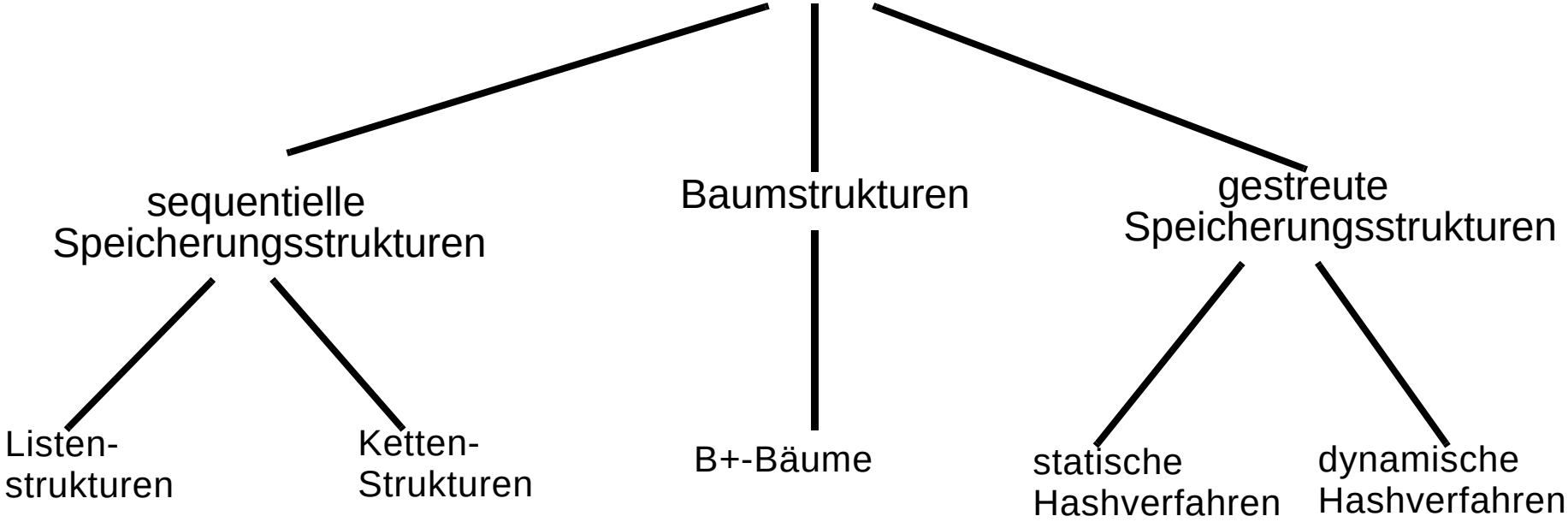
3. Eindimensionale Indexstrukturen

- 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


sequentielle Speicherungsstrukturen
Baumstrukturen
gestreute Speicherungsstrukturen
Listen-strukturen
Ketten-Strukturen
B+-Bäume
statische Hashverfahren
dynamische Hashverfahren

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:

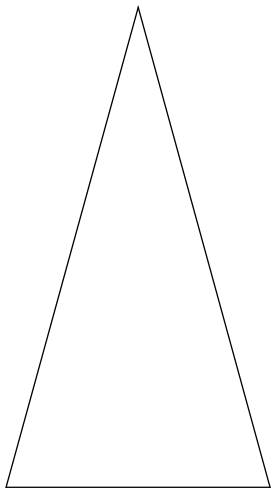
– Höhe: O(log n)

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

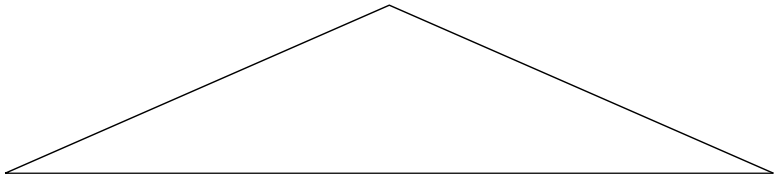
Die schlechte Nachricht!

- Binärbaumstrukturen sind für die Organisation von Daten auf dem Hintergrundspeicher nicht geeignet.
  - zu hoch (z. B. für 10^6 Datensätze 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?

3.1 B^+-Bäume

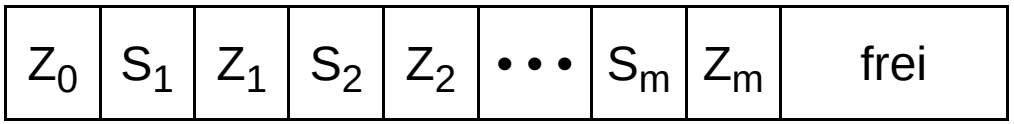
- im Gegensatz zu binären Bäumen enthält ein Knoten viele Einträge/Sätze
  - 1:1-Beziehung zwischen Knoten und Seiten
  - Daten liegen exklusiv in den Blättern
- Vorfahr:
  - ISAM (ist jedoch statisch und benötigt periodisch globale Reorganisation)
  - B-Baum (Bayer & McCreight, 1972)
- Suchfunktionen:
  - 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

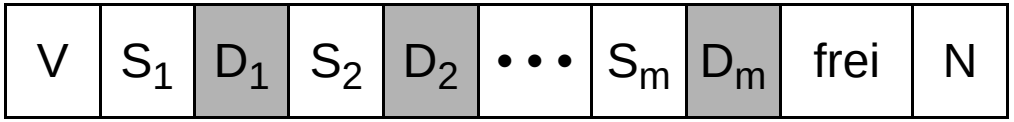
1. Jeder Weg von der Wurzel zum Blatt hat die gleiche Länge.
2. Jeder Zwischenknoten hat mindestens b+1 Söhne.
   Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.
   Jedes Blatt hat mindestens b* Einträge.
3. Jeder Zwischenknoten hat höchstens 2b+1 Söhne.
   Jedes Blatt hat höchstens 2b* Einträge.

- Zwischenknoten:



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

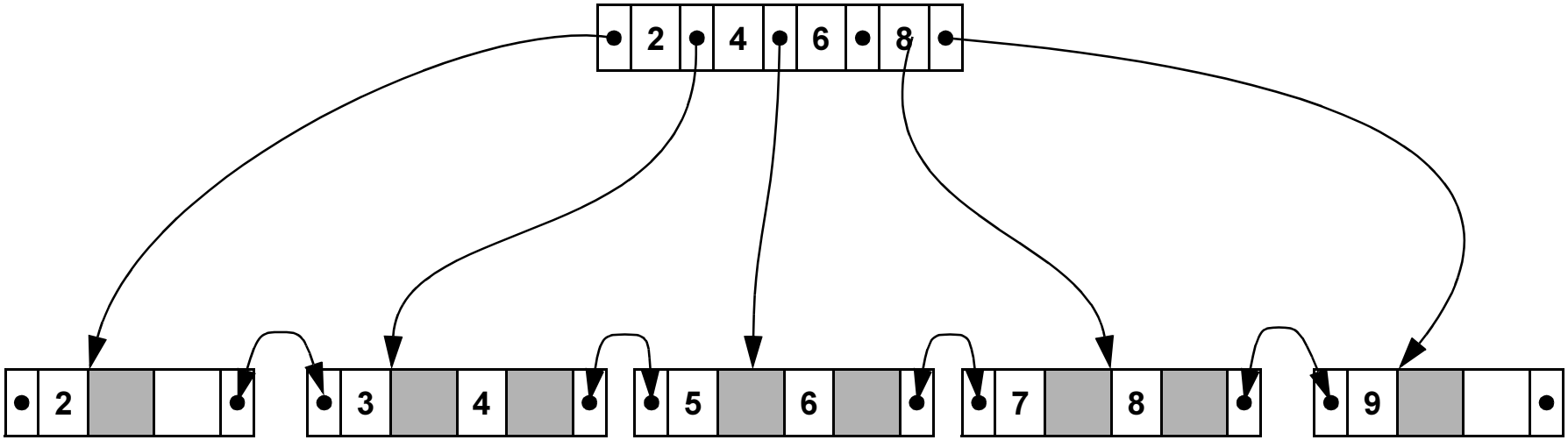
- Blattknoten:



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

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.

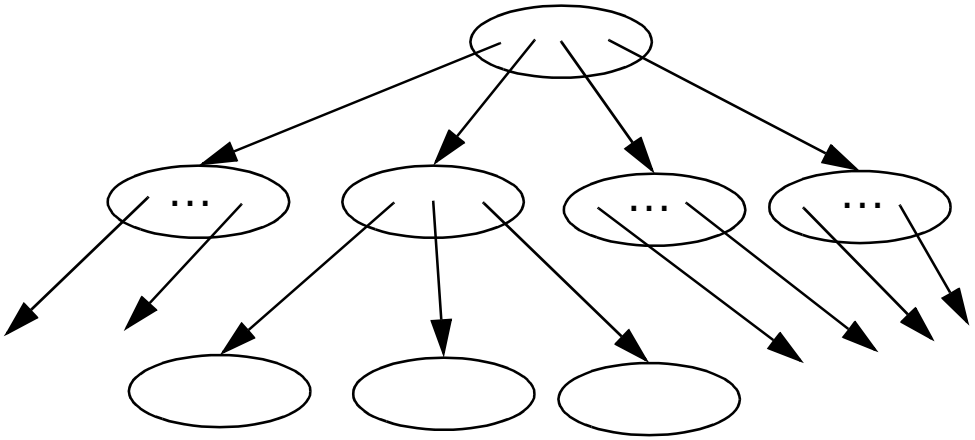
Eigenschaften des B^+-Baums

- lokale Ordnungserhaltung:

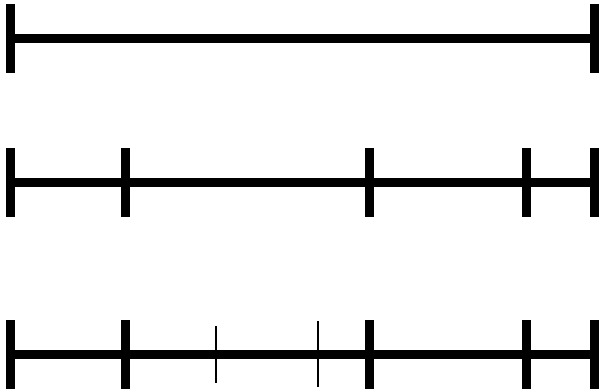
  Für jeden Zwischenknoten K mit j Schlüsseln $k_1,\ldots,k_j$ und (j+1) Söhnen $p_0,\ldots,p_j$ gilt:

  Für jedes i, $1 \leq i \leq j$, 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.

B^+-Baum



Datenraum



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 $N \geq 2*(b+1)^{h-1}$ und somit

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

Wieviel Speicher 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).

3.1.1 Suchoperationen im B+-Baum

exakte Suche:

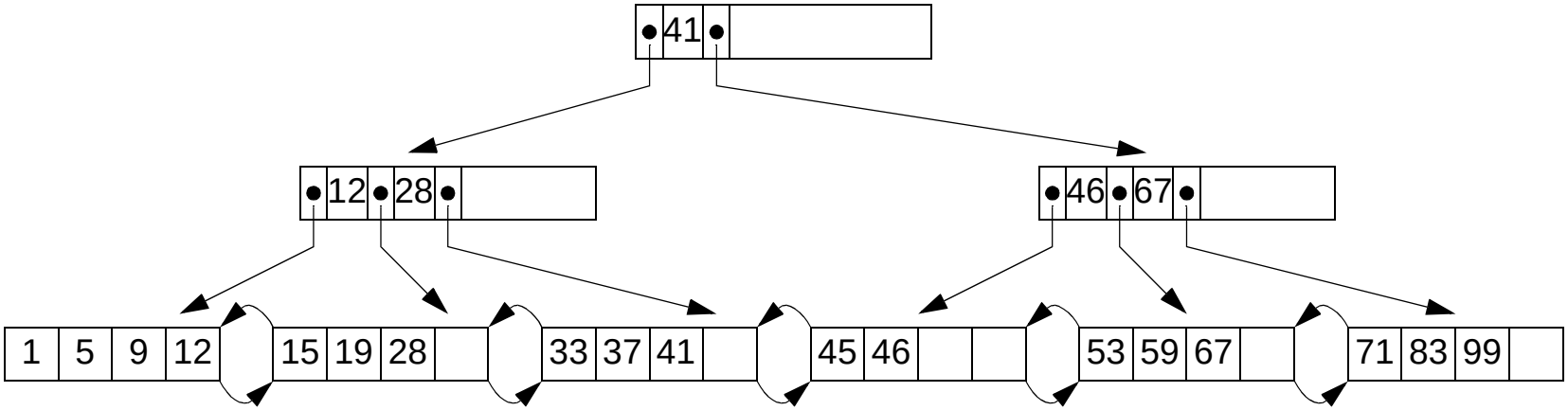
- gegeben ein Schlüssel x. Finde den Datensatz r mit r.key = x in dem B+-Baum mit Wurzel *root*.

Algorithmus EMQ(p_{akt}: Knoten, x: Key)

```
    ReadPage(p_akt);
    IF (p_akt ist ein Zwischenknoten) THEN
        index := m; (* m = Anzahl der Schlüssel im Zwischenknoten *)
        Bestimme im Knoten p_akt 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 p_akt den Datensatz mit Schlüssel x.
        IF (es gibt solch einen 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

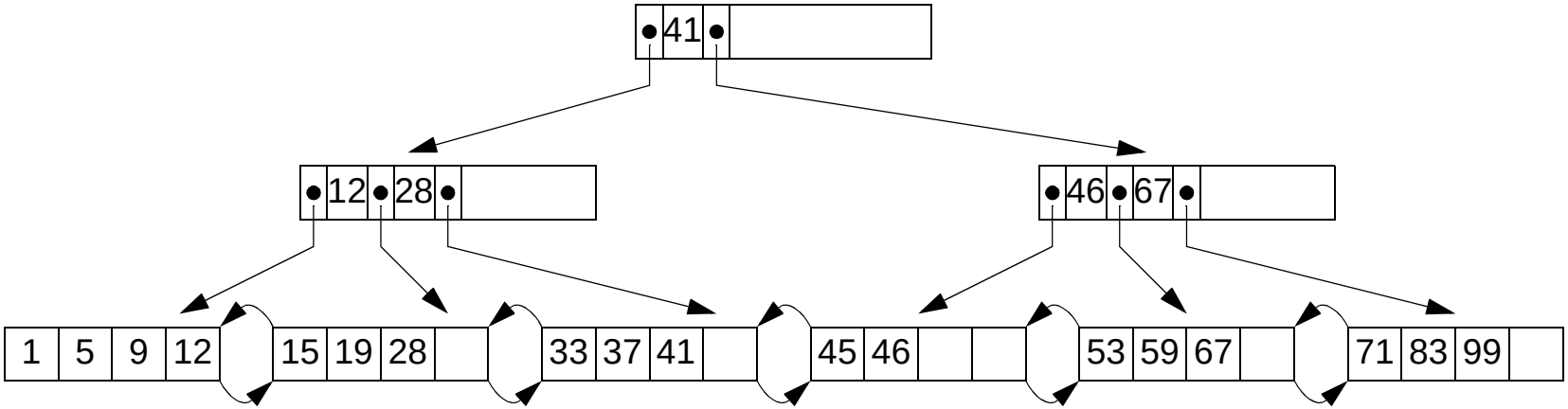
- gegeben ein Schlüsselpaar *low* und *up*, *low* $\leq$ *up*. Finde alle Datensätze r in dem B^+-Baum mit Wurzel *root* (in sortierter Reihenfolge) mit der Eigenschaft *low* $\leq$ r.key $\leq$ *up*.

Algorithmus RQ(p_{akt}: Knoten; low,up: Key)

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

Beispiel

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



3.1.2 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 und spätere 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(p_{akt}: Knoten; r: Record);

Suche nach einem Datensatz mit Schlüssel r.key; (* siehe EMQ(p_{akt}, r.key *)

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

Setze p_{akt} auf das zuletzt gelesene Blatt;

Füge r in p_{akt} ein;

IF (p_{akt} ist übergelaufen) THEN

teile die Datensätze in p_{akt} in zwei gleich große Gruppen L und R, so daß alle Datensätze in L kleiner sind als die Datensätze in R;

speichere die Datensätze in R in einem neuen Blatt p_{neu} und die in L in p_{akt};

Sei k_{max} der größte Schlüssel in L;

Füge das Paar (k_{max}, p_{neu}) in den Vaterknoten ein;

IF (Vaterknoten ist übergelaufen) THEN … // rekursiver Aufruf von Insert

END Insert;

Spezialfall: Überlauf der Wurzel

- Rekursion wird spätestens durch eine Überlaufbehandlung der Wurzel beendet.
- Der Teil des Algorithmus lautet dann:

IF (p_{akt} ist übergelaufen) THEN

…

Sei k_{max} der größte Schlüssel in L;

IF (p_{akt} ist Wurzel)

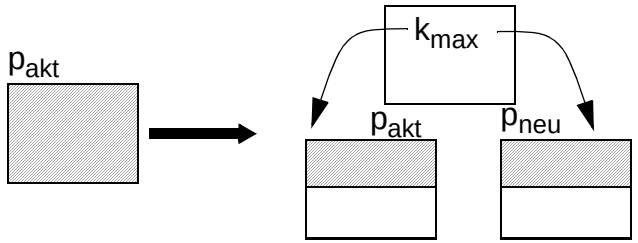
Füge das Tripel (p_{akt},k_{max},p_{neu}) in einen neuen Knoten ein und deklariere diesen zur neuen Wurzel des B+-Baums;

ELSE

Füge das Paar (k_{max}, p_{neu}) in den Vaterknoten ein;

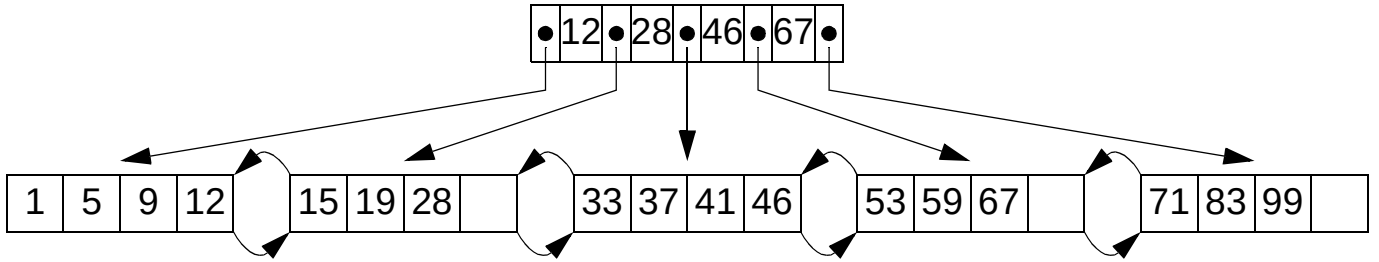
IF (Vaterknoten ist übergelaufen) THEN … // rekursiver Aufruf von Insert

END Insert;

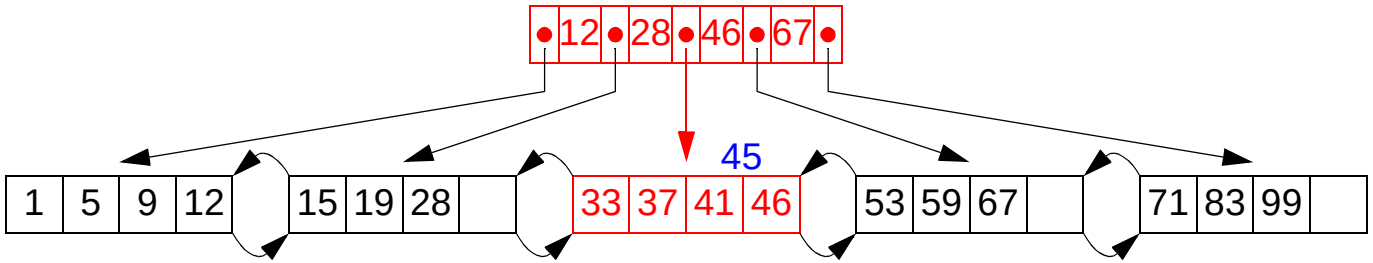


Beispiel

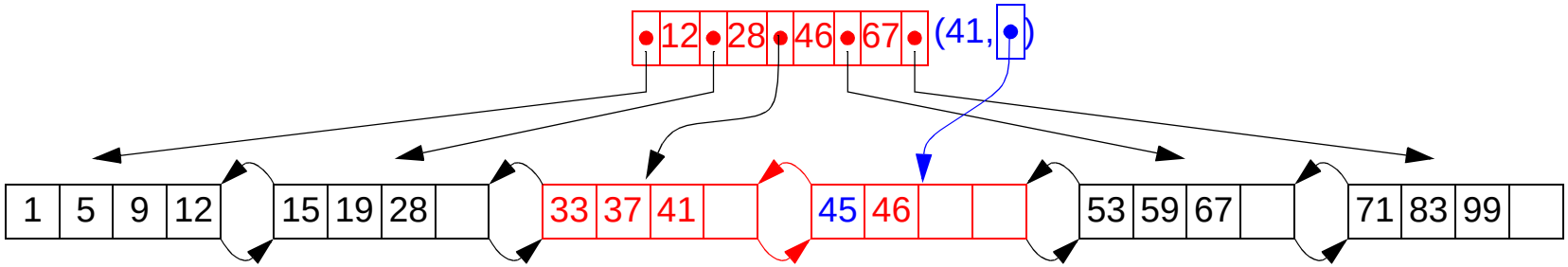
- Einfügen von 45 in folgenden B+-Baum



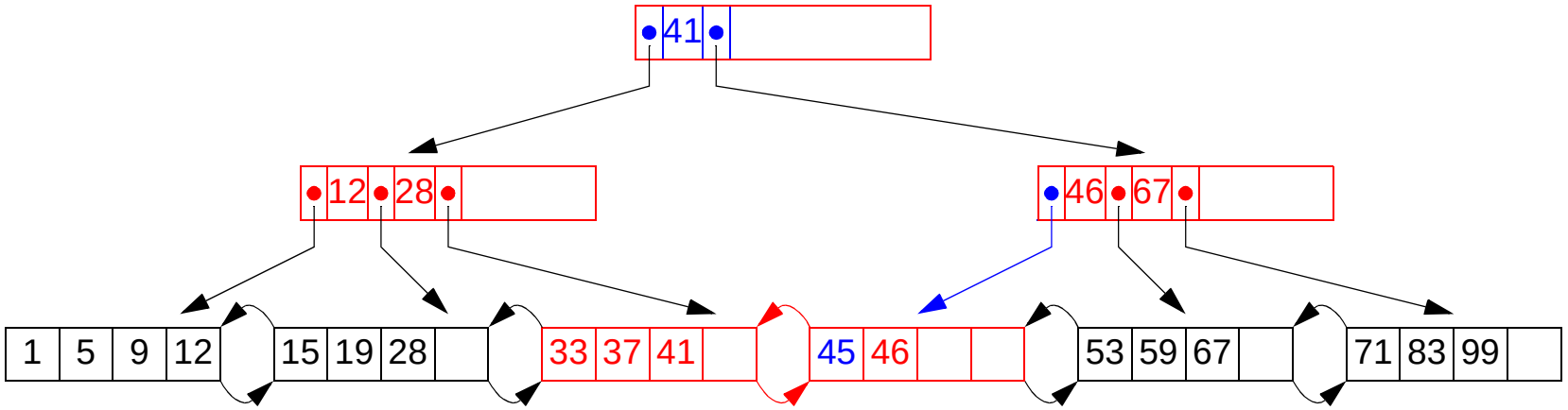
  - Suche nach dem Blatt



  - Einfügen in das Blatt, Spalten des Blatts und Einfügen in den Vater



– Einfügen in die Wurzel, Spalten der Wurzel und Erzeugen der neuen Wurzel



Wichtige Eigenschaften beim Einfügen:

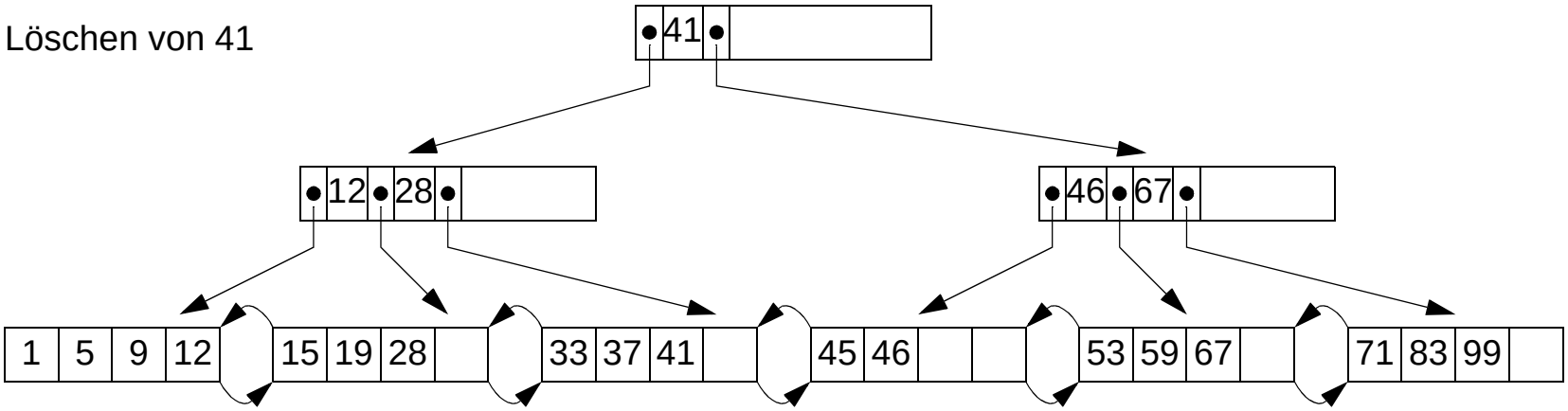
- Einfügeoperation bleibt auf einen Pfad des B^+-Baums beschränkt.
  – Pro Ebene wird höchstens ein neuer Knoten hinzugefügt.
- Beim Einfügen bleiben alle Invarianten des B^+-Baums erhalten.

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(p_{akt}, k *)
IF (Datensatz wurde nicht gefunden) THEN Print(“ERROR”); RETURN END;
Setze p_{akt} auf das zuletzt gelesene Blatt;
Entferne den Datensatz mit Schlüssel k aus p_{akt};
IF (p_{akt} enthält zu wenig Datensätze) THEN
    IF (einer der Geschwisterknoten enthält mehr als b Datensätze) THEN
        Teile die Datensätze, die sich in p_{akt} und in dem Geschwisterknoten befinden, gleichmäßig zwischen diesen Knoten auf;
        modifiziere den dazugehörigen Trennschlüssel im Elternknoten;
    ELSE
        Speichere die Datensätze aus p_{akt} in einem Geschwisterknoten;
        Entferne den Eintrag aus dem Elternknoten, der auf p_{akt} verweist;
        IF (Elternknoten ist Wurzel mit einem Eintrag)
            Entferne Wurzelknoten und deklariere p_{akt} als neue Wurzel;
        ELSE
            IF (Elternknoten enthält zu wenig Einträge) THEN
                // Rekursive Behandlung des Unterlaufs

Wichtige Eigenschaft:

- Löschen ist auf einen Pfad des B+-Baums beschränkt.
  - Pro Ebene werden maximal zwei Knoten miteinander verschmolzen.
- Alle Invarianten des B+-Baums bleiben erhalten.

Problem

- Jojo-Effekt durch kontinuierliches Einfügen und Löschen im B+-Baum
  - Eine Seite die durch das Einfügen eines Datensatzes x erzeugt wurde, wird bei einem darauffolgenden Löschen von x wieder verschmolzen.
- Beseitigung des Jojo-Effekts durch Festlegung eines anderen Mindestfüllgrads (z. B. 33%).
  - Verschmelzen einer Seite erfolgt erst dann, wenn der Füllgrad der Seite unter 33% liegt.
  - Alle anderen Invarianten und asymptotischen Aussagen bleiben erhalten.

Lemma:

Ist eine Seite durch eine Reorganisation verändert oder erzeugt worden, sind mindestens $\Theta(b)$ Einfüge- bzw. Löschoperationen notwendig, bevor die Seite wieder reorganisiert werden muß.

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: $O(\log_b N)$

- Wie viele 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

3.1.3 Bulkloading eines B+-Baums

Problem:

- Zu einer vorgegebenen Datenmenge mit N Datensätzen soll ein B+-Baum aufgebaut werden.

1. Lösung

- N-maliger Aufruf des bekannten Algorithmus zum Einfügen.
- Kosten: $O(N \log_b N)$

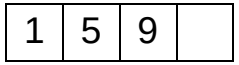
2. Lösung

- Bessere Ausnutzung des Hauptspeichers
  - M = verfügbarer Platz im Hauptspeicher (in der Anzahl der Datensätze)
- Externes Sortieren der Datenmenge
  - Aufwand: $O(N/b \log_{M/b} N/b)$
- Bottom-up Aufbau des B+-Baums (Parameter c, $b \leq c \leq 2b$: gewünschter Füllgrad)
  - Aufwand: $O(N/b)$

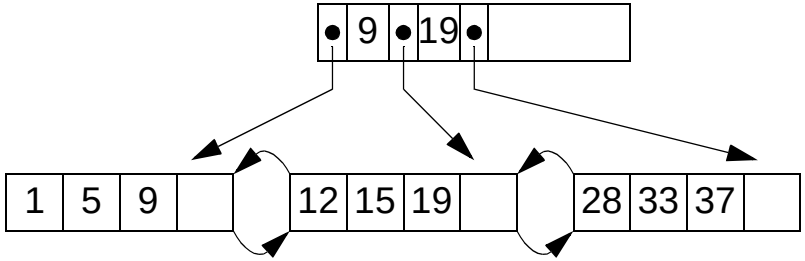
Beispiel:

- Parameter b = 2, c = 3
- Sortierte Datenfolge:
  99,83,71,67,59,53,46,45,41,37,33,28,19,15,12,9,5,1

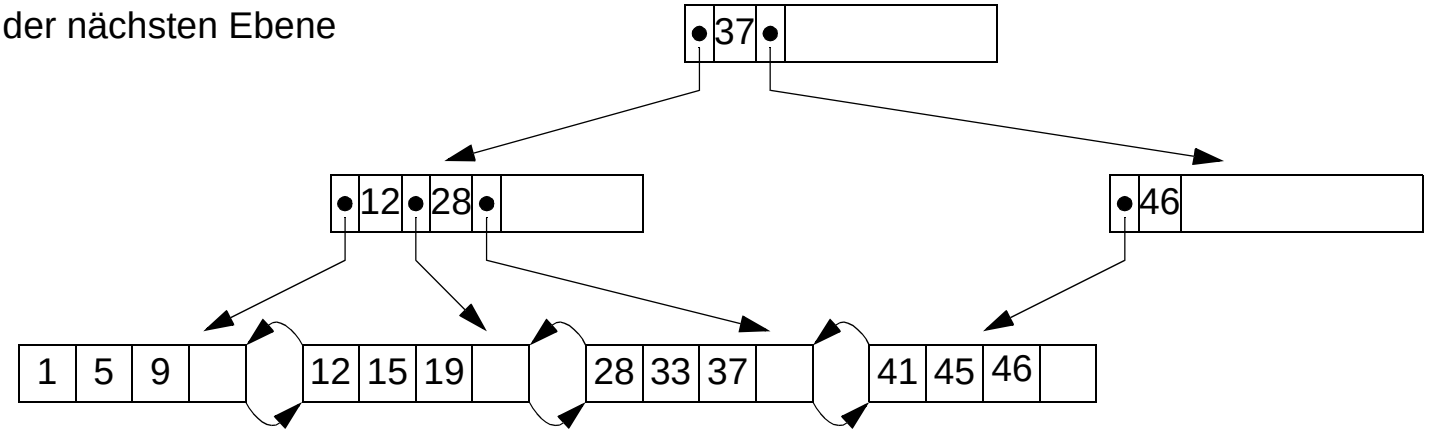
Aufbau des 1. Blatts:



Aufbau des 1. Indexknoten:



Aufbau der nächsten Ebene



Technisches Problem:

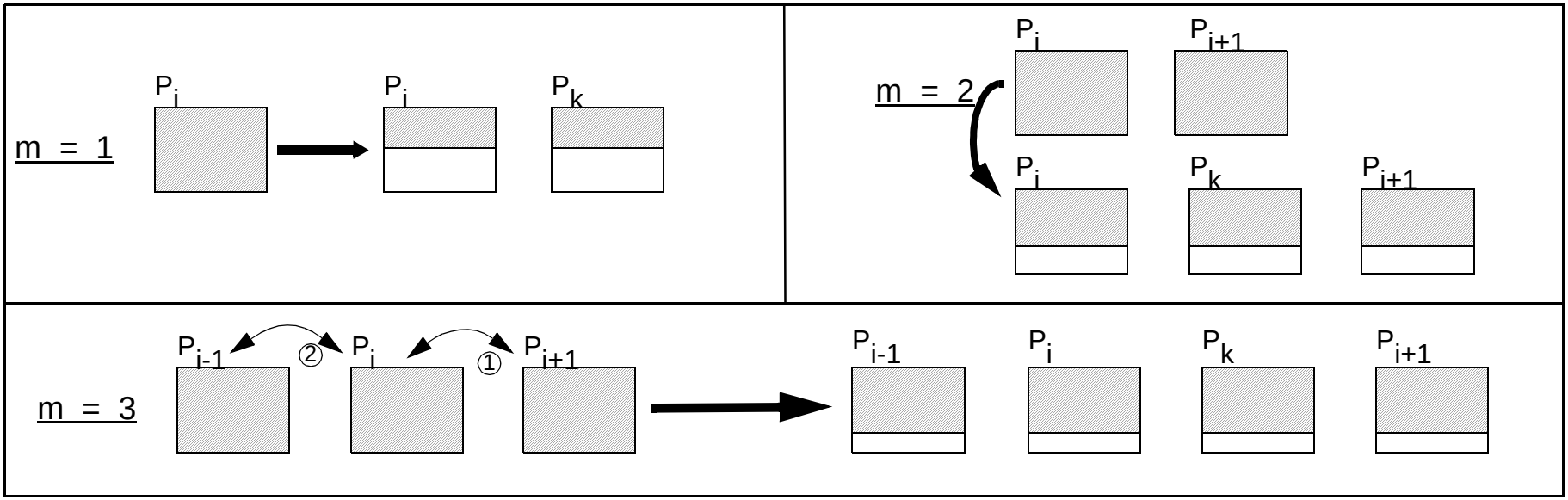
- Nach dem Aufbau können die Knoten auf dem rechten Pfad unterfüllt sein.
  - Verschmelzen mit dem linken Bruderknoten.

Bulk-Insertion

- Zu einem vorgegeben B+-Baum mit N_1 Datensätzen soll eine Datenmenge mit N_2 Datensätzen möglichst schnell eingefügt werden.
- 1. Lösung:
  Bulkloading eines neuen B+-Baums mit der kompletten Datenmenge
  - Ineffizient, wenn $N_1 >> N_2$
- 2. Lösung
  Einzelnes Einfügen der N_2 Datensätze.
  - Ineffizient, wenn N2 >> N1
- Frage: Was ist die beste Strategie für $N_1 = N_2$?

3.1.4 Verbesserung der SPAN

- 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 $m \leq 3$ sinnvoll

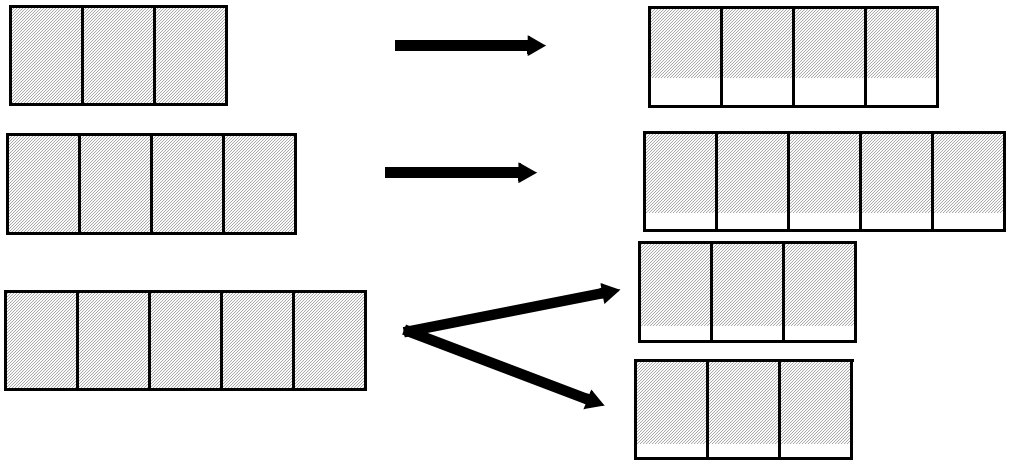
SPAN	m = 1	m = 2	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	ln 2 (69 %)		$m \cdot ln\left(\frac{m+1}{m}\right)$

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, $r \leq j < 2r-1$, so wird eine Seite mit j+1 Sektoren angefordert 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.

3.1.5 Präfix B^+-Bäume

Entwurfsziele beim B^+-Baum

- möglichst wenig Speicherplatz für die Zwischenknoten
  - ggf. paßt der gesamte Index in den Hauptspeicher
- möglichst geringe Höhe des B^+-Baums
  - niedrige Kosten für exakte Suche, Einfügen und Löschen

=> möglichst hoher Verzweigungsgrad

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. “Database”.
- Jeder Schlüssel K mit “Computing” $<$ K $\leq$ “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:

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

- Beispiel (oben): y = “D”

Präfix-Suffix-Kompression

- 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 übernehmen ist.
- Eintrag enthält
  - Anzahl **F** = V-1 der Zeichen des Schlüssels, die mit Vorgänger übereinstimmen (erster Eintrag: F=0)
  - Länge **L** = max(N-V+1,0) des komprimierten Schlüssels
  - dazugehörige Zeichenfolge
- Suchalgorithmus
  - ist stack-orientiert
  - Schlüssel können nicht vollständig, sondern nur noch bis zur Eindeutigkeitslänge rekonstruiert werden.

- Verfahren nur anwendbar, wenn die Schlüssel nur eine Wegweiserfunktion haben.

Beispiel: Oldies, but Goldies

Schlüssel	V	N	F	L	Wert
City_of_New_Orleans	1	6	0	6	City_o
City_to_City	6	2	5	0	
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	0	
Colorado	4	5	3	2	or
Colours	5	3	4	0	
Come_Inside	3	12	2	11	me_Inside_
Come_Inside_of_my_Guitar	12	6	11	0	
Come_on_over	6	6	5	1	o
Come_together	6	1	5	0	

Zusammenfassung

- Erhöhung des Verzweigungsgrads durch
  - Verwendung kurzer Separatoren
  - Kompression der Schlüssel
- Kritische Anmerkungen
  - Verwaltung variabel langer Daten innerhalb der Seiten
  - keine effektive binäre Suche innerhalb einer Seite
  - Bei einer Präfix-Suffix Kompression geht man davon aus, dass die Datensätze **nicht** im Index abgespeichert werden.

3.1.6 Effiziente Unterstützung von Bereichsanfragen

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

Entscheidend ist das Maß für die Effizienz:

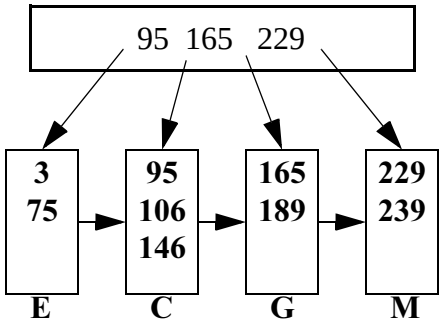
1. Ist das Maß für die Effizienz “nur” die Anzahl der Zugriffe, so ist der B^+-Baum bereits bzgl. der Anzahl der Datenseitenzugriffe nahezu optimal.
   – es wird hier implizit unterstellt, daß Zugriffe gleich teuer sind.

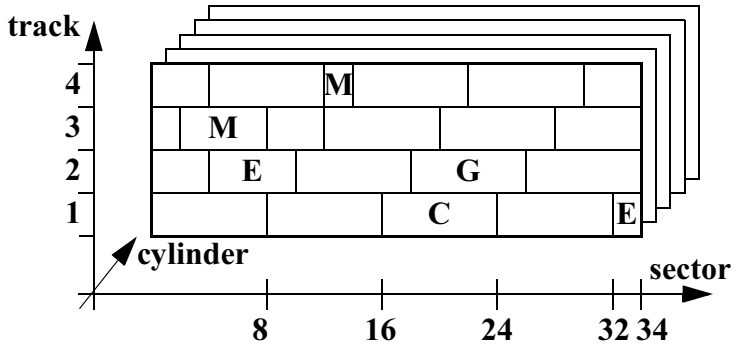
   Antwort auf Ausgangsfrage: **NEIN**
2. Wird nun berücksichtigt, daß die Kosten für einen Seitenzugriff sich aus den Komponenten Suchzeit, Rotationsverzögerung, Transferzeit und Kopfumschaltzeit zusammensetzen, so ist der B+-Baum i.a. sehr ineffizient.
   – Transferzeit ist konstant; Suchzeit und Rotationsverzögerung 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

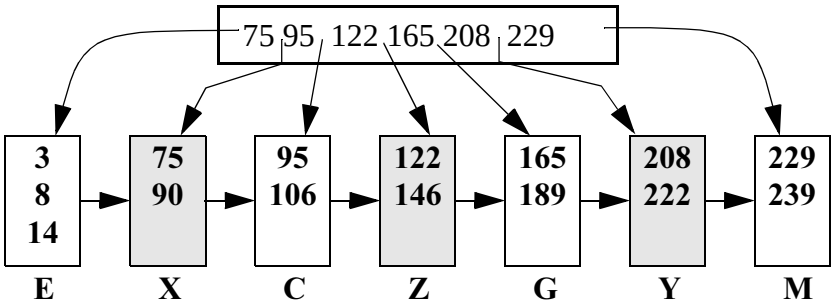
- Voraussetzungen:
  - Platte besteht aus 5 Zylindern, ein Zylinder aus 4 Spuren und eine Spur aus 34 Sektoren. Eine Seite setzt sich aus 8 Sektoren zusammen.
  - Kopfumschaltzeit = 4 Sektoren
  - Zylinder kann maximal 4 Seiten pro Datei aufnehmen.
  - Kapazität einer Datenseite ist 3 und einer Indexseite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

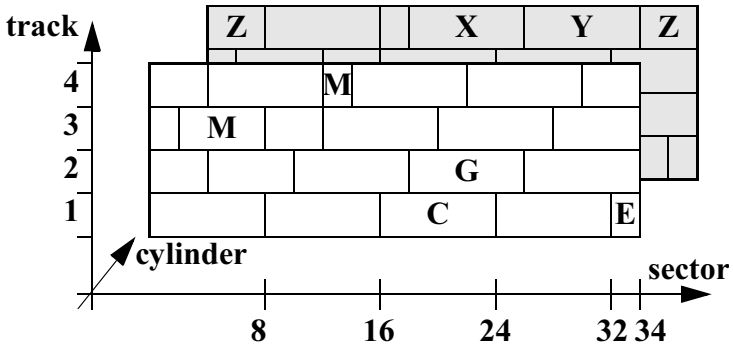




- für eine Bereichsanfrage ist die Suchzeit pro benötigter Seite niedriger als erwartet.

- 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. Lies 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

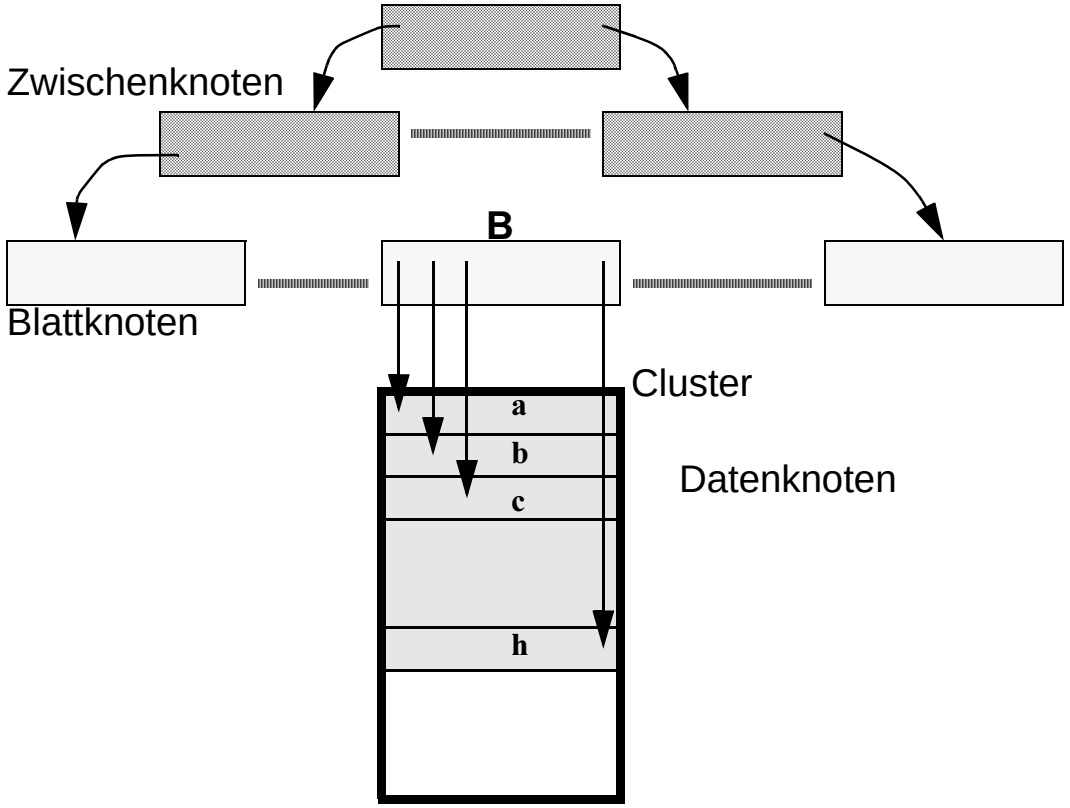
- Durch eine Verdopplung der Seitengröße ergibt sich folgendes Verhalten:
  - Bei einer Bereichsanfrage wird die Anzahl der Zugriffe auf die Datenseiten nahezu halbiert.
  - Der Aufwand für eine exakte Suche, Einfügen und Löschen erhöht sich um die zusätzlichen Transferkosten.
  - die Anzahl der benötigten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

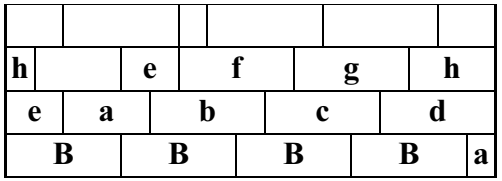
Problem:

- 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.

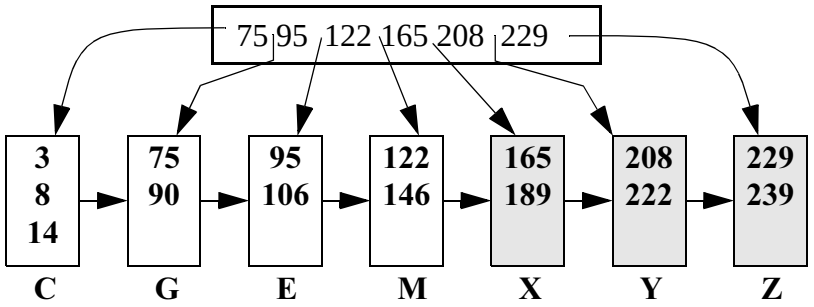
- Nachteil: Ausnutzung der Cluster liegt nur bei 69% und somit die SPAN von VSAM nur bei etwa 50% ($\approx$ $(\ln 2)^2$).

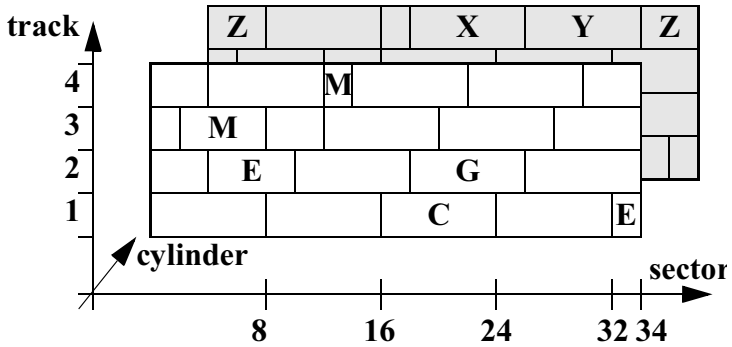
CB^+-Baum

- 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

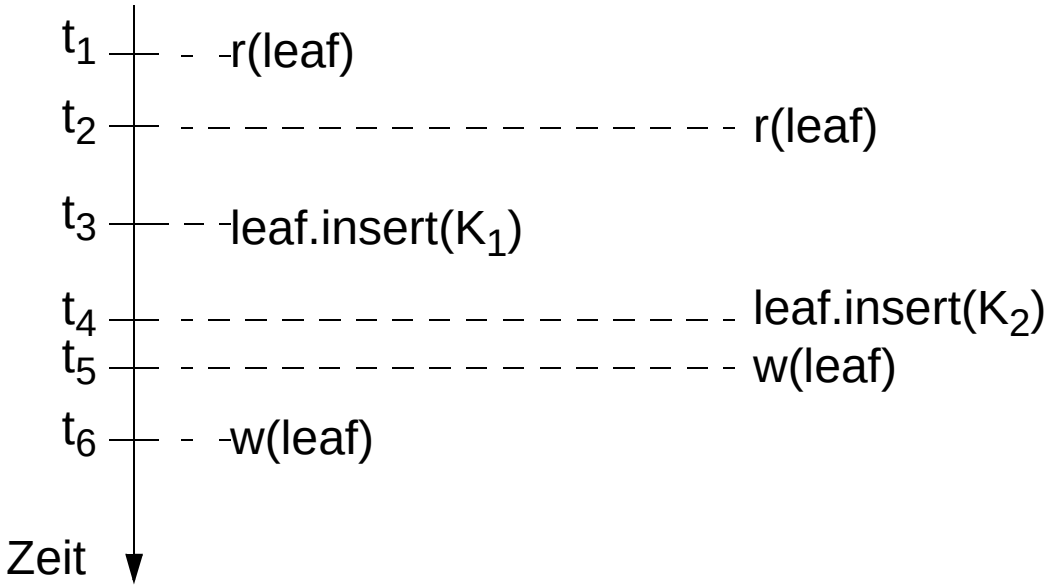
Vergleich B^+-Baum mit CB^+-Baum

- 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.

3.1.7 Synchronisation

Problem:

- Gleichzeitige Verarbeitung mehrerer Operationen im B+-Baum
  - hoher Durchsatz
  - Gewährleistung der Konsistenz der Daten und der Suchstruktur
- Problemfall (lost update)



- Spezielle Lösungen für Indexstrukturen (kein Zweiphasen-Sperrprotokoll)

Locks und Latches

Locks dienen zur Sicherstellung der Konsistenz, wobei

- im Folgenden Locks auf Knoten des B+-Baums vergeben werden,
- ein Lockmanager darüber wacht, dass keine Deadlock-Situation entsteht (und dann ggf. eine Operation zurücksetzt),
- es verschiedene Lockmodi gibt.
  - ***XLock***: exklusive Sperre einer Seite. Es sind dann keine anderen Sperren erlaubt.
  - ***SLock***: Es können mehrere SLocks gleichzeitig auf einer Seite gesetzt sein. Es darf kein XLock gleichzeitig mit einem SLock gesetzt sein.

Latches dienen ebenfalls der Konsistenzsicherung

- im Gegensatz zu Locks wird keine Deadlockerkennung vorgenommen
  - “Benutzer” muß die Deadlockfreiheit garantieren
  - weniger Aufwand beim Setzen und Freigeben eines Latchs (im Vergleich zu einem Lock)
- Implementierung erfolgt mittels primitiven Techniken z. B. Semaphore

Das Verfahren von Lehmann & Yao

- Erweiterung des B+-Baums
  - **Jeder** Knoten besitzt einen Zeiger auf seinen rechten Nachbarknoten.
  - Ein Blatt speichert zusätzlich den größten Schlüssel bei dem Zeiger auf den rechten Nachbarn ab.

- Modifikation der Suchoperation nach Datensatz mit Schlüssel K

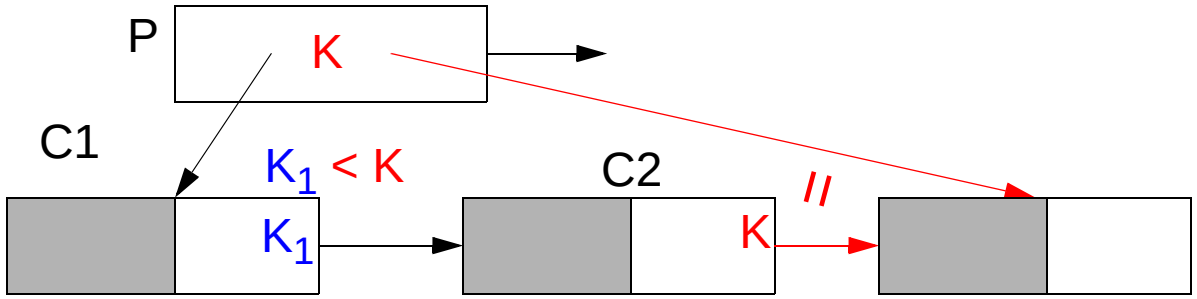
Knoten := Wurzel; *P* := NULL;
WHILE (*Knoten* nicht Blatt)
    Lies *Knoten* und setze eine Lesesperre (Latch);
    Falls P != NULL, gibt die Lesesperre von P frei;
    Solange der größte Schlüssel in *Knoten* < *K* ist
        Lies den rechten Nachbarknoten (*Rechts*) und setze eine Lesesperre;
        Zurückgeben der Lesesperre von *Knoten*;
        Knoten := *Rechts*;
    Berechne den Nachfolger *Kind*;
    P := *Knoten; Knoten* := *Kind*;

Einfügen

- Zunächst wird eine Suchoperation durchgeführt.
- Die Sperre auf dem Blatt muß dann in eine Schreibsperre umgewandelt werden.
  - ggf. muß auf lesende Operationen gewartet werden.
  - Danach wird der Datensatz eingetragen und die Schreibsperre aufgehoben.

Was passiert beim Überlauf?

- Es wird noch die Schreibsperre auf dem Knoten (C1) gehalten.
- Aufspalten von C1 in C1 und C2 und Anpassen der Rechtszeiger.



- Anfordern einer Schreibsperre auf dem Vaterknoten (P)
  - ggf. ist P zwischenzeitlich aufgespalten worden und man muß zunächst rechts die Kette durchlaufen

Löschen

- Zunächst wird eine Suchoperation durchgeführt.
- Die Sperre auf dem Blatt muß dann in eine Schreibsperre umgewandelt werden.
  - ggf. muß auf lesende Operationen gewartet werden.
  - Danach wird der Datensatz gelöscht und die Schreibsperre aufgehoben.

Was passiert beim Unterlauf?

Optimierung

Unterscheiden zwischen Datenkonsistenz und Effizienzgesichtspunkten (Salzberg & Lomet, 1996)

- Aufspalten einer Datenseite und Verschmelzen einer Datenseite muß auf der Transaktionsebene unterstützt werden.
- Interne Operationen auf einem Index können später ausgeführt werden
  - Einfügen eines neuen Indexeintrags kann durch eine Operation durchgeführt werden, die nicht innerhalb der Benutzertransaktion läuft.
  - Eintrag der Operationen (zur Verbesserung der Effizienz) in einen Scheduler

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, …
- Mehrbenutzerbetrieb auf einem B^+-Baum