Algorithmen und Datenstrukturen

Teil 1: Elementare Datentypen und -strukturen

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 02/2020

Gliederung

- Grundlagen
- Speicherstrukturen

Elementare Datentypen

- In Programmiersprachen eingebaute Typen:
  - Ganzzahlige Typen, z.B. int, long, usw.
  - Fließkommazahlen, z.B. float, double, usw.
  - Boolesche Werte, z.B. boolean
  - Zeichen, z.B. char
  - Zeichenketten, z.B. String
  - Seltener: Datum

- Jeder Datentyp:
  - Wertebereich
  - Zulässige Operationen

Aufzählungen

- Datentyp mit endlichem Wertebereich, z.B. Ampelfarben
- Definition durch Auflistung
- Dabei in manchen Sprachen: Implizite Ordnung
- Beispiel Java:

```
enum Ampelfarbe {GRUEN, ROT, GELB}
```

- Möglichkeit Variablen des Typs Ampelfarbe anzulegen
- Vorteil: Typsicherheit

Datensätze

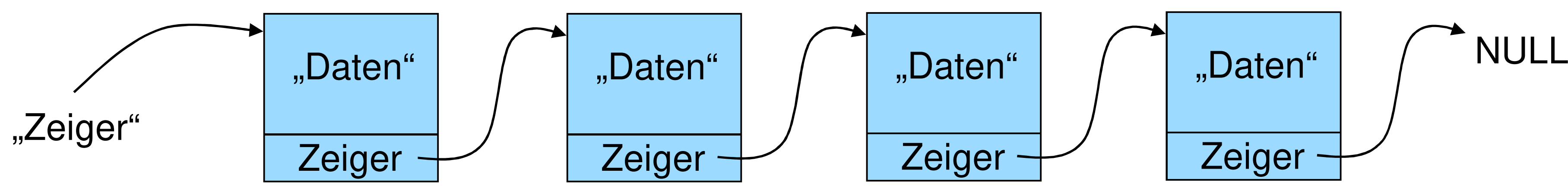
- Zusammenfassung von Elementen verschiedenen Typs, z.B. Beschreibung einer Person:
  - Ganzes Element heißt Datensatz
  - Einzelne Bestandteile Feld oder Attribut
- In imperativen/prozeduralen Sprachen, z.B.
  - Strukturen und Unions in C
  - Records in Pascal
- In objektorientierten Sprachen über Klassen

Feld und Liste

- Sammlung von Elementen gleichen Typs
- Feld:
  - Größe der Sammlung muss anfangs feststehen
  - Wahlfreier Zugriff auf einzelnes Element
- Liste:
  - Dynamische Größe
  - Zugriff auf Element durch Navigation
- Realisierung in Programmiersprachen:
  - Feld eigenes Konstrukt mit Angabe der Größe, z.B. in Java/C `int myArray[100]`
  - Liste konventionell über Zeiger
  - Geht aber auch in Java

Dynamische Datenstrukturen (I)

- Neben „normalen“ Variablen gibt es in den meisten Programmier-sprachen auch Zeiger
- Diese enthalten keine „richtigen“ Werte, sondern eine Adresse des Speichers des Rechners („Sie zeigen darauf.“)
- Neben dem Zeigen auf eine Adresse, die einen Datenwert enthält, kann ein Zeiger auch auf den speziellen Wert NULL (oder NIL) verweisen
- Eine dynamische Datenstruktur hat folgenden Aufbau:
  - Einen Zeiger als Startpunkt (Anker)
  - Eine Reihe von Datenobjekten, jeweils aus den eigentlichen Datenwerten und einem weiteren Zeiger bestehend
  - Dieser Zeiger verweist auf das nächste Datenobjekt
  - Der Zeiger des letzten Datenobjekts zeigt auf NULL
- Darstellung:



Dynamische Datenstrukturen (II)

- Diese Struktur nennt man (einfach verkettete) Liste
- Viele weitere Varianten werden verwendet:
  - Doppelt verkettete Liste
  - Ringförmige Liste
  - ...
- Umsetzung in Programmiersprachen:
  - In einigen Sprachen (z.B. C) gibt es speziell gekennzeichnete Zeigervariablen
  - In Java gibt es dies nur implizit (Details siehe VL Programmierung)
  - In einigen (vor allem älteren Sprachen) wie COBOL gibt es gar keine Zeiger

Realisierung verkettete Liste in Java (I)

- Dynamische Listen flexibles Konstrukt
- Was wird benötigt?
  - Zelle: Besteht aus Objekt und Zeiger auf folgende Struktur (auch wieder Objekt und Zeiger)
- Frage: Wie wird dies in Java realisiert, da die Sprache keine Zeiger kennt?
- Zelle aus Objekt und Zeiger:

```
(1) public class Cell<T>{
(2)    T data;
(3)    Cell<T> next;
(4)
(5)    public Cell(T o) {data = o;}
(6)  }
```

- Verwendung von Klasse `Cell` als innere Klasse zum Aufbau verketteter Liste

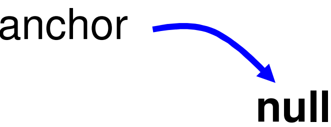
Realisierung verkettete Liste in Java (II)

```
( 1) public class VerketteteListe<T> {
( 2)
( 3)   private class Cell<T>{
( 4)          ...
( 5)   }
( 6)
( 7)   private Cell<T> anchor; // Points to first element
( 8)
( 9)
(10)   public void add(T o){
(11)     Cell<T> neu = new Cell<T>(o);
(12)     if (anchor == null){     // Empty List
(13)       anchor    = neu;
(14)       neu.next = null;
(15)     }
(16)     else{                    // Non-Empty List
(17)       neu.next = anchor;
(18)       anchor    = neu;
(19)     }
(20)   }
(21) }
```

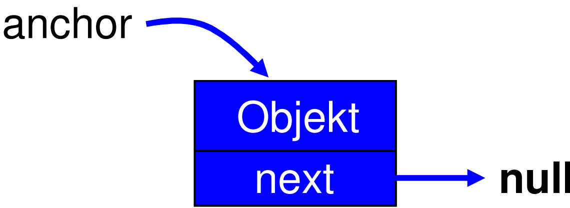
Realisierung verkettete Liste in Java (III)

- Was geschieht nun in der Methode `add()`?
  - Anlegen neue Zelle:

  - Dann Fallunterscheidung
  - Fall 1: Liste ist leer (d.h. anchor zeigt auf `null`)

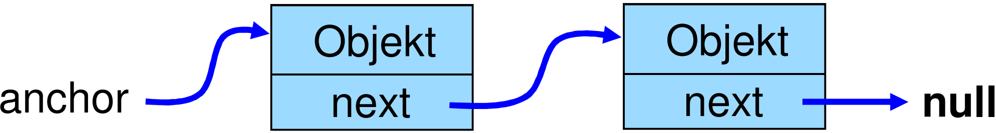


  - Anker zeigt auf neue Zelle, neue Zelle auf `null`:

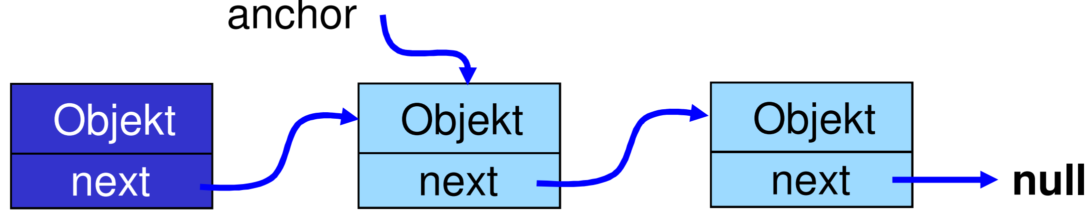


Realisierung verkettete Liste in Java (IV)

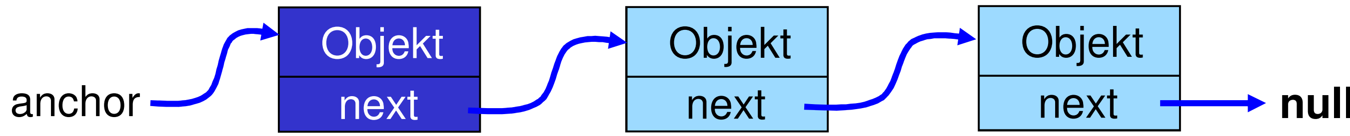
- Fall 2: Liste nicht leer (d.h. anchor zeigt nicht auf `null`)



  - Neue Zelle zeigt auf (alten) Anker



  - Anker zeigt auf neues Element



- Analog: Lesen, Aktualisieren und Löschen von Elementen

Abstrakter Datentyp (ADT) (I)

- ADT besitzt Namen und Operationen
- Keine Angaben über Interna, d.h. Implementierung
- Spezifikation:
  - Signatur: Angabe der Operationen (Name, Parameter, Rückgabewert)
  - Semantik: Angabe von Voraussetzung, Effekt und Ergebnis
- Beispiel: Kollektion von Objekten
- Signatur:

Kollektion<T>
`add(T t)`
`delete(T t)`
`boolean isEmpty()`
`boolean isElement(T t)`

Abstrakter Datentyp (ADT) (II)

- Semantik:
  - `add(T t)`
    - Effekt: Element t wird Kollektion hinzugefügt
    - Ergebnis: Kollektion nach Hinzufügen des Elements t
  - `delete(T t)`
    - Effekt: Sofern vorhanden wird Element t wird aus Kollektion entfernt
    - Ergebnis: Kollektion nach Entfernen des Elements t
  - `isEmpty()`
    - Ergebnis: Wahr, wenn Kollektion keine Elemente enthält, andernfalls falsch
  - `isElement(T t)`
    - Ergebnis: Wahr, wenn Element t in Kollektion enthalten, andernfalls falsch

Abstrakter Datentyp (ADT) (III)

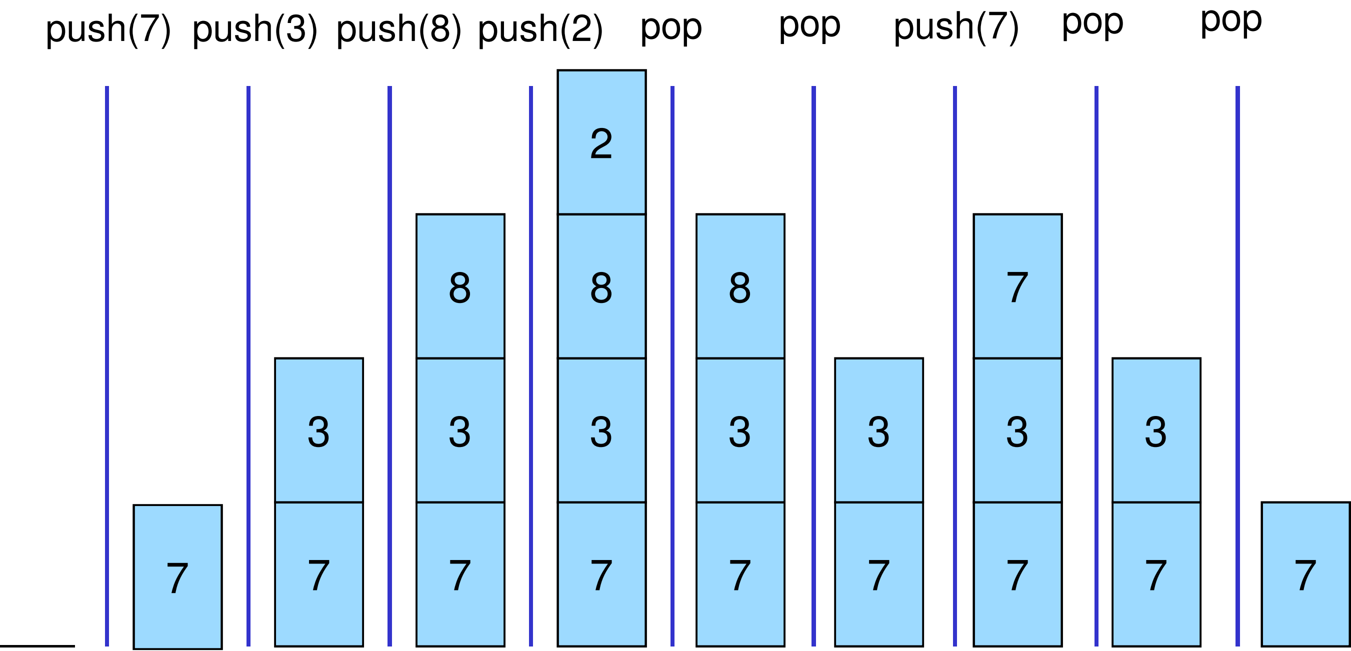
- Eigenschaften:
  - Kapselung: Anwender weiß, was ADT macht, aber nicht wie
  - Präzise Beschreibung: Schnittstelle zwischen Anwendung und Implementierung eindeutig
  - Einfachheit: Innere Implementierung interessiert nicht
  - Integrität: Anwender kann innere Struktur nicht manipulieren
  - Modularität: Übersichtliches, sicheres Entwickeln, leichter Austausch von Programmteilen

Übersicht

- Grundlagen
- Speicherstrukturen

Stapel (I)

- Speicherstruktur für gleichartige Objekte
  - Nur zuletzt gespeichertes Element wird als erstes wieder entnommen
  - LIFO-Prinzip: Last In, First Out
  - Ablegen: push-Operation
  - Entnehmen: pop-Operation
- Beispiel: Stapel mit Integer-Werten



Stapel (II): ADT

Stack<T>

`push(T t)`
  Effekt: Element t wird auf Stapel gelegt
  Ergebnis: Stapel nach Einfügen von Element t

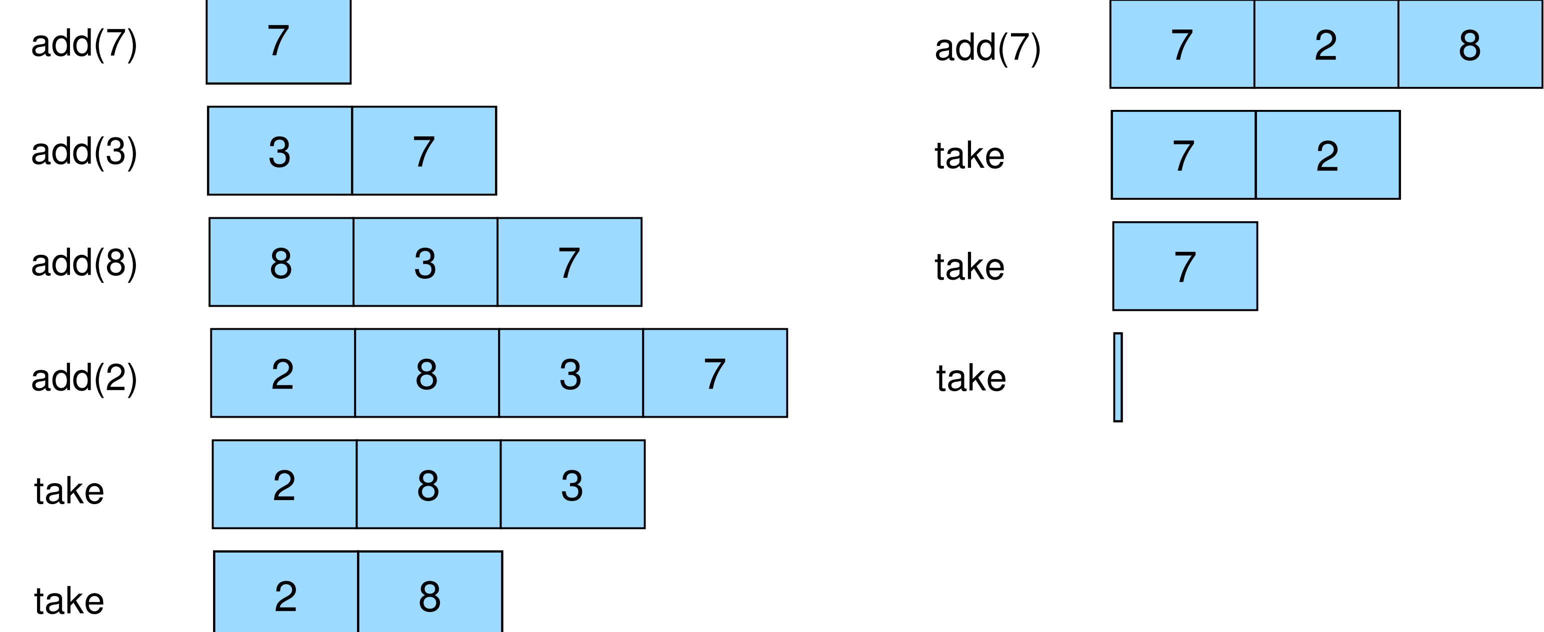
`T pop()`
  Voraussetzung: Stapel nicht leer
  Effekt: Element t wird vom Stapel entnommen
  Ergebnis: Stapel nach Entnehmen von Element t

`T top()`
  Voraussetzung: Stapel nicht leer
  Ergebnis: Stapel wie vorher

`boolean isEmpty()`   Ergebnis: Wahr, wenn Stapel leer, falsch andernfalls

Schlange (I)

- Speicherstruktur für gleichartige Objekte
  - Zuerst gespeichertes Element wird als erstes wieder entnommen
  - FIFO-Prinzip: First In, First Out
  - Ablegen: enqueue-/add-Operation
  - Entnehmen: dequeue-/take-Operation
- Beispiel: Schlange mit Integer-Werten



Schlange (II): ADT

Queue<T>

```
add(T t)
  Effekt: Element t wird Schlange hinzugefügt
  Ergebnis: Schlange nach Hinzufügen von Element t
```

```
T take()
  Voraussetzung: Schlange nicht leer
  Effekt: Element t wird aus Schlange entnommen
  Ergebnis: Schlange nach Entnehmen von Element t
```

```
T head()
  Voraussetzung: Schlange nicht leer
  Ergebnis: Schlange wie vorher
```

```
boolean isEmpty()
  Ergebnis: Wahr, wenn Schlange leer, falsch andernfalls
```

Vorrangwarteschlange (I)

- Erweiterte Form der Schlange:
  - Elementen in Schlange wird Priorität mitgegeben
  - Diese regelt die Reihenfolge der Abarbeitung
  - Vereinbarung: Kleiner Wert = Hohe Priorität
- ADT:

PriorityQueue<T>

```
add(T t, long p)
  Effekt: Element t mit Priorität p wird Schlange
     hinzugefügt
  Ergebnis: Schlange nach Hinzufügen von Element t
```

```
T takeMin()
  Voraussetzung: Schlange nicht leer
  Effekt: Element t mit kleinstem Wert (höchster
    Priorität) wird entnommen
  Ergebnis: Schlange nach Entnehmen von Element t
```

Vorrangwarteschlange (II)

- ADT:

PriorityQueue<T>

```
T ReadMin()
  Effekt: Liefert Wert mit höchster Priorität
  Voraussetzung: Schlange nicht leer
  Ergebnis: Schlange wie vorher
```

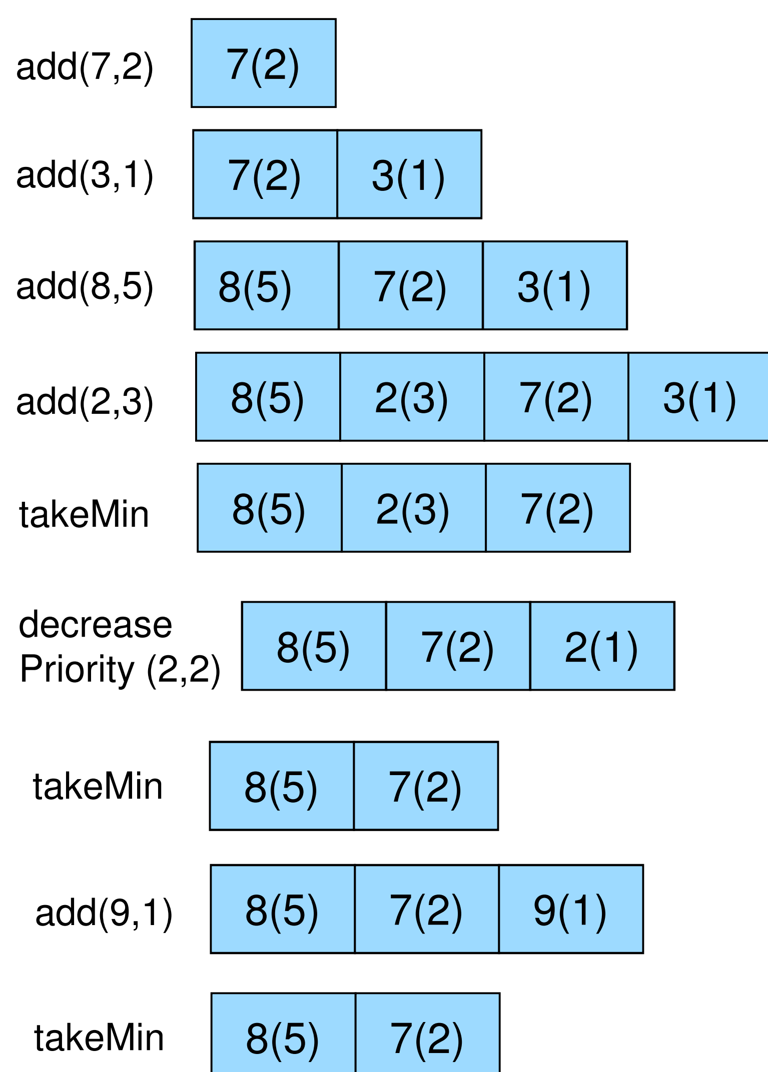
```
boolean isEmpty()
  Ergebnis: Wahr, wenn Schlange leer, falsch andernfalls
```

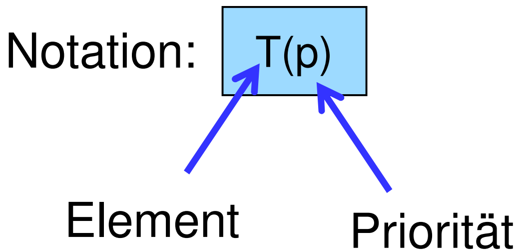
```
decreaseKey (T t)
  Voraussetzung: Schlange nicht leer
  Ergebnis: Priorität des Elements t wird (um 1) gesenkt
```

```
decreaseKey (T t, long l)
  Voraussetzung: Schlange nicht leer
  Ergebnis: Priorität des Elements t wird (um l) gesenkt
```

Vorrangwarteschlange (III)

- Beispiele:





Vorrangwarteschlange (IV)

- Implementierung:
  - Feld oder verkettete Liste „unhandlich“
  - Daher: Nutzung von Baumstrukturen (kommen später in VL)
  - Beispiele: Heaps, AVL-Bäume

Container

- Synonym: Sammlung, Kollektion, Collection
- Feld und Liste nicht einzige Realisierungsmöglichkeiten
- Klassenbibliotheken bieten meistens Vielzahl von Containern an
- Unterscheiden sich hinsichtlich:
  - Speicherplatzverbrauch
  - Effizientes Einfügen
  - Effizientes Lesen
  - Ordnung
  - Duplikate (Menge/Multimenge)
  - Thread-Sicherheit (Java)

Notation

- Klasse: Name
- Abstrakte Klasse: <<abstract>> Name
- Interface: <<interface>> Name
- Markierung aus Rahmen:

- Vererbung (extends):

- Implementierung (implements): - - - - - - - - - - - - -

Paket `java.util` (I): Rahmen (Java 13)

Paket `java.util` (II): Maps (Java 13)

*
Object
<<abstract>>
AbstractMap<K,V>
EnumMap
<K extends Enum<K>,V>
TreeMap<K,V>
HashMap<K,V>
Weak
HashMap<K,V>
Identity
HashMap<K,V>
Linked
HashMap<K,V>
<<interface>>
Map<K,V>
<<interface>>
SortedMap<K,V>
Hashtable<K,V>
<<interface>>
NavigableMap<K,V>
<<abstract>>
Dictionary

Paket `java.util` (III): Mengen (Java 13)

Paket `java.util` (IV): Listen (Java 13)

<<abstract>>
AbstractCollection<E> *

<<abstract>>
AbstractList<E>

ArrayList<E>

Vector<E>

<<abstract>>
AbstractSequentialList<E>

<<interface>>
RandomAccess

Stack<E>

LinkedList<E>

<<interface>>
List *

Paket `java.util` (V): Queues (Java 13)

ArrayDeque<E>

<<abstract>>
AbstractCollection<E>

*

<<abstract>>
AbstractQueue<E>

PriorityQueue<E>

<<interface>>
Collection

*

<<interface>>
Queue

<<interface>>
Deque

Paket `java.util` (VI)

- Grundsätzlich gibt es folgende Typen von Containern:
  - Vom Interface `Collection` und dem Interface `Set` werden Kollektionen als Mengen (d.h. ohne Duplikate) abgeleitet, im einzelnen:
    - `HashSet, LinkedHashSet`
    - `TreeSet`
    - `EnumSet`
  - Vom Interface `Collection` und dem Interface `List` werden Kollektionen als Listen (d.h. mit geordneten Elementen) abgeleitet, im einzelnen:
    - `LinkedList`
    - `ArrayList`
    - `Vector/Stack`
  - Vom Interface `Map` werden Kollektionen als Abbildungen zwischen Objekten abgeleitet, im einzelnen
    - `HashMap, LinkedHashMap`
    - `IdentityHashMap, WeakHashMap,`
    - `TreeMap, EnumMap`
    - `HashTable`

Paket `java.util` (VII)

- Vom Interface `Collection` und dem Interface `Queue` werden Schlangen abgeleitet, im einzelnen:
  - `PriorityQueue`
  - `ArrayDequeue`
- Iteratoren, Komparatoren

- Anmerkung:
  - Hier nur kurzer Überblick
  - Details:
    - Collections Framework Documentation (https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/doc-files/coll-index.html)
    - Collections Framework Tutorial (http://docs.oracle.com/javase/tutorial/collections/index.html)

Zusammenfassung

- Grundlagen:
  - Datentypen
  - Aufzählung
  - Datensatz
  - Feld und Liste
  - Dynamische Datenstrukturen
  - Verkettete Liste in Java
  - Abstrakter Datentyp (ADT)

- Speicherstrukturen:
  - Container
  - Stapel
  - Schlange
  - Vorrangwarteschlange
  - Beispiel: Bibliothek in Java

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
B B T Q J R E S J F E D F U D S A H N A
S S M N D H F C M O A D F Y L I Y W Z B
M B C T Z N C U Z I F D F D E U S S E S
L W I M H D D J U L G D J I F V Y V Q T
F C N E R M F A B F R O R T D J V U N R
U E D Z G K Z B I L E P N R H F B H C A
Y U I F E W M E P Y I N X G C I D O F K
Y T Q J T T P V C E G N A L H C S P Z T
Z W C U S Q M O A I C I A U I Z F D A E
X L O I J F N X R N O A G J O S K L Y R
V K L A P T O D T B I Q G C Y S G K F D
S F U J A J G X M Z T V Y Z S J G N Y A
H Y O I A W X W T U X S U E K K N Z L T
G E N L A X I A O Q J M R D P C G K D E
K E O C P R X X T K D L T Y C Z B P U N
R K N B Y G E I F H F Q S R N Q K V V T
R M B A U I J G W G G U Z S T B E G U Y
G A Z N Z K L D I R T D B H G C F X G P
S V A E H F T M B E Y M W L G P Y J O R
T A P T A W K X I S Z O D P K G H Y G Q
```

Übungen (II)

- **Aufgabe 2**
  a) Welche elementaren Datentypen gibt es in Java?
  b) Was ist der Vorteil von Aufzählungen?
  c) Was ist ein ADT?
  d) Welche Vorteile hat ein ADT?
  e) Was ist der Unterschied zwischen Feld und Liste?
  f) Nennen Sie fünf Containertypen in Java!
  g) Was ist ein Stapel?
  h) Was ist eine Schlange?

Übungen (III)

- **Aufgabe 3**
  Überlegen Sie sich eine Datenstruktur zur Verwaltung von Produkten. Ein Produkt soll dabei eine eindeutige Kennung haben, einen Namen und einen Nettopreis. Weiterhin soll jedem Produkt eine Regalnummer zugeordnet werden, wobei hierfür eine Liste fester Werte bekannt ist.

- **Aufgabe 4**
  Skizzieren Sie die Implementierung Ihrer Datenstruktur aus Aufgabe 3 als Javacode!

Übungen (IV)

- **Aufgabe 5**
  Implementieren Sie eine generische Schlange in Java!
  Verwenden Sie dabei zunächst ein Feld mit der (Anfangs-)Dimension 100 als innere Datenstruktur!
  Speichern und Entnehmen Sie Objekte einer selbstdefinierten Klasse!
  Ändern Sie nun die innere Struktur in eine Liste um!

- **Aufgabe 6**
  Erweitern Sie die im Skript eingeführte Klasse `VerketteteListe` um eine Methode `printAll`, die alle Listenelemente nacheinander ausgibt, und um eine Methode `delete(T o)`, die in der Liste das erste Element, das gleich o ist, entfernt!

Übungen (V)

- **Aufgabe 7**

  Lesen Sie den zur Verfügung gestellten Artikel über Algorithmen und ihre Eigenschaften.

  Anschließend sollten Sie:

  - Wissen, was ein Algorithmus ist
  - Verschiedene Beschreibungsformen von Algorithmen nennen können
  - Wichtige Eigenschaften von Algorithmen nennen und erklären können
  - Den Unterschied zwischen Algorithmus und Programm erklären können

Algorithmen und Datenstrukturen

Teil 2: Komplexitätstheorie – Laufzeit von Algorithmen

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 04/2018

Gliederung

- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Motivation

- Bisher in VLen Programmierung und Informatik:
  - Algorithmen und Datenstrukturen kennen gelernt
  - Einige Probleme verschiedene Lösungen
  - Einige „besser“, andere „weniger gut“
  - Beispiel: Fibonacci-Zahlen iterativ vs. rekursiv
- Nun:
  - Algorithmen systematisch bewerten
  - Möglichst abstraktes Niveau
  - D.h. ohne Kenntnisse der konkreten Programmiersprache, der konkreten Hardware usw.

Methoden der Effizienzberechnung (I)

- Methode 1: Sowohl Rechner/Maschine als auch Eingabedaten festgelegt
  - Methode der Beurteilung: Zeitmessung
  - Problem: Abhängigkeit vom Rechner und den Eingabedaten, Aussage kann immer nur für konkreten Fall getroffen werden
- Methode 2: Rechner/Maschine abstrakt, Eingabedaten festgelegt
  - Methode der Beurteilung: Zählen von Operationen
  - Problem: Abhängigkeit von den Eingabedaten, Aussage kann immer nur für konkreten Fall getroffen werden
- Methode 3: Sowohl Rechner/Maschine als auch Eingabedaten abstrakt
  - Methode der Beurteilung: Berechnung (der Anzahl der Operationen)
  - Problem: Abstrakte Beschreibung ist zwar erreicht, aber Resultat ist immer noch vom Parameter der Eingabedaten abhängig

Methoden der Effizienzberechnung (II)

- Beispiel:
  - Zwei Algorithmen A_1 und A_2 berechnen das gleiche Problem
  - Sei dabei V Anzahl der Vergleiche und W Anzahl der Wertzuweisungen
  - A_1 benötigt $V_1=W_1=n \cdot \log(n)$ und A_2 benötigt $V_2=n^2$ und $W_2=n$
  - Welcher Algorithmus ist nun schneller?
  - Ohne Kenntnisse von n und Ausführungszeiten für Vergleich und Wertzuweisung können immer noch keine Aussagen getroffen werden
  - Dennoch: Wenn n genügend groß ist, dann wird auf jeden Fall A_1 schneller sein, weil sein Aufwand nur logarithmisch wächst, während sich bei A_2 dann der quadratische Anteil bemerkbar macht

- Methode 4: Rechner/Maschine abstrakt, Eingabedaten $\rightarrow \infty$
  - Methode der Beurteilung: Asymptotische Abschätzung

Komplexitätstheorie (I)

- Beschäftigt sich mit Maßen, die die Effizienz eines Algorithmus beschreiben
- Stellt Modelle zur Verfügung, um a priori Aussagen über die Laufzeit und den Speicheraufwand eines Algorithmus machen zu können
- Damit können Algorithmen verglichen werden, ohne konkret realisiert zu werden

- Komplexitätsmaß:
  - Ideal wäre als Maß eine Funktion, die jeder Eingabe die Zeit zuordnet, die zur Ausgabe benötigt wird
  - Damit wäre man aber wieder von der konkreten Umgebung (Rechner, Compiler, ...) abhängig
  - Abstraktion gelingt an dieser Stelle, indem keine konkrete Zeit, sondern die Anzahl der notwendigen Rechenschritte betrachtet wird
  - Rechenschritt: Arithmetische Grundoperation, Speicherzugriff, Vergleiche, Wertzuweisungen, ...

Komplexitätstheorie (II)

- Charakterisierung der Eingabe erfolgt durch Angabe der Länge (Zahl der zu verarbeitenden Zeichen)
- Beispiele:
  - Länge einer Zeichenkette, die verarbeitet werden muss
  - Anzahl der Knoten als Größe eines Graphen
  - Anzahl der zu sortierenden Elemente bei Sortieralgorithmen
- Wie schon bei der Zeitangabe erfolgt auch hier keine quantitative Aussage über die Länge eines Zeichens, diese muss lediglich konstant sein
- Sowohl bei Zeitangabe als auch bei der Eingabe bezieht man sich auf relative Größen
- Damit sind Aussagen der Form „Wenn die Länge der Eingabe proportional zu n ist, ist die Laufzeit des Algorithmus proportional zu n^2“ möglich

Komplexitätstheorie (III)

- Unterschieden werden kann die Komplexität für folgende Fälle:
  - Bester Fall (best case complexity): Minimale Laufzeit bzw. minimaler Speicherplatz für eine beliebige Eingabe der Länge n
  - Durchschnittlicher Fall (average case complexity): Durchschnittliche Laufzeit bzw. durchschnittlicher Speicherplatz für eine beliebige Eingabe der Länge n
  - Schlechtester Fall (worst case complexity): Maximale Laufzeit bzw. maximaler Speicherplatz für eine beliebige Eingabe der Länge n
- Meistens wird nur der letzte Fall betrachtet, weil
  - Nicht für alle Eingaben der Länge n muss ein Algorithmus gleichviel Zeit benötigen (siehe nächste Folie)
  - Berechnung der average case complexity ist für viele Algorithmen in der Praxis mathematisch sehr anspruchsvoll
    - Stichwort: Annahmen über Wahrscheinlichkeitsverteilungen
    - So ist es bis heute für viele Algorithmen nicht gelungen die average case complexity zu bestimmen
  - Für eine Reihe von Algorithmen sind average case und worst case complexity gleich

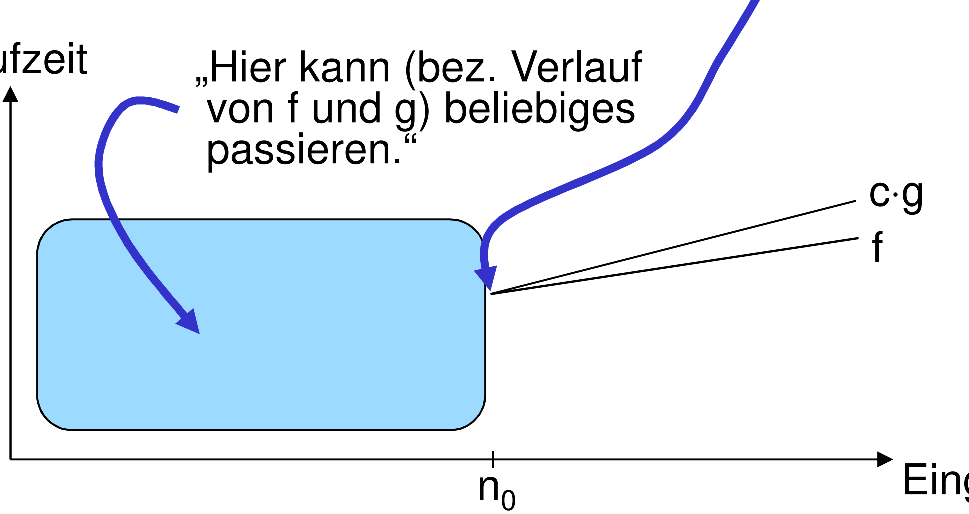
Einfluss der Eingabelänge auf Zeitkomplexität

- Warum muss ein Algorithmus nicht für alle Werte von n gleiche Zeitkomplexität haben?
- Sei für einen Algorithmus A der Aufwand $T(n) = 100 \cdot n + n^2 + 10^{n/1000-2}$
  - Für Werte kleiner als 100 dominiert der lineare Summand
  - Darüber dominiert zunächst der quadratische Summand
  - Erst ab Werten größer als 10000 kommt die exponentielle Komponente ins Spiel
- Ohne Kenntnis von T kann man den Aufwand nicht bestimmen
- Problem: Wie weit muss man messen?

Komplexitätstheorie (IV)

- Platzbedarf und Laufzeit werden als Funktion $f : N \rightarrow R^+$ angegeben
- Um die Bestimmung der Funktion zu erleichtern, verzichtet man auf die Angabe des genauen Wertes f(n) und gibt nur den qualitativen Verlauf, d.h. die Größenordnung an
- Dies geschieht mit dem Landauschen Symbol O
- Seien f und g Funktionen von N nach R^+.

  f hat die Ordnung von g, wenn es eine Konstante c > 0 und ein $n_0 \in N$ gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$
- Man schreibt auch: f(n) = O(g(n))
- Anschaulich:



Zeitkomplexitätsklassen

- Sei A ein Algorithmus mit der Eingabe der Länge n
- S(w) := Anzahl der Schritte/Operationen, die A bei einer Eingabe w benötigt, bis er anhält
- $s_A(n)$:= Maximale Anzahl an Schritten zur Verarbeitung einer Eingabe der Länge n
- Anmerkung: Max. Anzahl, weil worst case betrachtet werden soll
- Unterschieden werden drei zentrale Zeitkomplexitätsklassen:
  - A heißt linear-zeitbeschränkt, wenn $s_A \in O(n)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n$)
  - A heißt polynomial-zeitbeschränkt, wenn $s_A \in O(n^k)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n^k$)
  - A heißt exponentiell-zeitbeschränkt, wenn $s_A \in O(k^n)$
    (d.h. $\exists c > 0 \exists n_0 \in N_0 \forall n \geq n_0 : s_A(n) \leq c \cdot k^n$)
- Von besonderem Interesse sind die polynomial-zeitbeschränkten Algorithmen, weil in dieser Klasse viele praxisrelevante Probleme liegen und die Laufzeit dieser Algorithmen häufig noch vertretbar ist

Übersicht

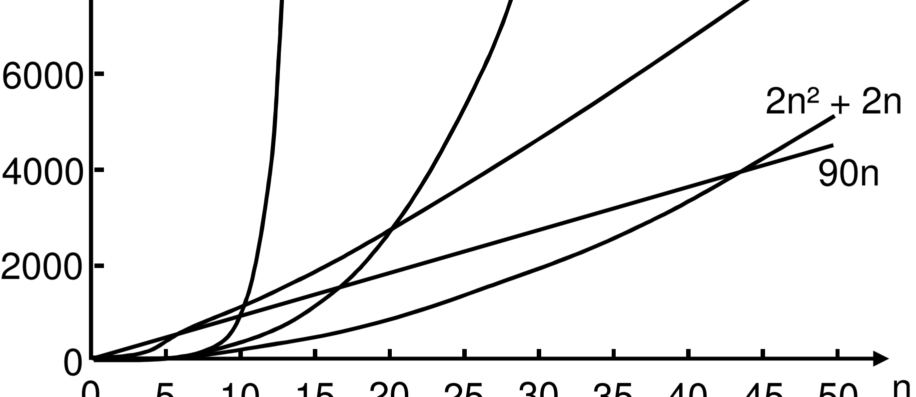
- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Beispiel

- Die Algorithmen A_1 bis A_5 lösen ein gegebenes Problem:

	A_1	A_2	A_3	A_4	A_5
Benötigte Zeiteinheiten	2^n	$n^3/3$	$31n \cdot \log_2 n$	$2n^2+2n$	$90n$
Komplexität	$O(2^n)$	$O(n^3)$	$O(n \cdot \log n)$	$O(n^2)$	$O(n)$

- Laufzeit der Algorithmen:



- Fazit: Obwohl A_5 die beste Komplexität aufweist, sind für kleine Werte für n andere Algorithmen effizienter

Komplexität der Fibonacci-Berechnung

- Iterative Lösung:
  - Schleife muss n-mal durchlaufen werden $\Rightarrow$ O(n)
- Rekursive Lösung:
  - Maß für Komplexität: Anzahl rekursiver Aufrufe
  - Entsprechen etwa Fibonacci-Zahl fib(n)
  - Damit: Komplexität entspricht fib(n)
  - Keine Beschränkung durch Polynom möglich
  - D.h. für jedes Polynom p existiert $n_0 \in N$, so dass fib(n) > p für $n \geq n_0$
  - Damit: Rekursive Fibonacci-Berechnung in Klasse der exponentiell-zeitbeschränkten Algorithmen
- Anmerkung:
  - Als weiterer Nachweis, dass die rekursive Lösung exponentiell-zeitbeschränkt ist, dient folgende Formel zur Berechnung der Fibonacci-Zahl:

$$\frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

  - Die exponentielle Laufzeit kann aus dem Auftreten von n im Exponenten geschlossen werden

Felder

- Gegeben: Feld mit Eingabegröße n

Problem	Komplexität
Speicherplatzverbrauch	n
Lesen des ersten Elements	1
Finde größtes Element	n
Vergleich „Jeder mit Jedem“	n*n
Wie oft ist Wert x im Feld?	n
Dreifache Summe aller Elmente	
Lösung 1: • Schleife zum Aufaddieren • Anschließend Multiplikation mit 3	n
Lösung 2: • Schleife • In der Schleife Element mit 3 multiplizieren • Summe bilden	n aber langsam
Lösung 3: • Drei Schleifen hintereinander • Aufaddieren	n aber sehr langsam

Matrizenmultiplikation (I)

- Zwei Matrizen lassen sich multiplizieren, wenn die Spaltenzahl der ersten der Zeilenzahl der zweiten entspricht
- Definition:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{pmatrix}$$

mit $c_{ik} := \sum_{j=1}^{n} a_{ij} \cdot b_{jk}$ für $i = 1, \dots, m$ und $k = 1, \dots, p$

$$\begin{pmatrix} & \star & \\ a_{i1} \cdots & a_{ij} & \cdots a_{in} \\ & \star & \end{pmatrix} \cdot \begin{pmatrix} & b_{1j} & \\ \star & \vdots \; b_{ij} \; \vdots & \star \\ & b_{nj} & \end{pmatrix} = \begin{pmatrix} * & \vdots & * \\ \cdots & a_{i1}b_{1j} + \ldots + a_{in}b_{nj} & \cdots \\ * & \vdots & * \end{pmatrix}$$

Spalte j

← Zeile i

[https://homepages.physik.uni-muenchen.de/~milq/spiele/iupages/matrix_m.html]

Matrizenmultiplikation (II)

- Implementierung:

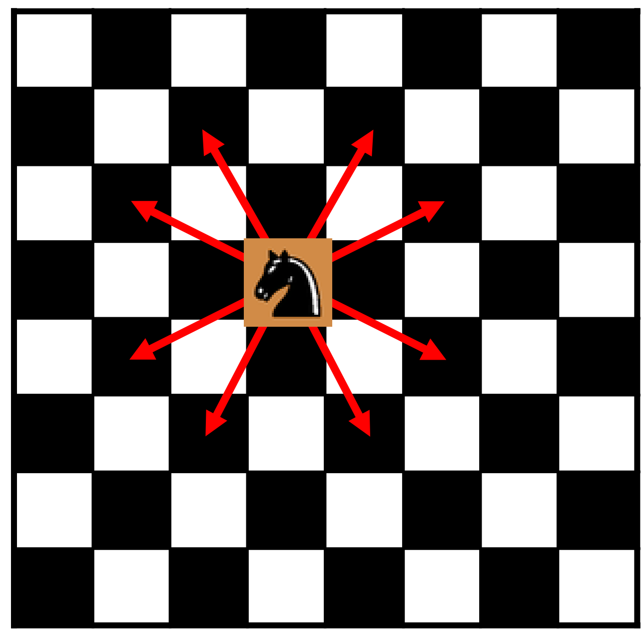
```
( 1) // Methode multiply zum Multiplizieren der Matrizen m1 und m2
( 2) // Rückgabe des Resultats in m3
( 3) public boolean multiply(Matrix m1, Matrix m2) throws
( 4)             MatrixIncompatibleForMultiplicationException{
( 5) int wert = 0;
( 6)   if (m1.spalten != m2.zeilen){
( 7)      throw new MatrixIncompatibleForMultiplicationException
( 8)                           ("Multiplikation nicht möglich");
( 9)   }
(10)   else{
(11)     for (int i=0;i<m1.zeilen;i++){
(12)       for (int j=0;j<m2.spalten;j++){
(13)         wert = 0;
(14)         for (int k=0;k<m1.spalten;k++){
(15)           wert += m1.getMatrix(i,k) * m2.getMatrix(k,j);
(16)         }
(17)         this.setMatrix(i,j,wert);
(18)       }
(19)     }
(20)     return true;
(21)   }
(22)}
```

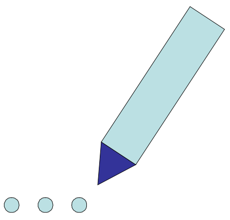
Matrizenmultiplikation (III)

- Komplexität:
  - Eingabegröße: n := max {n,m,p}
  - Laufzeit: $O(n^3)$

Springerproblem

- Springer auf Schachbrett (Größe nxn)
- Springer startet auf beliebigem Feld
- Jedes Feld auf Brett soll genau einmal besucht werden





- Lösungsidee?
- Komplexität?

Übersicht

- Komplexitätstheorie
- Beispiele zur Bewertung von Algorithmen
- Nicht-Determinismus, Sprachklassen P, NP

Nichtdeterminismus

- Algorithmus A heißt nichtdeterministisch, wenn A folgendes Sprachelement enthält: `Anweisung_1` ***OR*** `Anweisung_2`
- Es kann entweder mit `Anweisung_1` oder `Anweisung_2` fortgesetzt werden, Auswahl erfolgt dabei willkürlich, insbesondere unabhängig vom bisherigen Programmverlauf
- Ein nichtdeterministischer Algorithmus rät dabei stets die richtige Lösung
- Dieses Vorgehen ist natürlich fiktiv
- Mit diesem Mittel kann man beispielsweise den Weg aus einem Labyrinth in kürzester Zeit finden

Entscheidungs-/Optimierungsprobleme

- Entscheidungsproblem:
  - Gesucht ist Lösung ja/nein
  - Beispiele:
    - Ist gegebene Folge sortiert?
    - Ist gegebener Graph zusammenhängend?
    - Terminiert gegebenes Java-Programm?
- Optimierungsproblem:
  - Gesucht ist optimale Lösung (z.B. Maximum oder Minimum einer Funktion)
  - Definierte Randbedingungen legen Menge zulässiger Lösungen fest
  - Lösung des Optimierungsproblems ist zulässige Lösung, die gefordertes Optimierungskriterium bestmöglich erfüllt
  - Beispiele:
    - Färbe Knoten eines Graphen, so dass adjazente Knoten keine gemeinsame Farbe haben und Anzahl der Farben minimal ist
    - Finde in kantenmarkiertem Graphen Weg von Knoten k_1 zu Knoten k_2 mit minimaler Kantensumme

Problemklasse P

- Definition:

  Problemklasse P (P wie polynomial) umfasst alle Probleme, für die Algorithmus mit deterministisch polynomial-zeitbeschränkter Laufzeit existiert.
- Beispiele:
  - Kürzester Weg in Graphen
  - Sortieren einer Folge von Werten
- Beispiele für Probleme außerhalb von P:
  - Knotenfärbung eines Graphen
  - Rundreiseproblem
- Bedeutung der Problemklasse P:
  - In Praxis wichtig, da diese Algorithmen mit vertretbarem Aufwand zu Ergebnis kommen
  - Werden daher auch „Klasse der effizient lösbaren Probleme“ genannt

Problemklasse NP

- Definition:

  Problemklasse NP (NP wie nichtdeterministisch polynomial) umfasst alle Probleme, für die Algorithmus mit nicht-deterministisch polynomial-zeitbeschränkter Laufzeit existiert.

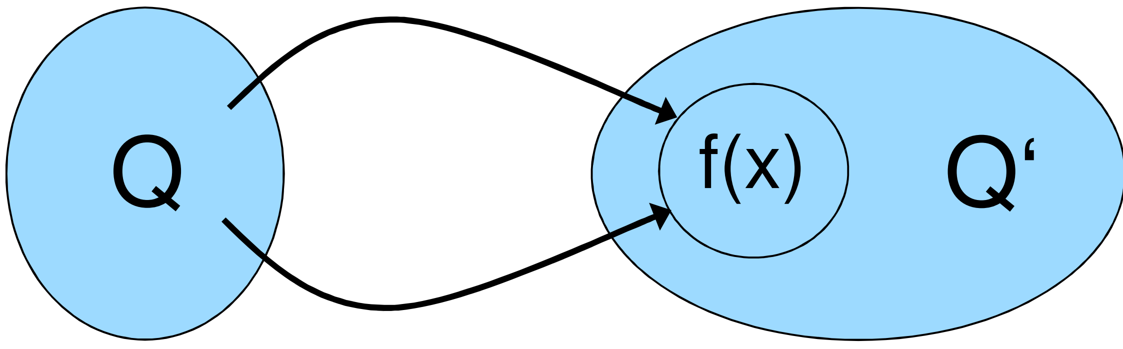
- Beziehung P zu NP:
  - P Teilmenge gleich NP
  - Frage, ob P = NP oder P ≠ NP offen
  - Bekannt als P=NP-Problem
  - DAS herausragende offene Problem der Theoretischen Informatik
  - Vermutung: P ≠ NP

Transformation von Problemen

- Definition:

  Problem Q heißt auf Problem Q‘ <u>transformierbar</u>, wenn eine Funktion f existiert, die Q auf Q‘ abbildet.

  In Zeichen: $Q \leq_f Q'$



- Ist f Polynom, spricht man von <u>polynomialer Transformierbarkeit</u>:

  $Q \leq_p Q'$ mit p Polynom

Problemklasse NP-schwer

- Definition:

  Problemklasse NP-schwer umfasst alle Probleme, auf die sich beliebiges Problem P‘ aus NP in Polynomialzeit transformieren lässt.

  Formal: P heißt NP-schwer g.d.w. $\forall$ P‘ $\in$ NP: P‘ $\leq_p$ P.

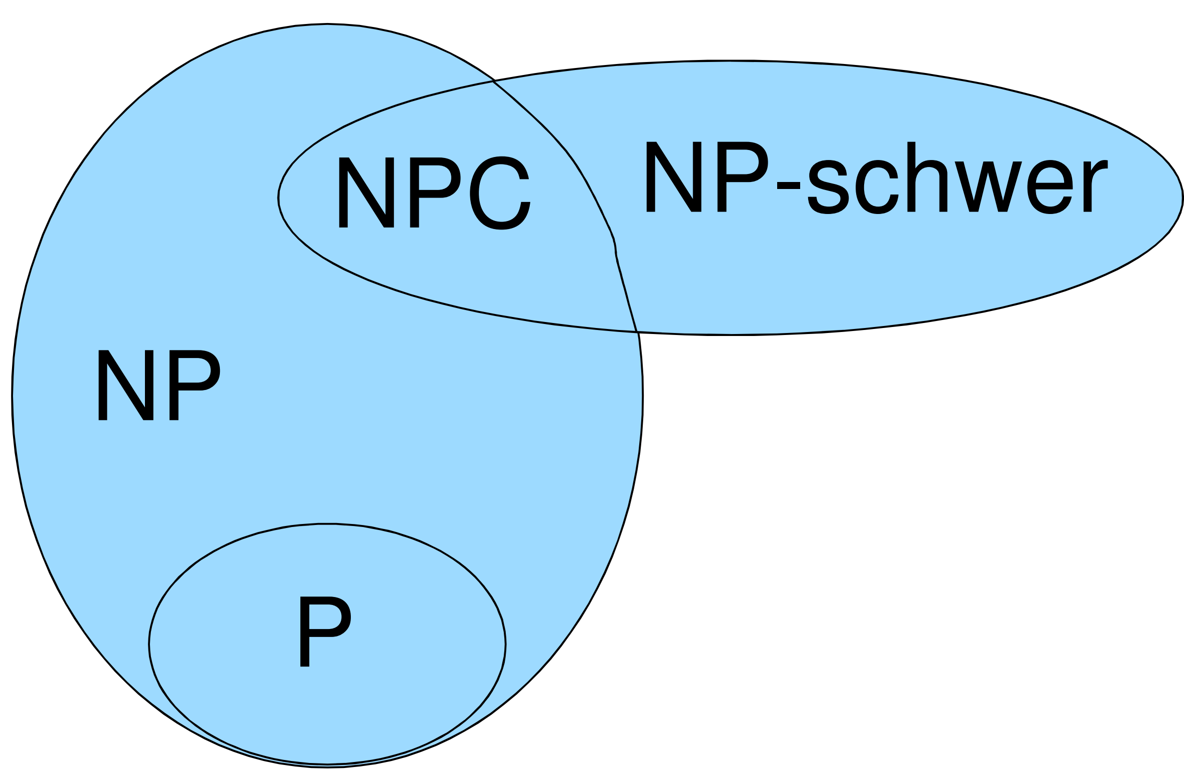
- Anmerkung:
  - In Literatur manchmal Bezeichnung NP-hart

Problemklasse NP-vollständig

- Definition:
  Problemklasse NPC (NP-vollständig, NP complete) umfasst alle Probleme, die selber Element von NP und NP-schwer sind.
  Formal: P heißt NP-vollständig g.d.w.
  (1) $P \in NP$
  und
  (2) $\forall P' \in NP: P' \leq_p P$ mit p Polynom.
- Anschaulich: NPC ist Klasse der schwierigsten Probleme innerhalb von NP
- Anmerkung:
  - NPC- Definition ist „rekursiv“
  - D.h. mindestens ein Problem notwendig, dessen NPC-Zugehörigkeit anders bewiesen wird
  - Erfüllbarkeitsproblem: Für Boolesche Funktion $f(x_1, \ldots, x_n)$ wird gefragt, ob Belegung von $x_1, \ldots, x_n$ existiert, so dass f wahr wird
  - Erfüllbarkeitsproblem ist in NPC (Satz von Cooke 1971)

Zusammenhang P, NP, NP-schwer, NPC

- Unter Annahme P ≠ NP gilt:



NP-vollständige Probleme

- Diverse NP-vollständige, praxisrelevante Probleme bekannt (z.B. viele aus Bereichen Transport und Logistik)
- Beispiele:
  - Rucksackproblem (Knapsack Problem)
  - Anschaulich: „Packe einen Rucksack so voll wie möglich“
  - Formaler: Gegeben sind n Werte a_i, und das Rucksackvolumen v, gesucht ist eine Indexmenge $I \in \{1,\ldots,n\}$ mit $\sum_{i \in I} a_i = c \le v$ und $v - c$ ist minimal
  - Problem des Handlungsreisenden (Traveling Salesman Problem)
  - Anschaulich: „Finde kürzeste Rundreise“
  - Formaler: Ein Handlungsreisender muss n Orte besuchen und will zum Ausgangsort zurückkehren, zwischen zwei Orten ist die Entfernung bekannt (Graph mit den Orten als Knoten, markierte Kanten geben die Entfernungen an)
    Gesucht ist ein geschlossener Weg im Graphen

Zusammenfassung

- Komplexitätstheorie:
  - Methoden der Bewertung
  - Komplexitätsklassen

- Beispiele zur Bewertung von Algorithmen:
  - Fibonacci-Zahlen
  - Felder
  - Matrizen
  - Springerproblem

- Nicht-Determinismus, Sprachklassen P, NP:
  - Nicht-Determinismus (Sprachelement ***OR***)
  - Klassen P und NP, P=NP-Problem
  - NP-Vollständigkeit

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
P Z Z W D C C Q P N N P J N T R D C N Q Q B B R C
I O K S A Q P X L T H V T S U H T Q I I T O Q E W
L Q L P M J X Z D G B L J C A O N F C D L M Z G Z
I G F Y A R D F H I V G K C P J W B H N N C M H I
P O J H N G D J K S M S L B K O V H T V I T H W J
K G O W I O H B S B A U W U U W C U D O D N B R U
F N Z S V L M B G C F T B R W C C L E D Z W I E A
L Y Q T G R E I K K R K M E H Z P L T Y T D O U E
F O M A E P Y P A L A A Z Q T F P O E A K C Y F V
W P O D B V R Q A L T I Y A D B B V R J R S P C W
H M R H M O T Q M H Z T Y C D H B N M N L B P M G
S P T Z B I O T M K T E G W T M Z H I X A O F P Z
S R P L V T C X N H A W I Q C I Z P N M O O H N Q
X O E Y U P P Z M A W S C T I V G H I K R S D G G
C M C M K I M G M Y B I D S B Z I F S M W J E N H
X N W H U O F K N K O R Z K R E D D M K M L E S T
S U Y X R V N N B A Y O J D E R S D U N I F K G Q
F Z G W C B B P V K S Q V F C S M C S K E F G D F
V Q Q X J J R P D Q I P Q B O N A E H S Z D V M W
G N O J W N I C S X L X Y R F U A C Q R B G Y B K
V U B T T I A B Z E E X Z S W X A Y T T A B Q F Q
P N Z D O L G O X V A V Q K I Y N H I S M E P Z Q
N O Z E Q O G I D N E A T S L L O V P N R L N T E
E C N Z T M C E X M J C E T I V F B G R Z O F K B
Z D N X H A S D O N P M U K N H W U Q H S X W U T
```

Übungen (II)

- **Aufgabe 2**
  a) Wie lässt sich die Komplexität eines Algorithmus prinzipiell messen?
  b) Was ist die wünschenswerte Methode?
  c) Was ist das Landausche Symbol?
  d) Was sind polynomial zeitbeschränkte Algorithmen?
  e) Was ist Nicht-Determinismus?
  f) Wie sind die Problemklassen P und NP definiert?
  g) Wann ist ein Problem/Algorithmus NP-vollständig?
  h) Gilt P=NP?

Übungen (III)

- **Aufgabe 3**

  Geben Sie zu den folgenden benötigten Zeiteinheiten jeweils den Komplexitätswert O an!

  a) $4n^2 + 7n + 5$
  b) $2^n + n^7$
  c) log(log n)
  d) 42
  e) 2n + n!
  f) $6n^{15} + n^2 - 5$

  Welche dieser Komplexitäten gelten als effizient berechenbar?

- **Aufgabe 4**

  Geben Sie für die folgenden Pseudocode-Algorithmen jeweils die Laufzeitkomplexität in Abhängigkeit von n an!

  a) Gegeben sei ein Feld A der Länge n.

```
Für alle Permutationen von a_1, ...,a_n ∈ A:
  Wenn ∀ i ∈ {1, ..., n-1} : a_i ≤ a_i+1 ⇒
        Feld ist sortiert, beende Algorithmus
```

Übungen (IV)

b)

```
for (i = 0; i<=n; i++){
    for (j = 0; j<=n; j++){
        “Do something”
    }
}
```

c) Gegeben sei ein Feld A der Länge n.

```
Gebe den Wert von a[1] zurück;
```

d) Gegeben sei ein Liste L mit n Elementen.

```
Gebe alle Elemente von L viermal aus.
```

e) Gegeben sei ein Feld A der Länge n.

```
Addiere die Werte in A und gebe das Resultat aus.
```

Algorithmen und Datenstrukturen

Teil 3:
Graphen (I): Grundbegriffe, Eigenschaften und Datenstrukturen

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 04/2019

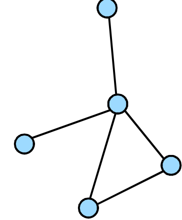
Gliederung

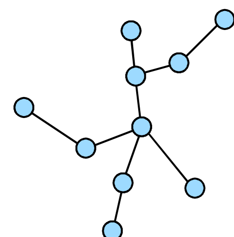
- Definitionen
- Operationen auf Graphen
- Eigenschaften von Graphen
- Datenstrukturen für Graphen

Motivation (I)

Einfärbung der Länder, so dass zwei Nachbarn keine gleichen Farben haben

Auffinden des kürzesten/ schnellsten Weges von A nach B

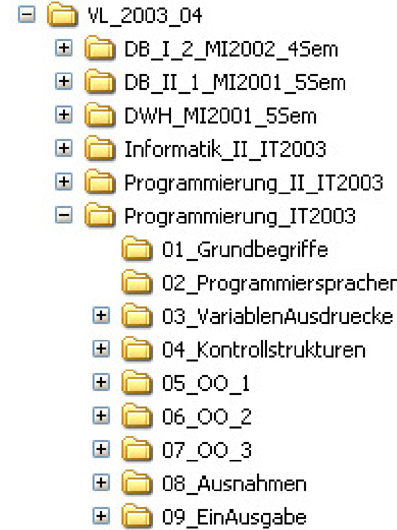




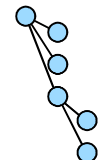
Punkte: Länder
Verbindungen: Nachbarschaft

Punkte: Orte/Bahnhöfe
Verbindungen: Strecke

Motivation (II)

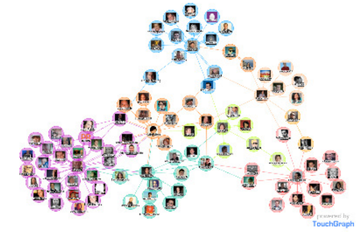


Durchsuchen aller Verzeichnisse einschl. Unterverzeichnissen ab bestimmtem Einstiegspunkt



Punkte: Verzeichnisse
Verb.: Verzeichnis-Unterverzeichnis-Beziehung

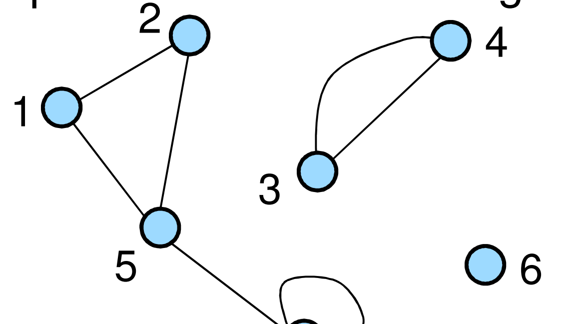
Auffinden „Freunde der Freunde“, „Personen mit ähnlichen Interessen“, etc.

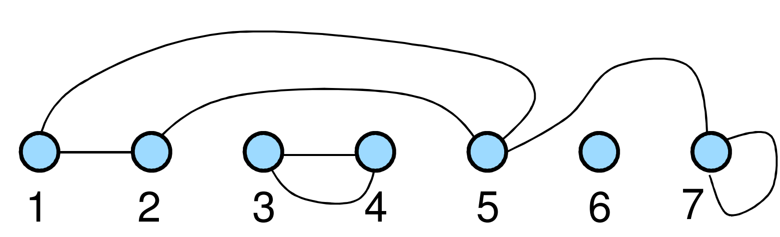


Punkte: Personen
Verb.: „Sich kennen“

Ungerichtete Graphen

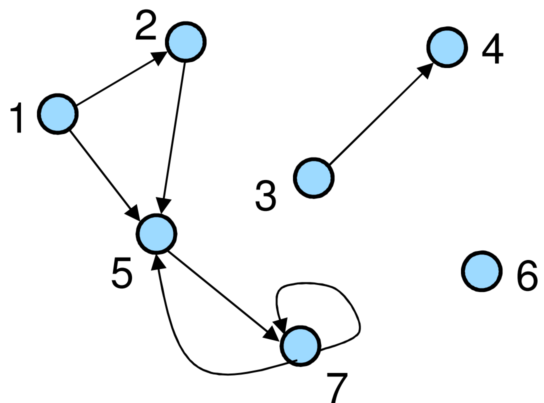
- Ein (ungerichteter) Graph G = (V,E) ist eine Struktur mit
  - V endliche Menge (Menge der Knoten oder Ecken, v = vertex)
  - E eine Menge ein- oder zweielementiger Teilmengen von V (Menge der Kanten, e = edge).
- Beispiel: G = ({1,2,3,4,5,6,7}, {{1,2}, {1,5}, {2,5}, {3,4}, {3,4}, {5,7}, {7}}).
- Die Knoten v_i, v_j einer Kante $\{v_i, v_j\}$ heißen Endknoten dieser Kante.
- Graphen werden meistens visualisiert:
  - Knoten als Punkte/Kreise
  - Kanten als Linien zwischen den Knoten
  - Wie die Knoten bzw. Kanten in der Ebene angeordnet werden, ist nicht vorgegeben; Frage der Zweckmäßigkeit, Lesbarkeit, Ästhetik, ...
  - Beispiele der Visualisierung von G:





Gerichtete Graphen

- Motivation: Manchmal ist es wichtig, von wo nach wo eine Kante verläuft (z.B. fährt der Zug von A nach B oder B nach A)
- Ein gerichteter Graph G = (V,E) ist eine Struktur mit
  - V endliche Menge (Menge der Knoten oder Ecken, v = vertex)
  - $E \subseteq V \times V$ (Menge der Kanten, e = edge).
- Beispiel: G = ({1,2,3,4,5,6,7}, {(1,2), (1,5), (2,5), (3,4), (5,7), (7,5), (7,7)}).
- Bezeichnungen:
  - In einer Kante $(v_i,v_j) \in E$ heißt v_i Start- oder Quellknoten und v_j Ziel- oder Endknoten
- Visualisierung: Pfeilspitze zum Zielknoten
- Beispiel:



Anmerkungen und Notationen

- Anmerkung:
  - In deutschsprachiger Literatur manchmal Definition G = (E,K) (E = Ecken, K = Kanten)
  - Kann zur Verwirrung führen (Bedeutung von E)!

- Notation und Sprechweisen: Sei G = (V,E) Graph, dann
  - Sagt man auch „G ist ein Graph auf V“
  - Schreibt man auch V(G) statt V
  - Schreibt man auch E(G) statt E
  - |V(G)| Ordnung oder Grad von G
  - Graphen der Ordnung 0 oder 1 werden als triviale Graphen bezeichnet
  - Kante mit gleichem Start- und Zielknoten bzw. gleichen Endknoten im ungerichteten Fall heißt Schlinge
  - Knoten ohne Kanten zu anderen Knoten, heißt isoliert
  - Existieren zwischen zwei Knoten zwei oder mehr Kanten, so spricht man von Zweifach-, Dreifach-, ... oder allgemein Mehrfachkante
  - Manchmal auch: Parallele Kanten

Inzidenz und Adjazenz

- Informal:
  - Inzidenz: Beziehung von Knoten zu Kanten
  - Adjazenz: Beziehung von Knoten untereinander
- Sei G = (V,E) ungerichteter Graph. Dann:
  - $v \in V$ heißt inzident zu $e \in E$, wenn $v \in e$
  - E(v) Menge aller zu $v \in V$ inzidenten Kanten
  - deg(v) := |E(v)| wird als Grad (Valenz) des Knoten v bezeichnet
  - $v_i, v_j \in V$ heißen adjazent (oder benachbart), wenn $\{v_i, v_j\} \in E$
- Sei G = (V,E) gerichteter Graph. Dann:
  - $E_{in}(v)$ Menge aller zu $v \in V$ eingehenden Kanten
  - $E_{out}(v)$ Menge aller von $v \in V$ ausgehenden Kanten
  - $deg_{in}(v) := | E_{in}(v) |$ Eingangsgrad von Knoten v
  - $deg_{out}(v) := | E_{out}(v) |$ Ausgangsgrad von Knoten v
  - $v_i, v_j \in V$ adjazent, wenn $(v_i, v_j) \in E$ oder $(v_j, v_i) \in E$
  - $v_i, v_j \in V$ stark adjazent, wenn $(v_i, v_j) \in E$ und $(v_j, v_i) \in E$

Markierte Graphen (I)

- Motivation:
  - Zusätzliche Informationen zu Knoten und/oder Kanten notwendig, z.B.:
    - Welche Farbe hat ein Land?
    - Wie lange dauert die Fahrt von Bahnhof A nach B?
- Dazu: Knoten und/oder Kanten mit entsprechenden Informationen versehen
- Man sagt: Graph ist knoten- bzw. kantenmarkiert.
- Definition: Sei G = (V,E) ein (un)gerichteter Graph, M_V und M_E Mengen und $f : V \rightarrow M_V$ und $g : E \rightarrow M_E$ Abbildungen
  - G‘ = (V,E,f) heißt knotenmarkierter Graph
  - G“ = (V,E,g) heißt kantenmarkierter Graph
  - G‘“ = (V,E,f,g) heißt knoten- und kantenmarkierter Graph
- Anm.: Mengen M_V und M_E häufig Zahlen oder Aufzählungstypen

Markierte Graphen (II)

- Beispiele:
  - $V = \{D, NL, B, CH, ...\}$
  - $M_V = \{gelb, hellblau, lila, ...\}$
  - f(D) = Lila, f(NL) = hellblau, f(B) = gelb

  - $V = \{Uni, Österfeld, Vaihingen, ...\}$
  - $E = \{(Uni,Österfeld),$
    $(Österfeld,Vaihingen), ...\}$
  - $M_E = R^+$
  - g((Uni,Österfeld)) = 2
  - g((Österfeld,Vaihingen)) = 3
  - ...

Kantenfolge

- Sei G = (V,E) ein (un)gerichteter Graph und $k = (v_o, \ldots, v_n) \in V^{n+1}$ eine Folge von n+1 Knoten.
- Def.: k heißt Kantenfolge der Länge n von v_o nach v_n, wenn für alle $i \in \{0, \ldots, n-1\}$: $(v_i, v_{i+1}) \in E$
- Bezeichnungen:
  - G gerichtet: v_o Startknoten und v_n Endknoten
  - G ungerichtet: v_o und v_n Endknoten
  - In beiden Fällen: Knoten v_1 bis v_{n-1} innere Knoten
- Def.: Gilt $v_o = v_n$: Kantenfolge geschlossen
- Anschaulich: Kantenfolge kann jeden Knoten (und implizit jede Kante) beliebig oft enthalten
- Def.: $k = (v_o)$ heißt triviale Kantenfolge

- Def.: k heißt Kantenzug der Länge n von v_o nach v_n, wenn
  - k Kantenfolge der Länge n von v_o nach v_n ist
  - für alle $i, j \in \{0, ..., n-1\}$ mit $i \neq j$: $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$.
- Anschaulich: In Kantenzug kommt keine Kante mehrfach vor

- Def.: k heißt Weg (oder Pfad) der Länge n von v_o nach v_n, wenn
  - k Kantenfolge der Länge n von v_o nach v_n ist
  - für alle $i, j \in \{0, ..., n\}$ mit $i \neq j$: $v_i \neq v_j$.
- Anschaulich: In Weg kommt kein Knoten mehrfach vor

- Def.: k heißt Kreis oder Zyklus der Länge n (oder n-Zyklus) von v_o nach v_n, wenn
  - k geschlossene Kantenfolge der Länge n von v_o nach v_n ist
  - $k' = (v_o, ..., v_{n-1})$ ein Weg ist.
- Def.: Graph ohne Zyklus heißt kreisfrei, zyklenfrei oder azyklisch

Beispiel

G

- (1,2,3,4,5,2,3) ist Kantenfolge (aber nicht Kantenzug) der Länge 6 von 1 nach 3.
- (1,2,5,4,2,3) ist Kantenzug (aber nicht Weg) der Länge 5 von 1 nach 3.
- (1,2,5,4,3) ist Weg der Länge 4 von 1 nach 3.
- (2,3,4,5,2) ist Zyklus der Länge 4.

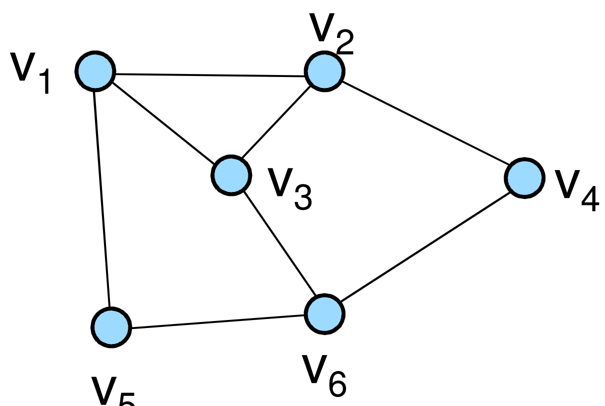
Übersicht

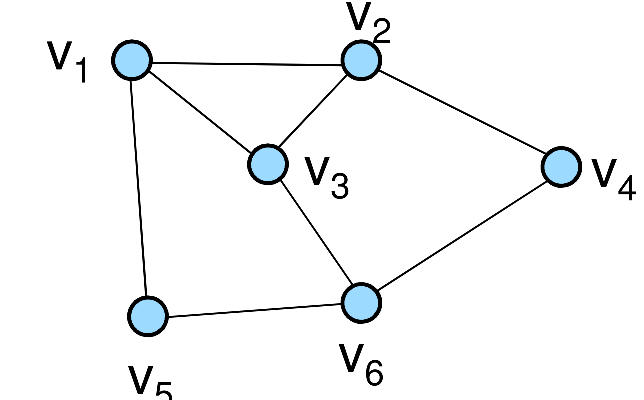
- Definitionen
- Operationen von Graphen
- Eigenschaften auf Graphen
- Datenstrukturen für Graphen

Hinzufügen Knoten

- Sei G = (V,E) ein Graph
- Hinzufügen eines Knotens v (Notation: G + v) erzeugt Graphen (V ∪ {v}, E)
- Beispiel:

G_1 $G_1 + v_7$

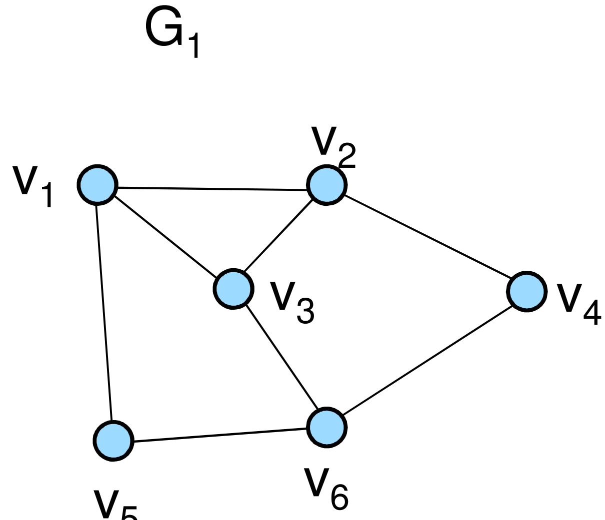




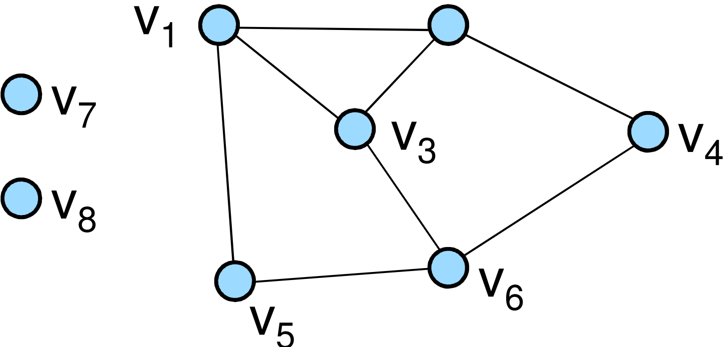
Hinzufügen Knotenmenge

- Sei G = (V,E) Graph, X Menge von Knoten
- Dann: G + X Graph, der durch Hinzufügen aller x ∈ X zu G entsteht
- Beispiel:

G_1

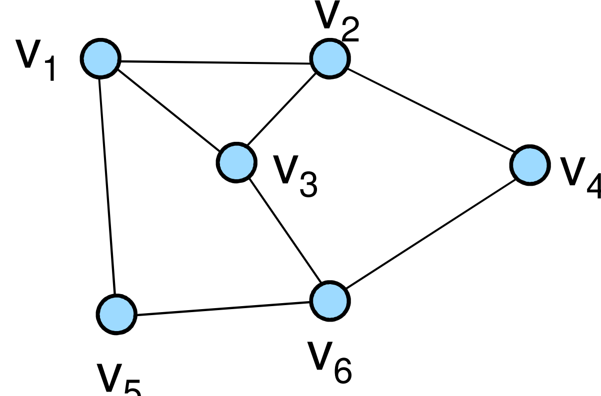


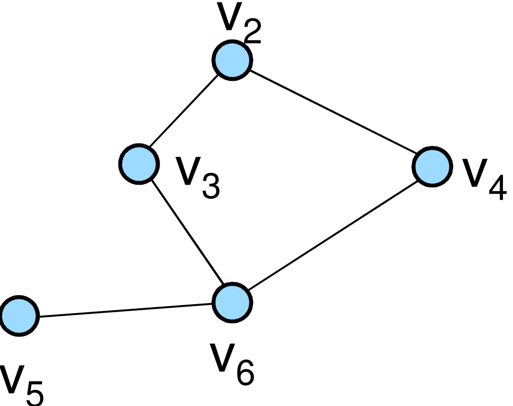
$G_1 + X$ mit $X = \{v_7, v_8\}$



Entfernen Knoten

- Sei G = (V,E) ein Graph
- Entfernen eines Knotens $v \in V$ (Notation: G – v) erzeugt Graphen (V\{v}, E\E(v)), wobei E(v) Menge aller zu v inzidenten Kanten
- Beispiel:

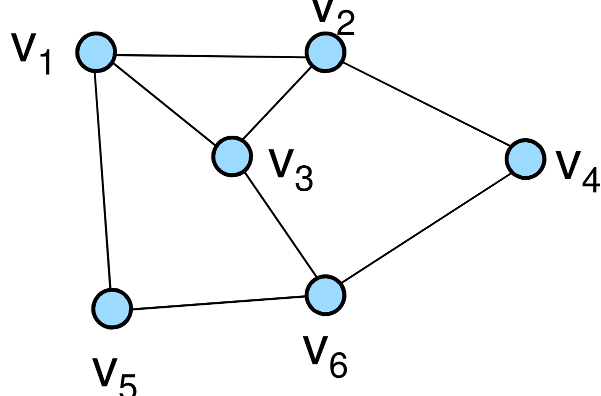


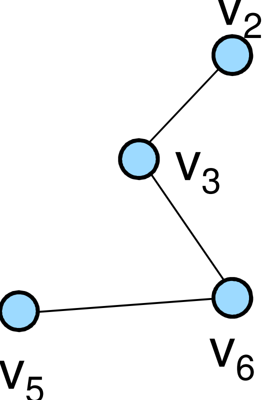


Entfernen Knotenmenge

- Sei $G = (V,E)$ ein Graph, $X \subseteq V$
- Dann: $G - X$ Graph $(V \setminus X, E \setminus E(X))$ mit $E(X)$ Menge aller Kanten, die zu mindestens einem $x \in X$ inzident sind
- Beispiel:

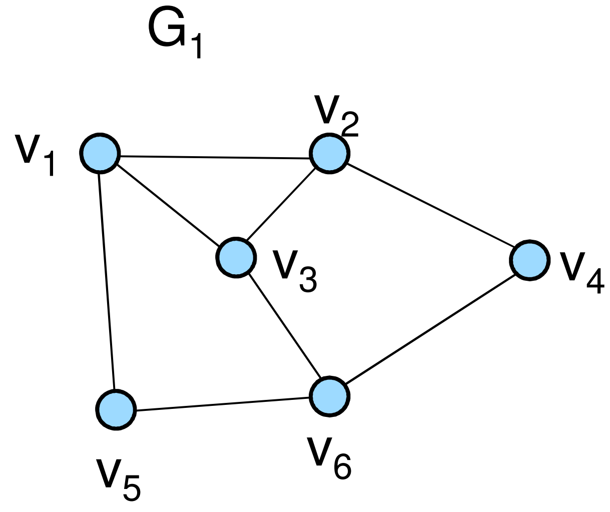
G_1 | $G_1 - X$ mit $X = \{v_1, v_4\}$

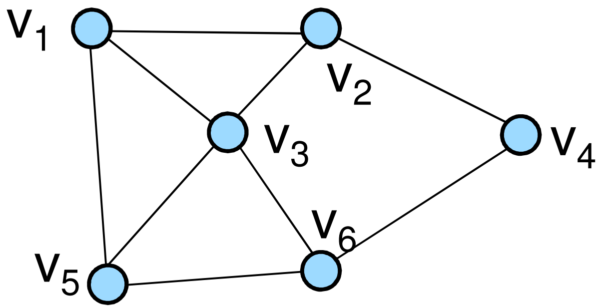




Hinzufügen Kante

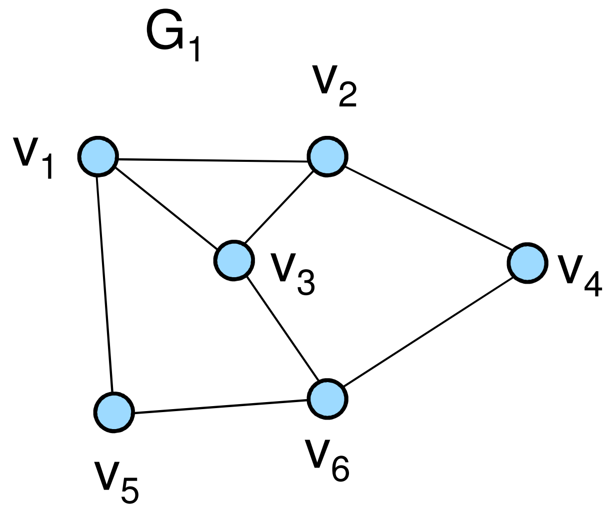
- Sei G = (V,E) ein Graph
- Hinzufügen einer Kante e (Notation: G + e) erzeugt Graphen (V, E ∪ {e})
- Beispiel:

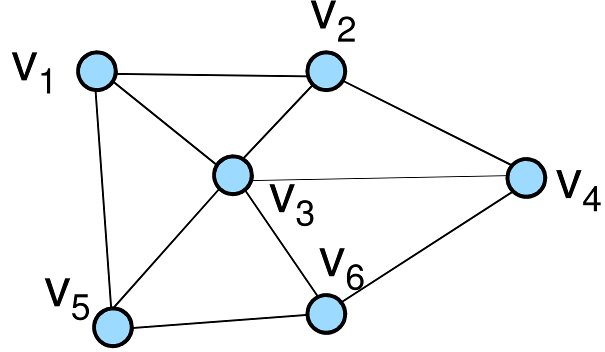




Hinzufügen Kantenmenge

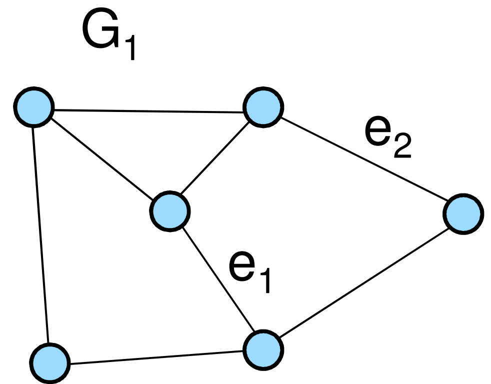
- Sei G = (V,E) ein Graph
- Ist F Menge von Kanten, dann ist G + F Graph, der durch Hinzufügen aller f $\in$ F zu G entsteht
- Beispiel:

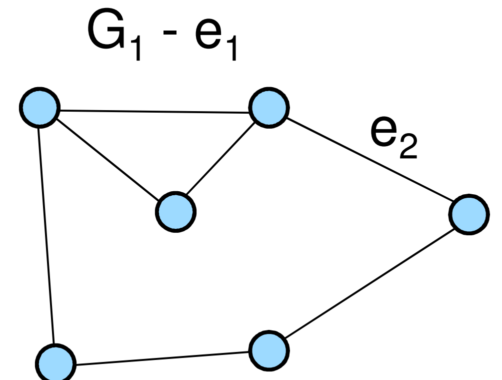




Entfernen Kante

- Sei G = (V,E) ein Graph
- Entfernen einer Kante $e \in E$ (Notation: $G – e$) erzeugt Graphen (V, E\{e})
- Beispiel:

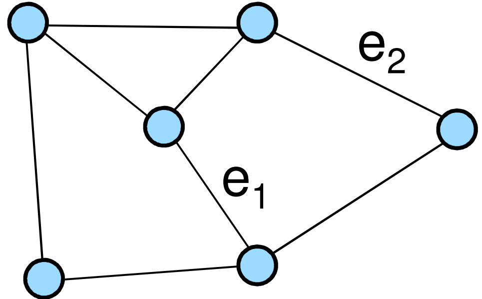




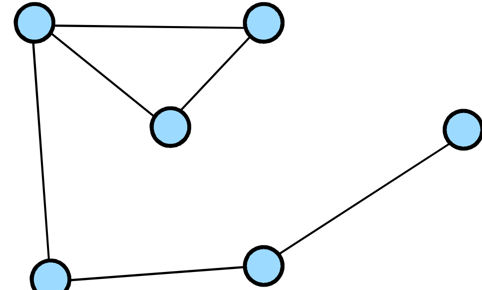
Entfernen Kantenmenge

- Sei G = (V,E) ein Graph
- Ist $F \subseteq E$, dann ist G – F Graph, der durch Entfernen aller $f \in F$ aus G entsteht
- Beispiel:

G_1

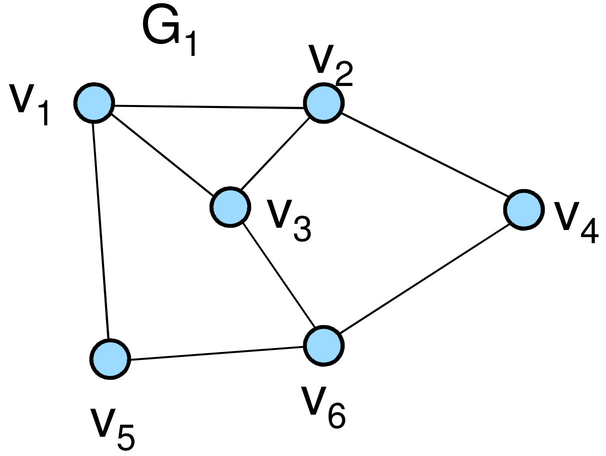


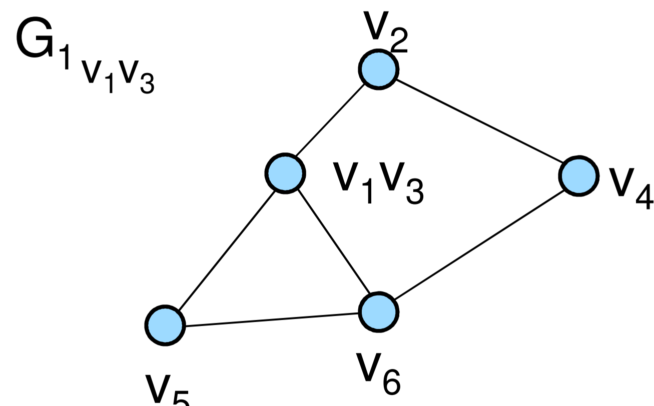
G_1 – F mit $F=\{e_1, e_2\}$



Fusion

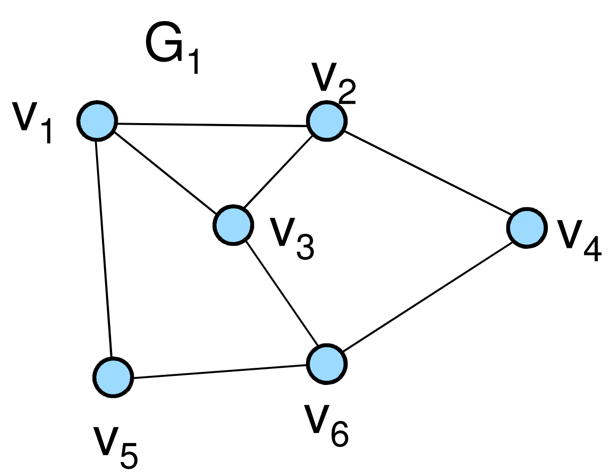
- Synonym: Verschmelzen
- Sei G = (V,E) ein Graph
- Fusion zweier Knoten u, v ∈ V (Notation: G_{uv}) erzeugt Graphen
  - mit Knoten uv und allen Kanten, die vorher u oder v als Endknoten hatten
  - Mehrfachkanten zu uv werden entfernt
- Beispiel:

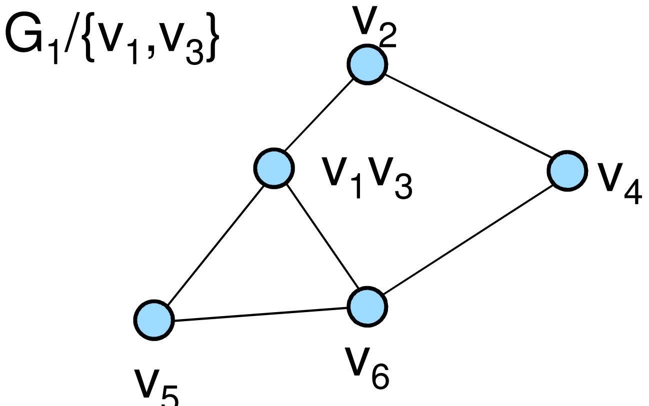




(Kanten-)Kontraktion

- Sei G = (V,E) ein Graph
- Kontraktion der Kante e={u,v} ∈ E (Notation: G/e) ist Entfernen von e mit anschließender Fusion von u und v
- Beispiel:



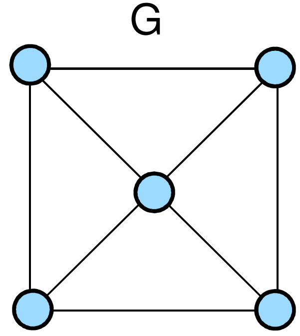


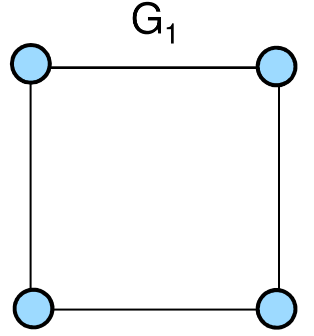
Übersicht

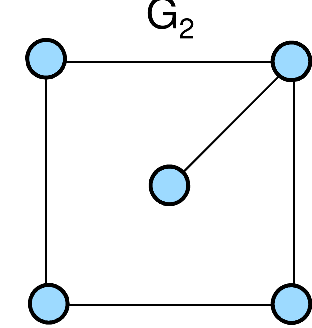
- Definitionen
- Operationen auf Graphen
- Eigenschaften von Graphen
- Datenstrukturen für Graphen

Teilgraph/Untergraph

- Seien G und G' Graphen.
  Ist $V' \subseteq V$ und $E' \subseteq E$, dann heißt G' Teilgraph von G.
  Man sagt auch: G enthält den Graphen G'.
- Sei G' Teilgraph von G.
  Enthält G' alle Kanten $(v_i,v_j) \in E$ mit $v_i,v_j \in V'$, dann heißt G' Untergraph von G.
  Man sagt auch: G' wird induziert (oder aufgespannt) von V' in G.
- Beispiel:



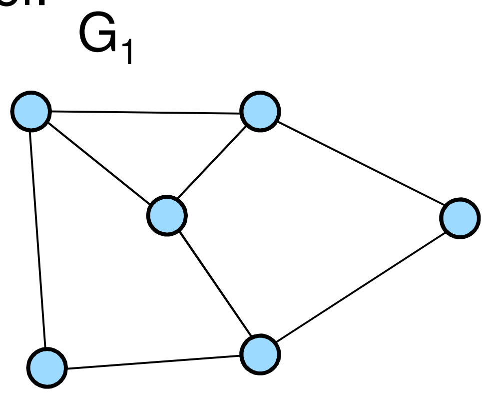


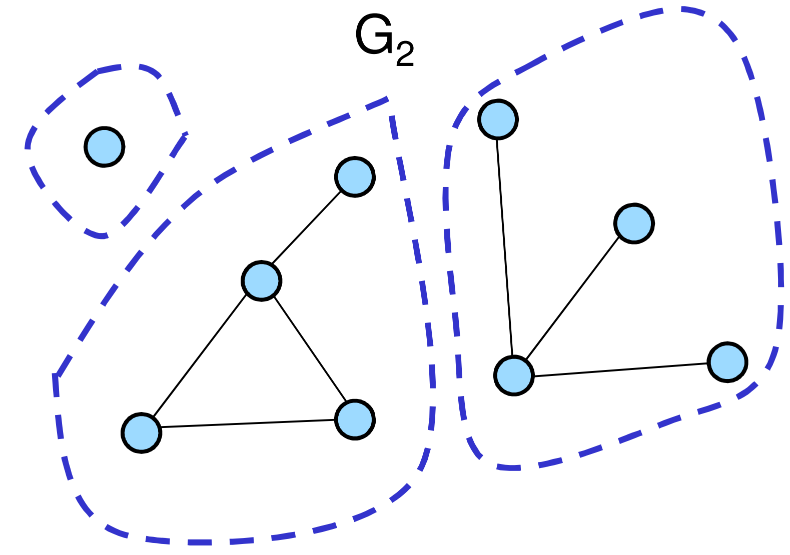


  - G_1 und G_2 sind Teilgraphen von G
  - G_1 ist Untergraph von G, G_2 nicht

Zusammenhängende Graphen

- Graph G = (V,E) heißt zusammenhängend, wenn es für alle $v_i, v_j \in V$ Weg von v_i nach v_j gibt.
- Zerfällt Knotenmenge von G in disjunkte Teilmengen $G_1, ..., G_s$ und zu jedem G_i gehöriger Untergraph ist zusammenhängend, dann werden diese Untergraphen Zusammenhangskomponenten von G genannt.
- comp(G) := Anzahl der Zusammenhangskomponenten von G.
- Beispiel:



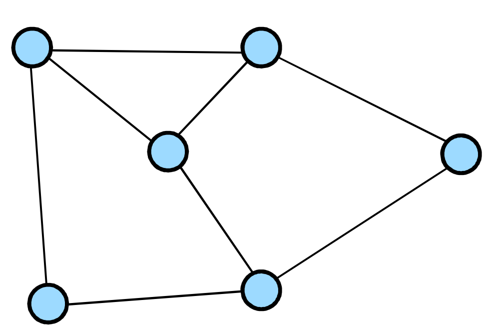


  - G_1 ist zusammenhängend, G_2 nicht
  - $comp(G_1) = 1$, $comp(G_2) = 3$
  - G_2 besteht aus (zerfällt in) in drei Zusammenhangskomponenten

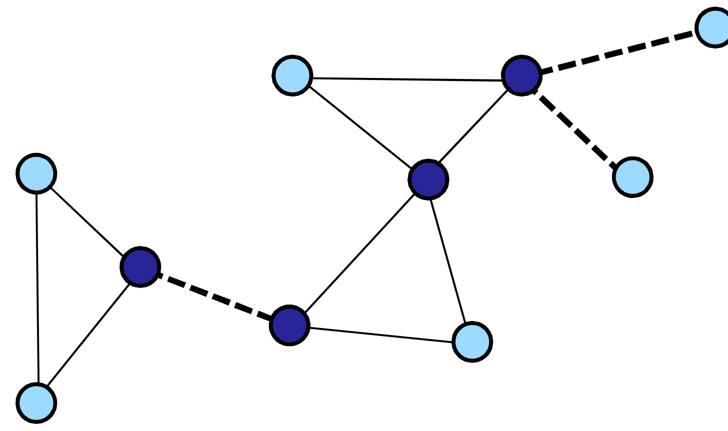
Brücke und Artikulation

- Sei G = (V,E) Graph
- Kante e ∈ E heißt Brücke, wenn comp(G-e) > comp(G).
- Knoten v ∈ V heißt Artikulation, wenn comp(G-v) > comp(G).
- Beispiel:

G_1



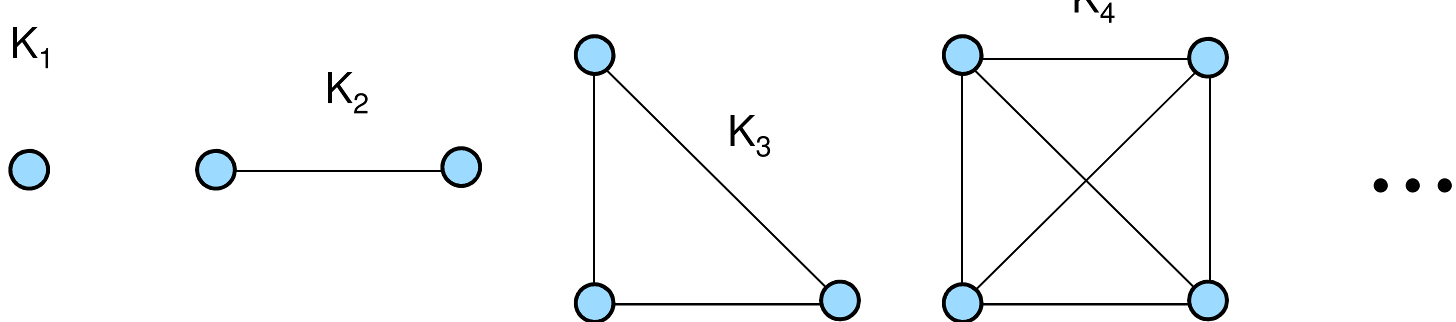
G_3



- G1 besitzt keine Brücken und keine Artikulationen
- G3: ● Artikulationen, ------ Brücken

Vollständige Graphen

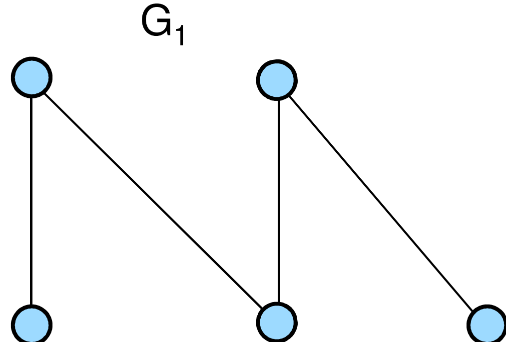
- Ein Graph G = (V,E) heißt vollständig, wenn je zwei Knoten v_i, $v_j \in V$ benachbart sind.
- Zu jeder Knotenzahl gibt es genau einen vollständigen Graphen, der als K_n bezeichnet wird.
- Beispiele:



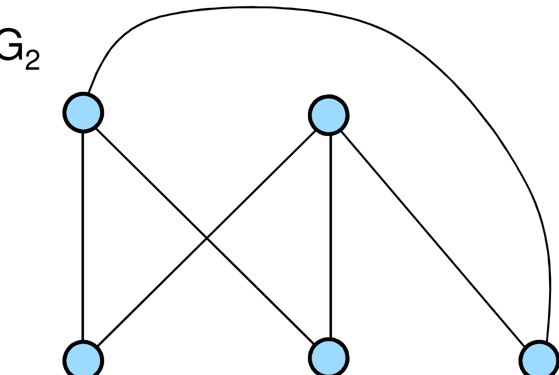
Bipartite Graphen

- Graph G = (V,E) heißt bipartit, falls sich die Knotenmenge V in zwei disjunkte Teilmengen V‘ und V“ zerlegen lässt, so dass weder Knoten aus V‘ noch solche aus V“ benachbart sind.
- Graph G = (V,E) heißt vollständig bipartit, falls G bipartit ist und jeder Knoten v‘∈ V‘ mit jedem Knoten v“∈ V“ benachbart ist.
- Vollständig bipartiter Graph mit |V‘|=n und |V“|=m wird als $K_{m,n}$ bezeichnet.
- Beispiele:

G_1



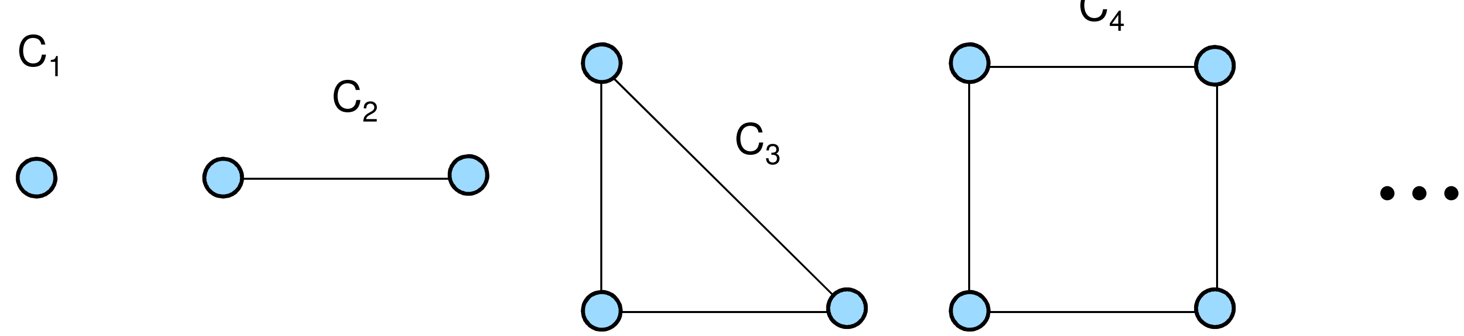
G_2



  - G_1 ist bipartit, aber nicht vollständig bipartit
  - G_2 ist vollständig bipartit, es ist der $K_{3,2}$
- Anm.: $K_{1,n}$ heißt auch Stern

Zyklische Graphen

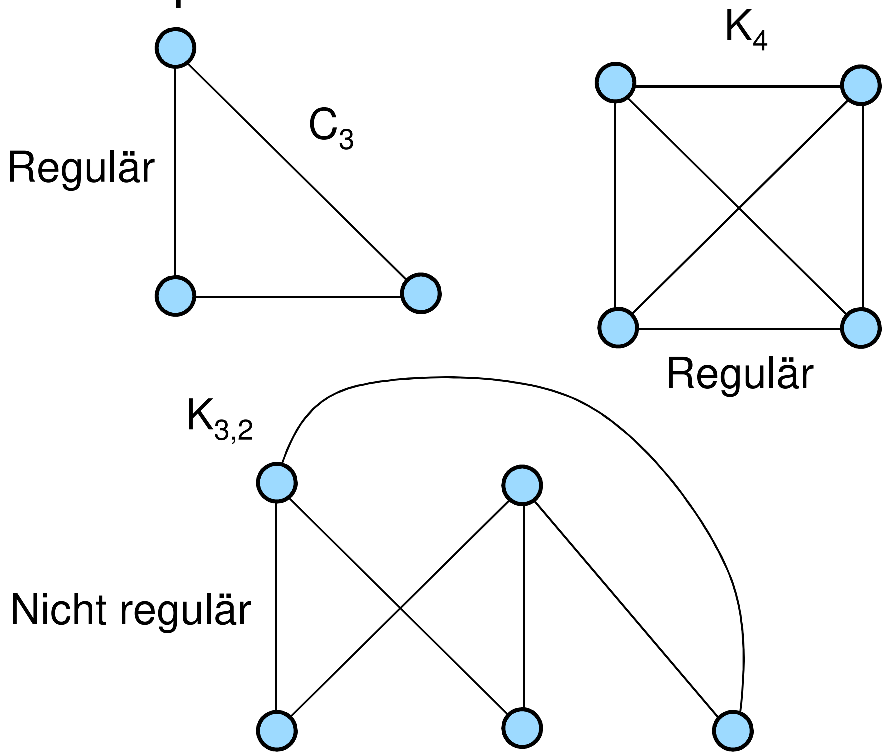
- Synonym: Kreis
- Graph G = (V,E) heißt zyklisch, wenn G aus einem einfachen geschlossenen Weg besteht.
- Zyklischer Graph mit n Knoten heißt C_n.
- Beispiele:

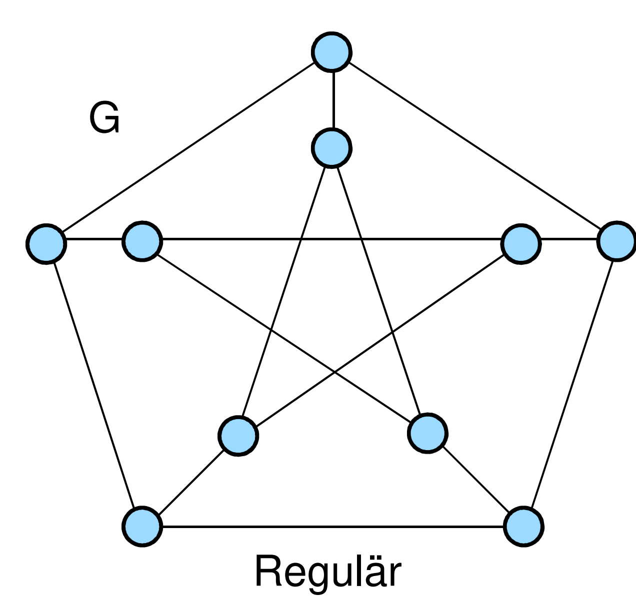


- Anmerkung: Für $n \leq 3$: $K_n = C_n$

Reguläre Graphen

- Sei G = (V,E) ein Graph.
- Hat jeder Knoten $v \in V$ gleichen Grad, dann heißt G regulär.
- Beispiele:

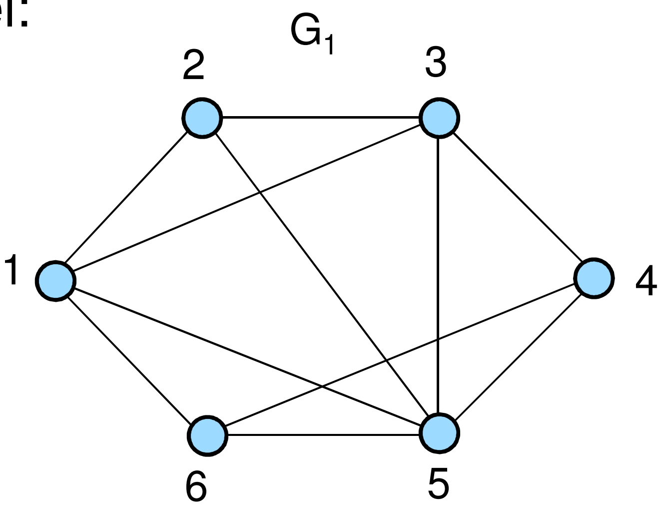


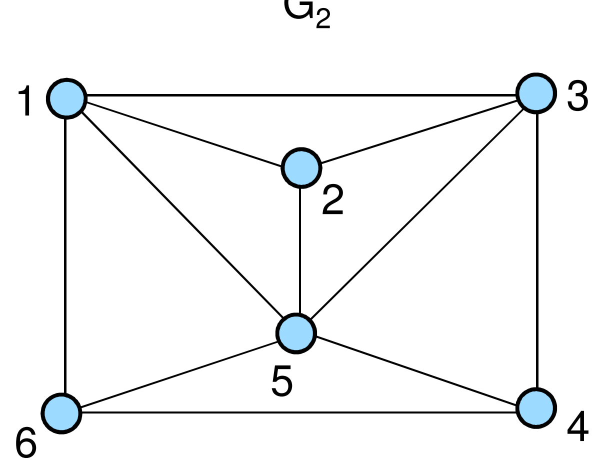


- Graphen C_n, K_n regulär für alle n
- Graphen $K_{n,m}$ sind regulär für n=m und nicht regulär für $n \neq m$
- Graph G heißt nach seinem „Entdecker“ Petersengraph

Planare Graphen

- Graph G = (V,E) heißt planar (oder plättbar oder eben), wenn in der Ebene kreuzungsfreie Darstellung von G existiert.
- Beispiel:

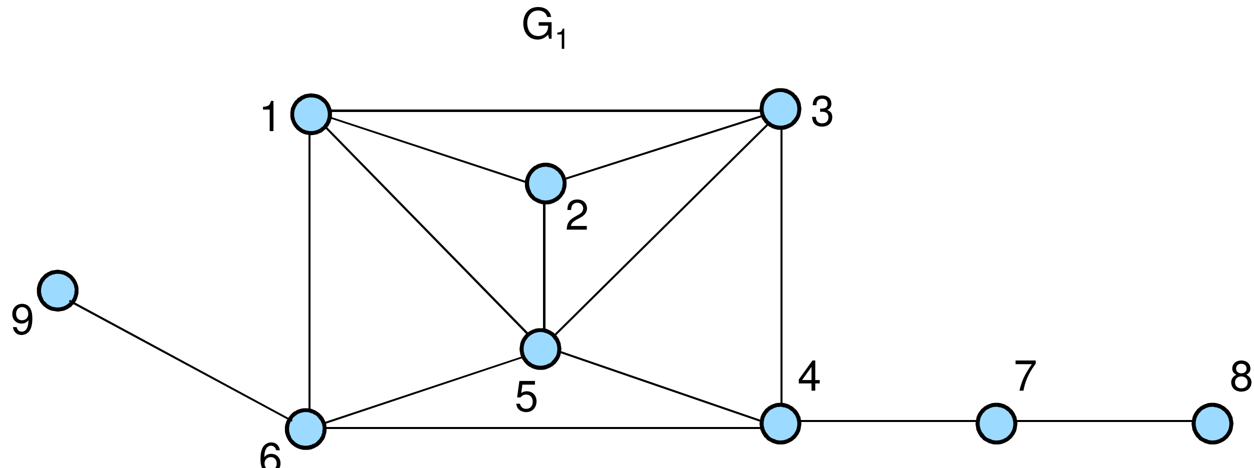




- G_1 ist planarer Graph (G_2 ist ein Beispiel seiner kreuzungsfreien Darstellung)
- Graphen $K_{3,3}$ und K_5 sind nicht planar:
  - Haben besondere theoretische Bedeutung
  - Sind „kleinste“ nicht-planare Graphen

Distanz und Extrenzität (I)

- Sei Graph G = (V,E)
- Distanz(v,v‘) := Minimaler Abstand zwischen v und v‘ in Kanten
- Extrenzität(v) := Maximum aller Distanzen von v
  (Maximaler Abstand zu allen anderen v‘ $\in$ V)
- Beispiel:



Distanz und Extrenzität (II)

- Distanz:

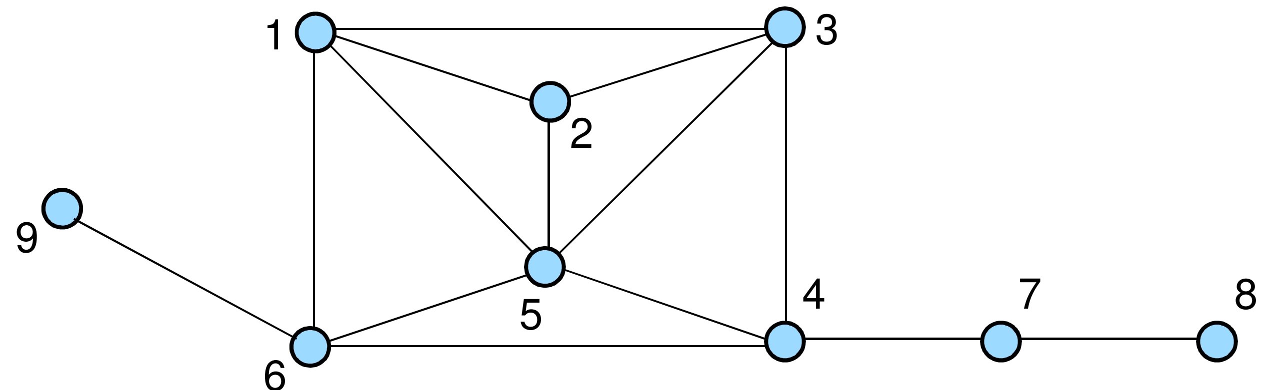
	1	2	3	4	5	6	7	8	9
1	0	1	1	2	1	1	3	4	2
2	1	0	1	2	1	2	3	4	3
3	1	1	0	1	1	2	2	3	3
4	2	2	1	0	1	1	1	2	2
5	1	1	1	1	0	1	2	3	2
6	1	2	2	1	1	0	2	3	1
7	3	3	2	1	2	2	0	1	3
8	4	4	3	2	3	3	1	0	4
9	2	3	3	2	2	1	3	4	0

- Extrenzität:

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

Rand, Durchmesser, Radius, Zentrum

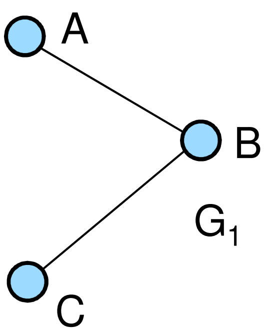
- Rand(G) := Menge der Knoten mit maximaler Exzentrizität
- Durchmesser(G) := Größter Abstand zwischen zwei Knoten
- Radius (G) := Minimum der Exzentrizität
- Zentrum (G) := Knoten, deren Exzentrizität gleich Radius
- Beispiel:

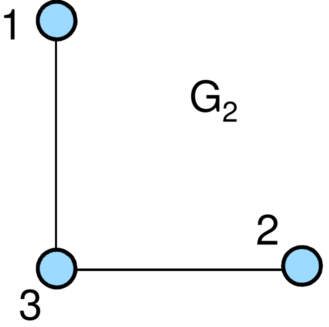


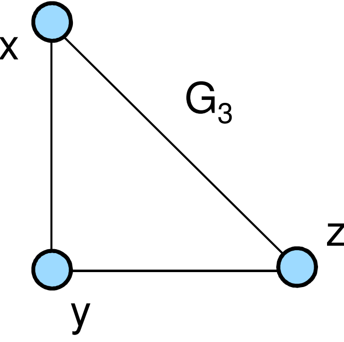
Eigenschaft	Wert
Rand(G_1)	{1-2-8-9}
Durchmesser(G_1)	4
Radius(G_1)	2
Zentrum(G_1)	{4}

Isomorphie von Graphen

- Seien G_1, G_2 Graphen. G_1 und G_2 sind isomorph ($G_1 \cong G_2$), wenn es eine bijektive Abbildung $\varphi : V_1 \rightarrow V_2$ gibt mit $(v_i,v_j) \in E_1 \Leftrightarrow (\varphi(v_i), \varphi(v_j)) \in E_2$.
- Beispiel:







  - $G_1 \cong G_2$ (φ(A)=1, φ(B)=3, φ(C)=2)
  - G_3 ist nicht isomorph zu den beiden anderen Graphen (ungleiche Kantenzahl)

Übersicht

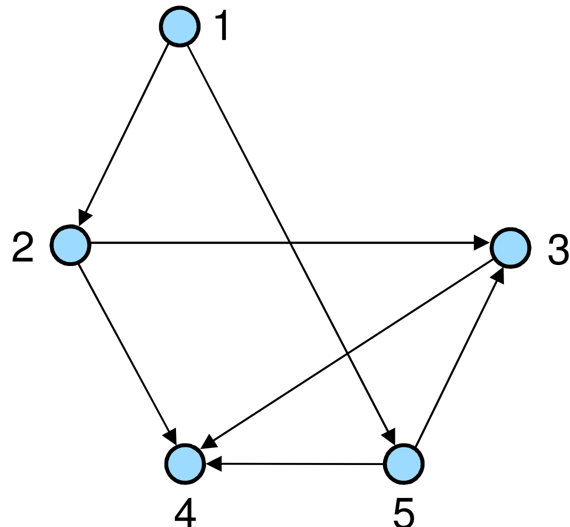
- Definitionen
- Operationen auf Graphen
- Eigenschaften von Graphen
- Datenstrukturen für Graphen

Anforderungen

- Vollständige Darstellung aller Informationen, d.h.
  - von Knoten
  - von Kanten
  - von möglicherweise vorliegenden Markierungen
- Möglichst effizientes Durchführen wichtiger lesender Operationen, z.B.
  - Adjazenz zweier Knoten feststellen
  - Bestimmung aller Nachbarn eines Knoten (ungerichteter Graph)
  - Bestimmung aller Vorgänger bzw. Nachfolger eines Knoten (gerichteter Graph)
- Möglichst effizientes Durchführen von Änderungsoperationen, z.B. Fusion und Kontraktion
- Möglichst geringer Speicherplatz für das Darstellen der Struktur
- Häufig: Wechselwirkung, d.h. Realisierung effizienter Strukturen geht zu Lasten des Speicherplatzes („Time-Space-Tradeoff“)

Adjazenzmatrix (I): Definition

- Darstellung eines Graphen als $n \times n$-Matrix mit n Anzahl Knoten
- Matrix-Eintrag an Stelle (i,j) ist 1, falls es eine Kante von Knoten v_i nach v_j gibt, ansonsten 0
- Matrix spiegelt Knoten-Nachbarschaft wieder, daher Adjazenzmatrix
- Beispiel:



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

Adjazenzmatrix (II): Eigenschaften

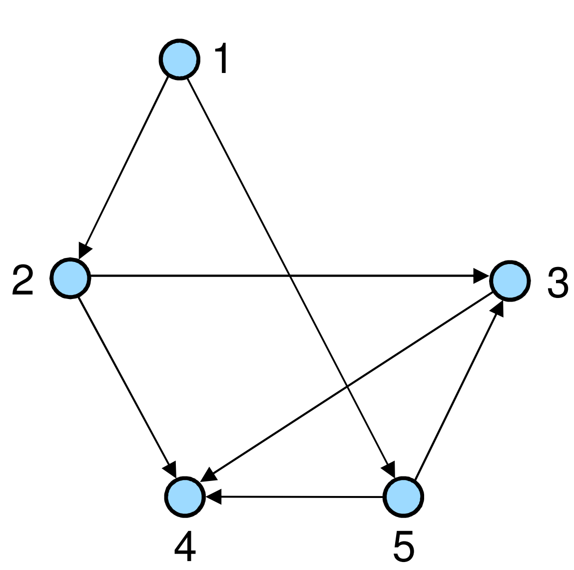
- Bei ungerichteten Graphen ist Adjazenzmatrix stets symmetrisch
- Wert 1 auf Diagonalelement zeigt Schlinge an
- Mehrfachkanten durch Eintragen er Kantenanzahl statt 0/1
- Kantenmarkierungen: Statt 0/1 Markierung in Matrix eintragen
- Knotenmarkierungen: Separate Liste/Feld

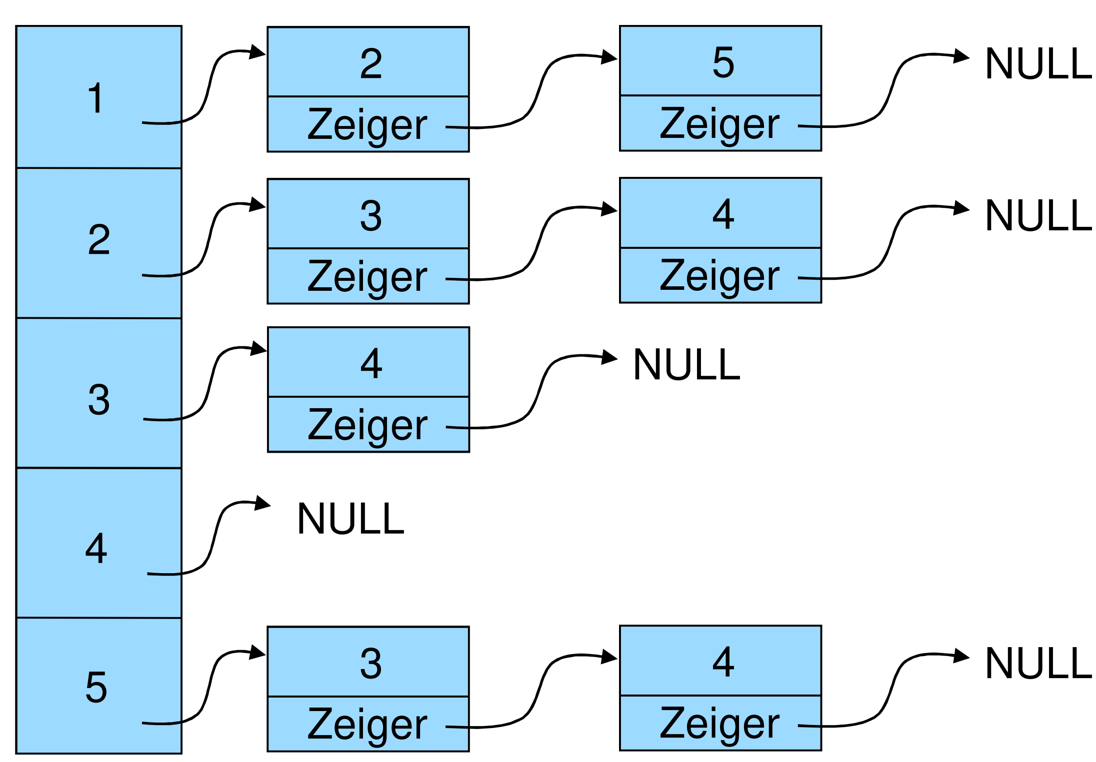
Adjazenzmatrix (III): Bewertung

- Erlaubt Darstellung gerichteter wie ungerichteter Graphen
- Erlaubt vollständige Darstellung inkl. Markierungen
- Lesende Operationen:
  - Adjazenz zweier Knoten unabhängig von Größe feststellbar
  - Bestimmung aller Nachbarn eines Knoten (ungerichteter Graph) bzw. aller Vorgänger bzw. Nachfolger eines Knoten (gerichteter Graph):
    - Durchsehen der i-ten Zeile/Spalte der Adjazenzmatrix (abhängig von Knotenzahl)
- Ändernde Operationen: Schwierig wegen fester Matrixgröße
- Speicherplatz:
  - Immer n^2 Speicherplätze
  - Relativ ineffizient:
    - Bei Graph mit relativ wenig Kanten sind viele Einträge 0 (Dünn/spärlich besetzte Matrix)
    - Bei ungerichteten Graphen aufgrund der Symmetrie auch etwa halber Speicherplatz ausreichend

Adjazenzlisten (I): Definition

- In Adjazenzlisten wird
  - Ein Feld der Knoten angelegt
  - Jeder Feldeintrag ist Anker einer Liste der Nachbarn dieses Knoten
- Beispiel:





Adjazenzlisten (II): Eigenschaften

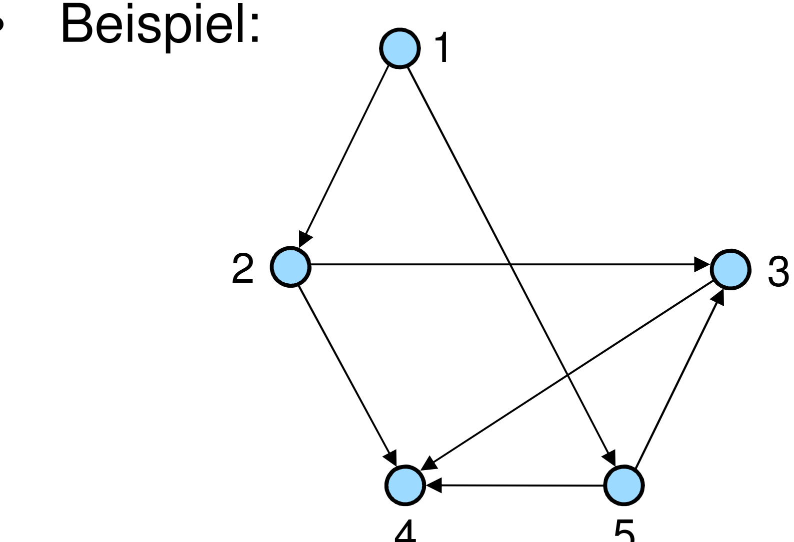
- Kantenmarkierungen: Kann beim Listeneintrag hinzugefügt werden
- Knotenmarkierungen: Kann beim Feld der Knoten hinzugefügt werden

Adjazenzlisten (III): Bewertung

- Erlaubt Darstellung gerichteter wie ungerichteter Graphen
- Erlaubt vollständige Darstellung inkl. Markierungen
- Lesende Operationen:
  - Adjazenz zweier Knoten: Durchsuchen entsprechender Liste
  - Bestimmung aller Nachbarn eines Knoten (ungerichteter Graph) bzw. aller Nachfolger eines Knoten (gerichteter Graph):
    - Sehr effizient (Ergebnis liegt schon vor)
  - Bestimmung aller Vorgänger eines Knoten (gerichteter Graph): Relativ aufwändig, alle Nachbarlisten des Knoten müssen durchsucht werden
- Ändernde Operationen: Effizienter als in Matrix
- Speicherplatz:
    - Für gerichteten Graphen n+m Speicherplätze
    - Für ungerichteten Graphen n+2m Speicherplätze
    - Bei Graphen mit relativ wenigen Kanten (dünn besetzte Matrix) erheblich effizienter als Adjazenzmatrix

Adjazenzlisten (IV): Alternative Darstellung

- Alternative zur Realisierung mit Zeigern: Realisierung von Adjazenzlisten durch zwei Felder:
  - Feld A zum Abspeichern der Nachbarn
  - Nachbarn in A lückenlos abgelegen (zuerst die von Knoten 1, dann von Knoten 2, usw.)
  - Festlegung, an welcher Position die Nachbarn eines bestimmten Knotens beginnen, durch zweites Feld N
  - N enthält die Indizes der Anfänge der Nachbarlisten, d.h. die Nachbarn von Knoten i sind in den Komponenten A[N[i]] bis A[N[i+1]]-1 abgelegt
- Beispiel:



N:

1	2	3	4	5
1	3	5	6	6

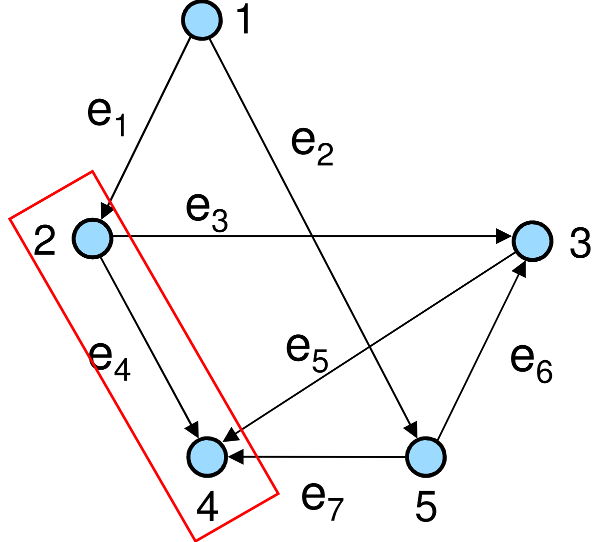
A:

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

Nachbarschaftslisten der Knoten 1,2,3 und 5

Inzidenzmatrix (I): Definition

- G mit n Knoten und m Kanten wird als nxm-Matrix I dargestellt
- Wert an Komponente (i,j) der Matrix gibt Inzidenz von Knoten i mit Kante j an:
  - Ungerichteter Graph: $I_{ij} := \begin{cases} 1, \; falls \; v_i \in e_j \\ 0, \; sonst \end{cases}$
  - Gerichteter Graph: $I_{ij} := \begin{cases} 1, \; falls \; e_j = (v_i, x) \\ -1, \; falls \; e_j = (x, v_i) \\ 0, \; sonst \end{cases}$
- Beispiel:



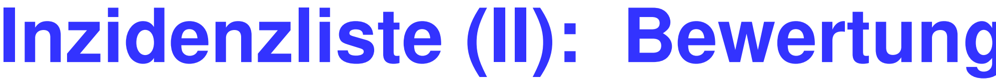
$$I = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Inzidenzmatrix (II): Bewertung

- Erlaubt Darstellung gerichteter wie ungerichteter Graphen
- Markierungen:
  - Kanten: Einträge in Matrix Markierung statt 0/1
  - Knoten: Separates Feld
- Lesende Operationen:
  - Inzidenz zwischen Knoten und Kante unabhängig von Größe feststellbar
  - Bestimmung aller Nachbarn eines Knoten (ungerichteter Graph) bzw. aller Vorgänger bzw. Nachfolger eines Knoten (gerichteter Graph):
    - Durchsehen der i-ten Zeile der Inzidenzmatrix (abhängig von Kantenzahl)
  - Effizient, wenn von Kante ausgehend inzidente Knoten benötigt werden
- Ändernde Operationen: Schwierig wegen fester Matrixgröße
- Speicherplatz:
  - Immer nxm Speicherplätze
  - Ineffizient:
    - Jede Spalte hat nur zwei von 0 verschiedene Einträge

Inzidenzliste (I): Definition

- In Inzidenzlisten wird
  - Liste der Kanten angelegt
  - Jeder Listeneintrag hat zwei Elemente (Start- und Zielknoten der Kante)
- Beispiel:

Inzidenzliste (II): Bewertung

- Erlaubt Darstellung gerichteter wie ungerichteter Graphen
- Markierungen:
  - Kanten: Listenelement um Eintrag erweitern
  - Knoten: Separates Feld
- Lesende Operationen:
  - Schnelles Navigieren entlang der Kanten
  - Alle weiteren Operationen (z.B. Nachbarn) extrem aufwändig
  - Effizient, wenn Fortlaufend Informationen über Kanten relevant sind
- Ändernde Operationen:
  - Kanten einfügen und löschen effizient
  - Knotenänderungen aufwändig
- Speicherplatz:
  - Linear wachsend mit Anzahl Kanten, d.h. O(m)
  - Effizient bei Graphen mit wenigen Kanten

Andere Darstellungen

- Verschiedene andere Darstellungen aus Literatur bekannt:
  - Kantenliste
  - Implizite Darstellung
- Weiterhin:
  - Varianten von Adjazenzmatrix und –liste
  - Basieren auf redundanten, im konkreten Fall vorteilhaften Ergänzungen

Zusammenfassung (I)

- Definitionen:
  - Gerichtete und ungerichtete Graphen
  - Inzidenz und Adjazenz
  - Markierte Graphen
  - Kantenfolge, Kantenzug, Weg, Zyklus

- Operationen auf Graphen:
  - Hinzufügen/Entfernen von Knoten(mengen)
  - Hinzufügen/Entfernen von Kanten(mengen)
  - Fusion
  - Kontraktion

Zusammenfassung (II)

- Eigenschaften von Graphen:
  - Teilgraph, Untergraph
  - Zusammenhang
  - Brücke und Artikulation
  - Vollständige, bipartite, zyklische Graphen
  - Reguläre Graphen
  - Planare Graphen
  - Rand, Durchmesser, Radius und Zentrum
  - Isomorphie von Graphen

- Datenstrukturen zur Darstellung von Graphen:
  - Adjazenzmatrix
  - Adjazenzliste
  - Inzidenzmatrix
  - Inzidenzliste
  - Bewertung der Darstellungen
  - Andere Darstellungsformen

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
X I R T A M Z N E D I Z N I A A H D P B
N J Z T Y Q N B J L N F B K L F P H G P
A I Y G W Q Z S G E P X T Z Y K R U G F
S Y Z Q V O N G Z O V N E K V C O D W R
D L X A B Y S A S Q K W T D C K M O T O
M I D U Y T J L K R A J K N C O O X A Y
H E B L H D C Y F X O R X O N I S Q I U
D P T R A B E E J M Y Z S K J G I N L H
T S Q Q Z W U Y J H C M K J M P V G R P
C I V C F Q E N B F G B J S T T A B G W
U J T Q A Y O P Y W U T A H T O V H O W
Y B J R M Z K C C A B H J A T I J X R Y
F J N Y A Y K F A Q S G D X S J U I M B
W P D V P P N K I Q J L V O O Y C R R K
J V K V P T I A K B H A Q W K A T J X Y
C B R I G I G B X T B J T U U C C L X V
A T D W V R K D D N S B Z I N W B Z G X
N A Z M W D N S A P Q T R I R J B S C L
T C V B W G E Q I I Q M R W J J W C O O
K N J Z T W Q T Z X P T P D Z Y K L U S
```

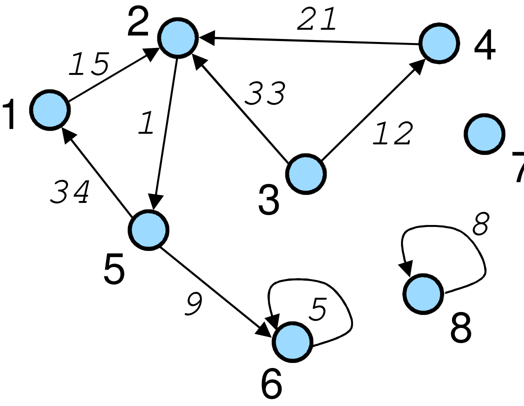
Übungen (II)

- **Aufgabe 2**
  a) Visualisieren Sie den Graphen G = ({1,2,3,4,5,6,7,8}, {{1,2}, {2,7}, {4}, {1,5}, {8}, {1,3}, {3,4}, {4,5}, {5,7}, {4,5}, {7}}.
  b) Welchen Grad hat G?
  c) Was liegt zwischen den Knoten 4 und 5 vor?
  d) Wie nennt man die Knoten {4}, {7} und {8}?
  e) Welchen Grad hat Knoten 5?
  f) Ist (1,3,4,5,1,2,7,5,1) ein Zyklus?
  g) Welche Eigenschaft hat Knoten 6?
  h) Wie viele Zusammenhangskomponenten hat G?
  i) Geben Sie die Adjazenzmatrix von G an!
  j) Geben Sie die Adjazenzliste von G mittels Feldern an!

Übungen (III)

- **Aufgabe 3**

  Gegeben sei der folgende gerichtete Graph G.



a) Geben Sie eine formale Notation des Graphen an!
b) Welchen Eingangsgrad hat Knoten 3?
c) Ist G zyklenfrei?
d) Welchen Ausgangsgrad hat Knoten 5?
e) Gibt es einen Weg von Knoten 4 nach Knoten 1?
f) Geben Sie die Adjazenzmatrix von G an!
g) Geben Sie die Adjazenzliste von G als verzeigerte Struktur an!

Übungen (IV)

- **Aufgabe 4**

  Zeichnen Sie den K_4, den $K_{3,3}$ und den C_5!

  Welche dieser Graphen sind planar?

- **Aufgabe 5**

  a) Zeichnen Sie den zu folgender Adjazenzmatrix gehörenden Graphen!

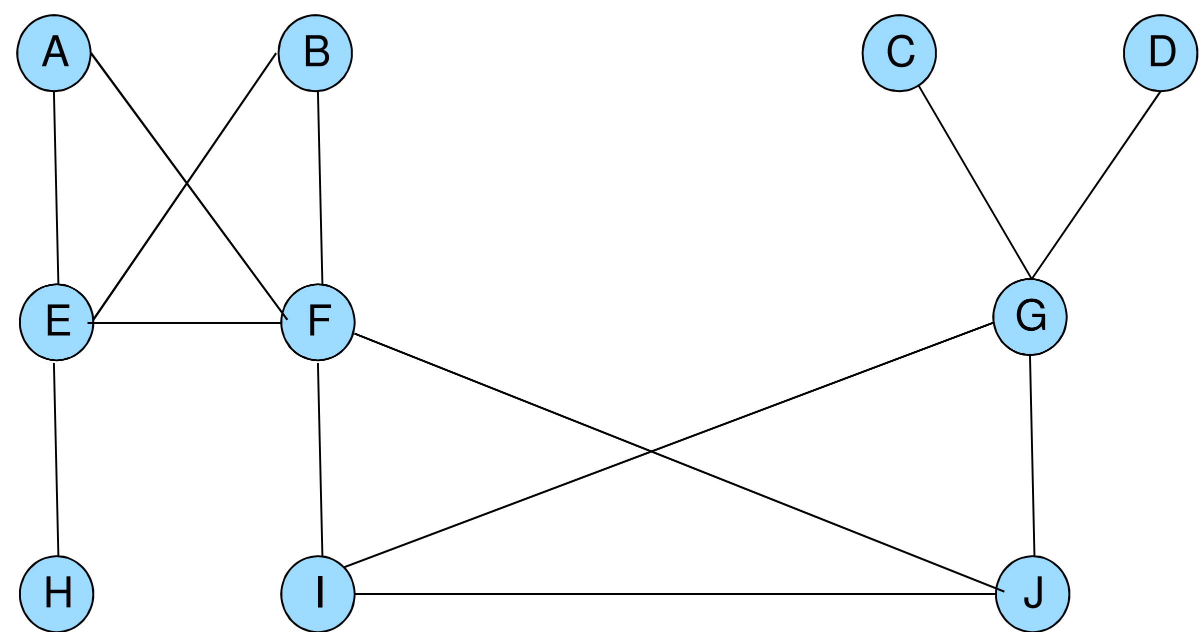
$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

  b) Stellen Sie den Graphen als Adjanzenzliste dar!

Übungen (V)

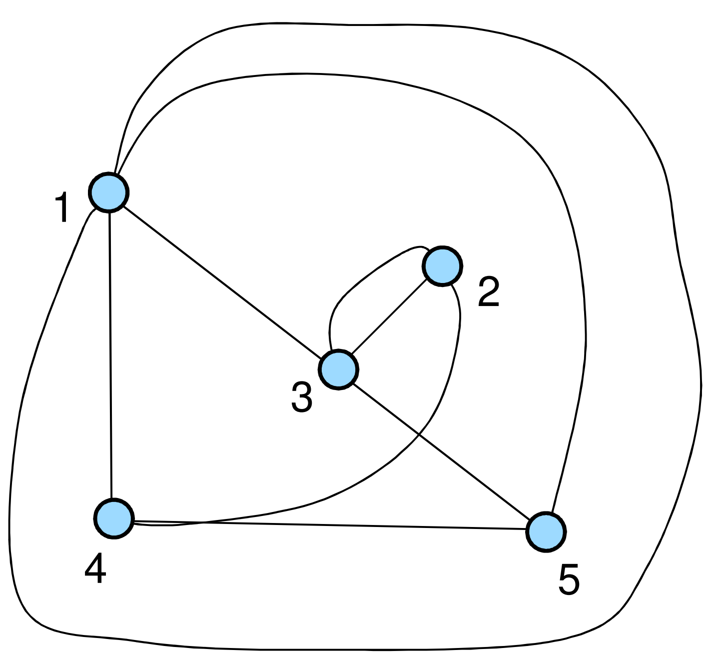
- **Aufgabe 6**

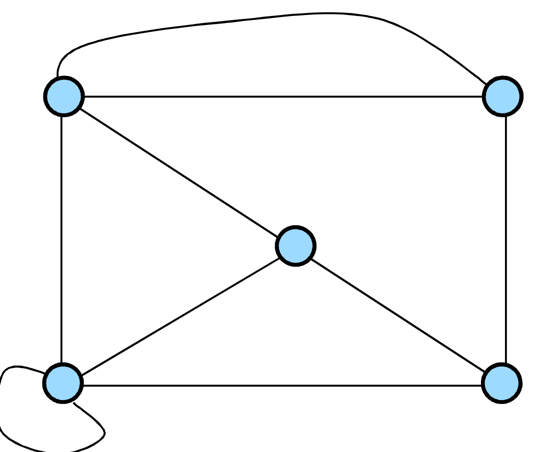
  Berechnen Sie für den folgenden Graphen Rand, Durchmesser, Radius und Zentrum!



Übungen (VI)

- **Aufgabe 7**
  Sind die beiden folgenden Graphen isomorph?





Übungen (VII)

- **Aufgabe 8**

  Lesen Sie sich die folgenden Aussagen genau durch und entscheiden Sie, ob sie wahr oder falsch sind. Eine richtige Antwort wird mit 1 Punkt bewertet, eine falsche Antwort mit einem halben Minuspunkt, gar keine Antwort mit 0 Punkten. Insgesamt kann jedoch bei dieser Aufgabe keine negative Punktzahl erreicht werden.

	Wahr	Falsch
Ein isolierter Knoten darf keine Schlingen haben.		
Zwischen zwei Knoten darf es höchstens eine Kante geben.		
Die Adjazenzmatrix verschenkt bei ungerichteten Graphen ca. die Hälfte des Speicherplatzes.		
Eine Kante umfasst ein bis beliebig viele Knoten.		
In einem Kantenzug kommt keine Kante mehrfach vor.		
Kantenmarkierungen lassen sich in der Adjazenzmatrix einfach darstellen.		
C_3 und K_3 sind identisch.		

Übungen (VIII)

	Wahr	Falsch
Ziel- und Endknoten sind synonyme Begriffe.		
In einem regulären Graphen hat jeder Knoten den gleichen Grad.		
Bipartite Graphen können nicht zu anderen Graphen isomorph sein.		
In Adjazenzlisten wird für jeden Knoten die Liste der Nachbarn abgespeichert.		
Die Beziehung von Knoten zu Kanten wird als Inzidenz bezeichnet.		
Der Eingangsgrad eines Knoten in einem gerichteten Graphen ist die Summe ein- und ausgehender Kanten.		
Zeit und Speicherplatz lassen sich meistens nicht gleichzeitig optimieren.		
C_n bezeichnet einen Kreis mit n Knoten.		
Adjazenzmatrix und –liste lassen sich in konkreten Fällen vorteilhaft um redundante Strukturen erweitern.		

Algorithmen und Datenstrukturen

Teil 4:
Graphen (II): Bäume

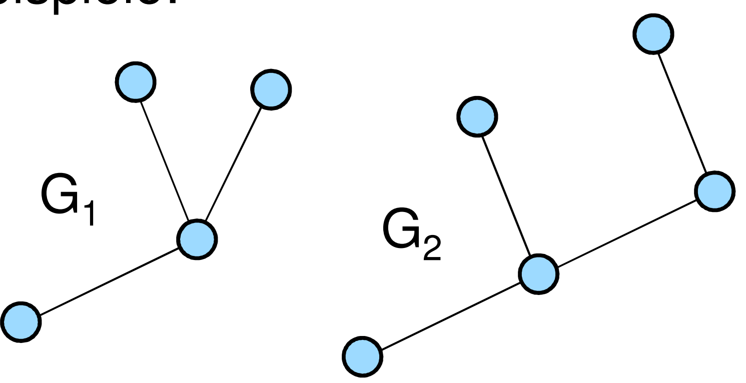
DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 01/2019

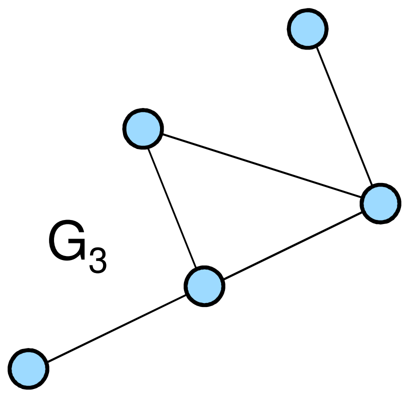
Gliederung

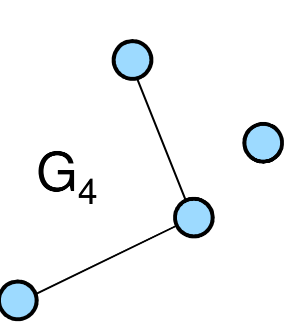
- Definitionen und Eigenschaften
- Datenstrukturen für Bäume
- Anwendungen
- Algorithmen

Bäume

- Ungerichteter Graph ohne geschlossenen Weg heißt Wald
- Zusammenhangkomponenten eines Waldes heißen (ungerichtete) Bäume
- Anders ausgedrückt: (Ungerichteter) Baum ist zusammenhängender ungerichteter Graph, der keinen Zyklus enthält
- Beispiele:



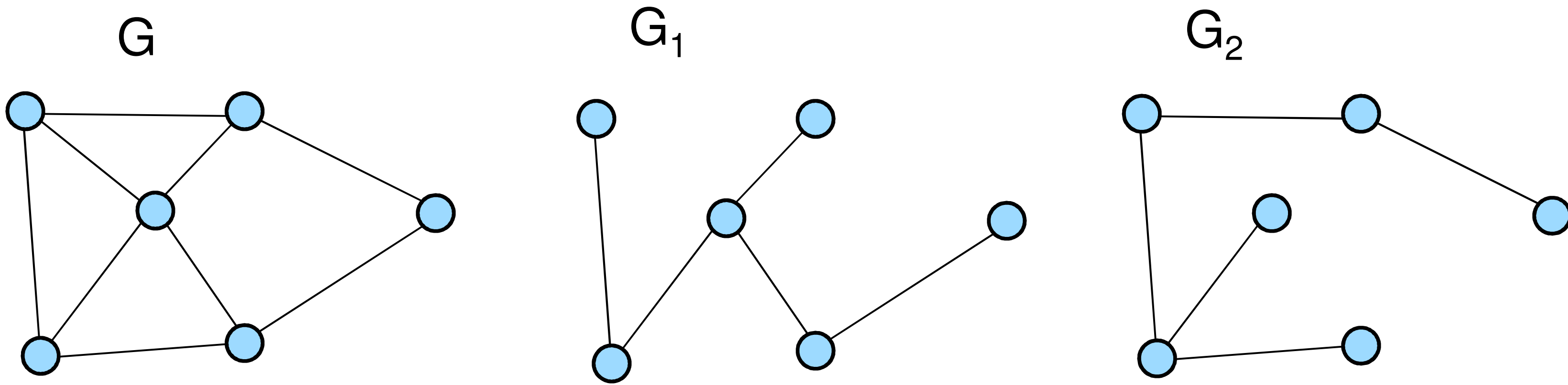




  - G_1 und G_2 sind Bäume
  - G_3 ist kein Baum (enthält Zyklus)
  - G_4 ist Wald mit zwei Bäumen

Aufspannende Bäume

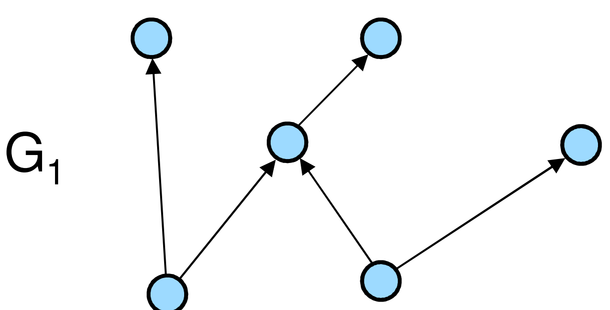
- Wenn G ein Graph ist, dann heißt G‘ aufspannender Baum von G, wenn
  - G‘ hat die gleiche Knotenmenge wie G
  - G‘ enthält Teilmenge der Kantenmenge von G
  - G‘ ist ein Baum
- Beispiel:

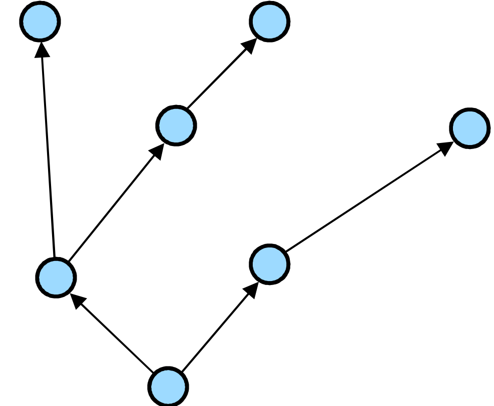


  - G_1 und G_2 sind aufspannende Bäume von G

Gerichtete Bäume und Wurzelbäume (I)

- Gerichteter Graph heißt gerichteter Baum, wenn der zugrundeliegende ungerichtete Graph Baum ist
- Sind v_1 und v_2 Knoten eines gerichteten Baumes und v_1 ist von v_2 aus erreichbar, dann heißt v_2 Vorgänger von v_1 und v_1 Nachfolger von v_2.
- Knoten v heißt Wurzel eines gerichteten Baumes, wenn von v aus jeder andere Knoten erreichbar
- Gerichteter Graph G heißt Wurzelbaum oder Baum, wenn G gerichteter Baum ist und genau eine Wurzel besitzt
- Beispiele:

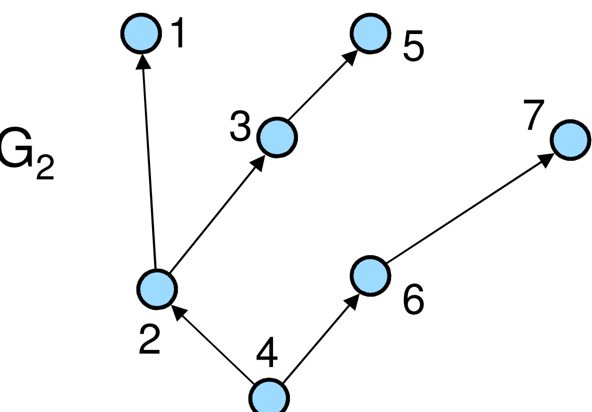




  - G_1 ist ein Baum, aber kein Wurzelbaum
  - G_2 ist ein Wurzelbaum

Gerichtete Bäume und Wurzelbäume (II)

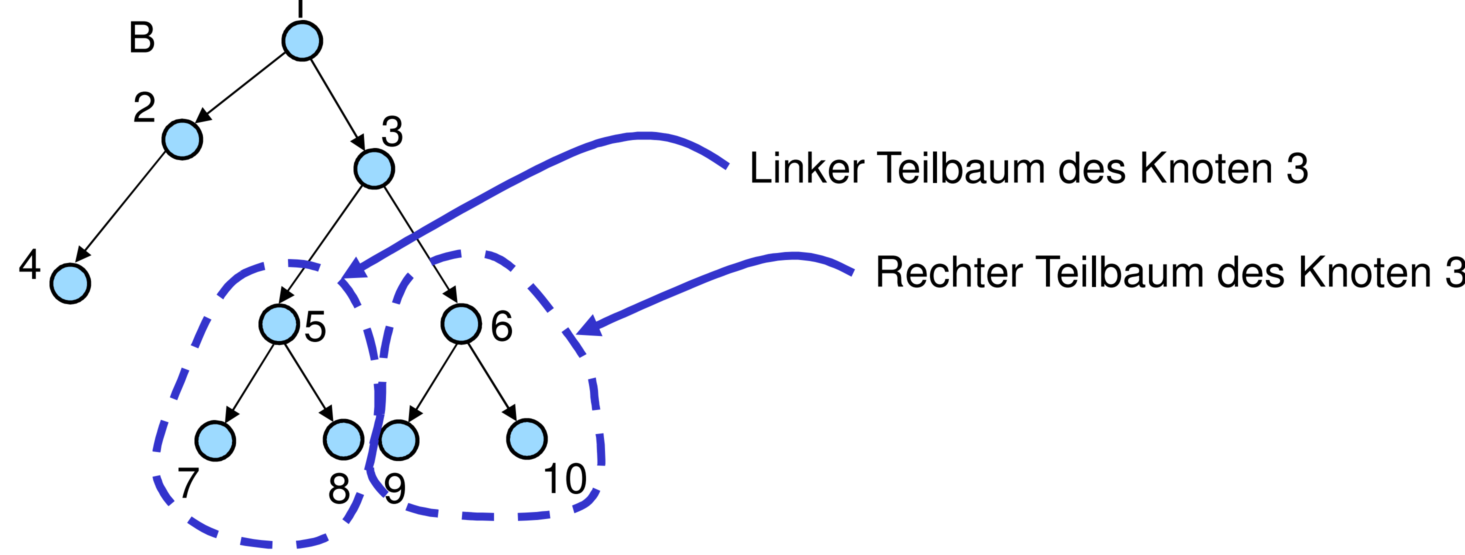
- Knoten eines gerichteten Baumes mit Ausgangsgrad 0 heißt Blatt.
- Andere Knoten des Baumes heißen innere Knoten.
- Niveau (oder Ebene) eines Knotens ist die Entfernung von der Wurzel in Anzahl Kanten + 1.
- Höhe eines Wurzelbaums ist das maximale Niveau eines Knotens.
- Beispiel:



  - Knoten 4 ist Wurzel, Knoten 1,5 und 7 sind Blätter, restliche Knoten innere Knoten
  - Niveau von Knoten 2 und 6 ist 2, Knoten 1, 3 und 7 haben Niveau 3 und der Knoten 5 Niveau 4
  - G_2 besitzt die Höhe 4
- Anmerkung: Häufig wird nicht zwischen Wurzelbaum und Baum unterschieden

Binärbäume

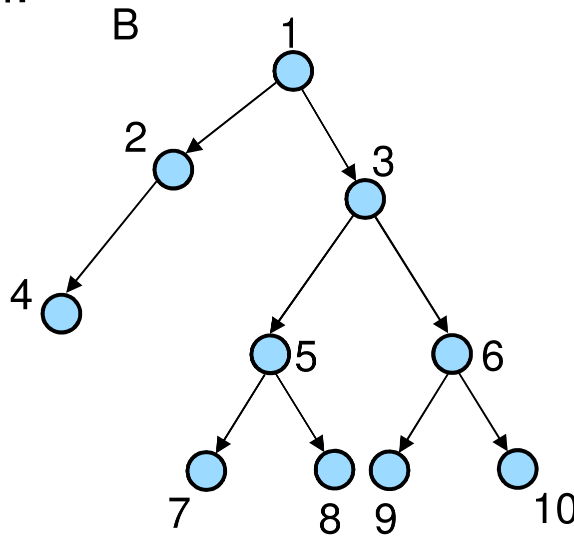
- Wurzelbaum heißt Binärbaum, wenn maximaler Ausgangsgrad eines Knotens 2 ist.
- Nachfolger eines Knoten v bilden Wurzeln des linken und rechten Teilbaums von v.
- Beispiel:



  - B ist Binärbaum

Namenskonvention

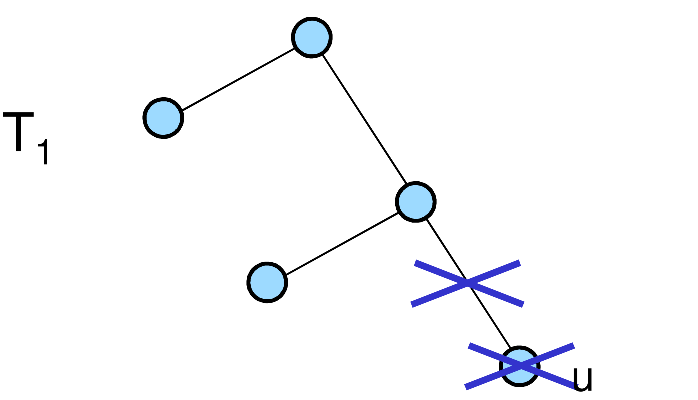
- Statt Nachfolger und Vorgänger wird bei Bäumen häufig von Eltern- und Kindknoten gesprochen
- Benachbarte Knoten heißen dann auch Geschwister
- Beispiel:

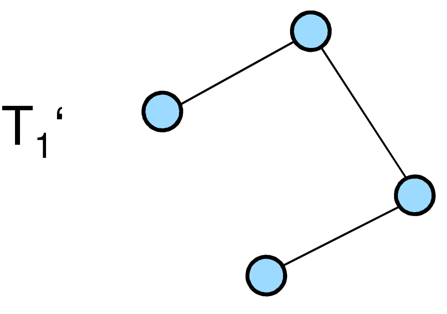


  - 3 ist Elternknoten (oder Vorgänger) von 5 und 6
  - 5 und 6 sind Kindknoten (oder Nachfolger) von 3
  - 5 und 6 sind Geschwisterknoten

Eigenschaften (I)

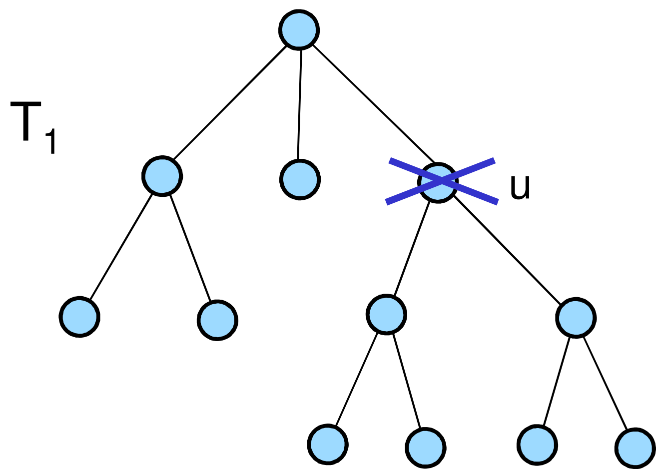
- Sei T=(V,E) Baum. Dann gilt:
  - T hat genau eine Kante weniger als Knoten (Kantenzahl = Knotenzahl – 1, |E| = |V| -1)
  - Wenn |V| ≥ 2 und u ∈ V Blatt, so bleibt T nach Entfernen von u und seinen ausgehenden Kanten ein Baum
  - Beispiel:

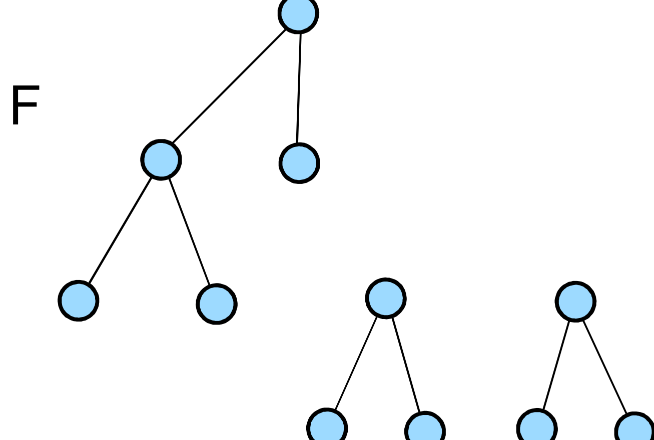




Eigenschaften (II)

- Wenn u ∈ V beliebiger Knoten, so entstehen nach Entfernen von u und seinen ausgehenden Kanten deg(u) Bäume
- Beispiel: Durch Entfernen von u aus T1 entsteht Wald F mit drei Bäumen, weil deg(u) = 3





Anzahl Binärbäume mit n Knoten

- Unterscheidung:
  - Knotennamen relevant (markierte Bäume)
  - Knotennamen irrelevant, nur Form zählt (Strukturell verschiedene Bäume)
- Es gilt:

Knoten	2	3	4	5	6	7	8
Markierte Bäume	1	3	16	125	1296	16807	262144
Knoten-namen irrelevant	2	5	14	42	132	429	1430

- Für markierte Bäume gilt (Satz von Cayley):
  Für $n \geq 2$ Knoten gibt es genau n^{n-2} markierte Bäume.
- Für strukturell verschiedene Bäume gilt (Catalan-Zahlen):

$$b_n = \frac{1}{n+1}\binom{2n}{n} = \begin{cases} 1, & \text{falls } n = 0, \\ \sum_{k=0}^{n-1} b_k b_{n-1-k}, & \text{falls } n > 0. \end{cases}$$

Übersicht

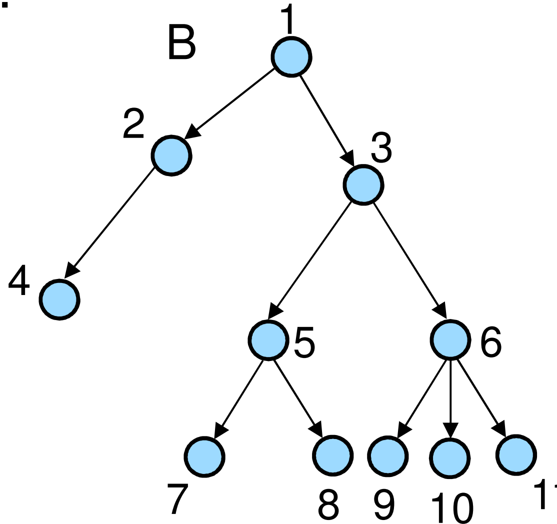
- Definitionen und Eigenschaften
- Datenstrukturen für Bäume
- Anwendungen
- Algorithmen

Motivation

- Bäume sind spezielle Form von Graphen
- Für Graphen bekannte Datenstrukturen auch für Bäume verwendbar
- Aber: Spezialfall lässt Optimierungen zu
- Im Folgenden:
  - Effiziente Strukturen für Wurzelbäume
  - Spezielle Strukturen für Binärbäume
  - Beliebige Bäume als Binärbäume

Wurzelbaum als Feld (I)

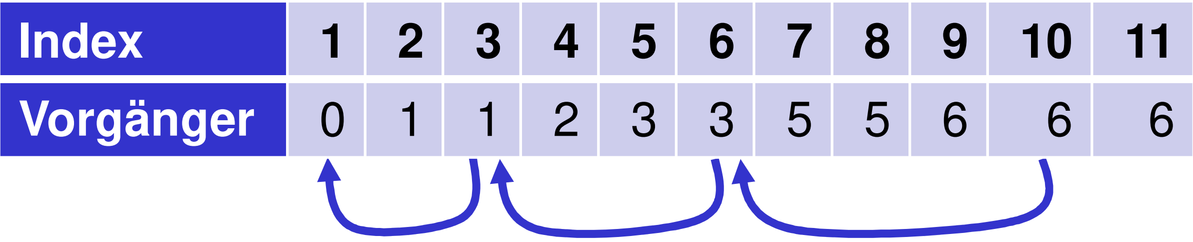
- Wurzelbaum: Jeder Knoten (außer Wurzel) hat genau einen Vorgänger
- D.h. durch Angabe der Vorgänger ist Baum eindeutig bestimmt
- D.h. Feld der Vorgänger ist einfache, kompakte Datenstruktur
- Beispiel:



Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Vorgänger	0	1	1	2	3	3	5	5	6	6	6

Wurzelbaum als Feld (II)

- Vorteil:
  - Abfrage nach Vorgänger (bzw. gesamter Weg zurück zur Wurzel) effizient
  - Beispiel:

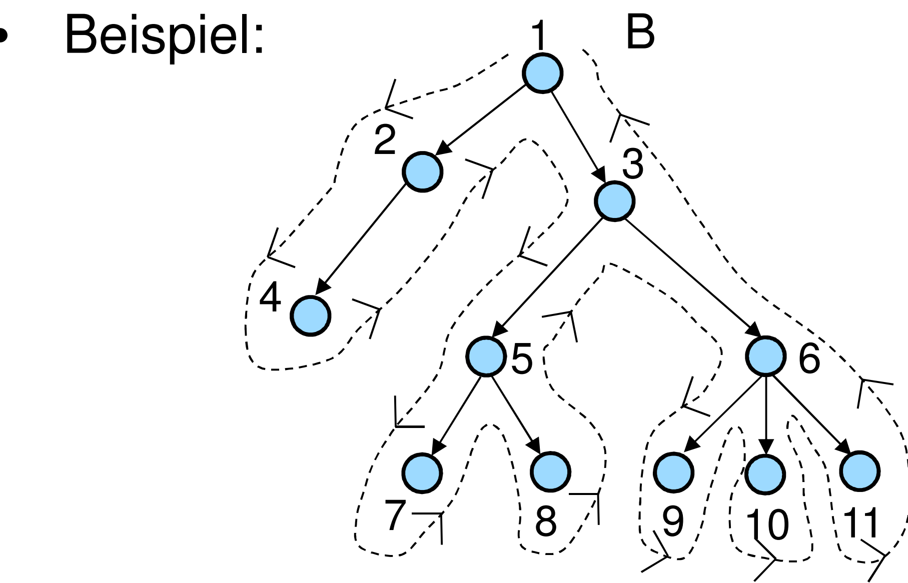


Index	1	2	3	4	5	6	7	8	9	10	11
Vorgänger	0	1	1	2	3	3	5	5	6	6	6

- Nachteil:
  - Weg von Wurzel zu einem bestimmten Knoten ineffizient (ganzes Feld muss durchsucht werden)
  - Keine Ordnungsbäume darstellbar (später in VL mehr)

Wurzelbaum als Binärcode (I)

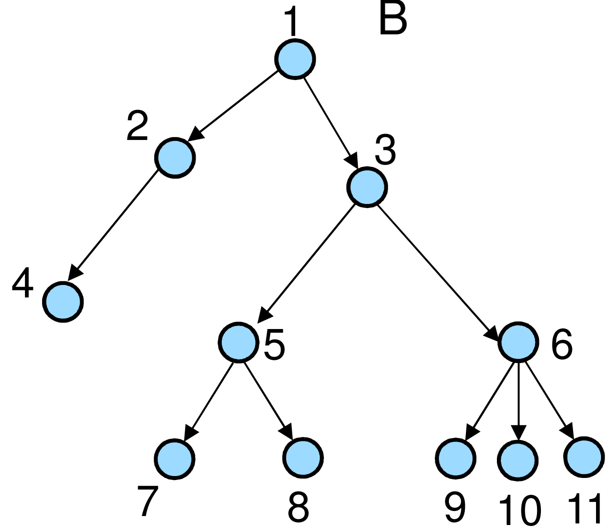
- Bijektion zwischen Wurzelbaum mit n Knoten und Binärcode der Länge 2•n - 2
- Von B zum Binärcode:
  - Starte an Wurzel
  - Umwandere Baum nach links unten (Gestrichelte Linie in Pfeilrichtung)
  - Bei Schritt zu tiefer gelegenem Knoten: Notiere 1
  - Bei Schritt zu höher gelegenem Knoten: Notiere 0
- Beispiel:



Binärcode: 11001110100110101000

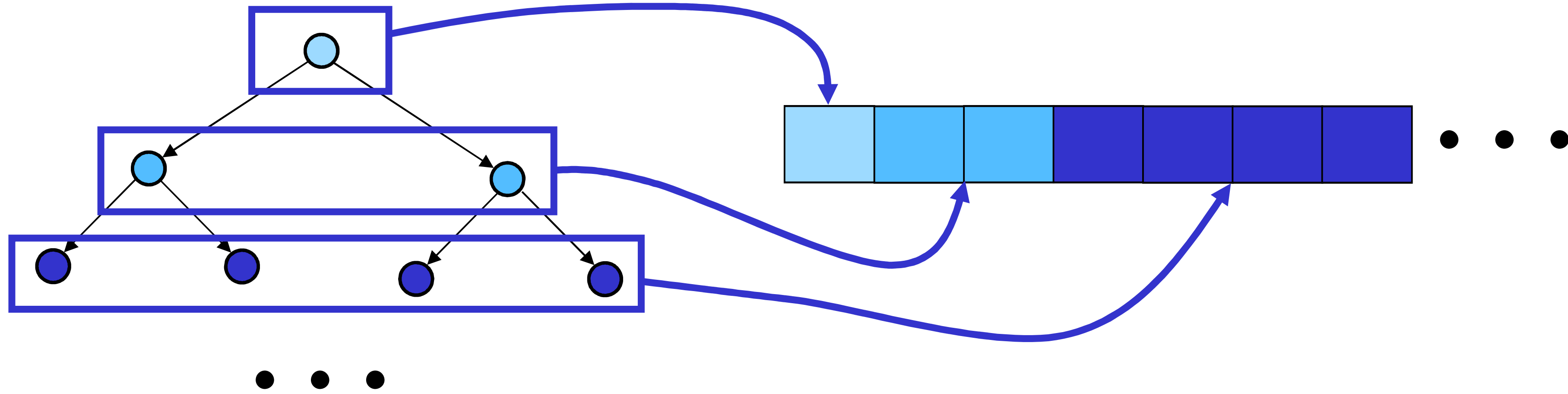
Wurzelbaum als Binärcode (II)

- Vom Binärcode zu B:
  - Starte an Wurzel
  - Gehe Binärcode von links nach rechts durch
  - Bei 1: Zeichne Knoten als Nachfolger von aktuellem Knoten
  - Bei 0: Gehe Schritt zu höher gelegenem Knoten
- Beispiel: Binärcode: 1100111010011010100 0



Binärbaum als Feld (I)

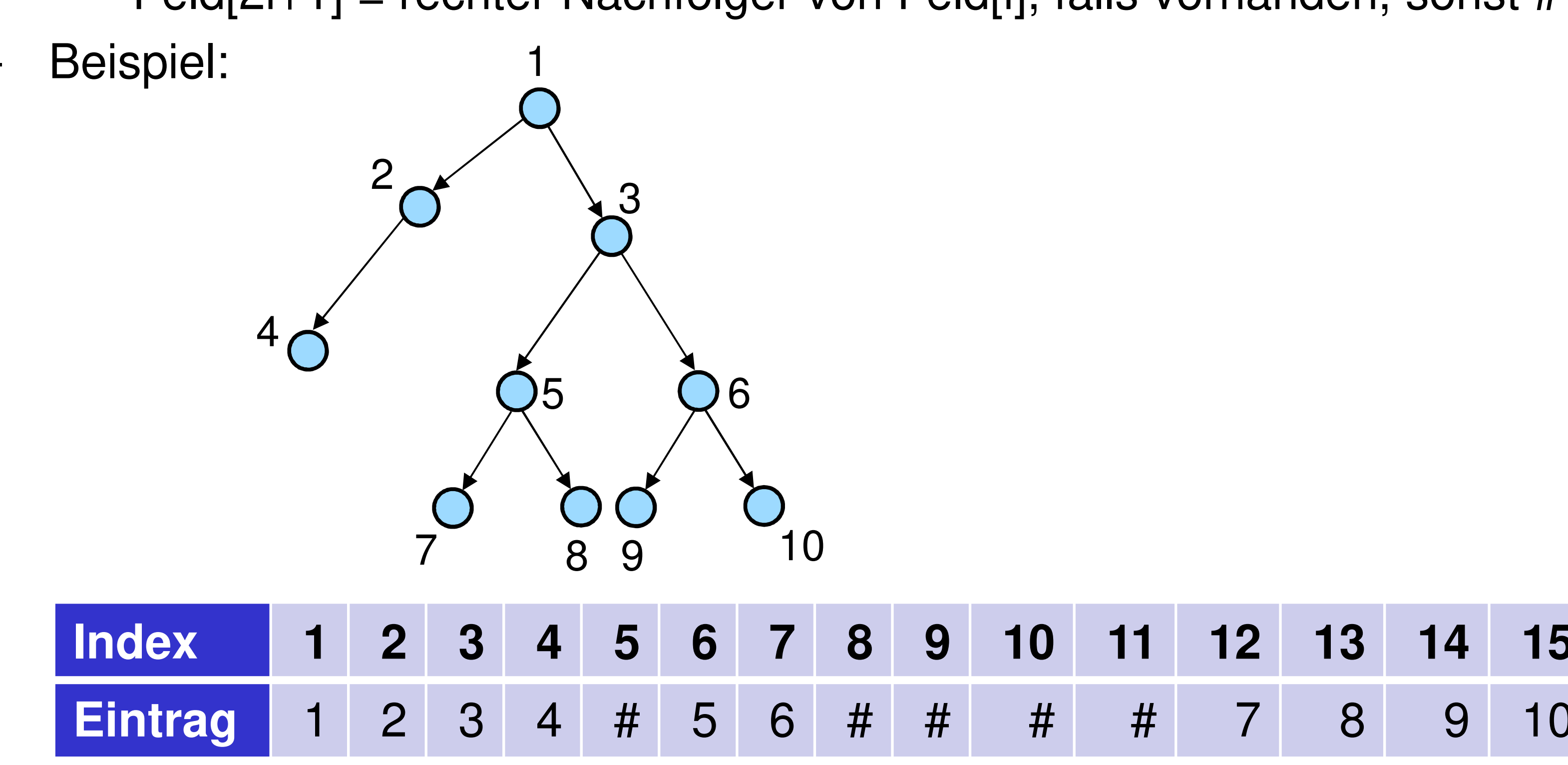
- Binärbäume lassen sich gut als Feld darstellen:
  - Baum der Höhe h hat max. 2^h-1 Knoten
  - i-te Element bei reihenweiser Traversierung wird an Position i gespeichert



- Vorteil: Leichte Navigation in beide Richtungen:
  - Vorgänger(Feld[i]) = Feld[i div 2]
  - Linker Nachfolger(Feld[i]) = Feld[2i]
  - Rechter Nachfolger(Feld[i]) = Feld[2i+1]

Binärbaum als Feld (II)

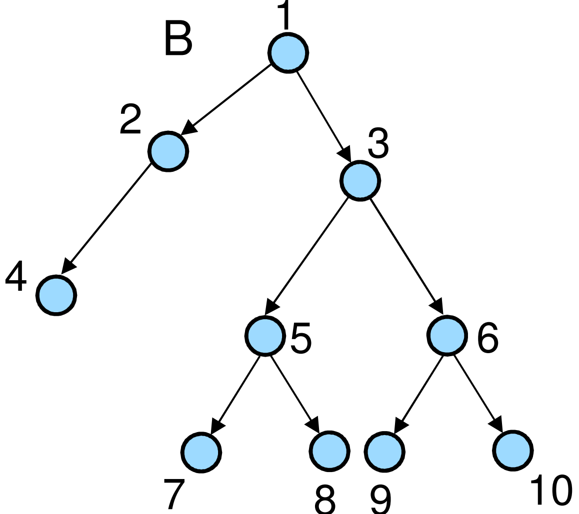
- Unvollständiger Binärbaum:
  - Nehme auch Feld der Länge 2^h-1
  - Trage für nicht vorhandene Knoten Sondersymbol (z.B. #) ein:
    - Feld[1] = Wurzel
    - Feld[2i] = linker Nachfolger von Feld[i], falls vorhanden, sonst #
    - Feld[2i+1] = rechter Nachfolger von Feld[i], falls vorhanden, sonst #
  - Beispiel:



Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Eintrag	1	2	3	4	#	5	6	#	#	#	#	7	8	9	10

Adjazenzliste als Feld

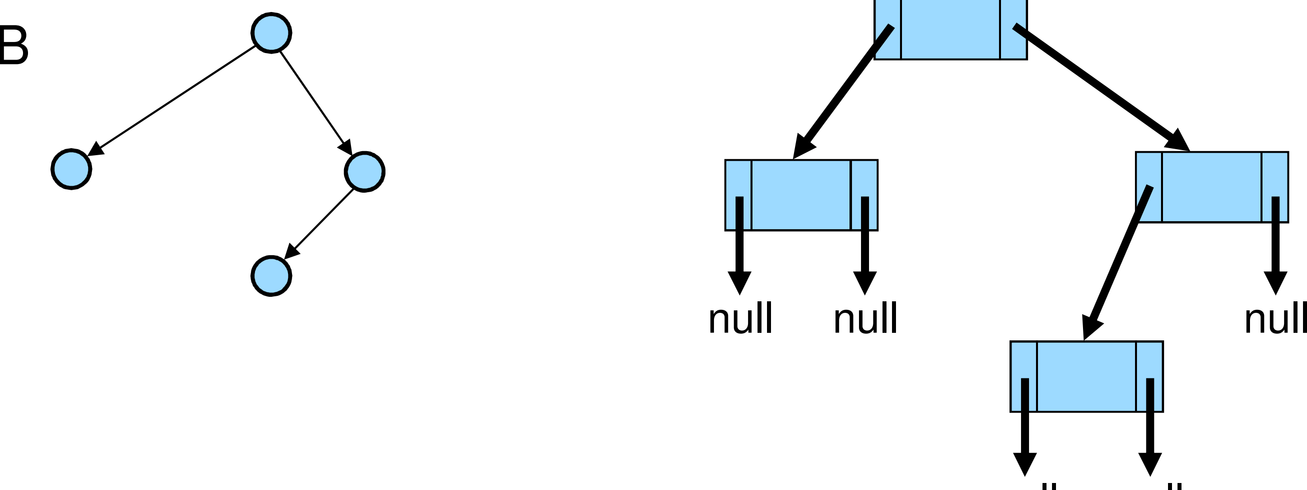
- Da bei Binärbäumen die Listen maximal die Länge 2 haben können, können diese gut als Feld realisiert werden
- Für Binärbäume sehr effiziente Struktur (Navigation in beide Richtungen schnell, ebenso Abfrage nach Wurzel bzw. Blatt)
- Bei jedem Knoten werden die eigentlichen Knoteninformationen (Name und Markierung), Vorgänger und die beiden Nachfolger gespeichert
- Beispiel:



Index	1	2	3	4	5	6	7	8	9	10
Vorgänger	0	1	1	2	3	3	5	5	6	6
Linker Nachfolger	2	4	5	0	7	9	0	0	0	0
Rechter Nachfolger	3	0	6	0	8	10	0	0	0	0

Binärbaum mit Verzeigerungen

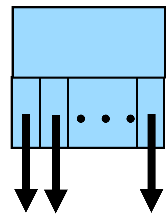
- Für Binärbäume:
  - Jeder Knoten besteht aus Schlüssel und Daten sowie zwei Zeigern, die auf den linken bzw. rechten Nachfolger verweisen
  - Beispiel:



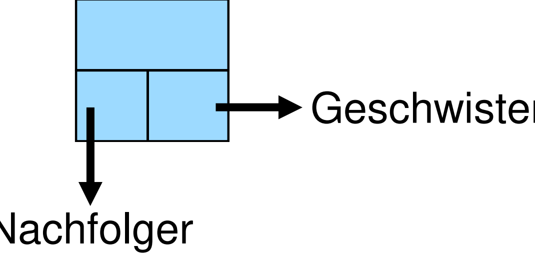
  - Variante/Erweiterung: Auch Rückverweise, wenn Navigation von Blättern zur Wurzel notwendig

Beliebige Bäume als Binärbäume

- Erste Idee:
  - Obergrenze an Nachfolgerknoten festlegen (k)
  - Jeder Knoten damit folgende Struktur:



  - Lösung unschön:
    - Feste Obergrenze
    - Speicherplatzverschwendung
- Daher:
  - Stelle Baum als Binärbaum dar
  - Knotenstruktur:



Beispiel

1
2
3
4
5
6
7
8
9
10
11
12
13

1
null
2
3
4
null
8
null null
9
10
null null
5
6
7
null
null
null null
11
12
13
null
null
null null

Übersicht

- Definitionen und Eigenschaften
- Datenstrukturen für Bäume
- Anwendungen
- Algorithmen

Dateisysteme

- Dateisysteme in Betriebssystemen sind hierarchisch organisiert:
  - Jeder Knoten entspricht einem Verzeichnis/einer Datei
  - Die in einem Verzeichnis enthaltenen Verzeichnisse/Dateien sind seine Nachfolger
  - Leere Verzeichnisse und Dateien sind Blätter des Baumes
- Beispiel:

/
bin usr dev etc tmp
sort cc gerd uli jan
.profile mail c .profile mail text
prim.c a.out Seminar lit

[Tura04]

Compilerbau: Ableitungsbäume

- Compiler-Teilschritt des Parsens: Ist Ausdruck (z.B. `y=x+200*z`) syntaktisch korrekt?
- Knoten sind Symbole, Kanten ihr Zusammenhang
- Innere Knoten sind Nichtterminalsymbole, Blätter sind Terminalsymbole
- Beispiel:

Wertzuweisung

Variable | = | Ausdruck

Ganze Variable | Einfacher Ausdruck

Name | Term | + | Term

y | Faktor | Faktor | * | Faktor

Variable | Vorzeichen-lose Int-Zahl | Variable

Ganze Variable | 200 | Ganze Variable

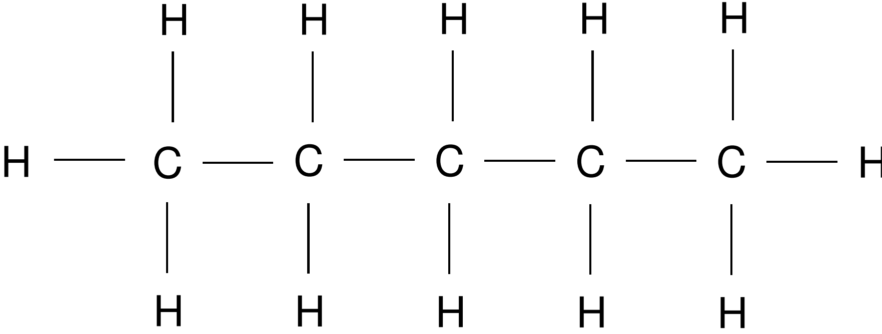
Name | Name

x | z

[Tura04]

Chemie: Darstellung Alkane

- Alkane (Kohlenstoff-Wasserstoff-Verbindungen) lassen sich als Baum darstellen
- Beispiel Pentan:



- Fragestellung: Wie viele verschiedene Pentane gibt es?
- Bzw.: Wie viele nichtisomorphe Bäume mit 5 Knoten von Grad 4 und 12 Knoten vom Grad 1 gibt es?
- Lösung: 3 (Beweis siehe Übung)

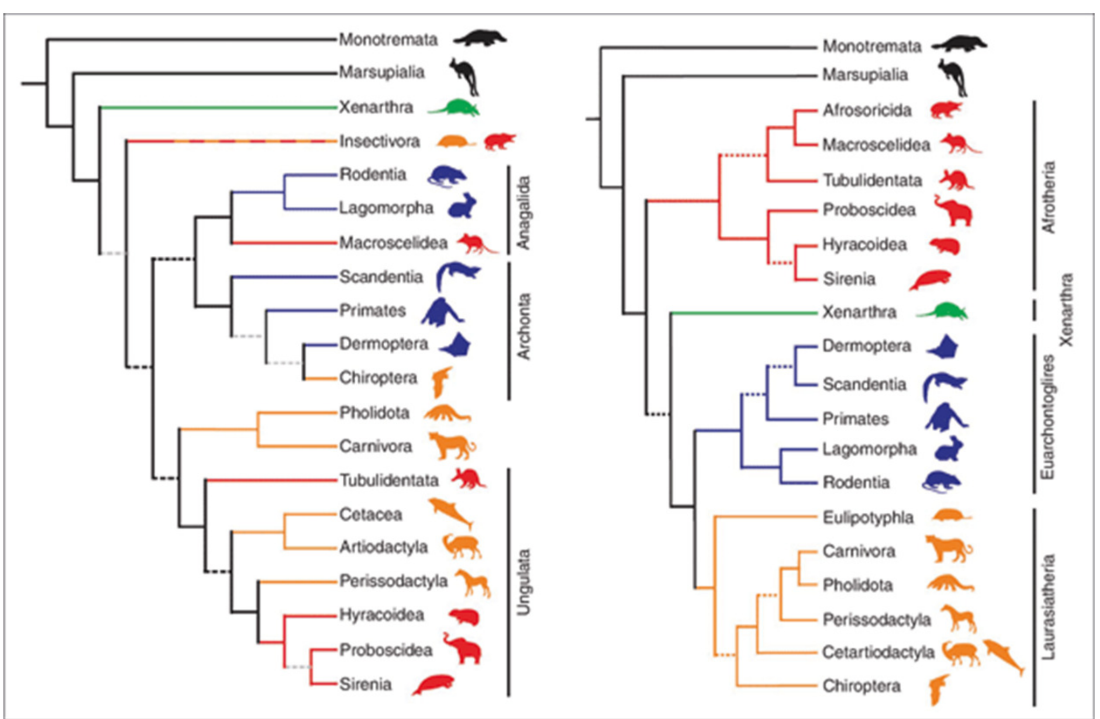
Entscheidungsbäume (I)

- Beispiel:
  - Behälter mit 12 l Wasser in zwei gleiche Teile aufteilen
  - Hilfsmittel: Zwei Behälter mit 8 bzw. 5 l Fassungsvermögen
  - Frage: Wie oft muss man mindestens umfüllen?
- Vorgehen:
  - Wasserverteilungen der drei Behälter bilden Knoten:
    - Von links nach rechts: 12l-, 8l- und 5l-Gefäß
  - Wurzelknoten: 12|0|0
  - Nachfolger:
    - Mögliche Verteilungen durch Umschütten
    - Nur Knoten, die bisher nicht Bestandteil des Baums

Entscheidungsbäume (II)

Phylogenetischer Baum

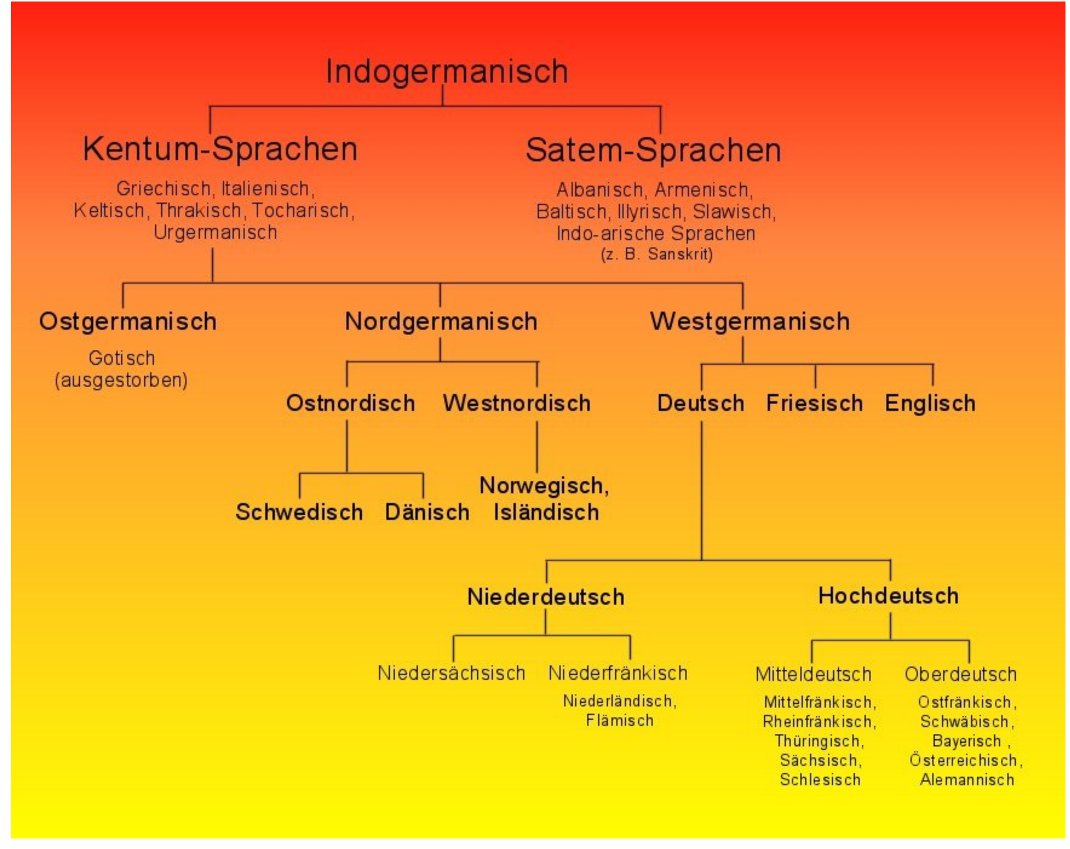
- Geben evolutionäre Entwicklung wieder
- Binäre Verzweigung
- Objekte an Blättern
- Innere Knoten mit oder ohne Label
- Kanten können beschriftet sein (z.B. zeitliche Distanz)
- Beispiel:



[http://www.si-journal.de/index2.php?artikel=jg20/heft2/sij202-s.html]

Entwicklung natürlicher Sprachen

- Binäre oder n-äre Verzweigung
- Innere Knoten Sprachgruppen/-familien
- Blätter aktuelle Sprachen
- Beispiel: Indogermanische Sprachen



http://www.udoklinger.de/Deutsch/Grammatik/Gesch.htm

Übersicht

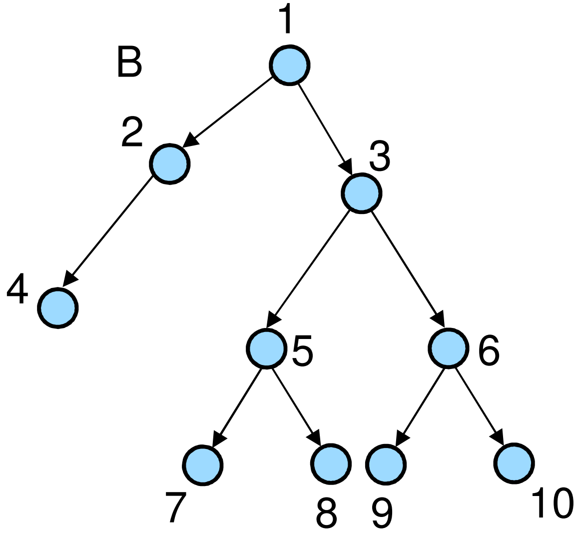
- Definitionen und Eigenschaften
- Datenstrukturen für Bäume
- Anwendungen
- Algorithmen

Traversierung Binärbaum (I)

- Traversierung: Systematisches Abarbeiten aller Knoten
- Drei mögliche Varianten (Einstieg jeweils über Wurzel)
  - Inorder-Durchlauf:
    - Gehe in linken Teilbaum des aktuellen Knoten, führe Inorder rekursiv aus
    - Bearbeite aktuellen Knoten
    - Gehe in rechten Teilbaum des aktuellen Knoten , führe Inorder rekursiv aus
  - Präorder-Durchlauf:
    - Bearbeite aktuellen Knoten
    - Gehe in linken Teilbaum des aktuellen Knoten, führe Präorder rekursiv aus
    - Gehe in rechten Teilbaum des aktuellen Knoten , führe Präorder rekursiv aus
  - Postorder-Durchlauf:
    - Gehe in linken Teilbaum des aktuellen Knoten, führe Postorder rekursiv aus
    - Gehe in rechten Teilbaum des aktuellen Knoten , führe Postorder rekursiv aus
    - Bearbeite aktuellen Knoten

Traversierung Binärbaum (II)

- Beispiel:



  - Inorder-Durchlauf: 4 2 1 7 5 8 3 9 6 10
  - Präorder-Durchlauf: 1 2 4 3 5 7 8 6 9 10
  - Postorder-Durchlauf: 4 2 7 8 5 9 10 6 3 1

Erstellung Phylogenetischer Baum (I)

- Gegeben: Gene von sechs Lebewesen

A ATCGTGGTACTG

B CCGGAGAACTAG

C AACGTGCTACTG

D ATGGTGAAAGTG

E CCGGAAAACTTG

F TGGCCCTGTATC

- Beispiel entnommen aus [https://www.youtube.com/watch?v=09eD4A_HxVQ]

Erstellung Phylogenetischer Baum (II)

- Vorverarbeitung:
  - Schritt 1: Ähnlichkeit feststellen: Ausrichtung (Alignment)

```
A   ATCGTGGTACTG
B   CCGGAGAACTAG
C   AACGTGCTACTG
D   ATGGTGAAAGTG
E   CCGGAAAACTTG
F   TGGCCCTGTATC
```

  - Schritt 2:
    - Sequenzen vergleichen
    - Metrik: Anzahl Unterschiede
    - Beispiel: A und B vergleichen

```
A   ATCGTGGTACTG
B   CCGGAGAACTAG
```

    - Differenz (A,B) = 9

- Quelle: Beispiel entnommen aus [https://www.youtube.com/watch?v=09eD4A_HxVQ]

Erstellung Phylogenetischer Baum (III)

- Schritt 3: Erstellen Differenzmatrix

	A	B	C	D	E	F
A		9	2	4	9	10
B			9	6	2	10
C				5	9	10
D					6	10
E						10
F						

Erstellung Phylogenetischer Baum (IV)

- Algorithmus:
  - Finde zwei ähnlichste Einträge E_1 und E_2
  - Bilde Baum aus neuem Knoten und Nachfolgern E_1 und E_2
  - Berechne neue Distanzmatrix mit $\{E_1, E_2\}$ als ein Eintrag
  - Dabei: $\forall x \in E\ mit\ x \neq E_1 und\ x \neq E_2$:

$$Diff(x, \{E_1, E_2\}) := \frac{Diff(x,E_1) + Diff(x,E_2)}{2}$$

  - Fahre fort, bis Baum erreicht

Erstellung Phylogenetischer Baum (V)

- Erstellen Sie den phylogenetischen Baum!

	a	b	c	d	e	f
a	x	9,0	2,0	4,0	9,0	10,0
b	x	x	9,0	6,0	2,0	10,0
c	x	x	x	5,0	9,0	10,0
d	x	x	x	x	6,0	10,0
e	x	x	x	x	x	10,0
f	x	x	x	x	x	x

	a/c	b	d	e	f
a/c	x	9,0	4,5	9,0	10,0
b	x	x	6,0	2,0	10,0
d	x	x	x	6,0	10,0
e	x	x	x	x	10,0
f	x	x	x	x	x

	a/c	b/e	d	f
a/c	x	9,0	4,5	10,0
b/e	x	x	6,0	10,0
d	x	x	x	10,0
f	x	x	x	x

	a/c/d	b/e	f
a/c/d	x	6,75	10,0
b/e	x	x	10,0
f	x	x	x

	a/b/c/d/e	f
a/b/c/d/e	x	8,375
f	x	x

A C D B E F

"C:\Users\dh10mbo\OneDrive - Durr Group\Documents\Uni\S2\Algos\PhylogenetischerBaum.xlsx"

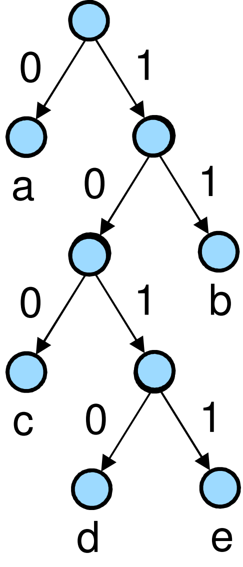
Datenkompression (I)

- Verlustfreie Kompression:
  - Reduzierung der Datenmenge unter Beibehaltung des Informationsgehaltes
  - Ursprüngliche Daten müssen sich aus komprimierten Daten wieder rekonstruieren lassen
- Gegenteil: Verlustbehaftete Kompression

- Idee: Einzelne Zeichen nicht durch Codes konstanter Länge darstellen (wie z.B. im ASCII-Code oder Unicode), sondern die Häufigkeit der Zeichen berücksichtigen
- Häufiger vorkommende Zeichen erhalten dann einen kürzeren Code als die weniger häufig vorkommenden
- Also: Zeichen nach ihrer Häufigkeit sortieren und dann Binärwerte zuordnen, erstes Zeichen erhält 0, zweites 1, drittes 00, viertes 01, ...
- Problem: Rekonstruktion unmöglich (Was bedeutet „00"?)
- Lösung bietet sog. Präfix-Code: Die Darstellung eines Zeichens ist niemals der Anfang der Darstellung eines anderen Zeichens

Datenkompression (II)

- Darstellung Präfix-Code als Binärbaum:
  - Zeichen Blätter, Codierung Weg von Wurzel zum Blatt
  - Kante zu linkem Nachfolger wird mit 0 markiert, Kante zu rechtem Nachfolger mit 1
- Beispiel: Zeichen {a,b,c,d,e}



Zeichen	Code
a	0
b	11
c	100
d	1010
e	1011

  - Code(„bade“) = 11010101011
  - Decode(010101010111010) = „addbd“

Datenkompression (III)

- Für gegebene Zeichenmenge Präfix-Code nicht eindeutig
- Bewertung: Was ist „guter“ Code?
  - Erste Idee: Maximale Länge eines Codeworts
  - Besser: Mittlere Codewortlänge $l = \sum_{i=1}^{n} p_i \cdot l_i$
  - l_i ist die Länge des i-ten Codeworts, p_i die Häufigkeit seines Auftretens
- Gesucht: Algorithmus, der Präfix-Code mit minimaler mittlerer Wortlänge systematisch konstruiert
- Huffman-Algorithmus:
  - Erzeugt Huffman-Code
  - Aufwand O(n·logn)
  - Vorgeschlagen von David A. Huffman (1952)
- Anmerkung: (Noch) besserer Komprimierungsgrad wird erreicht, wenn nicht einzelne Zeichen, sondern Paare, Tripel oder größere Gruppen von Zeichen der Ausgangsmenge zusammen codiert werden

Huffman-Algorithmus: Prinzip

- Voraussetzungen:
  - Gegeben ist ein Text mit n unterschiedlichen Zeichen
  - Zeichen $a_1, a_2, ..., a_n$ bilden das Alphabet A des Textes
  - Relative Häufigkeit des Auftretens des Zeichens a_i wird mit h_i bezeichnet
  - Dabei gilt: $h_1 + h_2 + \ldots + h_n = 1$
- Vorgehen:
  - Erzeuge für jedes Zeichen einen Wurzelbaum, der aus nur einem Knoten besteht, markiere diesen mit dem Zeichen und der relativen Häufigkeit
  - Die beiden Wurzelbäume mit den geringsten relativen Häufigkeiten werden zu einem neuen Baum vereint, wobei die Wurzel des neuen Baums die summierte relative Häufigkeit der beiden zusammengefassten Bäume bekommt
  - Fahre mit diesem Schritt solange fort, bis nur noch ein Baum existiert
  - Beschrifte jeweils die linke Kante mit 0 und die rechte mit 1
  - Präfix-Code kann nun direkt abgelesen werden
  - Anm. zu Schritt 2: Gibt es mehrere Auswahlmöglichkeiten für die Bäume mit geringster relativer Häufigkeit, können beliebige dieser Bäume genommen werden

Huffman-Algorithmus: Aufgabe (I)

- Gegeben sei der folgende Text, für den ein Präfix-Code mit minimaler mittlerer Codewortlänge konstruiert werden soll:

  `AGNES HAT ANGST.`

- Daraus ergibt sich:

Zeichen	A	E	G	H	N	S	T	<Leer>	.
Absolute Häufigkeit	3	1	2	1	2	2	2	2	1
Relative Häufigkeit	$\frac{3}{16}$	$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

- Führen Sie den Huffman-Algorithmus aus!

Huffman-Algorithmus: Aufgabe (VIII)

- Codieren:
  - Codiere(AGNES HAT ANGST) = 01010011001101110010111010000001010110100111000101
- Decodieren:
  - Decodiere(010110110011000111100001100111000101) = ???

Zusammenfassung (I)

- Definitionen:
  - Wald, Baum, Wurzelbaum
  - Wurzel, Blätter, Innere Knoten
  - Niveau, Höhe

- Datenstrukturen für Bäume:
  - Wurzelbaum als Feld
  - Binärbäume mit zwei Verweisen
  - Binärbaum: Adjazenzliste als Feld
  - Binärbaum mit Verzeigerungen
  - Beliebige Bäume als Binärbäume

Zusammenfassung (II)

- Anwendungen:
  - Dateisysteme
  - Compiler
  - Alkane
  - Entscheidungsbäume
  - Präfix-Code

- Algorithmen:
  - Traversierung:
    - Präorder, Inorder, Postorder
  - Konstruktion Phylogenetischer Baum
  - Kompression:
    - Huffman-Algorithmus: Auffinden optimaler Präfix-Code

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
N R A T L L J P I L S V W D Z H R Z O R
T E I O E S S L K X C T M R X B E J Y S
Z Y T Z L E D O C N A M F F U H D T U E
A Q R O P P L J O O J I M J N V R T T A
M U H U N I L W U B G W R B L X O E Y Y
W O R H T K X I K K D O I A N Z T P Z U
A F E C V Y D P E R M F A E D A S N D W
H Q L H F L I N B W U S Q Z C C O B O G
G B I Y R P M E I W K A B E W T P T Q F
C U D V B L A T T K Y P D Z M F Z S T I
E W E N L Q H W E C A Q G C U H T M F T
M I X Z C V P R B Y J I N R I M K J J E
A M H W S I D H U J S V R L L Y N N N A
U P B I E S X H Y N H B H U X F A I K U
P E S O E W E N J I J T N K N I C I D N
N Y T P X J B G O X F U B D L L K L U U
M M I L S D K J D T D S N J K U T L W M
T U H J D P Z B T Y Y S U I O P S Q I L
F H A K H P X S T I O L Y U F N T U O X
P C F I E C F T Z Q Y Q H G A B I K Y I
```

Übungen (II)

- **Aufgabe 2**

  Für den Zeichenvorrat {v, w, x, y, z} liegen die folgenden Präfix-Codes und die angegebenen Häufigkeiten des Auftretens vor:

Zeichen	Code 1	Code 2	Häufigkeit
v	0	00	35%
w	11	01	20%
x	100	10	30%
y	1010	110	10%
z	1011	111	5%

a) Geben Sie die mittlere Codewortlänge der beiden Codierungen an. Was besagen die Werte?

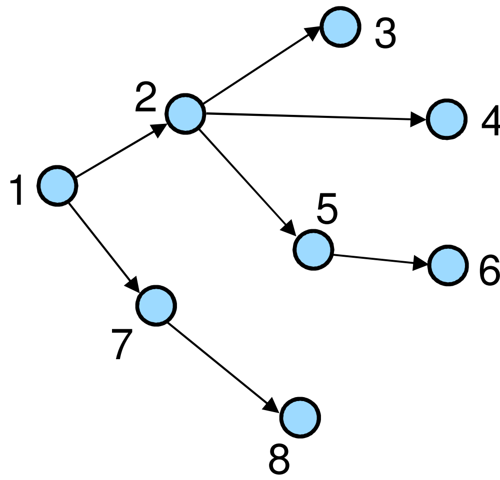
b) Zeichnen Sie Binärbäume der beiden Codes!

c) Durch welche Idee lassen sich Codes mit höherem Komprimierungsgrad erreichen?

Übungen (III)

- **Aufgabe 3**

  Gegeben sei der folgende Baum B.

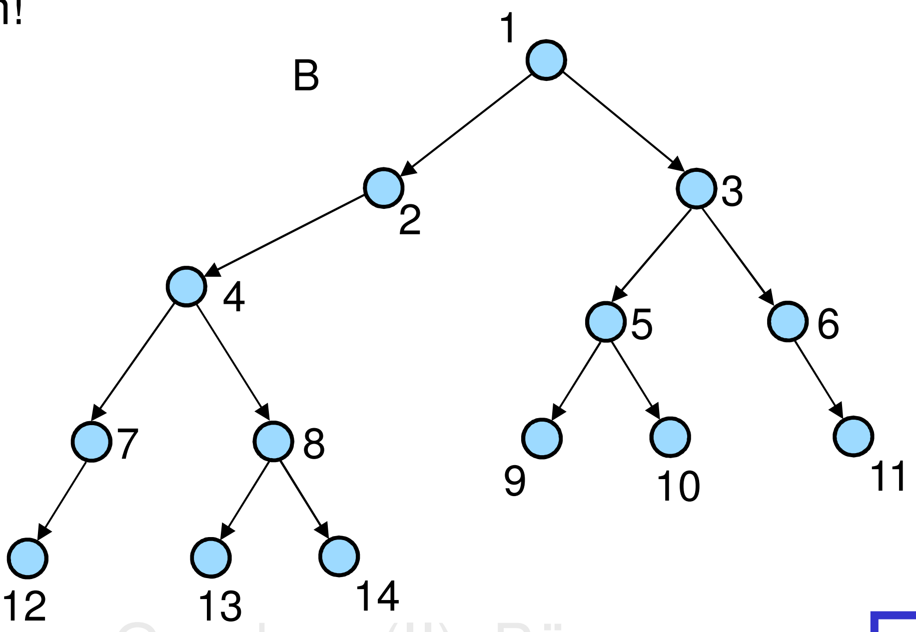


a) Welcher Knoten bildet die Wurzel?
b) Wie viele Blätter hat B?
c) Welches Niveau hat Knoten 4?
d) Ist B ein Binärbaum? Warum (nicht)?
e) Welche Höhe hat B?

Übungen (IV)

- **Aufgabe 4**
  a) Welche Aussagen lassen sich über die Adjazenzmatrix eines Wurzelbaums treffen?
  b) Welche Aussagen lassen sich über die Adjazenzmatrix eines Baums treffen?
  c) Welche Aussagen lassen sich über die Adjazenzmatrix eines Binärbaums treffen?
- **Aufgabe 5**

  Geben Sie für den folgenden Binärbaum B die Knotenfolge von Prä-, In- und Postorder-Durchlauf an!



Übungen (V)

- **Aufgabe 6**

  Bestimmen Sie durch Anwendung des Huffman-Algorithmus einen Präfix-Code für die Zeichen a, b, c, d, e und f mit den Häufigkeiten 0.12, 0.32, 0.04, 0.2, 0.16 und 0.16.

  Bestimmen Sie die mittlere Codewortlänge!

- **Aufgabe 7**

  a) Geben Sie die drei unterschiedlichen Pentane an!

  b) Wie viele unterschiedliche Hexane gibt es? Geben Sie diese an!

Übungen (VI)

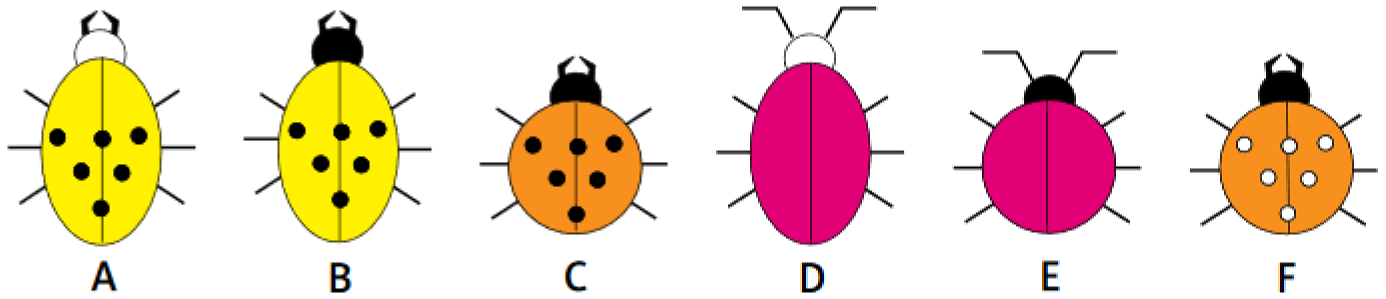
- **Aufgabe 8**

	Wahr	Falsch
In einem Binärbaum hat jeder Knoten zwei ausgehende Kanten.		
Bei der Ausgabe von Dateien aus aktuellem Verzeichnis und allen Unterverzeichnissen kann das Postorder-Verfahren angewendet werden.		
Für einen gegebenen Zeichenvorrat ist der Präfix-Code immer eindeutig.		
Bei einem Postorder-Durchlauf in einem Binärbaum wird die Wurzel stets zuletzt besucht.		
Bäume können keine isolierten Knoten haben.		
Ein Baum ist immer zyklenfrei.		
Schlingen können in Bäumen vorkommen.		
Der Huffman-Algorithmus berechnet einen Präfix-Code mit minimaler Codewortlänge.		

Übungen (VII)

- **Aufgabe 9**

  Gegeben seien die folgenden sechs Käfer:



Erstellen Sie einen Phylogenetischen Baum!

Gehen Sie folgendermaßen vor:

- Erstellen Sie eine Tabelle mit den Merkmalen „Körperform“, „Farbe“, „Kopffarbe“, „Fühler vorhanden“, „Kiefer sichtbar“, „Punkte auf Rücken“, „Punktfarbe“
- Erstellen Sie basierend hierauf eine Differenzmatrix
- Erstellen Sie den Baum

Übungen (VIII)

- **Aufgabe 10**

  a) Wie viele markierte und wie viele strukturell unterschiedliche Binärbäume mit 4 Knoten gibt es?

  b) Zeichnen Sie die strukturell unterschiedlichen Bäume aus Aufgabe a)!

  c) Erläutern Sie die Formel für die Anzahl strukturell unterschiedlicher Binärbäume!

Algorithmen und Datenstrukturen

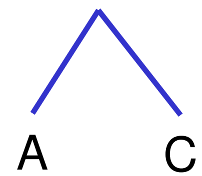
Teil 4:
Graphen (II): Bäume
Lösungen der Stift-Aufgaben

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 01/2019

Erstellung Phylogenetischer Baum (V)

- Durchgang 1:

	A	B	C	D	E	F
A		9	2	4	9	10
B			9	6	2	10
C				5	9	10
D					6	10
E						10
F						



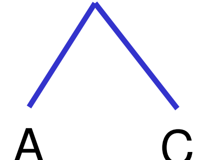
Neue Matrix:

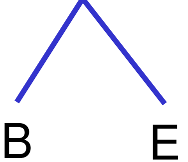
	{A,C}	B	D	E	F
{A,C}		9	4,5	9	10
B			6	2	10
D				6	10
E					10
F					

Erstellung Phylogenetischer Baum (VI)

- Durchgang 2:

	{A,C}	B	D	E	F
{A,C}		9	4,5	9	10
B			6	2	10
D				6	10
E					10
F					





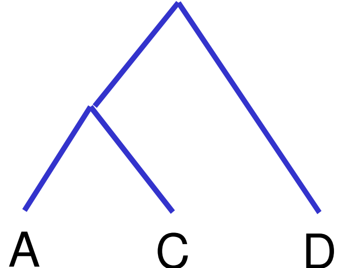
Neue Matrix:

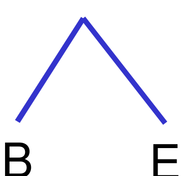
	{A,C}	{B,E}	D	F
{A,C}		9	4,5	10
{B,E}			6	10
D				10
F				

Erstellung Phylogenetischer Baum (VI)

- Durchgang 3:

	{A,C}	{B,E}	D	F
{A,C}		9	4,5	10
{B,E}			6	10
D				10
F				





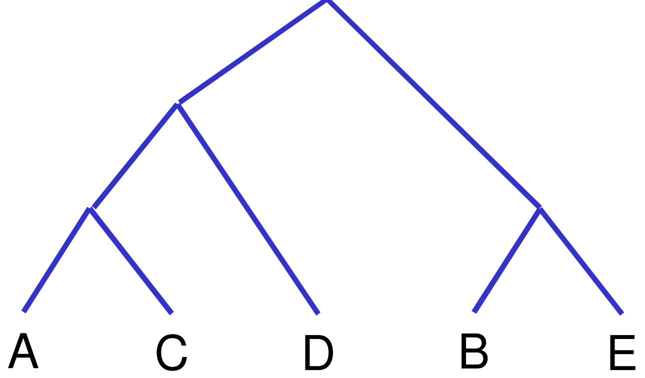
Neue Matrix:

	{A,C,D}	{B,E}	F
{A,C,D}		7,5	10
{B,E}			10
F			

Erstellung Phylogenetischer Baum (VII)

- Durchgang 4:

	{A,C,D}	{B,E}	F
{A,C,D}		7,5	10
{B,E}			10
F			

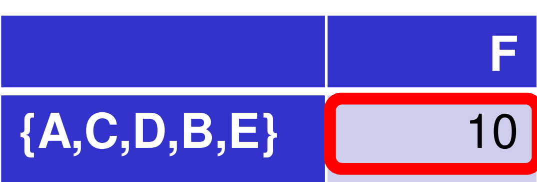


Neue Matrix:

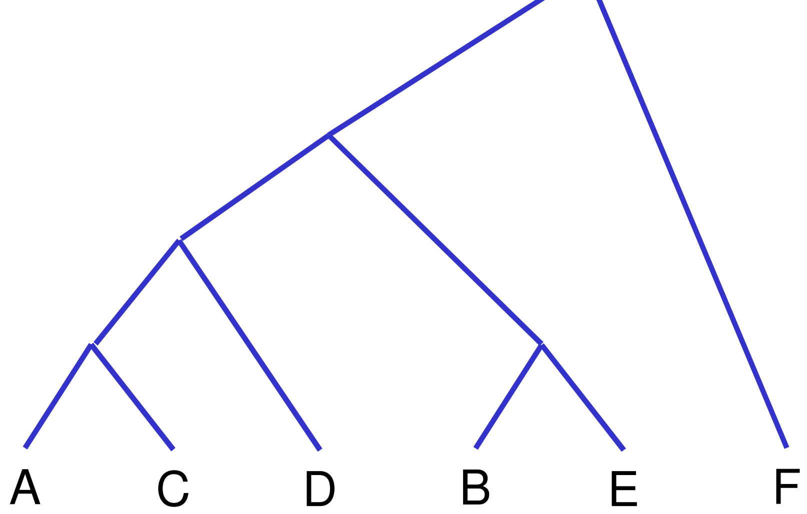
	F
{A,C,D, B,E}	10

Erstellung Phylogenetischer Baum (VIII)

- Durchgang 5:



	F
{A,C,D,B,E}	10



- Algorithmus terminiert

Huffman-Algorithmus: Aufgabe (I)

- Gegeben sei der folgende Text, für den ein Präfix-Code mit minimaler mittlerer Codewortlänge konstruiert werden soll:

```
AGNES HAT ANGST.
```

- Daraus ergibt sich:

Zeichen	A	E	G	H	N	S	T	<Leer>	.
Absolute Häufigkeit	3	1	2	1	2	2	2	2	1
Relative Häufigkeit	$\frac{3}{16}$	$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

- Schritt 1:

A $\frac{3}{16}$ E $\frac{1}{16}$ G $\frac{2}{16}$ H $\frac{1}{16}$ N $\frac{2}{16}$ S $\frac{2}{16}$ T $\frac{2}{16}$ <Leer> $\frac{2}{16}$. $\frac{1}{16}$

Huffman-Algorithmus: Aufgabe (II)

DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart
Campus Horb

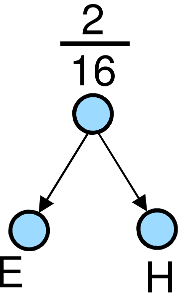
- Schritt 2:
  - E und H haben die niedrigsten Werte und werden daher zusammengefasst

$\frac{3}{16}$ A — $\frac{2}{16}$ (E, H) — $\frac{2}{16}$ G — $\frac{2}{16}$ N — $\frac{2}{16}$ S — $\frac{2}{16}$ T — $\frac{2}{16}$ <Leer> — $\frac{1}{16}$.

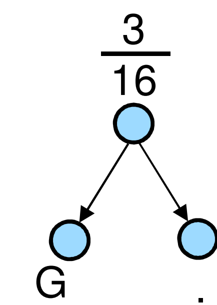
- Schritt 3:
  - G und . haben die niedrigsten Werte und werden daher zusammengefasst

$\frac{3}{16}$ A

$\frac{2}{16}$



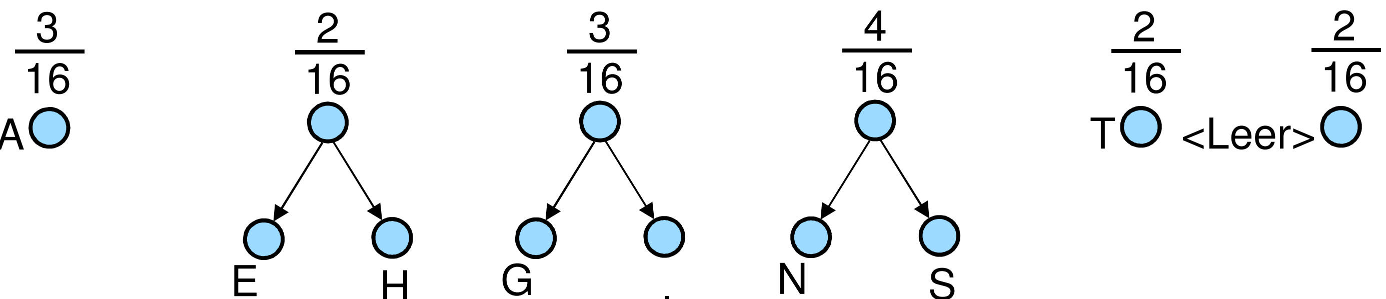
$\frac{3}{16}$



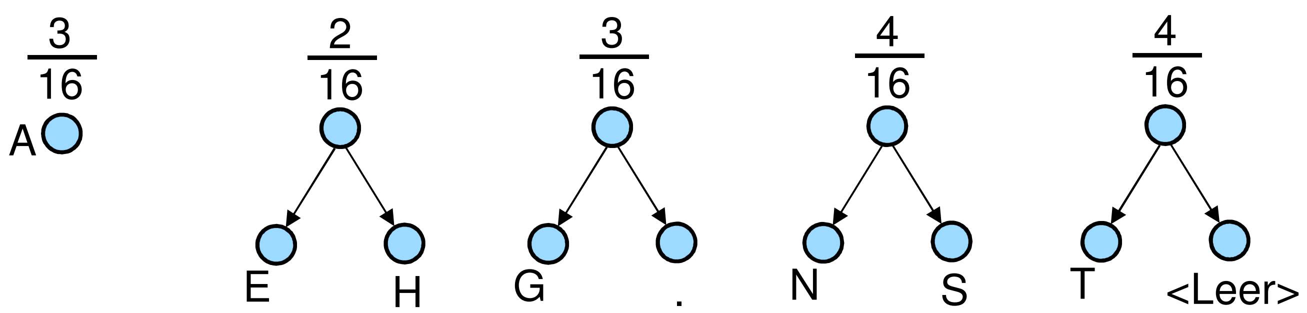
$\frac{2}{16}$ N — $\frac{2}{16}$ S — $\frac{2}{16}$ T — $\frac{2}{16}$ <Leer>

Huffman-Algorithmus: Aufgabe (III)

- Schritt 4:
  - N und S haben die niedrigsten Werte und werden daher zusammengefasst

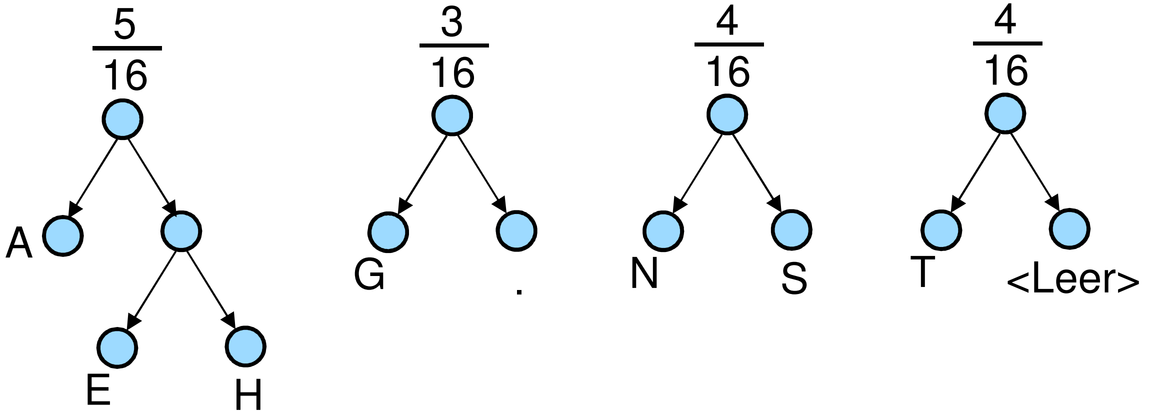


- Schritt 5:
  - T und <Leer> haben die niedrigsten Werte und werden daher zusammengefasst

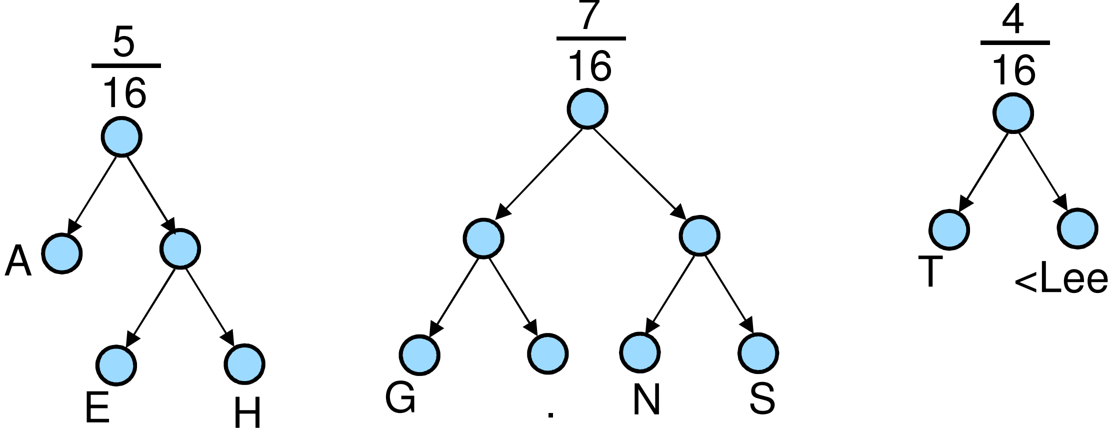


Huffman-Algorithmus: Aufgabe (IV)

- Schritt 6:
  - A und EH haben die niedrigsten Werte und werden daher zusammengefasst

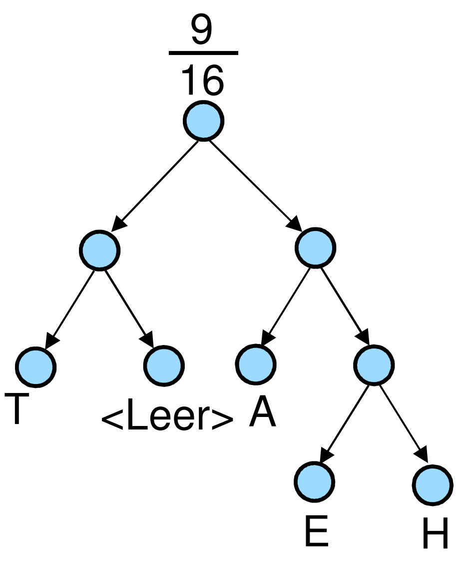


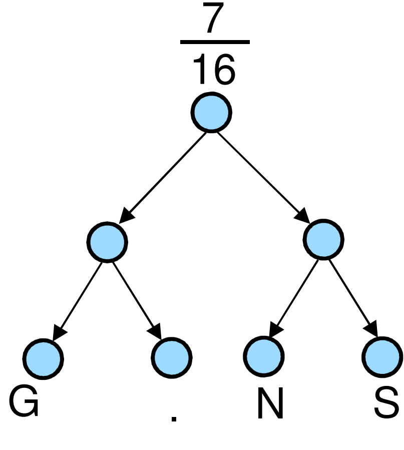
- Schritt 7:
  - G. und NS haben die niedrigsten Werte und werden daher zusammengefasst



Huffman-Algorithmus: Aufgabe (V)

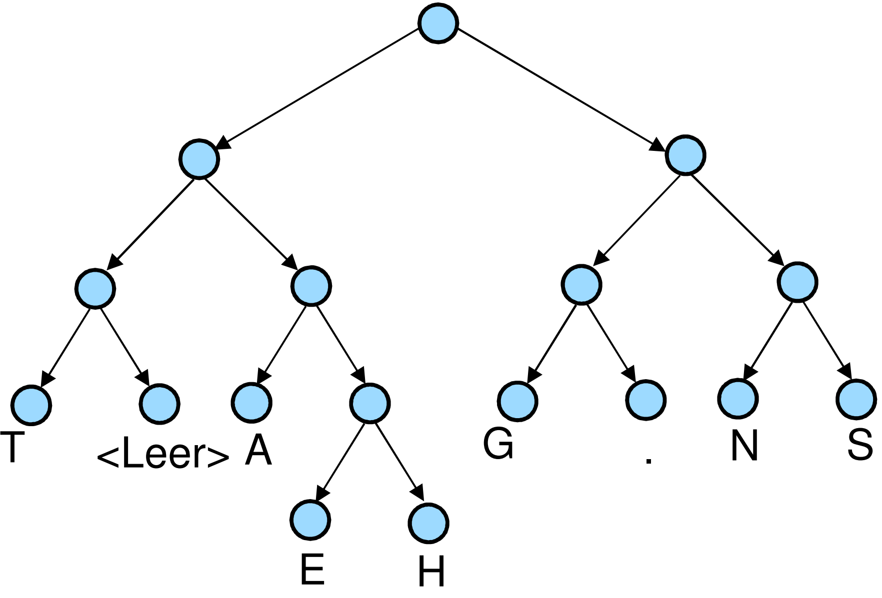
- Schritt 8:
  - AEH und T<Leer> haben die niedrigsten Werte und werden daher zusammengefasst





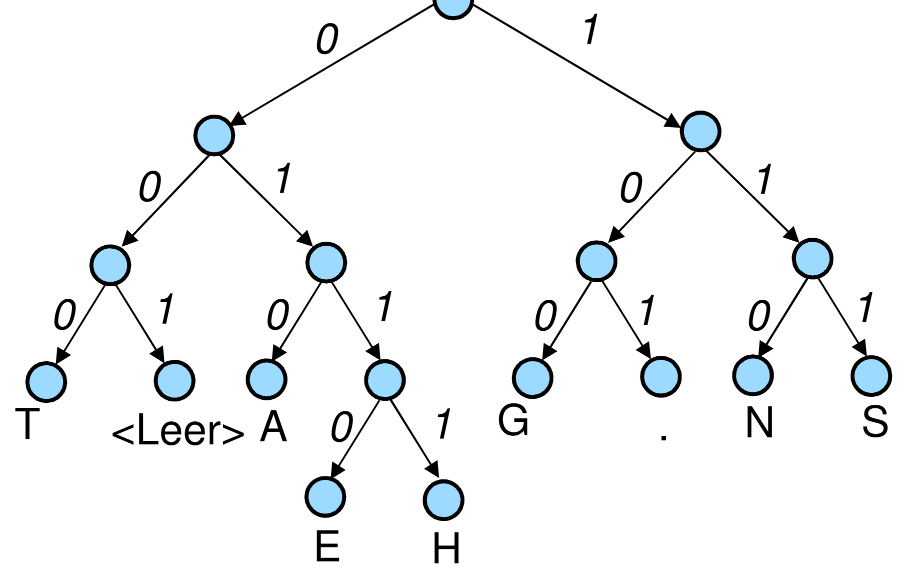
Huffman-Algorithmus: Aufgabe (VI)

- Schritt 9:
  - Zusammenfassung der beiden verbleibenden Bäume



Huffman-Algorithmus: Aufgabe (VII)

- Schritt 10:
  - Beschriftung der Kanten



- Schritt 11:
  - Code ablesen

Zeichen	A	E	G	H	N	S	T	<Leer>	.
Code	010	0110	100	0111	110	111	000	001	101

Huffman-Algorithmus: Aufgabe (VIII)

- Codieren:
  - Codiere(AGNES HAT ANGST) = 0101001100110111001011101000000101011010011100010 1

- Decodieren:
  - Decodiere(0101101100110001111000011001110001 01) = ???
  - Lösung: ANNE STEHT.

Algorithmen und Datenstrukturen

Teil 5:
Algorithmen auf Graphen (I) – Transitiver Abschluss, Aufspannende Bäume und Suchverfahren

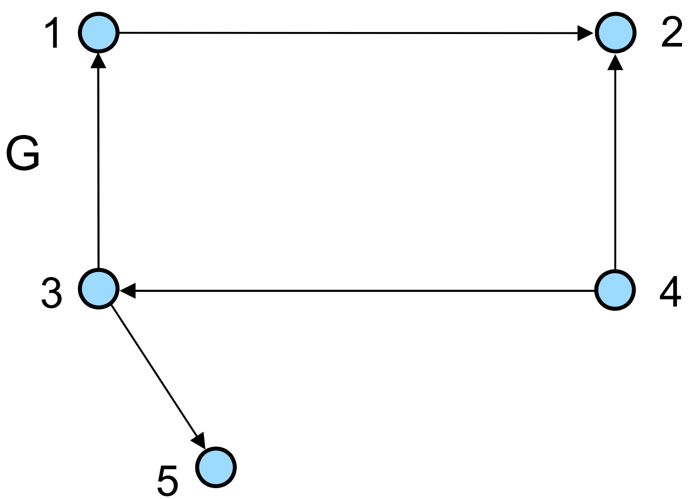
DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 04/2021

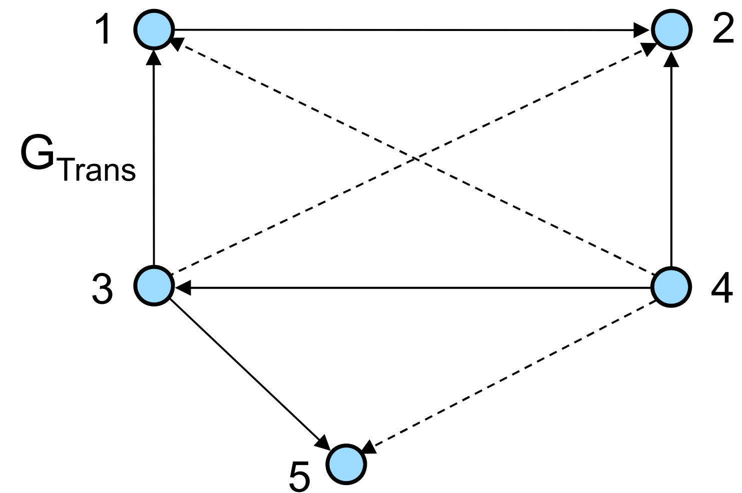
Gliederung

- Transitiver Abschluss
- Minimal aufspannende Bäume
- Suchverfahren in Graphen
- Breitensuche
- Tiefensuche

Transitiver Abschluss eines Graphen

- Transitiver Abschluss eines gerichteten Graphen: Existieren Kanten (v_1,v_2) und (v_2,v_3) ⇒ Füge Kante (v_1,v_3) hinzu
- Beispiel: G_{Trans} ist transitiver Abschluss von G





- Anwendungsbeispiel:
  - Vererbungshierarchie in Java:
    - Ist Klasse Unterklasse einer anderen?
    - Transitiver Abschluss beantwortet Frage durch die Existenz eines Vorgängers unmittelbar

Berechnung transitiver Abschluss

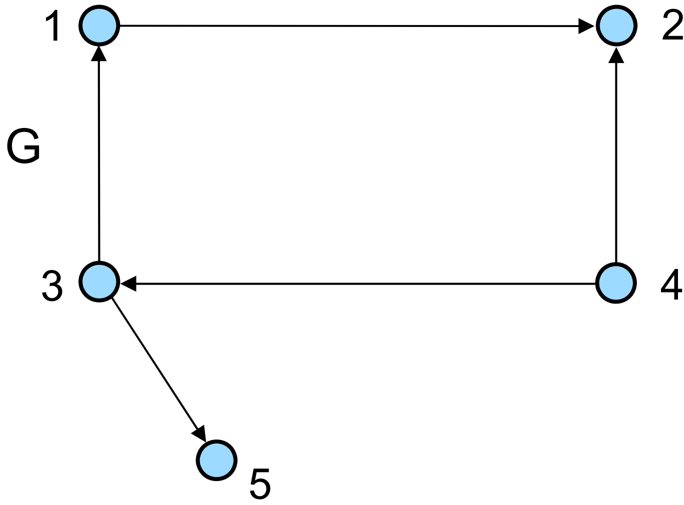
- Systematische Berechnung:
  - Verfahren 1: Ermittlung der Erreichbarkeitsmatrix durch Adjazenzmatrizenmultiplikation (Adjazenzmatrizenmultiplikationsverfahren)
  - Verfahren 2: Sukzessives Ermitteln (Warshall-Algorithmus)

- Satz als Basis für Verfahren 1:
  - Sei G gerichteter Graph mit Adjazenzmatrix A und A^s die s-te Potenz von A
  - Dann gilt: Wert an Stelle (i,j) von A^s gibt Anzahl verschiedener Kantenzüge der Länge s vom Startknoten i zum Zielknoten j an.
  - Beweis: Siehe z.B. [Tura04].

Adjazenzmatrizenmultiplikationsverfahren (I)

- Daraus folgt Algorithmus:
  - G habe n Knoten
  - Gibt es Weg zwischen v_i und v_j $\Rightarrow$ Zwischen v_i und v_j gibt es auch Weg der maximal Länge n-1
  - Mit anderen Worten: Ist der Knoten i vom Knoten j aus erreichbar, dann gibt es eine Zahl s mit $1 \leq s \leq n-1$, so dass an der Stelle (i,j) in A^s ein Wert ungleich 0 steht
  - Damit kann aus der Matrix $E = \sum_{s=1}^{n-1} A^S$ die Adjazenzmatrix des transitiven Abschluss gebildet werden: $e_{ij} = \begin{cases} 1 & falls\ s_{ij} \neq 0 \\ 0 & sonst \end{cases}$
  - Anm.: E steht für Erreichbarkeitsmatrix

Beispiel (I)



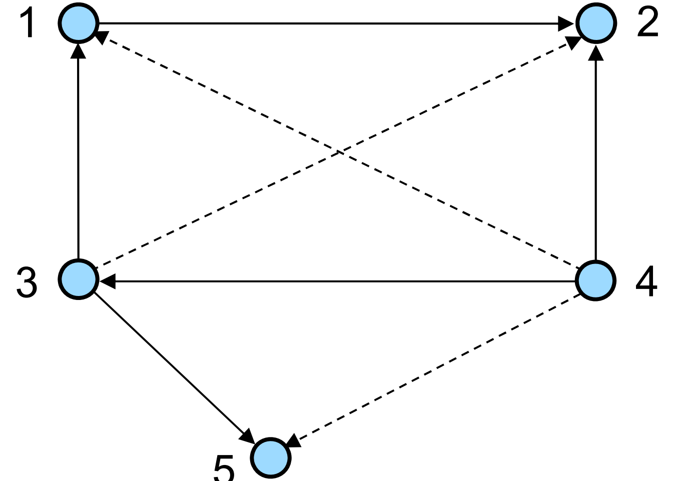
$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad A^3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Beispiel (II)

$$A^4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

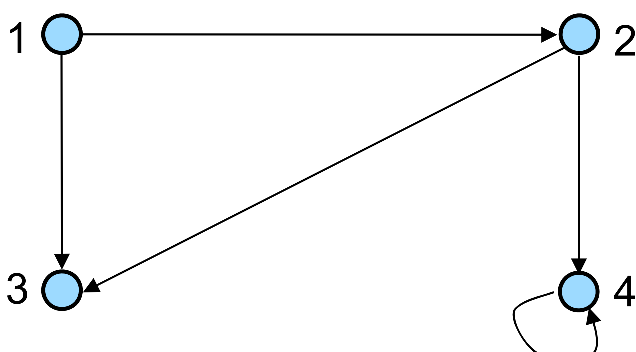
- Damit ergibt sich folgende Erreichbarkeitsmatrix und damit der transitive Abschluss:

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



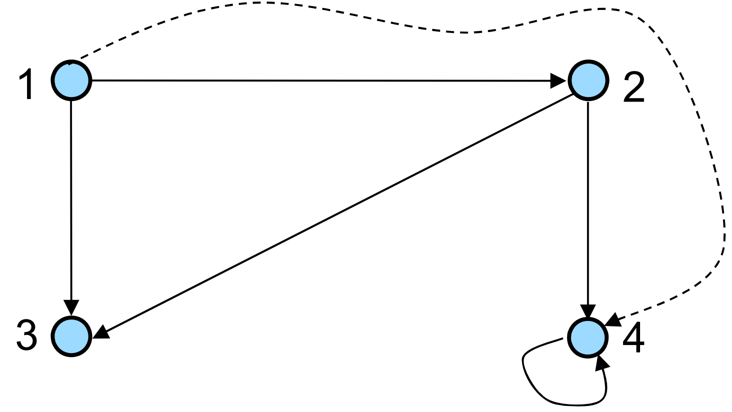
Anmerkungen

- Verfahren funktioniert auch für Graphen mit Schlingen
- Allerdings: Aussage, dass A^s die Anzahl der Kantenzüge der Länge s enthält, stimmt nicht mehr
- Beispiel:



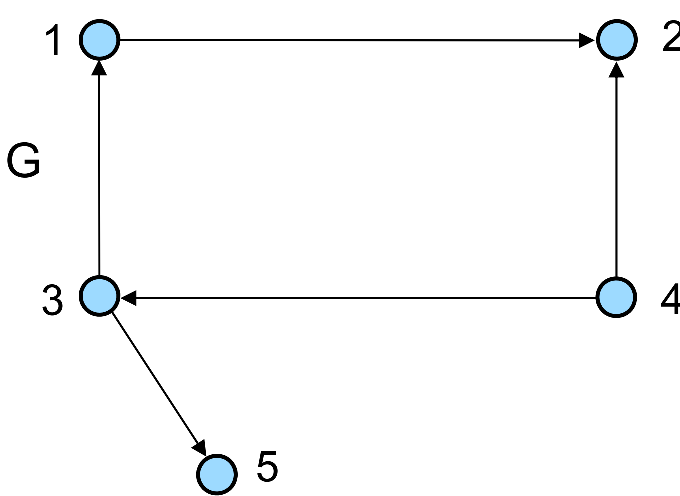
$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad E = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Warshall-Algorithmus

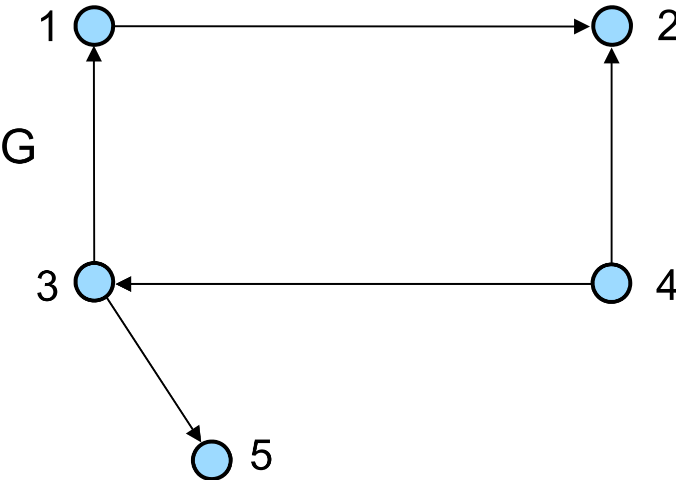
- Prinzip:
  - Sei G gerichteter Graph
  - Adjazenzmatrix A(G) wird in n Schritten in Adjazenzmatrix des transitiven Abschlusses überführt
  - Aus dem i-tem Schritt resultierender Graph wird mit G_i bezeichnet ($G=G_0$)
  - Übergang von G_k nach G_{k+1}:
    - Füge jedem Knotenpaar i, j Kante von i nach j hinzu, falls in G_k Kanten von i nach k+1 und von k+1 nach j existieren
- Beispiel:



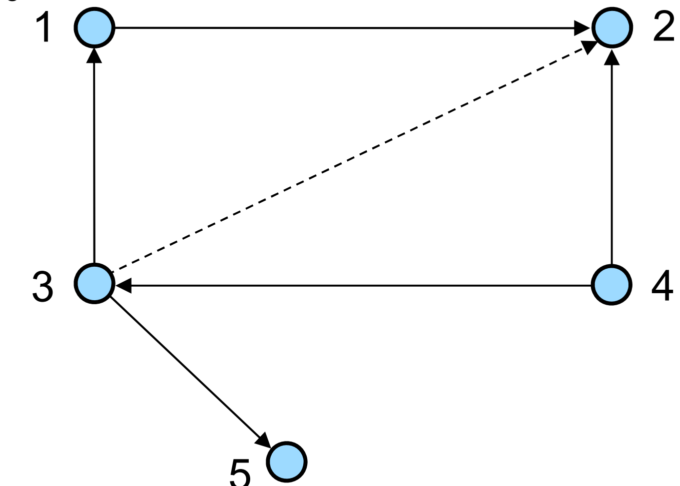
  - Knotenreihenfolge für die Abarbeitung: 1, 2, 3, 4, 5

Beispiel Warshall-Algorithmus (I)

- G_0 ist der Ausgangsgraph:

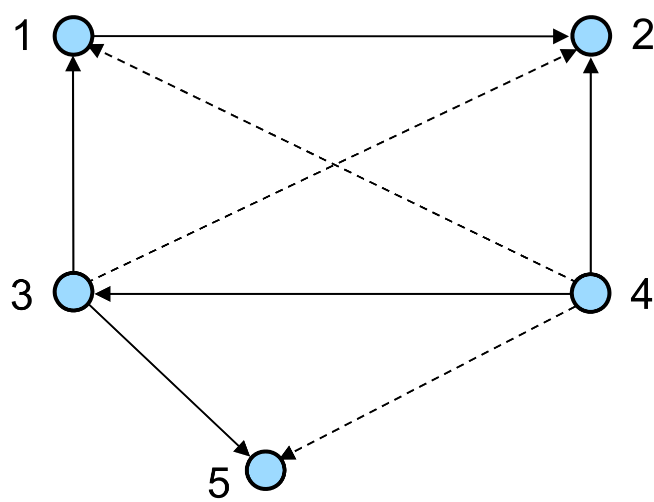


- $G_1 = G_0 + (3,2)$, da Kante (3,1) und (1,2) existiert (k=1)



Beispiel Warshall-Algorithmus (II)

- $G_2 = G_1$, da keine Kante die Bedingung erfüllt
- $G_3 = G_2$ + (4,5), da Kante (4,3) und (3,5) existiert (k=3)
  + (4,1), da Kante (4,3) und (3,1) existiert (k=3)
  [+ (4,2), da Kante (4,3) und (3,2) existiert (k=3)]



Wenn diese Kante nicht schon da wäre, würde sie jetzt eingefügt werden

- $G_4 = G_3$, da keine Kante die Bedingung erfüllt
- $G_5 = G_4$, da keine Kante die Bedingung erfüllt

Anmerkungen Warshall-Algorithmus

- In einem Schritt können u.U. auch mehrere Kanten eingefügt werden:
  - Hat der aktuell betrachtete Knoten n eingehende und m ausgehende Kanten, können bis zu n*m neue Kanten eingefügt werden

- Abarbeitungsreihenfolge der Knoten ist egal:
  - Stets wird das gleiche Resultat erreicht
  - Unterschiedlich ist aber die Reihenfolge, in der die neuen Kanten hinzugefügt werden

Komplexität des transitiven Abschluss

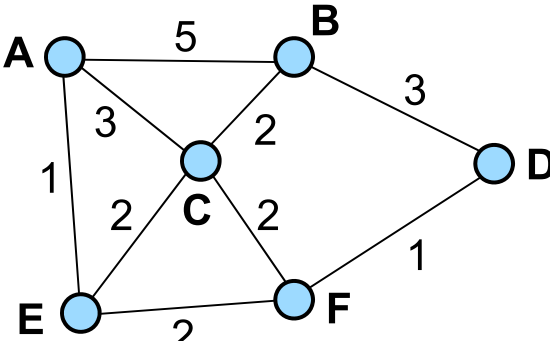
- Adjazenzmatrizenmultiplikationsverfahren:
  - Bestimme Potenzen A^s der Adjazenzmatrix A für s = 1, ..., n-1 und addiere diese:
    - Einfache Verfahren zur Multiplikation zweier nxn-Matrizen: $O(n^3)$
    - Addieren: $O(n^2)$
    - Aktionen hintereinander ⇒ Gesamtaufwand $O(n^3)$
- Warshall-Algorithmus:
  - Drei ineinander geschachtelte for-Schleifen ⇒ $O(n^3)$
- Fazit: Im worst case unterscheiden sich beiden Algorithmen nicht!
- Speicherplatzbedarf:
  - Bei Adjazentmatrizenmultiplikationsverfahren höher (zu jedem Zeitpunkt drei Matrizen)
  - Warshall-Algorithmus eine Matrix

Übersicht

- Transitiver Abschluss
- Minimal aufspannende Bäume
- Suchverfahren in Graphen
- Breitensuche
- Tiefensuche

Motivation/Beispiel

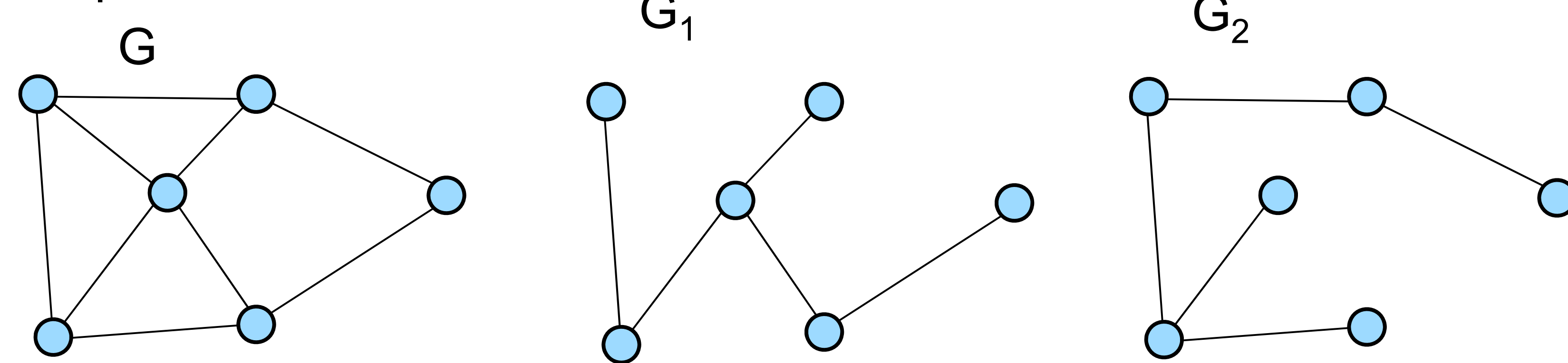
- Planung eines Kommunikationsnetzes:
  - Zwischen n Orten ist Kommunikationsnetz zu planen
  - Je zwei verschiedene Orte sollen miteinander verbunden werden (entweder direkt oder indirekt über andere Orte)
  - Verzweigungspunkte sind nur in Orten erlaubt
  - Kosten für einzelne Direktverbindungen sind bekannt
  - Gesucht: Kommunikationsnetz mit minimalen Kosten
  - Beispiel:



- Wie sieht das zugehörige Kommunikationsnetz (Graph) mit minimalen Kosten aus?

Definition (I): Aufspannender Baum

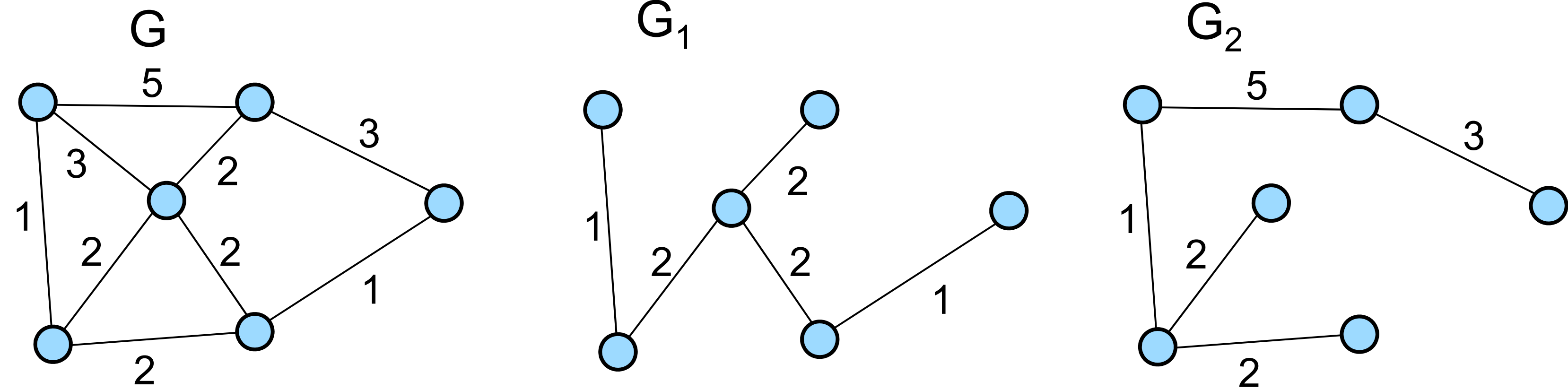
- Wenn G ein Graph ist, dann heißt G‘ aufspannender Baum (oder Spannbaum) von G, wenn
  - G‘ hat gleiche Knotenmenge wie G
  - G‘ enthält Teilmenge der Kantenmenge von G
  - G‘ ist Baum
- Beispiel:



  - G_1 und G_2 sind aufspannende Bäume von G

Definition (II): Minimal aufspannender Baum

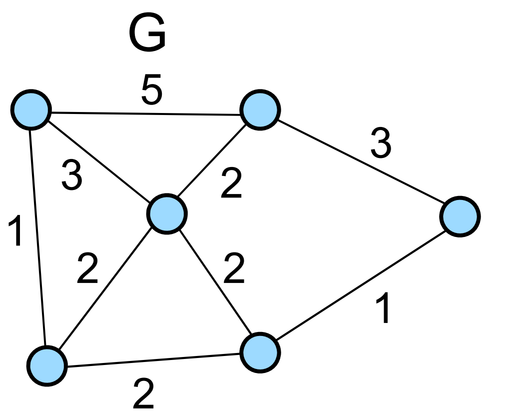
- Wenn G kantenbewerteter Graph, dann heißt G‘ minimal aufspannender Baum (oder minimaler Spannbaum) von G, wenn
  - G‘ aufspannender Baum von G
  - Kantensumme in G‘ kleiner gleich Kantensumme jedes beliebigen aufspannenden Baumes von G
- Beispiel:

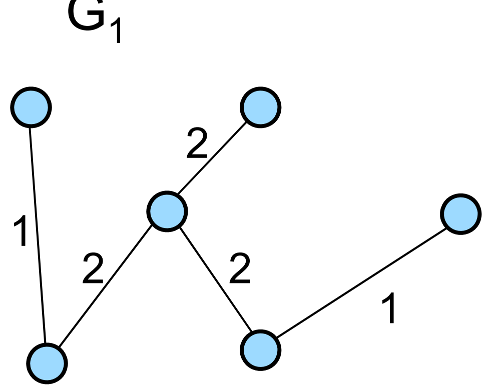


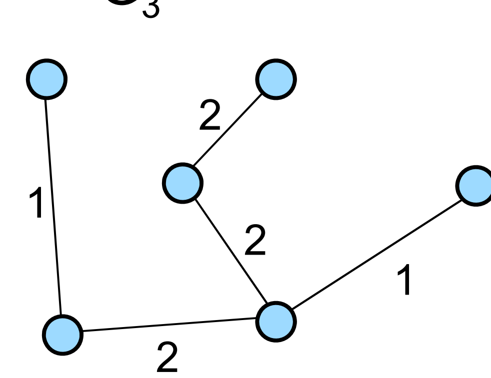
  - G_1 und G_2 sind aufspannende Bäume von G
  - G_1 ist minimal aufspannender Baum von G

Nicht-Eindeutigkeit

- Minimal aufspannende Bäume müssen nicht eindeutig sein
- Beispiel:







  - G_1 und G_3 sind minimal aufspannende Bäume von G

Algorithmus von Prim

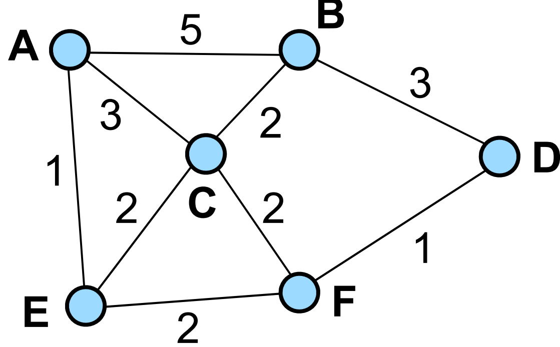
- Prinzip:
  - Kantenbewerteter Graph G = (V,E,g), Startknoten v_s
  - Notation: $\{v_1,v_2\}_x$ g.d.w. $\{v_1,v_2\} \in V$ und $g(\{v_1,v_2\})=x$
  - Minimal aufspannende Baum G'= (V',E',g') wird schrittweise aufgebaut
  - Menge offener Kanten E_{open}: Teilmenge der Kanten in G, die von einem Knoten aus $e \in V'$ zu einem Knoten in $f \in V \setminus V'$ führen
- Algorithmus:

```
( 1) insert vs into V';
( 2) while (|V'| < |V|){
( 3)   calculate Eopen;
( 4)   select {e,f} ∈ Eopen with minimal weight;
( 5)   insert f into V';
( 6)   insert {e,f} into E';
( 7) }
```

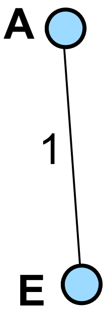
- Anmerkung:
  - Unterschiedliche minimal aufspannende Bäume erhält man in Abhängigkeit von der Wahl der Startknotens und von der Wahl der offenen Kante bei mehreren Kandidaten mit minimalem Gewicht

Beispiel (I)

- Ausgangsgraph (Startknoten sei A):



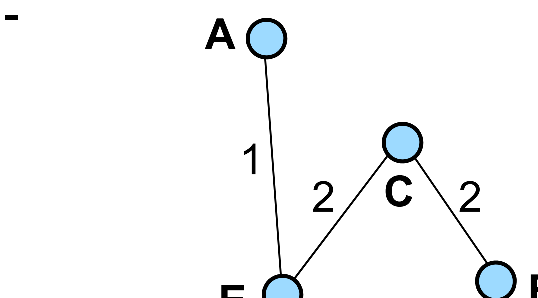
- Durchgang 1:
  - A
  - Menge der offenen Kanten: $\{\{A,B\}_5, \{A,C\}_3, \{A,E\}_1\}$
  - Ausgewählt wird Kante {A,E}
  -



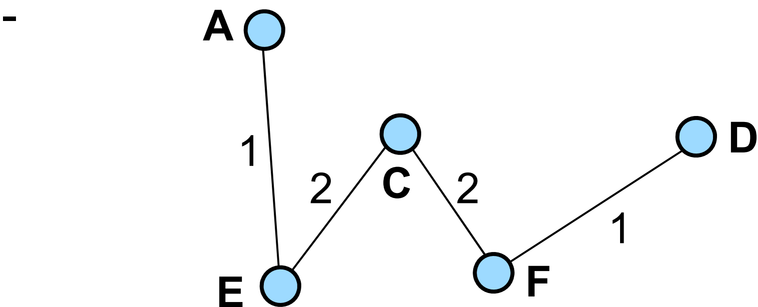
Beispiel (II)

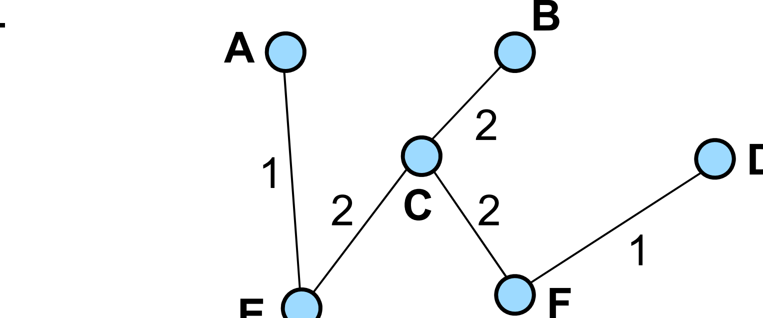
- Durchgang 2:
  - Menge der offenen Kanten: $\{\{A,B\}_5, \{A,C\}_3, \{E,C\}_2, \{E,F\}_2\}$
  - Ausgewählt wird Kante {E,C}
  -

- Durchgang 3:
  - Menge der offenen Kanten: $\{\{A,B\}_5, \{E,F\}_2, \{C,F\}_2, \{C,B\}_2\}$
  - Ausgewählt wird Kante {C,F}
  -



Beispiel (III)

- Durchgang 4:
  - Menge der offenen Kanten: $\{\{A,B\}_5, \{C,B\}_2, \{F,D\}_1\}$
  - Ausgewählt wird Kante {F,D}
  - 

- Durchgang 5:
  - Menge der offenen Kanten: $\{\{A,B\}_5, \{C,B\}_2, \{D,B\}_3\}$
  - Ausgewählt wird Kante {C,B}
  - 

  - Algorithmus terminiert, da V = V‘

Komplexität

- n-1 Iterationen (Schleife Zeile 2f.)
- Pro Durchgang wird ein Knoten in Baum eingefügt $\Rightarrow$ O(n)
- In jedem dieser Durchgänge:
  - Bestimmung offener Kanten (Zeile 3)
  - Auswahl minimaler Kante (Zeile 4)
  - Komplexität O(n)
- Schachtelung beider Teile ergibt $O(n^2)$

- Anmerkung:
  - Verbesserung Komplexität durch andere Datenstruktur (Vorrang-Warteschlange)

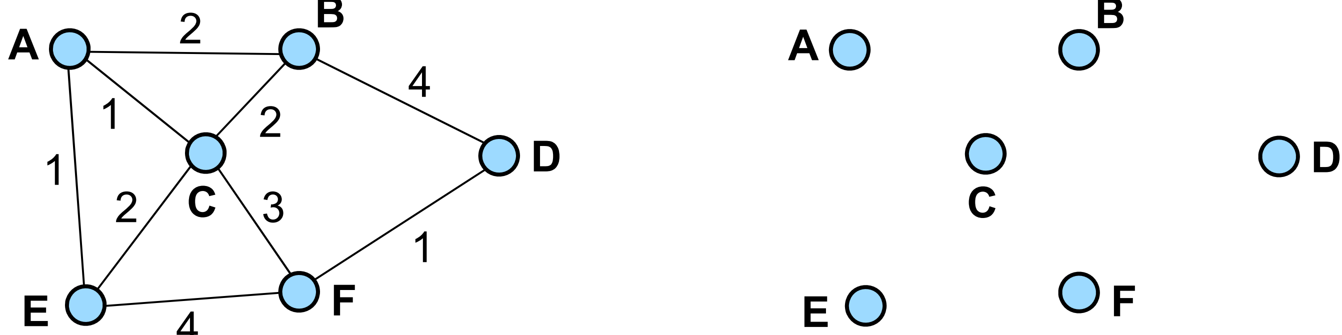
Algorithmus von Kruskal

- Prinzip:
  - Kantenbewerteter Graph G = (V,E,g)
  - Notation: $\{v_1,v_2\}_x$ g.d.w. $\{v_1,v_2\} \in V$ und $g(\{v_1,v_2\})=x$
  - Minimaler Spannbaum G'= (V',E',g'):
    - Anfangs nur Knoten (V':= V)
    - E_{sorted}: Kantenliste sortiert nach Markierung
    - Füge sukzessive Kante mit minimalem Gewicht hinzu
- Algorithmus:

```
( 1) E_sorted := Sort(E,g);
( 2) V` := V;
( 3) while (G` not connected){
( 4)   e_actual := Edge from E_sorted with minimal weight;
( 5)   E_sorted := E_sorted \ e_actual;
( 6)   if ((V`,E` ∪ (e_1,e_2),g`) has cycle)
( 7)     doNothing;
( 8)   else
( 9)     G` := (V`,E` ∪ (e_1,e_2),g`);
(10) }
```

Beispiel (I)

- Ausgangsgraph und initialer Spannbaum:

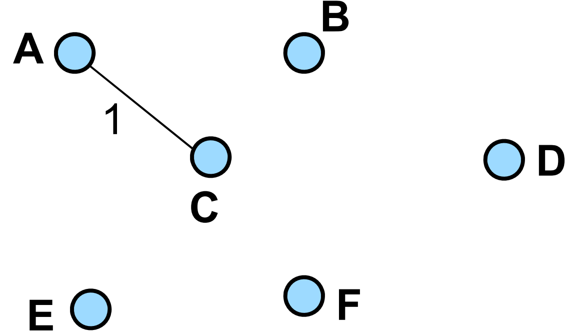


- Kantenliste E_{Sorted}:

Kante	Gewicht
(A,C)	1
(A,E)	1
(D,F)	1
(A,B)	2
(B,C)	2
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4

Beispiel (II)

- Wähle Kante mit minimalem Gewicht: $(A,C)_1$
- Kante erzeugt keinen Zyklus, wird hinzugefügt

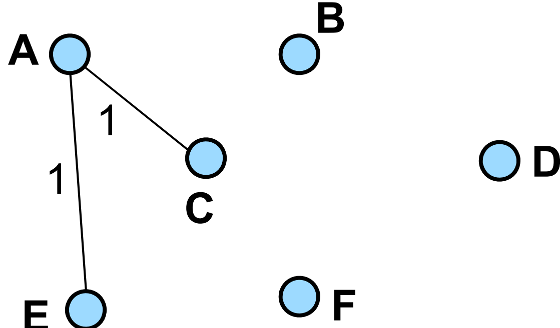


- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(A,C)~~	~~1~~
(A,E)	1
(D,F)	1
(A,B)	2
(B,C)	2
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4

Beispiel (III)

- Wähle Kante mit minimalem Gewicht: $(A,E)_1$
- Kante erzeugt keinen Zyklus, wird hinzugefügt

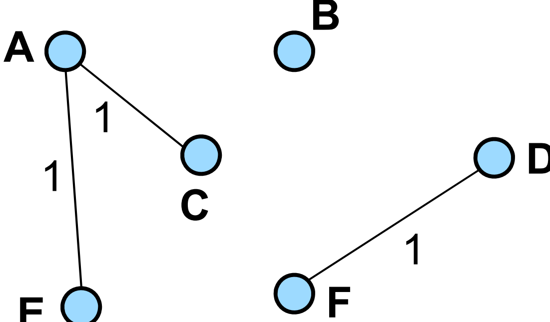


- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(A,E)~~	~~1~~
(D,F)	1
(A,B)	2
(B,C)	2
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4

Beispiel (IV)

- Wähle Kante mit minimalem Gewicht: $(D,F)_1$
- Kante erzeugt keinen Zyklus, wird hinzugefügt

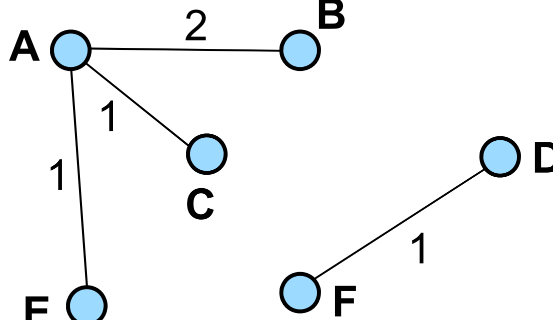


- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(D,F)~~	~~1~~
(A,B)	2
(B,C)	2
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4

Beispiel (V)

- Wähle Kante mit minimalem Gewicht: $(A,B)_2$
- Kante erzeugt keinen Zyklus, wird hinzugefügt



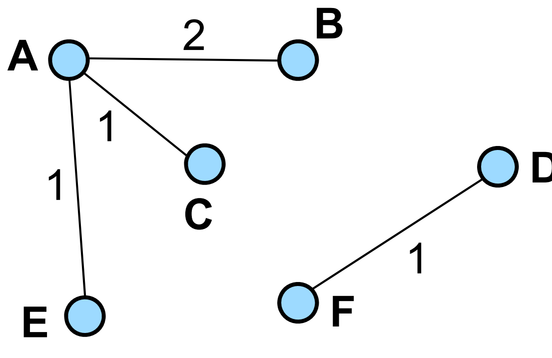
- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(A,B)~~	~~2~~
(B,C)	2
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4

Beispiel (VI)

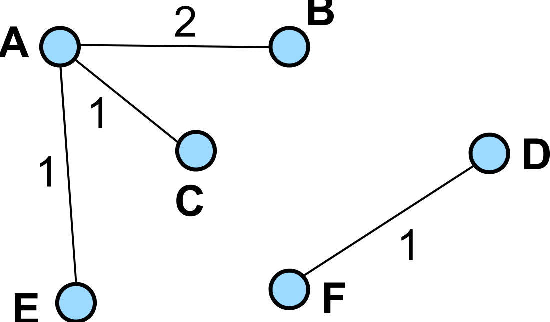
- Wähle Kante mit minimalem Gewicht: $(B,C)_2$
- Kante erzeugt Zyklus, wird verworfen
- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(B,C)~~	~~2~~
(C,E)	2
(C,F)	3
(B,D)	4
(E,F)	4



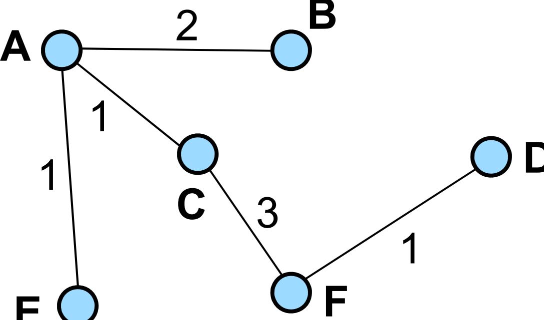
- Wähle Kante mit minimalem Gewicht: $(C,E)_2$
- Kante erzeugt Zyklus, wird verworfen
- Aktualisiere Kantenliste E_{sorted}:

Kante	Gewicht
~~(C,E)~~	~~2~~
(C,F)	3
(B,D)	4
(E,F)	4



Beispiel (VII)

- Wähle Kante mit minimalem Gewicht: $(C,F)_3$
- Kante erzeugt keinen Zyklus, wird hinzugefügt



- Alle Knoten verbunden ⇒ Algorithmus terminiert

Komplexität / Anmerkung

- Wichtige Operationen:
  - Find: Herausfinden, ob Zyklus entsteht
  - Union: Vereinigen zweier Bäume
- Pro Kante:
  - Maximal zwei find-Operationen
  - Maximal eine union-Operationen
- Insgesamt: $O(n + e \cdot \log_2 n)$ mit e Anzahl Kanten
- Hinweis: Setzt „gute" Datenstruktur und „geschicktes" Zusammenfügen der Bäume voraus

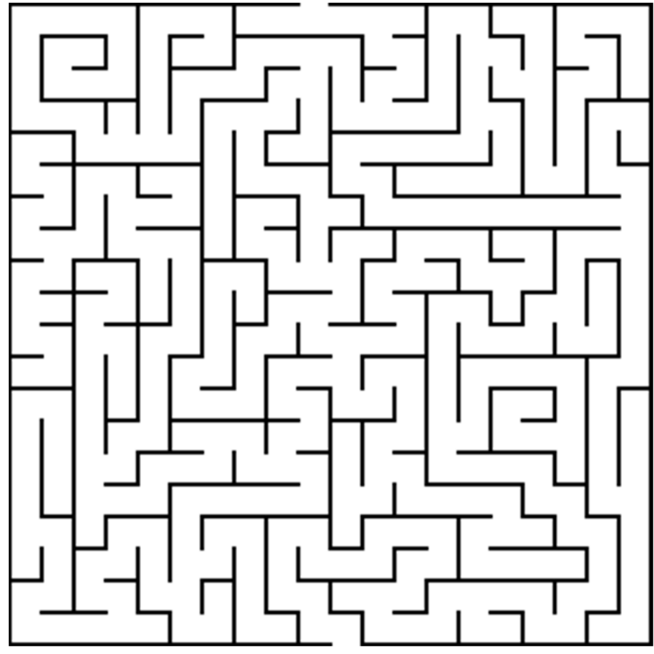
- Anmerkung:
  - Unterschiedliche minimal aufspannende Bäume entstehen in Abhängigkeit der Kantenwahl in Zeile (4) bei mehreren Kanten mit minimalem Gewicht

Anwendungen (I)

- Kommunikationsnetze (siehe Motivation):
  - Telefonnetze
  - Elektrische Netze
- Computernetzwerke:
  - STP (Spanning Tree Protocol):
    - Standard in geswitchten Umgebungen (IEEE-Norm 802.1D, 1998)
    - Zentraler Teil von Switch-Infrastrukturen
    - Rechnernetz mit Vielzahl von Switches
    - STP stellt sicher, dass zwischen zwei Rechnern eindeutiger Datenpfad
  - Modifizierte Version RSTP (Rapid STP), (IEEE-Norm 802.1D, 2004):
    - Bei signalisierten Topologieänderungen:
      - Keine Löschung, sondern Weiterarbeiten wie bisher
      - Alternativpfade berechnen
    - Damit: Deutliche Reduzierung Ausfallzeit

Anwendungen (II)

- Spieleprogrammierung:
  - Labyrintherzeugung
  - Knoten entspricht Feld, Kante Übergang zum Nachbarfeld
  - Also: Fehlende Kante entspricht Wand
  - Mittels Spannbaum erzeugtes Labyrinth besitzt nur einen Lösungsweg



[http://www.mazegenerator.net/]

Übersicht

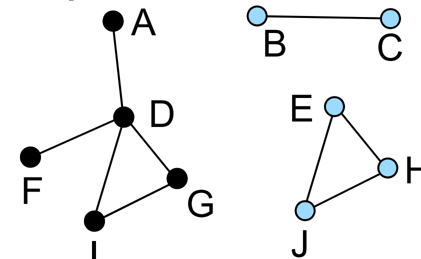
- Transitiver Abschluss
- Minimal aufspannende Bäume
- Suchverfahren in Graphen
- Breitensuche
- Tiefensuche

Suchverfahren in Graphen (I)

- Klasse von Verfahren zum systematischen Durchlaufen aller Knoten/Kanten eines Graphen
- Problem tritt z.B. auf bei Fragestellung
  - ob ein ungerichteter Graph zusammenhängend ist
  - ob ein gerichteter Graph einen Zyklus enthält
- Grundlegende Idee für diese Algorithmen:
  - Beginne an einem bestimmten Startknoten v_s
  - Besuche alle von v_s erreichbaren Knoten
  - Verlauf der Suche spiegelt sich im sog. Erreichbarkeitsbaum EB wider
  - EB ist ein Wurzelbaum mit v_s als Wurzel
  - EB besteht aus den von v_s aus erreichbaren Knoten und den Kanten, die zu diesen Knoten führen
  - Zu einem v_s können verschiedene EB existieren (Gleiche Knotenmenge, aber verschiedene Kantenmenge)

Suchverfahren in Graphen (II)

- Basis aller Suchverfahren ist folgender Markierungsalgorithmus (Name kommt daher, dass bereits besuchte Knoten markiert werden):
  (1) Markiere den Startknoten
  (2) Solange noch Kanten von markierten zu unmarkierten Knoten existieren, wähle eine solche Kante aus und markiere den Endknoten dieser Kante
- Beispiel:



Welche Knoten sind von A aus erreichbar?

- Markiere A (Schritt (1))
- Sind noch Kanten von markierten zu unmarkierten Knoten vorhanden? JA: {A,D}
- Wähle {A,D} und markiere D
- Sind noch Kanten von markierten zu unmarkierten Knoten vorhanden? JA: {D,F}, {D,I}, {D,G}
- Wähle {D,G} und markiere G
- Sind noch Kanten von markierten zu unmarkierten Knoten vorhanden? JA: {D,F}, {D,I}, {G,I}
- Wähle {D,I} und markiere I
- Sind noch Kanten von markierten zu unmarkierten Knoten vorhanden? JA: {D,F}
- Wähle {D,F} und markiere F
- Sind noch Kanten von mark. zu unmark. Knoten vorhanden? NEIN $\Rightarrow$ Ende

Suchverfahren in Graphen (III)

- Eigenschaften des Verfahrens:
  - Die verwendeten Kanten bilden am Ende den EB
  - Da in jedem Schritt ein Knoten markiert wird, folgt aus der Endlichkeit des Graphen die Terminierung des Algorithmus
  - Menge der am Ende markierten Knoten ist die Menge der erreichbaren Knoten
  - Bei entsprechender Realisierung der Auswahl in Schritt (2) hat der Algorithmus eine Laufzeit von O(n+m)
  - Algorithmus kann durch Modifizierung die Frage der Erreichbarkeit eines Knoten von s aus in n Schritten beantworten
  - Alle Suchverfahren basieren auf dieser Grundidee, unterscheiden sich jedoch in der Art und Weise, wie in Schritt (2) Kanten ausgewählt werden

Suchverfahren in Graphen (IV)

- Tiefensuche (depth-first-search):
  - Versucht, den am weitesten vom Startknoten entfernten Knoten so früh wie möglich zu besuchen
  - Dazu wählt man in Schritt (2) eine Kante aus, deren Startknoten der zuletzt markierte Knoten ist
  - Sind alle von diesem Knoten ausgehenden Kanten schon abgearbeitet, dann arbeite die markierten Knoten in umgekehrter Reihenfolge ab
  - Herkunft Verfahrensnamen: Man geht bei jedem Schritt „so tief wie möglich in den Graphen hinein“
- Breitensuche (breadth-first-search):
  - Suche wird so breit wie möglich angelegt, d.h. für jeden besuchten Knoten werden zunächst alle Nachbarn besucht
  - Dazu wählt man in Schritt (2) jeweils eine Kante, deren Startknoten der am längsten markierte Knoten ist

Übersicht

- Transitiver Abschluss
- Minimal aufspannende Bäume
- Suchverfahren in Graphen
- Breitensuche
- Tiefensuche

Breitensuche (I)

- Idee: Bearbeite Knoten, der in n Schritten von u erreichbar ist, erst dann, wenn alle in n-1 Schritten erreichbaren Knoten bereits abgearbeitet worden sind
- Eingabe: Ungerichteter Graph G = (V,E), Startknoten $v_s \in V$
- Als Hilfsstruktur wird eine Schlange von Knoten benötigt:
  - Schlange ist Struktur von prinzipiell unendlicher Kapazität
  - Arbeitet nach dem FIFO-Prinzip (First In First Out), d.h. es kann jeweils nur das älteste Element entnommen werden
  - Funktionen auf dem Datentyp Schlange:
    - `put(v)`: Lege den Knoten v in der Schlange ab
    - `get()`: Entnehme Knoten aus der Schlange
    - `read()`: Lesen aus der Schlange ohne zu Entnehmen
    - `isEmpty()`: Prüft, ob die Schlange leer ist oder nicht
- Als weitere Hilfsstruktur werden zu jedem Knoten aktueller Farbwert, Abstand d zu Startknoten v_s und der Vorgänger pred verwaltet, von dem aus der Knoten erreicht worden ist

Breitensuche (II)

- Farbwerte:
  - weiß = unbehandelt
  - grau = in Schlange
  - schwarz = abgearbeitet
- Funktion `succ(v)` liefert die Menge der direkten Nachbarn (bzw. direkten Nachfolger im gerichteten Fall) des Knotens v
- Resultat:
  - Aus Distanzwerten kann am Ende das Ergebnis der Breitensuche abgelesen werden
  - Ist G zusammenhängend, liefern pred-Werte einen Spannbaum
  - Ist G nicht zusammenhängend, liefern pred-Werte einen Spannwald

Algorithmus (I)

- Sei v_s Startknoten, Q Schlange

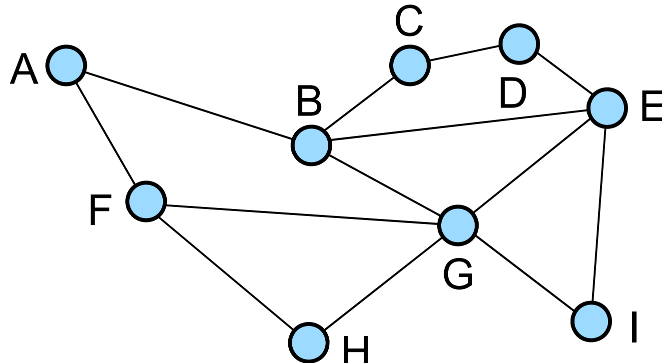
```
( 1) // Initialisierungen
( 2) for each v ∈ V/{v_s} {
( 3)   farbe[v] = weiß;
( 4)   d[v] = ∝;
( 5)   pred[v] = null;
( 6) }
( 7) farbe[v_s] = grau;
( 8) d[v_s] = 0;
( 9) Q.put(v_s);
...
```

Algorithmus (II)

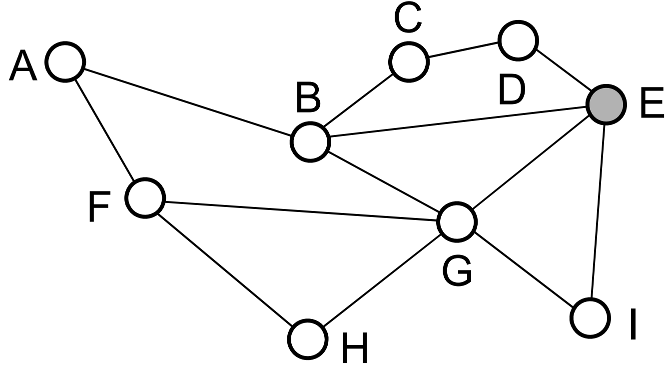
```
...
(10) // Schleife zum Abarbeiten der Schlange
(11) while (!Q.isEmpty()){
(12)   v = Q.read();
(13)   for each u ∊ succ(v){
(14)      if (farbe[u] == weiß){
(15)        // Unbearbeitete Knoten in Schlange einfügen
(16)        farbe[u] = grau;
(17)        d[u] = d[v] + 1;
(18)        pred[u] = v;
(19)        Q.put(u);
(20)      }
(21)   }
(22)   Q.get();
(23)   farbe[v] = schwarz;
(24) }
```

Beispiel (I)

- Beispiel: Breitensuche in folgendem Graphen mit Startknoten E



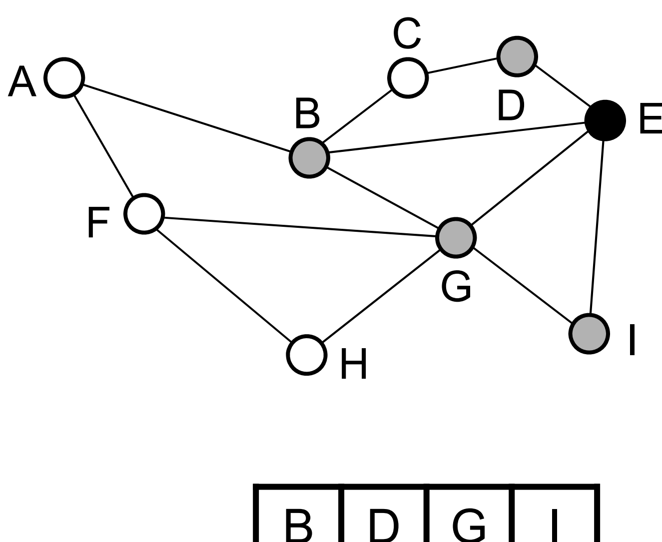
- Initialisierung:



Knoten	d	pred
A	∞	null
B	∞	null
C	∞	null
D	∞	null
E	0	null
F	∞	null
G	∞	null
H	∞	null
I	∞	null

Beispiel (II)

- Durchgang 1:
  - v = E
  - Alle unbearbeiteten (d.h. weißen) Nachbarn von E: Grau färben, Distanz eintragen, E als Vorgänger eintragen und in Schlange eintragen
  - E aus Schlange entnehmen und schwarz färben



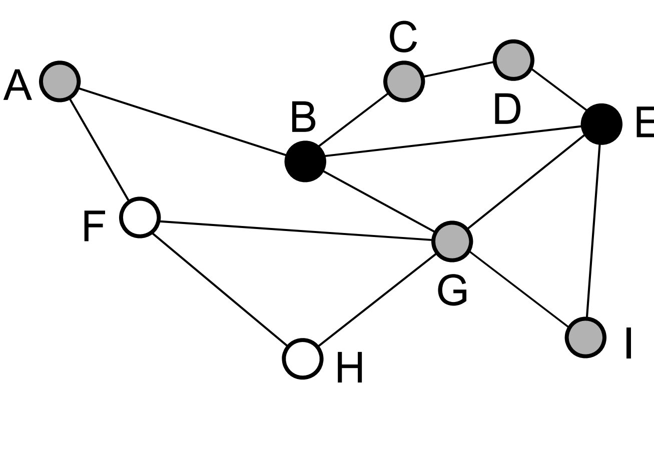
B	D	G	I

Knoten	d	pred
A	∞	null
B	1	E
C	∞	null
D	1	E
E	0	null
F	∞	null
G	1	E
H	∞	null
I	1	E

  - Schlange nicht leer $\Rightarrow$ Weitermachen

Beispiel (III)

- Durchgang 2:
  - v = B
  - Alle unbearbeiteten (d.h. weißen) Nachbarn von B: Grau färben, Distanz eintragen, B als Vorgänger eintragen und in Schlange eintragen
  - B aus Schlange entnehmen und schwarz färben



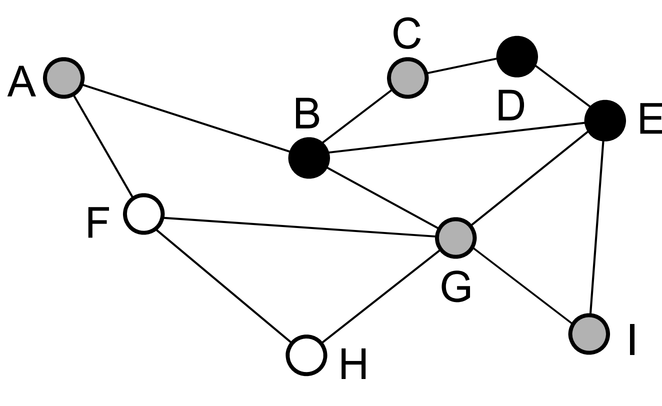
D	G	I	A	C

Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	∞	null
G	1	E
H	∞	null
I	1	E

  - Schlange nicht leer $\Rightarrow$ Weitermachen

Beispiel (IV)

- Durchgang 3:
  - v = D
  - Alle unbearbeiteten (d.h. weißen) Nachbarn von D: Grau färben, Distanz eintragen, B als Vorgänger eintragen und in Schlange eintragen
  - Entfällt, da diese Menge leer ist
  - D aus Schlange entnehmen und schwarz färben



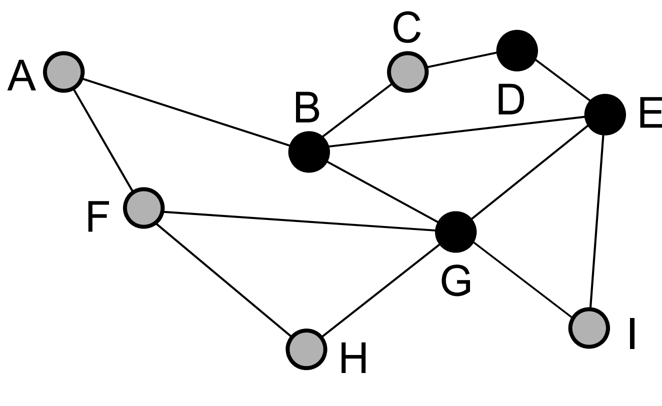
G	I	A	C

Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	∞	null
G	1	E
H	∞	null
I	1	E

- Schlange nicht leer $\Rightarrow$ Weitermachen

Beispiel (V)

- Durchgang 4:
  - v = G
  - Alle unbearbeiteten (d.h. weißen) Nachbarn von G: Grau färben, Distanz eintragen, G als Vorgänger eintragen und in Schlange eintragen
  - G aus Schlange entnehmen und schwarz färben



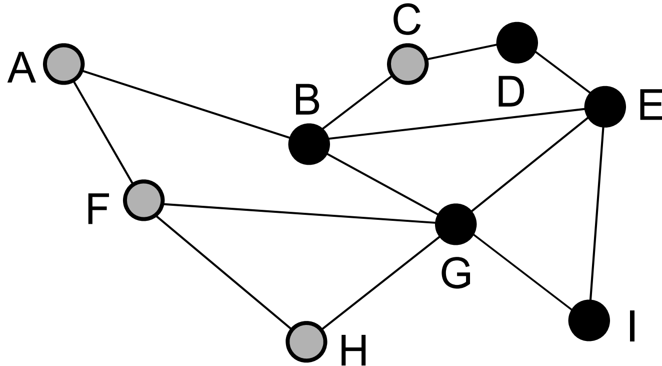
I	A	C	F	H

Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	2	G
G	1	E
H	2	G
I	1	E

  - Schlange nicht leer $\Rightarrow$ Weitermachen

Beispiel (VI)

- Durchgang 5:
  - v = I
  - Alle unbearbeiteten (d.h. weißen) Nachbarn von I: Grau färben, Distanz eintragen, G als Vorgänger eintragen und in Schlange eintragen
  - Entfällt, da die Menge leer ist
  - I aus Schlange entnehmen und schwarz färben



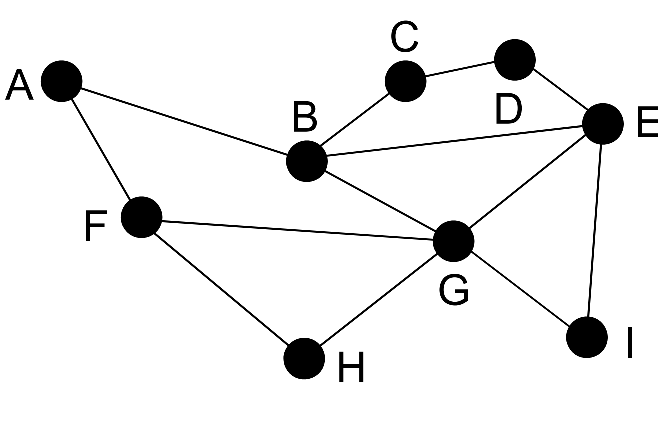
A	C	F	H

Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	2	G
G	1	E
H	2	G
I	1	E

  - Schlange nicht leer ⇒ Weitermachen

Beispiel (VII)

- Durchgang 6-9:
  - Analog zu Durchgang 5 mit v = A bzw. C bzw. F bzw. H
  - Schritte im Schleifenrumpf entfallen jeweils, da keine unmarkierten Knoten mehr
  - A bzw. C bzw. F bzw. H werden aus Schlange entnommen und schwarz gefärbt



Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	2	G
G	1	E
H	2	G
I	1	E

- Schlange leer $\Rightarrow$ Algorithmus terminiert

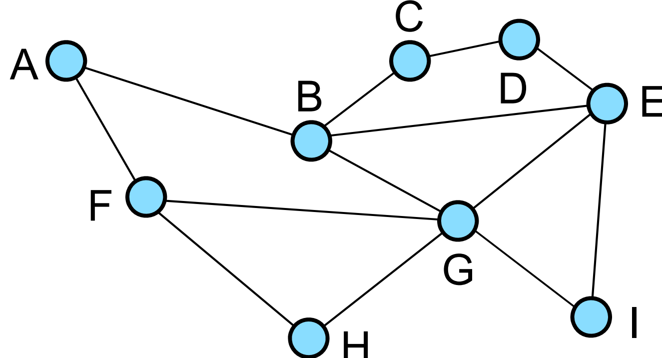
Beispiel (VIII) / Komplexität

- Resultat:
  - Aus der Spalte d kann das Resultat direkt abgelesen werden
  - Es gibt i.A. mehrere Lösungen, da mehrere Knoten mit einer bestimmten Distanz vom Startknoten aus erreichbar sind
  - Bsp.: E B D G I A C F H

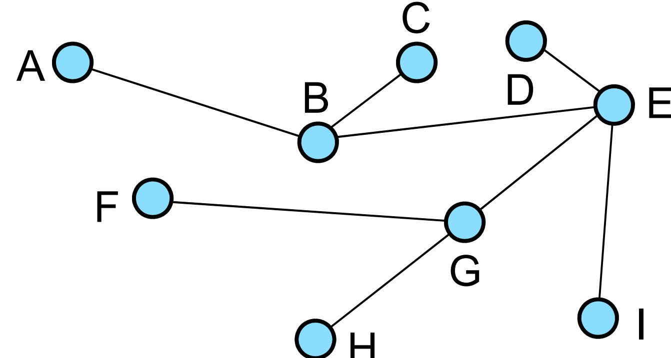
- Komplexität der Breitensuche:
  - Ein Besuch pro Knoten und Kante, d.h. O(n+m)
  - Falls G zusammenhängend, gilt $|E| > |V|-1 \Rightarrow$ Komplexität O(m)

Breitensuche/Spannbaum

- Nimmt man die ausgewählten Kanten (d.h. die am Ende in der Tabelle stehen), so ist das Resultat der Breitensuche ein Spannbaum
- Im Beispiel ergibt sich zum Graphen



der Spannbaum

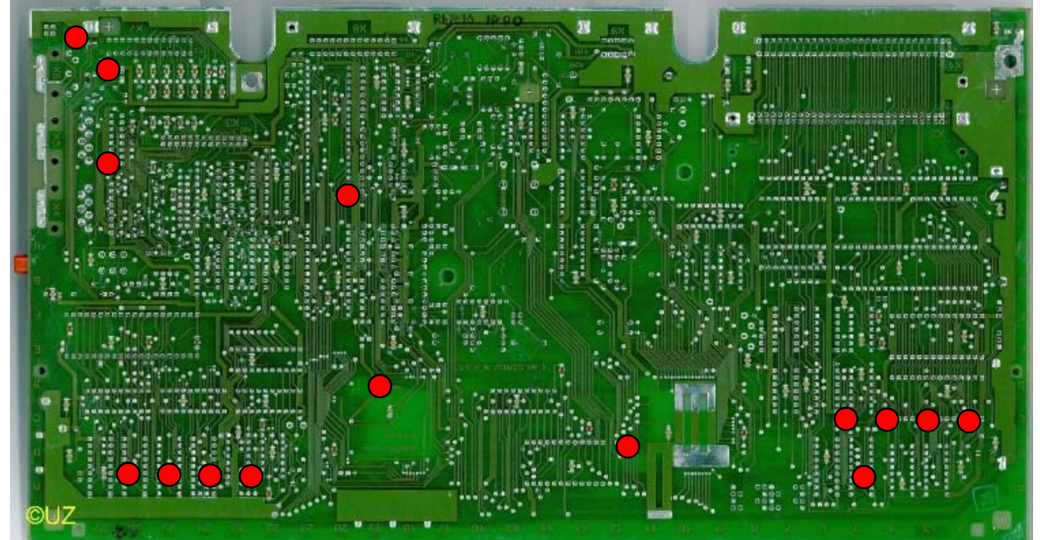


Knoten	d	pred
A	2	B
B	1	E
C	2	B
D	1	E
E	0	null
F	2	G
G	1	E
H	2	G
I	1	E

Anwendungen (I)

- Distanzprobleme (Kürzester unbewerteter Pfad)
  - z.B. Bearbeitung von Platinen oder Holzplatten
  - Löt- oder Bohrkopf soll über Platte fahren und dabei unterschiedliche Löt- bzw. Bohrstellen anfahren
  - Kosten für Bewegung zwischen zwei Stellen gleich
  - Durch Breitensuche wird dabei minimaler Weg ermittelt

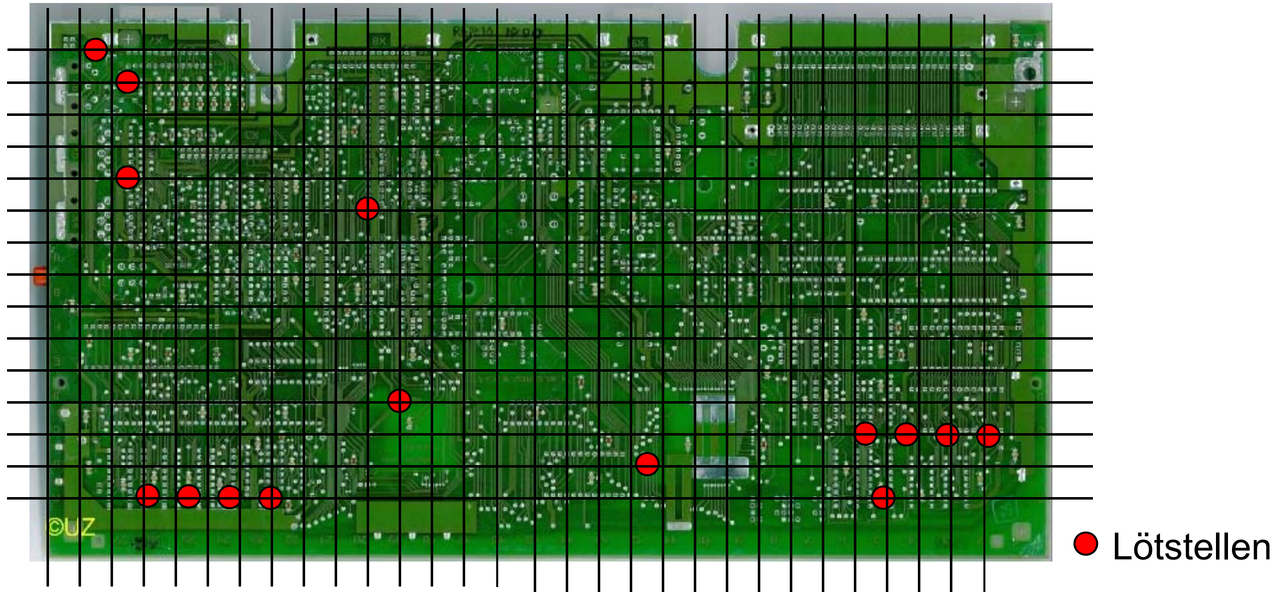
[http://www.mec-elektronik.de/]



● Lötstellen

[http://greigo.com/computer-platine#]

Anwendungen (II)



- Erläuterung:
  - Lege Gitter über Platine und markiere Lötstellen:
  - Bohrkopf verarbeitet Startknoten
  - Dann alle Knoten mit Abstand 1 zum Startknoten
  - Dann alle Knoten mit Abstand 2 zum Startknoten
  - usw.

- Auffinden aller Knoten innerhalb einer Zusammenhangskomponente

Anwendungen (III)

- Click-Stream-Analyse
  - Weblog: Aufrufreihenfolgen von Seiten durch Benutzer/innen
  - Fragestellungen Weblog-Analyse:
    - In welcher Reihenfolge werden Seiten nacheinander aufgerufen?
    - Wie viele Schritte braucht Nutzer/in durchschnittlich von Seite A zu B?

mydomain.com Top 15 Referrer Graph

Dates Analyzed:
Start: 28/Jan/2005:12:53:00
End: 02/Feb/2005:23:58:56
Top Pairs account for 522 of 1020 clicks (51.18%)

Legend:
Referrers shown as brown boxes; intensity shows relative number of referrals.
Site Pages shown in ovals, colored from red to blue, representing popularity of page.
Hit and Referral Counts are totals for period, not just those shown on graph.

Direct Link (424 referrals)
/search (36 referrals)
/gallery.htm (19 referrals) (32 hits)
/trifolds.html (10 referrals) (24 hits)
/service.html (20 referrals) (35 hits)
/ (173 referrals) (324 hits)
/post-card-gallery.htm (11 referrals) (13 hits)
/businesscards.html (37 referrals) (59 hits)
/contactus.html (8 referrals) (48 hits)
/directmail.html (10 referrals) (22 hits)
/postcards.html (78 referrals) (133 hits)
/bcardlayout.html (10 referrals) (16 hits)
/b-card-form.htm (3 referrals) (20 hits)
/pcardorderform.html (11 referrals) (28 hits)
/contactform.html (3 referrals) (20 hits)

mydomain.com Session Track #11

Session Length: 27:24 minutes. 14 Clicks.
SessionID: 65.3.168.167

Direct Link
1 - 10:22:06 PM
/
2 - 10:22:25 PM
14 - 10:49:30 PM
/contactus.html
/directmail.html
3 - 10:23:17 PM
/gallery.htm
4 - 10:23:33 PM
4 BACK
5 - 10:23:42 PM
10 - 10:48:22 PM
13 - 10:48:58 PM
/custom-card-form.htm
/post-card-gallery.htm
/trifolds.html
6 - 10:23:59 PM
7 - 10:24:16 PM
6 BACK
/pcardorderform.html
11 - 10:48:28 PM
9 - 10:48:14 PM
7 BACK
/service.html
/page.php/index
11 BACK
8 - 10:47:29 PM
/postcards.html
12 - 10:48:43 PM

[http://statviz.sourceforge.net/]

Übersicht

- Transitiver Abschluss
- Minimal aufspannende Bäume
- Suchverfahren in Graphen
- Breitensuche
- Tiefensuche

Algorithmus mit Stapel (I)

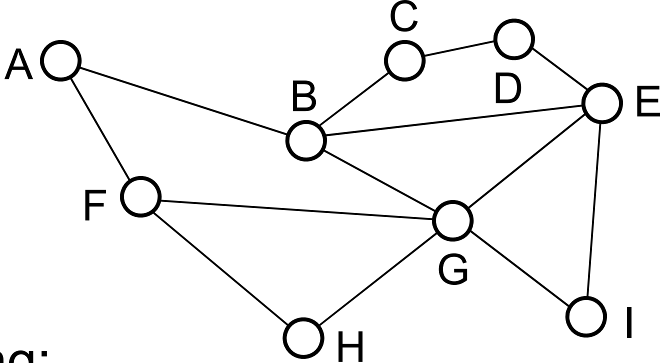
- Gegeben:
  - Ungerichteter Graph G = (V,E)
  - Stapel S zur Verwaltung von Knoten:
    - `push(v)`: Lege Knoten v auf S ab
    - `pop()`: Lese Knoten von S
    - `peek()`: Lese Knoten von S ohne diesen zu entfernen
    - `empty()`: Liefert `true` bei leerem Stapel, `false` sonst
  - Startknoten $v_S \in V$
  - Feld `marked` gibt an, ob Knoten markiert (true) oder nicht (false)
  - Funktion `unmarkedNeighbours(v)` liefert unmarkierte Nachbar von Knoten v
  - Funktion `output(v)` gibt Knoten v aus bzw. verarbeitet diesen

Algorithmus mit Stapel (II)

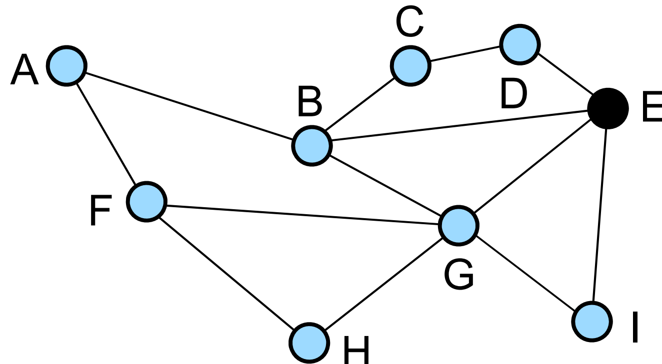
```
( 1) // Initialisierungen
( 2) for each v ∈ V {
( 3)   marked[v] = false;
( 4) }
( 5) marked[vS] = true;
( 6) S.push(vS);
( 7) // Hauptschleife
( 8) while (!S.empty()){
( 9)   actNode = S.pop();
(10)   output(actNode);
(11)   for each v ∈ unmarkedNeighbours(actNode){
(12)     marked[v] = true;
(13)     S.push(v);
(14)   }
(15) }
```

Beispiel (I)

- Tiefensuche in folgendem Graphen mit Startknoten E



- Initialisierung:
  - Markiere alle Knoten mit `false`
  - Markiere Startknoten E und lege diesen auf den Stapel

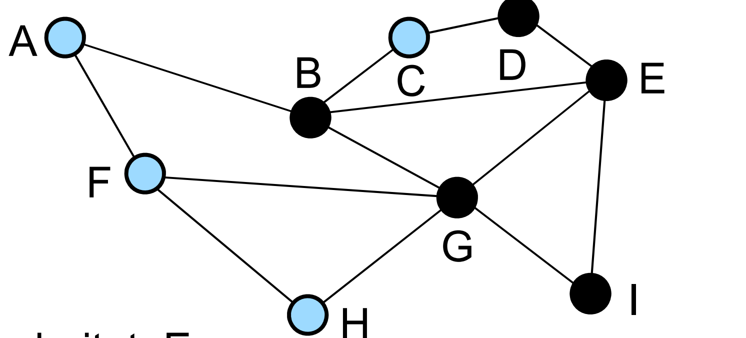


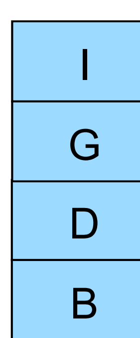
E

  - Bisher besucht: -

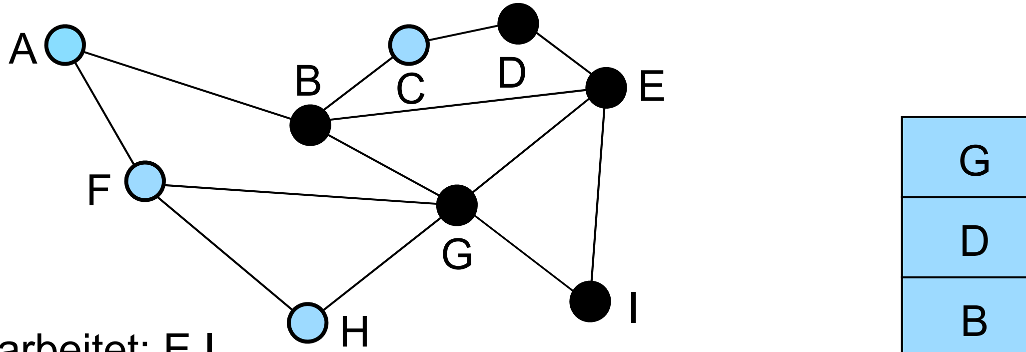
Beispiel (II)

- Durchgang 1:
  - $v_{Aktuell}$ = E, entferne E vom Stapel, bearbeite E
  - Sind noch nicht markierte Nachbarn von E vorhanden? Ja: B, D, G, I
  - Markiere diese und lege sie auf Stapel





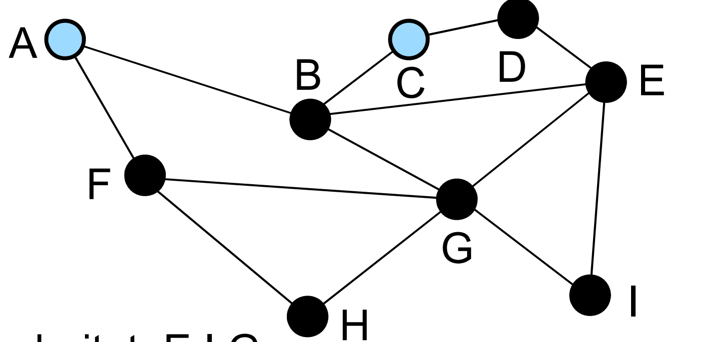
  - Bisher bearbeitet: E
- Durchgang 2:
  - $v_{Aktuell}$ = I , entferne I vom Stapel, bearbeite I
  - Sind noch nicht markierte Nachbarn von I vorhanden? Nein



  - Bisher bearbeitet: E I

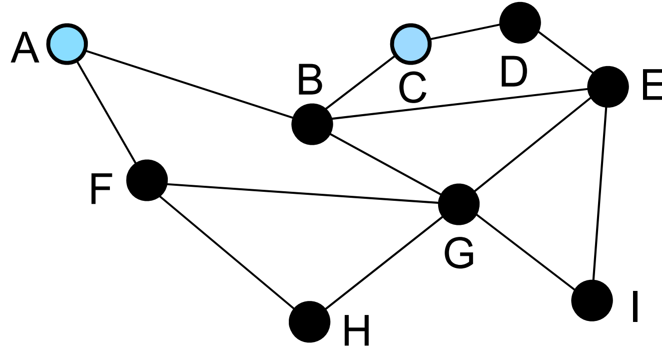
Beispiel (III)

- Durchgang 3:
  - $v_{Aktuell}$ = G, entferne G vom Stapel, bearbeite G
  - Sind noch nicht markierte Nachbarn von G vorhanden? Ja: F, H
  - Markiere diese und lege sie auf Stapel



H
F
D
B

  - Bisher bearbeitet: E I G
- Durchgang 4:
  - $v_{Aktuell}$ = H , entferne H vom Stapel, bearbeite H
  - Sind noch nicht markierte Nachbarn von H vorhanden? Nein

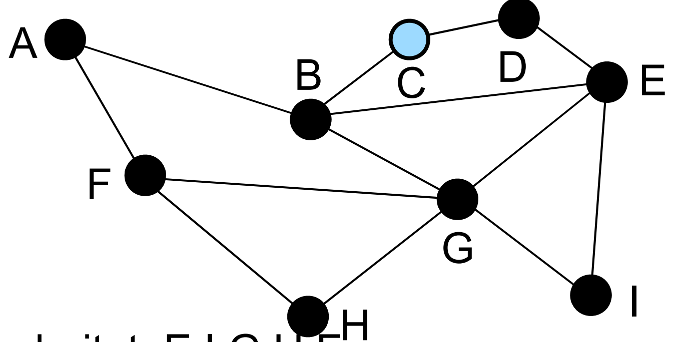


F
D
B

  - Bisher bearbeitet: E I G H

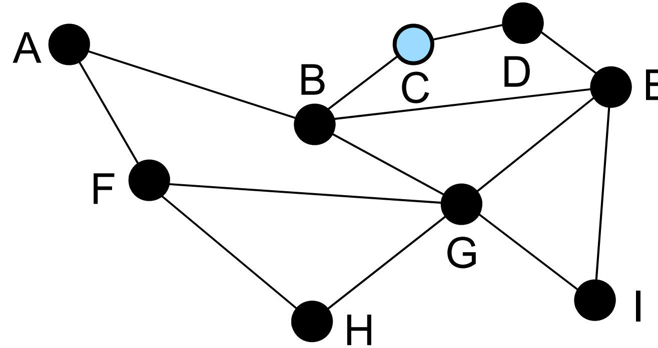
Beispiel (IV)

- Durchgang 5:
  - $v_{Aktuell}$ = F, entferne F vom Stapel, bearbeite F
  - Sind noch nicht markierte Nachbarn von F vorhanden? Ja: A
  - Markiere diese und lege sie auf Stapel



A
D
B

  - Bisher bearbeitet: E I G H F
- Durchgang 6:
  - $v_{Aktuell}$ = A , entferne A vom Stapel, bearbeite A
  - Sind noch nicht markierte Nachbarn von A vorhanden? Nein

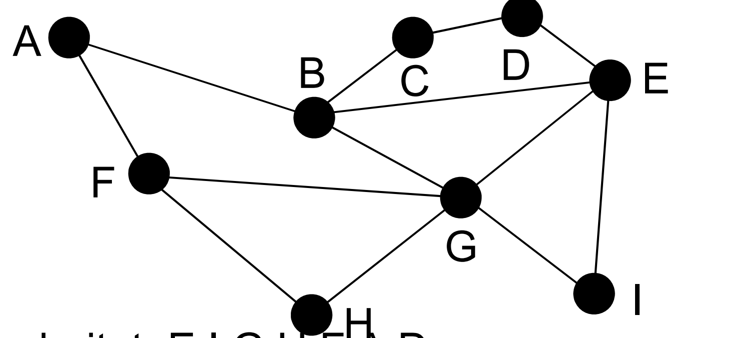


D
B

  - Bisher bearbeitet: E I G H F A

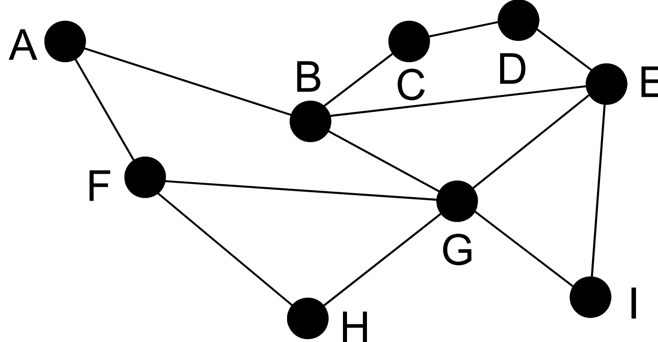
Beispiel (V)

- Durchgang 7:
  - $v_{Aktuell}$ = D, entferne D vom Stapel, bearbeite D
  - Sind noch nicht markierte Nachbarn von D vorhanden? Ja: C
  - Markiere diese und lege sie auf Stapel



C
B

  - Bisher bearbeitet: E I G H F A D
- Durchgang 8:
  - $v_{Aktuell}$ = C , entferne C vom Stapel, bearbeite C
  - Sind noch nicht markierte Nachbarn von C vorhanden? Nein

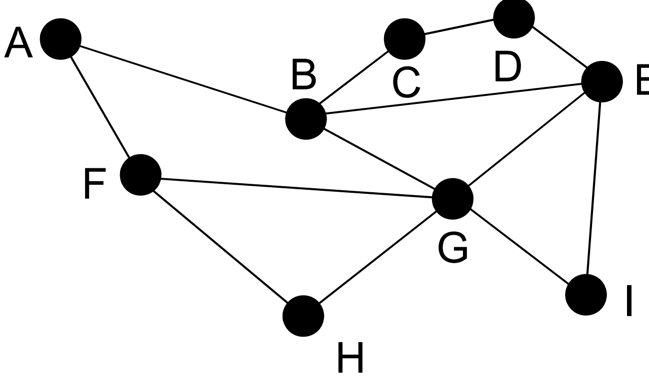


B

  - Bisher bearbeitet: E I G H F A D C

Beispiel (VI)

- Durchgang 9:
  - $v_{Aktuell}$ = B, entferne B vom Stapel, bearbeite B
  - Sind noch nicht markierte Nachbarn von B vorhanden? Nein



  - Bisher bearbeitet: E I G H F A D C B
- Stapel leer ⇒ Algorithmus terminiert

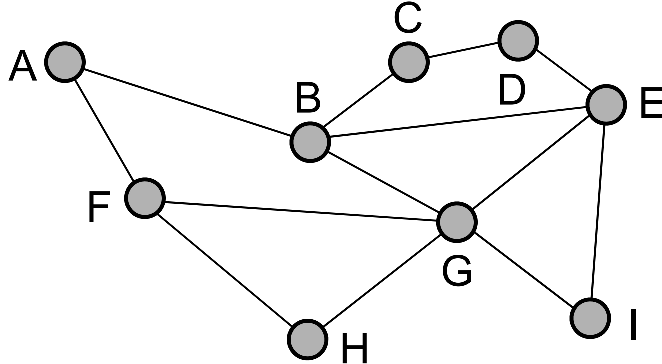
Algorithmus rekursiv

- Gegeben:
  - Ungerichteter Graph G = (V,E)
  - Startknoten $v_s \in V$
  - Feld `marked` gibt an, ob Knoten markiert (true) oder nicht (false)
  - Funktion `neighbours(v)` liefert Menge der Nachbarn von Knoten v

```
( 1) // Initialisierungen
( 2) for each v ∈ V {
( 3)   marked[v] = false;
( 4) }
( 5) depthFirstSearchRecursive(vs);
( 6) // Rekursive Prozedur
( 7) proc depthFirstSearchRecursive(Knoten k){
( 8)   if (!marked[k]){
( 9)     marked[k] = true;
(10)     output(k);
(11)     for each v ∈ neighbours(k){
(12)       depthFirstSearchRecursive(v);
(13)     }
(14)   }
(15) }
```

Beispiel (I)

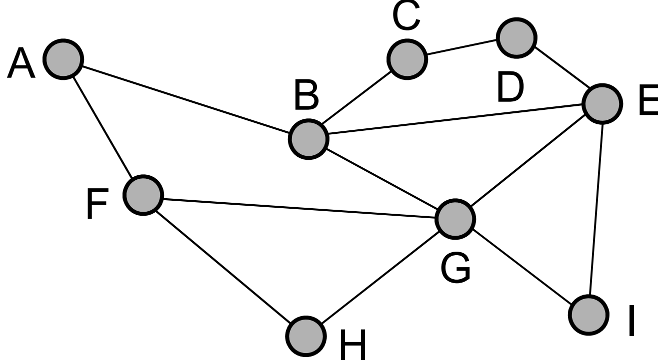
- Tiefensuche in folgendem Graphen mit Startknoten E



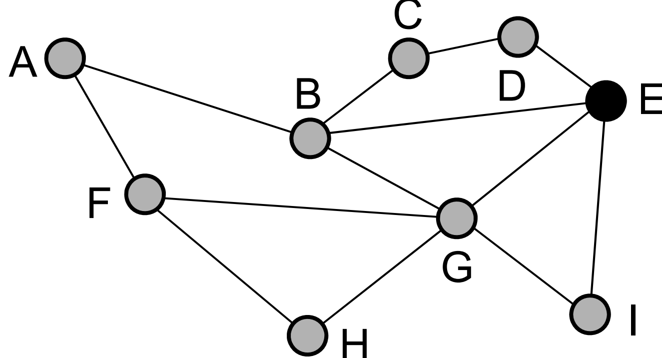
- Annahme:
  - Nachbar eines Knoten in aufsteigender Folge behandeln (`for each`-Schleife in Zeile 11)
  - Markierter Knoten = schwarz
  - Nicht markierter Knoten = grau

Beispiel (II)

- Initialisierung: Markiere alle Knoten mit `false`

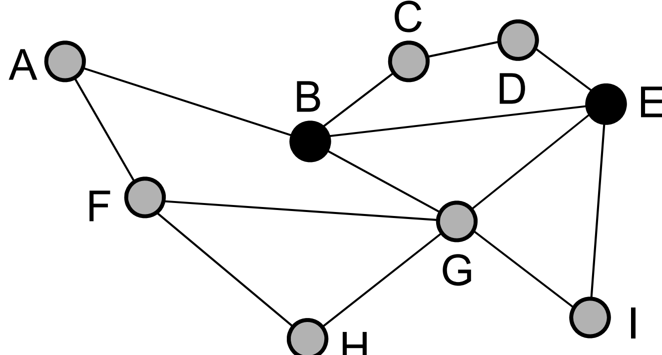


- Durchgang 1: Rufe Prozedur für Startknoten E auf
  - Markiere E
  - Gebe E aus, bisherige Reihenfolge: E
  - Rufe den Algorithmus rekursiv für die Nachbarn von E auf: B, D, G, I



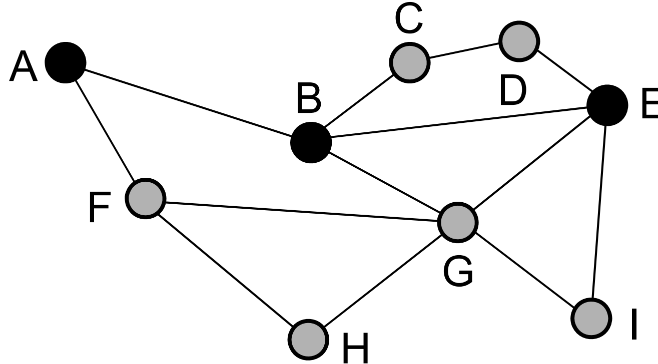
Beispiel (III)

- Durchgang 2: Rufe Prozedur für Knoten B auf
  - Markiere B
  - Gebe B aus, bisherige Reihenfolge: E B
  - Rufe den Algorithmus rekursiv für die Nachbarn von B auf: A, C, E, G



Beispiel (IV)

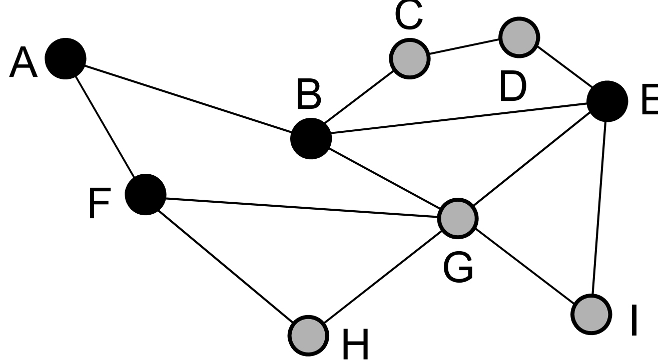
- Durchgang 3: Rufe Prozedur für Knoten A auf
  - Markiere A
  - Gebe A aus, bisherige Reihenfolge: E B A
  - Rufe den Algorithmus rekursiv für die Nachbarn von A auf: B, F



- Durchgang 4: Rufe Prozedur für Knoten B auf
  - B bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (V)

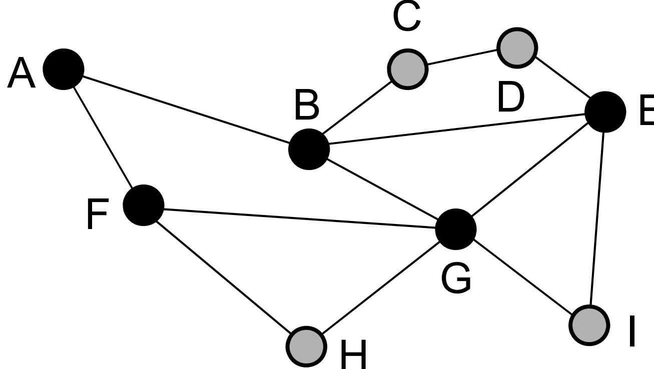
- Durchgang 5: Rufe Prozedur für Knoten F auf
  - Markiere F
  - Gebe F aus, bisherige Reihenfolge: E B A F
  - Rufe den Algorithmus rekursiv für die Nachbarn von F auf: A, G, H



- Durchgang 6: Rufe Prozedur für Knoten A auf
  - A bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (VI)

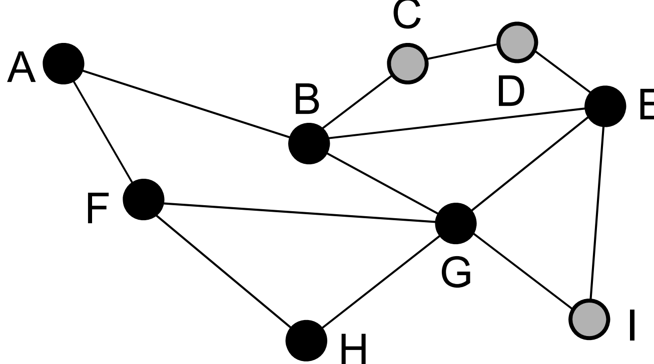
- Durchgang 7: Rufe Prozedur für Knoten G auf
  - Markiere G
  - Gebe G aus, bisherige Reihenfolge: E B A F G
  - Rufe den Algorithmus rekursiv für die Nachbarn von G auf: B, E, F, H, I



- Durchgang 8: Rufe Prozedur für Knoten B auf
  - B bereits markiert, keine Aktion im Prozedurrumpf
- Durchgang 9: Rufe Prozedur für Knoten E auf
  - E bereits markiert, keine Aktion im Prozedurrumpf
- Durchgang 10: Rufe Prozedur für Knoten F auf
  - F bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (VII)

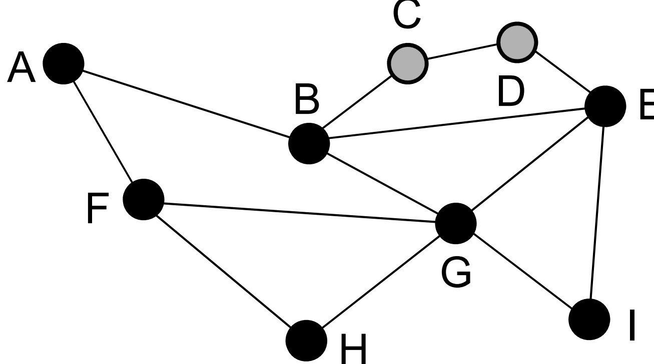
- Durchgang 11: Rufe Prozedur für Knoten H auf
  - Markiere H
  - Gebe H aus, bisherige Reihenfolge: E B A F G H
  - Rufe den Algorithmus rekursiv für die Nachbarn von H auf: F, G



- Durchgang 12: Rufe Prozedur für Knoten F auf
  - F bereits markiert, keine Aktion im Prozedurrumpf
- Durchgang 13: Rufe Prozedur für Knoten G auf
  - G bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (VIII)

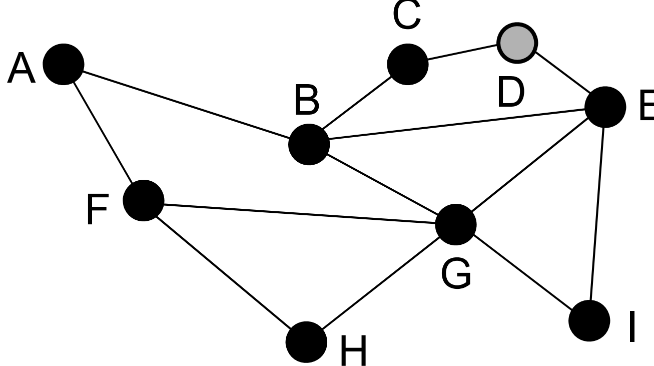
- Durchgang 14: Rufe Prozedur für Knoten I auf (aus Durchgang 7)
  - Markiere I
  - Gebe I aus, bisherige Reihenfolge: E B A F G H I
  - Rufe den Algorithmus rekursiv für die Nachbarn von I auf: E, G



- Durchgang 15: Rufe Prozedur für Knoten E auf
  - E bereits markiert, keine Aktion im Prozedurrumpf
- Durchgang 16: Rufe Prozedur für Knoten G auf
  - G bereits markiert, keine Aktion im Prozedurrumpf
- Durchgang 17: Rufe Prozedur für Knoten H auf (aus Durchgang 5)
  - H bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (IX)

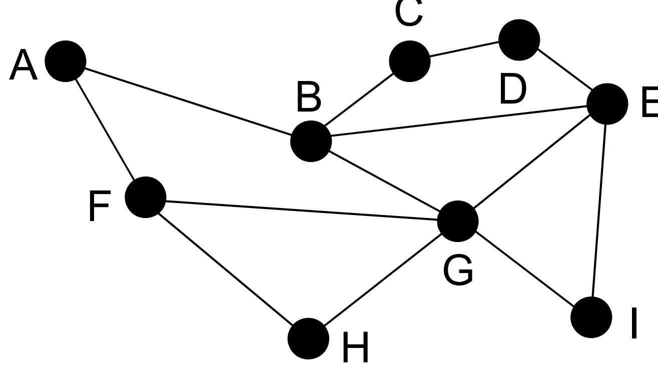
- Durchgang 18: Rufe Prozedur für Knoten C auf (aus Durchgang 2)
  - Markiere C
  - Gebe C aus, bisherige Reihenfolge: E B A F G H I C
  - Rufe den Algorithmus rekursiv für die Nachbarn von C auf: B, D



- Durchgang 19: Rufe Prozedur für Knoten B auf
  - B bereits markiert, keine Aktion im Prozedurrumpf

Beispiel (X)

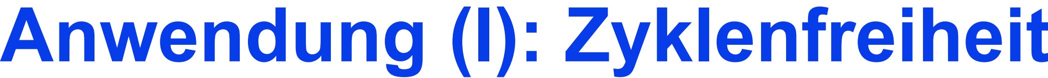
- Durchgang 20: Rufe Prozedur für Knoten D auf
  - Markiere D
  - Gebe D aus, bisherige Reihenfolge: E B A F G H I C D
  - Rufe den Algorithmus rekursiv für die Nachbarn von D auf: C, E



- Durchgang 21ff: Rekursion baut sich ab

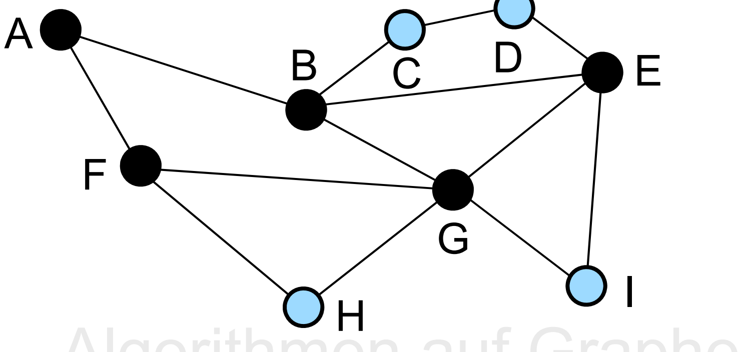
Komplexität

- Komplexität: $O(n+m)$
  - n Schritte sind notwendig, um alle Knoten als nicht markiert zu initialisieren
  - Maßgebend ist Anzahl rekursiver Aufrufe
  - Entspricht $2 \cdot m$ (Jede Kante wird von jedem Knoten aus aufgerufen)



Anwendung (I): Zyklenfreiheit

- Häufiges Problem: Zyklenfreiheit eines Graphen muss nachgewiesen werden
- Beispiel:
  - Knoten sind zu erledigende Aufgaben
  - Kante von A nach B existiert, falls A vor B erledigt werden muss
  - Nur zyklenfreier Graph gibt konsistente Definitionen wieder
- Test auf Zyklenfreiheit kann mittels Tiefensuche gelöst werden
- Bedingung hierbei: Ist beim rekursiven Aufruf ein Nachbar (außer dem gerade zuvor besuchten) bereits markiert, dann liegt ein Zyklus vor
- Im Beispiel ist dies in Durchgang 7 der Fall:
  - G ist der aktuelle Knoten
  - Mit B und E gibt es bereits markierte Nachbarn



Anwendung (II): Zusammenhang (I)

- Modifikation der Tiefensuche
- Idee:
  - Iteration über alle Knoten
  - Finde von hier alle erreichbaren Knoten
  - Vergebe pro Zusammenhangskomponente eine Nummer
  - Knoten werden mit dieser Nummer markiert (Feld `ZKNummer`)
  - Knoten mit gleicher Nummer bilden Zusammenhangskomponente

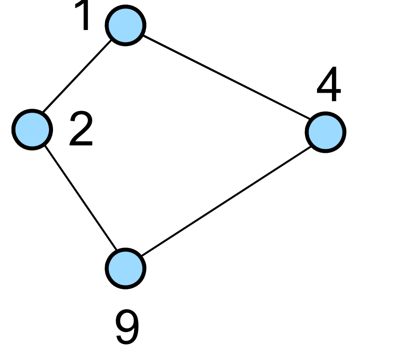
Anwendung (II): Zusammenhang (II)

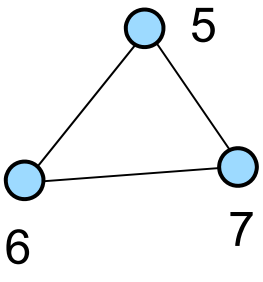
```
( 1) // Initialisierungen
( 2) for each v ∈ V {
( 3)   ZKNumber[v] = 0;
( 4) }
( 5) counter = 0;
( 6) // Iteration über alle Knoten
( 7) for each v ∈ V {
( 8)   if (ZKNumber[v] == 0){
( 9)     counter++;
(10)     connectivity(v);
(11)   }
(12) }
```

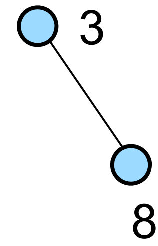
```
( 1) void connectivity(v){
( 2)   ZKNumber[v] = counter
( 3)   for each w ∈ neighbours(v){
( 4)     if (ZKNumber[w] == 0){
( 5)       connectivity(w);
( 6)     }
( 7)    }
( 8) }
```

Beispiel (I)

- Finde Zusammenhangskomponenten für Graphen G:







- Initialisierung:

Knoten	1	2	3	4	5	6	7	8	9
ZKNumber	0	0	0	0	0	0	0	0	0

```
counter = 0
```

Beispiel (II)

- Iteration für Knoten 1:
  - Bedingung Zeile 8 erfüllt, `counter` wird inkrementiert, `connectivity` aufgerufen, alle Knoten der Komponente mit 1 markiert

Knoten	1	2	3	4	5	6	7	8	9
ZKNumber	1	1	0	1	0	0	0	0	1

`counter = 1`

- Iteration für Knoten 2:
  - Bedingung Zeile 8 nicht erfüllt, keine Aktion
- Iteration für Knoten 3:
  - Bedingung Zeile 8 erfüllt, `counter` wird inkrementiert, `connectivity` aufgerufen, alle Knoten der Komponente mit 2 markiert

Knoten	1	2	3	4	5	6	7	8	9
ZKNumber	1	1	2	1	0	0	0	2	1

`counter = 2`

Beispiel (III)

- Iteration für Knoten 4:
  - Bedingung Zeile 8 nicht erfüllt, keine Aktion
- Iteration für Knoten 5:
  - Bedingung Zeile 8 erfüllt, `counter` wird inkrementiert, `connectivity` aufgerufen, alle Knoten der Komponente mit 3 markiert

Knoten	1	2	3	4	5	6	7	8	9
ZKNumber	1	1	2	1	3	3	3	2	1

`counter = 3`

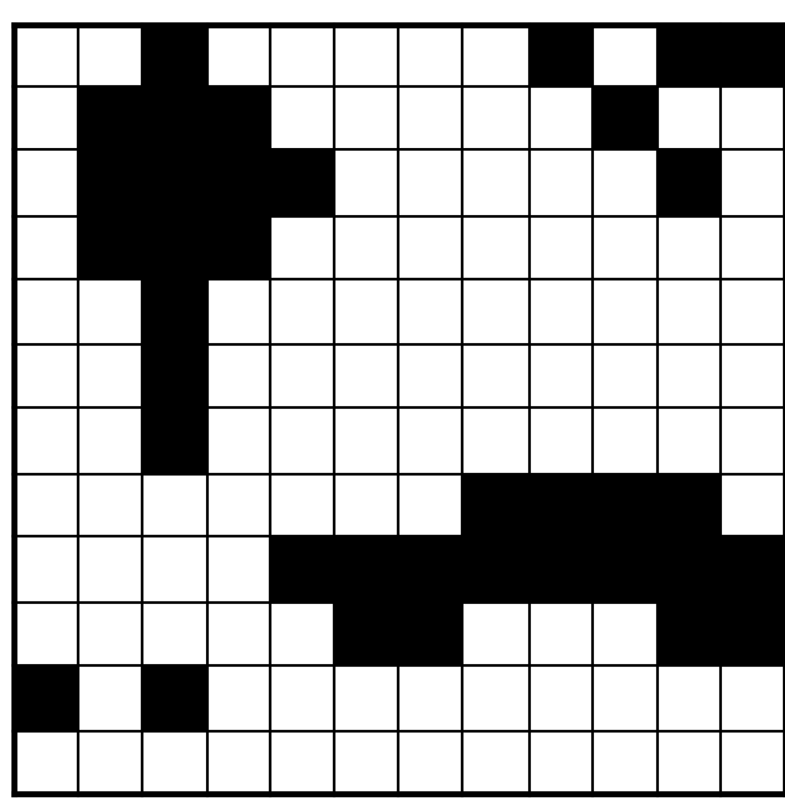
- Iterationen für Knoten 6 bis 9:
  - Bedingung Zeile 8 jeweils nicht erfüllt, keine Aktion

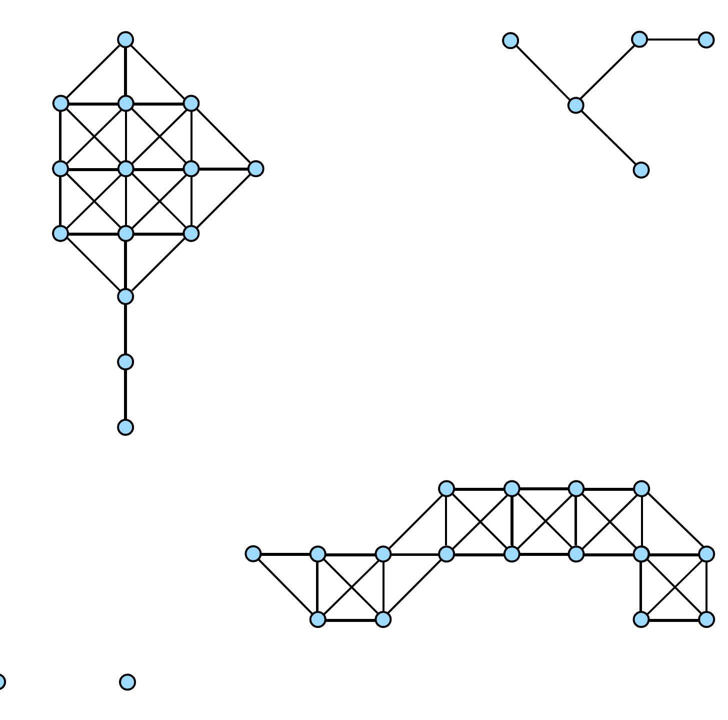
Anwendung (III): Bildverstehen (I)

- Wesentliches Teilgebiet der Künstlichen Intelligenz (KI)
- Einzelne Bildpunkte müssen zu Gebilden zusammengefasst und als Objekte erkannt werden
- Ausgangspunkt: Digitalisiertes Bild (d.h. Grauwerte)
- Überführung in binäres (schwarz-weißes) Bild
- Bildpunkte mit Wert 1 sind Objekte, die anderen Hintergrund
- Definition folgender Graph (bez. als Nachbarschaftsgraph):
  - Bildpunkte mit Wert 1 sind Knoten
  - Kante zwischen zwei Bildpunkten, wenn diese benachbart
  - Nachbarschaft:
    - 8-Zusammenhang: Berücksichtigung aller 8 Nachbarn
    - 4-Zusammenhang: Berücksichtigung nur Nachbarn oben, unten, links und rechts
- Graph analysieren:
  - Zusammenhangskomponenten Teile des Bildes
  - Zu kleine Komponenten können Rauschen sein

Anwendung (III): Bildverstehen (II)

- Beispiel: Binäres Bild und zugehöriger Nachbarschaftsgraph





[Tura04]

- Tiefensuche kann unmittelbar auf binäre Bildmatrix angewendet werden

Zusammenfassung (I)

- Transitiver Abschluss:
  - Definition
  - Adjazenzmatrizenmultiplikationsverfahren
  - Warshall-Algorithmus
  - Anwendungsbeispiel

- Minimal aufspannende Bäume:
  - Definition
  - Algorithmus von Prim
  - Algorithmus von Kruskal
  - Anwendungsbeispiele

Zusammenfassung (II)

- Suchverfahren in Graphen:
  - Tiefensuche ↔ Breitensuche

- Breitensuche:
  - Algorithmus mit Schlange
  - Anwendungsbeispiele

- Tiefensuche:
  - Algorithmus mit Stapel
  - Rekursiver Algorithmus
  - Zyklenfreiheit
  - Zusammenhangskomponenten
  - Anwendungsbeispiele

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
V P T V A Q F E U G X Z C B G C Y S C W B O M M L
O P Z R F K V G U T O X R N W W G W C M T Q H I R
E V J M A S H C N L B E E I Z N Q T P J I Z D N E
V E U U L N E P L P I M K V K S E J K X I C U I H
P H R N S U S O R T Y J K X H G J I G D N R X M H
D Y X X Q T C I E H C U S N E F E I T V P A E A I
E C O J Q C S N T X F D O O B M D M U F O N V L S
I Q L C Q N S S D I K U P Z J S U W O Y W O Z E Q
E P G Y D U G V O N V U K N Z U M H V Y I F Z R Y
B X I U C C G E D U H E Z B F T N Y M N X U Y S L
L O Z H N Z Y V B I I T R K F Y I T A H H D K P Y
F V E I N T W L R A Z M U A H N A P S Z Q B R A J
Z R A E M V T J Y R Z U Q V B Y S P V M R D K N S
M M G U R Y Z E Y E K Y C P T S L C E Q H I M N W
H H Y S B G L H N O V I U T H C C M H I N W G B J
W R N P F Y D W M D S A S O J K K H Z X S H O A X
S F X O Y M P F V E S U T V S V V P L S C I G U S
Q D P J Q O L U U G H S X O R X S B M U V B E M X
K A K D I R K R J T P V Y Y H I A X G F S M S M P
E R R E I C H B A R K E I T S M A T R I X S J A U
```

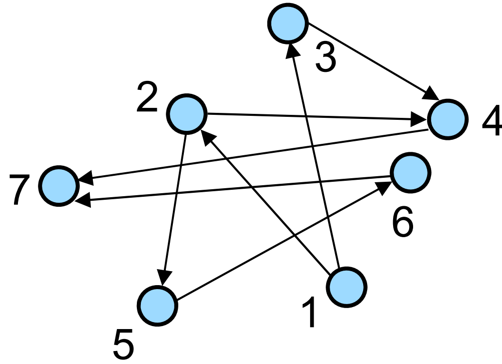
Übungen (II)

- **Aufgabe 2**
  a) Was ist der transitive Abschluss eines Graphen?
  b) Was ist die Erreichbarkeitsmatrix?
  c) Was berechnet der Warshall-Algorithmus und wie funktioniert er?
  d) Was ist ein (minimaler) Spannbaum?
  e) Welcher Algorithmus berechnet einen minimalen Spannbaum?
  f) Welche Suchverfahren in Graphen gibt es?
  g) Wie funktioniert Breitensuche, wie Tiefensuche?
  h) Nennen Sie Anwendungen der Breiten- und der Tiefensuche?

Übungen (III)

- **Aufgabe 3**

  Gegeben sei der folgende gerichtete Graph G.



a) Geben Sie den transitiven Abschluss von G visuell an!

b) Ermitteln Sie den transitiven Abschluss mit Hilfe der beiden in der VL vorgestellten Verfahren!

c) Wie ist der Aus- bzw. Eingangsgrad von Knoten 4?

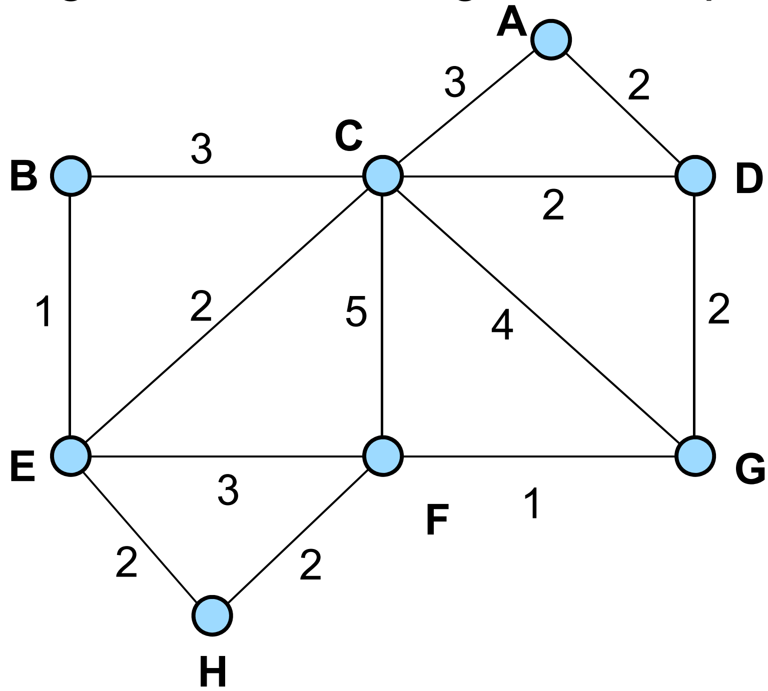
d) Ist G regulär? Warum (nicht)?

e) Ist G azyklisch?

f) Ist G planar?

Übungen (IV)

Gegeben sei der folgende Graph G:



- **Aufgabe 4**

  Ermitteln Sie für G durch Anwendung des Algorithmus von Prim drei verschiedene minimal aufspannende Bäume!

- **Aufgabe 5**

  Ermitteln Sie für G durch Anwendung des Algorithmus von Kruskal einen minimal aufspannenden Baum!

Übungen (V)

- **Aufgabe 6**

  Der folgende rekursive Algorithmus A konstruiert einen aufspannenden Baum:

  Gegeben: Ungerichteter Graph G = (V,E)

  (1) Markiere einem beliebigen Knoten $v \in V$

  (2) Wiederhole für alle von v ausgehenden Kanten $(v,v') \in E$:

  - Wenn v' unmarkiert, dann markiere v' und rufe A rekursiv für v' auf
  - Sonst: Lösche Kante (v,v'), wenn v' nicht die Kante ist, von der aus v markiert worden ist

  a) Welche Komplexität hat dieser Algorithmus?

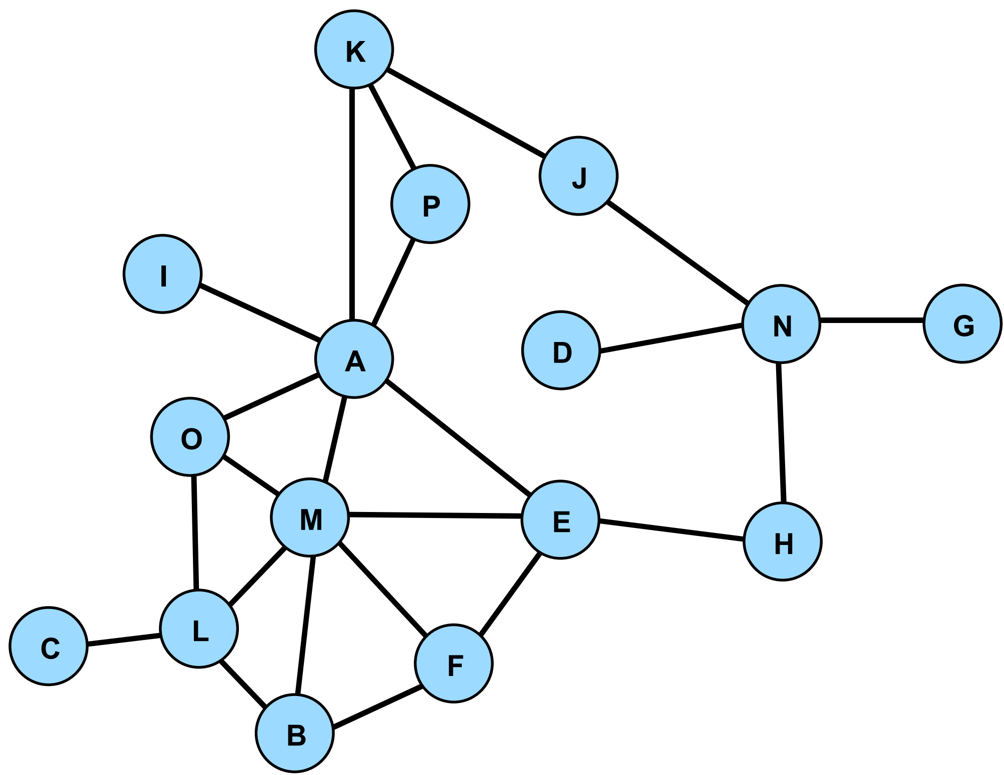
  b) Demonstrieren Sie die Arbeitsweise des Algorithmus am Beispiel des Graphen aus Aufgabe 4!

  c) Ist der ermittelte Spannbaum minimal?

Übungen (VI)

- **Aufgabe 7**

  Gegeben sei der folgende Graph:

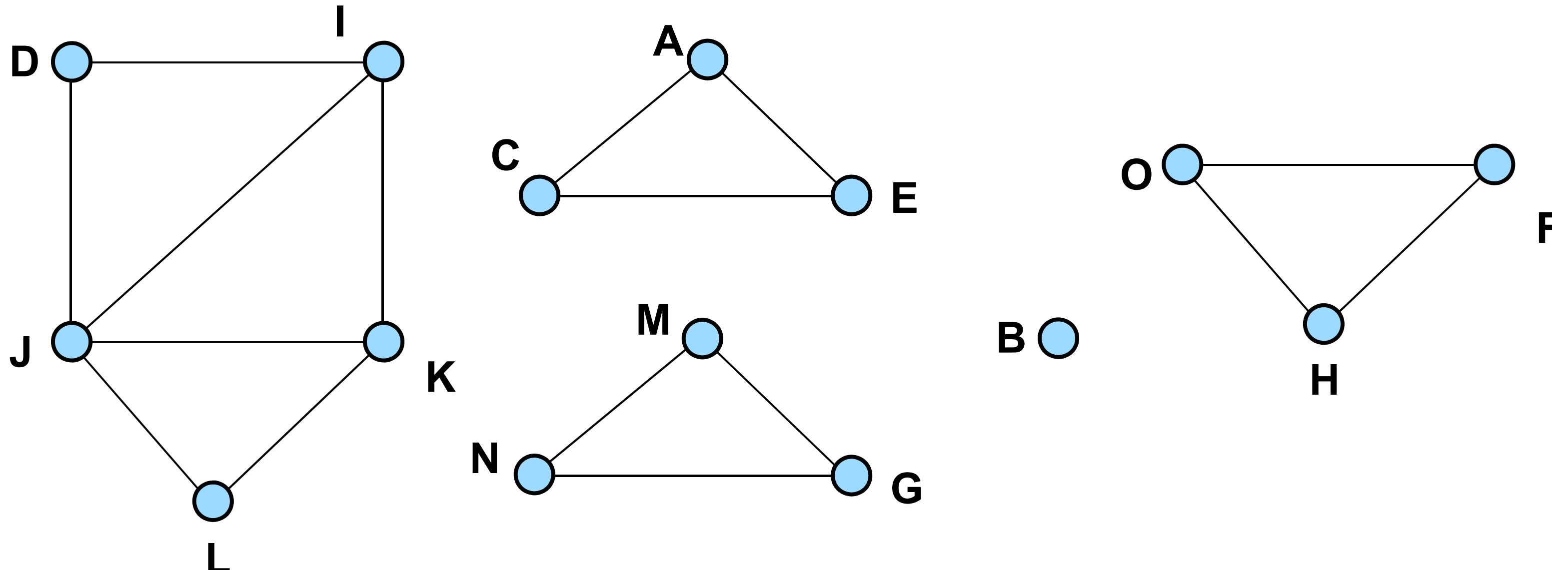


Geben Sie das Resultat eines Breiten- und eines Tiefendurchlaufs, beginnend bei Knoten A, an! Geben Sie auch die einzelnen Schritte des jeweiligen Algorithmus an!

Übungen (VII)

- **Aufgabe 8**

  Wenden Sie auf folgenden Graphen den Algorithmus zum Auffinden von Zusammenhangskomponenten an:



Algorithmen und Datenstrukturen

Teil 6:
Algorithmen auf Graphen (II) –
Färbungen, kürzeste Wege, Euler/Hamilton

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 05/2020

Gliederung

- Färbungen

- Kürzeste Wege

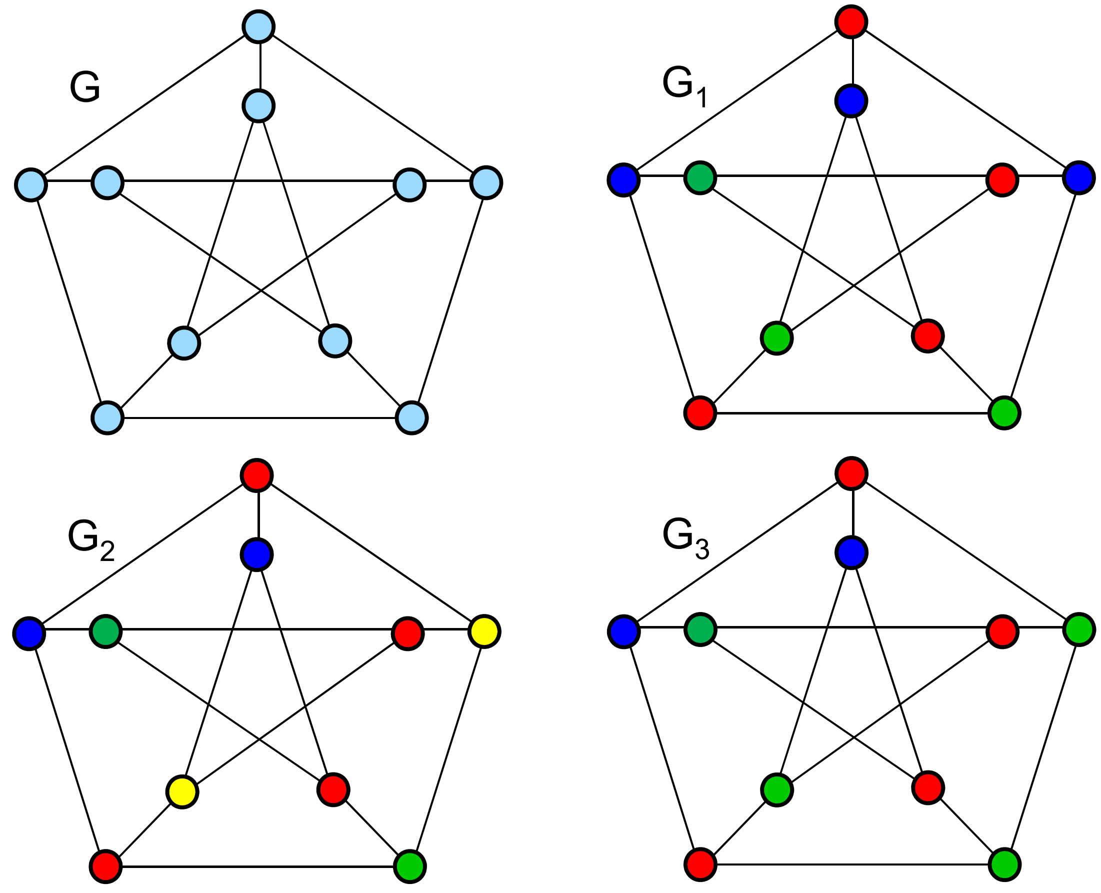
- Euler- und Hamilton-Kreise

Färbungsproblem (I)

- Def.: Färbung eines (ungerichteten) Graphen G:
  - Zuordnung von Farben zu Knoten von G
  - Zwei benachbarte Knoten dürfen nie gleiche Farbe haben
- Positive Ganzzahl m gibt Anzahl Farben an (m-Färbung bzw. m-färbbar))
- Def.:

  Sei G Graph. Chromatische Zahl *chromatic*(G) (oder χ(G)) kleinste Zahl m, für die m-Färbung existiert
- Färbung mit genau *chromatic*(G) Farben heißt minimale Färbung
- Färbung mit |V| Farben heißt triviale Färbung

Färbungsproblem (II): Beispiele



- G_1 und G_2 Färbungen von G, G_3 keine Färbung von G
- G_1 3-Färbung, G_2 4-Färbung
- *chromatic*(G) = 3
- G_1 ist eine minimale Färbung von G

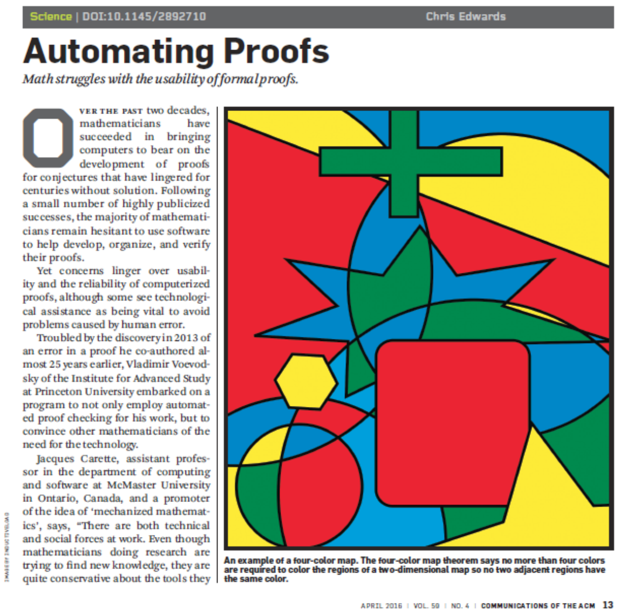
Färbungsproblem (III): Versionen

- Entscheidungsproblem:
  - Gegeben ungerichteter Graph G und positive Ganzzahl m
  - Ist G mit m Farben färbbar?

- Optimierungsproblem:
  - Gegeben ungerichteter Graph G
  - Wie viele Farben werden mindestens für die Färbung benötigt? (= Finde *chromatic*(G))

Färbungsproblem (IV): Geschichte

- Färbungsproblem eines der ältesten bekannten Probleme der Graphentheorie
- Vermutung seit 19. Jhdt.: Jeder planare Graph ist 4-färbbar
- Endgültiger Beweis erst 1976
- Computer(unterstütze) Vorgehensweise [z.B. Ed16]:



Science | DOI:10.1145/2892710 Chris Edwards

Automating Proofs

Math struggles with the usability of formal proofs.

Over the past two decades, mathematicians have succeeded in bringing computers to bear on the development of proofs for conjectures that have lingered for centuries without solution. Following a small number of highly publicized successes, the majority of mathematicians remain hesitant to use software to help develop, organize, and verify their proofs.

Yet concerns linger over usability and the reliability of computerized proofs, although some see technological assistance as being vital to avoid problems caused by human error.

Troubled by the discovery in 2013 of an error in a proof he co-authored almost 25 years earlier, Vladimir Voevodsky of the Institute for Advanced Study at Princeton University embarked on a program to not only employ automated proof checking for his work, but to convince other mathematicians of the need for the technology.

Jacques Carette, assistant professor in the department of computing and software at McMaster University in Ontario, Canada, and a promoter of the idea of 'mechanized mathematics', says, "There are both technical and social forces at work. Even though mathematicians doing research are trying to find new knowledge, they are quite conservative about the tools they

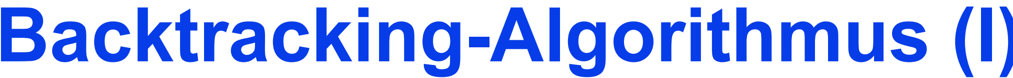
An example of a four-color map. The four-color map theorem says no more than four colors are required to color the regions of a two-dimensional map so no two adjacent regions have the same color.

APRIL 2016 | VOL. 59 | NO. 4 | COMMUNICATIONS OF THE ACM 13

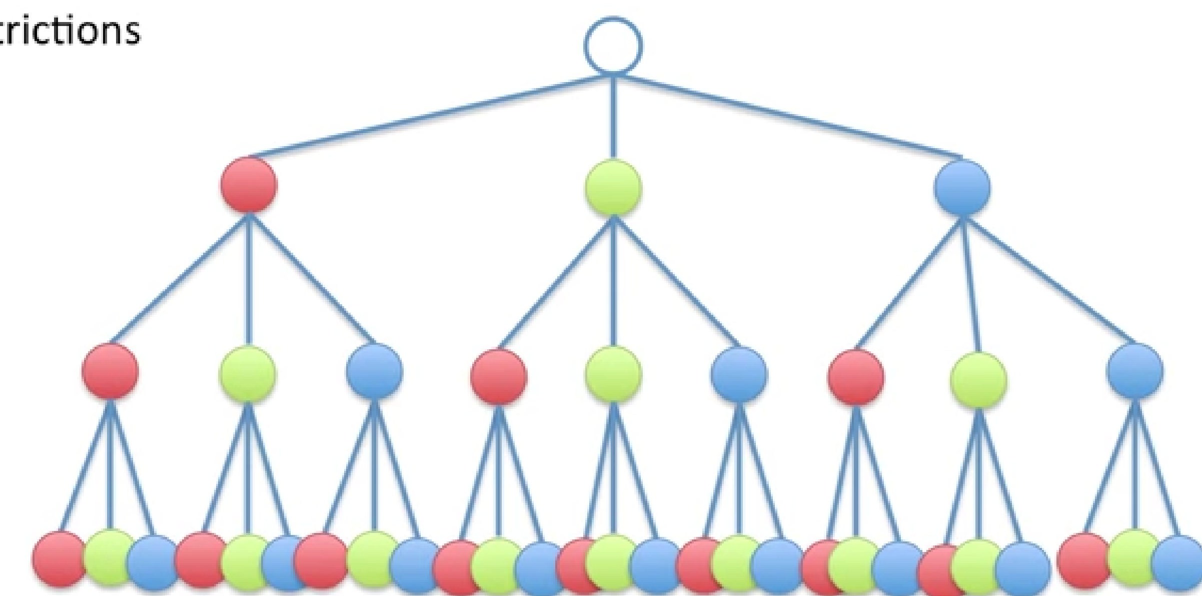
[Chris Edwards: Automating Proofs. Commun. ACM 59(4): 13-15 (2016)]

Similar issues of trust greeted the first computerized proof of the theorem that argued only four colors are needed to distinguish between adjacent regions on a 2D map. Forty years ago, working at the University of Illinois at Urbana-Champaign, Kenneth Appel and Wolfgang Haken developed computer programs to demonstrate there were no counterexamples to the theorem. Still, the programs were tedious to check by hand.

Backtracking-Algorithmus (I)



- Prinzip:
  - Systematisches Durchsuchen gesamter Lösungsraum
- Beispiel:
  - Sei n Anzahl der Knoten und m Anzahl der Farben
  - Sei n = 3, m = 3
  - Lösungsraum ohne Restriktionen:



[https://www.youtube.com/watch?v=miCYGGrTwFU]

Backtracking-Algorithmus (II)

- Algorithmus:

```
( 1) graphColour(int k){
( 2)    for c=1 to m{
( 3)       if(isSafe(k,c)){
( 4)          x[k] = c
( 5)          if((k+1)<n){
( 6)             graphColour(k+1)}
( 7)          else{
( 8)             print x[]
( 9)             return
(10)          }
(11)       }
(12)    }
(13) }
```

  - m: Anzahl Farben
  - k: In aktuellem Rekursionslevel zu färbender Knoten
  - x[k]: Feld mit aktuellen Farben für alle Knoten
  - Methode `isSafe` prüft, ob Knoten k Farbe c zugewiesen werden kann

[https://www.youtube.com/watch?v=miCYGGrTwFU]

Backtracking-Algorithmus (III)

- Methode `isSafe`:

```
( 1) boolean isSafe(int k, int c){
( 2)   for i=1 to n{
( 3)     if(areAdjacent(k,i) && x[i]==c){
( 4)       return false;
( 5)     }
( 6)   }
( 7)   return true;
( 8) }
```

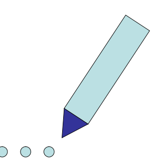
[https://www.youtube.com/watch?v=miCYGGrTwFU]

Backtracking-Algorithmus (IV)

- Algorithmus: Siehe Codebeispiel (https://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/)
- Experiment 1:
  - Laufzeitmessung für K_n
  - Anzahl Farben n-1

n	Laufzeit [HH:MM:SS]
1..8	0
9	00:00:00,016
10	00:00:00,107
11	00:00:01,164
12	00:00:13,725
13	00:03:00,390
14	>12 Stunden
15	

- Experiment 2:
  - Laufzeitmessung für K_n
  - Anzahl Farben n
  - Laufzeit für n=100 wenige Millisekunden
  - Warum?



Backtracking-Algorithmus (VI)

- Analyse:
  - An jedem inneren Knoten: O(n·m) Zeit zur Bestimmung der Kindknoten für legale Färbung
  - Damit totale Zeit:

$$\sum_{i=0}^{n-1} m^{i+1} \cdot n = \sum_{i=1}^{n} m^{i} \cdot n = \frac{n(m^{n+1} - 2)}{(m-1)} = O(n \cdot m^{n})$$

- Also:
  - Exponentielles Wachstum Zeitkomplexität mit Anzahl Knoten
  - Suche nach Alternativen notwendig

Greedy-Färbe-Algorithmus (I): Prinzip

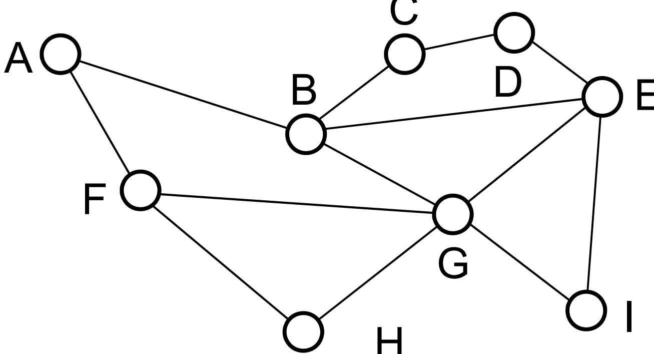
- farben = {1,2,...} stehen zur Verfügung
- Pro Durchgang:
  - Vergabe einer Farbe
  - Behandlung aller Knoten
- Wurde Knoten k gefärbt, dann:
  - Kein Nachbar von k kann aktuelle Farbe bekommen
  - Kennzeichnung entsprechender Knoten mit negativem Wert (-farbe)
- Wurde Knoten k noch nicht gefärbt und ist nicht –farbe, wird k mit aktueller Farbe markiert
- Letzter Punkt: Greedy-Schritt („Färbe alles, was Du kriegen kannst.“)

Greedy-Färbe-Algorithmus (II): Pseudocode

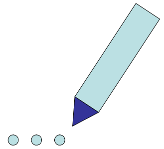
```
( 1) farbe=0;          // Aktuelle Farbe
( 2) z=0;              // Anzahl bereits markierter Knoten
( 3) f[1..n]=0;        // f[i] : Farbe des i-ten Knotens
( 4) while (z < n){
( 5)   farbe++;
( 6)   for „Jeden Knoten i“ {
( 7)     if (f[i]==0 && f[i]!=-farbe){
( 8)       f[i]=farbe;
( 9)       z++;
(10)       for „Jeden Nachbarn von i“{
(11)          if (f[j]==0){
(12)            f[j]=-farbe;
(13)          }
(14)       }
(15)     }
(16)   }
(17)   Setze alle f[i] mit f[i]=-farbe auf f[i]=0;
(18) }
```

Greedy-Färbe-Algorithmus (III): Beispiel

- Graph



  soll mit den Farben = {rot (1), gelb (2), blau(3), grün(4), ... } gefärbt werden
- Knoten sollen dabei in der Reihenfolge ihrer Namen durchlaufen werden



Laufzeit

- Straight-Forward-Implementierung: $O(n^2)$
- Verbesserung:
  - Verwalte für jeden Knoten i kleinste unbenutzte Farbnummer der Nachbarn
  - Kann mit Aufwand O(degree(i)) realisiert werden
  - Idee:
    - Feld `vergeben` der Länge Δ +1 (Δ max. Grad eines Knoten)
    - In `vergeben` werden in jedem Durchgang die schon an Nachbarn vergebenen Farben eingetragen (vergeben[i] = 0, wenn Farbe i schon vergeben, sonst 1)
    - Feld vergeben wird nicht in jedem Durchgang zurückgesetzt (wäre Aufwand $O(\Delta \cdot n)$)
    - Vielmehr: In jedem Durchgang zunächst vergebene Farben markieren und später rückgängig machen (Aufwand O(degree(i))
    - Bestimmung kleinster nicht vergebener Farbnummer (d.h. Suche nach der ersten 0 im Feld vergeben) hat gleichen Aufwand
    - Insgesamt: O (n+m)
- Also: Kleine Änderung am Algorithmus führt zu verbesserter Laufzeit

Verbesserter Färbe-Algorithmus

```
( 1) farbe=0;       // Aktuelle Farbe
( 2) c=0;           // Anzahl benötigter Farben
( 3) f[1..n]=0;     // f[i] : Farbe des i-ten Knotens
( 4) v[1..Δ+1]=0;  // v[i] : Farbe i vergeben ja(1), nein(0)
( 5) f[1]=1;
( 6) for „Jeden Knoten i außer 1“{
( 7)   for „Jeden Nachbarn j von i“{
( 8)     if (f[j]>0){
( 9)       v[f[j]]=1;
(10)     }
(11)  }
(12)     farbe=Minimum {i mit v[i]=0}
(13)     f[i]=farbe;
(14)     if (farbe>c){
(15)       c=farbe;
(16)     }
(17)     for „Jeden Nachbarn j von i“{
(18)       if (f[j]>0){
(19)           v[f[j]]=0;
(20)       }
(21)     }
(22) }
```

Fazit

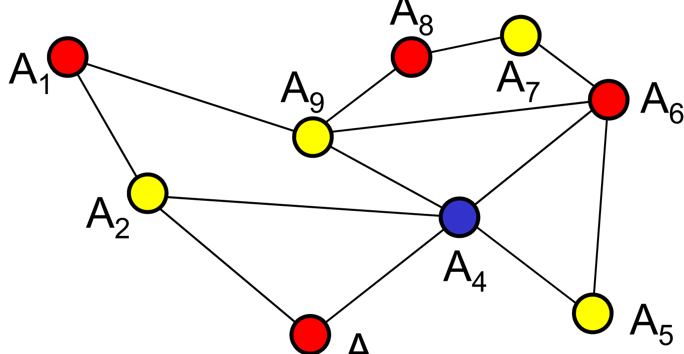
- Keine Rücknahme einmal getroffener Entscheidungen (Greedy-Strategie)
- Lösung:
  - Immer korrekt
  - Nicht immer optimal, d.h. Färbung nicht unbedingt minimal
  - Güte der Lösung stark abhängig von Nummerierung der Knoten
- Es gilt: Für einen Graphen G mit *chromatic*(G) = m existiert mindestens eine Knoten-Nummerierung, so dass Greedy-Algorithmus m-Färbung (d.h. minimale Färbung) liefert

Anwendungen (I)

- Färbung einer Landkarte (siehe auch Übung)
- Maschinenbelegung:
  - Aufgaben $A_1, \ldots, A_n$ auf verschiedene Maschinen durchführen
  - Durchführung von A_i erfordert jeweils Zeiteinheit T
  - Gleichzeitige Durchführung von Aufgaben, wenn nicht gleiche Maschine benötigt
  - Ziel der Maschinenbelegung:
    - Finden Aufgabenreihenfolge, die diese Einschränkung beachtet und zeitoptimal ist (d.h. möglichst hoher Parallelitätsgrad wird angestrebt)
  - Als Graph:
    - Jeder Knoten entspricht einer Aufgabe
    - Knoten benachbart, wenn Aufgaben nicht gleichzeitig durchführbar (Konfliktgraph)
  - Färbung des Konfliktgraphen liefert Aufgabenreihenfolge (Aufgaben gleicher Farbe zur gleichen Zeit durchführen)
  - Optimale Lösung entspricht minimaler Färbung
  - Benötigte Zeit ist Produkt aus T und der chromatischen Zahl

- Maschinenbelegung (Beispiel)
- Ausführung von neun Aufgaben $A_1, \ldots, A_9$ auf acht Maschinen $M_1, \ldots, M_8$:

M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
A_1, A_2	A_1, A_9	A_4, A_5, A_6	A_2, A_3, A_4	A_4, A_6, A_9	A_8, A_9	A_7, A_8	A_6, A_7



- Alle roten Aufgaben, alle gelben und die blaue Aufgabe können jeweils in einem Zeitslot bearbeitet werden:

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
T_1	A_1	A_1	A_6	A_3	A_6	A_8	A_8	A_6
T_2	A_2	A_9	A_5	A_2	A_9	A_9	A_7	A_7
T_3	--	--	A_4	A_4	A_4	--	--	--

Übersicht

- Färbungen

- Kürzeste Wege

- Euler- und Hamilton-Kreise

Problem

- Gegeben:
  - Kantenbewerteter Graph G = (V, E, g)
  - $v_S, v_Z \in V$ als Start- bzw. Zielknoten
- Def. Kürzester Weg:
  - Weg von v_S nach v_Z mit Kantensumme kleiner gleich jeder andere Weg von v_S nach v_Z
- Ausprägungen:
  - 1-1 : Kürzester Weg von einem Knoten v_S zu einem v_Z
  - 1-many : Kürzester Weg von einem Knoten v_S zu allen $v_Z \in V \setminus \{v_S\}$
  - many-many : Kürzester Weg von allen $v_S \in V$ zu allen $v_Z \in V \setminus \{v_S\}$

Dijkstra-Algorithmus (I): Idee

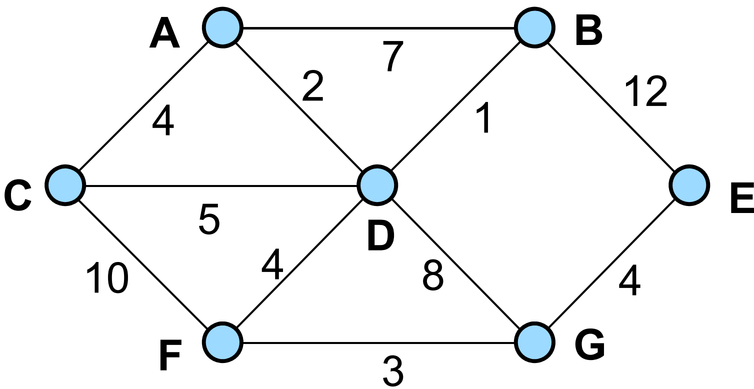
- Gegeben:
  - Kantenbewerteter Graph G = (V, E, g), G gerichtet, zusammenhängend
  - Jede Kantenbewertung sei positiv
  - $v_S, v_Z \in V$ als Start- bzw. Zielknoten
- Prinzip:
  - Aufbau eines Wurzelbaums G' = (V',E',g') mit folgenden Eigenschaften:
    - $V' \subseteq V$
    - v_S ist Wurzel von G'
    - $(v,v') \in E' \Rightarrow (v,v') \in E$ und $g'((v,v')) = g((v,v'))$
    - $\forall v \in V'$: dist(x,v) stimmt mit dem minimalen Weg von x nach v in G überein

Dijkstra-Algorithmus (II)

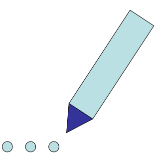
- Ablauf:
  (1) Füge den Startknoten v_S in V‘ ein
  (2) Bestimme die Menge der offenen Kanten (Teilmenge von E, die einen Startknoten v $\in$ E‘ und einen Zielknoten v‘ $\in$ V\V‘ haben)
  (3) Wähle eine offene Kante (v,v‘), für die gilt: dist(v_S,v)+g(v,v‘) ist minimal
  (4) Füge Knoten v‘ zu V‘ und Kante (v,v‘) zu E‘ hinzu
  (5) Wiederhole die Schritte (2), (3) und (4), bis G‘ den gesuchten Knoten v_Z enthält
  (6) Bestimme in G‘ den eindeutigen Weg von v_S nach v_Z
- Aufwand: $O(n^2)$
  - n Durchgänge sind im worst case notwendig
  - In jedem Durchgang sind die offenen Kanten zu bestimmen:
    - Dies sind im worst case n-1 im ersten Durchgang, n-2 im zweiten, n-3 im dritten usw.
    - Auch dies entspricht O(n)

Dijkstra-Algorithmus (III): Beispiel

- Gegeben sei der folgende Graph G:



- Anmerkung:
  - G soll gerichtet sein, die Bewertungen gelten für Hin- und Rückrichtung
  - Einzeichnen von Hin- und Rückkante wäre zu unübersichtlich
  - Annahme ist realistisch (z.B. Entfernungen auf Straßen oder Fahrtzeiten von Zügen)
- Gesucht: Kürzester Weg von C nach E



Dijkstra-Algorithmus (IV): Alle Wege berechnen

DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

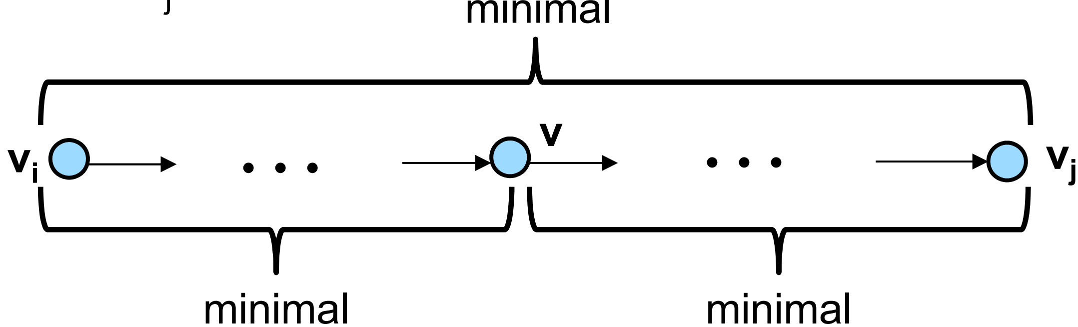
- Alle Wege := Von jedem Knoten v_1 zu jedem Knoten v_2 für alle v_1 , v_2 aus G
- Algorithmus modifizieren:
  - Starte nacheinander mit jedem Knoten als Startknoten
  - Terminierungskriterium: Statt Erreichen eines bestimmten Zielknotens Erreichen jeden Knotens

Floyd-Algorithmus (I): Idee

- Berechnung kürzester Distanz zwischen allen Knotenpaaren (v_i, v_j)
- Annahmen:
  - Jede Kante ist mit Distanzwert Wert dist >=0 markiert
  - Für alle i: dist (v_i, v_i) = 0
  - Existiert keine Kante (v_i, v_j) => dist (v_i, v_j) = ∞
  - Rechenregeln:
    - $x + \infty := \infty$
    - $\infty + \infty := \infty$
    - $\min(x, \infty) := x$
- Definiere Distanzmatrix D:
  - Hauptdiagonale in D ist 0
  - Bei direkter Kante trage dist (v_i, v_j) ein
  - Alle anderen Distanzen werden als ∞ angenommen

Floyd-Algorithmus (II): Prinzip (I)

- Prinzip: Dynamische Programmierung
  - Optimierungsproblem in Teilprobleme aufteilen
  - Systematische Speicherung von Zwischenresultaten
- Anwendung:
  - Führt kürzester Weg von v_i nach v_j durch v, dann sind auch Teilpfade v_i nach v und v nach v_j minimal

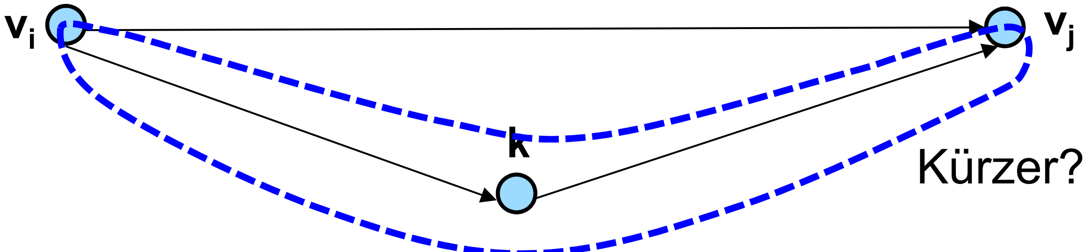


  - Kennt man alle kürzesten Wege zwischen allen Knotenpaaren mit Index kleiner k und sucht kürzeste Wege über alle Knotenpaare mit Index höchstens k, dann gibt es für Pfad von v_i nach v_j zwei Möglichkeiten:
    - Geht über k, dann Zusammensetzung aus bekannten Pfaden v_i nach k und k nach v_j
    - Geht nicht über k, dann ist es der bekannte Pfad von v_i nach v_j

Floyd-Algorithmus (III): Prinzip (II)

- Mit anderen Worten:

  In jeder Iteration wird Distanz von v_i nach v_j genommen und überprüft, ob Weg über v_k eventuell kürzer ist



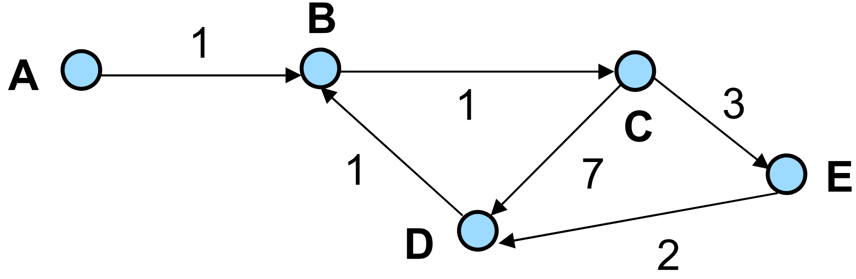
- Folgender Kernalgorithmus:

```
(1) for all k ∈ nodes(G)
(2)   for all i ∈ nodes(G)
(3)     for all j ∈ nodes(G)
(4)           D[i,j] = min(D[i,j],(D[i,k]+D[k,j]));
```

- Komplexität: $O(n^3)$

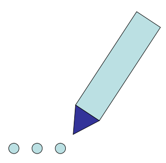
Floyd-Algorithmus (IV): Beispiel

- Finde alle kürzesten Wege in folgendem Graphen:



- Hinweis:
  - Protokollieren Sie die Schritte am besten in einer Tabelle

k	i	j	D[i,j]	D[i,k]+D[k,j]	Minimum	Änderung in D
A	A	A	0	0 + 0 = 0	0	---
A	A	B	1	0 + 1 = 1	1	---
A	A	C	∞	0 + ∞ = ∞	∞	---
…	…	…	…	…	…	…



Übersicht

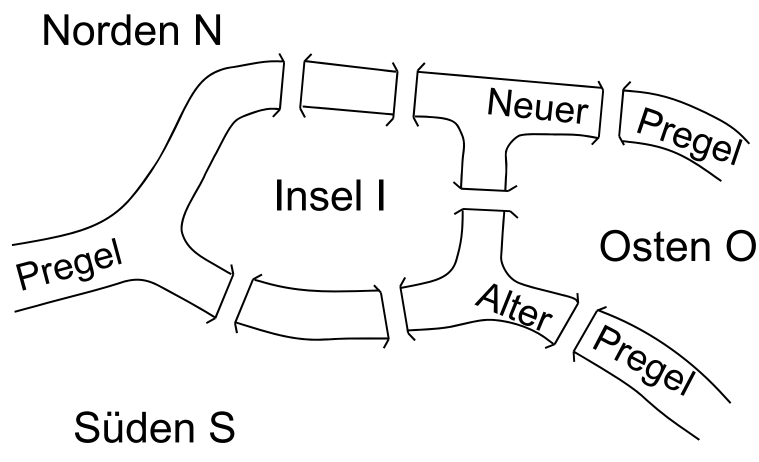
- Färbungen
- Kürzeste Wege
- Euler- und Hamilton-Kreise

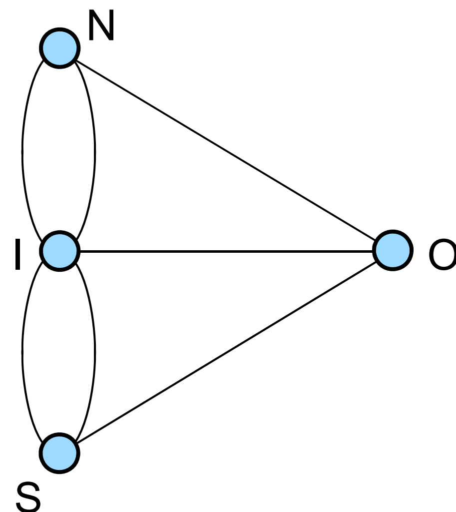
Königsberger Brückenproblem (I)

- Leonhard Euler (1707 - 1782)
- Bedeutender schweizer Mathematiker
- Arbeitete lange in Königsberg
- Stellte sich die Frage, ob es in Königsberg einen Weg gibt, so dass jede Brücke nur genau einmal überquert wird und man zum Ausgangspunkt zurückkehrt?
- Fragestellung hat große historische Bedeutung
- Gilt als Begründung der modernen Graphentheorie

Königsberger Brückenproblem (II)

- Modellierung als Graph:
  - Jeder Stadtteil Knoten
  - Jeder Weg über eine Brücke Kante





- Frage: Gibt es zu jedem Knoten einen Kantenzug, der jede Kante genau einmal enthält und wieder am Ausgangsknoten ankommt?
- Ein solcher Kantenzug wird als Eulerscher Kreis bezeichnet
- Graph mit dieser Eigenschaft heißt eulersch

Test auf Eulerschen Kreis

- Euler fand zwar nicht in Königsberg einen solchen Weg, aber folgenden allgemeinen Satz: Graph G ist genau dann eulersch, wenn der Grad eines jeden Knoten von G gerade ist.
- Gegeben:
  - Ungerichteter Graph G = (V,E)
  - Feld f der Dimension |V|, das die Gradzahlen der Knoten enthält
- Algorithmus:

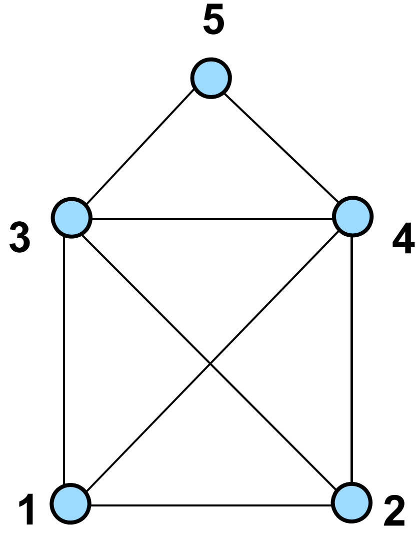
  (1) Setze f[i]=0 für alle i $\in$ {1, ,,,, n}

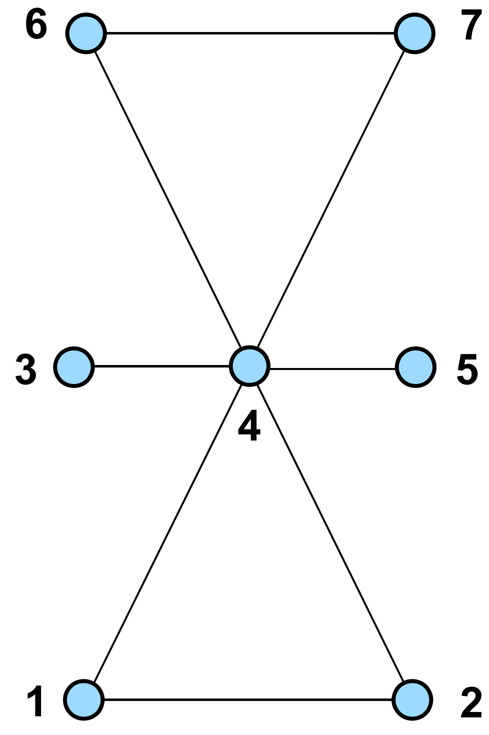
  (2) Gehe alle (v,v') $\in$ E durch:
  - Erhöhe f[v] und f[v'] jeweils um 1

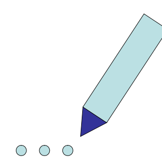
  (3) Wenn f[i] ist gerade für alle i $\in$ {1, ,,,, n} $\Rightarrow$ G eulersch
- Komplexität: O(n+m)
  - In Schritt (1) und (3) sind n Schritte notwendig
  - In Schritt (2) sind m Schritte notwendig

Eulerweg

- Eulerweg (synonym: offener Eulerzug) Kantenfolge, bei der jede Kante genau einmal durchlaufen wird
- Start- und Endknoten nicht identisch
- Graph mit Eulerweg heißt semi-eulersch
- Beispiele:

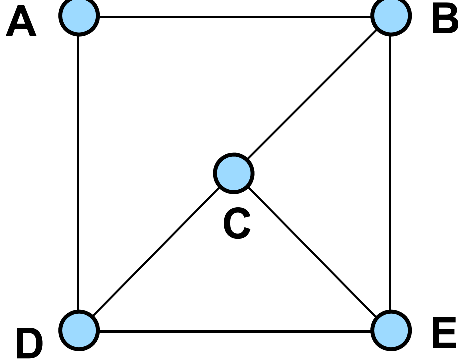






Hamiltonsches Problem (I)

- Sir William Hamilton (1805-1865) formulierte 1859 ein sehr ähnliches Problem
- Gibt es für einen vorgegebenen Graphen einen geschlossenen Weg, der jeden Knoten genau einmal besucht?
- Ein solcher Weg wird Hamiltonscher Kreis genannt
- Beispiel:



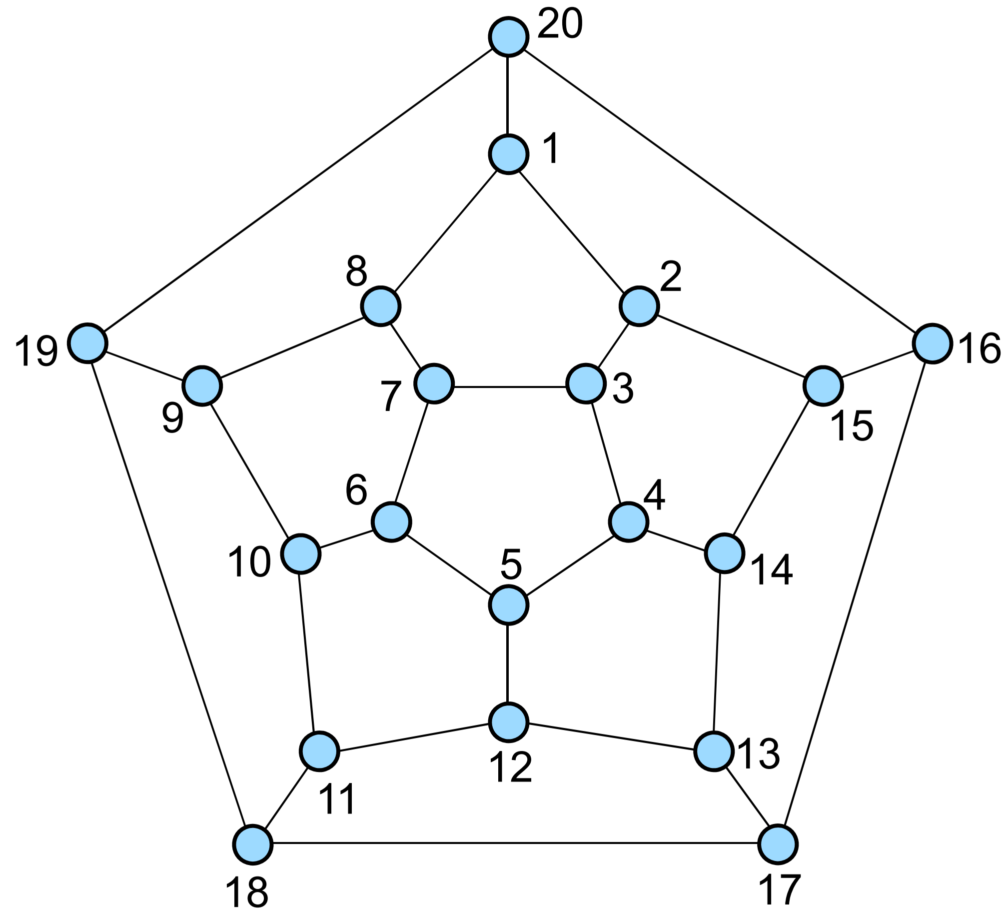
  - (A,B,C,E,D,A) ist ein Hamiltonscher Kreis
- Problem klingt dem Eulerschen Problem recht ähnlich
- Kann daraus die gleiche (gute) Laufzeitkomplexität geschlossen werden?
- Leider nein, d.h. es gibt keine allgemeingültige Aussage, die über einen Graphen und das Enthalten eines Hamiltonschen Kreises etwas sagt

Hamiltonsches Problem (II)

- Algorithmus zum Prüfen auf einen Hamiltonschen Kreis muss also alle Möglichkeiten durchprobieren
- Gegeben: Ungerichteter Graph G
- Algorithmus:
  - Für alle Permutationen von $v_1, ..., v_n \in V$:
    - Wenn $\forall\ i \in \{1, ..., n\} : \{v_i, v_{i+1}\} \in E \Rightarrow$ G hat Hamiltonschen Kreis
  - Trifft die Bedingung für keine Permutation zu $\Rightarrow$ G hat keinen Hamiltonschen Kreis
- Komplexität: O(n!)
  - Es gibt $n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1 = n!$ mögliche Permutationen der Knoten
  - Dies entspricht größenordnungsmäßig $O(n^n)$ (Stirlingsche Formel)
- Fazit: Obwohl Eulerproblem und Hamiltonsches Problem auf den ersten Blick recht ähnlich sind, unterscheiden sie sich in der Laufzeitkomplexität erheblich!

Beispiel Hamiltonscher Kreis

- Der folgende Graph enthält einen Hamiltonschen Kreis, wenn man die Knoten in der Reihenfolge der Nummerierung durchläuft



Zusammenfassung

- Färbungen:
  - Färbung, Chromatische Zahl
  - Backtracking-Algorithmus
  - Problem NP-vollständig
  - Greedy-Algorithmus

- Kürzeste Wege:
  - Algorithmus von Dijkstra
  - Floyd-Algorithmus

- Euler- und Hamilton-Kreise:
  - Euler- und Hamilton-Graphen
  - Ähnliche Probleme, verschiedene Komplexitäten

Übungen (I)

- **Aufgabe 1**

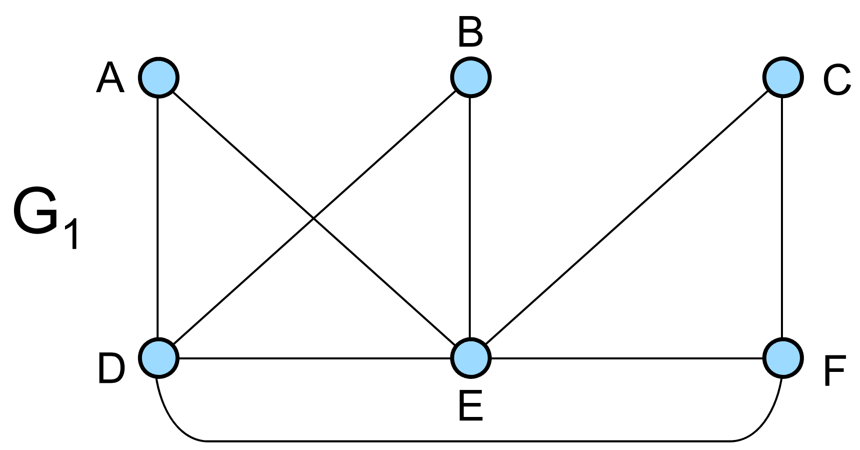
  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

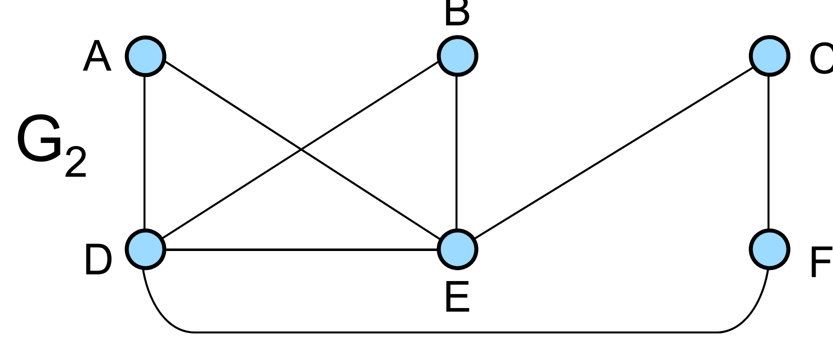
```
C S A N X M G K Y V E G Y J I Q K J R S
D T H D D Q B R N D J K M E Y T F R N I
J Q G Y L M B A Q O U K Z R F A S G S E
C X I N L I R O U B N U U O Z S I A N R
P L T H U T V H R L U N C H H C J Q V K
E I H R S B L J Z N J H J F B D J T D R
I U M K Y B R E T C Y U P B W X T D E E
P Y J L H A Z E H C S I T A M O R H C L
S I B L L R D M A S D M T M I N T C X U
D U H X I D P D O F Z O U B H L A Z Y E
O P O L N R Y V I U E Z Q G F P O J H D
X X K A N G X Y Y Y I L R V B Z Z M Q R
X Y U T A X M J K F D Y A F L O Y D J N
V A W R F P K F L F L N Q M U N Y G L Q
U J H P Q C U M F W Z H P L I L W L R Z
Y M Y A P P Q Y W R X Y S Z R N T B V X
B F H D X N I F F H F I Z W X G I K B A
Y I U Y F U X A S H N F K S D Y U M Y L
E X N F Q G N L O Z X P K Q V N X Q L A
O N U H Z A D A Y U D I H A V L K K N Y
```

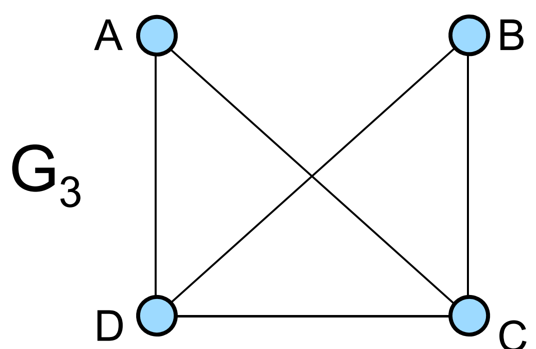
Übungen (II)

- **Aufgabe 2**

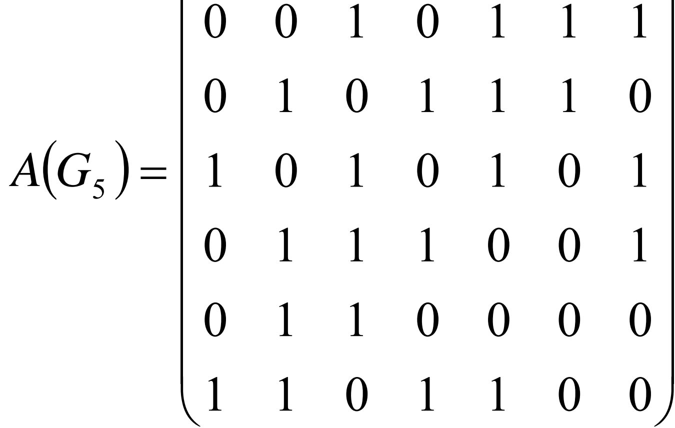
  Geben Sie für die folgenden Graphen an, ob diese einen Eulerschen und/oder einen Hamiltonschen Kreis besitzen! Geben Sie diesen auch jeweils an!







$G_4 = C_6$



$$A(G_5) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Übungen (III)

- **Aufgabe 3**

  Modellieren Sie die Karte der Bundesländer als Graphen und wenden Sie den Greedy-Färbealgorithmus an! Versuchen Sie durch Wahl des Startknotens und Auswahl des nächsten zu färbenden Knotens verschiedene Färbungen zu bekommen!

- **Aufgabe 4**

  Auf den folgenden Graphen ist der Algorithmus von Dijkstra zur Ermittlung des kürzesten Weges von Hamburg (HH) nach Stuttgart (S) angewendet worden. Die fett markierten Kanten sind vom Algorithmus bereits markiert worden.

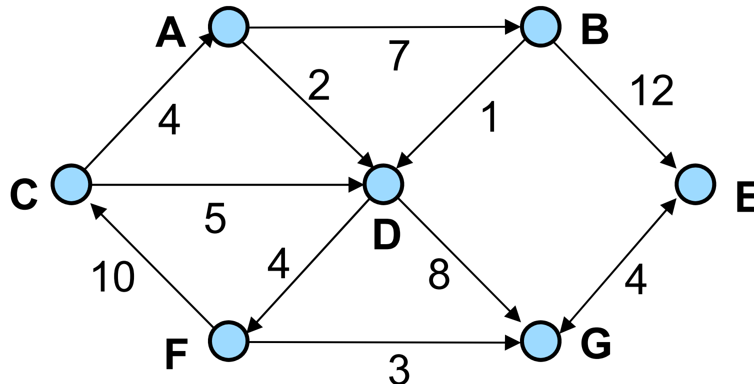
HH
OL
45
50
HB
60
80
SCH
20
H
10
OS
B
280
300
ERF
10
400
KAS
250
K
120
100
150
F
KO
100
10
S

Übungen (V)

a) Geben Sie die Reihenfolge an, in der die Kanten ausgewählt worden sind!
b) Erläutern Sie die Auswahl der nächsten beiden Kanten ausführlich!
c) Geben Sie die Knotenfolge an, die schließlich als Ergebnis des Algorithmus ausgegeben wird!

- **Aufgabe 5**

Gegeben sei der folgende Graph:



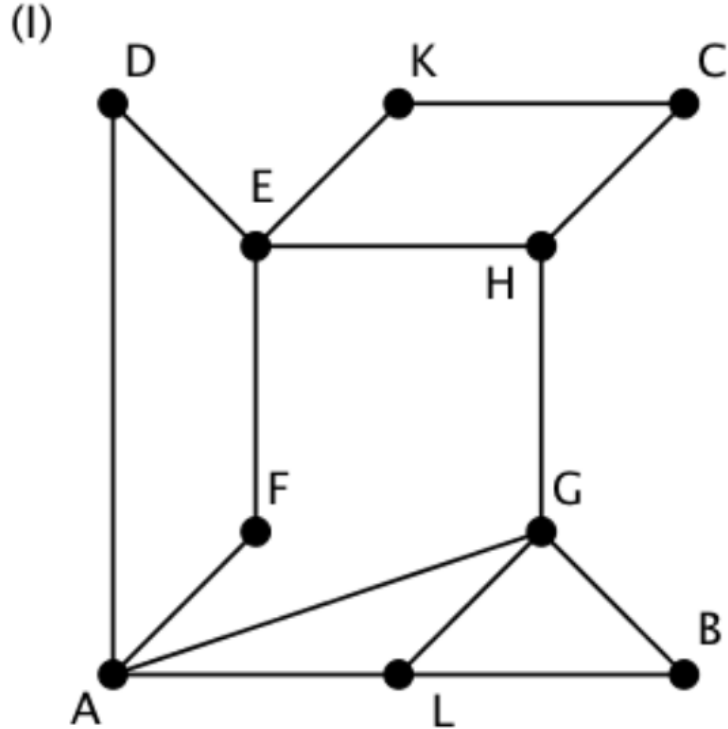
Ermitteln Sie mit dem Floyd-Algorithmus die Distanzmatrix!

Übungen (VI)

- **Aufgabe 6**

  a) Geben Sie für die beiden Beispielgraphen zum Eulerweg jeweils alle möglichen Eulerwege an!

  b) Besitzt der folgende Graph einen Eulerweg?



Algorithmen und Datenstrukturen

Teil 7:
Sortieren (I) –
Einfache Sortierverfahren

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2019

Gliederung

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Motivation

- Datenbestand sortieren häufig vorkommende Anforderung
- Reihe verschiedener Verfahren bekannt
- Einige (einfache) orientieren sich am menschlichen Vorgehen in Alltagssituationen
- Beispiel: Skatspieler ihm zugeteilte Karten auf und sortiert diese auf der Hand:
  - Spieler nimmt eine Karte nach der anderen auf sortiert sie in die bereits aufgenommenen Karten ein (Sortieren durch Einfügen = Insertion Sort)
  - Spieler nimmt die jeweils niedrigste der auf dem Tisch verbliebenen Karten auf und kann sie auf der Hand rechts oder links anfügen (Sortieren durch Auswählen = Selection Sort)
  - Spieler nimmt alle Karten auf und fängt an in der Hand zu sortieren, indem er benachbarte Karten solange tauscht, bis alle Karten in der richtigen Reihenfolge sind (Sortieren durch Vertauschen)

Grundbegriffe (I)

- Sortierproblem: In einem Behälter befindet sich Menge von n Elementen, die als Datensätze bezeichnet werden
- Jeder Datensatz besteht aus einem Sortierschlüssel und „Restdaten"
- Sortierschlüssel: Ausgezeichneter Teil eines Datensatzes, für den Ordnungsrelation R existiert
- Menge von Datensätzen gilt als sortiert, wenn ihre Schlüssel bezgl. R geordnet sind, der (restliche) Inhalt des Datensatzes spielt dabei keine Rolle
- Beispiel: Ausgangsmenge

MatNr.	Name
1007	Meier
1003	Zausel
1001	Tanner
1004	Abel

Grundbegriffe (II)

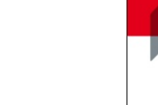
- Aufsteigende Sortierung mit Sortierschlüssel MatNr.

MatNr.	Name
1001	Tanner
1003	Zausel
1004	Abel
1007	Meier

- Absteigende Sortierung mit Sortierschlüssel Name

MatNr.	Name
1003	Zausel
1001	Tanner
1007	Meier
1004	Abel

Klassifikation von Sortierverfahren (I)



- Internes vs. externes Sortieren:
  - Internes Sortieren: Läuft vollständig im Hauptspeicher ab
  - Externes Sortieren: Datenmenge zu groß, um gesamten Sortiervorgang im Hauptspeicher ablaufen zu lassen
- Art des Vorgehens (Sortieren durch ...):
  - ... Austauschen: In der Folge v_1 ... v_n sucht man zwei Elemente v_i und v_j mit $v_i < v_j$ und $i > j$
  - ... Auswählen: In der Folge v_1 ... v_n sucht man ein Elemente v_i mit geeigneter Eigenschaft (z.B. kleinstes oder größtes Element) und bringt dieses an die „richtige“ Stelle
  - ... Einfügen: Die Folge v wird Element für Element in die Folge u umgesetzt; das jeweils nächste Element entfernt man aus v und fügt es in die bereits sortierte Folge u ein
  - ... Mischen: Liegen bereits zwei sortierte Folgen v_1 ... v_n und w_1 ... w_m vor, können diese gemischt werden, indem man die jeweils nächsten Elemente beider Folgen vergleicht und das kleinere übernimmt

Klassifikation von Sortierverfahren (II)

- Art des Vorgehens (Sortieren durch ...) (Fortsetzung):
  - ... Streuen und Sammeln: Besteht der Wertebereich der zu sortierenden Elemente aus einer festen, nicht allzu großen Anzahl, kann ein Feld mit der Länge der Anzahl der Elemente angelegt werden und die zu sortierenden Elemente werden in einem ersten Durchgang in die entsprechenden Feldposition eingetragen (Streuen) und abschließend wird das Feld abgearbeitet (Sammeln)
- Art des Algorithmus: Rekursiv vs. Iterativ
- Laufzeiteffizienz: Quadratische vs. unterquadratische Verfahren
- Speicherbedarf/Datenstruktur:
  - Feld oder Liste?
  - Zusätzliche Datenstrukturen notwendig?
- Einsatzgebiet:
  - Universell verwendbar oder sind bestimmte Voraussetzungen nötig
- Stabiles/Nicht stabiles Verfahren:
  - Wird Reihenfolge von Datensätzen mit gleichem Schlüssel beibehalten?

Übersicht

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Grundidee

- BubbleSort sortiert ein Feld von Datensätzen durch wiederholtes Vertauschen von Nachbarfeldern, die bezgl. der Ordnungsrelation nicht in der richtigen Reihenfolge sind
- Wird solange wiederholt, bis das Feld vollständig sortiert ist
- Feld wird dabei in mehreren Durchgängen von links nach rechts durchlaufen
- In jedem Durchgang werden alle benachbarten Feldeinträge verglichen und ggf. vertauscht
- Nach dem ersten Durchgang steht damit das größte Element ganz rechts
- Nach dem zweiten Durchgang steht das zweitgrößte Element an zweiter Position von rechts
- ...

Beispiel (I)

5	33	12	13	8	1	41

- Durchgang 1:
  - Schritt 1:

5	33	12	13	8	1	41

o.k.

  - Schritt 2:

5	12	33	13	8	1	41

Tausche

  - Schritt 3:

5	12	13	33	8	1	41

Tausche

  - Schritt 4:

5	12	13	8	33	1	41

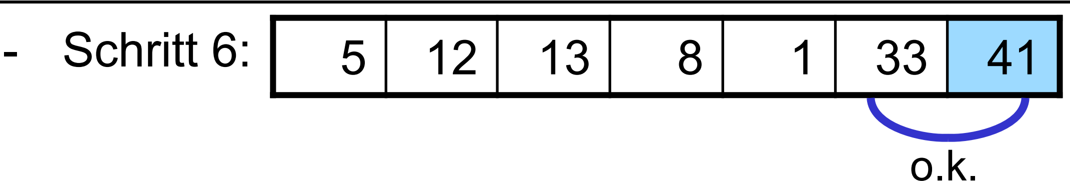
Tausche

  - Schritt 5:

5	12	13	8	1	33	41

Tausche

Beispiel (II)



- Nach Durchgang 2:

5	12	8	1	13	33	41

- Nach Durchgang 3:

5	8	1	12	13	33	41

- Nach Durchgang 4:

5	1	8	12	13	33	41

- Nach Durchgang 5:

1	5	8	12	13	33	41

- Nach Durchgang 6:

1	5	8	12	13	33	41

- Anmerkung zum Namen: Stellt man sich das Feld hochkant vor, steigen die größten Elemente nacheinander nach oben auf wie Blasen in einem Glas

Implementierung

- Sei A zu sortierendes Feld
- Methode `swap(x,y)` vertauscht Elemente x, y im Feld

```
Bubblesort(A)
( 1) for i = 0 to A.length-1{
( 2)   for j = A.length downto i+1{
( 3)     if A[j] < A[j-1]{
( 4)       swap(A[j], A[j-1])
( 5)     }
( 6)   }
( 7) }
```

Analyse und Bewertung (I)

- Algorithmus korrekt: Bei jedem Durchgang wird größtes Element nach rechts durchgeschoben
- Algorithmus terminiert: Sortierter Bereich vergrößert sich um mindestens ein Element
- Zeitaufwand:
  - Unterscheidung Vergleichs- und Vertauschoperation
  - n-1 Durchgänge notwendig, bei jedem Durchgang folgende Schritte (Schritt besteht aus Vergleich und evtl. Vertauschen)
    - Beim ersten Durchgang: n-1 Schritte
    - Beim zweiten Durchgang: n-2 Schritte
    - ...
    - Beim j-ten Durchgang: n-j Schritte
    - ...
    - Beim (n-1)-ten Durchgang: 1 Schritt

Analyse und Bewertung (II)

- Insgesamt: $$\sum_{j=1}^{n}(n-j)\times\left(T_{Vergleichen}+T_{Vertauschen}\right)$$
$$=\sum_{i=1}^{n-1} i\times\left(T_{Vergleichen}+T_{Vertauschen}\right)$$
$$=\frac{n\cdot(n-1)}{2}\times\left(T_{Vergleichen}+T_{Vertauschen}\right)$$
$$\approx c_1\cdot n^2-c_2\cdot n=O\left(n^2\right)$$
- Also: Bubble Sort sowohl von $T_{Vergleichen}$ als auch von $T_{Vertauschen}$ quadratisch abhängig
- Folgende Optimierung möglich:
  - Wenn Durchgang ohne Vertauschen, dann ist Feld sortiert und Algorithmus kann vorzeitig abbrechen
  - D.h. Vorsortierung wirkt sich positiv aus
  - Auswirkung auf Komplexität? (Siehe Übung)

Übersicht

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Grundidee

- Selection Sort sortiert ein Feld von Datensätzen, indem aus dem unsortierten Teil der kleinste Wert ausgewählt wird und als nächstes Element der Zielfolge angefügt wird
- Beispiel:

5	33	12	13	8	1	41

- Durchgang 1
  - Schritt 1:

5	33	12	13	8	1	41

Kleinster

  - Schritt 2:

1	12	33	13	8	5	41

Tausche

- Durchgang 2
  - Schritt 1:

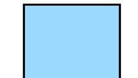
1	12	33	13	8	5	41

Kleinster

  - Schritt 2:

1	5	33	13	8	12	41

Tausche



Sortierter Teil des Feldes

Beispiel (Fortsetzung)

- Durchgang 3:

1	5	8	13	33	12	41

- Durchgang 4:

1	5	8	12	33	13	41

- Durchgang 5:

1	5	8	12	13	33	41

- Durchgang 6:

1	5	8	12	13	33	41

Implementierung

- Sei `A[0 … n-1]` zu sortierendes Feld
- Methode `swap(x,y)` vertauscht Elemente x, y im Feld

```
Selection Sort(A)
( 1) for i = 0 to A.length-2{
( 2)   min = i
( 3)   for j = i+1 to A.length-1{
( 4)     if A[j] < A[min]{
( 5)       min = j
( 6)     }
( 7)   }
( 8)   swap(A[i], A[min])
( 9) }
```

Analyse und Bewertung (I)

- Algorithmus korrekt: Bei jedem Durchgang wird kleinstes unsortiertes Element ausgewählt und an sortierten Teil angehängt
- Algorithmus terminiert; Sortierter Bereich wird bei jedem Durchgang um ein Element vergrößert
- Zeitaufwand:
  - Unterscheidung zwischen Vergleichs- und Vertauschoperation notwendig
  - n-1 Durchgänge sind notwendig, bei jedem Durchgang folgende Schritte (Schritt besteht aus Vergleich und evtl. Vertauschen)
  - Beim ersten Durchgang: n-1 Vergleiche
  - Beim zweiten Durchgang: n-2 Vergleiche
  - ...
  - Beim j-ten Durchgang: n-j Vergleiche
  - ...
  - Beim (n-1)-ten Durchgang: 1 Vergleiche
  - In jedem Durchgang ein Vertauschen notwendig

Analyse und Bewertung (II)

- Insgesamt:

$$\sum_{j=1}^{n}(n-j)\times T_{Vergleichen} + (n-1)\times T_{Vertauschen}$$

$$= T_{Vergleichen} \times \frac{n\cdot(n-1)}{2} + T_{Vertauschen} \times (n-1)$$

$$\approx c_1 \cdot n^2 + c_2 \cdot n = O(n^2)$$

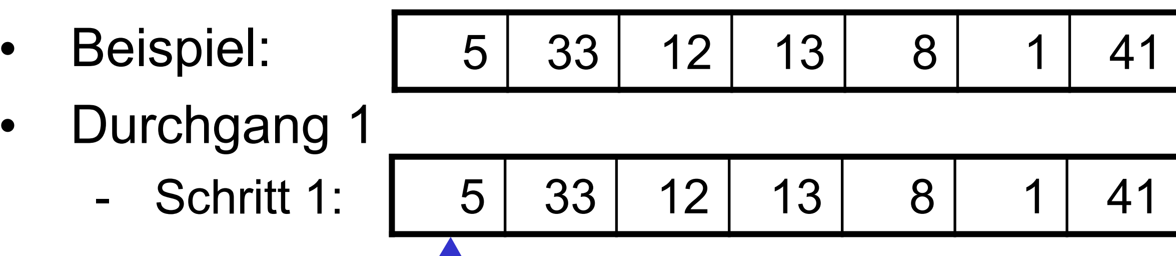
- Aufwand linear abhängig von $T_{Vertauschen}$, quadratisch von $T_{Vergleichen}$
- Vergleichen im Vergleich zum Vertauschen effizienter (d.h. $T_{Vergleichen} < T_{Vertauschen}$)
- Dennoch: Ab gewissem n dominiert quadratischer Anteil
- Vergleich mit BubbleSort:
  - Beide Verfahren quadratischer Aufwand
  - SelectionSort weniger Vertauschoperationen
  - Kein vorzeitiger Abbruch möglich, d.h. Vorsortierung unerheblich

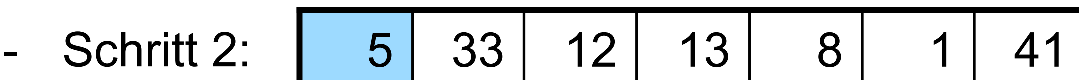
Übersicht

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

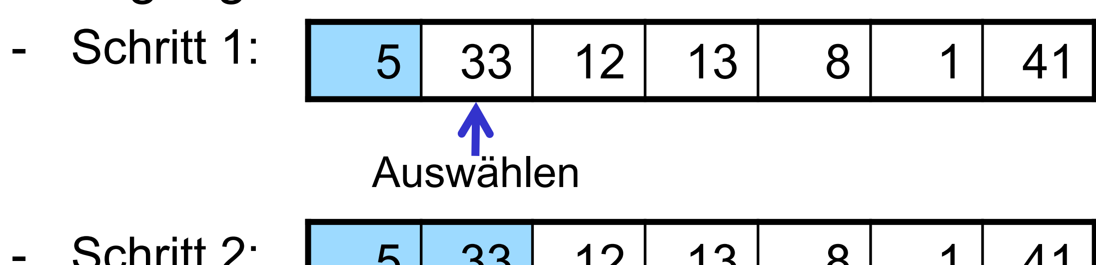
Grundidee

- Insertion Sort sortiert ein Feld von Datensätzen, indem aus dem unsortierten Teil ein beliebiger Wert ausgewählt wird (meistens der erste) und in die bereits bestehende Zielfolge an der richtigen Stelle eingefügt wird
- Beispiel:
- Durchgang 1
  - Schritt 1:
  - Schritt 2:





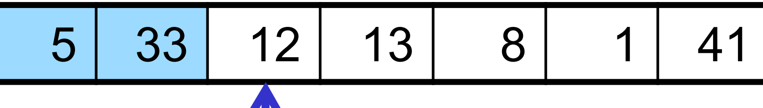
- Durchgang 2
  - Schritt 1:
  - Schritt 2:



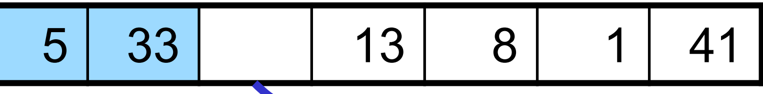
Beispiel (Fortsetzung)

-

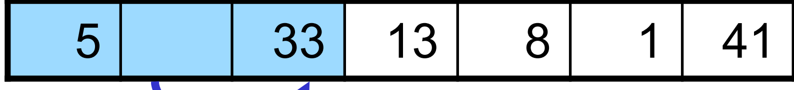
  -



  -



  - Schritt 3:



Einfügen

  - Schritt 4:

- Durchgang 4:

5	12	13	33	8	1	41

- Durchgang 5:

- Durchgang 6:

Implementierung

- Sei `A[1 … n]` zu sortierendes Feld

```
InsertionSort(A)
( 1) for j = 2 to A.length{
( 2)   key = A[j]
( 3)   // Fuege A[j] in sortierte Sequenz A[1..j-1] ein
( 4)   i = j-1
( 5)   while i > 0 AND A[i] > key{
( 6)     A[i+1] = A[i]
( 7)     i = i-1
( 8)   }
( 9)   A[i+1] = key
(10) }
```

Analyse und Bewertung (I)

- Algorithmus korrekt: In jedem Durchgang wird ausgewähltes unsortiertes Element an richtiger Stelle in sortierten Teil eingefügt
- Algorithmus terminiert: Sortierter Bereich wird in jedem Durchgang um ein Element vergrößert
- Zeitaufwand:
  - Unterscheidung zwischen Vergleichs- und Vertauschoperation notwendig
  - n-1 Durchgänge sind notwendig, bei jedem Durchgang folgende Schritte (Schritt besteht aus Merken (Zuweisung an Zwischenspeicher), Vergleich und evtl. Vertauschen)
  - Beim ersten Durchgang: 1 Vergleich und Vertauschung
  - Beim zweiten Durchgang: 2 Vergleiche und Vertauschungen
  - ...
  - Beim j-ten Durchgang: j Vergleiche und Vertauschungen
  - ...
  - Beim (n-1)-ten Durchgang: n-1 Vergleiche und Vertauschungen
  - In jedem Durchgang zwei Zuweisungen notwendig (Zum Merken des aktuellen Elementes im Zwischenspeicher)
- Anm.: „Eigentlich“ n Durchgänge notwendig, im 0-ten aber kein Aufwand, da erstes Element automatisch sortiert ist

Analyse und Bewertung (II)

- Insgesamt:

$$\sum_{j=1}^{n-1} j \cdot \left(T_{Vergleichen} + T_{Vertauschen}\right) + 2 \cdot (n-1) \cdot T_{Zuweisen}$$

$$= \left(T_{Vergleichen} + T_{Vertauschen}\right) \cdot \frac{n \cdot (n-1)}{2} + 2 \cdot (n-1) \cdot T_{Zuweisen}$$

$$\approx c_1 \cdot n^2 + c_2 \cdot n = O\left(n^2\right)$$

- Auch Insertion Sort quadratischer Aufwand
- Bei Vorsortierung: Best case-Aufwand linear (bei einem schon sortierten Feld keine weiteren Durchgänge mehr notwendig)

Übersicht

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Vergleich der einfachen Sortieralgorithmen

- Alle drei Verfahren haben gleiche Laufzeitkomplexität $O(n^2)$
- Unterschiede:
  - Insertion Sort braucht im Schnitt weniger Vergleiche, weil beim Auffinden der richtigen Stelle im Mittel nur die Hälfte des Feldes zu durchsuchen ist
  - Bei Selection Sort müssen immer alle Vergleiche durchgeführt werden, um die richtige Position zu finden
  - Verschieben beim Insertion Sort erfordert mehr Vertauschungen als bei Selection Sort in jedem einzelnen Durchgang
  - Insertion Sort und Bubble Sort nutzen im Gegensatz zu Selection Sort eine Vorsortierung des Feldes aus
- Fazit: Laufzeitverhalten aller drei Verfahren relativ ähnlich

Übersicht

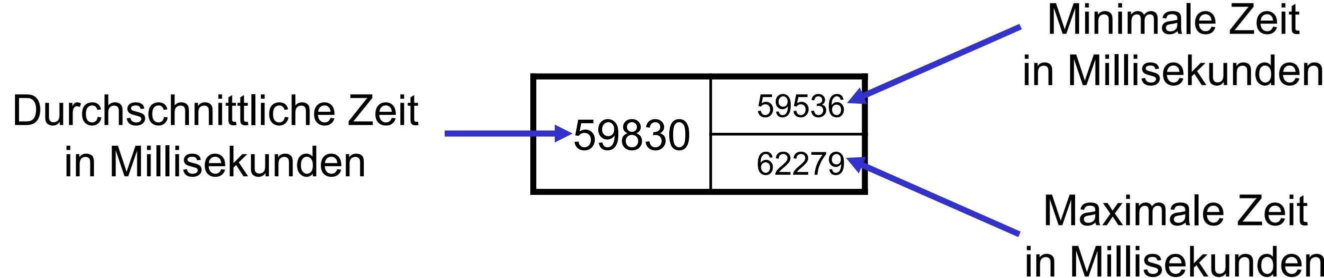
- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Implementierung (I)

- Feld mit Integer-Werten, anfangs zufällig belegt
- Feldgröße von 10.000, 20.000, ..., 100.000
- Laufzeitmessung für folgende Algorithmen:
  - Bubble Sort einfache Implementierung
  - Bubble Sort mit Wächter: Stoppt Bearbeitung, wenn keine Vertauschungen mehr durchgeführt worden sind
  - Selection Sort
  - Insertion Sort einfache Implementierung
  - Insertion Sort mit Wächter: Stoppt Bearbeitung, wenn keine Vertauschungen mehr durchgeführt worden sind

Implementierung (II)

- Durchführung:
  - Sortierung je 100-mal durchgeführt
- Plattform:
  - Java-Implementierung
  - Messung 1: Laptop, Intel 2.2 GHz, 256 MB RAM (2005)
  - Messung 2: Laptop, Intel Core i7 2.8 GHz, 32 GB RAM (2019)
- Ergebnisdarstellung:



Ergebnisse Messung 1

Daten-sätze	BubbleSort		BubbleSort mit Wächter		Selection Sort		Insertion Sort		Insertion Sort mit Wächter	
10.000	596	581	595	581	238	230	338	330	237	230
		661		661		301		401		310
20.000	2311	2293	2372	2344	944	931	1348	1331	948	931
		2363		2444		1011		1412		1002
30.000	5201	5177	5354	5307	2107	2083	3033	3004	2145	2113
		5318		5928		2194		3105		2204
40.000	9259	9213	9490	9454	3758	3735	5391	5358	3798	3746
		9373		9564		3825		5468		3915
50.000	14484	14401	14837	14771	5868	5829	8441	8402	5934	5888
		15683		14911		5978		8483		6008
75.000	32591	32516	34127	34019	13205	13169	19051	18957	13312	13249
		33418		36092		13309		20419		13399
100.000	58236	58124	59830	59536	23691	23513	34011	33928	23806	23664
		58434		62279		25097		34149		23995

Ergebnisse Messung 2

Datensätze	BubbleSort		BubbleSort mit Wächter		Selection Sort		Insertion Sort		Insertion Sort mit Wächter	
10.000	123	109	127	106	28	15	44	31	34	31
		172		200		47		63		63
20.000	547	515	541	500	102	93	178	156	138	124
		766		733		203		265		188
30.000	1312	1218	1148	1109	229	203	400	374	296	265
		1654		1390		375		546		359
40.000	2321	2234	2076	2015	403	375	698	671	521	453
		3093		2499		703		1016		593
50.000	3780	3546	3243	3171	624	593	1060	1015	811	719
		5086		3953		1047		1687		860
75.000	8652	8123	7359	7185	1393	1328	2451	2390	1820	1609
		11138		9092		2375		3390		1969
100.000	15352	14543	13029	12840	2464	2374	4260	4124	3231	2828
		19095		16652		4202		5920		3421

Ergebnisse grafisch

Zeit

Bubble Sort mit Wächter

Bubble Sort

Insertion Sort

Insertion Sort mit Wächter

Selection Sort

Anzahl Datensätze

Interpretation

- Alle Verfahren zeigen quadratisches Verhalten
- Dennoch unterscheiden sie sich in den absoluten doch erheblich (zwar alle quadratisch, aber unterschiedliche Konstante)
- Selection Sort ist das schnellste Verfahren (Grund: quadratisch nur in den relativ billigen Vergleichen, aber in den relativ teuren Vertauschungen linear)
- Bei Bubble Sort bringt die „Optimierung“ durch den Wächter nichts ein, im Gegenteil Laufzeit verschlechtert sich sogar noch etwas (Verwaltung des Wächters)
- Bei Insertion Sort hingegen verbessert sich die Laufzeit durch die Einführung des Wächters
- Unterschiedliche Hardware: Absolute Laufzeit verbessert sich, aber qualitativer Verlauf bleibt

Zusammenfassung

- Motivation und Grundbegriffe
- Bubble Sort
- Selection Sort
- Insertion Sort
- Vergleich
- Implementierung/Praktische Untersuchung

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
R F C S B Q J V Q Z F V J T H S N I P T
S M I Q E B S Y D Q Y X S U G E Y N B T
S T A B I L E S S O R T I E R E N S H E
L O I Z R G E O P B T F A E I S D E C U
M S A M G Y Z C U U Y P I B S A Y R T Q
K P M O W X R B T Z N T D R G M L T J L
A R P J S O B I C I R Z I J G A E I H D
M L O M E L P X F O O C X Y L P Z O D Y
F V U S E Y Y R S P V N W M R P X N S I
U Y X S A B T S C X M M S E C O F S C Z
D D O W R P E L W X N E Q O D I M O S W
I R A U F N B T Z S S S Q K R X N R D E
T U L C R K D N Q Q B B E U R T J T E K
C O L E H T Z I H V N O L E H Y F E S T
C C T O U N N N I U G X V P E N E J Q X
Y X I X V G L X Z J S K V C W A Y C D X
E K J I N O L C O T G Q G W Q M W N A N
Q Y E J M R M X L C I F H R Q M B R V O
H F Z G C I D K W O C W W P C V J E D J
X L H J Q E J N J S T Q K T Q I T Q H S
```

Übungen (II)

- **Aufgabe 2**

  Gegeben sei das folgende Feld mit Vornamen, das gemäß lexikographischer Ordnung absteigend sortiert werden soll!

Steffi	Michael	Stefanie	Astrid	Stefan	Stephanie	Stephan	Xaver

  Führen Sie diesen Sortiervorgang mithilfe der Verfahren Bubble Sort, Selection Sort und Insertion Sort durch. Geben Sie dabei das Zwischenresultat eines jeden Durchgangs an, für die beiden ersten Durchgänge geben Sie bitte auch jeweils die einzelnen Schritte an!

- **Aufgabe 3**

  Füllen Sie folgende Tabelle aus!

	Best Case	Average Case	Worst Case
Bubble Sort			
Insertion Sort			
Selection Sort			

Übungen (III)

- **Aufgabe 4**

	Wahr	Falsch
Bubble Sort nutzt Vorsortierung nicht aus.		
Externe Sortierverfahren sind zwar aufwändiger zu implementieren, aber auf jeden Fall effizienter in der Laufzeit.		
Selection Sort arbeitet stets schneller als Bubble Sort.		
Sortierverfahren sind immer rekursiv.		
Insertion Sort weist im besten Fall lineare Laufzeit auf.		
Algorithmen zum Sortieren von Zahlen können nicht auf das Sortieren von Zeichenketten übertragen werden.		
Interne Sortierverfahren verdanken ihren Namen der Tatsache, dass sie mit dem vorhandenem Speicherplatz auskommen.		
Bubble Sort schiebt bei jedem Durchgang das größte noch nicht sortierte Element nach rechts.		
Mit Bubble Sort kann man nur aufsteigend sortieren.		

Algorithmen und Datenstrukturen

Teil 8:
Sortieren (II) –
Schnelle Sortierverfahren

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 05/2020

Gliederung

- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren

Algorithmen mit Vorsortierung

- Bei einfachen Sortieralgorithmen: Vorsortierung u.U. positiv
- Daher: Idee der Vorsortierung von Teilen des Feldes
  - Zerlege Feld in Menge von Sequenzen
  - Sortiere diese Sequenzen (mit einem der einfachen Sortierverfahren)
  - Dann sinkender Gesamtaufwand
- Aufbauen auf Verfahren, das Vorsortierung nutzt
- Daher zwei Verfahren:
  - Shell Sort basiert auf Insertion Sort
  - Comb Sort basiert auf Bubble Sort

Shell Sort

- Variante von Insertion Sort, von Donald Shell 1959 vorgeschlagen
- Algorithmus wird um einen Parameter h erweitert, der die aktuelle Anzahl von Sequenzen angibt
- Für h=1 (Terminierungsbedingung) ergibt sich dabei die Originalversion des Algorithmus
- „Kunst“ besteht nun darin, die „richtige“ Folge von h‘s zu finden:
  - Vorschlag von Shell: $h_1=1$, $h_i = 3 \cdot h_{i-1}+1$
  - Also: h = ..., 1093, 364, 121, 40, 13, 4, 1
  - Andere Vorschläge aus der Literatur:

    h = ..., 16001, 3905, 2161, 929, 505, 209, 109, 41, 19, 5, 1 (M.A. Weiss)

    $h_i = 2^i-1$
- Ablauf von Shell Sort:

  (1) Bestimme den Anfangswert von h

  (2) Sortiere jede der h Sequenzen unter Anwendung von Insertion Sort

  (3) Bestimme den nächsten Wert von h

  (4) Wenn h=1, dann beende, sonst gehe zu Schritt (2)

Beispiel (I)

-

5	33	12	13	8	1	41	77	2	18	9	38

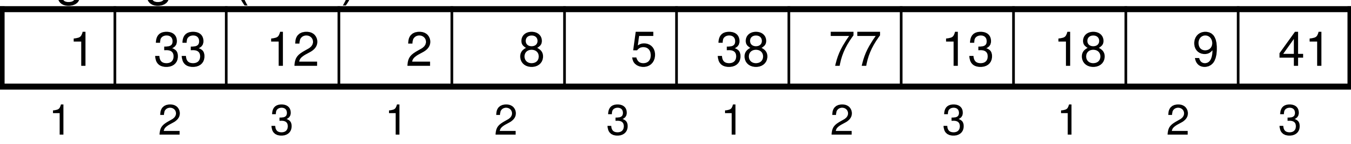
- Folge der h's sei $h_3 = 5$, $h_2 = 3$, $h_1 = 1$
- Durchgang 1:

5	33	12	13	8	1	41	77	2	18	9	38
1	2	3	4	5	1	2	3	4	5	1	2

  - Gruppe 1: 5, 1, 9
  - Gruppe 2: 33, 41, 38
  - Gruppe 3: 12, 77
  - Gruppe 4: 13, 2
  - Gruppe 5: 8, 18
  - Anwenden von Insertion Sort auf jede Gruppe ergibt:

1	33	12	2	8	5	38	77	13	18	9	41

- Durchgang 2 (h=3):

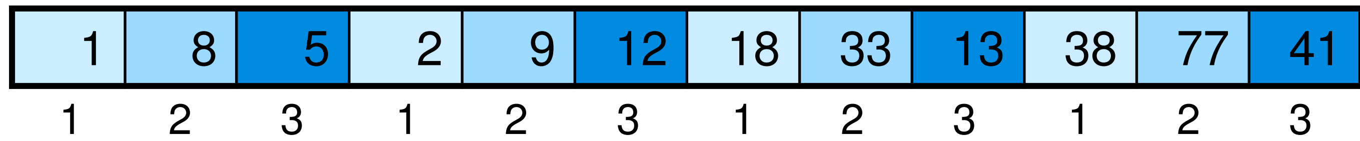


1	33	12	2	8	5	38	77	13	18	9	41
1	2	3	1	2	3	1	2	3	1	2	3

  - Gruppe 1: 1, 2, 38, 18
  - Gruppe 2: 33, 8, 77, 9
  - Gruppe 3: 12, 5, 13, 41

Beispiel (II)

- Anwenden von Insertion Sort auf jede Gruppe ergibt:



1	8	5	2	9	12	18	33	13	38	77	41
1	2	3	1	2	3	1	2	3	1	2	3

- Durchgang 3 (h=1): Insertion Sort führt zum Resultat:

1	2	5	8	9	12	13	18	33	38	41	77

Analyse und Bewertung

- Korrektheit von Shell Sort lässt sich auf Korrektheit von Insertion Sort zurückführen
- Terminierung ist durch fallende h‘s mit letztem h=1 gegeben
- Laufzeitkomplexität:
  - Theoretisch sehr schwer zu beweisen
  - Auch im ungünstigsten Falle ist Shell Sort besser als Insertion Sort ($O(n^{1,5})$ gegenüber $O(n^2)$)
  - Experimente ergeben Werte von ca. $O(n^{1,25})$
  - Wird $h_i = 2^i - 1$ gewählt, so kann für Shell Sort $O(n \cdot \sqrt{n})$ nachgewiesen werden
  - Werden die sortierten Elemente in einen Baum eingefügt und dieser in Inorder-Reihenfolge durchlaufen, so kann die Laufzeit gar auf $O(n \cdot \log_2 n)$ gedrückt werden
  - Natürlich kommt hier der Speicherbedarf für die Baumstruktur hinzu

Comb Sort

- Analog zum Basieren von Shell Sort auf Insertion Sort lässt sich basierend auf Bubble Sort ein Algorithmus unter Ausnutzung der Vorsortierung konzipieren
- Dieser wird als Comb Sort bezeichnet
- Typischerweise werden die Sequenzen „dichter" gewählt, z.B.

  $$h_n = \frac{n}{SF}, h_{n-1} = \frac{h_n}{SF}, \ldots, 0$$

  wobei SF ein Schrumpf-Faktor von 1.3 ist.
- Beispielfolge: ..., 11, 8, 6, 4, 3, 2, 1

Übersicht

- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren
- Sortieren durch Streuen und Sammeln

Grundidee (I)

- Heap Sort:
  - Zweiphasiges, baumbasiertes Verfahren
  - Phase 1: Aufbau des Heap
  - Phase 2: Ausgabe des Heap in sortierter Reihenfolge
- Phase 1:
  - Zu sortierende Folge wird schichtweise in binären Baum eingetragen
  - Knoteninhalte werden anschließend so vertauscht, dass sich das größte Element eines jeden Teilbaums in der Wurzel befindet
  - Größtes Element der Folge steht damit in der Wurzel des Baums
  - Baum mit diesen Eigenschaften heißt Heap
  - Genauer: MAX-HEAP
  - Definition auch „umgekehrt“ möglich (MIN-HEAP)
- Herstellung der Heap-Eigenschaft wird erreicht, indem ausgehend von den Blättern jeder Knoteninhalt, der kleiner als mindestens einer seiner direkten Nachfolger ist, durch Vertauschen nach unten „absinkt“

Grundidee (II)

- Phase 2: Ausgabe des Heaps:
  - In der Wurzel steht das größte Element
  - Dieses wird ausgegeben und an seine Stelle das am weitesten rechts in der untersten Schicht stehende Element gesetzt
  - Dann stellt man durch Absinken der Wurzel erneut die Heap-Eigenschaft her
  - Dieses wiederholt man bis der Baum leer ist

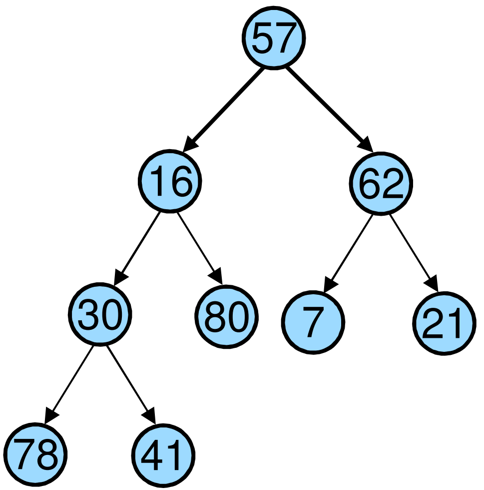
Beispiel (I)

- Phase 1:
  - Feld

57	16	62	30	80	7	21	78	41

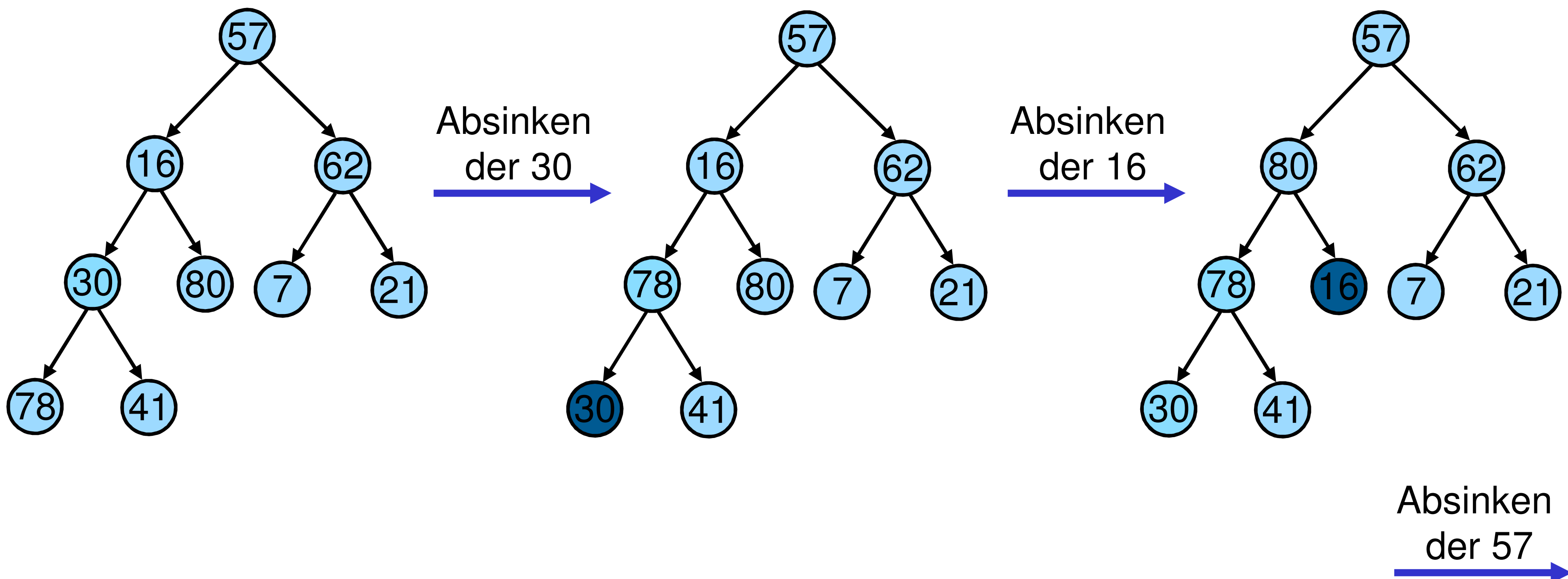
ergibt

Binärbaum

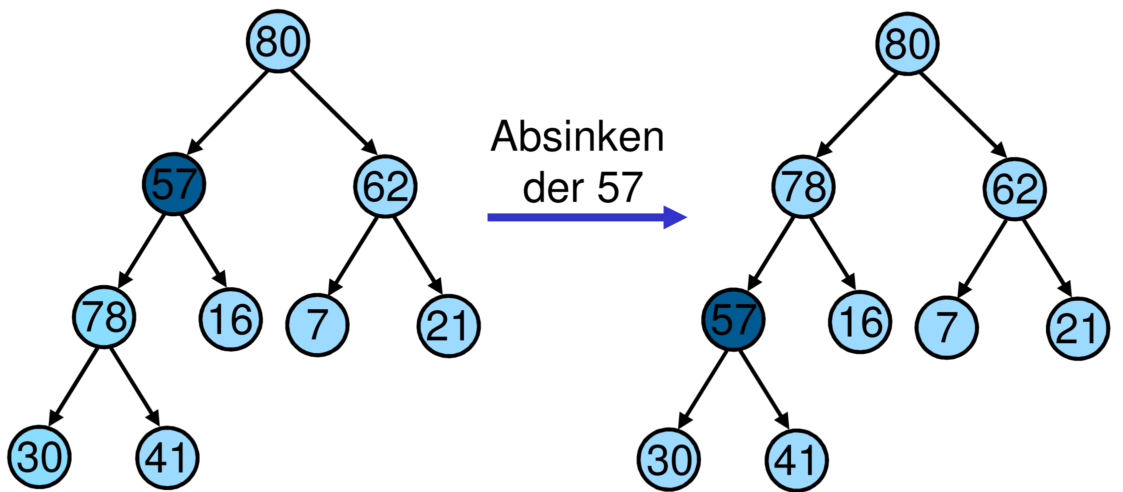


Beispiel (II)

- Absinken kann über mehrere Schichten erfolgen



Beispiel (III)



- Nun erfüllt der Baum die Heap-Eigenschaft
- Phase 2 kann beginnen

Beispiel (IV)

80
78 62
57 16 7 21
30 41

80 ausgeben
41 neue Wurzel

41
78 62
57 16 7 21
30

80

Absinken
der 41

78
57 62
41 16 7 21
30

80

78 ausgeben
30 neue Wurzel

30
57 62
41 16 7 21

80 78

Absinken
der 30

Beispiel (V)

62
57 30
41 16 7 21

80	78

62 ausgeben
21 neue Wurzel

21
57 30
41 16 7

80	78	62

Absinken
der 21

57
41 30
21 16 7

80	78	62

57 ausgeben
7 neue Wurzel

7
41 30
21 16

80	78	62	57

Absinken
der 7

Beispiel (VI)

(Baum: 41; Kinder 21, 30; Kinder von 21: 7, 16)

80	78	62	57

41 ausgeben
16 neue Wurzel →

(Baum: 16; Kinder 21, 30; Kind von 21: 7)

80	78	62	57	41

Absinken der 16 →

(Baum: 30; Kinder 21, 16; Kind von 21: 7)

80	78	62	57	41

30 ausgeben
7 neue Wurzel →

(Baum: 7; Kinder 21, 16)

80	78	62	57	41	30

Absinken der 7 →

Beispiel (VII)

Baum: 21 (Kinder: 7, 16)

80	78	62	57	41	30

21 ausgeben
16 neue Wurzel

Baum: 16 (Kind: 7)

80	78	62	57	41	30	21

Kein Absinken nötig
16 ausgeben
7 neue Wurzel

Baum: 7

80	78	62	57	41	30	21	16

7 ausgeben
Baum leer
$\Rightarrow$ Fertig

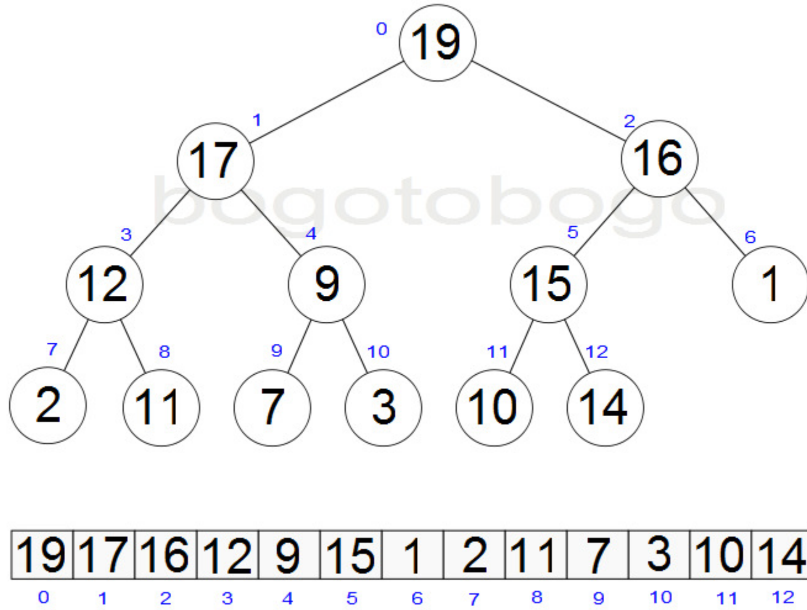
80	78	62	57	41	30	21	16	7

Analyse und Bewertung

- Korrektheit folgt aufgrund der Heap-Eigenschaft (es wird immer das größte Element vom Heap entnommen)
- Algorithmus terminiert, denn
  - Aufbau erfolgt in endlich vielen Schritten
  - Beim Abbau wird immer ein Knoten entfernt und der leere Baum ist das Terminierungskriterium
- Laufzeitkomplexität:
  - Aufbau des Heaps erfolgt in O(n) Schritten
  - Absinken von der Wurzel benötigt höchstens $O(\log_2 n)$ Schritte, denn ein ausgeglichener Binärbaum hat logarithmische Höhe
  - Insgesamt: $O(n \cdot \log_2 n)$

Implementierung (I)

- Trage Knoten schichtweise in Feld A ein



- Funktionen zum Navigieren:
  - PARENT(i): Liefert Elternknoten, Wert ist (i - 1) DIV 2
  - LEFT(i): Liefert linken Nachfolger, Wert ist 2i + 1
  - RIGHT(i): Liefert rechten Nachfolger, Wert 2i + 2
- Heap-Eigenschaft: Für alle i außer Wurzel: A[PARENT(i)] >= A[i]

Implementierung (II)

- Methode MAX-HEAPIFY(A,i):
  - A zu sortierendes Feld, i Index
  - Voraussetzung bei Aufruf:
    - Binärbäume mit Wurzeln LEFT(i) und RIGHT(i) Heaps
    - A[i] kann kleiner als seine Nachfolger sein (Heap-Eigenschaft verletzt)
  - Methode lässt Wert von A[i] absinken
  - Anschließend: Teilbaum mit Wurzel A[i] besitzt Heap-Eigenschaft

```
MAX-HEAPIFY(A,i)
( 1) l = LEFT(i)
( 2) r = RIGHT(i)
( 3) if l<=A.heap-size && A[l] > A[i]
( 4)   largest = l
( 5) else largest = i
( 6) if r<=A.heap-size && A[r] > A[largest]
( 7)   largest = r
( 8) if largest<>i{
( 9)   swap(A[i], A[largest])
(10)   MAX-HEAPIFY(A, largest)
(11) }
```

Implementierung (III)

- Erstmalige Herstellung Heap-Eigenschaft unter Verwendung von MAX-HEAPIFY:

```
BUILD-MAX-HEAP(A)
( 1) A.heapSize = A.length
( 2) for i = floor(A.length/2) downto 1{
( 3)   MAX-HEAPIFY(A, i)
( 4) }
```

- Damit Algorithmus:

```
HEAPSORT(A)
( 1) BUILD-MAX-HEAP(A)
( 2) for i = A.length downto 2{
( 3)   swap(A[1], A[i])
( 4)   A.heapSize--
( 5)   MAX-HEAPIFY(A, 1)
( 6) }
```

Übersicht

- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren

Grundidee

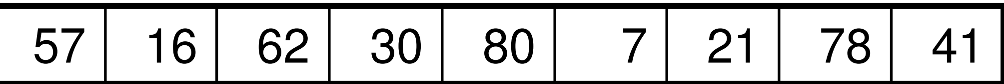
- Quicksort wurde 1962 von Hoare vorgeschlagen
- Zwei Basisideen:
  - Rekursion
  - Divide and Conquer-Prinzip (Teile und (be)herrsche)
- Lösungsidee:
  - Wähle ein beliebiges Schlüsselelement p (Pivotelement) aus der zu sortierenden Menge heraus
  - Zerlege die Restmenge in zwei Teilmengen KL (Menge der Elemente kleiner als p) und GR (Menge der Elemente größer als p):

KL: Menge der Elemente kleiner als p (unsortiert)	p	GR: Menge der Elemente größer als p (unsortiert)

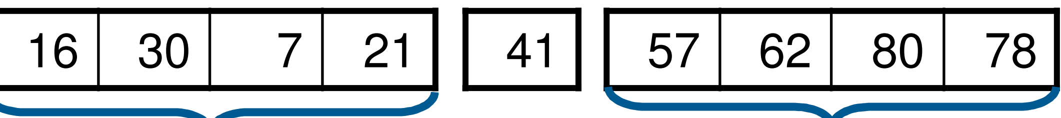
  - Sortiere KL und GR jeweils rekursiv mit Quicksort und füge die sortierten Teile wieder zusammen:

KL: Menge der Elemente kleiner als p (sortiert)	p	GR: Menge der Elemente größer als p (sortiert)

Beispiel

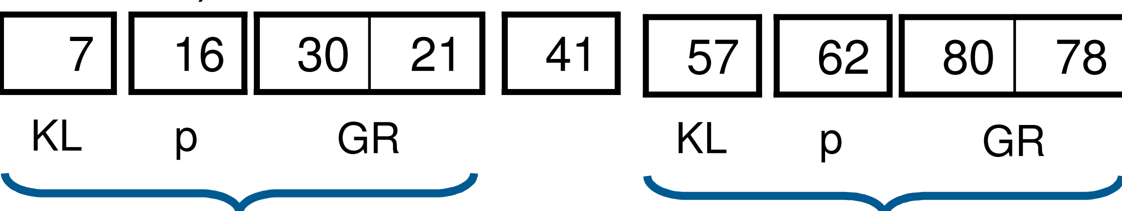


- Wähle Pivotelement p (z.B. p=41)
- Teile auf:



Menge KL (unsortiert) p Menge GR (unsortiert)

- Rufe für beide Mengen rekursiv Quicksort auf (Pivotelemente seien 16 bzw. 62):



Menge KL (unsortiert) p Menge GR (unsortiert)

- Schließlich landet man bei einelementigen Feldern (die naturgemäß sortiert sind) und die Rekursion baut sich ab:

7	16	21	30	41	57	62	78	80

Wahl des Pivotelements

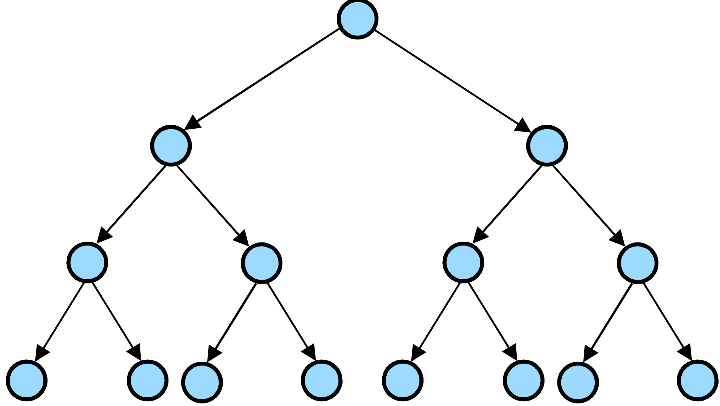
- Wahl von p ist für das Funktionieren von Quick Sort unerheblich
- Quick Sort ist aber besonders schnell, wenn es gelingt p immer so zu wählen, dass die Mengen KL und GR bei jeder Zerlegung etwa gleich groß sind
- Strategien zur Wahl von p:
  - Nehme immer das erste Element im Feld
  - Nehme immer das letzte Element im Feld
  - Nehme immer das mittlere Element im Feld
  - Nehme den Median der drei zuvor genannten Elemente
  - Ist über die Daten nichts bekannt, sind die ersten drei Strategien gleichwertig
  - Sind die Daten vorsortiert, so ist die Wahl des mittleren Elements vorzuziehen
  - Bei der Wahl des Median steigt die Chance ein besseres Pivotelement finden, allerdings steigt der Aufwand zur Ermittlung

Bewertung (I)

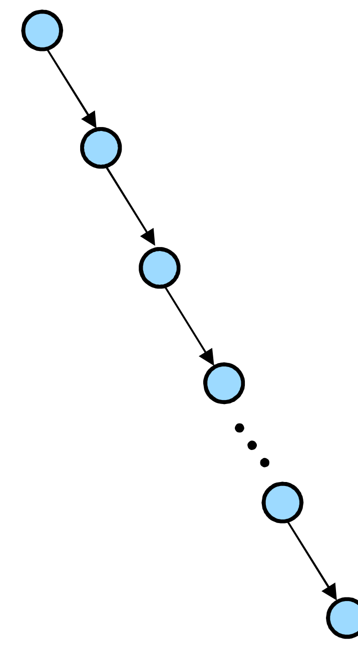
- Korrektheit folgt aus der Tatsache, dass bei jedem Pivotelement p die anderen Elemente in die richtige Teilmenge sortiert werden
- Terminierung folgt aus der Tatsache, dass die Teilmengen immer kleiner werden und schließlich bei einelementigen Mengen die Rekursion stoppt
- Laufzeiteffizienz:
  - Gesamtaufwand setzt sich aus dem Aufwand für den Teile-Schritt und für das Sortieren der entstehenden Teilfelder zusammen
  - Teile-Schritt:
    - Pivotelement wird mit jedem anderen Element verglichen, zusätzlich ein Vergleich für den Abbruch $\Rightarrow$ O(n)
  - Sortieren der Teilfelder:
    - Wie viele rekursive Aufrufe sind notwendig, d.h. wie hoch ist der Binärbaum der Rekursion?
    - Hängt sehr stark von der Wahl des Pivotelements ab
    - Günstigster Fall: Immer das mittlere Element ist Pivotelement $\Rightarrow$ Binärbaum ist ausgeglichen $\Rightarrow$ Logarithmische Höhe $\Rightarrow$ $O(\log_2 n)$ für diesen Schritt und insgesamt $O(n \cdot \log_2 n)$
    - Ungünstigster Fall: Immer das kleinste/größte Element ist Pivotelement $\Rightarrow$ Binärbaum ist Liste $\Rightarrow$ Höhe n $\Rightarrow$ O(n) für diesen Schritt und insgesamt $O(n^2)$

Bewertung (II)

- Günstigster Fall:



Ungünstigster Fall:



- Im worst case ist Quick Sort somit nicht besser als die einfachen Sortierverfahren
- Es lässt sich jedoch nachweisen, dass Quick Sort im mittleren Fall die Komplexität $O(n \cdot \log_2 n)$ besitzt

Implementierung (I)

- Algorithmus:

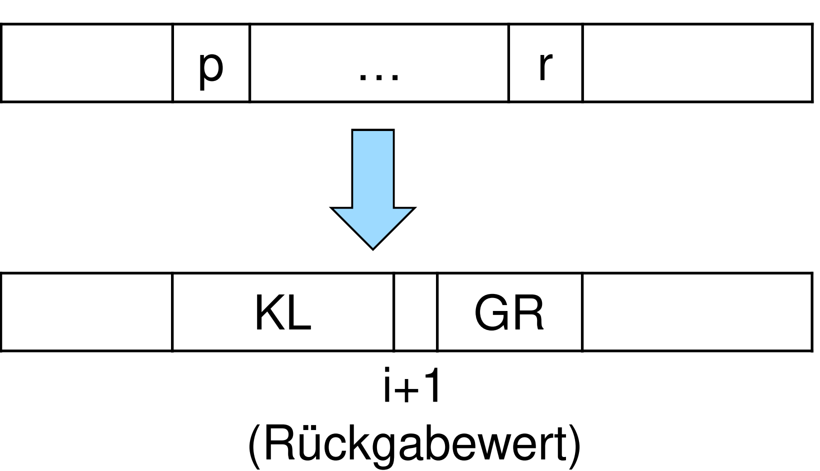
```
QuickSort(A,p,r)
( 1) if p<r {
( 2)    q = PARTITION(A,p,r)
( 3)    QuickSort(A,p,q-1)
( 4)    QuickSort(A,q+1,r)
( 5) }
```

- Erläuterung:
  - p : linker Index
  - r : rechter Index
  - q : Index Pivotelement
  - Initialer Aufruf : (A,1,A.length)

Implementierung (I)

- Methode `PARTITION(A,p,r)`:
  - Ordnet Teilfeld A[p … r] in-place um
  - p : linker Index
  - r : rechter Index
  - i : Index Pivotelement

```
Partition(A,p,r)
( 1) x = A[r]
( 2) i = p-1
( 3) for j = p to r-1{
( 4)   if A[j]<=x{
( 5)     i = i+1
( 6)     swap(A[i], A[j])
( 7)   }
( 8) }
( 9) swap(A[i+1], A[r])
(10) return i+1
```



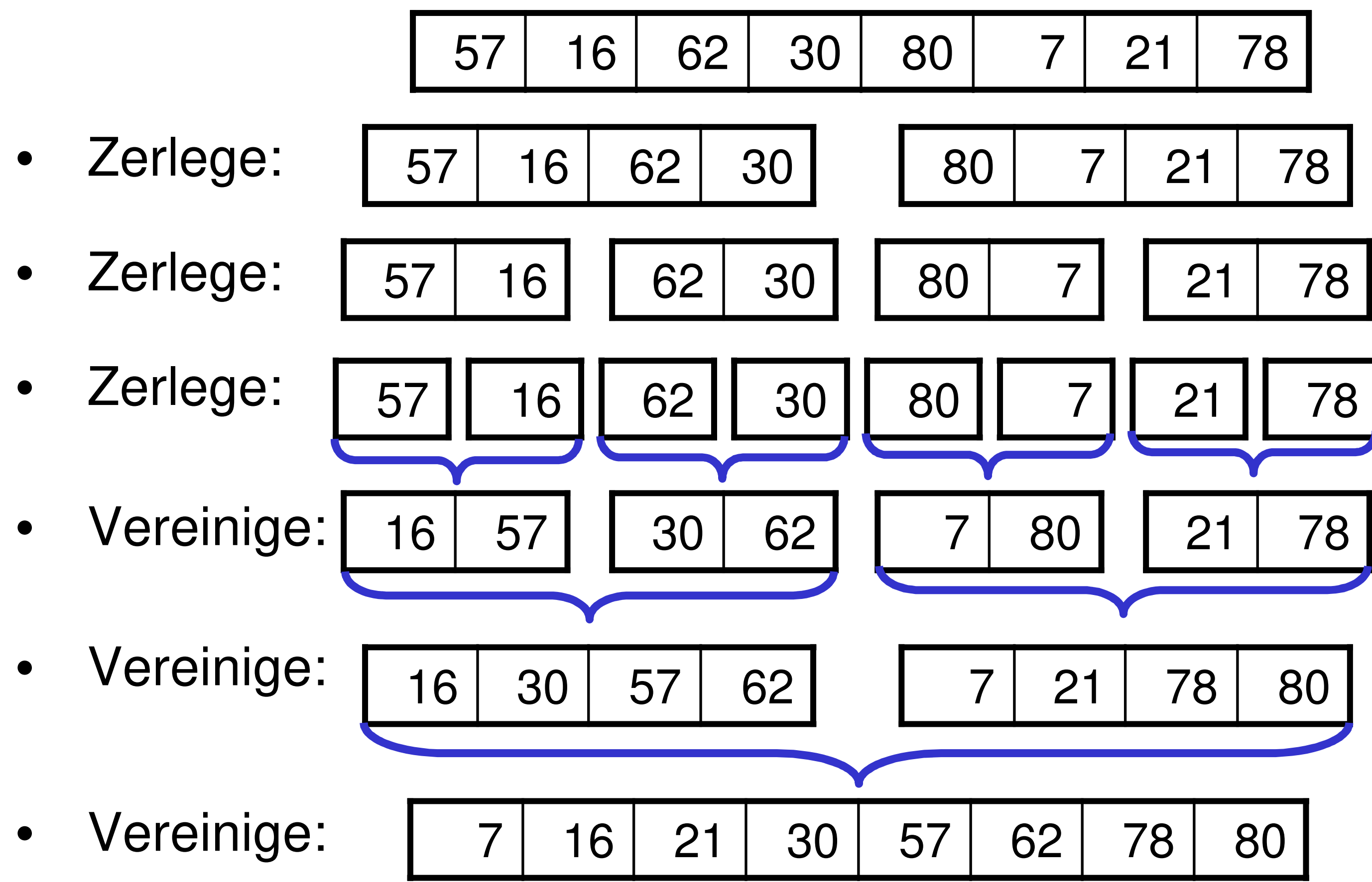
Übersicht

- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren

Grundidee

- Merge Sort folgt ähnlich wie Quick Sort einem rekursiven Divide and Conquer-Prinzip:
  - Divide-Schritt: Zerlege die Eingangsmenge rekursiv in Teilmengen, bis jede nur noch aus einem Element besteht
  - Vereinigungsschritt: Vereinige zwei (bezgl. der Zerlegungshierarchie) benachbarte Teilmengen und sortiere dabei der Größe nach
  - Fasse auf diese Weise rekursiv immer größere Teilmengen zusammen
  - Nach dem letzten Vereinigungsschritt ist das gesamte Feld sortiert

Beispiel



Bewertung (I)

Teile | Herrsche | Kombiniere

Merge Sort (n/2)

Merge Sort (n/2)

Merge(n)

A unsortiert | B,C unsortiert | B',C' sortiert | A' sortiert

Bewertung (II)

- Korrektheit folgt aus dem Sortieren beim Wiederzusammenfügen
- Terminierung folgt aus Zerlegung bis zu Feldern der Größe 1 und anschließendem Wiederzusammenfügen bis zum Erreichen der ursprünglichen Feldgröße
- Laufzeitverhalten:
  - Maß ist die Anzahl der Schlüsselvergleiche, T(n) die Zeitfunktion
  - Ein Rekursionsschritt benötigt für ein Feld von n Elementen:
    - Für die Teilung: 0 Vergleiche
    - Für das Herrschen: 2 · T(n/2) Vergleiche
    - Für das Vereinigen: n - 1 Vergleiche
    - Damit ergibt sich folgende Gesamtlaufzeit:

$$T(n) = \begin{cases} 0, & falls\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n - 1 & sonst \end{cases}$$

    - Lösung der Gleichung ergibt unter der (vereinfachenden) Annahme $n = 2^k$:

Bewertung (III)

- (Fortsetzung):

$$T(2^k) = 2 \cdot T(2^{k-1}) + 2^k$$
$$= 2 \cdot (2 \cdot T(2^{k-2}) + 2^{k-1}) + 2^k$$
$$= 4 \cdot T(2^{k-2}) + 2^k + 2^k$$
$$= \ldots$$
$$= \underbrace{2^k \cdot T(2^0)}_{=0} + \underbrace{2^k + \ldots + 2^k}_{k-mal}$$

- Also: $T(n) = T(2^k) = O(k \cdot n) = O(\log_2 n \cdot n)$ (wegen $k = \log_2 n$)

- Laufzeitverhalten unterscheidet sich im besten, durchschnittlichen und schlechtesten Fall nicht
- Anm.: Bei jedem Vereinigungsschritt wird zusätzlicher Speicher benötigt

Implementierung (I)

- Methode zum Mischen:
  - Füge als letztes Pseudoelement ein
  - Erleichtert Prüfung, ob Stapel aufgebraucht

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

	p	sortiert →	q	sortiert →	r	

L (p..q), R (q+1..r)

Mische Bereiche A[p..q] und A[q+1..r]

	p	sortiert →	r	

Implementierung (II)

- Methode MergeSort(A,p,r):
  - Sortiert Elemente im Teilfeld A[p … r]
  - Falls p >= r, dann Teilfeld max. ein Element, damit sortiert
  - Sonst:
    - Teile-Schritt: Unterteile A[p … r] in A[p … q] und A[q+1 … r]

```
MergeSort(A,p,r)
( 1) if p<r {
( 2)    q = (p+r ) DIV 2
( 3)    MergeSort(A,p,q)
( 4)    MergeSort(A,q+1,r)
( 5)    Merge(A,p,q,r)
( 6) }
```

Vergleich Quick Sort – Merge Sort

- Beide Verfahren basieren auf Divide and Conquer-Ansatz
- Quick Sort:
  - Teilen relativ kompliziert, Vereinigen einfach
- Merge Sort:
  - Teilen einfach, Vereinigen relativ kompliziert
- Laufzeitverhalten:

	Best case	Average case	Worst case
Quick Sort	$O(n \cdot \log_2 n)$	$O(n \cdot \log_2 n)$	$O(n^2)$
Merge Sort	$O(n \cdot \log_2 n)$	$O(n \cdot \log_2 n)$	$O(n \cdot \log_2 n)$

- Somit:
  - Merge Sort garantiertes $O(n \cdot \log_2 n)$-Verfahren
  - Quick Sort nicht
- Dennoch: Empirische Untersuchungen ergeben, dass Quick Sort im Mittel erheblich besser

Übersicht

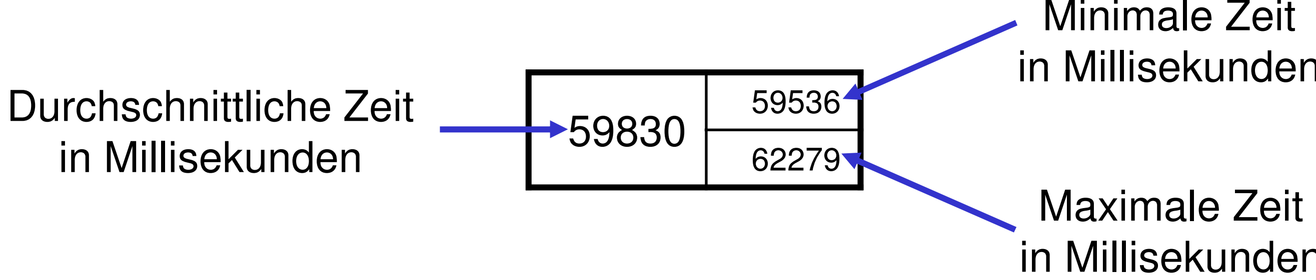
- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren

Implementierung (I)

- Feld mit Integer-Werten, anfangs zufällig belegt
- Feldgröße von 10.000, 20.000, ..., 100.000 (wie bei einfachen Verfahren im letzten Abschnitt)
- Hierbei aber alle „sehr schnell“
- Daher: Feldgröße von 1.000.000, 2.000.000, ..., 10.000.000
- Laufzeitmessung für folgende Algorithmen:
  - Shell Sort
  - Quick Sort
  - Heap Sort
  - Merge Sort

Implementierung (II)

- Durchführung:
  - Sortierung je 100-mal durchgeführt
- Plattform:
  - Java-Implementierung
  - Messung 1: Java 5, Laptop, Intel 2.2 GHz, 256 MB RAM (2005)
  - Messung 2: Java 11, Laptop, Intel Core i7 2.8 GHz, 32 GB RAM (2019)
- Ergebnisdarstellung:



Ergebnisse Messung 1 (I)

Daten-sätze	Shell Sort			Quick Sort			Heap Sort			Merge Sort		
10000	3	0	20	2	0	11	5	0	20	6	0	20
20000	7	0	30	4	0	20	10	0	30	12	10	30
30000	12	10	30	7	0	20	15	10	40	17	10	50
40000	16	10	41	10	0	30	21	20	40	31	30	50
50000	21	10	50	12	10	30	28	20	60	32	30	60
75000	34	30	70	20	10	50	45	40	80	51	50	100
100000	47	40	60	28	20	60	61	60	71	68	60	80

Ergebnisse Messung 1 (II)

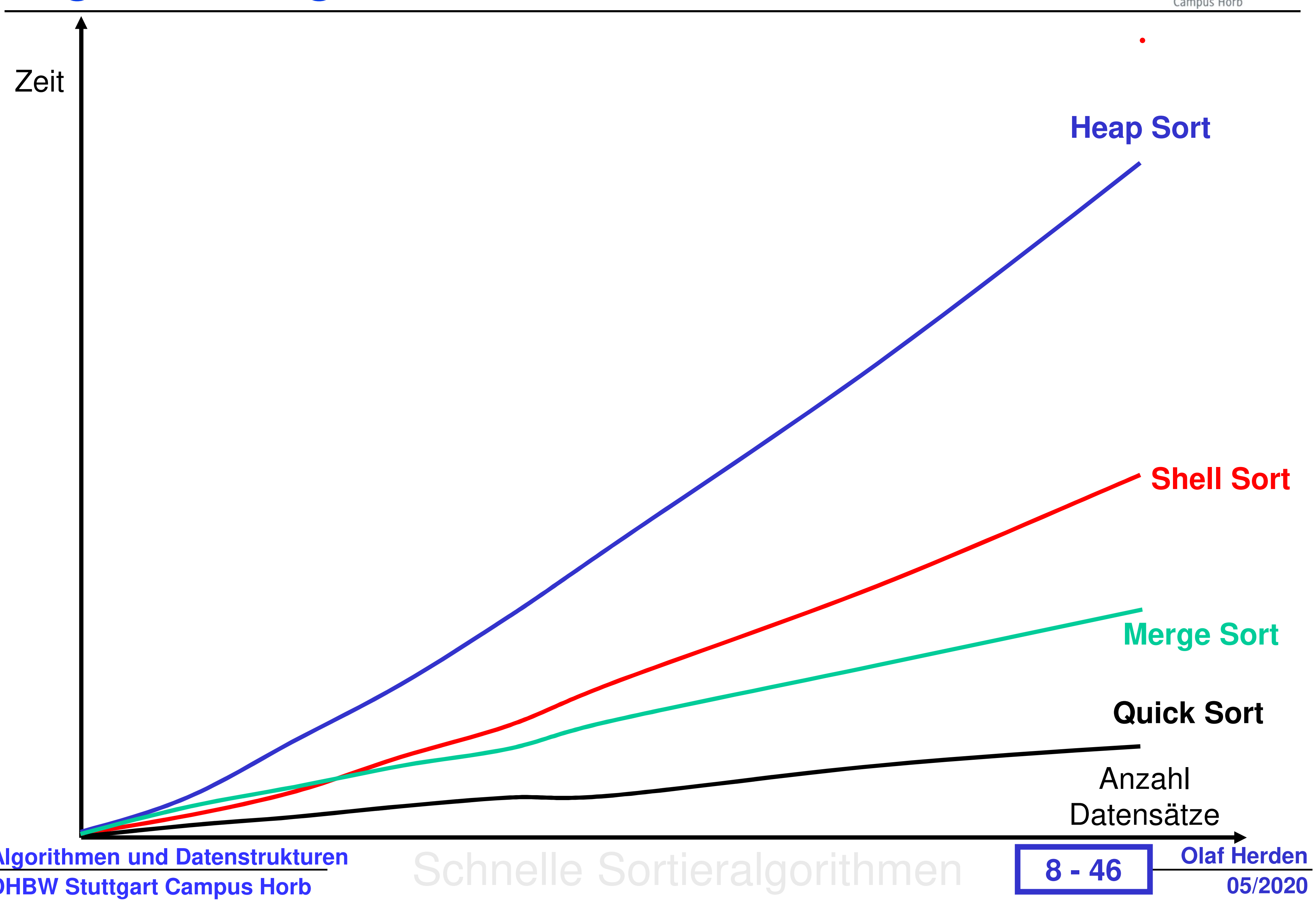
Daten-sätze	Shell Sort		Quick Sort		Heap Sort		Merge Sort	
1000000	731	701 781	325	310 401	1419	1402 1452	961	941 1032
2000000	1787	1732 1923	681	651 721	3500	3485 3536	1705	1692 1863
3000000	2968	2844 3245	1048	1011 1091	5855	5819 5949	2619	2603 2934
4000000	4295	4156 4526	1422	1382 1463	8333	8302 8423	3416	3395 3715
5000000	5711	5478 6069	1664	1612 1723	10959	10925 11085	4313	4286 4677
7500000	9492	9073 10004	2530	2473 2654	17856	17816 18026	6400	6369 6810
10000000	13640	13039 14711	3439	3365 3555	25296	25217 26088	8764*	8702 9303

*: Hier musste der Heap auf 128MB vergrößert werden, d.h. die Option –Xmx128m beim Aufruf verwendet werden

Ergebnisse Messung 2

Datensätze	Shell Sort		Quick Sort		Heap Sort		Merge Sort	
1.000.000	160	147 / 203	92	78 / 116	149	130 / 210	114	101 / 169
2.000.000	344	318 / 421	185	173 / 214	325	306 / 425	229	216 / 294
3.000.000	536	508 / 579	284	266 / 344	536	514 / 667	352	332 / 442
4.000.000	803	734 / 969	379	366 / 409	781	754 / 939	485	469 / 629
5.000.000	999	959 / 1166	482	468 / 565	1026	1001 / 1236	606	588 / 733
7.500.000	1720	1488 / 1863	751	732 / 776	1720	1688 / 1930	929	909 / 1053
10.000.000	2284	2092 / 2530	1030	1005 / 1155	2420	2390 / 2731	1261	1236 / 1389

Ergebnisse grafisch



Interpretation

- Alle Verfahren erheblich besser als einfache Verfahren aus letztem Abschnitt
- Alle Verfahren: Linear-logarithmischer Verlauf deutlich zu erkennen
- Dennoch: Erhebliche Unterschiede bei absoluten Werten
- Ursache: Konstante ist in Praxis nicht unerheblich
- Quick Sort (im Mittel) schnellstes internes Sortierverfahren

Übersicht

- Algorithmen mit Vorsortierung (Shell Sort)
- Heap Sort
- Quick Sort
- Merge Sort
- Implementierung/Vergleich
- Komplexität von Sortierverfahren

Aufwand des Sortierens (I)

- Bisher:
  - Schnellste Verfahren mit $O(n \cdot \log_2 n)$
  - Geht es noch besser?
- Antwort: Nein!
- Ein Sortierverfahren, das auf dem Vergleich von Elementen untereinander beruht, kann im worst case bestenfalls eine Komplexität von $O(n \cdot \log_2 n)$ besitzen.
- Beweis:
  - Sei $V = (v_1, ..., v_n)$ eine Folge von zu sortierenden Elementen, die o.B.d.A. alle voneinander verschieden seien
  - Sortieren bedeutet umordnen, d.h. permutieren
  - Insgesamt gibt es n! Permutationen einer n-elementigen Folge
  - Jede mögliche Permutation σ kann in Frage kommen, um die ursprünglich ungeordnete Folge V zur geordneten Folge V‘ zu ordnen
  - Jeder Vergleich $v_i<v_j$ zerlegt Menge der in Frage kommenden Permutationen in zwei Teilmengen T_1 und T_2: In den Permutationen in T_1 ist v_i vor v_j angeordnet, in denen von T_2 nicht

Aufwand des Sortierens (II)

- Beweis (Fortsetzung):
  - Wenn 1,2, ..., n die Indizes der Elemente sind, dann suchen wir diejenige unter den n! Permutationen diejenige, die die Elemente in sortierte Reihenfolge bringt, d.h. $(i_1, ..., i_n)$ und $v_{i_j} \leq v_{i_k}$ für alle Paare i_j und i_k mit j<k
  - Bestenfalls kann die korrekte Permutation durch einen Entscheidungsbaum gefunden werden, dessen Blätter alle n! Permutationen sind:

Menge aller Permutationen einer n-elementigen Menge

$v_i<v_j$

Alle Permutationen mit $v_i<v_j$

Alle Permutationen mit $v_i>v_j$

$v_k<v_i$

$v_k<v_i$

$v_i<v_j$ und $v_k<v_i$

$v_i<v_j$ und $v_k>v_i$

$v_i>v_j$ und $v_k<v_i$

$v_i>v_j$ und $v_k>v_i$

n! Elemente

Mindestens n!/2 Elemente

Mindestens n!/2² Elemente

Aufwand des Sortierens (III)

- Beweis (Fortsetzung):
  - Frühestens nach $\log_2(n!)$ vielen Zerlegungen, d.h. nach $\log(n!)$ vielen Vergleichen, hat man die Gesamtmenge so zerlegt, dass in jeder Teilmenge höchstens ein Element übrigbleibt
  - Für große n gilt aufgrund der Stirlingschen Formel: $n! \approx n^n$
  - Daraus folgt: $O(\log_2(n!)) = O(n \cdot \log_2 n)$.

Zusammenfassung

- Algorithmen mit Vorsortierung:
  - Idee
  - Shell Sort
- Heap Sort:
  - Baumbasiertes Verfahren
  - Garantiert $O(n \cdot \log_2 n)$
- Divide and Conquer-Verfahren:
  - Quick Sort
  - Merge Sort
  - Vergleich
- Implementierung/Vergleich
- Komplexität von Sortierverfahren
  - Auf binären Vergleichen basierende Verfahren können bestenfalls Komplexität $O(n \cdot \log_2 n)$ haben

Übungen (I)

- **Aufgabe 1**

  Gegeben sei das folgende Feld mit Vornamen, das gemäß lexikographischer Ordnung absteigend sortiert werden soll!

Steffi	Michael	Stefanie	Astrid	Stefan	Stephanie	Stephan	Xaver

  Führen Sie diesen Sortiervorgang mithilfe der Verfahren Shell Sort, Heap Sort, Quick Sort und Merge Sort durch. Geben Sie dabei die Zwischenresultate eines jeden Durchgangs an!

- **Aufgabe 2**

  Konstruieren Sie für das Feld aus Aufgabe 1 einen Ablauf des Sortierverfahrens Quick Sort an, der worst case-Verhalten aufzeigt.

Übungen (II)

- **Aufgabe 3**

	Wahr	Falsch
Heap Sort ist ein Divide and Conquer-Verfahren.		
Im schlimmsten Fall ist die Laufzeit von Quick Sort nicht besser als die von Bubble Sort.		
Auf binären Vergleichen beruhende Sortierverfahren können bestenfalls quadratisches Laufzeitverhalten haben.		
Ein Heap ist immer zyklenfrei.		
Shell Sort arbeitet mit Vorsortierung.		
In einem vorsortierten Feld kann Bubble Sort schneller sein als Quick Sort.		
Merge Sort und Quick Sort beruhen beide auf dem Divide and Conquer-Prinzip.		
Heap Sort ist ein garantiertes $O(n \cdot \log_2(n))$-Verfahren.		
Quick Sort ist das schnellste bekannte interne Sortierverfahren.		

Übungen (IV)

	Wahr	Falsch
Bucket Sort ist nicht in jedem Fall anwendbar.		
Comb Sort ist ein auf Bubble Sort basierendes Sortierverfahren.		
Die Folge fallender Werte für die Sequenzen bei Shell Sort ist leicht zu bestimmen.		
Die Wahl des Pivotelements bei Quick Sort hat keine Auswirkungen auf die Laufzeit.		

Algorithmen und Datenstrukturen

Teil 9:
Sortieren (III) –
Spezielle und externe Sortierverfahren

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2019

Übersicht

- Counting Sort

- Sortieren durch Streuen und Sammeln

- Externes Sortieren

- Mischverfahren

Idee

- Nicht auf Vergleichen basierend
- Setzt Wissen über mögliche Werte des Sortierschlüssels voraus
- Gegeben:
  - Menge M = {0, 1, …, k} von Schlüsselwerten
  - Unsortiertes Feld A[n] mit Werten aus M
- Lösungsidee:
  - Initialisiere Hilfsfeld H[k]
  - Navigiere über Feld A, erhöhe dabei H[j], wenn A[i] = j
  - Navigiere über H, gebe dabei Wert so oft aus wie vorhanden

Beispiel

- Sei M = {0, 1, …, 7}
- Zu sortierende Werte (Eingabefeld A):

  `5 4 1 4 6 1 3 0 0 5 5 7 2 2 5 7 0 2 7 7`

- Initialisieren von Hilfsfeld H:

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

- Füllen von Hilfsfeld H:

0	1	2	3	4	5	6	7
3	2	3	1	2	4	1	4

- Ausgabefeld B:

  `0 0 0 1 1 2 2 2 3 4 4 5 5 5 5 6 7 7 7 7`

Implementierung (I)

- Gegeben:
  - Wertebereich {0, …, k}
  - Eingabefeld A[n] mit Werten aus {0, …, k}
- Ausgabe:
  - Ausgabefeld B[n]

```
( 1) // Hilfsfeld initialisieren
( 2) for i = 0 to k{
( 3)   H[i] = 0
( 4) }
( 5) // Über Eingabefeld iterieren und suchen
( 6) for j = 1 to n{
( 7)   H[A[j]]++;
( 8) }
( 9) // H[i] enthält nun Vorkommen von in A
     ...
```

Implementierung (II)

```
      ...
(10) // Werte ins Ergebnisfeld kopieren
(11) actualIndex = 0;
(12) for i = 0 to k{
(13)   for j = 1 to H[i]{
(14)     B[actualIndex] = i;
(15)     actualIndex++;
(16)   }
(17) }
```

- Anmerkungen:
  - Anderer Schlüsselbereich mit leichten Modifikationen realisierbar
  - Statt separatem Ergebnisfeld kann Ergebnis auch in A gespeichert werden
  - Variante Tally Sort:
    - Keine Duplikate vorhanden oder diese sollen eliminiert werden
    - Hilfsfeld kann Typ `boolean` haben

Aufwand

- Laufzeit:
  - Abhängig von n (Anzahl zu sortierender Werte)
  - Abhängig von k (Anzahl Schlüsselwerte)
  - Schleifen werden nacheinander durchlaufen
  - Insgesamt O(n+k)

- Speicher:
  - Zusätzlich zur Eingabe ist mindestens noch Feld der Dimension k notwendig

Übersicht

- Counting Sort
- Sortieren durch Streuen und Sammeln
- Externes Sortieren
- Mischverfahren

Grundidee

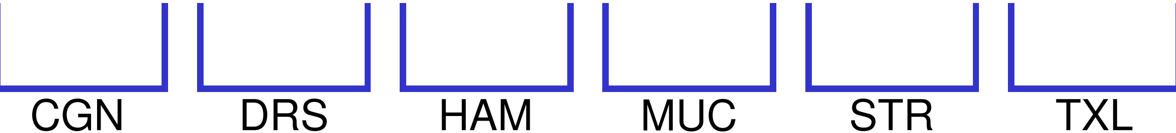
- Verallgemeinerung von Counting Sort:
  - Beliebige Schlüsselwerte
  - Auch hier: „Nicht allzu viele Elemente“ (im Vergleich zur zu sortierenden Datenmenge)
  - Mehrstufigkeit bei hierarchischem Aufbau
- Ablauf:
  - Streuphase: Verteilung der Elemente auf Fächer (Buckets)
  - Evtl. Sortierung innerhalb des Buckets
  - Einsammelphase: Inhalt der Buckets nacheinander ausgeben
- Spezialfall:
  - Schlüsselwert fester Länge und hierarchisch aufgebaut, z.B. PLZ
  - Innerhalb der Buckets kann gleiches Verfahren erneut angewendet werden

Beispiel

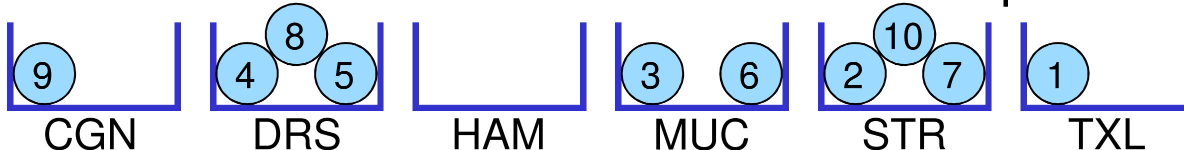
- Feld mit Daten über Flüge soll nach Abflughäfen sortiert werden:

```
(1) TXL    HAM    19:50   ...
(2) STR    MUC    12:10   ...
(3) MUC    TXL    17:05   ...
(4) DRS    DRS    08:15   ...
(5) DRS    HAM    14:20   ...
(6) MUC    HAM    21:00   ...
(7) STR    CGN    17:00   ...
(8) DRS    STR    16:10   ...
(9) CGN    STR    13:00   ...
(10)STR    HAM    11:30   ...
```

- Lege für jedes Flughafenkürzel Fach an:



- Evtl.: Sortieren innerhalb der Buckets mit bekanntem Verfahren
- Gehe Datensätze durch und diesen in entsprechendem Fach ab:



- Gehe die Fächer durch und gebe die Datensätze aus

Bewertung

- Korrektheit folgt aus bekannter Sortierung der Fächer
- Terminierung folgt aus Endlichkeit des Feldes
- Laufzeitverhalten:
  - Streuen: n Schritte (Einsortieren in Fächer)
  - Sammeln: m Schritte (Ausgeben der Fächerinhalte)
  - Insgesamt: O(n+m)
  - Da m als konstant angenommen werden kann, gilt: O(n)

Übersicht

- Counting Sort
- Sortieren durch Streuen und Sammeln
- Externes Sortieren
- Mischverfahren

Motivation

- Bisher: Sortieren von Daten, die vollständig im Hauptspeicher
- Zu sortierende Datenmenge größer als Hauptspeicher => bisherige Verfahren nicht anwendbar
- Externe Sortierverfahren als Lösung
- Verschiedene Ansätze:
  - Externes Sortieren durch horizontales Zerlegen
  - Externes Sortieren durch vertikales Zerlegen
  - Mischverfahren
- Externes Sortieren durch horizontales Zerlegen:
  - Zerlege den Datenbestand in kleinere Datenmengen
  - Sortiere diese mit einem internen Sortierverfahren
  - Füge die Teillösungen wieder zusammen
- Externes Sortieren durch vertikales Zerlegen:
  - Zerlege zu sortierende Elemente in Schlüssel und „Restdaten“
  - Sortiere Schlüssel mit internem Sortierverfahren

Ext. Sortieren durch horizontales Zerlegen

1. Zerlegen der Datei D
2. Sortieren der Teildateien
3. Zusammenfügen

Unsortierte Datei → D_1, D_2, …, D_n → Sortierprogramm / Datei im Hauptspeicher → D_1‘, D_2‘, …, D_n‘ → Sortierte Datei

Ext. Sortieren durch vertikales Zerlegen

Daten-
satz

Zerlegung

Schlüssel

Zeiger auf
Restdaten

Rest-
daten

Vor dem
Sortieren:

X | F | V | B | Z

Rest-
daten

Nach dem
Sortieren:

B | F | V | X | Z

Rest-
daten

Übersicht

- Counting Sort
- Sortieren durch Streuen und Sammeln
- Externes Sortieren
- Mischverfahren

Mischverfahren

- Mischen: Zusammenfügen einzelner sortierter Läufe (Synonym: Folgen oder Sequenzen)
- Dabei: Übernehme jeweils kleineres Element aus Ausgangsläufen in Ergebnislauf
- Ist jede Sequenz einer Datei zugeordnet, ist dies besonders geeignet
  - Lauf 1:

2	12	13	15	20	29

  - Lauf 2:

4	5	9	14	22

  - Ergebnislauf:

2	4	5	9	12	13	14	15	20	22	29

- Erweiterung Mischtechnik auf allgemeines externes Sortierproblem: Gegeben ist Datei unbekannter Länge, deren Elemente sortiert werden sollen

Direktes Mischen (I)

- Ständiger Wechsel zwischen Trennphase (Verteilung der Läufe auf meist zwei sequentielle Dateien) und Mischphase
- Wird auch als Zwei-Phasen-Mischen bezeichnet
- Beide Phasen zusammen bilden Durchlauf
- Bei jedem Durchlauf verdoppelt sich Länge der Ausgangsläufe:
  - Anfangs in unsortierter Datei Läufe der Länge 1
  - Daraus folgt, dass nach maximal $\log_2(n)+1$ Durchläufen gesamte Sequenz sortiert ist

Direktes Mischen (II): Beispiel (I)

- Notation: Benachbarte, farbgleiche Elemente bilden einen Lauf
- Datei:

7	13	9	4	25	1	30	12

- Trennen
  - Hilfsdatei 1:

7	9	25	30

  - Hilfsdatei 2:

13	4	1	12

- Mischen
  - Datei:

7	13	4	9	1	25	12	30

Direktes Mischen (III): Beispiel (II)

- Datei:

7	13	4	9	1	25	12	30

- Trennen
  - Hilfsdatei 1:

7	13	1	25

  - Hilfsdatei 2:

4	9	12	30

- Mischen
  - Datei:

4	7	9	13	1	12	25	30

- Trennen
  - Hilfsdatei 1:

4	7	9	13

  - Hilfsdatei 2:

1	12	25	30

- Mischen
  - Datei:

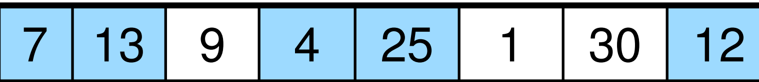
1	4	7	9	12	13	25	30

Natürliches Mischen (I)

- Direktes Mischen nutzt vorhandene Sortierung nicht aus
- Länge aller gemischten Teilsequenzen nach dem k-ten Durchlauf ist 2^k
- Zwei bereits vorhandene, sortierte Sequenzen können nicht direkt verwertet werden
- Daher: Beim natürlichen Mischen werden nicht Sequenzen fester Länge, sondern jeweils Teilsequenzen maximaler Länge gemischt

Natürliches Mischen (II): Beispiel (I)

- Notation: Benachbarte, farbgleiche Elemente bilden Lauf
- Datei:



- Trennen
  - Hilfsdatei 1:

7	13	4	25	12

  - Hilfsdatei 2:

9	1	30

- Mischen
  - Datei:

7	9	13	1	4	25	30	12

- Trennen
  - Hilfsdatei 1:

7	9	13	12

  - Hilfsdatei 2:

1	4	25	30

- Mischen
  - Datei:

1	4	7	9	13	25	30	12

Natürliches Mischen (III): Beispiel (II)

- Datei:

1	4	7	9	13	25	30	12

- Trennen
  - Hilfsdatei 1:

1	4	7	9	13	25	30

  - Hilfsdatei 2:

12

- Mischen
  - Datei:

1	4	7	9	12	13	25	30

- Bewertung:
  - Algorithmus erheblich aufwändiger als direktes Mischen
  - Außer bei sehr langen vorsortierten Sequenzen kaum Leistungssteigerung
  - Im unsortierten Feld natürliche Sequenzen im Mittel Länge 2 => (nur) ein Durchlauf wird gespart

Mehr-Wege-Mischen (I)

- Natürliches Mischen bringt gegenüber direktem Mischen kaum einen Vorteil
- Folgende effizientere Form des Mischens ist aber möglich:
  - Verzicht auf das Trennen der Läufe (außer beim ersten Mal)
  - Stattdessen: Unmittelbar beim Mischen sofort wieder trennen, d.h. z.B. zwei Dateien mischt und die entstehenden Läufe abwechselnd in zwei andere Dateien schreibt
  - Diesen Vorgang wiederholt man in umgekehrter Richtung, bis nur noch eine Sequenz übrigbleibt
  - Z.B. bei drei Dateien: Zu jeder Zeit wird von zwei Dateien auf die dritte gemischt
- Beispiel: (Benachbarte, farbgleiche Elemente bilden einen Lauf)
  - Datei:

7	13	9	4	25	1	30	12

  - Trennen
    - Hilfsdatei 1:

7	9	25

    - Hilfsdatei 2:

13	4	1	30	12

Mehr-Wege-Mischen (II)

- Beispiel (Fortsetzung):
  - Trennen
    - Hilfsdatei 1:

7	9	25

    - Hilfsdatei 2:

13	4	1	30	12

  - Mischen der ersten drei Paare aus Hilfsdateien 1 und 2 nach 3:
    - Hilfsdatei 3:

7	13	4	9	1	25

  - Mischen der ersten zwei Paare aus Hilfsdateien 2 und 3 nach 1:
    - Hilfsdatei 1:

7	13	30	4	9	12

  - Mischen des ersten Paares aus Hilfsdateien 1 und 3 nach 2:
    - Hilfsdatei 2:

1	7	13	25	30

  - Mischen des ersten Paares aus Hilfsdateien 1 und 2 in die Ausgangsdatei:
    - Datei:

1	4	7	9	12	13	25	30

Mehr-Wege-Mischen (III)

- Bewertung:
  - Zahl der Läufe auf den beiden nicht-leeren Dateien ist ein Paar aufeinander-folgender Fibonacci-Zahlen
  - Durch eine solche Verteilung wird die Anzahl der Mischvorgänge für eine gegebene Anzahl von Ausgangsläufen minimiert
  - Diese Tatsache stellt aber auch genau das Problem dar: Größe der zu sortierenden Datei ist im Allgemeinen nicht bekannt, so dass Fibonacci-Verteilung nicht möglich ist
  - Dies führt dann zu „Restelementen“, deren Verteilung auf die Läufe und Behandlung im Algorithmus erhebliche Probleme bereiten kann

Zusammenfassung

- Counting Sort:
  - Spezialverfahren bei bekannten Schlüsseln
- Sortieren durch Streuen und Sammeln:
  - Bessere Laufzeit, falls Sortierschlüssel spezielle Eigenschaft erfüllen
  - Verfahren damit aber nicht mehr universell anwendbar
- Externes Sortieren:
  - Externes Sortieren bei großen Datenmengen
  - Horizontale vs. vertikale Zerlegung der Daten
- Mischverfahren:
  - Direktes Mischen
  - Natürliches Mischen
  - Mehr-Wege-Mischen

Übungen (I)

- **Aufgabe 1**

  Gegeben ist das folgende Feld mit Zahlen:

23	66	42	11	69	5	77	55

  Führen Sie diesen Sortiervorgang mithilfe der drei in der Vorlesung vorgestellten Mischverfahren durch. Geben Sie dabei auch jeweils die einzelnen Schritte an!

- **Aufgabe 2**

  Gegeben ist das folgende Feld mit Zahlen:

23	5	8	11	5	5	11	8

  Sortieren Sie dieses Feld unter Verwendung von Counting Sort!

  Die Menge zulässiger Schlüsselwerte sei {5,8,11,17,23}!

Übungen (II)

- **Aufgabe 3**

  Sortieren Sie die folgenden Briefe durch das Streu- und Sammel-Verfahren Bucketsort:

```
55777 ...
70543 ...
71083 ...
54222 ...
70666 ...
71083 ...
71245 ...
71888 ...
71248 ...
71222 ...
55777 ...
55779 ...
70662 ...
71083 ...
71248 ...
```

Übungen (III)

- **Aufgabe 4**

	wahr	falsch
Das Ausnutzen vorhandener Vorsortierung bringt für natürliches Mischen erhebliche Laufzeitvorteile.		
Vertikale Zerlegung von Datensätzen zum externen Sortieren bietet sich besonders bei sehr langen Datensätzen an.		
Beim direkten Mischen verdoppelt sich mit jedem Durchlauf die Länge der Ausgangsläufe.		
Mehr-Wege-Mischen ist einfach und effizient zu implementieren.		
Externe Sortierverfahren werden angewendet, um bekannte Sortierverfahren zu beschleunigen.		
Mehr-Wege-Mischen erreicht Effizienzvorteile, da auf das Trennen von Läufen verzichtet wird.		
Bei der vertikalen Zerlegung von Datensätzen wird der Schlüssel von den „Restdaten“ abgetrennt.		
Natürliches Mischen garantiert konstante Laufzeit.		

Algorithmen und Datenstrukturen

Teil 10:
Suchen (I) – Felder und Bäume

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

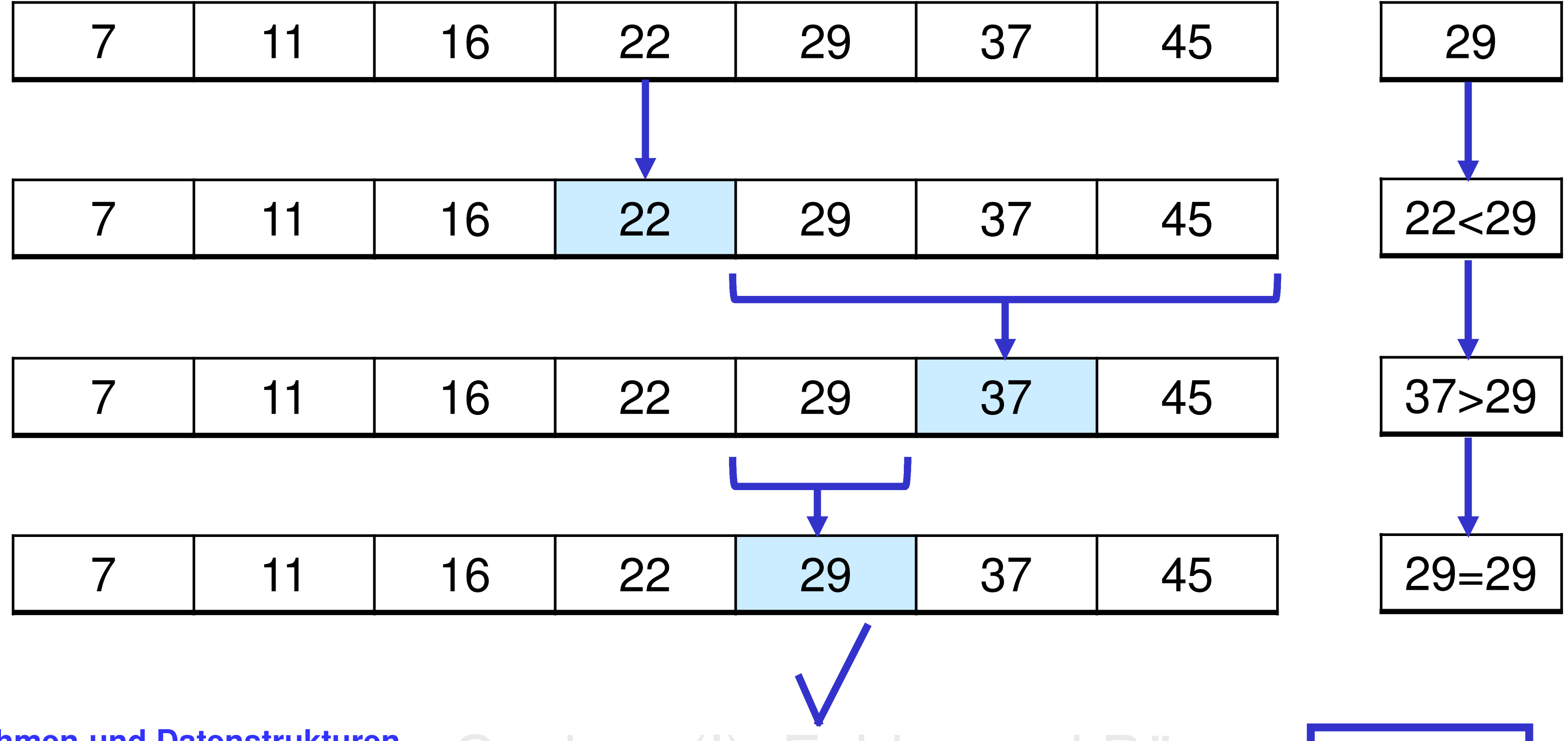
- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

Suchen in Feldern

- Suchen in Feld ohne weitere Annahmen
- Vorgehen:
  - Beginnen beim ersten Element
  - Bis zum Ende durchgehen
- Komplexität:
  - Maß: Anzahl von Vergleichen
  - Best case (Gesuchtes Element an erster Position): O(1)
  - Worst case (Gesuchtes Element an letzter Position oder nicht im Feld): O(n)
  - Average case: Im Mittel sind n/2 Vergleiche notwendig, falls gesuchtes Element im Feld $\Rightarrow$ O(n)

Binäres Suchen (I)

- Ist Feld sortiert, kann binäres Suchen angewendet werden:
  - Zunächst gesuchtes Element e mit mittlerstem vergleichen
  - Ist e größer, Vorgehen im rechten Teilfeld fortsetzen, ansonsten im linken
  - Verfahren fortsetzen, bis Element gefunden oder einelementiges Feld übrigbleibt
- Beispiel:



Binäres Suchen (II)

- Komplexität:
  - Maß: Anzahl Vergleiche
  - Best case (Gesuchtes Element an mittlerster Position): O(1)
  - Worst case (Gesuchtes Element nicht im Feld):
    - Frage: Wann ist einelementiges Feld erreicht?
    - Mit jedem Element halbiert sich die Anzahl der Element im relevanten Feld: n, n/2, n/4, n/8, ... $\Rightarrow$ Nach $\log_2 n$ Schritten ist einelementiges Feld erreicht $\Rightarrow O(\log_2 n)$
  - Average case: Im Mittel halb so viele Vergleiche notwendig wie im worst case, falls gesuchtes Element im Feld enthalten ist $\Rightarrow O(\log_2 n)$

Vergleich lineare vs. binäre Suche (I)

- Effizienzsteigerung:

Anzahl Elemente	Maximale Anzahl Vergleiche Lineare Suche	Maximale Anzahl Vergleiche Binäre Suche
1.000	1.000	10
2.000	2.000	11
5.000	5.000	13
1.000.000	1.000.000	20
2.000.000	2.000.000	21
5.000.000	5.000.000	23
1.000.000.000	1.000.000.000	30
2.000.000.000	2.000.000.000	31
5.000.000.000	5.000.000.000	33

Vergleich lineare vs. binäre Suche (II)

- Zusammenfassung Komplexitäten:

	Best Case	Average Case	Worst Case
Lineare Suche	O(1)	O(n)	O(n)
Binäre Suche	O(1)	$O(\log_2 n)$	$O(\log_2 n)$

- Fazit: Binäre Suche in großen Datenbeständen deutlich schneller
- Allerdings: Vernachlässigung von Kosten für:
  - Sortierung
  - Einfügen an richtiger Stelle
  - Schließen von Lücken beim Löschen (Umspeichern des Feldes)

Interpolationssuche

- Weitere Verbesserung binärer Suche
- Idee:
  - Verbesserung binärer Zugriff durch Ersetzen des Teilungsfaktors ½ durch angepassten Interpolationswert m
  - Annahme: Datensätze <d_1, ..., d_n> sind bezgl. ihrer Schlüsselwerte $s(d_1)$, ..., $s(d_n)$ etwa gleich verteilt
  - Ermittlung erwartete Position m mit gesuchtem Schlüsselwert s:

$$m = 1 + \left(\frac{s - s(d_1)}{s(d_n) - s(d_1)} \right) \cdot (n-1)$$

s

$s(d_1)$ $s(d_n)$

  - Analogie: Bei Suchen von „Zausel“ im Telefonbuch weiter hinten aufschlagen, bei „Abel“ recht weit vorne und bei „Meier“ etwa in der Mitte
- Eigenschaften:
  - Im Durchschnitt liegt der Aufwand bei $O(\log_2(\log_2(n)))$
  - Im worst case liegt der Aufwand bei O(n)

Suchen in dynamischen Strukturen

- Listen müssen (unabhängig von Sortierung) immer von vorne nach hinten durchsucht werden
- Komplexität:
  - Maß: Anzahl von Vergleichen
  - Best case (Gesuchtes Element steht an erster Position): O(1)
  - Worst case und Average case: O(n)
  - Zusätzlich: Konstanten für die Navigation (Umhängen der Zeiger)

Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

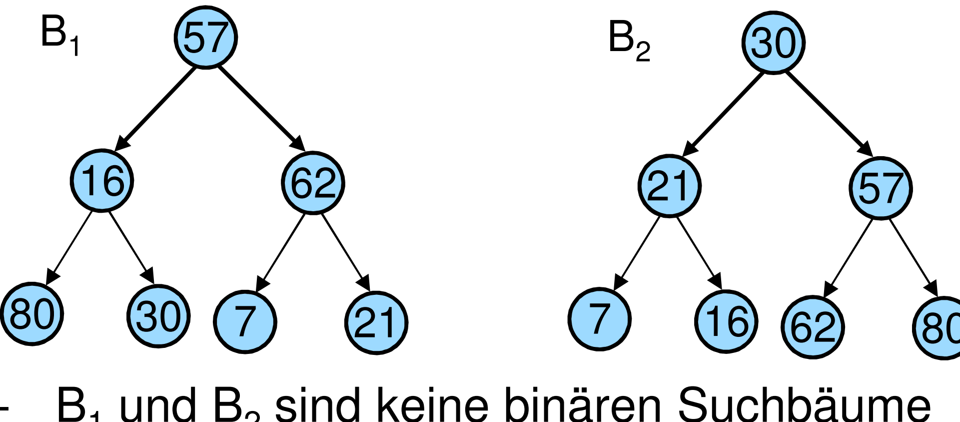
Motivation

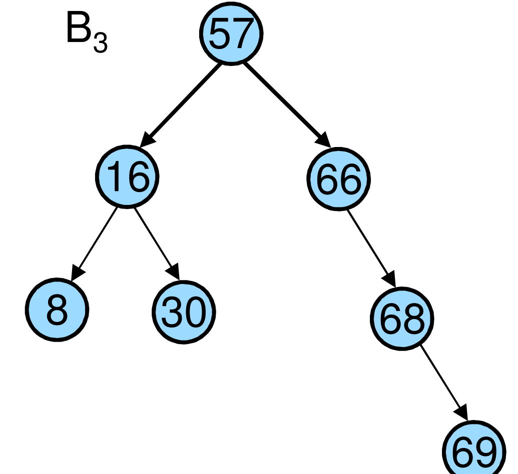
- Ziele:
  - Effiziente Verarbeitung großer geordneter Datenbestände
  - Insbesondere effiziente Aufsuchoperationen
  - Sortierte Verarbeitung des gesamten Datenbestandes
- Annahmen:
  - Zugriff auf Datensätze erfolgt über Schlüssel
  - Auf den Schlüsselwerten ist eine Ordnung definiert
  - Schlüsselwerte sind eindeutig im gesamten Datenbestand
  - Letzte Annahme vereinfacht die Darstellung etwas, durch leichte Modifikation der Algorithmen kann diese Forderung aufgehoben werden

Definition

- Binärbaum B heißt binärer Suchbaum, wenn
  - B ist leer

  oder jeder Knoten in B enthält einen Schlüssel mit:
  - Alle Schlüssel im linken Unterbaum von B sind kleiner als der Schlüssel in der Wurzel von B
  - Alle Schlüssel im rechten Unterbaum von B sind größer als der Schlüssel in der Wurzel von B
  - Linker und rechter Unterbaum von B sind jeweils auch binäre Suchbäume.
- Beispiele:

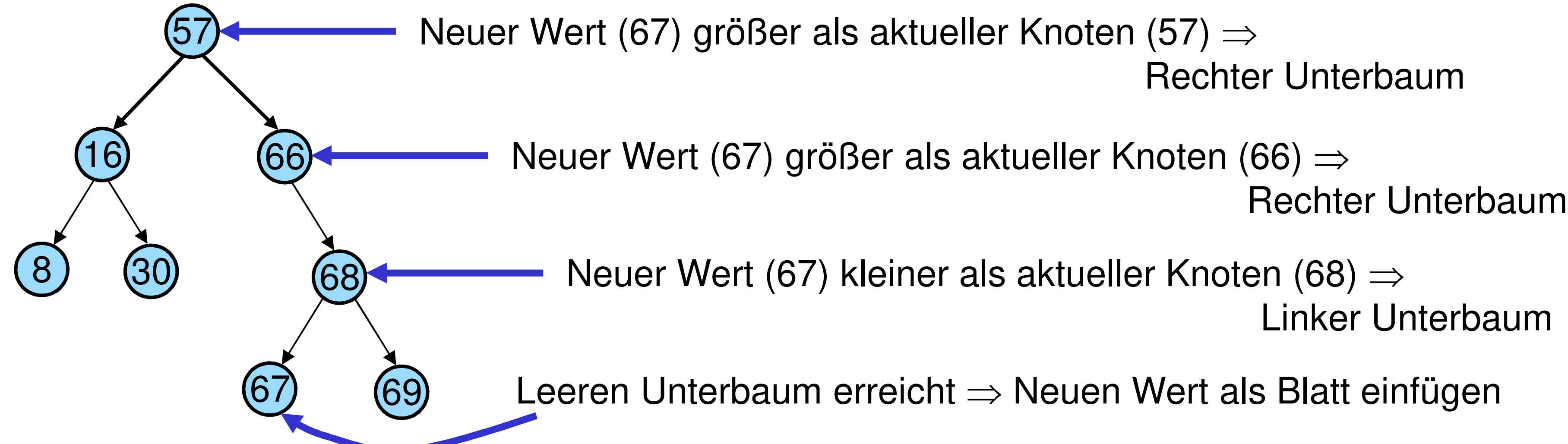




  - B_1 und B_2 sind keine binären Suchbäume
  - B_3 ist binärer Suchbaum

Einfügen eines Knotens

- Neue Knoten werden immer als Blätter eingefügt
- Position des Blattes wird durch den Schlüssel des neuen Knotens festgelegt
- Ist der Baum leer, so bildet der einzufügende Knoten die Wurzel
- Ist der Baum nicht leer, so wird an der Wurzel beginnend in den rechten (wenn neuer Schlüssel größer als Knotenwert ist) oder linken Unterbaum (wenn neuer Schlüssel kleiner als Knotenwert ist) gegangen, bis man bei einem leeren Unterbaum angekommen ist
- Beispiel: Füge 67 ein

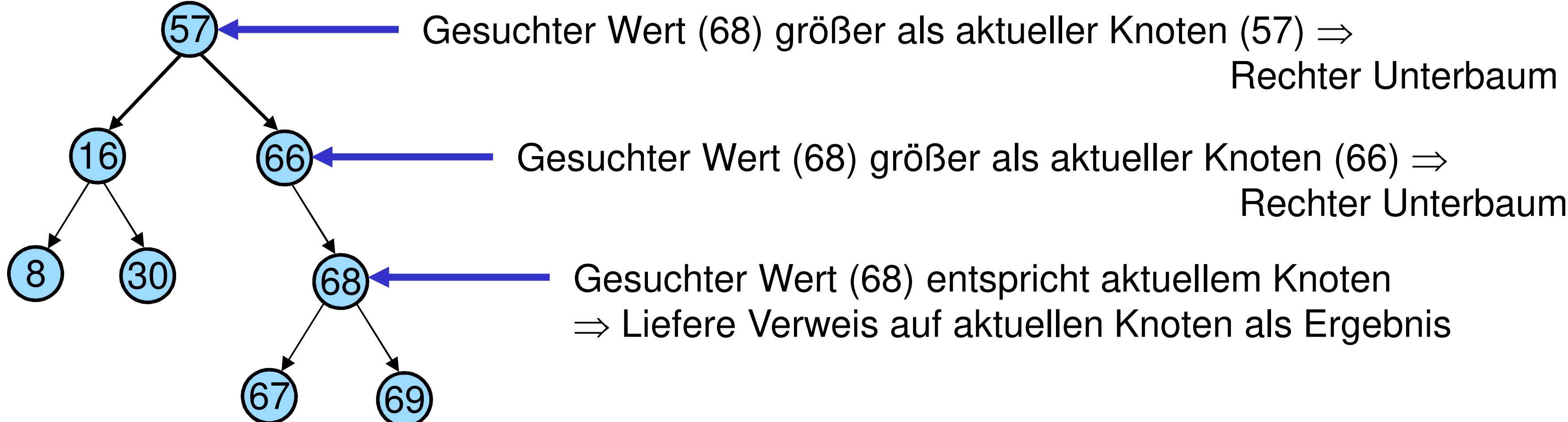


Anmerkungen

- Zu gegebener Schlüsselmenge gibt es eine Vielzahl von binären Suchbäumen
- Reihenfolge des Einfügens bestimmt hierbei Aussehen des binären Suchbaums
- Es gilt:
  - n Schlüssel $\Rightarrow$ n! verschiedene Schlüsselfolgen
  - Einige führen aber auf den gleichen binären Suchbaum
  - Bei n Schlüsseln gibt es $\frac{1}{n+1} \cdot \binom{2n}{n}$ verschiedene binäre Suchbäume
- Einfügereihenfolge beeinflusst maßgeblich die Höhe des Baums
- Schlimmster Fall bei sortierter Einfügereihenfolge: Binärer Suchbaum degeneriert zu linearer Liste
- Einfügealgorithmus sehr einfach, da keine Ausgleichs- oder Reorganisationsoperationen notwendig sind

Suchen eines Knoten (I)

- Position eines Knotens wird nach gleichem Verfahren wie beim Einfügen eines Knotens gesucht
- Methoden liefern eine Referenz auf den Knoten zurück, oder den Wert null bei nicht erfolgreicher Suche
- Beispiele: Suche den Wert 68



Suchen eines Knoten (II)

- Beispiel: Suche den Wert 44

57, 16, 66, 8, 30, 68, 67, 69

Gesuchter Wert (44) kleiner als aktueller Knoten (57) ⇒ Linker Unterbaum

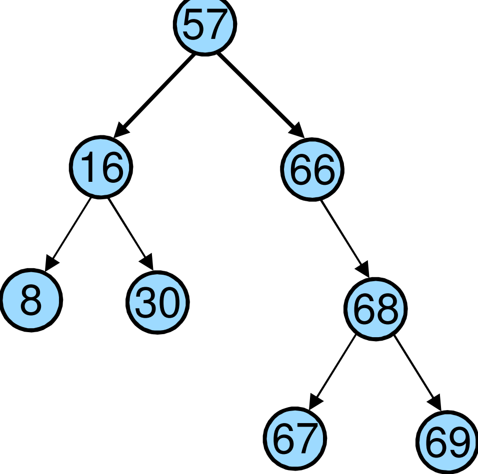
Gesuchter Wert (44) größer als aktueller Knoten (16) ⇒ Rechter Unterbaum

Gesuchter Wert (44) größer als aktueller Knoten (30) ⇒ Rechter Unterbaum

Aktueller Unterbaum ist leer
⇒ Gesuchtes Element nicht im Baum
⇒ Liefere null als Resultat zurück

Sortierte Ausgabe aller Knoten

- Sortierte Ausgabe aller Knoten wird durch Inorder-Durchlauf des binären Suchbaums erreicht
- Zur Erinnerung: Inorder-Durchlauf:
  - Besuche linken Unterbaum
  - Besuche Wurzel
  - Besuche rechten Unterbaum
- Beispiel:

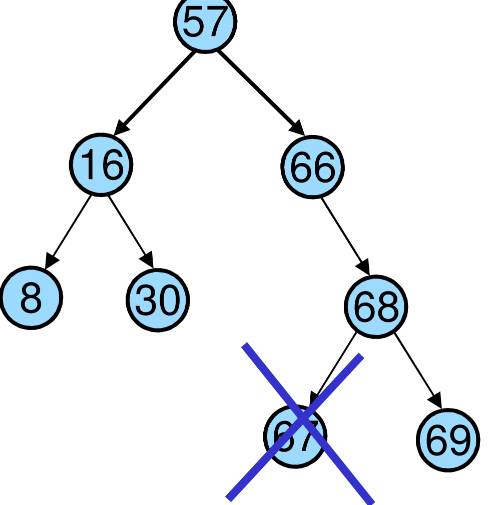


  - 8 16 30 57 66 67 68 69

Löschen eines Knoten (I)

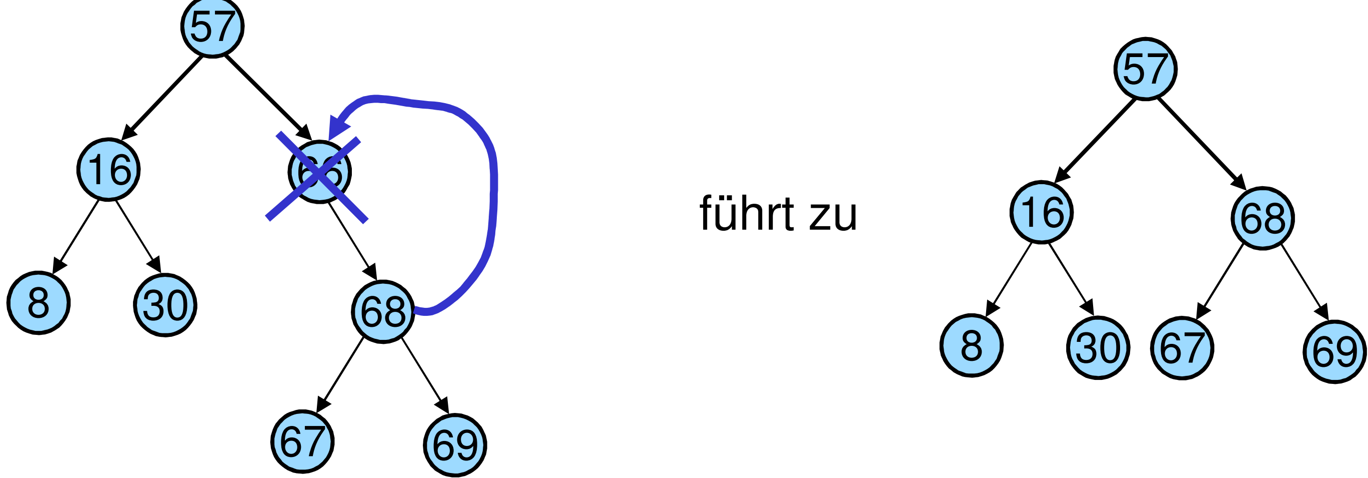
- Vorgehensweise in zwei Teilschritten:
  - Position des Knoten bestimmen (analog zum Vorgehen beim Einfügen)
  - Eigentliches Löschen des Knoten
  - Hierbei müssen drei Fälle unterschieden werden:
    (1) Zu löschender Knoten ist Blatt
    (2) Zu löschender Knoten besitzt linken oder rechten leeren Unterbaum
    (3) Zu löschender Knoten besitzt zwei nicht-leere Unterbäume

- Fall 1: Zu löschender Knoten ist Blatt
  - Blatt kann ohne weitere Maßnahmen entfernt werden
  - Beispiel: Lösche 67



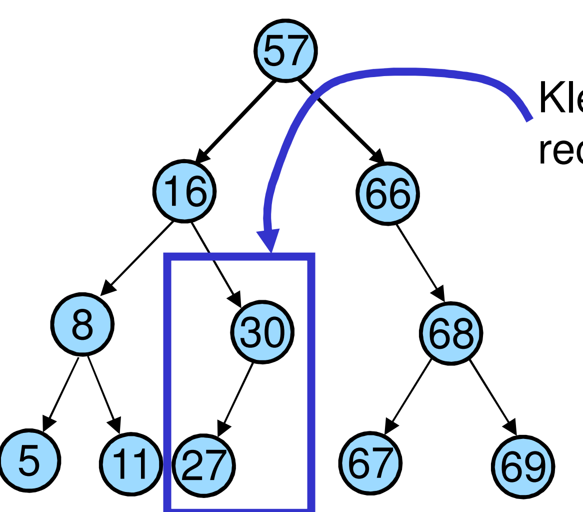
Löschen eines Knoten (II)

- Fall 2: Zu löschender Knoten besitzt linken oder rechten leeren Unterbaum
  - Zu löschender Knoten wird entfernt und durch die Wurzel des nicht-leeren Unterbaums ersetzt
  - Beispiel: Lösche 66



Löschen eines Knoten (III)

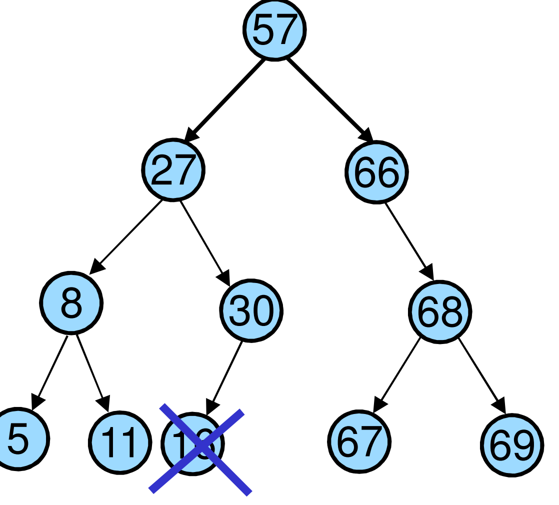
- Fall 3: Zu löschender Knoten x besitzt zwei nicht-leere Unterbäume
  - Schwierigste der drei Fälle: Wo sollen Unterbäume eingehängt werden?
  - Lösungsmöglichkeit 1:
    - Sei x‘ der Knoten mit dem kleinsten Schlüssel im rechten Unterbaum von x (Inorder-Nachfolger)
    - x‘ kann rechten Nachfolger haben, aber keinen linken
    - Vertausche die Werte von x und x‘
    - Lösche aktuellen Knoten x $\Rightarrow$ Fall 1 oder Fall 2
  - Beispiel: Lösche 16



Kleinstes Element im rechten Unterbaum ist 27.

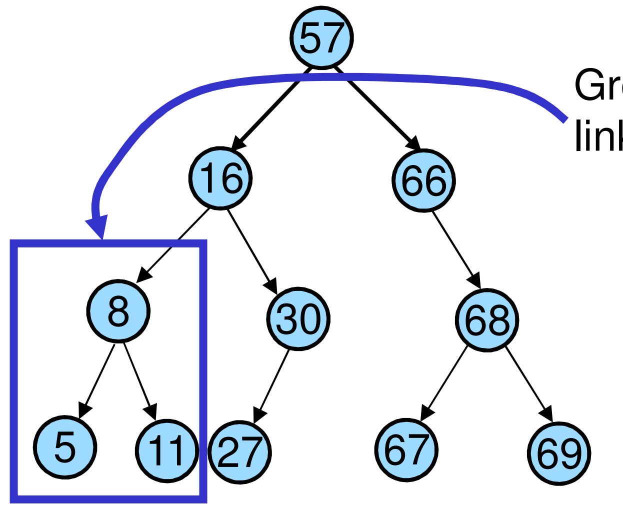
Vertauschen von 16 und 27 führt zu folgendem Baum

16 ist jetzt Blatt und kann gemäß Fall 1 gelöscht werden



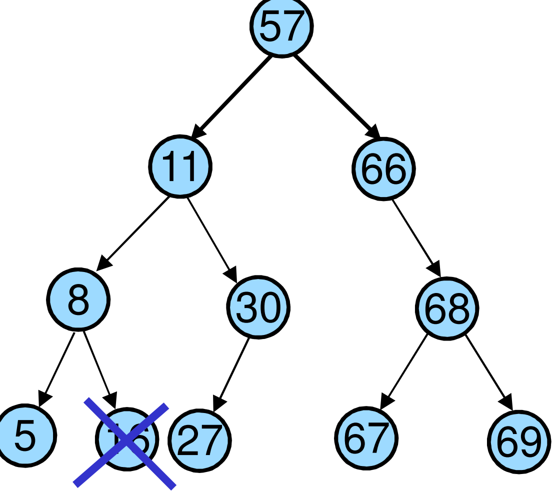
Löschen eines Knoten (IV)

- Fall 3 (Fortsetzung):
  - Lösungsmöglichkeit 2:
    - Sei x‘ der Knoten mit dem größten Schlüssel im linken Unterbaum von x (Inorder-Vorgänger)
    - x‘ kann linken Nachfolger haben, aber keinen rechten
    - Vertausche die Werte von x und x‘
    - Lösche aktuellen Knoten x $\Rightarrow$ Fall 1 oder Fall 2
  - Beispiel: Lösche 16



Größtes Element im linken Unterbaum ist 11

Vertauschen von 16 und 11 führt zu folgendem Baum



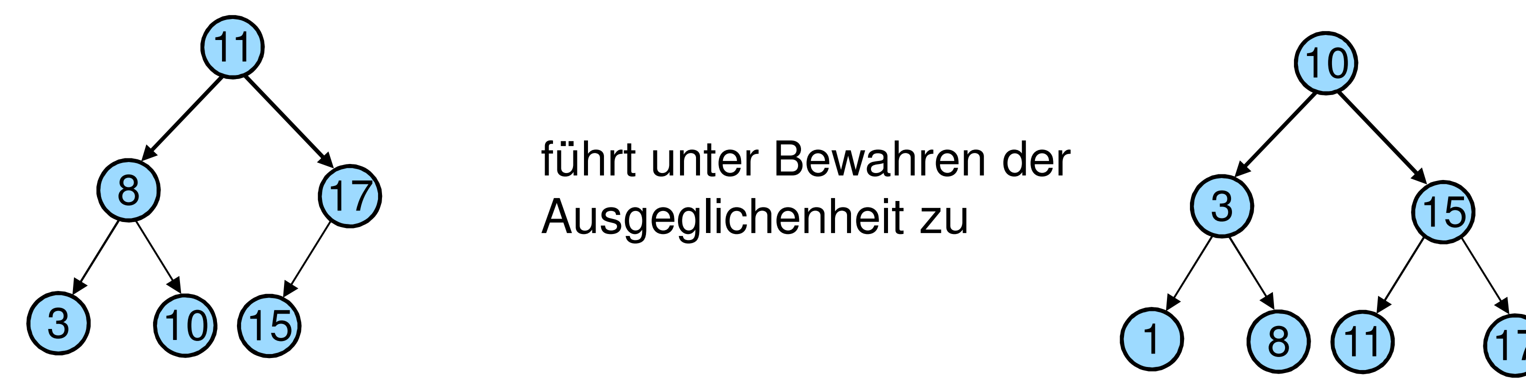
16 ist jetzt Blatt und kann gemäß Fall 1 gelöscht werden

Aufwand der Operationen

- Ausgabe aller Knoten:
  - Jeder Knoten muss einmal besucht werden $\Rightarrow$ O(n)
- Für Einfügen, Suchen, Löschen:
  - Maß für die Komplexität: Anzahl der Vergleiche
  - Aufwand ist maximal jeweils der Weg von Wurzel zu Blatt
  - Damit ist Aufwand entscheidend von der Höhe des Baums abhängig
  - Worst case: Baum ist lineare Liste $\Rightarrow$ O(n)
  - Best case: Baum ist ausgeglichen $\Rightarrow$ $O(\log_2 n)$
  - Average case: Hierfür kann man beweisen, dass nur ein mittlerer Mehraufwand von ca. 39% gegenüber dem best case notwendig ist
  - Aber: Keine Garantie hierfür, was aber in vielen Anwendungen gewünscht ist
- Idee: Bäume bei Einfügen und Löschen durch Reorganisationsoperationen so gestalten, dass sie immer (möglichst) ausgeglichen sind

Ausgeglichene binäre Suchbäume

- Reorganisationsoperationen kosten natürlich auch Aufwand
- Daher muss abgeklärt werden, ob dieser gerechtfertigt ist
- Beispiel: Einfügen von 1



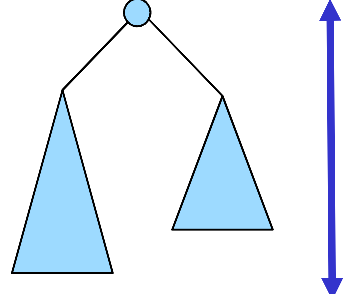
- Dies ist ein worst case-Szenario: Alle Knoten müssen verschoben werden, was Aufwand O(n) bedeutet
- Damit übersteigen die Reorganisationskosten die eigentlichen Suchkosten (O(n) gegenüber $O(\log_2 n)$)
- Fazit: Ausgeglichene Bäume sind in der Praxis nicht geeignet!

Balancierte binäre Suchbäume (I)

- Balancierte binäre Suchbäume als „fast ausgeglichene“ binäre Suchbäume
- Stellen Kompromiss zwischen ausgeglichenen binären Suchbäumen und natürlichen binären Suchbäumen dar
- Gewisse Abweichungen von der Ausgeglichenheit werden erlaubt, ohne das Zugriffsverhalten total zu zerstören
- Reorganisation darf nicht mehr global den ganzen Baum betreffen
- Transformationen dürfen nur noch lokal auf dem Pfad von der Wurzel bis zur Position der Änderung wirken
- Durch diese Einschränkung erreicht man, dass auch der Reorganisationsaufwand maximal $O(\log_2 n)$ beträgt

Balancierte binäre Suchbäume (II)

- „Fast ausgeglichen“ etwas formaler: Für alle Knoten im Baum soll gelten, dass Anzahl der Knoten im linken Unterbaum „ungefähr gleich“ Anzahl der Knoten im rechten Unterbaum entspricht
- Zwei Ansätze, um dies einzuhalten:
  - Höhenbalancierte Bäume
    - Differenz der Höhen beider Unterbäume ist beschränkt
    - Beispiele:
      - AVL-Baum (Adelson-Velsky und Landis)
      - HB-Baum



  - Gewichtsbalancierte Bäume

    - Differenz zwischen Knotenmengen beider Unterbäume muss beschränkt sein
    - Beispiel:
      - BB(α)-Baum (Bounded Balance, α Faktor für Gleichgewicht)

B_1 B_2

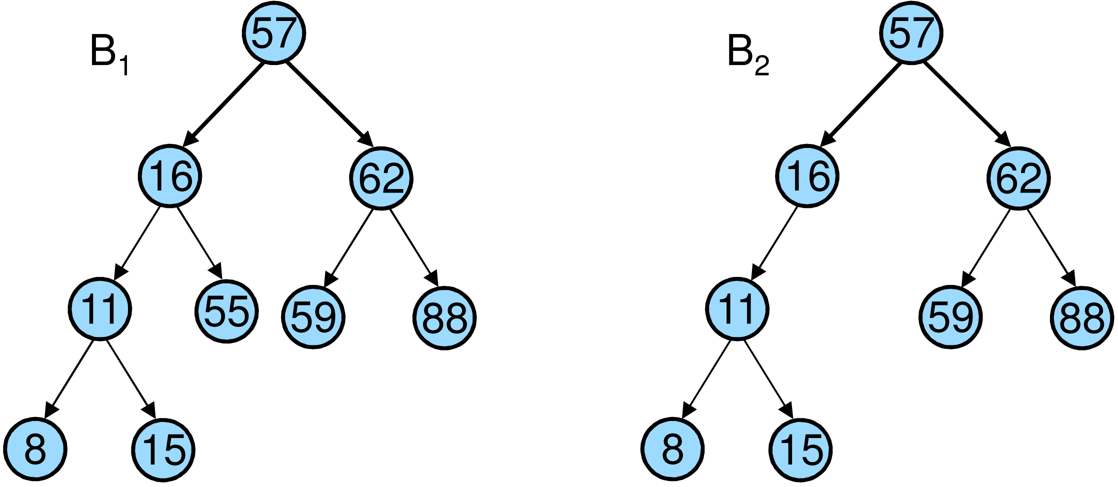
$|grad(B_1) - grad(B_2)| \leq$ Grenze

Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

AVL-Bäume

- Definition: Ein binärer Suchbaum B heißt <u>AVL-Baum</u>, genau dann wenn für alle Knoten k von B gilt: Höhendifferenz der beiden Unterbäume von k ist höchstens 1.
- Beispiele:

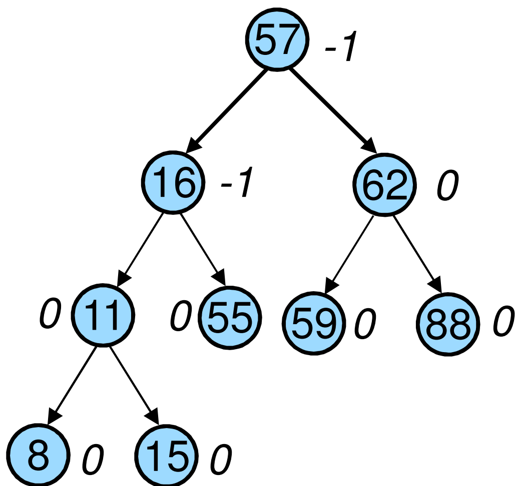


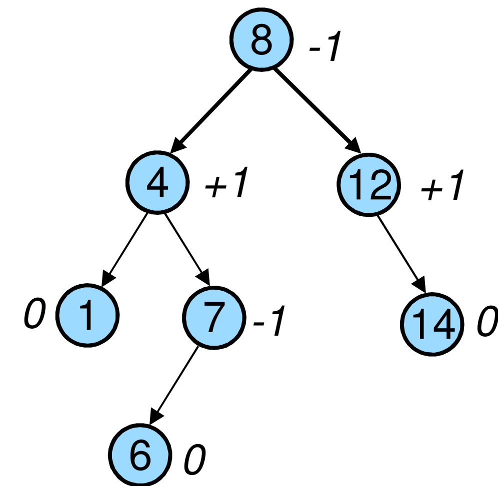
  - B_1 ist AVL-Baum
  - B_2 ist kein AVL-Baum, da Knoten 16 einen linken Unterbaum der Höhe 2 und einen rechten Unterbaum der Höhe 0 hat

Balancefaktor in AVL-Bäumen

- Definition: Sei B AVL-Baum.

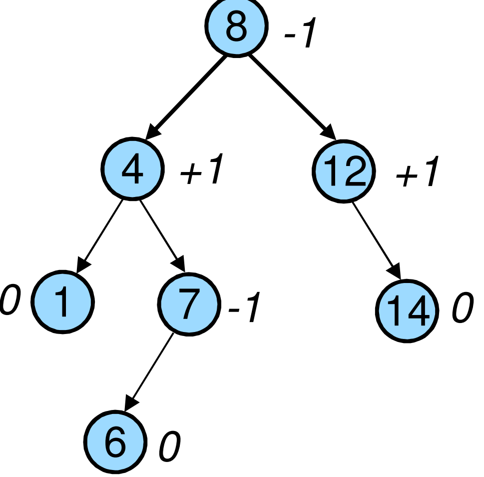
  Jedem Knoten k von B wird als Balancefaktor die Höhendifferenz zwischen linkem und rechtem Unterbaum von k zugeordnet. Mögliche Werte sind –1, 0 und 1.
- Beispiel:



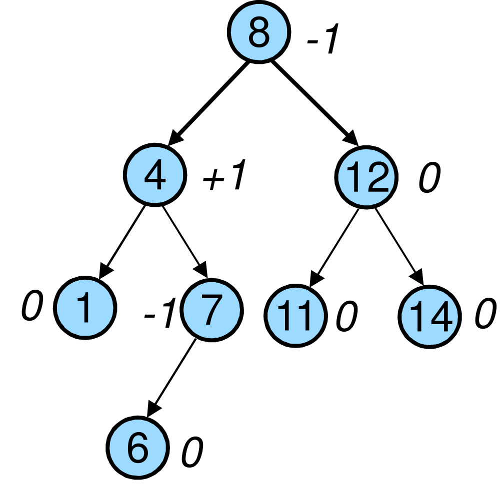


Einfügen in AVL-Bäumen: Problem

- Einfügen ist nicht so einfach wie in binären Suchbäumen
- Beispiel:

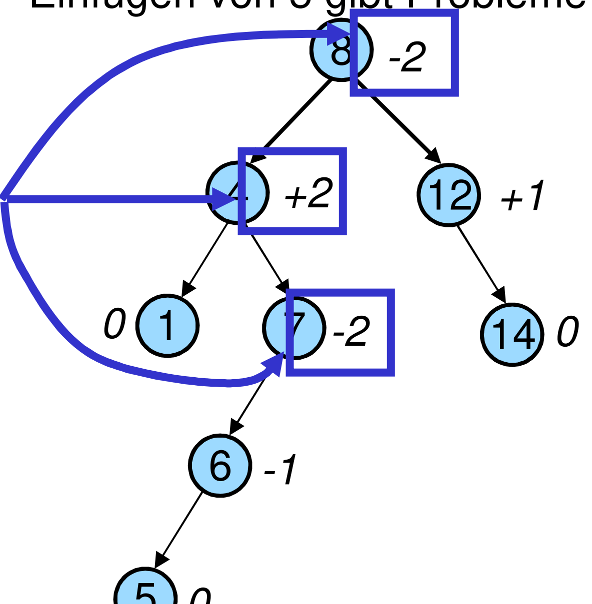


Einfügen von 11 problemlos:



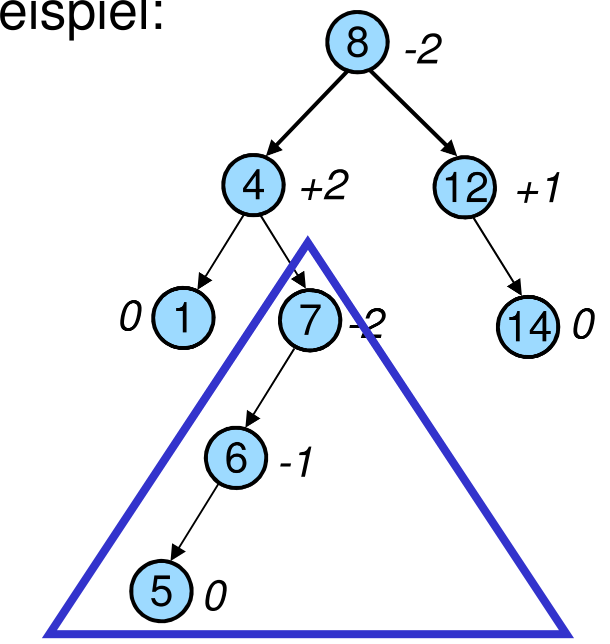
AVL-Eigenschaft verletzt

Einfügen von 5 gibt Probleme:

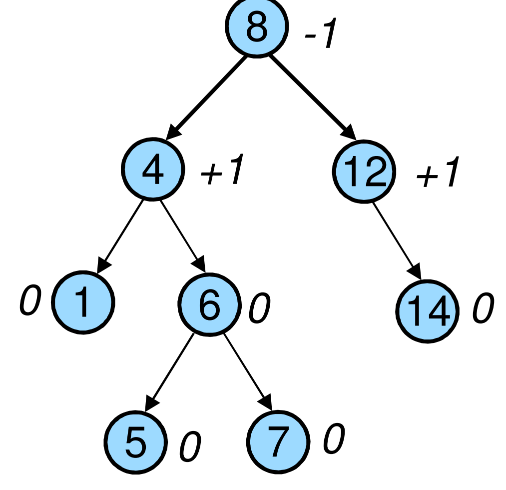


Einfügen in AVL-Bäumen: Problem

- Wie kann AVL-Eigenschaft wieder hergestellt werden?
- Beispiel:



führt zu



- Reorganisiere diesen Unterbaum
- Geschieht durch Rechtsverschiebung der Knoten
- Wird daher als Rechtsrotation (R-Rotation) bezeichnet

Einfügen in AVL-Bäumen

- Generelles Vorgehen zum Einfügen in AVL-Bäume:
  - Wie bei binären Suchbäumen wird neues Element x als Blatt eingefügt
  - Balancefaktor kann nur entlang des Weges von der Wurzel zur Einfügestelle verändert worden sein
  - Aufzusuchen ist der unterste Knoten k mit Balancefaktor 2 bzw. –2
  - Gibt es keinen solchen Knoten, ist AVL-Eigenschaft durch Einfügen nicht verloren gegangen
  - Gibt es einen solchen Knoten, dann unterscheide folgende Fälle:

    (1) Überhang im linken Unterbaum links von k

    (2) Überhang im mittleren Unterbaum links von k

    Symmetrisch dazu:

    (3) Überhang im rechten Unterbaum rechts von k

    (4) Überhang im mittleren Unterbaum rechts von k
  - Anm.: In den Fällen (1) und (2) besitzt k jeweils den Wert –2, in den Fällen (3) und (4) den Wert 2

Fall 1: Überhang im linken Unterbaum links von k

- Situation ist links skizziert, Abhilfe schafft eine Rechts-Rotation

k -2
l -1
B_1
B_2
B_3
x
R-Rotation
l 0
k 0
B_1
x
B_2
B_3

x Eingefügtes Element

Fall 2: Überhang im mittleren Unterbaum links von k

- Links-Rechts-Rotation um Knoten l und dann um Knoten m

Löschen in AVL-Bäumen

- Geschieht zunächst analog zum Löschen in binären Suchbäumen (zu löschendes Element wird durch seinen Inorder-Nachfolger ersetzt)
- Beginnend an der untersten veränderten Stelle muss der Baum nach der Löschoperation längs des Pfades zur Wurzel ausgeglichen werden
- Hierbei sind im Gegensatz zum Einfügen u.U. mehrere Rotationen notwendig
- Folgende Fälle sind für einen Knoten k zu unterscheiden, dessen linker Unterbaum durch Löschen in der Höhe um 1 verringert wurde:
  - (1) Balancefaktor k = -1

    Setze Balancefaktor k = 0 und mache mit dem Vorgänger von k weiter
  - (2) Balancefaktor k = 0

    Setze Balancefaktor k = +1 und beende das Ausgleichen
  - (3) Balancefaktor k = +1 und l sei der rechte Nachfolger von k, unterscheide
    - a) Balancefaktor von l = +1
    - b) Balancefaktor von l = 0
    - c) Balancefaktor von l = -1
- Anm.: Symmetrisch für rechten Unterbaum

Löschen in AVL-Baum: Fall 3a

- Balancefaktor von l ist +1 ⇒ Links-Rotation von k
- Gesamthöhe des Unterbaums unter k bzw. l um 1 erniedrigt
- Daher anschließend mit dem Vorgänger von l weitermachen

k +2
l +1
B_1
B_2
B_3
Gelöschtes Element
L-Rotation von k
l 0
k 0
B_1
B_2
B_3

Löschen in AVL-Baum: Fall 3b

- Balancefaktor von l ist 0 $\Rightarrow$ Links-Rotation von k
- Gesamthöhe des Unterbaums unter k bzw. l bleibt unverändert
- Daher kann das Ausgleichen beendet werden

k +2
l 0
B_1
B_2
B_3
Gelöschtes Element

L-Rotation von k

l -1
k +1
B_1
B_2
B_3

Löschen in AVL-Baum: Fall 3c

- Balancefaktor von l ist –1, m sei linker Nachfolger von l mit Balancefaktor $q \in \{-1, 0, +1\}$, d.h. in B_2 ($q = -1$) oder B_3 ($q = 1$) fehlt eine Stufe oder nicht ($q = 0$)
- RL-Rotation, zunächst R-Rotation von l, dann L-Rotation von m
- Anschließend beim Vorgänger von m weitermachen

k +2
l -1
m q
B_1 B_2 B_3 B_4
? ?
Gelöschtes Element

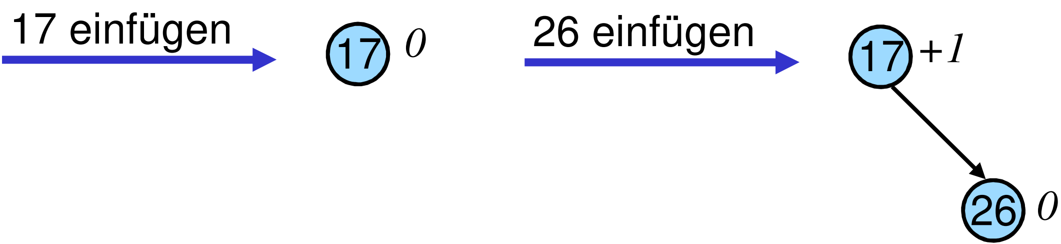
RL-Rotation von l und m

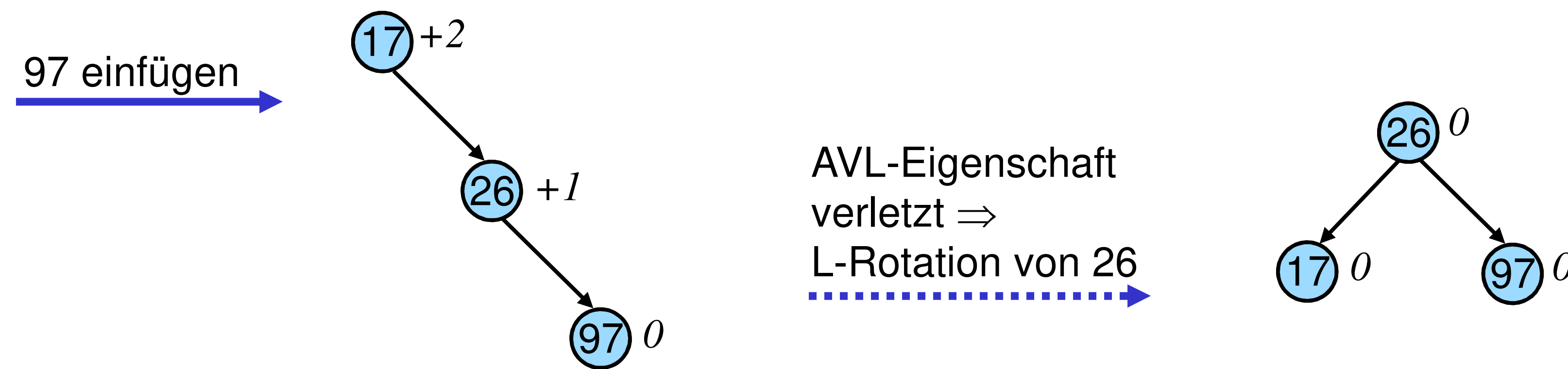
m 0
k 0/-1
l 0/+1
B_1 B_2 B_3 B_4
? ?

? Mindestens eins dieser Elemente vorhanden

Beispiel (I)

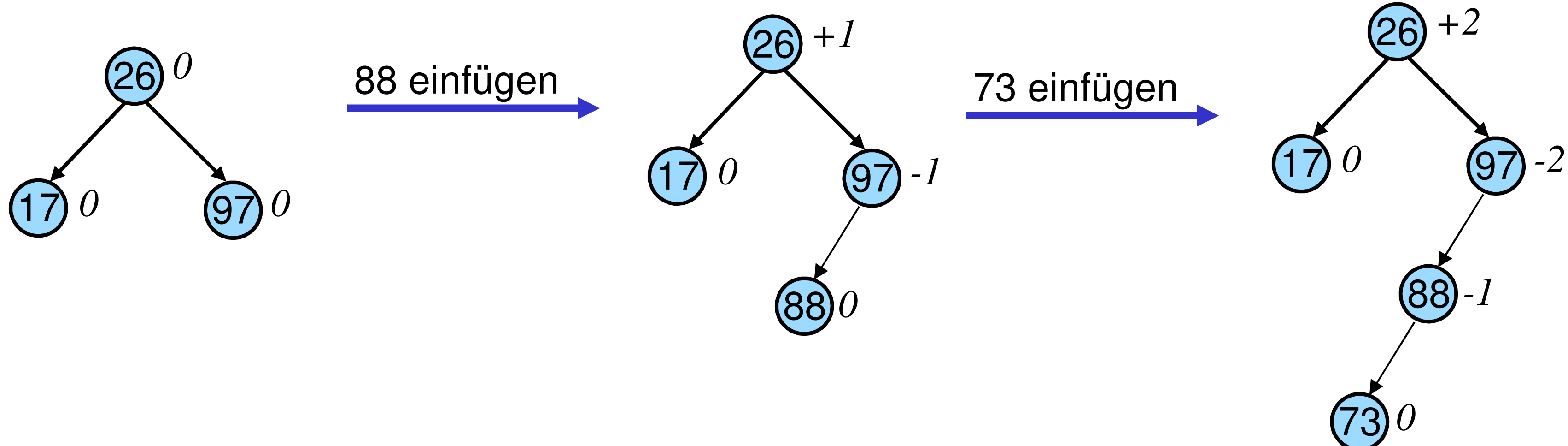
- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

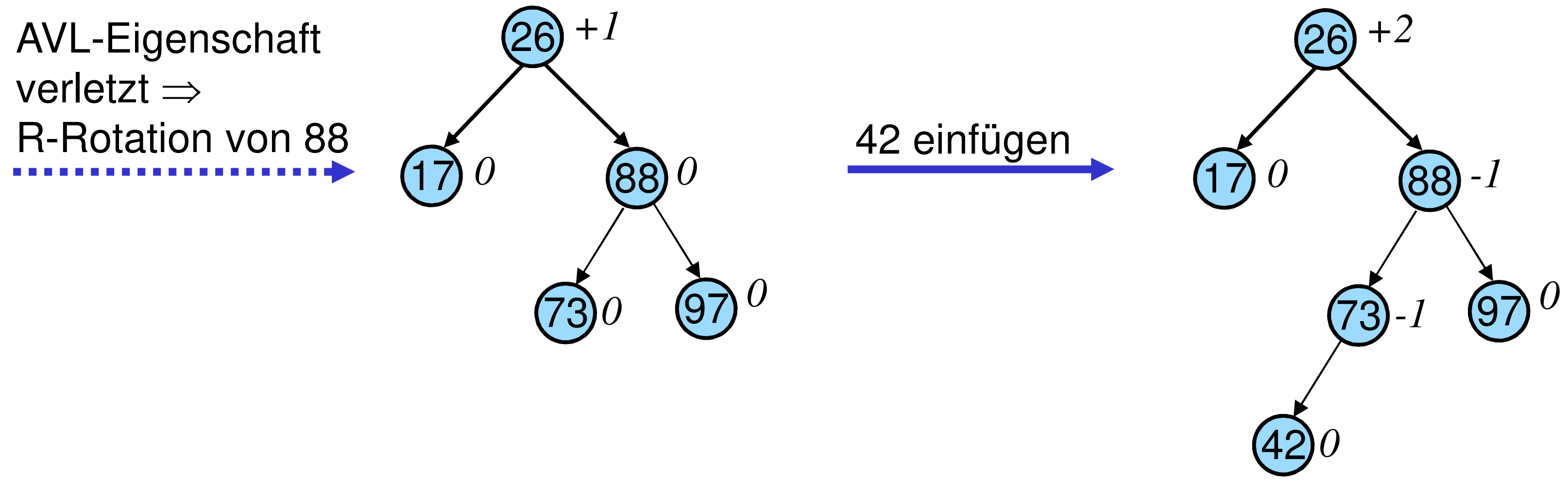




Beispiel (II)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69





Beispiel (III)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

26 +2
17 0 88 -1
73 -1 97 0
42 0

AVL-Eigenschaft verletzt ⇒ RL-Rotation von 73

73 0
26 0 88 +1
17 0 42 0 97 0

15 einfügen

73 -1
26 -1 88 +1
17 -1 42 0 97 0
15 0

24 einfügen

73 -1
26 -1 88 +1
17 0 42 0 97 0
15 0 24 0

Beispiel (IV)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

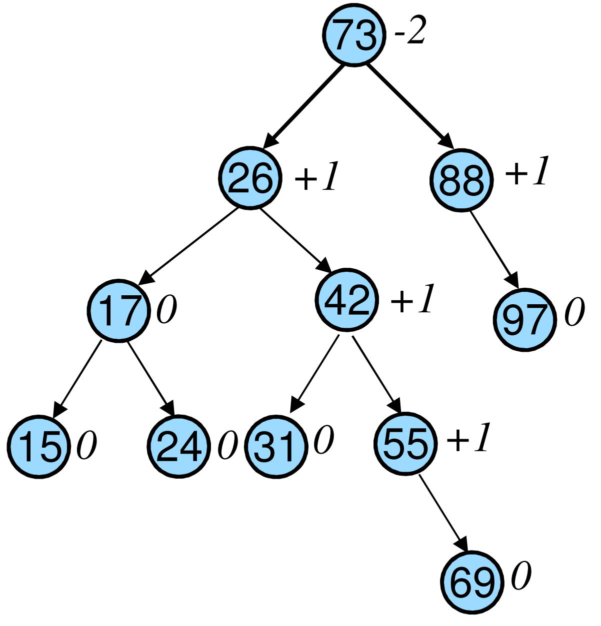
31 einfügen

55 einfügen

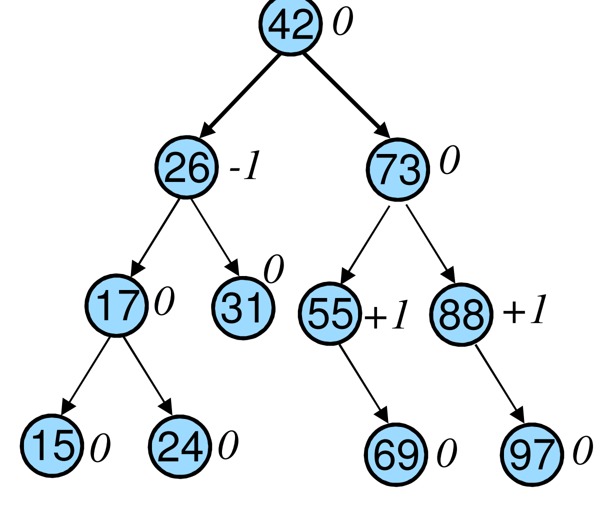
69 einfügen

Beispiel (V)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

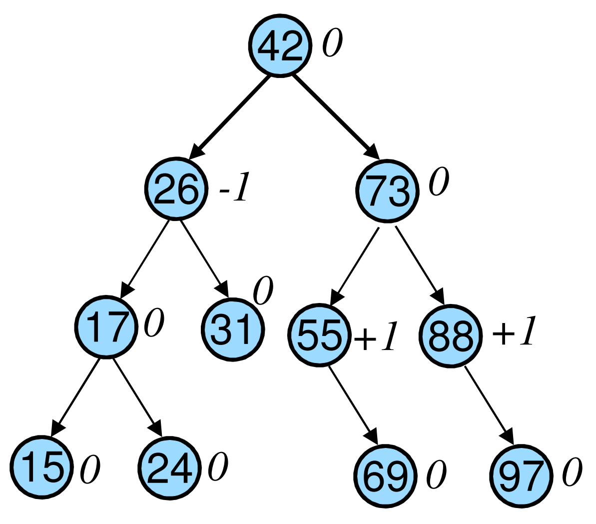


AVL-Eigenschaft verletzt ⇒ LR-Rotation von 42

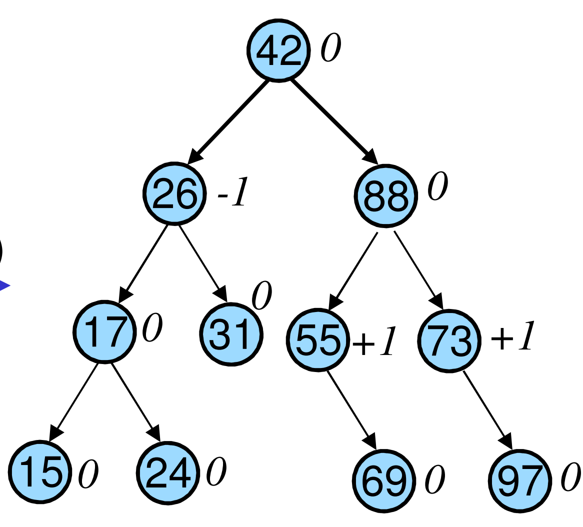


Beispiel (VI)

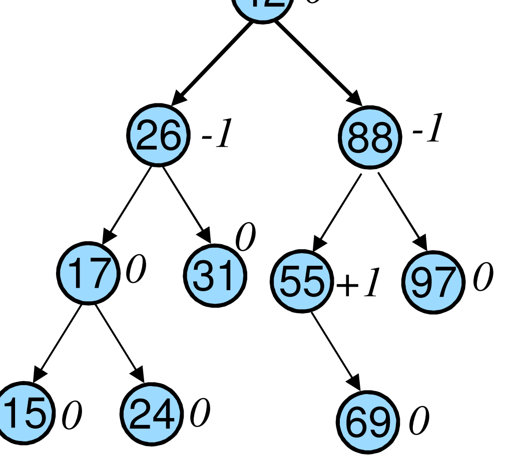
- Lösche Knoten 73



Vertausche zu lö-schenden Knoten (73) mit Inorder-Nachfolger (88)



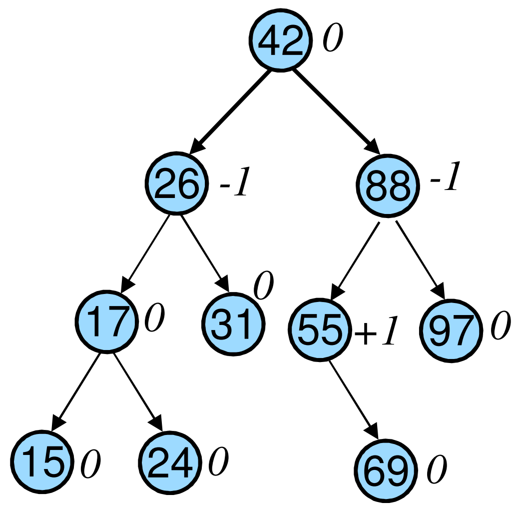
Knoten (73) kann an neuer Position durch Umhängen gelöscht werden



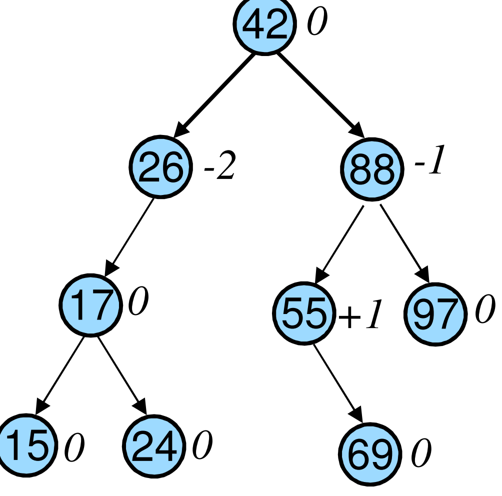
AVL-Eigenschaft nicht verletzt
$\Rightarrow$ Operation beendet

Beispiel (VII)

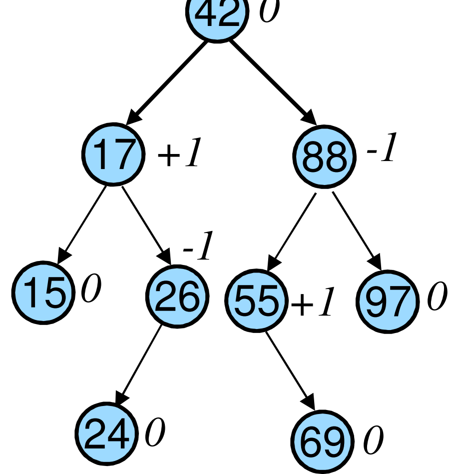
- Lösche Knoten 31



Zu löschender Knoten (31) kann als Blatt direkt gelöscht werden



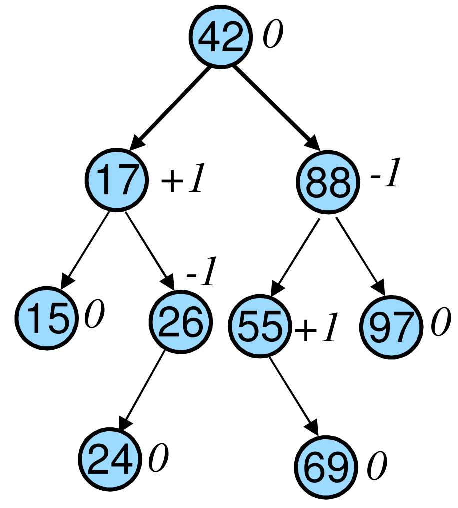
Knoten 26 verletzt AVL-Eigenschaft (symm. zu Fall 3b)
$\Rightarrow$ R-Rotation



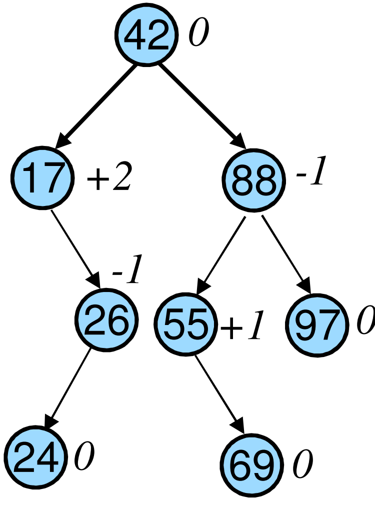
AVL-Eigenschaft wieder hergestellt
$\Rightarrow$ Operation beendet

Beispiel (VII)

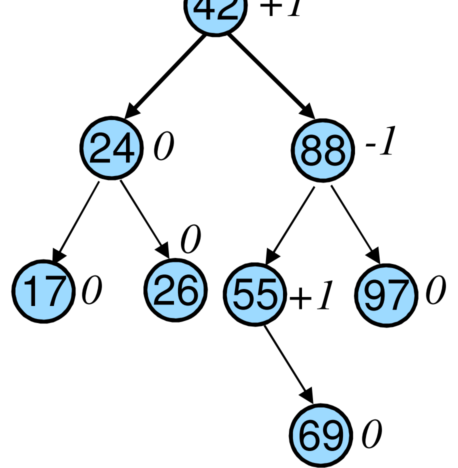
- Lösche Knoten 15



Zu löschender Knoten (15) kann als Blatt direkt gelöscht werden



Knoten 17 verletzt AVL-Eigenschaft (Fall 3c)
⇒ RL-Rotation



AVL-Eigenschaft wieder hergestellt
⇒ Operation beendet

- Einfügen und Löschen kosten im worst case bzw. average case soviel wie die maximale bzw. mittlere Höhe eines AVL-Baums mit n Knoten beträgt
- Folgende Resultate wurden empirisch ermittelt:
  - Mittlere Höhe eines AVL-Baums mit n Knoten ist ca. $\log_2 n + 0.25$ (also fast genauso gut wie vollständig ausgeglichene Bäume)
  - Wahrscheinlichkeit einer Rotation beim Einfügen: ca. 50%
  - Wahrscheinlichkeit einer Rotation beim Löschen: ca. 20%
- Weiterhin kann man zeigen:
  - Für die Höhe h_b eines AVL-Baums mit n Knoten gilt:

$$\lfloor \log_2 n \rfloor + 1 < h_b < 1.44 \cdot \log_2 (n+1)$$

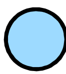
  - Maximale Suchzeit eines AVL-Baum damit höchstens ca. 44% höher als bei einem vollständig ausgeglichenen Baum
  - Mittlere Suchzeit aber erheblich günstiger, nie schlechter als O(log n)

Fibonacci-Bäume (I)

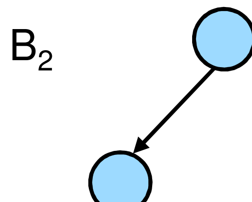
- Obere Schranke lässt sich durch eine als Fibonacci-Bäume bezeichnete Unterklasse herleiten
- Aufbau von Fibonacci-Bäumen:
  - Leerer Baum ist Fibonacci-Baum der Höhe 0
  - Einzelner Knoten ist Fibonacci-Baum der Höhe 1
  - Sind B_{h-1} und B_{h-2} Fibonacci-Bäume der Höhe h-1 bzw. h-2, dann ist der Baum mit einer neuen Wurzel und B_{h-1} und B_{h-2} als linkem und rechtem Unterbaum Fibonacci-Baum der Höhe h
  - Kein anderer Baum ist Fibonacci-Baum
- Fibonacci-Bäume:

B_0

B_1



B_2



Fibonacci-Bäume (II)

B_3

B_4

B_5

Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

Gewichtsbalancierte Bäume (I)

- Gewichtsbalancierte Bäume (Synonym: BB-Bäume (Bounded Balance)): Zulässige Abweichung der Struktur vom ausgeglichenen Binärbaum wird als Differenz zwischen Anzahl der Knoten im rechten und linken Unterbaum festgelegt
- Definition:

  Sei B ein binärer Suchbaum mit n Knoten, sei B_l linker Unterbaum mit n_l Knoten.

  Dann heißt $p(B) = \frac{(n_l + 1)}{(n + 1)}$ Wurzelbalance von B.

- Definition:

  Binärbaum B heißt gewichtsbalanciert (Schreibweise: BB(α)), wenn für jeden Unterbaum B‘ von B $\alpha \leq p(B') \leq 1 - \alpha$ gilt.

Gewichtsbalancierte Bäume (II)

- Wahl des Parameters α bestimmt wie streng die Gewichtsbalancierung ausgelegt wird:
  - $\alpha = ½$: Es werden nur vollständig ausgeglichene Binärbäume akzeptiert
  - $\alpha < ½$: Kriterium wird zunehmend gelockert
- Je kleiner α gewählt wird, desto weniger Reorganisationsoperationen sind notwendig, umso degenerierter kann der Baum aber auch werden
- Im Extremfall $\alpha = 0$ kann der Baum zur Liste werden
- Praktisch realistischer Wert: $\alpha = 0{,}3$
- Für Reorganisation zum Einhalten der Balance-Eigenschaften kommen die selben Rotationstypen wie beim AVL-Baum zum Einsatz
- Kosten für Suche und Aktualisierung bei $\alpha = 0{,}3$: $O(\log_2 n)$
- Damit in Effizienz mit AVL-Bäumen vergleichbar

Vergleich

	Feld	Liste	Binärer Suchbaum	AVL-Baum
Knoten Suchen	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Alle Knoten sortiert ausgeben	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Knoten einfügen	$O(\log_2 n) + O(n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Knoten löschen	$O(\log_2 n) + O(n)$	$O(n)$	$O(n)$	$O(\log_2 n)$

Zusammenfassung

- Suchen in Feldern und Listen:
  - Lineare Suche
  - Binäre Suche
  - Interpolationssuche
- Binäre Suchbäume:
  - Definition
  - Operationen: Einfügen, Suchen, Löschen, Ausgeben
  - Problem: Degenerierung zur Liste
- AVL-Bäume:
  - Höhenbalancierter Baum, Balancefaktoren
  - Einfügen, Löschen
  - Rotationen
- Gewichtsbalancierte Bäume:
  - Anzahl der Knoten in linken und rechtem Unterbaum darf sich nicht „allzu stark" unterscheiden

Übungen (I)

- **Aufgabe 1**

  Gegeben sei die folgenden Zahlenfolge:

  23, 66, 42, 11, 5, 69, 77, 55, 16.

  a) Bauen Sie schrittweise einen binären Suchbaum auf!
  b) Löschen Sie aus dem binären Suchbaum nacheinander die Knoten 77, 42 und 23.
  c) Bauen Sie schrittweise einen AVL-Baum auf!
  d) Löschen Sie aus dem AVL-Baum nacheinander die Knoten 5, 42 und 77.

- **Aufgabe 2**

  Zeichnen Sie die zu den in der Vorlesung vorgestellten Rotationen beim Einfügen und Löschen in einem AVL-Baum die symmetrischen Fälle.

- **Aufgabe 3**

  a) Wie viele Knoten hat ein Fibonacci-Baum der Höhe n?
  b) Zeichnen Sie alle verschiedenen Fibonacci-Bäume der Höhe 4!
  c) Wie viele verschiedene Fibonacci-Bäume der Höhe n gibt es?

Übungen (II)

- **Aufgabe 4**

	wahr	falsch
Einfügen im binären Suchbaum geschieht immer im linken Teilbaum.		
Jeder Heap ist ein binärer Suchbaum.		
Suchen in einem Feld hat konstanten Aufwand.		
Bei gewichtsbalancierten Bäumen ist der Faktor α möglichst klein zu wählen.		
AVL-Bäume sind höhenbalancierte Suchbäume.		

Übungen (III)

- **Aufgabe 5**
  In einem Binärbaum seien die Zahlen zwischen 1 und 1000 gespeichert.
  Nun soll die Zahl 363 gesucht werden.
  Welche der folgenden Sequenzen kann nicht die überprüfte Knotenfolge sein?
  a) 2, 252, 401, 398, 330, 344, 397, 363
  b) 924, 220, 911, 244, 898, 258, 362, 363
  c) 925, 202, 911, 240, 912, 245, 363
  d) 2, 399, 387, 219, 266, 382, 381, 278, 363
  e) 935, 278, 347, 621, 299, 392, 358, 363

Übungen (IV)

- **Aufgabe 6**
  Karl-Heinz glaubt, er habe eine bemerkenswerte Eigenschaft binärer Suchbäume entdeckt. Nehmen Sie an, die Suche in einem binären Suchbaum nach dem Schlüssel k würde in einem Blatt enden. Betrachten Sie drei Mengen: A, die Menge der Schlüssel, die links vom Suchpfad liegen, B, die Menge der Schlüssel auf dem Suchpfad, und C, die Menge der Schlüssel rechts vom Suchpfad. Karl-Heinz behauptet, dass jedes Tripel von Schlüsseln $a \in A$, $b \in B$ und $c \in C$ die Ungleichung $a <= b <= c$ erfüllen muss.
  Geben Sie ein möglichst kleines Gegenbeispiel zu der Behauptung an!

Algorithmen und Datenstrukturen

Teil 11:
Suchen (II) – B-Bäume

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

- Motivation und Definition
- Operationen auf B-Bäumen
- Beispiel
- B*-Bäume

Motivation (I)

- Bisherige Suchstrukturen Hauptspeicher-basiert
- Praxisanforderung:
  - Effiziente Speicherung und Reorganisation sehr großer Datenmengen
  - Datenvolumen größer als Hauptspeicher
- Ziele:
  - Reduzierung der Plattenspeicherzugriffe (Operationen etwa um Faktor 10^6 langsamer als Hauptspeicherzugriffe)
  - Plattenplatz „möglichst gut“ ausnutzen (Verhältnis tatsächlich genutzter Platz zu reserviertem Platz auf der Platte sollte über 50% liegen)
- Zur Technologie:
  - Tausch zwischen Haupt- und Plattenspeicher:
    - Nicht einzelne Datensätze
    - Blöcke oder Seiten
  - Typische Größen: 2 kB, 4 kB, 8 kB, 16 kB oder 32 kB

Motivation (II)

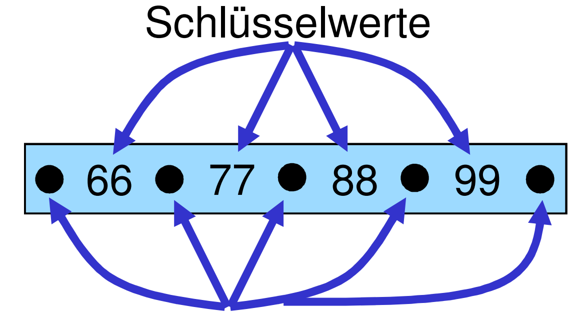
- Erste Idee:
  - Direkte Übertragung binärer Suchbäume
  - Verweise auf linken und rechten Nachfolger zeigen nicht auf Hauptspeicheradressen, sondern auf Blöcke des Plattenspeichers
  - Schlechte Ausnutzung des Plattenspeichers:
    - Blockgröße 4kB (4096 Byte)
    - Platzbedarf Zeiger (8 Byte)
    - Platzbedarf Schlüsselwert (32 Byte)
    - Platzbedarf restlicher Datensatzinhalt (200 Byte)
    - Jeder Block wäre mit 248/4096 ≈ 6% belegt
    - Vorgehensweise viel zu ineffizient
- Problem: Schlechter Füllungsgrad der Blöcke
- Idee zur Verbesserung: Mehrere Datensätze und Verweise in einem Knoten des Baums (d.h. in einem Block) speichern
- Strukturen werden als Mehrwege- oder Vielwege-Suchbäume bezeichnet

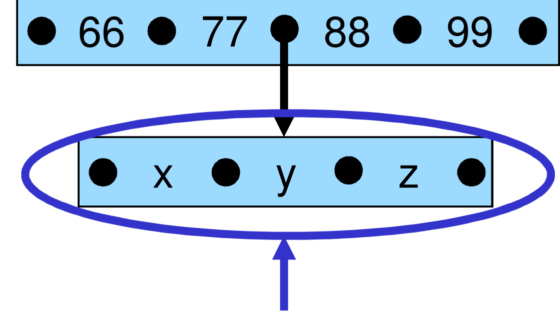
Motivation (III)

- Bekanntester Vertreter dieser Kategorie sind B-Bäume:
  - Verwalten in einem Knoten mehrere Datensätze und Verweise
  - Baumhöhe ist ausgeglichen, d.h. alle Pfade von der Wurzel zu einem Blatt sind gleich lang
  - Füllungsgrad eines Knoten ist ≥ 50%
  - Kein innerer Knoten hat leeren Unterbaum
  - Verzweigungsgrad kann variieren
- B-Baum wäre vollständig ausgeglichen, wenn
  - Alle Wege von der Wurzel zu den Blättern gleich lang
  - Jeder Knoten hat gleich viele Einträge
  - Reorganisationsaufwand hierfür aber viel zu groß
- B-Baum-Kriterium: Jeder Knoten (außer der Wurzel) enthält zwischen m und 2m Schlüsselwerte
- Anmerkung: B steht für balanciert und nicht für binär!

Definition

- Ein Baum heißt B-Baum (der Ordnung m), g.d.w.
  (1) Jeder Knoten außer der Wurzel enthält mindestens m Datensätze
  (2) Jeder Knoten enthält höchstens 2m Datensätze
  (3) Knoten mit k Datensätzen, der kein Blatt ist, besitzt genau k+1 Nachfolger
  (4) Alle Blätter besitzen das gleiche Niveau, d.h. sie stehen auf einer Ebene
  (5) Sind S_1, S_2, ..., S_k, mit $m \leq k \leq 2m$ die Schlüssel eines Knotens x, dann sind alle Schlüssel des ersten (d.h. linkesten) Nachfolgers von x kleiner als S_1, alle Schlüssel des (k+1)-ten Nachfolger größer als S_k und alle Schlüssel des i-ten Nachfolgers mit $1 < i < k+1$ größer als S_{i-1} und kleiner als S_i.
- Struktur einer Knotens:





Berechnung der Ordnung

- Anzahl x der Einträge pro Block kann wie folgt berechnet werden:
  - Blockgröße b
  - Platzbedarf Zeiger z
  - Platzbedarf Schlüsselwert s
  - Platzbedarf restlicher Datensatzinhalt d
  - Es gilt: $x \cdot (s+d) + (x+1) \cdot z = b$
  - $\left\lfloor \frac{b-z}{s+d+z} \right\rfloor$ Einträge passen in einen Knoten,

    wobei $\lfloor\ \rfloor$ nächst kleinere ganze Zahl berechnet (Floor-Funktion)
- Beispiel:
  - Im einleitenden Beispiel ergeben sich 17 Werte
  - D.h. B-Baum der Ordnung 8 ist zu wählen

Übersicht

- Motivation und Definition
- Operationen auf B-Bäumen
- Beispiel
- B*-Bäume

Suchen

- Suchen eines Elements:
  - Einstieg über Wurzel
  - Schlüsselwerte eines Knotens zerlegen Wertebereich in Intervalle
  - Gehe in Nachfolger, dessen Verweis im entsprechenden Intervall liegt
  - Fahre fort, bis Schlüsselwert gefunden oder Blatt erreicht
- Beispiel: Suche Element 63

Einfügen (I)

- Entsprechende Blattseite suchen
- Blattseite hat < 2m Einträge ⇒ Neuen Wert einfügen
- Beispiel: Einfügen von 43

30 | 40

8 | 11 | 24 | 27 — 31 | 33 | 35 | 37 — 42 | 44

30 | 40

8 | 11 | 24 | 27 — 31 | 33 | 35 | 37 — 42 | 43 | 44

Einfügen (II)

- Blattseite hat 2m Einträge $\Rightarrow$ Überlauf des Knotens
- Behandlung Überlauf:
  - Erzeugen eines neuen Knotens unter Verwendung des Vorgängers
  - Erste m Einträge verbleiben im Originalknoten
  - Letzte m Einträge kommen in neuen Knoten
  - Mittleren Eintrag in Vorgängerknoten eintragen
- Beispiel: Einfügen von 36

30 | 40

8 | 11 | 24 | 27 | 31 | 33 | 35 | 37 | 42 | 44

Knotenüberlauf

Mittleren Eintrag in Vorgängerknoten

30 | 35 | 40

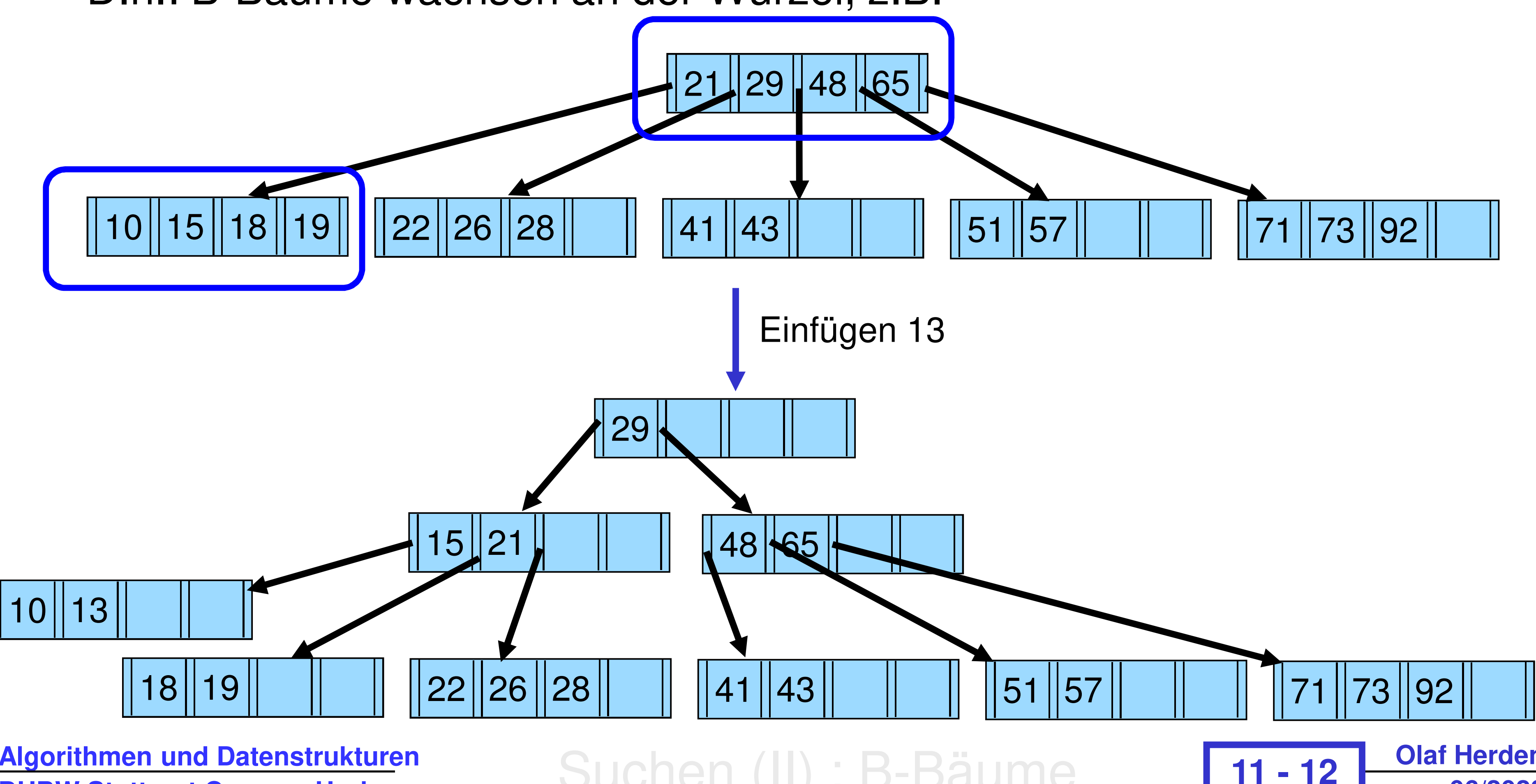
31 | 33 | 36 | 37

Erste m Einträge

Letzte m Einträge

Einfügen (III)

- Bei dieser Vorgehensweise kann auch der Vorgängerknoten überlaufen
- Knotenaufspalten rekursiv weiter nach oben fortgesetzen, evtl. bis zur Wurzel
- D.h.: B-Bäume wachsen an der Wurzel, z.B.



Löschen (I)

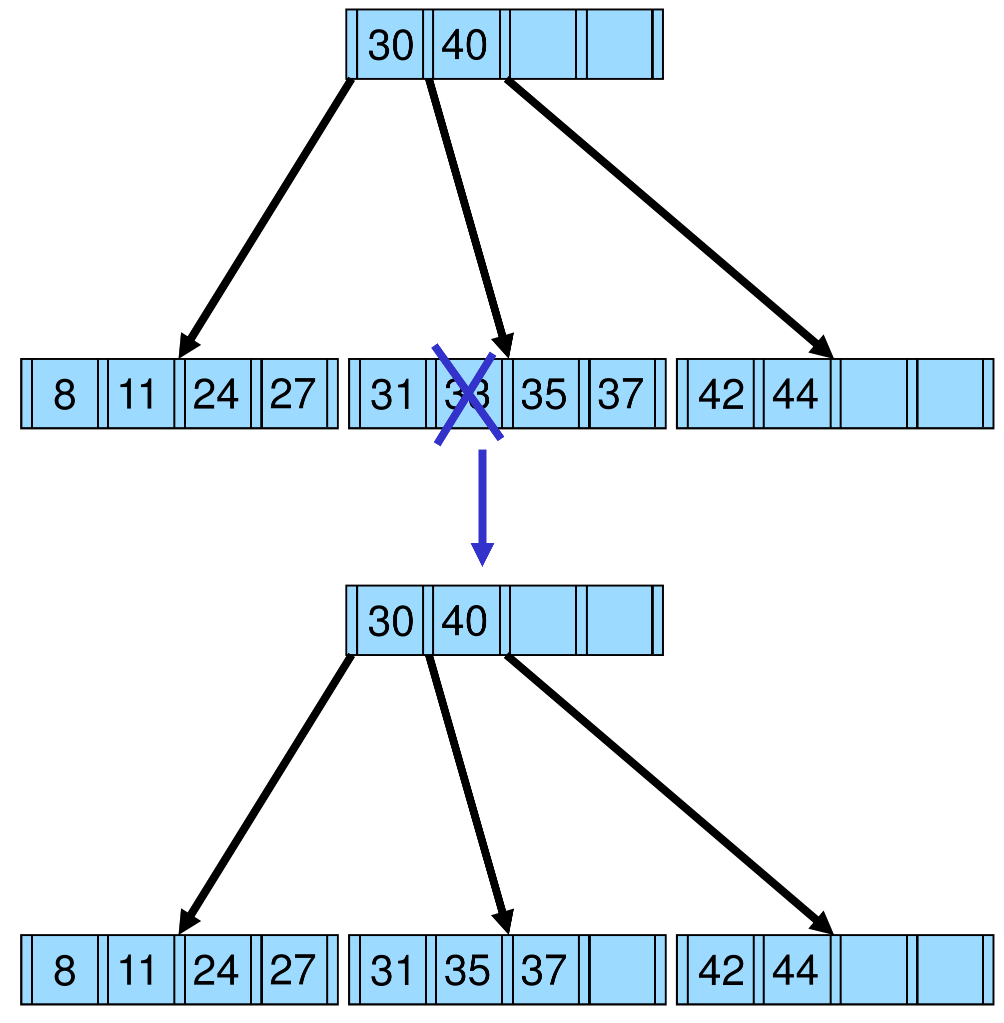
- Gegenteiliges Problem zum Einfügen kann auftreten: Unterlauf, wenn Knoten nur m Einträge besitzt
- Vorgehen:
  - Entsprechenden Knoten k suchen
  - Ist k Blatt, dann Wert löschen, evtl. Unterlauf korrigieren
  - Ist k nicht auf Blattebene, dann analog zu binärem Suchbaum k mit Inorder-Nachfolger tauschen, evtl. Unterlauf korrigieren
- Behandlung Unterlauf:

  (1) Besitzt Nachbarknoten mindestens m+1 Schlüssel, wird unter Beteiligung des Vorgängers ein Schlüssel zur Seite hinzugefügt und ein Schlüssel beim Nachbarn entfernt

  (2) Besitzen der/die Nachbarn nur m Schlüssel, wird ein Nachbarknoten mit aktuellem Knoten unter Hinzunahme des entsprechenden Schlüssels aus dem Vorgänger verschmolzen und beim Vorgänger wird rekursiv geprüft, ob dieser noch mindestens m Schlüssel besitzt, evtl. setzt sich diese Rekursion bis zur Wurzel fort

Löschen (II)

- Beispiel: Löschen ohne Unterlauf
  - Löschen von Element 33



Löschen (III)

- Beispiel: Löschen mit Unterlauf, Nachbar hat mehr als m Einträge
  - Löschen von Element 44

30 | 40

8 | 11 | 24 | 27 — 31 | 33 | 35 | 37 — 42 | 44

30 | 37

8 | 11 | 24 | 27 — 31 | 33 | 35 — 40 | 42

Unter Beteiligung des Vorgängers wird Knoten zur Löschstelle hinzugefügt

Löschen (IV)

- Beispiel: Löschen mit Unterlauf, Nachbar hat nur m Einträge
  - Löschen von Element 33

30 | 40

8 | 11 — 31 | ~~33~~ — 42 | 44

40

8 | 11 | 30 | 31 — 42 | 44

Evtl. Unterlauf beim Vorgänger beheben

Unter Beteiligung des Vorgängers wird Löschstelle mit Nachbarknoten verschmolzen

Löschen (V)

- Beispiel (Fortsetzung):
  - Löschen von Element 33; Verschmelzen mit rechtem Nachbarknoten (Alternativ möglich)

30 | 40

8 | 11 — 31 | 33 (gestrichen) — 42 | 44

30

8 | 11 — 31 | 40 | 42 | 44

Evtl. Unterlauf beim Vorgänger beheben

Unter Beteiligung des Vorgängers wird Löschstelle mit Nachbarknoten verschmolzen

Aufwand der Operationen

- Komplexität beim Einfügen, Suchen und Löschen in einem B-Baum verlangt immer $O(\log_m(n))$ Operationen
- Entspricht Höhe des Baumes
- Ein paar konkrete Werte:
  - Größe eines Knotens sollte ca. der Seitengröße des Hintergrundspeichers entsprechen
  - B-Baum der Ordnung 8 (aus letztem Beispiel)

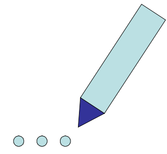
Datensätze n	Baumhöhe = Seitenzugriffe ($\log_8(n)$)
1.000	4
10.000	5
100.000	6
1.000.000	7
2.000.000	7
5.000.000	8
10.000.000	8

Übersicht

- Motivation und Definition
- Operationen auf B-Bäumen
- Beispiel
- B*-Bäume

Beispiel

- Bauen Sie einen B-Baum der Ordnung 2 mit den Werten 10, 11, 22, 17, 12, 13, 27, 25, 18, 15, 16, 19, 26 auf!
- Löschen Sie anschließend die Werte 25, 15, 13, und 22 aus dem B-Baum!



Übersicht

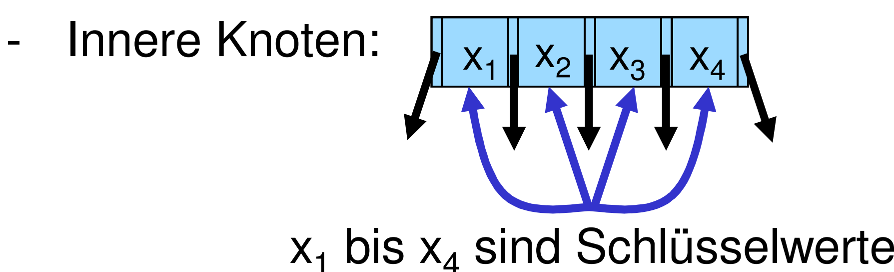
- Motivation und Definition
- Operationen auf B-Bäumen
- Beispiel
- B*-Bäume

B*-Bäume (I)

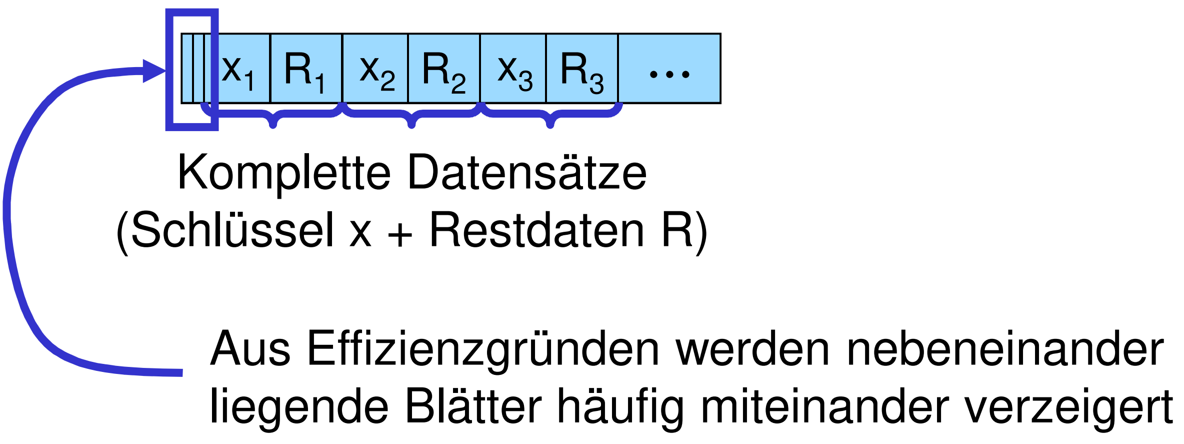
- Suchen in B-Bäumen: Durchlaufen aller Ebenen
- Pro Ebene Übertragung eines Blocks zwischen Haupt- und Plattenspeicher
- Wichtigstes Optimierungsziel: Reduzierung Anzahl der Übertragungen
- Hierzu: Höhe des Baums reduzieren, d.h. Verzweigungsgrad (und damit die Ordnung m) erhöhen
- B*-Bäume: Innere Knoten speichern Schlüsselwerte ohne ihre Restdaten
- Komplette Datensätze (Schlüssel + Restdaten) stehen nur noch in den Blättern

B*-Bäume (II)

- B*-Bäume besitzen dabei zwei unterschiedliche Arten von Knoten:
  - Innere Knoten:

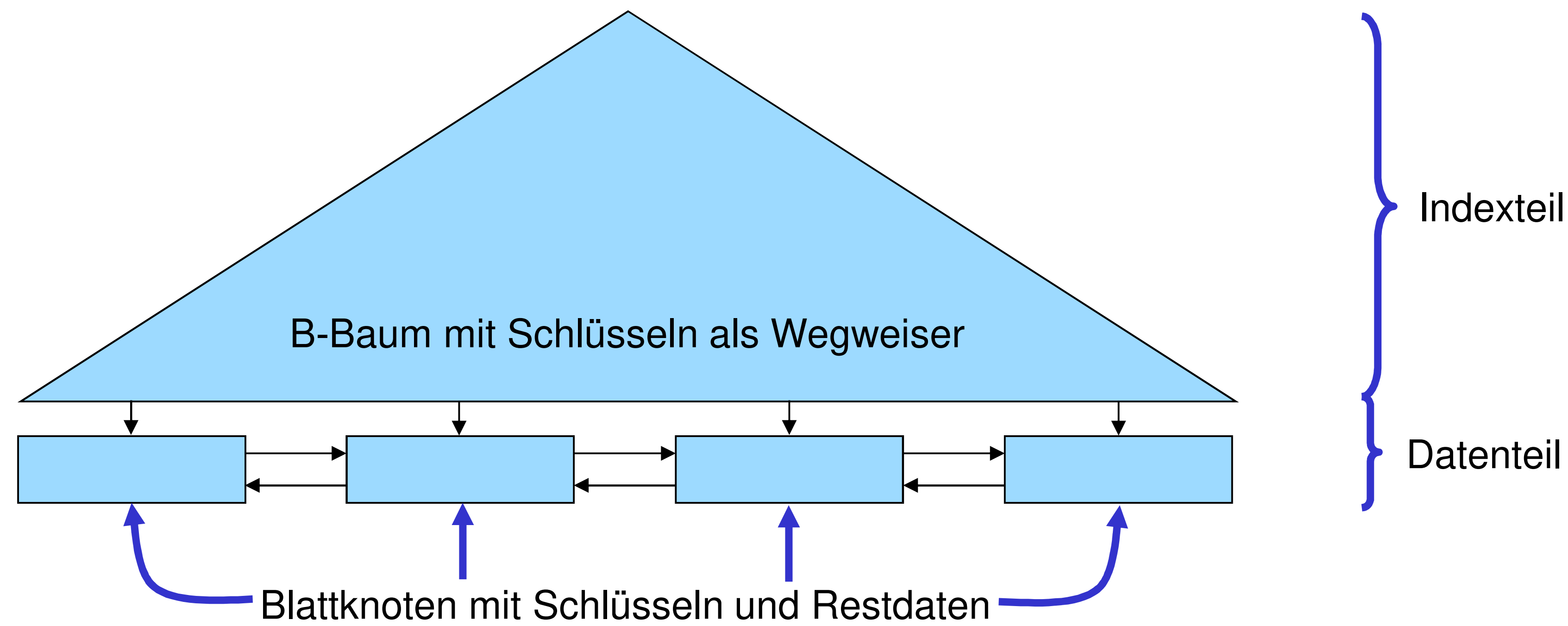


  - Blattknoten:



B*-Bäume (III)

- Insgesamt ergibt sich für B*-Bäume damit folgende Struktur:



- Folgende Variante ist (z.B. in Datenbanken) weit verbreitet:
  - Blätter beinhalten nur Schlüssel plus Verweis auf eigentlichen Datensatz

B*-Bäume (IV)

- Entscheidender Vorteil von B*-Bäumen: Erheblich mehr Einträge passen in einen Knoten
- Es passen $\left\lfloor \frac{b-z}{s+z} \right\rfloor$ Einträge in einen Knoten (d=0)
- Im Beispiel von vorhin: 102 Einträge möglich, d.h. Ordnung 51 anstelle 8
- Angewendet auf die Höhe des Baums ergibt sich:

Datensätze n	Baumhöhe Ordnung 51	Baumhöhe Ordnung 8
1.000	2	4
10.000	3	5
100.000	3	6
1.000.000	4	7
2.000.000	4	7
5.000.000	4	8
10.000.000	5	8

- Baumhöhe und damit Anzahl der Plattenzugriffe lässt sich fast halbieren

B*-Bäume (V)

- Anzahl der Nicht-Blattknoten in B*-Bäumen gegenüber B-Bäumen erheblich reduziert
- Dadurch können häufig alle inneren Knoten im Hauptspeicher gepuffert werden
- Dadurch brauchen bei der Suche keine Indexknoten übertragen werden
- Nur noch Übertragung eines einzigen Knotens (Blocks) notwendig, der die Daten enthält
- Dies ist für den Fall über den Hauptspeicher hinausgehender Datenmengen nicht mehr zu verbessern
- Ein weiterer wichtiger Vorteil von B*-Bäumen ist die leichte Handhabbarkeit von Datensätzen variabler Länge
- Aus all diesen Gründen werden in der Praxis praktisch nur B*-Bäume verwendet

Zusammenfassung

- Motivation und Definition:
  - Datenstrukturen für effizientes Suchen außerhalb des Hauptspeichers
  - Einfaches Übertragen der Suchbaum-Konzepte zu ineffizient
  - Mehrwege-Suchbäume/B-Bäume
- Operationen auf B-Bäumen:
  - Suchen
  - Einfügen
  - Löschen
  - Aufwand
- Beispiel
- B*-Bäume
  - Optimierung durch Trennen von Schlüsseln und Restdaten
  - Werden in der Praxis fast ausschließlich eingesetzt

Übungen (I)

- **Aufgabe 1**

  Gegeben sei die folgenden Zahlenfolge: 23, 66, 42, 11, 5, 77, 69, 55, 16, 84.

  a) Bauen Sie schrittweise einen B-Baum der Ordnung 2 auf!

  b) Löschen Sie aus dem B-Baum nacheinander die Einträge 66, 42 und 23.

- **Aufgabe 2**

  Erläutern Sie die Idee des B*-Baums und geben Sie an, warum mit dieser Datenstruktur gegenüber gewöhnlichen B-Bäumen eine Verbesserung erreicht wird.

- **Aufgabe 3**

  Es liegen die folgenden Daten vor: Das zugrunde liegende Betriebssystem hat eine Blockgröße von 8 kB und es sollen Datensätze verwaltet werden, die aus 16 Byte breiten Schlüssel und 400 Byte Restdaten bestehen. Ermitteln Sie die optimale Ordnung des B-Baums. Geben Sie außerdem für 1.000, 10.000, 100.000, 1.000.000, 2.000.000, 5.000.000 und 10.000.000 Datensätze die maximale Anzahl an Plattenzugriffen an!

  Wie verändern sich Ordnung und Anzahl der Zugriffe, wenn ein B*-Baum zum Einsatz kommt?

Übungen (II)

- **Aufgabe 4**

	wahr	falsch
Plattenspeicherzugriffe sind etwa halb so schnell wie Hauptspeicherzugriffe.		
Eine typische Seitengröße auf dem Plattenspeicher ist 4 kB.		
In B-Baum steht das „B“ für binär.		
B-Bäume sind eine spezielle Form von Mehrwege-Suchbäumen.		
Suchen in B-Bäumen ist von der Baumhöhe abhängig.		
Einfügen in B-Bäumen ist vom Füllungsgrad der Knoten abhängig.		
Im Gegensatz zu binären Suchbäumen wird bei B-Bäumen nicht im Blatt gelöscht.		

Algorithmen und Datenstrukturen

Teil 12:
Suchen (III) - Hashing

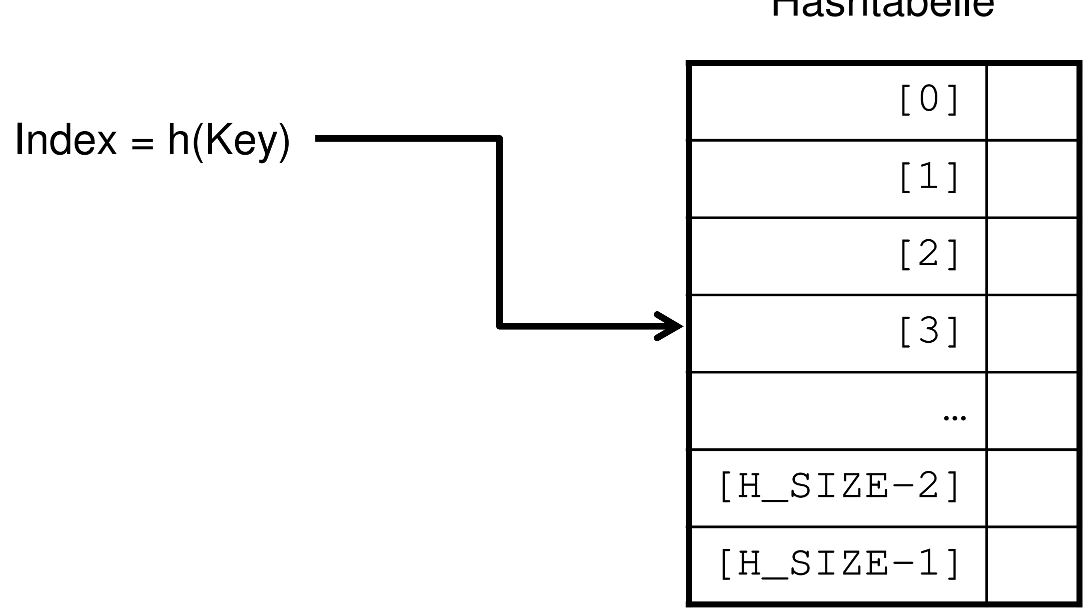
DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Hashing/Hash-Funktion/Hash-Tabelle

- Hashing (to hash = zerhacken): Abbildung von Ausgangsmenge auf Zielmenge
- Abbildung h (Hash-Funktion)
- h(k) Hash-Wert (Synonym: Hash-Code, Index) von Schlüssel k
- Zugrundeliegende Datenstruktur wird als Hash-Tabelle H bezeichnet
- H habe Größe `H_SIZE`



Perfektes Hashing und Kollisionen

- Perfektes Hashing:
  Für alle Schlüssel k_1, k_2 mit $k_1 \neq k_2$ gilt stets $h(k_1) \neq h(k_2)$
- Kollision: $h(k_1) = h(k_2)$ für $k_1 \neq k_2$
- Perfektes Hashing kann nicht immer garantiert werden
- Daher notwendig: Kollisionsbehandlung
  - Verkette Überläufer (Chaining, Open Hashing)
  - Sondierung (Probing, Open Adressing, Closed Hashing):
    - Suche alternative Position
    - Strategien:
      - Lineares Sondieren
      - Quadratisches Sondieren
      - …
  - Überlaufbereich (Overflow Area):
    - Speicherung kollidierender Elemente in separatem Bereich

Hash-Funktion: Eigenschaften (I)

- Eindeutigkeit/Reproduzierbarkeit:
  - Funktion muss eindeutig von Quellmenge auf Zielmenge abbilden
  - Wiederholtes Berechnen des Hash-Wertes desselben Quellelements muss dasselbe Ergebnis liefern
- Unumkehrbarkeit:
  - Zur Hash-Funktion gibt es keine inverse Funktion, mit der es möglich wäre, für ein gegebenes Zielelement ein passendes Quellelement zu berechnen
- Datenreduktion:
  - Speicherbedarf Hash-Wert muss deutlich kleiner sein als Eingabewert
- Zufälligkeit:
  - Ähnliche Quellelemente sollten möglichst zu völlig verschiedenen Hash-Werten führen
- Gute Verteilung:
  - Hash-Werte sollten möglichst gut über Zielmenge verteilt sein

Hash-Funktion: Eigenschaften (II)

- Kollisionsfreiheit/-armut:
  - Optimal ist Kollisionsfreiheit, d.h. zwei verschiedene Quellelemente haben stets unterschiedliche Hash-Werte („Perfektes Hashing“)
  - I.A. nicht garantierbar, daher schwächere Eigenschaft Kollisionsarmut, d.h. seltenes Auftreten von Kollisionen
- Effizienz:
  - Hash-Funktion muss schnell berechenbar sein

Hash-Funktion: Beispiel (I)

- Szenario:
  - Ausgangsdaten: Menge der Vornamen
  - Zielmenge: Werte von 0 bis 9
  - Enfügen von "Ali", "Babsy", "Alfred", "Arno", "Alice", "Benno", "Kurt", "Alex", "Angy", "Bine", "Max", "Franz", "Susi", "Alf" in Hash-Tabelle
  - Kollisionsbehandlung: Verkettete Überläufer
  - Betrachtung verschiedener Hash-Funktionen
  - Maß für Bewertung: Summe Suchschritte für jedes einzelne Element
  - Optimales Maß: 14 (ein Suchschritt pro Element)
- Beispiel stammt von http://www.tfh-berlin.de/~oo-plug/Ad/Hash.html (Seite nicht mehr verfügbar)

Hash-Funktion: Beispiel (II)

- Variante 1 (Konstante Hash-Funktion)

```
int hash_1(String s) {return 3;}
```

Resultat:

- Bewertung: Sehr schlechte Hash-Funktion (105 Suchschritte), entspricht Liste

Hash-Funktion: Beispiel (III)

- Variante 2 (Erstes Zeichen wird gerade/ungerade codiert)

```
int hash_2(String s) {if (s.charAt(0) % 2 == 0) return 3;
                      else return 4;}
```

Resultat:

Index	Liste
[0]	→ NULL
[1]	→ NULL
[2]	→ NULL
[3]	→ Babsy → Benno → Bine → Franz → NULL
[4]	→ Ali → Alfred → Arno → Alice → Kurt → Alex → Angy → Max → Susi → Alf → NULL
[5]	→ NULL
[6]	→ NULL
[7]	→ NULL
[8]	→ NULL
[9]	→ NULL

- Bewertung: Etwas besser als erste Variante (65 Suchschritte), aber ein Großteil der Zielmenge bleibt nach wie vor ungenutzt

Hash-Funktion: Beispiel (IV)

- Variante 3 (Berechnung aus erstem Zeichen)

```
int hash_3(String s) {return s.charAt(0) % H_SIZE;}
```

Resultat:

- Bewertung: Besser (45 Suchschritte), aber: viele Listen bleiben leer

Hash-Funktion: Beispiel (V)

- Variante 4 (Berechnung aus drei Zeichen)

```
(1) int hash_4(String s){
(2)  int first = s.charAt(0) % 3;
(3)  int middle = s.charAt(s.size()/2) % 5;
(4)  int last = s.charAt(s.size()-1) % 7;
(5)  return (first+middle+last) % H_SIZE;}
```

Resultat:

Index	Liste
[0]	NULL
[1]	NULL
[2]	Susi → NULL
[3]	Bine → NULL
[4]	Alex → NULL
[5]	Ali → Babsy → Alice → Max → NULL
[6]	Benno → Franz → NULL
[7]	Angy → NULL
[8]	Alfred → Arno → Kurt → NULL
[9]	Alf → NULL

- Bewertung: Recht gute Verteilung (24 Suchschritte)

Hash-Funktion: Beispiel (VI)

- Variante 5 (Berücksichtigung aller Zeichen)

```
(1) int hash_5(String s){
(2)   int i = 0;
(3)   for (int j=0; j<s.size(); j++)
(4)      i += s[j] % 32 + j;
(5)  return i % H_SIZE;}
```

Resultat:

Index	Liste
[0]	Alice → Benno → NULL
[1]	Alfred → Max → NULL
[2]	Alf → NULL
[3]	Angy → NULL
[4]	Arno → Susi → NULL
[5]	Ali → Franz → NULL
[6]	Kurt → Bine → NULL
[7]	NULL
[8]	Alex → NULL
[9]	Babsy → NULL

- Bewertung: Nahezu optimale Verteilung (19 Suchschritte)

Hash-Funktionen: Datentypen

- Hash-Funktion hängt von Datentyp und konkreter Anwendung ab
- Ganzzahlige Werte: Direkte Eingabe in Hash-Funktion
- Andere Datentypen:
    - Rückführung auf ganze Zahlen:
        - Fließkommazahlen z.B. Addition von Mantisse und Exponent
        - Zeichenketten z.B. Addition der ASCII- oder Unicode-Werte ausgewählter Zeichen

- Prinzipielle Konzepte für Hash-Funktionen:
    - Divisionsmethode
    - Folding
    - Mid-Square-Methode

Divisionsmethode

- Bilde ganzzahligen Rest der Division durch Größe Hash-Tabelle
- $h(k) = k \; mod \; N$
- Funktioniert am besten (Verteilung), wenn N Primzahl
- Beispiel:
  - Hash-Tabelle mit $H_SIZE = 13$
  - Einfügen 55.456
  - $h(55.456) = 55.456 \; MOD \; 13 = 11$

Folding

- Zerlege k in gleich große Teile (außer letztem Teil)
- Addiere diese Teile unter Vernachlässigung des letzten Übertrag
- Beispiel:
  - Hash-Tabelle Kapazität 1000
  - Zerlege k in Teil der Länge 3 (+Rest) und bilde Summe
  - Beispiel: k = 53746312526

537	463	125	26

```
    537
+   463
+   125
+    26
-------
   1151
```

  - h(53726312526) = 151

Mid-Square-Methode

- Bilde Quadrat von k
- Nimm mittleren Teil hiervon als Hash-Wert
- Beispiel:
  - Hash-Tabelle $H_SIZE = 1000$
  - Hash-Wert ab Position p mit $p = Länge\ k^2\ DIV\ 2$
  - Beispiel:
    - $k = 102.726$
    - $k^2 = 10.552.631.080$
    - $p = 11\ DIV\ 2 = 5$

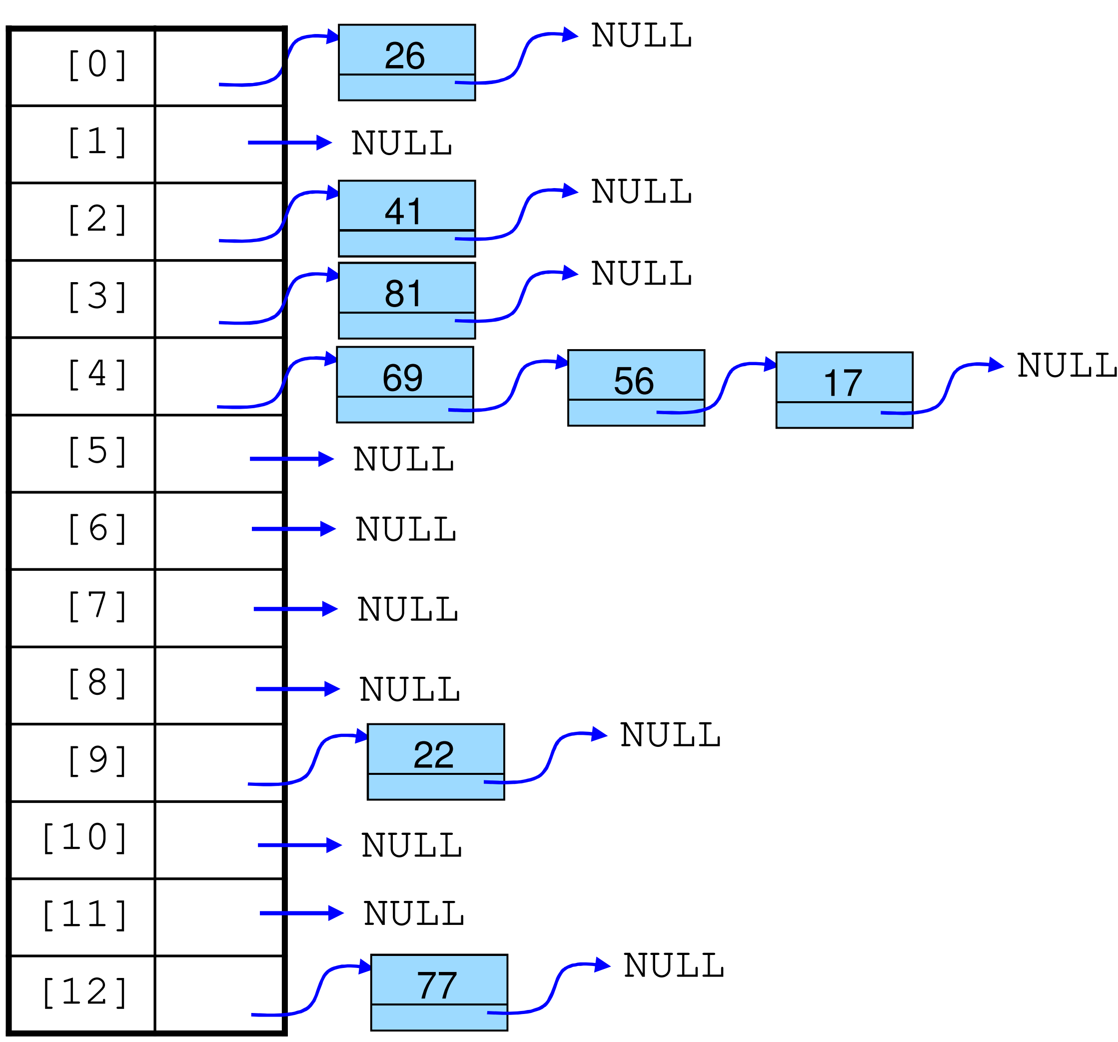
      10.552.631.080
    - $h(102.726) = 263$

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Verkettete Überläufer (I)

- Realisierung Hash-Tabelle als Feld von Listen
- Synonym: Chaining, Open Hashing
- Beispiel:
  - $h(k) = k\ MOD\ 13$
  - Füge Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein

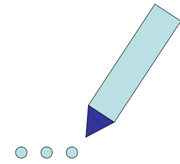


Verkettete Überläufer (II)

- Handhabung langer Listen:
  - Zielbereich Hash-Funktion zu klein ⇒ Effizienz kann nachlassen (lange Listen müssen durchsucht werden)
  - Vorgehensweise:
    - Überschreiten Listenlängen gewisse Schwellwerte, wird dynamisch „nachgebessert“
    - Geschieht durch Vergrößerung der Zielmenge
    - Alle Werte müssen per Hash-Funktion neu zugeordnet werden

Lineares Sondieren (I): Prinzip

- Sei Q Ausgangsmenge, T Hash-Tabelle, h Hash-Funktion
- Einfügen eines neuen Wertes k ∈ Q:
  - Wenn $T[h(k)]$ frei, dann füge k ein
  - Sonst: Versuche $T[h(k) + i]$ für $i = 1, 2, 3, \ldots$
- Beispiel:
  - Hash-Funktion $h(k) = k\ MOD\ 13$
  - Füge nacheinander Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein

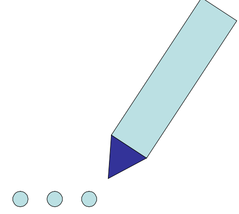


Lineares Sondieren (VI): Anmerkungen

- Sekundärkollision: Kollision, die erst beim Sondieren auftritt
- Lineares Sondieren wird auch in folgenden Varianten verwendet:
  - Versuche $T[h(k) + i{\cdot}c]$ für $i = 1, 2, 3, \ldots$ und eine Konstante c
  - Versuche $T[h(k) + i], T[h(k) - i]$ für $i = 1, 2, 3, \ldots$

Quadratisches Sondieren (I): Prinzip

- Sei Q Ausgangsmenge, T Hash-Tabelle, h Hash-Funktion
- Einfügen eines neuen Wertes k ∈ Q:
  - Wenn $T[h(k)]$ frei, dann füge k ein
  - Sonst: Versuche $T[h(k) + i^2]$ für $i = 1, 2, 3, \ldots$
- Beispiel:
  - Hash-Funktion $h(k) = k \; MOD \; 13$
  - Füge nacheinander Werte 22, 41, 69, 26, 56, 17, 77 und 81 ein



Überlaufbereich

- Speichere kollidierende Elemente in separatem Bereich
- Beispiel:

[0]	26
[1]	
[2]	41
[3]	81
[4]	69
[5]	
[6]	
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

Überlaufbereich:

[0]	56
[1]	17
[2]	
[3]	
[4]	

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Suchen (I)

- Berechnung Hash-Wert des gesuchten Elements
- Zugriff auf Hash-Tabelle
- Evtl. Weitersuchen gemäß Sondierungsstrategie
- Beispiel:
  - Ann.: Lineare Sondierung
  - Suchen von 22
  - $h(22) = 22\ MOD\ 13 = 9$

Index	Wert
[0]	26
[1]	
[2]	41
[3]	81
[4]	69
[5]	56
[6]	17
[7]	
[8]	
[9]	22 (Treffer)
[10]	
[11]	
[12]	77

Suchen (II)

- Beispiel:
  - Ann.: Lineare Sondierung
  - Suchen von 17
  - $h(17) = 17\ MOD\ 13 = 4$

[0]	26
[1]	
[2]	41
[3]	81
[4]	69
[5]	56
[6]	17
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

Weitersuchen

Weitersuchen

Treffer

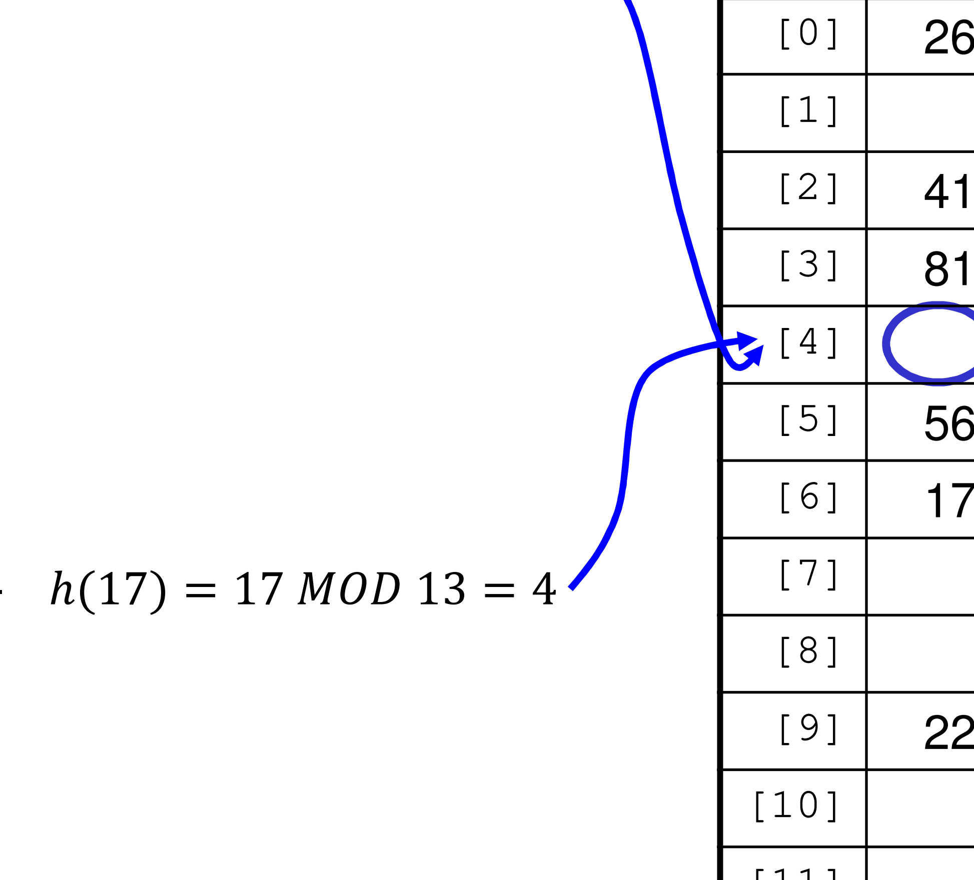
- Bezeichnung: Pfad von ursprünglichem Hash-Wert über Sondierungswerte nennt man Sondierungskette

Löschen (I)

- Vorgehen scheint zunächst offensichtlich:
  - Suchen (gemäß Hash-Funktion und Sondierungsstrategie) und Entfernen zu löschender Eintrag
  - Funktioniert bei verkettetem Überläufer
  - Mögliches Problem bei Sondierungen:
    - Auf zu löschenden Eintrag folgen weitere Einträge, die aufgrund von Sondierungen dahin gelangt sind und nun nicht mehr gefunden werden
  - Beispiel: Resultat von vorhin mit linearem Sondieren

Löschen (II): Beispiel

- Löschen von 69, Suchen von 17:
  - $h(69) = 69\ MOD\ 13 = 4$
  - $h(17) = 17\ MOD\ 13 = 4$



Kein Treffer, obwohl Element in Hash-Tabelle

Löschen (III)

- Strategien zur Vermeidung von Unterbrechungen in Sondierkette:
  - Reorganisation:
    - Alle in Frage kommenden weiteren Einträge werden überprüft und gegebenenfalls verschoben
  - Markierung:
    - Markierung gelöschter Eintrag mit Status „Gelöscht“
    - Hier können neue Werte eingefügt werden
    - Keine Unterbrechung Sondierkette

Löschen (IV): Reorganisation

- Löschen von 69:
  - $h(69) = 69\ MOD\ 13 = 4$

[0]	26
[1]	
[2]	41
[3]	81
[4]	56
[5]	17
[6]	
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

Durch Kollisionen verschobene Elemente reorganisieren

Löschen (IV): Markierung

- Löschen von 69:
  - $h(69) = 69\ MOD\ 13 = 4$

Index	Wert
[0]	26
[1]	
[2]	41
[3]	81
[4]	Gelöscht
[5]	56
[6]	17
[7]	
[8]	
[9]	22
[10]	
[11]	
[12]	77

Löschen (V)

- Bewertung:
  - Reorganisation:
    - Sehr aufwändig
    - Daher in Praxis kaum praktikabel
  - Markierung:
    - Performant
    - Bei häufigem Löschen: Sondierketten länger als nötig

Übersicht

- Grundbegriffe
- Kollisionsbehandlung
- Suchen und Löschen
- Bewertung

Bewertung Hashing

- Aufwand bei geringer Kollisionswahrscheinlichkeit:
  - Suchen und Einfügen in O(1)
  - Löschen:
    - O(1) bei Markierungsstrategie
    - O(n) bei Reorganisation

- Belegungsgrad Hash-Tabelle bei Sondierung:
  - Dramatische Verschlechterung von Einfüge- und Suchverhalten ab Füllungsgrad von 80%

- Nachteile gegenüber Bäumen:
  - Keine Unterstützung von Reihenfolge berücksichtigende Operationen:
    - Finde Minimum bzw. Maximum
    - Bestimme Vorgänger bzw. Nachfolger
    - Werte aus bestimmtem Bereich
    - Alle Werte geordnet ausgeben

Andere Anwendungsbereiche

- Hashing nicht nur zum Suchen
- Andere Anwendungen:
  - Kontrollsummen, z.B. bei Downloads
  - Verschlüsselung von Passwörtern
  - Signaturen
- Bekannte Algorithmen:
  - MD5 (Message-Digest Algorithm)
  - SHA-1 (Secure Hash Algorithm): 160 Bit-Hashwert
  - SHA-2:
    - SHA-256, SHA-384 und SHA-512

Zusammenfassung (I)

- Grundbegriffe:
  - Hashing
  - Hash-Funktion
  - Hash-Tabelle
  - Eigenschaften Hash-Funktion
  - Konstruktion von Hash-Funktionen:
    - Divisionsmethode
    - Folding
    - Mid-Square-Methode
- Kollisionsbehandlung:
  - Verkettete Überläufer
  - Sondierungen:
    - Lineares Sondieren
    - Quadratisches Sondieren
  - Überlaufbereich

Zusammenfassung (II)

- Suchen und Löschen
  - Suchen, Sondierungsketten
  - Löschen durch Reorganisieren
  - Löschen durch Markieren

- Bewertung:
  - Effizienz
  - Andere Anwendungsgebiete

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
H M G H Z F R K O Y I Y Q B I L Z N O L G E I M Y
M U Z E T B E W X F B G Y U M U J O V Q T V S I T
Z G B U D Z M X F B V L X C T Z V I B T V W U H T
Q N S G D D J T M S E B R F Z E V T E E U S R S H
C A J Q N S N K S X A Y F G X W U K P Q T I Y G F
K O A E W J I J B N R C S J B G R N P R M Q C S U
H K I X N G K Y S Z T J W T Q E I U K J N U Y A T
Q C Y J C A G K J L Q U L Y I S Z F H W Y O W N X
B N I O I J D H Y C G G M D L W O H V J L Y N Y V
U D S U C Y T Y I G Q W N R E M G S N U X K E A F
V A H A N R G S G M D O K G A V N A H D B C Z B Q
U F X B W W I S P C S S P U F S M H B P P H E R C
Z Y K H J A P R O M F E J G L Y N Z C D H X S H A
H D X Z E H B T K C F S M K X F T O M I G X P S C
H A S H T A B E L L E O A K L M Z R I L M H G M T
T V Z E Q P L P D D B M V K I L V Y M S F C O I R
B Z L H I C U T W U C X H Z E V M Y W E I J P Z Z
X D L X T J A R G R X T W H R P E D C W Y L M J C
R E F U E A L R E B E U E T E T T E K R E V L G O
K P F Q P I P X N K Y M I P R M Y F U N T C O O M
N Z L K O E N P Y E X H M A O F F H G P X B W H K
E I Z J H M K Q Y I N S P U G I O U U N Y E G M I
G R Q A D J O J U K S N Z I X E N G L V D B F B A
Z E H Z G D P X F M Y B Q R I G C L U R U S Z H Q
K M K L P E P L G U Q B N W X Y B Y Y G Z V J C N
```

Übungen (II)

- **Aufgabe 2**
  a) Was ist perfektes Hashing?
  b) Was ist eine Kollision?
  c) Welche Verfahren zur Kollisionsbehandlung gibt es?
  d) Welche Eigenschaften sollte eine Hash-Funktion haben?
  e) Welche Sondierungsstrategien gibt es?
  f) Was ist bei Einsatz einer Sondierungsstrategie beim Suchen zu beachten?
  g) Was ist bei Einsatz einer Sondierungsstrategie beim Löschen zu beachten?
  h) Welche Anwendungen findet Hashing neben dem Suchen?

Übungen (III)

- **Aufgabe 3**

  Gegeben sei die folgenden Zahlenfolge: 23, 24, 41, 34, 59, 61, 63, 55, 16, 84.

  a) Fügen Sie diese nacheinander in eine Hash-Tabelle mit verketteten Überläufern ein! Als Hash-Funktion soll hierbei h(x) = x MOD 9 verwendet werden!

  b) Fügen Sie die Zahlenfolge nun nacheinander in eine Hash-Tabelle als Feld ein! Es soll die gleiche Hash-Funktion verwendet werden und eine lineare Sondierungsstrategie angewendet werden!

     Übersteigt der Belegungsgrad 80% der Kapazität der Hash-Tabelle, so wird deren Kapazität um 50% erhöht. Die neue Hash-Funktion lautet dann x MOD m mit m Größe der neuen Tabelle.

- **Aufgabe 4**

  Welche zwei Strategien sind beim Löschen in Hash-Tabellen denkbar? Nennen und erläutern Sie diese kurz und nennen Sie jeweils ein Problem, das bei den Strategien auftritt!

Übungen (IV)

- **Aufgabe 5**

	wahr	falsch
Eine Hash-Funktion h(x) = x mod n arbeitet am besten, wenn n eine Primzahl ist.		
Ab einem Füllungsgrad von 30% ist das Einfügen in Hash-Tabellen nicht mehr effizient.		
Kollisionen können beim Hashing nur entstehen, wenn zweimal der gleiche Wert eingefügt werden soll.		
Hashing ist gut geeignet, wenn relativ selten gelöscht wird.		
Hash-Verfahren sind zum schnellen Suchen stets Baumstrukturen vorzuziehen.		
Bei geringer Kollisionswahrscheinlichkeit hat das Einfügen in eine Hash-Tabelle einen Aufwand $O(\log_2 n)$.		
Quadratisches Sondieren erleichtert gegenüber linearem Sondieren das Löschen von Einträgen.		

Algorithmen und Datenstrukturen

Teil 13:
Suchen (IV) – Suchen in Texten

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

- Motivation
- Brute Force Algorithmus
- Algorithmus von Knuth-Morris-Pratt
- Algorithmus von Boyer-Moore

Motivation

- Ziel: Suche bestimmte Zeichenkette (Pattern/Muster) in Text
- Anwendung:
  - Editor
  - Textverarbeitung
  - Webseite
  - ...

Suchen und Ersetzen
Suchen | Ersetzen | Gehe zu
Suchen nach: Suchalgorithmus
Reduzieren | Weitersuchen | Abbrechen
Gesamt
-/Kleinschreibung
ganzes Wort suchen
Mustervergleich
. Schreibweise (Englisch)
Wortformen suchen (Englisch)
Format | Sonstiges | Keine Formatierung

Suchen nach
Suchen nach: Suchalgorithmus
Nächstes suchen
Alle markieren
Schließen
Bedingungen: Text | Hex
Ganze Wörter
Groß-/Kleinschreibung
Regulärer Ausdruck
Beim Suchen Umbruch
Richtung: Aufwärts | Abwärts
Auswahl erweitern
In allen Dokumenten

In Seite suchen | Hervorheben | Groß-/Kleinschreibung | Ganze Wörter

Motivation (II)

- Dynamische vs. statische Texte
  - Dynamische Texte (z.B. im Texteditor): Aufwändige Vorverarbeitung / Indizierung i.a. nicht sinnvoll
  - Relativ statische Texte: Erstellung von Indexstrukturen zur Suchbeschleunigung denkbar
- Suche nach beliebigen Strings/Zeichenketten vs. Worten/Begriffen
- Exakte Suche vs. approximative Suche (Ähnlichkeitssuche)

Die Suche nach allen (exakten) Vorkommen eines Musters in einem Text ist ein Problem, das häufig auftritt (z.B. in Editoren, Datenbanken etc.). Wir vereinbaren folgende Konventionen (wobei Σ ein Alphabet ist):

$$\begin{aligned} \text{Muster } P &= P[0 \ldots m-1] \in \Sigma^m \\ \text{Text } T &= T[0 \ldots n-1] \in \Sigma^n \\ P[j] &= \text{Zeichen an der Position } j \\ P[b \ldots e] &= \text{Teilwort von } P \text{ zwischen den Positionen } b \text{ und } e \end{aligned}$$

Beispiel

$P = \mathrm{abcde}$, $P[0] = \mathrm{a}$, $P[3] = \mathrm{d}$, $P[2 \ldots 4] = \mathrm{cde}$.

[Steffen 2009]

Definitionen (I)

- leeres Wort: ε
- Länge eines Wortes x: $|x|$
- Konkatenation zweier Worte x und y: xy

Definition

Ein Wort w ist ein *Präfix* eines Wortes x ($w \sqsubset x$), falls $x = wy$ gilt für ein $y \in \Sigma^*$. w ist ein *Suffix* von x ($w \sqsupset x$), falls $x = yw$ gilt für ein $y \in \Sigma^*$. x ist also auch ein Präfix bzw. Suffix von sich selbst.

Lemma

Es seien x, y und z Zeichenketten, so dass $x \sqsupset z$ und $y \sqsupset z$ gilt. Wenn $|x| \leq |y|$ ist, dann gilt $x \sqsupset y$. Wenn $|x| \geq |y|$ ist, dann gilt $y \sqsupset x$. Wenn $|x| = |y|$ ist, dann gilt $x = y$.

[Steffen 2009]

Definitionen (III)

Definition

Ein Muster P kommt mit der Verschiebung s im Text T vor, falls $0 \leq s \leq n - m$ und $T[s \ldots s + m - 1] = P[0 \ldots m - 1]$ gilt. In dem Fall nennen wir s eine *gültige Verschiebung*.

Das Problem der exakten Textsuche ist nun, alle gültigen Verschiebungen zu finden. Mit den Bezeichnungen $S_0 = \varepsilon$ und $S_k = S[0 \ldots k - 1]$ (d.h. S_k ist das k-lange Präfix eines Wortes $S \in \Sigma^\ell$ für $0 \leq k \leq \ell$) können wir das Problem der exakten Textsuche folgendermaßen formulieren:

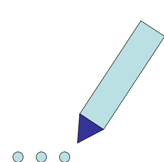
Definition

Finde alle Verschiebungen s, so dass $P \sqsupset T_{s+m}$.

[Steffen 2009]

Aufgabe

- Gegeben sei Σ = {a,b,c}, T = „abccabcabc“, P = „ab“
  (1) Geben sie alle Worte der maximalen Länge 3 über Σ an!
  (2) Geben Sie alle Präfixe und Suffixe des Wortes „abcac“ an!
  (3) Geben Sie T[0], T[7] und T[3...5] an!
  (4) Geben Sie alle gültigen Verschiebungen von P in T an!



Übersicht

- Motivation
- Brute Force Algorithmus
- Algorithmus von Knuth-Morris-Pratt
- Algorithmus von Boyer-Moore

Idee

- Schiebe Pattern zeichenweise am Text entlang
- Vergleiche jeweils von Anfang an, bis das Ende des Pattern oder eine Unstimmigkeit (Mismatch) zwischen korrespondierenden Elementen des Pattern und des zu durchsuchenden Textes auftritt
- Beispiel: Suche nach „eine“

```
Nächste Woche schreiben wir eine Informatik-Klausur.
eine
 eine
  eine
   ...
    eine
       ...
              eine
                   ...
                        eine
```

Brute Force-Algorithmus

- Pseudocode:

Beispiel (Naive-StringMatcher)

```
Naive-StringMatcher(T, P)
  1 n ← length[T]
  2 m ← length[P]
  3 for s ← 0 to n − m do
  4     if P[0 ... m − 1] = T[s ... s + m − 1] then
  5         print "Pattern occurs with shift" s
```

[Steffen 2009]

- Aufwand: O((n-m)*m) = O(n*m)
- Begründung:
  - (n-m) Durchgänge der äußeren Schleife
  - m Durchgänge der inneren Schleife

Bewertung

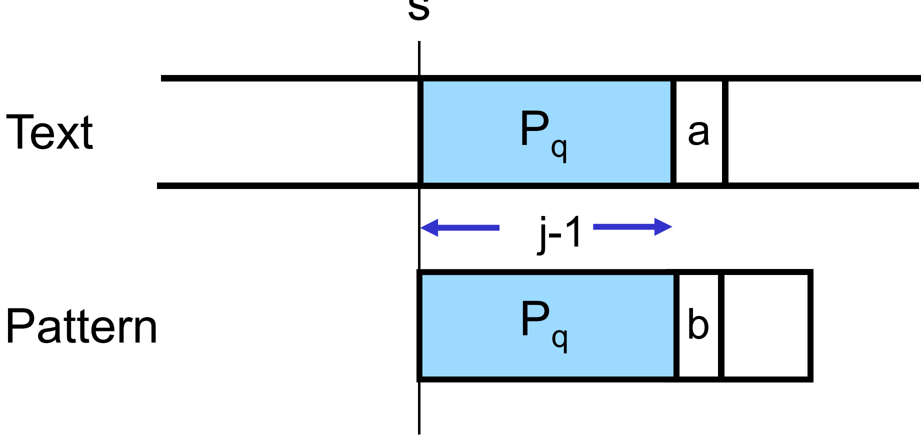
- Brute Force-Algorithmen sind (im worst case) zu ineffizient
- Ideen zur Verbesserung:
  - Nutzung der Musterstruktur, Kenntnis der im Muster vorkommenden Zeichen
  - Knuth-Morris-Pratt (1974): Nutze bereits geprüften Musteranfang um ggf. Muster um mehr als eine Stelle nach rechts zu verschieben
  - Boyer-Moore (1976): Teste Muster von hinten nach vorne
  - Horspool (1980): Optimierung Boyer-Moore-Algorithmus

Übersicht

- Motivation
- Brute Force Algorithmus
- Algorithmus von Knuth-Morris-Pratt
- Algorithmus von Boyer-Moore

Funktionsweise (I)

- Vergleiche Muster P von links nach rechts mit Text T
- Zur Vermeidung unnützer Vergleiche: Nutze Information über durchgeführte Vergleiche aus
- Sei P_q Präfix von P, tritt an Position s in T auf



- Frage: Wie weit darf verschoben werden?
- Abhängig von P_q, z.B.

Text:	DATENSTRUKTUREN	GEGEBENENFALLS
Muster	DATUM	GEGENSATZ

- Unterscheidung verschiedener Fälle

Präfixfunktion π (I)

- Gibt an, wie weit Muster verschoben werden darf
- Gegeben Muster P[1 … m]
- Präfixfunktion $\pi : \{1,2,\ldots,m\} \rightarrow \{0,1,\ldots,m-1\}$

  $\pi(q) \coloneqq max\{k: k < q \text{ und } P_k \sqsupset P_q\}$
- In Worten: π liefert Länge des längsten Präfix von P, das echtes Suffix von P_q ist
- Berechnung:

```
COMPUTE-PREFIX-FUNCTION(P)
1  m = P.length
2  let π[1..m] be a new array
3  π[1] = 0
4  k = 0
5  for q = 2 to m
6      while k > 0 and P[k + 1] ≠ P[q]
7          k = π[k]
8      if P[k + 1] == P[q]
9          k = k + 1
10     π[q] = k
11 return π
```

Präfixfunktion π (II): Beispiel (I)

- Berechnung π für Muster „ababaca“!

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1				

- $q = 1 \Rightarrow \pi(1) = 0$ (Initialisierung)
- $q = 2$:

```
ababaca
 ababaca
```

  Mismatch
  $\Rightarrow \pi(2) = 0$, Schiebe P um 1 weiter
- $q = 3$:

```
ababaca
  ababaca
```

  Match
  $\Rightarrow \pi(3) = 1$, Erhöhe k auf 2

Präfixfunktion π (III): Beispiel (II)

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3		

- $q = 4$:
  ```
  ababaca
    ababaca
  ```
  Match
  $\Rightarrow \pi(4) = 2$, Erhöhe k auf 3

- $q = 5$:
  ```
  ababaca
    ababaca
  ```
  Match
  $\Rightarrow \pi(5) = 3$, Erhöhe k auf 4

Präfixfunktion π (IV): Beispiel (III)

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	

- $q = 6$:

```
ababaca
  ababaca
```

Mismatch, Lookup bei $\pi(3) = 1$
$\Rightarrow$ Schiebe P um 2 weiter

```
ababaca
    ababaca
```

Mismatch, Lookup bei $\pi(1) = 0$
$\Rightarrow$ Schiebe P um 1 weiter

```
ababaca
     ababaca
```

Mismatch
$\Rightarrow \pi(6) = 0$, Schiebe P um 1 weiter

Präfixfunktion π (V): Beispiel (IV)

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

- $q = 7$:

  ```
  ababaca
        ababaca
  ```

  Match
  $\Rightarrow \pi(7) = 1$, Erhöhe k auf 2

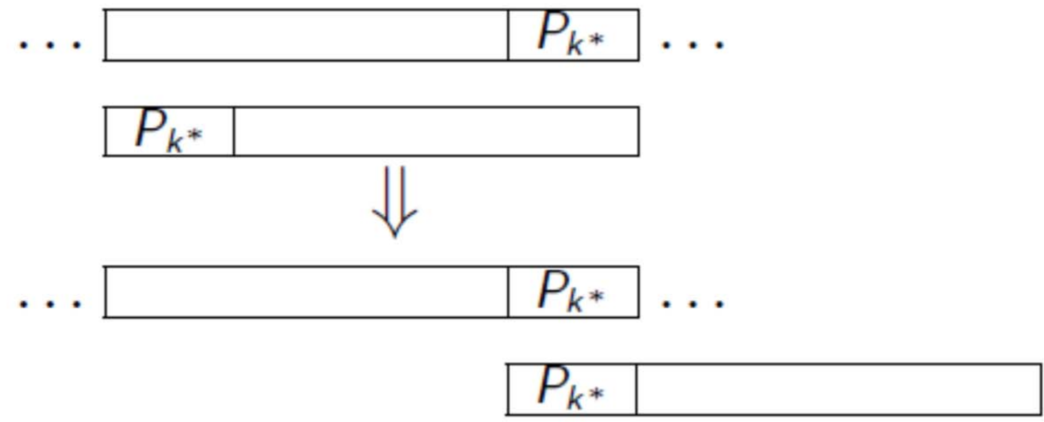
- Ende T erreicht $\Rightarrow$ Terminierung

Funktionsweise (II)

(A) $q = m$, d.h. s ist gültige Verschiebung:

1. Bestimme $k^* = \pi[m] = \max\{k : k < m \text{ und } P_k \sqsupset P_m\}$.

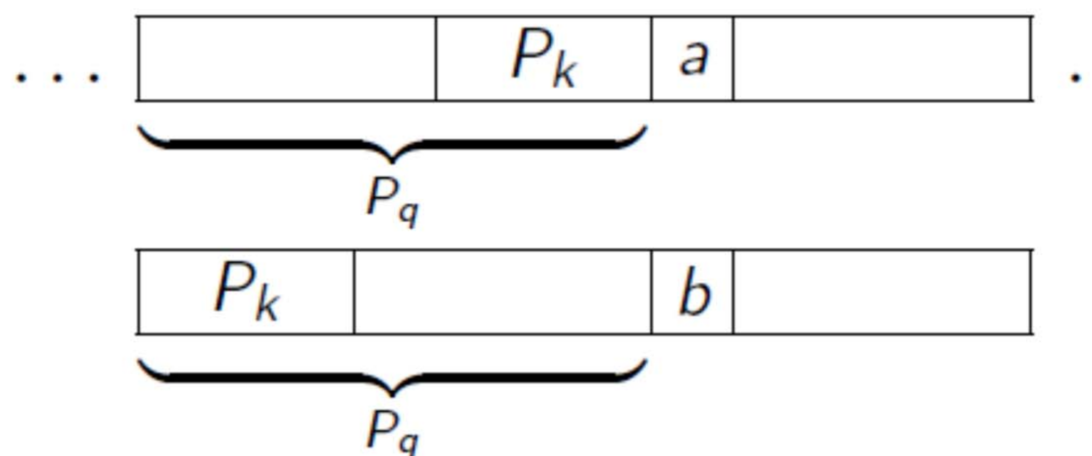
2.a) Falls $k^* > 0$, verschiebe das Muster nach rechts, so dass das Präfix P_{k^*} von P unter dem Suffix P_{k^*} von $T[s \ldots s + |P| - 1]$ liegt.



2.b) Falls $k^* = 0$, verschiebe das Muster P um die Länge $|P|$.

[Steffen 2009]

Funktionsweise (III)



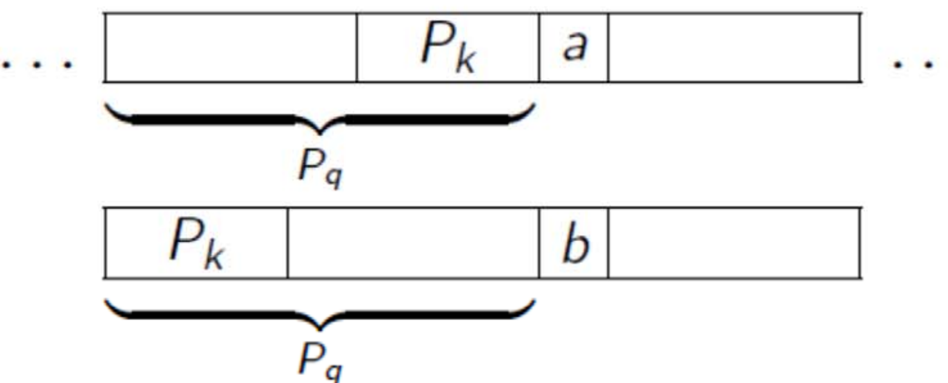
- Nun:
  a) $a \neq b$ und P_q länger als ein Zeichen ($q > 0$)
  b) $a \neq b$ und P_q ein Zeichen ($q = 0$)
  c) $a = b$

[Steffen 2009]

#

Funktionsweise (IV)

(B) $q < m$:



(a) $a \neq b$ und $q \neq 0$:

- bestimme $\pi[q] = \max\{k : k < q \text{ und } P_k \sqsupset P_q\}$, d.h. die Länge des längsten Präfixes von P, das auch Suffix von P_q ist;
- verschiebe P nach rechts, so dass das Suffix P_k von P_q über dem Präfix P_k von P liegt.

... P_k a ...

P_k b

[Steffen 2009]

Funktionsweise (V)

(b) $a \neq b$ und $q = 0$:

- verschiebe P um 1 nach rechts.

b

(c) $a = b$:

- mache weiter mit $q + 1$.

... P_q | a ...

P_q | a

... P_{q+1} ...

[Steffen 2009]

Algorithmus

- Ähnlich zu Präfixberechnung: Beide vergleichen Text T gegen Muster P
- Mustersuche: Vergleicht Text T gegen Muster P
- Präfixberechnung: Vergleicht Muster P gegen sich selbst (d.h. T = P)

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                   // number of characters matched
 5  for i = 1 to n                          // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                        // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                       // next character matches
10      if q == m                           // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                        // look for the next match
```

Beispiel (I)

- Gegeben seien der Text „babababacababacaababababab" und das Muster „ababaca"

```
babababacababacaababababab
ababaca
```

  - Mismatch ⇒ Schiebe P um 1 weiter (Fall B.b)

```
babababacababacaababababab
 ababaca
```

  - Match ⇒ Setze q = 1 (Fall B.c)

```
babababacababacaababababab
 ababaca
```

  - Match ⇒ Setze q = 2 (Fall B.c)
  - …
  - Match ⇒ Setze q = 5 (Fall B.c)

```
babababacababacaababababab
 ababaca
```

Beispiel (II)

```
bababababacababacaabababab
 ababaca
```

- Mismatch $\Rightarrow$ Lookup $\pi(5) = 3$, Schiebe P um 2 weiter (Fall B.a)

```
bababababacababacaabababab
   ababaca
```

- Match $\Rightarrow$ Setze q = 1 (Fall B.c)
- …
- Match $\Rightarrow$ Setze q = 7 (Fall B.c)

```
bababababacababacaabababab
   ababaca
```

- Match und $q = m \Rightarrow$ Treffer an Position 4, Lookup $\pi(7) = 1$,
  Schiebe P um $\mathrm{m} - \pi(7) = 6$ (Fall A.2a)

```
bababababacababacaabababab
         ababaca
```

- Match $\Rightarrow$ Setze q = 1 (Fall B.c)
- …
- Match $\Rightarrow$ Setze q = 7 (Fall B.c)

Beispiel (III)

```
bababababacababacaabababab
          ababaca
```

- Match und $q = m \Rightarrow$ Treffer an Position 10, Lookup $\pi(7) = 1$,
  Schiebe P um $\mathrm{m} - \pi(7) = 6$ (Fall A.2a)

```
babababacababacaabababab
                ababaca
```

- Match $\Rightarrow$ Setze $q = 1$ (Fall B.c)

```
babababacababacaabababab
                ababaca
```

- Mismatch $\Rightarrow$ Lookup $\pi(1) = 0$, Schiebe P um 1 weiter (Fall B.a)

```
babababacababacaabababab
                 ababaca
```

- Match $\Rightarrow$ Setze q = 1 (Fall B.c)
- …
- Match $\Rightarrow$ Setze q = 5 (Fall B.c)

Beispiel (IV)

```
babababacababacaababab
                ababaca
```

- Mismatch ⇒ Lookup $\pi(5) = 3$, Schiebe P um 2 ($q - \pi(5)$) weiter (Fall B.a)

```
babababacababacaababab
                  ababaca
```

- Match ⇒ Setze q = 1 (Fall B.c)
- …
- Match ⇒ Setze q = 5 (Fall B.c)

```
babababacababacaababab
                  ababaca
```

- Mismatch ⇒ Lookup $\pi(5) = 3$, Schiebe P um 2 ($q - \pi(5)$) weiter (Fall B.a)

```
babababacababacaababab
                    ababaca
```

- Match ⇒ Setze q = 1 (Fall B.c)
- …
- Match ⇒ Setze q = 4 (Fall B.c)

Beispiel (V)

```
babababacababacaababababab
                    abababaca
```

- Ende des Textes $\Rightarrow$ Terminierung

Komplexität

- In extremen Fällen deutlich geringer als Brute Force Algorithmus
- Zwischen zwei Vergleichen wird entweder Pattern verschoben (kann höchstens (n-m)-mal geschehen) oder Vergleichsposition wandert nach rechts (höchstens (n-1)-mal)
- Daher: Maximale Zahl an Vergleichen: $n - m + n - 1 + 1 = 2n - m$
- Da i.A. $n >> m$ ist dies O(n)
- Aufwand O(m) für Präfixfunktion
- Insgesamt also O(n+m)
- Da $n >> m$ angenommen werden

  $\Rightarrow$ Komplexität linear zur Länge des Textes, d.h. O(n)

Anmerkungen

- Weiterer Vorteil KMP-Algorithmus gegenüber Brute Force Ansatz:
  - Im Text wird nie zurückgegangen
  - D.h. Textzeiger wird zu keinem Zeitpunkt zurückgesetzt
  - Vorteilhaft bei Suche in sequentiellen Dateien
- Zusammenfassend: KMP-Algorithmus entschärft den ungünstigsten Fall erheblich
- Korrektheitsbeweis relativ aufwändig (nicht in dieser Vorlesung)

Übersicht

- Motivation
- Brute Force Algorithmus
- Algorithmus von Knuth-Morris-Pratt
- Algorithmus von Boyer-Moore

Idee

- Auswertung Muster von rechts nach links
- Ziel: Bei Mismatch Muster möglichst weit verschieben
- Nutze zur Verschiebung zwei Heuristiken:
  - Bad-Character
  - Good-Suffix

Bad-Character-Heuristik (I)

Falls beim Vergleich von $P[0 \ldots m-1]$ und $T[s \ldots s+m-1]$ (von rechts nach links) ein *Mismatch*

$$P[j] \neq T[s+j] \text{ für ein } j \text{ mit } 0 \leq j \leq m-1$$

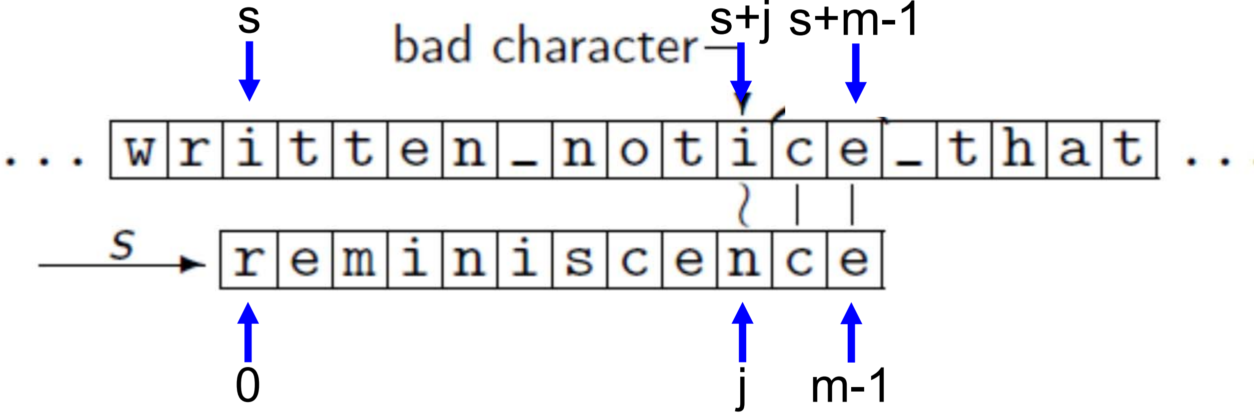
festgestellt wird, so schlägt die *bad-character* Heuristik eine Verschiebung des Musters um $j-k$ Positionen vor, wobei k der größte Index $(0 \leq k \leq m-1)$ ist mit $T[s+j] = P[k]$. Wenn kein k mit $T[s+j] = P[k]$ existiert, so sei $k = -1$. Man beachte, dass $j-k$ negativ sein kann.

[Steffen 2009]

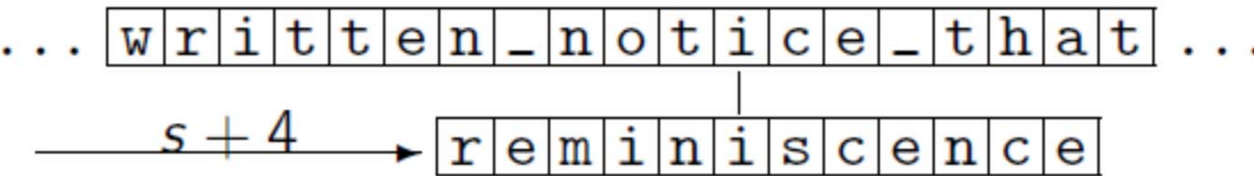
```
          s     s+j  s+m-1
##########xabcd#######
          ####yabcd
          0     j    m-1
```

- Suche am weitesten rechts im Muster stehendes x (Position k)
- Subtrahiere j-k

Bad-Character-Heuristik (II): Beispiel



- k = Größter Index im Muster mit Bad Character „i“ = 5
- Verschiebe um j – k = 9 – 5 = 4



[Steffen 2009]

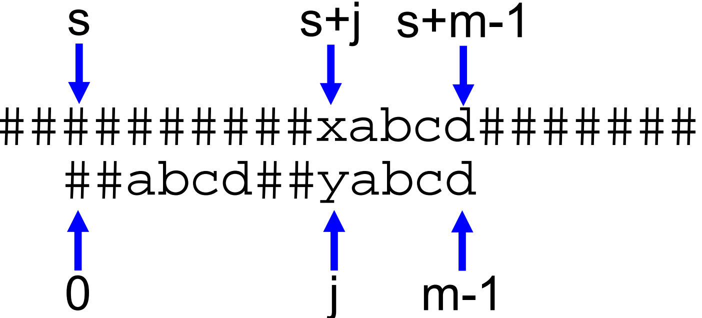
Good-Suffix-Heuristik (I)

Falls beim Vergleich von $P[0 \ldots m-1]$ und $T[s \ldots s+m-1]$ (von rechts nach links) ein *Mismatch*

$$P[j] \neq T[s+j] \text{ für ein } j \text{ mit } 0 \leq j \leq m-1$$

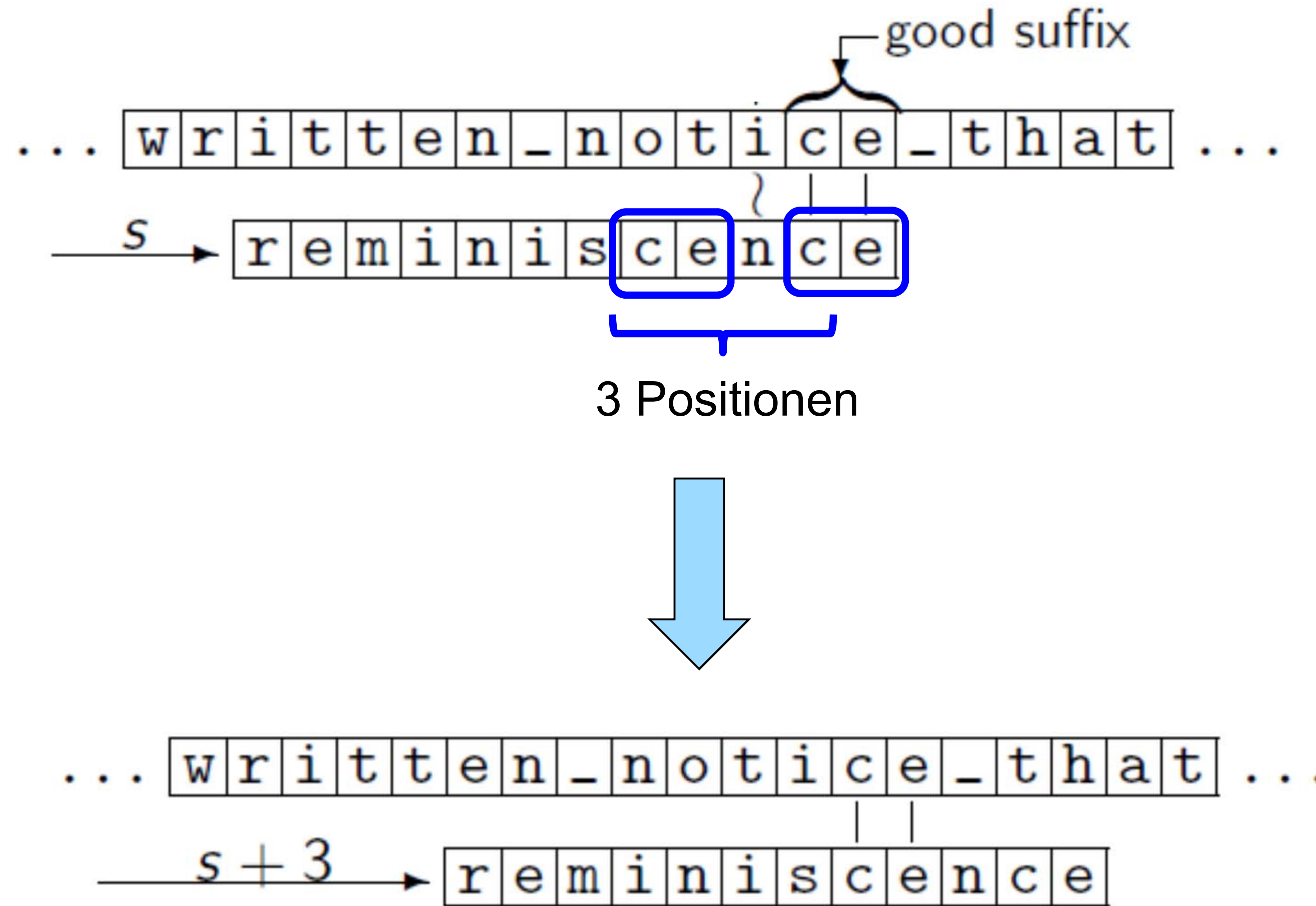
festgestellt wird, so wird das Muster so weit nach rechts geschoben, bis das bekannte Suffix $T[s+j+1 \ldots s+m-1]$ wieder auf ein Teilwort des Musters passt. Hierfür ist eine Vorverarbeitung des Musters notwendig, auf die wir hier aber nicht näher eingehen wollen.

[Steffen 2009]



– Verschiebe Muster, bis nächstes Good Suffix in Muster passt

Good-Suffix-Heuristik (II): Beispiel



[Steffen 2009]

Optimierung (I)

- Vorgestellte Variante Originalarbeit
- Wesentlicher Performancetreiber: Bad-Character-Heuristik
- Verbesserung (Horspool 1980):
  - Modifiziere Bad-Character-Heuristik
  - Soll immer positive Verschiebung vorschlagen
  - Damit:
    - Good-Suffix-Heuristik überflüssig
    - Vorverarbeitung einfacher
- Verbesserter Algorithmus: Boyer-Moore-Horspool-Algorithmus (BMH)

- falls $P[j] \neq T[s+j]$ für ein j $(0 \leq j \leq m-1)$

$$s \leftarrow s + m - 1 - k$$

  wobei k größter Index zwischen 0 und $m-2$
  mit $T[s+m-1] = P[k]$

- wenn kein k $(0 \leq k \leq m-2)$ mit $T[s+m-1] = P[k]$ existiert

$$s \leftarrow s + m$$

- Verschiebespanne:

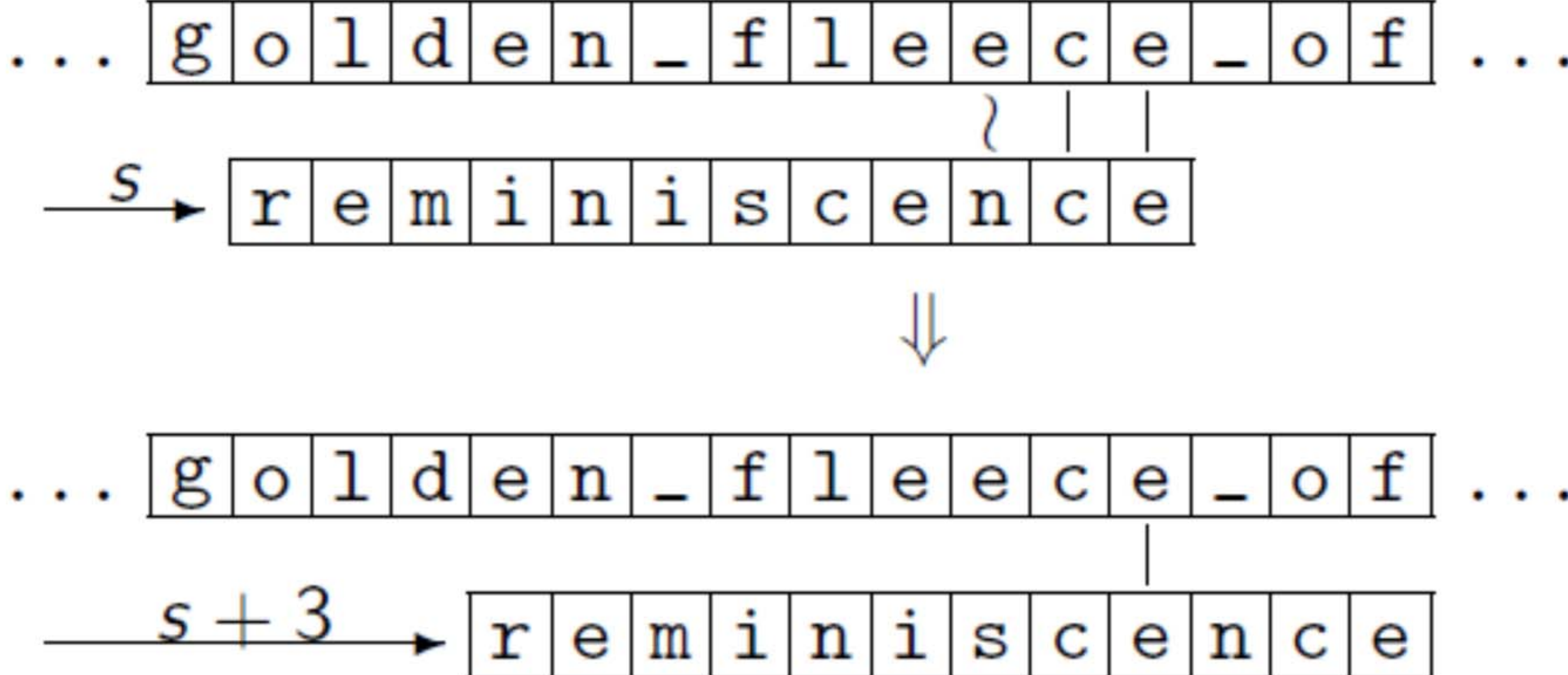
$$\lambda\big[T[s+m-1]\big] = \min\Big(\{m\} \cup \{m-1-k \mid 0 \leq k \leq m-2 \text{ und } T[s+m-1] = P[k]\}\Big)$$

  „Letztes Vorkommen vom Zeichen im Muster ohne letztes Zeichen“

- bei einem Match wird um $\lambda\big[T[s+m-1]\big]$ verschoben

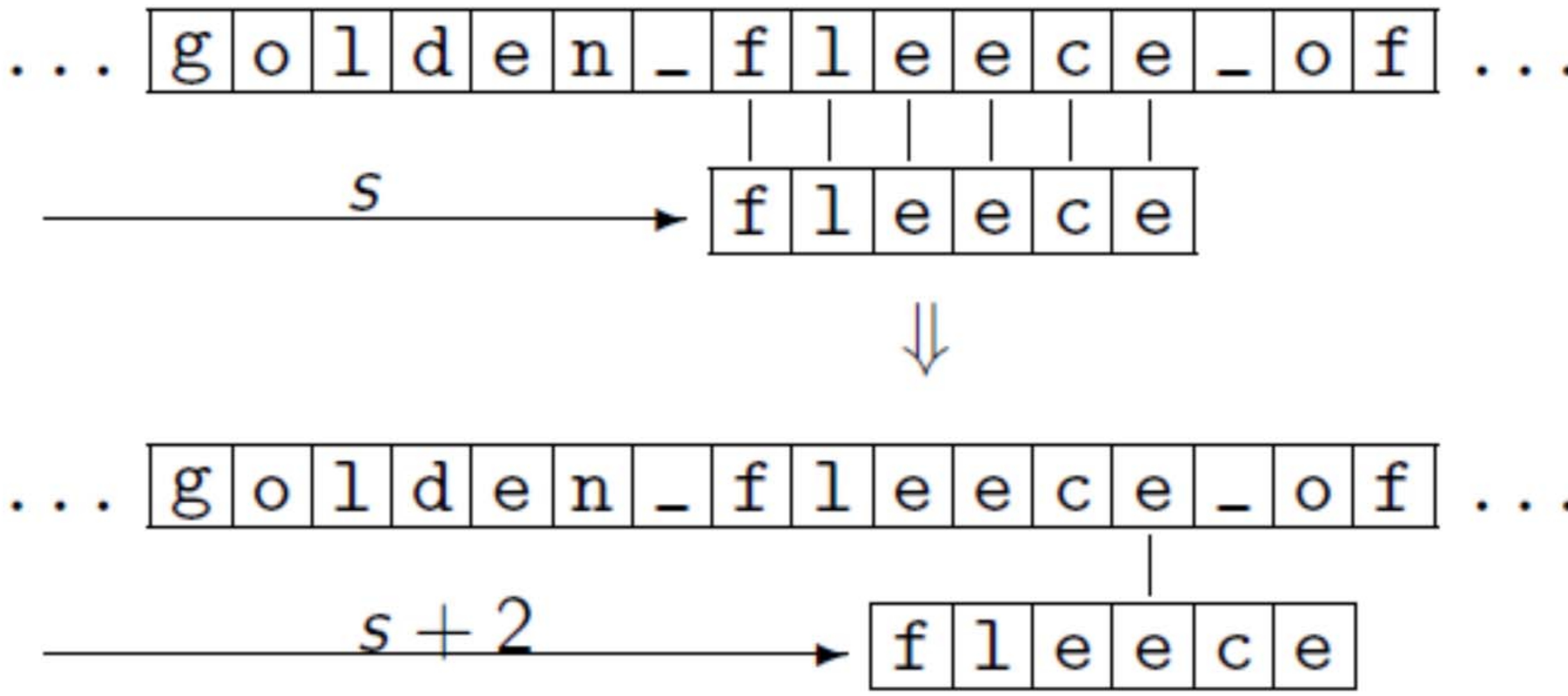
[Steffen 2009]

BMH: Verhalten bei Mismatch



[Steffen 2009]

BMH: Verhalten bei Treffer



[Steffen 2009]

BMH: Vorverarbeitung

Beispiel (Berechnung der Funktion λ)

computeLastOccurrenceFunction(P, Σ)

1 $m \leftarrow$ length$[P]$
2 for each character $a \in \Sigma$ do
3 $\quad \lambda[a] = m$
4 for $j \leftarrow 0$ to $m - 2$ do
5 $\quad \lambda\big[P[j]\big] \leftarrow m - 1 - j$
6 return λ

Die *worst-case*-Komplexität von computeLastOccurrenceFunction ist $\mathcal{O}\big(|\Sigma| + m\big)$.

[Steffen 2009]

BMH: Vorverarbeitung (II): Beispiel (I)

- Berechnung λ für Muster „ababaca“
- Initialisierung:
  - $m = 7$

	a	b	c
λ	7	7	7

- $j = 0$:
  - $\lambda[P(0)] = \lambda[a] = m - 1 - j = 6$

	a	b	c
λ	6	7	7

- $j = 1$:
  - $\lambda[P(1)] = \lambda[b] = m - 1 - j = 5$

	a	b	c
λ	6	5	7

BMH: Vorverarbeitung (III): Beispiel (II)

- $j = 2$:
  - $\lambda[P(2)] = \lambda[a] = m - 1 - j = 4$

	a	b	c
λ	4	5	7

- $j = 3$:
  - $\lambda[P(3)] = \lambda[b] = m - 1 - j = 3$

	a	b	c
λ	4	3	7

- $j = 4$:
  - $\lambda[P(4)] = \lambda[a] = m - 1 - j = 2$

	a	b	c
λ	2	3	7

BMH: Vorverarbeitung (IV): Beispiel (III)

- $j = 5$:
  - $\lambda[P(5)] = \lambda[c] = m - 1 - j = 1$

	a	b	c
λ	2	3	1

BMH: Algorithmus (I)

Beispiel (BMH-Matcher)

BMH-Matcher(T, P, Σ)

1 $n \leftarrow$ length[T]
2 $m \leftarrow$ length[P]
3 $\lambda \leftarrow$ computeLastOccurrenceFunction(P, Σ)
4 $s \leftarrow 0$
5 while $s \leq n - m$ do
6 $\quad j \leftarrow m - 1$
7 $\quad$ while $j \geq 0$ and $P[j] = T[s + j]$ do
8 $\quad\quad j \leftarrow j - 1$
9 $\quad$ if $j = -1$ then
10 $\quad\quad$ print "Pattern occurs with shift" s
11 $\quad s \leftarrow s + \lambda[T[s + m - 1]]$

[Steffen 2009]

BMH: Algorithmus (II): Beispiel (I)

- Gegeben seien der Text „bababababacababacaababababab“ und das Muster „ababaca“

```
babababacababacaababab
ababaca
```

  - Mismatch $\Rightarrow$ Schiebe P um $\lambda[b] = 3$ weiter, setze $j = 6$

```
babababacababacaababab
   ababaca
```

  - Match $\Rightarrow$ Setze $j = j - 1 = 5$, weitervergleichen
  - …
  - Match $\Rightarrow$ Setze $j = j - 1 = -1$

```
babababacababacaababab
   ababaca
```

  - Treffer ($j = -1$) $\Rightarrow$ „Muster gefunden an Position 3“
  - Schiebe P um $\lambda[a] = 2$ weiter , setze $j = 6$

```
babababacababacaababab
     ababaca
```

BMH: Algorithmus (III): Beispiel (II)

```
babababacababacaabababab
      ababaca
```

- Match $\Rightarrow$ Setze $j = j - 1 = 5$, weitervergleichen

```
babababacababacaabababab
      ababaca
```

- Mismatch $\Rightarrow$ Schiebe P um $\lambda[a] = 2$ weiter, setze $j = 6$

```
babababacababacaabababab
        ababaca
```

- Match $\Rightarrow$ Setze $j = j - 1 = 5$, weitervergleichen

```
babababacababacaabababab
        ababaca
```

- Mismatch $\Rightarrow$ Schiebe P um $\lambda[a] = 2$ weiter, setze $j = 6$

```
babababacababacaabababab
          ababaca
```

BMH: Algorithmus (IV): Beispiel (III)

```
bababababacababacaabababab
          ababaca
```

- Match ⇒ Setze $j = j - 1 = 5$, weitervergleichen
- …
- Match ⇒ Setze $j = j - 1 = -1$

```
bababababacababacaabababab
          ababaca
```

- Treffer ($j = -1$) ⇒ „Muster gefunden an Position 9“
- Schiebe P um $\lambda[a] = 2$ weiter, setze $j = 6$

```
bababababacababacaabababab
            ababaca
```

- Mismatch ⇒ Schiebe P um $\lambda[b] = 3$ weiter, setze $j = 6$

```
bababababacababacaabababab
               ababaca
```

- Match ⇒ Setze $j = j - 1 = 5$, weitervergleichen

BMH: Algorithmus (V): Beispiel (IV)

```
bababababacababacaabababab
                  ababaca
```

- Mismatch $\Rightarrow$ Schiebe P um $\lambda[a] = 2$ weiter, setze $j = 6$

```
bababababacababacaabababab
                    ababaca
```

- Match $\Rightarrow$ Setze $j = j - 1 = 5$, weitervergleichen

```
bababababacababacaabababab
                    ababaca
```

- Mismatch $\Rightarrow$ Schiebe P um $\lambda[a] = 2$ weiter, setze $j = 6$
- Ende Text erreicht ($s > n - m$) $\Rightarrow$ Terminierung

BMH: Komplexität

- Überprüfung auf Gültigkeit Verschiebung: O(m)
- Worst-Case:
  - O($n * m + |\Sigma|$)
  - Beispiel: T = a^n und P = a^m
- In „normalen“ Texten:
  - $m << n$ und $|\Sigma| << n$
  - Damit: O(n)
- Für große Alphabete/kleine Muster wird meist O(n/m) erreicht, d.h. zumeist ist nur jedes m-te Zeichen zu inspizieren

Zusammenfassung (I)

- Motivation:
  - Beispiele
  - Präfix, Suffix, Verschiebung

- Brute Force Algorithmus:
  - „Probiere durch“
  - Mustervergleich links nach rechts um 1
  - Worst Case „unangenehm“

- Algorithmus von Knuth-Morris-Pratt:
  - Mustervergleich links nach rechts
  - Längere Verschiebungen
  - Präfixfunktion

Zusammenfassung (II)

- Algorithmus von Boyer-Moore:
  - Mustervergleich rechts nach links
  - Heuristiken:
    - Bad-Character
    - Good-Suffix
  - Modifiziere Bad-Character-Heuristik
  - Boyer-Moore-Horspool-Algorithmus

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
B J H S W N P Y D Z M M U F P K M P N C
U C C M R S Q Z A M I E Y R M N G V Y T
P M E O Y J W T B J C N A P L U X I M B
N F A H U B T N B M F E E C Q T F Z K N
C Z V J K H M C L W F N B U L H H U R Q
C F Y K K W C R S I D T E A F M V E J L
M H E S B U C T X Z G J F G A O P H D I
C O Z W N P H F A Y H I G Z X R O L U K
P E N Z B N U M A M B E S I S R A D K O
K P L O J N T Y J H S J F L Q I V Q Y B
Y L N U K R V Q I D K I G S K S R U P U
P U L T Z E D Z L M N K M U Q P Q W O N
X N I G K T I C H J Q V C Y Y R C A J K
Z O A U F T I U A Z Y K D E Z A Q S X R
N B C Y I A E Z Y L Q H B H C T F N L A
M Y X U A P H G B U O Y N U O T H T L U
E R O O M R E Y O B F H D W M C D Z G D
N N O I Q J P Q Q Y R M F U Q M X H U B
C V S Y A Y K G C T W I M X U P A O I Q
A N Q O D W Y U F S O U U D R J T Q O X
```

Übungen (II)

- **Aufgabe 2**
  - Welche Komplexität hat der Brute-Force-Algorithmus?
  - Was ist der Unterschied zwischen dem Brute-Force-Algorithmus und dem Algorithmus von Knuth-Morris-Pratt?
  - Welche Algorithmen benötigen eine Vorverarbeitung?
  - Welche Algorithmen vergleichen von links nach rechts?
  - Was unterscheidet den BMH-Algorithmus von den anderen?
  - Was ist die Good-Suffix-Heuristik?

Übungen (III)

- **Aufgabe 3**
  Wenden Sie auf das folgende Beispiel den Brute-Force-Algorithmus, den Algorithmus von Knuth-Morris-Pratt und den Algorithmus von Boyer-Moore-Horspool an!
  Vergleichen Sie die Anzahl der durchgeführten Vergleiche!
  Text: `ABCAABABAABABC`
  Muster: `ABABC`

Algorithmen und Datenstrukturen

Teil 14: Codierung

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2023

Gliederung

- Grundbegriffe
- Code-Erzeugung
- Code-Sicherung
- Lineare Codes
- Nicht-binäre Codes

Definitionen (I)

- Seien $A = \{a_1, \dots, an\}$ Menge elementarer Zeichen
- A heißt Alphabet
- Aneinanderreihung von Zeichen aus Alphabet A heißen Wort w
- A^* := Menge aller Wörter über A
- Beispiel:
  - Sei $A = \{a, b, c\}$
  - Beispiele für Wörter über A: $a, b, abba, ccbbaa, \dots$
  - Gegenbeispiele (keine Wörter über A): $ae, xa, \dots$
  - $A* = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc,$
    $aaa, aab, aac, aba, abb, abc, aca, acb, acc, \dots \}$
- Anzahl der Zeichen eines Wortes w: Länge von w, Notation $|w|$

Definitionen (II)

- Spezialfall:
  - $A = \{0,1\}$
  - Worte heißen Binärwort
  - A^* Menge der Binärwörter
- Gegeben sei ein Binärwort $b_{n-1}b_{n-2} \ldots b_0$ der Länge n
- Dann:
  - b_{n-1} MSB (Most significant bit)
  - b_0 LSB (Least significant bit)

Definitionen (III)

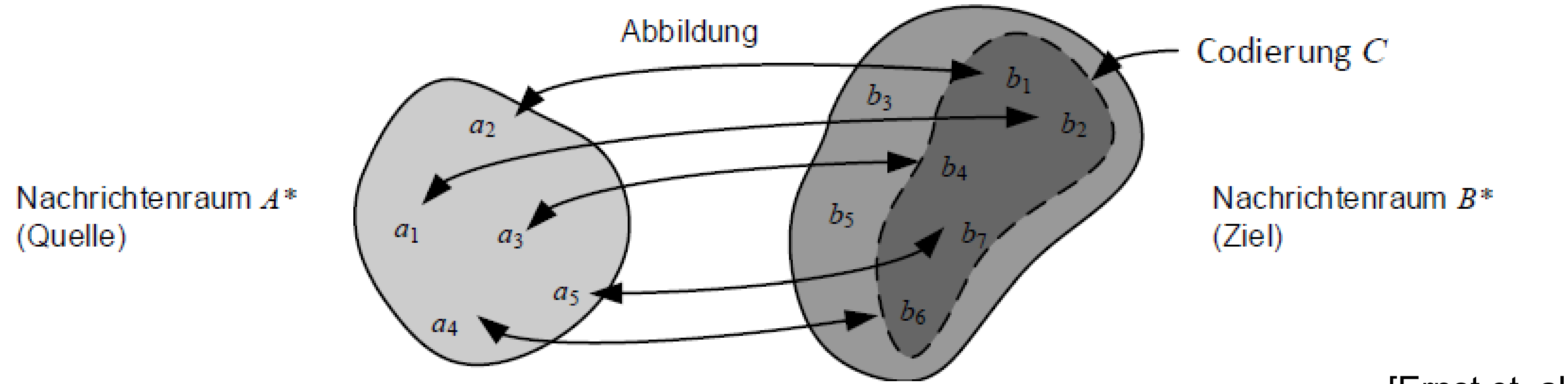
- Entropie: Mittlerer Informationsgehalt pro Zeichen einer Quelle
- Formaler:
  - Gegeben Quelle $X = \{x_1, \dots, x_n\}$, Wahrscheinlichkeit $p(x)$ für alle $x \in X$
  - Entropie H (eigentlich großes Eta) definiert als

$$H(X) = \sum_{i=1}^{n} p(x_i) \cdot log_2 \left(\frac{1}{p(x_i)}\right) = -\sum_{i=1}^{n} p(x_i) \cdot log_2 \left(p(x_i)\right)$$

  - Dabei gilt: $0 \cdot log_2 0 \coloneqq 0$ und $a \cdot log_2 \left(\frac{a}{0}\right) \coloneqq \infty$
- Anmerkungen:
  - Es gilt immer $H(X) \geq 0$
  - Entropie wird maximal, wenn alle Zeichen gleich wahrscheinlich ($H_0 = \log_2(n)$)
  - Entropie deutsche Sprache ca. $4{,}1$
  - Maximaler Wert bei 26 Buchstaben $\log_2(26) \approx 4{,}7$

Codierung

- Gegeben:
  - Nachrichtenraum A^* über Alphabet $A = \{a_1, \dots, an\}$ (Quelle)
  - Nachrichtenraum B^* über Alphabet $B = \{b_1, \dots, b_m\}$ (Ziel)
- Codierung C ist bijektive Abbildung $A^* \rightarrow B^*$
- Beachte: $C \subseteq B^*$

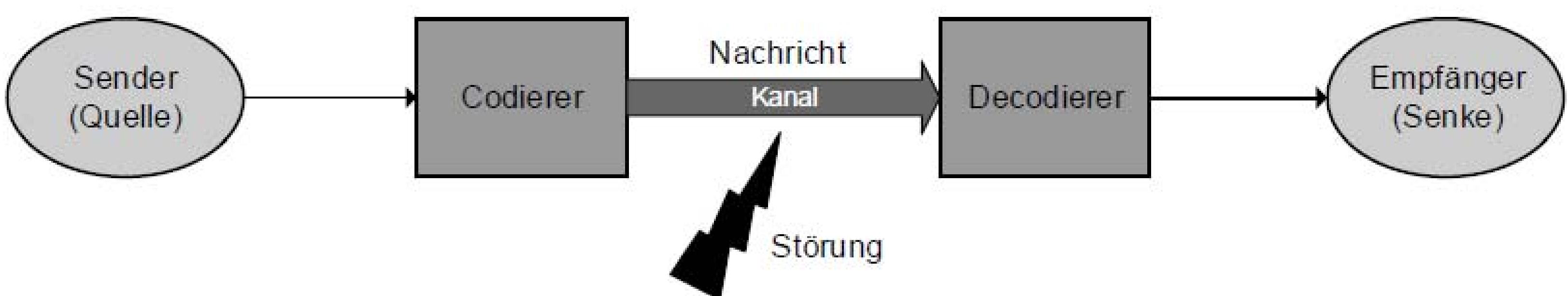


[Ernst et. al. 2016]

- Ist $B = \{0,1\}$, dann heißt C Binärcodierung

Übertragung/Anforderungen

- Übertragung:



[Ernst et. al. 2016]

- Anforderungen an Codierung:
  - Möglichst kompakte Darstellung
  - Unempfindlich gegen Störungen
  - Maschinell leicht zu verarbeiten

Mittlere Wortlänge

- Def.: Mittlere Wortlänge L
- Gegeben:
  - Zeichen $X = \{x_1, \dots, x_n\}$
  - l_i Wortlänge von Zeichen x_i
  - p_i Auftretenswahrscheinlichkeit von Zeichen x_i
- Dann:

$$L = \sum_{i=1}^{n} p_i \cdot l_i$$

- Shannonsches Codierungstheorem:

  $H \leq L$ („Entropie ist immer kleiner gleich der mittleren Wortlänge“)
- Begründung: Entropie maximal, wenn alle Zeichen gleich wahrscheinlich

Code-Redundanz

- Redundanz R eines Codes ist definiert als

$$R = L - H \; [Bit\ pro\ Zeichen]$$

- Anschaulich: Anteil einer Nachricht, der im statistischen Sinn keine Information enthält
- Geringe Redundanz:
  - Schnelle Übertragung
  - Platzsparend
- Andererseits:
  - Redundanz kann Störsicherheit bringen
  - Fehlererkennung, Fehlerbehebung

Quellen-Redundanz

- Quellen-Redundanz R_Q ist definiert als

$$R_Q = H_0 - H \; [Bit\; pro\; Zeichen]$$

- Anschaulich: Abweichung mittlerer Informationsgehalt von Q vom maximal möglichen Informationsgehalt bei gleichem Alphabet, aber anderer Auftretenswahrscheinlichkeit der Zeichen

Beispiel (I): BCD-Code (I)

- BCD: Binary Coded Decimal
- Dezimalsystem nach BCD:
  - Dezimalziffer wird zu 4 Binärzeichen (Tetradencode)
  - Umwandlung jeder Dezimalziffer separat
- Beispiel: Erste 15 Zahlen

Dezimal	Binär	BCD
0	0000	0000 0000
1	0001	0000 0001
2	0010	0000 0010
3	0011	0000 0011
4	0100	0000 0100
5	0101	0000 0101
6	0110	0000 0110
7	0111	0000 0111

Dezimal	Binär	BCD
8	1000	0000 1000
9	1001	0000 1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101

- Pseudotetraden: In BCD-Code nicht verwendete Tetraden (1010 bis 1111)

Beispiel (II): BCD-Code (II)

- Beispiele:
  - 745 -> 0111 0100 0101
  - 63,25 -> 0110 0011, 0010 0101
  - 0,1 -> 0000, 0001
- Bewertung:
  - Vorteile:
    - Anwendungen mit Rechengenauigkeit
    - Stellengenaue Ergebnisse möglich
    - Binärsystem hingegen (z.B. 0,1 nur näherungsweise darstellbar)
  - Nachteile:
    - Mehr Platzbedarf
    - Geringere Rechengeschwindigkeit
- Begriffe (nicht nur bei BCD-Code):
  - Nutzwort: Code, der tatsächlich genutzt wird
  - Fehlerwort: Code, der nicht genutzt wird (hier: Pseudotetraden)

Beispiel (III): ASCII-Code (I)

- ASCII: American Standard Code for Information Interchange
- Erste Standardisierung 1968: ANSI X3.4 bzw. ISO 8859/1.2
- Enthält alle Buchstaben, Ziffern und Steuerzeichen
- Ursprünglich 7-Bit-Code mit 128 Zeichen

BIT 5–7 1–4	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0/ 0000	NUL	DLE	SP	0	@	P	‘	p
1/ 0001	SOH	DC1	!	1	A	Q	a	q
2/ 0010	STX	DC2	“	2	B	R	b	r
3/ 0011	ETX	DC3	#	3	C	S	c	s
4/ 0100	EOT	DC4	$	4	D	T	d	t
5/ 0101	ENQ	NAK	%	5	E	U	e	u
6/ 0110	ACK	SYN	&	6	F	V	f	v
7/ 0111	BEL	ETB	’	7	G	W	g	w
8/ 1000	BS	CAN	(	8	H	X	h	x
9/ 1001	HT	EM	)	9	I	Y	i	y
A/ 1010	LF	SUB	*	:	J	Z	j	z
B/ 1011	VT	ESC	+	;	K	[	k	{
C/ 1100	FF	FS	,	<	L	\	l	|
D/ 1101	CR	GS	−	=	M	]	m	}
E/ 1110	SO	RS	.	>	N	^	n	~
F/ 1111	SI	US	/	?	O	_	o	DEL

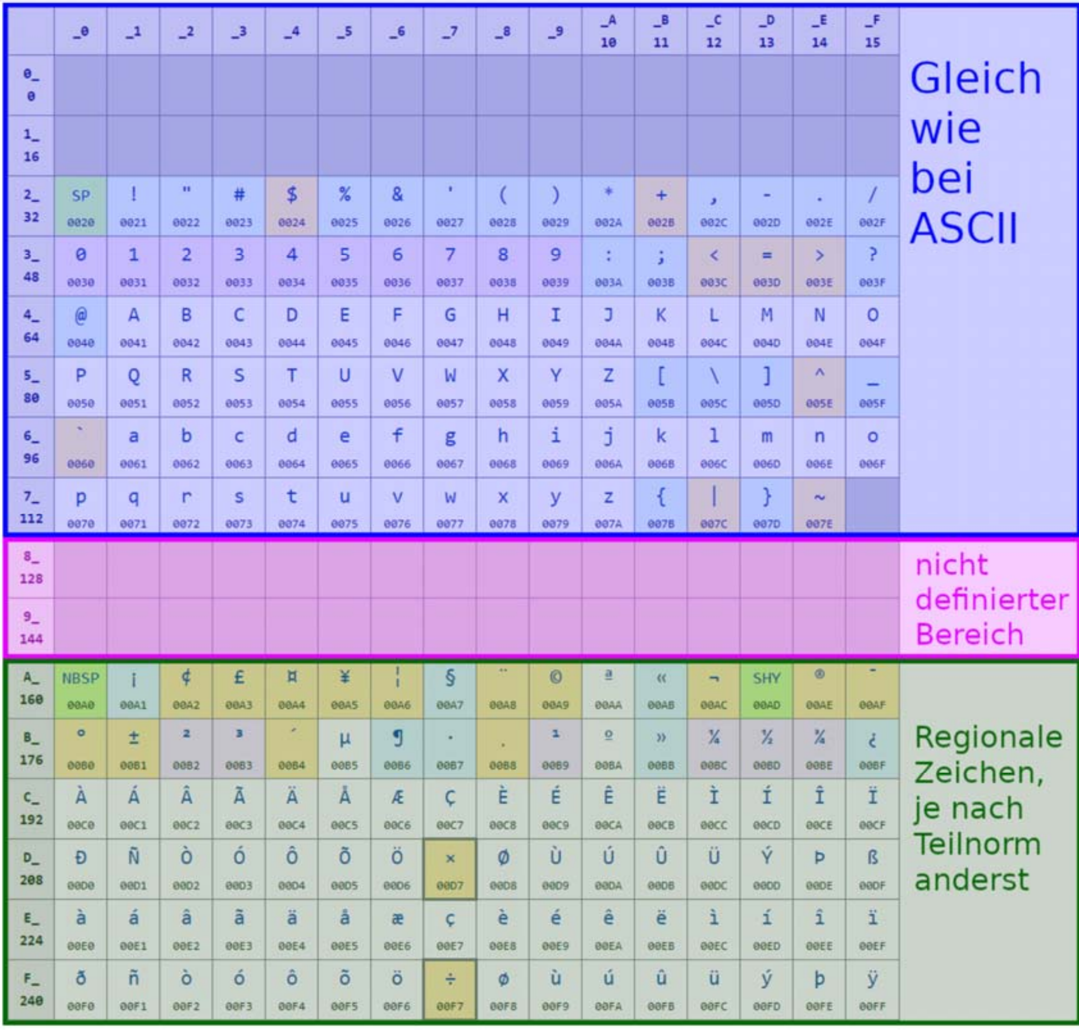
[https://www.cfd.tu-berlin.de/Lehre/EDV1/skripte/alles/node77.html]

ASCII-Code (II)

- (De)codierung erfolgt durch „Ablesen“ in Tabelle
- Beispiele:
  - 100 0111 -> G
  - ? -> 011 1111
- Nachteil: Wesentliche Zeichen europäischer Sprachen fehlen
- Daher:
  - Nutzung achtes Bit
  - Neben Sonderzeichen auch mathematische Zeichen und grafische Zeichen
- Trotz allem: Vielzahl an (verschiedenen) Sonderzeichen nicht handhabbar
- Ausweg ISO 8859:
  - 8 Bit ASCII Zeichensatz
  - Erste 128 Zeichen gleich (entsprechen 7 Bit ASCII)
  - Zweite 128 Zeichen variabel zur Anpassung an Sprachraum

ASCII-Code (III)

- Aufbau und normierte ISO 8859 Zeichensätze:



ISO 8859

-1	*Latin-1*, Westeuropäisch
-2	*Latin-2*, Mitteleuropäisch
-3	*Latin-3*, Südeuropäisch
-4	*Latin-4*, Nordeuropäisch
-5	Kyrillisch
-6	Arabisch
-7	Griechisch
-8	Hebräisch
-9	*Latin-5*, Türkisch
-10	*Latin-6*, Nordisch
-11	Thai
~~-12~~	(existiert nicht)
-13	*Latin-7*, Baltisch
-14	*Latin-8*, Keltisch
-15	*Latin-9*, Westeuropäisch
-16	*Latin-10*, Südosteuropäisch

- Für Deutschland: ISO-8859-1 (Latin-1)

[https://www.marcelwicki-softwaredesign.ch/2018/08/08/zeichencodierungen/]

Unicode (I)

- Idee: Alle (Sonder-)Zeichen aller Sprachen in einem Code
- Erweiterung von 8 auf 32 Bit
- D.h. $2^{32} \approx 4{,}29$ Milliarden Zeichen
- Bezeichnung eines 16 Bit codierten Unicodezeichens:
  - U+XXXX
  - Jedes X hexadezimale Ziffer
- Kompatibilität zu ASCII-Code:
  - ASCII: U+0000 bis U+007F
  - ISO 8859-1: U+0080 bis U+00FF

Unicode (II)

- Unicode weist Zeichen Code zu
- Keine Aussage über Länge der Codierung
- UTF: Unicode Transformation Format
  - UTF-32:
    - Immer 32 Bit
    - Sehr speicherplatzintensiv
  - UTF-16 (Windows, OS-X):
    - Für „gebräuchliche“ Zeichen 2 Byte pro Zeichen
    - 4 Byte für „exoterische“ Zeichen
  - UTF-8 (www, E-Mail, Unix):
    - Zwischen einem und vier Byte pro Zeichen
    - Vorteil:
      - 1 Byte-Codierung entspricht 7 Bit ASCII
      - Sehr effizient bei lateinischen Zeichen (gegenüber 2 Byte in UTF-16)
      - Bei anderen Schriften typischerweise 3 Byte (gegenüber 2 Byte in UTF-16)

Übersicht

- Grundbegriffe
- Code-Erzeugung
- Code-Sicherung
- Lineare Codes
- Nicht-binäre Codes

- Gegeben:
  - Alphabet mit Zeichen
  - Auftretenshäufigkeiten der Zeichen
- Gesucht:
  - Code mit minimaler mittlerer Codewortlänge

- Siehe Kapitel 4:
  - Huffman-Algorithmus
  - Kurze Wiederholung (wenn nicht mehr bekannt):
    - Präfixcode
    - (Minimale) mittlere Codewortlänge
    - Ablauf Algorithmus

Algorithmus von Shannon und Fano

- Gegeben:
  - Menge von Zeichen mit Auftretenshäufigkeiten
- Vorgehen:
  - Sortiere Zeichen nach ihrer Häufigkeit
  - Teile Zeichen in zwei Gruppen, so dass Summe der Häufigkeiten in Teilgruppen möglichst gleich
  - Erweitere Code einer Gruppe um 0, der anderen um 1 (bzw. trage Gruppen in Binärbaum ein)
  - Enthält Teilgruppe mehr als 1 Zeichen, fahre rekursiv fort

Beispiel (I)

- Gegeben:

Zeichen	A	N	B	T	E	H
Häufigkeit	16	20	8	6	42	8

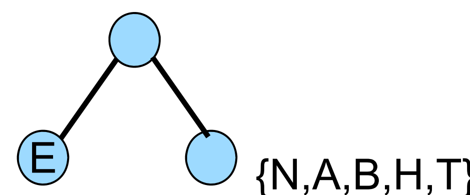
- Ordne nach Häufigkeit:

Zeichen	Häufigkeit	
E	42	0
N	20	1
A	16	1
B	8	1
H	8	1
T	6	1

Summe = 100

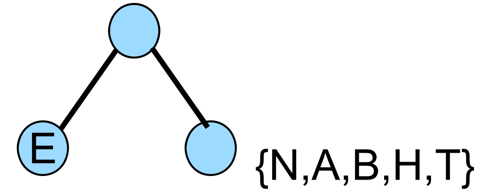
An nächsten an Hälfte: 42

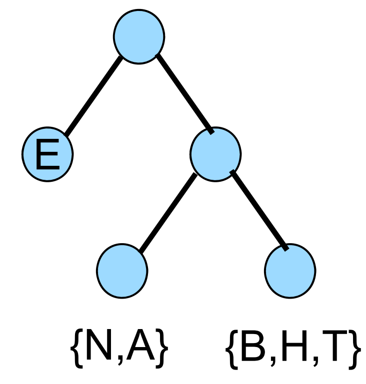
Also: In {E} und {N,A,B,H,T} unterteilen



Beispiel (II)

Zeichen	Häufigkeit	
E	42	0
N	20	1 0
A	16	1 0
B	8	1 1
H	8	1 1
T	6	1 1





Summe = 58

Unterteilungen:

- {N} und {A,B,H,T} : 20 und 38
- {N,A} und {B,H,T} : 36 und 22

An nächsten an Hälfte: 36

Also: In {N,A} und {B,H,T} unterteilen

Beispiel (III)

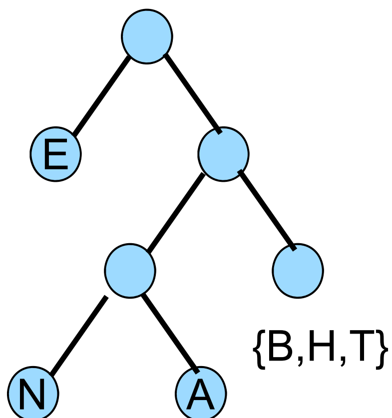
Zeichen	Häufigkeit	
E	42	0
N	20	1 0 0
A	16	1 0 1
B	8	1 1
H	8	1 1
T	6	1 1

E

{N,A} {B,H,T}

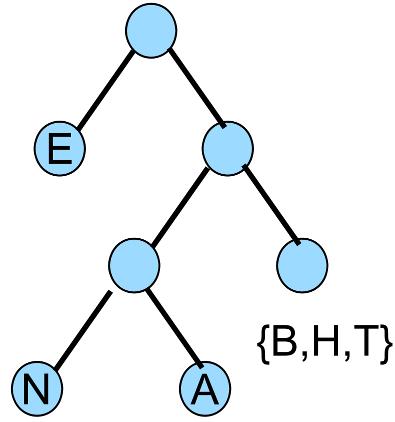
Gruppe {N,A}

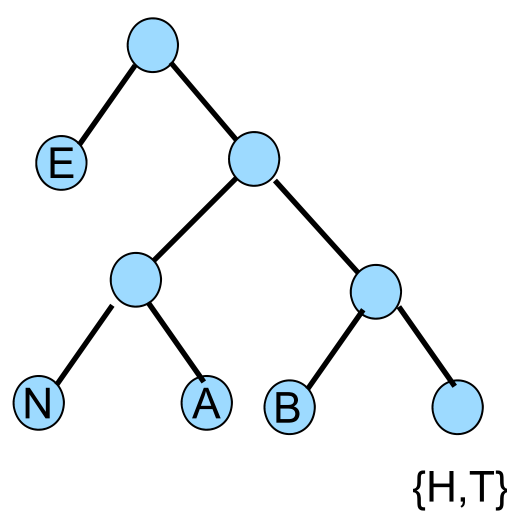
Nur eine Unterteilung möglich



Beispiel (IV)

Zeichen	Häufigkeit	
E	42	0
N	20	1 0 0
A	16	1 0 1
B	8	1 1 0
H	8	1 1 1
T	6	1 1 1





Unterteile {B,H,T}, Summe = 22

Unterteilungen:

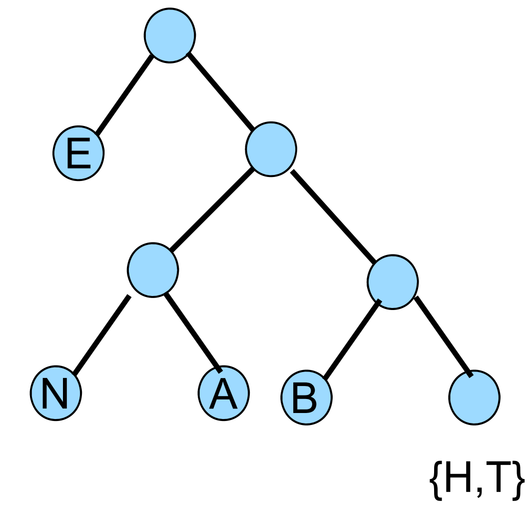
- {B} und {H,T} : 8 und 14
- {B,H} und {T} : 16 und 6

An nächsten an Hälfte: 8 und 14

Also: In {B} und {H,T} unterteilen

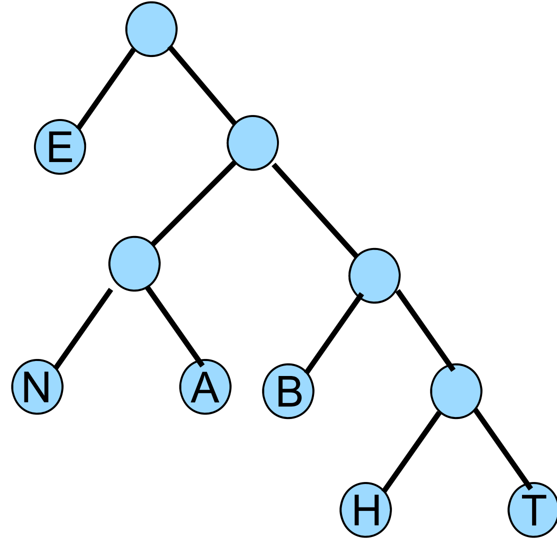
Beispiel (V)

Zeichen	Häufigkeit	
E	42	0
N	20	1 0 0
A	16	1 0 1
B	8	1 1 0
H	8	1 1 1 0
T	6	1 1 1 1



Gruppe {H,T}

Nur eine Unterteilung möglich



Bezug Baum - Code:

- Linke Kante mit 0 beschriften
- Rechte Kante mit 1 beschriften
- Pfad Wurzel-Blatt ergibt Code

Übersicht

- Grundbegriffe
- Code-Erzeugung
- Code-Sicherung
- Lineare Codes
- Nicht-binäre Codes

Stellen- und Hamming-Distanz

- Seien x und y Binärwörter gleicher Länge
- Stellen-Distanz oder Hamming-Distanz $d(x, y)$:= Anzahl Stellen, an denen sich x und y unterscheiden
- Berechnung: $x\ XOR\ y$, dann Anzahl Einsen zählen
- Bei korrekter Übertragung ist $x = y$ und $d(x, y) = 0$
- Hamming-Distanz $h(C)$ eines Codes C: Minimale Stellendistanz eines Codes

Beispiel

- Gegeben:

Ziffer	Code C_1
1	001
2	010
3	011
4	100

- Wie groß ist die Hamming-Distanz des Code und was bedeutet dies für die Fehlererkennung?

- Berechne Stellendistanzen von je zwei Codewörtern
- Am übersichlichsten in Matrix eintragen:

	001	010	011	100
001	-	-	-	-
010	2	-	-	-
011	1	1	-	-
100	2	2	3	-

- $h(C_1) = 1$

  $\Rightarrow$ Fehler lassen sich nicht in jedem Fall erkennen (es gibt Nutzwörter zwischen denen keine Fehlerwörter liegen)

Fehlererkennung und –korrektur (I)

- Es gilt:
  - Sind max. $h-1$ Bit in Wort fehlerhaft $\Rightarrow$ Erkennung möglich
  - Sind max. $(h-1)/2$ Bit in Wort fehlerhaft $\Rightarrow$ Korrektur möglich
- $h = 1$:
  - Keine Fehlererkennung, da Fehler wieder zu gültigem Codewort führen
- $h = 2$:
  - Fehlererkennung (von 1 Bit-Fehlern)
  - Keine Korrektur möglich, da
    - ungültiges Codewort, das in mind. einem Fall mehr als ein Codewort mit Stellendistanz 1 als Nachbarn hat
    - Damit: Keine Entscheidung, in welche Richtung zu korrigieren ist
- $h = 3$ oder $h = 4$:
  - 1-Bit-Fehler können korrigiert werden
  - 2-Bit- bzw. 3-Bit-Fehler können erkannt werden
- $h = 5$ oder $h = 6$:
  - 2-Bit-Fehler können korrigiert werden
  - 4-Bit- bzw. 5-Bit-Fehler können erkannt werden
-

Jeweils „entweder oder“

Fehlererkennung und –korrektur (II)

- Beispiel („entweder oder“):
  - Betrachte Codewörter $x = 10101110$ und $y = 00101011$
  - Code habe $h = 3$
  - x und y unterscheiden sich an genau drei Stellen
  - Annahme: Codewort x wurde gesendet, Fehler bei Übertragung
  - Fall 1: Einzelnes Bit falsch:
    - Durch Kippen des MSB wird aus x $x' = 00101110$
    - Das nächstliegende gültige Codewort zu x‘ ist x mit $d(x, x') = 1 \Rightarrow$ Fehler wird korrigiert
  - Fall 2: Zwei Bits bei Übertragung verändert
    - Übertragung ergibt $x'' = 00101111$
    - Fehlererkennung, da kein gültiges Codewort (nächstes gültiges muss Mindestabstand 3 haben)
    - Keine Fehlerkorrektur möglich
      - da x'' auch aus y entstehen könnte (durch Fehler im dritten Bit von rechts)
      - Auf Empfängerseite nicht unterscheidbar

Fehlererkennung und –korrektur (III)

- Beispiel (Ursprünglicher Code C_1 und alternativer 3-Bit-Code C_2):

Ziffer	Code C_1	Code C_2
1	001	000
2	010	011
3	011	101
4	100	110

- $h(C_2) = 2$, denn:

	000	011	101	110
000	-	-	-	-
011	2	-	-	-
101	2	2	-	-
110	2	2	2	-

- 1-Bit-Fehler können in Code C_2 erkannt werden

Fehlererkennung und –korrektur (IV)

- Codeüberdeckungsproblem: Wie kann optimaler Code mit vorgegebener Hamming-Distanz generiert werden?
- „Optimal“ := Möglichst kurze Codewörter

m-aus-n-Codes (I)

- Blockcode mit Wortlänge n
- Jedes Wort hat
  - genau m Einsen und
  - $n-m$ Nullen
- Beispiele:

Ziffer	2-aus-5-Code	1-aus-10-Code
0	00011	0000000001
1	00101	0000000010
2	00110	0000000100
3	01001	0000001000
4	01010	0000010000
5	01100	0000100000
6	10001	0001000000
7	10010	0010000000
8	10100	0100000000
9	11000	1000000000

m-aus-n-Codes (II)

- m-aus-n-Codes haben Hamming-Distanz 2:
  - Jedes Codewort gleich viele Einsen $\Rightarrow$ Zwei Codewörter müssen sich in mindestens zwei Stellen unterscheiden
- Gegeben $n, m \Rightarrow$ Anzahl Codewörter $\binom{n}{m}$

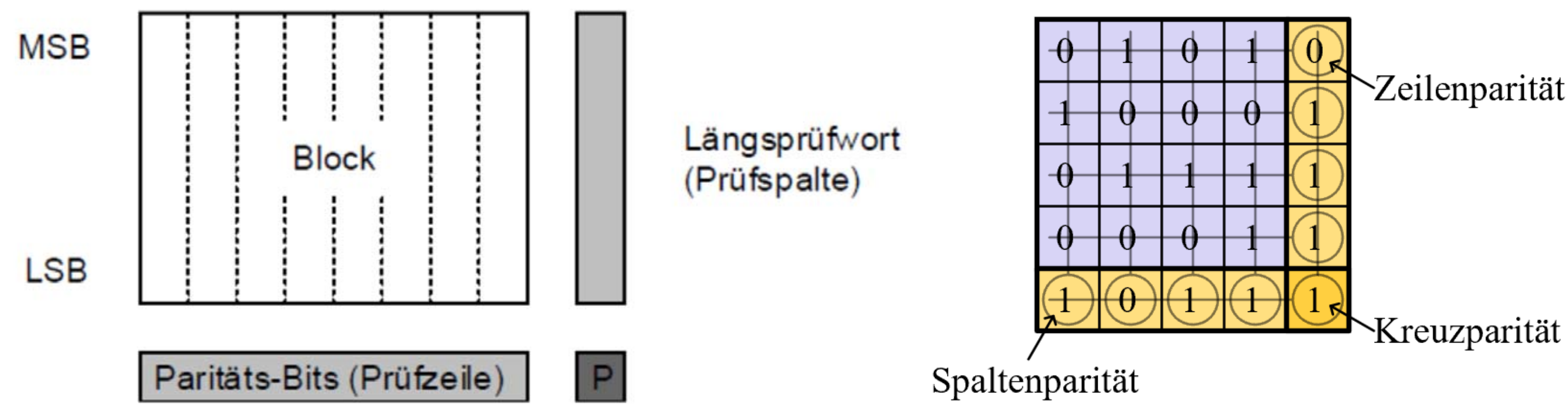
Einfacher Paritätscode

- Füge an Codewort Paritätsbit (synonym: Prüfbit) als LSB an, so dass Anzahl Einsen jedes Codeworts gerade (gerade Parität, even parity) oder ungerade (ungerade Parität, odd parity) ist
- Beispiel:
  - Gerade Parität
  - `10110101` → `101101011`
  - `00100100` → `001001000`

  Paritätsbit
- Erkennung von 1-Bit-Fehlern
- Keine Korrektur möglich

Kreuzparität (I): Konzept

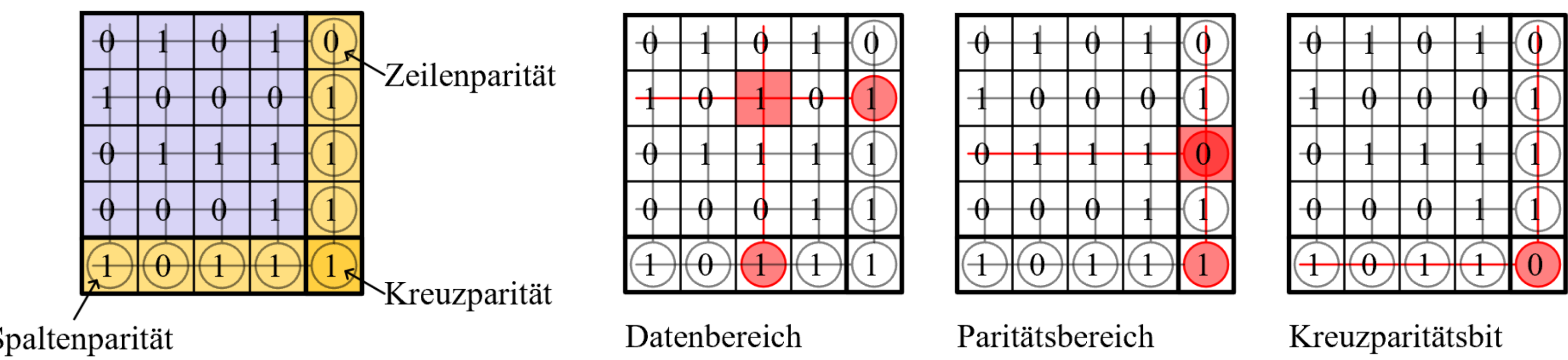
- Idee: Prüfzeilen und –spalten



- Gebe jedem Wort Paritätsbit
- Fasse Paritätsbits von Wörtern zu Prüfzeile zusammen (LRC, Longitudinal Redundancy Check)
- Nach Block von k Wörtern: Bilde Parität der Zeilen des Blocks, fasse diese zu Prüfspalte zusammen (VRC, Vertical Redundancy Check, Längsprüfwort)
- Bilde aus Prüfzeile und –spalte Paritätsbit P (ergänzt Anzahl Einsen im Block auf gerade Anzahl)

Kreuzparität (II): 1-Bit-Fehler

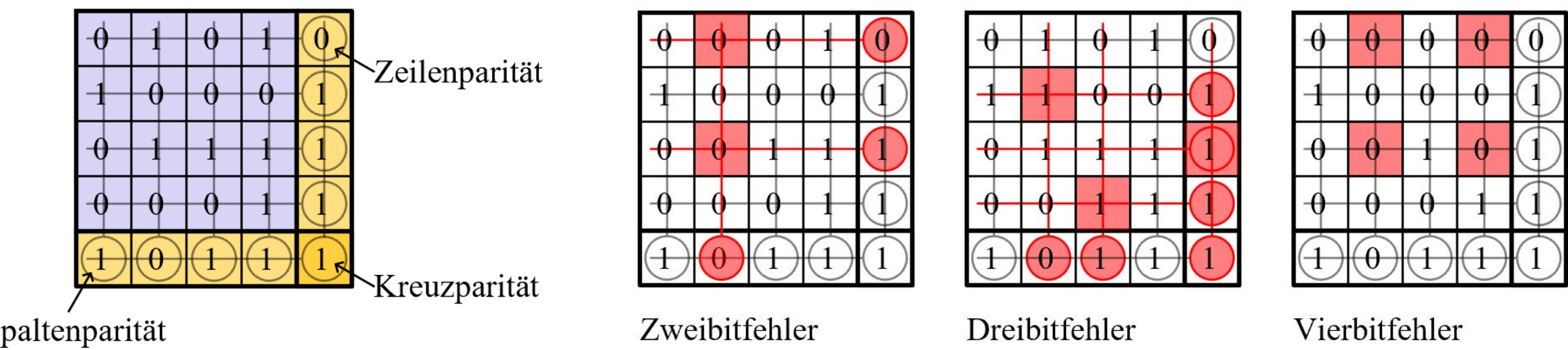
- Erkennung und Korrektur von 1-Bit-Fehlern



- Fall 1: Fehler im Datenbereich
  - Sowohl in Paritätszeile wie auch –spalte je 1 Bit verletzt Parität
  - Positionen dieser beiden Bits definiert eindeutig Position im Block
  - Invertiere dieses Bit
- Fall 2: Fehler in Prüfwort, nicht in P
  - Paritätsverletzung in Prüfzeile oder –spalte, nicht in beiden
  - Keine Korrektur notwendig
- Fall 3: Fehler in P
  - Sowohl Prüfzeile wie auch –spalte in Ordnung
  - Keine Korrektur notwendig

Kreuzparität (III): 2-/3-/4-Bit-Fehler

- Erkennung von 2- und 3-Bit-Fehlern



- 2-Bit-Fehler (beide in Daten):
  - Paritätszeile oder –spalte verletzt Parität
  - Sind beide Fehler in einer Zeile/Spalte ⇒ Fehler hebt sich auf
- 3-Bit-Fehler (zwei in Daten, einer in Prüfspalte)
  - Paritätsverletzung in Prüfzeile und –spalte
  - Ungünstige Fälle denkbar: sehen aus wie 1-Bit-Fehler
- 4-Bit-Fehler (alle in Daten bei (un)günstiger Lage):
  - Sowohl Prüfzeile wie auch –spalte in Ordnung
  - Fehler werden nicht erkannt

Kreuzparität (IV): Eigenschaften

DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart
Campus Horb

- Redundanz, abhängig von Wortlänge s und Anzahl k Worte pro Block:

$$R = \frac{k+s+1}{k} \quad [Bit\ pro\ Wort]$$

- Hamming-Distanz $h = 4$:
  - Entweder 1-Bit-Fehler korrigieren
  - oder 2-Bit-Fehler und 3-Bit-Fehler erkennen

Kreuzparität (V): Beispiel (I)

Bsp. 3.8 Absicherung des Wortes INFORMATIK mit Kreuzparitätskontrolle

Zur binären Codierung des Wortes INFORMATIK wird der ASCII-Code (siehe Tabelle 3.2) benützt, wobei die Anzahl der Einsen zu einer geraden Zahl in einem Paritätsbit und nach jedem vierten Wort in einem Längsprüfwort ergänzt wird. Bei der Übertragung seien 1-Bit Fehler aufgetreten, so dass das Wort ANFORMAPIK empfangen wird. Wie oben beschrieben, lassen sich diese beiden Übertragungsfehler erkennen und korrigieren, das korrekte Wort INFORMATIK lässt sich also wieder restaurieren. Der Decodiervorgang ist in Tabelle 3.6 dargestellt.

Tabelle 3.6 Das im ASCII-Zeichensatz codierte Wort INFORMATIK wird mit Paritätsbits und Längsprüfwörtern in gerader Parität seriell übertragen. Die Übertragung erfolgte in Blöcken, wobei die beiden ersten Blöcke je vier und der dritte Block nur noch zwei Zeichen enthält, er wurde daher mit Nullen zur vollen Länge von vier Zeichen aufgefüllt. Es sind zwei Fehler (A statt I in Spalte 1 und P statt T in Spalte 8) aufgetreten. Diese lassen sich lokalisieren und korrigieren. Die zur Fehleridentifikation führenden Paritätsbits sowie die entsprechenden Bits der Längsprüfworte sind durch Pfeile markiert, die fehlerhaft übertragenen Bits sind grau unterlegt

	empfangene Daten				Längs-prüfwort	empfangene Daten				Längs-prüfwort	empfangene Daten				Längs-prüfwort	
MSB	1	1	1	1	0	1	1	1	1	0	1	1	0	0	0	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	
	0	1	0	1	1 ⇐	0	1	0	0	1	1	1	0	0	0	
	0	1	1	1	1	0	1	0	0	0 ⇐	0	0	0	0	0	
	0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	
LSB	1	0	0	1	0	0	1	1	0	0	1	0	0	0	1	
	1	0	1	1	1	1	0	0	1	0	1	1	0	0	0	Paritätsbits
	⇑								⇑							
	A	N	F	O		R	M	A	P		I	K				empfangener Text
	⇑								⇑							
	I								T							Korrekturen

[Ernst et. al. 2016]

Kreuzparität (VI): Beispiel (II)

Bsp. 3.9 Kreuzparitätskontrolle hat eine Hamming-Distanz von vier

Dies soll am ersten Datenblock von Bsp. 3.8 erläutert werden, bei dem also nur der Wortteil INFO im ASCII-Code übertragen wurde. Beispiele für Einzel-, Doppel-, Dreifach- und Vierfachfehler sind in Tabelle 3.7 gezeigt. Am Dreifach-Fehler in Tabelle 3.7c mit nur zwei falschen Paritätsbits erkennt man die mögliche Verwechslung mit einem 1-Bit Fehler; der 3-Bit Fehler würde also bei Einsatz als korrigierender Code als 1-Bit Fehler interpretiert und falsch korrigiert. Ein vierfacher Fehler ist wie in 3.7d zu sehen nicht in jedem Fall erkennbar.

Tabelle 3.7 Die zur Fehleridentifikation führenden Paritätsbits sowie die entsprechenden Bits der Längsprüfworte sind durch Pfeile markiert, die fehlerhaft übertragenen Bits sind grau unterlegt

(a) 2-Bit Fehler mit zwei falschen Paritätsbits

1	1	1	1	0
0	0	0	0	0
0	0	0	0	0
0	1	0	0	1
0	1	1	1	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
⇑			⇑	

(b) 2-Bit Fehler mit vier falschen Paritätsbits

1	1	1	1	0	
0	0	0	0	0	
0	0	0	1	0	⇐
0	1	0	1	1	⇐
0	1	1	1	1	
0	1	1	1	1	
1	0	0	1	0	
1	0	1	1	1	
⇑			⇑		

(c) 3-Bit Fehler mit zwei falschen Paritätsbits

1	1	1	1	0	
1	0	0	0	0	⇐
0	0	0	0	0	
0	1	0	0	1	
0	1	1	1	1	
0	1	1	1	1	
1	0	0	1	0	
1	0	1	1	1	
			⇑		

(d) 4-Bit Fehler: alle Paritätsbits korrekt

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

[Ernst et. al. 2016]

Tetraden mit drei Paritätsbits (I)

- Ordne Codewort mehr als 1 Paritätsbit zu
- Vorteil: Jedes Wort kann für sich geprüft werden
- Beispiel: Tetraden mit 3 Paritätsbits

$b_6 b_5 b_4 b_3$ $b_2 b_1 b_0$

Code Paritätsbit

Ziffer	Code
0	0000111
1	0001100
2	0010010
3	0011001
4	0100001
5	0101010
6	0110100
7	0111111
8	1000000
9	1001011

- Regeln Bildung der Paritätsbits:
  - $b_2 = 1$, wenn Anzahl Einsen in b_6, b_5 und b_4 gerade
  - $b_1 = 1$, wenn Anzahl Einsen in b_6, b_5 und b_3 gerade
  - $b_0 = 1$, wenn Anzahl Einsen in b_6, b_4 und b_3 gerade

Tetraden mit drei Paritätsbits (II)

- Drei Paritätsbits $\Rightarrow 2^3 = 8$ Zustände („Kein Fehler“ und „Fehler in einer der sieben Stellen“)
- D.h. Code kann 1-Bit-Fehler erkennen und korrigieren
- Ermittlung fehlerhafter Stelle: (r = richtig, f = falsch)

Fehlerhaftes Bit	b_2	b_1	b_0
0	r	r	f
1	r	f	r
2	f	r	r
3	r	f	f
4	f	r	f
5	f	f	r
6	f	f	f

Verletzt ein Prüfbit p die Parität $\Rightarrow p$ selbst fehlerhaft

Tetraden mit drei Paritätsbits (III)

- Entropie:
  - Annahme: Alle 10 Ziffern gleiche Wahrscheinlichkeit $p = 0{,}1$
  - $H = log_2\left(\frac{1}{p}\right) = log_2(10) \approx 3{,}3219$
- Redundanz:
  - $R = L - H \approx 7 - 3{,}3219 = 3{,}6781\ Bit/Zeichen$
  - Setzt sich zusammen aus 3 Prüfbits
  - Und 0,6781 Bit/Zeichen Informationsbits (nur 10 von 16 möglichen verwendet)
- Hamming-Distanz:
  - Für Informationsbits: $h = 1$
  - Zusammen mit Prüfbits: $h = 3$
  - Damit: Entweder
    - Erkennung und Korrektur von 1-Bit-Fehlern
  - Oder
    - Erkennung (aber keine Korrektur) von 2-Bit-Fehlern

Übersicht

- Grundbegriffe
- Code-Erzeugung
- Code-Sicherung
- Lineare Codes
- Nicht-binäre Codes

- Def.: Körper
  Körper ist Menge K mit zwei inneren zweistelligen Verknüpfungen „+“ und „·“ (Addition und Multiplikation) mit:
  - Existenz neutrales Element 0, d.h. $\forall k \in K: k + 0 = k$
  - Existenz neutrales Element 1, d.h. $\forall k \in K \setminus \{0\}: k \cdot 1 = k$
  - Distributivgesetzen:

    $a \cdot (b + c) = a \cdot b + a \cdot c$ für alle $a, b, c \in K$.

    $(a + b) \cdot c = a \cdot c + b \cdot c$ für alle $a, b, c \in K$.

- Beipiele:
  - Menge der rationalen, reellen und komplexen Zahlen
  - Boolescher Körper:
    - $K = \{0,1\}$
    - Konjunktion AND als Multiplikation
    - Antivalenz XOR als Addition

Exkurs: Vektorraum

- Sei V Menge, K Körper, $\oplus: VxV \rightarrow V$ innere zweistellige Verknüpfung (Vektoraddition) und $\otimes: KxV \rightarrow V$ äußere zweistellige Verknüpfung (Skalarmultiplikation)
- $(V, \oplus, \otimes)$ heißt Vektorraum (über Körper K), wenn folgendes gilt:

Vektoraddition:

V1: $u \oplus (v \oplus w) = (u \oplus v) \oplus w$ (Assoziativgesetz)

V2: Existenz eines neutralen Elements $0_V \in V$ mit $v \oplus 0_V = 0_V \oplus v = v$

V3: Existenz eines zu $v \in V$ inversen Elements $-v \in V$ mit $v \oplus (-v) = (-v) \oplus v = 0_V$

V4: $v \oplus u = u \oplus v$ (Kommutativgesetz)

Skalarmultiplikation:

S1: $\alpha \odot (u \oplus v) = (\alpha \odot u) \oplus (\alpha \odot v)$ (Distributivgesetz)

S2: $(\alpha + \beta) \odot v = (\alpha \odot v) \oplus (\beta \odot v)$

S3: $(\alpha \cdot \beta) \odot v = \alpha \odot (\beta \odot v)$

S4: Neutralität des Einselements $1 \in K$, also $1 \odot v = v$

- Unterraum: Teilmenge, die die Eigenschaften „mitnimmt“

Definitionen (I)

- Def.: Linearer Code (bzw. linearer (s,r)-Code))
  - Sei B^s Vektorraum über booleschem Körper B
  - Sei $C \subseteq B^s$ Code mit $n = 2^r$ Codewörtern der Länge s
  - C heißt linearer Code g.d.w C ist Unterraum von B^s

- Def.: Gewicht eines Codeworts
  - Sei C linearer Code, $x \in C$ Codewort
  - Gewicht $g(x) \coloneqq Anzahl\ Einsen\ in\ x$
- Satz: „Gewicht = Stellendistanz zum Nullvektor“
  - Sei **0** das nur aus Nullen bestehende Codewort
  - Dann: $g(x) = d(x, \mathbf{0})$

- Def.: Minimales Gewicht eines Codes
  - Sei C linearer Code
  - Dann: $g_{min} \coloneqq \min\{g(x) \mid x \in C, x \neq \mathbf{0}\}$

Definitionen (II)

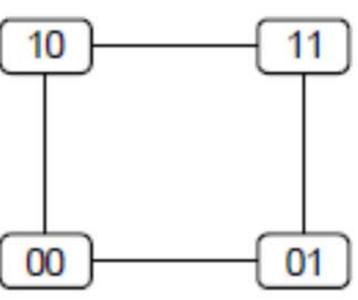
- Satz:
  - Sei C linearer Code
  - Hamming-Distanz h = Minimalgewicht von C $(h = g_{min})$

- Anwendung:
  - Reduktion Berechnungsaufwand Hamming-Distanz
  - Für Codes mit n Codewörtern allgemein: Ermittlung aller Stellendistanzen $\frac{(n^2-n)}{2}$ Vergleiche
  - Für lineare Codes: Berechne n Stellendistanzen zu **0**, bilde Minimum

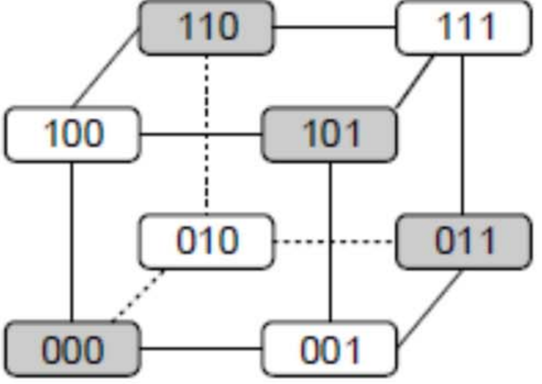
Geometrische Interpretation (I)

- Anordnung $n = 2^r$ Codewörter als Ecken r-dimensionalen Würfels/Kugel
- Anm.: Zugrundeliegender Raum ist diskret $\Rightarrow$ Würfel = Kugel
- Zeichne Kante, wenn sich Codewörter um genau $1\ Bit$ unterscheiden
- Beispiele:

(a) Für $r = 1$ enthält der Code $2^1 = 2$ Codewörter. Der zugehörige eindimensionale Würfel ist eine Gerade



(b) Für $r = 2$ enthält der Code $2^2 = 4$ Codewörter. Der zugehörige zweidimensionale Würfel ist ein Quadrat

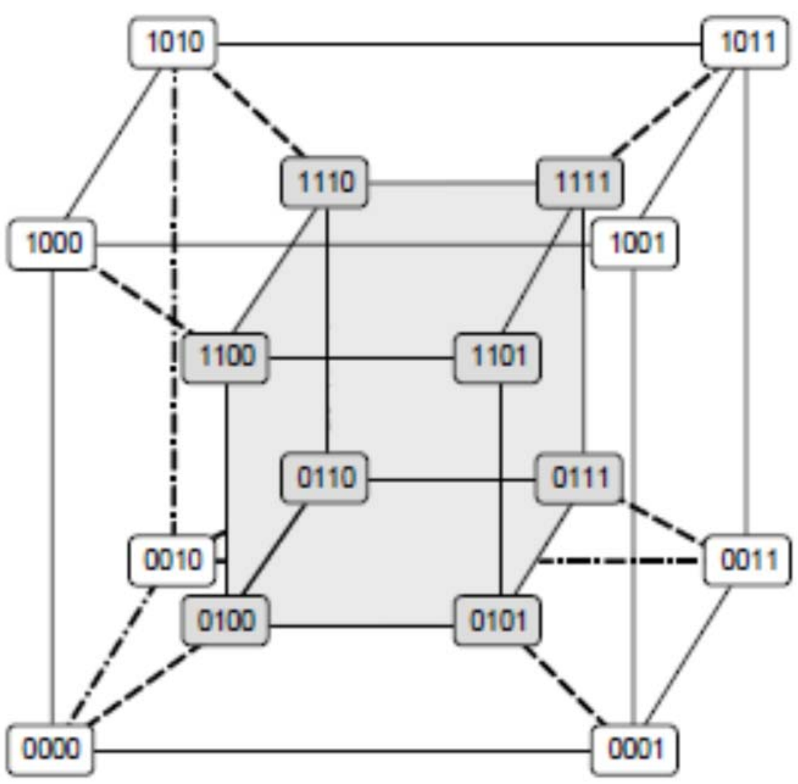


(c) Für $r = 3$ enthält der Code $2^3 = 8$ Codewörter. Die geometrische Anordnung aller aus drei Bit bildbaren Wörter führt dann zu einem dreidimensionalen Würfel. Die grau unterlegten Wörter bilden einen Code mit Hamming-Distanz $h = 2$, ebenso die hell unterlegten Wörter

- Hamming-Distanz h: Kürzester Weg von einem Nutzwort zu anderem Nutzwort führt über mindestens h Kanten

Geometrische Interpretation (II)

- Idee Anordnung auch für $r > 3$ (Hyperwürfel, Hypercube)
- Beispiel:



Perfekte Codes (I)

- Ziel: Konstruktion Code, bei dem bis zu x-Bit-Fehler korrigierbar sein sollen
- Basis: Geometrische Interpretation
- Ordne Codewörter so an,
  - dass jedes Codewort Mittelpunkt einer Kugel mit Radius x ist
  - und sich diese Kugeln nicht überlappen
  - Dann: $h = 2x + 1$
- Code mit 1-Bit-Fehler korrigieren:
  - $h = 3$ notwendig
  - Kugeln müssen mindestens Radius $x = 1$ haben
  - Obergrenze Anzahl Codewörter: $n_x \leq \frac{V_{ges}}{V_x}$
    - V_{ges}: Gesamtvolumen (=Anzahl Codewörter)
    - V_x: Volumen (=Anzahl Codewörter) Kugel mit Radius x

Perfekte Codes (II)

- Gesamter Raum B^s hat Volumen 2^s
- Volumen Kugel mit Radius 1:
  - Umfasst Mittelpunkt und alle über eine Kante erreichbaren Codewörter
  - D.h. $V_1 = 1 + s$
  - Damit gilt: $n_1 \leq \frac{2^s}{V_1} = \frac{2^s}{1+s}$
- Verallgemeinerung:
  - Bei x pro Codewort zu korrigierenden Fehlern: Setze an Stelle V_1 das Volumen V_x
  - Abzählen aller Codewörter, die von betrachtetem Punkt aus über maximal x Kanten erreichbar sind: $V_x = 1 + \sum_{i=1}^{x} \binom{s}{i}$
- Damit: Obere Grenze n_x s-stelliger Codewörter eines Codes mit Korrigierbarkeit von x Fehlern: $n_x \leq \frac{2^s}{V_x} = \frac{2^s}{1+\sum_{i=1}^{x} \binom{s}{i}}$

Perfekte Codes (III)

- Formel
  - besagt: n_x ist Obergrenze Anzahl Codewörter
  - Sagt nichts aus, ob n_x erreichbar ist und wie diese Codes konstruiert werden können
- Def.: Perfekter Code
  - Sei C linearer Code mit Fehlerkorrigierbarkeit x
  - C heißt perfekter Code g.d.w. n_x wird erreicht
- Beispiel:
  - Für $s = 3, x = 1$ ergibt sich $n_1 \leq \frac{2^3}{(1+3)} = 2$
  - Für Code $\{000,111\}$ ist $x = 1$ und $h = 3$
  - Code ist perfekt

Perfekte Codes (IV)

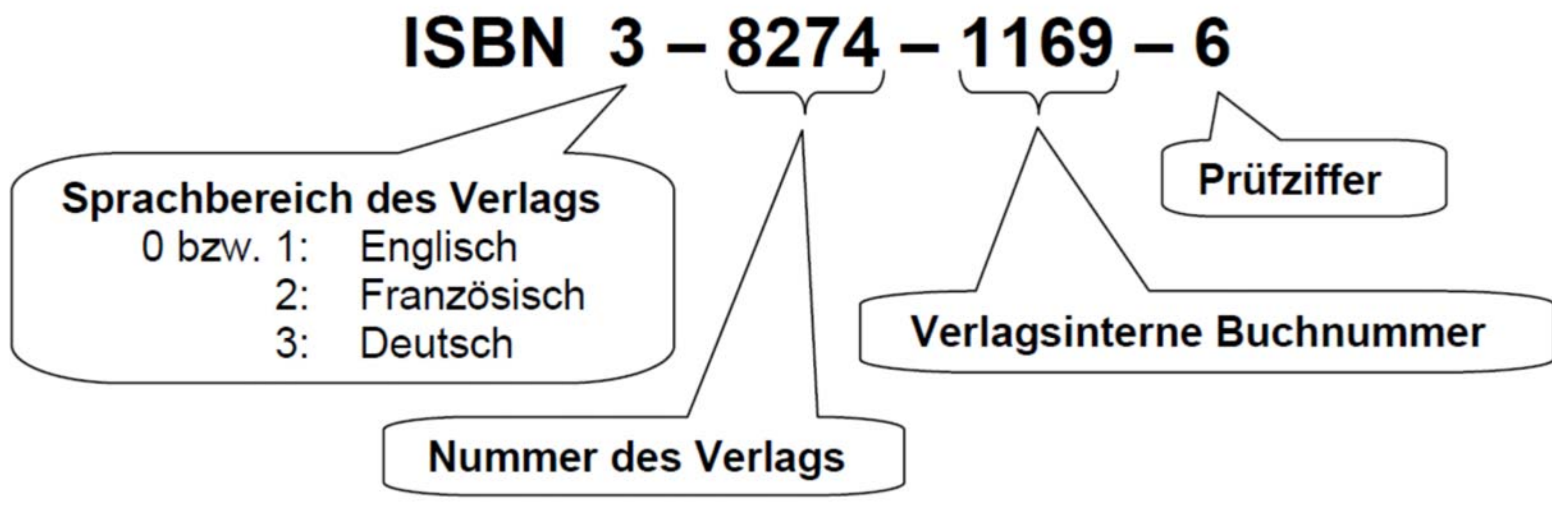
- Beispiel:
  - Für $s = 8, x = 1, h = 5$
  - $n_2 \leq \frac{2^8}{\left(1+\sum_{i=1}^{2}\binom{8}{i}\right)} = \frac{256}{(1+8+28)} \approx 6{,}9$
  - Für $s = 8$ gibt es einige Code mit $h = 5$
  - Diese umfassen aber maximal 4 Codewörter, sind nicht linear
  - 8-stelliger Code mit vier Codewörtern $\Rightarrow$ bestenfalls Hamming-Distanz 4 möglich

Übersicht

- Grundbegriffe
- Code-Erzeugung
- Code-Sicherung
- Lineare Codes
- Nicht-binäre Codes

ISBN (I)

- Ursprünglich: 10-stelliger Code



- Berechnung Prüfziffer P:
  - Jeder der ersten neun Ziffern wird Multiplikator zugeordnet (von 10 abwärts)
  - Ziffer wird mit Multiplikator multipliziert und Produkte addiert
  - P Zahl, die zu Summe addiert werden muss, so dass nächstes ganzes Vielfaches von 11 erreicht wird
  - $P \in \{0, 1, \ldots, 10\} \Rightarrow$ Schreibe X für 10

ISBN (II)

- Beispiel:

ISBN	3	8	2	7	4	1	1	6	9
Multiplikator	10	9	8	7	6	5	4	3	2
Produkt	30	72	16	49	24	5	4	18	18
Summe	30	102	118	167	191	196	200	218	236

30 + 72 + 16 + 49 + 24 + 5 + 4 + 18 + 18 = 236

236 : 11 = 21 Rest 5 ➔ 11 – 5 = 6 ➔ 6 ist die berechnete Prüfziffer.

- Aktueller ISBN-Code 13-stellig
- Bisherige Ziffern bleiben
- Zusätzlich: Präfix vorangestellt, je nach Buch 978 oder 979
- Prüfziffernberechnung wie bei EAN-Code

EAN

- EAN: Europäische Artikel Nummer
  - 13-stellig
  - Letzte Ziffer Kontrollziffer
  - Berechnung:
    - Ordne ersten 12 Ziffern alternierend Faktoren 1 und 3 zu
    - Bilde Produkte und Summe der Produkte
    - Prüfziffer ist Zahl, die zur Summe addiert werden muss, um nächstes gnazes Vielfaches von 10 zu erhalten
- Beispiel:

EAN	9	7	8	3	8	2	7	4	1	1	6	9
Multiplikator	1	3	1	3	1	3	1	3	1	3	1	3
Produkt	9	21	8	9	8	6	7	12	1	3	6	27

Die Summe der Produkte beträgt 117, die Ergänzung zur nächsten Zehnerzahl ist 3.

IBAN (I)

- IBAN: International Bank Account Number (https://www.iban.de/)
  - Bis zu 34 Zeichen
  - In Deutschland 22 Zeichen: $DEp_1p_2b_1b_2b_3b_4b_5b_6b_7b_8k_1k_2k_3k_4k_5k_6k_7k_8k_9k_{10}$ mit
    - k_1 bis k_{10} und mit b_1 bis b_8 ehemalige Kontonummer und BLZ
    - p_1 und p_2 Prüfziffern
  - Berechnung Prüfziffern:
    - Stelle Länderkürzel und auf 0 gesetzte Prüfziffern ganz nach rechts
    - Setze Buchstaben der Länderkennung auf Position im Alphabet $+9$
    - Berechne für diese Zahl Rest der Division durch 97
    - Prüfziffern: $98 - Rest$
  - Prüfung:
    - Stelle Länderkürzel und Prüfziffern ganz nach rechts
    - Setze Buchstaben der Länderkennung auf Position im Alphabet $+9$
    - $Zahl\ modulo\ 97$ muss 1 ergeben

IBAN (II)

- Beispiel:

Für die deutsche Bankleitzahl 711 500 00 und Kontonummer 215 632 soll die IBAN berechnet werden. Kombination der beiden Zahlen mit rechts angehängtem Länderkürzel und Prüfziffern Null lautet: 711 500 00 0000 215 632 DE 00.
Nach dem Ersetzen des Kürzels DE mit den Positionen im Alphabet plus 9 (D=13, E=14) erhält man: 711 500 00 0000 215 632 1314 00.
Der Rest bei Division durch 97 liefert: 711 500 00 0000 215 632 1314 00 mod 97 = 49.
Die gesuchten Prüfziffern lauten also 98 − 49 = 49, die gesamte IBAN ergibt sich zu DE 49 711 500 00 0000 215 632.

Eine Validierung der IBAN liefert 711 500 00 0000 215 632 1314 49 mod 97 = 1, diese ist also korrekt.

- Gewährung Sicherheit:
  - Arithmetik mit bis zu 36-stelligen Zahlen notwendig
  - Mit gängigen Programmiersprachen (Datentypen bis 64 Bit) nicht möglich

Zusammenfassung (I)

- Grundbegriffe:
  - Definitionen
  - Entropie
  - Codierung
  - Code-Redundanz
  - Quellen-Redundanz
  - Beispiele: BCD, ASCII, Unicode

- Code-Erzeugung:
  - Code-Erzeugungsproblem
  - (Huffmann-Algorithmus)
  - Algorithmus von Fano und Shannon

Zusammenfassung (II)

- Code-Sicherung:
  - Stellen- und Hamming-Distanz
  - Fehlererkennung- und –korrektur
  - m-aus-n-Codes
  - Paritätscodes: Bit und Kreuzparität
  - Tetraden mit drei Paritätsbits

- Lineare Codes:
  - Linearer Code
  - Gewicht Codewort
  - Minimales Gewicht eines Codes
  - Geometrische Interpretation
  - Perfekte Codes

- Nicht-binäre Codes:
  - ISBN, EAN, IBAN

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
Q Y B C P V D S D F H W L F O R L Y E L
F D J U B G W H S M S P L D T Q T C T U
H E M M Z L E A C T K Q M P E T H P U X
R I N F R V K N T E A U E P S F G B H P
S P S Y L D Z N R Y Q F V R W Q G T M Z
F O H T M B W O O W J D O J D P W P L N
O R W P O B C N S M N K M Z V G T V W P
O T L E P F L N Z N T K W N M P O I H B
C N X R J P O L P T E M B G G K Q W D O
Q E D F H Z N Y Q H K F D R U K X F P E
T D Q E A W U R Y O I B F A I N W K M Y
J N I K M T S O N I Z D G R H G P P Y W
L Q O T R T E A T I R A P Z U E R K V D
L Z H E N Q V N W M N J R X T B K U X E
T C N R R X H A M M I N G D I S T A N Z
Z E K C J Y H H U H I F T O J M P O L M
G G W O J M U E Z N M Y T O R D Q Z E F
J K K D H I F R P O D R Q C K F F X W R
S B G E H S D S L T G G E A J Y F T D K
C R P M P M S D Q P G S D B T F R D Y F
```

Übungen (II)

- **Aufgabe 2**
  - Was ist ein Blockcode?
  - Was ist der BCD-Code?
  - Was sind Pseudotetraden?
  - Was ist Entropie und wie wird diese berechnet?
  - Was ist Redundanz und wie wird diese berechnet?
  - Was ist Quellenredundanz?
  - Was bedeutet UTF?
  - Was ist die Hammingdistanz?
  - Was ist ein Paritätscode?
  - Wie ist Kreuzparität aufgebaut?
  - Was ist ein linearer Code?
  - Was ist ein perfekter Code?
  - Wie funktioniert die Prüfziffernberechnung beim ISBN?
  - Wie wird die Kontrollziffer im EAN berechnet?
  - Wie ermittelt man die Prüfziffern beim IBAN?

Übungen (II)

- **Aufgabe 3**
  Geben Sie für die Umwandlung von Klein- in Großbuchstaben im ASCII-Code eine einfache logische Operation mit Bit-Maske an!
- **Aufgabe 4**
  Das Alphabet $A = \{a, e, i, o, u\}$ mit den Auftrittswahrscheinlichkeiten $P(a) = 0{,}25$, $P(e) = 0{,}2$, $P(i) = 0{,}1$, $P(o) = 0{,}3$, $P(u) = 0{,}15$ sei mit $\{11{,}10{,}010{,}00{,}011\}$ binär codiert.
  Berechnen Sie die mittlere Codewortlänge und die Redundanz des Codes!
- **Aufgabe 5**
  Bestimmen Sie die Stellendistanzen und die Hamming-Distanzen für die folgenden Codes:
  a) $\{110101{,}101011{,}010011{,}101100\}$
  b) $\{00101011{,}01001010{,}01111000{,}10101001\}$

Übungen (IV)

- **Aufgabe 6**

  Bei einer seriellen Datenübermittlung werden mit 7 Bit codierte ASCII-Zeichen mit einem zusätzlichen Paritätsbit und einem Längsprüfwort nach jeweils 8 Zeichen gesendet. Es gilt gerade Parität.

  Folgende Nachricht wird empfangen:

MSB	1 0 1 1 1 1 1 1	0
	0 1 1 1 1 1 1 1	1
	0 1 0 0 0 0 0 1	0
Datenbits	0 0 0 1 0 1 0 0	0 Prüfspalte
	1 0 1 0 0 0 1 0	1
	1 1 0 0 1 0 0 1	0
LSB	0 0 1 1 0 1 1 0	0
Paritätsbits	1 0 0 0 1 0 0 0	0

  a) Wie lautet die empfangene Nachricht?

  b) Sind Übertragungsfehler aufgetreten? Wenn ja, wie lautet die korrekte Nachricht?

  c) Bestimmen Sie die durch die Paritätsbits bedingte zusätzliche Redundanz!

Übungen (V)

- **Aufgabe 7**
  Wie viele verschiedene Codewörter kann ein 2-aus-6-Code und wie viele ein 1-aus-15-Code enthalten? Geben Sie für die beiden Codes die Hamming-Distanz und die Redundanzen an! Dabei kann für alle Codewörter die gleiche Auftretenswahrscheinlichkeit angenommen werden.
- **Aufgabe 8**
  Gegeben seien die folgenden Zeichen und ihre Auftretenshäufigkeiten:

Zeichen	A	B	C	D	E	F
Häufigkeit	25	25	20	15	10	5

a) Berechnen Sie die Codierung mit dem Fano-Shannon-Algorithmus!

b) Geben Sie die mittlere Codewortlänge und die Entropie an!

Übungen (VI)

- **Aufgabe 9**
  a) Berechnen Sie den Prüfcode zur 10-stelligen ISBN 3-8689-4379
  b) Ist der EAN 5702016604137 korrekt?
  c) Geben Sie die korrekte EAN-Prüfziffer für 978-386894379
  d) Ist die IBAN DE14200800000816170700 korrekt?
  e) Ermitteln Sie die IBAN für die Kontonummer 1212557188 und die BLZ 10020099!

Algorithmen und Datenstrukturen

Teil 15: Kompression

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

- Einleitung
- Arithmetische Codierung
- Lauflängencodierung
- Differenzcodierung
- LZW-Algorithmus

Grundbegriffe (I)

- Blockcode vs. Codes variabler Länge
- Blockcode := Alle Codes haben feste Wortlänge
- Haben in der Regel hohe Redundanz

- Messung der Kompression:
  - Kompressionsfaktor KF:

$$KF = \frac{Platz\ unkomprimiert}{Platz\ komprimiert}$$

  - Eingesparter Speicherplatz:

$$ES = 1 - \frac{1}{KF}$$

- Beispiel:
  - Daten unkomprimiert: 1000
  - Daten komprimiert: 200
  - Dann: $KF = \frac{1000}{200} = 5\ und\ ES = 1 - \frac{1}{5} = 0{,}8 = 80\%$

Grundbegriffe (II)

- Statistisches Kompressionsverfahren:
  - Arbeitet auf Basis der Häufigkeiten der Daten in zu komprimierendem Objekt

- Verlustfreie vs. verlustbehaftete Kompression:
  - Verlustfrei: Informationsgehalt der Daten bleibt vollständig erhalten
  - Beispiele: Texte, Programmdateien, PNG
  - Verlustbehaftet: Informationen bleiben „im Wesentlichen“ erhalten, „gewisser“ Informationsverlust wird in Kauf genommen
  - Bsp.: JPEG, MP3
  - Vorteile:
    - Erhebliche höhere Kompressionsfaktoren
    - Anwender/in kann Kompressionsfaktor steuern (zu Lasten der Qualität)

Übersicht

- Einleitung
- Arithmetische Codierung
- Lauflängencodierung
- Differenzcodierung
- LZW-Algorithmus

Arithmetische Codierung

- Grundidee:
  - Wie beim Huffman-Verfahren:
    - Verlustfrei
    - Ausnutzung unterschiedlicher Häufigkeitsverteilungen in Ausgangsdatei
    - Codierung von Einzelzeichen, d.h. keine Berücksichtigung von Korrelationen benachbarter Zeichen
  - Jedem Ausgangstext wird Gleitkommazahl $0 \leq x < 1$ zugeordnet
- Ablauf:
  - Vor eigentlicher Codierung Ermittlung der Häufigkeiten einzelner Zeichen
  - Aufteilung Intervall $[0; 1[$ in n Intervalle, wobei jedes Intervall einem Zeichen entspricht und Intervallbreite Auftretenshäufigkeit entspricht
  - Tabelle ist für Codierung wie Decodierung erforderlich

Beispiel (I)

- Codierung Wort `ESSEN`
- Tabelle mit Häufigkeiten und Intervallen

Zeichen c_i	Auftrittswahrscheinlichkeit p_i	Intervall [u;o[
E	2/5	[0,0;0,4[
S	2/5	[0,4;0,8[
N	1/5	[0,8;1,0[

Kompressionsvorgang

```
( 1) u = 0
( 2) o = 1
( 3) REPEAT
( 4)   Lese nächstes Eingabezeichen ci
( 5)   d = o – u          // Aktuelle Intervalllänge
( 6)   o = u + d*o(ci) // Neue Obergrenze
( 7)   u = u + d*u(ci) // Neue Untergrenze
( 8) UNTIL Textende ist erreicht
( 9) Gebe u als Ergebnis aus
```

- Erläuterungen:
  - Intervallgrenzen u und o werden pro Schritt vergrößert bzw. verkleinert (abhängig von Intervall des Zeichens)
  - Weil stets $d < 1 \Rightarrow$ u und o können nie Grenzen des durch erstes Zeichen gegebenen Intervalls unter- bzw. überschreiten
  - Durch nächstes Zeichen hinzukommende Zuwachs kann nie größer sein als Intervalllänge dieses Zeichens
  - Dadurch ist eindeutige Umkehrbarkeit garantiert
  - Ergebnis abspeichern als Hexadezimalzahl statt Dezimalzahl

Beispiel (II)

Zeichen c_i	Intervalllänge d	Untergrenze u	Obergrenze o
	---	0,0	1,0
	0 … 1		
E	1,0	0,0	0,4
	0 … 0,4		
S	0,4	0,16	0,32
	0,16 … 0,32		
S	0,16	0,224	0,288
	0,224 … 0,288		
E	0,064	0,224	0,2496
	0,224 … 0,2496		
N	0,0256	0,24448	0,2496
	0,24448 … 0,2496		

- Ergebnis x = 0,24448
- Nachkommastellen speichern

Dekompression

- Kompressionsalgorithmus umkehren:
  - D.h. aktuelles Intervall auf [0;1[strecken
  - Zu decodierendes Zeichen kann aus Tabelle abgelesen werden
  - Ende der Dekompression muss explizit geregelt werden:
    - Abspeichern Anzahl der Zeichen oder
    - Einführen spezielles Endezeichen

```
( 1) WHILE (Nicht alle Zeichen decodiert)
( 2)   Lese Code x
( 3)   Suche Zeichen c_i, in dessen Intervall x liegt
( 4)   Gebe Zeichen c_i aus
( 5)   d = o(c_i) – u(c_i)       // Neue Intervalllänge
( 6)   x = (x-u(c_i)) / d        // Neuer Code
( 7) END WHILE
```

Beispiel

- Rückcodierung von 0,24448:

Code x	Intervall-länge d	Unter-grenze u	Ober-grenze o	Ausgabe-zeichen c_i
0,24448	0,4	0,0	0,4	E
0,6112	0,4	0,4	0,8	S
0,528	0,4	0,4	0,8	S
0,32	0,4	0,0	0,4	E
0,8	0,2	0,8	1,0	N
0,0	---	---	---	Ende

- Zur Erinnerung:

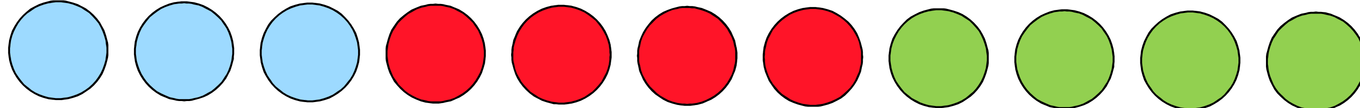
Zeichen c_i	Auftrittswahrscheinlichkeit p_i	Intervall [u;o[
E	2/5	[0,0;0,4[
S	2/5	[0,4;0,8[
N	1/5	[0,8;1,0[

Übersicht

- Einleitung
- Arithmetische Codierung
- Lauflängencodierung
- Differenzcodierung
- LZW-Algorithmus

Lauflängen

- Idee: Stelle Folge gleicher Zeichen (Run) durch Angabe des Zeichens und Anzahl an



- Lauflängencodierung = RLE (Run Length Encoding)
- Anwendungsgebiete:
  - Allgemein: Datensequenzen mit wenigen verschiedenen Werten
  - Speziell: Bilder mit wenigen verschiedenen Farben
- Erreichbare Kompressionsrate abhängig von:
  - Anzahl unterschiedlicher Ausgangswerte
  - Länge der Runs

Vorgehen

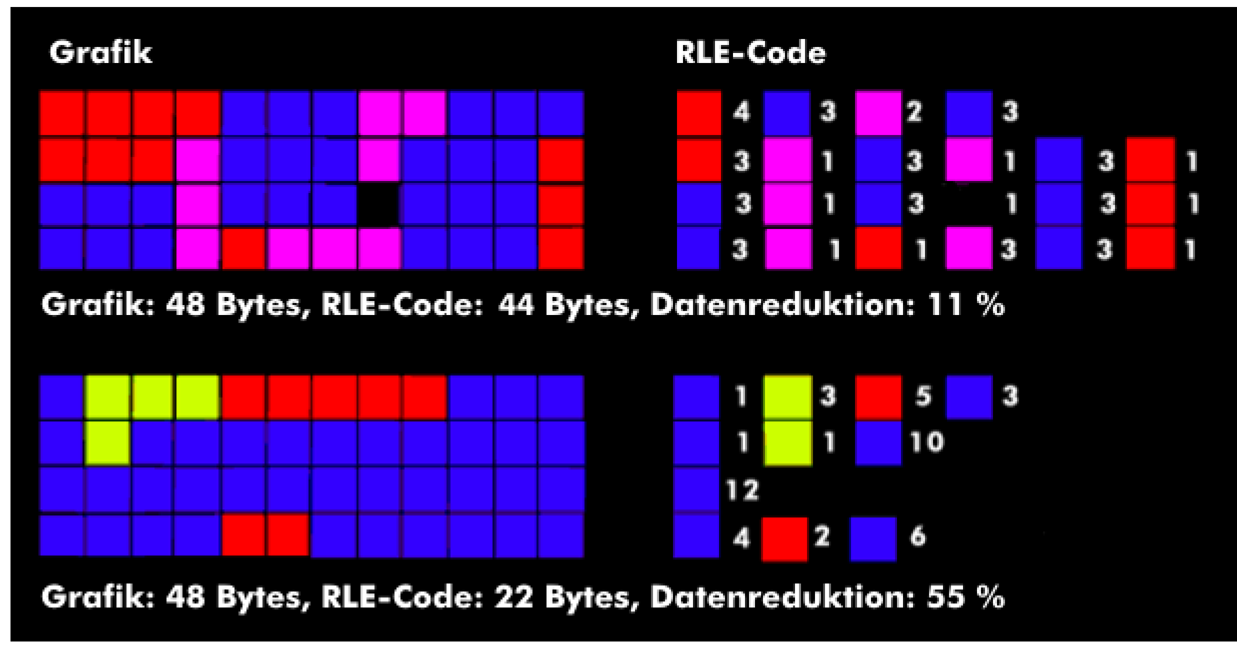
- Sonderzeichen festlegen, z.B. #
- Codierung:
  - Run Länge n von x als x#<n> darstellen
  - $n \in [4, 259]$ mit einem Byte darstellbar
  - # als ## darstellen
- Beispiele:
  - Eingabe: XXXXXYYZZZZZZZZZU
  - RLE: X#5YYZ#9U

  - Eingabe: X#Y##
  - RLE: X##Y####

Beispiel binäre Werte

- Originalwerte: `0000000000000001111111000000011111111111`
- Idee zur Speicherung:
  - Ermittle längsten Run
  - Wähle Wortbreite, so dass längster Run repräsentierbar
  - Speichere nacheinander alternierende Anzahl 0en und 1en ab
- Im Beispiel:
  - Längster Run = 15
  - D.h. 4 Bit-Wort zum Speichern einer Run-Länge
  - Also: 1111 0111 0111 1011 (15, 7, 7, 11)
- Kompressionsfaktor:
  - Unkomprimierte Speicherung: 40 Bit
  - Komprimierte Speicherung: 16 Bit
  - Kompressionsfaktor: 2,5

Beispiel Grafiken



[http://www.itwissen.info]

- Anwendung: TIFF-, Bitmap-, TGA-Format

Quadtrees (I)

- Erweiterung der Lauflängencodierung auf zwei Dimensionen
- Idee:
  - Baumstruktur von rechteckigen Bereichen abnehmender Größe mit jeweils einheitlichem Wert
- Aufbau Quadtree:
  - Wurzel als unterste Ebene entspricht Gesamtbereich (bzw. ausgewähltem rechteckigen Bereich)
  - Wurzel hat vier Nachfolger, jeder repräsentiert ein Viertel der Wurzel
  - Dieses wird weitergeführt, bis alle Werte eines Knoten gleichen Wert haben
- Zwei Extremfälle:
  - Alle Daten in Wurzel haben gleichen Wert $\Rightarrow$ Baum besteht nur aus Wurzel
  - Verfahren muss weitergeführt werden, bis Knoten nur noch Einzelwerte besitzen

Quadtrees (II)

- Speicherung Quadtree:
  - Markiere Knoten mit 1, falls er weiter zerlegt werden muss
  - Markiere Knoten mit 0 sonst
  - Durch schichtweise Bitfolge ist Baum eindeutig identifiziert
  - Zusätzlich müssen Wert der Blattknoten gemerkt werden

Beispiel

1	2	3	3	3	3	2	1
2	3	3	3	3	3	3	2
3	3	4	4	4	4	3	3
3	3	4	4	4	4	3	3
3	4	5	5	6	6	6	6
3	4	5	5	6	6	6	6
4	4	5	6	6	6	6	6
4	4	5	6	6	6	6	6

Abb. 3.14 Beispiel für einen Quadtree für ein einfaches Bild. Gespeichert werden die Knotenadressen als Folge von 1en (innere Knoten) und 0en (Blätter mit gerundeten Ecken) und danach die den Blättern zugeordneten Grauwerte. Hier ergeben sich 17 Bit für die Baumstruktur gefolgt von 25 Grauwerten: 1 1110 1000 0100 1001 6334343541223213234345656. Bei einer Codierung der Grauwerte mit 8 Bit ergibt sich eine Kompression von 512 Bit auf 217 Bit

Übersicht

- Einleitung
- Arithmetische Codierung
- Lauflängencodierung
- Differenzcodierung
- LZW-Algorithmus

Idee

- Speichere nicht einzelne Werte, sondern Differenz aufeinanderfolgender Werte
- Anwendungsgebiet: Messwerte
- Aufeinanderfolgende Werte hier meist nah beieinander
- Lässt gute Kompressionsraten erwarten

Verlustfreie Differenz-Codierung

- Erster Wert sei f_0
- Speichere diesen ab
- Speichere danach jeweils Differenzen $d_i = f_i - f_{i-1}$
- Typische Vorgehensweise:
  - Führe dies nur bis bestimmter Differenz durch
  - Nehme dann tatsächlichen Wert als neuen Referenzpunkt

  - Verwendung variabler Codelängen
  - Dadurch bekommen häufige Differenzen kurze Codes

  - Spezielles Codewort für Fall, dass Differenz größer als Maximalwert, d.h. tatsächlicher Wert wird gespeichert

Beispiel

- D_1: Differenzcode variabler Wortlänge
- D_2: Differenzcode konstante Wortlänge (4 Bit, Werte 0 bis 7)

Differenz d	Code D_1	Code D_2
0	1	0000
1	0100	0001
-1	0101	1001
2	0110	0010
-2	0111	1010
3	00100	0011
-3	00101	1011
4	00110	0100
-4	00111	1100
5	000100	0101
-5	000101	1101

Differenz d	Code D_1	Code D_2
6	000110	0110
-6	000111	1110
7	0000100	0111
-7	0000101	1111
8	0000110	$\lvert d \rvert>7$: 1000 und danach 8 Bit für den Datenwert
-8	0000111	
$\lvert d \rvert>8$	00000 und danach Daten-wert	

Ungünstiges Szenario

- Treten häufig den Maximalwert überschreitende Differenzen auf $\Rightarrow$ Keine/schlechte Kompressionsrate

Verlustbehaftete Differenzcodierung

- Erweiterung des verlustfreien Verfahrens
- Kann größere Differenzen verarbeiten
- Codewort wird nicht einzelnem Zahlwert, sondern Intervall zugeordnet

Tabelle 3.22 Zwei Differenz-Codes (variable und konstante Wortlänge) für verlustbehaftete Differenz-Codierung. Hier werden mehrere Differenzen dem gleichen Codewort zugeordnet

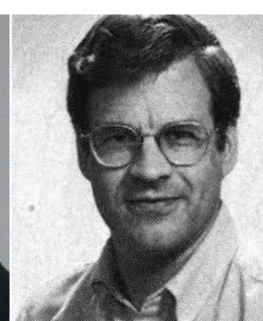
Differenz d	Code D_1	Code D_2
−1, 0, 1	1	0000
2, 3, 4	0100	0001
−2, −3, −4	0101	0010
5, 6, 7	0110	0011
−5, −6, −7	0111	0100
8, 9, 10	00100	0101
−8, −9, −10	00101	0110
11, 12, 13	00110	0111
−11, −12, −13	00111	1000
14, 15, 16	000100	1001
−14, −15, −16	000101	1010
17, 18, 19	000110	1011
−17, −18, −19	000111	1100
20, 21, 22	0000100	1101
−20, −21, −22	0000101	1110
23, 24, 25	0000110	$\|d\| > 22$ 1111 und
−23, −24, −25	0000111	danach 8 Bit für
		den Datenwert
$\|d\| > 25$	00000	und danach 8 Bit
		für den Datenwert

Übersicht

- Einleitung
- Arithmetische Codierung
- Lauflängencodierung
- Differenzcodierung
- LZW-Algorithmus

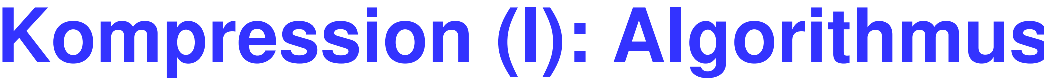
Prinzip

- 1977: LZW Abraham Lempel, Jakob Ziv und Terry A. Welch
- Dynamisches Wörterbuchverfahren



[http://www.geeksforgeeks.org/]

- Unterschied zu Huffman- und Arithmetischer Codierung: Berücksichtigung aufeinanderfolgender Zeichengruppen
- Prinzip:
  - Codetabelle:
    - Jeder Eintrag String aus Quellalphabet und zugehöriger Code
    - Wird anfangs mit Einzelzeichen vorbelegt
    - Wird während der Kompression nach und nach erweitert und an Eingabe angepasst
    - Daher: Im Vorfeld keine statistischen Informationen notwendig
    - Codetabelle für Decodierung nicht notwendig



Kompression (I): Algorithmus

- Variablen:
  - P: Präfix
  - c: Aktuelles gelesenes Zeichen
  - Z: Eingabestring

```
( 1) Initialisiere Codetabelle mit Einzelzeichen
( 2) P = ""
( 3) WHILE (Eingabezeichen vorhanden)
( 4)   Lies nächstes Zeichen c aus Z
( 5)   IF Pc in Codetabelle
( 6)      P = Pc
( 7)   ELSE
( 8)     IF Codetabelle nicht voll
( 9)       Trage Pc in nächste freie Position ein
(10)     END IF
(11)     Gib Code für P aus
(12)     P = c
(13)   END IF
(14) END WHILE
(15) Gib Code für P aus
```

Kompression (II): Beispiel (I)

- Komprimiere ABABCBABAB
- Annahme: Codetabelle habe 8 Einträge, Code 3 Bit breit
- Initiale Codetabelle:

Zeichenkette	Ausgabe-Code
A	0 = 000
B	1 = 001
C	2 = 010
	3 = 011
	4 = 100
	5 = 101
	6 = 110
	7 = 111

Kompression (III): Beispiel (II): Ablauf

Schritt	Zeichen c	Präfix P	Eintrag in Codetabelle	Ausgabe
0	---	---	Vorbelegung	---
1	A	A	---	---
2	B	B	AB=3	0
3	A	A	BA=4	1
4	B	AB	---	---
5	C	C	ABC=5	3
6	B	B	CB=6	2
7	A	BA	---	---
8	B	B	BAB=7	4
9	A	BA	---	---
10	B	BAB	---	---
11	---	---	---	7

Resultierender Code: 013247 = 000 001 011 010 100 111

Kompression (IV): Beispiel (III)

- Finale Codetabelle:

Zeichenkette	Ausgabe-Code
A	0 = 000
B	1 = 001
C	2 = 010
AB	3 = 011
BA	4 = 100
ABC	5 = 101
CB	6 = 110
BAB	7 = 111

Dekompression (I): Idee

- Prinzipielle Vorgehensweise:
  - Lege Code-Tabelle an, belege mit Eingabezeichen vor
  - Algorithmus liest Codewort, sucht in Codetabelle und gibt Zeichen aus
  - Zusätzlich: Hänge an zuletzt decodierten String erstes Zeichen des aktuell decodierten String an und trage Ergebnis in Codetabelle ein
- Sonderfall:
  - Wird String in Codetabelle eingetragen und im nächsten Schritt bereits wieder verwendet, so kann er bei Dekompression noch nicht vorhanden sein
  - Fehlender Code: Verlängerung des Präfix um erstes Zeichen des zuvor ausgegebenen String
  - In Codetabelle einzutragender String ist gleich auszugebendem String

Dekompression (II): Algorithmus

- Variablen:
  - P: Präfix, c: Aktuelles gelesenes Codewort

```
( 1) Initialisiere Codetabelle mit Einzelzeichen
( 2) P = ""
( 3) WHILE (Codeworte vorhanden)
( 4)   Lese nächstes Codewort c
( 5)   IF c in Codetabelle
( 6)     Gib zu c gehörenden String s aus
( 7)     k = Erstes Zeichen von s
( 8)     Trage Pk in Codetabelle ein, falls nicht vorhanden
( 9)     Setze P auf zu Code c gehörenden String
(10)   ELSE //Sonderfall
(11)     k = Erstes Zeichen von P
(12)     Gebe Pk aus
(13)     Trage Pk in Codetabelle ein
(14)     P = Pk
(15)   END IF
(16) END WHILE
```

Dekompression (III): Beispiel (I)

- Zeichen A,B,E,N,U Vorbelegung mit 0,1,2,3,4
- Code 4 Bit breit
- Codierte Eingabe: `0001 0000 0011 0110 0010 0111 0011 0101 0100`

Schritt	Code c	Zeichen k	Ausgabe	P	Eintrag in Codetabelle
0	---	---	---	---	Vorbelegung
1	0001	B	B	B	---
2	0000	A	A	A	BA(5)
3	0011	N	N	N	AN(6)
4	0110	A	AN	AN	NA(7)
5	0010	E	E	E	ANE(8)
6	0111	N	NA	NA	EN(9)
7	0011	N	N	N	NAN(10)
8	0101	B	BA	B	NB(11)
9	0100	U	U	U	BU(12)

Dekompression (IV): Beispiel (II)

- Finale Codetabelle:

Code	Zeichenkette
A	0000
B	0001
E	0010
N	0011
U	0100
BA	0101
AN	0110

Code	Zeichenkette
NA	0110
ANE	1000
EN	1001
NAN	1010
NB	1011
BU	1100

Zusammenfassung (I)

- Einleitung:
  - Kompressionsfaktor
  - Kompressionsrate
  - Statistisches Kompressionsverfahren
  - Verlustfrei vs. verlustbehaftete Kompression

- Arithmetische Codierung:
  - Idee
  - (De-)Kompression

- Lauflängencodierung:
  - Idee
  - Binärwerte
  - Bilder
  - Quadtrees

Zusammenfassung (II)

- Differenzcodierung:
  - Verlustfreie Variante
  - Verlustbehaftete Variante

- LZW-Algorithmus:
  - Prinzip
  - Kompression
  - Dekompression

Übungen (I)

- **Aufgabe 1**

  Finden Sie im folgenden Rätsel fünf Begriffe aus diesem Vorlesungsabschnitt!

```
O Q K W B Z G Z B J B T K L V R C K E F
J Z S X R D D C V Q H L C V V I O J E B
F D M S W U B M B P E I T L F M L R R Z
Y N V O M P X P S P A T E X P X R Z T F
E V Z A K D I A M E K F V R V N S C D C
G N I D O C N E H T G N E L N U R R A R
D L J M K V L Y R T O S P I S O C S U Q
G B P M A V K G R F S E P Y T X A F Q V
K O A L M Q W W S I T W V I O T E P F U
D M K X Y Y O W O Z H S F F M N O H D D
L R C B L Q L N U J Y M U X S D O U R Q
G N H K I E S T W O D F T L H X L T M A
E U D F I F Z O B W U I I K R F G I O L
N V J R A L P R D A D I N C Q E C M L D
G M M K Z Y X Y O J I Y G T C B V P T H
E T T E N X F V P X V O W W W V D G P F
Q O Z W J P J F B O N X T J T O B D L P
R T S Z N V D Z X Y O Y U A D X R D R O
D V Y K F D N G W O C E Q W M I P E L G
O H K N M O P I I O G P W H K G W R M S
```

Übungen (II)

- **Aufgabe 2**

  Führen Sie eine Lauflängencodierung durch für

  a) AAAAAAAAADDDDDAABBBBBBCCCCCBABABDDDDDDDEEEEEE

  b) 00000011111100000000000000000000001111111111111111100000

  c) 01010010100101100101001001001010101010100001010101010101

  Geben Sie auch jeweils den Kompressionsfaktor an!

  Bitte geben Sie auch alle Annahmen und Rahmenbedingungen an!

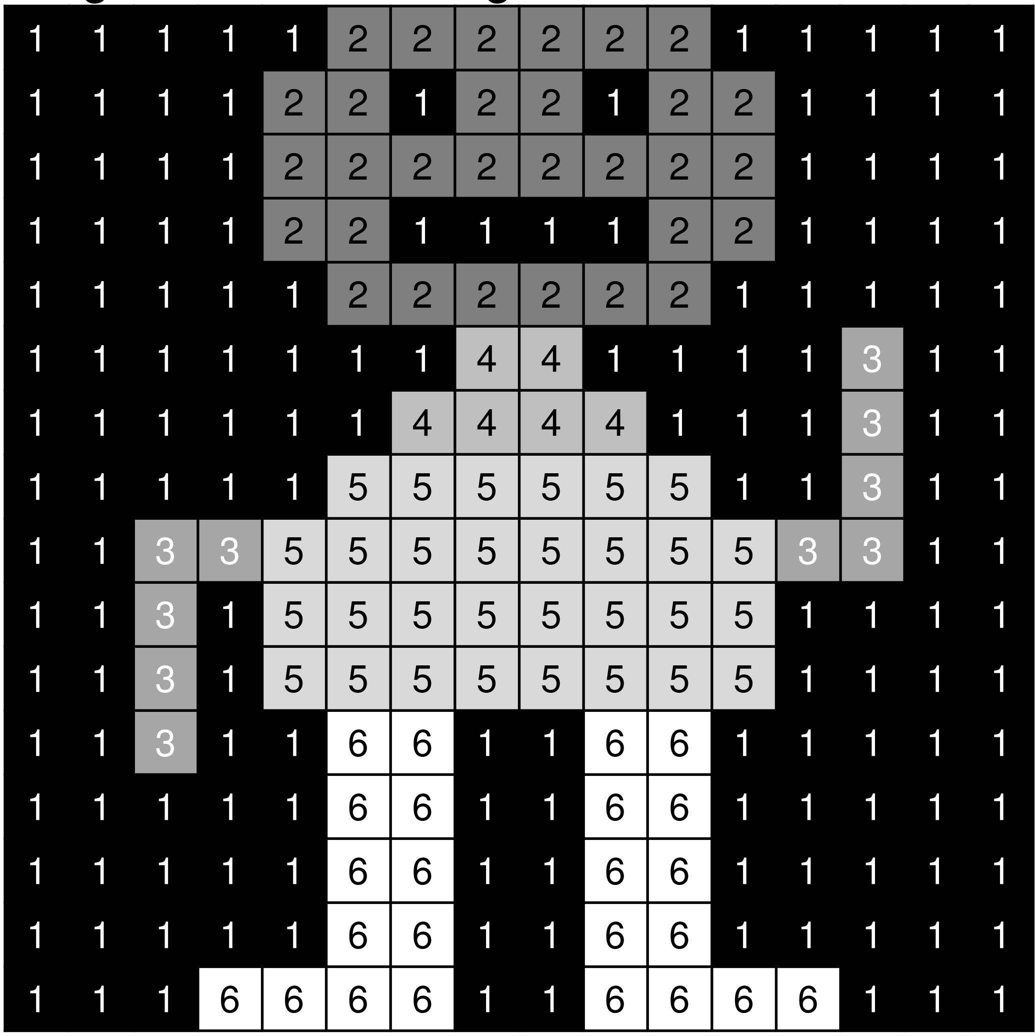
- **Aufgabe 3**

  Lässt sich die Eingabe `ANTANANARIVO` (Hauptstadt von Madagaskar) oder `PAPAYAPALMEN` durch den LZW-Algorithmus effizienter komprimieren?

Übungen (III)

- **Aufgabe 4**

  Gegeben sei das folgende Bild mit Grauwerten 1 bis 6:



1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
1	1	1	1	2	2	1	2	2	1	2	2	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
1	1	1	1	2	2	1	1	1	1	2	2	1	1	1	1
1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
1	1	1	1	1	1	1	4	4	1	1	1	1	3	1	1
1	1	1	1	1	1	4	4	4	4	1	1	1	3	1	1
1	1	1	1	1	5	5	5	5	5	5	1	1	3	1	1
1	1	3	3	5	5	5	5	5	5	5	5	3	3	1	1
1	1	3	1	5	5	5	5	5	5	5	5	1	1	1	1
1	1	3	1	5	5	5	5	5	5	5	5	1	1	1	1
1	1	3	1	1	6	6	1	1	6	6	1	1	1	1	1
1	1	1	1	1	6	6	1	1	6	6	1	1	1	1	1
1	1	1	1	1	6	6	1	1	6	6	1	1	1	1	1
1	1	1	1	1	6	6	1	1	6	6	1	1	1	1	1
1	1	1	6	6	6	6	1	1	6	6	6	6	1	1	1

Übungen (IV)

a) Bestimmen Sie die Auftrittswahrscheinlichkeiten der Grauwerte!

b) Geben Sie für die Grauwerte einen Binärcode mit minimaler konstanter Wortlänge an und berechnen Sie die Größe des so codierten Bildes.

c) Konstruieren Sie unter Verwendung des Huffman-Algorithmus einen optimalen Code variabler Wortlänge. Wie viele Bit benötigt man hiermit zur Speicherung des Bildes? Wie groß ist der Kompressionsfaktor?

d) Konstruieren Sie einen Lauflängencode. Wie viele Bit benötigt man hiermit zur Speicherung des Bildes? Wie groß ist der Kompressionsfaktor? (Platzbedarf für Zeilenumbrüche kann vernachlässigt werden)

e) Geben Sie eine Quadtree-Speicherung des Bildes an!

Übungen (V)

- **Aufgabe 5**

  Gegeben sei das Wort „AAAAHHABBLLABBLAAA“

  a) Führen Sie eine arithmetische Codierung durch!

  b) Führen Sie eine Huffman-Codierung durch!