Datenstrukturen und Algorithmen

Das Team ...

- Leif Kobbelt
- Stephan Bischoff
- Antje Nowack
- Thomas von der Maßen
- 20 studentische Tutoren

Termine

- Vorlesung:

Fr. 19.04.2002
14:00 - 15:30
Aula 1 (Hauptgebäude)

- „kleine“ Übung:
  ...

„kleine“ Übungen

- Termine: (ca. 20 Tutorien)
  Mo. 8:15 - 9:45
  Mo. 12:15 - 13:45
  Mo. 15:45 - 17:15
  Di. ???
- Gruppen bis zu 25 Teilnehmer

Anmeldung

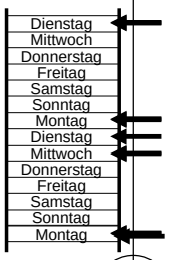
- www.rwth-graphics.de
  (ab Mittwoch 12:00 freigeschaltet)
- weitere Informationen ...
  - ggf. Ankündigungen
  - Übungsblätter als PDF
  - Literaturverzeichnis
  - ...

Übungsblätter

- 11 Blätter x 4 Aufgaben
- Blatt (1) am **23.04.**
- Gruppenarbeit (2-3 Studenten)
- Programmieraufgaben (***Java***)
  → Einführung am **17.04.**
- Abgabe der Programmieraufgaben
  per Email

Übungsblätter

- Ausgabe:
  Dienstag nach der Vorlesung
- Abgabe:
  Montag in den Tutorien
- Besprechung:
  Mittwoch in der großen Übung
- Rückgabe:
  Montag in den Tutorien



Übungsscheine

- Ab 50 % der Punkte
- Blatt (1) - (5), Blatt (6) - (11)
  zählen *separat*
- Individueller Leistungsnachweis
  → *aktive* Teilnahme an den Tutorien

PC Arbeitsplätze

- PC-Pool:
  Di. 9:00 - 14:00
  Mi. 16:00 - 21:00
  Do. 16:30 - 21:00
- Informatik-Zentrum (Ahornstr. 55)
- First come, first serve
- CD-Restposten für Heimarbeiter

Literatur

- *„Inoffizielle“* Vorlesungsmitschrift
- evtl. weitere/überarbeitete Kapitel im Laufe des Semesters
- www.rwth-graphics.de
- Informatikbibliothek (Handapparat)

Klausur

- Datum:
  Do. 22.08.2002
  14:00 - 16:00
  (fast alle Hörsäle)

Sprechstunden

- Leif Kobbelt (nach Vereinbarung)
- Stephan Bischoff Di 11:00 - 12:00
- Antje Nowack Mi 10:30 - 11:30
- Thomas von der Maßen Mi 13:00 - 14:00

... weitere Fragen ???

Datenstrukturen und Algorithmen

Positionsbestimmung

- Informatik =

  Wissenschaft von der algorithmischen Problemlösung

- Geg: Problem
- Ges: Lösung

Positionsbestimmung

- Informatik =

  Wissenschaft von der algorithmischen Problemlösung

- Geg: Problem(klasse)
- Ges: Lösungsprozedur

Positionsbestimmung

- Informatik =

  Wissenschaft von der algorithmischen Problemlösung

- Geg: Problem(klasse)
- Ges: (automatisierbare) Lösungsprozedur aus elementaren Operationen

Informationsverarbeitung

- Was ?
  - Repräsentation/Organisation von Daten
    $\rightarrow$ *Datenstruktur*
- Wie ?
  - (schrittweise) Modifikation von Daten
    $\rightarrow$ *Algorithmus*
- Software = Datenstrukturen + Algorithmen

Abu Ja‘far Muhammed Ibn Musa-Al-Khawarizimi

- 13734 : 42 = 327
- 126
- 113
- 84
- 294
- 294
- 0

Algorithmus

- Definition (Rezept, Prozedur, ...)
  Schrittweise Modifikation von Daten zur Lösung eines Problems
- Kriterien (Donald Knuth)
  - Determinismus (*in*deterministische?)
  - Input (# ≥ 0)
  - Output (# ≥ 1)
  - Terminierung (#steps < ∞)

Analyse von Algorithmen

- (partielle) Korrektheit
- Vollständigkeit
- Komplexität (Speicherplatz, Rechenzeit)
- Robustheit (bei inkorrekten Eingaben)
  „garbage in garbage out“

Einbettung in die Informatik

Problem → Spezifikation

Spezifikation → Algorithmus

Algorithmus → Programm

Programm → Computer

Idealer Software-Entwurf

- Spezifiziere: Eingabe / Ausgabe
- Problemlösung
  - Beziehungen, Referenzen
    → Datenstrukturen
  - Wesentliche Verarbeitungsschritte
    → Blockbildung, Prozeduren, Algorithmen
- Analyse ...
  - Korrektheit, Aufwand (Speicher / Rechenzeit)
- Implementieren / Testen / Dokumentieren

Software-Entwicklung

- Datenstrukturen
  – Statische Beziehungen
  – (explizite) Funktionale Zusammenhänge
- Algorithmen
  – Dynamische Prozesse
  – (implizite) Funktionale Zusammenhänge

Beispiel: Menge

Daten-struktur
- $S \in 2^{IN}$ (d.h. $S \subseteq IN$)
- $min : 2^{IN} \rightarrow IN$
- $min(S) \in S \ \wedge \ \forall x \in S : min(S) \leq x$

Implementierung
- int S[ ], size, min()

Algorithmus
- int i,min = S[0];
  for (i=1 ; i<size ; i++)
  if (S[i] < min)
  min = S[i];

Analyse von Algorithmen

- (partielle) Korrektheit
- Vollständigkeit
- Komplexität (Speicherplatz, Rechenzeit)
- Robustheit (bei inkorrekten Eingaben)
  „garbage in garbage out“

Ziele der Vorlesung

- Grundlegende Konzepte für Entwurf und Analyse von Algorithmen
- Effiziente Implementierung
- Komplexitätsanalyse
- Repertoire an Standardalgorithmen

Thematische Übersicht

- (1.1) Abstrakte Datentypen
- (1.2...n) Konkrete Datentypen
  (Array, List, Queue, Stack, Tree, Heap, Graph, ...)
- (2.1) Entwurf und Analyse von Algorithmen
- (2.2...n) Spezielle Algorithmen
  (Sortieren, Suchen, Bäume, Graphen, *Optimieren, Kodieren,* ...)

(1.1) Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

Zeit: hh:mm:ss

Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

Add : Zeit x Zeit $\rightarrow$ Zeit

Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

$$\text{Add}([h_1,m_1,s_1],[h_2,m_2,s_2]) = [\,(h_1 + h_2 + b(m_1 + m_2)) \,\%\, 24,\ (m_1 + m_2 + b(s_1 + s_2)) \,\%\, 60,\ (s_1 + s_2) \,\%\, 60\,]$$

Abstrakte Datentypen

- Warum ***abstrakt*** ?
  - Keine Hinweise darauf, wie die jeweiligen Funktionen implementiert werden können.
- Warum ***Typendefinition*** ?
  - Spezifikation / Verifikation
  - Top-down Software-Entwurf

Beispiel: Bool

- Wertebereich: { true, false }
- $\neg$: bool $\rightarrow$ bool
  $\wedge$: bool x bool $\rightarrow$ bool
  $\vee$: bool x bool $\rightarrow$ bool
- $\neg$true = false, $\neg$false = true,
  x $\wedge$ true = x, x $\wedge$ false = false,
  x $\vee$ true = true, x $\vee$ false = x

Aufzählungstypen

- Endlicher Wertebereich
  - Vollständige Spezifikation durch Tabellen
- Verallgemeinerung
  - Aufzählungstypen: ***enum***
  - z.B. Wochentage, chemische Elemente, ...
  - Kodierung als ganze Zahlen modulo n (implizite Ordnungsrelation)

Beispiel: Integer

- Wertebereich: IZ = $IN^+ \cup \{0\} \cup -IN^+$
- + : IZ x IZ $\rightarrow$ IZ
  - : IZ x IZ $\rightarrow$ IZ
  - : IZ $\rightarrow$ IZ
  * : IZ x IZ $\rightarrow$ IZ
  / : IZ x IZ $\rightarrow$ IZ
  ...

Beispiel: Integer

- +(0,y) = y
  +(x+1,y) = +(x,y)+1
  +(x -1,y) = +(x,y) -1
- -(y ,y) = 0
  -(x+1,y) = -(x,y)+1
  -(x -1,y) = -(x,y) -1
- -(x) = -(0,x)
- *(0,y) = 0
  (x+1,y) = +((x,y), y)
  (x -1,y) = - ((x,y), y)
- / (x ,0) = ERROR
  / (x ,y) = 0 if |x|<|y|
  / (x+y,y) = / (x,y)+1
  / (x -y,y) = / (x,y) -1

Beispiel: Integer

- +(0,y) = y
  +(x+1,y) = +(x,y)+1
  +(x -1,y) = +(x,y) -1
- -(y ,y) = 0
  -(x+1,y) = -(x,y)+1
  -(x -1,y) = -(x,y) -1
- -(x) = -(0,x)
- *(0,y) = 0
  (x+1,y) = +((x,y), y)
  (x -1,y) = - ((x,y), y)
- / (x ,0) = ERROR
  / (x ,y) = 0 if |x|<|y|
  / (x+y,y) = / (x,y)+1
  / (x -y,y) = / (x,y) -1

Skalare Typen

- „Zahlen“ mit 1-dim Wertebereich
  - Char
  - Integer
  - Float, Double
- ***Achtung:*** In realen Implementierungen haben skalare Typen meistens einen endlichen Wertebereich
  - Integer : $[1-2^{b-1} \ldots 2^{b-1}]$
  - Double : $[10^{-300} \ldots 10^{300}]$

Overflow Error

Zusammengesetzte Typen

- **Endliche / feste** Kombination von skalaren Typen oder zus.ges. Typen
  - k-dim Vektoren
  - Adressen-Eintrag
  - ...
- Mehrdimensionaler Wertebereich

Beispiel: Sequenz

- Wertebereich: $W = \{\} \cup IN \cup IN^2 \cup IN^3 \cup ... \neq 2^{IN}$!!!
- Create : $\rightarrow W$
  Append : $IN \times W \rightarrow W$
  Remove : $IN \times W \rightarrow W$

Beispiel: Sequenz

- Append(x,{y_1, ..., y_n}) = {y_1, ..., y_n,x}

Append(z,Append(y,Append(x,Create()))) = {x,y,z}

Remove(x,Append(x,z)) = z

Remove(x,Append(y,z)) =
Append(y,Remove(x,z)) if $x \neq y$

Beispiel: Sequenz

- Append(x,{y_1, ..., y_n}) = {y_1, ..., y_n,x}
- Remove(x,Create()) = Create()

Remove(x,Append(x,z)) = z

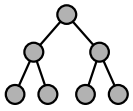
Remove(x,Append(y,z)) =
Append(y,Remove(x,z)) if $x \neq y$

Lineare Strukturen

- Beliebige Sequenz von Basisobjekten (skalare / zusammengesetzte Typen) mit variabler Länge
  - Arrays
  - Listen (einfach/doppelt verkettet)
  - Queue (FIFO)
  - Stack (LIFO)
- Unterscheiden sich im wesentlichen in der ***Zugriffsfunktionalität***

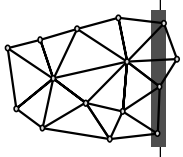
Baumstrukturen

- Hierarchische Strukturen
- Vater/Sohn-Beziehung („der“ Knoten)
- Keine Zyklen
- Eindeutige Vorfahren



Graphen

- Beliebige topologische Struktur
- Nachbarschaftsbeziehungen zwischen Knoten
- Wichtige Spezialfälle
  - Planare Graphen
  - Gerichtete Graphen
  - Zyklenfreie Graphen
  - ...



Datentypen-Typen

- Aufzählungstypen *(bool, enum)*
- Skalare Typen *(char, int, float, ..)*
- Zusammengesetzte Typen *(struct, class)*
- Lineare Strukturen *(list, queue, stack)*
- Bäume
- Graphen

(1.1) Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

Beispiel: Integer

- Wertebereich: IZ = $IN^+ \cup \{0\} \cup -IN^+$
- + : IZ x IZ $\rightarrow$ IZ
  - : IZ x IZ $\rightarrow$ IZ
  - : IZ $\rightarrow$ IZ
  * : IZ x IZ $\rightarrow$ IZ
  / : IZ x IZ $\rightarrow$ IZ
  ...

Beispiel: Integer

- +(0,y) = y
  +(x+1,y) = +(x,y)+1
  +(x -1,y) = +(x,y) -1
- -(y ,y) = 0
  -(x+1,y) = -(x,y)+1
  -(x -1,y) = -(x,y) -1
- -(x) = -(0,x)

- *(0,y) = 0
  (x+1,y) = +((x,y), y)
  (x -1,y) = - ((x,y), y)
- / (x ,0) = ERROR
  / (x ,y) = 0 if |x|<|y|
  / (x+y,y) = / (x,y)+1
  / (x -y,y) = / (x,y) -1

Beispiel: Integer

- +(0,y) = y
  +(x+1,y) = +(x,y)+1
  +(x -1,y) = +(x,y) -1
- -(y ,y) = 0
  -(x+1,y) = -(x,y)+1
  -(x -1,y) = -(x,y) -1
- -(x) = -(0,x)

- *(0,y) = 0
  (x+1,y) = +((x,y), y)
  (x -1,y) = - ((x,y), y)
- / (x ,0) = ERROR
  / (x ,y) = 0 if |x|<|y|
  / (x+y,y) = / (x,y)+1
  / (x -y,y) = / (x,y) -1

Beispiel: Sequenz

- Wertebereich: W = { } $\cup$ IN $\cup$ IN^2 $\cup$ IN^3 $\cup$ ...
  $\neq 2^{IN}$!!!
- Create : $\rightarrow$ W
  Append : IN x W $\rightarrow$ W
  Remove : IN x W $\rightarrow$ W

Beispiel: Sequenz

- Append(x,{y_1, ..., y_n}) = {y_1, ..., y_n,x}

Append(z,Append(y,Append(x,Create()))) = {x,y,z}

Remove(x,Append(x,z)) = z

Remove(x,Append(y,z)) =
Append(y,Remove(x,z)) if $x \neq y$

Beispiel: Sequenz

- Append(x,{y_1, ..., y_n}) = {y_1, ..., y_n,x}
- Remove(x,Create()) = Create()

  Remove(x,Append(x,z)) = z

  Remove(x,Append(y,z)) =
  Append(y,Remove(x,z)) if $x \neq y$

Datentypen-Typen

- Aufzählungstypen *(bool, enum)*
- Skalare Typen *(char, int, float, ..)*
- Zusammengesetzte Typen *(struct, class)*
- Lineare Strukturen *(list, queue, stack)*
- Bäume
- Graphen

(1.1) Abstrakte Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- Axiome

Konkrete Datentypen

- Spezifizieren ***Form*** und ***Funktionalität*** der zu verarbeitenden Daten
- Datenobjekte, Datenfelder
- Funktionen
- ~~Axiome~~ Methoden / Algorithmen

Java Klassen

- Datenobjekte → Attribute
- Funktionen → Methoden-Header (Rückgabetyp, formale Parameter)
- Algorithmen → Methoden-Body
- Überprüfe korrekte Methodenimplementierung anhand der abstrakten Axiome

(1.2) Lineare Strukturen

- Sequenz {x_1, ..., x_n} von beliebigen Datenobjekten x_i
- Typische Operationen
  - Füge y am Anfang / am Ende / hinter x_i ein
  - Ersetze x_i durch y
  - Entferne x_i
  - Lese das i-te Element

(1.2) Lineare Strukturen

- Typische Operationen
  - Verknüpfe zwei Sequenzen
  - Zerlege eine Sequenz
  - Bestimme die Länge der Sequenz
  - Teste, ob ein Element y vorhanden ist
  - Sortiere die Elemente x_i
  - ...

(1.2) Lineare Strukturen

- (1.2.1) Listen
- (1.2.2) Warteschlangen
- (1.2.3) Stacks

(1.2.1) Listen

- $L = \{x_1, ..., x_n\}$
- Zugriff auf beliebige Elemente x_i
  - Per Index (random access)
    - Get(i);
  - Per Marker (sequential access)
    - GetFirst();
    - GetNext();
    - GetPrevious();

Random Access

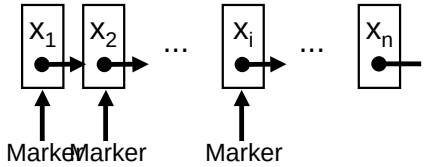
- Implementierung durch ***Arrays*** L[ ]
- Get(i) = L[i]
- Nachteile
  - Elemente löschen erzeugt Lücken oder alle Elemente mit höherem Index müssen verschoben werden *(garbage collection)*
  - Statische Obergrenze für Listenlänge

Sequential Access

- Implementierung durch ***Pointer*** oder ***Container***
- ***Marker*** zeigt auf aktuelle Position
- Nachteil: Elementzugriff erfordert lineare Suche (kann jedoch meistens vermieden werden, z.B. „foreach“).
- Vorteil: beliebiges Erweitern und Löschen

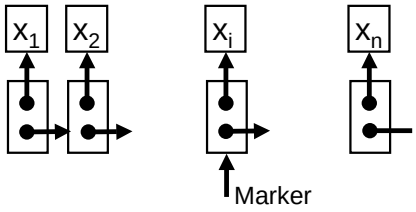
Sequential Access

- Pointer



Sequential Access

- Container



Listen

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$

- Create : $\rightarrow L$
- Get : $L \rightarrow W$
- Reset : $L \rightarrow L$
- Next : $L \rightarrow L$
- Insert : $W \times L \rightarrow L$
- Delete : $L \rightarrow L$
- Empty : $L \rightarrow$ Bool
- IsLast : $L \rightarrow$ Bool

Listen

- Achtung: der Marker beschreibt einen ***Zustand***, was sich mit ***funktionalen*** Axiomen schlecht formulieren läßt.
- *Standard-Trick:* führe ein zusätzliches Prädikat Insert* ein, dass aber ***keine*** weitere Listen-Funktion darstellt.

Listen

Empty(Create()) = true
Empty(Insert(x,z)) = false
Empty(Insert*(x,z)) = false

IsLast(Create()) = true
IsLast(Insert(x,z)) = false
IsLast(Insert*(x,z)) = IsLast(z)

Get(Insert(x,z)) = x
Get(Insert*(x,z)) = Get(z)

Delete(Create()) = Create()
Delete(Insert(x,z)) = z
Delete(Insert*(x,z)) = Insert*(x,Delete(z))

Next(Create()) = Create()
Next(Insert(x,z)) = Insert*(x,z)
Next(Insert*(x,z)) = Insert*(x,Next(z))

Reset(Create()) = Create
Reset(Insert(x,z)) = Insert(x,z)
Reset(Insert*(x,z)) = Insert(x,Reset(z))

Insert(x,Insert*(y,z)) = Insert*(y,Insert(x,z))

Listen

- Satz: Jede Liste hat die Form
  $Insert^*(x_1, ... Insert^*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create())))))$
- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Get()

$Get(Insert^*(x_1, ... Insert^*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create())))))$

$Get(... Insert^*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))$

$Get(Insert^*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))$

$Get(Insert(x_{i+1}, ... Insert(x_n,Create()))))$

x_{i+1}

Next()

Next(Insert*(x_1, ... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1,Next(... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Next(Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert*(x_i,Next(Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert*(x_i,Insert*(x_{i+1}, ... Insert(x_n,Create()))))

Insert()

Insert(y,Insert*(x_1, ... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1,Insert(y, ... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert(y,Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert*(x_i,Insert(y,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Delete()

Delete(Insert*(x_1, ... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1,Delete(... Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Delete(Insert*(x_i,Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert*(x_i,Delete(Insert(x_{i+1}, ... Insert(x_n,Create()))))))

Insert*(x_1, ... Insert*(x_i, ... Insert(x_n,Create()))))))

Weitere Funktionen ...

- Overwrite(y,Insert(x,z)) = Insert(y,z)
  Overwrite(y,Insert*(x,z)) = Insert*(x,Overwrite(y,z))
- Join(Create(),z) = z
  Join(Insert(x,y),z) = Insert*(x,Join(y,z))
  Join(Insert*(x,y),z) = Insert*(x,Join(y,z))
- ...

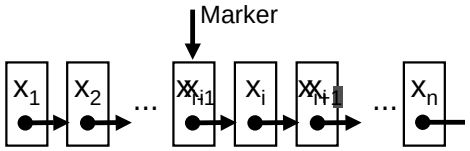
Implementierung

- Finden, Lesen, Überschreiben
- Löschen (Sonderfälle bei leerer Liste)
- Einfügen (Sonderfälle an den Enden)
- Anchor, Sentinel
- Einfach / Doppelt verkettete Listen

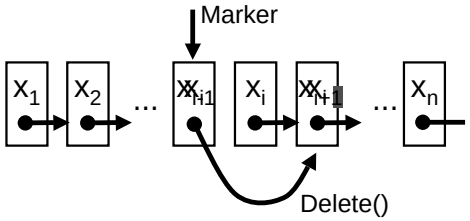
Einfach verkettete Liste

- public class Element {
  Datentyp X;
  Element next;
  }

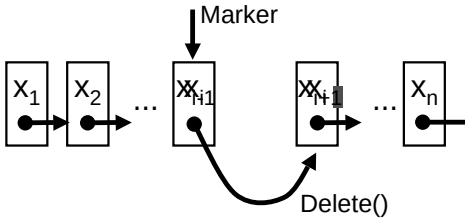
Einfach verkettete Listen



Einfach verkettete Listen



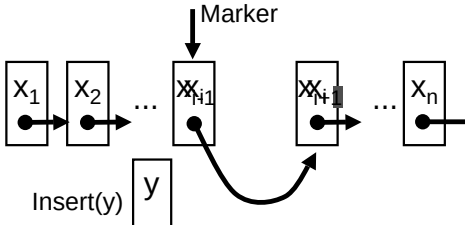
Einfach verkettete Listen



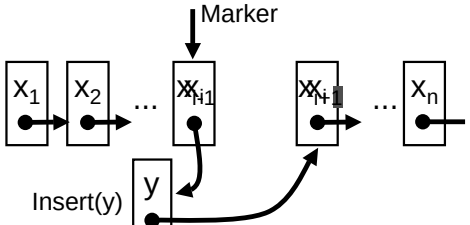
Einfach verkettete Listen

- Delete()
  Marker.next = Marker.next.next;

Einfach verkettete Listen



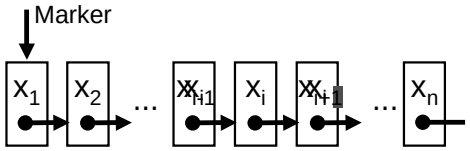
Einfach verkettete Listen



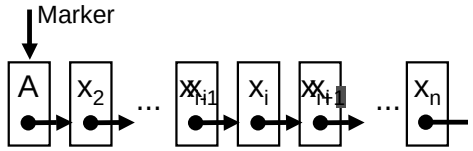
Einfach verkettete Listen

- Insert(Y)
  Y.next = Marker.next;
  Marker.next = Y;

Modifikation am Listenanfang



Modifikation am Listenanfang



Anchor
(erstes Element enthält keine Daten)

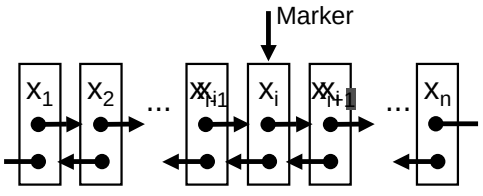
Doppelt verkettete Listen

- Flexiblerer Zugriff (upstream/downstream)
  - Next(), Prev()
- Bisher: Einfügen/Löschen ***nach*** dem Marker (wg. Pointer update)
- Jetzt: Marker zeigt auf ***aktuelles*** Element

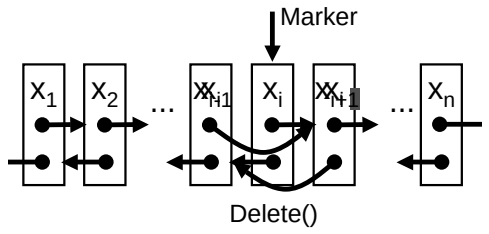
Doppelt verkettete Liste

```
• public class Element {
      Datentyp   X;
      Element    prev, next;
  }
```

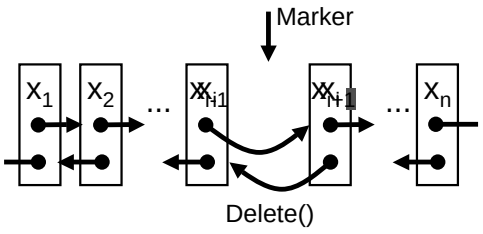

Doppelt verkettete Listen



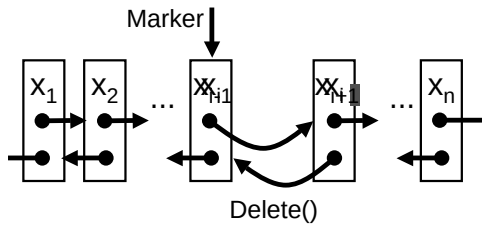
Doppelt verkettete Listen



Doppelt verkettete Listen



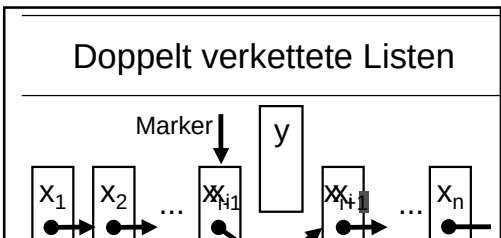
Doppelt verkettete Listen



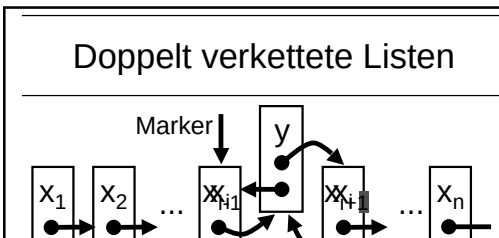
Doppelt verkettete Listen

- Delete()
  Marker.prev.next = Marker.next
  Marker.next.prev = Marker.prev
  Marker = Marker.prev

Doppelt verkettete Listen



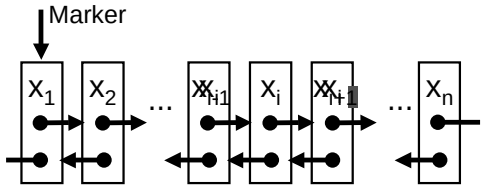
Doppelt verkettete Listen



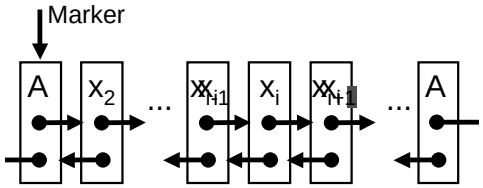
Doppelt verkettete Listen

- Insert(Y)
  Y.prev = Marker
  Y.next = Marker.next
  Y.prev.next = Y
  Y.next.prev = Y

Modifikation an den Enden



Modifikation an den Enden



Anchor / Sentinel (keine Daten)

(1.2.2) Warteschlange

- Liste mit eingeschränkter Funktionalität
- Einfügen nur am Ende
- Auslesen/Entfernen nur am Anfang
- „First in, first out“ (FIFO)

Warteschlange

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$
- Create : $\rightarrow$ L
- Enq : W x L $\rightarrow$ L
- Deq : L $\rightarrow$ L
- Get : L $\rightarrow$ W
- Empty : L $\rightarrow$ Bool

Warteschlange

- Empty(Create()) = true
  Empty(Enq(x,z)) = false
- Deq(Enq(x,Create())) = Create()
  Deq(Enq(x,z)) = Enq(x,Deq(z)) if $z \neq \{\}$
- Get(Enq(x,Create())) = x
  Get(Enq(x,z)) = Get(z) if $z \neq \{\}$

Warteschlange

- Satz: Jede Warteschlange hat die Form
  $Enq(x_n,Enq(x_{n-1}, ... Enq(x_1,Create())))$
- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Warteschlange

$Get(Enq(x_n,Enq(x_{n-1}, ... Enq(x_1,Create()))))$

$Enq(x_n,Get(Enq(x_{n-1}, ... Enq(x_1,Create()))))$

$Enq(x_n,Enq(x_{n-1},Get(... Enq(x_1,Create()))))$

$Enq(x_n,Enq(x_{n-1}, ... Get(Enq(x_1,Create()))))$

x_1

Warteschlange

$Deq(Enq(x_n,Enq(x_{n-1}, ... Enq(x_1,Create()))))$

$Enq(x_n,Deq(Enq(x_{n-1}, ... Enq(x_1,Create()))))$

$Enq(x_n,Enq(x_{n-1},Deq(... Enq(x_1,Create()))))$

$Enq(x_n,Enq(x_{n-1}, ... Deq(Enq(x_1,Create()))))$

$Enq(x_n,Enq(x_{n-1}, ... Create()))))$

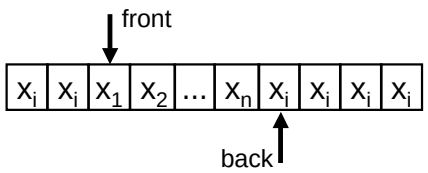
Array-Implementierung

- Da nur am Anfang eingefügt und am Ende gelöscht wird, ist keine garbage collection notwendig.
- Maximale Länge wird als bekannt vorausgesetzt.

Array-Implementierung

- public class Schlange {
  Datentyp S[Länge];
  int front, back;
  }

Array-Implementierung



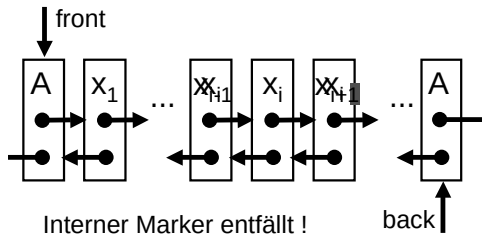
Array-Implementierung

- Create()
  front = back = 0;
- Empty()
  return (front == back);
- Get()
  if (front != back)
  return S[front];
  else
  Q-EMPTY-ERROR;

Array-Implementierung

- Deq()
  if (front != back)
  front = (front + 1) % Länge;
  else
  Q-EMPTY-ERROR;
- Enq(x)
  S[back] = x;
  back = (back + 1) % Länge;
  if (front == back)
  Q-FULL-ERROR

Pointer-Implementierung



Interner Marker entfällt !

(1.2.3) Stack (*Keller* (?))

- Liste mit eingeschränkter Funktionalität
- Einfügen nur am Anfang
- Auslesen/Entfernen nur am Anfang
- „Last in, first out“ (LIFO)

Stack

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$
- Create : $\rightarrow$ L
- Push : W x L $\rightarrow$ L
- Pop : L $\rightarrow$ L
- Top : L $\rightarrow$ W
- Empty : L $\rightarrow$ Bool

Stack

- Empty(Create()) = true
  Empty(Push(x,z)) = false
- Pop(Push(x,z)) = z
- Top(Push(x,z)) = x

Stack

- Satz: Jeder Stack hat die Form
  $Push(x_1,Push(x_2, ... Push(x_n,Create())))$
- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

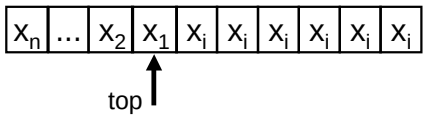
Array-Implementierung

- Da nur am Anfang eingefügt und gelöscht wird, ist keine garbage collection notwendig.
- Maximale Größe wird als bekannt vorausgesetzt.

Array-Implementierung

```
• public class Stack {
      Datentyp   S[Größe];
      int        top;
  }
```


Array-Implementierung



top

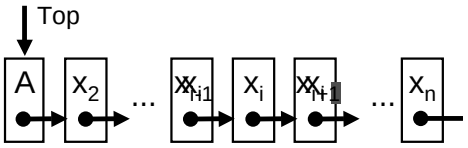
Array-Implementierung

```
• Create()
      top = -1;
• Empty()
      return (top == -1);
• Top()
      if (top != -1)
          return (S[top]);
      else
          STACK-EMPTY-ERROR
```


Array-Implementierung

```
• Push(x)
      top++;
      if (top < Größe)
          S[top] = x;
      else
          STACK-FULL-ERROR
• Pop()
      if (top != -1)
          top--;
      else
          STACK-EMPTY-ERROR
```

Pointer-Implementierung



Suche im Labyrinth

- Geg:
  - Spielfläche W = [0..n] x [0..m]
  - mögliche Positionen P = (x,y) $\in$ W
  - Bewegungsrichtungen: R = {N,S,W,O}
  - Labyrinth ... L : W x R $\rightarrow$ Bool
  - Startposition P_{begin}
  - Zielposition P_{end}
- Ges: Pfad S $\in$ R^k von P_{begin} nach P_{end}

Suche im Labyrinth

- Funktion Go : W x R $\rightarrow$ W beschreibt einen Schritt Go(P,r)=Q von P in die Richtung r nach Q.
- Wenn L(P_{begin},r) = true und Go(P_{begin},r) = Q und es existiert ein Pfad {r_1, ..., r_n} von Q nach P_{end} dann ist {r,r_1, ..., r_n} ein Pfad von P_{begin} nach P_{end}.

Suche im Labyrinth

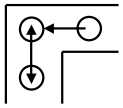
- Annahme: Labyrinth hat keine Zyklen
- Pfad = Liste
- Erweiterung nur am Anfang der Liste
  $\rightarrow$ Pfad = Stack
- Rekursive Formulierung

Rekursiver Algorithmus

```
• Bool Suche(P,S,Q)
      if (P == Q)
          S = Create();
          return true;
      else
          for r ∈ {N,S,W,O}
              if (L(P,r) & Suche(Go(P,r),S,Q))
                  S = Push(r,S);
                  return true;
          return false;
```


Rekursiver Algorithmus

- Problem: *unendliche Suche*
- Lösung: verbiete Schritte zurück.



- „Negative“ Richtung:
  -N = S, -S = N, -W = O, -O = W

Rekursiver Algorithmus

```
• Bool Suche(P,S,Q,r‘)
      if (P == Q)
          S = Create();
          return true;
      else
          for r ∈ {N,S,W,O} \ r‘
               if (L(P,r) & Suche(Go(P,r),S,Q,-r))
                    S = Push(r,S);
                    return true;
          return false;
```

Iterativer Algorithmus

- Rekursion steuert die Suche
- Direkte Lösung unter Verwendung eines Stack
- Ordnung der Richtungen N < O < S < W (next(N) = O, next(O) = S, ...)
- Sortiere Pfade *lexikographisch*

Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

- P = P_{begin}; Q = P_{end}; S = Create(); r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
    while (P != Q) & L(P,r)
      S = Push(r,S); P = Go(P,r); r = ´N´;
    if (P != Q)
      while (r == ´W´) & !Empty(S)
        r = Top(S); P = Go(P,-r); S = Pop(S);
      if (r < ´W´)
        r = next(r);

Iterativer Algorithmus

- P = P_{begin}; Q = P_{end}; S = Create(); r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
    while (P != Q) & L(P,r)
      S = Push(r,S); P = Go(P,r); r = ´N´;
    if (P != Q)
      while (r == ´W´) & !Empty(S)
        r = Top(S); P = Go(P,-r); S = Pop(S);
      if (r < ´W´)
        r = next(r);

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

(1.2) Lineare Strukturen

- (1.2.1) Listen
- (1.2.2) Warteschlangen
- (1.2.3) Stacks

Listen

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$
- Create : $\rightarrow L$
- Get : $L \rightarrow W$
- Reset : $L \rightarrow L$
- Next : $L \rightarrow L$
- Insert : $W \times L \rightarrow L$
- Delete : $L \rightarrow L$
- Empty : $L \rightarrow$ Bool
- IsLast : $L \rightarrow$ Bool

Listen

Empty(Create()) = true
Empty(Insert(x,z)) = false
Empty(Insert*(x,z)) = false

IsLast(Create()) = true
IsLast(Insert(x,z)) = false
IsLast(Insert*(x,z)) = IsLast(z)

Get(Insert(x,z)) = x
Get(Insert*(x,z)) = Get(z)

Delete(Create()) = Create()
Delete(Insert(x,z)) = z
Delete(Insert*(x,z)) = Insert*(x,Delete(z))

Next(Create()) = Create()
Next(Insert(x,z)) = Insert*(x,z)
Next(Insert*(x,z)) = Insert*(x,Next(z))

Reset(Create()) = Create
Reset(Insert(x,z)) = Insert(x,z)
Reset(Insert*(x,z)) = Insert(x,Reset(z))

Insert(x,Insert*(y,z)) = Insert*(y,Insert(x,z))

Normalform

- Satz: Jede Liste hat die Form

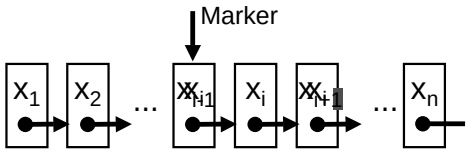
  $\text{Insert*}(x_1, ... \text{Insert*}(x_i,\text{Insert}(x_{i+1}, ... \text{Insert}(x_n,\text{Create}())))))$

- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Einfach verkettete Liste

```
public class Element {
    Datentyp   X;
    Element    next;
}
```

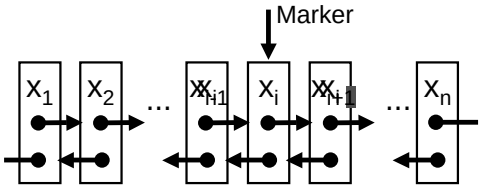

Einfach verkettete Listen



Doppelt verkettete Liste

- public class Element {
  Datentyp X;
  Element prev, next;
  }

Doppelt verkettete Listen



Warteschlange

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$
- Create : $\rightarrow$ L
- Enq : W x L $\rightarrow$ L
- Deq : L $\rightarrow$ L
- Get : L $\rightarrow$ W
- Empty : L $\rightarrow$ Bool

Warteschlange

- Empty(Create()) = true
  Empty(Enq(x,z)) = false
- Deq(Enq(x,Create())) = Create()
  Deq(Enq(x,z)) = Enq(x,Deq(z)) if $z \neq \{\}$
- Get(Enq(x,Create())) = x
  Get(Enq(x,z)) = Get(z) if $z \neq \{\}$

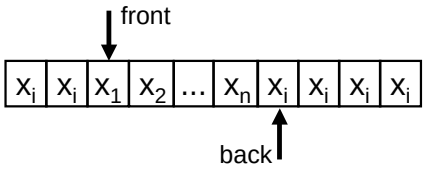
Normalform

- Satz: Jede Warteschlange hat die Form
  $Enq(x_n,Enq(x_{n-1}, ... Enq(x_1,Create())))$
- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Array-Implementierung

- public class Schlange {
  Datentyp S[Länge];
  int front, back;
  }

Array-Implementierung



Stack

- Wertebereich : $L = \{\} \cup W \cup W^2 \cup W^3 \cup ...$
- Create : $\rightarrow L$
- Push : $W \times L \rightarrow L$
- Pop : $L \rightarrow L$
- Top : $L \rightarrow W$
- Empty : $L \rightarrow$ Bool

Stack

- Empty(Create()) = true
  Empty(Push(x,z)) = false
- Pop(Push(x,z)) = z
- Top(Push(x,z)) = x

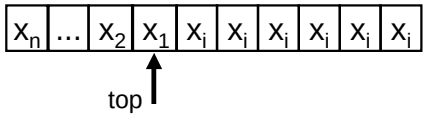
Normalform

- Satz: Jeder Stack hat die Form
  $Push(x_1,Push(x_2, ... Push(x_n,Create())))$
- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Array-Implementierung

```
• public class Stack {
      Datentyp   S[Größe];
      int        top;
  }
```


Array-Implementierung



Suche im Labyrinth

- Geg:
  - Spielfläche W = [0..n] x [0..m]
  - mögliche Positionen P = (x,y) $\in$ W
  - Bewegungsrichtungen: R = {N,S,W,O}
  - Labyrinth ... L : W x R $\rightarrow$ Bool
  - Startposition P_{begin}
  - Zielposition P_{end}
- Ges: Pfad S $\in$ R^k von P_{begin} nach P_{end}

Suche im Labyrinth

- Funktion Go : W x R $\rightarrow$ W beschreibt einen Schritt Go(P,r)=Q von P in die Richtung r nach Q.
- Wenn $L(P_{begin},r)$ = true und $Go(P_{begin},r) = Q$ und es existiert ein Pfad $\{r_1, ..., r_n\}$ von Q nach P_{end} dann ist $\{r,r_1, ..., r_n\}$ ein Pfad von P_{begin} nach P_{end}.

Suche im Labyrinth

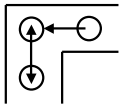
- Annahme: Labyrinth hat keine Zyklen
- Pfad = Liste
- Erweiterung nur am Anfang der Liste
  $\rightarrow$ Pfad = Stack
- Rekursive Formulierung

Rekursiver Algorithmus

```
• Bool Suche(P,S,Q)
    if (P == Q)
        S = Create();
        return true;
    else
        for r ∈ {N,S,W,O}
            if (L(P,r) & Suche(Go(P,r),S,Q))
                S = Push(r,S);
                return true;
        return false;
```


Rekursiver Algorithmus

- Problem: *unendliche Suche*
- Lösung: verbiete Schritte zurück.



- „Negative“ Richtung:
  -N = S, -S = N, -W = O, -O = W

Rekursiver Algorithmus

```
• Bool Suche(P,S,Q,r')
    if (P == Q)
        S = Create();
        return true;
    else
        for r ∈ {N,S,W,O} \ r'
            if (L(P,r) & Suche(Go(P,r),S,Q,-r))
                S = Push(r,S);
                return true;
        return false;
```

Iterativer Algorithmus

- Rekursion steuert die Suche
- Direkte Lösung unter Verwendung eines Stack
- Ordnung der Richtungen N < O < S < W (next(N) = O, next(O) = S, ...)
- Sortiere Pfade *lexikographisch*

Lexikographische Reihenfolge

- N
- NN
- NNN
- ...
- NNNNO
- NNNNON
- ...
- NNNNOO
- ...
- NNNNOS
- ..
- NNNNOW
- ...
- NNNNS
- ...
- NNNO
- ...
- WW

Iterativer Algorithmus

```
• P = $P_{begin}$;  Q = $P_{end}$;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = $P_{begin}$;  Q = $P_{end}$;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = $P_{begin}$;  Q = $P_{end}$;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = $P_{begin}$;  Q = $P_{end}$;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```

Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next(r);
```


Iterativer Algorithmus

```
• P = P_begin;  Q = P_end;  S = Create();  r = ´N´;
  while (P != Q) & (L(P,r) | (r < ´W´) | !Empty(S))
      while (P != Q) & L(P,r)
          S = Push(r,S);  P = Go(P,r);  r = ´N´;
      if (P != Q)
          while (r == ´W´) & !Empty(S)
              r = Top(S);  P = Go(P,-r);  S = Pop(S);
          if (r < ´W´)
              r = next®;
```


Stack

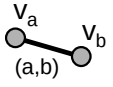
- Merke: Rekursive Algorithmen lassen sich in der Regel mit einer ***Stack*** Datenstruktur auch iterativ formulieren.
- Das LIFO-Prinzip entspricht der Abarbeitungsreihenfolge der geschachtelten Prozeduren (was zuletzt aufgerufen wird, wird als erstes bearbeitet).
- Beispiel: systematische Suche in einem Labyrinth (bei Sackgassen zurückgehen).

(1.3) Bäume

- *Hierarchische* Datenstruktur
  - Zusammenfassung von Gruppen (z.B. Bund / Länder / Gemeinden)
  - Eindeutige Schachtelung
- Typische Anwendungen
  - Such- / Entscheidungsprobleme
  - Strukturierte Aufzählung
  - Komplexitätsreduktion

Definitionen

- Menge von Knoten $V = \{v_1, ..., v_n\}$ ($v_i \in W$ ... beliebiger Datentyp)
- Menge von Kanten $E = \{(a_1,b_1), ... (a_m,b_m)\}$ ($a_i,b_i \in IN$ ... Knotenindizes)
- Falls $(a,b) \in E$, dann ist
  - v_a Vorgänger von v_b
  - v_b Nachfolger von v_a



Definitionen

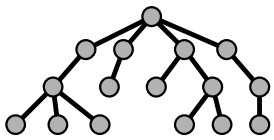
- Eine Folge von Kanten (i_1,i_2), (i_2,i_3), ... (i_{k-1},i_k) heißt Pfad von v_{i1} nach v_{ik}
- Für $i_1 = i_k$ ist dieser Pfad ein gerichteter Zyklus
- Existieren zwei *verschiedene* Pfade P_1 und P_2 von v_{i1} nach v_{ik} so bilden sie zusammen einen ungerichteten Zyklus

Definitionen

- Ein Baum B=(V,E) besteht aus einer Menge von Knoten V und einer Menge von Kanten E, so dass ...
  - Es existieren keine gerichteten oder ungerichteten Zyklen
  - Jeder normale Knoten hat genau einen Vorgänger
  - Es existiert genau ein Wurzelknoten, der keinen Vorgänger hat

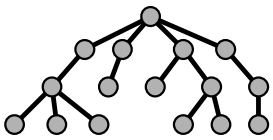
Abgeleitete Eigenschaften

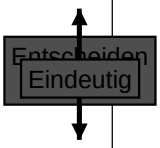
- Für jeden Knoten existiert ein eindeutiger Pfad, der ihn mit dem Wurzelknoten verbindet.



Abgeleitete Eigenschaften

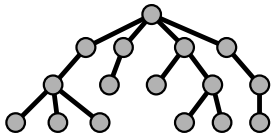
- Für jeden Knoten existiert ein eindeutiger Pfad, der ihn mit dem Wurzelknoten verbindet.





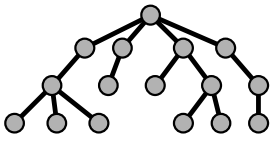
Abgeleitete Eigenschaften

- Für jeden Knoten existiert ein eindeutiger Pfad, der ihn mit dem Wurzelknoten verbindet.



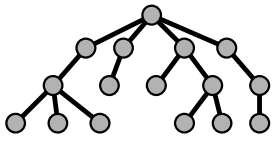
Abgeleitete Eigenschaften

- Jeder Knoten ist Wurzelknoten eines zugehörigen Sub-Baumes (wird später zur Definition des ADT verwendet).



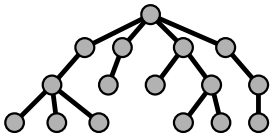
Abgeleitete Eigenschaften

- Jeder Knoten ist Wurzelknoten eines zugehörigen Sub-Baumes (wird später zur Definition des ADT verwendet).



Abgeleitete Eigenschaften

- Jeder Knoten ist Wurzelknoten eines zugehörigen Sub-Baumes (wird später zur Definition des ADT verwendet).



Weitere Begriffe

- Die Tiefe eines Knotens ist gleich der Länge des zugehörigen Pfades von der Wurzel.
- Die Höhe eines Baumes ist gleich der Tiefe des tiefsten Knotens.
- Der Grad eines Knotens ist die Anzahl seiner Nachfolger.

Weitere Begriffe

- Der eine Knoten ohne Vorgänger heißt Wurzelknoten
- Knoten ohne Nachfolger heißen Blätter
- Alle anderen Knoten sind innere Knoten

Weitere Begriffe

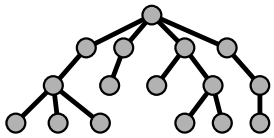
- Seien w_1, ..., w_k die Nachfolger des Knotens v, dann gilt für alle w_i: $\text{height}(\text{subtree}(w_i)) \leq \text{height}(\text{subtree}(v)) - 1$
- Gilt außerdem für alle w_i: $\text{height}(\text{subtree}(w_i)) \geq \text{height}(\text{subtree}(v)) - 2$ dann heißt der Baum balanciert.

Weitere Begriffe

- Gilt für alle Knoten v mit Nachfolgern w_1, ... w_k eines Baumes: $\text{height}(\text{subtree}(w_i)) = \text{height}(\text{subtree}(v)) - 1$ so heißt er vollständig.
- Bei vollständigen Bäumen ist allerdings erlaubt, dass für $\text{height}(\text{subtree}(v)) = 1$ die Sub-Bäume $\text{subtree}(w_i)$ entweder die Höhe 0 haben oder leer sind.

Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse



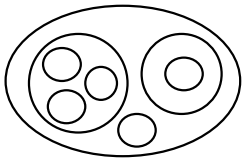
Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse

A(B(C,D),E(F,G))

Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse



Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse

1) A
  1.1) B
  1.2) C
2) D
  2.1) E
    2.1.1) F

Konventionen

- In den meisten Anwendungen betrachten wir nur Bäume mit beschränktem oder konstantem Knotengrad (z.B. *Binärbäume*).
- Wie bei Listen benötigen wir einen Marker, der anzeigt, ***wo*** die nächste Operation angewendet werden soll. In der Regel ergibt sich dieser aus dem Algorithmus.

Datentyp: Binärbaum

- Knotentyp ... beliebig
- Create : $\rightarrow$ Tree
- Node : Tree x Val x Tree $\rightarrow$ Tree
- Left : Tree $\rightarrow$ Tree
- Right : Tree $\rightarrow$ Tree
- Value : Tree $\rightarrow$ Val
- Empty : Tree $\rightarrow$ Bool

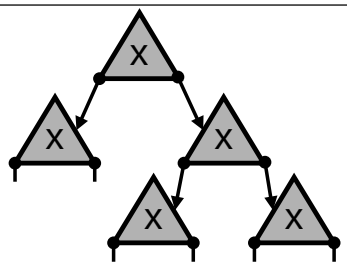
Datentyp: Binärbaum

- Empty(Create()) = true
  Empty(Node(L,V,R)) = false
- Left(Node(L,V,R)) = L
- Right(Node(L,V,R)) = R
- Value(Node(L,V,R)) = V

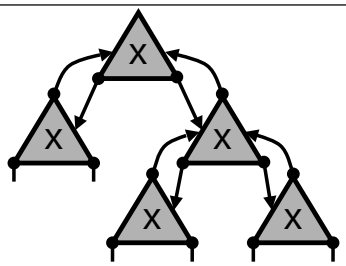
Implementierung

- Pointer (einfach / doppelt)
- Anchor / Sentinel
- Knotengrade (fest / variabel)
- Array-Implementierung für vollständige Bäume

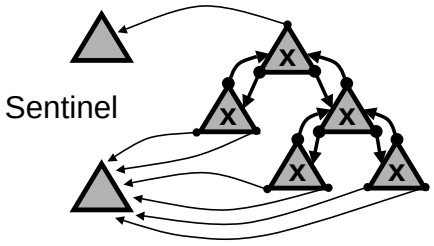
Implementierung



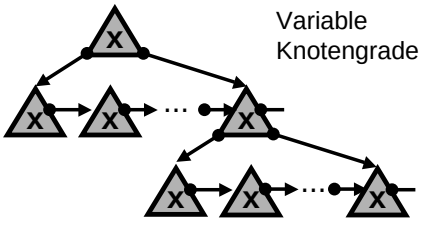
Implementierung



Implementierung



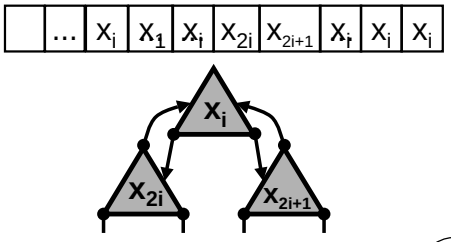
Implementierung



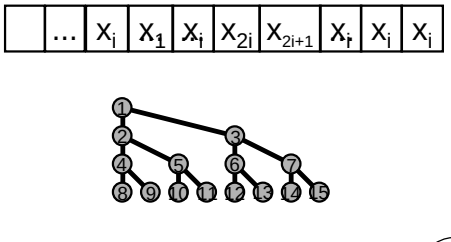
Array-Implementierung

- Vollständige Bäume (z.B. *Binärbäume*)
- N(k) = Anzahl der Knoten der Tiefe k
- $N(k) = 2\,N(k-1) \rightarrow N(k) = 2^k$
- Speichere die Knoten der Tiefe k in den Array-Einträgen $A[2^k \ldots 2^{k+1}-1]$
- Jeder Knoten A[i] findet seine Nachfolger in A[2i] und A[2i+1]

Array-Implementierung



Array-Implementierung

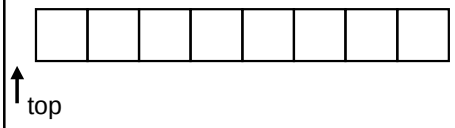


Beispiel: Arithmetische Terme

- „normale“ (Infix) Notation:
  A + B * (C + D) / E
- Postfix Notation: A B C D + * E / +
- Präfix Notation: + A / * B + C D E
  (Polnische Notation)
- Vorteil: keine Klammern notwendig !

Stack-Computer

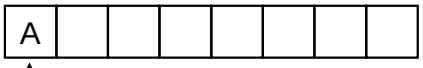
A B C D + * E / +



top

Stack-Computer

A B C D + * E / +



top

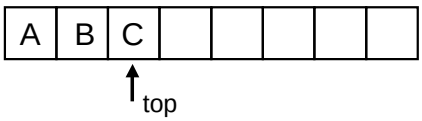
Stack-Computer

A B C D + * E / +

top

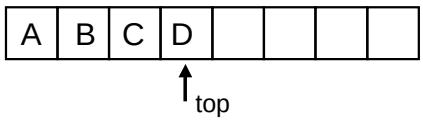
Stack-Computer

A B C D + * E / +



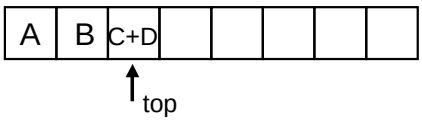
Stack-Computer

A B C D + * E / +



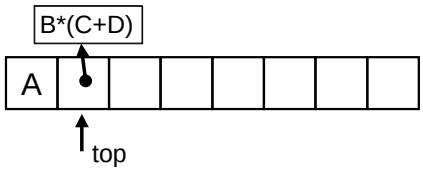
Stack-Computer

A B C D + * E / +



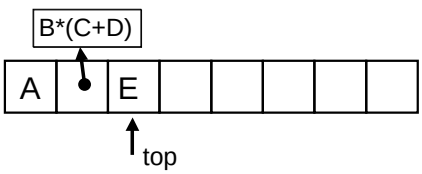
Stack-Computer

A B C D + * E / +



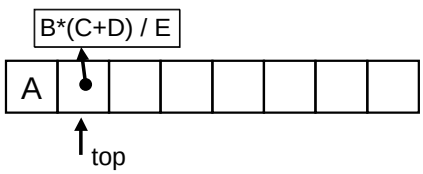
Stack-Computer

A B C D + * E / +



Stack-Computer

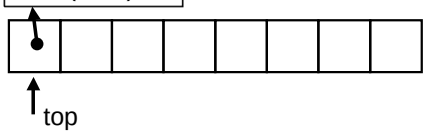
A B C D + * E / +



Stack-Computer

A B C D + * E / +

A+B*(C+D) / E

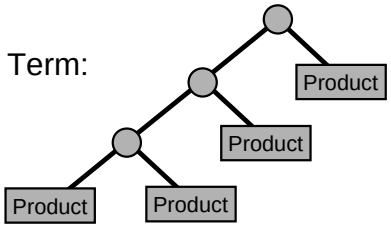


Arithmetische Terme

- Term = Variable *oder*
  Summe von Produkten
- Produkt = Multiplikation von Termen
- Hierarchische Struktur → Baumstruktur
  - Grundoperationen : IR x IR → IR
  - Binärbaum
- Rekursive Struktur → Rekursive Prozedur

Term → Binärbaum

Term:



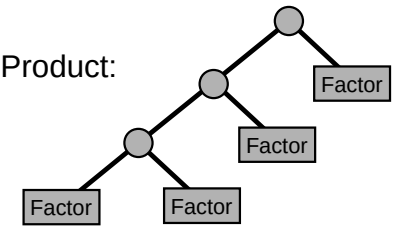
Term → Binärbaum

- BinTree ReadTerm()

```
L = ReadProduct();
while ((op = getChar()) in { `+` , `-` })
      R = ReadProduct();
      L  = Node(L,op,R);
return L;
```


Term → Binärbaum

Product:



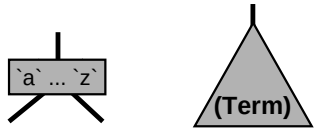
Term → Binärbaum

- BinTree ReadProduct()

```
L = ReadFactor();
while ((op = getChar()) in { `*` , `÷` })
      R = ReadFactor();
      L  = Node(L,op,R);
return L;
```

Term → Binärbaum

Factor:



Term → Binärbaum

- BinTree ReadFactor()

```
c = getChar();
if ( c in { `a`, ..., `z` } )
      return Node(Create(),c,Create());
else  // c == `(`
      L = ReadTerm();
      c = getChar();  // c == `)`
return L;
```


Term → Binärbaum

A + B * (C + D) / E

Term → Binärbaum

A + B * (C + D) / E

A

ReadTerm()
ReadProduct()
ReadFactor()

Term → Binärbaum

A + B * (C + D) / E ⊕

A

ReadTerm()

Term → Binärbaum

A + B * (C + D) / E ⊕

A

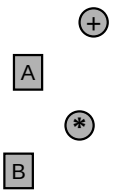
ReadTerm()
ReadProduct()
ReadFactor()

B

Term → Binärbaum

A + B * (C + D) / E

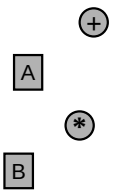
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B * (C + D) / E

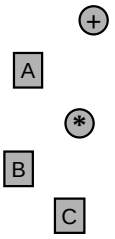
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B * (C + D) / E

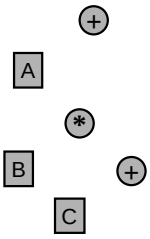
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B * (C + D) / E

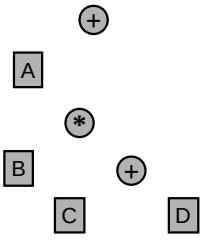
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()



Term → Binärbaum

A + B * (C + D) / E

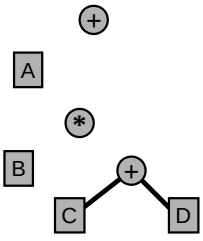
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B * (C + D) / E

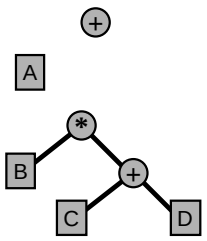
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B * (C + D) / E

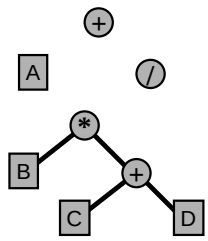
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B * (C + D) / E

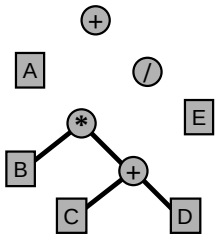
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B * (C + D) / E

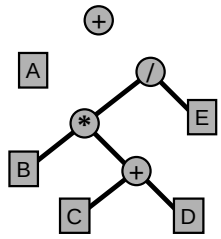
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B * (C + D) / E

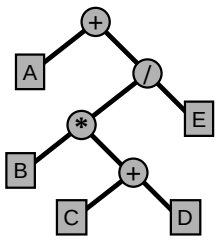
ReadTerm()
ReadProduct()



Term → Binärbaum

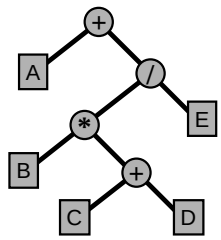
A + B * (C + D) / E

ReadTerm()

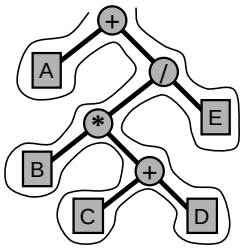


Term → Binärbaum

A + B * (C + D) / E



Traversierung

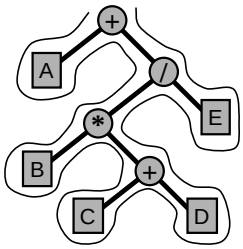


Traversierung

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        Traverse(Left(T));
        Traverse(Right(T));
```


Traversierung



Traversierung

- Operator (= Knotenwert) steht
  - vor den Argumenten → Präfix
  - zwischen den Argumenten → Infix
  - nach den Argumenten → Postfix
- Einfache Varianten der rekursiven Traverse

Präfix

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        output(Value(T));
        Traverse(Left(T));
        Traverse(Right(T));
```


Infix

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        Traverse(Left(T));
        output(Value(T));
        Traverse(Right(T));
```

Infix

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        output(`(`);
        Traverse(Left(T));
        output(Value(T));
        Traverse(Right(T));
        output(`)`);
```

Achtung: Klammern für korrekte Syntax notwendig !!!

Postfix

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        Traverse(Left(T));
        Traverse(Right(T));
        output(Value(T));
```


Traversierungsstrategien

- Rekursive Algorithmen
  - Depth-first
    - Präfix / Infix / Postfix
- Nicht-rekursive Algorithmen
  - Depth-first (Stack)
  - Breadth-first (Queue)

Depth-First

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        Traverse(Left(T));
        Traverse(Right(T));
```


Depth-First

- Traverse(BinTree T)

```
S = CreateStack();  S = Push(T,S);
DepthFirst(S);
```

- DepthFirst(Stack S)

```
while (!Empty(S))
    T = Top(S);  S = Pop(S);
    ... do something with T ...
    if (!Empty(Right(T))  S = Push(Right(T),S);
    if (!Empty(   Left(T))  S = Push(  Left(T),S);
```


Breadth-First

- Zähle die Knoten nach ihrer Tiefe sortiert auf, d.h. alle Knoten mit Tiefe k vor dem ersten Knoten mit Tiefe k+1
- Labyrinth-Suche: Gehe nicht bis zur Sackgasse (depth-first), sondern probiere erst alle Pfade der Länge k vor den Pfaden mit Länge k+1 ...
- Verhindert unendliches Suchen

Breadth-First

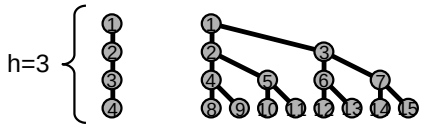
- Traverse(BinTree T)
  Q = CreateQueue(); Q = Enq(T,Q);
  BreadthFirst(Q);
- BreadthFirst(Queue Q)
  while (!Empty(Q))
  T = Get(Q); Q = Deq(Q);
  ... do something with T ...
  if (!Empty(Left(T)) Q = Enq(Left(T),Q);
  if (!Empty(Right(T)) Q = Enq(Right(T),Q);

Suchbäume

- Bäume ermöglichen ***wesentlich*** schnelleren Zugriff auf Elemente als bei lineare Strukturen
- Sortiere die Knoten (kommt später)
  - 1-dimensional
  - k-dimensional

Anzahl der Knoten

- Minimale Anzahl von Knoten in einem Binärbaum der Höhe h: $N_{min}(h) = h+1$
- Maximale Anzahl: $N_{max}(h) = 2^{h+1} -1$

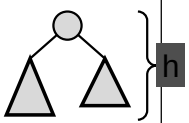


Höhe des Baumes

- Für n Knoten ist die maximale Höhe des Binärbaumes: $H_{max}(n) = n-1$
- Die minimale Höhe:
  $H_{min}(n) = \lceil \log(n+1)/\log(2) \rceil - 1$
- Die Tiefe+1 eines Knoten beschreibt die Anzahl der Zugriffe, um ihn zu finden.

Fragen der Effizienz

- **Optimaler** Zugriff bei **vollständigen** Bäumen. Diese sind aber nicht immer einfach zu konstruieren.
- Balancierte Bäume ...
  $N_{bal,max}(h) = 2^{h+1} -1$
  $N_{bal,min}(h) = 1 + N_{bal,min}(h-1) + N_{bal,min}(h-2)$



Fragen der Effizienz

- Minimale Anzahl von Knoten

$$N_{bal,min}(h) = 1 + N_{bal,min}(h-1) + N_{bal,min}(h-2)$$
$$\geq 2\, N_{bal,min}(h-2)$$
$$\geq 4\, N_{bal,min}(h-4)$$
$$\geq \begin{cases} 2^{(h-1)/2}\, N_{bal,min}(1) & \text{... H ungerade} \\ 2^{h/2}\, N_{bal,min}(0) & \text{... H gerade} \end{cases}$$
$$\geq \sqrt{2}^{\,h}$$

Fragen der Effizienz

- Minimale Anzahl von Knoten

  $N_{bal,min}(h) \geq \sqrt{2}^{\,h}$

- Maximale Höhe für n Knoten

  $H_{bal,max}(n) \leq \lceil \log(n)/\log(\sqrt{2}) \rceil$

  $= 2 \lceil \log(n)/\log(2) \rceil$

  $= 2\, H_{min}(n)$

(1.3) Bäume

- *Hierarchische* Datenstruktur
  - Zusammenfassung von Gruppen (z.B. Bund / Länder / Gemeinden)
  - Eindeutige Schachtelung
- Typische Anwendungen
  - Such- / Entscheidungsprobleme
  - Strukturierte Aufzählung
  - Komplexitätsreduktion

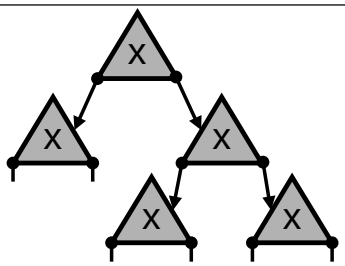
Definitionen

- Ein Baum B=(V,E) besteht aus einer Menge von Knoten V und einer Menge von Kanten E, so dass ...
  - Es existieren keine gerichteten oder ungerichteten Zyklen
  - Jeder normale Knoten hat genau einen Vorgänger
  - Es existiert genau ein Wurzelknoten, der keinen Vorgänger hat

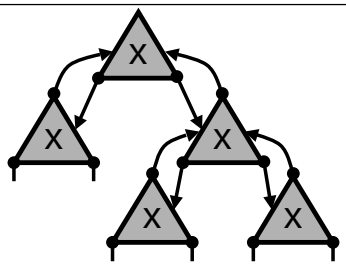
Datentyp: Binärbaum

- Knotentyp ... beliebig
- Create : $\rightarrow$ Tree
- Node : Tree x Val x Tree $\rightarrow$ Tree
- Left : Tree $\rightarrow$ Tree
- Right : Tree $\rightarrow$ Tree
- Value : Tree $\rightarrow$ Val
- Empty : Tree $\rightarrow$ Bool

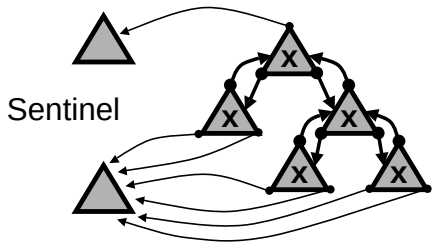
Implementierung



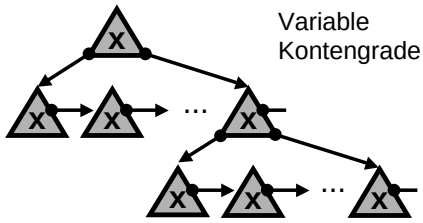
Implementierung



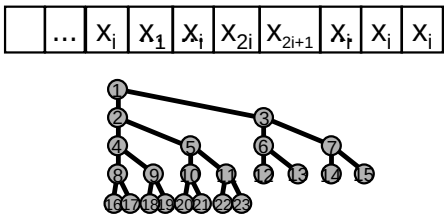
Implementierung



Implementierung



Array-Implementierung



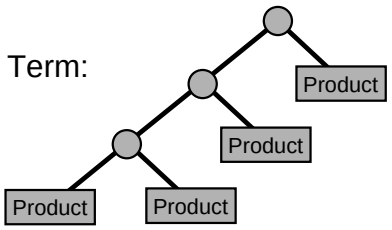
Beispiel: Arithmetische Terme

- „normale“ (Infix) Notation:
  A + B * (C + D) / E
- Postfix Notation: A B C D + * E / +
- Präfix Notation: + A / * B + C D E
  (Polnische Notation)
- Vorteil: keine Klammern notwendig !

Arithmetische Terme

- Term = Variable *oder*
  Summe von Produkten
- Produkt = Multiplikation von Termen
- Hierarchische Struktur → Baumstruktur
  - Grundoperationen : IR x IR → IR
  - Binärbaum
- Rekursive Struktur → Rekursive Prozedur

Term → Binärbaum



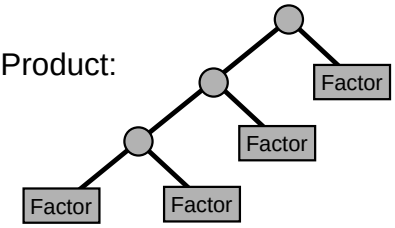
Term → Binärbaum

- BinTree ReadTerm()

```
L = ReadProduct();
while ((op = getChar()) in { `+` , `-` })
      R = ReadProduct();
      L  = Node(L,op,R);
return L;
```

Term → Binärbaum

Product:



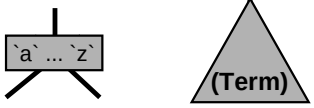
Term → Binärbaum

- BinTree ReadProduct()

```
L = ReadFactor();
while ((op = getChar()) in { `*` , `÷` })
        R = ReadFactor();
        L  = Node(L,op,R);
return L;
```


Term → Binärbaum

Factor:

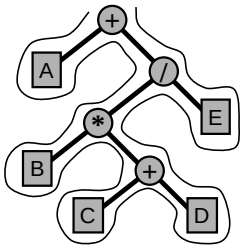


Term → Binärbaum

- BinTree ReadFactor()

```
c = getChar();
if ( c in { `a`, ..., `z` } )
        return Node(Create(),c,Create());
else  // c == `(`
        L = ReadTerm();
        c = getChar();  // c == `)`
return L;
```


Traversierung



Traversierung

- Traverse(BinTree T)

```
if (Leaf(T))
        output(Value(T));
else
        Traverse(Left(T));
        Traverse(Right(T));
```

Traversierung

- Operator (= Knotenwert) steht
  - vor den Argumenten → Präfix
  - zwischen den Argumenten → Infix
  - nach den Argumenten → Postfix
- Einfache Varianten der rekursiven Traverse

Traversierungsstrategien

- Rekursive Algorithmen
  - Depth-first
    - Präfix / Infix / Postfix
- Nicht-rekursive Algorithmen
  - Depth-first (Stack)
  - Breadth-first (Queue)

Depth-First

- Traverse(BinTree T)
  S = CreateStack(); S = Push(T,S);
  DepthFirst(S);
- DepthFirst(Stack S)
  while (!Empty(S))
  T = Top(S); S = Pop(S);
  ... do something with T ...
  if (!Empty(Right(T)) S = Push(Right(T),S);
  if (!Empty(Left(T)) S = Push(Left(T),S);

Breadth-First

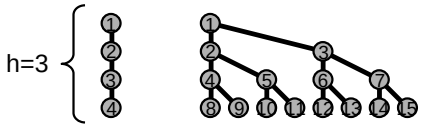
- Traverse(BinTree T)
  Q = CreateQueue(); Q = Enq(T,Q);
  BreadthFirst(Q);
- BreadthFirst(Queue Q)
  while (!Empty(Q))
  T = Get(Q); Q = Deq(Q);
  ... do something with T ...
  if (!Empty(Left(T)) Q = Enq(Left(T),Q);
  if (!Empty(Right(T)) Q = Enq(Right(T),Q);

Suchbäume

- Bäume ermöglichen ***wesentlich*** schnelleren Zugriff auf Elemente als bei lineare Strukturen
- Sortiere die Knoten (kommt später)
  - 1-dimensional
  - k-dimensional

Anzahl der Knoten

- Minimale Anzahl von Knoten in einem Binärbaum der Höhe h: $N_{min}(h) = h+1$
- Maximale Anzahl: $N_{max}(h) = 2^{h+1} - 1$

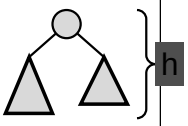


Höhe des Baumes

- Für n Knoten ist die maximale Höhe des Binärbaumes: $H_{max}(n) = n-1$
- Die minimale Höhe:
  $H_{min}(n) = \lceil \log(n+1)/\log(2) \rceil - 1$
- Die Tiefe+1 eines Knoten beschreibt die Anzahl der Zugriffe, um ihn zu finden.

Fragen der Effizienz

- **Optimaler** Zugriff bei **vollständigen** Bäumen. Diese sind aber nicht immer einfach zu konstruieren.
- Balancierte Bäume ...
  - $N_{bal,max}(h) = 2^{h+1} - 1$
  - $N_{bal,min}(h) = 1 + N_{bal,min}(h-1) + N_{bal,min}(h-2)$



Fragen der Effizienz

- Minimale Anzahl von Knoten
  - $N_{bal,min}(h) \geq \sqrt{2}^{\,h}$
- Maximale Höhe für n Knoten
  - $H_{bal,max}(n) \leq \lceil \log(n)/\log(\sqrt{2}) \rceil$
    $= 2 \lceil \log(n)/\log(2) \rceil$
    $= 2\, H_{min}(n)$

Suchen in Suchbäumen

- Knoten sind sortiert angeordnet
  (Sortieralgorithmen später)
  - max(T) = maximaler Knotenwert im Baum T
  - min(T) = minimaler Knotenwert im Baum T
  - Sortierung ... für alle Knoten/Sub-Bäume gilt:
    $\max(\text{Left}(T)) \leq \text{Value}(T) < \min(\text{Right}(T))$
- Depth-first Traversal
  - Mit jedem Test kann bis zu der Hälfte der Knoten ausgeschlossen werden.

Suchen in Suchbäumen

```
• Bool Search(Element X, BinTree T)
    if (X == Value(T))
        return true;
    else if (Leaf(T))
        return false;
    else if (X < Value(T))
        Search(X,Left(T));
    else
        Search(X,Right(T));
```


Einfügen und Löschen

- Bei Suchbäumen bestimmt der Wert des Elementes die Position im Baum (impliziter Marker)
- Einfüge**reihenfolge** bestimmt die Struktur (bessere Algorithmen später)
- Löschen **innerer** Knoten nicht trivial

Einfügen

- Insert(X,Create()) =
  Node(Create(),X,Create())
- Insert(X,Node(L,Y,R)) =
  if (X ≤ Y) Insert(X,L);
  if (X > Y) Insert(X,R);

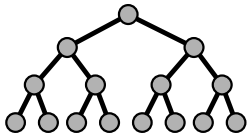
Löschen

- Remove(X,Node(L,Y,R)) =
  if (X < Y) Remove(X,L);
  if (X > Y) Remove(X,R);
  // X ≠ Y
- Max(Node(L,X,Create()) = X
  Max(Node(L,X,R)) = Max(R)
- Min(Node(Create(),X,R) = X
  Min(Node(L,X,R)) = Min(L)

Löschen

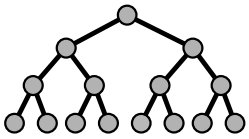
- Remove(X,Node(L,X,R)) =
  if (L ≠ Create())
  Node(Remove(Max(L),L),Max(L),R)
  else if (R ≠ Create())
  Node(L,Min(R),Remove(Min(R),R))
  else
  Create(); // L = R = Create()

Beispiel



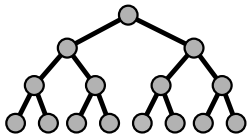
Beispiel

- Remove(,T);



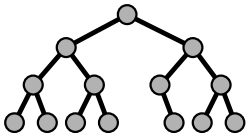
Beispiel

- Remove(,Node(Create(), ,Create());

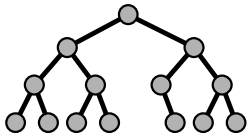


Beispiel

- Create()

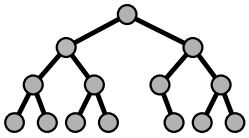


Beispiel



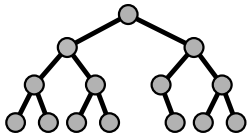
Beispiel

- Remove(○,T);



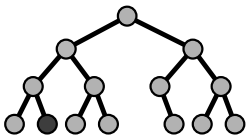
Beispiel

- Remove(○,Node(L, ○,R));



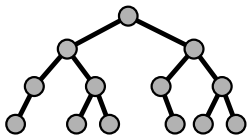
Beispiel

- Remove(○,Node(L, ○,R));
  ● = Max(L);



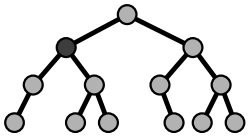
Beispiel

- Remove(○,Node(L, ○,R));
  ● = Max(L);
  Remove(●,L)

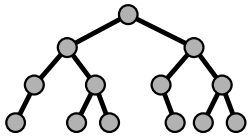


Beispiel

- Remove(,Node(L, ,R));
  = Max(L);
  Node(Remove(,L), ,R)



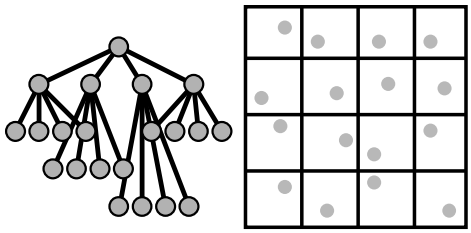
Beispiel



k-dimensionale Suchbäume

- Beispiel: Suche nach Punkten im IR^2 (nächste Tankstelle, ...) oder im IR^k
- Verwende 2^k-näre Bäume
  - Quadtree (IR^2)
  - Octree (IR^3)
  - ...
- kD-Bäume (k-dimensionale Binärbäume)

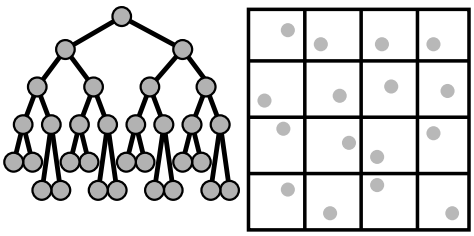
Quadtree



kD-Bäume

- Datentyp: Binärbaum
- Jeder Knoten enthält einen k-dim. Wert
- Sortierung für Knoten T der Tiefe h:
  $\max_h(\text{Left}(T)) \leq \text{Value}_h(T) < \min_h(\text{Right}(T))$
- Subskript h bedeutet: Verwende die (h mod k)-te Koordinate

kD-Bäume



Weitere Baumstrukturen ...

- Baum-Strukturen beschleunigen die Suche in großen Datenmengen.
- Einfügeoperationen beeinflussen die Verteilung der Knoten im Baum.
- Löschungsoperationen können eine komplexe Umstrukturierung des Baumes bewirken.

(1.4) Prioritätsschlangen

- Warteschlangen mit „Vordrängeln“
- Einfügen am Ende der Schlange
- Jedes Element hat eine Priorität
- Deq() liefert immer das Element mit der höchsten Priorität

Prioritätsschlangen

- Elemente: $X \in W' = W \times IR$
  Wertebereich: $L = \{\} \cup W' \cup W'^2 \cup W'^3 \cup \ldots$
- Create : $\rightarrow L$
- Enq* : $W' \times L \rightarrow L$
- Deq : $L \rightarrow L$
- Get : $L \rightarrow W'$
- Empty : $L \rightarrow$ Bool

Prioritätsschlangen

- Empty(Create()) = true
  Empty(Enq(x,z)) = false
- Deq(Enq(x,Create())) = Create()
  Deq(Enq(x,z)) = Enq(x,Deq(z)) if $z \neq \{\}$
- Get(Enq(x,Create())) = x
  Get(Enq(x,z)) = Get(z) if $z \neq \{\}$

Prioritätsschlangen

- Enq() : Standardoperator
- Enq*() : füge Element x bzgl. seiner Priorität x.prio ein.
- Enq*(x,Enq(y,z)) = Enq(y,Enq*(x,z)) if x.prio > y.prio

  Enq*(x,Enq(y,z)) = Enq(x,Enq(y,z)) if x.prio ≤ y.prio

Prioritätsschlangen

- Aufwand steigt proportional zur Länge der Liste (= Zahl der Axiom-Anwendungen).
- Liste wird vollständig sortiert, obwohl eigentlich nur das Element mit maximaler Priorität gefunden werden muß.
- Ein- und Ausfüge-Reihenfolge nicht vorhersagbar.

Effiziente Implementierung

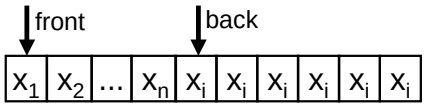
- Warteschlangen: einfache Implementierung (front und back Pointer)
- Binär-Bäume: schneller Elementzugriff
- Array-Implementierung vollständiger Bäume kombiniert beide Vorteile
- Aber: Bei Suchbäumen ist das maximale Element ***unten*** im Baum ...

Heap-Bedingung

- Neue Sortierung der Knoten im Baum
  - Für alle Knoten: T = Node(L,x,R)
  - max_prio(L) ≤ x.prio
  - max_prio(R) ≤ x.prio
- Element mit maximaler Priorität ist im Wurzelknoten gespeichert.
- Sortierung der Elemente nur entlang der Pfade im Binärbaum.

Effiziente Implementierung

- Array-Implementierung
  - Front-Pointer bleibt fest
  - Element mit maximaler Priorität in S[1]
  - Back-Pointer variabel
  - Einfügen in S[back]



Effiziente Implementierung

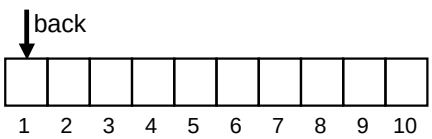
- Enq(x)

```
if (back+1 == Länge)
    Q-FULL-ERROR
else
    i      = back;   back++;
    S[ i ] = x;
    while (i != 1) & (S[ i ] > S[ i / 2 ])
        swap(S[ i ], S[ i / 2 ]);
        i = i / 2;
```


- Deq()

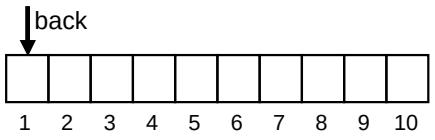
```
if (back == 0)
    Q-EMPTY-ERROR;
else
    i = 1;   back--;   S[1] = S[back];
    while (i <= (back-1) / 2)
        j = 2*i;
        if (j < back-1) & (S[ j ] < S[ j+1 ])
            j++;
        if (S[ i ] < S[ j ])
            swap(S[ i ], S[ j ]);
            i = j;
        else
            i = back;
```


Beispiel

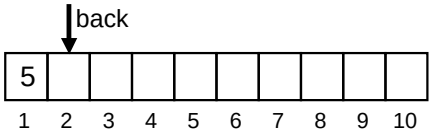


Beispiel

Enq(5)

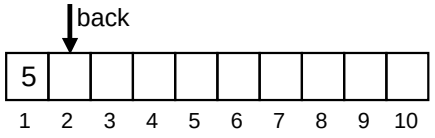


Beispiel

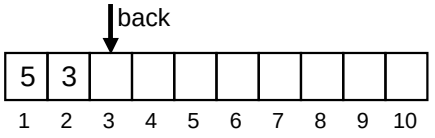


Beispiel

Enq(3)

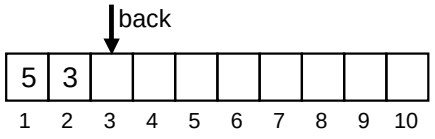


Beispiel

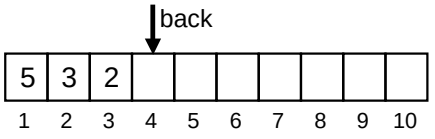


Beispiel

Enq(2)

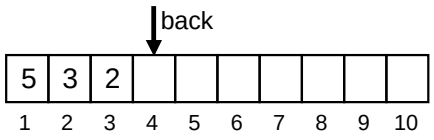


Beispiel

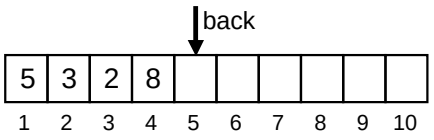


Beispiel

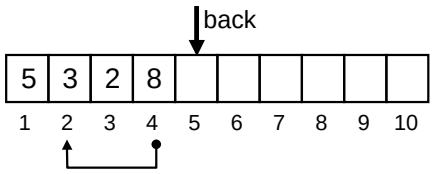
Enq(8)



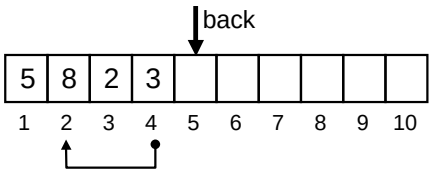
Beispiel



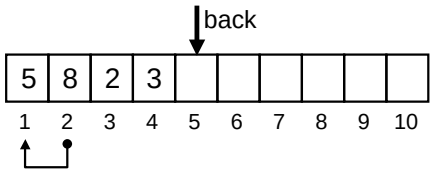
Beispiel



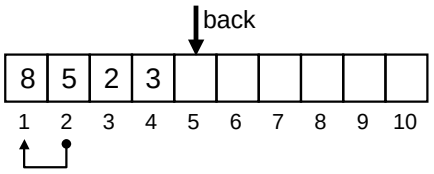
Beispiel



Beispiel

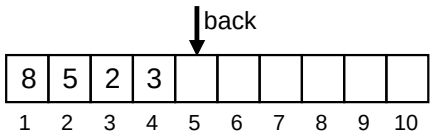


Beispiel

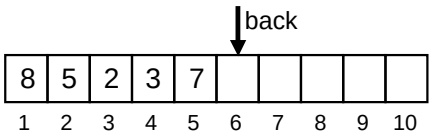


Beispiel

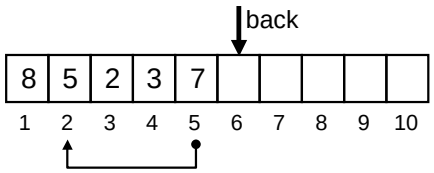
Enq(7)



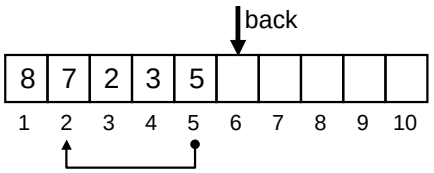
Beispiel



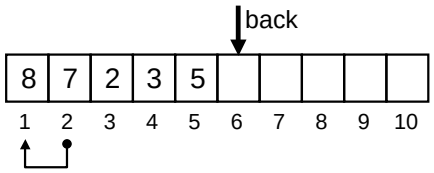
Beispiel



Beispiel

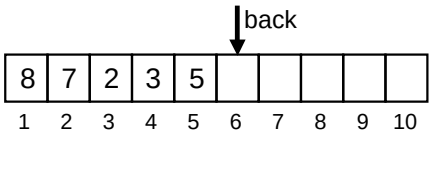


Beispiel

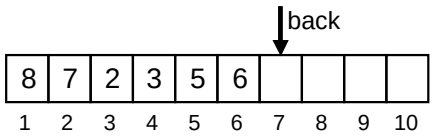


Beispiel

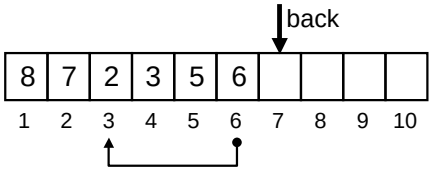
Enq(6)



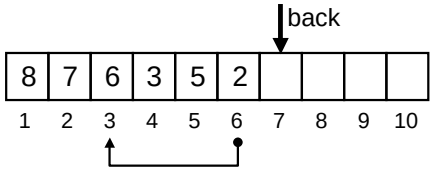
Beispiel



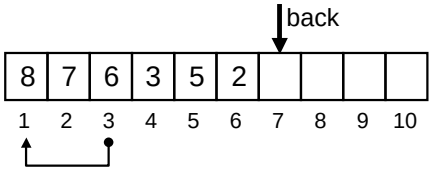
Beispiel



Beispiel

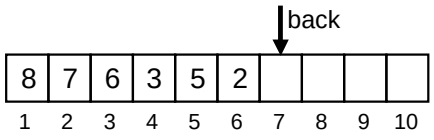


Beispiel

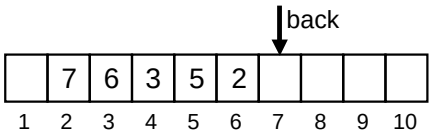


Beispiel

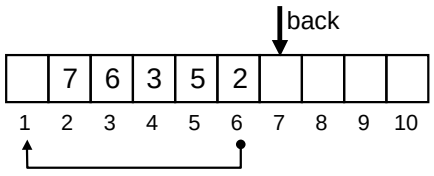
Deq()



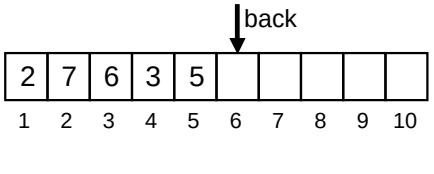
Beispiel



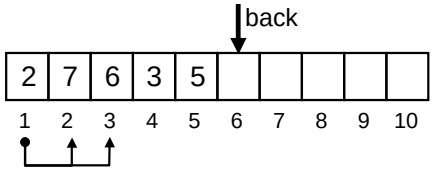
Beispiel



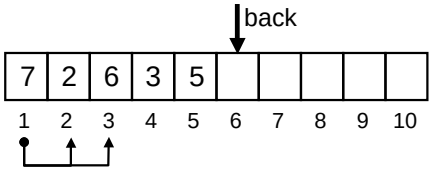
Beispiel



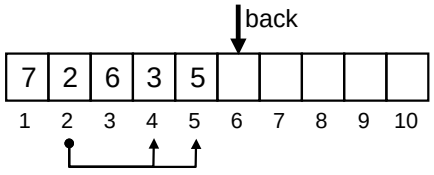
Beispiel



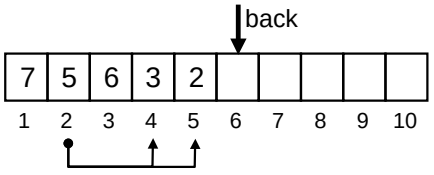
Beispiel



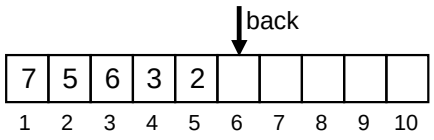
Beispiel



Beispiel



Beispiel



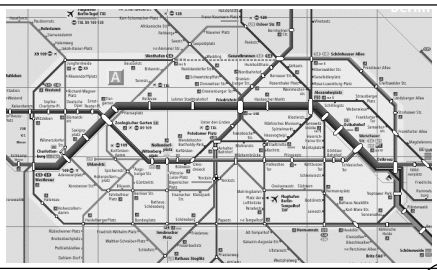
(1.5) Graphen

- Darstellung allgemeiner Beziehungen zwischen Objekten/Elementen
  - Objekte = Knoten: $V = \{v_1, ..., v_n\}$
  - Beziehungen = Kanten: $E = \{(a_1,b_1), ... (a_m,b_m)\}$
- Kanten können gerichtet (Vorgänger/ Nachfolger) oder ungerichtet (Nachbarschaft) definiert werden.

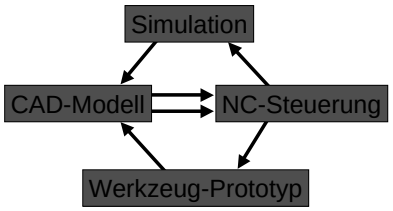
Beispiele

- Flugverbindungen *(ungerichtet)*
- Straßennetz in Aachen *(gerichtet)*
- Abhängigkeiten von Arbeitspaketen in einem Großprojekt *(gerichtet)*
- 3D Polygonmodelle in der Computergraphik *(ungerichtet)*
- Querbezüge in einem Hypertext-Dokument (HTML) *(gerichtet)*
- ...

Beispiele

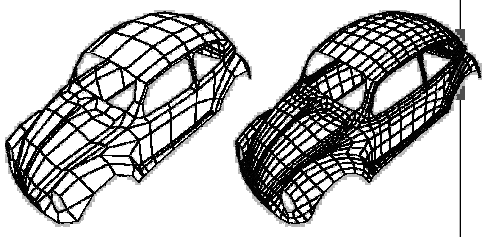


Beispiele



Workflow *Mechatronic*

Beispiele



Begriffe

- Zusammenhängend (stark)
- Vollständig
- Isomorph
- Planare Graphen

Begriffe

- Zusammenhängend (*stark*)
  - Für jedes Paar v_i, v_j von Knoten existiert ein Pfad von Kanten von v_i nach v_j oder (*und*) von v_j nach v_i.
- Vollständig
- Isomorph
- Planare Graphen

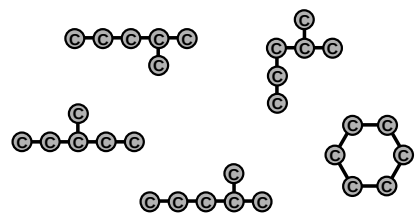
Begriffe

- Zusammenhängend (stark)
- Vollständig
  - Zwischen je zwei Knoten existiert eine Kante
  - m = n*(n-1) / 2
- Isomorph
- Planare Graphen

Begriffe

- Zusammenhängend (stark)
- Vollständig
- Isomorph
  - Graphen (V_1,E_1) und (V_2,E_2) durch Permutation der Knoten(indizes) ineinander überführbar.
  - *Notwendige* Kriterien: $n_1 = n_2$, $m_1 = m_2$, Anzahl der Knoten mit Grad k, ...
- Planare Graphen

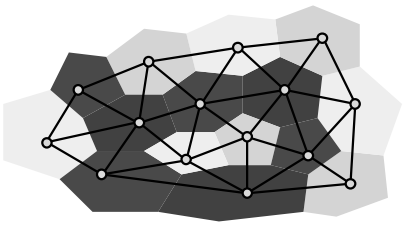
Kohlenwasserstoffe



Begriffe

- Zusammenhängend (stark)
- Vollständig
- Isomorph
- Planare Graphen
  - Zerlegung der Ebene in disjunkte Zellen
  - Kompatibilitätsbedingungen zwischen Kanten

Mobilfunknetze



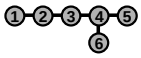
Repräsentationen

- Allgemeine Graphen
  - Adjazenzmatrix
  - Adjazenzliste
- Planare Graphen
- Simplex-Strukturen

Adjazenzmatrix

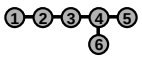
- Knoten: $V = \{v_1, ..., v_n\}$
- $A \in \text{Bool}^{n \times n}$
- $a_{ij} = 1$ falls $(i,j) \in E$, $a_{ij} = 0$ sonst
- j-te Spalte: alle Nachfolger von v_j
- i-te Zeile: alle Vorgänger von v_i
- A *symmetrisch* für *ungerichtete* Graphen

Adjazenzmatrix



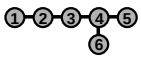
$$A = \begin{pmatrix} 0&1&0&0&0&0 \\ 1&0&1&0&0&0 \\ 0&1&0&1&0&0 \\ 0&0&1&0&1&1 \\ 0&0&0&1&0&0 \\ 0&0&0&1&0&0 \end{pmatrix}$$

Zusammenhangskomponenten



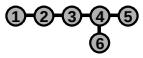
$$B = A + I = \begin{pmatrix} 1&1&0&0&0&0 \\ 1&1&1&0&0&0 \\ 0&1&1&1&0&0 \\ 0&0&1&1&1&1 \\ 0&0&0&1&1&0 \\ 0&0&0&1&0&1 \end{pmatrix}$$

Zusammenhangskomponenten



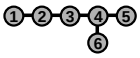
$$B^2 = \begin{pmatrix} 1&1&1&0&0&0 \\ 1&1&1&1&0&0 \\ 1&1&1&1&1&1 \\ 0&1&1&1&1&1 \\ 0&0&1&1&1&1 \\ 0&0&1&1&1&1 \end{pmatrix}$$

Zusammenhangskomponenten



$$B^3 = \begin{pmatrix} 1&1&1&1&0&0 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 0&1&1&1&1&1 \\ 0&1&1&1&1&1 \end{pmatrix}$$

Zusammenhangskomponenten



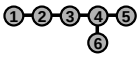
„Durchmesser“

$$B^4 = \begin{pmatrix} 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \\ 1&1&1&1&1&1 \end{pmatrix}$$

Adjazenzlisten

- Bei komplexen Graphen bestehen oft nur lokale Knotenbeziehungen
- Die meisten Einträge der Adjazenzmatrix sind „0“ (dünn besetzte Matrix)
- Speichere nur die existierenden Kanten

Adjazenzlisten



1: 2
2: 1,3
3: 2,4
4: 3,5,6
5: 4
6: 4

Adjazenzlisten

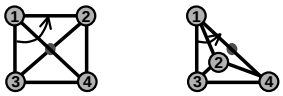
- Für gerichtete Graphen werden Vorgänger- und Nachfolger-Listen verwaltet.
- Speicherverbrauch: (#Nachbarn $\leq$ k)
  - A-Matrix: n^2 bit
  - A-Listen: n * k * b = n * k * log(n)
- Beispiel:
  Regionalbahnen in Aachen und Wladiwostok

Planare Graphen

- ... beschreiben eine Partition der Ebene in disjunkte Zellen.
- Knoten sind durch Kanten verbunden und diese bilden Zellen (Maschen).
- Planar: es ist möglich die Knoten so zu platzieren, dass sich die Kanten nicht kreuzen.

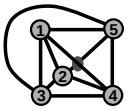
Planare Graphen

- Kompatibilitätsbedingungen
  - eindeutiger Umlaufsinn für die Nachbarn jedes Knotens
  - eindeutiger Umlaufsinn für die Kanten jeder Zelle / Masche



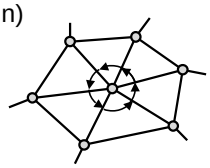
Planare Graphen

- Kompatibilitätsbedingungen
  - eindeutiger Umlaufsinn für die Nachbarn jedes Knotens
  - eindeutiger Umlaufsinn für die Kanten jeder Zelle / Masche



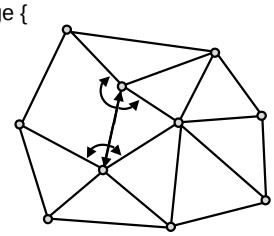
Kantenstrukturen

- Kanten verbinden Knoten
- Kanten verweisen auf benachbarte Kanten (Umlaufsinn)
- Garantiert die Planarität



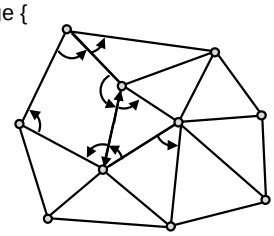
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge N_1,N_2;
  Edge P_1,P_2;
  }



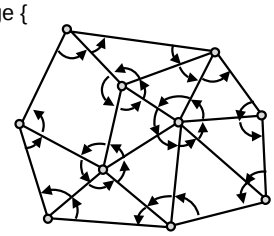
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }



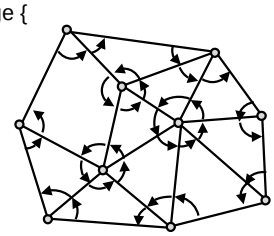
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }



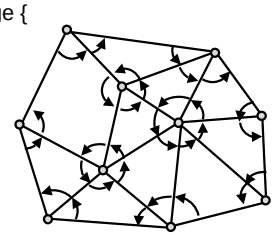
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }

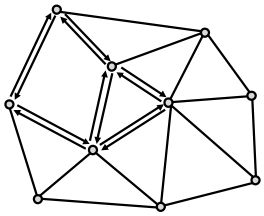


Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }

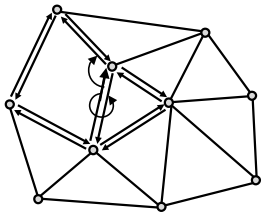


Halb-Kantenstrukturen



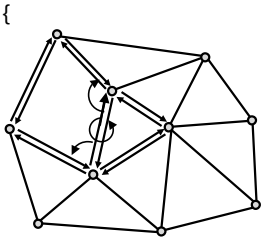
Halb-Kantenstrukturen

- public class HEdge {
  Node A;
  Edge Next;
  Edge Opposite;
  }



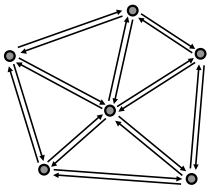
Halb-Kantenstrukturen

- public class HEdge {
  Node A;
  Edge Next;
  Edge Prev;
  Edge Opposite;
  }



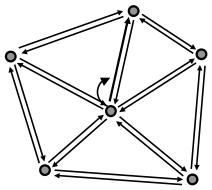
Liste der Nachbarn

1. Start at vertex



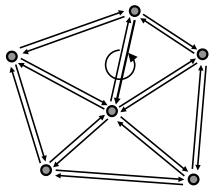
Liste der Nachbarn

1. Start at vertex
2. Outgoing halfedge



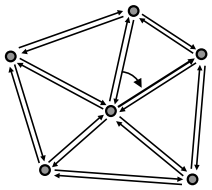
Liste der Nachbarn

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge



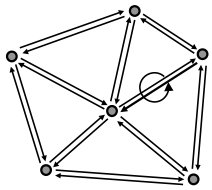
Liste der Nachbarn

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge



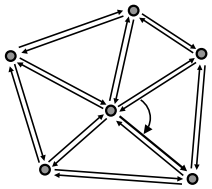
Liste der Nachbarn

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite



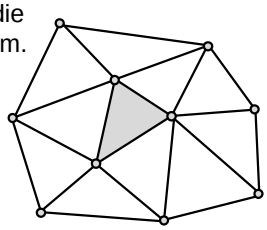
Liste der Nachbarn

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite
6. Next
7. ...



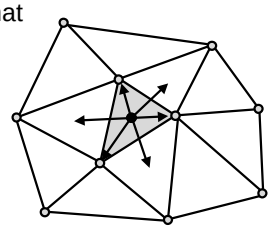
Simplex-Strukturen

- Simplices sind die einfachsten k-dim. Strukturen
  - Punkt
  - Strecke
  - Dreieck
  - Tetraeder
  - ...



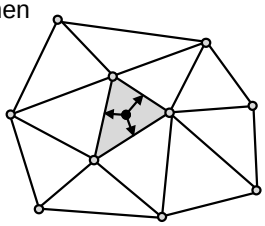
Simplex-Strukturen

- Ein k-Simplex hat
  - k+1 Knoten
  - k+1 Nachbarn



Simplex-Strukturen

- Seine k+1 Flächen sind (k-1)-Simplices



Simplex-Strukturen

- public class k_Simplex {
  Node Vertices[k+1];
  (k-1)_Simplex Faces[k+1];
  }
- Darstellung beliebig dimensionaler Komplexe mit disjunkten Zellen
- Navigation / Aufzählung schwieriger

Einfügen und Löschen

- Bei Suchbäumen bestimmt der Wert des Elementes die Position im Baum (impliziter Marker)
- Einfüge**reihenfolge** bestimmt die Struktur (bessere Algorithmen später)
- Löschen **innerer** Knoten nicht trivial

Einfügen

- Insert(X,Create()) =
  Node(Create(),X,Create())
- Insert(X,Node(L,Y,R)) =
  if (X ≤ Y) Insert(X,L);
  if (X > Y) Insert(X,R);

Löschen

- Remove(X,Node(L,Y,R)) =
  if (X < Y) Remove(X,L);
  if (X > Y) Remove(X,R);
  // X ≠ Y
- Max(Node(L,X,Create()) = X
  Max(Node(L,X,R)) = Max(R)
- Min(Node(Create(),X,R) = X
  Min(Node(L,X,R)) = Min(L)

Löschen

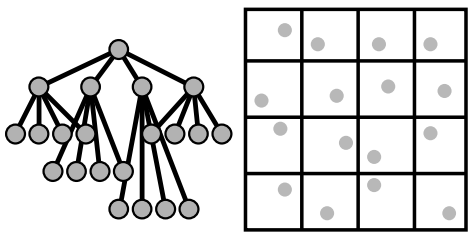
- Remove(X,Node(L,X,R)) =

```
if (L ≠ Create())
    Node(Remove(Max(L),L),Max(L),R)
else if (R ≠ Create())
    Node(L,Min(R),Remove(Min(R),R))
else
    Create();   // L = R = Create()
```


k-dimensionale Suchbäume

- Beispiel: Suche nach Punkten im IR^2 (nächste Tankstelle, ...) oder im IR^k
- Verwende 2^k-näre Bäume
  - Quadtree (IR^2)
  - Octree (IR^3)
  - ...
- kD-Bäume (k-dimensionale Binärbäume)

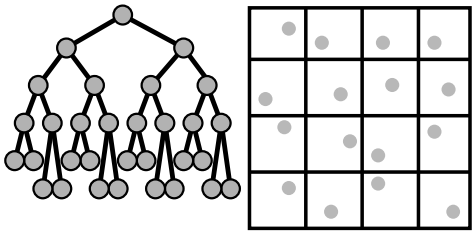
Quadtree



kD-Bäume

- Datentyp: Binärbaum
- Jeder Knoten enthält einen k-dim. Wert
- Sortierung für Knoten T der Tiefe h:
  $max_h(Left(T)) \leq Value_h(T) < min_h(Right(T))$
- Subskript h bedeutet: Verwende die (h mod k)-te Koordinate

kD-Bäume



(1.4) Prioritätsschlangen

- Warteschlangen mit „Vordrängeln“
- Einfügen am Ende der Schlange
- Jedes Element hat eine Priorität
- Deq() liefert immer das Element mit der höchsten Priorität

Prioritätsschlangen

- Elemente: $X \in W' = W \times IR$
  Wertebereich: $L = \{\} \cup W' \cup W'^2 \cup W'^3 \cup \ldots$
- Create : $\rightarrow L$
- Enq* : $W' \times L \rightarrow L$
- Deq : $L \rightarrow L$
- Get : $L \rightarrow W'$
- Empty : $L \rightarrow$ Bool

Prioritätsschlangen

- Enq() : Standardoperator
- Enq*() : füge Element x bzgl. seiner Priorität x.prio ein.
- Enq*(x,Enq(y,z)) = Enq(y,Enq*(x,z)) if x.prio > y.prio

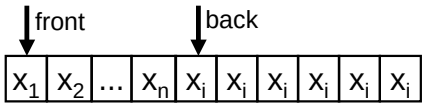
  Enq*(x,Enq(y,z)) = Enq(x,Enq(y,z)) if x.prio ≤ y.prio

Heap-Bedingung

- Neue Sortierung der Knoten im Baum
  - Für alle Knoten: T = Node(L,x,R)
  - max_prio(L) ≤ x.prio
  - max_prio(R) ≤ x.prio
- Element mit maximaler Priorität ist im Wurzelknoten gespeichert.
- Sortierung der Elemente nur entlang der Pfade im Binärbaum.

Effiziente Implementierung

- Array-Implementierung
  - Front-Pointer bleibt fest
  - Element mit maximaler Priorität in S[1]
  - Back-Pointer variabel
  - Einfügen in S[back]



Effiziente Implementierung

- Enq(x)

```
if (back+1 == Länge)
    Q-FULL-ERROR
else
    i      = back;   back++;
    S[ i ] = x;
    while (i != 1) & (S[ i ] > S[ i / 2 ])
        swap(S[ i ], S[ i / 2 ]);
        i = i / 2;
```


- Deq()

```
if (back == 0)
    Q-EMPTY-ERROR;
else
    i = 1;   back--;   S[1] = S[back];
    while (i <= (back-1) / 2)
        j = 2*i;
        if (j < back-1) & (S[ j ] < S[ j+1])
            j++;
        if (S[ i ] < S[ j ])
            swap(S[ i ], S[ j ]);
            i = j;
        else
            i = back;
```


(1.5) Graphen

- Darstellung allgemeiner Beziehungen zwischen Objekten/Elementen
  - Objekte = Knoten: $V = \{v_1, ..., v_n\}$
  - Beziehungen = Kanten: $E = \{(a_1,b_1), ... (a_m,b_m)\}$
- Kanten können gerichtet (Vorgänger/ Nachfolger) oder ungerichtet (Nachbarschaft) definiert werden.

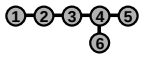
Begriffe

- Zusammenhängend (stark)
- Vollständig
- Isomorph
- Planare Graphen

Repräsentationen

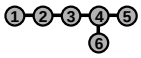
- Allgemeine Graphen
  - Adjazenzmatrix
  - Adjazenzliste
- Planare Graphen
- Simplex-Strukturen

Adjazenzmatrix



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

Adjazenzlisten



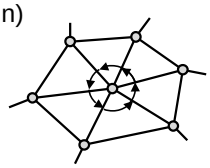
1: 2
2: 1,3
3: 2,4
4: 3,5,6
5: 4
6: 4

Planare Graphen

- ... beschreiben eine Partition der Ebene in disjunkte Zellen.
- Knoten sind durch Kanten verbunden und diese bilden Zellen (Maschen).
- Planar: es ist möglich die Knoten so zu platzieren, dass sich die Kanten nicht kreuzen.

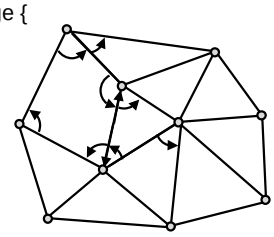
Kantenstrukturen

- Kanten verbinden Knoten
- Kanten verweisen auf benachbarte Kanten (Umlaufsinn)
- Garantiert die Planarität



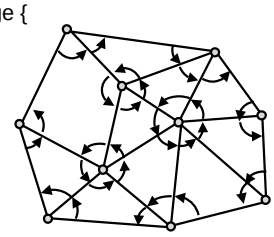
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }



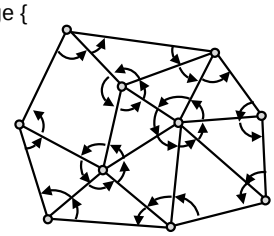
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }



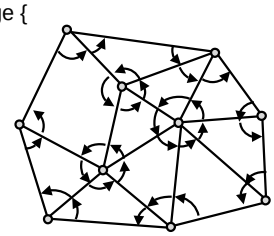
Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }

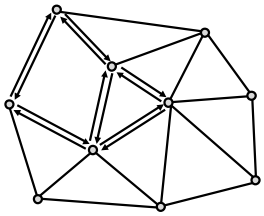


Kantenstrukturen

- public class Edge {
  Node A,B;
  Edge Next;
  Edge Prev;
  }

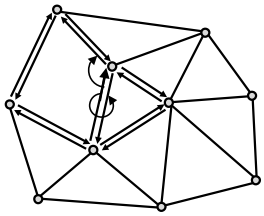


Halb-Kantenstrukturen



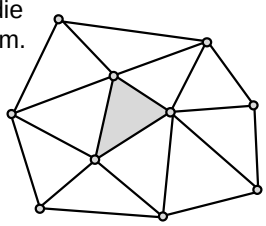
Halb-Kantenstrukturen

- public class HEdge {
  Node A;
  Edge Next;
  Edge Opposite;
  }



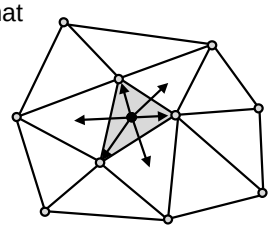
Simplex-Strukturen

- Simplices sind die einfachsten k-dim. Strukturen
  - Punkt
  - Strecke
  - Dreieck
  - Tetraeder
  - ...



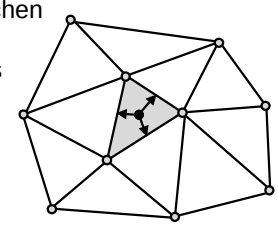
Simplex-Strukturen

- Ein k-Simplex hat
  - k+1 Knoten
  - k+1 Nachbarn



Simplex-Strukturen

- Seine k+1 Flächen sind (k-1)-Simplices



Simplex-Strukturen

- public class k_Simplex {
  Node Vertices[k+1];
  (k-1)_Simplex Faces[k+1];
  }
- Darstellung beliebig dimensionaler Komplexe mit disjunkten Zellen
- Navigation / Aufzählung schwieriger

Summary: Datenstrukturen

- Abstrakte Datentypen (Sorten, Funktionen, Axiome)
- Elementare Typen (Aufzählung, Skalar, Zusammengesetzt)
- Lineare Strukturen (Liste, Stack, Queue)
- Bäume (Suchbäume, kD-Bäume, ...)
- Heap
- Graphen (A-Matrix, A-Listen, Kanten, Simplices)

(2.0) Algorithmen

- (2.1) Analyse von Algorithmen
- (2.2) Entwurfsparadigmen
- (2.3) Sortieren
- (2.4) Suchen
- (2.5) ...

Begriffe

- Effizienz
  = gute Ausnutzung von Resourcen
- Resourcen
  = Rechenzeit, Speicherplatz
- Aufwand / Komplexität
  = tatsächlicher Verbrauch von Resourcen

Performanzfaktoren

- Prozessorleistung
- Hauptspeicher
- Caches
- Compiler-Version
- Betriebssystem
- ...

Beispiel

- Matrix mit 1024 x 1024 Einträgen
- Entry(i,j) = A[i*1024+j]
- Löschen 1:
```
for (i=0 ; i<1024 ; i++)
    for (j=0 ; j<1024 ; j++)
        Entry(i,j) = 0;
```
- Löschen 2:
```
for (j=0 ; j<1024 ; j++)
    for (i=0 ; i<1024 ; i++)
        Entry(i,j) = 0;
```


Effizienz eines Algorithmus

- Implement / Run / Test
  - Abhängig vom speziellen System
  - Abhängig von der aktuellen Auslastung
  - Abhängig von den Testdaten
  - Abhängig von der Begabung des Programmierers
  - Gibt es noch bessere Algorithmen?

Effizienz eines Algorithmus

- ***Tatsächliche*** Laufzeit variiert auf unterschiedlichen Computer-Systemen
- ***Erwartete*** Laufzeit verschiedener Algorithmen auf demselben Computer kann nur für ***lange*** Berechnungen verglichen werden (variierende Systemauslastung).

Effizienz eines Algorithmus

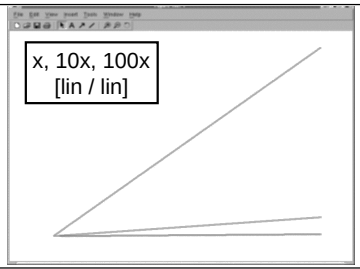
- In der Regel interessiert man sich nicht für die exakte Anzahl von Operationen, sondern nur für die **Komplexitätsklasse**.

  Beispiel: lineare Liste vs. Suchbaum :
  - 1000 vs. 10
  - 2000 vs. 11
  - ...
  - 1000000 vs. 20
  - 2000000 vs. 21
  - ...

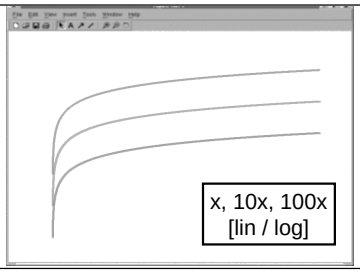
Komplexitätsklassen

- Performance Charts
  - X-Achse : Größe der Eingabedaten
  - Y-Achse : Rechenzeit
- Lineare vs. logarithmische Achsen
  - Exakte Werte
  - Größenordnungen

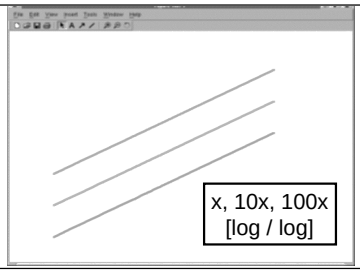
Komplexitätsklassen



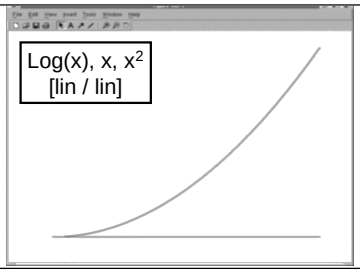
Komplexitätsklassen



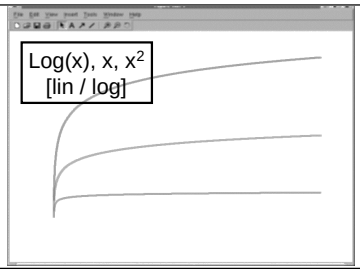
Komplexitätsklassen



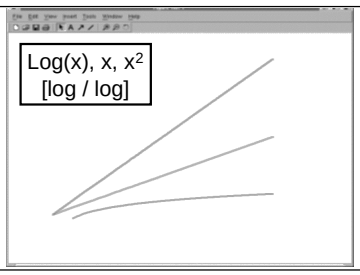
Komplexitätsklassen



Komplexitätsklassen



Komplexitätsklassen



Abstrakte Analyse

- Abstrakte Analyse
  Zähle die ***wesentlichen*** Operationen auf einem ***idealen*** Computer.
- Die Komplexität eines Algorithmus ist **unabhängig** von der Programmiersprache.

Stripped Java

- Teilmenge von Java
  - Prozedur-Aufrufe
  - Integer-Arithmetik
  - Zuweisungen (Lesen, Schreiben)
  - if-then-else
  - while
- Vollständig aber einfacher zu analysieren als *full* Java

Berechnungsaufwand

- Prozedur-Aufruf
  - Speichere Kontext auf dem Stack
  - Speichere Rücksprungadresse
  - Kopiere aktuelle Parameter
  - Generiere neuen Prozedurkontext
  - ...
  - Kopiere Rückgabewert
  - Stelle alten Kontext wieder her
- Kosten: $C_{PC} = C_{PC}(S_{old}, Args, S_{new})$

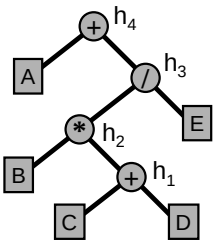
Berechnungsaufwand

- Integer-Arithmetik
  (Einheitskosten der Elementaroperationen)
  $$C_+ \leq C_- \leq C_* \leq C_/$$
- Konvertiere komplizierte Terme in die 3-Adress-Form
- **Achtung:** bei genauer Analyse muß die *Anzahl der Stellen* berücksichtigt werden

3-Adress-Form

A + B * (C + D) / E

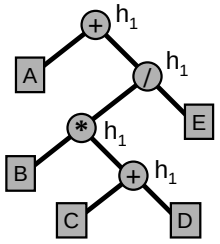
$h_1 = C + D$
$h_2 = B * h_1$
$h_3 = h_2 / E$
$h_4 = A + h_3$



3-Adress-Form

A + B * (C + D) / E

$h_1 = C + D$
$h_1 = B * h_1$
$h_1 = h_1 / E$
$h_1 = A + h_1$



Kostenmaße

- $C_+ \leq C_- \leq C_* \leq C_/$
- Addition
  - Addition einzelner Ziffern: x + y = z + c
  - Berücksichtigung des Überlaufs
  - # Sub-Operationen = # Stellen ≈ log(z)
  - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leq C_- \leq C_* \leq C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: x * y = z
  - $(x_1 + 10 * x_2) * (y_1 + 10 * y_2) =$
    $x_1 * y_1 + 10 * (x_2 * y_1 + x_1 * y_2) + 100 * x_2 * y_2$
  - # Sub-Operationen = # Stellen^2 $\approx$ log(z)^2
  - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leq C_- \leq C_* \leq C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: x * y = z
  - $(x_1 + b * x_2) * (y_1 + b * y_2) =$
    $x_1 * y_1 + b * (x_2 * y_1 + x_1 * y_2) + b^2 * x_2 * y_2$
  - # Sub-Operationen = # Stellen^2 $\approx$ log(z)^2
  - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leq C_- \leq C_* \leq C_/$
- Multiplikation
  - Multiplikation einzelner Ziffern: x * y = z
  - $(x_1 + b * x_2) * (y_1 + b * y_2) =$
    $x_1 * y_1 + b * ((x_1+x_2)*(y_1+y_2) - x_1 * y_1 - x_2 * y_2) + b_2 * x_2 * y_2$
  - # Sub-Operationen = # Stellen$^{1.58}$ $\approx$ log(z)$^{\log(3)}$
  - Java-Integer : # Stellen = 32

Kostenmaße

- $C_+ \leq C_- \leq C_* \leq C_/$
- Multiplikation
  - *Karatsuba*-Multiplikation
  - Spart eine Sub-Multiplikation aber
    benötigt drei zusätzliche Additionen
  - Lohnt sich nur für *sehr große* Zahlen !!!
  - Es gibt theoretisch noch bessere Algorithmen

Abstrakte Analyse

- ***Theorie:*** Effiziente Algorithmen lohnen sich vor allem für große Probleme
- ***Praxis:*** Für kleinere Probleme können einfache Algorithmen unter Umständen sinnvoller sein.
- Die Klassifizierung groß/klein hängt vom Problem und vom Algorithmus ab.

Berechnungsaufwand

- Zuweisung (Lesen, Schreiben)
  - Einheitskosten C_{ass}
  - Caching-Effekte werden vernachlässigt
- Array-Zugriff A[i] enthält eine *versteckte* Addition.

Berechnungsaufwand

- If (X) then (Y) else (Z)
  - Worst Case:
    #OP = #OP(X) + max{ #OP(Y), #OP(Z) }
  - Average Case:
    P = Probability(X = true)
    #OP = #OP(X) + P * #OP(Y) + (1-P) * #OP(Z)

Berechnungsaufwand

- While (X) { (Y) }
  - Konstanter Aufwand:
    #OP = #Loops * [#OP(X) + #OP(Y)]
  - Variabler Aufwand:
    #OP = #Loops * #OP(X) + SUM(#OP(Y_i))

Beispiel

- Sortiere(A,n)

```
i = 0;
while (i < n)
    min = i;  j = i+1;
    while (j < n)
        if (A[j] < A[min])  min = j;
        j++:
    swap(A[i],A[min]);
    i++;
```


Beispiel

- Aufwand der inneren Schleife C_{inner}
- Aufwand der äußeren Schleife C_{outer}
- Anzahl der Durchläufe

$$C(n) = \Sigma_i C_{outer} + (n-i) * C_{inner}$$

$$= n\, C_{outer} + n\,(n-1) / 2 * C_{inner}$$

$$\approx n^2 * C_{inner} / 2 \quad \text{... für „große“ n}$$

Best / Worst / Average Case

- Verschieden optimistische Abschätzungen
  - Untere Schranke
  - Erwartungswert
  - Obere Schranke
- ***Erwartungswert:*** Mittelwert des Aufwandes über alle möglichen Eingaben gewichtet mit der Auftrittswahrscheinlichkeit.

Beispiel

- Multipliziere zwei n-bit Zahlen in Binärdarstellung
- Multiply(x,y)

```
i = 0; result = 0;
while (i < n)
    if (bit(x,i) == 1)
        result = result + shift(y,i);
    i++;
```

Beispiel

- Aufwand ≈ Anzahl der Additionen
  = Anzahl der Einsen in der Binärdarstellung von x
- Best Case : x = 0 ... 0 Additionen
- Worst Case : x = 2^n-1 ... n Additionen
- Average Case ???

Beispiel

- Anzahl der möglichen Eingaben
  $N = 2^n$
- Anzahl der Eingaben mit k Einsern in x
  $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$
- Mittlerer Aufwand ...

Beispiel

- Mittlerer Aufwand

$$k\binom{n}{k} = \frac{k\ n!}{k!\,(n-k)!} = \frac{n\ (n-1)!}{(k-1)!\,(n-1-k+1)!}$$

$$= n\binom{n-1}{k-1}$$

$$\sum_k k\binom{n}{k} = \sum_k n\binom{n-1}{k-1} = n\,2^{n-1}$$

Beispiel

- Mittlerer Aufwand
  $S / N = n\,2^{n-1} / 2^n = n / 2$ Additionen
- Bemerkung zur ***Kombinatorik***
  - Permutationen n!
  - Kombinationen $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$

Asymptotische Komplexität

- In der Regel ist das **quantitative** Zählen der Elementaroperationen mühsam und wenig ergiebig.
- Man ist daher eher an **qualitativen** Aussagen interessiert, z.B. wie verändert sich der Rechenaufwand, wenn man die Eingabedaten verdoppelt?

Beispiele

- Addition
  - Verdopplung der Stellen
  - Verdopplung der Sub-Operationen
- Multiplikation / Sortierbeispiel
  - Verdopplung der Stellen / Elemente
  - Vervierfachung der Sub-Operationen
- Suchen im Suchbaum:
  - Verdopplung der Knoten im Baum
  - Erhöhung des Suchaufwandes um einen Test (vollständiger Baum erhöht sich um ein Level)

Asymptotische Komplexität

- Definition von Schranken für Funktionen f,g : IN $\rightarrow$ IR$^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Asymptotische Komplexität

- Definition von Schranken für Funktionen f,g : IN $\rightarrow$ IR$^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Asymptotische Komplexität

- Definition von Schranken für Funktionen f,g : IN $\rightarrow$ IR$^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rechenregeln

- $f + g \in O(\max\{f,g\}) = \begin{cases} O(f) & \text{falls } g \in O(f) \\ O(g) & \text{falls } f \in O(g) \end{cases}$
- $f * g \in O(f * g)$
- $g(x) := a\, f(x) + b \wedge f \in \Omega(1) \Rightarrow g \in O(f)$
- Schreibweise
  $n\, C_{outer} + n\,(n-1) / 2 * C_{inner} = O(n^2)$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fibonacci-Zahlen

- $F(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F(n-1)+F(n-2) & \text{für } n > 1 \end{cases}$
- F(n) positiv für alle n
- F(n) wächst für n > 2 streng monoton

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fibonacci-Zahlen

- $F(n) = F(n-1) + F(n-2)$
  $< 2 * F(n-1)$
  $< 4 * F(n-2)$
  $< \ldots$
  $< 2^{n-1} * F(1) = 2^n / 2$
- $F \in O(2^n),\ c = 1/2$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fibonacci-Zahlen

- $F(n) = F(n-1) + F(n-2)$
  $> 2 * F(n-2)$
  $> 4 * F(n-4)$
  $> ...$
  $> \begin{cases} 2^{(n-1)/2} * F(1) & n \text{ ungerade} \\ 2^{n/2-1} * F(2) & n \text{ gerade} \end{cases}$
- $F \in \Omega(2^{n/2})$, $c = 1 / \sqrt{2}$

Alternative Definition

- „Wachstumsrate“ $(f,g : IN \rightarrow IR^+)$
- $\lim_{x \rightarrow \infty} g(x) / f(x) = \begin{cases} 0 & \text{... g wächst langsamer} \\ c & \text{... g und f wachsen gleich} \\ \infty & \text{... g wächst schneller} \end{cases}$
- g wächst langsamer: $g \in O(f) \setminus \Theta(f)$
- g und f wachsen gleich: $g \in \Theta(f)$
- g wächst schneller: $g \notin O(f)$

Typische Komplexitätsklassen

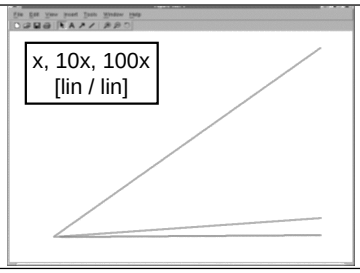
• $O(1)$	Elementaroperation
• $O(\log(n))$	Binäre Suche
• $O(n)$	Lineare Suche
• $O(n \log(n))$	Sortieren (später)
• $O(n^2)$	Sortieren (vorhin)
• $O(n^3)$	Invertieren von Matrizen
• $O(2^n)$	Labyrinth-Suche (vollständig)
• $O(n!)$	Zahl von Permutationen

Effizienz eines Algorithmus

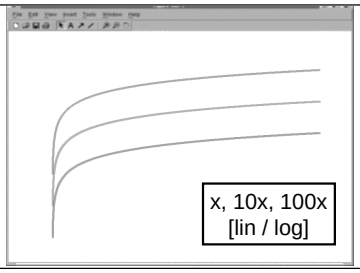
- In der Regel interessiert man sich nicht für die exakte Anzahl von Operationen, sondern nur für die **Komplexitätsklasse**.

  Beispiel: lineare Liste vs. Suchbaum :
  - 1000 vs. 10
  - 2000 vs. 11
  - ...
  - 1000000 vs. 20
  - 2000000 vs. 21
  - ...

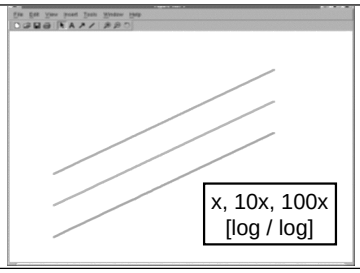
Komplexitätsklassen



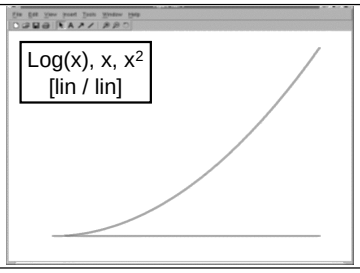
Komplexitätsklassen



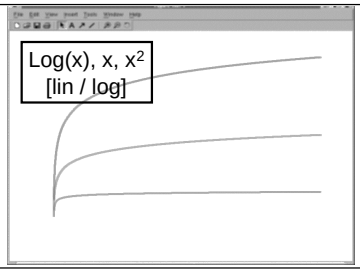
Komplexitätsklassen



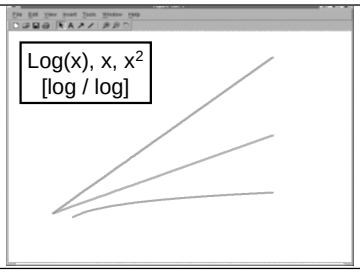
Komplexitätsklassen



Komplexitätsklassen



Komplexitätsklassen



Stripped Java

- Teilmenge von Java
  - Prozedur-Aufrufe
  - Integer-Arithmetik
  - Zuweisungen (Lesen, Schreiben)
  - if-then-else
  - while
- Vollständig aber einfacher zu analysieren als *full* Java

Best / Worst / Average Case

- Verschieden optimistische Abschätzungen
  - Untere Schranke
  - Erwartungswert
  - Obere Schranke
- ***Erwartungswert:*** Mittelwert des Aufwandes über alle möglichen Eingaben gewichtet mit der Auftrittswahrscheinlichkeit.

Asymptotische Komplexität

- In der Regel ist das **quantitative** Zählen der Elementaroperationen mühsam und wenig ergiebig.
- Man ist daher eher an **qualitativen** Aussagen interessiert, z.B. wie verändert sich der Rechenaufwand, wenn man die Eingabedaten verdoppelt?

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : \mathbb{N} \rightarrow \mathbb{R}^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Asymptotische Komplexität

- Definition von Schranken für Funktionen $f,g : \mathbb{N} \rightarrow \mathbb{R}^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$
    $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Asymptotische Komplexität

- Definition von Schranken für Funktionen f,g : IN $\rightarrow$ IR$^+$
  - $O(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \leq c\, f(x) \}$
  - $\Omega(f) = \{ g \mid \exists c>0, \exists n>0, \forall x \geq n : g(x) \geq c\, f(x) \}$
  - $\Theta(f) = O(f) \cap \Omega(f) =$ $\{ g \mid \exists c>0, \exists n>0, \forall x \geq n : f(x)/c \leq g(x) \leq c\, f(x) \}$

Alternative Definition

- „Wachstumsrate“ (f,g : IN $\rightarrow$ IR$^+$)
- $\lim_{x\rightarrow\infty} g(x) / f(x) = \begin{cases} 0 & \text{... g wächst langsamer} \\ c & \text{... g und f wachsen gleich} \\ \infty & \text{... g wächst schneller} \end{cases}$
- g wächst langsamer: $g \in O(f) \setminus \Theta(f)$
- g und f wachsen gleich: $g \in \Theta(f)$
- g wächst schneller: $g \notin O(f)$

Typische Komplexitätsklassen

• O(1)	Elementaroperation
• O(log(n))	Binäre Suche
• O(n)	Lineare Suche
• O(n log(n))	Sortieren (später)
• $O(n^2)$	Sortieren (vorhin)
• $O(n^3)$	Invertieren von Matrizen
• $O(2^n)$	Labyrinth-Suche (vollständig)
• O(n!)	Zahl von Permutationen

Typische Komplexitätsklassen

• O(1)	Elementaroperation
• O(log(n))	Binäre Suche
• O(n)	Lineare Suche
• O(n log(n))	Sortieren (später)
• $O(n^2)$	Sortieren (vorhin)
• $O(n^3)$	Invertieren von Matrizen
• $O(2^n)$	Labyrinth-Suche (vollständig)
• O(n!)	Zahl von Permutationen

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fibonacci-Zahlen

- $F(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F(n-1)+F(n-2) & \text{für } n > 1 \end{cases}$
- $F(n) \in O(2^n)$
- $F(n) \in \Omega(2^{n/2}) = \Omega(\sqrt{2}^n)$

Fibonacci rekursiv

```
F(n)
    if n > 1
        return F(n-1) + F(n-2);
    else
        return n;
```

- T(0) = T(1) = t
- T(n+2) = t + T(n+1) + T(n)

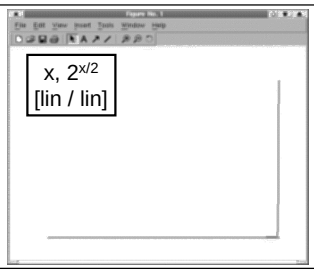
Fibonacci rekursiv

- Behauptung: $T(n) \geq t\, F(n+1)$
- Beweis: Vollständige Induktion ...
  - Anfang: $T(0) = t \geq t\, F(1) = t$
    $T(1) = t \geq t\, F(2) = t$
  - Schritt: $T(n+2) = t + T(n+1) + T(n)$
    $\geq t + t\, F(n+2) + t\, F(n+1)$
    $\geq t + t\, F(n+3)$
    $\geq t\, F(n+3)$
- $T(n) \in \Omega(2^{n/2})$

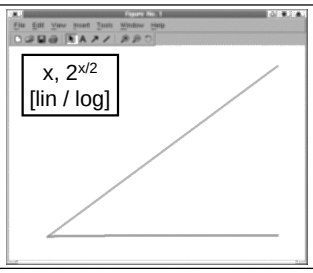
Fibonacci iterativ

- F(n)
  $f_{old} = 1$; $f_{new} = 0$;
  while (n > 0)
  $f_{new} = f_{new} + f_{old}$;
  $f_{old} = f_{new} - f_{old}$;
  n--;
  return f_{new};
- $T(n) = t * n = \Theta(n)$

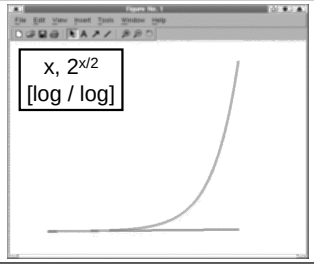
Fibonacci Vergleich



Fibonacci Vergleich



Fibonacci Vergleich



Standardabschätzungen

- Reihensummen
  - Geschachtelte Schleifen
  - Abschätzung durch Integrale
  - Vollständige Induktion
- Rekursionsgleichungen
  - Eliminierung
  - Master-Theorem
  - Direkter Ansatz

Reihensummen

- Geschachtelte Schleifen
  - for (i=0 ; i<n ; i++) O(1) = O(n)
  - for (i=0 ; i<n ; i++) O(i) = O(n^2)

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O\left(\frac{n^2-n}{2}\right) = O\left(n^2-n\right) = O(n^2)$$

Reihensummen

- Geschachtelte Schleifen
  - for (i=0 ; i<n ; i++) O(1) = O(n)
  - for (i=0 ; i<n ; i++) O(i) = O(n^2)
  - for (i=0 ; i<n ; i++)
    for (j=0 ; j<n ; j++) O(1) = O(n^2)
  - for (i=0 ; i<n ; i++)
    for (j=0 ; j<i ; j++) O(1) = O(n^2)
  - ...

Reihensummen

- Geschachtelte Schleifen
  - Generell: der Aufwand $T_k(n)$ einer k-fach geschachtelten Schleife (mit linear beschränkten Laufintervallen) besitzt als obere Schranke ein Polynom vom Grad k

    $T_k(n) = O(\sum_{i=0}^{k} \alpha_i n^i) = O(n^k)$

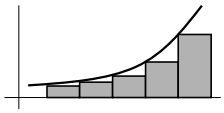
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals

$$\int_0^n f(x)dx \approx \sum_{i=0}^{n-1} f(i+h) \qquad h \in [0,1)$$

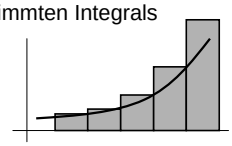
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



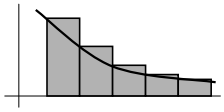
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



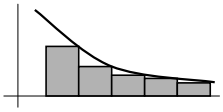
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Abschätzung durch Integral

- Beispiel ...

$$\sum_{i=0}^{n-1} i^2 \le \int_0^n x^2\,dx = \frac{x^3}{3}\Big|_0^n = \tfrac{1}{3}n^3 = O(n^3)$$

$$\sum_{i=0}^{n-1} i^2 = \sum_{i=1}^{n-1} i^2 \ge \int_0^{n-1} x^2\,dx = \frac{x^3}{3}\Big|_0^{n-1} = \tfrac{1}{3}\left(n^3 - 3n^2 + 3n - 1\right) = \Theta(n^3)$$

$$\sum_{i=0}^{n-1} i^2 = \Theta(n^3)$$

Vollständige Induktion

- Behauptung
- Zeige: Behauptung gilt für $n=n_0$
- Zeige: Wenn die Behauptung für ein n gilt, dann gilt sie auch für n+1

Vollständige Induktion

- Behauptung: $\sum_{i=0}^{n-1} i^2 = \tfrac{1}{6}\left(2n^3 - 3n^2 + n\right)$
- Beweis:

$n=1$: $\quad \sum_{i=0}^{0} i^2 = \tfrac{1}{6}(2-3+1) = 0$

Vollständige Induktion

- Behauptung: $\sum_{i=0}^{n-1} i^2 = \tfrac{1}{6}\left(2n^3 - 3n^2 + n\right)$
- Beweis: $n-1 \to n$:

$$\sum_{i=0}^{n} i^2 = n^2 + \sum_{i=0}^{n-1} i^2 = n^2 + \tfrac{1}{6}\left(2n^3 - 3n^2 + n\right) = \tfrac{1}{6}\left(2n^3 + 3n^2 + n\right)$$

$$\tfrac{1}{6}\left(2(n+1)^3 + 3(n+1)^2 + n + 1\right)$$

$$= \tfrac{1}{6}\left(2n^3 + 6n^2 + 6n + 2 - 3n^2 - 6n - 3 + n + 1\right)$$

$$= \tfrac{1}{6}\left(2n^3 + 3n^2 + n\right)$$

Rekursionsgleichungen

- Elimination
- Master-Theorem
- Direkter Ansatz

Rekursionsgleichungen

- F(n) = a + b F(n-1)
- F(n) = a + b F(n-1) + c F(n-2)
- ...
- F(n) = a + b F(n/c)
- G(m) := F(c^m)
  $\rightarrow$ G(m) = a + b G(m-1)

Rekursionsgleichungen

- F(n) = a(n) + b(n) F(n-1)
- F(n) = a(n) + b(n) F(n-1) + c(n) F(n-2)
- ...
- F(n) = a(n) + b(n) F(n/c)
- G(m) := F(c^m)
  $\rightarrow$ G(m) = a(n) + b(n) G(m-1)

Elimination

einfaches Sortieren

F(n) = n + F(n-1)
= n + (n-1) + F(n-2)
= ...
= n + (n-1) + ... + 1 + F(0)
= n (n-1) / 2 + F(0)
= O(n^2)

Elimination

Fibonacci

F(n) = F(n-1) + F(n-2)
= 2 F(n-2) + F(n-3)
= 3 F(n-3) + 2 F(n-4)
= 5 F(n-4) + 3 F(n-5)
= ...
= ???

Abschätzung nach oben

- Typisch: T(n) = a T(n/2) + b
- Einfach, wenn n = 2^k
- Sonst: finde m = 2^k mit m/2 < n $\leq$ m
- Dann gilt: T(n) $\leq$ T(m)

Elimination

$T(n) = T(n/2) + 1$
$= T(n/4) + 1 + 1$
$= T(n/8) + 1 + 1 + 1$
$= ...$
$= T(1) + \log(n)$
$= O(\log(n))$

Binäre Suche

Elimination

$T(n) = 4\,T(n/2)$
$= 16\,T(n/4)$
$= 64\,T(n/8)$
$= ...$
$= 4^{\log_2(n)}\,T(1)$
$= n^2\,T(1)$
$= O(n^2)$

Multiplikation großer Zahlen

Elimination

$T(n) = 4\,T(n/2) + 3n$
$= 16\,T(n/4) + 12\,n/2 + 3n$
$= 64\,T(n/8) + 48\,n/4 + 12\,n/2 + 3n$
$= ...$
$= 3n\,(1 + 2 + 4 + ... + n)$
$= 3n\,(2^{\log(n)+1}-1)$
$= 3n\,(2n-1) = O(n^2)$

Multiplikation großer Zahlen

Elimination

$T(n) = 2\,T(n/2) + n$
$= 4\,T(n/4) + 2\,n/2 + n$
$= 8\,T(n/8) + 4\,n/4 + 2\,n/2 + n$
$= ...$
$= 2^{\log(n)-1}\,n / 2^{\log(n)-1} + ... + 2\,n/2 + n$
$= n \log(n)$

effizientes Sortieren

Master-Theorem

- $T(n) = \begin{cases} c & n = 1 \\ a\,T(n/b) + c\,n & n > 1 \end{cases}$

- $T(n) = \begin{cases} O(n) & a < b \\ O(n \log(n)) & a = b \\ O(n^{\log_b(a)}) & a > b \end{cases}$

Master-Theorem

... sei $n = b^k$:

$T(n) = a\,T(n/b) + c\,n$
$= a^2\,T(n/b^2) + a\,c\,n/b + c\,n$
$= a^3\,T(n/b^3) + a^2\,c\,n/b^2 + a\,c\,n/b + c\,n$
$= ...$
$= cn \sum_{i=0}^{k} (a/b)^i$

Master-Theorem

- T(n) = $cn\sum_{i=0}^{k}(a/b)^i$
- a < b: T(n) ≤ $cn\frac{1}{1-a/b} = O(n)$
- a = b: T(n) = $cn\cdot(k+1) = O(n\cdot k) = O(n\log n)$
- a > b: T(n) = $cn\frac{(a/b)^{k+1}-1}{a/b-1}$

Master-Theorem

- T(n) = $cn\sum_{i=0}^{k}(a/b)^i$
- a < b: T(n) ≤ $cn\frac{1}{1-a/b} = O(n)$
- a = b: T(n) = $cn\cdot(k+1) = O(n\cdot k) = O(n\log n)$
- a > b: T(n) = $cn\frac{(a/b)^{k+1}-1}{a/b-1}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Master-Theorem

- T(n) = $cn\sum_{i=0}^{k}(a/b)^i$
- a < b: T(n) ≤ $cn\frac{1}{1-a/b} = O(n)$
- a = b: T(n) = $cn\cdot(k+1) = O(n\cdot k) = O(n\log n)$
- a > b: T(n) = $cn\frac{(a/b)^{k+1}-1}{a/b-1}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Master-Theorem

$$\frac{(a/b)^{k+1}-1}{a/b-1} = O((a/b)^k) = O((a/b)^{\log_b(n)}) = O(a^{\log_b(n)}/n)$$

$$cn\frac{(a/b)^{k+1}-1}{a/b-1} = O(a^{\log_b(n)}) = O(n^{\log_b(a)})$$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Master-Theorem

- T(n) = $\begin{cases} c & n = 1 \\ a\,T(n/b) + c\,n & n > 1 \end{cases}$
- T(n) = $\begin{cases} O(n) & a < b \\ O(n\log(n)) & a = b \\ O(n^{\log_b(a)}) & a > b \end{cases}$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Multiplikation großer Zahlen

- Standard: T(n) = 4 T(n/2) + 3n
  = $O(n^{\log_2(4)})$
  = $O(n^2)$
- Karatsuba: T(n) = 3 T(n/2) + 6n
  = $O(n^{\log_2(3)})$
  = $O(n^{1.58})$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Direkter Ansatz

- F(n) = F(n-1) + F(n-2)
- Ansatz : $F(n) = \phi^n$
- $\phi^n = \phi^{n-1} + \phi^{n-2} \quad \Leftrightarrow \quad \phi^2 = \phi + 1$
  $\Leftrightarrow \quad \phi^2 - \phi - 1 = 0$
  $\Leftrightarrow \quad \phi_{\pm} = (1 \pm\sqrt{5}) / 2$

Direkter Ansatz

- $\phi_{\pm} = (1 \pm\sqrt{5}) / 2$
- $F(n) := \alpha\, \phi_+^n + \beta\, \phi_-^n$
- $F(0) = \alpha + \beta = 0 \Rightarrow \alpha = -\beta$
- $F(1) = \alpha\, \phi_+ + \beta\, \phi_- = \alpha\, (\phi_+ - \phi_-) = 1$
  $\Rightarrow \alpha = 1/\sqrt{5}$
- $F(n) = (\phi_+^n - \phi_-^n) / \sqrt{5}$

Standardabschätzungen

- Reihensummen
  - Geschachtelte Schleifen
  - Abschätzung durch Integrale
  - Vollständige Induktion
- Rekursionsgleichungen
  - Eliminierung
  - Master-Theorem
  - Direkter Ansatz

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Top-Down

- Zerlege das gegebene Problem in Teilschritte
- Zerlege Teilschritte in Sub-Teilschritte
- Zerlege Sub-Teilschritte in Sub-Sub-Teilschritte
- usw.

Top-Down

- Bei der Implementierung eines Teilschrittes werden die Sub-Teilschritte als Black-Boxes verwendet (Spezifikation!)
- Betrachte die Teilschritte in der Reihenfolge der geringsten Querbezüge
- Vermeide Seiteneffekte

Bottom-up

- Löse Teilprobleme und kombiniere diese zu einer Gesamtlösung
- Möglich selbst bei unvollständiger Spezifikation
- Bessere Test-Suites parallel zur Entwicklung (Entwurf & Implementierung)
- Code reuse !?

Divide & Conquer

- Geg.: Problem der Größe n
- Divide: zerlege das Problem rekursiv in k Teilprobleme der Größe n/k
- Conquer: löse kleine Probleme direkt
- Kombiniere die Teillösungen zur Gesamtlösung

Divide & Conquer

- Meistens: k = 2
- Rekursive Implementierung
- Intuitiv leicht verständlich (eigentlicher Algorithmus in den Prozeduraufrufen versteckt)
- Aufwand durch Rekursionsgleichungen beschrieben (Master-Theorem)

Find MinMax iterativ

- MinMax(A,1,n)
  min=1; max=1; i=2;
  while(i ≤ n)
  if (A[i] < A[min]) min = i;
  if (A[i] > A[max]) max = i;
  i++;
  return (A[min],A[max]);
- T(n) = 2n-2 ... (= 3n-3)

Find MinMax D&C

- MinMax(A,a,b)
  if (a+1 ≥ b)
  return(Min(A[a],A[b]),Max(A[a],A[b]));
  else
  (u_1,v_1) = MinMax(A,a,$\lfloor (a+b)/2 \rfloor$);
  (u_2,v_2) = MinMax(A, $\lfloor (a+b)/2 \rfloor$+1,b);
  return(Min(u_1,u_2),Max(v_1,v_2));

Find MinMax D&C

- $T(n) = \begin{cases} 2 & n = 2 \\ 2\,T(n/2) + 2 & n > 2 \end{cases}$
- T(n) = 2 T(n/2) + 2
  = 4 T(n/4) + 4 + 2
  = ...
  = n + n/2 + ... + 4 + 2 = 2n-2

Find MinMax D&C

- $T(n) = \begin{cases} 1 & n = 2 \\ 2\,T(n/2) + 2 & n > 2 \end{cases}$

- $T(n) = 2\,T(n/2) + 2$
  $= 4\,T(n/4) + 4 + 2$
  $= \ldots$
  $= n/2 + n/2 + \ldots + 4 + 2 = 3n/2-2$

Fibonacci rekursiv

- F(n)
  ```
  if n > 1
      return F(n-1) + F(n-2);
  else
      return n;
  ```
- T(0) = T(1) = t
- T(n+2) = t + T(n+1) + T(n)

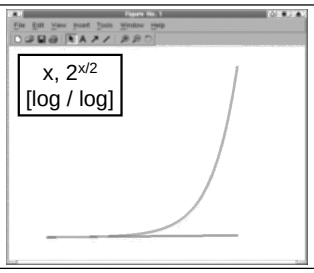
Fibonacci rekursiv

- Behauptung: $T(n) \geq t\, F(n+1)$
- Beweis: Vollständige Induktion ...
  - Anfang: $T(0) = t \geq t\, F(1) = t$
    $T(1) = t \geq t\, F(2) = t$
  - Schritt: $T(n+2) = t + T(n+1) + T(n)$
    $\geq t + t\, F(n+2) + t\, F(n+1)$
    $\geq t + t\, F(n+3)$
    $\geq t\, F(n+3)$
- $T(n) \in \Omega(2^{n/2})$

Fibonacci iterativ

- F(n)
  ```
  f_old = 1; f_new = 0;
  while (n > 0)
      f_new = f_new + f_old;
      f_old = f_new - f_old;
      n--;
  return f_new;
  ```
- $T(n) = t * n = \Theta(n)$

Fibonacci Vergleich



Standardabschätzungen

- Reihensummen
  - Geschachtelte Schleifen
  - Abschätzung durch Integrale
  - Vollständige Induktion
- Rekursionsgleichungen
  - Eliminierung
  - Master-Theorem
  - Direkter Ansatz

Reihensummen

- Geschachtelte Schleifen
  - Generell: der Aufwand $T_k(n)$ einer k-fach geschachtelten Schleife (mit linear beschränkten Laufintervallen) besitzt als obere Schranke ein Polynom vom Grad k

  $T_k(n) = O(\sum_{i=0}^{k} \alpha_i n^i) = O(n^k)$

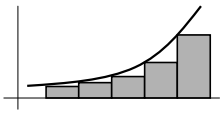
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals

$$\int_0^n f(x)dx \approx \sum_{i=0}^{n-1} f(i+h) \qquad h \in [0,1)$$

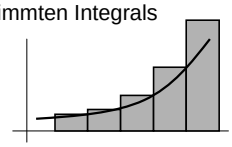
Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Abschätzung durch Integral

- Abschätzungen der Summe
  - Summanden monoton wachsend / fallend
  - Interpretation als Approximation eines bestimmten Integrals



Vollständige Induktion

- Behauptung
- Zeige: Behauptung gilt für $n=n_0$
- Zeige: Wenn die Behauptung für ein n gilt, dann gilt sie auch für n+1

Rekursionsgleichungen

- Elimination
- Master-Theorem
- Direkter Ansatz

Elimination

T(n) = T(n/2) + 1
= T(n/4) + 1 + 1
= T(n/8) + 1 + 1 + 1
= ...
= T(1) + log(n)
= O(log(n))

Binäre Suche

Elimination

$T(n) = 4\,T(n/2)$
$= 16\,T(n/4)$
$= 64\,T(n/8)$
$= \ldots$
$= 4^{\log_2(n)}\,T(1)$
$= n^2\,T(1)$
$= O(n^2)$

Multiplikation großer Zahlen

Master-Theorem

- $T(n) = \begin{cases} c & n = 1 \\ a\,T(n/b) + c\,n & n > 1 \end{cases}$
- $T(n) = \begin{cases} O(n) & a < b \\ O(n \log(n)) & a = b \\ O(n^{\log_b(a)}) & a > b \end{cases}$

Master-Theorem

- $T(n) = \begin{cases} c & n = 1 \\ a\,T(n/b) + O(n^p) & n > 1 \end{cases}$
- $T(n) = \begin{cases} O(n^p) & a < b^p \\ O(n^p \log(n)) & a = b^p \\ O(n^{\log_b(a)}) & a > b^p \end{cases}$

Multiplikation großer Zahlen

- Standard: $T(n) = 4\,T(n/2) + 3n$
  $= O(n^{\log_2(4)})$
  $= O(n^2)$
- Karatsuba: $T(n) = 3\,T(n/2) + 6n$
  $= O(n^{\log_2(3)})$
  $= O(n^{1.58})$

Direkter Ansatz

- $F(n) = F(n-1) + F(n-2)$
- Ansatz : $F(n) = \phi^n$
- $\phi^n = \phi^{n-1} + \phi^{n-2} \quad \Leftrightarrow \quad \phi^2 = \phi + 1$
  $\Leftrightarrow \quad \phi^2 - \phi - 1 = 0$
  $\Leftrightarrow \quad \phi_{\pm} = (1 \pm \sqrt{5}) / 2$

Direkter Ansatz

- $\phi_-^2 = \phi_- + 1 \quad \Rightarrow \quad \alpha\,\phi_-^n = \alpha\,\phi_-^{n-1} + \alpha\,\phi_-^{n-2}$
- $\phi_+^2 = \phi_+ + 1 \quad \Rightarrow \quad \beta\,\phi_+^n = \beta\,\phi_+^{n-1} + \beta\,\phi_+^{n-2}$
- $\Rightarrow \quad \alpha\,\phi_-^n + \beta\,\phi_+^n = \ldots$
- $F(n) := \alpha\,\phi_+^n + \beta\,\phi_-^n$

Direkter Ansatz

- $\phi_{\pm} = (1 \pm \sqrt{5}) / 2$
- $F(n) := \alpha \phi_+^n + \beta \phi_-^n$
- $F(0) = \alpha + \beta = 0 \Rightarrow \alpha = -\beta$
- $F(1) = \alpha \phi_+ + \beta \phi_- = \alpha (\phi_+ - \phi_-) = 1$
  $\Rightarrow \alpha = 1/\sqrt{5}$
- $F(n) = (\phi_+^n - \phi_-^n) / \sqrt{5}$

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Divide & Conquer

- Geg.: Problem der Größe n
- Divide: zerlege das Problem rekursiv in k Teilprobleme der Größe n/k
- Conquer: löse kleine Probleme direkt
- Kombiniere die Teillösungen zur Gesamtlösung

Find MinMax iterativ

- MinMax(A,1,n)
  min=1; max=1; i=2;
  while(i ≤ n)
  if (A[i] < A[min]) min = i;
  if (A[i] > A[max]) max = i;
  i++;
  return (A[min],A[max]);
- T(n) = 2n-2 ... (= 3n-3)

Find MinMax D&C

- MinMax(A,a,b)
  if (b - a ≤ 1)
  return(Min(A[a],A[b]),Max(A[a],A[b]));
  else
  (u_1,v_1) = MinMax(A,a,$\lfloor (a+b)/2 \rfloor$);
  (u_2,v_2) = MinMax(A, $\lfloor (a+b)/2 \rfloor$+1,b);
  return(Min(u_1,u_2),Max(v_1,v_2));

Find MinMax D&C

- $T(n) = \begin{cases} 1 & n = 2 \\ 2\,T(n/2) + 2 & n > 2 \end{cases}$
- T(n) = 2 T(n/2) + 2
  = 4 T(n/4) + 4 + 2
  = ...
  = n/2 + n/2 + ... + 4 + 2 = 3n/2-2

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Dynamisches Programmieren

- Reduziere gegebenes Problem auf kleinere Teilprobleme
- Löse Teilprobleme
- Speichere Teillösungen in Tabelle
- Kombiniere Teillösungen zur Gesamtlösung

Dynamisches Programmieren

- Typisches Muster ...
- *Gesucht:* beste Lösung $L_{1,n}$ von 1 bis n
- *Bestimme:* optimale Teillösungen $L_{1,k}$ und $L_{k+1,n}$ für k = 1, ..., n-1
- *Finde:* beste Kombination $L_{1,k}$, $L_{k+1,n}$

Dynamisches Programmieren

- Im Unterschied zu Divide & Conquer wird keine *top-down* Zerlegung, sondern eine *bottom-up* Kombination durchgeführt.
  - Iterative Implementierung
  - Speichere Zwischenergebnisse in einer Tabelle und vermeide dadurch doppelte Berechnungen

Beispiel

- Matrizenmultiplikation
- $A = [a_{ij}] \in IR^{p \times q}$, $B = [b_{ik}] \in IR^{q \times r}$
- $C = [c_{ik}] = A*B \in IR^{p \times r}$
- $c_{ik} = \sum_j a_{ij} b_{jk}$
- Zur Berechnung von C sind p*q*r skalare Multiplikationen notwendig

Beispiel

- Matrizenmultiplikation
- $A \in IR^{p \times q}$, $B \in IR^{q \times r}$, $C \in IR^{r \times s}$
- D = (A*B)*C = A*(B*C)
- (A*B)*C ... p*q*r + p*r*s Multiplikationen
- A*(B*C) ... q*r*s + p*q*s Multiplikationen

Beispiel

- Matrizenmultiplikation
- $A \in IR^{10 \times 1}$, $B \in IR^{1 \times 10}$, $C \in IR^{10 \times 1}$
- D = (A*B)*C = A*(B*C)
- (A*B)*C ... 10*1*10 + 10*10*1 = 200
- A*(B*C) ... 1*10*1 + 10*1*1 = 20

Beispiel

- Matrizenmultiplikation
- $A_i \in IR^{r_{i-1} \times r_i}$, i = 1, ..., n
- $M_{1,n}$: minimale Anzahl von skalaren Multiplikationen zur Berechnung von $B = A_1 * A_2 * ... * A_n$
- $M_{i,j}$: Zahl der Multiplikationen zur Berechnung von $A_i * ... * A_j$

Beispiel

- Matrizenmultiplikation
- $M_{1,n} = \min_{1 \le k < n} (M_{1,k} + M_{k+1,n} + r_0 * r_k * r_n)$
- $M_{i,j} = \min_{i \le k < j} (M_{i,k} + M_{k+1,j} + r_{i-1} * r_k * r_j)$
- Speichere die $M_{i,j}$ in einer Tabelle
- Basis-Fall: $M_{i,i} = 0$
- Berechne sukzessive $M_{i,i+k}$ für k=1, 2, ...


```
MatrixMultiply(r0, ..., rn)
  for (i=1 ; i≤n ; i++)
    Mi,i = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = Mi+1,i+k + ri-1*ri*ri+k;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if (Mi,j + Mj+1,i+k + ri-1*rj*ri+k < val)
          min = j;  val = Mi,j + Mj+1,i+k + ri-1*rj*ri+k;
      Si,i+k = min;  Mi,i+k = val;
```



```
MatrixMultiply(r0, ..., rn)
  for (i=1 ; i≤n ; i++)
    Mi,i = 0                          ... Initialisierung
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = Mi+1,i+k + ri-1*ri*ri+k;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if (Mi,j + Mj+1,i+k + ri-1*rj*ri+k < val)
          min = j;  val = Mi,j + Mj+1,i+k + ri-1*rj*ri+k;
      Si,i+k = min;  Mi,i+k = val;
```



```
MatrixMultiply(r0, ..., rn)
  for (i=1 ; i≤n ; i++)
    Mi,i = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)           ... Längenindex
      min = i;  val = Mi+1,i+k + ri-1*ri*ri+k;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if (Mi,j + Mj+1,i+k + ri-1*rj*ri+k < val)
          min = j;  val = Mi,j + Mj+1,i+k + ri-1*rj*ri+k;
      Si,i+k = min;  Mi,i+k = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}$*$r_i$*$r_{i+k}$;   … berechne $M_{i,i+k}$
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}$*$r_i$*$r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}$*$r_i$*$r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}$*$r_i$*$r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}$*$r_j$*$r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

Rechenaufwand

- Innerste Schleife: for (j=i+1 ; j≤i+k-1 ; j++)
  → O(i+k-1-i-1) = O(k-2) = O(k)
- Mittlere Schleife: for (i=1 ; i≤n-k ; i++)
  → O((n-k)*O(k)) = O(nk - k^2) = O(nk)
- Äußere Schleife: for (k=1 ; k≤n-1 ; k++)
  → O(n * O(nk)) = O(n^3)

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Memoization

- Effizienzsteigerung beim dynamischen Programmieren durch Speichern der Zwischenergebnisse in einer Tabelle
  → vermeide doppelte Berechnungen
- Allgemeines Konzept
  (lohnt sich, wenn der Tabellenzugriff schneller als die Neuberechnung ist)

Fibonacci (zum letzten Mal)

- MemoFibonacci(n)
  ```
  allocate array F[n+1];
  for (i=0 ; i≤n ; i++) F[ i ] = -1
  Fibo(F,n);
  ```
- Fibo(F,n)
  ```
  if (F[n] < 0)
      if n > 1
          F[n] = Fibo(F,n-1) + Fibo(F,n-2);
      else
          F[n] = n;
  ```


(2.0) Algorithmen

- (2.1) Analyse von Algorithmen
- (2.2) Entwurfsparadigmen
- (2.3) Sortieren
- (2.4) Suchen
- (2.5) ...

(2.3) Sortieren

- Bringe eine gegebene Folge von Datenobjekten in eine wohldefinierte Reihenfolge.
- Folge ≠ Menge !! (doppelte Objekte)
- Reihenfolge (Ordnungsrelation)

Ordnungsrelationen

- Menge M, Relation ≤
- Partielle Ordnung
  - Reflexivität $x \le x$
  - Transitivität $x \le y \wedge y \le z \Rightarrow x \le z$
  - Antisymmetrie $x \le y \wedge y \le x \Rightarrow x = y$
- Strikter Anteil
  - $x < y := x \le y \wedge x \ne y$

Ordnungsrelationen

- Menge M, Relation ≤
- Totale Ordnung
  - Trichotomie $\forall x,y : x<y \vee_x x=y \vee_x x>y$
- Sortierschlüssel
  - Object . Key
  - z.B. Adresse nach Nachnamen sortiert

Ordnungsrelationen

- Skalare Typen besitzen in der Regel eine totale Ordnung
- Mengen von Integer (2^{IN}), Mengeninklusion $\subseteq$
  - Partielle Ordnung ... {1,2}, {2,3}
- Punkte in der Ebene (IR x IR)
  - Keine Ordnung ... $(x,y) \leq (u,v) \Leftrightarrow x^2+y^2 \leq u^2+v^2$
  - Totale Ordnung ... $(x,y) \leq (u,v) \Leftrightarrow x \leq u \vee (x=u \wedge y \leq v)$ (***Lexikographisch***)

Sortierproblem

- Gegeben eine Folge von Objekten $R_1, ..., R_n$
- Finde eine Permutation $\Pi = \{ \pi(i) \}$, so dass $\forall i=1...n-1 : R_{\pi(i)}.key \leq R_{\pi(i+1)}.key$
- Insbesondere $\forall i,j : i<j \Rightarrow R_{\pi(i)}.key \leq R_{\pi(j)}.key$
- Schreibweise $R_{\pi(i)}.key \leq R_{\pi(j)}.key \Leftrightarrow R_i \leq R_j$

Warum sortieren?

- Bringe Objekte mit gleichen oder ähnlichen Schlüsseln zusammen
- Detektiere doppelte Objekte
- Vorbereitung zum Suchen
  - $O(n) \Rightarrow O(\log(n))$
  - Lohnt sich nur, wenn $m * n * C_1 > C_{sort}(n) + m * \log(n) * C_2$

Begriffe

- Direktes / Indirektes Sortieren
  - Direkt: Umkopieren der Objekte R_i
  - Indirekt: Erzeuge Permutationstabelle $\Pi = \{ \pi(i) \}$
    - Initialisierung $\pi(i) = i$
    - Zugriff per $R_{\pi(i)}$
    - Sortieren bzgl. ***mehrerer*** Schlüssel / Relationen
- Stabilität (Reihenfolge von Objekten mit gleichem Schlüssel bleibt erhalten)

Kriterien

- Berechnungsaufwand
  - $O(n^2)$, $O(n \log(n))$, $O(n)$
- Worst vs. Average Case
  - Ausnutzen von Vorsortierungen
- Speicherbedarf
  - Kopieren
  - In-place

Typen von Sortieralgorithmen

- Einfache
  - Selection-Sort, Bubble-Sort, Insertion-Sort, ...
- Höhere
  - Quick-Sort, Merge-Sort, Heap-Sort, ...
- Spezielle
  - Bucket-Sort, Radix-Sort, ...

Selection-Sort

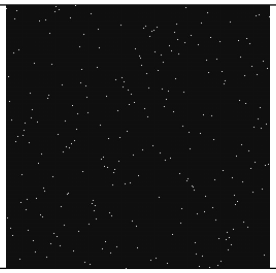
- Idee:
  - Suche kleinstes / größtes Element
  - Kopiere es an die erste Stelle
  - Wende den gleichen Algorithmus auf die restlichen (n-1) Elemente an
- Vorteil: Jedes Objekt wird nur dreimal kopiert (lohnt sich bei großen Objekten)

Selection-Sort

- SelectionSort(A,n)

```
i = 0;
while (i < n)
    min = i;  j = i+1;
    while (j < n)
        if (A[j] < A[min])  min = j;
        j++:
    swap(A[i],A[min]);
    i++;
```


Selection-Sort



Selection-Sort

- Aufwandsabschätzung
  - Swap(a,b) : { c = a; a = b; b = c } = 3 x Kopieren
  - n x Swap = 3n C_{ass} = O(n)
  - Im i-ten Schleifendurchlauf werden (n-i) Vergleiche durchgeführt
  - (n-1) + (n-2) + ... + 1 = n (n-1) / 2 = $O(n^2)$
- Best = Worst = Average Case

Bubble-Sort

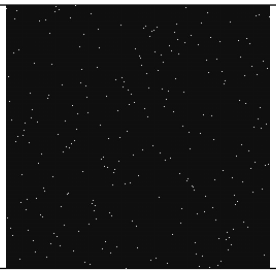
- Idee
  - Nutze die Vergleiche in Selection-Sort, um die unsortierten Teil A[i] ... A[n] der Objekte vorzusortieren.
  - Innere Schleife läuft in ungekehrter Richtung
  - Swap-Operation in der inneren Schleife

Bubble-Sort

- BubbleSort(A,n)

```
i = 0;
while (i < n)
    j = n;
    while (j > i)
        if (A[ j ] < A[ j-1])
            swap(A[ j ], A[ j-1]);
        j--;
    i++;
```

Bubble-Sort



Bubble-Sort

- ***Vorteil:*** pro inneren Schleifendurchlauf wird mehr Ordnung geschaffen
- ***Nachteil:*** Vertauschung nur zwischen direkten Nachbarn → Objekte, die weit wandern werden oft kopiert.
- Vorsortierung kann nicht ausgenutzt werden

Bubble-Sort

- Aufwandsabschätzung
  - # Vergleiche unabhängig von der Verteilung
    $n (n-1) / 2 = O(n^2)$
  - # Swaps
    - Best Case: 0 (Elemente schon sortiert)
    - Worst Case: $n (n-1) / 2 = O(n^2)$
      (Inverse Vorsortierung)
    - Average Case: $n (n-1) / 4 = O(n^2)$

Insertion-Sort

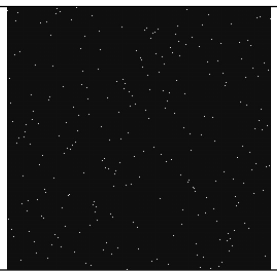
- Idee
  - Vorbild: manuelles Sortieren
  - In jedem Schritt:
    Nehme das nächste Element und füge es in der schon sortierten Teilfolge an der richtigen Stelle ein

Insertion-Sort

- InsertionSort(A,N)

```
i=1;
while (i < n)
    h = A[i];  j = i;
    while (j > 0) ∧ (A[j-1] > h)
        A[j] = A[j-1];
        j--;
    A[j] = h;
    i++;
```


Insertion-Sort



Insertion-Sort

- Aufwandsabschätzung
  - Vergleiche
    - Best Case: n-1 (vollständig vorsortiert)
    - Worst Case: $n (n-1) / 2 = O(n^2)$ (invers vorsortiert)
    - Average Case: $n (n-1) / 4 = O(n^2)$ (alle Listenplätze gleich wahrscheinlich)
  - Kopieren:
    Faktor 3 besser, da einfaches Kopieren anstatt Swap(), aber noch immer $O(n^2)$

Vorsortierung

- Größere Elemente treten mit höherer Wahrscheinlichkeit am Ende der Folge auf
- Größere Elemente werden später in die sortierte Teilfolge eingeordnet.
- Dazu sind weniger Vergleiche notwendig, denn größere Elemente stehen auch in der sortierten Teilfolge weiter hinten. (Einsortieren von hinten)
- Nahezu linearer Aufwand für „fast sortierte“ Folgen

Einfache Sortieralgorithmen

- Vergleiche (Best / Average / Worst)
  - Selection-Sort $n^2/2$ / $n^2/2$ / $n^2/2$
  - Bubble-Sort $n^2/2$ / $n^2/2$ / $n^2/2$
  - Insertion-Sort n / $n^2/4$ / $n^2/2$
- Kopieren (Best / Average / Worst)
  - Selection-Sort $3(n-1)$ / $3(n-1)$ / $3(n-1)$
  - Bubble-Sort 0 / $3n^2/4$ / $3n^2/2$
  - Insertion-Sort $2(n-1)$ / $n^2/4$ / $n^2/2$

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Dynamisches Programmieren

- Typisches Muster ...
- *Gesucht:* beste Lösung $L_{1,n}$ von 1 bis n
- *Bestimme:* optimale Teillösungen $L_{1,k}$ und $L_{k+1,n}$ für k = 1, ..., n-1
- *Finde:* beste Kombination $L_{1,k}$, $L_{k+1,n}$

Dynamisches Programmieren

- Im Unterschied zu Divide & Conquer wird keine *top-down* Zerlegung, sondern eine *bottom-up* Kombination durchgeführt.
  - Iterative Implementierung
  - Speichere Zwischenergebnisse in einer Tabelle und vermeide dadurch doppelte Berechnungen

Beispiel

- Matrizenmultiplikation
- $A_i \in IR^{r_{i-1} \times r_i}$, i = 1, ..., n
- $M_{1,n}$: minimale Anzahl von skalaren Multiplikationen zur Berechnung von $B = A_1 * A_2 * ... * A_n$
- $M_{i,j}$: Zahl der Multiplikationen zur Berechnung von $A_i * ... * A_j$

Beispiel

- $M_{1,n} = \min_{1 \le k < n} (M_{1,k} + M_{k+1,n} + r_0 * r_k * r_n)$
- $M_{i,j} = \min_{i \le k < j} (M_{i,k} + M_{k+1,j} + r_{i-1} * r_k * r_j)$
- $(A_i * ... * A_k) * (A_{k+1} * ... * A_j)$
- Speichere die $M_{i,j}$ in einer Tabelle
- Basis-Fall: $M_{i,i} = 0$
- Berechne sukzessive $M_{i,i+k}$ für k=1, 2, ...


```
MatrixMultiply(r0, ..., rn)
  for (i=1 ; i≤n ; i++)
    Mi,i = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = Mi+1,i+k + ri-1*ri*ri+k;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if (Mi,j + Mj+1,i+k + ri-1*rj*ri+k < val)
          min = j;  val = Mi,j + Mj+1,i+k + ri-1*rj*ri+k;
      Si,i+k = min;  Mi,i+k = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0                                   ... Initialisierung
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)                         ... Längenindex
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;   ... berechne $M_{i,i+k}$
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

```
MatrixMultiply($r_0$, ..., $r_n$)
  for (i=1 ; i≤n ; i++)
    $M_{i,i}$ = 0
  for (k=1 ; k≤n-1 ; k++)
    for (i=1 ; i≤n-k ; i++)
      min = i;  val = $M_{i+1,i+k}$+ $r_{i-1}*r_i*r_{i+k}$;
      for (j=i+1 ; j≤i+k-1 ; j++)
        if ($M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$ < val)
          min = j;  val  = $M_{i,j}$ + $M_{j+1,i+k}$+ $r_{i-1}*r_j*r_{i+k}$;
      $S_{i,i+k}$ = min;  $M_{i,i+k}$ = val;
```

Rechenaufwand

- Innerste Schleife: for (j=i+1 ; j≤i+k-1 ; j++)
  → O(i+k-1-i-1) = O(k-2) = O(k)
- Mittlere Schleife: for (i=1 ; i≤n-k ; i++)
  → O((n-k)*O(k)) = O(nk - k^2) = O(nk)
- Äußere Schleife: for (k=1 ; k≤n-1 ; k++)
  → O(n * O(nk)) = O(n^3)

(2.2) Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)

Fibonacci (zum letzten Mal)

- MemoFibonacci(n)
  allocate array F[n+1];
  for (i=0 ; i≤n ; i++) F[i] = -1
  Fibo(F,n);
- Fibo(F,n)
  if (F[n] < 0)
  if n > 1
  F[n] = Fibo(F,n-1) + Fibo(F,n-2);
  else
  F[n] = n;

(2.3) Sortieren

- Bringe eine gegebene Folge von Datenobjekten in eine wohldefinierte Reihenfolge.
- Folge ≠ Menge !! (doppelte Objekte)
- Reihenfolge (Ordnungsrelation)

Sortierproblem

- Gegeben eine Folge von Objekten
  $R_1, ..., R_n$
- Finde eine Permutation $\Pi = \{ \pi(i) \}$, so dass
  $\forall i=1...n-1 : R_{\pi(i)}.key \leq R_{\pi(i+1)}.key$
- Insbesondere
  $\forall i,j : i<j \Rightarrow R_{\pi(i)}.key \leq R_{\pi(j)}.key$
- Schreibweise
  $R_{\pi(i)}.key \leq R_{\pi(j)}.key \Leftrightarrow R_i \leq R_j$

Kriterien

- Berechnungsaufwand
  – $O(n^2)$, O(n log(n)), O(n)
- Worst vs. Average Case
  – Ausnutzen von Vorsortierungen
- Speicherbedarf
  – Kopieren
  – In-place

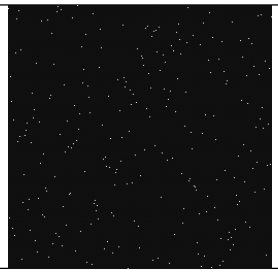
Typen von Sortieralgorithmen

- Einfache
  - Selection-Sort, Bubble-Sort, Insertion-Sort, ...
- Höhere
  - Quick-Sort, Merge-Sort, Heap-Sort, ...
- Spezielle
  - Bucket-Sort, Radix-Sort, ...

Selection-Sort

- Idee:
  - Suche kleinstes / größtes Element
  - Kopiere es an die erste Stelle
  - Wende den gleichen Algorithmus auf die restlichen (n-1) Elemente an
- Vorteil: Jedes Objekt wird nur dreimal kopiert (lohnt sich bei großen Objekten)

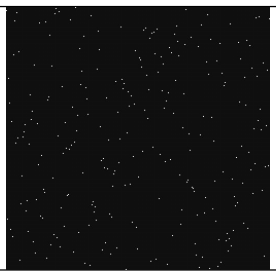
Selection-Sort



Bubble-Sort

- Idee
  - Nutze die Vergleiche in Selection-Sort, um die unsortierten Teil A[i] ... A[n] der Objekte vorzusortieren.
  - Innere Schleife läuft in umgekehrter Richtung
  - Swap-Operation in der inneren Schleife

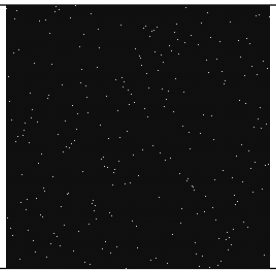
Bubble-Sort



Insertion-Sort

- Idee
  - Vorbild: manuelles Sortieren
  - In jedem Schritt:
    Nehme das nächste Element und füge es in der schon sortierten Teilfolge an der richtigen Stelle ein

Insertion-Sort



Einfache Sortieralgorithmen

- Vergleiche (Best / Average / Worst)
  - Selection-Sort $n^2/2$ / $n^2/2$ / $n^2/2$
  - Bubble-Sort $n^2/2$ / $n^2/2$ / $n^2/2$
  - Insertion-Sort n / $n^2/4$ / $n^2/2$
- Kopieren (Best / Average / Worst)
  - Selection-Sort $3(n-1)$ / $3(n-1)$ / $3(n-1)$
  - Bubble-Sort 0 / $3n^2/4$ / $3n^2/2$
  - Insertion-Sort $2(n-1)$ / $n^2/4$ / $n^2/2$

Typen von Sortieralgorithmen

- Einfache
  - Selection-Sort, Bubble-Sort, Insertion-Sort, ...
- Höhere
  - Quick-Sort, Merge-Sort, Heap-Sort, ...
- Spezielle
  - Bucket-Sort, Radix-Sort, ...

Quick-Sort

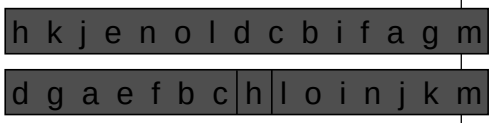
- Idee
  - Divide & Conquer
  - Wähle ein Element R aus
  - Teile Folge in zwei Sub-Folgen
    - $R_L \leq R$
    - $R_R \geq R$
  - Sortiere R_L und R_R durch rekursiven Aufruf
  - Setze Teilfolgen zusammen: $R_L \oplus R \oplus R_R$

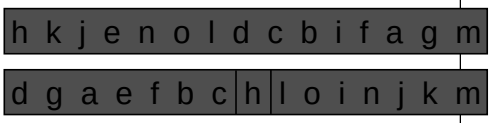
Quick-Sort

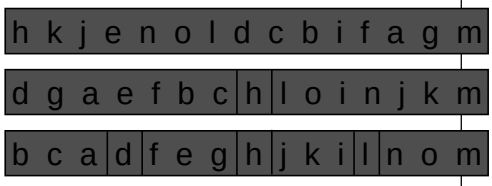
h k j e n o l d c b i f a g m

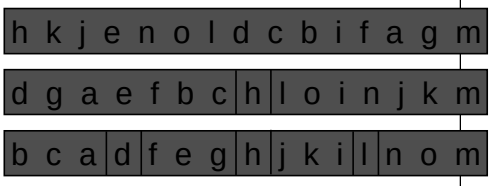
Quick-Sort

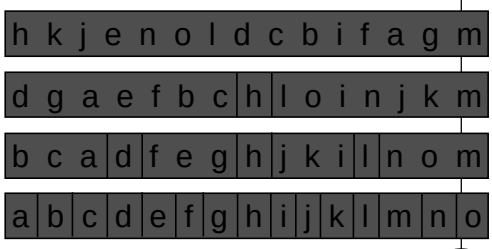
h k j e n o l d c b i f a g m











Quick-Sort

h k j e n o l d c b i f a g m

d g a e f b c | h | l o i n j k m

Quick-Sort

h k j e n o l d c b i f a g m

d g a e f b c | h | l o i n j k m

Quick-Sort

h k j e n o l d c b i f a g m

d g a e f b c | h | l o i n j k m

b c a | d | f e g | h | j k i | l | n o m

Quick-Sort

h k j e n o l d c b i f a g m

d g a e f b c | h | l o i n j k m

b c a | d | f e g | h | j k i | l | n o m

Quick-Sort

h k j e n o l d c b i f a g m

d g a e f b c | h | l o i n j k m

b c a | d | f e g | h | j k i | l | n o m

a | b | c | d | e | f | g | h | i | j | k | l | m | n | o

Quick-Sort

- QuickSort(A,l,r)

```
if (l < r)
        m = Partition(A,l,r);
        QuickSort(A,l,m-1    );
        QuickSort(A,  m+1,r);
```

- Aufruf mit: QuickSort(A,1,n);

Implementierung

- Partition(A,l,r)

```
i = l-1;   j = r;
while (i < j)
    i++;  while (A[ i ] < A[ r ])  i++;
    j--;    while (A[ j ] > A[ r ])  j--;
    swap(A[ i ], A[ j ]);
swap(A[ i ], A[ j ]);
swap(A[ i ], A[ r ]);
```


Implementierung

- Achtung: Verwende ***Sentinel***, um korrekte Terminierung der inneren Schleife zu garantieren.
- Aufruf mit: A[0] := -∞
  QuickSort(A,1,n);
- while (A[i] < A[r]) i++;
  while (A[j] > A[r]) j--;

Bricht spätestens für i = r ab.

für j = r+1 ab.

Beispiel

k j e n o c d l b i f a g m h

Beispiel

k j e n o c d l b i f a g m h

Beispiel

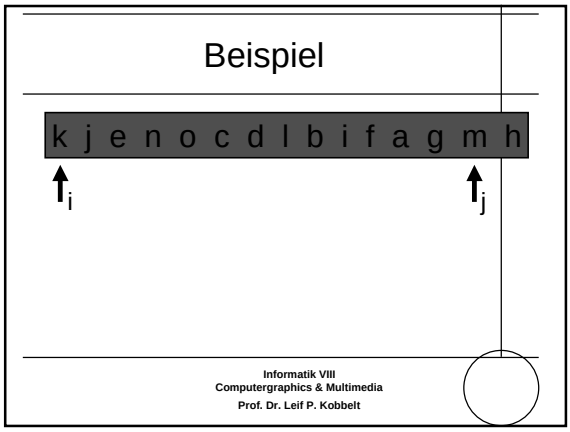
k j e n o c d l b i f a g m h

i j

Beispiel

k j e n o c d l b i f a g m h

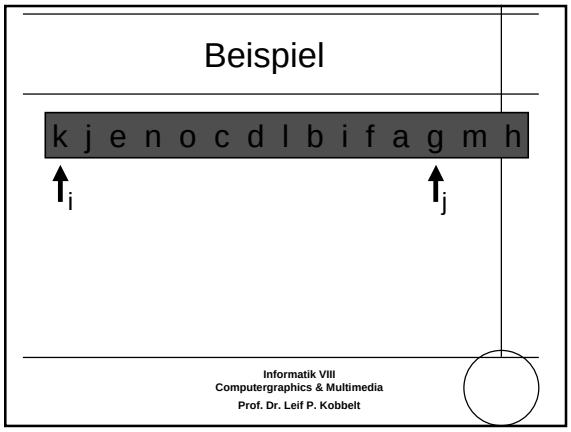
i j



Beispiel

k j e n o c d l b i f a g m h

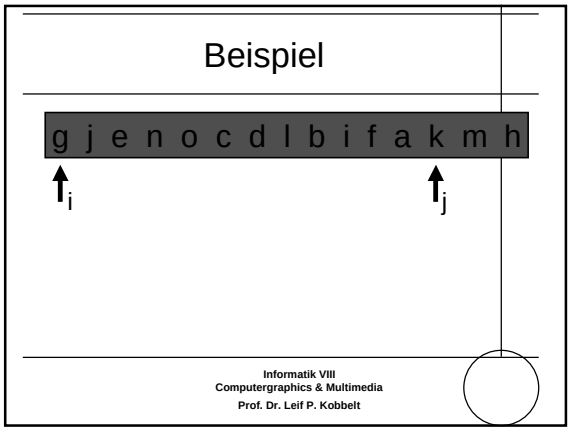
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

k j e n o c d l b i f a g m h

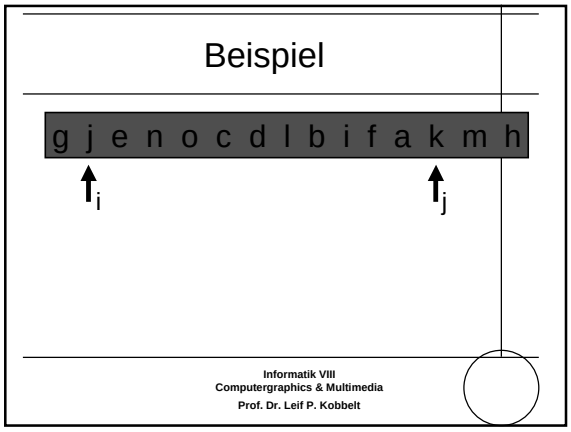
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g j e n o c d l b i f a k m h

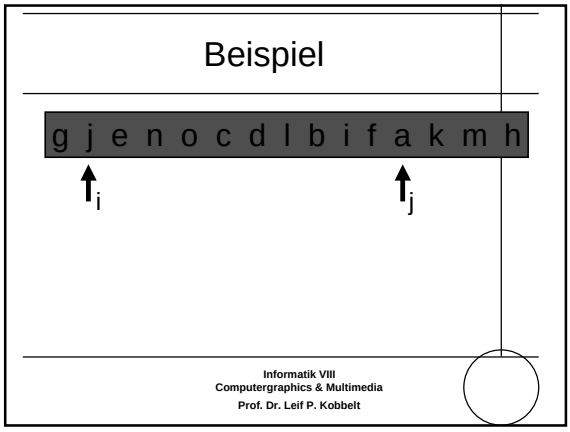
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g j e n o c d l b i f a k m h

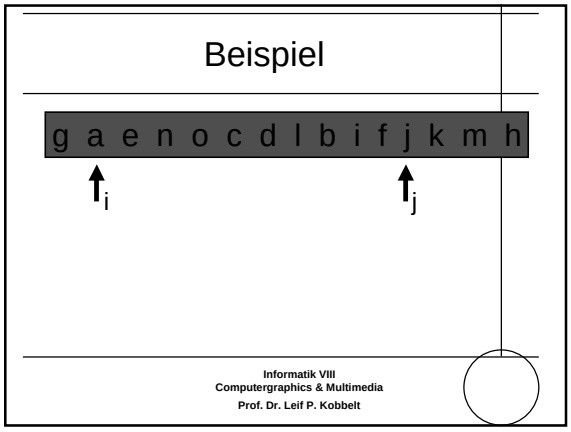
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g j e n o c d l b i f a k m h

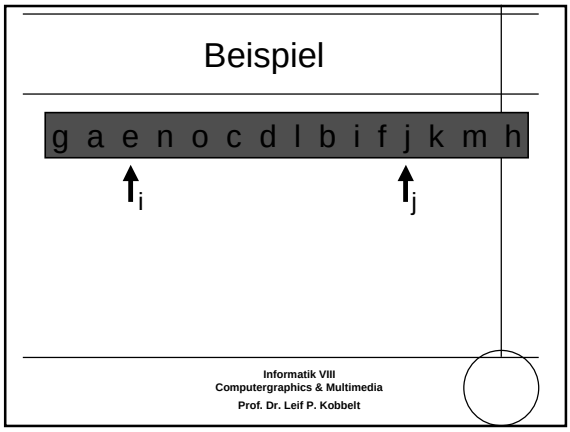
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e n o c d l b i f j k m h

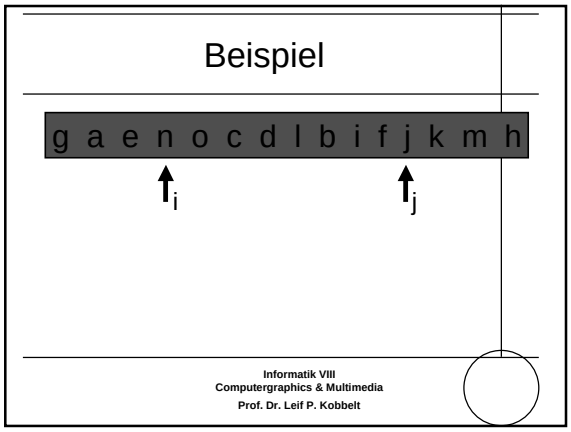
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e n o c d l b i f j k m h

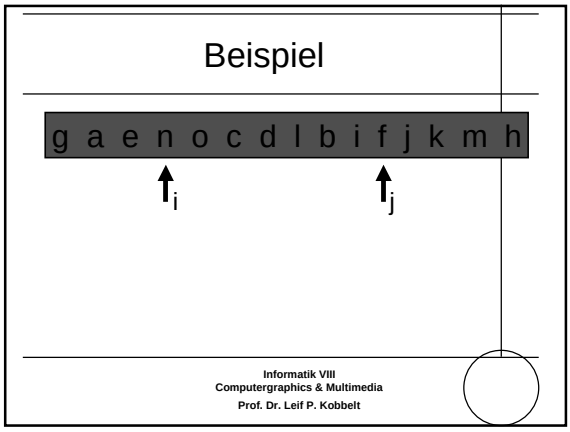
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e n o c d l b i f j k m h

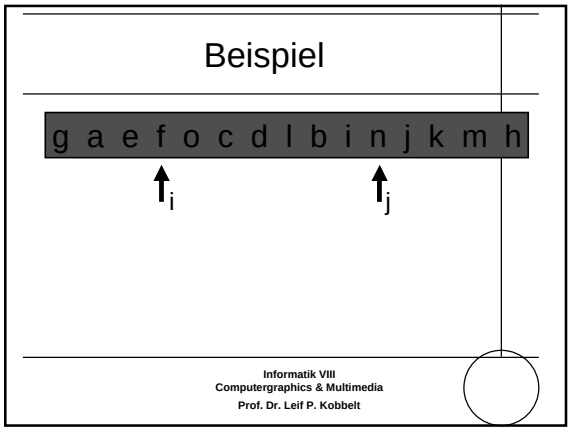
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e n o c d l b i f j k m h

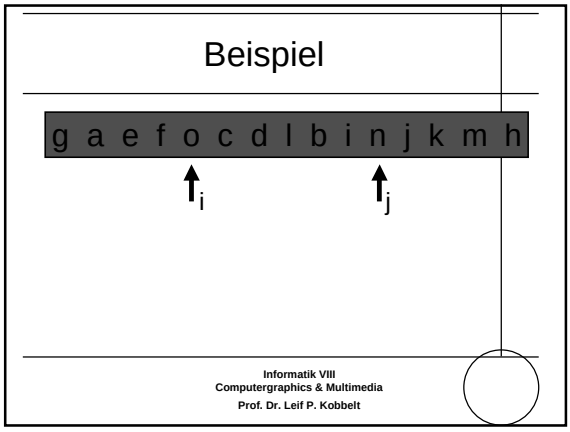
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e f o c d l b i n j k m h

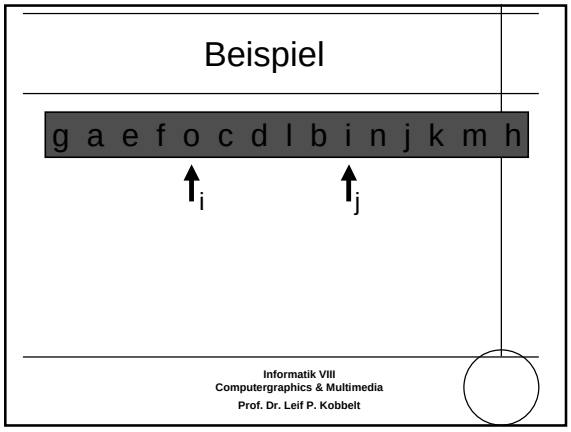
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e f o c d l b i n j k m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Beispiel

g a e f o c d l b i n j k m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Beispiel

Beispiel

Beispiel

Beispiel

Beispiel

Beispiel

Beispiel

g a e f b c d l o i n j k m h

$\uparrow_j$ $\uparrow_i$

Beispiel

g a e f b c l d o i n j k m h

$\uparrow_j$ $\uparrow_i$

Beispiel

g a e f b c d l o i n j k m h

$\uparrow_j$ $\uparrow_i$

Beispiel

g a e f b c d h o i n j k m l

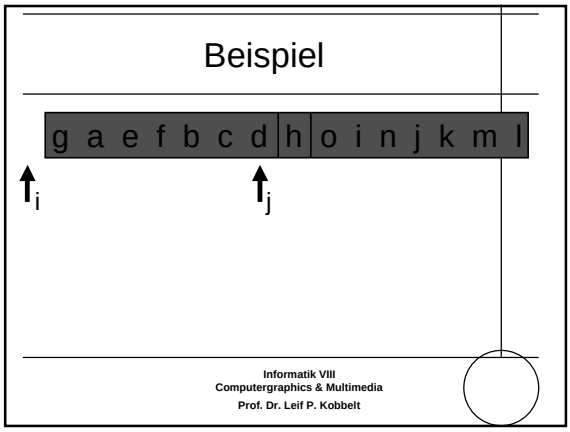
$\uparrow_j$ $\uparrow_i$

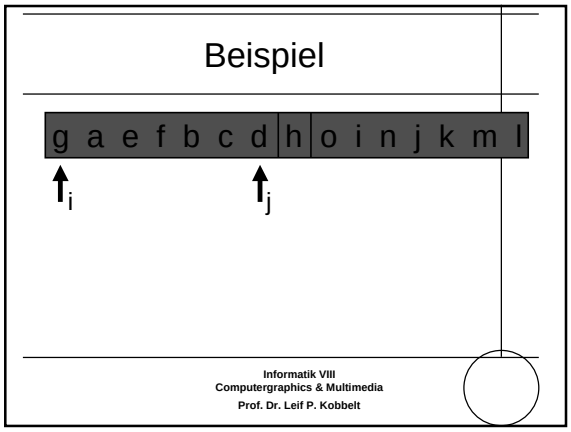
Beispiel

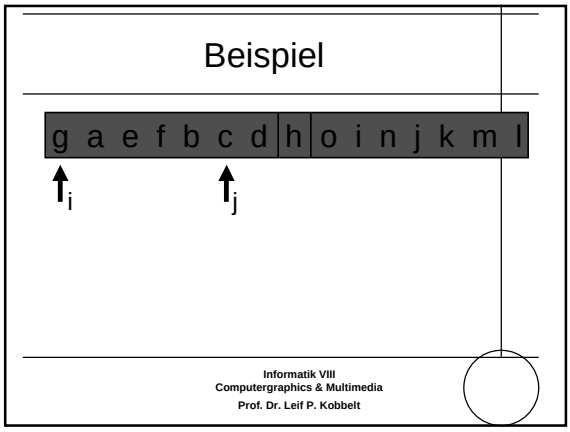
g a e f b c d | h | o i n j k m l

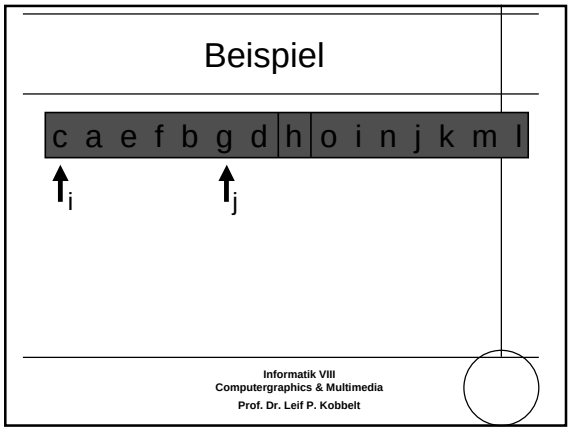
Beispiel

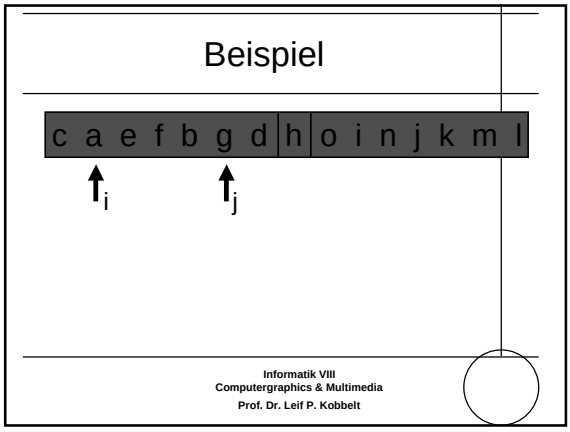
g a e f b c d | h | o i n j k m l

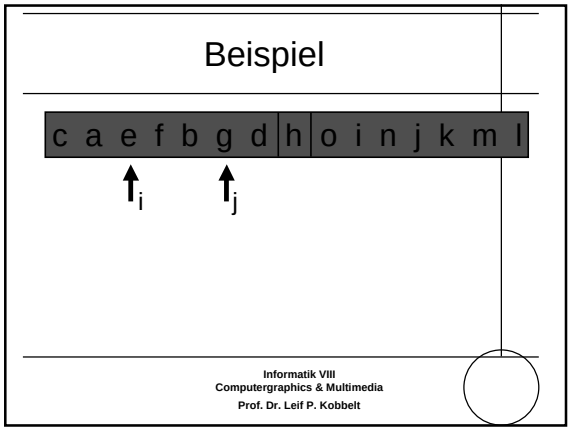












Beispiel

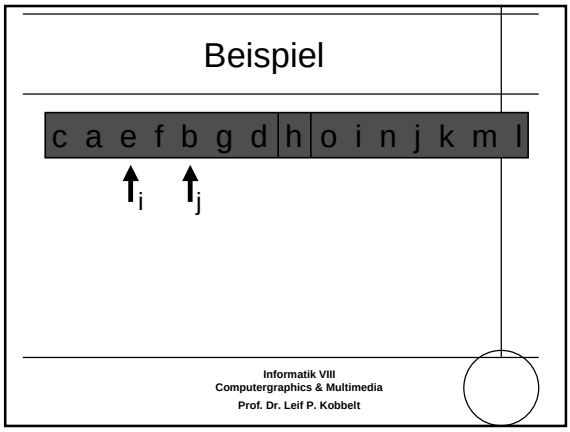
Beispiel

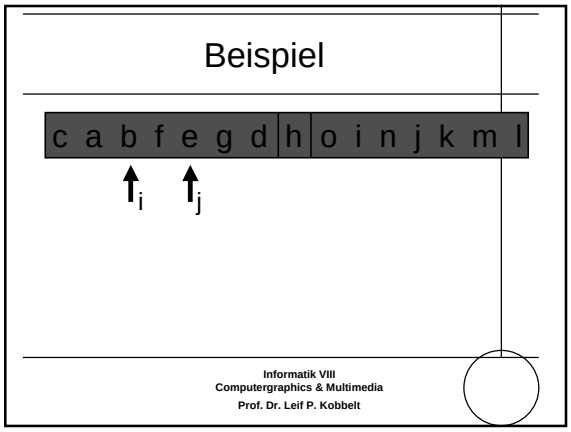
Beispiel

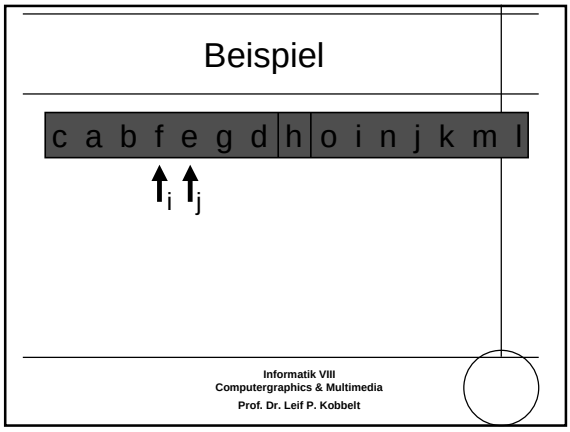
Beispiel

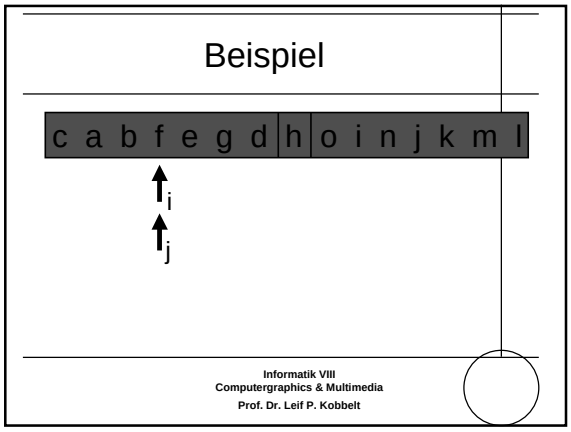
Beispiel

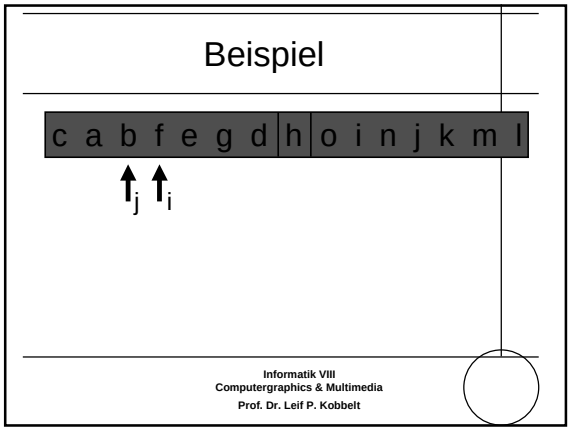
Beispiel

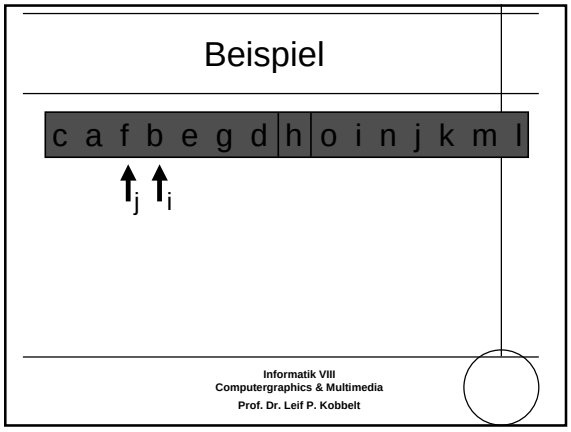

Beispiel
c a e f b g d h o i n j k m l
i
j
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Beispiel
c a b f e g d h o i n j k m l
i
j
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

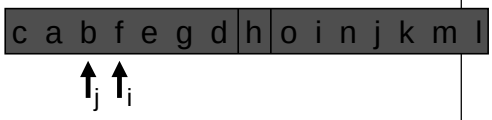

Beispiel
c a b f e g d h o i n j k m l
i j
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Beispiel
c a b f e g d h o i n j k m l
i
j
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

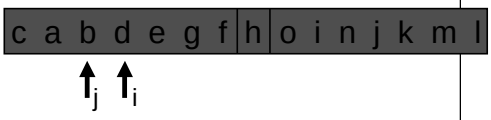

Beispiel
c a b f e g d h o i n j k m l
j i
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Beispiel
c a f b e g d h o i n j k m l
j i
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

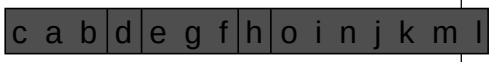
Beispiel



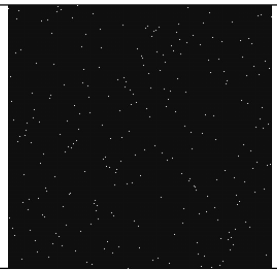
Beispiel



Beispiel



Quick-Sort



Aufwandsabschätzung

- Aufwand
  - Best Case: $T(n) \approx 2\,T(n/2) + O(n) \Rightarrow O(n \log(n))$
    (Folge zerfällt immer in zwei gleich große Teile)
  - Worst Case: $T(n) \approx T(n-1) + O(n) \Rightarrow O(n^2)$
    (Folge ist bereits auf- oder absteigend sortiert)
  - Average Case: ...

Aufwandsabschätzung

- Aufwand
  - Average Case:

$$T(n) \approx O(n) + \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i) + T(n-1-i)\right)$$

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

Aufwandsabschätzung

$$n \cdot T(n) = cn^2 + 2\sum_{i=0}^{n-1} T(i)$$

$$(n-1) \cdot T(n-1) = c(n-1)^2 + 2\sum_{i=0}^{n-2} T(i)$$

$$n \cdot T(n) - (n-1) \cdot T(n-1) = cn^2 - c(n-1)^2 + 2T(n-1)$$

Aufwandsabschätzung

$$n \cdot T(n) = 2cn - c + (n+1) \cdot T(n-1)$$

$$T(n) = c' + \frac{n+1}{n} T(n-1)$$

$$= c' + \frac{n+1}{n} c' + \frac{n+1}{n-1} T(n-2)$$

$$= c' + \frac{n+1}{n} c' + \frac{n+1}{n-1} c' + \frac{n+1}{n-2} T(n-3)$$

Aufwandsabschätzung

$$T(n) = c'\left(\frac{n+1}{n+1} + \frac{n+1}{n} + \frac{n+1}{n-1} + \ldots + \frac{n+1}{1}\right)$$

$$= c'(n+1)\sum_{i=1}^{n+1} \frac{1}{i}$$

$$\leq c''(n+1)\log(n+1) = O(n\log(n))$$

Aufwandsabschätzung

$$\sum_{i=1}^{n+1} \frac{1}{i} \leq 1 + \int_1^{n+1} \frac{1}{x}\, dx = 1 + \ln(x)\Big|_1^{n+1}$$

$$= 1 + \ln(n+1) = O(\log(n+1))$$

Aufwandsabschätzung

- Average Case: T(n) = O(n log(n))
- Heuristiken zur Verbesserung der Performanz
  - Pivot-Element: (A[l]+A[r])/2, ...
  - Bei kleinen Folgen auf Insertion-Sort wechseln (oder auch nicht: *Cache-Kohärenz*)
  - Iterative Implementierung mit Stack

Merge-Sort

- Idee
  - Quick-Sort ist schnell, wenn die beiden Teilfolgen nach Partition() ungefähr die gleiche Größe haben.
  - Teile die Folge ohne Vorsortieren in zwei gleiche Teile
  - Sortiere die Teilfolgen
  - Kombiniere die Teilergebnisse

Merge-Sort (rekursiv)

- MergeSort(A,l,r)

```
if (l < r)
    m = ⌊(l+r)/2⌋;
    MergeSort(A,l,m      );
    MergeSort(A,  m+1,r);
    Merge(A,l,m,r);
```

... vgl. mit Quick-Sort

- Aufruf mit: MergeSort(A,1,n);

Merge-Sort (rekursiv)

- Merge(A,l,m,r)

```
new B[r-l+1];   i = 0;   j = l;   k = m+1;
while (j ≤ m) ∧ (k ≤ r)
    if (A[ j ] ≤ A[k])
        B[ i++] = A[ j++];
    else
        B[i++] = A[k++];
while (j ≤ m) B[i++] = A[ j++];
while (k ≤ r ) B[i++] = A[k++];
for (i = l ; i ≤ r ; i++)  A[ i ] = B[ i-l ]
```


Merge-Sort (rekursiv)

k j e n o c d l b i f a g m h

Merge-Sort (rekursiv)

k j e n o c d l | b i f a g m h

Merge-Sort (rekursiv)

k j e n | o c d l | b i f a g m h

Merge-Sort (rekursiv)

k j | e n | o c d l | b i f a g m h

Merge-Sort (rekursiv)

k j e n o c d l b i f a g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

j k e n o c d l b i f a g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

j k e n o c d l b i f a g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

j k e n o c d l b i f a g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

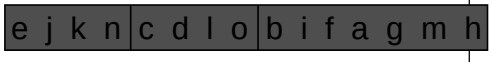
e j k n o c d l b i f a g m h

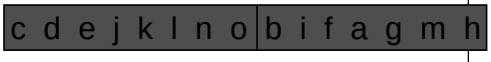
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

e j k n o c d l b i f a g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt





Merge-Sort (rekursiv)

e j k n	o	c	d l	b i f a g m h

Merge-Sort (rekursiv)

e j k n	c o	d l	b i f a g m h

Merge-Sort (rekursiv)

e j k n	c o	d	l	b i f a g m h

Merge-Sort (rekursiv)

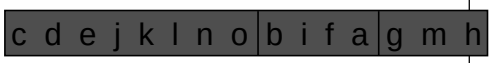
e j k n	c o	d l	b i f a g m h

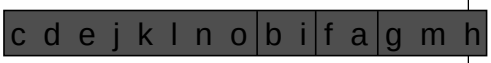
Merge-Sort (rekursiv)

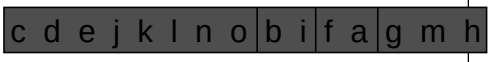
e j k n	c d l o	b i f a g m h

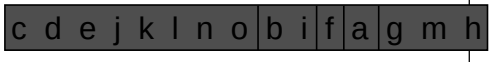
Merge-Sort (rekursiv)

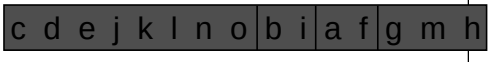
c d e j k l n o	b i f a g m h











Merge-Sort (rekursiv)

c d e j k l n o	b i f a	g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o	b i	f a	g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o	b	i	f a	g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o	b i	f a	g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

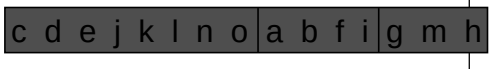
c d e j k l n o	b i	f	a	g m h

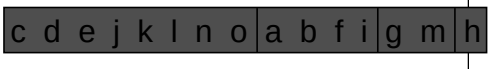
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

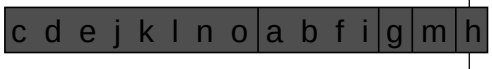
Merge-Sort (rekursiv)

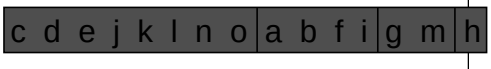
c d e j k l n o	b i	a f	g m h

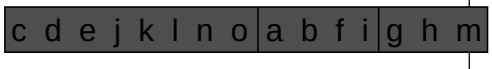
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

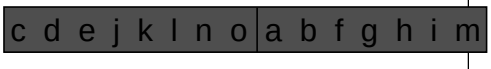












Merge-Sort (rekursiv)

c d e j k l n o a b f i g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o a b f i g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o a b f i g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o a b f i g m h

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

c d e j k l n o a b f i g h m

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

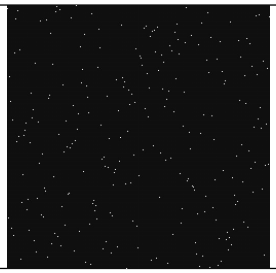
c d e j k l n o a b f g h i m

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Merge-Sort (rekursiv)

a b c d e f g h i j k l m n o

Merge-Sort (depth-first)



Aufwandsabschätzung

- Das Mischen zweier Folgen der Länge L_1 und L_2 erfordert ***höchstens*** L_1+L_2 Vergleiche und ***genau*** $2\,(L_1+L_2)$ Kopieroperationen *(i++ in jedem Schleifendurchlauf)*
- Summe aller Folgenlängen auf jeder Rekursionsstufe = n
- Anzahl der Rekursionsstufen = log(n)
- Best = Worst = Average Case = O(n log(n))
- Aber: ***Doppelter Speicherbedarf !***

Merge-Sort (iterativ)

- Die Zerlegung verändert die Folge nicht.
- Die *top-down* Rekursion steuert nur die Reihenfolge beim Mischen.
- Fasse die unsortierte Liste als Folge von ***sortierten einelementigen Folgen*** auf.
- Mische die Teilfolgen *bottom-up*

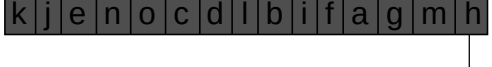
- MergeSort(A,n)

```
new B[n];   m = 1;   k = n;
while (m < n)
    B = A;   k = ⌈k/2⌉;
    for (i=0 ; i<2*k*m ; i+=2*m)
        p = 0;   q = 0;
        for (j=0 ; j<Min(2*m,n-i) ; j++)
            if (p ≤ m) ∧ (B[i+p] ≤ B[i+m+q])
                                ∨ (q ≥ Min(m,n-i-m) )
                A[i+j] = B[i+p];   p++;
            else
                A[i+j] = B[i+m+q];   q++;
        m = 2*m;
```

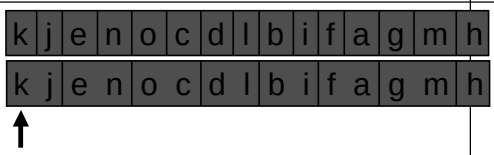

Merge-Sort (iterativ)

k j e n o c d l b i f a g m h

Merge-Sort (iterativ)

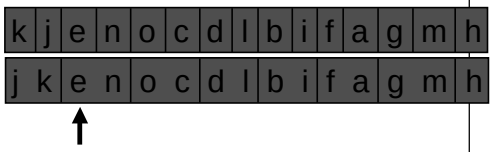

k j e n o c d l b i f a g m h

Merge-Sort (iterativ)


k j e n o c d l b i f a g m h

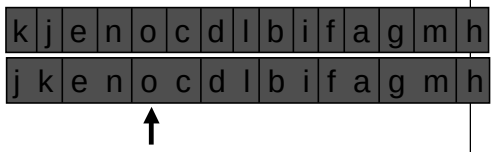
k j e n o c d l b i f a g m h

Merge-Sort (iterativ)


k j e n o c d l b i f a g m h

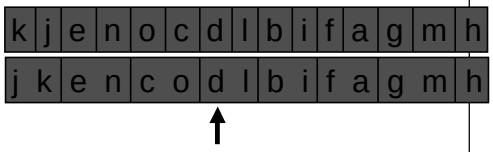
j k e n o c d l b i f a g m h

Merge-Sort (iterativ)


k j e n o c d l b i f a g m h

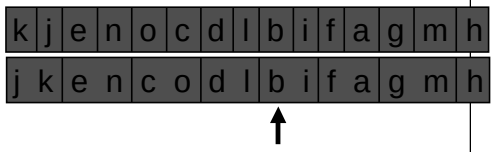
j k e n o c d l b i f a g m h

Merge-Sort (iterativ)

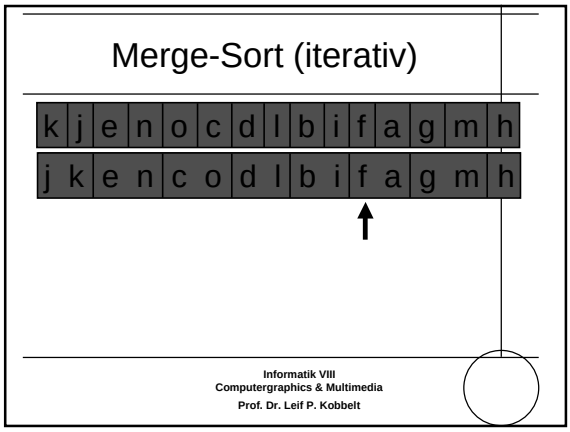

k j e n o c d l b i f a g m h

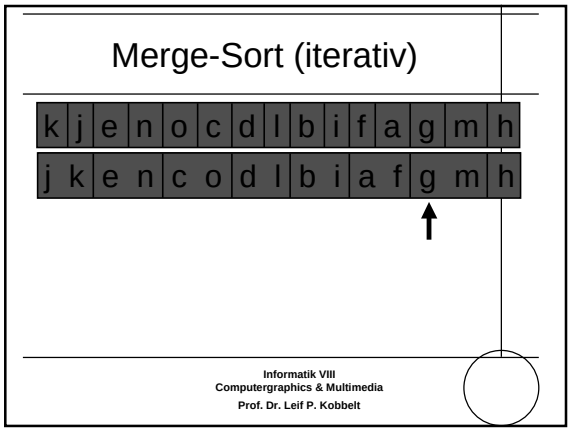
j k e n c o d l b i f a g m h

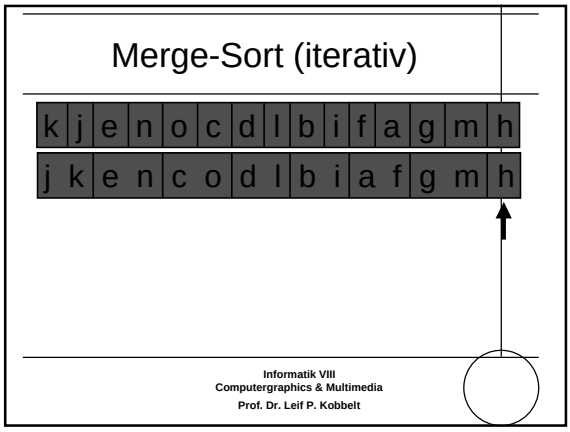
Merge-Sort (iterativ)

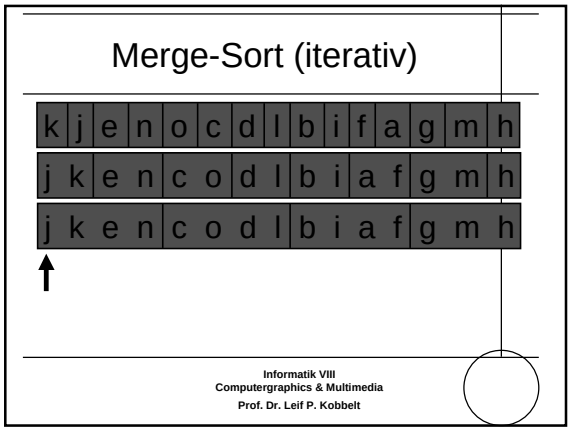

k j e n o c d l b i f a g m h

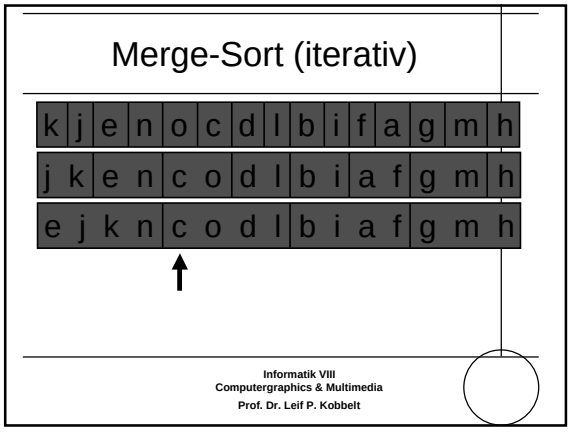
j k e n c o d l b i f a g m h

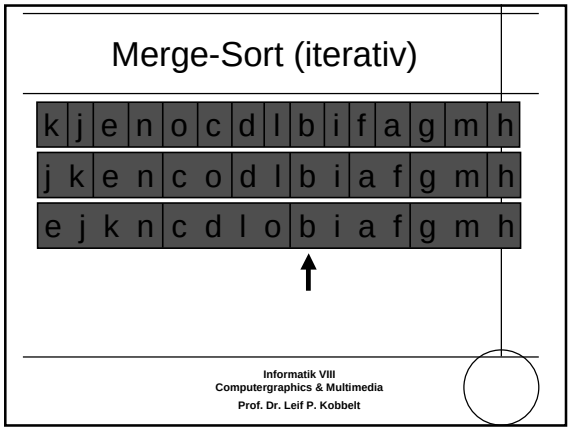












Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

j k | e n | c o | d l | b i | f | a | g | m | h

Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

j k | e n | c o | d l | b i | a f | g | m | h

Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

j k | e n | c o | d l | b i | a f | g | m | h

Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

j k | e n | c o | d l | b i | a f | g | m | h

j k e n | c o d l | b i a f | g m h

Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

j k | e n | c o | d l | b i | a f | g | m | h

e j k n | c o d l | b i a f | g m h

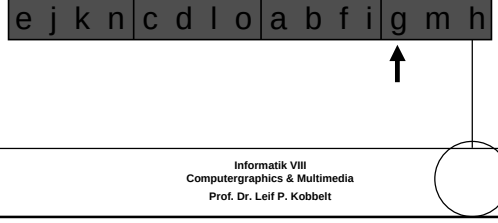
Merge-Sort (iterativ)

k | j | e | n | o | c | d | l | b | i | f | a | g | m | h

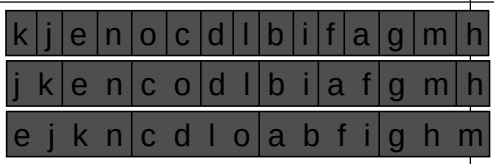
j k | e n | c o | d l | b i | a f | g | m | h

e j k n | c d l o | b i a f | g m h

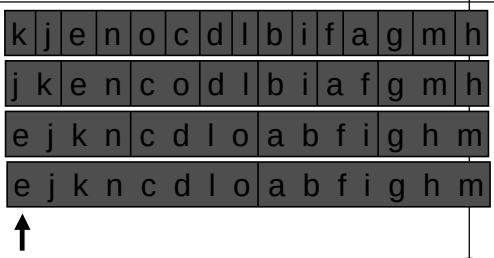
Merge-Sort (iterativ)



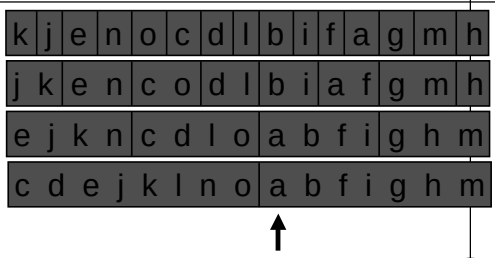
Merge-Sort (iterativ)



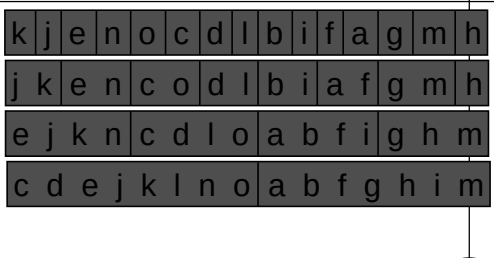
Merge-Sort (iterativ)



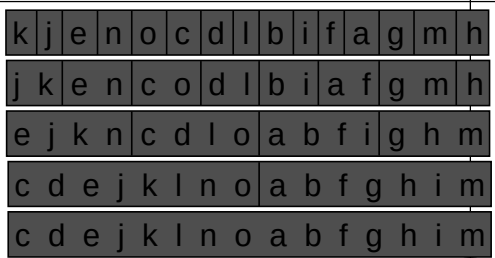
Merge-Sort (iterativ)



Merge-Sort (iterativ)



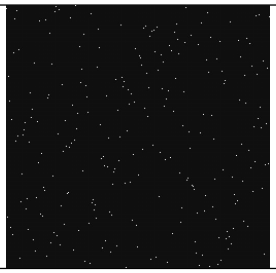
Merge-Sort (iterativ)



Merge-Sort (iterativ)

k	j	e	n	o	c	d	l	b	i	f	a	g	m	h
j	k	e	n	c	o	d	l	b	i	a	f	g	m	h
e	j	k	n	c	d	l	o	a	b	f	i	g	h	m
c	d	e	j	k	l	n	o	a	b	f	g	h	i	m
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o

Merge-Sort (breadth-first)



Heap-Sort

- Idee
  - Bei Selection-Sort entsteht der $O(n^2)$-Aufwand durch die lineare Suche nach dem kleinsten (oder größten) Element in der Rest-Liste.
  - Mit einer Prioritätsschlange (Heap) kann man das kleinste Element schneller finden (O(log(n))-Aufwand zum Wiederherstellen der Heap-Bedingung)

Heap-Definition (*reprise*)

- (Links-)vollständiger Binärbaum in Array-Darstellung
  - Vaterknoten A[i] hat die Nachfolger A[2i] und A[2i+1]
- Heap-Bedingung
  - Node $\geq$ max(Left-Subtree) $\wedge$ Node $\geq$ max(Right-Subtree)
  - $A[i] \geq A[2i] \ \wedge \ A[i] \geq A[2i+1]$

Heap-Definition (*reprise*)

- (Links-)vollständiger Binärbaum in Array-Darstellung
  - Vaterknoten A[i] hat die Nachfolger A[2i] und A[2i+1]
- Heap-Bedingung
  - Node $\geq$ min(Left-Subtree) $\wedge$ Node $\geq$ min(Right-Subtree)
  - $A[\lfloor i/2 \rfloor] \geq A[i]$

Heap-Definition (*reprise*)

- Ein Array hat die *Heap-Eigenschaft* ab Index p, wenn $A[\lfloor i/2 \rfloor] \geq A[i]$ ab Index 2p gilt (weil dann $\lfloor i/2 \rfloor = p$).
- Insbesondere: jedes beliebige Array A[1 ... n] hat die Heap-Eigenschaft ab Index $\lfloor n/2 \rfloor + 1$.

Heap-Sort

- Konvertiere ein beliebiges Array in einen Heap ($A[\lfloor i/2 \rfloor] \geq A[i]$)
- Entferne sukzessive das Top-Element und stelle die Heap-Eigenschaft wieder her (vgl. **Deq()-**Operation für Prioritäts-schlangen ... *Abschnitt (1.4)*)
- Heap liefert sortierte Elemente.

Heap-Sort

- Sei A[1 ... n] ein Heap mit Top-Element A[1]
- Nach dem Entfernen des Top-Elementes wird A[n] nach A[1] kopiert und durch „Versickern“ die Heap-Eigenschaft wieder hergestellt.
- Danach benutzt der Heap noch die Array-Elemente A[1 ... n-1].
- A[n] ist also frei und das entfernte Top-Element kann dort abgelegt werden.

Implementierung

- ConvertToHeap(A,n)
  for ($i = \lfloor n/2 \rfloor$; $i \geq 1$; i--)
    Sink(A,n,i);
- Wenn die Schleife bis i=p durchgelaufen ist, hat das Array A ab Index p die Heap-Eigenschaft.

Implementierung

- Sink(A,n,i)

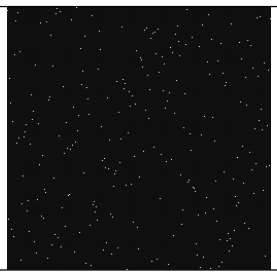
```
while ( i ≤ ⌊n/2⌋ )
    j = 2*i;
    if (j < n) ∧ (A[j+1] > A[j])  j++;
    if (A[j] > A[i])
      Swap(A[j],A[i])
      i = j;
    else
      i = n;
```


Implementierung

- HeapSort(A,n)

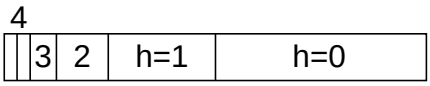
```
ConvertToHeap(A,n);
for (i = n ; i ≥ 1 ; i--)
    Swap(A[1],A[i]);
    Sink(A,i-1,1);
```


Heap-Sort



Aufwandsabschätzung

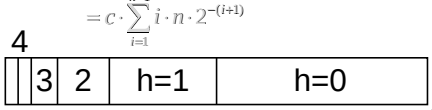
- ConvertToHeap()
  - Sink()-Operation = O(Höhe des Heap)
  - Sei n = 2^k -1...
  - $T_1(n) \leq c * (1*n/4 + 2*n/8 + ... + (\log(n)-1)*n/n)$



Aufwandsabschätzung

- ConvertToHeap()
  - Sink()-Operation = O(Höhe des Heap)
  - Sei n = 2^k -1...
  - $T_1(n) \leq c * (1*n/4 + 2*n/8 + ... + (\log(n)-1)*n/n)$
    $= c * (1*n/4 + ... + i*n/2^{i+1} + ... + (k-1)*1)$

$$= c \cdot \sum_{i=1}^{k-1} i \cdot n \cdot 2^{-(i+1)}$$



Aufwandsabschätzung

- ConvertToHeap()

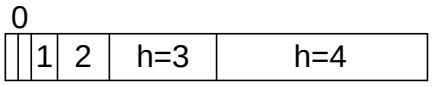
$$T_1(n) \leq c \cdot \sum_{i=1}^{k-1} i \cdot n \cdot 2^{-(i+1)} = 2cn \sum_{i=1}^{k-1} i \cdot 2^{-i}$$

$$= 2cn \cdot \left(2 - (k+1) \cdot 2^{1-k}\right)$$

$$= O(n)$$

Aufwandsabschätzung

- Selection-Sort Phase
  - Kosten für die Wiederherstellung der Heap-Eigenschaft = O(Höhe des restlichen Heaps)
  - Sei n = 2^k -1 ...
  - $T_2(n) \leq c * ((k-1)*n/2 + (k-2)*n/4 + ... + 0*n/n)$



Aufwandsabschätzung

- Selection-Sort Phase
  - Kosten für die Wiederherstellung der Heap-Eigenschaft = O(Höhe des restlichen Heaps)
  - Sei n = 2^k -1 ...
  - $T_2(n) \leq c * ((k-1)*n/2 + (k-2)*n/4 + ... + 0*n/n)$
    $= c * ((k-1)*n/2 + ... + (k-i)*n/2^i + ...)$

$$= c \cdot \sum_{i=1}^{k-1} (k-i) \cdot n \cdot 2^{-i}$$

Aufwandsabschätzung

$$T_2(n) = c \cdot \sum_{i=1}^{k-1} (k-i) \cdot n \cdot 2^{-i} = c \cdot \sum_{i=1}^{k-1} i \cdot n \cdot 2^{i-k}$$

$$= c \cdot n \cdot 2^{-k} \cdot \sum_{i=1}^{k-1} i \cdot 2^i$$

$$= c \cdot n \cdot 2^{-k} \cdot \left(2^k \cdot (k-2) + 2\right)$$

$$= c \cdot n \cdot \left(k - 2 + 2^{1-k}\right) = O(n \log(n))$$

Aufwandsabschätzung

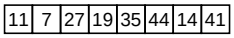
- Insgesamt ... worst / average case

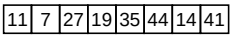
  $T(n) = T_1(n) + T_2(n)$
  $= O(n) + O(n \log(n))$
  $= O(n \log(n))$

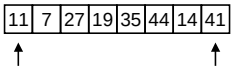
- Best case: O(n) ... alle Elemente gleich
- Vorsortierung wird nicht ausgenutzt

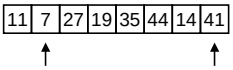
Vergleich

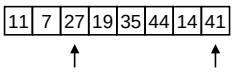
	Average Case	Worst Case
• Quick-Sort:	O(n log(n))	$O(n^2)$
• Merge-Sort: (doppelter Speicher)	O(n log(n))	O(n log(n))
• Heap-Sort:	O(n log(n))	O(n log(n))

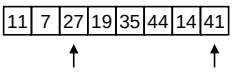












Wdh: Quicksort

11	7	27	19	35	44	14	41

Wdh: Quicksort

11	7	27	19	35	44	14	41

Pivot: 20

Wdh: Quicksort

11	7	27	19	35	44	14	41
↑							↑

Pivot: 20

Wdh: Quicksort

11	7	27	19	35	44	14	41
	↑						↑

Pivot: 20

Wdh: Quicksort

11	7	27	19	35	44	14	41
		↑					↑

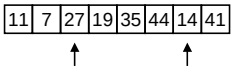
Pivot: 20

Wdh: Quicksort

11	7	27	19	35	44	14	41
		↑					↑

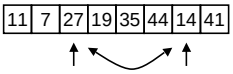
Pivot: 20

Wdh: Quicksort



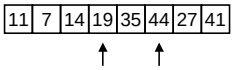
Pivot: 20

Wdh: Quicksort



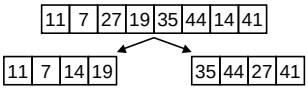
Pivot: 20

Wdh: Quicksort

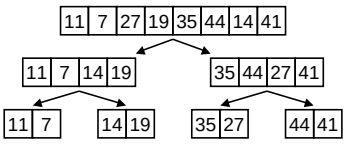


Pivot: 20

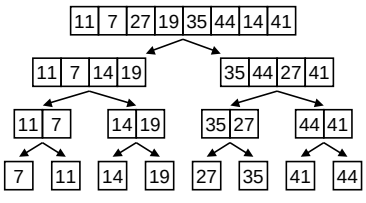
Wdh: Quicksort



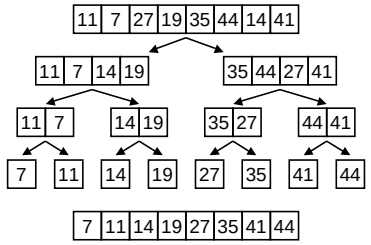
Wdh: Quicksort



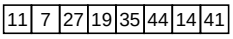
Wdh: Quicksort



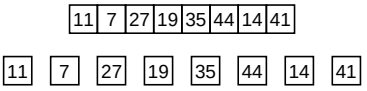
Wdh: Quicksort



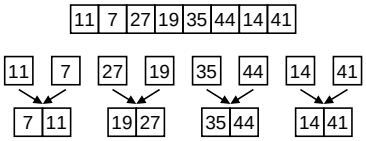
Wdh: Mergesort



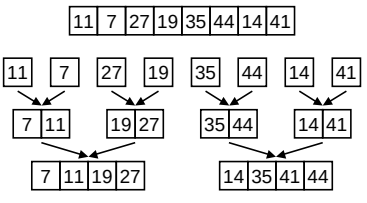
Wdh: Mergesort



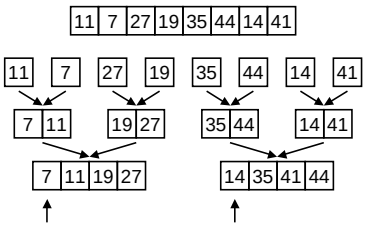
Wdh: Mergesort



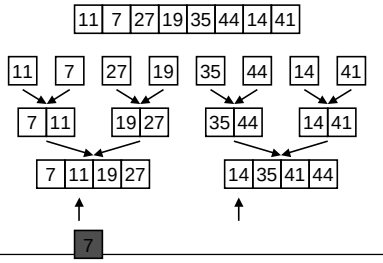
Wdh: Mergesort



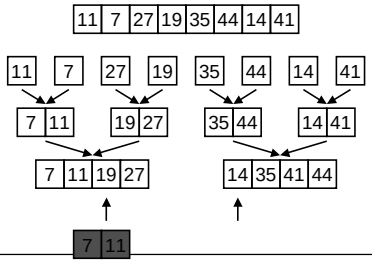
Wdh: Mergesort



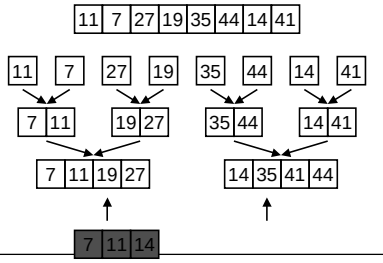
Wdh: Mergesort



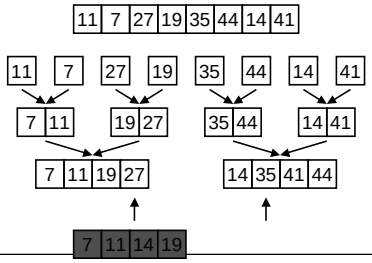
Wdh: Mergesort



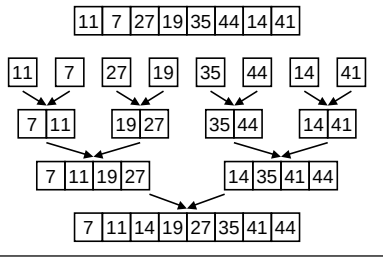
Wdh: Mergesort



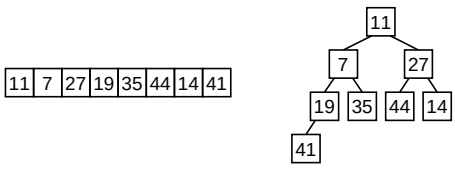
Wdh: Mergesort

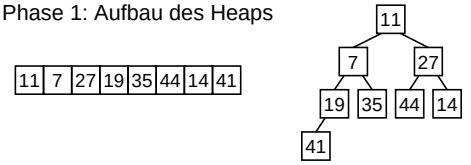


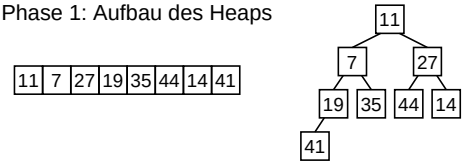
Wdh: Mergesort

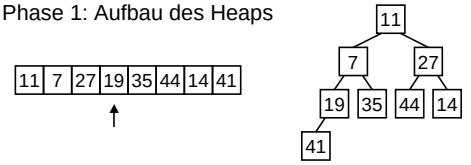


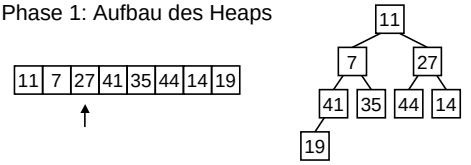
Wdh: Heapsort

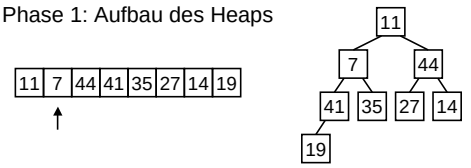


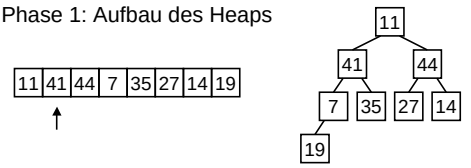












Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	7	27	19	35	44	14	41

Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	7	27	19	35	44	14	41

Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	7	27	19	35	44	14	41
			↑				

Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	7	27	41	35	44	14	19
		↑					

Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	7	44	41	35	27	14	19
	↑						

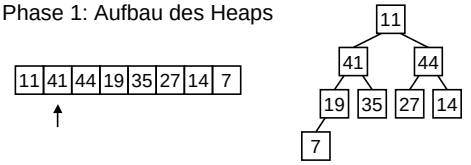
Wdh: Heapsort

Phase 1: Aufbau des Heaps

11	41	44	7	35	27	14	19
	↑						

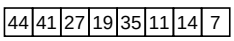
Wdh: Heapsort

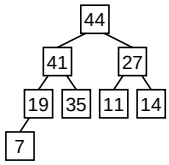
Phase 1: Aufbau des Heaps



Wdh: Heapsort

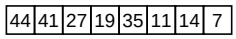
Phase 1: Aufbau des Heaps

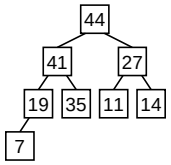




Wdh: Heapsort

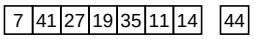
Phase 2: Selection-Sort

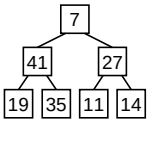




Wdh: Heapsort

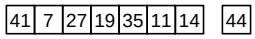
Phase 2: Selection-Sort

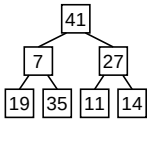




Wdh: Heapsort

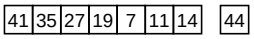
Phase 2: Selection-Sort

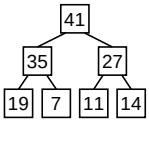




Wdh: Heapsort

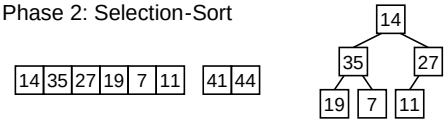
Phase 2: Selection-Sort





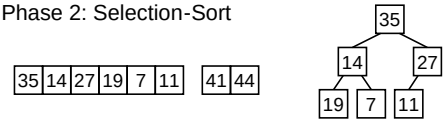
Wdh: Heapsort

Phase 2: Selection-Sort



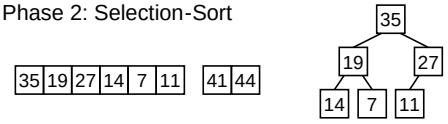
Wdh: Heapsort

Phase 2: Selection-Sort



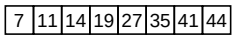
Wdh: Heapsort

Phase 2: Selection-Sort



Wdh: Heapsort

Phase 2: Selection-Sort



Vergleich

	Average Case	Worst Case
• Quick-Sort:	O(n log(n))	$O(n^2)$
• Merge-Sort: (doppelter Speicher)	O(n log(n))	O(n log(n))
• Heap-Sort:	O(n log(n))	O(n log(n))

Optimaler Sortieralgorithmus

- Was ist die theoretische Untergrenze für die Komplexität eines Sortieralgorithmus?
- Wann können wir mit einem Sortier-algorithmus zufrieden sein?

Optimaler Sortieralgorithmus

- Reprise: Sortierproblem
  - gegeben:
    - Objektmenge R
    - Ordnung $\leq$ auf R^2
    - $R_1, R_2, ..., R_n$
  - gesucht: Permutation π mit

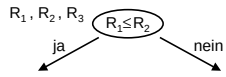
$$R_{\pi(1)} \leq R_{\pi(2)} \leq \cdots \leq R_{\pi(n)}$$

- sogenanntes ***„Sortieren durch Vergleichen“***

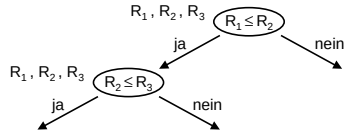
Optimaler Sortieralgorithmus

- Der Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus kann durch einen Entscheidungsbaum dargestellt werden
  - innere Knoten = Vergleich
  - Blätter = sortierende Permutation

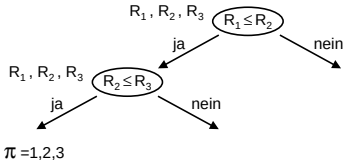
Insertion-Sort Entscheidungsbaum



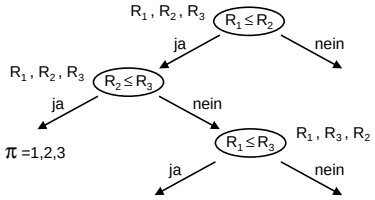
Insertion-Sort Entscheidungsbaum



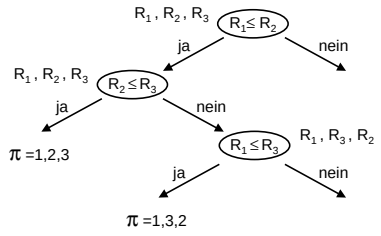
Insertion-Sort Entscheidungsbaum



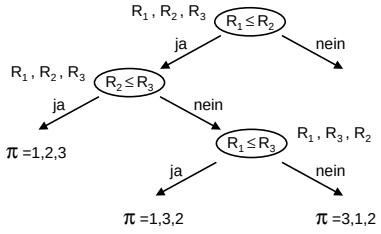
Insertion-Sort Entscheidungsbaum



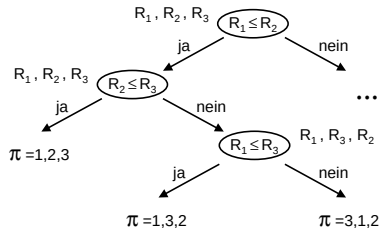
Insertion-Sort Entscheidungsbaum



Insertion-Sort Entscheidungsbaum



Insertion-Sort Entscheidungsbaum



Optimaler Sortieralgorithmus

1. Zahl der möglichen Permutationen: n!
   ⇒ jeder Entscheidungsbaum hat mindestens n! Blätter
2. maximale Zahl von Blättern in einem Binärbaum der Höhe h: 2^h
   ⇒ jeder Entscheidungsbaum hat höchstens 2^h Blätter

also gilt: $2^h \geq n!$

Optimaler Sortieralgorithmus

Möglichkeit 1:

$$2^h \geq n! \Leftrightarrow h \geq \mathrm{ld}(n!)$$
$$= \mathrm{ld}\left(\prod_{i=1}^{n} i\right)$$
$$= \sum_{i=1}^{n} \mathrm{ld}(i)$$
$$\geq \sum_{i=1}^{\lceil n/2 \rceil - 1} \mathrm{ld}(i) + \sum_{i=\lceil n/2 \rceil}^{n} \mathrm{ld}(i)$$

Optimaler Sortieralgorithmus

Möglichkeit 1:

$$2^h \geq n! \Leftrightarrow h \geq \cdots$$
$$\geq \sum_{i=1}^{\lceil n/2 \rceil - 1} \mathrm{ld}(i) + \sum_{i=\lceil n/2 \rceil}^{n} \mathrm{ld}(i)$$
$$\geq 0 + \left\lceil \frac{n}{2} \right\rceil \mathrm{ld}\left(\left\lceil \frac{n}{2} \right\rceil\right)$$
$$\geq \frac{n}{2} \mathrm{ld}\left(\frac{n}{2}\right)$$

Optimaler Sortieralgorithmus

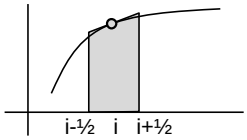
Möglichkeit 1:

$$2^h \geq n! \Leftrightarrow h \geq \cdots$$
$$\geq \frac{n}{2} \operatorname{ld}\left(\frac{n}{2}\right)$$
$$\geq \frac{n}{2}\left(\operatorname{ld}(n) - \operatorname{ld}(2)\right)$$
$$\geq \frac{1}{2} n \operatorname{ld}(n) - \frac{1}{2} n$$
$$= \Omega(n \log n)$$

Optimaler Sortieralgorithmus

Möglichkeit 2: Integralmethode

$$h \geq \operatorname{ld}(n!) = \sum_{i=1}^{n} \operatorname{ld}(i)$$
$$= c \sum_{i=1}^{n} \ln(i)$$
$$\geq c \int_{1/2}^{n+1/2} \ln(x)\,dx$$
$$= c\left(x \ln x - x\Big|_{1/2}^{n+1/2}\right)$$
$$= \Omega(n \log n)$$



Optimaler Sortieralgorithmus

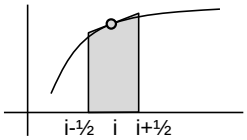
$$\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log(i)$$

- Abschätzung mit der Integralmethode

$$\sum_{i=1}^{n} \log(i) = O\left(\sum_{i=1}^{n} \ln(i)\right)$$

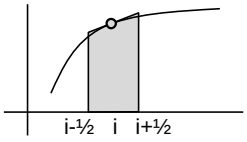
Optimaler Sortieralgorithmus

$$\sum_{i=1}^{n} \ln(i) \geq \int_{1/2}^{n+1/2} \ln(x)\,dx = x \ln(x) - x\Big|_{1/2}^{n+1/2}$$



Optimaler Sortieralgorithmus

$$x \ln(x) - x\Big|_{1/2}^{n+1/2} = \Omega(n \log(n))$$



Optimaler Sortieralgorithmus

$$x \ln(x) - x\Big|_{1/2}^{n+1/2} = \Omega(n \log(n))$$

Ω(n log(n)) ist eine untere Schranke für die Komplexität des **allgemeinen** Sortierproblems

Optimaler Sortieralgorithmus

- Fazit: jeder nur auf Vergleichen basierende Sortieralgorithmus hat einen Mindestaufwand von

$$T(n) = \Omega(n \log n)$$

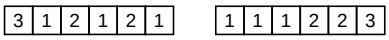
- *Mergesort* und *Heapsort* sind **optimale** Sortieralgorithmen

Typen von Sortieralgorithmen

- Einfache
  - Selection-Sort, Bubble-Sort, Insertion-Sort
- Höhere
  - Quick-Sort, Merge-Sort, Heap-Sort
- Spezielle
  - Counting-Sort, Radix-Sort, Bucket-Sort

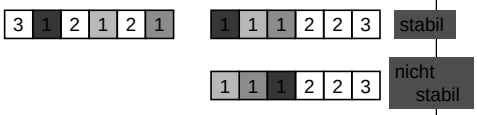
Stabilität von Sortieralgorithmen

- Ein Sortieralgorithmus heißt **stabil**, wenn sich die relative Reihenfolge von gleichen Elementen während des Sortierens nicht ändert.
- Beispiel:



Stabilität von Sortieralgorithmen

- Ein Sortieralgorithmus heißt **stabil**, wenn sich die relative Reihenfolge von gleichen Elementen während des Sortierens nicht ändert.
- Beispiel:



Counting-Sort

- Annahme: $R_1, R_2, ..., R_n \in \{1,...,k\}$
- Idee: Bestimme zu jedem R_i die Zahl der Elemente $\leq R_i$ und sortiere R_i an die entsprechende Stelle

Counting-Sort: Algorithmus

```
CountingSort(A,B)
  for (i = 1; i <= k; ++i)
    C[i]=0;
  for (i = 1; i <= n; ++i)
    ++C[A[i]];
  for (i = 2; i <= k; ++i)
    C[i] += C[i-1];
  for (i = n; i >= 1; --i)
    B[C[A[i]]] = A[i];
    C[A[i]]--;
```

Counting-Sort: Algorithmus

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

→ Eingabe: A[1..n]
Ausgabe: B[1..n]

Counting-Sort: Algorithmus

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

→ C ist mit 0en initialisiert

Counting-Sort: Algorithmus

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

→ C[j] enthält die Anzahl der Elemente =j

Counting-Sort: Algorithmus

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

→ C[j] enthält die Anzahl der Elemente ≤ j

Counting-Sort: Algorithmus

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

→ B[j] enthält das j-te Element

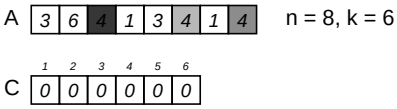
Counting-Sort: Beispiel

A

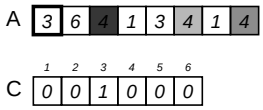
3	6	4	1	3	4	1	4

n = 8, k = 6

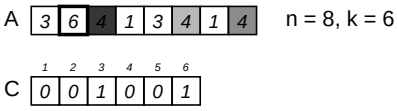
Counting-Sort: Beispiel



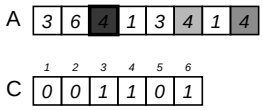
Counting-Sort: Beispiel



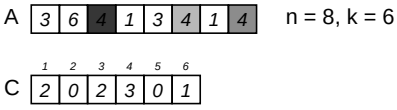
Counting-Sort: Beispiel



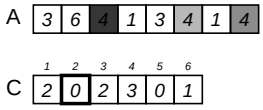
Counting-Sort: Beispiel



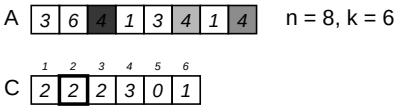
Counting-Sort: Beispiel



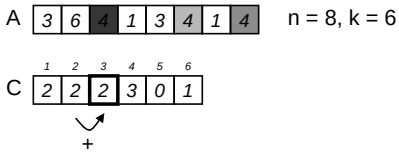
Counting-Sort: Beispiel



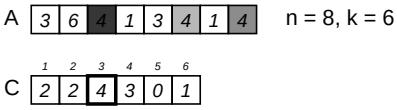
Counting-Sort: Beispiel



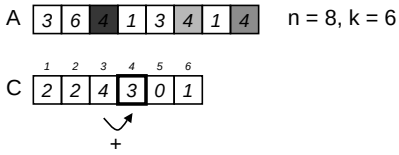
Counting-Sort: Beispiel



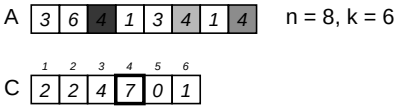
Counting-Sort: Beispiel



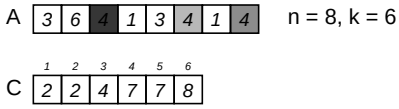
Counting-Sort: Beispiel



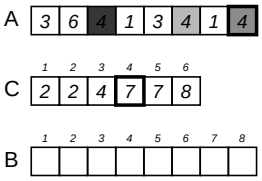
Counting-Sort: Beispiel



Counting-Sort: Beispiel

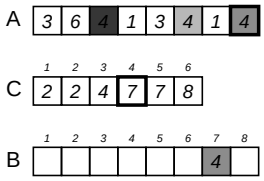


Counting-Sort: Beispiel



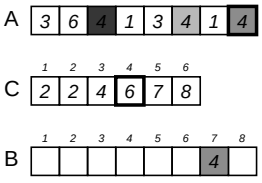
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



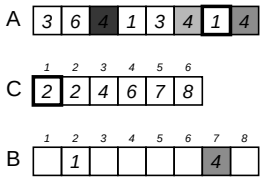
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



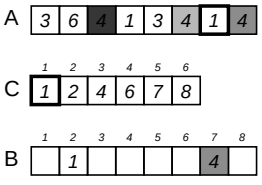
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



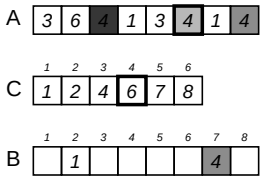
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



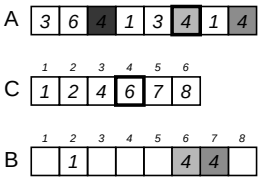
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



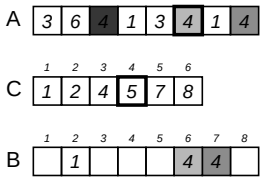
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



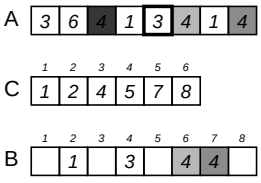
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



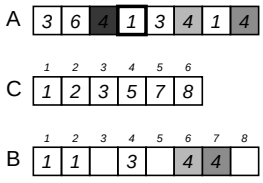
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



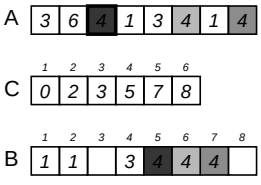
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



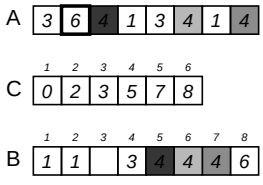
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



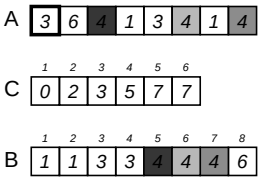
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

O(k)

Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

O(k)

O(n)

Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

O(k)

O(n)

O(k)

Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

O(k)

O(n)

O(k)

O(n)

Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)
      C[i]=0;
   for (i = 1; i <= n; ++i)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)
      B[C[A[i]]] = A[i];
      C[A[i]]--;
```

O(k)

O(n)

O(k)

O(n)

O(n+k)

Counting-Sort: Aufwand

- Counting-Sort ist nur sinnvoll, wenn k = O(n) und damit

  T(n) = O(n)

- Counting-Sort verwendet ***keine*** Vergleiche
- Counting-Sort ist stabil

Radix-Sort

- Annahme:

  $R_1, R_2, ..., R_n \in \{0,...,k^d-1\}$

- „d-stellige Zahlen zur Basis k“
- „Wörter der Länge d aus einem Alphabet der Größe k“

Radix-Sort: Algorithmus

```
RadixSort(A)
  for (i = 0; i < d; ++i)
    „sortiere A stabil nach $k^i$-wertiger Stelle“
```

Radix-Sort: Beispiel

F C T
H L P
N L P
R F T
H F N
P C A
F L L

Radix-Sort: Beispiel

F C T
H L P
N L P
R F T
H F N
P C A
F L L

Radix-Sort: Beispiel

F C T		P C A
H L P		F L L
N L P		H F N
R F T	⇨	H L P
H F N		N L P
P C A		F C T
F L L		R F T

Radix-Sort: Beispiel

F C T	P C A
H L P	F L L
N L P	H F N
R F T ⇨	H L P
H F N	N L P
P C A	F C T
F L L	R F T

Radix-Sort: Beispiel

F C T	P C A
H L P	F L L
N L P	H F N
R F T ⇨	H L P
H F N	N L P
P C A	F C T
F L L	R F T

Radix-Sort: Beispiel

F C T	P C A	P C A
H L P	F L L	F C T
N L P	H F N	H F N
R F T ⇨	H L P ⇨	R F T
H F N	N L P	F L L
P C A	F C T	H L P
F L L	R F T	N L P

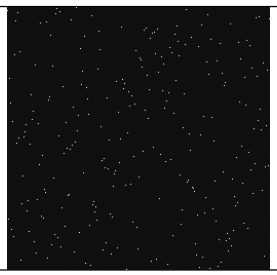
Radix-Sort: Beispiel

F C T	P C A	P C A
H L P	F L L	F C T
N L P	H F N	H F N
R F T ⇨	H L P ⇨	R F T
H F N	N L P	F L L
P C A	F C T	H L P
F L L	R F T	N L P

Radix-Sort: Beispiel

F C T	P C A	P C A	F C T
H L P	F L L	F C T	F L L
N L P	H F N	H F N	H F N
R F T ⇨	H L P ⇨	R F T ⇨	H L P
H F N	N L P	F L L	N L P
P C A	F C T	H L P	P C A
F L L	R F T	N L P	R F T

Radix-Sort: Beispiel



Radix-Sort: Aufwand

RadixSort(A)

for (i = 0; i < d; ++i)

„sortiere A stabil nach k^i-wertiger Stelle“

Radix-Sort: Aufwand

RadixSort(A)

for (i = 0; i < d; ++i)

„sortiere A stabil mit Counting-Sort nach k^i-wertiger Stelle“

Radix-Sort: Aufwand

RadixSort(A)

for (i = 0; i < d; ++i)

„sortiere A stabil mit Counting-Sort nach k^i-wertiger Stelle“

$T(k,d,n)=O(d(n+k))$

Radix-Sort: Aufwand

- Beachte: Ist k gegeben, so benötigt man zur Darstellung von n verschiedenen Elementen mindestens $d \geq \log_k(n)$ Stellen

$\Rightarrow T(n) = \mathbf{\Omega}(n \log n)$

Bucket-Sort

- Annahme:
  $R_1, R_2, ..., R_n$ gleichverteilt aus [0,1)
- Idee:
  1. Unterteile [0,1) in n „Buckets“
     [0,1/n), [1/n,2/n), ..., [(n-1)/n,1)
  2. Füge die R_i in die Buckets ein (erwartet: ein Element pro Bucket)
  3. Sortiere die Buckets
  4. Hänge die Buckets aneinander

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
  for (i = 0; i < n; ++i)
    B[i] = [];
  for (i = 0; i < n; ++i)
    B[ ⌊nA[i]⌋ ].push(A[i]);
  for (i = 0; i < n; ++i)
    sort(B[i]);
  R = [];
  for (i = 0; i < n; ++i)
    for (j = 0; j < B[i].size(); ++j)
      R.push(B[i][j]);
```

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

Eingabe: A[0..n-1]
Ausgabe: R[0..n-1]

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

alle Buckets leer

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

Buckets gefüllt

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

Buckets gefüllt

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)          0 ≤   A[i]   < 1
      B[i] = [];                ⇒   0 ≤   nA[i]  < n
   for (i = 0; i < n; ++i)      ⇒   0 ≤  ⌊nA[i]⌋ < n
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

Buckets gefüllt

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

Buckets sortiert

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

→ Buckets zusammengehängt

Prof. Dr. Leif P. Kobbelt

Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6)
6 [0.6,0.7)
7 [0.7,0.8)
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8)
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

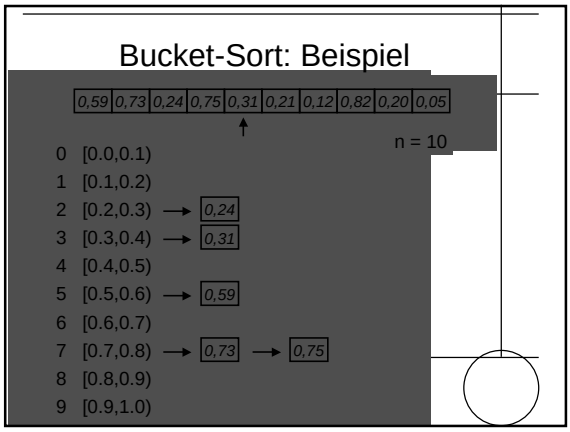
0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0,24
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73
8 [0.8,0.9)
9 [0.9,1.0)

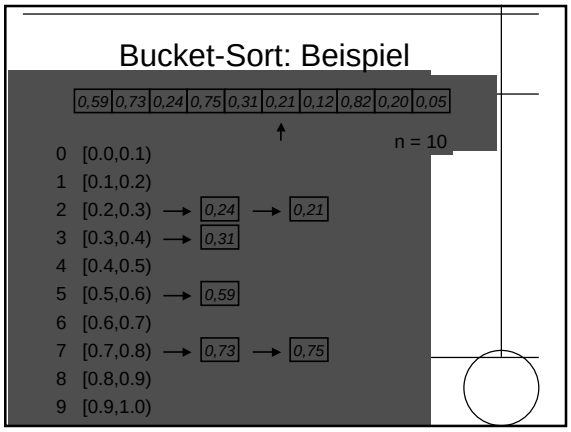
Bucket-Sort: Beispiel

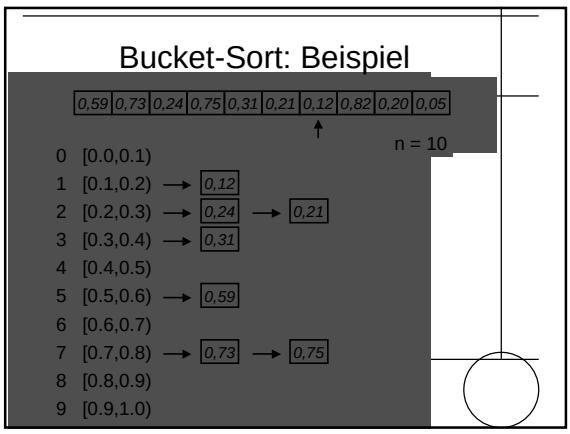
0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

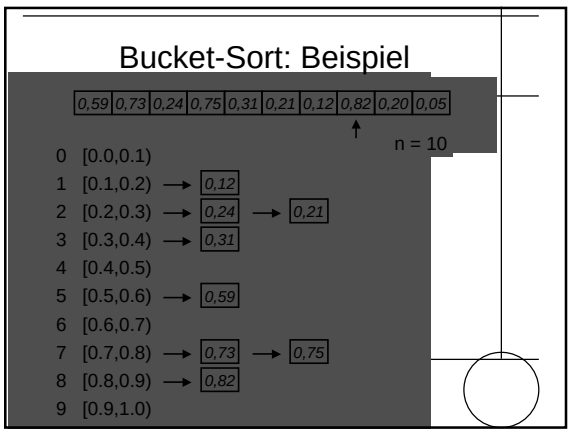
n = 10

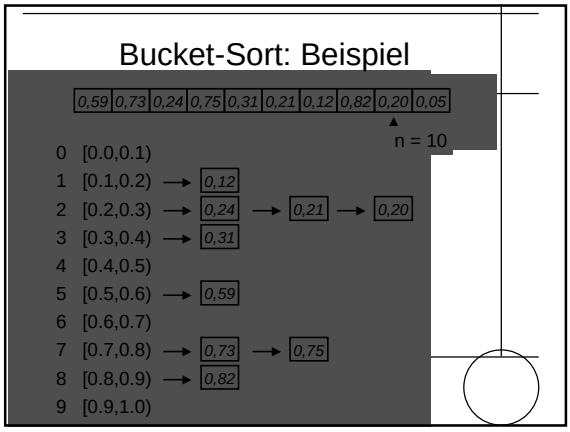
0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0,24
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9)
9 [0.9,1.0)

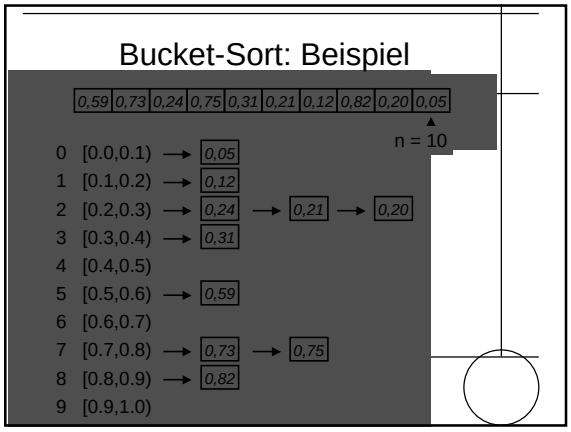

Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0,24
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9)
9 [0.9,1.0)

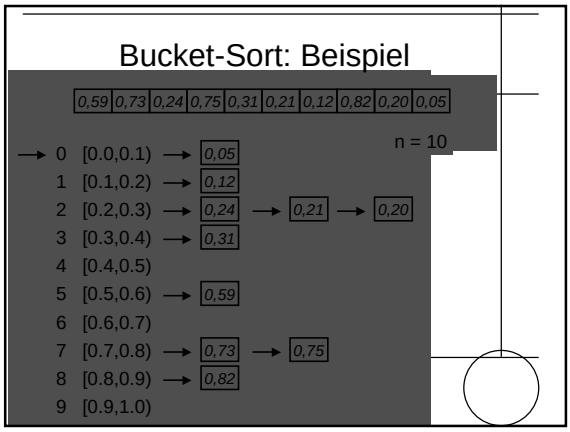

Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0,24 → 0,21
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9)
9 [0.9,1.0)


Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1)
1 [0.1,0.2) → 0,12
2 [0.2,0.3) → 0,24 → 0,21
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9)
9 [0.9,1.0)


Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1)
1 [0.1,0.2) → 0,12
2 [0.2,0.3) → 0,24 → 0,21
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9) → 0,82
9 [0.9,1.0)


Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1)
1 [0.1,0.2) → 0,12
2 [0.2,0.3) → 0,24 → 0,21 → 0,20
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9) → 0,82
9 [0.9,1.0)


Bucket-Sort: Beispiel
0,59 0,73 0,24 0,75 0,31 0,21 0,12 0,82 0,20 0,05
n = 10
0 [0.0,0.1) → 0,05
1 [0.1,0.2) → 0,12
2 [0.2,0.3) → 0,24 → 0,21 → 0,20
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9) → 0,82
9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

→ 0 [0.0,0.1) → 0,05

1 [0.1,0.2) → 0,12

2 [0.2,0.3) → 0,24 → 0,21 → 0,20

3 [0.3,0.4) → 0,31

4 [0.4,0.5)

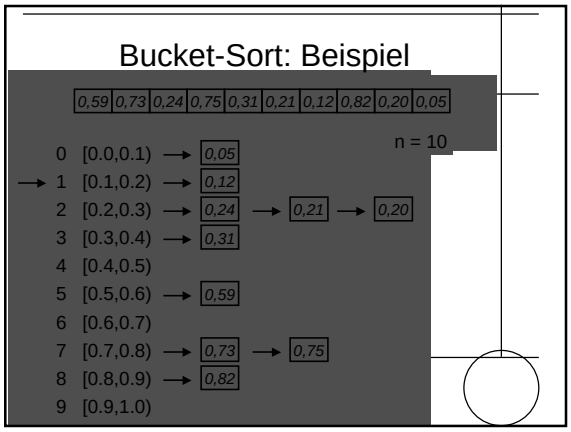
5 [0.5,0.6) → 0,59

6 [0.6,0.7)

7 [0.7,0.8) → 0,73 → 0,75

8 [0.8,0.9) → 0,82

9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1) → 0,05

→ 1 [0.1,0.2) → 0,12

2 [0.2,0.3) → 0,24 → 0,21 → 0,20

3 [0.3,0.4) → 0,31

4 [0.4,0.5)

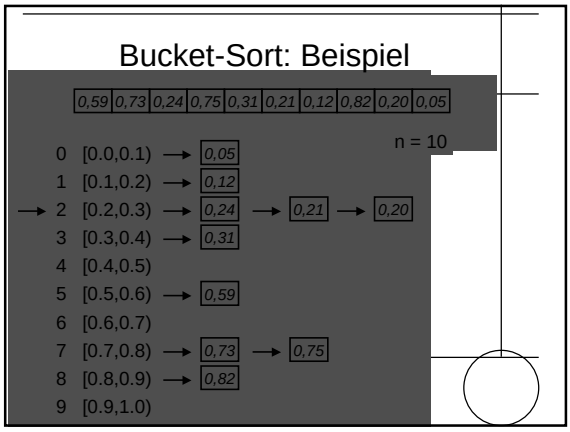
5 [0.5,0.6) → 0,59

6 [0.6,0.7)

7 [0.7,0.8) → 0,73 → 0,75

8 [0.8,0.9) → 0,82

9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1) → 0,05

1 [0.1,0.2) → 0,12

→ 2 [0.2,0.3) → 0,24 → 0,21 → 0,20

3 [0.3,0.4) → 0,31

4 [0.4,0.5)

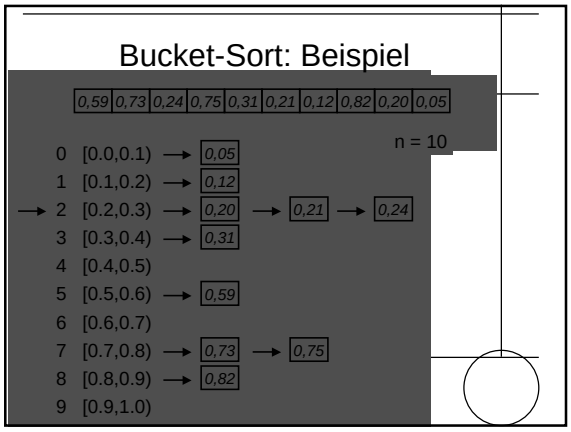
5 [0.5,0.6) → 0,59

6 [0.6,0.7)

7 [0.7,0.8) → 0,73 → 0,75

8 [0.8,0.9) → 0,82

9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1) → 0,05

1 [0.1,0.2) → 0,12

→ 2 [0.2,0.3) → 0,20 → 0,21 → 0,24

3 [0.3,0.4) → 0,31

4 [0.4,0.5)

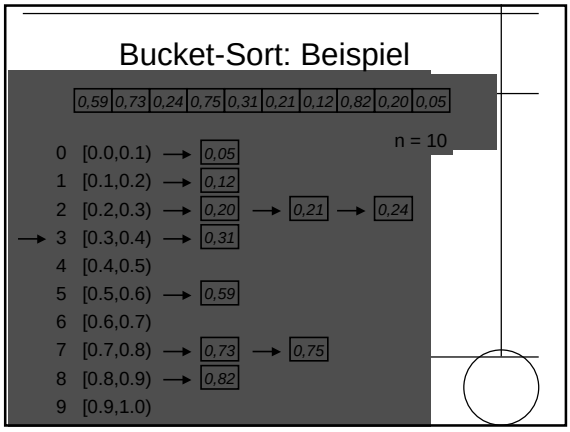
5 [0.5,0.6) → 0,59

6 [0.6,0.7)

7 [0.7,0.8) → 0,73 → 0,75

8 [0.8,0.9) → 0,82

9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1) → 0,05

1 [0.1,0.2) → 0,12

2 [0.2,0.3) → 0,20 → 0,21 → 0,24

→ 3 [0.3,0.4) → 0,31

4 [0.4,0.5)

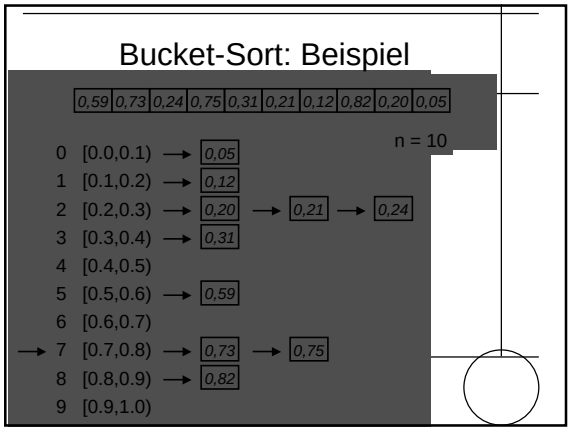
5 [0.5,0.6) → 0,59

6 [0.6,0.7)

7 [0.7,0.8) → 0,73 → 0,75

8 [0.8,0.9) → 0,82

9 [0.9,1.0)



Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1) → 0,05

1 [0.1,0.2) → 0,12

2 [0.2,0.3) → 0,20 → 0,21 → 0,24

3 [0.3,0.4) → 0,31

4 [0.4,0.5)

5 [0.5,0.6) → 0,59

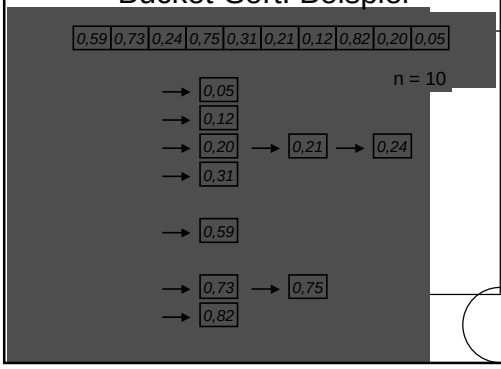
6 [0.6,0.7)

→ 7 [0.7,0.8) → 0,73 → 0,75

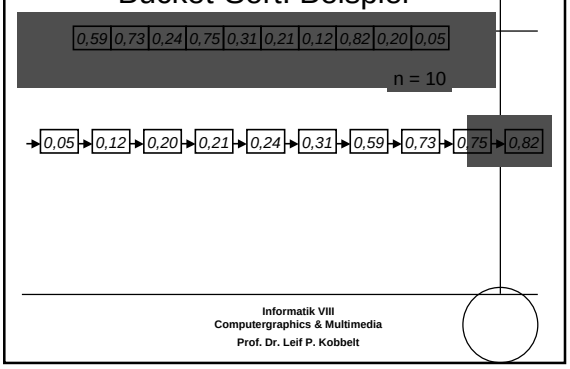
8 [0.8,0.9) → 0,82

9 [0.9,1.0)

Bucket-Sort: Beispiel



Bucket-Sort: Beispiel



Bucket-Sort: Analyse

Sei X eine diskrete Zufallsvariable, d.h.

$$X \in \{ x_1, x_2, x_3, \cdots \}$$

Erwartungswert:

$$E(X) = \mu = \sum_{i=1}^{\infty} x_i \cdot P(X = x_i)$$

Varianz:

$$V(X) = \sigma^2 = E\left((X-\mu)^2\right) = E\left(X^2\right) - E^2(X)$$

Bucket-Sort: Analyse

- Sei n_i die Zahl der Elemente in Bucket i
- Wahrscheinlichkeit, dass ein Element in Bucket i fällt ist $p=1/n$
- Wahrscheinlichkeit, dass genau k Elemente in Bucket i fallen ist

$$P(n_i = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

„Binomialverteilung“

Bucket-Sort: Analyse

Erwartungswert:

$$E(n_i) = np = 1$$

Varianz:

$$V(n_i) = np(1-p) = 1 - \frac{1}{n}$$

Sortieraufwand, z.B. für Insertion-Sort:

$$T(l) = O\left(l^2\right) \Rightarrow \exists c > 0 : T(l) \le cl^2$$

Bucket-Sort: Analyse

Erwarteter Sortieraufwand für Bucket i:

$$\begin{aligned} E(T(n_i)) &= \sum_{k=0}^{n} T(k) P(n_i = k) \\ &\le \sum_{k=0}^{n} ck^2 P(n_i = k) \\ &\le c \sum_{k=0}^{n} k^2 P(n_i = k) \\ &= cE\left(n_i^2\right) \end{aligned}$$

Bucket-Sort: Analyse

Erwarteter Sortieraufwand für Bucket i:

$$\begin{aligned} E(T(n_i)) &\leq \cdots \\ &\leq cE(n_i^2) \\ &= c\left(V(n_i) + E^2(n_i)\right) \\ &= c\left(1 - \frac{1}{n} + 1\right) \\ &\leq 2c = O(1) \end{aligned}$$

Bucket-Sort: Analyse

Erwarteter Sortieraufwand für alle Buckets:

$$\begin{aligned} E\left(\sum_{i=0}^{n-1} T(n_i)\right) &= \sum_{i=0}^{n-1} \underbrace{E(T(n_i))}_{=O(1)} \\ &= O(n) \end{aligned}$$

Bucket-Sort: Analyse

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

O(n)

Bucket-Sort: Analyse

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

O(n)

O(n)

Bucket-Sort: Analyse

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

O(n)

O(n)

O(n) (erwartet!)

Bucket-Sort: Analyse

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)
      B[i] = [];
   for (i = 0; i < n; ++i)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
```

O(n)

O(n)

O(n) (erwartet!)

O(n) (!)

Bucket-Sort: Analyse

```
BucketSort(A,R)
   for (i = 0; i < n; ++i)                          O(n)
      B[i] = [];
   for (i = 0; i < n; ++i)                          O(n)
      B[⌊nA[i]⌋].push(A[i]);
   for (i = 0; i < n; ++i)                          O(n)  (erwartet!)
      sort(B[i]);
   R = [];
   for (i = 0; i < n; ++i)                          O(n)  (!)
      for (j = 0; j < B[i].size(); ++j)
         R.push(B[i][j]);
                                                    ---------------
                                                    O(n)  (erwartet!)
```

Spezielle Sortierverfahren

- Counting-Sort: O(n+k)
- Radix-Sort: O(d(n+k))
- Bucket-Sort: O(n) erwartet

Vergleich

	Average Case	Worst Case
• Quick-Sort:	O(n log(n))	$O(n^2)$
• Merge-Sort: (doppelter Speicher)	O(n log(n))	O(n log(n))
• Heap-Sort:	O(n log(n))	O(n log(n))

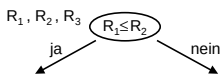
Optimaler Sortieralgorithmus

- Was ist die theoretische Untergrenze für die Komplexität eines Sortieralgorithmus?
- Wann können wir mit einem Sortier-algorithmus zufrieden sein?

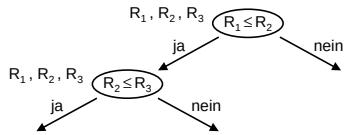
Optimaler Sortieralgorithmus

- Der Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus kann durch einen Entscheidungsbaum dargestellt werden
  - innere Knoten = Vergleich
  - Blätter = sortierende Permutation

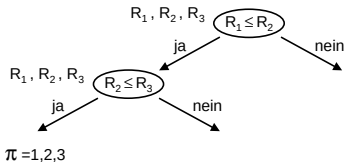
Insertion-Sort Entscheidungsbaum



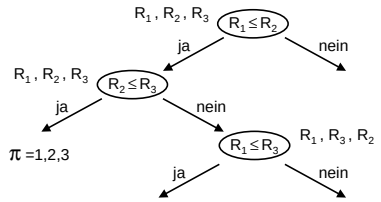
Insertion-Sort Entscheidungsbaum



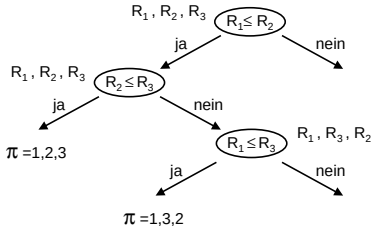
Insertion-Sort Entscheidungsbaum



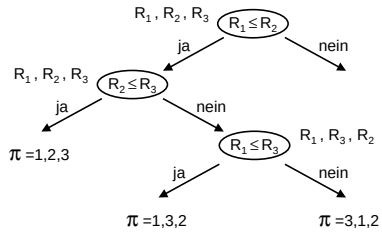
Insertion-Sort Entscheidungsbaum



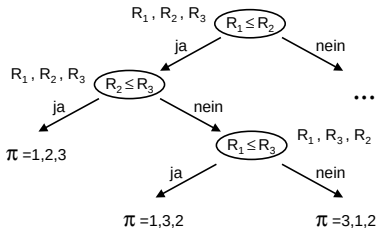
Insertion-Sort Entscheidungsbaum



Insertion-Sort Entscheidungsbaum



Insertion-Sort Entscheidungsbaum



Optimaler Sortieralgorithmus

1. Zahl der möglichen Permutationen: n!
   ⇒ jeder Entscheidungsbaum hat mindestens n! Blätter
2. Maximale Zahl von Blättern in einem Binärbaum der Höhe h beträgt 2^h
   ⇒ jeder Entscheidungsbaum hat höchstens 2^h Blätter

also gilt: $2^h \geq n!$

Optimaler Sortieralgorithmus

1. Zahl der möglichen Permutationen: n!
   ⇒ jeder Entscheidungsbaum hat mindestens n! Blätter
2. Maximale Zahl von Blättern in einem Binärbaum der Höhe h beträgt 2^h
   ⇒ jeder Entscheidungsbaum hat höchstens 2^h Blätter

also gilt: $h \geq c \log(n!) = \Omega(n \log(n))$

Typen von Sortieralgorithmen

- Einfache
  - Selection-Sort, Bubble-Sort, Insertion-Sort
- Höhere
  - Quick-Sort, Merge-Sort, Heap-Sort
- Spezielle
  - Counting-Sort, Radix-Sort, Bucket-Sort

Counting-Sort

- Annahme: $R_1, R_2, ..., R_n \in \{1,...,k\}$
- $k << n$
- Idee: Bestimme zu jedem R_i die Zahl der Elemente $\leq R_i$ und sortiere R_i an die entsprechende Stelle

Counting-Sort: Algorithmus

CountingSort(A,B)

```
for (i = 1; i <= k; ++i)
    C[ i ]=0;
for (i = 1; i <= n; ++i)
    ++C[A[ i ]];
for (i = 2; i <= k; ++i)
    C[ i ] += C[ i-1];
for (i = n; i >= 1; --i)
    B[C[A[ i ]]] = A[ i ];
    C[A[ i ]]--;
```


Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n = 8, k = 6

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n = 8, k = 6

C

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

Counting-Sort: Beispiel

A

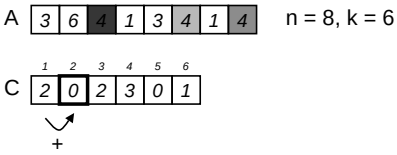
3	6	4	1	3	4	1	4

n = 8, k = 6

C

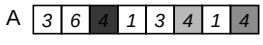
1	2	3	4	5	6
2	0	2	3	0	1

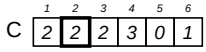
Counting-Sort: Beispiel



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

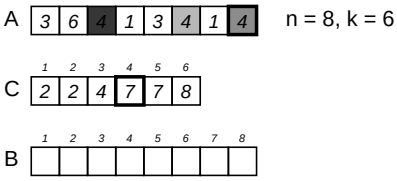
Counting-Sort: Beispiel





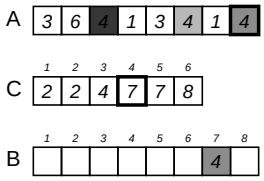
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



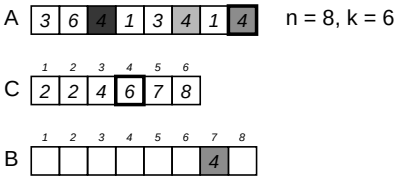
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



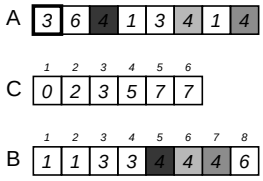
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Beispiel



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Counting-Sort: Aufwand

```
CountingSort(A,B)
   for (i = 1; i <= k; ++i)          O(k)
      C[i]=0;
   for (i = 1; i <= n; ++i)          O(n)
      ++C[A[i]];
   for (i = 2; i <= k; ++i)          O(k)
      C[i] += C[i-1];
   for (i = n; i >= 1; --i)          O(n)
      B[C[A[i]]] = A[i];
      C[A[i]]--;                     O(n+k)
```


Counting-Sort: Aufwand

- Counting-Sort ist nur sinnvoll, wenn k = O(n) und damit

  T(n) = O(n)

- Counting-Sort verwendet ***keine*** Vergleiche
- Counting-Sort ist stabil

Radix-Sort

- Annahme:
  $R_1, R_2, ..., R_n \in \{0,...,k^d-1\}$
- „d-stellige Zahlen zur Basis k“
- „Wörter der Länge d aus einem Alphabet der Größe k“

Radix-Sort: Algorithmus

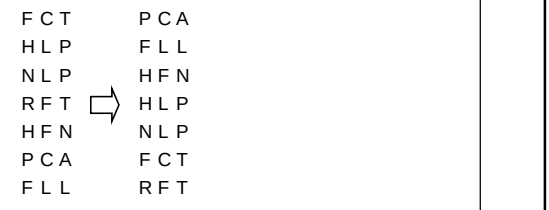
```
RadixSort(A)
   for (i = 0; i < d; ++i)
      „sortiere A stabil nach k^i-wertiger Stelle“
```

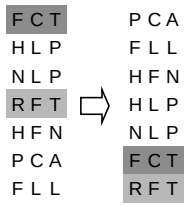

Radix-Sort: Beispiel

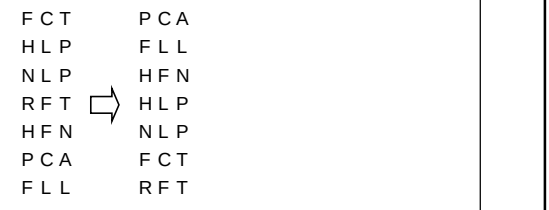
F C T
H L P
N L P
R F T
H F N
P C A
F L L

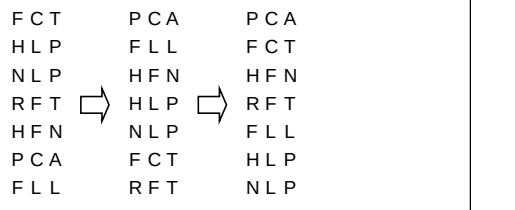
Radix-Sort: Beispiel

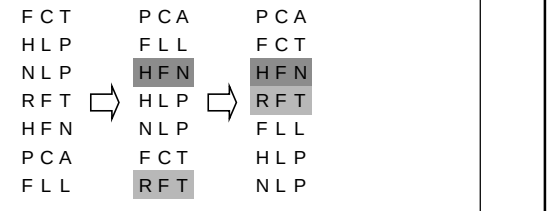
F C T
H L P
N L P
R F T
H F N
P C A
F L L

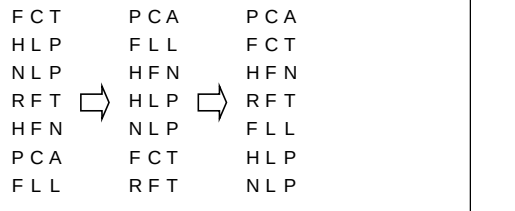












Radix-Sort: Beispiel

F C T		P C A
H L P		F L L
N L P		H F N
R F T	⇨	H L P
H F N		N L P
P C A		F C T
F L L		R F T

Radix-Sort: Beispiel

F C T		P C A
H L P		F L L
N L P		H F N
R F T	⇨	H L P
H F N		N L P
P C A		F C T
F L L		R F T

Radix-Sort: Beispiel

F C T		P C A
H L P		F L L
N L P		H F N
R F T	⇨	H L P
H F N		N L P
P C A		F C T
F L L		R F T

Radix-Sort: Beispiel

F C T		P C A		P C A
H L P		F L L		F C T
N L P		H F N		H F N
R F T	⇨	H L P	⇨	R F T
H F N		N L P		F L L
P C A		F C T		H L P
F L L		R F T		N L P

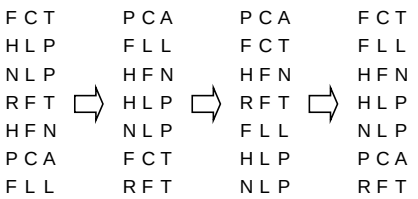
Radix-Sort: Beispiel

F C T		P C A		P C A
H L P		F L L		F C T
N L P		H F N		H F N
R F T	⇨	H L P	⇨	R F T
H F N		N L P		F L L
P C A		F C T		H L P
F L L		R F T		N L P

Radix-Sort: Beispiel

F C T		P C A		P C A
H L P		F L L		F C T
N L P		H F N		H F N
R F T	⇨	H L P	⇨	R F T
H F N		N L P		F L L
P C A		F C T		H L P
F L L		R F T		N L P

Radix-Sort: Beispiel



Radix-Sort: Aufwand

RadixSort(A)

for (i = 0; i < d; ++i)

„sortiere A stabil mit Counting-Sort nach k^i-wertiger Stelle“

$$T(k,d,n)=O(d\,(n+k))$$

Bucket-Sort

- Annahme:
  $R_1, R_2, ..., R_n$ gleichverteilt aus [0,1)
- Idee:
  1. Unterteile [0,1) in n „Buckets“
     [0,1/n), [1/n,2/n), ..., [(n-1)/n,1)
  2. Füge die R_i in die Buckets ein
     (erwartet: ein Element pro Bucket)
  3. Sortiere die Buckets
  4. Hänge die Buckets hintereinander

Bucket-Sort: Algorithmus

```
BucketSort(A,R)
  for (i = 0; i < n; ++i)
    B[i] = [];
  for (i = 0; i < n; ++i)
    B[⌊nA[i]⌋].push(A[i]);
  for (i = 0; i < n; ++i)
    sort(B[i]);
  R = [];
  for (i = 0; i < n; ++i)
    for (j = 0; j < B[i].size(); ++j)
      R.push(B[i][j]);
```


Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

n = 10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6)
6 [0.6,0.7)
7 [0.7,0.8)
8 [0.8,0.9)
9 [0.9,1.0)

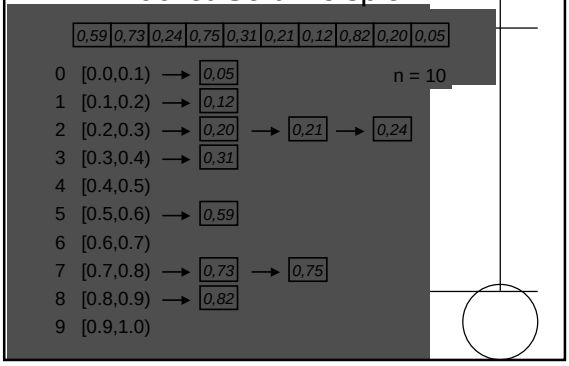
Bucket-Sort: Beispiel

0,59	0,73	0,24	0,75	0,31	0,21	0,12	0,82	0,20	0,05

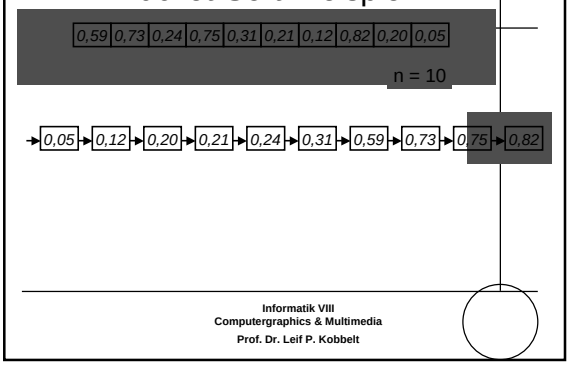
n = 10

0 [0.0,0.1) → 0,05
1 [0.1,0.2) → 0,12
2 [0.2,0.3) → 0,24 → 0,21 → 0,20
3 [0.3,0.4) → 0,31
4 [0.4,0.5)
5 [0.5,0.6) → 0,59
6 [0.6,0.7)
7 [0.7,0.8) → 0,73 → 0,75
8 [0.8,0.9) → 0,82
9 [0.9,1.0)

Bucket-Sort: Beispiel



Bucket-Sort: Beispiel



Bucket-Sort: Analyse

Sei X eine diskrete Zufallsvariable, d.h.

$$X \in \{ x_1, x_2, x_3, \cdots \}$$

Erwartungswert:

$$E(X) = \mu = \sum_{i=1}^{\infty} x_i \cdot P(X = x_i)$$

Varianz:

$$V(X) = \sigma^2 = E\left((X-\mu)^2\right) = E(X^2) - E^2(X)$$

Beispiel

- X=1 mit 50 % Wahrscheinlichkeit
  X=3 mit 50 % Wahrscheinlichkeit
- Erwartungswert: E(X) = 1*0.5 + 3*0.5 = 2.0
- X=1 mit 75 % Wahrscheinlichkeit
  X=3 mit 25 % Wahrscheinlichkeit
- Erwartungswert: E(X) = 1*0.75 + 3*0.25 = 1.5

Beispiel

- X=1 mit 50 % Wahrscheinlichkeit
  X=3 mit 50 % Wahrscheinlichkeit
- Varianz: V(X) = $(1-2)^2$*0.5 + $(3-2)^2$*0.5 = 1.0
- X=1 mit 75 % Wahrscheinlichkeit
  X=3 mit 25 % Wahrscheinlichkeit
- Varianz: V(X) = $(1-1.5)^2$*0.75 + $(3-1.5)^2$*0.25 = 0.75

Bucket-Sort: Analyse

- Sei n_i die Zahl der Elemente in Bucket i
- Wahrscheinlichkeit, dass ein Element in Bucket i fällt ist $p=1/n$
- Wahrscheinlichkeit, dass ***genau*** k Elemente in Bucket i fallen ist

$$P(n_i = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

„Binomialverteilung“

Bucket-Sort: Analyse

Erwartungswert:

$$E(n_i) = np = 1$$

Varianz:

$$V(n_i) = np(1-p) = 1 - \frac{1}{n}$$

Bucket-Sort: Analyse

- Der Sortieraufwand für jeden einzelnen Bucket kann im erwarteten Fall durch O(1) abgeschätzt werden.
- Verteilen und Zusammenfügen: O(n)
- Gesamtaufwand im average case: O(n)
- Performanz ???

Spezielle Sortierverfahren

- Counting-Sort: O(n+k)
- Radix-Sort: O(d (n+k))
- Bucket-Sort: O(n) *erwartet*

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

(2.4.1) k-Selektion

Selektions-Problem:

– gegeben:
  - n (verschiedene) Zahlen A[1..n]
  - eine Zahl k mit $1 \le k \le n$

– gesucht: ***k-kleinstes Element*** von A, d.h. dasjenige $x \in A$ mit

$$|\{ i : A[i] < x \}| = k - 1$$

k-Selektion

Man nennt das k-kleinste Element $x \in A$ für

– k = 1 das *Minimum*
– $k = \lfloor (n+1)/2 \rfloor$, $k = \lceil (n+1)/2 \rceil$ den unteren / oberen *Median*
– k = n das *Maximum*

von A

k-Selektion: Beispiel

Beispiel:

A[1..10] = 41, 76, 32, 62, 21, 52, 19, 83, 0, 91

Minimum: 0

Maximum: 91

k-Selektion: Beispiel

Beispiel:

A[1..10] = 41, 76, 32, 62, 21, 52, 19, 83, 0, 91

Unterer Median: 41, denn

$|\{ i : A[i] < 41 \}| = |\{ 3, 5, 7, 9\}| = 4 = \lfloor (10+1)/2 \rfloor - 1$

Oberer Median: 52, denn

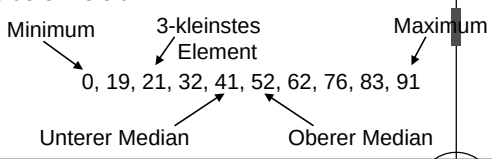
$|\{ i : A[i] < 52 \}| = |\{ 1,3,5,7,9 \}| = 5 = \lceil (10+1)/2 \rceil - 1$

k-Selektion: Beispiel

Beispiel:

A[1..10] = 41, 76, 32, 62, 21, 52, 19, 83, 0, 91

Sortieren liefert:



Selektion durch Sortieren

```
SortSelect(A,k)
  sort(A);
  return A[k];
```

Aufwand = Aufwand für Sortierung:

$$T(n) = O(n \log n)$$

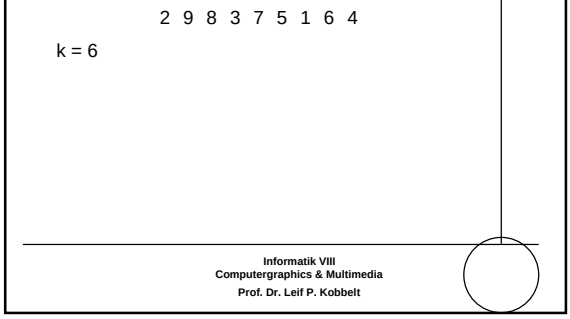
Selektion durch Partitionierung

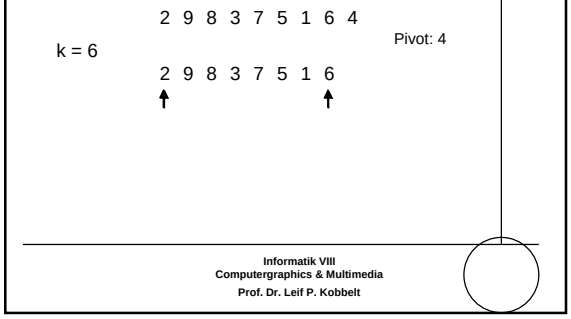
```
PartitionSelect(A,l,r,k)
if (l == r)
   return A[p];
m = Partition(A,l,r);
i = m – l + 1;
if (k == i)
   return A[m];
elseif (k < i)
   return PartitionSelect(A,l,m-1,i);
else
   return PartitionSelect(A,m+1,r,k-i)
```

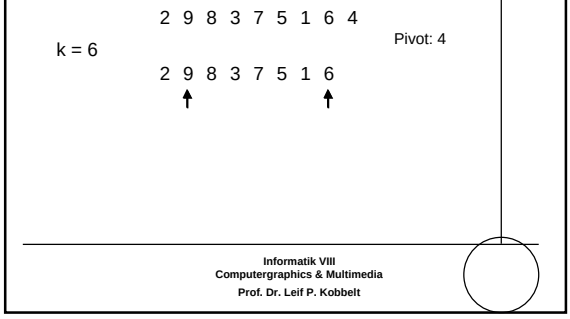
- Berechne das k-kleinste Element von A[l..r]
- Precondition: $1 \le k \le r-l+1$
- Initialer Aufruf mit l = 1, r = n

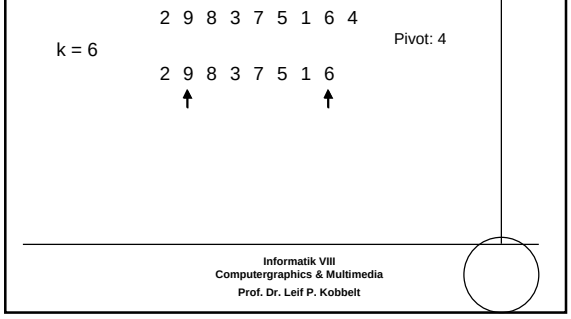
Selektion durch Partitionierung

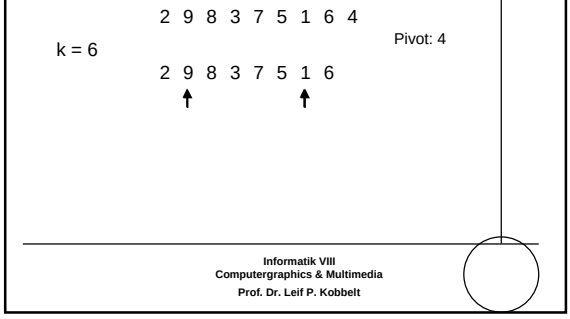
```
PartitionSelect(A,l,r,k)
if (l == r)
   return A[p];
m = Partition(A,l,r);
i = m – l + 1;
if (k == i)
   return A[m];
elseif (k < i)
   return PartitionSelect(A,l,m-1,i);
else
   return PartitionSelect(A,m+1,r,k-i)
```

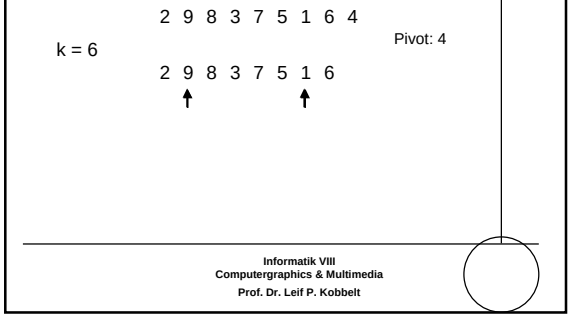












Selektion durch Partitionierung: Beispiel

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

k = 6

Selektion durch Partitionierung: Beispiel

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

Pivot: 4

k = 6

```
2 9 8 3 7 5 1 6
↑             ↑
```

Selektion durch Partitionierung: Beispiel

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

Pivot: 4

k = 6

```
2 9 8 3 7 5 1 6
  ↑           ↑
```

Selektion durch Partitionierung: Beispiel

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

Pivot: 4

k = 6

```
2 9 8 3 7 5 1 6
  ↑           ↑
```

Selektion durch Partitionierung: Beispiel

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

Pivot: 4

k = 6

```
2 9 8 3 7 5 1 6
  ↑         ↑
```

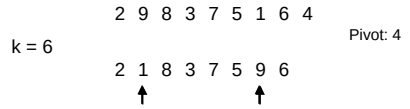
Selektion durch Partitionierung: Beispiel

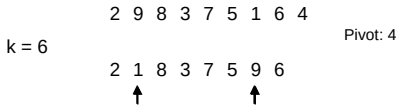
```
2 9 8 3 7 5 1 6 4
```

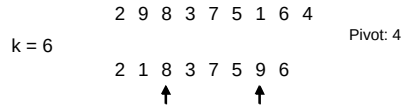
Pivot: 4

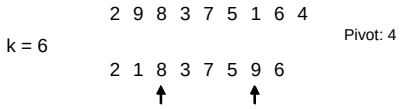
k = 6

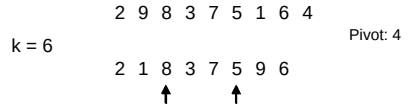
```
2 9 8 3 7 5 1 6
  ↑         ↑
```

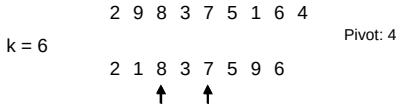












Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Selektion durch Partitionierung: Beispiel

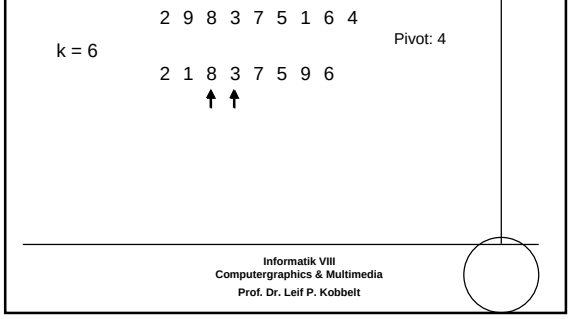
2 9 8 3 7 5 1 6 4

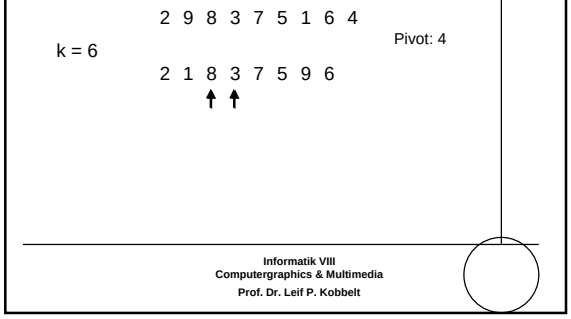
k = 6

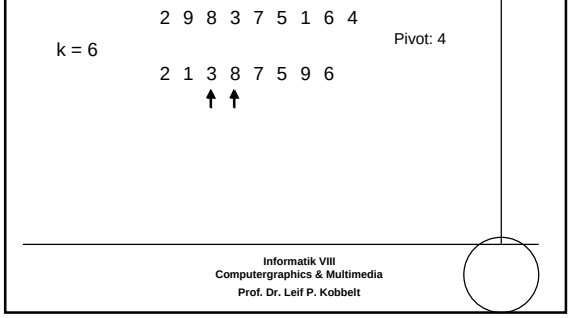
Pivot: 4

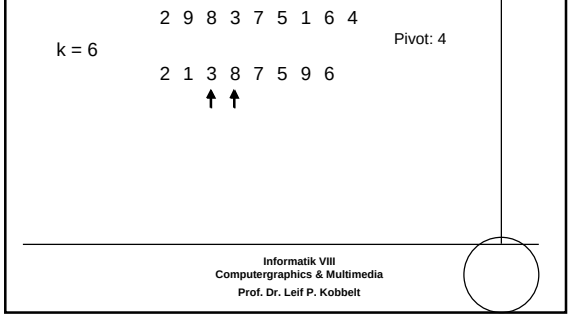
2 1 8 3 7 5 9 6

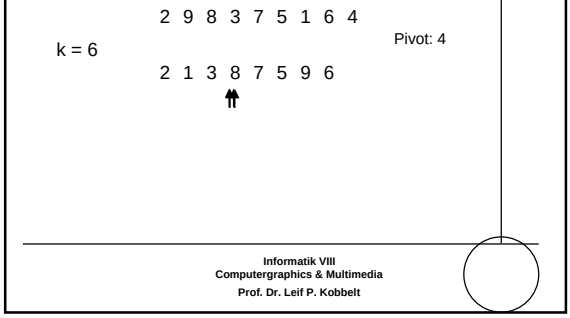
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

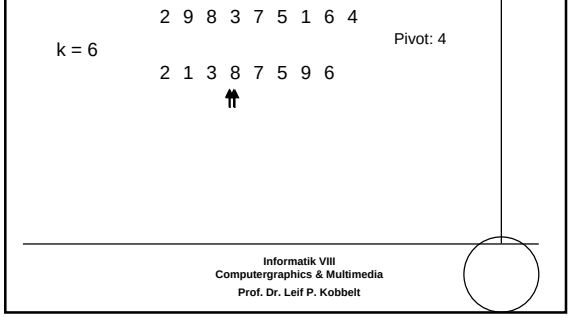












Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

↑ ↑ (under 8 3)

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 8 3 7 5 9 6

↑ ↑

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 3 8 7 5 9 6

↑ ↑

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 3 8 7 5 9 6

↑ ↑

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 3 8 7 5 9 6

⇈

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

Pivot: 4

2 1 3 8 7 5 9 6

⇈

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6 Pivot: 4

2 1 3 8 7 5 9 6

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6 Pivot: 4

2 1 3 4 7 5 9 6 8

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6 Pivot: 4

2 1 3 4 7 5 9 6 8

m = 4, i = m - l + 1 = 4

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

Selektion durch Partitionierung: Beispiel

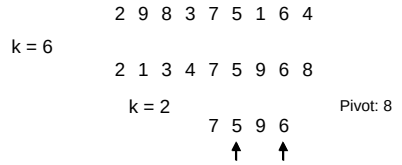
2 9 8 3 7 5 1 6 4

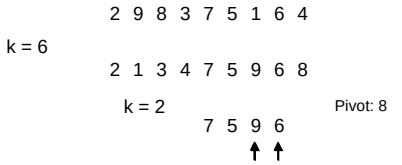
k = 6

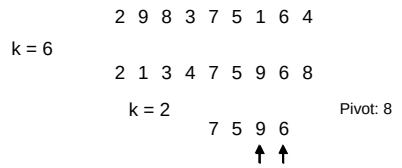
2 1 3 4 7 5 9 6 8

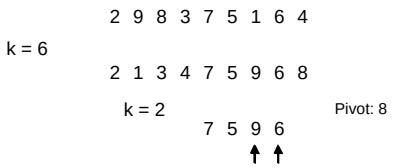
k = 2 Pivot: 8

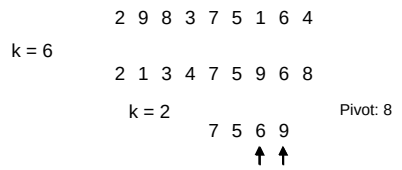
7 5 9 6

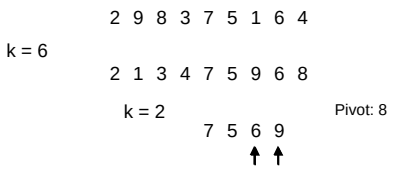












Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 9 6

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 9 6

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 9 6

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 9 6

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

7 5 6 8 9

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2 Pivot: 8

$\underbrace{7\ 5\ 6\ 8}\ \underbrace{9}$

m = 8, i = m - l + 1 = 4

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2

$\boxed{7\ 5\ 6\ 8}\ 9$

k = 2

...

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2

7 5 6 8 9

k = 2

...

5 6 7 8 9

k = 1

Selektion durch Partitionierung: Beispiel

2 9 8 3 7 5 1 6 4

k = 6

2 1 3 4 7 5 9 6 8

k = 2

7 5 6 8 9

k = 2

...

5 6 7 8 9

k = 1

6

Selektion durch Partitionierung

Aufwand von PartitionSelect:

– Worst-case: (vgl. Quick-Sort)
$T(n) = O(n^2)$

– Average-case (ohne Beweis):
$T(n) = O(n)$

– anschaulich: $n/2 + n/4 + \ldots \leq n$

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Basisfall der Rekursion, die *magische* Konstante wird später klar.

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Bilde $\lfloor n/5 \rfloor$ Gruppen von je 5 Elementen (es bleiben 0-4 Elemente übrig) und bestimme deren Median.

Median der Mediane: Algorithmus

Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
$S_<$ = { A[i] : A[i] < m };
$S_=$ = { A[i] : A[i] = m };
$S_>$ = { A[i] : A[i] > m };
if (|$S_<$| > k)
    return Select($S_<$, k);
if (|$S_<$| + |$S_=$| < k)
    return m;
return Select($S_>$, k - |$S_<$| - |$S_=$|);

Bestimme rekursiv den Median der Mediane.

Median der Mediane: Algorithmus

Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
$S_<$ = { A[i] : A[i] < m };
$S_=$ = { A[i] : A[i] = m };
$S_>$ = { A[i] : A[i] > m };
if (|$S_<$| > k)
    return Select($S_<$, k);
if (|$S_<$| + |$S_=$| < k)
    return m;
return Select($S_>$, k - |$S_<$| - |$S_=$|);

Nach Voraussetzung sind alle Elemente verschieden und damit ist |$S_=$| = |{m}| = 1.

Median der Mediane: Algorithmus

Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
$S_<$ = { A[i] : A[i] < m };
$S_=$ = { A[i] : A[i] = m };
$S_>$ = { A[i] : A[i] > m };
if (|$S_<$| > k)
    return Select($S_<$, k);
if (|$S_<$| + |$S_=$| < k)
    return m;
return Select($S_>$, k - |$S_<$| - |$S_=$|);

Das k-te Element liegt in $S_<$.

Median der Mediane: Algorithmus

Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
$S_<$ = { A[i] : A[i] < m };
$S_=$ = { A[i] : A[i] = m };
$S_>$ = { A[i] : A[i] > m };
if (|$S_<$| > k)
    return Select($S_<$, k);
if (|$S_<$| + |$S_=$| < k)
    return m;
return Select($S_>$, k - |$S_<$| - |$S_=$|);

Das k-te Element liegt in $S_=$={m}.

Median der Mediane: Algorithmus

Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
$S_<$ = { A[i] : A[i] < m };
$S_=$ = { A[i] : A[i] = m };
$S_>$ = { A[i] : A[i] > m };
if (|$S_<$| > k)
    return Select($S_<$, k);
if (|$S_<$| + |$S_=$| < k)
    return m;
return Select($S_>$, k - |$S_<$| - |$S_=$|);

Das k-te Element ist das (k - |$S_<$| - |$S_=$|)- te Element aus $S_<$.

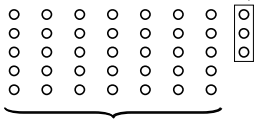
Median der Mediane: Aufwand

Ausgangs-Situation:

Median der Mediane: Aufwand

Anordnung in 5er Gruppen:

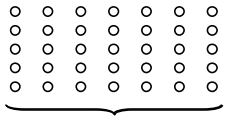
werden vernachlässigt



$\lfloor n/5 \rfloor$ Gruppen

Median der Mediane: Aufwand

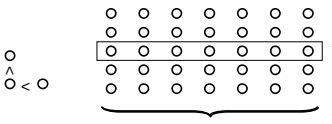
Anordnung in 5er Gruppen:



$\lfloor n/5 \rfloor$ Gruppen

Median der Mediane: Aufwand

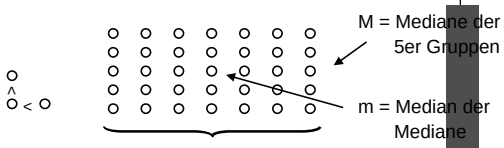
Anordnung in 5er Gruppen, sortiert dargestellt:



$\lfloor n/5 \rfloor$ Gruppen

Median der Mediane: Aufwand

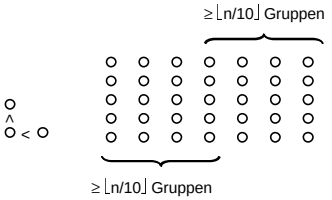
Anordnung in 5er Gruppen, sortiert dargestellt:



$\lfloor n/5 \rfloor$ Gruppen

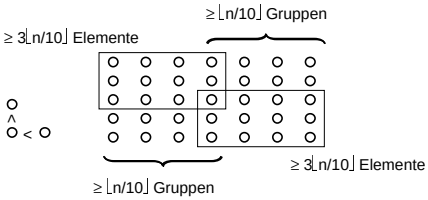
Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:



Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:



Median der Mediane: Aufwand

mind. $3\lfloor n/10 \rfloor$ Elemente sind $\leq m$

$\Rightarrow$ höchstens $n - 3\lfloor n/10 \rfloor$ Elemente sind $> m$

$\Rightarrow$ für $n \geq 50$ ist $|S_>| \leq n - 3\lfloor n/10 \rfloor < 3n/4$

Median der Mediane: Aufwand

mind. $3\lfloor n/10 \rfloor$ Elemente sind $\leq m$

$\Rightarrow$ höchstens $n - 3\lfloor n/10 \rfloor$ Elemente sind $> m$

$\Rightarrow$ für $n \geq 50$ ist $|S_>| \leq n - 3\lfloor n/10 \rfloor < 3n/4$

n = 47	$n - 3\lfloor n/10 \rfloor = 35$	$3n/4 = 35.25$
n = 48	$n - 3\lfloor n/10 \rfloor = 36$	$3n/4 = 36.00$
n = 49	$n - 3\lfloor n/10 \rfloor = 37$	$3n/4 = 36.75$
n = 50	$n - 3\lfloor n/10 \rfloor = 35$	$3n/4 = 37.50$
n = 51	$n - 3\lfloor n/10 \rfloor = 36$	$3n/4 = 38.25$

Median der Mediane: Aufwand

mind. $3\lfloor n/10 \rfloor$ Elemente sind $\leq m$

$\Rightarrow$ höchstens $n - 3\lfloor n/10 \rfloor$ Elemente sind $> m$

$\Rightarrow$ für $n \geq 50$ ist $|S_>| \leq n - 3\lfloor n/10 \rfloor < 3n/4$

Analog: für $n \geq 50$ ist $|S_<| < 3n/4$

Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)                                          } O(1)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ };
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;
return Select(S>, k - |S<| - |S=|);
```


Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);                         O(1)
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ }; } O(n)
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;
return Select(S>, k - |S<| - |S=|);
```


Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);                         O(1)
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ }; O(n)
m = Select(M, ⌊|M|/2⌋);                           } T(n/5)
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;
return Select(S>, k - |S<| - |S=|);
```

Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);                                   O(1)
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ };            O(n)
m = Select(M, ⌊|M|/2⌋);                                        T(n/5)
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };                                      O(n)
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;
return Select(S>, k - |S<| - |S=|);
```


Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);                                   O(1)
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ };            O(n)
m = Select(M, ⌊|M|/2⌋);                                        T(n/5)
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };                                      O(n)
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;                                 < T(3n/4)
return Select(S>, k - |S<| - |S=|);
```


Median der Mediane: Aufwand

Rekurrenz:

$$T(n) \le \begin{cases} c_1 & \text{für } n < 50 \\ c_2 n + T(n/5) + T(3n/4) & \text{für } n \ge 50 \end{cases}$$

Behauptung: Für $c = \max(c_1, 20c_2)$ ist

$$T(n) < cn = O(n)$$

Median der Mediane: Aufwand

Induktionsanfang: für $n < 50$: okay, da $c \ge c_1$.

Induktionsannahme: $T(m) \le cm$ für $m < n$.

Induktionsschluss:

$$\begin{aligned} T(n) &\le c_2 n + T(n/5) + T(3n/4) \\ &\le c_2 n + cn/5 + 3cn/4 \\ &\le c_2 n + 19cn/20 \\ &= cn + n(c_2 - c/20) \\ &\le cn \end{aligned}$$

$20c_2 \le c \Leftrightarrow c_2 \le c/20 \Leftrightarrow c_2 - c/20 \le 0$

Selektion: Zusammenfassung

- Selektion durch Sortieren: O(n log n)
- Selektion durch Partitionieren:
  - average case: O(n)
  - worst case: $O(n^2)$
- Median der Mediane: O(n)

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

(2.4.1) k-Selektion

Selektions-Problem:

– gegeben:
- n (verschiedene) Zahlen A[1..n]
- eine Zahl k mit $1 \le k \le n$

– gesucht: ***k-kleinstes Element*** von A, d.h. dasjenige $x \in A$ mit

$$\left|\{ i : A[i] < x \}\right| = k - 1$$

k-Selektion

- k=1 ... suche Minimum ... O(n)
- k=2 ... zweitkleinstes Element ... O(n)
- ...
- k-kleinstes Element ... O(k n)

- allgemeiner Fall: k = O(n) $\Rightarrow$ O(k n) = O(n^2)

Selektion durch Sortieren

```
SortSelect(A,k)
  sort(A);
  return A[k];
```

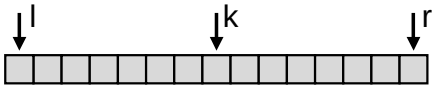
Aufwand = Aufwand für Sortierung:

T (n) = O(n log n)

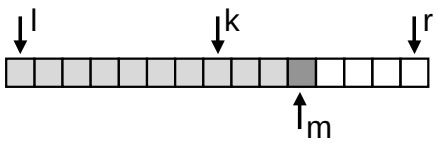
Selektion durch Partitionierung

```
PartitionSelect(A,l,r,k)
if (l == r)
   return A[p];
m = Partition(A,l,r);
i = m – l + 1;
if (k == i)
   return A[m];
elseif (k < i)
   return PartitionSelect(A,l,m-1,k);
else
   return PartitionSelect(A,m+1,r,k-i)
```

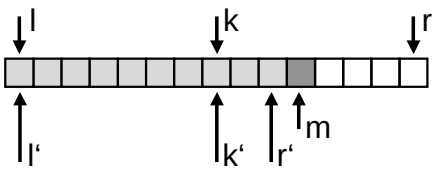

Selektion durch Partitionierung



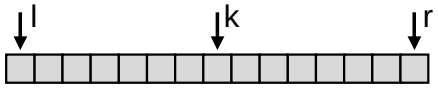
Selektion durch Partitionierung



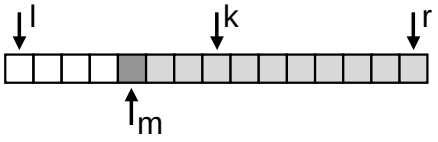
Selektion durch Partitionierung



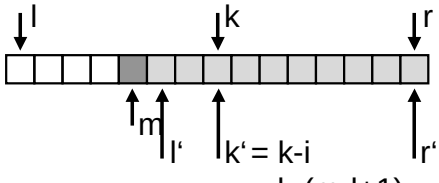
Selektion durch Partitionierung



Selektion durch Partitionierung



Selektion durch Partitionierung



Selektion durch Partitionierung

Aufwand von PartitionSelect:

- Average-case (ohne Beweis):
  $T(n) = O(n)$
- anschaulich: $n/2 + n/4 + \ldots \leq n$
  $(b \geq 1)$ $n/b + n/b^2 + \ldots \leq c\,n$
- Worst-case: (vgl. Quick-Sort)
  $T(n) = O(n^2)$

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Basisfall der Rekursion (Die *magische* Konstante wird später klar.)

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Bilde $\lfloor n/5 \rfloor$ Gruppen von je 5 Elementen (es bleiben 0-4 Elemente übrig) und bestimme deren Median.

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Bestimme rekursiv den Median der Mediane.

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Das k-te Element liegt in $S_<$

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Das k-te Element liegt in $S_= = \{m\}$

Median der Mediane: Algorithmus

```
Select(A,k)
n = |A|;
if (n < 50)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋};
m = Select(M, ⌊|M|/2⌋);
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };
S> = { A[i] : A[i] > m };
if (|S<| > k)
    return Select(S<, k);
if (|S<| + |S=| < k)
    return m;
return Select(S>, k - |S<| - |S=|);
```

Das k-te Element ist das $(k - |S_<| - |S_=|)$- te Element aus $S_>$

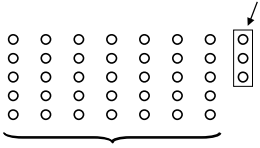
Median der Mediane: Aufwand

Ausgangs-Situation:

Median der Mediane: Aufwand

Anordnung in 5er Gruppen:

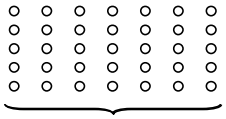
werden vernachlässigt



$\lfloor n/5 \rfloor$ Gruppen

Median der Mediane: Aufwand

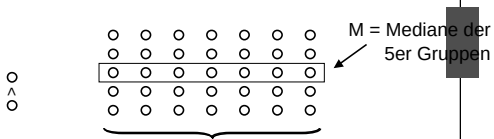
Anordnung in 5er Gruppen:



$\lfloor n/5 \rfloor$ Gruppen

Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:

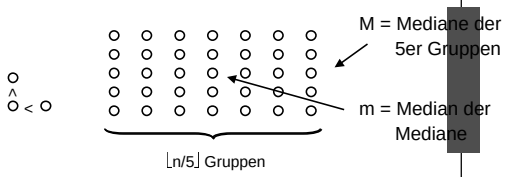


M = Mediane der 5er Gruppen

$\lfloor n/5 \rfloor$ Gruppen

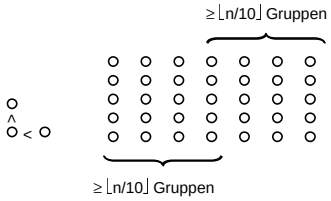
Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:



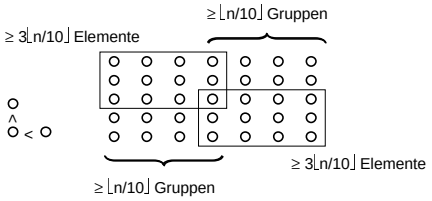
Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:



Median der Mediane: Aufwand

Anordnung in 5er Gruppen, sortiert dargestellt:



Median der Mediane: Aufwand

```
Select(A,k)
n = |A|;
if (n < 50)                                          O(1)
    return SortSelect(A, k);
M = { SortSelect(A[5i+1..5i+5]) : i = 0,...⌊n/5⌋ };   O(n)
m = Select(M, ⌊|M|/2⌋);                              T(n/5)
S< = { A[i] : A[i] < m };
S= = { A[i] : A[i] = m };                            O(n)
S> = { A[i] : A[i] > m };
if (|S<| > k) return Select(S<, k);
if (|S<| + |S=| < k) return m;                       < T(3n/4)
return Select(S>, k - |S<| - |S=|);
```


Median der Mediane: Aufwand

Rekurrenz:

$$T(n) \le \begin{cases} c_1 & \text{für } n < 50 \\ c_2 n + T(n/5) + T(3n/4) & \text{für } n \ge 50 \end{cases}$$

Behauptung: Für $c = \max(c_1, 20c_2)$ ist

$$T(n) < cn = O(n)$$

Selektion: Zusammenfassung

- Selektion durch Sortieren: O(n log n)
- Selektion durch Partitionieren:
  - average case: O(n)
  - worst case: $O(n^2)$
- Median der Mediane: O(n)

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

(2.4.2) String Matching

- String-Matching Problem:
  - gegeben: Text T[1..n] *(Folge von Symbolen)*
    Muster P[1..m]
  - gesucht: erstes / alle Vorkommen von P in T
- Anwendung in Text-Editoren,
  DNA Analyse, Video-Kompression,
  Computer-Vision ...

String Matching: Begriffe

- Alphabet: Σ *(Menge von Symbolen)*
- Menge aller Wörter der Länge n:

$$\Sigma^0 = \{\varepsilon\}$$
$$\Sigma^n = \{a_1 a_2 \cdots a_n \mid a_i \in \Sigma\}$$

String Matching: Begriffe

Beispiel: $\Sigma = \{0, 1\}$

$$\Sigma^0 = \{\varepsilon\}$$
$$\Sigma^1 = \{0,1\}$$
$$\Sigma^2 = \{00, 01, 10, 11\}$$
$$\Sigma^3 = \{000,001,01\,0,011,100, 101,110,11\,1\}$$

String Matching: Begriffe

Beispiel: $\Sigma = \{a, b, c, ..., x, y, z\}$

$\Sigma^0 = \{\varepsilon\}$ „leeres“ Wort

$$\Sigma^1 = \{a, b, c, ..., x, y, z\}$$
$$\Sigma^2 = \{aa, ab, ac, ..., fu, ..., kp, ..., zx, zy, zz\}$$
$$\Sigma^3 = \{aaa, aab, aac, ..., dtp, ..., zzz\}$$

String Matching: Begriffe

Menge aller ***endlichen*** Wörter:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$$
$$= \bigcup_{n \geq 0} \Sigma^n$$

... Elemente dieser Menge
nennt man auch ***Strings***

String Matching: Begriffe

Beispiel: $\Sigma = \{0,1\}$

$01001 \in \Sigma^*$ $\quad 01021 \notin \Sigma^*$

$011 \in \Sigma^*$ $\quad abc \notin \Sigma^*$

$00001111 \in \Sigma^*$ $\quad 0000\cdots \notin \Sigma^*$

$\varepsilon \in \Sigma^*$

String Matching: Begriffe

- String: $P = \{a_1, a_2, ..., a_n\}$
- **Präfix:** $P_k = \{a_1, a_2, ..., a_k\}$ $\quad P_k \subset P$
- **Suffix:** $P_{-k} = \{a_{n-k+1}, ..., a_n\}$ $\quad P_{-k} \supset P$

Naives String Matching

```
NaiveStringMatch(T,P)
for (s = 0 ; s ≤ n-m ; s++)
    i = 1;
    while (i ≤ m && P[ i ] == T[ s+i ])
        i++;
    if (i == m + 1)
        print „Muster gefunden an Stelle“ s+1;
```


Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
          00010010001100
s=0       00100
          ↑
        i = 1
```


Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
          00010010001100
s=0       00100
           ↑
        i = 2
```


Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
          00010010001100
s=0       00100
            ↑
          i = 3
```

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~
s=1 00100
↑
i = 1

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~
s=1 00100
↑
i = 2

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~
s=1 00100
↑
i = 3

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~
s=1 00100
↑
i = 4

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

00010010001100
~~00100~~
s=1 00100
↑
i = 5

Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
00010010001100
00100
s=1  00100
       ↑
     i = 6
```


Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
00010010001100
00100
  00100    (Output s+1=2)
s=2  00100
       ↑
     i = 1
```


Naives String Matching: Beispiel

T = 00010110001100
P = 00100

```
00010010001100
00100
  00100    (Output s+1=2)
s=2  00100
       ↑
     i = 2
```


Naives String Matching

```
NaiveStringMatch(T,P)
for (s = 0 ; s ≤ n-m ; s++)
  i = 1;
  while (i ≤ m && P[ i ] == T[ s+i ])   } ≤ m
    i++;                                 } n-m+1
  if (i == m + 1)
    print „Muster gefunden an Stelle“ s+1;
```

$T(n,m) = O((n-m+1)m) = O(n\,m)$

Rabin-Karp Algorithmus

- Der Einfachheit halber sei

$$\Sigma = \{ 0,1,2,3,4,5,6,7,8,9 \}$$

- Idee: Fasse Wörter aus Σ als Zahlen aus IN auf:

String 145 = 1 ⊕ 4 ⊕ 5 „Concatenation“
Zahl 145 = 100 + 40 + 4 „Addition“

Rabin-Karp Algorithmus

- Zwei Worte aus Σ^* sind gleich, wenn die entsprechenden Zahlen gleich sind
  ⇒ Vergleich von Zahlen statt Vergleich von Worten (mehrere Ziffern parallel)
- Sei p die zu P[1..m] korrespondierende Zahl
- Sei t_s die zu T[s+1...s+m] korrespondierende Zahl

Rabin-Karp Algorithmus

Beispiel:

P = „72153“
m = 5
p = 72153

T = „58721537543“, n = 11
t_0 = 58721
t_1 = 87215
t_2 = 72153
t_3 = 21537
t_4 = 15375
t_5 = 53754
t_6 = 37543

Rabin-Karp Algorithmus

Berechnung von p aus P[1..m]:

$p = P[1] * 10^{m-1} + P[2] * 10^{m-2} + ... + P[m-1] * 10 + P[m]$

Aufwand:

- Additionen: m-1 = O(m)
- Multiplikationen: m*(m-1)/2 = $O(m^2)$
- Gesamt: $O(m^2)$

$\Rightarrow$ diese naive Berechung ist ineffizient!

Rabin-Karp Algorithmus

Berechnung von p mit Horner-Schema:

$P[1] * 10^{m-1} + P[2] * 10^{m-2} + ... + P[m-2] * 10^2 + P[m-1] * 10 + P[m]$

$= (P[1] * 10^{m-2} + P[2] * 10^{m-3} + ... + P[m-2] * 10^1 + P[m-1]) * 10 + P[m]$

$= ((P[1] * 10^{m-3} + P[2] * 10^{m-2} + ... + P[m-2]) * 10 + P[m-1]) * 10 + P[m]$

$= ...$

$= ((...(P[1] * 10 + P[2]) * 10 + ... + P[m-2]) * 10 + P[m-1]) * 10 + P[m]$

Rabin-Karp Algorithmus

Berechnung von p mit Horner-Schema:

$p = ((...(P[1] * 10 + P[2]) ...) * 10 + P[m-1]) * 10 + P[m]$

Algorithmus:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```


Rabin-Karp Algorithmus

Horner-Schema:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0					

Rabin-Karp Algorithmus

Horner-Schema:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0	7				

*10

Rabin-Karp Algorithmus

Horner-Schema:
```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0	7	72			

*10

Rabin-Karp Algorithmus

Horner-Schema:
```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0	7	72	721		

*10

Rabin-Karp Algorithmus

Horner-Schema:
```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0	7	72	721	7215	

*10

Rabin-Karp Algorithmus

Horner-Schema:
```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

Beispiel:

P = 72153, m = 5

i	-	1	2	3	4	5
P[i]	-	7	2	1	5	3
p	0	7	72	721	7215	72153

*10

Rabin-Karp Algorithmus

Aufwand des Horner-Schemas:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```

m Durchläufe:
- Additionen: 1
- Multiplikationen: 1

⇒ Aufwand des Horner-Schemas: O(m)

Rabin-Karp Algorithmus

- Analog zu p lassen sich auch die t_s mittels des Horner-Schemas berechnen.
- Aufwand zur Berechung aller t_s: $O((n-m+1)m)$

Rabin-Karp Algorithmus

- Analog zu p lassen sich auch die t_s mittels des Horner-Schemas berechnen.
- Aufwand zur Berechung aller t_s: O((n-m+1)m)
- Noch besser:
  - Berechne nur t_0 mit Horner-Schema
  - Berechne t_{s+1} aus t_s durch

$$t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$$

Rabin-Karp Algorithmus

Beispiel: $t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$

$T = 38721537543,\ m = 5$

$t_0 = 38721$

$t_1 = 87215$

Rabin-Karp Algorithmus

Beispiel: $t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$

$T = 38721537543,\ m = 5$

$t_0 = 38721$

$t_1 = 87215 \qquad = 10 (t_0 - 30000) + 5$

Rabin-Karp Algorithmus

Beispiel: $t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$

$T = 38721537543,\ m = 5$

$t_0 = 38721$

$t_1 = 87215 \qquad = 10 (t_0 - 30000) + 5$

$t_2 = 72153 \qquad = 10 (t_1 - 80000) + 3$

Rabin-Karp Algorithmus

Beispiel: $t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$

$T = 38721537543,\ m = 5$

$t_0 = 38721$

$t_1 = 87215 \qquad = 10 (t_0 - 30000) + 5$

$t_2 = 72153 \qquad = 10 (t_1 - 80000) + 3$

$t_3 = 21537 \qquad = 10 (t_2 - 70000) + 7$

Rabin-Karp Algorithmus

Beispiel: $t_{s+1} = 10 \left(t_s - T[s+1] * 10^{m-1} \right) + T[s+m+1]$

$T = 38721537543,\ m = 5$

$t_0 = 38721$

$t_1 = 87215 \qquad = 10 (t_0 - 30000) + 5$

$t_2 = 72153 \qquad = 10 (t_1 - 80000) + 3$

$t_3 = 21537 \qquad = 10 (t_2 - 70000) + 7$

$t_4 = 15375 \qquad = \ldots$

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```


Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```

Vorberechnung, einmaliger Aufwand von O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```

2 x Horner-Schema: O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```

Schleife wird O(n)-mal durchlaufen

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```


Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t  = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t   += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```

Muster nach rechts schieben. Aufwand O(1) vs. O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = 10^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += 10 * p + P[i];
   t  += 10 * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = 10 * (t – h*T[s+1])+T[s+m+1];
```

Nicht-uniformes Kostenmaß
⇒ Aufwand für Vergleich: O(m)

Exkurs: *Modulo*

- $a \equiv b \bmod q \;:\Leftrightarrow\; \exists\, k:\ a = b + k*q$
- $\forall a \in \mathbb{N}\ \exists b \in \{0,..,q-1\}:\ a \equiv b \bmod q$
- Es gilt:

$$\begin{aligned}(a+b) \bmod q &= (a_1 + q*a_2 + b_1 + q*b_2) \bmod q \\ &= (a_1 + b_1 + q*(a_2 + b_2)\,) \bmod q \\ &= (a_1 + b_1) \bmod q \\ &= ((a \bmod q) + (b \bmod q)) \bmod q\end{aligned}$$

Exkurs: *Modulo*

- Es gilt

$$\begin{aligned}(a*b) \bmod q &= ((a_1 + q*a_2) * (b_1 + q*b_2)) \bmod q \\ &= (a_1*b_1 + q*(a_1*b_2 + a_2*b_1) + q^2*a_2*b_2) \bmod q \\ &= (a_1*b_1) \bmod q \\ &= ((a \bmod q) * (b \bmod q)) \bmod q\end{aligned}$$

Exkurs: *Modulo*

- Beispiele ...
  - $10 \equiv -4 \equiv 3 \mod 7$
  - $14 * 27 \equiv 3 * 5 \equiv 15 \equiv 4 \mod 11$
  - $100 + 5462 \equiv 0 + 0 \equiv 0 \mod 2$
- Es gilt

  $a = b \;\Rightarrow\; a \equiv b \bmod q$

  aber nicht umgekehrt!

Rabin-Karp Algorithmus

```
RabinKarp(T,P)
h = 10^(m-1) mod q;
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += (10 * p + P[i]) mod q;
   t  += (10 * t + T[i]) mod q;
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        if (P[1..m] = T[s+1..s+m])
             print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = (10 * (t – h*T[s+1])+T[s+m+1]) mod q;
```


Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, $h \equiv 10^{m-1} \equiv (-3)^2 \equiv 9 \bmod 13$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, $h \equiv 10^{m-1} \equiv (-3)^2 \equiv 9 \bmod 13$

Initialisierung: p = 0

Horner-Schema: mod 13: $p \equiv (1 * 10 + 5) * 10 + 3$
$\equiv (-3 + 5) * 10 + 3$
$\equiv 20 + 3$
$\equiv 10$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 10*(t_0 - h*3) + 1 \equiv 10*(6 - 27) + 1 \equiv 10*5 + 1 \equiv 51 \equiv 12$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 12$
$t_2 \equiv 10*(t_1 - h*8) + 5 \equiv 10*(12 - 72) + 5 \equiv 10*(-1+6) + 5 \equiv 3$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 12$
$t_2 \equiv 3$
$t_3 \equiv 10*(t_2 - h*3) + 3 \equiv 10*(3 - 27) + 3 \equiv 10*2 + 3 \equiv 10$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 12$
$t_2 \equiv 3$
$t_3 \equiv 10$
$t_4 \equiv 10*(t_3 - h*1) + 0 \equiv 10*(10 - 9) + 0 \equiv 10$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 12$
$t_2 \equiv 3$
$t_3 \equiv 10$
$t_4 \equiv 10$

Rabin-Karp Algorithmus: Beispiel

T = 383153054324562, P = 153, n = 15, m = 3

q = 13, h = 9, p = 10

$t_0 \equiv 6$
$t_1 \equiv 12$
$t_2 \equiv 3$
$t_3 \equiv 10$
$t_4 \equiv 10$

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rabin-Karp Algorithmus: Aufwand

```
RabinKarp(T,P)
h = 10^(m-1) mod q;                                   } O(m)
p = 0;                                                }
t  = 0;                                               }
for (i = 1 ; i ≤ m ; i++)                             } O(m)
   p  += (10 * p + P[i]) mod q;                       }
   t   += (10 * t + T[i]) mod q;                      }
for (s = 0 ; s ≤ n – m ; s++)                         }
   if (p = t)                                         }
        if (P[1..m] = T[s+1..s+m])      } O(m)        } O((n-m+1)m)
              print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)                                     }
        t = (10 * (t – h*T[s+1])+T[s+m+1]) mod q;     }
```

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rabin-Karp Algorithmus: Aufwand

```
RabinKarp(T,P)
h = 10^(m-1) mod q;                                   } O(m)
p = 0;                                                }
t  = 0;                                               }
for (i = 1 ; i ≤ m ; i++)                             } O(m)
   p  += (10 * p + P[i]) mod q;                       }
   t   += (10 * t + T[i]) mod q;                      }
for (s = 0 ; s ≤ n – m ; s++)                         }
   if (p = t)                      } O(log(q))        }
        if (P[1..m] = T[s+1..s+m])      } O(m)        } O(n)
              print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)                                     }
        t = (10 * (t – h*T[s+1])+T[s+m+1]) mod q;     }
```

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rabin-Karp Algorithmus: Zusammenfassung

Aufwand von RabinKarp:

- Worst case:
  O((n-m+1)*m)
- Average case: Ist q ≥ m und ist die erwartete Anzahl von Matches in O(1), dann hat RabinKarp einen erwarteten Aufwand von O(n)

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rabin-Karp Algorithmus: Zusammenfassung

Aufwand von RabinKarp:

- Average case: O(n)
- Annahme: (mod q) - Schlüssel sind *gleichverteilt*
- Bei O(n) möglichen Sub-Strings sind O(n/q) passende Schlüssel zu erwarten
- Aufwand: O(n+m*n/q) = O(n) für q ≥ m

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Knuth-Morris-Pratt

- Geg.: Text T[1...n], Muster P[1..m]
- Beim naiven Ansatz und bei Rabin-Karp werden Symbole aus T[ ] *mehrfach* verglichen (bis zu m Mal !)
- Finde Algorithmus, der jedes T[i] nur *einmal* vergleicht ...

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

- Sei s der Index der äußeren Schleife und i der Index der inneren Schleife beim naiven String-Matching
- Dann gilt T[s+1 ... s+i] = P[1 ... i]
- Eine Verschiebung des Index s nach s+1 macht nur Sinn wenn
  T[s+2 ... s+i] = P[1 ... i-1]
  bzw.
  P[1 ... i-1] = P[2 ... i]

Knuth-Morris-Pratt

- Die Bedingung P[1 ... i-1] = P[2 ... i] hängt *nur* vom Suchmuster P[ ] ab, kann also *vorberechnet* werden.
- Genauso: Verschiebung auf s+2 macht nur Sinn, wenn P[1 ... i-2] = P[3 ... i]

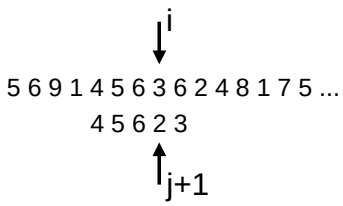
Knuth-Morris-Pratt

- Allgemein: suche das *größte* j, für das P[1 ... j] = P[i-j+1 ... i] gilt.
- Dann ist s+i-j die nächste Möglichkeit für einen Match

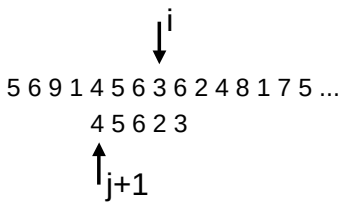
Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
   Q = PrefixFunction(P);
   j   = 0;
   for (i=1 ; i≤n ; i++)
        while ( j>0 && P[ j+1] ≠ T[ i ] )
             j = Q[ j ];
        if (P[ j+1] = T[ i ])  j++;
        if (j = m)
             print „Muster an der Stelle“ i-m „gefunden“;
             j = Q[j];
```

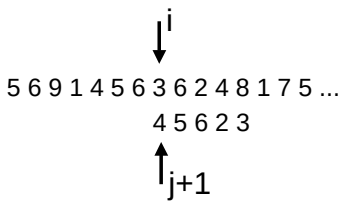
Knuth-Morris-Pratt



Knuth-Morris-Pratt



Knuth-Morris-Pratt



Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
   Q = PrefixFunction(P);
   j   = 0;
   for (i=1 ; i≤n ; i++)
        while ( j>0 && P[ j+1] ≠ T[ i ] )
             j = Q[ j ];
        if (P[ j+1] = T[ i ])  j++;
        if (j = m)
             print „Muster an der Stelle“ i-m „gefunden“;
             j = Q[ j ];
```

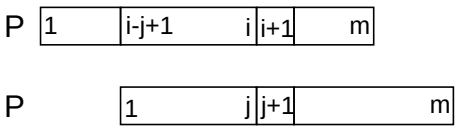
Knuth-Morris-Pratt

- Bedeutung von Q:
  - $Q[i] = j \Leftrightarrow P[1...j] = P[i-j+1 ... i]$ und es gibt kein größeres $j < i$
  - Insbesondere: $0 \le Q[i] < i$
  - Offensichtlich $Q[1] = 0$
  - Aus $P[1...j] = P[i-j+1 ... i]$ und $P[j+1] = P[i+1]$ folgt $P[1...j+1] = P[i-j+1 ... i+1]$
  - Aus $Q[i] = j$ und $P[j+1] = P[i+1]$ folgt $Q[i+1] = j+1$

Knuth-Morris-Pratt

- Bedeutung von Q:
  - $Q[i] = j \Leftrightarrow P[1...j] = P[i-j+1 ... i]$ und es gibt kein größeres $j < i$
  - Insbesondere: $0 \le Q[i] < i$
  - Offensichtlich $Q[1] = 0$
  - Aus $P[1...j] = P[i-j+1 ... i]$ und $P[j+1] \neq P[i+1]$ folgt $Q[i+1] \le Q[i] + 1$

Knuth-Morris-Pratt



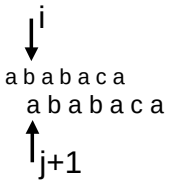
Knuth-Morris-Pratt

- PrefixFunction(P[1...m]);

```
Q[1] = 0;
j = 0;
for (i=2 ; i≤m ; i++)
    while (j>0 && P[j+1] ≠ P[i])
        j = Q[j]
    if (P[j+1] = P[i])  j++;
    Q[i] = j;
return Q;
```

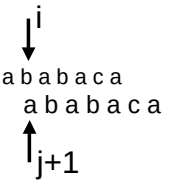

Knuth-Morris-Pratt

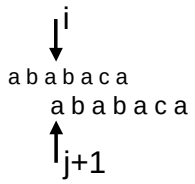
PrefixFunction() : Q[ ] = [0, ...]

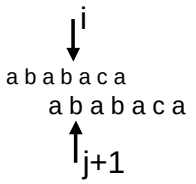


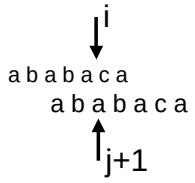
Knuth-Morris-Pratt

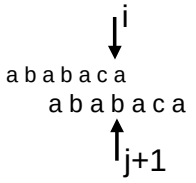
PrefixFunction() : Q[ ] = [0,0, ...]

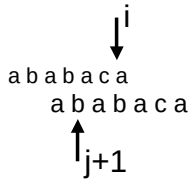


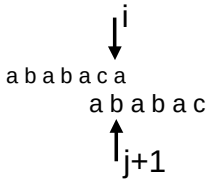












Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0, ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1 ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2 ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3 ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3 ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

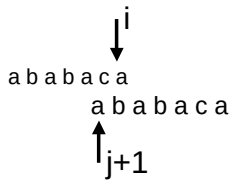
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3 ...]

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

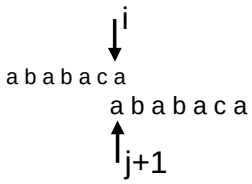
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3 ...]



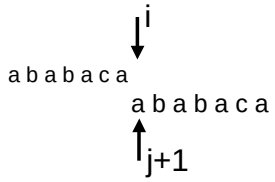
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3 ...]



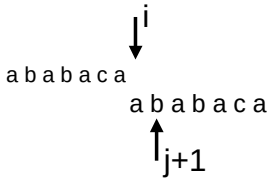
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3,0 ...]



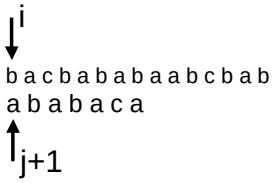
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,0,1,2,3,0,1]



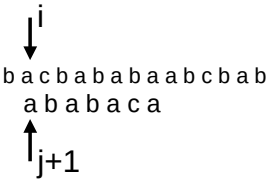
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



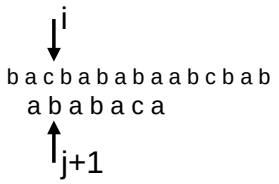
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

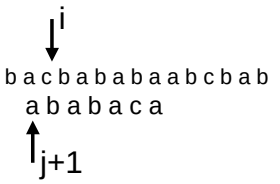
bacbababaabcbab

ababaca

j+1

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

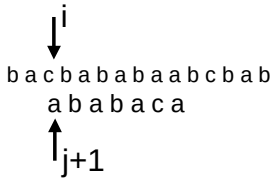
bacbababaabcbab

ababaca

j+1

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

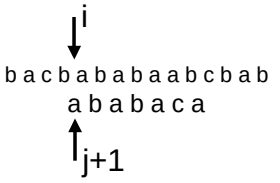
bacbababaabcbab

ababaca

j+1

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

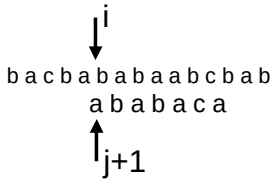
bacbababaabcbab

ababaca

j+1

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

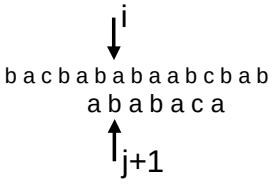
bacbababaabcbab

ababaca

j+1

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



i

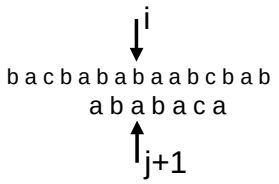
bacbababaabcbab

ababaca

j+1

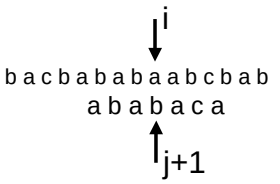
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



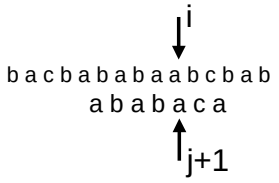
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



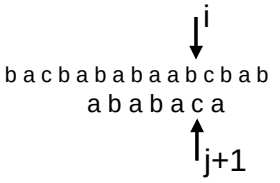
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



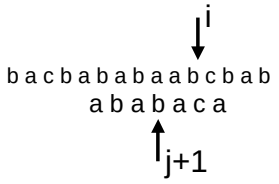
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



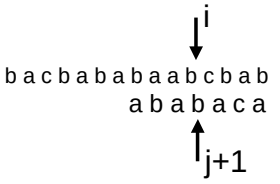
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



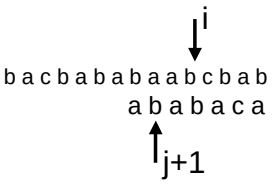
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



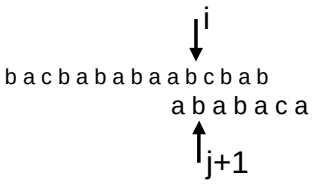
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



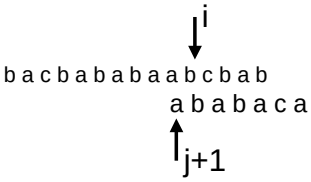
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



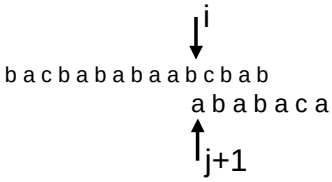
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



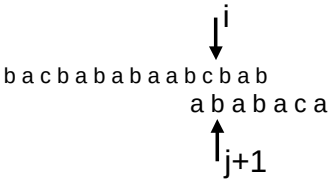
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



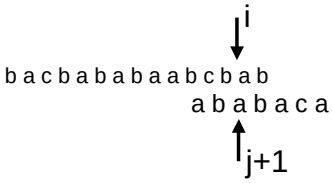
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



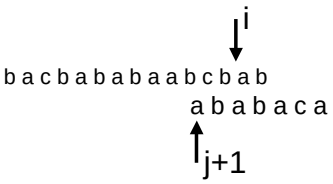
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



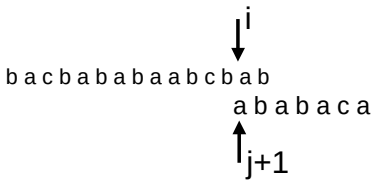
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



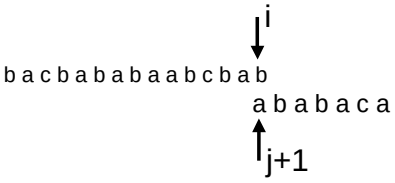
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



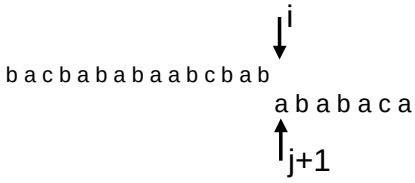
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



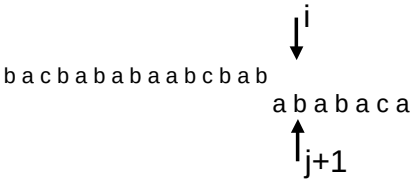
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,0,1,2,3,0,1]



Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
   Q = PrefixFunction(P);
   j   = 0;
   for (i=1 ; i≤n ; i++)
        while ( j>0 && P[ j+1] ≠ T[ i ] )
             j = Q[ j ];
        if (P[ j+1] = T[ i ])  j++;
        if (j = m)
             print „Muster an der Stelle“ i-m „gefunden“;
             j = Q[ j ];
```

Knuth-Morris-Pratt

- Aufwand
  - äußere Schleife wird O(n)-mal durchlaufen
  - In jedem Durchlauf wird j maximal um 1 erhöht.
  - In der inneren Schleife wird j nur verringert
  - Innere Schleife kann insgesamt nur O(n) mal durchlaufen werden **(Potential-Argument)**
  - Gesamtaufwand O(n)

Knuth-Morris-Pratt

- Aufwand
  - KnuthMorrisPratt() ... O(n)
  - PrefixFunction() ... O(m)
  - ... O(n+m)

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

(2.4.2) String Matching

- String-Matching Problem:
  - gegeben: Text T[1..n] *(Folge von Symbolen)*
    Muster P[1..m]
  - gesucht: erstes / alle Vorkommen von P in T
- Anwendung in Text-Editoren,
  DNA Analyse, Video-Kompression,
  Computer-Vision ...

String Matching: Begriffe

- Alphabet: Σ *(Menge von Symbolen)*
- Menge aller Wörter der Länge n:

$$\Sigma^0 = \{\varepsilon\}$$
$$\Sigma^n = \{a_1 a_2 \cdots a_n \mid a_i \in \Sigma\}$$

String Matching: Begriffe

Menge aller ***endlichen*** Wörter:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{n \geq 0} \Sigma^n$$

... Elemente dieser Menge nennt man auch ***Strings***

String Matching: Begriffe

- String: $P = \{a_1, a_2, ..., a_n\}$
- **Präfix:** $P_k = \{a_1, a_2, ..., a_k\}$ $\quad P_k \subset P$
- **Suffix:** $P_{-k} = \{a_{n-k+1}, ..., a_n\}$ $\quad P_{-k} \supset P$

Naives String Matching

```
NaiveStringMatch(T,P)
for (s = 0 ; s ≤ n-m ; s++)
    i = 1;
    while (i ≤ m && P[ i ] == T[ s+i ])
        i++;
    if (i == m + 1)
        print „Muster gefunden an Stelle“ s+1;
```

Naives String Matching

```
NaiveStringMatch(T,P)
for (s = 0 ; s ≤ n-m ; s++)
  i = 1;
  while (i ≤ m && P[ i ] == T[ s+i ])   } ≤ m
      i++;                              } n-m+1
  if (i == m + 1)
      print „Muster gefunden an Stelle“ s+1;
```

$T(n,m) = O((n-m+1)m) = O(n\,m)$

Rabin-Karp Algorithmus

- Kodiere Folgen von Symbolen als Zahlen
- Zwei Worte aus Σ^* sind gleich, wenn die entsprechenden Zahlen gleich sind
  ⇒ Vergleich von Zahlen statt Vergleich von Worten (mehrere Ziffern parallel)
- Sei p die zu P[1..m] korrespondierende Zahl und t_s die zu T[s+1...s+m] korrespondierende Zahl

Rabin-Karp Algorithmus

Beispiel:

P = „72153“
m = 5
p = 72153

T = „58721537543“, n = 11
t_0 = 58721
t_1 = 87215
t_2 = 72153
t_3 = 21537
t_4 = 15375
t_5 = 53754
t_6 = 37543

Rabin-Karp Algorithmus

- Beispiel: Σ = { a,b, ... ,z, _ }, $\#\Sigma = 27$
- a=0, b=1, ... , z=25, _=26
- the ≅ $19*27^2 + 7*27 + 4 = 14044$
  pet ≅ $15*27^2 + 4*27 + 19 = 11062$
  got ≅ $6*27^2 + 14*27 + 19 = 4771$
  wet ≅ $22*27^2 + 4*27 + 19 = 16165$
- Achtung: nur eindeutig, bei fester String-Länge !

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772
15295

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772
15295
19309

Rabin-Karp Algorithmus

Berechnung von p mit Horner-Schema: $\#\Sigma = 10$

$p = ((...(P[1] * 10 + P[2]) ...) * 10 + P[m-1]) * 10 + P[m]$

Algorithmus:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 10 + P[i];
```


Rabin-Karp Algorithmus

Berechnung von p mit Horner-Schema: $\#\Sigma = 27$

$p = ((...(P[1] * 27 + P[2]) ...) * 27 + P[m-1]) * 27 + P[m]$

Algorithmus:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * 27 + P[i];
```


Rabin-Karp Algorithmus

Aufwand des Horner-Schemas:

```
p = 0;
for (i = 1; i ≤ m; ++i)
  p = p * #Σ + P[i];
```

m Durchläufe:
- Additionen: 1
- Multiplikationen: 1

⇒ Aufwand des Horner-Schemas: O(m)

Rabin-Karp Algorithmus

- Analog zu p lassen sich auch die t_s mittels des Horner-Schemas berechnen.
- Aufwand zur Berechung aller t_s: $O((n-m+1)m)$

Rabin-Karp Algorithmus

- Analog zu p lassen sich auch die t_s mittels des Horner-Schemas berechnen.
- Aufwand zur Berechung aller t_s: $O((n-m+1)m)$
- ***Noch besser:***
  - Berechne nur t_0 mit Horner-Schema
  - Berechne t_{s+1} aus t_s durch

$$t_{s+1} = \#\Sigma \left(t_s - T[s+1] * \#\Sigma^{m-1} \right) + T[s+m+1]$$

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772 = $27*(17894 - 24*27^2)+26$

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772
15295 = $27*(10772 - 14*27^2)+13$

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894
10772
15295
19309 = $27*(15295 - 20*27^2)+4$

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```


Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```

Vorberechnung, einmaliger Aufwand von O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```

2 x Horner-Schema: O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```

Schleife wird O(n)-mal durchlaufen

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```


Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
    p  += (#Σ) * p + P[i];
    t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
    if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
    if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```

Muster nach rechts schieben: Aufwand O(1) vs. O(m)

Rabin-Karp Algorithmus

```
RabinKarpPre(T,P)
h = (#Σ)^(m-1);
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += (#Σ) * p + P[i];
   t  += (#Σ) * t + T[i];
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
        print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
        t = (#Σ) * (t – h*T[s+1])+T[s+m+1];
```

Nicht-uniformes Kostenmaß
⇒ Aufwand für Vergleich: O(m)

Exkurs: *Modulo*

- $a \equiv b \bmod q \;:\Leftrightarrow\; \exists\, k:\; a = b + k*q$
- $\forall a \in \mathbb{N}\;\; \exists b \in \{0,..,q-1\} :\; a \equiv b \bmod q$
- $(a+b) \bmod q = ((a \bmod q) + (b \bmod q)) \bmod q$
- $(a*b) \bmod q = ((a \bmod q) * (b \bmod q)) \bmod q$
- Es gilt

  $a = b \;\Rightarrow\; a \equiv b \bmod q$

  aber nicht umgekehrt!

Rabin-Karp Algorithmus

```
RabinKarp(T,P)
h = (#Σ)^(m-1) mod q;
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += (#Σ) * p + P[i] mod q;
   t  += (#Σ) * t + T[i] mod q;
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
      if (P[1..m] = T[s+1..s+m])
         print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
      t = (#Σ) * (t – h*T[s+1])+T[s+m+1] mod q;
```


Rabin-Karp Algorithmus

```
RabinKarp(T,P)
h = (#Σ)^(m-1) mod q;
p = 0;
t = 0;
for (i = 1 ; i ≤ m ; i++)
   p  += (#Σ) * p + P[i] mod q;
   t  += (#Σ) * t + T[i] mod q;
for (s = 0 ; s ≤ n – m ; s++)
   if (p = t)
      if (P[1..m] = T[s+1..s+m])
         print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
      t = (#Σ) * (t – h*T[s+1])+T[s+m+1] mod q;
```


Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

  $17894 \equiv 184 \bmod 253$

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

  $17894 \equiv 184 \bmod 253$

  $10772 \equiv 146 \bmod 253$

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894 ≡ 184 mod 253
10772 ≡ 146 mod 253
15295 ≡ 115 mod 253

Rabin-Karp Algorithmus

- T = „you_never_met_a_pet_as_wet_ ...“

17894 ≡ 184 mod 253
10772 ≡ 146 mod 253
15295 ≡ 115 mod 253
19309 ≡ 81 mod 253

Rabin-Karp Algorithmus: Aufwand

```
RabinKarp(T,P)
h = (#Σ)^(m-1) mod q;                                   } O(m)
p = 0;
t  = 0;                                                 } O(m)
for (i = 1 ; i ≤ m ; i++)
   p  += (#Σ) * p + P[ i ] mod q;
   t   += (#Σ) * t + T[ i ] mod q;
for (s = 0 ; s ≤ n – m ; s++)                           } O((n-m+1)m)
   if (p = t)
      if (P[1..m] = T[s+1..s+m])        } O(m)
         print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
      t = (#Σ) * (t – h*T[s+1])+T[s+m+1] mod q;
```


Rabin-Karp Algorithmus: Aufwand

```
RabinKarp(T,P)
h = (#Σ)^(m-1) mod q;                                   } O(m)
p = 0;
t  = 0;                                                 } O(m)
for (i = 1 ; i ≤ m ; i++)
   p  += (#Σ) * p + P[ i ] mod q;
   t   += (#Σ) * t + T[ i ] mod q;
for (s = 0 ; s ≤ n – m ; s++)                           } O(n log(q))
   if (p = t)                        } O(log(q))
      if (P[1..m] = T[s+1..s+m])     } O(m)
         print „Muster an der Stelle“ s+1 „gefunden“;
   if (s < n – m)
      t = (#Σ) * (t – h*T[s+1])+T[s+m+1] mod q;
```


Rabin-Karp Algorithmus: Zusammenfassung

Aufwand von RabinKarp:

- Worst case:
  O((n-m+1)*m)
- Average case: Ist q ≥ m und ist die erwartete Anzahl von Matches in O(1), dann hat RabinKarp einen erwarteten Aufwand von O(n)

Knuth-Morris-Pratt

- Geg.: Text T[1...n], Muster P[1..m]
- Beim naiven Ansatz und bei Rabin-Karp werden Symbole aus T[ ] *mehrfach* verglichen (bis zu m Mal !)
- Finde Algorithmus, der jedes T[i] nur *einmal* vergleicht ...

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

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

Knuth-Morris-Pratt

- Sei s der Index der äußeren Schleife und i der Index der inneren Schleife beim naiven String-Matching
- Dann gilt T[s+1 ... s+i] = P[1 ... i]
- Eine Verschiebung des Index s nach s+1 macht nur Sinn wenn
  T[s+2 ... s+i] = P[1 ... i-1]
  bzw.
  P[1 ... i-1] = P[2 ... i]

Knuth-Morris-Pratt

- Die Bedingung P[1 ... i-1] = P[2 ... i] hängt *nur* vom Suchmuster P[ ] ab, kann also *vorberechnet* werden.
- Genauso: Verschiebung auf s+2 macht nur Sinn, wenn P[1 ... i-2] = P[3 ... i]

Knuth-Morris-Pratt

- Allgemein: suche das *größte* j, für das P[1 ... j] = P[i-j+1 ... i] gilt.
- Dann ist s+i-j die nächste Möglichkeit für einen Match (nächste sinnvolle Verschiebung)

Knuth-Morris-Pratt

```
~~KnuthMorrisPratt(T[1...n],P[1...m])~~
~~    Q = PrefixFunction(P);~~
~~    j  = 0;~~
~~    for (i=1 ; i≤n ; i++)~~
~~        while ( j>0 && P[ j+1] ≠ T[ i ] )~~
~~            j = Q[ j ];~~
~~        if (P[ j+1] = T[ i ])  j++;~~
~~        if (j = m)~~
~~            print „Muster an der Stelle“ i-m „gefunden“;~~
~~            j = Q[j];~~
```

Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
    i = 1;  j = 1;  Q = PrefixFunction(P);
    while ( i < n )
        if ( j = 0  ∨  P[ j ] = T[ i ] )
            i++;   j++;
        else
            j = Q[ j ];
        if (j = m)
            print „Muster an der Stelle“ i-m „gefunden“;
            j = Q[ j ];
```

Knuth-Morris-Pratt

```
              ↓i
5 6 9 1 4 5 6 3 6 2 4 8 1 7 5 ...
        4 5 6 2 3
              ↑j
```

Knuth-Morris-Pratt

```
              ↓i
5 6 9 1 4 5 6 3 6 2 4 8 1 7 5 ...
        4 5 6 2 3
        ↑j
```

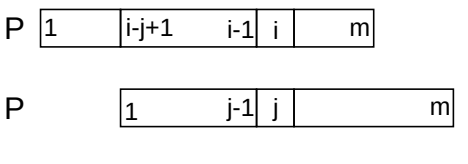
Knuth-Morris-Pratt

```
              ↓i
5 6 9 1 4 5 6 3 6 2 4 8 1 7 5 ...
              4 5 6 2 3
              ↑j
```

Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
    i = 1;  j = 1;  Q = PrefixFunction(P);
    while ( i < n )
        if ( j = 0  ∨  P[ j ] = T[ i ] )  → Index range!!!
            i++;   j++;
        else
            j = Q[ j ];
        if (j = m)
            print „Muster an der Stelle“ i-m „gefunden“;
            j = Q[ j ];
```

Knuth-Morris-Pratt



Knuth-Morris-Pratt

- Bedeutung von Q:
  - $Q[i] = j \Leftrightarrow P[1... j-1] = P[i-j+1 ... i-1]$ und es gibt kein größeres $j < i$
  - Insbesondere: $0 \le Q[i] < i$
  - Basisfall: $Q[1] = 0$
  - Aus $P[1... j-1] = P[i-j+1 ... i-1]$ und $P[j] = P[i]$ folgt $P[1... j] = P[i-j+1 ... i]$
  - Aus $Q[i] = j$ und $P[j] = P[i]$ folgt $Q[i+1] = j+1$

Knuth-Morris-Pratt

- Bedeutung von Q:
  - $Q[i] = j \Leftrightarrow P[1... j-1] = P[i-j+1 ... i-1]$ und es gibt kein größeres $j < i$
  - Insbesondere: $0 \le Q[i] < i$
  - Basisfall: $Q[1] = 0$
  - Aus $P[1... j-1] = P[i-j+1 ... i-1]$ und $P[j] \neq P[i]$ folgt $Q[i+1] \le Q[j]$

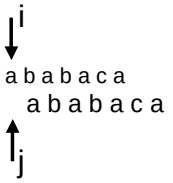
Knuth-Morris-Pratt

```
PrefixFunction(P[1...m])
    i = 1;  j = 0;  Q[1] = 0;
    while ( i < m )
        if ( j = 0  ∨  P[ i ] = P[ j ] )
            i++;  j++;  Q[ i ] = j;
        else
            j = Q[ j ];
```

Index range!!!

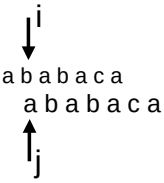
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0, ...]



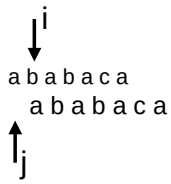
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1, ...]



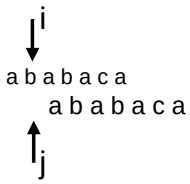
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1, ...]



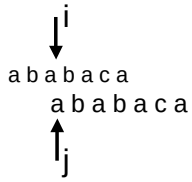
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1, ...]



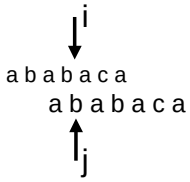
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1, ...]



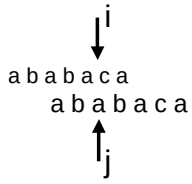
Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2, ...]



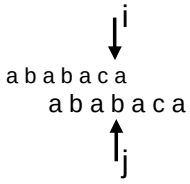
Knuth-Morris-Pratt

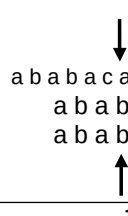
PrefixFunction() : Q[ ] = [0,1,1,2,3, ...]

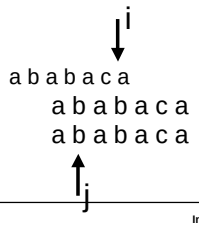


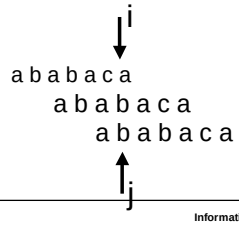
Knuth-Morris-Pratt

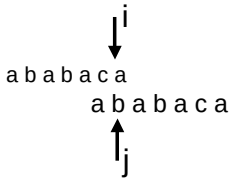
PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

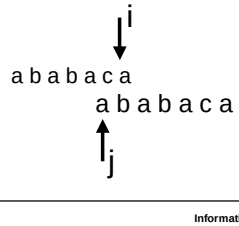


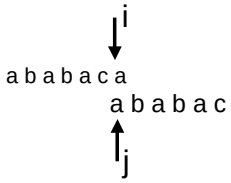












Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

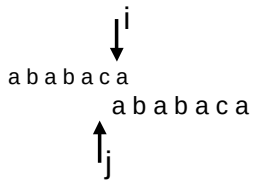
PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]



i

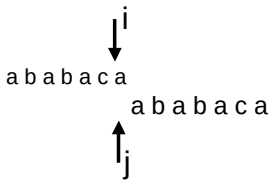
a b a b a c a

a b a b a c a

j

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4, ...]



i

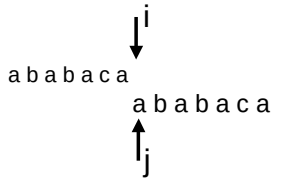
a b a b a c a

a b a b a c a

j

Knuth-Morris-Pratt

PrefixFunction() : Q[ ] = [0,1,1,2,3,4,1]



i

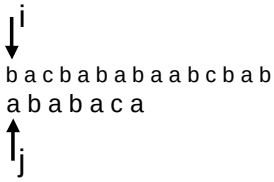
a b a b a c a

a b a b a c a

j

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



i

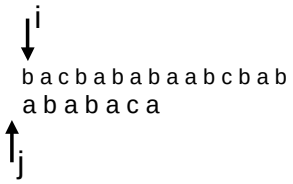
b a c b a b a b a a b c b a b

a b a b a c a

j

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



i

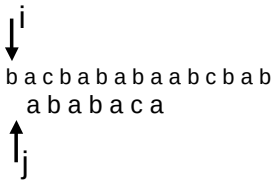
b a c b a b a b a a b c b a b

a b a b a c a

j

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

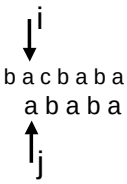


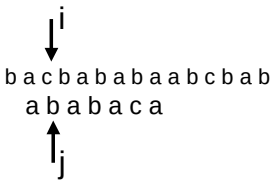
i

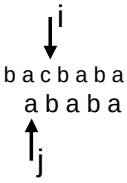
b a c b a b a b a a b c b a b

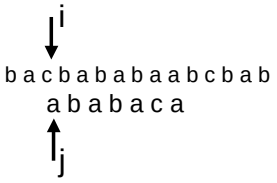
a b a b a c a

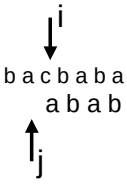
j

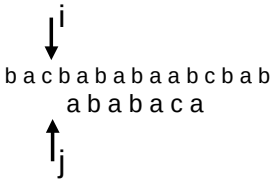












Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

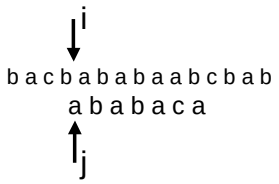
KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

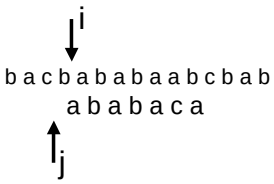
i

bacbababaabcbab

ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

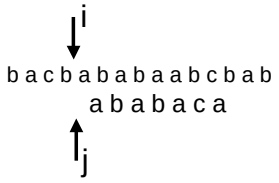
i

bacbababaabcbab

ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

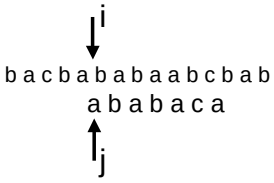
i

bacbababaabcbab

ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

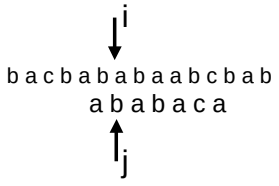
i

bacbababaabcbab

ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

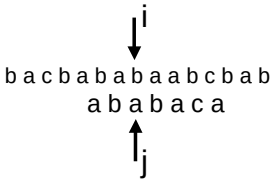
i

bacbababaabcbab

ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

i

bacbababaabcbab

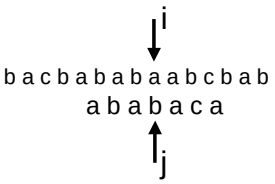
ababaca

j

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

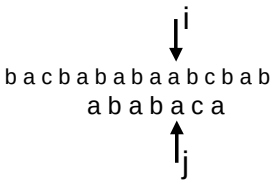
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



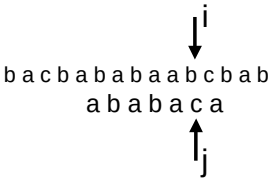
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



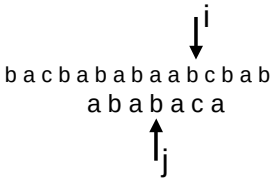
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



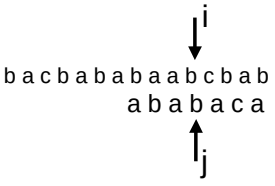
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



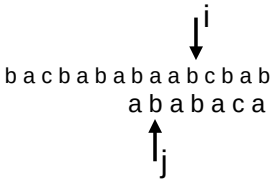
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



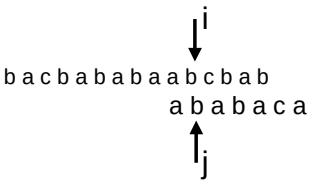
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



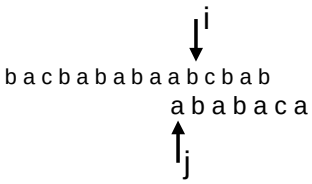
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



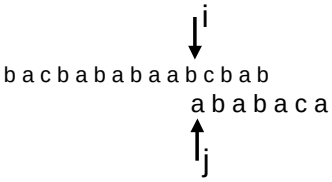
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



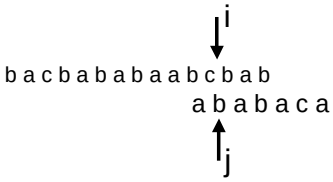
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



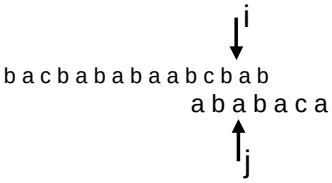
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



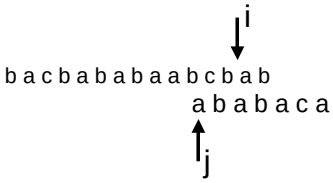
Knuth-Morris-Pratt

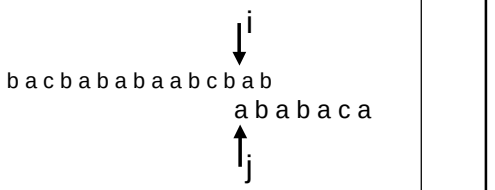
KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

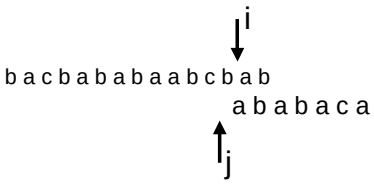


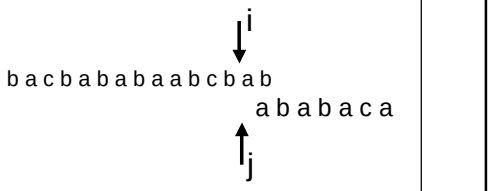
Knuth-Morris-Pratt

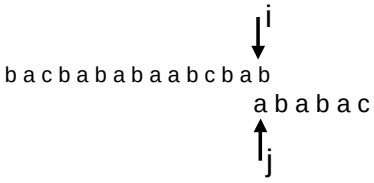
KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

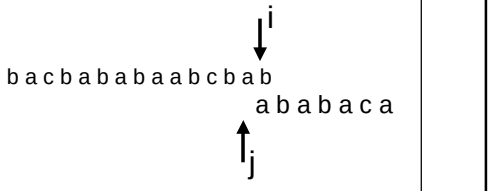


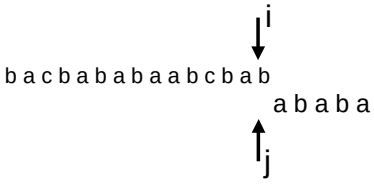












Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

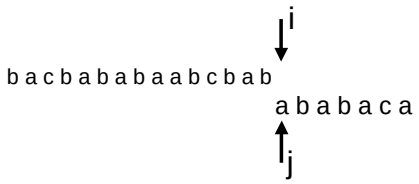
KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]

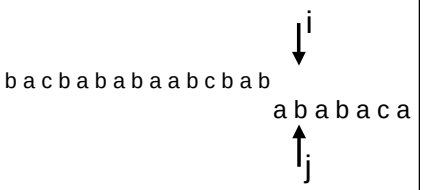
Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



Knuth-Morris-Pratt

KnuthMorrisPratt() : Q[ ] = [0,1,1,2,3,4,1]



Knuth-Morris-Pratt

```
KnuthMorrisPratt(T[1...n],P[1...m])
   i = 1;  j = 1;  Q = PrefixFunction(P);
   while ( i < n )
      if ( j = 0  ∨  P[ j ] = T[ i ] )
         i++;   j++;
      else
         j = Q[ j ];
      if (j = m)
         print „Muster an der Stelle“ i-m „gefunden“;
         j = Q[ j ];
```


Knuth-Morris-Pratt

- Aufwand
  - Die äußere Schleife wird mehr als n-mal durchlaufen !
  - If-Zweig wird maximal n-mal durchlaufen
  - Der Wert von j wird maximal n-mal erhöht
  - Der Else-Zweig verringert den Wert von j
  - Der Else-Zweig kann maximal n-mal durchlaufen werden **(Potential-Argument)**
  - Gesamtaufwand O(n)

Knuth-Morris-Pratt

- Aufwand
  - KnuthMorrisPratt() ... O(n)
  - PrefixFunction() ... O(m)
  - ... O(n+m)

Boyer-Moore Algorithmus

- KMP: nutze Vorwissen über die ***Gleichheit*** von Symbolen aus.
- BM: nutze Vorwissen über die ***Ungleichheit*** von Symbolen aus.
- Beide Ansätze können kombiniert werden

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

Naiv: a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

Heuristik: a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

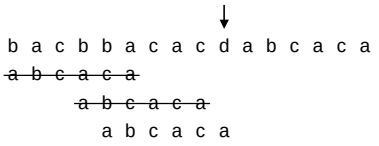
b a c b b a c a c d a b c a c a

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

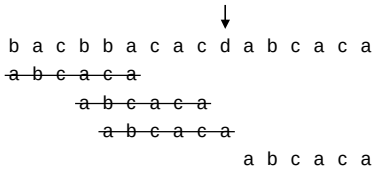
b a c b b a c a c d a b c a c a

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

b a c b b a c a c d a b c a c a

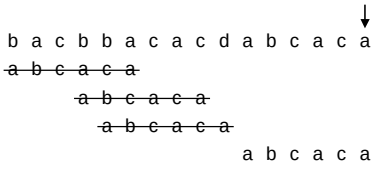
~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

b a c b b a c a c d a b c a c a

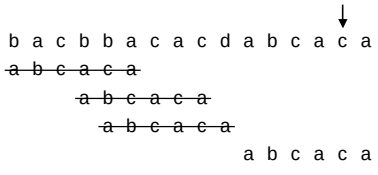
~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

b a c b b a c a c d a b c a c a

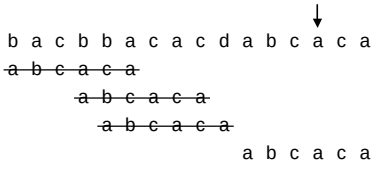
~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

b a c b b a c a c d a b c a c a

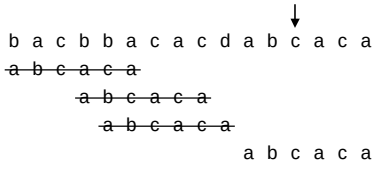
~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus



Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus

Vergleichsrichtung

b a c b b a c a c d a b c a c a

~~a b c a c a~~

~~a b c a c a~~

~~a b c a c a~~

a b c a c a

Boyer-Moore Algorithmus

Muster:

a b c a c a

Skip-Array:

a → 0	a b c a c a
b → 4	a b c a c a
c → 1	a b c a c a
d → 6	a b c a c a

Boyer-Moore Algorithmus

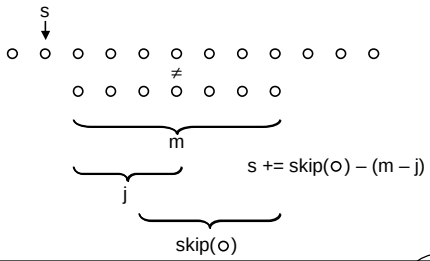
```
InitSkipArray(∑,P);
  for (each a ∈ ∑)                 } O(|∑|)
    skip[a] = m;
  for (j = 1; j ≤ m; ++j)          } O(m)
    skip[P[ j ]] = m – j;
```

„Länge minus letztes Auftreten eines Symbols“

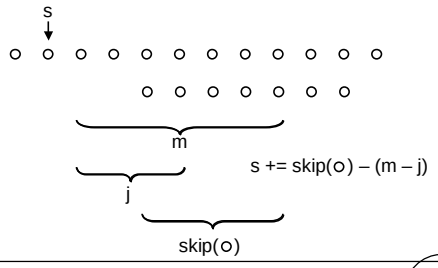
Boyer-Moore Algorithmus

```
BoyerMoore(T,P)
  InitSkipArray(∑,P);
  s = 0;
  while (s ≤ n – m)                          } O(n-m+1)
    j = m;
    while (j > 0 && P[ j ] = T[s+j])         } O(m)
      --j;
    if (j == 0)
      print “Muster an Stelle” s+1
      ++s;
    else
      s += max(1, skip[T[s+j]] – m + j);
```


Boyer-Moore Algorithmus



Boyer-Moore Algorithmus



Boyer-Moore Algorithmus: Beispiel

Alphabet: Σ = { a,b,c,d }

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

s j

b c b c b a a c a a a

a b c a a

Boyer-Moore Algorithmus: Beispiel

Alphabet: Σ = { a,b,c,d }

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

s j

b c b c b a a c a a a

a b c a a

s += max(1,skip(b) – (5 – 4))
= 2

Boyer-Moore Algorithmus: Beispiel

Alphabet: Σ = { a,b,c,d }

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

s j

b c b c b a a c a a a

a b c a a

j = 5

Boyer-Moore Algorithmus: Beispiel

Alphabet: Σ = { a,b,c,d }

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

s j

b c b c b a a c a a a

a b c a a

s += max(1,skip(c) – (5-5))
= 2

Boyer-Moore Algorithmus: Beispiel

Alphabet: Σ = { a,b,c,d }

Muster: a b c a a

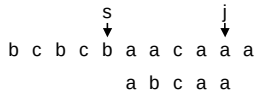
SkipArray:

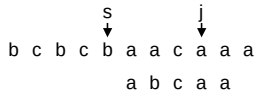
a	b	c	d
0	3	2	5

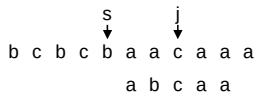
s j

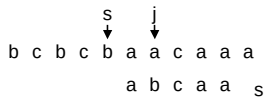
b c b c b a a c a a a

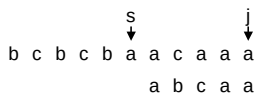
a b c a a











Boyer-Moore Algorithmus: Beispiel

Alphabet: $\Sigma = \{ a,b,c,d \}$

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

```
        s         j
        ↓         ↓
b c b c b a a c a a a
          a b c a a
```

Boyer-Moore Algorithmus: Beispiel

Alphabet: $\Sigma = \{ a,b,c,d \}$

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

```
        s         j
        ↓         ↓
b c b c b a a c a a a
          a b c a a
```

Boyer-Moore Algorithmus: Beispiel

Alphabet: $\Sigma = \{ a,b,c,d \}$

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

```
        s     j
        ↓     ↓
b c b c b a a c a a a
          a b c a a
```

Boyer-Moore Algorithmus: Beispiel

Alphabet: $\Sigma = \{ a,b,c,d \}$

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

```
        s   j
        ↓   ↓
b c b c b a a c a a a
          a b c a a
```

s += max(1,skip(a) – (5-2))
= max(1, -3) = 1

Boyer-Moore Algorithmus: Beispiel

Alphabet: $\Sigma = \{ a,b,c,d \}$

Muster: a b c a a

SkipArray:

a	b	c	d
0	3	2	5

```
          s         j
          ↓         ↓
b c b c b a a c a a a
            a b c a a
```

Boyer-Moore Algorithmus

Aufwand:

- Worst case: $O((n-m+1)m + |\Sigma|)$
  (konstante Folge, alle Skips = 1)
- In der Praxis ergibt sich eine ***signifikante*** Beschleunigung durch große Skip-Werte (vor allem bei langen Such-Strings)

Vergleich

- Naiver Algorithmus: O(n m)
- Rabin-Karp: O(n + m) *erwartet*
- Knuth-Morris-Pratt: O(n + m)
- Boyer-Moore: O(n m)
- BM wird in der Regel mit KMP kombiniert.

  $\Rightarrow$ schneller O(n + m) Algorithmus

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

(2.4.3) Hashing

- Suche ein bestimmtes Element in einer gegebenen Menge
  - Größe der Menge
  - Anzahl der Anfragen
  - Größe des Wertebereiches
  - ...

Beispiel-Problem

- Geg.: zwei Mengen $A = \{a_1, ..., a_m\}$ und $B = \{b_1, ..., b_n\}$
- Wertebereich: $p \le a_i, b_j \le q$
- Frage: $A \subseteq B$?
- Parameter: m, n, q-p

Naiver Algorithmus

- SubSet(A,B)

```
for (i = 1 ; i ≤ m ; i++)
    j = 1;
    while (j ≤ n ∧ A[ i ] ≠ B[ j ])
        j++;
    if (j > n)
        return (false);
return (true);
```


Naiver Algorithmus

- SubSet(A,B)

```
for (i = 1 ; i ≤ m ; i++)
    j = 1;
    while (j ≤ n ∧ A[ i ] ≠ B[ j ])      } O(n)
        j++;                             } O(n m)
    if (j > n)
        return (false);
return (true);
```

Sortieren und Suchen

- SubSetSort(A,B)

```
sort(B);
for (i = 1 ; i ≤ m ; i++)
    if (!find(A[ i ],B))
        return (false);
return (true);
```


Sortieren und Suchen

Aufwand: O(n)

- Find(a,B)

```
for (i = 1 ; i ≤ n ; i++)
    if (a = B[ i ])
        return (true);
return(false);
```


Sortieren und Suchen

Aufwand: O(log(n))

- Find(a,B)

```
l = 1;  r = n;
while (l ≤ r)
    m = (l+r) / 2;
    if      (a < B[m])  r = m-1;
    elseif (a > B[m])  l = m+1;
    else  return (true);
return(false);
```


Sortieren und Suchen

- SubSetSort(A,B)

```
sort(B);                          } O(n log(n))
for (i = 1 ; i ≤ m ; i++)
    if (!find(A[ i ],B))          } O(log(n) )   } O(m log(n))
        return (false);
return (true);
```

Gesamtaufwand: O((n+m) log(n))

Kleiner Wertebereich

- Wenn der Wertebereich klein ist, können Mengen als ***Array of Bool*** implementiert werden.
- setA[0, ..., q-p+1] = { true, false }
- z.B. setze Bit-Werte auf „1“ oder „0“

Kleiner Wertebereich

- SubSet(A,B)

```
setA = Empty(q-p+1);
setB = Empty(q-p+1);
for (i = 1 ; i ≤ m ; i++)
    setA[A[ i ]-p] = true;
for (i = 1 ; i ≤ n ; i++)
    setB[B[ i ]-p] = true;
for (i = 0 ; i ≤ q-p ; i++)
    if (setA[ i ] = true ∧ setB[ i ] = false)
        return (false);
return (true);
```

Kleiner Wertebereich

- SubSet(A,B)

```
setA = Empty(q-p+1);
setB = Empty(q-p+1);                          } O(q-p)
for (i = 1 ; i ≤ m ; i++)
    setA[A[ i ]-p] = true;                    } O(m)
for (i = 1 ; i ≤ n ; i++)
    setB[B[ i ]-p] = true;                    } O(n)
for (i = 0 ; i ≤ q-p ; i++)
    if (setA[ i ] = true ∧ setB[ i ] = false) } O(q-p)
        return (false);
return (true);
```


Kleiner Wertebereich

- Gesamtaufwand:

  $O(n+m+(q-p)) = O(n+m)$

- Wertebereich wird als ***konstant*** angenommen.
- Anwendung bei ***Mehrfachmengen***

Großer Wertebereich

- z.B. Matrikelnummern : 6-stellig
- Große Wertebereiche sind in der Regel ***nicht dicht*** besetzt
- Bilde den Wertebereich auf eine (kleinere) Index-Menge ab.

Hash-Funktion

- Hash-Funktion ... $F : W \rightarrow I$
- Schlüssel ... $F(w) = i$
- Effiziente Suche in der Index-Menge I, wenn die Funktion F einfach ist.

Hash-Funktion

- Index-Menge in der Regel deutlich kleiner als der Wertebereich
- Kollisionen: $F(w) = F(w') \text{ obwohl } w \neq w'$
- Wie häufig sind Kollisionen ?

Kollisionen

- Identifiziere Studenten über die Matrikelnummer ... 10^6 Möglichkeiten
- Studenten in dieser Vorlesung ≤ 800
- Finde eine Funktion F, die den Bereich $[0..10^6\text{-}1]$ möglichst gleichverteilt in den Bereich [0..799] abbildet.

  (z.B. Matrikelnummer mod 800)

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

Rabin-Karp Algorithmus

- Kodiere Folgen von Symbolen als Zahlen
- Zwei Worte aus Σ^* sind gleich, wenn die entsprechenden Zahlen gleich sind
  $\Rightarrow$ Vergleich von Zahlen statt Vergleich von Worten (mehrere Ziffern parallel)
- Schnelltest durch Berechnungen mod q

Knuth-Morris-Pratt

- Geg.: Text T[1...n], Muster P[1..m]
- Beim naiven Ansatz und bei Rabin-Karp werden Symbole aus T[ ] *mehrfach* verglichen (bis zu m Mal !)
- Finde Algorithmus, der jedes T[i] nur *einmal* vergleicht ... ***Präfixtabelle***

Boyer-Moore Algorithmus

- KMP: nutze Vorwissen über die ***Gleichheit*** von Symbolen aus.
- BM: nutze Vorwissen über die ***Ungleichheit*** von Symbolen aus.
- Beide Ansätze können kombiniert werden

Vergleich

- Naiver Algorithmus: O(n m)
- Rabin-Karp: O(n + m) *erwartet*
- Knuth-Morris-Pratt: O(n + m)
- Boyer-Moore: O(n m)
- BM wird in der Regel mit KMP kombiniert.
  $\Rightarrow$ schneller O(n + m) Algorithmus

(2.4) Suchen

- (2.4.1) k-Selektion
- (2.4.2) String-Suche
- (2.4.3) Hashing

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

(2.4.3) Hashing

- Suche ein bestimmtes Element in einer gegebenen ***Menge***
  - Größe der Menge
  - Anzahl der Anfragen
  - Größe des Wertebereiches
  - ...

Beispiel-Problem

- Geg.: zwei Mengen $A = \{a_1, ..., a_m\}$ und $B = \{b_1, ..., b_n\}$
- Wertebereich: $p \le a_i, b_j \le q$
- Frage: $A \subseteq B$?
- Parameter: m, n, q-p

Naiver Algorithmus

```
• SubSet(A,B)
    for (i = 1 ; i ≤ m ; i++)
        j = 1;
        while (j ≤ n ∧ A[ i ] ≠ B[ j ])
            j++;
        if (j > n)
            return (false);
    return (true);
```


Naiver Algorithmus

```
• SubSet(A,B)
    for (i = 1 ; i ≤ m ; i++)
        j = 1;
        while (j ≤ n ∧ A[ i ] ≠ B[ j ])      } O(n)
            j++;                             } O(n m)
        if (j > n)
            return (false);
    return (true);
```


Sortieren und Suchen

```
• SubSetSort(A,B)
    sort(B);
    for (i = 1 ; i ≤ m ; i++)
        if (!find(A[ i ],B))
            return (false);
    return (true);
```


Halbierungssuche

Aufwand: O(log(n))

```
• Find(a,B)
    l = 1;  r = n;
    while (l ≤ r)
        m = (l+r) / 2;
        if      (a < B[m])  r = m-1;
        elseif (a > B[m])  l = m+1;
        else  return (true);
    return(false);
```

Sortieren und Suchen

```
• SubSetSort(A,B)
      sort(B);                          } O(n log(n))
      for (i = 1 ; i ≤ m ; i++)
          if (!find(A[ i ],B))          } O(log(n))   } O(m log(n))
              return (false);
      return (true);
```

Gesamtaufwand: O((n+m) log(n))

Kleiner Wertebereich

- Wenn der Wertebereich klein ist, können Mengen als ***Array of Bool*** implementiert werden.
- setA[p, ..., q] = { true, false }
- z.B. setze Bit-Werte auf „1“ oder „0“

Kleiner Wertebereich

```
• SubSet(A,B)
      setA = Empty(p,q);
      setB = Empty(p,q);
      for (i = 1 ; i ≤ m ; i++)
          setA[A[ i ]] = true;
      for (i = 1 ; i ≤ n ; i++)
          setB[B[ i ]] = true;
      for (i = p ; i ≤ q ; i++)
          if (setA[ i ] = true ∧ setB[ i ] = false)
              return (false);
      return (true);
```


Kleiner Wertebereich

```
• SubSet(A,B)
      setA = Empty(p,q);                } O(q-p)
      setB = Empty(p,q);
      for (i = 1 ; i ≤ m ; i++)         } O(m)
          setA[A[ i ]] = true;
      for (i = 1 ; i ≤ n ; i++)         } O(n)
          setB[B[ i ]] = true;
      for (i = p ; i ≤ q ; i++)
          if (setA[ i ] = true ∧ setB[ i ] = false)   } O(q-p)
              return (false);
      return (true);
```


Kleiner Wertebereich

```
• SubSet(A,B)
      setB = Empty(p,q);                } O(q-p)

      for (i = 1 ; i ≤ n ; i++)         } O(n)
          setB[B[ i ]] = true;
      for (i = 1 ; i ≤ m ; i++)
          if (setB[A[ i ]] = false)     } O(m)
              return (false);
      return (true);
```


Kleiner Wertebereich

- Gesamtaufwand:

  O(n+m+(q-p)) = O(n+m)

- Wertebereich wird als ***konstant*** (und klein) angenommen.
- Anwendung bei ***Mehrfachmengen***

Mittlerer Wertebereich

- Implementierung des Datentyps ***Menge*** als ***ArrayOfBool*** ist elegant, da Einfügen, Zugriff und Löschen in **O(1)** realisiert werden können.
- Probleme
  - Speicherbedarf O(q-p)
  - Initialisierung O(q-p)

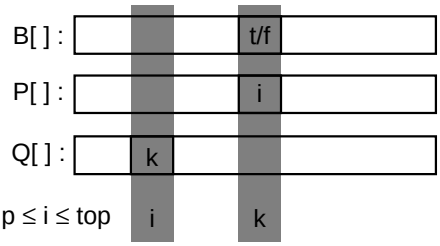
Lazy Initialization

- Verwende:
  - array of bool: B[p...q]
  - array of int: P[p...q]
  - array of int: Q[p...q]
  - int top

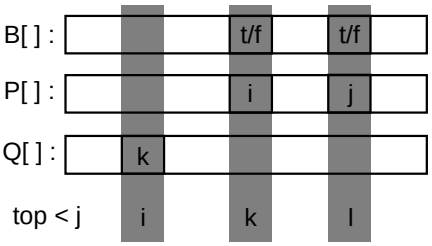
Lazy Initialization

- top: Anzahl der bereits initialisierten Einträge im Array B[ ]
- P[ ], Q[ ]: zeigen an, ob der entsprechende Eintrag bereits initialisiert ist
  - P[ ] : in welchem Insert/Delete Schritt
  - Q[ ] : Bestätigung, dass P[ ] gültig ist

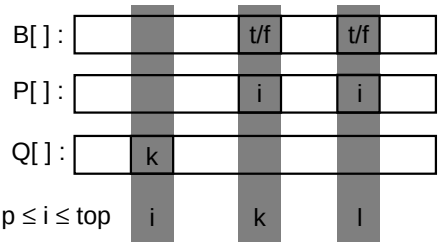
Lazy Initialization



Lazy Initialization



Lazy Initialization



Lazy Initialization

- Init()
  top = p-1;

Lazy Initialization

- Insert(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
    B[k] = true;
  else
    top++;
    P[k] = top;
    Q[top] = k;
    B[k] = true;

Lazy Initialization

- Delete(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
    B[k] = false;
  else
    top++;
    P[k] = top;
    Q[top] = k;
    B[k] = false;

Lazy Initialization

- Search(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
    return B[k];
  return (false);

Großer Wertebereich

- z.B. Matrikelnummern : 6-stellig
  Filenamen: $(26+10+3)^8 = 10^{12}$
- Große Wertebereiche sind in der Regel ***nicht dicht*** besetzt
- Bilde den Wertebereich auf eine (kleinere) Index-Menge ab.

Hashing

- Speichere Elemente $a_1 \dots a_n$ mit $a_i \in W$ in einer Tabelle der Größe O(n)
- Hash-Funktion ... $F : W \rightarrow J$
  - $W \subseteq$ IN: beliebig großer Wertebereich
  - $J \subseteq$ IN: Hash-Tabelle
- Schlüssel ... F(w) = i
- Wähle F so, dass die Schlüssel $F(a_i)$ ***gleichverteilt*** in J liegen.

Hashing

- Bucket-Sort
  - Gleichverteilte Schlüssel
  - Berechne Speicheradresse aus Schlüssel
  - Erwarteter Füllungsgrad der Buckets O(1)
- Hashing
  - Berechne Index-Adresse aus dem Schlüssel
  - Wähle die Größe der Indexmenge in der Größenordnung der Anzahl der Schlüssel
  - Erzeuge Gleichverteilung durch „Streuung“

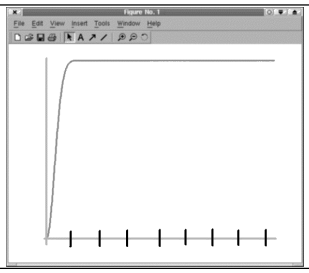
Hash-Funktion

- Index-Menge J in der Regel deutlich kleiner als der Wertebereich W
- Kollisionen: F(w) = F(w‘) obwohl w ≠ w‘
- Wie häufig sind Kollisionen ?

Beispiel

- Identifiziere Studenten über die Matrikelnummer ... 10^6 Möglichkeiten
- Studenten in dieser Vorlesung ≤ 800
- Finde eine Funktion F, die den Bereich [0..10^6-1] möglichst gleichverteilt in den Bereich [0..799] abbildet.
  (z.B. Matrikelnummer mod 800)

Kollisionen



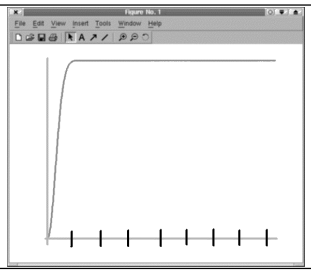
Kollisionen

- Kollisionswahrscheinlichkeit 50% schon bei 34 Studenten erreicht !
- Bei 50 Studenten bereits 79%
- Vgl. ***Geburtstagsparadoxon***

Kollisionen

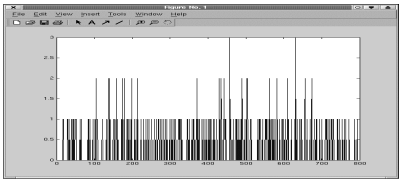
- Wahrscheinlichkeit für Kollisions***freiheit***
  - 1. Student: P_1 = 800/800
  - 2. Student: P_2 = 800/800 * 799/800
  - 3. Student: P_3 = 800/800 * 799/800 * 798/800
  - ...
  - k. Student: P_k = 800/800 * ... * (801-k)/800
- Kollisionswahrscheinlichkeit: Q_k = 1- P_k

Kollisionen



Kollisionen

- 226 Studenten haben schon mal ein Übungsblatt abgegeben ...



Kollisionen

- 226 Studenten haben schon mal ein Übungsblatt abgegeben ...
  - 20 einfache Kollisionen
  - 2 mehrfache Kollisionen
  - Obwohl Tabelle nur zu 28% gefüllt

Hashing

- Finde gute Hash-Funktionen
  - Surjektivität
  - Gute Streuung auch bei kohärenten Originalschlüsseln
- Kollisionsbehandlung
  - geschlossen
  - offen

Hash-Funktionen

- ***Perfekte*** Hash-Funktion
  - $S \subseteq W = \{ 0, ..., N-1 \}$, $|W| = N$, $|S| = n$
  - Finde eine Funktion F, die die Elemente von S ***injektiv*** auf die Indexmenge $\{ 0, ..., m-1 \}$ abbildet ($m \geq n$)
  - Existiert immer, aber schwer zu finden
  - Für $m \geq 3n$ in $O(n N)$

Hash-Funktionen

- ***Universelle*** Hash-Funktion
  - Erwarteter Aufwand für Elementzugriff O(1)
  - Worst-case Aufwand O(n) *(vgl. Bucket-Sort)*
  - Hängt von der Struktur der Menge $S \subseteq W$ ab
  - Definiere eine Menge $H = \{ F_i() \}$ von Hash-Funktionen, so dass die Wahrscheinlichkeit für $F_i(w) = F_i(w')$ und $F_j(w) = F_j(w')$ bei $w \neq w'$ und $i \neq j$ kleiner als 1/m ist. (*m = Größe der Indexmenge*)
  - Wähle Hash-Funktion aus dieser Menge zufällig aus.

Hash-Funktionen

- Präfix, Suffix ...
- F(w) = w mod q
  - $q = 2^k$ ... benutzt nur die letzten k Binärstellen
  - $q = 10^k$ ... benutzt nur die letzten k Dezimalstellen
  - q = Primzahl ... Optimale Streuung
- F(w) = (w*s + t) mod q
  – Beliebige q möglich, aber Primzahlen besser

Hash-Funktionen

- Problem: Tabellengröße ≠ Primzahl
  - F(w) = (w*s + t) mod q)
    (q große Primzahl)
  - G(w) = w mod q‘
    (q‘ Tabellengröße)
  - G(F(w))

Hash-Funktionen

- Mittel-Quadrat-Methode
  - Seien die Originalschlüssel *p-stellige* Dezimalzahlen
  - Quadrate sind *2p-stellig*
  - Tabellengröße 10^q, $q < p$
  - $F(w) = \lfloor 10^{-\lceil p/2 \rceil} w^2 \rfloor \bmod 10^q$
  - Hängt von allen p Stellen ab

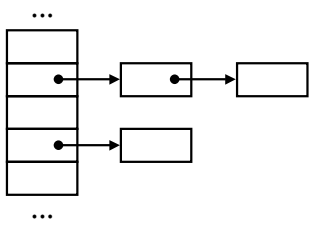
Kollisionsbehandlung

- ***Offenes*** Hashing
  (geschlossene Adressierung)
- ***Geschlossenes*** Hashing
  (offene Adressierung)

Offenes Hashing

- Jeder Tabelleneintrag ist Anchor-Element einer verketteten Liste
- Alle Objekte mit gleichem Hash-Key werden in die entsprechende Liste eingefügt.
- *Offen* : es wird dynamisch mehr Speicherplatz belegt
- *Geschlossen* : es werden keine alternativen Adressen berechnet.

Offenes Hashing



Offenes Hashing

- Die erwartete Länge der Listen bei Belegungsfaktor **α = n / m** ist ebenfalls gleich **α**
  (**n**: Anzahl der Objekte, **m**: Größe der Tabelle)
- Vgl. Analyse von Bucket-Sort
- Erwarteter Aufwand: O(1+ **α**)
  - α << 1 ... „Array"-Verhalten
  - α ≈ 1
  - α >> 1 ... „Listen"-Verhalten

Geschlossenes Hashing

- Wenn eine Kollision auftritt (weil der adressierte Tabellenplatz schon belegt ist) wird ein alternativer Schlüssel berechnet.
- *Geschlossen*: es wird kein neuer Speicherplatz belegt
- *Offen*: es werden andere Adressen verwendet

Geschlossenes Hashing

- Anstatt nur den Index F(w) zu verwenden, wird eine Sequenz von Indizes F(w,0), F(w,1), ..., F(w,m-1) berechnet.
- Das Objekt w wird im ersten freien Tabellenplatz dieser Sequenz gespeichert.
- *Nachteil:* Die Sequenz muß bei jedem Zugriff durchsucht werden.

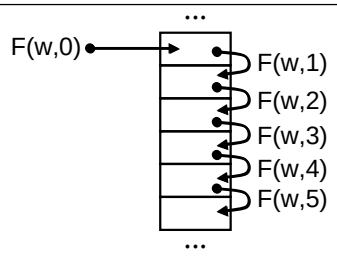
Geschlossenes Hashing

- *Nachteil:* Die Sequenz muß bei jedem Zugriff durchsucht werden.
  - Positiver Fall ... erwartet bis zur Hälfte
  - Negativer Fall ... immer bis zum Ende
- Die Sequenzen sind typischerweise ***länger*** als die Listen beim offenen Hashing !!!

Kollisionsauflösung

- Lineares Sondieren
  F(w,i) = (F(w) + i) mod m
- Führt in der Regel zur ***Cluster***-Bildung, da jedes Element, das mit einem kleinen Cluster kollidiert, an dessen Ende angefügt wird und damit den Cluster vergrößert.

Lineares Sondieren



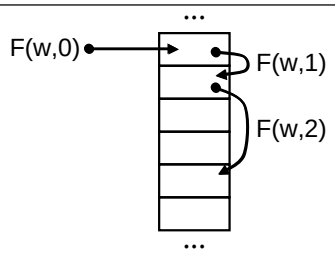
Kollisionsauflösung

- Quadratisches Sondieren
  $F(w,i) = (F(w) + i^2) \bmod m$
- Besseres Streuverhalten, insbesondere wenn m eine Primzahl ist.

Kollisionsauflösung

- Quadratisches Sondieren
  $F(w,i) = (F(w) + i^2) \bmod m$
- Besseres Streuverhalten, insbesondere wenn m eine Primzahl ist.
- Noch besser: m Primzahl mit $m \equiv 3 \bmod 4$
  $F(w,2i+1) = (F(w) + i^2) \bmod m$
  $F(w,2i) = (F(w) - i^2) \bmod m$

Lineares Sondieren



Kollisionsauflösung

- Doppeltes Hashing
  $F(w,i) = (F_1(w) + F_2(w) * i^2) \bmod m$
- Fast ideales Verhalten
  - Kollisionswahrscheinlichkeiten:

    $Prob(F_1(w) \equiv F_1(w') \bmod m) = 1/m$
    $Prob(F_2(w) \equiv F_2(w') \bmod m) = 1/m$
    $Prob(F_1(w) \equiv F_1(w') \bmod m \wedge F_2(w) \equiv F_2(w') \bmod m) = 1/m^2$

Kollisionsauflösung

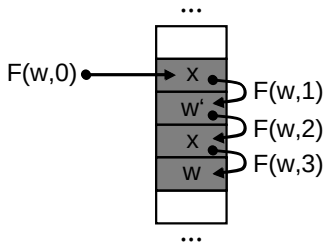
- Doppeltes Hashing
  $F(w,i) = (F_1(w) + F_2(w) * i^2) \bmod m$
- Fast ideales Verhalten
  - Kollisionswahrscheinlichkeiten:

    $Prob(F_1(w) \equiv F_1(w') \bmod m) = 1/m$
    $Prob(F_2(w) \equiv F_2(w') \bmod m) = 1/m$
    $Prob(F_1(w) + F_2(w) \equiv F_1(w') + F_2(w') \bmod m) = 1/m$

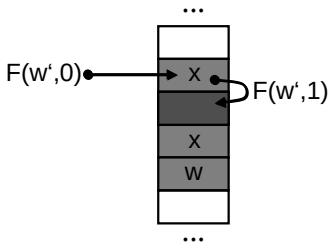
Löschen

- Bei offenem Hashing einfach
- Bei geschlossenem Hashing
  - Markiere Tabelleneinträge als ***gelöscht***
  - Gelöschte Einträge können überschrieben werden
  - Gelöschte Einträge werden beim Suchen berücksichtigt

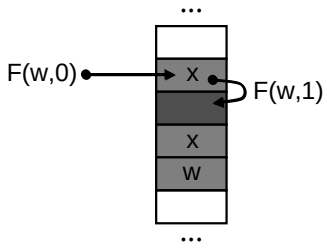
Löschen



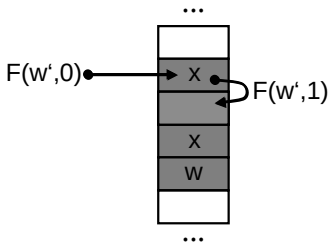
Löschen



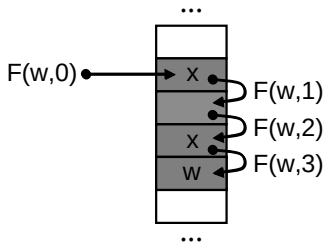
Löschen



Löschen



Löschen



Aufwandsabschätzung

- Wie viele Kollisionen treten auf?
- Wie oft muß sondiert werden?
- Annahme: ideale Hash-Funktion (aber nicht perfekt)
- Schlüssel werden gleichverteilt

Aufwandsabschätzung

- Q(i,n,m) ... Wahrscheinlichkeit, dass
  - *mindestens* i Sondierungsschritte
  - bei bereits n Elementen
  - in einer Tabelle der Größe m

  notwendig sind.

Aufwandsabschätzung

- Q(0,n,m) = 1
- Q(1,n,m) = n / m
- Q(2,n,m) = n / m * (n-1) / (m-1)
- Q(3,n,m) = ...
- Q(i,n,m) = n / m * ... * (n+1-i) / (m+1-i)

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elementes:

$$C_{ins}(n,m) = \sum_{j=0}^{n} (j+1)\,(Q(j,n,m) - Q(j+1,n,m))$$

$$= \sum_{j=0}^{n} (j+1)\,Q(j,n,m) - \sum_{j=1}^{n+1} j\,Q(j,n,m)$$

$$= \sum_{j=0}^{n} Q(j,n,m) - (n+1)\,Q(n+1,n,m)$$

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elementes:

$$C_{ins}(n,m) = \sum_{j=0}^{n} Q(j,n,m) - (n+1)\,Q(n+1,n,m)$$

$$= \sum_{j=0}^{n} Q(j,n,m)$$

$$= \frac{m+1}{m+1-n}$$

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elementes:

$$C_{ins}(n,m) = \frac{m+1}{m+1-n}$$

- Sei $\alpha = n / m$, dann gilt

$$C_{ins}(n,m) \approx \frac{1}{1-\alpha}$$

Aufwandsabschätzung

$$C_{ins}(n,m) \approx \frac{1}{1-\alpha}$$

α =	50%	70%	90%	95%	99%	99.9%
C_{ins} =	2	3.3	10	20	100	1000

(2.4.3) Hashing

- Suche ein bestimmtes Element in einer gegebenen ***Menge***
  - Größe der Menge
  - Anzahl der Anfragen
  - Größe des Wertebereiches
  - ...

Kleiner Wertebereich

- Wenn der Wertebereich klein ist, können Mengen als ***Array of Bool*** implementiert werden.
- setA[p, ..., q] = { true, false }
- z.B. setze Bit-Werte auf „1“ oder „0“

Kleiner Wertebereich

- SubSet(A,B)

```
setB = Empty(p,q);                    } O(q-p)

for (i = 1 ; i ≤ n ; i++)             } O(n)
    setB[B[ i ] ] = true;
for (i = 1 ; i ≤ m ; i++)             } O(m)
    if (setB[A[ i ] ] = false)
        return (false);
return (true);
```


Kleiner Wertebereich

- Gesamtaufwand:

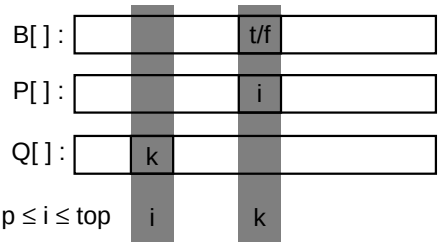
  $O(n+m+(q-p)) = O(n+m)$

- Wertebereich wird als ***konstant*** (und klein) angenommen.
- Anwendung bei ***Mehrfachmengen***

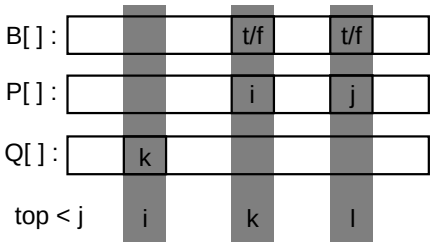
Mittlerer Wertebereich

- Implementierung des Datentyps ***Menge*** als ***ArrayOfBool*** ist elegant, da Einfügen, Zugriff und Löschen in **O(1)** realisiert werden können.
- Probleme
  - Speicherbedarf O(q-p)
  - Initialisierung O(q-p)

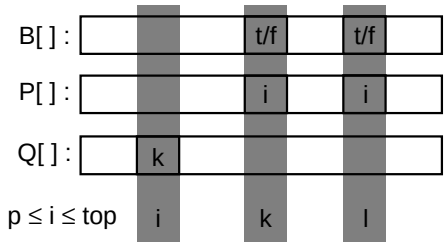
Lazy Initialization



Lazy Initialization



Lazy Initialization



Lazy Initialization

- Init()
  top = p-1;

Lazy Initialization

- Insert(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
  B[k] = true;
  else
  top++;
  P[k] = top;
  Q[top] = k;
  B[k] = true;

Lazy Initialization

- Delete(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
  B[k] = false;
  else
  top++;
  P[k] = top;
  Q[top] = k;
  B[k] = false;

Lazy Initialization

- Search(k)
  if ($p \le P[k] \le top \wedge Q[P[k]] = k$)
  return B[k];
  return (false);

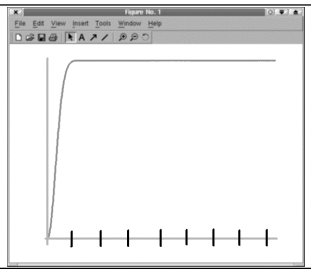
Hashing

- Speichere Elemente a_1 ... a_n mit $a_i \in W$ in einer Tabelle der Größe O(n)
- Hash-Funktion ... F : W $\rightarrow$ J
  - W$\subseteq$ IN: beliebig großer Wertebereich
  - J $\subseteq$ IN: Hash-Tabelle
- Schlüssel ... F(w) = i
- Wähle F so, dass die Schlüssel F(a_i) ***gleichverteilt*** in J liegen.

Kollisionen

- Wahrscheinlichkeit für Kollisions***freiheit***
  - Tabellengröße: n
  - erster Eintrag : $P_1 = n / n$
  - zweiter Eintrag : $P_2 = n / n * (n-1)/n$
  - ...
  - k-ter Eintrag : $P_k = n / n * ... * (n+1-k) / n$
- Kollisionswahrscheinlichkeit: $Q_k = 1 - P_k$

Kollisionen



Hashing

- Finde gute Hash-Funktionen
  - Surjektivität
  - Gute Streuung auch bei kohärenten Originalschlüsseln
- Kollisionsbehandlung
  - geschlossen
  - offen

Hash-Funktionen

- ***Perfekte*** Hash-Funktion
  - S $\subseteq$ W = { 0, ..., N-1 }, |W| = N, |S| = n
  - Finde eine Funktion F, die die Elemente von S ***injektiv*** auf die Indexmenge { 0, ..., m-1 } abbildet (m $\geq$ n)
  - Existiert immer, aber schwer zu finden
  - Für m $\geq$ 3n in O(n N)

Hash-Funktionen

- Präfix, Suffix ...
- F(w) = w mod q
  - q = 2^k ... benutzt nur die letzten k Binärstellen
  - q = 10^k ... benutzt nur die letzten k Dezimalstellen
  - q = Primzahl ... Optimale Streuung
- F(w) = (w*s + t) mod q
  - Beliebige q möglich, aber Primzahlen besser

Hash-Funktionen

- Problem: Tabellengröße ≠ Primzahl
  - F(w) = (w*s + t) mod q
    (q große Primzahl)
  - G(w) = w mod q‘
    (q‘ Tabellengröße)
  - G(F(w))

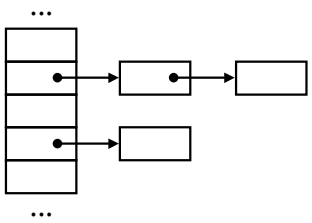
Kollisionsbehandlung

- ***Offenes*** Hashing
  (geschlossene Adressierung)
- ***Geschlossenes*** Hashing
  (offene Adressierung)

Offenes Hashing

- Jeder Tabelleneintrag ist Anchor-Element einer verketteten Liste
- Alle Objekte mit gleichem Hash-Key werden in die entsprechende Liste eingefügt.
- *Offen* : es wird dynamisch mehr Speicherplatz belegt
- *Geschlossen* : es werden keine alternativen Adressen berechnet.

Offenes Hashing



Offenes Hashing

- Die erwartete Länge der Listen bei Belegungsfaktor $\alpha = n / m$ ist ebenfalls gleich α
  (**n**: Anzahl der Objekte, **m**: Größe der Tabelle)
- Vgl. Analyse von Bucket-Sort
- Erwarteter Aufwand: $O(1+\alpha)$
  - $\alpha << 1$ ... „Array“-Verhalten
  - $\alpha \approx 1$
  - $\alpha >> 1$ ... „Listen“-Verhalten

Geschlossenes Hashing

- Wenn eine Kollision auftritt (weil der adressierte Tabellenplatz schon belegt ist) wird ein alternativer Schlüssel berechnet.
- *Geschlossen*: es wird kein neuer Speicherplatz belegt
- *Offen*: es werden andere Adressen verwendet

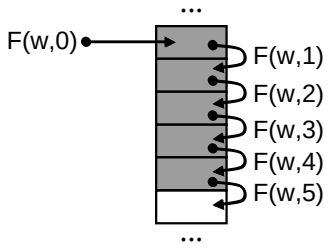
Geschlossenes Hashing

- Anstatt nur den Index F(w) zu verwenden, wird eine Sequenz von Indizes F(w,0), F(w,1), ..., F(w,m-1) berechnet.
- Das Objekt w wird im ersten freien Tabellenplatz dieser Sequenz gespeichert.
- *Nachteil:* Die Sequenz muß bei jedem Zugriff durchsucht werden.

Kollisionsauflösung

- Lineares Sondieren
  $F(w,i) = (F(w) + i) \bmod m$
- Führt in der Regel zur ***Cluster***-Bildung, da jedes Element, das mit einem kleinen Cluster kollidiert, an dessen Ende angefügt wird und damit den Cluster vergrößert.

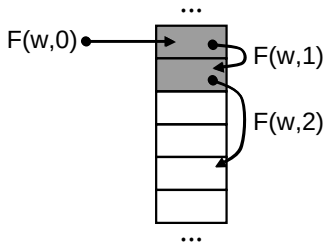
Lineares Sondieren



Kollisionsauflösung

- Quadratisches Sondieren
  $F(w,i) = (F(w) + i^2) \bmod m$
- Besseres Streuverhalten, insbesondere wenn m eine Primzahl ist.
- Noch besser: m Primzahl mit $m \equiv 3 \bmod 4$
  $F(w,2i+1) = (F(w) + i^2) \bmod m$
  $F(w,2i) = (F(w) - i^2) \bmod m$

Lineares Sondieren



Kollisionsauflösung

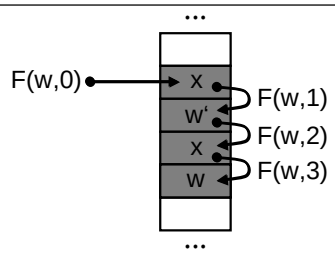
- Doppeltes Hashing
  $F(w,i) = (F_1(w) + F_2(w) * i^2) \bmod m$
- Fast ideales Verhalten
  - Kollisionswahrscheinlichkeiten:

    $Prob(F_1(w) \equiv F_1(w') \bmod m) = 1/m$
    $Prob(F_2(w) \equiv F_2(w') \bmod m) = 1/m$
    $Prob(F_1(w) + F_2(w) \equiv F_1(w') + F_2(w') \bmod m) = 1/m$

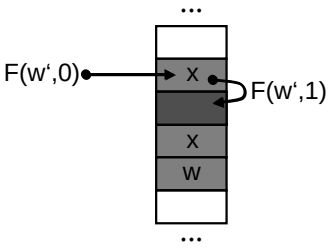
Löschen

- Bei offenem Hashing einfach
- Bei geschlossenem Hashing
  - Markiere Tabelleneinträge als ***gelöscht***
  - Gelöschte Einträge können überschrieben werden
  - Gelöschte Einträge werden beim Suchen berücksichtigt

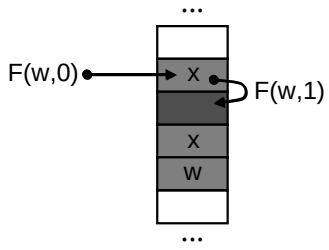
Löschen



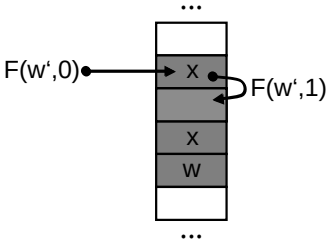
Löschen



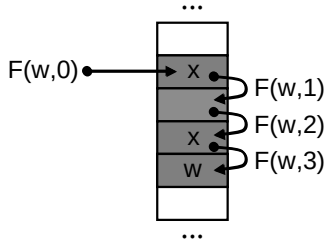
Löschen



Löschen



Löschen



Aufwandsabschätzung

- Wie viele Kollisionen treten auf?
- Wie oft muß sondiert werden?
- Annahme: ideale Hash-Funktion (aber nicht perfekt)
- Schlüssel werden gleichverteilt

Aufwandsabschätzung

- Q(i,n,m) ... Wahrscheinlichkeit, dass
  - *mindestens* i Sondierungsschritte
  - bei bereits n Elementen
  - in einer Tabelle der Größe m

  notwendig sind.

Aufwandsabschätzung

- Q(0,n,m) = 1
- Q(1,n,m) = n / m
- Q(2,n,m) = n / m * (n-1) / (m-1)
- Q(3,n,m) = ...
- Q(i,n,m) = n / m * ... * (n+1-i) / (m+1-i)

Aufwandsabschätzung

- Mittlere Kosten für das Einfügen des (n+1)sten Elementes:

$$C_{ins}(n,m) = \sum_{j=0}^{n} Q(j,n,m) = \frac{m+1}{m+1-n}$$

- Sei α = n / m, dann gilt

$$C_{ins}(n,m) \approx \frac{1}{1-\alpha}$$

Aufwandsabschätzung

$$C_{ins}(n,m) \approx \frac{1}{1-\alpha}$$

α =	50%	70%	90%	95%	99%	99.9%
C_{ins} =	2	3.3	10	20	100	1000

Aufwandsabschätzung

- Suchen eines Elementes in der Hash-Tabelle
  - Es muß dieselbe Sondierungssequenz zur Kollisionsauflösung abgearbeitet werden
  - *Negative* Suche endet beim ersten freien Tabelleneintrag $\Rightarrow C_{search}(n,m) = C_{ins}(n,m)$
  - *Positive* Suche bricht früher ab ...

Aufwandsabschätzung

- Positive Suche
  - Kosten entsprechen den Kosten bei der Einfügung (gleiche Sondierungssequenz)
  - Bei welchem Füllungsgrad wurde das Element eingefügt?
  - Bilde Mittelwert über alle möglichen Einfüge-Reihenfolgen

Aufwandsabschätzung

$$C_{search}(n,m) = \frac{1}{n}\sum_{j=0}^{n-1} C_{ins}(j,m) = \ldots = \frac{-\ln(1-\alpha)}{\alpha}$$

α =	50%	70%	90%	95%	99%	99.9%
C_{search} =	1.38	1.72	2.56	3.15	4.65	6.91

(2.5) Bäume

- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

Bäume

- Nochmal: *Suchen in Mengen ...*
- Sortieren und Suchen
  - Nur sinnvoll für ***statische*** Mengen
  - Insert(), Delete() sind O(n)
  - Search() ist O(log(n))
  - Speichermanagement

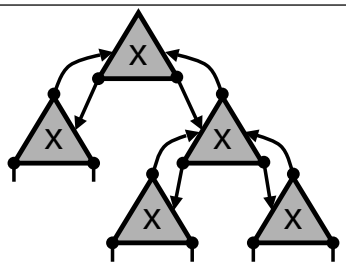
Bäume

- Nochmal: *Suchen in Mengen ...*
- Hashing
  - Anzahl der Objekte muß vorher bekannt sein ⇒ ***Tabellengröße***
  - Insert(), Search() sind O(1)
  - Delete() nicht praktikabel bei geschlossenem Hashing

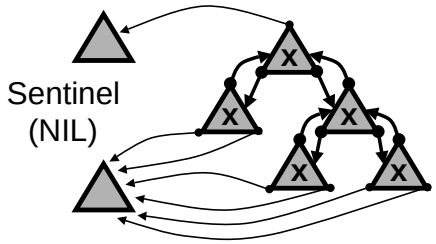
Bäume

- Nochmal: *Suchen in Mengen ...*
- Binär-Bäume
  - Beliebig dynamisch erweiterbar
  - Insert(), Delete(), Search() sind O(log(n))
- Knoten ... enthalten mindestens
  - Zeiger P zum Vorgänger
  - Zeiger L und R zum linken und rechten Nachfolger
  - Suchschlüssel X

Binär-Bäume



Binär-Bäume

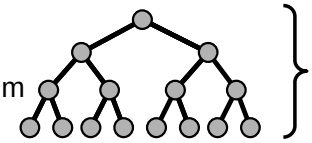


Bäume

- Effiziente und flexible Suchstruktur
  - Speichere Daten in jedem Knoten
  - Speichere Daten nur in den Blättern (z.B. Suchstruktur im Hauptspeicher, Datenblöcke auf der Festplatte)
  - Begriffe: *Ordnung, Höhe, Pfad, ...*

Binär-Bäume

Grad = 2 Höhe = 3

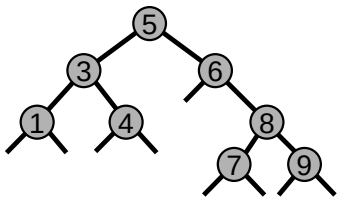


Pfad

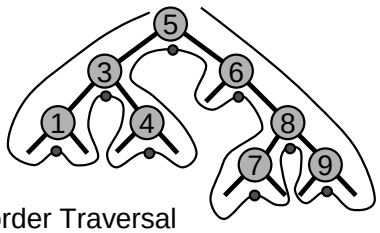
Bäume

- Suchbäume
  - Sei X_i ein Knoten und $X_1 \dots X_{i-1}$ die Knoten im *linken* und $X_{i+1} \dots X_n$ die Knoten im *rechten* Teilbaum, dann gilt
  - $\max\{X_1 \dots X_{i-1}\} < X_i < \min\{X_{i+1} \dots X_n\}$
- Insert(), Delete(), Search(), ...
  - Aufwand proportional zur Pfadlänge

Bäume

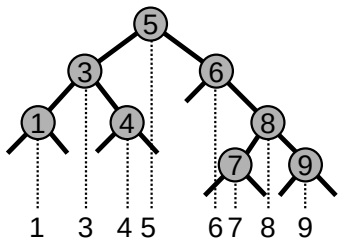


Bäume



Inorder Traversal

Bäume



Operationen

- Search(X,R)
  Finde Element mit Schlüssel X in Teilbaum R
- Min(N), Max(N)
  Finde minimales/maximales Element in Teilbaum N
- Successor(N), Predecessor(N)
  Finde Vorgänger / Nachfolger zum Knoten N
- Insert(N,R) ...
- Delete(N,R) ...

Operationen

- Search(X,R)

```
if (R = NIL ∨ X = R.X)
    return R;
else
    if (X < R.X)
        Search(X,R.L);
    else
        Search(X,R.R);
```


Operationen

- Min(N)

```
while (N.L ≠ NIL)
    N = N.L;
return N;
```

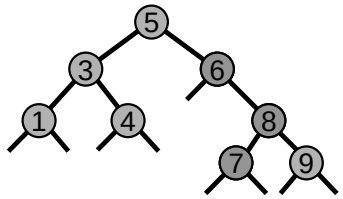
- Max(N)

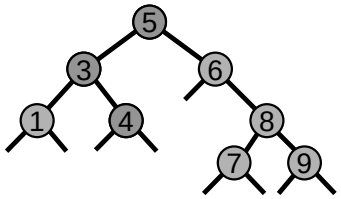
```
while (N.R ≠ NIL)
    N = N.R;
return N;
```

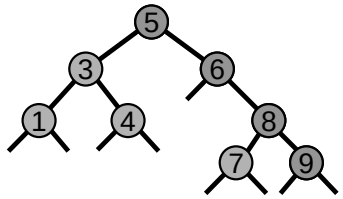

Operationen

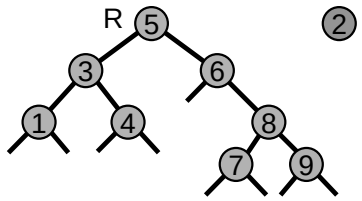
- Successor(N)

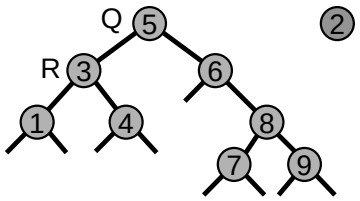
```
if (N.R ≠ NIL)
    return Min(N.R);
else
    Q = N.P;
    while (Q ≠ NIL ∧ N = Q.R)
        N = Q;
        Q = Q.P;
    return Q;
```











Operationen

Operationen

Operationen

Operationen

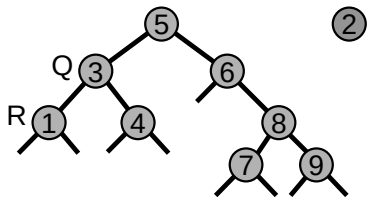
- Insert(N,R)

```
Q = NIL;
while (R ≠ NIL)
    Q = R;
    if (N.X < R.X)  R = R.L;  else  R = R.R;
N.P = Q;  N.L = NIL;  N.R = NIL
if (Q = NIL)
    root = N;
else
    if (N.X < Q.X)  Q.L = N;  else  Q.R = N;
```

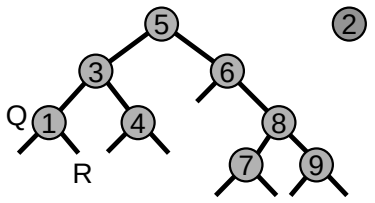
Operationen

Operationen

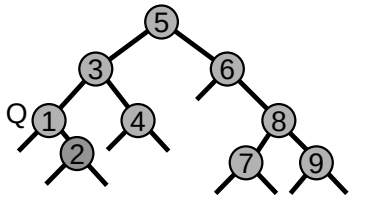
Operationen



Operationen



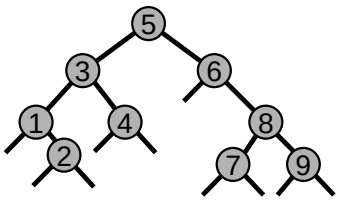
Operationen



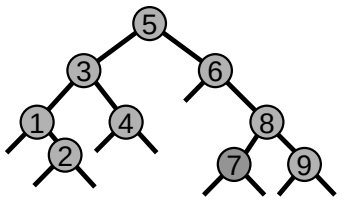
Operationen

- Delete(N,R)
- Fallunterscheidung:
  1) N hat keine Kinder (Blatt)
  2) N hat ein Kind
  3) N hat zwei Kinder

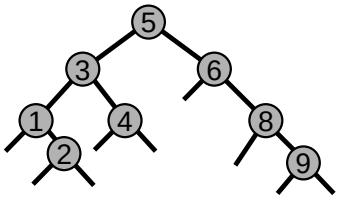
Operationen



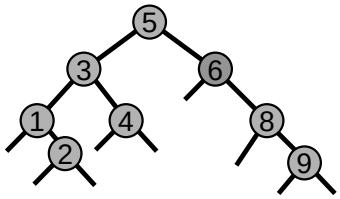
Operationen



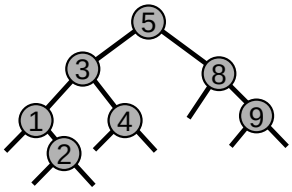
Operationen



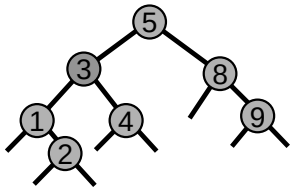
Operationen



Operationen

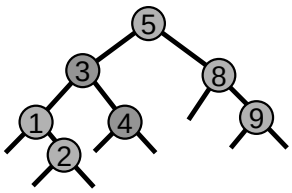


Operationen



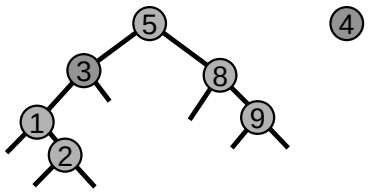
Depth(Successor(3)) > Depth(3)

Operationen

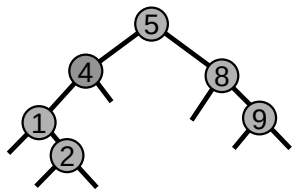


Successor(3) hat maximal ein Kind

Operationen



Operationen



Kopiere Daten von (4) nach (3)

Operationen

- Delete(N,R)

```
if (N.L = NIL ∨ N.R = NIL)  Q = N;
                            else  Q = Successor(N);
if (Q.L ≠ NIL)  S = Q.L;  else  S = Q.R;
if (S ≠ NIL)  S.P = Q.P;
if (Q.P = NIL)  root = S;
else
    if (Q = Q.P.L)  Q.P.L = S;  else  Q.P.R = S;
if (Q ≠ N)
    N.K = Q.K;  ... copy data ...
return Q;
```


Operationen

- Search(X,R)
- Min(N), Max(N)
- Successor(N), Predecessor(N)
- Insert(N,R)
- Delete(N,R)

- Alle Operationen bearbeiten genau einen Pfad im Baum
- Komplexität = O(erwartete Pfadlänge)

Aufwandsabschätzung

- Erwartete Höhe eines Suchbaums mit n Knoten N_1, ..., N_n
  - Alle Schlüssel verschieden
  - $N_i.X < N_{i+1}.X$
- Hängt von der Einfügereihenfolge ab
- Mittelwert über alle *Permutationen*

Aufwandsabschätzung

- Höhe eines binären Suchbaums mit n Knoten: H_n
- Exponentielle Höhe: $X_n = 2^{H_n}$
- Sei N_i der Wurzelknoten, dann bilden $N_1 ... N_{i-1}$ und $N_{i+1} ... N_n$ die Teilbäume
- $\Rightarrow H_n = 1 + \max \{ H_{i-1}, H_{n-i} \}$
- $\Rightarrow X_n = 2 * \max \{ X_{i-1}, X_{n-i} \}$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$E(X_n) = \frac{2}{n}\sum_{i=1}^{n} E(\max\{X_{i-1}, X_{n-i}\})$$

$$\leq \frac{2}{n}\sum_{i=1}^{n} E(X_{i-1}) + E(X_{n-i}) = \frac{4}{n}\sum_{i=0}^{n-1} E(X_i)$$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$E(X_n) = \frac{4}{n}\sum_{i=0}^{n-1} E(X_i)$$

$$E(X_n) = \ldots = \frac{1}{4}\binom{n+3}{3} = O(n^3)$$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$2^{E(H_n)} \le E(2^{H_n}) = E(X_n) \le c\,n^3$$

$$E(H_n) = \log(c\,n^3) = 3\log(n) + \log(c) = O(\log(n))$$

Aufwandsabschätzung

- $E(H_n) = O(\log(n))$ bedeutet, dass die Pfadlängen in einem Suchbaum mit n Knoten im erwarteten Fall O(log(n)) sind.
- Alle Operationen: Insert(), Delete(), ... haben einen ***erwarteten*** Aufwand von O(log(n))
- ABER: worst-case Aufwand O(n) !!!

(2.5) Bäume

- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

Bäume

- Nochmal: *Suchen in Mengen ...*
- Sortieren und Suchen
  - Nur sinnvoll für ***statische*** Mengen
  - Insert(), Delete() sind O(n)
  - Search() ist O(log(n))
  - Speichermanagement

Bäume

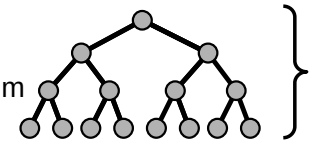
- Nochmal: *Suchen in Mengen ...*
- Hashing
  - Anzahl der Objekte muß vorher bekannt sein $\Rightarrow$ ***Tabellengröße***
  - Insert(), Search() sind O(1)
  - Delete() nicht praktikabel bei geschlossenem Hashing

Bäume

- Nochmal: *Suchen in Mengen ...*
- Binär-Bäume
  - Beliebig dynamisch erweiterbar
  - Insert(), Delete(), Search() sind O(log(n))
- Knoten ... enthalten mindestens
  - Zeiger P zum Vorgänger
  - Zeiger L und R zum linken und rechten Nachfolger
  - Suchschlüssel X

Binär-Bäume

Grad = 2 Höhe = 3



Pfad

Bäume

- Suchbäume
  - Sei X_i ein Knoten und $X_1 \ldots X_{i-1}$ die Knoten im *linken* und $X_{i+1} \ldots X_n$ die Knoten im *rechten* Teilbaum, dann gilt
  - $\max\{X_1 \ldots X_{i-1}\} < X_i < \min\{X_{i+1} \ldots X_n\}$
- Insert(), Delete(), Search(), ...
  - Aufwand proportional zur Pfadlänge

Operationen

- Search(X,R)
  Finde Element mit Schlüssel X in Teilbaum R
- Min(N), Max(N)
  Finde minimales/maximales Element in Teilbaum N
- Successor(N), Predecessor(N)
  Finde Vorgänger / Nachfolger zum Knoten N
- Insert(N,R) ...
- Delete(N,R) ...

Operationen

- Search(X,R)

```
if (R = NIL ∨ X = R.X)
    return R;
else
    if (X < R.X)
        Search(X,R.L);
    else
        Search(X,R.R);
```


Operationen

- Min(N)

```
while (N.L ≠ NIL)
    N = N.L;
return N;
```

- Max(N)

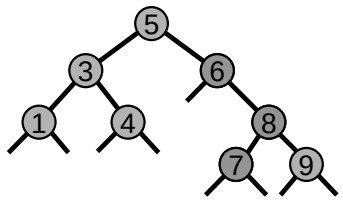
```
while (N.R ≠ NIL)
    N = N.R;
return N;
```


Operationen

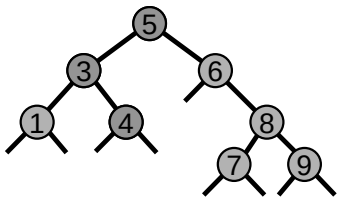
- Successor(N)

```
if (N.R ≠ NIL)
    return Min(N.R);
else
    Q = N.P;
    while (Q ≠ NIL ∧ N = Q.R)
        N = Q;
        Q = Q.P;
    return Q;
```

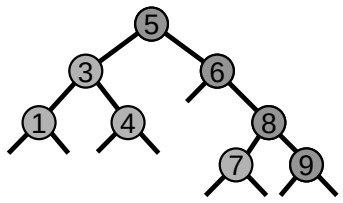

Operationen



Operationen



Operationen

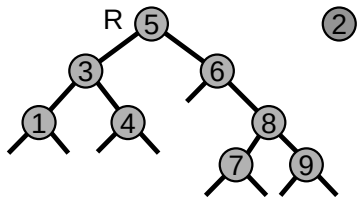


Operationen

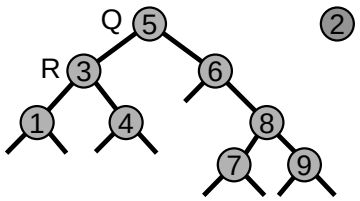
- Insert(N,R)

```
Q = NIL;
while (R ≠ NIL)
    Q = R;
    if (N.X < R.X)  R = R.L;  else  R = R.R;
N.P = Q;  N.L = NIL;  N.R = NIL
if (Q = NIL)
    root = N;
else
    if (N.X < Q.X)  Q.L = N;  else  Q.R = N;
```

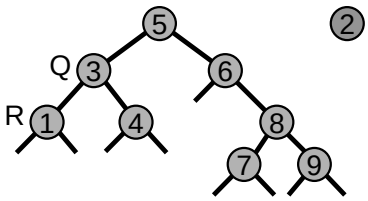

Operationen



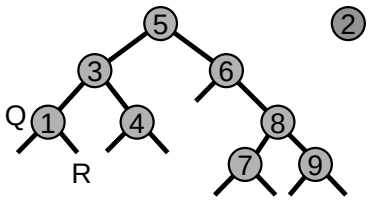
Operationen



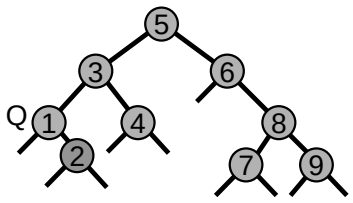
Operationen



Operationen



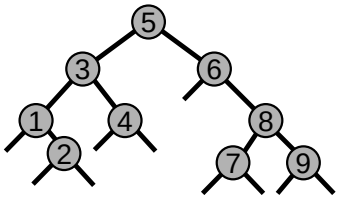
Operationen



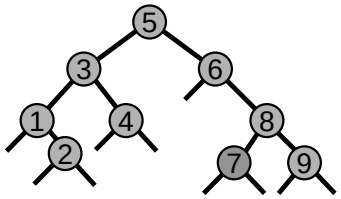
Operationen

- Delete(N,R)
- Fallunterscheindung:
  1) N hat keine Kinder (Blatt)
  2) N hat ein Kind
  3) N hat zwei Kinder

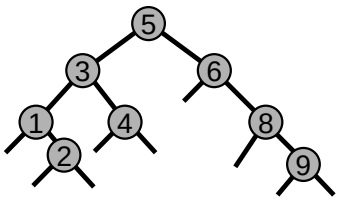
Operationen



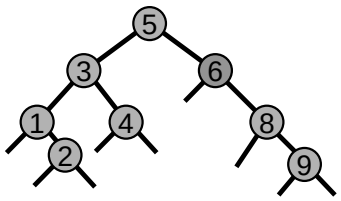
Operationen



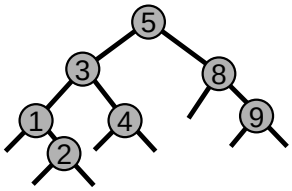
Operationen



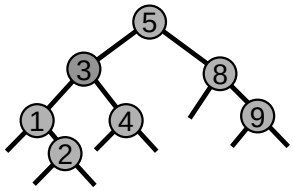
Operationen



Operationen

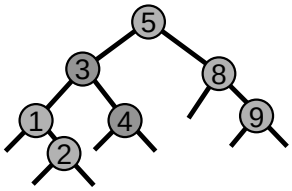


Operationen



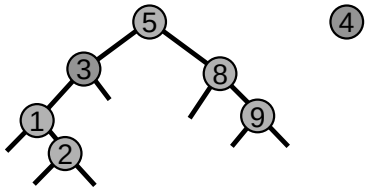
Depth(Successor(3)) > Depth(3)

Operationen

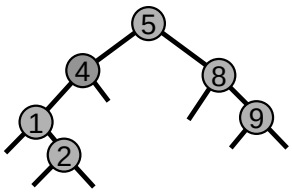


Successor(3) hat maximal ein Kind

Operationen



Operationen



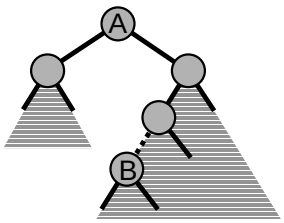
Kopiere Daten von (4) nach (3)

Operationen

- Delete(N,R)

```
if (N.L = NIL ∨ N.R = NIL)  Q = N;
                            else  Q = Successor(N);
if (Q.L ≠ NIL)  S = Q.L;  else  S = Q.R;
if (S ≠ NIL)  S.P = Q.P;
if (Q.P = NIL)  root = S;
 else
     if (Q = Q.P.L)  Q.P.L = S;  else  Q.P.R = S;
if (Q ≠ N)
    N.K = Q.K;  ... copy data ...
return Q;
```

Successor()



Operationen

- Search(X,R)
- Min(N), Max(N)
- Successor(N), Predecessor(N)
- Insert(N,R)
- Delete(N,R)

- Alle Operationen bearbeiten genau einen Pfad im Baum
- Komplexität = O(erwartete Pfadlänge)

Aufwandsabschätzung

- Erwartete Höhe eines Suchbaums mit n Knoten N_1, ..., N_n
  - Alle Schlüssel verschieden
  - $N_i.X < N_{i+1}.X$
- Hängt von der Einfügereihenfolge ab
- Mittelwert über alle *Permutationen*

Aufwandsabschätzung

- Höhe eines binären Suchbaums mit n Knoten: H_n
- Exponentielle Höhe: $X_n = 2^{H_n}$
- Sei N_i der Wurzelknoten, dann bilden $N_1...N_{i-1}$ und $N_{i+1}...N_n$ die Teilbäume
- $\Rightarrow H_n = 1 + \max \{ H_{i-1}, H_{n-i} \}$
- $\Rightarrow X_n = 2 * \max \{ X_{i-1}, X_{n-i} \}$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$E(X_n) = \frac{2}{n}\sum_{i=1}^{n} E(\max\{X_{i-1}, X_{n-i}\})$$

$$\leq \frac{2}{n}\sum_{i=1}^{n} E(X_{i-1}) + E(X_{n-i}) = \frac{4}{n}\sum_{i=0}^{n-1} E(X_i)$$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$E(X_n) = \frac{4}{n}\sum_{i=0}^{n-1} E(X_i)$$

$$E(X_n) = \ldots = \frac{1}{4}\binom{n+3}{3} = O(n^3)$$

Aufwandsabschätzung

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$2^{E(H_n)} \le E(2^{H_n}) = E(X_n) \le c\,n^3$$

$$E(H_n) = \log(c\,n^3) = 3\log(n) + \log(c) = O(\log(n))$$

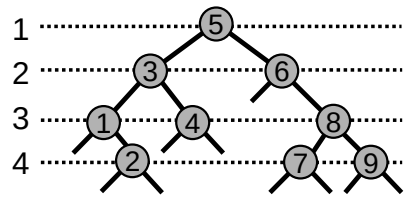
Aufwandsabschätzung

- $E(H_n) = O(\log(n))$ bedeutet, dass die Pfadlängen in einem Suchbaum mit n Knoten im erwarteten Fall O(log(n)) sind.
- Alle Operationen: Insert(), Delete(), ... haben einen ***erwarteten*** Aufwand von O(log(n))
- ABER: worst-case Aufwand O(n) !!!

(2.5.2) Optimale Suchbäume

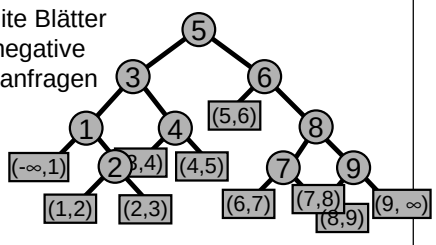
- In manchen Anwendungen ist die Zugriffswahrscheinlichkeit auf die Elemente a priori bekannt.
- Versuche die Elemente, auf die am häufigsten zugegriffen wird, in der Nähe der Wurzel zu speichern
- Beispiel: Worthäufigkeiten, Übersetzungstabelle

Optimale Suchbäume



Optimale Suchbäume

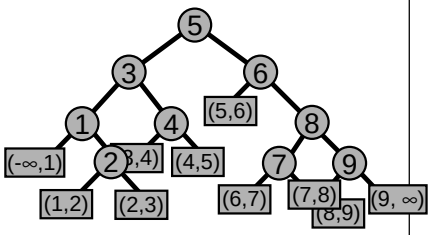
Explizite Blätter für negative Suchanfragen



Zugriffswahrscheinlichkeiten

- Seien N_1 ... N_n die (echten) Knoten des Suchbaums
- Dann gibt es genau n+1 Intervalle
- p_i sei die Wahrscheinlichkeit, dass auf Knoten N_i zugriffen wird.
- q_i sei die Wahrscheinlichkeit, dass nach einem Schlüssel X mit $N_i.X < X < N_{i+1}.X$ gesucht wird.

Optimale Suchbäume



Aufwandsabschätzung

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

$$C_{average} = \sum_{i=1}^{n} (depth(N_i)+1)\, p_i + \sum_{i=0}^{n} (depth(I_i)+1)\, q_i$$

$$= 1 + \sum_{i=1}^{n} depth(N_i)\, p_i + \sum_{i=0}^{n} depth(I_i)\, q_i$$

Aufwandsabschätzung

- Betrachte einen Teilbaum mit Wurzel N_k des optimalen Suchbaums
- Dieser Baum enthält eine Teilmenge N_i, ..., N_j der Knoten und I_{i-1}, ..., I_j der Blatt-Intervalle
- Dieser Teilbaum ist ebenfalls ein optimaler Suchbaum (*cut & paste*)

Aufwandsabschätzung

- Die Wahrscheinlichkeit, dass auf den Teilbaum mit Wurzel N_k zugegriffen wird, ist
$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$
- Seien C(i,j) die Kosten, die durch den Teilbaum N_i ... N_j verursacht werden.

Rekursive Optimierung

- Der Teilbaum mit Wurzel N_k besitzt zwei weitere Teilbäume mit den Knoten
  - N_i ... N_{k-1} und I_{i-1}... I_{k-1} bzw.
  - N_{k+1} ... N_j und I_k ... I_j

  die jeder für sich wieder optimal sind.
- Jeder Pfad im Teilbaum N_i ... N_{k-1} wird durch N_k um einen Schritt verlängert

Rekursive Optimierung

$$C(i,j) = p_k + C(i,k-1) + w(i,k-1) + C(k+1,j) + w(k+1,j)$$

$$w(i,j) = w(i,k-1) + p_k + w(k+1,j)$$

$$C(i,j) = C(i,k-1) + C(k+1,j) + w(i,j)$$

$$C(i,i-1) = q_{i-1}$$

Rekursive Optimierung

- Für jeden Teilbaum N_i ... N_j soll der optimale Wurzelknoten bestimmt werden ...

$$C(i, i-1) = q_{i-1}$$

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + w(i, j)$$

Rekursive Optimierung

- Für jeden Teilbaum N_i ... N_j soll der optimale Wurzelknoten bestimmt werden ...

$$C(i, i-1) = q_{i-1}$$

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + w(i, j)$$

- $\Rightarrow$ ***dynamische Programmierung***

Dynamische Programmierung

- Speichere Zwischenergebnisse in Tabellen ...
  - C[i,j] : Kosten des optimalen Suchbaums für die Knoten N_i ... N_j und I_{i-1} ... I_j
  - W[i,j] : Zugriffswahrscheinlichkeit für den entsprechenden Teilbaum
  - R[i,j] : Wurzelknoten des optimalen Suchbaums

Dynamische Programmierung

- OptTree(P[1..n], Q[0..n], n)

```
for (i=1 ; i≤n+1 ; i++)
    C[i,i-1] = W[i,i-1] = Q[i-1];
for (l=1 ; l≤n ; l++)
    for (i=1 ; i≤n-l+1 ; i++)
        j = i+l-1;
        W[i,j] = W[i,j-1] + P[j] + Q[j];
        C[i,j] = min { C[i,k-1]+C[k+1,j] } + W[i,j];
        R[i,j] = argmin { C[i,k-1]+C[k+1,j] };
```


Optimaler Suchbaum

- Der optimale Suchbaum muß nicht immer minimale Höhe besitzen
- Das häufigst gefragte Element muß nicht der Wurzelknoten sein
- Berechnungsaufwand mit D.P.: $O(n^3)$
- Lohnt nur für viele Anfragen
- Wahrscheinlichkeiten können aus der Zugriffsstatistik extrapoliert werden

(2.5.3) Balancierte Bäume

- Nachteil bei normalen Suchbäumen: worst-case Aufwand ist O(n)
- Tritt bei „*degenerierten Bäumen*“ auf
- Kann durch ***Balancierung*** verhindert werden.
- Problem: stelle Balance in O(log(n)) wieder her.

Balancierte Bäume

- ***Gewichtsbalance:*** Für jeden Knoten N unterscheidet sich die Anzahl der Knoten im *linken* und *rechten* Teilbaum um maximal eins.
- ***Höhenbalance:*** Für jeden Knoten N unterscheidet sich die Höhe des *linken* und *rechten* Teilbaums um maximal eins.

Balancierte Bäume

- Zur Erinnerung:
  - Maximale Anzahl von Knoten in einem Baum der Höhe h ist $O(2^h)$
  - Minimale Anzahl von Knoten in einem Baum der Höhe h ist O(h)
  - Minimale Anzahl von Knoten in einem ***balancierten*** Baum der Höhe h ist $O(2^{h/2})$

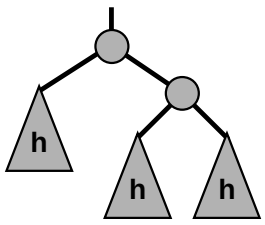
Balancierte Bäume

- Zur Erinnerung:
  - Minimale Höhe eines Baumes mit n Knoten ist O(log(n))
  - Maximale Höhe eines Baumes mit n Knoten ist O(n)
  - Maximale Höhe eines ***balancierten*** Baumes mit n Knoten ist O(2 log(n)) = O(log(n))

AVL-Bäume

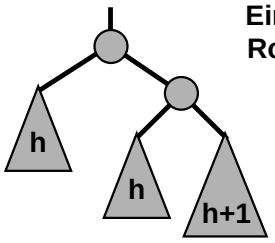
- Werden wie normale binäre Suchbäume behandelt
- Jeder Knoten speichert sein Ungleichgewicht (+1, 0, -1, X)
- Nach Insert() oder Delete() Wiederherstellung der Balance durch Rotationen

AVL-Bäume



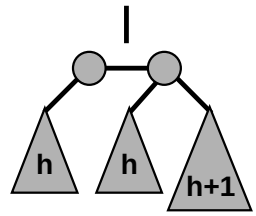
AVL-Bäume

Einfache Rotation



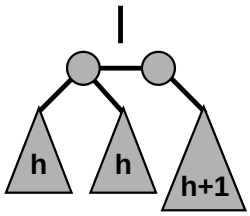
AVL-Bäume

Einfache Rotation



AVL-Bäume

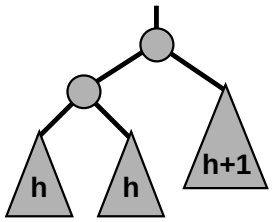
Einfache Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

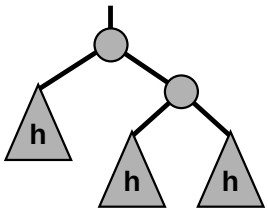
AVL-Bäume

Einfache Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

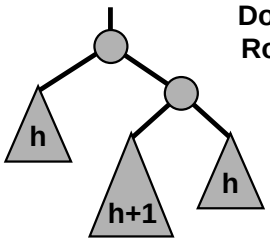
AVL-Bäume



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

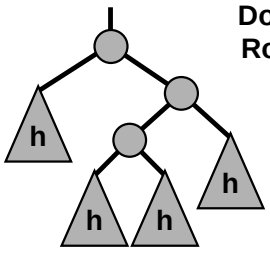
Doppelte Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

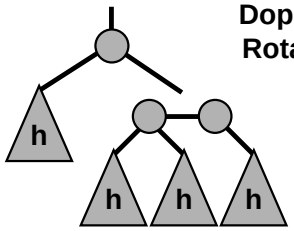
Doppelte Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

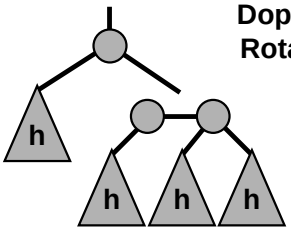
AVL-Bäume

Doppelte Rotation



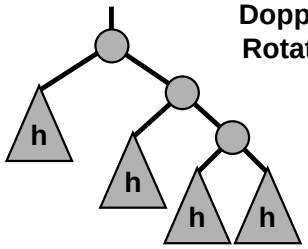
AVL-Bäume

Doppelte Rotation



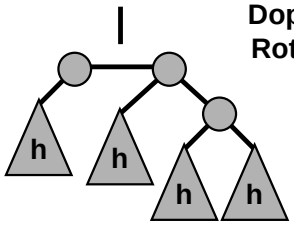
AVL-Bäume

Doppelte Rotation



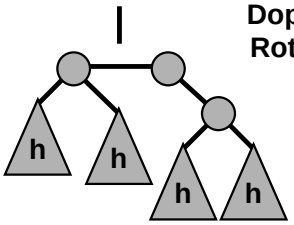
AVL-Bäume

Doppelte Rotation



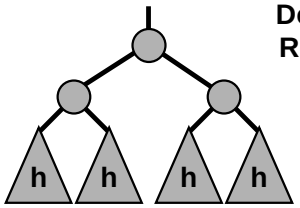
AVL-Bäume

Doppelte Rotation



AVL-Bäume

Doppelte Rotation



AVL-Bäume

- Rotationen müssen ggf. nach oben propagiert werden
- Im schlechtesten Fall O(log(n)) Rotationen bei Delete()
- Nur praktikabel, wenn gesamte Datenstruktur im Hauptspeicher

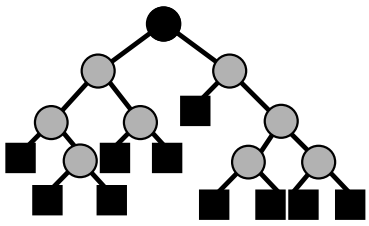
Rot/Schwarz-Bäume

- Binärer Suchbaum
- Rote und Schwarze Knoten
- Verhältnis
  längster zu kürzester Pfad = 2 : 1
- Re-Balancing durch Rotationen und Umfärben

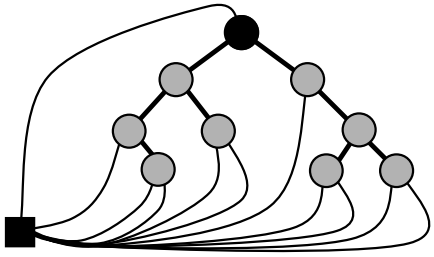
Rot/Schwarz-Bäume

- Konsistenzregeln ...
  - Jeder Knoten ist rot oder schwarz
  - Auf keinem Pfad folgen zwei rote Knoten direkt aufeinander
  - Alle Pfade von der Wurzel zu den Blättern haben dieselbe Anzahl schwarzer Knoten
  - Wurzel und Blätter sind immer Schwarz

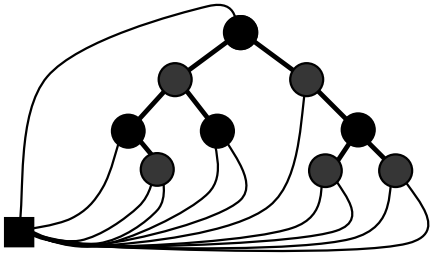
Rot/Schwarz-Bäume

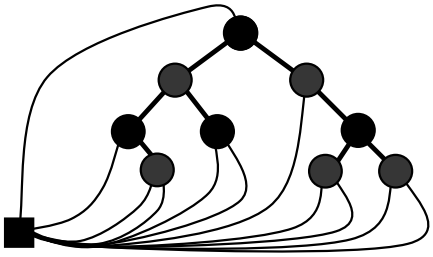


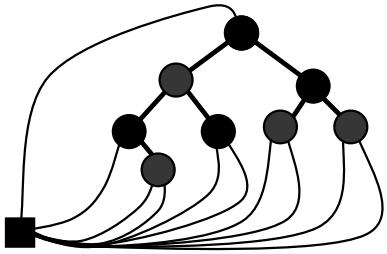
Rot/Schwarz-Bäume

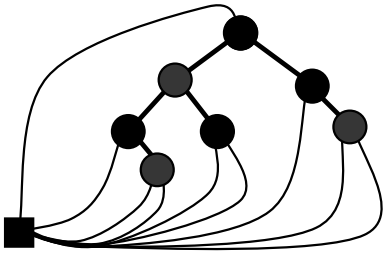


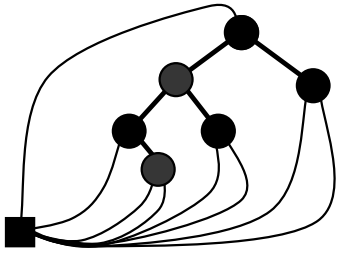
Rot/Schwarz-Bäume

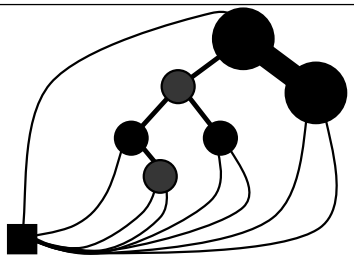












Rot/Schwarz-Bäume

- Eigenschaften
  - – Jeder Knoten hat zwei Söhne oder keinen
  - – Maximales Pfadlängenverhältnis 2 : 1
  - – Pfadlängen Θ(log(n))

Rot/Schwarz-Bäume

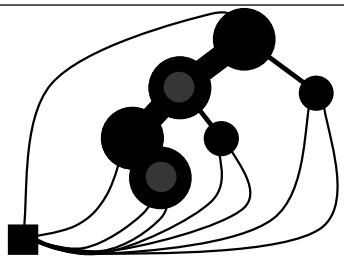
Rot/Schwarz-Bäume

Rot/Schwarz-Bäume

Rot/Schwarz-Bäume

Rot/Schwarz-Bäume

Rot/Schwarz-Bäume



Rot/Schwarz-Bäume

- Insert(), Delete()
  - Wie bei normalen binären Suchbäumen mit anschließender Widerherstellung der Konsistenzbedingungen
  - Insert() ... drei verschiedene Fälle
  - Delete() ... vier verschiedene Fälle
  - Pseudo-Code ... siehe ***„T. H. Cormen“***

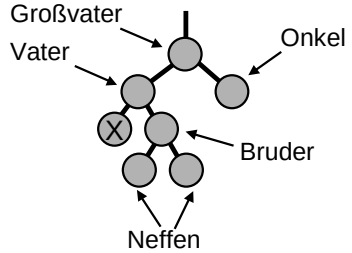
Rot/Schwarz-Bäume

- Insert()
  - Jeder eingefügte Knoten ist zunächst rot
  - Es tritt **genau eine** Konsistenzverletzung auf
  - Bei Wiederherstellung durch Umfärben oder Rotation kann genau eine weitere Verletzung auftreten
  - Propagiere Modifikationen nach oben

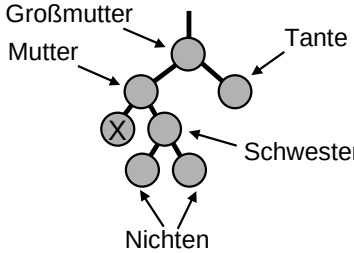
Rot/Schwarz-Bäume

- Insert()
- Mögliche Konsistenzverletzungen
  - Neuer Knoten ist Wurzel
  - Neuer Knoten ist Nachfolger eines roten Knotens
  - Fallunterscheidung nach der Farbe des „Onkels“

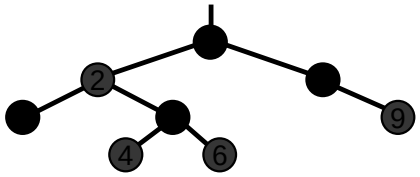
Männliche Verwandtschaft



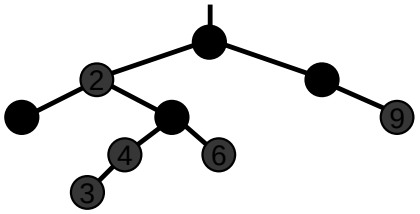
Weibliche Verwandtschaft



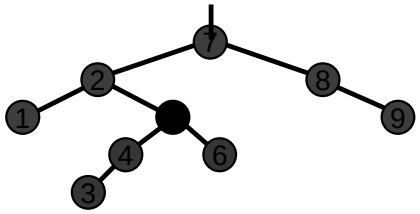
Rot/Schwarz-Bäume



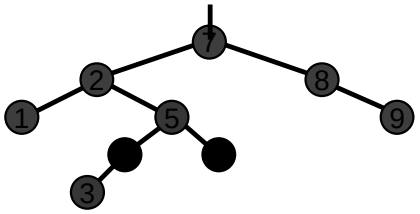
Rot/Schwarz-Bäume



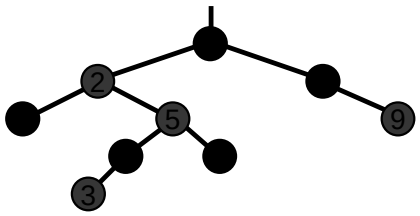
Rot/Schwarz-Bäume



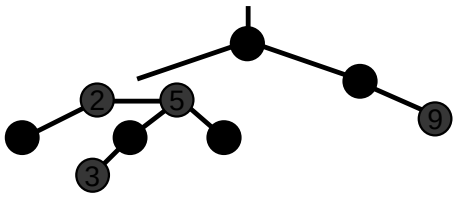
Rot/Schwarz-Bäume



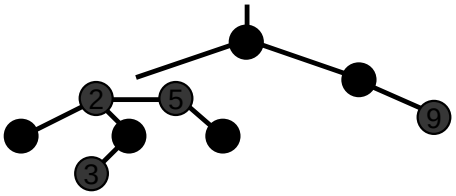
Rot/Schwarz-Bäume



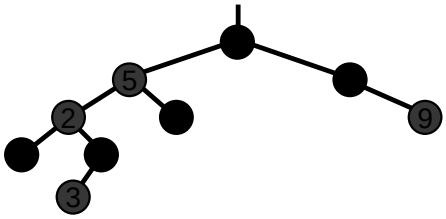
Rot/Schwarz-Bäume



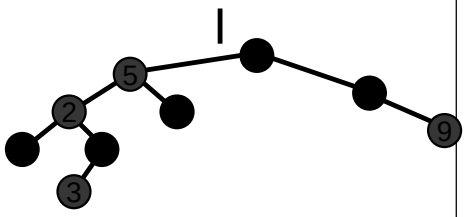
Rot/Schwarz-Bäume



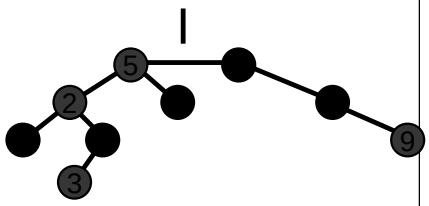
Rot/Schwarz-Bäume



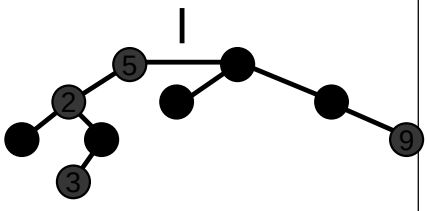
Rot/Schwarz-Bäume



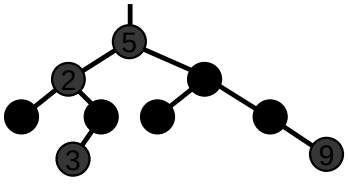
Rot/Schwarz-Bäume



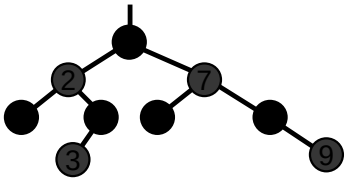
Rot/Schwarz-Bäume



Rot/Schwarz-Bäume



Rot/Schwarz-Bäume



Fallunterscheidung Insert()

1) Onkel ist rot
   - *Umfärben*
2) Onkel ist schwarz
   a) Sohn < Vater < Großvater oder
      Sohn > Vater > Großvater
      - *Einfache Rotation, Umfärben*
   b) Sohn < Vater > Großvater oder
      Sohn > Vater < Großvater
      - *Doppelte Rotation (Rückführung auf Fall 2a)*

(2.5) Bäume

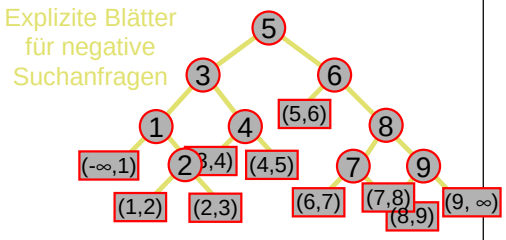
- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

(2.5.2) Optimale Suchbäume

- In manchen Anwendungen ist die Zugriffswahrscheinlichkeit auf die Elemente a priori bekannt.
- Versuche die Elemente, auf die am häufigsten zugegriffen wird, in der Nähe der Wurzel zu speichern
- Beispiel: Worthäufigkeiten, Übersetzungstabelle

Optimale Suchbäume

Explizite Blätter für negative Suchanfragen



Aufwandsabschätzung

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

$$C_{average} = \sum_{i=1}^{n} (depth(N_i)+1)\, p_i + \sum_{i=0}^{n} (depth(I_i)+1)\, q_i$$

$$= 1 + \sum_{i=1}^{n} depth(N_i)\, p_i + \sum_{i=0}^{n} depth(I_i)\, q_i$$

Aufwandsabschätzung

- Betrachte einen Teilbaum mit Wurzel N_k des optimalen Suchbaums
- Dieser Baum enthält eine Teilmenge $N_i, \ldots, N_j$ der Knoten und $I_{i-1}, \ldots, I_j$ der Blatt-Intervalle
- Dieser Teilbaum ist ebenfalls ein optimaler Suchbaum (*cut & paste*)

Aufwandsabschätzung

- Die Wahrscheinlichkeit, dass auf den Teilbaum mit Wurzel N_k zugegriffen wird, ist

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

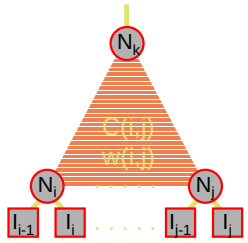
- Seien $C(i,j)$ die Kosten, die durch den Teilbaum $N_i \ldots N_j$ verursacht werden.

Rekursive Optimierung

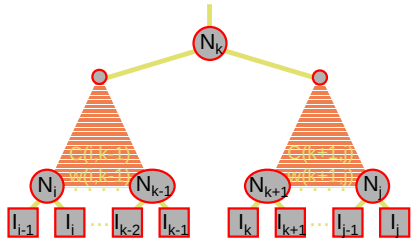
- Der Teilbaum mit Wurzel N_k besitzt zwei weitere Teilbäume mit den Knoten
  - $N_i \dots N_{k-1}$ und $l_{i-1} \dots l_{k-1}$ bzw.
  - $N_{k+1} \dots N_j$ und $l_k \dots l_j$

  die jeder für sich wieder optimal sind.
- Jeder Pfad im Teilbaum $N_i \dots N_{k-1}$ wird durch N_k um einen Schritt verlängert

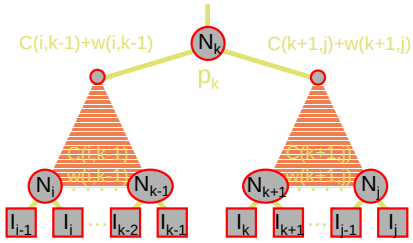
Rekursive Optimierung



Rekursive Optimierung



Rekursive Optimierung



Rekursive Optimierung

$$C(i,j) = p_k + C(i,k-1) + w(i,k-1) + C(k+1,j) + w(k+1,j)$$

$$w(i,j) = w(i,k-1) + p_k + w(k+1,j)$$

$$C(i,j) = C(i,k-1) + C(k+1,j) + w(i,j)$$

$$C(i,i-1) = q_{i-1}$$

Rekursive Optimierung

- Für jeden Teilbaum $N_i \dots N_j$ soll der optimale Wurzelknoten bestimmt werden ...

$$C(i,i-1) = q_{i-1}$$

$$C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j)\} + w(i,j)$$

Rekursive Optimierung

- Für jeden Teilbaum N_i ... N_j soll der optimale Wurzelknoten bestimmt werden ...

$$C(i,i-1) = q_{i-1}$$

$$C(i,j) = \min_{i \le k \le j} \{C(i,k-1) + C(k+1,j)\} + w(i,j)$$

- ⇒ ***dynamische Programmierung***

Dynamische Programmierung

- Speichere Zwischenergebnisse in Tabellen ...
  - C[i,j] : Kosten des optimalen Suchbaums für die Knoten N_i ... N_j und l_{i-1} ... l_j
  - W[i,j] : Zugriffswahrscheinlichkeit für den entsprechenden Teilbaum
  - R[i,j] : Wurzelknoten des optimalen Suchbaums

Dynamische Programmierung

- OptTree(P[1..n], Q[0..n], n)

```
for (i=1 ; i≤n+1 ; i++)
    C[i,i-1] = W[i,i-1] = Q[i-1];
for (l=1 ; l≤n ; l++)
    for (i=1 ; i≤n-l+1 ; i++)
        j = i+l-1;
        W[i,j] = W[i,j-1] + P[j] + Q[j];
        C[i,j] = min { C[i,k-1]+C[k+1,j] } + W[i,j];
        R[i,j] = argmin { C[i,k-1]+C[k+1,j] };
```


Optimaler Suchbaum

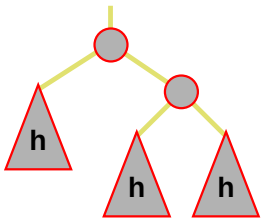
- Der optimale Suchbaum muß nicht immer minimale Höhe besitzen
- Das häufigst gefragte Element muß nicht der Wurzelknoten sein
- Berechnungsaufwand mit D.P.: $O(n^3)$
- Lohnt nur für viele Anfragen
- Wahrscheinlichkeiten können aus der Zugriffsstatistik extrapoliert werden

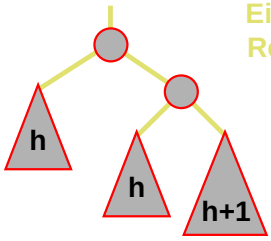
(2.5.3) Balancierte Bäume

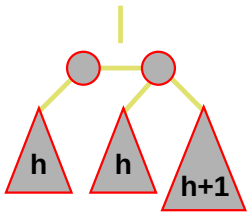
- Nachteil bei normalen Suchbäumen: worst-case Aufwand ist O(n)
- Tritt bei „*degenerierten Bäumen*“ auf
- Kann durch ***Balancierung*** verhindert werden.
- Problem: stelle Balance in O(log(n)) wieder her.

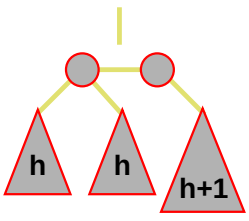
Balancierte Bäume

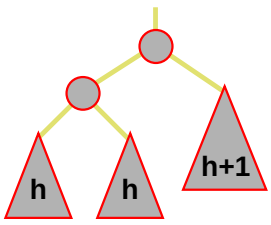
- ***Gewichtsbalance:*** Für jeden Knoten N unterscheidet sich die Anzahl der Knoten im *linken* und *rechten* Teilbaum um maximal eins.
- ***Höhenbalance:*** Für jeden Knoten N unterscheidet sich die Höhe des *linken* und *rechten* Teilbaums um maximal eins.











AVL-Bäume

- Werden wie normale binäre Suchbäume behandelt
- Jeder Knoten speichert sein Ungleichgewicht (+1, 0, -1, X)
- Nach Insert() oder Delete() Wiederherstellung der Balance durch Rotationen

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

Einfache Rotation

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

Einfache Rotation

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

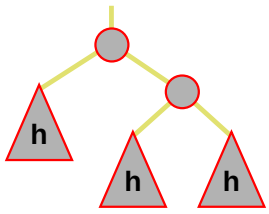
Einfache Rotation

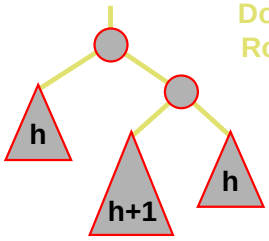
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

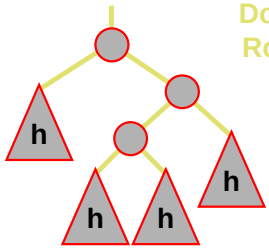
AVL-Bäume

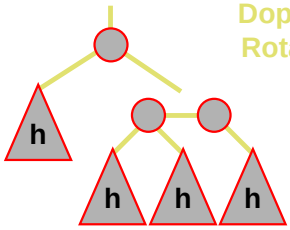
Einfache Rotation

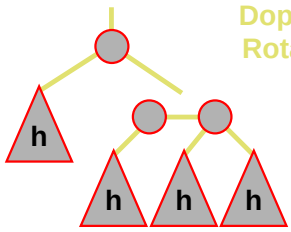
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

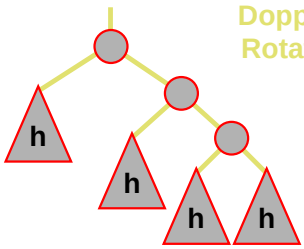












AVL-Bäume

AVL-Bäume

Doppelte Rotation

AVL-Bäume

Doppelte Rotation

AVL-Bäume

Doppelte Rotation

AVL-Bäume

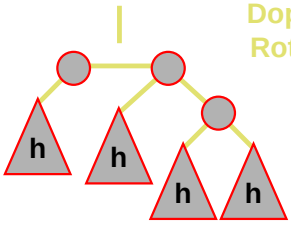
Doppelte Rotation

AVL-Bäume

Doppelte Rotation

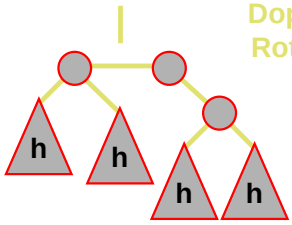
AVL-Bäume

Doppelte Rotation



AVL-Bäume

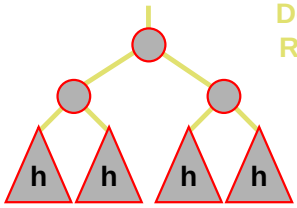
Doppelte Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

Doppelte Rotation



Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

AVL-Bäume

- Rotationen müssen ggf. nach oben propagiert werden
- Im schlechtesten Fall O(log(n)) Rotationen bei Delete()
- Nur praktikabel, wenn gesamte Datenstruktur im Hauptspeicher

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rot/Schwarz-Bäume

- Binärer Suchbaum
- Rote und Schwarze Knoten
- Verhältnis
  längster zu kürzester Pfad = 2 : 1
- Re-Balancing durch Rotationen und Umfärben

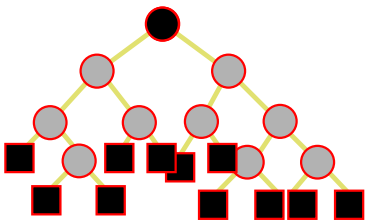
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Rot/Schwarz-Bäume

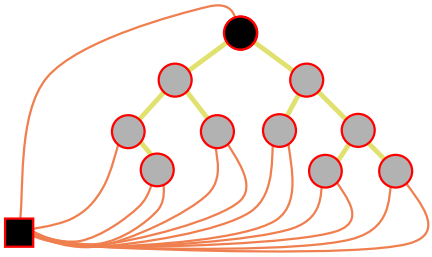
- Konsistenzregeln ...
  - Jeder Knoten ist rot oder schwarz
  - Auf keinem Pfad folgen zwei rote Knoten direkt aufeinander
  - Alle Pfade von der Wurzel zu den Blättern haben dieselbe Anzahl schwarzer Knoten
  - Wurzel und Blätter sind immer schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

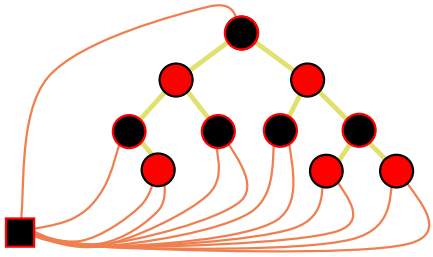
Rot/Schwarz-Bäume



Rot/Schwarz-Bäume



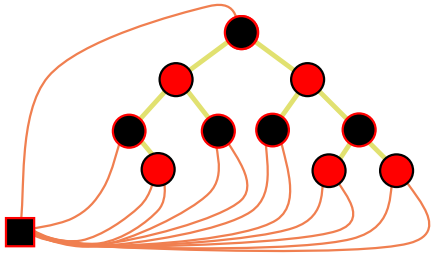
Rot/Schwarz-Bäume



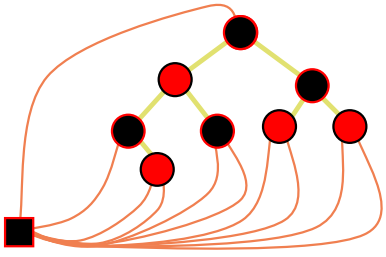
Rot/Schwarz-Bäume

- Eigenschaften
  - Jeder Knoten hat zwei Söhne oder keinen
  - Maximales Pfadlängenverhältnis 2 : 1
  - Pfadlängen $\Theta(\log(n))$

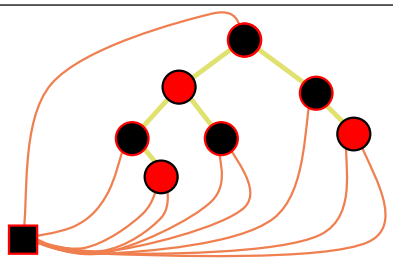
Rot/Schwarz-Bäume



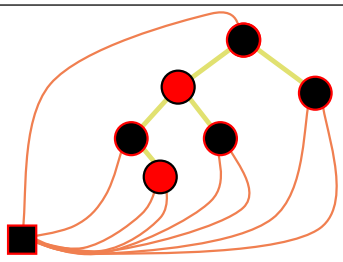
Rot/Schwarz-Bäume



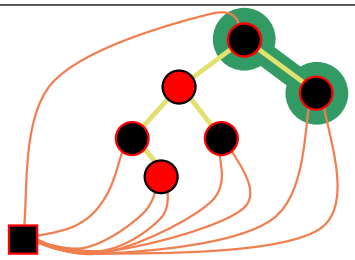
Rot/Schwarz-Bäume



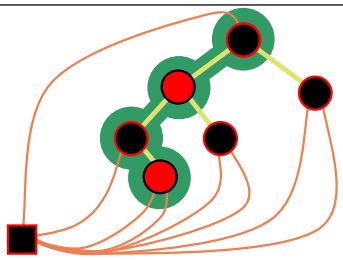
Rot/Schwarz-Bäume



Rot/Schwarz-Bäume



Rot/Schwarz-Bäume



Rot/Schwarz-Bäume

- Insert(), Delete()
  - Wie bei normalen binären Suchbäumen mit anschließender Widerherstellung der Konsistenzbedingungen
  - Insert() ... drei verschiedene Fälle
  - Delete() ... vier verschiedene Fälle
  - Pseudo-Code ... siehe ***„T. H. Cormen“***

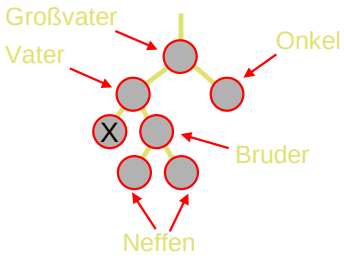
Rot/Schwarz-Bäume

- Insert()
  - Jeder eingefügte Knoten ist zunächst rot
  - Es tritt ***genau eine*** Konsistenzverletzung auf
  - Bei Wiederherstellung durch Umfärben oder Rotation kann ***genau eine*** weitere Verletzung auftreten
  - Propagiere Modifikationen nach oben

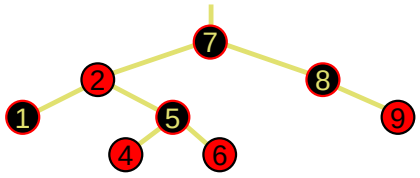
Rot/Schwarz-Bäume

- Insert()
- Mögliche Konsistenzverletzungen
  - Neuer Knoten ist Wurzel
  - Neuer Knoten ist Nachfolger eines roten Knotens
  - Fallunterscheidung nach der Farbe des „Onkels“

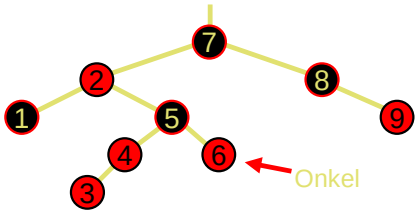
Männliche Verwandtschaft



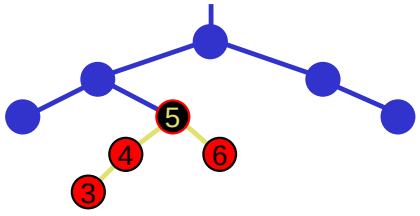
Rot/Schwarz-Bäume



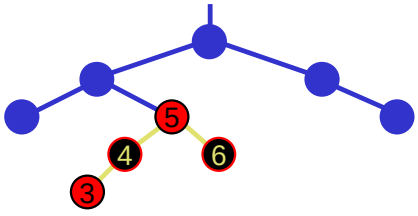
Rot/Schwarz-Bäume



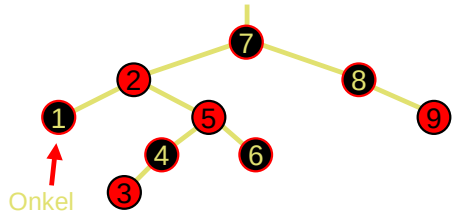
Rot/Schwarz-Bäume



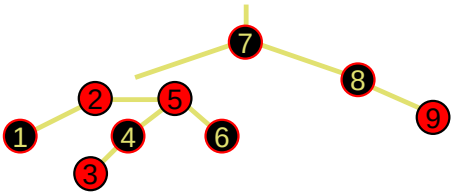
Rot/Schwarz-Bäume



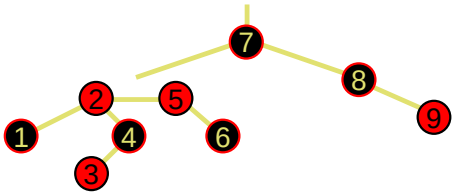
Rot/Schwarz-Bäume



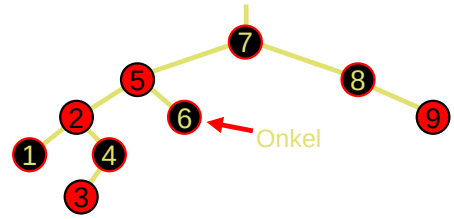
Rot/Schwarz-Bäume



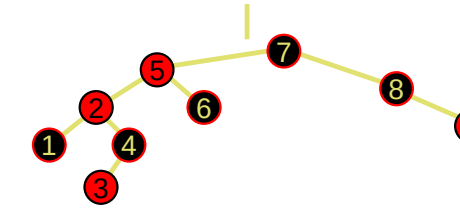
Rot/Schwarz-Bäume



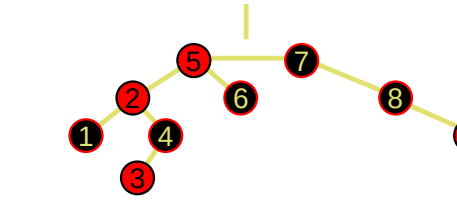
Rot/Schwarz-Bäume



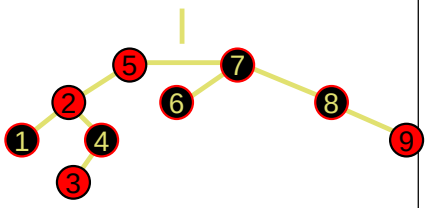
Rot/Schwarz-Bäume



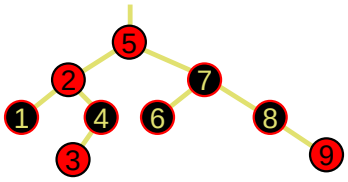
Rot/Schwarz-Bäume



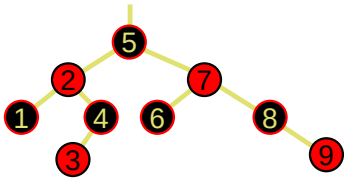
Rot/Schwarz-Bäume



Rot/Schwarz-Bäume

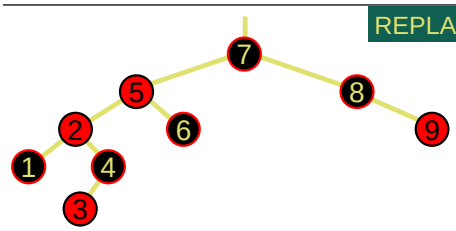


Rot/Schwarz-Bäume



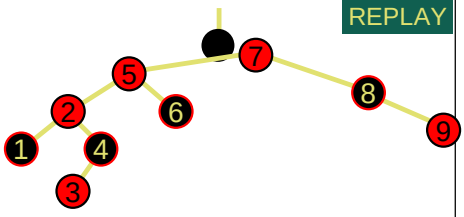
Rot/Schwarz-Bäume

REPLAY



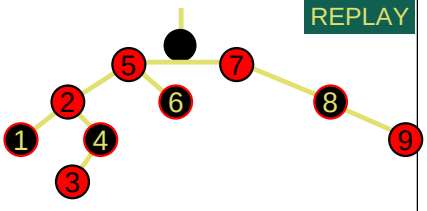
Rot/Schwarz-Bäume

REPLAY



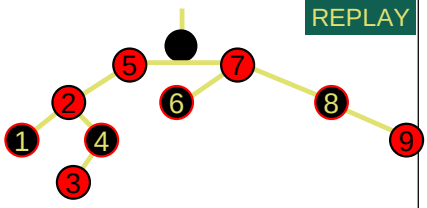
Rot/Schwarz-Bäume

REPLAY



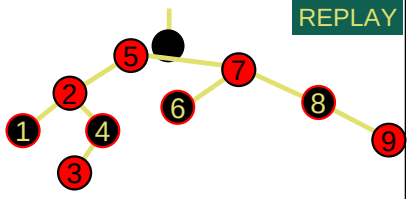
Rot/Schwarz-Bäume

REPLAY



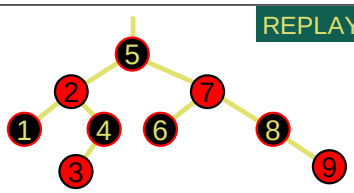
Rot/Schwarz-Bäume

REPLAY



Rot/Schwarz-Bäume

REPLAY



Fallunterscheidung Insert()

1) Onkel ist rot
   - *Umfärben*
2) Onkel ist schwarz
   a) Sohn < Vater < Großvater oder
      Sohn > Vater > Großvater
      - *Einfache Rotation, Umfärben*
   b) Sohn < Vater > Großvater oder
      Sohn > Vater < Großvater
      - *Doppelte Rotation (Rückführung auf Fall 2a)*

Aufwandsabschätzung

- Suchen der Einfügestelle: O(log(n))
- Einfügen: O(1)
- Konsistenzwiederherstellung
  – Umfärbe-Schritte: O(log(n))
    (Schritt vom Enkel zum Großvater)
  – Rotationen: O(1) !!! (maximal zwei)
- Insgesamt: O(log(n))

Rot/Schwarz-Bäume

- Delete()
  – Wie bei normalen binären Suchbäumen wird der Knoten entweder direkt gelöscht oder sein Successor wird gelöscht und überschreibt den eigentlichen Knoten
  – Konsistenzwiederherstellung
    - Wenn der Knoten rot war, ist nichts zu tun
    - Wann der Knoten schwarz war, gibt es 4 Fälle

Rot/Schwarz-Bäume

- Delete()
  - Falls der (tatsächlich) entfernte Knoten schwarz ist, fehlt auf allen Pfaden innerhalb des entsprechenden Teil-Baumes eine schwarze Marke
  - Weise diese schwarze Marke dem Nachfolger des gelöschten Knotens zu.

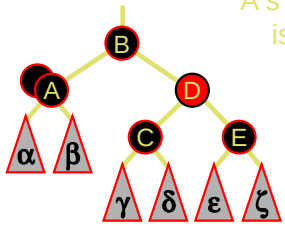
    Dadurch wird dieser schwarz-rot oder doppel-schwarz
  - Verschiebe die schwarze Marke im Baum nach oben

Rot/Schwarz-Bäume

- Delete()
- Umfärben und Rotationen
  - Erhalte die Anzahl schwarzer Marken der Sub-Bäume
- Abbruchbedingungen
  - Ein schwarz-roter Knoten wird schwarz gefärbt
  - Der Wurzelknoten wird erreicht
    (doppel-schwarz kann einfach wegfallen,
    da die Wurzel auf allen Pfaden liegt)

Fallunterscheidung Delete()

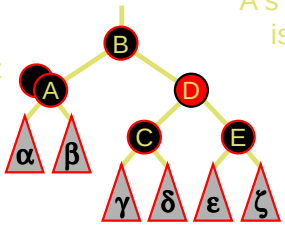
A‘s Bruder ist rot



Fallunterscheidung Delete()

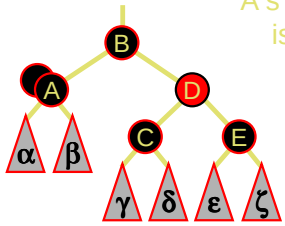
C und E müssen schwarz sein!

A‘s Bruder ist rot



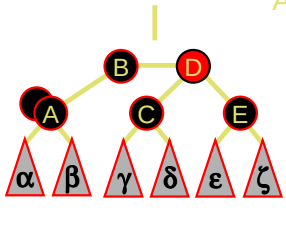
Fallunterscheidung Delete()

A‘s Bruder ist rot



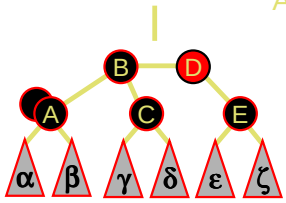
Fallunterscheidung Delete()

A‘s Bruder ist rot



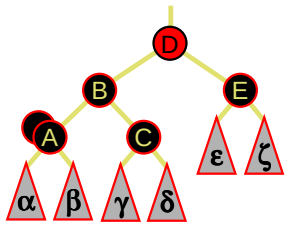
Fallunterscheidung Delete()

A‘s Bruder
ist rot



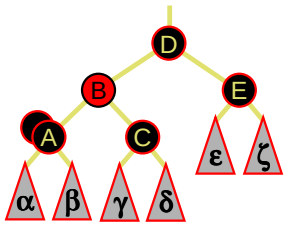
Fallunterscheidung Delete()

A‘s Bruder
ist rot



Fallunterscheidung Delete()

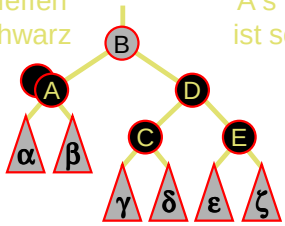
A‘s Bruder
ist rot



Fallunterscheidung Delete()

Beide Neffen
auch schwarz

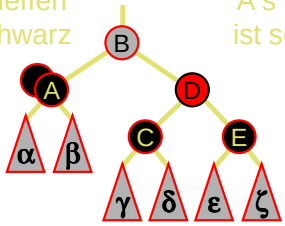
A‘s Bruder
ist schwarz



Fallunterscheidung Delete()

Beide Neffen
auch schwarz

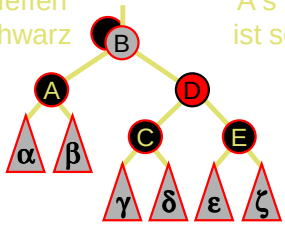
A‘s Bruder
ist schwarz

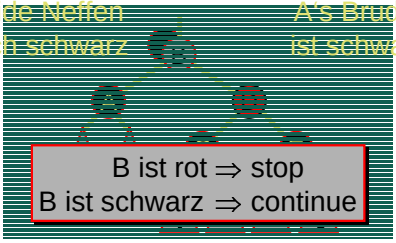


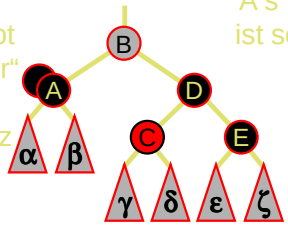
Fallunterscheidung Delete()

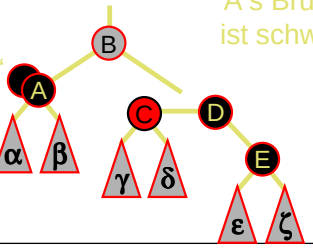
Beide Neffen
auch schwarz

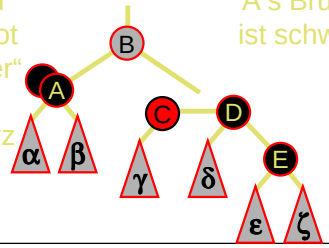
A‘s Bruder
ist schwarz

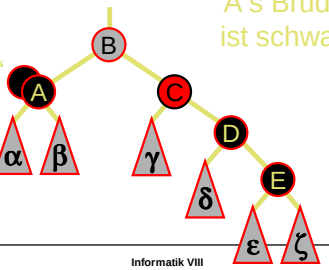


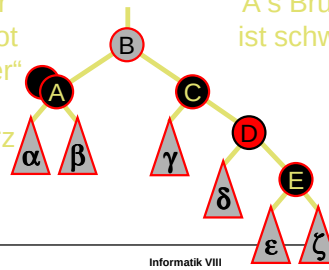












Fallunterscheidung Delete()

Beide Neffen auch schwarz

A‘s Bruder ist schwarz

B ist rot ⇒ stop
B ist schwarz ⇒ continue

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„innerer“ Neffe rot „äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„innerer“ Neffe rot „äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„innerer“ Neffe rot „äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„innerer“ Neffe rot „äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„innerer“ Neffe rot „äußerer“ Neffe schwarz

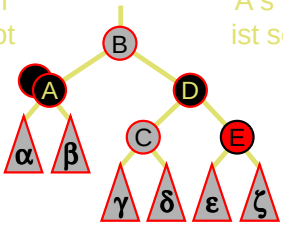
A‘s Bruder ist schwarz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Fallunterscheidung Delete()

„äußerer“ Neffe rot

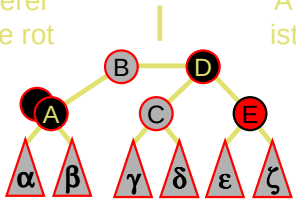
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

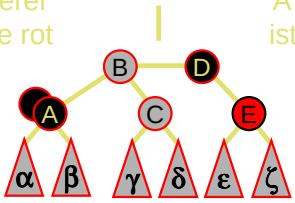
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

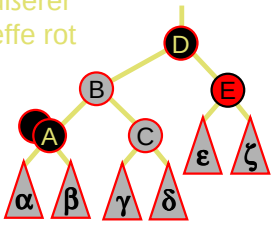
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

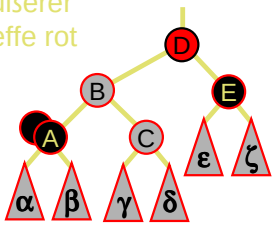
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

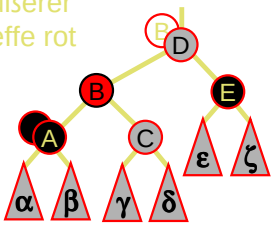
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

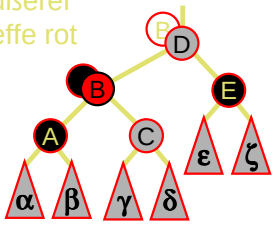
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

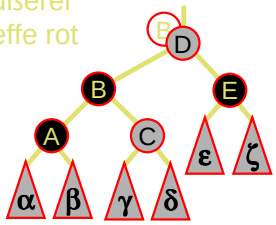
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

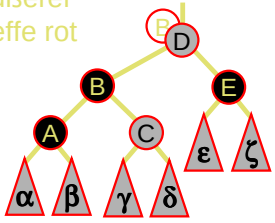
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

A‘s Bruder ist schwarz



Balancierung abgeschlossen ! (Farbe von B war konsistent)

Fallunterscheidung Delete()

1) Bruder ist rot
   - *Rotation, Umfärben ⇒ Bruder schwarz*
2) Bruder ist schwarz
   a) beide Neffen schwarz
      - Umfärben ... continue ?
   b) innerer Neffe rot, äußerer schwarz
      - *Rotation, Umfärben ⇒ Fall 2c*
   c) äußerer Neffe rot
      - Rotation, Umfärben ... done !

Aufwandsabschätzung

- Suchen des Löschknotens: O(log(n))
- Löschen: O(1)
- Konsistenzwiederherstellung
  - Umfärbe-Schritte: O(log(n))
  - Rotationen: O(1) !!! (maximal drei)
- Insgesamt: O(log(n))

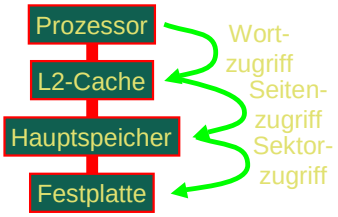
(2.5) Bäume

- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

B-Bäume

- Alternative Interpretation/Implementierung der Rot/Schwarz-Bäume führt auf [2-4]-Bäume
- Diese sind ein Spezialfall des allgemeineren Konzepts von B-Bäumen
- Vorteil: größere Mengen von Knoten werden zusammengefaßt und passen dadurch besser in einen Festplatten-Block
- ***Hysterese:*** nicht in jedem Schritt muß re-balanciert werden.

Speicherhierarchie



Speicherhierarchie

- Mit zunehmendem „Abstand“ vom Prozessor ...
  - *Steigt* die Speicherkapazität
  - *Wächst* die zugreifbare Blockgröße
  - *Sinkt* die Zugriffsgeschwindigkeit

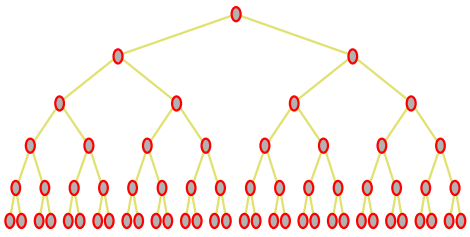
Speicherhierarchie

- Effiziente Algorithmen ...
  - führen häufige Berechnungen auf kleinen Datenmengen durch („innere Schleife“)
  - minimieren die Zugriffe auf externe Speicher-Ebenen
  - passen die Größe der Datenstrukturen an die jeweiligen Blockgrößen an

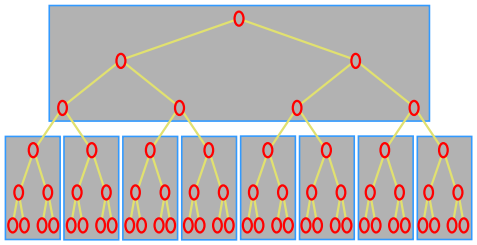
Speicherhierarchie

- Bei der Suche in ***externen*** Binär-Baumstrukturen hängt die Zugriffszeit im wesentlichen von der Verteilung der Knoten auf Festplatten-Sektoren ab.
- Fasse Teilbäume in Sektoren zusammen

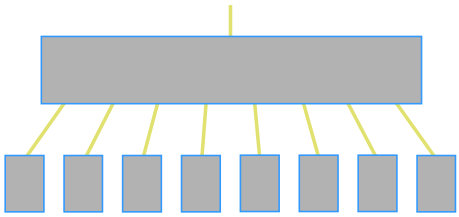
Speicherhierarchie



Speicherhierarchie



Speicherhierarchie



N-äre Suchbäume

- Jeder Knoten hat bis zu n-1 verschiedene Schlüssel und bis zu n Nachfolger
- In K.n wird der aktuelle ***Füllungsgrad*** gespeichert
- In K.x[1], ..., K.x[K.n] stehen die Schlüssel
- In K.s[0], ..., K.s[K.n] stehen die Nachfolger
- Der Bool-Wert K.leaf zeigt, ob K ein ***Blatt*** ist

N-äre Suchbäume

- Seien $y_1, ..., y_m$ die Schlüssel, die in einem Teil-Baum K.s[i] vorkommen, dann gilt
  $K.x[i] < y_1, ..., y_m < K.x[i+1]$
- Für die Schlüssel y_i aus K.s[0] gilt
  $y_i < K.x[1]$
- Für die Schlüssel y_i aus K.s[K.n] gilt
  $K.x[K.n] < y_i$

N-äre Suchbäume

- Wähle n so, dass die Daten genau in einen Festplattensektor passen
- Die Höhe eines n-ären Suchbaums mit k Knoten beträgt maximal
  $O(\log_t(k)) = O(\log(k) / \log(t))$
  wenn der Füllgrad mindestens $t \leq n$ ist

N-äre Suchbäume

- Halte den Wurzelknoten immer im Hauptspeicher (für *jeden* Zugriff notwendig)
- Erwartete Festplattenzugriffe $O(\log_t(k))$
- Beispiel:
  - 10^8 Telefonnummern in Deutschland
  - Blockgröße 512
  - Maximal 2 Festplattenzugriffe

B-Bäume

- *„Balancierte n-äre Suchbäume“*
- Alle Blätter haben dieselbe Tiefe
- Alle Knoten, außer der Wurzel, haben mindestens t-1 Schlüssel mit $t \geq 2$
- Alle inneren Knoten, außer der Wurzel, haben mindestens t Nachfolger
- Normalerweise $t = n / 2$

B-Bäume

- Die meisten Daten stehen in Blättern
- Re-Balancierung durch Rotationen
- „Hysterese“
  - Insert() in Blätter mit Füllgrad $< n$ benötigen keine Re-Balancierung
  - Delete() aus Blättern mit Füllgrad $\geq t$ benötigen keine Re-Balancierung

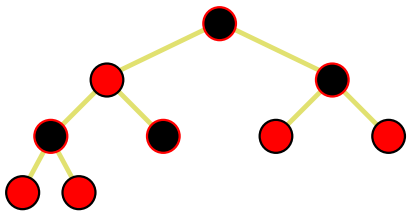
B-Bäume

- Was ist der Zusammenhang mit Rot/Schwarz-Bäumen?
  - Die schwarzen Knoten entsprechen den B-Baum Knoten
  - Die evtl. roten Nachfolger eines schwarzen Knotens werden demselben B-Baum Knoten zugeordnet

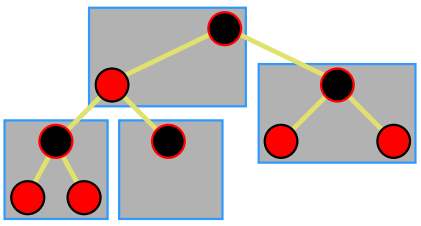
B-Bäume

- Was ist der Zusammenhang mit Rot/Schwarz-Bäumen?
  - Minimaler Füllgrad: 2 Nachfolger
  - Maximaler Füllgrad: 4 Nachfolger
  - $n = 4, \quad t = n / 2 = 2$
  - [2,4]-Bäume

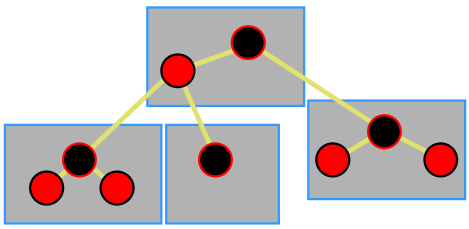
2-4-Bäume



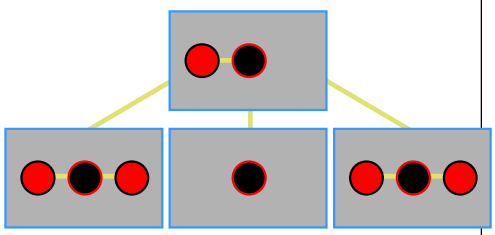
2-4-Bäume



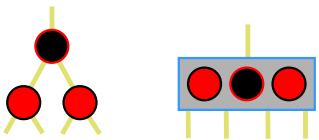
2-4-Bäume



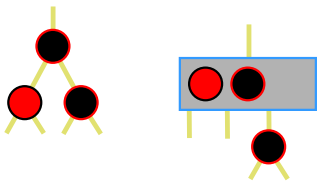
2-4-Bäume



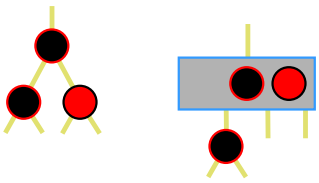
2-4-Bäume



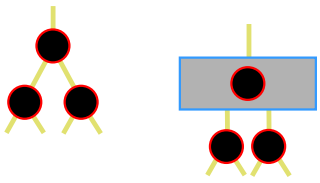
2-4-Bäume



2-4-Bäume



2-4-Bäume



(2.5) Bäume

- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

Rot/Schwarz-Bäume

- Delete()
  - Wie bei normalen binären Suchbäumen wird der Knoten entweder direkt gelöscht oder sein Successor wird gelöscht und überschreibt den eigentlichen Knoten
  - Konsistenzwiederherstellung
    - Wenn der Knoten rot war, ist nichts zu tun
    - Wann der Knoten schwarz war, gibt es 4 Fälle

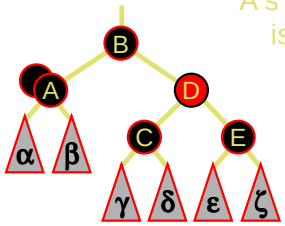
Rot/Schwarz-Bäume

- Delete()
  - Falls der (tatsächlich) entfernte Knoten schwarz ist, fehlt auf allen Pfaden innerhalb des entsprechenden Teil-Baumes eine schwarze Marke
  - Weise diese schwarze Marke dem Nachfolger des gelöschten Knotens zu.

    Dadurch wird dieser schwarz-rot oder doppel-schwarz
  - Verschiebe die schwarze Marke im Baum nach oben

Fallunterscheidung Delete()

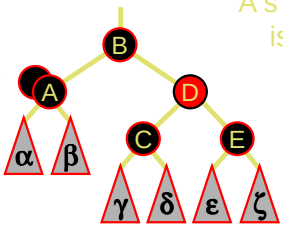
A's Bruder ist rot



Fallunterscheidung Delete()

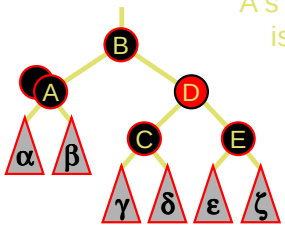
C und E müssen schwarz sein!

A's Bruder ist rot



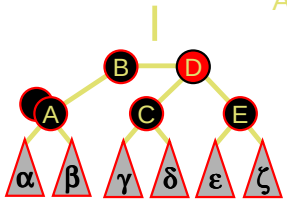
Fallunterscheidung Delete()

A's Bruder ist rot



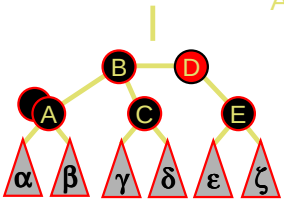
Fallunterscheidung Delete()

A‘s Bruder ist rot



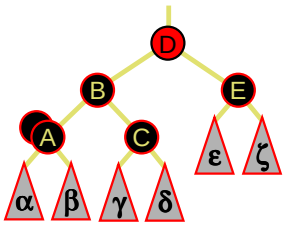
Fallunterscheidung Delete()

A‘s Bruder ist rot



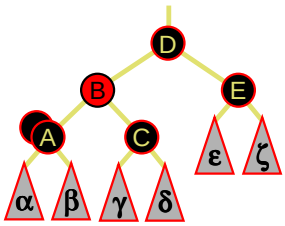
Fallunterscheidung Delete()

A‘s Bruder ist rot



Fallunterscheidung Delete()

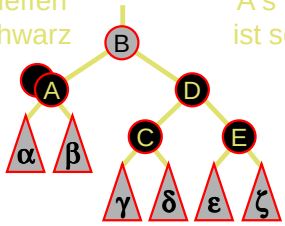
A‘s Bruder ist rot



Fallunterscheidung Delete()

Beide Neffen auch schwarz

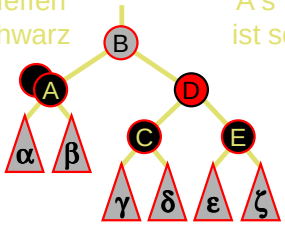
A‘s Bruder ist schwarz

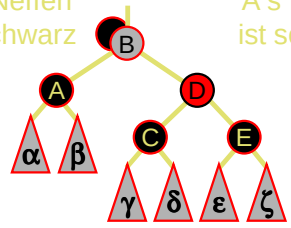


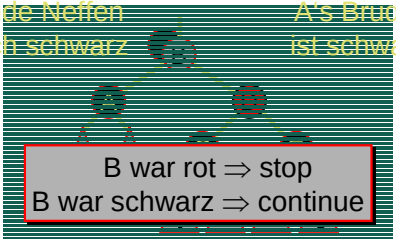
Fallunterscheidung Delete()

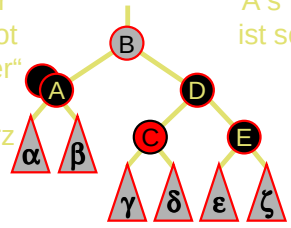
Beide Neffen auch schwarz

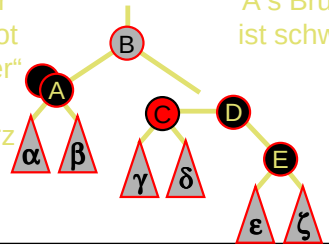
A‘s Bruder ist schwarz

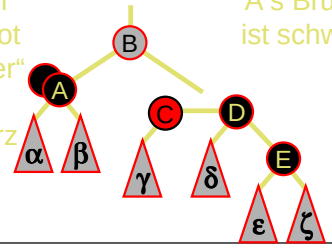


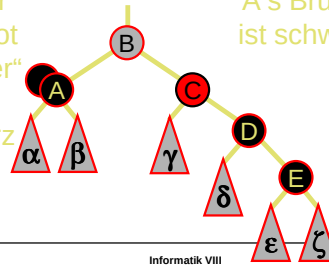












Fallunterscheidung Delete()

Beide Neffen auch schwarz

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

Beide Neffen auch schwarz

A‘s Bruder ist schwarz

B war rot ⇒ stop
B war schwarz ⇒ continue

Fallunterscheidung Delete()

„innerer“ Neffe rot
„äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„innerer“ Neffe rot
„äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„innerer“ Neffe rot
„äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

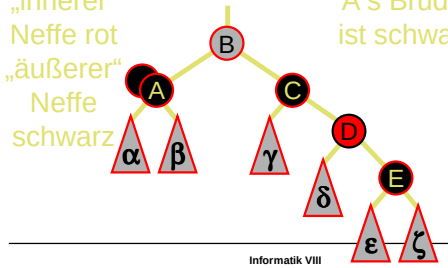
„innerer“ Neffe rot
„äußerer“ Neffe schwarz

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„innerer“ Neffe rot
„äußerer“ Neffe schwarz

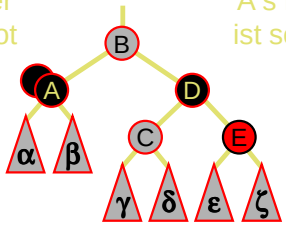
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

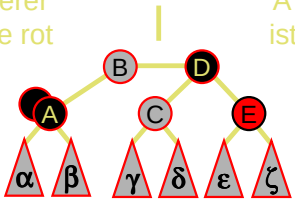
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

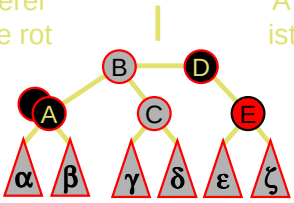
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

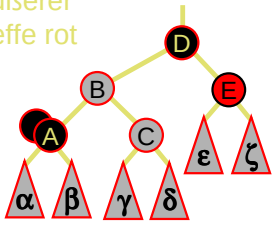
A‘s Bruder ist schwarz



Fallunterscheidung Delete()

„äußerer“ Neffe rot

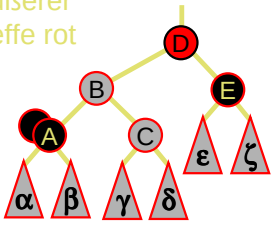
A‘s Bruder ist schwarz

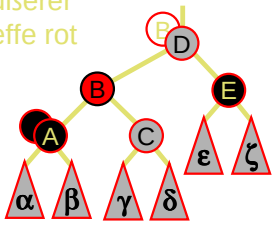


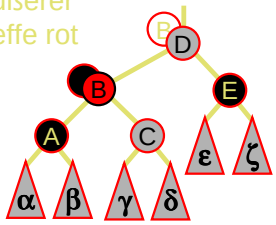
Fallunterscheidung Delete()

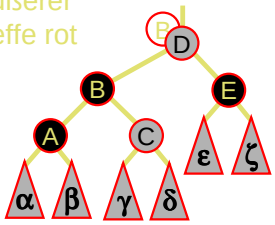
„äußerer“ Neffe rot

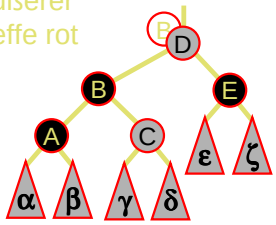
A‘s Bruder ist schwarz











Fallunterscheidung Delete()

„äußerer“ Neffe rot

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„äußerer“ Neffe rot

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„äußerer“ Neffe rot

A‘s Bruder ist schwarz

Fallunterscheidung Delete()

„äußerer“ Neffe rot

A‘s Bruder ist schwarz

Balancierung abgeschlossen ! (Farbe von B war konsistent)

Fallunterscheidung Delete()

1) Bruder ist rot
    - *Rotation, Umfärben ⇒ Bruder schwarz*
2) Bruder ist schwarz
    - a) beide Neffen schwarz
        - Umfärben ... continue ?
    - b) innerer Neffe rot, äußerer schwarz
        - *Rotation, Umfärben ⇒ Fall 2c*
    - c) äußerer Neffe rot
        - Rotation, Umfärben ... done !

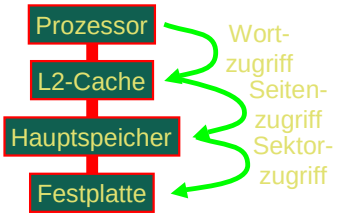
Aufwandsabschätzung

- Suchen des Löschknotens: O(log(n))
- Löschen: O(1)
- Konsistenzwiederherstellung
    - Umfärbe-Schritte: O(log(n))
    - Rotationen: O(1) !!! (maximal drei)
- Insgesamt: O(log(n))

(2.5) Bäume

- (2.5.1) Binäre Suchbäume
- (2.5.2) Optimale Suchbäume
- (2.5.3) Balancierte Bäume
  - AVL-Bäume
  - Red-Black Bäume
  - B-Bäume

Speicherhierarchie



Speicherhierarchie

- Mit zunehmendem „Abstand“ vom Prozessor ...
  - *Steigt* die Speicherkapazität
  - *Wächst* die zugreifbare Blockgröße
  - *Sinkt* die Zugriffsgeschwindigkeit

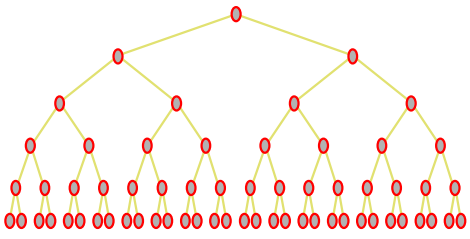
Speicherhierarchie

- Effiziente Algorithmen ...
  - führen häufige Berechnungen auf kleinen Datenmengen durch („innere Schleife“)
  - minimieren die Zugriffe auf externe Speicher-Ebenen
  - passen die Größe der Datenstrukturen an die jeweiligen Blockgrößen an

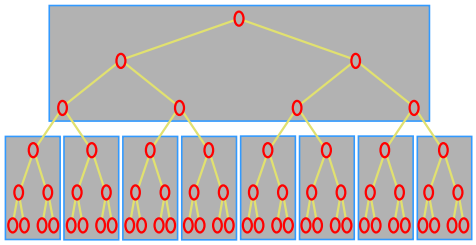
Speicherhierarchie

- Bei der Suche in ***externen*** Binär-Baumstrukturen hängt die Zugriffszeit im wesentlichen von der Verteilung der Knoten auf Festplatten-Sektoren ab.
- Fasse Teilbäume in Sektoren zusammen

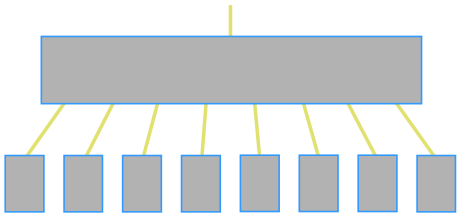
Speicherhierarchie



Speicherhierarchie



Speicherhierarchie



N-äre Suchbäume

- Jeder Knoten hat bis zu n-1 verschiedene Schlüssel und bis zu n Nachfolger
- In K.n wird der aktuelle ***Füllungsgrad*** gespeichert
- In K.x[1], ..., K.x[K.n] stehen die Schlüssel
- In K.s[0], ..., K.s[K.n] stehen die Nachfolger
- Der Bool-Wert K.leaf zeigt, ob K ein ***Blatt*** ist

N-äre Suchbäume

- Seien $y_1, ..., y_m$ die Schlüssel, die in einem Teil-Baum K.s[i] vorkommen, dann gilt
  $K.x[i] < y_1, ..., y_m < K.x[i+1]$
- Für die Schlüssel y_i aus K.s[0] gilt
  $y_i < K.x[1]$
- Für die Schlüssel y_i aus K.s[K.n] gilt
  $K.x[K.n] < y_i$

B-Bäume

- *„Balancierte n-äre Suchbäume“*
- Alle Blätter haben dieselbe Tiefe
- Alle Knoten, außer der Wurzel, haben mindestens t-1 Schlüssel mit $t \geq 2$
- Alle inneren Knoten, außer der Wurzel, haben mindestens t Nachfolger
- Normalerweise $t = n / 2$

B-Bäume

- Die meisten Daten stehen in Blättern
- Re-Balancierung
- „Hysterese“
  - Insert() in Blätter mit Füllgrad < n benötigen keine Re-Balancierung
  - Delete() aus Blättern mit Füllgrad ≥ t benötigen keine Re-Balancierung

B-Bäume

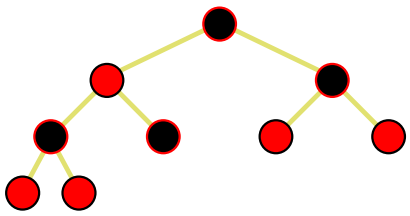
- Was ist der Zusammenhang mit Rot/Schwarz-Bäumen?
  - Die schwarzen Knoten entsprechen den B-Baum Knoten
  - Die evtl. roten Nachfolger eines schwarzen Knotens werden demselben B-Baum Knoten zugeordnet

---

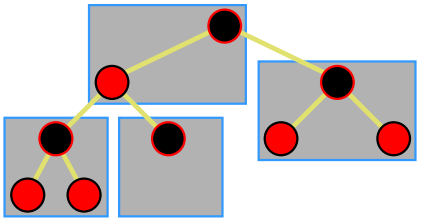
B-Bäume

- Was ist der Zusammenhang mit Rot/Schwarz-Bäumen?
  - Minimaler Füllgrad: 2 Nachfolger
  - Maximaler Füllgrad: 4 Nachfolger
  - n = 4, t = n / 2 = 2
  - [2,4]-Bäume

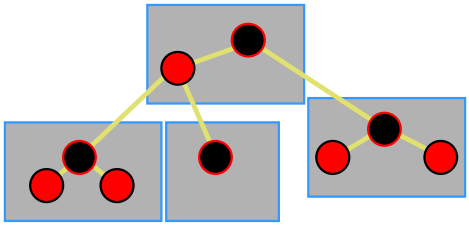
2-4-Bäume



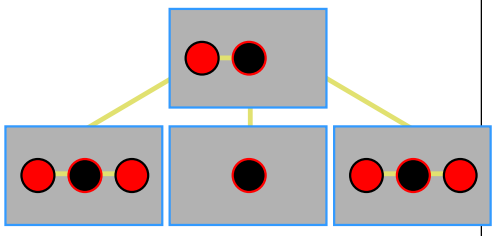
2-4-Bäume



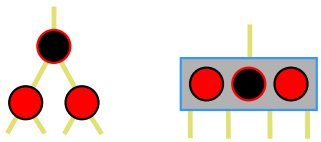
2-4-Bäume



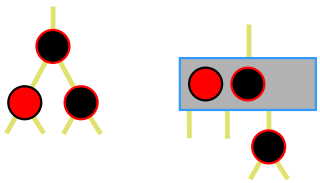
2-4-Bäume



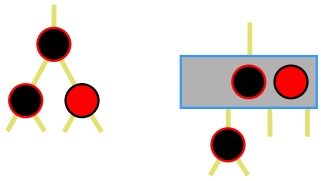
2-4-Bäume



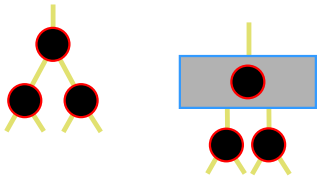
2-4-Bäume



2-4-Bäume



2-4-Bäume



B-Bäume

- Operationen
  - Search()
  - Insert()
  - Delete()
- In den meisten Fällen kommen die Operationen ohne Re-Strukturierung aus
- Sonst: O(1) verallgemeinerte Rotationen

Zugriff auf externen Speicher

- DiskRead(N), DiskWrite(N)
- N = ...
  DiskRead(N);
  access / modify the contents of N
  DiskWrite(N); *// only if modified*
  access but don‘t modify the contents of N

Create()

- Create()
  N = AllocateNode();
  N.leaf = true;
  N.n = 0
  DiskWrite();

Search()

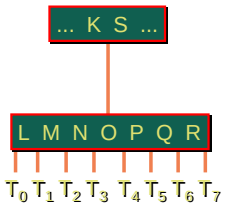
- Search(X,R)

```
for (i = 1 ; i ≤ R.n ; i++)
    if (X ≤ R.x[i])  break;
if (i ≤ R.n ∧ X = R.x[i])
    return (R,i);
else
    if (R.leaf)
        return NIL;
    else
        DiskRead(R.s[i-1]);
        Search(X,R.s[i-1]);
```

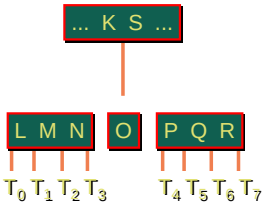

Insert()

- Bei Binärbäumen wird ***immer*** ein neuer Blattknoten ergänzt
- Bei B-Bäumen ist kein neuer Knoten nötig, solange der Blattknoten noch nicht voll ist
- Erst wenn der Füllgrad über n steigt, wird das Blatt ***geteilt*** ... $n \geq 2t$

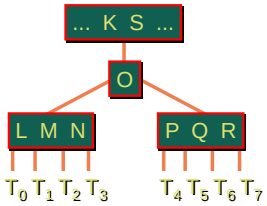
Split()



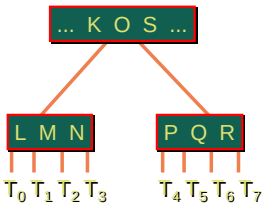
Split()



Split()



Split()

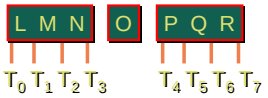


Split()

Spezialfall
Wurzelknoten

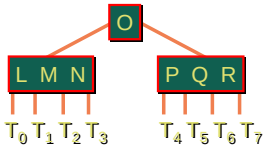
Split()

Spezialfall
Wurzelknoten



Split()

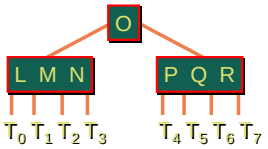
Spezialfall
Wurzelknoten



Split()

B-Bäume wachsen
an der Wurzel und
nicht an den Blättern

Spezialfall
Wurzelknoten



Insert()

- Naiver Ansatz:
  - Suche das entsprechende Blatt
  - evtl. Splitting, falls überfüllt
  - Kann zurück nach oben propagiert werden, wenn Vaterknoten ebenfalls voll
  - ***Problem:*** auf die Knoten entlang des Pfades wird jeweils ***zweimal*** zugegriffen

Insert()

- One-Pass Algorithmus
  - Teile die Knoten entlang des Suchpfades bereits auf dem „Hinweg“, wenn diese schon voll sind
  - Evtl. unnötige Split-Operationen amortisieren sich durch den schnelleren Average-Case

Insert()

- Insert(N)

```
R = root;
if (R.n = 2t-1)
    S = AllocateNode();
    root = S;   S.leaf = false;
    S.n = 0;    S.s[0] = R;
    Split(S,R);
    InsertNonFull(N,S);
else
    InsertNonFull(N,R);
```


Insert()

- InsertNonFull(N,R) *// assert: R not full !*

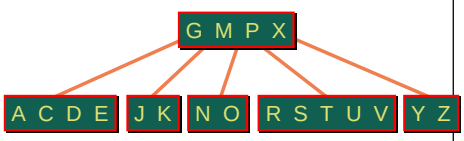
```
if (R.leaf)
    find correct position for N and insert
    DiskWrite(R);
else
    find correct subtree R.s[i];
    DiskRead(R.s[i]);
    if (R.s[i].n = 2t-1)
        Split(R,R.s[i]);
        if (N.x > R.x[i])  i++;
    InsertNonFull(N,R.s[i]);
```


Beispiel

- Sequenz von Einfügeoperationen

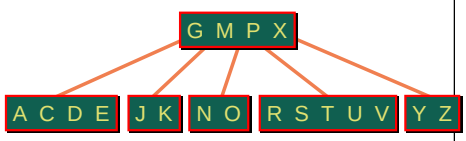
Beispiel

Initial tree

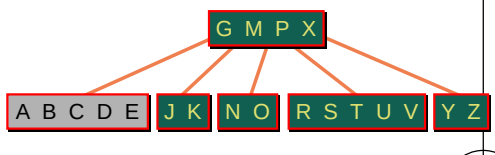


Beispiel

Insert(B)

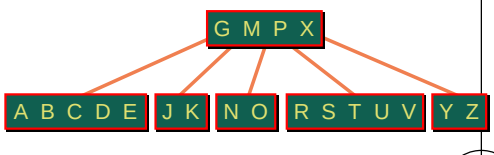


Beispiel



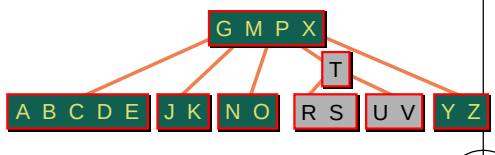
Beispiel

Insert(Q)



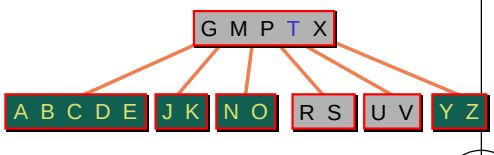
Beispiel

Insert(Q)

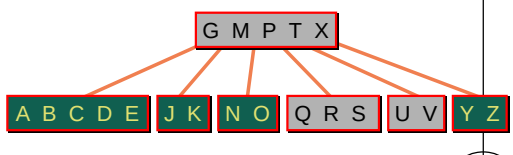


Beispiel

Insert(Q)

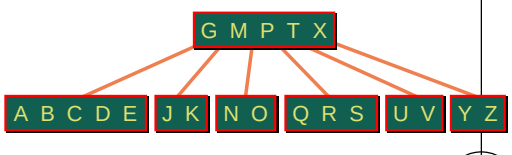


Beispiel



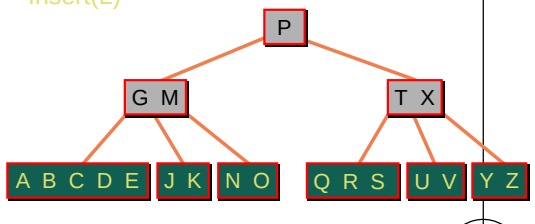
Beispiel

Insert(L)

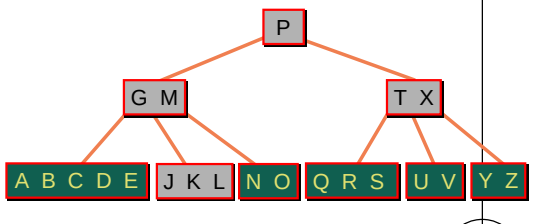


Beispiel

Insert(L)

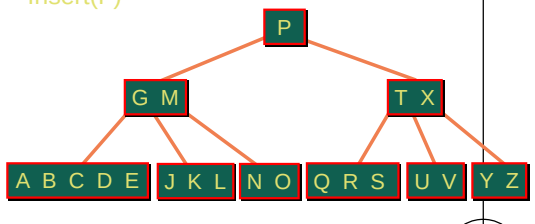


Beispiel



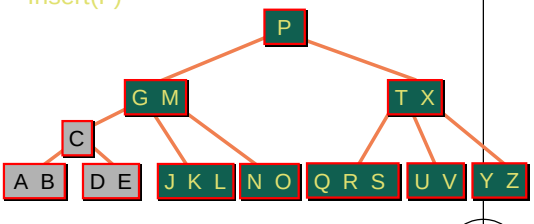
Beispiel

Insert(F)



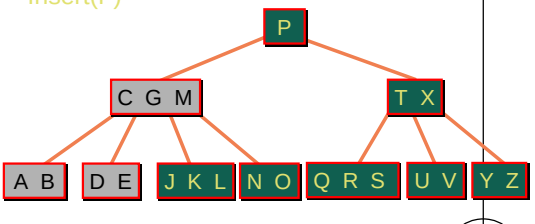
Beispiel

Insert(F)



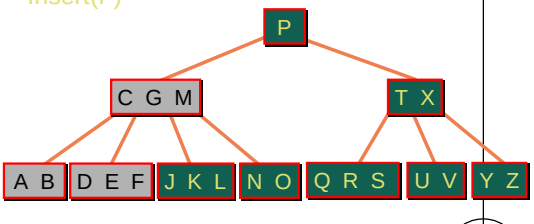
Beispiel

Insert(F)

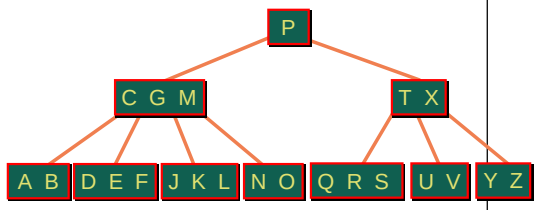


Beispiel

Insert(F)



Beispiel



Delete()

- Unkritisch für ***Blatt***knoten mit hinreichendem Füllungsgrad (häufigster Fall)
- Bei Unterlauf müssen Knoten verschmolzen werden
- Kann/muss nach oben propagiert werden
- One-Pass-Formulierung ???

Delete()

- Der Delete-Algorithmus sucht den Schlüssel entlang eines Pfades
- Eine Verschmelzung muß nur stattfinden, wenn der entsprechende Knoten einen Füllgrad $\leq t-1$ hat
- Teste ***vor*** dem Abstieg, ob der Nachfolger Füllgrad $\geq t$ hat (dann ist Verschmelzung ausgeschlossen)
- Falls dies nicht der Fall ist, erzwinge dies durch direkte Umstrukturierung

Delete()

- Wie bei Insert(), sorgen wir dafür, dass der aktuelle Knoten hinreichend gefüllt ist, so dass keine Verschmelzung notwendig wird (Schleifeninvariante)
- Falls ein Knoten nicht hinreichend gefüllt wäre, hätten wir das bereits bei der Bearbeitung des Vaterknotens bemerkt und behoben.

Delete()

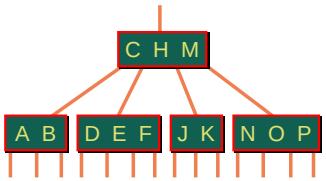
- Fallunterscheidung:
  - Blattknoten
    - Nach Voraussetzung hinreichend gefüllt
    - Schlüssel kann einfach entfernt werden

Delete()

- Fallunterscheidung:
  - Blattknoten
  - Innerer Knoten
    - Nach Voraussetzung hinreichend gefüllt
    - Beim Löschen müssen die nachfolgenden Teilbäume korrekt angeordnet werden
    - „Stehle“ ein Element ...

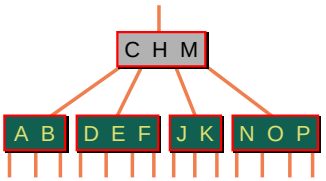
Steal()

Entferne: H t = 3



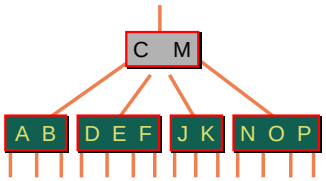
Steal()

Entferne: H t = 3



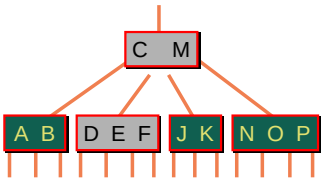
Steal()

Entferne: H t = 3



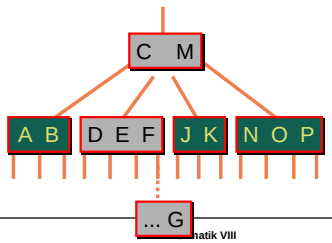
Steal()

Finde: Predecessor(H) t = 3



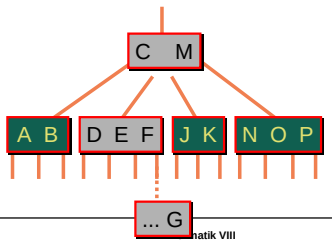
Steal()

Finde: Predecessor(H) t = 3



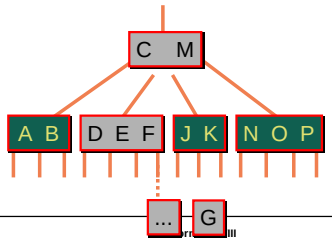
Steal()

Lösche Predecessor(H) rekursiv t = 3



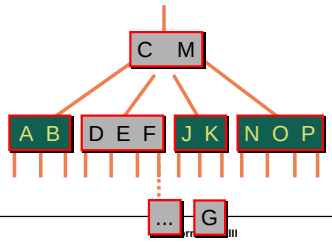
Steal()

Lösche Predecessor(H) rekursiv t = 3



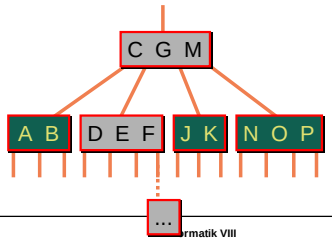
Steal()

Kopiere Predecessor(H) an H's Stelle t = 3



Steal()

Kopiere Predecessor(H) an H's Stelle t = 3

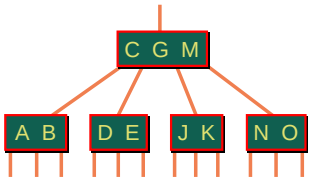


Delete()

- Fallunterscheidung:
  - Blattknoten
  - Innerer Knoten
    - Nach Voraussetzung hinreichend gefüllt
    - Beim Löschen müssen die nachfolgenden Teilbäume korrekt angeordnet werden
    - „Stehle" ein Element ...
    - Verschmelze die Nachfolger

Merge()

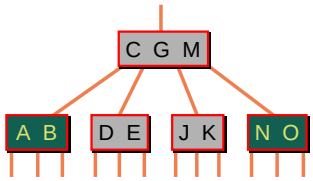
t = 3
n = 6



Merge()

Entferne: G

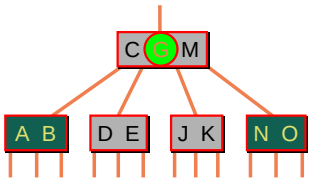
t = 3
n = 6



Merge()

Entferne: G

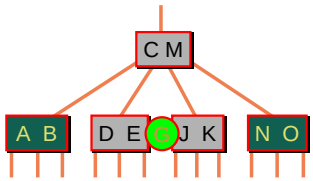
t = 3
n = 6



Merge()

Entferne: G

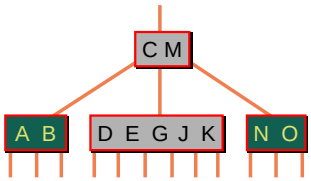
t = 3
n = 6



Merge()

Entferne: G

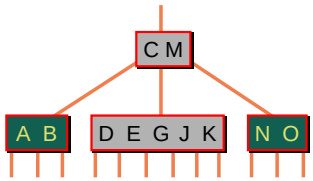
t = 3
n = 6



Merge()

Entferne: G (rekursiver Aufruf)

t = 3
n = 6

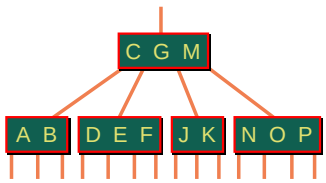


Delete()

- Fallunterscheidung:
  - Blattknoten
  - Innerer Knoten
  - Weitersuchen
    - Sorge für Aufrechterhaltung der Schleifeninvariante
    - Rotiere Elemente
    - Verschmelze die Nachfolger

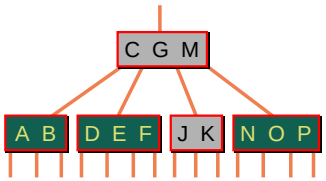
Rotate()

t = 3



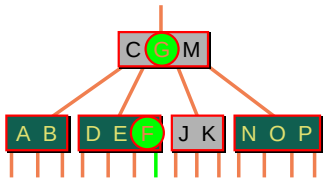
Rotate()

t = 3



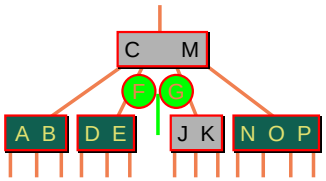
Rotate()

t = 3



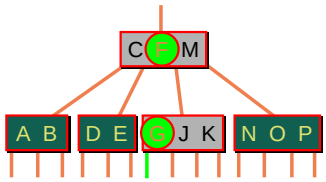
Rotate()

t = 3



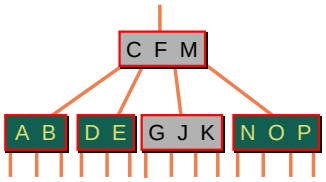
Rotate()

t = 3



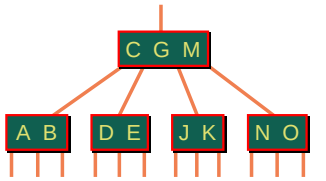
Rotate()

t = 3



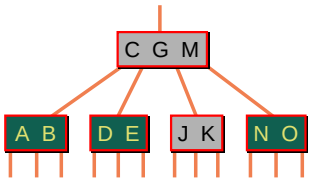
Merge()

t = 3
n = 6



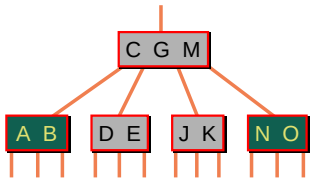
Merge()

t = 3
n = 6



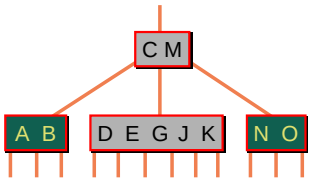
Merge()

t = 3
n = 6



Merge()

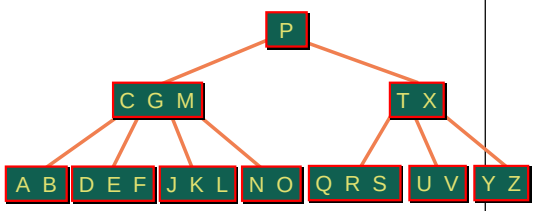
t = 3
n = 6



Beispiel

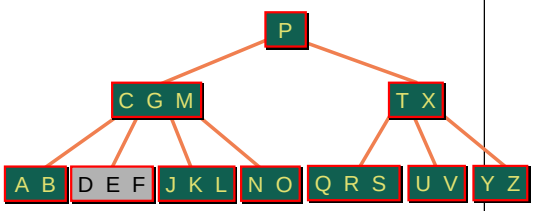
- Sequenz von Löschungsoperationen

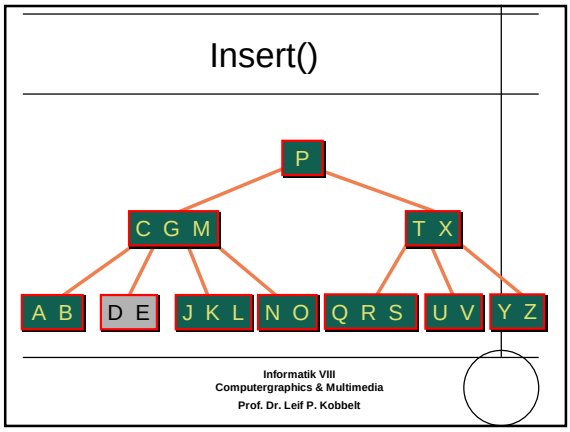
Insert()

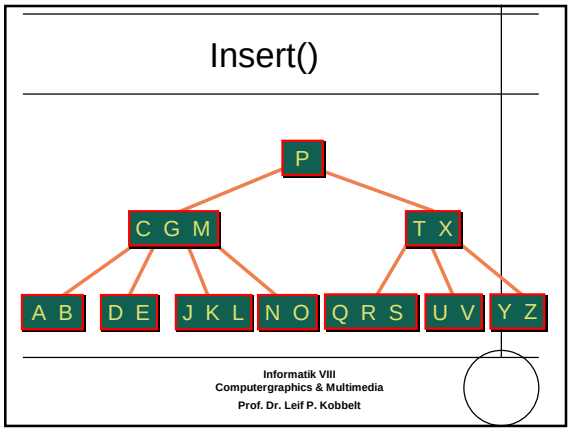


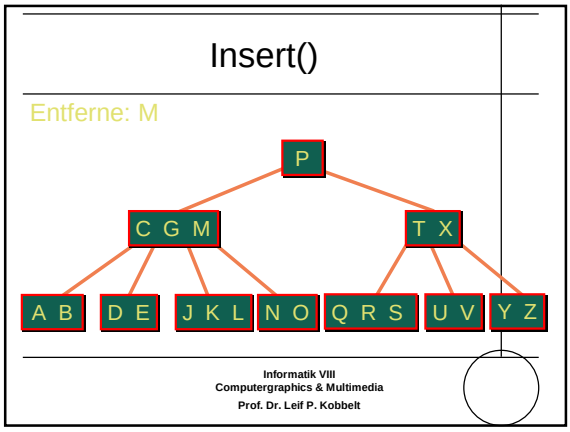
Insert()

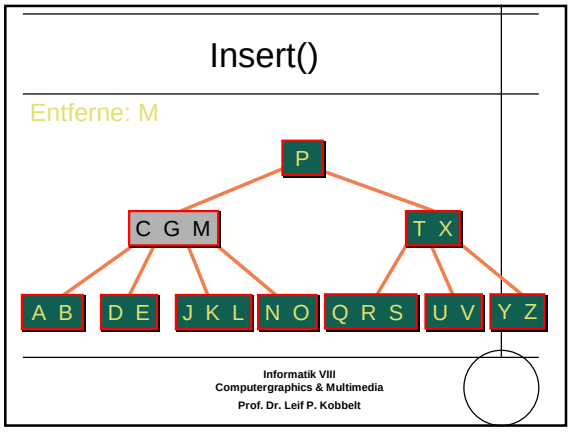
Entferne: D

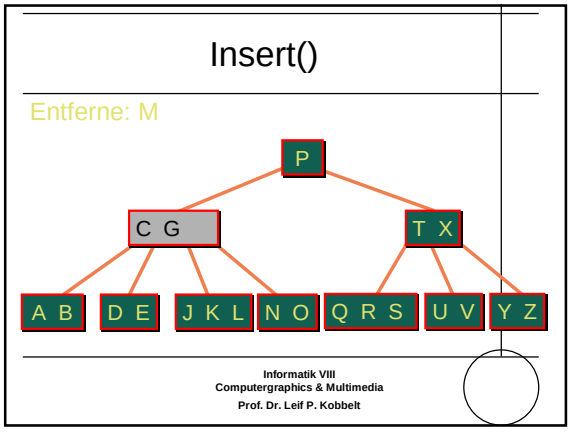


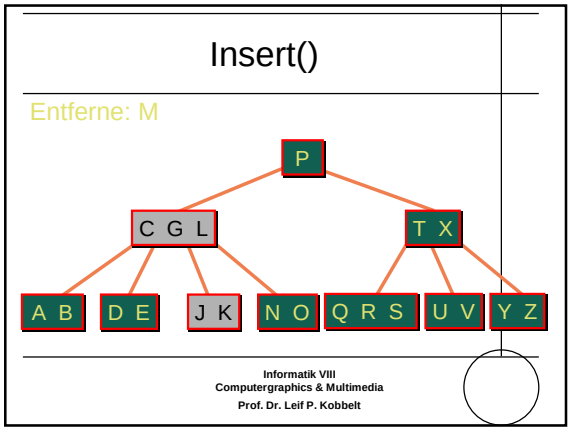












Insert()

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Insert()

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Insert()

Entferne: M

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Insert()

Entferne: M

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Insert()

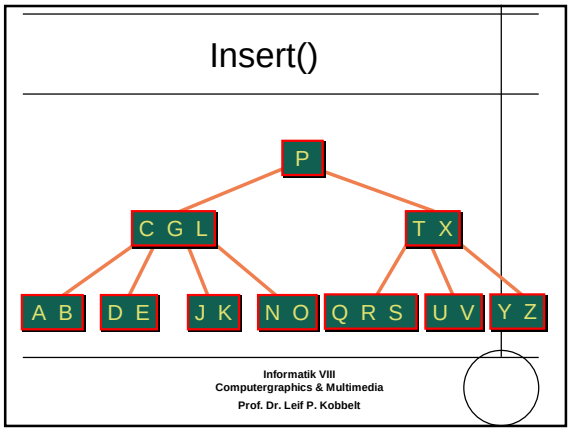
Entferne: M

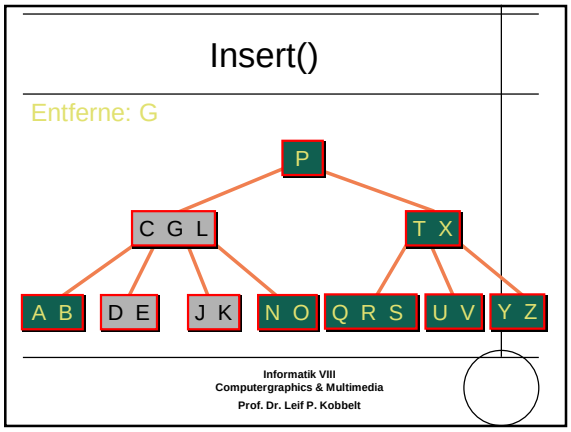
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

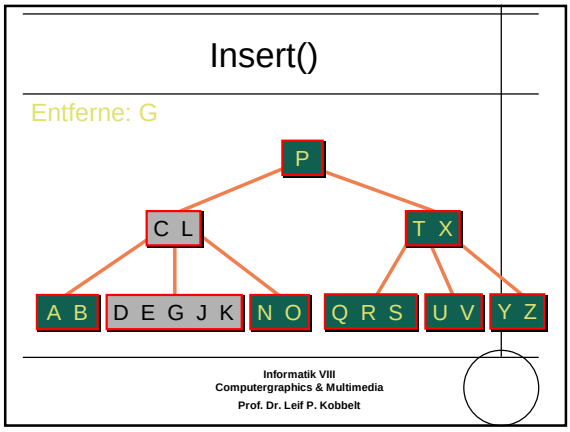
Insert()

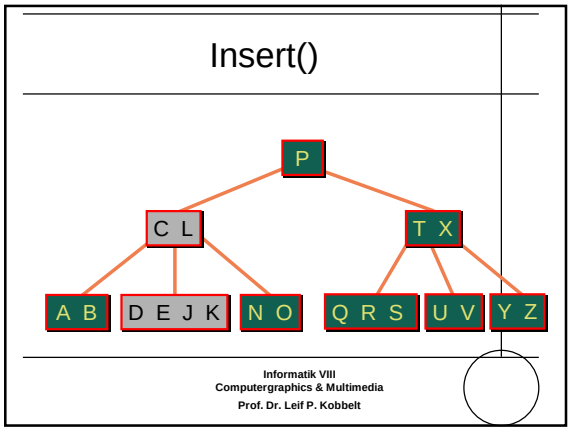
Entferne: M

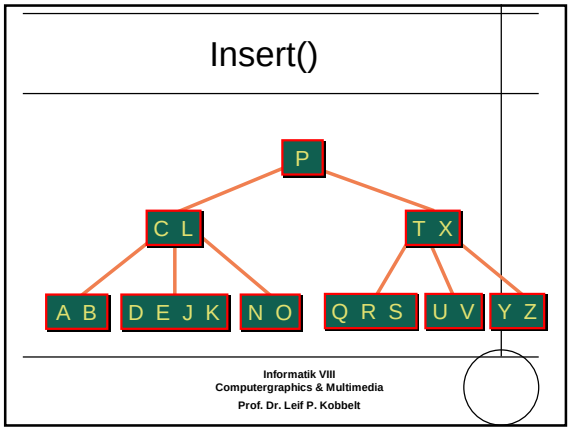
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

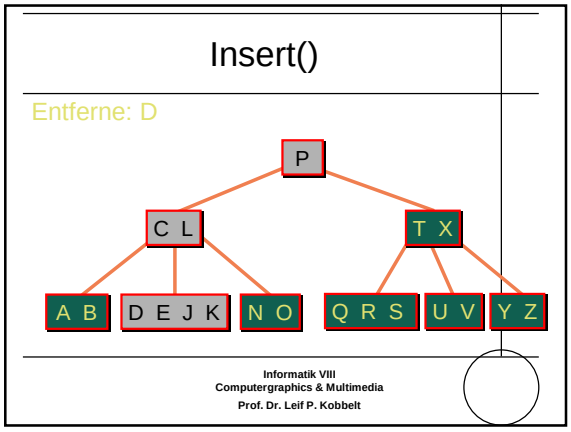

Insert()
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

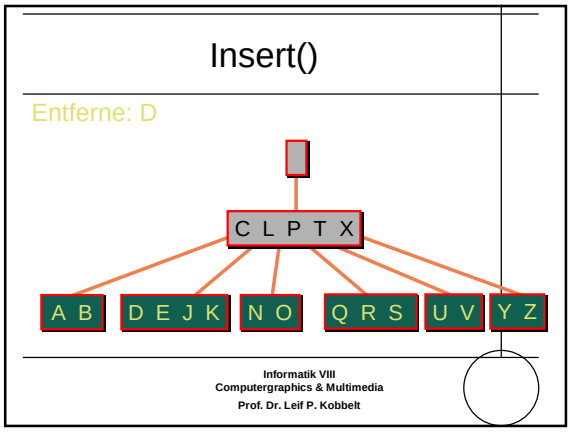

Insert()
Entferne: G
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

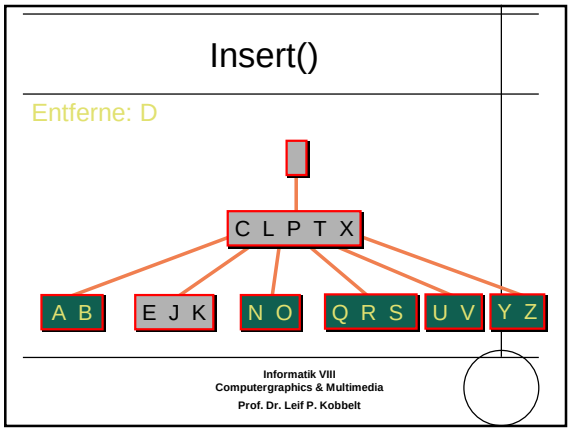

Insert()
Entferne: G
P
C L
T X
A B
D E G J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

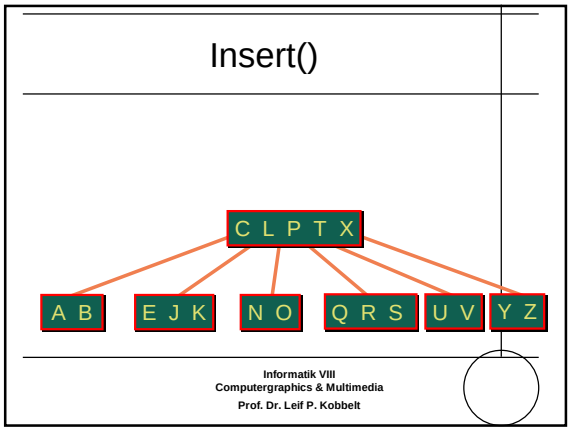

Insert()
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

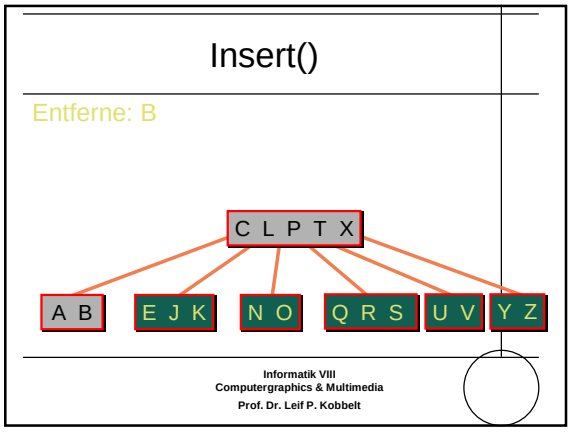

Insert()
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

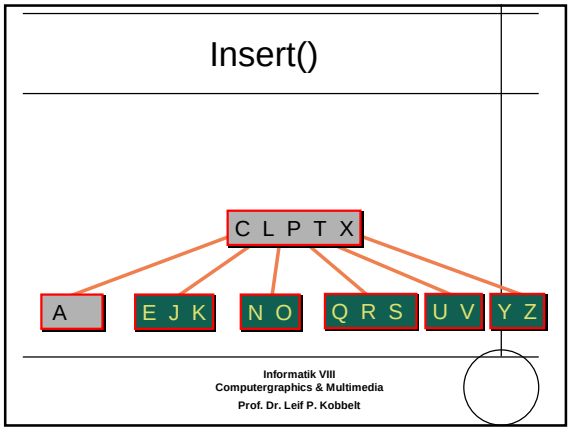

Insert()
Entferne: D
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

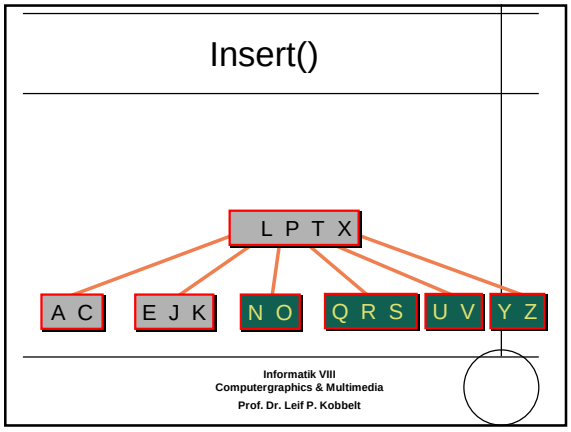

Insert()
Entferne: D
C L P T X
A B
D E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Insert()
Entferne: D
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

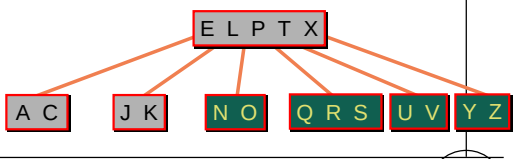

Insert()
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Insert()
Entferne: B
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Insert()
C L P T X
A
E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Insert()
L P T X
A C
E J K
N O
Q R S
U V
Y Z
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Insert()



Zusammenfassung

- B-Bäume
  - Optimierter Zugriff auf externen Speicher durch Blockung mehrerer Schlüssel
  - Minimaler / maximaler Füllungsgrad
  - Garantierte Balancierung
  - Hysterese bei der Umstrukturierung
  - Steal(), Merge(), Rotate()

(2.6) Graphen

1

- (2.6.1) Definitionen, Darstellung
- (2.6.2) Ausspähen von Graphen
- (2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade

Graphen

2

Gerichteter Graph G=(V,E)

- V : Knoten, Vertices
- E : Kanten, Edges

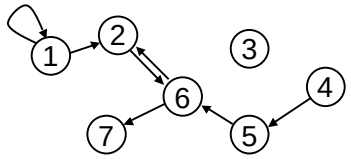
mit $|V| < \infty$ und $E \subseteq V \times V$.

Graphen

3

V = {1,2,3,4,5,6,7}

E = {(1,1),(1,2),(2,6),(6,2),(6,7),(5,6),(4,5)}



Graphen

4

Ungerichteter Graph G=(V,E)

- V : Knoten, Vertices
- E : Kanten, Edges

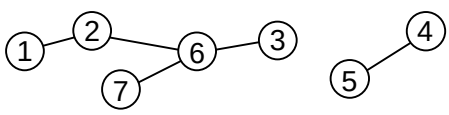
mit $|V| < \infty$ und $E \subseteq V \times V$ symmetrisch.

Graphen

5

V = {1,2,3,4,5,6,7}

E = {(1,2),(2,1),(2,6),(6,2),(6,7),(7,6),
(3,6),(6,3),(5,4),(4,5)}



Graphen

6

Konvention:

Ist keine Verwechslung zu befürchten, so bezeichnen wir die Anzahl |V| der Knoten bzw. |E| Kanten einfach mit V bzw. E.

Adjazenz

7

Wir nennen

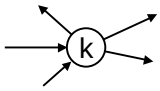
Adj[u] = { v : (u,v) in E }

die Nachbarn von u.

Adjazenz

8

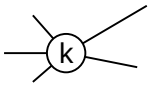
Der Eingangs-/Ausgangsgrad eines Knotens k ist die Zahl der auf k hin- / von k weggerichteten Kanten.



Adjazenz

9

Ungerichteter Graph:
- E ist symmetrisch
- Eingangsgrad = Ausgangsgrad = Grad = Zahl der Nachbarn



Wege

10

Ein Weg oder Pfad der Länge k vom Knoten u zum Knoten v ist eine Sequenz

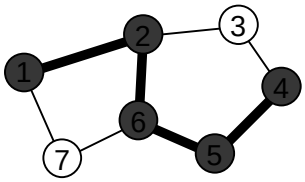
$v_0,v_1,v_2,\ldots,v_{k-1},v_k$

von Knoten mit
- $u = v_0$, $v = v_k$
- $(v_{i-1},v_i) \in E$ für i = 1,2,...,k

Wege

11

1,2,6,5,4



Wege

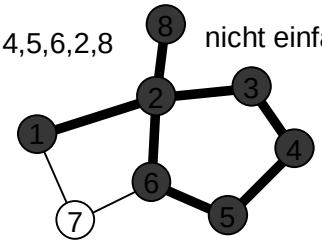
12

Ein Weg heißt einfach, wenn alle seine Knoten verschieden sind.

Wege

13

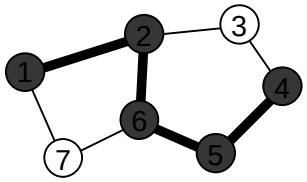
1,2,3,4,5,6,2,8 nicht einfach!



Wege

14

1,2,6,5,6,5,4 nicht einfach!



Zyklen und Kreise

15

Ein Weg

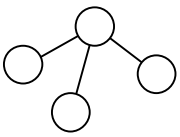
$$v_0, v_1, \ldots, v_k$$

mit $v_0 = v_k$ heißt Zyklus (gerichtete Graphen) bzw. Kreis (ungerichtete Graphen).

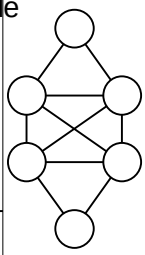
Zyklen und Kreise

16

Ein Eulerkreis eines ungerichteten Graphen ist ein Kreis der jede Kante genau einmal enthält.



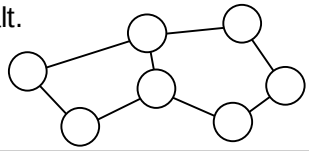
Wann existiert ein Eulerkreis?



Zyklen und Kreise

17

Ein Hamiltonkreis eines ungerichteten Graphen ist ein Kreis der jeden Knoten genau einmal enthält.



Traveling Salesman Problem

Zusammenhang

18

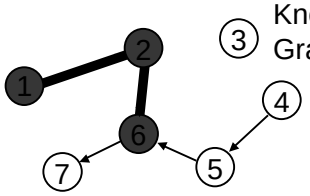
Ein Knoten v ist von einem Knoten u erreichbar, $u \rightsquigarrow v$, wenn es einen Weg von u nach v gibt.

Zusammenhang

19

$1 \rightsquigarrow 6$, aber $6 \not\rightsquigarrow 1$

Isolierter Knoten: Grad = 0



Zusammenhang

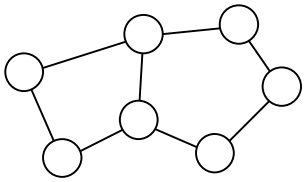
20

Ein ungerichteter Graph heißt zusammenhängend, wenn jedes Knotenpaar durch einen Weg verbunden ist.

Zusammenhang

21

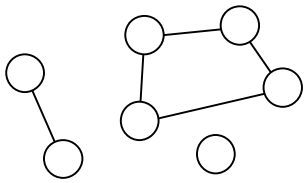
zusammenhängend



Zusammenhang

22

nicht zusammenhängend



Zusammenhang

23

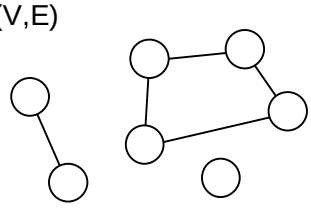
Die Zusammenhangs-komponenten eines ungerichteten Graphen sind die Äquivalenz-klassen der Relation $\rightsquigarrow$

(Zeige: Für ungerichtete Graphen ist $\rightsquigarrow$ eine Äquivalenzrelation!)

Zusammenhang

24

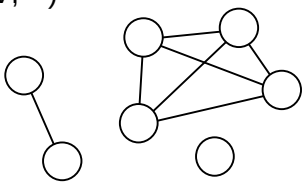
(V,E)



Zusammenhang

25

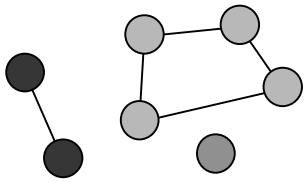
$(V, \rightsquigarrow)$



Zusammenhang

26

3 Zusammenhangskomponenten



Zusammenhang

27

Ein gerichteter Graph heißt schwach, stark, einseitig zusammenhängend, wenn ...

Wald

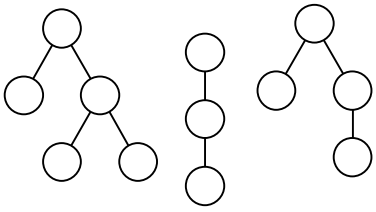
28

Ein ungerichteter Graph G heißt Wald, falls G keinen Kreis enthält. Ist G zusätzlich zusammenhängend, so heisst G ein Baum.

Wald

29

Ein Wald besteht aus Bäumen:



Wald

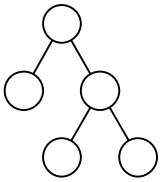
30

Für einen Baum (V,E) gilt:

$|E| = |V| - 1$

$|E| = 4$

$|V| = 5$



Wald

Sei G=(V,E) ein Wald mit k Bäumen. Baum i habe n_i Knoten und m_i Kanten. Dann gilt

$$|E| = \sum_{i=1}^{k} m_i = \sum_{i=1}^{k} (n_i - 1) = \sum_{i=1}^{k} n_i - k = |V| - k$$

Wald

Folge: Ein Graph G=(V,E) mit k Zusammenhangskomponenten enthält genau dann einen Kreis, wenn

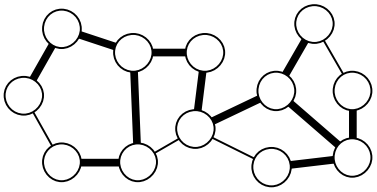
$$|E| > |V| - k$$

Wald

Ein Spannbaum zu einem Graphen G=(V,E) ist ein Baum (V,E´) mit $E' \subseteq E$.

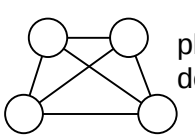
Wald

Spannbaum

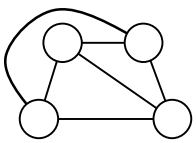


Spezielle Graphen

Ein Graph G=(V,E) heisst planar, wenn er sich überschneidungsfrei in der Ebene abbilden lässt.

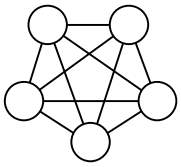


planar, denn



Spezielle Graphen

Ein ungerichteter Graph G=(V,E) ist vollständig, falls $E = V \times V$.



K_5

nicht planar

Gewichtete Graphen

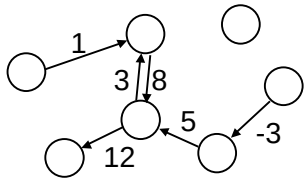
37

Zu einem gewichteten Graph gehört eine Gewichtsfunktion $w: E \rightarrow R$, die jeder Kante e ein Gewicht w(e) zuordnet.

Gewichtete Graphen

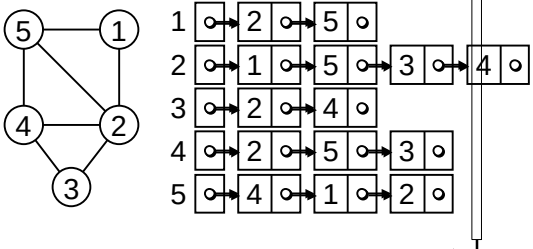
38

gewichteter Graph



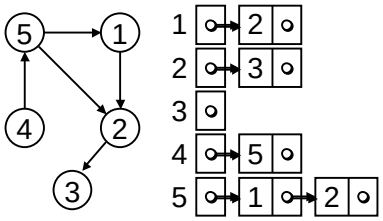
Adjazenzlisten

39



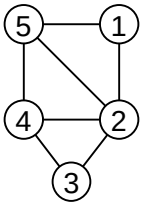
Adjazenzlisten

40



Adjazenzmatrizen

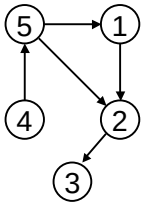
41



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjazenzmatrizen

42



	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	0	0
4	0	0	0	0	1
5	1	1	0	0	0

Speicheraufwand

43

- Adjazenzlisten: O(|V| + |E|)
- Adjazenzmatrix: O($|V|^2$)

Ist nichts anderes angegeben, so verwenden die nachfolgend vorgestellten Algorithmen Adjazenzlisten zur Darstellung der Graphen.

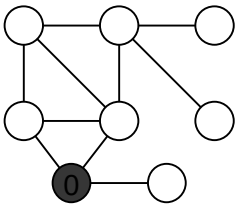
(2.6.2) Ausspähen von Graphen

44

- (2.6.2.1) Breitensuche
- (2.6.2.2) Tiefensuche
- (2.6.2.3) Topologisches Sortieren

Breitensuche

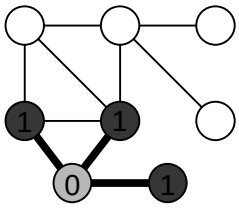
45



Source

Breitensuche

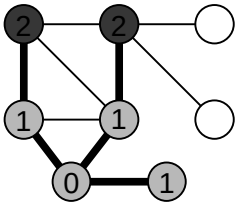
46



Source

Breitensuche

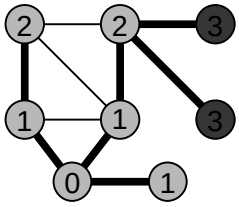
47



Source

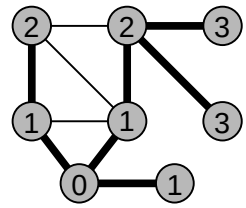
Breitensuche

48



Source

Breitensuche



Source

Breitensuche

- Von einem gegeben ein Startknoten s („source“) ausgehend, werden nacheinander alle von s erreichbaren Knoten besucht.
- Die Grenze zwischen besuchten und nicht besuchten Knoten wird gleichmäßig vorangetrieben („Breadth-First“).

Breitensuche

- Während der Breitensuche wird implizit oder explizit ein Breitensuchbaum aufgebaut.
- Die Tiefe eines Knotens u im Breitensuchbaum ist gleich der Länge des kürzesten Pfades von s zu u.
- Implementierung der Front durch eine Queue.

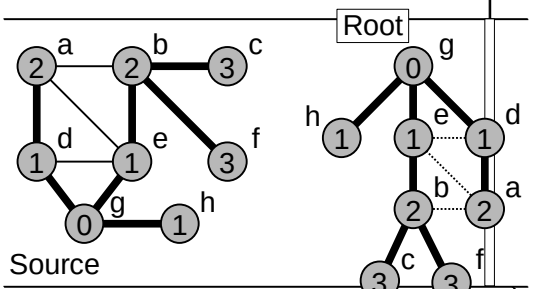
Breitensuche


```
BreadthFirstSearch( G )
for all v ∈ V do
    depth[v] = ∞;
for all v ∈ V do
    if (depth[v] = ∞)
        VisitBreadthFirst( G, v );
```


Breitensuche


```
VisitBreadthFirst( G, s )
depth[s] = 0;
Q = ∅; enqueue( Q, s );
while ( ! empty( Q ) )
    u = dequeue( Q );
    for all v ∈ Adj[u] do
        if ( depth[v] = ∞ )
            depth[v] = depth[u] + 1;
            enqueue( Q, v );
```

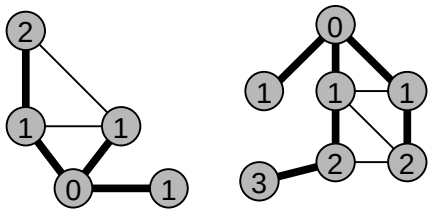

Breitensuchbaum



Source

Breitensuchwald

55



Breitensuche: Aufwand

56

```
BreadthFirstSearch( G )
for all v ∈ V do
   depth[v] = ∞;                    } O(V)
for all v ∈ V do
   if (depth[v] = ∞)
      VisitBreadthFirst( G, v );
```


Breitensuche: Aufwand

57

```
VisitBreadthFirst( G, s )
depth[s] = 0;
Q = ∅; enqueue( Q, s );
while ( ! empty( Q ) )
   u = dequeue( Q );
   for all v ∈ Adj[u] do
      if ( depth[v] = ∞ )
         depth[v] = depth[u] + 1;   } ?
         enqueue( Q, v );
```


Breitensuche: Aufwand

58

- Jeder Knoten wird genau einmal in die Queue geschoben.
- Jeder Knoten wird genau einmal aus der Queue genommen.
- Adj[u] wird genau dann abgearbeitet, wenn u aus der Queue genommen wird.

Breitensuche: Aufwand

59

- Der Aufwand zur Abarbeitung aller Adjazenzlisten ist also

$$\sum |Adj[u]| = O(E)$$

- Gesamtaufwand der Breitensuche:

$$O(V + E)$$

Breitensuche: Eigenschaften

60

- Die Abstände zwischen allen Knotenpaaren eines Graphen lassen sich in Zeit $O(V (V+E))$ berechnen.

Breitensuche: Eigenschaften

61

- Die Zahl der Zusammenhangskomponenten eines ungerichteten Graphen lässt sich in O(V+E) berechnen.
- Jeder ungerichtete Graph kann in Zeit O(V+E) auf Zusammenhang getestet werden.

Breitensuche: Eigenschaften

62

```
Components( G )
for all v ∈ V do
   comp[v] = 0;
components = 0;
for all v ∈ V do
   if (comp[v] = 0)
      ++components;
      VisitBreadthFirst´( G, v );
```


Breitensuche: Eigenschaften

63

```
VisitBreadthFirst´( G, s )
comp[s] = components;
Q = ∅; enqueue( Q, s );
while ( ! empty( Q ) )
   u = dequeue( Q );
   for all v ∈ Adj[u] do
      if ( comp[v] = 0 )
         comp[v] = components;
         enqueue( Q, v );
```


Breitensuche: Eigenschaften

64

- Ein ungerichteter Graph lässt sich in O(V) auf Kreisfreiheit testen.
- Erinnerung: Ein ungerichteter Graph G=(V,E) mit k Zusammenhangskomponenten enthält einen Kreis, genau dann wenn $|E| > |V| - k$.

Breitensuche: Eigenschaften

65

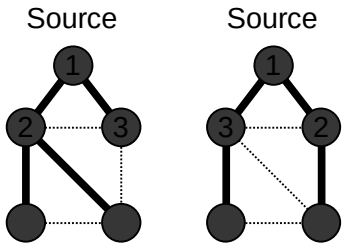
- Ist $|E| \geq |V|$ (Nachzählen und abbrechen sobald Zähler größer |V|), so enthält G einen Kreis.
- Ist $|E| < |V|$ so zählt man die Komponenten wie oben beschrieben und zwar nur noch mit Aufwand $O(V + E) = O(V)$

Breitensuche: Eigenschaften

66

- Breitensuche berechnet für jeden Knoten u die Länge dist[u] des kürzesten Pfades von der Quelle s zu u.
- Der Breitensuchbaum ist nicht eindeutig sondern hängt von der Reihenfolge der Knoten in den Adjazenzlisten ab.

Breitensuchbaum



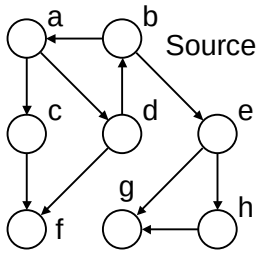
Tiefensuche

- Von einem gegeben ein Startknoten s („source“) ausgehend, werden nacheinander alle von s erreichbaren Knoten besucht.
- Vom zuletzt besuchten Knoten werden die zunächst die folgenden Knoten besucht („Depth-First“), Backtracking falls alle Nachbarn besucht.

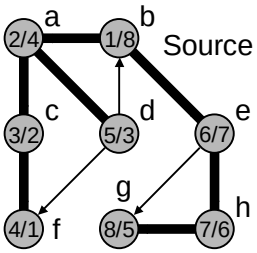
Tiefensuche

- In jedem Knoten w3rden die Eintritts-und Austrittszeitpunkte gespeichert. Dies entspricht einer Prefix bzw. Postfixordnung auf dem entstehenden Tiefensuchwald.

Tiefensuche



Tiefensuche



Tiefensuche


```
DepthFirstSearch( G )
for all v ∈ V do
    pre[v] = 0;
    post[v] = 0;
precntr = 0;
postcntr = 0;
for all v ∈ V do
    if (pre[v] = 0) DepthFirstVisit(v);
```

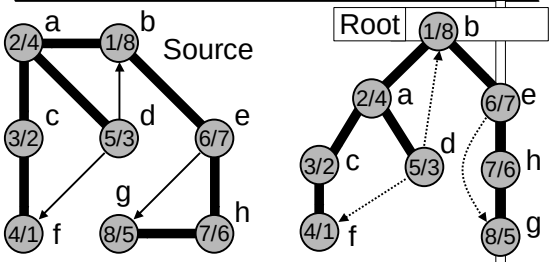
73

Tiefensuche

```
DepthFirstVisit( u )
++precntr;
pre[u] = precntr;
for all v ∈ Adj[u] do
    if ( pre[v] = 0 )
        DepthFirstVisit(v);
++postcntr;
post[u] = postctr;
```

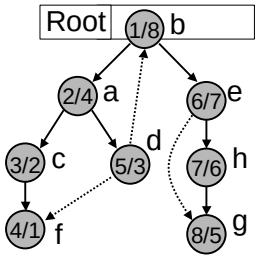

74

Tiefensuchbaum



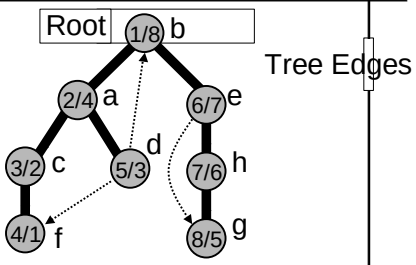
75

Tiefensuchbaum



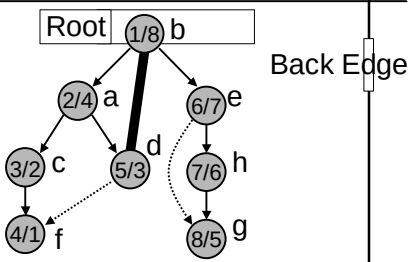
76

Tiefensuchbaum



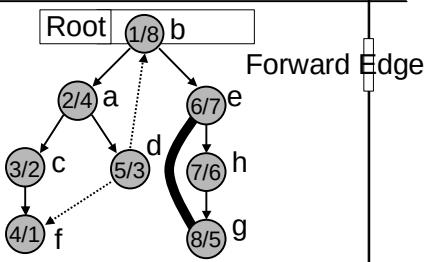
77

Tiefensuchbaum



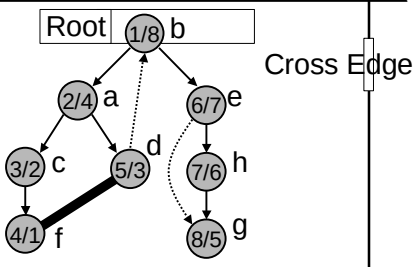
78

Tiefensuchbaum



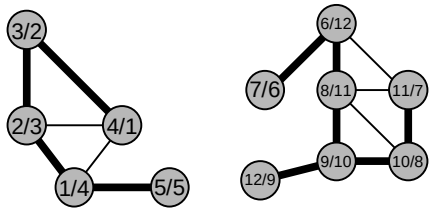
Tiefensuchbaum

79



Tiefensuchwald

80



Tiefensuche: Aufwand

81

```
DepthFirstSearch( G )
for all v ∈ V do
    pre[v] = 0;
    post[v] = 0;
precntr = 0;
postcntr = 0;
for all v ∈ V do
    DepthFirstVisit(v);
```

} O(V)

Tiefensuche: Aufwand

82

```
DepthFirstVisit( u )
++precntr;
pre[u] = precntr;
for all v ∈ Adj[u] do
    if ( pre[v] = 0 )
        DepthFirstVisit(v);
++postcntr;
post[u] = postctr;
```

} ?

Tiefensuche: Aufwand

83

- Jeder Knoten wird genau einmal besucht.
- Adj[u] wird genau dann abgearbeitet, wenn u besucht wird.

Tiefensuche: Aufwand

84

- Der Aufwand zur Abarbeitung aller Adjazenzlisten ist also

$$\sum |Adj[u]| = O(E)$$

- Gesamtaufwand der Tiefensuche:

$$O(V + E)$$

Topologisches Sortieren

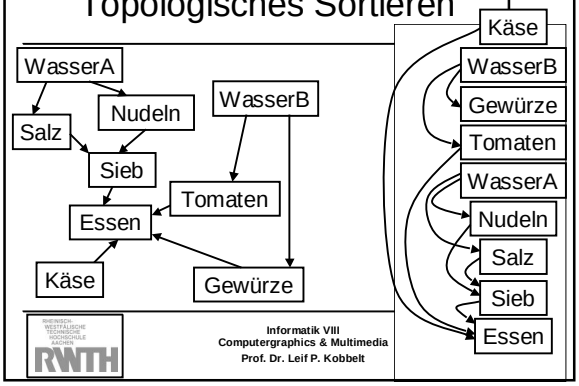
85

Eine vollständige Ordnung $(V, \leq)$ heißt eine topologische Sortierung des gerichteten, zyklenfreien Graphen G=(V,E), falls gilt:

$$(u,v) \in E \Rightarrow u \leq v$$

Topologisches Sortieren

86



Topologisches Sortieren

87

Sei $p:V \rightarrow N$ die Postfixnumerierung des Tiefensuchwalds von G. Dann ist die Ordnung der Knoten v nach absteigendem p[v] eine topologische Sortierung von G.

Topologisches Sortieren: Aufwand

88

Implementierung des topologischen Sortierens durch modifizierte Tiefensuche:
Beim Verlassen eines Knotens wird dieser am Kopf einer zunächst leeren Liste eingefügt.
Am Ende enthält die Liste die Knoten in sortierter Reihenfolge.

Topologisches Sortieren: Aufwand

89

Der Aufwand um einen zyklenfreien, gerichteten Graphen topologisch zu Sortieren ist

$$O(V + E)$$

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- (2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade

(2.6.3) Minimal Spannende Bäume

- (2.6.3.1) Generischer Algorithmus
- (2.6.3.2) Prims Algorithmus
- (2.6.3.3) Kruskals Algorithmus

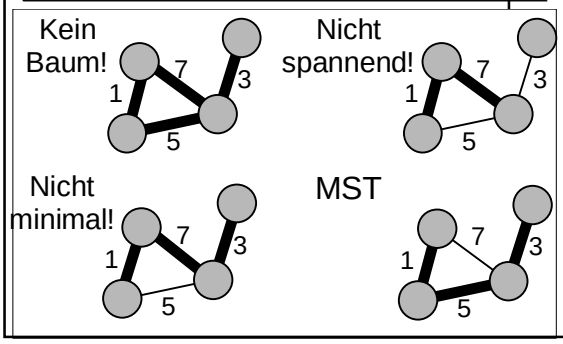
Minimal Spannende Bäume

Gegeben sei ein zusammenhängender, ungerichteter, gewichteter Graph $G=(V,E)$ mit Gewichtsfunktion $w:E\rightarrow IR$. Gesucht ist ein Spannbaum $T \subseteq E$ der

$$w(T) = \sum_{e \in T} w(e)$$

minimiert, der sogenannte MST.

Minimal Spannende Bäume



Generischer Algorithmus

Greedy-Strategie: Füge nach und nach Kanten zu einer anfangs leeren Kantenmenge A hinzu, und zwar unter Beibehaltung der folgenden Invariante:

A ist Teilmenge eines MST. (∗)

Generischer Algorithmus

Sei $A \subseteq E$ eine Menge von Kanten die die Invariante (∗) erfüllt. Eine Kante e heißt sicher für A, falls $A \cup \{e\}$ ebenfalls die Invariante (∗) erfüllt.

Generischer Algorithmus

7

```
GenericMST(G,w)
A = ∅;
while (A ist kein MST)
    suche eine sichere Kante e für A
    A = A ∪ { e };
return A;
```

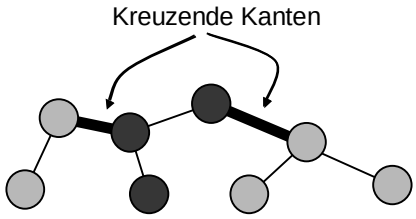

Generischer Algorithmus

8

- Ein Schnitt eines ungerichteten Graphen G=(V,E) ist ein Tupel (S, V−S) mit $S \subseteq V$.
- Eine Kante (u,v) kreuzt den Schnitt (S, V−S) falls einer ihrer Endpunkte in S und der andere Endpunkt in V−S liegt.

Generischer Algorithmus

9



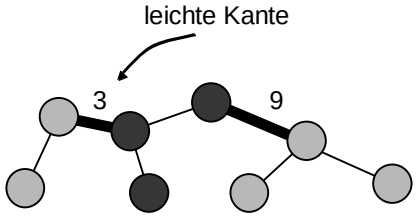
Generischer Algorithmus

10

- Ein Schnitt respektiert $A \subseteq E$, falls keine Kante aus A den Schnitt kreuzt.
- Eine kreuzende Kante heißt leicht, falls ihr Gewicht minimal unter allen kreuzenden Kanten ist.

Generischer Algorithmus

11



Generischer Algorithmus

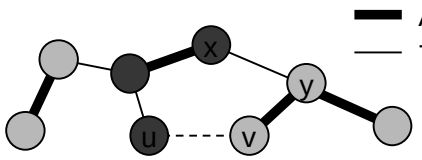
12

Satz:
$A \subseteq E$ erfülle die Invariante (∗).
(S, V-S) sei ein Schnitt der A respektiert und (u,v) eine leichte Kante, die (S,V-S) kreuzt. Dann ist (u,v) sicher für A.

Generischer Algorithmus

13

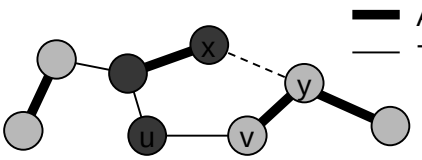
Sei T ein MST der A enthält, aber nicht (u,v).



Generischer Algorithmus

14

$T' = T - \{(x,y)\} \cup \{(u,v)\}$

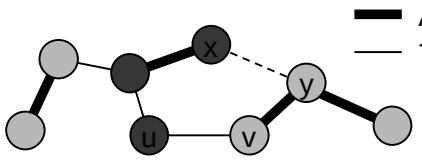


Generischer Algorithmus

15

T´ ist ein MST, denn

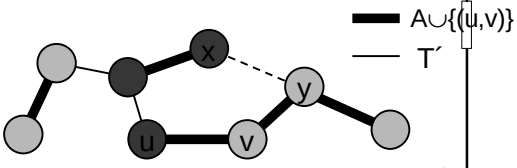
$w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$



Generischer Algorithmus

16

Wegen $A \subseteq T$ und $(x,y) \notin A$ folgt $A \cup \{(u,v)\} \subseteq T'$, also ist (u,v) sicher für A.



Prims Algorithmus

17

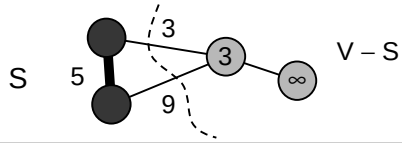
- Die Kanten in A bilden einen Baum. Jeder Knoten u erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt (S,V – S) ist gegeben durch

$$S = \{ u : (u,v) \in A \}$$

Prims Algorithmus

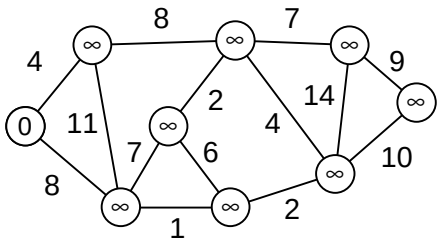
18

Knoten u ausserhalb von S erhalten einen Schlüssel, der das minimale Gewicht einer Kante angibt, die u mit S verbindet (= ∞ falls es eine solche Kante nicht gibt).



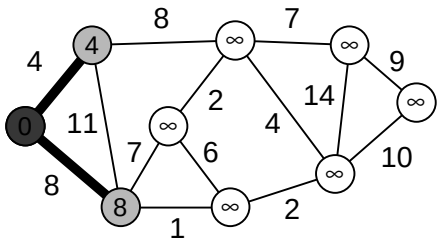
Prims Algorithmus

19



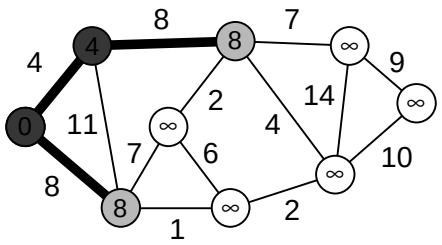
Prims Algorithmus

20



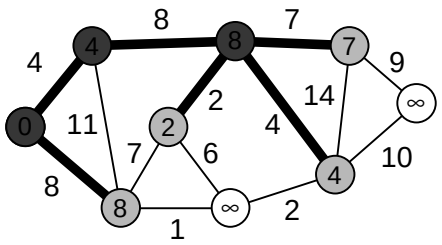
Prims Algorithmus

21



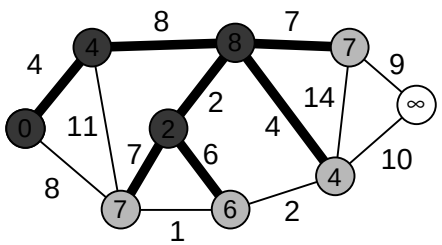
Prims Algorithmus

22



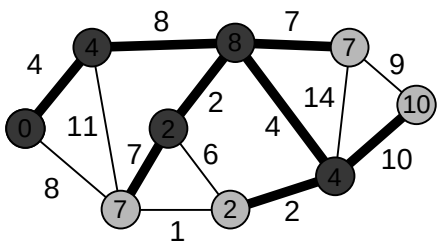
Prims Algorithmus

23



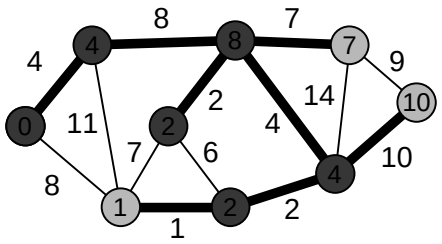
Prims Algorithmus

24



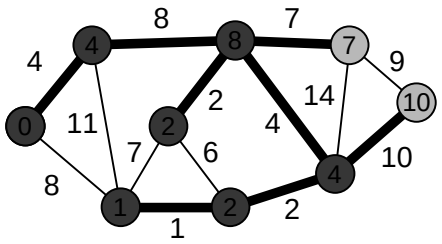
Prims Algorithmus

25



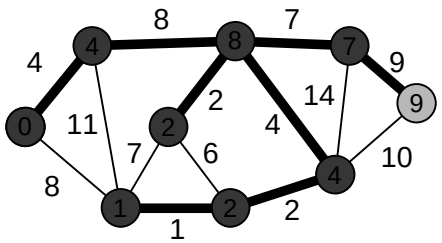
Prims Algorithmus

26



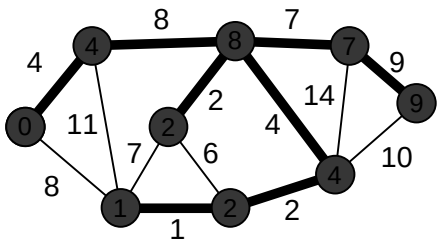
Prims Algorithmus

27



Prims Algorithmus

28



Prims Algorithmus

29

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```


Prims Algorithmus

30

Nach der Ausführung von Prim(G,w,r) ist der MST durch

$$A = \{ (u,p[u]) : u \in V - \{r\} \}$$

gegeben.

Prims Algorithmus

31

Implementierung von Prims Algorithmus z.B. mit binärem Heap.

Prims Algorithmus

32

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;        } O( V )
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```


Prims Algorithmus

33

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;              } O( V )
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Heap Aufbau!

Prims Algorithmus

34

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)                   }
    u = extract-minimum(Q);     }
    for each v ∈ Adj[u]          } V Durch-
        if (v ∈ Q and w(u,v) < key[v])  } läufe
            p[v] = u; key[v] = w(u,v);
```


Prims Algorithmus

35

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);     } O( log V )
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Entfernen eines Elements aus dem Heap.

Prims Algorithmus

36

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]          }
        if (v ∈ Q and w(u,v) < key[v])  }
            p[v] = u; key[v] = w(u,v);
```

O(E) Durch-läufe insge-samt!

Prims Algorithmus

37

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Heap muss aktualisiert werden!

O(log V)

Prims Algorithmus

38

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

O(V)

O(V log V + E log V)

Prims Algorithmus

39

- Implementiert man Prim() mittels eines binären Heaps, so ist der Aufwand

  $$O(E \log V)$$

- Implementiert man Prim() mittels eines Fibonacci-Heaps, so ist der Aufwand $O(E + V \log V)$

Kruskals Algorithmus

40

- Die Kanten in A bilden einen Wald. Jeder Knoten u erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt (S,V – S) ist gegeben durch

  $$S = \{ u : (u,v) \in A \}$$

Kruskals Algorithmus

41

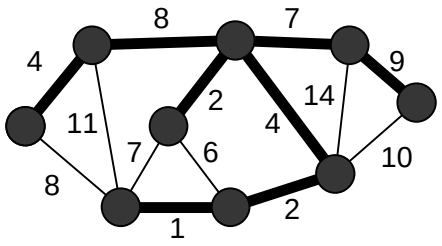
In jedem Schritt wird die Kante mit kleinstem Gewicht gesucht, die zwei Bäume des Waldes A verbindet. Diese Kante wird zur aktuellen Kantenmenge hinzugefügt.

Kruskals Algorithmus

42

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
    MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈ E
    if ( FindSet(u) ≠ FindSet(v) )
        A = A ∪ { (u,v) };
        Union(u,v);
return A;
```

Kruskals Algorithmus



Kruskals Algorithmus


```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈ E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

Implementierung?

Union-Find-Strukturen

Exkurs: Union-Find-Strukturen

- Datenstruktur zur Darstellung von **disjunkten** Mengen.
- Jede Menge A wird durch einen Repräsentant $x \in A$ identifiziert.
- Operationen: MakeSet(), Union(), FindSet()

Union-Find-Strukturen

MakeSet(x)

- Erzeugt eine neue Menge S_x deren einziges Element x ist.
- Der Repräsentant von MakeSet(x) ist x.
- Disjunktheit ⇒
  - x darf nicht schon in einer anderen Menge enthalten sein!

Union-Find-Strukturen

z = UnionSet(x,y)

- Erzeugt die Vereinigung $S_x \cup S_y$ der Mengen S_x und S_y (x und y müssen keine Repräsentanten sein!)
- z ist der Repräsentant der Vereinigung $S_x \cup S_y$
- Disjunktheit ⇒
  - S_x und S_y müssen disjunkt sein!
  - S_x und S_y werden zerstört!

Union-Find-Strukturen

z = FindSet(x)

- Liefert den Repräsentant der Menge die x enthält

Union-Find-Strukturen

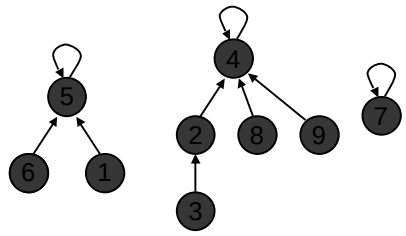
49

Idee:

- Jede Menge A wird durch einen Baum dargestellt.
- Die Knoten und Blätter des Baumes enthalten die Elemente von A.
- Die Wurzel des Baumes enthält den Repräsentant von A.
- Implementierung: Jeder Knoten enthält einen Zeiger „parent“ auf seinen Vater!

Union-Find-Strukturen

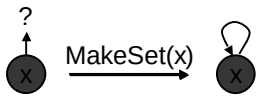
50



Union-Find-Strukturen

51

MakeSet(x)
x.parent = x;

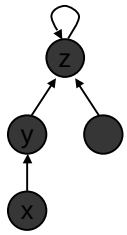


Union-Find-Strukturen

52

FindSet(x)
if (x = x.parent)
  return x;
return FindSet(x.parent);

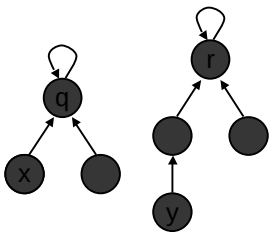
z = FindSet(x)



Union-Find-Strukturen

53

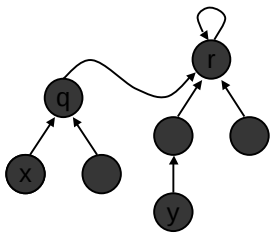
Union(x,y)
q = FindSet(x);
r = FindSet(y));
q.parent = r;
return r;



Union-Find-Strukturen

54

Union(x,y)
q = FindSet(x);
r = FindSet(y));
q.parent = r;
return r;



Union-Find-Strukturen

55

Laufzeitverbesserungen:

1. Höhenbalancierung
2. Pfadkompression

Union-Find-Strukturen

56

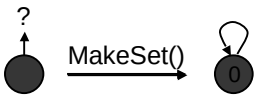
Höhenbalancierung

- Speichere in jedem Element x die Höhe x.height des darunterhängenden Baumes.
- Hänge bei Vereinigung stets die niedrigeren Bäume an die höheren und aktualisiere ggfs die Wurzel.

Union-Find-Strukturen

57

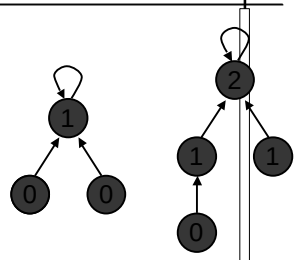
```
MakeSet(x)
x.parent = x;
x.height = 0;
```



Union-Find-Strukturen

58

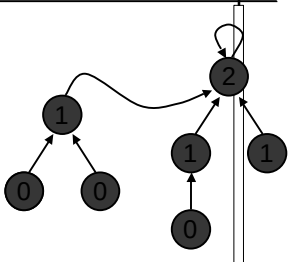
```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

59

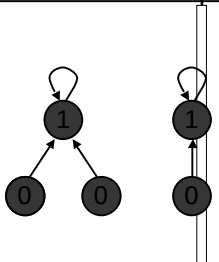
```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

60

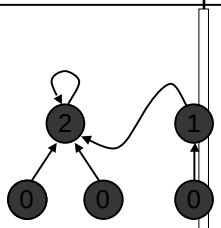
```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

61

```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

62

Pfadkompression:

Hänge nach jeder Find(x) Operation die Elemente auf dem Pfad zu x an die Wurzel.

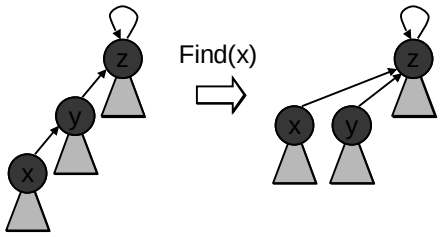
Union-Find-Strukturen

63

```
FindSet(x)
if (x ≠ x.parent)
   x.parent = FindSet(x.parent);
return x.parent;
```


Union-Find-Strukturen

64



Union-Find-Strukturen

65

Aufwand (ohne Beweis)
Sei n die Zahl der MakeSet Operationen und m die Gesamtzahl aller MakeSet, Union und FindSet Operationen. Dann lässt sich der Aufwand bei Verwendung von Höhen-balancierung und Pfadkompression durch

$$O(m * a(n))$$

abschätzen, wobei in allen praktischen Fällen $a(n) \leq 5$ ist (Details: Cormen et al.)

Kruskals Algorithmus

66

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈ E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

Implementierung durch Union-Find-Sturkturen!

Kruskals Algorithmus


```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈ E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

O(E log E)

Kruskals Algorithmus


```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈ E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

V MakeSet Operationen

O(E) FindSet und Union Operationen

Kruskals Algorithmus

Aufwand:

- Sortieren: O(E log E) = O(E log V)
- Union-Find:
  - V MakeSe
  - O(E) Fin
    Operation
  - ⇒ O((V + E)*a(V)) = O(E log V)
- Gesam

$E < V^2$
⇒
log E = O(log V)

E > V – 1 und a(V) = O(log V)

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- (2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- (2.6.3) Minimal **auf**spannende Bäume
- (2.6.4) Kürzeste Pfade

Minimal Spannende Bäume

- (2.6.3.1) Generischer Algorithmus
- (2.6.3.2) Prim's Algorithmus
- (2.6.3.3) Kruskal's Algorithmus

Minimal Spannende Bäume

Gegeben sei ein *zusammenhängender, ungerichteter, gewichteter* Graph
$G = (V,E)$ mit Gewichtsfunktion $w : E \rightarrow IR$

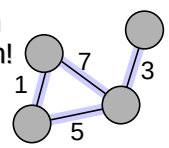
Gesucht ist ein Spannbaum $T \subseteq E$ der

$$w(T) = \sum_{e \in T} w(e)$$

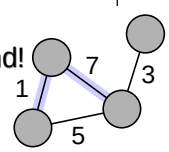
minimiert, der sogenannte MST.

Minimal Spannende Bäume

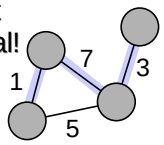
Kein Baum!



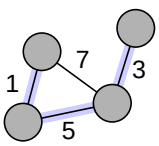
Nicht spannend!



Nicht minimal!



MST



Generischer Algorithmus

Greedy-Strategie: Füge nach und nach Kanten zu einer anfangs leeren Kantenmenge A hinzu, und zwar unter Beibehaltung der folgenden Invariante:

A ist Teilmenge eines MST (*)

Generischer Algorithmus

- Terminierung: Der Spannbaum für die |V| Knoten hat genau |V|-1 Kanten
- Korrektheit: Die leere Menge ist Teilmenge jedes MST
- Korrektheit: Am Ende ist A ein MST

Generischer Algorithmus

Sei $A \subseteq E$ eine Menge von Kanten, die die Invariante (∗) erfüllt. Eine Kante e heißt ***sicher*** für A, falls $A \cup \{e\}$ ebenfalls die Invariante (∗) erfüllt.

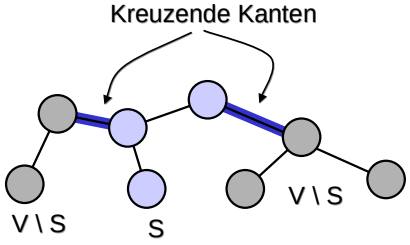
Generischer Algorithmus

GenericMST(G,w)
A = ∅;
while (A ist kein MST)
  suche eine *sichere* Kante e für A
  A = A ∪ { e };
return A;

Generischer Algorithmus

- Ein *Schnitt* eines ungerichteten Graphen G=(V,E) ist ein Tupel (S, V \ S) mit $S \subseteq V$.
- Eine Kante (u,v) *kreuzt* den Schnitt (S, V \ S), falls einer der Endpunkte in S und der andere Endpunkt in V \ S liegt.

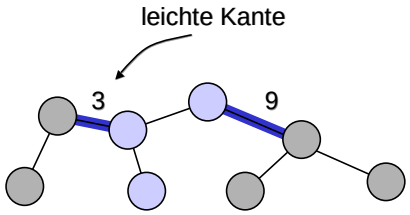
Generischer Algorithmus



Generischer Algorithmus

- Ein Schnitt *respektiert* $A \subseteq E$, falls keine Kante aus A den Schnitt kreuzt.
- Eine kreuzende Kante heißt *leicht*, falls ihr Gewicht minimal unter allen kreuzenden Kanten ist.

Generischer Algorithmus



Generischer Algorithmus

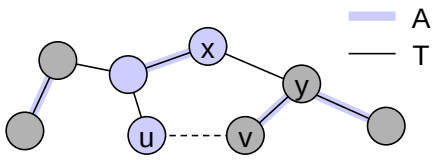
Satz:

A ⊆ E erfülle die Invariante (∗) und (S, V \ S) sei ein Schnitt, der A respektiert und (u,v) eine leichte Kante, die (S,V \ S) kreuzt.

Dann ist (u,v) sicher für A.

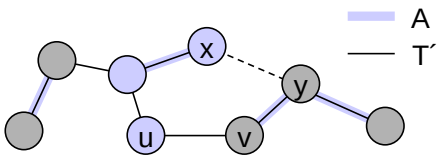
Generischer Algorithmus

Sei T ein MST der A enthält, aber nicht (u,v).



Generischer Algorithmus

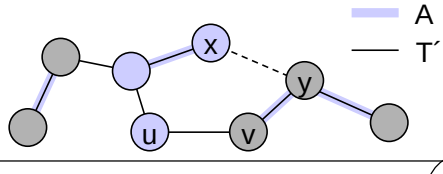
$T' = T - \{(x,y)\} \cup \{(u,v)\}$



Generischer Algorithmus

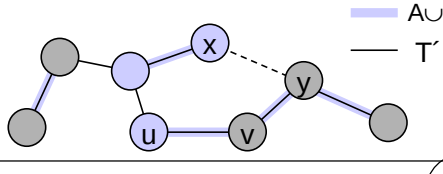
T´ ist ein MST, denn

$w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$



Generischer Algorithmus

Wegen A ⊆ T und (x,y) ∉ A folgt A∪{(u,v)} ⊆ T´, also ist (u,v) sicher für A.



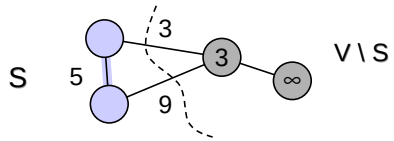
Prims Algorithmus

- Die Kanten in A bilden einen Baum. Jeder Knoten u erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt (S,V \ S) ist gegeben durch

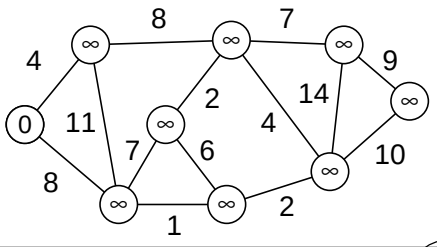
$$S = \{ u : (u,v) \in A \}$$

Prims Algorithmus

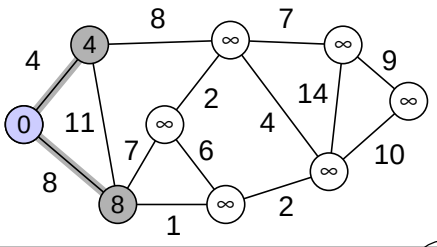
Knoten v ausserhalb von S erhalten einen Schlüssel, der das minimale Gewicht einer Kante angibt, die v mit S verbindet (= ∞ falls es eine solche Kante nicht gibt).



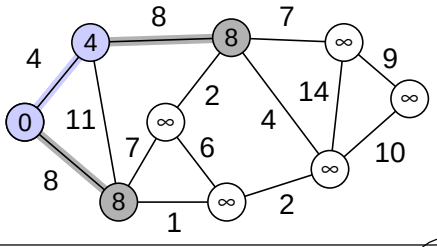
Prims Algorithmus



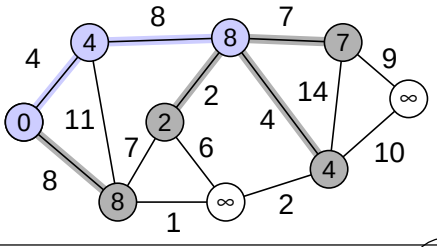
Prims Algorithmus



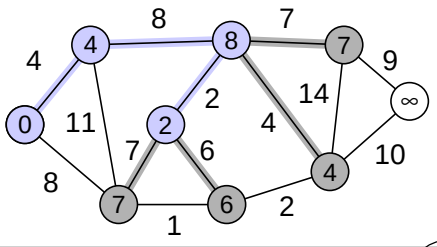
Prims Algorithmus



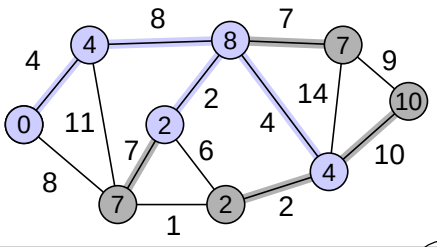
Prims Algorithmus



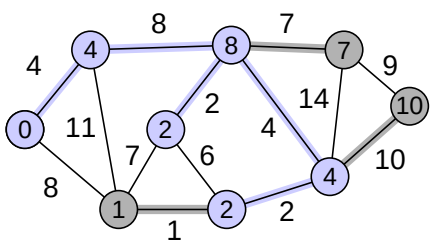
Prims Algorithmus



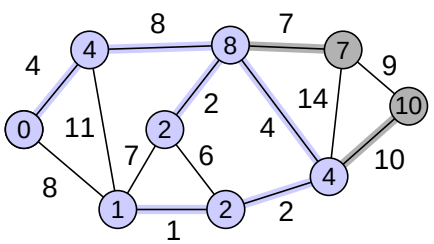
Prims Algorithmus



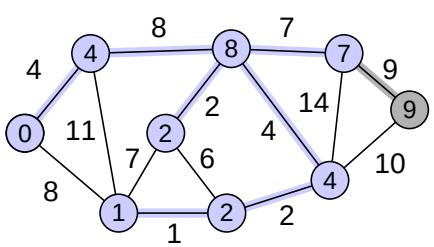
Prims Algorithmus



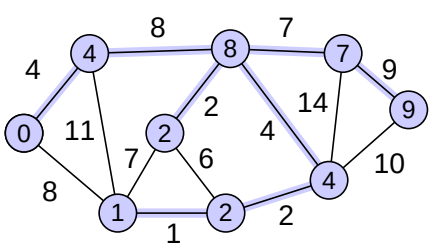
Prims Algorithmus



Prims Algorithmus



Prims Algorithmus



Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```


Prims Algorithmus

Nach der Ausführung von Prim(G,w,r) ist der MST durch

$$A = \{ (u,p[u]) : u \in V \setminus \{r\} \}$$

gegeben.

Aufwandsabschätzung

- Zugriff auf minimales Element
- Prioritätsschlange (Heap)
- Gesamtaufwand O (E log(V))

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V                          } O( V )
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```


Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;                      } O( V )
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Heap Aufbau!

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)                           } V Durch-
    u = extract-minimum(Q);             } läufe
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Entfernen eines Elements aus dem Heap.

} O(log V)

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

} O(E) Durch-läufe insge-samt!

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

Heap muss aktualisiert werden!

} O(log V)

Prims Algorithmus

```
Prim(G,w,r)
for each u ∈ V
    key[u] = ∞; p[u] = NULL;
key[r] = 0; Q = V;
while (Q ≠ ∅)
    u = extract-minimum(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u; key[v] = w(u,v);
```

} O(V)

} O(V log V + E log V)

Aufwandsabschätzung

$$O(V) + O(E \log(V) + V \log(V))$$

$$= O((E+V) \log(V))$$

$$= O(E \log(V))$$

Kruskal‘s Algorithmus

- Die Kanten in A bilden einen *Wald*. Jeder Knoten u erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt (S,V \ S) ist gegeben durch

$$S = \{ u : (u,v) \in A \}$$

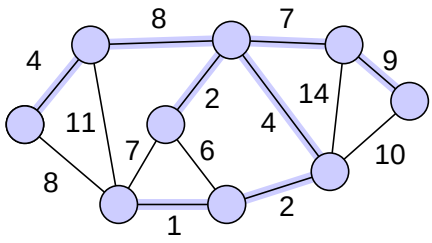
Kruskals Algorithmus

In jedem Schritt wird die Kante mit kleinstem Gewicht gesucht, die zwei Bäume des Waldes A verbindet. Diese Kante wird zur aktuellen Kantenmenge hinzugefügt.

Kruskals Algorithmus

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```


Kruskals Algorithmus



Kruskals Algorithmus

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

Implementierung?

Union-Find-Strukturen

Exkurs: Union-Find-Strukturen

- Datenstruktur zur Darstellung von **disjunkten** Mengen.
- Jede Menge A wird durch einen Repräsentant x ∈ A identifiziert.
- Operationen: MakeSet(), Union(), FindSet()

Union-Find-Strukturen

MakeSet(x)

- Erzeugt eine neue Menge S_x deren einziges Element x ist.
- Der Repräsentant von MakeSet(x) ist x.
- ***Disjunktheit*** ⇒
  x darf nicht schon in einer anderen Menge enthalten sein!

Union-Find-Strukturen

z = UnionSet(x,y)

- Erzeugt die Vereinigung $S_x \cup S_y$ der Mengen S_x und S_y (x und y müssen *keine* Repräsentanten sein!)
- z ist der Repräsentant der Vereinigung $S_x \cup S_y$
- ***Disjunktheit*** $\Rightarrow$
  S_x und S_y müssen vorher disjunkt sein!
  S_x und S_y werden zerstört!

Union-Find-Strukturen

z = FindSet(x)

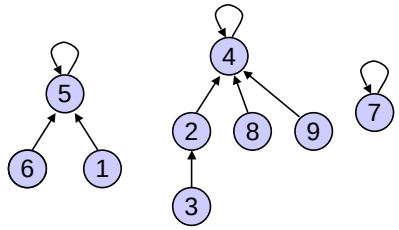
- Liefert den Repräsentanten, der Menge die x enthält

Union-Find-Strukturen

Idee:

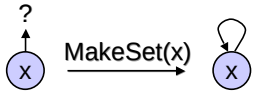
- Jede Menge A wird durch einen *Baum* dargestellt.
- Die Knoten und Blätter des Baumes enthalten die Elemente von A.
- Die Wurzel des Baumes enthält den *Repräsentant* von A.
- Implementierung: Jeder Knoten enthält einen Zeiger „parent“ auf seinen Vater!

Union-Find-Strukturen



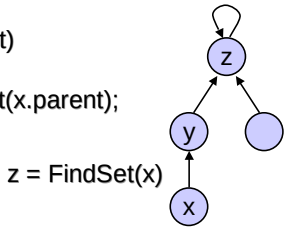
Union-Find-Strukturen

MakeSet(x)
x.parent = x;



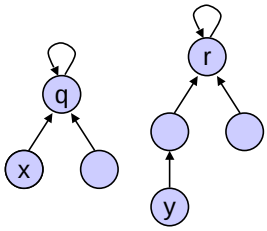
Union-Find-Strukturen

FindSet(x)
if (x = x.parent)
  return x;
return FindSet(x.parent);



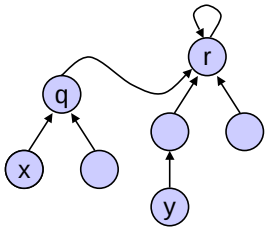
Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r  = FindSet(y));
q.parent = r;
return r;
```



Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r  = FindSet(y));
q.parent = r;
return r;
```



Union-Find-Strukturen

Laufzeitverbesserungen:

1. Höhenbalancierung
2. Pfadkompression

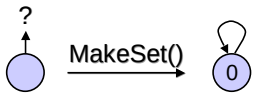
Union-Find-Strukturen

Höhenbalancierung

- Speichere in jedem Element x die Höhe x.height des darunterhängenden Baumes.
- Hänge bei Vereinigung stets die niedrigeren Bäume an die höheren und aktualisiere ggfs die Wurzel.

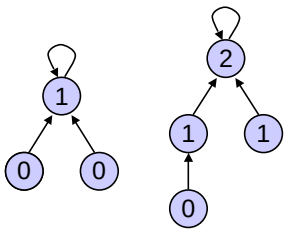
Union-Find-Strukturen

```
MakeSet(x)
x.parent = x;
x.height = 0;
```



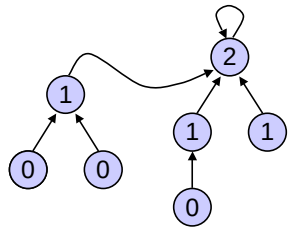
Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



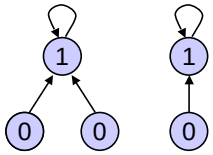
Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



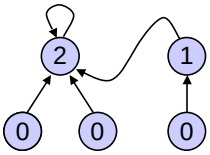
Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

```
Union(x,y)
q = FindSet(x);
r = FindSet(y);
if (q.height > r.height)
  r.parent = q;
  return q;
else
  q.parent = r;
  if (q.height == r.height)
     ++r.height;
  return r;
```



Union-Find-Strukturen

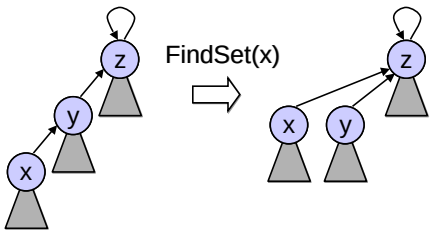
Pfadkompression:

Hänge nach jeder FindSet(x) Operation die Elemente auf dem Pfad zu x an die Wurzel.

Union-Find-Strukturen

```
FindSet(x)
if (x ≠ x.parent)
   x.parent = FindSet(x.parent);
return x.parent;
```


Union-Find-Strukturen



Union-Find-Strukturen

Aufwand (ohne Beweis)
Sei n die Zahl der MakeSet Operationen und m die Gesamtzahl aller MakeSet, Union und FindSet Operationen. Dann lässt sich der Aufwand bei Verwendung von Höhen-balancierung und Pfadkompression durch

$$O(m * a(n))$$

abschätzen, wobei in allen praktischen Fällen $a(n) \leq 5$ ist (Details: Cormen et al.)

Kruskals Algorithmus

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

Implementierung durch Union-Find-Sturkturen!

Kruskals Algorithmus

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

} O(E log E)

Kruskals Algorithmus

```
Kruskal(G, w)
A = ∅;
for all v ∈ V
   MakeSet(v);
sort E into nondecreasing order
for each (u,v)∈E
   if ( FindSet(u) ≠ FindSet(v) )
      A = A ∪ { (u,v) };
      Union(u,v);
return A;
```

} V MakeSet Operationen

} O(E) FindSet und Union Operationen

Kruskals Algorithmus

Aufwand:

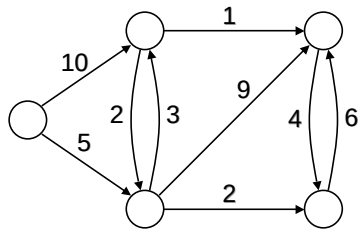
- Sortieren: O(E log E) = O(E log V)
- Union-Find:
  - V MakeSet Operationen
  - O(E) FindSet und Union Operationen

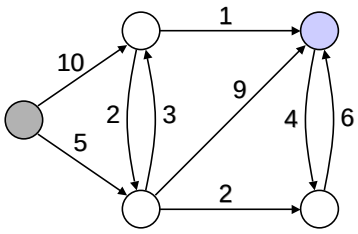
  $\Rightarrow$ O((V + E)*a(V)) = O(E log V)
- Gesamtaufwand:

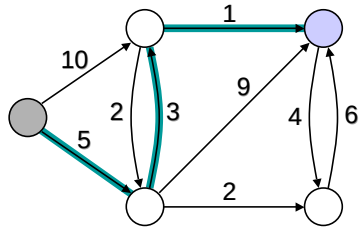
$$O(E \log V)$$

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- ✓(2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade







Kürzeste Pfade

- Definitionen
- Dijkstra Algorithmus
- Floyd-Warshall Algorithmus
- Transitive Hülle

Kürzeste Pfade

Gegeben sei ein *gerichteter, gewichteter* Graph G=(V,E) mit Gewichtsfunktion $w : E \rightarrow IR$. Das Gewicht w(p) eines Pfades

$$p = v_0, v_1, \ldots, v_k$$

ist definiert als

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Kürzeste Pfade

Kürzeste Pfade

Kürzeste Pfade

Kürzeste Pfade

Das *kürzeste Pfad Gewicht* $\delta(u,v)$ von Knoten u zu Knoten v ist definiert als

$$\delta(u,v) = \min \{ w(p) : p \text{ ist ein Pfad von u nach v} \}$$

wobei wir $\min \varnothing = \infty$ setzen.

Kürzeste Pfade

Ein *kürzester Pfad* von Knoten u zu Knoten v ist ein beliebiger Pfad p mit

$w(p) = \delta(u,v)$

Eigenschaften

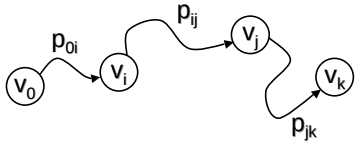
Lemma:

Teilpfade von kürzesten Pfaden sind ebenfalls kürzeste Pfade.

Ist $p = v_0,...,v_k$ ein kürzester Pfad von v_0 nach v_k, so ist für $0 \le i < j \le k$ der Sub-Pfad $p_{ij} = v_i,...,v_j$ ein kürzester Pfad von v_i nach v_j.

Eigenschaften

Beweis:

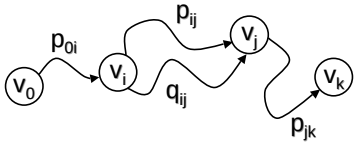


Es ist $p = p_{0i},p_{ij},p_{jk}$ und damit

$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$

Eigenschaften

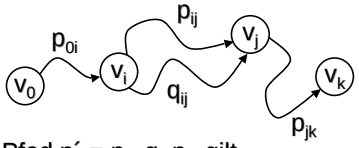
Beweis:



Annahme: Es gibt einen Pfad q_{ij} von v_i nach v_j mit $w(q_{ij}) < w(p_{ij})$.

Eigenschaften

Beweis:



Für den Pfad $p' = p_{0i},q_{ij},p_{jk}$ gilt

$w(p') = w(p_{0i}) + w(q_{ij}) + w(p_{jk})$
$< w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
$= w(p)$

Kürzeste Pfade

Single-source shortest-paths Problem

Gegeben sei ein Graph G=(V,E) und ein Startknoten $s \in V$. Bestimme zu jedem Knoten $u \in V$ einen kürzesten Pfad von s nach u.

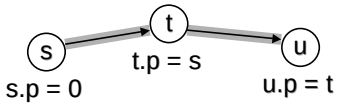
Kürzeste Pfade

All-pairs shortest-paths Problem

Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar u∈V und v∈V einen kürzesten Pfad von u nach v.

Darstellung

Vorgänger:
Ein Pfad von s nach v wird dargestellt, indem jeder Knoten des Pfades seinen Vorgänger in einem Attribut p speichert.



Darstellung

Oberschranke:
In jedem Knoten v des Graphen wird ein Attribut d gespeichert, das eine *obere Schranke* für das Gewicht des kürzesten Pfades von s nach v darstellt, d.h. es gilt immer die Invariante

$$\delta(s,v) \leq v.d$$

Darstellung

Oberschranke:
Ein Knoten heißt final, falls die Oberschranke *minimal* ist, d.h. falls

$$\delta(s,v) = v.d$$

Darstellung

```
InitializeSingleSource(G, s)
    for each v∈V do
        v.d = ∞;
        v.p = 0;
    s.d = 0;
```


Darstellung

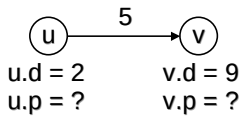
Relaxation einer Kante (u,v)
Die Oberschranke v.d kann verbessert werden, wenn eine Kante von u nach v existiert, so dass

$$u.d + w(u,v) < v.d$$

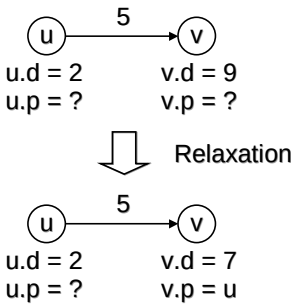
ist. In diesem Fall setzt man

$$v.d = u.d + w(u,v)$$
$$v.p = u$$

Relaxation



Relaxation

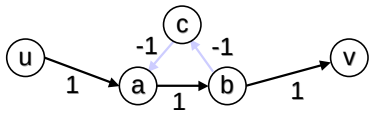


Implementierung

```
Relax(u,v)
    if ( v.d > u.d + w(u,v) )
        v.d = u.d + w(u,v);
        v.p = u;
```


Negative Gewichte

- Prinzipiell sind gewichtete Graphen mit negativen Gewichten zulässig
- Dadurch kann aber die Frage nach kürzesten Pfaden unsinnig werden



Dijkstras Algorithmus

Dijkstra's Algorithmus löst das single-source shortest-path Problem für den Fall, dass alle Kantengewichte positiv sind, d.h.

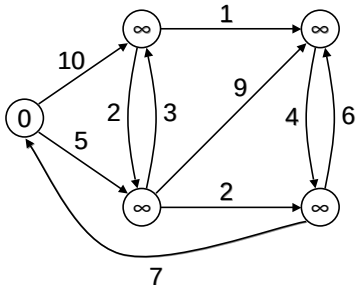
$$w(u,v) \geq 0$$

für alle Kanten $(u,v) \in E$.

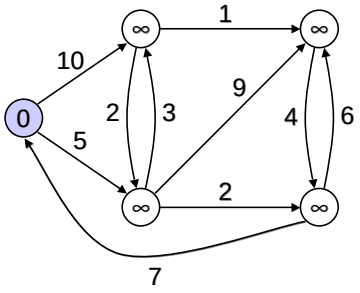
Dijkstras Algorithmus

1. Sei S eine anfangs leere Menge.
2. Wähle einen Knoten $v \in V \setminus S$ mit minimaler Oberschranke v.d.
3. Füge v in S ein und relaxiere alle von v ausgehenden Kanten
4. Gehe zu Schritt 2, falls $V \setminus S \neq \emptyset$.

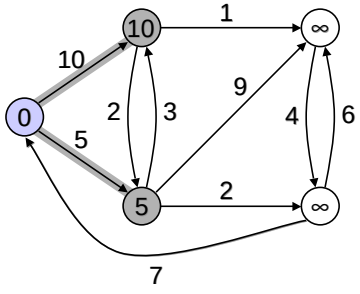
Dijkstras Algorithmus



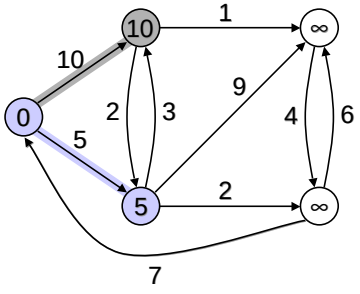
Dijkstras Algorithmus



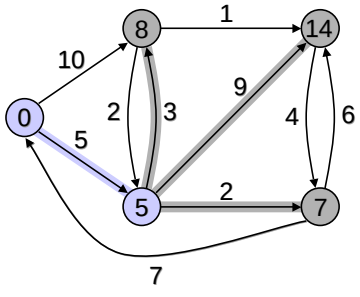
Dijkstras Algorithmus



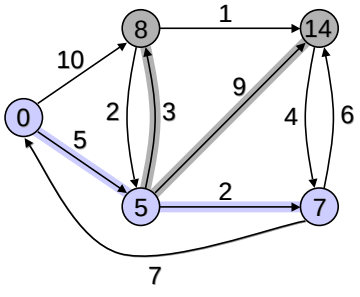
Dijkstras Algorithmus



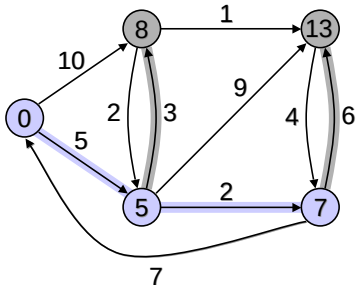
Dijkstras Algorithmus



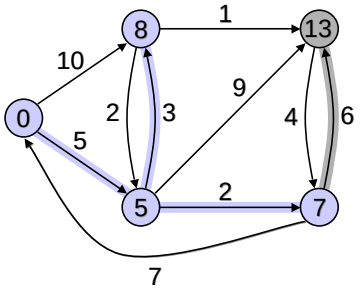
Dijkstras Algorithmus



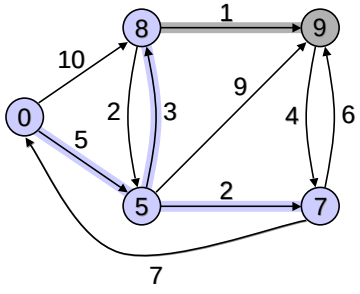
Dijkstras Algorithmus



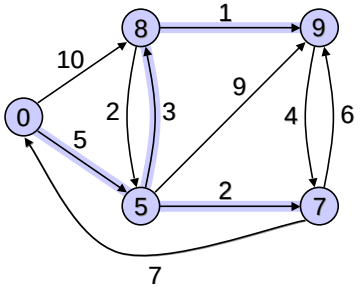
Dijkstras Algorithmus



Dijkstras Algorithmus



Dijkstras Algorithmus



Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
    u = ExtractMinimum(Q);
    S = S ∪ {u};
    for each v ∈ Adj[u]
        Relax(u,v);
```


Dijkstras Algorithmus

Korrektheit von Dijkstra's Algorithmus

Sei G ein gerichteter, gewichteter Graph mit nichtnegativer Gewichtsfunktion und $s \in V$ ein Startknoten. Dann gilt: Dijkstra's Algorithmus angewandt auf G terminiert mit $v.d = \delta(s,v)$ für alle Knoten $v \in V$.

Dijkstras Algorithmus

Beweis:

Schleifeninvariante:
Es gilt v.d = δ(s,v) für alle Knoten v∈S, d.h. alle Knoten in S sind *final*.

Dijkstras Algorithmus

Initialisierung:

Vor Ausführung der Schleife ist

$S = \emptyset$

die Schleifeninvariante gilt damit trivialerweise.

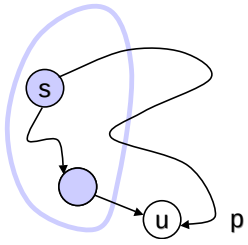
Dijkstras Algorithmus

Schleife:

Behauptung: Es gilt v.d = δ(s,v) für jeden Knoten der zu S hinzugefügt wird.

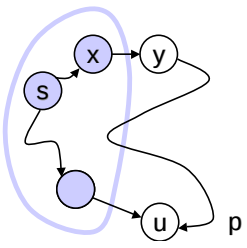
Beweis durch Widerspruch: Sei u der erste Knoten für den u.d ≠ δ(s,u) ist ...

Dijkstras Algorithmus



u ist von s erreichbar, es gibt also einen kürzesten Pfad p von s nach u

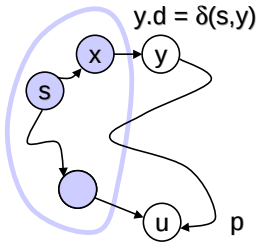
Dijkstras Algorithmus



Sei y der erste Knoten auf p der nicht in S liegt und sei x sein Vorgänger

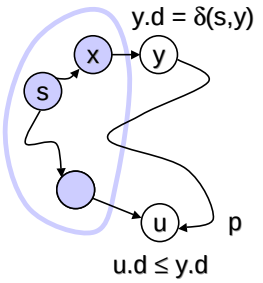
Dijkstras Algorithmus

y.d = δ(s,y)



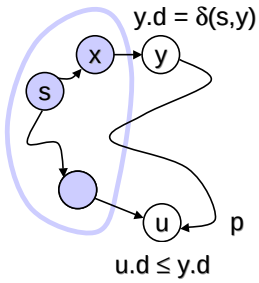
Teilpfade von kürzesten Pfaden sind kürzeste Pfade, (x,y) wurde bereits relaxiert, also ist y.d = δ(s,y).

Dijkstras Algorithmus



u wird von Dijkstras Algorithmus vor y ausgewählt, also ist u.d ≤ y.d

Dijkstras Algorithmus



Es gilt

$y.d = \delta(s,y) \leq \delta(s,u) \leq u.d$

und

$u.d \leq y.d$

also

$y.d = \delta(s,y) = \delta(s,u) = u.d$

und insbesondere

$u.d = \delta(s,u)$

Dijkstras Algorithmus

Terminierung:

Terminierung wenn $Q = \emptyset$, also wegen $Q = V \setminus S$, wenn $S = V$.

Dijkstras Algorithmus

Aufwand:

Implementierung der Priority Queue durch einen binären Heap.

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```

Initialisierung: O(V)

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```

Aufwand für binären Heap

O(V)

O(log V)

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```

Schleife wird V mal durchlaufen. Insgesamt also O(V log V)

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```

Jede Kante wird höchstens einmal relaxiert. Insgesamt also O(E log V)

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```

O(V)

O((E+V) log V)

Dijkstras Algorithmus

Fazit:

Falls G zusammenhängend ist, so hat Dijkstra's Algorithmus einen Aufwand von

O(E log V)

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- ✓(2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade

Kürzeste Pfade

- Definitionen
- Dijkstra Algorithmus
- Floyd-Warshall Algorithmus
- Transitive Hülle

Kürzeste Pfade

Gegeben sei ein *gerichteter, gewichteter* Graph G=(V,E) mit Gewichtsfunktion $w : E \rightarrow IR$. Das Gewicht w(p) eines Pfades

$$p = v_0, v_1, \ldots, v_k$$

ist definiert als

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Kürzeste Pfade

Das *kürzeste Pfad Gewicht* $\delta(u,v)$ von Knoten u zu Knoten v ist definiert als

$$\delta(u,v) = \min \{ w(p) : p \text{ ist ein Pfad von } u \text{ nach } v \}$$

wobei wir $\min \varnothing = \infty$ setzen.

Kürzeste Pfade

Ein *kürzester Pfad* von Knoten u zu Knoten v ist ein beliebiger Pfad p mit

$$w(p) = \delta(u,v)$$

Eigenschaften

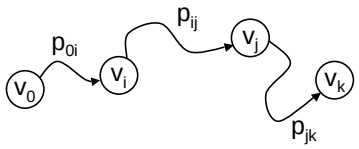
Lemma:

Teilpfade von kürzesten Pfaden sind ebenfalls kürzeste Pfade.

Ist $p = v_0, \ldots, v_k$ ein kürzester Pfad von v_0 nach v_k, so ist für $0 \le i < j \le k$ der Sub-Pfad $p_{ij} = v_i, \ldots, v_j$ ein kürzester Pfad von v_i nach v_j.

Eigenschaften

Beweis:

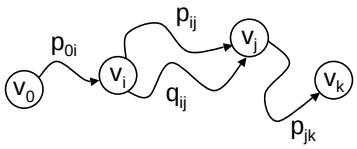


Es ist $p = p_{0i},p_{ij},p_{jk}$ und damit

$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$

Eigenschaften

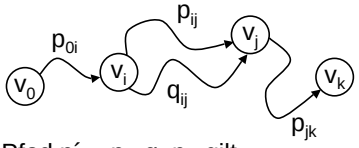
Beweis:



Annahme: Es gibt einen Pfad q_{ij} von v_i nach v_j mit $w(q_{ij}) < w(p_{ij})$.

Eigenschaften

Beweis:



Für den Pfad $p' = p_{0i},q_{ij},p_{jk}$ gilt

$w(p') = w(p_{0i}) + w(q_{ij}) + w(p_{jk})$

$< w(p_{0i}) + w(p_{ij}) + w(p_{jk})$

$= w(p)$

Kürzeste Pfade

Single-source shortest-paths Problem

Gegeben sei ein Graph G=(V,E) und ein Startknoten $s \in V$. Bestimme zu jedem Knoten $u \in V$ einen kürzesten Pfad von s nach u.

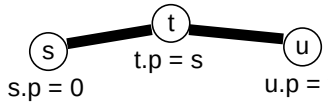
Kürzeste Pfade

All-pairs shortest-paths Problem

Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar $u \in V$ und $v \in V$ einen kürzesten Pfad von u nach v.

Darstellung

Vorgänger:
Ein Pfad von s nach v wird dargestellt, indem jeder Knoten des Pfades seinen Vorgänger in einem Attribut p speichert.



Darstellung

Oberschranke:
In jedem Knoten v des Graphen wird ein Attribut d gespeichert, das eine *obere Schranke* für das Gewicht des kürzesten Pfades von s nach v darstellt, d.h. es gilt immer die Invariante

$$\delta(s,v) \leq v.d$$

Darstellung

Oberschranke:
Ein Knoten heißt final, falls die Oberschranke *minimal* ist, d.h. falls

$$\delta(s,v) = v.d$$

Darstellung

```
InitializeSingleSource(G, s)
    for each v∈V do
        v.d = ∞;
        v.p = 0;
    s.d = 0;
```


Darstellung

Relaxation einer Kante (u,v)
Die Oberschranke v.d kann verbessert werden, wenn eine Kante von u nach v existiert, so dass

$$u.d + w(u,v) < v.d$$

ist. In diesem Fall setzt man

$$v.d = u.d + w(u,v)$$
$$v.p = u$$

Dijkstras Algorithmus

Dijkstra's Algorithmus löst das single-source shortest-path Problem für den Fall, dass alle Kantengewichte positiv sind, d.h.

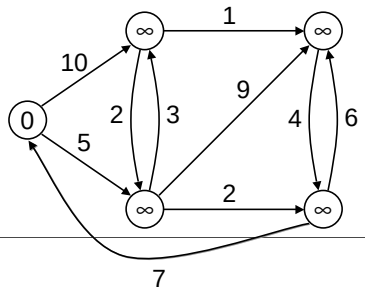
$$w(u,v) \geq 0$$

für alle Kanten $(u,v) \in E$.

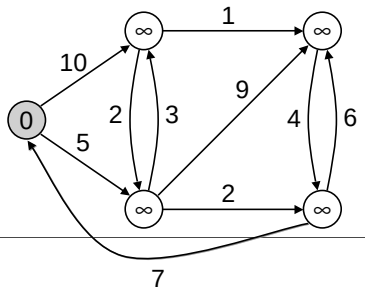
Dijkstras Algorithmus

1. Sei S eine anfangs leere Menge.
2. Wähle einen Knoten $v \in V \setminus S$ mit minimaler Oberschranke v.d.
3. Füge v in S ein und relaxiere alle von v ausgehenden Kanten
4. Gehe zu Schritt 2, falls $V \setminus S \neq \emptyset$.

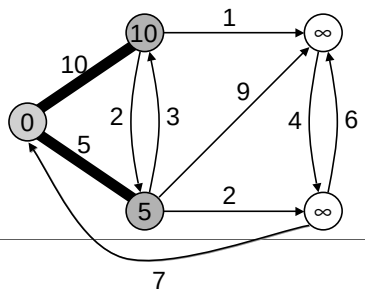
Dijkstras Algorithmus



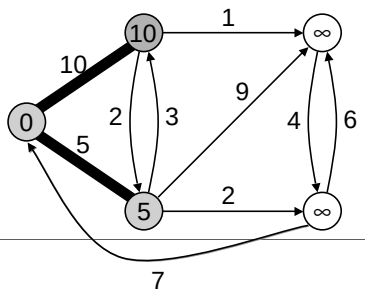
Dijkstras Algorithmus



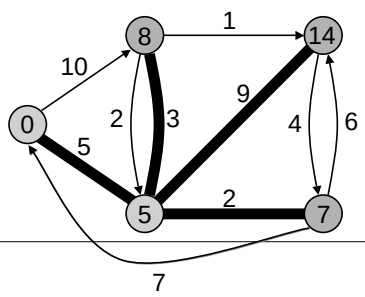
Dijkstras Algorithmus



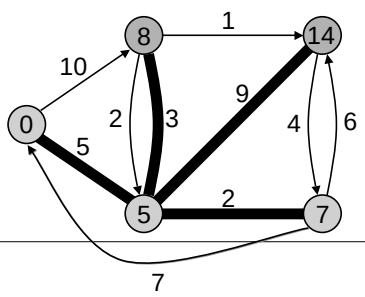
Dijkstras Algorithmus



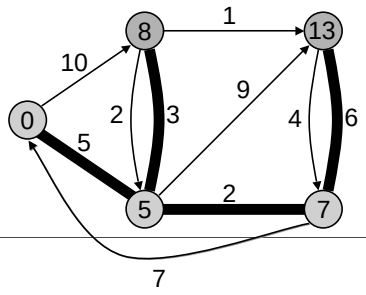
Dijkstras Algorithmus



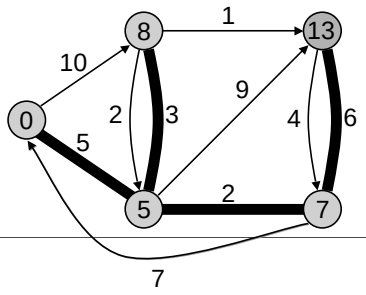
Dijkstras Algorithmus



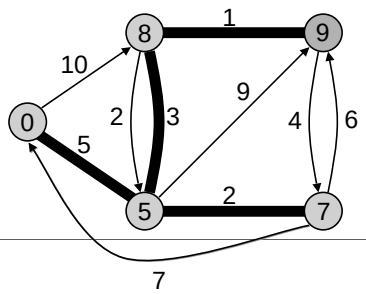
Dijkstras Algorithmus



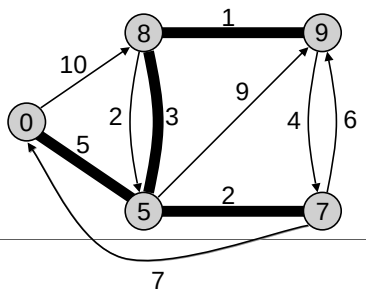
Dijkstras Algorithmus



Dijkstras Algorithmus



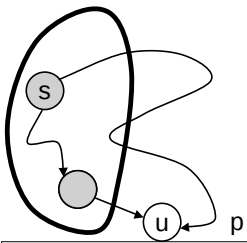
Dijkstras Algorithmus



Dijkstras Algorithmus

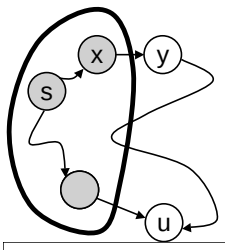
```
Dijkstra(G,s)
InitializeSingleSource(G,s);
S = ∅;
Q = V;
while (Q ≠ ∅)
    u = ExtractMinimum(Q);
    S = S ∪ {u};
    for each v ∈ Adj[u]
        Relax(u,v);
```


Dijkstras Algorithmus



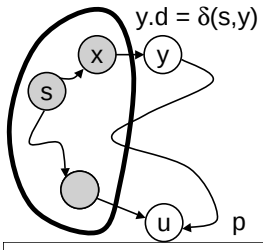
u ist von s erreichbar, es gibt also einen kürzesten Pfad p von s nach u

Dijkstras Algorithmus



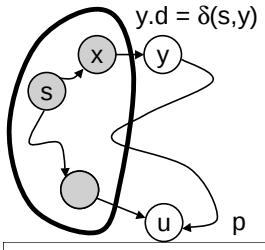
Sei y der erste Knoten auf p der nicht in S liegt und sei x sein Vorgänger

Dijkstras Algorithmus



Teilpfade von kürzesten Pfaden sind kürzeste Pfade, (x,y) wurde bereits relaxiert, also ist y.d = δ(s,y).

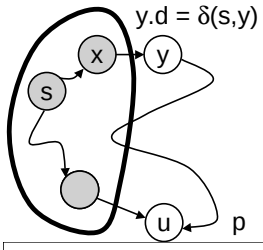
Dijkstras Algorithmus



u wird von Dijkstras Algorithmus vor y ausgewählt, also ist u.d ≤ y.d

u.d ≤ y.d

Dijkstras Algorithmus



u.d ≤ y.d

Es gilt

$y.d = \delta(s,y) \leq \delta(s,u) \leq u.d$

und

$u.d \leq y.d$

also

$y.d = \delta(s,y) = \delta(s,u) = u.d$

und insbesondere

$u.d = \delta(s,u)$

Dijkstras Algorithmus

```
Dijkstra(G,s)
InitializeSingleSource(G,s);      } O(V)
S = ∅;
Q = V;
while (Q ≠ ∅)                     } O((E+V) log V)
   u = ExtractMinimum(Q);
   S = S ∪ {u};
   for each v ∈ Adj[u]
      Relax(u,v);
```


Dijkstras Algorithmus

Fazit:

Falls G zusammenhängend ist, so hat Dijkstra's Algorithmus einen Aufwand von

$$O(E \log V)$$

Kürzeste Pfade

All-pairs shortest-paths Problem

Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar $u \in V$ und $v \in V$ einen kürzesten Pfad von u nach v.

Kürzeste Pfade

In diesem Abschnitt gehen wir davon aus, dass die Knotenmenge des Graphen oBdA V = {1,...,n} ist. Weiter seien die gewichteten Kanten des Graphen in einer Adjazenzmatrix $W = (w_{ij})$ gespeichert.

Floyd-Warshall Algorithmus

Es sei

$$p = v_0, v_1, \ldots, v_{l-1}, v_l$$

ein einfacher Pfad. Die Knoten $v_1,\ldots,v_{l-1}$ heißen Zwischenknoten des Pfades p. Wir bezeichnen die Menge der Pfade, deren sämtliche Zwischenknoten in {1...k}, $0 \le k \le n$, liegen mit P^k.

Floyd-Warshall Algorithmus

Idee: Dynamisches Programmieren

1. Rekursionsformel für die Gewichte der kürzesten Pfade d
2. Rekursionsformel für die Vorgängerattribute p

Floyd-Warshall Algorithmus

Rekursionsformel für die Gewichte

Es sei d_{ij}^k das Gewicht eines kürzesten Pfades von i nach j in P^k mit $0 \le k \le n$. Insbesondere ist also d_{ij}^n das Gewicht eines kürzesten Pfades von i nach j in G.

Floyd-Warshall Algorithmus

Initialisierung: k = 0

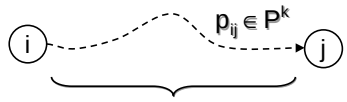
Falls k = 0, so dürfen auf den Pfaden in $P^k=P^0$ überhaupt keine Zwischenknoten liegen, d. h. die Länge der Pfade ist höchstens 1 und wir haben

$$d_{ij}^0 = w_{ij}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Es sei p_{ij} ein kürzester Pfad von i nach j in P^k.



enthält nur Knoten v mit $1 \le v \le k$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

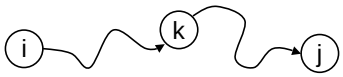
Fall 1: k ist kein Zwischenknoten von p_{ij}
⇒ p_{ij} ist auch ein kürzester Pfad von i nach j in P^{k-1}.

$$d_{ij}^k = d_{ij}^{k-1}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 2: k ist ein Zwischenknoten von p_{ij}



p_{ik} p_{kj}

p_{ik} ist kürzester Pfad von i nach k in P^{k-1}
p_{kj} ist kürzester Pfad von k nach j in P^{k-1}

$$d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

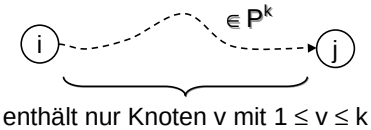
Rekursionsformel:

$$d_{ij}^k = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min\left\{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} \right\} & \text{falls } k > 0 \end{cases}$$

Floyd-Warshall Algorithmus

Rekursionsformel für die Vorgänger

Es sei p_{ij}^k der Vorgänger von Knoten j auf einem kürzesten Pfad aus P^k der bei i beginnt und bei j endet.



enthält nur Knoten v mit $1 \le v \le k$

Floyd-Warshall Algorithmus

Initialisierung: k = 0

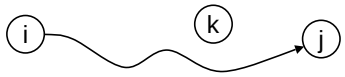
Möglich sind nur Pfade der Länge 1.

$$p_{ij}^0 = \begin{cases} 0 & \text{falls } i = j \text{ oder } w_{ij} = \infty \\ i & \text{falls } i \neq j \text{ und } w_{ij} < \infty \end{cases}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 1: Ein kürzester Pfad $\in P^k$ enthält den Knoten k nicht

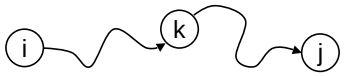


$$\Rightarrow p_{ij}^k = p_{ij}^{k-1} \text{ falls } d_{ij}^{k-1} \le d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 2: Ein kürzester Pfad $\in P^k$ enthält den Knoten k



$$\Rightarrow p_{ij}^k = p_{kj}^{k-1} \text{ falls } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

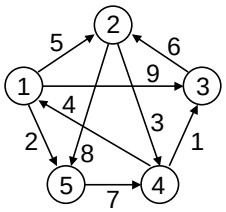
Rekursionsformel:

$$p_{ij}^k = \begin{cases} p_{ij}^{k-1} & \text{falls } d_{ij}^{k-1} \le d_{ik}^{k-1} + d_{kj}^{k-1} \\ p_{kj}^{k-1} & \text{sonst} \end{cases}$$

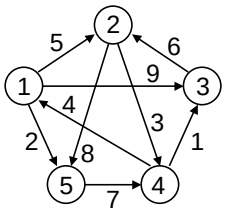
Floyd-Warshall Algorithmus

FloydWarshall(W)
$D^0 = W$; $P^0 = ...$;
for (k = 1; k ≤ n; ++k)
  for (i = 1; i ≤ n; ++i)
    for (j = 1; j ≤ n; ++j)
      $d_{ij}^k = \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right)$
      $p_{ij}^k = ...$
return D^n;

Floyd-Warshall Algorithmus

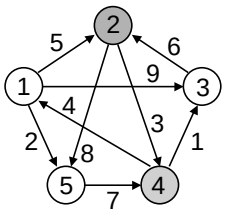


Floyd-Warshall Algorithmus



$$D^0 = \begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

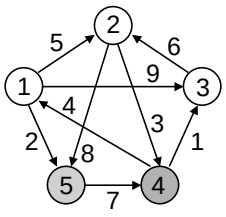
Floyd-Warshall Algorithmus



$D^0 =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

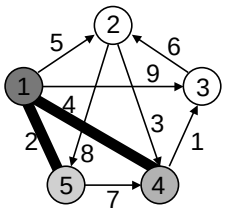
Floyd-Warshall Algorithmus



$D^0 =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

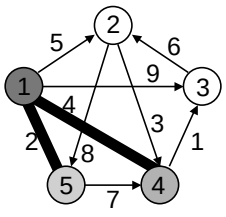
Floyd-Warshall Algorithmus



$D^{0\to1} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

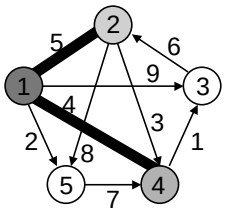
Floyd-Warshall Algorithmus



$D^{0\to1} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

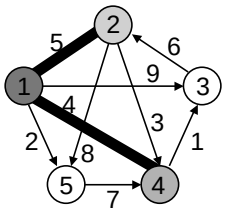
Floyd-Warshall Algorithmus



$D^{0\to1} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & \infty & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

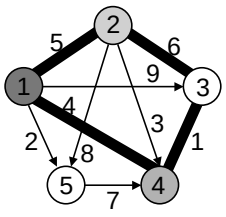
Floyd-Warshall Algorithmus



$D^{0\to1} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

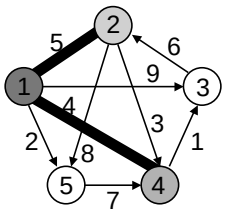
Floyd-Warshall Algorithmus



$D^{0\to1}$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

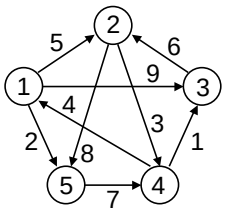
Floyd-Warshall Algorithmus



$D^{0\to1}$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

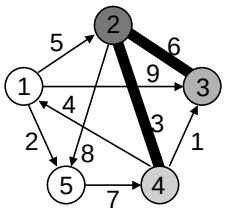
Floyd-Warshall Algorithmus



$D^{1} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

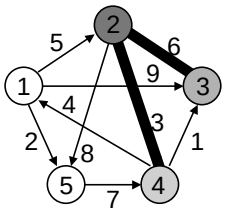
Floyd-Warshall Algorithmus



$D^{1\to2} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & \infty & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

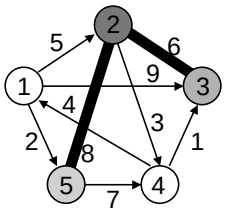
Floyd-Warshall Algorithmus



$D^{1\to2} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

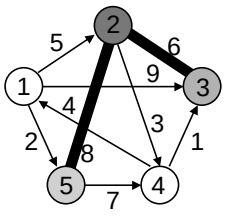
Floyd-Warshall Algorithmus



$D^{1\to2} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & \infty \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

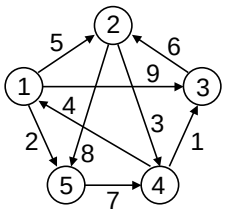
Floyd-Warshall Algorithmus



$D^{1\to 2} =$

$$\begin{pmatrix} 0 & 5 & 9 & \infty & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & 14 \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

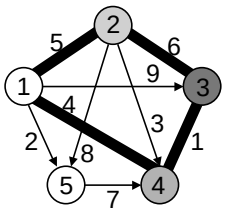
Floyd-Warshall Algorithmus



$D^{2} =$

$$\begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & 14 \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

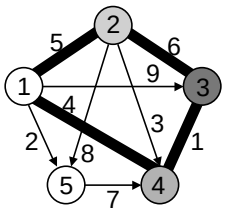
Floyd-Warshall Algorithmus



$D^{2\to 3} =$

$$\begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & 14 \\ 4 & 9 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

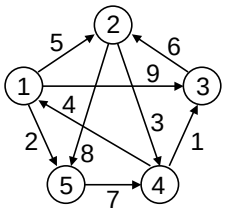
Floyd-Warshall Algorithmus



$D^{2\to 3} =$

$$\begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & 14 \\ 4 & 7 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

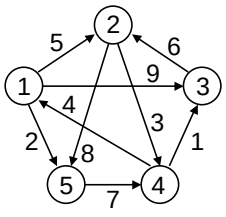
Floyd-Warshall Algorithmus



$D^{3} =$

$$\begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ \infty & 0 & \infty & 3 & 8 \\ \infty & 6 & 0 & 9 & 14 \\ 4 & 7 & 1 & 0 & 6 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix}$$

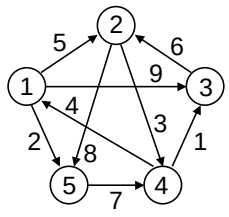
Floyd-Warshall Algorithmus



$D^{4} =$

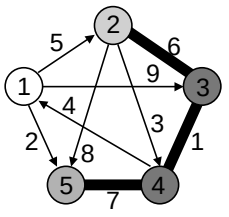
$$\begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ 7 & 0 & 4 & 3 & 8 \\ 13 & 6 & 0 & 9 & 14 \\ 4 & 7 & 1 & 0 & 6 \\ 11 & 14 & 8 & 7 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithmus



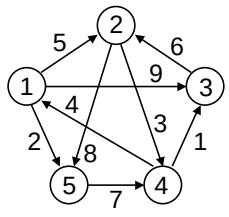
$$D^5 = \begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ 7 & 0 & 4 & 3 & 8 \\ 13 & 6 & 0 & 9 & 14 \\ 4 & 7 & 1 & 0 & 6 \\ 11 & 14 & 8 & 7 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithmus



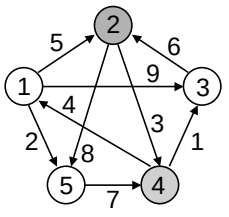
$$D^5 = \begin{pmatrix} 0 & 5 & 9 & 8 & 2 \\ 7 & 0 & 4 & 3 & 8 \\ 13 & 6 & 0 & 9 & 14 \\ 4 & 7 & 1 & 0 & 6 \\ 11 & \boxed{14} & 8 & 7 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithmus



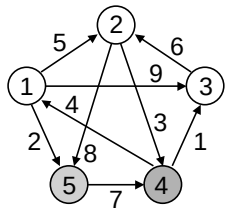
$$P^0 = \begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & \infty \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus



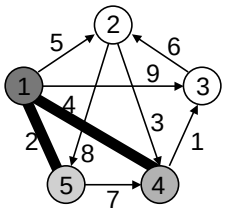
$$P^0 = \begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & \boxed{2} & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & \infty \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus



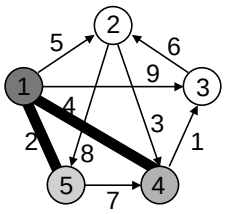
$$P^0 = \begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & \boxed{\infty} \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus



$$P^{0 \to 1} = \begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & \boxed{\infty} \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

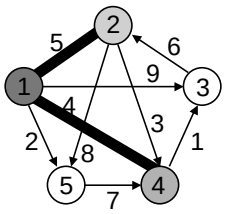
Floyd-Warshall Algorithmus



$P^{0\to1} =$

$$\begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

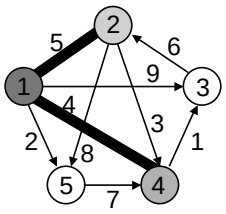
Floyd-Warshall Algorithmus



$P^{0\to1}$

$$\begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

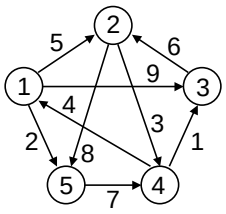
Floyd-Warshall Algorithmus



$P^{0\to1}$

$$\begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & 1 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

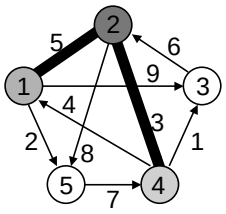
Floyd-Warshall Algorithmus



$P^{1} =$

$$\begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & 1 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

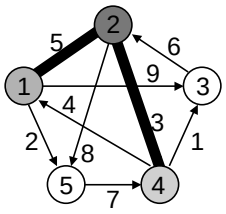
Floyd-Warshall Algorithmus



$P^{1\to2} =$

$$\begin{pmatrix} \infty & 1 & 1 & \infty & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & 1 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

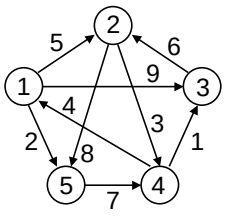
Floyd-Warshall Algorithmus



$P^{1\to2} =$

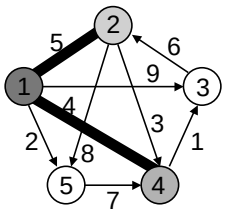
$$\begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & \infty & \infty \\ 4 & 1 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus



$$P^2 = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & 2 & 2 \\ 4 & 1 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

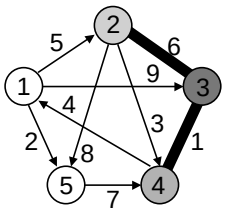
Floyd-Warshall Algorithmus



$$P^{2\to3}$$

$$\begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & 2 & 2 \\ 4 & \textcircled{1} & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

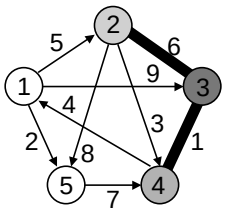
Floyd-Warshall Algorithmus



$$P^{2\to3}$$

$$\begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & 2 & 2 \\ 4 & \textcircled{1} & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

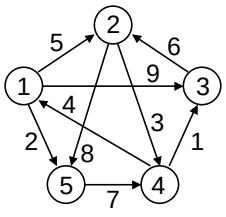
Floyd-Warshall Algorithmus



$$P^{2\to3}$$

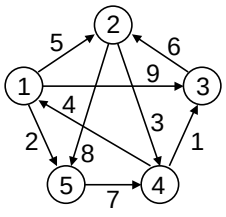
$$\begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & 2 & 2 \\ 4 & \textcircled{3} & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus

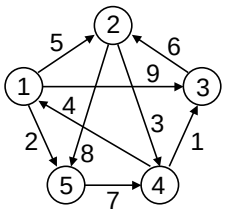


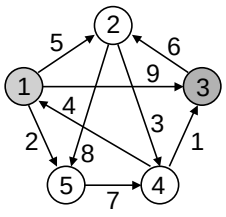
$$P^3 = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ \infty & \infty & \infty & 2 & 2 \\ \infty & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

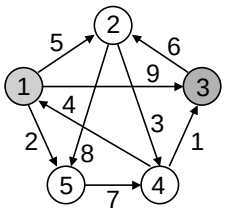
Floyd-Warshall Algorithmus

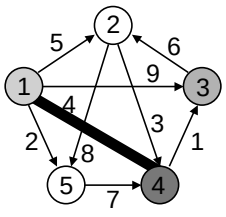


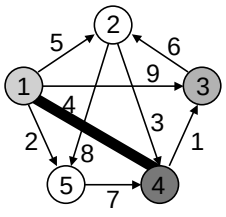
$$P^4 = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ 4 & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

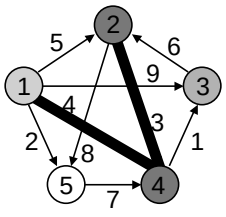












Floyd-Warshall Algorithmus

$$P^5 = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ 4 & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus

$$P = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ 4 & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus

$$P = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ \boxed{4} & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus

$$P = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ \boxed{4} & 3 & \infty & 2 & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

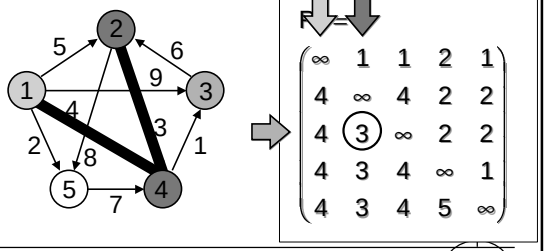
Floyd-Warshall Algorithmus

$$P = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ 4 & 3 & \infty & \boxed{2} & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

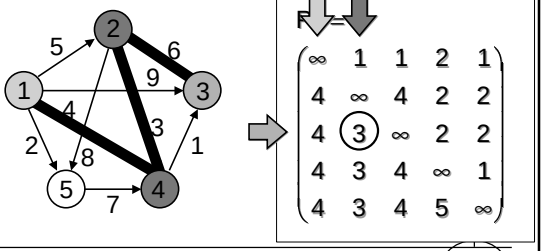
Floyd-Warshall Algorithmus

$$P = \begin{pmatrix} \infty & 1 & 1 & 2 & 1 \\ 4 & \infty & 4 & 2 & 2 \\ 4 & 3 & \infty & \boxed{2} & 2 \\ 4 & 3 & 4 & \infty & 1 \\ 4 & 3 & 4 & 5 & \infty \end{pmatrix}$$

Floyd-Warshall Algorithmus



Floyd-Warshall Algorithmus



Floyd-Warshall Algorithmus

FloydWarshall(W)
$D^0 = W$; $P^0 = ...$; } $O(n^2)$
for (k = 1; k ≤ n; ++k)
  for (i = 1; i ≤ n; ++i)
    for (j = 1; j ≤ n; ++j)
      $d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$
      $p_{ij}^k = ...$
} $O(n^3)$
return D^n;

Floyd-Warshall Algorithmus

Fazit:

Der Floyd-Warshall Algorithmus hat einen Aufwand von

$$O(V^3)$$

Transitive Hülle

Sei G=(V,E) mit V = {1...n} ein Graph. Die transitive Hülle von G ist definiert als

$$G^* = (V,E^*)$$

mit

E^* = { (i,j): es gibt einen Pfad von i nach j in G^* }

Transitive Hülle

Lösung

Ordne jeder Kante in E das Gewicht 1 zu und führe den Floyd-Warshall Algorithmus aus. Gibt es einen Pfad von i nach j so erhalten wir $d_{ij} < n$, andernfalls $d_{ij} = \infty$.

Transitive Hülle

In der Praxis schneller

Ersetze min und + im Floyd-Warshall Algorithmus durch die logischen Operatoren $\vee$ (oder) und $\wedge$ (und).

Transitive Hülle

Definition

Es sei t_{ij}^k gleich 1, falls ein Pfad von i nach j in P^k mit $0 \le k \le n$ existiert. Insbesondere ist also t_{ij}^n gleich 1, falls $(i,j) \in E^*$.

Transitive Hülle

Rekursionsformel:

k = 0:

$$t_{ij}^0 = \begin{cases} 0 & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 1 & \text{falls } i = j \text{ oder } (i,j) \in E \end{cases}$$

k > 0:

$$t_{ij}^k = t_{ij}^{k-1} \vee \left(t_{ik}^{k-1} \wedge t_{kj}^{k-1} \right)$$

Transitive Hülle

```
TransitiveClosure(G)
for (i = 1; ≤ n; ++i)
  for (j = 1; j ≤ n;++j)
    t_ij^0 = (i == j || (i,j) ∈ E);
for (k = 1; k ≤ n; ++k)
  for (i = 1; i ≤ n; ++i)
    for (j = 1; j ≤ n; ++j)
      t_ij^k = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1));
return D^n;
```

Transitive Hülle

Der Algorithmus zur Berechnung der transitiven Hülle hat den gleichen Aufwand wie der Floyd-Warshall Algorithmus, nämlich

$$O(V^3)$$

In der Praxis können jedoch logische Operatoren schneller und bearbeitet werden als arithmetische Operatoren.

Zusammenfassung

Graphen ...

- Traversierung
- Minimale Spannbäume
- Kürzeste Pfade

Zusammenfassung

Graphen ...
- Traversierung
  - Depth-first
  - Breadth-first
- Minimale Spannbäume
- Kürzeste Pfade

Zusammenfassung

Graphen ...
- Traversierung
- Minimale Spannbäume
  - Prim's Algorithmus
  - Kruskal's Algorithmus
- Kürzeste Pfade

Zusammenfassung

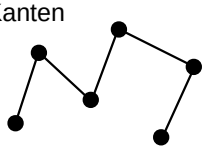
Graphen ...
- Traversierung
- Minimale Spannbäume
- Kürzeste Pfade
  - Dijkstra's Algorithmus
  - Floyd-Warshall Algorithmus

(2.7) Geometrische Algorithmen

- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

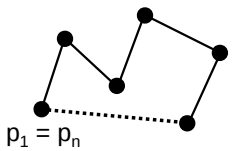
Polygone

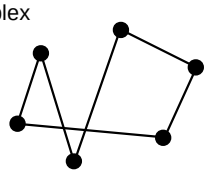
- Sequenz von Knoten / Eckpunkten $p_1, \ldots, p_n$
- Sequenz von Kanten $e_i = [\, p_i \; p_{i+1} \,]$

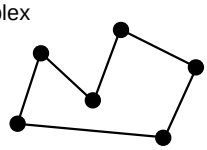


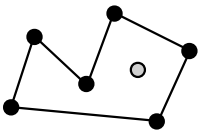
Polygone

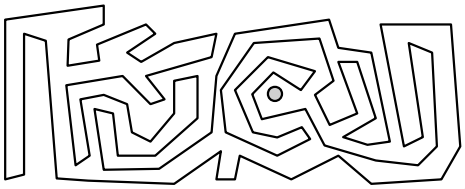
- Eigenschaften
  - offen / geschlossen

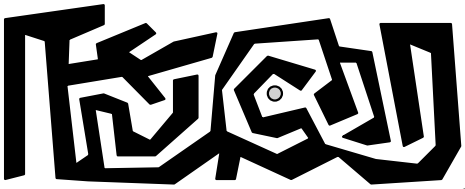


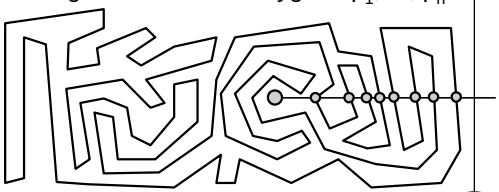












Polygone

- Eigenschaften
  - offen / geschlossen
  - einfach / komplex

Polygone

- Eigenschaften
  - offen / geschlossen
  - einfach / komplex

Inside-Test

- Liegt ein gegebener Punkt q innerhalb des geschlossenen Polygons $p_1, ..., p_n$?

Inside-Test

- Liegt ein gegebener Punkt q innerhalb des geschlossenen Polygons $p_1, ..., p_n$?

Inside-Test

- Liegt ein gegebener Punkt q innerhalb des geschlossenen Polygons $p_1, ..., p_n$?

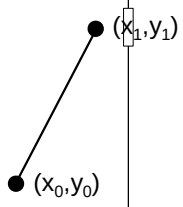
Inside-Test

- Liegt ein gegebener Punkt q innerhalb des geschlossenen Polygons $p_1, ..., p_n$?

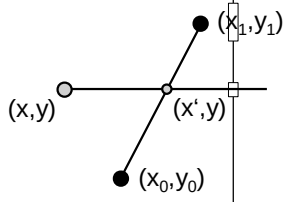
Implementierung

```
Inside(x,y,p[ 1..n ])
  k = 0;
  for (i = 1 ; i ≤ n ; i++)
    if (p[ i ].y < p[ i+1].y)
        x0 = p[i].x; y0 = p[i].y; x1 = p[i+1].x; y1 = p[i+1].y;
    else
        x1 = p[i].x; y1 = p[i].y; x0 = p[i+1].x; y0 = p[i+1].y;
    if (y0 ≤ y < y1)
        if ((x-x0)*(y1-y0) < (y-y0)*(x1-x0))
            k++;
  return ( odd(k) );
```


Schnittberechnung

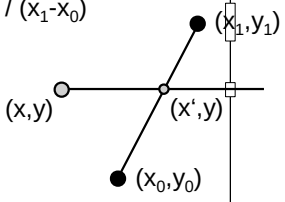


Schnittberechnung



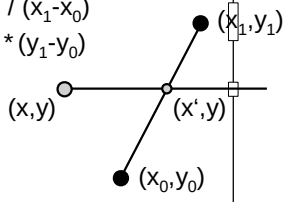
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x‘-x_0) / (x_1-x_0)$



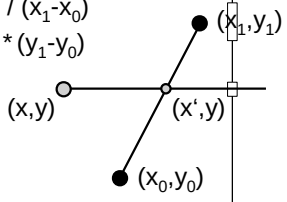
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x‘-x_0) / (x_1-x_0)$
$(y-y_0) * (x_1-x_0) = (x‘-x_0) * (y_1-y_0)$



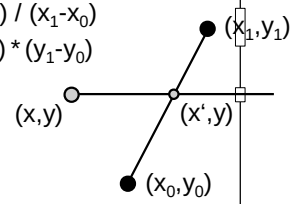
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x‘-x_0) / (x_1-x_0)$
$(y-y_0) * (x_1-x_0) = (x‘-x_0) * (y_1-y_0)$
$x < x‘$



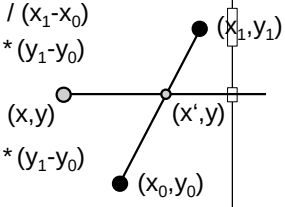
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x'-x_0) / (x_1-x_0)$
$(y-y_0) * (x_1-x_0) = (x'-x_0) * (y_1-y_0)$
$x < x'$
$x-x_0 < x'-x_0$



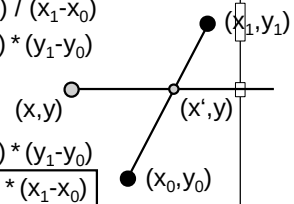
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x'-x_0) / (x_1-x_0)$
$(y-y_0) * (x_1-x_0) = (x'-x_0) * (y_1-y_0)$
$x < x'$
$x-x_0 < x'-x_0$
$(x-x_0) * (y_1-y_0) < (x'-x_0) * (y_1-y_0)$



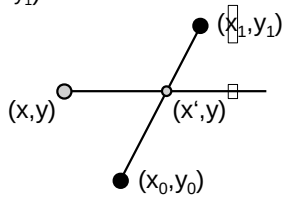
Schnittberechnung

$(y-y_0) / (y_1-y_0) = (x'-x_0) / (x_1-x_0)$
$(y-y_0) * (x_1-x_0) = (x'-x_0) * (y_1-y_0)$
$x < x'$
$x-x_0 < x'-x_0$
$(x-x_0) * (y_1-y_0) < (x'-x_0) * (y_1-y_0)$
$\boxed{(x-x_0) * (y_1-y_0) < (y-y_0) * (x_1-x_0)}$



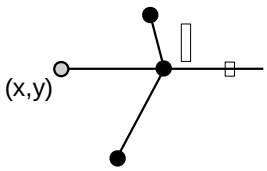
Spezialfälle

…
if $(y_0 \le y < y_1)$
…



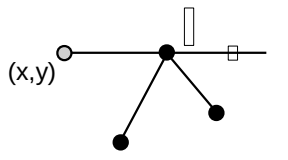
Spezialfälle

…
if $(y_0 \le y < y_1)$
…



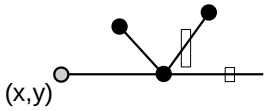
Spezialfälle

…
if $(y_0 \le y < y_1)$
…



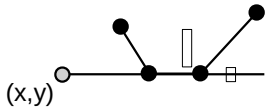
Spezialfälle

...
if ($y_0 \leq y < y_1$)
...



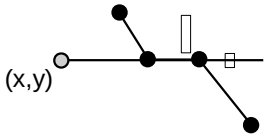
Spezialfälle

...
if ($y_0 \leq y < y_1$)
...



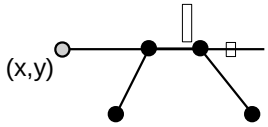
Spezialfälle

...
if ($y_0 \leq y < y_1$)
...



Spezialfälle

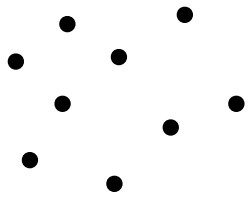
...
if ($y_0 \leq y < y_1$)
...

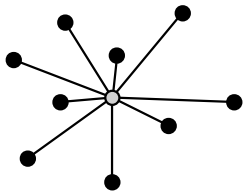


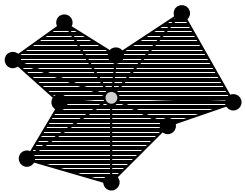
Generierung von Polygonen

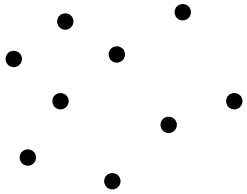
- Gegeben sei eine ungeordnete Menge von Eckpunkten p_i
- Generiere ein einfaches, geschlossenes Polygon (durch „Sortieren“ der p_i)

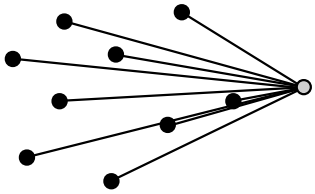
Generierung von Polygonen

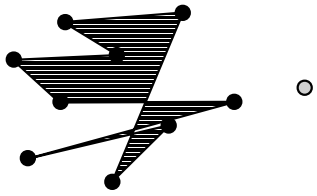


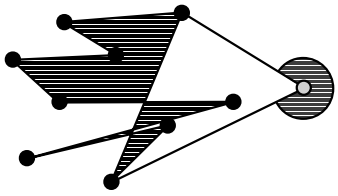












Generierung von Polygonen

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

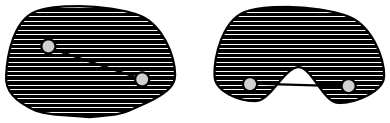
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Konvexität

- Eine Teilmenge S des IR^2 ist ***konvex***, wenn für jedes Paar von Punkten p,q aus S auch alle Zwischenpunkte $\alpha\, p+(1-\alpha)\, q$ mit $\alpha \in [0,1]$ in S liegen.

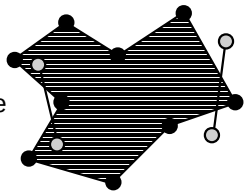
Konvexität

konvex nicht-konvex



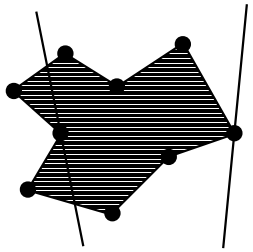
Lokale Konvexität

„nicht-konvexe Ecke“



„konvexe Ecke“

Supporting Lines



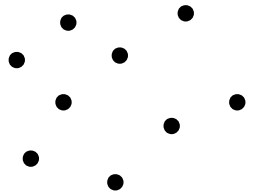
Lokale Konvexität

- ***Satz:***
  Der Eckpunkt mit der kleinsten x-Koordinate (und ggf. zusätzlich mit der kleinsten y-Koordinate) ist immer konvex.
- ***Beweis:***
  Supporting Line parallel zur y-Achse

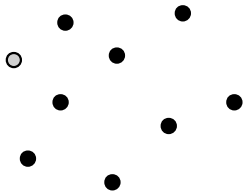
Generierung von Polygonen

- Suche eine konvexe Ecke
- Sortiere die übrigen Ecken nach dem Winkel zur x-Achse
- Verbinde aufeinanderfolgende Ecken

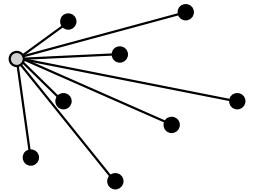
Generierung von Polygonen



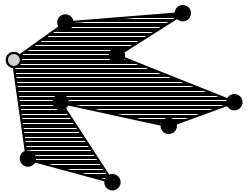
Generierung von Polygonen



Generierung von Polygonen



Generierung von Polygonen



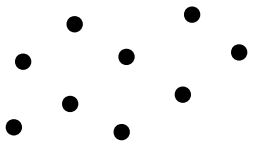
Konvexe Hülle

- Sei eine Menge von (Eck-)Punkten p_i gegeben, dann ist die ***konvexe Hülle*** CH($\{p_i\}$) die Menge aller Punkte q, für die gilt:

$$q = \sum_{i=1}^{n} \alpha_i \, p_i \quad \text{mit} \quad \forall i : \alpha_i \geq 0$$

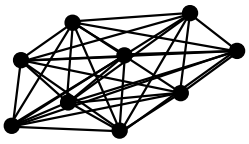
Konvexe Hülle

- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



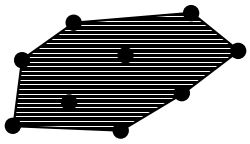
Konvexe Hülle

- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



Konvexe Hülle

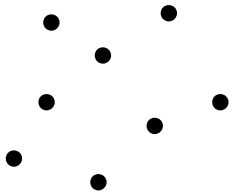
- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



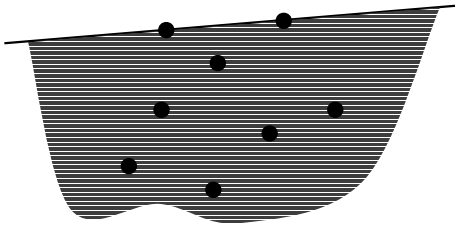
Konvexe Hülle

- Jede Supporting Line definiert einen Halbraum, in dem sich alle Eckpunkte befinden
- Die konvexe Hülle ergibt sich aus der Schnittmenge aller dieser Halbräume

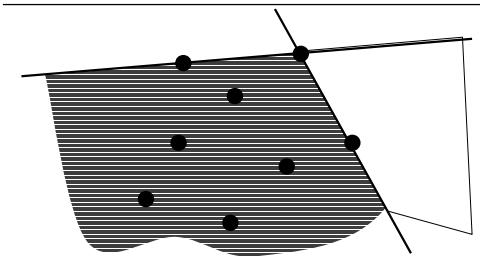
Konvexe Hülle



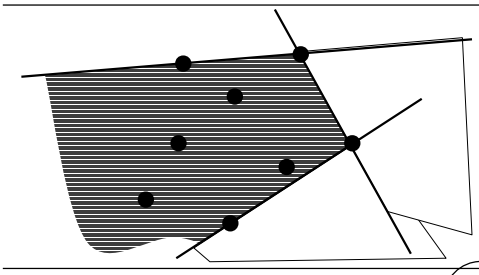
Konvexe Hülle



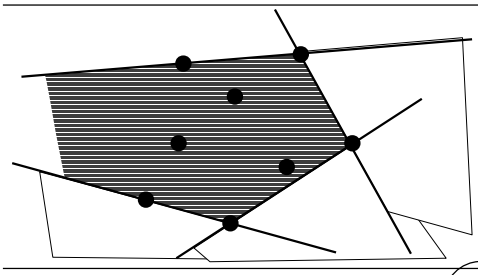
Konvexe Hülle



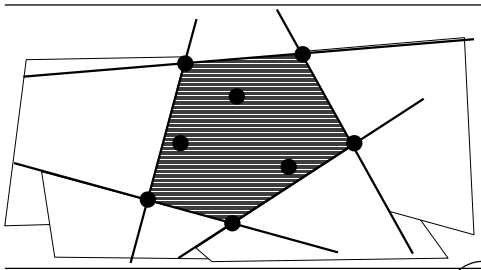
Konvexe Hülle



Konvexe Hülle



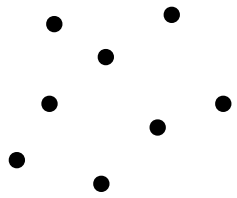
Konvexe Hülle



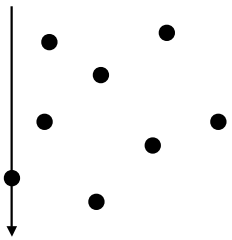
Gift Wrapping Ansatz

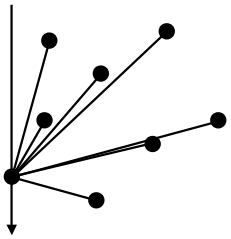
- Beginne mit einer Supporting Line
- Ergänze in jedem Schritt den Eckpunkt, der den geringsten Winkel zur aktuellen Supporting Line aufspannt

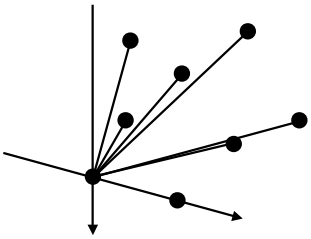
Gift Wrapping Ansatz

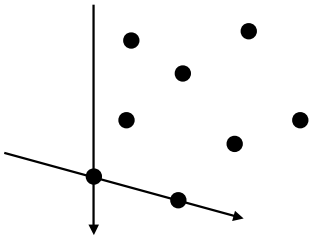


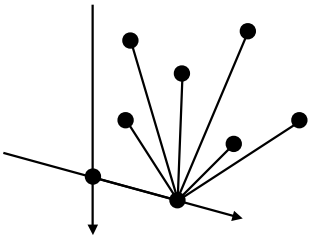
Gift Wrapping Ansatz

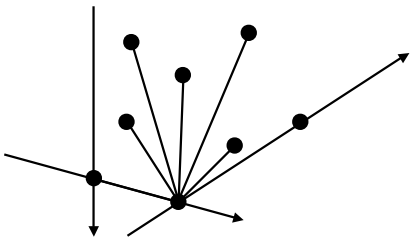


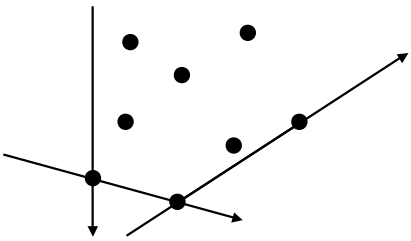












Gift Wrapping Ansatz

Gift Wrapping Ansatz

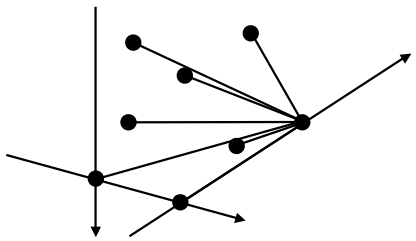
Gift Wrapping Ansatz

Gift Wrapping Ansatz

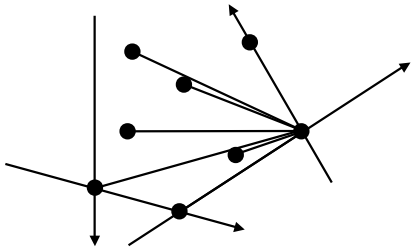
Gift Wrapping Ansatz

Gift Wrapping Ansatz

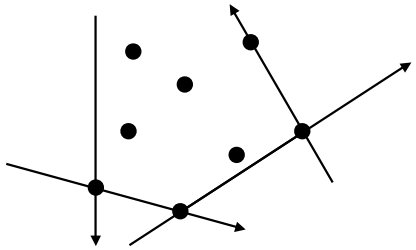
Gift Wrapping Ansatz



Gift Wrapping Ansatz



Gift Wrapping Ansatz



Gift Wrapping Ansatz

- GiftWrapping(Pts[1..n])
  p = left-most point Pts[i]; q = NULL;
  CH = Create(); CH.Add(p);
  L = negative Y-axis
  while ($q \neq$ CH.Top())
    q = select Pts[j] with smallest angle to (p,L)
    CH.Add(q);
    L = line(p,q);
    p = q;

Gift Wrapping Ansatz

- Die Gift Wrapping Prozedur entspricht im wesentlichen dem Selection Sort.
- Aufwand $O(m^2)$, wobei m die Anzahl der Eckpunkte der konvexen Hülle ist
- Worst case: m = n
- Best case: m = 3

(2.6) Graphen

- ✓(2.6.1) Definitionen, Darstellung
- ✓(2.6.2) Ausspähen von Graphen
- ✓(2.6.3) Minimal spannende Bäume
- (2.6.4) Kürzeste Pfade

Kürzeste Pfade

All-pairs shortest-paths Problem

Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar $u \in V$ und $v \in V$ einen kürzesten Pfad von u nach v.

Floyd-Warshall Algorithmus

Es sei

$$p = v_0, v_1, \ldots, v_{l-1}, v_l$$

ein einfacher Pfad. Die Knoten $v_1,\ldots,v_{l-1}$ heißen Zwischenknoten des Pfades p. Wir bezeichnen die Menge der Pfade, deren sämtliche Zwischenknoten in $\{1...k\}$, $0 \le k \le n$, liegen mit P^k.

Floyd-Warshall Algorithmus

Idee: Dynamisches Programmieren

1. Rekursionsformel für die Gewichte der kürzesten Pfade d
2. Rekursionsformel für die Vorgängerattribute p

Floyd-Warshall Algorithmus

Rekursionsformel für die Gewichte

Es sei d_{ij}^k das Gewicht eines kürzesten Pfades von i nach j in P^k mit $0 \le k \le n$. Insbesondere ist also d_{ij}^n das Gewicht eines kürzesten Pfades von i nach j in G.

Floyd-Warshall Algorithmus

Initialisierung: k = 0

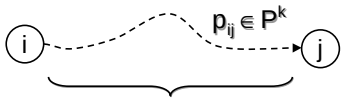
Falls k = 0, so dürfen auf den Pfaden in $P^k=P^0$ überhaupt keine Zwischenknoten liegen, d. h. die Länge der Pfade ist höchstens 1 und wir haben

$$d_{ij}^0 = w_{ij}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Es sei p_{ij} ein kürzester Pfad von i nach j in P^k.



enthält nur Knoten v mit $1 \le v \le k$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

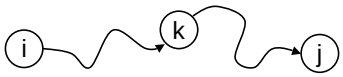
Fall 1: k ist kein Zwischenknoten von p_{ij}
⇒ p_{ij} ist auch ein kürzester Pfad von i nach j in P^{k-1}.

$$d_{ij}^k = d_{ij}^{k-1}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 2: k ist ein Zwischenknoten von p_{ij}



p_{ik} p_{kj}

p_{ik} ist kürzester Pfad von i nach k in P^{k-1}
p_{kj} ist kürzester Pfad von k nach j in P^{k-1}

$$d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

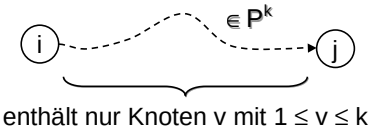
Rekursionsformel:

$$d_{ij}^k = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min\left\{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} \right\} & \text{falls } k > 0 \end{cases}$$

Floyd-Warshall Algorithmus

Rekursionsformel für die Vorgänger

Es sei p_{ij}^k der Vorgänger von Knoten j auf einem kürzesten Pfad aus P^k der bei i beginnt und bei j endet.



enthält nur Knoten v mit $1 \le v \le k$

Floyd-Warshall Algorithmus

Initialisierung: k = 0

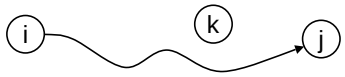
Möglich sind nur Pfade der Länge 1.

$$p_{ij}^0 = \begin{cases} 0 & \text{falls } i = j \text{ oder } w_{ij} = \infty \\ i & \text{falls } i \neq j \text{ und } w_{ij} < \infty \end{cases}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 1: Ein kürzester Pfad $\in P^k$ enthält den Knoten k nicht

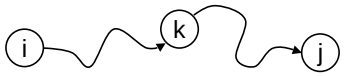


$$\Rightarrow p_{ij}^k = p_{ij}^{k-1} \text{ falls } d_{ij}^{k-1} \le d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

Iteration: k–1→ k

Fall 2: Ein kürzester Pfad $\in P^k$ enthält den Knoten k



$$\Rightarrow p_{ij}^k = p_{kj}^{k-1} \text{ falls } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$$

Floyd-Warshall Algorithmus

Rekursionsformel:

$$p_{ij}^k = \begin{cases} p_{ij}^{k-1} & \text{falls } d_{ij}^{k-1} \le d_{ik}^{k-1} + d_{kj}^{k-1} \\ p_{kj}^{k-1} & \text{sonst} \end{cases}$$

Floyd-Warshall Algorithmus

FloydWarshall(W)
$D^0 = W$; $P^0 = ...$;
for (k = 1; k ≤ n; ++k)
  for (i = 1; i ≤ n; ++i)
    for (j = 1; j ≤ n; ++j)
      $d_{ij}^k = \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right)$
      $p_{ij}^k = ...$
return D^n;

Floyd-Warshall Algorithmus

FloydWarshall(W)
$D^0 = W$; $P^0 = ...$; } $O(n^2)$
for (k = 1; k ≤ n; ++k)
  for (i = 1; i ≤ n; ++i)
    for (j = 1; j ≤ n; ++j)
      $d_{ij}^k = \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right)$
      $p_{ij}^k = ...$
return D^n;

} $O(n^3)$

Floyd-Warshall Algorithmus

Fazit:

Der Floyd-Warshall Algorithmus hat einen Aufwand von

$$O(V^3)$$

Zusammenfassung

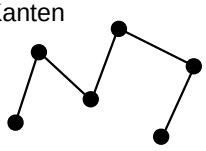
Graphen ...
- Traversierung
- Minimale Spannbäume
- Kürzeste Pfade

(2.7) Geometrische Algorithmen

- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

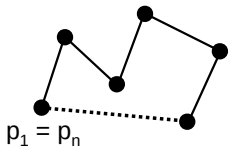
Polygone

- Sequenz von Knoten / Eckpunkten $p_1, ..., p_n$
- Sequenz von Kanten $e_i = [\, p_i \; p_{i+1} \,]$



Polygone

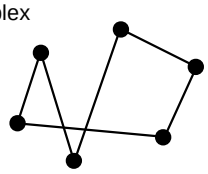
- Eigenschaften
  - offen / geschlossen



$p_1 = p_n$

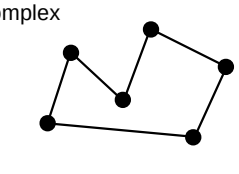
Polygone

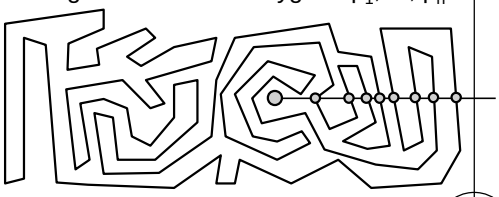
- Eigenschaften
  - offen / geschlossen
  - einfach / komplex

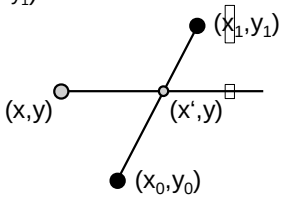


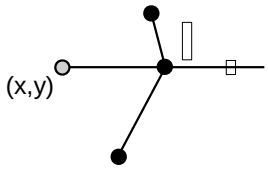
Polygone

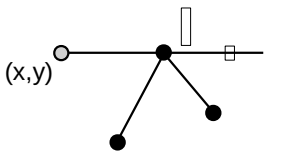
- Eigenschaften
  - offen / geschlossen
  - einfach / komplex

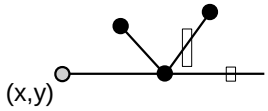












Inside-Test

- Liegt ein gegebener Punkt q innerhalb des geschlossenen Polygons $p_1, ..., p_n$?

Implementierung

```
Inside(x,y,p[ 1..k ])
  k = 0;
  for (i = 1 ; i ≤ n ; i++)
      if (p[ i ].y < p[ i+1].y)
          $x_0$ = p[i].x; $y_0$ = p[i].y; $x_1$ = p[i+1].x; $y_1$ = p[i+1].y;
      else
          $x_1$ = p[i].x; $y_1$ = p[i].y; $x_0$ = p[i+1].x; $y_0$ = p[i+1].y;
      if ($y_0 \leq y < y_1$)
          if ($(x-x_0)*(y_1-y_0) < (y-y_0)*(x_1-x_0)$)
              k++;
  return ( odd(k) );
```

Spezialfälle

...
if ($y_0 \leq y < y_1$)
...

Spezialfälle

...
if ($y_0 \leq y < y_1$)
...

Spezialfälle

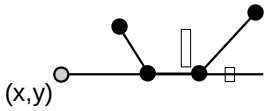
...
if ($y_0 \leq y < y_1$)
...

Spezialfälle

...
if ($y_0 \leq y < y_1$)
...

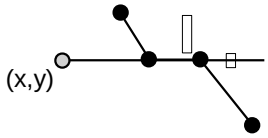
Spezialfälle

...
if ($y_0 \le y < y_1$)
...



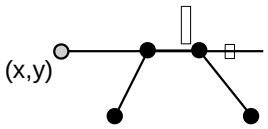
Spezialfälle

...
if ($y_0 \le y < y_1$)
...



Spezialfälle

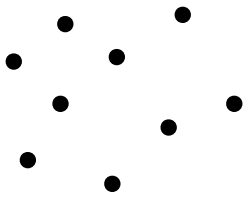
...
if ($y_0 \le y < y_1$)
...



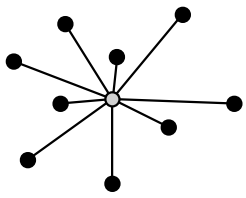
Generierung von Polygonen

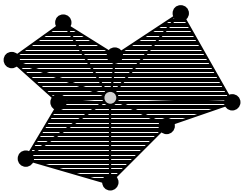
- Gegeben sei eine ungeordnete Menge von Eckpunkten p_i
- Generiere ein einfaches, geschlossenes Polygon (durch „Sortieren“ der p_i)

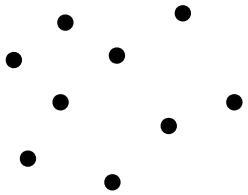
Generierung von Polygonen

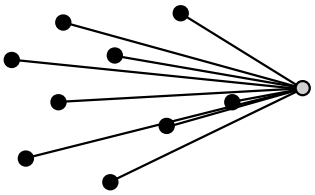


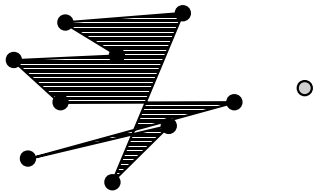
Generierung von Polygonen

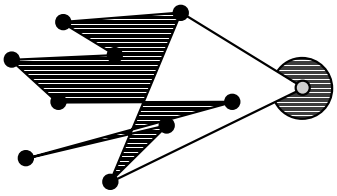












Generierung von Polygonen

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Generierung von Polygonen

aber ...

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

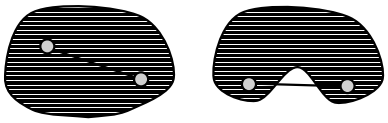
Konvexität

- Eine Teilmenge S des IR^2 ist ***konvex***, wenn für jedes Paar von Punkten p,q aus S auch alle Zwischenpunkte $\alpha\, p+(1-\alpha)\, q$ mit $\alpha \in [0,1]$ in S liegen.

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

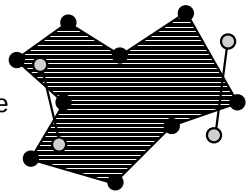
Konvexität

konvex nicht-konvex



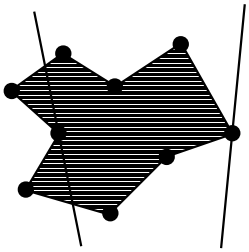
Lokale Konvexität

„nicht-konvexe Ecke“



„konvexe Ecke“

Supporting Lines



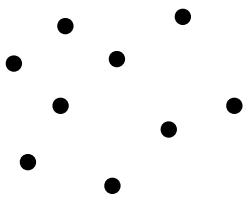
Lokale Konvexität

- ***Satz:***
  Der Eckpunkt mit der kleinsten x-Koordinate (und ggf. zusätzlich mit der kleinsten y-Koordinate) ist immer konvex.
- ***Beweis:***
  Supporting Line parallel zur y-Achse

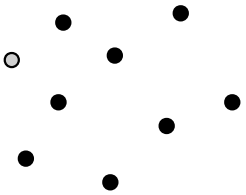
Generierung von Polygonen

- Suche eine konvexe Ecke
- Sortiere die übrigen Ecken nach dem Winkel zur x-Achse
- Verbinde aufeinanderfolgende Ecken

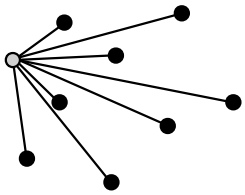
Generierung von Polygonen



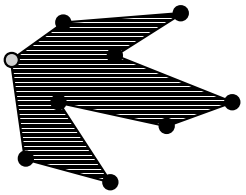
Generierung von Polygonen



Generierung von Polygonen



Generierung von Polygonen



(2.7) Geometrische Algorithmen

- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

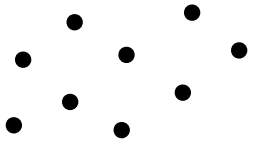
Konvexe Hülle

- Sei eine Menge von (Eck-)Punkten p_i gegeben, dann ist die ***konvexe Hülle*** CH($\{p_i\}$) die Menge aller Punkte q, für die gilt:

$$q = \sum_{i=1}^{n} \alpha_i \, p_i \quad \textbf{mit} \quad \sum_{i=1}^{n} \alpha_i = 1 \quad \textbf{und} \quad \forall i : \alpha_i \geq 0$$

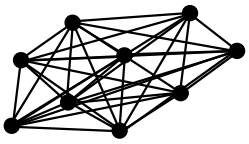
Konvexe Hülle

- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



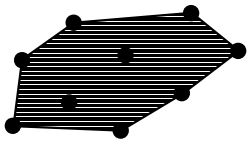
Konvexe Hülle

- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



Konvexe Hülle

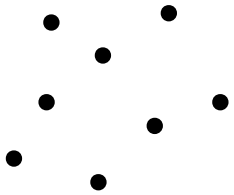
- Insbesondere enthält die konvexe Hülle alle Kanten zwischen zwei Eckpunkten



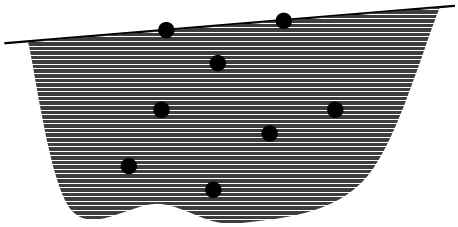
Konvexe Hülle

- Jede Supporting Line definiert einen Halbraum, in dem sich alle Eckpunkte befinden
- Die konvexe Hülle ergibt sich aus der Schnittmenge aller dieser Halbräume

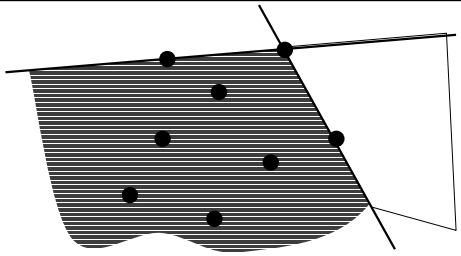
Konvexe Hülle



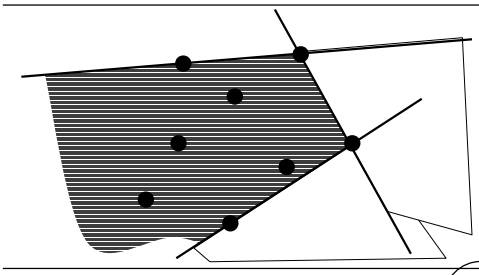
Konvexe Hülle



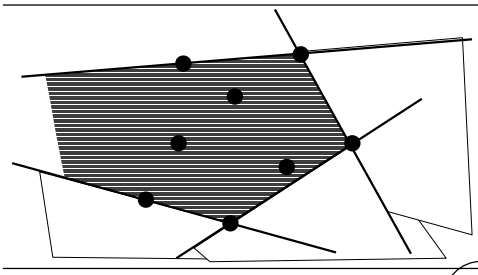
Konvexe Hülle



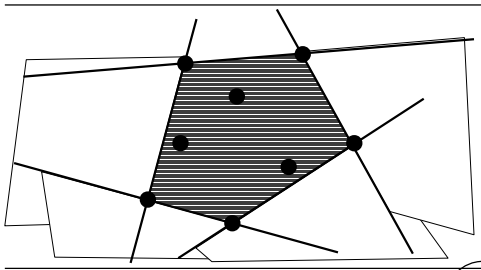
Konvexe Hülle



Konvexe Hülle



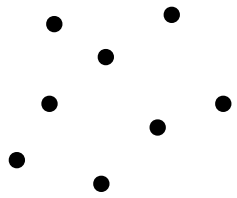
Konvexe Hülle



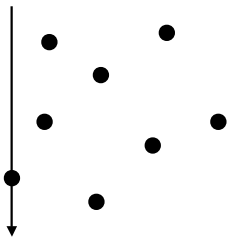
Gift Wrapping Ansatz

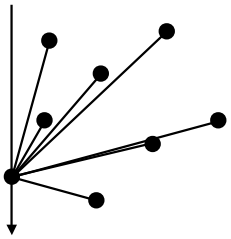
- Beginne mit einer Supporting Line
- Ergänze in jedem Schritt den Eckpunkt, der den geringsten Winkel zur aktuellen Supporting Line aufspannt

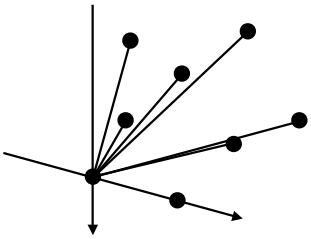
Gift Wrapping Ansatz

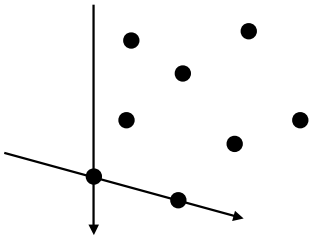


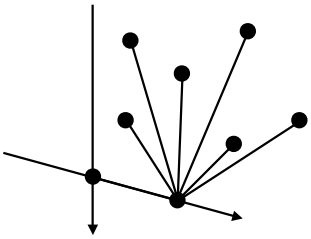
Gift Wrapping Ansatz

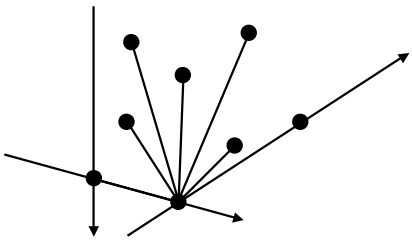


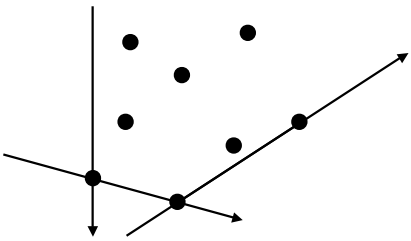












Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

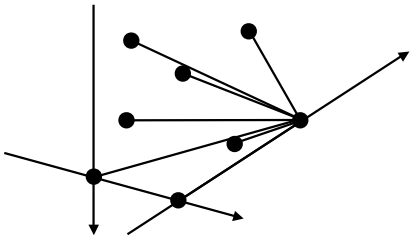
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

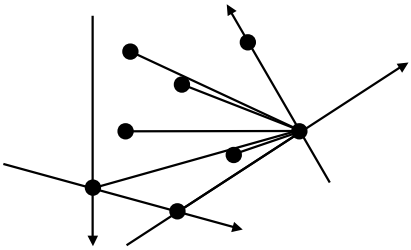
Gift Wrapping Ansatz

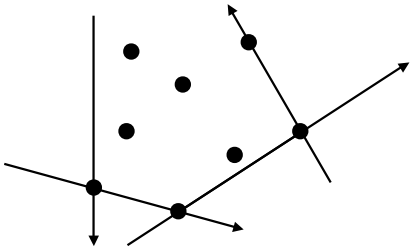
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

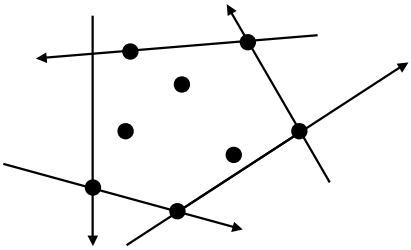
Gift Wrapping Ansatz

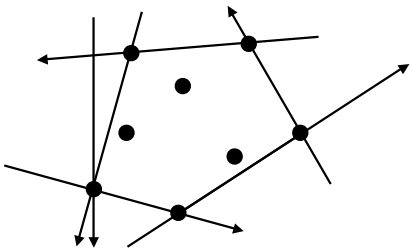
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

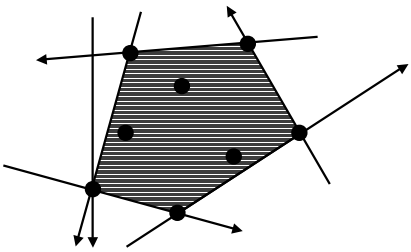












Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Gift Wrapping Ansatz

- GiftWrapping(Pts[1..n])
  p = left-most point Pts[i]; q = NULL;
  CH = Create(); CH.Add(p);
  L = negative Y-axis
  while (q ≠ CH.Top())
    q = select Pts[j] with smallest angle to (p,L)
    CH.Add(q);
    L = line(p,q);
    p = q;

Gift Wrapping Ansatz

- Die Gift Wrapping Prozedur entspricht im wesentlichen dem Selection Sort.
- Aufwand O(m n), wobei m die Anzahl der Eckpunkte der konvexen Hülle ist
- Worst case: m = n
- Best case: m = 3

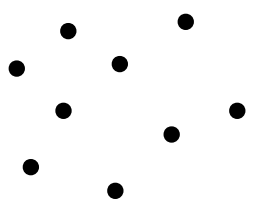
Graham‘s Scan

- Gift Wrapping führt zu viele redundante Berechnungen durch
- Bei der einfachen Generierung von Polygonen genügt ein einziger globaler Sortierungsschritt
- ⇒ entferne die nicht-konvexen Ecken aus dem einfachen Polygon

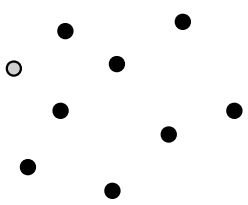
Graham‘s Scan

- **Graham(Pts[1..n])**
  p = left-most point Pts[i];
  CH = Create(); CH.Add(p);
  CH.Add(Pts[i] sorted by the angle of line(p,Pts[i]) to the x-axis);
  i = p; j = CH.next(i); k = CH.next(j);
  while (k ≠ p)
    if (convex(i,j,k))
      i = j; j = k; k = CH.next(k);
    else
      CH.remove(j);
      j = i; i = CH.prev(i);

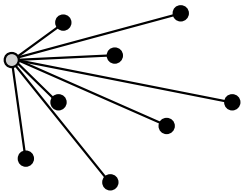
Graham‘s Scan



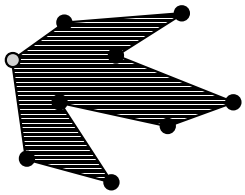
Graham‘s Scan



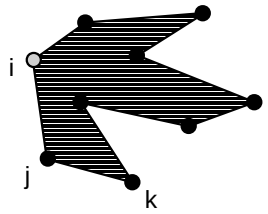
Graham‘s Scan



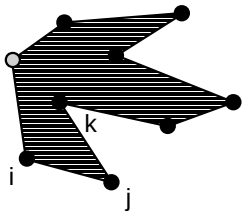
Graham‘s Scan



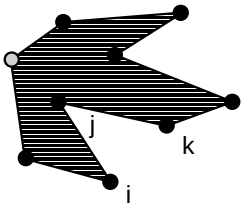
Graham‘s Scan



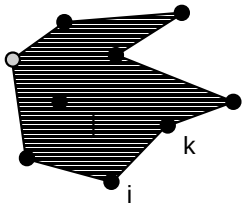
Graham‘s Scan



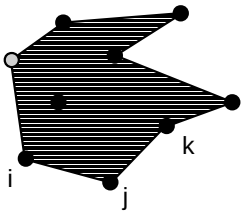
Graham‘s Scan



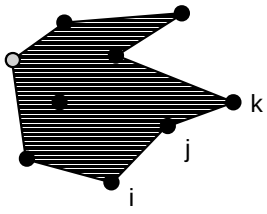
Graham‘s Scan



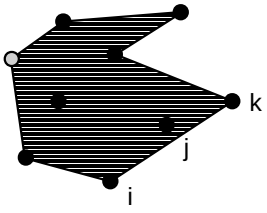
Graham‘s Scan



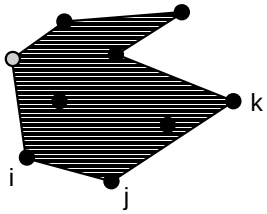
Graham‘s Scan



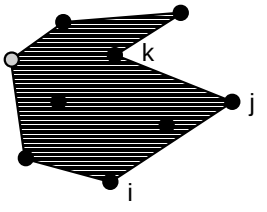
Graham‘s Scan



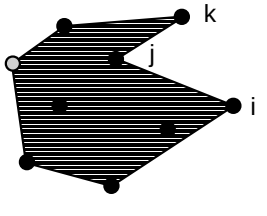
Graham‘s Scan



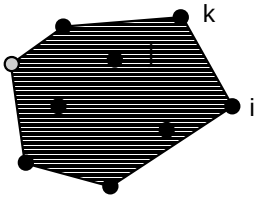
Graham‘s Scan



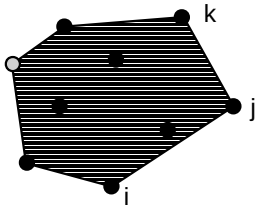
Graham‘s Scan



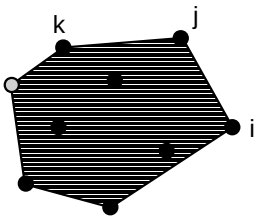
Graham‘s Scan



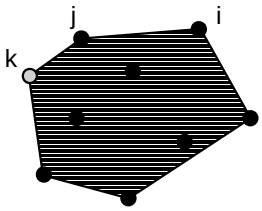
Graham‘s Scan



Graham‘s Scan



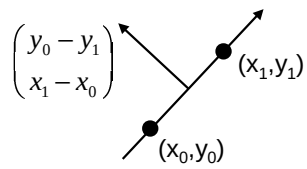
Graham‘s Scan



Halbraum-Test

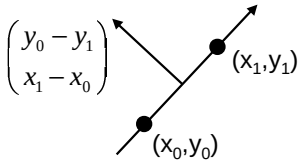
- Sei eine Supporting Line durch die Punkte (x_0,y_0) und (x_1,y_1) gegeben
- Die Reihenfolge der Punkte definiert eine Orientierung
- Links ist innen, d.h. der Vektor zeigt senkrecht nach innen $\begin{pmatrix} y_0 - y_1 \\ x_1 - x_0 \end{pmatrix}$

Halbraum-Test



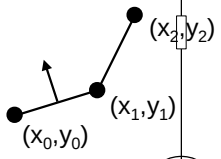
Halbraum-Test

- Punkt (x,y) im Inneren:
  $(y_0-y_1)*x+(x_1-x_0)*y \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$



Konvexitäts-Test

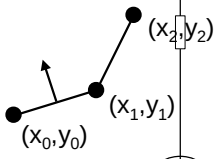
$(y_0-y_1)*x_2+(x_1-x_0)*y_2 \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$



Konvexitäts-Test

$(y_0-y_1)*x_2+(x_1-x_0)*y_2 \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$

$(y_0-y_1)*(x_2-x_0)+(x_1-x_0)*(y_2-y_0) \geq 0$

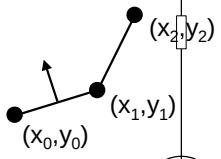


Konvexitäts-Test

$(y_0-y_1)*x_2+(x_1-x_0)*y_2 \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$

$(y_0-y_1)*(x_2-x_0)+(x_1-x_0)*(y_2-y_0) \geq 0$

$\det\begin{pmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{pmatrix} \geq 0$

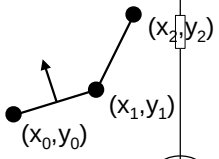


Konvexitäts-Test

$(y_0-y_1)*x_2+(x_1-x_0)*y_2 \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$

$(y_0-y_1)*(x_2-x_0)+(x_1-x_0)*(y_2-y_0) \geq 0$

$\det\begin{pmatrix} x_1-x_0 & x_2-x_1 \\ y_1-y_0 & y_2-y_1 \end{pmatrix} \geq 0$



Winkelberechnung

- Winkel der Kante von (x_0,y_0) nach (x_1,y_1) zur x-Achse

$$\tan\alpha = \frac{y_1-y_0}{x_1-x_0}$$

$$(\sin\alpha)^2 = \frac{(y_1-y_0)^2}{(x_1-x_0)^2+(y_1-y_0)^2}$$

Aufwandsabschätzung

- **Graham(Pts[1..n])**
  p = left-most point Pts[i]; } O(n)
  CH = Create(); CH.Add(p);
  CH.Add(Pts[i] sorted by the angle of line(p,Pts[i]) to the x-axis); } O(n log n)
  i = p; j = CH.next(i); k = CH.next(j);
  while (k ≠ p) } O(n)
  if (convex(i,j,k))
  i = j; j = k; k = CH.next(k); } O(m)
  else
  CH.remove(j);
  j = i; i = CH.prev(i); } O(n-m)

Aufwandsabschätzung

- O(n) + O(m) + O(n-m) = O(n)
- Gesamtaufwand:
  O(n) + O(n log n) = O(n log n)

Divide & Conquer

- Die konvexe Hülle von drei Punkten (in allg. Lage) ist ein Dreieck
- Teile die Gesamtmenge der p_i in zwei Teilmengen mit ***disjunkter*** konvexer Hülle
- Vereinige die konvexen Hüllen der Teilmengen

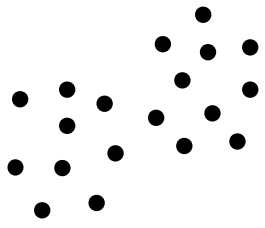
Divide & Conquer

- ConvexHull(P[1...n])
  sort P[ ] by increasing x-coordinate
  remove interior points if ≥ 3 points have the same x-coordinate
  ConvexHullRec(P[1...n])

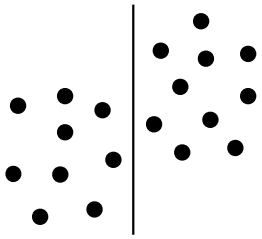
Divide & Conquer

- ConvexHullRec(P[1...n])
  $Q_1 = P[1...\lfloor n/2 \rfloor]$;
  $Q_2 = P[\lfloor n/2 \rfloor+1...n]$;
  $[q_1 \dots q_m] = \text{ConvexHullRec}(Q_1)$;
  $[q'_1 \dots q'_{m'}] = \text{ConvexHullRec}(Q_2)$;
  find upper supporting line (q_i, q'_j)
  find lower supporting line (q_k, q'_l)
  return $[q_i \dots q_k, q'_l \dots q'_j]$;

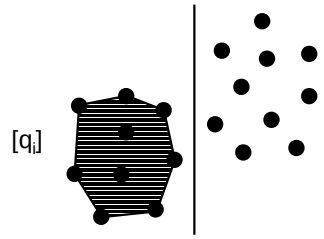
Divide & Conquer



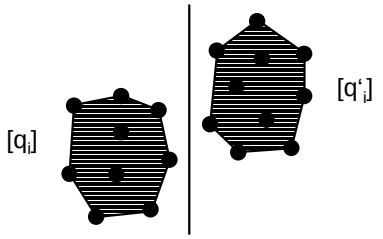
Divide & Conquer



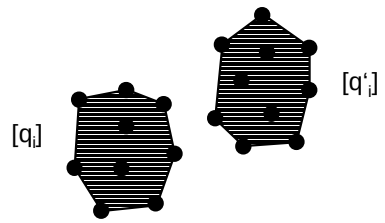
Divide & Conquer



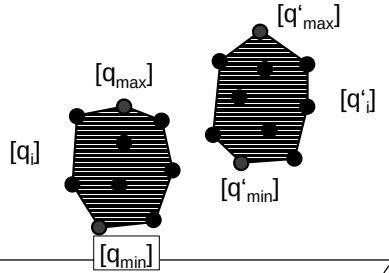
Divide & Conquer



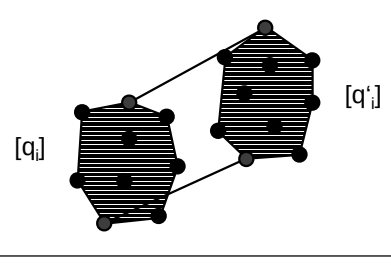
Divide & Conquer



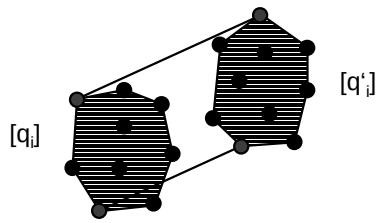
Divide & Conquer



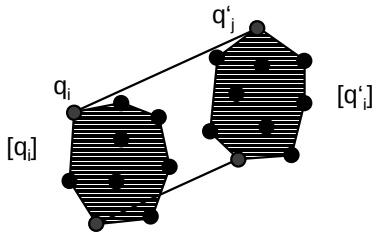
Divide & Conquer



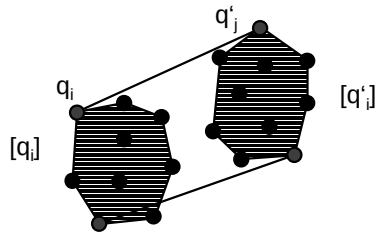
Divide & Conquer



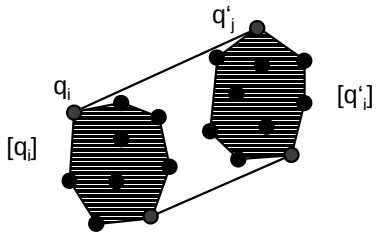
Divide & Conquer



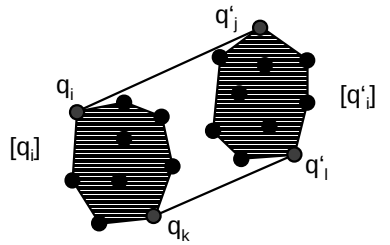
Divide & Conquer



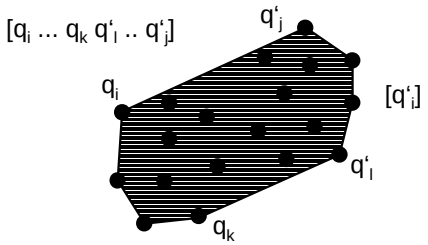
Divide & Conquer



Divide & Conquer



Divide & Conquer



Supporting Lines

- Bestimme Punkte mit minimaler und maximaler y-Koordinate
- Verschiebe die Supporting Line solange die entsprechende Ecke nicht konvex ist oder der Nachbarpunkt außerhalb liegt
- Fallunterscheindung nach $q_{min} \gtrless q'_{min}$ und $q_{max} \gtrless q'_{max}$

Aufwandsabschätzung

ConvexHull(P[1...n])

sort P[ ] by increasing x-coordinate } O(n log n)

remove interior points if ≥ 3 points have the same x-coordinate } O(n)

ConvexHullRec(P[1...n])

Aufwandsabschätzung

ConvexHullRec(P[1...n])

$Q_1 = P[1...\lfloor n/2 \rfloor]$;
$Q_2 = P[\lfloor n/2 \rfloor+1...n]$; } O(1)

$[q_1 \ldots q_m] = \text{ConvexHullRec}(Q_1)$;
$[q'_1 \ldots q'_{m'}] = \text{ConvexHullRec}(Q_2)$;
find upper supporting line (q_i, q'_j)
find lower supporting line (q_k, q'_l) } O(n)
return $[q_i \ldots q_k, q'_l \ldots q'_j]$;

Aufwandsabschätzung

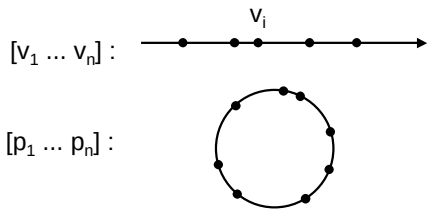
- T(n) = 2 T(n/2) + O(n)
- Master-Theorem: T(n) = O(n log n)
- Vorverarbeitung: O(n log n)
- Gesamtaufwand: O(n log n)

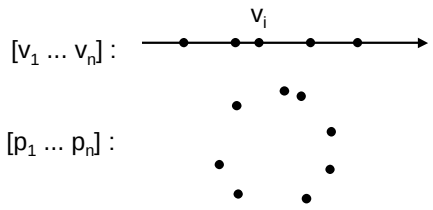
Optimaler Algorithmus

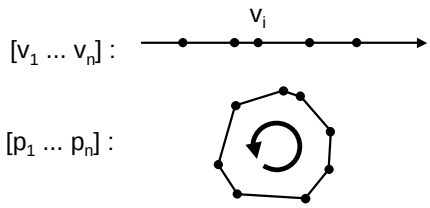
- Zur Erinnerung:
  Der optimale Sortieralgorithmus hat O(n log n) Aufwand *(beweisbar!)*
- Jeder ConvexHull-Algorithmus benutzt einen Sortierschritt ⇒ O(n log n)
- Gibt es einen schnelleren Algorithmus?

Optimaler Algorithmus

- Sei A ein optimaler CH-Algorithmus
- Verwende ihn zum Sortieren ...
- Eingabe : $[v_1 \ldots v_n]$ oBdA: $0 \le v_i \le 2\pi$
- Konvertierung: $v_i \rightarrow p_i = (\sin v_i, \cos v_i)$
- CH-Polygon ergibt Reihenfolge







Optimaler Algorithmus

Optimaler Algorithmus

Optimaler Algorithmus

Optimaler Algorithmus

Lower Bound

- Angenommen A sei ein CH-Algorithmus, der schneller als O(n log n) ist
- Dann liefert die Konvertierung $v_i \rightarrow p_i$ einen Sortieralgorithmus, der schneller als O(n log n) ist
- Dies ist ein Widerspruch zur Lower Bound für Sortieralgorithmen

---

(2.7) Geometrische Algorithmen

- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

Nachbarschaften

- Geg. Punktmenge p_1, ..., p_n
- Typische Problemstellungen :
  - geringster Abstand zwischen zwei p_i
  - k näheste Punkte zu p_i
  - k näheste Punkte zu (x,y)
  - alle Punkte innerhalb eines Kreises (x,y,r)
  - ...

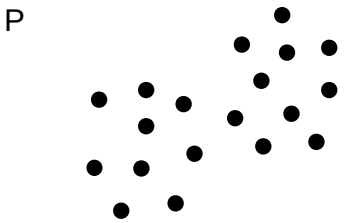
Maximale Punktdichte

- Geg. Punktmenge p_1, ..., p_n
- $d_{ij} = || p_j - p_i ||$
- finde $\min_{ij} \{ d_{ij} \}$
- triviale Lösung in $O(n^2)$
- schnellerer Algorithmus?

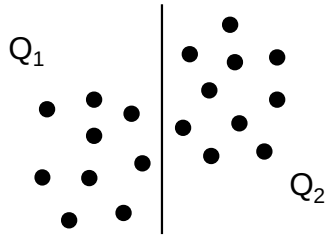
Maximale Punktdichte

- Versuche, bestimmte Punktpaare p_i, p_j ohne weitere Berechnung auszuschließen
- Ansatz: *Divide & Conquer*
  - teile die Menge $P = \{p_1 ... p_n\}$ in $Q_1 = \{p_1 ... p_{n/2}\}$ und $Q_2 = \{p_{n/2+1} ... p_n\}$
  - minimaler Abstand innerhalb Q_1
  - minimaler Abstand innerhalb Q_2
  - minimaler Abstand zwischen Q_1 und Q_2

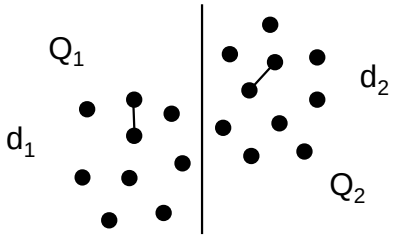
Divide & Conquer



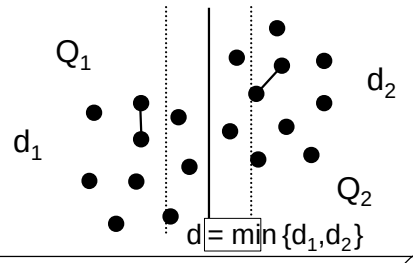
Divide & Conquer



Divide & Conquer



Divide & Conquer



1. Algorithmus

- MinDist(p[1 ... n])

  sort p[i] by increasing x-coordinate

  d = MinDistRec(p[1 ... n]);

  return d;

1. Algorithmus

MinDistRec(p[1 ... n])
  d_1 = MinDistRec(p[1 ... $\lfloor n/2 \rfloor$]);
  d_2 = MinDistRec(p[$\lfloor n/2 \rfloor$+1 ... n]);
  d = min { d_1, d_2 };
  x = (p[$\lfloor n/2 \rfloor$].x + p[$\lfloor n/2 \rfloor$+1].x) / 2;
  Q_1= select p[i ≤ $\lfloor n/2 \rfloor$] where |p[i].x - x| ≤ d
  Q_2= select p[i > $\lfloor n/2 \rfloor$] where |p[i].x - x| ≤ d
  d_3 = MinDistCross(Q_1,Q_2);
  return min { d,d_3 };

1. Algorithmus

MinDistRec(p[1 ... n])
  d_1 = MinDistRec(p[1 ... $\lfloor n/2 \rfloor$]);
  d_2 = MinDistRec(p[$\lfloor n/2 \rfloor$+1 ... n]);
  d = min { d_1, d_2 };
  x = (p[$\lfloor n/2 \rfloor$].x + p[$\lfloor n/2 \rfloor$+1].x) / 2;
  Q_1= select p[i ≤ $\lfloor n/2 \rfloor$] where |p[i].x - x| ≤ d
  Q_2= select p[i > $\lfloor n/2 \rfloor$] where |p[i].x - x| ≤ d
  d_3 = MinDistCross(Q_1,Q_2);
  return min { d,d_3 };

Q_1 , Q_2 können alle n Elemente enthalten

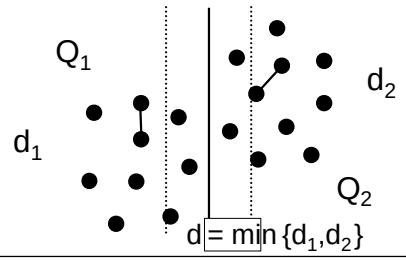
Aufwandsabschätzung

- $T(n) = 2\,T(n/2) + O(n^2)$
- $T(n) = O(n^2)$
- ... nicht besser als die triviale Lösung

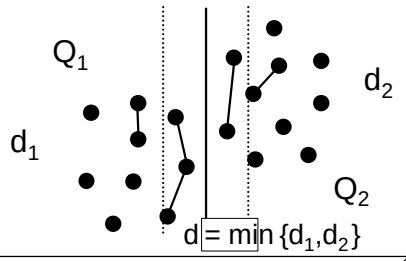
2. Algorithmus

- Die Kandidaten für minimale Distanzen in der Menge Q können weiter eingeschränkt werden ...

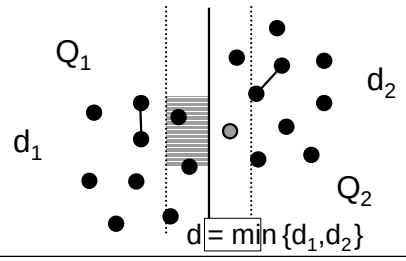
Divide & Conquer



Divide & Conquer



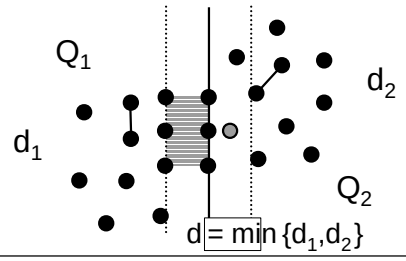
Divide & Conquer



2. Algorithmus

- Die Kandidaten für minimale Distanzen in der Menge Q können weiter eingeschränkt werden ...
- ... jeder Punkt muß mit maximal ***sechs*** Kandidaten verglichen werden

Divide & Conquer



2. Algorithmus

```
MinDistRec(p[1 ... n])
    $d_1$ = MinDistRec(p[1 ... ⌊n/2⌋]);
    $d_2$ = MinDistRec(p[⌊n/2⌋+1 ... n]);
    d  = min { $d_1$, $d_2$ };
    x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
    $Q_1$= select p[ i ≤ ⌊n/2⌋] where |p[ i ].x - x| ≤ d
    $Q_2$= select p[ i > ⌊n/2⌋] where |p[ i ].x - x| ≤ d
    sort $Q_1$, $Q_2$ by increasing y-coordinate
    $d_3$ = MinDistRestrict($Q_1$, $Q_2$, d);
    return min { d,$d_3$ };
```

2. Algorithmus

```
MinDistRestrict(q[1 ... k], q‘[1 ... k‘], d)
  dist = d;   l = 1;   r = 1;
  for (i = 1 ; i ≤ k ; i++)
      while (q‘[l].y < q[ i ] - d ∧ l < k‘) l++;
      while (q‘[r].y < q[ i ] + d ∧ r ≤ k‘) r++;
      for (j = l ; j < r ; j++)
          dist = min { dist, dist(q[ i ],q‘[ j ]) };
  return dist;
```


2. Algorithmus

```
MinDistRestrict(q[1 ... k], q‘[1 ... k‘], d)
  dist = d;   l = 1;   r = 1;
  for (i = 1 ; i ≤ k ; i++)                          } O(n)
      while (q‘[l].y < q[ i ] - d ∧ l < k‘) l++;      }
      while (q‘[r].y < q[ i ] + d ∧ r ≤ k‘) r++;      } O(1)
      for (j = l ; j < r ; j++)                      }
          dist = min { dist, dist(q[ i ],q‘[ j ]) }; } O(1)
  return dist;
```


2. Algorithmus

```
MinDistRec(p[1 ... n])
  d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
  d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
  d  = min { d1, d2 };
  x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
  Q1 = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d   } O(n)
  Q2 = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d   }
  sort Q1, Q2 by increasing y-coordinate                 } O(n log n)
  d3 = MinDistRestrict(Q1, Q2, d);                       } O(n)
  return min { d,d3 };
```


Aufwandsabschätzung

- $T(n) = 2\,T(n/2) + O(n) + O(n \log n)$
  $= 2\,T(n/2) + O(n \log n)$
- $T(n) = O(n \log^2 n)$

3. Algorithmus

- Beobachtung: der MinDist-Algorithmus hat dieselbe Struktur wie *Merge-Sort*
  - Splitting in zwei gleichgroße Teile
  - Sortierung beim Zusammenfügen
- Merging von sortierten Folgen kostet nur O(n)
- MinDistRec() gibt nach der Y-Koordinate sortierte Folgen zurück ...

3. Algorithmus

```
MinDistRec(p[1 ... n])
  d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
  d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
  d  = min { d1, d2 };
  x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
  Q1 = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
  Q2 = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
  merge pre-sorted lists Q1, Q2
  d3 = MinDistRestrict(Q1, Q2, d);
  merge pre-sorted lists p[ i ≤ ⌊n/2⌋ ], p[ i > ⌊n/2⌋ ]
  return min { d,d3 };
```

3. Algorithmus

```
MinDistRec(p[1 ... n])
   d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
   d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
   d  = min { d1, d2 };
   x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
   Q1= select p[ i ≤ ⌊n/2⌋] where |p[ i ].x - x| ≤ d   } O(n)
   Q2= select p[ i > ⌊n/2⌋] where |p[ i ].x - x| ≤ d   }
   merge pre-sorted lists Q1, Q2                       } O(n)
   d3 = MinDistRestrict(Q1, Q2, d);                    } O(n)
   merge pre-sorted lists p[ i ≤ ⌊n/2⌋], p[ i > ⌊n/2⌋] } O(n)
   return min { d,d3 };
```


Aufwandsabschätzung

- $T(n) = 2\,T(n/2) + O(n)$
- $T(n) = O(n \log n)$

(2.7) Geometrische Algorithmen

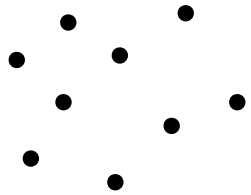
- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

Graham‘s Scan

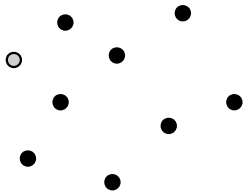
- Gift Wrapping führt zu viele redundante Berechnungen durch
- Bei der einfachen Generierung von Polygonen genügt ein einziger globaler Sortierungschritt
- ⇒ entferne die nicht-konvexen Ecken aus dem einfachen Polygon

---

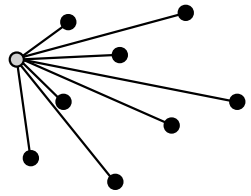
Graham‘s Scan



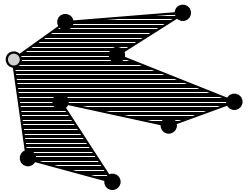
Graham‘s Scan



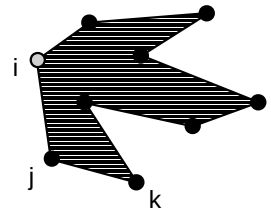
Graham‘s Scan



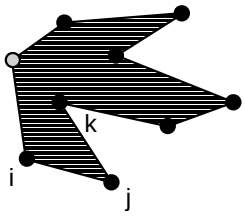
Graham‘s Scan



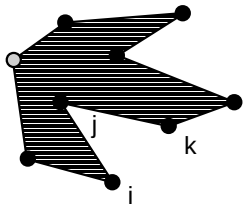
Graham‘s Scan



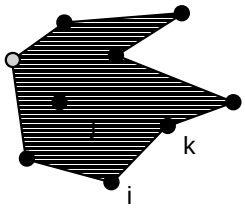
Graham‘s Scan



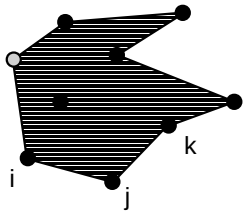
Graham‘s Scan



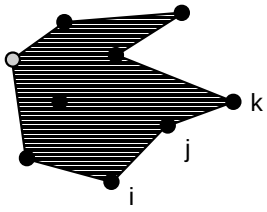
Graham‘s Scan



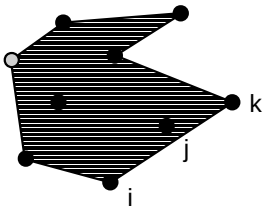
Graham‘s Scan



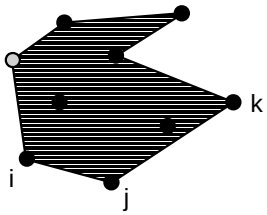
Graham‘s Scan



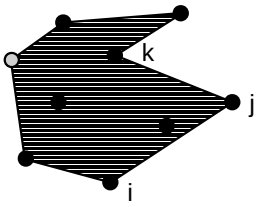
Graham‘s Scan



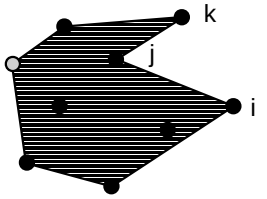
Graham‘s Scan



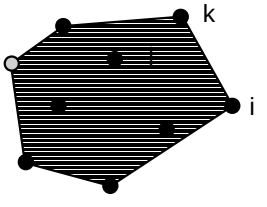
Graham‘s Scan



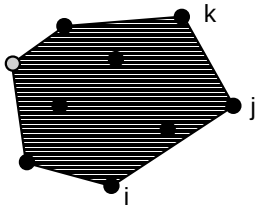
Graham‘s Scan



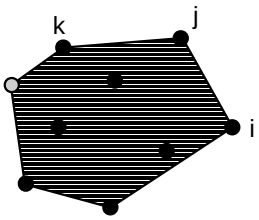
Graham‘s Scan



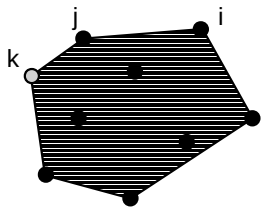
Graham‘s Scan



Graham‘s Scan



Graham‘s Scan

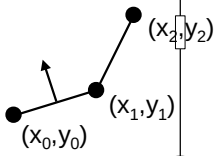


Konvexitäts-Test

$(y_0-y_1)*x_2+(x_1-x_0)*y_2 \geq (y_0-y_1)*x_0+(x_1-x_0)*y_0$

$(y_0-y_1)*(x_2-x_0)+(x_1-x_0)*(y_2-y_0) \geq 0$

$$\det\begin{pmatrix} x_1-x_0 & x_2-x_1 \\ y_1-y_0 & y_2-y_1 \end{pmatrix} \geq 0$$



Aufwandsabschätzung

- **Graham(Pts[1..n])**
  - **p = left-most point Pts[i];** } O(n)
  - **CH = Create(); CH.Add(p);**
  - **CH.Add(Pts[i] sorted by the angle of line(p,Pts[i]) to the x-axis);** } O(n log n)
  - **i = p; j = CH.next(i); k = CH.next(j);**
  - **while (k ≠ p)** } O(n)
    - **if (convex(i,j,k))** } O(m)
      - **i = j; j = k; k = CH.next(k);**
    - **else**
      - **CH.remove(j);**
      - **j = i; i = CH.prev(i);** } O(n-m)

Divide & Conquer

- Die konvexe Hülle von drei Punkten (in allg. Lage) ist ein Dreieck
- Teile die Gesamtmenge der p_i in zwei Teilmengen mit ***disjunkter*** konvexer Hülle
- Vereinige die konvexen Hüllen der Teilmengen

Divide & Conquer

- ConvexHull(P[1...n])

  sort P[ ] by increasing x-coordinate

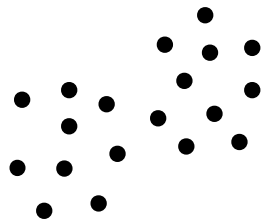
  remove interior points if ≥ 3 points have the same x-coordinate

  ConvexHullRec(P[1...n])

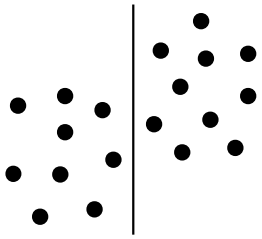
Divide & Conquer

- ConvexHullRec(P[1...n])
  - $Q_1 = P[1...\lfloor n/2 \rfloor]$;
  - $Q_2 = P[\lfloor n/2 \rfloor+1...n]$;
  - $[q_1 \dots q_m] = \text{ConvexHullRec}(Q_1)$;
  - $[q'_1 \dots q'_{m'}] = \text{ConvexHullRec}(Q_2)$;
  - find upper supporting line (q_i,q'_j)
  - find lower supporting line (q_k,q'_l)
  - return $[q_i \dots q_k,q'_l \dots q'_j]$;

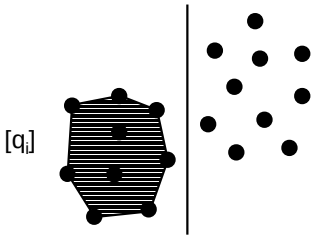
Divide & Conquer



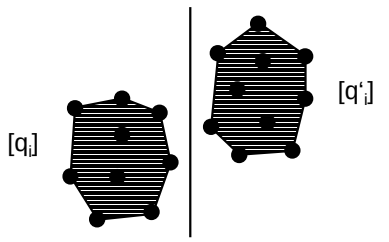
Divide & Conquer



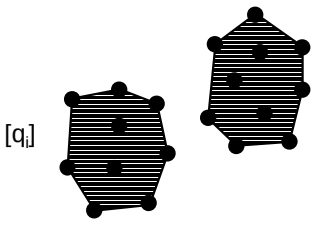
Divide & Conquer



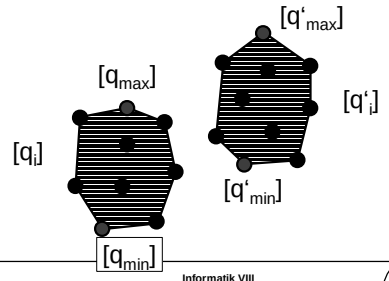
Divide & Conquer



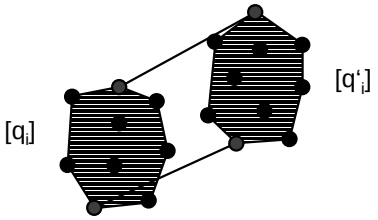
Divide & Conquer



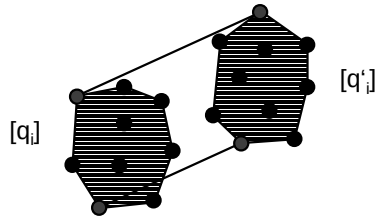
Divide & Conquer



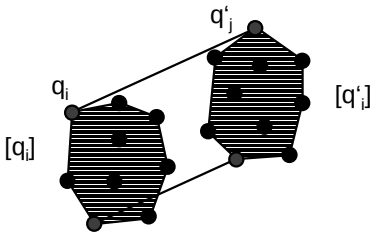
Divide & Conquer



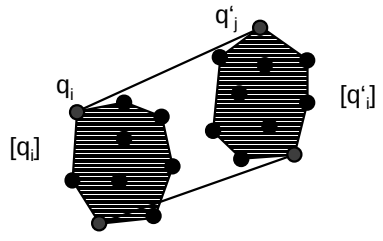
Divide & Conquer



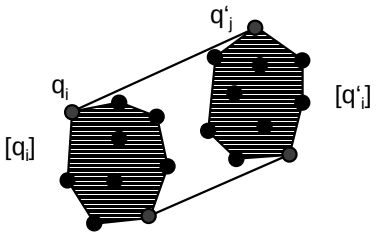
Divide & Conquer



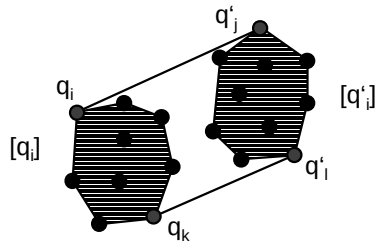
Divide & Conquer



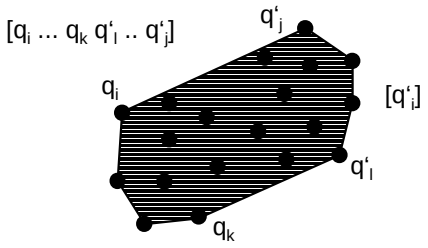
Divide & Conquer



Divide & Conquer



Divide & Conquer



Aufwandsabschätzung

ConvexHull(P[1...n])

sort P[ ] by increasing x-coordinate } O(n log n)

remove interior points if ≥ 3 points have the same x-coordinate } O(n)

ConvexHullRec(P[1...n])

Aufwandsabschätzung

ConvexHullRec(P[1...n])

$Q_1 = P[1...\lfloor n/2 \rfloor]$; } O(1)

$Q_2 = P[\lfloor n/2 \rfloor+1...n]$;

$[q_1 ... q_m] = \text{ConvexHullRec}(Q_1)$;

$[q'_1 ... q'_{m'}] = \text{ConvexHullRec}(Q_2)$;

find upper supporting line (q_i,q'_j) } O(n)

find lower supporting line (q_k,q'_l)

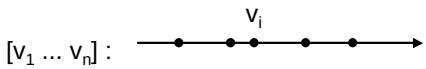
return $[q_i ... q_k,q'_l ... q'_j]$;

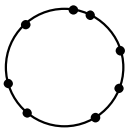
Aufwandsabschätzung

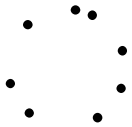
- T(n) = 2 T(n/2) + O(n)
- Master-Theorem: T(n) = O(n log n)
- Vorverarbeitung: O(n log n)
- Gesamtaufwand: O(n log n)

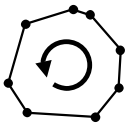
Optimaler Algorithmus

- Zur Erinnerung:
  Der optimale Sortieralgorithmus hat O(n log n) Aufwand *(beweisbar!)*
- Jeder ConvexHull-Algorithmus benutzt einen Sortierschritt ⇒ O(n log n)
- Gibt es einen schnelleren Algorithmus?









Optimaler Algorithmus

- Sei A ein optimaler CH-Algorithmus
- Verwende ihn zum Sortieren ...
- Eingabe : $[v_1 \dots v_n]$ oBdA: $0 \leq v_i \leq 2\pi$
- Konvertierung: $v_i \rightarrow p_i = (\sin v_i , \cos v_i)$
- CH-Polygon ergibt Reihenfolge

Optimaler Algorithmus

$[v_1 \dots v_n]$:

Optimaler Algorithmus

$[v_1 \dots v_n]$:

$[p_1 \dots p_n]$:

Optimaler Algorithmus

$[v_1 \dots v_n]$:

$[p_1 \dots p_n]$:

Optimaler Algorithmus

$[v_1 \dots v_n]$:

$[p_1 \dots p_n]$:

Lower Bound

- Angenommen A sei ein CH-Algorithmus, der schneller als O(n log n) ist
- Dann liefert die Konvertierung $v_i \rightarrow p_i$ einen Sortieralgorithmus, der schneller als O(n log n) ist
- Dies ist ein Widerspruch zur Lower Bound für Sortieralgorithmen

(2.7) Geometrische Algorithmen

- Polygone
- Inside-Test
- Konvexe Hüllen
- Nachbarschaften
- Freiformkurven (Polynome)

Nachbarschaften

- Geg. Punktmenge p_1, ..., p_n
- Typische Problemstellungen :
  - geringster Abstand zwischen zwei p_i
  - k näheste Punkte zu p_i
  - k näheste Punkte zu (x,y)
  - alle Punkte innerhalb eines Kreises (x,y,r)
  - ...

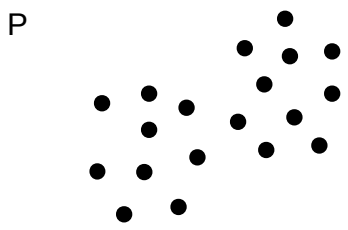
Maximale Punktdichte

- Geg. Punktmenge p_1, ..., p_n
- $d_{ij} = || p_j - p_i ||$
- finde $\min_{ij} \{ d_{ij} \}$
- triviale Lösung in $O(n^2)$
- schnellerer Algorithmus?

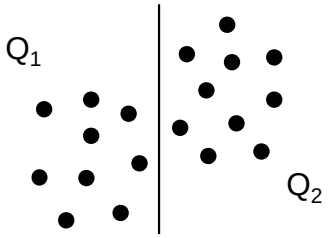
Maximale Punktdichte

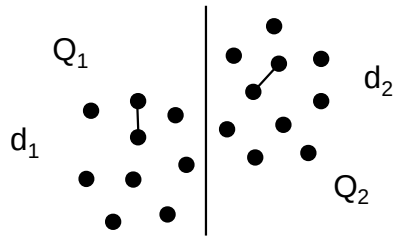
- Versuche, bestimmte Punktpaare p_i, p_j ohne weitere Berechnung auszuschließen
- Ansatz: *Divide & Conquer*
  - teile die Menge $P = \{p_1 ... p_n\}$ in $Q_1 = \{p_1 ... p_{n/2}\}$ und $Q_2 = \{p_{n/2+1} ... p_n\}$
  - minimaler Abstand innerhalb Q_1
  - minimaler Abstand innerhalb Q_2
  - minimaler Abstand zwischen Q_1 und Q_2

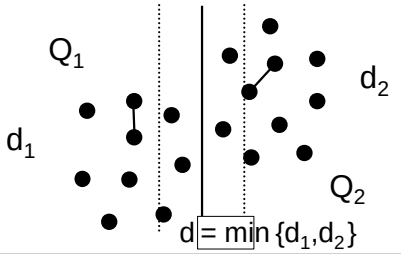
Divide & Conquer

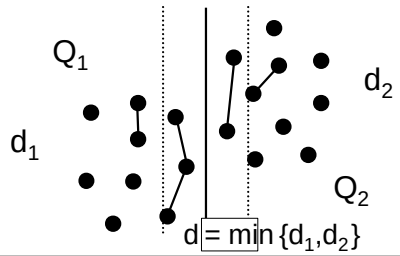


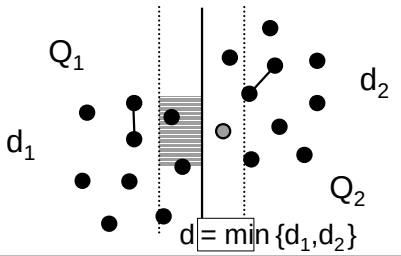
Divide & Conquer

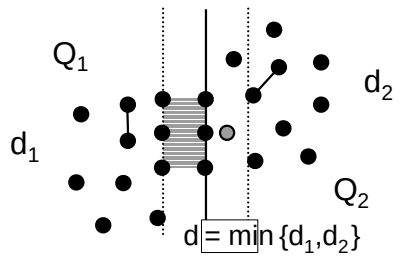












Divide & Conquer

Divide & Conquer

$d = \min\{d_1,d_2\}$

Divide & Conquer

$d = \min\{d_1,d_2\}$

Divide & Conquer

$d = \min\{d_1,d_2\}$

Divide & Conquer

$d = \min\{d_1,d_2\}$

2. Algorithmus

```
MinDistRec(p[1 ... n])
    $d_1$ = MinDistRec(p[1 ... ⌊n/2⌋]);
    $d_2$ = MinDistRec(p[⌊n/2⌋+1 ... n]);
    d  = min { $d_1$, $d_2$ };
    x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
    $Q_1$ = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
    $Q_2$ = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
    sort $Q_1$, $Q_2$ by increasing y-coordinate
    $d_3$ = MinDistRestrict($Q_1$, $Q_2$, d);
    return min { d,$d_3$ };
```

2. Algorithmus

```
MinDistRestrict(q[1 ... k], q‘[1 ... k‘], d)
  dist = d;   l = 1;   r = 1;
  for (i = 1 ; i ≤ k ; i++)
      while (q‘[l].y < q[ i ] - d ∧ l < k‘) l++;
      while (q‘[r].y < q[ i ] + d ∧ r ≤ k‘) r++;
      for (j = l ; j < r ; j++)
          dist = min { dist, dist(q[ i ],q‘[ j ]) };
  return dist;
```


2. Algorithmus

```
MinDistRestrict(q[1 ... k], q‘[1 ... k‘], d)
  dist = d;   l = 1;   r = 1;
  for (i = 1 ; i ≤ k ; i++)                          } O(n)
      while (q‘[l].y < q[ i ] - d ∧ l < k‘) l++;      }
      while (q‘[r].y < q[ i ] + d ∧ r ≤ k‘) r++;      } O(1)
      for (j = l ; j < r ; j++)                      }
          dist = min { dist, dist(q[ i ],q‘[ j ]) }; } O(1)
  return dist;
```


2. Algorithmus

```
MinDistRec(p[1 ... n])
  d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
  d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
  d  = min { d1, d2 };
  x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
  Q1 = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d   } O(n)
  Q2 = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d   }
  sort Q1, Q2 by increasing y-coordinate                 } O(n log n)
  d3 = MinDistRestrict(Q1, Q2, d);                       } O(n)
  return min { d,d3 };
```


Aufwandsabschätzung

- $T(n) = 2\,T(n/2) + O(n) + O(n \log n)$
  $= 2\,T(n/2) + O(n \log n)$
- $T(n) = O(n \log^2 n)$

3. Algorithmus

- Beobachtung: der MinDist-Algorithmus hat dieselbe Struktur wie *Merge-Sort*
  - Splitting in zwei gleichgroße Teile
  - Sortierung beim Zusammenfügen
- Merging von sortierten Folgen kostet nur O(n)
- MinDistRec() gibt nach der Y-Koordinate sortierte Folgen zurück ...

3. Algorithmus

```
MinDistRec(p[1 ... n])
  d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
  d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
  d  = min { d1, d2 };
  x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
  Q1 = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
  Q2 = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d
  d3 = MinDistRestrict(Q1, Q2, d);
  merge pre-sorted lists p[ i ≤ ⌊n/2⌋ ], p[ i > ⌊n/2⌋ ]
  return min { d,d3 };
```

3. Algorithmus

```
MinDistRec(p[1 ... n])
   d1 = MinDistRec(p[1 ... ⌊n/2⌋]);
   d2 = MinDistRec(p[⌊n/2⌋+1 ... n]);
   d  = min { d1, d2 };
   x  = (p[⌊n/2⌋].x + p[⌊n/2⌋+1].x) / 2;
   Q1 = select p[ i ≤ ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d }
   Q2 = select p[ i > ⌊n/2⌋ ] where |p[ i ].x - x| ≤ d } O(n)
   d3 = MinDistRestrict(Q1, Q2, d); } O(n)
   merge pre-sorted lists p[ i ≤ ⌊n/2⌋ ], p[ i > ⌊n/2⌋ ] } O(n)
   return min { d,d3 };
```


Aufwandsabschätzung

- $T(n) = 2\,T(n/2) + O(n)$
- $T(n) = O(n \log n)$

Weitester Abstand

- Geg. Punktmenge $p_1, ..., p_n$
- $d_{ij} = || p_j - p_i ||$
- finde $\max_{ij} \{ d_{ij} \}$
- *„Durchmesser der Punktmenge“*
- triviale Lösung in $O(n^2)$

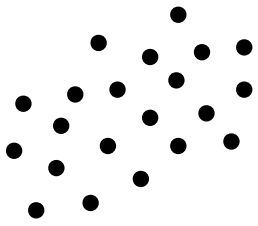
Weitester Abstand

- Versuche wieder, bestimmte Punktpaare p_i, p_j auszuschließen ...

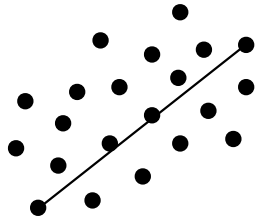
Weitester Abstand

- Versuche wieder, bestimmte Punktpaare p_i, p_j auszuschließen ...
- Der maximale Abstand ***muß*** zwischen Extremalpunkten auftreten
- Beschränke die Suche auf die konvexe Hülle

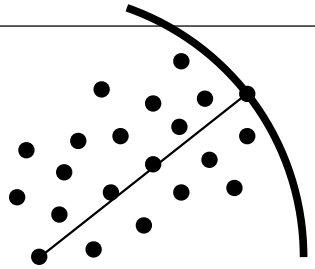
Weitester Abstand



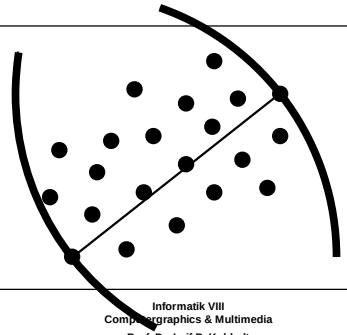
Weitester Abstand



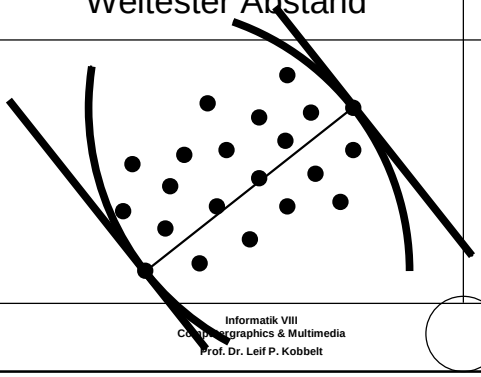
Weitester Abstand



Weitester Abstand



Weitester Abstand



Weitester Abstand

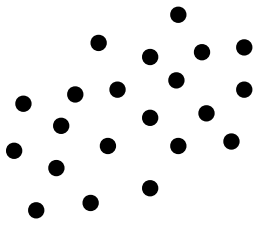
Geg. Punktmenge $p_1, ..., p_n$

(1) Berechne die konvexe Hülle

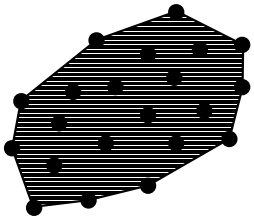
(2) Suche Extremalpunkte mit parallelen Supporting Lines

(3) Finde den maximalen Abstand zwischen diesen Kandidaten

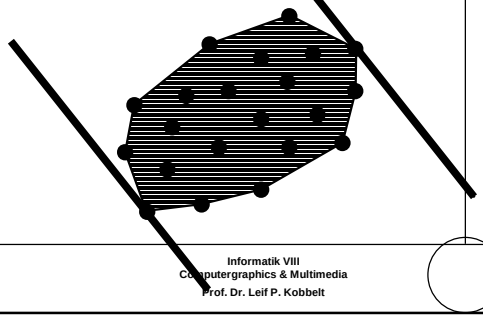
Weitester Abstand



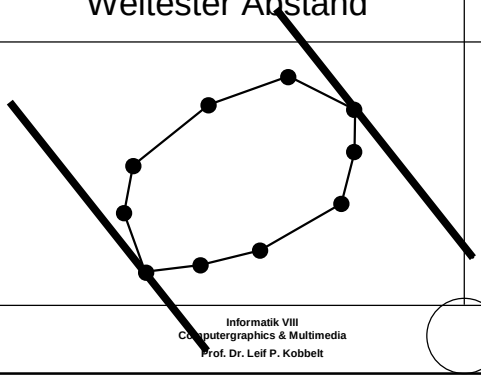
Weitester Abstand



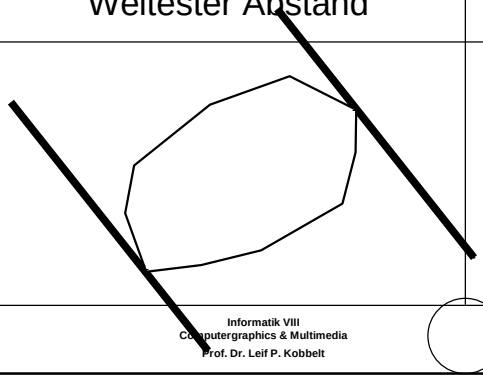
Weitester Abstand



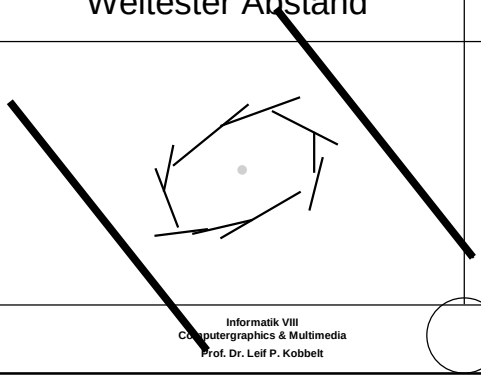
Weitester Abstand



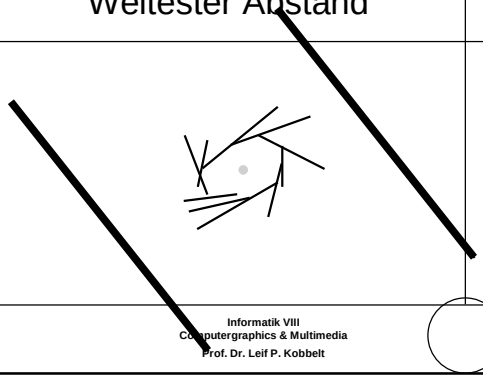
Weitester Abstand



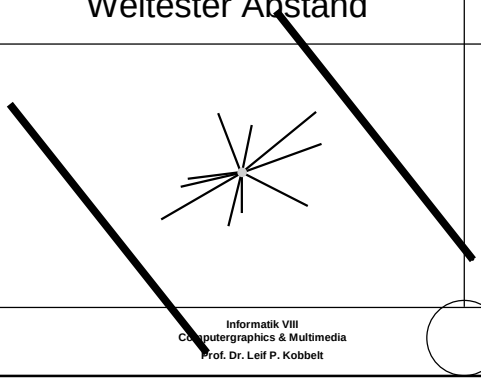
Weitester Abstand



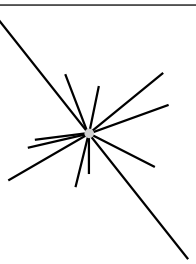
Weitester Abstand



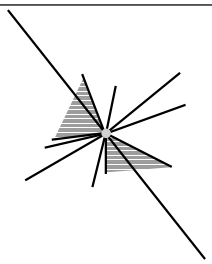
Weitester Abstand



Weitester Abstand



Weitester Abstand



Weitester Abstand

(1) Berechne die konvexe Hülle } O(n log n)

(2) Suche Extremalpunkte mit parallelen Supporting Lines
   a) die Kanten der konvexen Hülle bilden Sektoren
   b) gegenüberliegende Sektoren enthalten gemeinsame Gerade } O(n)

(3) Finde den maximalen Abstand zwischen diesen Kandidaten } O(n)

Nachbarschaften

- Geg. Punktmenge p_1, ..., p_n
- Typische Problemstellungen :
  - min. / max. Abstand zwischen zwei p_i
  - k näheste Punkte zu p_i
  - k näheste Punkte zu (x,y)
  - alle Punkte innerhalb eines Kreises (x,y,r)
  - ...

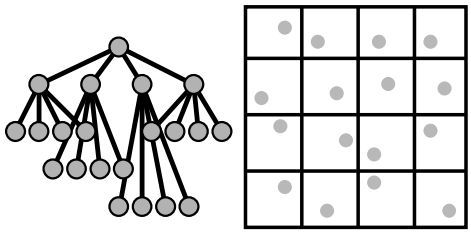
Näheste Punkte

- Beispiele:
  - Datenbanksuche mit Unsicherheit
  - Finde näheste Mobilfunkstation
  - Globale Beleuchtungssimulation
  - ...

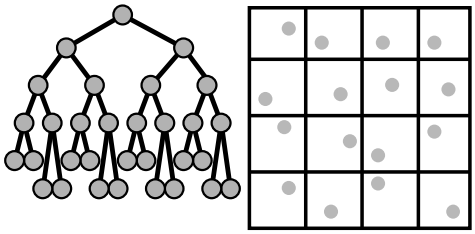
Näheste Punkte

- Geg. Punktmenge p_1, ..., p_n
- Anfrage: (x,y) [*hier:* IR^2, *allg.:* IR^k]
- Optimale Datenstruktur zum Suchen
  $\Rightarrow$ ***Bäume***
- Mehrdimensionale Schlüssel
  $\Rightarrow$ z.B. ***kD-Bäume***

Quadtree



kD-Bäume



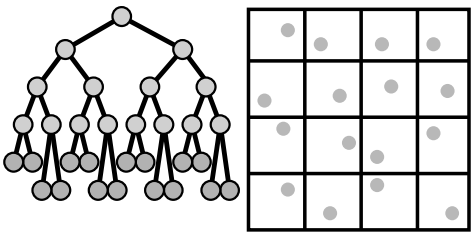
kD-Baum Suche

- In welcher Zelle liegt der Anfragepunkt
- In welchen Zellen muß nach dem nähesten Punkt gesucht werden
- Effizienter Auschluß weit entfernter Zellen

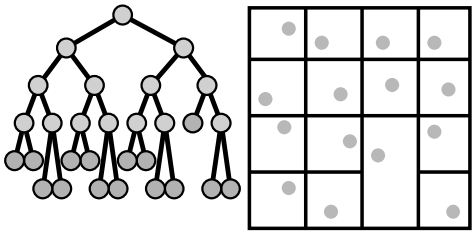
kD-Baum Suche

- Jedes kD-Baum ***Blatt*** enthält einen der gegebenen Punkte p_i
- Jeder ***innere*** kD-Baum Knoten enthält eine Ebene, die die Punkte im linken und rechten Teilbaum separiert.

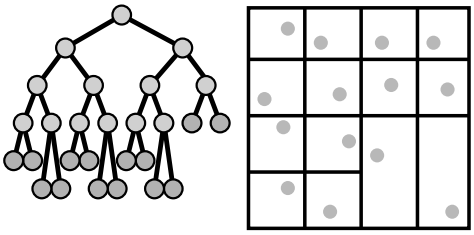
kD-Bäume



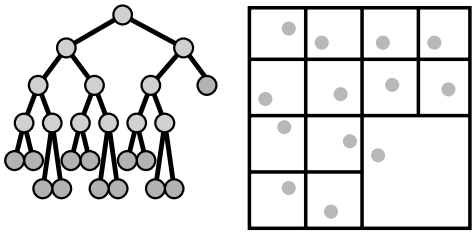
kD-Bäume



kD-Bäume



kD-Bäume



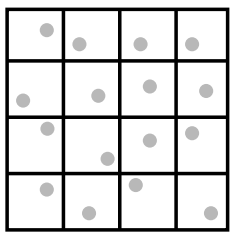
kD-Baum Suche

```
Traverse(p,N)
  if (N is a leaf-node)
      return [ || p - N.point ||, N.point ];
  else
      if (p lies in positive half-space of N)
          [dist,near] = Traverse(p,N.left);
      else
          [dist,near] = Traverse(p,N.right);
      return [dist,near]
```

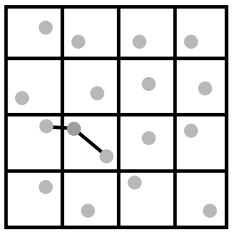

kD-Baum Suche

```
Traverse(p,N)
  if (N is a leaf-node)
      return [ || p - N.point ||, N.point ];
  else
      if (p * N.normal - N.offset ≥ 0)
          [dist,near] = Traverse(p,N.left);
      else
          [dist,near] = Traverse(p,N.right);
      return [dist,near]
```

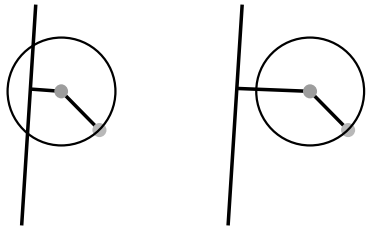

kD-Baum Suche



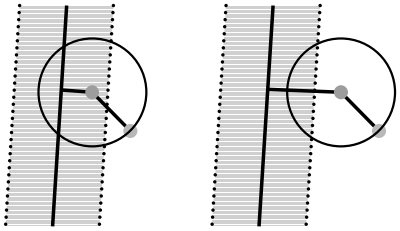
kD-Baum Suche



kD-Baum Suche



kD-Baum Suche



kD-Baum Suche

```
Traverse(p,dist,near,N)
  if (N is a leaf-node)
    if (|| p - N.point || < dist)
      return [ || p - N.point ||, N.point ];
    else
      return [ dist, near ];
  else
    if (p * N.normal - N.offset ≥ -dist)
      [dist,near] = Traverse(p,dist,near,N.left);
    if (p * N.normal - N.offset ≤  dist)
      [dist,near] = Traverse(p,dist,near,N.right);
    return [dist,near]
```


Aufwandsabschätzung

- Bei regelmäßig verteilten Datenpunkten ist der erwartete Aufwand O(log n)
- Der Algorithmus funktioniert auch für allgemeinere BSP-Bäume, bei denen in jedem Knoten eine ***beliebige*** separierende Ebene definiert wird.

k näheste Punkte

- Geg. Punktmenge p_1, ..., p_n
- Anfrage: (x,y) [*hier:* IR^2, *allg.:* IR^k]
- Suche die k besten Kandidaten
- Verwalte die aktuell k Besten in einem Heap
- Neue Kandidaten verdrängen die alten

k näheste Punkte

```
Nearest(p,neighbors,N)
  if (N is a leaf-node)
    if (|| p - N.point || < neighbors.top())
      neighbors.remove_top();
      neighbors.add(|| p - N.point ||, N.point);
    return neighbors;
  else
    if (p * N.normal - N.offset ≥ -neighbors.top())
      neighbors = Traverse(p,neighbors,N.left);
    if (p * N.normal - N.offset ≤  neighbors.top())
      neighbors = Traverse(p,neighbors,N.right);
    return neighbors;
```


Aufwandsabschätzung

- Fast der gleiche Aufwand wie bei der einfachen Suche:
- O(log n * log k)

Range Queries

- Bereichsabfragen
  - Suche alle Punkte in einem Kreis (x,y,r)
  - Suche alle Punkte in einem Intervall [a,b] x [c,d]
  - ...
- Einfache Modifikation des Suchalgorithmus

Range Queries

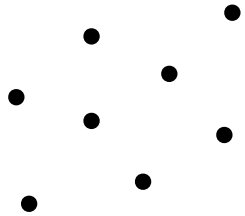
```
Traverse(p,maxdist,N)
  if (N is a leaf-node)
    if (|| p - N.point || < maxdist)
      output  N.point;
    return;
  else
    if (p * N.normal - N.offset ≥ -maxdist)
      Traverse(p,maxdist,N.left);
    if (p * N.normal - N.offset ≤  maxdist)
      Traverse(p,maxdist,N.right);
    return;
```

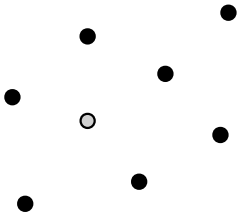

Abstandsdiagramme

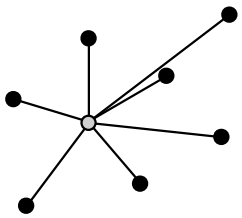
- Bei häufigen Nachbarschafts-Anfragen kann der Zugriff durch ***Vorberechnung*** von Abstandsdiagrammen beschleunigt werden
- Diese zerlegen den IR^n (hier den IR^2) in kleine Bereiche, für die die Nachbarschaften jeweils vorberechnet werden können

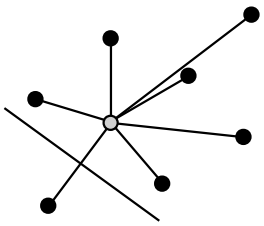
Voronoi-Gebiete

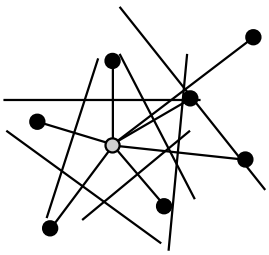
- Geg. Punktmenge $p_1, ..., p_n$
- Das Voronoi-Gebiet zu einem Punkt p_i enthält den Bereich des IR^2, der näher an p_i als an jedem anderen p_j liegt
- $V(p_i) = \{ q \mid \forall\, j \neq i : ||q - p_i|| < ||q - p_j|| \}$

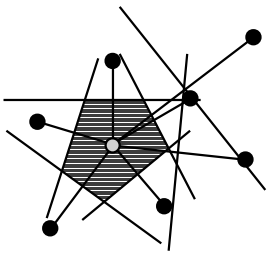












Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

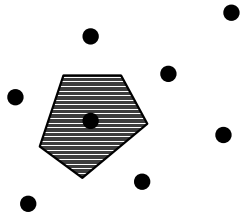
Voronoi-Gebiete

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

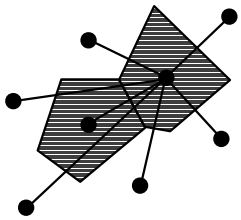
Voronoi-Gebiete

- Polygonale Gebiete
- Schnitt von Halbräumen
- Konvexe Gebiete

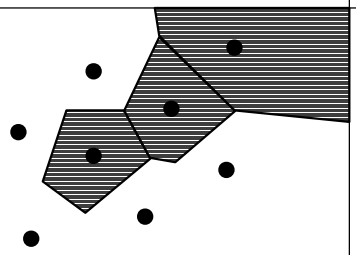
Voronoi-Diagramm



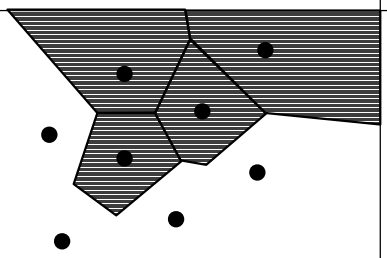
Voronoi-Diagramm



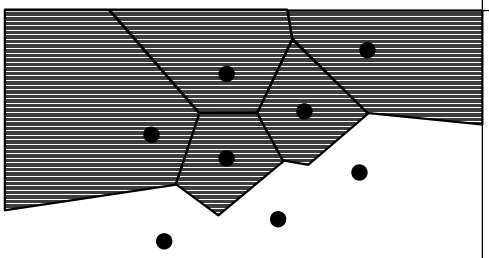
Voronoi-Diagramm



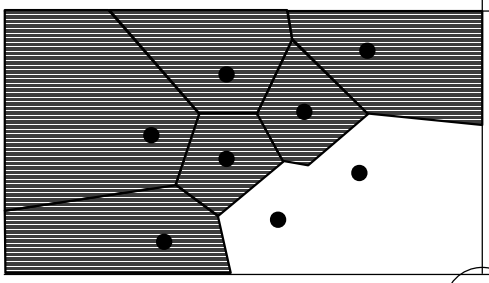
Voronoi-Diagramm



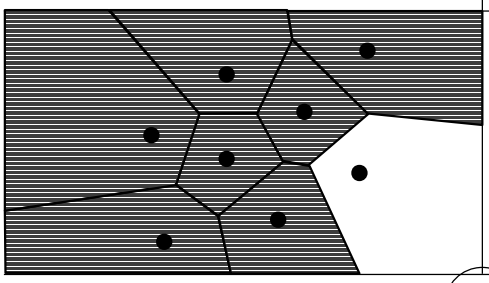
Voronoi-Diagramm



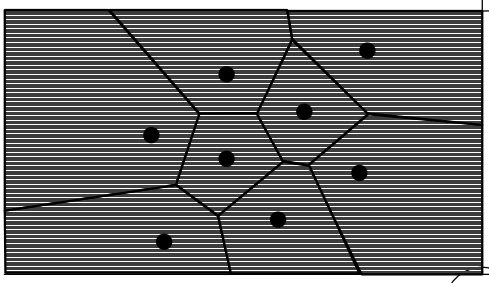
Voronoi-Diagramm



Voronoi-Diagramm



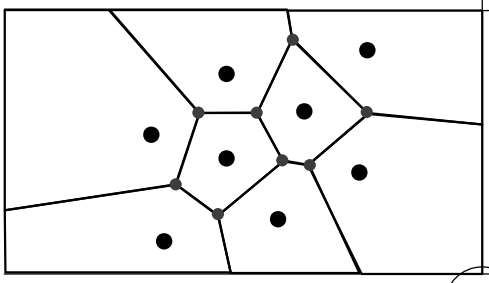
Voronoi-Diagramm



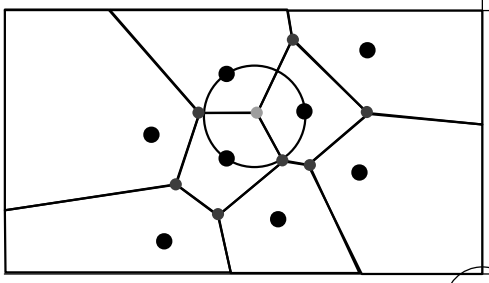
Voronoi-Diagramm

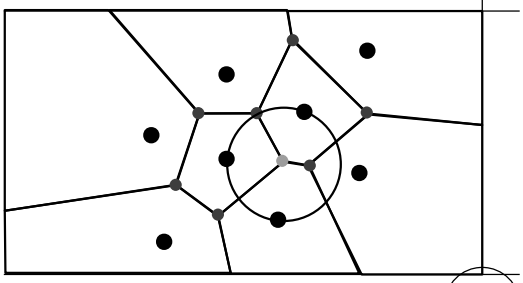
- Ein VD ist eine (planare) Graph-Struktur, die den IR^2 in konvexe Regionen unterteilt
- Die ***Voronoi-Kanten*** liegen auf den Mittelsenkrechten der Verbindungsstrecken
- Im allg. Treffen an einem Voronoi-Vertex genau 3 Kanten zusammen
- Voronoi-Vertices sind Umkreismittelpunkte

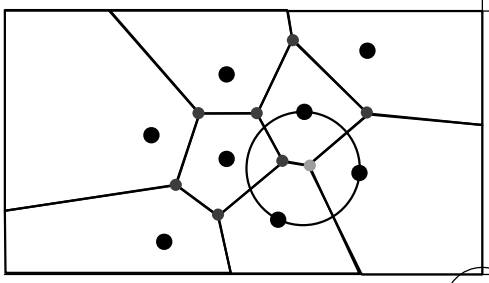
Voronoi-Diagramm

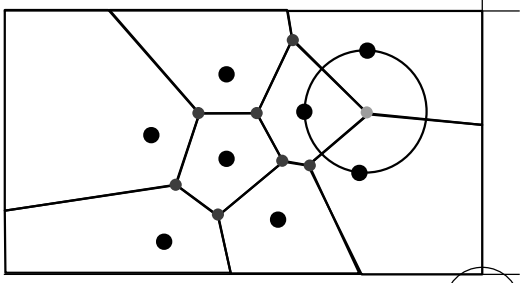


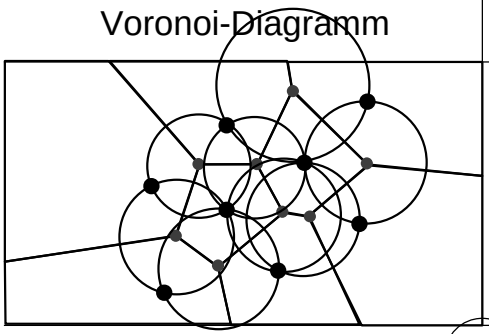
Voronoi-Diagramm

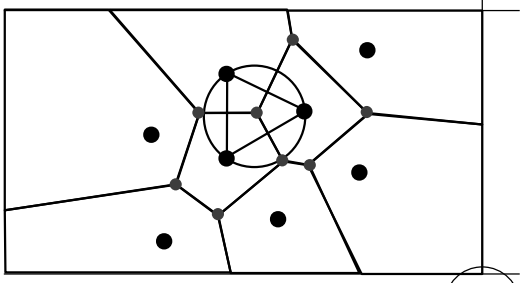


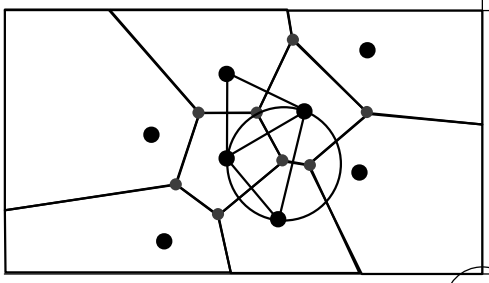












Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

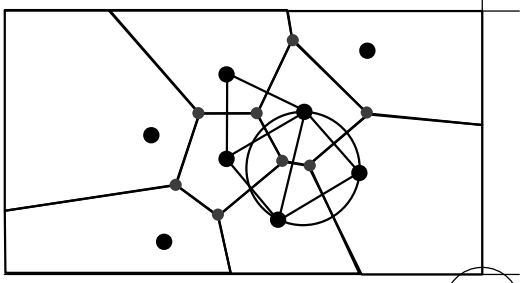
Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

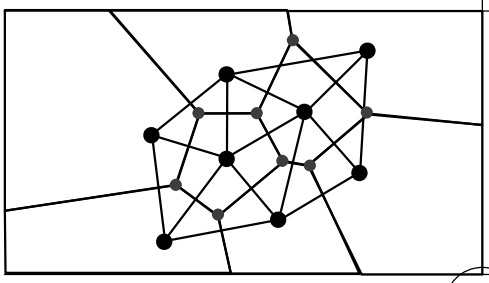
Voronoi-Diagramm

Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

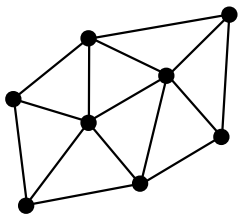
Voronoi-Diagramm



Voronoi-Diagramm



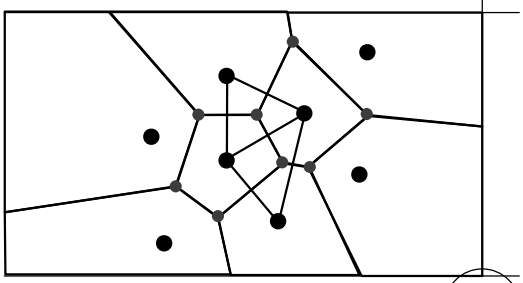
Delaunay-Triangulierung



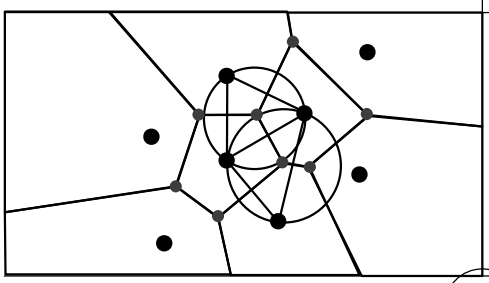
Delaunay-Triangulierung

- Eigenschaften (1):
  - Dualer Graph zum Voronoi-Diagramm
  - Zerlegt die konvexe Hülle der Punkte in disjunkte Dreiecke (mit den p_i als Ecken)
  - Die Region innerhalb eines Dreiecks ist einem der drei Eckpunkte am nächsten
  - Der Umkreis jedes Dreiecks enthält keinen weiteren Punkt p_j

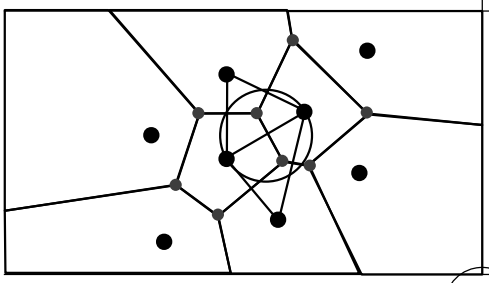
Voronoi-Diagramm



Voronoi-Diagramm



Voronoi-Diagramm



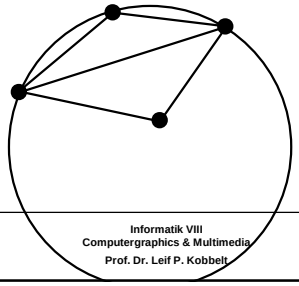
Delaunay-Triangulierung

- Eigenschaften(2):
  - Der Umkreis jeder Kante enthält keinen weiteren Punkt p_j
  - Die DT ist ***eindeutig*** definiert
  - ***Maximaler minimaler Innenwinkel***
  - Einfacher zu handhaben als VD, da keine allg. Graph-Struktur verarbeitet werden muß

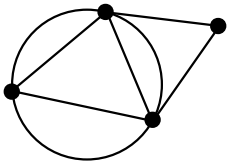
Delaunay-Triangulierung

- ***Inkrementeller*** Algorithmus zur Erzeugung …
  - Für nur drei Punkte p_1, p_2, p_3 trivial
  - Füge weitere Punkte p_4, ... ein und stelle die Delaunay-Eigenschaften wieder her

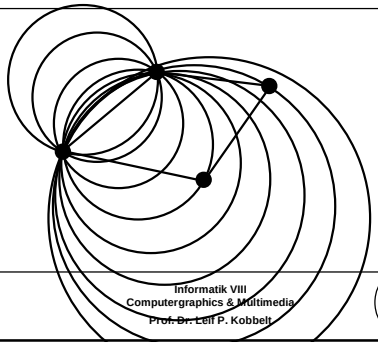
Edge-Flipping

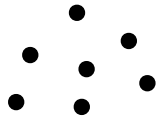


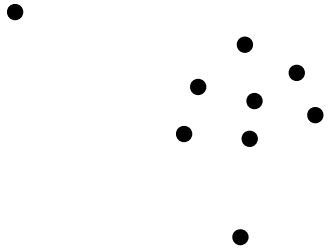
Edge-Flipping

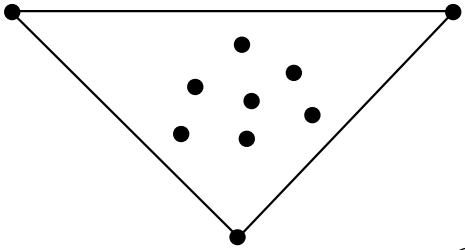


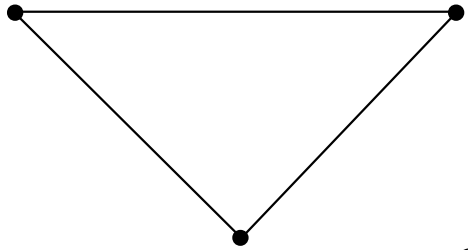
Edge-Flipping

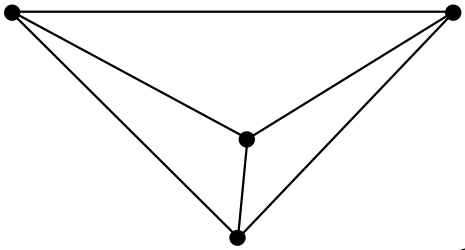












Inkrementeller Algorithmus

Geg: Punktmenge $p_1, p_2, ..., p_n$

Zur Vermeidung von Spezialfällen:

Ergänze drei zusätzliche Punkte q_1, q_2, q_3, so dass alle p_i innerhalb des Dreiecks $[q_1,q_2,q_3]$ liegen ⇒ Einfügen nur im Inneren

q_1, q_2, q_3 können leicht entfernt werden

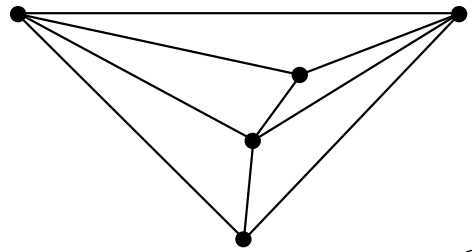
Inkrementeller Algorithmus

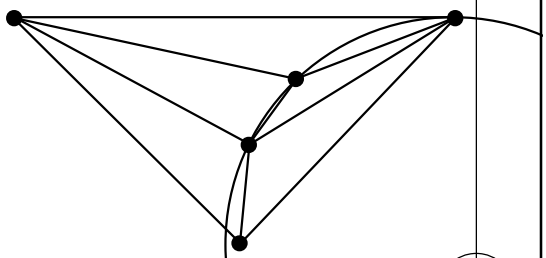
Inkrementeller Algorithmus

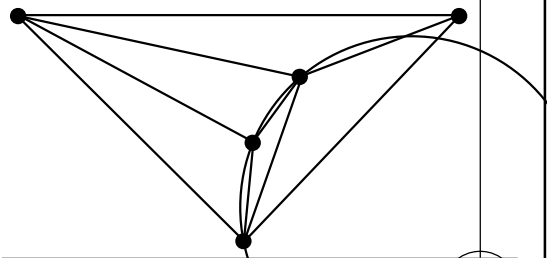
Inkrementeller Algorithmus

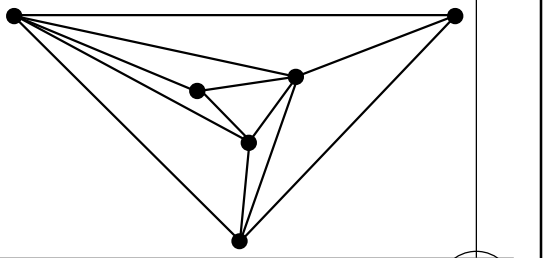
Inkrementeller Algorithmus

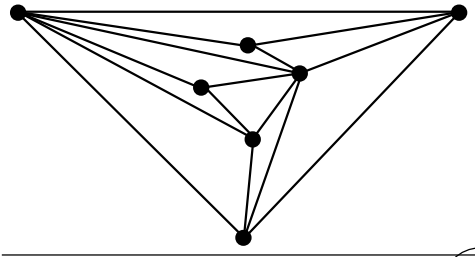
Inkrementeller Algorithmus

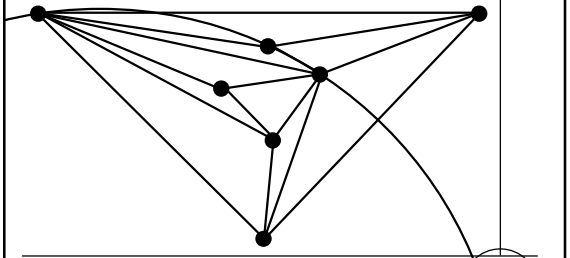












Inkrementeller Algorithmus

Inkrementeller Algorithmus

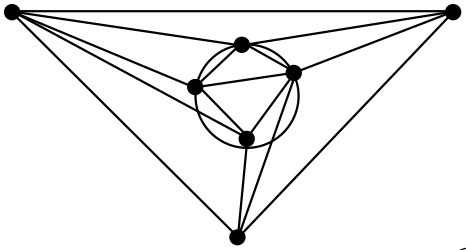
Inkrementeller Algorithmus

Inkrementeller Algorithmus

Inkrementeller Algorithmus

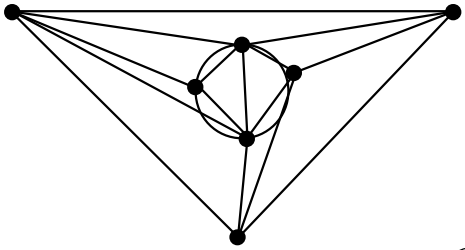
Inkrementeller Algorithmus

Inkrementeller Algorithmus



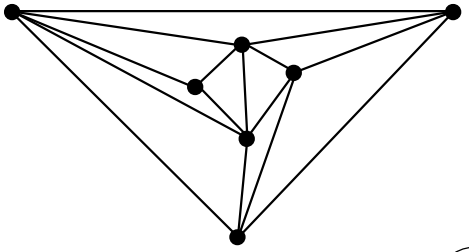
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Inkrementeller Algorithmus



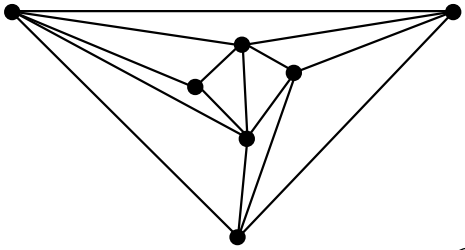
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Inkrementeller Algorithmus



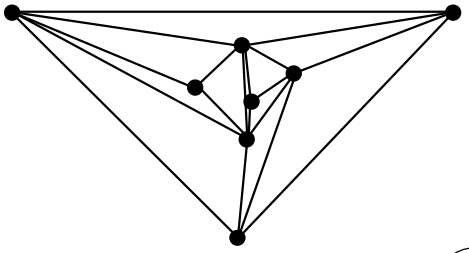
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Inkrementeller Algorithmus



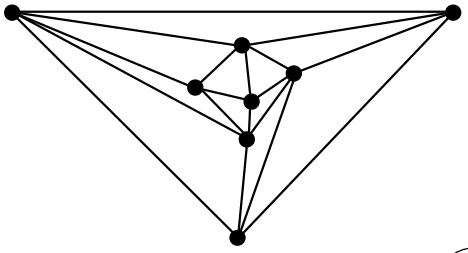
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Inkrementeller Algorithmus



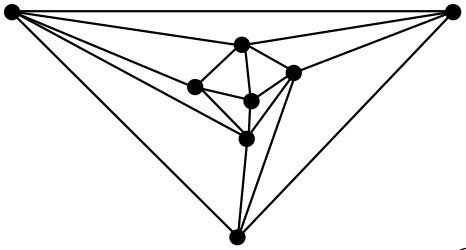
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Inkrementeller Algorithmus

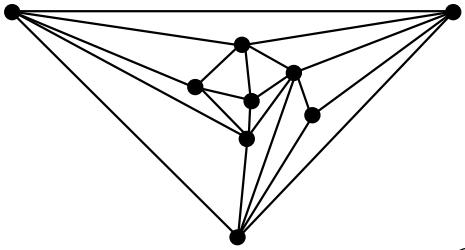


Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

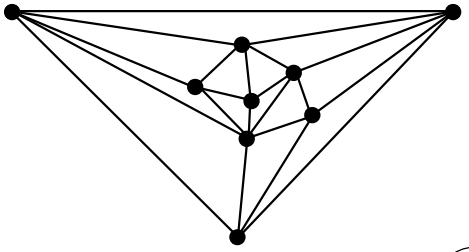
Inkrementeller Algorithmus



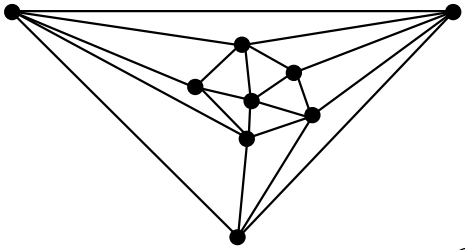
Inkrementeller Algorithmus



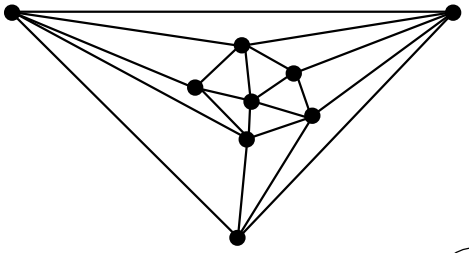
Inkrementeller Algorithmus



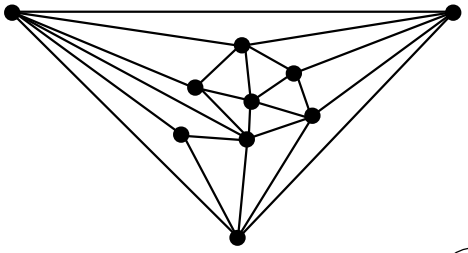
Inkrementeller Algorithmus



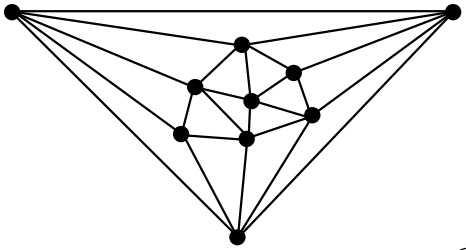
Inkrementeller Algorithmus



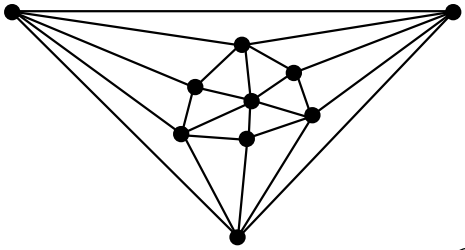
Inkrementeller Algorithmus



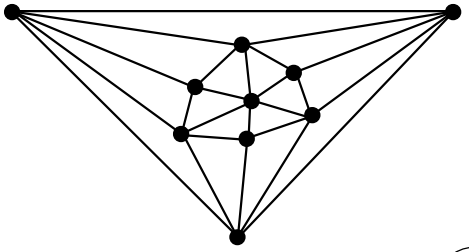
Inkrementeller Algorithmus



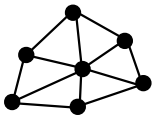
Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus

- Für jeden Punkt p_i $\}$ O(n)
  - Suche das entsprechende Dreieck $\}$ O(n)
  - Füge den neuen Punkt ein $\}$ O(1)
  - Flippe Kanten bis die Delaunay-Bedingungen wieder hergestellt sind. $\}$ O(k)

Aufwandsabschätzung

- Der maximale Knotengrad k (Valenz) kann normalerweise als beschränkt angenommen werden (zumindest als unabhängig von n)
- Gesamtaufwand $O(n^2)$
- Beschleunigung durch bessere Suche

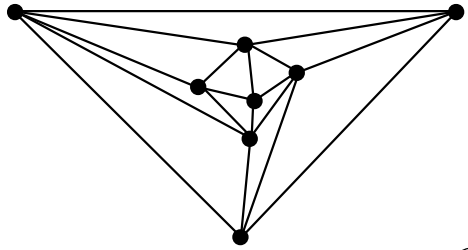
Beschleunigte Suche

- *Standard Ansatz:*
  Verwalte die Dreiecke in einem **Suchbaum**, um schneller auf die Elemente zuzugreifen
- Verbesserung: O(log n) statt O(n)

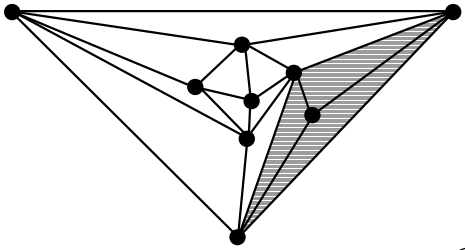
Beschleunigte Suche

- Am Anfang existiert nur ein Dreieck
- Bei jedem Einfügeschritt werden m Dreiecke entfernt und m+2 Dreiecke ergänzt

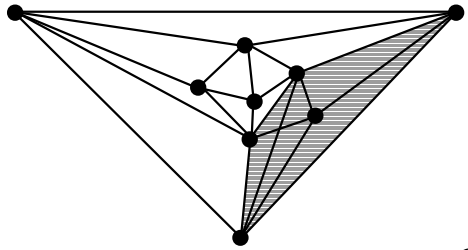
Inkrementeller Algorithmus



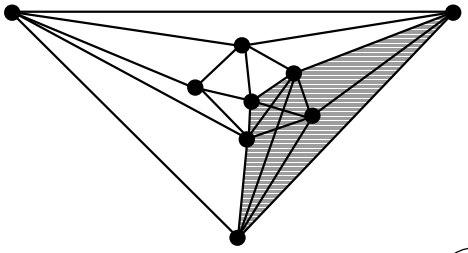
Inkrementeller Algorithmus



Inkrementeller Algorithmus



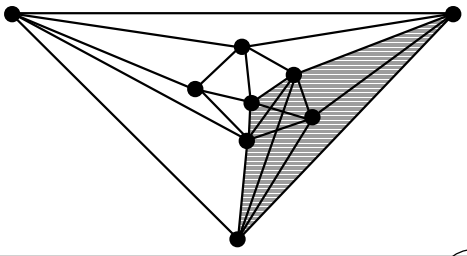
Inkrementeller Algorithmus



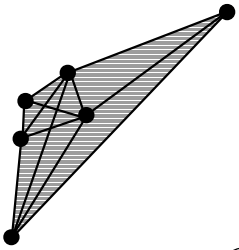
Beschleunigte Suche

- Am Anfang existiert nur ein Dreieck
- Bei jedem Einfügeschritt werden m Dreiecke entfernt und m+2 Dreiecke ergänzt
- $m+2 \le k$ ... die maximale Valenz
- $m+2 : m \ge c > 1$ ... minimaler Verfeinerungsfaktor

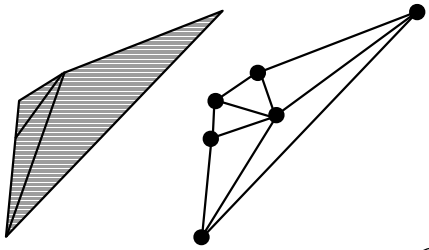
Inkrementeller Algorithmus



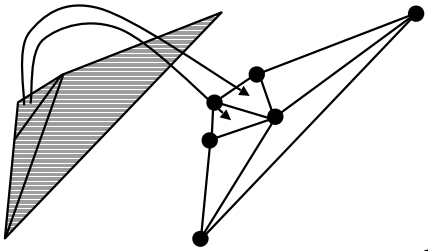
Inkrementeller Algorithmus



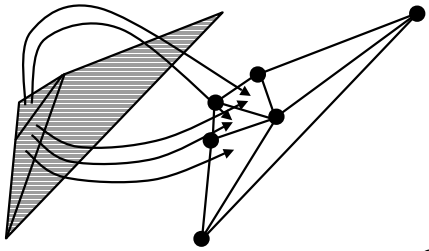
Inkrementeller Algorithmus



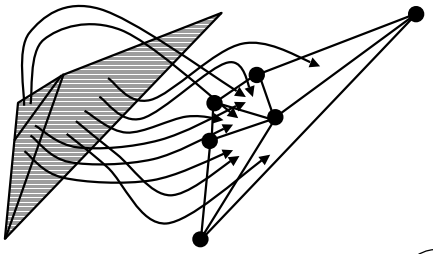
Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus



Suchbaum für Dreiecke

- Seien die Dreiecke, die im Laufe des Algorithmus generiert werden die Knoten des Suchbaumes
- Durch den minimalen Verfeinerungsfaktor $m+2 : m \geq c > 1$ ist garantiert, dass die Anzahl von Knoten von Level zu Level exponentiell steigt

Suchbaum für Dreiecke

- Obwohl diese Datenstruktur keinen (zyklenfreien) Baum darstellt, ergibt sich trotzdem ein gerichteter Graph, dessen Pfade maximal *logarithmische* Länge haben, wenn die Punkte hinreichend gleichmäßig verteilt sind (und die maximale Valenz beschränkt ist).
- Kosten:
  - Aufbau des Baumes O(n)
  - Suche im Baum O(log n)

Inkrementeller Algorithmus

- Für jeden Punkt p_i } O(n)
  - Suche das entsprechende Dreieck } O(log n)
  - Füge den neuen Punkt ein } O(1)
  - Flippe Kanten bis die Delaunay-Bedingungen wieder hergestellt sind. } O(k)

Gesamtaufwand: O(n log n)

Kurven-Konstruktion

- Polynome
- Interpolation
- Evaluation
- Bezier-Darstellung
- Splines

Kurven-Konstruktion

- Anwendungen
  - Computergraphik
  - CAD
  - Analyse von Meßwerten
  - Datenkompression
  - Approximation von Funktionen

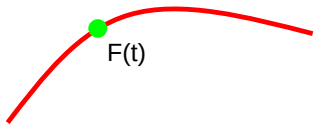
Kurven-Konstruktion

- Kurven werden über Funktionen definiert

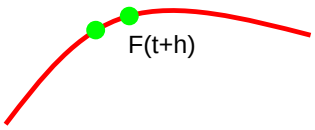
$$F(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} \qquad F(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

- „Parameterisierung“
- „Trajektorie“ (t = Zeitparameter)

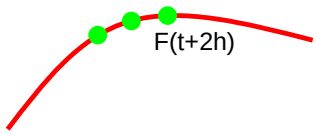
Kurven-Konstruktion



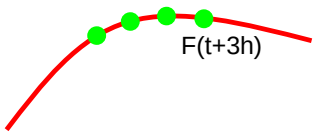
Kurven-Konstruktion



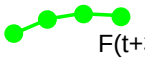
Kurven-Konstruktion



Kurven-Konstruktion

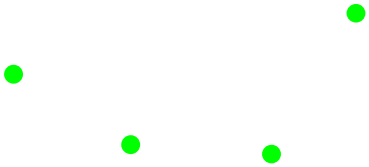


Kurven-Konstruktion



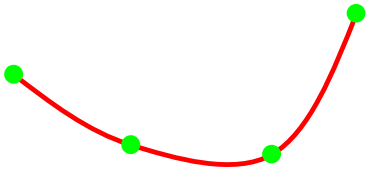
Kurven-Konstruktion

- Interpolation



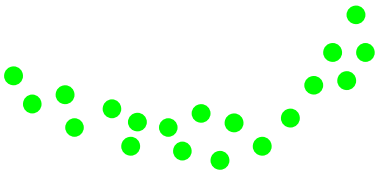
Kurven-Konstruktion

- Interpolation



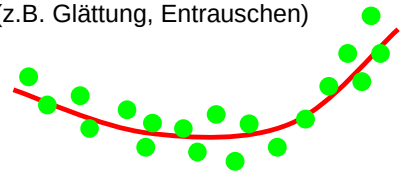
Kurven-Konstruktion

- Approximation



Kurven-Konstruktion

- Approximation
  (z.B. Glättung, Entrauschen)



Polynome

- Effiziente Berechnung:
  Verwende nur die ***Grundrechenarten***

$$F(t)=\sum_{i=0}^{n} f_i \cdot t^i \qquad F(t) \in \Pi_n$$

- Polynome vom ***Grad n***
  ($f_i = 0$ erlaubt, insbesondere $f_n = 0$)

Polynome

- Vektorraum der Polynome

$$F(t)=\sum_{i=0}^{n} f_i \cdot t^i$$

$$F(t) \cong [f_0,\ldots,f_n] \in R^{n+1}$$

Polynome

- Vektorraum der Polynome

$$F(t)=\sum_{i=0}^{n} f_i \cdot t^i \qquad G(t)=\sum_{i=0}^{n} g_i \cdot t^i$$

$$(\alpha \cdot F + \beta \cdot G)(t) = \sum_{i=0}^{n} (\alpha \cdot f_i + \beta \cdot g_i) \cdot t^i$$

Polynome

- Vektorraum der Polynome

$$F(t) \cong [f_0,\ldots,f_n] \qquad G(t) \cong [g_0,\ldots,g_n]$$

$$(\alpha \cdot F + \beta \cdot G)(t) \cong [\ldots(\alpha \cdot g_i + \beta \cdot f_i)\ldots]$$

Polynome

- Basisvektoren

$$E_0(t) \cong [1,0,0,\ldots,0]$$

$$E_1(t) \cong [0,1,0,\ldots,0]$$

$$\ldots$$

$$E_n(t) \cong [0,0,\ldots,0,1]$$

Polynome

- Basis***funktionen***

$$E_0(t) \cong [1,0,0,\ldots,0] \cong 1$$

$$E_1(t) \cong [0,1,0,\ldots,0] \cong t$$

$$\ldots$$

$$E_n(t) \cong [0,0,\ldots,0,1] \cong t^n$$

Polynome

- Linearkombination von Basisfunktionen

$$F(t) = \sum_{i=0}^{n} f_i \cdot t^i = \sum_{i=0}^{n} f_i \cdot E_i(t)$$

- Basistransformation ...

$$B^{-1} = \left[b_{i,j}\right] \in R^{n+1 \times n+1} \qquad B_j(t) = \sum_{i=0}^{n} b_{i,j} \cdot t^i$$

Interpolation

- Geg.: $p_0, \ldots, p_n$ und $t_0, \ldots, t_n$
- Ges.: Polynom F vom Grad n, so dass $F(t_i) = p_i$ für i=0, ..., n
- Lineare Bedingung pro Interpolationspunkt

$[1, t_i, t_i^2, \ldots, t_i^n] \times [f_0, \ldots, f_n]^T = p_i$

Interpolation

- Löse lineares Gleichungssystem

$$\begin{pmatrix} 1 & t_0 & t_0^2 & \cdots & t_0^n \\ 1 & t_1 & t_1^2 & \cdots & t_1^n \\ 1 & t_2 & t_2^2 & \cdots & t_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^n \end{pmatrix} \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix} = \begin{pmatrix} p_0 \\ \vdots \\ p_n \end{pmatrix}$$

Interpolation

- Löse lineares Gleichungssystem

$$\begin{pmatrix} E_0(t_0) & \cdots & E_n(t_0) \\ \vdots & \ddots & \vdots \\ E_0(t_n) & \cdots & E_n(t_n) \end{pmatrix} \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix} = \begin{pmatrix} p_0 \\ \vdots \\ p_n \end{pmatrix}$$

Interpolation

- Lagrange-Basis

$$L_i(t) = \frac{\prod_{j \neq i}(t - t_j)}{\prod_{j \neq i}(t_i - t_j)}$$

$$L_i(t_k) = \delta_{i,k} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

Interpolation

- Lagrange-Basis

$$L_i(t) = \frac{\prod_{j \neq i}(t - t_j)}{\prod_{j \neq i}(t_i - t_j)}$$

$$F(t) = \sum_{i=0}^{n} p_i \, L_i(t)$$

Aitken‘s Scheme

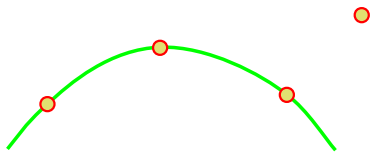
- Geg.: $p_0, ..., p_n$ und $t_0, ..., t_n$
- Ges.: Polynom F vom Grad n, so dass $F(t_i) = p_i$ für i=0, ..., n
- Werte $F(t)$ an einer beliebigen Zwischenstelle s aus ...

Aitken‘s Scheme

- Sei $G_{i,j}(t)$ das Polynom vom Grad $j-i$, das die Bedingungen $G_{i,j}(t_k) = p_k$ für $k=i ... j$ erfüllt.
- Basisfall $G_{i,i}(t) = p_i$
- Rekursion

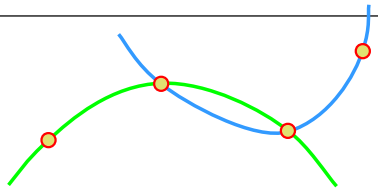
$$G_{i,j+1}(t) = [(t_{j+1} - t)\, G_{i,j}(t) + (t - t_i)\, G_{i+1,j+1}(t)] / (t_{j+1} - t_i)$$

Aitken‘s Scheme



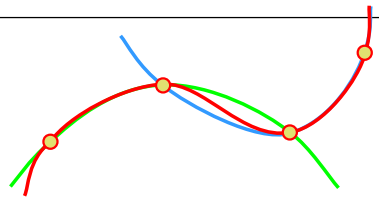
$$G_{i,j+1}(t) = [(t_{j+1} - t)\, G_{i,j}(t) + (t - t_i)\, G_{i+1,j+1}(t)] / (t_{j+1} - t_i)$$

Aitken‘s Scheme



$$G_{i,j+1}(t) = [(t_{j+1} - t)\, G_{i,j}(t) + (t - t_i)\, G_{i+1,j+1}(t)] / (t_{j+1} - t_i)$$

Aitken‘s Scheme



$$G_{i,j+1}(t) = [(t_{j+1} - t)\, G_{i,j}(t) + (t - t_i)\, G_{i+1,j+1}(t)] / (t_{j+1} - t_i)$$

Aitken‘s Scheme

- Aitken(p[0..n], t[0..n], s)

```
for (i=0 ; i≤n ; i++)
    q[ i ] = p[ i ];
for (j=1 ; j≤n ; j++)
    for (i=0 ; i≤n-j ; i++)
        q[ i ] = (t[ i+j ]-s)*q[ i ] + (s-t[ i ])*q[ i+1 ];
        q[ i ] = q[ i ] / (t[ i+j ]- t[ i ]);
```

Aitken‘s Scheme

$P[0] = G_{0,0}$
$P[1] = G_{1,1}$
$P[2] = G_{2,2}$
$P[3] = G_{3,3}$

...

$P[n] = G_{n,n}$

Aitken‘s Scheme

$P[0] = G_{0,0} \rightarrow G_{0,1}$
$P[1] = G_{1,1} \rightarrow G_{1,2}$
$P[2] = G_{2,2} \rightarrow G_{2,3}$
$P[3] = G_{3,3} \rightarrow G_{3,4}$

...

$P[n] = G_{n,n} \rightarrow G_{n-1,n}$

Aitken‘s Scheme

$P[0] = G_{0,0} \rightarrow G_{0,1} \rightarrow G_{0,2}$
$P[1] = G_{1,1} \rightarrow G_{1,2} \rightarrow G_{1,3}$
$P[2] = G_{2,2} \rightarrow G_{2,3} \rightarrow G_{2,4}$
$P[3] = G_{3,3} \rightarrow G_{3,4}$

...

$P[n] = G_{n,n} \rightarrow G_{n-1,n} \rightarrow G_{n-2,n}$

Aitken‘s Scheme

$P[0] = G_{0,0} \rightarrow G_{0,1} \rightarrow G_{0,2} \rightarrow \dots \rightarrow G_{0,n}$
$P[1] = G_{1,1} \rightarrow G_{1,2} \rightarrow G_{1,3}$
$P[2] = G_{2,2} \rightarrow G_{2,3} \rightarrow G_{2,4}$
$P[3] = G_{3,3} \rightarrow G_{3,4}$

...

$P[n] = G_{n,n} \rightarrow G_{n-1,n} \rightarrow G_{n-2,n}$

Aitken‘s Scheme

$G_{i,j} \xrightarrow{\frac{t_{j+1} - s}{t_{j+1} - t_i}} G_{i,j+1}$

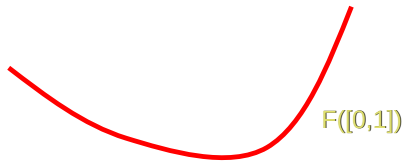
$G_{i+1,j+1} \xrightarrow{\frac{s - t_i}{t_{j+1} - t_i}} G_{i,j+1}$

Aitken‘s Scheme

- Aitken ist eigentlich ein Algorithmus zum ***Auswerten*** des Interpolationspolynoms
- Löst Interpolation und Evaluierung in einem Schritt
- Implizite Repräsentation des Polynoms durch eine Menge von n+1 Stützstellen

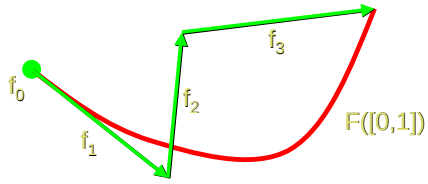
Monome vs. Lagrange / Aitken

$[f_0, f_1, f_2, f_3]$ vs. $[p_0, p_1, p_2, p_3]$



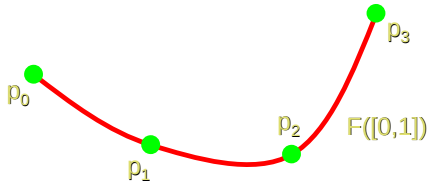
Monome vs. Lagrange / Aitken

$[f_0, f_1, f_2, f_3]$ vs. $[p_0, p_1, p_2, p_3]$



Monome vs. Lagrange / Aitken

$[f_0, f_1, f_2, f_3]$ vs. $[p_0, p_1, p_2, p_3]$



Evaluation

- Geg.: Koeffizienten $[f_0, \ldots, f_n]$
- Ges.: Funktionswert $F(s)$

- Aitken: $O(n^2)$
- Horner Schema: $O(n)$

Evaluation

- Geg.: Koeffizienten $[f_0, \ldots, f_n]$
- Ges.: Funktionswerte
  $F(s), F(s+h), F(s+2h), \ldots$

- Vorwärtsdifferenzen $O(n)$
  (nur n Additionen pro Wert)

Differenzenoperator

- $\Delta F(t) = F(t+h) - F(t)$

- $\Delta^{k+1}F(t) = \Delta(\Delta^k F(t)) = \Delta^k F(t+h) - \Delta^k F(t)$

- $\Delta^2 F(t) = \Delta F(t+h) - \Delta F(t)$
  $= F(t+2h) - F(t+h) - F(t+h) + F(t)$
  $= F(t+2h) - 2F(t+h) + F(t)$

Differenzenoperator

- Wenn F(t) ein Polynom vom Grad n ist, dann ist ΔF(t) ein Polynom vom Grad n-1.

$$\Delta F(t) = F(t+h) - F(t) = \sum_{i=0}^{n} f_i (t+h)^i - \sum_{i=0}^{n} f_i t^i$$

Differenzenoperator

- Wenn F(t) ein Polynom vom Grad n ist, dann ist ΔF(t) ein Polynom vom Grad n-1.

$$\sum_{i=0}^{n} f_i (t+h)^i - \sum_{i=0}^{n} f_i t^i = f_n t^n + \ldots - f_n t^n - \ldots$$

Differenzenoperator

- Wenn F(t) ein Polynom vom Grad n ist, dann ist $\Delta^k F(t)$ ein Polynom vom Grad n-k.
- Insbesondere $\Delta^n F(t)$ ist ***konstant*** für alle t (Polynom vom Grad n-n=0)

Vorwärtsdifferenzen

F(s)
F(s+h)
F(s+2h)
F(s+3h)

Vorwärtsdifferenzen

F(s) ↘ -
F(s+h) → + ΔF(s)
F(s+2h)
F(s+3h)

Vorwärtsdifferenzen

F(s)
F(s+h) ↘ - ΔF(s)
F(s+2h) → + ΔF(s+h)
F(s+3h)

Vorwärtsdifferenzen

$F(s)$				
$F(s+h)$		$\Delta F(s)$	$\overset{-}{\searrow}$	
$F(s+2h)$		$\Delta F(s+h)$	$\xrightarrow[+]{}$	$\Delta^2 F(s)$
$F(s+3h)$				

Vorwärtsdifferenzen

$F(s)$				
$F(s+h)$		$\Delta F(s)$		
$F(s+2h)$	$\overset{-}{\searrow}$	$\Delta F(s+h)$		$\Delta^2 F(s)$
$F(s+3h)$	$\xrightarrow[+]{}$	$\Delta F(s+2h)$		

Vorwärtsdifferenzen

$F(s)$			
$F(s+h)$	$\Delta F(s)$		
$F(s+2h)$	$\Delta F(s+h)$	$\overset{-}{\searrow}$	$\Delta^2 F(s)$
$F(s+3h)$	$\Delta F(s+2h)$	$\xrightarrow[+]{}$	$\Delta^2 F(s+h)$

Vorwärtsdifferenzen

$F(s)$				
$F(s+h)$	$\Delta F(s)$			
$F(s+2h)$	$\Delta F(s+h)$	$\Delta^2 F(s)$	$\overset{-}{\searrow}$	
$F(s+3h)$	$\Delta F(s+2h)$	$\Delta^2 F(s+h)$	$\xrightarrow[+]{}$	$\Delta^3 F(s)$

Vorwärtsdifferenzen

$F(s)$			
$F(s+h)$	$\Delta F(s)$		
$F(s+2h)$	$\Delta F(s+h)$	$\Delta^2 F(s)$	
$F(s+3h)$	$\Delta F(s+2h)$	$\Delta^2 F(s+h)$	$\Delta^3 F(s)$

Vorwärtsdifferenzen

$F(s)$			
$F(s+h)$	$\Delta F(s)$		
$F(s+2h)$	$\Delta F(s+h)$	$\Delta^2 F(s)$	
$F(s+3h)$	$\Delta F(s+2h)$	$\Delta^2 F(s+h)$	$\Delta^3 F(s)$
			$\Delta^3 F(s+h)$

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
			Δ^3F(s+h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
		Δ^2F(s+2h)	Δ^3F(s+h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)
			Δ^3F(s+2h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)
		Δ^2F(s+3h)	Δ^3F(s+2h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)
	ΔF(s+4h)	Δ^2F(s+3h)	Δ^3F(s+2h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)
F(s+5h)	ΔF(s+4h)	Δ^2F(s+3h)	Δ^3F(s+2h)

Vorwärtsdifferenzen

F(s)			
F(s+h)	ΔF(s)		
F(s+2h)	ΔF(s+h)	Δ^2F(s)	
F(s+3h)	ΔF(s+2h)	Δ^2F(s+h)	Δ^3F(s)
F(s+4h)	ΔF(s+3h)	Δ^2F(s+2h)	Δ^3F(s+h)
F(s+5h)	ΔF(s+4h)	Δ^2F(s+3h)	Δ^3F(s+2h)

Vorwärtsdifferenzen

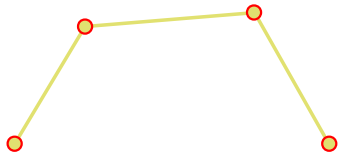
```
TraceCubic(F(),s,h)
  a = F(s);  b = F(s+h);  c = F(s+2h);  d = F(s+3h);
  a = b-a;   b = c-b;       c = d-c;
  a = b-a;   b = c-b;
  a = b-a;
  while (!finished)
      b += a;  c += b;  d += c;
      output(d);
```


Konstruktion / Modellierung

- *bisher:* Polynom gegeben oder durch Interpolation berechnet
- Design: interaktive Formgebung
- Intuitiver Zusammenhang zwischen Koeffizienten und Form der Kurve !?!? („Kontrollpunkte“)

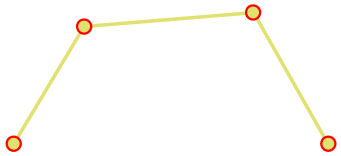
Bezier-Kurven

- De Casteljau - Algorithmus



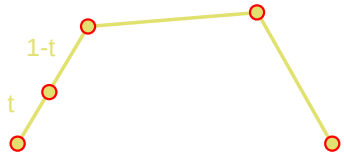
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



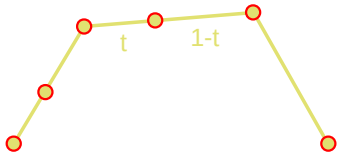
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



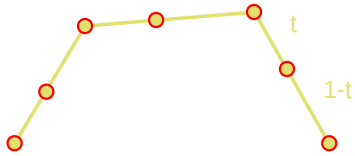
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



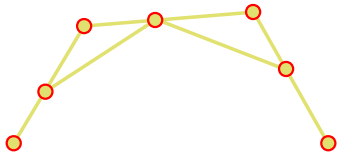
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



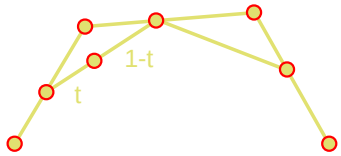
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



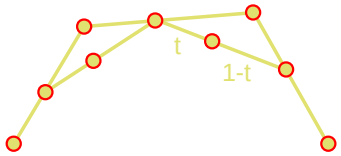
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



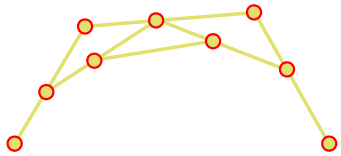
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



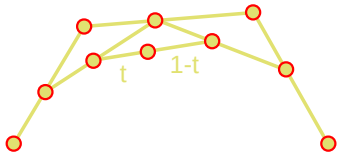
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



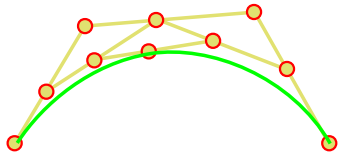
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



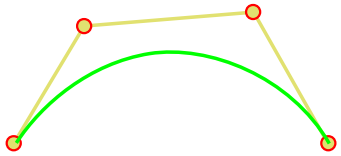
Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



Bezier-Kurven

- De Casteljau - Algorithmus ... t ∈ [0,1]



Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1
  b_2
  ...
  b_n

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0 (1-t)
  b_1 (t) b_{01}
  b_2 b_{12}
  ... ...
  b_n $b_{n-1\ n}$

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 (1-t) b_{01}
  b_2 (t) b_{12}
  ... ...
  b_n $b_{n-1\ n}$

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01}
  b_2 b_{12}
  ... (1-t) ...
  b_n (t) $b_{n-1\ n}$

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01}
  b_2 b_{12}
  ... ...
  b_n $b_{n-1\ n}$
  linear

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01} (1-t)
  b_2 b_{12} (t) b_{02}
  ... ...
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01}
  b_2 b_{12} b_{02}
  ... ...
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$

1-t, t

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01}
  b_2 b_{12} b_{02}
  ... ...
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$

quadratisch

Bezier-Kurven

- Kurve durch ***Kontrollpunkte*** definiert, die den ungefähren Verlauf festlegen
- b_0
  b_1 b_{01}
  b_2 b_{12} b_{02}
  ... ...
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$ ... b_{0n}

Grad n

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?
- b_0
  b_1 b_{01}
  b_2 b_{12} b_{02}
  ... ...
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$ ... b_{0n}

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?
- b_0 1-t
  b_1 b_{01} 1-t
  b_2 b_{12} b_{02}
  ... ... 1-t
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$ ... b_{0n}

$b_0\ (1-t)^n$

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?
- b_0
  b_1 b_{01} 1-t
  b_2 b_{12} b_{02}
  ... ... 1-t
  b_n $b_{n-1\ n}$ $b_{n-2\ n}$ ... b_{0n}

t

$b_1\ t\ (1-t)^{n-1}$

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?
- b_0
  b_1 1-t b_{01} $\quad b_1\ 2\ t\ (1-t)^{n-1}$
  b_2 $\quad b_{12}$ t b_{02}
  ... $\quad$ ... $\quad$ ... 1-t
  b_n $\quad b_{n-1\ n}$ $\quad b_{n-2\ n}$ ... b_{0n}

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?
- b_0
  b_1 1-t b_{01} $\quad b_1\ n\ t\ (1-t)^{n-1}$
  b_2 $\quad b_{12}$ 1-t b_{02}
  ... $\quad$ ... $\quad$ ...
  b_n $\quad b_{n-1\ n}$ $\quad b_{n-2\ n}$ ... t b_{0n}

Bezier-Kurven

- Welchen Einfluß hat der Kontrollpunkt b_i auf die Kurve ?

$$B(t) = \sum_{i=0}^{n} b_i \binom{n}{i} t^i (1-t)^{n-i}$$

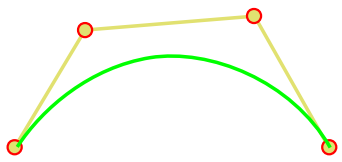
- Bernstein-Polynome: $B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$

Bezier-Kurven

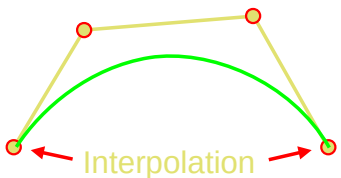
- Die Bernstein-Polynome bilden eine ***Basis*** ...
- Interpolation:

$$\begin{pmatrix} B_0^n(t_0) & \cdots & B_n^n(t_0) \\ \vdots & \ddots & \vdots \\ B_0^n(t_n) & \cdots & B_n^n(t_n) \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} p_0 \\ \vdots \\ p_n \end{pmatrix}$$

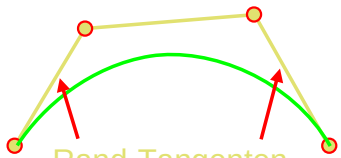
Eigenschaften



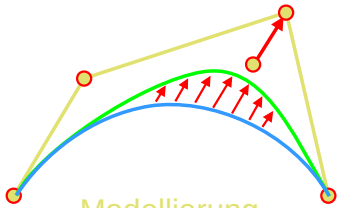
Eigenschaften



Eigenschaften



Eigenschaften



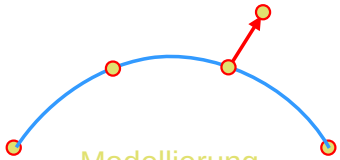
Modellierung

- Warum ist die **Bernstein**-Basis besser als die **Lagrange**-Basis ?

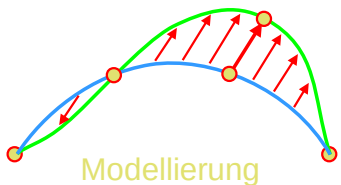
$$L_i(t) = \frac{\prod_{j \neq i}(t - t_j)}{\prod_{j \neq i}(t_i - t_j)} \qquad B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

- $\Rightarrow$ ***Positivität !***

Lagrange-Basis



Lagrange-Basis



Blossoms

- Neue „Schreibweise“ für Bezier-Kurven (hier: kubische Kurven als Beispiel)
- Kontrolpunkte: b_0, b_1, b_2, b_3
- Schreibe: (0,0,0), (0,0,1), (0,1,1), (1,1,1)
- „Polarform“, „Multi-affin-Form“, ...

Blossoms

- Erlaubte Operationen:
  - Permutation: (a,b,c) = (a,c,b) = (b,a,c) = ...
  - Affinkombinationen:
    α (a,b,c) + (1-α) (a,b,d) = (a,b,α c + (1-α) d)
- Kurve: F(t) := (t,t,t)

De Casteljau

(0,0,0)

(0,0,1)

(0,1,1)

(1,1,1)

De Casteljau

(0,0,0) 1-t

(0,0,0*(1-t)+1*t) = (0,0,t)

(0,0,1) t

(0,1,1)

(1,1,1)

De Casteljau

(0,0,0)

(0,0,t)

(0,0,1) 1-t

(0,1,t)

(0,1,1) t

(1,1,1)

De Casteljau

(0,0,0)

(0,0,t)

(0,0,1)

(0,1,t)

(0,1,1) 1-t

(1,1,t)

(1,1,1) t

De Casteljau

(0,0,0)

(0,0,t) 1-t

(0,0,1)

(0,t,t)

(0,1,t) t

(0,1,1)

(1,1,t)

(1,1,1)

De Casteljau

(0,0,0)			
	(0,0,t)		
(0,0,1)		(0,t,t)	
	(0,1,t) 1-t →		
(0,1,1)		(1,t,t)	
	(1,1,t) → t		
(1,1,1)			

De Casteljau

(0,0,0)			
	(0,0,t)		
(0,0,1)		(0,t,t) 1-t →	
	(0,1,t)		(t,t,t)
(0,1,1)		(1,t,t) → t	
	(1,1,t)		
(1,1,1)			

De Casteljau

(0,0,0)			
	(0,0,t)		
(0,0,1)		(0,t,t)	
	(0,1,t)		(t,t,t)
(0,1,1)		(1,t,t)	
	(1,1,t)		
(1,1,1)			

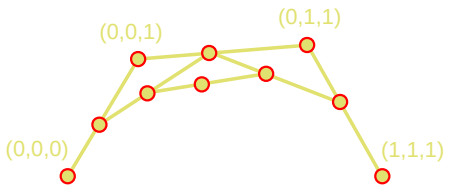
De Casteljau

(0,0,0)			
	(0,0,t)		
(0,0,1)		(0,t,t)	
	(0,1,t)		(t,t,t)
(0,1,1)		(1,t,t)	
	(1,1,t)		
(1,1,1)			

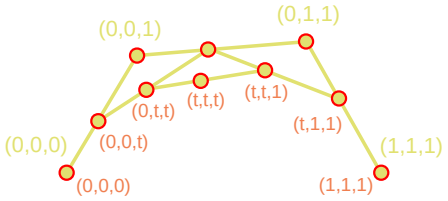
De Casteljau

(0,0,0)			
	(0,0,t)		
(0,0,1)		(0,t,t)	
	(0,1,t)		(t,t,t)
(0,1,1)		(1,t,t)	
	(1,1,t)		
(1,1,1)			

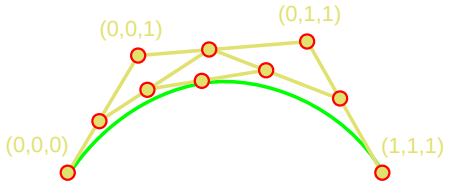
De Casteljau



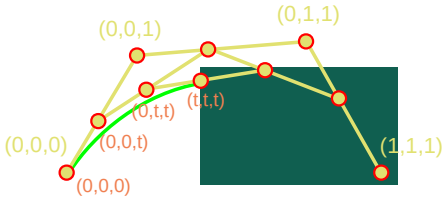
De Casteljau



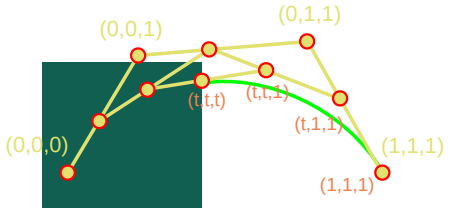
De Casteljau



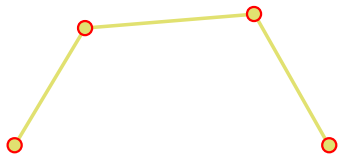
De Casteljau



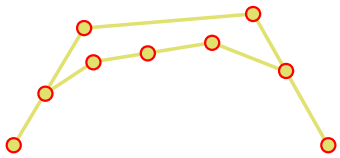
De Casteljau



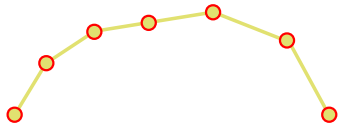
Subdivision



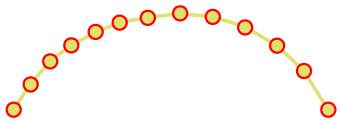
Subdivision



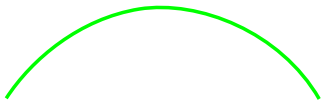
Subdivision



Subdivision



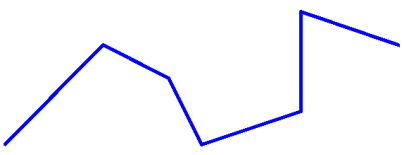
Subdivision



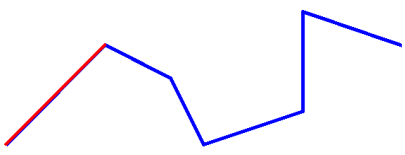
Komplexität

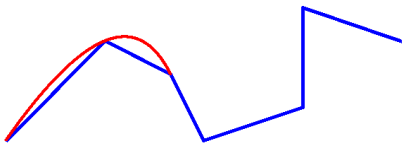
- Problem: Anzahl der Kontrollpunkte (= Freiheitsgrade für die Modellierung) bestimmt den **Polynomgrad**
- Rechenaufwand steigt wie $O(n^2)$
- Besser: Splines = Stückweise Polynome

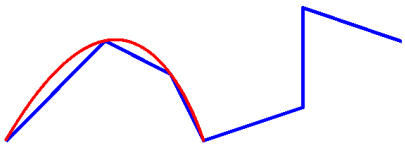
Stabilität

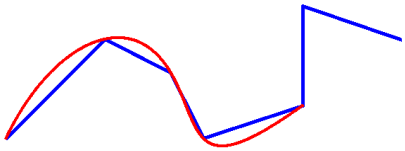


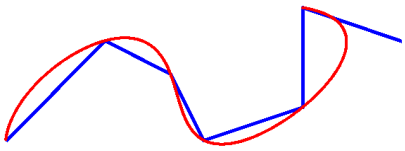
Stabilität

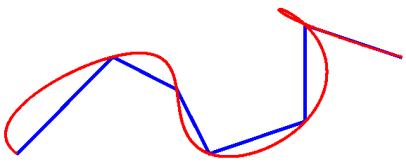












Stabilität

Stabilität

Stabilität

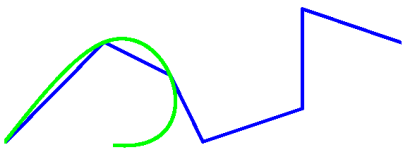
Stabilität

Stabilität

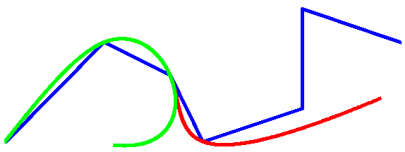
Splines

- Setze eine Kurve aus mehreren polynomiellen Segmenten zusammen
- Wie garaniert man glatte Übergänge zwischen den Segmenten?
- Mathematisch: stetige Ableitungen (= Differenzierbarkeit)

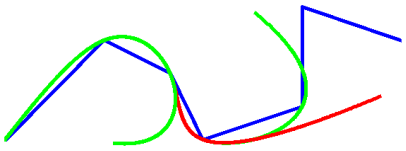
Splines



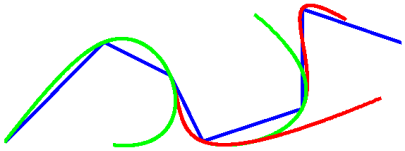
Splines



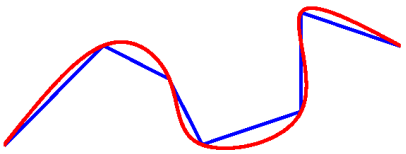
Splines



Splines



Splines



Blossoms

- Definiere eine Sequenz von Kontrollpunkten $d_0, \ldots, d_m$
- Lege Polynomgrad n fest (in unserem Fall wählen wir $n = 3$, kubisch)
- Schreibweise: $d_i = (i,i+1,i+2)$

De Boor

(0,1,2)

(1,2,3)

(2,3,4)

(3,4,5)

De Boor

t ∈ [2,3)

(0,1,2)

(1,2,3)

(2,3,4)

(3,4,5)

De Boor

t ∈ [2,3)

(0,1,2) (3-t)/3

(1,2,t)

(1,2,3) (t-0)/3

(2,3,4)

(3,4,5)

De Boor

t ∈ [2,3)

(0,1,2)

(1,2,t)

(1,2,3) (4-t)/3

(2,3,t)

(2,3,4) (t-1)/3

(3,4,5)

De Boor

t ∈ [2,3)

(0,1,2)

(1,2,t)

(1,2,3)

(2,3,t)

(2,3,4) (5-t)/3

(3,4,t)

(3,4,5) (t-2)/3

De Boor

t ∈ [2,3)

(0,1,2)

(1,2,t) (3-t)/2

(1,2,3)

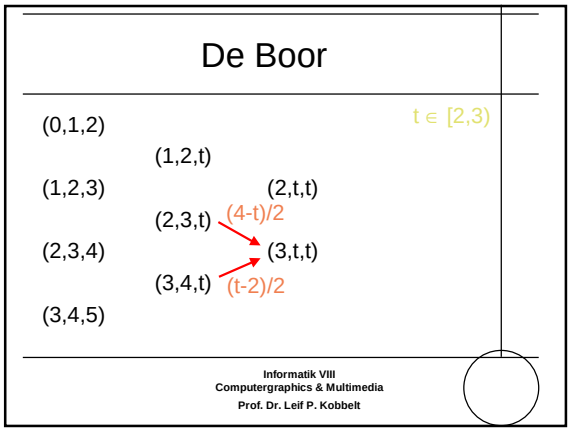
(2,t,t)

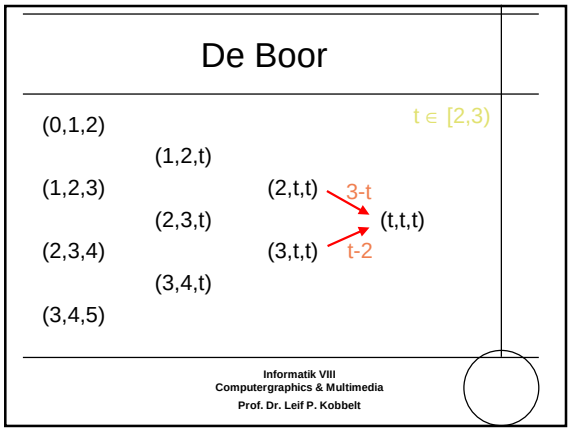
(2,3,t) (t-1)/2

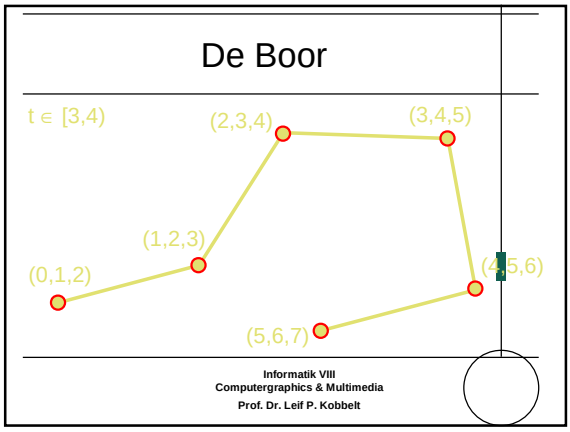
(2,3,4)

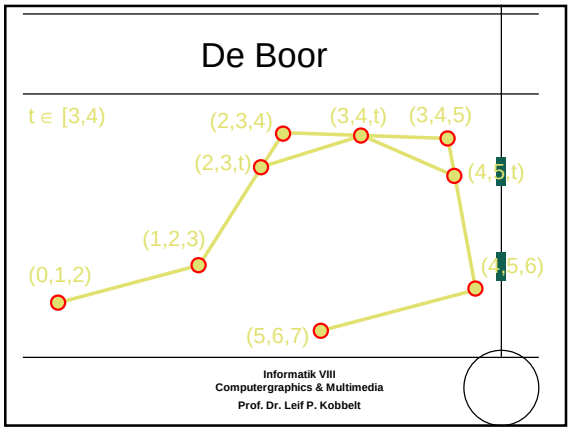
(3,4,t)

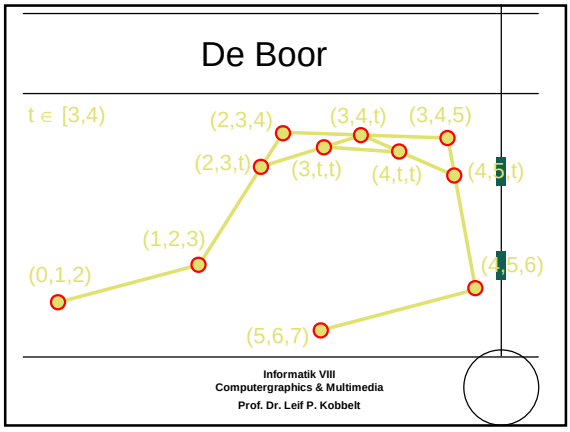
(3,4,5)

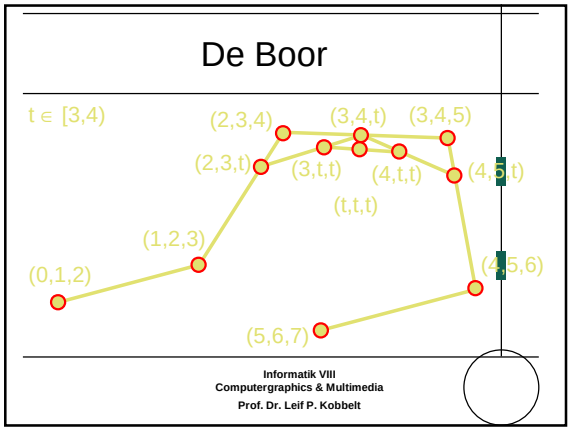

De Boor
t ∈ [2,3)
(0,1,2)
(1,2,t)
(1,2,3)
(2,t,t)
(2,3,t)
(4-t)/2
(2,3,4)
(3,t,t)
(3,4,t)
(t-2)/2
(3,4,5)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


De Boor
t ∈ [2,3)
(0,1,2)
(1,2,t)
(1,2,3)
(2,t,t)
3-t
(2,3,t)
(t,t,t)
(2,3,4)
(3,t,t)
t-2
(3,4,t)
(3,4,5)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

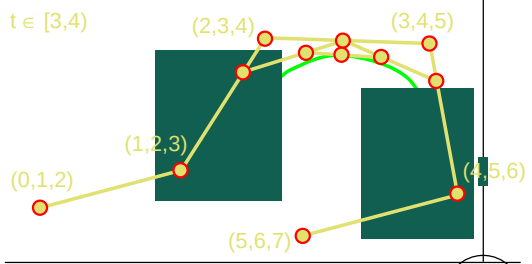

De Boor
t ∈ [3,4)
(2,3,4)
(3,4,5)
(1,2,3)
(0,1,2)
(4,5,6)
(5,6,7)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


De Boor
t ∈ [3,4)
(2,3,4)
(3,4,t)
(3,4,5)
(2,3,t)
(4,5,t)
(1,2,3)
(0,1,2)
(4,5,6)
(5,6,7)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

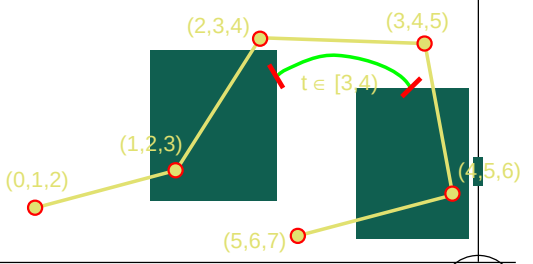

De Boor
t ∈ [3,4)
(2,3,4)
(3,4,t)
(3,4,5)
(2,3,t)
(3,t,t)
(4,t,t)
(4,5,t)
(1,2,3)
(0,1,2)
(4,5,6)
(5,6,7)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


De Boor
t ∈ [3,4)
(2,3,4)
(3,4,t)
(3,4,5)
(2,3,t)
(3,t,t)
(4,t,t)
(4,5,t)
(t,t,t)
(1,2,3)
(0,1,2)
(4,5,6)
(5,6,7)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

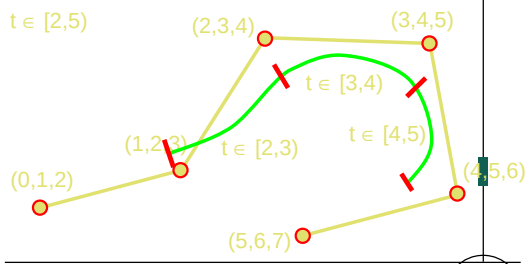
De Boor



De Boor



De Boor



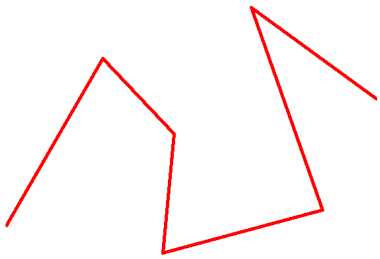
Splines

- Stückweise Polynomial
  - Der de Boor Algorithmus wendet nur affine Kombinationen an (vgl. de Casteljau)
- Glattheit
  - Garantiert durch Überlappung der Kontrollpunkte *(ohne Beweis!)*
- Basis-Funktionen

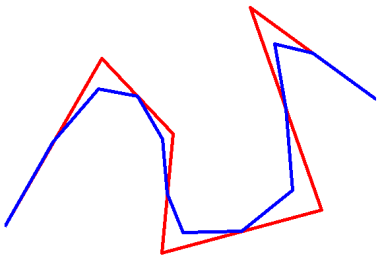
Subdivision

- Schneller Approximationsalgorithmus bei Bezier-Kurven ...
  - Bilde Kontrollpolygonen auf verfeinerte Kontrollpolygone ab (anstatt Auswertung)
- Ähnliche Technik für Splines führt auf extrem effiziente Algorithmen

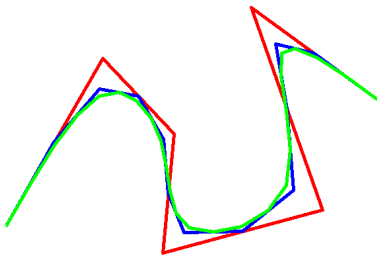
Subdivision



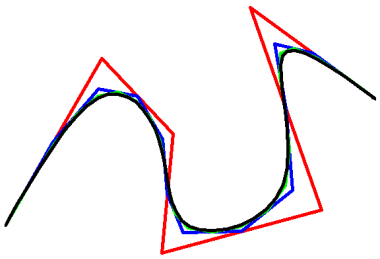
Subdivision



Subdivision



Subdivision



Subdivision

- Originalkurve wird durch $d_0, \ldots, d_m$ repräsentiert
- d.h.: (0,1,2), (1,2,3), ..., (m,m+1,m+2)
- Berechne eine verfeinerte Darstellung mit den Blossom-Werten:
  (1,1½,2), (1½,2,2½), ..., (m,m+½,m+1)

Subdivision

$$\tfrac{1}{2}(i,i+1,i+2)+\tfrac{1}{2}(i+1,i+2,i+3)=(i+1,i+\tfrac{3}{2},i+2)$$

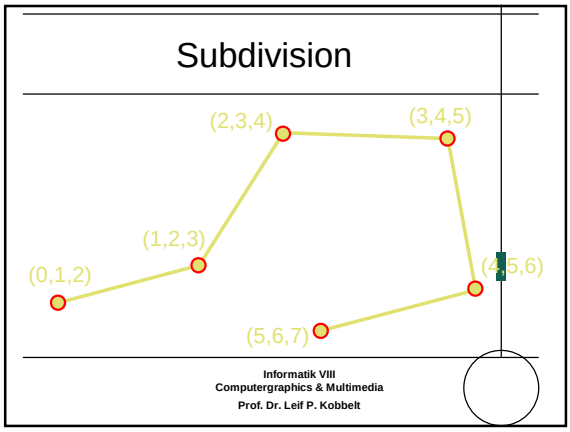
$$\tfrac{5}{6}(i+1,i+2,i+3)+\tfrac{1}{6}(i+2,i+3,i+4)=(i+\tfrac{3}{2},i+2,i+3)$$

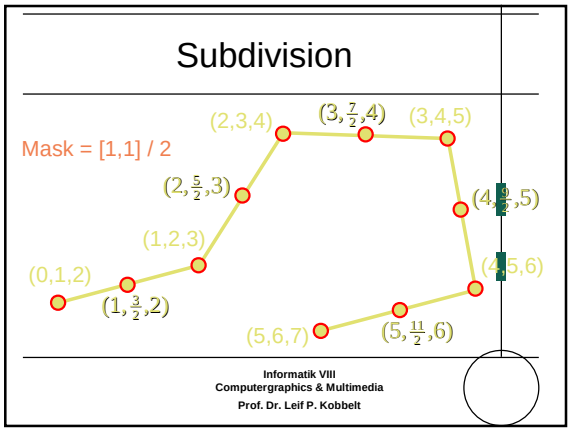
$$\tfrac{1}{4}(i+1,i+\tfrac{2}{3},i+2)+\tfrac{3}{4}(i+\tfrac{2}{3},i+2,i+3)=(i+\tfrac{3}{2},i+2,i+\tfrac{5}{2})$$

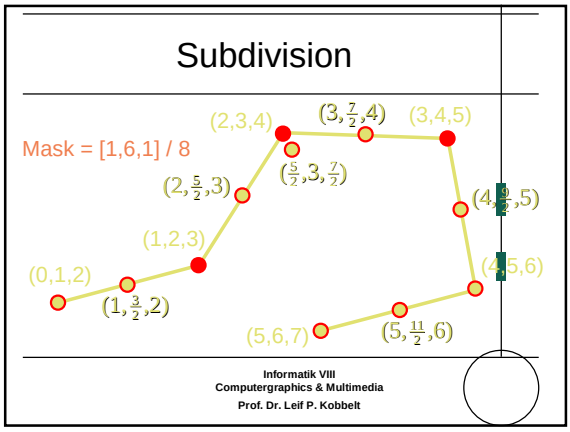
Subdivision

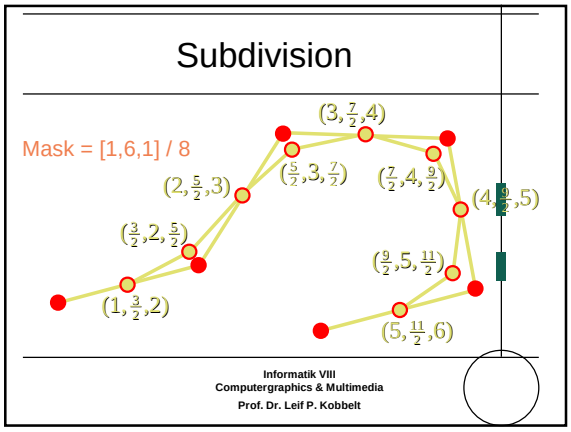
$$\tfrac{1}{2}(i,i+1,i+2)+\tfrac{1}{2}(i+1,i+2,i+3)=(i+1,i+\tfrac{3}{2},i+2)$$

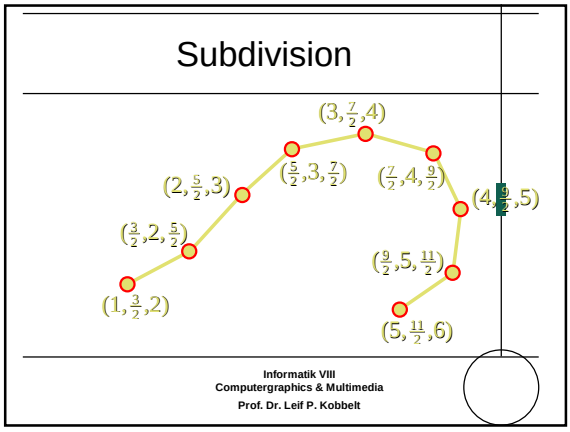
$$\tfrac{1}{8}(i,i+1,i+2)+\tfrac{6}{8}(i+1,i+2,i+3)+\tfrac{1}{8}(i+2,i+3,i+4)$$
$$=(i+\tfrac{3}{2},i+2,i+\tfrac{5}{2})$$

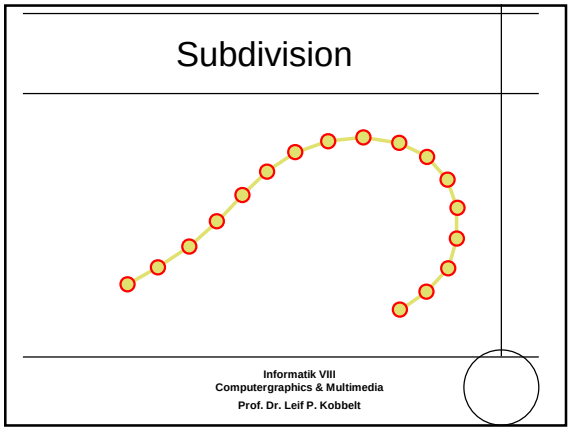

Subdivision
(2,3,4)
(3,4,5)
(1,2,3)
(0,1,2)
(4,5,6)
(5,6,7)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Subdivision
Mask = [1,1] / 2
(2,3,4)
(3, 7/2 ,4)
(3,4,5)
(2, 5/2 ,3)
(1,2,3)
(0,1,2)
(1, 3/2 ,2)
(4,5,6)
(5,6,7)
(5, 11/2 ,6)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Subdivision
Mask = [1,6,1] / 8
(2,3,4)
(3, 7/2 ,4)
(3,4,5)
(2, 5/2 ,3)
(5/2,3, 7/2)
(1,2,3)
(0,1,2)
(1, 3/2 ,2)
(4,5,6)
(5,6,7)
(5, 11/2 ,6)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Subdivision
Mask = [1,6,1] / 8
(3, 7/2 ,4)
(2, 5/2 ,3)
(5/2,3, 7/2)
(7/2,4, 9/2)
(3/2,2, 5/2)
(9/2,5, 11/2)
(1, 3/2 ,2)
(5, 11/2 ,6)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Subdivision
(3, 7/2 ,4)
(2, 5/2 ,3)
(5/2,3, 7/2)
(7/2,4, 9/2)
(3/2,2, 5/2)
(9/2,5, 11/2)
(1, 3/2 ,2)
(5, 11/2 ,6)
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt


Subdivision
Informatik VIII
Computergraphics & Multimedia
Prof. Dr. Leif P. Kobbelt

Subdivision

- Subdiv(d[0..m],d‘[1..2m-1])

```
for (i=0 ; i<m ; i++)
    d‘[2*i+1] = (d[i] + d[i+1]) / 2;
```

```
for (i=1 ; i<m ; i++)
    d‘[2*i] = (d[i-1] + 6*d[i] + d[i+1]) / 8
```


Zusammenfassung

- Polynome
  - hinreichend flexibel
  - verwenden nur Grundrechenarten
- Spezielle Basen
  - *Lagrange* für Interpolation
  - *Bernstein* für Konstruktion / Modellierung

Zusammenfassung

- Auswertung in O(n)
  - Horner Schema
  - Vorwärtsdifferenzen
- Blossoms
- Komplexe Kurven
  - Splines (mehrere Polynom-Segmente)
  - Effiziente Approximation durch **Subdivision**

Zusammenfassung

- Algorithmen
  - Aitken ... Auswerten des Interpolationspolynoms
  - Vorwärtsdifferenzen
  - de Casteljau (Evaluation, Subdivision)
  - de Boor (Spline-Evaluation)
  - Uniform Subdivision für Splines