Zusammenfassung

- JPA als Mittler zwischen objektorientierter und relationaler Technologie
  - Überwindung des O/R Impedance Mismatch
- JPA de facto Standard für Java Persistenz
- JPQL als Anfragesprache
  - Verwendung des Java-Klassenmodells (und nicht der Relationen des DBMS)

- Vorteil:
  - Keine Unterschiede der DBMS auf der Anwendungsebene
  - Vermittlungssoftware übernimmt die Abbildung auf das jeweilige System.

- Nachteil
  - JPA erstellt Modell auf der konzeptionellen Ebene, aber ein physisches Datenbankdesign ist dadurch i. A. nicht möglich.

Diskussion der Ansätze

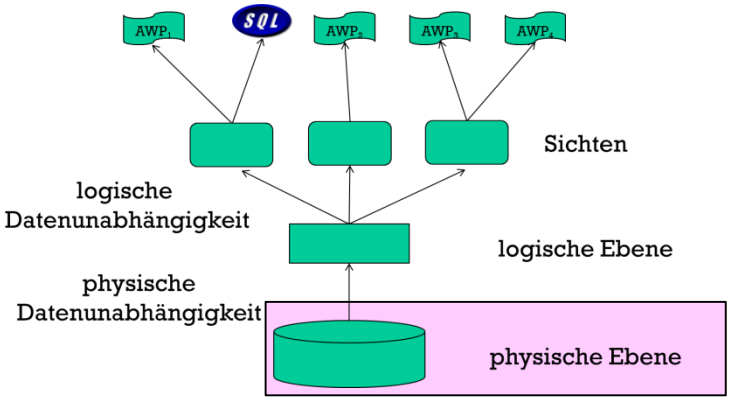
- **Vielfältige Kopplungsmöglichkeiten zwischen SQL und Programmiersprachen**
  - Erweiterungen von Programmiersprachen
    - Bibliothek
      - JDBC in Java
    - Spracherweiterungen
      - LINQ in C#
  - Erweiterungen von SQL
    - PL/SQL → Standard SQL/PSM
    - Serverseitige Prozeduren
  - JPA
    - Vermittlungsschicht zwischen Java und Datenbanken
    - Annotation von Java-Klassen

Literatur

- Zemke F, Hammerschmidt B, Kulkarni K, Liu Z, McMahon D, Melton J, Michels J, Özcan F, Pirahesh H (2014) ANSI SQL/JSON: part 1. https://www.wiscorp.com/pub/DM32.2-2014-00024R1_JSON-SQL-Proposal-1.pdf.
- Zemke F, Hammerschmidt B, Kulkarni K, Liu Z, McMahon D, Melton J, Michels J, Özcan F, Pirahesh H (2014) ANSI SQL/JSON: part 2 : querying JSON. www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-2.pdf.

7. Physischer Datenbankentwurf: Speicher- und Indexstrukturen

- **Bisher**
  - Entwurf der logischen (konzeptionellen) Ebene
    - Welche Relationen sollen im Datenbanksystem angelegt werden?



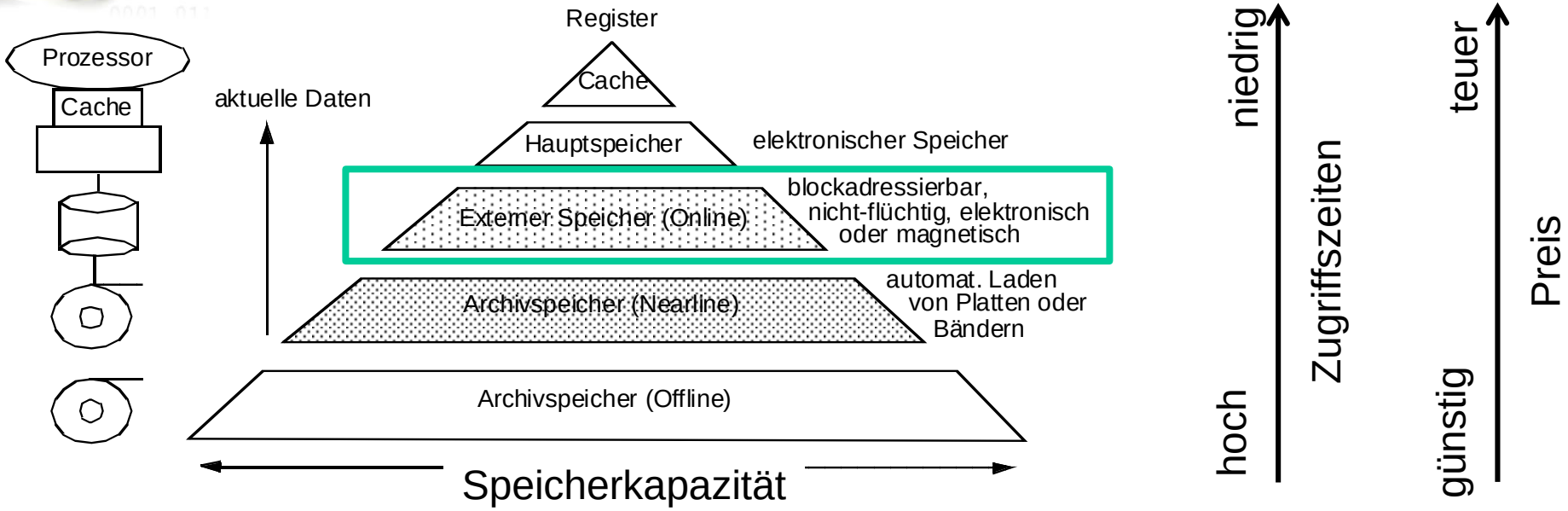
- **Jetzt**
  - Welche Möglichkeiten bietet ein DBS auf der physischen Ebene?
    - Wie werden Relationen auf die physische Ebene abgebildet?

7.1 Speichersystem

- **Wichtige Komponente für die Zuordnung der Datenobjekte zu physischen Speicher**

- **Idealer Speicher besitzt folgende Eigenschaften:**
  - Nicht-funktionale Eigenschaften
    - nahezu **unbegrenzte Speicherkapazität**
    - **kurze Zugriffszeit** bei wahlfreiem Zugriff
    - **niedrige Zugriffskosten** (Kosten/Zugriff/Sekunde möglichst niedrig)
    - **geringe Speicherkosten** (Kosten pro GB möglichst niedrig)
  - Funktionale Eigenschaften
    - nichtflüchtig
    - Unterstützung logischer und arithmetischer Verknüpfungen

Speicherhierarchie



Approximation der Eigenschaften eines idealen Speichers durch eine Hierarchie

- Ausnutzung von Lokalität auf den Ebenen.
  - Allokation von Daten mit hoher Zugriffswahrscheinlichkeit im schnellen (relativ teurerem) Speicher → kurze Zugriffszeiten
  - Verwaltung der wenig verwendeten Daten im langsamen Speicher → günstige Speicherkosten

Preis, Kapazität und Zugriffszeit

- **Prozessoren**
  - L1-Cache: je Kern 64 + 64 KB (Daten + Instruktionen)
  - L2-Cache: je Kern 512 KB mit Prozessortakt
  - L3-Cache: 6144 KB
- **Hauptspeicher (DDR3-1600)**
  - Latenz: 10 ns
  - Transferrate: 12,8 GB/s
  - Preis: 10 Euro/GB
- **Magnetplattenspeicher**
  - Zugriffszeit: 9 ms
  - Transferrate: 300 MB/s
  - Preis: 0.1 Euro/GB
- **Solid State Disks**
  - Zugriffszeit: 80 µs
  - Transferrate: 500 MB/s
  - Preis: 1 Euro/GB

➔ **Laufzeit der Algorithmen werden im Wesentlichen durch die Anzahl der Zugriffe auf den Externspeicher bestimmt.**

Verwaltungsaufgaben auf jeder Ebene der Speicherhierarchie

- Lokalisieren der Datenobjekte
- Allokation und Freigabe von Speicherplatz
- Ersetzung
- Schreibstrategie (write-through vs. write-back)
- ggf. Anpassung an verschiedene Transfergranulate zwischen den Ebenen

7.2 Verwaltung von Daten im Externspeicher

- **Der größte Unterschied in der Speicherhierarchie ist zwischen**
  - Hauptspeicher und
  - Externspeicher (Festplatten und Solid State Disks).

- **Charakteristika von Festplatten**
  - Zugriffzeiten
    - langsamer Direktzugriff
    - Schneller Datentransfer
  - Kosten
    - Günstig
  - Persistenz

Seitenbasierte Organisation

- Verwaltung der Datensätze in größeren physischen Einheiten fester Größe → Seiten (Blöcke)
  - Transfer zwischen Externspeicher und Hauptspeicher erfolgt nur im Granulat von Seiten
- Eigenschaften von Seiten
  - Sie besitzen alle die gleiche Größe
    - Voreingestellte Seitengröße in PostrgeSQL: 8 KB
  - Seiten besitzen eine eindeutige Kennung
    - Jede Relation entspricht einem Array von Seiten auf dem Externspeicher.

- **Fragen**
  - Wie werden Datensätze auf Seiten abgebildet?
  - Was passiert mit Datensätzen, die größer sind als die Seitengröße?

Seitenbasierte Dateien

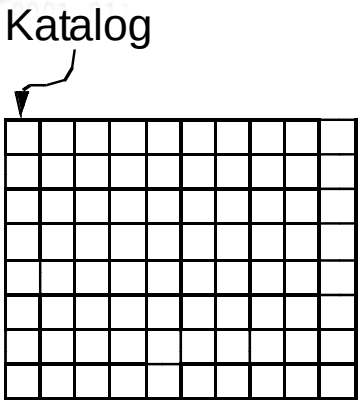
- **Relationen im Externspeicher = Folge von Seiten**
  - In PostgresSQL können sogenannte Tablespaces erzeugt werden, in denen die Seiten der Relationen verwaltet werden.
    - create tablespace erpspace location ’/data/dbs’;
    - create database ERP tablespace erpspace;

- **Im Speichersystem gibt es eine API, um direkt auf die Seiten der Dateien zuzugreifen.**

  → siehe Technische Informatik II
  - Lesen und Schreiben einzelner Seiten

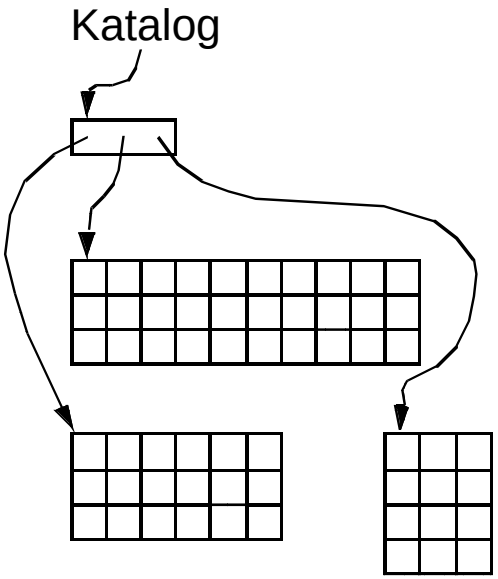
- **Dynamische Verwaltung der Seiten in Dateien**
  - Wie werden Dateien bei Bedarf vergrößert?

Seitenzuordnungsverfahren



Statische Datei-Zuordnung

- direkte Adressierung
- minimale Zugriffskosten
- keine Flexibilität

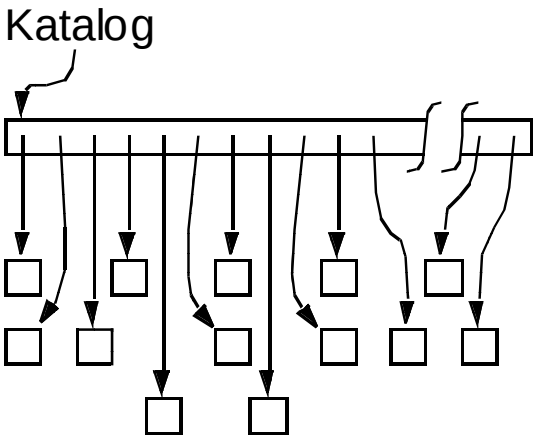


Dynamische Extent-Zuordnung

Adressierung über eine kleine Tabelle

geringe Zugriffskosten

moderate Flexibilität



Dynamische Block-Zuordnung

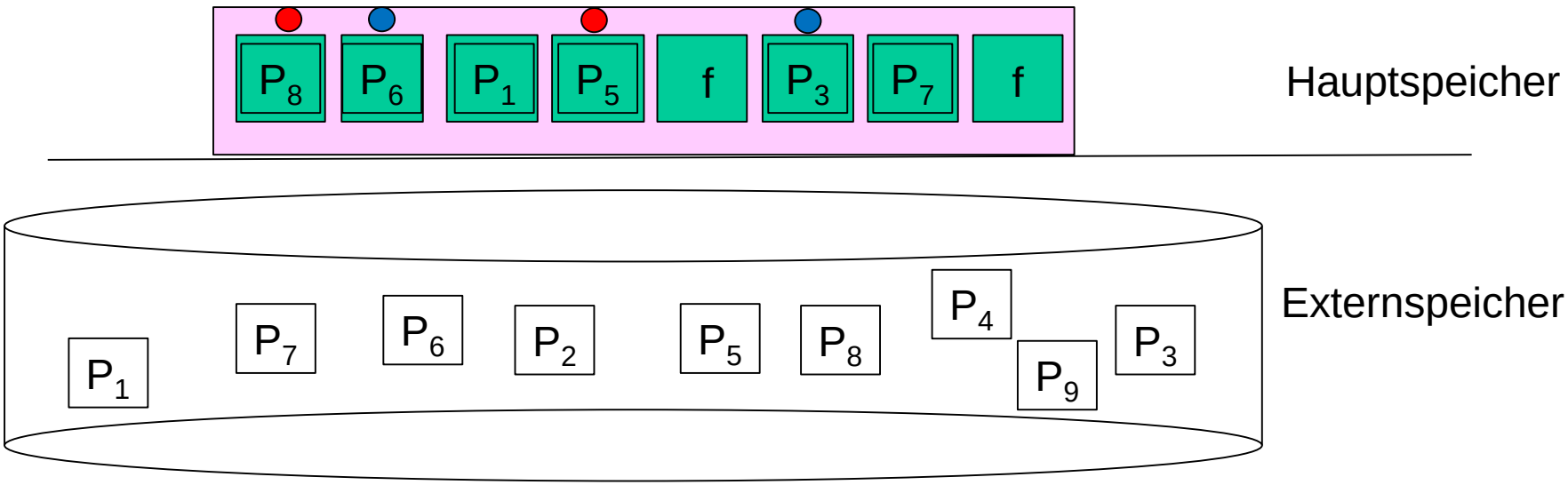
Adressierung über eine große Tabelle

hohe Zugriffskosten

maximale Flexibilität

Systempuffer

- **Motivation**
  - Vorhalten von oft benötigten Seiten in einem Puffer
    - Puffer ist ein vorreservierter Speicher von sogenannten Frames im Hauptspeicher.
    - Eine physische Seite passt genau in einen Frame.



Anforderung an die Pufferverwaltung

- **Aufgaben**
  - Prüft, ob Seite im Puffer liegt.
    - Wenn ja, in welchem Frame die Seite liegt.
  - Stellt einen freien Frame zur Verfügung, um eine Seite dort einzulagern.
    - Ggf. muss eine Seite aus einem Frame verdrängt werden.
  - Erkennt, ob eine Seite im Puffer derzeit benutzt wird.
    - Und wann die Seite nicht mehr in Benutzung ist.
  - Erkennt, ob eine Seite im Puffer geändert wurde.

Implementierungsmöglichkeiten

- Effiziente Suche im Puffer
  - ➔ Verwaltung der (Seiten, Frame)-Paare durch Hash-Map
- Suche nach freiem Frame im Puffer
  - ➔ Verkettung der freien Frames in einer Liste
- Bestimmen einer Seite (Opfer), die aus dem Puffer entfernt wird.
  - ➔ Least-Recently-Used
    - Verkettung der belegten Frames nach dem letzten Zeitpunkt der Nutzung der Seite
- Schreiben modifizierter Seiten
  - Zeitpunkt erfolgt in Absprache mit der Transaktionsverwaltung (ACID-Bedingungen)

Puffer in PostgreSQL

- **Einstellen und Lesen der Puffergröße (und viele andere Parameter)**
  - Direkt durch Ändern der Konfigurationsdatei posgresql.conf
  - Alternative (über SQL)
    - Lesen
      - SELECT * FROM pg_settings
        WHERE name = 'shared_buffers';
    - Ändern durch ALTER-SYSTEM-Befehl

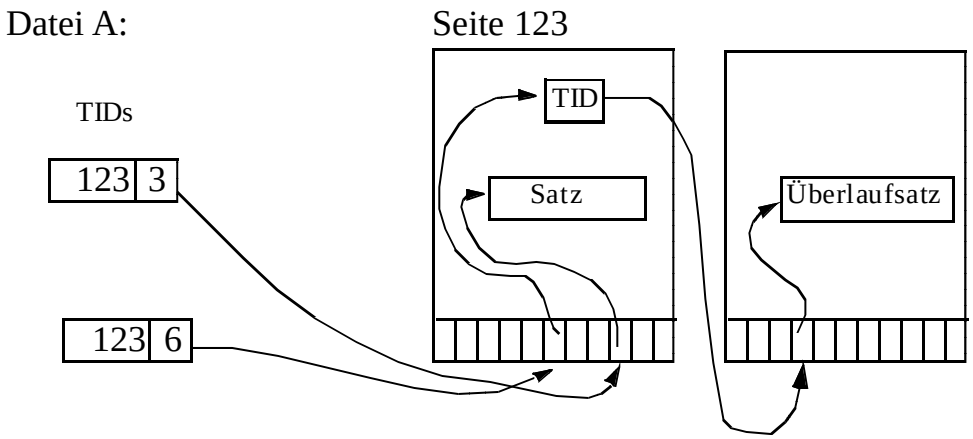
7.3 Zugriffssystem

- **Motivation**
  - Abbildung der Datensätze einer Relation auf die Seiten

- **Tuple-Identifier (TID auch RowID und RID genannt)**
  - TID ist eine eindeutige Kennung des Datensatzes innerhalb der Relation/Datenbank.
    - TID setzt sich zusammen aus der Seitenadresse und einer relativen Adresse innerhalb der Seite.
    - In PostgreSQL gibt es in jeder Relation das Attribut **ctid**.
      - Dies wird nicht bei select * from … ausgegeben, sondern muss explizit in der select-Klausel genannt werden.

Stabile TIDs

- **Anwender können TIDs nutzen**
  - Anforderung
    - TID (tuple identifier) eines Datensatzes soll sich nicht ändern. Man spricht dann von stabilen TIDs.
- **Migration eines Satzes in andere Seite ohne TID-Änderung**
  - Einrichten eines Stellvertreter-TID in der Primärseite
    - Überlaufkette: Länge ≤ 1



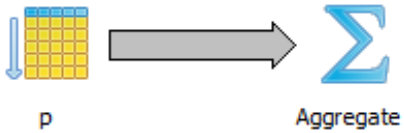
Recordmanager

- Komponente zur Verwaltung der Datensätze (in Seiten)
- Zentrale Aufgabe des Recordmanagers
  - Suche nach einer Seite zur Speicherung eines neuen Datensatzes.
    - Ggf. muss hierfür eine neue Seite angefordert werden.
  - Wünschenswert : Clusterung der Datensätze:
    - Datensätze, die oft gemeinsam zugegriffen werden, sollen auch gemeinsam in einer Seite liegen.
- **Lösungen**
  - Datensätze mit konstanter Länge

    ➔ z. B. Verkettung der Seiten, die noch Platz haben.
  - Datensätze mit variabler Länge
    - relativ kompliziert (→ siehe Datenbanksysteme II)

Zugriff auf Tupel einer Relation

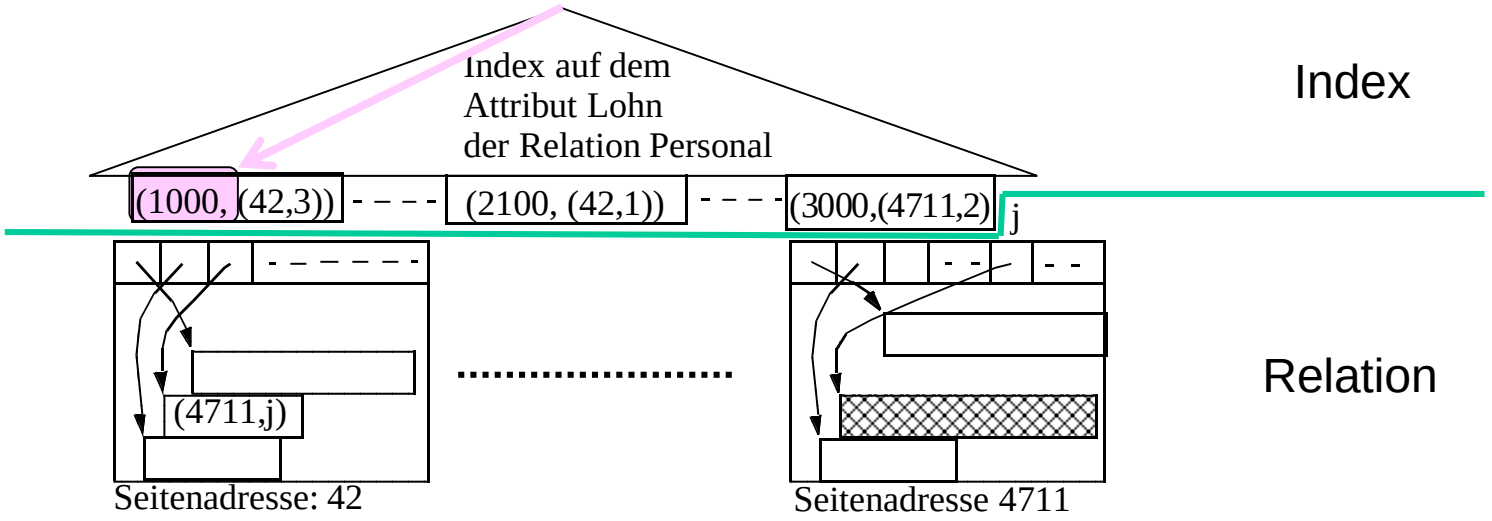
- Zwei Zugriffsvarianten
  - **Relationen-Scan**
    - Durchlaufen der zu der Relation gehörenden Seiten.
    - Beispiel
      select count(*) from p where gehalt > 20000;



  - **Index-Scan**
    - Zugriff erfolgt indirekt über einen Index
      - Indexe müssen vorher angelegt sein!
        - Verwaltung der TID zusammen mit einem oder mehreren Attributen
    - Beispiel
      select count(*) from p where gehalt > 90000;

Beispiel

- **Suche nach dem Datensatz mit Schlüssel 1000**

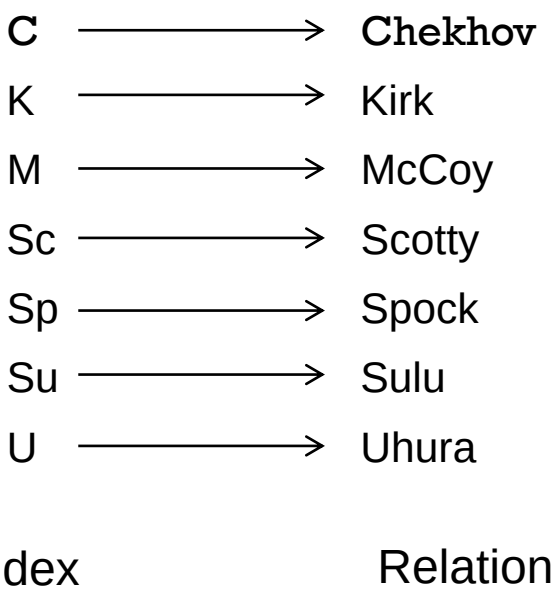


Welcher Scan wird benutzt?

- **Diese Entscheidung erfolgt durch den Optimierer eines Datenbanksystems in Abhängigkeit der Selektivität.**
  - Hohe Selektivität = Zugriff auf wenige Datensätze
    - ➔ Index-Scan
  - Niedrige Selektivität = Zugriff auf viele Datensätze
    - ➔ Relationen-Scan
- **In PostgreSQL gibt es die Möglichkeit die Verwendung eines Scans zu verbieten.**
  - set enable_seqscan= false;

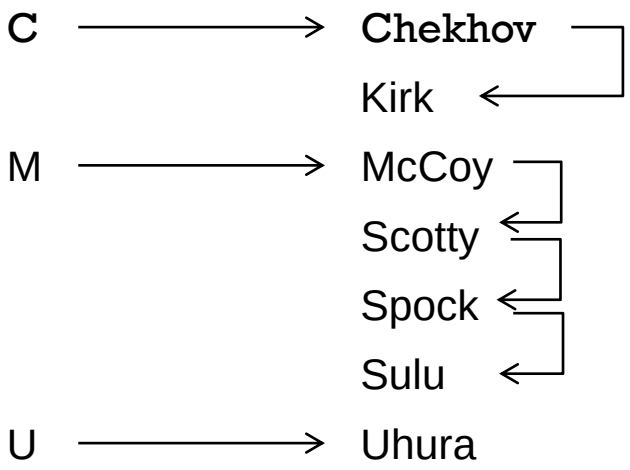
Indexe

- **Die Default-Implementierung in DBMS für ein Index ist der B+-Baum (Details siehe unten).**
- **Ein Primärindex bzw. Clusterindex wird auf einer sortierten Relation angelegt.**
  - Ein Primär- bzw. Clusterindex pro Relation
    - Beide Varianten folgen der Ordnung der Daten.



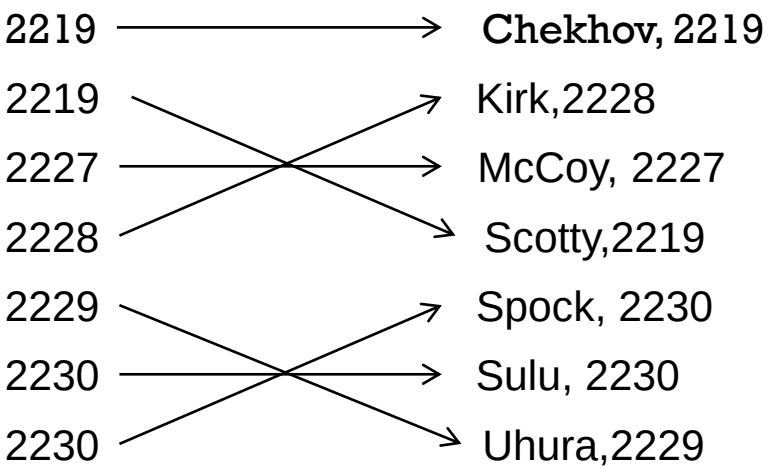
Dichte und dünne Indexe

- **Man spricht von einem dichten Primärindex, wenn für jeden Datensatz ein Indexeintrag existiert.**
  - Ansonsten spricht man von einem dünnen Index.



Sekundärindex

- **Daten werden indexiert auf einem (oder mehreren) Attributen ohne Schlüsseleigenschaft.**
  - Nehmen wir an, dass ein Index auf dem Geburtsjahr der Personen angelegt wurde.



7.4 Indexstrukturen

- **Ziele**
  - Effizienter Zugriff auf die Datensätze einer Relation, die ein bestimmtes Suchprädikat erfüllen.
  - Strukturen sollen keinen erheblichen Mehraufwand verursachen.
    - Änderungsoperationen
    - Speicherplatz

- **Klassifizierung**
  - Eindimensionale Prädikate (bzgl. einem Attribut)
    - Exakte Prädikate: Dynamische Hash-Verfahren
    - Bereichsprädikate und exakte Prädikate: **B^+- Bäume**
    - Komplexere Prädikate → siehe Datenbanksysteme II

B+-Bäume

- **Originalarbeit**
  - Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices Acta Inf. 1: 173-189 (1972)

- **Gegensatz zu binären Suchbäumen**
  - Entwurf für die Verwaltung von Daten im **Externspeicher**
    - ➔ **Ziel: Minimierung der Seitenzugriffe**
      - viele Einträge/Sätze in einem Knoten
      - alle Daten liegen in den Blattknoten

- **Gegensatz zu ISAM**
  - ISAM ist eine statische Indexstruktur für den Externspeicher
    ➔ Periodische Reorganisationen erforderlich
  - B+-Bäume sind voll dynamisch
    ➔ Anpassen der Struktur beim Einfügen/Löschen eines Datensatzes

Binäre Suchbäume

- **Zusammenfassung der wichtigsten Resultate**
  - n = Anzahl der Datensätze.
  - **Minimale Höhe** eines binären Baums: $\log_2(n+1)$
  - Es gibt binäre Suchbäume
    - z. B. AVL-Bäume, Rot-Schwarz-Bäume

    mit folgendem Leistungsverhalten **im schlechtesten Fall**:
    - Linearer Speicherplatzbedarf: O(n)
    - Logarithmische Höhe: O(log n)
    - Kosten für exakte Suche, Einfügen und Löschen: O(log n)
    - Kosten für Bereichssuche

      O(log n + r)

      wobei r die Anzahl der Antworten ist.
- **Kann dieses Resultat für den Externspeicher verallgemeinert werden?**
  - Kostenmaß = Anzahl der Externspeicherzugriffe

Erster Ansatz

- **Jeder binäre Knoten eines Suchbaums in einer Seite auf dem Externspeicher**

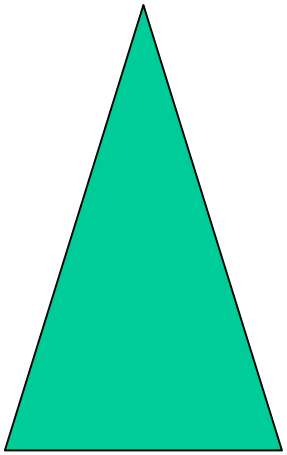
- **Probleme**
  - Direkte Abbildung von binären Knoten auf Seiten führt zu schlechten Strukturen.
    - im schlechtesten Fall
      - ein Knotenzugriff = ein Plattenzugriff
    - exakte Suche ist dann sehr teuer
      - z. B. für 10^7 Datensätze ist die Höhe bereits 23

  ➔ **Binäre Suchbäume sind also nicht für die Verwaltung von Daten auf dem Externspeicher geeignet.**

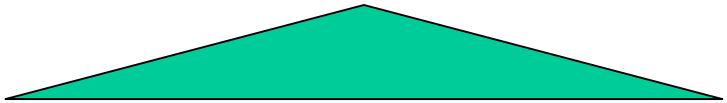
Idee

- **Erzeuge „fette“ Knoten mit möglichst vielen Einträgen**
  - Wähle Anzahl der Einträge maximal, so dass ein Knoten gerade noch in eine Seite passt.

  ➔ Erhebliche **Reduktion der Höhe** des Baums



Binärer Suchbaum



B+-Baum

Definition (B+-Baum)

- **Ein B^+-Baum vom Typ (b, c) ist ein Baum mit folgenden Eigenschaften:**
  - Jeder Weg von der Wurzel zum Blatt hat die gleiche Länge.
  - Die Wurzel ist ein Blatt oder hat mindestens 2 und höchstens 2b-1 Kinder.
  - Jeder Zwischenknoten hat mindestens b und höchstens 2b-1 Kinder.
  - Jedes Blatt hat mindestens c und höchstens 2c-1 Einträge.

Knotentypen im B+-Baum

Zwischenknoten

- p_i = Zeiger auf Kindknoten, k_i = Trennschlüssel
- Es gilt stets: $k_i < k_{i+1}$ für $0 < i < m$.

p_0	k_1	p_1	k_2	p_2	• • •	k_m	p_m	frei

Blattknoten

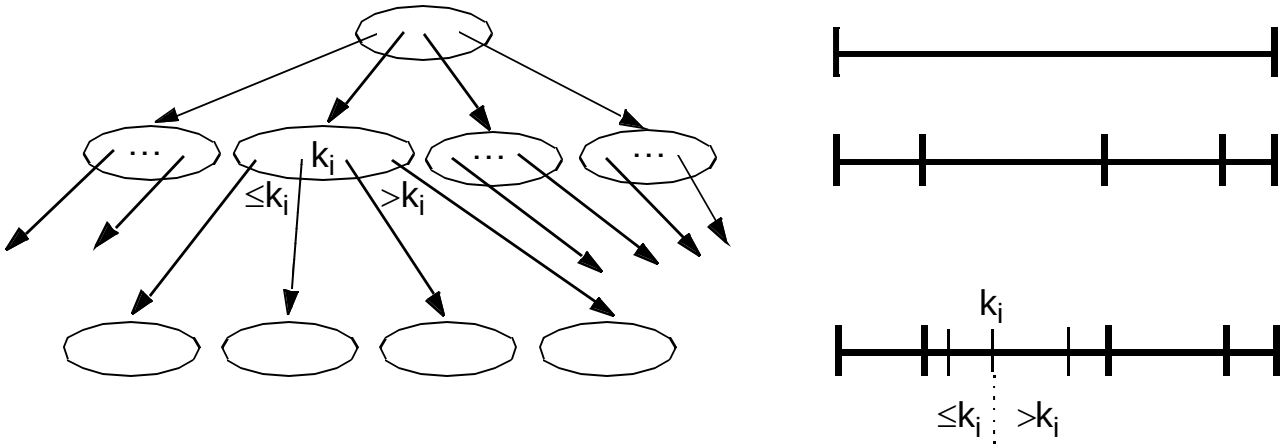
- TID_i = Verweis auf den Datensatz in der Relation mit Attributwert k_i
- Ein Paar **(k_i, TDI_i)** wird als Eintrag im Blatt bezeichnet.
- N = Zeiger auf den rechten Nachbarblattknoten
- V = Zeiger auf den linken Nachbarblattknoten

V	k_1	TID_1	k_2	TID_2	• • •	k_m	TID_m	frei	N

Suchbaumeigenschaft des B+-Baums

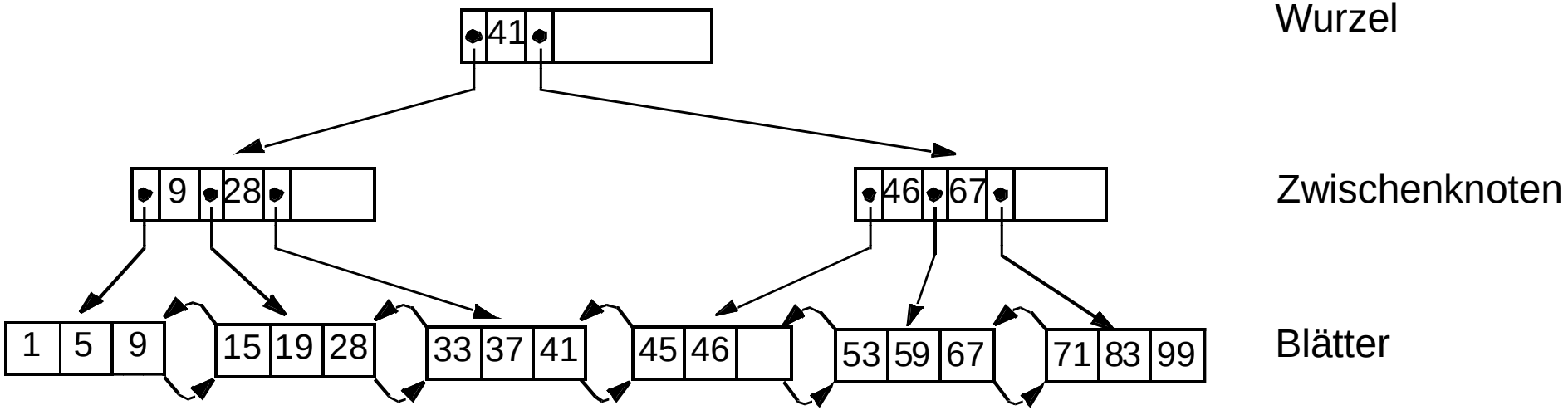
- **Lokale Ordnungserhaltung**
  - Für jeden Zwischenknoten Z mit j Trennschlüsseln $k_1,\ldots,k_j$ und (j+1) Verweisen $p_0,\ldots,p_j$ auf Kindknoten gilt:

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



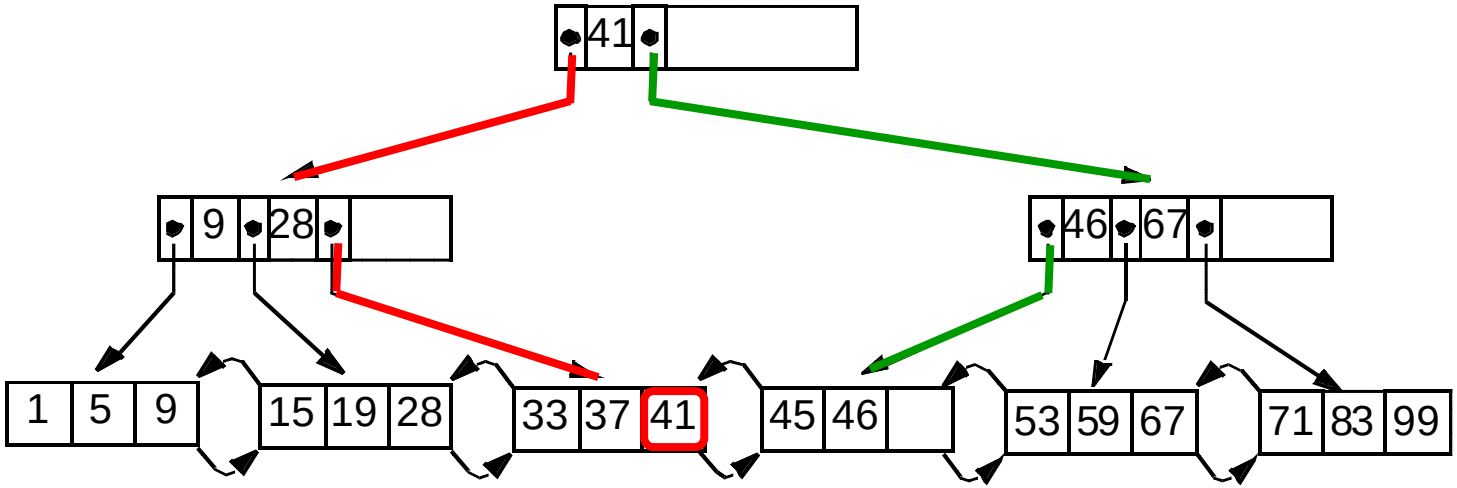
Beispiel

- **b=c=2, n = 17**
  - Beachte, dass b und c nur aus Gründen der Übersicht so klein gewählt wurden.
  - Die TIDs in den Blättern des B+-Baums wurden weggelassen.



Exakte Suche (Beispiel)

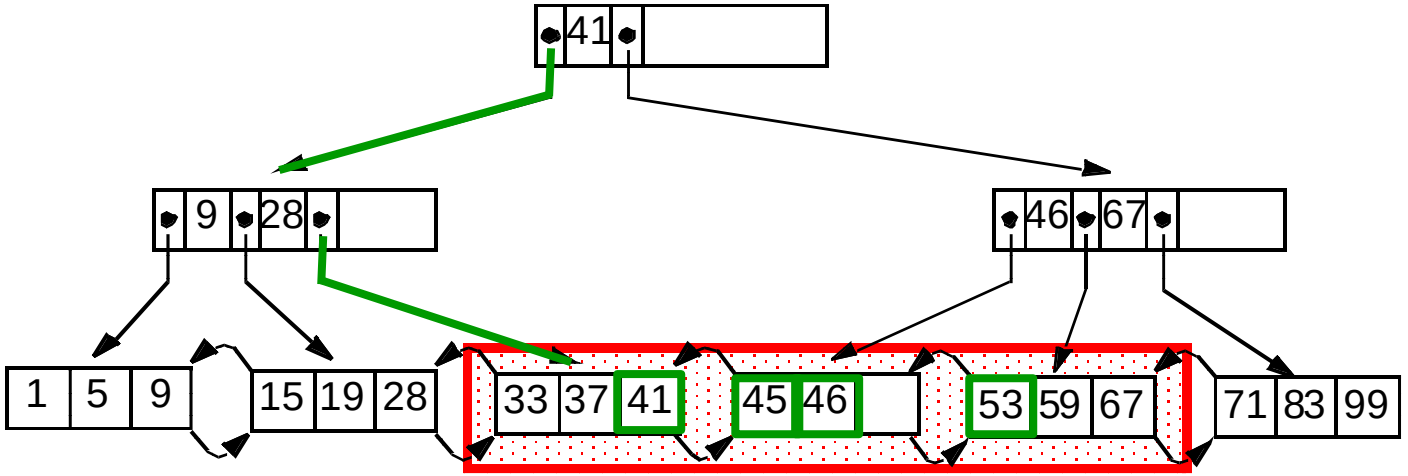
- **Vereinfachende Annahme**
  - Es wird nur als Ergebnis geliefert, ob der Datensatz im Baum ist.
    - Suche den Datensatz mit Attributwert 42.
    - Suche den Datensatz mit Attributwert 41



- **Wichtige Eigenschaft**
  - Suche ist auf einen Pfad beschränkt.
    ➔ Kosten O(h), h = Höhe des Baums

Bereichssuche (Beispiel)

- **Suche alle Datensätze im Bereich [40, 54]**



- **Algorithmus**
  - Suche durchläuft den Suchpfad bis zum Blatt, in dem die **linke Bereichsgrenze** liegt oder liegen könnte.
    - ➔ O(h) Knotenzugriffe
  - Danach folgt man den Blättern bis zu dem Blatt, das ein Attributwert größer als die **rechte Bereichsgrenze** enthält.
    - ➔ O(r/c) Blätter werden besucht, r = Anzahl der Antworten

Einfügen im B+-Baum

- **Top-Down/Bottom-Up Algorithmus**

  Eingabe ein Datensatz <k, TID>

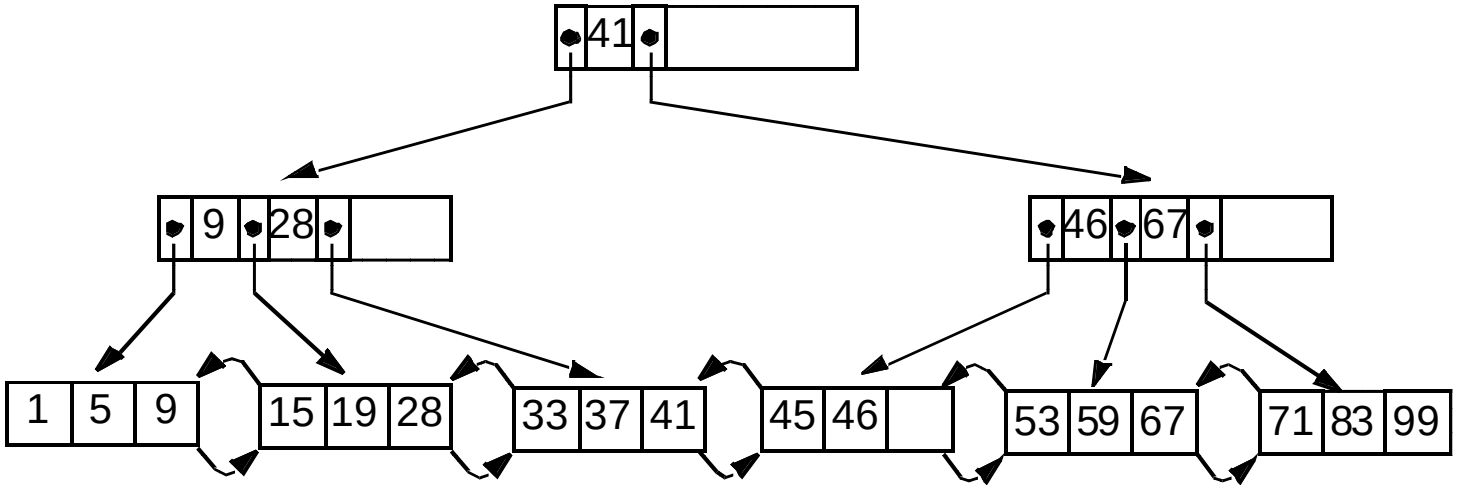
  - Suche den Attributwert k und füge <k, TID> in das zugehörige Blatt ein.
  - Falls (das Blatt 2c Datensätze enthält)
    - Aufspalten des Knotens in zwei Knoten mit jeweils der Hälfte der Datensätze / Indexeinträge.
    - Falls der Knoten nicht die Wurzel ist:
      - Füge neuen Eintrag in den Elternknoten ein und behebe den Überlauf ggf. rekursiv (durch Zurücklaufen des Suchpfads).

    Ansonsten
      - Erzeuge neue Wurzel mit zwei Verweisen und einem Trennschlüssel.

- **Kosten**
  - O(h), h = Höhe des Baums

Beispiel

- Einfügen des Datensatzes 44
- Einfügen des Datensatzes 25



Löschen im B+-Baum

- **Klassischer Top-Down/Bottom-Up Algorithmus**

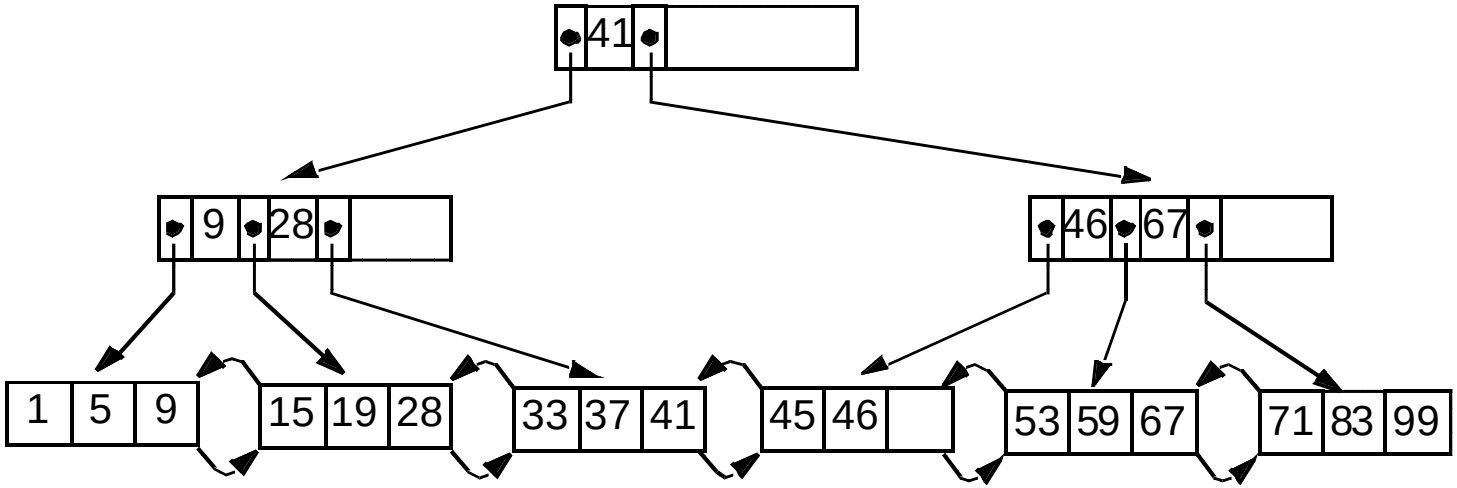
  Eingabe ein Attributwert k

  - Suche den Datensatz t mit Attributwert k und lösche t aus dem Blatt.
  - Falls der Knoten zu wenige Datensätze / Indexeinträge besitzt
    - Falls der rechte Nachbarknoten zu mehr als die Hälfte gefüllt ist,
      - Behebe den Unterlauf durch Verschieben der Datensätze / Indexeinträge aus dem rechten Nachbarknoten.
      - Passe den Indexeintrag im Elternknoten an.
    - Falls der rechte Knoten nur zur Hälfte gefüllt ist,
      - Verschmelze den Knoten mit seinem rechten Nachbarknoten.
      - Lösche Eintrag im Elternknoten und behebe den Unterlauf rekursiv.
    - Falls der Knoten die Wurzel ist (und n > 0)
      - Mache das verbleibende Kind zur neuen Wurzel

- **Problem**
  - JoJo-Effekt: kontinuierliches Aufspalten und Verschmelzen möglich
    - Verschmelze Seiten erst dann, wenn der Füllgrad eines Knotens unter 33% liegt. ➔ Keine asymptotische Laufzeitverschlechterung.

Beispiel

- Löschen des Datensatzes 19
- Löschen des Datensatzes 46



Leistung

- **Die Höhe h des B+-Baums $O(\log_b N)$.**
  - Beweisidee
    - Berechne die minimale Anzahl von Daten in einem B+-Baum der Höhe h.

➔ Kosten für Suchen, Einfügen und Löschen

- Exakte Suche, Einfügen und Löschen sind auf einen Pfad beschränkt
- Im schlechtesten Fall ergeben sich folgende Kosten für den B+-Baum:

exakte Suche:	$O(\log_b N)$
Bereichanfrage:	$O(\log_b N + r/b)$
Einfügen:	$O(\log_b N)$
Löschen:	$O(\log_b N)$

- **Speicherplatzausnutzung beträgt mindestens 50%**

Viele Varianten von B+-Bäumen

- **Optimierungsmöglichkeiten**
  - Verbesserung des Verzweigungsgrads
  - Schlüsselkomprimierung
  - Verbesserung des Belegungsgrades durch verallgemeinerte Splittingverfahren
- **Varianten**
  - Präfix-B-Bäume
    - Kompression des Index
  - Parallele B+-Bäume
    - Verteilung der Daten über mehrere Knoten/Festplatten
  - Log-basierte B+-Bäume
    - Günstige Einfügekosten durch Verwalten mehrer Indexe verschiedener Größen.