Algorithmen und Datenstrukturen 2

Sommersemester 2006
2. Vorlesung

Peter F. Stadler

Universität Leipzig
Institut für Informatik
studla@bioinf.uni-leipzig.de

Wdhlg.: 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 wie viele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden

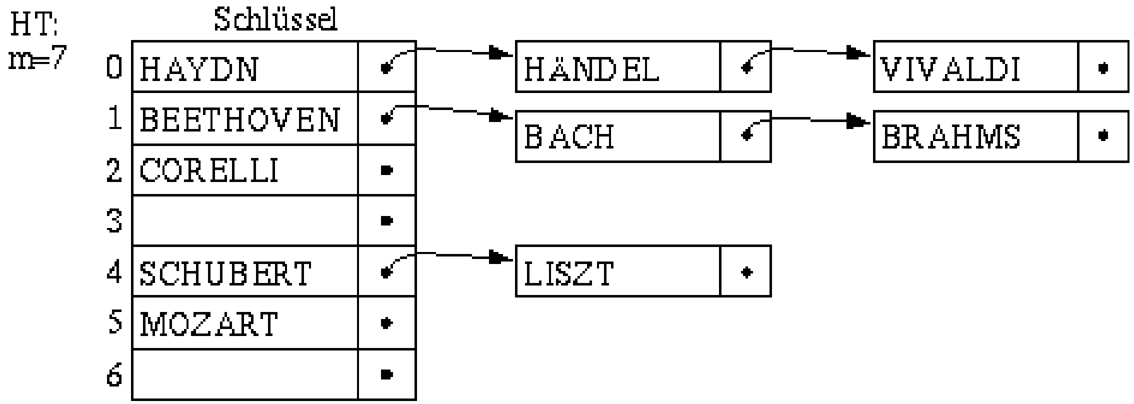
Adressfolge bei Speicherung und Suche für Schlüssel K_p sei $h_0(K_p)$, $h_1(K_p)$, $h_2(K_p)$, ...

- Bei einer Folge der Länge n treten also n-1 Kollisionen auf
- Primärkollision: $h(K_p) = h(K_q)$
- Sekundärkollision: $h_i(K_p) = h_j(K_q)$, $i \neq j$

Wdhlg.: 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 !



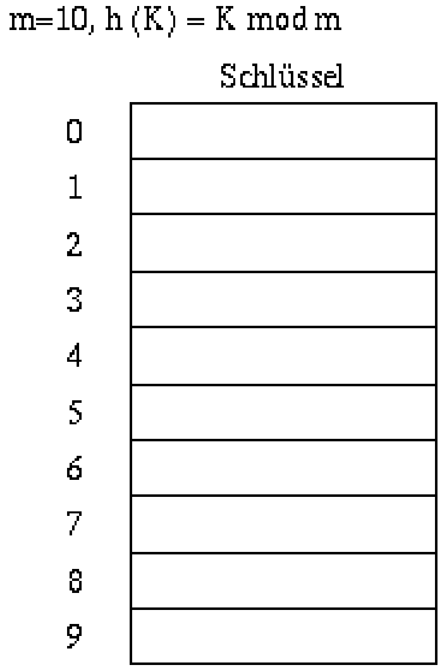
Entartung zur linearen Liste prinzipiell möglich

Nachteil: Anlegen von Überläufern, auch wenn Hash-Tabelle (Primärbereich) noch wenig belegt ist

Wdhlg.: Offene Hash-Verfahren: Lineares Sondieren

Eigenschaften

- 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:

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

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

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

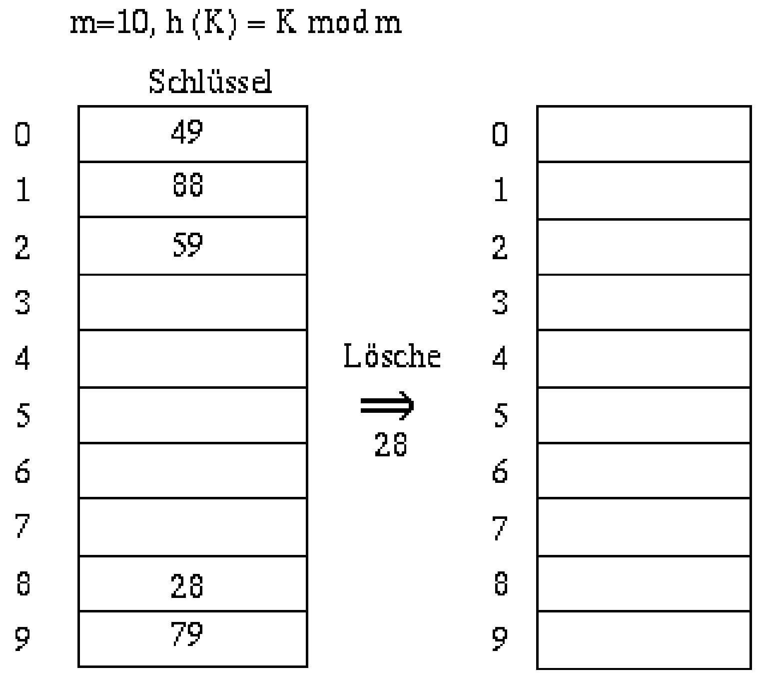
- Häufung von Kollisionen durch "Klumpenbildung"

  => lange Sondierungsfolgen möglich

Wdhlg.: Lineares Sondieren (2)

Aufwendiges Löschen

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



Lineares Sondieren (3)

Verbesserung: Modifikation
der Überlauffolge

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + f(i)) \bmod m \qquad \text{oder}$$
$$h_i(K_p) = (h_0(K_p) + f(i,h(K_p))) \bmod m, \quad i = 1, 2, \ldots$$

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 ist wichtig:

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

Beispiel:

Einfügereihenfolge 79, 28, 49, 88, 59

$m=10$,
$h(K) = K \bmod m$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

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:

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

Double Hashing

Einsatz einer zweiten Funktion für die Sondierungsfolge

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

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 I

Kostenmaße

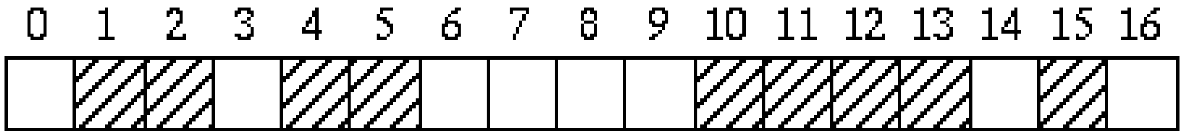
- ß = n/m: Belegung von HT mit n Schlüsseln
- S_n = # der Suchschritte für das Auffinden eines Schlüssels - entspricht den Kosten für erfolgreiche Suche und Löschen (ohne Reorganisation)
- U_n = # der Suchschritte für die erfolglose Suche - das Auffinden des ersten freien Platzes - entspricht den Einfügekosten

Grenzwerte

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

Analyse des Hashing II: Modell für das lineare Sondieren

- Sobald ß 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 (bei Platz 3 und 14), so dass durch neue Schlüssel eine Art Verdopplung der Listenlänge eintreten kann
- Ergebnisse für das lineare Sondieren nach Knuth:

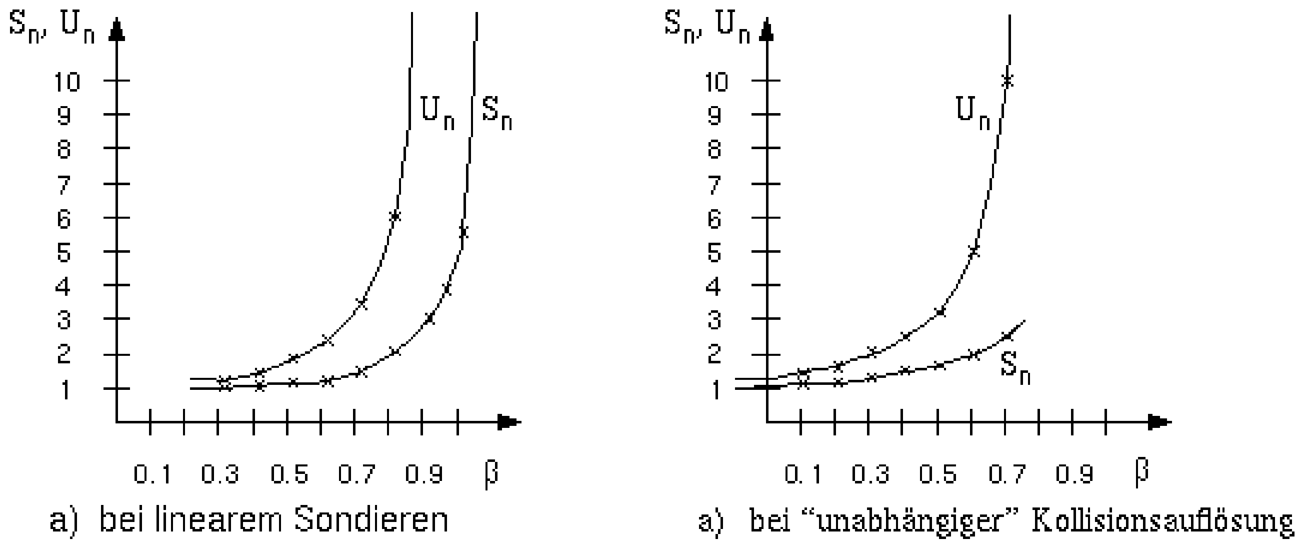
$$S_n \approx 0{,}5\left(1 + \frac{1}{1-\beta}\right) \quad \text{mit} \quad 0 \le \beta = \frac{n}{m} < 1 \qquad U_n \approx 0{,}5\left(1 + \frac{1}{(1-\beta)^2}\right)$$

Analyse des Hashing III

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

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

Anzahl der Suchschritte in HT



Analyse des Hashing IV: Modell für separate Überlaufbereiche

Annahme: n Schlüssel verteilen sich gleichförmig über die m mögl. Ketten.

- Jede Synonymkette hat also im Mittel n/m = 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: $S_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{n-1}{2 \cdot m} \approx 1 + \frac{\beta}{2}$

- Erfolglose Suche: es muss immer die ganze Kette durchlaufen werden

  U_n = 1 + 1 · WS (zu einer Hausadresse existiert 1 Überläufer)+ 2 · 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 (ß>1) erhält man bei separater Kettung noch günstige Werte

Hashing auf Externspeichern I

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

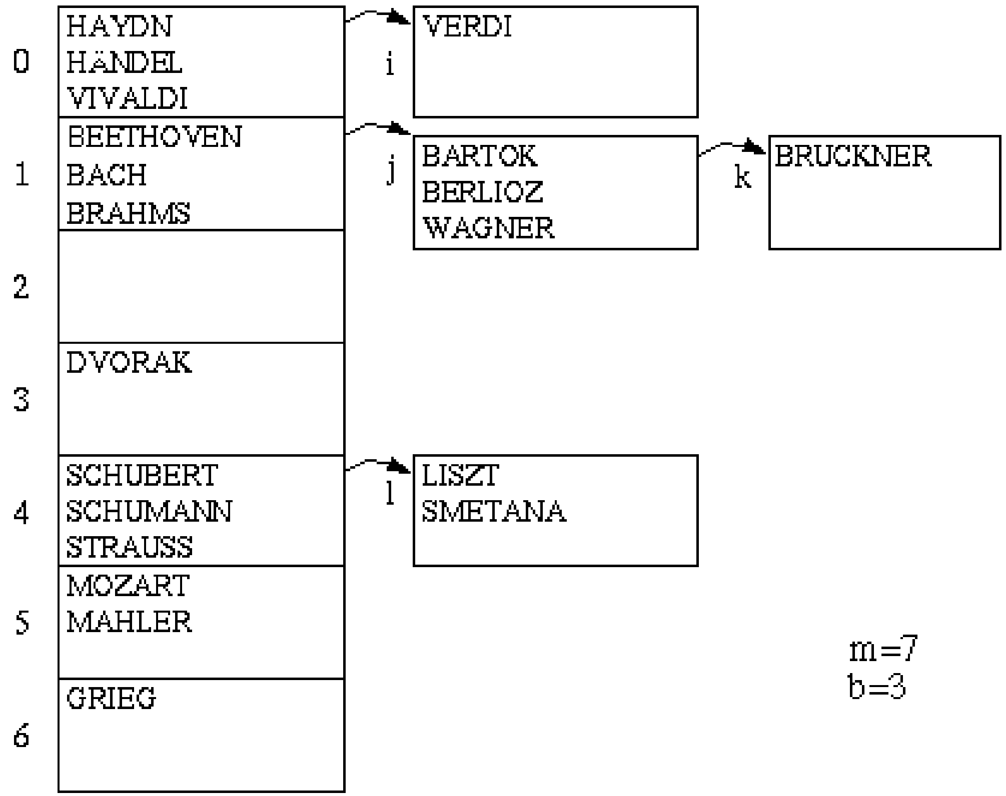
Speicherungsreihenfolge im Bucket

- ohne Ordnung (Einfügefolge)
- nach der Sortierfolge des Schlüssels: aufwendiger, jedoch Vorteile beim Suchen (sortierte Liste!)

Hashing auf Externspeichern II

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



Bucket-Adressierung mit separaten Überlauf-Buckets

- weithin eingesetztes Hash-Verfahren für Externspeicher
- jede Kollisionsklasse hat eine separate Überlaufkette.

Hashing auf Externspeichern III

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

Hashing auf Externspeichern IV

Belegungsfaktor:

ß = n / (b · m)

- bezieht sich auf Primär-Buckets (kann größer als 1 werden!)

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

Zugriffsfaktoren

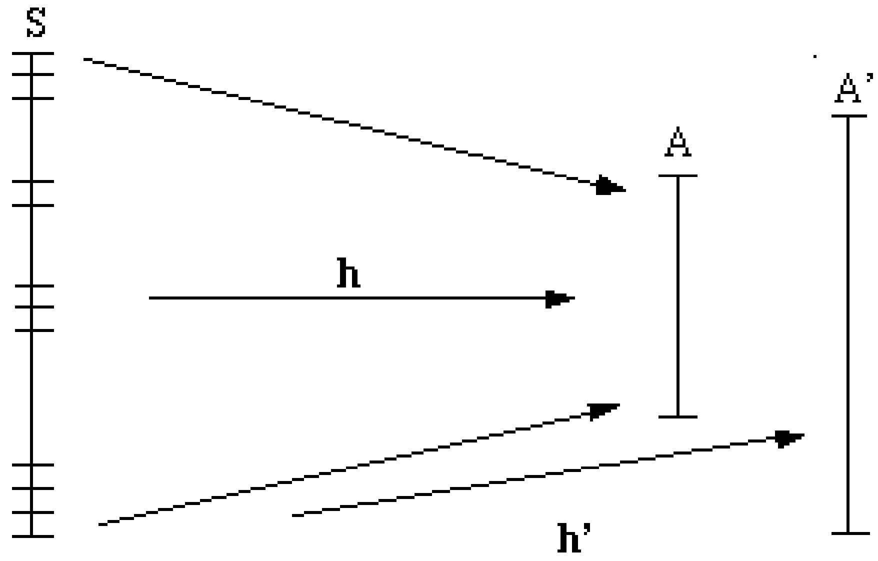
- Gute Annäherung an idealen Wert
- Bei Vergleich mit Mehrwegbäumen ist zu beachten, daß Hash-Verfahren sortiert sequentielle Verarbeitung aller Sätze nicht unterstützen. Außerdem stellen sie statische Strukturen dar. Die Zahl der Primär-Buckets m lässt sich nicht dynamisch an die Zahl der zu speichernden Sätze n anpassen.

Dynamische Hash-Verfahren I

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

Dynamische Hash-Verfahren II

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)

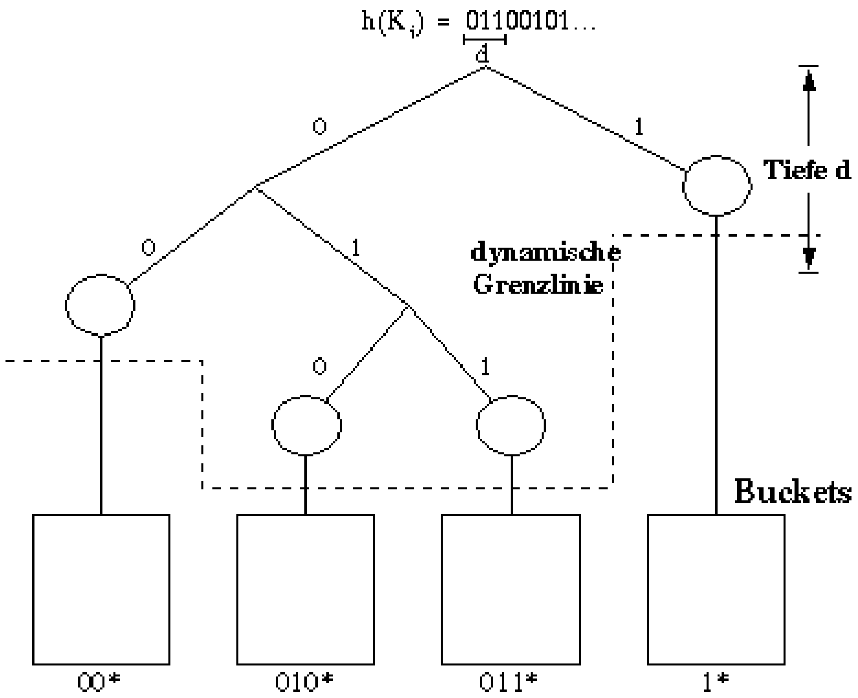
Erweiterbares Hashing I

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

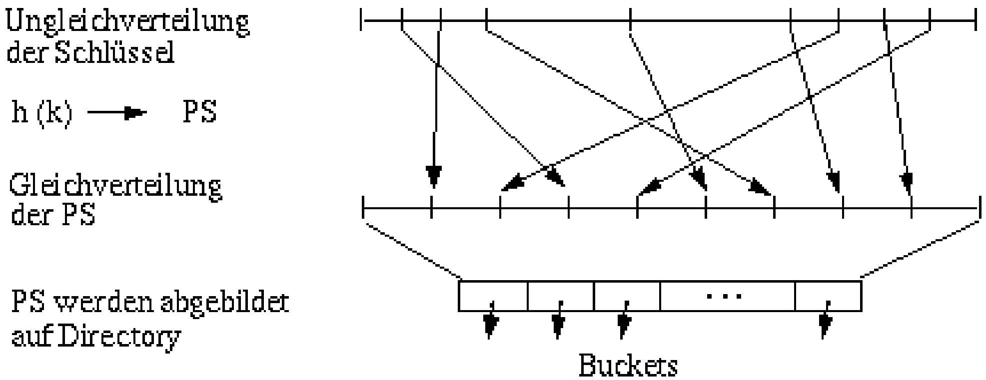
- Die einzelnen Bits eines Schlüssels steuern der Reihe nach den Weg durch den zur Adressierung benutzten Digitalbaum $K_i = (b_0, b_1, b_2, ...)$
- Verwendung der Schlüsselwerte kann bei Ungleichverteilung zu unausgewogenem Digitalbaum führen (Digitalbäume kennen keine Höhenbalancierung!)
- Verwendung von $h(K_i)$ als sog. Pseudoschlüssel (PS) soll bessere Gleichverteilung gewährleisten. $h(K_i) = (b_0, b_1, b_2, ...)$
- Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann



Erweiterbares Hashing II

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})



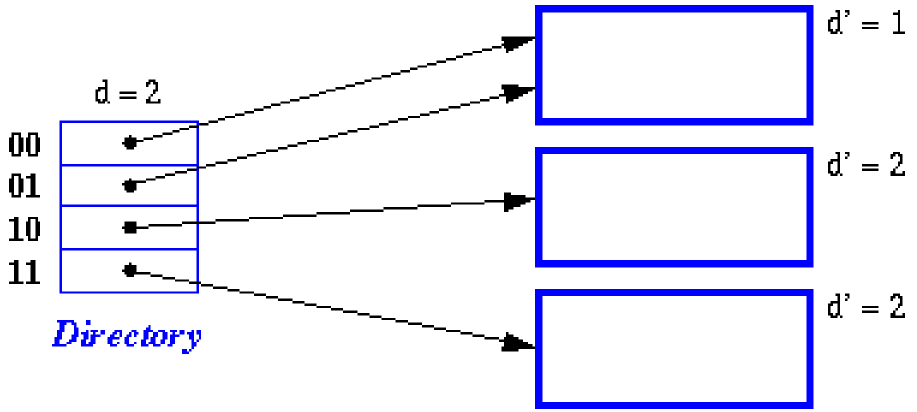
Dynamisches W

- Buckets werden erst bei Bedarf bereitgestellt: kein statisch dimensionierter Primärbereich, keine Überlauf-Buckets
- nur belegte Buckets werden gespeichert
- hohe Speicherplatzbelegung möglich

Erweiterbares Hashing III

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

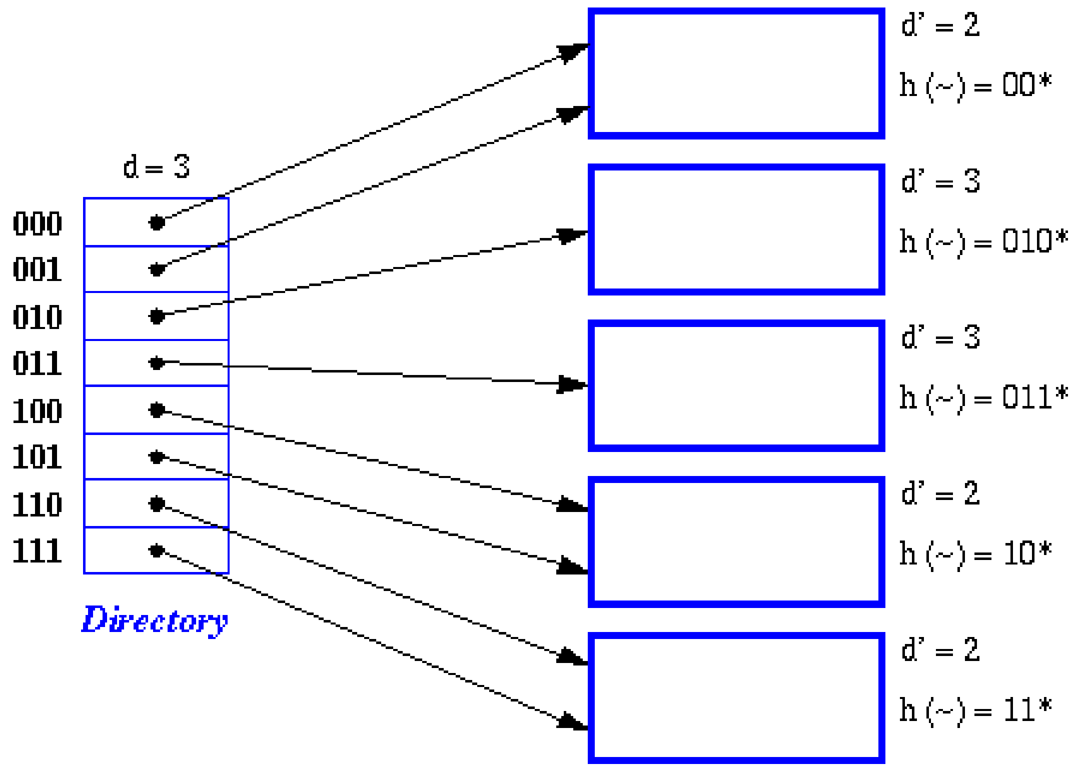


Splitting von Buckets I

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

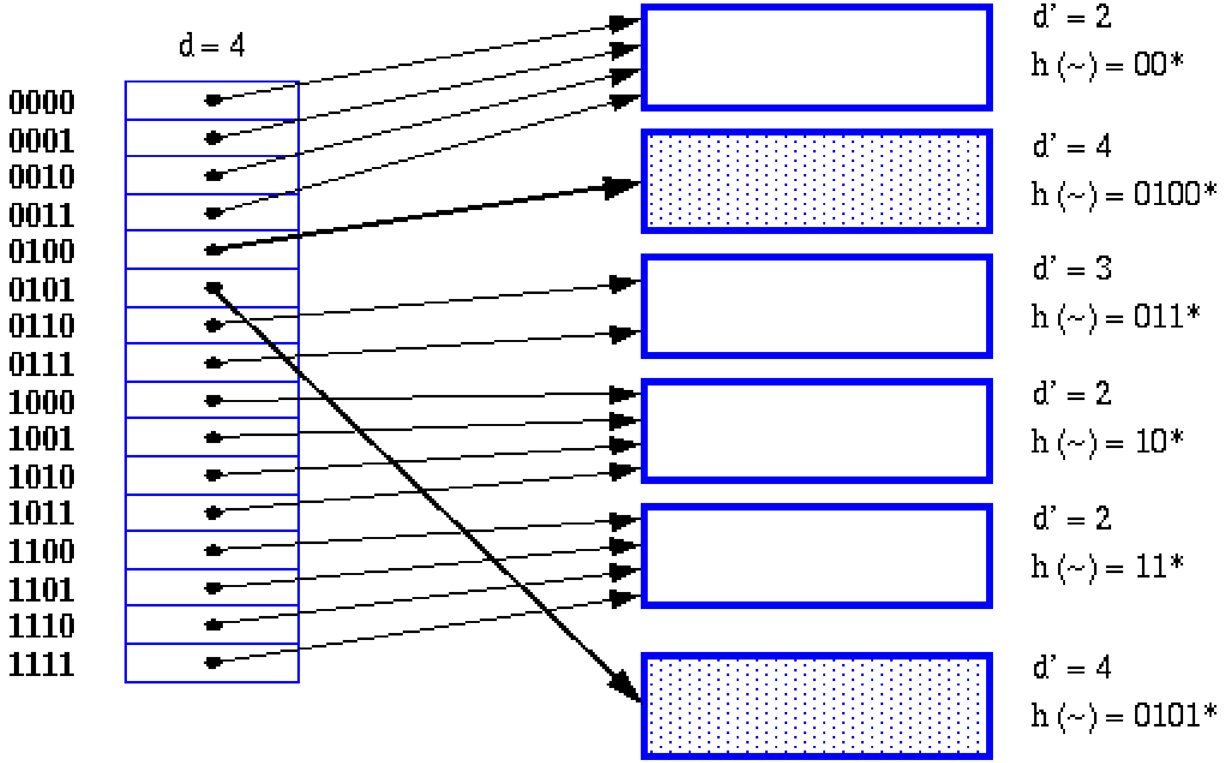


Splitting von Buckets II

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 I

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: $m \cdot 2^L$
- Split-Zeiger p (Initialwert 0) zeigt auf nächstes zu splittende Bucket im Hash-Bereich mit $0 <= p < m \cdot 2^L$
- Split führt zu neuem Bucket mit Adresse $p + m \cdot 2^L$; p wird um 1 inkrementiert $p:=p+1 \bmod (m \cdot 2^L)$
- wenn p wieder auf 0 gesetzt wird (Verdoppelung des Hash-Bereichs beendet), wird L um 1 erhöht

Lineares Hashing II

Hash-Funktion

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

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

z.B. $h_j(k) = k \bmod m \cdot 2^j$

- 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 $h(k) + m \cdot 2^L$)

- gleiche Wahrscheinlichkeit für beide Fälle erwünscht

Beispiel für Lineares Hashing I

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

Schwellwert $ß_S = 0{,}8$

p

0	1	2	3	4
105	111	512	413	144
790	076	477	243	
335		837		
995		002		

Primärbereich

0	1	2	3	4
055		117		
010				

Überlaufsätze

h_0 h_0 h_0 h_0 h_0

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

0	1	2	3	4	5
	111	512	413	144	
	076	477	243		
		837			
		002			

0	1	2	3	4	5
		117			

h_0 h_0 h_0 h_0

Beispiel für Lineares Hashing II

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

0	1	2	3	4	5
790	111	512	413	144	105
010	076	477	243		335
		837	888		995
		002			055

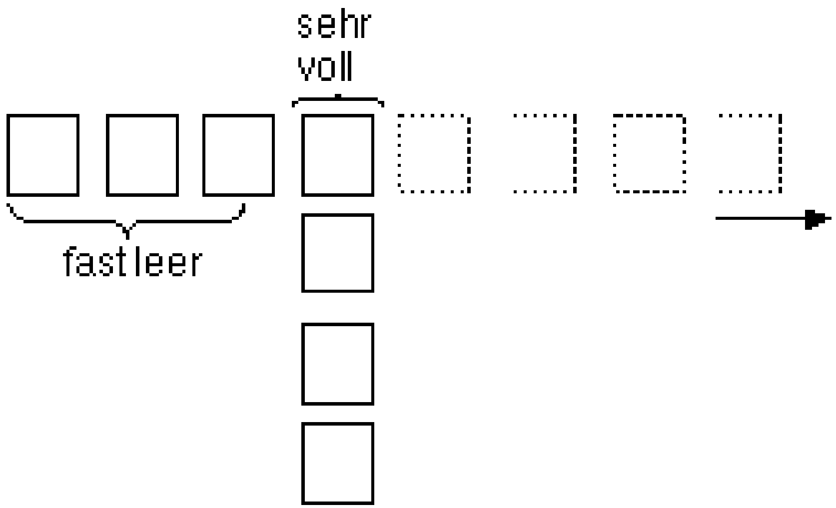
117

h_0 h_0 h_0 h_0

Lineares Hashing: Bewertung

Überläufer weiterhin erforderlich

ungünstiges Split-Verhalten / ungünstige Speicherplatznutzung möglich
(Splitten unterbelegter Seiten)



Zugriffskosten 1 + x

Zusammenfassung I

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)

Zusammenfassung II

Hash-Funktion

- Standard: Divisionsrest-Verfahren
- ggf. zunächst numerischer Wert aus Schlüsseln zu erzeugen
- Verwendung einer Primzahl für Divisor (Größe der Hash-Tabelle) wichtig

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 <0.85 dringend zu empfehlen

Zusammenfassung III

Hash-Verfahren für Externspeicher

- reduzierte Kollisionsproblematik, da Bucket b Sätze aufnehmen kann
- direkte Suche ~ 1 + δ 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)"