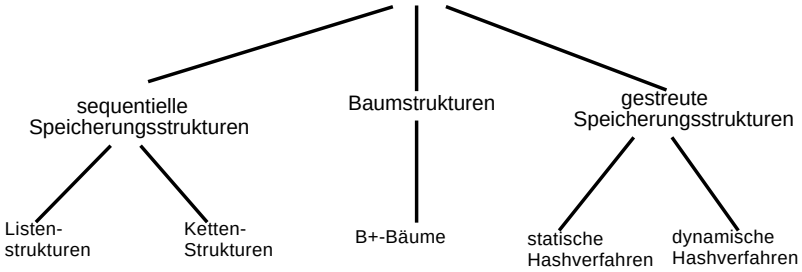
2. Eindimensionale Zugriffsstrukturen

- Anforderungen
- eindimensionale Zugriffsstrukturen:
  - B^+- Bäume und Varianten
    - erweitertes Splitting
    - Schlüsselkomprimierung
  - Hash-Verfahren
    - statische Verfahren
    - Externes Hashing mit Separatoren
    - Erweiterbares Hashing
    - Lineares Hashing
    - Spiralen-Hashing

Überblick



Anforderungen

allgemeine Ziele beim Entwurf von Zugriffsstrukturen:

- hohe Speicherplatzausnutzung
- kurze Antwortzeiten für eine Operation
  - benötigte Zeit wird in der Anzahl der Seitenzugriffe ausgedrückt.

Operationen

- Suchanfragen
  - Daten werden nur eingelesen, aber nicht verändert:
    exakte Suche, Bereichssuche
- Einfügen, Löschen und Ändern
  - erfordert Reorganisationen des Datenbestands.
  - Reorganisationen sollen nur lokal auf einem kleinen Teil des Datenbestands einwirken (**dynamische** Zugriffsstrukturen)

Bäume

- wichtige Datenstruktur für Hauptspeicher und Hintergrundspeicher (siehe Info II)

Ein **Baum** ist eine endliche Menge T von Elementen, **Knoten** genannt, mit:

(1) Es gibt einen ausgezeichneten Knoten w(T), die **Wurzel** von T

(2) Die restlichen Knoten sind in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ zerlegt, die ihrerseits Bäume sind. $T_1, \ldots, T_m$ heißen **Teilbäume** der Wurzel w(T).

Der **Grad** eines Knotens *x*, ***deg*(*x*)**, ist gleich der Anzahl der Teilbäume von *x*. Gilt *deg*(*x*) = 0, so nennt man *x* ein **Blatt**.

Jeder Knoten *x außer w(T)* hat einen eindeutigen Vorgänger *v(x)*.

s(x) bezeichnet die Menge aller Nachfolger von *x*.

b(x) die Menge aller Geschwister von *x*.

Ein **Pfad** in einem Baum ist eine Folge von Knoten $p_1, \ldots, p_n$ mit: $p_i = v(p_{i+1})$, i = 1, …, *n*-1. Die **Länge des Pfades** ist *n*.

Der **Level** eines Knotens *x*, ***lev*(*x*)** ist: $lev(x) = \begin{cases} 1 & \text{für } x = w(T) \\ lev(v(x)) + 1 & \text{für } x \neq w(T) \end{cases}$

Damit ist *lev*(*x*) gleich der Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten *x*.

Die **Höhe** eines Baumes ist gleich dem maximalen Level seiner Knoten. Dies entspricht der Länge des längsten von der Wurzel ausgehenden Pfades innerhalb des Baumes.

Ein **binärer Baum** ist eine endliche Menge B von Knoten, die

– entweder leer ist
– oder aus einer Wurzel und zwei disjunkten binären Bäumen besteht, dem linken und dem rechten Teilbaum der Wurzel.

Wichtiges Resultat für binäre Bäume:

Für n Datensätze ist die **minimale Höhe** eines binären Baums $\lceil \log_2(n+1) \rceil$

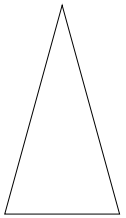
Es gibt binäre Bäume (z. B. AVL-Bäume), die folgendes Leistungsverhalten aufweisen:

– Höhe: O(log n)
– Kosten für exakte Suche, Einfügen und Löschen: O(log n)

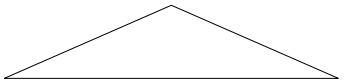
Die schlechte Nachricht!

❑ Binärbaumstrukturen sind für die Organisation von Daten auf dem Hintergrundspeicher nicht geeignet.
  – zu hoch (z. B. für 10^6 Datensaätze beträgt die Höhe bereits 20)
  – im schlechtesten Fall ist ein Knotenzugriff gleich einem Plattenzugriff

binärer Baum



ideale Baumstruktur für den Externspeicher



Zentrale Frage (bis Anfang der 70er Jahre):

❑ Gibt es eine effiziente Zugriffsstruktur für einen seitenorientierten Externspeicher?

2.1 B^+-Bäume

❑ im Gegensatz zu binären Bäumen enthält ein Knoten viele Einträge/Sätze
  – 1:1-Beziehung zwischen Knoten und Seiten
  – Daten liegen exklusiv in den Blättern
❑ Vorfahr:
  – ISAM (ist jedoch statisch und benötigt periodisch globale Reorganisation)
  – B-Baum (Bayer & McCreight, 1972)
❑ Suchfunktionen:
  – direkter Schlüsselzugriff: exact match query
  – sortiert sequentieller Zugriff: range query
❑ Effizienz (Speicherplatz u. Antwortzeiten) ist asymtotisch unabhängig von der Einfügereihenfolge.
❑ Verbesserung der Baumbreite (fan-out)
  – Schlüsselkomprimierung
  – Präfix-B-Bäume
❑ Erhöhung des Belegungsgrads durch verallgemeinerte Splittingverfahren
  – elastische Seiten
  – B*-Baum

Definition:

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

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

❑ Zwischenknoten:

Z_0	S_1	Z_1	S_2	Z_2	• • •	S_m	Z_m	frei

  – Z_i = Zeiger Sohnseite, S_i = Schlüssel
  – es gilt stets: $S_i < S_{i+1}$ für $0 < i < m$.

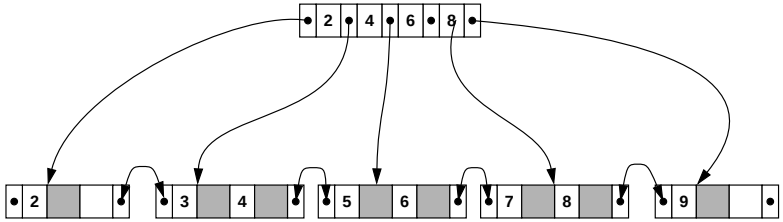
❑ Blattknoten:

V	S_1	D_1	S_2	D_2	• • •	S_m	D_m	frei	N

  – D_i = Verweis auf Satz, N =Nachfolger-Zeiger, V =Vorgänger-Zeiger
  – es gilt stets: $S_i < S_{i+1}$ für $0 < i < m$.

Beispiel

- b=2, b*=1
- Beachte: b und b* 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

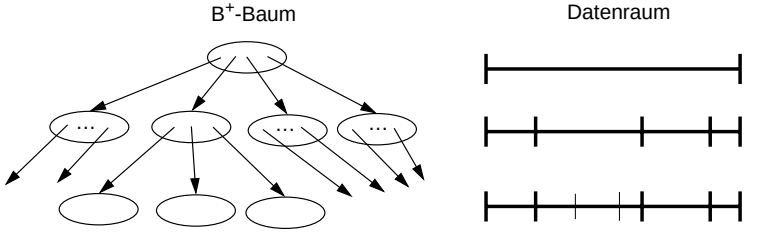
  dies ergibt ca. 500 Einträge in einem Zwischenknoten.

Eigenschaften des B^+-Baums

- lokale Ordnungserhaltung:

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

  Für jedes i, $1 \le i \le j$, 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.



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: b+1 = b*

Wurzel hat mindestens	2 Einträge
Zwischenknoten in der Ebene 2 hat mindestens	b +1 Einträge
Zwischenknoten in der Ebene 2 hat mindestens	b +1 Einträge
…	b +1 Einträge
Blattknoten in der Ebene h hat mindestens	b + 1 Sätze

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

$$h \le 1 + \log_{b+1}\left(\frac{N+1}{2}\right)$$

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 *ln 2* (etwa 69%).
- Die SPAN kann aber durch gewisse Techniken noch erheblich verbessert werden (siehe unten).

2.1.1 Suchoperationen im B+-Baum

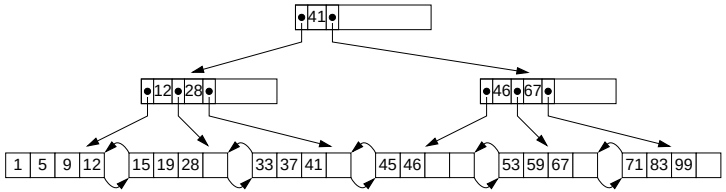
exakte Suche:

- 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(p_act);
    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 ≤ k_i.
        IF (es gibt solch ein k_i) THEN index := i-1; END;
        EMQ(p_index, 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;
```


Beispiel

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



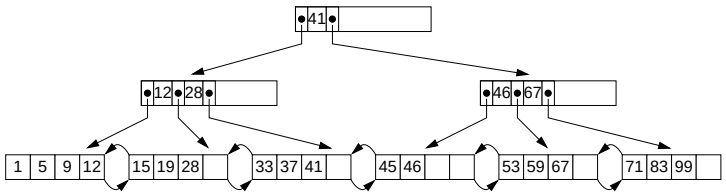
Bereichsanfrage im B^+-Baum

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

```
Algorithmus RQ(pact: Knoten; low,up: Key)
    Bestimme analog zur exakten Suche das Blatt first, in dem ein Datensatz mit Schlüs-
    sel 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 ≥ up) OR
             (pact enthält den größten Schlüssel der Datei) THEN
             EXIT
        END;
        pact := RightLeaf(pact); (* bestimmt rechten Nachbarknoten *)
    END;
END RQ;
```


Beispiel

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



2.1.2 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:

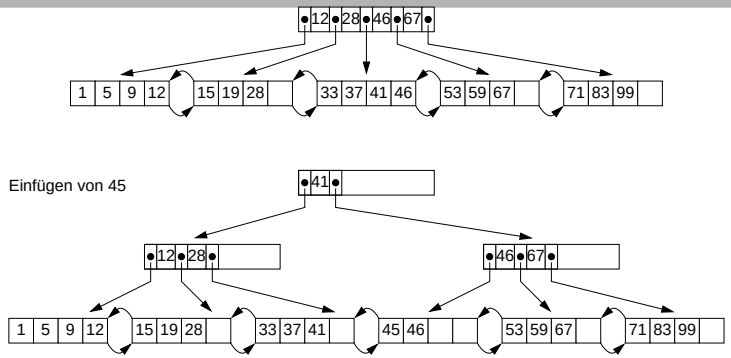
- Was passiert wenn die Seite keinen Datensatz mehr aufnehmen kann?
  1. Lösung: Einführung von Überlaufseiten und verketten mit der Primärseite.
     - Nachteil: Kosten für Suche, Einfügen und Löschen erhöhen sich.
  2. Lösung: Reorganisation der Datenstruktur
     - sofort: Überlaufseiten werden nicht 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 mit einer benachbarten Seite.

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 pnew und die in Glinks in pact;
    Sei kmax der größte Schlüssel in Glinks.
    Füge das Paar (kmax, pneu) in den Vaterknoten ein;
    IF (Vaterknoten ist übergelaufen) THEN …
END Insert;

Beispiel



wichtige Eigenschaften:

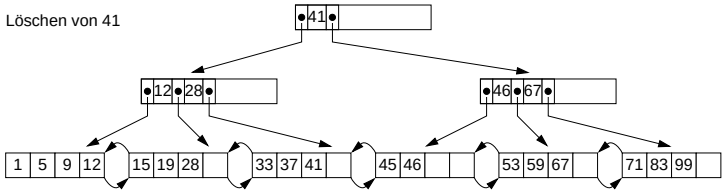
- Einfügeoperation bleibt auf einen Pfad des B^+-Baums beschränkt.
- beim Einfügen bleiben die Eigenschaften eines B^+-Baums erhalten.

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?



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 Geschwisterknoten enthält mehr als b Datensätze) THEN
    teile die Datensätze, die sich in pact und in dem Geschwisterknoten befinden, gleichmäßig zwischen diesen Knoten auf;
    modifiziere den dazugehörigen Trennschlüssel im Elternknoten;
    RETURN;
  ELSE
    Speichere die Datensätze aus pact in einem Geschwisterknoten;
    Entferne den Eintrag aus dem Elternknoten, der auf pact verweist;
    IF (Elternkoten enthält zu wenig Einträge) THEN …

wichtige Eigenschaft:

- Löschen ist auf einen Pfad beschränkt

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: $O(\log_b N)$

  Bereichanfrage: $O(\log_b N + r/b)$

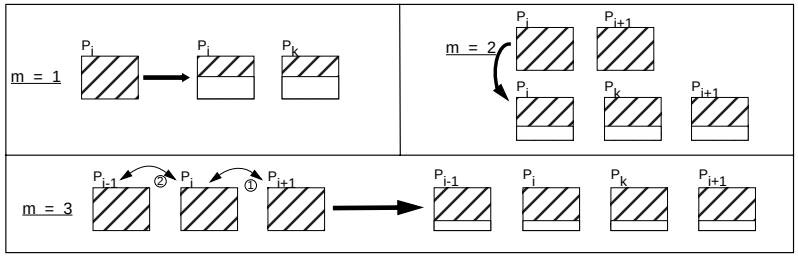
  Einfügen: $O(\log_b N)$

  Löschen: $O(\log_b N)$

- wieviel Datensätze können in einem B^+-Baum der Höhe 3 gespeichert werden?
  Beispiel (b = 200, 4 KB pro Seite);
  - im schlechtesten Fall: $400*200*200 = 16*10^6$ Datensätze, $8*10^4$ Datenseiten = 320 MB Speicherplatz für die Blattebene des B^+-Baums.
  - im Durchschnitt: Da Knoten zu etwa 2/3 im Durchschnitt gefüllt sind, können voraussichtlich $400*270*270 = 29*10^6$ Datensätze verwaltet werden. Es wird nun 430 MB an Speicherplatz für die Blattebene benötigt.
- in vielen Anwendungen:
  Wurzel im HSP ==> 2 Plattenzugriffe für exakte Suche

2.1.3 Verbesserung der SPAN von B^+-Bäumen

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



- erhöht aber die Einfügekosten erheblich: nur für m ≤ 3 sinnvoll

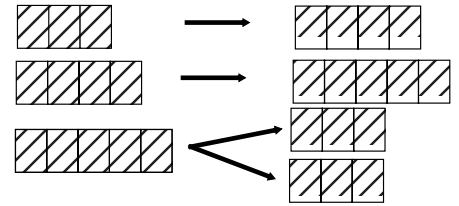
SPAN	m = 1	m = 2	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	ln 2 (69 %)		$m \cdot ln\left(\frac{m+1}{m}\right)$

Elastische Seiten

Idee:

- 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, $r \le j < 2r-1$, so wird eine Seite mit j+1 Sektoren allokiert 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 Anwachsen einer Seite von ihrer kleinsten Größe bis zum Zeitpunkt des Splits nennt man eine **vollständige Erweiterung**.

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 (elastische Seiten)	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 ist aber i.a. etwas komplizierter bei elastischen Seiten und kann zu einer leichten Verschlechterung der oben angegebenen SPAN führen.

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 Höhe des B+-Baums (= niedrige Kosten für exakte Suche)
- 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 Separatoreneigenschaften genauso gut erfüllen.
  - Kürzere Separatoren erhöhen den Verzweigungsgrad.

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” ≤ 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 Schlüsseln. 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”

Präfix-Suffix-Komprimierung

- Fortlaufende Kompression der Schlüssel einer Seite, so daß nur der Teil des Schlüssels
  - vom Zeichen, in dem er sich vom Vorgänger unterscheidet (Position V),
  - bis zum Zeichen, in dem er sich vom Nachfolger unterscheidet (Position N)

  zu übernehemn ist.
- Eintrag enthält
  - Anzahl F = V-1 der Zeichen des Schlüssels, die mit Vorgänger übereinstimmen (erster Eintrag: F=0)
  - Länge L = max(N-V,0) + 1 des komprimierten Schlüssels
  - dazugehörige Zeichenfolge
- Suchalgorithmus
  - ist stack-orientiert
  - Schlüssel können nicht vollständig, sondern nur nioch bis zur Eindeutigkeitslänge rekonstruiert werden.
- Verfahren nur anwendbar, wenn die Schlüssel nur eine Wegweiserfunktion haben.

Beispiel: Oldies, but Goldies

Schlüssel	V	N	F	L	Komprimat
City_of_New_Orleans	1	6	0	6	City_o
City_to_City	6	2	5	1	t
Closet_Chronicles	2	2	1	1	l
Cocaine	2	3	1	2	oc
Cold_as_Ice	3	6	2	4	ld_a
Cold_Wind_to_Walhalla	6	4	5	1	w
Colorado	4	5	3	2	or
Colours	5	3	4	1	u
Come_Inside	3	12	2	10	me_Inside#
Come_Inside_of_my_Guitar	12	6	11	1	_
Come_on_over	6	6	5	1	o
Come_together	6	1	5	1	t

2.1.4 Effiziente Unterstützung von Bereichsanfragen

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

Entscheidend ist das Maß für die Effizienz:

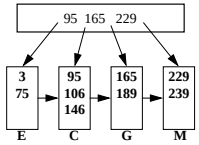
1. Ist das Maß für die Effizienz “nur” die Anzahl der Zugriffe, so ist der B^+-Baum bereits bzgl. der Anzahl der Datenseitenzugriffe nahezu optimal.
   – es wird hier implizit unterstellt, daß Zugriffe gleich teuer sind.

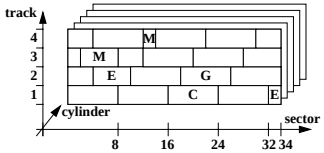
   Antwort auf Ausgangsfrage: **NEIN**
2. Wird nun berücksichtigt, daß die Kosten für einen Seitenzugriff sich aus den Komponenten Suchzeit, Rotationsverzögerung, Transferzeit und Kopfumschaltzeit zusammensetzen, so ist der B+-Baum i.a. sehr ineffizient.
   – Transferzeit ist konstant; Suchzeit und Rotationsverzögerung hängen von der vorhergehenden Position des Plattenarms ab.
   – Optimierungsziel: Reduziere möglichst Suchzeit u. Rotationsverzögerung.

   Antwort: **JA**
3. Wie sieht der Sachverhalt aus, wenn mehrere Disks verfügbar sind?

Beispiel

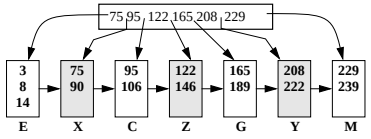
- Voraussetzungen:
  – Platte besteht aus 5 Zylindern, ein Zylinder aus 4 Spuren und eine Spur aus 34 Sektoren. Eine Seite setzt sich aus 8 Sektoren zusammen.
  – Kopfumschaltzeit = 4 Sektoren
  – Zylinder kann maximal 4 Seiten pro Datei aufnehmen.
  – Kapazität einer Datenseite ist 3 und einer Indexseite 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:

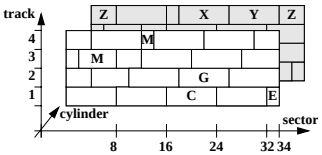




- für eine Bereichsanfrage ist die Suchzeit pro benötigter Seite niedriger als erwartet.

- 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.

Zusätzliche 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.

Große Datenseiten

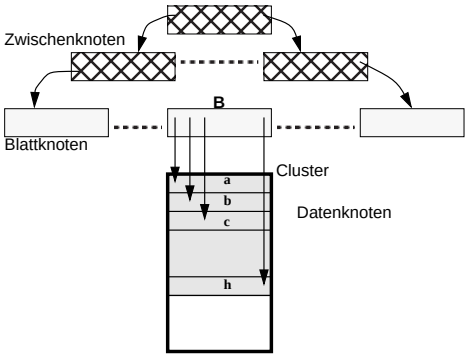
- Durch eine Verdopplung der Seitengröße ergibt sich folgendes Verhalten:
  - Bei einer Bereichsanfrage wird die Anzahl der Zugriffe auf die Datenseiten nahezu halbiert.
  - Der Aufwand für eine exakte Suche, Einfügen und Löschen erhöht sich um die zusätzlichen Transferkosten.
  - die Anzahl der benötigten 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.

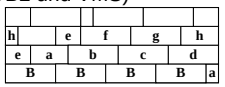
Problem:

- 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.

VSAM

- Implementierung von B^+-Bäumen bei der Firma IBM (DB2 und VMS)





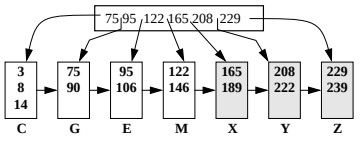
- Abbildung eines Blattknotens und der Datenknoten eines Clusters auf einen Zylinder.
- ein Cluster mit Datenknoten wird entsprechend behandelt wie eine Seite mit Datensätzen.
- Nachteil: Ausnutzung der Cluster liegt nur bei 69% und somit die SPAN von VSAM nur bei etwa 50% ($\approx$ $(\ln 2)^2$).

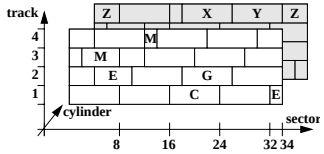
CB^+-Baum

- ähnliches Konzept wie VSAM, aber mit folgenden wesentlichen Unterschieden:
  - ein Cluster ist nicht notwendigerweise physisch zusammenhängend auf der Platte.
  - ein Cluster belegt keine Seiten, die nicht referenziert sind.
  - ein Cluster kann bis zu einer maximalen Größe um weitere Seiten erweitert werden.
  - Referenzen auf Seiten in einem Cluster liegen alle benachbart in einem Zwischenknoten. Ein Zwischenknoten besitzt i.a. Referenzen auf mehrere Cluster.
  - das Aufteilen der Seiten eines Clusters auf zwei Cluster kann in mehreren Schritten ablaufen.
  - Bereichsanfragen werden effizient bearbeitet, indem mehrere nicht notwendigerweise zusammenhängende Seiten mit einer Anforderung eingelesen werden

Beispiel

- Annahme: ein Cluster liegt auf einem Zylinder.





Bereichsanfrage: 100 < K < 240

- werden **nicht** durch sequentielles Durchlaufen der Datenebene beantwortet, sondern …
- relevante Seiten: E, M, X, Y, Z
  - 1.-ter Zugriff: E und M
    Kosten (Ann.: Plattenarm ist auf Sektor 0): 44 Sektoren
  - 2.-ter Zugriff: X, Y, Z
    Kosten (Ann.: Plattenarm ist auf Sektor 0): 38 Sektoren

Vergleich B^+-Baum mit CB^+-Baum

- Belegung in den Blattknoten ist etwas niedriger im CB^+-Baum als im B^+-Baum
  - Grund: Separator muß am Rand eines Clusters liegen.
- je nach Größe der Anfrage, Kapazität der Cluster und den Eigenschaften der Platte ist die Antwortzeit bei einer Bereichsanfrage bei einem CB^+-Baum erheblich niedriger (typischerweise liegt dies bei einem Faktor 4).
- CB^+-Baum kann nicht auf einem “einfachen” Dateisystem implementiert werden:
  - Datei muß sowohl um Seiten als auch um Cluster erweiterbar sein.
  - Größe der Cluster muß sich dynamisch ändern.

Zusammenfassung

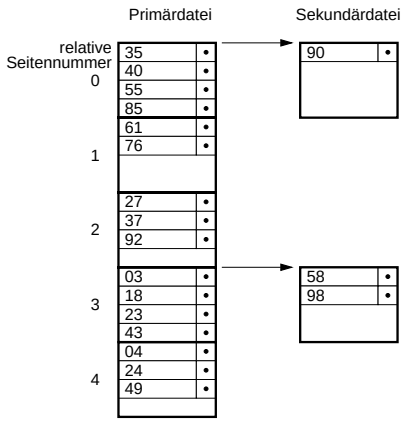
- B^+-Bäume sind **die** Zugriffsstruktur in heutigen DBS
- B^+-Bäume bieten folgende Eigenschaften:
  - SPAN liegt stets über 50% und im Durchschnitt sogar bei 69%
  - die Höhe des B^+-Baums ist $O(\log_b N)$, b = Seitenkapazität, N = #Datensätze.
  - exakte Suche, Einfügen und Löschen ist auf einen 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).

2.2 Statische Hash-Verfahren

- direkte Berechnung der Speicheradresse (Seitenadresse) eines Satzes über Schlüssel (Schlüsseltransformation)
- Hash-Funktion h: S $\rightarrow$ {1, 2,..., n}
  S = Schlüsselraum, n = Größe des **statischen** Hash-Bereiches in Seiten (***Buckets***)
- Idealfall: h ist injektiv (keine Kollisionen)
  - nur in Ausnahmefällen möglich (’dichte’ Schlüsselmenge)
  - jeder Satz kann mit 1 Seitenzugriff referenziert werden
- Statische Hash-Bereiche mit Kollisionsbehandlung
  - vorhandene Schlüsselmenge K (K $\subseteq$ S) soll möglichst gleichmäßig auf die n Buckets verteilt werden.
  - Behandlung von Synonymen:
    - Aufname im selben Bucket, 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 = #Buckets
- 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 $0 \le g_j < n$, $j \le 0$.

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 schlechteste 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.
  - globale Reorganisation der Datei

FRAGE:

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

2.3 Dynamische Hash-Verfahren

- Ziel:

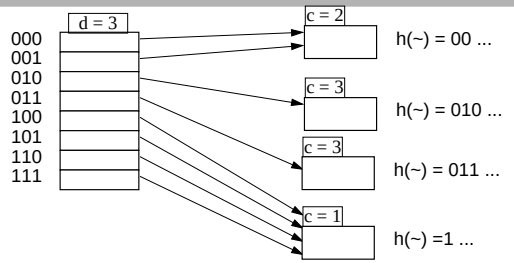
  Wenn die Leistung des Hash-Verfahren durch Einfügen bzw. Löschen von Datensätzen zu schlecht ist (SPAN zu niedrig, Ketten zu lang), soll n dynamisch angepaßt werden, d.h. ohne globale Reorganisation der Datensätze.
- Beispiel:
  - h(K) = K MOD 5 ist die bisherige Hash-Funktion.
  - wenn h(K) = K MOD 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

2.3.1 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 (incl. 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 wird auch als **globale Tiefe** des Directory bezeichnet); Eintrag verweist auf eine Seite, in dem alle zugehörigen Sätze gespeichert sind.

Beispiel



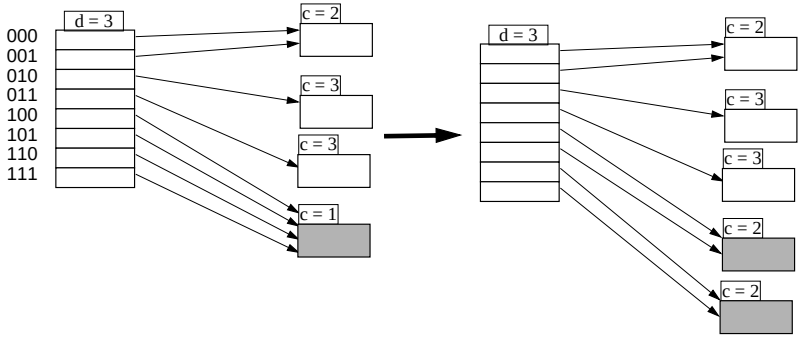
- In einer Seite können nur Datensätze gespeichert werden, die in den ersten c Bits, $c \leq d$, übereinstimmen.
  - c wird auch als **lokale Tiefe** der Seite bezeichnet.
  - es gibt genau 2^{d-c} (benachbarte) Einträge im Directory, die auf eine Seite der Tiefe c verweisen.
  - es gibt mindestens eine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

Einfügen eines Datensatzes

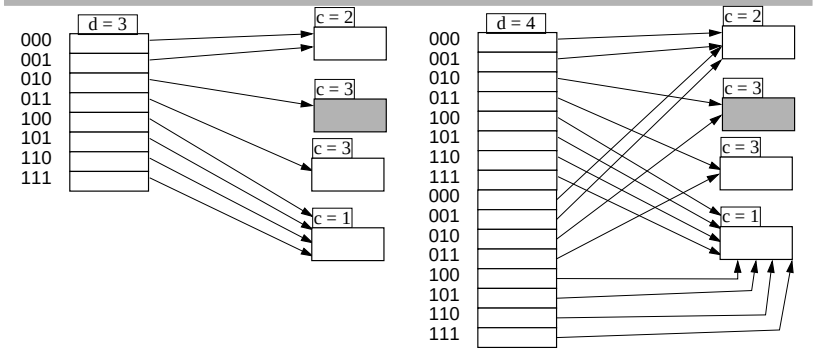
- Gegeben: Datensatz mit Schlüssel K
  1. Berechne die ersten d Bits des Pseudoschlüssels (b_0, b_1,…).
  2. Lese die Seitenreferenz S aus dem Eintrag $\sum_{j=1}^{d} b_{d-j} \cdot 2^{j-1}$ des Directory.
  3. Lese die Seite S und prüfe, ob es bereits ein Schlüssel K in der Seite gibt. Wenn ja, verlasse 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:



Verdopplung des Directory



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.

2.3.2 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 Hash-Funktionen
  - Folge von Hash-Funktionen $\{h_j(K)\}_{j\geq 0}$ mit folgenden Bedingungen:

    Bereichsbedingung:

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

    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.

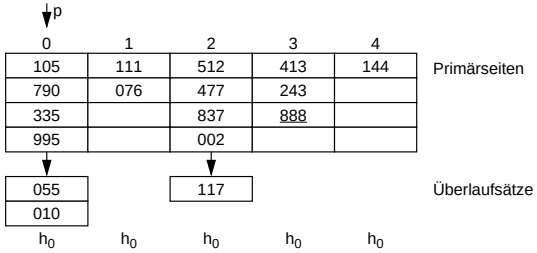
- Beispiel:

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

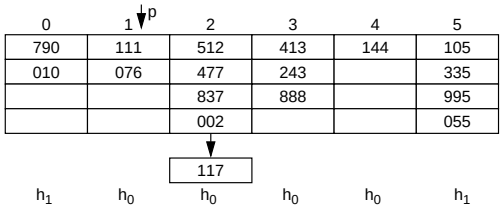
- Lineares Hashing löst ein Aufspalten einer Seit aus, wenn eine Bedingung erfüllt ist (**Kontrollfunktion**)
  - z. B.: falls der Belegungsfaktor > 80%, dann führe ein Aufteilen einer Seite aus.
  - für die Auswahl der Seite gibt es einen Zeiger p, der auf die Seite verweist, die als nächstes aufgespaltet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

Beispiel: Prinzip des LH

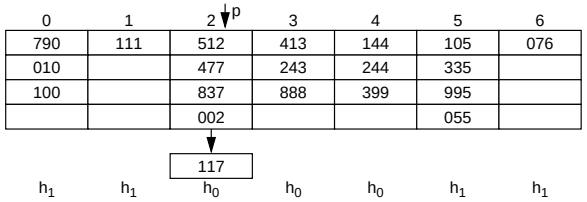
- Vorgaben:
  - n = 5, L = 0, b = 4, $h_0(K) = K \bmod 5$ und $h_1(K) = K \bmod 10$
  - Splitting, sobald $\beta > \beta_s = 0.8$



- Einfügen von Satz 888 erhöht Belegungsfaktor auf β = 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 $p + 2^L$ abgelegt.



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



Erweitern der Datei

- wird ausgelöst durch eine Kontrollfunktion, z. B. β > 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 $p + 2^L$ abgelegt.
- p wird um 1 erhöht: $p = (p+1) \bmod (n\ 2^L)$.
  Falls p = 0, so hat sich die Anzahl der Seiten in der Datei verdoppelt und L wird nun um 1 erhöht.

Split-Strategien:

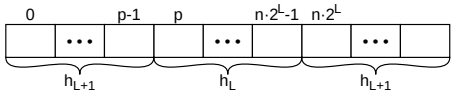
- Unkontrolliertes Splitting
  - Splitting, sobald ein Satz in den Überlaufbereich kommt
  - β ~ 0.6, schnelleres Aufsuchen
- Kontrolliertes Splitting
  - Splitting, wenn ein Satz in den Überlaufbereich kommt und $\beta > \beta_s$
  - $\beta \sim \beta_s$, längere Überlaufketten möglich

Suchen und Adressenberechnung

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

  h := $h_L(K)$;

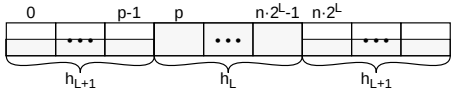
  IF (h < p) THEN h := $h_{L+1}(K)$;



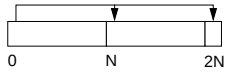
- exakte Suche:
  - Berechne die Hash-Adresse;
  - Lese die Primärseite und durchsuche die Seite, ob der Schlüssel K vorhanden ist.
  - Falls ja, STOP
  - Andernfalls, durchsuche die Kette der Sekundärseiten nach dem Satz.

2.3.3 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.



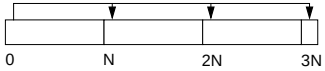
- Ziel: gleichmäßigere Belegung
  - Verdoppeln der Seitenanzahl durch eine Folge partieller Expansions-Schritte
- Idee:
  - Ausgangspunkt: Datei mit 2N Buckets, die logisch in N Paare unterteilt ist: (j, j+N) für j = 0,1,…,N-1
  - erste partielle Expansion:



2N → 3N

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

  - zweite partielle Expansion



3N → 4N

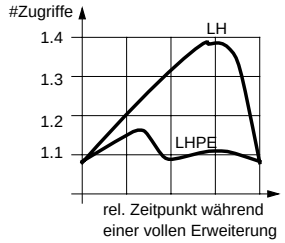
Verteilen der Sätze aus den Seiten (j, N + j, 2N + j) auf die vier Seiten (j, N + j, 2N + j, 3N + j).

  - 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.

2.3.4 Spiralen-Hashing

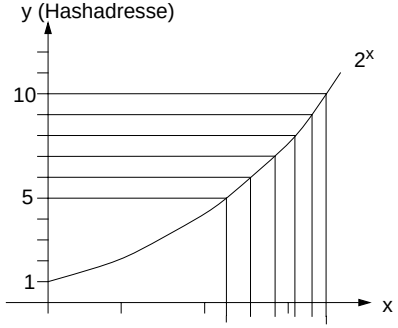
- Problem bei LH und LHPE
  - durchschnittliche Suchkosten werden durch den rel. Zeitpunkt in einer vollen Erweiterung bestimmt.
- Spiralen-Hashing vermeidet dieses Problem durch
  - ungleiche Verteilung der Daten über die Seiten
  - Hashfunktion erfüllt folgende Bedingung:
    a) Aufspalten der Seite 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

- Pramter d, d > 1, ist gegeben.
  - d wird auch der Wachstumsfaktor der Datei genannt.
- Schlüssel K wird zunächst auf ein reeles 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 + 1wird der Adreßraum ungefähr d-mal größer.



Beispiel

- Initialgröße der Datei: 5 Seiten
- Wachstumsfaktor: d = 2
- S ergibt sich aus der Gleichung $2^{S+1} - 2^S = 5$ ==> $S = \log_2 5 = 2.3219$
- 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.