universität wien

Fakultät für Informatik

Vorlesung

Algorithmen und Datenstrukturen

PI.ADS.AD.VO

3 Stunden / 4 ECTS Punkte

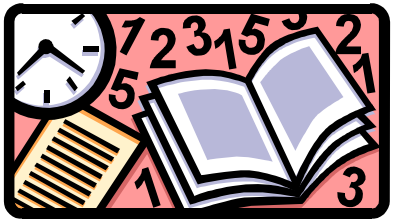
Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Institut für Knowledge and Business Engineering

Fakultät für Informatik, Universität Wien

SS 2009

Inhaltsüberblick



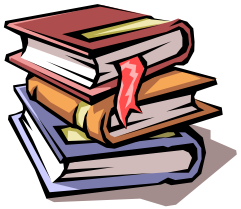
1. Algorithmen
   - Paradigmen, Analyse
2. Datenstrukturen
   - Allgemeiner Überblick
3. Vektoren
   - Hashing, Sortieren
4. Listen
   - Lineare Speicherstrukturen, Stack, Queue
5. Bäume
   - Suchstrukturen
6. Graphen
   - Traversierungs- und Optimierungsalgorithmen

Literatur

R. Sedgewick, *Algorithmen in C++* (Teil 1-4), Addison Wesley, 3. überarbeitete Auflage, 2002

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, published by MIT Press and McGraw-Hill. (First published in 1990.)

Jim Gray, Vortrag: Parallel Database Systems Analyzing LOTS of Data, Microsoft Research, 1997



Danksagung

Für Mitarbeit und Durchsicht der Folien geht mein besonderer Dank an Helmut Wanek und Martin Polaschek

Mein weiterer Dank geht an zahlreiche Studierende der letzten Jahre, die im Rahmen ihrer Übungen die Basis für einige der dynamischen Beispiele der VO lieferten.

Kapitel 1
Algorithmen

1.1 Motivation

Algorithmen

Verfahrensvorschriften, Anweisungsfolgen, Vorgangsmodellierungen, beschriebene Lösungswege

Ziele

- Algorithmen zu Problemstellungen finden!
  - Lösungsansätze finden, „konstruieren“
- 2. „Gute“ Algorithmen finden!
  - “bessere” Algorithmen
    - schneller, vollständiger, korrekter, ...
  - “leistungsfähigere” Datenstrukturen
    - kompakter, effizienter, flexibler, ...
  - Generell: Ersparnis an Rechenzeit und/oder Speicherplatz

Algorithmen zu Problemstellungen finden!

Aufgabe: “Summe der ganzen Zahlen bis n”

Straight-forward solution: “Aufsummieren der einzelnen Werte zwischen 1 und n”

$$summe \leftarrow \sum_{i \leftarrow 1}^{n} i$$

Realisierung (C/C++ Programm)

```
int sum(int n) {
  int i, summe = 0;
  for(i=1; i<=n; i++)
    summe += i;
  return summe;
}
```

Vergleiche mit anderem Programmieransatz, z.B. for- statt while-Schleife!

⇒ alternativer Programmierstil (warum?)

Alternative Realisierung (1)

Zwei Alternativen

1. Alternativer Programmierstil
   - Problemlösungsansatz beibehalten, aber programmiertechnische Umsetzung überarbeiten

Beispiel:

- Schleifenform (siehe oben)
- Rekursion statt Iteration

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

Alternative Realisierung (2)

2. Alternativer Lösungsweg
   - Wahl eines anderen Problemlösungsweges, z.B.
   - Gauß´sche Summenformel

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

Umsetzung

```
int sum(int n) {
  return (n*(n+1))/2;
}
```

Ableitung der Gauß'sche Summenformel

1	…	n
n	…	1
n+1	…	n+1

n * (n+1)

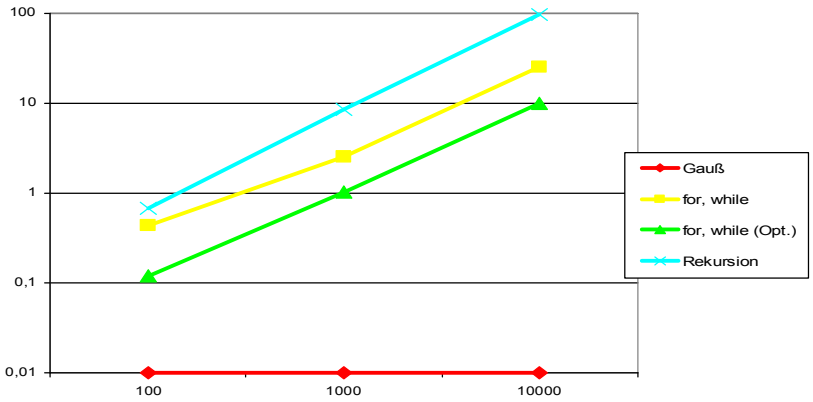
= 2 * Summe von 1 bis n

Daraus folgt die Summen-Formel n * (n+1) /2

„Gute" Algorithmen finden!

Vergleich der Laufzeiten

Summenberechnung, 100000 Wiederholungen, CPU: 200 MHz Pentium



Was ist gut? Oder vielleicht besser?

Problem: Laufzeitenvergleich führt nur zu punktueller Qualitätsbestimmung
- Laufzeit, Speicherplatzverbrauch
- Abhängig von
  - Computer
  - Betriebssystem
  - Compiler, ...

Ziel: Methodik für generellen Qualitätsvergleich zwischen Algorithmen
- Unabhängig von äußeren Einflüssen

1.2 Algorithmen-Paradigmen

Generelle Techniken zur Lösung großer Klassen von Problemstellungen

Greedy algorithms
- "gefräßiger, gieriger" Ansatz, Wahl des lokalen Optimums

Divide-and-conquer algorithms
- schrittweise Zerlegen des Problems der Größe n in kleiner Teilprobleme

Dynamic programming
- dynamischer sukkzessiver Aufbau der Lösung aus schon berechneten Teillösungen

1.2.1 Greedy (1)

In jedem Schritt eines *Greedy* Algorithmus wird die Möglichkeit gewählt, die *unmittelbar* (lokal) den *optimalen* (kleinsten bzw. größten) *Wert* bezüglich der *Zielfunktion* liefert. Dabei wird die globale Sicht auf das Endziel vernachlässigt.

Vorteil:

Effizienter Problemlösungsweg, oft sehr schnell, kann in vielen Fällen relativ gute Lösung finden

Nachteil:

Findet oft keine optimale Lösung

Greedy (2)

Beispiel: Münzwechselmaschine

"Wechsle den Betrag von 18.- in eine möglichst kleine Anzahl von Münzen der Größe 10.-, 5.- und 1.-"

Greedy Ansatz:

```
wähle größte Münze
 kleiner als Betrag
gib die Münze aus
subtrahiere ihren
 Wert vom Betrag
wiederhole solange
 bis Differenz gleich 0
```

d.h.:

```
18.- - 10.- =  8.-  •
 8.- -  5.- =  3.-  •
 3.- -  1.- =  2.-  Œ
 2.- -  1.- =  1.-  Œ
 1.- -  1.- =  0    Œ
```

Lösung für diese Problemstellung nicht nur "gut" sondern sogar optimal!

Greedy (3)

DOCH

Problem bei kleiner Änderung der Problemstellung:

5.- **è** 6.-

Münzwerte 10.-, ***6.-*** , 1.-

greedy Ansatz liefert

```
18.- - 10.- =  8.-  •
 8.- -  6.- =  2.-  ‘
 2.- -  1.- =  1.-  Œ
 1.- -  1.- =  0    Œ
      ⇒ 4 Münzen
```

optimal wäre aber 3 x 6.-

```
18.- -  6.- = 12.-  ‘
12.- -  6.- =  6.-  ‘
 6.- -  6.- =  0    ‘
      ⇒ 3 Münzen
```


1.2.2 Divide-and-conquer (1)

Ausgehend von einer generellen Abstraktion wird das Problem iterativ verfeinert, bis Lösungen für vereinfachte Teilprobleme gefunden wurden, aus welchen eine Gesamtlösung konstruiert werden kann.

Diese Vorgangsweise wird auch oft mit "stepwise refinement" oder "top-down approach" bezeichnet

Divide-and-conquer (2)

Verschiedene Ansätze

Problem size division

Zerlegung eines Problems der Größe n in eine endliche Anzahl von Teilproblemen kleiner n

Step division

Aufteilen einer Aufgabe in eine Sequenz (Folge) von individuellen Teilaufgaben

Case division

Identifikation von Spezialfällen zu einem generellen Problem ab einer gewissen Abstraktionsstufe

…

Problem size division

Durch Zerlegung Verringerung der Problemgröße, d.h.

$P(n) \Rightarrow k*P(m)$,

wobei k, m < n

Binäre Suche

Suche eine Zahl x in der (aufsteigend) sortierten Folge $z_1,z_2,..,z_n$ (allgemein: $z_l,z_{l+1},..,z_r$ mit l=1, r=n) und ermittle ihre Position i

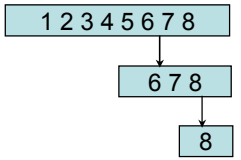
Zerlegung: k = 1 und m ≈ n/2

Trivial: finde mittleren Index m = (l+r)÷2

Falls z_m=x Ergebnis m, sonst

Falls x<z_m suche x im Bereich $z_l,z_2,..,z_{m-1}$ sonst suche x im Bereich $z_{m+1},z_{m+2},..,z_r$

Suche Zahl 8



Step division

Idee: Straßenkehrer-Philisophie:

"Atemzug - Besenstrich - Schritt"

Beispiel

Gehaltserhöhung

Bestimme Mitarbeiter
Finde eindeutige Identifikation
Suche im Datenbestand
Stelle aktuelles Gehalt fest
Ändere auf neues Gehalt
Speichere Information
Vermerke Änderungsvorgang

Speichere Information

Lösche alte Datensatz
Füge neuen Datensatz ein

Case division

Ansatz: Identifikation von Fallunterscheidungen im Problemdatenbereich

Beispiel

Berechnung der Lösungen zu einer quadratischen Gleichung

$$az^2 + bz + c = 0$$

$$z_{1,2} = -(\frac{b}{2a}) \pm \frac{1}{2a} q^{\frac{1}{2}}, q = b^2 - 4ac$$

if $q > 0$, 2 reelle Wurzeln

$q = 0$, reelle Doppellösung

$q < 0$, Paar komplexer Wurzeln

1.2.3 Dynamic Programming (1)

Problem

Oft ist eine Teilung des Originalproblems in eine ‘kleine’ Anzahl von Teilproblemen nicht möglich, sondern führt zu einem exponentiellen Algorithmus.

Man weiß aber, es gibt aber nur eine polynomiale Zahl von Teilproblemen.

Idee

nicht Start von Problemgröße n und Aufteilung bis Größe 1,

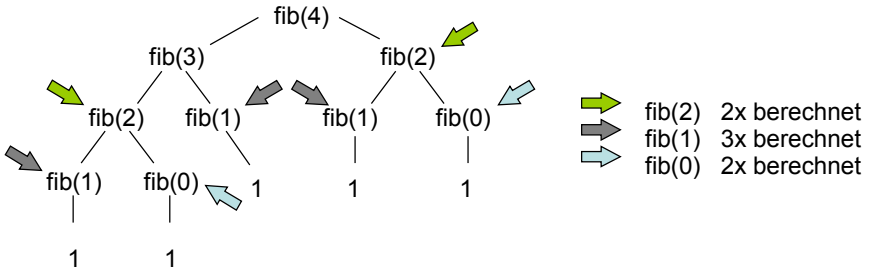
SONDERN

Start mit Lösung für Problemgröße 1, Kombination der berechneten Teillösungen bis eine Lösung für Problemgröße n erreicht wurde.

Dynamic Programming (2)

Beispiel: Berechnung der Fibonacci Zahlen

```
int fib(int n) {
  if(n <= 1) return 1;
  return fib(n-1) + fib(n-2)
}
```

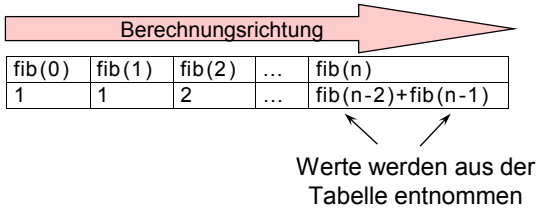


Problem: Wiederholte Lösung eines Teilproblems

Dynamic Programming (3)

Lösungsweg

Beginn mit Berechnung für **`fib(0)`**, Anlegen einer Tabelle aller berechneten Werte und Konstruktion der neuen Werte aus den berechneten Tabelleneinträgen.



Beachte

Verbesserung der Laufzeit

ABER

zusätzlicher Speicherplatzbedarf

1.3 Analyse u. Bewertung von Algorithmen

Ziel ist objektive Bewertung von Algorithmen

Kriterien

Effektivität

Ist das Problem lösbar, ist der Ansatz umsetzbar in ein Programm?

Korrektheit

Macht der Algorithmus was er soll?

Termination

Hält der Algorithmus an, besitzt er eine endliche Ausführungszeit?

Komplexität

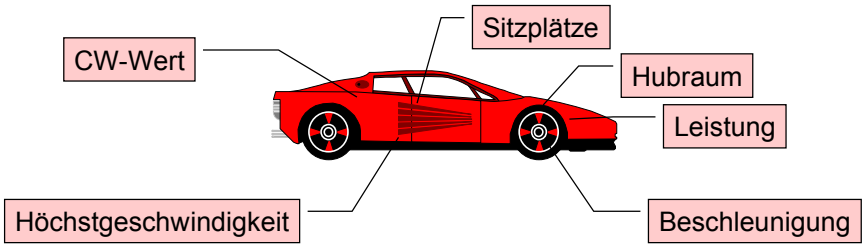
Wie strukturiert ist der Algorithmus ⇒ Strukturkomplexität?

Wie schnell ist der Algorithmus ⇒ Laufzeitkompexität?

Analogie: Auto

Spezifische Kriterien

Auswahl



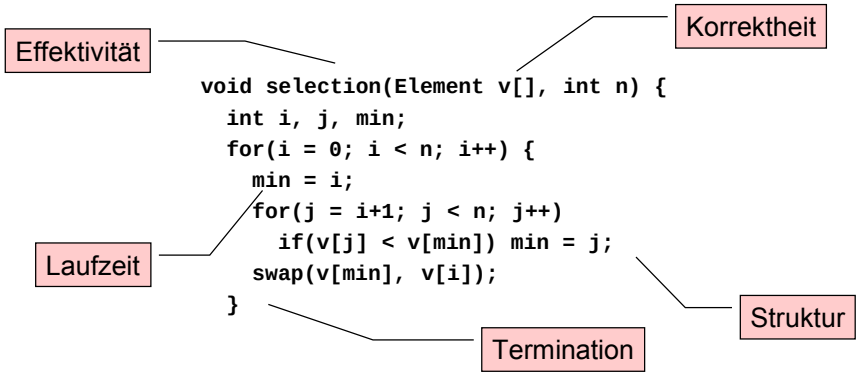
Statistische Kennzahlen

Beurteilungsmöglichkeit, Klassifikationsmöglichkeit

Sportwagen, Lastwagen, Familienlimousine, ...

Bewertung von Programmen

Beipiel Sortierprogramm



Klassfikation

schnell, korrekt, wartbar, problemüberdeckend, endlich, …

1.3.1 Effektivität

Prinzip

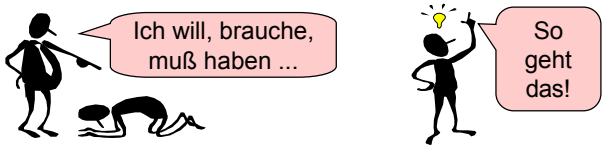
Algorithmus kann als lauffähiges Computerprogramm formuliert werden.

effective ⇒ it does work

Problem

Formulierung

Transformation der Problembeschreibung in einen exekutierbaren Algorithmus



1.3.2 Korrektheit

Produziert der Algorithmus das gewünschte Ergebnis?

Zwei Vorgangsweisen möglich

Testen

Formales Testen

Testen

Vollständiges Austesten meist nicht möglich

Statistischer Ansatz meist verfolgt, z.B. Pfadüberdeckung (Strukturelle Komplexität)

Bestenfalls „Falsifizierung“ erreichbar

Es können nur Fehler gefunden werden, aber es kann keine Korrektheit bewiesen werden

Formales Testen

Mathematisch orientierte Verifikationstechniken erlauben es, die Korrektheit von Programmstücken in Abhängigkeit von Bedingungen an die Eingabedaten (Prämissen) zu beweisen

McCarthy, Naur, Floyd, Hoare, Knuth, Dijkstra, etc.

zwingt den Entwickler Entwurfsentscheidungen noch einmal nachzuvollziehen und hilft beim Auffinden logischer Fehler

Algorithmus muss verstanden werden

je größer ein Programmsystem, umso schwieriger die Verifikation (in der Praxis kaum von Bedeutung)

Beweisansatz abhängig vom Problem

Mathematischer Beweis

Vollständige Induktion

Beispiel: Gauß´sche Summenformel

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

Ind. Anfang

für n = 1 → 1*2/2 = 1 **ü**

Ind. Voraussetzung

(1+2+…+n-1) = (n-1)*n/2

Ind. Schluss

(1+2+…+n-1)+n = (n-1)*n/2 +n =

= ((n-1)*n +2n)/2 = (n2-n+2n)/2 =

= n*(n+1)/2 **ü**

Verifikation

Programm-Verifikation

verschiedene Ansätze

“Rückwärtsbeweis”:

Strukturierter Ansatz, beruht auf *Prädikatentransformation*

Überprüfung, ob die Voraussetzungen (*weakest preconditions, wp*) an die Eingabedaten garantieren, dass das Programm in einem Zustand terminiert, der das gewünschte Programmziel erfüllt.

Man bezeichnet diese Programmziel, welches die Aufgabe des Programms beschreibt, als *Korrektheitsbedingung*

Hierzu darf das Programm nur aus einfachen Zuweisungen, Vergleichen, while-Form Schleifen und Anweisungsfolgen bestehen. Alle anderen Konstrukte müssen übersetzt werden.

Verifikation - Beispiel

Beispiel

```
int sum(int n) {
  int s = 0;
  i = 1;
  while(i <= n) {
    s = s + i;
    i = i + 1;
  }
  return s;
}
```

Rückwärtsbeweis

Korrektheitsbedingung : $s = \sum_{i=1}^{n} i$

(I) *Schleifeninvariante SI*

SI: $(s = \sum_{j=1}^{i-1} j) \wedge (i \le n+1)$

(1) $SI \wedge \neg Bed \Rightarrow KB$

$s = (\sum_{j=1}^{i-1} j) \wedge (i \le n+1) \wedge (i > n) \Rightarrow s = (\sum_{j=1}^{i-1} j) \wedge (i = n+1) \Rightarrow s = \sum_{j=1}^{n} j \Rightarrow KB$

(2) $SI \wedge Bed \Rightarrow wp(Schleifenblock \mid SI)$

$wp(s = s+i \mid wp(i = i+1 \mid (s = \sum_{j=1}^{i-1} j) \wedge (i \le n+1))) \Rightarrow wp(s = s+i \mid (s = \sum_{j=1}^{i} j) \wedge$

$\wedge (i \le n+1)) \Rightarrow (s+i = \sum_{j=1}^{i} j) \wedge (i \le n+1) \Rightarrow (s = \sum_{j=1}^{i-1} j) \wedge (i \le n+1) \equiv SI$

(II) *Rest des Programms*

$wp(s = 0 \mid wp(i = 1 \mid SI)) \Rightarrow wp(s = 0 \mid (s = \sum_{j=1}^{1-1} j) \wedge (1 \le n+1)) \Rightarrow$

$\Rightarrow (0 = \sum_{j=1}^{0} j) \wedge (0 \le n) \Rightarrow \quad n \ge 0$

1.3.3 Termination (1)

Hält der Algorithmus an, d.h. besitzt er eine endliche Ausführungszeit?

Falls nicht klar, manchmal folgende Technik einsetzbar:

Man finde die bestimmende Größe oder Eigenschaft des Algorithmus der die folgenden 3 Charakteristiken erfüllt:

- Eine 1-1 Abbildung dieser Größe auf die ganzen Zahlen kann aufgestellt werden.
- Diese Größe ist positiv.
- Die Größe nimmt während der Ausführung des Algorithmus kontinuierlich ab (dekrementiert).

Termination (2)

Idee:

Die gefundene Größe besitzt bei Algorithmusbeginn einen vorgegebenen positiven Startwert, der sich kontinuierlich verringert. Da die Größe nie negativ werden kann, folgt, dass der Algorithmus terminieren muss, bevor die Größe kleiner 0 ist.

Beispiel:

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

Größe n

Termination bevor n < 0

Dekrement

1.3.4 Strukturelle Komplexität

Bewertende Aussage über den strukturellen Aufbau des Programms und der Programmteile untereinander, d.h. Bewertung des Programmierstils

interne Attribute

Allgemein angenommen, dass Programmierstil mit den zu erwartenden Softwarewartungskosten korreliert.

Aussagen über die Qualität eines Softwareproduktes (externe Attribute)

- Fehleranfälligkeit
- Wartungsaufwand
- Kosten

Annahme:
interne Attribute sind mit externen Attributen korreliert!

These

- komplexes Programm ⇒ hohe Kosten
- klares Programm ⇒ geringere Kosten

Grundsatz

„When you can measure what you are speaking about and express in numbers you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.“

Lord Kelvin

Softwaremetriken

Ziel Metriken (Maßzahlen) zu finden, die den Aufbau, Struktur, Stil eines *Moduls* (Programmstücks) bewerten und vergleichen lassen.

Anforderungen

- Gültigkeit
  - Misst tatsächlich was es vorgibt zu messen
- Einfachheit
  - Resultate sind einfach verständlich und interpretierbar
- Sensitivität
  - Reagiert ausreichend auf unterschiedliche Ausprägungen
- Robustheit
  - Reagiert nicht auf im Zusammenhang uninteressante Eigenschaften

Messung der strukturellen Komplexität

Fokus der Messung ist das Software-Modul

- Definition schwierig – Kann Funktion, Methode, Klasse, etc. sein

Die *Intra-modulare Komplexität* beschreibt die Komplexität eines einzelnen SW Moduls

- *Modul Komplexität* (intern)
  - LOC, Line-of-codes (simpel)
  - NCSS, non commenting source statements
  - McCabe (zyklomatische Komplexität), …
- *Kohäsion* (extern)
  - Henry-Kafura Metrik (Informationsfluss), …

Die *Inter-modulare Komplexität* beschreibt Komplexität zwischen Moduln - *Kupplung*

- Fenton und Melton, …

Zusammenhang Kupplung und Kohäsion

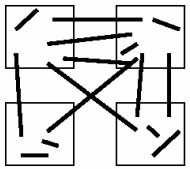
Kupplung

- Misst Komplexität der Beziehungen zwischen Moduln

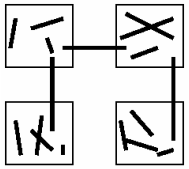
Kohäsion

- Misst Informationsfluss von und nach Außen abhängig von der Modulgröße

Meist Beziehung zwischen Kupplung und Kohäsion



Starke Kupplung
Schwache Kohäsion



Schwache Kupplung
Starke Kohäsion
(üblicherweise das Ziel guter SW-Entwicklung)

1.3.4.1 Intra-modulare Komplexität – Metrik von McCabe

Die *Metrik von McCabe* ist ein Maß zur Beurteilung der Modul Komplexität

- Basiert auf der *zyklomatischen Komplexität* V = die Anzahl der unabhängigen Pfade in einem Programmgraph

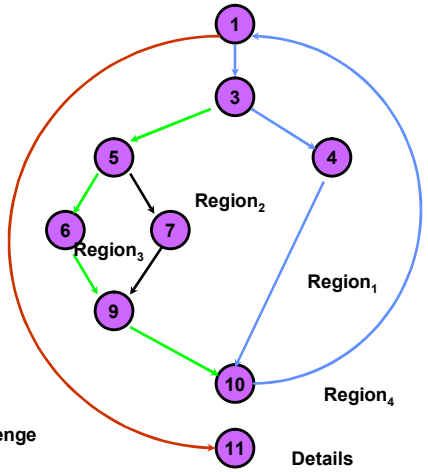
Bei einem unabhängigen Pfad wird mindestens eine 'neue' Kante im Programmablaufplan beschritten.

Erfahrungswerte für die zyklomatische Komplexität

V(G)	Einschätzung im Normalfall
< 5	einfach
5-10	normal
> 10	komplex, sollte restrukturiert werden
> 20	schwer verständlich, wahrscheinlich fehlerhaft

Programmbeispiel

```
0:  private sub foo (a as integer)
1:    while a < limit do
2:      process_1 a
3:      if bar(a) then
4:        process_2 a
5:      elseif mumble(a) then
6:        process_3 a
7:      else
8:        process_4 a
9:      end if
10:   end while
11: end sub
```



Basis Menge
Pfad 1: 1
Pfad 2: 1,3,4,10,1,11
Pfad 3: 1,3,5,6,9,10,1,11
Pfad 4: 1,3,5,7,9,10,1,11

Details
Kanten = 11
Knoten = 9
Bedingungsknoten = 3

Zyklomatische Komplexität

Mehrere Berechnungsmöglichkeiten

1. Die Anzahl der Regionen im Programmgraph G
2. V(G) = E - N + 2 (E = Anz. d. Kanten, N = Anz. d. Knoten)
3. V(G) = P + 1 (P = Anzahl der binären Bedingungsknoten)

Zyklomatische Komplexität des Beispiels

1. Regionen = 4
2. V(G) = 11 - 9 + 2 = 4
3. V(G) = 3 + 1 = 4

In unserem Beispiel ist die magische Zahl 4, d.h. „einfaches“ Programm (Metrik McCabe < 5)

1.3.4.2 Intra-modulare Komplexität – Henry-Kafura Metrik

Maß zur Bestimmung der Kohäsion

Beschreibt die funktionale Stärke des Moduls; zu welchem Grad die Modulkomponenten dieselbe Aufgabe erfüllen

Die *Henry-Kafura Metrik* (Sallie Henry and Dennis Kafura) basiert auf Zusammenhang zwischen Modulkomplexität und Verbindungskomplexität zwischen Moduln

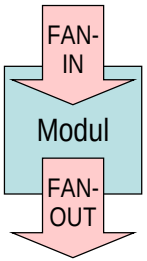
Maß für Modulkomplexität

- LOC
- NCSS
- McCabe

Verbindungskomplexität

Quantifizierung des lesenden und verändernden Zugriffs des Moduls auf die Umgebung

Zählt die Datenflüsse zwischen Moduln



$FAN\text{-}IN_m$: „Anzahl der Module die m verwenden“
genauer: Anzahl der Datenflüsse, die im Modul m terminieren + Anzahl der Datenstrukturen, aus denen der Modul m Daten ausliest

$FAN\text{-}OUT_m$: „Anzahl der Module die m verwendet“
genauer: Anzahl der Datenflüsse, die vom Modul m ausgehen + Anzahl der Datenstrukturen, die der Modul m verändert

Henry-Kafura Formel

$C_{im} * (FAN\text{-}IN_m * FAN\text{-}OUT_m)^2$

C_{im} … Modulkomplexität (z.B. LOC, McCabe, …)

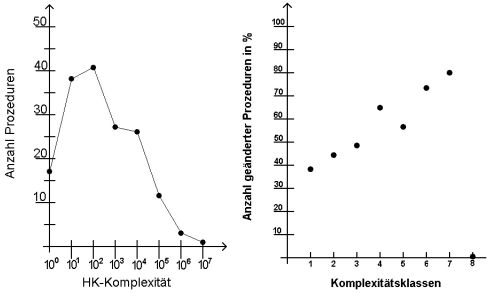
Anwendung

Von Henry und Kafura auf Unix Code angewendet

- Annahme: Zusammenhang zwischen Komplexität und Änderungshäufigkeit (Fehlerkorrektur) einer Prozedur
- 165 Prozeduren untersucht, Patch-Information aus Newsgroups
- Annahme bestätigt: Änderungen nehmen mit Komplexitätsklasse zu

Probleme HK

- Abh. vom Datenfluss, bei einem FAN = 0, gesamt 0 auch bei hoher Modulkomplexität
- Wiederverwendbarkeit wird durch hohen FAN Wert „bestraft“



1.3.4.3 Inter-modulare Komplexität – Fenton und Melton Metrik

Fenton und Melton Metrik ist ein Maß zur Bestimmung der Kupplung, d.h. die Unabhängigkeit zwischen Moduln

- Globale Kupplung eines Programms wird abgeleitet aus den Kupplungswerten zwischen allen möglichen Modulpaaren

Kupplungstypen

- Binäre Relationen definiert auf Paaren von Moduln x, y
  - Nach dem Grad der „Unerwünschtheit“ geordnet
- 0. No coupling: keine Kommunikation zwischen x und y
- 1. Data coupling: Kommunikation über Parameter (Daten)
- 2. Stamp coupling: akzeptieren selben Record-Typ als Parameter
- 3. Control coupling: Kommunikation über Parameter (Kontrolle)
- 4. Common coupling: Zugriff auf dieselbe globale Datenstruktur
- 5. Content coupling: x greift direkt auf interne Struktur von y zu (ändert Daten, Anweisungen)

Berechnung der Kupplung

Fenton-Maß für die Kupplung zwischen 2 Moduln

c(x,y) = i + n/(n+1)

- i ist der schlechteste Kupplungstyp zwischen Modul x und y
- n ist die Anzahl der „Kupplungen“ vom Typ i

Maß C(S) für die globale Kupplung eines Systems S bestehend aus n Moduln $D_1, \ldots, D_n$

C(S) = Median der Menge der Kupplungswerte aller Modulpaare

1.3.5 Laufzeitkomplexität

Die *Laufzeitkomplexität* liefert Aussagen über das Laufzeitverhalten von Algorithmen in Abhängigkeit von der Problemgröße

Ziel

- Algorithmen zu vergleichen

Ansatz

- Messen der Ausführungszeit der einzelnen Anweisungen
- Bestimmen, wie oft jede Anweisung beim Programmablauf ausgeführt wird
- Summe berechnen

Problem

- Ausführungszeiten abhängig von Maschinen- bzw. Systemarchitektur, Übersetzungsqualität des Compilers, etc.

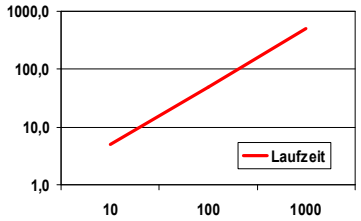
Summe-Beispiel: simpler Ansatz

```
int sum(int n)
  int s = 0;
  int i = 1;
  while(i <= n) {
    s = s + i;
    i = i + 1;}
  return s;
}
```

Zeit in msec

$T_1 \Rightarrow 0.1$
$T_2 \Rightarrow 0.1$
$T_3 \Rightarrow 0.3$
$T_4 \Rightarrow 0.1$
$T_5 \Rightarrow 0.1$
$T_6 \Rightarrow 0.1$

Gemessene Zeiten im Programm



Berechnung der Laufzeit

$f_{sum}(n) = T_1+T_2+n*(T_3+T_4+T_5)+T_3+T_6$
$f_{sum}(10) = 0.1+0.1+10*(0.3+0.1+0.1)+0.3+0.1 = 5.6$
$f_{sum}(100) = 0.1+0.1+100*(0.3+0.1+0.1)+0.3+0.1 = 50.6$
$f_{sum}(1000) = 0.1+0.1+1000*(0.3+0.1+0.1)+0.3+0.1 = 500.6$
$f_{sum}(n) = 0.6 + n*(0.5)$

Probleme des simplen Ansatzes

Entspricht einem Ansatz des „Ausprobierens“

Nur punktuell möglich
- bestimmte Problemgröße, Datenverteilung

Sonderfälle problematisch

Systemabhängig
- Hardware, Prozessor, ...
- Betriebssysteme, Compiler, Bibliotheken, ...

Lastabhängig

Frage schwer zu beantworten, ob „graduelle“ oder „grundsätzliche“ Verbesserung erreichbar

Ordnungsnotation

Grundprinzip
- Die exakten Werte der Ausführungszeiten sind uninteressant!

Über die *Ordnungsnotation* möchte man Aussagen treffen (siehe Ziel), dass Algorithmus A grob gesehen gleich schnell wie Algorithmus B ist.
- Durch Beschreiben der Laufzeiten von A und B über Funktionen f(n) und g(n) wird diese Fragestellung auf Vergleich dieser Funktionen zurückgeführt.

Man betrachtet das *asymptotische Wachstum* der Ausführungszeiten der Algorithmen bei wachsender Problemgröße n.

Big - O / W / Q - Notation

Big-O-Notation: Eine Funktion f(n) heißt von der Ordnung O(g(n)), wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \leq c_0 g(n)$ für alle $n > n_0$.
- liefert eine Obergrenze für die Wachstumsrate von Funktionen
- $f \in O(g)$, wenn f höchstens so schnell wie g wächst.
  - z.B.: $17n^2 \in O(n^2)$, $17n^2 \in O(2^n)$

Big-W-Notation: Eine Funktion f(n) heißt von der Ordnung $\Omega(g(n))$, wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \geq c_0 g(n)$ für alle $n > n_0$.
- liefert eine Untergrenze für die Wachstumsrate von Funktionen
- $f \in O(g)$, wenn f mindestens so schnell wie g wächst.
  - z.B.: $17n^2 \in \Omega(n^2)$, $2^n \in \Omega(n^2)$, $n^{37} \in \Omega(n^2)$

Q-Notation: Das Laufzeitverhalten ist $\Theta(n)$ falls gilt: $f(n)=O(n)$ und $f(n)=\Omega(n)$ (beschreibt das Laufzeitverhalten exakt)

Summen-Beispiel

Laufzeitschätzung: $f_{sum}(n) = T_1+T_2+n*(T_3+T_4+T_5)+T_3+T_6$

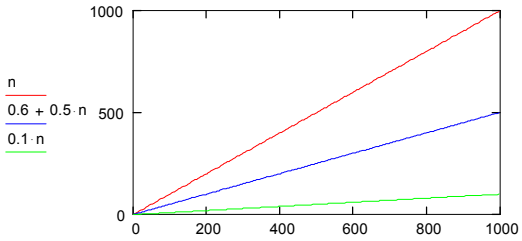
Messung: $T_1=0.1$, $T_2=0.1$, $T_3=0.3$, $T_4=0.1$, $T_5=0.1$, $T_6=0.1$

reale Werte, in der Implementierung gemessen

ergibt $f_{sum}(n) = 0.6+n*(0.5)$

$g(n)=n \Rightarrow$ Untergrenze: $0.1*g(n)$ Obergrenze: $h(n)=1*g(n)$

eine von vielen möglichen Grenzen



daraus folgt bezüglich de

$f_{sum}(n) \in O(n)$ und weiters $f_{sum}(n) \in \Omega(n)$, d.h. $f_{sum}(n) \in \Theta(n)$.

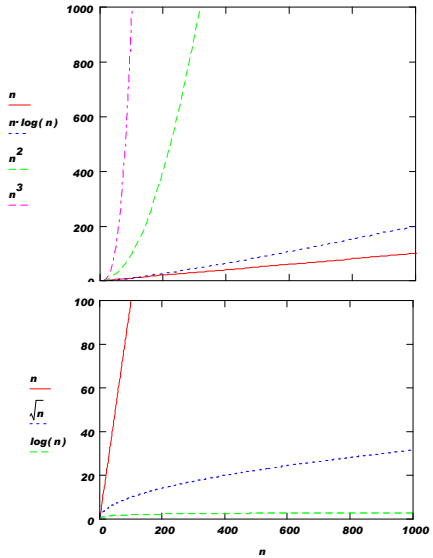
Die daraus für unser Beispiel ableitbare Aussage lautet, dass die Laufzeit des Algorithmus direkt proportional zur Problemgröße n ist

Laufzeitvergleich (1)

Laufzeitenvergleich

Annahme vorgegebene Problemgröße und unterschiedliches Laufzeitverhalten

Ordnung	Laufzeit
log n	1.2×10^{-5} sec
$\sqrt{n}$	3.2×10^{-4} sec
n	0.1 sec
n log n	1.2 sec
n $\sqrt{n}$	31.6 sec
n^2	2.8 h
n^3	31.7 a
2^n	über 1 Jahrhundert

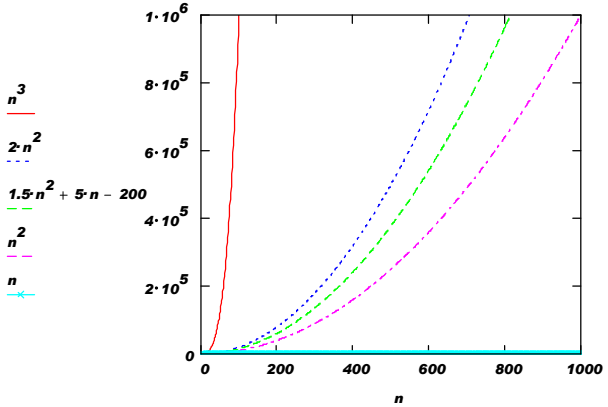


Laufzeitvergleich (2)

Beschreibung des Laufzeitverhaltens

durch obere O(n) und untere Schranke $\Omega(n)$

z.B.: $T(n) = 1.5n^2 + 5n - 200 \Rightarrow O(n^2)$, da $n^2 \leq T(n) \leq 2n^2$



Analyseansatz

Prinzipien der mathematischen Analyse

Definition der Problemgröße n

Finden eines geeigneten Problemparameters der die Problem-größe beschreibt. Dieser charakterisiert die Belastung (= f(n))

worst-case versus *average-case*

worst-case: Verhalten im schlechtesten Fall, maximal zu erwartender Aufwand

average-case: ∅–Verhalten, Erwartungswert des Aufwandes

Laufzeitanalyse über O(n) und $\Omega(n)$

oft schwierig zu bestimmen

Ignoriere konstante Faktoren

Merke nur höchste Potenzen

Praktische Laufzeitanalyse

Ignoriere konstante Faktoren

Konstante Werte werden auf den Faktor 1 reduziert, d.h.

$T(n) = 13 * n \Rightarrow O(n)$

$T(n) = \log_2(n) \Rightarrow O(\log(n))$, da $\log_x(n) = \log_a(n) / \log_a(x)$

Merke nur höchste Potenz

Ignoriere alle anderen Potenzen außer der höchsten

$T(n) = n^2 - n + 1 \Rightarrow O(n^2)$

Daher in Kombination:

$T(n) = 13*n^2 + 47*n - 11*\log_2(n) + 50000 \Rightarrow O(n^2)$

Beispiel: Laufzeitanalyse 'Bubblesort'

```
void bubble(Element v[], int n) {
  int i, j;                                  T1
  for(i = n-1; i >= 1; i--)                  T2
    for(j = 1; j <= i; j++)                  T3
      if(v[j-1] > v[j])                      T4
        swap(v[j-1], v[j]);                  T5
}
```

Problemgröße n

Informeller Ansatz:

Rekurrenzgleichung finden

Annahme $T_1 = ... T_5 = 1$

$T(n) = T_1 + (n-1)*(T_2 + (n-1)*(T_3+T_4+T_5))$

$T(n) = 1+(n-1)*(1+(n-1)*(1+1+1)) = 3n^2-5n-3$

Ignoriere konstante Faktoren

$T(n) = n^2-n-1$

Merke höchste Potenz

$T(n) = O(n^2) = \Omega(n^2) = \Theta(n^2)$

worst-case = best-case = average case

Rekurrenzen

Einfache Rekursion

```
int sum(int n) {
  if(n <= 0) return 0;
  if(n == 1) return 1;
  return n + sum(n-1);
}
```

T_1
T_2
$T_3+T(n-1)$

$T(n) = T_1+T_2+T_3+T(n-1)$
$T(n) = T(n-1) + 1$ $\quad T(0)=T(1)=1$
$T(n) = T(n-2) + 1 + 1$
…
$T(n) = \underbrace{1 + \ldots + 1 + 1}_{n}$
$T(n) = O(n)$

Einige wichtige Rekurrenzen

$T(n) = \mathbf{T(n-1)+n} \Rightarrow$
$\Rightarrow T(n) = n*(n+1)/2 = \mathbf{O(n^2)}$

$T(n) = \mathbf{T(n/2)+1} \Rightarrow T(n) = \text{ld } n = \mathbf{O(\log n)}$

$T(n) = \mathbf{T(n/2)+n} \Rightarrow T(n) = 2*n = \mathbf{O(n)}$

$T(n) = \mathbf{2*T(n/2)+n} \Rightarrow$
$\Rightarrow T(n) = n*\text{ld } n = \mathbf{O(n*\log n)}$

$T(n) = \mathbf{2*T(n/2)+1} \Rightarrow T(n) = 2*n = \mathbf{O(n)}$

Algorithmische Lücke

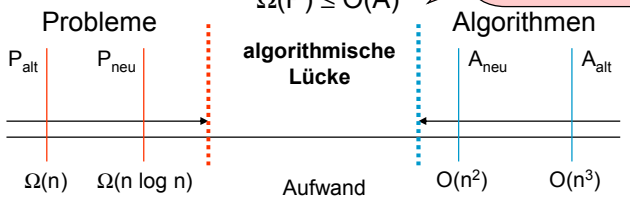
Eine *algorithmische Lücke* für ein Problem existiert, falls für die bekannten problemlösenden Algorithmen A gilt, dass ihr Aufwand größer als der ableitbare Lösungsaufwand für das Problem P ist.

O(A) dient zur Beschreibung des Algorithmus

Ω(P) dient zur Beschreibung des Problems

$\Omega(P) \leq O(A)$

Ziel: die algorithmische Lücke zu schließen, d.h. Ω(P) = O(A)



Geschlossene Lücke

Suchen in sortierter Sequenz, T(A) = O(log(n)) = Ω(log(n))

Sortieren, T(A) = O(n*log(n)) = Ω(n*log(n))

Offene Lücke

Graphenisomorphie, $T(A_{naiv})$ aus O(|V|!)

1.3.6 Laufzeitanalyse von rekursiven Programmen

Die Laufzeitanalyse von rekursiven Programmen ist meist nicht-trivial

- Bis jetzt nur taxativ gelöst

Laufzeitverhalten eines rekursiven Programms lässt sich durch eine Rekurrenz beschreiben

Beispiel: Mergesort

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

wobei folgende Lösung angegeben wurde

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

Lösungsansätze

Es existieren unterschiedliche Lösungsansätze, wie z.B.

Substitutionsmethode

- Erraten einer asymptotischen Grenze und Beweis dieser Grenze durch Induktion

Iterationsmethode

- Umwandlung der Rekurrenz in eine Summe und Anwendung von Techniken zur Grenzwertberechnung von Summen

Mastermethode

- Liefert Grenzen für Rekurrenzen der Form
  T(n) = aT(n/b) + f(n), wobei a ≥ 1, b > 1 und f(n) ist eine gegebene Funktion

Master Theorem

Das *Master Theorem* stellt ein „Kochrezept“ zur Bestimmung des Laufzeitverhaltens dar

- Vereinfachte Form (generelle Version Cormen et al. pp. 62)

Es seien a ≥ 1, b ≥ 1 und c ≥ 0 Konstante

T(n) ist definiert durch $aT(n/b) + \Theta(n^c)$
wobei n/b entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ ist

T(n) besitzt dann den folgenden asymptotischen Grenzwert

- Fall 1: wenn $c < \log_b a$ dann $T(n) = \Theta(n^{\log_b a})$
- Fall 2: wenn $c = \log_b a$ dann $T(n) = \Theta(n^c \log n)$
- Fall 3: wenn $c > \log_b a$ dann $T(n) = \Theta(n^c)$

Beispiel

Binäres Suchen

```
int bs (int x, int z[], int l, int r) {
  if (l > r) // Schwelle, kein Element vorhanden
    return -1; // 0 verboten, da gültiger Indexwert!
  else {
    int m = (l + r) / 2;
    if (x == z[m])      // gefunden!
      return m;
    else if (x < z[m])
      return bs(x, z, l, m-1);
    else // x > z[m]
      return bs(x, z, m+1, r);
  }
}
```

Fall 1: wenn $c < \log_b a$ dann $T(n) = \Theta(n^{\log_b a})$
Fall 2: wenn $c = \log_b a$ dann $T(n) = \Theta(n^c \log n)$
Fall 3: wenn $c > \log_b a$ dann $T(n) = \Theta(n^c)$

Laufzeit: $T(n) = T(n/2) + 1 = T(n/2) + \Theta(1)$
a = 1, b = 2, c = 0
Fall 2, da $c = \log_b a \Rightarrow 0 = \log_2 1$, ergibt $T(n) = \Theta(\log n)$

Weitere Beispiele

Fall 1: wenn $c < \log_b a$ dann $T(n) = \Theta(n^{\log_b a})$
Fall 2: wenn $c = \log_b a$ dann $T(n) = \Theta(n^c \log n)$
Fall 3: wenn $c > \log_b a$ dann $T(n) = \Theta(n^c)$

Beispiel: Mergesort

$T(n) = 2T(n/2) + n = 2T(n/2) + \Theta(n)$

$a = 2, b = 2, c = 1$

Fall 2, da $c = \log_b a \Rightarrow 1 = \log_2 2$, ergibt $T(n) = \Theta(n \log n)$

Beispiel: $T(n) = 4T(n/2) + n = 4T(n/2) + \Theta(n)$

$a = 4, b = 2, c = 1$

Fall 1, da $c < \log_b a \Rightarrow 1 < \log_2 4$, ergibt $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Beispiel: $T(n) = T(n/2) + n = T(n/2) + \Theta(n)$

$a = 1, b = 2, c = 1$

Fall 3, da $c > \log_b a \Rightarrow 1 > \log_2 1$, ergibt $T(n) = \Theta(n^1) = \Theta(n)$

Was nehmen wir mit?

Algorithmenparadigmen
- Greedy, Divide and Conquer, Dynamic Programming

Analyse und Bewertung von Algorithmen
- Effektivität, Korrektheit, Termination

Strukturkomplexität
- McCabe, Henry-Kafura, Fenton

Laufzeitkomplexität
- Ordnungsnotation, Algorithmische Lücke, Master Theorem

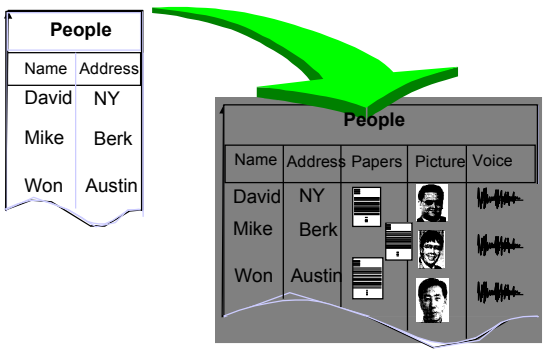
Kapitel 2
Datenstrukturen

2.1 Situation
(Jim Gray, 97)

Datenstrukturen (Datenbanken) speichern ALLE Daten

The New World:
- Milliarden von Objekten
- Große Objekte (1MB)

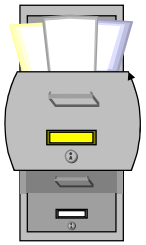
The Old World:
- Millionen von Objekten
- 100-Byte Objekte

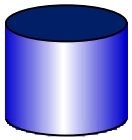


Paperless office
Library of congress online
All information online
- entertainment
- publishing
- business

Information Network,
Knowledge Navigator,
Information at your fingertips

Magnetische Platten sind billiger als Papier *(Jim Gray, 97)*





Dateischrank: cabinet (4 drawer) 250$
paper (24,000 sheets) 250$
space (2x3 @ 10$/ft2) 180$
total 700$
3 ¢/sheet

Platte: disk (4 GB =) 500$
ASCII: 2 m pages
(100x cheaper) 0.025 ¢/sheet

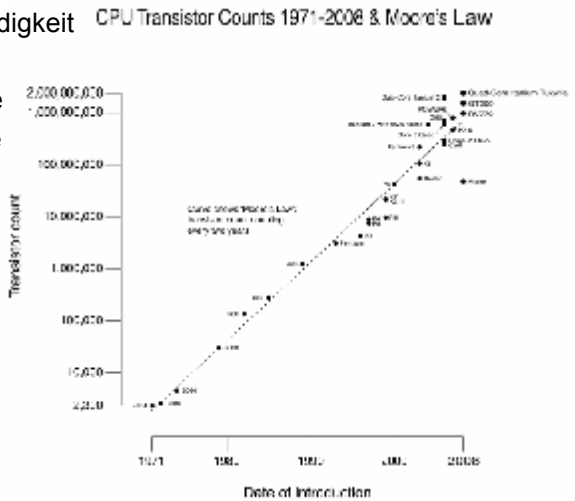
Image: 200 k pages
(10x cheaper) .25 ¢/sheet

Conclusio: Speichere alles auf Platten

Moore's Law

XXX verdoppelt sich alle 18 (24) Monate
60% Steigerung pro Jahr

- Micro Prozessor Geschwindigkeit
- Chip Dichte
- Magnetische Platten Dichte
- Kommunikationsbandbreite
  - WAN Bandbreite nähert sich LAN
- Plattenzugriffs-geschwindigkeit ???



Beispiele (1)

Magellan Projekt

- Satellit umkreiste die Venus, Radarabtastung zur Oberflächendarstellung
- Sandte 3 Terabyte Daten
- Rendering der Daten benötigt 13 Gigabyte / sec
  - Zur Zeit technisch nicht durchführbar!

Wettervorhersage

- Zirkulationsmodelle der Atmosphäre und der Ozeane
  - 1000 Jahre Simulation, 150 km^2 Auflösung, 0.2 simulierte Jahre / Maschinenstunde
  - Ein Durchlauf auf Intel Touchstone Delta 57 Wochen
- 40 MB Daten /Simulationsminute = 20 Terabytes

Beispiele (2)

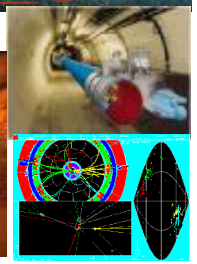
CERNs Herausforderung: Datagrid

Neuer Beschleuniger LHC mit 4 Detektoren

- Large Hadron Collider, 14 TeV
- Ziele: Suche nach Higgs Boson und Graviton (et al.)
- Start 2007

Ziele

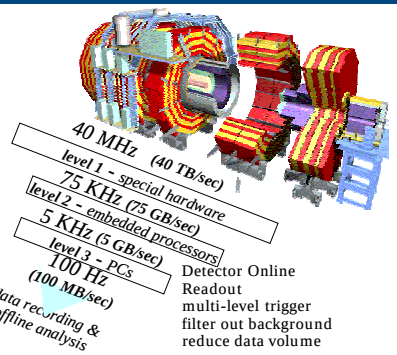
- Weltweiter Zugriff auf die Daten
  - CERN und Regional Centers (Europa, Asien, Amerika)
  - 2000 Benutzer
- Riesige Datenvolumen
- Daten Semantik
- Performance und throughput



Gigantische Datenmengen

Charakteristische Größen (2005)

- 1-6 (vielleicht 100 ?) Petabyte / Jahr
  - Zeitraum 15 bis 20 Jahre
- 10500 Knoten
- 2.1 Petabyte Platten Platz
- 340 Gigabyte IO Bandbreite
- Tapes



Größe: Was ist ein Petabyte?

1 Petabyte = 2 hoch 50, i.e.
(1,125,899,906,842,624) bytes ~ 10^{15} bytes

1,000,000,000,000 business letters
100,000,000,000 book pages
50,000,000,000 FAX images
10,000,000,000 TV pictures (mpeg)
4,000,000 LandSat images

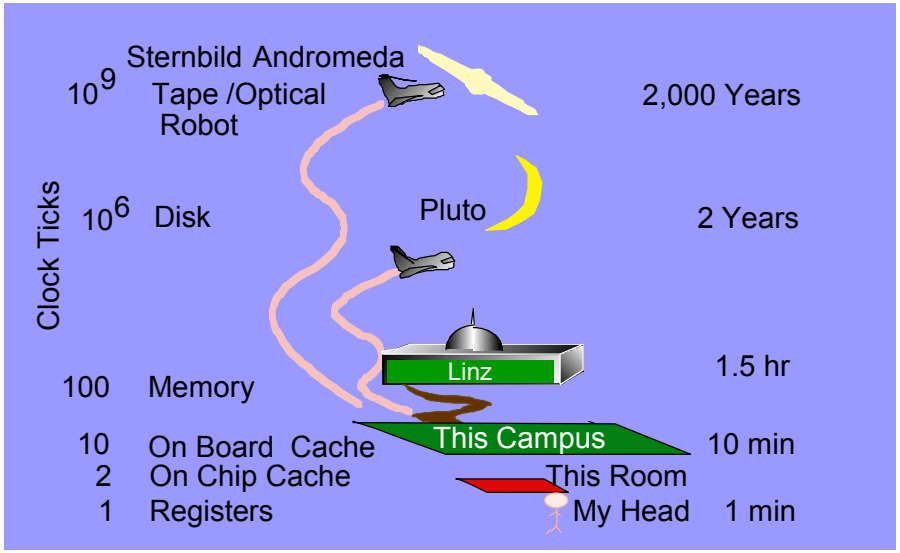
150,000 miles of bookshelf
15,000 miles of bookshelf
7,000 miles of bookshelf
10,000 days of video

Library of Congress (in ASCII)
enthält 0.025 Petabyte

Aktuelle und zukünftige Projekte generieren weit mehr Daten
Auf uns warten Exa-, Zeta-, Yotta Byte !!!

Geschwindigkeit: Speicherzugriffszeiten

Wie weit sind die Daten entfernt ? (Jim Gray, 97)



2.2 Motivation

Beispiel: 100 Telefonnummern zu verwalten

Ein „Haufen“ Zettel mit Namen und Nummern

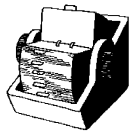
- Finden einer Telefonnummern durch sequentielle Suche
- benötigt im Durchschnitt 50 „Zugriffe“

Zettel nach dem Namen sortiert

- Suche durch binäres Aufteilen („in die Mitte, Schlüssel-vergleich und dann links oder rechts davon weitersuchen“)
- Ungefähr ld 100 ≈ 7 Zugriffe

Rolodex

- Zettel sortiert, in Ordnern und mit Namensindex
- Ziel mit einem (1!) Zugriff gewünschte Nummer



Ziele

Information “effizient” zu verwalten!

Was bedeutet “effizient”?

- Quantitative Ziele
  - Zugriffszeit
    - schnelles Einfügen, Verändern, Löschen, ... (d.i. “Bearbeiten” im weitesten Sinn) der Daten
  - Speicherplatz
    - kompaktes Speichern der Information
- Qualitative Ziele
  - Unterstützung spezifischer Zugriffsarten auf Eigenschaften bzw. Charakteristiken der Daten

Datenstrukturen

Erfüllung dieser Ziele führte zur Entwicklung von

Datenstrukturen

Datenstrukturen dienen zur Verwaltung großer Mengen ähnlicher Objekte

- Unterschiedliche Datenstrukturen dienen zur Verwaltung unterschiedlicher Objekte, die durch unterschiedliche Eigenschaften charakterisiert sind, daraus folgt:

Für unterschiedliche Problemstellungen unterschiedliche Datenstrukturen!

Beispiel: Telefonbuchverwaltung

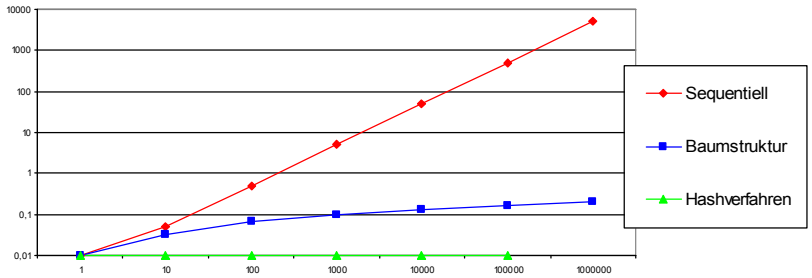
Suche in einem Telefonbuch mit 2000000 Einträgen

- Annahme: Zugriff auf einen Datensatz in 0,01 ms

Ansätze (Zeit für einen Zugriff)

- Sequentielles Suchen (im Mittel 1000000 * 0,01 ms = 10 s)
- Baumstruktur (ungefähr ld 2000000 * 0,01 = 0,21 ms)
- Hashverwaltung (1 Zugriff = 0,01 ms)

Zugriffzeit im Verhältnis zur Dateigröße



2.3 Überblick

Alle bekannten Datenorganisationsformen bauen auf einigen wenigen einfachen Techniken auf

Sequentielle Techniken

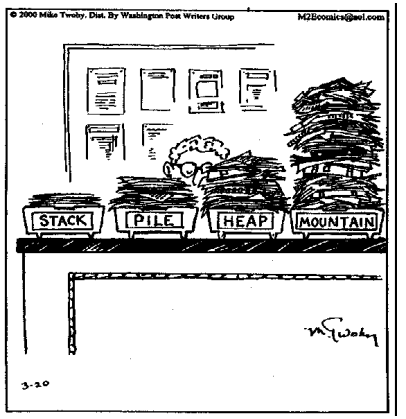
- Listen
- Stack, Queue

Hashverfahren

- Dictionary, Hash Tabelle, Kollisionsverfahren

Baumstrukturen

- Binärer Baum, B+-Baum, Priority Queue



Vektor

Ein Vektor (vector, Feld) verwaltet eine fix vorgegebene Anzahl von Elementen eines einheitlichen Typs.

- Zugriff auf ein Element über einen ganzzahligen Index (die Position im Vektor)
- Aufwand des Zugriffes für alle Elemente konstant

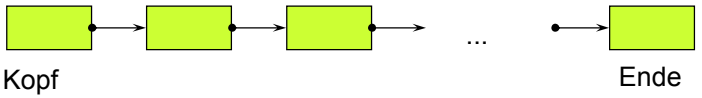
Anwendungen

- Verwaltung fix vorgegebener Anreihungen, Strings, math. Konzepte

Liste

Eine Liste (list) dient zur Verwaltung einer beliebig Anzahl von Elementen eines einheitlichen Typs.

- Die Elemente werden in einer Sequenz angeordnet, die sich (meist) aus der Eintragereihenfolge ableiten lässt (ungeordnet).
- Der Aufwand des Zugriffes auf ein einzelnes Element ist abhängig von der Position in der Sequenz.



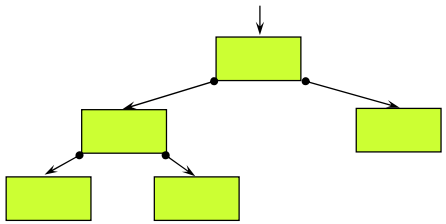
Anwendungen

- sequentielle Datenbestände, große Datenmengen, externe Speicherung

Baum

Der Baum (tree) stellt eine Generalisierung der Liste auf eine 2-dimensionale Datenstruktur dar.

- besteht aus Knoten und Kanten
- Exponentieller Zusammenhang zwischen Tiefe des Baumes und Anzahl der Knoten
- Anwendungen: allgemeine Schlüsselverwaltung, Haupt- und Externspeichermanagement



Vergleichskriterien

Dynamik

- Datenverwaltung
  - Einfügen, Löschen
- Datenmenge
  - beliebige oder fixe Anzahl von Elementen

Aufwand

- Laufzeit der Operationen
- Speicherplatzverbrauch

Modell

- Operationenumfang

Informeller Vergleich

Datenstruktur	Stärken	Schwächen
Vektor	dynamische Verwaltung direkter Elementzugriff konstanter Aufwand der Operationen geringer Speicherplatz	oft statisch (nur beschränkte Datenmenge) eingeschränkte Operationen
Liste	dynamische Verwaltung beliebige Datenmenge klares Modell	linearer Aufwand der Operationen simples Modell
Baum	meist dynamische Verwaltung beliebige Datenmenge logarithmischer Aufwand der Operationen	Balanzierungsalgorithmen relativ hoher Speicherplatzverbrauch komplexes Modell manchmal nur Einfüge-Operation unterstützt

Was nehmen wir mit?

Datenorganisation
- Effizienz
- Quantität - Qualität

Datenstrukturen
- Typen
- Vergleichskriterien

Kapitel 3
Vektoren

3 Vektoren

Ein Vektor (Feld, array) verwaltet eine fix vorgegebene Anzahl von Elementen eines einheitlichen Typs.

- Zugriff auf ein Element über einen ganzzahligen Index (die Position im Vektor)
- Aufwand des Zugriffes für alle Elemente konstant

Methoden
- Hashing: Datenorganisationsform
- Sortieren: Datenreorganisationsmethoden

3.1 Dictionary

Ein *Dictionary* ist eine Datenstruktur, bei der die einzelnen Elemente jeweils aus einem *Schlüssel-* (key) und einem *Information*-Teil (info) bestehen

Unterstützt nur die einfachen Operationen des Einfügens, Löschens und der Suche zur Verfügung stellt.

Beispiele sind

Wörterbücher, Namenslisten, Bestandslisten, etc.

Vektoren sind zur Realisierung von Dictionaries gut geeignet

Struktur

key_0	key_1	key_2	…	key_{n-1}
$info_0$	$info_1$	$info_2$		$info_{n-1}$

Beispiele

Wörterbuch

(Ausschnitt)

Schlüssel	Information
computable	berechenbar
computation	Berechnung
compute	(be)rechnen
computer	Rechenautomat

Bestandsliste

Schlüssel	Information
1	CPU
4	Bildschirm
17	Tastatur
25	Maus

Operationen

Construct

Erzeugen eines leeren Dictionary

IsEmpty

Abfrage auf leeres Dictionary

Insert

Einfügen eines Elementes

Delete

Löschen eines Elementes

LookUp

Abfrage eines Elementes über seinen Schlüssel und liefern der Information

Klasse Dictionary

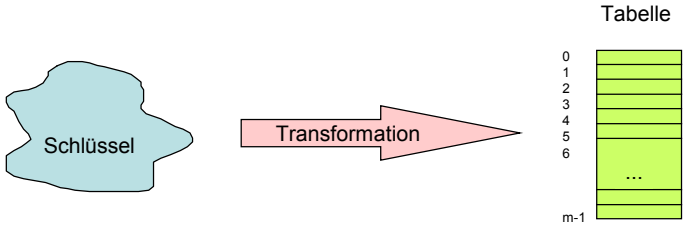
```
class Dictionary {
  private:
    node { KeyType Key; InfoType Info; };
    node *Dict;
    int NumberElements;
  public:
    Dictionary(int max) {
      Dict = new node[max];
      NumberElements = 0;
    }
    ~Dictionary() { delete Dict; }
    void Insert(KeyType k, InfoType I);
    void Delete(KeyType k);
    InfoType LookUp(KeyType k);
};
```

3.2 Hashing

Hashing ist eine Methode Elemente in einer Tabelle direkt zu adressieren, in dem ihre Schlüsselwerte durch arithmetische Transformationen direkt in Tabellenadressen (Indizes) übersetzt werden

Mit anderen Worten, der Index (die Position in der Tabelle) eines Elements wird aus dem Schlüsselwert des Elements selbst berechnet.

Schlüssel → Adresse (Index)



Hashing, allgemein

Ansatz

Menge K von n Schlüsseln $\{k_0, k_1, \ldots, k_{n-1}\}$

Hashtabelle T der Größe m

Randbedingung: n >> m

(Anzahl der möglichen Schlüsselwerte viel größer als Plätze in der Tabelle)

Transformation

Hash-Funktion h

$$h: K \rightarrow \{0, 1, \ldots m-1\}$$

Für jedes j soll der Schlüssel k_j an der Stelle $h(k_j)$ in der Tabelle gespeichert werden. Der Wert h(K) wird als Hash-Wert von K bezeichnet.

Wahl der Hashfunktion (eigentlich) beliebig!

Hashfunktion

Gewünschte Eigenschaften

Einfach und schnell zu berechnen.

Elemente werden gleichmäßig in der Tabelle verteilt.

Alle Positionen in der Tabelle werden mit gleicher Wahrscheinlichkeit berechnet.

Beispiele

Modulo Bildung, Schlüssel mod m

Man sollte darauf achten, daß m eine Primzahl ist, sonst kann es bei Schlüsseltransformationen (Strings) zu Clusterbildungen (Anhäufungen) durch gemeinsame Teiler kommen.

Bitstring-Interpretation

Teile aus der binären Repräsentation des Schlüsselwertes werden als Hashwert herangezogen.

Transformationstabellen

Beispiel Hashing

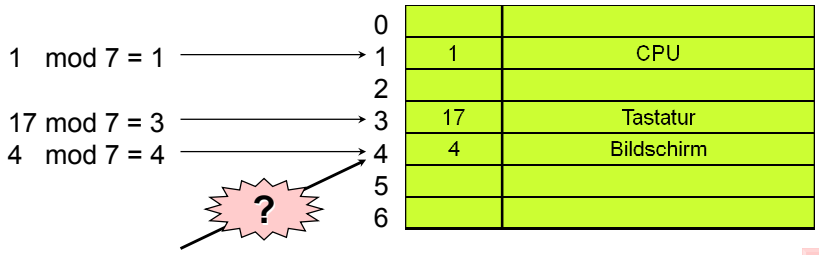
Bestandsliste

Größe der Liste 7

Hashfunktion h(k) = k mod 7

Datensätze

Schlüssel	Information
1	CPU
4	Bildschirm
17	Tastatur
25	Maus



25 mod 7 = **4** → Position in der Tabelle schon besetzt

Kollision

Kollision

Eine *Kollision* tritt auf, wenn zwei oder mehrere Schlüsselwerte auf dieselbe Tabellenposition abgebildet (*gehasht)* werden.

Dies bedeutet, dass für 2 Schlüssel k_i, k_j, mit $k_i \neq k_j$, gilt $h(k_i) = h(k_j)$.

- Diese Situation ist aber zu erwarten, da es viel mehr mögliche Schlüssel als Tabellenplätze gibt (n >> m als Randbedingung).
- Die Hashfunktion h ist i.A. nicht injektiv, d.h. aus h(x) = h(y) folgt nicht notwendigerweise x=y.

Kollisionsbehandlung

Eine Kollision löst eine notwendige *Kollisions-behandlung* aus.

D.h., für den kollidierenden Schlüsselwert muß ein Ausweichsplatz gefunden werden.

- Maßnahmen zur Kollisionsbehandlung sind meist recht aufwendig und beeinträchtigen die Effizienz von Hashverfahren.

der Kollisionspfad ist definiert durch die Ausweichplätze aller Elemente, die auf den gleichen Platz "ge-hasht" wurden

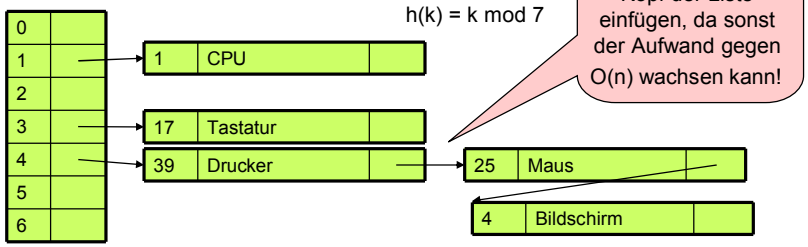
Wir betrachten nun 2 simple Kollisionsbehandlungen

Separate Chaining und *Double Hashing*

3.2.1 Separate Chaining

Beim Separate (Simple) Chaining wird die Kollisions-behandlung mit linearen Listen bewerkstelligt.

- Kollidierende Schlüsselwerte werden in einer linearen Überlaufkette (Liste) ausgehend vom ursprünglich gehashten Tabellenplatz gespeichert.
- Beispiel:

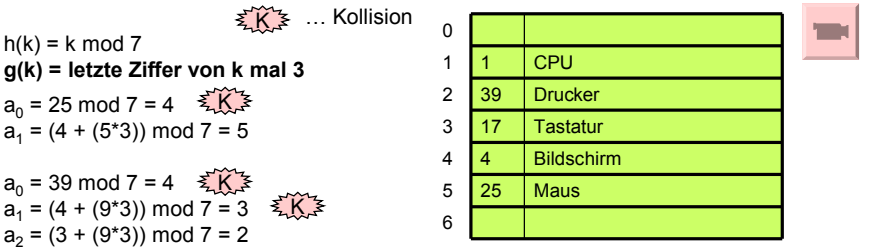


Der Eintrag des Elementes (39, Drucker) führt zu einer Verlängerung der Überlaufkette.

3.2.2 Double Hashing

Beim *Double Hashing* wird bei Kollision eine zweite, von h unabhängige, Hashfunktion zur Bestimmung einer alternativen Position verwendet.

- Überlauf auf Position $a_0 = h(k)$
- Bestimmung einer alternativen Position mit Kollisionsfunktion $g : K \rightarrow \{1, \ldots m-1\}$, z.B. $a_{i+1} = (a_i + g(k)) \bmod m$
- Beispiel



Linear Probing

Spezialfall des Double Hashing

Kollisionsbehandlung: Man verwendet den nächsten freien Platz

Bei Erreichen des Tabellenendes sucht man am Tabellenanfang weiter

Hashfuktionen: $a_0 = h(k)$, $a_{i+1} = (a_i + g(k)) \bmod m$

g(k) = 1

Beispiel

25 mod 7 = 4 K
(4+1) mod 7 = 5

39 mod 7 = 4 K
(4+1) mod 7 = 5 K
(5+1) mod 7 = 6

0		
1	1	CPU
2		
3	17	Tastatur
4	4	Bildschirm
5	*25*	*Maus*
6	*39*	*Drucker*

Suchen und Löschen in Hashtabellen

Suchen

Solange eine Zieladresse durch ein Element belegt ist, und der gesuchte Schlüsselwert noch nicht gefunden wurde, muß über den Kollisionspfad weitergesucht werden.

Separate Chaining: Verfolgen der linearen Liste

Double Hashing: Aufsuchen aller möglichen Plätze der Kollisionsfkt.

Löschen

Separate Chaining: Entfernen des Listenelements

Double Hashing: Positionen, an denen ein Element gelöscht wurde, müssen gegebenenfalls mit einem speziellen Wert (“wiederfrei”) markiert werden, damit der entsprechende Kollisionspfad für die restlichen Elemente nicht unterbrochen wird.

Eigenschaften

Generell für Hashverfahren

+ Aufwand beim Zugriff auf ein Element im besten Fall konstant, O(1), einfache Evaluation der Hashfunktion.
- Kollisionsbehandlung aufwendig, volle Hashtabelle führt zu vielen Kollisionen, daher (Faustregel) nie über 70% füllen.
- Kein Zugriff in Sortierreihenfolge.

Separate Chaining

+ Dynamische Datenstruktur, beliebig viele Elemente.
- Speicherplatzaufwendig (zusätzliche Zeigervariable).

Double Hashing

+ Optimale Speicherplatzausnützung.
- Statische Datenstruktur, Tabellengröße vorgegeben.
- Kollisionbehandlung komplex, „wiederfrei“ Markierung beim Löschen.

Analyse Hashing

Datenverwaltung

Einfügen und Löschen unterstützt

Datenmenge

beschränkt

abhängig von der Größe der vorhandenen Hashtabelle

Modelle

Hauptspeicherorientiert

Unterstützung simpler Operationen

keine Bereichsabfragen, keine Sortierreihenfolge

Analyse Hashing (2)

Speicherplatz	O(1)
Konstruktor	O(1)
Zugriff	O(1)
Einfügen	O(1)
Löschen	O(1)

gültig nur für reines Hashverfahren ohne Kollisions-behandlung

Bitte beachten: Aufwand von Hashing stark abhängig vom Kollisionsverfahren, geht aber oft gegen O(n)

3.3 Dynamische Hash-Verfahren

Dynamischen Hash-Verfahren versuchen im Falle von Kollisionen die ursprüngliche Hashtabelle zu erweitern

Anlegen von Überlaufbereichen (z.B. lineare Listen)

Vergrößerung des Primärbereichs (ursprüngliche Tabelle)

- Erfordert Modifikation der Hash-Funktion
- Ansatz: Familien von Hash-Funktionen
  - h_1: K $\rightarrow$ $addr_1$
  - ...
  - h_n: K $\rightarrow$ $addr_n$
  - Wobei $|addr_1| < |addr_2| < ... < |addr_n|$
- Ziel: Wechsel von $addr_i$ auf $addr_{i+1}$ erfordert minimale Reorganisation der Hash-Tabelle (d.h. Umspeichern von Daten minimieren)

Hash-Funktionen Schema

Simpler Ansatz

- $|addr_{i+1}| = 2 * |addr_i|$
- h(x) ist eine Hash-Funktion
- $h_i(x)$ seien die i „least significant bits“ von h(x)

Somit gilt

- $h_{i+1}(x) = h_i(x)$ oder
- $h_{i+1}(x) = h_i(x) + 2^i$
- Beispiel:
  - $addr_2$ = 0 .. 3,
  - $addr_3$ = 0 .. 7,
  - h(x) = x
  - 4 Adresswechsel

Wert	h(x)	binär	$h_2(x)$	$h_3(x)$
7	7	00111	11=3	111=7
8	8	01000	00=0	000=0
9	9	01001	01=1	001=1
10	10	01010	10=2	010=2
13	13	01101	01=1	101=5
14	14	01110	10=2	110=6
23	23	10111	11=3	111=7
26	26	11010	10=2	010=2

Dynamische Hashverfahren

„Two-Disk-Access“ Prinzip

- Finden eines Schlüssel mit maximal 2 Platten Zugriffen

Hashverfahren für externe Speichermedien

Linear Hashing

- Verzeichnisloses Hashverfahren
- Primärbereiche, Überlaufbereiche

Extendible Hashing

- Verzeichnis mit binärer Expansion
- Primärbereiche

Bounded Index Size Extendible Hashing

- Verzeichnis mit beschränkter Größe
- Binäre Expansion der Blöcke

3.3.1 Linear Hashing

W. Litwin 1980

Organisation der Elemente in Blöcken (Buckets)

- Analog zu B-Bäumen
- Blockgröße b

Idee

- Bei Überlauf eines Blocks (Splitting) wird der Primär- und Überlaufbereich um jeweils einen Block erweitert
- Primärbereich durch Hinzufügen eines Blocks am Ende der Hash-Tabelle
- Überlaufbereich durch lineare Liste

Splitting und Suche

Splitting wird „Runden"-weise durchgeführt

- Eine Runde endet, wenn alle ursprünglichen N Blocks (für Runde R) gesplittet worden sind
  - Blocks 0 bis NextToSplit-1 sind schon gesplittet
  - Nächster Block zum Splitten ist definiert durch Index NextToSplit
- Aktuelle Rundennummer ist definiert durch d

Suche

- Suche nach Wert x

```
a = h_d(x);
if(a < NextToSplit) a = h_{d+1}(x);
```

d=2, NextToSplit=1, b=3

000	8		
01	17	25	
10	34	50	2
11			
100	28		

2=000010
8=001000
17=010001
25=011001
28=011100
34=100010
50=110010

Einfügen

Einfügen

- Suche Block (h_d oder h_{d+1})
- Falls Block voll
  - Füge Element in einen Überlaufblock
    - Neues Element in der linearen Liste
  - Splitte Block NextToSplit und erhöhe NextToSplit um 1
- Beispiel: Einfügen von 6 (=000110)

d=2, NextToSplit=1, b=3

000	8		
01	17	25	
10	34	50	2
11			
100	28		

2=000010
8=001000
17=010001
25=011001
28=011100
34=100010
50=110010

d=2, NextToSplit=2, b=3

000	8			
001	17	25		
10	34	50	2	→ 6
11				
100	28			
101				

Beispiel: linear Hashing

Aktuelle $h_i(x) = h_3(x)$

d=3, NextToSplit=0, b=3

000	8		
001	17	25	
010	34	50	2
011			
100	28		
101	5		
110			
111	55		

2=000010 28=011100
5=000101 34=100010
8=001000 50=110010
17=010001 55=110111
25=011001

Einfügen: 16, 13, 21, 37

d=3, NextToSplit=0, b=3

000	8	16	
001	17	25	
010	34	50	2
011			
100	28		
101	5	13	21
110			
111	55		

16=010000
13=001101
21=010101
37=100101

Split

d=3, NextToSplit=1, b=3

0000	16		
001	17	25	
010	34	50	2
011			
100	28		
101	5	13	21
110			
111	55		
1000	8		

37

37

Überlaufblock

Bemerkungen

Die Rundennummer d wird erhöht, wenn das Splitting einmal durch ist, d.h.

```
if(NextToSplit == 2^d) { d++; NextToSplit=0; }
```

Expansionspolitik

- Bei jedem Überlauf Splitten nicht wirklich sinnvoll
- Salzberg schlägt vor Expansion durchzuführen, wenn seit der letzten Expansion genau L * b Datensätze eingefügt wurden
- L ist definiert durch den Belegungsfaktor, d.h
  L = Rohdatenvolumen / Volumen der Datenstruktur
- Lange Überlaufketten werden in jedem Fall vermieden

In der Praxis keine linearen Adressräume sondern Aufteilen des Primärbereiches auf mehrere Hash Dateien pro Platte

- Zuordnungssystem über Verwaltungsblöcke

3.3.2 Extendible Hashing

Hash-Verfahren mit Index

- R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, 1979

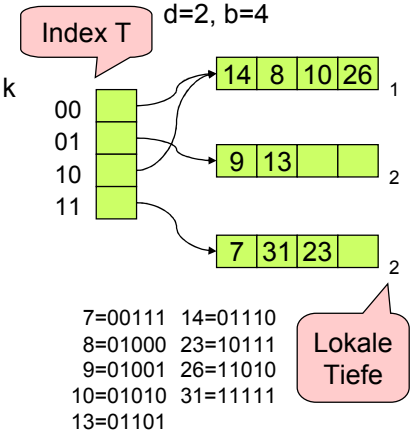
Idee

- Wenn Primärbereich zu klein wird, Vergrößerung durch Verdopplung
- Primärbereich wird über Index T verwaltet
- Bei Überlauf eines Blocks Split dieses Blocks und Verwaltung des Blocks durch Verdoppeln des Index
  - Index ist viel kleiner als die Datei, daher Verdoppeln viel günstiger
- Keine Überlaufbereiche

Struktur und Suche

Eigenschaften

- Jeder Indexeintrag referenziert genau einen Datenblock
- Jeder Datenblock wird von genau 2^k mit k aus N_0 Indexeinträgen referenziert
- Die Anzahl der Indexeinträge, die den Block referenzieren ist im Block lokal bekannt
  - Wird durch eine lokale Tiefe t verwaltet (Gegensatz globale Tiefe d für die gesamte Hash-Tabelle)
  - Anzahl der den Block referenzierenden Verzeichniseinträge entspricht 2^{d-t}



Suche

- Element x im Block $T[h_d(x)]$

Einfügen

Zwei Fälle zu unterscheiden falls ein Block überläuft

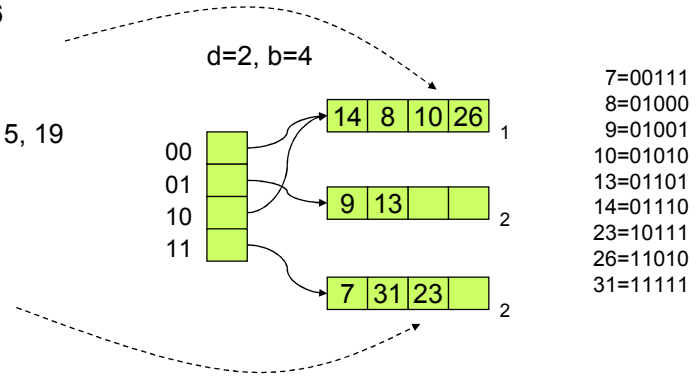
- t < d: mehrere Indexeinträge referenzieren diesen Block
- t = d: ein Indexeintrag referenziert diesen Block

Fall 1: t < d

- Einfügen von 6
  - 6=00110

Fall 2: t = d

- Einfügen von 15, 19
  - 15=01111
  - 19=10011

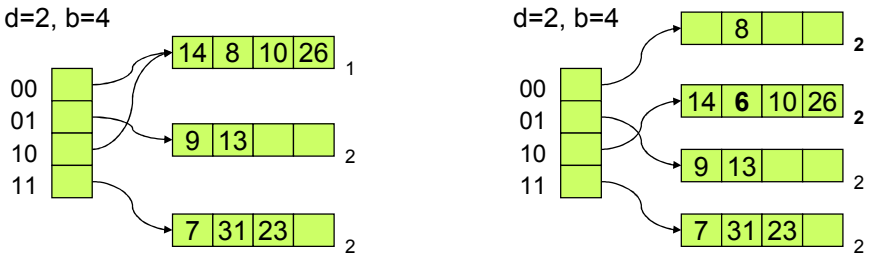


Einfügen Fall 1: t < d

Versuch den Split ohne Indexexpansion durchzuführen

- Neuen Datenblock anfordern
- Aufteilung der Daten des überlaufenden Blocks nach h_{t+1}
- t = t + 1 für alten und neuen Datenblock
- Falls Split wieder zu einem Überlauf führt, wiederholen

Einfügen von 6 (=00110)



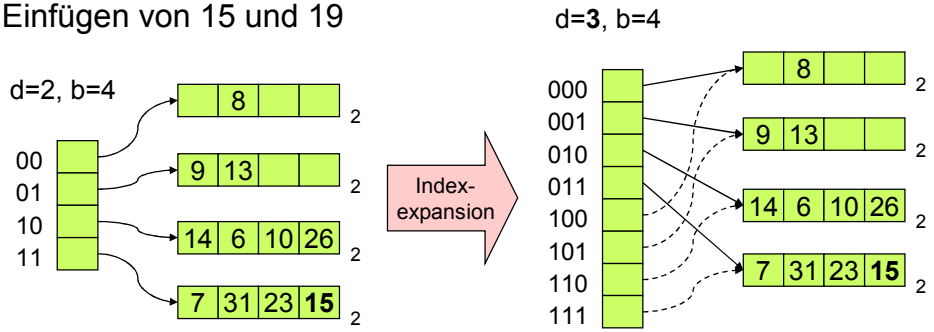
Einfügen Fall 2: t = d

Erfordert eine Indexexpansion (Verdoppelung)

- Neuen Speicherbereich für 2^d zusätzliche Referenzen anfordern
- Für jedes T[x], für das gilt $x \geq 2^d$: $T[x] = T[x-2^d]$
- d = d+1

Danach Rückführung auf Fall 1

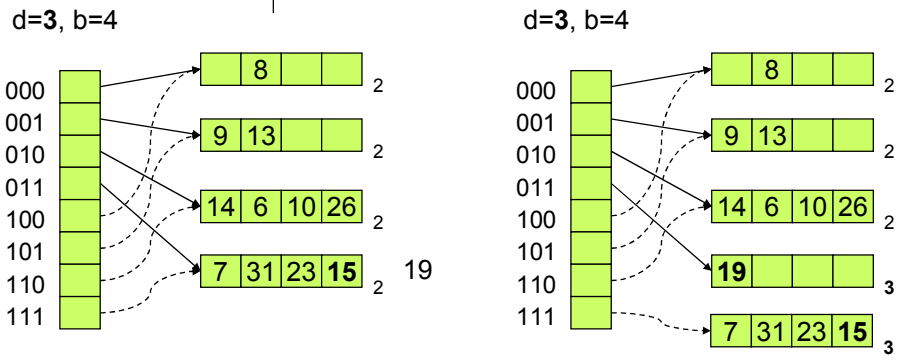
Einfügen von 15 und 19



Einfügen Fall 2: t = d (2)

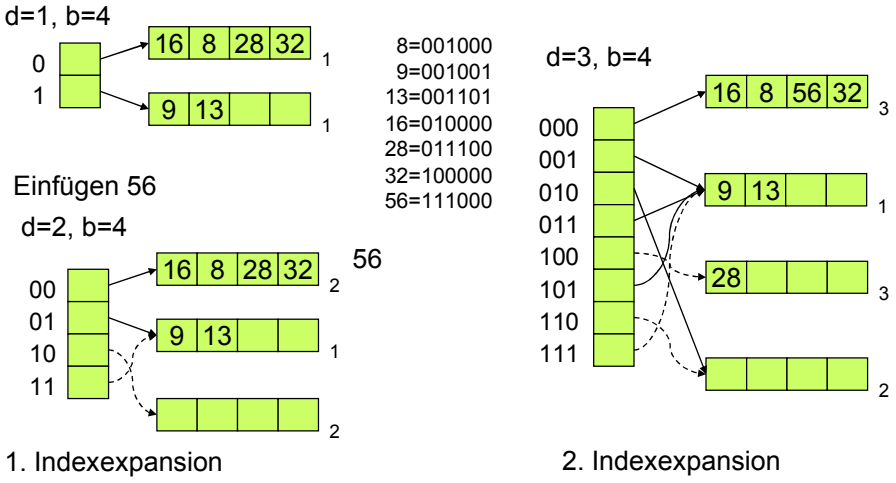
Jetzt Fall 1

7=00111
15=01111
19=10011
23=10111
31=11111



Problem des Indexwachstums

Indexexpansion kann scheitern, mehrfache Verdoppelungen



Kommentare

- Löschen leerer Blöcke nicht einfach
  - Verschmelzung mit ihren Split-Buddies („Split-Images“)
  - Analog für Indexverkleinerung
- Falls Index im Hauptspeicher gehalten werden kann, nur ein externer Zugriff notwendig
- Im worst case exponentielles Indexwachstum
  - Typisch bei clustered data (Daten sind nicht gleichverteilt)
  - Daher Speicherplatzbedarf für Index im worst case nicht akzeptierbar
    - Index normalerweise im Hauptspeicher
- Problem der Blockung des Index auf der Platte
- Keine garantierte Mindestauslastung

Analyse

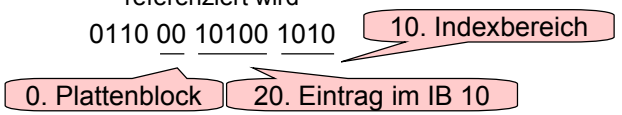
- Analyse komplex
  - Es lässt sich zeigen, dass Extendible Hashing zur Speicherung einer Datei mit n Datensätzen und einer Blockgröße von b Datensätzen ungefähr 1.44*(n/b) Blöcke benötigt
  - Das Verzeichnis hat durchschnittlich für gleichverteilte Datensätze $n^{1+1/b}/b$ Einträge
- Großer Vorteil falls das Verzeichnis in den Hauptspeicher passt, nur ein Plattenzugriff auf einen Datensatz notwendig

3.3.3 Bounded Index Size Extendible Hashing

- Ansatz durch
  - Primärbereich (analog zu Extendible Hashing)
  - Index mit beschränkter Größe
  - Binäre Expansion der Datenbereiche
    - (1 Block, 2, 4, ... Blöcke)
- Versucht dem Unvermögen des Extendible Hashings, den Index auf der Platte unterzubringen, mit einer speziellen Indexstruktur zu begegnen

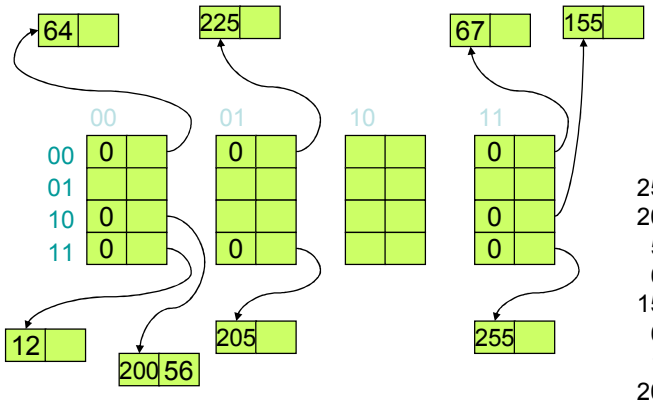
Struktur

- Index besteht aus x Bereichen
- Ein Bereich enthält y Einträge der Form <z, ptr> wobei
  - ptr ist eine Adresse auf einen Plattenblock
  - z gibt die Anzahl der Verdoppelungen der entspr. Datenbereiche
  - d.h. ptr ist die Adresse eines Datenbereichs mit 2^z Plattenblöcken
  - x und y sind Potenzen von 2
- Hash-Werte werden gezont interpretiert, z.B.
  - 16 Indexbereiche (x=16), 32 Einträge pro Indexbereich (y=32), 4 Blöcke in jedem Datenbereich
    - Hash-Signatur für den 0. Block der vom 20. Eintrag im Indexbereich 10 referenziert wird



Beispiel BISEH

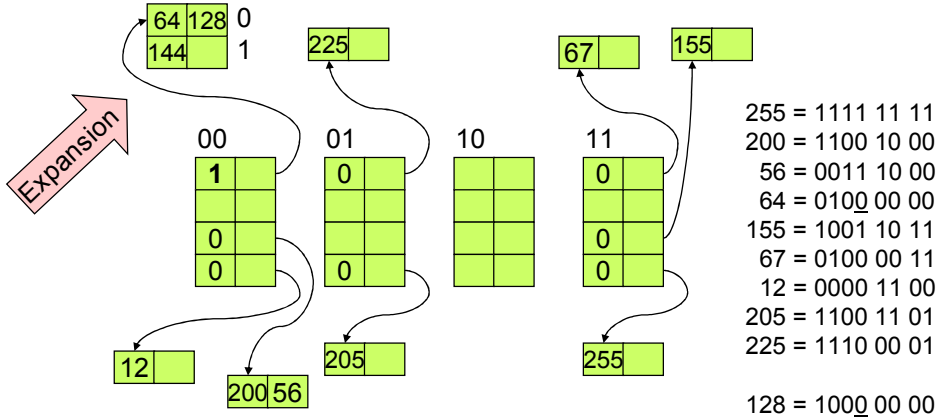
b=2 (Blockgröße), x=4 (Indexbereiche), y=4 (Einträge pro IB)



255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01

Beispiel BISEH (2)

Einfügen von 128 und 144, im ersten Versuch erfolgreich

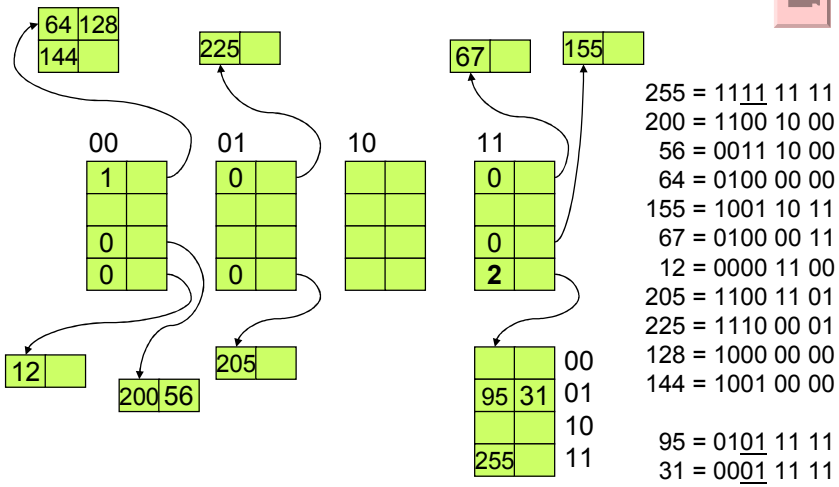


255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01

128 = 1000 00 00
144 = 1001 00 00

Beispiel BISEH (2)

Einfügen von 95 und 31, erst im zweiten Versuch erfolgreich



255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01
128 = 1000 00 00
144 = 1001 00 00

95 = 0101 11 11
31 = 0001 11 11

Allgemeiner Vergleich von Hash- zu Index-Verfahren

Punkte, die zu beachten sind:

- Kosten der periodischen Reorganisation
- Häufigkeit von Einfügen und Löschen
- Average Case versus Worst Case Aufwand
- Erwartete Abfrage Typen
  - Hashing ist generell besser bei exakten Schlüsselabfragen
  - Indexstrukturen sind bei Bereichsabfragen vorzuziehen

3.4 Sortierverfahren

Das *Sortieren* von Elementen (Werten, Datensätzen, etc.) stellt in der Praxis der EDV eines der wichtigsten und aufwendigsten Probleme dar.

- Es existieren hunderte Sortieralgorithmen in den verschiedensten Varianten für unterschiedlichste Anwendungsfälle.
  - Hauptspeicher - Externspeicher
  - Sequentiell - Parallel
  - Insitu - Exsitu
  - Stable - Unstable
  - ...
- Sortierverfahren gehören zu den am umfangreichsten analysierten und verfeinerten Algorithmen in der Informatik.

Kriterien für die Auswahl von Sortierverfahren

Hauptspeicher - Externspeicher

- Kann der Algorithmus nur Daten im Hauptspeicher oder auch Files (Dateien) oder Bänder auf dem Externspeicher (Platte, Bandstation) sortieren?

Insitu - Exsitu

- Kommt das Sortierverfahren mit ursprünglichen Datenbereich aus (insitu) oder braucht es zusätzlichen Speicherplatz (exsitu)?

Worst Case - Average Case

- Ist das (asymptotische) Laufzeitverhalten des Sortierverfahrens immer gleich oder kann es bei speziellen Fällen “entarten” (schneller oder langsamer werden)?

Stable - Unstable

- Wenn mehrere Datensätze denselben Schlüsselwert haben, bleibt dann die ursprüngliche Reihenfolge nach dem Sortieren erhalten?

Sortierverfahren

Aufgabe

- Ein Vektor der Größe n der Form V[0], V[1], …, V[n-1] ist zu sortieren.
  - Lexikographische Ordnung auf den Elementen.
- Nach dem Sortieren soll gelten: $V[0] \leq V[1] \leq \ldots \leq V[n-1]$.

Elementare (Simple) Verfahren (Auswahl)

- Selection Sort
- Bubble Sort

Verfeinerte („Intelligentere“) Verfahren (Auswahl)

- Quicksort
- Mergesort
- Heapsort

Klasse Vector

```
class Vector {
  int *a, size;
public:
  Vector(int max) { a = new int[max]; size = max; }
  ~Vector() { delete[] a; }
  void Selectionsort();
  void Bubblesort();
  void Quicksort();
  void Mergesort();
  int Length();
private:
  void quicksort(int, int);
  void mergesort(int, int);
  void swap(int&, int&);
};
```

3.4.1 Selection Sort

Selection Sort oder Minimumsuche Algorithmus

Finde das kleinste Element im Vektor und vertausche es mit dem Element an der ersten Stelle. Wiederhole diesen Vorgang für alle noch nicht sortierten Elemente.

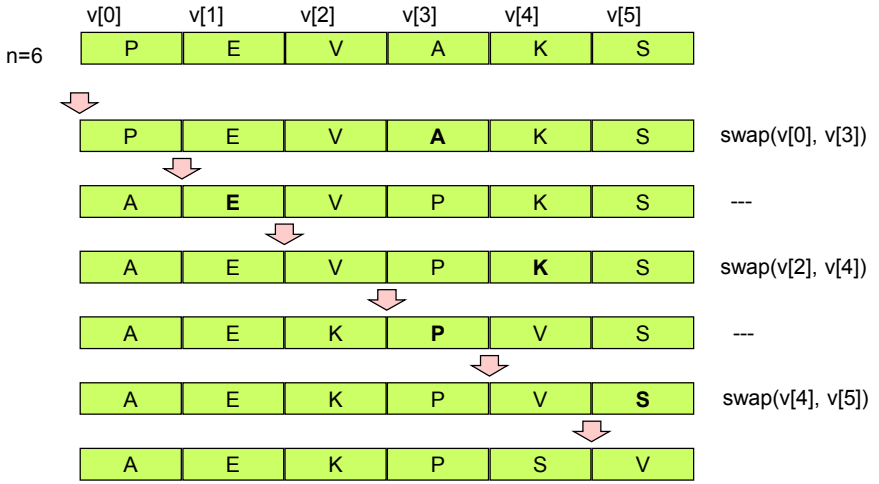
Swap

Das Vertauschen (*Swap*) zweier Elemente (**int**) stellt einen zentralen Vorgang beim Sortieren dar.

```
void swap(int &x, int &y) {
  int help = x;
  x = y;
  y = help;
}
```


Selection Sort, Beispiel

Beispiel



Selection Sort, Algorithmus

Algorithmus

```
void Vector::Selectionsort() {
  int n = Length();
  int i, j, min;
  for(i = 0; i < n; i++) {
    min = i;
    for(j = i+1; j < n; j++)
      if(a[j] < a[min]) min = j;
    swap(a[min], a[i]);
  }
}
```


Selection Sort, Eigenschaften

Eigenschaften

- Hauptspeicheralgorithmus
- Insitu Algorithmus
  - sortiert im eigenen Datenbereich, braucht nur eine temporäre Variable, konstanter Speicherplatzverbrauch
- Aufwand
  - generell $O(n^2)$

3.4.2 Bubble Sort

Bubble Sort Algorithmus

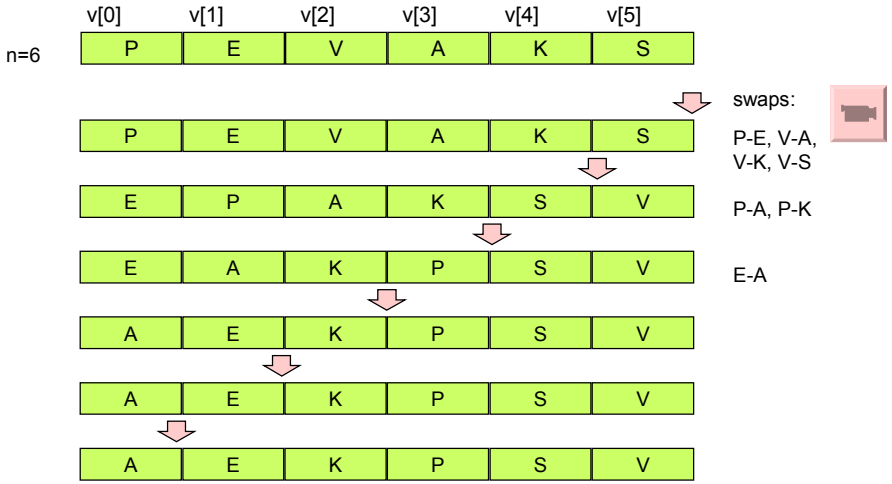
Wiederholtes Durchlaufen des Vektors.

Wenn 2 benachbarte Elemente aus der Ordnung sind (größeres vor kleinerem), vertausche (swap) sie.

Dadurch wird beim ersten Durchlauf das größte Element an die letzte Stelle gesetzt, beim 2. Durchlauf das 2.größte an die vorletzte Stelle, usw. Die Elemente steigen wie Blasen (bubbles) auf.

Bubble Sort, Beispiel

Beispiel



Bubble Sort, Algorithmus

Algorithmus

```
void Vector::Bubblesort() {
  int n = Length();
  int i, j;
  for(i = n-1; i >= 1; i--)
    for(j = 1; j <= i; j++)
      if(a[j-1] > a[j])
        swap(a[j-1], a[j]);
}
```


Bubble Sort, Eigenschaften

Eigenschaften

Hauptspeicheralgorithmus

Insitu Algorithmus

sortiert im eigenen Datenbereich, braucht nur eine temporäre Variable, konstanter Speicherplatzverbrauch

Aufwand

generell $O(n^2)$

3.4.3 Quicksort

Quicksort (Hoare 1962) Algorithmus

Der Vektor wird im Bezug auf ein frei wählbares Pivotelement durch Umordnen der Vektorelemente in 2 Teile geteilt, sodaß alle Elemente links vom Pivotelement kleiner und alle Elemente rechts größer sind.

Die beiden Teilvektoren werden unabhängig voneinander wieder mit quicksort (rekursiv) sortiert.

divide-and-conquer Paradigma

Das Pivotelement steht dadurch auf seinem endgültigen Platz. Es bleiben daher nur mehr n-1 Element (in den beiden Teile) zu sortieren (Verringerung der Problemgröße)

Beruht auf dem “divide-and-conquer” Ansatz

Pivotelement umordnen

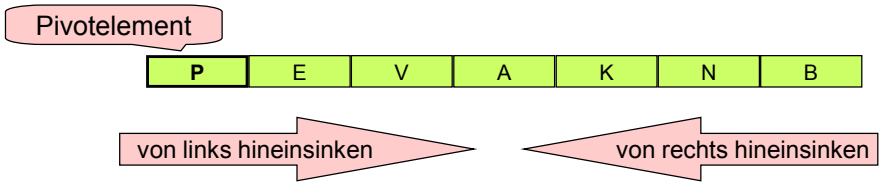
Beliebiges Element als Pivotelement wählen

(im u.a. Bsp. linkes Element im Vektor, auch zufällige Wahl oder Median aus 3 zufälligen üblich)

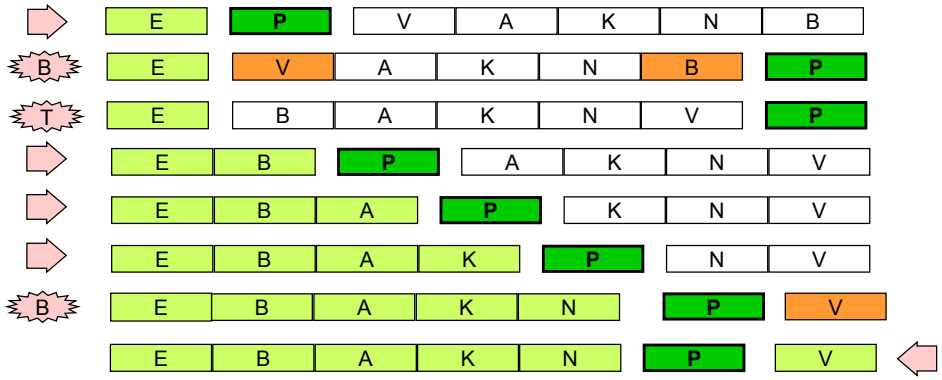
Pivotelement von links bzw. rechts in den Vektor ‘hineinsinken’ lassen,

d.h. jeweils sequentiell von links mit allen kleineren Elementen, von rechts mit allen größeren vertauschen.

Bei Blockierung, d.h. links kleineres und rechts größeres Element, diese beiden Elemente vertauschen



Pivotelement umordnen (2)

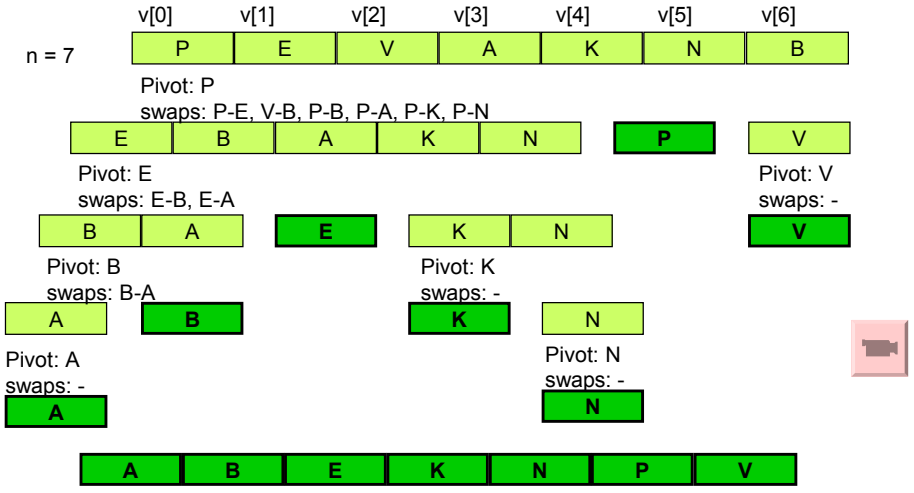


von links hineinsinken E umgeordnete Elemente

von rechts hineinsinken B blockierende Elemente

B,T Blockierung u. Tausch P Pivotelement

Quicksort, Rekursion



Quicksort, Algorithmus

```
void Vector::Quicksort() {
  quicksort(0, Length()-1);
}
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
  }
}
```


Quicksort, Eigenschaften

Hauptspeicheralgorithmus

Insitu Algorithmus

- sortiert im eigenen Datenbereich, braucht nur temporäre Hilfsvariable (Stack), konstanter Speicherplatzverbrauch

Aufwand

- Im Durchschnitt O(n*log(n))
- Kann zu $O(n^2)$ entarten
  - Beispiel: Vektor ist vorsortiert und als Pivotelement wird immer das erste oder letzte Vektorelement gewählt

Empfindlich auf unvorsichtige Wahl des Pivotelements

Stabilität nicht einfach realisierbar

Rekursiver Algorithmus

- Komplex zu realisieren, wenn Rekursion nicht zur Verfügung steht

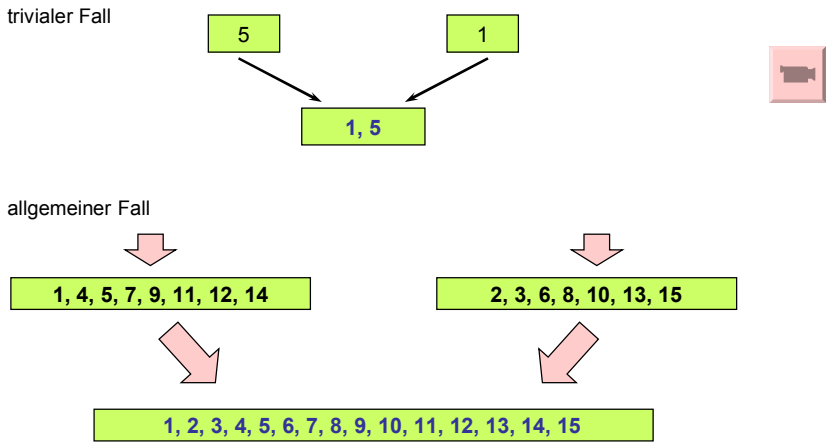
3.4.4 Mergesort

Mergesort (???, 1938) Algorithmus

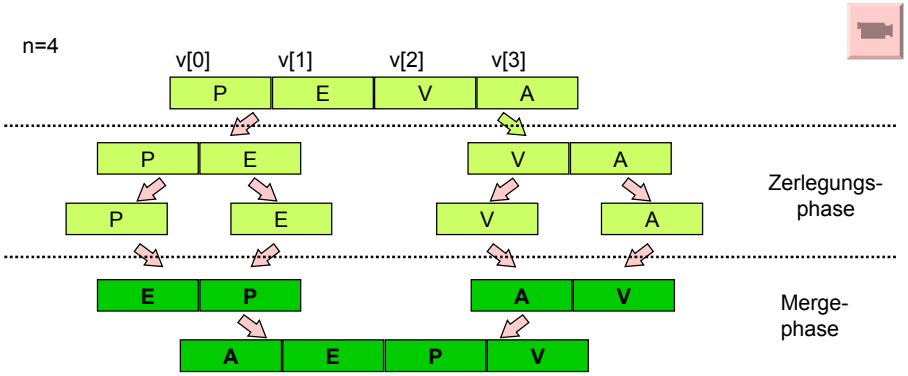
- Der zu sortierende Vektor wird rekursiv in Teilvektoren halber Länge geteilt, bis die (trivialen, skalaren) Vektoren aus einem einzigen Element bestehen
- Danach werden jeweils 2 Teilvektoren zu einem doppelt so großen Vektor gemerged (sortiert)
  - Beim Mergen werden sukkzessive die ersten beiden Element der Vektoren verglichen und das kleinere in den neuen Vektor übertragen, bis alle Elemente im neuen Vektor sortiert gespeichert sind
  - Das Mergen wird solange wiederholt, bis in der letzten Phase 2 Vektoren der Länge n/2 zu einem sortierten Vektor der Länge n verschmelzen
- “divide-and-conquer” Ansatz

Merging

Merging zweier Vektoren



Mergesort, Rekursion



Mergesort, Algorithmus

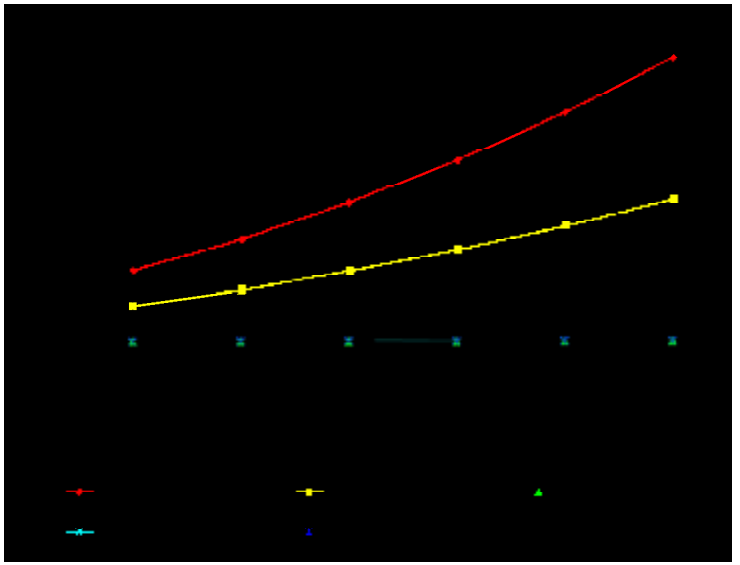
```
void Vector::Mergesort() {
  mergesort(0, Length()-1);
}
void Vector::mergesort(int l, int r) {
  int i, j, k, m;
  Vector help(r-l+1);
  if(r > l) {
    m = (r+l)/2;
    mergesort(l, m);
    mergesort(m+1, r);
    for(i=m+1; i>l; i--) help.a[i-1] = a[i-1];
    for(j=m; j<r; j++) help.a[r+m-j] = a[j+1];
    for(k=l; k<=r; k++)
      a[k]=(help.a[i]<help.a[j])?help.a[i++]:help.a[j--];
  }
}
```


Mergesort, Eigenschaften

Eigenschaften

- Hauptspeicheralgorithmus, aber auch als Externspeicheralgorithmus verwendbar.
  - Extern Sortieren: statt Teilphase wird oft vorgegebene Datenaufteilung genommen, oder nur soweit geteilt, bis sich Teilvektor (Datei) im Hauptspeicher sortieren lässt.
- Exsitu Algorithmus
  - braucht einen zweiten ebenso großen Hilfsvektor zum Umspeichern
- Aufwand
  - Generell O(n*log(n))
  - Braucht doppelten Speicherplatz
- Rekursiver Algorithmus
  - Komplex zu realisieren, wenn Rekursion nicht zur Verfügung steht.

Laufzeitvergleich der Verfahren



3.4.5 Comparison Sort Verfahren

Man nennt diese bekannten Verfahren *Vergleichende Sortierverfahren* (Comparison sort)

- Die Reihenfolge der Elemente wird dadurch bestimmt, dass die Elemente miteinander verglichen werden
  - Nur Vergleiche der Form $x_i < x_j$, $x_i \leq x_j$, ... angewendet
- Diese Algorithmen zeigen als günstigste Laufzeit eine Ordnung von O(n log n), d.h. wir konnten bis jetzt als untere Grenze Ω(n log n) feststellen

Vermutung: Das Problem des Sortierens durch Vergleichen der Elemente lässt sich mit Ω(n log n) beschreiben

Falls gültig $\Rightarrow$ Algorithmische Lücke geschlossen, da $\Omega(P) = O(A)$

Entscheidungsbaum

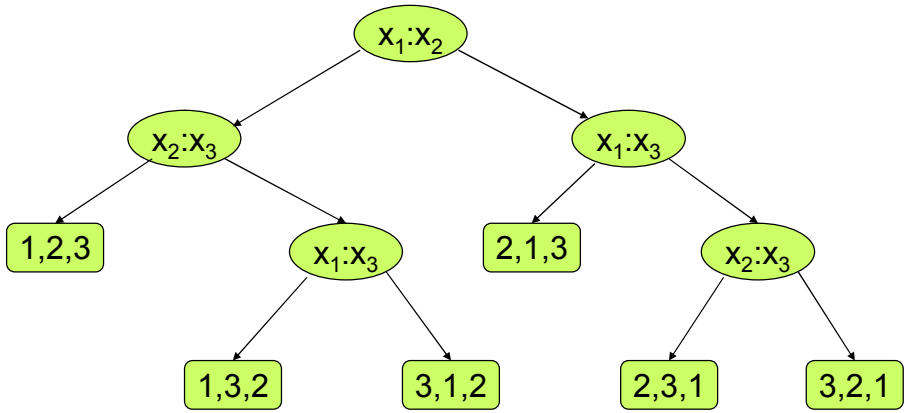
Betrachtung der Vergleichenden Sortierverfahren durch Entscheidungsbäume

- Der Entscheidungsbaum enthält alle möglichen Vergleiche eines Sortierverfahrens um eine beliebige Sequenz einer vorgegebenen Länge zu sortieren
  - Die Blattknoten repräsentieren alle möglichen Permutationen der Eingabe Sequenz
- Die Wege von der Wurzel zu den Blättern definieren die notwendigen Vergleiche um die entsprechenden Permutationen zu erreichen

Beispiel Entscheidungsbaum

Entscheidungsbaum für 3 Elemente

d.h. 3! = 6 mögliche Permutationen der Eingabe Elemente
$< x_1, x_2, x_3 >$



Beweis: Untere Grenze für den worst case

Wir ignorieren Swappen, Administrationsoperationen, etc.

Die Anzahl der Blätter im Entscheidungsbaum ist n!

Die Anzahl der Blätter in einem binären Baum ist $\leq 2^h$

daher

$$2^h \geq n!$$

$$h \geq \lg(n!)$$

Stirlingsche Näherung $\quad n! = \sqrt{2pn}\left(\frac{n}{e}\right)^n (1 + q(1/n)) > \left(\frac{n}{e}\right)^n$

$$h > \lg\left(\frac{n}{e}\right)^n = n \lg n - n \lg e$$

$$h = \Omega(n \lg n)$$

d.h. Algorithmische Lücke für Vergleichende Sortierverfahren geschlossen

3.5 Sortieren in linearer Zeit

Sortierverfahren, die nicht auf dem Vergleich zweier Werte beruhen, können eine Eingabe Sequenz in linearer Zeit sortieren

- Erreichen die Ordnung O(n)

Beispiele

- Counting Sort
- Radix Sort
- Bucket Sort

3.5.1 Counting Sort

1954 Erstmals verwendet von Harold H.Seward MIT Thesis „Information Sorting in the Application of Electronic Digital Computers to Bussiness Operations“ als Grundlage für Radix Sort.

1956 Erstmals publiziert von E.H. Fried

1960 unter dem Namen Mathsort von W. Feurzeig.

Counting Sort

Bedingung

- Die zu sortierenden n Elemente sind Integer Werte im Bereich 0 bis k
- Falls k von der Ordnung n ist (k = O(n)) benötigt das Sortierverfahren O(n)

Ansatz

- Gegeben sei eine exakte Permutation

Jedes Element wird einfach auf die Position seines Wertes in einem Zielvektor gestellt

5	8	4	2	7	1	6	3

Counting Sort (2)

Erweiterung

- Die Elemente stammen aus dem Bereich [1..k] und es gibt keine doppelten Werte
- Frage: wie können wir den ersten Ansatz erweitern?

Allgemeiner Fall

- Die Elemente sind aus dem Bereich [1..k], Doppelte sind erlaubt

Idee des Counting Sort

- Bestimme für jedes Element x die Anzahl der Elemente kleiner als x im Vektor, verwende diese Information um das Element x auf seine Position im Ergebnisvektor zu platzieren

Counting Sort Algorithmus

Eingangsvektor **A**, Ergebnisvektor **B**, Hilfsvektor **C**

```
for(i=1; i<=k; i++)
  C[i] = 0;
for(j=1; j<=length(A); j++)
  C[A[j]] = C[A[j]] + 1;
for(i=2; i<=k; i++)
  C[i] = C[i] + C[i-1];
for (j=length(A); j>=1; j--) {
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]] – 1;
}
```

1: C[i] enthält die Anzahl der Elemente gleich i, i = 1,2,...,k

2: C[i] enthält die Anzahl der Elemente kleiner oder gleich i

3: Jedes Element wird an seine korrekte Position platziert

Falls alle Element A[j] unterschiedlich sind ist die korrekte Position in C[A[j]], da Elemente gleich sein können wird der Wert C[A[j]] um 1 verringert

Counting Sort Beispiel

1:

A	3	6	4	1	3	4	1	4

C	2	0	2	3	0	1

2:

C	2	2	4	7	7	8

3: 1. Durchlauf

B							4	

C	2	2	4	**6**	7	8

3: 2. D.

B		1					4	

C	**1**	2	4	6	7	8

3: 3. D.

B		1				4	4	

C	1	2	4	**5**	7	8

3: Ende

B	1	1	3	3	4	4	4	6

Counting Sort Analyse

```
for(i=1; i<=k; i++)
  C[i] = 0;                          O(k)
for(j=1; j<=length(A); j++)
  C[A[j]] = C[A[j]] + 1;             O(n)
for(i=2; i<=k; i++)
  C[i] = C[i] + C[i-1];              O(k)
for (j=length(A); j>=1; j--) {
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]] – 1;             O(n)
}
```

Daraus folgt, dass der Gesamtaufwand O(n + k) ist

In der Praxis haben wir sehr oft k = O(n), wodurch wir einen realen Aufwand von O(n) erreichen

Man beachte, dass kein Vergleich zwischen Werten stattgefunden hat, sondern die Werte der Elemente zur Platzierung verwendet wurden (vergl. Hashing)

Counting Sort ist ein stabiles Sortierverfahren

Garantiert durch letzte Schleife, die von hinten nach vorne arbeitet

3.5.2 Radix Sort

Radixsort betrachtet die Struktur der Schlüsselwerte

Die Schlüsselwerte der Länge b werden in einem Zahlensystem der Basis M dargestellt (M ist der Radix)

z.B. M=2, die Schlüssel werden binär dargestellt, b = 4

	8	4	2	1	
9 =	1	0	0	1	b = 4
	3	2	1	0	Stelle #

Gewicht

Es werden Schlüssel sortiert, indem ihre einzelnen Bits an derselben Bit Position miteinander verglichen werden

Das Prinzip lässt sich auch auf alphanumerische Strings erweitern

Radix Sort Algorithmus

Allgemeiner Radix Sort Algorithmus

```
for(Index i läuft über alle b Stellen) {
  sortiere Schlüsselwerte mit einem (stabilen)
    Sortierverfahren bezüglich Stelle i
}
```

Richtung des Indexlaufs, Stabilität

- Von links nach rechts
  - d.h. `for(int i=b-1; i>=0; i--) { ... }`
- Führt zum *Binären Quicksort (Radix Exchange Sort)*
- Von rechts nach links und stabiles Sortierverfahren
  - d.h. `for(int i=0; i<b; i++) { ... }`
- Führt zum *LSD-Radixsort (Straight Radix Sort)*

Binärer Quicksort

Sehr altes Verfahren

1929 erstmals für maschinelles Sortieren von Lochkarten.

1954 H.H. Seward [MIT R-232 (1954), p25-28]

eigenständig und ausführlich: P.Hildebrandt , H. Isbitz, H. Rising, J. Schwartz
[JACM 6 (1959), 156-163]

1 Jahr vor Quicksort

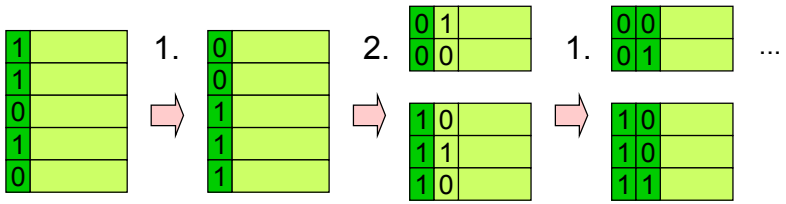
Binärer Quicksort

Binärer Quicksort Algorithmus

Betrachte Stellen von links nach rechts

1. Sortiere Datensätze bezogen auf das betrachtete Bit
2. Teile die Datensätze in M unabhängige, der Größe nach geordnete, Gruppen und sortiere rekursiv die M Gruppen wobei die schon betrachteten Bits ignoriert werden

Beispiel

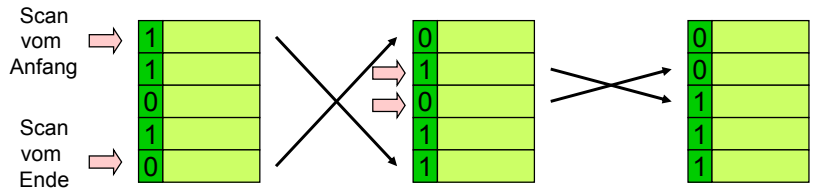


Binärer Quicksort Partition

Aufteilung der Datensätze durch insitu Ansatz ähnlich des Partitionierens beim Quicksort

```
do {
  scan top-down to find key starting with 1;
  scan bottom-up to find key starting with 0;
  exchange keys;
} while (not all indices visited)
```

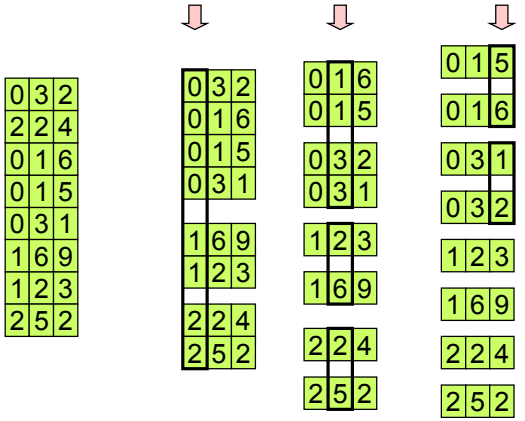
Beispiel



Binärer Quicksort für Dezimalzahlen

Anzahl der Gruppen M ist 10

Zahlenwerte 0 bis 9



Binärer Quicksort Eigenschaften

Binärer Quicksort verwendet im Durchschnitt ungefähr N log N Bitvergleiche

- log N (Kodierung des Zahlenwerts) = konstanter Faktor
- N = Unsicherheit in der Zahlendarstellung

Gut geeignet für kleine Zahlen

Benötigt relativ weniger Speicher als andere Lineare Sortierverfahren

Es ist ein Instabiles Sortierverfahren

Ähnlich dem Quicksort nur für positive ganze Zahlen

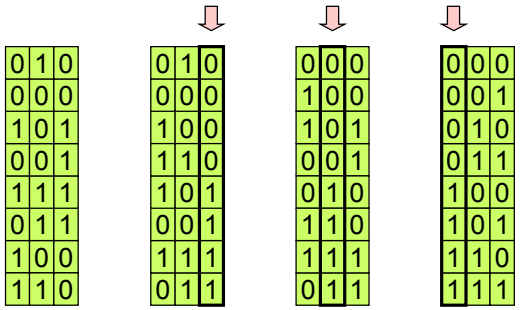
LSD-Radixsort

LSD-Radixsort Algorithmus

- Betrachte Bits von rechts nach links
  - LSD … „least significant digit“
- Sortiere Datensätze stabil (!) bezogen auf das betrachtete Bit

Bei Binärem Quicksort von links nach rechts!

Zu sortierende Bits pro Durchlauf



Was bedeutet „stabil“

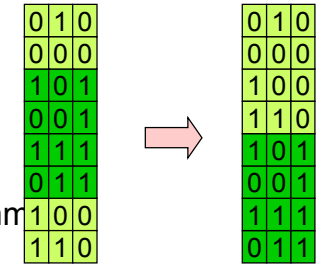
Bei einem stabilen Sortierverfahren wird die relative Reihenfolge von gleichen Schlüsseln nicht verändert

Beispiel

- Erster Durchgang von letztem Beispiel
- Die relative Reihenfolge der Schlüssel die mit 0 enden bleibt unverändert, dasselbe gilt für die Schlüssel, die mit 1 enden

Stabilität garantiert Korrektheit des Algorithm

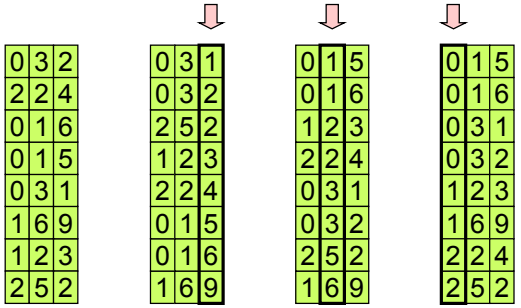
- Mögliches Verfahren: Counting Sort



LSD-Radixsort für Dezimalzahlen

Anzahl der Gruppen M ist 10

Ziffern 0 bis 9



Analyse von Radix Sort

Sortieren von n Zahlen, Schlüssellänge b

```
for(Index i läuft über alle b Stellen) {
  sortiere n Schlüsselwerte mit einem (stabilen)
    Sortierverfahren bezüglich Stelle i
}
```

Bei der Anwendung eines Sortierverfahrens mit der Laufzeit O(n) (wie z.B. Partitionierung bei Radix Exchange Sort, Counting Sort bei Straight Radix Sort) ist daher die Ordnung von Radix Sort O(bn)

- Counting Sort ist kein insitu Verfahren, daher doppelter Speicherplatz notwendig, führt in der Praxis zum Einsatz von O(n log n) Verfahren

Wenn man b als Konstante betrachtet kommt man auf O(n)

- Bei Zahlenbereich von m Werten Darstellung am Computer mit b = log(m) Stellen, da aber Wertebereich am Computer begrenzt, Ansatz b konstant akzeptierbar

3.5.3 Bucket Sort

Annahme über die n Eingabe Elemente, dass sie gleichmäßig über das Intervall [0,1) verteilt sind

Bucket Sort Algorithmus

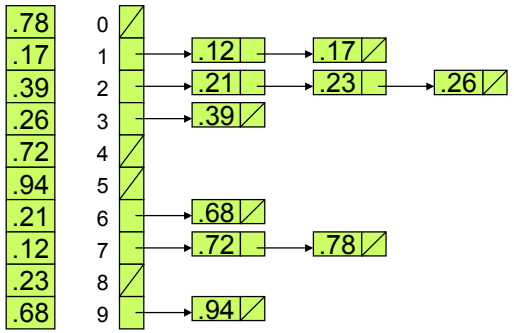
- Teile das Intervall [0,1) in n gleichgroße Teil-Intervalle (Buckets) und verteile die n Eingabe Elemente auf die Buckets
  - Da die Elemente gleichmäßig verteilt sind, fallen nur wenige Elemente in das gleiche Bucket
- Sortiere die Elemente pro Bucket mit Selection Sort
- Besuche die Buckets entlang des Intervalls und gib die sortierten Elemente aus

Bucket Sort Beispiel

Eingabe Vektor A der Größe 10

Bucket Vektor B verwaltet Elemente über lineare Listen

Bucket B[i] enhält die Werte des Intervalls [i/10, (i+1)/10)



Bucket Sort Algorithmus

```
n = length(A);
for(i=1; i<=n; i++)
  insert A[i] into list B[ën*A[i]û];
for(i=0; i<n; i++)
  sort list B[i] with insertion sort;
Concatenate the lists B[0], B[1], ..., B[n-1]
  together in order
```

Analyse Bucket Sort

Alle Anweisungen außer Sortieren sind von der Ordnung O(n)

Analyse des Sortierens

- n_i bezeichnet die Zufallszahl der Elemente pro Bucket
- Selection Sort läuft in $O(n^2)$ Zeit, d.h. die erwartete Zeit zum Sortieren eines Buckets ist $E[O(n^2)] = O(E[n^2])$, da alle Buckets zu sortieren ist

$$\sum_{i=0}^{n-1} O(E[n_i^2]) = O(\sum_{i=0}^{n-1} E[n_i^2]) \Rightarrow O(1)$$

- Um den Ausdruck zu berechnen Verteilung der Zufallsvariable n_i bestimmen
- Es gibt n Elemente mit n Buckets, d.h. die Wahrscheinlichkeit eines Elements in eine Bucket zu fallen ist p = 1/n
- n_i folgt der Binomial-Verteilung
  - $E[n_i] = n*p = 1$ und $Var[n_i] = n*p*(1-p) = 1 - 1/n$
- Da $E[n_i^2] = Var[n_i] + E^2[n_i] = 1 - 1/n + 1^2 = 2 - 1/n = \Theta(1)$

Daraus folgt, dass der Aufwand für Bucket Sort O(n) ist

3.6 Externes Sortieren

Externes Sortieren bezeichnet alle Sortierverfahren, die nicht ausschließlich im Hauptspeicher ablaufen

- Diese Verfahren benötigen sekundäre (Platten) oder tertiäre (Bänder) Speichermedien
- Üblicherweise versteht man darunter Verfahren, die Bändern einsetzen

Grund für externe Verfahren sind zu sortierende Datenmengen, die nicht mehr vollständig in den Hauptspeicher passen

- In der Praxis häufig anzutreffen

Ziel ist die Anzahl der Transfers zwischen Hauptspeicher und Externspeicher zu minimieren

Eigenschaften von Bändern

Bändern entsprechen allgemein Sequenzen

Die Datenelemente sind hintereinander angeordnet und man kann auf sie nur sequentiell lesend und schreibend zugreifen

- Verwaltung elementweise oder blockweise (ein Block umfasst mehrere gemeinsam verwaltete Elemente)
- Es ist nicht möglich auf ein beliebiges Element ohne Aufwand O(n) zuzugreifen
- Bänder können üblicherweise von beiden Seite gelesen und geschrieben werden bzw. von einem Ende zum anderen gespult werden

Strategie für externes Sortieren

Klassisches „Sortieren durch Mischen“

- Seit den 50er Jahren im Einsatz

Soviel Daten wie möglich in den Hauptspeicher laden

Diese Daten im Hauptspeicher sortieren

Sortierte Daten (Run) auf externes Speichermedium schreiben

- Ein Run ist eine geordnete Teil-Sequenz der urspr. Datenelemente

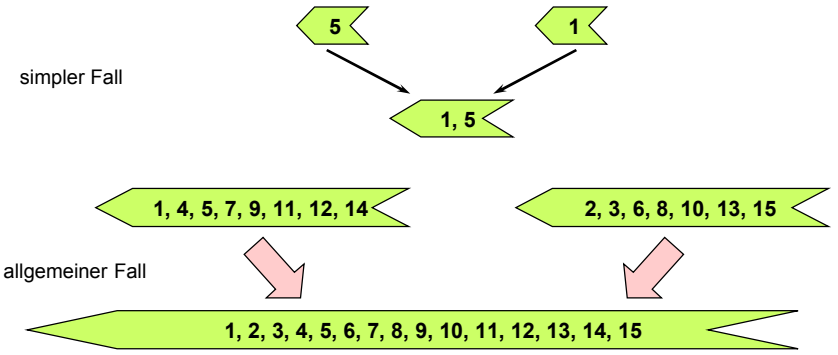
Runs über den Hauptspeicher zu größeren Runs zusammenmischen (Merging)

Merging

Ansatz analog zu Mergesort im Hauptspeicher

- Statt Hauptspeicherfeld eben Band

Merging zweier Sequenzen



3.6.1 Balanced Multiway Merging

Balanced Multiway Merging Idee

- Anfänglicher Verteilungsdurchgang
- Mehrere Mehrweg Mischdurchgänge

Annahme

- N zu sortierende Datensätze auf externem Gerät
- Platz für M Datensätze im Hauptspeicher
- 2 P externe Geräte (Bänder) zur Verfügung
  - P Inputbänder
  - P Outputbänder
- Input auf Band 0, anderen Bänder 1, 2, …, 2P – 1
- Ziel: sortiertes Ergebnis auf Band 0

Algorithmus

Bilden initaler Runs durch Sortieren der Daten im Hauptspeicher und gleichmäßiges Verteilen auf P Output Bänder

Alternieren der Bänder

- d.h. Input wird Output und vice cersa

Solange

- Merging der „Runs“ von den Input auf die Output Bänder und alternieren der Bänder

bis EIN sortierter Run entsteht

Balanced Multiway Merging Beispiel

Dreiweg Mischen, P = 3

Band 0	A S O R T I N G A N D M E R G I N G E X A M P L E *
Band 3	A O S * D M N * A E X *
Band 4	I R T * E G R * L M P *
Band 5	A G N * G I N * E *
Band 0	A A G I N O R S T *
Band 1	D E G G I M N N R *
Band 2	A E E L M P X *
Band 3	A A A D E E E G G G I I L M M N N N O P R R S T X *

Im Beispiel muss noch einmal Band 3 auf Band 0 (Ziel) kopiert werden

Balanced Multiway Merging Analyse

Durch Hardwareparameter (langsames Band) Anzahl der Banddurchläufe interessant

N ... Anzahl der Elemente

M ... Größe des Hauptspeichers

Jeder Sortier-Durchlauf erzeugt N/M sortierte Runs

Daher bei p-Weg Merging braucht man ungefähr $\log_P(N/M)$ Durchläufe

- Jeder Durchlauf verringert die Anzahl der Durchläufe um Faktor P
- Beispiel: Zu sortierende Datei 200 GB, Hauptspeicher 1 GB, Sortieren benötigt 5 Durchläufe

Organisation im Hauptspeicher

Organisation der Elemente im Hauptspeicher über Priority Queue (z.B. Heap)

Kleinstes Element (nächstes zu Mergen) ist direkt zugreifbar

Einfügen eines neuen Elementes vom Band von der Ordnung O(log M)

- M beschreibt die Größe des zur Verfügung stehenden Hauptspeichers

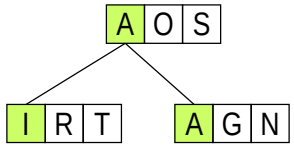
Erlaubt auch das Erzeugen von sortierten Runs die viel größer sind als der Hauptspeicher

3.6.2 Replacement Selection

Elemente im Hauptspeicher werden über eine Priority Queue der Größe p verwaltet (sortiert)

- PQ wird mit den kleinsten Elementen der p Runs gefüllt
- Das kleinste Element der PQ wird auf das Output Band geschrieben und das nächste Element nachgeschoben
- Die PQ Eigenschaft wird mit einer heapify Funktion erhalten

Beispiel



Im Hauptspeicher sind nur die kleinsten Elemente der Runs, (dargestellt werden aber die ganzen Runs (zur Erklärung)) Realisierung über Pointer auf die echten Runs = indirekter Heap

Erzeugung von initialen Runs

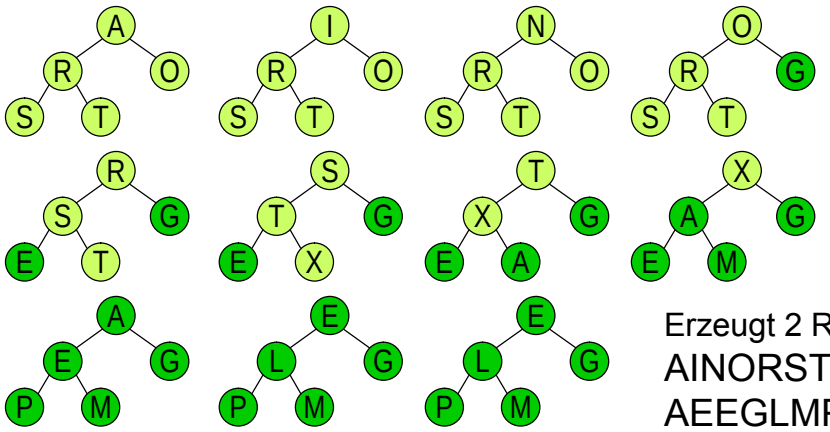
- Idee ist den ungeordneten Input durch eine große PQ durchzuschleusen
- Das kleinste Element rausschreiben und das nächste Element in die PQ aufzunehmen
  - (falls notwendig) die PQ Bedingung wiederherstellen (heapify)
- Spezielle Situation falls ein Element kleiner als das letzte geschrieben ist
  - Kann nicht mehr Teil des Runs werden, wird markiert und als größer als alle Elemente des aktuellen Runs behandelt
- Dieses Element beginnt einen neuen Run

- Es lässt sich zeigen, dass Runs, die mit Replacement Selection erzeugt wurden ungefähr 2 Mal so groß wie die PQ sind

Erzeugung von initialen Runs Beispiel

ASORTINGEXAMPLE

Heap der Größe 5



Erzeugt 2 Runs:
AINORSTX
AEEGLMP

3.6.3 Polyphase Merging

- Nachteil des Balanced Multiway Merging ist die relativ große Anzahl von benötigten Bändern
  - Ansatz mit P Bändern und 1 Output Band benötigt zwar weniger Bänder aber exzessives Kopieren
    - Output Band muss wieder auf P Bänder aufgeteilt werden
- **Idee des Polyphase Merging**
  - Verteile die mit Replacement Selection erzeugten initialen Runs ungleichmäßig über die Bänder (ein Band bleibt leer)
  - Führe ein Merging-until-empty durch
    - Merging bis ein Band leer ist, welches das neue Output Band wird

Polyphase Merging Beispiel

Band 0	A O R S T	I N A G N D E M R G I N
Band 1	E G X	A M P E L
Band 2		
Band 0	D E M R	G I N
Band 1		
Band 2	A E G O R S T X	A I M N P A E G L N
Band 0		
Band 1	A D E E G M O R R S T X	A G I I M N N P
Band 2	A E G L N	

Initiale Runs durch Replacement Selection erzeugt

Eigenschaften

Diese „Merge-until-empty“ Strategie kann auf beliebige Bandzahlen (> 2) angewendet werden

Analyse ist kompliziert

- Unterschied zwischen Balanced Multiway Sorting und Polyphase Merging eher gering
- Polyphase Merging nur für geringe Bandzahlen (p < 9) besser als BMS
  - Dient eher zum Verringern der Bandzahlen

Was nehmen wir mit?

Dictionary

Hashing

- Statische und dynamische Verfahren

Sortieren

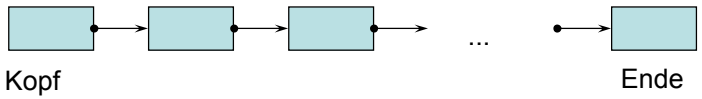
- Klassische Verfahren $O(n^2)$ und $O(n \log n)$
- Lineare Verfahren $O(n)$
- Externes Sortieren

Kapitel 4
Listen

4.1 Definition Listen

Eine (lineare) Liste ist eine Datenstruktur zur Verwaltung einer beliebig großen Anzahl von Elementen eines einheitlichen Typs.

- Der Zugriff auf die einzelnen Elemente einer (simplen) Liste ist nur vom Kopf (Head) aus möglich.
- Das Ende der Liste wird auch als Tail bezeichnet.
- Die Elemente werden in einer Sequenz angeordnet, die sich (meist) aus der Eintragereihenfolge ableiten läßt (ungeordnet).



Abstraktion

Datenstrukturen stellen eine Abstraktion eines Vorstellungsmodells dar

- Begriff des „abstrakten Datentyps“ ADT

Sagt nichts über die physische Realisierung am Computer aus

- Verschiedene Realisierungen denkbar!
- Realisierung oft abhängig von Problemstellung, Programmierumgebung, Zielsetzungen, ...

Mögliches Vorstellungsmodell Liste
„Perlenschnur“, Perlen werden an einem Ende aufgefädelt

Liste

Definition:

Eine Liste L ist eine geordnete Menge von Elementen

$$L = (x_1, x_2, \ldots, x_n)$$

Die Länge einer Liste ist gegeben durch

$$|L| = |(x_1, x_2, \ldots, x_n)| = n$$

Eine leere Liste hat die Länge 0.

Das i-te Element einer Liste L wird mit L[i] bezeichnet, daher gilt $1 \leq i \leq |L|$

Methoden auf Listen

Einfügen
- Element am Kopf einfügen

Zugriff
- Kopfelement bestimmen

Löschen
- Kopfelement entfernen

Erzeugen
- Liste neu anlegen

Längenbestimmung
- Anzahl der Elemente bestimmen

Inklusionstest
- Test, ob Element enthalten ist

andere Operationen denkbar siehe später!

Methode Einfügen

Methode ‘Add’

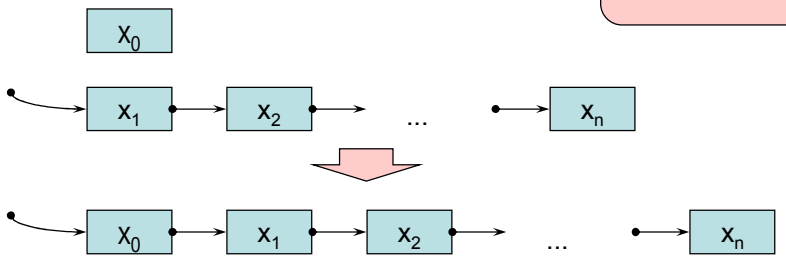
Einfügen eines Elementes a am Kopf einer Liste L, d.h.

$$L = (x_1, x_2, \ldots, x_n), x_0$$

$$\text{Add}(L, x_0)$$

$$L = (x_0, x_1, \ldots, x_n)$$

Vorstellungsmodell „Perlenschnur“: *Perle auffädeln*



Methode Zugriff

Methode ‘FirstElement’

Zugriff über das Kopfelement (x_1, das erste Listen-element) auf die Liste , d.h.

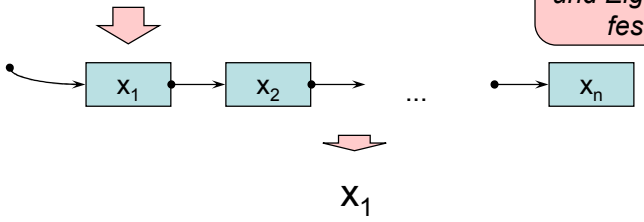
$L = (x_1, x_2, \ldots, x_n)$

$\Downarrow$

$FirstElement(L) \rightarrow x_1$

Fehler falls Liste leer

Vorstellungsmodell „Perlenschnur“: *Perle betrachten und Eigenschaften feststellen*



liefert den Wert des Elements, NICHT das Listenelement !

Methode Löschen

Methode ‘RemoveFirst’

Löscht das Kopfelement (x_1, das erste Listenelement) aus der Liste L, d.h.

$L = (x_1, x_2, \ldots, x_n)$, mit $|L| = n$

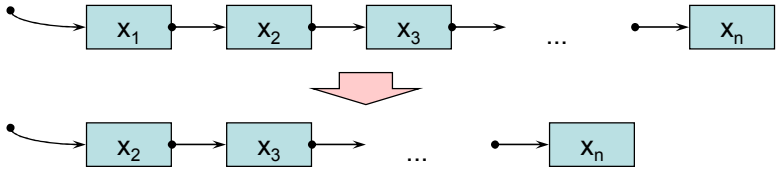
$\Downarrow$

$RemoveFirst(L)$

$\Downarrow$

$L = (x_2 \ldots, x_n)$, mit $|L| = n-1$

Vorstellungsmodell „Perlenschnur“: *Perle abziehen*



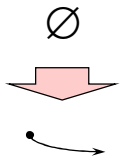
Methode Erzeugen

Methode ‘Constructor'

Erzeugt eine neue Liste, die leer ist, d.h. keine Elemente enthält und daher die Länge 0 hat,

$Constructor() \rightarrow L$, mit $|L| = 0$

Vorstellungsmodell „Perlenschnur“: *Perlenschnur vorbereiten*



Methode Längenbestimmung

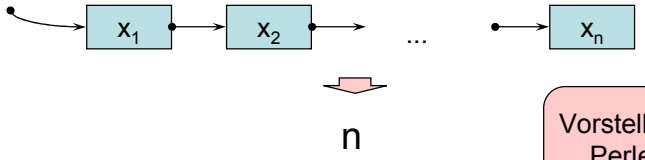
Methode ‘Length’

Bestimmt die Anzahl der Elemente der Liste L, d.h.

$L = (x_1, x_2, \ldots, x_n)$, mit $|L| = n$

$\Downarrow$

$Length(L) \rightarrow n$



liefert den ganzzahligen Wert n

Vorstellungsmodell „Perlenschnur“: *Anzahl Perlen bestimmen*

Methode Inklusionstest

Methode ‘Member’

Überprüft, ob ein gegebenes Element a in der Liste L enthalten ist, d.h.

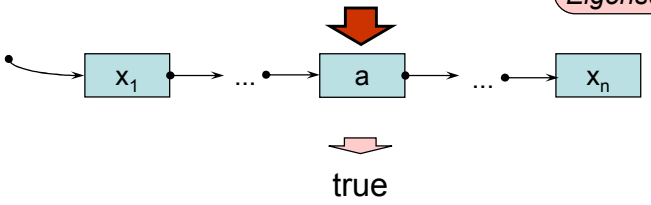
$L = (x_1, x_2, \ldots, x_n)$

Member(L, a) → [true, false]

true …$\exists\, i \mid 1 \le i \le |L| \wedge a = x_i$

false … sonst

Vorstellungsmodell „Perlenschnur“: *Perle mit spezifischer Eigenschaft suchen*



true

Klasse Listen

Deklaration C++, Klasse

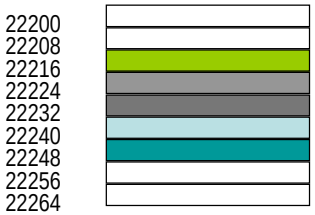
```
typedef … ItemType;
…
class List {
public:
  List(); // Constructor
  void Add(itemType a);
  ItemType FirstElement();
  void RemoveFirst();
  int Length();
  int Member(itemType a);
}
```

zur Verwendung in der Klassen Def., besserer Ansatz mit C++ Templates

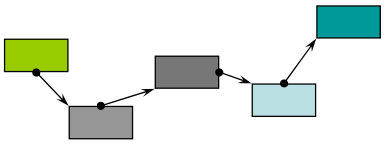
4.2 Implementierung von Listen

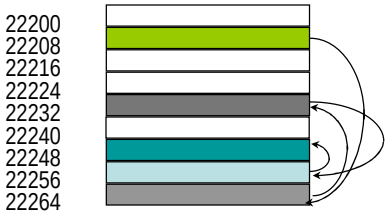
Speichertypen

Contiguous memory



Scattered (Linked) memory





Memory Typen

Contiguous memory

- Physisch zusammenhängender Speicherplatzbereich, äußerst starr, da beim Anlegen die endgültige Größe fixiert wird.
- Verwaltung über das System.
- Datenstrukturen auf der Basis von contiguous memory können nur eine begrenzte Anzahl von Elementen aufnehmen.

Scattered (Linked) memory

- Physisch verteilter Speicherbereich, sehr flexibel, da die Ausdehnung dynamisch angepaßt werden kann.
- Verwaltung über das Programm.
- Datenstrukturen können beliebig groß werden.

C++ Konstrukt: Feld

Contiguous memory

Ein Feld bzw. Array ist einer Anreihung einer fixen Anzahl von Elementen des gleichen Typs.

Der Zugriff auf ein einzelnes Feldelement erfolgt über eine Index (die relative Position im Feld). Der Index startet oBdA mit *0* und endet mit *Anzahl-1*.

Vereinbarung

<ETyp> <Feldname>'['<EZahl>']'

z.B.:

```
double x[10];
int a[10000];
char name[25];
```

Zugriff

<Feldname>'['index']'

z.B.:

```
x[0] = 3.1415 * r * r;
a[9999] = 0;
```


C++ Konstrukt: Dynamische Objekte

Scattered memory

Dynamische Objekte

Objekte die zur Laufzeit des Programmes durch Programmanweisungen erzeugt und für die Speicherplatz angelegt wird.

Diese Objekte besitzen keinen Namen und werden über sog. Zeiger (Pointer) verwaltet.

Zeiger bzw. Pointer

Eine Variable, die die Adresse eines Objekts (Adressoperator '&') enthält

z.B.:

```
double x = 3.14159;
double* p = &x;
  // & liefert Adr. von x
cout << *p;
  // * deref. Ptr., d.h.
  // druckt 3.14159
```

x	≡	22192	3.14158
p	≡	22200	22192

C++ Konstrukt: Dynamische Objekte

Erzeugung bzw. Zerstörung

new … erzeugt ein neues Objekt

delete … zerstört ein existierendes Objekt

```
double * p = new double;
* p = 3.14159;
```

	p	≡ 22200	22208
		22208	3.14159

```
delete p;
```

	p	≡ 22200	22208
			3.14159

alter, undefinierter Wert

vom System freigegeben

Achtung:

```
p = 0 // erlaubt,
      // Initialisierung
p = 4711 // verboten
  // Adressmanipulation
```

Operatoren

& ... liefert die Adresse des Objekts

* ... dereferenziert den Zeiger und liefert den Inhalt der ref. Objektes

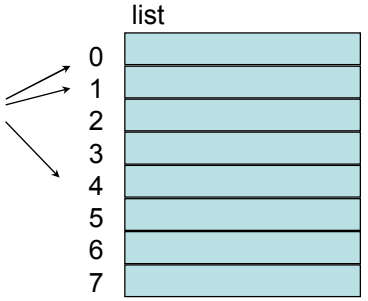
-> ... Zugriff auf eine Strukturkomponente über einen Zeiger,
d.h. x->y entspricht (*x).y

4.2.1 Liste - statisch - Struktur

Statische Implementation - contiguous memory

Speichern der Elemente in einem Feld begrenzter Länge

```
typedef int ItemType;
class List {
private:
  ItemType list[8];
    // Datenstruktur
  int p;
    // nächste freie Position
  …
}
```



Länge = 8

Liste - statisch - Erzeugen - Zerstören - Einfügen

Erzeugen,

```
List::List() {p = 0;}
```

Zerstören

```
List::~List() {p = 0;}
```

Einfügen

```
void List::Add(ItemType a) {
  if(p < 8) {
   list[p] = a;
   p++;
  }
  else cout << "Error-add\n";
}
```


Liste - statisch - Zugriff - Löschen

Zugriff

```
ItemType List::FirstElement() {
  if(p > 0) return list[p-1];
  else cout << "Error-first\n";
}
```

Löschen

```
void List::RemoveFirst() {
  if(p > 0) p--;
  else cout << "Error-remove\n";
}
```


Liste - statisch - Länge - Inklusionstest

Länge,

```
int List::Length() {
  return p;
}
```

Inklusionstest

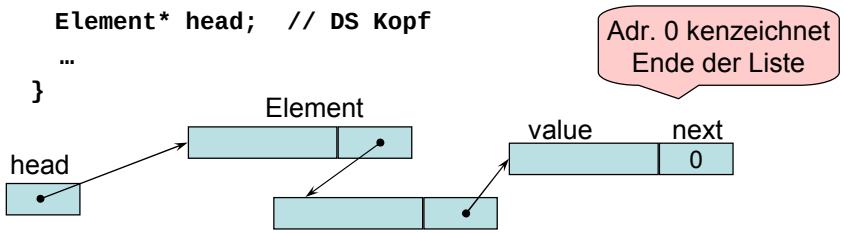
```
int List::Member(ItemType a) {
  int i = 0;
  while(i < p && list[i] != a) i++;
  if(i < p) return 1;
  else return 0;
}
```


4.2.2 Liste - dynamisch - Struktur (1)

Dynamische Implementation - linked memory

Dynamisch erweiterbare Liste unbegrenzter Länge

```
typedef int ItemType;
class List {
public:
  class Element {       // Elementklasse
       ItemType value;
       Element* next;
  };
  Element* head;  // DS Kopf
  …
}
```



Rekursive Datenstruktur

Eine Datenstruktur heißt rekursiv oder zirkulär, wenn sie sich in ihrer Definition selbst referenziert

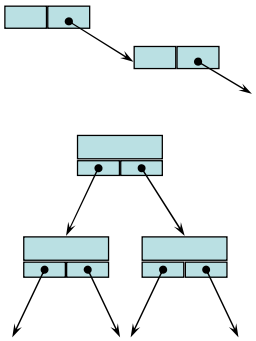
Basismodell für dynamisch erweiterbare Datenstrukturen

Liste

```
class Element{
  InfoType Info;
  Element* Next;
}
```

Baum

```
class Node {
  KeyType Key;
  Node* LeftChild;
  Node* RightChild;
}
```



Liste - dynamisch - Einfügen

Einfügen

```
void List::Add(ItemType a) {
  Element* help;
  help = new Element;
  help->next = head;
  help->value = a;
  head = help;
}
```

Typischerweise am Kopf der Liste

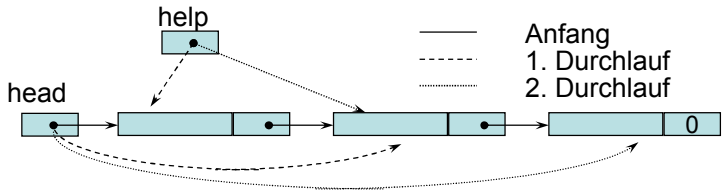
Vor dem Einfügen

Liste - dynamisch - Erzeugen - Zerstören

Erzeugen, Löschen

```
List::List() { head = 0;}

List::~List() {
  Element* help;
  while(head != 0) {
   help = head;
   head = head->next;
   delete help;
  }
}
```



Liste - dynamisch - Zugriff

Zugriff

```
ItemType List::FirstElement() {
  if(head != 0)
    return head->value;
  else
    cout << "Error-first\n";
}
```

Liste - dynamisch - Löschen

Löschen

```
void List::RemoveFirst() {
  if(head != 0) {
   Element* help;
   help = head;
   head = head->next;
   delete help;
  } else cout << "Error-remove\n";
}
```

Vor dem Löschen

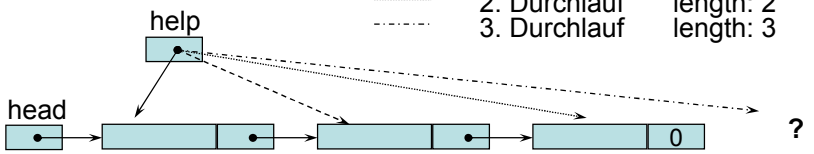
Liste - dynamisch - Länge

Länge

```
int List::Length() {
  Element* help = head;
  int length = 0;
  while(help != 0) {
   length++;
   help = help->next;
  }
  return length;
}
```

Checkpoint

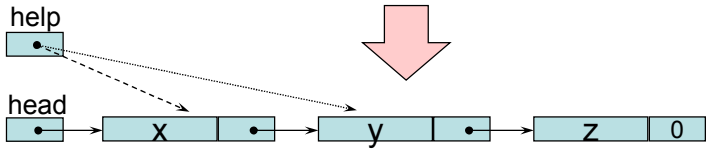
Anfang, length: 0
1. Durchlauf length: 1
2. Durchlauf length: 2
3. Durchlauf length: 3



Liste - dynamisch - Inklusionstest

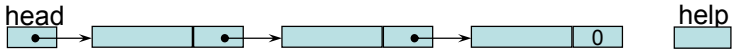
Inklusionstest

```
int List::Member(ItemType a) {
  Element* help = head;
  while(help != 0 && help->value != a)
    help = help->next;
  if(help != 0) return 1;
  else return 0;
}
```



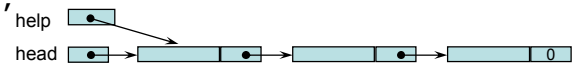
Liste - Traversieren

Sequentielles Abarbeiten einer Liste, Besuchen aller Elemente



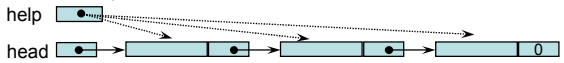
1. Initialisieren des Hilfszeigers

```
help = head;
```



2. Weitersetzen des Hilfszeigers (Position)

```
help = help -> next;
```



3. Abfrage auf Listenende (und Suchkriterium)

```
while ( help != 0 && … ) { … }
```

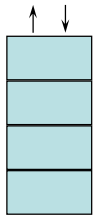
4.3 Stack

Der Stack (Kellerspeicher) ist ein Spezialfall der Liste, die die Elemente nach dem LIFO (last-in, first-out) Prinzip verwaltet

Idee des Stacks: Man kann nur auf das oberste, zuletzt daraufgelegte Element zugreifen (vergleiche Buchstapel, Holzstoß, ...)Anwendungen: Kellerautomaten, Speicherverwaltung, HP-Taschenrechner (UPN), ...

Das Verhalten des Stacks lässt sich über seine (recht einfachen) Operationen beschreiben

- push: Element am Stack ablegen
- top: Auf oberstes Element des Stacks zugreifen
- pop: Element vom Stack entfernen
- isEmpty: Test auf leeren Stack



Methoden auf Stacks

Methode 'Push'

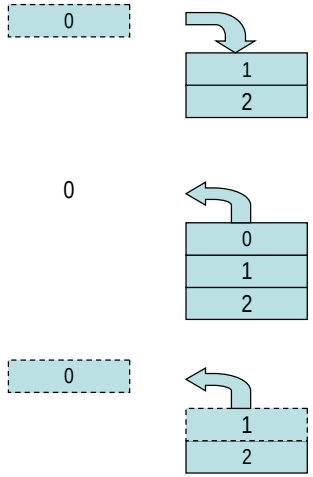
Element wird auf dem Stack abgelegt (an oberster Position).

Methode 'Top'

Liefert den Inhalte des obersten Elementes des Stacks.

Methode 'Pop'

Oberstes Element des Stacks wird entfernt.



Stack als Liste

Stack ist eine spezielle Liste, daher können die Stack-operationen durch Listenoperationen ausgedrückt werden.

Push(S,a) ⇒ Add(S,a)
Top(S) ⇒ FirstElement(S)
Pop(S) ⇒ RemoveFirst(S)

IsStackEmpty(S) ⇒
wenn Length(S) = 0 return true
sonst false

4.4 Queue

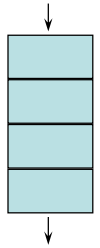
Die Queue (Warteschlange) ist ein Spezialfall der Liste, die die Elemente nach dem FIFO (first-in, first-out) Prinzip verwaltet

Idee: Die Elemente werden hintereinander angereiht, wobei nur am Ende der Liste Elemente angefügt und vom Anfang der Liste weggenommen werden können

Anwendungen: Warteschlangen, Bufferverwaltung, Stoffwechsel, Prozessmanagement, …

Einfache Operationen

- Enqueue
  - Element am Ende der Queue ablegen
- Front
  - Erstes Element der Queue zugreifen
- Dequeue
  - Erstes Element aus der Queue entfernen
- IsQueueEmpty
  - Test auf leere Queue



Methoden auf Queue

Methode ‘Enqueue’

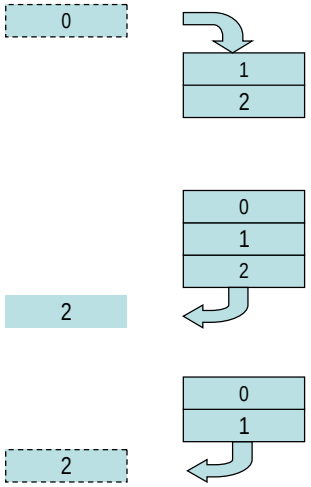
Element wird am Ende der Queue abgelegt (an letzter Position)

Methode ‘Front’

Liefert den Inhalte des ersten Elementes der Queue

Methode ‘Dequeue’

Erstes Element der Queue wird entfernt



Queue als Liste

Queue ist ebenfalls eine spezielle Liste, daher sollten alle Queueoperationen auch durch Listenoperationen ausgedrückt werden können.

Enqueue(Q,a) ⇒ Add(S,a)
Front(Q) ⇒ ?
Dequeue(Q) ⇒ ?

Nicht trivial !

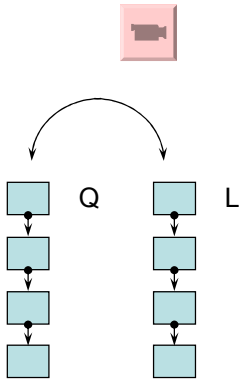
Möglichkeit (umständlich!)

Zugriff auf das erste Queueelement (letzte in der Liste) durch iteratives Entfernen aller Elemente und gleichzeitigen Aufbau einer ‘gestürzten’ Hilfsliste. Danach Vorgang umkehren.

Front(Q)

Front(Q)

```
ItemType e;     // Hilfselement
List L;         // Hilfsliste
int n = Q.Length();
for (int i = 1; i <= n - 1; i++) {
  L.Add(Q.FirstElement());
  Q.RemoveFirst();
}
e = Q.FirstElement();
n = L.Length();
for (i = 1; i <= n; i++) {
  Q.Add(L.FirstElement());
  L.RemoveFirst();
}
return e;
```



Dequeue

Dequeue(Q)

```
List L;
int l = Q.Length();
for (int i = 1; i <= n - 1; i++) {
  L.Add(Q.FirstElement());
  Q.RemoveFirst();
}
Q.RemoveFirst();
int n = L.Length();
for (i = 1; i <= n; i++) {
  Q.Add(L.FirstElement());
  L.RemoveFirst();
}
```

Zugriff auf beliebiges Listenelement

Besser: Einführen einer neuen Listenoperation

Methode 'AccessElement'

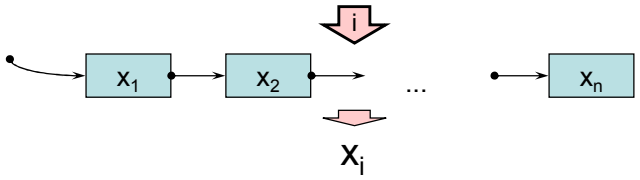
Zugriff auf ein beliebiges Listenelement über die Position in der Liste, d.h.

$L = (x_1, \ldots, x_n), i$

$\Downarrow$

AccessElement(L, i) $\rightarrow x_i$

Fehler falls Position nicht definiert



C++-Methode: **`ItemType AccessElement(position i);`**

Methode AccessElement

```
ItemType List::AccessElement(int pos) {
  Element* act = head;
  int actpos = 1;
  …
  while(actpos < pos) {
   act = act->next;
   actpos++;
  }
  return act->value;
}
```


Queue als Liste (2. Versuch)

Neuerlicher Definitionsansatz mit zusätzlicher Listenoperation etwas einfacher, aber ...

Enqueue(Q,a)	⇒	Add(Q,a)
Front(Q)	⇒	AccessElement(Q,Length(Q))
Dequeue(Q)	⇒	?

RemoveElement(Q, Length(Q))

Möglichkeit:

Analog zu positionsbezogener Zugriffsfunktion eine positionsbezogene Teilungsfunktion entwerfen, die eine Liste an einer vorgegebener Stelle in 2 Teillisten zerlegt.

Löschen eines beliebigen Listenelements

Methode 'RemoveElement'

Löschen eines beliebiges Listenelement über die Position in der Liste, d.h.

$L = (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n), i$

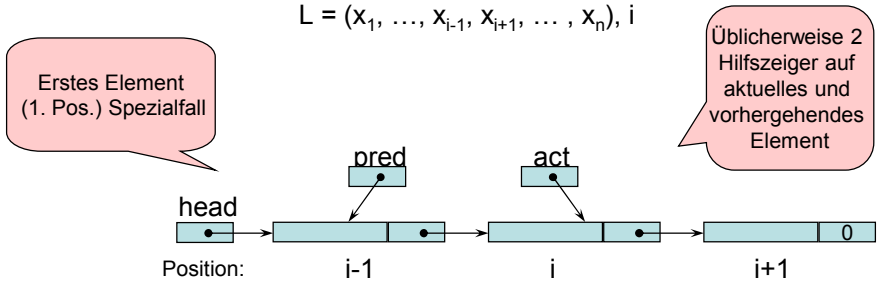
$\Downarrow$

RemoveElement(L, i) $\rightarrow L$

Fehler falls Position nicht definiert

$\Downarrow$

$L = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n), i$



Methode RemoveElement

```
void List::RemoveElement(int pos) {
  Element* pred, * act;
  int actpos = 2;
  if(pos == 1) RemoveFirst();
  else {
    pred = head;
    act = head->next;
    while(act != 0 && actpos < pos) {
      pred = act;
      act = act->next;
      actpos++;
    }
    if(act == 0) return;
    pred->next = act->next;
    delete act;
  }
}
```

erstes Element

2 Hilfszeiger

Methode AddElement

Analog auch AddElement mgl.: Einfügen an beliebiger Stelle

```
void List::AddElement(ItemType a, int pos) {
  Element* pred, * act;
  int actpos = 2;
  if(pos == 1) Add(a);
  else {
    pred = head;
    act = head->next;
    while(act != 0 && actpos < pos) {
          pred = act;
          act = act->next;
          actpos++;
    }
    pred->next = new Element;
    pred->next->value = a;
    pred->next->next = act;
  }
}
```

erstes Element

2 Hilfszeiger

Stack - Queue Klasse

C++ Klassen Deklaration (Skizze)

Stack

```
class Stack {
…
public:
       Stack();
       bool Push(ItemType a);
       ItemType Top();
       bool Pop();
       bool IsStackEmpty();
}
```

Queue

```
class Queue {
…
public:
       Queue();
       bool Enqueue(ItemType a);
       ItemType Front();
       bool Dequeue();
       bool IsQueueEmpty();
}
```


Vergleich "unserer" Datenstrukturen

Generelle Unterscheidung zwischen statischer und dynamischer Realisierung

- statische R.: contigous memory, Felder
- dynamische R.: dynamic memory, dynamische Objekte

Datenverwaltung

- Einfügen und Löschen wird unterstützt

Datenmenge

- statische R.: beschränkt, abhängig von der Feldgröße
- dynamische R.: unbeschränkt
  - abhängig von der Größe des vorhandenen Speicherplatzes

eher simple Modelle

Aufwandsvergleich "unserer" Listen Implementationen

	Liste statisch	Liste dynamisch
Speicherplatz	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	**O(n)**
Add	O(1)	O(1)
FirstElement	O(1)	O(1)
RemoveFirst	O(1)	O(1)
Length	O(1)	**O(n)**
Member	**O(n)**	**O(n)**

	Liste statisch	Liste dynamisch
AccessElement	O(1)	**O(n)**
RemoveElement	**O(n)**	**O(n)**
AddElement	**O(n)**	**O(n)**

Achtung: Eigentlicher Aufwand O(n) in Add und RemoveFirst versteckt

Aufwandsvergleich "unserer" Stack - Queue Implementationen

	Stack statisch	Stack dynamisch
Speicherplatz	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	O(n)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Top	O(1)	O(1)
IsStackEmpty	O(1)	O(1)

	Queue statisch	Queue dynamisch
Speicherplatz	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	**O(n)**
Enqueue	O(1)	O(1)
Dequeue	O(1)	**O(n)**
Front	O(1)	**O(n)**
IsQueueEmpty	O(1)	O(1)

4.5 Spezielle Listen

Doubly Linked List
- doppelt verkettet Liste

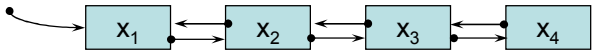
Circular List
- Zirkulär verkettete Liste

Ordered List
- Geordnete Liste

Double Ended List
- Doppelköpfige Liste

Doubly Linked List

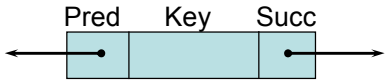
Doppelt verkettete Liste



jedes Element besitzt 2 Zeiger, wobei der eine auf das vorhergehende und der andere auf das nachfolgende Element zeigt

Basis-Operationen einfach

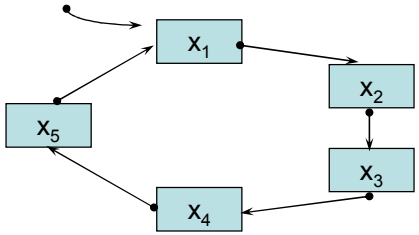
```
class Node {
     KeyType Key;
     Node* Pred;
     Node* Succ;
}
```



Circular List

Zirkulär verkettete Liste

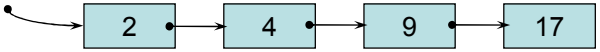
- Zeiger des letzten Element verweist wieder auf das erste Element
- Ring Buffer
- Vorsicht beim Eintragen und Löschen des ersten Elementes!



Ordered List

Geordnete Liste

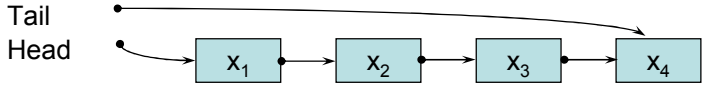
- Elemente werden entsprechend ihres Wertes in die Liste an spezifischer Stelle eingetragen
- Meist der Größe nach geordnet
- Eintragen an spezifischer Stelle, die erst gefunden werden muß → Traversieren



Double Ended List

Liste mit 2 “Köpfen”

- Jede Liste besitzt 2 Zeiger, die zum Kopf und zum Ende der Liste zeigen
- Vereinfacht das Einfügen am Kopf und am Ende der Liste



Was nehmen wir mit?

Listen

- Operationen
- Speicherung
- Contigous - Scattered memory

Stack – Queue

Vergleich

Spezielle Listen

- Doubly Linked List
- Circular List
- Ordered List
- Double Ended List

Kapitel 5
Bäume

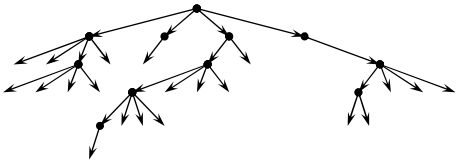
5.1 Definition Bäume

Ein Baum (tree) ist ein spezieller Graph, der eine hierarchische Struktur über eine Menge von Objekten definiert

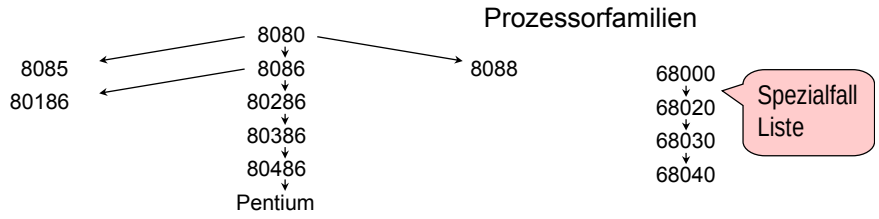
Intuitives Vorstellungsmodell - Nicht-lineare Datenstruktur

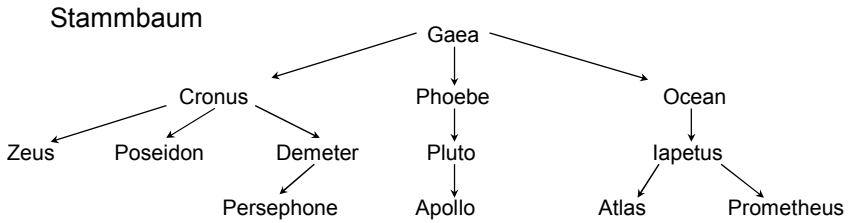
Anwendungen

Repräsentieren Wissen, Darstellung von Strukturen, Abbildung von Zugriffspfaden, Analyse hierarchischer Zusammenhänge, ...



Beispiel Bäume





Definition Bäume (1)

Ein Baum besteht aus einer Menge von *Knoten*, die durch *gerichtete Kanten* verbunden sind

Ein *Pfad oder Weg* ist eine Liste sich unterscheidender Knoten, wobei aufeinander folgende Knoten durch eine Kante verbunden sind

Es gibt genau einen Pfad der 2 Knoten miteinander verbindet und jeder Knoten hat nur einen direkten Vorgänger; alle Vorgänger sind von ihm selbst verschieden (definierende Eigenschaft eines Baumes)

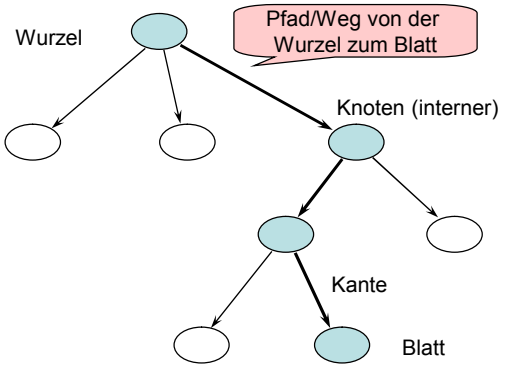
Ein Baum enthält daher keine Kreise, d.i. ein Pfad bei dem der Startknoten gleich dem Endknoten ist

Komponenten

Die *Wurzel* ist der einzige Knoten mit nur wegführenden Kanten

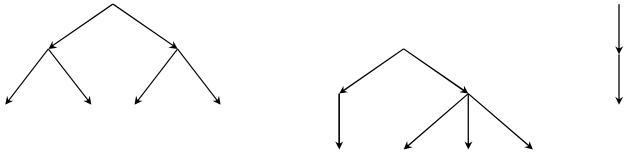
Knoten von denen keine Kanten wegführen heißen *Blatt (leaf)*

Knoten, in die Kanten hinein- und von denen Kanten wegführen, heißen *interne Knoten*

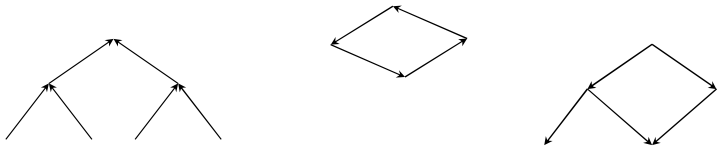


Gültige und ungültige Bäume

Gültige Bäume



Ungültige Bäume



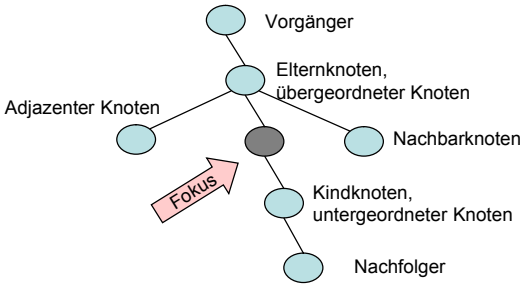
Definition Bäume (1)

Jeder Knoten (mit Ausnahme der Wurzel) hat genau einen Knoten über sich, den man als *Elternknoten* oder *übergeordneten* Knoten bezeichnet

Die Knoten direkt unterhalb eines Knotens heißen *Kindknoten* oder *untergeordnete* Knoten

Transitive Eltern werden als Vorgänger bzw. transitive Kinder als Nachfolger bezeichnet

Räumlich nebeneinander-liegende Knoten auf derselben Tiefe heißen *adjazent* oder *Nachbarn*

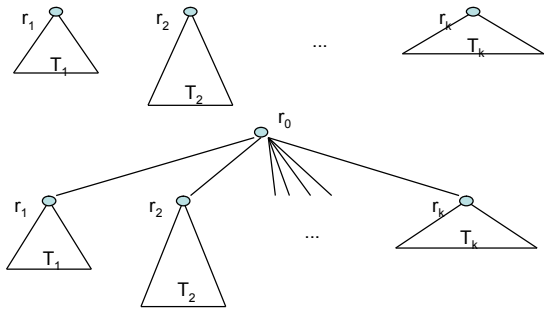


Rekursive Baum Definition

Ein einzelner Knoten ohne Kanten ist ein Baum

Seien $T_1, \ldots, T_k$ ($k > 0$) Bäume ohne gemeinsame Knoten. Die Wurzeln dieser Bäume seien $r_1, \ldots, r_k$. Ein Baum T_0 mit der Wurzel r_0 besteht aus den Knoten und Kanten der Bäume $T_1, \ldots, T_k$ und neuen Kanten von r_0 zu den Knoten $r_1, \ldots, r_k$

Der Knoten r_0 ist dann die neue Wurzel und $T_1, \ldots, T_k$ sind Unterbäume von T_0



r_i... Wurzel des Baumes i
T_i... Unterbaum i

Länge, Höhe, Tiefe

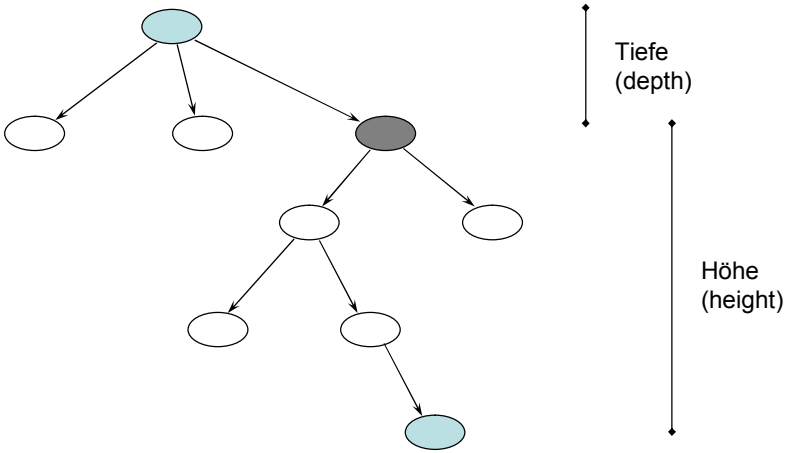
Die *Länge eines Weges* zwischen 2 Knoten entspricht der Anzahl der Kanten auf dem Weg zwischen den beiden Knoten

Die *Höhe eines Knoten* ist die Länge des längsten Weges von diesem Knoten zu den erreichbaren Blättern

Die *Tiefe eines Knoten* ist die Länge des Weges zur Wurzel

Die *Höhe eines Baumes* entspricht der Höhe der Wurzel

Länge, Höhe, Tiefe grafisch



Beispiel Parse Tree

Parsierungsbäume (parse trees) analysieren Anweisungen bzw. Programme einer Programmiersprache bezüglich einer gegebenen Grammatik

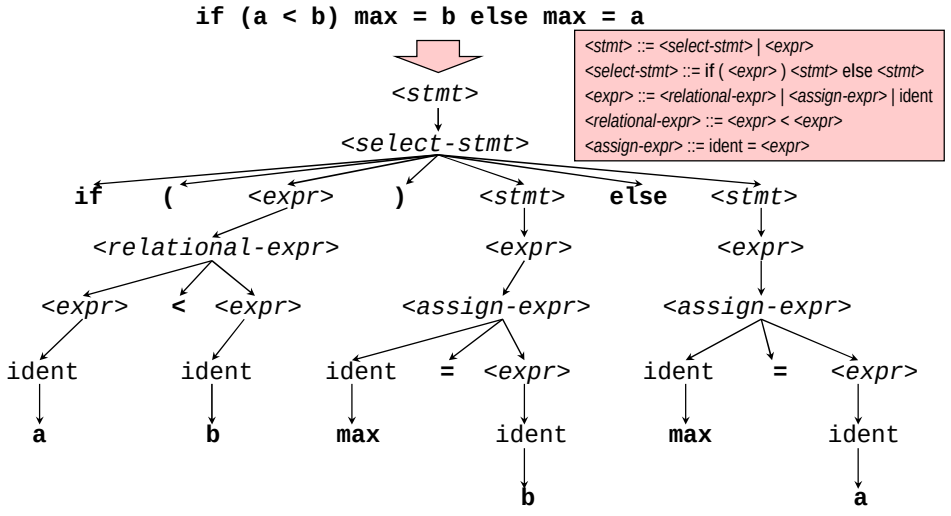
Eine Grammatik definiert Regeln, wie und aus welchen Elementen eine Programmiersprache aufgebaut ist

Beispiel: C++ (Ausschnitt)

<stmt>	Nonterminal Symbole (werden aufgelöst)
if, ident	Terminalsymbole (nicht mehr aufgelöst)
::=, |	Grammatiksymbolik

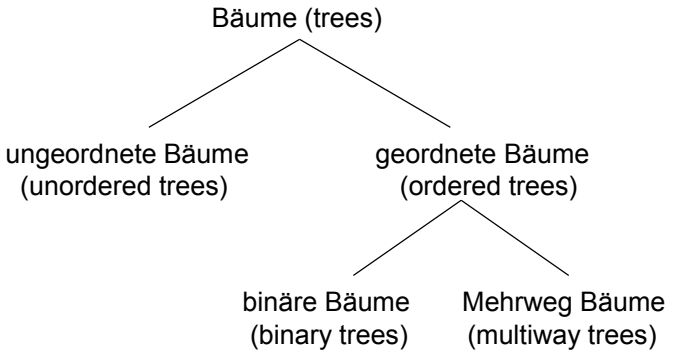
```
<stmt> ::= <select-stmt> | <expr>
<select-stmt> ::= if ( <expr> ) <stmt> else <stmt>
<expr> ::= <relational-expr> | <assign-expr> | ident
<relational-expr> ::= <expr> < <expr>
<assign-expr> ::= ident = <expr>
```


Beispiel Parse Tree (2)



Spezielle Bäume

Die Ordnung bezieht sich auf die Position der Knoten im Baum (Knoten A links von Knoten B) und nicht auf die Werte der Knotenelemente



5.2 Binäre Bäume

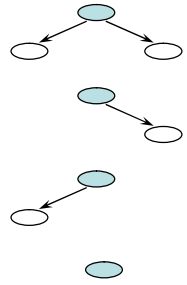
Binäre Bäume sind geordnete Bäume, in denen jeder Knoten maximal 2 Kinder hat und die Ordnung der Knoten mit links und rechts festgelegt ist

Binärer Baum mit 2 Kindern

Binärer Baum mit rechtem Kind (right child)

Binärer Baum mit linkem Kind (left child)

Binärer Baum ohne Kind (Blatt)

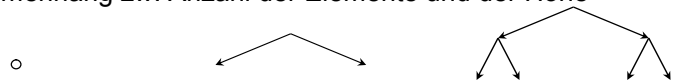


Binäre Bäume - Eigenschaften

Häufiger Einsatz in Algorithmen

Effiziente Manipulationsalgorithmen

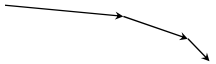
Zusammenhang zw. Anzahl der Elemente und der Höhe



Höhe 0	Höhe 1	Höhe 2
1 Knoten	3 Knoten	7 Knoten
1 Blatt	2 Blätter	4 Blätter

In jeder neuen Ebene wird im optimalen Fall die Anzahl der Blätter verdoppelt

Entartung möglich: Jeder Knoten hat genau ein Kind → entspricht linearer Liste



Formen binärer Bäume

Leerer binärer Baum

Binärer Baum ohne Knoten

Voller binärer Baum

Jeder Knoten hat keine oder 2 Kinder

Perfekter binärer Baum

Ein voller binärer Baum bei dem alle Blätter dieselbe Tiefe besitzen

Kompletter binärer Baum

Ein perfekter binärer Baum mit der Ausnahme, dass die Blattebene nicht vollständig, dafür aber von links nach rechts, gefüllt ist

Höhen-balanzierter binärer Baum

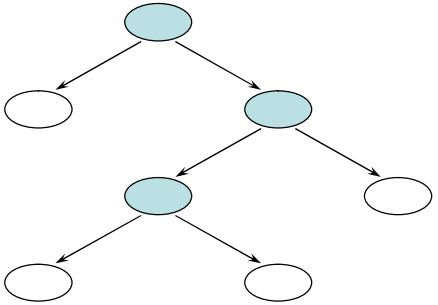
Für jeden Knoten ist der Unterschied der Höhe des linken und rechten Kindes maximal 1

Voller binärer Baum

Full binary tree

Jeder Knoten im Baum hat keine oder genau 2 Kinder.

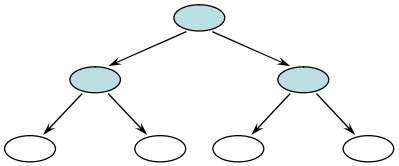
Mit anderen Worten, kein Knoten besitzt nur 1 Kind



Perfekter binärer Baum

Perfect binary tree

Ein voller binärer Baum (alle Knoten haben keine oder genau 2 Kinder) bei dem alle Blätter dieselbe Tiefe besitzen.



Eigenschaften des perfekten binären Baumes

Eigenschaften

Frage: welche Höhe h muss ein perfekter binärer Baum haben um n Blätter zu besitzen

In jeder Ebene Verdoppelung, d.h. $2^h = n$

$h * \log 2 = \log n$

$h = \log_2 n = \text{ld } n$

Ein perfekter binärer Baum der Höhe h besitzt 2^{h+1}-1 Knoten

davon sind 2^h Blätter

Beweis mit vollständiger Induktion

Zusammenhang zwischen Knoten/Blätter und Höhe:

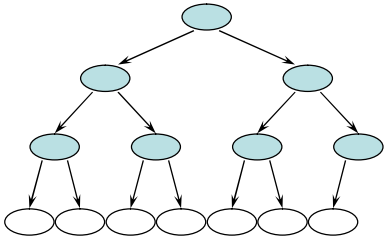
O(n) Knoten/Blätter : O(*log n)* Höhe

wichtigste Eigenschaft von Bäumen

Kompletter binärer Baum

Complete binary tree

Ein kompletter binärer Baum ist ein perfekter binärer Baum mit der Ausnahme, dass die Blattebene nicht vollständig, dafür aber von links nach rechts, gefüllt ist



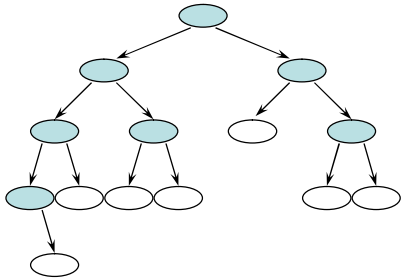
Eigenschaft

Ein kompletter binärer Baum mit n Knoten hat eine Höhe von maximal $h = \lfloor \log_2 n \rfloor$

Höhen-balanzierter binärer Baum

Height-balanced binary tree

- Für jeden Knoten ist der Unterschied der Höhe des linken und rechten Kindes maximal 1
  - Dies garantiert, dass lokal für jedes Kind die Balanzierungs-eigenschaft relativ gut erfüllt ist, global aber die gesamten Baumdifferenzen größer sein können → einfachere Algorithmen



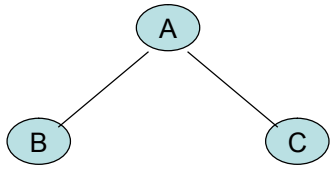
Baum Traversierung

Traversieren eines Baumes bezeichnet das systematische Besuchen aller seiner Knoten

- Unterschiedliche Methoden unterscheiden sich in der Reihenfolge der besuchten Knoten

Mögliche Reihenfolgen

- A, B, C
- B, A, C
- B, C, A
- …

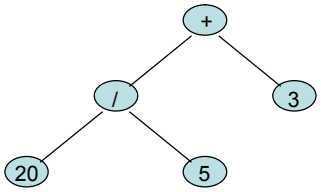


Expression Tree

Systematische Auswertung eines mathematischen Ausdrucks

Ein mathematischer Ausdruck kann in Form eines Expression Trees angegeben werden

(20 / 5) + 3



- Blätter repräsentieren Operanden (Zahlenwerte), interne Knoten Operatoren
- Baumdarstellung erspart Klammernnotation

Traversierungsalgorithmus

Traversierungsalgorithmen bestehen prinzipiell aus 3 verschiedenen Schritten

- Bearbeiten eines Knotens (process node)
- Rekursiv besuchen und bearbeiten des linken Kindes (visit left child)
- Rekursiv besuchen und bearbeiten des rechten Kindes (visit right child)

Durch unterschiedliches Anordnen der 3 Schritte unterschiedliche Reihenfolgen

3 Bearbeitungsreihenfolgen interessant

- Preorder Traversierung
- Postorder Traversierung
- Inorder Traversierung

Preorder Traversierung

Algorithmus

```
preorder(node) {
  if(node != 0) {
    process(node)
    preorder(left child)
    preorder(right child)
  }
}
```

Besucht die Knoten im Baum in *Prefix-Notation-Reihenfolge*

(Operator, $Operand_1$, $Operand_2$)

Faustregel

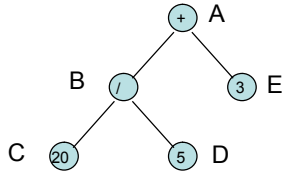
Knoten bearbeiten beim 1. Besuch

Anwendung

LISP, Assembler

Beispiel

20/5+3



Process-Reihenfolge: A B C D E

Notation-Reihenfolge: + / 20 5 3

Postorder Traversierung

Algorithmus

```
postorder(node) {
  if(node != 0) {
    postorder(left child)
    postorder(right child)
    process(node)
  }
}
```

Postfix-Notation-Reihenfolge

($Operand_1$, $Operand_2$, Operator)

Faustregel

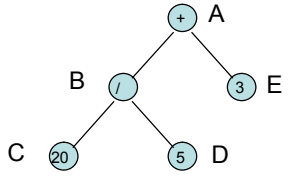
Knoten bearbeiten beim letzten Besuch

Anwendung

Invers Polish Notation, HP Taschenrechner, FORTH

Beispiel

20/5+3



Process-Reihenfolge: C D B E A

Notation-Reihenfolge: 20 5 / 3 +

Inorder Traversierung

Algorithmus

```
inorder(node) {
  if(node != 0) {
    inorder(left child)
    process(node)
    inorder(right child)
  }
}
```

Infix-Notation-Reihenfolge

($Operand_1$, Operator, $Operand_2$)

Faustregel

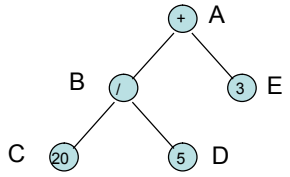
Knoten bearbeiten beim 2. oder letzten Besuch

Anwendung

einfacher algebraischer Taschenrechner (ohne Klammern)

Beispiel

20/5+3



Process-Reihenfolge: C B D A E

Notation-Reihenfolge: 20 / 5 + 3

5.3 Binäre Suchbäume

In einem Binärbaum besitzt jeder (interne) Knoten eine linke und eine rechte Verbindung, die auf einen Binärbaum oder einen externen Knoten verweist

Verbindungen zu externen Knoten heißen Nullverbindungen, externe Knoten besitzen keine weiteren Verbindungen

Ein binärer Suchbaum (binary search tree, BST) ist ein Binärbaum, bei dem jeder interne Knoten einen Schlüssel besitzt

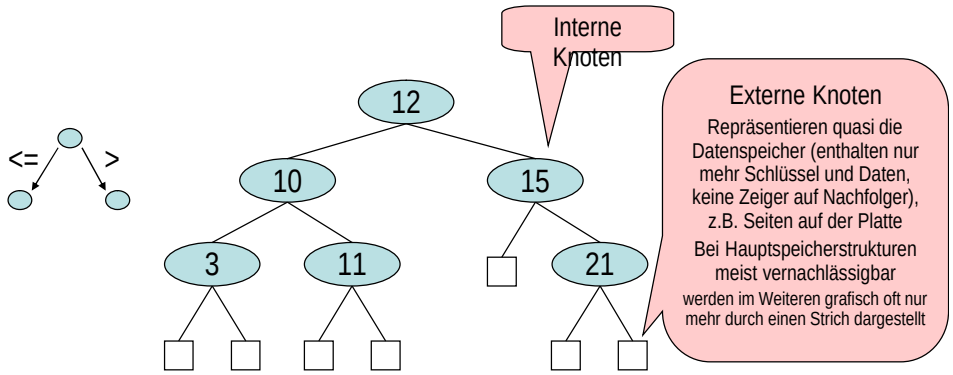
externe Knoten besitzen keine Schlüssel

Auf den Schlüsseln ist eine lineare Ordnung “<“ definiert

Wenn x und y verschieden sind, so gilt x<y oder y<x; ist x<y und y<z, dann gilt auch x<z

Schlüsseleigenschaft im BST

Für jeden internen Knoten gilt, dass alle Werte der Nachfolger im linken Unterbaum kleiner (oder gleich) dem Knotenwert und die Werte der Nachfolger im rechten Unterbaum größer als der Knotenwert sind



Externe Knoten

Repräsentieren quasi die Datenspeicher (enthalten nur mehr Schlüssel und Daten, keine Zeiger auf Nachfolger), z.B. Seiten auf der Platte

Bei Hauptspeicherstrukturen meist vernachlässigbar

werden im Weiteren grafisch oft nur mehr durch einen Strich dargestellt

Eigenschaften binärer Suchbäume

Verwaltung beliebig großer Datenbestände

- dynamische Struktur

Effiziente Administration

- Aufwand proportional zur Höhe des Baumes und nicht zur Anzahl der Elemente
- Einfügen, Zugriff und Löschen im Durchschnitt von O(log n)
- Signifikante Verbesserung im Vergleich zum linearen Aufwand (O(n)) bei Liste
- Zugriff auf Elemente in sortierter Reihenfolge durch inorder Traversierung

Operationen

Erstellen

- Erzeugen eines leeren Suchbaumes

Einfügen

- Einfügen eines Elementes in den Baum unter Berücksichtigung der Ordnungseigenschaft

Suche

- Test auf Inklusion

Löschen

- Entfernen eines Elementes aus dem Baum unter Erhaltung der Ordnungseigenschaft

Ausgabe

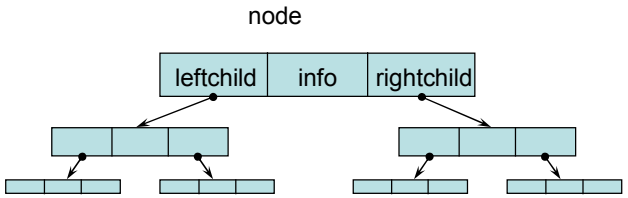
- Ausgabe aller Elemente in sortierter Reihenfolge

...

Klassendefinition

```
typedef int ItemType;
class SearchTree {
  class node {
  public:
    ItemType info;
    node * leftchild, * rightchild;
    node(ItemType x, node * l, node * r) {info=x; leftchild=l; rightchild=r;}
  };
  typedef node * link;
  link root;
  void AddI(ItemType);
  void AddR(link&, ItemType);
  bool MemberI(ItemType);
  bool MemberR(link, ItemType);
  void PrintR(link, int);
public:
  SearchTree(){root = 0;}
  void Add(ItemType);
  int Delete(ItemType);
  bool Member(ItemType);
  void Print();
};
```

Einfachheitshalber in **node** alles **public**

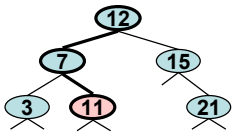


Suchen in einem binären Baum

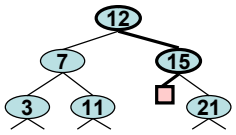
Beim Suchen eines Schlüssels wird ein Pfad von der Wurzel abwärts verfolgt

- Bei jedem internen Knoten wird der Schlüssel x mit dem Suchschlüssel s verglichen
- Falls x = s liegt ein Treffer vor, falls s <= x suche im linken Teilbaum, sonst im rechten
- Wenn man einen externen Knoten erreicht, war die Suche erfolglos

Erfolgreiche Suche (z.B. 11)



Erfolglose Suche (z.B. 13)

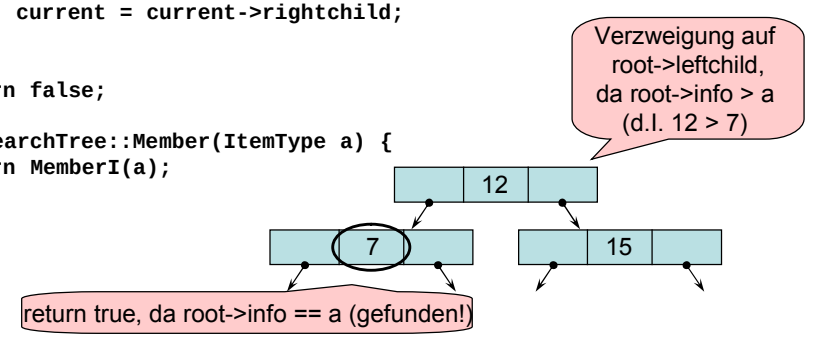


Suche (iterativ)

```
bool SearchTree::MemberI(ItemType a) {
  if(root) {
    link current = root;
    while(current) {
      if(current->info == a) return true;
      if(a < current->info)
        current = current->leftchild;
      else
        current = current->rightchild;
    }
  }
  return false;
}
bool SearchTree::Member(ItemType a) {
  return MemberI(a);
}
```

Aufruf:

```
SearchTree t;
…
t.Member(7);
```



Suche (rekursiv)

```
bool SearchTree::MemberR(link h, ItemType a) {
  if(!h) return false;
  else {
    if(a == h->info) return true;
    if(a < h->info)
      return MemberR(h->leftchild, a);
    else
      return MemberR(h->rightchild, a);
  }
}
bool SearchTree::Member(ItemType a) {
  return MemberR(root, a);
}
```

Aufruf:

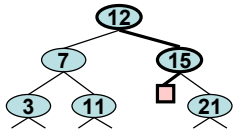
```
SearchTree t;
…
t.Member(7);
```


Einfügen in einem binären Suchbaum

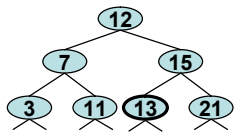
Das Einfügen entspricht einer erfolglosen Suche (wobei gleiche Schlüssel übergangen werden) und dem Anfügen eines neuen Knotens an der Nullverbindung wo die Suche endet (anstelle des externen Knotens)

Einfügen 13

Suche



Knoten anfügen



Einfügen (iterativ)

Ähnlich dem Einfügen in eine Liste (2 Hilfszeiger)

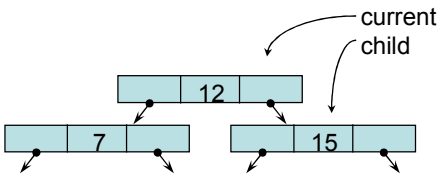
```
void SearchTree::AddI(ItemType a) {
  if(root) {
    link current = root;
    link child;
    while(1) {
      if(a <= current->info) {
        child = current->leftchild;
        if(!child) {current->leftchild = new node(a, 0, 0); return;}
      } else {
        child = current->rightchild;
        if(!child) {current->rightchild = new node(a, 0, 0); return;}
      }
      current = child;
    }
  } else {
    root = new node(a, 0, 0);
    return;
  }
}
void SearchTree::Add(ItemType a) {
  AddI(a);
}
```

Hilfszeiger: **current, child**

Aufruf:

```
SearchTree t;
…
t.Add(7);
```

LUCAS



Einfügen (rekursiv)

```
void SearchTree::AddR(link& h, ItemType a) {
  if(!h) {h = new node(a, 0, 0); return;}
  if(a <= h->info)
    AddR(h->leftchild, a);
  else
    AddR(h->rightchild, a);
}
void SearchTree::Add(ItemType a) {
  AddR(root, a);
}
```

Aufruf

```
SearchTree t;
…
t.Add(7);
```

Man beachte die Verwendung eines Referenzparameters (**link&**), erspart die 2 Hilfszeiger

Traversieren

```
void SearchTree::PrintR(link h, int n) {
  if(!h) return;
  PrintR(h->rightchild, n+2);
  for(int i = 0; i < n; i++) cout << " ";
  cout << h->info << endl;
  PrintR(h->leftchild, n+2);
}
void SearchTree::Print() {
  PrintR(root, 0);
}
```

Inorder Traversierung

Gibt Baum um 90 Grad gegen den Uhrzeigersinn verdreht aus

Aufruf

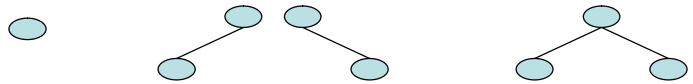
```
SearchTree t;
…
t.Print();
```


Löschen in einem binären Baum

Der Löschvorgang unterscheidet 3 Fälle

Löschen von internen Knoten mit

(1) keinem (2) einem (3) zwei internen Knoten als Kinder



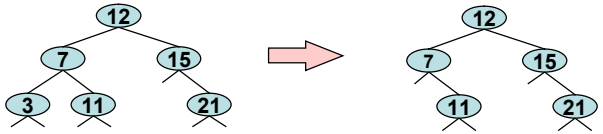
Fälle 1 und 2 sind einfach durch Anhängen des verbleibenden Teilbaums an den Elternknoten des zu löschenden Knotens zu lösen.

Für Fall 3 muss ein geeigneter Ersatzknoten gefunden werden. Hierzu eignen sich entweder der kleinste im rechten (Inorder Nachfolger) oder der größte im linken Teilbaum (Inorder Vorgänger)

Löschen (2)

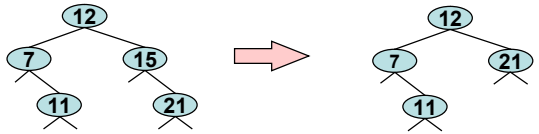
Fall 1

Löschen von Knoten 3 durch Ersetzen des linken Kindzeigers von Knoten 7 durch eine Nullverbindung (externer Knoten)



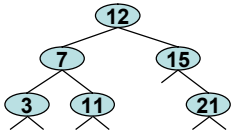
Fall 2

Löschen von Knoten 15 durch Einhängen des Knoten 21 als rechtes Kind von 12



Löschen (3)

Fall 3



Im Falle des Löschens von 12 eignen sich als Ersatz die Knoten mit den Werten 11 (größte im linken) oder 15 (der kleinste im rechten Teilbaum).
Dies resultiert in 2 möglichen Bäumen:

Somit muss der Löschvorgang für Fall 3 in zwei Teile zerlegt werden:

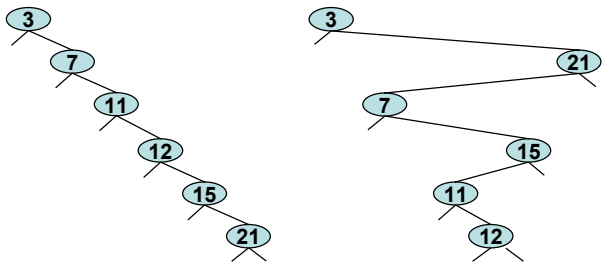
1. Finden eines geeigneten Ersatzknotens
2. Ersetzen des zu löschenden Knotens (was wiederum aus dem Entfernen des Ersatzknotens aus seiner ursprünglichen Position (entspricht Fall 1 oder 2) und dem Einhängen an der neuen Stelle besteht)

Laufzeit

Erwartungswert der Einfüge- und Suchoperationen bei n zufälligen Schlüsselwerten ist ungefähr 1,39 ld n

Im ungünstigsten Fall kann der Aufwand zu ungefähr n Operationen „entarten“

Beispiele für ungünstige Suchbäume



Analyse binärer Suchbaum

Datenverwaltung

unterstützt Einfügen und Löschen

Datenmenge

unbeschränkt

Modelle

Hauptspeicherorientiert

Unterstützung komplexer Operationen

Bereichsabfragen, Sortierreihenfolge

Laufzeit

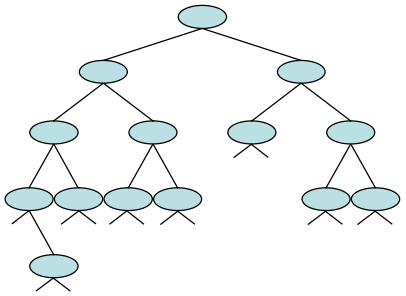
Speicherplatz	O(n)
Konstruktor	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Bitte beachten:
beträchtlicher konstanter Aufwand ist notwendig!

5.4 Höhenbalanzierter binärer Baum

Höhenbalanziert (bekannt!): Für jeden Knoten ist der Unterschied der Höhe des linken und rechten Kindes maximal 1

Geeignete Algorithmen für Einfügen und Löschen erhalten die Balanzierungseigenschaft und vermeiden dadurch die „Entartung“ der Laufzeit zu O(n) und garantieren O(log n)



AVL Bäume

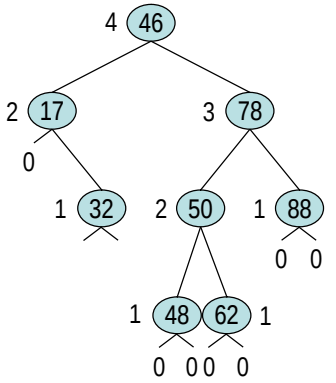
AVL Bäume sind höhenbalanzierte binäre Suchbäume

Adelson-Velski und Landau, 1962

Beispiel

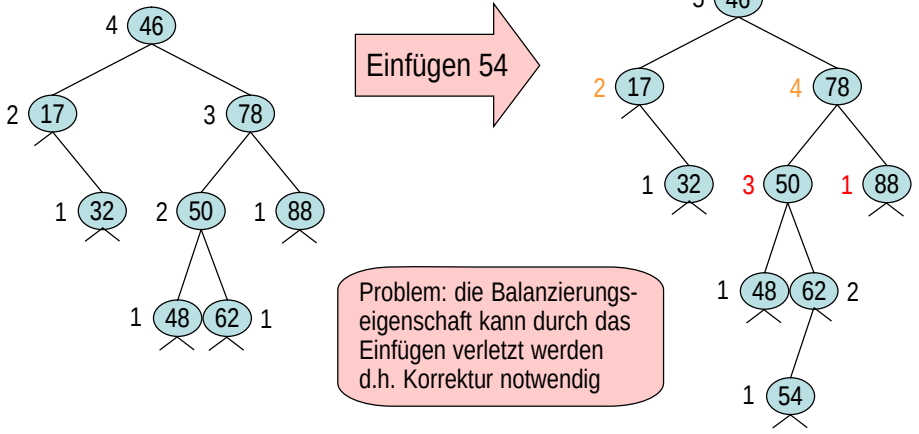
Höhen sind neben den Knoten angegeben

Anmerkung: externe Knoten nicht vergessen, sie haben die Höhe 0!



Einfügen in einen AVL Baum

Das Einfügen verläuft analog zum Einfügen im binären Suchbaum

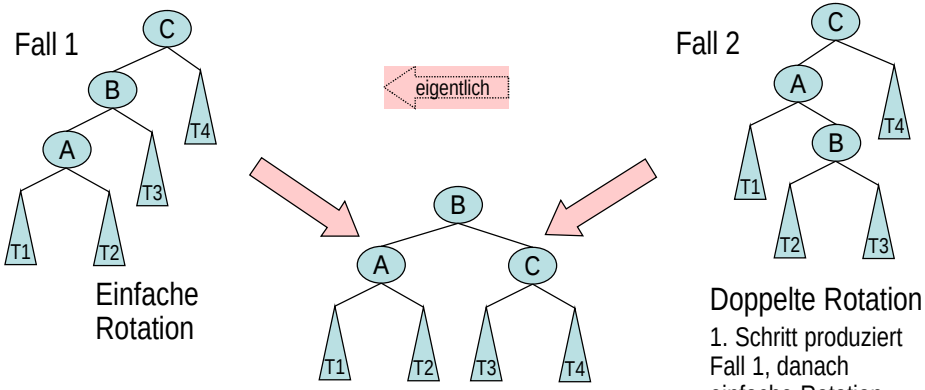


Rotationen

Balanzierungskorrektur durch Knotenrotationen

2 generelle Fälle (einfach und doppelt) zu unterscheiden

Jeweils 2 weitere symmetrische Fälle

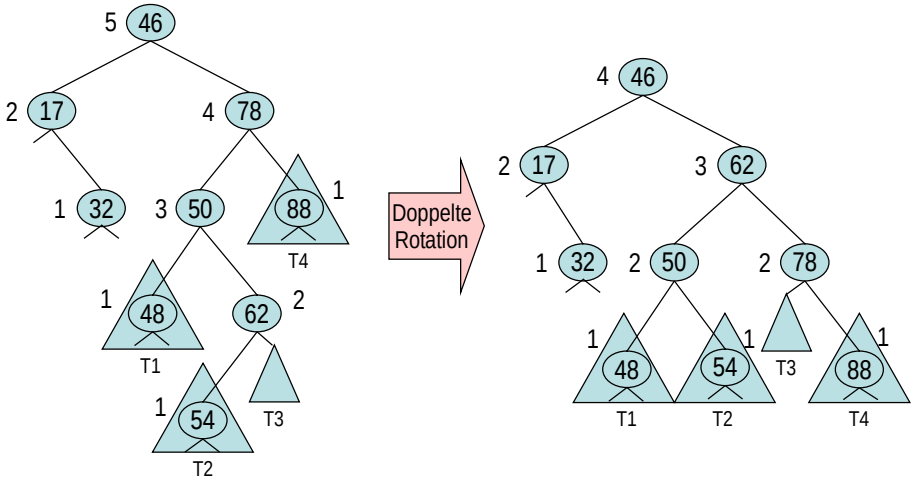


Einfache Rotation

Doppelte Rotation

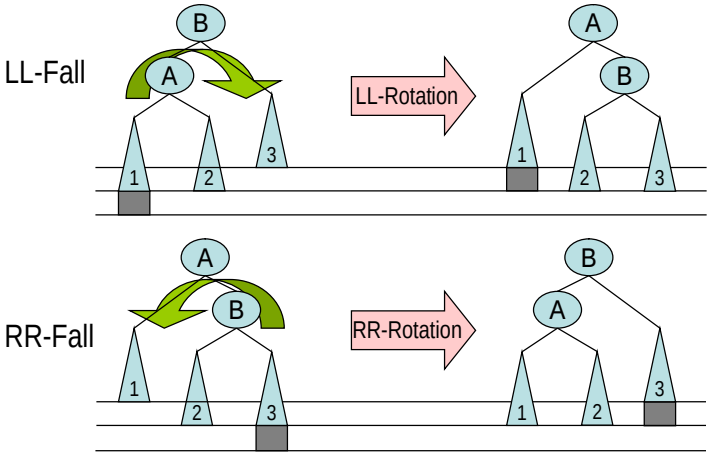
1. Schritt produziert Fall 1, danach einfache Rotation

Einfügen Beispiel



Einfache Rotationen

Nach Einfügen eines Elementes ( ) Baum nicht mehr balanziert
Durch Symmetrie 2 einfache Rotationsfälle zu unterscheiden



„Fall“ Bestimmung

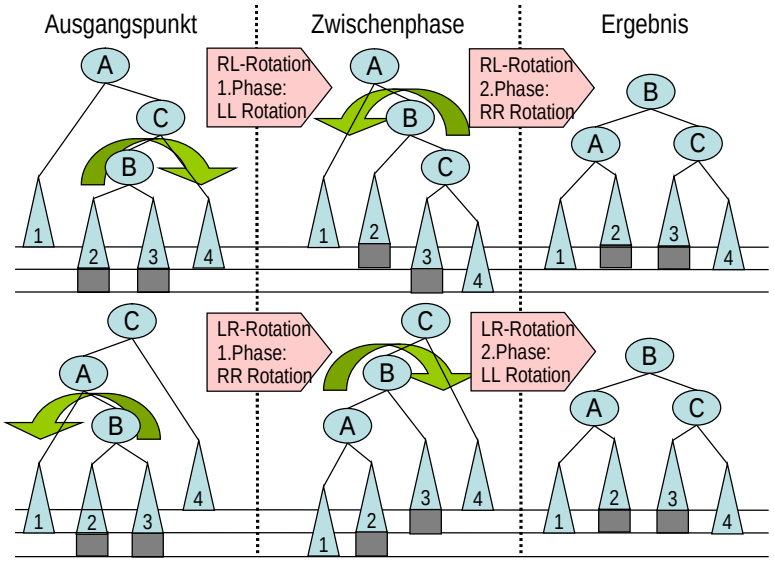
1. Ausgehend vom Knoten, bei dem die Balanzierung verletzt ist, überprüft man, welcher Teilbaum länger ist (L oder R).
2. Danach beim Kindknoten im nicht balanzierten Teilbaum feststellen, welcher weiterer (Sub)-Teilbaum länger ist (L oder R) (bei gleicher Länge einfacher Fall)

Doppelte Rotationen

Die doppelte Rotation besteht eigentlich aus 2 einfachen Rotationen

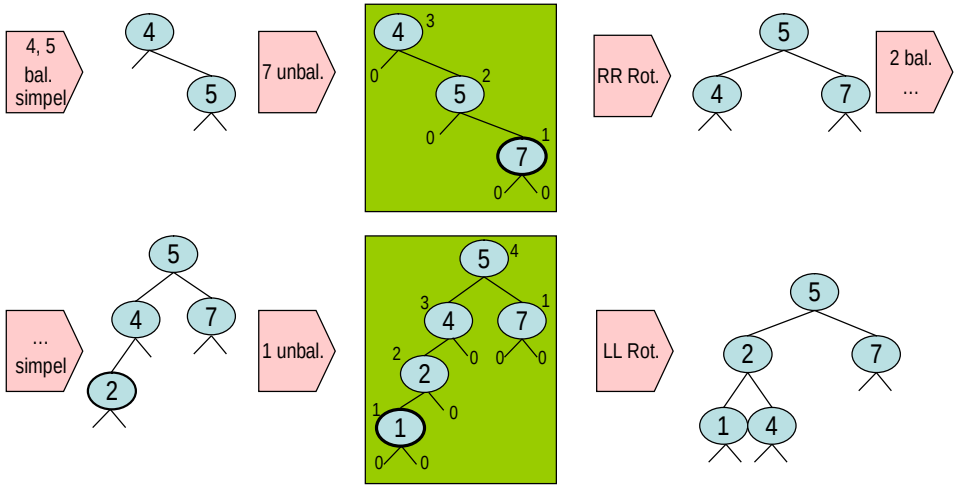
Die RL Rotation aus einer LL Rotation des rechten Subbaumes und danach einer RR Rotation des gesamten Baumes der nicht balanziert ist

(die LR Rotation funktioniert entsprechend umgekehrt)

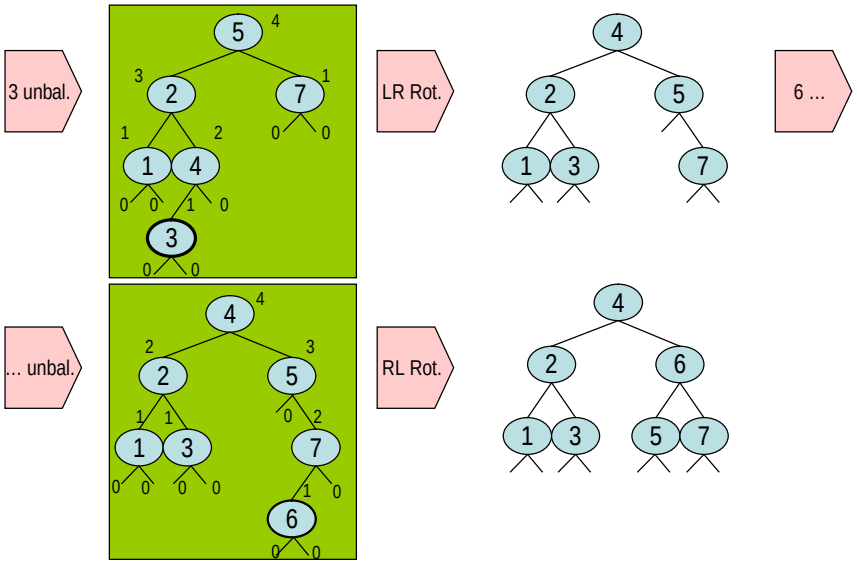


Beispiel Einfügen im AVL Baum

Einfügereihenfolge: 4,5,7,2,1,3,6



Beispiel Einfügen im AVL Baum (2)



Löschen im AVL Baum

Das Löschen funktioniert analog zum Löschen im binären Suchbaum

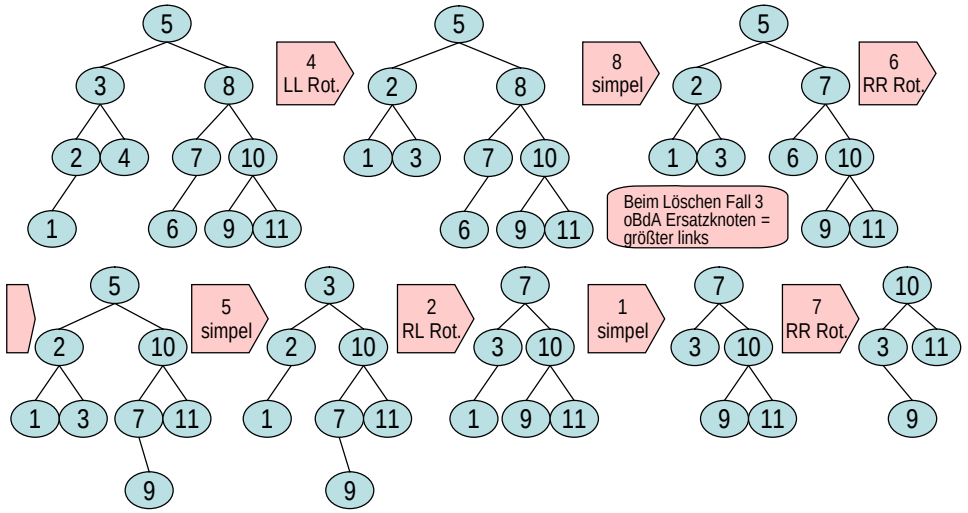
Gelöschte Knoten können zu nicht balanzierten Bäumen führen

Lösung durch Rotationen analog zum Einfügen

- Eine (Lösch) Rotation kann aber die Balanzierung eines anderen Knoten verletzen
- Daher kann ein Löschvorgang eine Serie von Rotationen auslösen um Unbalanziertheiten vom Löschort bis zur Wurzel aufzulösen
- Die ist unterschiedlich zum Einfügen, wo mit einer Rotation die Unbalanziertheit behoben werden kann

Beispiel Löschen im AVL Baum

Löschreihenfolge: 4,8,6,5,2,1,7



Analyse AVL Baum

Datenverwaltung

- unterstützt Einfügen und Löschen

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Laufzeit

Speicherplatz	O(n)
Rotation	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

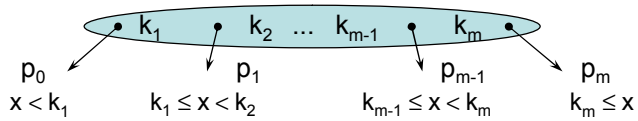
Heuristische Analyse (Wirth 79) zeigt, dass eine Rotation bei jeder 2-ten Einfügeoperationen und nur bei jeder 5-ten Löschoperation vorkommt

Die Höhe eines AVL-Baumes mit n Knoten ist $\leq 1.45 \log n$, d.h. höchstens 45 % größer als erforderlich.

5.5 Mehrwegbäume

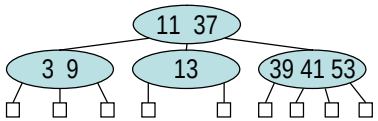
Mehrwegbäume sind Bäume mit Knoten, die mehr als 2 Kinder besitzen können

Intervallbasierten Suchdatenstrukturen



Knoten besteht aus einer Menge von m Schlüsselwerten k_1, k_2, …, k_m und m+1 Verweisen (Kanten) p_0, p_2, …, p_m, sodass für alle Schlüssel x_j im Unterbaum, der durch p_i referenziert wird, gilt $k_i \le x_j < k_{i+1}$ (Schlüssel liegen im Intervall $[k_i, k_{i+1}[$).

Beispiel



Interne Knoten

Externe Knoten

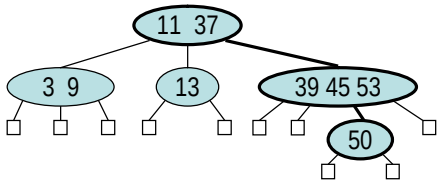
Suchen im Mehrwegbaum

Ähnlich zur Suche im binären Suchbaum

Bei jedem internen Knoten mit Schlüsseln k_1, k_2, …, k_m und Verbindungen p_0, p_1, … , p_m

- Suchschlüssel $s = k_i$ ($i = 1, …, m$): Suche erfolgreich
- $S \le k_1$: fortsetzen im Unterbaum p_0
- $k_i \le s < k_{i+1}$: fortsetzen im Unterbaum p_i
- $s > k_m$: fortsetzen im Unterbaum p_m
- Falls man einen externen Knoten erreicht: Suche erfolglos

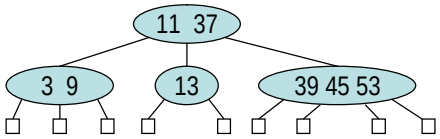
Beispiel: Suche 50



2-3-4 Bäume

Ein 2-3-4 Baum ist ein Mehrwegbaum mit den folgenden Eigenschaften

- Größeneigenschaft: jeder Knoten hat mindestens 2 und maximal 4 Kinder
- Tiefeneigenschaft: alle externe Knoten besitzen dieselbe Tiefe
- Abhängig von der Anzahl der Kinder heißt ein interner Knoten 2-Knoten, 3-Knoten oder 4-Knoten



Höhe eines 2-3-4 Baumes

Satz: Ein 2-3-4 Baum, der n interne Knoten speichert, hat eine Höhe von O(log n)

Beweis

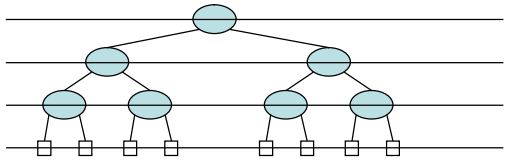
Die Höhe des 2-3-4 Baumes mit n internen Knoten sei h

Da es mindestens 2^i interne Knoten auf den Tiefen $i = 0, … , h-1$ gibt und keine internen Knoten auf Tiefe h, gilt

$$n \ge 1 + 2 + 4 + … + 2^{h-1} = 2^h - 1$$

Daher gilt $h \le \log_2 (n + 1)$

$n \ge 2^h - 1$
$n + 1 \ge 2^h$
$\log (n + 1) \ge h \log 2$
$h \le \log (n + 1) / \log 2 = \log_2 (n + 1)$

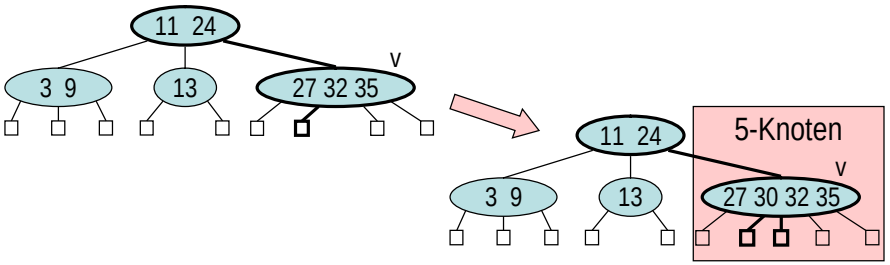


Einfügen in einen 2-3-4 Baum

Ein neuer Schlüssel s wird im Elternknoten v des externen Knotens eingefügt, den man bei der Suche nach s erreicht hat

- Die Tiefeneigenschaft des Baumes wird erhalten, aber es wird möglicherweise die Größeneigenschaft verletzt
- Ein (Knoten-) Überlauf ist möglich (es entsteht ein 5-Knoten)

Beispiel: Einfügen von 30 erzeugt Überlauf

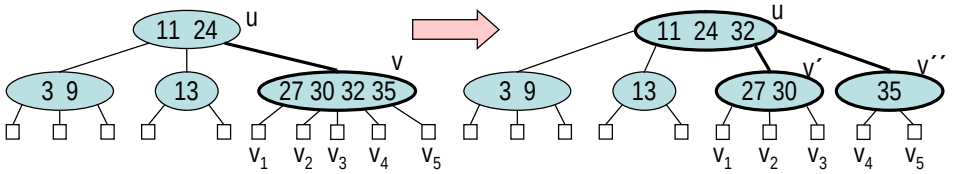


Überlauf und Split

Ein Überlauf (Overflow) bei einem 5-Knoten v wird durch eine Split Operation aufgelöst

- Die Kinder von v seien $v_1, \ldots, v_5$ und die Schlüssel von v seien $k_1, \ldots, k_4$
- Knoten v wird durch die Knoten v´ und v´´ ersetzt, wobei
  - v´ ein 3-Knoten mit den Schlüsseln k_1 und k_2 und Kindern v_1, v_2 und v_3,
  - v´´ ein 2-Knoten mit Schlüssel k_4 und Kindern v_4 und v_5 ist
- Schlüssel k_3 wird in den Elternknoten u von v eingefügt (dadurch kann eine neue Wurzel entstehen)

Ein Überlauf kann an die Vorgänger von u propagiert werden



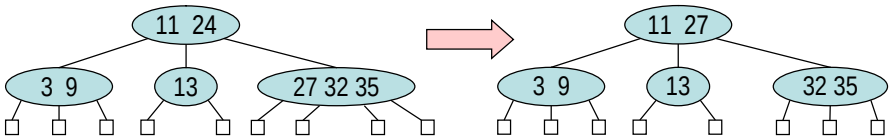
Löschen

Der Löschvorgang wird auf den Fall reduziert, wo der zu löschende Schlüsselwert in einem internen Knoten mit externen Knotenkindern liegt

Andernfalls wird der Schlüsselwert mit seinem Inorder Nachfolger (oder Inorder Vorgänger) ersetzt

- Analog Fall 3 binäre Suchbäume

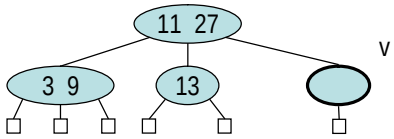
Beispiel Löschen Schlüssel 24



Unterlauf

Durch das Löschen eines Schlüssels in einem Knoten v kann es zu einem Unterlauf (underflow) kommen

- Knoten v degeneriert zu einem 1-Knoten mit einem Kind und keinem Schlüssel



Bei einem Unterlauf kann man 2 Fälle unterscheiden

- Fall 1: Verschmelzen
  - Benachbarte Knoten werden zu einem erlaubten Knoten verschmolzen
- Fall 2: Verschieben
  - Schlüsselwerte und entsprechende Kinder werden zwischen Knoten verschoben

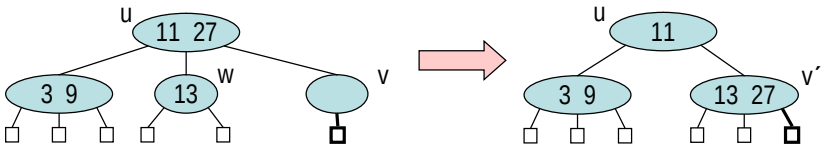
Verschmelzen im 2-3-4 Baum

Fall 1: Verschmelzen von Knoten

Bedingung: Alle adjazenten Knoten (benachbarte Knoten auf derselben Tiefe) zum unterlaufenden Knoten v sind 2-Knoten

Man verschmilzt v mit einem/dem adjazenten Nachbarn w und verschiebt den nicht mehr benötigten Schlüssel vom Elternknoten u zu dem verschmolzenen Knoten v´

Das Verschmelzen kann den Unterlauf zum Elternknoten propagieren



Verschieben im 2-3-4 Baum

Fall 2: Verschieben von Schlüsseln

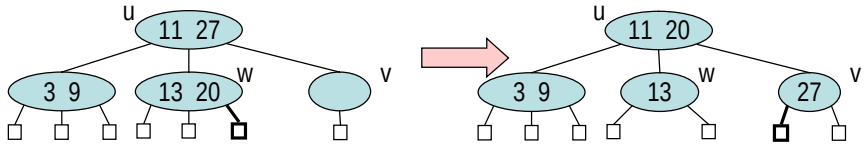
Bedingung: Ein adjazenter Knoten (benachbarter Knoten auf derselben Tiefe) w zum unterlaufenden Knoten v ist ein 3-Knoten oder 4-Knoten

Man verschiebt ein Kind von w nach v

Man verschiebt einen Schlüssel von u nach v

Man verschiebt einen Schlüssel von w nach u

Nach dem Verschieben ist der Unterlauf behoben



Analyse 2-3-4 Baum

Datenverwaltung

Einfügen und Löschen unterstützt

Datenmenge

unbeschränkt

Modelle

Hauptspeicherorientiert

Unterstützung komplexer Operationen

Bereichsabfragen, Sortierreihenfolge

Laufzeit

Speicherplatz	O(n)
Verschmelzen, Verschieben	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Vergleich der Baumvarianten

Baumvariante	*Baumhöhe + Aufwand der Operationen*	*Balanzierungs-form*	*Methode*
Allgemeiner Binärer Suchbaum	Im Durchschnitt log(n)	Abhängig von der Eingabe	Zufall, Gesetz der großen Zahlen
AVL-Baum	log(n)	höhenbalanziert	Rotationen
2-3-4 Baum	log(n)	perfektbalanziert	Split, Verschmelzen, Verschieben

5.6 Externspeicherverwaltung

Ziel

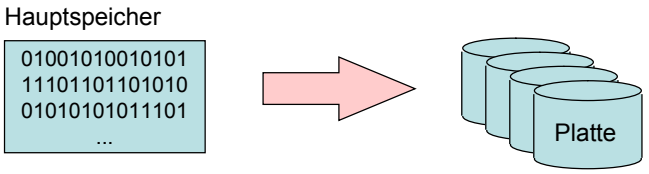
Erstellen eines Schlüsselbaumes für eine große Anzahl von Elementen

Problem

Hauptspeicher zu klein

Lösung

Speicherung der Knoten auf dem Externspeicher (Platte)



Plattenzugriffe

Ansatz

Referenzieren eines Knotens entspricht Zugriff auf die Platte

Bei Aufbau eines binären Schlüsselbaumes für eine eine Datenmenge von z.B. einer Million Elementen ist die durchschnittliche Baumhöhe $\log_2 10^6 \approx 20$, d.h. 20 Suchschritte $\rightarrow$ 20 Plattenzugriffe

Dilemma

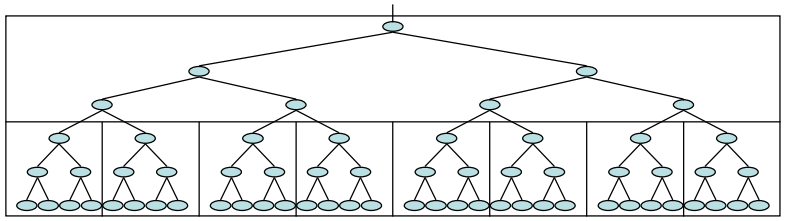
Unterschied Aufwand Platten- zu Hauptzugriff Faktor 100000, milli- zu nano-Sekunden (z.B. 5ms – 60 ns)

Jeder Suchschritt benötigt einen Plattenzugriff $\Rightarrow$ hoher Aufwand an Rechenzeit

Seitenverwaltung

Lösung

Zerlegung des binären Baumes in Teilbäume, diese in sog. *Seiten (pages)* speichern



$\Rightarrow$ Mehrwegbaum (multiway-tree)

Bei 100 Knoten pro Seite für 1 Million Elemente nur mehr $\log_{100} 10^6 = 3$ Seitenzugriffe

Im schlimmsten Fall (lineare Entartung) aber immer noch 10^4 Zugriffe (10^6 / 100)

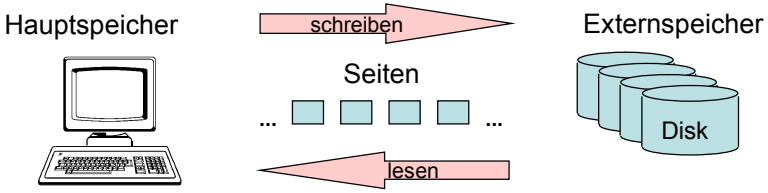
Seitenverwaltung (2)

Knoten entsprechen Seiten (Blöcke)

besitzen eine Größe, Seitengröße (Anzahl der enthaltenen Einträge bzw. Speichergröße in Bytes)

repräsentieren die Transfereinheit zwischen Haupt- und Externspeicher

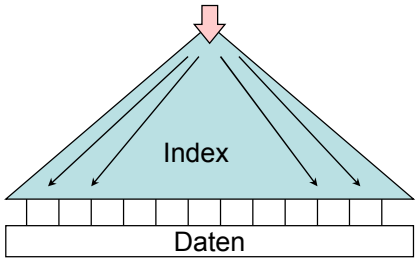
Zur Effizienzsteigerung des Transfers wird die Größe der Seiten an die Blockgröße der Speichertransfereiheiten des Betriebssystems angepaßt (Unit of Paging/Swapping)



Indexstruktur

Indexstruktur besteht aus Schlüsselteil (Index) und Datenteil

Graphische Struktur



Index

ermöglicht den effizienten Zugriff auf die Daten

Daten

enthalten die gespeicherte Information (vergleiche externe Knoten)

Knotenarten

2 Knotenarten

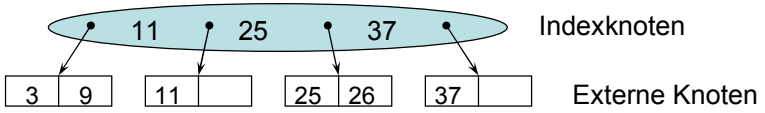
Indexknoten

- Erlauben effizienten Zugriff auf die externe Knoten
- Definieren die Intervallbereiche der Elemente, die im zugeordneten Teilbaum gespeichert sind.
- realisiert durch interne Knoten

Externe Knoten

- Enthalten die zu verwaltende Information
- realisiert durch Blattknoten

Beispiel



5.7 B+-Baum

Höhenbalanzierter (perfektbalanzierter) Mehrwegbaum

Eigenschaften

- Externspeicher-Datenstruktur
- Eine der häufigsten Datenstrukturen in Datenbanksystemen
- Dynamische Datenstruktur (Einfügen und Löschen)
- Algorithmen für Einfügen und Löschen erhalten die Balanzierungseigenschaft
- Garantiert einen begrenzten (worst-case) Aufwand für Zugriff, Einfügen und Löschen
- Der Aufwand für die Operationen Zugriff, Einfügen und Löschen ist bedingt durch die Baumstruktur maximal von der Ordnung log n (O(log n)).
- Besteht aus Index und Daten
- Der Weg zu allen Daten ist gleich lang

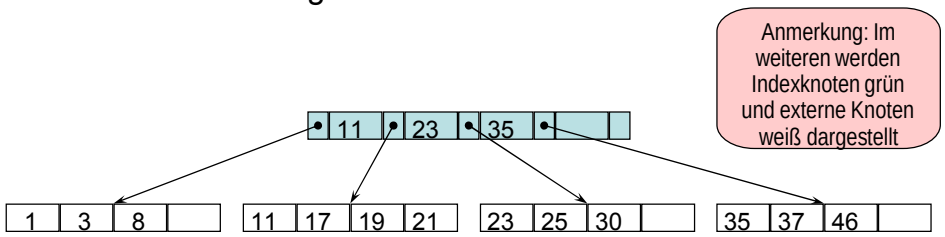
Definition B+-Baum

B+-Baum der Ordnung k

- alle Blattknoten haben die gleiche Tiefe (gleiche Weglänge zur Wurzel)
- die Wurzel ist entweder ein Blatt oder hat mindestens 2 Kinder
- jeder (interne) Knoten besitzt mindestens k und maximal 2k Schlüsselwerte, daher mindestens k+1 und maximal 2k+1 Kinder
  - Für jeden in einem Knoten referenzierten Unterbaum gilt, dass die in ihm gespeicherten Elemente kleiner als die Elemente im rechten und größer als die Elemente im linken Unterbaum sind. Die Intervallgrenzen werden durch die Schlüsselwerte bestimmt.

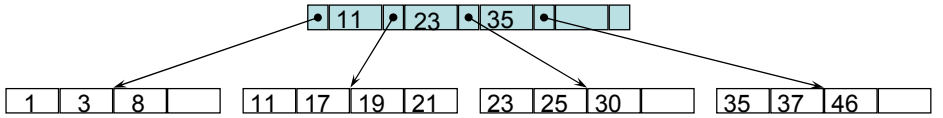
Beispiel B+-Baum

B+-Baum der Ordnung 2



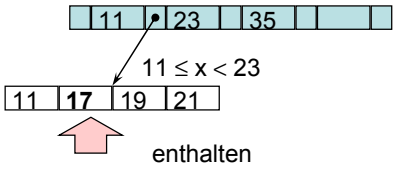
- Jeder Indexknoten hat mindestens 2 und maximal 4 Grenzwerte und folglich mindestens 3 und maximal 5 Kinder
- Ausnahme ist die Wurzel
- Die Größe der externen Knoten ist eigentlich durch die Ordnung nicht definiert, wird aber üblicherweise in der Literatur gleichgesetzt

Suche

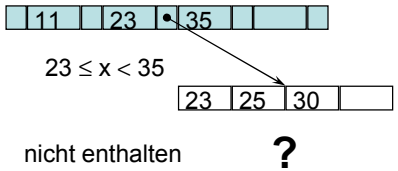


Suchen

Element 17



Element 27



Aufwand der Suche

Höhe eine B⁺-Baumes der Ordnung k mit *Datenblockgröße* b (mind. b, max 2b Elemente) ist maximal $\log_{k+1}(n/b)$.

Warum?

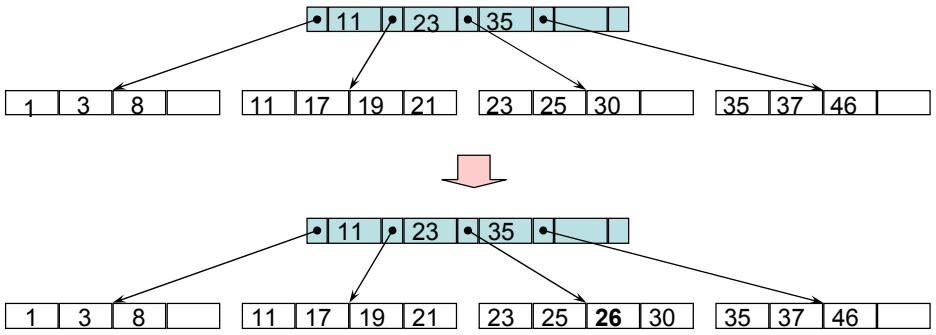
Bereichsabfrage möglich

Suche alle Element im Bereich [Untergrenze, Obergrenze]

B⁺-Baum Einfügen (1)

Fall 1: Einfügen Element 26

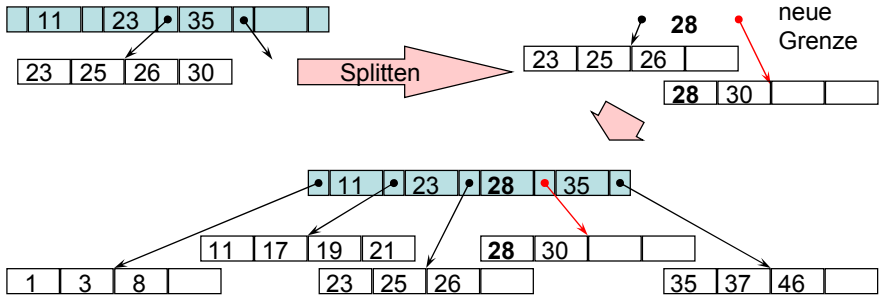
Platz im externen Knoten vorhanden, einfaches Einfügen



B⁺-Baum Einfügen (2)

Fall 2: Einfügen Element 28

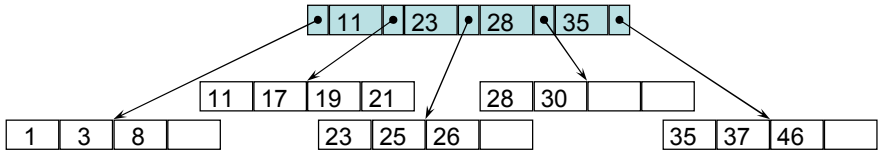
Kein Platz mehr im externen Knoten (*Überlauf, Overflow*), externer Knoten muss geteilt werden ⇒ Split



B+-Baum Einfügen (3)

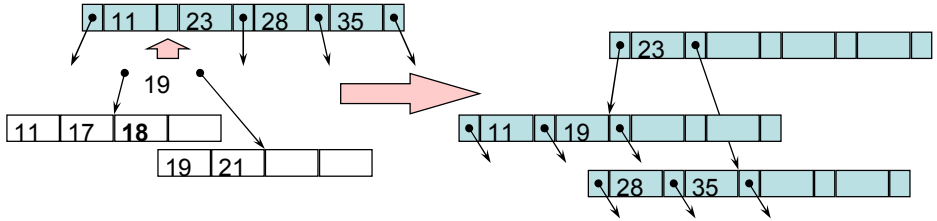
Fall 3: Einfügen Element 18

Kein Platz mehr im Knoten und kein Platz mehr im darüberliegenden Indexknoten, Indexknoten muss gesplittet werden



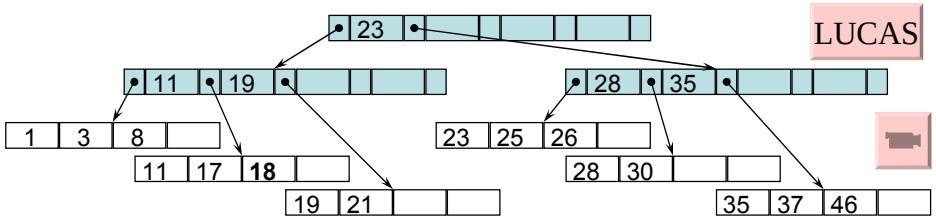
B+-Baum Einfügen (4)

Beim Splitten eines Indexknoten wird das mittlere Indexelement im darüberliegenden Indexknoten eingetragen. Falls der nicht existiert (*Wurzelsplit*), wird eine neue Wurzel erzeugt



B+-Baum Einfügen (5)

Der resultierende Baum hat folgendes Aussehen



Unterscheidung des Splits für externe und interne Knoten

Externe Knoten: 2k+1 Elemente werden auf 2 benachbarte externe Knoten aufgeteilt

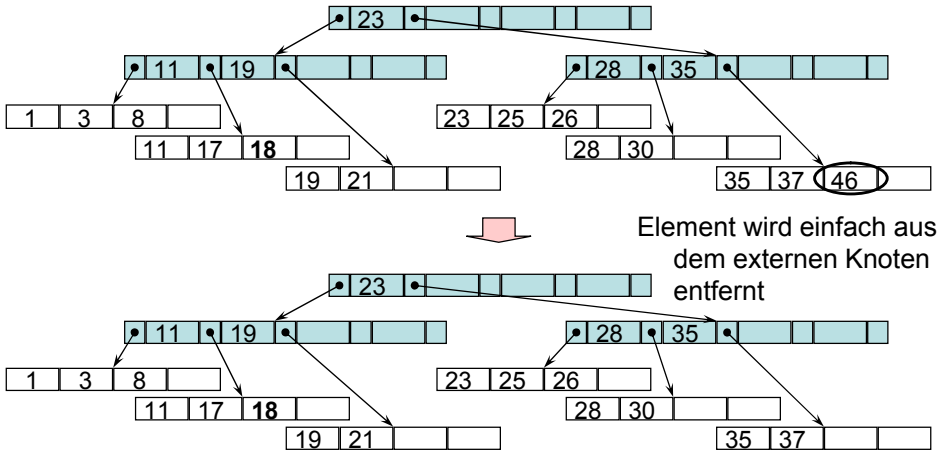
Interne Knoten: 2k+1 Indexelemente (Schlüsselwerte) werden auf 2 benachbarte Indexknoten zu je k Elemente aufgeteilt, das mittlere Indexelement wird in die darüberliegende Indexebene eingetragen.

Falls keine darüberliegende Ebene existiert, wird eine neue Wurzel erzeugt, der Baum wächst um eine Ebene ("Baum wächst von den Blättern zur Wurzel"), der Zugriffsweg zu den Daten erhöht sich um 1.

B+-Baum Löschen (1)

Fall 1: Entfernen von Element 46

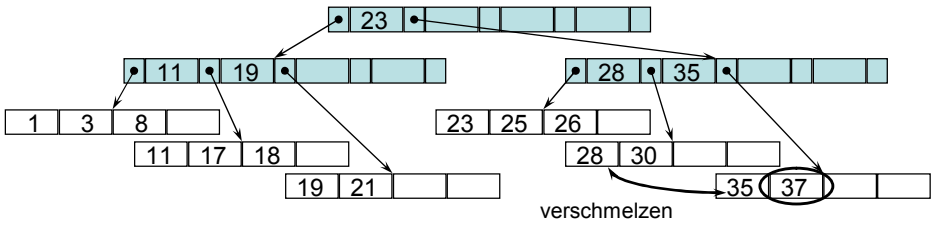
Externer Knoten nach Löschen nicht unterbesetzt



B+-Baum Löschen (2)

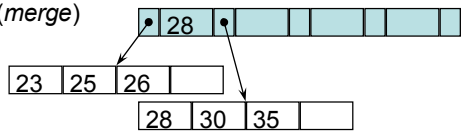
Fall 2: Entfernen von Element 37

Externer Knoten nach dem Löschen unterbesetzt



Element wird entfernt, ist Knoten unterbesetzt (*Underflow*), Elementanzahl < k

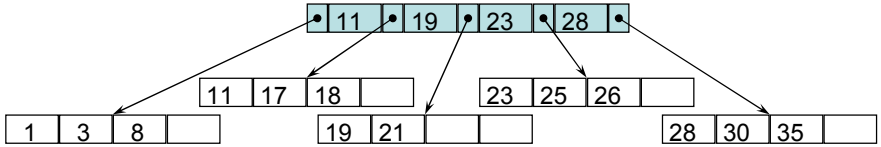
Umgekehrter Vorgang zum Split ⇒ benachbarte Knoten werden *verschmolzen* (*merge*)



Interner Knoten unterbesetzt, muss ebenfalls mit Nachbarn verschmolzen werden.

B+-Baum Löschen (3)

Das Verschmelzen kann zur Verringerung der Baumhöhe führen, 2 Knoten werden zur neuen Wurzel verschmolzen, alte Wurzel wird entfernt.

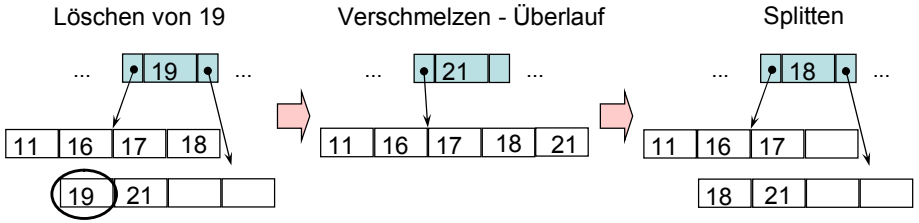


B+-Baum Löschen (4)

Anmerkung

Das Verschmelzen zweier Knoten kann zu einem Überlauf (analog Einfügen) des neuen Knoten führen, was einen nachfolgenden Split notwendig macht.

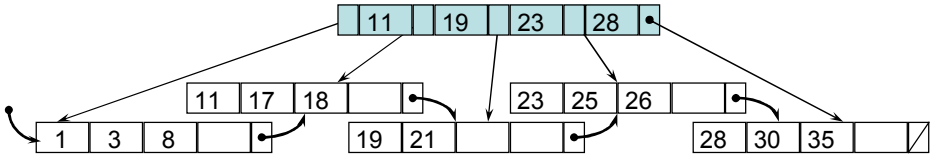
Beispiel: (Baumausschnitt)



Diese (Verschmelzen-Splitten) Sequenz erzeugt eine besseren Aufteilung der Datensätze zwischen benachbarten Knoten. Dies wird auch hier (siehe 2-3-4 Baum, eigentlich fälschlicherweise) als Datensatzverschiebung (shift) bezeichnet.

Datenblockverkettung

In der Praxis werden die Datenblöcke linear verkettet, um einen effizienten, sequentiellen Zugriff in der Sortierreihenfolge der gespeicherten Elemente zu ermöglichen



Analyse B+ - Baum

Datenverwaltung
- Einfügen und Löschen unterstützt

Datenmenge
- unbeschränkt
  - abhängig von der Größe des vorhandenen Speicherplatzes

Modelle
- Externspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Analyse B+ - Baum (2)

Speicherplatz	O(n)
Split, Verschmelzen	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Bitte beachten: Ordnung des Baumes ist konstanter Faktor!

5.8 Trie

Ein Trie ist ein digitaler Suchbaum

Dient zur Speicherung von Strings (Verwaltung von Wörterbüchern, Telefonbüchern, Spellcheckern, etc.)
- Bezeichnung wird abgeleitet von “retrieval”, wird aber wie “try” ausgesprochen

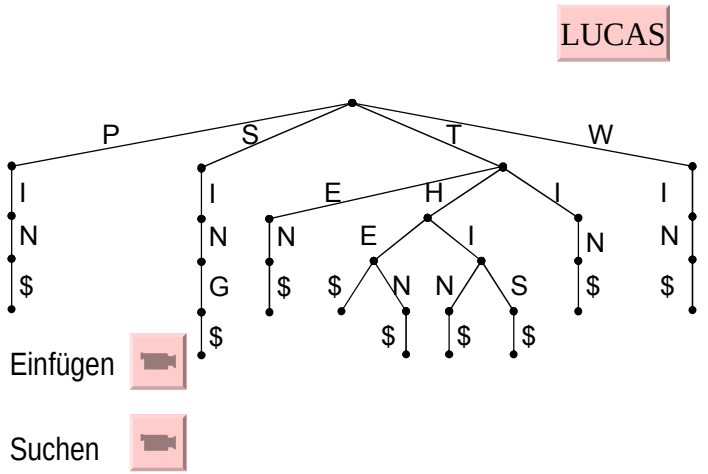
Bei k Buchstaben ein “k+1-ary Tree”, wobei jeder Knoten durch eine Tabelle mit k+1 Kanten auf Kinder repräsentiert wird.

Spezialformen
- Patricia Tree, de la Briandais Tree

Beispiel Trie

Trie

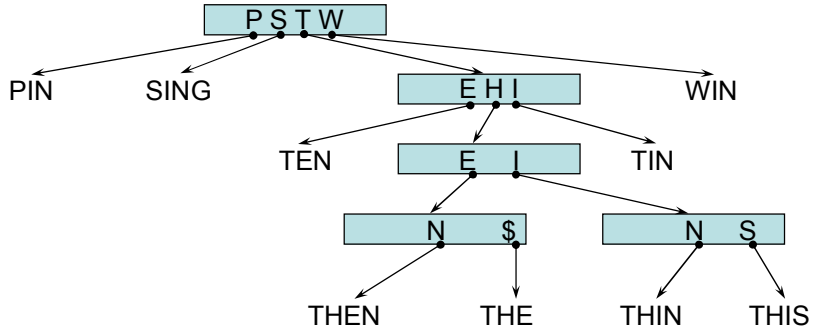
TEN
THE
THEN
THIN
THIS
TIN
SING
PIN
WIN



Patricia Trie

*P*ractical *A*lgorithm *t*o *R*etrieve *I*nformation *C*oded *i*n *A*lphanumeric

Ziel ist die Komprimierung des Tries



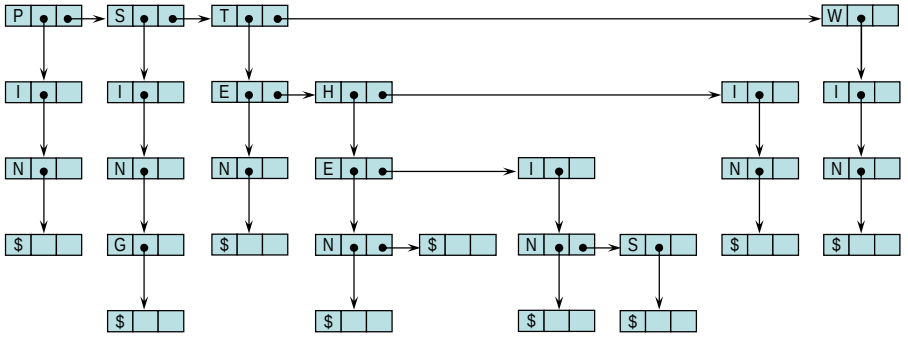
de la Briandais Trie

Listen-Repräsentation eines Tries

statt der Tabellen im Knoten lineare Liste

Knoten besteht aus 2 Kinderkomponenten

Nächster Wert - Nächster Level



Analyse Trie

Datenverwaltung

- Einfügen und Löschen unterstützt
- Struktur des Tries unabhängig von der Einfügereihenfolge

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

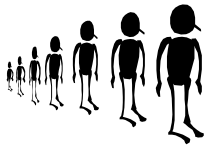
Laufzeit

Speicherplatz	O(n)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Bei der exakten Berechnung ist die Basis des Logarithmus abh. von der Kardinalität der Zeichenmenge, z.B. 26 (Grossbuchstaben), 2 (Bitstrings)

5.9 Priority Queues

Datenstruktur zum wiederholten Finden und Entfernen des Elementes mit dem kleinsten (bzw. größten) Schlüsselwert aus einer Menge



Anwendungen

- Simulationssysteme (Schlüsselwerte repräsentieren Exekutionszeiten von Ereignissen)
- Prozessverwaltung (Prioritäten der Prozesse)
- Numerische Berechnungen (Schlüsselwerte entsprechen den Rechenfehlern, wobei zuerst die großen beseitigt werden)
- Grundlage für eine Reihe komplexer Algorithmen (Graphentheorie, Filekompression, etc.)
  - Anmerkung: Im folgenden betrachten wir o.B.d.A. nur Priority Queues die den Zugriff auf die kleinsten Elemente unterstützen

Operationen

Construct
- Erzeugen einer leeren Priority Queue

IsEmpty
- Abfrage auf leere Priority Queue

Insert
- Einfügen eines Elementes

FindMinimum
- Zurückgeben des kleinsten Elementes

DeleteMinimum
- Löschen des kleinsten Elementes

Implementationsansätze

Ungeordnete Liste
- Elemente werden beliebig in die Liste eingetragen (Aufwand O(1)).
- Beim Zugriff bzw. Löschen des 'kleinsten' Elementes muss die Liste abgesucht werden Aufwand → O(n).

Geordnete Liste
- Elemente werden in der Reihenfolge ihrer Größe eingetragen (Aufwand O(n)).
- Zugriff bzw. Löschen konstanter Aufwand → O(1)

Problem
- Aufwand O(n) schwer zu vermeiden.

Priority Queue als Baumstruktur

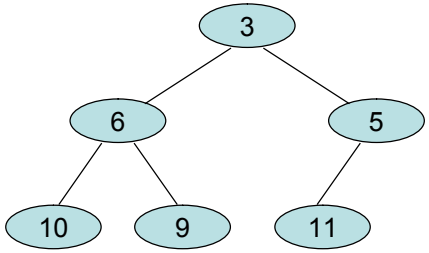
Balanzierte Schlüsselbäume
- Einfügen im balanzierten Schlüsselbaum vom Aufwand O(log n)
- Zugriff realisieren durch Verfolgen des äußersten linken Weges im Baum (zum kleinsten Element) → Aufwand O(log n)

Günstigste Realisierung über ungeordnete, komplette Schlüsselbäume
- mit der Eigenschaft, dass für alle Knoten die Schlüsselwerte ihrer Nachfolger größer (oder kleiner) sind

5.9.1 Heap

Wertemenge

3 6 5 10 9 11



Heap
- Ungeordneter, binärer, kompletter Schlüsselbaum
- Wert jedes Knotens ist kleiner (größer) oder gleich den Werten seiner Nachfolgerknoten
- Wurzel enthält kleinstes (größtes) Element
- Anordnung der Unterbäume bez. Ihrer Wurzel undefiniert (ungeordneter Baum)

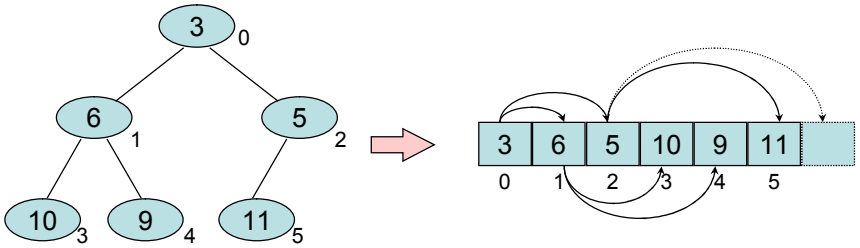
Heap Datenstruktur

Realisierung eines Heaps effizient durch ein Feld

Die n Schlüsselwerte des Heaps können als Folge von Elementen x_0, x_1, x_2, … x_{n-1} interpretiert werden, wobei die Position der einzelnen Knotenwerte im Feld durch folgende Regel bestimmt wird:

Wurzel in Position 0, die Kinder des Knotens i werden an Position 2i+1 und 2i+2 gespeichert.

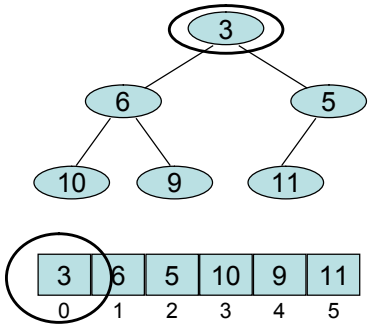
Beispiel



Zugriff

Zugriff auf das kleinste Element mit konstanten Aufwand: O(1)

→ Wurzel = erstes Element im Feld.



Klasse Priority Queue

```
class PQ {
  int *a, N;
public:
  PQ(int max) { a = new int[max]; N = -1; }
  ~PQ() { delete[] a; }
  int FindMinimum(){ return a[0]; }
  int IsEmpty() { return (N == -1); }
  void Insert(int);
  int DeleteMinimum();
  …
};
```


Einfügen in einen Heap

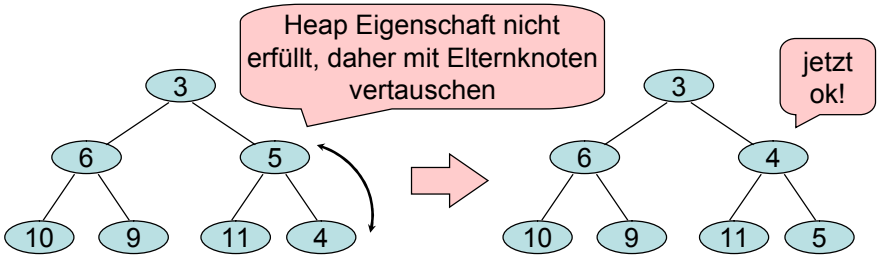
Neues Element an der letzten Stelle im Feld eintragen

Überprüfen, ob die Heap Eigenschaft erfüllt ist

Wenn nicht, mit Elternknoten vertauschen und solange wiederholen bis erfüllt

Eintragen von 4

(an der letzten Stelle im Feld) Aufwand: O(log n)

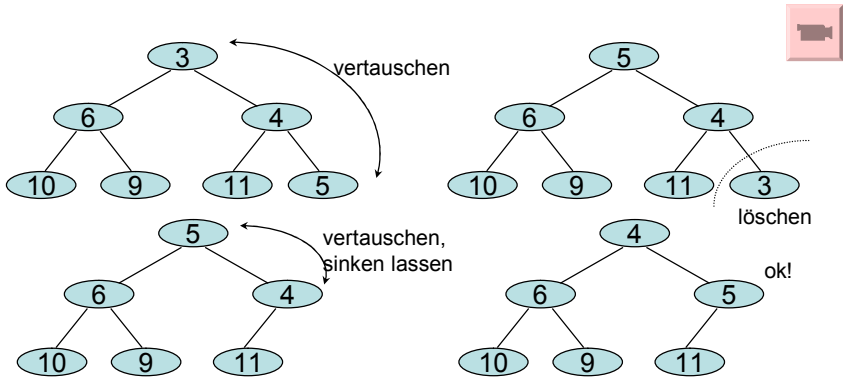


Methode Insert

```
void PQ::Insert(int v) {
  int child, parent;
  a[++N] = v;
  child = N;
  parent = (child - 1)/2;
  while(child != parent) {
    if(a[parent] > a[child]) {
      swap(a[parent], a[child]);
      child = parent;
      parent = (parent - 1)/2;
    } else break; // to stop the loop
  }
}
```


Löschen aus einen Heap

Wurzel mit äußerst rechten Blattknoten tauschen, diesen Blattknoten löschen, Wurzel in den Baum sinken lassen (mit kleinerem Kindknoten vertauschen), bis Heap Eigenschaft gilt. Aufwand: O(log n)



Methode DeleteMinimum

```
int PQ::DeleteMinimum() {
  int parent = 0, child = 1;
  int v = a[0];
  a[0] = a[N];
  N--;
  while(child <= N) {
    if(a[child] > a[child+1]) child++;
    if(a[child] < a[parent]) {
      swap(a[parent], a[child]);
      parent = child;
      child = 2*child + 1;
    } else break; // to stop the loop
  }
  return v;
}
```


Analyse Heap

Datenverwaltung

- Einfügen und Löschen unterstützt

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Nur simple Operationen

Laufzeit

Speicherplatz	O(n)
Zugriff (Minimum/Maximum)	O(1)
Einfügen	O(log n)
Löschen	O(log n)

Der Heap stellt eine effiziente Basisdatenstruktur für viele weitere darauf aufbauende Datenstrukturen dar

Heaps und Heaps

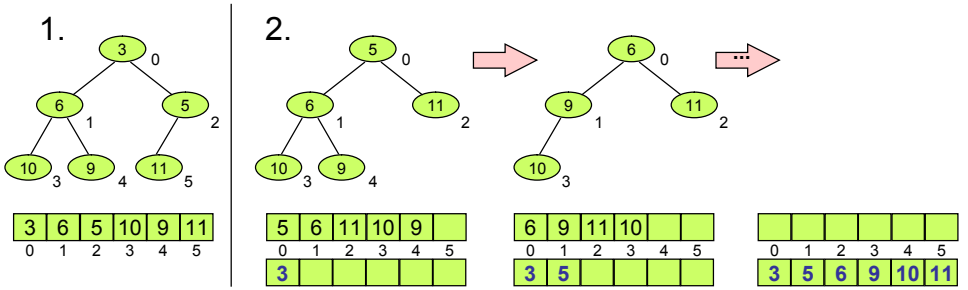
Bitte zu unterscheiden! Der Begriff Heap wird in der Informatik zur Bezeichnung verschiedener Konzepte verwendet

- Heap als Datenstruktur zur Speicherung von Priority Queues
- Heap als Speicherbereich zur Verwaltung (vom Benutzer selbst verwalteter) dynamisch allozierter Speicherbereiche
- Heap als Speicherform in Datenbanken

5.9.2 Heapsort

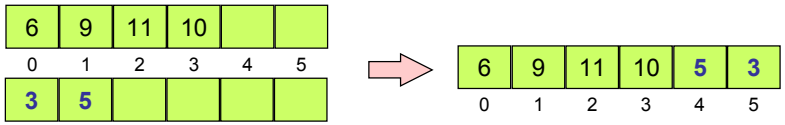
Eine Heap (Priority Queue) kann Basis zum Sortieren bilden (Williams 1964):

1. ein Element nach dem anderen in einen Heap einfügen
2. sukzessive das kleinste bzw. größte Element entfernt
   die Element werden in aufsteigender bzw. absteigender Ordnung geliefert

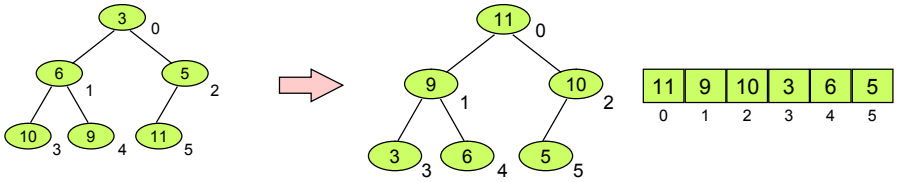


Heapsort (modifizierter Ansatz)

Kombination der beiden Arrays, Vermeidung von doppeltem Speicherplatz



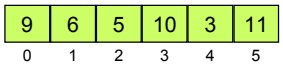
Heap-Eigenschaft umdrehen: Wert jedes Knotens ist größer oder gleich den Werten seiner Kinderknoten

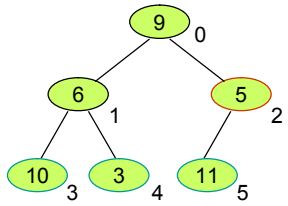


Beispiel: buildheap

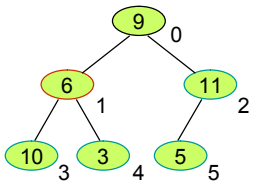
Wertemenge

9 6 5 10 3 11

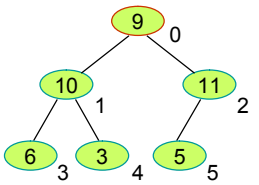




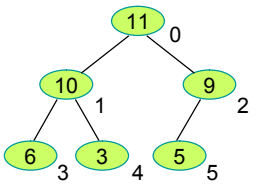
Idee: Heap-Eigenschaft erzeugen; für Blattknoten trivialerweise erfüllt
Nr. 2 erster zu prüfender Knoten, 11 passt nicht, in Baum sinken lassen, d.h. 5 mit 11 tauschen



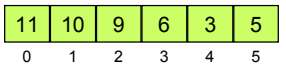
Wert 6 (Nr. 1) passt nicht, in Teilbaum sinken lassen, d.h. mit größerem der Nachfolger vertauschen, d.h. 6 mit 10



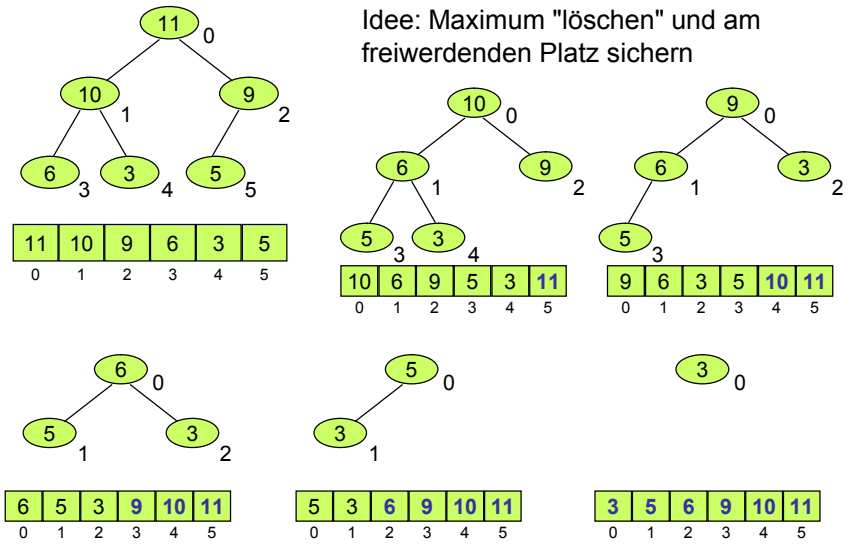
Nr. 0 (9) in Baum sinken.



Heap-Eigenschaft erfüllt!



Beispiel: heapsort



Heapsort Algorithmus (1)

```
void Vector::Heapsort() {
  int heapsize = Length();
  BuildMaxheap();
  for(int i = Length()-1; i >= 1; i--) {
    swap(a[0], a[i]);
    heapsize--;
    heapify(0, heapsize);
  }
}

void Vector::BuildMaxheap() {
  for(int i = Length()/2 - 1; i >= 0; i--)
    heapify(i, Length());
}
```


Heapsort Algorithmus (2)

```
void Vector::heapify(int i, int heapsize) {
  int left = 2*i + 1;
  int right = 2*i + 2;
  int largest;
  if (left < heapsize && a[left] > a[i])
    largest = left;
  else
    largest = i;
  if (right < heapsize && a[right] > a[largest])
    largest = right;
  if (largest != i) {
    swap(a[i], a[largest]);
    heapify(largest, heapsize);
  }
}
```


Eigenschaften der komplexen Verfahren

Quicksort

- Entartung zu $O(n^2)$ möglich
  - Wahl des Pivotelements!

Mergesort

- immer Laufzeit O(n * log(n))
- doppelter Speicherplatz notwendig

Heapsort

- keinen der obigen Nachteile
- aber höherer konstanter Aufwand

Was nehmen wir mit?

Baumstrukturen
- Notation
- spez. Eigenschaften, von O(n) auf O(log n)

Suchbäume
- Binäre Suchbäume
- AVL-Bäume
- 2-3-4 Bäume
- B^+-Baum
- Balanzierung

Trie

Priority Queues
- Heap

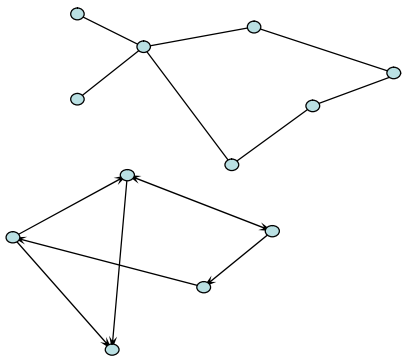
Kapitel 6
Graphen

Graphen

Graphen sind eine dominierende Datenstruktur in der Informatik
- Viele Probleme der Informatik lassen sich durch Graphen beschreiben und über Graphenalgorithmen lösen

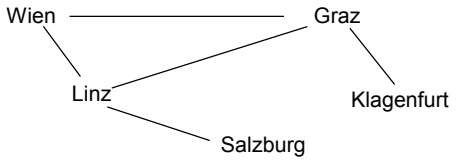
Ungerichteter Graph

Gerichteter Graph

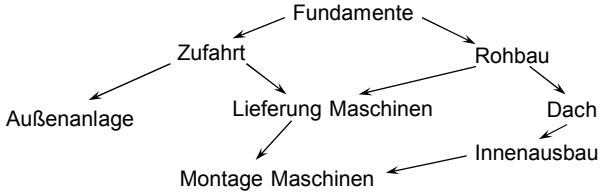


Graphen-Beispiele

Ortsverbindungen
- z.B. Züge



Ablaufbeschreibungen
- z.B. Projektplanung Maschinenhalle
  - Kanten rep. Abhängigkeiten



6.1 Ungerichteter Graph

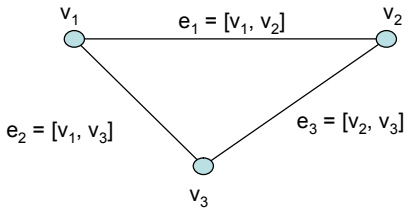
Ein *ungerichteter Graph* G(V, E) besteht aus einer Menge V (vertex) von *Knoten* und einer Menge E (edge) von *Kanten*, d.h.

$V = \{v_1, v_2, \ldots, v_{|V|}\}$

$E = \{e_1, e_2, \ldots e_{|E|}\}$

Eine *Kante* e ist eine ungeordnetes Paar von Knoten aus V, d.h. $e = [v_i, v_j]$ mit $v_i, v_j \in V$

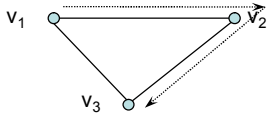
G(V, E)
$V = \{v_1, v_2, v_3\}$
$E = \{e_1, e_2, e_3\}$



Kantenfolge, -zug, Weg

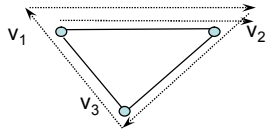
Eine *Kantenfolge* von v_1 nach v_n in einem Graphen G ist eine endliche Folge von Kanten $[v_1,v_2]$, $[v_2,v_3]$, …, $[v_{n-1},v_n]$, wobei je 2 aufeinanderfolgende Kanten einen gemeinsamen Endpunkt haben

$KF_1 = [v_1,v_2], [v_2,v_3]$



oder

$KF_2 = [v_1,v_2], [v_2,v_3], [v_3,v_1], [v_1,v_2]$



Eine *Kantenzug* ist eine Kantenfolge, in der alle Kanten verschieden sind, (im Beispiel ist KF_1 ein Kantenzug, KF_2 nicht.)

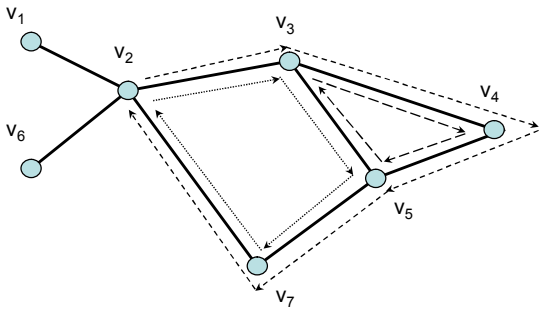
Ein *Weg* oder *Pfad* ist eine Kantenfolge in der alle Knoten verschieden sind (ein einzelner Knoten gilt auch als Weg)

Kreis

Ein *Kreis* ist ein Kantenzug, bei dem die Knoten $v_1, v_2, \ldots, v_{n-1}$ alle verschieden sind und $v_n = v_1$ gilt

Kreise:

v_2, v_3, v_5, v_7, v_2
v_3, v_4, v_5, v_3
$v_2, v_3, v_4, v_5, v_7, v_2$



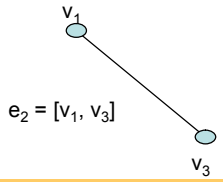
Ein Graph heißt *verbunden oder zusammenhängend*, wenn für alle möglichen Knotenpaare v_j, v_k ein Pfad existiert, der v_j mit v_k verbindet

Ein Baum ist daher ein verbundener kreisloser (azyklischer) Graph

Teilgraph

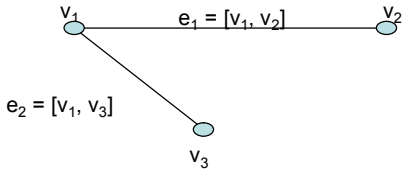
Ein Graph G'(V', E') heißt *Teilgraph* von G(V, E), wenn $V' \subseteq V$ und $E' \subseteq E$

G'(V', E')
$V' = \{v_1, v_3\}$
$E' = \{e_2\}$



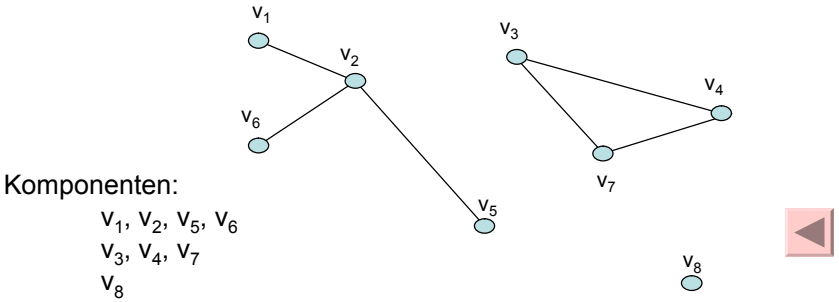
Ein Teilgraph G''(V'',E'') ist *spannender Teilgraph* von G(V,E), wenn V'' = V und G'' einen Baum bildet

G''(V'', E'')
$V'' = \{v_1, v_2, v_3\}$
$E'' = \{e_1, e_2\}$



Komponente

Alle Knoten, die durch einen Weg verbunden werden können, heißen *Komponente* eines Graphen



Komponenten:

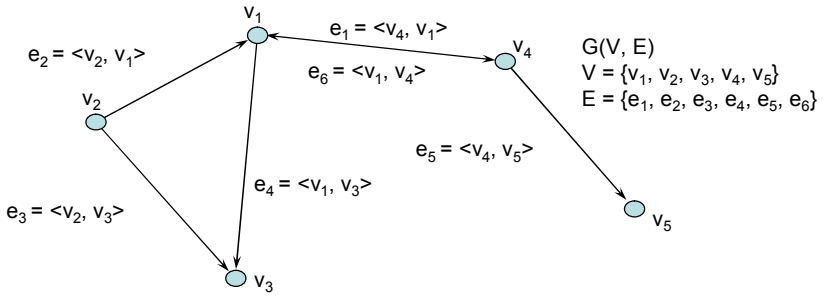
v_1, v_2, v_5, v_6

v_3, v_4, v_7

v_8

6.2 Gerichteter Graph

Ein *gerichteter Graph* G(V, E) besteht aus einer Menge V von Knoten und einer Menge E von Kanten, wobei die Kanten *geordnete* Paare $<v_i,v_j>$ von Knoten aus V sind



Zwischen v_1 und v_4 existieren 2 Kanten, eine Hin- und Rückkante (auch Doppelkante).

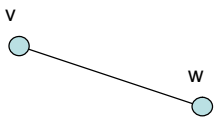
6.3 Speicherung von Graphen

Wir unterscheiden 2 Methoden zur Speicherung von Graphen

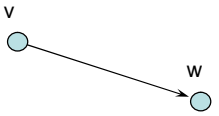
- Adjazenzmatrix-Darstellung
- Adjazenzlisten-Darstellung

Ein Knoten v heißt *adjazent* zu einem Knoten w, wenn eine Kante von v nach w führt, z.B.

ungerichtete Kante:
v adjazent zu w
w adjazent zu v



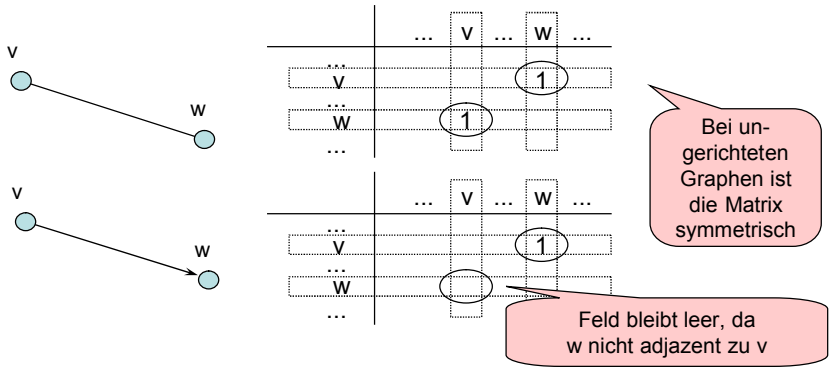
gerichtetet Kante:
v adjazent zu w
w aber NICHT adjazent zu v



6.3.1 Adjazenzmatrix

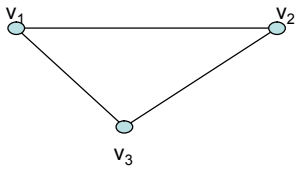
Bei der Adjazenzmatrix-Darstellung repräsentieren die Knoten Indexwerte einer 2-dimensionalen Matrix A

Wenn der Knoten v adjazent zum Knoten w ist, wird das Feld A[v,w] in der Matrix gesetzt (z.B. 1, 'true', Wert, etc.)



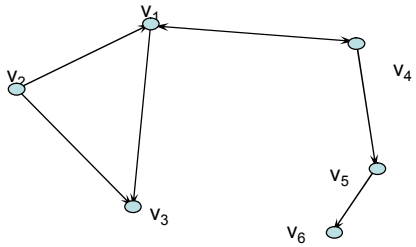
Adjazenzmatrix-Beispiele

Ungerichteter Graph



	v_1	v_2	v_3
v_1		1	1
v_2	1		1
v_3	1	1	

Gerichteter Graph



	v_1	v_2	v_3	v_4	v_5	v_6
v_1			1	1		
v_2	1		1			
v_3						
v_4	1				1	
v_5						1
v_6						

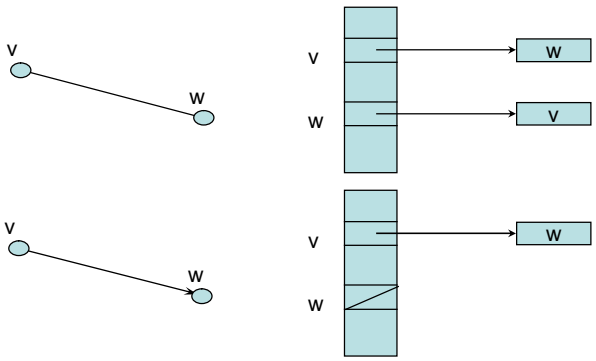
Eigenschaften

+ gut für kleine Graphen oder Graphen mit vielen Kanten
+ Überprüfung von Adjazenzeigenschaft O(1)
+ manche Algorithmen einfacher
- dfs schlecht, Rechenaufwand $O(|V|^2)$
- quadratischer Speicheraufwand $O(|V|^2)$

6.3.2 Adjazenzliste

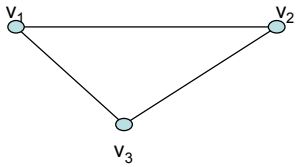
Bei der Adjazenzlistendarstellung werden für jeden Knoten alle adjazenten Knoten in einer linearen Liste gespeichert

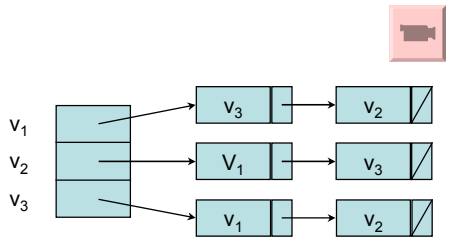
Somit werden nur die auftretenden Kanten vermerkt



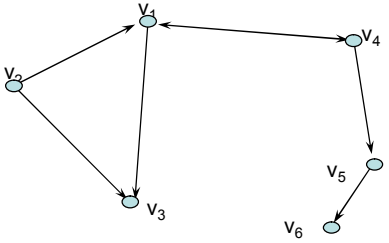
Adjazenzliste-Beispiele

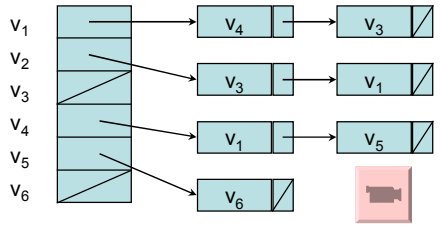
Ungerichteter Graph





Gerichteter Graph





Code, Eigenschaften

C-Code

```
struct node {
  int v;
  struct node *next;
};
struct node *adjliste[maxV];
```

Eigenschaften:

+ gut für Graphen mit wenigen Kanten
+ linearer Speicheraufwand O(n)
+ dfs gut, Rechenaufwand O(|V|+|E|)
- Überprüfung Adjazenzeigenschaft O(|V|)
- manche Algorithmen komplexer

6.4 Topologisches Sortieren

Problem: Ausgehend von einer binären Beziehung (z.B. „muß erledigt sein, bevor man weitermachen kann mit“) von Elementen ist zu klären, ob es eine Reihenfolge der Elemente gibt, ohne eine der Beziehungen zu verletzen.

Beispiel:

Professor Bumstead kleidet sich an

Kleidungsstücke sind: Unterhose, Socken, Schuhe, Hosen, Gürtel, Hemd, Krawatte, Sakko, Uhr

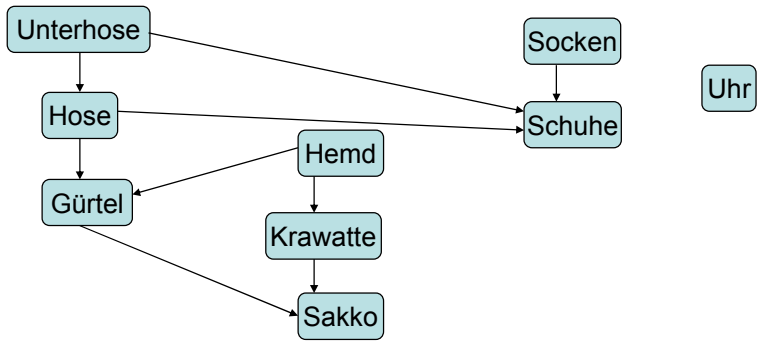
Beziehung: Kleidungsstück A muss vor B angezogen werden

Problem: Finde eine Ankleidereihenfolge damit sich Prof. Bumstead anziehen kann.

Interpretation durch gerichteten Graphen

Binäre Beziehung zwischen Elementen ist gerichtete Kante zwischen entsprechenden Knoten

Prof. Bumstead

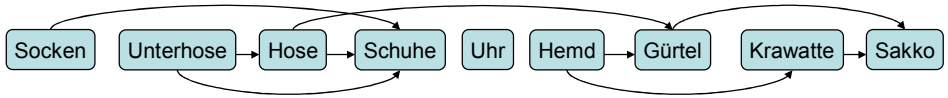


Topologische Ordnung eines DAG

Eine Topologische Ordnung eines gerichteten azyklischen Graphs G (directed acyclic graph, DAG) ist eine lineare Anordnung aller Knoten, sodass wenn G eine Kante <u, v> enthält, u vor v in der Anordnung steht

Falls der Graph nicht azyklisch ist, gibt es keine topologische Ordnung

Lösung Professor Bumstead



Eine mögliche Vorgangsweise

1. Suche einen Knoten aus dem nur Kanten hinausführen
2. Ordne ihn in der topologischen Ordnung an
3. Lösche ihn aus dem Graphen
4. Weiter bei 1.

Knotengrad

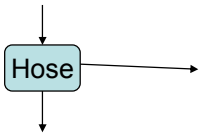
Eingangs- und Ausgangsgrad eines Knoten

indegree(v) mit v aus V: | { v' | <v', v> aus E} |

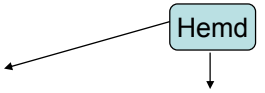
d.h. Anzahl der in v einmündenden Kanten

outdegree(v) mit v aus V: | { v' | <v, v'> aus E} |

d.h. Anzahl der von v ausgehenden Kanten



indegree: 1, outdegree: 2

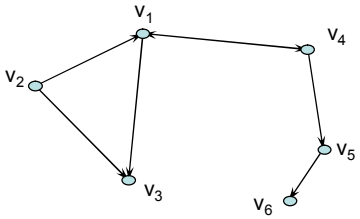


indegree: 0, outdegree: 2

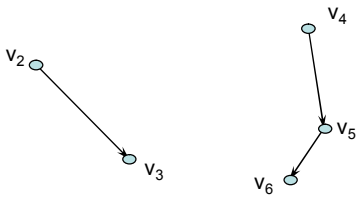
induzierter Teilgraph

Ein Graph G'(V', E') heißt *induzierter Teilgraph (Untergraph)* von G(V, E), wenn V' ⊆ V und E'=E∩{V'xV'}.

$V=\{v_1, v_2, v_3, v_4, v_5, v_6\}$



$V' = V \setminus \{v_1\} = \{v_2, v_3, v_4, v_5, v_6\}$



Algorithmus zum toplogischen Sortieren

```
lfdNr = 0;
while ($ v Î G: indegree(v) = 0) {
  lfdNr = lfdNr + 1;
  N[v] = lfdNr;
  G = induzierter Teilgraph von V\{v};
}
if (V = {})
  G ist azyklisch; // N enthält die topologische Ordnung für G
else
  G ist nicht azyklisch; // es existiert keine top. Ordnung
```

6.5 Traversieren eines Graphen

Unter dem Traversieren eines Graphen versteht man das systematische und vollständige Besuchen aller Knoten des Graphen

Es lassen sich prinzipiell 2 Ansätze unterscheiden:

- Tiefensuche, dfs
  depth-first search - Traversierung
- Breitensuche, bfs
  breadth-first search - Traversierung

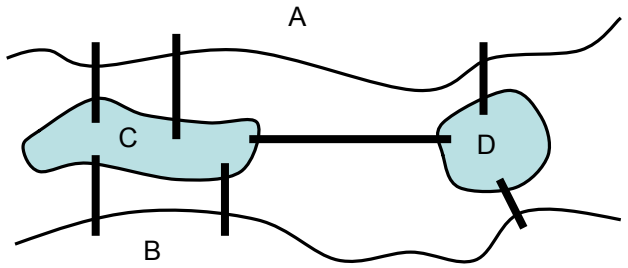
Über diese beiden Ansätzen lassen sich fast alle wichtigen Problemstellungen auf Graphen lösen, z.B.

- Suche einen Weg vom Knoten v nach w?
- Besitzt der Graph einen Zyklus?
- Finde alle Komponenten?
- ...

Das „Königsberger Brücken“ Problem

Leonhard Euler, 1736

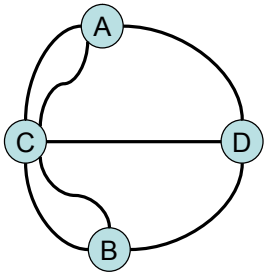
Die Stadt Königsberg (Kaliningrad) liegt an den Ufern und auf 2 Inseln des Flusses Prigel und ist durch 7 Brücken verbunden



Frage: Gibt es einen Weg auf dem ich die Stadt besuchen kann, alle Brücken genau einmal überquere und an den Anfangspunkt zurückkehre?

Abstraktion

Frage: Gibt es einen Kreis im Graphen der alle Kanten genau einmal enthält?



Euler löste das Problem, indem er bewies, dass so ein Kreis genau dann möglich ist, wenn der Graph zusammenhängend und der Knotengrad aller Knoten gerade ist

Solche Graphen werden Eulersche Graphen genannt.

Für Kaliningrad gilt dies offensichtlich nicht.

Eulersche Graphen werden als das erste gelöste Problem der Graphentheorie angesehen

Beweisskizze

Satz: Damit so ein Kreis existieren kann, müssen alle Knoten geraden Knotengrad haben

Knoten muss einmal betreten und einmal verlassen werden

Bedingung suffizient? Existiert immer ein Kreis?

Beweis durch Induktion

Hypothese: verbundener Graph G mit < m Kanten, wobei alle Knoten geraden Knotengrad besitzen, enthält einen Kreis, dessen Kanten genau einmal besucht werden

Ansatz: Entferne einen Kreis P (gerader Knotengrad, gerade Anzahl von Kanten) aus dem Graph, es bleibt ein Graph G‘ mit geraden Knotengrad zurück (weiteren Kreis entfernen usw.)

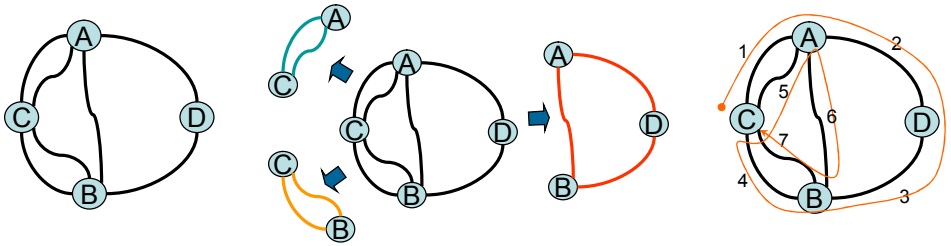
Problem: Graph könnte durch das Entfernen eines Kreises in Komponenten G'_1, G'_2, ... G'_k zerfallen, jede Komponente erfüllt aber wieder Bedingung gerader Knotengrad, d.h. Hypothese auf Komponenten ebenfalls anwendbar, ergibt k Kreise P_1, P_2, ... P_k

Kombination dieser Kreise zu Gesamtkreis, indem man bei beliebigem Knoten im Kreis P beginnt und Kreis solange bis zu einem Knoten v_j, verfolgt, der zu Komponente G'_j gehört. Von dort Verfolgung von entsprechenden Kreis P_j

Beweisskizze (2)

Graph erfüllt Bedingung, d.h. Kreis muss existieren

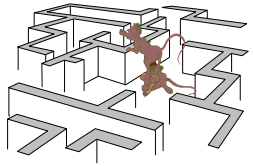
- Laut Beweis entferne einen Kreis nach dem anderen, bis alle Kanten entfernt sind, Kombination der Kreise liefert Lösung



- Aus der Regel für die Kombination der Kreise lässt sich ein entsprechender Algorithmus zum Finden eines Eulerschen Kreises ableiten
- führt zur Problemstellung des Traversieren eines Graphen

6.5.1 Depth-first Search

Idee



- Wir interpretieren den Graphen als Labyrinth, wobei die Kanten Wege und die Knoten Kreuzungen darstellen.
- Beim depth-first search Ansatz versuchen wir einen möglichst langen neuen Pfad zurückzulegen. Wenn wir keinen neuen Weg mehr finden, gehen wir zurück und versuchen den nächsten Pfad.
- Wenn wir zu einer Kreuzung kommen, markieren wir sie (im Labyrinth durch einen Stein). Wir merken uns mögliche Pfadalternativen.
- Kommen wir zu einer markierten Kreuzung über einen noch nicht beschrittenen Weg, gehen wir diesen Weg wieder zurück (wir waren schon mal da).

Ansatz

Algorithmus

- rekursiver Ansatz, zu besuchende Knoten werden am Stack (Rekursion) vermerkt.
- "Zuerst in die Tiefe , danach in die Breite gehen"

```
void besuche-dfs ( Knoten x ) {
  markiere Knoten x mit 'besucht';
  for ( alle zu x adjazente nicht besuchte Knoten v )
    besuche-dfs ( v );
```

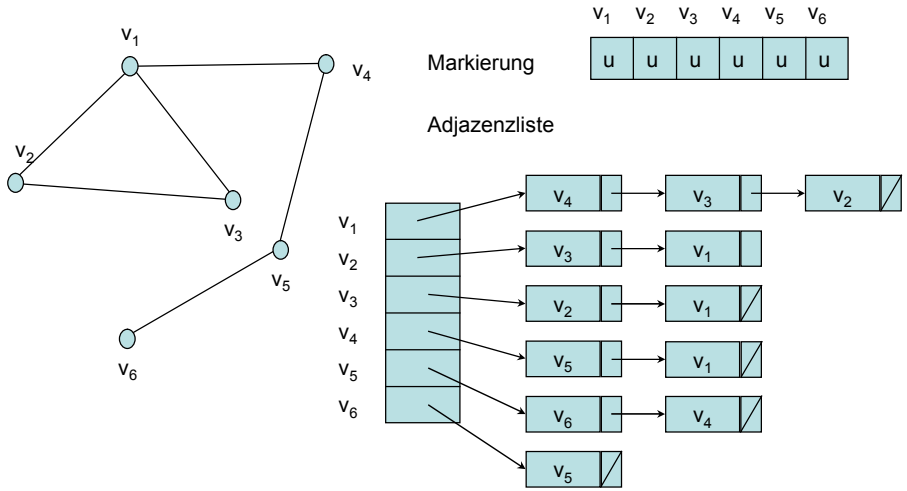
Rekursiver Ansatz

- Markierung: Ein Feld mit den Knotenbezeichnungen als Index, initialisiert mit 'unbesucht' (keine Steine).

Analogie

- Buch lesen, Detailinformation suchen, Sukkzessiver Lernansatz, Entscheidungsbaum, Auswahlkriterien, etc.

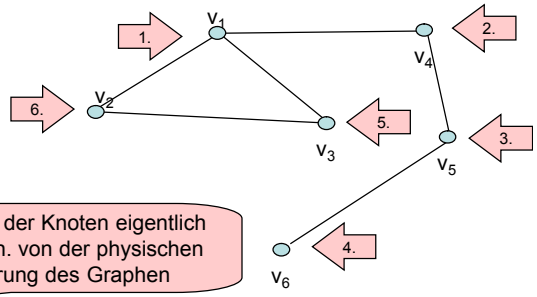
Beispiel, dfs



Methode dfs

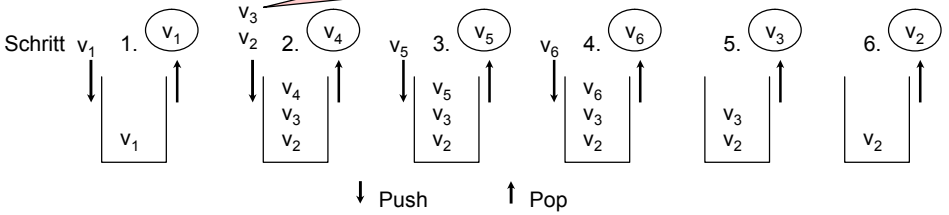
dfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1 wird der Graph mit dem dfs-Ansatz traversiert. Besuchte Knoten werden auf einem Stack (Rekursion) vermerkt.

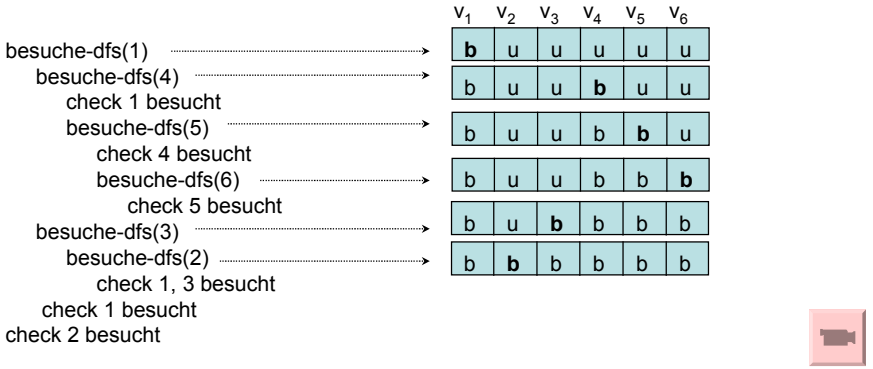


Verhalten des Stacks

Reihenfolge der Knoten eigentlich beliebig, abh. von der physischen Speicherung des Graphen



Beispiel, dfs (2)



C++ - Code

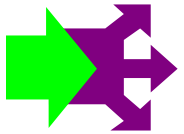
```
#define besucht 1
#define unbesucht 0
// maxV defined elsewhere

class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste
int mark[maxV];  // Knotenmarkierung

void traversiere() {
  int k;
  for(k = 0; k <= maxV; ++k) mark[k] = unbesucht;
  for(k = 0; k <= maxV; ++k) if(mark[k]==unbesucht)
    besuche-dfs(k);
}
void besuche-dfs(int k) {
  node *t;
  mark[k] = besucht;
  for(t = adjliste[k]; t != NULL; t = t->next)
    if(mark[t->v] == unbesucht)
      dfs(t->v);
}
```


6.5.2 Breadth-first Search

Idee



Man leert einen Topf mit Tinte auf den Startknoten. Die Tinte ergießt sich in alle Richtungen (über alle Kanten) auf einmal

Beim breadth-first search Ansatz werden alle möglichen Alternativen auf einmal erforscht, über die gesamte Breite der Möglichkeiten

Dies bedeutet, dass zuerst alle möglichen, von einem Knoten weggehenden, Kanten untersucht werden, und danach erst zum nächsten Knoten weitergegangen wird

Analogie

Überblick über Buch verschaffen, Generelle Information suchen, Hierarchischer Lernansatz, Auswahlüberblick, etc.

Algorithmus

Iterativer Ansatz

zu besuchende Knoten werden in einer Queue gemerkt

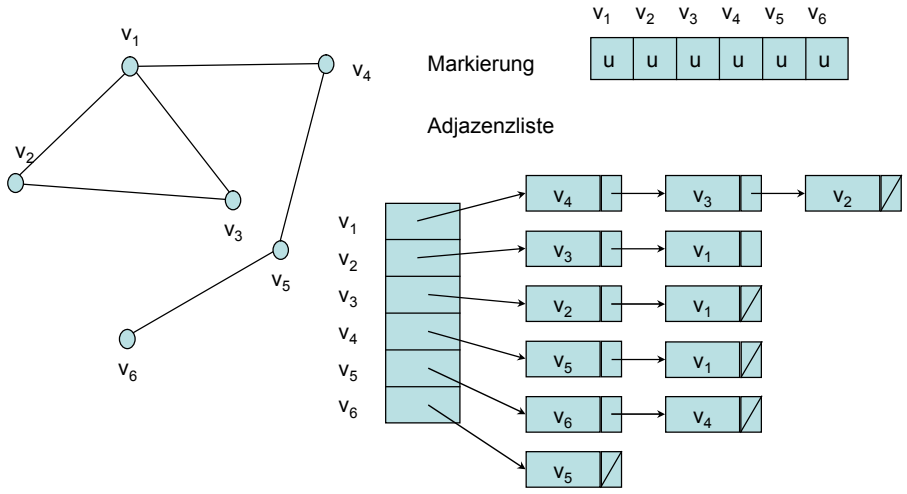
"Zuerst in die Breite , danach in die Tiefe gehen"

Achtung: Da kein Stack „automatisch" zur Verfügung steht, muss eine geeignete Datenstruktur zum „Merken" der zu besuchenden Knoten vorgesehen werden: Queue

```
void besuche-bfs(Knoten x) {
  stelle (Enqueue) Knoten x in Queue;
  markiere Knoten x 'besucht';
  while(Queue nicht leer) {
    hole (Dequeue) ersten Knoten y von der Queue;
    for(alle zu y adjazente nicht besuchte Knoten v) {
      stelle (Enqueue) v in die Queue;
      markiere Knoten v 'besucht';
    }  // for
  }  // while
}
```

Iterativer Ansatz

Beispiel, bfs

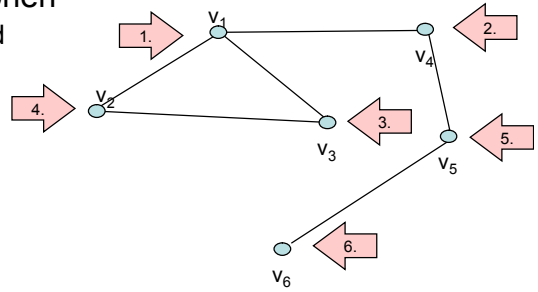


Methode bfs

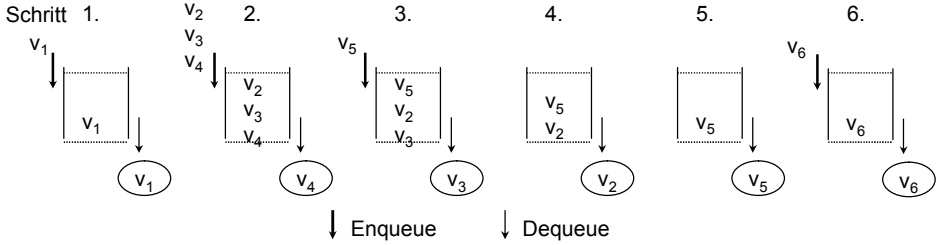
bfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1 wird der Graph mit dem bfs-Ansatz traversiert

Besuchte Knoten werden in einer Queue vermerkt



Verhalten der Queue



Beispiel, bfs (2)

besuche-bfs(1)
Enqueue(1), Dequeue(1)
Enqueue(4,3,2)
Dequeue(4)
Enqueue(5)
check 1 besucht
Dequeue(3)
check 2,1 besucht
Dequeue(2)
check 3,1 besucht
Dequeue(5)
Enqueue(6)
check 4 besucht
Dequeue(6)
check 5 besucht

v1	v2	v3	v4	v5	v6
b	u	u	u	u	u
b	u	u	**b**	u	u
b	u	**b**	b	u	u
b	**b**	b	b	u	u
b	b	b	b	**b**	u
b	b	b	b	b	**b**

C++ - Code

```
#define besucht 1
#define unbesucht 0
// maxV defined elsewhere ,mark initialized „unbesucht“

class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste
int mark[maxV];  // Knotenmarkierung
Queue queue(maxV);

void besuche-bfs(int k) {
  node *t;
  queue.Enqueue(k);
  mark[k]= besucht;
  while(!queue.IsQueueEmpty()) {
    k = queue.Dequeue();
    for(t = adjliste[k]; t != NULL; t = t->next)
      if(mark[t->v] == unbesucht) {
        queue.Enqueue(t->v); mark[t->v] = besucht;
      }
    }
  }
}
```


Aufwand von dfs und bfs

Aufwandsabschätzung abhängig von der Speicherungsform des Graphen

Bestimmend Aufwand für das Finden der adjazenten Knoten

Adjazenzliste

dfs und bfs: O (|V| + |E|)

Jeder Knoten wird einmal besucht, Adjazenzlisten werden iterativ abgearbeitet

Bei vollständig verbundenen Graphen O ($|V^2|$), da jeder Knoten |V|-1 Kanten besitzt, d.h. $|E| \approx |V^2|$

Adjazenzmatrix

dfs und bfs: O ($|V|^2$)

Aufwand zum Finden der adj. Nachfolger eines Knoten O(|V|)

Schichtenkonzept zur Problemlösung

Anwendungsebene:
Beispiele
- günstigste Weg von a nach b
- gute Züge in einem Spiel

Werkzeugebene:
Beispiele
- Graphdurchquerung von v1 nach v2
- Topologische Anordnung

Implementationsebene:
Beispiele
- Rekursive Traversierung
- Iterative Traversierung

Lösungsskizzen

Beispiele

Weg von Knoten x nach y finden

Von x ausgehend Graph traversieren bis man zu y kommt (d.h. y markiert wird).

Beliebigen Kreis im ungerichteten Graph finden

Von jedem Knoten Traversierung des Graphen starten. Kreis ist gefunden, falls man zu einem markierten Knoten kommt (man war schon einmal da).

Beliebigen Kreis im gerichteten Graph finden

Von jedem Knoten Traversierung des Graphen starten. Kreis ist gefunden, falls man zu einem markierten Knoten kommt. Wichtig: Gesetzte Markierungen müssen beim Zurücksteigen wieder gelöscht werden.

…

6.5.3 Das “Bauer, Wolf, Ziege und Kohlkopf”-Problem

Klassisches Problem

Ein Bauer möchte mit einem Wolf, einer Ziege und einem Kohlkopf einen Fluss überqueren. Es steht ihm hierzu ein kleines Boot zur Verfügung, in dem aber nur 2 Platz haben.

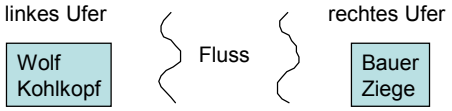
Weiters stellt sich das Problem, dass nur der Bauer rudern kann, und der Wolf mit der Ziege und die Ziege mit dem Kohlkopf nicht allein gelassen werden kann, da sonst der eine den anderen frisst.

Es soll eine Transportfolge gefunden werden, dass alle ‘ungefressen’ das andere Ufer erreichen.

Kodierung des Problems (1)

Das Problem wird durch einen Graphen dargestellt, wobei die Knoten die Positionen der zu Transportierenden und die Kanten die Bootsfahrten repräsentieren.

Eine Position (Knoten) definiert, wer auf welcher Seite des Flusses ist,



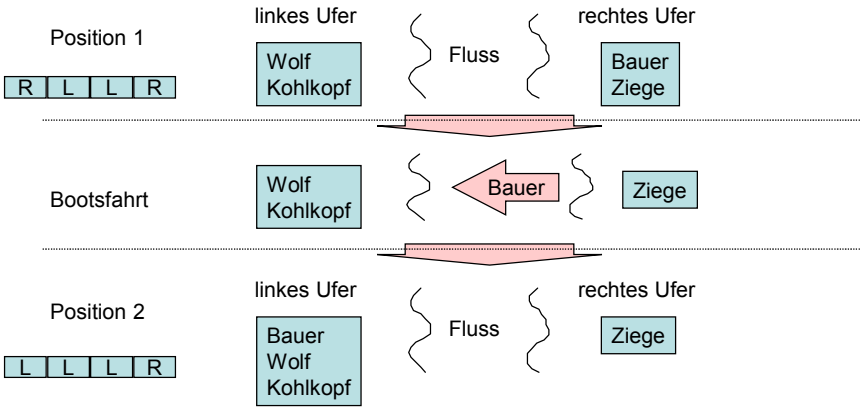
Dieses ist eine ‘sichere’ Position, da niemand gefressen wird.

Eine Position kann in einem 4 elementigen Vektor kodiert werden, wobei jedem der 4 zu Transportierenden ein Vektorelement zugeordnet ist und L das linke Ufer bzw. R das rechte bezeichnet, d.h. der obigen Position entspricht der folgende Vektor

Bauer	Wolf	Kohlkopf	Ziege
R	L	L	R

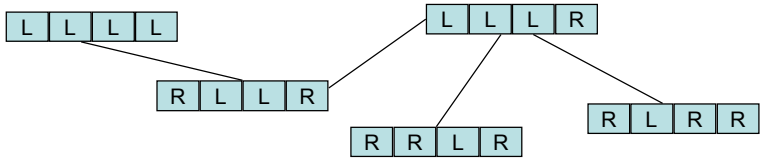
Kodierung des Problems (2)

Eine Bootsfahrt (Kante) gibt an, wer im Boot übersetzt, d.h. führt eine Position in die nächste über, d.h



Problemrepräsentation

Der gesamte Problembereich lässt sich somit durch einen Graphen darstellen, wobei die Positionen durch die Knoten und die Bootsfahrten durch die Kanten repräsentiert werden, z.B. (Ausschnitt)

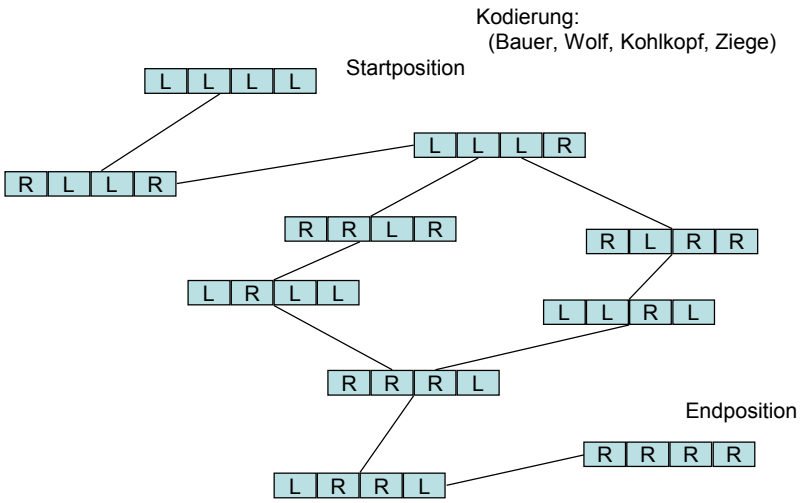


Die Lösung wird somit durch einen Weg bestimmt, der von der Anfangsposition (alle 4 auf dem linken Ufer = LLLL) zu der Endposition (alle 4 auf dem rechten Ufer = RRRR) führt.

Dies lässt sich durch eine simple Traversierung des Graphen lösen.

Lösung

Lösungsweg



C-Programm

Wir wählen einen bfs-Ansatz (iterativ, mit Hilfe einer Queue)

- Die Position wird in einer integer Variable binär (stellenwertig) kodiert
  - Bauer 3. Bit, Wolf 2. Bit, Kohlkopf 1. Bit, Ziege 0.Bit, wobei 0 linkes Ufer und 1 rechtes Ufer bedeutet
- Die Funktionen 'Bauer', 'Wolf', 'Kohl' und 'Ziege' liefern die aktuelle Position zurück.
- Die Funktion 'sicher' bestimmt, ob eine Position sinnvoll ist, d.h. niemand wird gefressen.
- 'DruckeOrt' dient zur verständlichen Ausgabe der Positionen.
- In der Queue 'Zug' werden die zu besuchenden Knoten vermerkt.
- Der beschrittene Weg (Traversierung) wird im Feld 'Weg' gespeichert (max. 16 Positionen).

C-Spezialitäten		
	0x08	Hexadezimaldarstellung einer Zahl
	^	bitweise exklusives Oder, XOR
	<<	Linksshift Operator

Hilfsfunktionen

```
int Bauer(int Ort) {return 0 != (Ort & 0x08);}
int Wolf(int Ort) {return 0 != (Ort & 0x04);}
int Ziege(int Ort) {return 0 != (Ort & 0x02);}
int Kohl(int Ort) {return 0 != (Ort & 0x01);}

int sicher(int Ort) {
   if((Ziege(Ort) == Kohl(Ort)) &&
      (Ziege(Ort) != Bauer(Ort))) return 0;
   if((Wolf(Ort) == Ziege(Ort)) &&
      (Wolf(Ort) != Bauer(Ort))) return 0;
   return 1;
}

void DruckeOrt(int Ort) {
   cout << ((Ort & 0x08) ? "R " : "L ");
   cout << ((Ort & 0x04) ? "R " : "L ");
   cout << ((Ort & 0x02) ? "R " : "L ");
   cout << ((Ort & 0x01) ? "R " : "L ");
   cout << endl;
}
```


Traversierung

```
void main() {
  Queue<int> Zug;
  int Weg[16];
  for(int i = 0; i < 16; i++) Weg[i] = -1;
  Zug.Enqueue(0x00);
  while(!Zug.IsEmpty()) {
    int Ort = Zug.Dequeue();
    for (int Pers = 1; Pers <= 8; Pers <<= 1) {
       int nOrt = Ort ^ (0x08 | Pers);
       if(sicher(nOrt) && (Weg[nOrt] == -1)) {
          Weg[nOrt] = Ort;
          Zug.Enqueue(nOrt);
       }
    }
  }
  cout << "Weg:\n";
  for(int Ort = 15; Ort > 0; Ort = Weg[Ort]) DruckeOrt(Ort);
  cout << '\n';
}
```

Wie wird aus diesem bfs Ansatz ein dfs Ansatz?

6.6 Minimaler Spannender Baum

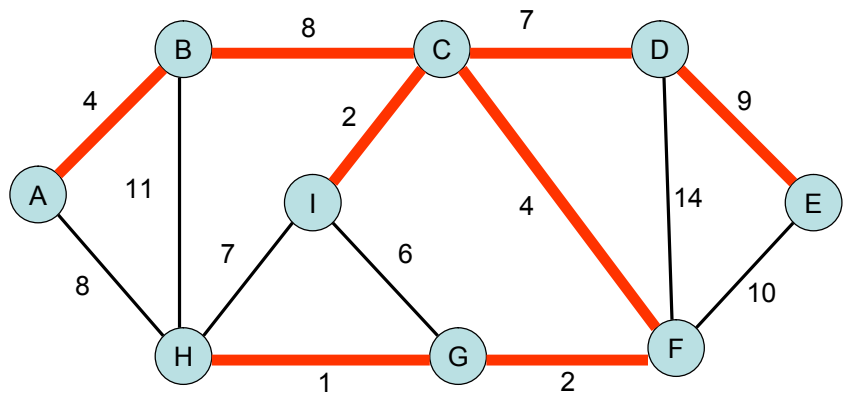
Den Kanten des Graphen G(V, E) ist ein Wert zugeordnet (Netzwerk). Ein Minimaler Spannender Baum MSB ist ist ein spannender Teilgraph, dessen Summe der Kantenwerte minimal ist.

- Für jede Kante [u, v] Î E existiert ein Gewicht w(u, v), welches die Kosten der Verbindung angibt
- Ziel: Finde eine azyklische Teilmenge T Í E, die alle Knoten verbindet und für die gilt, w(T) = $\Sigma_{(u,v) \hat{I} T}$ w(u, v) ist ein Minimum

Beispiel:

- Verdrahtungsproblem in einem elektronischen Schaltplan
- Alle Pins müssen untereinander verdrahtet werden, wobei die Drahtlänge minimal sein soll
  - d.h. V ist die Menge der Pins, E die Menge der möglichen Verbindungen zwischen Pin-Paaren, w(u,v) Drahtlänge zwischen Pin u und v

Beispiel: Minimaler Spannender Baum



Gewicht des MSB: 37
MSB ist nicht eindeutig, Kante [B, C] könnte durch [A, H] ersetzt werden

Generischer MSB Algorithmus

Ansatz durch Greedy Algorithmus

1. Algorithmus verwaltet eine Menge A von Kanten, die immer eine Teilmenge eines möglichen MSB sind
2. MSB wird schrittweise erzeugt, Kante für Kante, wobei für jede Kante überprüft wird, ob sie zu A hinzugefügt werden kann, ohne Bedingung 1 zu verletzen

   So eine Kante heißt „sichere Kante“ (safe edge)

```
Generic-MSB(G, w) {
  A = {};
  while (A bildet keinen MSB) {
    finde eine Kante [u,v] die für A "sicher" ist
    A = A È { [u,v] }
  }
}
```

Wie finde ich so eine sichere Kante?

6.6.1 Kruskal Algorithmus

Grundidee:

1. Die Menge der Knoten repräsentiert einen Wald bestehend aus |V| Komponenten
   - an Beginn keine Kante
2. Die sichere Kante, die hinzugefügt wird, ist immer die Kante mit dem niedrigsten Gewicht, die zwei unterschiedliche Komponenten verbindet

Greedy Ansatz

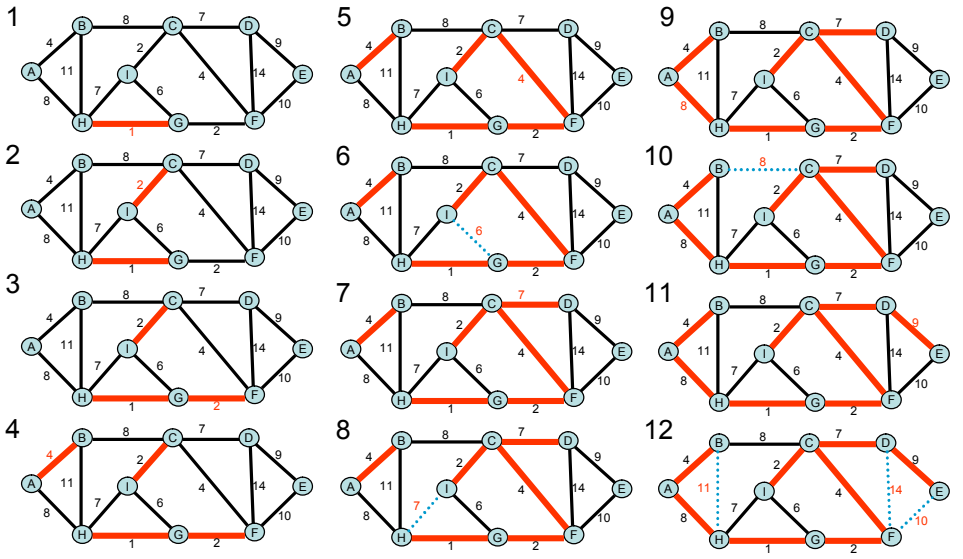
- in jedem Schritt wird die Kante mit niedrigstem Gewicht zum Wald hinzugefügt

Kruskal Algorithmus (2)

```
MSB-Kruskal(G(V, E), w) {
  A = {};
  for (jeden Knoten vÎV)
    make-set(v);
  sortiere die Kanten aus E nach aufsteigendem Gewicht w
  for(jede Kante [u,v]ÎE in der Reihenfolge der Gewichte)
    if(find-set(u) != find-set(v)) {
      A = AÈ{ [u,v] }
      union(u, v)
  }
  return A;
}
```

w enthält die Gewichte zwischen den Knoten, z.B. Gewicht zwischen u und v = **w(u,v)**
make-set(v) erzeugt eine Baum mit Knoten v
find-set(v) liefert ein repräsentatives Element für die Menge die v enthält
union vereinigt 2 Bäume zu einem Baum

Beispiel: Kruskal Algorithmus



Mengenoperationen für Kruskal

```
make-set(x) {
  p[x] = x;
  rank[x] = 0;
}
link(x,y) {
  if (rank[x] > rank[y]) p[y] = x;
  else {
    p[x] = y;
    if (rank[x] = rank[y])
      rank[y] = rank[y] + 1;
  }
}
find-set(x) {
  if(x != p[x]) p[x] = find-set(p[x]);
  return p[x];
}
```

```
union(x, y) {
  link(find-set(x), find-set(y));
}
```

p[x] enthält den Elternknoten von x (Vertreter der Teilmenge)
rank[] ist eine obere Grenze der Höhe von x (Anzahl der Kanten zwischen x und einem Nachfolger-Blatt a
find-set(v) sucht die Wurzel und trägt sie danach jedem Knoten als Elternknoten ein

6.6.2 Prim Algorithmus

Grundidee:

1. Die Menge A bildet einen einzelnen Baum
2. Die sichere Kante, die hinzugefügt wird, ist immer die Kante mit dem niedrigsten Gewicht, die den Baum mit einem Knoten verbindet, der noch nicht im Baum ist

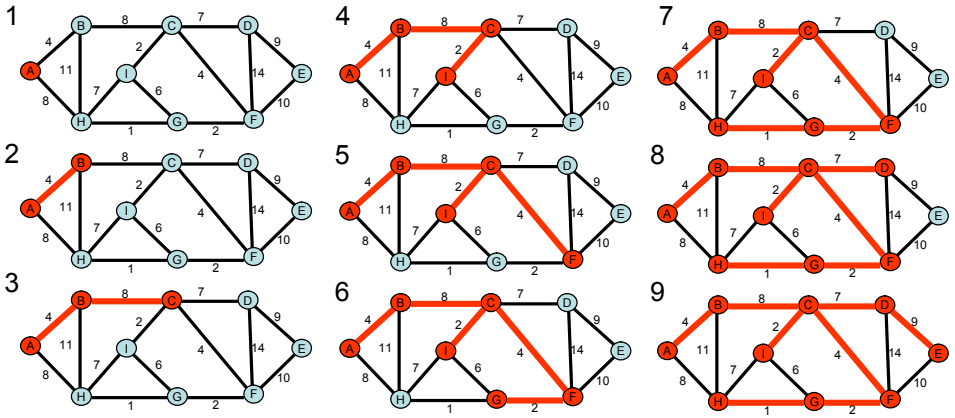
Greedy Ansatz

Prim Algorithmus

```
MSB-Prim(G(V,E), w, r) {
  Q = V;
  for(jeden Knoten u Î Q)
    key[u] = ¥;
  key[r] = 0;
  pred[r] = NIL;
  while(Q != {}) {
    u = Extract-Min(Q)
    for(jeden Knoten v adjazent zu u)
      if(vÎQ && w(u,v) < key[v]) {
        pred[v] = u;
        key[v] = w(u,v);
      }
  }
}
```

Q ist eine Priority Queue die alle Knoten entsprechend ihres **key** Wertes speichert
key[v] ist das minimale Gewicht der Kante die **v** mit einem Knoten im Baum verbindet
r ist die Wurzel (Startknoten) des MSB
pred(v) speichert den Vorgänger von **v**

Beispiel: Prim Algorithmus



Aufwand

Aufwand der Algorithmen stark abhängig von der Implementation der Mengenoperationen bzw. der Datenstrukturen für die Mengenverwaltung (Priority Queue, ...)

Kruskal: Im günstigsten Fall O(E log E)

- Initialisierung O(V)
- Kanten sortieren O(E log E)
- O(E) mal disjunkte Teilmengen finden O(E log E)

Prim: Im günstigsten Fall O(E log V)

- PQ als Heap, d.h. Aufbau O(V log V)
- while Schleife |V| mal, Extract-Min O(log V), d.h. O(V log V)
- For Schleife O(E) mal, key verändern in PQ O(log V), d.h. O(E log V)
- Summe beider Schleifen O(E log V + V log V) = O(E log V)

Lässt sich durch Fibonacci-Heap verbessern auf O(E + V log V)

6.7 Kürzeste Wege

Gewichte zwischen Knoten können als Weglängen oder Kosten angesehen werden

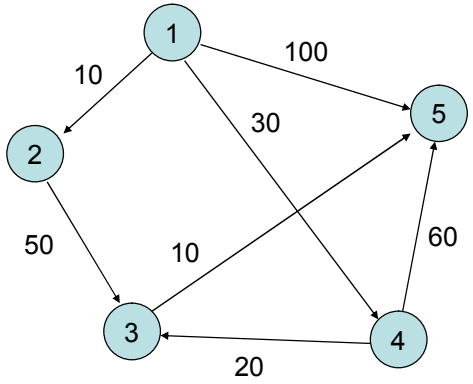
Es ergibt sich oft die Fragestellung nach dem (der) kürzesten Weg(e) zwischen

- Einem Knoten und allen anderen Knoten
- Zwei Knoten
- Zwischen allen Knoten

Annahme: alle Kosten sind gespeichert in der Kostenmatrix C (C[u,v] enthält Kosten von u nach v)

Beispiel: Streckennetz einer Fluglinie

Annahme: nur positive Kantenwerte
Manche Algorithmen funktionieren nur mit positiven Werten



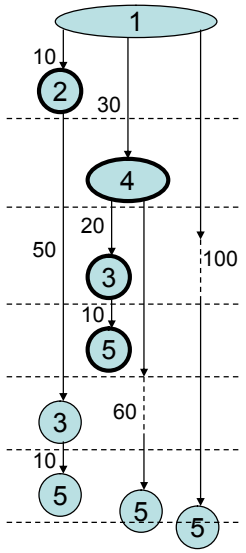
6.7.1 Dijkstra Algorithmus
Single Source Shortest Paths

Algorithmus zur Suche des kürzesten Weges von einem Startknoten zu allen anderen Knoten

Idee

Ausgehend vom Startknoten werden beginnend mit dem kürzesten Weg zu einem Knoten, die nächst-längeren Wege zu allen anderen Knoten gesucht und die kürzesten Wege für die entsprechenden Knoten vermerkt.

Ähnlich Prim's Algorithmus auf der Basis eines bfs mit Prioritätsfunktion der Weglänge

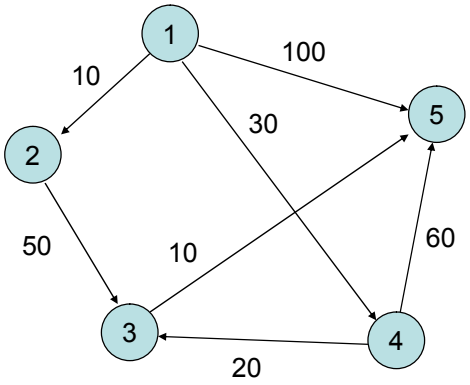


Dijkstra Algorithmus

```
S = { 1 };
for ( i = 2; i <= |V|; i++ )
  MinC[i] = C[1, i];
while ( | V \ S | ) > 1) {
  Akt = Knoten aus V \ S, sodaß MinC[Akt] = Minimum;
  S = S È{ Akt };
  for (jeden Knoten v1 aus V \ S)
    MinC[ v1 ] = MIN (MinC[ v1 ], MinC[Akt] + w[Akt, v1]);
}
```

S Menge der bereits bearbeiteten Knoten
Akt Knoten, der gerade bearbeitete wird
C Kostenmatrix
MinC bisher bekanntester kürzester Weg

Berechnung



S	Akt	MinC
{1}	-	[/, 10, ∞, 30, 100]
{1,2}	2	[/, 10, 60, 30, 100]
{1,2,4}	4	[/, 10, 50, 30, 90]
{1,2,3,4}	3	[/, 10, 50, 30, 60]

6.7.2 All Pairs Shortest Paths

Bevor wir die Fragestellung der kürzesten Wege zwischen allen Knoten klären, wollen wir zuerst die (einfachere) Frage behandeln, zwischen welchen Knoten existieren überhaupt Wege

Führt zu 2 Algorithmen

- Warshall: welche Knoten sind durch Wege verbunden
- Floyd: alle kürzesten Wege zwischen den Knoten

Transitive Hülle

Fragestellung welche Knoten durch Wege verbunden (erreichbar) sind führt zur Frage nach der Transitiven Hülle

Ein gerichteter Graph G‘(V, E‘) wird transitive (und reflexive) Hülle eines Graphen G(V, E) genannt, genau dann wenn:

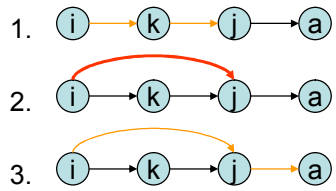
(v, w) ∈ E‘ ⇔ es existiert ein Pfad von v nach w in G

Ausgangspunkt: Adjazenzmatrix von G

Warshall Algorithmus

Idee

- Iteratives Hinzufügen von Kanten im Graphen für Pfade der Länge 2
  - d.h. Pfad (i,k) und (k,j) wird durch neue Kante (i,j) beschrieben
    - in weiterer Folge wird dann natürlich auch Pfad (i,j) und (j,a) als Pfad der Länge 2 gefunden (in Wirklichkeit klarerweise Pfad der Länge 3), usw.
  - Führt dazu, dass die neuen Kanten immer längere Pfade beschreiben, bis alle möglichen Pfade gefunden sind
- Ansatz durch 3 verschachtelte Schleifen, ähnlich der Matrizenmultiplikation



Warshall Algorithmus (2)

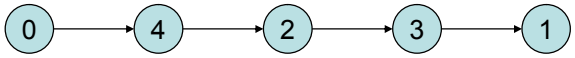
```
warshall (Matrix a) {
  n = a.numberofRows();
  for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
        a[i,j] = a[i,j] | a[i,k] & a[k,j];
}
```

a Adjazenzmatrix
bei **a** als Bit-Matrix
| binärer OR Operator
& binärer AND Operator

Fügt Pfad als neue Kante (falls sie noch nicht existiert) in den Graphen ein

True (1), falls Weg zwischen i und j über k existiert

Beispiel: Warshall (1)



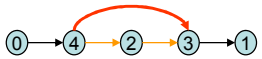
	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	0

	0	1	2	3	4
0	0	1	1	1	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0

Beispiel: Warshall (2)

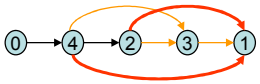
Werte für K (Durchläufe äußere Schleife):
0: keine Kante, da a(?,0) nicht möglich
1: keine Kante, da a(1,?) nicht möglich

2: Kante a(4,2), a(2,3) ⇒ a(4,3)



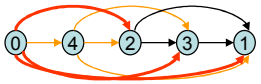
	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	1	0

3: a(2,3), a(3,1) ⇒ a(2,1)
a(4,3), a(3,1) ⇒ a(4,1)



	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0

4: a(0,4), a(4,1) ⇒ a(0,1)
a(0,4), a(4,2) ⇒ a(0,2)
a(0,4), a(4,3) ⇒ a(0,3)



	0	1	2	3	4
0	0	1	1	1	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0

Floyd Algorithmus

Berechnung der kürzesten Wege zwischen allen Knoten ist simpel aus dem Warshall Algorithmus ableitbar

- Wird wie Treesort Floyd zugeschrieben

Unterschied

- Adjazenzmatrix speichert die Weglängen zwischen den Knoten (entspricht der Kostenmatrix)
  - Statt 0/1 Werte die Wegkosten
- Beim Feststellen eines Pfades über 2 Kanten einfache Berechnung der Weglänge dieser neuen Kante und Vergleich mit aktueller Weglänge (falls schon vorhanden)
  - Statt logisches AND die Berechnung der Kostensumme

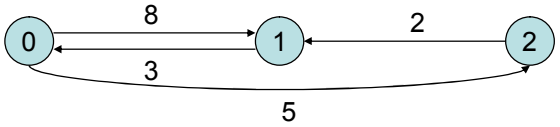
Floyd Algorithmus (2)

```
floyd (Matrix a) {
  n = a.numberofRows();
  for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++) {
        int newPathLength = a[i,k] + a[k,j];
        if(newPathLength < a[i,j])
          a[i,j] = newPathLength;
      }
}
```

Berechnung der Weglänge über neuen Pfad

Eintrag in die Kostenmatrix, falls neuer Weg kürzer als möglicherweise vorhandener alter Weg

Beispiel: Floyd



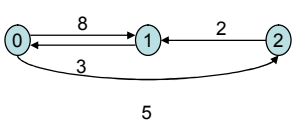
	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

⇒

	0	1	2
0	0	7	5
1	3	0	8
2	5	2	0

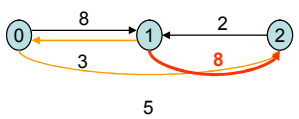
Beispiel: Floyd (2)

Ausgangsposition



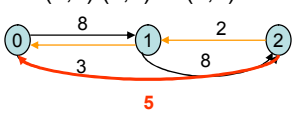
	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

k: 0
Kanten: (1,0) (0,2) ⇒ (1,2), Kosten: 8



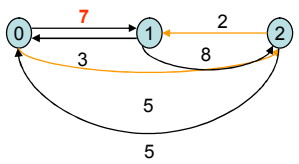
	0	1	2
0	0	8	5
1	3	0	8
2	¥	2	0

k: 1
(0,1) (1,2) ⇒ (0,2): 16
(2,1) (1,0) ⇒ (2,0): 5
(2,1) (1,2) ⇒ (2,2):10



	0	1	2
0	0	8	5
1	3	0	8
2	5	2	0

k: 2
(0,2) (2,0) ⇒ (0,0): 10
(0,2) (2,1) ⇒ (0,1): 7
(1,2) (2,0) ⇒ (1,0): 13



	0	1	2
0	0	7	5
1	3	0	8
2	5	2	0

6.8 Genereller iterativer Ansatz

Genereller iterativer Ansatz zur Traversierung eines Graphen mit Hilfe 2 (simpler) Listen

- OpenList
  - Speichert bekannte aber noch nicht besuchte Knoten
- CloseList
  - Speichert alle schon besuchten Knoten

Expandieren eines Knoten

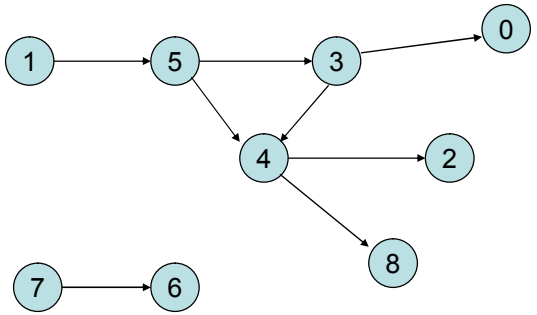
- Generieren der Nachfolger eines Knoten
- d.h. OpenList enthält alle generierten (bekannten), aber noch nicht expandierten Knoten, CloseList alle expandierten Knoten

Ohne weitere Steuerung Traversierungsansatz unsystematisch

Unsystematische Traversierung

```
Search(Knoten v) {
  OpenList = [v];
  CloseList = [];
  while (OpenList != []) {
    Akt = beliebigerKnotenAusOpenList;  // ???
    OpenList = OpenList \ [Akt];
    CloseList = CloseList + [Akt];
    process(Akt);  // whatever to do
    for(alle zu Akt adjazente Knoten v1){
      if (v1 Ï OpenList && v1 Ï CloseList)
        OpenList = OpenList + [v1];
    }
  }
}
```

Beispiel (F)



z.B.: **Search(5)**
OpenList: 5|3,4|4,0|0,2,8|2,8|8|
CloseList: 5,3,4,0,2,8
Akt: 5|3|4|0|2|8

Es werden alle Knoten besucht, die vom Startknoten aus erreichbar sind. Unterschied gerichteter – ungerichtetere Graph?

6.8.1 Zusammenhangskomponente im ungerichteten Graphen

Erweiterung von **Search**

Feld zum Vermerken der Komponenten K[|V|]

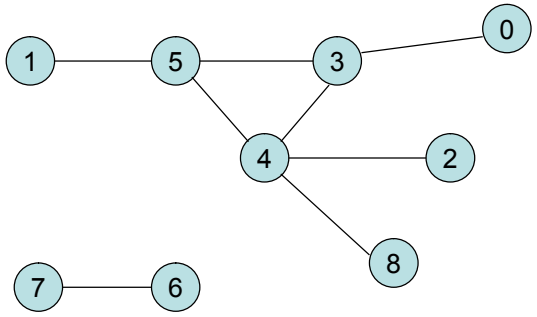
```
int Vanz = |V|;
int Knum = 0;
for (int i = 0; i < Vanz; i++)
  if(K[i] == 0) {
    Knum = Knum + 1;
    SearchAndMark(i, Knum);
  }

SearchAndMark(Knoten v; int Knum) {
  neue Version von Search wobei
  process(Akt); entspricht K[Akt] = Knum;
}
```


Zusammenhangskomponente

```
SearchAndMark(Knoten v; int Knum) {
  OpenList = [v];
  CloseList = [];
  while (OpenList != []) {
    Akt = beliebigerKnotenAusOpenList;
    OpenList = OpenList \ [Akt];
    CloseList = CloseList + [Akt];
    K[Akt] = Knum;
    for(alle zu Akt adjazente Knoten v1){
      if (v1 Ï OpenList && v1 Ï CloseList)
        OpenList = OpenList + [v1];
    }
  }
}
```


Beispiel (F)



Frage: Was wäre bei einem gerichteten Graphen?

OpenList: 5|1,3,4|3,4|4,0|0,2,8|2,8|2|6|7
CloseList: 5,1,3,4,0,2,8|6,7
Akt: 5|1|3|4|0|2|8|6|7

K

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

Systematisierung der Traversierung

Systematische Traversierung schon bekannt

- Tiefensuche dfs
- Breitensuche bfs

Unklarer Programmteil

- **`beliebigerKnotenAusOpenList`**

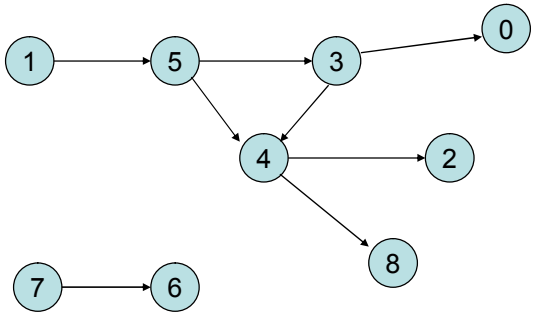
Systematisierung durch Organisation der Auswahl

- Aufbau der Liste
  - Position des einzufügenden Elements in der Liste
  - bfs: **`OpenList = OpenList + [v1];`** am Ende
  - dfs: **`OpenList = [v1] + OpenList;`** am Anfang
- Zugriff auf die OpenList
  - Zugriff: **`First(OpenList);`** Zugriff immer auf das erste Element

Systematische Traversierung

```
Search(Knoten v) {
  OpenList = [v];
  CloseList = [];
  while (OpenList != []) {
    Akt = First(OpenList);
    OpenList = OpenList \ [Akt];
    CloseList = CloseList + [Akt];
    process(Akt);
    for(alle zu Akt adjazente Knoten v1){
      if (v1 Ï OpenList && v1 Ï CloseList)
        dfs: OpenList = [v1] + OpenList;
        bfs: OpenList = OpenList + [v1];
    }
  }
}
```


Beispiel (F)



z.B.: **`Search(5)`** mit dfs
OpenList: 5|3,4|0,4|4|2,8|8|
CloseList: 5,3,0,4,2,8
Akt: 5|3|0|4|2|8

z.B.: **`Search(5)`** mit bfs
OpenList: 5|3,4|4,0|0,2,8|2,8|8|
CloseList: 5,3,4,0,2,8
Akt: 5|3|4|0|2|8

Noch ein kleines Problem

Systematisierung noch nicht ganz vollständig

Anweisung

`for(alle zu Akt adjazente nicht besuchte Knoten v1){`

führt zu undefinierter Reihenfolge aller adjazenter Knoten bei der Weiterbearbeitung

- Reihenfolge definiert durch die interne Graphenspeicherung
  - Adjazenzliste, Adjazenzmatrix

Annahme: adjazente Knoten werden aufsteigend sortiert eingetragen

6.8.2 Dijkstra Algorithmus mit OL/CL Kürzester Weg von S nach Z

```
OpenList = [S]; CloseList = [];
while (true) {
  if ( OpenList == [] )  return –1 // war nix!
  Akt = ElementAusOpenList, sodass Akt.c = min;
  if (Akt = Z) return Akt.c;
  OpenList = OpenList \ [Akt];
  CloseList = CloseList + [Akt];
  for (jeden Knoten v1 adjazent zu Akt) {
    if ( v1 Ï OpenList && v1 Ï CloseList ) {
      v1.c = Akt.c + C[Akt, v1];
      OpenList = OpenList + [v1];
    } else {
      if (v1 Î OpenList)
        if (Akt.c + C[Akt,v1] < v1.c)
          v1.c = Akt.c + C[Akt,v1];
    }
}
```

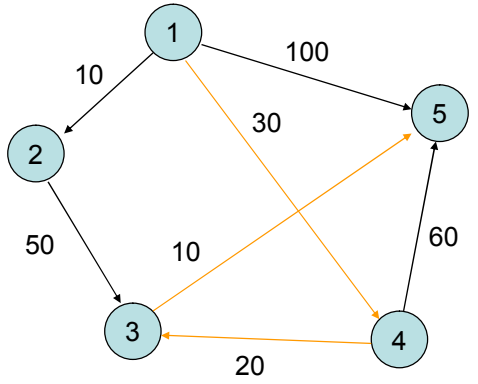
v.c Speichert die Länge des aktuell kürzesten Weges vom Startknoten zum Knoten v

Beispiel (F)

Gesucht: kürzester Weg von 1 nach 5

OpenList:
1|2,4,5|5,3|5|
CloseList:
1,2,4,3
Akt:
1|2|4|3|5
c:

1	2	3	4	5
0	10	~~60~~ 50	30	~~100~~ ~~90~~ 60



Beweisskizze

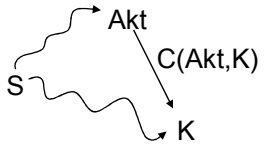
Satz: Falls ein Knoten v bereits in der CloseList ist, kann es keinen günstigeren Pfad vom Startknoten nach v geben als jenen, der bereits berechnet wurde

d.h. nochmals besucht Knoten, die schon in der CloseList sind, können einen Pfad nicht verkürzen

Annahme: K in CloseList und Akt.c + C(Akt, K) < K.c

In dem Schritt, als K aus der OpenList in die CloseList gebracht wurde, galt offenbar
K.c ≤ Akt.c ∨ K.c ≤ pred(Akt).c

pred(Akt) bezeichnet einen Vorgänger von Akt



Es gilt aber auch pred(Akt).c < Akt.c, somit also
K.c ≤ Akt.c ∨ K.c ≤ pred(Akt).c < Akt.c ⇒ K.c ≤ Akt.c

Widerspruch: es gibt keine positiven Akt.c, C(Akt, K), K.c, sodass diese Ungleichung gilt

Was nehmen wir mit?

Graphen
- Definitionen
- Gerichtete und ungerichtete Graphen

Topologisches Sortieren

Traversierung
- Bfs und dfs
- „Bauer, Wolf, Ziege du Kohlkopf“ Problem

Spannende Bäume

Kürzeste Wege

Generell iterativer Ansatz