

## Algorithmen und Datenstrukturen ELV

Dr. Jürgen Falb

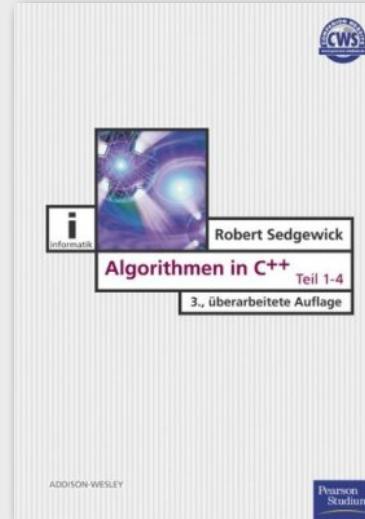
[juergen.falb@fh-campuswien.ac.at](mailto:juergen.falb@fh-campuswien.ac.at)

## Inhalte

Block	Thema
1	Einführung, Grundlagen, Entwurf und Analyse von Algorithmen, Rekursionen, Klassen von Algorithmen
2	Datenstrukturen (Verkettete Listen, Stacks, Queues, Hashtabellen)
3	Hashverfahren, Sortierverfahren (Insertion-Sort, Selection-Sort, Merge-Sort, Quick-Sort)
4	Baumstrukturen (Binäre Bäume, Binäre Suchbäume, AVL-Bäume, Heaps, Heap-Sort, Priority Queue)
5	Suchalgorithmen (Lineare Suche, Binäre Suche, Tries), Graphen (Definition und Darstellung)
6	Graphenalgorithmen (Durchlauf von Graphen, Breiten- und Tiefensuche, kürzeste Wege, Spannbäume)
7	Spezialalgorithmen (Durchfluss, Constraint-Satisfaction-Probleme, Genetische Algorithmen)
8	Textsuche (Brute Force, Knuth-Morris-Pratt, Boyer-Moore-Horspool, Regular Expressions)

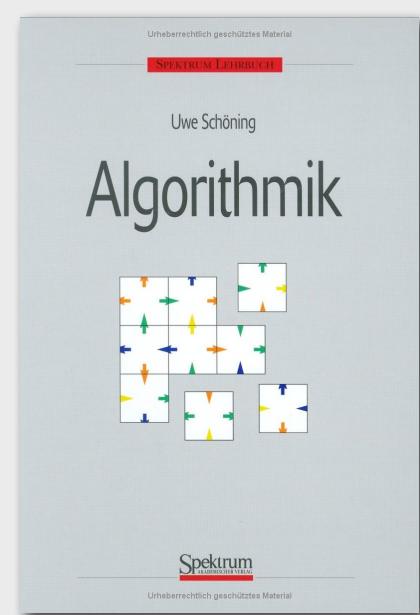
## Literaturempfehlungen (I)

- > Für die „Praktiker“:
  - » Algorithmen in C++
  - » Robert Sedgewick
  - » 3. Auflage 2002
  - » Addison-Wesley Longman Verlag
  - » ISBN 978-3827370266



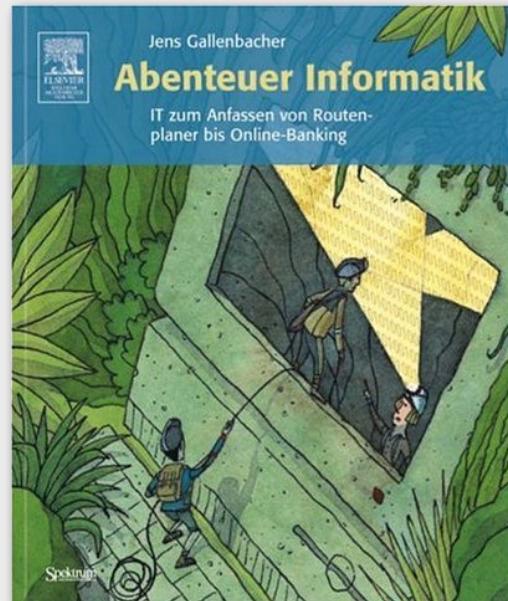
## Literaturempfehlungen (II)

- > Für die „Theoretiker“ und „Mathematiker“:
  - » Algorithmik
  - » Uwe Schöning
  - » 1. Auflage 2001
  - » Spektrum Akademischer Verlag
  - » ISBN 978-3827410924



### Literaturempfehlungen (III)

- > Für die „Lesebegeisterten“:
  - » Abenteuer Informatik
  - » Jens Gallenbacher
  - » 1. Auflage 2006
  - » Spektrum Akademischer Verlag
  - » ISBN 978-3827416353



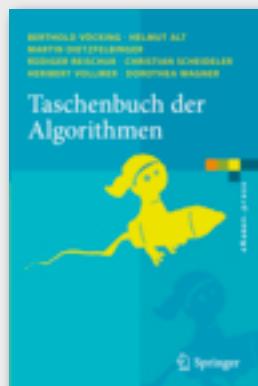
Erstellt von: Jürgen Falb

10.02.2017

Seite 5

### Literaturempfehlungen (IV)

- > „Algorithmus der Woche“
- > Anschauliche Erklärungen einzelner Algorithmen im Rahmen des Informatikjahrs 2006
- > <http://www-i1.informatik.rwth-aachen.de/~algorithmus/>
- > Ist bei Springer auch in Buchform erhältlich!



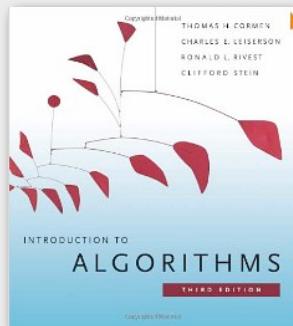
Erstellt von: Jürgen Falb

10.02.2017

Seite 6

## Literaturempfehlungen (V)

- > „Introduction to Algorithm“ - Mein persönlicher Favorit
- > Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, und Clifford Stein
- > Mit Press; Auflage: 3rd edition. ISBN 978-0262033848
- > Deutsche Übersetzung der älteren Auflage verfügbar.



## Grundlagen

## Was ist eine Datenstruktur?

- > Daten sind in allen möglichen Größen und Formen vorhanden:
  - » Buchliste für eine Vorlesung
  - » Zutaten für ein Rezept
  - » Adressliste für einen Newsletter
- > Daten können aber meist in gleicher Art und Weise organisiert / strukturiert werden:
  - » z.B. eine einfache Liste, in der ein Element auf das nächste folgt.  
→ einfache Datenstruktur

## Organisation von Daten

- > Typische Organisationsstrukturen f. Daten:
  - » Listen
  - » Queues (Warteschlangen)
  - » Stacks (Stapel)
  - » Sets (Mengen)
  - » Hash Tables (Assoziative Tabellen)
  - » Trees (Baumstrukturen)
  - » Graphen
  - » Heaps (Haufen)

## Gründe f. Verwendung v. Datenstrukturen (I)

- > Effizienz
  - » Datenstrukturen organisieren Daten in einer Weise, die Algorithmen effizienter macht.
  - » Beispiel Suchprobleme: Daten eher in einer Hashtabelle oder in einem Suchbaum speichern, als in einem Array.
- > Abstraktion
  - » Datenstrukturen ermöglichen eine verständlichere Betrachtungsweise der Daten.
  - » → Stellen für die Problemlösung eine Abstraktionsebene zur Verfügung.
  - » → Erlauben es, über Programme in einer weniger programmtechnisch orientierten Form zu sprechen.

## Gründe f. Verwendung v. Datenstrukturen (II)

- > Wiederverwendbarkeit
  - » Datenstrukturen sind wiederverwendbar weil sie dazu tendieren modular und kontextunabhängig zu sein.
  - » **Modular:** Es gibt eine vorgeschriebene Schnittstelle für den Zugriff auf die Datenstruktur.
    - Zugriff auf die Daten nur über Operationen (Suche, Füge\_Ein, Lösche, ...)
  - » **Kontextunabhängig:** Strukturen können mit allen Arten von Daten und in einer Vielzahl von Situationen und Kontexten verwendet werden.

## Abstrakter Datentyp (ADT)

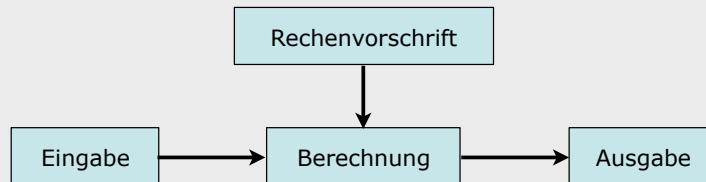
- > Datenstrukturen sind immer auch verbunden mit den Operationen bzw. Aktionen die man mit ihnen durchführen kann:
  - » Z.B.: Operationen auf Liste: Elemente einfügen, entfernen, durchgehen und zählen
- > Die Datenstruktur zusammen mit solch elementaren Operationen wird als **abstrakter Datentyp (ADT)** bezeichnet.
- > Die Operationen eines ADT bestehen aus seiner **öffentlichen Schnittstelle**.
- > Keine Rücksicht auf die programmtechnische Realisierung (abstrakt).

## Was ist ein Algorithmus? (I)

- > Algorithmen sind wohldefinierte Prozeduren zur Lösung von Problemen
- > Algorithmen sind in der Informatik von grundlegender Bedeutung
  - » entsprechen genau den systematischen Prozeduren, die ein Computer verlangt
- > Ein guter Algorithmus ist wie das richtige Werkzeug in einer Werkstatt
- > Übernimmt die gestellte Aufgabe mit genau dem richtigen Maß an Aufwand

## Was ist ein Algorithmus? (II)

- > Algorithmus - abgeleitet vom Namen des Gelehrten Muhammed Al Chwarizmi (ca. 800). Er schrieb das Lehrbuch „*Vom Rechnen mit indischen Zahlen*“
- > Brockhaus: Ein Algorithmus ist ein Rechenverfahren, das in genau festgelegten Schritten vorgeht.
- > Algorithmus besitzt:
  - » eine Eingabe
  - » eine Ausgabe
  - » sowie eine Berechnung, die durch eine Rechenvorschrift vorgegeben ist



## Was ist ein Algorithmus? (III)

- > nach Wirth sind Programme letztlich konkrete Formulierungen abstrakter Algorithmen, die sich auf bestimmte Darstellungen wie Datenstrukturen stützen
- > Programmerstellung und Datenstrukturierung sind untrennbare ineinandergreifende Themen

## Eigenschaften von Algorithmen (I)

- > Allgemeine Eigenschaften:
  - » **Finitheit:** Das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein.
  - » **Ausführbarkeit:** Jeder Schritt des Verfahrens muss auch tatsächlich ausführbar sein.
  - » **Platzkomplexität:** Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen.
  - » **Terminiertheit:** Das Verfahren darf nur endlich viele Schritte benötigen.

## Eigenschaften von Algorithmen (II)

- > In der Praxis müssen zusätzlich folgende Eigenschaften erfüllt sein:
  - » **Determiniertheit:** Der Algorithmus muss bei denselben Voraussetzungen das gleiche Ergebnis liefern.
  - » **Determinismus:** Die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert.

## Gründe f. Verwendung von Algorithmen (I)

- > Drei Gründe für die Verwendung formaler Algorithmen:
  - » Effizienz
    - Bestimmte Arten von Problemen kommen in der Informatik häufig vor. Mit der Zeit wurden für diese Probleme effiziente Lösungen gefunden.
  - » Abstraktion
    - Algorithmen stellen bei der Problemlösung eine Abstraktionsebene dar. Aufteilen von komplexen Problemen in einfachere Probleme, für die es Lösungen gibt.
    - Beispiel: Kürzester Weg eines Datenpaketes zw. zwei Rechnern im Internet finden, Lösung durch Algorithmus: Finden des kürzesten Weges zwischen zwei Punkten.

## Gründe f. Verwendung von Algorithmen (II)

- » Wiederverwendbarkeit
  - Algorithmen können häufig für unterschiedliche Situationen verwendet werden.
  - Viele allgemein bekannte Algorithmen lösen Probleme, bei denen es sich um Verallgemeinerungen etwas komplizierterer Probleme handelt.
  - Viele komplexe Probleme können in einfachere Teilprobleme aufgeteilt werden.

## Korrektheit von Algorithmen (I)

- > Ein Algorithmus heißt korrekt, wenn seine Durchführung für jede mögliche und gültige Eingabeinstanz mit der korrekten Ausgabe terminiert.
- > Ein korrekter Algorithmus löst das gegebene Problem.

## Korrektheit von Algorithmen (II)

- > **Spezifikation:** Eindeutige Festlegung der berechneten Funktion und des Terminierungsverhaltens einer Funktion
- > **Verifikation:** Formaler mathematischer Beweis dass ein Algorithmus korrekt bezüglich einer Spezifikation ist
- > **Validation:** (Nichtformaler) Nachweis der Korrektheit durch Testen

### Korrektheit von Algorithmen (III)

> Vor- und Nachbedingungen

{ VOR } ANW { NACH }

> VOR und NACH sind Aussagen über den Zustand vor bzw. nach der Ausführung von ANW

→ Gilt die Bedingung VOR unmittelbar vor Ausführung von ANW und terminiert ANW, dann gilt NACH unmittelbar nach Ausführung von ANW

### Korrektheit von Algorithmen (IV)

> Beispiele:

Vorbed.	Algorithmus	Nachbed.	Korr.
{X=0}	X=X+1	{X=1}	wahr
{true}	X=Y	{X=Y}	wahr
{Y=a}	X=Y	{X=a $\wedge$ Y=a}	wahr
{X=a $\wedge$ Y=b}	X=Y; Y=X	{X=b $\wedge$ Y=a}	falsch
{X=a $\wedge$ Y=b}	Z=X; X=Y; Y=Z	{X=b $\wedge$ Y=a}	wahr

**Beispiel: Parkplatzproblem**

> **Aufgabe:** Auf einem Parkplatz befinden sich Motorräder (2 Räder) und PKWs (4 Räder). Zusammen sind es  $n$  Fahrzeuge mit insgesamt  $r$  Rädern. Bestimmen Sie die Anzahl  $P$  der PKWs.

**Parkplatzproblem**

$$P + M = n$$

$$4P + 2M = r$$

Durch Lösung obigen Gleichungssystems ergibt sich für die Anzahl  $P$  der PKWs:

$$P = (r - 2n) / 2$$

## Parkplatzproblem

- > Eingabe von z.B.  $r=1$ , und  $n=2$  ergibt Anzahl der PKWs von  $P = -1.5$ .
- > Fehler durch Abstraktion
  - » Entwickler erstellt abstraktes mathematisches Modell der Wirklichkeit (Parkplatz)
  - » → Mathematik hat aber von Autos, Motorrädern, und PKWs „keine Ahnung“
  - » → Es wurden zu viele wichtige Details ignoriert

## Parkplatzproblem – Spezifikation

**Eingabe:**  $n, r \in \mathbb{N}$

**Vorbedingung:**  $r$  ist gerade,  $2n \leq r \leq 4n$

**Ausgabe:**  $P \in \mathbb{N}$ , falls Nachbedingung erfüllbar, sonst: „Keine Lösung“

**Nachbedingung:** Für gewisse  $M, P \in \mathbb{N}$  gilt

$$M + P = n$$

$$2M + 4P = r$$

## Entwurf von Algorithmen

## Spezifikation von Algorithmen

- > Ein Algorithmus kann auf verschiedene Art und Weise spezifiziert werden:
  - » Als Text (Deutsch, Englisch, Pseudocode, ...)
  - » Als Computerprogramm
  - » Als Hardwaredesign
- > Am besten formuliert man in Pseudocode:
  - » Weglassen aller Details (notwendige Zeilen und Befehle) die vom eigentlichen Algorithmus ablenken.
  - » Also alles weglassen was „unmittelbar klar“ ist.
  - » Nur auf das wesentliche konzentrieren.
  - » Merkregel: bis zu 3-5 Programmierzeilen können zu einer Zeile Pseudocode zusammengefasst werden.

## Entwurf von Algorithmen (I)

- > Im weitesten Sinne gehen viele Algorithmen Probleme auf die gleiche Art und Weise an.
  - » Daher ist es häufig praktisch, Algorithmen nach dem verwendeten Ansatz zu klassifizieren.
  - » Gewisser Einblick in einen Algorithmus ist gegeben, wenn allgemeiner Ansatz verstanden wird.
  - » Bringt eine gewisse Vorstellung, wie ähnlich gelagerte Probleme gelöst werden können, für die es noch keinen Algorithmus gibt.
- > Klassifizierung ist allerdings nicht für alle Algorithmen möglich.
- > Manche Probleme sind nur durch Kombination von Algorithmen lösbar.

## Entwurf von Algorithmen (II)

- > Algorithmenmuster:
  - » Entwicklung generischer Muster für allgemeine Problemklassen.
  - » Anpassen des Musters an konkretes Problem.
- > Schrittweise Verfeinerung:
  - » Entwerfe Pseudocode der Problem „grob“ beschreibt.
  - » Ersetze Teile des Pseudocodes durch verfeinerten Pseudocode.
  - » Implementiere Pseudocode in einer Programmiersprache.

## Entwurf von Algorithmen (III)

> Rekursion:

- » Programmteile rufen sich selbst wieder auf bis eine Abbruchbedingung erfüllt ist.
- » Compiler formen Rekursionen oft in Iterationen um.
- » Gefahr eines Stack Overruns!

## Exkurs: Rekursion

## Rekursion (I)

- > Ein leistungsfähiges Prinzip, das es erlaubt, etwas über kleinere Versionen seiner selbst zu definieren
- > Beispiele:
  - » Natur: Farn: jeder einzelne Zweig vom Stamm des Blatts ist nur eine kleinere Kopie des gesamten Blatts
  - » Reflexionen: die sich wiederholenden Muster einer Reflexion, wenn sich zwei glänzende Objekte gegenseitig spiegeln.
  - » Solche Beispiele überzeugen davon, dass die Natur häufig eine paradoxe Einfachheit besitzt, die wirklich elegant ist.
- > Rekursiven Algorithmen: Sie sind ebenfalls einfach und elegant und dennoch leistungsfähig.

## Rekursion (II)

- > Die Rekursion in der Informatik wird über rekursive Funktionen erreicht.
  - » Eine rekursive Funktion ist eine Funktion die sich selbst aufruft.
  - » Jeder weitere Aufruf arbeitet mit einer verfeinerten Eingabemenge, und bringt uns so die Lösung der Aufgabe näher und näher.
  - » Viele Entwickler neigen dazu, große Probleme in mehrere kleine zu zerteilen, und separate Funktionen zu deren Lösung zu entwickeln.
  - » In vielen Fällen ist es aber sehr viel eleganter eine rekursive Funktion zu verwenden.

## Rekursion: Beispiel Fakultät (I)

- > Zum Einstieg ein einfaches Problem, das normalerweise nicht mittels Rekursion gelöst würde:
- > Berechnung der Fakultät einer Zahl n
  - » Die Fakultät von n, geschrieben  $n!$  ist das Produkt aller Zahlen von n bis hinunter zu 1.
    - » z.B.:  $4! = (4)*(3)*(2)*(1) = 24$
  - > Eine Möglichkeit der Berechnung ist der iterative Ansatz (ohne Rekursion), wobei in einer Schleife alle Zahlen durchlaufen und jede Zahl mit dem Produkt aller vorherigen Zahlen multipliziert wird.
    - » formal:  $n! = (n)*(n-1)*(n-2)*.....(1)$

## Rekursion: Beispiel Fakultät (II)

- > Eine weitere Möglichkeit, dieses Problem zu lösen, besteht darin,  $n!$  als Produkt kleinerer Fakultäten zu sehen.
  - » Hierbei definiert man  $n!$  als n-mal die Fakultät von  $(n-1)$
  - » Natürlich ist die Lösung von  $(n-1)!$  das gleiche Problem wie  $n!$ , nur eben ein wenig kleiner.
    - $(n-1)! = (n-1)*(n-2)!$
    - $(n-2)! = (n-2)*(n-3)!$
    - ..... bis  $n=1$ , danach endet die Berechnung von  $n!$
- » dies ist ein rekursiver Ansatz:
  - formal:

$$F(n) = \begin{cases} 1 & \text{wenn } n = 0, n = 1 \\ n * F(n - 1) & \text{wenn } n > 1 \end{cases}$$

### Rekursion: Beispiel Fakultät (III)

> Berechnung von  $4!$  mit Hilfe des oben beschriebenen rekursiven Ansatzes:

$$\begin{aligned}
 F(4) &= 4 * F(3) \\
 F(3) &= 3 * F(2) && \text{Abstiegsphase} \\
 F(2) &= 2 * F(1) \\
 && F(1) = 1 & \text{Abbruchbedingung} \\
 && F(2) = 2 * 1 \\
 F(3) &= 3 * 2 && \text{Aufstiegsphase} \\
 F(4) &= 4 * 6 \\
 24 &&& \text{Abschluss der Rekursion}
 \end{aligned}$$

### Rekursion: Beispiel Fakultät (IV)

- > Grundlegende Phasen eines Rekursionsprozesses:
  - » Abstiegsphase (winding)
  - » Aufstiegsphase (unwinding)
- > In der Abstiegsphase wird die Rekursion fortgesetzt, indem sich die Funktion rekursiv selbst aufruft.
- > Eine Abbruchbedingung definiert den Zustand, bei dem sich eine rekursive Funktion nicht erneut aufruft, sondern zurückkehrt.
  - » Im Beispiel: Abbruchbedingung ( $n=1$ ) und ( $n=0$ )
- > Jede Rekursion muss zumindest eine Abbruchbedingung haben, da die Abstiegsphase sonst nie abgeschlossen ist.
- > Sobald Abstiegsphase abgeschlossen – Prozess tritt in die so genannte Aufstiegsphase ein.

## Rekursion: Beispiel Fakultät (V)

> zugehörige C-Funktion `fact`, die eine Zahl `n` erwartet und rekursiv deren Fakultät berechnet:

```
int fact(int n) {  
    if (n<0)  
        return 0;  
    else if (n==0)  
        return 1;  
    else if (n==1)  
        return 1;  
    else  
        return n * fact(n-1);  
}
```

## Rekursionen in C/C++ (I)

- > Um zu verstehen wie Rekursion wirklich funktioniert ist es hilfreich sich anzusehen auf welche Weise Funktionen in C ausgeführt werden.
- > Organisation eines C-Programms im Speicher:
  - » Besteht grundsätzlich aus 4 Bereichen
    - **Codebereich** (enthält die Maschinenbefehle, die während des Programmlaufs ausgeführt werden)
    - **Bereich für statische Daten** (besteht aus Daten die während des gesamten Lebenszyklus eines Programms erhalten bleiben; z.B.: globale Variablen)
    - **Heap** (enthält dynamisch allozierten Speicher wie er bspw. von malloc alloziert wird; dynamische Speicherreservierung und Nutzung)
    - **Stack** (enthält Informationen über Funktionsaufrufe)

## Rekursionen in C/C++ (II)

- > Wird in einem C-Programm eine Funktion aufgerufen, wird auf dem Stack ein Speicherbereich alloziert, um die mit diesem Aufruf verknüpften Informationen festzuhalten.
- > Jeder Aufruf wird als Activation (Aktivierung) bezeichnet.
- > Der auf dem Stack abgelegte Speicherblock wird als Activation-Record bzw. Stack-Frame bezeichnet
  - » besteht aus 5 Bereichen
    - *eingehende Parameter* (sind Parameter die an die Activation übergeben werden)
    - Platz für den *Rückgabewert*
    - zur Evaluierung von Ausdrücken verwendeter *temporärer Speicher*
    - bei der Beendigung einer Activation verwendete, *gesicherte Statusinformationen*
    - *ausgehende Parameter* (sind Parameter die an Funktionen übergeben werden, die innerhalb dieser Activation aufgerufen werden → werden zu eingehenden Parametern in der neuen Activation)
- > Activation Record für einen Funktionsaufruf bleibt bis zum Funktionsende am Stack.

## Rekursionen in C/C++ (III)

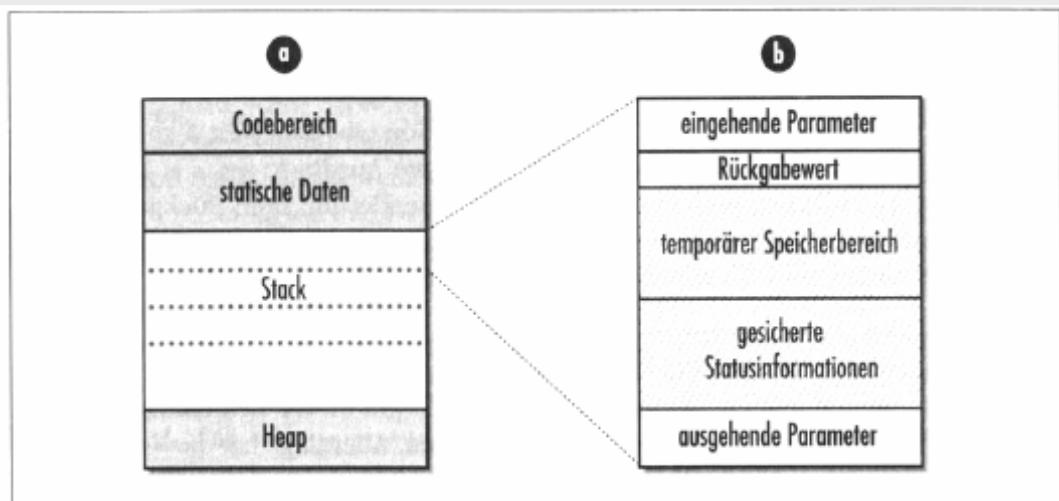


Abbildung : Die Speicher-Organisation eines (a) C-Programms und (b) eines Activation Records

## Rekursionen in C/C++ (IV)

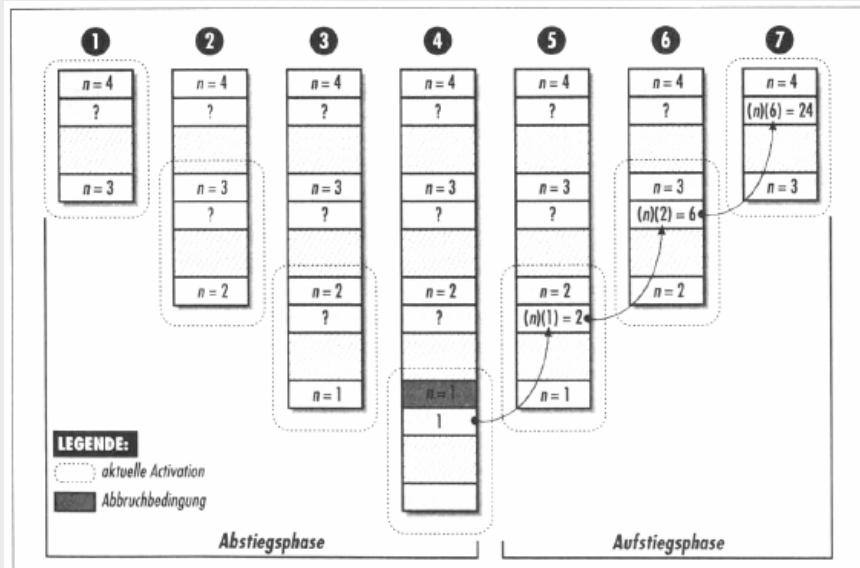


Abbildung : Der Stack eines C-Programms während der rekursiven Berechnung von  $4!$

Erstellt von: Jürgen Falb

10.02.2017

Seite 45

## Endrekursion (I)

- > Eine Funktion ist endrekursiv, wenn alle in ihr enthaltenen Aufrufe endrekursiv sind.
- > Ein rekursiver Aufruf ist endrekursiv, wenn er die zuletzt ausgeführte Anweisung innerhalb der Funktion bildet, und der Rückgabewert kein Teil eines Ausdrucks ist.
- > Endrekursive Funktionen haben während der Aufstiegsphase nichts zu tun. Diese Charakteristik ist wichtig, da die meisten modernen Compiler automatisch Code generieren, der diese Tatsache zu seinem Vorteil nützt.

## Endrekursion (II)

- > Erkennt ein Compiler einen Aufruf als endrekursiv, überschreibt er das aktuelle Activation-Record, statt ein neues auf dem Stack abzulegen.
  - » Der Compiler kann dies tun, da der rekursive Aufruf die letzte Anweisung innerhalb der aktuellen Activation ist, und es somit bei der Rückkehr des Aufrufs innerhalb der Activation nichts mehr zu tun gibt.
  - » D.h.: es gibt keinen Grund, die aktuelle Activation beizubehalten, da die darin enthaltenen Daten nicht mehr gebraucht werden → Stack wird nicht mehr so intensiv benutzt → Performance-Verbesserungen
- > Rekursionen sollten möglichst endrekursiv sein.

## Endrekursion: Beispiel Fakultät (I)

- > Warum war die vorangegangene Berechnung der Fakultät nicht endrekursiv?
  - » Die ursprüngliche Definition hat  $n!$  berechnet, indem  $n$  in jeder Activation mit  $(n-1)!$  multipliziert wurde.
  - » Dieser Vorgang wurde von  $n=(n-1)$  bis  $n=1$  wiederholt
  - » Nicht endrekursiv, weil der Rückgabewert jeder Activation davon abhing, dass  $n$ -mal der Rückgabewert der nachfolgenden Activation multipliziert wurde.
  - » Aus diesem Grund musste der Activation-Record jedes Aufrufs auf dem Stack verbleiben, bis die Rückgabewerte der nachfolgenden Aufrufe ermittelt waren.

## Endrekursion: Beispiel Fakultät (II)

> Betrachtet man nun eine endrekursive Definition von  $n!$ , die formal wie folgt aussieht:

$$F(n, a) = \begin{cases} a & \text{wenn } n = 0, n = 1 \\ F(n - 1, na) & \text{wenn } n > 1 \end{cases}$$

- » Diese Definition ist der vorigen sehr ähnlich, nur dass hier ein zweiter Parameter  $a$  (zu Beginn auf 1 gesetzt) verwendet wird, der den Wert der Fakultät enthält, der innerhalb des rekursiven Prozesses bislang errechnet wurde.
- » Damit verhindert man, dass der Rückgabewert jeder Activation mit  $n$  multipliziert werden muss.
- » Stattdessen weist man in jedem rekursiven Aufruf  $a=n*a$  und  $n=n-1$  zu.
- » Das wird wiederholt, bis  $n=1$  ist, was der Abbruchbedingung entspricht; danach wird einfach  $a$  zurückgegeben

## Endrekursion: Beispiel Fakultät (III)

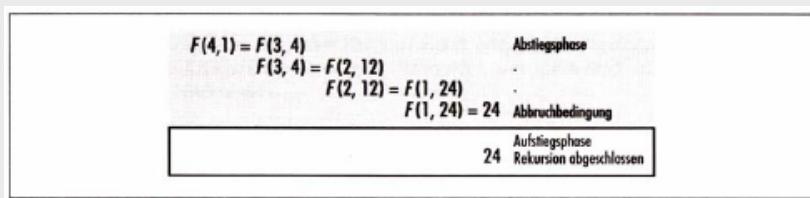


Abbildung 3-4: Endrekursive Berechnung von 4!

```

int fact_tail(int n, int a) {
    if (n<0)
        return 0;
    else if (n==0)
        return 1;
    else if (n==1)
        return a;
    else
        return fact_tail(n-1,n*a);
}
  
```

## Endrekursion: Beispiel Fakultät (IV)

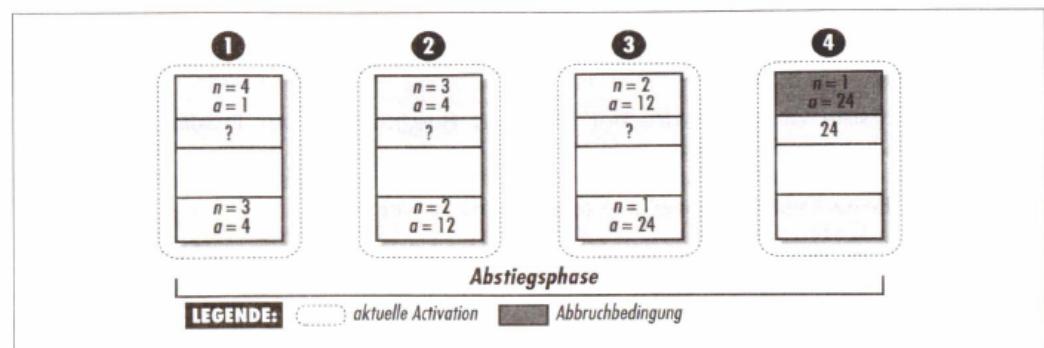


Abbildung 3-5: Der Stack eines C-Programms während einer endrekursiven Berechnung von  $4!$

## Rekursion: Beispiel 2

- > Rekursive Funktionen bieten häufig kurze und knappe Beschreibungen nützlicher Berechnungen.
- > Beschreiben Sie, welche Berechnung die nachfolgende rekursive Definition beschreibt:

$$H(n) = \begin{cases} 1 & \text{wenn } n = 1 \\ H(n - 1) + (1/n) & \text{wenn } n > 1 \end{cases}$$

### Rekursion: Beispiel 3

> Ist die gegebene Funktion endrekursiv? Wenn ja, dann beschreiben Sie warum, wenn nicht, warum nicht, und stellen Sie eine endrekursive Funktion vor.

$$H(n) = \begin{cases} 1 & \text{wenn } n = 1 \\ H(n - 1) + (1/n) & \text{wenn } n > 1 \end{cases}$$

### Analyse von Algorithmen

## Analyse von Algorithmen (I)

- > Die Analyse von Algorithmen ist sehr eng mit dem Algorithmenentwurf verbunden.
- > Dabei geht es darum festzustellen, wie gut bzw. wie **effizient** ein vorliegender Algorithmus ist.
- > Es gibt verschiedene Maße für die Effizienz von Algorithmen:
  - » **Speicherplatz**
  - » **Laufzeit**
  - » Besondere **charakterisierende Parameter** (problemabhängig), das sind z.B. beim Sortieren von Daten:
    - die Anzahl der Vergleichsoperationen
    - die Anzahl der Bewegungen von Datensätzen

## Analyse von Algorithmen (II)

- > *Meist ist die Laufzeit eines Algorithmus der ausschlaggebende Faktor.*
- > **1. Möglichkeit:** Messen der absoluten Laufzeit (z.B. in ms):
  - » Dazu muss die Maschine festgelegt werden, auf der gerechnet wird, da der Rechner einen Einfluss auf die absolute Laufzeit hat.
  - » Betrachtungen für eine RAM (random access machine):
    - 1 Prozessor
    - Alle Daten liegen im Hauptspeicher.
    - Alle Speicherzugriffe dauern gleich lange.
  - » Ebenso hat der Compiler einen Einfluss auf die Laufzeit, je nachdem welche Optimierungen er bei der Programmerstellung vornimmt.
- > In der Praxis interessiert die absolute Laufzeit nicht, sondern meist geht es um die Relation von mehreren Algorithmen zueinander.

## Analyse von Algorithmen (III)

- > **2. Möglichkeit:** Zählen der primitiven Operationen
  - » Wird nichts genauer spezifiziert, dann zählt man die **primitiven Operationen** eines Algorithmus als jeweils **eine Zeiteinheit**.
  - » unter primitiven Operationen versteht man:
    - Zuweisungen:  $a = b$
    - Arithmetische Befehle:  $a = b \text{ o } c$ ; mit  $\text{o} \in \{+, -, *, /, \dots\}$
    - Logische Operationen:  $\wedge, \vee, \neg$
    - Sprungbefehle: „gehe zu label“, falls  $(a <> b) \dots$  sonst ..., mit  $<> \in \{<, \leq, ==, !=, \geq, >\}$
  - » Vernachlässigt werden also:
    - Indexrechnungen
    - Typ der Operation
    - Länge der Operanden

## Analyse von Algorithmen (IV)

- > Laufzeit eines Algorithmus = die Anzahl der von diesem Algorithmus ausgeführten primitiven Operationen.
- > Klarerweise dauert das Sortieren von 1000 Zahlen länger als das Sortieren von 10 Zahlen.
- > → Beschreibung der Laufzeit als **Funktion der Eingabegröße**:
  - » Beim Sortieren einer  $n$ -elementigen Folge ist diese Eingabegröße  $n$
  - » Das Sortieren von „fast schon sortierten“ Folgen geht schneller als das Sortieren von komplett unsortierten Folgen.
  - » → Es sollte auch beim Sortieren von gleich langen Eingabegrößen eine **Unterscheidung** gemacht werden.

## Analyse von Algorithmen (V)

> Laufzeitfunktion:

- » Es wird jeder Zeile des Algorithmus eine Laufzeit  $c_i$  zugewiesen.
- » Es wird bestimmt wie oft der Befehl in der Zeile ausgeführt wird.
- »  $T(n)$  wird als Laufzeitfunktion in  $n$  bezeichnet.

Zeile	Kosten	Wie oft?
2	$c_2$	$n$
3	$c_3$	$n - 1$
5	$c_5$	$n - 1$

$$T(n) = c_2 * n + c_3 * (n - 1) + c_5 * (n - 1)$$

## Asymptotisches Laufzeitverhalten

> Kleine Problem sind meist uninteressant:

z.B. Sortieren von  $n=5$  Elementen.

» Hier ist die Laufzeit hauptsächlich abhängig von anderen Faktoren, wie z.B. der Dauer zum Starten des Programms.

» Für ein kleines "n" ist somit meistens die Wahl des Algorithmus nicht entscheidend

> Interessanter: Wie verhält sich der Algorithmus/Programm, wenn die Anzahl der Elemente sehr groß wird?

» z.B. wenn die Problemgröße "n" um das Tausendfache steigt

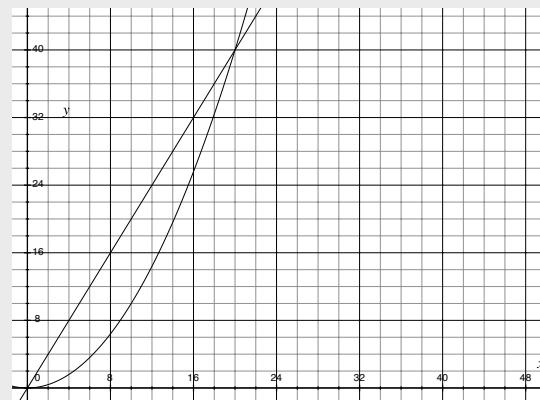
> → **Asymptotisches Laufzeitverhalten** ( $n$  geht gegen Unendlich)

## Größenordnung der Laufzeitfunktion

- > Die ausführliche Laufzeitfunktion ist oft zu umfangreich.
- > D.h. es wird eine obere Schranke  $O(T(n))$  gesucht, die für alle  $n$  garantiert größer ist als  $T(n)$ .
- > → Es wird  $T(n)$  meist vereinfacht, indem man nur die **Ordnung** der Laufzeitfunktion betrachtet.
- > Beispiele:
  - » Ordnung der Funktion:  $T(n)=an^2+bn+c$  ist  $n^2$
  - » Ordnung der Funktion:  $T(n)=an+b$  ist  $n$
  - » man schreibt:
  - »  $an^2+bn+c \Rightarrow O(n^2)$
  - »  $an+b \Rightarrow O(n)$

## Definition der **O**-Notation

- > Es seien  $f : N \rightarrow N$  und  $s : N \rightarrow N$  zwei Funktionen ( $s$  wie Schranke), die Zahlen von  $N$  wieder auf  $N$  abbilden.
- > **Die Funktion  $f$  ist von der Größenordnung  $O(s)$ ,**  
geschrieben  $f \in O(s)$ ,  
wenn es ein  $k \in N$  und ein  
 $m \in N$  gibt,  
so dass gilt:  
**Für alle  $n \in N$  mit  $n \geq m$**   
**ist  $f(n) \leq k * s(n)$ .**



## Rechnen mit der O-Notation

- > Elimination von Konstanten:
  - »  $2 \cdot n \in O(n)$
  - »  $n/2 + 1 \in O(n)$
- > Bei einer Summe zählt nur der am stärksten wachsende Summand (mit dem höchsten Exponenten):
  - »  $2n^3 + 5n^2 + 10n + 20 \in O(n^3)$
  - »  $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq \dots \subseteq O(2^n)$
- > **Beachte:**  $1000 n^2$  ist nach diesem Leistungsmaß immer „besser“ als  $0,001 n^3$ , auch wenn das  $m$ , ab dem die  $O(n^2)$ -Funktion unter der  $O(n^3)$ -Funktion verläuft, sehr groß ist ( $m = 10^6$ )

## Wichtige Klassen von Funktionen / Algorithmen

$O(1)$	konstante Funktionen
$O(\log n)$	logarithmische Funktionen (z.B. Suchen)
$O(n)$	lineare Funktionen (z.B. Abarbeiten von Mengen)
$O(n \log n)$	(typ. Sortierverfahren)
$O(n^2)$	quadratische Funktionen (z.B. primitive Sortierverfahren)
$O(n^k)$	polynomielle Funktionen
$O(2^n)$	exponentielle Funktionen (Ausprobieren)

## Analysearten

- > „Best-Case“-Analyse
  - » Misst die kürzeste mögliche Laufzeit über alle möglichen Eingabe-Instanzen.
- > „Worst-Case“-Analyse
  - » Misst die längste Laufzeit über alle möglichen Instanzen bei vorgegebener Eingabegrösse.
  - » → die Worst-Case Analyse ist eine obere Schranke für die Laufzeit einer beliebigen Instanz dieser Grösse
- > „Average-Case“-Analyse
  - » Misst die durchschnittliche Laufzeit über alle Instanzen der Grösse n (einer gegebenen Instanzklasse)
  - » Kann manchmal erheblich besser, aber auch manchmal genauso schlecht wie die Worst-Case Analyse sein.

## Klassen von Algorithmen

## Klassen von Algorithmen / Techniken zur Programmentwicklung

- > Zufallsbasierte (randomisierte) Algorithmen
- > „Gierige“ Algorithmen (Greedy-Algorithmen)
- > Teile-und-Herrsche-Algorithmen (Divide-and-Conquer Algorithmen)
- > Exhaustionsalgorithmen (Erschöpfende Alg. - Backtracking)
- > Dynamische Programmierung
- > Näherungsalgorithmen

## Zufallsbasierte (randomisierte) Algorithmen

- > Vertrauen auf die statistischen Eigenschaften von Zufallszahlen (z.B.: Quicksort).
- > Normalverteilung von Zufallszahlen wird ausgenutzt.
- > Ausgeglichene Partitionierung bei der Problemlösung
- > Eine Zufallszahl bestimmt den Ablauf des Algorithmus

## Greedy Algorithmen

- > Treffen Entscheidungen die im Augenblick am besten erscheinen, d.h. versuchen in jedem Teilschritt „so viel als möglich“ zu erreichen
- > Es werden Entscheidungen getroffen, die lokal gesehen optimal sind, in der Hoffnung das sie zu globalen optimalen Lösungen führen
- > allerdings führt eine lokal optimale Lösung nicht immer zu einer global optimalen Lösung
- > Beispiel: Geldrückgabe mit *cent* Münzen

$$78 = 50 + 10 + 10 + 5 + 2 + 1$$

bei anderer Aufteilung der Münzen (11,5,1)

$$15 = 11 + 1 + 1 + 1 + 1, \text{ anstatt: } 5 + 5 + 5$$

## Divide-and-Conquer Algorithmen

- > Kreisen um drei Schritte:
  - » *Teilen* (Unterteilung der Daten in kleinere, einfacher zuhandhabende Stücke)
  - » *Herrschern* (Verarbeiten jedes Stücks, indem eine Operation an ihm vorgenommen wird)
  - » *Kombinieren* (Neuzusammenstellung der verarbeitenden Stücke)
- > Algorithmus wird rekursiv auf die kleineren Teilaufgaben angewandt.
- > Abbruchbedingung wenn Teilproblem explizit lösbar ist.
- > Beispiel: Merge-Sort

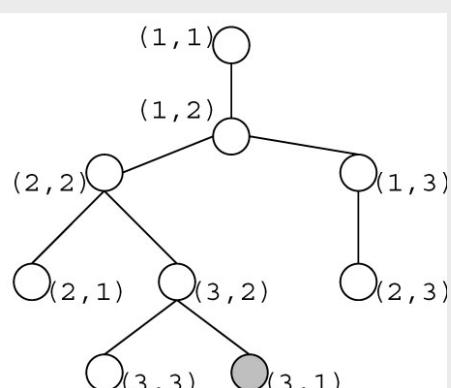
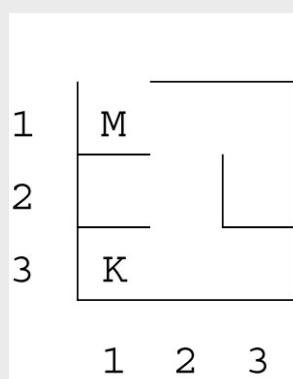
## Exhaustionsalgorithmen - Erschöpfende Algorithmen

- > sind Algorithmen, die alle Möglichkeiten ausprobieren um eine Lösung zu finden.
- > d.h. sie sind eine systematische Suchtechnik um den kompletten Lösungsraum abzusuchen.
- > Voraussetzung: Die Anzahl der möglichen Aktionen/Operationen in jedem Schritt muss beschränkt sein.
- > Beispiele:
  - » Backtracking
  - » Brute-Force / Exhaustive Search

## Backtracking

- > Systematische Suchtechnik um einen Lösungsraum komplett abzusuchen

Labyrinth



## Backtracking: 8 Damen Problem (I)

> Gesucht sind alle Konfigurationen auf einem Schachbrett, sodass keine Dame eine andere bedroht.

	1	2	3	4	5	6	7	8
1				D				
2							D	
3								D
4		D						
5							D	
6	D							
7			D					
8					D			

## 8 Damen Problem (II)

```

Algorithmus: Platziere(i)

for h = 1..8 do
    if feld in Zeile i, Spalte h nicht bedroht
    then
        Setze Dame auf i, h;
        if Brett voll /* i = 8 */
        then
            Gib Konfiguration aus;
        else
            Platziere(i+1);
        fi
        Entferne Dame von i, h;
    fi
od

```

## Dynamische Programmierung (I)

- > Ist der Teile-und-Herrsche-Methode ähnlich.
  - » Auch hier werden Probleme gelöst, indem größere Probleme in kleinere Unterprobleme aufgeteilt werden
- > Unterschied besteht darin, in welcher Beziehung die Unterprobleme zueinander stehen.
  - » Bei Teile-und-Herrsche ist jedes Unterproblem **unabhängig** vom anderen (→ Lösung durch Rekursion)
  - » Bei Lösungen mittels dynamischer Programmierung sind die Unterprobleme **voneinander abhängig** → Unterprobleme können andere Unterprobleme gemeinsam haben.

## Dynamische Programmierung (II)

- > Vereint Aspekte von Greedy, Divide-and-Conquer und Backtracking
  - » Greedy: Wahl optimaler Teillösungen
  - » D&C, Backtracking: Rekursiver Ansatz
  - » Mehrfach auftretende Teilprobleme werden nur **einmal** gelöst.

## Dynamische Programmierung (III)

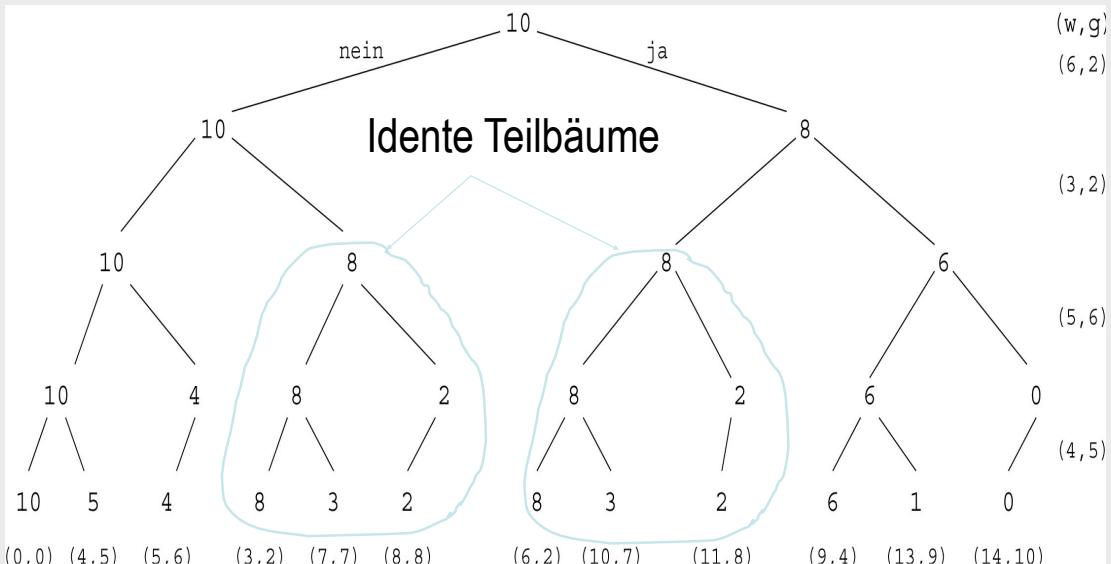
Beispiel: Rucksackproblem

- > Gegebene Kapazität C
- > Gegenstände haben Gewicht g und Wert w
- > Rucksack ist so zu füllen, dass:
  - » Die Kapazität (Gewicht) nicht überschritten wird
  - » Wert der Gegenstände maximal wird

## Rucksackproblem (I)

- > Für n Gegenstände gibt es  $2^n$  Möglichkeiten den Rucksack zu packen
- > Lösung mittels Backtracking zu aufwendig für grössere n
- > Beispiel: 4 Gegenstände:
  - » Gewichte:  $g_1=2, g_2=2, g_3=6, g_4=5$
  - » Werte:  $w_1=6, w_2=3, w_3=5, w_4=4,$

## Rucksackproblem (II)



## Rucksackproblem (III)

> Backtracking Version ohne dynamischer Programmierung:

```

procedure rucksack(i, Restkapazität)
if i = AnzahlObjekte
    if Restkapazität < gi then
        return 0;
    else
        return wi;
    fi
else
    if Restkapazität < gi then
        return rucksack(i+1, Restkapazität);
    else return max( rucksack(i+1, Restkapazität),
                    rucksack(i+1, Restkapazität - gi) + wi);
    fi
fi

```

## Rucksackproblem (IV)

$$g_1=2, g_2=2, g_3=6, g_4=5, g_5=4$$

$$w_1=6, w_2=3, w_3=5, w_4=4, w_5=6$$

(zusätzlich 5. Gegenstand zur besseren Illustration)

i \ rc	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6

4	0	0	0	0	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	10	11
1	0	0	6	6	9	9	12	12	15	15

## Rucksackproblem (V)

```

algorithm rucksack(n, C)           f(i,rc) ... Tabelleneintrag für
    for rc = 0 to C do             Gegenstand i und
        if rc < gi then f(n,rc) = 0   Restkapazität rc
        else                      f(n,rc) = wi
        fi
    od
    for i = n-1 downto 2 do
        for rc = 0 to C do
            if rc < gi then f(i,rc) = f(i+1,rc);
            else                  f(i,rc) = max(f(i+1,rc), f(i+1,rc-gi)+wi);
            fi
        od
        if C < gi then return f(2,C);
        else                  return max(f(2,C), f(2,C-gi)+wi)
        fi

```

## Näherungsalgorithmen

- > Sind Algorithmen die keine genauen Lösungen berechnen.
- > Ermitteln vielmehr Lösungen die „gut genug“ sind.
- > Häufig zur Lösung von Problemen eingesetzt, die rechentechnisch sehr „teuer“ sind, aber gleichzeitig von zu großer Bedeutung sind, um sie aufzugeben.
- > Beispiel: Travelling-Salesman-Problem (kürzestmögliche Route für einen Handelsvertreter zwischen mehreren Städten finden)

## Lineare / Verkettete Listen

## Abstrakte Datentypen - Wiederholung

- > Ein abstrakter Datentyp (ADT) besteht aus einer oder mehreren **Mengen von Objekten** und darauf definierten **Operationen**.
- > Bei der Definition von ADT's wird keine Rücksicht auf programmtechnische Realisierungen genommen.

## ADT Lineare Liste (I)

- > **Objekte**: Menge aller endlichen Folgen eines gegebenen Grundtyps.
- > Die Schreibweise für eine lineare Liste ist:  
 $L = \langle a_1, a_2, \dots, a_n \rangle$ , wobei  $L = \langle \rangle$  eine leere Liste ist.
- > **Operationen**: Im folgenden gilt:
  - »  $x$  ist der Grundtyp
  - »  $p$  eine Position
  - »  $L$  eine lineare Liste

## ADT Lineare Liste (II)

### > *Füge\_ein (x,p,L)*

»  $L \neq <>$ ,  $1 \leq p \leq n$  ( $L$  ist nicht die leere Liste):

– Vorher:  $\langle a_1, a_2, \dots, a_p, a_{p+1}, \dots, a_n \rangle$

– Nachher:  $\langle a_1, a_2, \dots, a_{p-1}, x, a_p, a_{p+1}, \dots, a_n \rangle$

»  $L = <>$ ,  $p=1$  ( $L$  ist die leere Liste):

– Vorher:  $<>$

– Nachher:  $\langle x \rangle$

» Sonst undefiniert

## ADT Lineare Liste (III)

### > *Entferne (p,L)*

»  $L \neq <>$ ,  $1 \leq p \leq n$

– Vorher:  $\langle a_1, a_2, \dots, a_{p-1}, a_p, a_{p+1}, \dots, a_n \rangle$

– Nachher:  $\langle a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n \rangle$

» Sonst undefiniert

### > *Suche (x,L)*

» liefert die erste Position  $p$  in  $L = \langle a_1, a_2, \dots, a_n \rangle$  mit  $a_p = x$ , falls diese existiert, sonst „nicht gefunden“.

## ADT Lineare Liste (IV)

### > Zugriff ( $p, L$ )

» liefert  $a_p$  in  $L = \langle a_1, a_2, \dots, a_n \rangle$  mit  $a_p = x$ , falls  $1 \leq p \leq n$ ,

» sonst undefiniert

### > Verkette ( $L_1, L_2$ )

» kann auch zum Anhängen einzelner Elemente verwendet werden

» Liefert für  $L_1 = \langle a_1, a_2, \dots, a_n \rangle$  und  $L_2 = \langle b_1, b_2, \dots, b_m \rangle$  die Liste  
 $L = \langle a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m \rangle$

## ADT Lineare Liste - Implementierung

> Sequentiell gespeicherte lineare Listen

> Speicherung in einem Feld (Array)

»  $L[1], L[2], \dots, L[n_{\max}]$

»  $n_{\max}$  muß gross genug sein, damit alle eventuellen Elemente Platz haben.

## Verkettete Liste

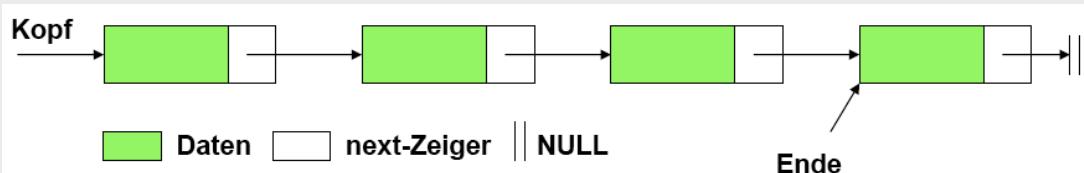
- > Verkettete Listen bestehen aus einer Reihe von Elementen, die in einer bestimmten Reihenfolge angeordnet sind.
- > Sind zur Verwaltung von Datenmengen geeignet
- > Form der Verwaltung erinnert stark an Arrays, allerdings bieten verkettete Listen wichtige Vorteile
  - » Wesentlich effizienter bei Einfüge- und Lösch-Operationen
  - » Nutzen dynamisch allozierten Speicher (d.h.: erst zur Laufzeit reservierten Speicher, weil bei vielen Anwendungen die Größe der Daten während zur Compile-Zeit nicht bekannt ist.)

## Einfach Verkettete Liste (I)

- > Ist die einfachste Form verketteter Listen.
- > Die Elemente sind über einen einzelnen Zeiger verknüpft.
- > Erlaubt das Durchgehen der Liste vom ersten bis zum letzten Element (in Vorwärtsrichtung).
- > Jedes Element besteht aus zwei Teilen:
  - » *Datenelement*
  - » *Zeiger* (= next-Zeiger)
    - zeigt auf das nächste Element.
    - zeigt auf NULL (bzw. NIL), wenn es kein nächstes Element gibt.
- > Das erste Element ist der *Kopf (Head)*, das letzte Element ist das *Ende (Tail)*.

## Einfach Verkettete Liste (II)

- > Um auf ein Element in der verketteten Liste zuzugreifen, wird beim Kopf angefangen.
- > Mittels next-Zeiger kommt man zum jeweils nächsten Element.
- > Die Elemente werden dynamisch im Speicher alloziert (malloc in C oder new in C++).
- > Ein Element kann nur über den Zeiger gefunden werden.
- > Wird irrtümlich ein Zeiger gelöscht, sind alle nachfolgenden Elemente verloren!!

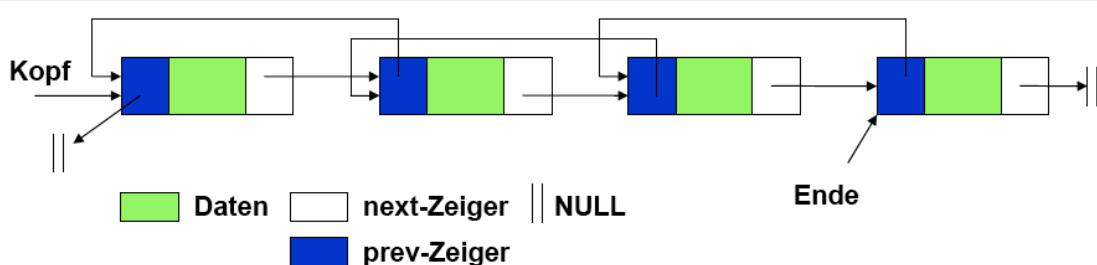


## Doppelt Verkettete Liste (I)

- > Doppelt verkettete Listen bestehen aus:
  - » *Datenelement*
  - » *next-Zeiger* (zeigt auf das nächste Element)
  - » *prev-Zeiger* (zeigt auf das vorherige Element)
- > Um Anfang und Ende zu markieren, zeigen der prev-Zeiger des ersten Elements und der next-Zeiger des letzten Elements auf NULL.

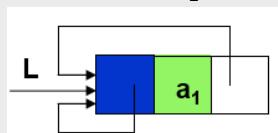
## Doppelt Verkettete Liste (II)

- > Um eine doppelt verkettete Liste **vorwärts** – vom Anfang bis zum Ende – zu durchlaufen, wird der **next-Zeiger** benutzt.
- > Um eine doppelt verkettete Liste **rückwärts** – vom Ende bis zum Anfang – zu durchlaufen, wird der **prev-Zeiger** benutzt.
- > ⇒ Bessere Beweglichkeit innerhalb der Liste.

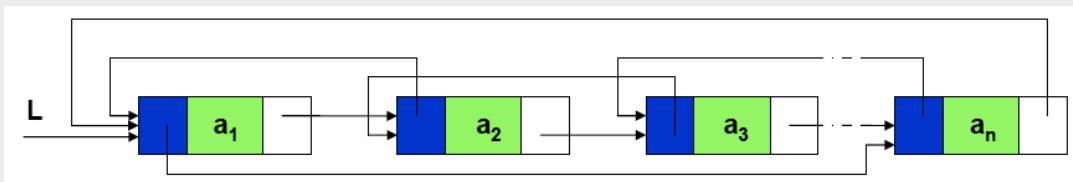


## Ringförmig Verkettete Liste (I)

- > Leere Liste:
  - » Es gibt kein Listenelement.
  - » Dargestellt durch  $L=NULL$  und  $n=0$ .
- > Eine Liste  $L$  mit einem Element  $L=<a_1>$ :



- > Allgemeine Liste  $L = <a_1, a_2, a_3, \dots, a_n>$ :



## Ringförmig Verkettete Liste (II)

> *Füge\_ein(x,p,L):*

```

1: falls p < 1 v p > max(1,n) dann {
2:   STOP: Fehler;
3: }
4: h = L;
5: für i=1,...,p-1 {
6:   h = h.Nachfolger
7: }
8: neu = neues Listenelement(x);
9: falls n == 0 dann {
10:   neu.Nachfolger = neu.Vorgänger = neu;
11: } sonst {
12:   neu.Nachfolger = h;
13:   neu.Vorgänger = h.Vorgänger;
14:   neu.Nachfolger.Vorgänger = neu;
15:   neu.Vorgänger.Nachfolger = neu;
16: }
17: falls p == 1 dann {
18:   L = neu;
19: }
20: n = n+1;

```

## Ringförmig Verkettete Liste (III)

> *Entferne(p,L):*

```

1: falls p<1 v p>max(1,n) dann {
2:   STOP: Fehler;
3: }
4: falls n == 1 dann {
5:   L = NULL;
6: } sonst {
7:   h = L;
8:   für i=1,...,p-1 {
9:     h = h.Nachfolger;
10:   }
11:   h.Vorgänger.Nachfolger = h.Nachfolger;
12:   h.Nachfolger.Vorgänger = h.Vorgänger;
13:   falls p == 1 dann {
14:     L = L.Nachfolger;
15:   }
16: }
17: n = n - 1;

```

## Ringförmig Verkettete Liste (IV)

> *Suche(x,L):*

```
1: h = L;
2: i = 1;
3: solange i≤n und h.Inhalt≠x {
4:     h = h.Nachfolger;
5:     i = i + 1;
6: }
7: falls i ≤ n dann {
8:     retourniere i;
9: }
10: retourniere „nicht gefunden“;
```

## Ringförmig Verkettete Liste (V)

> *Zugriff(p,L):*

```
1: falls p<1 v p>n dann {
2:     STOP; Fehler
3: }
4: h = L;
5: für i=1,...,p-1 {
6:     h = h.Nachfolger;
7: }
8: retourniere h.Inhalt
```

## Ringförmig Verkettete Liste (VI)

> *Verkette(L1,L2):*

```
1: falls L1 == NULL dann {
2:   L1 = L2;
3: } sonst falls L2 ≠ NULL dann {
4:   last = L2.Vorgänger;
5:   L1.Vorgänger.Nachfolger = L2;
6:   last.Nachfolger = L1;
7:   L2.Vorgänger = L1.Vorgänger
8:   L1.Vorgänger = last;
9: }
10: n1 = n1 + n2;
11: retourniere L1;
```

## Stacks und Queues

## Stacks und Queues

### > Stacks

- » Effiziente Datenstrukturen, mit denen Daten in einer sogenannten **LIFO-Reihenfolge** („last in, first out) abgelegt und wieder eingelesen werden.
- » Ermöglicht das erneute Einlesen von Daten in der umgekehrten Reihenfolge.

### > Queues

- » Effiziente Datenstrukturen, mit denen Daten in einer sogenannten **FIFO-Reihenfolge** („first in, first out) abgelegt und wieder eingelesen werden können.
- » Ermöglicht das erneute Einlesen der Daten in der Reihenfolge ihrer Speicherung.

## Stack (I)

- > Stacks werden im Deutschen auch manchmal Keller(-speicher) genannt.
- > Das zuletzt auf dem Stack platzierte Element wird als erstes wieder entfernt (vgl. Dose mit Tennisbällen).
- > Operationen:
  - » **push**: Ablegen von Daten am Stack
  - » **pop**: Entfernen von Daten vom Stack
  - » **peek**: Ansehen des obersten Elements, ohne es vom Stack zu entfernen.

## Stack (II)

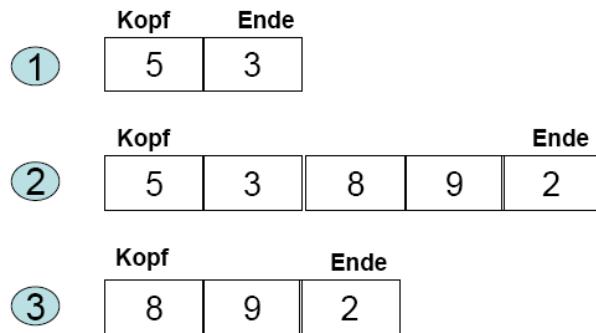
<u>Aktion</u>	<u>Stack</u>
Vor dem Start	leer
push (2)	2
push (4) push (5)	5 4 2

↓

<u>Aktion</u>	<u>Stack</u>
pop()	4 2
pop()	2
pop()	leer

## Queue (I)

- > Im Deutschen auch manchmal „Warteschlange“ genannt.
- > Das zuerst in der Queue platzierte Element wird als erstes wieder entfernt, neu hinzugefügte Elemente werden am Ende der Queue angehängt.
- > Operationen:
  - » **enqueue**: Entfernen eines Elements vom Anfang der Queue.
  - » **enqueue**: Hinzufügen eines Elements am Ende der Queue.
  - » **peek**: Ansehen des ersten Elements, ohne es aus der Queue zu.
- > entfernen

**Queue (II)**

Eine Queue (1) mit einigen Elementen;  
(2) nach der Aufnahme von 8, 9 und 2 sowie (3)  
nach dem Entfernen von 5 und 3.

**Hashtabellen**

## Hashtabellen (I)

- > Hashtabellen unterstützen eine der effizientesten Suchformen: das **Hashing**
- > Eine Hashtabelle besteht grundsätzlich aus einem Array bei dem der Datenzugriff über einen speziellen Index, den sogenannten **Schlüssel (Key)** erfolgt.
- > Die Hauptidee: über eine sogenannte Hashfunktion alle möglichen **Schlüssel auf die entsprechenden Positionen im Array abbilden**.
- > Eine **Hashfunktion** erwartet einen Schlüssel und gibt die **Hashcodierung**, auch **Hashwert** genannt zurück.

## Hashtabellen (II)

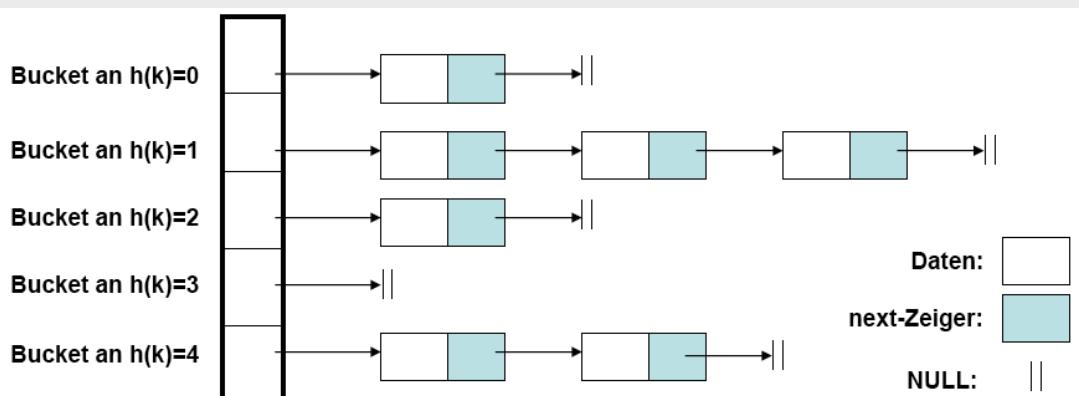
- > Während die Schlüssel unterschiedliche Typen aufweisen können, sind die Hashcodes immer Integerwerte.
- > Da sowohl die Berechnung des Hashwertes als auch die Indizierung des Arrays in **konstanter Zeit** ablaufen kann, kann die Suchfunktion ebenfalls in konstanter Zeit ausgeführt werden.
- > Kann eine Hashfunktion garantieren, dass zwei unterschiedliche Schlüssel nicht den gleichen Hashcode erzeugen, bezeichnet man die resultierende Hashtabelle als **direkt adressiert**.
- > Dies wäre der ideale Fall, allerdings ist die direkte Adressierung in der Praxis nur selten möglich.

## Hashtabellen (III)

- > Üblicherweise ist die Zahl der Einträge in einer Hashtabelle, verglichen mit der Gesamtzahl möglicher Schlüssel, recht klein.
- > Darum bilden die meisten Hashfunktionen einige Schlüssel auf die gleichen Stellen (Positionen) in der Tabelle ab.
- > Werden zwei Schlüssel auf die gleiche Position abgebildet, spricht man von einer *Kollision*.
- > Eine gute Hashfunktion minimiert solche Kollisionen, aber man muss mit Kollisionen umgehen können.

## Verkettete Hashtabelle (I)

- > Eine verkettete Hashtabelle besteht grundsätzlich aus einem Array verketteter Listen.
- > Jede Liste bildet ein so genanntes Bucket („Eimer“), in dem alle Elemente untergebracht werden, deren Hashcode einer bestimmten Position im Array entspricht.



## Verkettete Hashtabelle (II)

- > Um ein Element einzufügen, wird dessen Schlüssel an die Hashfunktion übergeben = **Hashing**
- > Ergebnis der Hashfunktion:
  - » Das Bucket, in welches das Element gehört.
  - » Das Element wird dann am Kopf der entsprechenden Liste eingefügt.
- > Um ein Element einzulesen oder zu löschen, führt man erneut ein Hashing des Schlüssels durch, um das Bucket zu bestimmen. Dann wird die entsprechende Liste durchgegangen bis das gesuchte Element gefunden wird
- > Weil jedes Bucket eine verkettete Liste darstellt, ist eine verkettete Hashtabelle nicht auf eine feste Zahl von Elementen beschränkt (Die Performance sinkt aber bei größeren Listen).

## Verkettete Hashtabelle (III)

- > Verweisen zwei Schlüssel auf die gleiche Position in einer Hashtabelle, dann liegt eine Kollision vor.
- > Verkettete Hashtabellen verwenden eine einfache Methode zur Auflösung von Kollisionen:
  - » Elemente werden einfach in dem Bucket abgelegt, in dem die Kollision auftritt.
  - » Problem dabei: Der Zugriff auf diese Elemente dauert daher länger (abhängig von der Listenlänge!).
  - » Alle Buckets sollten gleichmäßig wachsen (gleiche Grösse) !
- > Elemente so **gleichmäßig** und zufällig wie möglich über die Tabelle **verteilen**. Dieser perfekte Fall wird **gleichförmiges Hashing** genannt.

## Hashfunktion (I)

- > Wahl der Hashfunktion:
  - » Ziel: Annäherung an ein gleichförmiges Hashing.
  - » Eine *Hashfunktion h* ist eine definierte Funktion, die einen Schlüssel  $k$  auf irgendeine Position  $x$  der Hashtabelle abbildet.  $x$  wird dabei als *Hashcode* von  $k$  bezeichnet.
  - » Formal:  $x = h(k)$
  - » Generell erwarten die meisten Hashmethoden, dass  $k$  ein Integerwert ist, der mathematisch einfach verarbeitet werden kann, um es  $h$  zu ermöglichen, die Elemente in der Tabelle gleichmäßiger zu verteilen.

## Hashfunktion (II)

- » Wie eine bestimmte Menge von Schlüsseln genau umgewandelt wird, hängt größtenteils von den Eigenschaften der Schlüssel ab.
- » Eine schlechte Lösung wäre die Anfangszeichen oder Endezeichen der Schlüssel zu nehmen, da diese sehr oft gleich sind (Prefix, Postfix)
- » Divisionsmethode
  - Sobald ein Schlüssel  $k$  als Integerwert vorliegt, kann eine der einfachsten Hashfunktionen verwendet werden.
  - Dabei wird der Schlüssel auf eine der  $m$  Positionen in der Tabelle abgebildet, indem man den Rest verwendet, der bei der Division von  $k$  durch  $m$  übrigbleibt.
  - Formal:  $h(k) = k \bmod m$

## Sortierverfahren

### Sortierverfahren - Allgemein

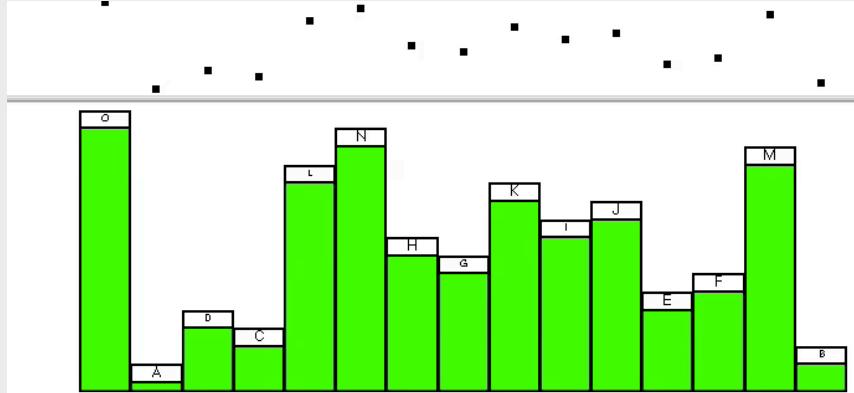
- > Mehr als ein Viertel der kommerziell benutzten Rechenzeit wird für das Sortieren verschiedenster Daten verwendet!
- > ⇒ Möglichst effiziente Sortieralgorithmen wurden deshalb entwickelt.
- > Sortierproblem formal:
  - » Gegeben: Folge von Datensätzen  $s_1, s_2, \dots, s_n$  mit den Schlüsseln  $k_1, k_2, \dots, k_n$ , auf denen eine Ordnungsrelation „ $\leq$ “ erklärt ist.
  - » Gesucht: Permutationen  $\sigma : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$  der Zahlen 1 bis  $n$ , so dass die Umordnung der Sätze gemäß  $\sigma$  die Schlüssel in aufsteigende Reihenfolge bringt:  
 $k_{\sigma(1)} \leq k_{\sigma(2)} \leq \dots \leq k_{\sigma(n)}$

## Insertion Sort (I)

- > Sortieren durch Einfügen
- > Einfach zu verwenden
- > Benötigt keinen zusätzlichen Speicher (in-place Sortierung)
- > Wird am besten zur inkrementellen Sortierung kleiner Datenmengen verwendet.

## Insertion Sort (II)

- > **Eingabe:** Ein Array A[1 .. n] von Integer-Zahlen
- > **Invariante:** Nach dem k-ten Durchlauf liegen die ersten k Zahlen in sortierter Reihenfolge vor.
- > **Vorgehen:** Im (k+1)-ten Durchlauf wird die (k+1)-te Zahl (durch Vertauschen) an die richtige Stelle eingefügt.
- > **Terminierung:** Nach dem n-ten Durchlauf.
- > **Ausgabe:** Die sortierte Folge (an Stelle der gegebenen).

**Insertion Sort (III)****Insertion Sort (IV)****Insertion-Sort(A):**

```
1: VAR key, i, j: INTEGER
2: für j = 2 ... n {
3:   key = A[j];
4:   //füge A[j] in sortierte Folge A[1],...,A[j-1] ein
5:   i = j - 1;
6:   solange i > 0 und A[i] > key {
7:     A[i+1] = A[i];
8:     i = i - 1;
9:   }
10:  A[i+1] = key;
11: }
```

## Insertion Sort (V)

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

## Insertion Sort (VI)

- > Laufzeitanalyse: Anzahl von Vergleichen und Verschiebungen
  - » Best-Case: sortierte Folge  
 $O(n)$  (besser als bei z.B. Quicksort) es sind nur n-Vergleiche notwendig.
  - » Worst-Case: umgekehrt sortierte Folge  
 Anzahl an Schiebeoperationen beträgt  
 $((n * (n + 1)) / 2) - n \Rightarrow O(n^2)$

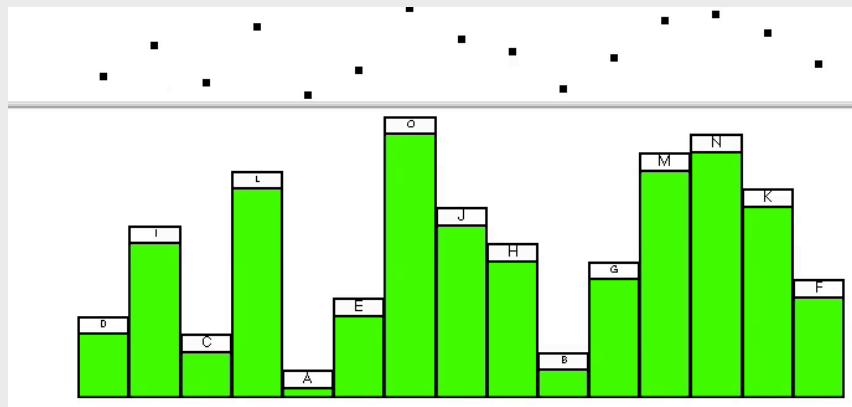
## Selection Sort (I)

- > Sortieren durch Auswahl
- > Einfacher, elementarer Algorithmus zum Sortieren
- > Genauso wie der Insertion-Sort geht der Selection-Sort davon aus, dass links der aktuellen Position  $j$  bereits alles sortiert ist.
- > Während jedoch Insertion-Sort den Schlüssel an der aktuellen Position  $j$  jeweils an die richtige Stelle in der bereits sortierten Teilfolge links von Position  $j$  einsortiert, wählt sich Selection Sort aus dem rechten, noch unsortierten Teil des Feldes ab Position  $j$  jeweils den kleinsten Schlüssel aus und verschiebt diesen an Position  $j$ .

## Selection Sort (II)

- > **Methode:** Bestimme die Position  $i_1 \in \{1, 2, \dots, n\}$ , an der das Element mit minimalem Schlüssel auftritt, vertausche  $A[1]$  mit  $A[i_1]$ ; bestimme  $i_2 \in \{2, 3, \dots, n\}$ , vertausche  $A[2]$  mit  $A[i_2]$ ; usw.

$j=1$	5	2	4	3	1
$j=2$	1	2	4	3	5
$j=3$	1	2	4	3	5
$j=4$	1	2	4	3	5
$j=5$	1	2	3	4	5
	1	2	3	4	5

**Selection Sort (III)****Selection Sort (IV)****Selection-Sort(A):**

```
1: für j = 1,2,...,n-1 {  
2:   //Bestimme Position des Minimums aus A[i],...,A[n]  
3:   minpos = j;  
4:   für i = j+1,...,n {  
5:     falls A[i].key < A[minpos].key dann {  
6:       minpos = i;  
7:     }  
8:   }  
9:   falls minpos > j dann {  
10:    Vertausche A[minpos] mit A[j];  
11:  }  
12: }
```

## Selection Sort (V)

> Laufzeitanalyse: Anzahl von Vergleichen und Verschiebungen

» Anzahl an notwendigen Vergleichen:

$$(n - 1) + (n - 2) + \dots + 3 + 2 + 1 = \frac{(n - 1) \cdot n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

»  $\Rightarrow O(n^2)$  gilt für alle Fälle

## Merge Sort (I)

> Ist einer der ältesten, für den Computer entwickelten, Sortieralgorithmen.

> Er wurde erstmals durch John von Neumann vorgestellt.

> Merge-Sort folgt einem wichtigen Entwurfsprinzip für Algorithmen, die als „Teile und Herrsche“ (divide and conquer) Algorithmen bekannt sind.

> Idee:

» **Teile:** Teile die n-elementige Folge in der Mitte in zwei Teilstücke.

» **Erobere:** Sortiere beide Teilstücke rekursiv mit Merge-Sort.

» **Kombiniere:** Verschmelze die beiden Teilstücke zu einer sortierten Gesamtfolge.

## Merge Sort (II)

- > Im folgenden gilt:
  - » A: Feld, in dem die Datensätze gespeichert sind.
  - » l,r,m: Indizes mit  $l \leq m \leq r$
  - » Merge (A,l,m,r) verschmilzt die beiden Teilarrays A[l,...,m] und A[m+1,...,r] die bereits sortiert sind. Das Resultat ist ein sortiertes Array A[l,...,r].
  - » Merge-Sort (A,l,r) sortiert A[l,...,r] (falls  $l \geq r$  gilt, so ist nichts zu tun).
  - » Das Sortieren eines n-elementigen Arrays geschieht durch Merge-Sort (A,1,n).

## Merge Sort (III)

### Merge-Sort(A,l,r):

```
1: //sortiert A[l],...,A[r]; falls l ≥ r ist nichts zu tun
2: falls l < r dann {
3:     m = [(l+r)/2];
4:     Merge-Sort(A,l,m);
5:     Merge-Sort(A,m+1,r);
6:     Merge(A,l,m,r);
7: }
```

## Merge Sort (IV)

**Merge ( $A, l, m, r$ ):**

```

1: // Input: zwei sortierte Teilfolgen A[1]...A[m] und A[m+1]...A[r]
2: // Output: verschmolzene, sortierte Teilfolge in A[1]...A[r]
3: B[1,...,m] = A[1,...,m];
4: B[m+1,...,r] = A[r,...,m+1]; // Daten aus Feld A werden umgekehrt
5: p = 1;                      // ins Feld B kopiert
6: q = r;
7: für i=1,...,r {
8:   falls B[p] ≤ B[q] dann {
9:     A[i] = B[p];
10:    p = p + 1;
11:  } sonst {
12:    A[i] = B[q];
13:    q = q - 1;
14:  }
15: }
```

## Merge Sort (V)

> Laufzeitanalyse:

- » Merge-Funktion benötigt  $2n$  Schritte ( $n$  Schritte zur Kopie ins Hilfsarray und  $n$  Schritte zum zurückkopieren)
- » Rekursive Darstellung der Laufzeit:  $T(n) \leq 2n + 2 T(n/2)$  und  $T(1) = 0$ .
- » Auflösung der Rekursion ergibt:  $T(n) \leq 2n \log(n)$
- »  $\Rightarrow O(n \log(n))$  gilt für alle Fälle

> *Ist optimal, da  $O(n \log(n))$  die untere Schranke des Sortierproblems darstellt.*

## Quick Sort (I)

- > Sehr schneller, und viel verwendeter Algorithmus in der Praxis.
- > Wurde 1960 von C.A.R. Hoare entwickelt.
- > Basiert ebenfalls auf der Strategie „Teile und Herrsche“.
- > Während beim Merge-Sort die Folgen zuerst aufgeteilt werden bevor sie sortiert werden, ist dies bei Quicksort umgekehrt.

## Quick Sort (II)

- > Idee:
  - » Wähle ein Pivotelement (z.B.: das Letzte in der Folge) und bestimme die Position, die dieses Element am Ende in der vollständigen sortierten Folge einnehmen wird.
  - » Dafür werden in einem einfachen Schleifendurchlauf diejenigen Elemente, die kleiner (bzw. kleiner gleich) als das Pivotelement sind, an die linke Seite des Feldes gebracht, und diejenigen, die größer gleich sind, an die rechte Seite.
  - » Nach diesem Schleifendurchlauf sitzt das Pivotelement an seiner richtigen Endposition, alle kleineren Elemente liegen links davon, alle größeren rechts davon - jedoch noch unsortiert.
  - » Um diese Teilstücke zu sortieren, wird die Idee rekursiv angewandt.

## Quick Sort (III)

> Methode:

» Falls A die leere Folge ist oder nur ein Element hat, so bleibt A unverändert.

» Teile:

– Wähle „Pivotelement“ k aus A. Dann teile A ohne k in zwei Teilstufen A<sub>1</sub> und A<sub>2</sub>, so dass gilt:  
- A<sub>1</sub> enthält nur Elemente  $\leq k$   
- A<sub>2</sub> enthält nur Elemente  $\geq k$

» Erobere:

– Rekursiver Aufruf: Quicksort (A<sub>1</sub>);  
– Rekursiver Aufruf: Quicksort (A<sub>2</sub>);  
– danach sind A<sub>1</sub> und A<sub>2</sub> sortiert

» Kombiniere: Bilde A durch Hintereinanderfügen in der Reihenfolge A<sub>1</sub>, k, A<sub>2</sub>

## Quick Sort (IV)

**Quicksort(A, l, r):**

```
1: //Input: Feld A[l]...A[r]
2: falls l < r dann {
3:   x = A[r].key;
4:   p = Partition(A,l,r,x);
5:   Quicksort(A,l,p-1);
6:   Quicksort(A,p+1,r);
7: }
```

> Der Aufruf lautet: Quicksort (A,1,n)

## Quick Sort (V)

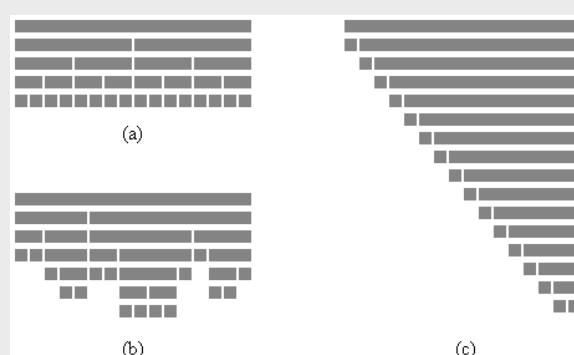
```

Partition (A,l,r,x):
1: i = l-1; j = r;
2: wiederhole
3:   wiederhole
4:     i = i + 1;
5:     bis A[i].key ≥ x ;
6:   wiederhole
7:     j = j - 1;
8:     bis A[j].key ≤ x ;
9:   falls i < j dann {
10:    vertausche A[i] und A[j];
11:  }
12: bis i ≥ j;
13: //i ist Pivotstelle: dann sind:
14: //alle links ≤ x, alle rechts ≥ x
15: vertausche A[i] und A[r]
16: retourniere i;

```

## Quick Sort (VI)

> Laufzeitanalyse:  
» Aufteilungen:



- » (a) ... günstigster Fall (Best Case)
- » (b) ... durchschnittlicher Fall (Average Case)
- » (c) ... schlechtester Fall (Worst Case)

## Quick Sort (VII)

- > Laufzeitanalyse:
  - » Average-Case:  $O(n \log(n))$ , da die Rekursionstiefe  $\log(n)$  beträgt und in jeder Schicht  $n$  Elemente zu behandeln sind.
  - » Worst-Case:  $O(n^2)$ , da die Rekursionstiefe  $n-1$  beträgt.
  - » Die Wahl des Vergleichselement bestimmt entscheidend die Aufteilung in jeder Ebene.
  - » Optimal wäre der Median-Wert als Vergleichselement, diesen zu bestimmen würde die Laufzeit erhöhen.
- > Ist in der Praxis das schnellste Sortierverfahren. (Um einen konstanten Faktor schneller als Merge-Sort und Heap-Sort)

## Lineare und Binäre Suche

## Suchverfahren

- > Suchen ist eine der grundlegendsten Operationen die mit Computern ausgeführt wird.
  - » Suchen von Datensätzen in Datenbanken.
  - » Suchen von Wörtern in Wörterbüchern (Rechtschreibprüfung).
  - » Aktuell: Suchen von Stichwörtern im WWW
- > Elementare Suchverfahren
  - » Nur Vergleichsoperationen zwischen den Schlüsseln (wie bei den Sortierverfahren).
  - » Suchen in sequentiell gespeicherten Folgen.
    - Datenelemente befinden sich in einem Feld A[1],...,A[n]
    - Gesucht wird ein Element in A mit dem Schlüssel s.

## Lineare Suche

- > Naives Verfahren, macht das was man intuitiv tun würde.
- > Idee
  - » Durchlaufe alle Elemente eines Feldes vom Beginn bis zum Ende.
  - » Vergleiche jeden Schlüssel mit dem Suchschlüssel, solange bis der Schlüssel gefunden wird.
- > Analyse
  - » Best-Case:  $O_{\min}(n)=1$
  - » Worst-Case:  $O_{\max}(n)=n$
- > einfach; eignet sich auch für einfach verkettete Listen.
- > Lineare Suche ist nur für **kleine n** empfehlenswert.

## Binäre Suche (I)

### > Idee

» Vergleiche das Suchen in einem Telefonbuch

- Wird ein Name gesucht, so schlägt man das Buch meist in der Mitte auf.
- Ist der gesuchte Name „kleiner“ als die Namen auf der aufgeschlagenen Seite, so vergisst man die rechte Hälfte des Buches und wendet das selbe Verfahren auf die linke Hälfte an, usw.
- Ist der gesuchte Name „größer“ als die Namen auf der aufgeschlagenen Seite, so vergisst man die linke Hälfte des Buches und wendet das selbe Verfahren auf die rechte Hälfte an, usw.

### > Methode

» 1.) Sortieren der Liste, damit gilt:

-  $A[1].key \leq A[2].key \leq \dots \leq A[n].key$

» 2.) „Divide and Conquer“-Methode

## Binäre Suche (II)

### > „Divide and Conquer“-Methode

- » Vergleiche den Suchschlüssel s mit dem Schlüssel des Elementes in der Mitte des Arrays.
- » s gefunden (Treffer): STOPP.
- » s ist kleiner: Durchsuche Elemente „links der Mitte“
- » s ist größer: Durchsuche Elemente „rechts der Mitte“

### > Analyse

» Best-Case:  $O_{\min}(n) = O(1)$

» Worst-Case:  $O_{\max}(n) = O(\log n)$

## Binäre Suche (III)

```
Binäre-Suche(A,s,l,r) // Aufruf: (A,s,l,n)
Input: Feld A[l..r], Schlüssel s
Output: Pos. von s oder 0, falls s nicht gefunden
1: wiederhole
2:   m = &(l+r)/2';
3:   falls s < A[m].key dann {
4:     r = m - 1;
5:   } sonst {
6:     l = m + 1;
7:   }
8: bis s == A[m].key v l > r;
9: falls s == A[m].key dann {
10:   retourniere m;
11: } sonst {
12:   retourniere 0;
13: }
```

## Bäume

## Bäume

- > Wie können folgende Daten strukturiert gespeichert werden?
  - » Familienstammbaum
  - » Spielplan eines Turniers
  - » Domain-Name Service
  - » Verzeichnisdienste (Speicherung von hierarchischen Daten, Namespaces)
- > Bestehende Hierarchie in Datenbeständen muss abgebildet werden.
- > ⇒ Speicherung in einer Baumstruktur.

## Bäume in der Informatik (I)

- > Bestehen aus einzelnen Elementen, den sogenannten *Knoten (Nodes)*.
- > Die Knoten werden *hierarchisch* angeordnet.
- > Der Knoten am Anfang der Hierarchie wird *Wurzel (Root)* bezeichnet.
- > Die Knoten direkt unter der Wurzel werden *Kinder (Children)* genannt, die selbst auch meist wieder Kinder besitzen.
- > Mit Ausnahme der Wurzel besitzt jeder Knoten genau einen *Vater-Knoten (Parent)*.

## Bäume in der Informatik (II)

- > Der Vaterknoten ist der genau darüberliegende Knoten.
- > Die Anzahl der Child-Elemente eines Parent-Elementes hängt vom Typ des Baumes ab.
- > Dieser Wert ist der *Verzweigungsgrad (branching factor)* eines Baumes.
- > Der Verzweigungsgrad bestimmt wie schnell ein Baum verzweigt, wenn weitere Knoten eingefügt werden.
- > Am wichtigsten: *Binärer Baum*
  - » Verzweigungsgrad ist 2.

## Terminologie für Bäume (I)

- > Binäre Bäume
  - » Sind Bäume, deren Knoten bis zu zwei Child-Elemente enthalten.
  - » Sind sehr weit verbreitete Baumtypen, die in einer Vielzahl von Problemen zum Einsatz kommen.
  - » Bilden oft die Basis für komplexere Baumstrukturen.
- > Methoden zum Durchlaufen von Baumstrukturen
  - » Techniken, die es ermöglichen, die Knoten eines Baumes in einer bestimmten Reihenfolge zu besuchen.
  - » Da die Knoten hierarchisch angeordnet sind, stehen dafür verschiedene Methoden zur Verfügung (Tiefensuche, Breitensuche).

## Terminologie für Bäume (II)

- > Bäume ausgleichen (Tree Balancing)
  - » Ist ein Prozess, der den Baum für eine gegebene Anzahl von Knoten so flach wie möglich hält.
  - » Ist besonderes bei Suchbäumen wichtig, weil die „Höhe“ des Baumes zu einem großen Teil die Gesamt-Performance bestimmt.
- > Binäre Suchbäume
  - » Speziell für Suchoperationen optimierte binäre Bäume.

## Terminologie für Bäume (III)

- > Rotation
  - » Methoden, mit deren Hilfe binäre Suchbäume ausgeglichen gehalten werden.
  - » Speziell AVL-Rotationen bei AVL-Bäumen, AVL Baum ist eine Form des ausgeglichenen binären Suchbaumes.
  - » Siehe AVL-Bäume später.

## Anwendungen von Bäumen (I)

- > Benutzerschnittstellen
  - » Graphische Benutzerschnittstellen und Schnittstellen für Dateisysteme.
  - » Bei graphischen Schnittstellen bilden die Fenster durch Ihre hierarchische Anordnung einen Baum.
    - Jedes Fenster/Widget, außer dem obersten besitzt ein Parent-Fenster/Widget, von dem aus es geöffnet/verwaltet wurde.
  - » Die Verzeichnisse in einem hierarchischen Dateisystem (Linux, Windows) weisen eine ähnliche Organisation auf.

## Anwendungen von Bäumen (II)

- > Datenbanksysteme
  - » Insbesondere solche, die neben einem effizienten sequentiellen und wahlfreien Zugriff auch häufig Elemente einfügen und löschen müssen.
  - » Der sogenannte B-Baum (B-Tree) ist ein ausgeglichener Suchbaum mit hohem Verzweigungsgrad und ist für solche Fälle besonders gut geeignet.
- > XML Dokumente
  - » Jedes XML Dokument stellt eine hierarchische Struktur dar und lässt sich als Baum repräsentieren.
  - » Standardisierte Darstellung mittels DOM-Trees.

## Anwendungen von Bäumen (III)

- > Verarbeitung von Ausdrücken
  - » Eine Aufgabe die häufig von Compilern und Taschenrechnern durchgeführt wird.
  - » Ein Ansatz zur Verarbeitung arithmetischer Ausdrücke ist der Ausdrucksbaum (= ein binärer Baum, in dem die Operatoren und Operanden des Ausdrucks hierarchisch angeordnet sind).

## Anwendungen von Bäumen (IV)

- > Künstliche Intelligenz (KI)
  - » Logik-basierte Spiele wie bspw. Schach, Packprobleme, ...
  - » Viele KI-Probleme werden mit Hilfe von Entscheidungsbäumen gelöst.
  - » Ein Entscheidungsbaum besteht aus Knoten, die bestimmte Zustände innerhalb eines Problems darstellen.
  - » Jeder Knoten stellt einen Punkt dar, an dem eine Entscheidung getroffen werden muss, um fortfahren zu können.
  - » Mit Hilfe von verschiedenen Logik-Regeln werden Verzweigungen aussortiert, die nicht zu akzeptablen Lösungen führen können  
⇒ Zeit zur Lösungsfundierung kann minimiert werden.

## Anwendungen von Bäumen (V)

- > Event-Scheduling
  - » Anwendungen zur Verteilung und Auslösung von Echtzeit-Ereignissen.
  - » Echzeitsysteme müssen häufig die neuesten, mit einem Ereignis verbundenen Informationen einsehen und einlesen ⇒ ein binärer Suchbaum kann die Suche nach diesen Informationen effizienter gestalten.
- > Priority Queues
  - » Diese Datenstrukturen verwenden einen binären Baum, um die Elemente einer Menge festzuhalten, die die höchste Priorität besitzen.

## Beschreibung binärer Bäume (I)

- > Hierarchische Anordnung von Knoten, von denen jeder *maximal 2 Nachfolger* hat.
- > Knotenarten: Eltern, Kinder, Geschwister
- > Nachfahren (alle Knoten unterhalb eines bestimmten Knotens), Vorfahren (alle Knoten zwischen diesem Knoten und der Wurzel)
- > Die mit dem Baum verknüpfte *Performance* wird häufig über dessen Höhe beschrieben, d.h.: über die Anzahl an *Ebenen (Levels)*, in denen Knoten vorhanden sind.

## Beschreibung binärer Bäume (II)

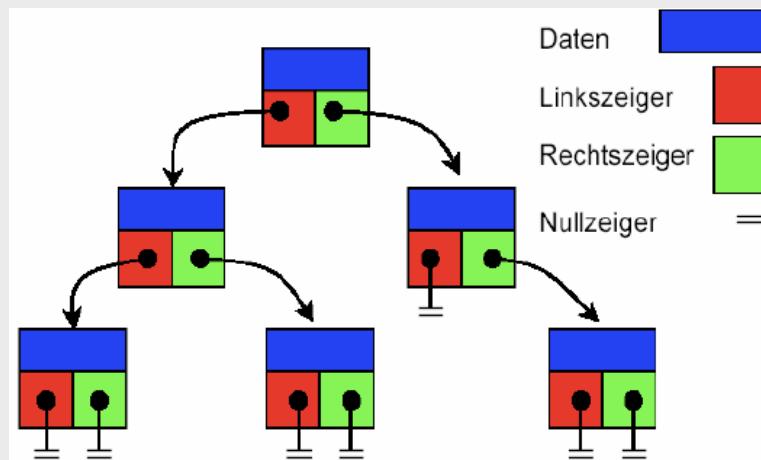
- > Ein **Zweig** ist eine Reihe von Knoten, die an der Wurzel beginnen, und an irgendeinem Blatt-Knoten (leaf-node) enden.
- > **Blattknoten** sind Knoten am Rand des Baumes, die keine Kind-Elemente aufweisen.
- > Arbeitet man mit mehreren Bäumen gleichzeitig, dann spricht man auch davon, dass die Bäume einen **Wald** bilden.

## Beschreibung binärer Bäume (III)

- > Jeder Knoten eines binären Baumes besteht aus drei Teilen:
  - » Zeiger „left“
  - » Zeiger „right“
  - » Datenelement
- > Die Zeiger left und right eines Knotens zeigen auf dessen Kind-Elemente.
- > Gibt es kein Kind-Element, zeigt der Zeiger auf NULL.
- > Der NULL-Zeiger ist ein bequemer Indikator für das Ende eines Zweiges.

## Realisierung eines binären Baumes

> Realisierung mittels Zeiger:

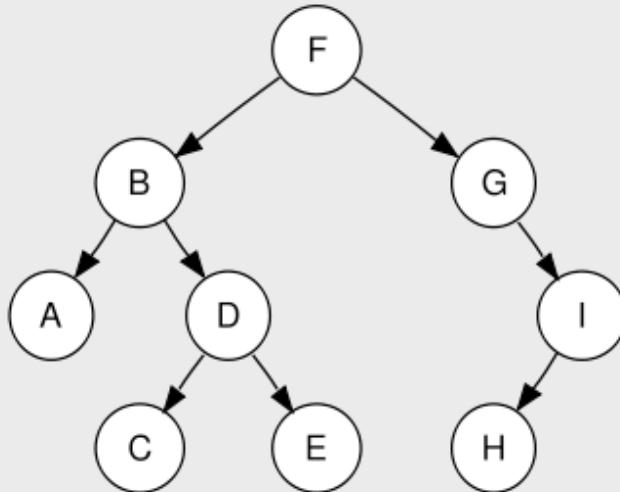


## Durchlaufen von Bäumen (I)

- > Beim *Durchlaufen (Traversing)* eines binären Baumes werden die einzelnen Knoten in einer bestimmten Reihenfolge besucht.
- > Im Gegensatz zu verketteten Listen ist die Navigation durch einen Baum *nicht direkt ersichtlich*.
- > Tatsächlich gibt es mindestens 4 Wege durch den Baum:
  - » *Preorder (Prefix)*
  - » *Inorder (Infix, Inline)*
  - » *Postorder (Postfix)*
  - » *Level Order (ebenenweise)*
- > Implementierung kann bspw. rekursiv erfolgen.

## Durchlaufen von Bäumen (II)

> Beispiel zur Illustration des Durchlaufens:



## Durchlaufen von Bäumen (III)

> Preorder (Prefix) (Root-Left-Right)

- » Zuerst der Wurzelknoten, dann nach links und dann nach rechts.
- » Gibt es links und rechts wieder Unterbäume zu durchlaufen, dann wird auf die gleiche Weise verfahren, wobei der linke bzw. rechte Knoten als Wurzel des neuen Unterbaumes verwendet wird.

> ⇒ Die Preorder-Bewegung sucht zuerst die tiefsten Elemente auf (Depth-First-Suche).

> Beispiel: F, B, A, D, C, E, G, I, H

## Durchlaufen von Bäumen (IV)

- > Inorder (Infix, Inline) (Left-Root-Right)
  - » Es wird zuerst nach ganz links navigiert und von dort begonnen.
  - » Dann zum Wurzelknoten und dann nach rechts.
  - » Wenn es links und rechts wiederum Unterbäume zu durchlaufen gibt, wird auf die gleiche Weise verfahren, wobei der linke bzw. rechte Knoten wieder als Wurzel des neuen Unterbaumes verwendet werden kann.
- > Beispiel: A, B, C, D, E, F, G, H, I

## Durchlaufen von Bäumen (V)

- > Postorder (Postfix) (Left-Right-Root)
  - » Zuerst nach links,
  - » Dann nach rechts
  - » Und dann zum Wurzelknoten
  - » Bei Unterbäumen gleiches Verfahren anwenden.
- > Beispiel: A, C, E, D, B, H, I, G, F

## Durchlaufen von Bäumen (VI)

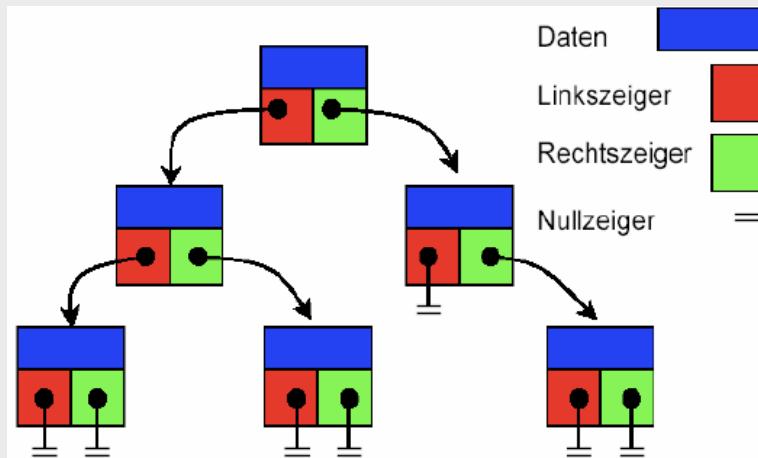
- > Level Order (ebenenweise)
  - » Die Knoten werden ausgehend vom Root-Knoten auf jeder Ebene zuerst von Links nach Rechts besucht,
  - » Dann wird die nächste Ebene besucht
  - » Bis zur untersten Ebene.
  - » ⇒ Wird auch als Breitensuche (Breadth-First) bezeichnet.
- > Beispiel: F, B, G, A, D, I, C, E, H

## Implementierung binärer Bäume (I)

- > Ein binärer Baum kann als zweifach verkettete Liste realisiert werden.
- > Ein Knoten beinhaltet die eigentliche Information und zwei Zeiger:
  - » Zeiger auf den linken Nachfolger
  - » Zeiger auf den rechten Nachfolger
- > Blattknoten enthalten zwei NULL-Zeiger

## Implementierung binärer Bäume (II)

> Realisierung mittels Zeiger:



## Implementierung binärer Bäume (III)

> Implementierung in C:

» Vereinbarung einer Knotenstruktur:

```
typedef struct _node {
    char *data;
    int key /* zur steuerung der suche */
    struct _node * left;
    struct _node * right;
} node;
```

» Vereinbarung eines Datentyps für den Baum:

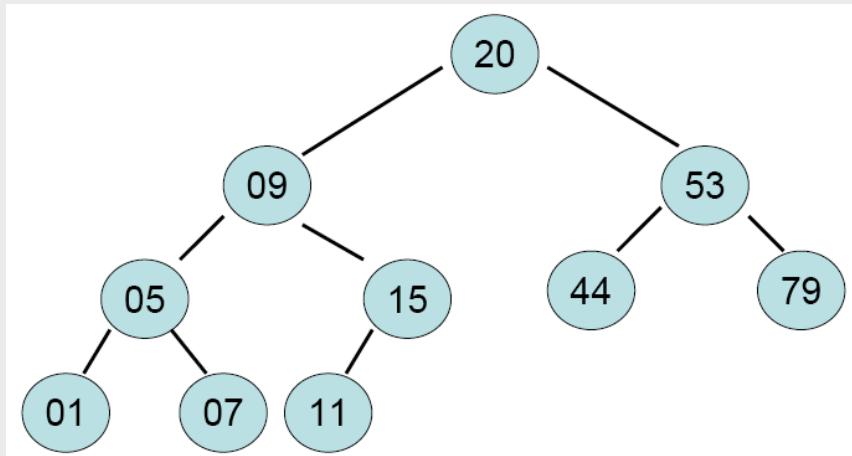
```
typedef struct _binaryTree {
    int size;
    node * root;
} binaryTree;
```

## Binäre Suchbäume (I)

- > Sind binäre Bäume, deren Aufbau sich speziell an den Bedürfnissen von Suchoperationen orientiert.
- > Finden eines Knotens:
  - » Beginn bei der Wurzel.
  - » Ebene für Ebene nach unten durchgehen, bis der Knoten gefunden wird.
  - » Wird ein Knoten gefunden der größer ist als der gesuchte, dann wird dem linken Zeiger gefolgt.
  - » Wird ein Knoten gefunden der kleiner ist als der gesuchte, dann wird dem rechten Zeiger gefolgt.
  - » Erreicht man das Ende eines Zweiges (NULL-Pointer) ohne den Knoten gefunden zu haben, dann existiert dieser Knoten nicht.

## Binäre Suchbäume (II)

- > Die Suche in einem binären Suchbaum erfordert, dass die Knoten alle in der gleichen Art und Weise eingefügt werden.
  - » Beginn an der Wurzel
  - » Ebene für Ebene tiefer gehen
  - » Entsprechend des neu einzufügenden Knotens wird nach links oder rechts navigiert.
  - » Wird das Ende eines Zweiges erreicht (NULL-Pointer), dann wird das Element (der Knoten) eingefügt.

**Binäre Suchbäume (III)****Binäre Suchbäume (IV)**

- > Sind **effiziente Datenstrukturen**, für Suchoperationen, da im schlimmsten Fall nur die Daten eines Zweiges untersucht werden müssen.
- > **Vorbedingung:** Der Baum muss **ausgeglichen** sein, damit nicht allzu lange Zweige entstehen.
- > Was passiert wenn ein Suchbaum nicht ausgeglichen ist (im schlimmsten Fall stehen alle Knoten innerhalb eines Zweiges)?
- > ⇒ Baum ausgleichen durch Rotation, siehe dazu AVL Baum.

## Implementierung Einfügen (I)

- > Falls der Baum leer ist, wird der Knoten direkt an der Wurzel eingefügt.
- > Nach dem erfolgreichen Einfügen wird die Anzahl der Knoten um 1 erhöht.

```
node * insert(binaryTree *t, int key, char* data) {  
    node *newNode;  
  
    newNode = insertNode(t->root, key, data);  
    if (t->root == NULL) t->root = newNode;  
    if (newNode != NULL) t->size++;  
  
    return newNode;  
}
```

## Implementierung Einfügen (II)

- > Zum Einfügen eines Knotens wird der Baum von der Wurzel bis zu den Blättern durchlaufen.
  - » Ist der besuchte Knoten der NULL-Zeiger so wurde die Einfügestelle erreicht.
  - » Andernfalls wird die einzufügende Information mit der Information des aktuellen Knoten verglichen.
  - » Je nach Ergebnis des Vergleichs wird entweder nach links oder nach rechts verzweigt.

## Implementierung Einfügen (III)

> Fall 1: Der besuchte Knoten ist der NULL-Zeiger:

```
node * insertNode(node *n, int key, int data) {  
    node *newNode;  
  
    if (n == NULL) {  
        newNode = (node *)malloc(sizeof(node));  
        if (newNode != NULL) {  
            newNode->data = data;  
            newNode->key = key;  
            newNode->right = newNode->left = NULL;  
        }  
  
        return newNode;  
    }  
  
    // Fortsetzung folgt
```

## Implementierung Einfügen (IV)

> Fall 2: Der besuchte Knoten ist ein innerer Knoten:

```
// Fortsetzung  
  
if (key < n->key) {  
    newNode = insertNode(n->left, key, data);  
    if (n->left == NULL)  
        n->left = newNode;  
}  
else {  
    newNode = insertNode(n->right, key, data);  
    if (n->right == NULL)  
        n->right = newNode;  
}  
  
return newNode;
```

## Bestimmung der Baumtiefe

```
int max(int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}  
  
int treeDepth(node *n) {  
    if (n == NULL) {  
        return 0;  
    }  
    else {  
        return 1 + max(treeDepth(n->left),  
                        treeDepth(n->right));  
    }  
}
```

## Finden eines Knoten im Baum (I)

- > Dazu wird der Baum von der Wurzel bis zu den Blättern durchlaufen.
- > Stößt man auf einen Knoten mit den gesuchten Key, so wird dieser Knoten zurückgeliefert.
- > Ansonsten wird entsprechend des Keys entweder nach links oder rechts verzweigt und weitergesucht.

## Finden eines Knoten im Baum (II)

```
node * lookup(node *n, int key) {  
    if (n == NULL) {  
        return NULL; // Knoten nicht gefunden  
    }  
    if (key == n->key) {  
        return n; // Knoten wurde gefunden  
    }  
    if (key < n->key) {  
        return lookup(n->left, key);  
    }  
    else {  
        return lookup(n->right, key);  
    }  
}
```

## Bäume ausgleichen

- > Das Ausgleichen eines Baumes (*Tree Balancing*) dient dazu, dass der Baum so flach wie möglich gehalten wird.
- > D.h.: Es muss sichergestellt werden, dass eine Ebene des Baumes vollständig aufgefüllt wird, bevor ein Knoten auf der nächsten Ebene auftauchen darf.
  - » Formal: Baum ist ausgeglichen,
    - wenn alle Blattknoten auf einer Ebene liegen,
    - bzw. wenn alle Blattknoten auf zwei Ebenen liegen, und die vorletzte Reihe voll ist.
- > Ein ausgeglichener Baum ist links-ausgeglichen, wenn alle Blätter der letzten Ebene die am weitesten links liegenden Positionen besetzen.

## AVL-Baum - Allgemein (I)

- > Binäre Suchbäume funktionieren nur dann gut, wenn der Baum ausgeglichen bleibt.
- > Ausgleichen eines binären Baumes ist komplex
  - » Ansatz: Implementierung des Baumes als AVL-Baum
- > Ein AVL-Baum ist eine besondere Form des binären Baums.
  - » Es wird bei jedem Knoten eine zusätzliche Information abgelegt: der *Gleichgewichtskoeffizient (balance factor)*

## AVL-Baum - Allgemein (II)

- > Der Gleichgewichtskoeffizient ist die Höhe des vom linken Child ausgehenden Teilbaumes minus der Höhe des vom rechten Child ausgehenden Teilbaumes:

```
treeDepth(n->left) - treeDepth(n->right)
```

### AVL-Baum - Allgemein (III)

- > Während des Einfügens stellt sich der AVL-Baum immer so ein, dass alle Gleichgewichtskoeffizienten bei +1, -1 oder 0 bleiben.
  - » Teilbaum, dessen Wurzelknoten den Koeffizienten +1 aufweist, wird als *linkslastig* bezeichnet.
  - » Teilbaum, dessen Wurzelknoten den Koeffizienten -1 aufweist, wird als *rechtslastig* bezeichnet.
  - » Teilbaum, dessen Wurzelknoten den Koeffizienten 0 aufweist, ist *ausgeglichen*.
- > AVL-Baum bleibt insgesamt *annähernd ausgeglichen*.

### AVL-Baum - Einfügen

- > Die grundsätzliche Vorgangsweise beim Einfügen ist ident zum binären Baum.
- > Beim AVL-Baum erfolgt zusätzlich:
  - » Bestimmung der Gleichgewichtskoeffizienten der Teilbäume.
  - » Hat ein Knoten den Koeffizienten +2 oder -2, dann muss der Baum von diesem Punkt aus ausgeglichen werden.  
→ *Rotationen*.

## AVL-Baum - Rotationen (I)

- > Die Rotation gleicht einen Teil eines AVL-Baumes aus.
  - » Die Knoten werden neu angeordnet.
  - » Die Beziehung der Knoten zueinander bleibt erhalten (linker Knoten < Parent < rechter Knoten) ⇒ bleibt ein binärer Suchbaum.
  - » Nach der Rotation liegen die Gleichgewichtskoeffizienten aller Knoten im rotierten Teilbaum bei +1, -1 oder 0.

## AVL-Baum - Rotationen (II)

- > Es gibt vier Arten von Rotationen, die durchgeführt werden müssen:
  - » LL-Rotation (Links-Links)
  - » LR-Rotation (Links-Rechts)
  - » RR-Rotation (Rechts-Rechts)
  - » RL-Rotation (Rechts-Links)
- > Annahmen:
  - » x ist der gerade an der richtigen Position eingefügte Knoten.
  - » A soll der am nächsten liegende Vorfahre von x sein, dessen Gleichgewichtskoeffizient sich auf + (-) 2 verändert hat.

## AVL-Baum - LL-Rotation

- > Wird durchgeführt, wenn  $x$  im linken Teilbaum des linken Teilbaumes von  $A$  liegt.
- >  $left$  sei das linke Child-Element von  $A$
- > Der linke Zeiger von  $A$  wird auf das rechte Child-Element von  $left$  gesetzt.
- > Der rechte Zeiger von  $left$  wird auf  $A$  gesetzt.
- > Der  $A$  referenzierende Zeiger wird auf  $left$  gesetzt.
- > Nach der Rotation wird der Gleichgewichtskoeffizient von  $A$  und  $left$  auf 0 gesetzt, alle anderen Koeffizienten bleiben gleich.

## AVL-Baum - LR-Rotation (I)

- > Wird durchgeführt, wenn  $x$  im rechten Teilbaum des linken Teilbaumes von  $A$  liegt.
- >  $left$  ist das linke Child von  $A$ .
- >  $grandchild$  ist das rechte Child von  $left$ .
- > Das rechte Child von  $left$  wird auf das linke Child von  $grandchild$  gesetzt.
- > Das linke Child von  $grandchild$  wird auf  $left$  gesetzt.
- > Das linke Child von  $A$  wird auf das rechte Child von  $grandchild$  gesetzt.
- > Das rechte Child von  $grandchild$  wird auf  $A$  gesetzt.
- > Der  $A$  referenzierende Zeiger wird auf  $grandchild$  gesetzt.

## AVL-Baum - LR-Rotation (II)

- > Die Korrektur der Koeffizienten der Knoten hängt vom ursprünglichen Koeffizienten von *grandchild* ab.
- > Drei mögliche Fälle:
  - » War der ursprüngliche Gleichgewichtskoeffizient von *grandchild* +1, wird der Koeffizient von *A* auf -1 und der von *left* auf 0 gesetzt.
  - » War der Koeffizient von *grandchild* 0, so wird der Koeffizient von *A* und *left* auf 0 gesetzt.
  - » War der Koeffizient von *grandchild* -1, wird der Koeffizient von *A* auf 0 und der von *left* auf +1 gesetzt
- > In ALLEN Fällen wird der Koeffizient von *grandchild* auf 0 gesetzt.

## AVL-Baum - RR-Rotation

- > Wird durchgeführt, wenn *x* im rechten Teilbaum des rechten Teilbaumes von *A* liegt.
- > RR-Rotation ist symmetrisch zur LL-Rotation.
- > *right* sei das rechte Child-Element von *A*.
- > Setzen des rechten Zeigers von *A* auf das linke Child von *right*.
- > Setzen des linken Zeigers von *right* auf *A*.
- > Setzen des *A* referenzierenden Zeigers auf *right*.
- > Nach der Rotation werden die Gleichgewichts-koeffizienten von *A* und *right* auf 0 gesetzt; alle anderen Koeffizienten bleiben unverändert.

## AVL-Baum - RL-Rotation (I)

- > Wird durchgeführt, wenn  $x$  im linken Teilbaum des rechten Teilbaums von  $A$  liegt.
- > RL-Rotation ist symmetrisch zur LR-Rotation.
- >  $right$  sei das rechte Child von  $A$  und  $grandchild$  das linke Child von  $right$ .
- > Setzen des linken Child von  $right$  auf das rechte Child von  $grandchild$ .
- > Setzen des rechten Child von  $grandchild$  auf  $right$ .
- > Setzen des rechten Child von  $A$  auf das linke Child von  $grandchild$ .
- > Setzen des linken Child von  $grandchild$  auf  $A$ .
- > Setzen des  $A$  referenzierenden Zeigers auf  $grandchild$ .

## AVL-Baum - RL-Rotation (II)

- > Korrektur der Gleichgewichtskoeffizienten der Knoten nach der RL-Rotation hängt vom ursprünglichen Gleichgewichtskoeffizienten von  $grandchild$  ab.
- > Drei mögliche Fälle:
  - » War der ursprüngliche Koeffizient von  $grandchild$  bei +1 wird der Koeffizient von  $A$  auf 0 und der von  $right$  auf -1 gesetzt.
  - » War der ursprüngliche Koeffizient von  $grandchild$  bei 0, wird der Koeffizient von  $A$  und  $right$  auf 0 gesetzt.
  - » War der ursprüngliche Koeffizient von  $grandchild$  bei -1, wird der Koeffizient von  $A$  auf +1 und der von  $right$  auf 0 gesetzt.
- > In allen Fällen wird der Koeff. von  $grandchild$  auf 0 gesetzt; alle anderen Koeffizienten bleiben unverändert.

## Heaps und Priority Queues

### Heaps und Priority Queues (I)

- > Bei vielen Problemen kommt es darauf an sehr schnell das größte bzw. kleinste Element einer Menge zu bestimmen (in der oft Daten eingefügt und gelöscht werden).
- > Möglicher Ansatz: Die Menge sortiert vorzuhalten:
  - » Das erste Element ist das größte bzw. das kleinste Element (je nach Sortierung) und kann sehr schnell gefunden werden.
  - » Andererseits kostet ständige Sortierung sehr viel Zeit.
  - » Weiters wird zuviel sortiert (es muss nicht unbedingt jedes Element sortiert vorliegen).
  - » Um das größte bzw. kleinste Element schnell zu ermitteln, muss eigentlich nur festgehalten werden wo dieses Element zu finden ist.  
⇒ *Heaps & Priority Queues*

## Heaps und Priority Queues (II)

> Heaps:

- » Bäume, die so organisiert sind, dass der Knoten mit dem größten Wert sehr schnell ermittelt werden kann.
- » Der Aufwand dafür ist kleiner, als wenn alle Elemente vollständig sortiert werden.
- » Kann auch so organisiert werden, dass das kleinste Element sehr einfach bestimmt werden kann.

## Heaps und Priority Queues (III)

> Priority Queues:

- » Prioritätswarteschlangen
- » Lassen sich auf natürliche Weise aus Heaps ableiten.
- » In einer Priority Queue sind die Daten in einem Heap so organisiert, dass die Knoten mit der nächsthöheren Priorität schnell bestimmt werden können.
- » Die Priorität kann je nach Anwendung verschiedene Dinge bedeuten.

## Heaps und Priority Queues (IV)

- > Beispiel Sortieralgorithmus Heap-Sort:
  - » Die zu sortierenden Daten befinden sich in einem Heap.
  - » Knoten werden einer nach dem anderen aus dem Heap extrahiert, und am Ende einer sortierten Menge platziert.
  - » Während ein Knoten sortiert wird, wandert der nächste Knoten für die sortierte Menge an den Anfang des Heaps.
  - » Heap-Sort besitzt die gleiche Laufzeit-Komplexität wie Quick-Sort.

## Heaps und Priority Queues (V)

- > Beispiel Task-Scheduling:
  - » Operation die von Betriebssystemen vorgenommen wird
  - » Um zu bestimmen, welcher Prozess als nächstes von der CPU ausgeführt werden soll.
  - » Betriebssysteme ändern laufend die Prioritäten von Prozessen ⇒  
eine Priority-Queue bietet eine effiziente Möglichkeit, um sicherzustellen, dass der Prozess mit der höchsten Priorität als nächstes ausgeführt wird.

## Heaps und Priority Queues (VI)

- > Beispiel Packetsortierung:
  - » Ein von Transportunternehmen verwendeter Prozess, um den Paket-Transport prioritätsbezogen abzuwickeln.
  - » Pakete mit frühem Zustellungsdatum erhalten höhere Prioritäten ⇒ gelangen dadurch schneller durch das System und werden früher zugestellt.
  - » Implementierung mittels Priority Queue.

## Heaps und Priority Queues (VII)

- > Beispiel Huffman-Codierung:
  - » Methode der Datenkomprimierung
  - » Verwendung des Huffmann-Baumes
    - Symbole in den Daten bekommen Codes zugewiesen.
    - Häufig vorkommende Symbole bekommen kurze Codes.
    - Weniger vorkommende Symbole bekommen lange Codes.
  - » Huffmann-Baum ⇒ Zusammenfügen von kleineren binären Bäumen.

## Heaps und Priority Queues (VIII)

- > Beispiel Lastverteilung:
  - » Häufig werden Nutzungsstatistiken für Server mit ähnlichen Aufgaben geführt.
  - » Beispiel: Tritt eine neue Verbindungsanforderung ein, kann eine Priority-Queue verwendet werden, um festzustellen, welcher Server die Anforderung am besten bearbeiten kann.

## Heap (I)

- > Ein Heap ist üblicherweise ein *Baum*.
- > Jeder *Child-Knoten* hat einen *kleineren Wert* als sein Parent-Knoten.
- > Die *Wurzel* beinhaltet damit den *höchsten Wert*.
- > Für umgekehrte Sortierung:
  - » Die Wurzel hat den kleinsten Wert.
  - » Die Parents haben kleinere Werte als die Childs.

## Heap (II)

- > Bäume wie Heaps sind *teilgeordnet*:
  - » Die Knoten einer Ebene sind, verglichen mit den Knoten einer anderen, nicht notwendigerweise sortiert
  - » Die Knoten entlang eines Zweiges weisen jedoch eine bestimmte Ordnung auf.
- > Ein Heap, bei dem jedes Child-Element kleiner ist als der Parent wird als *top-heavy* bezeichnet.
- > Ein Heap, bei dem jedes Child-Element größer ist als der Parent wird als *bottom-heavy* bezeichnet.

## Heap (III)

- > Heaps sind links-ausgeglichen (vgl. Kapitel binäre Bäume)
  - » ⇒ beim Einfügen von Knoten wächst der Baum Ebene für Ebene von links nach rechts.
- > Gute Möglichkeit um links-ausgeglichene Bäume darzustellen:
  - » Knoten kontinuierlich in einem Array (0-indiziert) ablegen, und zwar in der Reihenfolge in der sie beim Level-Durchlauf vorgefunden werden.
  - » Der Parent jedes Knotens ausgehend von einer Position  $i$  im Array liegt an der Position  $\lfloor(i-1)/2\rfloor$
  - » Die linken und rechten Child-Elemente befinden sich an den Positionen  $2i+1$  und  $2i+2$
  - » Diese Organisation ist für Heaps wichtig, da der letzte Knoten damit sehr schnell gefunden werden kann.

## Element im Heap Einfügen

- > Platzieren des neuen Knotens am Ende des Arrays.
- > Wird die Heap-Eigenschaft verletzt, muss der Baum entsprechend korrigiert werden:
  - » Betrachten des Zweiges in den der Knoten eingefügt wurde (Baum war zu Beginn ein Heap ⇒ die Heapbedingung kann nur innerhalb eines Zweiges verletzt worden sein)
  - » Beginnend mit dem neuen Knoten wird Ebene für Ebene durch den Baum navigiert.
  - » Jeweils Child-Element mit Parent vergleichen.
  - » Bei falscher Reihenfolge werden Child und Parent vertauscht, solange bis nichts mehr zu vertauschen ist bzw. der Baumanfang erreicht wurde.

## Element aus dem Heap Extrahieren (I)

- > Es wird jeweils der oberste Knoten im Heap extrahiert.
- > Danach wird der letzte Knoten im Heap an die oberste Position kopiert.
- > Wurde die Heap-Eigenschaft verletzt, muss der Heap korrigiert werden.

## Element aus dem Heap Extrahieren (II)

- > Korrektur der Heapbedingung:
  - » Beginn an der Wurzel.
  - » Ebene für Ebene nach unten.
  - » Vergleichen von beiden Child-Elementen mit dem Parent.
  - » Liegen Parent und Child in der falschen Reihenfolge vor werden die Inhalte vertauscht.
  - » Bewegung zum Child-Element dessen Abweichung am größten war.
  - » Durchführen bis keine Vertauschung mehr notwendig ist bzw. bis ein Blattknoten erreicht wird.

## Implementierung von Priority Queues

- > Implementierung ist auf verschiedene Arten
- > Einfach:
  - » Sortierte Menge der Daten (Array).
  - » Nachteil: Die Sortierung im Worst-Case vom Aufwand  $O(n)$ .
- > Bessere Lösung:
  - » Menge mit Hilfe eines Heaps teilsortiert halten.
  - » Der oberste Knoten hat immer die höchste Priorität.
  - » Vorteil: die Korrektur des Heaps beim Einfügen/Entfernen von Elementen verursacht nur den Aufwand  $O(\log n)$ .
  - » ⇒ Implementierung wie beim Heap.

## Heap-Sort

- > Daten liegen als Heap vor:
  - » Heap-Bedingung muss erfüllt sein.
- > Heap-Sort:
  - » Sortieren durch Auswahl!
  - » Methode zur absteigenden Sortierung:
    - 1) Extrahieren des Wurzelknotens ⇒ Ausgabe Wurzelknoten.
    - 2) Heap wieder herstellen (siehe extrahieren von Elementen).
    - 3) Wiederholung von 1) und 2) bis alle Elemente ausgegeben wurden.

## Heap-Sort Implementierung (I)

### **Heapsort(A)**

```
1: Erstelle-Heap(A);  
2: für i=n,...,2 {  
3:   Tausche A[1] mit A[i];  
4:   Versickere(A, 1, i-1);  
5: }
```

### **Erstelle-Heap(A)**

```
1: für i=└n/2┘,...,1 {  
2:   Versickere(A, i, n);  
3: }
```

## Heap-Sort Implementierung (II)

**Versickere(A,i,m)**

```
1: // Versickere A[i] bis höchstens A[m]
2: solange (2*i <= m) {
3:   // A[i] hat linkes Kind
4:   j = 2*i;
5:   falls (j < m) dann {
6:     // A[i] hat rechtes Kind
7:     falls (A[j].key < A[j+1].key) dann {
8:       j = j + 1; // A[j] ist größtes Kind
9:     }
10:   }
11:   falls (A[i].key < A[j].key) dann {
12:     Vertausche A[i] mit A[j];
13:     i = j; //versickere weiter
14:   } sonst {
15:     i = m; //Stopp: Heap-Bedingung erfüllt
16:   }
17: }
```

## B- und B\*-Bäume

## B-Bäume - Einleitung (I)

- > Bisher behandelte Suchmethoden sind für internes Suchen gut geeignet.
  - » Daten liegen zur Gänze im Hauptspeicher.
  - » Kleine Datenmengen (Speichergrenzen!)
- > bei großen Datenmengen
  - » z.B.: bei Datenbanken
  - » Daten werden auf Festplatten bzw. großen externen Speichern abgelegt.
  - » Zugriff auf solche Daten dauert deutlich länger als auf Daten im Hauptspeicher.

## B-Bäume - Einleitung (II)

- > Zugriff auf externe Daten (Beispiel)
  - » Großer binärer und balancierter Suchbaum der Größe  $N$  der extern gespeichert wurde.
  - » Annahme: Zeiger auf die linken und rechten Unterbäume sind jeweils Adressen auf den externen Speicher.
  - » Schlüsselsuche würde ungefähr  $\log_2(N)$  Zugriffe auf das externe Medium mit sich ziehen → für  $N=10^6$  wären das 20 externe Speicherzugriffe.
- > Zugriff auf externe Daten (*Idee*)
  - » **Zusammenfassen** von jeweils 7 Knoten des Suchbaumes zu einer Speicherseite (page).

## B-Bäume - Einleitung (III)

- > Zugriff auf externe Daten
  - » Annahme: pro externem Speicherzugriff wird eine gesamte Speicherseite in den Hauptspeicher geladen.
  - » Anzahl der externen Zugriffe würde sich dritteln !
  - » Suche ist ungefähr 3 mal so schnell.
  - » Durch die Gruppierung der Knoten in *pages* wird aus dem binären Suchbaum (jeweils 2 Kinder) ein Suchbaum mit acht Kindern.
  - » In der Praxis wählt man die Größe einer *page* deutlich größer.
    - Oft entstehen dadurch Bäume mit 128 Kindern.
    - → Bei  $N=10^6$  gespeicherten Schlüsseln nur drei externe Zugriffe.

## B-Bäume (I)

- > Setzen die Idee der Seitenbildung in die Praxis um
- > → Effizientes Suchen, Einfügen und Entfernen von Datensätzen wird möglich.
- > Effizienz bezieht sich dabei speziell auf *externe* Speicherzugriffe.
- > B-Bäume wurden 1972 von Bayer und McCreight entwickelt.
- > In der Praxis wird die Seitengröße so gewählt, dass mit einem Zugriff genau eine Seite in den Hauptspeicher geladen werden kann.

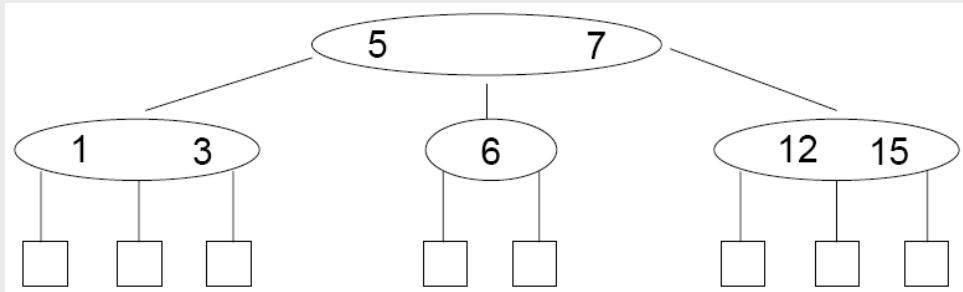
## B-Bäume (II)

- > Baumaufbau: jeder Knoten des B-Baums entspricht genau einer Speicherseite.
- > Jeder Knoten enthält jeweils mehrere Schlüssel und Zeiger auf weitere Knoten.
- > → B-Bäume sind natürliche Verallgemeinerungen von binären Suchbäumen.
  - » Wesentlicher Unterschied:
  - » Die inneren Knoten enthalten i.A. mehr als zwei Kinder.
  - » Für effiziente Suche enthalten sie auch mehrere Schlüssel (einen weniger als Kinder).

## B-Bäume (III)

- > Formale Definition (Annahme: ein Blatt eines Baumes ist ein Knoten ohne Kinder) für einen **B-Baum der Ordnung m**:
  - » Alle Blätter haben die gleiche Tiefe.
  - » Jeder Knoten, mit Ausnahme der Wurzel und der Blätter hat mindestens  $\lceil m/2 \rceil$  Kinder, die Wurzel hat mindestens 2 Kinder.
  - » Jeder Knoten mit  $l+1$  Kindern hat  $l$  Schlüssel.
  - » Für jeden Knoten  $r$  mit  $l$  Schlüsseln  $s_1, \dots, s_l$  und  $l+1$  Kindern  $v_0, \dots, v_l$  ( $\lceil m/2 \rceil \leq l+1 \leq m$ ) gilt: Für jedes  $i$ ,  $1 \leq i \leq l$ , sind alle Schlüssel in  $T_{v(i-1)}$  kleiner gleich  $s_i$ , und  $s_i$  ist kleiner gleich alle Schlüssel in  $T_{v_i}$ . Dabei bezeichnet  $T_{v_i}$  den Teilbaum mit Wurzel  $v_i$ .

## B-Bäume (IV)



- > Die Abbildung zeigt einen B-Baum der Ordnung 2.
- > Diese Bäume werden auch 2-3 Bäume genannt, weil jeder Knoten (bis auf die Blätter) entweder 2 oder 3 Kinder besitzt.

## B-Bäume (V)

- > Zweckmäßige Vorstellung:
- > Die  $l$  Schlüssel und die  $l+1$  ( $v_0$  bis  $v_l$ ) Zeiger auf die Kinder (wie in der vorigen Abbildung) sind folgendermaßen innerhalb eines Knotens  $p$  angeordnet:

$v_0$	$s_1$	$v_1$	$s_2$	$v_2$	....	$s_l$	$v_l$
-------	-------	-------	-------	-------	------	-------	-------

## B-Bäume Implementierung

- > Implementierung als verallgemeinerte Listenstruktur.
- > Jeder innere Knoten enthält die folgenden Eigenschaften:
  - » v.l = Anzahl der Schlüssel
  - » v.key[1], ... , v.key[l] = Schlüssel s1...,sl
  - » v.info[1], ... , v.info[l] = Datenfelder zu Schlüssel s1...,sl
  - » v.child[0], ... , v.child[l] = Zeiger auf Kinderknoten v0,...,vl
- > Blätter:
  - » v.l = 0, sie enthalten keinerlei Information (weder Schlüssel noch Datensätze) → In einer realen Implementierung werden sie nicht wirklich gespeichert.

## B-Bäume - Suche (I)

- > Suchen in B-Bäumen
  - » Das Suchen nach einem Schlüssel x in einem B-Baum der Ordnung m kann als natürliche Verallgemeinerung des von binären Suchbäumen bekannten Verfahrens aufgefasst werden.
    - Beginn bei der Wurzel p.
    - Suchen der Stelle i in p, die den kleinsten Schlüssel größer gleich x enthält.
    - Falls diese Stelle existiert, und p.key[i] ungleich x ist, dann sucht man im Knoten von Zeiger p.child[i-1] weiter.
    - Falls diese Stelle nicht existiert und wir in einem Blatt angekommen sind, wird „NULL“ bzw. „nicht gefunden“ zurückgegeben.
    - Insbesondere: Ist x größer als der größte Schlüssel sl in p, so ergibt sich i =l+1 und damit wird die Suche im Knoten von Zeiger p.child[l] fortgesetzt.
    - Ein Knoten wird jeweils linear durchsucht; bei großem m ist es jedoch vorteilhaft diese Suche als binäre Suche durchzuführen.

## B-Bäume - Suche (II)

**Suche(p,x):**

```
1: i=1;
2: solange i <= p.l UND x > p.key[i] {
3:   i = i + 1;
4: }
5: falls i <= p.l UND x == p.key[i] dann {
6:   retourniere(p,i) // Schlüssel gefunden
7: }
8: falls p.l == 0 dann {
9:   retourniere NULL; // Blatt erreicht:
10:                      // Suche erfolglos
11: } sonst {
11:   retourniere Suche(p.child[i-1],x); // rekursiv
12:                           // weitersuchen
13: }
```

## B-Bäume - Einfügen (I)

- > Neuen Schlüssel x in einen B-Baum einfügen:
  - » Zunächst Suche nach x
  - » Da x im Baum noch nicht vorkommt, endet die Suche erfolglos in einem Blatt  $p_0$ , das die erwartete Position des Schlüssels repräsentiert.
  - » Sei  $p$  der Vater von  $p_0$  und  $p.child[i]$  zeige auf  $p_0$
  - » Füge zunächst Schlüssel x zwischen den Schlüsseln  $s_i$  und  $s_{i+1}$  in  $p$  ein und erzeuge ein neues Blatt.
  - » ....

## B-Bäume - Einfügen (II)

- > Neuen Schlüssel  $x$  in einen B-Baum einfügen:
  - » Nun sind zwei Fälle möglich:
    - Falls  $p$  nun zu viele Kinder bzw. Schlüssel besitzt, muß  $p$  in zwei verschiedene Knoten aufgespalten werden.
    - Sind  $s_1, \dots, s_m$  die Schlüssel, so bildet man zwei neue Knoten, die jeweils die Schlüssel  $s_1, \dots, s_{\lceil m/2 \rceil - 1}$  und  $s_{\lceil m/2 \rceil + 1}, \dots, s_m$  enthalten, und fügt den mittleren Schlüssel  $s_{\lceil m/2 \rceil}$  auf dieselbe Weise in den Vater des Knotens  $p$  ein.
    - Dieses Teilen eines übergelaufenen Knotens wird solange rekursiv wiederholt, bis ein Knoten erreicht ist, der noch nicht die Maximalzahl von Schlüsseln gespeichert hat, oder bis die Wurzel erreicht ist.
    - Muss die Wurzel geteilt werden, dann erzeugt man eine neue Wurzel, welche die durch Teilung entstehenden Knoten als Kinder und den mittleren Schlüssel  $s_{\lceil m/2 \rceil}$  als einzigen Schlüssel hat.

## B-Bäume - Einfügen (III)

- > B-Bäume wachsen im Unterschied zu binären Suchbäumen immer nach oben, d.h. durch Aufspaltung der Wurzel.

## B-Bäume - Entfernen

- > Zum Entfernen eines Schlüssels aus einem B-Baum der Ordnung m sucht man den Wert und entfernt ihn aus Knoten p.
- > Enthält p nun „zu wenige“ Kinder, so „borgt“ man sich entweder Schlüssel von Geschwistern (Nachbarknoten derselben Ebene) aus, oder man verschmilzt p mit einem Geschwisterknoten.

## B-Bäume - Laufzeit

- > In allen drei Fällen (Suchen, Einfügen und Entfernen):
  - » Im **schlechtesten Fall** muss man dem Pfad von den Blättern zur Wurzel und/oder umgekehrt höchstens einmal folgen.
  - » Daraus ergibt sich eine asymptotische Laufzeit von  $O(\log \lceil m/2 \rceil (N+1))$
  - » Das heißt: mit Hilfe von B-Bäumen kann das Wörterbuchproblem in der Zeit  $O(\log \lceil m/2 \rceil N)$  gelöst werden.

## B\*-Bäume (I)

- > Unterscheiden sich von B-Bäumen im Wesentlichen dadurch, dass die Datensätze hier nur in den Blättern stehen.
- > Die Zwischenknoten enthalten statt der vollständigen Datensätze nur Schlüssel.
  - » Dienen der Steuerung des Suchvorganges.
  - » Enthalten die Adressen der jeweiligen Child-Knoten wie beim B-Baum.
- > Da keine Datensätze mehr in den Zwischenknoten gespeichert werden, kann  $m$  (die Ordnung des Baumes bzw. die Knotenbreite) vergrößert werden.
  - » Der Baum wird breiter.
  - » Einige Ebenen sind notwendig.

## B\*-Bäume (II)

- > Die Zwischenblätter enthalten:
  - » Wie beim B-Baum mindestens  $k$  und höchstens  $2k$  Schlüssel
  - » Und einen Nachfolger (Seitenadresse) mehr als Schlüssel.
- > Die Blätter beinhalten die Datensätze und keine Seitenadressen.
- > Die Operationen sind sehr ähnlich wie beim B-Baum:
  - » **Suchen**: Im Unterschied zum B-Baum muss in jedem Fall bis zu einem Blatt navigiert werden.
  - » **Einfügen**: Kleinere Unterschiede zum B-Baum ergeben sich dadurch, dass es nur einen trennenden Schlüssel (aber keinen trennenden Datensatz) gibt.
  - » **Entfernen**: Der zu entfernende Datensatz liegt immer in einem Blatt.

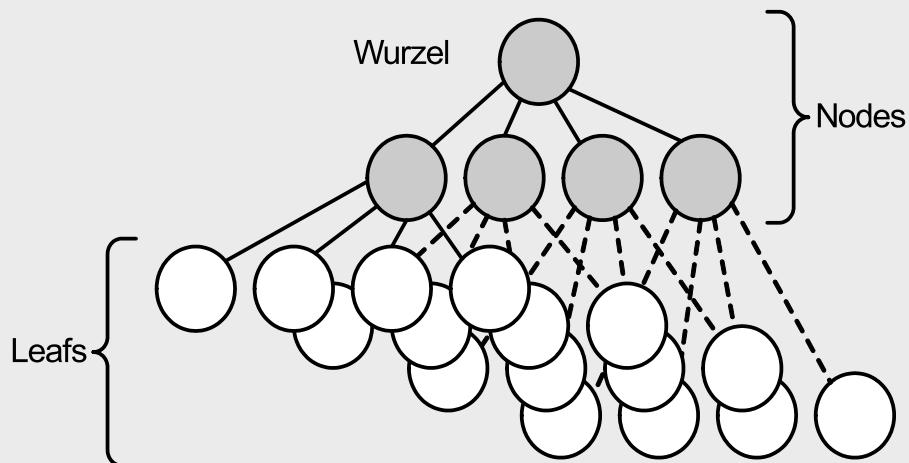
## Mehrdimensionale Bäume

## Quad Trees

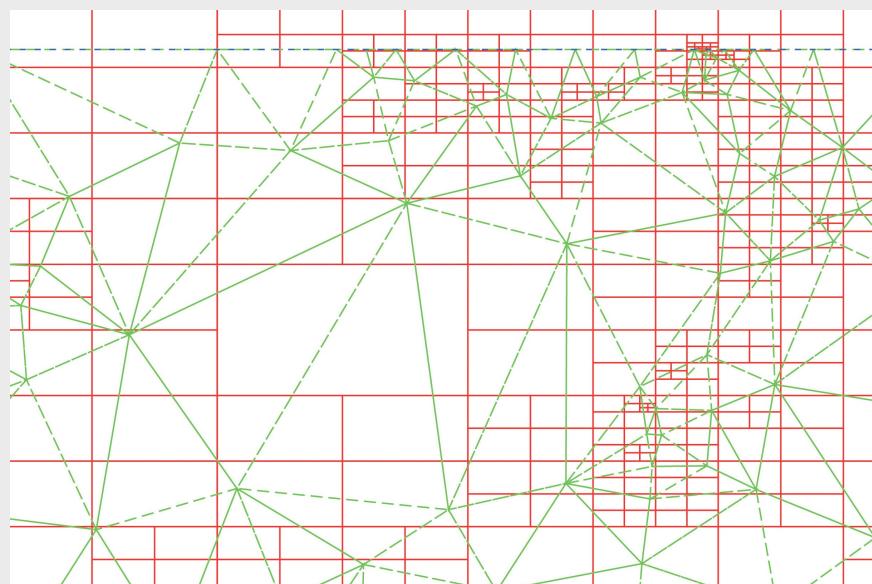
- > Problemstellung:
  - » point-location – Auffinden geometrischer Elemente an Hand von Koordinaten
  - » Bildrepräsentierung
  - » Kollisionserkennung
  - » Hidden Surface Removal
- > 4-fach verzweigter Baum zur geometrischen Bereichssuche
- > Baum speichert 2D Elemente

## Quad Trees (II)

> Topologische Ansicht



## Quad Trees (III)



## Quad Trees (IV)

```
Algorithmus: insert(element, blatt)

if blatt = node then
    for i = 1..4 do
        if not exists ( sub(blatt, i) )
            erzeugeNode( blatt, i );
        fi
        if element überlappt sub(blatt, i)
            return insert(element, sub(blatt, i));
        fi
    od
else /* blatt = leaf */
    return erzeugeÄndereLeaf(element);
fi
```

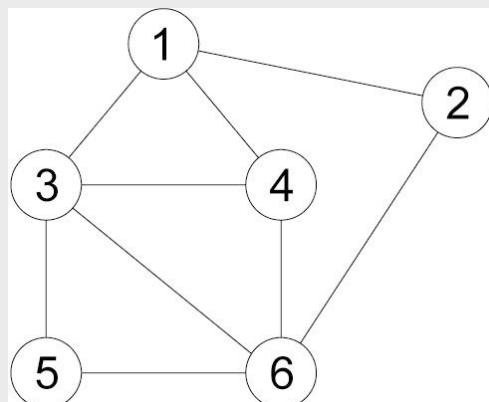
## Quad Trees (V)

- > Verschiedene Leaf Typen
  - » Point Leaf: Enthält gemeinsamen Punkt von Elementen
  - » Line Leaf: Enthält gemeinsame Linie von 2 Elementen
  - » Element Leaf: Enthält komplettes Element

## Oct Trees

- > 3D Erweiterung von quad-trees
- > Mehr Leaf Typen (z.B.: Face Leaf)
- > Sehr komplexe Geometrietests

## Graphentheorie



## Definition

- > Besteht aus mit Kanten (engl.: edges) verbundenen Knoten (engl.: vertices, nodes)
- > Unterschiede in Art der Kanten
- > Verschiedene Datenstrukturen zur Realisierung
- > Graphen definieren nur Topologie (Nachbarschaften) der Knoten, enthalten aber keine geometrischen Informationen

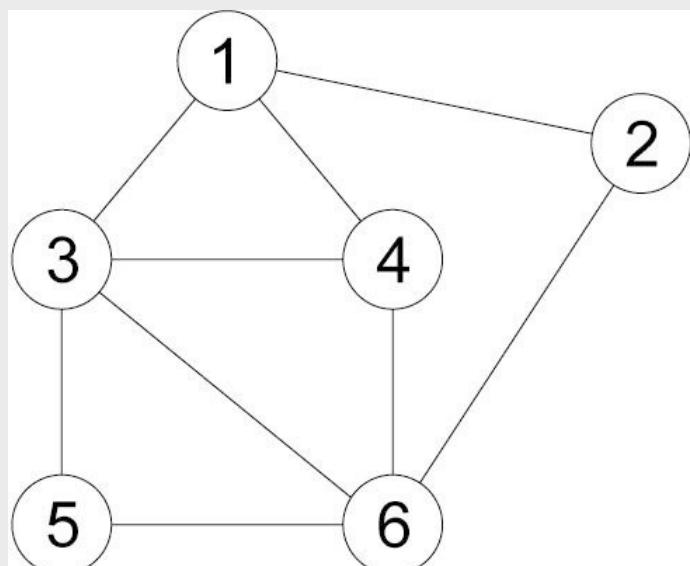
## Abgrenzung Graph – Baum

- > Bäume sind Spezialfälle von Graphen
  - » Wurzelknoten
  - » Jeder Knoten hat genau **einen** Vorgänger
  - » Zyklenfrei (keine geschlossenen Wege)
  - » Zusammenhängend
  - » Nur **ein Pfad** zwischen 2 Knoten
  - » Beispiel: Stammbäume

## Arten von Graphen

- > Ungerichtete Graphen
  - » Es wird angegeben **welcher Knoten mit welchem** verbunden ist.
  - » Kanten haben **keine Richtung!**
  - » Beispiele:
    - Strassennetz (ohne Einbahnen)
    - Ländergrenzen

## Ungerichtete Graphen



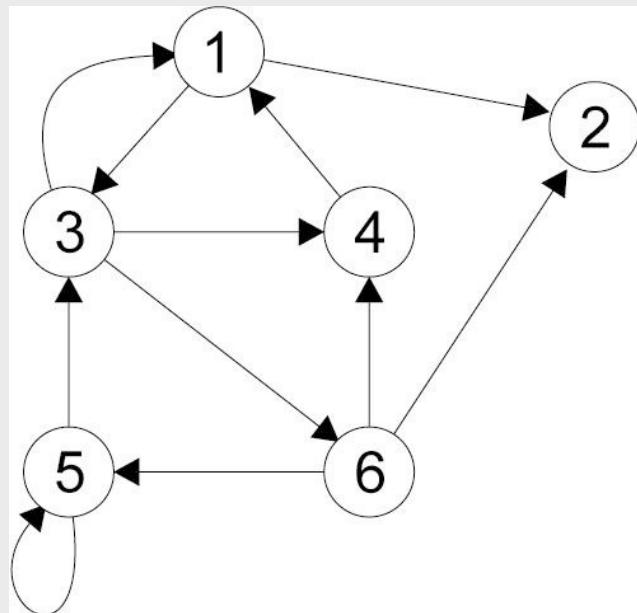
## Ungerichtete Graphen (II)

- > Definition via 2-Tupel:  $G=(V,E)$ 
  - » Endliche Menge  $V$  (vertices) von Knoten
  - » Menge  $E$  (edges eges) von Kanten
- > Beispiel:
  - »  $GU = (VU, EU)$
  - »  $VU = \{1, 2, 3, 4, 5, 6\}$
  - »  $EU = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 6\}, \{4, 6\}, \{3, 6\}, \{5, 6\}, \{3, 4\}, \{3, 5\}\}$
  - » Kanten in  $EU$  sind ungerichtet!

## Arten von Graphen (II)

- > Gerichtete Graphen
  - » Kanten haben eine **Richtung**.
  - » Bi-directionale Verbindungen zwischen Knoten haben 2 Kanten.
  - » Beispiele:
    - Strassennetz mit Einbahnen
    - Statemachines

## Gerichtete Graphen



Erstellt von: Jürgen Falb

10.02.2017

Seite 249

## Gerichtete Graphen (II)

- > Definition analog ungerichteten Graphen:  $G=(V,E)$
- > Jedes  $e \in E$  ist ein Tupel  $(a,b)$  mit  $a,b \in V$
- > Beispiel:
  - »  $GG= (VG,EG)$
  - »  $VG = \{1,2,3,4,5,6\}$
  - »  $EG = \{(1,2), (1,3), (3,1), (4,1), (3,4), (3,6), (5,3), (5,5), (6,5), (6,2), (6,4)\}$

Schleife

Erstellt von: Jürgen Falb

10.02.2017

Seite 250

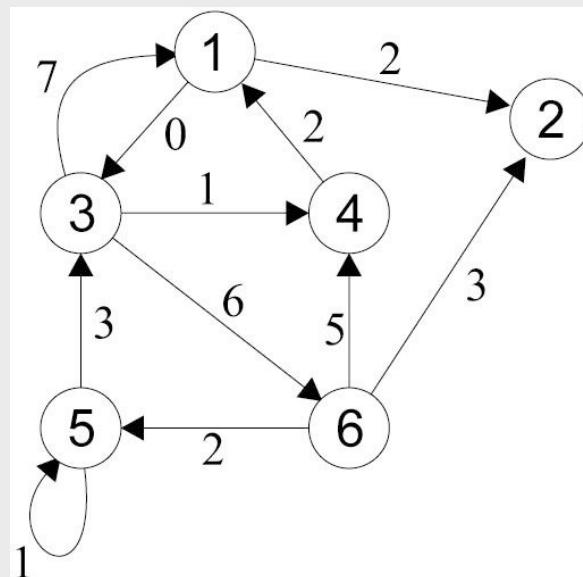
## Arten von Graphen (III)

- > Gerichtete Azyklische Graphen
  - » Kein geschlossener Rundweg entlang gerichteter Kanten.
  - » Beispiel: Stammbäume, Vererbung
- > Gewichtete Graphen

## Gerichtete Azyklische Graphen

- > Spezialfall gerichteter Graphen
- > Kein geschlossener Rundweg entlang Kanten (engl.: *direct acyclic graph, DAG*)
- > Beispiele:
  - » Stammbäume
  - » Vererbung in Programmiersprachen
- Effiziente Tests auf Zyklenfreiheit

## Gewichtete Graphen



## Gewichtete Graphen (II)

- > Zusätzlich noch Kantengewichte
- > Gewichtswerte können z.B. natürliche, oder reelle Zahlen sein.
- > Definition:  $G = (V, E, \gamma)$
- Suche nach kürzestem Weg
- Suche nach maximalem Fluss

## Realisierungen von Graphen

- > Als Liste von Zahlen in einem Array
- > Kantenliste
  - » 1.-te Zahl definiert Anzahl der Knoten
  - » 2.-te Zahl definiert Anzahl der Kanten
  - » Restliche Zahlen definieren Kanten, die jeweils durch 2 Zahlen gegeben sind
  - » Beispiel: 6,11,1,2,1,3,3,1,4,1,3,4,3,6,5,3,5,5,6,5,6,2,6,4

## Realisierungen von Graphen (II)

- > Knotenliste
  - » 1.-te Zahl definiert Anzahl der Knoten
  - » 2.-te Zahl definiert Anzahl der Kanten
  - » Restliche Zahlen definieren Knoteninformation in Form von Ausgangsgrad, sowie Liste der Zielknoten
  - » Beispiel: 6,11,2,2,3,0,3,1,4,6,1,1,2,3,5,3,2,4,5

### Realisierungen von Graphen (III)

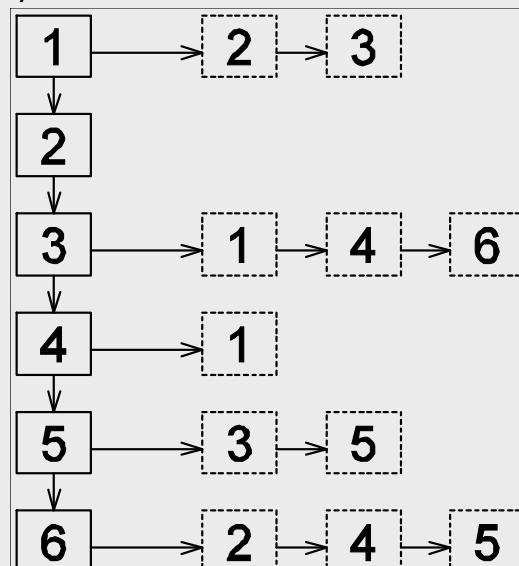
> Adjazenzmatrix

» Speicherung der Kanteninformation als  $n \times n$  Matrix  
(bei  $n$  Knoten)

$$G_g = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

### Realisierungen von Graphen (IV)

> Adjazenzliste – Dynamische Datenstruktur



## Verschiedene Realisierungen

> Vergleich der Komplexität

Operation	Kanten-liste	Knoten-liste	Adjazenz-matrix	Adjazenz-liste
Kante einfügen	O(1)	O(n+m)	O(1)	O(1)/O(n)
Kante löschen	O(m)	O(n+m)	O(1)	O(n)
Knoten einfügen	O(1)	O(1)	O(n <sup>2</sup> )	O(1)
Knoten löschen	O(m)	O(n+m)	O(n <sup>2</sup> )	O(n+m)

## Graphendurchlauf

## Breitendurchlauf

- > Alle Knoten eines Graphen werden aufsteigend nach Entfernung vom Startknoten durchlaufen
  - » Zuerst werden alle über eine Kante erreichbaren Knoten durchlaufen.
  - » Dann alle Knoten die über 2 Kanten erreichbar sind, u.s.w.

## Breitendurchlauf (II)

- > BFS Algorithmus (breadth-first-search)
- > Iterativer Breitendurchlauf
- > Warteschlange Q als Hilfsstruktur
  - » dequeue(Q) ... Entfernen des vordersten Elements
  - » enqueue(Q, s) ... Einfügen eines Elements s
  - » front(Q) ... Liefert vorderstes Element
  - » isEmpty ... Liefert true/false

### Breitendurchlauf (III)

- > Zustand eines Knotens wird durch Farbe dargestellt:
  - » Weiss → Knoten wurde noch nicht besucht
  - » Grau → Entfernung zum Startknoten bekannt
  - » Schwarz → Abgearbeitet
- > Pro Knoten wird Distanz  $d$  zum Startknoten gespeichert
- > Algorithmus erhält als Eingabe einen Graphen, sowie einen Startknoten

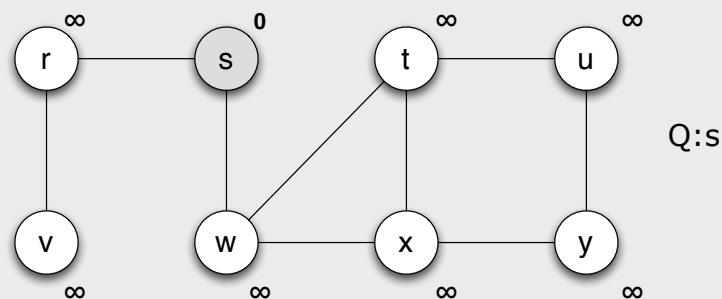
### BFS Algorithmus – Initialisierung

- > Bei der Initialisierung werden zunächst die Farbwerte aller Knoten ausser dem Startknoten auf weiss gesetzt
- > Farbe des Startknotens wird auf grau gesetzt
- > Die Warteschlange  $Q$  wird mit dem Startknoten  $s$  initialisiert
- > Alle Knoten ausser Startknoten erhalten Distanz  $d$  von unendlich ( $\infty$ )
- > Knoten  $s$  erhält Distanz 0

## BFS Algorithmus

> Zustand nach Initialisierung

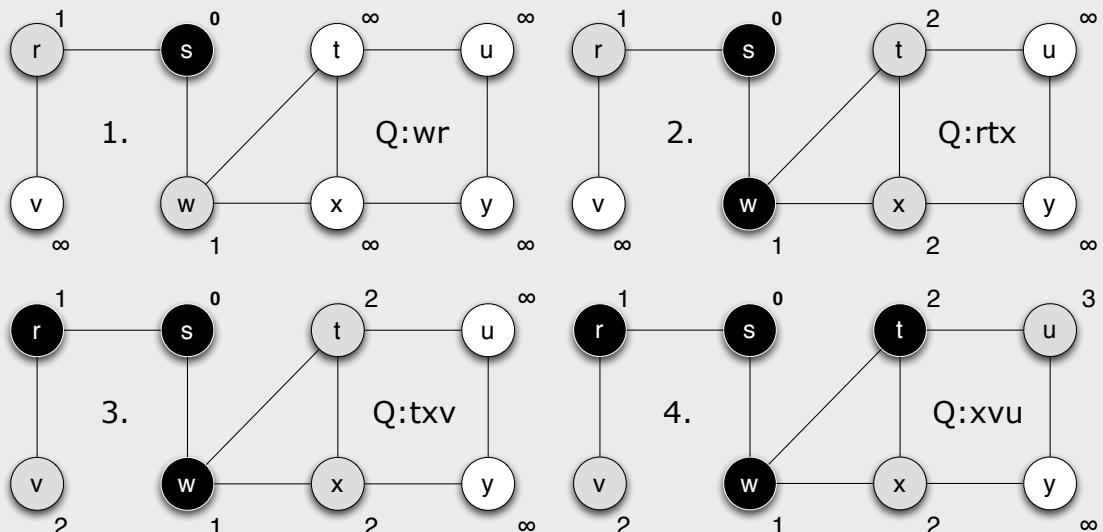
(white circle) unbearbeitet    (grey circle) Entfernung bestimmt    (black circle) abgearbeitet



## BFS – 1.-ter Schritt

- > Vorderster Knoten wird aus  $Q$  entfernt ( $s$ )
- > Die von  $s$  direkt erreichbaren Knoten  $x$  und  $w$  werden mit Distanz  $d=1$  versehen und grau eingefärbt.
- > Die Knoten  $x$  und  $w$  werden in  $Q$  gestellt
- > Knoten  $s$  wird auf schwarz gesetzt

## BFS Algorithmus (II)

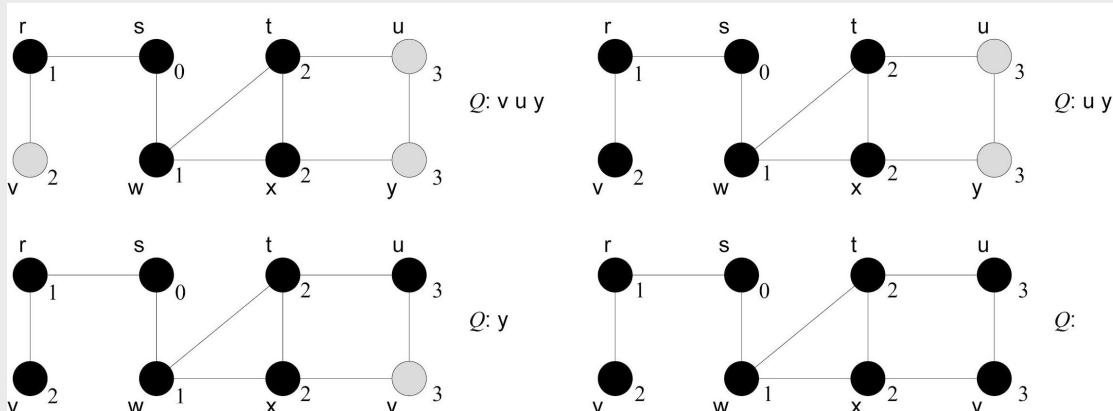


Erstellt von: Jürgen Falb

10.02.2017

Seite 267

## BFS Algorithmus (III)

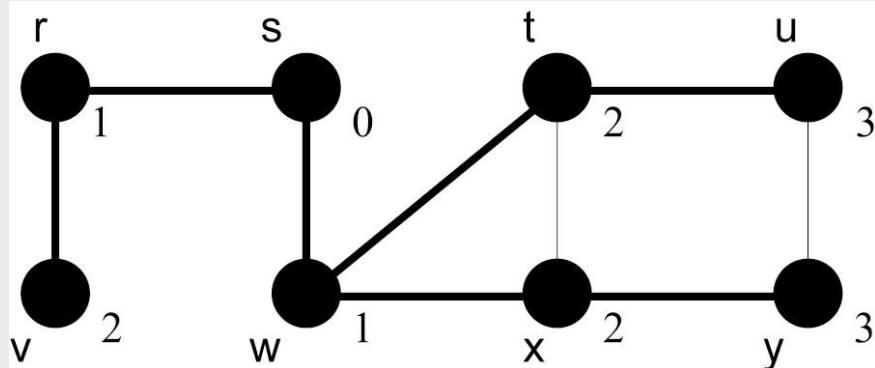


Erstellt von: Jürgen Falb

10.02.2017

Seite 268

## BFS Algorithmus (IV)



## BFS Algorithmus – Pseudocode

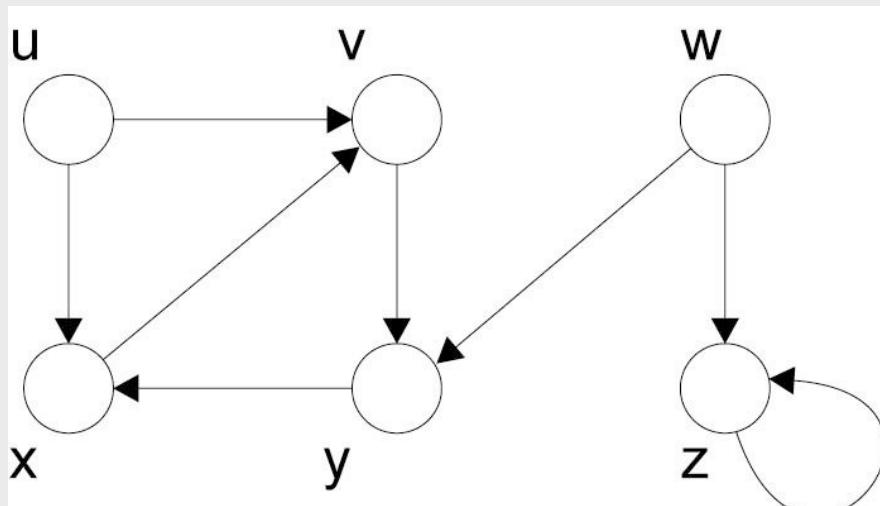
```

for each Knoten u ∈ V[G] – s do
    farbe[u]=weiss; d[u]=∞;
od
farbe[s]=grau; d[s]=0;
Q=emptyQueue; Q.enqueue(Q, s);
while not isEmpty(Q) do
    u=front(Q);
    for each v ∈ ZielknotenAusgehenderKanten(u) do
        if farbe(v)=weiss then
            farbe[v]=grau; d[v]=d[u]+1;
            Q.enqueue(Q, v);
        fi
    od
    dequeue(Q); farbe[u]=schwarz;
od
    
```

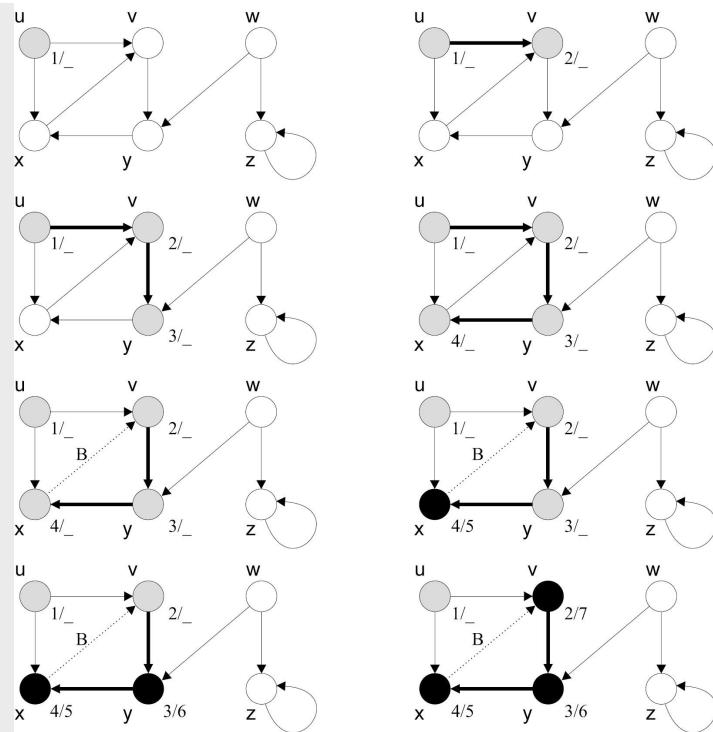
## Tiefendurchlauf

- > DFS Algorithmus (*depth-first-search*)
- > Rekursiver Algorithmus
- > DFS geht „soweit als möglich“ einen gewählten Pfad entlang
- > Pro Knoten werden Beginn (d) und Ende (f) der Bearbeitung eines Knotens abgespeichert

## DFS Algorithmus



## DFS Algorithmus (II)

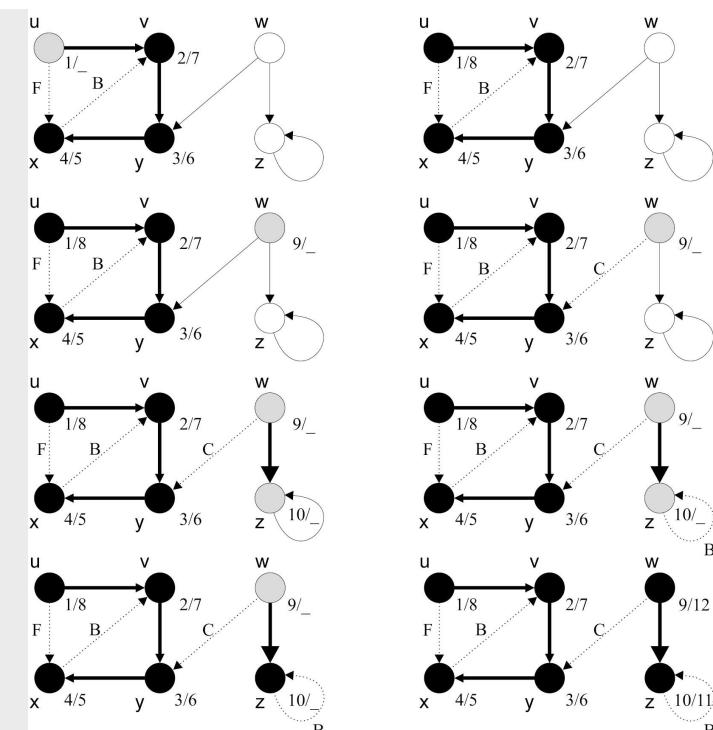


Erstellt von: Jürgen Falb

10.02.2017

Seite 273

## DFS Algorithmus (III)



Erstellt von: Jürgen Falb

10.02.2017

Seite 274

## DFS Algorithmus (IV)

### Algorithmus DFS(G)

```
for each Knoten u ∈ V[G] do
    farbe[u]=weiss;
od
zeit=0;
for each Knoten u ∈ V[G] do
    if farbe[u]=weiss then
        DFS-visit(u);
    fi
od
```

## DFS Algorithmus (V)

### Algorithmus DFS-visit(u)

```
farbe[u]=grau; zeit=zeit+1; d[u]=zeit;

for each v ∈ ZielknotenAusgehenderKanten(u) do
    if farbe[v]=weiss then
        DFS-visit(v);
    fi
od

farbe[u]=schwarz; zeit=zeit+1; f[u]=zeit;
```

## Test auf Zyklenfreiheit

- > Zyklenfreiheit wichtig für z.B.: logistische Transportstrecken, oder Konsistenzüber-prüfungen von Vererbungshierarchien
- > Kann einfach mit DFS realisiert werden
- > Zyklenfreiheit ist gegeben, wenn keine *back-edges* existieren

## Topologisches Sortieren

- > Sortieren der Knoten eines azyklischen Graphen, sodass jeder Knoten nach all seinen Vorgängern kommt
- > → Scheduling Probleme
- > Ist mittels DFS realisierbar
  - » Man lasse DFS laufen
  - » Beim Setzen von  $f[v]$  wird der Knoten vorne in eine lineare Liste eingehängt

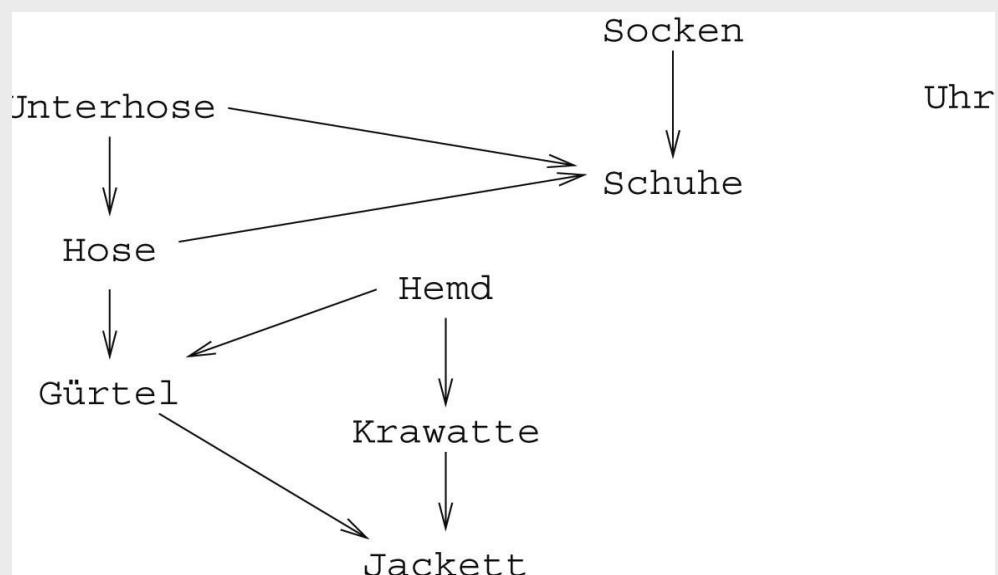
## Topologisches Sortieren (II)

> Beispiel: zerstreuter Professor

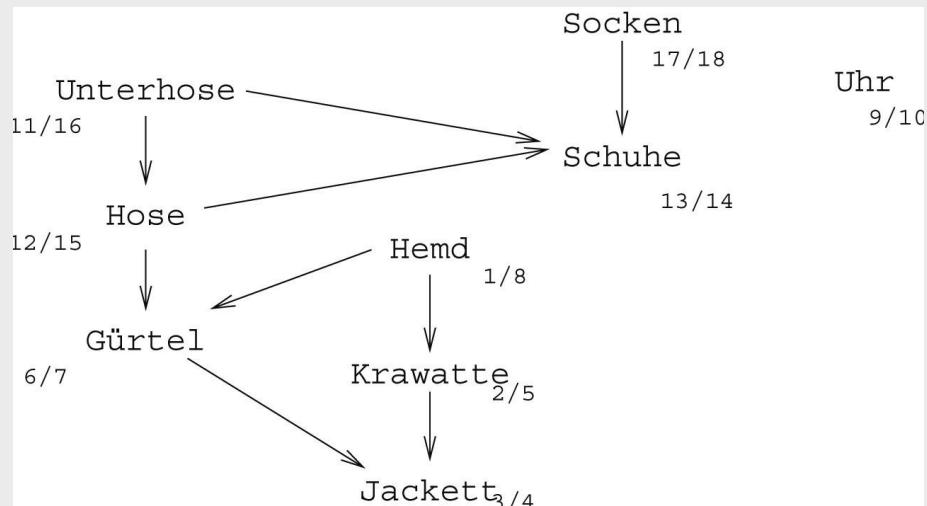
Unterhose vor Hose  
Hose vor Gürtel  
Hemd vor Gürtel  
Gürtel vor Jackett  
Hemd vor Krawatte

Krawatte vor Jackett  
Socken vor Schuhen  
Unterhose vor Schuhen  
Hose vor Schuhen  
Uhr: egal

## Topologisches Sortieren (III)



## Topologisches Sortieren (IV)



## Topologisches Sortieren (V)

> Ermittelte Reihenfolge

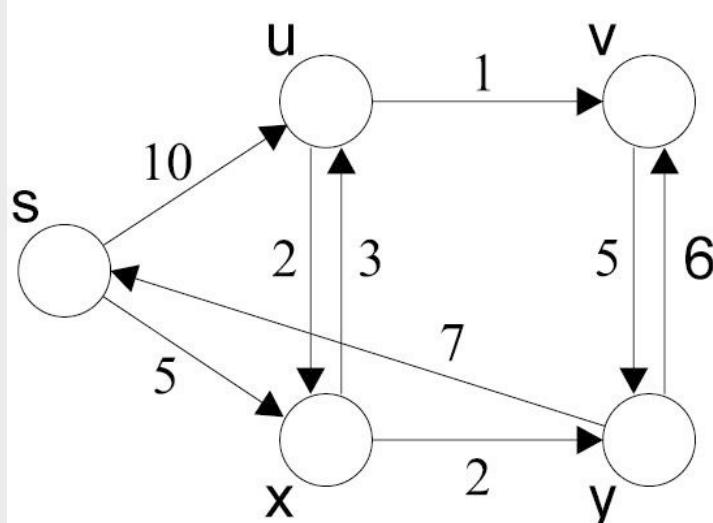
18 Socken	8 Hemd
16 Unterhose	7 Gürtel
15 Hose	5 Krawatte
14 Schuhe	4 Jackett
10 Uhr	

## Kürzeste Wege

## Anwendungsfälle f. gewichtete Graphen

- > Ungerichtet:
  - » Flugverbindungen mit Meilenangaben
  - » Straßennetze mit Kilometerangaben
- > Gerichtet:
  - » Straßennetze mit Einbahnstrassen

## Beispielgraph



## Kürzeste Wege

- > Aufgabe: Finde den günstigsten Weg durch einen Graphen
- > Formal:  $G = (V, E, \gamma)$
- > Pfad (Weg) durch  $G$  ist gegeben durch eine Liste von aufeinanderstoßenden Kanten
- $P = \langle (v_1, v_2), (v_2, v_3), (v_3, v_4), \dots, (v_{n-1}, v_n) \rangle$

## Kürzeste Wege (II)

> Bewertung von Pfaden zum Vergleich notwendig:

$$w(P) = \sum_{i=1}^{n-1} \gamma((v_i, v_{i+1}))$$

> Distanz  $d(u, v)$  zweier Punkte ist das Gewicht des kürzesten Pfades von  $u$  nach  $v$

## Dijkstras Algorithmus

- > Von Dijkstra 1959 veröffentlicht
- > Basiert auf Greedy Prinzip
- > Funktioniert nur für nichtnegative Gewichte

## Dijkstras Algorithmus (II)

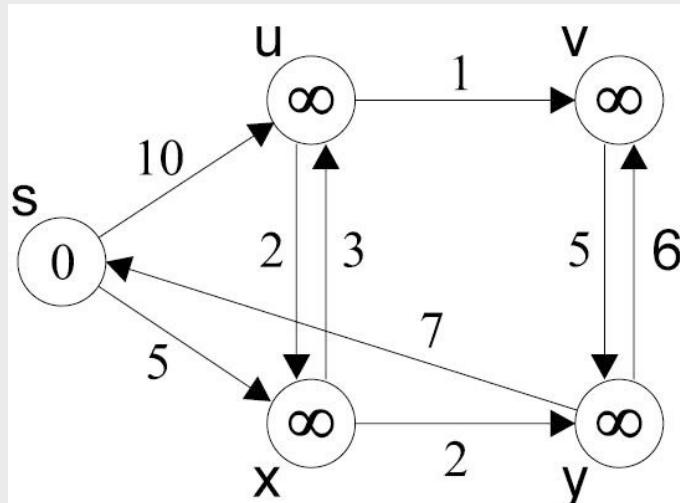
**Algorithmus Dijkstra( $G, s, t$ )**

```

for each Knoten  $u \in V[G] - s$  do
     $D[u] = \infty;$ 
od;
 $D[s] = 0;$  PriorityQueue  $Q = V;$ 
while not isEmpty( $Q$ ) do
     $u = \text{extractMinimum}(Q);$ 
    if  $u = t$  then end
    for each  $v \in \text{ZielknotenAusgehenderKanten}(u) \cap Q$  do
        if  $D[u] + \gamma((u,v)) < D(v)$  then
             $D[v] = D[u] + \gamma((u,v));$ 
            adjustiere  $Q$  an neuen Wert  $D[v]$ 
        fi
    od

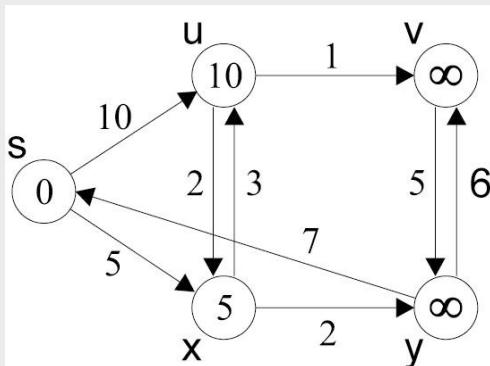
```

## Dijkstras Algorithmus (III)

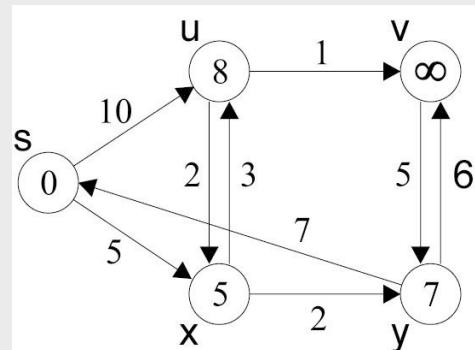


$$Q = \langle (s:0), (u:\infty), (v:\infty), (x:\infty), (y:\infty) \rangle$$

## Dijkstras Algorithmus (IV)



$Q = \langle (x:5), (u:10), (v:\infty), (y:\infty) \rangle$



$Q = \langle (y:7), (u:8), (v:\infty) \rangle$

## Bellman-Ford-Algorithmus

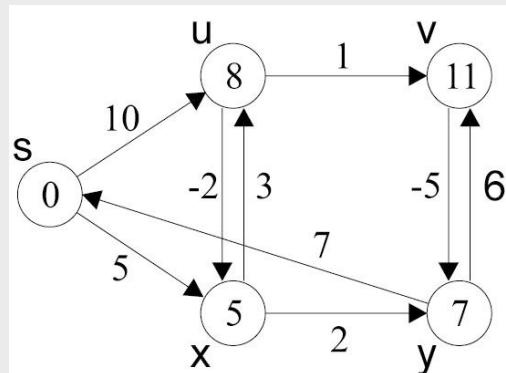
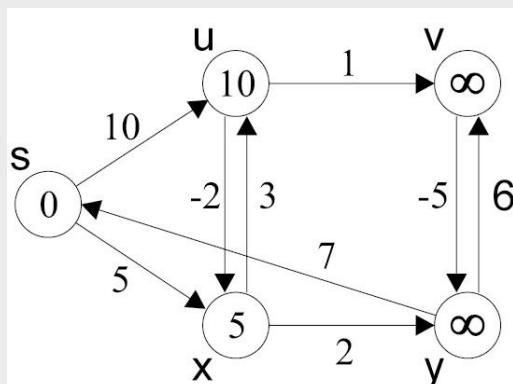
- > Alternativer „kürzester Weg“ Algorithmus
- > Funktioniert auch für negative Kantengewichte
- > Im i-ten Durchlauf werden alle Pfade der Länge i untersucht

## Bellman-Ford-Algorithmus (II)

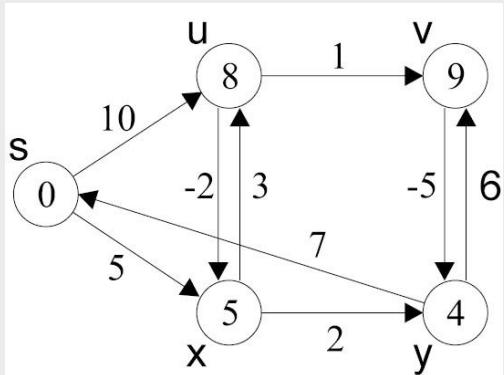
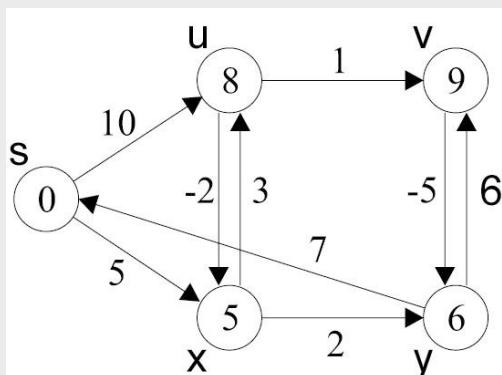
```
Algorithmus BF (G, s)

D[s] = 0;
for i = 1 to |V| -1 do
    for each (u,v) ∈ E do
        if D[u] + γ((u,v)) < D[v] then
            D[v] = D[u] + γ((u,v));
        fi
    od
od
```

## Bellman-Ford-Algorithmus (III)



## Bellman-Ford-Algorithmus (IV)

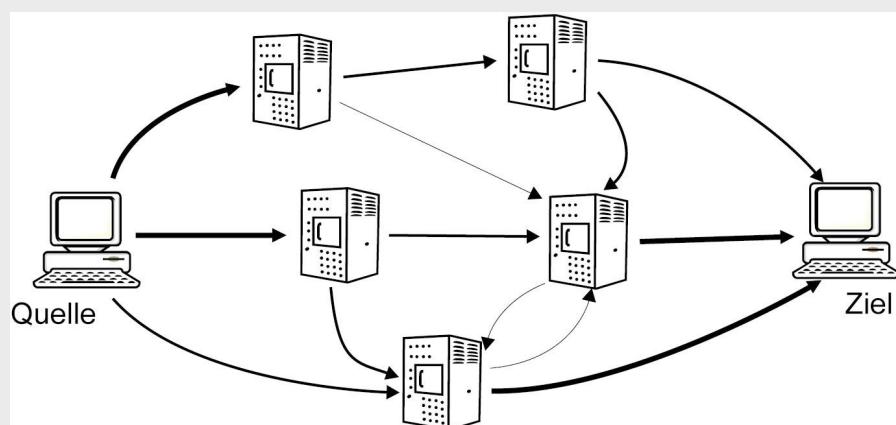


## Durchflußberechnung

## Maximaler Durchfluss

- > Meist durch logistische Aufgaben motiviert:
  - » Kapazitäten eines Transportunternehmens
  - » Paketvermittlung in Computernetzen
- > Jede Kante wird mit einer Kapazität  $c$  (maximaler Fluss) und einem aktuellen Fluss ( $f$ ) belegt
- > Verfügbare Kapazität:  $c - |f|$

## Maximaler Durchfluss (II)



### Maximaler Durchfluss (III)

Korrekter Durchfluss:

$$|f(u, v)| \leq c((u, v)) \rightarrow \text{Einhaltung der Kap.}$$

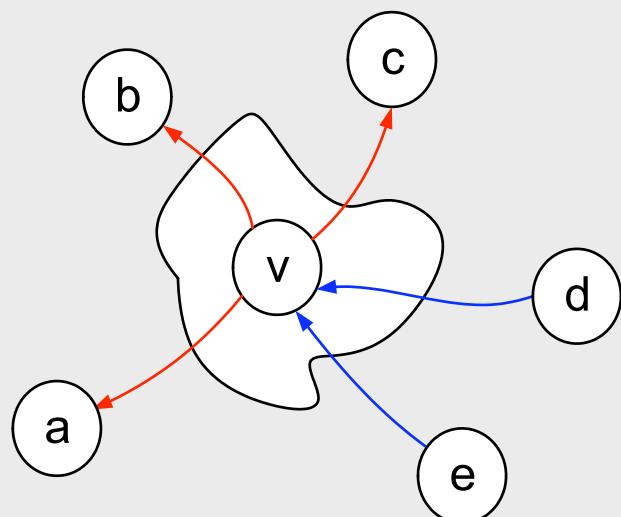
$$f(u, v) = -f(v, u) \rightarrow \text{Konsistenz}$$

$$\sum f(u, v) = 0 \quad \rightarrow \text{Bewahrung des Flusses}$$

mit  $v \in V - \{q, z\}$  pro Knoten  $u \in V$

### Maximaler Durchfluss (IV)

> Bewahrung des Flusses (Quellenfreiheit):



## Maximaler Durchfluss (V)

> Wert des Flusses einer Quelle:

$$val(G, F, q) = \sum_{u \in V} f(q, u)$$

> Maximaler Fluss:

$$\max\{val(G, F, q) | F \text{ korrig. in } G \text{ bez. } q, z\}$$

## Ford-Fulkerson

- > Bestimmt den maximalen Fluss zwischen Quelle und Ziel
- > Greedy Algorithmus mit Zufallskomponenten

## Ford-Fulkerson (II)

Initialisiere Graph mit leerem Fluss;

**do**

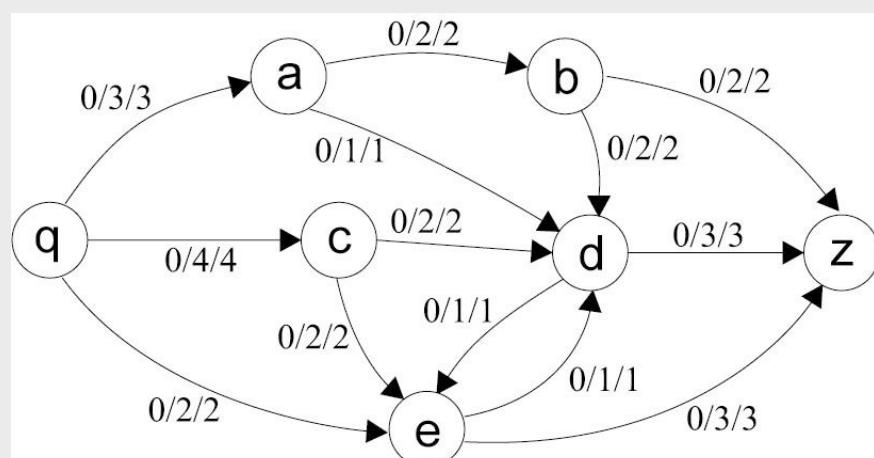
  wähle nutzbaren Pfad aus;

  füge Fluss des Pfades zum Gesamtfluss hinzu;

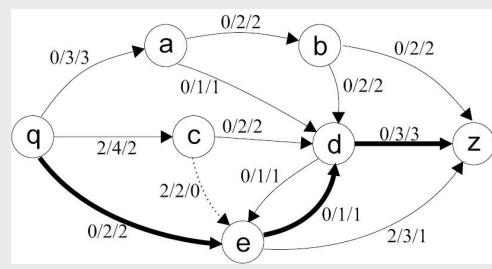
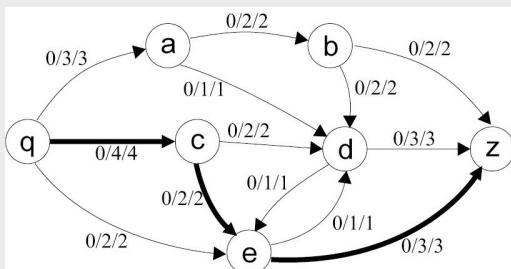
**while** noch nutzbare Pfade verfügbar

> Auswahl von nutzbaren Pfaden z.B. mittels Tiefensuche

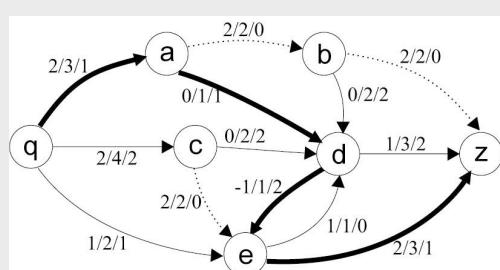
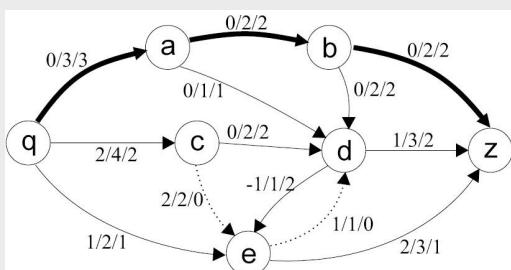
## Ford-Fulkerson (III)



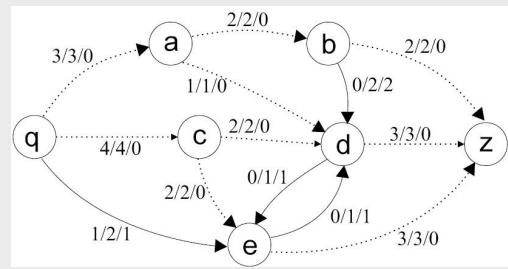
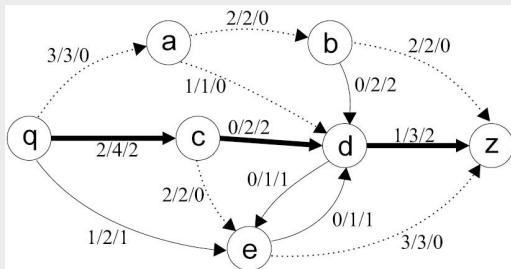
## Ford-Fulkerson (IV)



## Ford-Fulkerson (V)



## Ford-Fulkerson (VI)



## Planare Graphen und Färbeproblem

## Planare Graphen

- > Gegeben: beliebiger Graph G
- > Frage: lässt sich dieser Graph ohne überschneidende Kanten (planar) zeichnen?
- > Anwendung: Chip- oder Leiterplattendesign

## Planare Graphen (II)

- > Auffinden planarer Graphen mit Qualitätsmerkmalen
  - Winkel-, oder Längenkriterien
- > Minimierung der Kantenüberschneidungen (Anzahl der Brücken minimieren)
- > Zerlegen eines nichtplanaren Graphen in planare Teilgraphen (Leiterbahnen auf mehreren Ebenen)

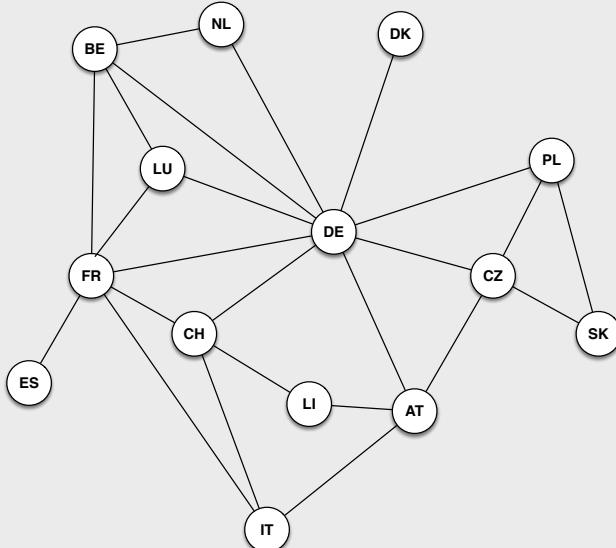
## Einfärben von Graphen

- > Einfärben von Knoten, sodass keine zwei benachbarten Knoten dieselbe Farbe haben
- > z.B.: zum Einfärben von Landkarten, oder zur Vergabe von überschneidungsfreien Klausurterminen

## Einfärben von Graphen (II)

- > 2 Subprobleme:
  - » Die eigentliche Einfärbung
  - » Die Bestimmung der minimal benötigten Anzahl von Farben

## Einfärben von Graphen (III)



Erstellt von: Jürgen Falb

10.02.2017

Seite 313

## Travelling Salesman

- > Problem: Handlungsreisender soll nacheinander  $n$  Städte besuchen und dann an den Ausgangspunkt zurückkehren
- > Jede Stadt soll nur einmal besucht werden
- > Naiver Ansatz untersucht alle möglichen Varianten
  - » Aufwand:  $O(2^n)$
- > Meist Optimierungsverfahren zur Lösung eingesetzt

Erstellt von: Jürgen Falb

10.02.2017

Seite 314

## Genetische Algorithmen

## Genetische Algorithmen

- > Manche Probleme können nicht analytisch gelöst werden, bzw. eine exakte Lösung wäre zu aufwendig
  - » Beispiel: Travelling Salesman
- > Idee:
  - » Erzeuge eine Anzahl von Lösungskandidaten (Population) zufällig
  - » Wähle jene Individuen aus, die ein bestimmtes Gütekriterium erfüllen
  - » Kombiniere die besten Individuen zu einer neuen Generation

## Genetische Algorithmen (II)

- > 1970 in den USA entwickelt
  - » J. Holland, K. DeJong, D. Goldberg
- > Durch Prinzipien der Informationsverwaltung in der Natur inspiriert
  - » Fortpflanzung (Generationen)
  - » Mutation
  - » Auslese
- > Eigenschaften
  - » Globaler Optimierungsalgorithmus
  - » Langsam
  - » Nicht Deterministisch

## Genetische Algorithmen (III)

In der Informatik:

- > Pool von Individuen (Population)
- > Individuen lösen ein Problem näherungsweise
- > Individuen werden bewertet (Fitness)
- > Aus den besten Individuen entsteht jeweils eine neue Generation
  - » Crossover
  - » Mutation

## Genetische Algorithmen

> Jeder genetische Algorithmus läuft wie folgt ab:

Erzeuge Startpopulation

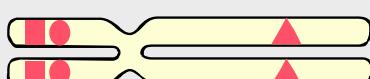
```
while true do
    Berechne Fitness der (neuen) Individuen;
    if bestes Individuum akzeptabel then
        return bestes Individuum;
    fi
    Selektiere Individuen fuer Reproduktion;
    Erzeuge Kinder;
    Mutiere Kinder;
    Ersetze Teile der Population durch Kinder;
od
```

## Variationen Genetischer Algorithmen

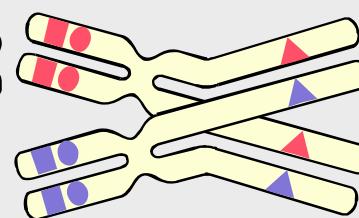
> Genetische Algorithmen werden nach Art der Selektion unterschieden:

- **Simple** – Jeweils die ganze Population einer Generation wird durch Kinder ersetzt (Klassischer Genetischer Algorithmus)
- **Steady-State** – Ein Teil der Population wird durch Kinder ersetzt; Einige Individuen überleben in die nächste Generation (Eltern, Grosseltern)
- **Incremental** (Nur ein – zwei Kinder pro Generation)

### Crossover – Natur

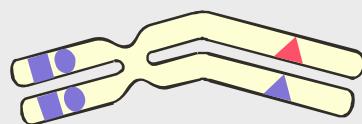


Crossing over  
der Chromosomen



Vorbild für den genetischen  
Algorithmus

+



### Crossover – Computer

1	0	0	0	1	1		0	0	1	0	0		1	0	0	1	1	0	1
1	0	1	0	1	1		1	0	1	0	1		1	1	0	1	1	0	1



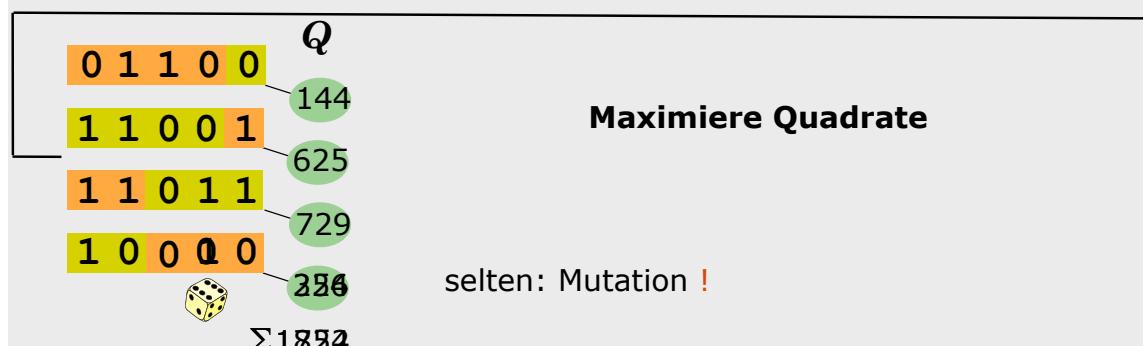
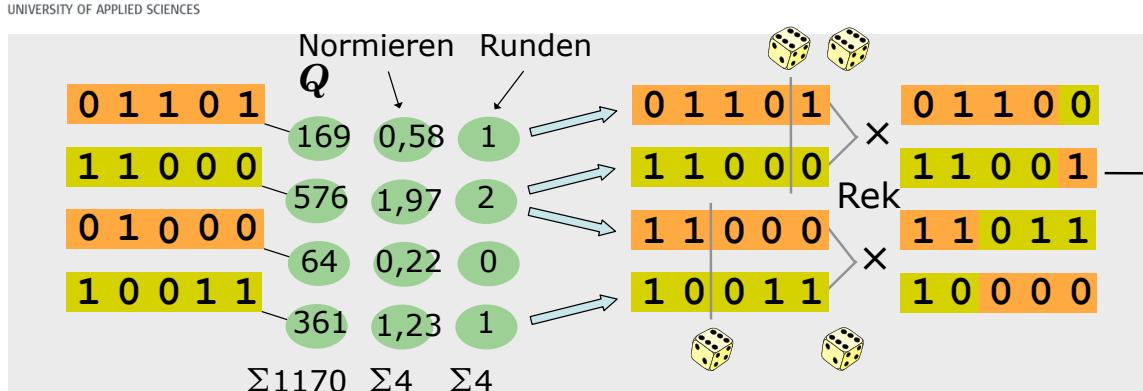
GA-Operation



1	0	0	1	1	1	0	1	0	1	0	1	0	1	1	0	1
1	0	1	0	1	0	0	1	0	0	1	1	0	1	1	0	1

### Einfaches Beispiel: Maximiere Quadrate

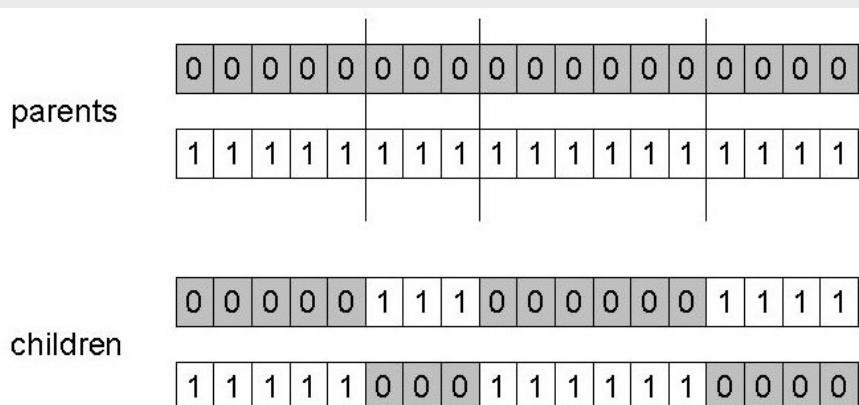
- > Gesucht ist das Maximum  $x^2$  der Zahlen  $\{0..31\}$
- > Representation als 5 bittige Binärzahl:  
[00000 ... 11111]
- > Populationsgrösse sei 4
- > Anfangspopulation zufällig gewählt:  
01101, 11000, 01000, 10011



## Crossover Varianten

### > N-point Crossover

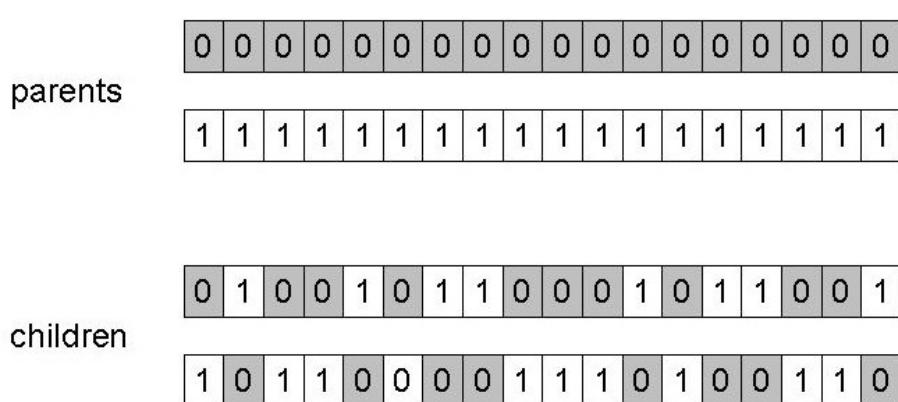
- » Genome werden an mehreren Punkten geteilt
- » Die Teile werden alternierend zusammengesetzt
- » Verallgemeinerung des 1 Punkt Crossover



## Crossover Varianten (II)

### > Uniform Crossover

- » Für jedes Gen wird eine Münze geworfen
- » Für das 2.-te Kind wird das Gen jeweils invertiert



### Crossover Varianten (III)

> Single Arithmetic Crossover

- » Wenn Representation nicht als Bitwert, sondern als Floatingpoint Zahl vorliegt
- » Parents:  $\langle x_1, \dots, x_n \rangle$  und  $\langle y_1, \dots, y_n \rangle$
- » Zufällige Auswahl eine Zahl  $k$

$$\text{Kind1: } \langle x_1, \dots, x_k, \alpha y_{k+1} + (1 - \alpha)x_{k+1}, \dots, x_n \rangle$$

» Für Kind 2 wird  $\alpha$  invertiert

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.5	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----

$$\alpha = 0.5$$



0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.2	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.5	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

Erstellt von: Jürgen Falb

10.02.2017

Seite 327

### Crossover Varianten (IV)

> Simple Arithmetic Crossover

- » Wähle eine Zufallzahl  $k$
- Alle Gene ab Spalte  $k$  werden gemischt

$$\langle x_1, \dots, x_k, \alpha y_{k+1} + (1 - \alpha)x_{k+1}, \dots, \alpha y_n + (1 - \alpha)x_n \rangle$$

» Für Kind 2 wird  $\alpha$  invertiert

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----

$$\alpha = 0.5$$



0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.2	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.1	0.2	0.3	0.4	0.5	0.6	0.5	0.5	0.6
-----	-----	-----	-----	-----	-----	-----	-----	-----

Erstellt von: Jürgen Falb

10.02.2017

Seite 328

### Crossover Varianten (V)

- > Whole Arithmetic Crossover
  - » Alle Gene werden gemischt

$$\alpha \bar{x} + (1 - \alpha) \bar{y}$$

» Für das 2.-te Kind wird  $\alpha$  invertiert

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----



$$\alpha = 0.5$$

0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.2	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6
-----	-----	-----	-----	-----	-----	-----	-----	-----

### Vergleich: Crossover – Mutation

- > Crossover führt „Sprünge“ zu Bereichen aus, die oftmals „zwischen“ den Eltern liegen
- > Crossover ist in der Lage Informationen von Elternteilen zu kombinieren
- > Mutation erzeugt lediglich kleine Abweichungen vom ursprünglichen Individuum
- > Mutation führt neue Information ein

## 8 Damen Problem

- > Problem: Positioniere 8 Damen auf einem Schachbrett, sodass keine Dame eine andere bedroht

	1	2	3	4	5	6	7	8
1			D					
2						D		
3								D
4	D							
5					D			
6							D	
7					D			
8		D						

## 8 Damen Problem – Representation

- > 8 dimensionaler Vektor
- > Einträge entsprechen Positionen der Damen am Schachbrett (Spalten)
- > Beispiel:  $v = [3,6,8,1,4,7,5,2]$  (Individuum)
- > Anfangspopulation durch zufällige Positionen
  - » Mehrere Individuen in Population

## 8 Damen Problem – Fitness

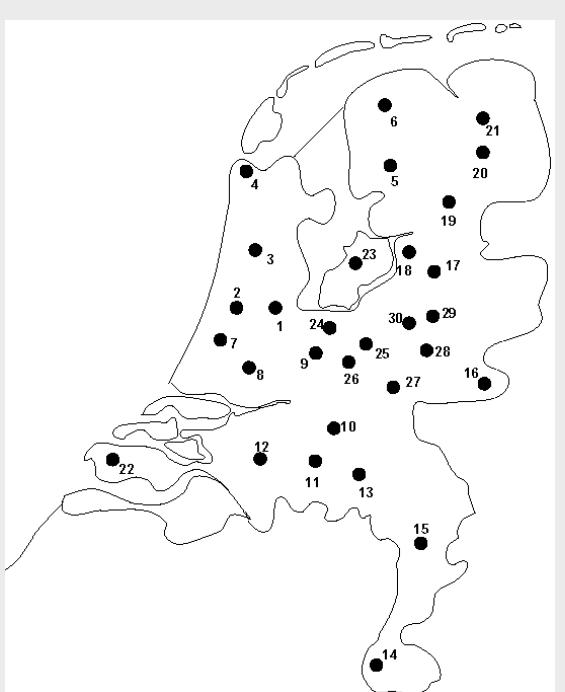
- > Bewertung eines Individuums durch Anzahl der Kollisionen (gleiche Spalte oder Diagonale)
- > Beispiel:  $v1=[1,1,1,1,1,1,1,1]$   
→  $\text{Fitness}(v1) = (-)28$
- > Negative Werte, da Fitness immer maximiert wird.  
→ Fitness = 0 ist eine Lösung

## 8 Damen Problem – Demonstration

- > Vergleich:
  - » Rekursive Lösung
  - » Lösung mittels genetischer Algorithmen

### Beispiel: Travelling Salesman

- > Problem:
  - » Bereise n Städte
  - » Finde die Tour mit minimaler Länge
- > Representation:
  - » Städte werden nummeriert:  $1, 2, \dots, n$
  - » Eine Tour wird als Permutation kodiert (z.B. für  $n = 4$   $[1,2,3,4]$ ,  $[3,4,2,1]$ )
- > Riesiger Suchbereich:
  - für 30 Städte gibt es  $30! \approx 10^{32}$  mögliche Touren



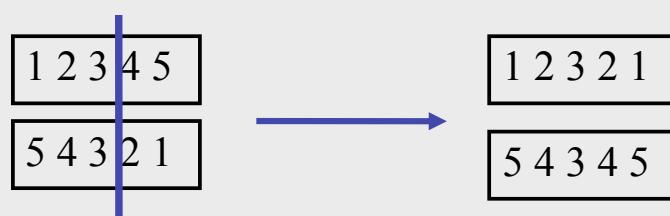
Erstellt von: Jürgen Falb

10.02.2017

Seite 335

### Crossover bei Permutationen

- > Bei TS Problem würde klassischer Crossover Reihenfolge verletzen, bzw. Städte mehrfach besuchen



- > Daher spezielle Operatoren

Erstellt von: Jürgen Falb

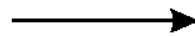
10.02.2017

Seite 336

## Crossover bei Permutationen (II)

- » Wähle zufällig einen Teil des 1.-ten Elternteiles (hier 4567)
- » Kopiere diesen Teil an der gleichen Position zum Kind

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



\_\_\_\_\_ | 4 | 5 | 6 | 7 | \_\_\_\_\_

9 | 3 | 7 | 8 | 2 | 6 | 5 | 1 | 4

- » Kopiere die verbleibenden Gene vom 2.-ten Elternteil

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



3 | 8 | 2 | 4 | 5 | 6 | 7 | 1 | 9

9 | 3 | 7 | 8 | 2 | 6 | 5 | 1 | 4

## Mutation bei Permutationen

- > Insert Mutation
  - » Wähle 2 zufällige Punkte
  - » Verschiebe den 2.-ten Punkt in Richtung des Ersten

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



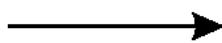
1 | 2 | 5 | 3 | 4 | 6 | 7 | 8 | 9

- > Dadurch wird die Ordnung beibehalten!
- > Nachbarschaftsbeziehungen grossteils aufrecht

## Mutation bei Permutationen (II)

- > Swap Mutation
  - » Wähle 2 zufällige Punkte
  - » Vertausche die beiden Gene

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



1	5	3	4	2	6	7	8	9
---	---	---	---	---	---	---	---	---

- > Verletzt die Ordnung mehr als bei Insert Mutation

## Algorithmen der Spieltheorie

## Minimax

- > Problem: minimizing the maximum possible loss
- > Algorithmus zur Ermittlung der optimalen Spielstrategie
- > 2 Spieler, abwechselnd
- > Einsatz bei Nullsummenspiele (Schach, Dame, TicTacToe aber auch in der Wirtschaft)

## Minimax (II)

- > Ausgangspunkt: aktuelle Spielsituation
  - » Spieler A werden große Zahlen zugeordnet
  - » Spieler B werden kleine Zahlen zugeordnet (üblicherweise invers)
  - » Spieler A maximiert, Spieler B minimiert
- > Algorithmus:
  - » Aufbau eines Suchbaums durch Ermittlung aller Folgespielsituationen bis zur Tiefe X
  - » Bewertung der Blätter
  - » Rekursive Übertragung der Bewertung bis zur Wurzel, wobei wenn A am Zug ist, das Maximum übertragen wird und bei B das Minimum.

### Minimax (III)

> Beispiel für einen Suchbaum:

Ebene 1:

+4

Ebene 2:

-2

+4

Ebene 3:

+4

-2

-3

-2

+7

+6

+4

-1

+4

Blätter:

### Minimax (IV)

> Ideale Bewertungsfunktion:

- » +1, wenn Spieler A gewinnt
- » -1, wenn Spieler B gewinnt
- » 0, wenn unentschieden

> Perfektes Spiel: Suchbaum kann bis zur maximalen Tiefe aufgebaut werden → meist zu aufwendig

> Daher: Abbruch bei entsprechender Suchtiefe

> Heuristik als Bewertungsfunktion

> Aufwand: steigt exponentiell mit Suchtiefe:  $O(a^n)$

## Minimax (V)

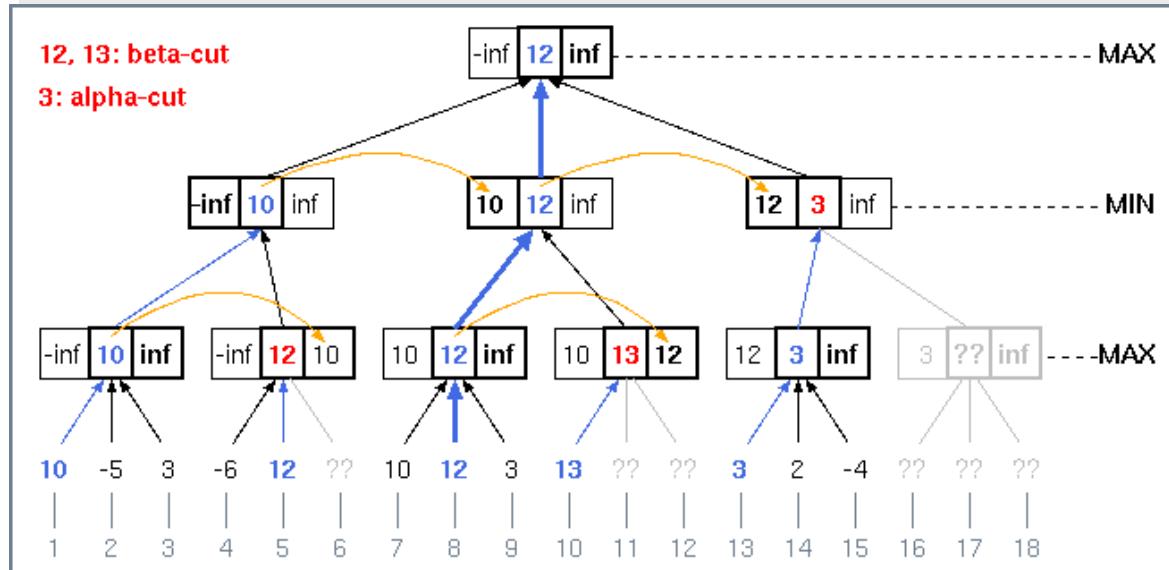
> Pseudocode (NegaMax-Variante):  
Initialisiere Spielposition p;  
**minimax(p, tiefe)**  
    initialisiere bester\_wert = -unendlich;  
    **if** p gewonnen **return** bester\_wert;  
    **if** tiefe = 0 **return** bewerte p;  
    z[] = Liste an möglichen Zügen;  
    **do**  
        führe Zug z[i] aus;  
        wert = -minimax(p, tiefe - 1);  
        mache Zug z[i] rückgängig;  
        bester\_wert = max(wert, bester\_wert);  
    **while** noch Züge verfügbar;  
    **return** bester\_wert;

## Minimax (VI)

> Optimierungen:  
    » Variable Suchtiefe  
    » Dynamische Suchtiefe  
    » Alpha-Beta-Suche

## Alpha-Beta-Suche

> Suchbaum:



## Alpha-Beta-Suche (II)

> Pseudocode (NegaMax-Variante):

Initialisiere Spielposition p;

```
alphabeta(p, tiefe, alpha, beta)
if p gewonnen return bester_wert;
if tiefe = 0 return bewerte p;
z[] = Liste an möglichen Zügen;
do
    führe Zug z[i] aus;
    wert = -alphabeta(p, tiefe - 1, -beta, -alpha);
    mache Zug z[i] rückgängig;
    if wert >= beta return wert;
    if wert > alpha alpha = wert;
while noch Züge verfügbar;
return alpha;
```

### Alpha-Beta-Suche (III)

> Vergleich: Schach mit vier Halbzügen

Algorithmus	#Bewertungen	#Cutoffs	%Cutoffs	Rechenzeit
Minimax	28018531	0	0.00	134.87s
AlphaBeta	2005246	136478	91.5009	9.88s
AlphaBeta + Zugsortierung	128307	27025	99.2765	0.99s

### Textsuche

## Textsuche

- > Problem der Worterkennung:
  - » Auffinden einer Zeichenkette (Muster) in einem Text
- > Muster und Text basieren auf einem gemeinsamen Alphabet
- > Effizienz: Anzahl elementarer Zeichenvergleiche

## Textsuche (II)

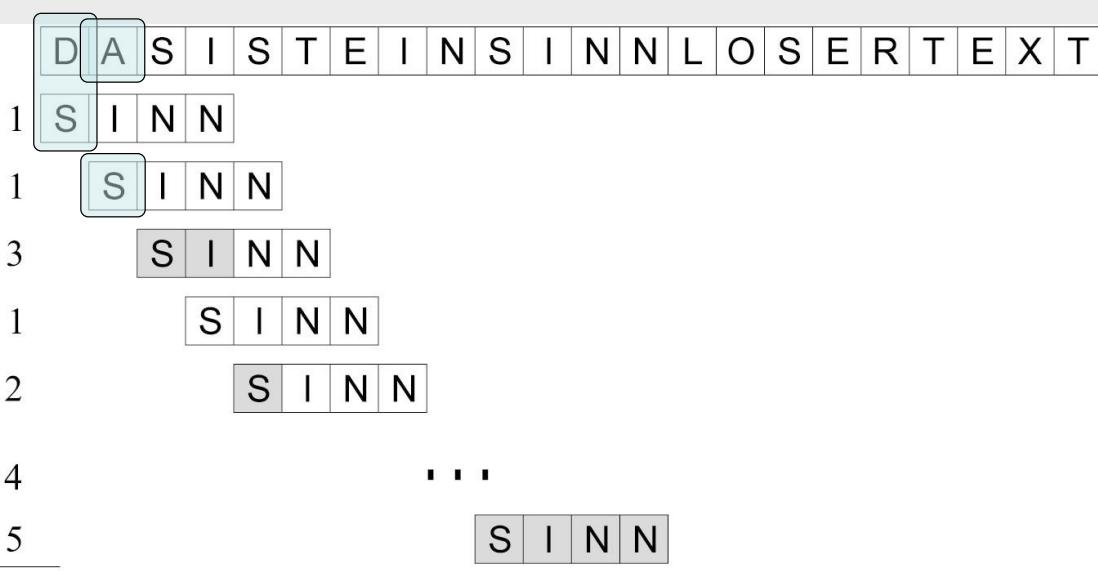
Naiver Ansatz (brute force):

- > An jeder Position im Text wird das Muster verglichen:
  - $\text{text}[0 \dots n-1]$  → zu durchsuchender Text
  - $\text{pat}[0 \dots m-1]$  → zu suchendes Muster (pattern)
- > Für jedes  $i$  im text prüfe ob:

```
pat = text[i ... i+m-1]
```

**Brute Force****Algorithmus BruteForce(text,pat)**

```
for i = 0 to n-m do
    j = 0;
    while j < m and pat[j]=text[i+j] do
        j = j + 1;
    od
    if j ≥ m then return i; fi
od
return -1;
```

**Brute Force (II)**

### Brute Force (III)

- > Laufzeitkomplexität im ungünstigsten Fall:
- >  $(n-m) * m$  elementare Zeichenvergleiche
- > → Aufwand:  $O(n*m)$

### Knuth-Morris-Pratt

- > Verbesserung von Brute Force
- > Gelesene Information wird bei Nichtübereinstimmung genutzt
- > Analyse des Musters
- > Zeiger wird nicht zurückgesetzt
- > Aufwand:  $O(n+m)$

## Knuth-Morris-Pratt (II)

A	B	C	A	B	A	B	A	B	C	A	A	B	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	A	A	B
---	---	---	---	---	---

1    2    3    4    5

A	B	C	A	A	B
---	---	---	---	---	---

6    7

A	B	C	A	A	B
---	---	---	---	---	---

8    9    10

A	B	C	A	A	B
---	---	---	---	---	---

11    12    13    14    15    16

## Knuth-Morris-Pratt (III)

**Algorithmus KMP(text,pat)**

```
j=0;
for i = 0 to n-1 do
    while j ≥ 0 and pat[j] ≠ text[i] do
        j = next[j];
    od
    j = j +1;
    if j = m then return i-m+1; fi
od
return -1;
```

## Knuth-Morris-Pratt (IV)

j		next[j]
	A B C A A B	
1	A B C A A B	0
	A B C A A B	
2	A B C A A B	0
	A B C A A B	
3	A B C A A B	0
	A B C A A B	
4	A B C A A B	1
	A B C A A B	
5	A B C A A B	1

Erstellt von: Jürgen Fahy

10.02.2017

Seite 359

## KMP – Beispiel

> Text: abcda~~b~~cxyz

> Muster: abcda**b**c**e**f

a			abcd		1	5
a	0	1	abcd	abcda		
ab			abcd	ab	2	6
ab	0	2	abcd	abcdab		
abc			abcd	abc	3	7
abc	0	3	abcd	abcabc		
abcd			abcd	abcdabc	3	8
abcd	0	4	abcd	abcdabce		
			abcd	abcdabce	0	

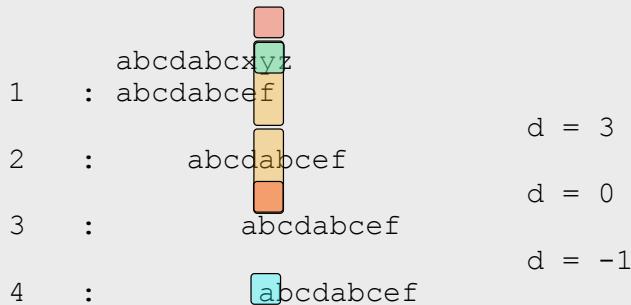
Erstellt von: Jürgen Fahy

10.02.2017

Seite 360

## KMP – Beispiel (II)

> Text: abcdabcxyz, Muster: abcdabcef



## Boyer-Moore

- > Muster wird von **rechts** nach **links** verglichen
- > Wenn bereits das erste Zeichen nicht stimmt, und das Zeichen im Text nicht im Muster vorkommt, kann das Muster um  $m$  Zeichen nach rechts verschoben werden

**Boyer-Moore (II)**

A	C	B	B	A	<b>D</b>	A	B	C	A	B	A	...
---	---	---	---	---	----------	---	---	---	---	---	---	-----

A	B	C	A	A	<b>B</b>
---	---	---	---	---	----------

A	B	C	A	A	B
---	---	---	---	---	---

A	C	B	B	A	<b>C</b>	A	B	C	A	B	...
---	---	---	---	---	----------	---	---	---	---	---	-----

A	B	C	A	A	<b>B</b>
---	---	---	---	---	----------

A	B	C	A	A	B
---	---	---	---	---	---

**Boyer-Moore (III)**

2 Strategien:

- > bad-character (occurrence) Heuristik -Bereits das erste Zeichen stimmt nicht
- > good suffix (match) Heuristik -Übereinstimmung eines Teils des Musters
  - 3 Fälle

### Boyer-Moore (IV)

C	B	A	<b>B</b>	B	C	B	B	C	A	B	A	...
---	---	---	----------	---	---	---	---	---	---	---	---	-----

A	B	B	<b>C</b>	B	C
---	---	---	----------	---	---

A	B	B	C	B	C
---	---	---	---	---	---

B	A	A	<b>B</b>	B	C	A	B	C	A	B	A	...
---	---	---	----------	---	---	---	---	---	---	---	---	-----

B	C	A	<b>A</b>	B	C
---	---	---	----------	---	---

B	C	A	A	B	C
---	---	---	---	---	---

C	B	A	<b>B</b>	B	C	B	B	C	A	B	A	...
---	---	---	----------	---	---	---	---	---	---	---	---	-----

A	B	B	<b>A</b>	B	C
---	---	---	----------	---	---

A	B	B	A	B	C
---	---	---	---	---	---

### Boyer-Moore (V)

- *last* Tabelle  
Enthält für jedes Zeichen des Alphabets eine Verschiebedistanz:  
 $\text{last}[c] = \max\{j \mid p_j=c\}$
- *shift* Tabelle  
Gibt für jedes Suffix des Musters eine Verschiebedistanz an

## Boyer-Moore (VI)

*shift* Tabelle:

> Shift Condition:

$C_s(i, s)$ : für jedes  $k$  mit  $i < k < m$  gilt:  
 $s \geq k \vee pat[k-s] = pat[k]$

> Occurrence Condition:

$C_o(i, s)$ : wenn  $s < i$  dann gilt:  
 $pat[i-s] \neq pat[i]$   
 $\rightarrow shift[i+1] = \min\{s > 0 \mid C_s(i, s) \text{ und } C_o(i, s)\}$

## Boyer-Moore (VII)

A	A	B	B	A	C	C	B	A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---

shift=1

A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---

shift=4

A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---

last=2

A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---

shift=4

A	C	B	A	B	C	B	A
---	---	---	---	---	---	---	---

## Boyer-Moore (VIII)

Algorithmus BMSearch(text, pat)

```
i=0;  
while i ≤ n-m do  
    j = m-1;  
    while j ≥ 0 and pat[j] ≠ text[i+j] do  
        j = j -1;  
    od  
    if j < 0 then  
        return i;  
    else  
        i = i + max(shift[j+1], j-last[ text[i+j] ]);  
    fi  
    od  
return -1;
```

## Pattern Matching

> Suche nach allgemeinen Mustern (wildcards)

\*, +, ...

> Notation zur Definition eines Musters

» Reguläre Ausdrücke (regular expressions)

## Regular Expressions

Symbole eines Alphabets  $\Sigma$  definieren

Operationen:

Verkettung:

- » Definiert Folgen von Zeichen
- » z.B.: AB bedeutet Zeichen A gefolgt von B
  - ABA → A dann B dann A

## Regular Expressions (II)

> Oder Veknüpfung:

- » Definiert Alternativen
  - A+B bedeutet entweder A oder B

> Hüllenbildung:

- » Beliebige Wiederholung von Teilmustern
  - AB\* bedeutet A gefolgt von 0 oder mehr B
  - (ABC)\* → beliebige Folge ( $\geq 0$ ) der Zeichenkette ABC

## Regular Expressions (III)

Beispiele:

- $AB^*(CD)^*$  passt auf folgende Texte:  
ABCD, A, ACD, ABBB, ABBBCDCDCD, ...
- $(A+B)CD(EF)^*$ : ACD, BCD, ACDEFEF, BCDEF, BCDEFEF, ...
- $(A+B)^*C^*E$ : ABABE, E, AAAACCCE, ACE, ABBAACCE, ...

## Endliche Automaten

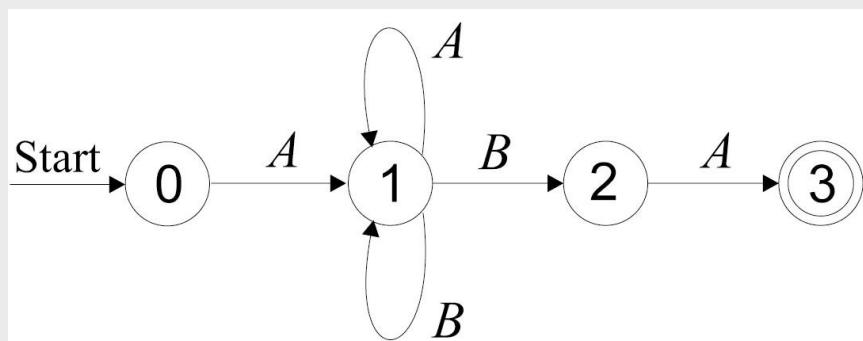
- > Zur Überprüfung regulärer Ausdrücke
- > Besteht aus:
  - » Einer endlichen Zustandsmenge  $S$
  - » Einem Alphabet  $\Sigma$  von Eingabewerten
  - » Einem Startzustand  $s_0$
  - » Einer Menge  $F$  von Endzuständen
  - » Einer Übergangsfunktion  $move$

## Endliche Automaten (II)

- > Automat lässt sich als gerichteter Graph darstellen
  - » Knoten bezeichnen Zustände
  - » Kanten repräsentieren Übergangsfunktionen
- > Übergangsfunktion liefert eine Menge von Zuständen → Nichtdeterministischer Endlicher Automat (NEA)

## NEA

Beispiel: A (A+B) \* BA

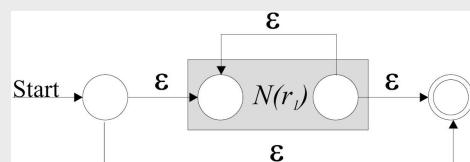
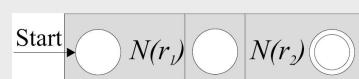
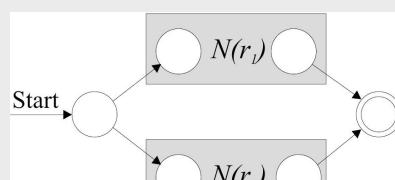
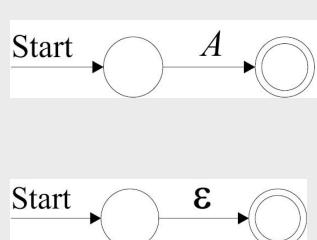


## NEA (II)

Beispiel:  $A (A+B)^* BA$

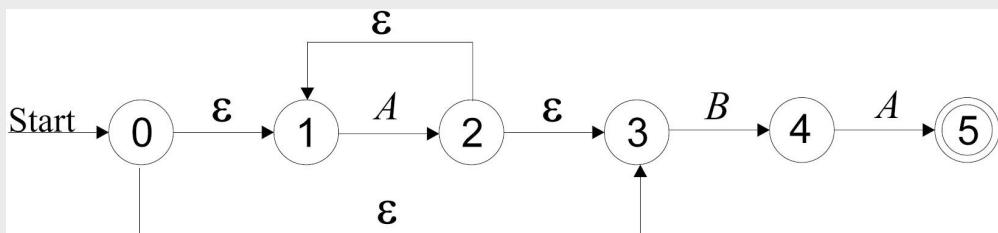
- > Zustandsmenge:  $S = \{0, \dots, 3\}$
- > Alphabet  $\Sigma = \{A, B\}$
- > Startzustand 0, Endzustand 3

## NEA (III)



## NEA (IV)

Beispiel: A\*BA



## Simulation eines NEA

- > Alle Zustände müssen gesichert und später abgearbeitet werden (Ähnlich BFS bei Graphen).
- > Implementiert in UNIX Kommandos:
  - » shell,
  - » sed
  - » grep, ...

## Reactive Extensions / Programming

## Reactive Programming

- > Reactive programming is an **asynchronous** programming paradigm concerned with **data streams and the propagation of change**
- > Reactive Programming is **Data Flow oriented**
- > e.g. Spreadsheets
- > e.g.  $c = a + b$
- > Implementation:
  - » as Graph
  - » Change Propagation:
    - pull
    - push
    - hybrid push

## Reactive Extensions (Rx)

- > Reactive Extensions (also known as ReactiveX) is a set of tools allowing imperative programming languages to operate on sequences of data regardless of whether the data is synchronous or asynchronous.
- > <http://reactivex.io>
- > ReactiveX is a combination of:
  - » Observer pattern
  - » Iterator pattern
  - » functional programming

## Operators

- > Categories:
  - » Creating Observables
  - » Transforming Observables
  - » Filtering Observables
  - » Combining Observables
  - » Error Handling Operators
  - » Observable Utility Operators
  - » Conditional and Boolean Operators
  - » Mathematical and Aggregate Operators
  - » Connectable Observable Operators