Algorithmen und Datenstrukturen 2

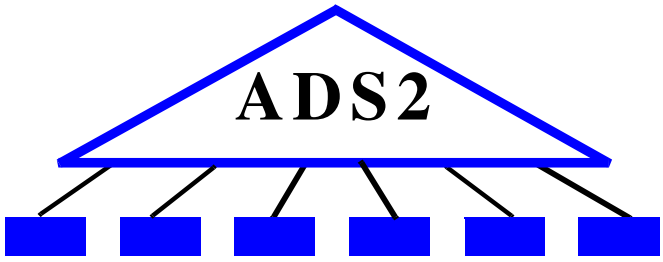
Prof. Dr. E. Rahm

Sommersemester 2002

Universität Leipzig

Institut für Informatik

http://dbs.uni-leipzig.de



Zur Vorlesung allgemein

- Vorlesungsumfang: 2 + 1 SWS
- Vorlesungsskript
  - im WWW abrufbar (PDF, PS und HTML)
  - Adresse *http://dbs.uni-leipzig.de*
  - ersetzt nicht die Vorlesungsteilnahme oder zusätzliche Benutzung von Lehrbüchern
- Übungen
  - Durchführung in zweiwöchentlichem Abstand
  - selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
  - Übungsblätter im WWW
  - praktische Übungen auf Basis von Java
- Vordiplomklausur ADS1+ADS2 im Juli
  - Zulassungsvoraussetzungen: Übungsschein ADS1 + erfolgreiche Übungsbearbeitung ADS2
  - erfordert fristgerechte Abgabe und korrekte Lösung der meisten Aufgaben sowie Bearbeitung aller Übungsblätter (bis auf höchstens eines)

Termine Übungsbetrieb

- Ausgabe 1. Übungsblatt: Montag, 8. 4. 2002; danach 2-wöchentlich
- Abgabe gelöster Übungsaufgaben bis spätestens Montag der übernächsten Woche, 11:15 Uhr
  - vor Hörsaal 13 (Abgabemöglichkeit 11:00 - 11:15 Uhr)
  - oder früher im Fach des Postschranks HG 2. Stock, neben Raum 2-22
  - Programmieraufgaben: dokumentierte Listings der Quellprogramme sowie Ausführung
- 6 Übungsgruppen

Nr.	Termin	Woche	Hörsaal	**Beginn**	Übungsleiter	#Stud.
1	Mo, 17:15	A	HS 16	**22.4.**	Richter	60
2	Mo, 9:15	B	SG 3-09	**29.4.**	Richter	30
3	Di, 11:15	A	SG 3-07	**23.4.**	Böhme	30
4	Di, 11:15	B	SG 3-07	**30.4.**	Böhme	30
5	Do, 11.15	A	SG 3-05	**25.4.**	Böhme	30
6	Do, 11.15	B	SG 3-05	**2.5.**	Böhme	30

  - Einschreibung über Online-Formular
  - Aktuelle Infos siehe WWW

Ansprechpartner ADS2

- Prof. Dr. E. Rahm
  - während/nach der Vorlesung bzw. Sprechstunde (Donn. 14-15 Uhr), HG 3-56
  - rahm@informatik.uni-leipzig.de
- Wissenschaftliche Mitarbeiter
  - Timo Böhme, boehme@informatik.uni-leipzig.de, HG 3-01
  - Dr. Peter Richter, prichter@informatik.uni-leipzig.de, HG 2-20
- Studentische Hilfskräfte
  - Tilo Dietrich, TiloDietrich@gmx.de
  - Katrin Starke, katrin.starke@gmx.de
  - Thomas Tym, mai96iwe@studserv.uni-leipzig.de
- Web-Angelegenheiten:
  - S. Jusek, juseks@informatik.uni-leipzig.de, HG 3-02

Vorläufiges Inhaltsverzeichnis

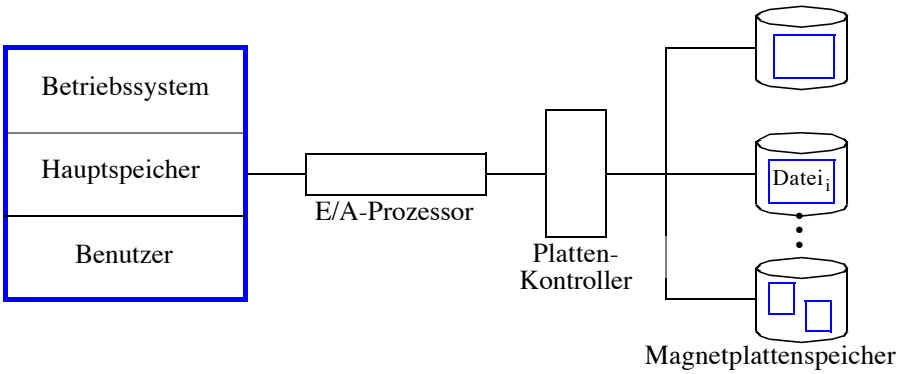
Literatur

Das intensive Literaturstudium zur Vertiefung der Vorlesung wird dringend empfohlen.
Auch Literatur in englischer Sprache sollte verwendet werden.

- *T. Ottmann, P. Widmayer:* Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 4. Auflage, Spektrum-Verlag, 2002
- Weitere Bücher
  - *V. Claus, A. Schwill:* Duden Informatik, BI-Dudenverlag, 3. Auflage 2001
  - *D.A. Knuth:* The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973
  - *R. Sedgewick:* Algorithmen. Addison-Wesley 1992
  - *G. Saake, K. Sattler:* Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 2002
  - *M.A. Weiss:* Data Structures & Problem Solving using Java. Addison-Wesley, 2. Auflage 2002

Suchverfahren für große Datenmengen

- bisher betrachtete Datenstrukturen (Arrays, Listen, Binärbäume) und Algorithmen waren auf im Hauptspeicher vorliegende Daten ausgerichtet
- effiziente Suchverfahren für große Datenmengen auf Externspeicher erforderlich (persistente Speicherung)
  - große Datenmengen können nicht vollständig in Hauptspeicher-Datenstrukturen abgebildet werden
  - Zugriffsgranulat sind Seiten bzw. Blöcke von Magnetplatten : z.B. 4-16 KB
  - Zugriffskosten 5 Größenordnungen langsamer als für Hauptspeicher (5 ms vs. 50 ns)

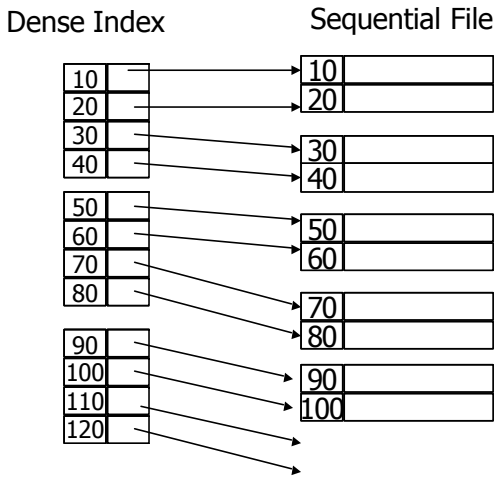


Sequentieller Dateizugriff

- Sequentielle Dateiorganisation
  - Datei besteht aus Folge gleichartiger Datensätze
  - Datensätze sind auf Seiten/Blöcken gespeichert
  - ggf. bestimmte Sortierreihenfolge (bzgl. eines Schlüssels) bei der Speicherung der Sätze (sortiert-sequentielle Speicherung)
- Sequentieller Zugriff
  - Lesen aller Seiten / Sätze vom Beginn der Datei an
  - sehr hohe Zugriffskosten, v.a. wenn nur ein Satz benötigt wird
- Optimierungsmöglichkeiten
  - „dichtes Packen“ der Sätze innerhalb der Seiten (hohe Belegungsdichte)
  - Clusterung zwischen Seiten, d.h. „dichtes Packen“ der Seiten einer Datei auf physisch benachbarte Plattenbereiche, um geringe Zeiten für Plattenzugriffsarm zu ermöglichen
- Schneller Zugriff auf einzelne Datensätze erfordert Einsatz von zusätzlichen *Indexstrukturen*, z.B. Mehrwegbäume
- Alternative: gestreute Speicherung der Sätze (-> Hashing)

Dichtbesetzter vs. dünnbesetzter Index

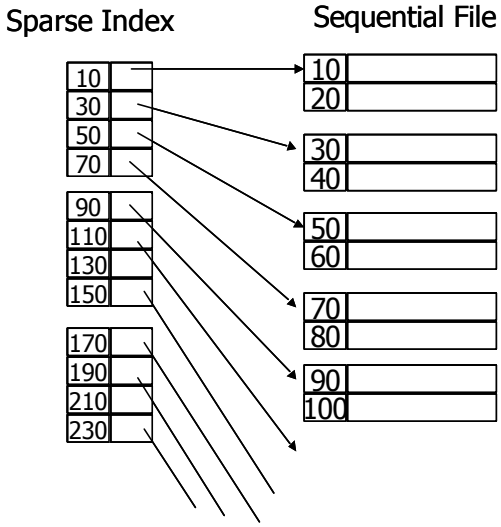
- Dichtbesetzter Index (dense index)
  - für jeden Datensatz existiert ein Eintrag in Indexdatei
  - höherer Indexaufwand als bei dünnbesetztem Index
  - breiter anwendbar, u.a auch bei unsortierter Speicherung der Sätze
  - einige Auswertungen auf Index möglich, ohne Zugriff auf Datensätze (Existenztest, Häufigkeitsanfragen, Min/Max-Bestimmung)



- Anwendungsbeispiel
  - 1 Million Sätze, B=20, 200 Indexeinträge pro Seite
  - Dateigröße:
  - Indexgröße:
  - mittlere Zugriffskosten:

Dichtbesetzter vs. dünnbesetzter Index (2)

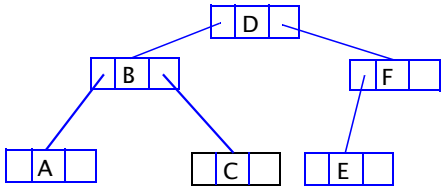
- Dünnbesetzter Index (sparse index)
  - nicht für jeden Schlüsselwert existiert Eintrag in Indexdatei
  - sinnvoll v.a. bei Clusterung gemäß Sortierreihenfolge des Indexattributes: ein Indexeintrag pro Datenseite
- indexsequentielle Datei (ISAM): sortierte sequentielle Datei mit dünnbesetztem Index für Sortierschlüssel



- Anwendungsbeispiel
  - 1 Million Sätze, B=20, 200 Indexeinträge pro Seite
  - Dateigröße:
  - Indexgröße:
  - mittlere Zugriffskosten:
- Mehrstufiger Index
  - Indexdatei entspricht sortiert sequentieller Datei -> kann selbst wieder indexiert werden
  - auf höheren Stufen kommt nur dünnbesetzte Indexierung in Betracht
  - beste Umsetzung im Rahmen von Mehrwegbäumen (B-/B*-Bäume)

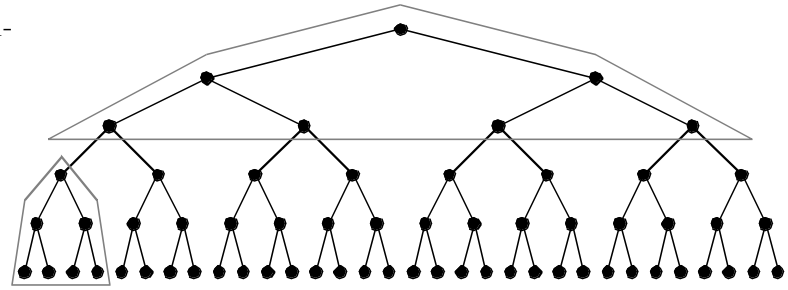
Mehrwegbäume

- Ausgangspunkt: Binäre Suchbäume (balanciert)
  - entwickelt für Hauptspeicher
  - ungeeignet für große Datenmengen



- Externspeicherzugriffe erfolgen auf Seiten
  - Abbildung von Schlüsselwerten/Sätzen auf Seiten
  - Index-Datenstruktur für schnelle Suche

Beispiel: Zuordnung von Binärbaum-Knoten zu Seiten

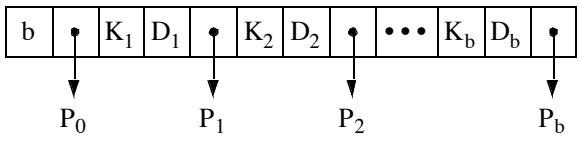


- Alternativen:
  - m-Wege-Suchbäume
  - B-Bäume
  - B*-Bäume
- Grundoperationen: Suchen, Einfügen, Löschen
- Kostenanalyse im Hinblick auf Externspeicherzugriffe

m-Wege-Suchbäume

- **Def.:** Ein m-Wege-Suchbaum oder ein m-ärer Suchbaum B ist ein Baum, in dem alle Knoten einen Grad ≤ m besitzen. Entweder ist B leer oder er hat folgende Eigenschaften:

(1) Jeder Knoten des Baums mit b Einträgen, $b \le m - 1$, hat folgende Struktur:



Die P_i, $0 \le i \le b$, sind Zeiger auf die Unterbäume des Knotens und die K_i und D_i, $1 \le i \le b$ sind Schlüsselwerte und Daten.

(2) Die Schlüsselwerte im Knoten sind aufsteigend geordnet: $K_i \le K_{i+1}$, $1 \le i < b$.

(3) Alle Schlüsselwerte im Unterbaum von P_i sind kleiner als der Schlüsselwert K_{i+1}, $0 \le i < b$.

(4) Alle Schlüsselwerte im Unterbaum von P_b sind größer als der Schlüsselwert K_b.

(5) Die Unterbäume von P_i, $0 \le i \le b$ sind auch m-Wege-Suchbäume.

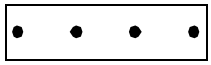
- Die D_i können Daten oder Zeiger auf die Daten repräsentieren
  - direkter Index: eingebettete Daten (weniger Einträge pro Knoten; kleineres m)
  - indirekter Index: nur Verweise; erfordert separaten Zugriff auf Daten zu dem Schlüssel

m-Wege-Suchbäume (2)

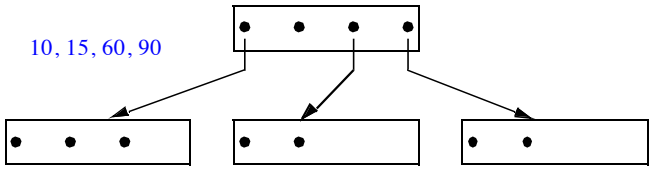
- **Beispiel:** Aufbau eines m-Wege-Suchbaumes (m = 4)

Einfügereihenfolge:

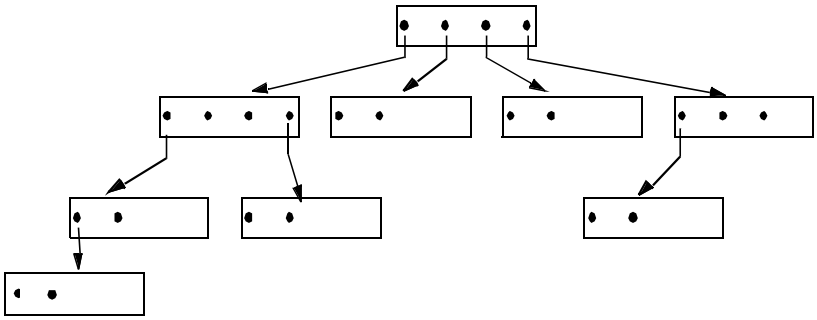
30, 50, 80



10, 15, 60, 90



20, 35, 5, 95, 1, 25



- Beobachtungen
  - Die Schlüssel in den inneren Knoten besitzen zwei Funktionen. Sie identifizieren Daten(sätze) und sie dienen als Wegweiser in der Baumstruktur
  - Der m-Wege-Suchbaum ist im allgemeinen nicht ausgeglichen

m-Wege-Suchbäume (3)

- Wichtige Eigenschaften für alle Mehrwegbäume:

  $S(P_i)$ sei die Seite, auf die P_i zeigt, und $K(P_i)$ sei die Menge aller Schlüssel, die im Unterbaum mit Wurzel $S(P_i)$ gespeichert werden können. Dann gelten folgende Ungleichungen:

  (1) $x \in K(P_0)$: $x < K_1$

  (2) $x \in K(P_i)$: $K_i < x < K_{i+1}$ für $i = 1, 2, \ldots, b-1$

  (3) $x \in K(P_b)$: $K_b < x$

- Kostenanalyse
  - Die Anzahl der Knoten N in einem vollständigen Baum der Höhe h, $h \geq 1$, ist

    $$N = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}$$

  - Im ungünstigsten Fall ist der Baum völlig entartet: $n = N = h$
  - Schranken für die Höhe eines m-Wege-Suchbaums: $\log_m(n+1) \leq h \leq n$

m-Wege-Suchbäume (4)

- Definition des Knotenformats:

```
class MNode {
  int m;               // max. Grad des Knotens (m)
  int b;               // Anzahl der Schluessel im Knoten (b <= m-1)
  Orderable[] keys;    // Liste der Schluessel
 Object[] data;      // Liste der zu den Schluesseln gehoerigen Datenobjekte
  MNode[] ptr;         // Liste der Zeiger auf Unterbaeume

  /** Konstruktor */
  public MNode(int m, Orderable key, Object obj) {
    this.m = m; b = 1;
    keys = new Orderable[m-1];
    data = new Object[m-1];
    ptr = new MNode[m];
    keys[0] = key;  // Achtung: keys[0] entspricht K1, keys[1] K2, ...
    data[0] = obj;  // Achtung: data[0] entspricht D1, data[1] D2, ...
  } }
```

- Rekursive Prozedur zum Aufsuchen eines Schlüssels

```
public Object search(Orderable key, MNode node) {
    if ((node == null) || (node.b < 1)) {
      System.err.println("Schluessel nicht im Baum.");
      return null; }
```



```
    if (key.less(node.keys[0]))
      return search(key, node.ptr[0]);       // key < K1

    if (key.greater(node.keys[node.b-1]))
      return search(key, node.ptr[node.b]);  // key > Kb

    int i=0;
    while ((i<node.b-1) && (key.greater(node.keys[i])))
        i++;                                 // gehe weiter, solange key > Ki+1

    if (key.equals(node.keys[i]))
      return node.data[i];                 // gefunden

    return search(key, node.ptr[i]);   // Ki < key < Ki+1
  }
```

- Durchlauf eines m-Wege-Suchbaums in symmetrischer Ordnung

```
public void print(MNode node) {
    if ((node == null) || (node.b < 1)) return;
    print(node.ptr[0]);
    for (int i=0; i<node.b; i++) {
      System.out.println("Schluessel: " + node.keys[i].getKey() +
                         " \tDaten: " + node.data[i].toString());
      print(node.ptr[i+1]);
    } }
```


Mehrwegbäume

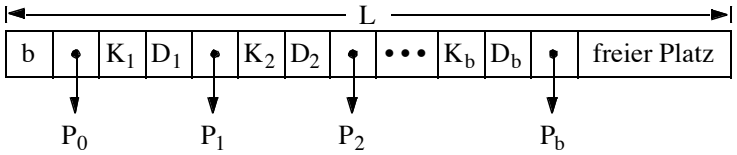
- Ziel: Aufbau sehr breiter Bäume von geringer Höhe
  - in Bezug auf Knotenstruktur vollständig ausgeglichen
  - effiziente Grundoperationen auf Seiten (= Transporteinheit zum Externspeicher)
  - Zugriffsverhalten weitgehend unabhängig von Anzahl der Sätze

  ⇒ Einsatz als Zugriffs-/Indexstruktur für 10 als auch für 10^{10} Sätze
- Grundoperationen:
  - direkter Schlüsselzugriff auf einen Satz
  - sortiert sequentieller Zugriff auf alle Sätze
  - Einfügen eines Satzes; Löschen eines Satzes
- Varianten
  - ISAM-Dateistruktur (1965; statisch, periodische Reorganisation)
  - Weiterentwicklungen: B- und B*-Baum
  - B-Baum: 1970 von R. Bayer und E. McCreight entwickelt

  ⇒ dynamische Reorganisation durch Splitten und Mischen von Seiten
- Breites Spektrum von Anwendungen ("The Ubiquitous B-Tree")
  - Dateiorganisation ("logische Zugriffsmethode", VSAM)
  - Datenbanksysteme (Varianten des B*-Baumes sind in allen DBS zu finden!)
  - Text- und Dokumentenorganisation . . .

B-Bäume

- **Def.:** Seien k, h ganze Zahlen, $h \geq 0$, $k > 0$.
  Ein B-Baum B der Klasse $\tau(k, h)$ ist entweder ein leerer Baum oder ein geordneter Baum mit folgenden Eigenschaften:
  1. Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge h-1.
  2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne
  3. Jeder Knoten hat höchstens 2k+1 Söhne
  4. Jedes Blatt mit der Ausnahme der Wurzel als Blatt hat mindestens k und höchstens 2k Einträge.
- Für einen B-Baum ergibt sich folgendes Knotenformat:



B-Bäume (2)

- Einträge
  - Die Einträge für Schlüssel, Daten und Zeiger haben die festen Längen l_b, l_K, l_D und l_p.
  - Die Knoten- oder Seitengröße sei L.
  - Maximale Anzahl von Einträgen pro Knoten: $b_{max} = \left\lfloor \frac{L - l_b - l_p}{l_K + l_D + l_p} \right\rfloor = 2k$
- Reformulierung der Definition

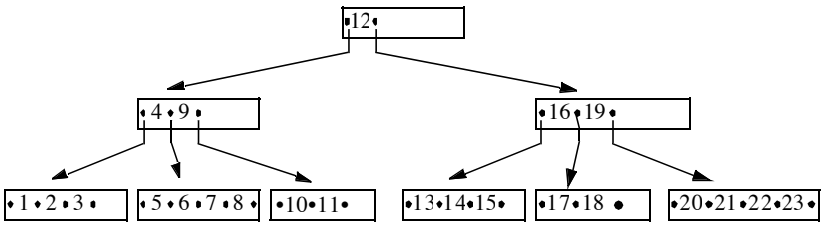
  (4) und (3). Eine Seite darf höchstens voll sein.

  (4) und (2). Jede Seite (außer der Wurzel) muß mindestens halb voll sein.
  Die Wurzel enthält mindestens einen Schlüssel.

  (1) Der Baum ist, was die Knotenstruktur angeht, vollständig ausgeglichen
- Balancierte Struktur:
  - unabhängig von Schlüsselmenge
  - unabhängig ihrer Einfügereihenfolge

B-Bäume (3)

- Beispiel: B-Baum der Klasse τ (2,3)



  - In jedem Knoten stehen die Schlüssel in aufsteigender Ordnung mit $K_1 < K_2 < \ldots < K_b$
  - Jeder Schlüssel hat eine Doppelrolle als Identifikator eines Datensatzes und als Wegweiser im Baum
  - Die Klassen $\tau(k,h)$ sind nicht alle disjunkt. Beispielsweise ist ein maximaler Baum aus $\tau(2,3)$ ebenso in $\tau(3,3)$ und $\tau(4,3)$ ist
- **Höhe h**: Bei einem Baum der Klasse $\tau(k,h)$ mit n Schlüsseln gilt für seine Höhe:

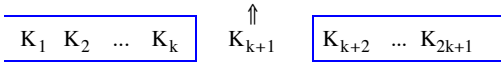
$$\log_{2k+1}(n+1) \le h \le \log_{k+1}((n+1)/2) + 1 \quad \text{für } n \ge 1$$

und $h = 0$ für $n = 0$

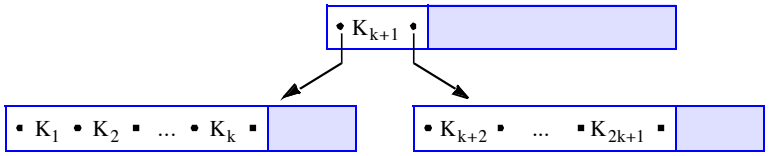
Einfügen in B-Bäumen

- Was passiert, wenn Wurzel überläuft ?

- Fundamentale Operation: Split-Vorgang
  1. Anforderung einer neuen Seite und
  2. Aufteilung der Schlüsselmenge nach folgendem Prinzip



- mittlere Schlüssel (Median) wird zum Vaterknoten gereicht
- Ggf. muß Vaterknoten angelegt werden (Anforderung einer neuen Seite).



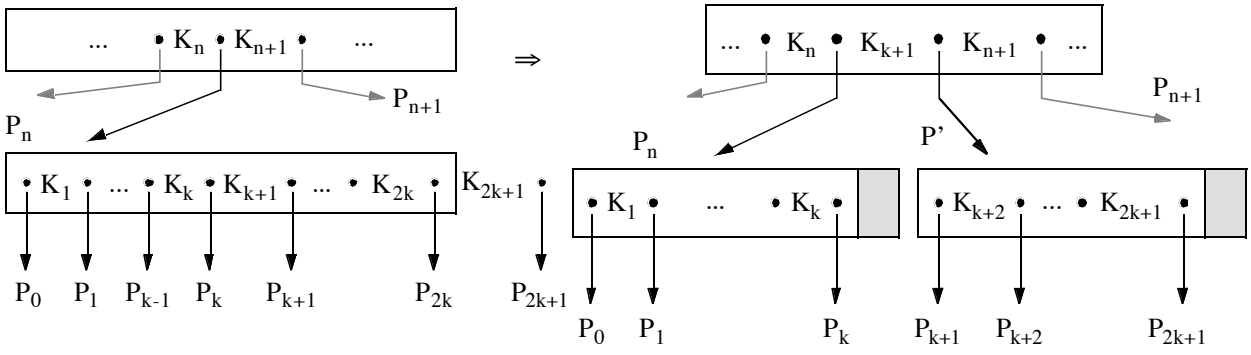
Split führt hier zu einer neuen Wurzel a

- Blattüberlauf erzwingt Split-Vorgang, was Einfügung in den Vaterknoten impliziert
- Wenn dieser überläuft, folgt erneuter Split-Vorgang
- Split-Vorgang der Wurzel führt zu neuer Wurzel: Höhe des Baumes erhöht sich um 1

- Bei B-Bäumen ist Wachstum von den Blättern zur Wurzel hin gerichtet

Einfügen in B-Bäumen (2)

- Einfügealgorithmus (ggf. rekursiv)
  - Suche Einfügeposition
  - Wenn Platz vorhanden ist, speichere Element, sonst schaffe Platz durch Split-Vorgang und füge ein
- Split-Vorgang als allgemeines Wartungsprinzip

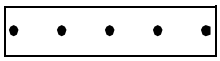


Einfügen in B-Bäumen (3)

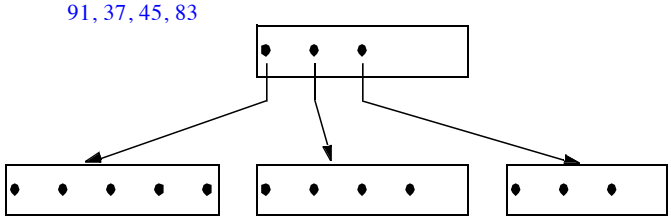
- Aufbau eines B-Baumes der Klasse τ(2, h)

Einfügereihenfolge:

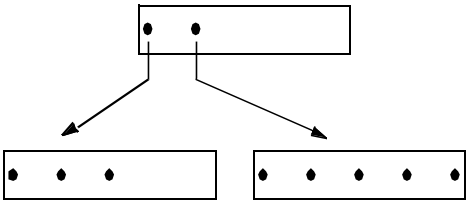
77, 12, 48, 69



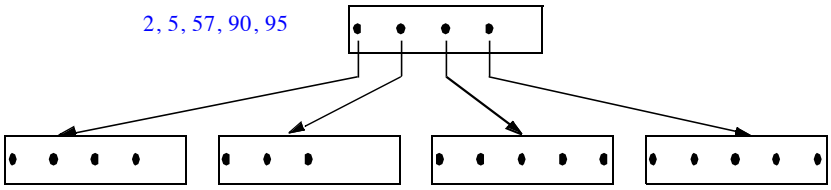
91, 37, 45, 83



33, 89, 97

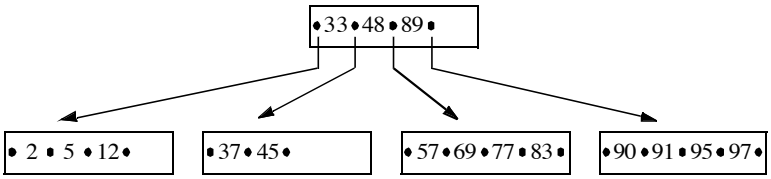


2, 5, 57, 90, 95

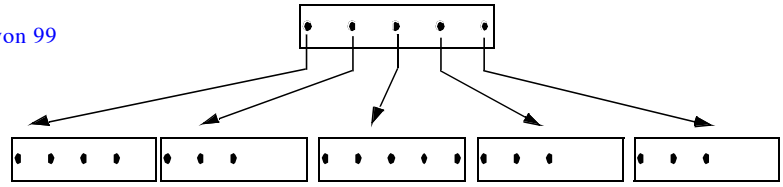


Einfügen in B-Bäumen (4)

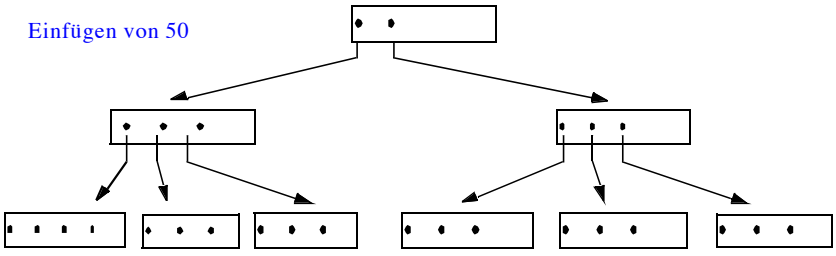
- Aufbau eines B-Baumes der Klasse τ(2, h)



Einfügen von 99



Einfügen von 50

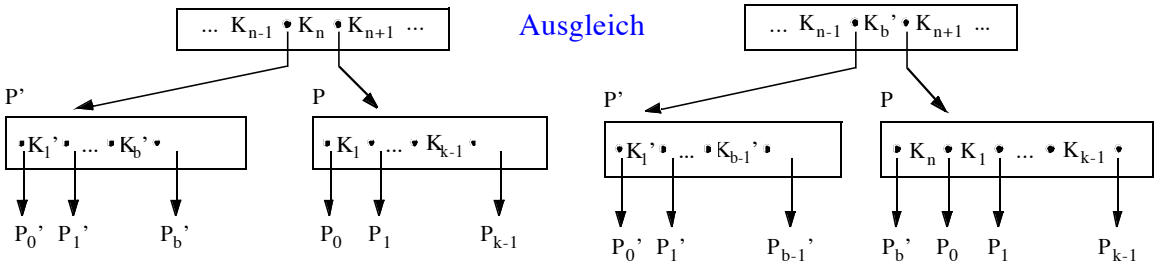


Kostenanalyse für Suche und Einfügen

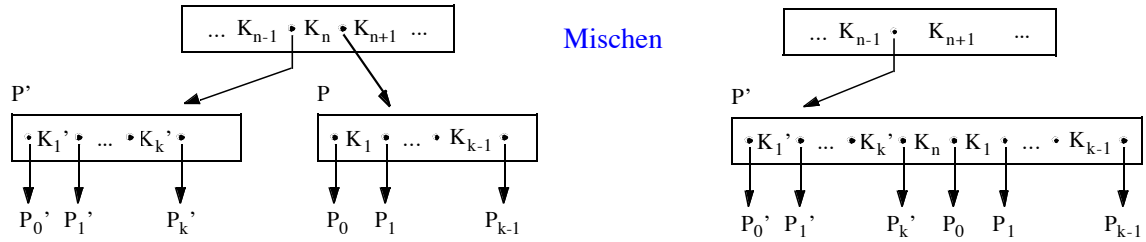
- Kostenmaße
  - Anzahl der zu holenden Seiten: f (fetch)
  - Anzahl der zu schreibenden Seiten (#geänderter Seiten): w (write)
- Direkte Suche
  - $f_{min} = 1$: der Schlüssel befindet sich in der Wurzel
  - $f_{max} = h$: der Schlüssel ist in einem Blatt
  - **mittlere Zugriffskosten** $h - \frac{1}{k} \le f_{avg} \le h - \frac{1}{2k}$ (für $h > 1$)
- Beim B-Baum sind die maximalen Zugriffskosten h eine gute Abschätzung der mittleren Zugriffskosten.
  ⇒ Bei $h = 3$ und einem $k = 100$ ergibt sich $2.99 \le f_{avg} \le 2.995$
- Sequentielle Suche
  - Durchlauf in symmetrischer Ordnung : $f_{seq} = N$
  - Pufferung der Zwischenknoten im Hauptspeicher wichtig!
- Einfügen
  - günstigster Fall - kein Split-Vorgang: $f_{min} = h;\ w_{min} = 1$
  - durchschnittlicher Fall: $f_{avg} = h;\ w_{avg} < 1 + \frac{2}{k}$

Löschen in B-Bäumen

- Die B-Baum-Eigenschaft muß wiederhergestellt werden, wenn die Anzahl der Elemente in einem Knoten kleiner als k wird.
- Durch **Ausgleich** mit Elementen aus einer Nachbarseite oder durch **Mischen** (Konkatenation) mit einer Nachbarseite wird dieses Problem gelöst.
  - Maßnahme 1: Ausgleich durch Verschieben von Schlüsseln (Voraussetzung: Nachbarseite P' hat mehr als k Elemente; Seite P hat k-1 Elemente)



  - Maßnahme 2: Mischen von Seiten



Löschen in B-Bäumen (2)

- Löschalgorithmus

(1) Löschen in Blattseite

- Suche x in Seite P
- Entferne x in P und wenn
  - a) #E ≥ k in P: tue nichts
  - b) #E = k-1 in P und #E > k in P': gleiche Unterlauf über P' aus
  - c) #E = k-1 in P und #E = k in P': mische P und P'.

(2) Löschen in innerer Seite

- Suche x
- Ersetze $x = K_i$ durch kleinsten Schlüssel y in $B(P_i)$ oder größten Schlüssel y in $B(P_{i-1})$ (nächstgrößerer oder nächstkleinerer Schlüssel im Baum)
- Entferne y im Blatt P
- Behandle P wie unter 1

- Kostenanalyse für das Löschen

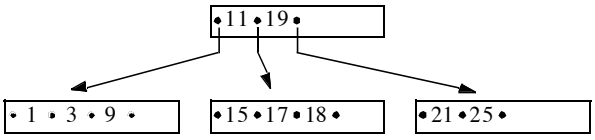
- günstigster Fall: $f_{min} = h$; $w_{min} = 1$

- obere Schranke für durchschnittliche Löschkosten (drei Anteile: 1. Löschen, 2. Ausgleich, 3. anteilige Mischkosten):

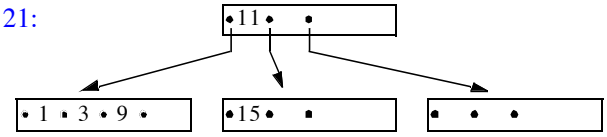
$$f_{avg} \le f_1 + f_2 + f_3 < h + 1 + \frac{1}{k} \qquad w_{avg} \le w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$

-

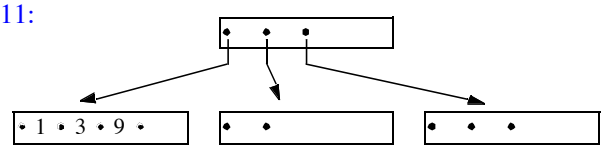
Löschen in B-Bäumen: Beispiel

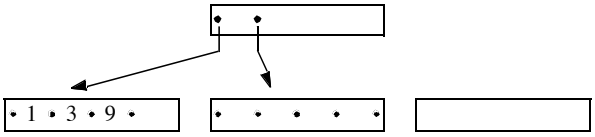


Lösche 21:



Lösche 11:



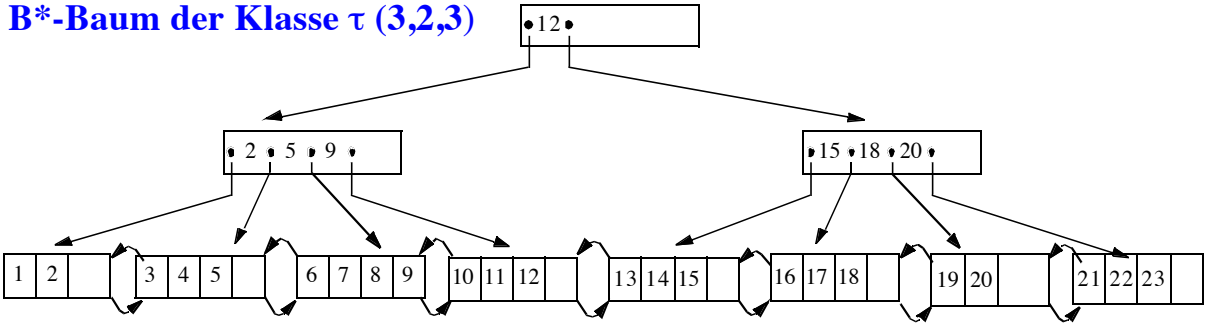


B*-Bäume

- Hauptunterschied zu B-Baum: in inneren Knoten wird nur die Wegweiser-Funktion ausgenutzt
  - innere Knoten führen nur (K_i, P_i) als Einträge
  - Information (K_i, D_i) wird in den Blattknoten abgelegt. Dabei werden alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge in den Blättern abgelegt werden.
  - Für einige K_i ergibt sich eine redundante Speicherung. Die inneren Knoten bilden also einen Index, der einen schnellen direkten Zugriff zu den Schlüsseln gestattet.
  - Der Verzweigungsgrad erhöht sich beträchtlich, was wiederum die Höhe des Baumes reduziert
  - Durch Verkettung aller Blattknoten läßt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte

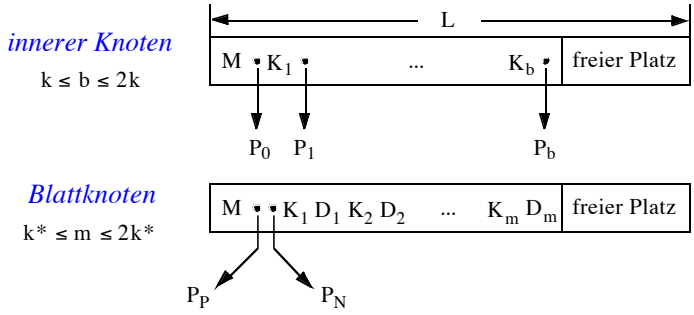
  ⇒ B*-Baum ist die für den praktischen Einsatz wichtigste Variante des B-Baums

B*-Baum der Klasse τ (3,2,3)



B*-Bäume (2)

- **Def.:** Seien k, k* und h* ganze Zahlen, h* ≥ 0, k, k* > 0.
  Ein B*-Baum B der Klasse τ (k, k*, h*) ist entweder ein leerer Baum oder ein geordneter Baum, für den gilt:
  1. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge h*-1.
  2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.
  3. Jeder innere Knoten hat höchstens 2k+1 Söhne.
  4. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k* und höchstens 2k* Einträge.
- Unterscheidung von zwei Knotenformaten:



Feld M enthalte Kennung des Seitentyps sowie Zahl der aktuellen Einträge

B*-Bäume(3)

- Da die Seiten eine feste Länge L besitzen, läßt sich aufgrund der obigen Formate k und k* bestimmen:

$$L = l_M + l_P + 2 \cdot k(l_K + l_P) \; ; \qquad k = \left\lfloor \frac{L - l_M - l_P}{2 \cdot (l_K + l_P)} \right\rfloor$$

$$L = l_M + 2 \cdot l_P + 2 \cdot k^*(l_K + l_D) \; ; \; k^* = \left\lfloor \frac{L - l_M - 2l_P}{2 \cdot (l_K + l_D)} \right\rfloor$$

- Höhe des B*-Baumes

$$1 + \log_{2k+1}\left(\frac{n}{2k^*}\right) \le h^* \le 2 + \log_{k+1}\left(\frac{n}{2k^*}\right) \quad \text{für} \quad h^* \ge 2 \; .$$

B- und B*-Bäume

- Quantitativer Vergleich
  - Seitengröße sei L= 2048 B. Zeiger P_i, Hilfsinformation und Schlüssel K_i seien 4 B lang. Fallunterscheidung:
  - eingebettete Speicherung: l_D = 76 Bytes
  - separate Speicherung: l_D = 4 Bytes, d.h., es wird nur ein Zeiger gespeichert.

	B-Baum	B*-Baum
n_{min}	$2 \cdot (k+1)^{h-1} - 1$	$2k^* \cdot (k+1)^{h^*-2}$
n_{max}	$(2k+1)^h - 1$	$2k^* \cdot (2k+1)^{h^*-1}$

- Allgemeine Zusammenhänge:

- Vergleich für Beispielwerte:

B-Baum

	Datensätze separat (k=85)		Datensätze eingebettet (k=12)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	170	1	24
2	171	29.240	25	624
3	14.791	5.000.210	337	15.624
4	1.272.112	855.036.083	4.393	390.624

B*-Baum

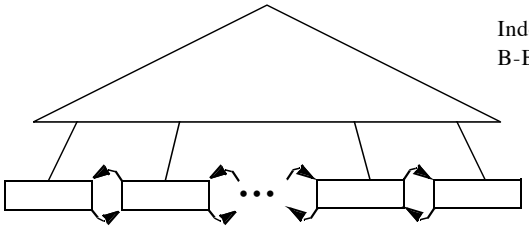
	Datensätze separat (k=127, k* = 127)		Datensätze eingebettet (k=12, k* = 127)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	254	1	24
2	254	64.770	24	6.120
3	32.512	16.516.350	3.072	1.560.600
4	4.161.536	4.211.669.268	393.216	397.953.001

Historie und Terminologie

- Originalpublikation B-Baum:
  - R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:4. 1972. 290-306.
- Überblick:
  - D. Comer: The Ubiquitous B-Tree. ACM Computing Surveys, 11:2, Juni 1979, pp. 121-137.
- B*-Baum Originalpublikation:
  D. E. Knuth: The Art of Programming, Vol. 3, Addison-Wesley, 1973.
- Terminologie:
  - Bei Knuth: B*-Baum ist ein B-Baum mit garantierter 2 / 3-Auslastung der Knoten
  - B+-Baum ist ein Baum wie hier dargestellt
  - Heutige Literatur: B*-Baum = B+-Baum.

B*-Bäume: Operationen

- B*-Baum entspricht einer geketteten sequentiellen Datei von Blättern, die einen Indexteil besitzt, der selbst ein B-Baum ist. Im Indexteil werden insbesondere beim Split-Vorgang die Operationen des B-Baums eingesetzt.



- Grundoperationen beim B*-Baum

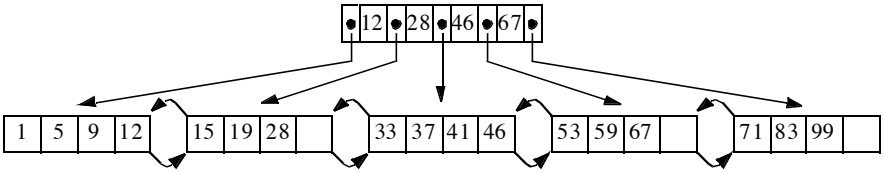
(1) Direkte Suche: Da alle Schlüssel in den Blättern, kostet jede direkte Suche h* Zugriffe. h* ist jedoch im Mittel kleiner als h in B-Bäumen (günstigeres f_{avg} als beim B-Baum)

(2) Sequentielle Suche: Sie erfolgt nach Aufsuchen des Linksaußen der Struktur unter Ausnutzung der Verkettung der Blattseiten. Es sind zwar ggf. mehr Blätter als beim B-Baum zu verarbeiten, doch da nur h*-1 innere Knoten aufzusuchen sind, wird die sequentielle Suche ebenfalls effizienter ablaufen.

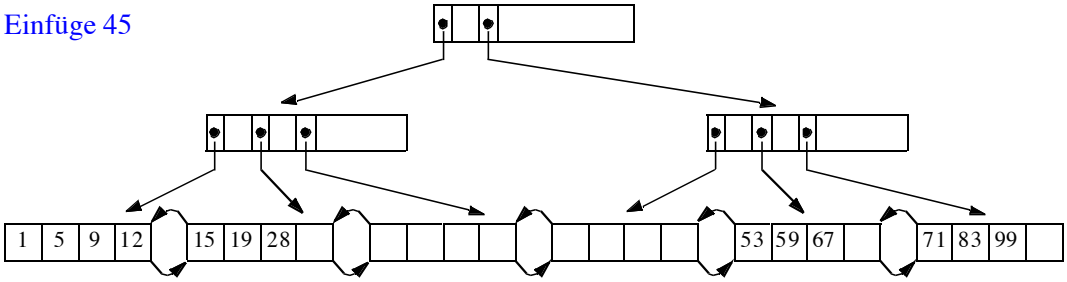
(3) Einfügen: Von Durchführung und Leistungsverhalten dem Einfügen in einen B-Baum sehr ähnlich. Bei inneren Knoten wird die Spaltung analog zum B-Baum durchgeführt. Beim Split-Vorgang einer Blattseite muß gewährleistet sein, daß jeweils die höchsten Schlüssel einer Seite als Wegweiser in den Vaterknoten kopiert werden.

(4) Löschen: Datenelemente werden immer von einem Blatt entfernt (keine komplexe Fallunterscheidung wie beim B-Baum). Weiterhin muß beim Löschen eines Schlüssels aus einem Blatt dieser Schlüssel nicht aus dem Indexteil entfernt werden; er behält seine Funktion als Wegweiser.

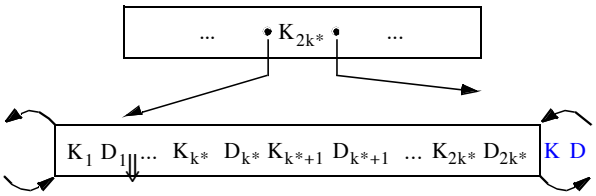
Einfügen im B*-Baum

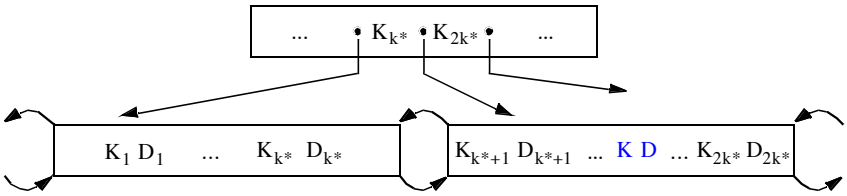


Einfüge 45

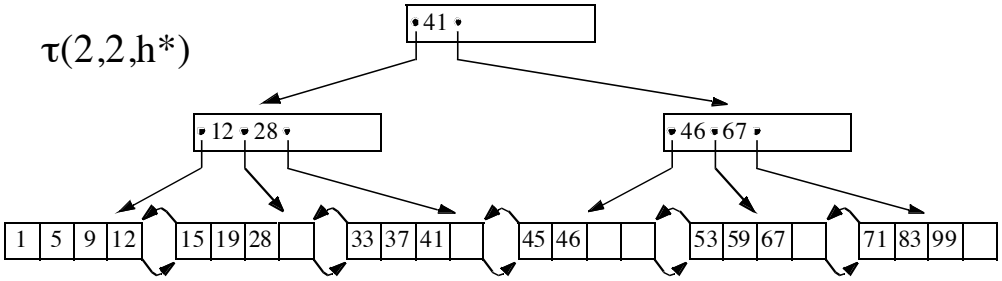


B*-Bäume: Schema für Split-Vorgang

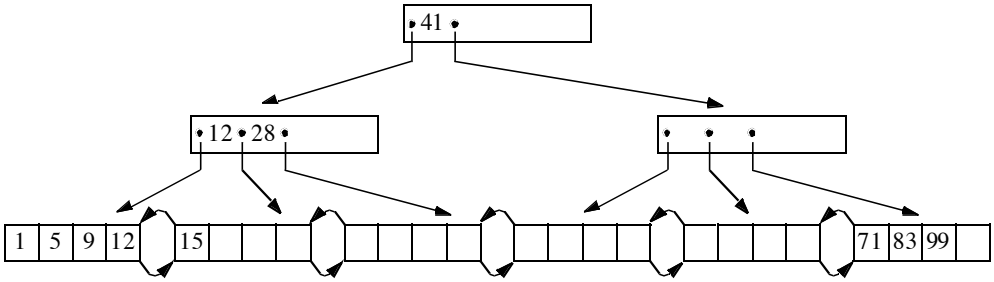




Löschen im B*-Baum:Beispiel

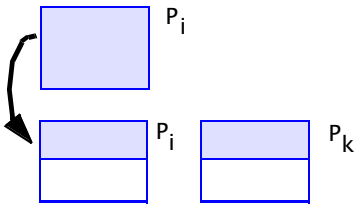


Lösche 28, 41, 46



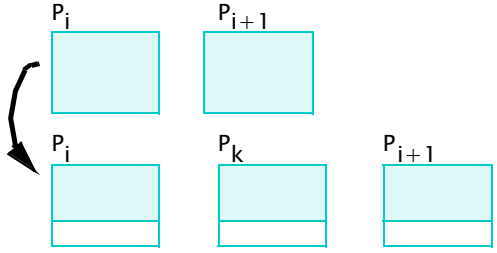
Verallgemeinerte Überlaufbehandlung

Standard (m= 1): Überlauf führt zu zwei halb vollen Seiten

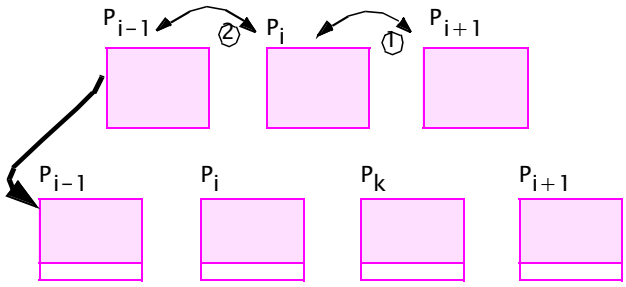


m > 1: Verbesserung der Belegung

m = 2



m = 3



Verallgemeinerte Überlaufbehandlung (2)

- **Speicherplatzbelegung** als Funktion des Split-Faktors

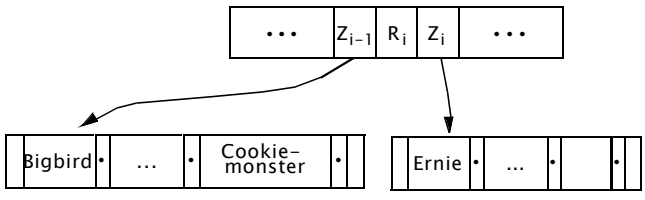
Split-Faktor	Belegung β_{min}	Belegung β_{avg}	Belegung β_{max}
1	1/2 = 50%	ln 2 ≈ 69%	1
2	2/3 = 66%	2 · ln (3/2) ≈ 81%	1
3	3/4 = 75%	3 · ln (4/3) ≈ 86%	1
m	$\frac{m}{m+1}$	$m \cdot \ln\left(\frac{m+1}{m}\right)$	1

- Vorteile der höheren Speicherbelegung
  - geringere Anzahl von Seiten reduziert Speicherbedarf
  - geringere Baumhöhe
  - geringerer Aufwand für direkte Suche
  - geringerer Aufwand für sequentielle Suche
- erhöhter Split-Aufwand (m > 3 i.a. zu teuer)

Schlüsselkomprimierung

- Zeichenkomprimierung ermöglicht weit höhere Anzahl von Einträgen pro Seite (v.a. bei B*-Baum)
  - Verbesserung der Baumbreite (höherer Fan-Out)
  - wirkungsvoll v.a. für lange, alphanumerische Schlüssel (z.B. Namen)
- Präfix-Komprimierung
  - mit Vorgängerschlüssel übereinstimmender Schlüsselanfang (Präfix) wird nicht wiederholt
  - v.a. wirkungsvoll für Blattseiten
  - höherer Aufwand zur Schlüsselrekonstruktion
- Suffix-Komprimierung
  - für innere Knoten ist vollständige Wiederholung von Schlüsselwerten meist nicht erforderlich, um Wegweiserfunktion zu erhalten
  - Weglassen des zur eindeutigen Lokalisierung nicht benötigten Schlüsselendes (Suffix)
  - *Präfix-B-Bäume*: Verwendung minimale Separatoren (Präfxe) in inneren Knoten

Schlüssel	F	kompr. Schlüssel
HARALD		
HARTMUT		
HEIN		
HEINRICH		
HEINZ		
HELMUT		
HOLGER		



Schlüsselkomprimierung (2)

- für Zwischenknoten kann Präfix- und Suffix-Komprimierung kombiniert werden: *Präfix-Suffix-Komprimierung* (Front and Rear Compression)
  - gespeichert werden nur solche Zeichen eines Schlüssels, die sich vom Vorgänger und Nachfolger unterscheiden
  - u.a. in VSAM eingesetzt
- *Verfahrensparameter*:

  V = Position im Schlüssel, in der sich der zu komprimierende Schlüssel vom *Vorgänger* unterscheidet

  N = Position im Schlüssel, in der sich der zu komprimierende Schlüssel vom *Nachfolger* unterscheidet

  F = V - 1 (Anzahl der Zeichen des komprimierten Schlüssels, die mit dem Vorgänger übereinstimmen)

  L = MAX (N-F, 0) Länge des komprimierten Schlüssels

Schlüssel	V	N	F	L	kompr. Schlüssel
HARALD					
HARTMUT					
HEIN					
HEINRICH					
HEINZ					
HELMUT					
HOLGER					

- Durschschnittl. komprimierte Schlüssellänge ca. 1.3 - 1.8

Präfix-Suffix-Komprimierung: weiteres Anwendungsbeispiel

Schlüssel (unkomprimiert)		*V*	*N*	*F*	*L*	*Wert*
CITY_OF_NEW_ORLEANS	... GUTHERIE, ARLO	1	6	0	6	CITY_O
CITY_TO_CITY	... RAFFERTTY, GERRY	6	2	5	0	
CLOSET_CHRONICLES	... KANSAS	2	2	1	1	L
COCAINE	... CALE, J.J	2	3	1	2	OC
COLD_AS_ICE	... FOREIGNER	3	6	2	4	LD_A
COLD_WIND_TO_WALHALLA	... JETHRO_TULL	6	4	5	0	
COLORADO	... STILLS, STEPHEN	4	5	3	2	OR
COLOURS	... DONOVAN	5	3	4	0	
COME_INSIDE	... COMMODORES	3	13	2	11	ME_INSIDE__
COME_INSIDE_OF_MY_GUITAR	... BELLAMY_BROTHERS	13	6	12	0	
COME_ON_OVER	... BEE_GEES	6	6	5	1	O
COME_TOGETHER	... BEATLES	6	4	5	0	
COMING_INTO_LOS_ANGELES	... GUTHERIE, ARLO	4	4	3	1	I
COMMOTION	... CCR	4	4	3	1	M
COMPARED_TO_WHAT?	... FLACK, ROBERTA	4	3	3	0	
CONCLUSION	... ELP	3	4	2	2	NC
CONFUSION	... PROCOL_HARUM	4	1	3	0	

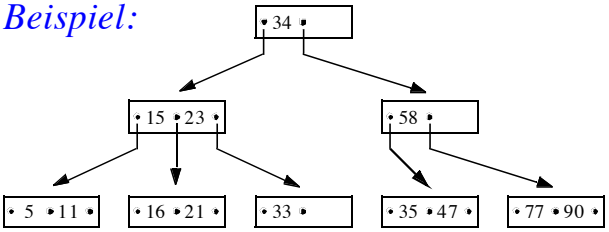
2-3-Bäume

- B-Bäume können auch als Hauptspeicher-Datenstruktur verwendet werden
  - möglichst kleine Knoten wichtiger als hohes Fan-Out
  - 2-3 Bäume: B-Bäume der Klasse $\tau(1,h)$, d.h. mit minimalen Knoten

  ⇒ Es gelten alle für den B-Baum entwickelten Such- und Modifikationsalgorithmen

- Ein 2-3-Baum ist ein m-Wege-Suchbaum (m=3), der entweder leer ist oder die Höhe $h \geq 1$ hat und folgende Eigenschaften besitzt:
  - Alle Knoten haben einen oder zwei Einträge (Schlüssel).
  - Alle Knoten außer den Blattknoten besitzen 2 oder 3 Söhne.
  - Alle Blattknoten sind auf derselben Stufe.

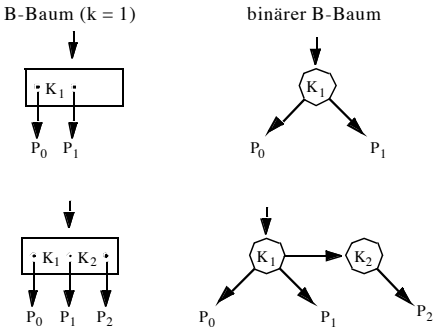
Beispiel:

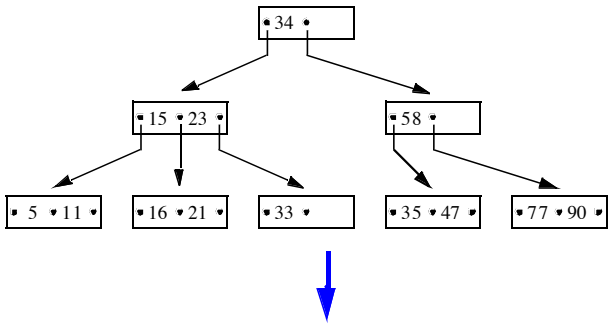


- Beobachtungen
  - 2-3-Baum ist balancierter Baum
  - ähnliche Laufzeitkomplexität wie AVL-Baum
  - schlechte Speicherplatznutzung (besonders nach Höhenänderung)

Binäre B-Bäume

- Verbesserte Speicherplatznutzung gegenüber 2-3-Bäumen durch Speicherung der Knoten als gekettete Listen mit einem oder zwei Elementen:

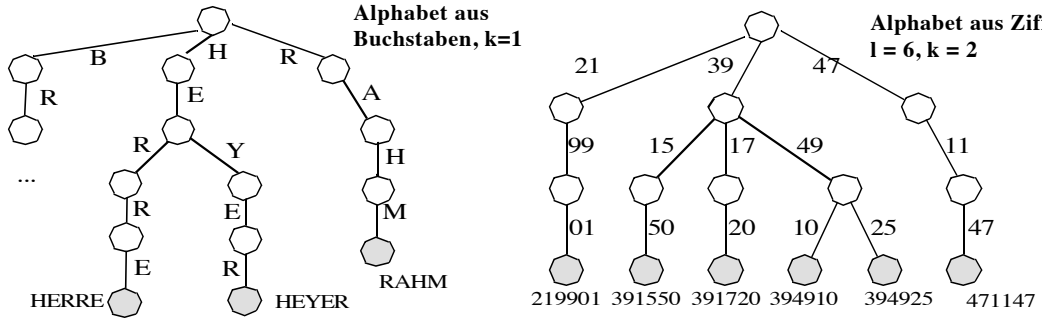




- Variante: symmetrischer binärer B-Baum

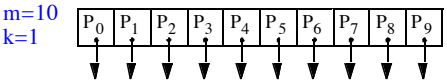
Digitale Suchbäume

- Prinzip des digitaler Suchbäume (kurz: Digitalbäume)
  - Zerlegung des Schlüssels - bestehend aus Zeichen eines Alphabets - in Teile
  - Aufbau des Baumes nach Schlüsselteilen
  - Suche im Baum durch Vergleich von Schlüsselteilen
  - jede unterschiedliche Folge von Teilschlüsseln ergibt eigenen Suchweg im Baum
  - alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg
  - vorteilhaft u.a. bei variabel langen Schlüsseln, z.B. Strings
- Was sind Schlüsselteile ?
  - Schlüsselteile können gebildet werden durch Elemente (Bits, Ziffern, Zeichen) eines Alphabets oder durch Zusammenfassungen dieser Grundelemente (z. B. Silben der Länge k)
  - Höhe des Baumes = l/k + 1, wenn l die max. Schlüssellänge und k die Schlüsselteillänge ist



m-ärer Trie

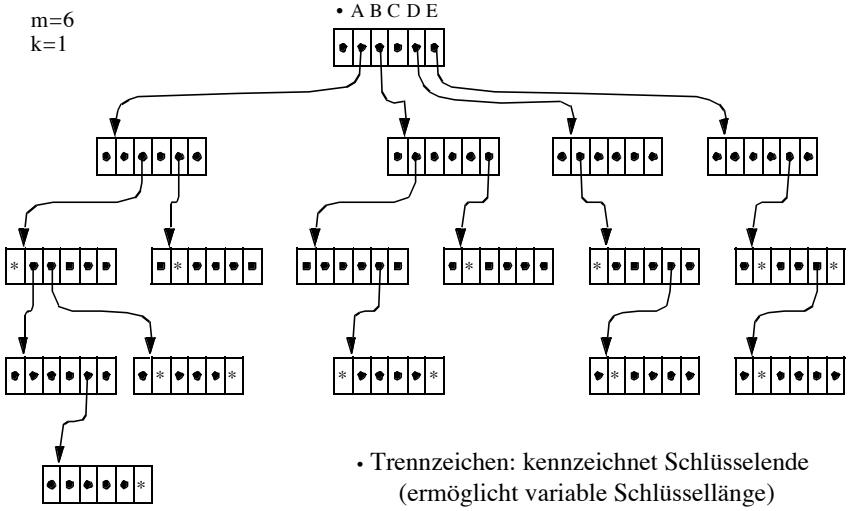
- Spezielle Implementierung des Digitalbaumes: Trie
  - Trie leitet sich von Information Retrieval ab (E.Fredkin, 1960)
- spezielle m-Wege-Bäume, wobei Kardinalität des Alphabets und Länge k der Schlüsselteile den Grad m festlegen
  - bei Ziffern: m = 10
  - bei Alpha-Zeichen: m = 26; bei alphanumerischen Zeichen: m = 36
  - bei Schlüsselteilen der Länge k potenziert sich Grad entsprechend, d. h. als Grad ergibt sich m^k
- Trie-Darstellung
  - Jeder Knoten eines Tries vom Grad m ist im Prinzip ein eindimensionaler Vektor mit m Zeigern
  - Jedes Element im Vektor ist einem Zeichen (bzw. Zeichenkombination) zugeordnet. Auf diese Weise wird ein Schlüsselteil (Kante) implizit durch die Vektorposition ausgedrückt.
  - Beispiel: Knoten eines 10-ären Trie mit Ziffern als Schlüsselteilen



  - implizite Zuordnung von Ziffer/Zeichen zu Zeiger. P_i gehört also zur Ziffer i. Tritt Ziffer i in der betreffenden Position auf, so verweist P_i auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist P_i mit NULL belegt
  - Wenn der Knoten auf der j-ten Stufe eines 10-ären Trie liegt, dann zeigt P_i auf einen Unterbaum, der nur Schlüssel enthält, die in der j-ten Position die Ziffer i besitzen

Grundoperationen

Beispiel: Trie für Schlüssel aus einem auf A-E beschränkten Alphabet



- Direkte Suche: In der Wurzel wird nach dem 1. Zeichen des Suchschlüssels verglichen. Bei Gleichheit wird der zugehörige Zeiger verfolgt. Im gefundenen Knoten wird nach dem 2. Zeichen verglichen usw.
  - Aufwand bei erfolgreicher Suche: l_i/k (+ 1 bei Präfix)
  - effiziente Bestimmung der Abwesenheit eines Schlüssels (z. B. CAD)
- Einfügen: Wenn Suchpfad schon vorhanden, wird NULL-Zeiger in *-Zeiger umgewandelt, sonst Einfügen von neuen Knoten (z. B. CAD)
- Löschen: Nach Aufsuchen des richtigen Knotens wird ein *-Zeiger auf NULL gesetzt. Besitzt daraufhin der Knoten nur NULL-Zeiger, wird er aus dem Baum entfernt (rekursive Überprüfung der Vorgängerknoten
- Sequentielle Suche ?

m-ärer Trie (3)

Beobachtungen:

- Höhe des Trie wird durch den längsten abgespeicherten Schlüssel bestimmt
- Gestalt des Baumes hängt von der Schlüsselmenge, also von der Verteilung der Schlüssel, nicht aber von der Reihenfolge ihrer Abspeicherung ab
- Knoten, die nur NULL-Zeiger besitzen, werden nicht angelegt

dennoch schlechte Speicherplatzausnutzung

- dünn besetzte Knoten
- viele Einweg-Verzweigungen (v.a. in der Nähe der Blätter)

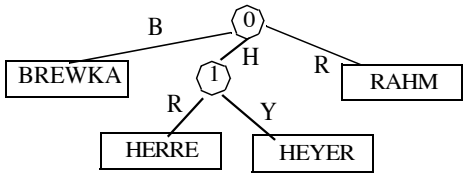
Möglichkeiten der Kompression

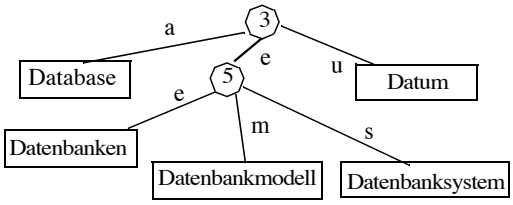
- Sobald ein Zeiger auf einen Unterbaum mit nur einem Schlüssel verweist, wird der (Rest-) Schlüssel in einem speziellen Knotenformat aufgenommen und Unterbaum eingespart
  -> vermeidet Einweg-Verzweigungen
- nur besetzte Verweise werden gespeichert (erfordert Angabe des zugehörigen Schlüsselteils)

PATRICIA-Baum

(Practical Algorithm To Retrieve Information Coded In Alphanumeric)

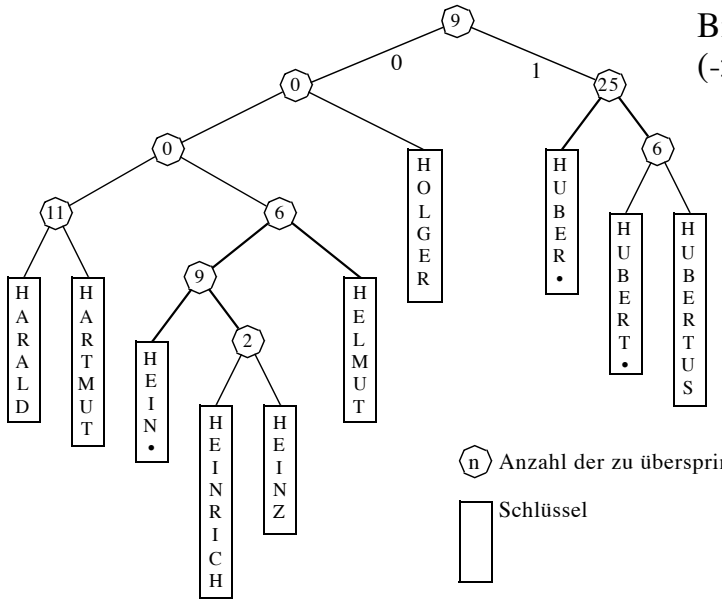
- Merkmale
  - Binärdarstellung für Schlüsselwerte -> binärer Digitalbaum
  - Speicherung der Schlüssel in den Blättern
  - **innere Knoten** speichern, wieviele Zeichen (Bits) beim Test zur Wegeauswahl zu überspringen sind
  - Vermeidung von Einwegverzweigungen, in dem bei nur noch einem verbleibenden Schlüssel direkt auf entsprechendes Blatt verwiesen wird





- Bewertung
  - speichereffizient
  - sehr gut geeignet für variabel lange Schlüssel und (sehr lange) Binärdarstellungen von Schlüsselwerten
  - bei jedem Suchschlüssel muß die Testfolge von der Wurzel beginnend ganz ausgeführt werden, bevor über Erfolg oder Mißerfolg der Suche entschieden werden kann

PATRICIA-Baum (2)



Binärdarstellung
(-> binärer Digitalbaum)

HANS = 1001000...
HEINZ = 1001000...
HOLGER= 1001000...
Bert = 1000010...
...
OTTO = 1001111...
...

(n) Anzahl der zu überspringenden Bits

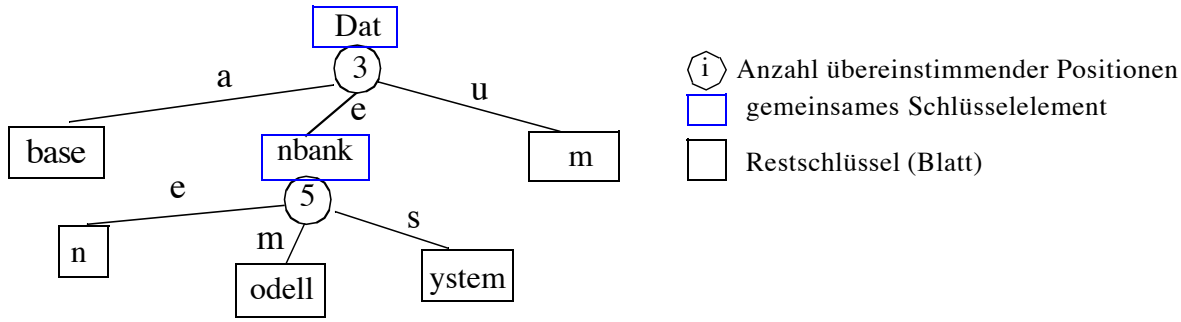
Schlüssel

- Suche nach dem Schlüssel HEINZ = X'10010001000101100100110011101011010' ?
- Suche nach ABEL = X'10000011000010100010110011100' ?

⇒ erfolgreiche und erfolglose Suche endet in einem Blattknoten

Präfix- bzw. Radix-Baum

- (Binärer) Digitalbaum als Variante des PATRICIA-Baumes
  - Speicherung variabel langer Schlüsselteile in den inneren Knoten, sobald sie sich als Präfixe für die Schlüssel des zugehörigen Unterbaums abspalten lassen
  - komplexere Knotenformate und aufwendigere Such- und Aktualisierungsoperationen
  - erfolglose Suche läßt sich oft schon in einem inneren Knoten abbrechen



Zusammenfassung

- Konzept des Mehrwegbaumes:
  - Aufbau sehr breiter Bäume von geringer Höhe
  - Bezugsgröße: Seite als Transporteinheit zum Externspeicher
  - Seiten werden immer größer, d. h., das Fan-out wächst weiter
- B- und B*-Baum gewährleisten eine balancierte Struktur
  - unabhängig von Schlüsselmenge
  - unabhängig ihrer Einfügereihenfolge
- Wichtigste Unterschiede des B*-Baums zum B-Baum:
  - strikte Trennung zwischen Datenteil und Indexteil. Datenelemente stehen nur in den Blättern des B*-Baumes
  - Schlüssel innerer Knoten haben nur Wegweiserfunktion. Sie können auch durch beliebige Trenner ersetzt oder durch Komprimierungsalgorithmen verkürzt werden
  - kürzere Schlüssel oder Trenner in den inneren Knoten erhöhen Verzweigungsgrad des Baumes und verringern damit seine Höhe
  - die redundant gespeicherten Schlüssel erhöhen den Speicherplatzbedarf nur geringfügig (< 1%)
  - Löschalgorithmus ist einfacher
  - Verkettung der Blattseiten ergibt schnellere sequentielle Verarbeitung

Zusammenfassung (2)

- Standard-Zugriffspfadstruktur in DBS: B*-Baum
- verallgemeinerte Überlaufbehandlung verbessert Seitenbelegung
- Schlüsselkomprimierung
  - Verbesserung der Baumbreite
  - Präfix-Suffix-Komprimierung sehr effektiv
  - Schlüssellängen von 20-40 Bytes werden im Mittel auf 1.3-1.8 Bytes reduziert
- Binäre B-Bäume: Alternative zu AVL-Bäumen als Hauptspeicher-Datenstruktur
- Digitale Suchbäume: Verwendung von Schlüsselteilen
  - Unterstützung von Suchvorgängen u.a. bei langen Schlüsseln variabler Länge
  - wesentliche Realisierungen: PATRICIA-Baum / Radix-Baum

2. Hashing

- Einführung
- Hash-Funktionen
  - Divisionsrest-Verfahren
  - Faltung
  - Mid-Square-Methode, . . .
- Behandlung von Kollisionen
  - Verkettung der Überläufer
  - Offene Hash-Verfahren: lineares Sondieren, quadratisches Sondieren, ...
- Analyse des Hashing
- Hashing auf Externspeichern
  - Bucket-Adressierung mit separaten Überlauf-Buckets
  - Analyse
- Dynamische Hash-Verfahren
  - Erweiterbares Hashing
  - Lineares Hashing

Einführung

- Gibt es bessere Strukturen für direkte Suche für Haupt- und Externspeicher ?
  - AVL-Baum: $O(\log_2 n)$ Vergleiche
  - B*-Baum: E/A-Kosten $O(\log_{k^*}(n))$, vielfach 3 Zugriffe
- Bisher:
  - Suche über Schlüsselvergleich
  - Allokation des Satzes als physischer Nachbar des "Vorgängers" oder beliebige Allokation und Verküpfung durch Zeiger
- Gestreute Speicherungsstrukturen / Hashing
  (Schlüsseltransformation, Adreßberechnungsverfahren, scatter-storage technique usw.)
  - Berechnung der Satzadresse SA(i) aus Satzschlüssel K_i --> **Schlüsseltransformation**
  - Speicherung des Satzes bei SA (i)
  - Ziele: schnelle direkte Suche + Gleichverteilung der Sätze (möglichst wenig Synonyme)

Einführung (2)

- Definition:

  S sei Menge aller möglichen Schlüsselwerte eines Satztyps (Schlüsselraum)

  A = {0,1,...,m-1} sei Intervall der ganzen Zahlen von 0 bis m-1 zur Adressierung eines Arrays bzw. einer Hash-Tabelle mit m Einträgen

  Eine Hash-Funktion $h : S \rightarrow A$

  ordnet jedem Schlüssel $s \in S$ des Satztyps eine Zahl aus A als Adresse in der Hash-Tabelle zu.

- Idealfall:
  1 Zugriff zur direkten Suche

- Problem: Kollisionen

Beispiel: m=10

h (s) = s mod 100

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

Perfektes Hashing: Direkte Adressierung

- Idealfall (perfektes Hashing): keine Kollisionen
  - h ist eine injektive Funktion.
  - Für jeden Schlüssel aus S muß Speicherplatz bereitgehalten werden, d. h., die Menge aller möglichen Schlüssel ist bekannt.

- Parameter

  l = Schlüssellänge, b = Basis, m = #Speicherplätze

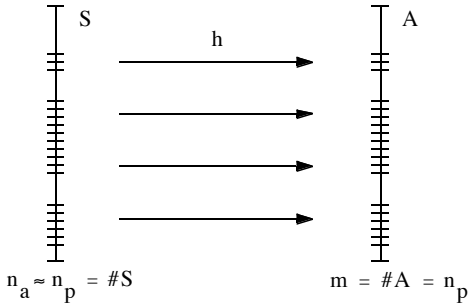
  n_p = #S = b^l mögliche Schlüssel

  n_a = #K = # vorhandene Schlüssel

  Wenn K bekannt ist und K fest bleibt, kann leicht eine injektive Abbildung

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

  z. B. wie folgt berechnet werden:



  - Die Schlüssel in K werden lexikographisch geordnet und auf ihre Ordnungsnummern abgebildet oder
  - Der Wert eines Schlüssels K_i oder eine einfache ordnungserhaltende Transformation dieses Wertes (Division/Multiplikation mit einer Konstanten) ergibt die Adresse: $A_i = h(K_i) = K_i$

Direkte Adressierung (2)

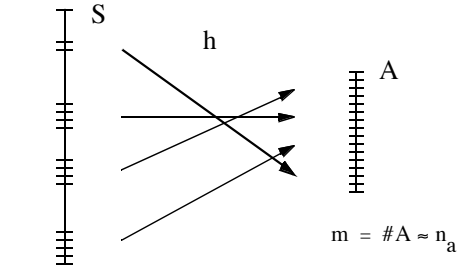
- Beispiel: Schlüsselmenge {00, . . . , 99}
- Eigenschaften
  - Statische Zuordnung des Speicherplatzes
  - Kosten für direkte Suche und Wartung ?
  - Reihenfolge beim sequentiellen Durchlauf ?

	Schlüssel	Daten
00		
01	01	D01
02	02	D02
03		
04	04	D04
05	05	D05
⋮		
95		
96	96	D96
97		
98		
99	99	D99

- Bestes Verfahren bei geeigneter Schlüsselmenge K, aber aktuelle Schlüsselmenge K ist oft nicht "dicht":
  - eine 9-stellige Sozialversicherungsnummer bei 10^5 Beschäftigten
  - Namen / Bezeichner als Schlüssel (Schlüssellänge k):

Allgemeines Hashing

- Annahmen
  - Die Menge der möglichen Schlüssel ist meist sehr viel größer als die Menge der verfügbaren Speicheradressen
  - h ist nicht injektiv
- Definitionen:
  - Zwei Schlüssel K_i, $K_j \in K$ kollidieren (bzgl. einer Hash-Funktion h) gdw. $h(K_i) = h(K_j)$.
  - Tritt für K_i und K_j eine Kollision auf, so heißen diese Schlüssel Synonyme.
  - Die Menge der Synonyme bezüglich einer Speicheradresse A_i heißt Kollisionsklasse.



- Geburtstags-Paradoxon

k Personen auf einer Party haben gleichverteilte und stochastisch unabhängige Geburtstage.
Mit welcher Wahrscheinlichkeit p (n, k) haben mindestens 2 von k Personen am gleichen Tag (n = 365) Geburtstag?

Die Wahrscheinlichkeit, daß keine Kollision auftritt, ist

$$q(n,k) = \frac{\text{Zahl der günstigen Fälle}}{\text{Zahl der möglichen Fälle}} = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k}{n} = \frac{(n-1) \cdot \ldots \cdot (n-k)}{n^k}$$

Es ist $p(365, k) = 1 - q(365, k) > 0.5$ für k

- Behandlung von Kollisionen erforderlich !

Hash-Verfahren: Einflußfaktoren

- Leistungsfähigkeit eines Hash-Verfahrens: Einflußgrößen und Parameter
  - Hash-Funktion
  - Datentyp des Schlüsselraumes: Integer, String, ...
  - Verteilung der aktuell benutzten Schlüssel
  - Belegungsgrad der Hash-Tabelle HT
  - Anzahl der Sätze, die sich auf einer Adresse speichern lassen, ohne Kollision auszulösen (Bucket-Kapazität)
  - Technik zur Kollisionsauflösung
  - ggf. Reihenfolge der Speicherung der Sätze (auf Hausadresse zuerst!)
- Belegungsfaktor der Hash-Tabelle
  - Verhältnis von aktuell belegten zur gesamten Anzahl an Speicherplätzen $\beta = n_a / m$
  - für $\beta \geq 0{,}85$ erzeugen alle Hash-Funktionen viele Kollisionen und damit hohen Zusatzaufwand
  - Hash-Tabelle ausreichend groß zu dimensionenieren ($m > n_a$)
- Für die Hash-Funktion h gelten **folgende Forderungen:**
  - Sie soll sich einfach und effizient berechnen lassen (konstante Kosten)
  - Sie soll eine möglichst gleichmäßige Belegung der Hash-Tabelle HT erzeugen, auch bei ungleich verteilten Schlüsseln
  - Sie soll möglichst wenige Kollisionen verursachen

Hash-Funktionen (2)

1. Divisionsrest-Verfahren (kurz: Divisions-Verfahren): $h(K_i) = K_i \bmod q, \quad (q \sim m)$

⇒ Der entstehende Rest ergibt die relative Adresse in HT

- Beispiel:

Die Funktion nat wandle Namen in natürliche Zahlen um:
nat(Name) = ord (1. Buchstabe von Name) - ord ('A')

h (Name) = nat (Name) mod m

HT: m=10

	Schlüssel	Daten
0		
1	BOHR	D1
2	CURIE	D2
3	DIRAC	D3
4	EINSTEIN	D4
5	PLANCK	D5
6		
7	HEISENBERG	D7
8	SCHRÖDINGER	D8
9		

- Wichtigste Forderung an Divisor q:
  q = Primzahl (größte Primzahl <= m)
  - Hash-Funktion muß etwaige Regelmäßigkeiten in Schlüsselverteilung eliminieren, damit nicht ständig die gleichen Plätze in HT getroffen werden
  - Bei äquidistantem Abstand der Schlüssel $K_i + j \cdot \Delta K$, $j = 0, 1, 2, \ldots$ maximiert eine Primzahl die Distanz, nach der eine Kollision auftritt. Eine Kollision ergibt sich, wenn

    $K_i \bmod q = (K_i + j \cdot \Delta K) \bmod q \quad \text{oder} \quad j \cdot \Delta K = k \cdot q, \; k = 1, 2, 3, \ldots$

  - Eine Primzahl kann keine gemeinsamen Faktoren mit ΔK besitzen, die den Kollisionsabstand verkürzen würden

Hash-Funktionen (3)

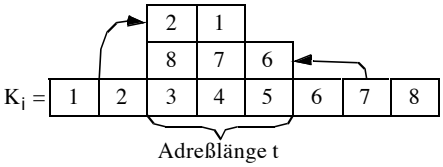
2. Faltung

- Schlüssel wird in Teile zerlegt, die bis auf das letzte die Länge einer Adresse für HT besitzen
- Schlüsselteile werden dann übereinandergefaltet und addiert.

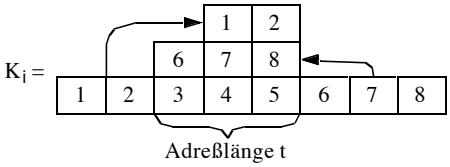
■ Variationen:

- Rand-Falten: wie beim Falten von Papier am Rand
- Shift-Falten: Teile des Schlüssels werden übereinandergeschoben
- Sonstige: z.B. EXOR-Verknüpfung bei binärer Zeichendarstellung
- Beispiel: b = 10, t = 3, m = 10^3

Rand-Falten



Shift-Falten



■ Faltung

- verkürzt lange Schlüssel auf "leicht berechenbare" Argumente, wobei alle Schlüsselteile Beitrag zur Adreßberechnung liefern
- diese Argumente können dann zur Verbesserung der Gleichverteilung mit einem weiteren Verfahren "gehasht" werden

Hash-Funktionen (4)

3. Mid-Square-Methode

- Schlüssel K_i wird quadriert. t aufeinanderfolgende Stellen werden aus der Mitte des Ergebnisses für die Adressierung ausgewählt.
- Es muß also $b^t = m$ gelten.
- mittlere Stellen lassen beste Gleichverteilung der Werte erwarten
- Beispiel für b = 2, t = 4, m = 16: $K_i = 1100100$ $K_i^2 = 10011\underbrace{1000}_{t}10000 \rightarrow h(K_i) = 1000$

4. Zufallsmethode:

- K_i dient als Saat für Zufallszahlengenerator

5. Ziffernanalyse:

- setzt Kenntnis der Schlüsselmenge K voraus. Die t Stellen mit der besten Gleichverteilung der Ziffern oder Zeichen in K werden von K_i zur Adressierung ausgewählt

Hash-Funktionen: Bewertung

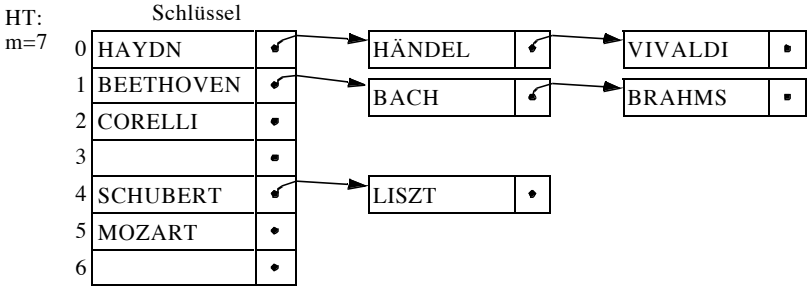
- Verhalten / Leistungsfähigkeit einer Hash-Funktion hängt von der gewählten Schlüsselmenge ab
  - Deshalb lassen sie sich auch nur unzureichend theoretisch oder mit Hilfe von analytischen Modellen untersuchen
  - Wenn eine Hash-Funktion gegeben ist, läßt sich immer eine Schlüsselmenge finden, bei der sie **besonders viele Kollisionen** erzeugt
  - **Keine Hash-Funktion** ist immer besser als alle anderen
- Über die Güte der verschiedenen Hash-Funktionen liegen jedoch eine Reihe von empirischen Untersuchungen vor
  - Das **Divisionsrest-Verfahren** ist im Mittel am leistungsfähigsten; für bestimmte Schlüsselmengen können jedoch andere Techniken besser abschneiden
  - Wenn die Schlüsselverteilung nicht bekannt ist, dann ist das Divisionsrest-Verfahren die bevorzugte Hash-Technik
  - Wichtig dabei: ausreichend große Hash-Tabelle, Verwendung einer Primzahl als Divisor

Behandlung von Kollisionen

- Zwei Ansätze, wenn $h(K_q) = h(K_p)$
  - K_p wird in einem separaten Überlaufbereich (außerhalb der Hash-Tabelle) zusammen mit allen anderen Überläufern gespeichert; Verkettung der Überläufer
  - Es wird für K_p ein freier Platz innerhalb der Hash-Tabelle gesucht („Sondieren“); alle Überläufer werden im Primärbereich untergebracht („offene Hash-Verfahren“)
- Methode der Kollisionsauflösung entscheidet darüber, welche Folge und wieviele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden
- Adreßfolge bei Speicherung und Suche für Schlüssel K_p sei $h_0(K_p), h_1(K_p), h_2(K_p), \ldots$
  - Bei einer Folge der Länge n treten also n-1 Kollisionen auf
  - **Primärkollision:** $h(K_p) = h(K_q)$
  - **Sekundärkollision:** $h_i(K_p) = h_j(K_q)\ ,\ i \neq j$

Hash-Verfahren mit Verkettung der Überläufer (separater Überlaufbereich)

- Dynamische Speicherplatzbelegung für Synonyme
  - Alle Sätze, die nicht auf ihrer Hausadresse unterkommen, werden in einem separaten Bereich gespeichert (Überlaufbereich)
  - Verkettung der Synonyme (Überläufer) pro Hash-Klasse
  - Suchen, Einfügen und Löschen sind auf Kollisionsklasse beschränkt
  - Unterscheidung nach Primär- und Sekundärbereich: n > m ist möglich !



- Entartung zur linearen Liste prinzipiell möglich
- Nachteil: Anlegen von Überläufern, auch wenn Hash-Tabelle (Primärbereich) noch wenig belegt ist

Java-Realisierung

```
/** Einfacher Eintrag in Hash-Tabelle */
class HTEntry {
  Object key;
  Object value;
  /** Konstruktor */
  HTEntry (Object key, Object value) {
    this.key = key; this.value = value; } }

/** Abstrakte Basisklasse für Hash-Tabellen */
public abstract class HashTable {
  protected HTEntry[] table;
  /** Konstruktor */
  public HashTable (int capacity) { table = new HTEntry[capacity]; }
  /** Die Hash-Funktion */
  protected int h(Object key) {
    return (key.hashCode() & 0x7fffffff) % table.length; }
  /** Einfuegen eines Schluessel-Wert-Paares */
  public abstract boolean add(Object key, Object value);
  /** Test ob Schluessel enthalten ist */
  public abstract boolean contains(Object key);
  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public abstract Object get(Object key);
  /** Entfernen eines Eintrags */
  public abstract void remove(Object key); }
```

```
/** Eintrag in Hash-Tabelle mit Zeiger für verkettete Ueberlaufbehandlung */
class HTLinkedEntry extends HTEntry {
  HTLinkedEntry next;
  /** Konstruktor */
  HTLinkedEntry (Object key, Object value) { super(key, value); } }

/** Hash-Tabelle mit separater (verketteter) Ueberlaufbehandlung */
public class LinkedHashTable extends HashTable {

  /** Konstruktor */
  public LinkedHashTable (int capacity) { super(capacity); }

  /** Einfuegen eines Schluessel-Wert-Paares */
  public boolean add(Object key, Object value) {
    int pos = h(key);        // Adresse in Hash-Tabelle fuer Schluessel
    if (table[pos] == null)  // Eintrag frei?
      table[pos] = new HTLinkedEntry(key, value);
    else {                   // Eintrag belegt -> Suche Eintrag in Kette
      HTLinkedEntry entry = (HTLinkedEntry) table[pos];
      while((entry.next != null) && (! entry.key.equals(key)))
        entry = entry.next;
      if (entry.key.equals(key)) // Schluessel existiert schon
        entry.value = value;
      else                       // fuege neuen Eintrag am Kettenende an
        entry.next = new HTLinkedEntry(key, value); }
    return true; }
```



```
  /** Test ob Schluessel enthalten ist */
  public boolean contains(Object key) {
    HTLinkedEntry entry = (HTLinkedEntry) table[h(key)];
    while((entry != null) && (! entry.key.equals(key)))
      entry = entry.next;
    return entry != null;
  }

  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public Object get(Object key) {
    HTLinkedEntry entry = (HTLinkedEntry) table[h(key)];
    while((entry != null) && (! entry.key.equals(key)))
      entry = entry.next;
    if (entry != null)
      return entry.value;
    return null;
  }
  ...
}
```


Offene Hash-Verfahren: Lineares Sondieren

- Offene Hash-Verfahren
  - Speicherung der Synomyme (Überläufer) im Primärbereich
  - Hash-Verfahren muß in der Lage sein, eine Sondierungsfolge, d.h. eine Permutation aller Hash-Adressen, zu berechnen
- Lineares Sondieren (linear probing)

Von der Hausadresse (Hash-Funktion h) aus wird sequentiell (modulo der Hash-Tabellen-Größe) gesucht. Offensichtlich werden dabei alle Plätze in HT erreicht:

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + i) \bmod m \quad , i = 1, 2, \ldots$$

m=10, h (K) = K mod m

	Schlüssel
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Beispiel: Einfügereihenfolge 79, 28, 49, 88, 59
  - Häufung von Kollisionen durch „Klumpenbildung“

⇒ lange Sondierungsfolgen möglich

Java-Realisierung

- Suche in einer Hash-Tabelle bei linearem Sondieren

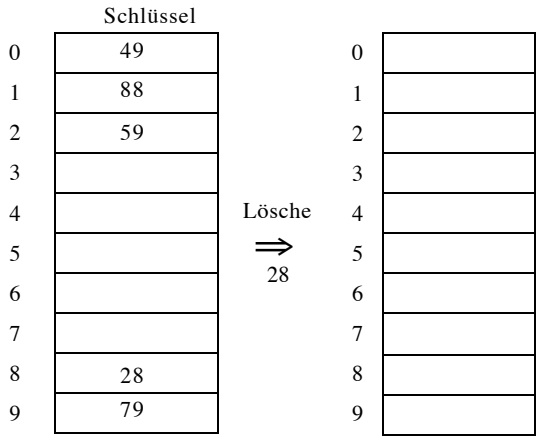
```
/** Hash-Tabelle mit Ueberlaufbehandlung im Primaerbereich (Sondieren) */
public class OpenHashTable extends HashTable {

  protected static final int EMPTY = 0;      // Eintrag ist leer
  protected static final int OCCUPIED = 1;   // Eintrag belegt
  protected static final int DELETED = 2;    // Eintrag geloescht

  protected int[] flag;  // Markierungsfeld; enthaelt Eintragsstatus

  /** Konstruktor */
  public OpenHashTable (int capacity) {
    super(capacity);
    flag = new int[capacity];
    for (int i=0; i<capacity; i++)  // initialisiere Markierungsfeld
      flag[i] = EMPTY;
  }

  /** (Lineares) Sondieren. Berechnet aus aktueller Position die naechste.*/
  protected int s(int pos) {
    return ++pos % table.length;
  }
```

```
  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public Object get(Object key) {
    int pos, startPos;
    startPos = pos = h(key);  // Adresse in Hash-Tabelle fuer Schluessel
    while((flag[pos] != EMPTY) && (! table[pos].key.equals(key))) {
      pos = s(pos);                          // ermittle naechste Position
      if (pos == startPos) return null;   // Eintrag nicht gefunden
    }
    if (flag[pos] == OCCUPIED)
      // Schleife verlassen, da Schluessel gefunden; Eintrag als belegt
      // markiert
      return table[pos].value;
    // Schleife verlassen, da Eintrag leer oder
    // Eintrag gefunden, jedoch als geloescht markiert
    return null;
  }
  ...
}
```


Lineares Sondieren (2)

- Aufwendiges Löschen
  - impliziert oft Verschiebungen
  - entstehende Lücken in Suchsequenzen sind aufzufüllen, da das Antreffen eines freien Platzes die Suche beendet.

m=10, h (K) = K mod m



- Verbesserung: Modifikation der Überlauffolge

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_{i-1}(K_p) + f(i)) \bmod m \quad \text{oder}$$

$$h_i(K_p) = (h_{i-1}(K_p) + f(i, h(K_p))) \bmod m \quad , \quad i = 1, 2, \ldots$$

- Beispiele:
  - Weiterspringen um festes Inkrement c (statt nur 1): f (i) = c * i
  - Sondierung in beiden Richtungen: f (i) = c * i * $(-1)^i$

Quadratisches Sondieren

- Bestimmung der Speicheradresse

$$h_0(K_p) = h(K_p) \qquad h_i(K_p) = (h_0(K_p) + a \cdot i + b \cdot i^2) \bmod m \quad , \; i = 1, 2, \ldots$$

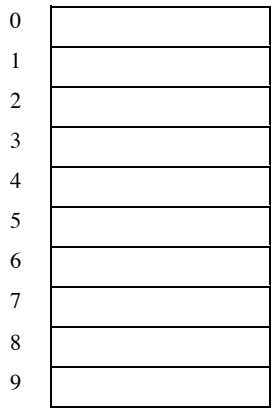
  - m sollte Primzahl sein

- Folgender **Spezialfall** sichert Erreichbarkeit aller Plätze:

$$h_0(K_p) = h(K_p) \qquad h_i(K_p) = \left(h_0(K_p) - \left(\left\lceil \frac{i}{2} \right\rceil\right)^2 (-1)^i\right) \bmod m \qquad 1 \le i \le m-1$$

- Beispiel:
  Einfügereihenfolge 79, 28, 49, 88, 59

m=10,
h (K) = K mod m



Weitere offene Hash-Verfahren

- Sondieren mit Zufallszahlen

  Mit Hilfe eines deterministischen Pseudozufallszahlen-Generators wird die Folge der Adressen [1 .. m-1] mod m genau einmal erzeugt:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + z_i) \bmod m \; , \; i = 1, 2, \ldots$$

- Double Hashing

  Einsatz einer zweiten Funktion für die Sondierungsfolge

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + i \cdot h'(K_p)) \bmod m \quad , \; i = 1, 2, \ldots$$

  Dabei ist h'(K) so zu wählen, daß für alle Schlüssel K die resultierende Sondierungsfolge eine Permutation aller Hash-Adressen bildet

- Kettung von Synonymen
  - explizite Kettung aller Sätze einer Kollisionsklasse
  - verringert nicht die Anzahl der Kollisionen; sie verkürzt jedoch den Suchpfad beim Aufsuchen eines Synonyms.
  - Bestimmung eines freien Überlaufplatzes (Kollisionsbehandlung) mit beliebiger Methode

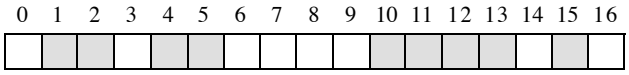
Analyse des Hashing

- Kostenmaße
  - β = n/m: Belegung von HT mit n Schlüsseln
  - S_n = # der Suchschritte für das Auffinden eines Schlüssels - entspricht den Kosten für erfolgreiche Suche und Löschen (ohne Reorganisation)
  - U_n = # der Suchschritte für die erfolglose Suche - das Auffinden des ersten freien Platzes entspricht den Einfügekosten
- Grenzwerte

best case:
$S_n = 1$
$U_n = 1$.

worst case:
$S_n = n$
$U_n = n+1$.

- Modell für das lineare Sondieren
  - Sobald β eine gewisse Größe überschreitet, verschlechtert sich das Zugriffsverhalten sehr stark.



  - Je länger eine Liste ist, umso schneller wird sie noch länger werden.
  - Zwei Listen können zusammenwachsen (Platz 3 und 14), so daß durch neue Schlüssel eine Art Verdopplung der Listenlänge eintreten kann

⇒ Ergebnisse für das lineare Sondieren nach Knuth:

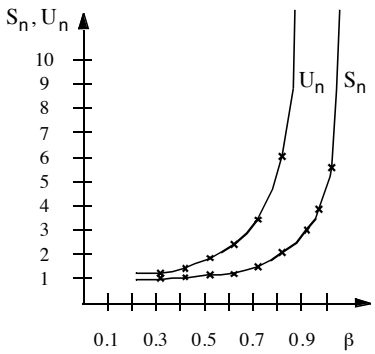
$$S_n \approx 0{,}5\left(1 + \frac{1}{1-\beta}\right) \quad \text{mit} \quad 0 \le \beta = \frac{n}{m} < 1 \qquad U_n \approx 0{,}5\left(1 + \frac{1}{(1-\beta)^2}\right)$$

Analyse des Hashing (2)

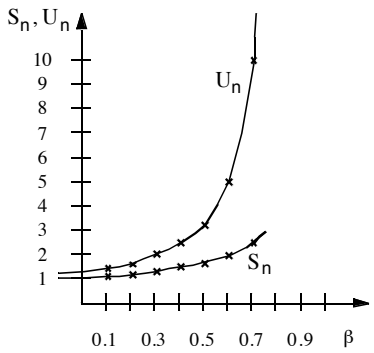
- Abschätzung für offene Hash-Verfahren mit optimierter Kollisionsbehandlung (gleichmäßige HT-Verteilung von Kollisionen)

$$S_n \sim -\frac{1}{\beta} \cdot \ln(1-\beta) \qquad U_n \sim \frac{1}{1-\beta}$$

- Anzahl der Suchschritte in HT



a) bei linearem Sondieren



a) bei "unabhängiger" Kollisionsauflösung

Analyse des Hashing (3)

- Modell für separate Überlaufbereiche
  - Annahme: n Schlüssel verteilen sich gleichförmig über die m mögl. Ketten.
  - Jede Synonymkette hat also im Mittel n/m = β Schlüssel
  - *Erfolgreiche Suche:* wenn der i-te Schlüssel K_i in HT eingefügt wird, sind in jeder Kette (i-1)/m Schlüssel. Die Suche nach K_i kostet also 1+(i-1)/m Schritte, da K_i an das jeweilige Ende einer Kette angehängt wird.
    Erwartungswert für erfolgreiche Suche: $S_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{n-1}{2 \cdot m} \approx 1 + \frac{\beta}{2}$
  - *Erfolglosen Suche:* es muß immer die ganze Kette durchlaufen werden

    $U_n = 1 + 1 \cdot$ WS (zu einer Hausadresse existiert 1 Überläufer)$+ 2 \cdot$ WS (zu Hausadresse existieren 2 Überläufer) + 3 ...

    $U_n \approx \beta - e^{-\beta}$.

β	0.5	0.75	1	1.5	2	3	4	5
S_n	1.25	1.37	1.5	1.75	2	2.5	3	3.5
U_n	1.11	1.22	1.37	1.72	2.14	3.05	4.02	5.01

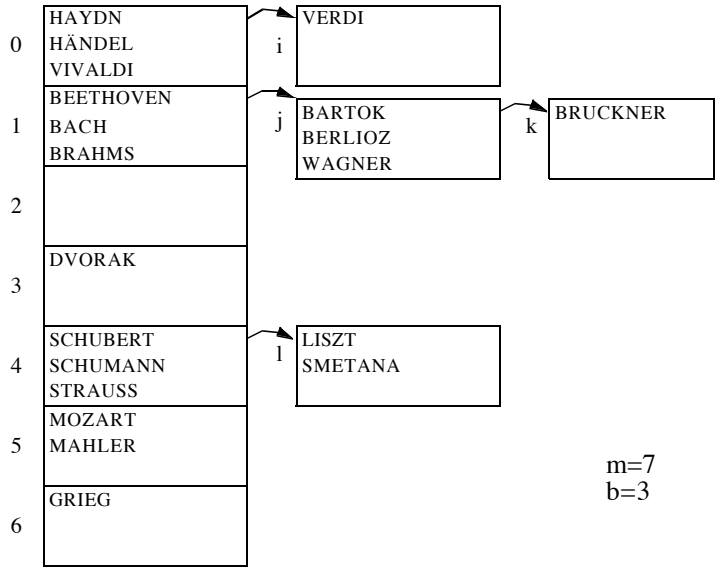
- Separate Kettung ist auch der “unabhängigen” Kollisionsauflösung überlegen
- Hashing ist i. a. sehr leistungsstark. Selbst bei starker Überbelegung (β>1) erhält man bei separater Kettung noch günstige Werte

Hashing auf Externspeichern

- Hash-Adresse bezeichnet Bucket (hier: Seite)
  - Kollisionsproblem wird entschärft, da mehr als ein Satz auf seiner Hausadresse gespeichert werden kann
  - Bucket-Kapazität b -> Primärbereich kann bis zu b*m Sätze aufnehmen !
- Überlaufbehandlung
  - Überlauf tritt erst beim (b+1)-ten Synonym auf
  - alle bekannten Verfahren sind möglich, aber lange Sondierungsfolgen im Primärbereich sollten vermieden werden
  - häufig Wahl eines separaten Überlaufbereichs mit dynamischer Zuordnung der Buckets
- Speicherungsreihenfolge im Bucket
  - ohne Ordnung (Einfügefolge)
  - nach der Sortierfolge des Schlüssels: aufwendiger, jedoch Vorteile beim Suchen (sortierte Liste!)
- Bucket-Größe meist Seitengröße (Alternative: mehrere Seiten / Spur einer Magnetplatte)
  - Zugriff auf die Hausadresse bedeutet 1 physische E/A
  - jeder Zugriff auf ein Überlauf-Bucket löst weiteren physischen E/A-Vorgang aus

Hashing auf Externspeichern (2)

- Bucket-Adressierung mit separaten Überlauf-Buckets
  - weithin eingesetztes Hash-Verfahren für Externspeicher
  - jede Kollisionsklasse hat eine separate Überlaufkette.



- Klassifikation

	Primär-Bucket	Überlauf-Bucket
inneres Bucket	0, 1, 4	j
Rand-Bucket	2, 3, 5, 6	i, k, l

Hashing auf Externspeichern (3)

- Grundoperationen
  - direkte Suche: nur in der Bucket-Kette
  - sequentielle Suche ?
  - Einfügen: ungeordnet oder sortiert
  - Löschen: keine Reorganisation in der Bucket-Kette - leere Überlauf-Buckets werden entfernt
- Kostenmodelle sehr komplex
- Belegungsfaktor:

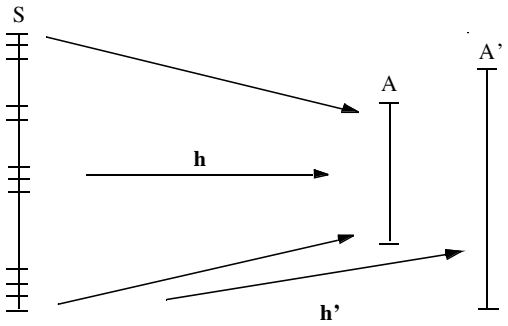
  $\beta = n/(b \cdot m)$

  - bezieht sich auf Primär-Buckets (**kann größer als 1 werden!**)
- **Zugriffsfaktoren**
  - Gute Annäherung an idealen Wert
  - Bei Vergleich mit Mehrwegbäumen ist zu beachten, daß Hash-Verfahren sortiert sequentielle Verarbeitung aller Sätze nicht unterstützen. Außerdem stellen sie statische Strukturen dar. Die Zahl der Primär-Buckets m läßt sich nicht dynamisch an die Zahl der zu speichernden Sätze n anpassen.

b		β = 0.5	0.75	1.0	1.25	1.5	1.75	2.0
b = 2	S_n	1.10	1.20	1.31	1.42	1.54	1.66	1.78
	U_n	1.08	1.21	1.38	1.58	1.79	2.02	2.26
b = 5	S_n	1.02	1.08	1.17	1.28	1.40	1.52	1.64
	U_n	1.04	1.17	1.39	1.64	1.90	2.15	2.40
b = 10	S_n	1.00	1.03	1.12	1.24	1.36	1.47	1.59
	U_n	1.01	1.13	1.41	1.72	1.96	2.19	2.44
b = 20	S_n	1.00	1.01	1.08	1.21	1.34	1.45	1.56
	U_n	1.00	1.08	1.44	1.81	1.99	2.17	2.45
b = 30	S_n	1.00	1.00	1.05	1.20	1.33	1.43	1.54
	U_n	1.00	1.02	1.46	1.93	2.00	2.08	2.47

Dynamische Hash-Verfahren

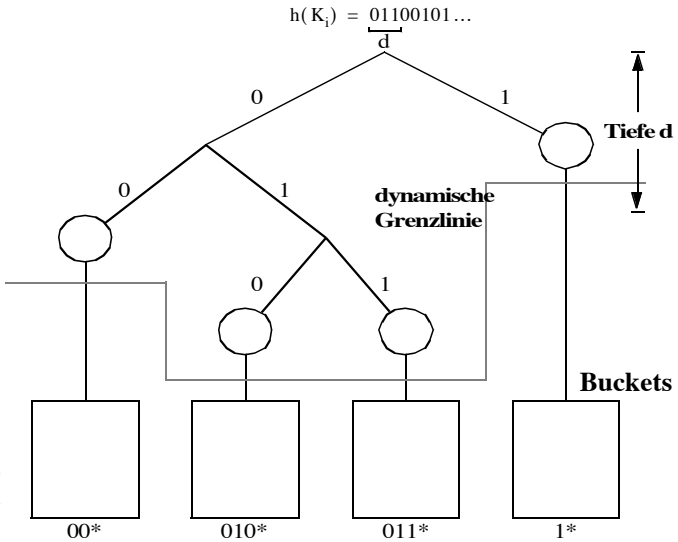
- Wachstumsproblem bei statischen Verfahren
  - Statische Allokation von Speicherbereichen: Speicherausnutzung?
  - Bei Erweiterung des Adreßraumes: Re-Hashing
    ⇒ Alle Sätze erhalten eine **neue Adresse**
  - Probleme: Kosten, Verfügbarkeit, Adressierbarkeit



- Entwurfsziele
  - Eine im Vergleich zum statischen Hashing dynamische Struktur , die Wachstum und Schrumpfung des Hash-Bereichs (Datei) erlaubt
  - Keine Überlauftechniken
  - Zugriffsfaktor ≤ 2 für die direkte Suche
- Viele konkurrierende Ansätze
  - Extendible Hashing (Fagin et al., 1978)
  - Virtual Hashing und Linear Hashing (Litwin, 1978, 1980)
  - Dynamic Hashing (Larson, 1978)

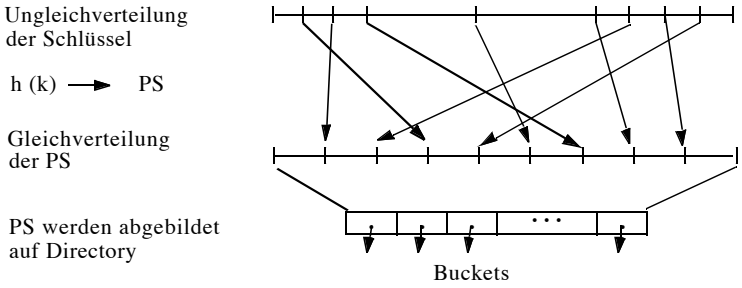
Erweiterbares Hashing

- Kombination mehrerer Ideen
  - Dynamik von B-Bäumen (Split- und Mischtechniken von Seiten) zur Konstruktion eines dynamischen Hash-Bereichs
  - Adressierungstechnik von Digitalbäumen zum Aufsuchen eines Speicherplatzes
  - Hashing: gestreute Speicherung mit möglichst gleichmäßiger Werteverteilung
- Prinzipielle Vorgehensweise
  - Die einzelnen Bits eines Schlüssels steuern der Reihe nach den Weg durch den zur Adressierung benutzten Digitalbaum $K_i = (b_0, b_1, b_2, \dots)$.
  - Verwendung der Schlüsselwerte kann bei Ungleichverteilung zu unausgewogenem Digitalbaum führen (Digitalbäume kennen keine Höhenbalancierung)
  - Verwendung von $h(K_i)$ als sog. Pseudoschlüssel (PS) soll bessere Gleichverteilung gewährleisten.
    $h(K_i) = (b_0, b_1, b_2, \dots)$
  - Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann



Erweiterbares Hashing (2)

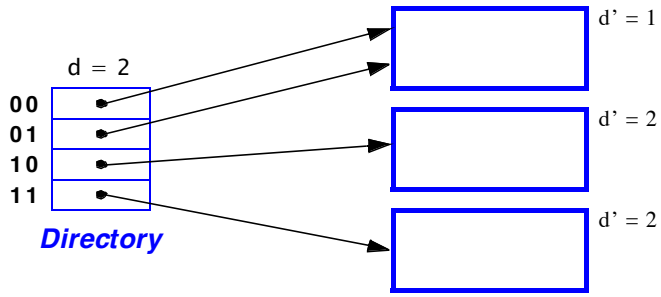
- Prinzipielle Abbildung der Pseudoschlüssel
  - Zur Adressierung eines Buckets sind d Bits erforderlich, wobei sich dafür i. a. eine dynamische Grenzlinie variierender Tiefe ergibt.
  - ausgeglichener Digitalbaum garantiert minimales d_{max}
  - Hash-Funktion soll möglichst Gleichverteilung der Pseudoschlüssel erreichen (minimale Höhe des Digitalbaumes, minimales d_{max})



- dynamisches Wachsen und Schrumpfen des Hash-Bereiches
  - Buckets werden erst bei Bedarf bereitgestellt:
    kein statisch dimensionierter Primärbereich, keine Überlauf-Buckets
  - nur belegte Buckets werden gespeichert
  - hohe Speicherplatzbelegung möglich

Erweiterbares Hashing (3)

- schneller Zugriff über *Directory* (Index)
  - binärer Digitalbaum der Höhe d wird durch einen Digitalbaum der Höhe 1 implementiert (entarteter Trie der Höhe 1 mit 2^d Einträgen).
  - d wird festgelegt durch den längsten Pfad im binären Digitalbaum.
  - In einem Bucket werden nur Sätze gespeichert, deren Pseudoschlüssel in den ersten d' Bits übereinstimmen (d' = lokale Tiefe).
  - d = MAX (d'): d Bits des PS werden zur Adressierung verwendet (d = globale Tiefe).
  - Directory enthält 2^d Einträge
  - alle Sätze zu einem Eintrag (d Bits) sind in einem Bucket gespeichert; wenn d' < d, können benachbarte Einträge auf dasselbe Bucket verweisen
  - max. 2 Seitenzugriffe

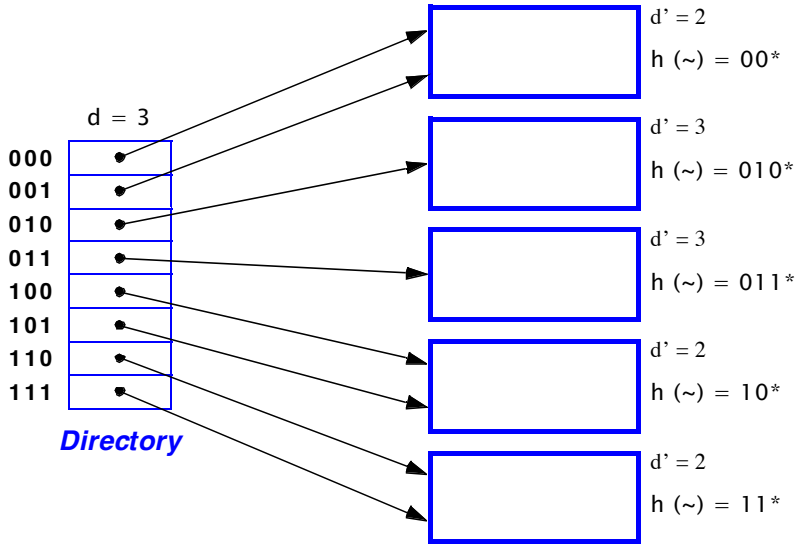


Erweiterbares Hashing: Splitting von Buckets

- Fall 1: Überlauf eines Buckets, dessen lokale Tiefe kleiner ist als globale Tiefe d

  ⇒ lokale Neuverteilung der Daten

  - Erhöhung der lokalen Tiefe
  - lokale Korrektur der Pointer im Directory

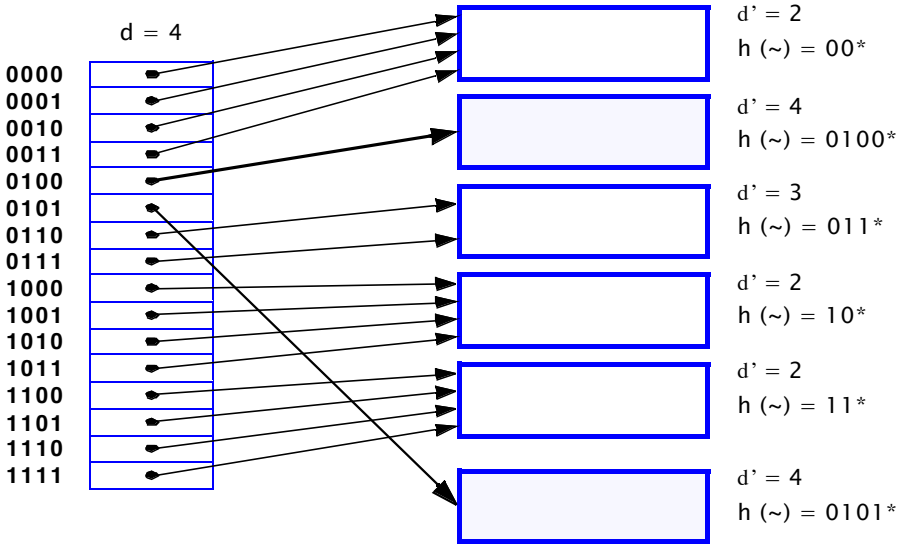


Erweiterbares Hashing: Splitting von Buckets (2)

- Fall 2: Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist

  ⇒ lokale Neuverteilung der Daten (Erhöhung der lokalen Tiefe)

  - Verdopplung des Directories (Erhöhung der globalen Tiefe)
  - globale Korrektur/Neuverteilung der Pointer im Directory



Lineares Hashing

- Dynamisches Wachsen/Schrumpfen des Hash-Bereiches ohne große Directories
  - inkrementelles Wachstum durch sukzessives Splitten von Buckets in fest vorgegebener Reihenfolge
  - Splitten erfolgt bei Überschreiten eines Belegungsfaktors ß (z.B. 80%)
  - Überlauf-Buckets sind notwendig
- Prinzipieller Ansatz
  - m: Ausgangsgröße des Hash-Bereiches (#Buckets)
  - sukzessives Neuanlegen einzelner Buckets am aktuellen Dateiende, falls Belegungsfaktor ß vorhandener Buckets einen Grenzwert übersteigt (Schrumpfen am aktuellen Ende bei Unterschreiten einer Mindestbelegung)
  - Adressierungsbereich verdoppelt sich bei starkem Wachstum gelegentlich, L=Anzahl vollständig erfolgter Verdoppelungen (Initialwert 0)
  - Größe des Hash-Bereiches: $m * 2^L$
  - Split-Zeiger p (Initialwert 0) zeigt auf nächstes zu splittende Bucket im Hash-Bereich mit $0 <= p < m*2^L$
  - Split führt zu neuem Bucket mit Adresse $p + m*2^L$; p wird um 1 inkrementiert $p:=p+1 \bmod (m*2^L)$
  - wenn p wieder auf 0 gesetzt wird (Verdoppelung des Hash-Bereichs beendet), wird L um 1 erhöht

Lineares Hashing (2)

- Hash-Funktion
  - Da der Hash-Bereich wächst oder schrumpft, ist Hash-Funktion an ihn anzupassen.
  - Folge von Hash-Funktionen $h_0, h_1, ...$ mit

    $h_j(k) \in \{0, 1, ..., m* 2^j - 1\}$,

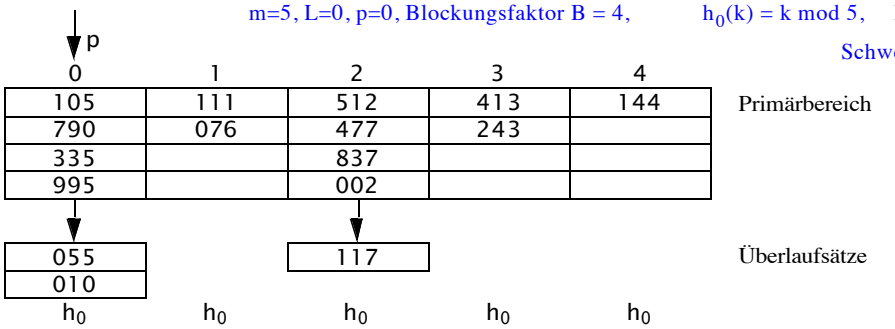
    z.B. $h_j(k) = k \bmod m * 2^j$
  - i.a. gilt $h = h_L(k)$
- Adressierung: 2 Fälle möglich
  - $h(k) >= p$ -> Satz ist in Bucket $h(k)$
  - $h(k) < p$ (Bucket wurde bereits gesplittet):
    Satz ist in Bucket $h_{L+1}(k)$ (d.h. in $h(k)$ oder $h(k) + m*2^L$)
  - gleiche Wahrscheinlichkeit für beide Fälle erwünscht

Lineares Hashing (3)

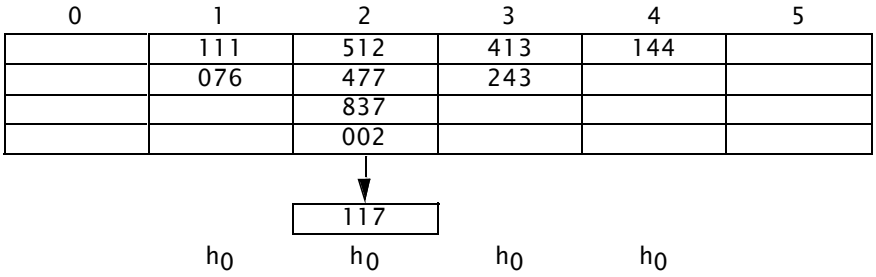
- Beispiel

m=5, L=0, p=0, Blockungsfaktor B = 4, $h_0(k) = k \bmod 5$, $h_1(k) = k \bmod 10$ ...

Schwellwert $ß_S = 0{,}8$

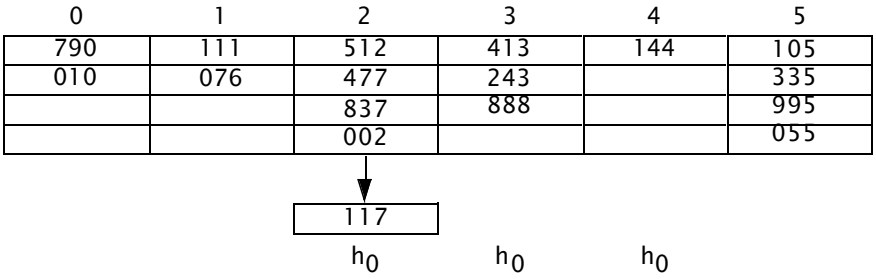


Einfügen von 888 erhöht Belegung auf 17/20=0,85 > ß -> Split-Vorgang



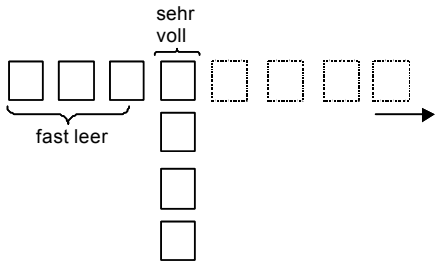
Lineares Hashing (4)

Einfügen von 244, 399, 100 erhöht Belegung auf 20/24=0,83 > ß -> Split-Vorgang



Lineares Hashing: Bewertung

- Überläufer weiterhin erforderlich
- ungünstiges Split-Verhalten / ungünstige Spleicherplatznutzung möglich (Splitten unterbelegter Seiten)



- Zugriffskosten 1 + x

Zusammenfassung

- Hashing: schnellster Ansatz zur direkten Suche
  - Schlüsseltransformation: berechnet Speicheradresse des Satzes
  - zielt auf bestmögliche Gleichverteilung der Sätze im Hash-Bereich (gestreute Speicherung)
  - anwendbar im Hauptspeicher und für Externspeicher
  - konstante Zugriffskosten O (1)
- Hashing bietet im Vergleich zu Bäumen eingeschränkte Funktionalität
  - i. a. kein sortiert sequentieller Zugriff
  - ordnungserhaltendes Hashing nur in Sonderfällen anwendbar
  - Verfahren sind vielfach statisch
- Idealfall: Direkte Adressierung (Kosten 1 für Suche/Einfügen/Löschen)
  - nur in Ausnahmefällen möglich (’dichte’ Schlüsselmenge)
- Hash-Funktion
  - Standard: **Divisionsrest-Verfahren**
  - ggf. zunächst numerischer Wert aus Schlüsseln zu erzeugen
  - Verwendung einer Primzahl für Divisor (Größe der Hash-Tabelle) wichtig

Zusammenfassung (2)

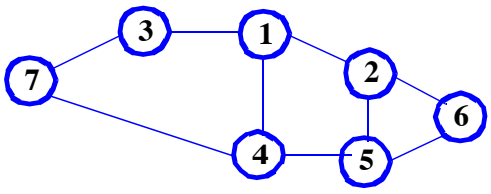
- Kollisionsbehandlung
  - Verkettung der Überläufer (separater Überlaufbereich) i.a. effizienter und einfacher zu realisieren als offene Adressierung
  - ausreichend große Hash-Tabelle entscheidend für Begrenzung der Kollisionshäufigkeit, besonders bei offener Adressierung
  - Belegungsgrad $\beta \leq 0.85$ dringend zu empfehlen
- Hash-Verfahren für Externspeicher
  - reduzierte Kollisionsproblematik, da Bucket b Sätze aufnehmen kann
  - direkte Suche ~ $1 + \delta$ Seitenzugriffe
  - statische Verfahren leiden unter schlechter Speicherplatznutzung und hohen Reorganisationskosten
- Dynamische Hashing-Verfahren: reorganisationsfrei
  - Erweiterbares Hashing: 2 Seitenzugriffe
  - Lineares Hashing: kein Directory, jedoch Überlaufseiten
- Erweiterbares Hashing widerlegt alte „Lehrbuchmeinungen“
  - „Hash-Verfahren sind immer statisch, da sie Feld fester Größe adressieren“
  - „Digitalbäume sind nicht ausgeglichen“
  - „Auch ausgeglichene Suchbäume ermöglichen bestenfalls Zugriffskosten von O (log n)“

3. Graphen

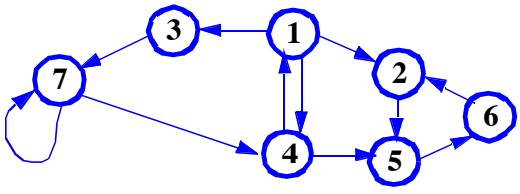
- Definitionen
- Implementierungsalternativen
  - Kantenliste, Knotenliste
  - Adjazenzmatrix, Adjazenzliste
  - Vergleich
- Traversierung von Graphen
  - Breitensuche
  - Tiefensuche
- Topologisches Sortieren
- Transitive Hülle (Warshall-Algorithmus)
- Kürzeste Wege (Dijkstra-Algorithmus etc.)
- Minimale Spannbäume (Kruskal-Algorithmus)
- Maximale Flüsse (Ford-Fulkerson)
- Maximales Matching

Einführung

- Graphen sind zur Repräsentation von Problemen vielseitig verwendbar, z.B.
  - Städte: Verbindungswege
  - Personen: Relationen zwischen ihnen
  - Rechner: Verbindungen
  - Aktionen: zeitliche Abhängigkeiten
- Graph: Menge von Knoten (Vertices) und Kanten (Edges)
  - ungerichtete Graphen
  - gerichtete Graphen (Digraph, Directed graph)
  - gerichtete, azyklische Graphen (DAG, Directed Acyclic Graph)



ungerichteter Graph G_u



gerichteter Graph G_g

Definitionen

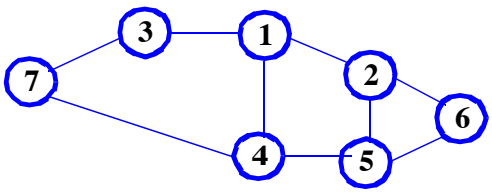
- $G = (V, E)$ heißt **ungerichteter Graph** : $\Leftrightarrow$
  - $V \neq \emptyset$ ist eine endliche, nichtleere Menge. V heißt Knotenmenge, Elemente von V heißen *Knoten*
  - E ist eine Menge von ein- oder zweielementigen Teilmengen von V. E heißt Kantenmenge, ein Paar $\{u,v\} \in E$ heißt *Kante*
  - Eine Kante $\{u\}$ heißt *Schlinge*
  - Zwei Knoten u und v heißen *benachbart* (adjazent): $\Leftrightarrow \{u,v\} \in E$ oder $(u=v) \wedge \{u\} \in E$.
- Sei $G = (V,E)$ ein ungerichteter Graph. Wenn E keine Schlinge enthält, so heißt G *schlingenlos*.

Bem. Im weiteren werden wir Kanten $\{u,v\}$ als Paare (u,v) oder (v,u) und Schlingen $\{u\}$ als Paar (u,u) schreiben, um spätere gemeinsame Definitionen für ungerichtete und gerichtete Graphen nicht differenzieren und notationell unterscheiden zu müssen.

- Seien $G = (V_G, E_G)$ und $H = (V_H, E_H)$ ungerichtete Graphen.
  - H heißt *Teilgraph* von G ($H \subset G$): $\Leftrightarrow V_G \supset V_H$ und $E_G \supset E_H$
  - H heißt vollständiger Teilgraph von G : $\Leftrightarrow$
    $H \subset G$ und $[(u,v) \in E_G$ mit $u,v \in V_H \Rightarrow (u,v) \in E_H]$.

Beispiele ungerichteter Graphen

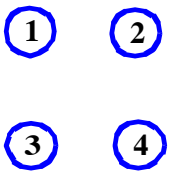
- Beispiel 1
  - $G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$,
  - $E_G = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$
- Beispiel 2



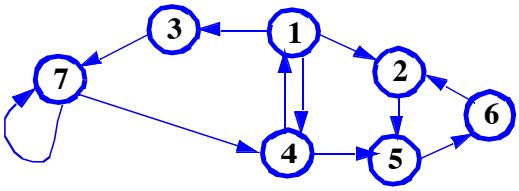
ungerichteter Graph G_u

Definitionen (2)

- G = (V,E) heißt gerichteter Graph (Directed Graph, Digraph) : ⇔
  - V ≠ ∅ ist endliche Menge. V heißt Knotenmenge, Elemente von V heißen Knoten.
  - E ⊆ V× V heißt Kantenmenge, Elemente von E heißen Kanten.
    Schreibweise: (u, v) oder u → v. u ist die Quelle, v das Ziel der Kante u → v.
  - Eine Kante (u, u) heißt Schlinge.
- Beispiel
  - G = (V_G, E_G) mit V_G = {1, 2, 3, 4} und E_G = {1 → 2, 1 → 3, 1 → 4, 2 → 3, 2 → 4, 3 → 4}



- Beispiel 2

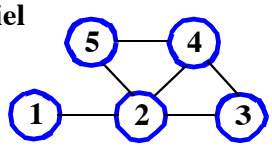


gerichteter Graph G_g

Definitionen (3)

- Sei G = (V,E) ein (un)gerichteter Graph und k = (v_0, ..., v_n) ∈ V^{n+1}.
  - k heißt *Kantenfolge* der Länge n von v_0 nach v_n, wenn für alle i ∈{0, ..., n-1} gilt: (v_i, v_{i+1}) ∈ E. Im gerichteten Fall ist v_0 der Startknoten und v_n der Endknoten, im ungerichteten Fall sind v_0 und v_n die Endknoten von k.
    v_1, ..., v_{n-1} sind die *inneren Knoten* von k. Ist v_0 = v_n, so ist die Kantenfolge *geschlossen*.
  - k heißt *Kantenzug* der Länge n von v_0 nach v_n, wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle i, j ∈{0, ..., n-1} mit i ≠ j gilt: (v_i, v_{i+1}) ≠ (v_j, v_{j+1}).
  - k heißt *Weg* (Pfad) der Länge n von v_0 nach v_n, wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle i, j ∈ {0, ..., n} mit i ≠ j gilt: v_i ≠ v_j.
  - k heißt *Zyklus* oder Kreis der Länge n, wenn k geschlossene Kantenfolge der Länge n von v_0 nach v_n und wenn k' = (v_0, ..., v_{n-1}) ein Weg ist. Ein Graph ohne Zyklus heißt kreisfrei oder *azyklisch*. Ein gerichteter azyklischer Graph heißt auch *DAG (Directed Acyclic Graph)*
  - Graph ist *zusammenhängend*, wenn zwischen je 2 Knoten ein Kantenzug existiert

Beispiel

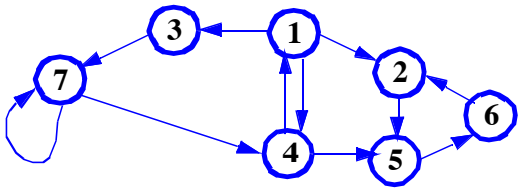


Kantenfolge:
Kantenzug:
Weg:
Zyklus:

- Sei G = (V, E) (un)gerichteter Graph, k Kantenfolge von v nach w. Dann gibt es einen Weg von v nach w.

Definitionen (4)

- Sei G = (V, E) ein gerichteter Graph
  - *Eingangsgrad:* eg (v) = |{v' | (v', v) ∈ E}|
  - *Ausgangsgrad:* ag(v) = |{v' | (v, v') ∈ E}|
  - G heißt *gerichteter Wald*, wenn G zyklenfrei ist und für alle Knoten v gilt eg(v) <= 1. Jeder Knoten v mit eg(v)=0 ist eine *Wurzel* des Waldes.
  - *Aufspannender Wald* (Spannwald) von G: gerichteter Wald W=(V,F) mit F ⊆ E
- *Gerichteter Baum (Wurzelbaum):* gerichteter Wald mit genau 1 Wurzel
  - für jeden Knoten v eines gerichteten Baums gibt es genau einen Weg von der Wurzel zu v
  - *Erzeugender / aufspannender Baum (Spannbaum)* eines Digraphen G: Spannwald von G mit nur 1 Wurzel
- zu jedem zusammenhängenden Graphen gibt es (mind.) einen Spannbaum



gerichteter Graph G_g

Definitionen (5)

- Markierte Graphen

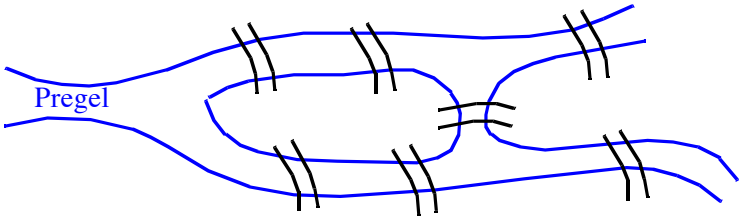
  Sei G = (V, E) ein (un)gerichteter Graph, M_V und M_E Mengen und $\mu : V \rightarrow M_V$ und $g : E \rightarrow M_E$ Abbildungen.
  - G' = (V, E, μ) heißt *knotenmarkierter Graph*
  - G" = (V, E, g) heißt *kantenmarkierter Graph*
  - G"' = (V, E, μ, g) heißt *knoten- und kantenmarkierter Graph*

  M_V und M_E sind die Markierungsmengen (z.B. Alphabete oder Zahlen)

Algorithmische Probleme für Graphen

Gegeben sei ein (un)gerichteter Graph G = (V, E)

- Man entscheide, ob G zusammenhängend ist
- Man entscheide, ob G azyklisch ist
- Man finde zu zwei Knoten, v, w ∈ V einen *kürzesten Weg* von v nach w (bzw. „günstigster" Weg bzgl. Kantenmarkierung)
- Man entscheide, ob G einen *Hamiltonschen Zyklus* besitzt, d.h. einen Zyklus der Länge $|V|$
- Man entscheide, ob G einen *Eulerschen Weg* besitzt, d.h. einen Weg, in dem jede Kante genau einmal verwendet wird, und dessen Anfangs- und Endpunkte gleich sind (*Königsberger Brückenproblem*)



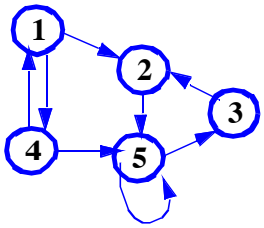
Algorithmische Probleme (2)

- *Färbungsproblem:* Man entscheide zu einer vorgegebenen natürlichen Zahl k („Anzahl der Farben"), ob es eine Knotenmarkierung $\mu : V \rightarrow \{1, 2, ..., k\}$ so gibt, daß für alle $(v, w) \in E$ gilt: $\mu(v) \neq \mu(w)$ [G azyklisch]
- *Cliquenproblem:* Man entscheide für ungerichteten Graphen G zu vorgegebener natürlichen Zahl k, ob es einen Teilgraphen G' („k-Clique") von G gibt, dessen Knoten alle paarweise durch Kanten verbunden sind
- *Matching-Problem:* Sei G = (V, E) ein Graph. Eine Teilmenge $M \subseteq E$ der Kanten heißt Matching, wenn jeder Knoten von V zu höchstens einer Kante aus M gehört. Problem: finde ein maximales Matching
- *Traveling Salesman Problem*: Bestimme optimale Rundreise durch n Städte, bei der jede Stadt nur einmal besucht wird und minimale Kosten entstehen

Hierunter sind bekannte NP-vollständige Probleme, z.B. das Cliquenproblem, das Färbungsproblem, die Hamilton-Eigenschaftsprüfung und das Traveling Salesman Problem

Speicherung von Graphen

- Knoten- und Kantenlisten
  - Speicherung von Graphen als Liste von Zahlen (z.B. in Array oder verketteter Liste)
  - Knoten werden von 1 bis n durchnumeriert; Kanten als Paare von Knoten
- Kantenliste
  - Liste: Knotenzahl, Kantenzahl, Liste von Kanten (je als 2 Zahlen)
  - Speicherbedarf: 2 + 2m (m = Anzahl Kanten)
- Knotenliste
  - Liste: Knotenzahl, Kantenzahl, Liste von Knoteninformationen
  - Knoteninformation: Ausgangsgrad und Zielknoten $ag(i)$, $v_1 .. v_{ag(i)}$
  - Speicherbedarf: 2 + n+m (n= Anzahl Knoten, m = Anzahl Kanten)
- Beispiel



Kantenliste:

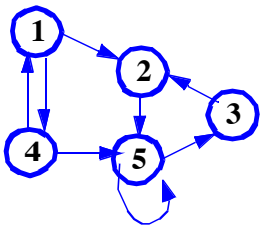
Knotenliste:

Speicherung von Graphen (2)

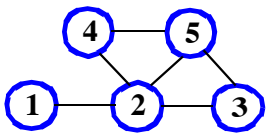
- Adjazenzmatrix

  Ein Graph G = (V,E) mit $|V| = n$ wird in einer Boole'schen n x n -Matrix

  $A_G = (a_{ij})$, mit $1 \le i,j \le n$ gespeichert, wobei $a_{ij} = \begin{cases} 0 & \text{falls } (i,j) \notin E \\ 1 & \text{falls } (i,j) \in E \end{cases}$
- Beispiel:



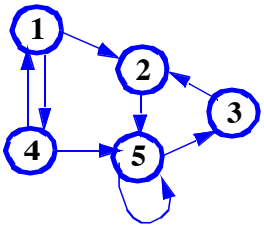
A_G	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	0	1
3	0	1	0	0	0
4	1	0	0	0	1
5	0	0	1	0	1

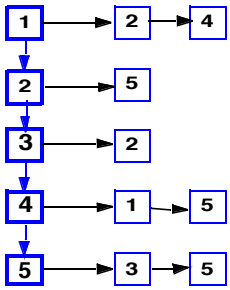


- Speicherplatzbedarf $O(n^2)$
  - jedoch nur 1 Bit pro Position (statt Knoten/Kantennummern)
  - unabhängig von Kantenmenge
  - für ungerichtete Graphen ergibt sich symmetrische Belegung (Halbierung des Speicherbedarfs möglich)

Speicherung von Graphen (3)

- Adjazenzlisten
  - verkettete Liste der n Knoten (oder Array-Realisierung)
  - pro Knoten: verkettete Liste der Nachfolger (repräsentiert die von dem Knoten ausgehenden Kanten)
  - Speicherbedarf: n+m Listenelemente





- Variante: doppelt verkettete Kantenlisten (doubly connected arc list, DCAL)

Speicherung von Graphen (4)

```
/** Repräsentiert einen Knoten im Graphen. */
public class Vertex {
  Object key = null;         // Knotenbezeichner
  LinkedList edges = null;   // Liste ausgehender Kanten

  /** Konstruktor */
  public Vertex(Object key) { this.key = key; edges = new LinkedList(); }

  /** Ueberschreibe Object.equals-Methode */
  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (obj instanceof Vertex) return key.equals(((Vertex) obj).key);
    else return key.equals(obj); }

  /** Ueberschreibe Object.hashCode-Methode */
  public int hashCode() { return key.hashCode(); } ... }

/** Repraesentiert eine Kante im Graphen. */
public class Edge {
  Vertex dest = null;  // Kantenzielknoten
  int weight = 0;      // Kantengewicht

  /** Konstruktor */
  public Edge(Vertex dest, int weight) {
    this.dest = dest;  this.weight=weight; } ... }
```

```
/** Graphrepräsentation. */
public class Graph {

  protected Hashtable vertices = null;  // enthaelt alle Knoten des Graphen

  /** Konstruktor */
  public Graph() { vertices = new Hashtable(); }

  /** Fuegt einen Knoten in den Graphen ein. */
  public void addVertex(Object key) {
    if (vertices.containsKey(key))
      throw new GraphException("Knoten exisitert bereits!");
    vertices.put(key, new Vertex(key)); }

  /** Fuegt eine Kante in den Graphen ein. */
  public void addEdge(Object src, Object dest, int weight) {
    Vertex vsrc = (Vertex) vertices.get(src);
    Vertex vdest = (Vertex) vertices.get(dest);
    if (vsrc == null)
      throw new GraphException("Ausgangsknoten existiert nicht!");
    if (vdest == null)
      throw new GraphException("Zielknoten existiert nicht!");
    vsrc.edges.add(new Edge(vdest, weight)); }

  /** Liefert einen Iterator ueber alle Knoten. */
  public Iterator getVertices() { return vertices.values().iterator(); }

  /** Liefert den zum Knotenbezeichner gehoerigen Knoten. */
  public Vertex getVertex(Object key) {
    return (Vertex) vertices.get(key); } }
```


Speicherung von Graphen: Vergleich

- Komplexitätsvergleich

Operation	Kantenliste	Knotenliste	Adjazenzmatrix	Adjazenzliste
Einfügen Kante	O(1)	O(n+m)	O(1)	O (1) / O (n)
Löschen Kante	O(m)	O(n+m)	O(1)	O(n)
Einfügen Knoten	O(1)	O(1)	$O(n^2)$	O(1)
Löschen Knoten	O(m)	O(n+m)	$O(n^2)$	O(n+m)
Speicherplatzbedarf	O(m)	O(n+m)	$O(n^2)$	O(n+m)

  - Löschen eines Knotens löscht auch zugehörige Kanten
  - Änderungsaufwand abhängig von Realisierung der Adjazenzmatrix und Adjazenzliste

- Welche Repräsentation geeigneter ist, hängt auch vom Problem ab:
  - *Frage:* Gibt es Kante von a nach b: *Matrix*
  - Durchsuchen von Knoten in durch Nachbarschaft gegebener Reihenfolge: *Listen*

- Transformation zwischen Implementierungsalternativen möglich

Traversierung

- Traversierung: Durchlaufen eines Graphen, bei dem jeder Knoten (bzw. jede Kante) genau 1-mal aufgesucht wird
  - Beispiel 1: Aufsuchen aller Verbindungen (Kanten) und Kreuzungen (Knoten) in einem Labyrinth
  - Beispiel 2: Aufsuchen aller Web-Server durch Suchmaschinen-Roboter
- Generische Lösungsmöglichkeit für Graphen G=(V, E)

```
for each Knoten v ∈ V do { markiere v als unbearbeitet};
B = {s}; // Initialisierung der Menge besuchter Knoten B mit Startknoten s ∈ V;
markiere s als bearbeitet;
while es gibt noch unbearbeitete Knoten v' mit (v,v') ∈ E und v ∈ B do {
        B = B ∪ {v'};
        markiere v' als bearbeitet;
        };
}
```

- Realisierungen unterscheiden sich bezüglich Verwaltung der noch abzuarbeitenden Knotenmenge und Auswahl der jeweils nächsten Kante

Traversierung (2)

- Breitendurchlauf (Breadth First Search, BFS)
  - ausgehend von Startknoten werden zunächst alle direkt erreichbaren Knoten bearbeitet
  - danach die über mindestens zwei Kanten vom Startknoten erreichbaren Knoten, dann die über drei Kanten usw.
  - es werden also erst die Nachbarn besucht, bevor zu den Söhnen gegangen wird
  - kann mit FIFO-Datenstruktur für noch zu bearbeitende Knoten realisiert werden
- Tiefendurchlauf (Depth First Search, DFS)
  - ausgehend von Startknoten werden zunächst rekursiv alle Söhne (Nachfolger) bearbeitet; erst dann wird zu den Nachbarn gegangen
  - kann mit Stack-Datenstruktur für noch zu bearbeitende Knoten realisiert werden
  - Verallgemeinerung der Traversierung von Bäumen
- Algorithmen nutzen „Farbwert“ pro Knoten zur Kennzeichnung des Bearbeitungszustandes
  - weiß: noch nicht bearbeitet
  - schwarz: abgearbeitet
  - grau: in Bearbeitung

Breitensuche

- Bearbeite einen Knoten, der in *n Schritten* von *u* erreichbar ist, erst, wenn alle Knoten, die in n-1 Schritten erreichbar sind, abgearbeitet wurden.
  - ungerichteter Graph G = (V,E); Startknoten s; Q sei FIFO-Warteschlange.
  - zu jedem Knoten u wird der aktuelle Farbwert, der Abstand d zu Startknoten s, und der Vorgänger pred, von dem aus u erreicht wurde, gespeichert
  - Funktion succ(u) liefert die Menge der direkten Nachfolger von u
  - pred-Werte liefern nach Abarbeitung für zusammenhängende Graphen einen *aufspannenden Baum (Spannbaum)*, ansonsten *Spannwald*

BFS(G,s):

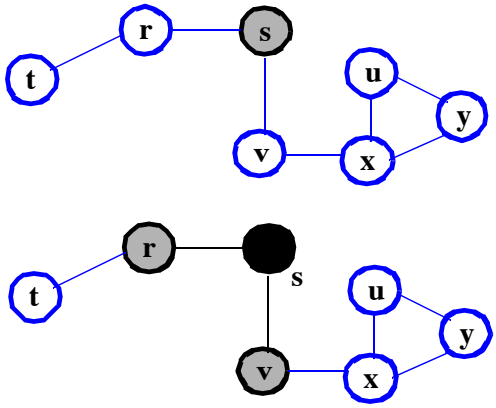
for each Knoten v ∈ V - s **do** { farbe[v]= weiß; d[v] = ∞; pred [v] = null };
farbe[s] = grau; d[s] = 0; pred [s] = null; Q = emptyQueue; Q = enqueue(Q,s);
while not isEmpty(Q) **do** { v = front(Q);
    for each u ∈ succ (v) **do** {
        if farbe(u) = weiß **then**
        { farbe[u] = grau; d[u] = d[v]+1; pred[u] = v; Q = enqueue(Q,u); };
    };
    dequeue(Q); farbe[v] = schwarz;
}

Breitensuche (2)

```
/** Liefert die Liste aller erreichbaren Knoten in Breitendurchlauf. */
  public List traverseBFS(Object root, Hashtable d, Hashtable pred) {
    LinkedList list = new LinkedList();
    Hashtable color = new Hashtable();
    Integer gray = new Integer(1);
    Integer black = new Integer(2);
    Queue q = new Queue();
    Vertex v, u = null;
    Iterator eIter = null;
    v = (Vertex)vertices.get(root);
    color.put(v, gray);
    d.put(v, new Integer(0));
    q.enqueue(v);
    while (! q.empty()) {
      v = (Vertex) vertices.get(((Vertex)q.front()).key);
      eIter = v.edges.iterator();
      while(eIter.hasNext()) {
        u = ((Edge)eIter.next()).dest;
        if (color.get(u) == null) {
          color.put(u, gray);
          d.put(u, new Integer(((Integer)d.get(v)).intValue() + 1));
          pred.put(u, v);
          q.enqueue(u);
        }
      }
      q.dequeue();
      list.add(v);
      color.put(v, black);
    }
    return list;
  }
```

Breitensuche (3)

- Beispiel:



- *Komplexität:* ein Besuch pro Kante und Knoten: O(n + m)
  - falls G zusammenhängend gilt |E| > |V| -1 -> Komplexität O(m)
- Breitensuche unterstützt Lösung von Distanzproblemen, z.B. Berechnung der Länge des *kürzesten Wegs* eines Knoten s zu anderen Knoten

Tiefensuche

- Bearbeite einen Knoten v erst dann, wenn alle seine Söhne bearbeitet sind (außer wenn ein Sohn auf dem Weg zu v liegt)
  - (un)gerichteter Graph G = (V,E); succ(v) liefert Menge der direkten Nachfolger von Knoten v
  - zu jedem Knoten v wird der aktuelle Farbwert, die Zeitpunkte *in* bzw. *out*, zu denen der Knoten im Rahmen der Tiefensuche erreicht bzw. verlassen wurden, sowie der Vorgänger *pred*, von dem aus v erreicht wurde, gespeichert
  - die in- bzw. out-Zeitpunkte ergeben eine Reihenfolge der Knoten analog zur Vor- bzw. Nachordnung bei Bäumen

DFS(G):

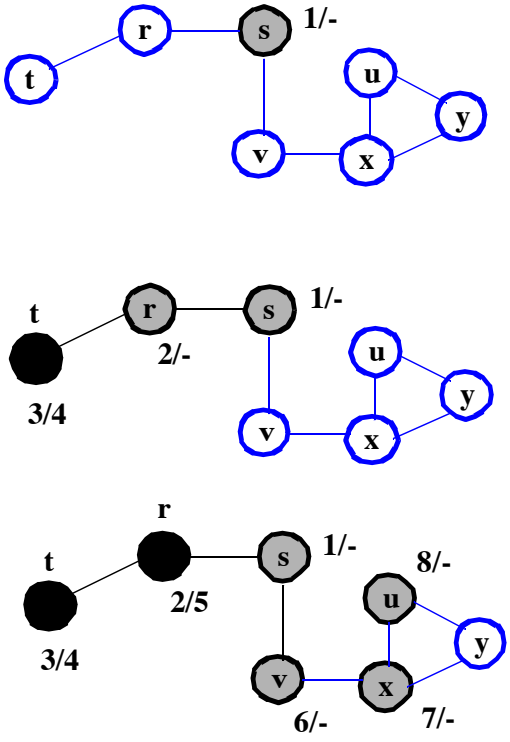
for each Knoten v ∈ V **do** { farbe[v]= weiß; pred [v] = null };
zeit = 0; **for each** Knoten v ∈ V **do** { **if** farbe[v]= weiß **then** DFS-visit(v) };

DFS-visit (v): **// rekursive Methode zur Tiefensuche**

farbe[v]= grau; zeit = zeit+1; in[v]=zeit;
for each u ∈ succ (v) **do** { **if** farbe(u) = weiß **then** { pred[u] = v; DFS-visit(u); }; };
farbe[v] = schwarz; zeit = zeit+1; out[v]=zeit;

- *lineare Komplexität* O(n+m)
  - DFS-visit wird genau einmal pro (weißem) Knoten aufgerufen
  - pro Knoten erfolgt Schleifendurchlauf für jede von diesem Knoten ausgehende Kante

Tiefensuche: Beispiel

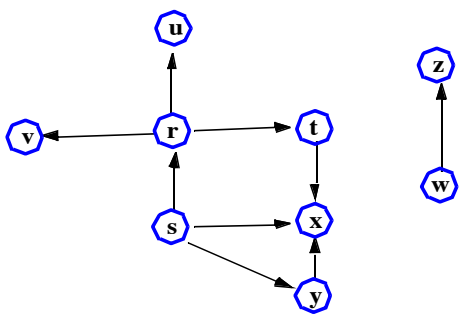


Topologische Sortierung

- gerichtete Kanten eines zyklenfreien Digraphs (DAG) beschreiben Halbordnung unter Knoten
- topologische Sortierung erzeugt vollständige Ordnung, die nicht im Widerspruch zur partiellen Ordnung steht
  - d.h. falls eine Kante von Knoten i nach j existiert, erscheint i in der linearen Ordnung vor j
- Topologische Sortierung eines Digraphen G = (V,E):

  Abbildung $ord: V \rightarrow \{1, ..., n\}$ mit $|V| = n$,

  so daß mit $(u,v) \in E$ auch $ord(u) < ord(v)$ gilt.
- Beispiel:



Topologische Sortierung (2)

- Satz: Digraph G = (V,E) ist zyklenfrei <=> für G existiert eine topologische Sortierung

  Beweis: <= klar

  => Induktion über |V|.

  Induktionsanfang: |V| = 1, keine Kante, bereits topologisch sortiert

  Induktionsschluß: |V| = n.

  - Da *G azyklisch* ist, muß es einen Knoten v ohne Vorgänger geben. Setze ord(v) = 1
  - Durch Entfernen von v erhalten wir einen azyklischen Graphen G' mit |V'| = n-1, für den es nach Induktionsvoraussetzung *topologische Sortierung ord'* gibt
  - Die gesuchte topologische Sortierung für G ergibt sich durch ord(v') = ord'(v') + 1, für alle v' ∈ V'

- Korollar: zu jedem DAG gibt es eine topologische Sortierung

Topologische Sortierung (3)

- Beweis liefert einen Algorithmus zur topologischen Sortierung

  Bestimmung einer Abbildung ord für gerichteten Graphen G = (V,E) zur topologischen Sortierung und Test auf Zyklenfreiheit

  TS (G):

  ```
  i=0;
  while G hat wenigstens einen Knoten v mit eg (v) = 0 do {
                  i = i+1; ord(v) := i; G = G - {v};    };
  if G = {} then „G ist zyklenfrei“ else „G hat Zyklen“;
  ```

  - (Neu-)Bestimmung des Eingangsgrades kann sehr aufwendig werden
  - Effizienter ist daher, den jeweils aktuellen Eingangsgrad zu jedem Knoten zu speichern

- effiziente Alternative: Verwendung der Tiefensuche
  - Verwendung der out-Zeitpunkte, in umgekehrter Reihenfolge
  - Realisierung mit Aufwand O(n+m)
  - Mit denselben Kosten O(n+m) kann die Zyklenfreiheit eines Graphen getestet werden (Zyklus liegt dann vor, wenn bei der Tiefensuche der Nachfolger eines Knotens bereits *grau* gefärbt ist!)

Topologische Sortierung (4)

- Anwendungsbeispiel

  zerstreuter Professor legt die Reihenfolge beim Ankleiden fest

  - Unterhose vor Hose
  - Hose vor Gürtel
  - Hemd vor Gürtel
  - Gürtel vor Jackett
  - Hemd vor Krawatte
  - Krawatte vor Jackett
  - Socken vor Schuhen
  - Unterhose vor Schuhen
  - Hose vor Schuhen
  - Uhr: egal

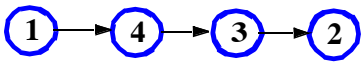
- Ergebnis der topologischen Sortierung mit Tiefensuche abhängig von Wahl der Startknoten (weissen Knoten)

Transitive Hülle

- Erreichbarkeit von Knoten
  - welche Knoten sind von einem gegebenen Knoten aus erreichbar?
  - gibt es Knoten, von denen aus alle anderen erreicht werden können?
  - Bestimmung der transitiven Hülle ermöglicht Beantwortung solcher Fragen

- Ein Digraph G* = (V,E*) ist die *reflexive, transitive Hülle* (kurz: Hülle) eines Digraphen G = (V,E), wenn genau dann (v,v') ∈ E* ist, wenn es einen Weg von v nach v' in G gibt.

Beispiel



A	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

- Algorithmus zur Berechnung von Pfeilen der reflexiven transitiven Hülle

```
boolean [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = true;
for (int i = 0; i < A.length; i++)
        for (int j = 0; j < A.length; j++)
                if A[i][j] for (int k = 0; k < A.length; k++) if A [j][k] A [i][k] = true;
```

  - es werden nur Pfade der Länge 2 bestimmt!
  - Komplexität $O(n^3)$

Transitive Hülle: Warshall-Algorithmus

- Einfache Modifikation liefert vollständige transitive Hülle

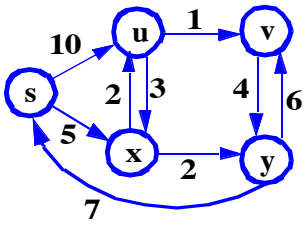
```
boolean [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = true;
for (int j = 0; j < A.length; j++)
        for (int i = 0; i < A.length; i++)
                if A[i][j] for (int k = 0; k < A.length; k++) if A [j][k] A [i][k] = true;
```

- Korrektheit kann über Indusktionsbeweis gezeigt werden
  - *Induktionshypothese P(j):* gibt es zu beliebigen Knoten i und k einen Weg von i nach k, so dass alle Zwischenknoten aus der Menge {0, 1, ...j} sind, so wird in der j-ten Iteration A [i][k]=true gesetzt.
    Wenn P(j) für alle j gilt, wird keine Kante der transitiven Hülle vergessen
  - *Induktionsanfang*: j=0: Falls A[i][0] und A[0][k] gilt, wird in der Schleife mit j=0 auch A[i][k] gesetzt
  - *Induktionsschluß*: Sei P(j) wahr für 0 .. j. Sei ein Weg von i nach k vorhanden, der Knoten j+1 nutzt, dann gibt es auch einen solchen, auf dem j+1 nur einmal vorkommt. Aufgrund der Induktionshypothese wurde in einer früheren Iteration der äußeren Schleife bereits (i,j+1) und (j+1,k) eingefügt. In der (j+1)-ten Iteration wird nun (i,k) gefunden. Somit gilt auch P(j+1).

- Komplexität
  - innerste for-Schleife wird nicht notwendigerweise n^2-mal (n=|V|) durchlaufen, sondern nur falls Verbindung von i nach j in E* vorkommt, also O(k) mit k=|E*| mal
  - Gesamtkomplexität $O(n^2+k \cdot n)$.

Kürzeste Wege

- *kantenmarkierter (gewichteter) Graph* G = (V, E, g)
  - Weg/Pfad P der Länge n: $(v_0,v_1), (v_1,v_2), \ldots, (v_{n-1}, v_n)$
  - *Gewicht* (Länge) des Weges/Pfads
    $$w(P) = \Sigma\, g((v_i, v_{i+1}))$$
  - *Distanz* d (u,v): Gewicht des kürzesten Pfades von u nach v



- Varianten
  - nichtnegative Gewichte vs. negative und positive Gewichte
  - Bestimmung der kürzesten Wege
    a) zwischen allen Knotenpaaren,
    b) von einem Knoten u aus
    c) zwischen zwei Knoten u und v
- Bemerkungen
  - kürzeste Wege sind nicht immer eindeutig
  - kürzeste Wege müssen nicht existieren:
    es existiert kein Weg;
    es existiert Zyklus mit negativem Gewicht

Kürzeste Wege (2)

- Warshall-Algorithmus läßt sich einfach modifizieren, um kürzeste Wege zwischen allen Knotenpaaren zu berechnen
  - Matrix A enthält zunächst Knotengewichte pro Kante, ∞ falls "keine Kante" vorliegt
  - A[i,i] wird mit 0 vorbelegt
  - Annahme: kein Zyklus mit negativem Gewicht vorhanden

```
int [ ] [ ] A = { ...}; for (int i = 0; i < A.length; i++) A [i] [i] = 0;
for (int j = 0; j < A.length; j++)
        for (int i = 0; i < A.length; i++)
                for (int k = 0; k < A.length; k++)
                        if (A [i][j] + A [j][k] < A [i][k])
                                A [i][k]= A [i][j] + A [j][k];
```

- Komplexität $O(n^3)$

Kürzeste Wege: *Dijkstra-Algorithmus*

- Bestimmung der von einem Knoten ausgehenden kürzesten Wege
  - *gegeben:* kanten-bewerteter Graph G = (V,E, g) mit g: E -> R^+ (Kantengewichte)
  - Startknoten s; zu jedem Knoten u wird die Distanz zu Startknoten s in D[u] geführt
  - Q sei Prioritäts-Warteschlange (sortierte Liste); Priorität = Distanzwert
  - Funktion succ(u) liefert die Menge der direkten Nachfolger von u

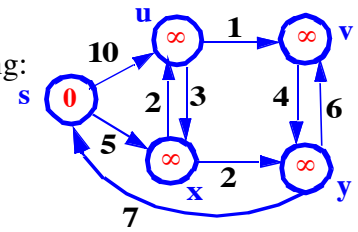
Dijkstra (G,s):

```
for each Knoten v ∈ V - s do { D[v] = ∞;};
D[s] = 0; PriorityQueue Q = V;
while not isEmpty(Q) do { v = extractMinimum(Q);
        for each u ∈ succ (v) ∩ Q do {
                if D[v] + g ((v,u)) < D[u] then
                        { D[u] = D[v] + g ((v,u));
                         adjustiere Q an neuen Wert D[u]; };
        };
}
```

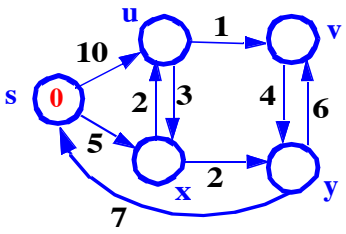
- Verallgemeinerung der Breitensuche (gewichtete Entfernung)
- funktioniert nur bei nicht-negativen Gewichten
- Optimierung gemäß Greedy-Prinzip

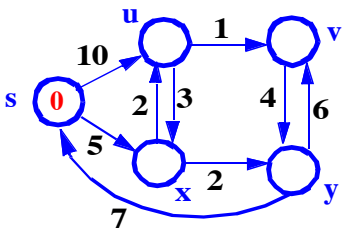
Dijkstra-Algorithmus: Beispiel

Initialisierung:



Q = < (s:0), (u: ∞), (v: ∞), (x: ∞), (y: ∞)>





Dijkstra-Algorithmus (3)

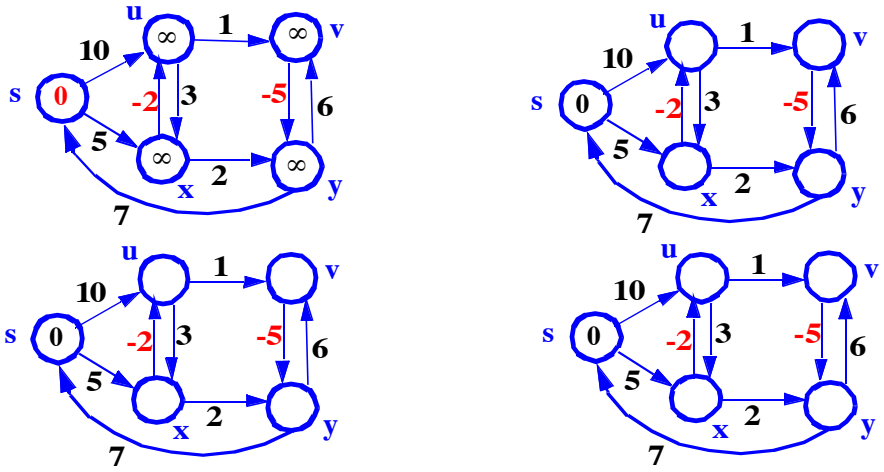
- Korrektheitsbeweis
  - nach i Schleifendurchgängen sind die Längen von i Knoten, die am nächsten an s liegen, korrekt berechnet und diese Knoten sind aus Q entfernt.
  - *Induktionsanfang:* s wird gewählt, D (s) = 0
  - *Induktionsschritt:* Nimm an, v wird aus Q genommen. Der kürzeste Pfad zu v gehe über direkten Vorgänger v' von v. Da v' näher an s liegt, ist v' nach Induktionsvoraussetzung mit richtiger Länge D(v') bereits entfernt. Da der *kürzeste Weg* zu v die Länge D(v') + g((v',v)) hat und dieser Wert bei Entfernen von v' bereits v zugewiesen wurde, wird v mit der richtigen Länge entfernt.
  - erfordert nichtnegative Kantengewichte (steigende Länge durch hinzugenommene Kanten)
- Komplexität <= $O(n^2)$
  - n-maliges Durchlaufen der *äußeren Schleife* liefert Faktor O(n)
  - innere Schleife: Auffinden des Minimums begrenzt durch O(n), ebenso das Aufsuchen der Nachbarn von v
- Pfade bilden aufspannenden Baum (der die Wegstrecken von s aus gesehen minimiert)
- Bestimmung des kürzesten Weges zwischen u und v: Spezialfall für Dijkstra-Algorithmus mit Start-Knoten u (Beendigung sobald v aus Q entfernt wird)

Kürzeste Wege mit negativen Kantengewichten

- Bellmann-Ford-Algorithmus **BF (G,s):**

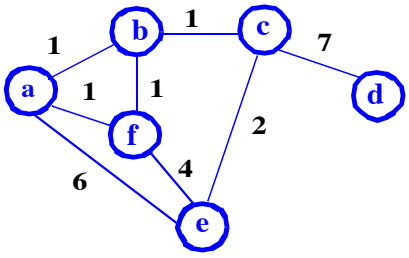
```
for each Knoten v ∈ V - s do { D[v] = ∞;}; D[s] = 0;
for i = 1 to |E|-1 do
        for each (u,v) ∈ E do {
                if D[u] + g ((u,v)) < D[v] then D[v] = D[u] + g ((u,v));
        };
```

Beispiel:



Minimale Spannbäume

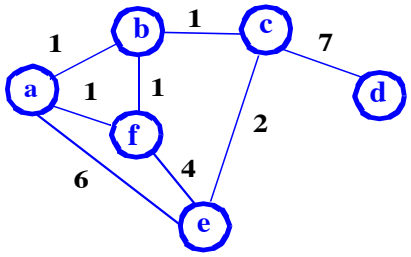
- Problemstellung: zu zusammenhängendem Graph soll Spannbaum (aufspannender Baum) mit minimalem Kantengewicht (minimale Gesamtlänge) bestimmt werden
  - relevant z.B. zur Reduzierung von Leitungskosten in Versorgungsnetzen
  - zusätzliche Knoten können zur Reduzierung der Gesamtlänge eines Graphen führen
- Kruskal-Algorithmus (1956)
  - Sei G = (V, E, g) mit g: E -> R (Kantengewichte) gegebener ungerichteter, zusammenhängender Graph. Zu bestimmen minimaler Spannbaum T = (V, E')
  - E' = {}; sortiere E nach Kantengewicht und bringe die Kanten in PriorityQueue Q; jeder Knoten v bilde eigenen Spannbaum(-Kandidat)
  - solange Q nicht leer:
    - entferne erstes Element e = (u,v)
    - wenn beide Endknoten u und v im selben Spannbaum sind, verwerfe e, ansonsten nehme e in E' auf und fasse die Spannbäume von u und v zusammen



- Analog: Bestimmung maximaler Spannbäume (absteigende Sortierung)

Minimale Spannbäume (2)

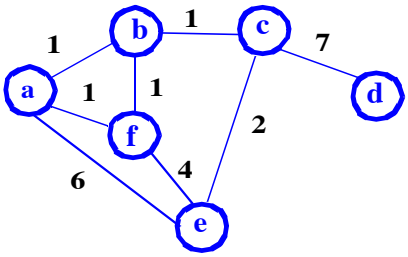
- Anwendung des Kruskal-Algorithmus



- Komplexität O (m log n)

Minimale Spannbäume (3)

- Alternative Berechnung (Dijkstra)
  - Startknoten s
  - Knotenmenge B enthält bereits abgearbeitete Knoten

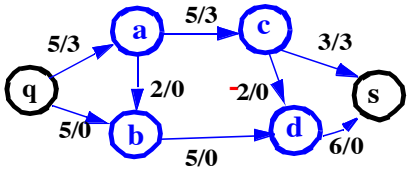


s = an Kante mit minimalem Gewicht beteiligter Knoten
B={ s }; E' = { };
while |B| < |V| do {
wähle (u,v) ∈ E mit minimalem Gewicht mit u ∈ B, v ∉ B;
füge (u,v) zu E' hinzu;
füge v zu B hinzu; }

- es wird nur 1 Spannbaum erzeugt
- effiziente Implementierbarkeit mit PriorityQueue über Kantengewicht

Flüsse in Netzen

- Anwendungsprobleme:
  - Wieviele Autos können durch ein Straßennetz fahren?
  - Wieviel Abwasser fasst ein Kanalnetz?
  - Wieviel Strom kann durch ein Leitungsnetz fließen?



Kantenmarkierung:
Kapazität c(e) / Fluß f(e)

- *Def.:* Ein (Fluß-) *Netzwerk* ist ein gerichteter Graph G = (V, E, c) mit ausgezeichneten Knoten *q (Quelle)* und *s (Senke),* sowie einer *Kapazitätsfunktion* c: E -> Z^+.

- Ein *Fluß* für das Netzwerk ist eine Funktion f: E -> Z^+, so daß gilt:
  - *Kapazitätsbeschränkung*: **f(e) ≤ c(e), für alle e in E.**
  - *Flußerhaltung:* für alle v in V \ {q,s}: $\Sigma_{(v',v) \in E}$ **f((v',v))** = $\Sigma_{(v,v') \in E}$ **f((v,v'))**
  - Der *Wert* von f, w(f), ist die Summe der Flußwerte der die Quelle q verlassenden Kanten: $\Sigma_{(q,v) \in E}$ f((q,v))

- Gesucht: Fluß mit maximalem Wert
  - begrenzt durch Summe der aus q wegführenden bzw. in s eingehenden Kapazitäten
  - jeder weitere „Schnitt“ durch den Graphen, der q und s trennt, begrenzt max. Fluss

Flüsse in Netzen (2)

- *Schnitt (A, B)* eines Fluß-Netzwerks ist eine Zerlegung von V in disjunkte Teilmengen A und B, so daß q ∈ A und s ∈ B.
  - Die Kapazität des Schnitts ist **c(A,B) = $\Sigma_{u \in A, v \in B}$ c((u,v))**
  - *minimaler Schnitt* (minimal cut): Schnitt mit kleinster Kapazität

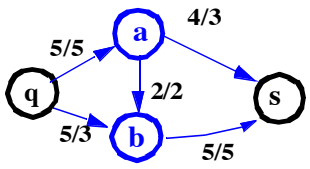
- Restkapazität, Restgraph

  Sei f ein zulässiger Fluß für G = (V,E). Sei E' = {(v,w) | (v,w) ∈ E oder (w,v) ∈ E}

  - Wir definieren die *Restkapazität einer Kante* e = (v,w) wie folgt:

    rest(e) = c(e) - f(e) falls e ∈ E
    f((w,v)) falls (w,v) ∈ E
  - Der *Restgraph* von f (bzgl. G) besteht aus den Kanten e ∈ E', für die rest(e) > 0

- Jeder gerichtete Pfad von q nach s im Restgraphen heißt *zunehmender Weg*

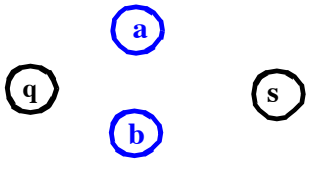


Kantenmarkierung: Kapazität c(e) / Fluß f(e)

verwendetete Wege:
1.) q, a, b, s (Kapaz. 2)
2.) q, b, s (3)
3.) q, a, s (3)

w (f) = 8, nicht maximal

Restgraph:



Kantenmarkierung: rest (e)

Flüsse in Netzen (3)

■ Theorem (Min-Cut-Max-Flow-Theorem):

Sei f zulässiger Fluß für G. Folgende Aussagen sind äquivalent:

1) f ist maximaler Fluß in G.

2) Der Restgraph von f enthält keinen zunehmenden Weg.

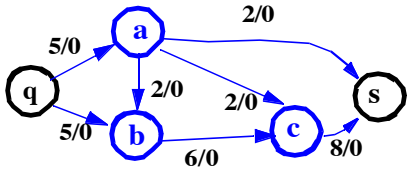
3) w(f) = c(A,B) für einen Schnitt (A,B) von G.

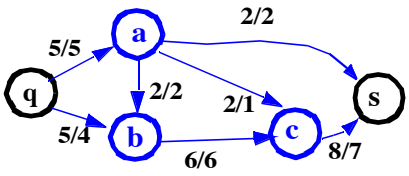
■ *Ford-Fulkerson-Algorithmus*

- füge solange zunehmende Wege zum Gesamtfluß hinzu wie möglich
- Kapazität erhöht sich jeweils um Minimum der verfügbaren Restkapazität der einzelnen Kanten des zunehmenden Weges

```
for each e ∈ E { f(e) = 0;}
while ( es gibt zunehmenden Weg p im Restgraphen von f ) {
        r = min{rest(e) | e liegt in p};
        for each e = (v,w) auf Pfad p {
            if (e in E)     f(e)= f(e) + r ;
            else            f((w,v)) = f((w,v)) - r; } }
```

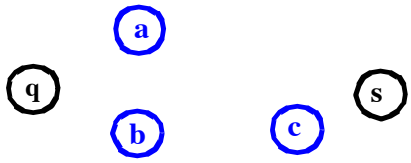

Ford-Fulkerson: Anwendungsbeispiel

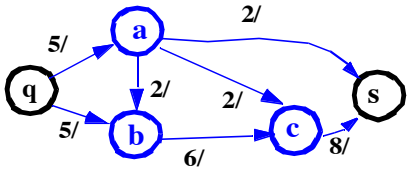




Kantenmarkierung:
Kapazität c(e) / Fluß f(e)

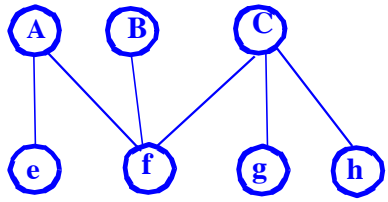
Restgraph:





Maximales Matching

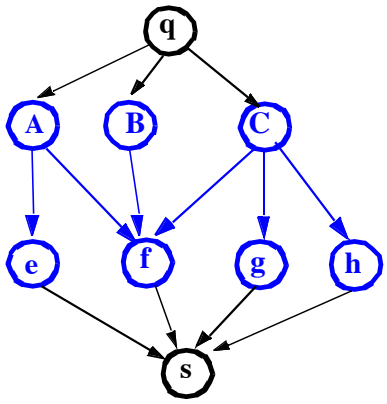
- *Beispiel:*
  - Eine Gruppe von Erwachsenen und eine Gruppe von Kindern besuchen Disneyland.
  - Auf der Achterbahn darf ein Kind jeweils nur in Begleitung eines Erwachsenen fahren.
  - Nur Erwachsene/Kinder, die sich kennen, sollen zusammen fahren. Wieviele Kinder können maximal eine Fahrt mitmachen?



- *Matching (Zuordnung) M* für ungerichteten Graphen G = (V, E) ist eine Teilmenge der Kanten, so daß jeder Knoten in V in höchstens einer Kante vorkommt
  - |M| = Größe der Zuordnung
  - *Perfektes Matching:* kein Knoten bleibt „allein“ (unmatched), d.h. jeder Knoten ist in einer Kante von M vertreten
- Matching M ist maximal, wenn es kein Matching M' gibt mit |M| < |M'|
- Verallgemeinerung mit gewichteten Kanten: Matching mit maximalem Gewicht

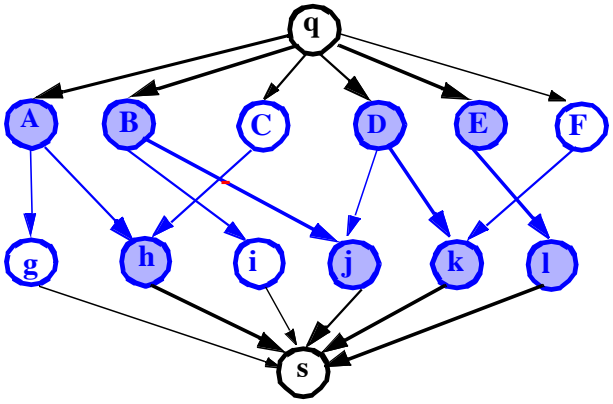
Matching (2)

- *Def.:* Ein *bipartiter Graph* ist ein Graph, dessen Knotenmenge V in zwei disjunkte Teilmengen V1 und V2 aufgeteilt ist, und dessen Kanten jeweils einen Knoten aus V1 mit einem aus V2 verbinden
- Maximales Matching kann auf maximalen Fluß zurückgeführt werden:
  - Quelle und Senke hinzufügen.
  - Kanten von V1 nach V2 richten.
  - Jeder Knoten in V1 erhält eingehende Kante von der Quelle.
  - Jeder Knoten in V2 erhält ausgehende Kante zur Senke.
  - Alle Kanten erhalten Kapazität c(e) = 1.
- Jetzt kann Ford-Fulkerson-Algorithmus angewendet werden



Matching (3)

- Weiteres Anwendungsbeispiel



  - ist gezeigtes Matching maximal?

Zusammenfassung

- viele wichtige Informatikprobleme lassen sich mit gerichteten bzw. ungerichteten Graphen behandeln
- wesentliche Implementierungsalternativen: Adjazenzmatrix und Adjazenzlisten
- Algorithmen mit linearem Aufwand:
  - Traversierung von Graphen: Breitensuche vs. Tiefensuche
  - Topologisches Sortieren
  - Test auf Azyklität
- Weitere wichtige Algorithmen†:
  - Warshall-Algorithmus zur Bestimmung der transitiven Hülle
  - Dijkstra-Algorithmus bzw. Bellmann-Ford für kürzeste Wege
  - Kruskal-Algorithmus für minimale Spannbäume
  - Ford-Fulkerson-Algorithmus für maximale Flüsse bzw. maximales Matching
- viele NP-vollständige Optimierungsprobleme
  - Traveling Salesman Problem, Cliquenproblem, Färbungsproblem ...
  - Bestimmung eines planaren Graphen (Graph-Darstellung ohne überschneidende Kanten)

† Animationen u.a. unter *http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/main/index.shtml*

4. Suche in Texten

- Einführung
- Suche in dynamischen Texten (ohne Indexierung)
  - Naiver Algorithmus (Brute Force)
  - Knuth-Morris-Pratt (KMP) - Algorithmus
  - Boyer-Moore (BM) - Algorithmus
  - Signaturen
- Suche in (weitgehend) statischen Texten -> Indexierung
  - Suffix-Bäume
  - Invertierte Listen
  - Signatur-Dateien
- Approximative Suche
  - k-Mismatch-Problem
  - Editierdistanz
  - Berechnung der Editierdistanz

Einführung

- Problem: Suche eines Teilwortes/Musters/Sequenz in einem Text
  - String Matching
  - Pattern Matching
  - Sequence Matching
- häufig benötigte Funktion
  - Textverarbeitung
  - Durchsuchen von Web-Seiten
  - Durchsuchen von Dateisammlungen etc.
  - Suchen von Mustern in DNA-Sequenzen (begrenztes Alphabet: A, C, G, T)
- Dynamische vs. statische Texte
  - dynamische Texte (z.B. im Texteditor): aufwendige Vorverarbeitung / Indizierung i.a. nicht sinnvoll
  - relativ statische Texte: Erstellung von Indexstrukturen zur Suchbeschleunigung
- Suche nach beliebigen Strings/Zeichenketten vs. Wörten/Begriffen
- Exakte Suche vs. approximative Suche (Ähnlichkeitssuche)

Einführung (2)

- Genauere Aufgabenstellung (exakte Suche)
  - *Gegeben:* Zeichenkette *text [1..n]* aus einem endlichen Alphabet Σ, Muster (Pattern) *pat [1..m]* mit pat[i] ∈ Σ, m <= n
  - *Fenster* w_i ist eine Teilzeichenkette von text der Länge m, die an Position i beginnt, also text [i] bis text[i+m-1]
  - Ein Fenster w_i, das mit dem Muster p übereinstimmt, heißt *Vorkommen* des Musters an Position i. w_i ist Vorkommen: text [i] = pat [1], text[i+1]=pat[2], ..., text[i+m-1]=pat[m]
  - Ein *Mismatch* in einem Fenster w_i ist eine Position j, an der das Muster mit dem Fenster nicht übereinstimmt
  - *Gesucht:* ein oder alle Positionen von Vorkommen des Pattern pat im Text
- Beispiele

Position:	`12345678...`	`12345678...`
Text:	`dieser testtext ist ...`	`aaabaabacabca`
Muster:	`test`	`aaba`

- Maß der Effizienz: Anzahl der (Zeichen-) Vergleiche zwischen Muster und Text

Naiver Algorithmus (Brute Force)

- Brute Force-Lösung 1
  - Rückgabe der ersten Position i an der Muster vorkommt bzw. -1 falls kein Vorkommen

```
FOR i=1 to n -m+1 DO BEGIN
   found := true;
   FOR j=1 to m DO IF text[i] ≠ pat [j] THEN found := false; { Mismatch }
   IF found THEN RETURN i;
END;
RETURN -1;
```

  - Komplexität O((n-m)*m) = O (n*m)
- Brute Force-Lösung 2
  - Abbrechen der Prüfung einer Textposition i bei erstem Mismatch mit dem Muster

```
FOR i=1 to n -m+1 DO BEGIN
   j := 1;
   WHILE j <= m AND pat[j] = text[i+j-1] DO j := j+1 END;
   IF j = m+1 THEN RETURN i;
END
RETURN -1;
```

  - Aufwand oft nur O(n+m)
  - Worst Case-Aufwand weiterhin O(n*m)

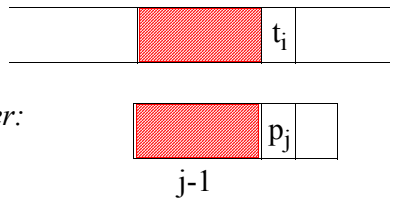
Naiver Algorithmus (2)

```
Text:    der erste testtext ist kurz
Muster:  test
          test
           test
            test
             test
              test
               test
                test
                 test
                  test
                   test
```

- Verschiedene bessere Algorithmen
  - Nutzung der Musterstruktur, Kenntnis der im Muster vorkommenden Zeichen
  - Knuth-Morris-Pratt (1974): nutze bereits geprüfter Musteranfang um ggf. Muster um mehr als eine Stelle nach rechts zu verschieben
  - Boyer-Moore (1976): Teste Muster von hinten nach vorne

Knuth-Morris-Pratt (KMP)

- Idee: nutze bereits gelesene Information bei einem Mismatch
  - verschiebe ggf. Muster um mehr als 1 Position nach rechts
  - gehe im Text nie zurück!
- Allgemeiner Zusammenhang
  - Mismatch an Textposition i mit j-tem Zeichen im Muster
  - j-1 vorhergehende Zeichen stimmen überein
  - mit welchem Zeichen im Muster kann nun das i-te Textzeichen verglichen werden, so daß kein Vorkommen des Musters übersehen wird?

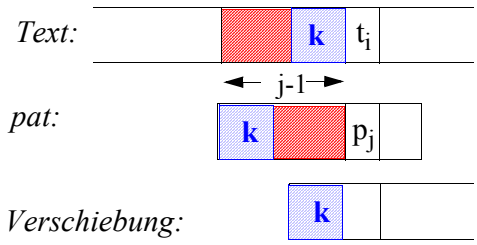


- Beispiele

```
Text:    DATENSTRUKTUREN        GEGEBENENFALLS
Muster:  DATUM                  GEGEN
```

KMP (2)

- Beobachtungen
  - wesentlich ist das längste Präfix des Musters (Länge k < j-1), das Suffix des übereinstimmenden Bereiches ist, d.h. gleich pat [j-k-1.. j-1] ist
  - dann ist Position k+1 = next (j) im Muster, die nächste Stelle, die mit Textzeichen t_i zu vergleichen ist (entspricht Verschiebung des Musters um j-k-1 Positionen)
  - für k=0 kann Muster um j-1 Positionen verschoben werden



- Hilfstabelle *next* spezifiziert die nächste zu prüfende Position des Musters
  - next [j] gibt für Mismatch an Postion j > 1, die als nächstes zu prüfende Musterposition an
  - next[j] = 1 + k (=Länge des längsten echten Suffixes von pat[1..j-1], das Präfix von pat ist)
  - next [1]=0
  - next kann ausschliesslich auf dem Muster selbst (vorab) bestimmt werden
- Beispiel zur Bestimmung der Verschiebetabelle next

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

KMP (3)

- KMP-Suchalgorithmus (setzt voraus, dass next-Tabelle erstellt wurde)

```
j:=1; i:=1;
WHILE (i <= n) DO BEGIN
        IF pat[j]= text[i] DO   BEGIN    IF j=m RETURN i-m+1; // Match
                                         j := j+1; i := i+1;
                                END
        ELSE                    IF j>1 THEN j := next [j]
                                ELSE i := i+1;
END
RETURN -1; // Mismatch
```

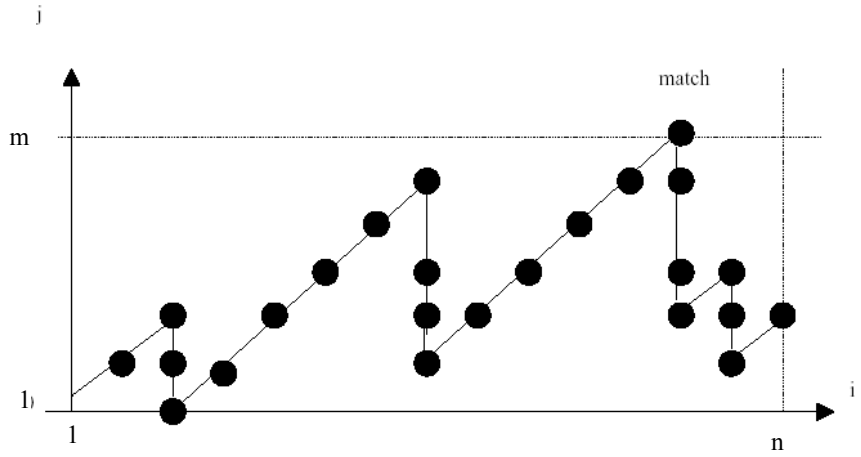
- Beispiel

Text:	A	B	C	A	A	B	A	B	A	A	B	A	B	C
Muster:	A	B	A	B	C									

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

KMP (4)

- Verlauf von i und j bei KMP-Stringsuche



- lineare Worst-Case-Komplexität O(n+m)
  - Suchverfahren selbst O(n)
  - Vorberechnung der next-Tabelle O (m)
- vorteilhaft v.a. bei Wiederholung von Teilmustern

Boyer-Moore

- Auswertung des Musters von rechts nach links, um bei Mismatch Muster möglichst weit verschieben zu können
- Nutzung von im Suchmuster vorhandenen Informationen, insbesondere vorkommenden Zeichen und Suffixen
- Vorkommens-Heuristik („bad character heuristic“)
  - Textposition i wird mit Muster von hinten beginnend verglichen; Mismatch an Muster-Position j für Textsymbol t
  - wenn t im Muster nicht vorkommt (v.a. bei kurzen Mustern sehr wahrscheinlich), kann Muster hinter t geschoben, also um j Positionen
  - wenn t vorkommt, kann Muster um einen Betrag verschoben werden, der der Position des letzten Vorkommens des Symbols im Suchmuster entspricht
  - Verschiebeumfang kann für jeden Buchstaben des Alphabets vorab auf Muster bestimmt und in einer Tabelle vermerkt werden
- Beispiel:

```
Text:    DATENSTRUKTUREN UND ALGORITHMEN ...
Muster:  DATUM           DATUM
```

Boyer-Moore (2)

- Vorberechnung einer Hilfstabelle *last*
  - für jedes Symbol des Alphabets wird die Position seines letzten Vorkommens im Muster angegeben
  - -1, falls das Symbol nicht im Muster vorkommt
  - für Mismatch an Musterposition j, verschiebt sich der Anfang des Musters um j - last [t] +1 Positionen

- Algorithmus

```
i:=1;
WHILE (i <= n-m) DO BEGIN
        j := m;
        WHILE j >= 1 AND pat[j]= text[i+j-1] DO j := j-1;
        IF j < 1        RETURN i;           // Match
        ELSE            i := (i+j-1) - last [text[i+j-1]]);
END;
RETURN -1; // Mismatch
```

- Komplexität:
  - für große Alphabete / kleine Muster wird meist O (n / m) erreicht, d.h zumeist ist nur jedes m-te Zeichen zu inspizieren
  - Worst-Case jedoch O (n*m)

Boyer-Moore: Beispiel

Text: PETER PIPER PICKED A PECK
Muster: PECK

Last-Tabelle:

A:	N:
B:	O:
C:	P:
D:	...
E:	
...	
J:	Y:
K:	Z:
...	...

Boyer-Moore (4)

- weitere Verbesserung durch Match-Heuristik („good suffix heuristic“)
  - Suffix s des Musters stimmt mit Text überein
  - Fall 1: falls s nicht noch einmal im Muster vorkommt, kann Muster um m Positionen weitergeschoben werden
  - Fall 2: es gibt ein weiteres Vorkommen von s im Muster: Muster kann verschoben werden, bis dieses Vorkommen auf den entsprechenden Textteil zu s ausgerichtet ist
  - Fall 3: Präfix des Musters stimmt mit Endteil von s überein: Verschiebung des Musters bis übereinstimmende Teile übereinander liegen

```
Text:    CBABBCBBCABA...        CBABBCBBCABA...
Muster:  ABBABC                 ABCCBC
```

```
Text:    BAABBCABCABA...
Muster:  CBAABC
```

- lineare Worst-Case-Komplexität O (n+m)

Signaturen

- Indirekte Suche über Hash-Funktion
  - Berechnung einer Signatur s für das Muster, z.B. über Hash-Funktion
  - für jedes Textfenster an Position i (Länge m) wird ebenfalls eine Signatur s_i berechnet
  - Falls $s_i = s$ liegt ein potentieller Match vor, der näher zu prüfen ist
  - zeichenweiser Vergleich zwischen Muster und Text wird weitgehend vermieden
- Pessimistische Philosophie
  - "Suchen" bedeutet "Versuchen, etwas zu finden". Optimistische Ansätze erwarten Vorkommen und führen daher viele Vergleiche durch, um Muster zu finden
  - Pessimistische Ansätze nehmen an, daß Muster meist nicht vorkommt. Es wird versucht, viele Stellen im Text schnell auszuschließen und nur an wenigen Stellen genauer zu prüfen
  - Neben Signatur-Ansätzen fallen u.a. auch Verfahren, die zunächst Vorhandensein seltener Zeichen prüfen, in diese Kategorie
- Kosten O (n) falls Signaturen effizient bestimmt werden können
  - inkrementelle Berechnung von s_i aus s_{i-1}
  - unterschiedliche Vorschläge mit konstantem Berechnungaufwand pro Fenster

Signaturen (2)

- Beispiel: Ziffernalphabet; Quersumme als Signaturfunktion

Text: 7 6 2 1 3 0 8 7 2 5 0 8 . . .
– 1 6 –

Muster: 1 3 0 8

Signatur: 1+3+0+8=12

- inkrementelle Berechenbarkeit der Quersumme eines neuen Fensters (Subtraktion der herausfallenden Ziffer, Addition der neuen Ziffer)
- jedoch hohe Wahrscheinlichkeit von Kollisionen (false matches)

- Alternative Signaturfunktion (Karp-Rabin)
  - Abbildung des Musters / Fensters in Dezimalzahl von max. 9 Stellen (mit 32 Bits repräsentierbar)
  - Signatur des Musters: $s\ (p_1, \ldots p_m) = \Sigma_{j=1..m}\ (10^{j-1} \cdot p_{m+1-j})\ \mathrm{mod}\ 10^9$
  - Signatur s_{i+1} des neuen Fensters $(t_{i+1} \ldots t_{i+m})$ abgeleitet aus Signatur s_i des vorherigen Fensters $(t_i \ldots t_{i+m-1})$:

$$s_{i+1} = ((s_i - t_i \cdot 10^{m-1}) \cdot 10 + t_{i+m})\ \mathrm{mod}\ 10^9$$

  - Signaturfunktion ist auch für größere Alphabete anwendbar

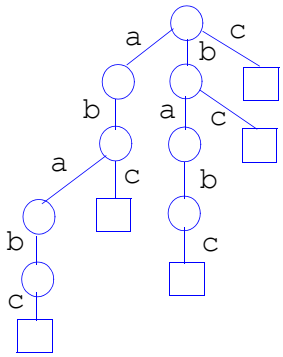
Statische Suchverfahren

- Annahme: weitgehend statische Texte / Dokumente
  - derselbe Text wird häufig für unterschiedliche Muster durchsucht
- Beschleunigung der Suche durch Indexierung (Suchindex)
- Vorgehensweise bei
  - Information Retrieval-Systemen zur Verwaltung von Dokumentkollektionen
  - Volltext-Datenbanksystemen
  - Web-Suchmaschinen etc.
- Indexvarianten
  - (Präfix-) B*-Bäume
  - Tries, z.B. Radix oder PATRICIA Tries
  - Suffix-Bäume
  - Invertierte Listen
  - Signatur-Dateien

Suffix-Bäume

- Suffix-Bäume: Digitalbäume, die alle Suffixe einer Zeichenkette bzw. eines Textes repräsentieren
- Unterstütze Operationen:
  - Teilwortsuche: in O (m)
  - Präfix-Suche: Bestimmung aller Positionen, an denen Worte mit einem Präfix p auftreten
  - Bereichssuche: Bestimmung aller Positionen von Worten, die in der lexikographischen Ordnung zwischen zwei Grenzen p1 und p2 liegen
- Suffix-Tries basierend auf Tries

Text: ababc
5 Suffixe: ababc
babc
abc
bc
c

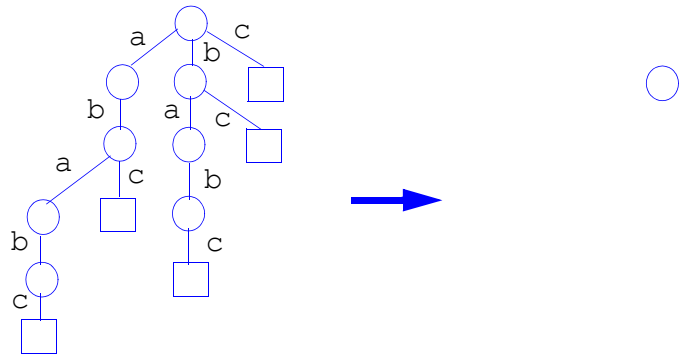


  - hoher Platzbedarf für Suffix-Tries $O(n^2)$ -> Kompaktierung durch Suffix-Bäume

Suffix-Bäume (2)

- alle Wege im Trie, die nur aus unären Knoten bestehen, werden zusammengezogen

Text: ababc
Suffixe: ababc
babc
abc
bc
c



- Eigenschaften für Suffix-Baum S
  - jede Kante in S repräsentiert nicht-leeres Teilwort des Eingabetextes T
  - die Teilworte von T, die benachbarten Kanten in S zugeordnet sind, beginnen mit *verschiedenen* Buchstaben
  - jeder innerer Knoten von S (außer der Wurzel) hat wenigstens zwei Söhne
  - jedes Blatt repräsentiert ein nicht-leeres Suffix von T
- linearer Platzbedarf O(n): n Blätter und höchsten n-1 innere Knoten

Suffix-Bäume (3)

- Vorgehensweise bei Konstruktion
  - beginnend mit leerem Baum T_0 wird pro Schritt Suffix $suff_i$ beginnend an Textposition i eingefügt und Suffix-Baum T_{i-1} nach T_i erweitert
  - zur Einfügung ist $head_i$ zu bestimmen, d.h. längstes Präfix von $suff_i$, das bereits im Baum präsent ist, d.h. das bereits Präfix von $suff_j$ ist (j < i)

Text: ababc

T_0 =

T_1 = ababc

T_2 = ababc babc

T_3 =

$suff_3$ = abc

$head_3$ =

$tail_3$ =

- naiver Algorithmus: O (n^2)
- linearer Aufwand O (n) gemäß Konstruktionsalgorithmus von McCreight
  - Einführung von Suffix-Zeigern
  - Einzelheiten siehe Ottmann/Widmayer (2001)

Invertierte Listen

- Nutzung vor allem zur Textsuche in Dokumentkollektionen
  - nicht nur 1 Text/Sequenz, sondern beliebig viele Texte / Dokumente
  - Suche nach bestimmten Wörtern/Schlüsselbegriffen/Deskriptoren, nicht nach beliebigen Zeichenketten
  - Begriffe werden ggf. auf Stammform reduziert; Elimination sogenannnter „Stop-Wörter“ (der, die, das, ist, er ...)
  - klassische Aufgabenstellung des Information Retrieval
- Invertierung: Verzeichnis (Index) aller Vorkommen von Schlüsselbegriffen
  - lexikographisch sortierte Liste der vorkommenden Schlüsselbegriffe
  - pro Eintrag (Begriff) Liste der Dokumente (Verweise/Zeiger), die Begriff enthalten
  - eventuell zusätzliche Information pro Dokument wie Häufigkeit des Auftretens oder Position der Vorkommen
- Beispiel 1: Invertierung eines Textes

```
1         10        20
Dies ist ein Text. Der Text hat viele
Wörter. Wörter bestehen aus ...
38                  53
```

Invertierter Index

Begriff	Vorkommen
bestehen	53
Dies	1
Text	14, 24
viele	33
Wörter	38, 46

Invertierte Listen (2)

- Beispiel 2: Invertierung mehrerer Texte / Dokumente

d1

Dieses objektorientierte Datenbanksystem unterstützt nicht nur multimediale Objekte, sondern ist überhaupt phänomenal.

d2

Objektorientierte Systeme unterstützen die Vererbung von Methoden.

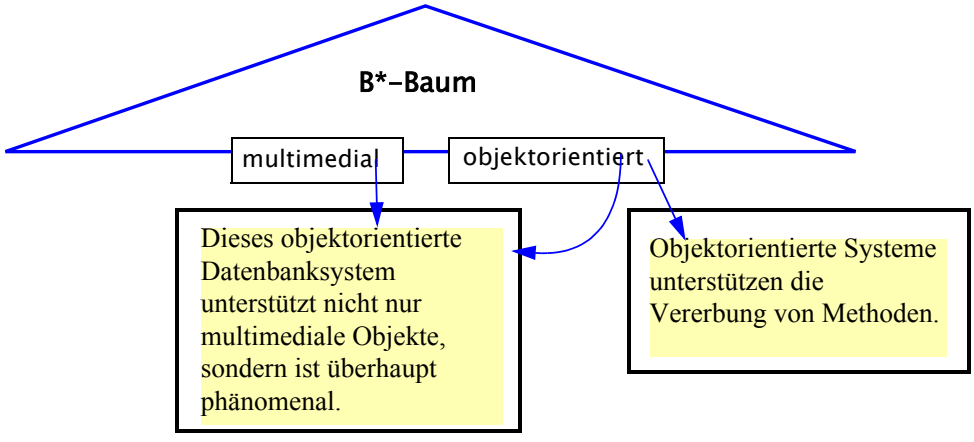
Invertierter Index

Begriff	Vorkommen
Datenbanksystem	d1
Methode	d2
multimedial	d1
objektorientiert	d1, d2
phänomenal	d1
System	d2
überhaupt	d1
unterstützen	d2
Vererbung	d2

- Zugriffskosten werden durch Datenstruktur zur Verwaltung der invertierten Liste bestimmt
  - B*-Baum
  - Hash-Verfahren ...

Invertierte Listen (3)

- effiziente Realisierung über (indirekten) B*-Baum
  - variabel lange Verweis/Zeigerlisten pro Schlüssel auf Blattebene



- Boole'sche Operationen: Verknüpfung von Zeigerlisten
  - Beispiel: Suche nach Dokumenten mit „multimedial"“ UND „objektorientiert"

Signatur-Dateien

- Alternative zu invertierten Listen: Einsatz von ***Signaturen***
  - zu jedem Dokument bzw. Textfragment wird Bitvektor fester Länge (*Signatur*) geführt
  - Begriffe werden über Signaturgenerierungsfunktion (Hash-Funktion) s auf Bitvektor abgebildet
  - OR-Verknüpfung der Bitvektoren aller im Dokument bzw. Textfragment vorkommenden Begriffe ergibt *Dokument- bzw. Fragment*-Signatur
- Signaturen aller Dokumente/Fragmente werden sequentiell gespeichert (bzw. in speziellem Signaturbaum)

```
s (bestehen)   = 000101
s (Text)       = 110000
s (bestehen)   = 100100
s (viele)      = 001100
s (Wörter)     = 100001
```

Signatur-File

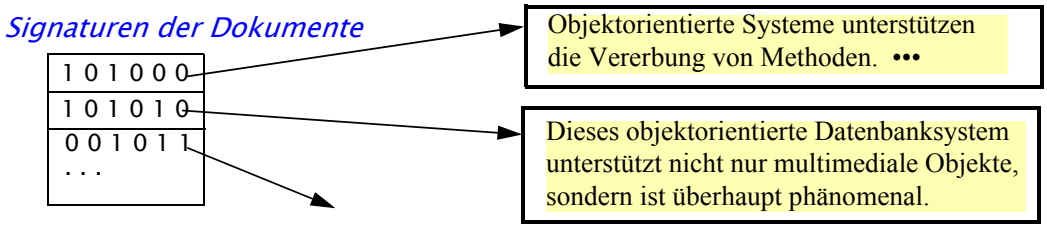
110001	→ Dies ist ein Text.
111101	→ Der Text hat viele Wörter.
100101	→ Wörter bestehen aus ...

- Suchbegriff wird über dieselbe Signaturgenerierungsfunktion s auf eine *Anfragesignatur* abgebildet
  - mehrere Suchbegriffe können einfach zu einer Anfragesignatur kombiniert werden (OR, AND, NOT-Verknüpfung der Bitvektoren)
  - wegen Nichtinjektivität der Signaturgenerierungsfunktion muß bei ermittelten Dokumenten/ Fragmenten geprüft werden, ob tatsächlich ein Treffer vorliegt

Signatur-Dateien (2)

- Beispiel bezüglich mehrerer Dokumente
  - Signaturgenerierungsfunktion:

    objektorientiert / multimedial / Datenbanksystem / Vererbung -> Bit 0 / 2 / 4 / 2



  - Anfrage: Dokumente mit Begriffen "objektorientiert" und "multimedial"

    Anfragesignatur:

- Eigenschaften
  - geringer Platzbedarf für Dokumentsignaturen
  - Zugriffskosten aufgrund Nachbearbeitungsaufwand bei False Matches meist höher als bei invertierten Listen

Approximative Suche

- Ähnlichkeitssuche erfordert Maß für die Ähnlichkeit zwischen Zeichenketten s1 und s2, z.B.
  - *Hamming-Distanz:* Anzahl der Mismatches zwischen s1 und s2 (s1 und s2 haben gleiche Länge)
  - *Editierdistanz:* Kosten zum Editieren von s1, um s2 zu erhalten (Einfüge-, Lösch-, Ersetzungsoperationen)

s1:	AGCAA	AGCACACA
s2:	ACCTA	ACACACTA

Hamming-Distanz:

- *k-Mismatch-Suchproblem*
  - Gesucht werden alle Vorkommen eines Musters in einem Text, so daß höchstens an k der m Stellen des Musters ein Mismatch vorliegt, d.h. Hamming-Distanz ≤ k
  - exakte Stringsuche ergibt sich als Spezialfall mit k=0

- Beispiel (k=2)

Text: erster testtext
Muster: test

k=2

Approximative Suche (2)

- Naiver Such-Algorithmus kann für k-Mismatch-Problem leicht angepasst werden

```
FOR i=1 to n -m+1 DO BEGIN
   z := 1;
   FOR j=1 to m DO IF text[i] ≠ pat [j] THEN z :=z+1;{ Mismatch }
   IF z <= k THEN write („Treffer an Position “, i, „ mit “, z, „ Mismatches“);
END;
RETURN -1;
```

  - analoges Vorgehen, um Sequenz mit geringstem Hamming-Abstand zu bestimmen
- Komplexität O(n*m)
- effzientere Suchalgorithmen (KMP, BM ...) können analog angepaßt werden
- Editierdistanz oft geeigneter als Hamming-Distanz
  - anwendbar für Sequenzen unterschiedlicher Länge
  - Hamming-Distanz ist Spezialfall ohne Einfüge-/Löschoperationen (Anzahl der Ersetzungen)
  - Bioinformatik: Vergleich von DNA-Sequenzen auf Basis der Editier (Evolutions)-Distanz

Editierdistanz

- 3 Arten von Editier-Operationen: *Löschen* eines Zeichens, *Einfügen* eines Zeichens und *Ersetzen* eines Zeichens x durch ein anderes Zeichen y
- Einfügeoperationen korrespondieren zu je einer Mismatch-Situation zwischen s1 und s2, wobei „-“ für leeres Wort bzw. Lücke (gap) steht:
  - (-, y) Einfügung von y in s2 gegenüber s1
  - (x, -) Löschung von x in s1
  - (x, y) Ersetzung von x durch y
  - (x, x) Match-Situation (keine Änderung)
- jeder Operation wird Gewicht bzw. Kosten w (x,y) zugewiesen
- *Einheitskostenmodell*: w (x, y) = w (- , y) = w (x, -) = 1; w (x, x) = 0
- *Editierdistanz D (s1,s2):* Minimale Kosten, die Folge von Editier-Operationen hat, um s1 nach s2 zu überführen
  - bei Einheitskostenmodell spricht man auch von *Levenshtein-Distanz*
  - im Einheitskostenmodell gilt von s1 und s2: $D(s1,s2)=D(s2,s1)$ und für Kardinalitäten n und m $abs(n-m) \le D(s1,s2) \le max(m,n)$
- Beispiel: Editier-Distanz zwischen „Auto“ und „Rad“ ?

Editierdistanz in der Bioinformatik†

- Bestimmung eines *Alignments* zweier Sequenzen s1 und s2:
  - Übereinanderstellen von s1 und s2 und durch Einfügen von Gap-Zeichen Sequenzen auf dieselbe Länge bringen: Jedes Zeichenpaar repräsentiert zugehörige Editier-Operation
  - Kosten des Alignment: Summe der Kosten der Editier-Operationen
  - *optimales Alignment*: Alignment mit minimalen Kosten (= Editierdistanz)

s1:	AGCACACA	AGCACAC-A	AG-CACACA
s2:	ACACACTA	A-CACACTA	ACACACT-A
	Match (A,A) Replace (G,C) Replace (C,A) Replace (A,C) Replace (C,A) Replace (A,C) Replace (C,T) Match (A,A)	Match (A,A) Delete (G, -) Match (C,C) Match (A,A) Match (C.C) Match (A,A) Match (C,C) Insert (-,T) Match (A,A)	Match (A,A) Replace (G,C) Insert (-, A) Match (C, C) Match (A,A) Match (C,C) Replace (A,T) Delete (C,-) Replace (A,A)

† www.techfak.uni-bielefeld.de/bcd/Curric/PrwAli/node2.html

Editierdistanz (3)

- Problem 1: Berechnung der Editierdistanz
  - berechne für zwei Zeichenketten / Sequenzen s1 und s2 möglichst effizient die Editierdistanz D(s1,s2) und eine kostenminimale Folge von Editier-Operationen, die s1 in s2 überführt
  - entspricht Bestimmung eines optimalen Alignments
- Problem 2: Approximate Suche
  - suche zu einem (kurzen) Muster p alle Vorkommen von Strings p' in einem Text, so daß die Editierdistanz D (p,p') <= k ist, für ein vorgegebenes k
  - Spezialfall 1: exakte Stringsuche (k=0)
  - Spezialfall 2: k-Mismatch-Problem, falls nur Ersetzungen und keine Einfüge- oder Lösch-Operationen zugelassen werden
- Variationen von Problem 2
  - Suche zu Muster/Sequenz das ähnlichste Vorkommen (lokales Alignment)
  - bestimme zwischen 2 Sequenzen s1 und s2 die ähnlichsten Teilsequenzen s1' und s2'

Berechnung der Editierdistanz

- Nutzung folgender Eigenschaften zur Begrenzung zu prüfender Editier-Operationen
  - optimale Folge von Editier-Operationen ändert jedes Zeichen höchstens einmal
  - jede Zerlegung einer optimalen Anordnung führt zur optimalen Anordnung der entsprechenden Teilsequenzen

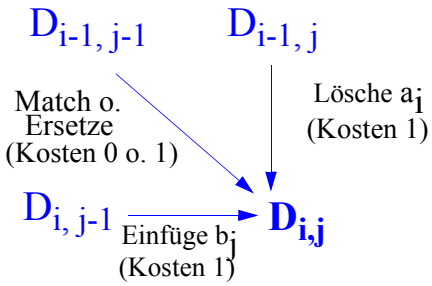
```
AGCACAC-A
A-CACACTA
```

- Lösung des Optimierungsproblems durch Ansatz der *dynamischen Programmierung*
  - Konstruktion der optimalen Gesamtlösung durch rekursive Kombination von Lösungen für Teilprobleme
- Sei $s_1 = (a_1, \ldots a_n)$, $s_2=(b_1, \ldots b_m)$.
  D_{ij} sei Editierdistanz für Präfixe $(a_1, \ldots a_i)$ und $(b_1, \ldots b_j)$; $0 \le i \le n$; $0 \le j \le m$
  - D_{ij} kann ausschließlich aus $D_{i-1,j}$, $D_{i,j-1}$ und $D_{i-1,j-1}$ bestimmt werden
  - es gibt triviale Lösungen für $D_{0,0}$, $D_{0,j}$, $D_{i,0}$
  - Eintragung der $D_{i,j}$ in (n+1, m+1)-Matrix
  - Editierdistanz zwischen s1 und s2 insgesamt ergibt sich für i=n, j=m
  - es wird hier nur das Einheitskostenmodell angenommen

Berechnung der Editierdistanz (2)

- Editierdistanz D_{ij} für i=0 oder j=0
  - $D_{0,0} = D(-, -) = 0$
  - $D_{0,j} = D(-, (b_1,.., b_j)) = j$ // j Einfügungen
  - $D_{i,0} = D((a_1,.., a_i), -) = i$ // i Löschungen
- Editierdistanz D_{ij} für i>0 und j>0 kann aus günstigstem der folgenden Fälle abgeleitet werden:
  - Match oder Ersetze:
    falls $a_i = b_j$ (Match): $D_{i,j} = D_{i-1,j-1}$;
    falls $a_i \neq b_j$: $D_{i,j} = 1 + D_{i-1,j-1}$
  - Lösche a_i: $D_{i,j} = D((a_1,.., a_{i-1}), (b_1,.., b_j)) + 1 = D_{i-1,j} + 1$
  - Einfüge b_j: $D_{i,j} = D((a_1,.., a_i), (b_1,.., b_{j-1})) + 1 = D_{i,j-1} + 1$

  Somit ergibt sich:



$$D_{i,j} = \min\left(D_{i-1,j-1} + \begin{cases} 0 \text{ falls } a_i = b_j \\ 1 \text{ falls } a_i \neq b_j \end{cases}, D_{i-1,j} + 1, D_{i,j-1} + 1 \right)$$

Berechnung der Editierdistanz (3)

- Beispiele

i \ j		0	1	2	3
		–	R	A	D
0	–				
1	A				
2	U				
3	T				
4	O				

	–	A	C	A	C	A	C	T	A
–	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	2	3	4	5	6	7
C	3	2	1	2	2	3	4	5	6
A	4	3	2	1	2	2	3	4	5
C	5	4	3	2	1	2	2	3	4
A	6	5	4	3	2	1	2	3	3
C	7	6	5	4	3	2	1	2	3
A	8	7	6	5	4	3	2	2	2

- jeder Weg von links oben nach rechts unten entspricht einer Folge von Edit-Operationen, die s1 in s2 transformiert
  - ggf. mehrere Pfade mit minimalen Kosten
- Komplexität: O (n*m)

Zusammenfassung

- naive Textsuche
  - einfache Realisierung ohne vorzuberechnende Hilfsinformationen
  - Worst Case O (n*m), aber oft linearer Aufwand O(n+m)
- schnellere Ansätze zur dynamischen Textsuche
  - Vorverarbeitung des Musters, jedoch nicht des Textes
  - Knuth-Morrison-Pratt: linearer Worst-Case-Aufwand O (n+m), aber oft nur wenig besser als naive Textsuche
  - Boyer-Moore: Worst-Case O(n*m) bzw. O(n+m), aber im Mittel oft sehr schnell O (n/m)
  - Signaturen: O (n)
- Indexierung erlaubt wesentlich schnellere Suchergebnisse
  - Vorverarbeitung des Textes bzw. der Dokumentkollektionen
  - hohe Flexibilität von Suffixbäumen (Probleme: Größe; Externspeicherzuordnung)
  - Suche in Dokumentkollektionen mit invertierten Listen oder Signatur-Dateien
- Approximative Suche
  - erfordert Ähnlichkeitsmaß, z.B. Hamming-Distanz oder Editierdistanz
  - Bestimmung der optimalen Folge von Editier-Operationen sowie Editierdistanz über dynamische Programmierung; O(n*m)