

Datenbanken 2

Indexstrukturen

Nikolaus Augsten

`nikolaus.augsten@sbg.ac.at`

FB Computerwissenschaften
Universität Salzburg

Wintersemester 2014/15

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - Statisches Hashing
 - Dynamisches Hashing
 - Mehrschlüssel Indices
 - Indizes in SQL

Lektüre zum Thema “Indexstrukturen”:

- Kapitel 7 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 11 in Silberschatz, Korth, and Sudarashan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Inhalt

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - Statisches Hashing
 - Dynamisches Hashing
 - Mehrschlüssel Indices
 - Indizes in SQL

Grundlagen/1

- **Index beschleunigt Zugriff**, z.B.:
 - Autorenkatalog in Bibliothek
 - Index in einem Buch
- **Index-Datei** besteht aus Datensätzen: den Index-Einträgen
- **Index-Eintrag** hat die Form
(Suchschlüssel, Pointer)
 - *Suchschlüssel*: Attribut(menge) nach der Daten gesucht werden
 - *Pointer*: Pointer auf einen Datensatz (TID)
- **Suchschlüssel dürfen doppelt vorkommen**
(im Gegensatz zu Schlüsseln von Relationen)
- Index-Datei meist viel **kleiner** als die indizierte Daten-Datei

Grundlagen/2

- Merkmale des Index sind:
 - Zugriffszeit
 - Zeit für Einfügen
 - Zeit für Löschen
 - Speicherbedarf
 - effizient unterstützte Zugriffsarten
- Wichtigste Zugriffsarten sind:
 - Punktanfragen: z.B. Person mit SVN=1983-3920
 - Mehrpunktanfragen: z.B. Personen, die 1980 geboren wurden
 - Bereichsanfragen: z.B. Personen die mehr als 100.000 EUR verdienen

Grundlagen/3

Indextypen werden nach folgenden Kriterien unterschieden:

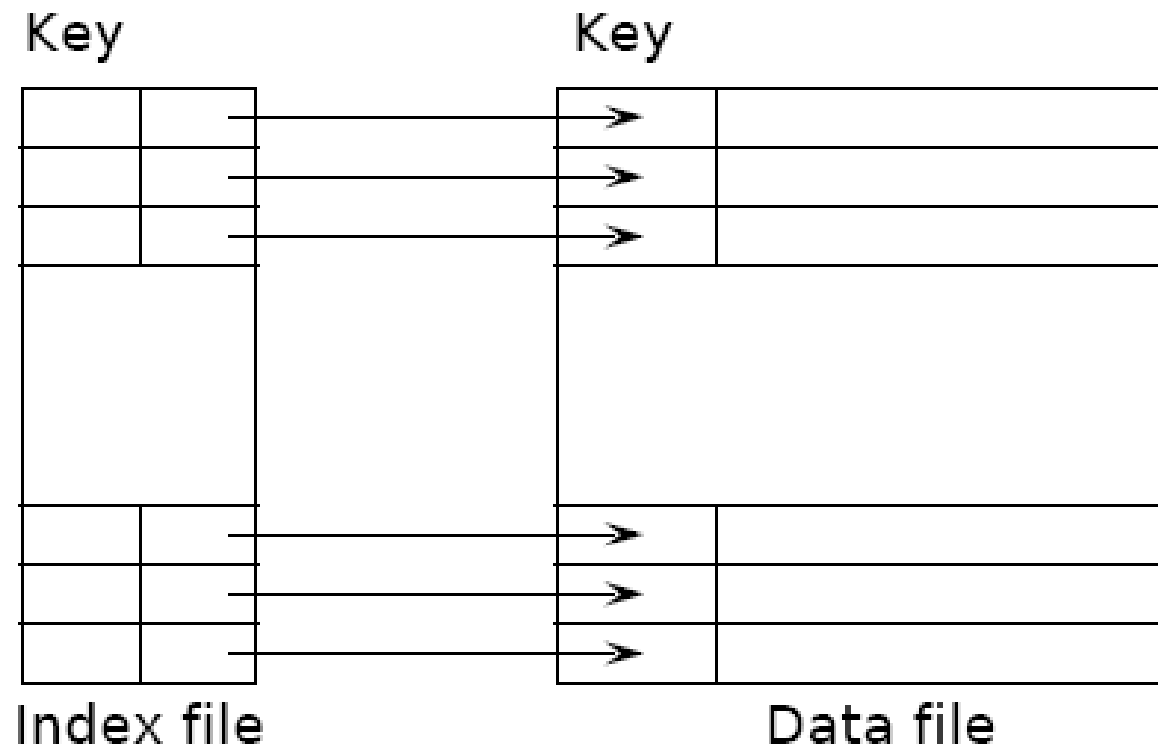
- Ordnung der Daten- und Index-Datei:
 - Primärindex
 - Clustered Index
 - Sekundärindex
- Art der Index-Einträgen:
 - sparse Index
 - dense Index

Nicht alle Kombinationen üblich/möglich:

- Primärindex ist oft sparse
- Sekundärindex ist immer dense

Primärindex/1

- Primärindex:
 - Datensätze in der Daten-Datei sind nach Suchschlüssel sortiert
 - Suchschlüssel ist eindeutig, d.h., Suche nach 1 Schlüssel ergibt 1 Tupel

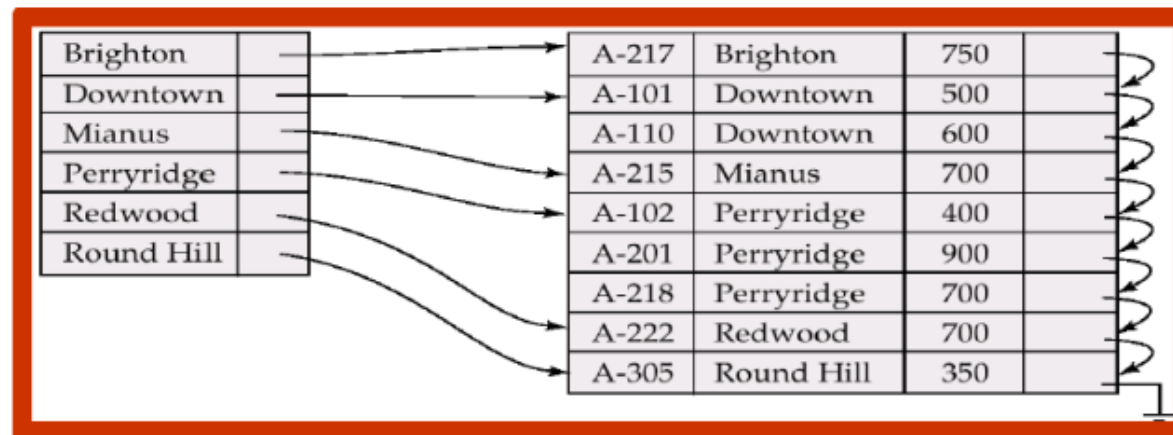


Primärindex/2

- Index-Datei:
 - sequentiell geordnet nach Suchschlüssel
- Daten-Datei:
 - sequentiell geordnet nach Suchschlüssel
 - jeder Suchschlüssel kommt nur 1 mal vor
- Effiziente Zugriffsarten:
 - Punkt- und Bereichsanfragen
 - nicht-sequentieller Zugriff (random access)
 - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

Clustered Index

- Index-Datei:
 - sequentiell geordnet nach Suchschlüssel
- Daten-Datei:
 - sequentiell geordnet nach Suchschlüssel
 - Suchschlüssel kann *mehrfach* vorkommen
- Effiziente Zugriffsarten:
 - Punkt-, Mehrpunkt-, und Bereichsanfragen
 - nicht-sequentieller Zugriff (random access)
 - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

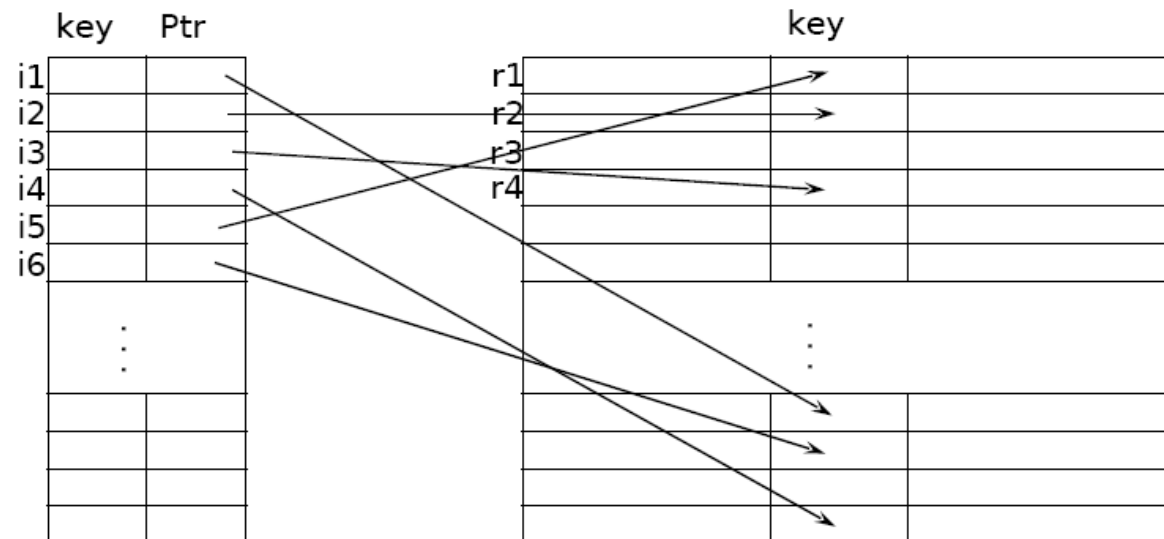


Sekundärindex/1

- Primär- vs. Sekundärindex:
 - nur 1 Primärindex (bzw. Clustered Index) möglich
 - beliebig viele Sekundärindizes
 - Sekundärindex für schnellen Zugriff auf alle Felder, die nicht Suchschlüssel des Primärindex sind
- **Beispiel:** Kontos mit Primärindex auf Kontonummer
 - Finde alle Konten einer bestimmten Filiale.
 - Finde alle Konten mit 1000 bis 1500 EUR Guthaben.
- **Ohne Index** können diese Anfragen **nur durch sequentielles Lesen** aller Knoten beantwortet werden – sehr langsam
- **Sekundärindex** für schnellen Zugriff **erforderlich**

Sekundärindex/2

- Index-Datei:
 - sequentiell nach Suchschlüssel geordnet
- Daten-Datei:
 - Suchschlüssel kann *mehrfach* vorkommen
 - *nicht* nach Suchschlüssel geordnet



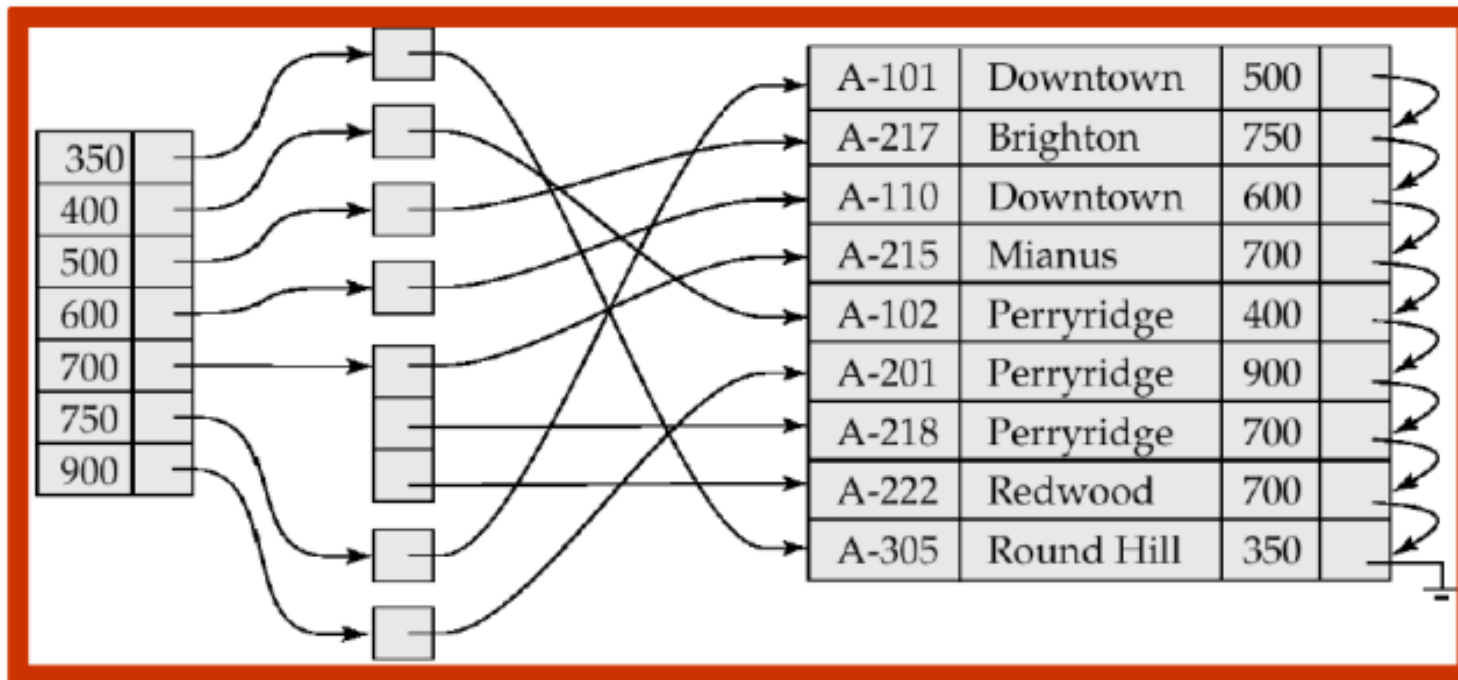
Sekundärindex/4

- Effiziente Zugriffsarten:
 - sehr schnell für Punktanfragen
 - Mehrpunkt- und Bereichsanfragen: gut wenn nur kleiner Teil der Tabelle zurückgeliefert wird (wenige %)
 - besonders für nicht-sequentiellen Zugriff (random access) geeignet

Sekundärindex/4

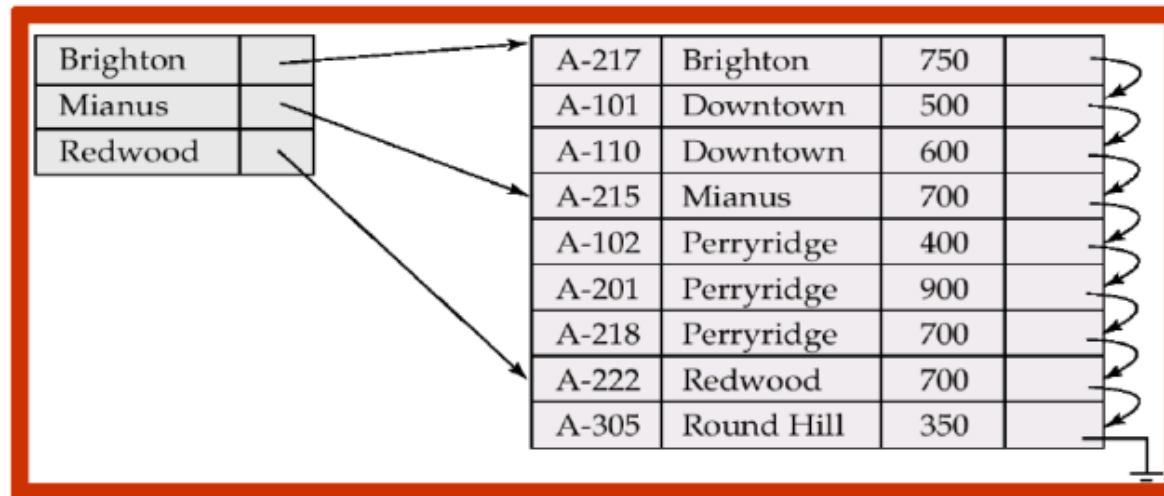
Zwei Möglichkeiten für Pointer:

- **Doppelte Indexeinträge:**
 - ein Indexeintrag für jeden Datensatz
- **Buckets:**
 - ein Indexeintrag für jeden Suchschlüssel
 - Index-Eintrag zeigt auf ein Bucket
 - Bucket enthält Pointer zu Datensätzen



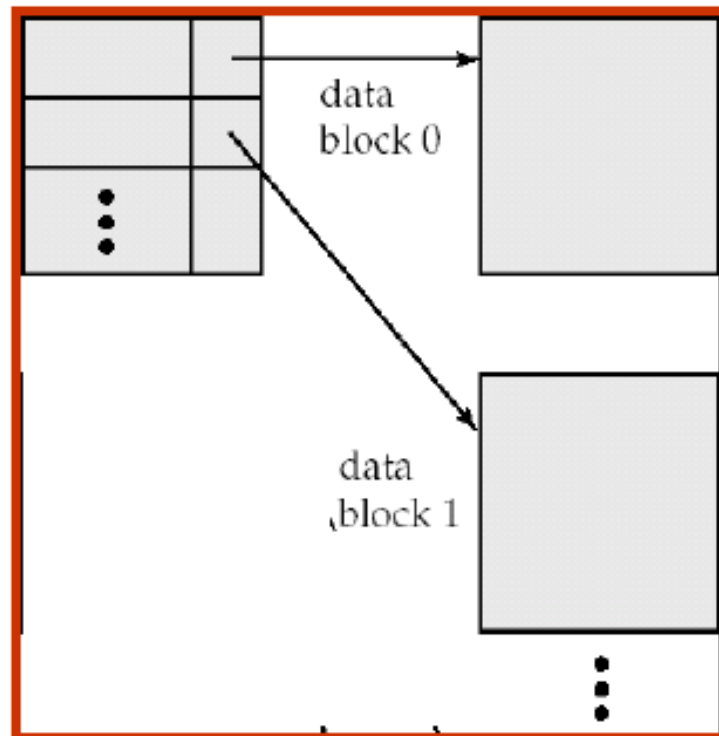
Sparse Index/1

- Sparse Index
 - ein Index-Eintrag für mehrere Datensätze
 - kleiner Index: weniger Index-Einträge als Datensätze
 - nur möglich wenn Datensätze nach Suchschlüssel geordnet sind (d.h. Primärindex oder Clustered Index)



Sparse Index/2

- Oft enthält ein sparse Index **einen Eintrag pro Block**.
- Der **Suchschlüssel**, der im Index für eine Block gespeichert wird, ist der **kleinste Schlüssel in diesem Block**.



Dense Index/1

- Dense Index:
 - Index-Eintrag (bzw. Pointer in Bucket) für **jeden Datensatz** in der Daten-Datei
 - dense Index kann groß werden (aber immer kleiner als Daten)
 - Handhabung einfacher, da ein Pointer pro Datensatz
- **Sekundärindex** ist immer dense

Gegenüberstellung von Index-Typen

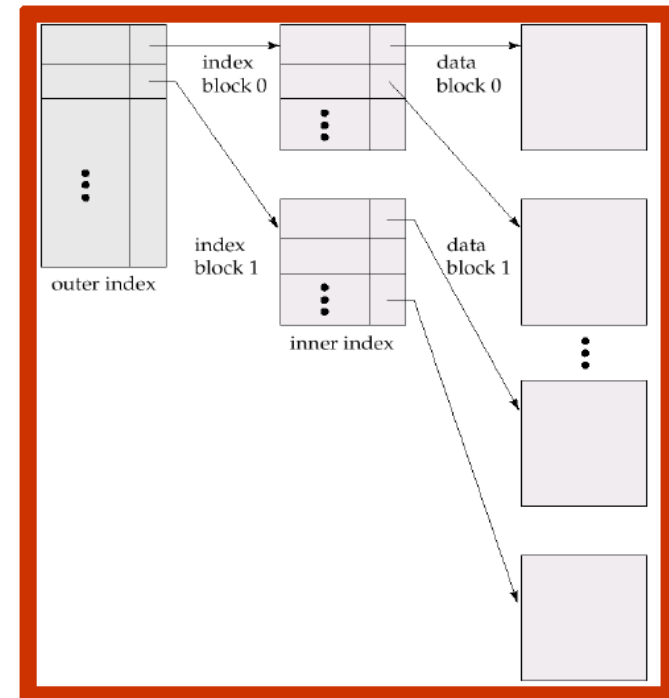
- Alle Index-Typen machen **Punkt-Anfragen** erheblich schneller.
- Index erzeugt **Kosten bei Updates**: Index muss auch aktualisiert werden.
- **Dense/Sparse** und **Primär/Sekundär**:
 - Primärindex kann dense oder sparse sein
 - Sekundärindex ist immer dense
- **Sortiert lesen** (=sequentielles Lesen nach Suchschlüssel-Ordnung):
 - mit Primärindex schnell
 - mit Sekundärindex teuer, da sich aufeinander folgende Datensätze auf unterschiedlichen Blocks befinden (können)
- **Dense vs. Sparse**:
 - sparse Index braucht weniger Platz
 - sparse Index hat geringere Kosten beim Aktualisieren
 - dense Index erlaubt bestimmte Anfragen zu beantworten, ohne dass Datensätze gelesen werden müssen (“covering index”)

Mehrstufiger Index/1

- Großer Index wird teuer:
 - Index passt nicht mehr in Hauptspeicher und mehrere Block-Lese-Operationen werden erforderlich
 - binäre Suche: $\lfloor \log_2 b \rfloor + 1$ Block-Lese-Operationen (Index mit b Blocks)
 - eventuelle Overflow Blocks müssen sequentiell gelesen werden
- Lösung: Mehrstufiger Index
 - Index wird selbst wieder indiziert
 - dabei wird der Index als sequentielle Daten-Datei behandelt

Mehrstufiger Index/2

- Mehrstufiger Index:
 - Innerer Index: Index auf Daten-Datei
 - Äußerer Index: Index auf Index-Datei
- Falls äußerer Index zu groß wird, kann eine **weitere Index-Ebene** eingefügt werden.
- Diese Art von (ein- oder mehrstufigem) Index wird auch als **ISAM** (Index Sequential Access Method) oder **index-sequentielle Datei** bezeichnet.



Mehrstufiger Index/3

- Index Suche
 - beginne beim Root-Knoten
 - finde alle passenden Einträge und verfolge die entsprechenden Pointer
 - wiederhole bis Pointer auf Datensatz zeigt (Blatt-Ebene)
- Index Update: Löschen und Einfügen
 - Indizes aller Ebenen müssen nachgeführt werden
 - Update startet beim innersten Index
 - Erweiterungen der Algorithmen für einstufige Indizes

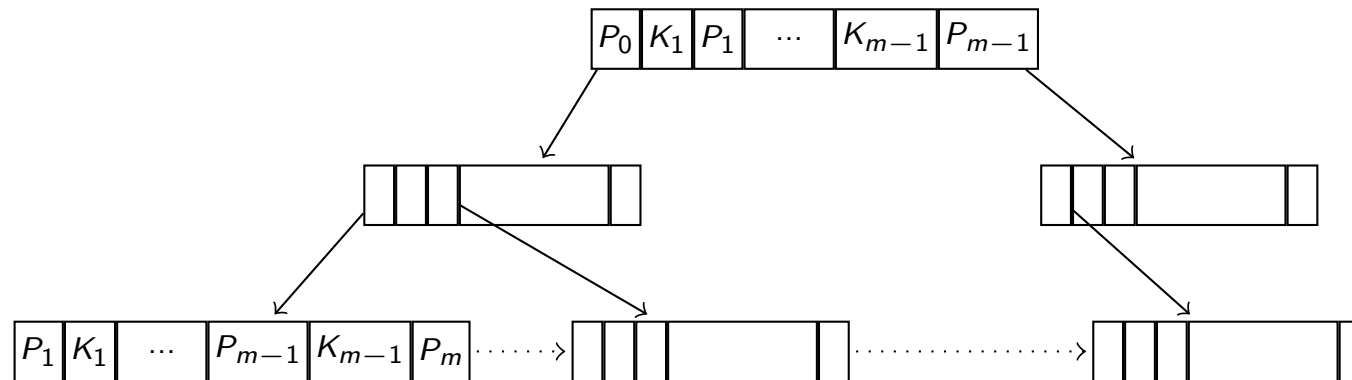
Inhalt

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - Statisches Hashing
 - Dynamisches Hashing
 - Mehrschlüssel Indices
 - Indizes in SQL

B^+ -Baum/1

B^+ -**Baum**: Alternative zu index-sequentiellen Dateien:

- **Vorteile** von B^+ -Bäumen:
 - Anzahl der Ebenen wird automatisch angepasst
 - reorganisiert sich selbst nach Einfüge- oder Löschoperationen durch kleine lokale Änderungen
 - reorganisieren der gesamten Datei ist nie erforderlich
- **Nachteile** von B^+ -Bäumen:
 - evtl. Zusatzaufwand bei Einfügen und Löschen
 - etwas höherer Speicherbedarf
 - komplexer zu implementieren
- Vorteile wiegen Nachteile in den meisten Anwendungen bei weitem auf, deshalb sind B^+ -Bäume die meist-verbreitete Index-Struktur

B^+ -Baum/2

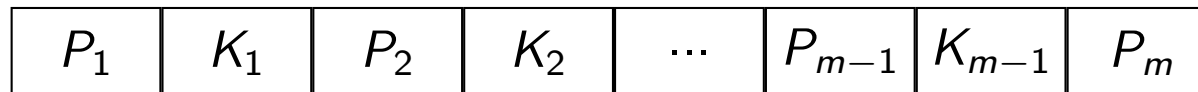
- **Knoten mit Grad m :** enthält bis zu $m - 1$ Suchschlüssel und m Pointer
 - Knotengrad $m > 2$ entspricht der maximalen Anzahl der Pointer
 - Suchschlüssel im Knoten sind sortiert
 - Knoten (außer Wurzel) sind mindestens halb voll
- **Wurzelknoten:**
 - als Blattknoten: 0 bis $m - 1$ Suchschlüssel
 - als innerer Knoten: mindestens 2 Kinder
- **Innerer Knoten:** $\lceil m/2 \rceil$ bis m Kinder (=Anzahl Pointer)
- **Blattknoten** $\lceil (m - 1)/2 \rceil$ bis $m - 1$ Suchschlüssel
- **balancierter Baum:** alle Pfade von der Wurzel zu den Blättern sind gleich lang (maximal $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$ Kanten für K Suchschlüssel)

Terminologie und Notation

- Ein Paar (P_i, K_i) ist ein Eintrag. Beachte:
 - Blatt: Pointer P_i ist links von Suchschlüssel K_i gespeichert
 - Innerer Knoten: P_i ist rechts von K_i gespeichert
- $L[i] = (P_i, K_i)$ bezeichnet den i -ten Eintrag von Knoten L
- **Daten-Pointer:** Pointer zu Datensätzen sind nur in den Blättern gespeichert
- **Verbindung zwischen Blättern:** der letzte Pointer im Blatt, P_m , zeigt auf das nächste Blatt

Anmerkung: Es gibt viele Varianten des B^+ -Baumes, die sich leicht unterscheiden. Auch in Lehrbüchern werden unterschiedliche Varianten vorgestellt. Für diese Lehrveranstaltung gilt der B^+ -Baum, wie er hier präsentiert wird.

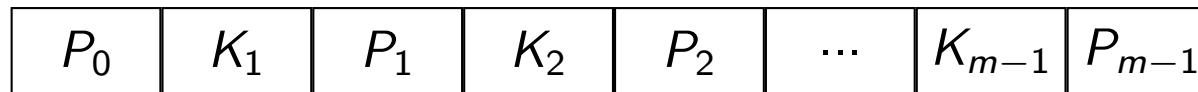
B^+ -Baum Knotenstruktur/1



Blatt-Knoten:

- K_1, \dots, K_{m-1} sind Suchschlüssel
- P_1, \dots, P_{m-1} sind Daten-Pointer
- Suchschlüssel sind sortiert: $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_{m-1}$
- Daten-Pointer P_i , $1 \leq i \leq m - 1$, zeigt auf
 - einen Datensatz mit Suchschlüssel K_i , oder
 - auf ein Bucket mit Pointern zu Datensätzen mit Suchschlüssel K_i
- P_m zeigt auf das nächste Blatt in Suchschlüssel-Ordnung

B^+ -Baum Knotenstruktur/2

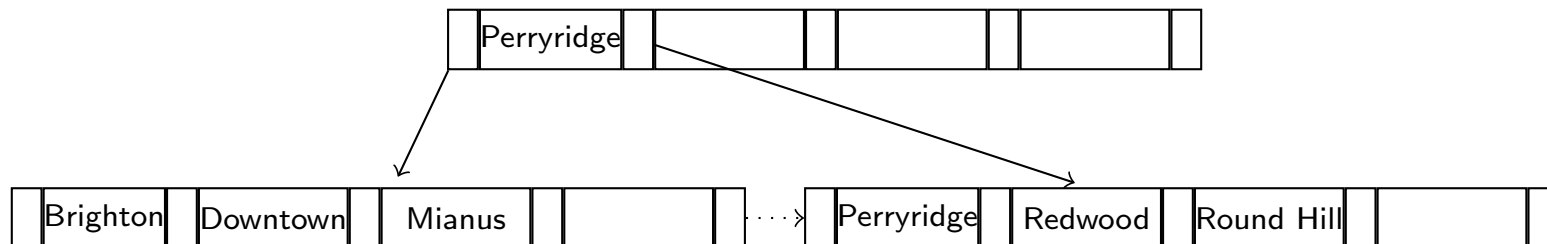


Innere Knoten:

- Stellen einen **mehrstufigen sparse Index** auf die Blattknoten dar.
- Suchschlüssel im Knoten sind **eindeutig**.
- P_0, \dots, P_{m-1} sind **Pointer zu Kind-Knoten**, d.h., zu Teilbäumen
- Alle **Suchschlüssel k** im Teilbaum von P_i haben folgende Eigenschaften:
 - $i = 0: k < K_1$
 - $0 < i < m - 1: K_{i-1} \leq k < K_{i+1}$
 - $i = m - 1: k > K_{m-1}$

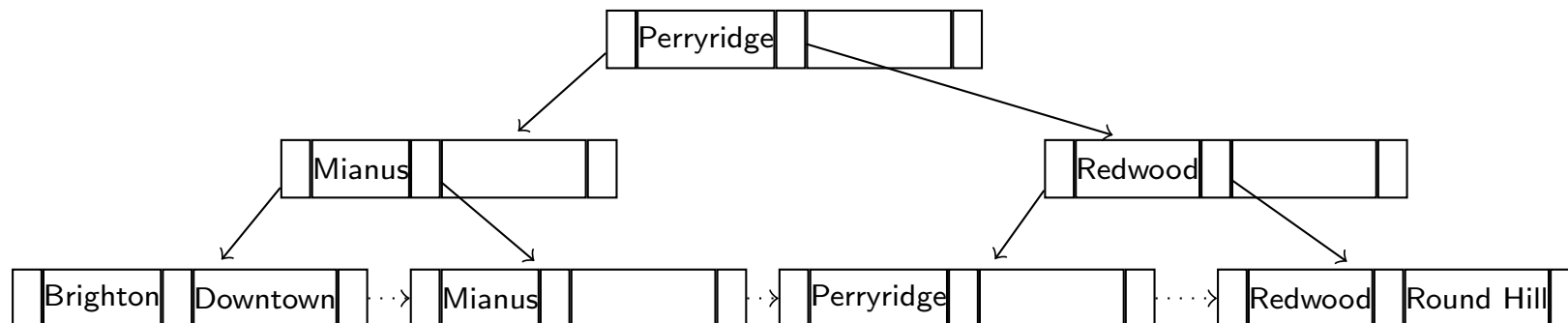
Beispiel: B^+ -Baum/1

- Index auf Konto-Relation mit Suchschlüssel Filiale
- B^+ -Baum mit Knotengrad $m = 5$:
 - Wurzel: mindestens 2 Pointer zu Kind-Knoten
 - Innere Knoten: $\lceil m/2 \rceil = 3$ bis $m = 5$ Pointer zu Kind-Knoten
 - Blätter: $\lceil (m - 1)/2 \rceil = 2$ bis $m - 1 = 4$ Suchschlüssel



Beispiel: B^+ -Baum/2

- B^+ -Baum für Konto-Relation (Knotengrad $m = 3$)
 - Wurzel: mindestens 2 Pointer zu Kind-Knoten
 - Innere Knoten: $\lceil m/2 \rceil = 2$ bis $m = 3$ Pointer zu Kind-Knoten
 - Blätter: $\lceil (m - 1)/2 \rceil = 1$ bis $m - 1 = 2$ Suchschlüssel

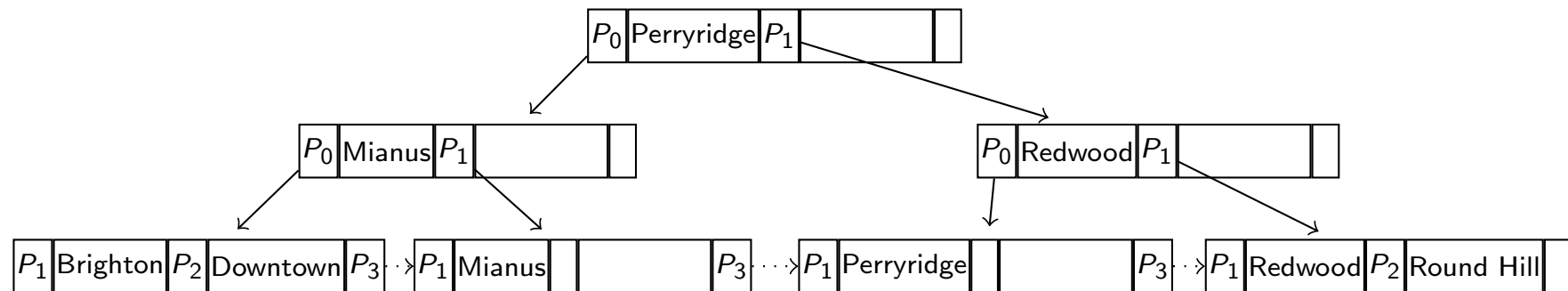


Suche im B^+ -Baum/1

- **Algorithmus: Suche** alle Datensätze mit Suchschlüssel k (Annahme: dense B^+ -Baum Index):
 1. $C \leftarrow$ Wurzelknoten
 2. **while** C keine Blattknoten **do**
 - suche im Knoten C nach dem größten Schlüssel $K_i \leq k$
 - if** ein Schlüssel $K_i \leq k$ existiert
 - then** $C \leftarrow$ Knoten auf den P_i zeigt
 - else** $C \leftarrow$ Knoten auf den P_o zeigt
 3. **if** es gibt einen Schlüssel K_i in C sodass $K_i = k$
 - then** folge Pointer P_i zum gesuchten Datensatz (oder Bucket)
 - else** kein Datensatz mit Suchschlüssel k existiert

Suche im B^+ -Baum/2

- **Beispiel:** Finde alle Datensätze mit Suchschlüssel $k = \text{Mianus}$
 - Beginne mit dem Wurzelknoten
 - Kein Schlüssel $K_i \leq \text{Mianus}$ existiert, also folge P_0
 - $K_1 = \text{Mianus}$ ist der größte Suchschlüssel $K_i \leq \text{Mianus}$, also folge P_1
 - Suchschlüssel Mianus existiert, also folge dem ersten Datensatz-Pointer P_1 um zum Datensatz zu gelangen



Suche im B^+ -Baum/3

- Suche durchläuft Pfad von Wurzel bis Blatt:
 - Länge des Pfads höchstens $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$
 - Wurzelknoten bleibt im Hauptspeicher, oft auch dessen Kinder, dadurch werden 1–2 Block-Zugriffe pro Suche gespart
- Suche effizienter als in sequentielllem Index:
 - $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$ Blocks lesen im B^+ -Baum
 - $\lceil \log_2(b) \rceil$ Blocks lesen im einstufigen sequentiellen Index (binäre Suche, Index mit b Blocks, $b = \lceil K/(m-1) \rceil$)

Integrierte Übung 1

Es soll ein Index mit 10^6 verschiedenen Suchschlüsseln erstellt werden. Ein Knoten kann im Durchschnitt 200 Schlüssel mit den entsprechenden Pointern speichern. Es soll nach einem bestimmten Suchschlüssel k gesucht werden.

- a) Wie viele Block-Zugriffe erfordert ein B^+ -Baum Index maximal, wenn kein Block im Hauptspeicher ist?
- b) Wie viele Block-Zugriffe erfordert ein einstufiger, sequentieller Index mit binärer Suche?

Einfügen in B^+ -Baum/1

- Datensatz mit Suchschlüssel k einfügen:
 1. füge Datensatz in Daten-Datei ein (ergibt Pointer)
 2. finde Blattknoten für Suchschlüssel k
 3. **falls** im Blatt noch Platz ist **dann**:
 - füge (Pointer, Suchschlüssel)-Paar so in Blatt ein, dass Ordnung der Suchschlüssel erhalten bleibt
 4. **sonst** (Blatt ist voll) teile Blatt-Knoten:
 - a) sortiere alle Suchschlüssel (einschließlich k)
 - b) die Hälfte der Suchschlüssel bleiben im alten Knoten
 - c) die andere Hälfte der Suchschlüssel kommt in einen neuen Knoten
 - d) füge den kleinsten Eintrag des neuen Knotens in den Eltern-Knoten des geteilten Knotens ein
 - e) **falls** Eltern-Knoten voll ist **dann**:
teile den Knoten und propagiere Teilung nach oben, sofern nötig

Einfügen in B^+ -Baum/2

- Aufteilvorgang:
 - falls nach einer Teilung der neue Schlüssel im Elternknoten nicht Platz hat wird auch dieser geteilt
 - im schlimmsten Fall wird der Wurzelknoten geteilt und der B^+ -Baum wird um eine Ebene tiefer

Algorithmus: Einfügen in B^+ -Baum

→ Knoten L , Suchschlüssel k , Pointer p (zu Datensatz oder Knoten)

Algorithm 1: B+TreeInsert(L, k, p)

if L is not yet full **then**

└ insert (k, p) into L

else

└ $L \leftarrow L \cup (k, p)$;

create new node L' ;

if L is a leaf **then**

└ $k' \leftarrow$ key of $L[\lceil (m+1)/2 \rceil]$;

└ move entries greater or equal to k' from L to L' ;

else

└ $k' \leftarrow$ key of $L[\lceil m/2 \rceil]$;

└ move entries greater or equal to k' from L to L' ;

└ delete entry with value k' from L' ;

└ **if** L is not the root **then** B+TreeInsert(parent(L), k', L') **else** create new root with children L and L' and value k'

Blatt teilen/1

Füge (k, p) in L ein:

p_1	k_1	p_2	k_2	p_3
-------	-------	-------	-------	-------

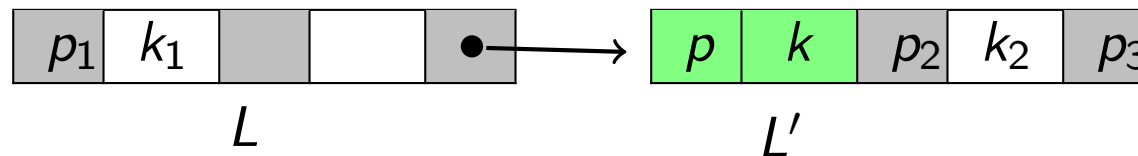
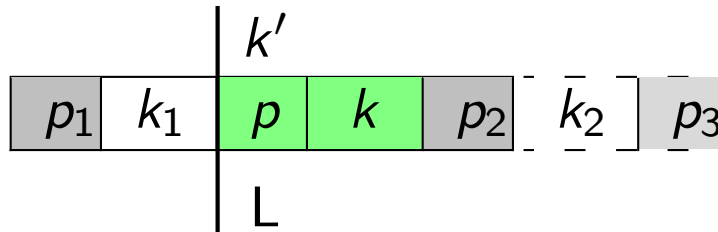
1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

L

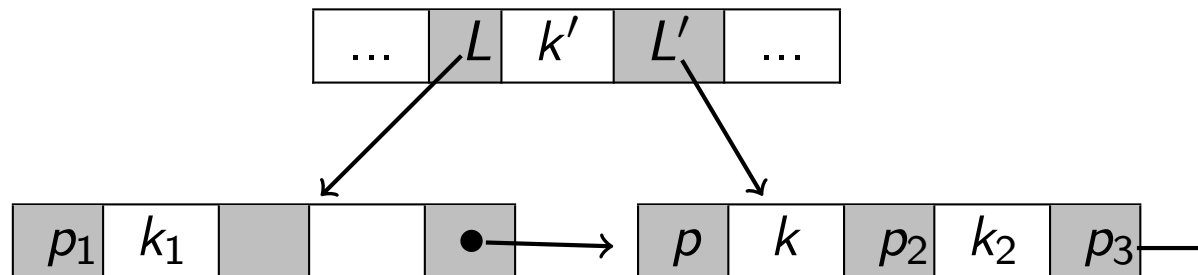
p_1	k_1	p	k	p_2	k_2	p_3
-------	-------	-----	-----	-------	-------	-------

 →

2. Teilen ($k' = L \lceil \frac{m+1}{2} \rceil = L[2]$)



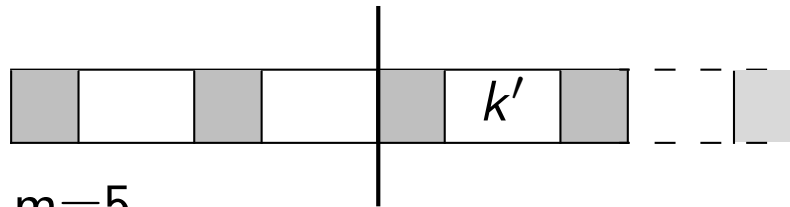
3. (k', L') in Elternknoten von L einfügen



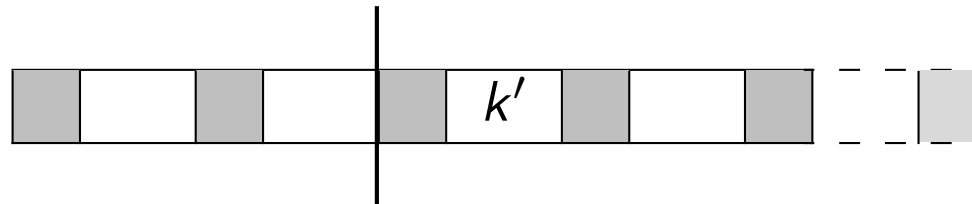
Blatt teilen/2

$$k' = L \left[\left\lceil \frac{m+1}{2} \right\rceil \right]$$

- m gerade, z.B.: m=4



- m ungerade, z.B.: m=5



Innere Knoten teilen/1

L

p_0	k_1	p_1	k_2	p_2
-------	-------	-------	-------	-------

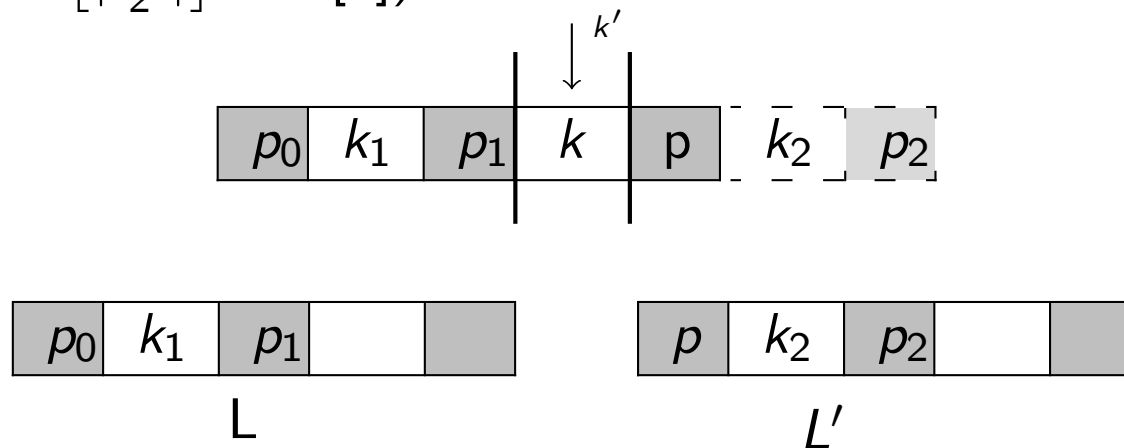
Füge (k, p) in L ein:

1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

L

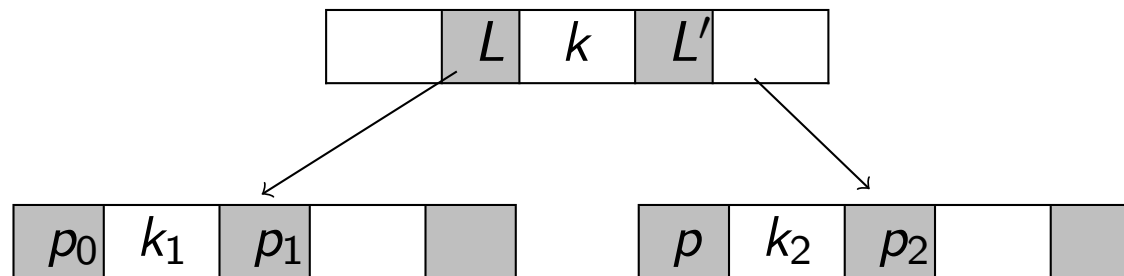
p_0	k_1	p_1	k	p	k_2	p_2
-------	-------	-------	-----	-----	-------	-------

2. Teilen ($k' = L \lceil \lceil \frac{m}{2} \rceil \rceil = L[2]$)



Innere Knoten teilen/2

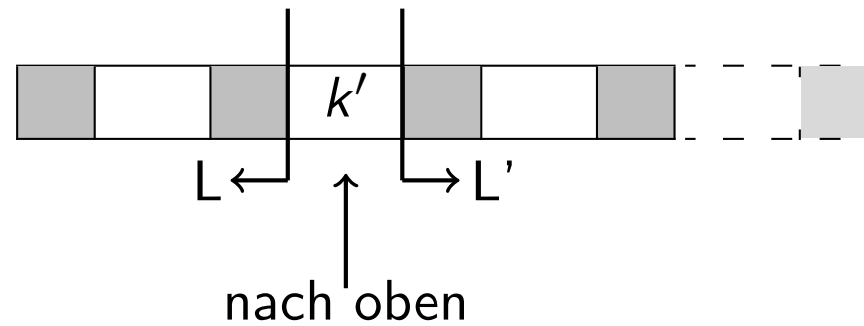
3. (k', L') in Elternknoten von L einfügen



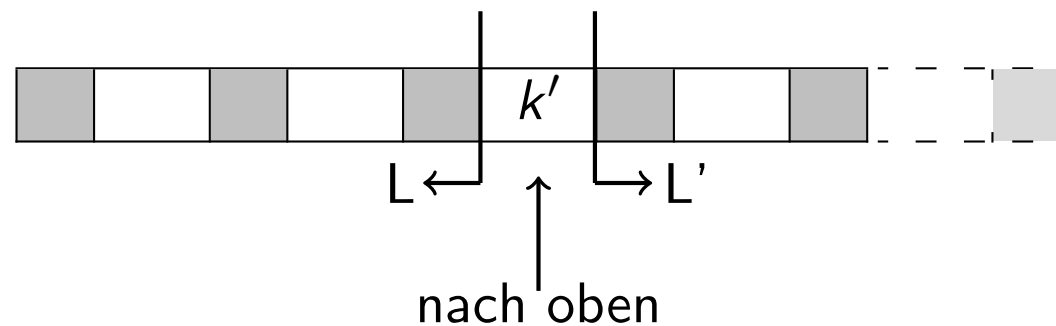
Innere Knoten teilen/3

$$k' = L \lceil \lceil \frac{m}{2} \rceil \rceil$$

- m gerade, z.B.: m=4

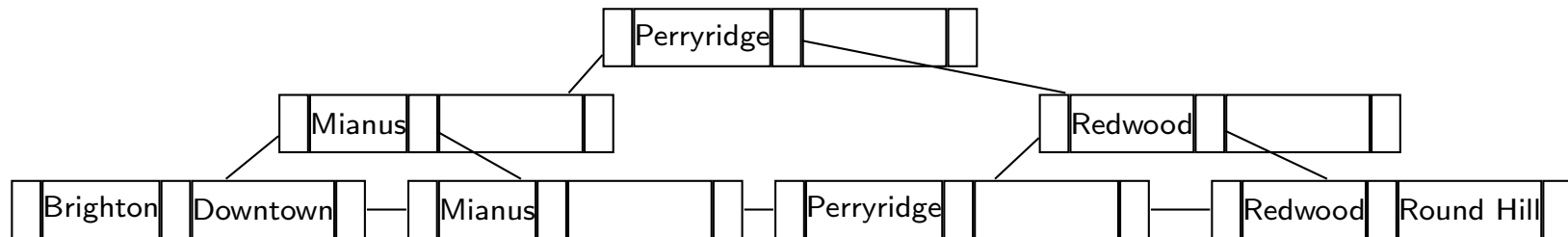


- m ungerade, z.B.: m=5

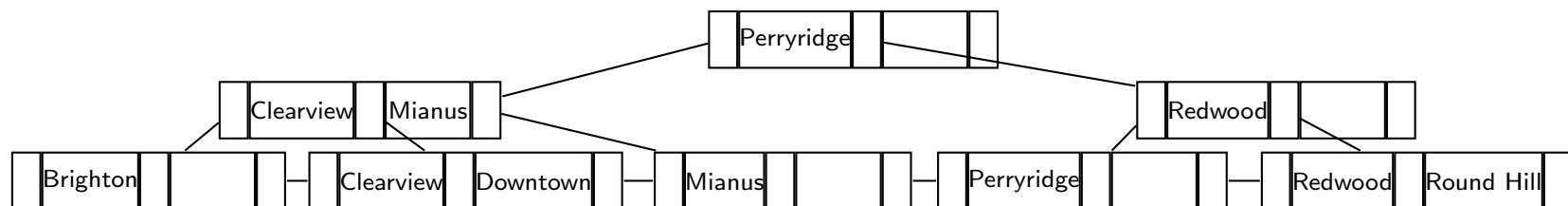


Beispiel: Einfügen in B^+ -Baum/1

- B^+ -Baum vor Einfügen von *Clearview*

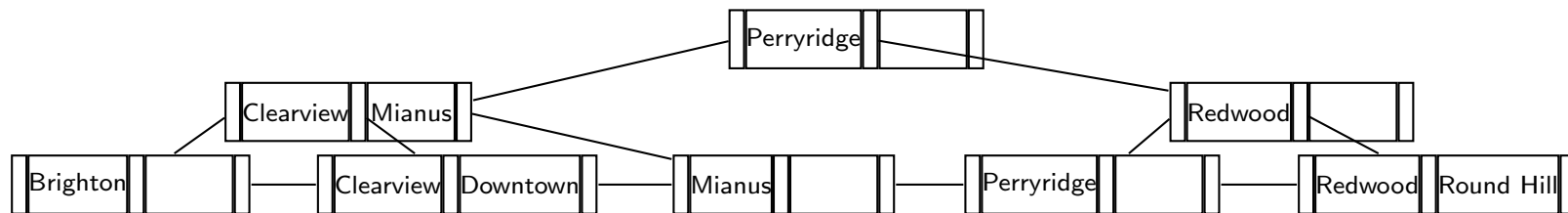


- B^+ -Baum nach Einfügen von *Clearview*

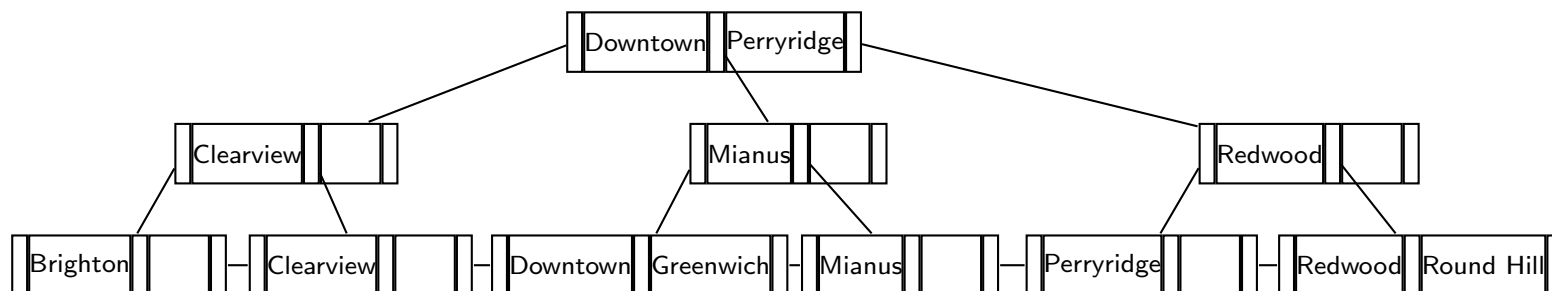


Beispiel: Einfügen in B^+ -Baum/2

- B^+ -Baum vor Einfügen von *Greenwich*



- B^+ -Baum nach Einfügen von *Greenwich*



Löschen von B^+ -Baum/1

Datensatz mit Suchschlüssel k löschen:

1. finde Blattknoten mit Suchschlüssel k
2. lösche k von Knoten
3. **falls** Knoten durch Löschen von k zu wenige Einträge hat:
 - a. Einträge im Knoten und einem Geschwisterknoten passen in 1 Knoten **dann**:
 - **vereinige** die beiden Knoten in einen einzigen Knoten (den linken, falls er existiert; ansonsten den rechten) und lösche den anderen Knoten
 - lösche den Eintrag im Elternknoten der zwischen den beiden Knoten ist und wende Löschen rekursiv an
 - b. Einträge im Knoten und einem Geschwisterknoten passen *nicht* in 1 Knoten **dann**:
 - **verteile** die Einträge zwischen den beiden Knoten sodass beide die minimale Anzahl von Einträgen haben
 - aktualisiere den entsprechenden Suchschlüssel im Eltern-Knoten

Löschen von B^+ -Baum/2

- Vereinigung:
 - Vereinigung zweier Knoten propagiert im Baum nach oben bis ein Knoten mit mehr als $\lceil m/2 \rceil$ Kindern gefunden wird
 - falls die Wurzel nach dem Löschen nur mehr ein Kind hat, wird sie gelöscht und der Kind-Knoten wird zur neuen Wurzel

Algorithmus: Löschen im B^+ -Baum

Algorithm 2: B+TreeDelete(L,k,p)

```

delete (p,k) from L;
if L is root with one child then root := child;
else if L has too few entries then
  L' is previous sibling of L [next if there is no previous] ;
  k' is value in parent that is between L and L';
  if entries L and L' fit on one page then                                // vereinigen
    if L is leaf then move entries from L to L';                                // Blatt
    else move k' and all entries from L to L';                                // innerer Knoten
    B+TreeDelete(parent(L),k',L)
  else                                                                    // verteilen
    if L is leaf then                                                    // Blatt
      move last [first] entry of L' to L;
      replace k' in parent(L) by value of first entry in L [L'];
    else                                                                    // innerer Knoten
      move last [first] entry of L' to L;
      replace k' in parent(L) by value of first entry of L [L'];
      replace value of first entry in L [L'] by k';

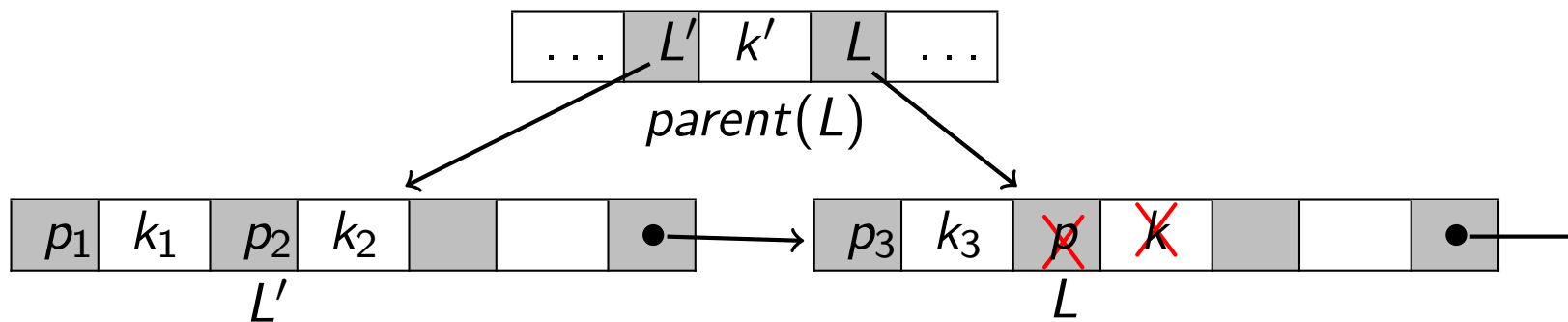
```

Löschen aus Blatt/1

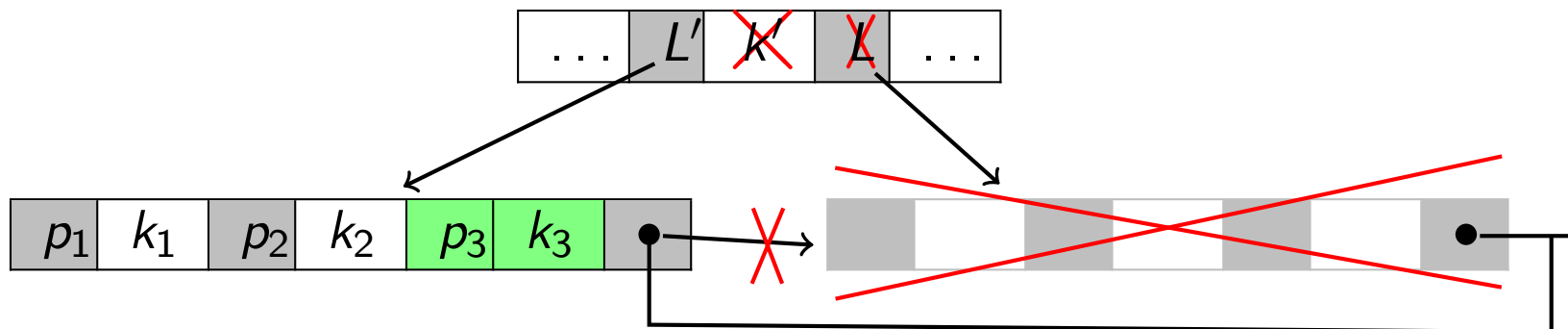
(k, p) wird aus L gelöscht:

1. Vereinigen ($m = 4$)

Vorher:



Nachher:

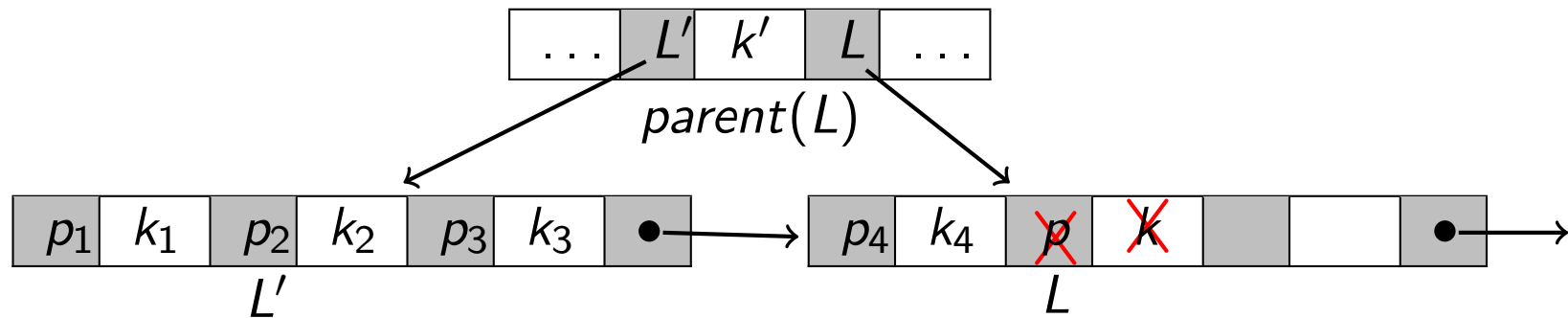


Löschen aus Blatt/2

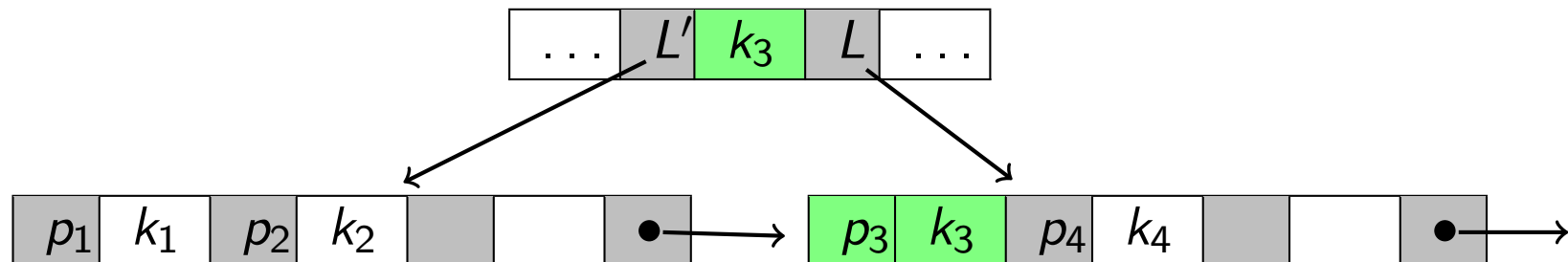
(k, p) wird aus L gelöscht:

2. Verteilen ($m = 4$)

Vorher:



Nachher:

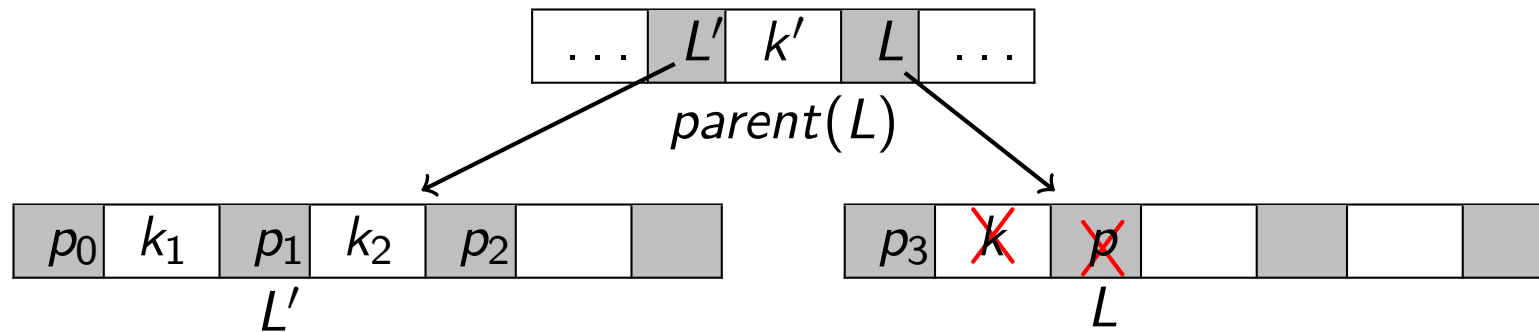


Löschen aus innerem Knoten/1

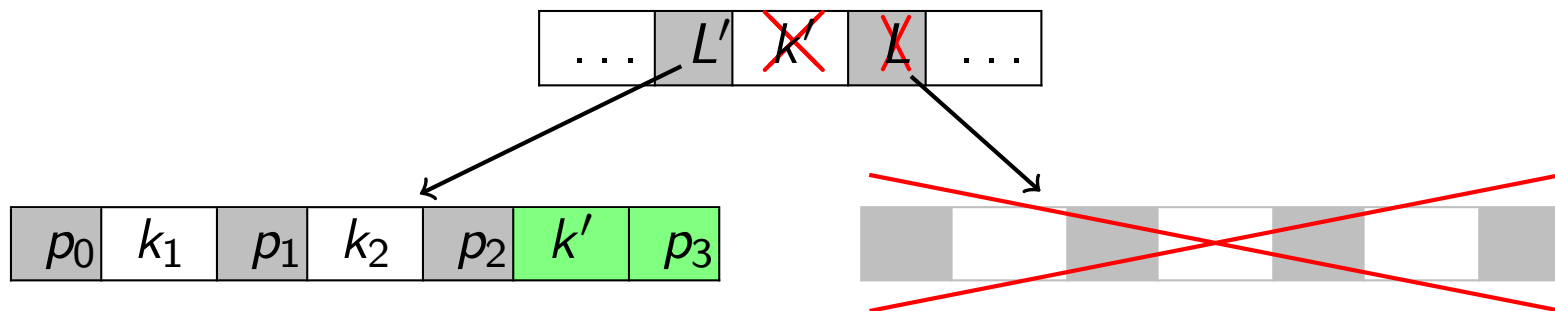
(k, p) wird aus L gelöscht:

1. Vereinigen ($m = 4$)

Vorher:



Nachher:

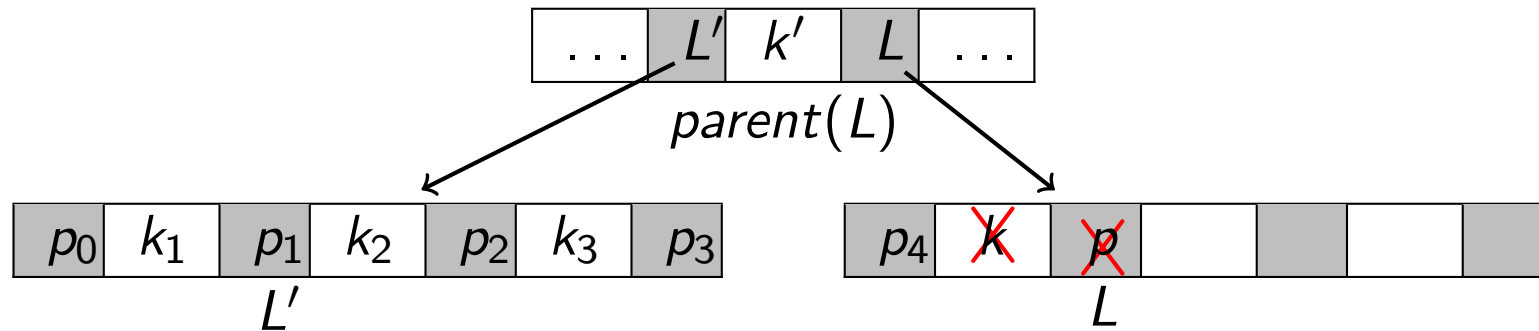


Löschen aus innerem Knoten/2

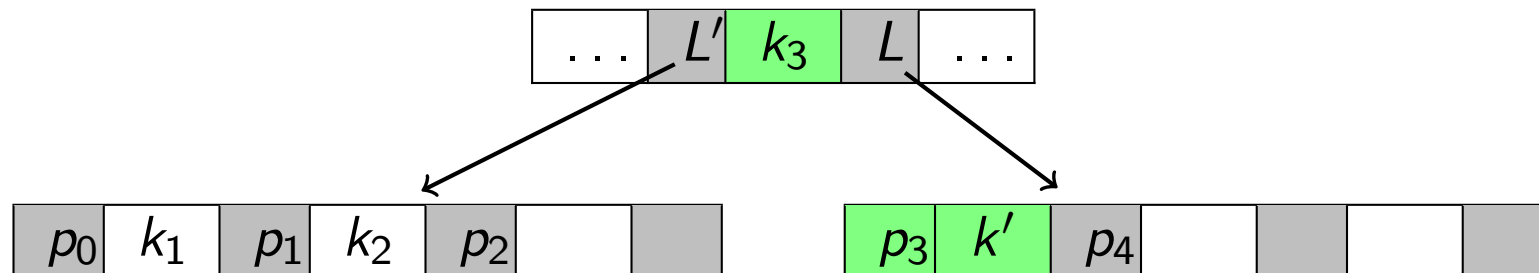
(k, p) wird aus L gelöscht:

2. Verteilen ($m = 4$)

Vorher:

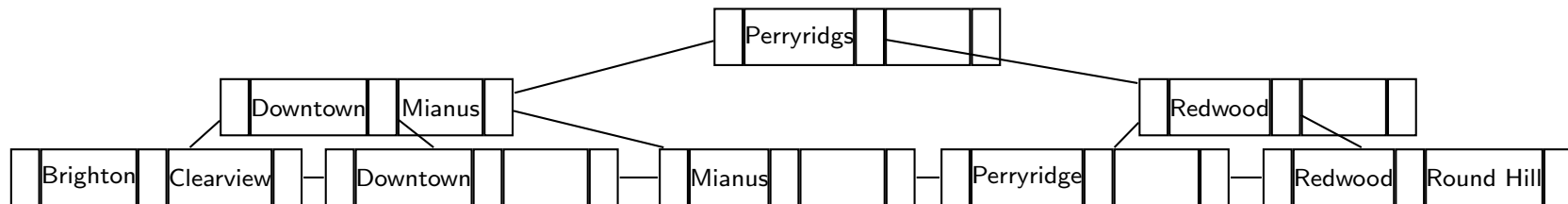


Nachher:

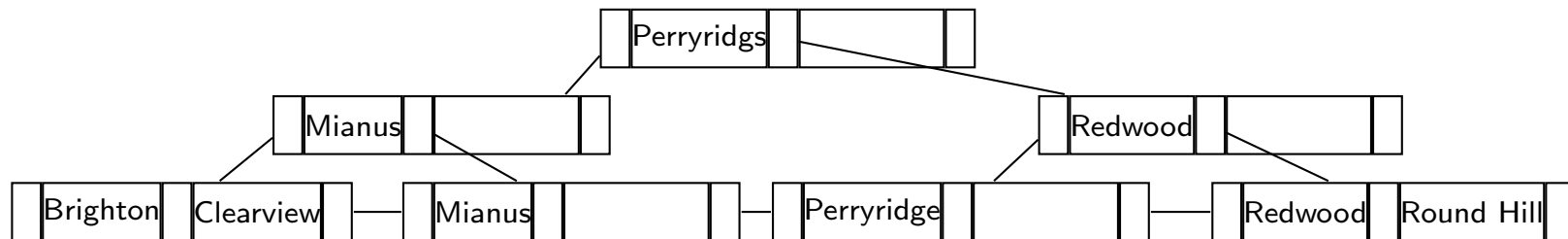


Beispiel: Löschen von B^+ -Baum/1

- Vor Löschen von *Downtown*:



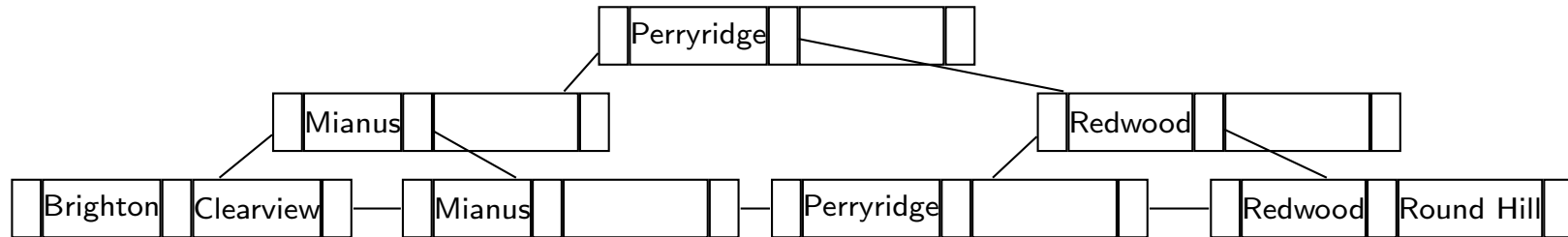
- Nach Löschen von *Downtown*:



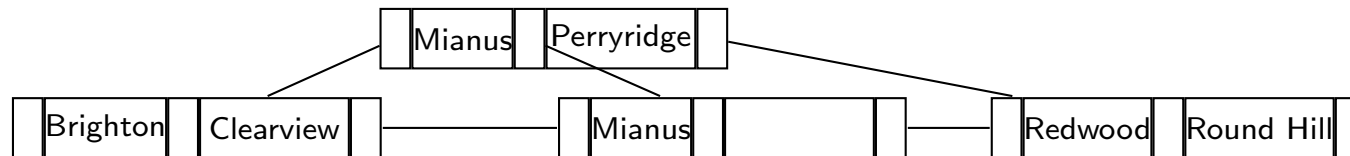
- Nach Löschen des Blattes mit *Downtown* hat der Elternknoten noch genug Pointer.
- Somit propagiert die Vereinigung durch Löschen nicht weiter nach oben.

Beispiel: Löschen von B^+ -Baum/2

- Vor Löschen von *Perryridge*:



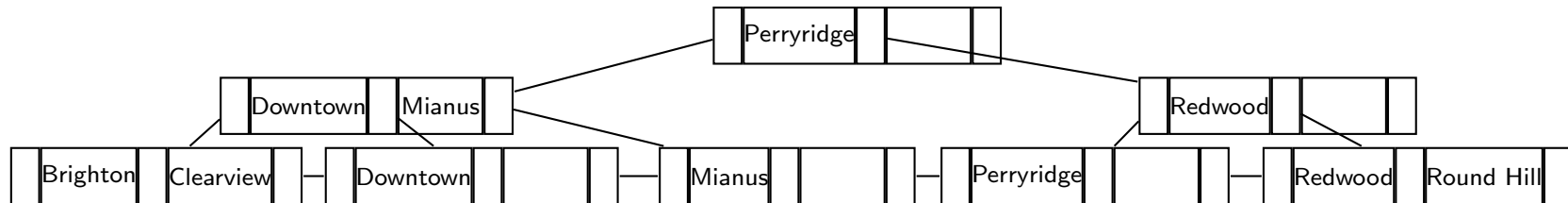
- Nach Löschen von *Perryridge*:



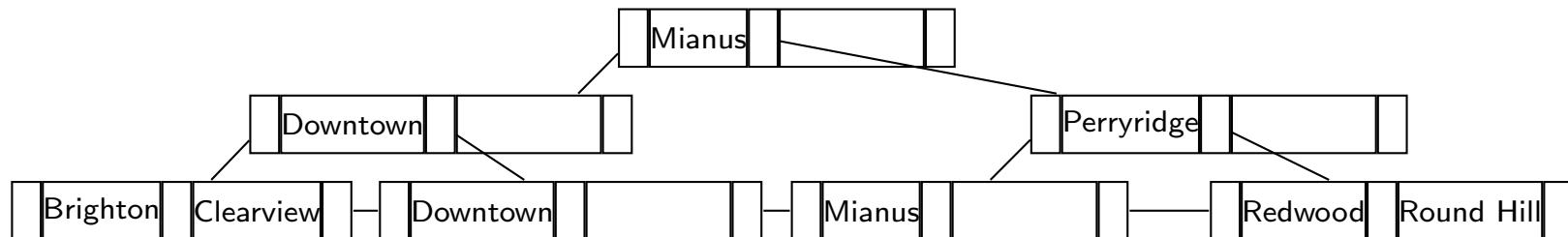
- Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und wird mit dem (rechten) Nachbarknoten **vereinigt**.
- Dadurch hat der Elternknoten zu wenig Pointer und wird mit seinem (linken) Nachbarknoten **vereinigt** (und ein Eintrag wird vom gemeinsamen Elternknoten gelöscht).
- Die Wurzel hat jetzt nur noch 1 Kind und wird gelöscht.

Beispiel: Löschen von B^+ -Baum/3

- Vor Löschen von *Perryridge*:



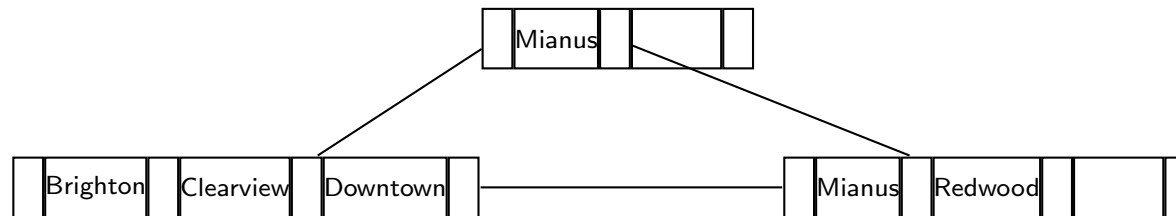
- Nach Löschen von *Perryridge*:



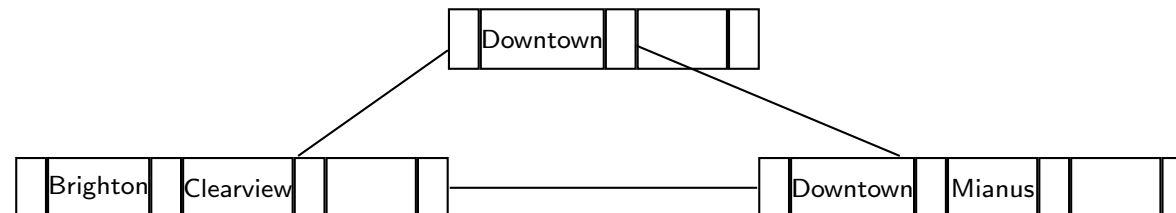
- Elternknoten von Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und erhält einen Pointer vom linken Nachbarn (**Verteilung** von Einträgen).
- Schlüssel im Elternknoten des Elternknotens (Wurzel in diesem Fall) ändert sich ebenfalls.

Beispiel: Löschen von B^+ -Baum/4

- Vor Löschen von *Redwood*:



- Nach Löschen von *Redwood*:



- Knoten von Blatt mit *Redwood* hat durch Löschen zu wenig Einträge und erhält einen Eintrag vom linken Nachbarn (**Verteilung** von Einträgen).
- Schlüssel im Elternknoten (Wurzel in diesem Fall) ändert sich ebenfalls.

Zusammenfassung B^+ -Baum

- Knoten mit Pointern verknüpft:
 - logisch nahe Knoten müssen nicht physisch nahe gespeichert sein
 - erlaubt mehr Flexibilität
 - erhöht die Anzahl der nicht-sequentiellen Zugriffe
- B^+ -Bäume sind flach:
 - maximale Tiefe $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$ für K Suchschlüssel
 - m ist groß in der Praxis (z.B. $m = 200$)
- Suchschlüssel als “Wegweiser”:
 - einige Suchschlüssel kommen als Wegweiser in einem oder mehreren inneren Knoten vor
 - zu einem Wegweiser gibt es nicht immer einen Suchschlüssel in einem Blattknoten (z.B. weil der entsprechende Datensatz gelöscht wurde)
- Einfügen und Löschen sind effizient:
 - nur $O(\log(K))$ viele Knoten müssen geändert werden
 - Index degeneriert nicht, d.h. Index muss nie von Grund auf rekonstruiert werden

Inhalt

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - **Statisches Hashing**
 - Dynamisches Hashing
 - Mehrschlüssel Indices
 - Indizes in SQL

Statisches Hashing

- Nachteile von ISAM und B^+ -Baum Indices:
 - B^+ -Baum: Suche muss Indexstruktur durchlaufen
 - ISAM: binäre Suche in großen Dateien
 - das erfordert zusätzliche Zugriffe auf Plattenblöcke
- Hashing:
 - erlaubt es auf Daten direkt und ohne Indexstrukturen zuzugreifen
 - kann auch zum Bauen eines Index verwendet werden

Hash Datei Organisation

- **Statisches Hashing** ist eine **Form der Dateiorganisation**:
 - Datensätze werden in Buckets gespeichert
 - Zugriff erfolgt über eine Hashfunktion
 - Eigenschaften: konstante Zugriffszeit, kein Index erforderlich
- **Bucket**: Speichereinheit die ein oder mehrere Datensätze enthält
 - ein Block oder mehrere benachbarte Blocks auf der Platte
 - alle Datensätze mit bestimmtem Suchschlüssel sind im selben Bucket
 - Datensätze im Bucket können verschiedene Suchschlüssel haben
- **Hash Funktion h** : bildet Menge der Suchschlüssel K auf Menge der Bucket Adressen B ab
 - wird in konstanter Zeit (in der Anzahl der Datensätze) berechnet
 - mehrere Suchschlüssel können auf dasselbe Bucket abbilden
- **Suchen eines Datensatzes** mit Suchschlüssel:
 - verwende Hash Funktion um Bucket Adresse aufgrund des Suchschlüssels zu bestimmen
 - durchsuche Bucket nach Datensätzen mit Suchschlüssel

Beispiel: Hash Datei Organisation

- **Beispiel:** Organisation der Konto-Relation als Hash Datei mit Filialname als Suchschlüssel.
- 10 Buckets
- Numerischer Code des i -ten Zeichens im 26-Buchstaben-Alphabet wird als i angenommen, z.B., $\text{code}(B)=2$.
- Hash Funktion h
 - Summe der Codes aller Zeichen modulo 10:
 - $h(\text{Perryridge}) = 125 \bmod 10 = 5$
 - $h(\text{Round Hill}) = 113 \bmod 10 = 3$
($\text{code}(' ') = 0$)
 - $h(\text{Brighton}) = 93 \bmod 10 = 3$

bucket 0		
bucket 1		
bucket 2		
bucket 3		
A-217	Brighton	750
A-305	Round Hill	350
bucket 4		
A-222	Redwood	700
bucket 5		
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
bucket 6		
bucket 7		
A-215	Mianus	700
bucket 8		
A-101	Downtown	500
A-110	Downtown	600
bucket 9		

Hash Funktionen/1

- Die **Worst Case Hash Funktion** bildet alle Suchschlüssel auf das gleiche Bucket ab.
 - Zugriffszeit wird linear in der Anzahl der Suchschlüssel.
- Die **Ideale Hash Funktion** hat folgende Eigenschaften:
 - Die Verteilung ist **uniform** (gleichverteilt), d.h. jedes Bucket ist der gleichen Anzahl von Suchschlüsseln aus der Menge aller Suchschlüssel zugewiesen.
 - Die Verteilung ist **random** (zufällig), d.h. im Mittel erhält jedes Bucket gleich viele Suchschlüssel unabhängig von der Verteilung der Suchschlüssel.

Hash Funktionen/2

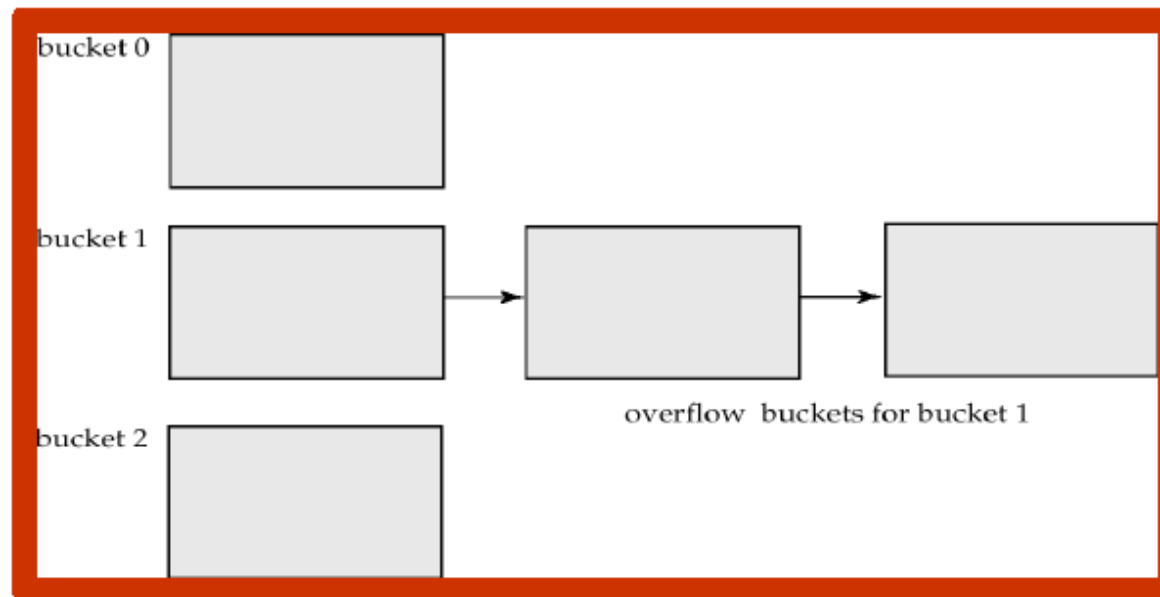
- **Beispiel:** 26 Buckets und eine Hash Funktion welche Filialnamen die mit dem i -ten Buchstaben beginnen dem Bucket i zuordnet.
 - keine Gleichverteilung, da es für bestimmte Anfangsbuchstaben erwartungsgemäß mehr Suchschlüssel gibt, z.B. erwarten wir mehr Filialen die mit B beginnen als mit Q.
- **Beispiel:** Hash Funktion die Kontostand nach gleich breiten Intervallen aufteilt: $1 - 10000 \rightarrow 0$, $10001 - 20000 \rightarrow 1$, usw.
 - uniform, da es für jedes Bucket gleich viele mögliche Werte von Kontostand gibt
 - nicht random, da Kontostände in bestimmten Intervallen häufiger sind, aber jedem Intervall 1 Bucket zugeordnet ist
- **Typische Hash Funktion:** Berechnung auf interner Binärdarstellung des Suchschlüssels, z.B. für String s mit n Zeichen, b buckets:
 - $(s[0] + s[1] + \dots + s[n - 1]) \bmod b$, oder
 - $(31^{n-1}s[0] + 31^{n-2}s[1] + \dots + s[n - 1]) \bmod b$

Bucket Overflow/1

- **Bucket Overflow:** Wenn in einem Bucket nicht genug Platz für alle zugehörigen Datensätze ist, entsteht ein Bucket overflow. Das kann aus zwei Gründen geschehen:
 - zu wenig Buckets
 - Skew: ungleichmäßige Verteilung der Hashwerte
- **Zu wenig Buckets:** die Anzahl n_B der Buckets muss größer gewählt werden als die Anzahl der Datensätze n geteilt durch die Anzahl der Datensätze pro Bucket f : $n_B > n/f$
- **Skew:** Ein Bucket ist überfüllt obwohl andere Buckets noch Platz haben. Zwei Gründe:
 - viele Datensätze haben gleichen Suchschlüssel (ungleichmäßige Verteilung der Suchschlüssel)
 - Hash Funktion erzeugt ungleichmäßige Verteilung
- Obwohl die Wahrscheinlichkeit für Overflows reduziert werden kann, können **Overflows nicht gänzlich vermieden** werden.
 - Overflows müssen behandelt werden
 - Behandlung durch Overflow Chaining

Bucket Overflow/2

- Overflow Chaining (closed hashing)
 - falls ein Datensatz in Bucket b eingefügt wird und b schon voll ist, wird ein Overflow Bucket b' erzeugt, in das der Datensatz gespeichert wird
 - die Overflow Buckets für Bucket b werden in einer Liste verkettet
 - für einen Suchschlüssel in Bucket b müssen auch alle Overflow Buckets von b durchsucht werden



Bucket Overflow/3

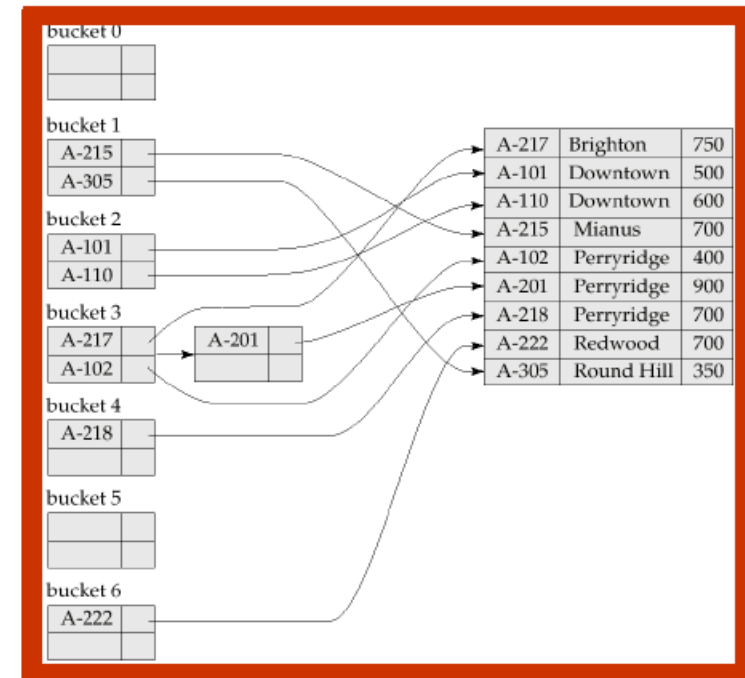
- **Open Hashing:** Die Menge der Buckets ist fix und es gibt keine Overflow Buckets.
 - überzählige Datensätze werden in ein anderes (bereits vorhandenes) Bucket gegeben, z.B. das nächste das noch Platz hat (linear probing)
 - wird z.B. für Symboltabellen in Compilern verwendet, hat aber wenig Bedeutung in Datenbanken, da Löschen schwieriger ist

Hash Index

- **Hash Index:** organisiert (Suchschlüssel, Pointer) Paare als Hash Datei
 - Pointer zeigt auf Datensatz
 - Suchschlüssel kann mehrfach vorkommen

- **Beispiel:** Index auf Konto-Relation.

- Hash Funktion h : Quersumme der Kontonummer modulo 7
- Beachte: Konto-Relation ist nach Filialnamen geordnet



- Hash Index ist immer **Sekundärindex**:
 - ist deshalb immer “dense”
 - Primär- bzw. Clustered Hash Index entspricht einer Hash Datei Organisation (zusätzliche Index-Ebene überflüssig)

Inhalt

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - Statisches Hashing
 - **Dynamisches Hashing**
 - Mehrschlüssel Indices
 - Indizes in SQL

Probleme mit Statischem Hashing

- **Richtige Anzahl** von Buckets ist kritisch für Performance:
 - zu wenig Buckets: Overflows reduzieren Performance
 - zu viele Buckets: Speicherplatz wird verschwendet (leere oder unterbesetzte Buckets)
- **Datenbank wächst oder schrumpft** mit der Zeit:
 - großzügige Schätzung: Performance leidet zu Beginn
 - knappe Schätzung: Performance leidet später
- **Reorganisation** des Index als einziger Ausweg:
 - Index mit neuer Hash Funktion neu aufbauen
 - sehr teuer, während der Reorganisation darf niemand auf die Daten schreiben
- **Alternative:** Anzahl der Bucket dynamisch anpassen

Dynamisches Hashing

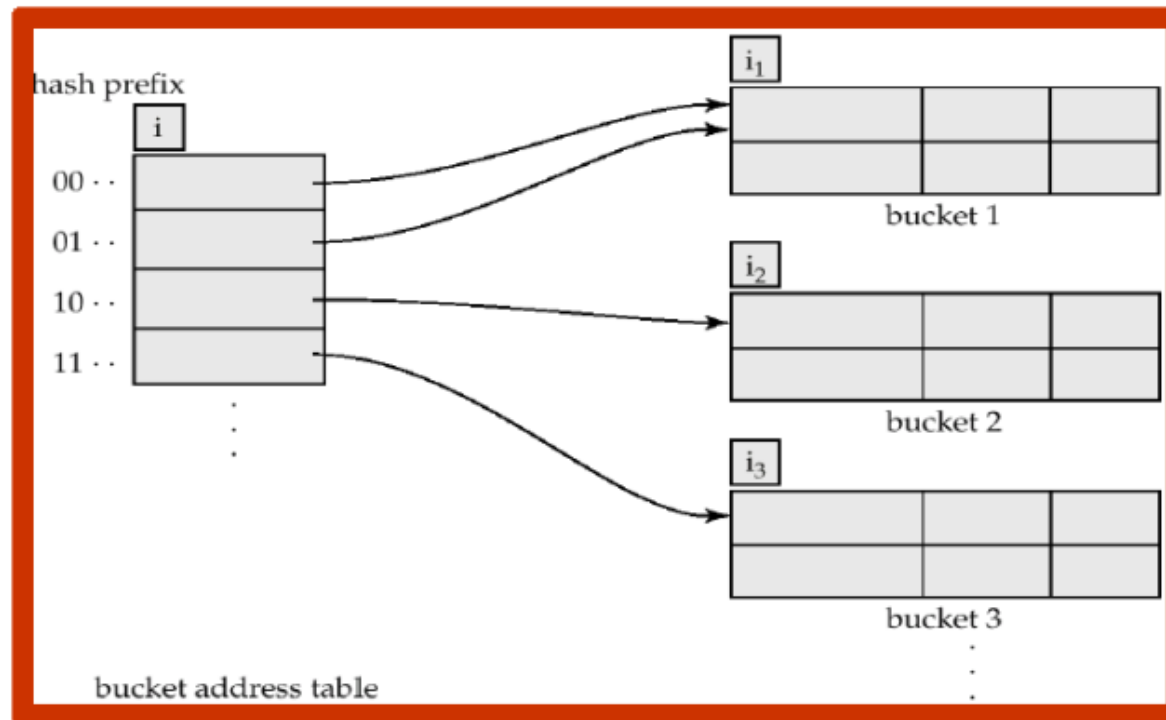
- **Dynamisches Hashing** (dynamic hashing): Hash Funktion wird dynamisch angepasst.
- **Erweiterbares Hashing** (extendible hashing): Eine Form des dynamischen Hashing.

Erweiterbares Hashing

- Hash Funktion h berechnet Hash Wert für sehr viele Buckets:
 - eine b -Bit Integer Zahl
 - typisch $b = 32$, also 4 Milliarden (möglichen) Buckets
- Hash-Prefix:
 - nur die i höchstwertigen Bits (MSB) des Hash-Wertes werden verwendet
 - $0 \leq i \leq b$ ist die *globale Tiefe*
 - i wächst oder schrumpft mit Datenmenge, anfangs $i = 0$
- Verzeichnis: (directory, bucket address table)
 - Hauptspeicherstruktur: Array mit 2^i Einträgen
 - Hash-Prefix indiziert einen Eintrag im Verzeichnis
 - jeder Eintrag verweist auf ein Bucket
 - mehrere aufeinanderfolgende Einträge im Verzeichnis können auf dasselbe Bucket zeigen

Erweiterbares Hashing

- Buckets:
 - Anzahl der Buckets $\leq 2^i$
 - jedes Bucket j hat eine *lokale Tiefe* i_j
 - falls mehrere Verzeichnis-Pointer auf dasselbe Bucket j zeigen, haben die entsprechenden Hash Werte dasselbe i_j Prefix.
- Beispiel: $i = 2, i_1 = 1, i_2 = i_3 = 2,$



Erweiterbares Hashing: Suche

- **Suche:** finde Bucket für Suchschlüssel K
 1. berechne Hash Wert $h(K) = X$
 2. verwende die i höchstwertigen Bits (Hash Prefix) von X als Adresse ins Verzeichnis
 3. folge dem Pointer zum entsprechenden Bucket

Erweiterbares Hashing: Einfügen

- **Einfügen:** füge Datensatz mit Suchschlüssel K ein
 1. verwende Suche um richtiges Bucket j zu finden
 2. **If** genug freier Platz in Bucket j **then**
 - füge Datensatz in Bucket j ein
 3. **else**
 - teile Bucket und versuche erneut

Erweiterbares Hashing: Bucket teilen

- **Bucket j teilen** um Suchschlüssel K einzufügen
 - If $i > i_j$** (mehrere Pointer zu Bucket j) **then**
 - lege neues Bucket z an und setze i_z und i_j auf das alte $i_j + 1$
 - aktualisiere die Pointer die auf j zeigen (die Hälfte zeigt nun auf z)
 - lösche alle Datensätze von Bucket j und füge sie neu ein (sie verteilen sich auf Buckets j und z)
 - versuche K erneut einzufügen
 - Else if $i = i_j$** (nur 1 Pointer zu Bucket j) **then**
 - erhöhe i und verdopple die Größe des Verzeichnisses
 - ersetze jeden alten Eintrag durch zwei neue Einträge die auf dasselbe Bucket zeigen
 - versuche K erneut einzufügen
- **Overflow Buckets** müssen nur erzeugt werden, wenn das Bucket voll ist und alle Suchschlüssel im Bucket identisch sind (d.h., teilen würde nichts nützen)

Integrierte Übung 2

Betrachten Sie die folgende Hashfunktion:

Schlüsesel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Nehmen Sie Buckets der Größe 2 an und erweiterbares Hashing mit einem anfangs leeren Verzeichnis. Zeigen Sie die Hashtabelle nach folgenden Operationen:

- füge 1 Brighton und 2 Downtown Datensätze ein
- füge 1 Mianus Datensatz ein
- füge 1 Redwood Datensatz ein
- füge 3 Perryridge Datensätze ein

Erweiterbares Hashing: Löschen

- Löschen eines Suchschlüssels K
 1. suche Bucket j für Suchschlüssel K
 2. entferne alle Datensätze mit Suchschlüssel K
 3. Bucket j kann mit Nachbarbucket(s) verschmelzen falls
 - alle Suchschlüssel in einem Bucket Platz finden
 - die Buckets dieselbe lokale Tiefe i_j haben
 - die $i_j - 1$ Prefixe der entsprechenden Hash-Werte identisch ist
 4. Verzeichnis kann verkleinert werden, wenn $i_j < i$ für alle Buckets j

Integrierte Übung 3

Betrachten Sie die folgende Hashfunktion:

Schlüsesel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Gehen Sie vom Ergebnis der vorigen Übung aus und führen Sie folgende Operationen durch:

- 1 Brighton und 1 Downtown löschen
- 1 Redwood löschen
- 2 Perryridge löschen

Erweiterbares Hashing: Pro und Kontra

- **Vorteile** von erweiterbarem Hashing
 - bleibt effizient auch wenn Datei wächst
 - Overhead für Verzeichnis ist normalerweise klein im Vergleich zu den Einsparungen an Buckets
 - keine Buckets für zukünftiges Wachstum müssen reserviert werden
- **Nachteile** von erweiterbarem Hashing
 - zusätzliche Ebene der Indirektion – macht sich bemerkbar, wenn Verzeichnis zu groß für den Hauptspeicher wird
 - Verzeichnis vergrößern oder verkleinern ist relativ teuer

B^+ -Baum vs. Hash Index

- Hash Index degeneriert wenn es sehr viele identische (Hashwerte für) Suchschlüssel gibt – Overflows!
- Im Average Case für Punktanfragen in n Datensätzen:
 - Hash index: $O(1)$ (sehr gut)
 - B^+ -Baum: $O(\log n)$
- Worst Case für Punktanfragen in n Datensätzen:
 - Hash index: $O(n)$ (sehr schlecht)
 - B^+ -Baum: $O(\log n)$
- Anfragetypen:
 - Punktanfragen: Hash und B^+ -Baum
 - Mehrpunktanfragen: Hash und B^+ -Baum
 - Bereichsanfragen: Hash Index nicht brauchbar

Berkeley DB

o Database access methods:

- B+tree
- Hash (Extended Linear Hashing)
- Fixed/Variable Length Records
- Duplicate records per key in the B+tree and Hash access methods.
- Retrieval by record number in the B+tree access method.
- Keyed and sequential (forward and reverse) retrieval, insertion, modification and deletion.
- Memory-mapped read-only databases.
- Retrieval into user-specified or allocated memory.
- Partial-record data storage and retrieval.
- Architecture independent databases.
- Maximum B+tree depth of 255.
- Individual database files up to 2^{48} bytes, individual key or data elements up to 2^{32} bytes (or available memory).

Berkeley DB

The Berkeley Database Package (DB) is being used by many different organizations in many different applications! Here are a few of which you've probably heard:



[Netscape SuiteSpot](#), an integrated suite of intranet and Internet server software, lets you communicate, access, and share information throughout your organization. The [Enterprise](#), [Catalog](#), [Directory](#) and [Mail](#) servers all use Berkeley DB.



The Isode Ltd. [LDAP/X.500 Enterprise Directory Server](#) uses Berkeley DB as its primary database. Berkeley DB is also used in its [X.400/Internet Message Switch](#) and [X.400 Message Store](#) products. Here's the [press release](#) from Isode about Berkeley DB.



[Sendmail](#) is the program that routes electronic mail throughout the Internet. Sendmail is used on almost every UNIX-like system, and it uses Berkeley DB.



The [OSF Distributed Computing Environment](#) (DCE) is an industry-standard, vendor-neutral set of distributed computing technologies. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. The DCE backing store library ([Open Group RFC #45](#)) is a subset of Berkeley DB.

Inhalt

1 Indexstrukturen für Dateien

- Grundlagen
- B^+ -Baum
- Statisches Hashing
- Dynamisches Hashing
- **Mehrschlüssel Indices**
- Indizes in SQL

Zugriffe über mehrere Suchschlüssel/1

- Wie kann Index verwendet werden, um folgende Anfrage zu beantworten?
select *AccNr*
from *account*
where *BranchName* = "Perryridge" **and** *Balance* = 1000
- Strategien mit mehreren Indices (jeweils 1 Suchschlüssel):
 - a) *BranchName* = "Perryridge" mit Index auf *BranchName* auswerten; auf Ergebnis-Datensätzen *Balance* = 1000 testen.
 - b) *Balance* = 1000 mit Index auf *Balance* auswerten; auf Ergebnis-Datensätzen *BranchName* = "Perryridge" testen.
 - c) Verwende *BranchName* Index um Pointer zu Datensätzen mit *BranchName* = "Perryridge" zu erhalten; verwende *Balance* Index für Pointer zu Datensätzen mit *Balance* = 1000; berechne die Schnittmenge der beiden Pointer-Mengen.

Zugriffe über mehrere Suchschlüssel/2

- Nur die dritte Strategie nützt das Vorhandensein mehrere Indices.
- Auch diese Strategie kann eine schlechte Wahl sein:
 - es gibt viele Konten in der "Perryridge" Filiale
 - es gibt viele Konten mit Kontostand 1000
 - es gibt nur wenige Konten die beide Bedingungen erfüllen
- Effizientere Indexstrukturen müssen verwendet werden:
 - (traditionelle) Indices auf kombinierten Schlüsseln
 - spezielle mehrdimensionale Indexstrukturen, z.B., Grid Files, Quad-Trees, Bitmap Indices.

Zugriffe über mehrere Suchschlüssel/3

- Annahme: Geordneter **Index mit kombiniertem Suchschlüssel** (*BranchName, Balance*)
- Kombinierte Suchschlüssel haben eine **Ordnung** (*BranchName* ist das erste Attribut, *Balance* ist das zweite Attribut)
 - Folgende Bedingung wird effizient behandelt (alle Attribute):
where *BranchName* = "Perryridge" **and** *Balance* = 1000
 - Folgende Bedingung wird effizient behandelt (Prefix):
where *BranchName* = "Perryridge"
 - Folgende Bedingung ist ineffizient (kein Prefix der Attribute):
where *Balance* = 1000

Inhalt

- 1 Indexstrukturen für Dateien
 - Grundlagen
 - B^+ -Baum
 - Statisches Hashing
 - Dynamisches Hashing
 - Mehrschlüssel Indices
 - Indizes in SQL

Index Definition in SQL

- SQL-92 definiert keine Syntax für Indices da diese nicht Teil des logischen Datenmodells sind.
- Jedoch alle Datenbanksystem stellen Indices zur Verfügung.
- Index erzeugen:
create index <IdxName> **on** <RelName> (<AttrList>)
z.B. **create index** BrNalIdx **on** branch (branch-name)
- **Create unique index** erzwingt eindeutige Suchschlüssel und definiert indirekt eine Schlüsselattribut.
- Primärschlüssel (**primary key**) und Kandidatenschlüssel (**unique**) werden in SQL bei der Tabellendefinition spezifiziert.
- Index löschen:
drop index <index-name>
z.B. **drop index** BrNalIdx

Beispiel: Indices in PostgreSQL

- **CREATE [UNIQUE] INDEX** name **ON** table_name
"(" col [**DESC**] { "," col [**DESC**] } ")" [...]
- **Beispiele:**
 - **CREATE INDEX** MajIdx **ON** Enroll (Major);
 - **CREATE INDEX** MajIdx **ON** Enroll **USING HASH** (Major);
 - **CREATE INDEX** MajMinIdx **ON** Enroll (Major, Minor);

Indexes in Oracle

- B^+ -Baum Index in Oracle:

```
CREATE [UNIQUE] INDEX name ON table_name
  "(" col [DESC] { "," col [DESC] } ")" [pctfree n] [...]
```

- Anmerkungen:

- pct_free gibt an, wieviel Prozent der Knoten anfangs frei sein sollen.
- UNIQUE sollte nicht verwendet werden, da es ein logisches Konzept ist.
- Oracle erstellt einen B^+ -Baum Index für jede **unique** oder **primary key** definition bei der Erstellung der Tabelle.

- Beispiele:

```
CREATE TABLE BOOK (
  ISBN INTEGER, Author VARCHAR2 (30) , ... );
CREATE INDEX book_auth ON book(Author);
```

- Hash-partitionierter Index in Oracle:

```
CREATE INDEX CustLNameIX ON customers (LName) GLOBAL
PARTITION BY HASH (LName) PARTITIONS 4;
```

Anmerkungen zu Indices in Datenbanksystemen

- Indices werden **automatisch nachgeführt** wenn Tupel eingefügt, geändert oder gelöscht werden.
- Indices **verlangsamen** deshalb Änderungsoperationen.
- Einen **Index zu erzeugen** kann lange dauern.
- **Bulk Load**: Es ist (viel) effizienter, zuerst die Daten in die Tabelle einzufügen und nachher alle Indices zu erstellen als umgekehrt.

Zusammenfassung

- **Index Typen:**
 - Primary, Clustering und Sekundär
 - Dense oder Sparse
- **B^+ -Baum:**
 - universelle Indexstruktur, auch für Bereichsanfragen
 - Garantien zur Tiefe, Füllgrad und Effizienz
 - Einfügen und Löschen
- **Hash Index:**
 - statisches und erweiterbares Hashing
 - keine Index für Primärschlüssel nötig
 - gut für Prädikate mit “=”
- **Mehrschlüssel Indices:** schwieriger, da es keine totale Ordnung in mehreren Dimensionen gibt
- Indices in **SQL**