

5. Konventionelle Zugriffsstrukturen

- ☐ Anforderungen
- ☐ Sequentieller Zugriff (ohne Unterstützung einer Zugriffsstruktur)
 - Zugriffsformen auf Satzmengen
 - Clusterbildung (Listen)
 - Ketten
- ☐ eindimensionale Zugriffsstrukturen:
 - B⁺- Bäume und Varianten
 - erweitertes Splitting
 - Schlüsselkomprimierung
 - Hashverfahren
 - statische Verfahren
 - Externes Hashing mit Separatoren
 - Erweiterbares Hashing
 - Lineares Hashing
- ☐ Zugriff über Sekundärschlüssel
- ☐ Hierarchische Zugriffsmethode
- ☐ Verallgemeinerte Zugriffsstruktur

116.

Änderungsoperationen

- ☐ sind relativ teuer, da die Änderung eines Datensatzes i. a. das Hinausschreiben einer Seite auf den Externspeicher verursacht.
- ☐ Durch viele Änderungsoperationen können die Suchkosten ansteigen

Strategien bei Änderungsoperationen

- ☐ sofortiges Einfügen, Löschen und Ändern eines Datensatzes:
 - niedrige Suchkosten ⇒ Datenreorganisationen.
 - a) **lokale** Reorganisationen im on-line Betrieb (**dynamische Zugriffsstrukturen**)
 - b) "Verwahrlosen" der Zugriffsstruktur und periodisches "Aufräumen"
- ☐ verzögertes Einfügen, Löschen und Ändern
 - Sammeln von Änderungsoperationen
 - Periodische Anwendung gesammelter Änderungsoperationen (**Bulk Update**)
- ☐ Aufbau einer komplett neuen Zugriffsstruktur
 - direkt vom Benutzer gefordert: create index on ...
 - Leistung der Zugriffsstruktur soll verbessert werden (**globale** Reorganisation)

118.

5.1 Anforderungen

allgemeine Ziele beim Entwurf von Zugriffsstrukturen:

- ☐ hohe Speicherplatzausnutzung
- ☐ kurze Antwortzeiten für eine Operation
 - Kostenmaß = Anzahl der E/A-Zugriffe
- ☐ hoher Durchsatz
 - Durchsatz = Anzahl der abgearbeiteten Operationen pro Zeiteinheit
- ☐ Implementierbarkeit und Integration in bestehende Systeme
 - Verhältnis Kosten/Nutzen sollte berücksichtigt werden.
 - Mehrbenutzerbetrieb

Operationen

- ☐ Suchanfragen
 - Daten werden nur gelesen, aber nicht verändert.
- ☐ Einfügen, Löschen und Ändern von Datensätzen

117.

Suchoperationen

Unterstützung folgender Zugriffsmuster:

- ☐ Single-Scan Anfragen
 - können stets durch lineares Durchsuchen einer Relation beantwortet werden.
 - Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)
 - Sequentieller Zugriff in Sortierreihenfolge eines Attributs
 - Direkter Zugriff über den Primärschlüssel
 - Direkter Zugriff über einen Sekundärschlüssel
 - Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle,...)
 - Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge des gleichen oder eines anderen Satztyps
- ☐ Multi-Scan Anfragen
 - müssen i. a. mehrmals den gleichen Datensatz aufsuchen.
 - Verbund-Operationen

119.

Beispiele

- Relation R mit k (ausgewählten) Attributen A_1, \dots, A_k .

$$\kappa$$

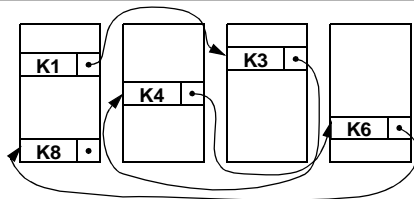
- Wertebereich $D = \prod_{i=1}^{\kappa} \text{dom}(A_i)$

Anfragen:

- exact match query:
Gegeben $(x_1, \dots, x_k) \in D$. Suche den Datensatz $r \in R$ mit $r.A_1 = x_1, \dots, r.A_k = x_k$.
- partial match query:
Gegeben $m \leq k$ und Index (i_1, \dots, i_m) mit $1 \leq i_1 \leq \dots \leq i_m \leq k$. Gegeben $(x_{i_1}, \dots, x_{i_m})$ mit $x_{i_j} \in \text{dom}(A_{i_j})$. Suche alle Datensätze $r \in R$ mit $r.A_{i_1} = x_{i_1}, \dots, r.A_{i_m} = x_{i_m}$.
- window query (Bereichsanfrage):
Gegeben $(u_1, \dots, u_k), (o_1, \dots, o_k) \in D$ mit $u_i \leq o_i$. Suche alle Datensätze $r \in R$ mit $u_1 \leq r.A_1 \leq x_1, \dots, u_k \leq r.A_k \leq x_k$.

120.

Liste über Datensätze



Leistung

- sequentieller Zugriff auf alle Sätze: N Seitenzugriffe
- fortlaufende Suche bei Verkettung in Sortierreihenfolge: $N/2$ Seitenzugriffe
- Einfügen relativ billig bei bekannter Einfügeposition: ≤ 2 Seitenzugriffe
- Die Leistung der Operationen kann dadurch verbessert werden, daß gemeinsam auf einer Seite liegende Datensätze erkannt werden (z. B. K1 und K8 im oberen Schaubild).

122.

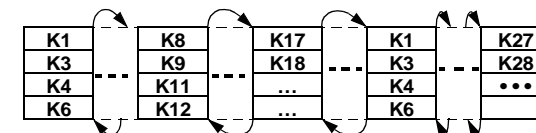
5.2 Sequentieller Zugriff

- Default-Zugriffsform in allen DBMS
- ausreichend / effizient bei:
 - kleinen Satztypen (z.B. ≤ 10 Seiten)
 - Anfragen mit großen Treffermengen (z.B. $> 5\%$)
- Optimierungsmöglichkeiten:
 - Clusterbildung
 - Allokation auf schnellen Speichermedien
 - wertabhängige Partitionierung eines Satztyps auf verschiedene Dateien (Bsp.: "partitioned tables" in DB2 und Oracle)
 - parallele Verarbeitung (durch "Decustering" und Partitionierung)
- DBMS kann Prefetching und Mehrseiten-Anforderungen zur Scan-Optimierung nutzen.
- Implementierung durch externe Listenstrukturen

121.

Liste über Seiten

- physisch benachbarte Speicherung der Datensätze in Seiten (lokale Clusterung)



- Blockungsfaktor:** Anzahl der Datensätze pro Seite
- Bei physisch benachbarter Speicherung der Seiten spricht man auch von **globaler** Clusterung
 - Vorteil: sequentielle statt zufällige Seitenzugriffe

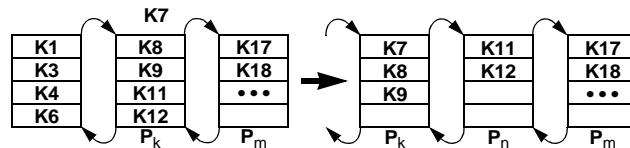
Leistung

- Bei N Sätzen und mittlerem Blockungsfaktor C : N/C Seitenzugriffe.

123.

Sortierte Anordnung der Datensätze in Seiten (lokale Clusterung)

- ❑ Pro Relation kann Clusterbildung nur bezüglich **eines** (Sortier-) Kriteriums erfolgen, falls keine Redundanz eingeführt werden soll.
- ❑ (erfolgreiche) exakte Suche kostet dann im Mittel: $N/2C$ Seitenzugriffe.
- ❑ Splitting-Technik beim Einfügen in eine volle Seite



Globale Clusterung

- ❑ Datenseiten liegen hintereinander (physisch oder logisch)
- ❑ Möglichkeit der binären Suche
- ❑ teure Änderungen durch (Domino-Effekt): $N/2C$ Seitenzugriffe

124.

5.3 B⁺-Bäume

B-Bäume von R. Bayer, 1972 und die Erweiterung auf B⁺-Bäume von D. Knuth, 1973.

Im Gegensatz zu binären Bäumen:

- ❑ viele Einträge/Sätze in einem Knoten
- ❑ alle Daten liegen in den Blattknoten
- ❑ 1:1-Beziehung zwischen Knoten und Seiten

B⁺-Bäume ist eine dynamische Variante von ISAM

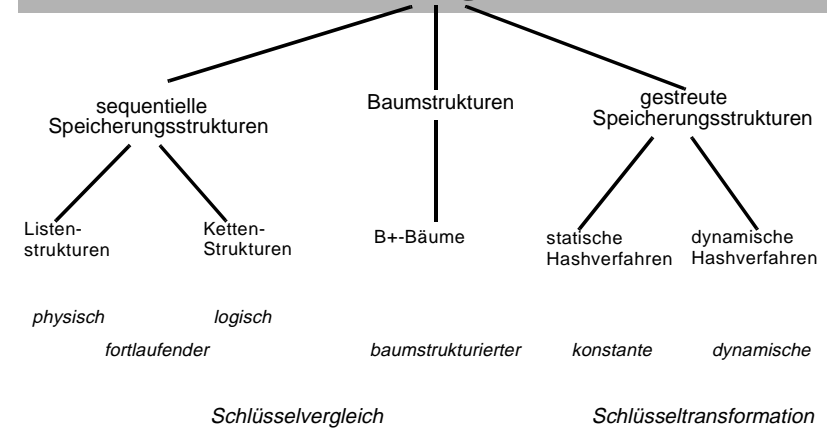
- ❑ keine periodisch Anwendung globaler Reorganisation

prinzipielle Idee

- ❑ Erzeuge für die Datensätze einen Index, der in einer Seite abgelegt wird.
- ❑ Falls die Seite überläuft, wird zusätzlich ein Index für den Index erzeugt.
- ❑ Dieses Vorgehen wird nun rekursiv angewendet.

126.

Eindimensionale Zugriffsstrukturen



125.

Funktionen

- direkter Schlüsselzugriff: exact match query
- sortiert sequentieller Zugriff: range query

Effizienz (Speicherplatz u. Antwortzeiten)

- ❑ asymptotisch unabhängig von der Einfügereihenfolge.

Optimierungsmöglichkeiten

- ❑ Verbesserung des Verzweigungsgrads (fan-out)
 - Schlüsselkomprimierung
 - Nutzung von "Wegweisern"
 - Präfix-B-Bäume
- ❑ Verbesserung des Belegungsgrades durch verallgemeinerte Splittingverfahren
 - B*-Baum
 - B⁺-Baum mit partiellen Erweiterungen
- ❑ Parallele B⁺-Bäume

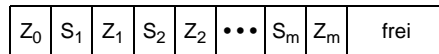
127.

Definition:

Ein B^+ -Baum vom Typ (c, c^*) ist ein Baum mit folgenden Eigenschaften

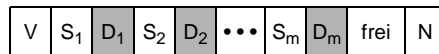
1. Jeder Weg von der Wurzel zum Blatt hat die gleiche Länge.
2. Jeder Zwischenknoten hat mindestens $c+1$ Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne. Jedes Blatt hat mindestens c^* Einträge.
3. Jeder Zwischenknoten hat höchstens $2c+1$ Söhne. Jedes Blatt hat höchstens $2c^*$ Einträge.

□ Zwischenknoten:



Z_i = Zeiger Sohnseite, S_i = Schlüssel

□ Blattknoten:



D_i = Verweis auf Satz, N = Nachfolger-Zeiger, V = Vorgänger-Zeiger

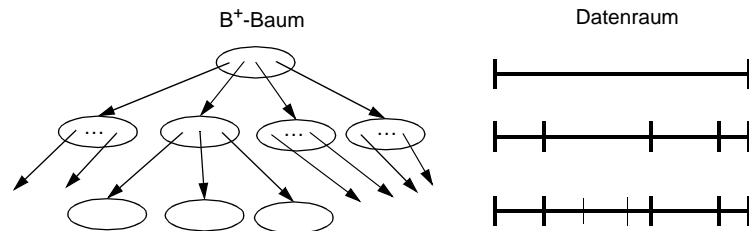
128.

Eigenschaften des B^+ -Baums

□ lokale Ordnungserhaltung:

Für jeden Zwischenknoten K mit j Schlüsseln k_1, \dots, k_j und $(j+1)$ Söhnen p_0, \dots, p_j gilt:

Für jedes i , $1 \leq i \leq j$, sind alle Schlüssel in dem zu p_{i-1} gehörenden Teilbaum nicht größer als k_i und k_i ist größer als alle Schlüssel, die im Teilbaum von p_i liegen.

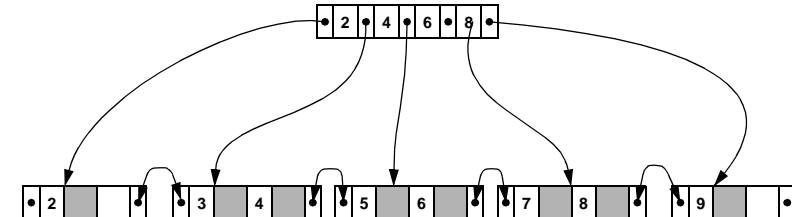


130.

Beispiel

□ $c=2, c^*=1$

□ Beachte: Parameter c und c^* sind nur aus Gründen der Übersicht so klein gewählt!



□ Wieviel Einträge passen in einen Zwischenknoten der Größe 4 KB?

- pro Zeiger: 4 Byte
- pro Schlüssel: 4 Byte

dies ergibt ca. 500 Einträge in einem Zwischenknoten.

129.

Wie hoch kann ein B^+ -Baum werden?

□ Welche Höhe besitzt ein B^+ -Baum zur Abspeicherung von N Datensätzen im schlechtesten Fall?

oder anders gefragt:

□ Wieviel Datensätze müssen mindestens (höchstens) in einem B^+ -Baum der Höhe h sein?

□ vereinfachende Annahme: $c+1 = c^*$

Wurzel hat mindestens 2 Einträge

Zwischenknoten in der Ebene 1 hat mindestens $c+1$ Einträge

Zwischenknoten in der Ebene 2 hat mindestens $c+1$ Einträge

... $c+1$ Einträge

Blattknoten in der Ebene h hat mindestens $c+1$ Sätze

Es folgt, daß in einem B^+ -Baum der Höhe h sich mindestens $2^{*(c+1)^h}$ Datensätze befinden. Es gilt also $N \geq 2^{*(c+1)^h}$ und somit

$$h \leq \log_{c+1} \frac{N}{2}$$

131.

Wieviel Speicherplatz benötigt der B⁺-Baum?

- ❑ Speicherplatzausnutzung (SPAN):

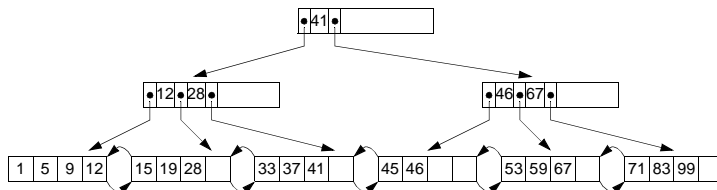
$$\frac{\text{minimal erforderlicher Speicherplatz}}{\text{tatsächlich reservierter Speicherplatz}}$$

- ❑ jeder Knoten (mit Ausnahme der Wurzel) ist mit mindestens der Hälfte der möglichen Schlüssel gefüllt.
 - ein B⁺-Baum braucht (im schlechtesten Fall) doppelt soviel Speicher wie ein optimal gefüllter Baum. Damit ergibt sich eine Speicherplatzausnutzung von mindestens 50 %.
- ❑ Unter der Annahme, daß die N Datensätze gleichverteilt sind und daß der neue Datensatz auch dieser Gleichverteilung folgt, gilt:
 - Die durchschnittliche Speicherplatzausnutzung von B⁺-Bäumen liegt bei etwa 69%.
- ❑ Die SPAN kann aber durch gewisse Techniken noch erheblich verbessert werden (siehe unten).

132.

Beispiel

- ❑ Suche den Datensatz mit Schlüssel 42.
- ❑ Suche den Datensatz mit Schlüssel 41.



134.

5.3.1 Suchoperationen im B⁺-Baum

exakte Suche:

- ❑ gegeben ein Schlüssel x . Finde den Datensatz r mit $r.key = x$ in dem B⁺-Baum mit Wurzel $root$.

Algorithmus EMQ(pact: Knoten, x : Key)

```

ReadPage(pact);
IF (pact ist ein Zwischenknoten) THEN
  index := m; (* m = Anzahl der Schlüssel im Zwischenknoten *)
  Bestimme im Knoten pact den kleinsten Schlüssel  $k_i$ , so daß  $x \leq k_i$ .
  IF (es gibt solch ein  $k_i$ ) THEN index := i-1; END;
  EMQ(pindex, x)
ELSE
  Bestimme im Knoten pact den Datensatz mit Schlüssel  $x$ .
  IF (es gibt solch ein Datensatz) Print("Datensatz wurde gefunden"); END;
END;
END EMQ;
```

133.

Bereichsanfrage im B⁺-Baum

- ❑ gegeben ein Schlüsselpaar low und up , $low \leq up$. Finde alle Datensätze r in dem B⁺-Baum mit Wurzel $root$ (in sortierter Reihenfolge) mit der Eigenschaft $low \leq r.key \leq up$.

Algorithmus RQ(pact: Knoten; low, up : Key)

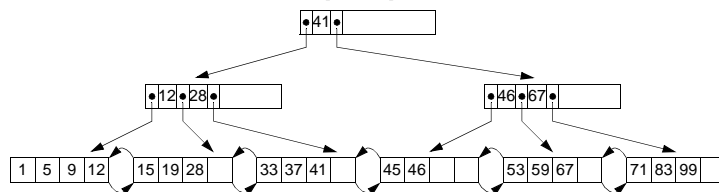
```

Bestimme analog zur exakten Suche das Blatt first, in dem ein Datensatz mit Schlüssel  $low$  liegen könnte;
pact := first;
LOOP
  ReadPage(pact);
  Bestimme im Knoten pact alle Datensätze mit Schlüssel  $x$  im Bereich  $[low, up]$ .
  IF (es gibt ein Datensatz  $r$  in pact mit  $r.key \geq up$ ) OR
    (pact enthält den größten Schlüssel der Datei) THEN
    EXIT
  END;
  pact := RightLeaf(pact); (* bestimmt rechten Nachbarknoten *)
END;
END RQ;
```

135.

Beispiel

- Suche alle Datensätze im Bereich [40, 52].



136.

Einfügen im B⁺-Baum

- gegeben ein Datensatz r und die Wurzel des B⁺-Baums. Füge den Datensatz in den B⁺-Baum ein.

Algorithmus Insert(pact: Knoten; r : Record);

Suche nach einem Datensatz mit Schlüssel $r.key$; (* siehe EMQ(pact, $r.key$ *)

IF (Datensatz wurde gefunden) THEN Print("ERROR"); RETURN END;

Setze pact auf das zuletzt gelesene Blatt;

Füge r in pact ein;

IF (pact ist übergelaufen) THEN

teile die Datensätze in pact in zwei gleich große Gruppen Glinks und Grechts;

so daß alle Datensätze in Glinks kleiner sind als die Datensätze in Grechts;

speichere die Datensätze in Grechts in einem neuen Blatt $pneu$ und die in Glinks in pact;

$kmax$ = größter Schlüssel in Glinks;

Füge das Paar ($kmax$, $pneu$) in den Vaterknoten ein;

IF (Vaterknoten ist übergelaufen) THEN ...

END Insert;

138.

5.3.2 Einfügen und Löschen in B⁺-Bäumen

Meistens ist das Einfügen und Löschen sehr einfach:

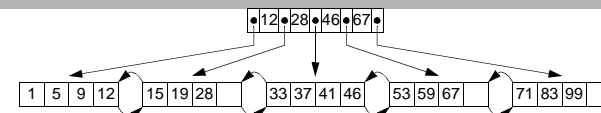
- entspricht fast immer einer exakten Suche und dem Zurückschreiben des modifizierten Blatts (Datenseite)

Manchmal treten aber folgende Problemfälle auf:

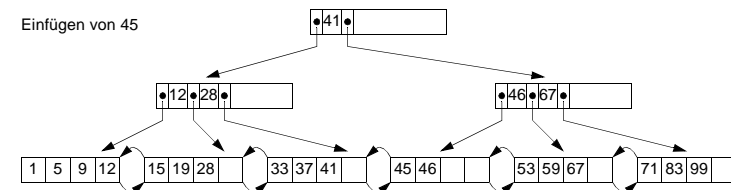
- Was passiert wenn die Seite keinen Datensatz mehr aufnehmen kann?
1. Lösung: Einführung von Überlaufseiten und verketten mit der Primärseite.
 - Nachteil: Kosten für Suche, Einfügen und Löschen erhöhen sich.
 2. Lösung: Reorganisation der Datenstruktur
 - sofort: Überlaufseiten werden nicht zugelassen. Reorganisation des B⁺-Baums soll aber lokal begrenzt bleiben.
 - später: kurzzeitige Verwendung von Überlaufseiten u. globale Reorganisation des Datenbestands.
- Was passiert wenn es zu wenige Datensätze in der Seite gibt?
1. Lösung: Verschmelzen der unterfüllten Seiten, aber möglichst mit benachbarten Seiten.

137.

Beispiel



Einfügen von 45



wichtige Eigenschaften:

- Einfügeoperation bleibt auf einen Pfad des B⁺-Baums beschränkt.
- beim Einfügen bleiben die Invarianten eines B⁺-Baums erhalten.

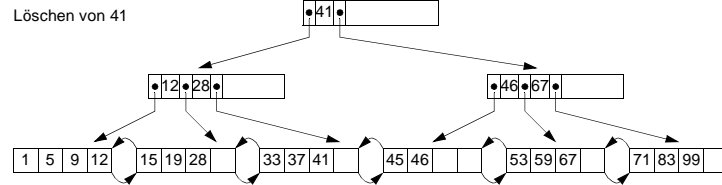
139.

Löschen im B⁺-Baum

- gegeben ein Schlüssel k und die Wurzel des B⁺-Baums. Finde den Datensatz mit Schlüssel k im B⁺-Baum und entferne diesen.

Problemfälle:

- Wie kann verhindert werden, daß ein Knoten zu wenig Datensätze enthält?
- Was passiert, wenn ein Datensatz gelöscht wird, dessen Schlüssel auch als Referenz in einem Elternknoten benutzt wird?



140.

Kosten für Suchen, Einfügen und Löschen

- exakte Suche, Einfügen und Löschen sind auf einen Pfad beschränkt
- im schlechtesten Fall haben wir folgende Kosten:
 - exakte Suche: $O(\log_c N)$
 - Bereichsanfrage: $O(\log_c N + r/c)$
 - Einfügen: $O(\log_c N)$
 - Löschen: $O(\log_c N)$
- wieviele Datensätze können in einem B⁺-Baum der Tiefe 2 gespeichert werden?

Beispiel ($c = 200$, 4 KB pro Seite);

 - im schlechtesten Fall: $400 \cdot 200 \cdot 200 = 16 \cdot 10^6$ Datensätze, $8 \cdot 10^4$ Datenseiten = 320 MB Speicherplatz für die Blattebene des B⁺-Baums.
 - im Durchschnitt: Da Knoten zu etwa $2/3$ im Durchschnitt gefüllt sind, können voraussichtlich $400 \cdot 270 \cdot 270 = 29 \cdot 10^6$ Datensätze verwaltet werden. Es wird nun 430 MB an Speicherplatz für die Blattebene benötigt.
- in vielen Anwendungen: 2 Plattenzugriffe für exakte Suche

142.

Algorithmus

Suche nach einem Datensatz mit Schlüssel k ; (* siehe EMQ(pact, k *))

IF (Datensatz wurde nicht gefunden) THEN Print("ERROR"); RETURN; END;

Setze pact auf das zuletzt gelesene Blatt;

Entferne den Datensatz mit Schlüssel k aus pact;

IF (pact enthält zu wenig Datensätze) THEN

IF (einer der Nachbarknoten enthält mehr als b Datensätze) THEN

teile die Datensätze, die sich in pact und in dem Nachbarknoten befinden, gleichmäßig zwischen diesen Knoten auf;

modifiziere den dazugehörigen Trennschlüssel im Elternknoten;

ELSE

Speichere die Datensätze aus pact in einem Nachbarknoten;

Entferne den Eintrag aus dem Elternknoten, der auf pact verweist;

IF (Elternknoten enthält zu wenig Einträge) THEN ...

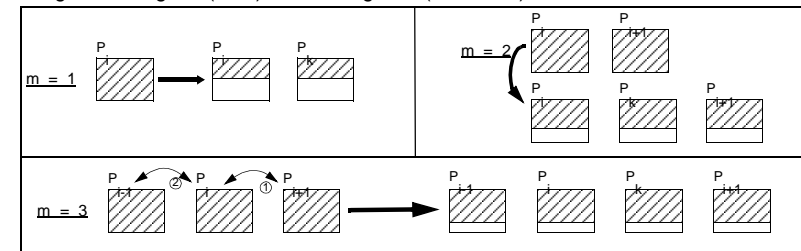
wichtige Eigenschaft:

- Löschen ist auf einen Pfad beschränkt

141.

5.3.3 Verbesserung der SPAN von B⁺-Bäumen

- Statt aufteilen einer vollen Seite auf zwei, werden die Datensätze aus m Seiten gleichmäßig auf $(m+1)$ Seiten aufgeteilt (**B⁺-Baum**).



- erheblich höhere Komplexität der Algorithmen: $m \leq 3$ guter Kompromiß

SPAN	$m = 1$	$m = 2$	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	$\ln 2$ (69 %)		$m \cdot \ln\left(\frac{m+1}{m}\right)$

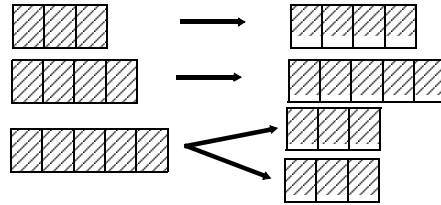
143.

Elastische Seiten

Idee (Larson, 1989)

- Eine Seite besteht zunächst aus r Sektoren, wobei jeder der Sektoren eine Kapazität von b Sätzen besitzt.
- Gibt es einen Überlauf in einer Seite mit j Sektoren, $r \leq j < 2r-1$, so wird eine Seite mit $j+1$ Sektoren reserviert und die Datensätze in die größere Seite übertragen.
- Gibt es einen Überlauf in einer Seite mit $2r-1$ Sektoren, so werden die Datensätze auf zwei Seiten mit jeweils r Sektoren übertragen.

Beispiel: $r = 3$



- Jeder Schritt wird auch als **partielle Erweiterung** bezeichnet und der Prozeß des Anwachsens einer Seite von ihrer kleinsten Größe bis zum Zeitpunkt des Splits nennt man eine **vollständige Erweiterung**.

144.

Präfix B⁺-Bäume

Ziel beim Entwurf eines B⁺-Baums

- möglichst wenig Speicherplatz für die Zwischenknoten (ggf. kann dann der gesamte Index im Hauptspeicher gehalten werden).
- möglichst geringe Tiefe des B⁺-Baums (geringe Zugriffskosten)
- durch Erhöhung des Verzweigungsgrads in den Zwischenknoten des B⁺-Baums können die oben genannten Ziele erreicht werden.

Separatoren mit Präfixeigenschaft: einfache Präfix-B⁺-Bäume

- Beobachtung:
 - Im B⁺-Baum werden vollständige Schlüssel als Separatoren verwendet.
 - Kürzere Separatoren können die Separatoreigenschaften genauso gut erfüllen.
 - Kürzere Separatoren erhöhen den Verzweigungsgrad.

146.

Vergleich B^{*}-Baum und B⁺-Baum mit elastischen Seiten

- durchschnittliche SPAN ist höher bei Verwendung von partiellen Erweiterungen:

SPAN	Anzahl der partiellen Erweiterungen (B ⁺ -Baum) Anzahl der Seiten beim Split (B [*] -Baum)		
	1	2	3
avg. case B [*] -Baum	0.69	0.81	0.86
avg. case B ⁺ -Baum	0.69	0.84	0.89

- B^{*}-Baum muß beim Überlauf stets benachbarte Seiten lesen und dann schreiben: höhere Kosten beim Einfügen und Löschen von Datensätzen.
- Partielle Erweiterungen sind einfacher zu implementieren als das Aufteilen der Datensätze über benachbarte Seiten beim B⁺-Baum.
- Verwaltung des Plattenspeicher (Freispeicherliste) ist aber i.a. etwas komplizierter bei elastischen Seiten und kann zu einer leichten Verschlechterung der oben angegebenen SPAN führen.

145.

Präfix

Beispiel:

- Datensatz "Database" soll in die folgende, volle Seite eingefügt werden:

Compiler	Computer	Computing	Design	Directory
----------	----------	-----------	--------	-----------

- Ein Trennschlüssel (Separator) muß gefunden werden, so daß die Datensätze gleichmäßig über zwei Seiten verteilt werden, z. B. "Computing".
- Jeder Schlüssel K mit "Computing" $\leq K <$ "Database" ist ein Separator mit der gewünschten Eigenschaft. Deshalb:
 - wähle den kürzesten Separator

Definition:

Seien die Schlüssel Worte über einem vorgegebenen Alphabet und sei die lexikographische Ordnung die Ordnungsrelation " $<$ " auf den Schlüssel. Dann ist y ein Präfix für die Schlüssel x und z , falls folgende Bedingungen erfüllt sind:

- $x \leq y < z$
- kein anderer Separator zwischen x und z ist kürzer als y .

- Beispiel (oben): $y = "D"$

147.

5.3.4 Effiziente Unterstützung von Bereichsanfragen

Können Bereichsanfragen noch effizienter als durch B⁺-Bäume unterstützt werden?

Entscheidend ist das Maß für die Effizienz:

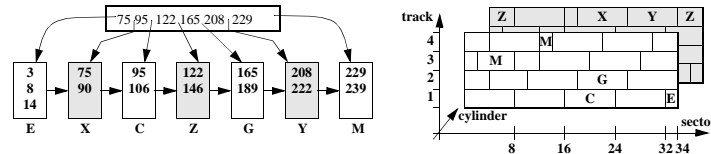
1. Ist das Maß für die Effizienz "nur" die Anzahl der Zugriffe, so ist der B⁺-Baum bereits bzgl. der Anzahl der Datenseitenzugriffe nahezu optimal.
 - es wird hier implizit unterstellt, daß Zugriffe gleich teuer sind.
2. Wird nun berücksichtigt, daß die Kosten für einen Seitenzugriff sich aus den Komponenten Suchzeit, Rotationsverzögerung, Transferzeit und Kopfschaltzeit zusammensetzen, so ist der B⁺-Baum i.a. sehr ineffizient.
 - Transferzeit ist konstant; Suchzeit und Rotationsverzögerung hängen von der vorhergehenden Position des Plattenarms ab.
 - Optimierungsziel: Reduziere möglichst Suchzeit u. Rotationsverzögerung.

Antwort auf Ausgangsfrage: **NEIN**

Antwort: **JA**

148.

□ nachdem alle Sätze eingefügt sind:



- Nachbarseiten liegen stets auf verschiedenen Zylindern: Suchzeit pro benötigter Seite ist hoch.
- zwei benötigte Seiten, die gemeinsam in einem Zylinder liegen, werden nicht gemeinsam eingelesen.

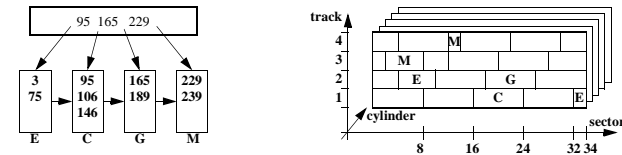
Ziele beim Entwurf einer effizienteren Zugriffsstruktur:

1. Speichere Seiten, die nahe beieinanderliegende Datensätze enthalten, auch physisch nah beieinander auf der Platte ab (globale Ordnungserhaltung).
2. Lese relevante Seiten, die auf einem Zylinder liegen und möglicherweise Antworten enthalten, in einem Zugriff (Mehrseiten-Zugriff).
3. Wähle eine möglichst effiziente Reihenfolge zum Lesen der Seiten einer Mehrseiten-Anforderung.

150.

Beispiel

- Voraussetzungen:
 - Platte besteht aus 5 Zylindern, ein Zylinder aus 4 Spuren und eine Spur aus 34 Sektoren. Eine Seite setzt sich aus 8 Sektoren zusammen.
 - Kopfschaltzeit = 4 Sektoren
 - Zylinder kann maximal 4 Seiten pro Datei aufnehmen.
 - Kapazität einer Datenseite ist 3 und einer Indexseite ist 8.
 - Strategie von UNIX für die Allokation von Seiten.
 - Datensätze: 146, 75, 3, 95, 189, 165, 106, 229, 239, 14, 208, 90, 8, 222, 122
- Situation nachdem Datensatz 239 eingefügt wurde:



- für eine Bereichsanfrage ist die Suchzeit pro benötigter Seite niedriger als erwartet.

149.

Große Datenseiten

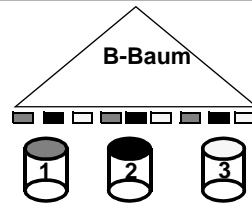
- Durch eine Verdopplung der Seitengröße ergibt sich folgendes Verhalten:
 - Bei einer Bereichsanfrage wird die Anzahl der Zugriffe auf die Datenseiten nahezu halbiert.
 - Der Aufwand für eine exakte Suche, Einfügen und Löschen erhöht sich um die zusätzlichen Transferkosten.
 - die Anzahl der benötigten Zwischenknoten wird niedriger (ggf. auch Baumtiefe)
 - Es ergibt sich ein erhöhter Aufwand beim Durchsuchen der Seite.
- Beachte: Die Datenseitengröße hat keinen Einfluß auf die erwartete SPAN.
- Da die Transferkosten einer Seite (8 KByte) sehr niedrig sind im Vergleich zu den Zugriffskosten ist es fast immer vorteilhaft große Seiten zu benutzen.

Problem:

- Dateisystem muß aber garantieren, daß eine logisch zusammenhängende Seite auch auf der Platte zusammenhängend ist.
- DBS unterstützen oft nur eine Seitengröße, da dies insbesondere die Pufferverwaltung wesentlich vereinfacht.

151.

5.3.5 Parallele B+-Bäume



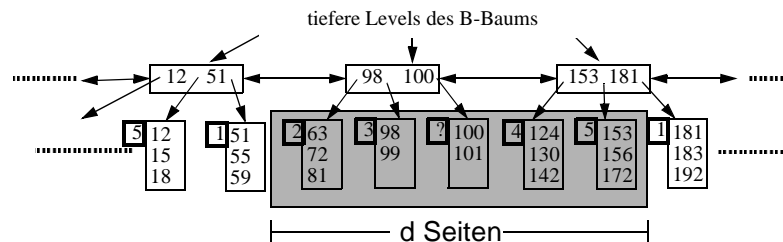
Problem:

- Gegeben ein B+-Baum: Verteile die Datenseiten des B-Baums über die Platten

folgende Techniken können genutzt werden:

- verteile die Datenseiten "zufällig" über die Platten
- "round robin"-Verteilung: Seite i wird auf Platte $i \bmod d + 1$ abgelegt
- "lokale Lastbalancierung"

156.

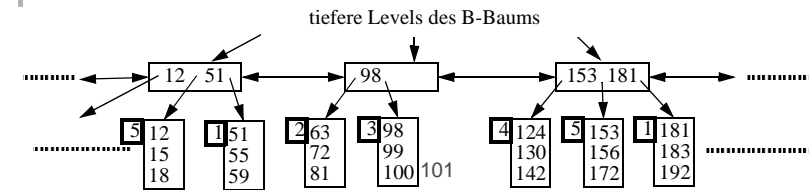


Vorgehen bei der Auswahl der Platte

- Lege ein Fenster um die neue Seite, welches d Seiten beinhaltet. Alle Platten, die keine Seite aus diesem Fenster enthalten, werden in einer Kandidatenmenge Z gehalten.
- Die Seite wird dann auf der Platte $\min \{j \in Z \mid \min \{\text{load}[i] \mid i \in Z\} = \text{load}[j]\}$ abgespeichert, wobei $\text{load}[r]$ die Anzahl der bereits gespeicherten Seiten auf der Platte r bezeichnet, $1 \leq r \leq d$.

158.

Lokale Lastbalancierung



Optimale Verteilung (bzgl. Bereichsfragen):

- Seiten werden zyklisch (bzgl. der Ordnung auf dem Level) den Platten zugeordnet. Antwortzeit einer Bereichsanfrage mit d Platten:

$$\lceil q_{seq}/d \rceil$$

dabei ist q_{seq} die Antwortzeit, wenn für die gleiche Anfrage ein B+-Baum auf einer Platte benutzt wird.

- zyklische Zuordnung ist nur im statischen Fall möglich
- Prinzipielle Idee im dynamischen Fall: Eine durch einen Split neu erzeugte Seite wird auf die Platte geschrieben, die keine "benachbarte" Seiten enthält.

157.

Leistung paralleler B-Bäume

- Datenlast ist nahezu gleichmäßig über die Platten verteilt.
- Im schlechtesten Fall kann eine Platte $2N/p$ Seiten eines Levels des B-Baums enthalten, wobei N die Gesamtanzahl der Seiten in diesem Level ist.
- Anwortzeitverhalten einer Bereichsanfrage ist im schlechtesten Fall

$$\lceil 2q_{seq}/d \rceil$$

- Durchsatz von Bereichsanfragen ist sehr hoch (kleine Bereichsanfragen benutzen nur wenige Platten)
- wenig Mehraufwand beim Einfügen und Löschen

159.

5.3.6 Bulk Loading

- ❑ Im folgenden bezeichnet b den maximalen Füllgrad eines Blattes im B+-Baum.
- ❑ Insgesamt werden zum Massenaufbau $\Theta(\log_b n)$ Pufferseiten benötigt.

Algorithmus (Parameter F mit $0.5 \leq F \leq 1$)

1. Sortiere die Daten aufsteigend (i. a. auf dem Externspeicher)
2. SOLANGE (die in Schritt 1 sortierte Folge weitere Daten enthält)
 - Lies die nächsten $b \cdot F$ Datensätze ein;
 - Speichere die Datensätze gemeinsam in einem neuen Blatt;
 - Häng das Blatt rechts an den B+-Baum;
 - Füge einen entsprechenden Indexeintrag in den Vaterknoten;

Aufwand

- ❑ Parameter F steuert den Füllgrad der Blattknoten.
- ❑ Schritt 1: Aufwand für externes Sortieren (siehe Kapitel 4)
- ❑ Schritt 2 des Verfahrens benötigt nur linearen Aufwand in der Anzahl der Datenseiten

160.

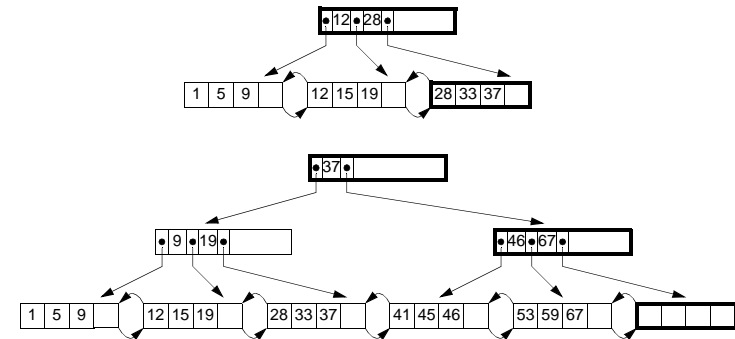
Zusammenfassung

- ❑ B⁺-Bäume sind **die** Zugriffsstruktur in heutigen DBS
- ❑ B⁺-Bäume bieten folgende Eigenschaften:
 - SPAN liegt stets über 50% und im Durchschnitt sogar bei 69%
 - die Höhe des B⁺-Baums ist $O(\log_b N)$, b = Seitenkapazität, N = #Datensätze.
 - exakte Suche, Einfügen und Löschen ist auf einen Pfad beschränkt
 - Bereichsanfragen werden sehr effizient beantwortet
- ❑ Die Familie der B+-Bäume ist sehr groß!
 - Verkleinerung des Index: Präfix-B Bäume, Verwendung von großen Seiten, ...
 - Erhöhung der Speicherplatzausnutzung: B-Bäume mit elastischen Seiten (partiellen Erweiterungen), B*-Bäume, ...
 - Verbesserung der Effizienz bei Bereichsanfragen: VSAM, B⁺-Baum mit großen Datenseiten, CB⁺-Baum, ...
- ❑ offenes Problem: Mehrbenutzerbetrieb auf einem B⁺-Baum (dazu später mehr).

162.

Beispiel

- ❑ Parameter: $b = 4$, $F = 0.75$
- ❑ Datensätze: 1, 5, 9, 12, 15, 19, 28, 33, 37, 41, 45, 46, 53, 59, 67, 71, 83, 99
- ❑ Schritt 2



161.

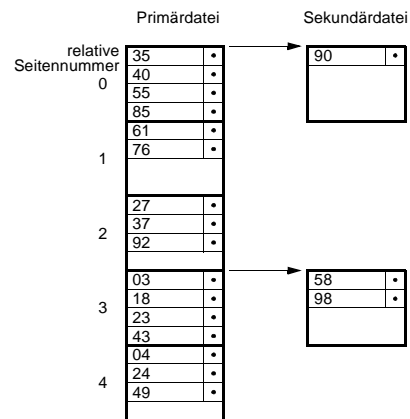
5.4 Statische Hashverfahren

- ❑ direkte Berechnung der Speicheradresse (Seitenadresse) eines Satzes durch eine Schlüsseltransformation
- ❑ Hashfunktion $h: D \rightarrow \{0, 1, \dots, n-1\}$
 D = Schlüsselraum, n = Größe des **statischen** Hashbereiches in Seiten (**Buckets**)
- ❑ Idealfall: h ist injektiv (keine Kollisionen)
 - nur in Ausnahmefällen möglich (wegen des Geburtstagsparadoxons)
 - jeder Satz kann mit 1 Seitenzugriff referenziert werden
- ❑ Statische Hashverfahren mit **Kollisionsbehandlung**
 - vorhandene Schlüsselmenge K ($K \subseteq S$) soll möglichst gleichmäßig auf die n Buckets verteilt werden.
 - Behandlung von **Synonymen**:
 - Aufnahme im selben Bucket, wenn möglich
 - ggf. spezielle Behandlung von sogenannten **Überlaufsätzen**
- ❑ Anforderung an Hashverfahren
 - möglichst hohe SPAN
 - möglichst wenig Zugriffe für exakte Suche, Einfügen und Löschen

163.

Getrennte Verkettung der Überläufer

- $S = \{0, \dots, 99\}$, $n = 5$, $h(K) = K \text{ MOD } 5$



Notation

- $N = \# \text{Sätze}$
- $n = n_p = \# \text{Buckets}$
- $n_s = \# \text{Sekundärseiten}$
- $b = \text{Kapazität der Primärseiten}$
- $c = \text{Kapazität der Sekundärseiten}$ (Annahme: $c = b$)
- Belegungsfaktor:

$$\beta = \frac{N}{n \cdot b}$$

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

164.

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: Hashverfahren mit Verkettung:

166.

Überlaufbehandlung ohne Verkettung

Lineares Sondieren (linear probing)

Ein Satz k wird in die erste nicht-volle Seite mit Adresse $(h(k) + i) \text{ MOD } n$ eingefügt, $i = 0, \dots, n-1$.

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	•
	90	•
	98	•
2	27	•
	37	•
	92	•
	90	•
3	03	•
	18	•
	23	•
	43	•
4	04	•
	24	•
	49	•
	58	•

165.

Vergleich der Verfahren

erwartete Kosten erfolgreiche Suche, $b = 10$ erwartete Kosten erfolglose 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

	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

167.

Zusammenfassung

für statische Dateien gilt:

- ❑ statische Hashverfahren 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 Hashverfahren sind ungeeignet, da
 - durch viele Einfügeoperationen Ketten zu lang werden können.
 - durch viele Löschoperationen die SPAN zu niedrig wird.
 ⇒ globale Reorganisation der Datei.
- ❑ dynamische Hashverfahren beseitigen diese Defizite durch lokale Reorganisationen.

168.

5.5.1 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**)
- ❑ Hashfunktion $h_d(K)$, $d \geq 0$
 - zunächst wird für ein Schlüssel K ein **Pseudoschlüssel** ($b_0, b_1, \dots, b_j, \dots$) 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 wird auch als **globale Tiefe** des Directory bezeichnet); Eintrag verweist auf eine Seite, in dem alle zugehörigen Sätze gespeichert sind.

170.

5.5 Dynamische Hashverfahren

❑ Ziel:

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

❑ Beispiel:

- $h(K) = K \text{ MOD } 5$ ist die bisherige Hashfunktion.
- wenn $h(K) = K \text{ MOD } 7$ die neue Hashfunktion ist, müssen alle (?) Adressen neu berechnet werden.

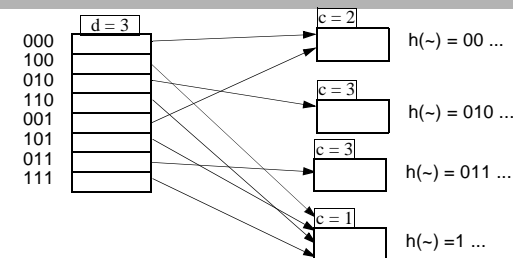
==> globale Reorganisation

❑ dynamische Hashverfahren unterscheiden sich im wesentlichen in folgenden Punkten:

- **Größe der Kapazität einer Primärseite**: für $b = 0$, spricht man auch von Verfahren mit Hashtabelle (Directory).
- **Überlaufbehandlung**: getrennte oder gemeinsame Verkettung
- Wahl der Folge von **Hashfunktionen**

169.

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.

171.

Einfügen eines Datensatzes

□ Gegeben: Datensatz mit Schlüssel K

1. Berechne die ersten d Bits des Pseudoschlüssels (b_0, b_1, \dots).

$$a - 1$$

2. Lies die Seitenreferenz S aus dem Eintrag $\sum_{j=0}^{a-1} b_j \cdot 2^j$ 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.

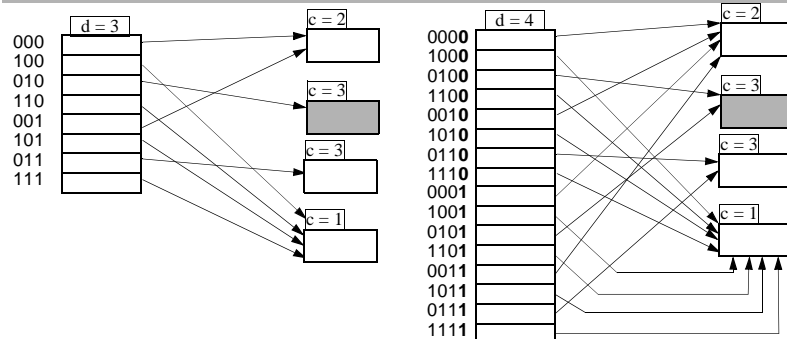
4. Füge den Datensatz in die Seite S ein und prüfe, ob es ein Überlauf gibt.

5. Wenn ja, dann verfähre 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 angeforderte Seite.
- Ändere die entsprechenden Seitenreferenzen im Directory.
- Falls immer noch ein Überlauf in einer der Seiten vorliegt, so wiederhole diesen Schritt entsprechend.

172.

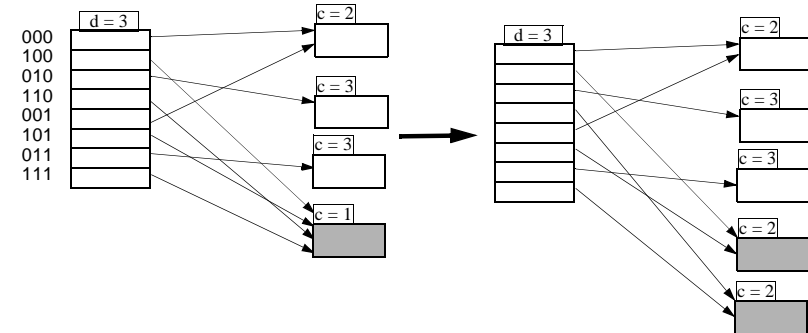
Verdoppelung des Directory



174.

Beseitigung eines Überlaufs

in einer Seite mit lokaler Tiefe $<$ globaler Tiefe:



173.

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* $= \ln 2 \approx 0.693$ (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.

175.

5.5.2 Lineares Hashing

- Verfahren mit $b > 0$, d.h. es gibt eine Primärdatei und eine Sekundärdatei.
 - benutzt somit kein Directory!
 - Primärdatei besteht am Anfang aus n 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 Hashfunktionen
 - Folge von Hashfunktionen $\{h_j(K)\}_{j \geq 0}$ mit folgenden Bedingungen:

Bereichsbedingung:

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

Splitbedingung:

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

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

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

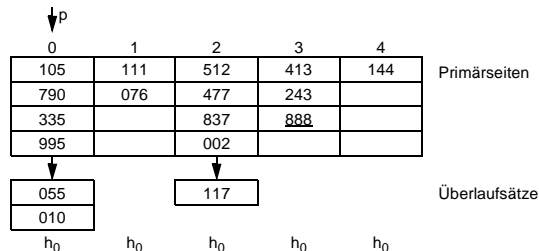
176.

- Beispiel:
 - $h_j(K) = K \bmod (2^j n)$ 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 aufgespalten werden soll.
- folgende Daten sind für lineares Hashing notwendig:
 - L : **Level** (Anzahl der bereits ausgeführten Verdopplungen)
 - n : Anzahl der Seiten bei der Initialisierung des Hashverfahrens
 - p : zeigt auf nächste zu splittende Primärseite, $0 \leq p < n \cdot 2^L$
 - β : Belegungsfaktor
 - N : Anzahl der gespeicherten Sätze
 - b : Kapazität einer Primärseite
 - c : Kapazität einer Sekundärseite (i.a. gilt $c = b$).

177.

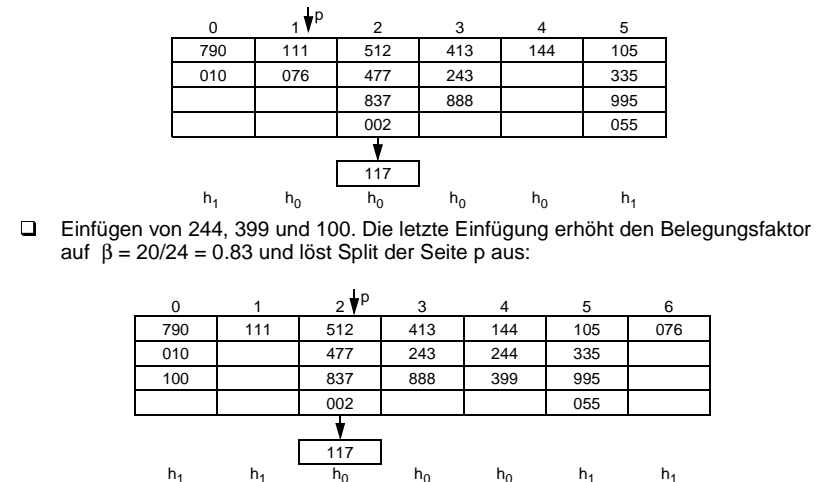
Beispiel: Prinzip des LH

- Vorgaben:
 - $n = 5$, $L = 0$, $b = 4$, $h_0(K) = K \bmod 5$ und $h_1(K) = K \bmod 10$
 - 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 $p + n \cdot 2^L$ abgelegt.

178.



179.

Erweitern der Datei

- ❑ wird ausgelöst durch eine Kontrollfunktion, 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 $p + n \cdot 2^L$ abgelegt.
- ❑ p wird um 1 erhöht: $p = (p+1) \text{ MOD } (n \cdot 2^L)$.
Falls $p = 0$, so hat sich die Anzahl der Seiten in der Datei verdoppelt und L wird nun um 1 erhöht.

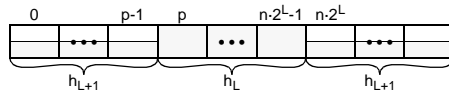
Split-Strategien:

- ❑ 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

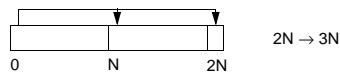
180.

5.5.3 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 Erweiterungsschritte
- ❑ Idee:
 - Ausgangspunkt: Datei mit $2N$ Buckets, die logisch in N Paare unterteilt ist: $(j, j+N)$ für $j = 0, 1, \dots, N-1$
 - erste partielle Erweiterung:



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

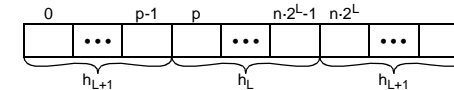
182.

Suchen

- ❑ Wenn $h_L(K) \geq p$, dann ist $h_L(K)$ die gewünschte Adresse
- ❑ Andernfalls ($h_L(K) < p$), dann ist die Seite bereits aufgeteilt worden und $h_{L+1}(K)$ liefert die gewünschte Adresse.

$h := h_L(K);$

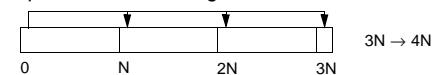
IF ($h < p$) THEN $h := h_{L+1}(K);$



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

181.

- zweite partielle Erweiterung



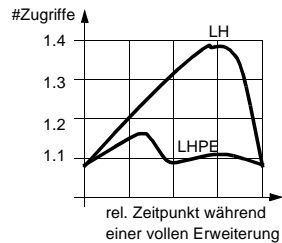
Verteilen der Sätze aus den Seiten $(j, N+j, 2N+j)$ auf die vier Seiten $(j, N+j, 2N+j, 3N+j)$.

- nach Abschluß der zweiten partiellen Erweiterung hat sich die Dateigröße verdoppelt und es kann wieder mit der ersten partiellen Erweiterung gestartet werden.
- ❑ Eigenschaften:
 - die Belegung in bereits aufgeteilten Seiten ist $2/3$ niedriger als die in noch nicht aufgeteilten Seiten der ersten partiellen Erweiterung.
 - nach der ersten partiellen Erweiterung 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 zweiten partiellen Erweiterung.
- ❑ allgemein: vollständige Erweiterung kann in n_0 partiellen Erweiterungen realisiert werden.

183.

Leistungsvergleich

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



	$n_0 = 1$	$n_0 = 2$	$n_0 = 3$
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

Überraschende Beobachtung

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

184.

Zusammenfassung

- LH mit partiellen Erweiterungen ist ein effizientes dynamisches Hashverfahren
 - 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
 - Spiralspeicherung
- erweiterbares Hashing sollte den Vorzug gegenüber LH erhalten, wenn die 2-Disk Suchgarantie wichtig ist.
- im Vergleich zu B⁺-Bäumen sind dynamische Hashverfahren effizienter, wenn Bereichsanfragen nicht oder kaum auftreten.
- B⁺-Bäume sollten immer dann benutzt werden, wenn eine sequentielle Verarbeitung der Daten wichtig ist.

186.

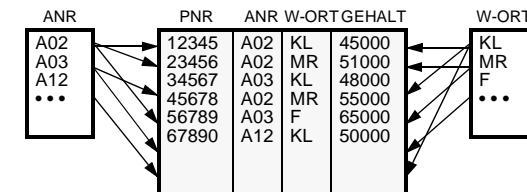
Historische Anmerkungen

- Erweiterbares Hashing wurde in dieser Form von Fagin, Pippenger, Nievergelt, Strong 1979 vorgestellt.
- Lineares Hashing stammt von W. Litwin; die Erweiterung um partielle Erweiterung von P. Larson. Beide Arbeiten wurden gleichzeitig veröffentlicht.
- Alle Arbeiten über dynamische Hashverfahren beruhen aber auf der Pionierarbeit von D. Knott, der bereits Anfang der 70er Jahre dynamische Hashverfahren vorstellte. Die Zeit war aber noch nicht reif für solche Ideen!

185.

5.6 Zugriffsstrukturen auf Sekundärschlüssel

- Suche nach Tupeln mit einem vorgegebenen Wert in einem Nicht-Schlüsselattribut (*Sekundärschlüssel*)
- i.a. mehr als 1 Treffer bei einer exakten Anfrage(Satzmengen)



- allgemeine Zugriffspfade auf Satzmenge anwendbar
- separate Speicherung der Satzverweise (TID) erlaubt effiziente Unterstützung mengenalgebraischer Operationen
 - Verwendung von invertierten Listen bzw. B⁺-Bäumen

187.

Sekundärsschlüssel-Zugriffspfade

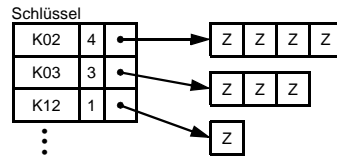
- Bei Trennung der Zugriffspfad-Daten von den Primärdaten kommen zwei Darstellungsmethoden in Betracht:

- gemeinsame Verwaltung der Suchstruktur und der Trefferlisten

Schlüssel		Zeigerlisten
K02	4	Z Z Z Z
K03	3	Z Z Z
K12	1	Z

⋮

- in der Suchstruktur ist nur ein Verweis pro Schlüsselwert vorhanden, der zu einer Liste mit Satzverweisen führt (Trefferliste)



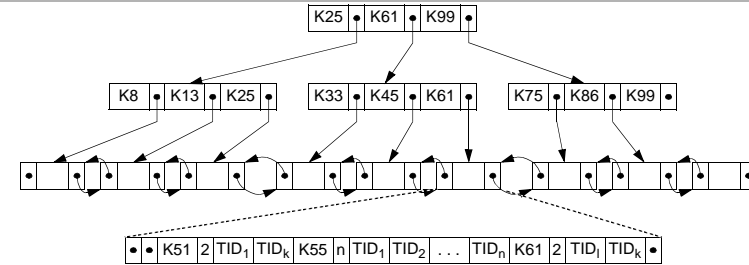
188.

Bearbeitung von Selektionsanfragen

- Bereichsanfragen und auch exakte Anfragen besitzen mehrere Antworten.
- Antworten sind meist verstreut über die Datendatei, die Seiten der Datendatei liegen wiederum auf wenigen Plattenzylindern.
- Variante 1:
 - Wenn ein Paar (K,TID) mit K erfüllt die Anfragebedingung gefunden wird, wird sofort auf den entsprechenden Datensatz auch zugegriffen.
- Variante 2:
 - Alle Paare (K,TID) werden erst in einem ersten Schritt bestimmt. Danach werden die TID sortiert und dann auf die entsprechenden Datensätze zugegriffen.
- Vergleich Variante 1 mit Variante 2
 - Variante 1 liefert schneller erste Resultate
 - für Variante 2 müssen die TIDs zusätzlich sortiert werden
 - Variante 2 ist i.a. erheblich effizienter als Variante 1, da Mehrfachzugriffe auf Seiten vermieden werden und die Zugriffszeit für das Lesen einer individuellen Seite (im Durchschnitt) sehr niedrig ist.

190.

Beispiel: B+-Baum



Relation PERS

B+-Baum

- wird auf dem Attribut ANR der Relation Pers angelegt.
- mit Sortierreihenfolge der Sekundärschlüssel (Bereichsfragen!) Vorwärts- und Rückwärtsverkettung

PERS (PNR,Name,ANR,...)

E1 Müller K55 ...
E17 Maier K51
E25 SchmittK55
⋮

189.

Variante 3:

- Es wird nun eine beschränkte Anzahl von TID im Hauptspeicher erlaubt.
 - Verwaltung der im Hauptspeicher liegenden TID in einem Heap.
 - Lesen des Datensatzes mit dem kleinsten TID.
 - Der vom Index gelieferte TID wird dann in einen Heap eingefügt.
- ⇒ Reduzierung der Kosten für individuellen Zugriff.

191.

Bearbeitung anderer Anfragen

- ❑ Berechnung von Aggregaten über dem Schlüssel des Index

```
select sum(Gehalt)
from Person
```

Statt auf die eigentliche Relation kann hierfür der Index verwendet werden.

- ❑ Verknüpfung von mehreren Indizes

```
select Name, Gehalt
from Person
```

Haben wir nun jeweils einen Index bzgl. des Attributs Name und Gehalt angelegt, so könnte diese Anfrage dadurch beantwortet werden, daß wir die Information aus beiden Indizes miteinander verknüpfen.

- ❑ Verarbeitung mehrdimensionaler Anfragen

```
select
from Person
where (Gehalt < 1000) and (Alter < 30)
```

Sind Indizes auf beiden Attributen definiert, so kann die Anfrage durch die Berechnung der entsprechenden TID-Mengen, die dann miteinander geschnitten werden, beantwortet.

192.

Leistung von Bitmaps

- ❑ Ist die Anzahl der Werte im Wertebereich klein (< 64), so ist die Verwendung einer Bitmap vom Speicherplatz günstiger als die Verwendung von TID.
- ❑ Einfache und effiziente Verarbeitung logisch zusammengesetzter Operationen:
 - Eine OR-Verknüpfung entspricht einem logischen OR zweier Bitvektoren.
 - Eine AND-Verknüpfung entspricht einem logischen AND zweier Bitvektoren.
- ❑ Besitzen die Datensätze in der Relation eine feste Länge (was i. A. aber nicht der Fall ist) und gibt es in jeder Seite c Datensätze, dann kann die Positionen der durch einen Bitvektor referenzierten Datensätze einfach berechnet werden.

Weiterer Aspekt

- ❑ Kompression einer Bitmap

194.

Bitmaps

- ❑ Neben dem Konzept durch Zeiger (TID) auf qualifizierende Datensätze zu verweisen, haben sich gerade im Bereich Data Warehousing sogenannte Bitmaps als sehr effizient erwiesen.
- ❑ Betrachten wir eine Relation R mit N Datensätzen und einen Index (z. B. B+-Baum) auf dem Attribut A von R. Der Wertebereich von A setze sich dabei aus Werten $\{a_1, \dots, a_k\}$ zusammen. Wir betrachten eine $k \times N$ -Matrix B mit:

	1	2	...	N
a_1	0	0	...	0
a_2	1	0	...	0
...	0	1	...	0
a_k	0	0	...	1

Dabei ist $B[i,j] = 1$, falls der j-te Datensatz der Relation R den Wert a_i im Attribut annimmt. Offensichtlich werden gerade $j \cdot N$ Bits benötigt, um die Matrix B zu speichern. B wird auch als Bitmap bezeichnet.

193.

Zugriffsstrukturen in kommerziellen DBS

- ❑ Stand vor etwa fünf Jahren

DB2 (IBM)	B*-Baum (clustered, non-clustered), partitionierte Relationen
INGRES	B-Baum, statisches Hashing, ISAM, HEAP
ORACLE	B*-Baum (mit Präfix-/Suffix-Komprimierung), (Join-) Clusterbildung
SYBASE	B*-Baum (clustered, non-clustered)
RDB (DEC)	B*-Baum (clustered, non-clustered), Hashing, Join-Clusterbildung
NonStop SQL (Tandem)	B*-Baum (clustered, non-clustered) mit Präfix-Komprimierung
UDS (Siemens)	B*-Baum, statisches Hashing, Clusterbildung (LIST), Invertierung (Pointer-Array), Kettung (CHAIN)

195.

Zusammenfassung

- ❑ Clusterbildung optimiert (sortiert) sequentielle Zugriffe
- ❑ Standard-Zugriffspfadstruktur in DBS: B⁺-Baum ("the ubiquitous B⁺-tree")
 - Familie der B⁺-Bäume ist sehr groß
 - Varianten optimieren SPAN, Zugriffskosten für exakte Anfragen oder Bereichsanfragen.
- ❑ schnellerer Schlüsselzugriff erfordert Hashverfahren
 - (nur) direkter Zugriff (= 1 Seitenzugriff)
 - Erweiterbares Hashing und LH unterstützen stark wachsende Datenbestände (<= 2 Seitenzugriffe)
- ❑ Zugriffspfade für Sekundärschlüssel
 - Einstiegsstruktur: B⁺-Baum u.a.
 - Unterstützung mengentheoretischer Operationen

196.

B.2 Massenaufbau von Hashverfahren

- ❑ vereinfachende Annahmen
 - statisches Hashverfahren mit Verkettung
 - Hashfunktion: $h(K) = K \bmod n$ mit $n = 2^k$
 - m = Anzahl der im Hauptspeicher verfügbaren Seiten mit $m = 2^j$
- ❑ Skizze des Algorithmus
 - Initialisiere ein Hashverfahren mit m Behältern. Verwende als Hashfunktion $h(K) = K \bmod m$.
 - Tritt ein Überlauf auf, so wird die volle Seite auf den Externspeicher geschrieben und eine leere Seite erzeugt, wo der Datensatz abgelegt wird.
 - Nachdem alle Datensätze eingefügt wurden, werden alle Seiten auf den Externspeicher geschrieben.
 - Das Verfahren wird nun rekursiv für jeden Behälter angewendet.
- ❑ Abbruch der Rekursion bei der Tiefe x , falls

$$(m^x \geq n) \quad \text{und} \quad (m^{x-1} < n) \Rightarrow x = \lceil \log_m n \rceil$$

198.

B. Massenaufbau von Indexstrukturen

Problemstellungen

Gegeben eine Menge von Datensätzen.

- ❑ Erzeuge einen neuen Index auf den Datensätzen.
- ❑ Füge die Datensätze in einen bereits existierenden Index ein.
- ❑ Erzeuge eine Menge von Indizes auf den Datensätzen

Beobachtung

- ❑ Ausführung eines Massenaufbaus durch Hintereinanderausführung der Einfügeoperation ist zu teuer.

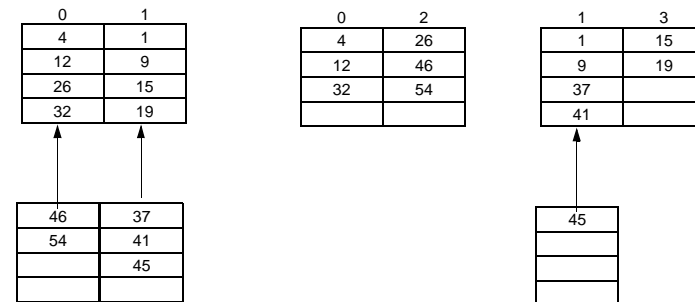
Bemerkung

- ❑ Dieses Thema ist derzeit aktueller Forschungsgegenstand. Es werden deshalb nur einige Ideen skizziert.

197.

Beispiel:

- ❑ Parameter: $b = 4, m = 2, n = 4$
- ❑ Datensätze: 1, 4, 9, 12, 15, 19, 26, 32, 37, 41, 45, 46, 54
 - 1. Rekursion
 - 2. Rekursion



- ❑ Wie im Beispiel gezeigt, kann die Primärseite der Hashtabelle nicht vollständig gefüllt sein, obwohl es Überlaufseiten gibt.
 - ⇒ Austausch der Primärseite und der letzten Seite in der Kette

199.

Hybrider Massenaufbau bei Hashverfahren

- ❑ Idee:
Es wird nun bereits im ersten Rekursionsschritt ein Teil der Zielstruktur aufgebaut.
- ❑ Beispiel:
Der verfügbare Hauptspeicher wird so aufgeteilt, daß z. B. die Hälfte des Speichers zum Aufbau des eigentlichen Hashverfahren verwendet wird und die andere Hälfte für den eigentlichen Iterationsschritt:
 - Falls $h(K) = K \bmod n \in \{0, \dots, m/2-1\}$, so werden die Datensätze in dem Behälter $h(K)$ abgespeichert.
 - Ansonsten, berechne $g(K) = K \bmod m/2$. Speichere den Datensatz im Behälter $g(K) + m/2$.

Vorteil des Verfahrens

- ❑ Liegt die verfügbare knapp unter der Speicherkapazität, um den Aufbau komplett im Hauptspeicher ablaufen zu lassen, so wird durch die oben beschriebene Technik es vermieden, daß ein großer Anteil der Behälter geschrieben und gelesen wird.
- ❑ Wieviel Speicher sollte im ersten Schritt nun tatsächlich dazu verwendet werden, um Behälter der Zielstruktur zu erzeugen?