5. Eindimensionale Zugriffspfade

- **Ziele**
  - Entwurfsprinzipien für Zugriffspfade auf die Sätze einer Tabelle, bei denen ein Suchkriterium unterstützt wird
  - Abbildungsmöglichkeiten für hierarchische Zugriffsanforderungen
- **Anforderungen und Klassifikation**
- **Zugriffspfade für Primärschlüssel**
  - Mehrwegbäume (B- und B*-Bäume)
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- **Zugriff über Sekundärschlüssel**
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- **Hierarchische Zugriffspfade**
- **Verallgemeinerte Zugriffspfadstruktur**
- **Wichtige Kenngrößen:**
  - n = #Sätze eines Satztyps
  - b = mittlere #Sätze/Seite (Blockungsfaktor)
  - q = #Treffer der Anfrage
  - N_S = #Seitenzugriffe
  - N_B = #Blattseiten des B*-Baums
  - h_B = Höhe des B*-Baums

Anforderungen an Zugriffspfade

- **Folgende Arten von Zugriffen müssen unterstützt werden:**
  - Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)

    Select * From Pers
  - Sequentieller Zugriff in Sortierreihenfolge eines Attributes

    ... Order by Name
  - Direkter Zugriff über den Primärschlüssel

    ... Where Pnr = 0815
  - Direkter Zugriff über einen Sekundärschlüssel

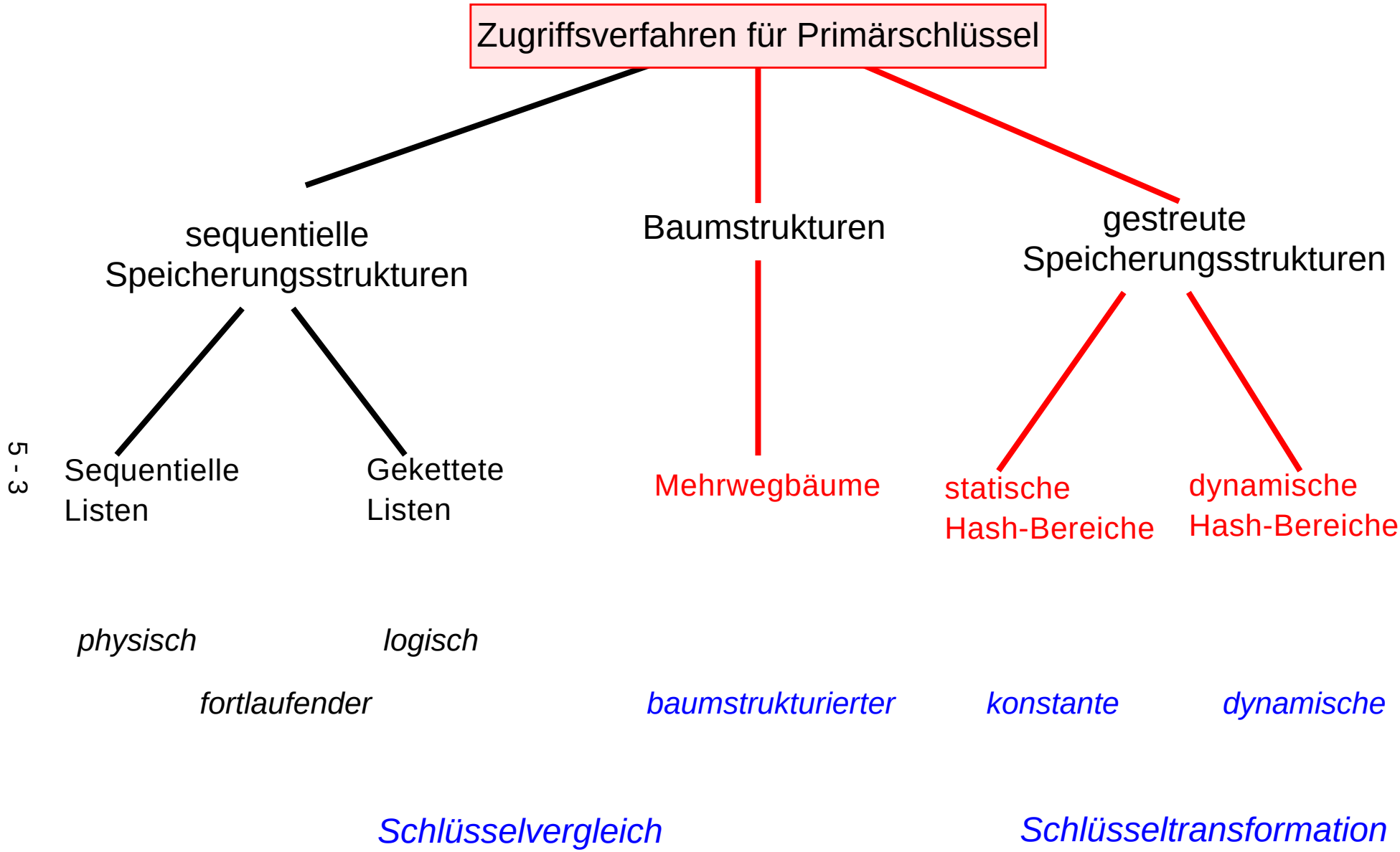
    ... Where Beruf = ‘Programmierer’
  - Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle, ...)

    ... Where Gehalt Between 50K And 100K
  - Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps

    ... Where P.Pnr = A.Pnr

➥ **Wenn kein geeigneter Zugriffspfad vorhanden ist, sind alle Zugriffsarten durch fortlaufende Suche (Scan) abzuwickeln**

- **Scan**
  - muß von allen DBMS unterstützt werden!
  - ist ausreichend / effizient bei:
    - kleinen Satztypen (z. B. ≤ 5 Seiten)
    - Anfragen mit großen Treffermengen (z. B. > 3%)
  - DBMS kann Prefetching zur Scan-Optimierung nutzen


Zugriffsverfahren für Primärschlüssel
sequentielle Speicherungsstrukturen
Baumstrukturen
gestreute Speicherungsstrukturen
Sequentielle Listen
Gekettete Listen
Mehrwegbäume
statische Hash-Bereiche
dynamische Hash-Bereiche
physisch
logisch
fortlaufender
baumstrukturierter
konstante
dynamische
Schlüsselvergleich
Schlüsseltransformation

Mehrwegbäume

- **Bezugsgröße: Seite = Transporteinheit zum Externspeicher**

  im Gegensatz zu Binärbäumen

- **Vorfahr: ISAM (statisch, periodische Reorganisation)**

- **Weiterentwicklung: B- und B*-Baum**
  - referenzierte und materialisierte Speicherung der Datensätze
  - dynamische Reorganisation durch Splitten und Mischen von Seiten

- **Funktion**
  - direkter Schlüsselzugriff
  - sortiert sequentieller Zugriff

- **Balancierte Struktur**
  - unabhängig von Schlüsselmenge
  - unabhängig von Einfügereihenfolge

- **Realisierung von Index-organisierten Tabellen**
  - oft nach Primärschlüssel geordnet
  - Cluster-Bildung durch eingebettete Datensätze

- **Verbesserung der Baumbreite (fan-out)**
  - Schlüsselkomprimierung
  - Nutzung von „Wegweisern” in B*-Bäumen
  - Präfix-B-Bäume

- **Verbesserung des Belegungsgrades**

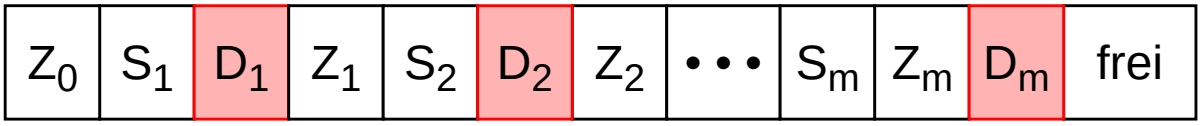
  ➥ verallgemeinertes Splittingverfahren

B-Baum

- **Def.: Ein B-Baum vom Typ (k, h) ist ein Baum mit folgenden Eigenschaften**

  1. Jeder Weg von der Wurzel zum Blatt hat die Länge h
  2. Jeder Zwischenknoten hat mindestens k+1 Söhne.
     Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne
  3. Jeder Knoten hat höchstens 2k+1 Söhne

- **Seitenformat:**

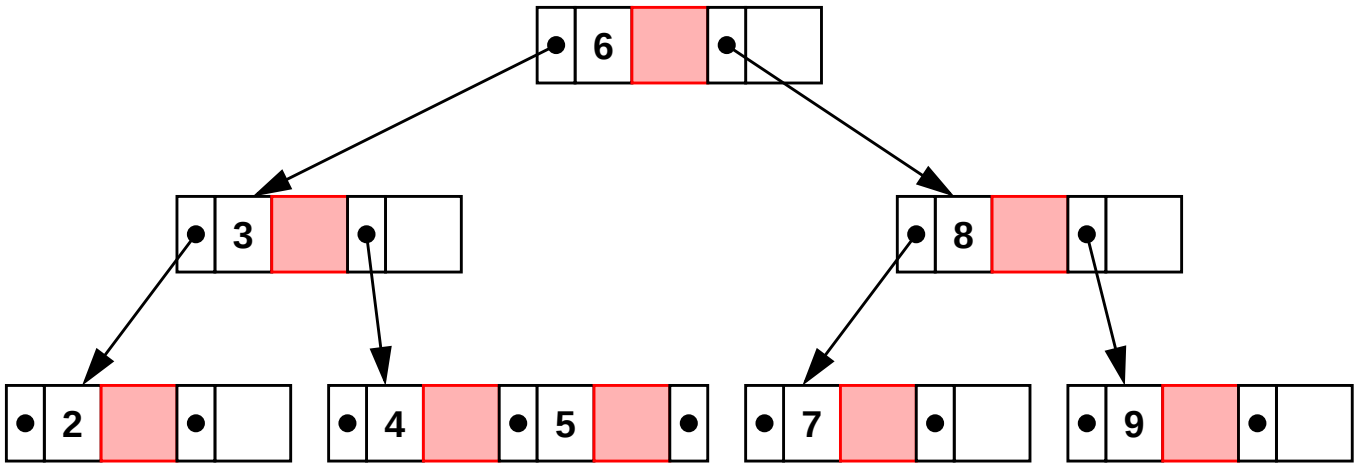


Z_i = Zeiger Sohnseite

S_i = Schlüssel

D_i = Daten des Satzes oder Verweis auf den Satz
(materialisiert oder referenziert)

- **Beispiel:**



bei 4 KB Seiten:

Z=4 B, S=4 B, D=92 B ⇒ 100 B pro Eintrag ⇒ ca. 40 Söhne

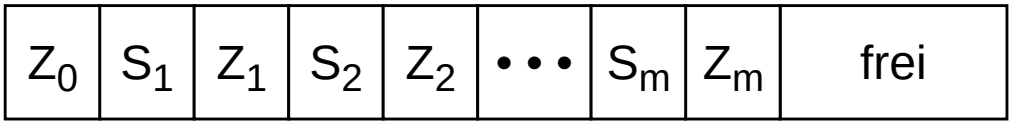
Z=4 B, S=4 B, D=4 B ⇒ 12 B pro Eintrag ⇒ ca. 330 Söhne

B*-Baum

- **Def.: Ein B*-Baum vom Typ (k, k*, h) ist ein Baum mit folg. Eigenschaften**

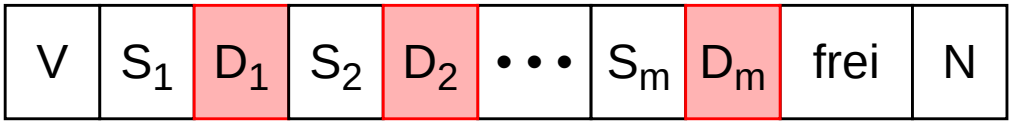
  1. Jeder Weg von der Wurzel zum Blatt hat die Länge h
  2. Jeder Zwischenknoten hat mindestens k+1 Söhne.
     Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.
     Jedes Blatt hat mindestens k* Einträge.
  3. Jeder Zwischenknoten hat höchstens 2k+1 Söhne.
     Jedes Blatt hat höchstens 2k* Einträge.

- **Zwischenknoten:**



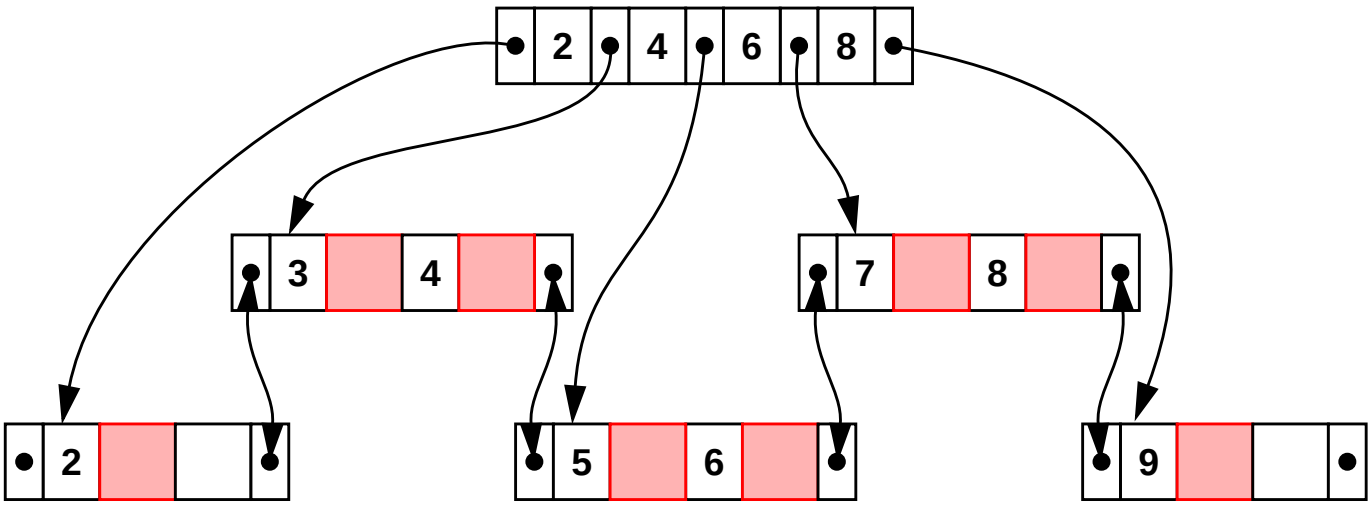
Z_i = Zeiger Sohnseite, S_i = Schlüssel

- **Blattknoten:**



D_i = Verweis auf Satz (materialisiert oder referenziert)

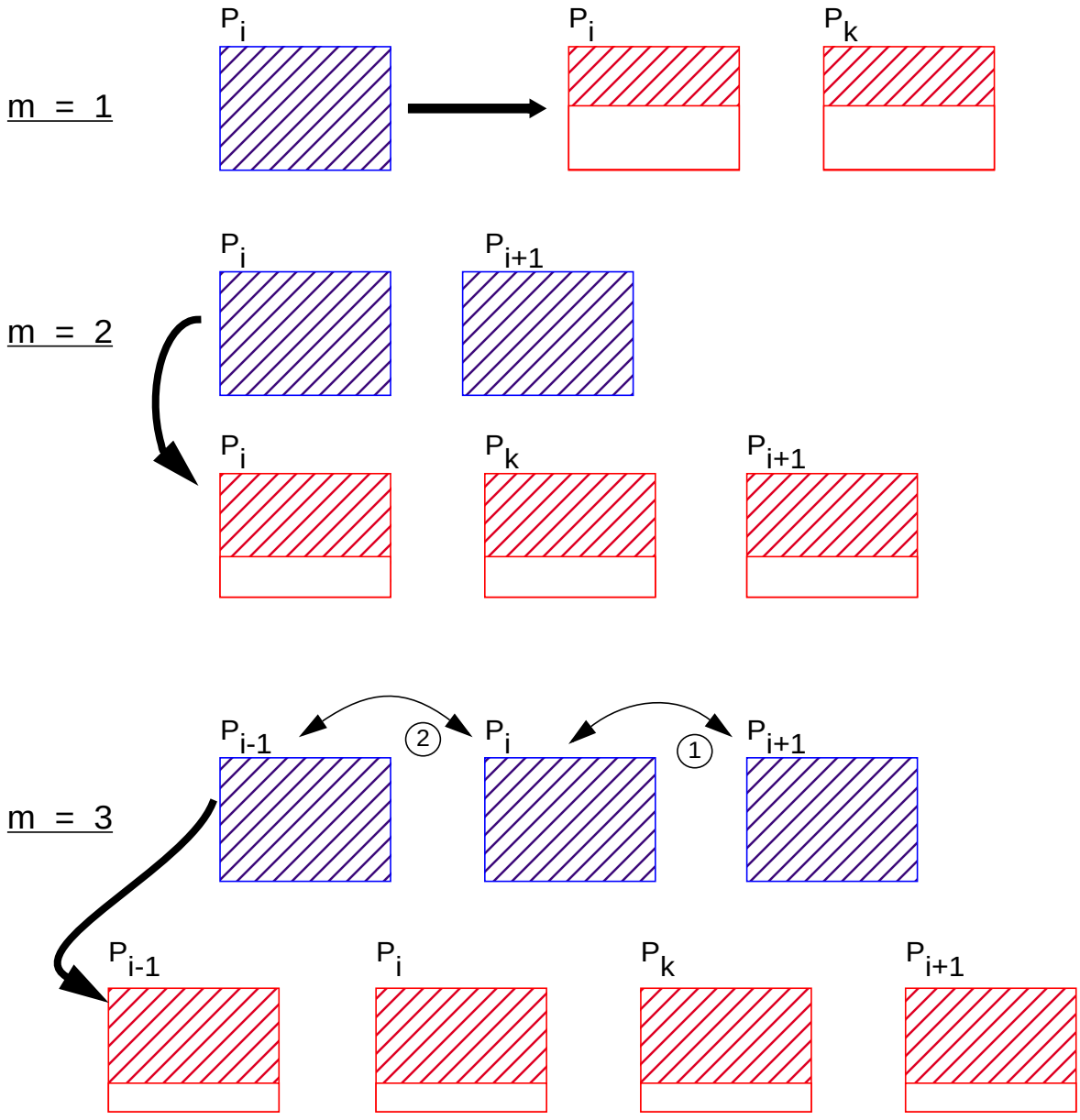
N = Nachfolger-Zeiger, V = Vorgänger-Zeiger



Z=4 B, S=4 B ⇒ 8 B pro Eintrag ⇒ ca. 500 Söhne bei 4 KB Seite

Splitting bei B*-Bäumen

- **Split-Faktor m**



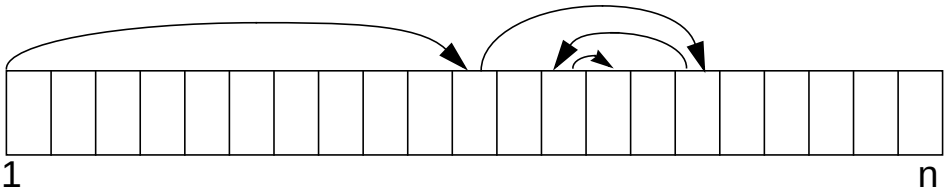
- **Belegung**

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

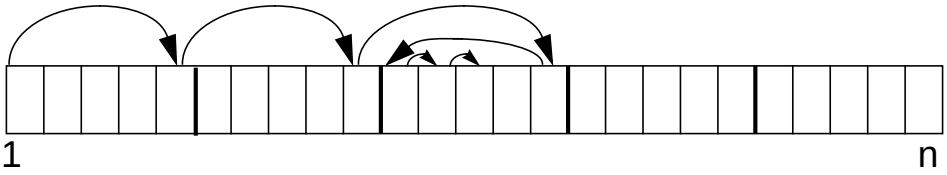
➥ **m ≤ 3: sonst zu aufwendig!**

Suche in der Seite

- **Interne Struktur sei eine Liste mit n Einträgen**
  - sequentielle Suche
    - sortierte oder ungeordnete Schlüsselmenge
    - $C_{avg}(n) \approx n/2$
    - nur geringe Verbesserungen auf sortierten Listen (bei erfolgloser Suche)
  - Binärsuche wesentlich effizienter (Divide-and-Conquer-Strategie)



    - Voraussetzung: Sortierung und Einträge fester Länge
    - $C_{avg}(n) \approx log_2(n+1) - 1$ für große n
  - Sprungsuche
    - Voraussetzung: Sortierung und Einträge fester Länge
    - Prinzip

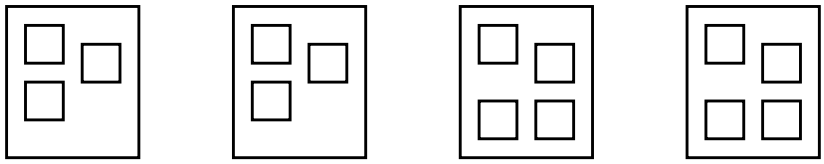


    - zunächst wird Liste in Sprüngen von m Einträgen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält
    - danach wird der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht
    - $C_{avg}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m-1)$

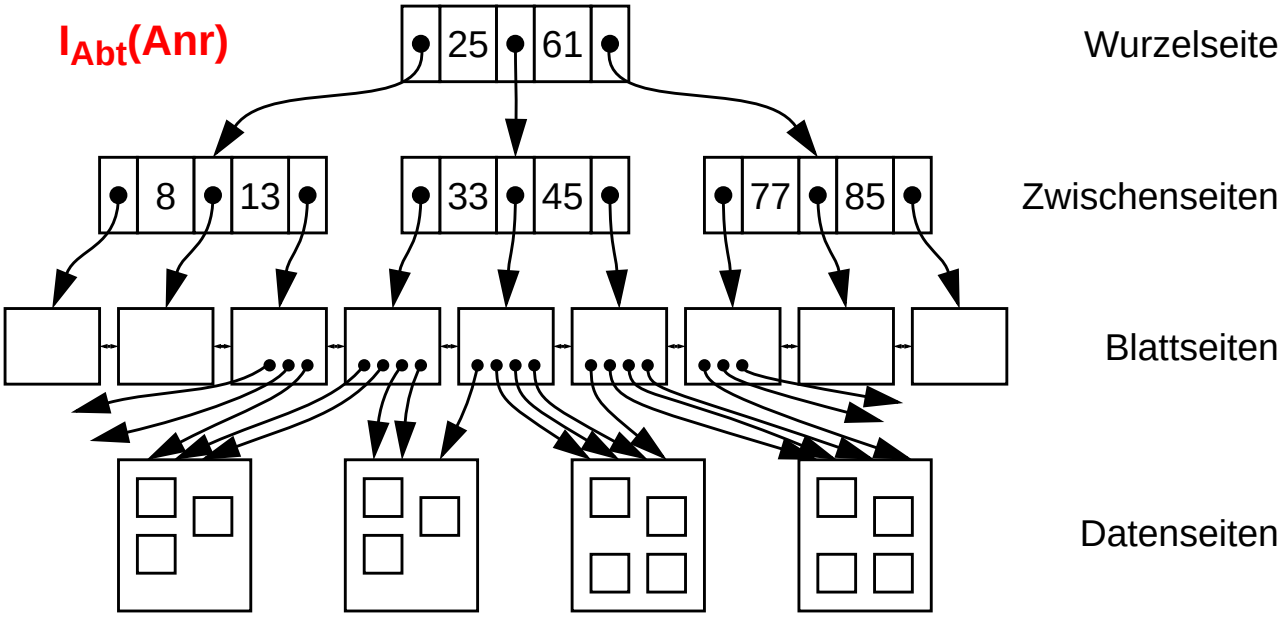
      wenn ein Sprung a und ein sequentieller Vergleich b Einheiten kostet
    - Was ist die optimale Sprungweite m?

Zugriff zu allen Sätzen eines Satztyps

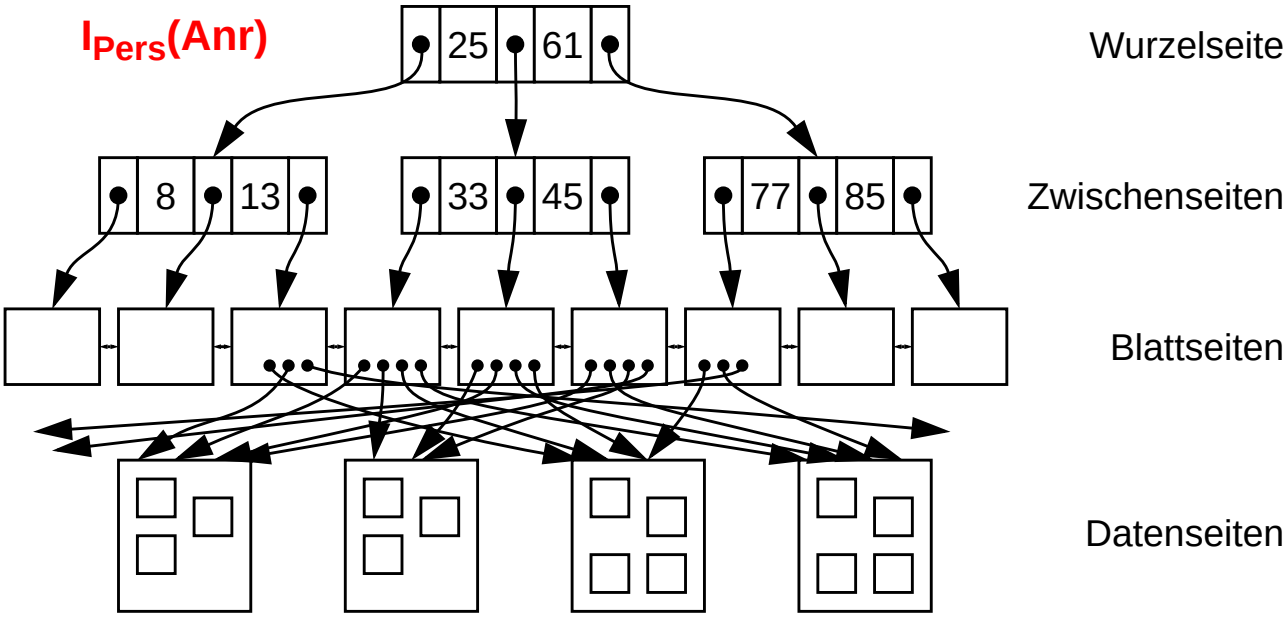
- **Tabellen-Scan**



- **Index-Scan mit Cluster-Bildung**



- **Index-Scan ohne Cluster-Bildung**

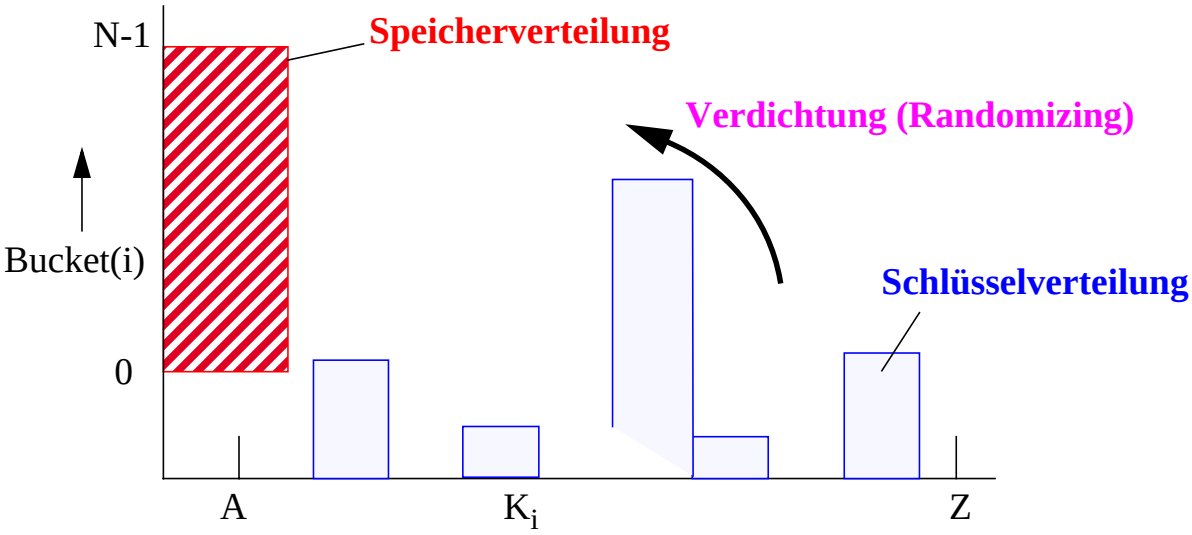


Gestreute Speicherungsstrukturen (Hash-Verfahren)

- **Direkte Berechnung** der Satzadresse über Schlüssel (Schlüsseltransformation)
- **Hash-Funktion**

  h: S $\rightarrow$ {0, 1, ..., N-1} S = Schlüsselraum
  N = Größe des statischen Hash-Bereiches in Seiten (***Buckets***)

- **Idealfall: h ist injektiv (keine Kollisionen)**
  - nur in Ausnahmefällen möglich ('dichte' Schlüsselmenge)
  - jeder Satz kann mit einem Seitenzugriff gefunden werden
- **Statische Hash-Bereiche mit Kollisionsbehandlung**
  - vorhandene Schlüsselmenge K (K $\subseteq$ S) soll möglichst gleichmäßig auf die N Buckets verteilt werden



  - Behandlung von Synonymen
    - Aufnahme im selben Bucket, wenn möglich
    - ggf. Anlegen und Verketten von Überlaufseiten
  - typischer Zugriffsfaktor: 1.1 bis 1.4
- **Vielzahl von Hash-Funktionen anwendbar**

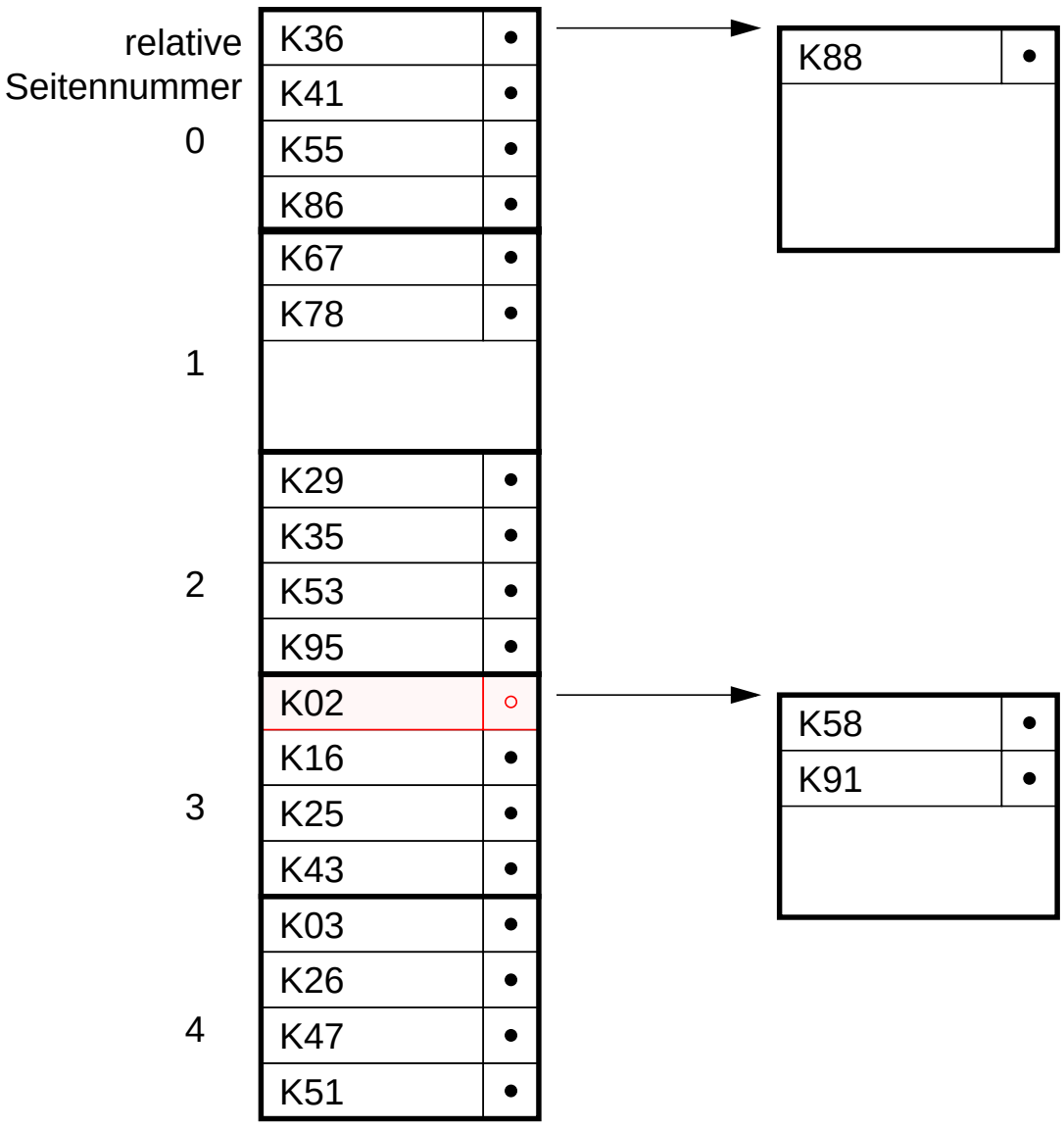
  z. B. Divisionsrestverfahren, Faltung, Codierungsmethode, ...

Statisches Hash-Verfahren mit Überlaufbereichen: Beispiel

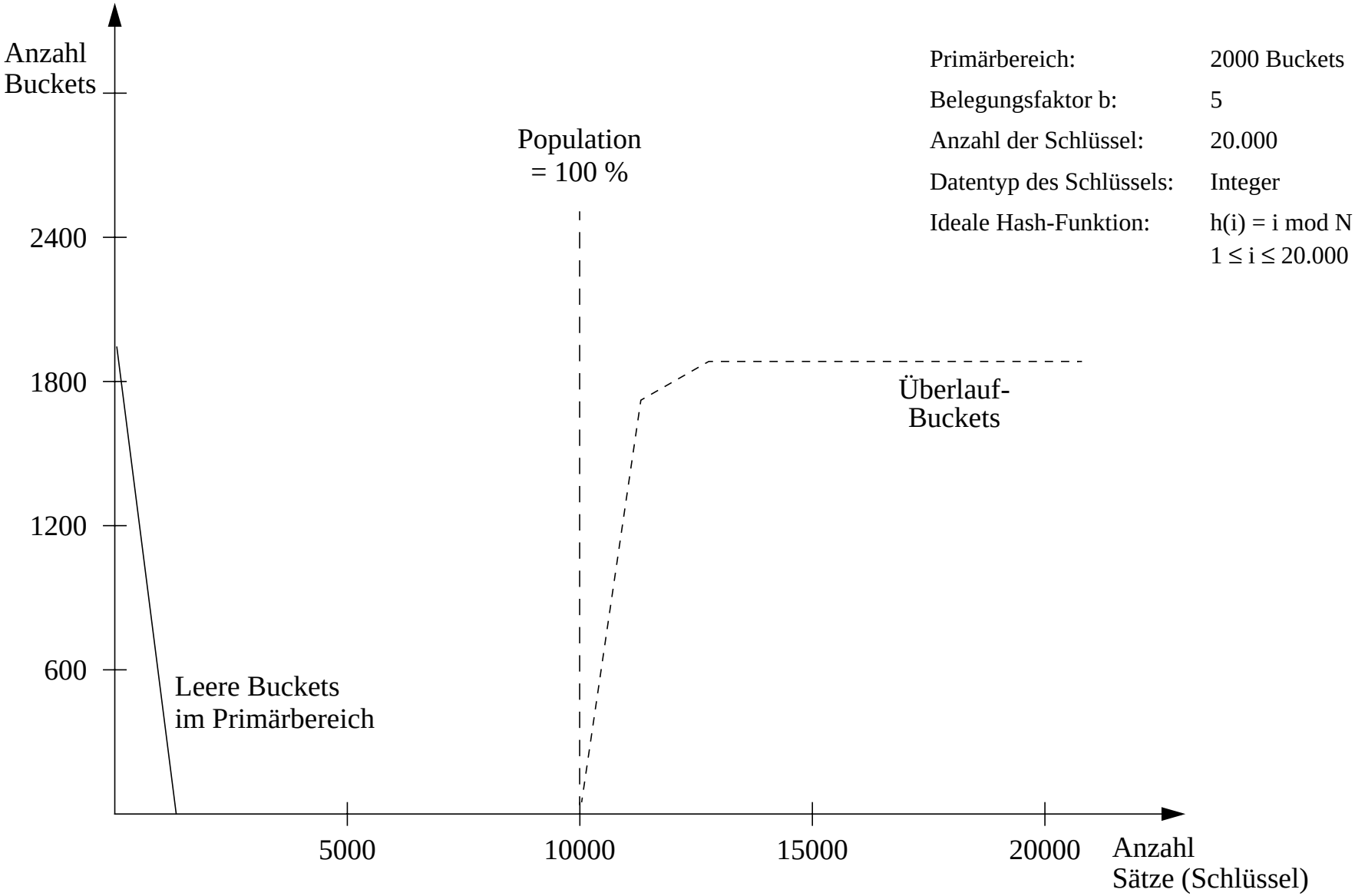
- **Adreßberechnung für Schlüssel K02:**

$$
\begin{array}{l}
\quad 1101\ 0010 \\
\oplus\ 1111\ 0000 \\
\oplus\ 1111\ 0010 \\
\hline
\quad 1101\ 0000 = 208_{10}
\end{array}
$$

208 mod 5 = 3

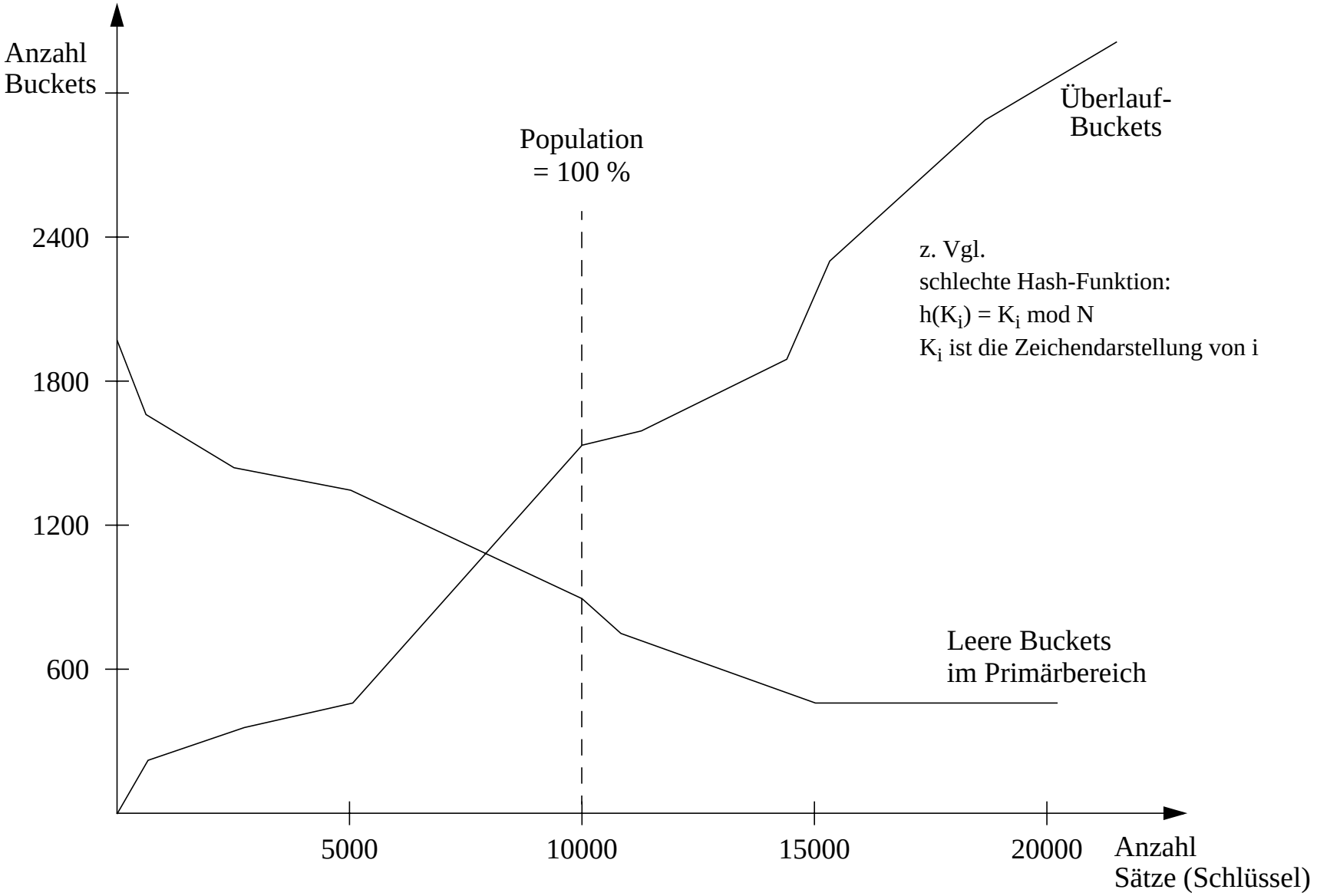


Belegung von Hash-Bereichen – Messung (1)



Primärbereich:	2000 Buckets
Belegungsfaktor b:	5
Anzahl der Schlüssel:	20.000
Datentyp des Schlüssels:	Integer
Ideale Hash-Funktion:	h(i) = i mod N $1 \leq i \leq 20.000$

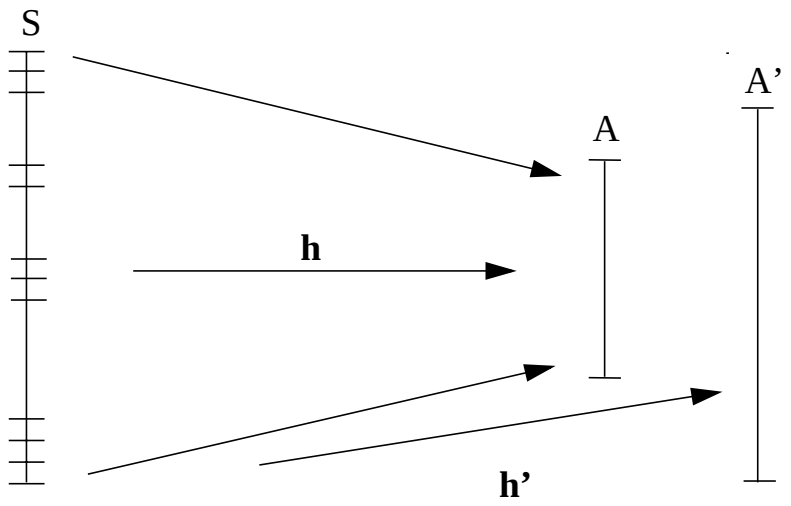
Belegung von Hash-Bereichen – Messung (2)



Dynamische Hash-Verfahren

- **Wachstumsproblem bei statischen Verfahren**
  - Statische Allokation von Speicherbereichen: Speicherausnutzung?
  - Bei Erweiterung des Adreßraums: Rehashing

  ➥ Kosten, Verfügbarkeit, Adressierbarkeit



  ➥ Alle Sätze erhalten eine **neue Adresse**

- **Entwurfsziele**
  - Dynamische Struktur erlaubt Wachstum und Schrumpfung des Hash-Bereichs (Datei)
  - Keine Überlauftechniken
  - Zugriffsfaktor ≤ 2 für die direkte Suche

- **Viele konkurrierende Ansätze**
  - Extensible Hashing (Fagin et al., 1978)
  - Virtual Hashing und Linear Hashing (Litwin, 1978, 1980)
  - Dynamic Hashing (Larson, 1978)

  ➥ Lösungsvorschläge **mit und ohne Index** (Hilfsdaten)

  „Any hashing which may dynamically change its hashing function".

Erweiterbares Hashing

- **Lösungsidee: Verknüpfung der**
  - von den B-Bäumen bekannten Split- und Mischtechniken von Seiten zur Konstruktion eines dynamischen Hash-Bereichs mit der
  - von den Digitalbäumen her bekannten Adressierungstechnik zum Aufsuchen eines Speicherplatzes

- **Prinzipielle Vorgehensweise**
  - Die einzelnen Bits eines Schlüssels steuern den Weg durch den zur Adressierung benutzten Digitalbaum.
  - $K_i = (b_0, b_1, b_2, \ldots)$. Es ist prinzipiell möglich, die Bitfolge von K_i direkt für die Adressierung heranzuziehen. Bei Ungleichverteilung der Schlüssel ist dann ein unausgewogener Digitalbaum zu erwarten.
  - Da Digitalbäume keinen Balancierungsmechanismus für ihre Höhe besitzen, muß die Ausgewogenheit “von außen” aufgezwungen werden.
  - $h(K_i) = (b_0, b_1, b_2, \ldots)$. Die Verwendung von $h(K_i)$ als sog. Pseudoschlüssel (PS) soll bessere Gleichverteilung gewährleisten.

- **Gleichverteilung der Pseudoschlüssel PS**

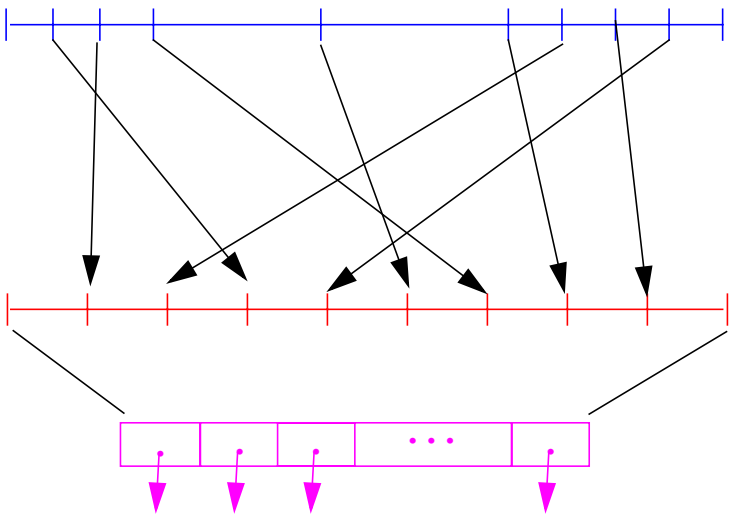
  impliziert minimale Höhe des Digitalbaumes

Ungleichverteilung der Schlüssel K

$h(K_i) \longrightarrow$ PS

Gleichverteilung der PS

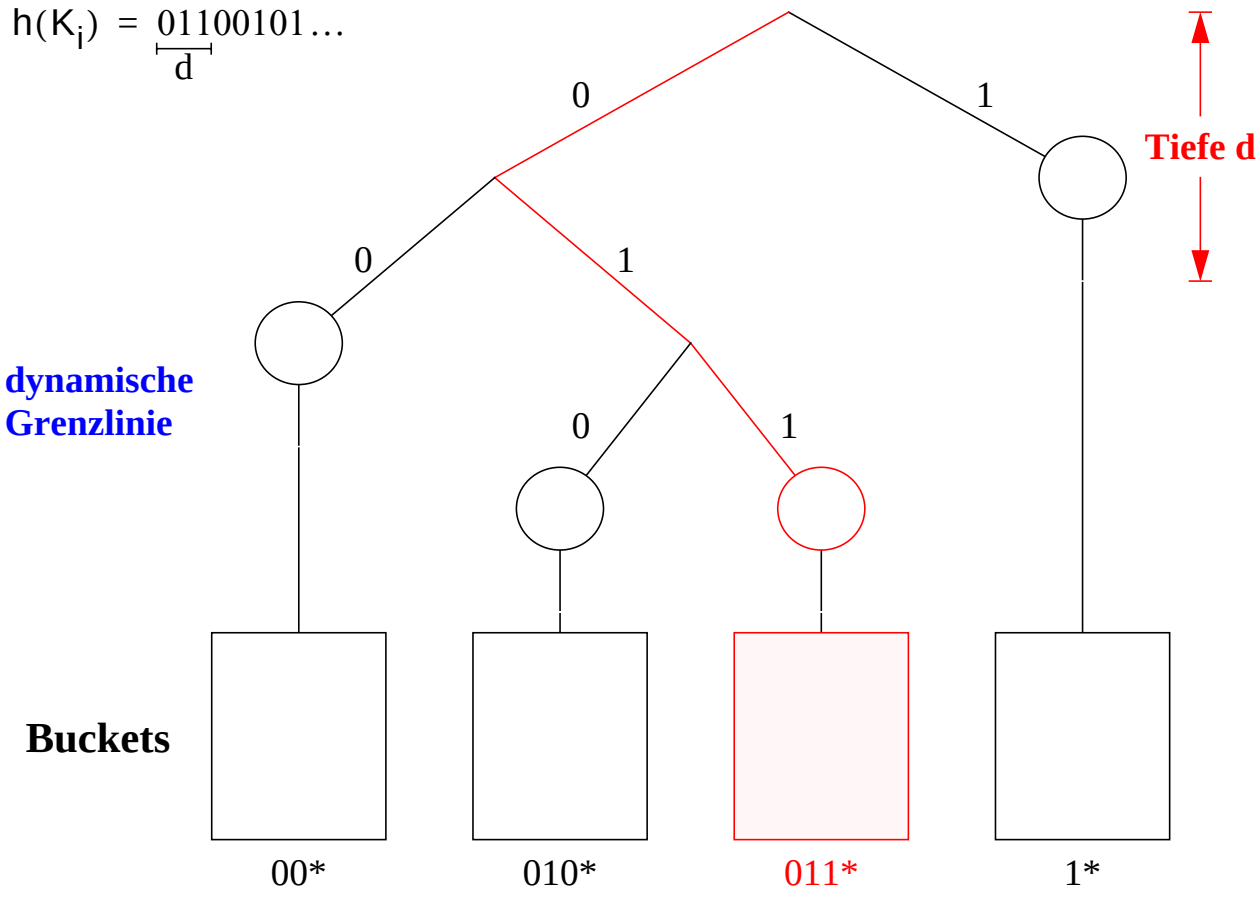
PS werden abgebildet auf Directory



Buckets

Erweiterbares Hashing[1] (3)

- **Prinzipielle Abbildung der Pseudoschlüssel**



  - Zur Adressierung eines Buckets sind d Bits erforderlich, wobei sich dafür i. allg. eine dynamische Grenzlinie variierender Tiefe ergibt.
  - Die Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann.

  ➥ ausgeglichener Digitalbaum garantiert minimales d_{max}

- **Dynamisches Wachsen und Schrumpfen des Hash-Bereiches**

  - Buckets werden erst bei Bedarf bereitgestellt
  - Knoten unterschiedlicher Tiefe verweisen auf ein Bucket

  ➥ **hohe Speicherplatzbelegung möglich**

---

1. Fagin, R., et. al: Extendible hashing– a fast access method for dynamic files. ACM Trans. Database Syst. 4:3. 1979. 315-344

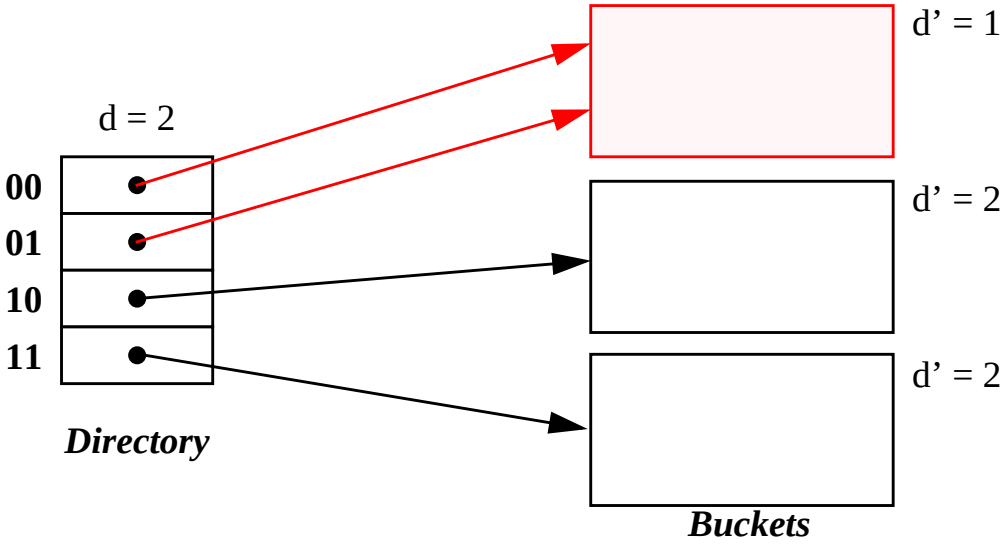
Erweiterbares Hashing (4)

- **Verfahren benötigt keine Überlaufbereiche, jedoch erfolgt Zugriff über *Directory* (Index)**

  - Binärer Digitalbaum der Höhe d wird durch einen (2^d)-Digitalbaum der Höhe 1 implementiert (Trie der Höhe 1 mit 2^d Einträgen)
  - d wird festgelegt durch den längsten Pfad im binären Digitalbaum
  - In einem Bucket werden nur Sätze gespeichert, deren PS in den ersten d' Bits übereinstimmen (d' = lokale Tiefe)
  - d = MAX (d'): d Bits des PS werden zur Adressierung verwendet (d = globale Tiefe)
  - Directory enthält 2^d Einträge

- **Speicherungsstruktur**

  Der Trie läßt sich als Directory oder Adreßverzeichnis auffassen. Die d Bits von $h(K_i)$ zeigen im Directory auf einen Eintrag mit der Adresse des Buckets, das den Schlüssel K_i enthält. Wenn d' < d, können (benachbarte) Einträge auf dasselbe Bucket verweisen.



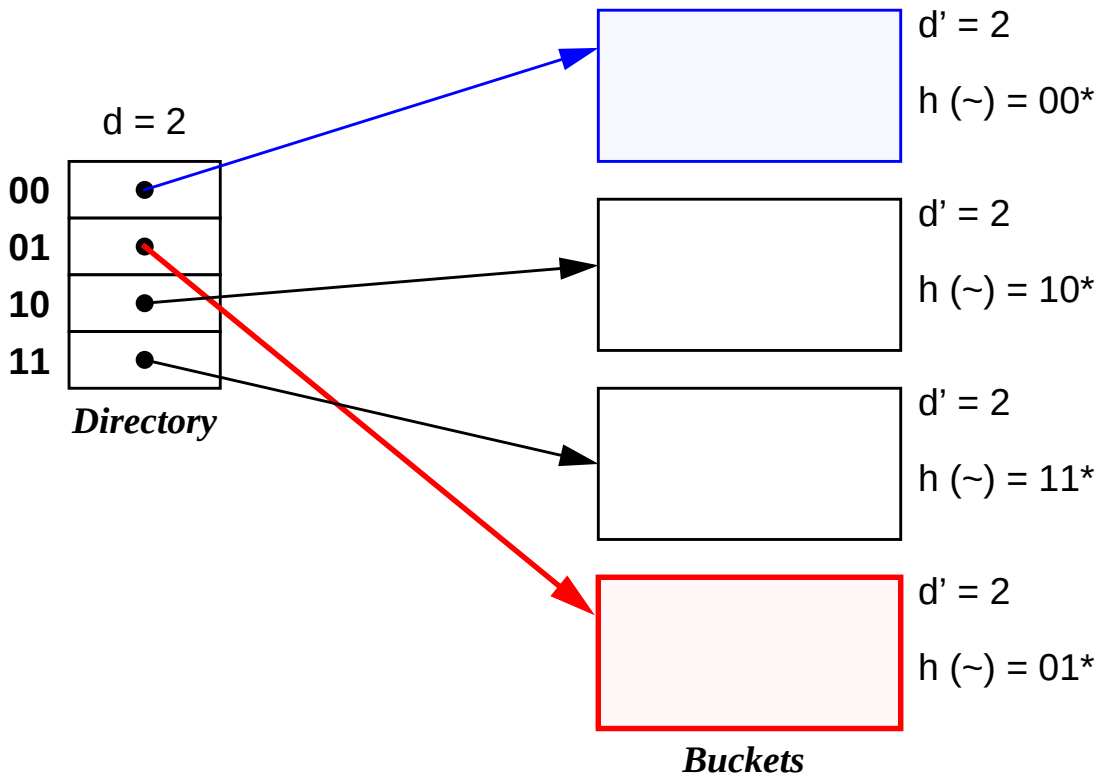
➥ Kosten der direkten Suche: **max. 2 Seitenzugriffe**

Erweiterbares Hashing: Splitting von Buckets (1)

- **Fall 1: Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist**

➥ Anlegen eines neuen Buckets (Split) mit

- lokaler Neuverteilung der Daten
- Erhöhung der lokalen Tiefe
- lokaler Korrektur der Verweise im Directory

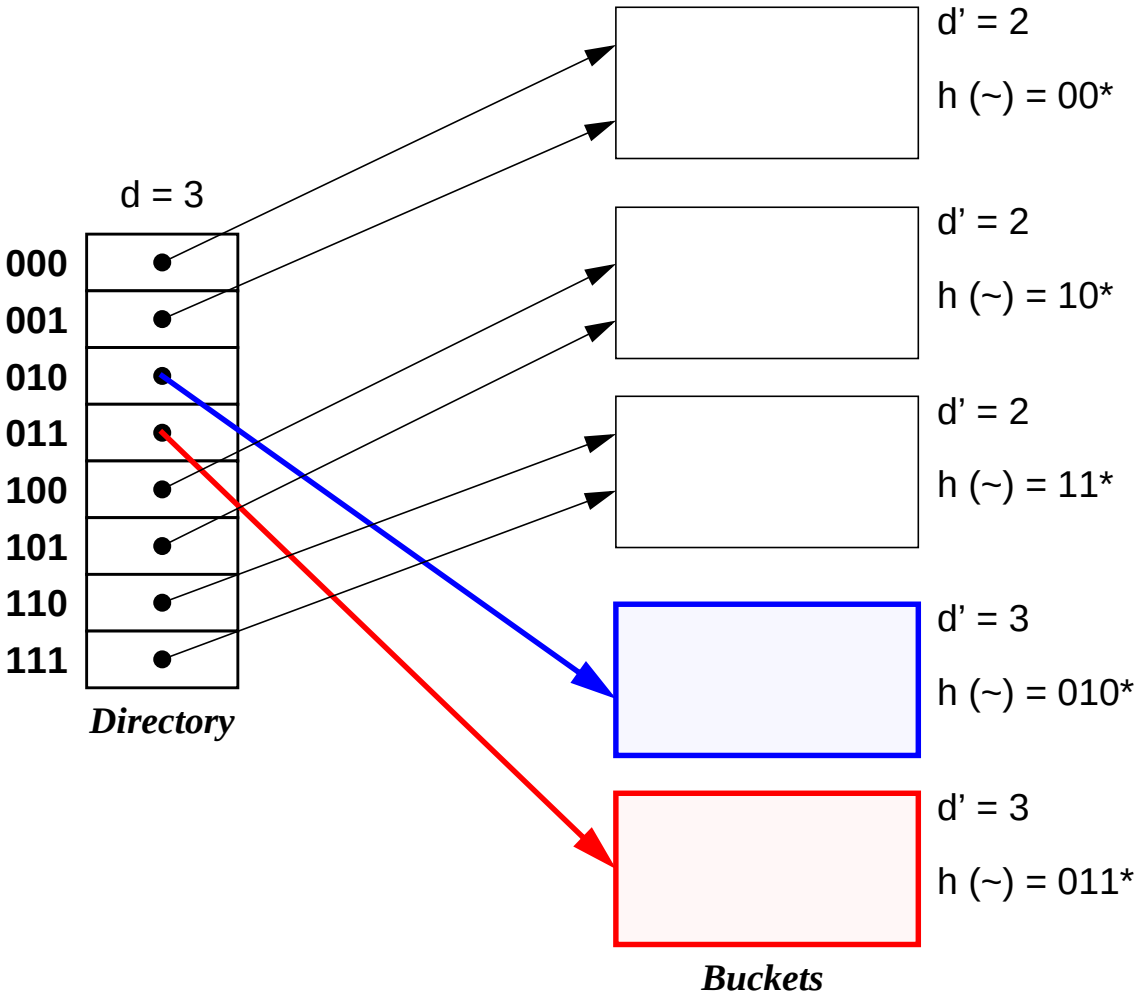


Erweiterbares Hashing: Splitting von Buckets (2)

- **Fall 2: Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist**

➥ Anlegen eines neuen Buckets (Split) mit

- lokaler Neuverteilung der Daten (Erhöhung der lokalen Tiefe)
- Verdopplung des Directories (Erhöhung der globalen Tiefe)
- globaler Korrektur/Neuverteilung der Verweise im Directory



Vergleich der wichtigsten Zugriffsverfahren

Zugriffsverfahren	Speicherungsstruktur	Direkter Zugriff	Sequentielle Verarbeitung	Änderungsdienst (Ändern ohne Aufsuchen)
Fortlaufender Schlüsselvergleich	Sequentielle Liste	$O(n) \approx 10^4$	$O(n) \approx 2 \cdot 10^4$	$O(1) \leq 2$
	Gekettete Liste	$O(n) \approx 5 \cdot 10^5$	$O(n) \approx 10^6$	$O(1) \leq 3$
Baumstrukturierter Schlüsselvergleich	Balancierte Binärbäume	$O(\log_2 n) \approx 20$	$O(n) \approx 10^6$	$O(1) = 2$
	Mehrwegbäume	$O(\log_k n) \approx 3 – 4$	$O(n) \approx 10^6$*	$O(1) = 2$
Konstante Schlüsseltrans-formationsverfahren	Externes Hashing mit separatem Überlaufbereich	$O(1) \approx 1.1 – 1.4$	$O(n \log_2 n)$**	$O(1) \approx 1.1$
	Externes Hashing mit Separatoren	$O(1) = 1$	$O(n \log_2 n)$**	$O(1) = 1$ (+D)
Variable Schlüsseltrans-formationsverfahren	Erweiterbares Hashing	$O(1) = 2$	$O(n \log_2 n)$**	$O(1) \approx 1.1$ (+R)
	Lineares Hashing	$O(1) = 1$	$O(n \log_2 n)$**	$O(1) < 2$

* Bei Clusterbildung bis zu Faktor 50 geringer

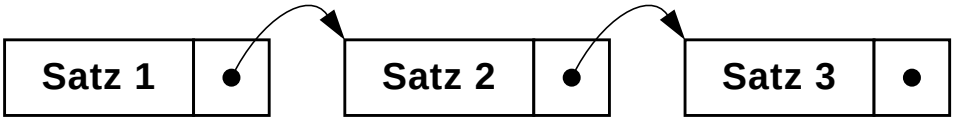
** Physisch sequentielles Lesen, Sortieren und sequentielles Verarbeiten der gesamten Sätze
Beispielangaben für $n = 10^6$

Verknüpfungsstrukturen für Satzmengen

- **Materialisierte Speicherung**

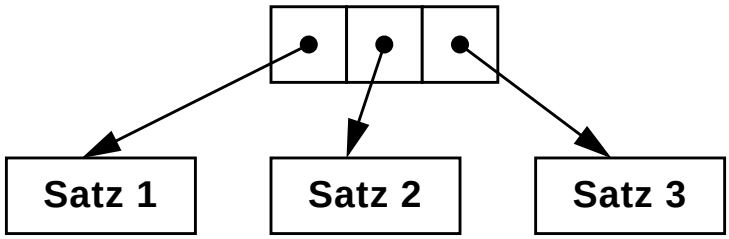
1. Physische Nachbarschaft der Sätze (Cluster-Bildung, Listen)

2. Verkettung der Sätze

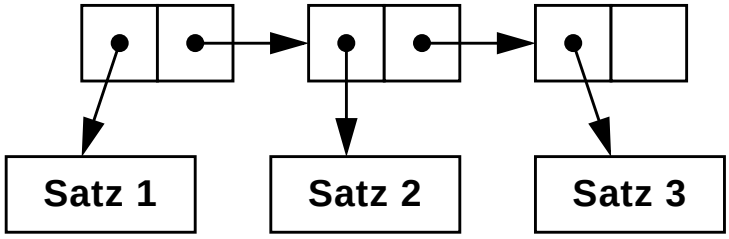


- **Referenzierte Speicherung**

3. Physische Nachbarschaft der Zeiger (Invertierung)



4. Verkettung der Zeiger



Zugriffspfade für Sekundärschlüssel

- **Suche nach Sätzen mit einem vorgegebenen Wert**
  eines nicht-identifizierenden Attributs (*Sekundärschlüssel*)

- **Ergebnis ist Satzmenge**

- **Realisierung: Einstiegsstruktur + Verknüpfungsstruktur**

  - Primärschlüssel-Zugriffspfade als Einstiegsstruktur auf Satzmengen anwendbar
  - Prinzipiell lassen sich alle Verknüpfungsstrukturen für Satzmengen heranziehen

  ➥ vor allem: Verwendung von B*-Bäumen und Invertierungstechniken

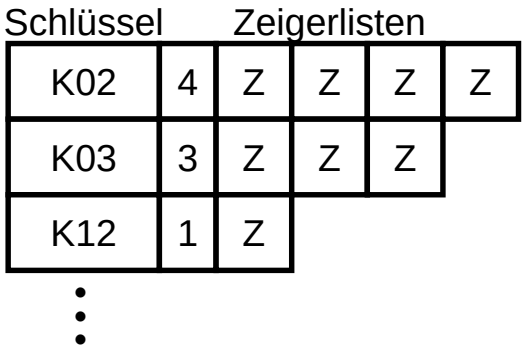
- **Standardlösung bei der Invertierung** sind sequentielle Verweislisten
  (oft OID-Listen oder TID-Listen genannt)

  - effiziente Durchführung von Mengenoperationen
  - kosteneffektive Wartung

Zugriffspfade für Sekundärschlüssel (2)

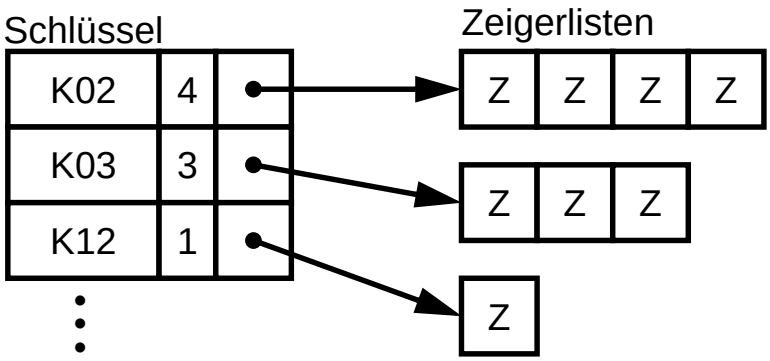
- **Häufige Realisierung: Invertierung**
  - Trennung der Zugriffspfaddaten von den Datensätzen (referenzierte Speicherung)
  - Verweis Z realisiert als TID, DBK/PPP, ...
  - Es kommen zwei Darstellungsmethoden in Betracht:

a) **Gemeinsame Verwaltung** der Suchstruktur und der Zeigerlisten



➥ relativ kurze Zeigerlisten erforderlich!

b) In der Suchstruktur ist (ähnlich wie bei Zugriffspfaden für Primärschlüssel) nur ein Verweis pro Schlüsselwert vorhanden, der zu einer Liste mit Satzverweisen führt (Zeigerliste)



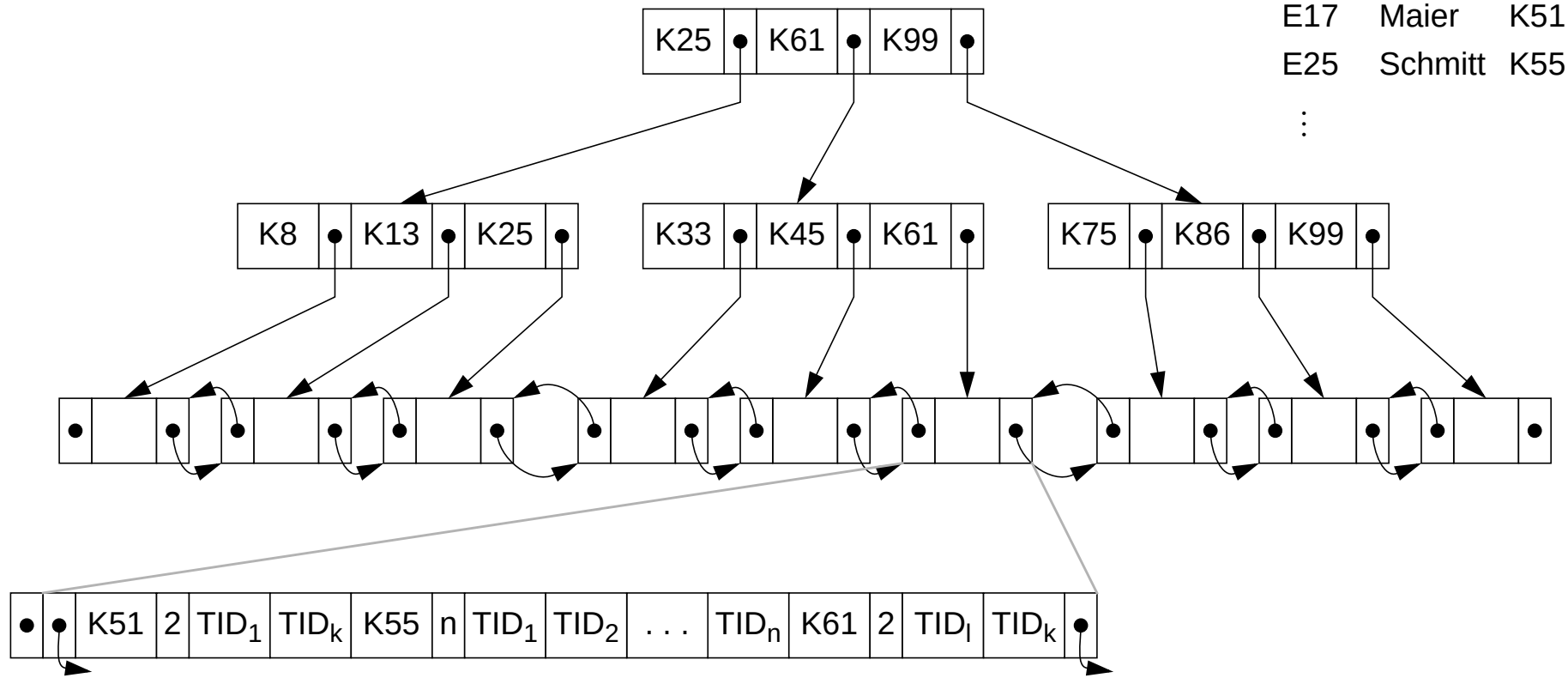
➥ Zeigerlisten können in separaten „Behältern” abgelegt werden

Zugriffspfade für Sekundärschlüssel (3)

Tabelle Pers

Pers (	Pnr,	Name,	Anr,	...)
	E1	Müller	K55	...
	E17	Maier	K51	
	E25	Schmitt	K55	
	⋮			

I_{Pers}(Anr)

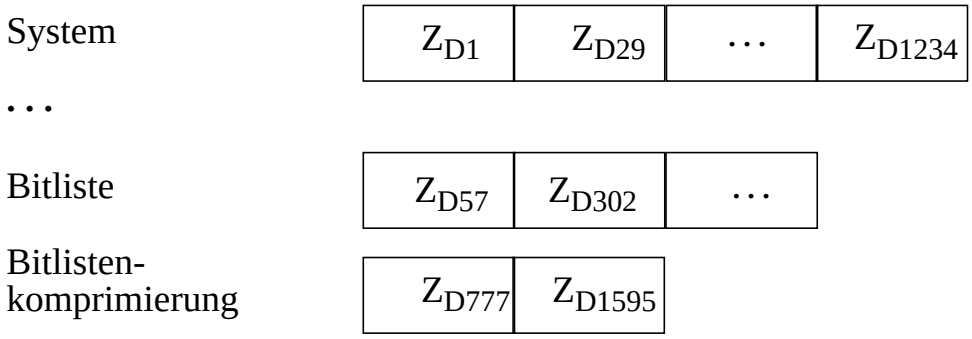


B*-Baum

- als Zugriffspfad für Sekundärschlüssel Anr
- mit Sortierreihenfolge der Sekundärschlüssel (Bereichsfragen!) sowie Vorwärts- und Rückwärtsverkettung

Zugriffspfade für Sekundärschlüssel (4)

- **Einsatzmöglichkeit auch beim Information Retrieval**
  - unformatierte Daten: Dokumente
  - Invertierung mit Hilfe von **Deskriptoren** (keine Zuordnung zu Attributen!)



➥ Es sind sehr viele und sehr wenige Verweise möglich

- **Invertierung mit Bitlisten**
  - Adressierung von Datensätzen oder Dokumenten
    - über Zuordnungstabelle ZT
    - direkt bei fester Länge und fortlaufender Speicherung
  - Markierungen der Bitliste entsprechen Einträgen von ZT oder berechenbaren Adressen (b Sätze pro Seite)
  - Attribut A habe j Attributwerte a_1, ..., a_j

- **Bitmatrix für A**

	1 2 3 . . .	**n**
$\mathbf{a_1}$	**0 1 0 0 1 0 0 . . .**	
$\mathbf{a_2}$	**1 0 0 0 0 0 1**	
	. . .	
$\mathbf{a_j}$	**0 0 0 1 0 1 0**	

➥ **Speicherung als vertikale Bitlisten** erlaubt Indexierung von mehrwertigen Attributen (Bsp: Warenkorb mit Produkten)

Zugriffspfade für Sekundärschlüssel (5)

- **Bitlisten fester Länge**
  - j_i Bitlisten von Attribut A_i
  - einfache Änderungsoperationen
  - schneller Vergleich
  - sehr speicherplatzaufwendig
  - nur für kleine j

    ➥ oft lange Nullfolgen: Komprimierung

- **Komprimierte Bitlisten variabler Länge**
  - Speicherplatzeinsparung
  - Reduktion der E/A-Zeit
  - Mehraufwand für Codierung und Decodierung
  - schneller Vergleich
  - umständliche Änderungsoperationen

- **Anwendungsgebiete der Komprimierung**
  - Data Warehouse (Invertierung der Faktentabelle)
  - Übertragung/Speicherung
    - von Multimedia-Objeklten (Image, Audio, Video, ...)
    - von dünn besetzten Matrizen
    - von Objekten in Geo-DB, ...

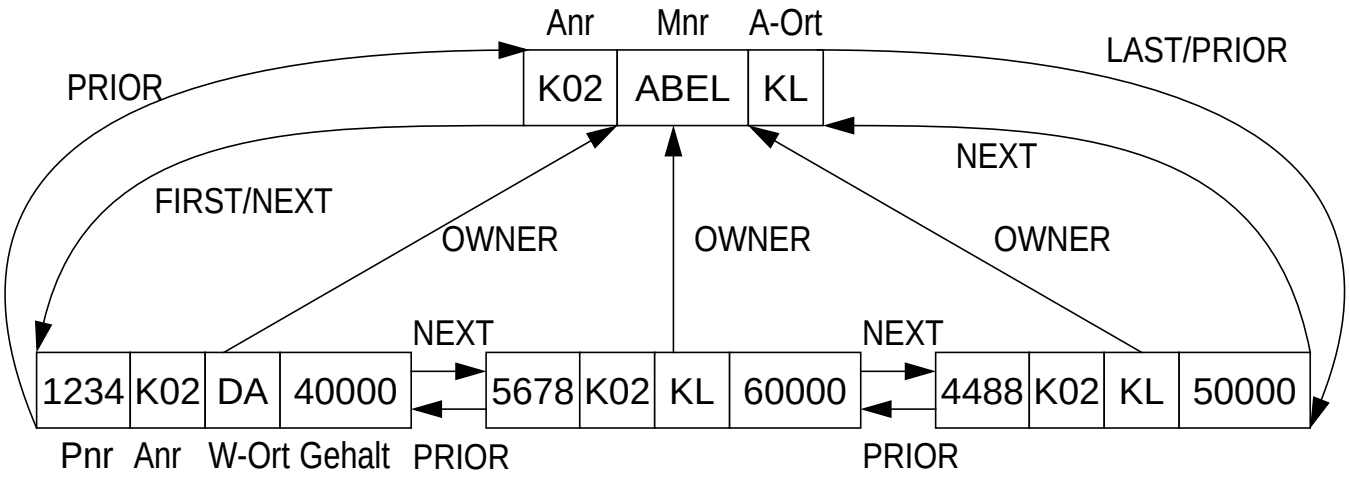
- **Viele Komprimierungstechniken verfügbar**
  - Laufkomprimierung (run length compression)
  - Nullfolgen-Komprimierung
  - Mehr-Modus-Komprimierung
  - Blockkomprimierung

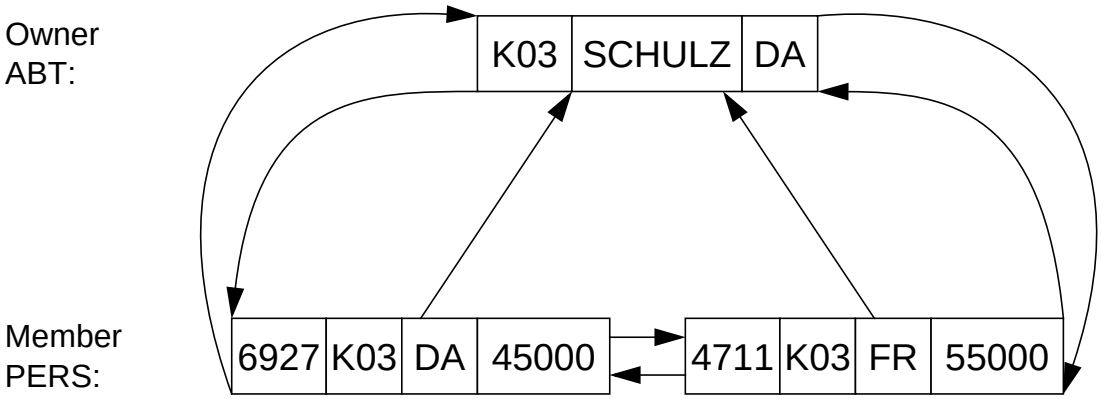
Hierarchische Zugriffspfade

- **Realisierung funktionaler Beziehungen zwischen zwei Satztypen**
  - Owner --> Member: Set-Typen nach dem Netzwerkmodell
  - Jede Ausprägung einer Owner-Satzart wird mit 0..n Ausprägungen der Member-Satzart verknüpft

Logische Sicht:

Darstellung der Navigationsmöglichkeiten

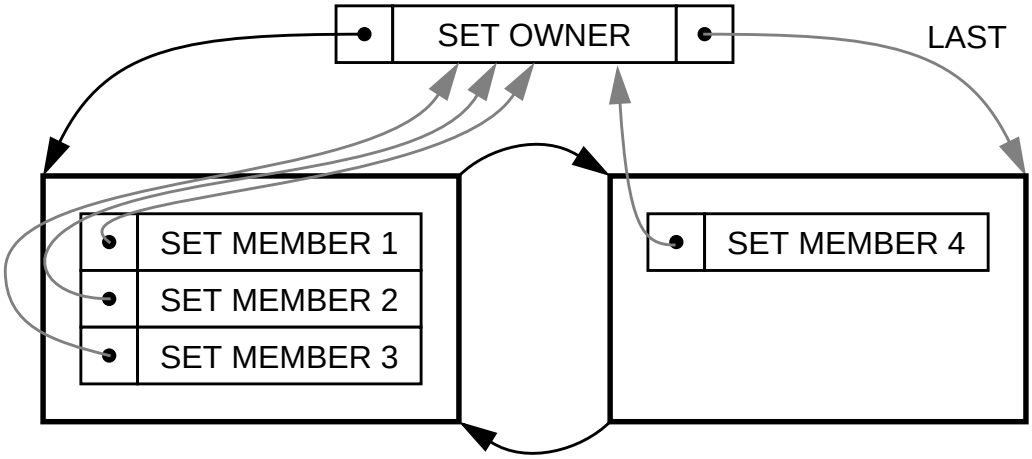




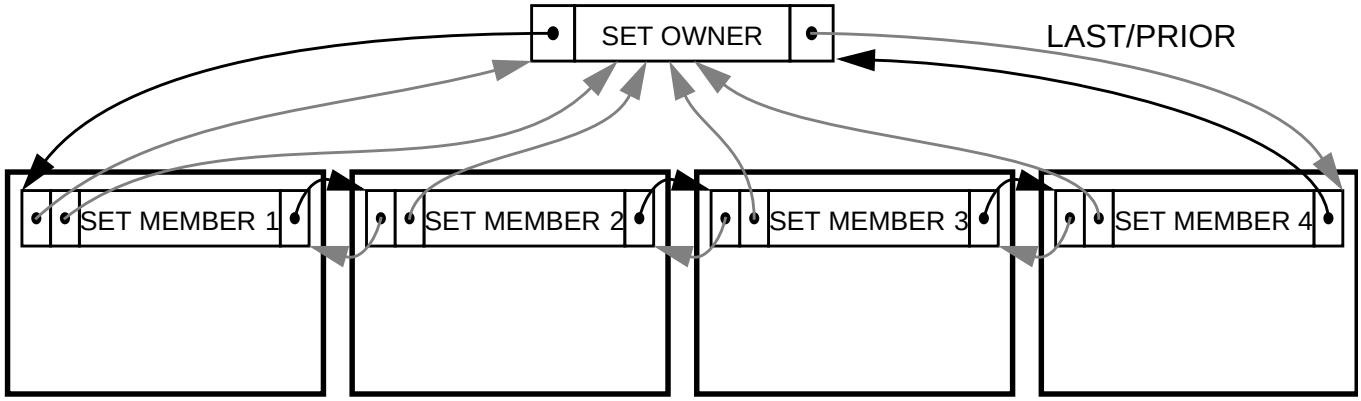
➥ drei Implementierungen für unterschiedliche Leistungsanforderungen

Hierarchische Zugriffspfade – Implementierung

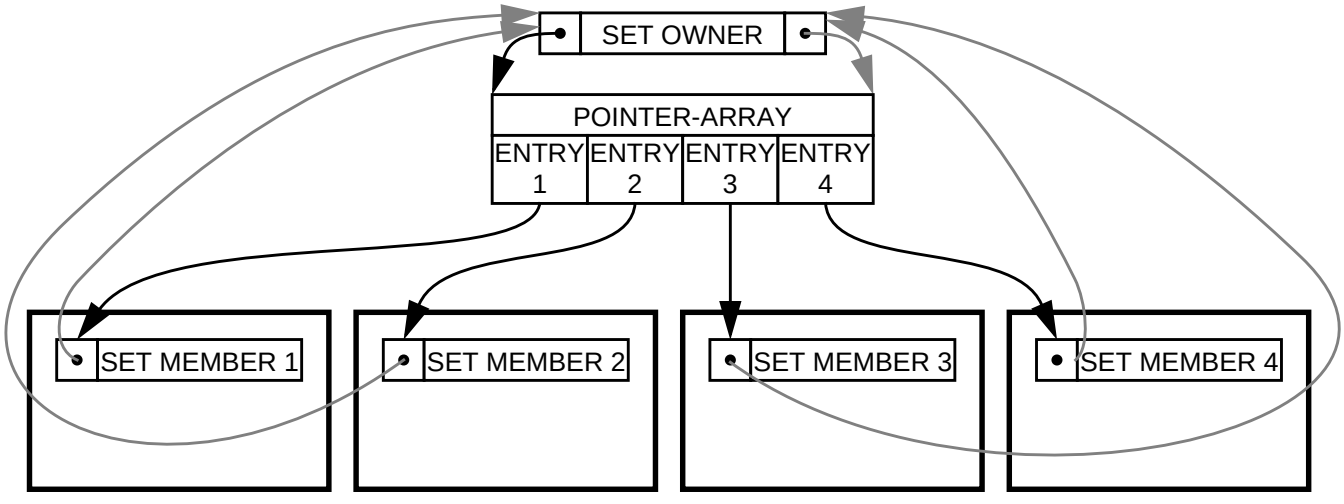
- **Sequentielle Liste auf Seitenbasis**



- **Gekettete Liste**



- **Pointer-Array-Struktur**



: optionaler Zeiger

Hierarchische Zugriffspfade – Bewertung der Implementierungstechniken

- **Pointer-Array**
  - stabiles Verhalten
  - Verhalten unabhängig vom Set-Wachstum und Set-Reihenfolge
  - 'Standard'-Verfahren bei unscharfen Informationen über Set-Größe und Zugriffshäufigkeit

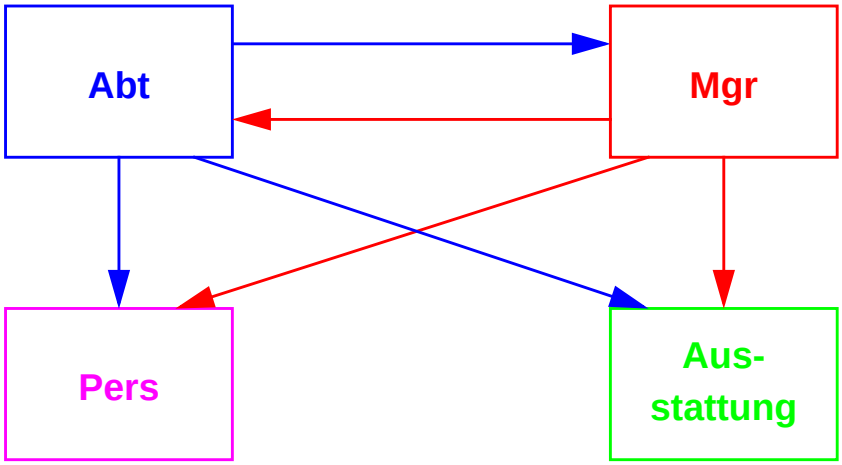
- **Sequentielle Liste**
  - auf einen Set-Typ pro Member-Satztyp beschränkt (Cluster-Bildung)
  - schnelles Aufsuchen / Einfügen in Set-Reihenfolge
  - Ändern teurer als bei Pointer-Array

- **Gekettete Liste**
  - Vorteile bei Mitgliedschaft des Member-Satztyps in mehreren Sets
  - billiger Wechsel auf andere Set-Ausprägungen
  - sequentieller Zugriff schneller als bei Pointer-Array
  - nur gut in kleinen Set-Ausprägungen

Verallgemeinerte Zugriffspfadstruktur

- **Idee:**

  gemeinsame Verwendung einer Indexstruktur (B*-Baum)
  für mehrere Satztypen, für die Beziehungen (1:1, 1:n, n:m)
  über demselben Wertebereich (z. B. für Anr) definiert und
  durch Gleichheit von Attributwerten repräsentiert sind

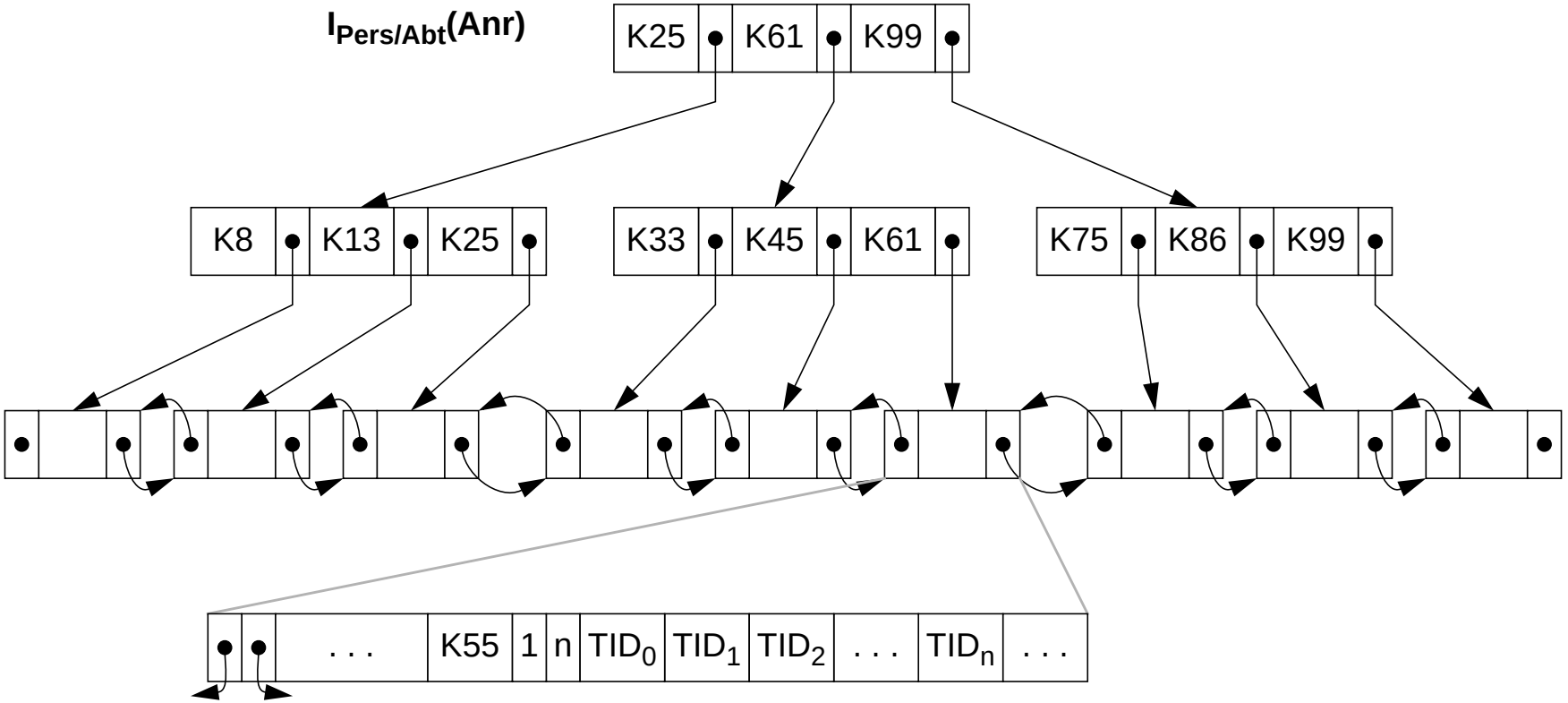


Alle Tabellen besitzen ein Attribut (z. B. Anr), das auf dem Wertebereich Abtnr definiert ist

- **Nutzung der Indexstruktur** für
  - Primärschlüsselzugriff z. B. als I_{Abt}(Anr)
  - Sekundärschlüsselzugriff z. B. als I_{Pers}(Anr)
  - hierarchischen Zugriff z. B. von Abt(Anr) nach Pers(Anr)
    oder in umgekehrter Richtung
  - Verbundoperationen (Join) z. B. von Abt.Anr = Pers.Anr

- **Kombinierte Realisierung** von Primärschlüssel-, Sekundärschlüssel- und hierarchischen Zugriffspfaden mit einem erweiterten B*-Baum
  - Innere Baumknoten bleiben unverändert
  - Blätter enthalten Verweise für primäre und sekundäre Zugriffspfade

B*-Baum als kombinierte Zugriffspfadstruktur

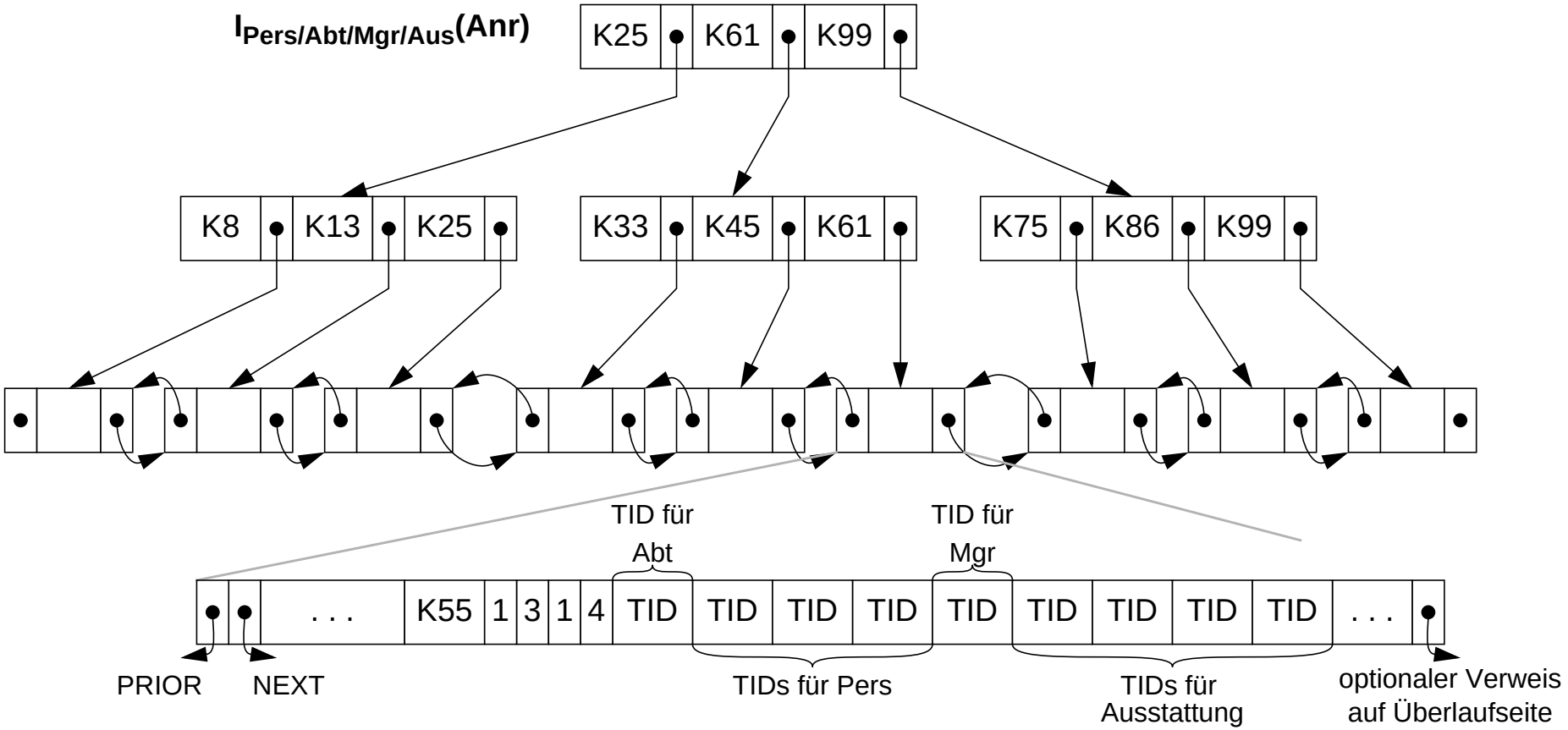


Struktur enthält Index für Abt, Pers und

Link für Abt-Pers mit direktem Zugriff von

1. OWNER zu jedem MEMBER,
2. jedem MEMBER zu jedem anderen MEMBER,
3. jedem MEMBER zum OWNER

B*-Baum als verallgemeinerte Zugriffspfadstruktur



Zugriffspfadstruktur umfaßt

- 4 Index-Strukturen
- 6 Link-Strukturen

Verallgemeinerte Zugriffspfadstruktur – Bewertung

- **Schlüssel werden nur einmal gespeichert**

  ➥ Speicherplatzersparnis

- **Einheitliche Struktur für alle Zugriffspfadtypen**

  ➥ Vereinfachung der Implementierung

- **Unterstützung der Join-Operation sowie bestimmter statistischer Anfragen**

- **Einfache Überprüfung der referentiellen Integrität sowie weiterer Integritätsbedingungen**

  (z. B. Kardinalitätsrestriktionen)

- **Erhöhung der Anzahl der Blattseiten**

  ➥ mehr Seitenzugriffe beim sequentiellen Lesen aller Sätze eines Satztyps in Sortierordnung

- **Höhe des Baumes bleibt meist erhalten**

  ➥ ähnliches Leistungsverhalten beim Aufsuchen von Daten und beim Änderungsdienst

Physischer DB-Entwurf

- **Physischer DB-Entwurf**
  - Sorgfältige Wahl der wichtigen Attribute (Spalten), nach deren Werte Cluster-Bildung vorgenommen wird
  - Oft werden die Primärschlüsselattribute oder OWNER-MEMBER-Beziehungen für die Cluster-Bildung ausgewählt
  - Geeignete Wahl des Belegungsgrads beim Laden von Tabellen mit Cluster-Eigenschaft
  - Welche Attribute sollen indexiert werden?

- **Indexierungsheuristik**

  Indexstrukturen werden angelegt
  - auf allen Primär- und Fremdschlüsselattributen, was auch mit verallgemeinerten Zugriffspfadstrukturen erreicht werden kann
  - auf Attributen vom Typ DATE
  - auf Attributen, die in (häufigen) Anfragen in Gleichheits- oder IN-Prädikaten vorkommen

  ➥ Primär- und Fremdschlüsselindexierung erlaubt Navigation durch mehrere Tabellen ausschließlich mit Hilfe der Indexstrukturen

- **Alternative Indexierungsheuristik**
  - Indexstrukturen werden auf Primärschlüssel- und (möglicherweise) auf Fremdschlüsselattributen angelegt
  - Zusätzliche Indexstrukturen werden nur angelegt, wenn für eine aktuelle Anfrage der neue Index zehnmal weniger Sätze liefert als irgendein existierender Index

  ➥ Beide Heuristiken liefern fast die gleichen Indexstrukturen für die meisten Datenbanken und Arbeitslasten

Zugriffspfade in kommerziellen Datenbanksystemen

DB2 (IBM)	B*-Baum (clustered, non-clustered), partitionierte Tabellen, ...
Informix	B-Baum, statisches Hashing, ISAM, HEAP, ...
Oracle	B*-Baum (mit Präfix-/Suffix-Komprimierung), (Join-) Cluster-Bildung, ...
Sybase	B*-Baum (clustered, non-clustered), ...
RDB (DEC)	B*-Baum (clustered, non-clustered), Hashing, Join-Cluster-Bildung, ...
NonStop SQL (Tandem)	B*-Baum (clustered, non-clustered) mit Präfix-Komprimierung, ...
UDS (Siemens)	B*-Baum, statisches Hashing, Cluster-Bildung (LIST), Invertierung (Pointer-Array), Kettung (CHAIN)

Zusammenfassung

- **Cluster-Bildung optimiert (sortiert) sequentielle Zugriffe**
- **Standard-Zugriffspfadstruktur: B*-Baum** (the ubiquitous B*-tree)
  - materialisierte und referenzierte Speicherung der Datensätze
  - Index-organisierte Tabelle mit Cluster-Bildung
- **Schnellerer Schlüsselzugriff erfordert Hash-Verfahren**
  - (nur) direkter Zugriff (= 1 Seitenzugriff)
  - Erweiterbares Hashing unterstützt stark wachsende Datenbestände ($\leq$ 2 Seitenzugriffe)
- **Zugriffspfade für Sekundärschlüssel**
  - Einstiegsstruktur: B*-Baum u.a.
  - Verknüpfungsstruktur: Zeigerlisten, Bitlisten
  - Bitlisten sind bei geringer Kardinalität des Wertebereichs hoch effizient
  - Viele Komprimierungsverfahren verfügbar

  ➥ Unterstützung mengentheoretischer Operationen
- **Hierarchische Zugriffspfade**
  - Unterstützung von Verbund-Operationen (Relationenmodell)
  - effiziente Abwicklung von Set-Operationen (Netzwerk-Modell)
  - Verknüpfungsstruktur: Ketten, Zeigerlisten, Listen
  - vielfältige Abstimmungsmöglichkeiten auf spezielle Arbeitslasten
- **Verallgemeinerte Zugriffspfadstruktur**
  - auch Join-Index genannt
  - Unterstützung von Primärschlüssel-, Sekundärschlüssel- und hierarchischen Zugriffen

Komprimierung von Bitlisten

• Laufkomprimierung

Ein „Lauf“ oder ein „Run“ ist eine Bitfolge gleichartig gesetzter Bits. Bei der Laufkomprimierung wird die unkomprimierte Bitliste in aneinanderhängende, alternierende Null- und Einsfolgen aufgeteilt. Die Komprimierungstechnik besteht nun darin, jeden „Lauf“ durch seine Länge in einer Codierfolge darzustellen. Eine Codierfolge kann sich aus mehreren Codiereinheiten fester Länge (k Bits) zusammensetzen. Aus implementierungstechnischen Gründen wird k meist als ein Vielfaches der Bytelänge 8 gewählt. Falls eine Lauflänge größer als (2^k-1) Bits ist, wird zu ihrer Abbildung als Codierfolge mehr als eine Codiereinheit benötigt. Dabei gilt allgemein für die Komprimierung einer Bitfolge der Länge L mit

$$(n-1) * (2^k-1) < L \leq n * (2^k-1), \qquad n = 1, 2, \ldots$$

daß n Codiereinheiten erforderlich sind, wobei die ersten (n-1) Codiereinheiten voll mit Nullen belegt sind. Durch dieses Merkmal kann die Zugehörigkeit aufeinanderfolgender Codiereinheiten zu einer Codierfolge erkannt werden. Die Überprüfung jeder Codiereinheit auf vollständige Nullbelegung ist bei der Dekomprimierung zwar aufwendiger; die Einführung dieser impliziten Kennzeichnung der Fortsetzung einer Folge verhindert aber, daß das Verfahren bei Folgen der Länge $> 2^k$ scheitert.

Ein Beispiel soll diese Technik verdeutlichen (k=6):

Lauflänge	Codierung
1	000001
2	000010
63	111111
64	000000 000001
65	000000 000010

• Nullfolgenkomprimierung

Eine Nullfolge ist eine Folge von 0-Bits zwischen zwei 1-Bits in der unkomprimierten Bitliste. Der Grundgedanke dieser Technik ist es, die Bitliste nur durch aufeinanderfolgende Nullfolgen darzustellen, wobei jeweils ein 1-Bit implizit ausgedrückt wird. Da jetzt auch die Länge L=0 einer Nullfolge auftreten kann, ist folgende Codierung zu wählen (k=6), was einer Addition von Binärzahlen ≤ 2k-1 entspricht:

Komprimierung von Bitlisten (2)

- **Nullfolgenkomprimierung (Forts.)**

Nullfolgenlänge	Codierung
0	000000
1	000001
62	111110
63	111111 000000
64	111111 000001

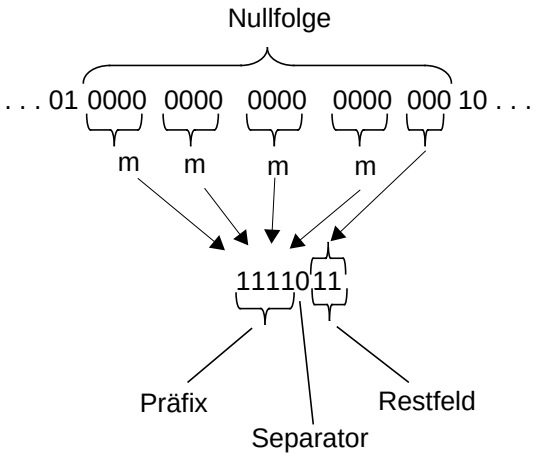
Durch die Möglichkeit, eine Codierfolge wiederum additiv durch mehrere Codiereinheiten zusammenzusetzen, lassen sich beliebig lange Nullfolgen darstellen. Es sind n Codiereinheiten erforderlich, wenn

$$(n-1) * (2^k - 1) \leq L < n * (2^k - 1), \qquad n = 1, 2, \ldots$$

gilt.

- **Golomb-Codierung**

Eine andere Art der Nullfolgenkomprimierung durch variabel lange Codierfolgen ist unter dem Namen Golomb-Code bekannt. Eine Nullfolge der Länge L wird durch eine Codierfolge bestehend aus einem variabel langen Präfix, einem Separatorbit und einem Restfeld fester Länge mit $\lceil \log_2 m \rceil$ Bit dargestellt. Der Präfix setzt sich aus $\lfloor L/m \rfloor$ 1-Bits gefolgt von einem 0-Bit als Separator zusammen. Das Restfeld beschreibt als Binärzahl die Anzahl der restlich 0-Bits der Nullfolge: $L - m*\lfloor L/m \rfloor$. Dieses Verfahren besitzt den Vorteil, unabhängig von der Wahl der Parameter die Komprimierung beliebig langer Nullfolgen zu gestatten. Wenn p die 0-Bit-Wahrscheinlichkeit in der Bitliste ist, sollte der Parameter m so gewählt werden, daß $p^m \approx 0.5$ ist. Die Komprimierung wird bei diesem Verfahren folgendermaßen durchgeführt (m=4):



Komprimierung von Bitlisten (3)

• Mehr-Modus-Komprimierung

Eine weitere Möglichkeit, Bitlisten zu verdichten, besteht darin, ein oder zwei Bits der Codierfolge fester Länge k als sogenannte Kennbits zu reservieren, um verschiedene Modi der Codierfolge zu kennzeichnen. Mit einem Kennbit lassen sich folgende zwei Modi unterscheiden:

1: k-1 Bits der Folge werden als „Bitmuster“ übernommen;

0: $\leq 2^{k-1}$-1 Bits werden als Nullfolge durch eine Binärzahl ausgedrückt.

Bei dieser Art der Codierung erweist es sich als nachteilig, daß wegen der Beschränkung von k eine Nullfolge oft durch mehrere aufeinanderfolgende Codierfolgen komprimiert werden muß. Weiterhin ist für freistehende Einsen in der Bitliste eine Codierfolge zu „opfern“, um sie als Bitmuster ausdrücken zu können. Durch Reservierung eines weiteren Kennbits sind diese Defekte zu beheben. Es besteht dann zusätzlich die Möglichkeit, lange Einsfolgen als Binärzahl verdichtet zu speichern. Mit zwei Kennbits lassen sich beispielsweise folgende vier Modi unterscheiden:

11: k-2 Bits der Folge werden als Bitmuster übernommen;

10: $\leq 2^{k-2}$ -1 Bits werden als Einsfolge durch eine Binärzahl codiert;

01: $\leq 2^{k-2}$ -1 Bits werden als Nullfolge durch eine Binärzahl codiert;

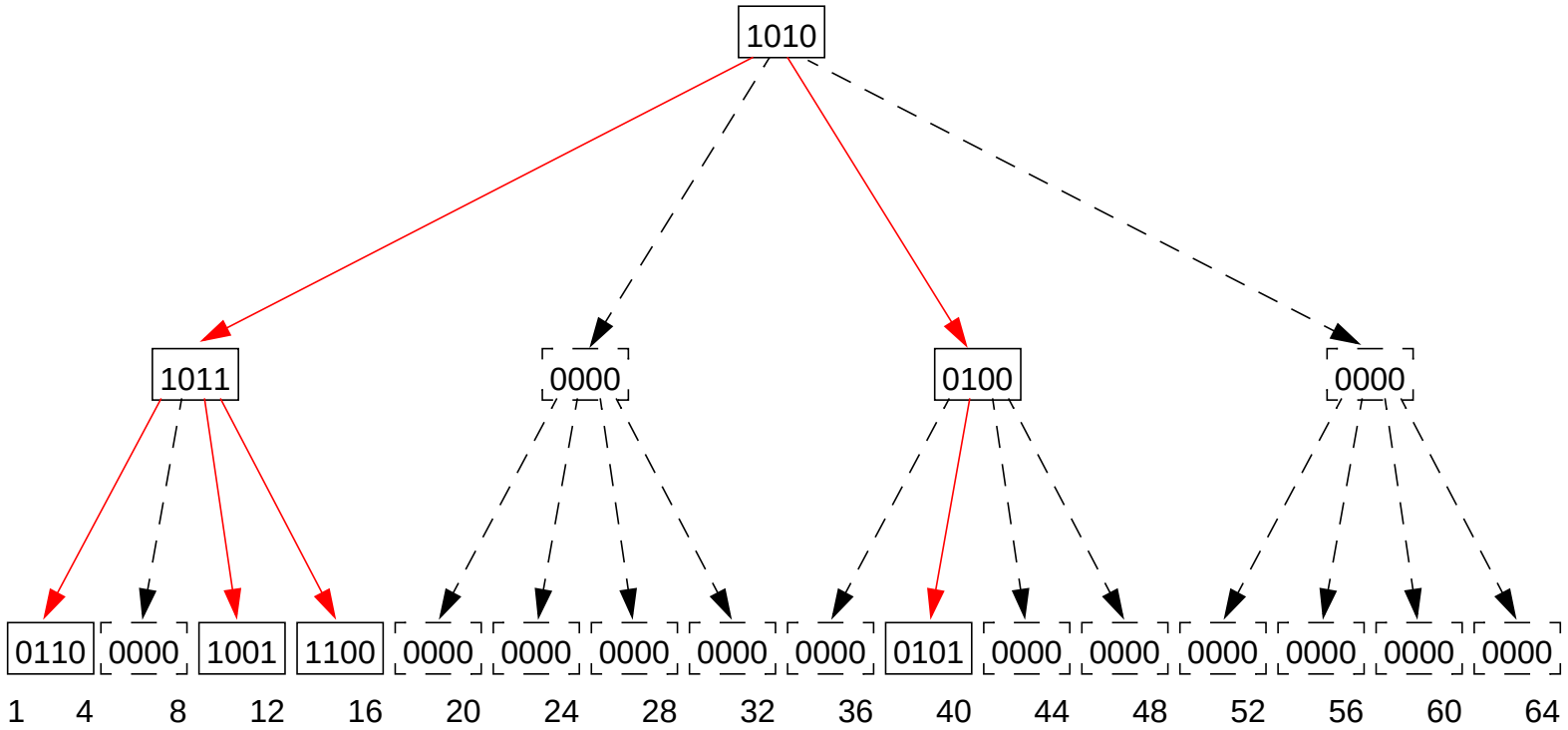
00: $\leq 2^{2k-2}$ -1 Bits werden in einer verdoppelten Codierfolge als Nullfolge durch eine Binärzahl ausgedrückt.

• Blockkomprimierung

Bei dieser Klasse von Verfahren wird die unkomprimierte Bitliste in Blöcke der Länge k eingeteilt. Eine erste Technik besteht darin, die einzelnen Blöcke durch einen Code variabler Länge zu ersetzen. Wenn die Wahrscheinlichkeit des Auftretens von Markierungen bekannt ist oder abgeschätzt werden kann, läßt sich ein sogenannter Huffman-Code anwenden. Bei einer Blocklänge k gibt es 2k verschiedene Belegungen, für die in Abhängigkeit der Wahrscheinlichkeit ihres Auftretens 2k Codeworte variabler Länge zu konstruieren sind. Komprimierung und Dekomprimierung sind bei dieser Technik recht aufwendig, da mit Hilfe einer Umsetztabelle jeder Block durch sein Codewort und umgekehrt zu ersetzen ist.

Bei einer zweiten Technik werden nur solche Blöcke gespeichert, in denen ein oder mehrere Bits gesetzt sind. Zur Kennzeichnung der weggelassenen Blöcke wird eine zweite Bitliste als Directory verwaltet, in der jede Markierung einem gespeicherten Block entspricht. Da im Directory wiederum lange Nullfolgen auftreten können, läßt es sich vorteilhaft mit Methoden der Nullfolgen- oder Mehr-Modus-Komprimierung verdichten.

Die Idee, auf das Directory wiederum eine Blockkomprimierung anzuwenden, führt auf die hierarchische Blockkomprimierung. Sie läßt sich rekursiv solange fortsetzen, bis sich die Eliminierung von Nullfolgen nicht mehr lohnt. Ausgehend von der obersten Hierarchiestufe kann die unkomprimierte Bitliste (Indextiefe d) leicht rekonstruiert werden.



Beispiel

- Knotenlänge l = 4 und Indextiefe d = 3
- indexierte Menge S = {2, 3, 9, 12, 13, 14, 38, 40}
- physische Speicherung