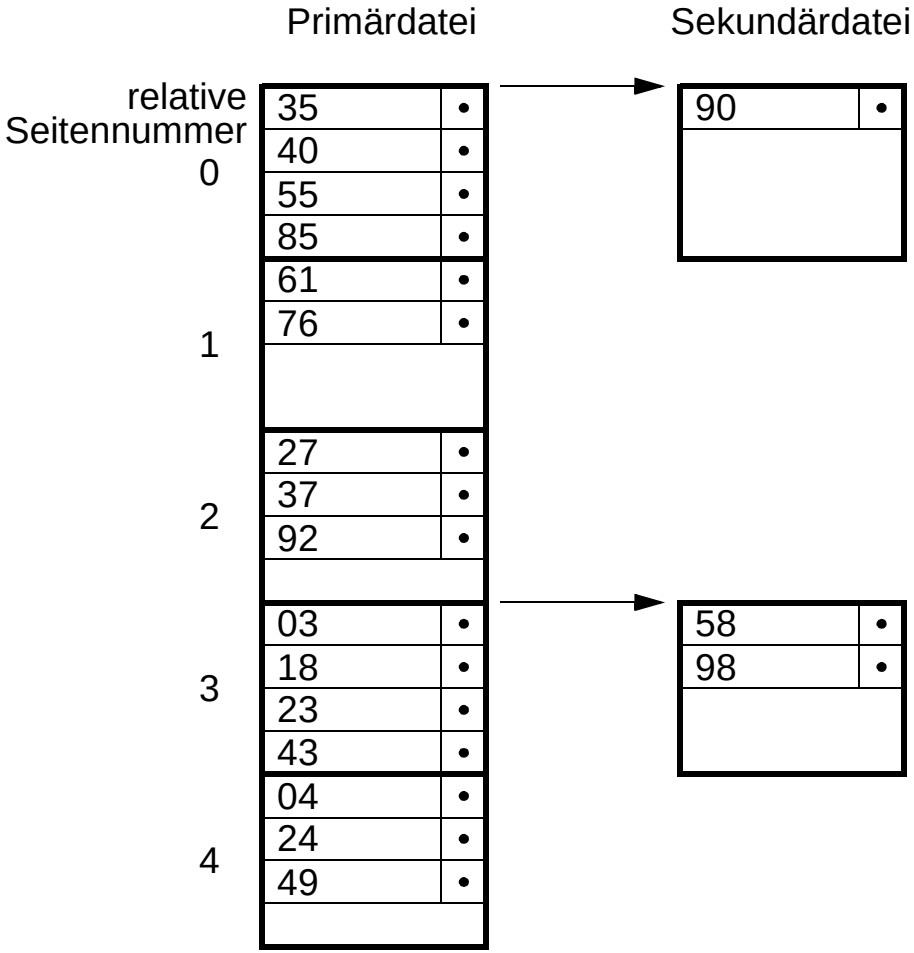
3.2 Statische Hash-Verfahren

- Direkte Berechnung der Speicheradresse (Seitenadresse) eines Satzes über einen Pseudoschlüssel, der durch eine Schlüsseltransformation erzeugt wird.
- Hash-Funktion h: S $\rightarrow$ {0, 2,..., n-1}

  S = Schlüsselraum, n = Größe des **statischen** Hash-Bereiches in Seiten (***Buckets***)
- Idealfall: h ist injektiv (keine Kollisionen)
  - Dies ist nur in Ausnahmefällen möglich (’dichte’ Schlüsselmenge).
  - Jeder Satz kann mit 1 Seitenzugriff gefunden werden.
- Statische Hash-Bereiche mit Kollisionsbehandlung
  - Vorhandene Schlüsselmenge K (K $\subseteq$ S) soll möglichst gleichmäßig auf die n Seiten verteilt werden.
  - Behandlung von Synonymen:
    - Aufnahme in die selben Seite, wenn möglich
    - ggf. spezielle Behandlung von sogenannten Überlaufsätzen
- Anforderung an Hash-Verfahren
  - möglichst hohe SPAN
  - möglichst wenig Zugriffe für exakte Suche, Einfügen und Löschen

Getrennte Verkettung der Überläufer

- S = {0,…, 99}, n = 5, h(K) = K *MOD* 5



Notation

- N = #Sätze
- n = n_p = #Seiten
- n_s = #Sekundärseiten
- b = Kapazität der Primärseiten
- c = Kapazität der Sekundärseiten (Annahme: c = b)
- Belegungsfaktor: $\beta = \frac{N}{n \cdot b}$
- Speicherplatzausnutzung:

$$\alpha = \frac{N}{n_p \cdot b + n_s \cdot c}$$

Überlaufbehandlung ohne Verkettung

- offene Adressierung: Überlaufsätze werden in der Primärdatei abgelegt.

Lineares Sondieren (linear probing)

Ein Satz k wird in die erste nicht-volle Seite mit Adresse (h(k) + i) MOD n eingefügt, i = 0,…,n-1.

0	35	•
	40	•
	55	•
	85	•
1	61	•
	76	•
	90	◦
	58	◦
2	27	•
	37	•
	92	•
3	03	•
	18	•
	23	•
	43	•
4	04	•
	24	•
	49	•
	98	◦

Zufälliges Sondieren

Ein Zufallsgenerator bestimmt in Abhängigkeit des Schlüssels K eine Reihenfolge $\{g_j\}$ mit $0 \leq g_j < n$, $j \leq 0$.

Beispiel:
Schlüssel 90: 2,…
Schlüssel 58: 3,2,4,…
Schlüssel 98: 2,1,…

0	35	•
	40	•
	55	•
	85	•
1	61	•
	76	•
	98	◦
2	27	•
	37	•
	92	•
	90	◦
3	03	•
	18	•
	23	•
	43	•
4	04	•
	24	•
	49	•
	58	◦

Analyse der Verfahren

- Für die exakte Suche muß im schlechtesten Fall auf alle Seiten zugegriffen werden!
- Was ist das erwartete Verhalten der Verfahren?
  - erfolgreiche Suche
  - erfolglose Suche
- Was ist der erwartete *schlechteste Fall* der Verfahren?
- Beispiel: Hash-Verfahren mit Verkettung

Vergleich der Verfahren

erwartete erfolgreiche Suche, b = 10

	Speicherplatzausnutzung			
	0.6	0.7	0.8	0.9
Verkettung (c=1)	1.024	1.064	1.152	1.372
lineares Sond.	1.015	1.042	1.110	1.345
zufälliges Sond.	1.014	1.035	1.079	1.177

erwartete erfolglose Suche, b = 10

	Speicherplatzausnutzung			
	0.6	0.7	0.8	0.9
Verkettung (c=1)	1.082	1.230	1.568	2.392
lineares Sond.	1.114	1.323	1.987	5.71
zufälliges Sond.	1.099	1.243	1.572	2.591

erwarteter schlechtester Fall für
erfolglose Suche, b = 10, N = 100,000

	Speicherplatzausnutzung			
	0.6	0.7	0.8	0.9
Verkettung (c=1)	10.9	12.9	15.1	18.0
lineares Sond.	7.4	13.1	29.1	111.0
zufälliges Sond.	5.2	7.2	10.9	20.8

Zusammenfassung

Für statische Dateien gilt:

- Statische Hash-Verfahren sind sehr effizient für exaktes Suchen, Einfügen und Löschen von Datensätzen.
- Erwartetes Verhalten ist besser als für B^+-Bäume, wobei gleichzeitig auch die Speicherplatzausnutzung höher ist.
- Worst-case Verhalten ist schlechter als für B^+-Bäume.
- Erwartungswert für die längste erfolglose Suche ist aber relativ niedrig, wenn die Überläufer verkettet werden, bzw. durch zufälliges Sondieren beseitigt werden.

Für dynamische wachsende (und schrumpfende) Dateien gilt:

- Statische Hash-Verfahren sind ungeeignet, da Ketten zu lang werden.
  - Erfordert eine globale Reorganisation der Datei

FRAGE:

- Können globale Reorganisationen bei Hash-Verfahren vermieden werden, ohne daß damit eine wesentliche Leistungsminderung in Kauf genommen werden muß?

3.3 Dynamische Hash-Verfahren

- Ziel:

  Wenn die Leistung des Hash-Verfahren durch Einfügen bzw. Löschen von Datensätzen zu schlecht ist (SPAN zu niedrig, Ketten zu lang), soll die Anzahl der Seiten n dynamisch angepaßt werden, d.h. ohne globale Reorganisation der Datensätze.

- Beispiel:
  - h(K) = K MOD 5 ist die bisherige Hash-Funktion.
  - wenn h(K) = K MOD 7 die neue Hash-Funktion ist, müssen alle (?) Adressen neu berechnet werden.

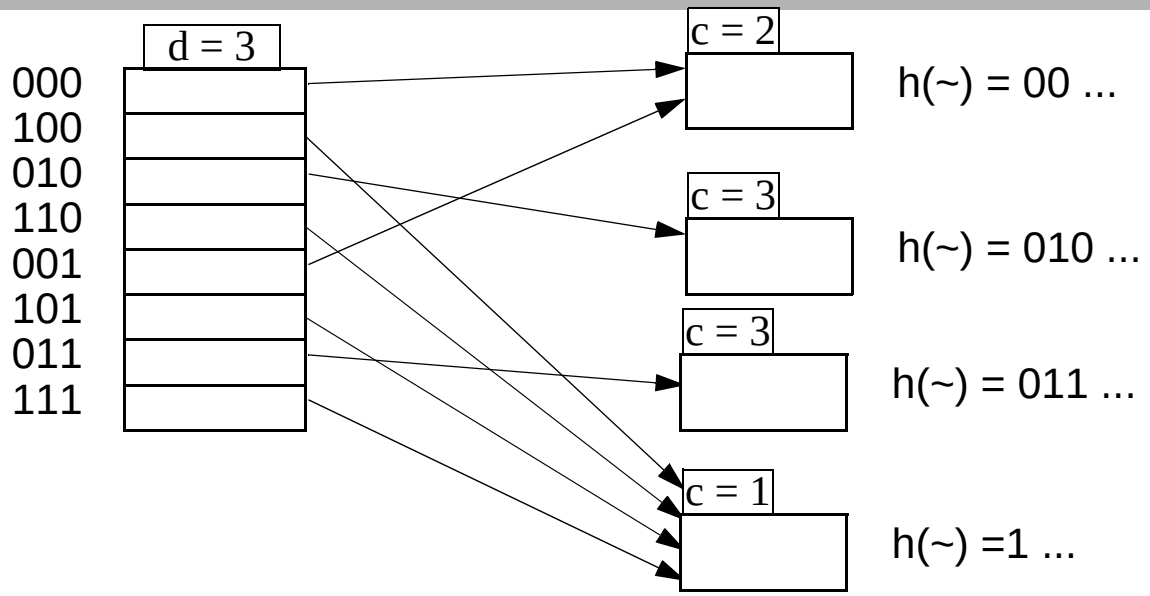
  ==> globale Reorganisation ist notwendig.

- Dynamische Hash-Verfahren unterscheiden sich im wesentlichen in folgenden Punkten:
  - Größe der Kapazität einer Primärseite:
    - Für b = 0, spricht man auch von Verfahren mit Hash-Tabelle (Directory).
  - Überlaufbehandlung: getrennte oder gemeinsame Verkettung
  - Wahl der Folge von Hash-Funktionen

3.3.1 Erweiterbares Hashing

- Verfahren mit b = 0: Primärdatei degeneriert zu einem **Directory**. Eine Primärseite entspricht dann einem **Eintrag** im Directory.
  - Eintrag verweist auf eine Seite, in dem alle zugehörigen Sätze gespeichert sind.
- Überlaufbehandlung
  - Alle Sätze sind de facto Überlaufsätze.
  - Überlaufketten besitzen höchstens die Länge 2 (inkl. der deg. Primärseite).
  - Eine Überlaufseite kann von “benachbarten” Einträgen gemeinsam benutzt werden (Nachbarverkettung).
- Hash-Funktion $h_d(K)$, $d \geq 0$
  - Zunächst wird für ein Schlüssel K ein Pseudoschlüssel (b_0, b_1, …, b_j, …) generiert, $b_j \in \{0,1\}$, $j \geq 0$.
  - Die ersten d Bits des Pseudoschlüssels werden zur Berechnung der Adresse verwendet.
  - Directory enthält 2^d Einträge. d ist die **globale Tiefe** des Directory
  - Eintrag verweist auf eine Seite, in dem alle zugehörigen Sätze gespeichert sind.

Beispiel



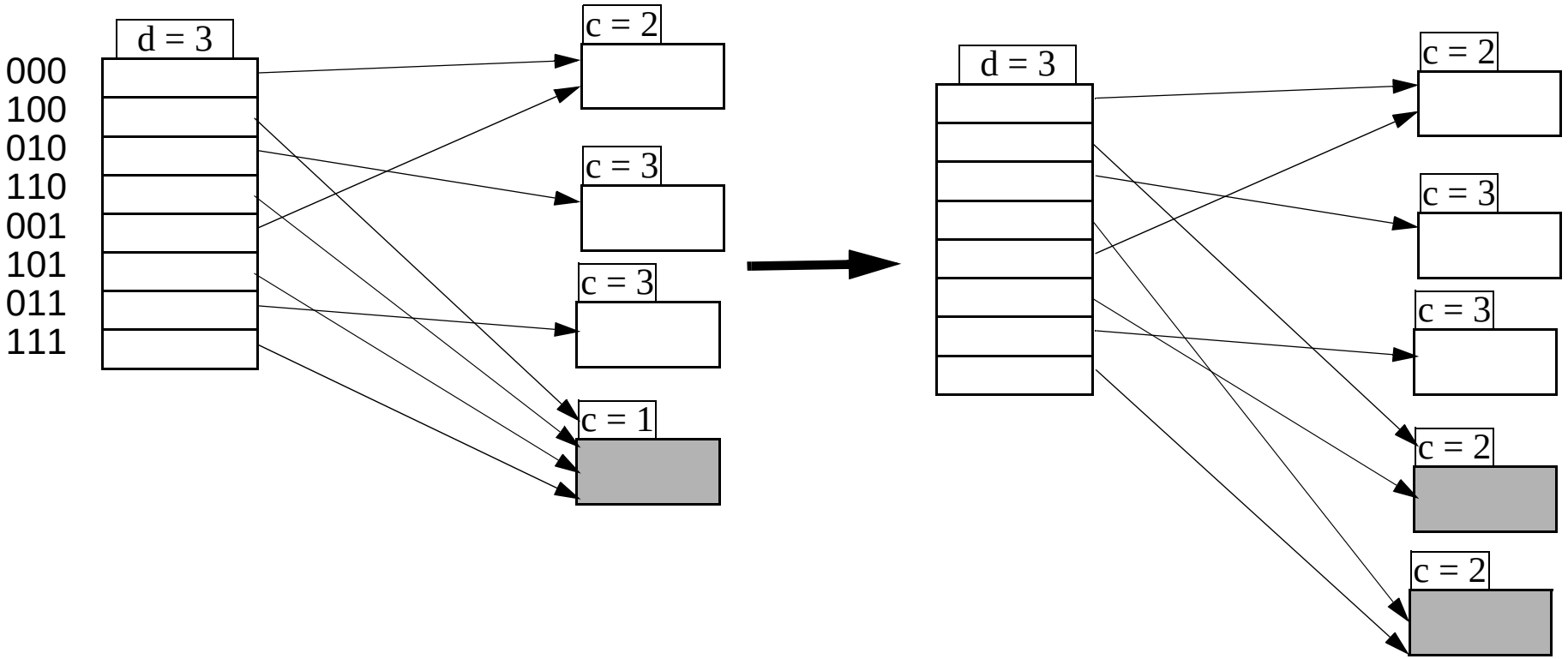
- In einer Seite können nur Datensätze gespeichert werden, die in den ersten c Bits, $c \leq d$, übereinstimmen.
  - c wird auch als **lokale Tiefe** der Seite bezeichnet.
  - Es gibt genau 2^{d-c} (benachbarte) Einträge im Directory, die auf eine Seite der Tiefe c verweisen.
  - Es gibt mindestens eine Seite, deren lokale Tiefe gleich der globalen Tiefe ist.

Einfügen eines Datensatzes

- Gegeben: Datensatz mit Schlüssel K
  1. Berechne die ersten d Bits des Pseudoschlüssels: (b_0, b_1,…,b_{d-1}).
  2. Lies die Seitenreferenz S aus dem Eintrag $\sum_{j=0}^{d-1} b_j \cdot 2^j$ des Directory.
  3. Lies die Seite S und prüfe, ob es bereits ein Schlüssel K in der Seite gibt. Wenn ja, verlasse die Prozedur mit einer entsprechenden Fehlermeldung.
  4. Füge den Datensatz in die Seite S ein und prüfe, ob es ein Überlauf gibt.
  5. Wenn ja, dann verfahre folgendermaßen (c = lokale Tiefe der Seite):
     - Falls c = d, erhöhe die globale Tiefe des Directory und verdopple die Anzahl der Elemente.
     - Kopiere die Datensätze, deren (c+1)-tes Bit des Pseudoschlüssels gesetzt ist, aus der Seite in eine neu allokierte Seite.
     - Ändere die entsprechenden Seitenreferenzen im Directory.
     - Falls immer noch ein Überlauf in einer der Seiten vorliegt, so wiederhole diesen Schritt entsprechend.

Beseitigung eines Überlaufs

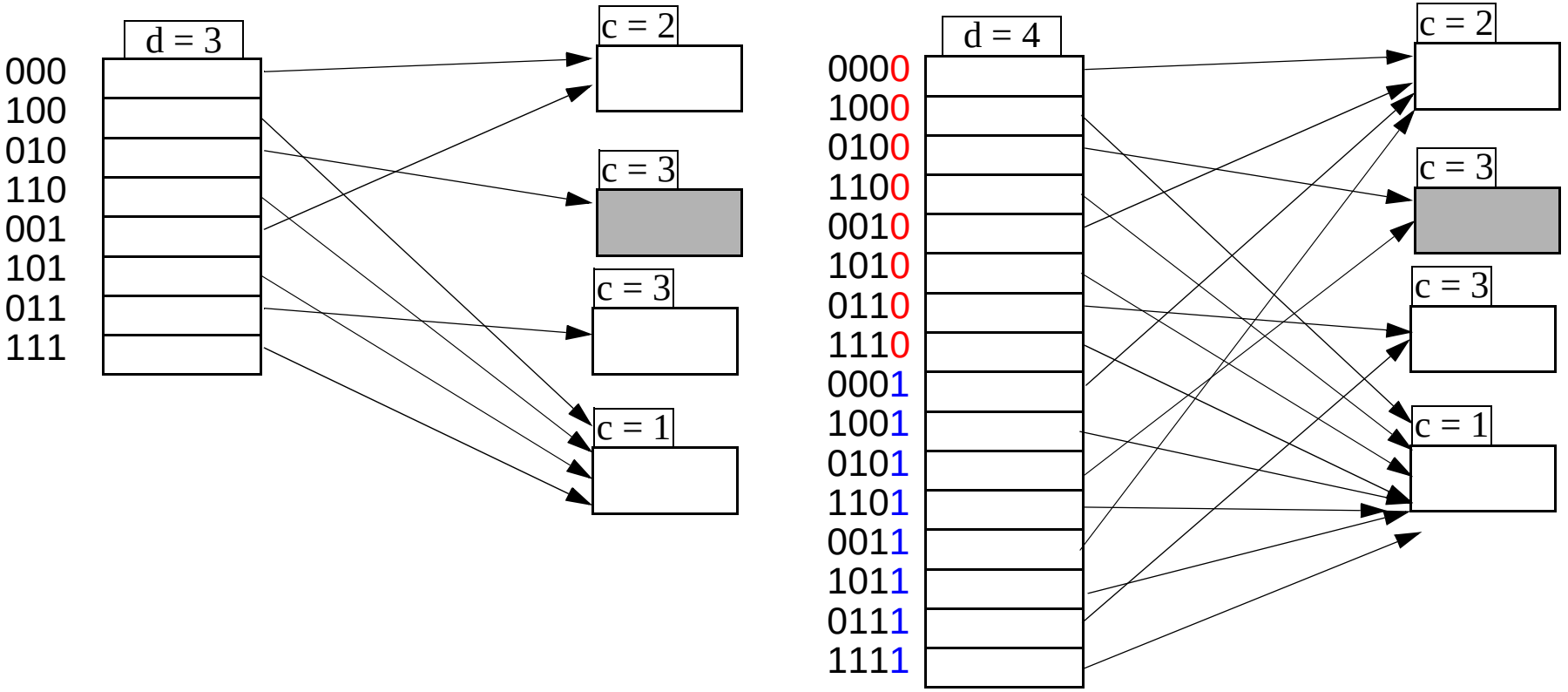
in einer Seite mit lokaler Tiefe < globaler Tiefe:



Beseitigung eines Überlaufs

in einer Seite mit lokaler Tiefe = globaler Tiefe:

- Verdopplung des Directory
  - Dann gilt: lokale Tiefe < globalen Tiefe



Eigenschaften des Verfahrens

- Directory muß i.a. aufgrund seiner Größe im Sekundärspeicher abgelegt werden.
- Jeder Datensatz kann garantiert mit 2 Zugriffen gefunden werden.
- *Erwartete Speicherplatzausnutzung* = ln 2 ≈ 0.693 (unter der Annahme der Gleichverteilung der Pseudoschlüssel).
- I.A. ist erweiterbares Hashing nicht zur Unterstützung von Bereichsanfragen geeignet.

Nachteile:

- *Directory* wächst superlinear *mit der Anzahl der Datensätze:* $O(N^{1+1/b})$ bereits bei Gleichverteilung der Schlüssel.
- Verdopplung des Directory kann bei großen Dateien sehr teuer sein.

3.3.2 Lineares Hashing (LH)

- Verfahren mit b > 0, d.h. es gibt eine Primärdatei und eine Sekundärdatei.
  - LH benutzt somit kein Directory!
  - Primärdatei besteht am Anfang aus q Datenseiten.
- Überlaufbehandlung
  - getrennte Verkettung der Sekundärseiten mit den Primärseiten.
  - andere Überlaufstrategien sind aber auch verwendbar.
- Der Clou bei linearem Hashing liegt in der Auswahl der Hash-Funktionen
  - Folge von Hash-Funktionen $\{h_j(K)\}_{j\geq 0}$ mit folgenden Bedingungen:

    Bereichsbedingung:

    $$h_j: \text{domain}(K) \rightarrow \{0, 1, \ldots, 2^j q-1\}, j \geq 0$$

    Splitbedingung:

    $$h_{j+1}(K) = h_j(K) \quad \text{oder}$$

    $$h_{j+1}(K) = h_j(K) + 2^j * q, j \geq 0$$

    Der **Level** L (= j) gibt an, wie oft sich die Datei bereits verdoppelt hat.

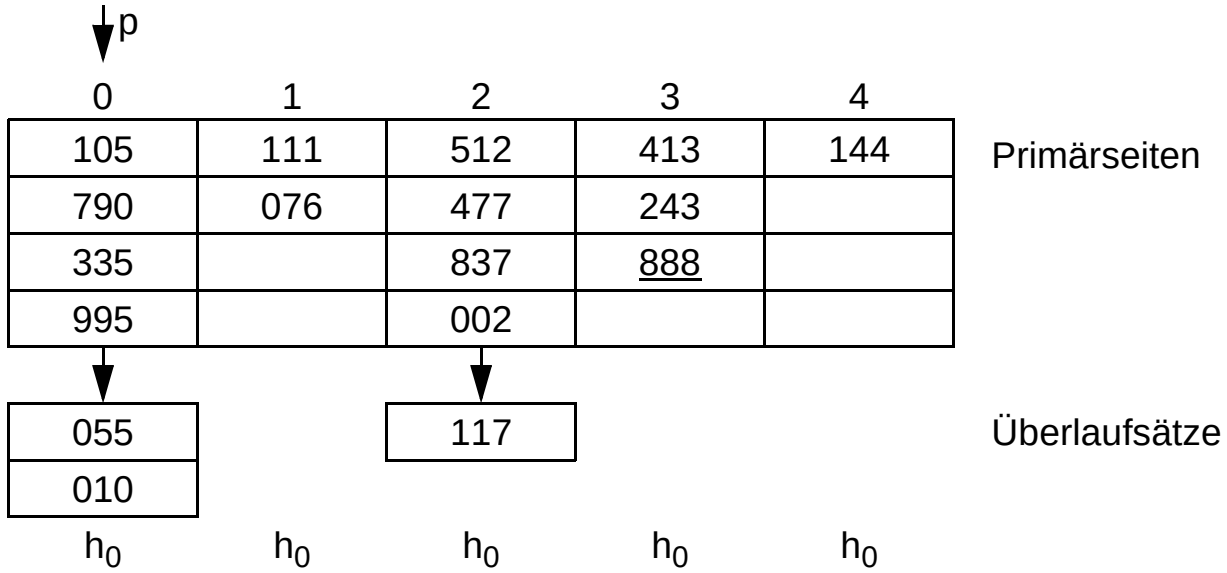
- Beispiel:

  $h_j(K) = K \bmod (2^j q)$ erfüllt beide Bedingungen.

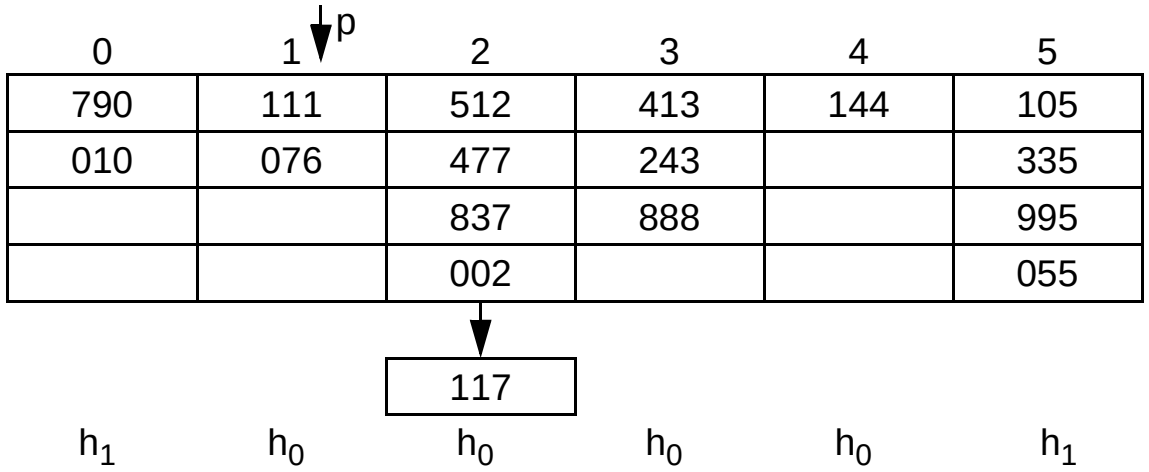
- Lineares Hashing löst ein Aufspalten einer Seite aus, wenn eine Bedingung erfüllt ist (**Kontrollfunktion**)
  - z. B.: falls der Belegungsfaktor > 80%, dann führe ein Aufteilen einer Seite aus.
  - Für die Auswahl der Seite gibt es einen Zeiger p, der auf die Seite verweist, die als nächstes aufgespaltet werden soll.
- folgende Information ist für lineares Hashing notwendig:
  - L: Level (Anzahl der bereits ausgeführten Verdopplungen)
  - p: zeigt auf nächste zu spaltende Primärseite, $0 \leq p < q\, 2^L$
  - β: Belegungsfaktor
  - N: Anzahl der gespeicherten Sätze
  - b: Kapazität einer Primärseite
  - c: Kapazität einer Sekundärseite (i.a. gilt c = b).

Beispiel: Prinzip des LH

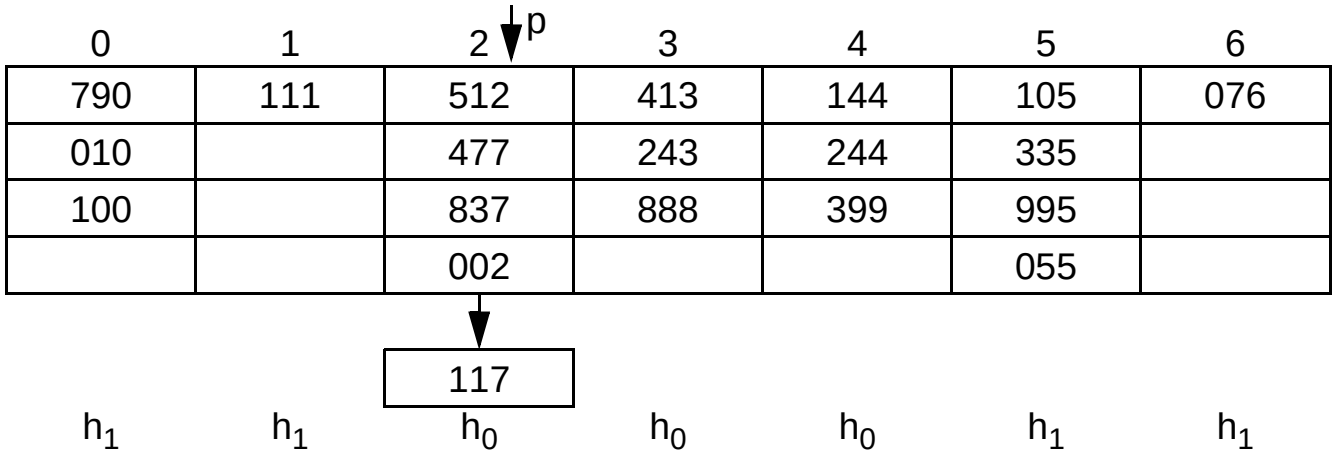
- Vorgaben:
  - q = 5, L = 0, b = 4, h_0 (K) = K mod 5 und h_1 (K) = K mod 10
  - Splitting, sobald $\beta > \beta_s = 0.8$



- Einfügen von Satz 888 erhöht Belegungsfaktor auf $\beta = 17/20 = 0.85$ und löst Aufspalten der Seite p aus:
  - Alle Sätze mit $h_1(K) = p$ verbleiben in Seite p und die anderen Sätze werden in der neu allokierten Seite $p + 2^L q$ abgelegt.



- Einfügen von 244, 399 und 100. Die letzte Einfügung erhöht den Belegungsfaktor auf $\beta = 20/24 = 0.83$ und löst Split der Seite p aus:



Erweitern der Datei

- Erweitern wird durch eine Kontrollfunktion ausgelöst, z. B. $\beta > 80\%$.
- Position p gibt an, welche Seite aufgespaltet wird.
- Alle Sätze mit $h_1(K) = p$ verbleiben in Seite p und die anderen Sätze werden in der neu allokierten Seite $p + q\ 2^L$ abgelegt.
- p wird um 1 erhöht: $p = (p+1)\ \text{MOD}\ (q\ 2^L)$.
  Falls p = 0, so hat sich die Anzahl der Seiten in der Datei verdoppelt und L wird nun um 1 erhöht.

Split-Strategien:

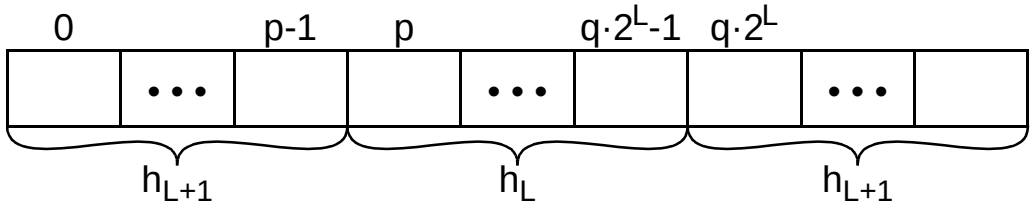
- Unkontrolliertes Splitting
  - Splitting, sobald ein Satz in den Überlaufbereich kommt
  - $\beta \sim 0.6$, schnelleres Aufsuchen
- Kontrolliertes Splitting
  - Splitting, wenn ein Satz in den Überlaufbereich kommt und $\beta > \beta_s$
  - $\beta \sim \beta_s$, längere Überlaufketten möglich

Suchen und Adressenberechnung

- Falls $h_0(K) \geq p$, dann ist $h_0(K)$ die gewünschte Adresse.
- Andernfalls ($h_0(K) < p$), ist die Seite bereits aufgeteilt worden und $h_1(K)$ liefert die gewünschte Adresse.
- allgemein:

  $h := h_L(K)$;

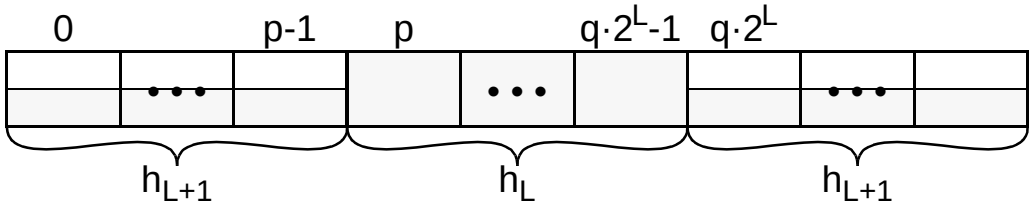
  IF ($h < p$) THEN $h := h_{L+1}(K)$;



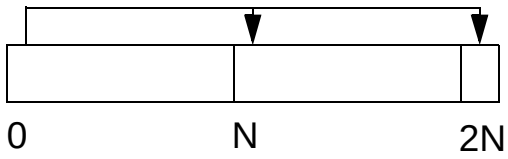
- exakte Suche:
  - Berechne die Hash-Adresse;
  - Lies die Primärseite und durchsuche die Seite, ob der Schlüssel K vorhanden ist.
  - Falls ja, STOP
  - Andernfalls, durchsuche die Kette der Sekundärseiten nach dem Satz.

3.3.3 LH mit partiellen Erweiterungen

- bisherige Vorgehensweise
  - jeder Erweiterungsschritt teilt die Datensätze von einem auf zwei Seiten auf.
  - es ergibt sich deshalb eine ungleichmäßige Belegung von bereits aufgeteilten und noch nicht aufgeteilten Seiten.



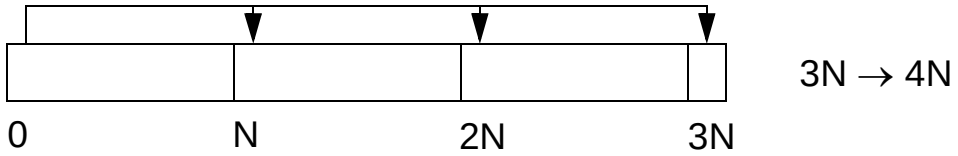
- Ziel: gleichmäßigere Belegung
  - Verdoppeln der Seitenanzahl durch eine Folge partieller Expansions-Schritte
- Idee (q=1):
  - Ausgangspunkt: Datei mit 2N Seiten, die logisch in N Paare unterteilt ist: (j, j+N) für j = 0,1,…,N-1
  - erste partielle Expansion:



$2N \rightarrow 3N$

Sätze aus Seiten (j,N+j) werden auf die Seiten (j, N+j, 2N+j) verteilt, j = 0,…,N-1

– zweite partielle Expansion



Verteilen der Sätze aus den Seiten (j, N + j, 2N + j) auf die vier Seiten (j, N + j, 2N + j, 3N + j).

– nach Abschluß der zweiten partiellen Expansion hat sich die Dateigröße verdoppelt und es kann wieder mit der ersten partiellen Expansion gestartet werden.

❑ Eigenschaften:

– die Belegung in bereits aufgeteilten Seiten ist 2/3 niedriger als die in noch nicht aufgeteilten Seiten der ersten partiellen Expansion.

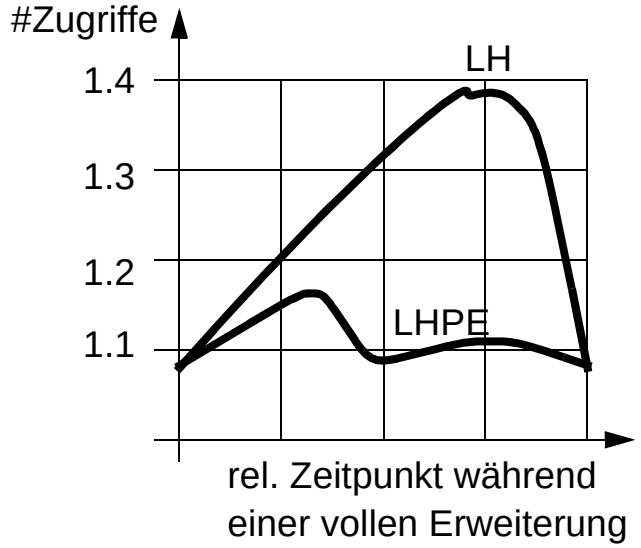
– nach der ersten partiellen Expansion ist die erwartete Belegung in allen Seiten gleich.

– die Belegung in bereits aufgeteilten Seiten ist 3/4 niedriger als die in noch nicht aufgeteilten Seiten der ersten partiellen Expansion.

❑ allgemein: vollständige Erweiterung kann in n_0 partiellen Erweiterungen realisiert werden.

Leistungsvergleich

- Durchschnittliche # Zugriffe für $n_0 = 1$, $n_0 = 2$ und $n_0 = 3$ für eine Datei mit b = 20, c = 5, SPAN = 0.85.



	$n_0 = 1$	$n_0 = 2$	$n_0 = 3$
Erfolgreiche Suche	1.27	1.12	1.09
Erfolglose Suche	2.12	1.58	1.48
Einfügen	3.57	3.21	3.31
Entfernen	4.04	3.53	3.56

- bei Benutzung von partiellen Erweiterungen ist der Aufwand für das Einfügen niedriger als beim LH ohne partielle Erweiterungen.

3.3.4 Spiralen-Hashing

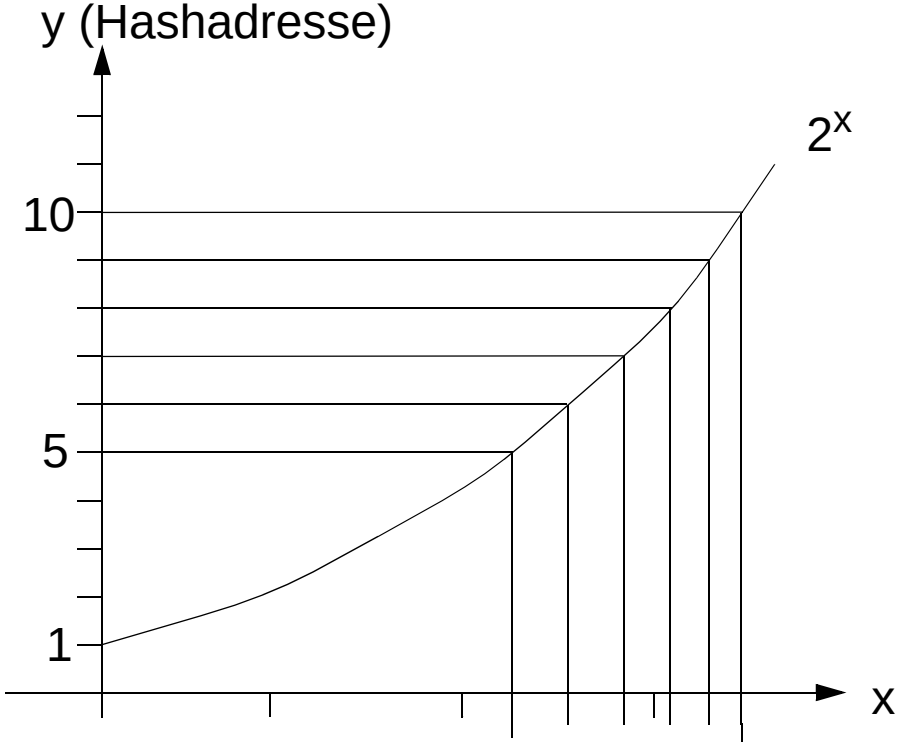
- Problem bei LH und LHPE
  - durchschnittliche Suchkosten werden durch den rel. Zeitpunkt in einer vollen Erweiterung bestimmt.
- Spiralen-Hashing vermeidet dieses Problem durch
  - ungleiche Verteilung der Daten über die Seiten
  - Hashfunktion erfüllt folgende Bedingung:
    a) Aufspalten der Seite mit dem höchsten (erwarteten) Belegungsfaktor
    b) die aus dem Aufspalten entstandenen Seiten besitzen den niedrigsten (erwarteten) Belegungsfaktor
- Veranschaulichung der Idee:
  - Seiten werden als Bereiche in einer Spirale veranschaulicht
  - Datei entspricht einer vollen Drehung in der Spirale
  - Vergrößern der Datei entspricht einem Herausdrehen der Datei aus der Spirale

Adressberechnung

- Parameter d, d > 1, ist gegeben.
  - d wird auch der Wachstumsfaktor der Datei genannt.
- Schlüssel K wird zunächst auf ein Intervall $[S, S+1)$ abgebildet, wobei auch S i.a. nicht ganzzahlig ist.
- Algorithmus
  1. Berechne Hashfunktion $h(K) \in [0, 1)$.
  2. Berechne $x = \lceil S - h(K) \rceil + h(K)$.
     Beachte, daß x und h(K) die gleichen Nachkommastellen besitzen.
  3. Adresse der Seite ergibt sich aus $y = \lfloor d^x \rfloor$
- Die Funktion $\lfloor d^x \rfloor$ wird auch als Wachstumsfunktion bezeichnet.

Eigenschaften der Adressberechnung

- Der aktuelle Adressraum geht von $\lfloor d^S \rfloor$ bis $\lceil d^{S+1} \rceil - 1$.
  - dies entspricht ungefähr $d^S(d-1)$ Adressen.
- Adressraum kann erweitert werden, indem S durch ein größeres S' ersetzt wird.
  - durch S' = S + 1 wird der Adressraum ungefähr d-mal größer.



Beispiel

- Initialgröße der Datei: 5 Seiten
- Wachstumsfaktor: d = 2
- S ergibt sich aus der Gleichung $2^{S+1} - 2^{S} = 5 \Longrightarrow S = \log_2 5 = 2.3219$
- erste Adresse: $y_S = \lfloor 2^{2.3219} \rfloor = 5$ letzte Adresse: $y_{S+1} = \lfloor 2^{3.3219} \rfloor - 1 = 9$
- Beispiel:

Adresse	untere Grenze für x	obere Grenze für x	relative Belegung
5	$\log_2 5$	$\log_2 6$	0.263
6	$\log_2 6$	$\log_2 7$	0.222
7	$\log_2 7$	$\log_2 8$	0.193
8	$\log_2 8$	$\log_2 9$	0.170
9	$\log_2 9$	$\log_2 10$	0.152
10	$\log_2 10$	$\log_2 11$	0.137
11	$\log_2 11$	$\log_2 12$	0.126

Zusammenfassung

- LH mit partiellen Erweiterungen ist ein effizientes dynamisches Hash-Verfahren
  - mit nahezu einem Plattenzugriff wird ein Datensatz gefunden.
  - hohe Speicherplatzausnutzung.
  - niedrige Kosten für das Einfügen von Datensätzen.
  - Leistung ist unabhängig von der Anzahl der Datensätze!
  - es wird kein Index benötigt, sondern nur einige Parameter.
- interessante Varianten von LH
  - Speicherung der Überläufer in Primärdatei
  - rekursives LH
  - Spiralenspeicherung
- erweiterbares Hashing sollte den Vorzug gegenüber LH erhalten, wenn die 2-Disk Suchgarantie wichtig ist.
- im Vergleich zu B^+-Bäumen sind dynamische Hash-Verfahren effizienter, wenn Bereichsanfragen nicht oder kaum auftreten.
- B^+-Bäume sollten immer dann benutzt werden, wenn eine sequentielle Verarbeitung der Daten wichtig ist.