11. Indexstrukturen

Vorlesung "Informationssysteme"

Sommersemester 2017

Technische Universität Kaiserslautern

Gliederung

- Motivation
- ISAM
- B-Bäume und B^+-Bäume
- Hashverfahren
- Mehrdimensionale Indexe

Überlegungen zu Performance

- Beobachtung
  - Je schneller die Anfrage berechnet wird, desto besser.
  - Daten passen nicht in den Hauptspeicher.
  - Es liegen Größenordnungen zwischen Performanz des Hauptspeichers und der Festplatte.
  - Zugriffe sind unglaublich teuer!
  - Insbesondere die wahlfreien (random access).
- Was muss getan werden?
  - Effiziente Speicherung auf Festplatte
  - Wie findet man die gesuchten Daten möglichst schnell?
  - Welche Garantien können bzgl. "Laufzeit" gegeben werden?

Grundidee eines Index

- Abbildung: Schlüssel → Menge von Einträgen
- Beispiele:
  - MatrikelNummer → persönliche Daten des Studenten
  - PLZ → Name und andere Informationen einer Stadt
  - Term → alle Dokumente in denen dieser Term enthalten ist
- Natürlich möchte man bei diesen häufig auftretenden Anfragen die Antwortzeit gering halten.
- Dafür wird ein Index angelegt: Ein Index materialisiert diese Abbildung!

Was ist mit der binären Suche?

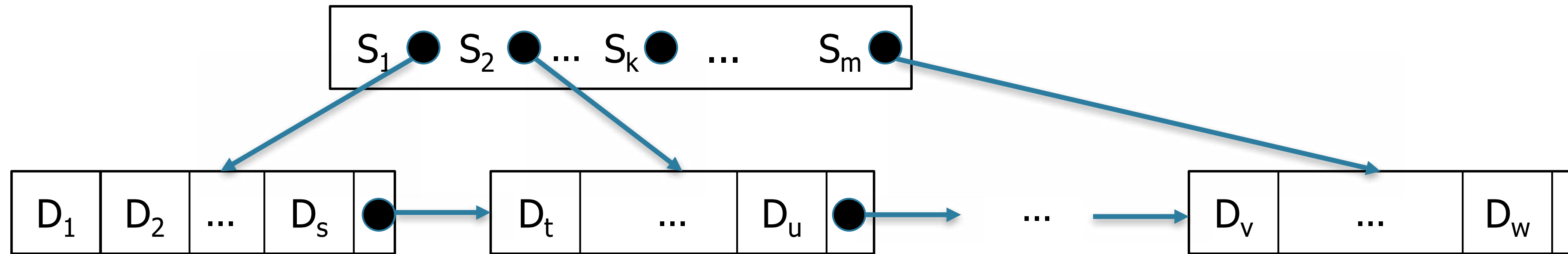
- Idee: Sortiertes Array, dann binär suchen
  - $O(n \log n)$ Kosten für das Sortieren
  - was passiert, wenn ein neuer Datensatz eingefügt werden muss?
- Binärer Suchbaum, am besten balanciert (AVL oder rot-schwarz Baum)

Aber....

- binäre Suchbäume sind für DBMS ungeeignet
  - zu hoher Speicherverbrauch
  - schwer abzubilden auf Externspeicher
  - zu viele Cache-Misses (warum?)

➔ viel zu langsam!

Index-Sequential Access Method (ISAM)



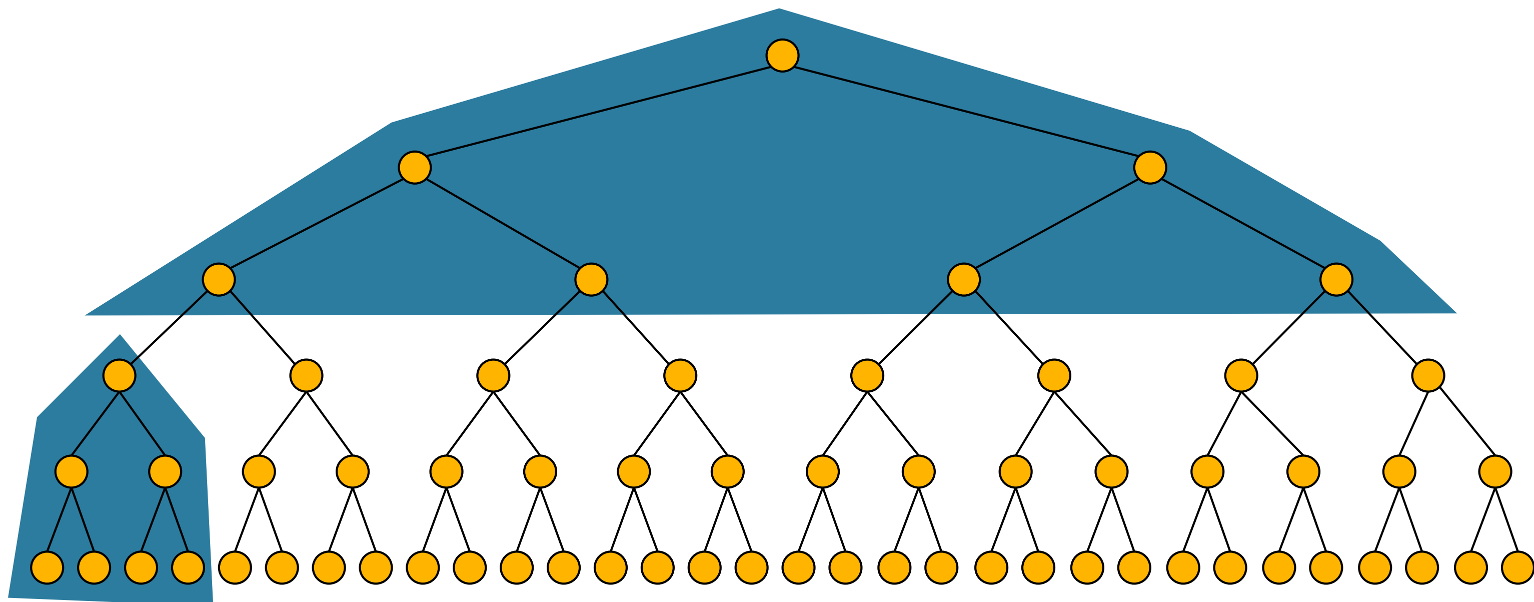
Besteht aus Schlüsseln S_j und Datensätzen D_i.

- Sowohl Schlüssel als auch Daten werden geordnet abgespeichert.
- Suche:
  - Binäre Suche in Schlüsseln zur gewünschten Position
  - Sequentielles Lesen in Datenseiten
- Einfügen:
  - Auffinden der Einfügeposition (wie bei Suche)
  - Was passiert wenn die Seite, in die eingefügt werden soll, voll ist? Nicht gut ...
- Löschen:
  - Auffinden der Löschposition (wie bei Suche und Einfügen)
  - Löschen des Datensatzes. Was passiert wenn die Seite leer wird?

Mehrwegbäume - Motivation

- Beobachtung
  - Binäre Bäume nicht optimal für Festplatten
  - Wichtig: Anpassung der Kapazität der Knoten an Größe der Seiten.
  - Warum?
- Fanout
  - Je breiter der Baum desto flacher (d.h. weniger tief).
  - Je flacher desto weniger "Sprünge" zwischen Knoten

  ➔ Weniger Zugriffe auf Seiten auf der Festplatte.



Mehrwegbäume (2)

- Vorfahr (1965): ISAM (statisch, periodische Reorganisation)
- Weiterentwicklung: B- und B^+-Baum
  - B-Baum: 1970 von R. Bayer und E. McCreight entwickelt
  - treffende Charakterisierung: "The Ubiquitous B-Tree" *
- Grundoperationen:
  - Einfügen und Löschen eines Satzes
  - direkter Schlüsselzugriff auf einen Satz
  - sortiert sequentieller Zugriff auf Satzbereiche

  ➔ dynamische Reorganisation durch Splitting und Mischen von Seiten
- Balancierte Struktur, unabhängig von Schlüsselmenge und Einfügereihenfolge
- Ausführliche Betrachtung von Mehrwegbäumen (von Prof. Härder):
  http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/courses/ss2007/
  Informationssysteme/addons/mehrwegbaeume.full.pdf

(*) D. Comer: *The Ubiquitous B-Tree*. ACM Computing Surveys 11(2), 1979

B-Bäume

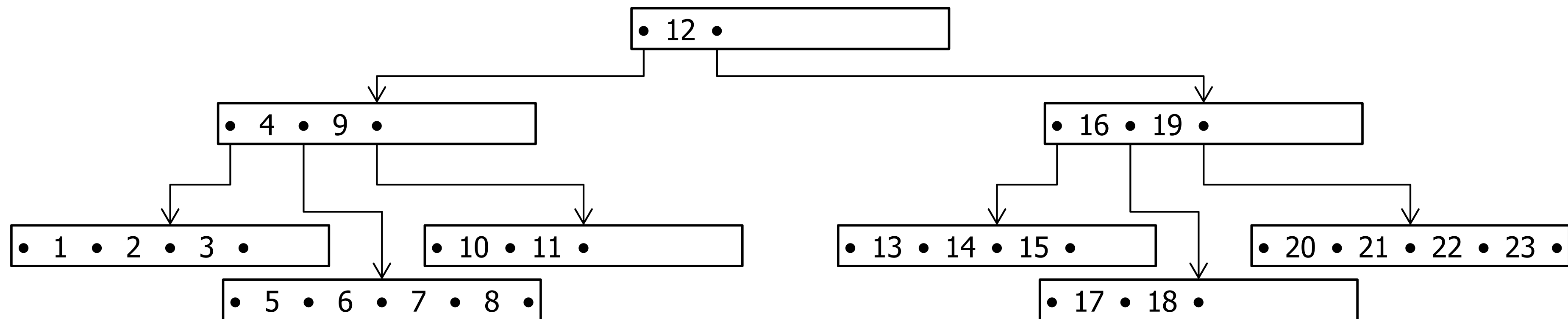
- Definition:

  Sei k eine ganze Zahl, k > 0. Ein **B-Baum** B mit Grad k ist entweder ein leerer Baum oder ein geordneter Suchbaum mit folgenden Eigenschaften:

  1. Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge.
  2. Jeder Knoten mit der Ausnahme der Wurzel hat mindestens k und höchstens 2k Einträge. Die Wurzel hat zwischen einem und 2k Einträgen. Die Einträge werden in den Knoten sortiert gehalten.
  3. Jeder Knoten mit n Einträgen, außer den Blättern, hat n+1 Kinder.
  4. Seien $S_1, \ldots, S_n$ die Schlüssel eines Knotens mit n+1 Kindern. $P_0, \ldots P_n$ seien die Verweise auf die Kinder. Dann gilt:
     a) P_0 verweist auf den Teilbaum mit Schlüsseln kleiner als S_1.
     b) P_i $(0 < i < n)$ weist auf den Teilbaum mit Schlüsseln zwischen S_i und S_{i+1}.
     c) P_n verweist auf den Teilbaum mit Schlüsseln größer als S_n.
     d) In den Blattknoten sind die Verweise nicht definiert.

B-Bäume – Beispiel und Eigenschaften

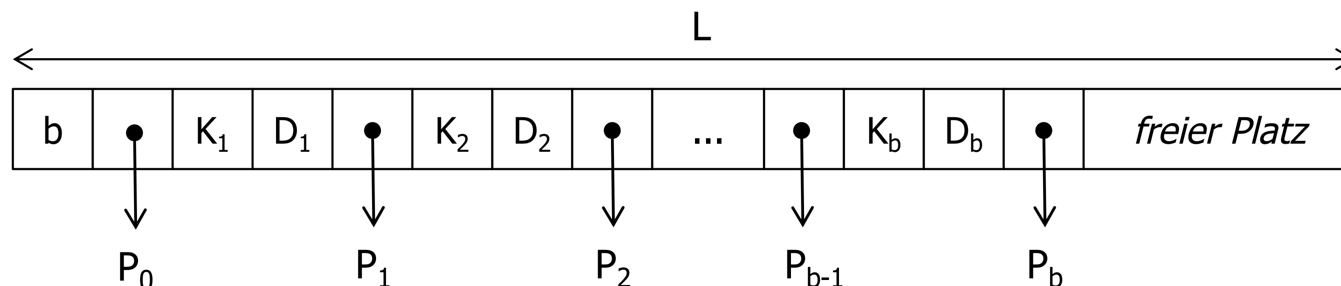
- B-Baum mit Grad 2



- In jedem Knoten stehen die Schlüssel in aufsteigender Ordnung mit $K_1 < K_2 < \ldots < K_b$.
- Jeder Schlüssel hat eine Doppelrolle als Identifikator eines Datensatzes und als Wegweiser im Baum
  - P_i verweist auf Teilbaum mit Schlüsseln > K_i und < K_{i+1}
  - in den Blattknoten sind die Verweise undefiniert
- Alle Schlüssel/Sätze können also sortiert aufgesucht werden
- Der Baum ist bzgl. der Knotenstruktur vollständig ausgeglichen.
- Jeder Knoten (außer der Wurzel) ist mindestens zur Hälfte gefüllt.

B-Bäume - Knotenformat

- Für einen B-Baum ergibt sich folgendes Knotenformat:



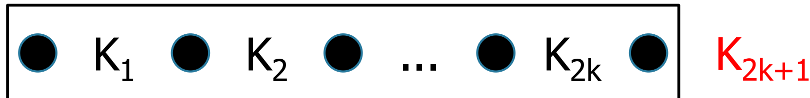
- Einträge (bei Knoten- oder Seitengröße L)
  - Die Felder für #Einträge (b), Schlüssel (K_i), Daten (D_i) und Zeiger haben die festen Längen l_b, l_K, l_D und l_p.
  - Maximale Anzahl von Einträgen pro Knoten: $b_{\max} = \left\lfloor \frac{L - l_b - l_p}{l_K + l_D + l_p} \right\rfloor = 2k$
- Was sind typische Größen in B-Bäumen?
  - (in Byte:) L = 8192, $l_b = 2$, $l_P = 4$, $l_K = 8$, $l_D = 88$

    → k = 40
  - Bei ausgelagerten Daten: $l_D = 4$ (Verweis auf Datenseite)

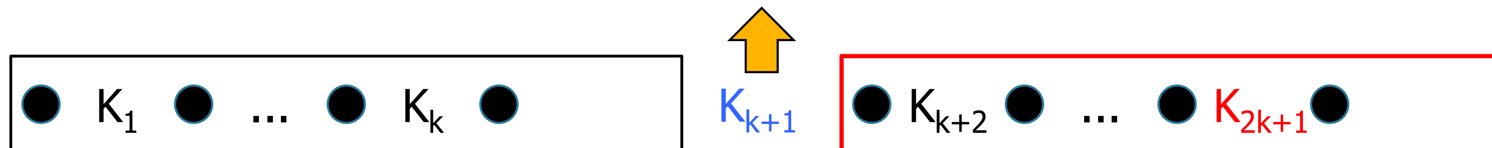
    → k = 255

B-Bäume - Einfügen

- Einfügen in B-Bäumen: Wurzel läuft über



- Fundamentale Operation: Split-Vorgang
  - Anforderung einer neuen Seite und
  - Aufteilung der Schlüsselmenge nach folgendem Prinzip



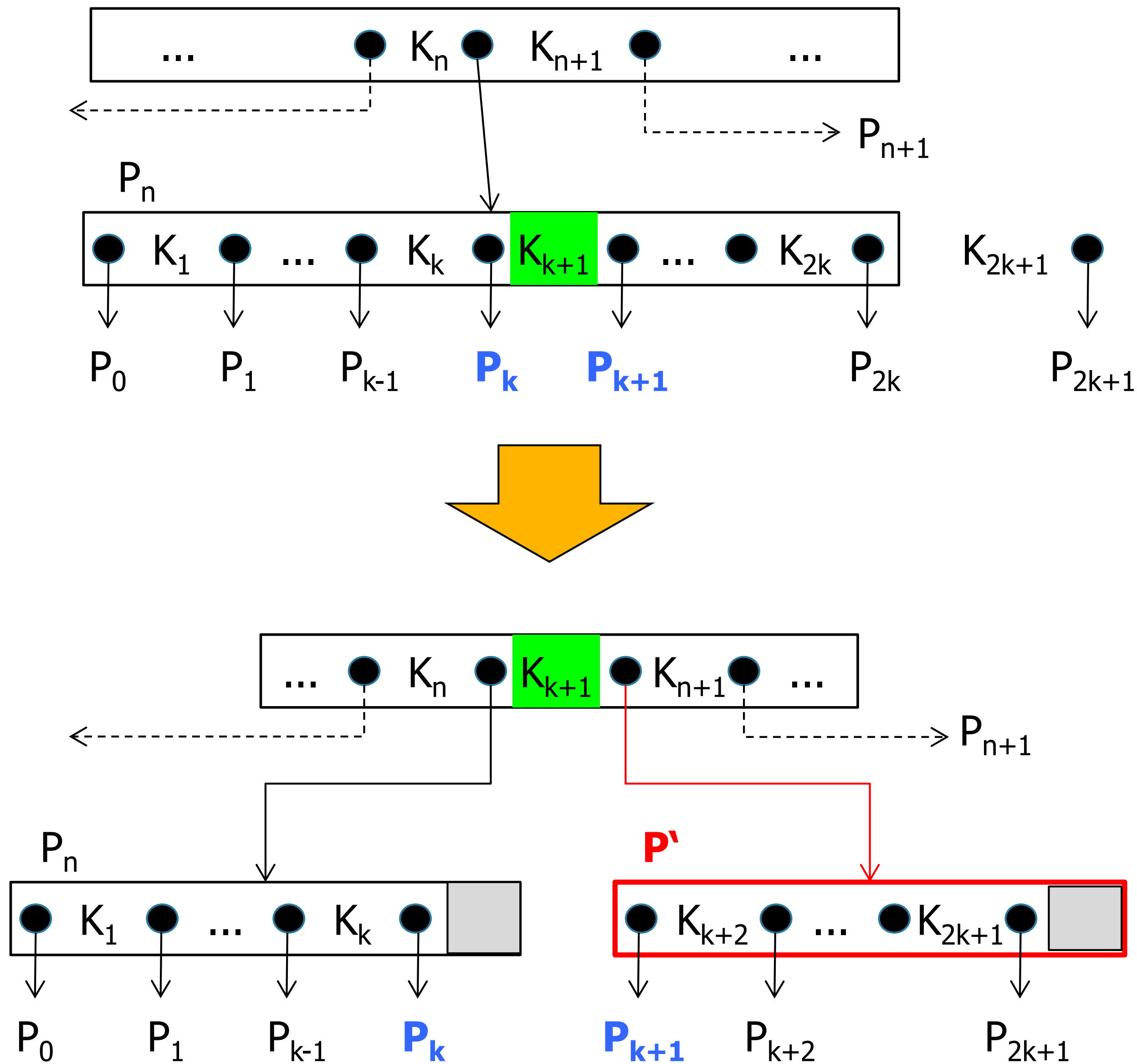
  - Der mittlere Schlüssel (Median) und der Verweis auf die neue Seite werden zum Vaterknoten gereicht.
  - Ggf. muss ein Vaterknoten angelegt werden (Anforderung einer neuen Seite).

B-Bäume - Einfügealgorithmus

- Ermittle Blattknoten
  - Suche nach dem Knoten P_n, in dem das neue Element gespeichert werden soll (immer ein Blatt)
  - Füge neues Element K_j in den ermittelten Knoten P_n ein
    - ElementEinfügen(P_n, $K_{j,}$ ∅)
- ElementEinfügen(P, K, V)
  - Suche Einfügeposition für K in P
  - Wenn Platz vorhanden ist, speichere Element K und V, sonst schaffe Platz durch Split-Vorgang und füge ein.
    - Fordere neue Seite (P‘) an
    - Ermittle Median K_{k+1} für 2k+1 Schlüsseleinträge (inkl. K_j)
    - Speichere Elemente mit $K < K_{k+1}$ in P, $K > K_{k+1}$ in P‘
    - Füge K_{k+1} und Verweis auf P‘ in die Vaterseite ein, falls diese vorhanden ist: ElementEinfügen(Vater(P), K_{k+1}, V(P‘))
    - Ansonsten: neue Wurzelseite mit V(P), K_{k+1}, V(P‘)

⇨ Split-Vorgang als allgemeines Wartungsprinzip

B-Bäume – Allgemeiner Splitvorgang



Aufbau eines B-Baumes mit Grad 2

Einfügereihenfolge: **77 12 48 69 33 89 95 90 37 45 83 2 15 87 97 98 99 50**

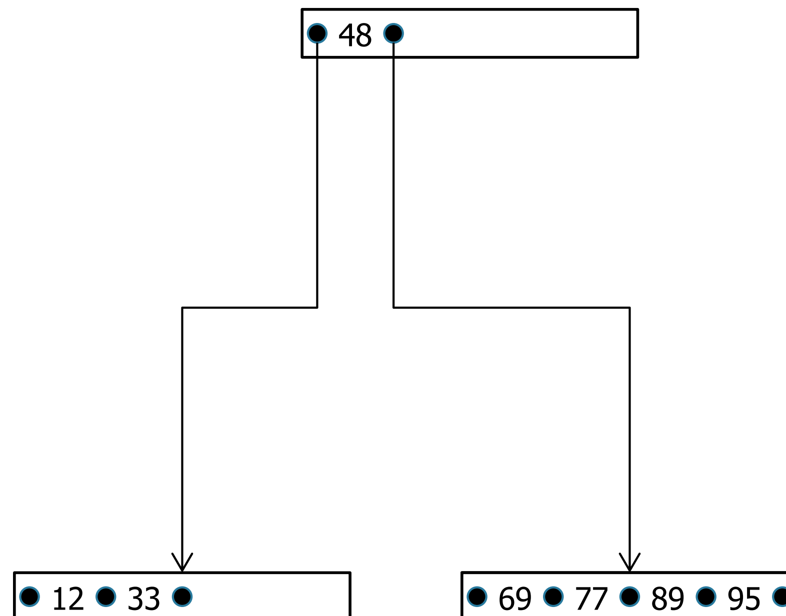
Aufbau eines B-Baumes mit Grad 2 (1)

Einfügereihenfolge: 77 12 48 69 **33 89 95 90 37 45 83 2 15 87 97 98 99 50**

● 12 ● 48 ● 69 ● 77 ●

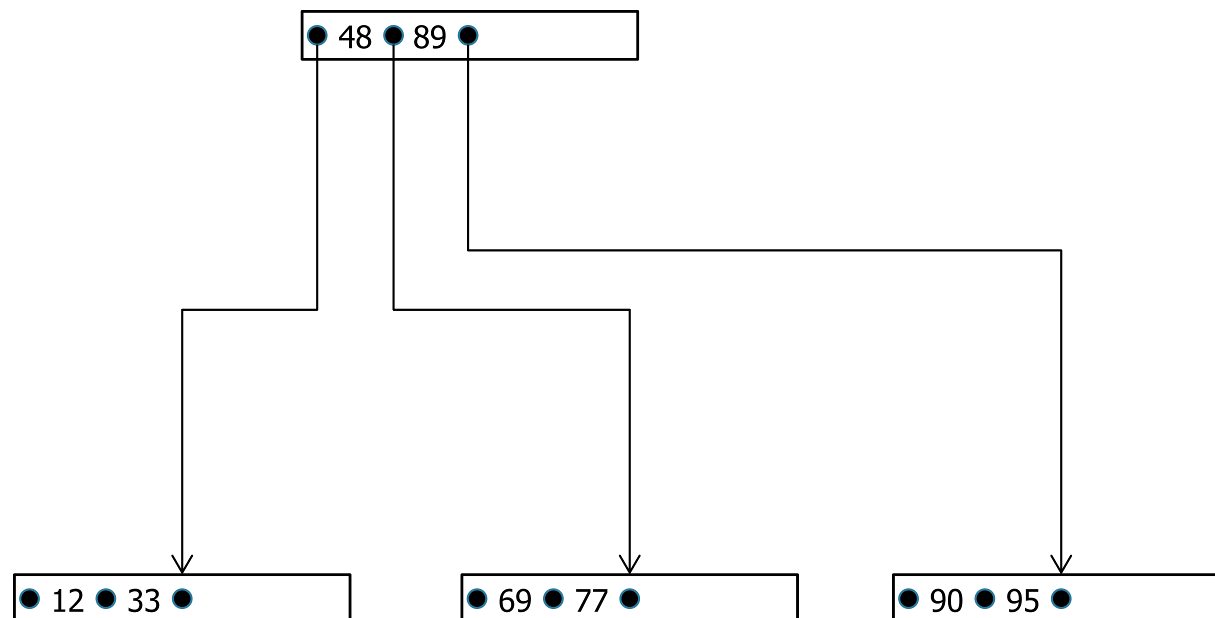
Aufbau eines B-Baumes mit Grad 2 (2)

Einfügereihenfolge: 77 12 48 69 33 89 95 **90 37 45 83 2 15 87 97 98 99 50**



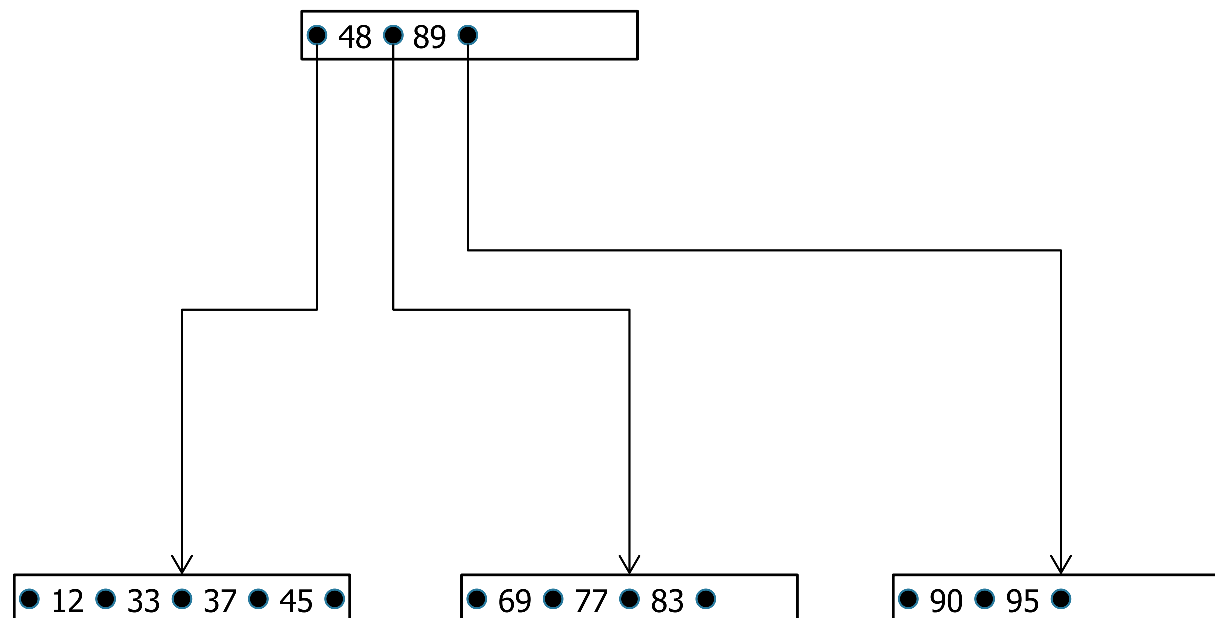
Aufbau eines B-Baumes mit Grad 2 (3)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 **37 45 83 2 15 87 97 98 99 50**



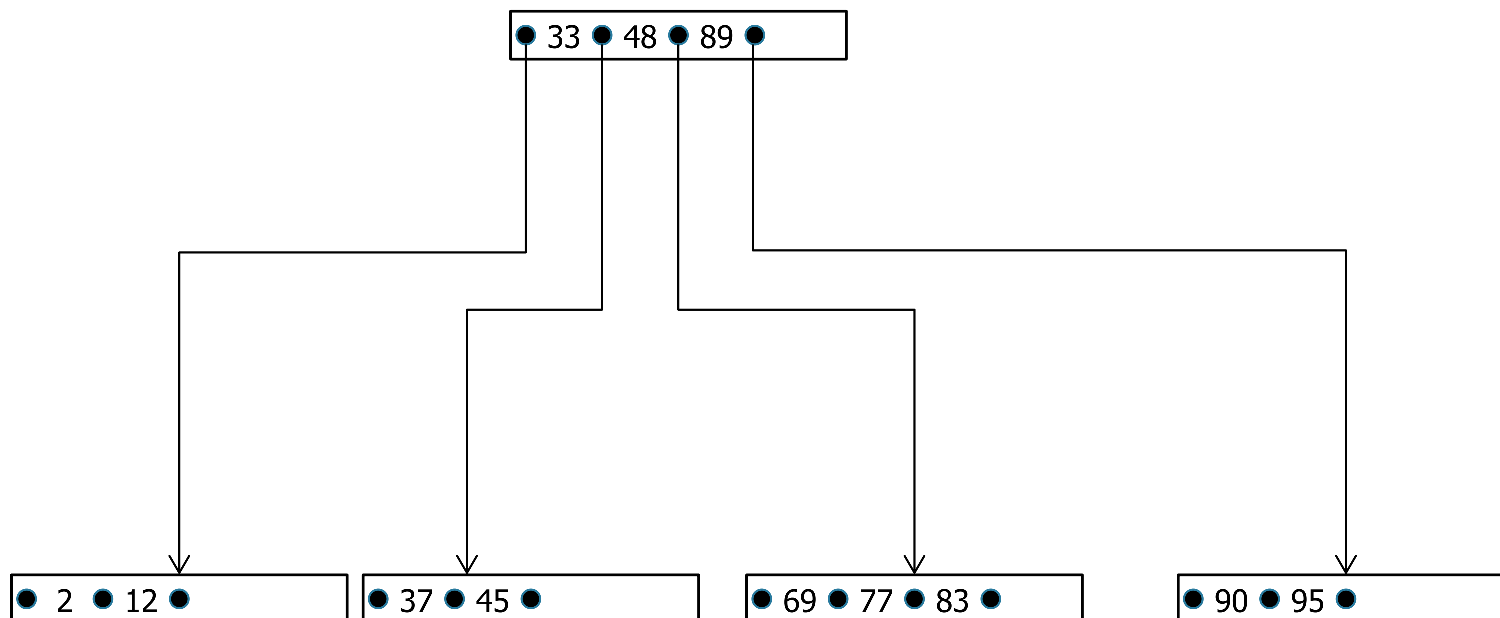
Aufbau eines B-Baumes mit Grad 2 (4)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 **2 15 87 97 98 99 50**



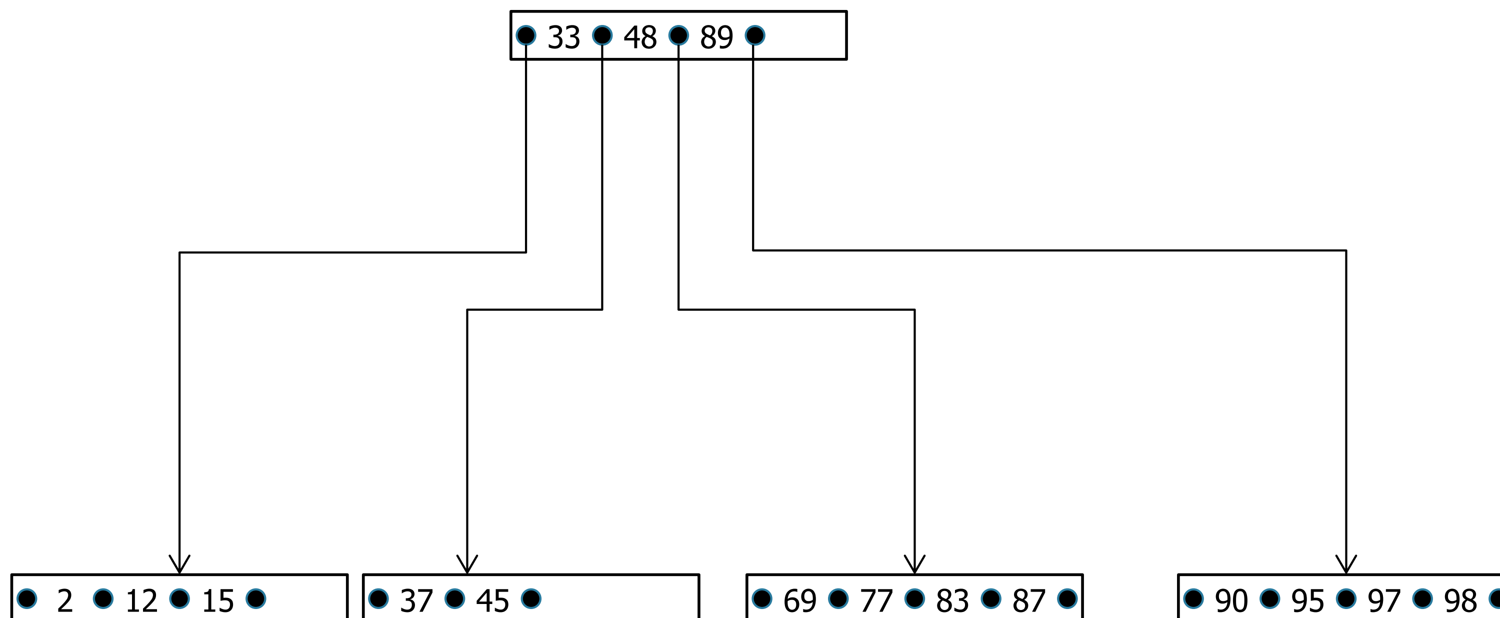
Aufbau eines B-Baumes mit Grad 2 (5)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 2 **15 87 97 98 99 50**



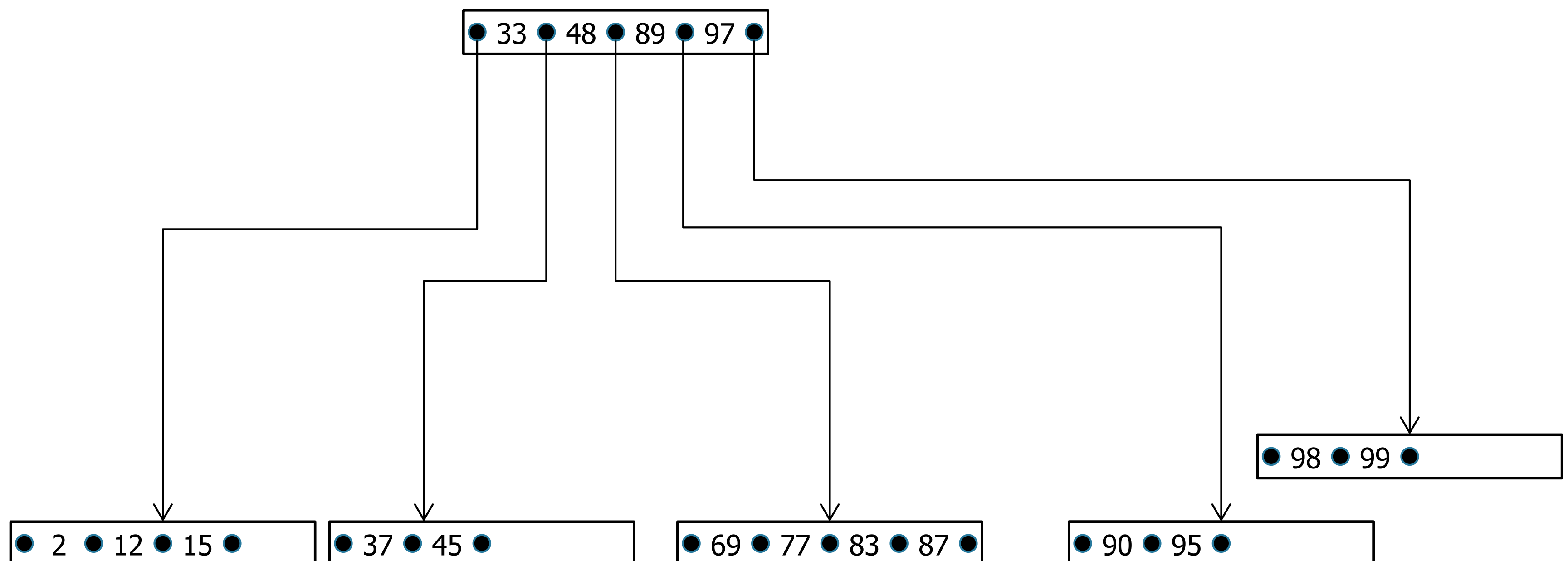
Aufbau eines B-Baumes mit Grad 2 (6)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 2 15 87 97 98 **99** **50**



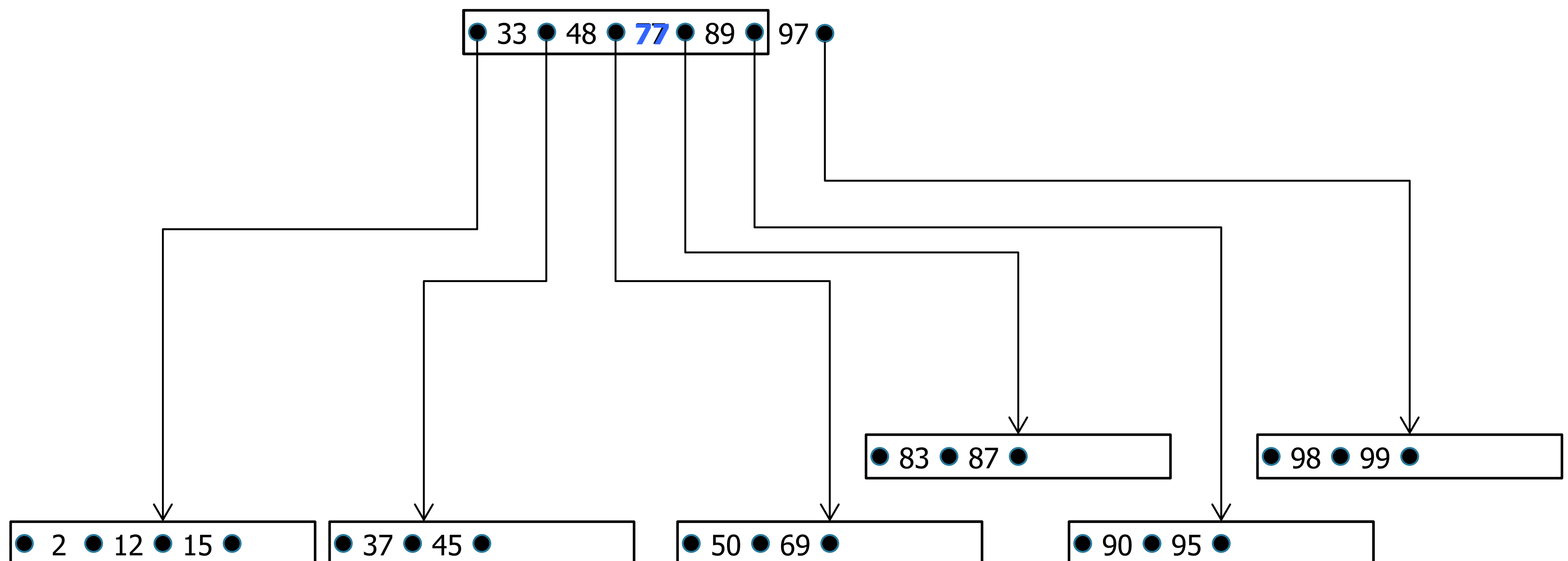
Aufbau eines B-Baumes mit Grad 2 (7)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 2 15 87 97 98 99 **50**



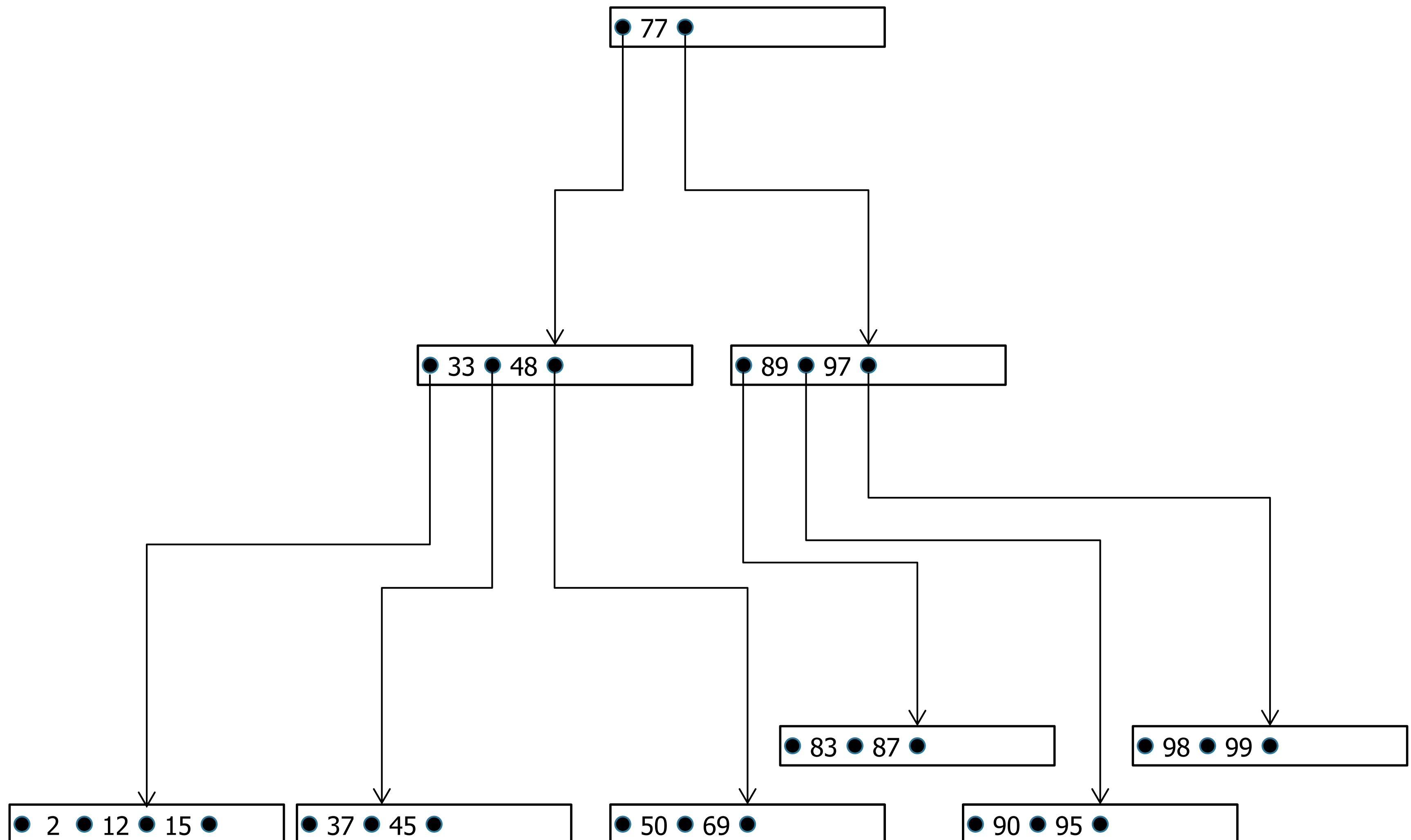
Aufbau eines B-Baumes mit Grad 2 (8)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 2 15 87 97 98 99 50



Aufbau eines B-Baumes mit Grad 2 (9)

Einfügereihenfolge: 77 12 48 69 33 89 95 90 37 45 83 2 15 87 97 98 99 50

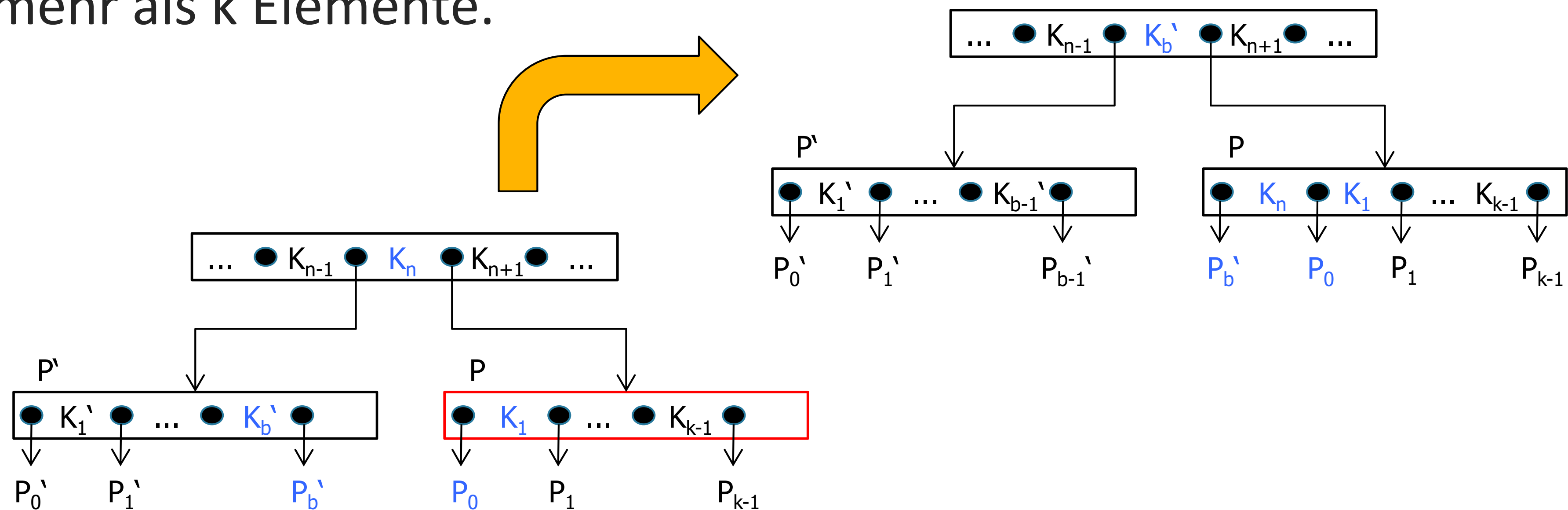


B-Bäume - Löschen

Die B-Baum-Eigenschaft muss wiederhergestellt werden, wenn die Anzahl der Elemente in einem Knoten kleiner als k wird. Durch **Ausgleich** mit Elementen aus einer Nachbarseite oder durch **Mischen** (Konkatenation) mit einer Nachbarseite wird dieses Problem gelöst.

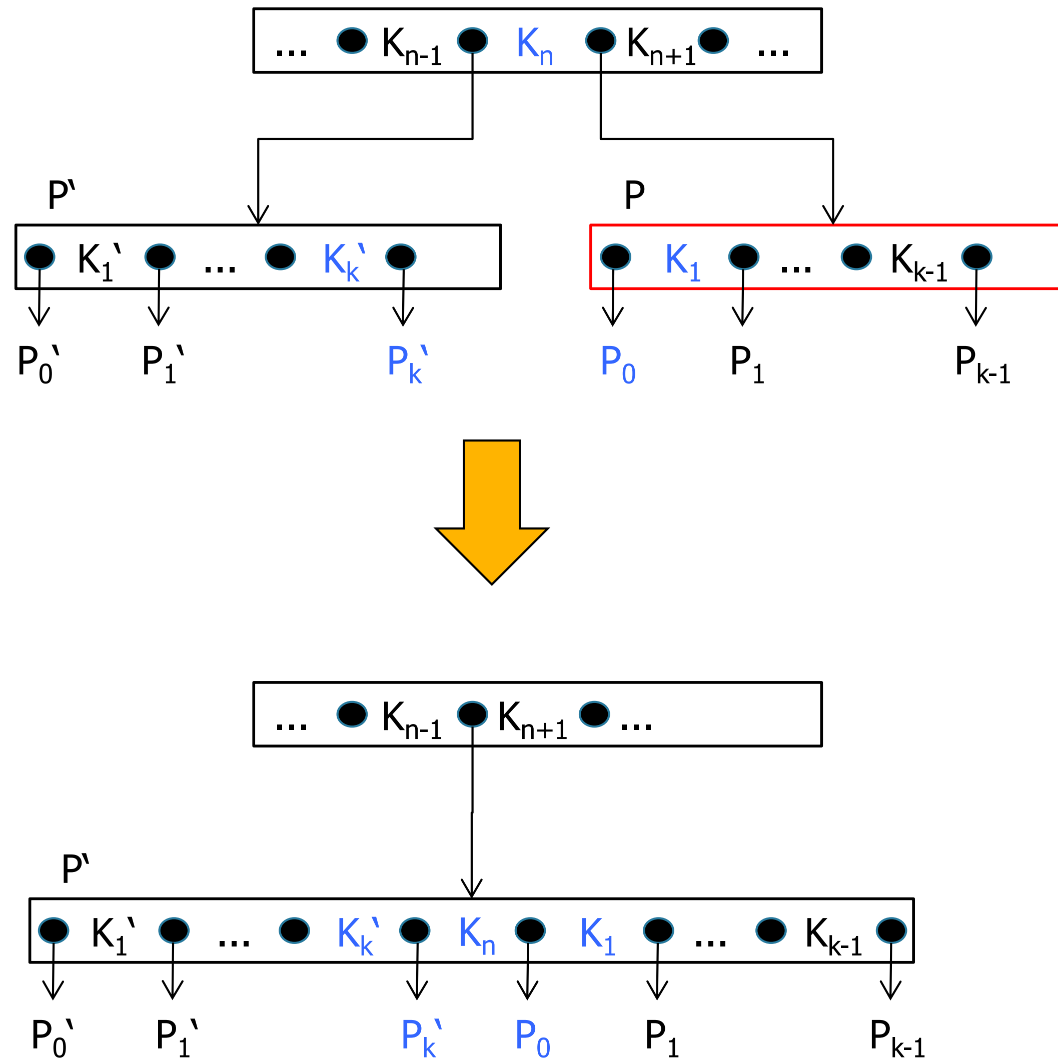
(1) Maßnahme: Ausgleich durch Verschieben von Schlüsseln

Beim Ausgleichsvorgang sind in der Seite P k–1 Elemente und in P' mehr als k Elemente.



B-Bäume – Löschen (2)

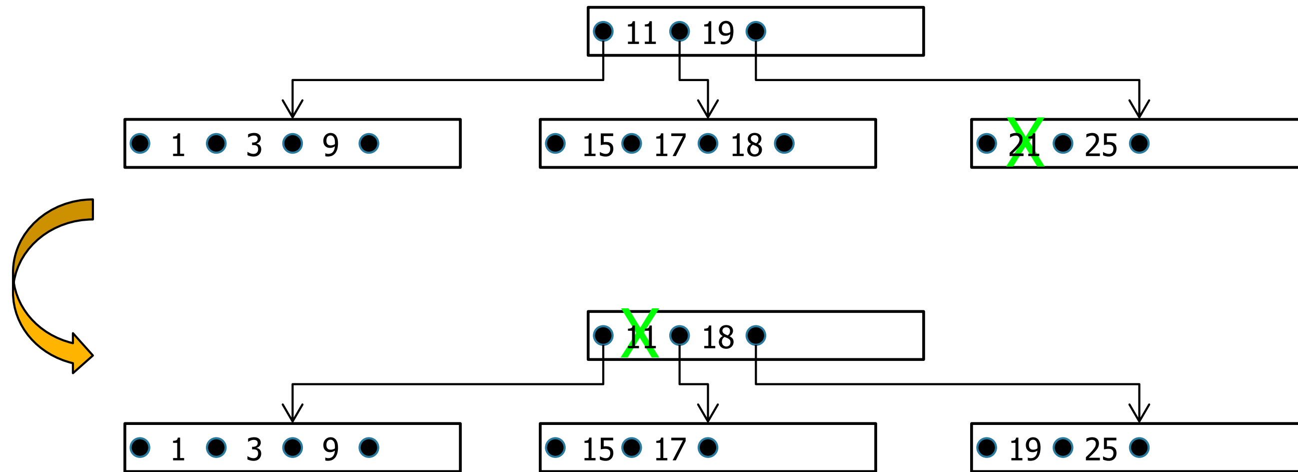
(2) Maßnahme: Mischen von Seiten



B-Bäume - Löschalgorithmus

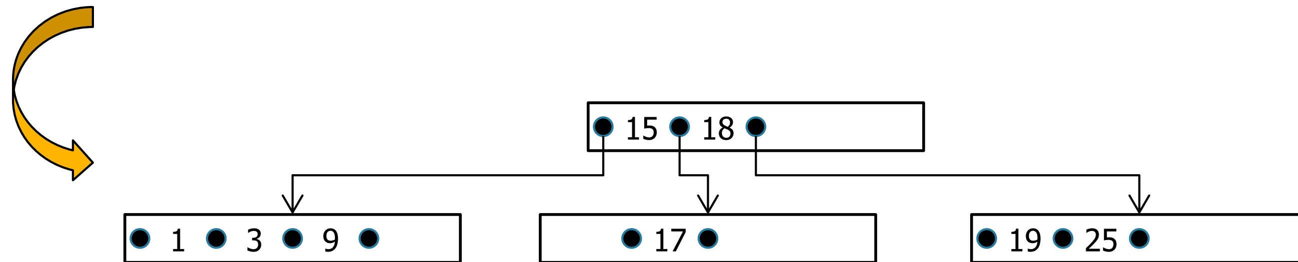
1. Löschen in Blattseite
   - Suche x in Seite P
   - Entferne x in P und wenn
     a) $b \geq k$ in P: tue nichts
     b) $b = k–1$ in P und $b > k$ in P': gleiche Unterlauf über P' aus
     c) $b = k–1$ in P und $b = k$ in P': mische P und P'.
2. Löschen in innerer Seite
   - Suche x
   - Ersetze $x = K_i$ durch kleinsten Schlüssel y in $B(P_i)$ oder größten Schlüssel y in $B(P_{i-1})$ (nächstgrößerer oder nächstkleinerer Schlüssel im Baum)
   - Entferne y im Blatt P
   - Behandle P wie unter (1)

B-Bäume – Löschbeispiele (3)

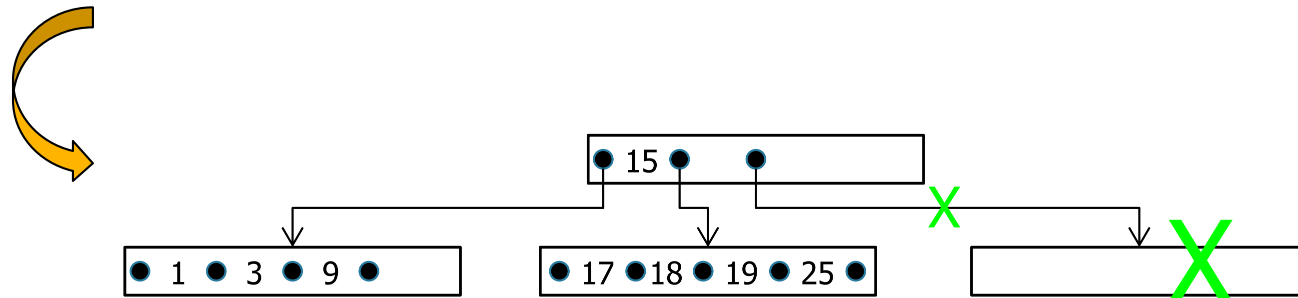


Lösche 21:
(Ausgleich)

Lösche 11:
(innerer Knoten)



Mischen:



B^+-Bäume im Vergleich zu B-Bäumen

In B-Bäumen spielen die Einträge (K_i, D_i, P_i) in den inneren Knoten zwei ganz verschiedene Rollen:

- Die zum Schlüssel K_i gehörenden Daten D_i werden gespeichert.
- Der Schlüssel K_i dient als Wegweiser im Baum.

Für diese zweite Rolle ist D_i vollkommen bedeutungslos. In B^+-Bäumen wird in inneren Knoten nur die Wegweiser-Funktion ausgenutzt, d. h., es sind nur (K_i, P_i) als Einträge zu führen.

- Wir sparen also Platz in den inneren Knoten und können mehr Wegweiser-Einträge dort speichern
- Der Fan-out erhöht sich beträchtlich, der Baum wird flacher

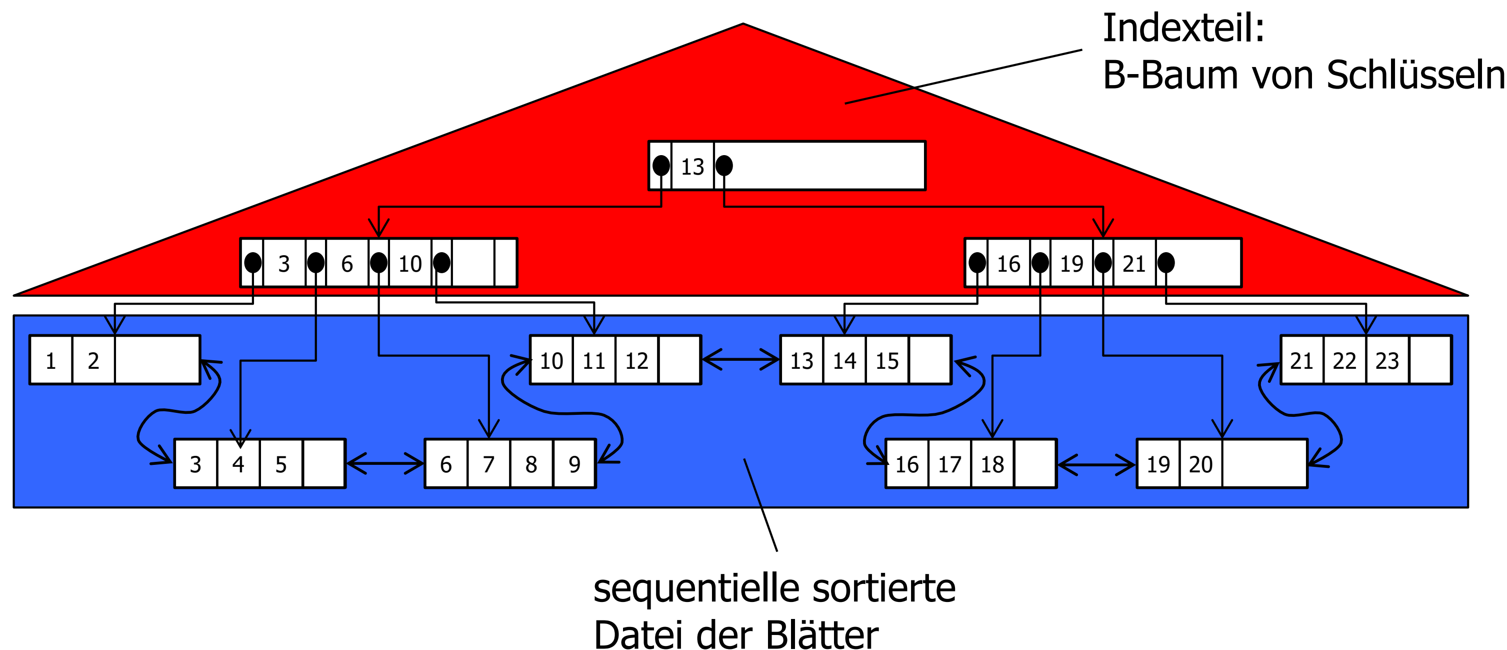
B^+-Bäume im Vergleich zu B-Bäumen (2)

- Die Information (K_i, D_i) wird in den Blattknoten abgelegt.
- Für einige K_i ergibt sich eine redundante Speicherung. Die inneren Knoten bilden also einen Index (index part), der einen schnellen direkten Zugriff zu den Schlüsseln gestattet.
- Die Blätter enthalten alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge. Durch Verkettung aller Blattknoten (sequence set) lässt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte.

⇨ Die für den praktischen Einsatz wichtigste Variante des B-Baums ist der B^+-Baum.

B^+-Bäume - Erklärungsmodell

Der B^+-Baum lässt sich auffassen als eine gekettete sequentielle Datei von Blättern, die einen Indexteil besitzt, der selbst ein B-Baum ist. Im Indexteil werden insbesondere beim Split-Vorgang die Operationen des B-Baums eingesetzt.



B^+-Bäume - Definition

- Definition:

Der so definierte Baum heißt in der Literatur gelegentlich auch B*-Baum.

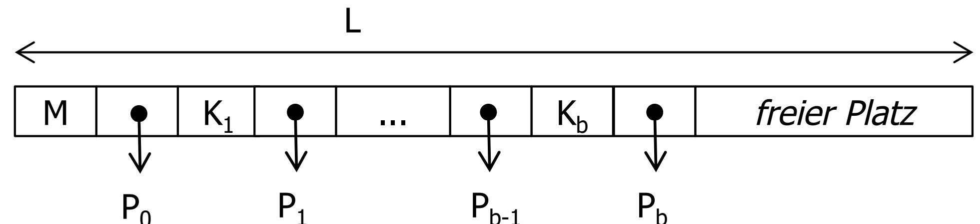
Seien k, und k* ganze Zahlen, k, k* > 0. Ein B^+-Baum B vom Typ (k, k*) ist entweder ein leerer Baum oder ein geordneter Suchbaum, für den gilt:

1. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge.
2. Jeder Knoten - außer Wurzeln und Blättern - hat mindestens k und höchstens 2k Einträge. Blätter haben mindestens k* und höchstens 2k* Einträge. Die Wurzel hat entweder maximal 2k Einträge oder sie ist ein Blatt mit maximal 2k* Einträgen.
3. Jeder Knoten mit n Einträgen, außer den Blättern, hat n+1 Kinder.
4. Seien R_1, ... R_n die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit n+1 Kindern. Seien V_0, ..., V_n die Verweise auf diese Kinder. Dann gilt:
   a) V_0 verweist auf den Teilbaum mit Schlüsseln kleiner als R_1.
   b) V_i $(0 < i < n)$ weist auf den Teilbaum mit Schlüsseln zwischen R_i und R_{i+1} (einschließlich R_i).
   c) V_n verweist auf den Teilbaum mit Schlüsseln größer oder gleich R_n

B^+-Bäume - Knotenformate

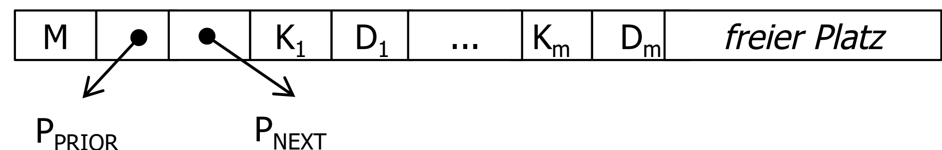
- Unterscheidung von zwei Knotenformaten:
  - M: Kennung des Seitentyps + #Einträge; feste Länge L
  - innere Knoten ($k \le b \le 2k$)
    - $L = l_M + l_p + 2k(l_K + l_p)$

$$k = \left\lfloor \frac{L - l_M - l_p}{2 * (l_K + l_p)} \right\rfloor$$



  - Blattknoten ($k^* \le m \le 2k^*$)
    - $L = l_M + 2l_p + 2k^*(l_K + l_D)$

$$k^* = \left\lfloor \frac{L - l_M - 2l_p}{2 * (l_K + l_D)} \right\rfloor$$



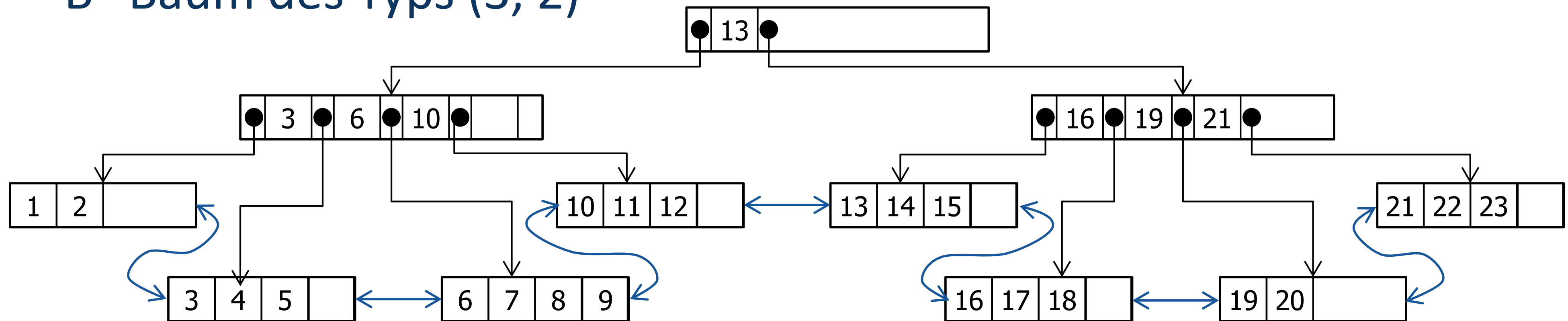
- Was sind typische Größen in B^+-Bäumen?

  (in Byte:) $L = 8192$, $l_M = 4$, $l_P = 4$, $l_K = 8$, $l_D = 88$

  Innerer Knoten: $k = 341$; Blatt: $k^* = 42$

B⁺-Bäume – Beispiel

- B⁺-Baum des Typs (3, 2)



- Jeder Knoten hat die Größe eines I/O Blocks.
- Ein Knoten ist zu mindestens 50% gefüllt.
- Das Lesen eines Knotens verursacht typischerweise einen wahlfreien Zugriff auf Festplatte
- Ein B+ Baum ist i.d.R. sehr flach, d.h. Suche verursacht nur wenige wahlfreie Zugriffe.
- Zudem, die ersten 1-2 Ebenen des Baums sind i.d.R. im Hauptspeicher "gecached".

B^+-Bäume - Grundoperationen

(1) Direkte Suche:

Da alle Schlüssel in den Blättern sind, kostet die direkte Suche h^+ Zugriffe. h^+ ist jedoch im Mittel kleiner als h in B-Bäumen. Da f_{avg} beim B-Baum gut mit h abgeschätzt werden kann, erhält man also durch den B^+-Baum eine effizientere Unterstützung der direkten Suche.

(2) Sequentielle Suche:

Sie erfolgt nach Aufsuchen des Linksaußen der Struktur unter Ausnutzung der Verkettung der Blattseiten. Es sind zwar ggf. mehr Blätter als beim B-Baum zu verarbeiten, doch da nur $h^+–1$ innere Knoten aufzusuchen sind, wird die sequentielle Suche ebenfalls effizienter ablaufen.

B^+-Bäume – Grundoperationen (2)

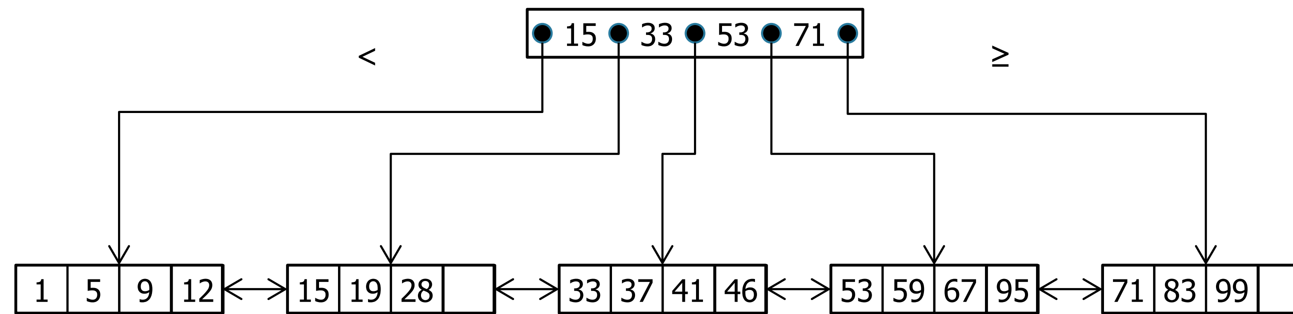
(3) Einfügen:

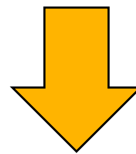
Es ist von der Durchführung und vom Leistungsverhalten her dem Einfügen in einen B-Baum sehr ähnlich. Bei inneren Knoten wird das Splitting analog zum B-Baum durchgeführt. Beim Split-Vorgang einer Blattseite muss gewährleistet sein, dass jeweils die niedrigsten Schlüssel einer Seite als Wegweiser in den Vaterknoten kopiert werden.
Die Verallgemeinerung des Split-Vorgangs ist analog zum B-Baum.

(4) Löschen:

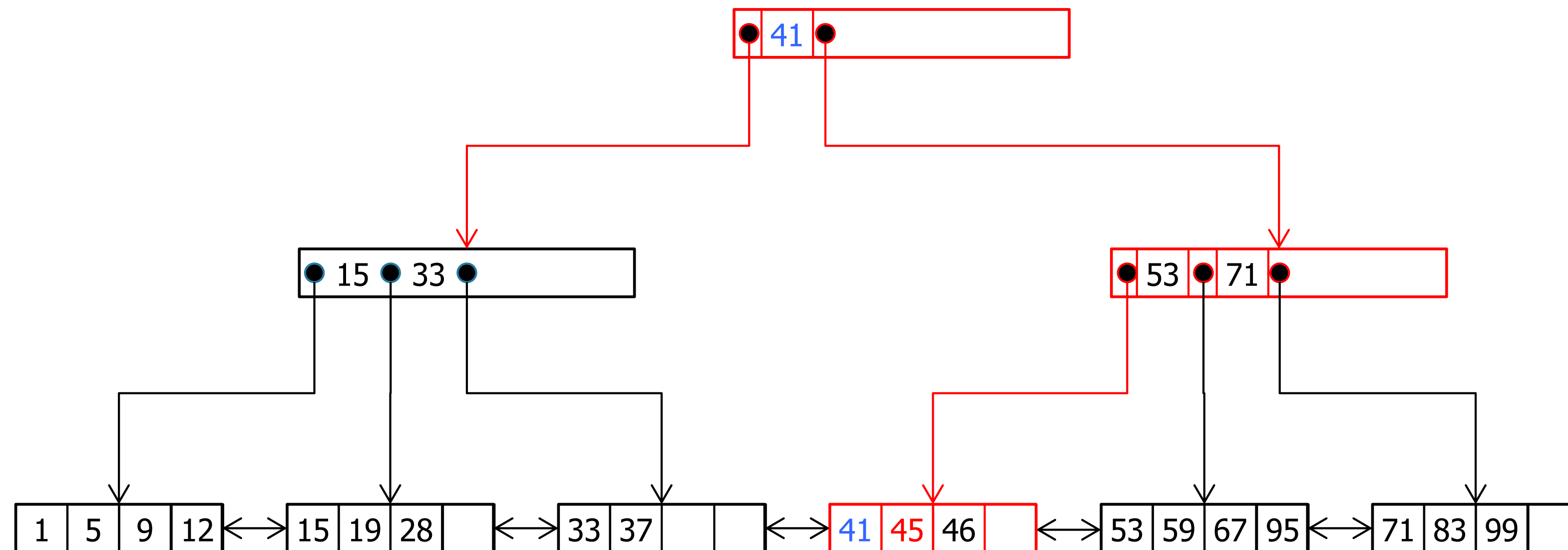
Datenelemente werden immer von einem Blatt entfernt (keine komplexe Fallunterscheidung wie beim B-Baum). Weiterhin muss beim Löschen eines Schlüssels aus einem Blatt dieser Schlüssel nicht aus dem Indexteil entfernt werden; er behält seine Funktion als Wegweiser.
Das Mischen von Blattknoten führt zur Löschung des Wegweisers im Vaterknoten.

B+-Bäume - Einfügebeispiel

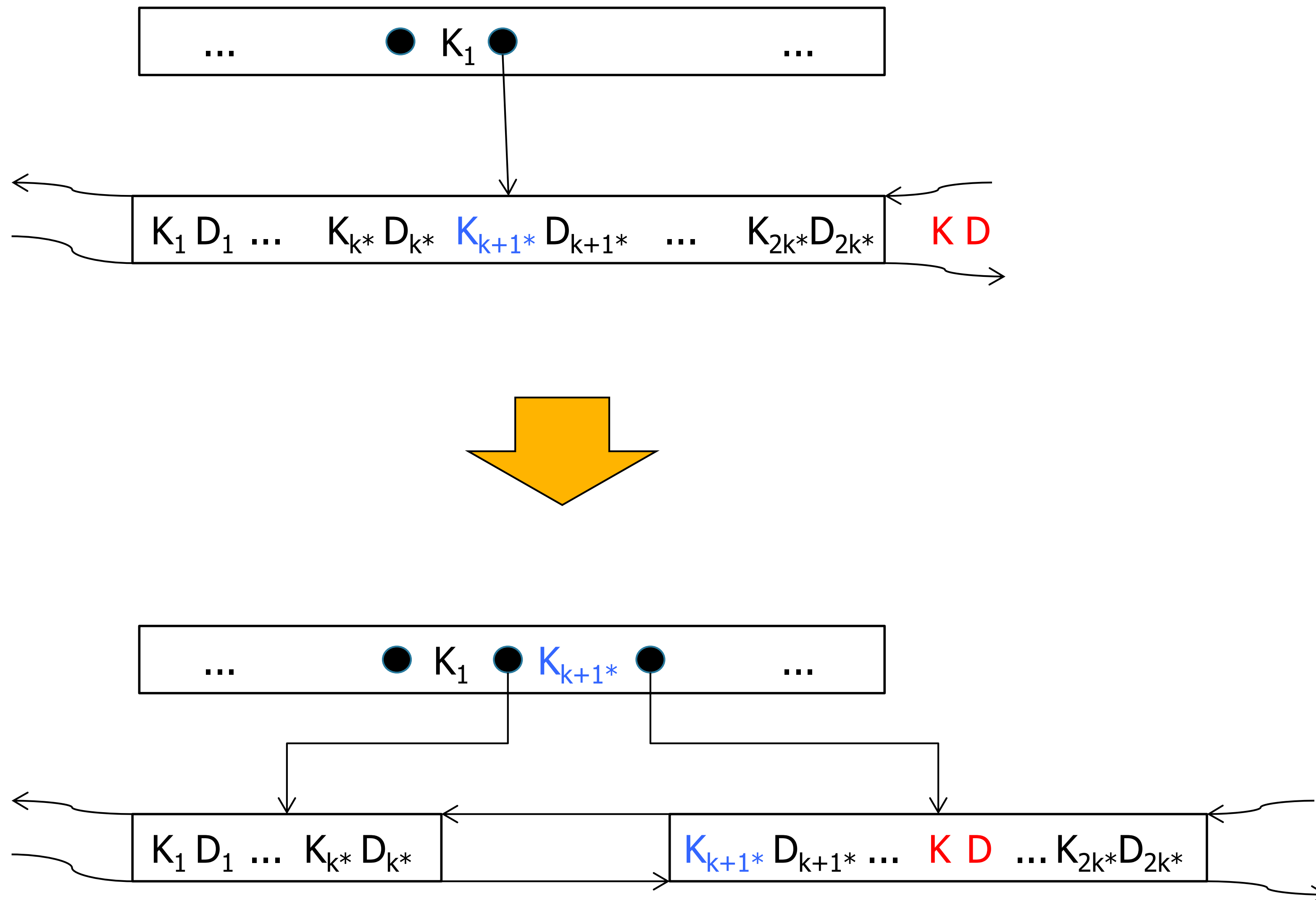




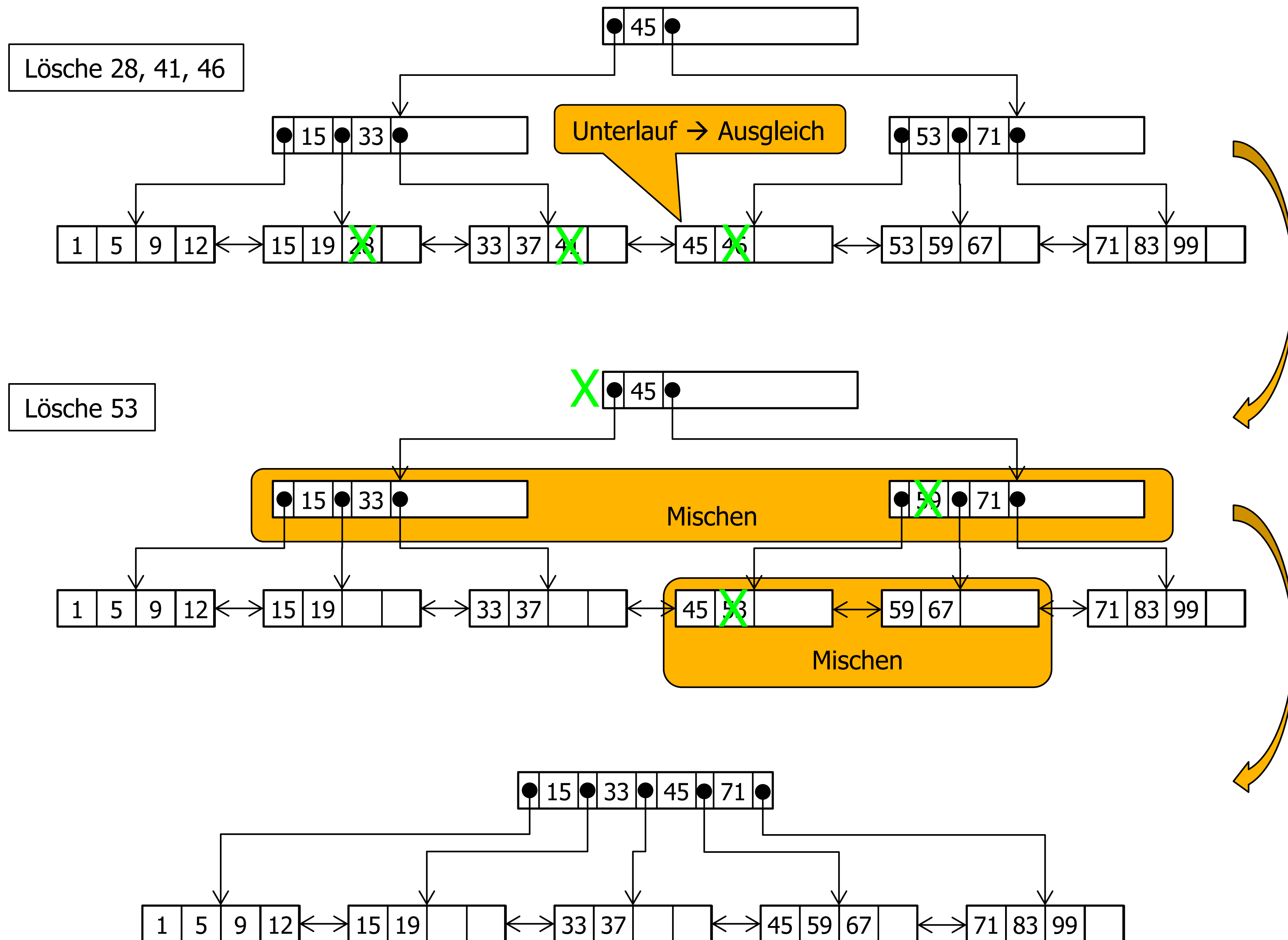
Einfügen von 45



B+-Bäume – Allgemeiner Splitvorgang



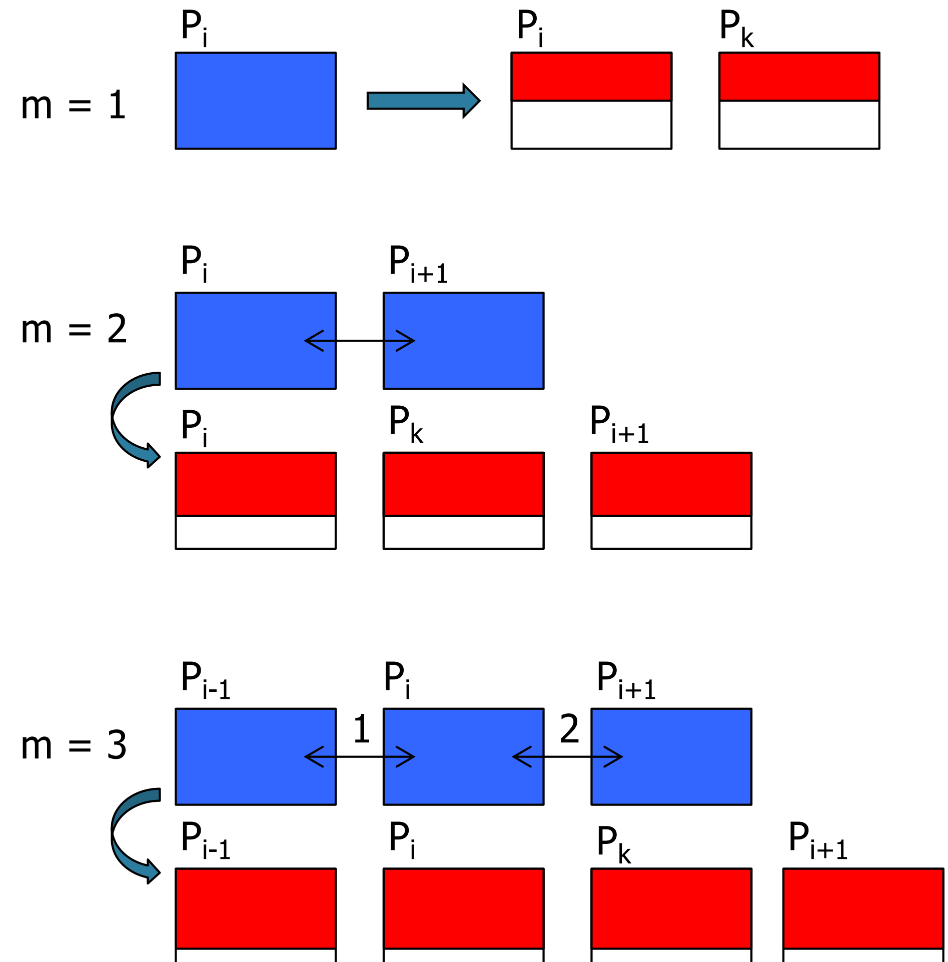
B+-Bäume - Löschen



B*-Bäume

- Verbesserung des Belegungsgrades
  - verallgemeinerte Überlaufbehandlung
  - Split-Faktor m > 1

➔ Verbesserte Speicherplatzbelegung, dafür höhere Einfügekosten



B*-Bäume – Belegungsgrad

- Welcher Belegungsgrad β des B*-Baums wird erzielt
  - bei einfacher Überlaufbehandlung (Split-Faktor m = 1)?
  - beim Einfügen einer sortierten Schlüsselfolge?
- Speicherplatzbelegung als Funktion des Split-Faktors

	Belegung		
Split-Faktor	β_{min}	β_{avg}	β_{max}
1	1/2 = 50%	$\ln 2 \approx 69\%$	1
2	2/3 = 66%	$2 \times \ln(3/2) \approx 81\%$	1
3	3/4 = 75%	$3 \times \ln(4/3) \approx 86\%$	1
m	m/(m+1)	$m \times \ln((m+1)/m)$	1

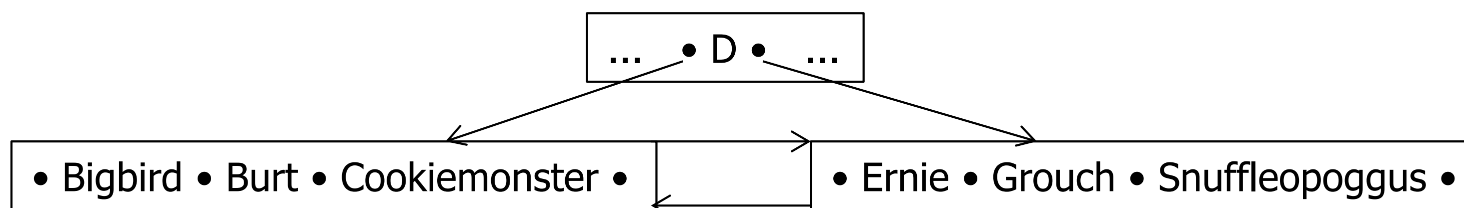
B^+-Bäume – Präfix-B^+-Bäume

Wegen der ausschließlichen Wegweiser-Funktion der Schlüssel in den inneren Knoten ist es nicht nötig, die Optimierungsmaßnahmen auf die Schlüssel zu stützen, die in den Blattknoten tatsächlich vorkommen.

➪ Einsatz von minimalen Separatoren als Wegweiser

- Beispiel: Konstruktion irgendeines Separators *S* mit der Eigenschaft

Cookiemonster ≤ *S* < Ernie



- Verbesserung der Baumbreite (fan-out)
  - Erhöhung der Anzahl der Zeiger in den inneren Knoten
  - Schlüsselkomprimierung und Nutzung von „Wegweisern”
  - Variabel lange Einträge erfordern Kontrolle des Unterlaufs über die tatsächliche Speicherplatzbelegung

R. Bayer, K. Unterauer: Prefix B-trees. ACM TODS 2, 1 (March 1977)

Präfix-Eigenschaft

Sind die Schlüssel Worte über einem Alphabet und ist die Ordnung der Schlüssel in alphabetischer (lexikographischer) Ordnung, dann gilt folgende Eigenschaft:

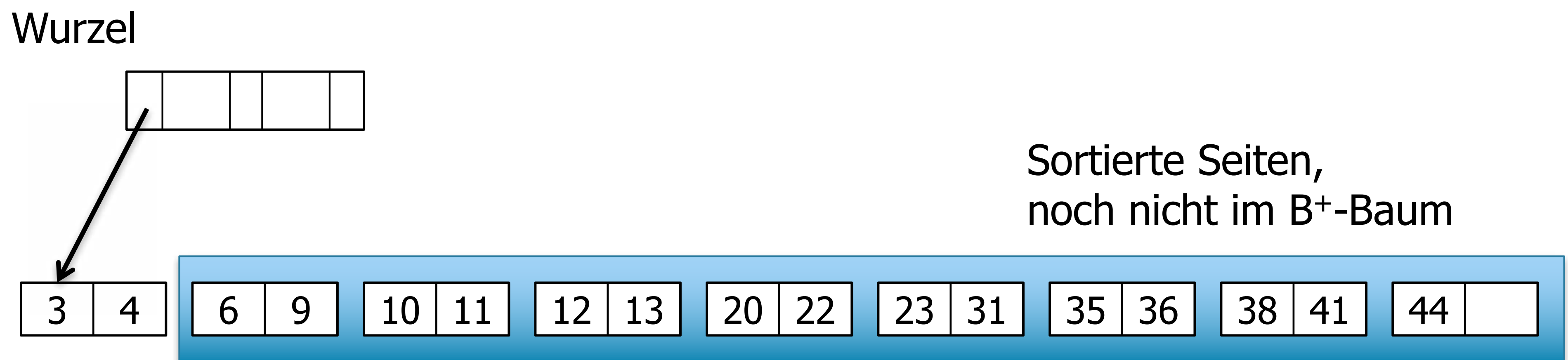
- Präfix-Eigenschaft:
  - Seien x und y zwei Schlüssel, so dass x < y. Dann gibt es einen eindeutigen Präfix ŷ von y mit:
    - (a) ŷ ist ein Separator zwischen x und y und
    - (b) kein anderer Separator zwischen x und y ist kürzer als ŷ.
- Beim Split einer Blattseite mit Split zwischen x und y wird nun ŷ als Wegweiser für den Vaterknoten bestimmt.
- Es kann auch sinnvoll sein, Index-Knoten nicht genau in der "Mitte" zu splitten, sondern eine leichte Abweichung davon zuzulassen. Wieso? Was ist ein guter Separator zwischen Donaudampfschiff-fahrtsdampferhersteller und Donaudampfschifffahrtsgesellschaft?

Bulkloading

- Problemstellung
  - Gegeben eine große Menge an Datensätzen
  - Wie kann ein B+ Baum darüber aufgebaut werden?
  - Das Problem ist natürlich allgemeiner und tritt auch bei anderen Bäumen/Indexstrukturen auf.
- Naive Möglichkeit
  - Datensätze einzeln nach und nach in B+ Baum einfügen
  - D.h. es wird immer von der Wurzel ab nach der passenden Einfügestelle gesucht
  - Nicht sehr effizient (auch wenn die oberen Ebenen in einem DB-Puffer sind)
- Idee
  - Sortiere Daten vor (resultiert in sortierte Liste von Seiten)
  - Dann baue Baum auf, indem Datenseiten der Reihe nach hinzugefügt werden

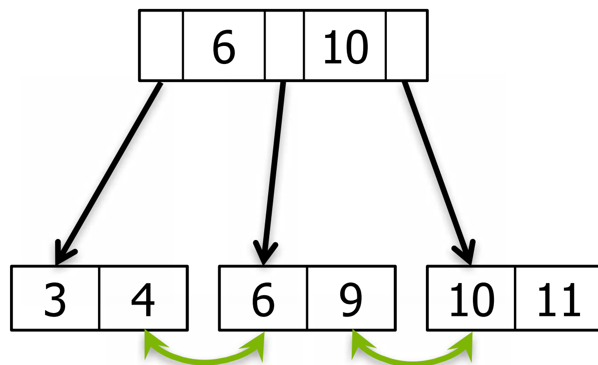
Ablaufbeispiel

- Annahme: der B⁺-Baum kann 2 Einträge pro Seite speichern.
- Wir beginnen mit einer leeren Wurzel und fügen immer ganze Seiten ein, hier zuerst die Seite mit Einträgen 3 und 4.

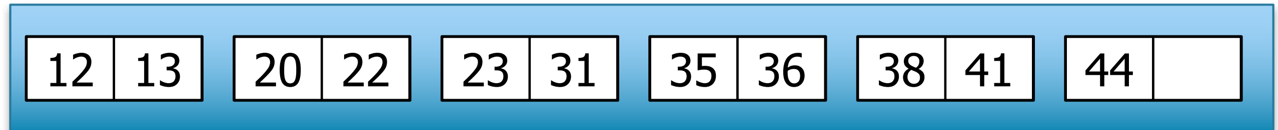


Ablaufbeispiel (2)

- Nachdem die beiden folgenden Seiten eingefügt worden sind ist die Wurzel nun voll.
- Beim Einfügen von (12,13) muss die Wurzel also aufgeteilt werden.

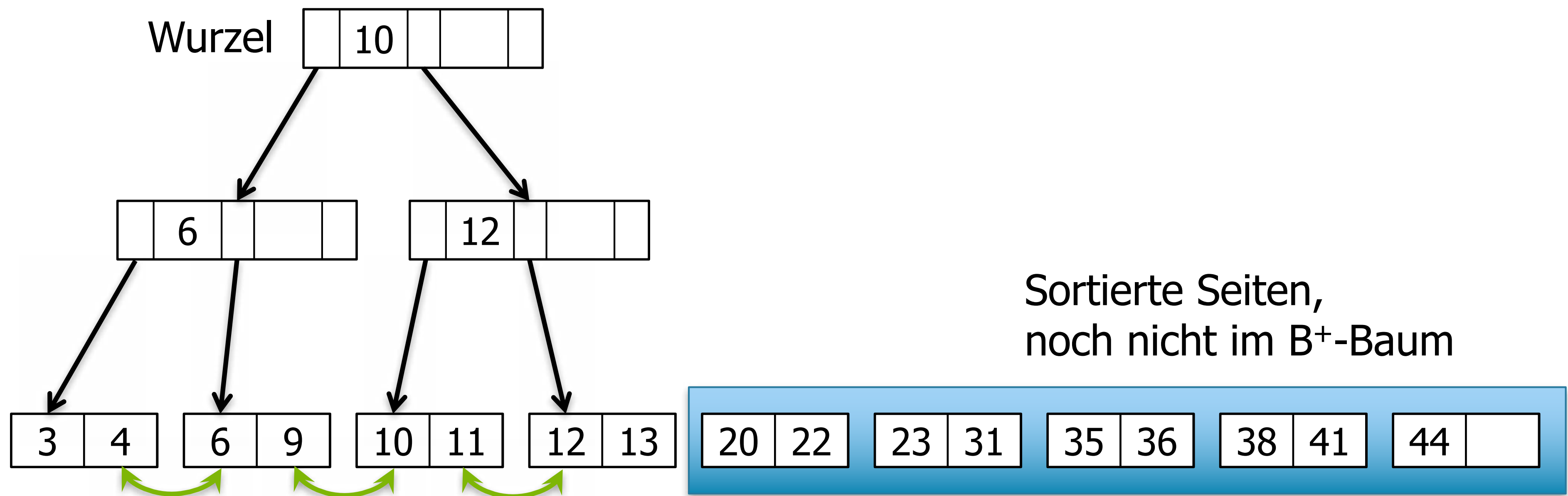


Sortierte Seiten,
noch nicht im B^+-Baum



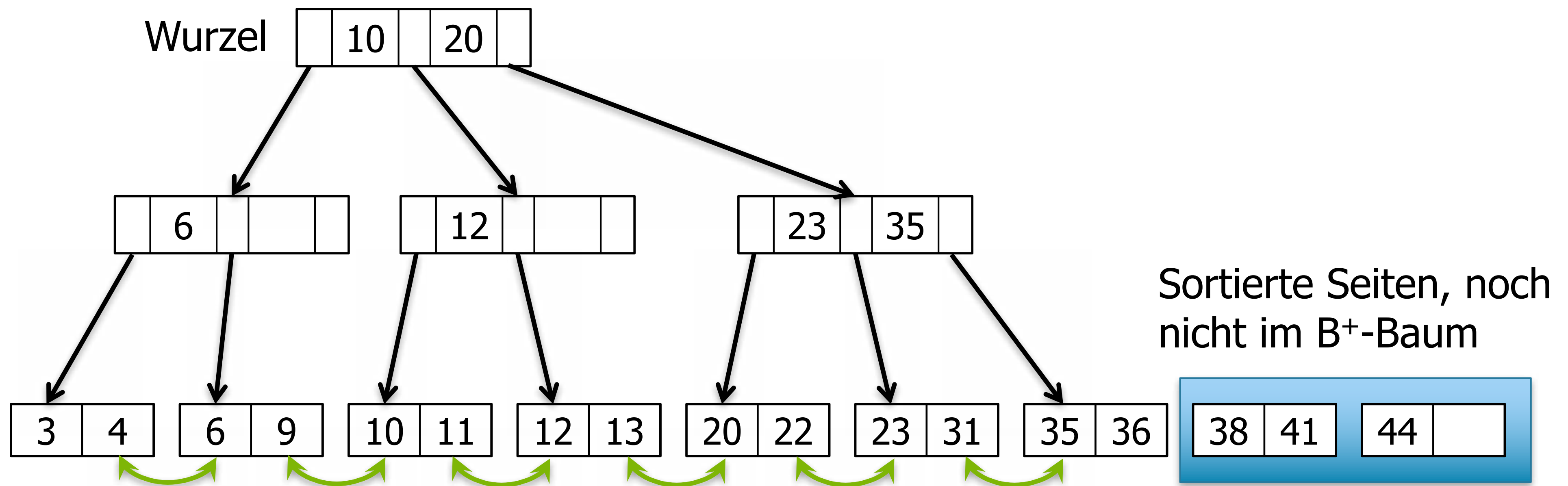
Ablaufbeispiel (3)

- Beim Aufteilen der Seiten wurden diese hier gleichmäßig auf die neuen Seiten unter der Wurzel aufgeteilt
- Im Allgemeinen hätte man auch anders aufteilen können
  - z.B. nach einem bestimmen Füllgrad (z.B. 80%)
  - oder man hätte auch alle alten Einträge in der linken Seite belassen können.

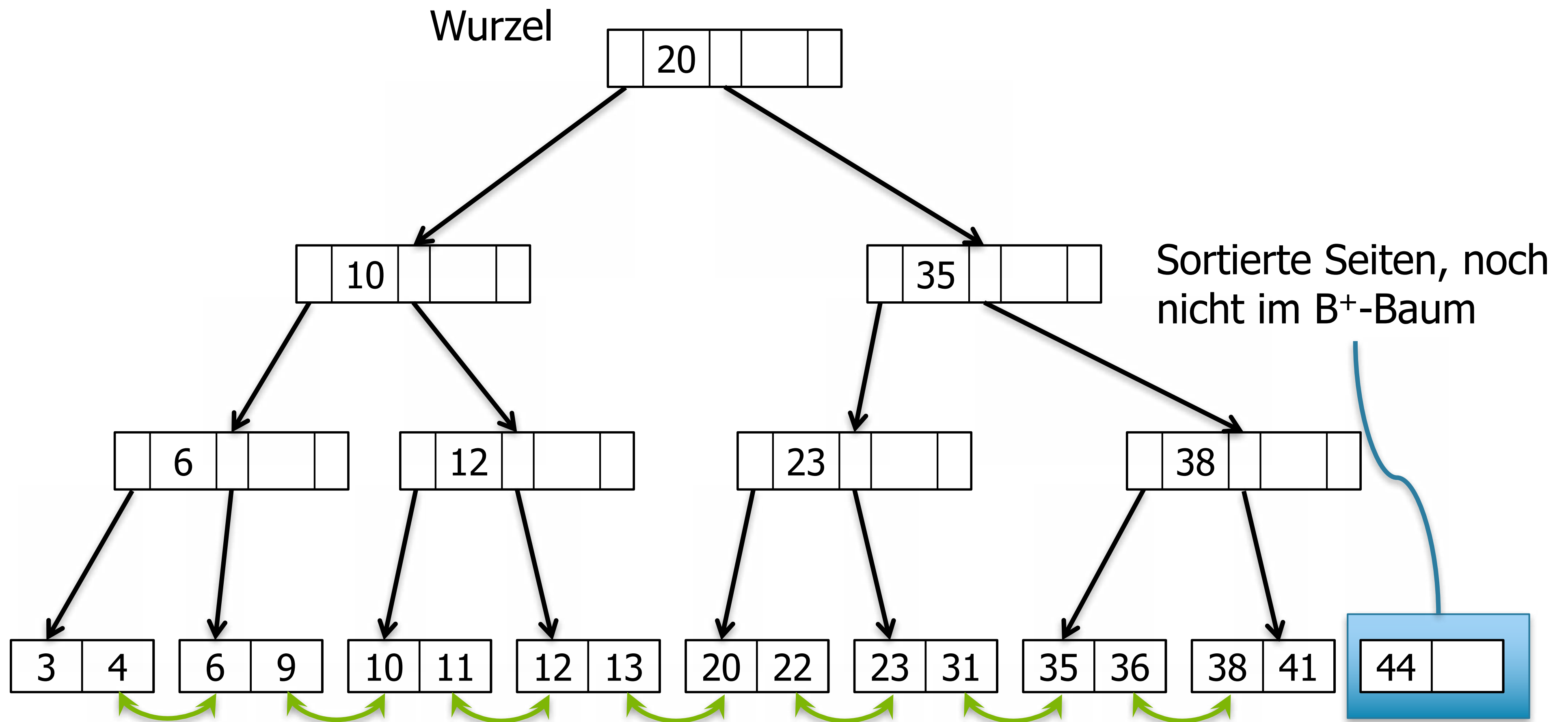


Ablaufbeispiel (4)

- Index-Einträge für die Blätter werden immer in den am weitesten rechts stehenden Index-Knoten direkt über der Blatt-Ebene eingefügt.
- Sollte dieser voll sein, so muss aufgeteilt werden.



Ablaufbeispiel (5)



Hashverfahren

- Gegeben: Domäne der Schlüssel $S = \{0, \dots M - 1\}$
- M kann sehr groß sein
- Anfragen der Form: welche Datensätze haben den Schlüssel x?
- Sogenannte Punktanfragen

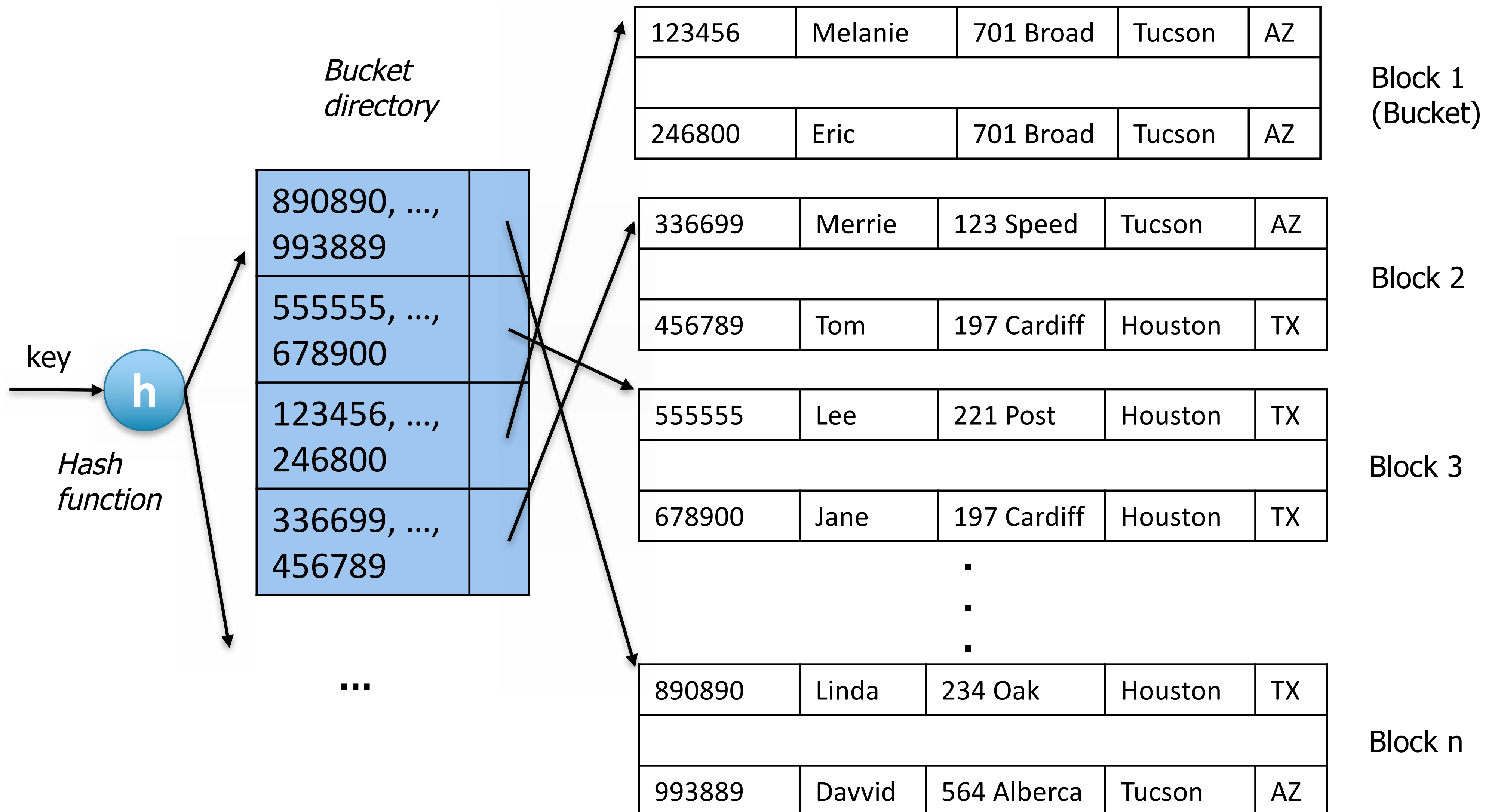
Grundidee

- Abbildung auf Adressbereich A, $|A| << |S|$
- $h : S \rightarrow A$ ordnet jedem Schlüssel einen Wert in A zu
- Gewünschte Eigenschaften:
  - Verteile Schlüssel gleichmäßig über Adressbereich
  - **Vermeide Kollisionen**
  - Kollision: verschiedene Werte in S werden auf den gleichen Wert in A abgebildet
- Einheiten im Adressbereich werden auch Eimer (Englisch: Buckets) genannt.

Hashverfahren (2)

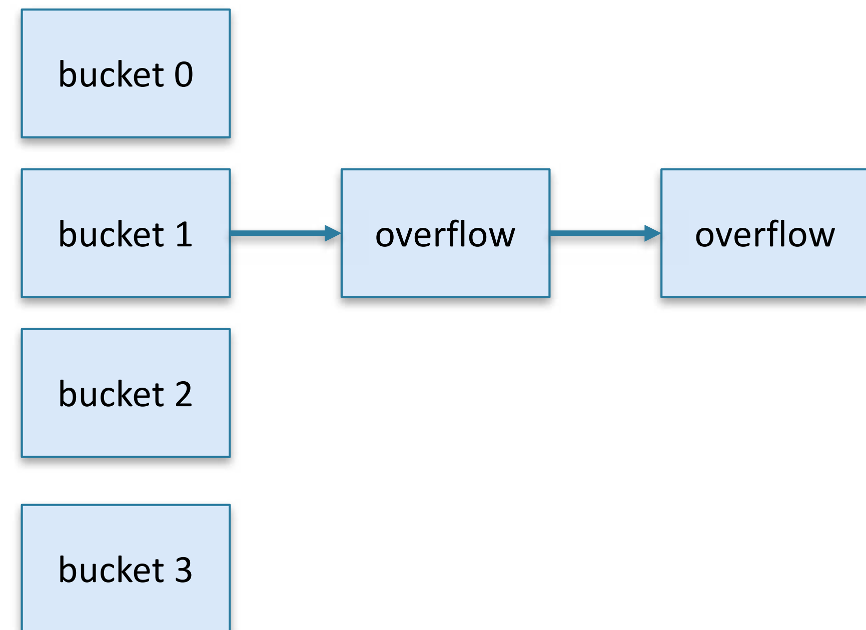
- Man nimmt an, dass die Zahl der benutzten Schlüssel K viel kleiner ist als die Domäne S. Abbildung h muss also nur die benutzten Schlüssel in K ordentlich abbilden.
- Gefahr von Kollisionen
- Möglichkeiten, die Hashfunktion zu wählen
  - $h(k) = k \; mod \; q$, q bestimmt Größe des Adressbereichs
  - $h(k) = k^2$ oder $h(k) = c * k$, dann Auswahl gewisser Bitpositionen, um gültigen Adressbereich zu erhalten.
- Fortgeschrittene Hashmethoden (➔ DBS VL im Winter)
  - Diese Verfahren sind sogenannte statische Verfahren
  - Es gibt aber auch dynamische Verfahren, die die Hashfunktion an den Bedarf anpassen (Erweiterbares Hashing).

Statischer Hash Index: Beispiel



Statischer Hash Index

- Suche (Lookup)
  - Ein Zugriff auf auf das Verzeichnis (directory)
  - Ein Zugriff auf die eigentliche Datei
- Performance hängt von Wahl der Hash-Funktion ab
- Überlauf von Buckets
  - Zu viele verschiedene Schlüssel-Werte werden auf gleichen Bucket abgebildet
  - Lösung: Überlaufbehandlung mittels Verkettung von Überlauf-Buckets



Indexe in Postgres: B+ Baum

```
CREATE TABLE MeineTabelle (
        ID int,
        major int,
        minor int,
        name varchar
);
```

- B+ Baum mit Schlüssel ID

  ```
  CREATE INDEX MeineTable_i1 ON MeineTabelle ID;
  ```

- B+ Baum mit Schlüssel (major, minor)

  ```
  CREATE INDEX MeineTable_i2 ON MeineTabelle (major, minor);
  ```

Indexe über mehrere Spalten

- Sogenannte Composite-Key Indexe.
  - Angabe einer Reihe (Achtung! Reihenfolge ist wichtig!) von Spalten.

```
CREATE INDEX indexName ON MeineTabelle (att1 , att2, att3);
```

- Tupel werden dann im Index sortiert anhand dieser Attribute
  - "Lexikographische Ordnung": att1 ist primäres Kriterium, gefolgt von att2, etc.
  - Optional mit Ordnung ASC oder DESC der einzelnen Attribute. Default ist ASC.

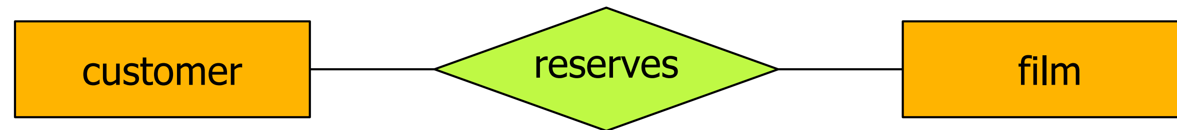
```
CREATE INDEX indexName
        ON MeineTabelle (att1 DESC , att2, att3);
```

Indexe in Postgres: Hash

```
CREATE INDEX name ON table USING hash (column);
```

- Ein Hash-Index macht Sinn, falls:
  - Keine Bereichsanfragen benötigt werden
  - Order by (Schlüssel) nicht benötigt wird
  - Keine oder nur kleine Joins über den Schlüsseln notwendig sind

Beispiel - Datenbank



- customer (customerID, name, street, city, state)
- film (filmID, title, kind, rentalPrice)
- reserved (customerID, filmID, resDate)

Verschiedene Selektionen

- Primärschlüssel, Punktanfrage

$$\sigma_{filmID=2}(film)$$

- Punktanfrage

$$\sigma_{title='Terminator'}(film)$$

- Bereichsanfrage

$$\sigma_{1<rentalPrice<4}(film)$$

- Konjunktion (d.h. logisches und)

$$\sigma_{kind='F' \wedge rentalPrice=4}(film)$$

- Disjunktion (d.h. logisches oder)

$$\sigma_{rentalPrice<2 \vee kind='D'}(film)$$

Ziel

Ersetze die Blätter des Anfrageplans durch spezifische Zugriffs-Methoden, d.h. kann/soll ich einen Index benutzen oder besser einen sequenziellen Scan der Datei?

Strategien für konjunktive Anfragen

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
    AND state = 'Arizona'
```

- Können die Indexe auf (name) und (street) benutzt werden?
- Kann der Index auf (name, street, state) benutzt werden?
- Kann der Index auf (name, street) benutzt werden?
- Kann der Index auf (name, street, city) benutzt werden?
- Kann der Index auf (city, name, street) benutzt?

Strategien für konjunktive Anfragen

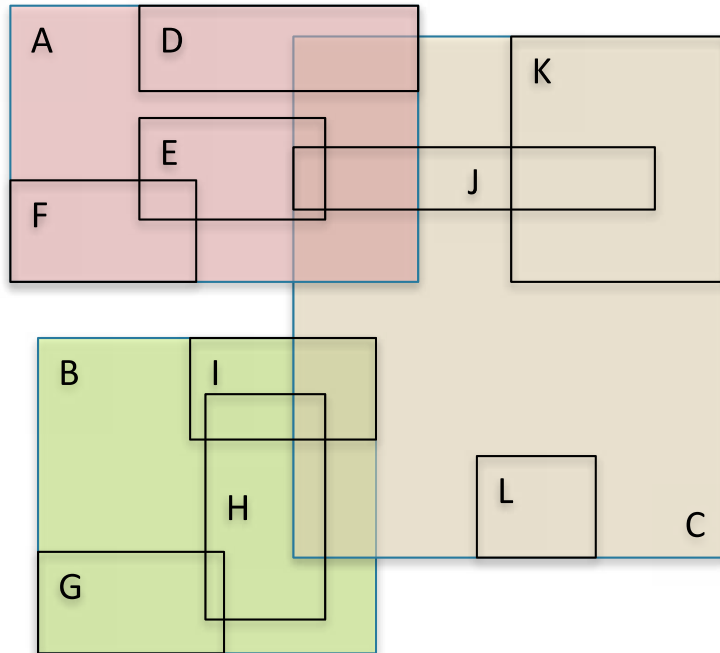
```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
    AND state = 'Arizona'
```

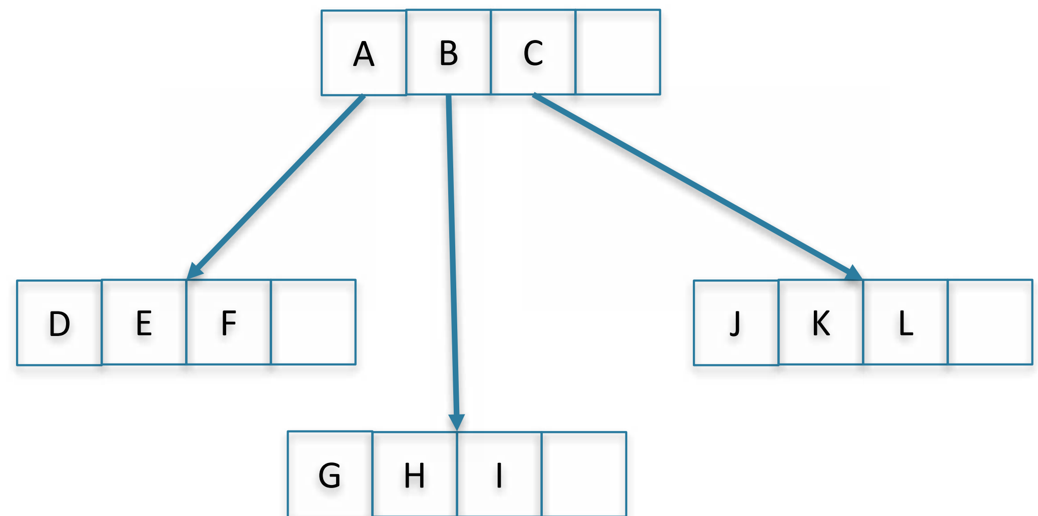
- Können die Indexe auf (name) und (street) benutzt werden? Ja
- Kann der Index auf (name, street, state) benutzt werden? Ja
- Kann der Index auf (name, street) benutzt werden? Ja
- Kann der Index auf (name, street, city) benutzt werden? Ja
- Kann der Index auf (city, name, street) benutzt? Nein

Mehrdimensionale Indexstrukturen

- Bislang: Eindimensionale Schlüssel.
- In vielen Anwendungen ist die Anzahl der Dimensionen höher, was tun?
- R-Baum
  - R steht für Rechteck/Rectangle
  - Knoten definieren minimale Rechtecke, die die enhaltenen Rechtecke umschließen
  - Balanciert. Aufbau (Algorithmus) ähnlich zum B+ Baum
  - Überlappung der MBRs (=Minimum Bounding Rectangles)
  - Überlappung führt zu Ineffizienz während der Anfrage.

R-Baum - Beispiel





Nachteile von Indexen

- Platzverbrauch
- Jedes insert/delete/update muss in den Indexen nachgepflegt werden
- Jede Änderungsoperation kostet etwas: CPU und/oder I/O
- Menge der Änderungsoperation kann soviel kosten, dass diese Kosten den Nutzen des Index überwiegen

Grundregel: je mehr Leseoperationen desto mehr lohnen sich Indexe.

Genauere Regeln zum"Index Tuning", Anwendbarkeit und weitere Details zur physischen Organisation von Tupeln bzw. Referenzen auf Tupel in Indexen werden in der VL Datenbanksysteme vorgestellt.