Sortieren entscheiden
maßgeblich den Sortieraufwand.
Große Elemente am Anfang wirken sich nicht gravierend aus,
da sie schnell nach hinten getauscht werden, kleine Elemente
am Ende bewegen sich jedoch eher langsam nach vorne.
Darum spricht man bei diesen Elementen von Hasen und Schildkröten.

[Quelle: http://de.wikipedia.org/wiki/Bubblesort]

2.3.2 Elementare Sortieralgorithmen Vergleichbasiert - Quick-Sort

Quicksort (von englisch quick ‚schnell' und to sort ‚sortieren') ist ein schneller, rekursiver, nicht-stabiler Sortieralgorithmus, der nach dem Prinzip Teile und herrsche (lat. Divide et impera!, engl. Divide and conquer) arbeitet.

Er wurde ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt[1] und seitdem von vielen Forschern verbessert. Der Algorithmus hat den Vorteil, dass er über eine sehr kurze innere Schleife verfügt (was die Ausführungsgeschwindigkeit stark erhöht) und ohne zusätzlichen Speicherplatz auskommt (abgesehen von dem für die Rekursion zusätzlichen benötigten Platz auf dem Aufruf-Stack).

Algorithmus – Funktion quicksort(links, rechts)

```
funktion quicksort(links, rechts)
    falls links < rechts dann
        teiler := teile(links, rechts)
        quicksort(links, teiler-1)
        quicksort(teiler+1, rechts)
    ende
ende
```

2.3.2 Elementare Sortieralgorithmen Vergleichbasiert - Quick-Sort

Algorithmus – Funktion teiler(links, rechts)

```
funktion teile(links, rechts)
    i := links
    // Starte mit j links vom Pivotelement
    j := rechts - 1
    pivot := daten[rechts]

    wiederhole

        // Suche von links ein Element, welches größer als das Pivotelement ist
        wiederhole solange daten[i] ≤ pivot und i < rechts
            i := i + 1
        ende

        // Suche von rechts ein Element, welches kleiner als das Pivotelement ist
        wiederhole solange daten[j] ≥ pivot und j > links
             j := j - 1
        ende

        falls i < j dann
            tausche daten[i] mit daten[j]
        ende

    solange i < j // solange i an j nicht vorbeigelaufen ist

    // Tausche Pivotelement (daten[rechts]) mit neuer endgültiger Position (daten[i])
    falls daten[i] > pivot dann
            tausche daten[i] mit daten[rechts]
    ende

    // gib die Position des Pivotelements zurück
    antworte i
ende
```

2.3.2 Elementare Sortieralgorithmen Vergleichbasiert - Quicksort

Laufzeitverhalten

Die Laufzeit des Algorithmus hängt im wesentlichen von der Wahl des Pivotelementes ab.

Im Worst Case (schlechtesten Fall) wird das Pivotelement stets so gewählt, dass es das größte oder das kleinste Element der Liste ist.
Dies ist etwa der Fall, wenn als Pivotelement stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt.
Die zu untersuchende Liste wird dann in jedem Rekursionsschritt nur um eins kleiner und die Zeitkomplexität wird beschrieben durch $\mathbf{O(n^2)}$.

Im Best Case (bester Fall) wird das Pivotelement stets so gewählt, dass die beiden entstehenden Teillisten etwa gleich groß sind. In diesem Fall gilt für die asymptotische Laufzeit des Algorithmus **O(n log (n))**.

Es gibt Algorithmen, beispielsweise Heapsort, deren Laufzeit auch im Worst Case durch **O(n log (n))** beschränkt sind. In der Praxis wird aber trotzdem Quicksort eingesetzt, da angenommen wird, dass bei Quicksort der Worst Case nur sehr selten auftritt und im mittleren Fall schneller als Heapsort ist, da die innerste Schleife von Quicksort nur einige wenige, sehr einfache Operationen enthält.

Quelle: http://de.wikipedia.org/wiki/Quicksort

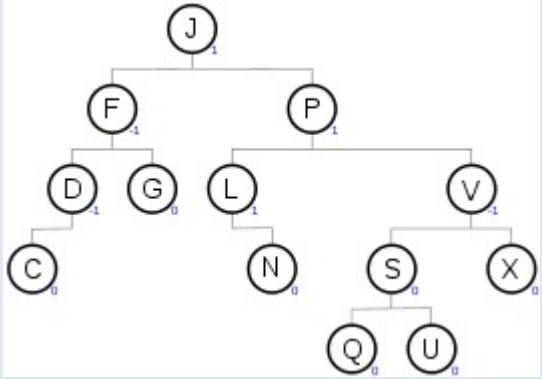
2.3.3 Elementare Sortieralgorithmen
Vergleichbasiert - binäre Bäume

In der Informatik ist ein binärer Suchbaum eine spezielle Implementierung der abstrakten Datenstruktur Suchbaum.

Ein binärer Suchbaum ist ein binärer Baum, bei dem die Knoten des linken Teilbaums eines Knotens **nur kleinere (oder gleiche) Schlüssel** und die Knoten des rechten Teilbaums eines Knotens **nur größere (oder gleiche) Schlüssel** als der Knoten selbst besitzen.

Die Bedeutung der Begriffe „kleiner gleich“ und „größer gleich“ ist völlig dem Anwender überlassen.
Sie müssen nur eine Totalordnung (genauer: eine totale Quasiordnung s. u.) darstellen – zumeist realisiert durch eine vom Anwender zur Verfügung zu stellende 3-Wege-Vergleichsfunktion.

[Quelle: http://de.wikipedia.org/wiki/Bin%C3%A4rer_Suchbaum]

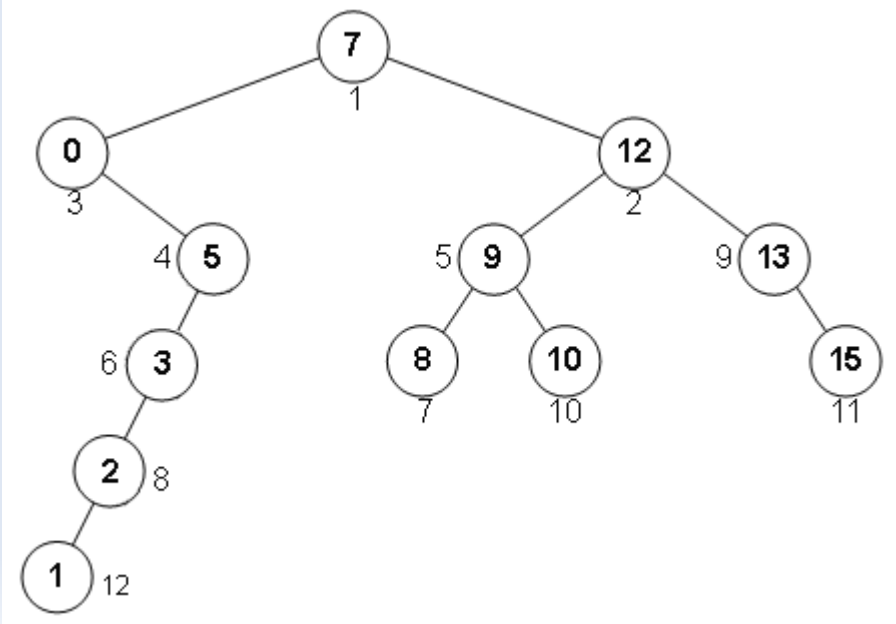


2.3.3 Elementare Sortieralgorithmen
Vergleichbasiert - binäre Bäume

Ein binärer Baum heißt binärer Suchbaum

- B ist leer oder
- Wenn B nicht leer ist gilt
  der rechte und der linke Unterbaum von B sind ebenfalls binäre Suchbäume.
  Ist w der Wert in der Wurzel, so sind alle Werte in den Knoten des linken Unterbaums kleiner als w und die Werte aller Knoten des rechten Unterbaums sind größer als w.
  (Insbesondere kommt also ein Wert nur einmal in einem Binären-Such-Baum vor, es sei denn, man ersetzt die Relation größer durch größer gleich.)

Beispiel:
Liste [7, 12, 0, 5, 9, 3, 8, 2, 13, 10, 15, 1]



Quelle: http://www.pohlig.de/Unterricht/Inf2002/Tag43/28.12_Was_ist_ein_binaerer_Suchbaum.htm

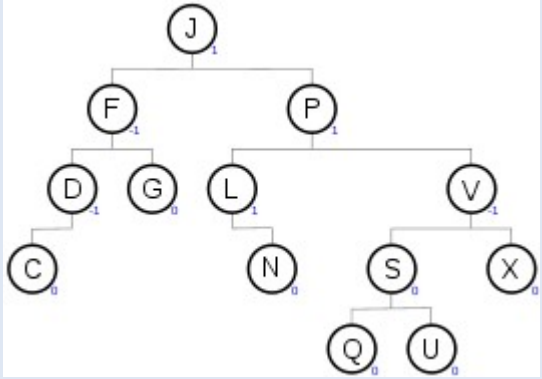
2.3.3 Elementare Sortieralgorithmen
Vergleichbasiert - binäre Bäume

Der AVL-Baum ist eine Datenstruktur in der Informatik, genauer ein **balancierter binärer Suchbaum.**
Als Invariante beim AVL-Baum gilt, dass sich für jeden Knoten die Höhen der beiden Teilbäume um **höchstens 1** unterscheiden.

Diese Bedingung verhindert (bei einem moderaten Aufwand), dass der Baum zu sehr aus der Balance gerät. Die Höhe eines AVL-Baums mit n Knoten liegt in O(log n)
und damit auch die maximale Anzahl der Schritte, um ein Element zu finden oder festzustellen, dass es nicht enthalten ist.

Der Name AVL leitet sich von den Erfindern Adelson-Welski und Landis ab, die diese Datenstruktur 1962 entwickelten.[1] Der AVL-Baum ist damit die älteste Datenstruktur für balancierte Bäume.

[Quelle: http://de.wikipedia.org/wiki/AVL-Baum]



2.3.3 Elementare Sortieralgorithmen Vergleichbasiert - binäre Bäume

- Laufzeitverhalten balancierter Binärbaum:
  - Sortieren einer Liste mit n Elementen O(n log(n))
  - Einfügen O(log(n))
  - Lesen O(log(n))
  - Löschen O(log(n))

- Laufzeitverhalten degenerierter Binärbaum (worst case):
  - Einfügen O(n)
  - Lesen O(n)
  - Löschen O(n)

2.3.4 Elementare Sortieralgorithmen
Nicht-vergleichsbasiert - Bucketsort

Bucketsort (von engl. bucket „Eimer“) ist ein Sortierverfahren, das für bestimmte Werte-Verteilungen eine Eingabe-Liste in **linearer** Zeit sortiert **(O(n))**.

Der Algorithmus ist in drei Phasen eingeteilt:

- Verteilung der Elemente auf die Buckets (Partitionierung)
- Jeder Bucket wird mit einem weiteren Sortierverfahren wie beispielsweise Insertionsort sortiert.
- Der Inhalt der sortierten Buckets wird konkateniert.

Quelle: https://de.wikipedia.org/wiki/Bucketsort
http://all-free-download.com/free-icon/icons/bucket_89351.html